NATIONAL RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS

Faculty of Computer Science
Bachelor's Programme "Data Science and Business Analytics"

Software Project Report on the Topic:

Data Structure Visualization

Fulfilled by:

Student of the group БПАД211
Spirina Mayya Alexandrovna

(signature) 07.06.2023 (date)

Assessed by the Project Supervisor:

Trushin Dmitrii Vitalievich
Associate Professor
Faculty of Computer Science, HSE University

(signature) 07.06.2023 (date)

Moscow 2023

Contents

Annotation

This paper describes a Qt application with an intuitive graphical interface, which is used to track step-by-step changes in AVL-tree data structure when add or delete operation is called. In addition, task of the application is to allow user get acquainted with the process of the node search by value and to study traversal types. The implementation is based on MVC concept, the usage of which helps to separate visualization from business logic. To establish links between MVC objects, it was decided to use a behavioral design pattern “Observer”.

Аннотация

В данной работе описывается Qt приложение, задачей которого является создание интуитивно понятного графического интерфейса для отслеживания пошаговых изменений в структуре АВЛ-дерева при выполнении операций добавления и удаления. Кроме того, приложение позволяет ознакомиться с тем, как происходит поиск узла дерева по ключу, и изучить виды обхода. В основе имплементации лежит концепция MVC, с помощью которой бизнес-логика отделена от визуализации. Для установления связей между MVC объектами было решено использовать поведенческий шаблон проектирования «Наблюдатель».

Keywords

AVL tree, balanced binary search tree, MVC, observer pattern, visualisation, GUI, widget based application.

1 Basic terms and definitions

Model – application object, which is responsible for all operations. In case of this project AVL Tree class itself is a model.

View – screen representation of the object, is able to visualize data stored in the model.

Controller – determines how user interface will react on the input of the user. Receives signals from view and determines which model methods should be called.

MVC (Model-View-Controller) – helps to build user interface connecting Model, View and Controller via two types of connections (direct and indirect), which are determined by observer pattern in the following way Picture1: Direst associations are shown by solid lines and mean that controller knows about model and view, view knows about model. As model has no direct knowledge about controller and view, the code can be reused. All indirect association are established with the usage of observer pattern.

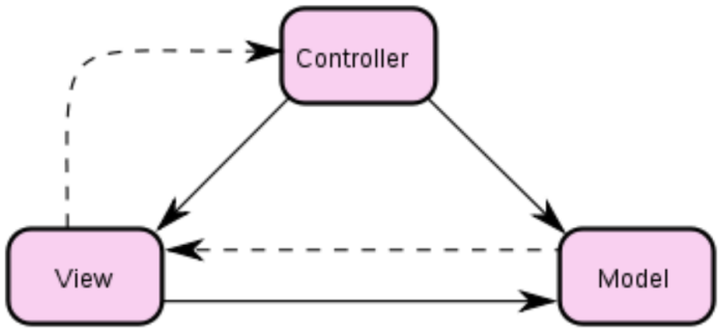

Figure 1.1: Connections between MVC components.

Observer pattern – behavioral design pattern, which is used to establish one-to-many relationship between objects, it is responsible for transfer of data. It creates safe connections, which guarantee notification of objects and deletion of connections in case observer or observable is deleted. Observer pattern allows to notify observers about changes in subject.

Subject (Observable) – an object, which notifies about its changes all objects (observers) that are subscribed to it.

Observer – objects that are notified about any changes made in the subject.

GUI – graphical user interface, which allows user to interact with the program via graphical components.

Self-balancing binary search tree (self-balancing BST) – binary search tree that automatically changes height to make it as small as possible.

Node of the tree – entity of a tree, which contains information about the value, children and height of the tree at this node. Tree consists of a collection of nodes.

2 Introduction

2.1 AVL Tree

All people, who are trying to study programming languages, sooner or later understand that they need to get familiar with the tree data structures. However, it can be a complicated task to capture how complex data structures work, especially when person encounters them for a first time. Visualization is a useful tool, which can make study of data structures more comprehensive.

