Engineering Method - FIBA players application

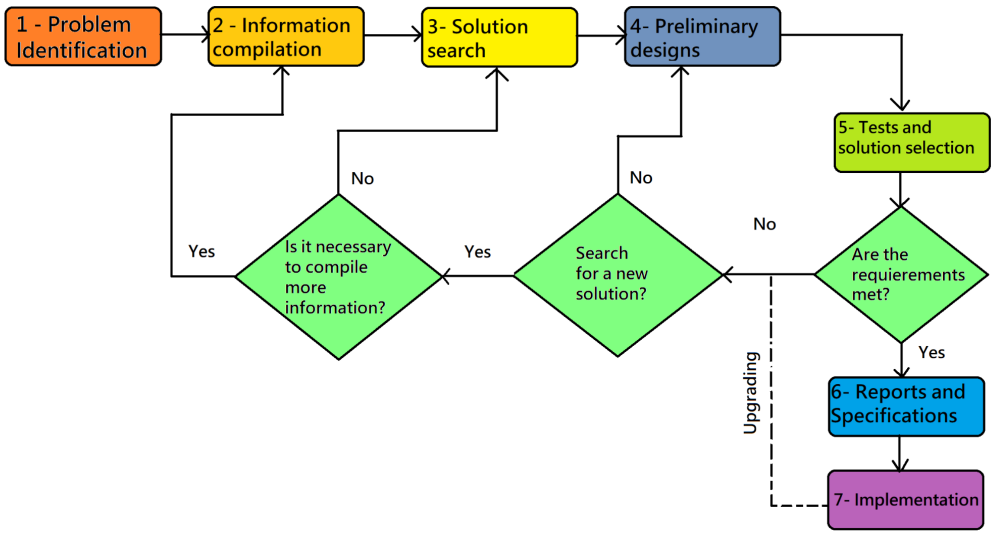
Problem Context[1]:

The International Basketball Federation, also known as FIBA, is the world governing body for basketball, the one that defines the rules of this sport at the international level and the body not only in charge of organizing and coordinating the most important orbital competitions but also of bringing together to all practitioners of this sport at a professional level. FIBA requires a first version of a software solution that handles a large volume of data: the most relevant data of each of the basketball professionals in the planet, so that different queries can be made that allow analysis of these data; the solution must also allow users to know: patterns about the development of the sport, the criteria that take more force or, in general, where the sport is currently heading.

Development of the Solution:

To resolve the above situation, the Engineering Method was chosen to develop the solution following a systematic approach and in line with the problematic situation established.

Based on the description of Paul Wright's “Introduction to Engineering”, the following flowchart was defined, the steps of which we will follow in the development of the solution.



Step 1. Problem Identification

[1] Fictional problem context.

The specific needs of the problematic situation are recognized as well as their symptoms and conditions under which it must be resolved.

Identification of needs and symptoms:

- The solution to the problem must be efficient so that it can make players queries as fast as possible according to desired criteria, with a time complexity of O(log n).
- The program must work with at least 200,000 valid data.

Functional requirements:

The program must be able to:

Name:	**FR1:** Enter player data.
Description:	The system must allow the submission of player data either in mass or through a graphic interface.
Inputs:	Name, age, team, and the 6 following statistical values: true shooting (%), usage (%), assist (%), rebound (%) and defensive (%) and blocks (%)
Output:	Player data entered.

Name:	**FR2**: Save the players' data.
Description:	The system must allow access to players' data previously submitted.
Inputs:	Nothing.
Output:	Players' data saved.

Name:	**FR3:** Modify player data.
Description:	The system must allow the modification of player data either in mass or through a graphic interface.

Inputs:	Player ID, data to modify, and new value of the data.
Output:	Message about the modification of the player data.

Name:	**FR4:** Delete player data.
Description:	The system must allow the deletion of player data either in mass or through a graphic interface.
Inputs:	ID of the player to be deleted.
Output:	Message about the deletion of the player data.

Name:	**FR5:** Search a player by statistical attributes.
Description:	The system must allow the search of a player through a statistical attribute.
Inputs:	Statistical attribute of the player to search.
Output:	Player(s) searched.

Name:	**FR6:** Display search time.
Description:	The system shows the user how long it takes to perform a search (FR5).
Inputs:	Nothing
Output:	Time taken is displayed.

Problem Definition:

FIBA requires the development of a software application that allows users to make queries of basketball players.

Step 2. Information Collection

In order to have complete clarity in the concepts involved, a search is made for the definitions of the terms most closely related to the problem established. It is important to perform this search with recognized and reliable sources to know which elements are part of the problem and which are not.

Sources:

Introduction to Algorithms. Cormen et al. Chapter 3. Growth of Functions.
https://www.basketball-reference.com/about/glossary.html#mp
https://thejumpball.net/2018/01/02/how-the-nba-is-getting-assists-wrong/
https://www.sportslingo.com/sports-glossary/r/rebound/

Definitions:

- Algorithmic Analysis:

  Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to [1].

- Asymptotic notation:

  The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N '0, 1, 2, ...'. Such notations are convenient for describing the worst-case running-time function T (n), which usually is defined only on integer input sizes. Frequently, asymptotic notation is used to describe the running times of algorithms.

- Θ-notation:

  The **Θ**-notation asymptotically bounds a function from above and below. For a given function g(n), we denote by **Θ**(g(n)) the set of functions: **Θ**(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0$ }

- O-notation:

  The O-notation provides an asymptotic upper bound function. For a given function g(n), we denote by O(g(n)) the set of functions: O(g(n)) = {f(n): there exist positive constants c and n0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$ }

- Ω-notation:

  The Ω-notation provides an asymptotic lower bound function. For a given function g(n), we denote by O(g(n)) the set of functions: Ω(g(n)) = {f(n): there exist positive constants c and n0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0$ } [1]

- True shooting (TS%):

  True Shooting Percentage; the formula is PTS / (2 * TSA). True shooting percentage is a measure of shooting efficiency that takes into account field goals, 3-point field goals, and free throws.

