

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МОЭВМ**

**КУРСОВАЯ РАБОТА**  
**по дисциплине «Алгоритмы и структуры данных»**  
**Тема: AVL-деревья – вставка и исключение. Демонстрация**

Студент гр. 9382

\_\_\_\_\_

Субботин М.О.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

## **ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ**

Студент Субботин М.О.

Группа 9382

Тема работы: АВЛ-деревья – вставка и исключение. Демонстрация

Исходные данные:

Пользователь при помощи графического интерфейса задает АВЛ-дерево.

Содержание пояснительной записки:

“Содержание”, “Введение”, “Ход выполнения работы”, “Заключение”, “Список использованных источников”

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 31.10.2020

Дата сдачи реферата: 21.11.2020

Дата защиты реферата:

Студент

\_\_\_\_\_

Субботин М.О.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

## **АННОТАЦИЯ**

В курсовой работе представлена реализация графического интерфейса для АВЛ-дерева. Пользователь вводит значение элемента для вставки или удаления из дерева, а программа выводит на экран пошаговые действия алгоритмов. Программа выводит промежуточные результаты, как в текстовом формате, так и в графическом. Графический интерфейс написан в Qt Creator на языке C++.

## **SUMMARY**

The course work presents the implementation of the graphical interface for the AVL tree. The user enters the value of the element to be inserted or removed from the tree, and the program displays the step-by-step actions of the algorithms on the screen. The program displays intermediate results, both in text format and in graphic format. The graphical interface is written in Qt Creator in C++.

## СОДЕРЖАНИЕ

Введение	5
1. Описание структуры данных	6
2. Описание алгоритмов	8
2.1. Алгоритм поворота дерева	8
2.2. Алгоритм вставки элемента	10
2.3. Алгоритм удаления элемента	11
3. Интерфейс программы	12
3.1. Элементы	12
3.2. Класс DrawArea	13
3.3. Методы insertClicked() и deleteClicked()	13
4. Тестирование	14
Заключение	21
Список использованных источников	22
Приложение А. main.cpp	23
Приложение Б. AVL.h	23
Приложение В. drawarea.h	38
Приложение Г. drawarea.cpp	39
Приложение Д. mainwindow.h	43
Приложение Е. mainwindow.cpp	44

## **ВВЕДЕНИЕ**

Цель состоит в том, чтобы изучить работу такой структуры данных, как АВЛ-дерево. Реализовать два основных алгоритма – вставка и удаление элемента дерева. Также одной из задач является представление дерева в графическом виде.

Для решения поставленных задач была изучена структура дерева и соответствующие алгоритмы. Также были изучены ресурсы, которые предоставляли информацию о графическом отображении элементов в среде Qt Creator.

# ХОД ВЫПОЛНЕНИЯ РАБОТЫ

## 1. Описание структуры данных

АВЛ-дерево, это прежде всего бинарное дерево поиска, т.е. каждая ее вершина удовлетворяет следующему требованию: все вершины из ее левого поддерева меньше по величине, чем ее значение, а все вершины из правого поддерева больше. Помимо этого для вершин АВЛ-дерева должно выполняться еще одно условие: разность высот левого и правого поддерева текущей вершины должно находиться в диапазоне  $-1..1$ . Иначе над деревом следует провести перебалансировку ее вершин.

В программе эта структура данных представлена следующим образом:

### Класс вершины

```
template<typename Elem>
class Node
{
    friend class AVL<Elem>;
public:
    Node(const Elem &);
    Elem value;
    Node<Elem> *leftChild;
    Node<Elem> *rightChild;
    Node<Elem> *parent;

    int height;
    QColor color;
    int x;
};
```

Где `Node<Elem> *leftChild` – указатель на левого ребенка

`Node<Elem> *rightChild` – указатель на правого ребенка

`Node<Elem> *parent` – указатель на родителя

`int height` – высота вершины

`QColor color` – цвет вершины для отображения

`int x` – положение вершины по горизонтали

## И сам класс AVL-дерева

```
template<typename Elem>
class AVL
{
public:
    AVL();

    /*
     * Здесь находятся методы
     */

    QPainter *painter;
    int yspace;
    int xspace;
    int nodeRadius;
    double scale;
private:
    Node<Elem> *root;
};
```

Здесь расписаны только поля этого класса, методы будут обсуждаться позже.

`Node<Elem> *root` – указатель на корень дерева

`QPainter *painter` – указатель на класс для рисования элементов дерева

`int yspace` – отступ по вертикали между вершинами дерева (для отображения)

`int xspace` – отступ по горизонтали между вершинами дерева (для отображения)

`int nodeRadius` – радиус вершины (для отображения)

`double scale` – переменная отвечающая за масштабирование выводимого окна.

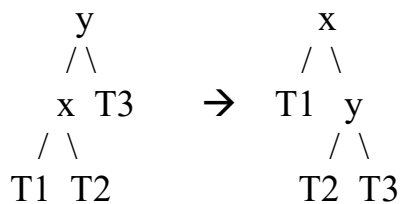
## 2. Описание алгоритмов

### 2.1. Алгоритм поворота дерева

При реализации алгоритма вставки и удаления элементов нам потребуется проводить перебалансировку вершин.

Следует определить два преобразования дерева: поворот направо и поворот налево.

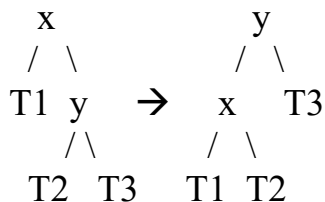
Поворот направо:



T1, T2, T3 – поддеревья.

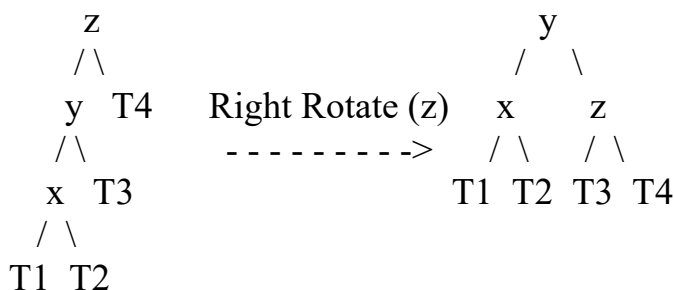
Мы буквально взяли, подвесили это дерево за вершину x, и немного передвинули поддерево T2.

Поворот налево:



Теперь же мы как будто подвесили дерево за вершину y и переместили T2.

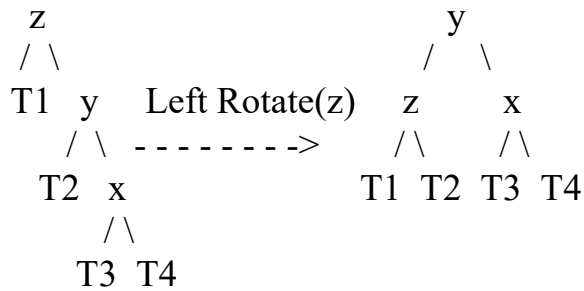
Если какая-то из вершин не сбалансированно, может получиться 4 случая.  
Left Left:



В этом случае вершина z имеет критерий равный 2, и она не сбалансирована. Для того, чтобы сбалансировать эту вершину, следует повернуть дерево направо в этой вершине.

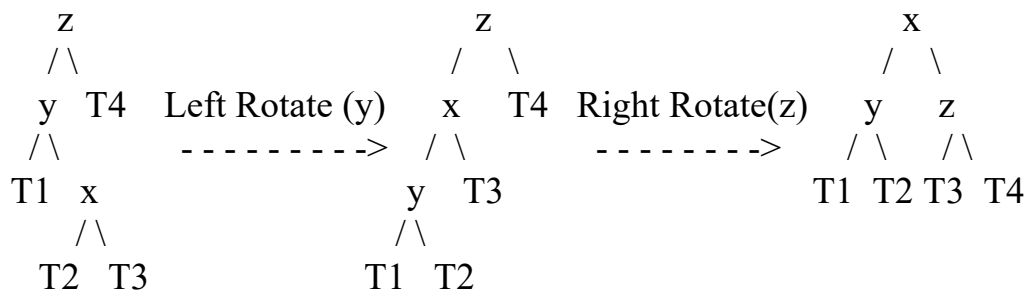


Right Right:



Здесь такая же ситуация с вершиной z, только теперь ее критерий равен -2. Для того, чтобы сбалансировать в таком случае, следует повернуть дерево в этой вершине налево.

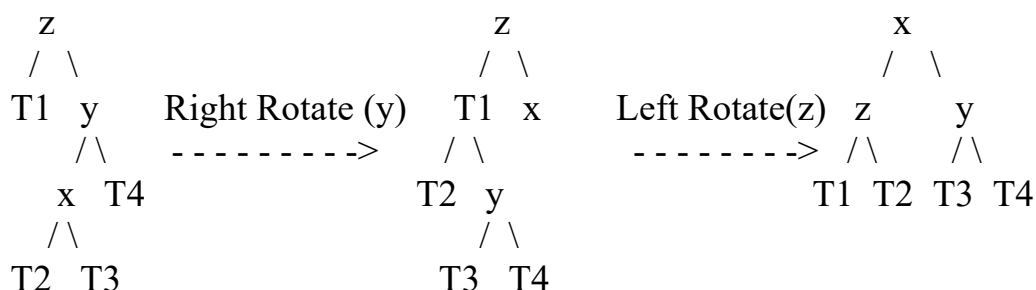
Left Right:



z имеет критерий баланса равный 2, так же как и в случае LL, но как тогда их различать? Дело в том, что дерево может стать не сбалансированным только после вставки элемента. В этом случае x это и есть тот самый вставляемый элемент. Если он находится слева от левого ребенка вершины z, то это случай LL, если справа, то LR.

Случай LR решается с помощью двух поворотов, сначала левый поворот для вершины y, затем правый для вершины z.

