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

ЛАБОРАТОРНАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: АВЛ Дерево

Студент гр. 8381 _____ Перелыгин Д.С.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

ВВЕДЕНИЕ

Цель: Реализовать АВЛ дерево и основные функции для работы с ним. Возможность использовать программу как средство обучения через демонстрацию.

Задача должна быть решена с помощью возможностей языка программирования C++; Программа должна иметь дружественный для пользователя интерфейс.

Задание:

В вариантах заданий 2-ой группы (БДП и хеш-таблицы) требуется:

1) По заданному файлу F (типа file of *Elem*), все элементы которого различны, построить структуру данных определённого типа – БДП или хеш-таблицу;
2) Выполнить одно из следующих действий:

а) Для построенной структуры данных проверить, входит ли в неё элемент *e* типа *Elem*, и если не входит, то добавить элемент *e* в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

АЛГОРИТМ РАБОТЫ ПРОГРАММЫ

Описание алгоритма вставки:

Элементы дерева вставляют как в обычное упорядоченное дерево (справа от элемента – элементы большие, а слева от элемента элементы меньшие). Но есть отличие от обычного упорядоченного бинарного дерева. По возвращению из рекурсии дерево преобразуется таким образом чтобы максимальное различие высоты между двумя его любыми ветвями было не более чем 1 – это обеспечивает логарифмическую сложность. Эти преобразования происходят если у элемента дерева `balance`не лежит в пределах [-1;1]. Здесь различаются 4 разных ситуации:

1) нарушение высоты в случае:лево-лево
2) нарушение высоты в случае:лево-право
3) нарушение высоты в случае:право-право
4) нарушение высоты в случае:право-лево

Все эти ситуации разрешаются с помощью левых и правых поворотов деревьев. Повороты БД изображены на рис. 1.

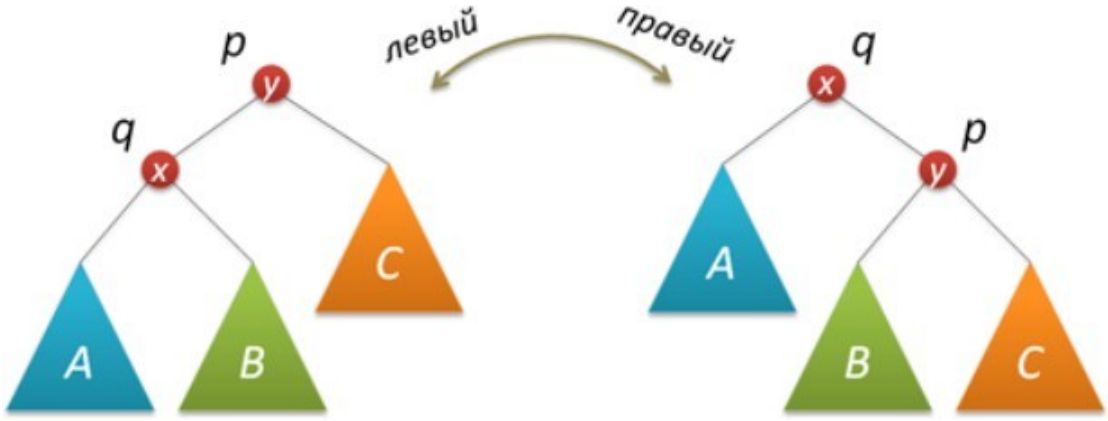


Рисунок 1 – повороты БД

Картинка описывает повороты бинарного дерева.

Описание алгоритма поиска элемента в АВЛ_дереве:

Поиск элемент в АВЛ_дереве происходит как в обычном упорядоченном дереве. Если узел, в котором мы находимся больше заданного значения

переходим к правому сыну, если меньше заданного значения переходим к левому сыну. Алгоритм заканчивает свою работу в 2-х случаях:

1) Если значения узла, в котором мы находимся равно заданному значению
2) Если мы дошли до листа и не встретили элемента равногонашему

1. ОПИСАНИЕ ФУНКЦИЙ И СТРУКТУРДАННЫХ

1) `Class Head_AVL_Tree` –реализована работа с АВЛ-деревом

   A) Методы класса:

      a) `void insert(Type)` –вставка элемента вАВЛ-дерево.

      b) `void print_tree1()` –вывод АВЛ-дерева на экран

      c) `bool is_contain(Type)`–возвращает true, если заданный элементы содержится в АВЛ-дереве.

      d) `Head_AVL_Tree()` – конструктор класса. Инициализирует данные.

      e) `~Head_AVL_Tree()` – деструктор класса. Очищает памятьвыделенную под АВЛ-дерево.

   B) Данныекласса:

      a) `class Node_AVL_Tree<Type>* head` –содержит указатель на АВЛ-дерево.

2) `class Node_AVL_Tree` – Реализована работа с элементами АВЛ-дерева.

   A) Методы класса:

      a) `bool is_contain(Type)` - возвращает true, если заданныйэлементы содержится в АВЛ-дереве.

      b) `int set_height()` – устанавливает высоту данного элемента АВЛ-Дерева.

      c) `int get_balance()` –Получает значение баланса для заданного элементаАВЛ-дерева.

      d) `void print_tree(int)` –выводит на экран АВЛ-дерево

      e) `class Node_AVL_Tree<Type>* insert(Type)` –вставка в АВЛ-дерево

      f) `class Node_AVL_Tree<Type>* left_rotate()` – левое вращение

g) `class Node_AVL_Tree<Type>* right_rotate()` – правое вращение
h) `class Node_AVL_Tree<Type>* make_balance()` – установка баланса путем вращения дерева вокруг элемента
i) `Node_AVL_Tree()` –конструктор класса. Инициализирует данные
j) `~Node_AVL_Tree()` – деструктор класса. Очищает память выделенную под АВЛ-дерево.

