



**Engineering Method
Quick Basket**

Alejandro Coronel Sánchez / A00365049

Julián Andrés Riascos / A00365548

Kevin Alejandro Mera / A00364415

Andrés Alberto Aristizábal, Ph.D.

2021-1

ICESI University

Santiago de Cali

Valle del Cauca

Colombia

Problematic Context

Basketball is a sport that, given its trajectory and reputation over the years, is currently strongly consolidated around the world. Over the years it has been of great importance to record those statistical data that allow an in-depth analysis to be carried out. FIBA therefore requires a data storage and quick query application per player that allows focusing on general basketball predictions.

Solution Development

To solve the problem of the quick basket properly, we will use the engineering method.

Step 1. Identification of the problem

- The amount of data to use will be very heavy.
- Data analysis is required.
- It is required to analyze patterns and trends in sport.
- The temporal complexity for search players cannot be linear.
- The solution needs to be efficient and quick.
- Binary search trees are required.
- It must have a graphical interface for user interaction.

Step 2. Information Compilation

First, we look for other databases that handle statistics of basketball players.

NBA database was the database that we look for references, it has a filter for each player statistic, and it also can be consulted by season.

NBA Games Schedule Watch News Standings Teams Stats Players Fantasy Playoffs NBA Official NBA TV League Pass Store Tickets Sign In																					
Stats Home Players Teams Leaders Stats 101 Tools Quick Links Events																					
Regular Season Per Game DREB																					
DREB x RECENT FILTERS GLOSSARY SHARE																					
233 Rows Page 1 of 5																					
#	PLAYER	GP	MIN	PTS	FGM	FGA	FG%	3PM	3PA	3P%	FTM	FTA	FT%	OREB	DREB	REB	AST	STL	BLK	TOV	EFF
1	Rudy Gobert	61	31.0	14.4	5.5	8.2	67.7	0.0	0.0	0.0	3.4	5.4	61.9	3.3	10.1	13.4	1.3	0.5	2.8	1.6	26.2
2	Giannis Antetokounmpo	52	33.7	28.5	10.3	18.2	56.8	1.1	3.6	30.5	6.8	9.9	68.5	1.7	9.7	11.4	6.0	1.2	1.3	3.7	33.8
3	Clint Capela	55	30.4	15.4	6.7	11.3	58.9	0.0	0.0	0.0	2.1	3.6	57.7	4.9	9.7	14.6	0.8	0.7	2.1	1.2	26.3
4	Russell Westbrook	55	35.8	21.7	8.3	18.9	43.7	1.3	4.0	31.1	3.9	6.2	63.0	1.7	9.5	11.2	11.0	1.3	0.3	5.0	27.6
5	Nikola Vucevic	63	33.6	23.9	9.7	19.8	48.9	2.6	6.2	41.8	2.0	2.4	83.3	2.0	9.4	11.4	3.7	1.0	0.7	1.8	28.4
6	Julius Randle	62	37.5	24.1	8.5	18.3	46.2	2.2	5.3	42.1	4.9	6.0	80.8	1.3	9.1	10.4	5.9	1.0	0.3	3.4	27.1
7	Domantas Sabonis	53	35.7	19.9	7.5	14.4	52.0	0.8	2.6	30.2	4.1	5.6	73.1	2.5	9.1	11.6	6.0	1.1	0.5	3.4	27.4
8	Jonas Valanciunas	53	27.9	17.0	7.1	12.2	58.0	0.4	1.0	37.0	2.5	3.2	77.3	4.0	8.2	12.2	1.8	0.5	0.7	1.7	24.8

Reference: <https://www.nba.com/stats/leaders/?Season=2020-21&SeasonType=Regular%20Season&StatCategory=DREB>

Trees

Trees are special case of graphs, where nodes and edges (links) do not form a cycle.

Tree Glossary

Root: is the top node in the tree.

Child: any node that is emerged from an upper node.

Parent/Internal Node: node with at least one child.

Siblings: nodes sharing the same parent.

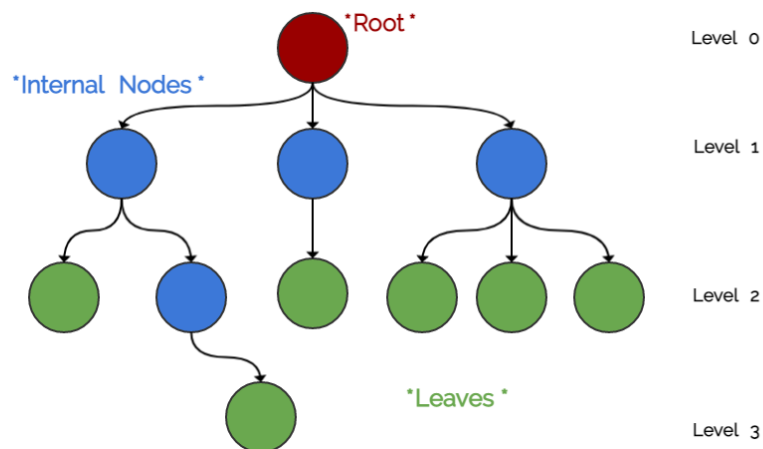
Leaf: node with no children.

Edge: the link between two nodes (a parent and its child).

Path: the sequence of links and nodes to reach from one node to a descendant.

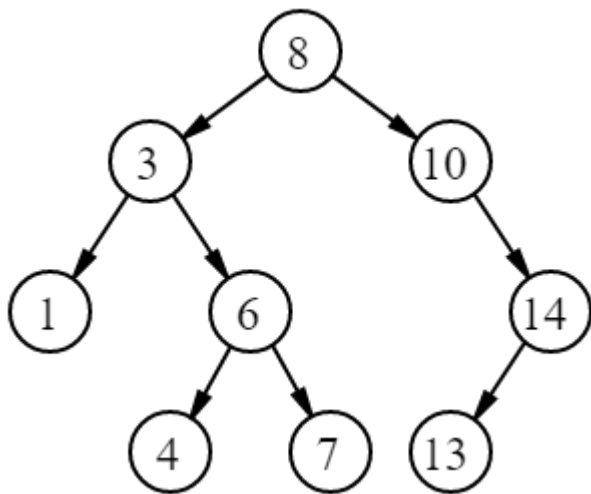
Height of node: the number of links between a node and the furthest leaf.

Depth of node: the number of links between a node and the root.



Binary Search Tree

Binary trees are a special case of trees where each node can have at most 2 children. Also, these children are named: left child or right child. A very useful specialization of binary trees is binary search tree (BST) where nodes are conventionally ordered in a certain manner. By convention, the left children < parent < right children, and this rule propagates recursively across the tree.



Linked List

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node point to the next node.



As per the above illustration, following are the important points to be considered:

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Advantages:

1. LinkedList is that insertions and deletion can be done very quickly.
2. If you just want to insert an element right to the beginning of the LinkedList, that can be done in constant time $O(1)$.
3. If you want to delete an element at the beginning of a LinkedList, again constant time $O(1)$.