Right Left:



Случай RL решается также с помощью двух поворотов: правый для вершины y и левый для вершины z.

В коде используется метод

```
Node<Elem>* rebalance(Node<Elem> * node, const Elem &item,  
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages);
```

Который производит перебалансировку заданной вершины.

Node<Elem> \*node – заданная вершина

Const Elem &item – элемент над которым производили действие(вставка/удаление)

std::vector<AVL<Elem>> &vectorAVL – переменная для сохранения состояний дерева для последующего их вывода в графический интерфейс.

std::vector<std::string> &messages – переменная для сохранения словесного описания производимых действий.

## 2.2. Алгоритм вставки элемента

Т.к. AVL-дерево — это бинарное дерево поиска, то и вставка здесь будет очень похожа на вставку для бинарного дерева поиска. Проходимся по дереву рекурсивно, если значение текущей вершины больше, чем вставляемое значение, то идем к левому ребенку, если меньше, то к правому, а если равное, то значит нам нечего вставлять. Как только достигнем null-вершины, вставляем на ее место новую вершину с заданным значением. Отличается этот алгоритм от алгоритма для простого бинарного дерева поиска только тем, что возвращаясь обратно по рекурсии мы должны проверить не сломался ли баланс вершин после вставки, и также проверить высоту вершин. Как только встречаем вершину с неправильным балансом — делаем для нее перебалансировку.

В коде за вставку элемента отвечает метод

```
Node<Elem>* AVL<Elem>::insertNode(Node<Elem>* node, const Elem &item,  
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages)
```

Node<Elem>\* node – текущая вершина

Const Elem &item – значение для вставки

std::vector<AVL<Elem>> &vectorAVL - переменная для сохранения состояний дерева для последующего их вывода в графический интерфейс.

std::vector<std::string> &messages – переменная для сохранения словесного описания производимых действий.

### 2.3. Алгоритм удаления элемента

Находим вершину которую надо удалить. Т.е. двигаемся по вершинам от корня, смотрим на значение текущей вершины, если оно больше чем, значением удаляемой вершины идем к левому ребенку, если меньше, то к правому, если равно, то мы нашли вершину. Если в какой-то момент наткнемся на null-вершину это означает, что в дереве нет такой вершины.

Если удаляемая вершина не имеет детей, то ее можно просто удалить.

Если удаляемая вершина имеет 1 ребенка, тогда надо переместить вершину ребенка в удаляемую(значение, указатели на детей и.т.д.)

Если удаляемая вершина имеет 2 детей, то надо удалить самую меньшую(левую) вершину из правого поддерева удаляемой вершины и значение этой меньшей вершины присвоить удаляемой вершине.

После всех действий с удалением следует провести перебалансировку вершин так же, как и при вставке.

В коде за удаление элемента отвечает метод

```
Node<Elem>* AVL<Elem>::deleteNode(Node<Elem>* node, const Elem &item,  
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages)
```

Node<Elem>\* node – текущая вершина

Const Elem &item – значение для удаления

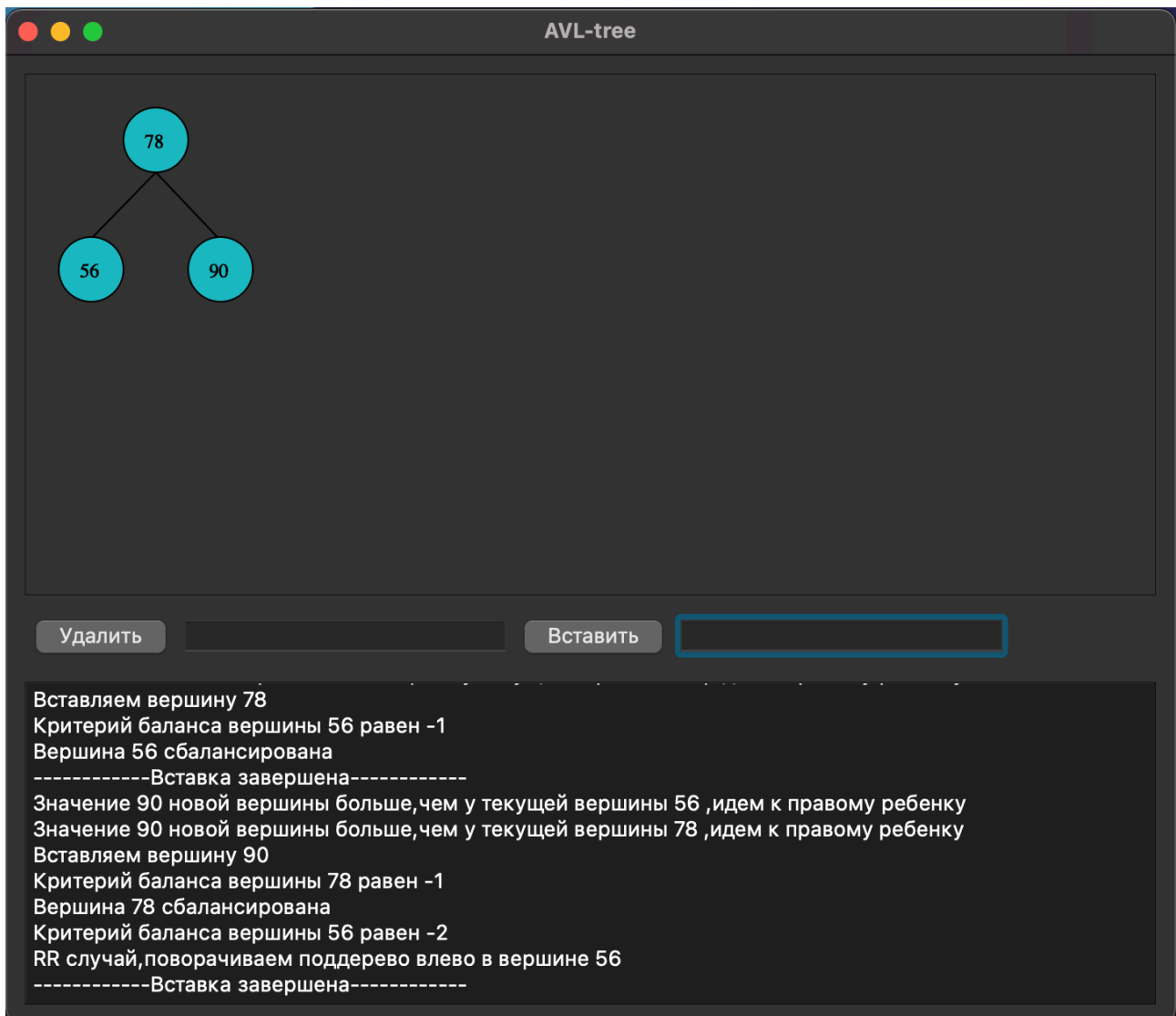
std::vector<AVL<Elem>> &vectorAVL - переменная для сохранения состояний дерева для последующего их вывода в графический интерфейс.

std::vector<std::string> &messages – переменная для сохранения словесного описания производимых действий.

### 3. Интерфейс программы

#### 3.1. Элементы

В интерфейсе программы существуют две кнопки, два поля для ввода значений, пространство для вывода рисунка и поле для отображения информации в текстовой форме.



Кнопки являются экземплярами класса `QPushButton`. Реализована кнопка, отвечающая за вставку элемента и кнопка для удаления элемента. Значения, которые надо вставить или удалить вводятся в поля класса `QLineEdit`. Сигнал нажатия на кнопки `clicked()` связывается с методом-слотом `insertClicked()` и `deleteClicked()` для кнопок вставки и удаления соответственно.

Следовательно по нажатию на кнопку выполняется соответствующий ей метод. В свою очередь экземпляры класса QLineEdit тоже связаны с этими методами, т.к. они передают информацию о том, какой элемент дерева надо вставить/удалить.

Пространство для вывода текстовой информации о действиях является экземпляром класса QTextEdit.

Пространство для вывода графической информации является экземпляром класса QScrollArea. Туда добавляется виджет, класс наследуемый от класса QWidget, чтобы производить отрисовку дерева.

### **3.2. Класс DrawArea**

Класс DrawArea наследуется от класса QWidget и имеет метод, который применяется для рисования дерева.

Он имеет три важных метода:  
void DrawArea::paintEvent(QPaintEvent \*) – метод, который создает необходимые экземпляры классов для рисования и вызывает метод draw(), для клиента этот метод служит, как метод для рисования.

void DrawArea::draw(QPainter \*painter, double &scale) – метод, настраивающий параметры для рисования(цвет, размер, и.т.д)

void DrawArea::recursiveDraw(Node<int> \*node) – метод, который рекурсивно проходит по дереву(Left-Root-Right) и непосредственно рисует элементы.

### **3.3. Методы insertClicked() и deleteClicked()**

При нажатии на кнопки вызываются методы insertClicked() и deleteClicked(). Они берут информацию из полей QLineEdit и вызывают соответствующий метод для дерева по удалению или вставки элемента. От этих методов получают состояния дерева на каждом шаге и сообщения об этих шагах. Затем выводят эти состояния на экран с некоторой задержкой, чтобы можно было уследить за этими изменениями.