- PTS: Points.

- TSA: True Shooting Attempts; the formula is FGA + 0.44 * FTA.

- FGA: Field Goal Attempts (includes both 2-point field goal attempts and 3-point field goal attempts).

- FTA: Free Throw Attempts.

- Usage percentage (Usg%): (available since the 1977-78 season in the NBA); the formula is 100 * ((FGA + 0.44 * FTA + TOV) * (Tm MP / 5)) / (MP * (Tm FGA + 0.44 * Tm FTA + Tm TOV)). Usage percentage is an estimate of the percentage of team plays used by a player while he was on the floor.

- TOV: Turnovers (available since the 1977-78 season in the NBA).

- Tm: Team.

- MP: Minutes Played (available since the 1951-52 season).

- Assist: An assist is a pass that directly leads to a basket. This can be a pass to the low post that leads to a direct score, a long pass for a layup,

a fast break pass to a teammate for a layup, and/or a pass that results in an open perimeter shot for a teammate. In basketball, an assist is awarded only if, in the judgment of the statistician, the last player's pass contributed directly to a made basket. An assist can be awarded for a basket scored after the ball has been dribbled if the player's pass led to the field goal being made

- Rebound: A rebound occurs in basketball when a player gains possession of the basketball after a missed field goal, three-point field goal or free throw attempt.

- Defensive: refers to how much a player prevents the opposing offense from scoring baskets.

- Blocks: occur when a defensive player legally deflects a field goal attempt from an offensive player to prevent a score.

Step 3. Finding Creative Solutions

For this step, even if we can think of our own solutions, we look at specialized texts various ways to implement the different abstract data structures/types (ADTs) necessary and that best suit the different cases of each stage of the process of making basketball players queries. The methods considered as alternatives are as follows:

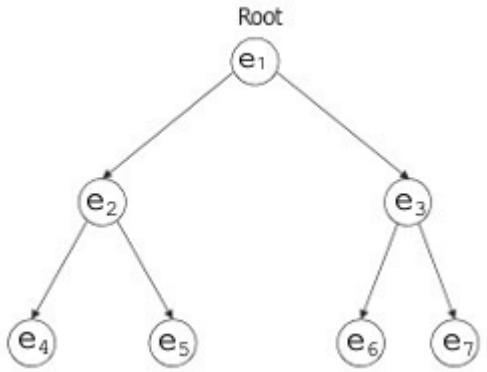
Alternative 1 - Use Binary Trees

Three types of binary trees would store all the player's data of the program. The AVL tree and the Standard Binary Search.

The ADT Binary Search Tree:

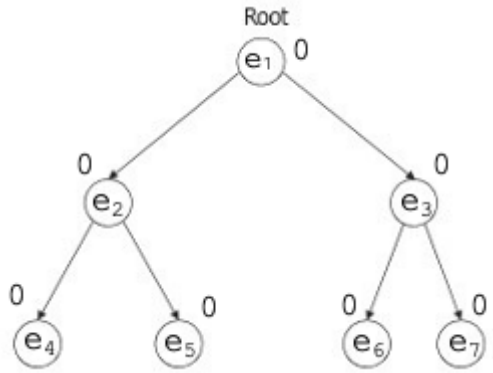
A binary search tree is a node-based binary tree data structure that quickly allows us to maintain a sorted list of items. Each node (item) has a Comparable key (and an associated value). This tree has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



The ADT AVL Tree:

AVL tree is a self-balancing Binary Search Tree (BST) in which each node is associated with a balance factor. The difference between heights of left and right subtrees cannot be more than one for all nodes. It's balanced if the balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.



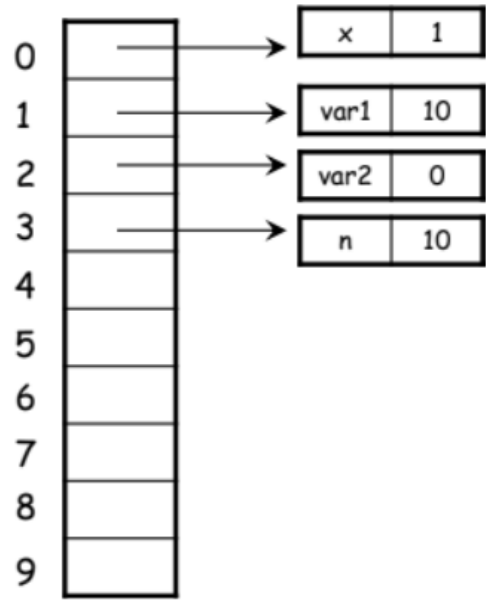
<u>Alternative 2 - Use Hash Tables:</u>

Hash tables would store all the player's data entered in the program, thanks to the chaining linked list method.

The ADT Hash Table:

The ADT hash table is a data structure that associates keys with values. The main operation that it efficiently supports is search:

- Allows access to stored items from a generated key.
- It works by transforming the key with a hash function into a hash, a number that the hash table uses to locate the desired value.



Access Procedures:

Constructor operations to construct a Hash Table:

i. createTable() → creates an empty table.
ii. tableInsert(newItem) → inserts newItem into a table in its proper sorted order according to the item's search key.

Predicate operations to test Hash Tables:

i. isEmpty( ) → determines whether a table is empty.

Selector operations to select items of a Hash Table:

i. tableLength( ) → determines the number of items in the table.
ii. tableDelete(searchKey) → deletes an item with a given search key from the table.
iii. tableRetrieve(searchKey) → retrieves an item with a given search key from a table.

Alternative 3 - Use Lists:

Lists would store all the player's data entered in the program.