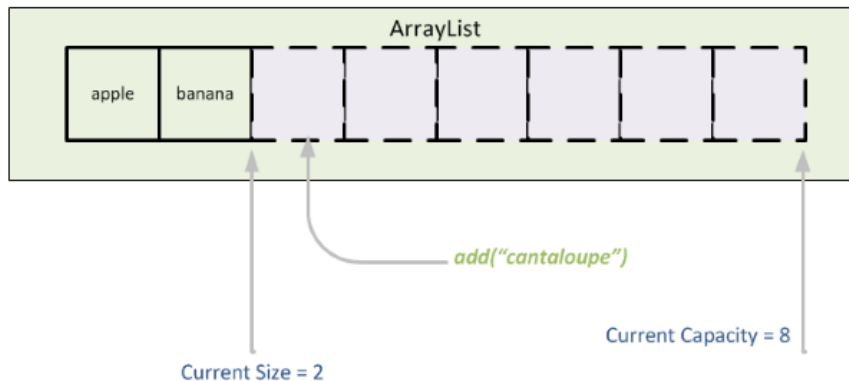
Disadvantages:

- If you want to append an item at the end of the list, that might require going through the whole LinkedList, until you reach the very last element, and then inserting the element, this will take Linear time $O(n)$.

ArrayList

An ArrayList, or dynamically resizing array, allows you to have the benefits of an array while offering flexibility in size. You won't run out of space in the ArrayList since its capacity will grow as you insert elements.

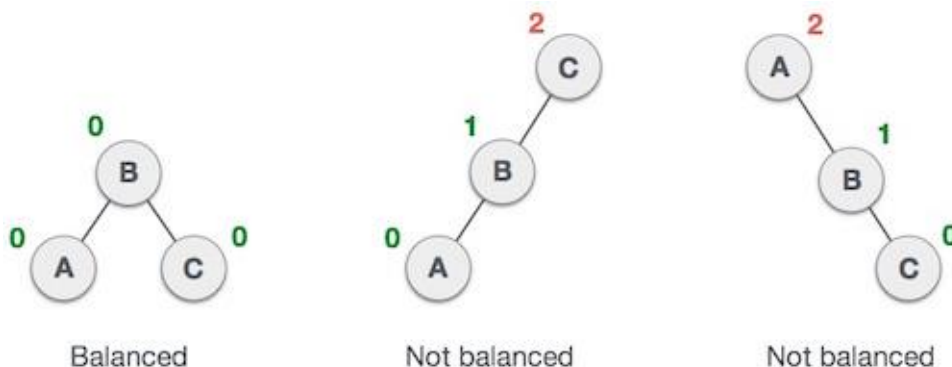
Array can contain both primitive data types as well as objects of a class depending on the definition of the array. However, ArrayList only supports object entries, not the primitive data types.



AVL Trees

AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced, and the next two trees are not balanced



In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) – height(right-sutree)

AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations:

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations, and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2.

What is a Hash Function?

A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as the index in the hash table.

Choosing a good hashing function, $h(k)$, is essential for hash-table based searching. h should distribute the elements of our collection as uniformly as possible to the "slots" of the hash table. The key criterion is that there should be a minimum number of collisions.

If the probability that a key, k , occurs in our collection is $P(k)$, then if there are m slots in our hash table, a uniform hashing function, $h(k)$, would ensure:

$$\sum_{k|h(k)=0} P(k) = \sum_{k|h(k)=1} P(k) = \dots = \sum_{k|h(k)=m-1} P(k) = \frac{1}{m}$$

Sometimes, this is easy to ensure. For example, if the keys are randomly distributed in $(0,r]$, then,

$$h(k) = \text{floor}((mk)/r)$$

will provide uniform hashing.

Most hashing functions will first map the keys to some set of natural numbers.

Having mapped the keys to a set of natural numbers, we then have several possibilities.

Use a **mod** function:

$$h(k) = k \bmod m.$$

When using this method, we usually avoid certain values of **m**. Powers of 2 are usually avoided, for $k \bmod 2^b$ simply selects the **b** low order bits of **k**.

Use the multiplication method:

- Multiply the key by a constant **A**, $0 < A < 1$,
- Extract the fractional part of the product,
- Multiply this value by **m**.

Thus, the hash function is:

$$h(k) = \text{floor}(m * (kA - \text{floor}(kA)))$$

In this case, the value of **m** is not critical.

Use universal hashing:

A malicious adversary can always choose the keys so that they all hash to the same slot, leading to an average $O(n)$ retrieval time. Universal hashing seeks to avoid this by choosing the hashing function randomly from a collection of hash functions.

Functional requirements

FR1: Massively enter data as a .csv file.

FR2: Enter the data of the players (full name, identification, age, team, points per game, re bounds per game, assistance per game, robberies per game, blocks per game and general evaluation) using the graphical interface.

FR3: Delete data from the players within the data base given his identification using the graphical interface.

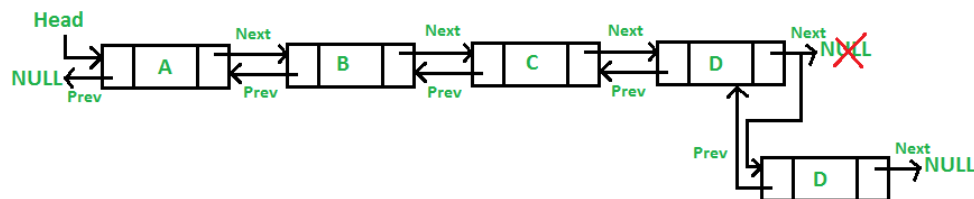
FR4: Make efficient consultations by points per game, re bounds per game, assistance per game, robberies per game, blocks per game and general evaluation for all the players in the data base.

Step 3. Search of creative solutions

Since we already investigated and inquired about the operation of data structures such as binary search tree (BST), AVL tree, a red-black tree, linked List, ArrayList and hash tables, we can consider different possible solutions to our problem. These can be:

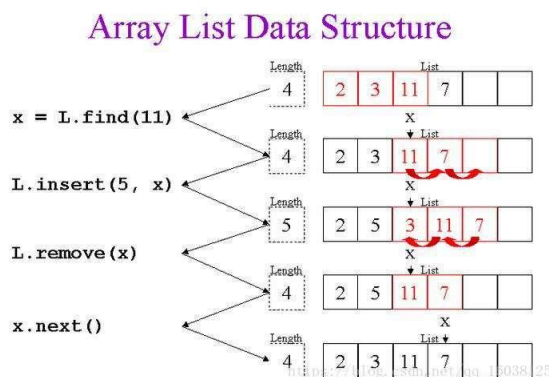
Option 1. Using only LinkedList

We can make use of a lot of double linked lists since it will help us to make recursive queries about the players. These lists could be built in a better way but with more complex methods to try to resemble this structure to our real functionality of the players headings and in general terms to the operation of the program.



Option 2. Using only ArrayList

For this option is possible to use the ArrayList data structure from the Java API as the unique for all the requirements mentioned. This will allow us to consult the players in a pleasant way without resorting to recursive methods. Score histories per player may be implemented in terms of statistics.