# 4. Тестирование

## 1. Добавление элемента в пустое дерево

AVL-tree

Удалить

Вставить

AVL-tree

2

Удалить

Вставить

Вставляем вершину 2  
-----Вставка завершена-----

## 2. Добавление элемента в уже существующее дерево

AVL-tree

14

10

17

Удалить

Вставить

Вставляем вершину 14  
Критерий баланса вершины 10 равен -1  
Вершина 10 сбалансирована  
-----Вставка завершена-----  
Значение 17 новой вершины больше, чем у текущей вершины 10, идем к правому ребенку  
Значение 17 новой вершины больше, чем у текущей вершины 14, идем к правому ребенку  
Вставляем вершину 17  
Критерий баланса вершины 14 равен -1  
Вершина 14 сбалансирована  
Критерий баланса вершины 10 равен -2  
RR случай, поворачиваем поддерево влево в вершине 10  
-----Вставка завершена-----

AVL-tree

14

10

17

16

Удалить

Вставить

Вершина 14 сбалансирована  
Критерий баланса вершины 10 равен -2  
RR случай, поворачиваем поддерево влево в вершине 10  
-----Вставка завершена-----  
Значение 16 новой вершины больше, чем у текущей вершины 14, идем к правому ребенку  
Значение 16 новой вершины меньше, чем у текущей вершины 17, идем к левому ребенку  
Вставляем вершину 16  
Критерий баланса вершины 17 равен 1  
Вершина 17 сбалансирована  
Критерий баланса вершины 14 равен -1  
Вершина 14 сбалансирована  
-----Вставка завершена-----

14

### 3. Добавление элемент с перебалансировкой для случая RR

AVL-tree

```
graph TD; 14((14)) --> 15((15));
```

Удалить  Вставить

Вставляем вершину 14  
-----Вставка завершена-----  
Значение 15 новой вершины больше, чем у текущей вершины 14, идем к правому ребенку  
Вставляем вершину 15  
Критерий баланса вершины 14 равен -1  
Вершина 14 сбалансирована  
-----Вставка завершена-----

AVL-tree

```
graph TD; 15((15)) --> 14((14)); 15((15)) --> 16((16));
```

Удалить  Вставить

Вставляем вершину 15  
Критерий баланса вершины 14 равен -1  
Вершина 14 сбалансирована  
-----Вставка завершена-----  
Значение 16 новой вершины больше, чем у текущей вершины 14, идем к правому ребенку  
Значение 16 новой вершины больше, чем у текущей вершины 15, идем к правому ребенку  
Вставляем вершину 16  
Критерий баланса вершины 15 равен -1  
Вершина 15 сбалансирована  
Критерий баланса вершины 14 равен -2  
RR случай, поворачиваем поддерево влево в вершине 14  
-----Вставка завершена-----

### 4. Добавление элемента с перебалансировкой для случая LL

AVL-tree

```
graph TD; 14((14)) --> 12((12));
```

Удалить  Вставить

Вставляем вершину 14  
-----Вставка завершена-----  
Значение 12 новой вершины меньше, чем у текущей вершины 14, идем к левому ребенку  
Вставляем вершину 12  
Критерий баланса вершины 14 равен 1  
Вершина 14 сбалансирована  
-----Вставка завершена-----

AVL-tree

```
graph TD; 12((12)) --> 11((11)); 12((12)) --> 14((14));
```

Удалить  Вставить

Вставляем вершину 12  
Критерий баланса вершины 14 равен 1  
Вершина 14 сбалансирована  
-----Вставка завершена-----  
Значение 11 новой вершины меньше, чем у текущей вершины 14, идем к левому ребенку  
Значение 11 новой вершины меньше, чем у текущей вершины 12, идем к левому ребенку  
Вставляем вершину 11  
Критерий баланса вершины 12 равен 1  
Вершина 12 сбалансирована  
Критерий баланса вершины 14 равен 2  
LL случай, поворачиваем поддерево вправо в вершине 14  
-----Вставка завершена-----

## 5. Добавление элемент с перебалансировкой для случая RL

AVL-tree

```

graph TD
    16((16)) --- 15((15))
    16 --- 25((25))
    25 --- 30((30))
  
```

Удалить  Вставить

Вершина 16 сбалансирована  
 Критерий баланса вершины 15 равен -2  
 RR случай, поворачиваем поддерево влево в вершине 15  
 -----Вставка завершена-----  
 Значение 30 новой вершины больше, чем у текущей вершины 16, идем к правому ребенку  
 Значение 30 новой вершины больше, чем у текущей вершины 25, идем к правому ребенку  
 Вставляем вершину 30  
 Критерий баланса вершины 25 равен -1  
 Вершина 25 сбалансирована  
 Критерий баланса вершины 16 равен -1  
 Вершина 16 сбалансирована  
 -----Вставка завершена-----

AVL-tree

```

graph TD
    16((16)) --- 15((15))
    16 --- 27((27))
    27 --- 25((25))
    27 --- 30((30))
  
```

Удалить  Вставить

Значение 27 новой вершины больше, чем у текущей вершины 16, идем к правому ребенку  
 Значение 27 новой вершины больше, чем у текущей вершины 25, идем к правому ребенку  
 Значение 27 новой вершины меньше, чем у текущей вершины 30, идем к левому ребенку  
 Вставляем вершину 27  
 Критерий баланса вершины 30 равен 1  
 Вершина 30 сбалансирована  
 Критерий баланса вершины 25 равен -2  
 RL случай, сначала поворачиваем дерево вправо в вершине 30  
 Затем поворачиваем дерево влево в вершине 25  
 Критерий баланса вершины 16 равен -1  
 Вершина 16 сбалансирована  
 -----Вставка завершена-----

## 6. Добавление элемента с перебалансировкой для случая LR

AVL-tree

```

graph TD
    16((16)) --- 15((15))
    16 --- 25((25))
    15 --- 1((1))
  
```

Удалить  Вставить

Вершина 16 сбалансирована  
 Критерий баланса вершины 15 равен -2  
 RR случай, поворачиваем поддерево влево в вершине 15  
 -----Вставка завершена-----  
 Значение 1 новой вершины меньше, чем у текущей вершины 16, идем к левому ребенку  
 Значение 1 новой вершины меньше, чем у текущей вершины 15, идем к левому ребенку  
 Вставляем вершину 1  
 Критерий баланса вершины 15 равен 1  
 Вершина 15 сбалансирована  
 Критерий баланса вершины 16 равен 1  
 Вершина 16 сбалансирована  
 -----Вставка завершена-----

AVL-tree

```

graph TD
    16((16)) --- 5((5))
    16 --- 25((25))
    5 --- 1((1))
    5 --- 15((15))
  
```

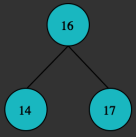
Удалить  Вставить

Значение 5 новой вершины меньше, чем у текущей вершины 16, идем к левому ребенку  
 Значение 5 новой вершины меньше, чем у текущей вершины 15, идем к левому ребенку  
 Значение 5 новой вершины больше, чем у текущей вершины 1, идем к правому ребенку  
 Вставляем вершину 5  
 Критерий баланса вершины 1 равен -1  
 Вершина 1 сбалансирована  
 Критерий баланса вершины 15 равен 2  
 LR случай, сначала поворачиваем дерево влево в вершине 1  
 Затем поворачиваем дерево вправо в вершине 15  
 Критерий баланса вершины 16 равен 1  
 Вершина 16 сбалансирована  
 -----Вставка завершена-----



## 7. Добавление уже существующего элемента

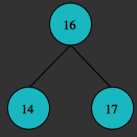
AVL-tree



Удалить  Вставить

Вставляем вершину 16  
-----Вставка завершена-----  
Значение 14 новой вершины меньше, чем у текущей вершины 16, идем к левому ребенку  
Вставляем вершину 14  
Критерий баланса вершины 16 равен 1  
Вершина 16 сбалансирована  
-----Вставка завершена-----  
Значение 17 новой вершины больше, чем у текущей вершины 16, идем к правому ребенку  
Вставляем вершину 17  
Критерий баланса вершины 16 равен 0  
Вершина 16 сбалансирована  
-----Вставка завершена-----

AVL-tree

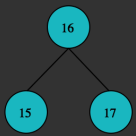


Удалить  Вставить

Вершина 16 сбалансирована  
-----Вставка завершена-----  
Значение 17 новой вершины больше, чем у текущей вершины 16, идем к правому ребенку  
Вставляем вершину 17  
Критерий баланса вершины 16 равен 0  
Вершина 16 сбалансирована  
-----Вставка завершена-----  
Значение 17 новой вершины больше, чем у текущей вершины 16, идем к правому ребенку  
Вершина с таким значением уже существует в дереве!  
Критерий баланса вершины 16 равен 0  
Вершина 16 сбалансирована  
-----Вставка завершена-----

## 8. Удаление не существующего элемента

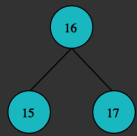
AVL-tree



Удалить  Вставить

Вставляем вершину 16  
Критерий баланса вершины 15 равен -1  
Вершина 15 сбалансирована  
-----Вставка завершена-----  
Значение 17 новой вершины больше, чем у текущей вершины 15, идем к правому ребенку  
Значение 17 новой вершины больше, чем у текущей вершины 16, идем к правому ребенку  
Вставляем вершину 17  
Критерий баланса вершины 16 равен -1  
Вершина 16 сбалансирована  
Критерий баланса вершины 15 равен -2  
RR случай, поворачиваем поддерево влево в вершине 15  
-----Вставка завершена-----