The ADT List:

The ADT List is a linear sequence of an arbitrary number of items that must be of the same type, together with the following access procedures that can be grouped into three broad categories:

Operations to construct Arrays constructor:

i. createList( ) → creates an empty list.

ii. add(index, item) → inserts item at position index of a list if 1 ≤ index ≤ size() + 1. If index ≤ size(), items at position index onwards are shifted one position to the right.

Preach operations to test Arrays:

i. isEmpty( ) → determine if an array is empty.

Selector operations to select items of a Array:

i. remove(index) → removes item at position index of a list if 1 ≤ index ≤ size(). Items at position index + 1 onwards are shifted one position to the left.

ii. get(index) → returns item at position index of a list if 1 ≤ index ≤ size().

iii. set(index, newValue) → sets the value of an specified index to a new value.

iv. size( ) → returns number of items in a list.

Step 4. Transitioning Ideas to Preliminary Designs

The first thing we do in this step is to dismiss ideas that are not feasible. In this sense, we ruled *out Alternative 3 - Use Lists* because the implementation of these linear structures for this problem increases the time complexity to its worst case, O(n), for the players' insertion, deletion and searching processes. Thus, efficiency would be ruined.

Careful review of the other alternatives leads to the following:

Alternative 1 - Use Binary Trees:

➢ These three trees would allow the player's data storage in optimal time complexity conditions, O(log n), even if the expected data to store is very huge. The AVL tree would allow an optimized searching proces and Standard Binary Search Tree would help to justify the usage of the two previously mentioned trees in terms of time complexity, mostly when searching players.

Alternative 2 - Use Hash Tables:

➢ Hash Tables would allow the player's data storage, but due to the immense quantity of expected data to store (200.000 basketball players), a lot of collisions would occur, making the searching, insertion and deletion processes quite slow, with a time complexity equals to O(n) in the linked-lists.

Step 5. Evaluation and Selection of the Best Solution

<u>Criteria</u>

The criteria for evaluating solution alternatives should be defined and based on this outcome, the solution that best meets the needs of the problem stated is chosen. The criteria we chose in this case are the ones listed below. A numerical value has been established next to each one in order to establish a weight indicating which of the possible values of each criterion have the most weight (i.e., they are most desirable).

Criterion A. Worst case in time complexity for the context problem. The most efficient solution would be prefered. Efficiency can be:

- [1] O(n!)
- [2] O(2))
- [3] O(n-2)
- [4] O(n log n)
- [5] O(n)
- [6] O(log n), O(1)

Criterion B. Data is stored following an specific order rule:

- [3] Follows an order rule
- [2] Follows partially an order rule
- [1] Does not follows an order rule

Evaluation

By evaluating the above criteria in the alternatives that are maintained, we get the following table:

	Criterion A	**Criterion B**	**Total**
Alternative 1	6 O(log n)	3 Follows an order rule	9
Alternative 2	5 O(n)	1 Does not follows an order rule	6

Selection

According to the previous evaluation Alternative 1 should be selected, as it obtained the highest score according to the defined criteria.

Step 6. Preparation of Reports and Specifications

Problem Specification:

Problem: Make queries of basketball player's data efficiently in a "database" of

200.000 players from the FIBA.

Input: 200.000 valid player's data extrapolated from a text file, that represents this "database" to ensure persistence. Then, the specification of a statistical attribute to make the query around a specified interval.

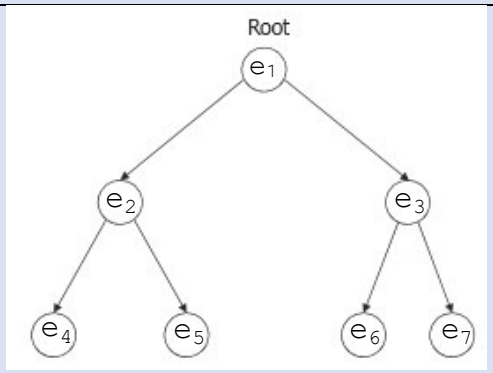
Output: The search results of all players that coincided with the mentioned statistical attribute in the specified interval.

Considerations

The following cases should be taken into account for testing this process:

- The time to create all the binary trees to insert into them all the 200.000 players can take a significant time.
- The longer the search interval is, the longer it will take to search for players with a specified statistical attribute.

ADT Generic Binary Search Tree



$BSTree = \{ p_1(l_1(k_1 ; v_1)r_1), p_2(l_2(k_2 ; v_2)r_2), \dots , p_n(l_n(k_n ; v_n)r_n) \}$
k_i = key of the element.
v_i = associated value to k_i
$(k_i ; v_i) = e_i$ = tree element $\in BSTree$
Root = $(k_1 ; v_1) \in BSTree$
l_i = left child of $(k_i ; v_i) \in BSTree$
r_i = right child of $(k_i ; v_i) \in BSTree$
p_i = parent of $(k_i ; v_i) \in BSTree$
$T(k_i)$ = type of k_i, comparable with other elements' keys.
$T(v_i)$ = type of v_i
Size = $|BSTree|$
Level = number of elements between an element until deepest element.
TreeHeight = tree Levels
ElementHeight = element levels

{ inv: $\forall\ k_i, k_j \in BSTree \implies (T(k_i) = T(k_j) \mid i \neq j)$

$\forall\ v_i, v_j \in BSTree \implies (T(v_i) = T(v_j) \mid i \neq j)$

Root = 1° element $\in BSTree$

$\exists!$ Root $\in BSTree$

Size ≥ 0

Size = 0 $\implies \nexists$ Root $\wedge \nexists$ Level $\wedge$ TreeHeight = 0 $\wedge$ $BSTree = \emptyset$

Size = 1 $\implies \exists$ Root $\wedge$ Height = 1 $\wedge$ Level = 0 $\wedge$ $BSTree \neq \emptyset$

TreeHeight = 1 $\implies \nexists$ Root.left $\wedge \nexists$ Root.right $\wedge \nexists$ Root.parent

