Jose Manuel Ossorio
Marco Antonio Pérez
Juan David Aguirre

Engineering method

Phase 1: Identifying the problem:

- Description of the problem context: a program which can perform different operations given a data set of different markets is needed. The software will be able to analyze big amounts of data and to do so in the most efficient way.
- Description of the problem: we need to build a program that can analyze different data sets with big amounts of information and output a result based on the requested calculation.
- Functional requirements:
    - R1: Consult the highest price of a share or foreign exchange market during a defined time period.
    - R2: Consult the lowest price of a share or foreign exchange market during a defined time period.
    - R3: Consult the period of time in which a share or foreign exchange market had the greatest growth.
    - R4: Display in a graphic the prices of up to 3 shares or foreign exchange markets, each of these must have a different colour.
    - R5: Consult which shares or foreign exchange markets surpass a value during a defined time period.
    - R6: Consult which 3 shares or foreign exchange markets display the greatest growth during a defined time period.
    - R7: Obtain data massively from text files.
    - R8: Add new data by direct input through the GUI
    - R9: Delete any data by direct usage of the GUI.
    - R10: Modify any data by direct input through the GUI
- Non-functional requirements:
    - The application must be time efficient
    - The application's GUI must be done using JavaFX
    - The application must use two different data structures

Phase 2: Compilation of necessary information:

- Financial market: A space whose objective is to channel the savings of families and companies to investment. It is ruled by the laws of offer and demand.
- Offer: The amount of goods and services that are willing to sell.
- Demand: The goods and services that are acquired by consumers on different prices.
- Financial system: Is the body of regulations, normative, tools, people and institutions that constitute the money market.
- Stock market: A type of capital market of those that operate around the world in which people negotiate the variable income and the fixed rent.

- Currency: Unit of change that facilitates the transfer of goods and services.

Phase 3: Creative solutions:

The problem to be solved has raised one complimentary issue, resulting from the possible quantity of the amount of data to be stored and analyzed which guides us towards the idea of using data structures to better utilize resources. Therefore we've identified the following possibilities related to the storage of information:

- Arrays: in this case, the information would be stored in ArrayLists, therefore, there would be no necessity to manually code any additional data structures.
- Linked Lists: in this data structure, one uses manually created Nodes which point the next one until there is no one to point to.
- Heaps: the idea of a normal Binary Tree but all data is stored in an array.
- Binary Search Tree: a data structure where any apparent node is smaller than it's right son and larger than it's left one.
- Red-Black Tree: same basic principles of a Binary Search Tree with the addition of coloring to each node, this creates a new subset of functionalities like rotations and fix-ups.
- AVL Tree: like the Red-Black, follows the principles of a Binary Search Tree adding rotations and fix ups. These trees do not implement colours but use a balancing factor, this means, the tree is capable of balancing itself after every operation is done.

Finally, the application must contain a visual representation of the information stored on the decided data structure. Therefore, tentatively we've selected the following libraries or functions:

- JavaFX
    - LineChart
    - Label
    - ListView
    - Button
    - HBox
    - VBox
    - ToggleButton
- JFreeChart
- JGraphT

Phase 4: Transition to preliminary design:

From a preliminary analysis of the optional possibilities of data structures for the storage of information, we've initially decided not to use arrays and Linked Lists. Both data structures are more prone than the others to generating time complexities of the order O(n) since, for example, to search the last position, one must go through every single element until the end. Now, for some other situations, these wouldn't be a big issue, but considering the nature of the application, in which we'd have to constantly get the greatest or lowest, no matter how the lists or arrays are organized, we'd have to do O(n) searches.

Afterwards, we noticed similar issues appeared when using Heaps for we could access to the lowest value or the highest value depending if it's a MinHeap or a MaxHeap but not both at a time. In order to obtain both, one would need to compare each of the values stored or use the Heapsort algorithm, which would mean a time complexity of O(n) in the former and the loss of the heap in the latter.