This application presents the work of AVL Tree - a self-balancing Binary Search Tree, which offers O(log(n)) complexity of search, insertion and deletion operations, where n is the number of nodes. This efficient data structure was proposed in 1968 by two Soviet mathematicians Georgy Maximovich Adelson-Velsky and Evgenii Mikhailovich Landis. Because of self-balancing and almost never degenerative structure, AVL Tree offers effective way to store data. To elaborate on what AVL tree is its features should be mentioned:

1) Key in the node is smaller than any key in the right subtree and greater than any key in the left subtree (feature of a Binary Search Tree).

2) Difference between heights of right and left subtrees(also known as balance factor) is either -1, 0 or 1.

To insert node first thing to do is to find appropriate leaf node to which insertion should be applied. After that, tree may become unbalanced, hence, to restore balance criteria, tree should be reconstructed using rotation operations. Deletion operation is similar, difference is that node to delete is not obligatory leaf node, thus, it should be interchanged with inorder successor first. Full algorithm can be found in "Data Structures using c++ book" [4].

2.2 Tasks and implementation details

Implementation of application is based on MVC and observer pattern, where observable and observer are fields in a model class and view class, respectively. Interchange of data is established in the following way: Controller receives information from the View object and calls respective methods of model (addition, deletion or search). These methods change state of the model and notify about that view through observer and observable fields. Via these fields data is transferred from subject to subscribed observer, which draws updated AVL Tree.

It was decided to create data structure visualizing application to acquire deeper knowledge in application creation and get familiar with one of the ways how Qt library can be used to visualize data structures.

The goal of this project was to learn how to create interactive graphical applications in c++ programming language and implement such application by fulfilling following tasks:

- Implement AVL-Tree data structure
- Examine observer pattern and MVC
- Examine Qt framework for graphical visualization
- Implement interactive application for data structure visualization
- Write supporting documentation

2.3 Results

As a result, Qt application for AVl tree visualisation was created, which can be used by students during their studies and make the process of information perception easier and faster. Also, application offers functionality distinct from what can be found via the Internet(more complete comparison can be found in the following chapter).

3 Comparative analysis of analogues and methods

3.1 Comparative analysis of analogues

There are several analogues of application, which also provide step-by-step execution of operations([1] and [2]). They have distinctive functionality that this application does not offer:

1 Wider range of data structures visualised. In addition to AVL tree users can get familiar with simple binary search tree and compare their work.

2 Some of the applications allow to pause visualisation at each step.

However, some useful features are not present in other available applications or only present in binary search tree visualizing application(not AVL tree):

1 In other applications nodes can only be deleted by value with the usage of text edit field and button. In addition to such function, this application allows to choose node for search or deletion by clicking on its graphical representation.

2 Other applications do not introduce function to load file with the text, which consists of the values in order they should be inserted into the tree, and see ready AVL tree without having to input values one by one.

3 Important non-functional feature is that those applications, for which code is provided in open source, do not use MVC architectural pattern, thus, it may be more complicated to introduce additional structures(such as binary search tree, red-black tree, B-tree etc).

3.2 Observer patter

In this project it was decided to use observer pattern to establish connections between model and view, view and controller. Alternatively, Qt ecosystem could be used, however, in this case, core of the program(model) would have been dependent on Qt libraries, complicating process of model's update and making it more difficult to change visualisation tools. With regard to implementation of the observer pattern, classical model [7], should be considered. In this book it is suggested to implement model as an observable object and view as an observer. However, such approach implies that model and view should be fixed in memory and a lot of inheritance is used, as classes should be inherited from observer and observable base classes. It may be inconvenient to implement many interfaces, thus, approach, in which observer and observable are fields in the view and model, was used.