TreeHeight $\geq 2 \implies (\exists\ e_i \in$ Level < MaxLevel $\implies \exists\ e_i.l_i \wedge/\vee \exists\ e_i.r_i)$

$(\nexists\ e_i.l_i \wedge \nexists\ e_i.r_i) \implies e_i$ = leaf element

$\forall\ e_i \neq$ Root $\implies \exists\ e_i.p_i$

$\forall\ k_i(e_i) \implies e_i.l_i.k_i < e_i.k_i < e_i.r_i.k_i$

(Insertion ∧ Deletion ∧ Searching) ⟹ Worst case = O(n) }

Primitive operations				Operation type
▪ CreateTree		→	BSTree	Constructor
▪ TreeInsert	Element(Key x Value) x BSTree	→	BSTree	Modifier
▪ TreeSearch	Key x BSTree	→	Value T	Analyzer
▪ TreeDelete	Key x BSTree	→	BSTree	Modifier
▪ PreOrder	BSTree	→	String	Analyzer
▪ InOrder	BSTree	→	String	Analyzer
▪ PostOrder	BSTree	→	String	Analyzer

CreateTree()

"Creates an empty Binary Search Tree"

{ pre: *TRUE* }

{ post: BSTree = ∅ }

TreeInsert(element(key, value), BSTree)

"Inserts a new element with key and value in the Binary Search Tree according to the key"

{ pre: ∃ BSTree, element(key, value) ∉ BSTree }

{ post: |BSTree| + 1, element(key, value) ∈ BSTree }

TreeSearch(key, BSTree)

"Searches and retrieves the value of an element with a given search key from the Binary Search Tree"

{ pre: ∃ BSTree, BSTree ≠ ∅, element(key, value) ∈ BSTree }

{ post: value ∈ element) }

TreeDelete(key, BSTree)

"Deletes an element with a given search key from the Binary Search Tree"

{ pre: ∃ BSTree, BSTree ≠ ∅, element(key, value) ∈ BSTree }

{ post: |BSTree| - 1, element(key, value) ∉ BSTree }

PreOrder(BSTree)

"Returns a text with the values of the elements of the Binary Search Tree, traversing it from the root, then by the left subtree, then the right subtree"

{ pre: ∃ BSTree, BSTree ≠ Ø }

{ post: String ∈ Text }

InOrder(BSTree)

"Returns a text with the values of the elements of the Binary Search Tree, traversing it from the left subtree, then by the root, then the right subtree"

{ pre: ∃ BSTree, BSTree ≠ Ø }

{ post: String ∈ Text }

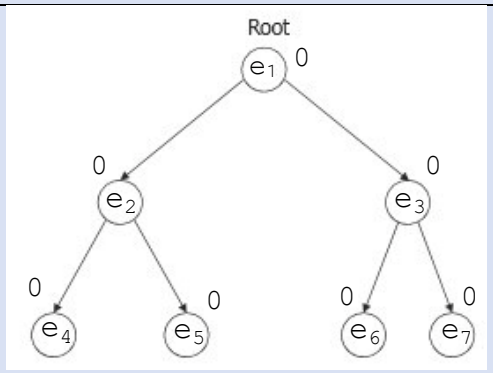
PostOrder(BSTree)

"Returns a text with the values of the elements of the Binary Search Tree, traversing it from the left subtree, then the right subtree, then by the root"

{ pre: ∃ BSTree, BSTree ≠ Ø }

{ post: String ∈ Text }

ADT Generic AVL Tree



$AVLTree = \{ p_1(l_1(k_1 ; v_1)r_1)bf_1, p_2(l_2(k_2 ; v_2)r_2)bf_2, \dots , p_n(l_n(k_n ; v_n)r_n)bf_n \}$
k_i = key of the element
v_i = associated value to k_i
$(k_i ; v_i) = e_i$ = tree element $\in AVLTree$
$Root = (k_1 ; v_1) \in AVLTree$
l_i = left child of $(k_i ; v_i) \in AVLTree$
r_i = right child of $(k_i ; v_i) \in AVLTree$
p_i = parent of $(k_i ; v_i) \in AVLTree$
$bf_i(e_i)$ = element balance factor = height(right subtree) - height(left subtree)
$T(k_i)$ = type of k_i, comparable with others keys.
$T(v_i)$ = type of v_i
$Size = |AVLTree|$
Level = number of elements between an element until deepest element.
TreeHeight = tree Levels
ElementHeight = element levels

$\{$ inv: $\forall\ k_i, k_j \in AVLTree \implies (T(k_i) = T(k_j) \mid i \neq j)$

$\forall\ v_i, v_j \in AVLTree \implies (T(v_i) = T(v_j) \mid i \neq j)$

$Root = 1° \text{ element} \in AVLTree$

$\exists!\ Root \in AVLTree$

$Size \geq 0$

$Size = 0 \implies \nexists\ Root \wedge \nexists\ Level \wedge TreeHeight = 0 \wedge AVLTree = \emptyset$

$Size = 1 \implies \exists\ Root \wedge Height = 1 \wedge Level = 0 \wedge AVLTree \neq \emptyset$

$TreeHeight = 1 \implies \nexists\ Root.left \wedge \nexists\ Root.right \wedge \nexists\ Root.parent$

$TreeHeight \geq 2 \implies (\exists\ e_i \in Level < MaxLevel \implies \exists\ e_i.l_i \ \veebar \ \exists\ e_i.r_i)$

$(\nexists\ e_i.l_i \wedge \nexists\ e_i.r_i) \implies e_i = \text{leaf element}$

$\forall\ e_i \neq Root \implies \exists\ e_i.p_i$

$\forall\ k_i(e_i) \implies e_i.l_i.k_i < e_i.k_i < e_i.r_i.k_i$

$\forall\ e_i \in AVLTree \Longrightarrow (|e_i.bf_i| \leq 1 \Longrightarrow$ balanced tree, else unbalanced tree)