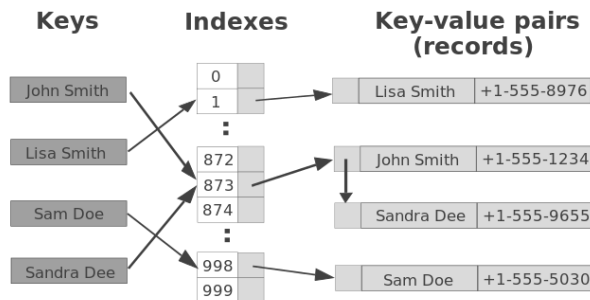
Option 3. Using LinkedList and ArrayList

Using these two list structures could provide us with a slightly more selective data management. Well, the recursive methods of the linked list would be useful for the queries while the array list could be used to order players, import, export, edit, delete, add in an

easier way, or make queries that are not complex in terms of time. A little more decisive analysis for use would have to be carried out.

Option 4. Using ArrayList and HashTable

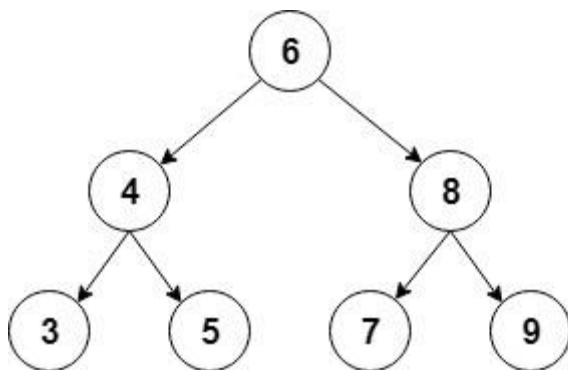
On the other hand, we could think of using hash tables to, in this case, store the statistics of the players, accessing in an efficient way and therefore having quick queries. while the players could be stored in an array list.



Here, the key could be the player name and the value an attribute. This option would need various hash tables to contain all the statistical information of each player.

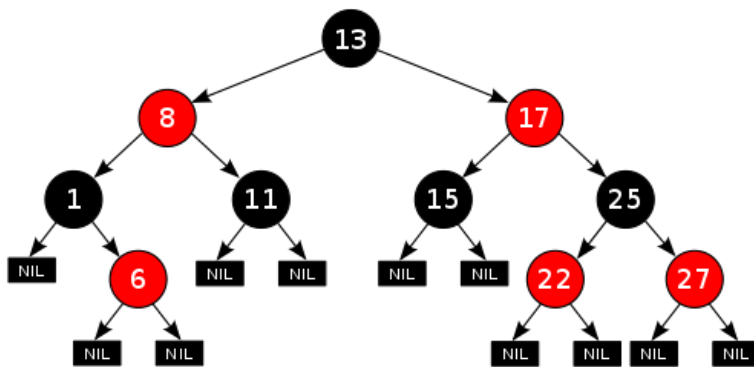
Option 5. Using binary search tree and AVL tree

We can see that it is convenient for this option to use binary search trees since it would be able to retrieve players according to a search category selected through indices associated with the players. This recursive data structure will store in it the value of the attribute and the disk position of the player to which it belongs. On the other hand, AVL trees will help us to maintain an ideal search time when the amount of data handled by the program is excessively high.



Option 6. Using binary search tree, AVL tree and red-black tree

We can see that it is convenient for this option to use binary search trees since it would be able to retrieve players according to a search category selected through indices associated with the players. This recursive data structure will store in it the value of the attribute and the disk position of the player to which it belongs. On the other hand, AVL trees will help us to maintain an ideal search time when the amount of data handled by the program is excessively high. Additionally, the red-black trees could also serve us to organize information composed of comparable data (for example, numbers) or, in our program, they could be used to compare statistics of the players when requesting said information from the person requesting it.



Here, we could assume minimum one of our auto-balanced binary search trees is a red-black tree.

Step 4. Ideas transitions to preliminary design

At this point, we **discard the Option 1. Using only LinkedList** since we have some data structures that are made for manage, order and access information much faster and in a clearer way. Assuming that only the linked list is used to solve our problem, it could lead to a much more expensive programming in terms of time and adaptation to our program. Therefore, this option is completely discarded, and the other options are reviewed.

We can see that **Option 2. Using only ArrayList** very similar to option 1, since here instead of using linked lists, only the java API array list will be used. Of course, with this structure recursive methods will not be used, which would facilitate the implementation of complex methods and that we already have a structure that provides a variety of methods for our solution, the problem here would be that it is not efficient enough since it would be cumbersome like option 1 in implementing this option to our solution. It does not leave any alternative to use other data structures that could be adapted in a better way to what you want to do in the program and thus fulfilling the requirements.

Now, we could consider the **Option 3. Using LinkedList and ArrayList**. But we can find that they are still very basic and non-modeling solutions to our problem in the best way. It would again be tedious in terms of time and complexity to adapt the arraylists and linked

lists to the required solution. Additionally, it would not comply with the conditions in which the obtaining of the statistical data of the players is in a maximum time of $O(\log n)$. With these structures it would be $O(n)$.

Careful review of the previously discarded alternatives leaves us with the following options to be analyzed:

Option 4. Using ArrayList and HashTable

- Would have a good organization when using hash tables.
- The access times to the hash table would be very fast.
- The array list could be ordered in the most convenient way and by independent criteria.
- It would be close to solving the problem.

Option 5. Using binary search tree and AVL tree

- Appropriate data structures are used.
- Better organization according to the problem that arises.
- Search algorithms are implemented that are more effective.
- Less ambiguity when using different data structures and easier to understand the code for future updates.

Option 6. Using binary search tree, AVL tree and red-black tree

- Data can be purchased in a timely and rapid manner.
- Does not require complex changes to adapt to the solution.
- Operations on red-black trees are more economical in time because it is not necessary to maintain the vector of values Red-black tree.
- For moderate volumes of values, inserts and deletions in a binary color tree are faster Red-black tree.

Step 5. Evaluation and best solution selection

The criteria we chose in this case are the ones we list below. Next to each one a numerical value has been established with the aim of establishing a value that indicates which of the possible values of each criterion have the most tendency to be a better candidate (i.e., they are more desirable).