4 Implementation details

4.1 Functional requirements

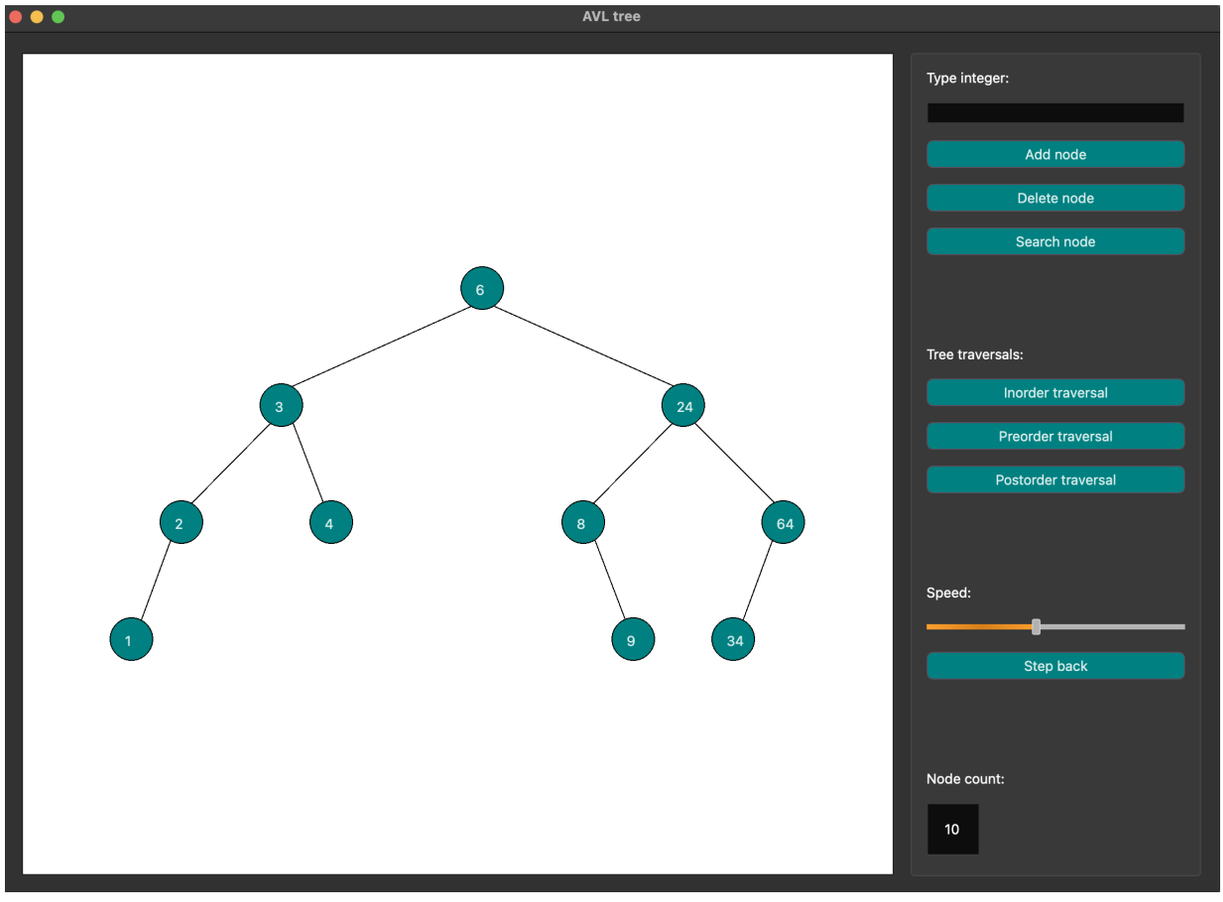

Figure 4.1: Application interface.

- User is able to insert, select and delete node with integer value as a key by inputting number in the text edit field and pressing respective button. All operations are shown in a step-by-step manner.
- Alternative option of node selection is to click on a graphical representation of a node. After that value of a selected node will appear in a text edit field and respective button can be pressed, as in the previous point.
- User is notified in case if value is already present in the tree(for insertion) or not present(for deletion and selection), via the message box.
- Inorder, preorder and postorder traversals are visualised by highlighting nodes in the respective order. To begin these operations, user should press a button.
- User is able to choose speed of visualisation with a usage of a slider.

- Step back, that is cancel previous addition or deletion, can be made.
- Statistics on the number of nodes presented is available.
- In the menu at the top of the application user can press on a load button, open a .txt file with numbers placed in desired order of insertion and separated with spaces. Numbers from file will be transformed into a tree.