For LeftRotate(e_i) $\Longrightarrow \exists\ e_i.r_i$

For RightRotate(e_i) $\Longrightarrow \exists\ e_i.l_i$

(Insertion $\wedge$ Deletion $\wedge$ Searching) $\Longrightarrow$ Worst case = O(log n)}

Primitive operations				Operation type
▪ CreateTree		→	AVLTree	Constructor
▪ TreeInsert	Element(Key x Value) x AVLTree	→	AVLTree	Modifier
▪ TreeSearch	Key x AVLTree	→	Value T	Analyzer
▪ TreeDelete	Key x AVLTree	→	AVLTree	Modifier
▪ LeftRotate	Element(Key x Value) x AVLTree	→	AVLTree	Modifier
▪ RightRotate	Element(Key x Value) x AVLTree	→	AVLTree	Modifier
▪ PreOrder	AVLTree	→	String	Analyzer
▪ InOrder	AVLTree	→	String	Analyzer
▪ PostOrder	AVLTree	→	String	Analyzer

CreateTree()

"Creates an empty AVL Tree"

{ pre: *TRUE* }

{ post: AVLTree = ∅ }

TreeInsert(element(key, value), AVLTree)

"Inserts a new element with key and value in the AVL Tree according to the key"

{ pre: ∃ AVLTree, element(key, value) ∉ AVLTree }

{ post: |AVLTree| + 1, element(key, value) ∈ AVLTree, ∃ bf(element) }

TreeSearch(key, BSTree)

"Searches and retrieves the value of an element with a given search key from the AVL Tree"

{ pre: ∃ AVLTree, AVLTree ≠ ∅, element(key, value) ∈ AVLTree }

{ post: value ∈ element) }

TreeDelete(key, AVLTree)

"Deletes an element with a given search key from the AVL Tree"

{ pre: ∃ AVLTree, AVLTree ≠ ∅, element(key, value) ∈ AVLTree }

{ post: |AVLTree| - 1, element(key, value) ∉ AVLTree }

LeftRotate(element(key, value), AVLTree)

"Rotates to the left an element of the AVL Tree, transferring a little of the weight (height) from the right sub-tree to the left sub-tree"

{ pre: ∃ AVLTree, AVLTree ≠ ∅, element(key, value) ∈ AVLTree, |bf(element)| > 1 }

{ post: AVLTree, |bf(element)| ≤ 1 }

RightRotate(element(key, value), AVLTree)

"Rotates to the right an element of the AVL Tree, transferring a little of the weight (height) from the left sub-tree to the right sub-tree"

{ pre: ∃ AVLTree, AVLTree ≠ ∅, element(key, value) ∈ AVLTree, |bf(element)| > 1 }

{ post: AVLTree, |bf(element)| ≤ 1 }

PreOrder(AVLTree)

"Returns a text with the values of the elements of the AVL Tree, traversing it from the root, then by the left subtree, then the right subtree"

{ pre: ∃ AVLTree, AVLTree ≠ ∅ }

{ post: String ∈ Text }

InOrder(AVLTree)

"Returns a text with the values of the elements of the AVL Tree, traversing it from the left subtree, then by the root, then the right subtree"

{ pre: ∃ AVLTree, AVLTree ≠ ∅ }

{ post: String ∈ Text }

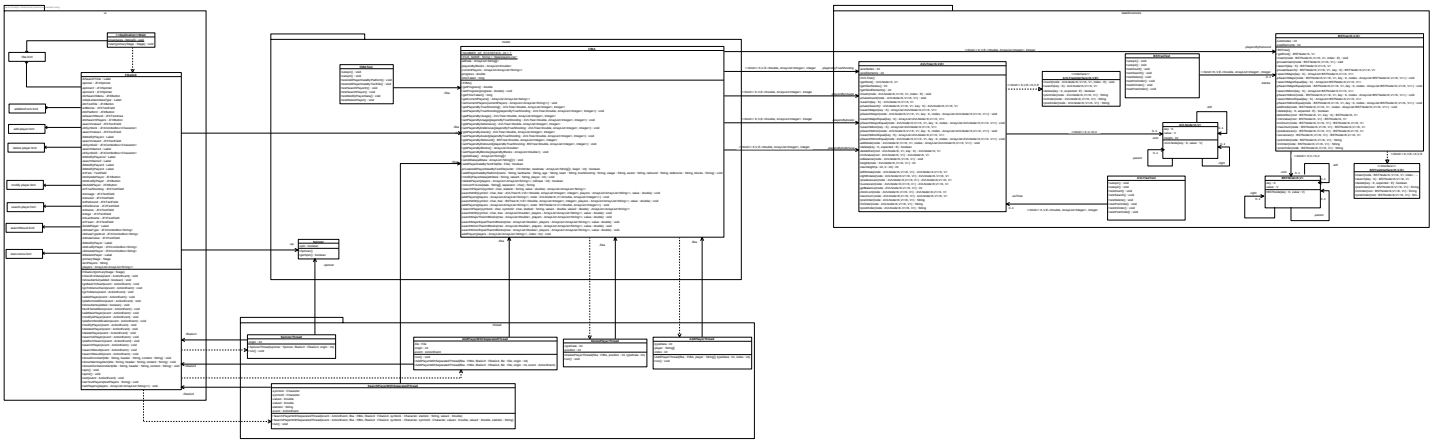
PostOrder(AVLTree)

"Returns a text with the values of the elements of the AVL Tree, traversing it from the left subtree, then the right subtree, then by the root"