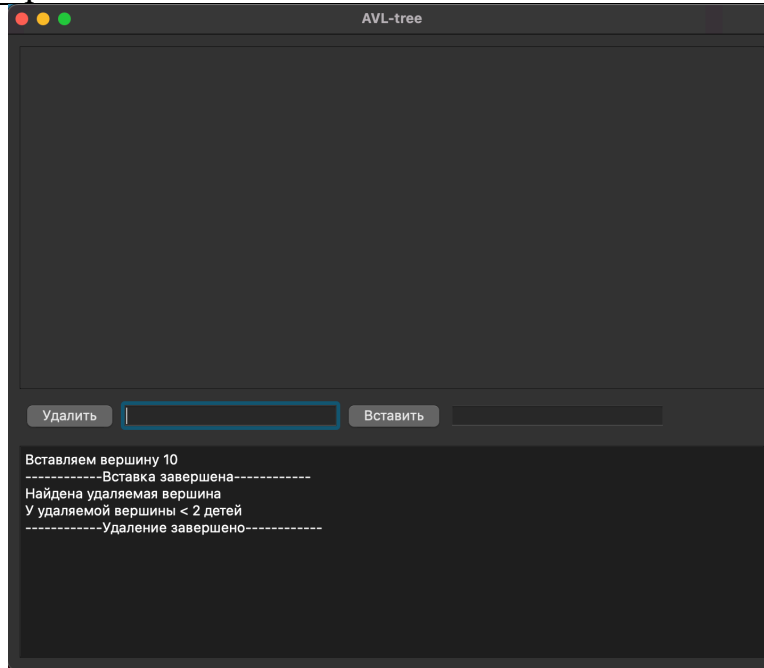
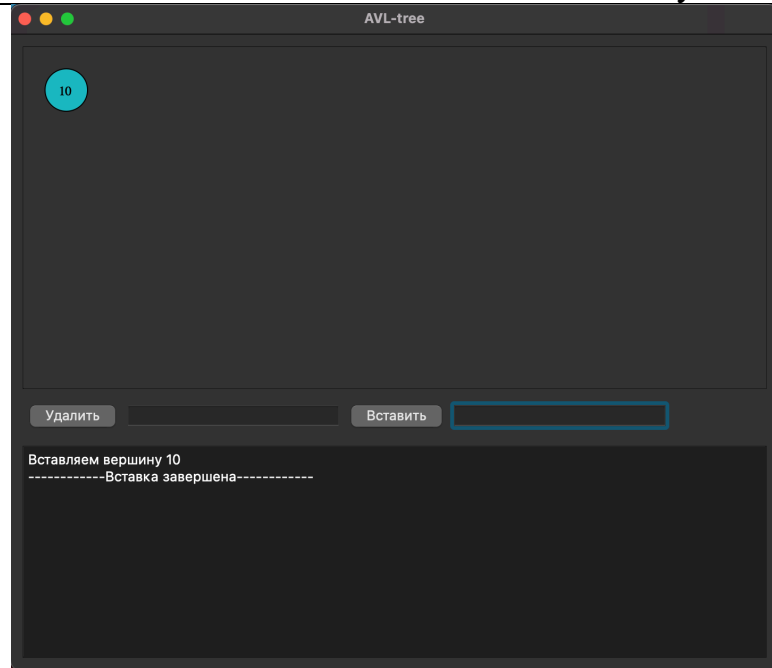
AVL-tree



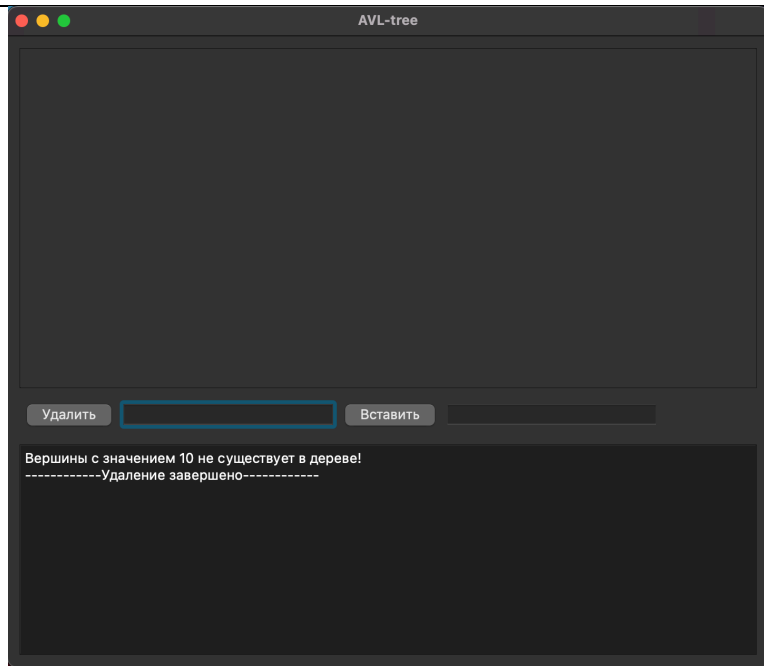
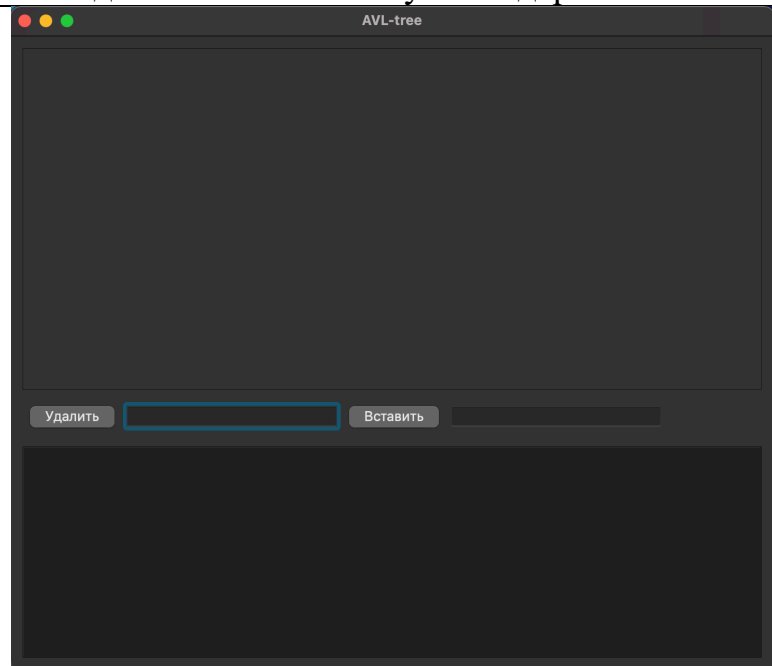
Удалить  Вставить

Вершина 16 сбалансирована  
Критерий баланса вершины 15 равен -2  
RR случай, поворачиваем поддерево влево в вершине 15  
-----Вставка завершена-----  
Значение 14 новой вершины меньше, чем у текущей вершины 16, идем к левому ребенку  
Значение 14 новой вершины меньше, чем у текущей вершины 15, идем к левому ребенку  
Вершины с значением 14 не существует в дереве!  
Критерий баланса вершины 15 равен 0  
Вершина 15 сбалансирована  
Критерий баланса вершины 16 равен 0  
Вершина 16 сбалансирована  
-----Удаление завершено-----

## 9. Удаление единственного элемента до пустого дерева



## 10. Удаление элемента пустого дерева



### 13. Удаление элемента, случай с 0 детьми

AVL-tree

```
graph TD; 17((17)) --- 16((16)); 17 --- 99((99));
```

Удалить  Вставить

Вставляем вершину 17  
Критерий баланса вершины 16 равен -1  
Вершина 16 сбалансирована  
-----Вставка завершена-----  
Значение 99 новой вершины больше, чем у текущей вершины 16, идем к правому ребенку  
Значение 99 новой вершины больше, чем у текущей вершины 17, идем к правому ребенку  
Вставляем вершину 99  
Критерий баланса вершины 17 равен -1  
Вершина 17 сбалансирована  
Критерий баланса вершины 16 равен -2  
RR случай, поворачиваем поддерево влево в вершине 16  
-----Вставка завершена-----

AVL-tree

```
graph TD; 17((17)) --- 16((16));
```

Удалить  Вставить

Вставляем вершину 99  
Критерий баланса вершины 17 равен -1  
Вершина 17 сбалансирована  
Критерий баланса вершины 16 равен -2  
RR случай, поворачиваем поддерево влево в вершине 16  
-----Вставка завершена-----  
Значение 99 новой вершины больше, чем у текущей вершины 17, идем к правому ребенку  
Найдена удаляемая вершина  
У удаляемой вершины < 2 детей  
Критерий баланса вершины 17 равен 1  
Вершина 17 сбалансирована  
-----Удаление завершено-----

### 14. Удаление элемента, случай с 1 ребенком

AVL-tree

```
graph TD; 67((67)) --- 35((35)); 67 --- 98((98)); 98 --- 345((345));
```

Удалить  Вставить

Вставляем вершину 35  
Критерий баланса вершины 67 равен 0  
Вершина 67 сбалансирована  
-----Вставка завершена-----  
Значение 345 новой вершины больше, чем у текущей вершины 67, идем к правому ребенку  
Значение 345 новой вершины больше, чем у текущей вершины 98, идем к правому ребенку  
Вставляем вершину 345  
Критерий баланса вершины 98 равен -1  
Вершина 98 сбалансирована  
Критерий баланса вершины 67 равен -1  
Вершина 67 сбалансирована  
-----Вставка завершена-----