Criteria:

1. Criterion A. Organization. The option is organized and understandable for future improvements and data structures provide a logic organization when executing the program:

[3] Fully uses the appropriate and correct data structures to avoid ambiguities when programming.
[2] Uses some data structures correctly to avoid ambiguities when programming.
[1] It uses some data structures, but it is not safe to avoid ambiguities when programming.
2. Criterion B. Efficiency. The maximum complexity that may exist at the time of sorting anything data.
[6] Logarithmic [$O(\log n)$]
[5] Lineal [$O(n)$]
[4] Quadratic [$O(n^2)$]
[3] Cubic [$O(n^3)$]
[2] Exponential [$O(x^n)$]
[1] Factorial [$O(n!)$]
3. Criterion C. Ease of implementation.
[1] Easy
[0] Hard
4. Criterion D. Can handle large volumes of data without affecting program performance.
[1] Yes
[0] No or partially no

Evaluation

Evaluating the criteria of the last three alternatives, we have the following table:

	Criterion A	Criterion B	Criterion C	Criterion D	Total
<u>Option 4.</u> <u>Using</u> <u>ArrayList and</u> <u>HashTable</u>	[1] It uses some data structures, but it is not safe to avoid ambiguities when programming.	[5] Lineal $[O(n)]$	[0] Hard	[0] No or partially no	6
<u>Option 5.</u> <u>Using binary</u> <u>search tree</u> <u>and AVL tree</u>	[2] Uses some data structures correctly to avoid ambiguities when programming.	[6] Logarithmic $[O(\log n)]$	[1] Easy	[1] Yes	9
<u>Option 6.</u> <u>Using binary</u> <u>search tree,</u> <u>AVL tree and</u> <u>red-black tree</u>	[3] Fully uses the appropriate and correct data structures to avoid ambiguities when programming.	[6] Logarithmic $[O(\log n)]$	[1] Easy	[1] Yes	10

Selection

According to the previous analysis of criteria, it is concluded that option 6 should be chosen since it accumulates a higher score compared to option 4 or 5.

Step 6. Preparation of Reports and Specifications

Problem specification (input / output)

Problem: an efficient system of basketball player inquiries

Inputs: an .csv file with players, id, full name, age, team, points per game, rebounds per game, assists per game, robberies per game, blocks per game and general evaluation.

Outputs: An interface to perform faster queries where the data is stored in binary trees.

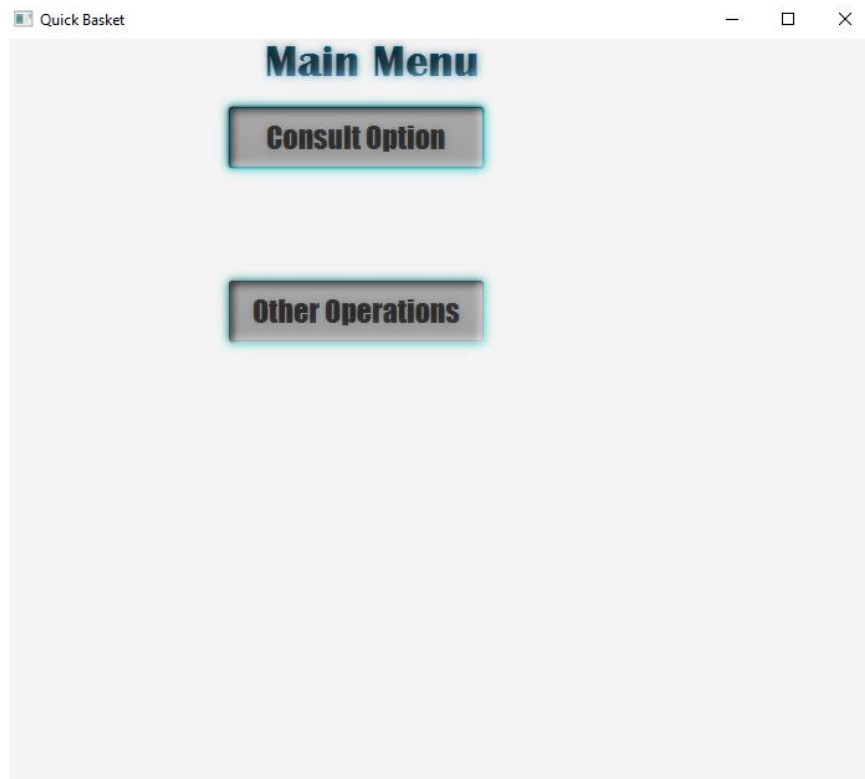
Step 7. Design implementation

Implementation on Java programming language.

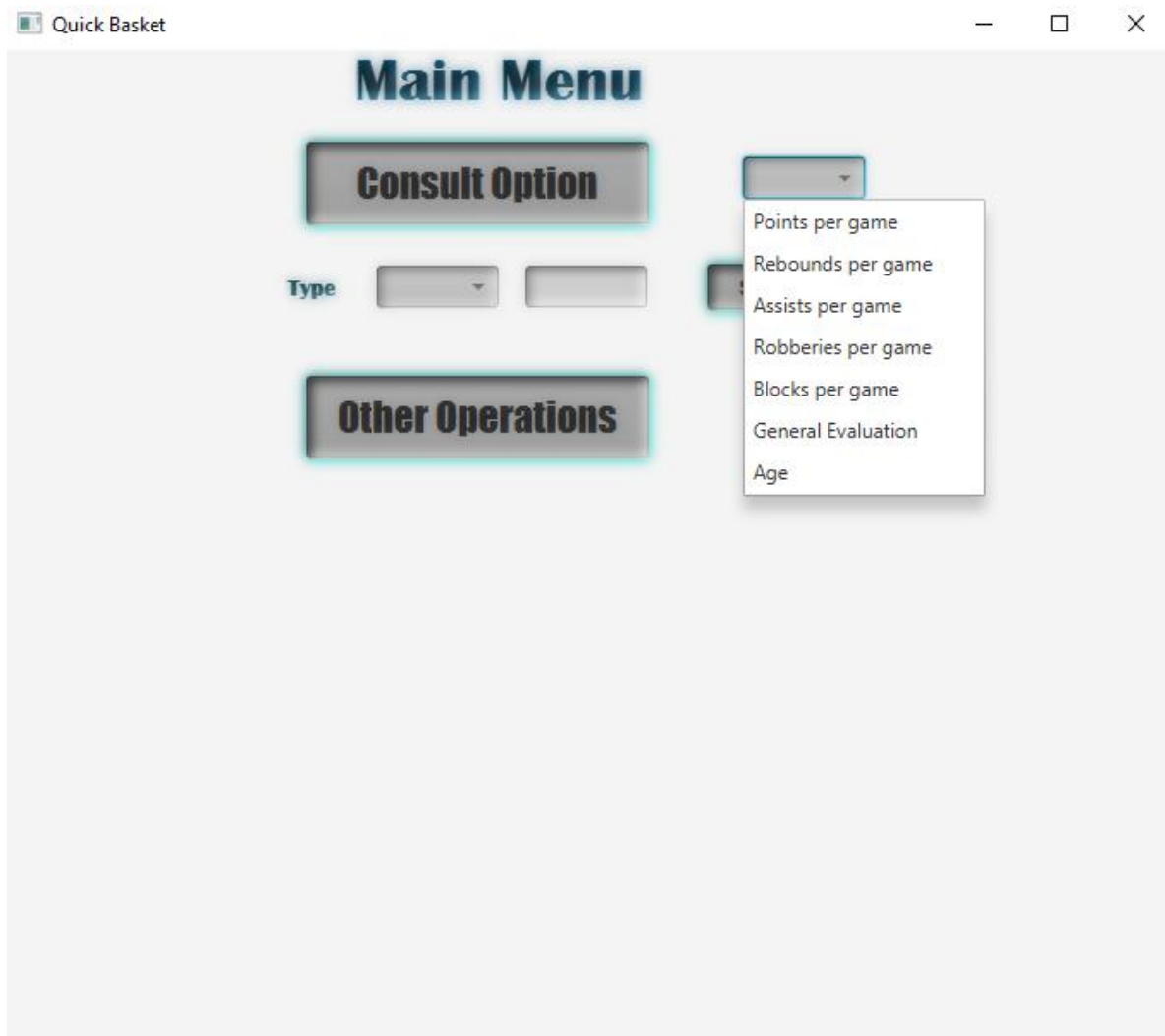
- a) List of things to implement:
- b) Import the players data to efficient data structures like AVL, BST and red-black trees.
- c) Be able to add or remove a player directly from the .csv file.
- d) Allow displaying the players depending on the type of query requested.