Therefore, we've decided to properly evaluate the following data structures:

- Binary Search Tree
- Red-Black Tree
- AVL Tree

Phase 5: Evaluating and selecting the best solution:

Criteria:

- Criterion A: satisfactory solution of the problem
  [1] The method does not solve the problem satisfactorily
  [2] The method partially solves the problem
  [3] The method solves the problem satisfactorily
- Criterion B: complexity of the implementation of the data structure
  [1] Very complex
  [2] Complex
  [3] Not very complex
  [4] Not complex
- Criterion C: complexity of the understanding of the data structure
  [1] Very complex
  [2] Complex
  [3] Not very complex
  [4] Not complex
- Criterion D: time efficiency
  [1] Not efficient
  [2] Not very efficient
  [3] Efficient
  [4] Very efficient
- Criterion E: space efficiency
  [1] Not efficient
  [2] Not very efficient
  [3] Efficient
  [4] Very efficient

Data Structure	Criterion A	Criterion B	Criterion C	Criterion D	Criterion E	Total
Binary Search Tree	3	3	4	3	1	14
Red-Black Tree	3	2	3	2	4	14
AVL Tree	3	2	4	3	4	16

Clearly the AVL tree is to be selected which seals one of the spots of our two non-functionally required data structures. These leads us to the comparison of Red-Black Trees and Binary Search Trees. One of the other non functional requirements is time efficiency, therefore, we should answer

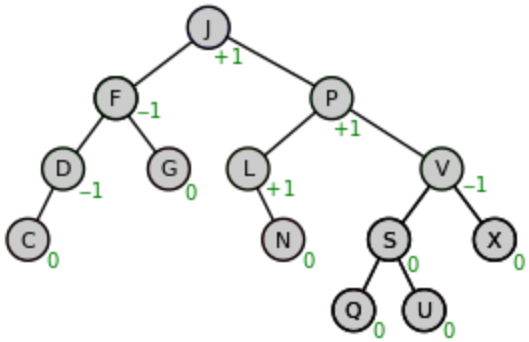
the following question: which is more time efficient? Binary Trees have one issue included in its worst case; imagine a tree whose root is the lowest or highest value possible and each following node is greater or lower than (respectively) it's parent, this would lead us to having what's basically a Linked List as there is no way for the tree to balance itself. On the other hand, while Red-Black Trees aren't fully balanced like AVLs, they are balanced to a certain degree thanks to its colour characteristic. Hence, the two data structures to be used are:

- Red-Black Tree
- AVL Tree

Furthermore, we've decided to use LineGraph from JavaFX as it allows us to save time on obtaining new libraries and meets our expectations for displaying the values required.

Phase 6: Preparation of reports and specifications:

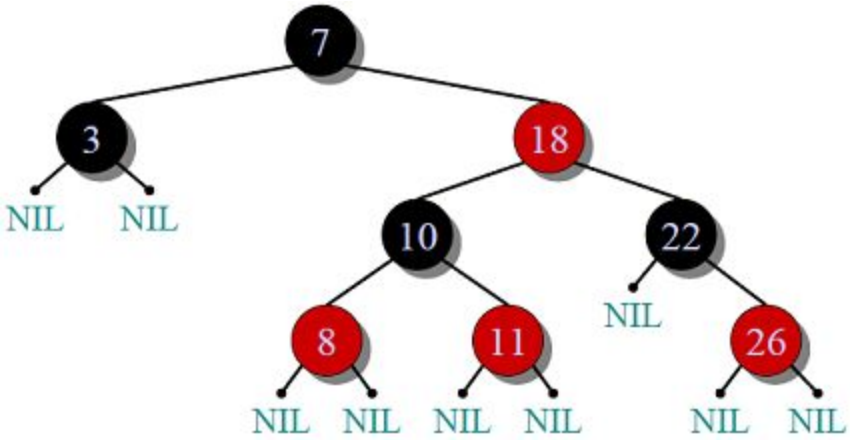
ADTs:

AVL Tree		
Every node has a left and right child associated. The first node of the tree is known as the root. Each node has a balancing factor.  Obtained from: https://en.wikipedia.org/wiki/AVL_tree		
Let N be a node belonging to an AVL Tree T • N.right >= N • N.left <= N • N.balancingFactor <= 2 ∧ N.balancingFactor >= -2 • If N.balancingFactor == -2 ∨ N.balancingFactor == 2 → fixUp		
AVLTree		AVLTree
insert	AVLTree x AVLNode	AVLTree
fixUp	AVLTree x AVLNode	AVLTree
deleteNode	AVLTree x RBNode	AVLNode
leftRotate	AVLTree x RBNode	AVLTree
rightRotate	AVLTree x RBNode	AVLTree
getMin	AVLTree	AVLNode

getMax	AVLTree	AVLNode
successor	RBTree x RBNode	RBNode
search	RBTree x RBNode	RBNode

Red-Black Tree

Every node has a left and right child associated. The first node of the tree is known as the root. Each node has a color property, which is either red or black.