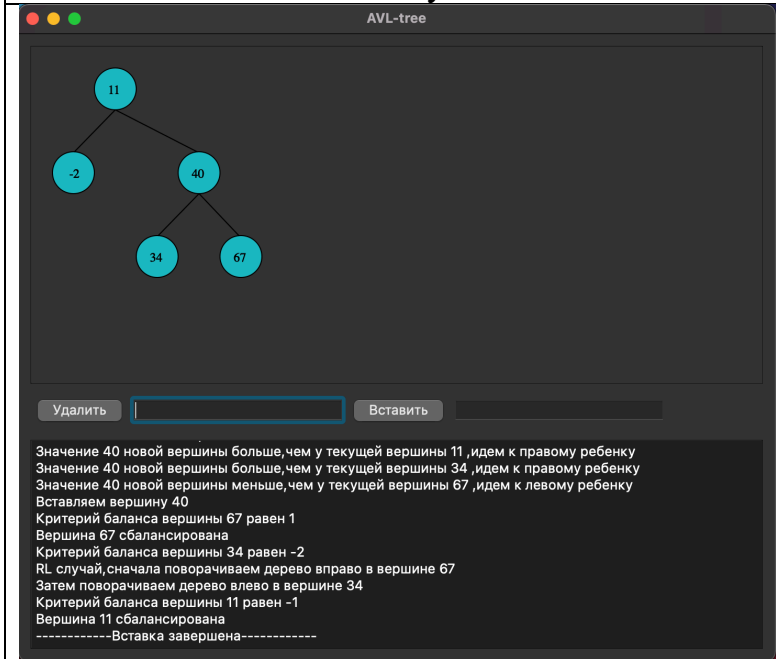
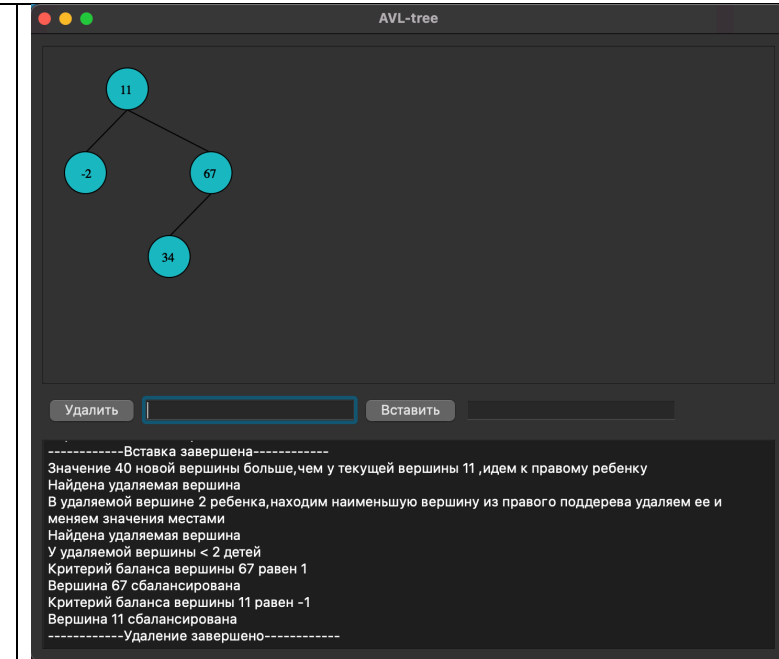
AVL-tree

```
graph TD; 67((67)) --- 35((35)); 67 --- 345((345));
```

Удалить  Вставить

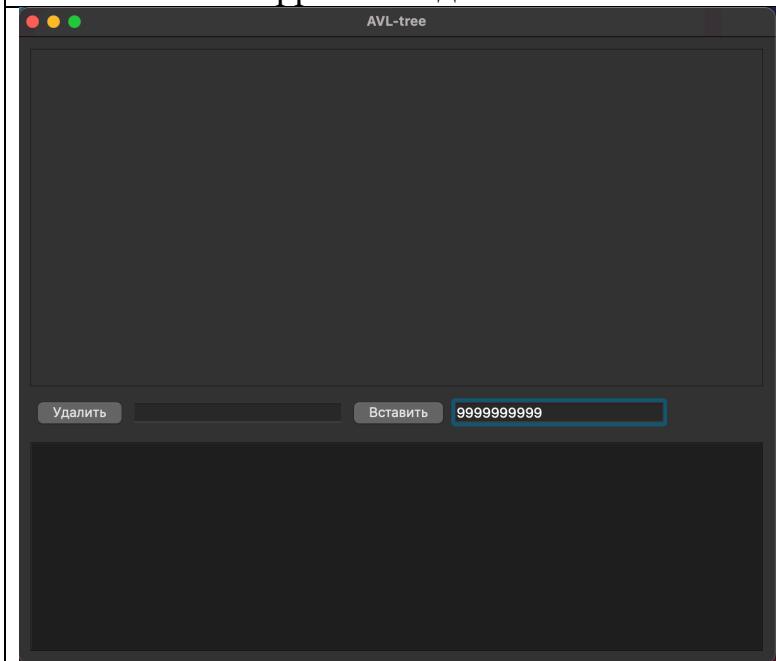
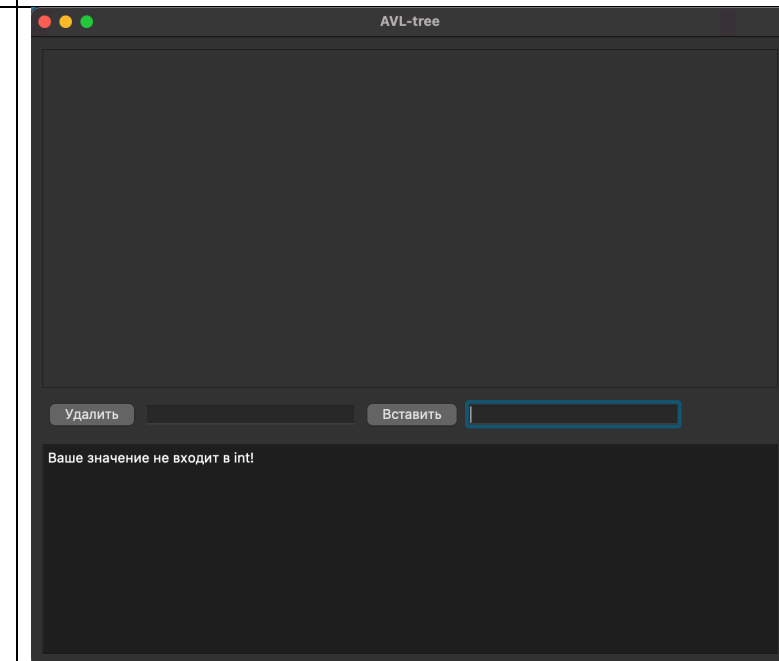
Вершина 98 сбалансирована  
Критерий баланса вершины 67 равен -1  
Вершина 67 сбалансирована  
-----Вставка завершена-----  
Значение 98 новой вершины больше, чем у текущей вершины 67, идем к правому ребенку  
Найдена удаляемая вершина  
У удаляемой вершины < 2 детей  
Критерий баланса вершины 345 равен 0  
Вершина 345 сбалансирована  
Критерий баланса вершины 67 равен 0  
Вершина 67 сбалансирована  
-----Удаление завершено-----

## 15. Удаление элемента случай с 2 детьми

 <p>Значение 40 новой вершины больше, чем у текущей вершины 11, идем к правому ребенку Значение 40 новой вершины больше, чем у текущей вершины 34, идем к правому ребенку Значение 40 новой вершины меньше, чем у текущей вершины 67, идем к левому ребенку Вставляем вершину 40 Критерий баланса вершины 67 равен 1 Вершина 67 сбалансирована Критерий баланса вершины 34 равен -2 RL случай, сначала поворачиваем дерево вправо в вершине 67 Затем поворачиваем дерево влево в вершине 34 Критерий баланса вершины 11 равен -1 Вершина 11 сбалансирована -----Вставка завершена-----</p>	 <p>-----Вставка завершена----- Значение 40 новой вершины больше, чем у текущей вершины 11, идем к правому ребенку Найдена удаляемая вершина В удаляемой вершине 2 ребенка, находим наименьшую вершину из правого поддерева удаляем ее и меняем значения местами Найдена удаляемая вершина У удаляемой вершины &lt; 2 детей Критерий баланса вершины 67 равен 1 Вершина 67 сбалансирована Критерий баланса вершины 11 равен -1 Вершина 11 сбалансирована -----Удаление завершено-----</p>
--	---

Поля ввода позволяют вводить только целые числа, но они все же позволяют вводить числа большие int, была сделана проверка и на это.

## 16. Тест на некорректных данных

 <p>Удалить Вставить 999999999</p>	 <p>Удалить Вставить</p> <p>Ваше значение не входит в int!</p>
--	--

Для удаления перебалансировка работает точно так же, как и для вставки.

## **ЗАКЛЮЧЕНИЕ**

В ходе работы была изучена структура данных АВЛ-дерево, освоены алгоритмы работы с этой структурой данных. Также была изучена среда QtCreator и ее инструменты для отображения интерфейса и прорисовки дерева.

## **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Qt документация // Qt documentation URL: <https://doc.qt.io>
2. Описание AVL-дерева // AVL tree URL :  
[https://en.wikipedia.org/wiki/AVL\\_tree](https://en.wikipedia.org/wiki/AVL_tree)

## ПРИЛОЖЕНИЕ А

### main.cpp

```
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

## ПРИЛОЖЕНИЕ Б

### AVL.h

```
#ifndef AVL_H
#define AVL_H

#include <iostream>
#include <QPainter>
#include <QColor>

template<typename Elem> class AVL;

//класс вершины
template<typename Elem>
class Node
{
    friend class AVL<Elem>;
public:
    Node(const Elem &);
    Elem value;
    Node<Elem> *leftChild;
    Node<Elem> *rightChild;
    Node<Elem> *parent;

    int height;
    QColor color;
    int x;
};
```

```

//класс AVL-дерева
template<typename Elem>
class AVL
{
public:
    AVL();

    //методы для копирования дерева
    AVL copy();
    Node<Elem>* recursiveCopy(Node<Elem> *p);

    //метод проверяющий не пусто ли дерево
    bool isEmpty() const;

    //методы по удалению и вставки элемента в дерево
    Node<Elem>* insertNode(Node<Elem>* node, const Elem &,
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages);
    bool insertValue(const Elem &, std::vector<AVL<Elem>> &vectorAVL,
std::vector<std::string> &messages);
    Node<Elem>* deleteNode(Node<Elem>* node, const Elem &,
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages);
    bool deleteValue(const Elem &, std::vector<AVL<Elem>> &vectorAVL,
std::vector<std::string> &messages);

    //перебалансировка заданной вершины
    Node<Elem>* rebalance(Node<Elem> * node, const Elem &item,
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages);

    //методы для осуществления поворота поддерева
    Node<Elem>* rightRotate(Node<Elem> *y);
    Node<Elem>* leftRotate(Node<Elem> *x);

    //подсчет баланса вершины
    int getBalance(Node<Elem> *node);

    //метод возвращающий минимальную вершину в поддереве
    Node<Elem>* minValueNode(Node<Elem>* node);