GUI Mockups

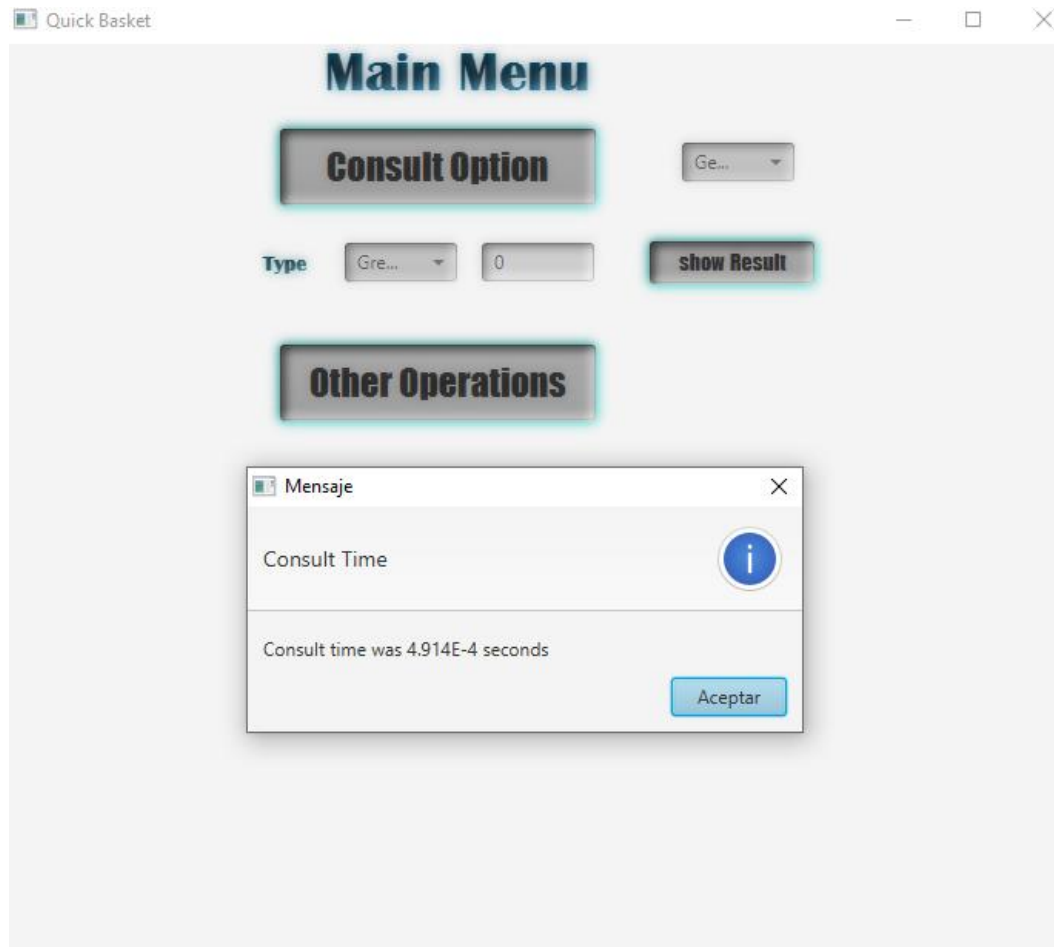
Initial view, consult button option let you choice the heading.



After press it, at the right there a combo box with headings, at the left another combo box to select if you want to search values equal to, greater or lowers regard to the numerical value to find.



A message will appear always when a query is done.



Then the table with the information that was selected is showed.

Quick Basket

ID	FullName	Age	Team	points...	rebounds...	assistspe...	robberie...	blocksperG...	generalEv...
2	Emelina	22	Virgin Isl...	53.0	7228.0	745.0	714.0	547.0	1.0
32	Odessa	31	Kuwait	60.0	4189.0	369.0	580.0	811.0	1.0
153	Theodora	32	Puerto R...	72.0	2045.0	333.0	972.0	673.0	1.0
307	Rosaline	29	Greece	4.0	9181.0	324.0	439.0	300.0	1.0
522	Hildegard	24	Yemen	136.0	7377.0	790.0	613.0	13.0	1.0
555	Dode	22	Cuba	3.0	2253.0	759.0	697.0	495.0	1.0
665	Karina	28	Swaziland	97.0	1161.0	308.0	12.0	140.0	1.0
675	Karly	23	Dominica	118.0	3088.0	1.0	920.0	665.0	1.0
768	Elie	40	"Lao Peo...	131.0	5772.0	85.0	653.0	306.0	1.0
795	Sheelagh	27	Jamaica	38.0	2824.0	73.0	164.0	63.0	1.0
846	Audrie	34	Monaco	84.0	4693.0	939.0	773.0	445.0	1.0
995	Vonny	22	Moldova...	94.0	6020.0	50.0	145.0	520.0	1.0
1073	Hope	28	Hungary	113.0	1060.0	663.0	921.0	618.0	1.0
1468	Brynna	36	Australia	30.0	4964.0	847.0	196.0	797.0	1.0
1532	Eolanda	23	Nauru	55.0	560.0	669.0	462.0	363.0	1.0
1695	Fanny	33	British In...	66.0	2806.0	11.0	711.0	581.0	1.0
1699	Alleen	32	Denmark	138.0	7963.0	712.0	822.0	277.0	1.0
1711	Marylou	28	Netherla...	84.0	3783.0	820.0	992.0	802.0	1.0

Return

If you want to add or delete a player, click on other operations button, then select the operation to do. The elements to complete your request will be displayed on the screen.