Obtained from: https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

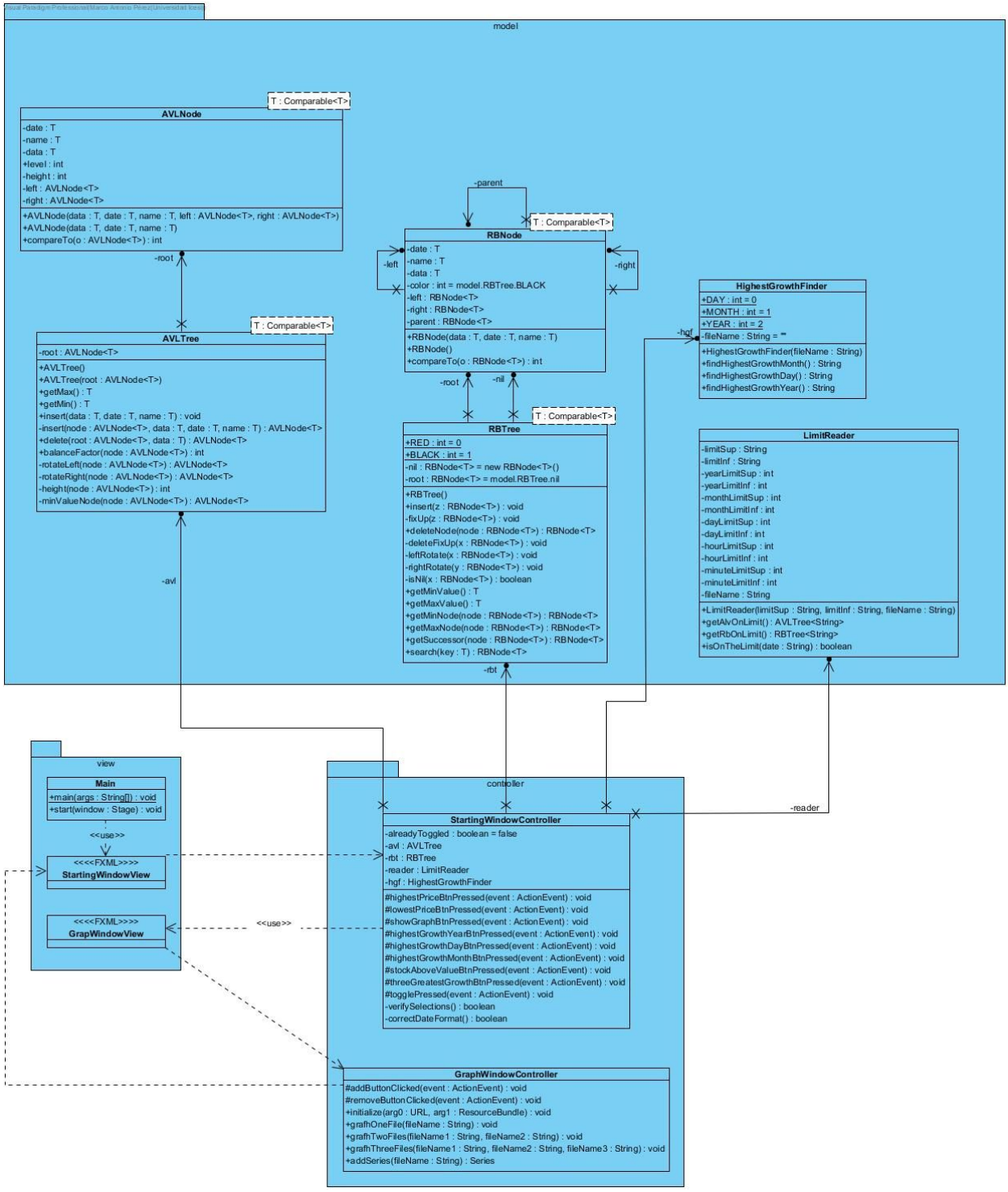
Let N be a node belonging to a Red-Black Tree T

- N.right >= N
- N.left <= N
- If N.color == RED →N.right.color = BLACK ∧ N.left.color = BLACK
- If T.root == N → N.color = BLACK
- The path from every node to its null descendants contains the same number of black nodes
    - N.right.blackHeight == N.left.blackHeight