    //метод сбрасывает положения по горизонтали всех вершин дерева до 0
    void resetNodePosition(Node<Elem> *node);

    //метод возвращает минимальную вершину в поддереве с корнем в вершине node

```



```

Node<Elem>* getLeftmostNode(Node<Elem>* node);

//метод возвращает уровень вершины
int getNodeLevel(Node<Elem> *node);

//метод возвращающий положение по горизонтали самой правой вершины
int getPxLocOfLeftTree(const Node<Elem> *node);

//метод возвращающий положение по горизонтали первого предка вершины
node(начиная с нее)
int getPxLocOfAncestor(const Node<Elem> *node);

//метод возвращающий корень дерева
Node<Elem> *getRoot();

//метод возвращающий ширину дерева
int getTotalX() const;

//метод возвращающий высоту дерева(расстояние)
int getTotalY() const;

//метод возвращающий высоту дерева(по количеству вершин)
int getTreeHeight(const Node<Elem> *node) const;

//метод возвращающий высоту вершины(по количеству вершин)
int getHeight(Node<Elem> *node);

int max(int a, int b) const;

QPainter *painter;
int yspace;
int xspace;
int nodeRadius;
double scale;

private:
Node<Elem> *root;

};

template<typename Elem>
AVL<Elem> AVL<Elem>::copy() {

```

```

    AVL<Elem> avl = AVL<Elem>();
    avl.root = recursiveCopy(root);
    avl.painter = painter;
    avl.yspace = yspace;
    avl.xspace = xspace;
    avl.scale = scale;
    avl.nodeRadius = nodeRadius;
    return avl;
}

template<typename Elem>
Node<Elem>* AVL<Elem>::recursiveCopy(Node<Elem> *p) {

    if(p == nullptr) {
        return nullptr;
    }

    Node<Elem> *n = new Node<Elem>(p->value);
    n->leftChild=recursiveCopy(p->leftChild);

    if(n->leftChild != nullptr)
        n->leftChild->parent = n;

    n->rightChild=recursiveCopy(p->rightChild);

    if(n->rightChild != nullptr)
        n->rightChild->parent = n;

    n->height=p->height;
    n->x = p->x;
    n->color = p->color;

    return n;
}

template<typename Elem>
AVL<Elem>::AVL() : scale(1), root(nullptr) {}

template<typename Elem>
int AVL<Elem>::max(int a, int b) const{
    return (a>b)? a : b;
}

```

```

template<typename Elem>
int AVL<Elem>::getHeight(Node<Elem> *node) {
    if(node == nullptr){
        return 0;
    }
    return node->height;
}

template<typename Elem>
int AVL<Elem>::getBalance(Node<Elem> *node){
    if(node == nullptr){
        return 0;
    }
    return getHeight(node->leftChild) - getHeight(node->rightChild);
}

template<typename Elem>
Node<Elem>::Node(const Elem &data) : value(data), leftChild(nullptr),
rightChild(nullptr), parent(nullptr), height(1), color(QColor(88, 181, 191)) ,
x(0) {}

template<typename Elem>
bool AVL<Elem>::isEmpty() const {
    return root == nullptr;
}

template<typename Elem>
bool AVL<Elem>::insertValue(const Elem & item, std::vector<AVL<Elem>>
&vectorAVL, std::vector<std::string> &messages){
    root = insertNode(root,item, vectorAVL, messages);
    vectorAVL.push_back(this->copy());
    messages.push_back("-----Вставка завершена-----");
    return true;
}

template<typename Elem>
Node<Elem>* AVL<Elem>::insertNode(Node<Elem>* node, const Elem &item,
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages){
    //если мы достигли null-вершины, то вместо нее можно вставить новую вершину
    if(node == nullptr){
        Node<Elem> *newNode = new Node<Elem>(item);

```

```

        vectorAVL.push_back(this->copy());
        messages.push_back("Вставляем вершину " + std::to_string(item));
        return newNode;
    }

    //если вставляемое значение меньше текущей вершины, то нам следует
    переместиться к левому ребенку
    if(item < node->value){
        node->color = Qt::green;
        vectorAVL.push_back(this->copy());
        messages.push_back("Значение " + std::to_string(item) + " новой вершины
меньше, чем у текущей вершины " + std::to_string(node->value) + " ,идем к левому
ребенку ");
        node->color = QColor(88, 181, 191);
        node->leftChild = insertNode(node->leftChild, item, vectorAVL,
messages);
        node->leftChild->parent = node;
    }

    //если вставляемое значение больше текущей вершины, то следует переместиться
к правому ребенку
    else if(item > node->value){
        node->color = Qt::green;
        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("Значение " + std::to_string(item) + " новой вершины
больше, чем у текущей вершины " + std::to_string(node->value) + " ,идем к
правому ребенку ");

        node->rightChild = insertNode(node->rightChild, item, vectorAVL,
messages);
        node->rightChild->parent = node;
    }

    //если значение вершины равно вставляемому значению, выведем сообщение об
этом
    else {
        node->color = QColor(255, 253, 111);
        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("Вершина с таким значением уже существует в
дереве!");
        return node;
    }
}

```

```

        //подстраиваем высоту вершины
        node->height = 1 + max(getHeight(node->leftChild), getHeight(node->rightChild));

        //проверяем баланс вершины
        return rebalance(node, item, vectorAVL, messages);
    }

template<typename Elem>
Node<Elem> * AVL<Elem>::rebalance(Node<Elem> *node, const Elem& item,
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages){
    int balance = getBalance(node);

    node->color = QColor(250, 184, 254);
    vectorAVL.push_back(this->copy());
    node->color = QColor(88, 181, 191);
    messages.push_back("Критерий баланса вершины " + std::to_string(node->value)
+ " равен " + std::to_string(balance));

    /*
    * Если balance > 1, значит проблема в левом поддереве
    * Если ключ меньше чем левый ребенок рассматриваемой вершины то это
    * случай Left Left
    */
    if(balance > 1 && item < node->leftChild->value){
        node->color = QColor(255, 0, 0);
        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("LL случай, поворачиваем поддерево вправо в вершине "
+ std::to_string(node->value));

        return rightRotate(node);
    }

    /*
    * Если balance < -1, значит проблема в правом поддереве
    * Если ключ больше чем правый ребенок рассматриваемой вершины то это
    * случай Right Right
    */
    if(balance < -1 && item > node->rightChild->value){
        node->color = QColor(255, 0, 0);

```

```

        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("RR случай, поворачиваем поддерево влево в вершине "
+ std::to_string(node->value));

        return leftRotate(node);
    }

    /*
    * Если balance > 1, значит проблема в левом поддереве
    * Если ключ больше чем левый ребенок рассматриваемой вершины то это
    * случай Left Right
    */
    if(balance > 1 && item > node->leftChild->value){
        node->leftChild->color = QColor(255, 0, 0);
        vectorAVL.push_back(this->copy());
        node->leftChild->color = QColor(88, 181, 191);
        messages.push_back("LR случай, сначала поворачиваем дерево влево в
вершине " + std::to_string(node->leftChild->value));

        node->leftChild = leftRotate(node->leftChild);

        node->color = QColor(255, 0, 0);
        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("Затем поворачиваем дерево вправо в вершине " +
std::to_string(node->value));

        return rightRotate(node);
    }

    /*
    * Если balance < -1, значит проблема в правом поддереве
    * Если ключ меньше чем правый ребенок рассматриваемой вершины то это
    * случай Right Left
    */
    if(balance < -1 && item < node->rightChild->value){
        node->rightChild->color = QColor(255, 0, 0);
        vectorAVL.push_back(this->copy());
        node->rightChild->color = QColor(88, 181, 191);
        messages.push_back("RL случай, сначала поворачиваем дерево вправо в
вершине " + std::to_string(node->rightChild->value));

```

```

        node->rightChild = rightRotate(node->rightChild);

        node->color = QColor(255, 0, 0);
        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("Затем поворачиваем дерево влево в вершине " +
std::to_string(node->value));

        return leftRotate(node);
    }

    node->color = QColor(197, 184, 254);
    vectorAVL.push_back(this->copy());
    node->color = QColor(88, 181, 191);
    messages.push_back("Вершина "+std::to_string(node->value)+"
сбалансирована");

    return node;
}

template<typename Elem>
Node<Elem>* AVL<Elem>::minValueNode(Node<Elem>* node) {
    Node<Elem>* current = node;
    while(current->leftChild != nullptr) {
        current = current->leftChild;
    }
    return current;
}

template<typename Elem>
bool AVL<Elem>::deleteValue(const Elem &item, std::vector<AVL<Elem>> &vectorAVL,
std::vector<std::string> &messages) {
    root = deleteNode(root, item, vectorAVL, messages);

    vectorAVL.push_back(this->copy());
    messages.push_back("-----Удаление завершено-----");
    return true;
}

template<typename Elem>

```

```

Node<Elem>* AVL<Elem>::deleteNode(Node<Elem>* node, const Elem &item,
std::vector<AVL<Elem>> &vectorAVL, std::vector<std::string> &messages){
    //если мы достигли null-вершины, то это значит, что вершины с таким
    значением нет в дереве и нам нечего удалять
    if(node == nullptr){
        vectorAVL.push_back(this->copy());
        messages.push_back("Вершины с значением " + std::to_string(item) + " не
    существует в дереве!");
        return node;
    }
    // если значение, которое мы хотим удалить, меньше значения текущей вершины,
    пойдём к левому дереву
    if(item < node->value){
        node->color = Qt::green;
        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("Значение " + std::to_string(item) + " новой вершины
    меньше, чем у текущей вершины " + std::to_string(node->value) + " ,идём к левому
    ребёнку ");

        node->leftChild = deleteNode(node->leftChild,item, vectorAVL, messages);
    }
    // если значение, которое мы хотим удалить, больше значения текущей вершины,
    пойдём к правому дереву
    else if(item > node->value){
        node->color = Qt::green;
        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("Значение " + std::to_string(item) + " новой вершины
    больше, чем у текущей вершины " + std::to_string(node->value) + " ,идём к
    правому ребёнку ");

        node->rightChild = deleteNode(node->rightChild,item, vectorAVL,
    messages);
    }
    //нашли вершину, которую надо удалить
    else {
        node->color = QColor(203, 205, 192);
        vectorAVL.push_back(this->copy());
        node->color = QColor(88, 181, 191);
        messages.push_back("Найдена удаляемая вершина");
    }
}

```