4.2 Non-functional requirements

- Application is written in c++ language and represents Qt project in Clion. To get familiar with features specific for Qt official documentation was used [6]
- Main window is scalable, visualisation can be scrolled to see all the parts of the tree.
- For visualisation Qt QGraphicsScene Class is used.
- To create unified style of buttons and fields .qss file is included in the project.
- Version control is supported with the usage of github.
- Identical code style is done with the usage of .clang-format file.
- Final documentation is done with the help of LaTeX.
- Operation system fails are not supported by the application.

4.3 Class overview

1 Application. Class is used to establish observer pattern connections between Model and View, View and Controller. For that function *subscribe*, which creates safe connections between objects, of observer library is used.

2 Model. Model is the core of the application, it is independent from Qt ecosystem and includes implementation of AVL Tree data structure. The only thing that is different from simple tree is that View is notified about any changes in the tree structure via the observer-observable connection. *notify* method is called inside model, which informs view about status of data to be sent(struct Data)4.3.

3 View. View is responsible for all interactions with GUI, including creation of the interface. When user presses a button, signal is sent to a respective slot and data preparation process

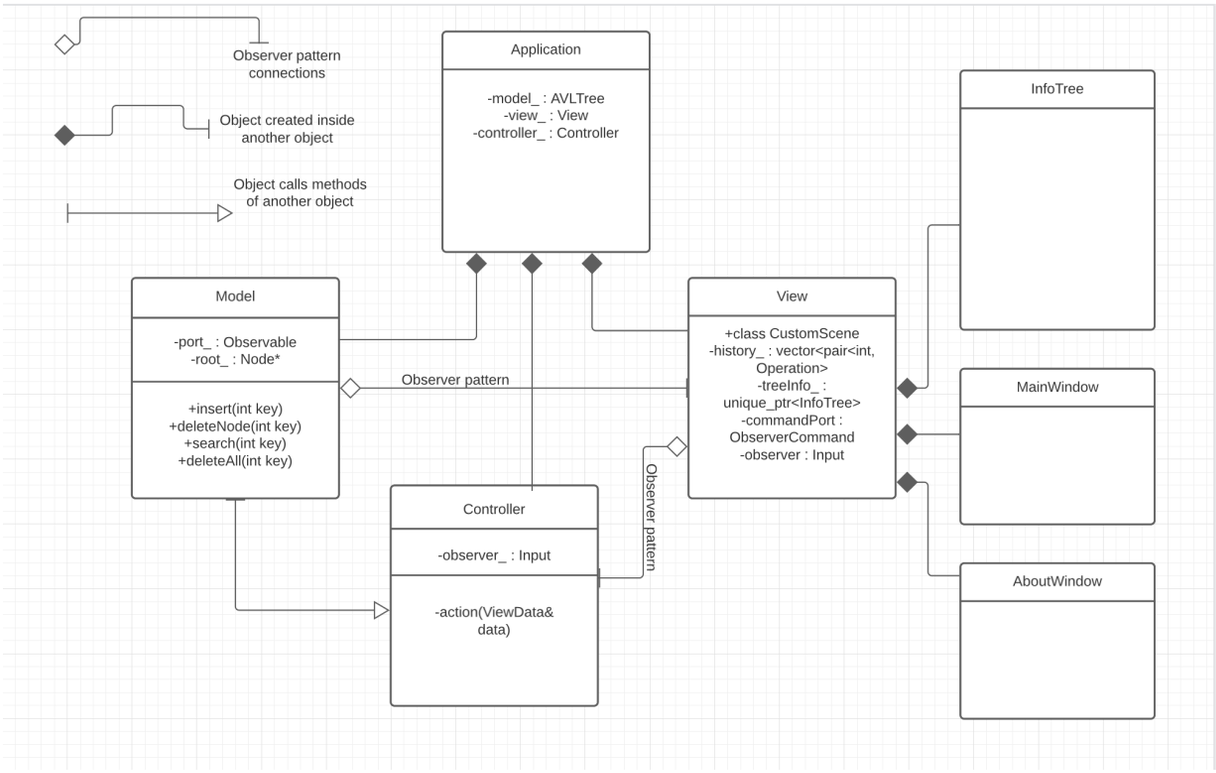

Figure 4.2: Class overview.

is initiated. Preparation is dependent on the type of operation done:

Traversals. These operations do not affect structure of the tree, hence, from beginning to the end they are done inside View, by highlighting nodes in order prescribed by the traversal. Between each step function *drawTree* is called with delay.