{ pre: ∃ AVLTree, AVLTree ≠ ∅ }

{ post: String ∈ Text }

Class Diagram



Design of the scenarios and test cases

Scenario configuration

Name	Class	Scenario
setup1	BSTreeTest	bstree, a BSTree of types <Double,ArrayList<Integer>,Integer> is initialized as BSTree<>()
setup2	BSTreeTest	bstree, a BSTree of types <Double,ArrayList<Integer>,Integer> is initialized as BSTree<>(). list, an ArrayList<Integer> is initialized as ArrayList<Integer>(). 0 is added to list. node, a BSTNode<Double,ArrayList<Integer>> is initialized as BSTNode<>(80.0,list). node is inserted in bstree with index 0. list is reinitialized as ArrayList<Integer>(). 1 is added to list. node is reinitialized as BSTNode<>(88.8,list). node is inserted in bstree with index 1. list is reinitialized as ArrayList<Integer>(). 2 is added to list. node is reinitialized as BSTNode<>(39.1,list). node is inserted in bstree with index 2. list is reinitialized as ArrayList<Integer>(). 3 is added to list. node is reinitialized as BSTNode<>(45.0,list). node is inserted in bstree with index 3. list is reinitialized as ArrayList<Integer>(). 4 is added to list. node is reinitialized as BSTNode<>(35.5,list). node is inserted in bstree with index 4. list is reinitialized as ArrayList<Integer>(). 5 is added to list. node is reinitialized as BSTNode<>(83.7,list). node is inserted in bstree with index 5. list is reinitialized as ArrayList<Integer>(). 6 is added to list. node is reinitialized as BSTNode<>(90.4,list). node is inserted in bstree with index 6.
setup1	AVLTreeTest	avlTree, an AVLTree of types <Double,ArrayList<Integer>,Integer> is initialized as AVLTree<>()
setup2	AVLTreeTest	avlTree, an AVLTree of types <Double,ArrayList<Integer>,Integer> is initialized as AVLTree<>(). list, an ArrayList<Integer> is initialized as ArrayList<Integer>().

		0 is added to list. node, an AVLNode<Double,ArrayList<Integer>> is initialized as AVLNode<>(80.0,list). node is inserted in avlTree with index 0. list is reinitialized as ArrayList<Integer>(). 1 is added to list. node is reinitialized as AVLNode<>(88.8,list). node is inserted in avlTree with index 1. list is reinitialized as ArrayList<Integer>(). 2 is added to list. node is reinitialized as AVLNode<>(39.1,list). node is inserted in avlTree with index 2. list is reinitialized as ArrayList<Integer>(). 3 is added to list. node is reinitialized as AVLNode<>(45.0,list). node is inserted in avlTree with index 3. list is reinitialized as ArrayList<Integer>(). 4 is added to list. node is reinitialized as AVLNode<>(35.5,list). node is inserted in avlTree with index 4. list is reinitialized as ArrayList<Integer>(). 5 is added to list. node is reinitialized as AVLNode<>(83.7,list). node is inserted in avlTree with index 5. list is reinitialized as ArrayList<Integer>(). 6 is added to list. node is reinitialized as AVLNode<>(90.4,list). node is inserted in avlTree with index 6.
setup1	FIBATest	fiba is initialized as FIBA(), all of fiba's attributes and relations are initialized with their empty constructors.
setup2	FIBATest	fiba is initialized as FIBA(), all of fiba's attributes and relations are initialized with their empty constructors. Information is loaded from file "resources/100.csv".

Test cases design

Test Objective: Verify that a new item is successfully inserted in the BSTree, in this case it is a BSTree<Double, ArrayList<Integer>, Integer>				
Class	**Method**	**Scenario**	**Input values**	**Result**
BSTree<Double, ArrayList<Integer>, Integer>	insert	setup1	a node with key=11.1 and value={0,}.	The node is successfully inserted in the tree, the tree's root is no longer null, searching for a node with key=11.1 does not return null but the new node's parent, left and right are still null.
BSTree<Double, ArrayList<Integer	insert	setup2	a node with	There are 7 nodes and 7 values. Then, the node is successfully

>, Integer>			key=30.0 and value={4,}.	inserted in the tree: There are now 8 nodes, the new node's left and right are null but its parent is not, the new node's parent's key is 35.5 and the new node is its parent's left child. The key of the tree's root is 80.0
BSTree<Double, ArrayList<Integer>, Integer>	insert	setup2	a node with key=80.0 and value={4,}.	The new node is successfully inserted in the tree: The root now has two values, the tree has 7 nodes but stores 8 values.

Test Objective: Verify that an item can be found in the BSTree given its key, in this case it is a BSTree<Double, ArrayList<Integer>, Integer>				
Class	**Method**	**Scenario**	**Input values**	**Result**
BSTree<Double, ArrayList<Integer>, Integer>	search	setup1	11.1	Pre: The tree's root must be null. Null must be returned since the tree is empty.
	search	setup2	45.0	Pre: The tree's root is not null. Method does not return null.

Test Objective: Verify that an item can be deleted from the BSTree given its key and value, in this case it is a BSTree<Double, ArrayList<Integer>, Integer>				
Class	**Method**	**Scenario**	**Input values**	**Result**
BSTree<Double, ArrayList<Integer>, Integer>	delete	setup1	50.0,0	Pre: The root is null, there are no nodes or values stored in the tree.
BSTree<Double, ArrayList<Integer>, Integer>	delete	setup2	35.5,0	Pre: The tree's root is not null, its key is 80.0, there are 7 nodes and 7 values stored in the tree, there is a node with key 35.5. The method returns true, there is no longer a node with key 35.5, the tree has 6 nodes and stores 6 values.
BSTree<Double, ArrayList<Integer>, Integer>	delete	setup2	11.4,0 80,0	Pre: There are 7 nodes and 7 values stored in the tree, there is no node with key 11.4 Method returns false for 11.4 and true for 80.0, there is no longer a

				node with key 80.0, the tree has 6 nodes and stores 6 values.
BSTree<Double, ArrayList<Integer>, Integer>	delete	setup2	39.1,0	Pre: There are 7 nodes and 7 values stored in the tree, there is a node with key 39.1 Method returns true, there is no longer a node with key 39.1, the tree has 6 nodes and stores 6 values.
BSTree<Double, ArrayList<Integer>, Integer>	delete	setup2	80.0, 0	A new node is inserted with key 80.0 and values{0,} Pre: There are 7 nodes and 8 values stored in the tree, there is a node with key 80.0 Method returns true, there is a node with key 80.0 with only one value, there are 7 nodes and 7 values stored in the tree.