Quick Basket

Main Menu

Consult Option

Other Operations

select the operation type

Id	<input type="text"/>	assistsPerGame	<input type="text"/>
full Name	<input type="text"/>	reBoundsPerGame	<input type="text"/>
Age	<input type="text"/>	robberlesPerGame	<input type="text"/>
Team	<input type="text"/>	BlocksPerGame	<input type="text"/>
pointsPerGame	<input type="text"/>	general Evaluation	<input type="text"/>

Add

Quick Basket

Main Menu

Consult Option

Other Operations

select the operation type

Id

Delete

Class Diagram Design

For better reading convenience of the diagram, the link of the file that is housed in the docs folder is attached. [Class diagram design link](#)

Tests cases design

GenericAVLTreeTest

Sceneries configuration

Name	Class	Stage
setupScenary1	GenericAVLTree	A GenericAVLTree with entries = {(A,1), (B,2), (C,3)}
setupScenary2	GenericAVLTree	A GenericAVLTree with entries = {(D,7), (C,5), (A,6)}
setupScenary3	GenericAVLTree	A GenericAVLTree with entries = {(F,10), (C,9), (D,13)}
SetupScenary4	GenericAVLTree	A GenericAVLTree with entries = {(A,6), (E,12), (B,2)}
SetupScenary5	GenericAVLTree	A GenericAVLTree with entries = {(A,6), (E,12), (B,2), (B,3), (B,8), (E,13), (E,14), (E,15), (E,16)}
SetupScenary6	GenericAVLTree	A GenericAVLTree with entries = {(A,1), (A,2), (A,3), (B,4), (B,5), (B,6), (C,7), (C,8), (C,9)}

Test cases design

Test objective: check if GenericAVLTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericAVLTree	search	setupScenary1	To search node key = "A"	Search key "A" and the value is 1
GenericAVLTree	getRoot	setupScenary1	None	Search in the root tree and get the position 0 of value list and is 2
GenericAVLTree	getRoot	setupScenary1	None	Search in the right of the root tree and get the position 0 of values list and is 3
GenericAVLTree	getRoot	setupScenary1	None	Search in the left of the root tree and get the key "A"

Test objective: check if GenericAVLTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericAVLTree	search	setupScenary2	To search node key = "D"	Search key "D" and the value is 7
GenericAVLTree	getRoot	setupScenary2	None	Search in the root tree and get the position 0 of value list and is 5
GenericAVLTree	getRoot	setupScenary2	None	Search in the right of the root tree and get the position 0 of values list and is 7
GenericAVLTree	getRoot	setupScenary2	None	Search in the left of the root tree and get the key "A"

Test objective: check if GenericAVLTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericAVLTree	search	SetupScenary3	To search node key = "D"	Search key "D" and the value is 13

GenericAV LTree	getRoot	SetupScenary3	None	Search in the root tree and get the position 0 of value list and is 13
GenericAV LTree	getRoot	SetupScenary3	None	Search in the right of the root tree and get the position 0 of values list and is 10
GenericAV LTree	getRoot	SetupScenary3	None	Search in the left of the root tree and get the key "C"

Test objective: check if GenericAVLTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericAV LTree	search	SetupScenary4	To search node key = "A"	Search key "A" and the value is 6
GenericAV LTree	getRoot	SetupScenary4	None	Search in the root tree and get the position 0 of value list and is 2
GenericAV LTree	getRoot	SetupScenary4	None	Search in the right of the root tree and get the position 0 of values list and is 12

GenericAVLTree	getRoot	SetupScenary4	None	Search in the left of the root tree and get the key “A”
----------------	---------	---------------	------	---

Test objective: check if GenericAVLTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericAVLTree	getRoot	SetupScenary5	None	Search in the root tree and get the position 0 of value list and is 2
GenericAVLTree	getRoot	SetupScenary5	None	Search in the root tree and get the position 1 of value list and is 3
GenericAVLTree	getRoot	SetupScenary5	None	Search in the root tree and get the position 2 of value list and is 8
GenericAVLTree	search	SetupScenary5	To search node key = “E”	Search key “E” and the value in position 0 on array is 12
GenericAVLTree	search	SetupScenary5	To search node key = “E”	Search key “E” and the value in position 1 on array is 13
GenericAVLTree	search	SetupScenary5	To search node key = “E”	Search key “E” and the value in position 2 on array is 14

Test objective: check if GenericAVLTree is returning the greater values by a key

Class	Method	Setup	Input	Result
GenericAVLTree	getGreater Than	SetupScenary6	Keys greater than "A"	Search greater than "A" counts 6
GenericAVLTree	getGreater Than	SetupScenary6	Keys greater than "B"	Search greater than "B" counts 3

Test objective: check if GenericAVLTree is returning the lower values by a key

Class	Method	Setup	Input	Result
GenericAVLTree	getGreater Than	SetupScenary6	Keys lower than "C"	Search lower than "C" counts 6
GenericAVLTree	getGreater Than	SetupScenary6	Keys lower than "B"	Search lower than "B" counts 3

GenericBinarySearchTree

Sceneries configuration

Name	Class	Stage
setupScenary1	GenericBinarySearchTree	A GenericBinarySearchTree with entries = {(A,1), (B,2), (C,3), (D,4), (B,5), (B,6)}
setupScenary2	GenericBinarySearchTree	A GenericBinarySearchTree with entries = {(C,1), (A,2), (B,3), (D,4), (E,5), (E,6)}
setupScenary3	GenericBinarySearchTree	A GenericBinarySearchTree with entries = {(G,1), (D,2), (E,3), (T,4), (U,5), (S,6), (C,7), (A,8), (B,9)}
SetupScenary4	GenericBinarySearchTree	A GenericBinarySearchTree with entries = {(A,1), (A,2), (A,3), (B,4), (B,5), (B,6), (C,7), (C,8), (C,9)}

Test cases design

Test objective: check if GenericBinarySearchTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericBinarySearchTree	search	SetupScenary1	To search node key = "A"	Search key "A" and the value is 1
GenericBinarySearchTree	search	SetupScenary1	To search node key = "B"	Search and get the position 0 of value list and is 2
GenericBinarySearchTree	search	SetupScenary1	To search node key = "B"	Search and get the position 1 of value list and is 5
GenericBinarySearchTree	search	SetupScenary1	To search node key = "B"	Search and get the position 2 of value list and is 6

Test objective: check if GenericBinarySearchTree is inserting, searching and relations are okay

Class	Method	Setup	Input	Result
GenericBinarySearchTree	search	SetupScenary2	To search node key = "A"	Search key "A" and the value is 2
GenericBinarySearchTree	search	SetupScenary2	To search node key = "C"	Search and get the position 0 of value list and is 1
GenericBinarySearchTree	search	SetupScenary2	To search node key = "D"	Search and get the position 0 of value list and is 4
GenericBinarySearchTree	search	SetupScenary2	To search node key = "E"	Search and get the position 1 of value list and is 6

Test objective: check if GenericBinarySearchTree is returning the greater values by a key

Class	Method	Setup	Input	Result
GenericBinarySearchTree	getGreater Than	SetupScenary4	Keys greater than "A"	Search greater than "A" counts 6

GenericBinarySearchTree	getGreater Than	SetupScenary4	Keys greater than "B"	Search greater than "B" counts 3
-------------------------	-----------------	---------------	-----------------------	----------------------------------

Test objective: check if GenericBinarySearchTree is returning the lower values by a key			
Method	Setup	Input	Result
getGreater Than	SetupScenary4	Keys lower than "C"	Search lower than "C" counts 6
getGreater Than	SetupScenary4	Keys lower than "B"	Search lower than "B" counts 3