RBTree		RBTree
insert	RBTree x RBNode	RBTree
fixUp	RBTree x RBNode	RBTree
deleteNode	RBTree x RBNode	RBNode
deleteFixUp	RBTree x RBNode	RBTree
leftRotate	RBTree x RBNode	RbTree
rightRotate	RBTree x RBNode	RBTree

getMin	RBTree	RBNode
getMax	RBTree	RBNode
successor	RBTree x RBNode	RBNode
search	RBTree x RBNode	RBNode

Class Diagram:


model
AVLNode
T : Comparable<T>
-date : T
-name : T
-data : T
+level : int
-height : int
-left : AVLNode<T>
-right : AVLNode<T>
+AVLNode(data : T, date : T, name : T, left : AVLNode<T>, right : AVLNode<T>)
+AVLNode(data : T, date : T, name : T)
+compareTo(o : AVLNode<T>) : int
-root
AVLTree
T : Comparable<T>
-root : AVLNode<T>
+AVLTree()
+AVLTree(root : AVLNode<T>)
+getMax() : T
+getMin() : T
+insert(data : T, date : T, name : T) : void
-insert(node : AVLNode<T>, data : T, date : T, name : T) : AVLNode<T>
+delete(root : AVLNode<T>, data : T) : AVLNode<T>
+balanceFactor(node : AVLNode<T>) : int
-rotateLeft(node : AVLNode<T>) : AVLNode<T>
-rotateRight(node : AVLNode<T>) : AVLNode<T>
-height(node : AVLNode<T>) : int
-minValueNode(node : AVLNode<T>) : AVLNode<T>
-avl
-parent
RBNode
T : Comparable<T>
-left
-right
-date : T
-name : T
-data : T
-color : int = model.RBTree.BLACK
-left : RBNode<T>
-right : RBNode<T>
-parent : RBNode<T>
+RBNode(data : T, date : T, name : T)
+RBNode()
+compareTo(o : RBNode<T>) : int
-root
-nil
RBTree
T : Comparable<T>
+RED : int = 0
+BLACK : int = 1
-nil : RBNode<T> = new RBNode<T>()
-root : RBNode<T> = model.RBTree.nil
+RBTree()
+insert(z : RBNode<T>) : void
-fixUp(z : RBNode<T>) : void
+deleteNode(node : RBNode<T>) : RBNode<T>
-deleteFixUp(x : RBNode<T>) : void
-leftRotate(x : RBNode<T>) : void
-rightRotate(y : RBNode<T>) : void
-isNil(x : RBNode<T>) : boolean
+getMinValue() : T
+getMaxValue() : T
+getMinNode(node : RBNode<T>) : RBNode<T>
+getMaxNode(node : RBNode<T>) : RBNode<T>
+getSuccessor(node : RBNode<T>) : RBNode<T>
+search(key : T) : RBNode<T>
-rbt
HighestGrowthFinder
+DAY : int = 0
+MONTH : int = 1
+YEAR : int = 2
-fileName : String = ""
+HighestGrowthFinder(fileName : String)
+findHighestGrowthMonth() : String
+findHighestGrowthDay() : String
+findHighestGrowthYear() : String
-hgf
LimitReader
-limitSup : String
-limitInf : String
-yearLimitSup : int
-yearLimitInf : int
-monthLimitSup : int
-monthLimitInf : int
-dayLimitSup : int
-dayLimitInf : int
-hourLimitSup : int
-hourLimitInf : int
-minuteLimitSup : int
-minuteLimitInf : int
-fileName : String
+LimitReader(limitSup : String, limitInf : String, fileName : String)
+getAlvOnLimit() : AVLTree<String>
+getRbOnLimit() : RBTree<String>
+isOnTheLimit(date : String) : boolean
-reader
view
Main
+main(args : String[]) : void
+start(window : Stage) : void
<<use>>
<<<<FXML>>>>
StartingWindowView
<<<<FXML>>>>
GrapWindowView
<<use>>
controller
StartingWindowController
-alreadyToggled : boolean = false
-avl : AVLTree
-rbt : RBTree
-reader : LimitReader
-hgf : HighestGrowthFinder
#highestPriceBtnPressed(event : ActionEvent) : void
#lowestPriceBtnPressed(event : ActionEvent) : void
#showGraphBtnPressed(event : ActionEvent) : void
#highestGrowthYearBtnPressed(event : ActionEvent) : void
#highestGrowthDayBtnPressed(event : ActionEvent) : void
#highestGrowthMonthBtnPressed(event : ActionEvent) : void
#stockAboveValueBtnPressed(event : ActionEvent) : void
#threeGreatestGrowthBtnPressed(event : ActionEvent) : void
#togglePressed(event : ActionEvent) : void
-verifySelections() : boolean
-correctDateFormat() : boolean
GraphWindowController
#addButtonClicked(event : ActionEvent) : void
#removeButtonClicked(event : ActionEvent) : void
+initialize(arg0 : URL, arg1 : ResourceBundle) : void
+grafhOneFile(fileName : String) : void
+grafhTwoFiles(fileName1 : String, fileName2 : String) : void
+grafhThreeFiles(fileName1 : String, fileName2 : String, fileName3 : String) : void
+addSeries(fileName : String) : Series

