Engineering method

1. **Identification of the problem**

1.1 Identification of needs and symptoms

- The solution to the problem must be able to work with large amounts of data.
- The solution to the problem must allow to enter data in a massive way through, for example, csv files.
- The solution to the problem must allow data to be entered through a graphical interface.
- The solution should allow faster searches based on one or more criteria.
- The information that is entered into the program must be stored and processed in secondary memory

1.2 Definition of the problem:

The International Basketball Federation wants an application that allows it to quickly store and access information on basketball players worldwide.

1.3 Functional requirements

The software must be able to:

REQ 1: Receive data either through database or using the graphical interface entering the required information.

-Req1.1: Allow user to create a player using the graphical interface.

-Req1.2: Read the database.

REQ 2: Create players, using the input data.

-Req2.1: Give to the player a name.

-Req2.2: Give to the player an age.

-Req2.3: Give to the player a team.

-Req2.4: Give to the player a points per game field.

-Req2.5: Give to the player a rebounds per game field.

-Req2.6: Give to the player a assists per game field.

-Req2.7: Give to the player a steals per game field.

-Req2.8: Give to the player a blocks per game field.

REQ 3: Manage the big amount data.

-Req3.1: Allow user to add a player.

-Req3.2: Allow user to delete a player.

-Req3.3: Allow user to update the players statistics information.

REQ 4: Allow user to ask for specific type of statistic players information giving to four of five types of information an index which means that type of information must be returned faster.

-Give to four of five types of player information an index.

-Collect all the players specific data and return it.

-Return the types with index specific information faster, using a complexity of O(log n).

-Allow user to ask about points per game.

-Allow user to ask about assists per game.

-Allow user to ask about steals per game.

-Allow user to ask about blocks per game.

-Allow user to ask about rebounds per game.

REQ5: Create balanced binary tree saving into it the specific statistic information with index, then justify the BBT usage by comparing the time taken by the search of a specific type of information with index and specific type of information without index.

-Req5.1: Save the players data.

-Req5.2: Show user how long it takes to ask for an information without index.

-Req5.3: Show user how long it takes asking for information with index

2. Compilation of information

Some terms that should be clear are:

- **Statistical item:** Category of statistical nature that allows bringing together entities that share certain characteristics in the same set.
- **Point:** It is an amount that allows to declare a winner. The team with the most points will be the winner.
- **Rebounds:** Is the act of gaining possession of the ball after a missed field goal or free throw.

- **Assist:** An assist is attributed to a player who passes the ball to a teammate in a way that leads to a score by field goal, meaning that they were "assisting" in the basket.
- **Steal:** Steals are credited to the defensive player who causes the fumble first, even if it does not end with possession of the live ball.
- **Block:** It is a game action in which the defensive player can cause the ball not to enter the basket.

It is also important to know the most effective data structures for this problem:

- **Tree data structure:** Is a collection of nodes, where each node is a data structure consisting of a value and a list of references to nodes. The start of the tree is the "root node" and the reference nodes are the "children"
- **Binary Tree:** Is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
- **Binary search Tree:** Is a node-based binary tree data structure which 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.
- **AVL Tree:** Is a self-balancing binary search tree in which the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.

3. Search of creative solutions

Bearing in mind that the problem requires both storing information and finding this information quickly through searches. The best thing is that the solutions are proposals of data structures.

- Using arrayList.
- Using ordered linked lists.
- Using hashtables.
- Using binary search trees.
- Using AVL Trees.
- Using queues.
- Using stacks.

4. Transition from formulation of ideas to preliminary designs

The arraylist is discarded because with large amounts of data, the complexity of the searches is too big. Same happens with ordered linked lists.

Stacks and queues are inefficient since they only allow one data to be obtained, either the last one entered in the case of stacks, or the first one entered in the case of queues.

Careful review of the other proposals leads us to the following:

Binary search trees:

- They are especially useful for searching.
- They are used to implement dynamic sets
- In an average case it has O(n log n) complexity.
- In the worst case it has O(n) complexity.

AVL Trees:

- They are especially useful for searching.
- They are used to implement dynamic sets
- In an average case it has O(n log n) complexity.
- In the worst case it has O(n log n) complexity.

Hashtables:

- They works through keys converted by hashing.
- There are different types of hashing for different situations.
- In an average case it has O(1) complexity.
- In the worst case it has O(n) complexity.

5. Evaluation and selection of the best solution
Criteria:

Criterion A. Data storage. The structure allows to save:
[2] All kinds of data
[1] Primitive data

Criterion B. Adaptability to sizes. The structure is fully efficient with:

[3] Large amounts of data and consequently, with all kinds of data sizes.

[2] Moderate amounts of data and small amounts.

[1] Only small amounts of data

Criterion C. Search complexity with average cases. In an average case, the complexity of searching is:
[5] O(1)
[4] O(log n)

[3] O(n)
[2] O(n^2)
[1] Greater than n^2

Criterion D. Search complexity with the worst case. In the worst case, the complexity of searching is:
[5] O(1)
[4] O(log n)
[3] O(n)
[2] O(n^2)
[1] Greater than n^2

Criterion E. Ease of implementation. The difficulty for developers to implement this structure for the context of the problem is:

[5] Low
[4] Moderately low
[3] Intermediate
[2] Moderately high
[1] High

Evaluation

	Criterion A	Criterion B	Criterion C	Criterion D	Criterion E	Total
Binary search trees	2	3	4	3	3	15
AVL Trees	2	3	4	4	3	16
Hashtables	2	2	5	3	2	14

Selection
According to the previous criteria, the two structures with the highest scores are chosen, in this case, AVL Tree with a score of 16 and binary search tree with a score of 15.

Webgraphy

- Wikipedia, the free encyclopedia
- GeeksforGeeks | A computer science portal for geeks