```

//смотрим случай, когда у вершины 0 или 1 ребенок
if(node->leftChild == nullptr || node->rightChild == nullptr){
    Node<Elem> *temp = node->leftChild ? node->leftChild: node->rightChild;

    node->color = QColor(165, 205, 192);
    vectorAVL.push_back(this->copy());
    node->color = QColor(88, 181, 191);
    messages.push_back("У удаляемой вершины < 2 детей");

    // если нет обоих детей
    // то просто удаляем вершину
    if(temp == nullptr){
        temp = node;
        node = nullptr;
    }
    //если один ребенок, то "заменяем" эту вершину одним ребенком
    else {
        node->height = temp->height;
        node->leftChild = temp->leftChild;
        node->rightChild = temp->rightChild;
        node->value = temp->value;
        node->x = temp->x;
        node->color = temp->color;
    }
    delete temp;
}

//если же ребенка 2, то надо найти самую наименьшую(левую) вершину из
правого поддерева
//значение найденной вершины подставляем в "удаляемую" вершину, а
найденную вершину удаляем
else {
    node->color = QColor(237, 205, 192);
    vectorAVL.push_back(this->copy());
    node->color = QColor(88, 181, 191);
    messages.push_back("В удаляемой вершине 2 ребенка, находим
наименьшую вершину из правого поддерева удаляем ее и меняем значения местами");

    Node<Elem>* temp = minValueNode(node->rightChild);
    node->value = temp->value;

```

```

        node->rightChild = deleteNode(node->rightChild, temp->value,
vectorAVL, messages);
    }
}

//если у вершины не было детей, то баланс не изменится
if(node == nullptr){
    return node;
}

node->height = 1 + max(getHeight(node->leftChild), getHeight(node-
>rightChild));

return rebalance(node, item, vectorAVL, messages);
}

/*
* Метод левого поворота
*
*      x                      y
*     / \                    / \
*    T1  y      --->      x    T3
*       / \                /  \
*      T2  T3             T1   T2
*
*/
template<typename Elem>
Node<Elem>* AVL<Elem>::leftRotate(Node<Elem>* x) {
    Node<Elem> *y = x->rightChild;
    Node<Elem> *T2 = y->leftChild;

    y->leftChild = x;
    y->parent = x->parent;
    x->parent = y;
    x->rightChild = T2;

    if(T2 != nullptr){
        T2->parent = x;
    }

    x->height = max(getHeight(x->leftChild), getHeight(x->rightChild)) + 1;
    y->height = max(getHeight(y->leftChild), getHeight(y->rightChild)) + 1;

```

```

        return y;
    }

/*
 * Метод правого поворота
 *
 *      y                      x
 *      / \                  /  \
 *      x  T3      ---->    T1   y
 *      /  \                /   \
 *      T1  T2              T2   T3
 */
template<typename Elem>
Node<Elem>* AVL<Elem>::rightRotate(Node<Elem>* y) {
    Node<Elem> *x = y->leftChild;
    Node<Elem> *T2 = x->rightChild;

    x->rightChild = y;
    x->parent = y->parent;
    y->parent = x;
    y->leftChild = T2;

    if(T2 != nullptr){
        T2->parent = y;
    }

    y->height = max(getHeight(y->leftChild), getHeight(y->rightChild)) + 1;
    x->height = max(getHeight(x->leftChild), getHeight(x->rightChild)) + 1;

    return x;
}

template<typename Elem>
Node<Elem> * AVL<Elem>::getRoot() {
    return this->root;
}

template<typename Elem>
void AVL<Elem>::resetNodePosition(Node<Elem> *node) {
    if(node == nullptr){

```

```

        return;
    }
    resetNodePosition(node->leftChild);
    node->x = 0;
    resetNodePosition(node->rightChild);
}

template<typename Elem>
Node<Elem>* AVL<Elem>::getLeftmostNode(Node<Elem> *node) {
    if(root == nullptr)
        return nullptr;
    if(node->leftChild == nullptr) {
        return node;
    }
    return getLeftmostNode(node->leftChild);
}

template<typename Elem>
int AVL<Elem>::getNodeLevel(Node<Elem> *node) {
    int level = 1;
    Node<Elem> *current = node;
    if(current != nullptr) {
        while(current->parent != nullptr) {
            current = current->parent;
            ++level;
        }
    }
    return level;
}

template<typename Elem>
int AVL<Elem>::getPxLocOfLeftTree(const Node<Elem> *node) {
    if(node->rightChild == nullptr) {
        return node->x;
    }
    return getPxLocOfLeftTree(node->rightChild);
}

template<typename Elem>
int AVL<Elem>::getPxLocOfAncestor(const Node<Elem> *node) {
    Node<Elem> *currentNode = node->parent;
    if(currentNode != nullptr) {

```

```

while(currentNode->x == 0)
    currentNode = currentNode->parent;
return currentNode->x;
}
else {
    return node->x;
}
}

template<typename Elem>
int AVL<Elem>::getTreeHeight(const Node<Elem> *node) const{
    if(node == nullptr || (node->leftChild == nullptr && node->rightChild ==
nullptr)){
        return 0;
    }
    return 1 + max(getTreeHeight(node->leftChild), getTreeHeight(node-
>rightChild));
}

template<typename Elem>
int AVL<Elem>::getTotalY() const {
    int level = getTreeHeight(root) + 1;
    return (level * nodeRadius * 2 + yspace * (level-1)) + nodeRadius * 2;
}

template<typename Elem>
int AVL<Elem>::getTotalX() const {
    if(this->root == nullptr){
        return nodeRadius*3;
    }
    Node<Elem> *current = root;
    while(current->rightChild != nullptr){
        current = current->rightChild;
    }
    return current->x + nodeRadius * 3;
}

#endif // AVL_H

```

## ПРИЛОЖЕНИЕ В

### drawarea.h

```
#ifndef DRAWAREA_H
#define DRAWAREA_H

#include <QWidget>
#include <QPen>
#include <QBrush>
#include <QColor>
#include "AVL.h"

class DrawArea : public QWidget
{
    Q_OBJECT
public:
    explicit DrawArea(AVL<int> *avl, QWidget *parent = nullptr);

    //методы для автоматического изменения размера картинки
    QSize minimumSizeHint() const override;
    QSize sizeHint() const override;
    void autoSize();
    void callRepaint();

    //метод для установки дерева
    void setTree(AVL<int> *tree);

    //методы, рисующие дерево
    void draw(QPainter *painter, double &scale);
    void recursiveDraw(Node<int> *node);

protected:
    //этот метод вызывается для рисования
    void paintEvent(QPaintEvent *event) override;

private:
    AVL<int> *avl;
    double scale;
    QPen pen;
    QBrush brush;
    QColor backgroundColor;
```

```

        QColor textColor;
    };

#endif // DRAWAREA_H

```

## ПРИЛОЖЕНИЕ Г

### drawarea.cpp

```

#include "drawarea.h"

DrawArea::DrawArea(AVL<int> *avl, QWidget *parent) : QWidget(parent), avl(avl),
scale(1.0), backgroundColor(Qt::white), textColor(Qt::black) {}

void DrawArea::setTree(AVL<int> *tree) {
    this->avl = tree;
}

void DrawArea::paintEvent(QPaintEvent *) {

    if (this->avl->isEmpty())
        return;

    QPainter painter(this);

    painter.setRenderHint(QPainter::Antialiasing);

    QPen pen;
    pen.setColor(this->textColor);

    painter.setBrush(brush);
    painter.setPen(pen);

    draw(&painter, this->scale);

    this->autoSize();
}

void DrawArea::draw(QPainter *painter, double &scale) {
    if (avl->getRoot() == nullptr)
        return;
}

```

```

//Свойства для отрисовки дерева
avl->painter = painter;
avl->painter->setFont(QFont("Times", 12*scale, QFont::Normal));

//Размерности дерева и его элементов
avl->scale = scale;
avl->nodeRadius = 20*scale;

//отступы между вершинами по горизонтали и вертикали
avl->xspace = avl->nodeRadius;
avl->yspace = avl->nodeRadius * 2;

//Перед тем, как начать рисовать, надо убедиться, что все вершины имеют
позицию по x=0
avl->resetNodePosition(avl->getRoot());

//для самой левой вершины делаем отступ равным радиусу вершины * 2
Node<int> *leftmost = avl->getLeftmostNode(avl->getRoot());
leftmost->x = avl->nodeRadius * 2;

//рекурсивно отрисовываем дерево, начиная с корня
recursiveDraw(avl->getRoot());
}

void DrawArea::recursiveDraw(Node<int> *node) {
    if(node == nullptr) {
        return;
    }

    //рисуем левое поддерево
    this->recursiveDraw(node->leftChild);

    //узнаем уровень вершины(порядок от корня вниз)
    int level = avl->getNodeLevel(node);

    //ставим отступ по вертикали
    int y = level*avl->nodeRadius*2 + avl->yspace*(level-1);

    //если у вершины есть левый ребенок
    if(node->leftChild != nullptr) {

```



```

        //ставим отступ по горизонтали, прибавляем к положению самой правой
вершины левого поддерева, радиус + отступ
        node->x = avl->getPxLocOfLeftTree(node->leftChild) + avl->nodeRadius +
avl->xspace;