GenericRedBlackTree

Sceneries configuration

Name	Class	Stage
setupScenary1	GenericRedBlackTree	A GenericAVLTree with entries = {(A,1), (B,2), (C,3), (D,4), (B,5), (B,6)}
setupScenary2	GenericRedBlackTree	A GenericAVLTree with entries = {(C,1), (A,2), (B,3), (D,4), (E,5), (E,6)}

setupScenary3	GenericRedBlackTree	A GenericAVLTree with entries = {(D,1), (C,2), (B,3), (A,4), (C,5), (B,6)}
SetupScenary4	GenericRedBlackTree	A GenericAVLTree with entries = {(C,1), (E,2), (D,3), (A,4), (B,5), (E,6)}
SetupScenary5	GenericRedBlackTree	A GenericAVLTree with entries = {(A,1), (A,2), (A,3), (B,4), (B,5), (B,6), (C,7), (C,8), (C,9)}

Test cases design

Test objective: check GenericRedBlackTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericRedBlackTree	search	SetupScenary1	To search node key = "A"	Search key "A" and the value is 1
GenericRedBlackTree	search	SetupScenary1	To search node key = "B"	Search and get the position 0 of value list and is 2
GenericRedBlackTree	search	SetupScenary1	To search node key = "C"	Search and get the position 0 of value list and is 3

GenericRedBlackTree	search	SetupScenario1	To search node key = "D"	Search and get the position 0 of value list and is 4
GenericRedBlackTree	search	SetupScenario1	To search node key = "B"	Search and get the position 1 of value list and is 5
GenericRedBlackTree	search	SetupScenario1	To search node key = "B"	Search and get the position 2 of value list and is 6

Test objective: check GenericRedBlackTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericRedBlackTree	search	SetupScenario2	To search node key = "C"	Search key "A" and the value is 1
GenericRedBlackTree	search	SetupScenario2	To search node key = "A"	Search and get the position 0 of value list and is 2
GenericRedBlackTree	search	SetupScenario2	To search node key = "B"	Search and get the position 0 of value list and is 3
GenericRedBlackTree	search	SetupScenario2	To search node key = "D"	Search and get the position 0 of value list and is 4

GenericRedBlackTree	search	SetupScenario2	To search node key = "E"	Search and get the position 0 of value list and is 5
GenericRedBlackTree	search	SetupScenario2	To search node key = "E"	Search and get the position 1 of value list and is 6

Test objective: check GenericRedBlackTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericRedBlackTree	search	SetupScenario3	To search node key = "D"	Search key "A" and the value is 1
GenericRedBlackTree	search	SetupScenario3	To search node key = "C"	Search and get the position 0 of value list and is 2
GenericRedBlackTree	search	SetupScenario3	To search node key = "B"	Search and get the position 0 of value list and is 3
GenericRedBlackTree	search	SetupScenario3	To search node key = "A"	Search and get the position 0 of value list and is 4
GenericRedBlackTree	search	SetupScenario3	To search node key = "C"	Search and get the position 1 of value list and is 5

GenericRedBlackTree	search	SetupScenario3	To search node key = "B"	Search and get the position 1 of value list and is 6
---------------------	--------	----------------	--------------------------	--

Test objective: check GenericRedBlackTree is inserting, searching and relations are okay				
Class	Method	Setup	Input	Result
GenericRedBlackTree	search	SetupScenario4	To search node key = "C"	Search key "A" and the value is 1
GenericRedBlackTree	search	SetupScenario4	To search node key = "E"	Search and get the position 0 of value list and is 2
GenericRedBlackTree	search	SetupScenario4	To search node key = "D"	Search and get the position 0 of value list and is 3
GenericRedBlackTree	search	SetupScenario4	To search node key = "A"	Search and get the position 0 of value list and is 4
GenericRedBlackTree	search	SetupScenario4	To search node key = "B"	Search and get the position 0 of value list and is 5
GenericRedBlackTree	search	SetupScenario4	To search node key = "E"	Search and get the position 1 of value list and is 6

Test objective: check if GenericRedBlackTree is returning the greater values by a key				
Class	Method	Setup	Input	Result
GenericRedBlackTree	getGreater Than	SetupScenary4	Keys greater than “A”	Search greater than “A” counts 6
GenericRedBlackTree	getGreater Than	SetupScenary4	Keys greater than “B”	Search greater than “B” counts 3

Test objective: check if GenericBinarySearchTree is returning the lower values by a key			
Method	Setup	Input	Result
getGreater Than	SetupScenary5	Keys lower than “C”	Search lower than “C” counts 6
getGreater Than	SetupScenary5	Keys lower than “B”	Search lower than “B” counts 3

QuickBasketManagerTest

Sceneries configuration

Name	Class	Stage
-------------	--------------	--------------

setupScenary1	QuickBasketManager	<p>Import all data from .csv file and trees are loaded.</p> <p>Then adds 2 players, Player1 = {200005, "Julian", 19, "Colombia", 123, 456, 789, 124, 125, 95}</p> <p>Player2 = { 200003, "Alejandro", 18, "Colombia", 123, 456, 789, 124, 125, 98}</p>
---------------	--------------------	--

Tests cases

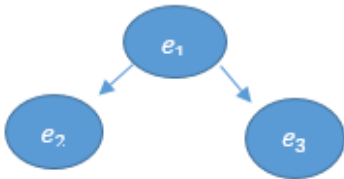
Test objective: check if program is searching, adding and deleting players correctly				
Class	Method	Setup	Input	Result
QuickBasketManager	binarySearch	SetupScenary1	To search player with key = 200005	Was found and returns true
QuickBasketManager	binarySearch	SetupScenary1	To search player with key = 200003	Was found and returns true
QuickBasketManager	deletePlayer	SetupScenary1	To delete player with key = 200005	Player is deleted

QuickBasketManager	binarySearch	SetupScenario1	To search player with key = 200005	Was not found and returns false
QuickBasketManager	deletePlayer	SetupScenario1	To delete player with key = 200003	Player is deleted
QuickBasketManager	binarySearch	SetupScenario1	To search player with key = 200003	Was not found and returns false

Test objective: check if program is resetting the data structures well (just with players list since make gets of the data structures it would not be a good practice)

Class	Method	Setup	Input	Result
QuickBasketManager	reset	SetupScenario1	None	The size of the list is now 0