Test Objective: Verify that a node's predecessor can be found.				
Class	**Method**	**Scenario**	**Input values**	**Result**
BSTree<Double, ArrayList<Integer>, Integer>	predecessor	setup2	39.1 45.0 80.0 88.8	The predecessors found for the nodes with the given keys have the following keys respectively: 35.5, 39.1, 45.0, 83.7

Test Objective: Verify that a node's successor can be found.				
Class	**Method**	**Scenario**	**Input values**	**Result**
BSTree<Double, ArrayList<Integer>, Integer>	successor	setup2	39.1 45.0 80.0 88.8	The successors found for the nodes with the given keys have the following keys respectively: 45.0,80.0,83.7,90.4

Test Objective: Verify that the tree is correctly traversed in preorder				
Class	**Method**	**Scenario**	**Input values**	**Result**
BSTree<Double, ArrayList<Integer>, Integer>	preOrder	setup2	The tree root	The method must return "80.0 39.1 35.5 45.0 88.8 83.7 90.4 "

Test Objective: Verify that the tree is correctly traversed in inorder				
Class	**Method**	**Scenario**	**Input values**	**Result**
BSTree<Double, ArrayList<Integer>, Integer>	inOrder	setup2	The tree root	The method must return "35.5 39.1 45.0 80.0 83.7 88.8 90.4 "

Test Objective: Verify that the tree is correctly traversed in preorder				
Class	**Method**	**Scenario**	**Input values**	**Result**
BSTree<Double, ArrayList<Integer>, Integer>	preOrder	setup2	The tree root	The method must return "35.5 45.0 39.1 83.7 90.4 88.8 80.0 "

Test Objective: Verify that a new item is successfully inserted in the AVLTree, in this case it is an AVLTree<Double, ArrayList<Integer>, Integer>				
Class	**Method**	**Scenario**	**Input values**	**Result**
AVLTree<Double, ArrayList<Integer>, Integer>	insert	setup1	a node with key=11.1 and value={0,}.	The node is successfully inserted in the tree, the tree's root is no longer null, searching for a node with key=11.1 does not return null but the new node's parent, left and right are still null.
AVLTree<Double, ArrayList<Integer>, Integer>	insert	setup2	a node with key=30.0 and value={4,}.	There are 7 nodes and 7 values. Then, the node is successfully inserted in the tree: There are now 8 nodes, the new node's left and right are null but its parent is not, the new node's parent's key is 35.5 and the new node is its parent's left child. The key of the tree's root is 80.0
AVLTree<Double, ArrayList<Integer>, Integer>	insert	setup2	a node with key=80.0 and value={4,}.	The new node is successfully inserted in the tree: The root now has two values, the tree has 7 nodes but stores 8 values.

Test Objective: Verify that an item can be found in the AVLTree given its key, in this case

it is a AVLTree<Double, ArrayList<Integer>, Integer>				
Class	**Method**	**Scenario**	**Input values**	**Result**
AVLTree<Double, ArrayList<Integer>, Integer>	search	setup1	11.1	Pre: The tree's root must be null. Null must be returned since the tree is empty.
	search	setup2	45.0	Pre: The tree's root is not null. Method does not return null.

Test Objective: Verify that an item can be deleted from the AVLTree given its key and value, in this case it is a AVLTree<Double, ArrayList<Integer>, Integer>				
Class	**Method**	**Scenario**	**Input values**	**Result**
AVLTree<Double, ArrayList<Integer>, Integer>	delete	setup1	50.0,0	Pre: The root is null, there are no nodes or values stored in the tree.
AVLTree<Double, ArrayList<Integer>, Integer>	delete	setup2	35.5,0	Pre: The tree's root is not null, its key is 80.0, there are 7 nodes and 7 values stored in the tree, there is a node with key 35.5. The method returns true, there is no longer a node with key 35.5, the tree has 6 nodes and stores 6 values.
AVLTree<Double, ArrayList<Integer>, Integer>	delete	setup2	11.4,0 80,0	Pre: There are 7 nodes and 7 values stored in the tree, there is no node with key 11.4 Method returns false for 11.4 and true for 80.0, there is no longer a node with key 80.0, the tree has 6 nodes and stores 6 values.
AVLTree<Double, ArrayList<Integer>, Integer>	delete	setup2	39.1,0	Pre: There are 7 nodes and 7 values stored in the tree, there is a node with key 39.1 Method returns true, there is no longer a node with key 39.1, the tree has 6 nodes and stores 6 values.
AVLTree<Double, ArrayList<Integer>, Integer>	delete	setup2	80.0, 0	A new node is inserted with key 80.0 and values{0,} Pre: There are 7 nodes and 8 values stored in the tree, there is a node with key 80.0

				Method returns true, there is a node with key 80.0 with only one value, there are 7 nodes and 7 values stored in the tree.