        //строим отрезок между двумя точками: самой нижней точкой текущей
вершины и самой верхней точкой левого ребенка
        avl->painter->drawLine(QPoint(node->x, y + avl->nodeRadius),
QPoint(node->leftChild->x + 1, ((level+1)* avl->nodeRadius*2 + avl->yspace *
level) - avl->nodeRadius));
    }
    else if(node->x == 0){
        //если у вершины нет левого ребенка и она имеет начальное положение
        //подвинем ее правее от самого первого предка(начиная с текущей
вершины)
        node->x = avl->getPxLocOfAncestor(node) + avl->nodeRadius + avl->xspace;
    }

    //ставим цвета
    QBrush brush;
    brush.setColor(node->color);
    brush.setStyle(Qt::SolidPattern);
    avl->painter->setBrush(brush);

    //рисует круг(вершину)
    avl->painter->drawEllipse(QPoint(node->x, y), avl->nodeRadius, avl-
>nodeRadius);

    //переменная для корректировки положения числа в зависимости от количества в
нем знаков
    int textAdjuster;
    if(fabs(node->value) < 10){
        textAdjuster = 4;
    }
    else if(fabs(node->value) < 100){
        textAdjuster = 7;
    }
    else if(fabs(node->value) < 1000){
        textAdjuster = 12;
    }
    else {
        textAdjuster = 16;
    }

```

```

    }

    //рисование числа
    avl->painter->drawText(QPoint(node->x - textAdjuster*scale, y+5*scale),
QString::number(node->value));

    //рисование правого поддерева
    this->recursiveDraw(node->rightChild);

    //рисование отрезка между вершиной и его правым ребенком аналогично с левым
отрезком
    if(node->rightChild != nullptr){
        avl->painter->drawLine(QPoint(node->x, y + avl->nodeRadius),
QPoint(node->rightChild->x - 2, ((level + 1) * avl->nodeRadius * 2 + avl->yspace
* level) - avl->nodeRadius));
    }
}

QSize DrawArea::sizeHint() const{
    return QSize(50,50);
}

QSize DrawArea::minimumSizeHint() const{
    return QSize(50,50);
}

void DrawArea::callRepaint(){
    if(this->avl->isEmpty()){
        return;
    }
    this->scale +=0.1;
    this->repaint();
    this->scale -=0.1;
    this->repaint();
}

void DrawArea::autoSize(){
    QSize size(avl->getTotalX(), avl->getTotalY());
    this->setMinimumSize(size);
    this->resize(size);
}

```

## ПРИЛОЖЕНИЕ Д

### mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QPushButton>
#include <QLineEdit>
#include <QScrollArea>
#include <QVBoxLayout>
#include <QLabel>
#include <QAction>
#include <QTextEdit>

#include "drawarea.h"
#include "AVL.h"
#include <vector>

class MainWindow : public QMainWindow
{
    Q_OBJECT
    QWidget *centralWidget;

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private:
    //Кнопки
    QPushButton *deleteButton;
    QPushButton *insertButton;

    //Поля для ввода значения, которое нужно удалить либо вставить в дерево
    QLineEdit *insertValueLineEdit;
    QLineEdit *deleteValueLineEdit;

    QTextEdit *treeActions;

    QScrollArea *treeScrollArea;

    QVBoxLayout *mainLayout;
```

```

        DrawArea *drawArea;

        AVL<int> *avl;

protected:
        virtual void resizeEvent(QResizeEvent *event);

private slots:
        void insertClicked() const;
        void deleteClicked() const;

};
#endif // MAINWINDOW_H

```

## ПРИЛОЖЕНИЕ Е

### mainwindow.cpp

```

#include "mainwindow.h"
#include <QTime>
#include <QCoreApplication>
#include <QIntValidator>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    this->avl = new AVL<int>;

    //Создаем кнопки
    deleteButton = new QPushButton("Удалить", this);
    insertButton = new QPushButton("Вставить", this);
    insertValueLineEdit = new QLineEdit;
    deleteValueLineEdit = new QLineEdit;
    treeActions = new QTextEdit;

    //Задаем параметры для кнопок и полей
    deleteButton->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
    insertButton->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
    insertValueLineEdit->setFixedWidth(200);
    deleteValueLineEdit->setFixedWidth(200);
    insertValueLineEdit->setValidator(new QIntValidator(-2147483648,
2147483647, this));

```

```

        deleteValueLineEdit->setValidator(new QIntValidator(-2147483648,
2147483647, this));
        treeActions->setFixedWidth(700);
        treeActions->setFixedHeight(200);
        treeActions->setReadOnly(true);

        // Связываем слоты с сигналами кнопок
        connect(deleteButton, SIGNAL(clicked()), this, SLOT(deleteClicked()));
        connect(insertButton, SIGNAL(clicked()), this, SLOT(insertClicked()));
        connect(insertValueLineEdit, SIGNAL(returnPressed()), this,
SLOT(insertClicked()));
        connect(deleteValueLineEdit, SIGNAL(returnPressed()), this,
SLOT(deleteClicked()));

        //Создаем слой для кнопок и добавляем туда кнопки
        QHBoxLayout *buttonLayout = new QHBoxLayout;
        buttonLayout->addWidget(deleteButton);
        buttonLayout->addWidget(deleteValueLineEdit);
        buttonLayout->addWidget(insertButton);
        buttonLayout->addWidget(insertValueLineEdit);
        buttonLayout->addStretch(10);

        //Создаем пространство для отрисовки дерева
        drawArea = new DrawArea(this->avl);

        treeScrollArea = new QScrollArea;
        treeScrollArea->setWidget(drawArea);
        treeScrollArea->installEventFilter(drawArea);

        //Создаем основной слой
        mainLayout = new QVBoxLayout;
        mainLayout->addWidget(treeScrollArea);
        mainLayout->addLayout(buttonLayout);
        mainLayout->addWidget(treeActions);

        //Создаем основное окно
        centralWidget = new QWidget(this);
        centralWidget->setLayout(mainLayout);
        this->setCentralWidget(centralWidget);
        this->setMinimumHeight(600);
        this->setMinimumWidth(700);

```

```

        this->setWindowTitle("AVL-tree");

        this->show();
    }

MainWindow::~MainWindow()
{
    delete drawArea;
    delete deleteButton;
    delete insertButton;
    delete treeScrollArea;
    delete avl;
    delete centralWidget;
}

//функция для задержки времени, чтобы можно было видеть шаги
void delay()
{
    QTime dieTime= QTime::currentTime().addSecs(1);
    while (QTime::currentTime() < dieTime)
        QCoreApplication::processEvents(QEventLoop::AllEvents, 100);
}

// метод, который вызывается при нажатии на кнопку delete
void MainWindow::deleteClicked() const {
    //считываем значение, которое хотим удалить
    QString value = deleteValueLineEdit->text();

    //проверка на некорректный ввод
    if(value==" " || value.contains(" ") || value.toLongLong() < INT_MIN ||
value.toLongLong() > INT_MAX){
        treeActions->append("Ваше значение не входит в int!");
        deleteValueLineEdit->setText("");
        return;
    }

    int val = value.toInt();

    //на время отработки алгоритма блокируем все кнопки и поля, чтобы ничего не
сломать
    deleteValueLineEdit->setReadOnly(true);
    insertValueLineEdit->setReadOnly(true);

```

```

deleteButton->setEnabled(false);
insertButton->setEnabled(false);

//инициализируем переменные для хранения промежуточного состояния дерева и
сообщений о действиях
std::vector<AVL<int>> trees;
std::vector<std::string> messages;

//вызываем метод, который производит сам алгоритм и записывает промежуточные
результаты в соответствующие переменные
avl->deleteValue(val, trees, messages);

//производим отображение всех шагов алгоритма
for(unsigned int i = 0; i<messages.size(); i++){
    treeActions->append(QString::fromStdString(messages[i]));
    drawArea->setTree(&trees[i]);
    drawArea->repaint();
    delay();
}

//после отработки алгоритмов возвращаем кнопки и поля в рабочее состояние
deleteValueLineEdit->setText("");
deleteValueLineEdit->setReadOnly(false);
insertValueLineEdit->setReadOnly(false);
deleteButton->setEnabled(true);
insertButton->setEnabled(true);
}

void MainWindow::insertClicked() const {
    //считываем значение, которое хотим вставить
    QString value = insertValueLineEdit->text();

    //проверка на некорректный ввод
    if(value==" " || value.contains(" ") || value.toLongLong() < INT_MIN ||
value.toLongLong() > INT_MAX){
        treeActions->append("Ваше значение не входит в int!");
        insertValueLineEdit->setText("");
        return;
    }

    int val = value.toInt();

```

```

        //на время отработки алгоритма блокируем все кнопки и поля, чтобы ничего не
        сломать
        deleteValueLineEdit->setReadOnly(true);
        insertValueLineEdit->setReadOnly(true);
        deleteButton->setEnabled(false);
        insertButton->setEnabled(false);

        //инициализируем переменные для хранения промежуточного состояния дерева и
        сообщений о действиях
        std::vector<AVL<int>> trees;
        std::vector<std::string> messages;

        //вызываем метод, который производит сам алгоритм и записывает промежуточные
        результаты в соответствующие переменные
        avl->insertValue(val, trees, messages);

        //производим отображение всех шагов алгоритма
        for(unsigned int i = 0; i<messages.size(); i++){
            treeActions->append(QString::fromStdString(messages[i]));
            drawArea->setTree(&trees[i]);
            drawArea->repaint();
            delay();
        }

        //после отработки алгоритмов возвращаем кнопки и поля в рабочее состояние
        insertValueLineEdit->setText("");
        deleteValueLineEdit->setReadOnly(false);
        insertValueLineEdit->setReadOnly(false);
        deleteButton->setEnabled(true);
        insertButton->setEnabled(true);
    }

    //этот метод предназначен для автоматического изменения размера отображаемой
    картинки
    void MainWindow::resizeEvent(QResizeEvent* event){
        QMainWindow::resizeEvent(event);
        this->drawArea->callRepaint();
    }

```