Addition, deletion, search. View collects all necessary data from the interface into struct Command, that is: value typed by the user and type of operation chosen, desired speed of visualisation is, also, set inside these functions.

```
Model.h
enum Operation{
        Add, Delete, Search, Traversal, DeleteAll
};
enum Message{
        Fine, NotPresent, AlreadyPresent
};
struct Data{
        Node*& value;
        Message& message;
        Operation& operation;
        int& passing_;
};
```

```
View.h
enum Operation{
        Add, Delete, Search, DeleteAll
};
struct Command{
        int& value;
        Operation& operation;
};
```

Figure 4.3: Structures to send via observer pattern connections.

Interaction with nodes by click. Calls method *findValue* of InfoTree, which returns value of a node and sets it in text edit. *findValue* goes through all nodes of the tree and checks whether coordinates of a clicked point lie inside coordinates of a node. More precisely

described in InfoTree class.

Load function. First it sets operation to DeleteAll, as model should be cleared before any data is read from the file. After that, function reads values one by one, after each read initiates insertion operation.

Step back. To implement this function vector of pairs<value, operation type>, called history, is stored inside view. Each time modification is made(that is successful addition or deletion) new element is added to history. When *StepBack* button is pressed, the last element of history is deleted and program repeats the whole procedure from the first element stored inside vector to the last.

View is, also, responsible for visualisation itself and is the only object that can draw. Thus, it has all functionality necessary for that purpose. When view receives data from model, function *draw* is called automatically. *Draw* checks message that was sent by the model and creates message box to notify user about incorrect input (that is already present value or, on the contrary, not present). The same function recreates treeInfo to get new coordinates of the nodes and calls *drawTree* with delay specified by slider. *drawTree* goes through treeInfo and adds ellipse items to the scene.

To draw nodes it was decided to create customScene class inherited from QGraphicsScene and override mousePressEvent to get point of mouse event.

4 Controller. This class knows about model, hence, depending on the data received from View (operation type) it calls required methods of the model.

5 InfoTree. Class responsible for creation of help tree, which contains coordinates of the nodes required for visualisation purposes, width of current node in addition to data stored in AVL tree. Calculation of coordinates is done in the following way:

Y coordinates are calculated using level order traversal by the formula 1

$$Ynode = Yparent + 2 \cdot radius + height, \tag{1}$$

For X coordinates we first calculate width of each subtree, starting from leaf nodes and using postorder traversal.

Width of leaf nodes:

$$2 \cdot radius, \tag{2}$$

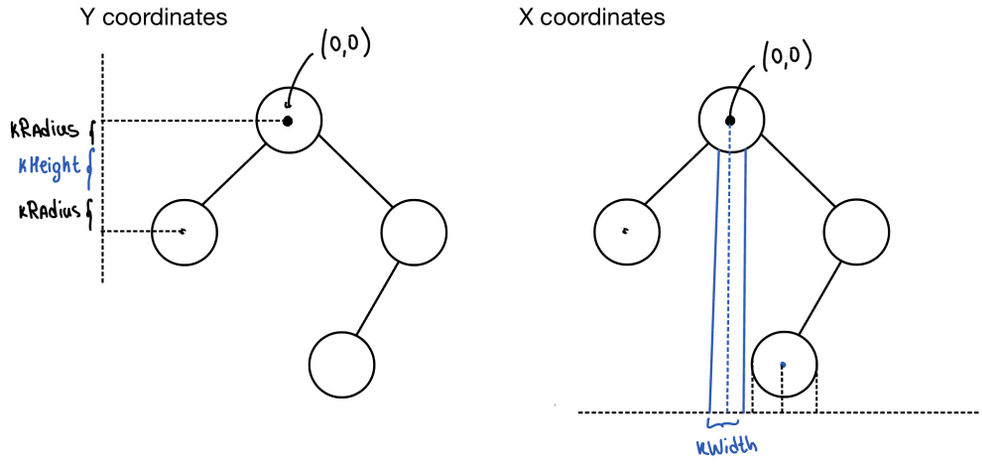

Figure 4.4: Coordinates calculation.