Test Objective: Verify that a node's predecessor can be found.				
Class	**Method**	**Scenario**	**Input values**	**Result**
AVLTree<Double, ArrayList<Integer>, Integer>	predecessor	setup2	39.1 45.0 80.0 88.8	The predecessors found for the nodes with the given keys have the following keys respectively: 35.5, 39.1, 45.0, 83.7

Test Objective: Verify that a node's successor can be found.				
Class	**Method**	**Scenario**	**Input values**	**Result**
AVLTree<Double, ArrayList<Integer>, Integer>	successor	setup2	39.1 45.0 80.0 88.8	The successors found for the nodes with the given keys have the following keys respectively: 45.0,80.0,83.7,90.4

Test Objective: Verify that the tree is correctly traversed in preorder				
Class	**Method**	**Scenario**	**Input values**	**Result**
AVLTree<Double, ArrayList<Integer>, Integer>	preOrder	setup2	The tree root	The method must return "80.0 39.1 35.5 45.0 88.8 83.7 90.4 "

Test Objective: Verify that the tree is correctly traversed in inorder				
Class	**Method**	**Scenario**	**Input values**	**Result**
AVLTree<Double, ArrayList<Integer>, Integer>	inOrder	setup2	The tree root	The method must return "35.5 39.1 45.0 80.0 83.7 88.8 90.4 "

Test Objective: Verify that the tree is correctly traversed in preorder

Class	Method	Scenario	Input values	Result
AVLTree<Double, ArrayList<Integer>, Integer>	preOrder	setup2	The tree root	The method must return "35.5 45.0 39.1 83.7 90.4 88.8 80.0 "

Test Objective: Verify that player data is successfully added via platform.				
Class	**Method**	**Scenario**	**Input values**	**Result**
FIBA	addPlayerDataByPlatform	setup1	"AAA", "BBB", "21", "CCC", "56.4", "23.6", "98.0", "77.2", "11.3", "33.3" "Rayna","Madox","26","Phoenix Suns","41.76","7.91","46.04","30.66","72.4","47.09"	The program creates a file “data/players.csv” where it stores its data. The second line of the file is "\"AAA\",\"BBB\",\"CCC\",\"21\",\"56.4\",\"23.6\",\"98.0\",\"77.2\",\"11.3\",\"33.3\"" The third line of the file is "\"Rayna\",\"Madox\",\"Phoenix Suns\",\"26\",\"41.76\",\"7.91\",\"46.04\",\"30.66\",\"72.4\",\"47.09\""

Test Objective: Verify that player data is successfully added via file.				
Class	**Method**	**Scenario**	**Input values**	**Result**
FIBA	addPlayerDataByTextFile	setup1	file: “resources/200k.csv” The file exists and it has 200001 lines.	The file created by the program exists and it has at least 200001 lines.

Test Objective: Verify that the program returns the correct players when given a search criteria				
Class	**Method**	**Scenario**	**Input values**	**Result**
FIBA	searchPlayerIn	setup2	'>', "True Shooting", 80.0	There are 22 results and every one has their true shooting above 80
FIBA	searchPlayerIn	setup2	'=', "Usage", 7.91	There is one result and its usage is 7.91

FIBA	searchPlayerIn	setup2	'<', "Rebound", 20.0	There are 19 results and every one has their rebound under 20
FIBA	searchPlayerIn	setup2	'≥', "Blocks", 70.0	There are 24 results and every one has their blocks at least at 70
FIBA	searchPlayerIn	setup2	'≤', "Assist", 50.0	There are 49 results and every one has their assist at most at 50
FIBA	searchPlayerIn	setup2	'≥', "Defensive", 30	There are 70 results and every one has their blocks at least at 30

Test Objective: Verify that the program returns the correct players when given a search criteria.				
Class	**Method**	**Scenario**	**Input values**	**Result**
FIBA	searchPlayer	setup2	'>', '<', "True Shooting", 50.0, 70.0	There are [x] results and every one has their true shooting above 50 and under 70
FIBA	searchPlayer	setup2	'≥', '≤', "Usage", 40.0, 60.0	There are [x] results and every one has their usage at at least 40 and at most at 60
FIBA	searchPlayer	setup2	'<', '>', "Rebound", 10.0, 30.0	There are [x] results and every one has their rebound under 10 and over 30
FIBA	searchPlayer	setup2	'≤', '≥', "Assist", 60.0, 80.0	There are [x] results and every one has their assist at most at 60 and at least at 80
FIBA	searchPlayer	setup2	'≥', '<', "Blocks", 70.0, 90.0	There are [x] results and every one has their blocks at least at 70 and under 90
FIBA	searchPlayer	setup2	'>', '≤', "Defensive", 80.0, 100.0	There are [x] results and every one has their blocks over 80
FIBA	searchPlayer	setup2	'≤', '>', "True Shooting", 0.0, 20.0	There are [x] results and every one has their true shooting at most at 0 and over 20
FIBA	searchPlayer	setup2	'<', '≥', "Usage", 40.0, 70.0	There are [x] results and every one has their usage at

				under 40 and at least at 70
FIBA	searchPlayer	setup2	'≤', '≤', "Rebound", 0.0, 30.0	There are [x] results and every one has their rebound at most at 0.0 and at most at 30.0
FIBA	searchPlayer	setup2	'≥', '≥', "Assist", 50.0, 80.0	There are [x] results and every one has their assist at least at 50 and at least at 80
FIBA	searchPlayer	setup2	'<', '>', "Blocks", 60.0, 90.0	There are [x] results and every one has their blocks under 60 and over 90
FIBA	searchPlayer	setup2	'>', '>', "Defensive", 70.0, 100.0	There are [x] results and every one has their blocks over 70 and over 100

Test Objective: Verify that the program modifies a player successfully.				
Class	**Method**	**Scenario**	**Input values**	**Result**
FIBA	modifyPlayerData	setup2 The file exists and the player on line three has the name Maria.	“Name”, “Maria”, 0	The name of the player is Maria instead of Zondra. The information changes in the file and in the program.

Test Objective: Verify that the program deletes a player successfully.				
Class	**Method**	**Scenario**	**Input values**	**Result**
FIBA	deletePlayer	setup2 The file exists.	listOfPlayers, 0	The player with data: fisrtName:Rayna, lastName:Madox, team:Phoenix Suns, age:26, trueShooting: 41.76, usage:7.91, assist:46.04, rebound:30.66, defensive:72.4,

				blocks:47.09 is deleted in the program. In the file, the line where the player was is replaced for a line break.

Step 7 - Design implementation

Link to the source folder of the design implementation: https://bit.ly/3tUL5tU