Data structures ADT

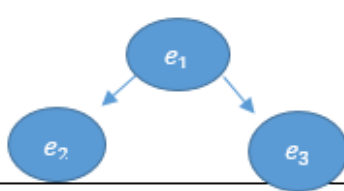
Binary Tree ADT		
$BST = \{e_1, e_2, e_3, \dots, e_n\}$ <p>e_1 is the main element, e_2 and e_3 are subtrees of e_1, any element less than e_1 goes to the left, and any element in the right is greater than e_1.</p> 		
<p>Inv: $\{e_1 > e_2, e_1 < e_3\}$ for any BST tree and sub tree, the elements less than an element are located on the left of this element and the greater ones on the right</p> <p style="text-align: center;">}</p>		
Primitive Operations:		
<ul style="list-style-type: none"> CreateBST put search getLowerThan getGreaterThan 	<ul style="list-style-type: none"> BST x Key x <u>Value</u> BST x Key BST x Key BST x Key 	<p>→ BST</p> <p>→ BST</p> <p>→ <u>Value</u></p> <p>→ <u>Value</u></p> <p>→ <u>Value</u></p>

createBST(): Constructor
“Create (Initialize) a new empty Binary search tree to add new elements”
<pre>{ pre: TRUE } { post: NewTree: The new created binary tree ready to add new elements }</pre>
put(K key,V value): Modifier
“insert a new key and a new element inside the binary tree, if the key already exists in the binary tree, insert the value(s) in the node that has the key”
<pre>{ pre: Binary Tree initialized } { post: Increments the depth of the branch with +1 in this specific sub-tree }</pre>

search(K key): Analyzer
“Search a specific key value inside the Binary Tree and returns the element(s) on it”
{ pre: Binary Tree initialized } (post: Return the value(s) of the searched key or return null if the key don't exists }

getLowerThan(K key): Analyzer
“search for keys lower than specified key and return the value(s) on them”
{ pre: Binary Tree initialized } { post: return the value(s) of the keys included in the specified range }

getGreaterThan(K key): Analyzer
“search for keys greater than specified key and return the value(s) on them”
{ pre: Binary Tree initialized } { post: return the value(s) of the keys included in the specified range }

AVL ADT		
$AVL = \{e_1, e_2, e_3 \dots e_n\}$ <p>e_1 is the main element, e_2 and e_3 are subtrees of e_1, any element less than e_1 goes to the left, and any element in the right is greater than e_1. The AVL tree element has a balance factor calculated by depth of its sub trees.</p> <div style="display: flex; align-items: center; justify-content: center;">  <div style="border: 1px solid black; padding: 10px; margin-left: 20px;"> $e_2 < e_1 < e_3$ </div> </div>		
<p>Balance factor: $\text{maxDepthRightSide} - \text{maxDepthLeftSide}$ Inv: $\{e_1 > e_2, e_1 < e_3\} \ \&\& \ BalanceFactor = 1 \text{ or } 0$ for any BST tree and sub tree, to the left of the element, elements are less than and to right of the element, elements are greater than. The height of the left branch can't be more than one unit than the right branch or viceversa.</p>		
Primitive Operations:		
<ul style="list-style-type: none"> • <u>CreateAVL</u> • <u>insert</u> • search • getLowerThan • getGreaterThan 	<ul style="list-style-type: none"> • AVL x Key x Value • AVL x Key • AVL x Key • AVL x Key 	<ul style="list-style-type: none"> → AVL → AVL → <u>Value</u> → <u>Value</u> → <u>Value</u>

createAVL(): Constructor
“Create (Initialize) a new empty AVL tree to add new elements”
<pre>{ pre: TRUE } { post: NewTree: The new created AVL tree ready to add new elements }</pre>

insert(K key,V value): Modifier
“insert a new key and a new element inside the binary tree, if the key already exists in the binary tree, insert the value(s) in the node that has the key”

<pre>{ pre: AVL tree initialized } { post: Increments the depth of the branch with +1 in this specific sub-tree }</pre>

search(K key): Analyzer

“Search a specific key value inside the AVL Tree and returns the element(s) on it”
--

<pre>{ pre: AVL tree initialized } (post: Return the value(s) of the searched key or return null if the key don't exists }</pre>

getLowerThan(K key): Analyzer

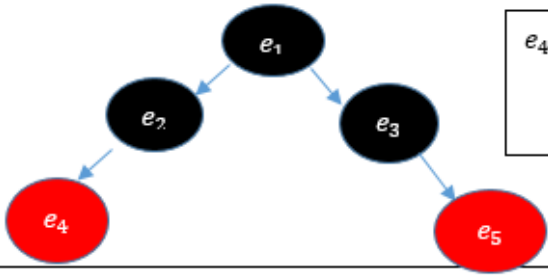
“search for keys lower than specified key and return the value(s) on them”
--

<pre>{ pre: AVL tree initialized } { post: return the value(s) of the keys included in the specified range }</pre>
--

getGreaterThan(K key): Analyzer

“search for keys greater than specified key and return the value(s) on them”
--

<pre>{ pre: AVL tree initialized } { post: return the value(s) of the keys included in the specified range }</pre>
--

ADT RBT		
$\text{RBT} = \{e_1, e_2, e_3, e_4, e_5 \dots e_n\}$ <p>e_1 is the main element, e_2 and e_3 are subtrees of e_1, any element less than e_1 goes to the left, and any element in the right is greater than e_1.</p> <p>The Black and Red tree elements has a balance factor calculated by depth of its sub trees.</p> <div style="display: flex; align-items: center; justify-content: space-around;">  <div style="border: 1px solid black; padding: 5px;"> $e_4 < e_2 < e_1 < e_3 < e_5$ </div> </div>		
<p>Balance factor: $\text{maxDepthRightSide} - \text{maxDepthLeftSide}$</p> <p>Inv: $\{[e_1 > e_2, e_1 < e_3] \ \&\& \ \{ \text{BalanceFactor} = 1 \text{ or } 0\} \ \&\& \ \{\text{Every red element has a black parent}\} \ \&\& \ \{\text{All paths from the root of the tree to a leaf contain exactly the same number of black elements}\}\}$.</p>		
Primitive Operations:		
<ul style="list-style-type: none"> • CreateRBT • insert • search • getLowerThan • getGreaterThan 	<ul style="list-style-type: none"> • RBT x Key x Value • RBT x Key • RBT x Key • RBT x Key 	<ul style="list-style-type: none"> → RBT → RBT → Value → Value → Value

createRBT(): Constructor
“Create (Initialize) a new empty Red Black tree to add new elements”
<pre>{ pre: TRUE } { post: NewTree: The new created RB tree ready to add new elements }</pre>

search(K key): Analyzer

“Search a specific key value inside the Red Black tree and returns the value(s) on it”

{ pre: Red Black tree initialized }
(post: Return the value(s) of the searched key or return null if the key don't exists }

insert(K key,V value): Modifier

“insert a new key and a new element inside the Red Black Tree, if the key already exists in the Red Black Tree, insert the value(s) in the node that has the key”

{ pre: Red Black Tree initialized }
{ post: Increments the depth of the branch with +1 in this specific sub-tree }

getLowerThan(K key): Analyzer

“search for keys lower than specified key and return the value(s) on them”

{ pre: Red Black tree initialized }
{ post: return the value(s) of the keys included in the specified range }

getGreaterThan(K key): Analyzer

“search for keys greater than specified key and return the value(s) on them”

{ pre: Red Black Tree initialized }
{ post: return the value(s) of the keys included in the specified range }

Post-mortem Report

In this second time that we worked, the work turned out to be much more efficient, each member worked in their GitHub branch and we divided the work evenly. At the beginning there was a problem because a colleague had created the repository with a different version of the java library than the others, so we had to create a new repository and the problem was fixed. In general terms, we enjoy this project more since, although faults were found, they could be solved in a timely and very fast way, we also determined that it is of great importance to make efficient programs, since our result was very good, extremely fast consultations, much less than half a second.