Width of nodes with one child:

$$2 \cdot widthChild + width, \tag{3}$$

Width of nodes with two children:

$$2 \cdot \max(widthLeftChild, widthRightChild) + width, \tag{4}$$

Knowing the width of each subtree x coordinates can be calculated using level order traversal for left child of a node 5 and for right child of a node 6.

$$XleftChild = Xnode - \frac{width}{2} - \frac{widthLeftChild}{2}, \tag{5}$$

$$XrightChild = Xnode - \frac{width}{2} - \frac{widthRightChild}{2}, \tag{6}$$

where width, height and radius are constants.

Another method in InfoTree is *findValue*. It receives coordinates of a point, which person clicked on and traverses through the tree, checking if point lies inside node. If point is indeed inside, function returns value of a node and bool value true, if point is not inside node, function returns 0 and bool value false.

6 MainWindow and AboutWindow. Actual interface, in which all graphical items are located.

4.4 Observer pattern

Before introducing how data is transformed into visual representation, it is important to mention how objects interact without having direct knowledge about each other. Due to the fact that in this project external observer library is used [5], focus will be on how it is used. To establish connections, *subscribe* function of observer is called inside Application class. From this point observable field of class can send data to its observers through the ports, when *notify* function is called. In this project there are 2 such ports. Data which is sent from View to Controller is described in the Command structure. Data sent from Model to View is different and described in Data structure, because Model has to share pointer to the tree, which should be visualised, error messages that occurred during the process and value of a node which should be highlighted(passing).

4.5 Data flow

In the following subsection it will be showed how data goes through the whole cycle, starting from user inputting information and ending at the point when visualisation changes.

1 Work of the application starts with the user interacting with GUI, either by pressing the button(signal goes to the respective slot in the view) or clicking on the node(mousePressEvent is called).

2 In View all preparations are done in the manner discussed in the View class description. For operations: add, delete, search and load, port is notified about the changes and Command structure is sent to Controller via observer port. For traversals points 3 and 4 are skipped.

3 Controller checks operation type and calls respective method of the Model.

4 Model stores the whole tree, so when any change is introduced, it uses observable-observer port to send pointer to a changed tree with some additional information to View.

5 At this point tree stored in the model is up-to-date, thus, it can be visualised. For that help tree has to be updated and only after that nodes are drawn. Important thing to mention is that delay is introduced to make visualisation process visible(otherwise it will be too fast to notice), time of delay is taken from the slider responsible for speed control.

6 From this point process can start from the beginning.

4.6 Complexity analysis and measurements

In total, size of the project is approximately 37 KB, excluding external libraries.

It was decided to conduct some test to see how much longer it takes to visualise tree rather than just insert it into the model. For that results of the same random test are shown, which illustrate time needed to insert one more node. For insertion to a model only, for first 1000 nodes time is less than 1 microsecond on average. However, time of insertion with visualisation, but without artificial delay is different.4.5:

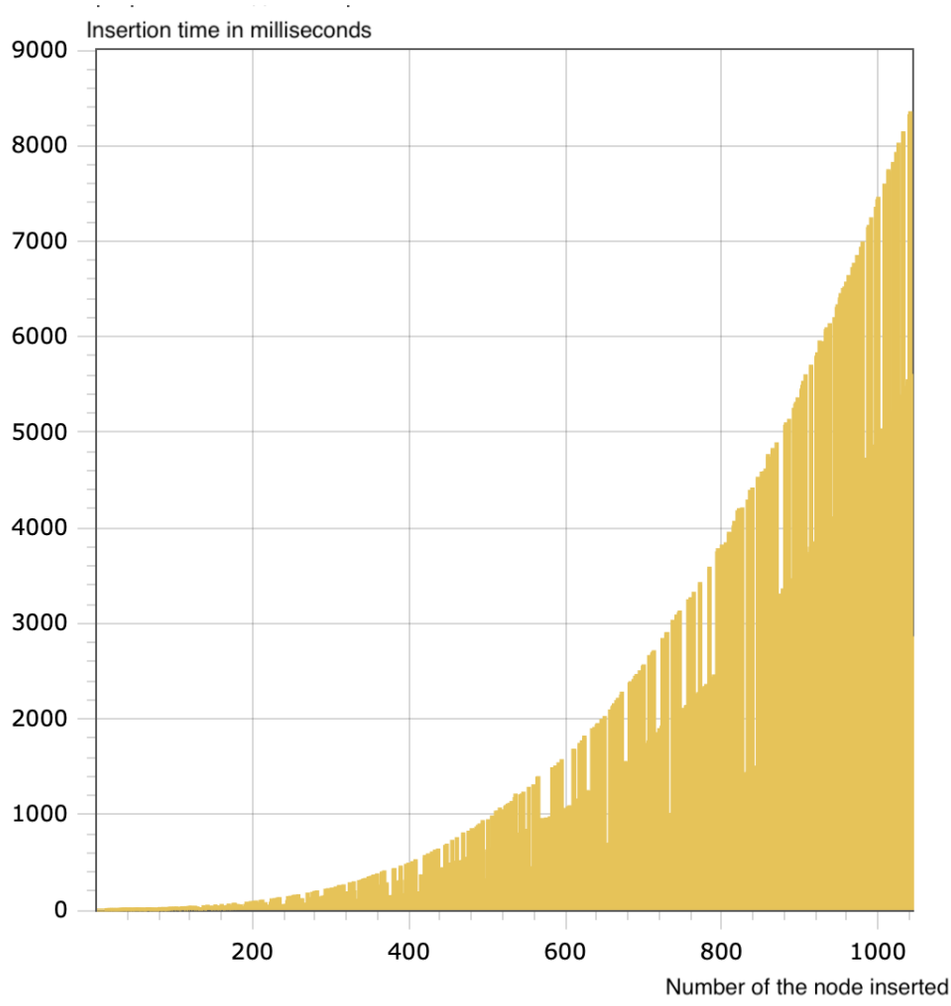

Figure 4.5: Measured time of insertion.

Complexities of the main operations are the following:

Addition operation: O(nlogn)

1 Read data from GUI and redirect operation to model - const

2 Insertion in AVL tree - O(logn)

- Build helpTree - O(n)
- Draw tree - O(n)

Search and deletion similar.

Load: O(n^2logn)

- Read data from file - O(n)
  - Add node - O(nlogn)

Traversal: O(n^2)

Speaking about memory consumption, most memory is consumed for the following purposes: inside model AVL tree is contained O(n), inside view instance of InfoTree is stored and recreated each time O(n), inside View history is stored. To ensure safe memory allocation and deallocation std::unique ptr was used for InfoTree instance in View.

5 Conclusion

In conclusion, application, which can be used to study AVL tree algorithms, was created[3]. The aim was to gain knowledge of how visualisation programs can be done, which was achieved. The whole program works correctly, however, it has some problems associated with the fact that the whole tree has to be rebuild even if no rotations have to be made, thus, the algorithm is quite slow when number of nodes is big. In perspective, more models (of different data structures) will be added and new functionality will be included, which application lacks at the moment compared to analogues, such as possibility to pause visualisation at certain steps.

References

[1] *AVL Tree*. URL: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html (visited on May 27, 2023).

[2] *AVL Tree*. URL: https://visualgo.net/en/bst?slide=1 (visited on May 27, 2023).

[3] *AVL Tree*. URL: https://github.com/spirinamayya/avltree/tree/master (visited on June 7, 2023).

[4] D.S. Malik. "Data structures using c++". In: Cengage Learning, 2010. Chap. 11.

[5] *Observer labrary*. URL: https://github.com/DimaTrushin/Library/tree/25926b670287f925089Observer (visited on May 27, 2023).

[6] *Qt documentation*. URL: https://doc.qt.io/ (visited on May 27, 2023).

[7] Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". In: Addison-Wesley, 1994. Chap. 5.