Test Cases:

For the AVL Tree

TestCase	1
Objective	Testing the constructor method of the class
Scene	A one-node-only tree
Input	new AVLNode<Integer>(new Integer(5), null, null)
Output	assertEquals() between 5 and the value related to the tree's root data. If the value returned is true, the creation is correct, if not, it isn't.

TestCase	2
Objective	Testing the insertion method
Scene	A tree containing three nodes
Input	avl.insert(8, null, null) avl.insert(6, null, null)
Output	assertEqualsTrue to the comparison between the data values expected in each position and those that are actually in the tree. If correct, each test should return true.

TestCase	3
Objective	Testing the insertion, fix-up and rotation methods. This is done by getting inserting nodes which will in the end necessitate fixing up due to unbalancing the tree
Scene	A tree containing six nodes
Input	avl.insert(8, null, null) avl.insert(6, null, null) avl.insert(10, null, null) avl.insert(4, null, null) avl.insert(3, null, null)
Output	assertEqualsTrue to the comparison between the data values expected in each position and those that are actually in the tree. If correct, each test should return true.

TestCase	4
Objective	Testing the deletion method. This is done by eliminating a node which then requires the movement of other ones.
Scene	A tree containing six nodes
Input	avl.insert(8, null, null) avl.insert(6, null, null) avl.insert(10, null, null) avl.insert(4, null, null) avl.insert(3, null, null) avl.insert(9, null, null) avl.delete(avl.getRoot(), 9)
Output	assertTrue to the comparison between the data values expected in each position and those that are actually in the tree. If correct, each test should return true.

For the Red-Black Tree

TestCase	1
Objective	Testing the constructor method of the class and insertion
Scene	A one-node-only tree
Input	new RBNode<Integer>(11, null, null)
Output	assertEquals() between 11 and the value related to the tree's root data. If the value returned is true, the creation is correct, if not, it isn't.

TestCase	2
Objective	Testing the insertion method
Scene	A tree containing three nodes
Input	rb.insert(new RBNode<Integer>(14, null, null)) rb.insert(new RBNode<Integer>(2, null, null))
Output	assertEqualsTrue to the comparison between the data values expected in each position and those that are actually in the tree. If correct, each test should return true.

TestCase	3
Objective	Testing the coloring, fix-up and rotation methods. This is done by inserting nodes which will in the end necessitate fixing up due to not satisfying the characteristics of a red-black tree
Scene	A tree containing six nodes
Input	rb.insert(new RBNode<Integer>(15, null, null)) rb.insert(new RBNode<Integer>(1, null, null)) rb.insert(new RBNode<Integer>(7, null, null)) rb.insert(new RBNode<Integer>(5, null, null)) rb.insert(new RBNode<Integer>(8, null, null))
Output	assertEqualsTrue to the comparison between the coloring values expected in each position and those that are actually in the tree. If correct, each test should return true.

TestCase	4
Objective	Testing the deletion and search method. This is done by eliminating a node which then requires the movement and recoloring of other ones.
Scene	A tree containing six nodes
Input	rb.insert(new RBNode<Integer>(15, null, null)) rb.insert(new RBNode<Integer>(1, null, null)) rb.insert(new RBNode<Integer>(7, null, null)) rb.insert(new RBNode<Integer>(5, null, null)) rb.insert(new RBNode<Integer>(8, null, null)) rb.search(7)
Output	assertEquals to the comparison between the color values expected in each position and those that are actually in the tree. If correct, each test should return true. assertNull in the value of the deleted position. This should be null as every deleted node shouñd turn into nil after the fix-ups.