В) Данныекласса:

a) `int height` –определяет высоту дерева.
b) `int balance` –определяет баланс дерева.
c) `Type data` –данные дерева.
d) `class Node_AVL_Tree<Type>* left` –левый нод
e) `class Node_AVL_Tree<Type>* right`-правый нод

2. ТЕСТИРОВАНИЕ

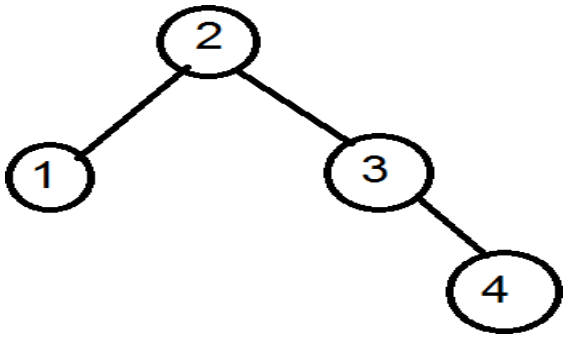
Тестирование:

Входные данные: 1 2 3

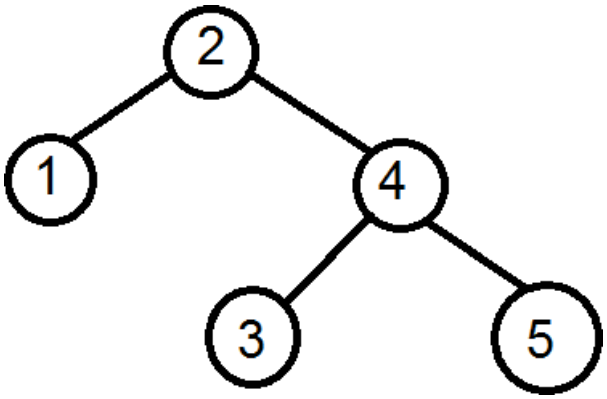
Рассматриваемые входные данные в обычном БД расположились бы линейно, что увеличило бы количество итераций в поисковом алгоритме.

Допустим, на добавление подано число 2. Тогда программа выдаст сообщение “element already in tree”.

Рассмотрен другой случай. На вход подано число 4, дерево будет выглядеть следующим образом:



Для более подробного разбора теста добавлено еще одно значение: 5



Произошел поворот вправо вокруг 3

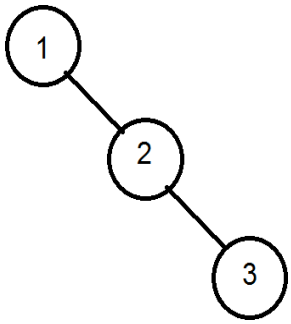
Всего различаются 4 различных случая для вставки элемента:

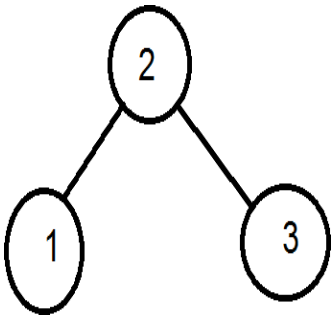
На следующих рисунках: левое дерево – исходное дерево.

правое дерево – видоизменённое дерево.

1) Balance == 2, right->balance ==1

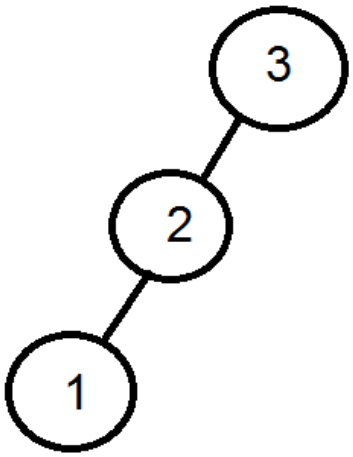
   Решается с помощью левого поворота вокруг 1

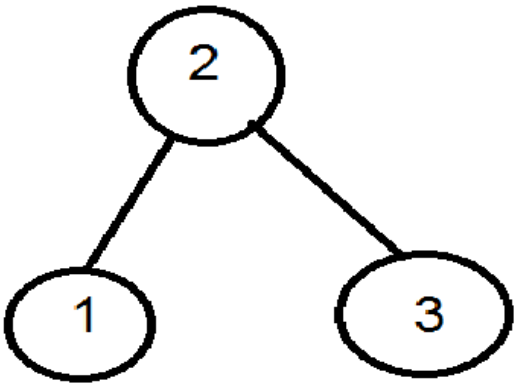




2) Balance == -2, left->balance ==-1

Решается с помощью правого поворота вокруг 3

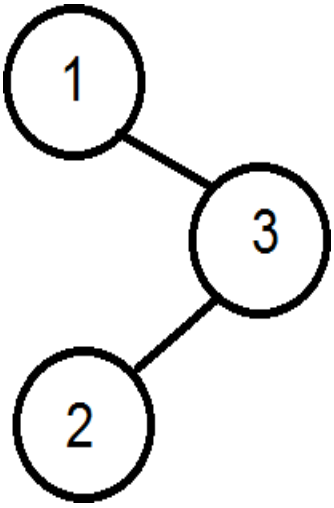


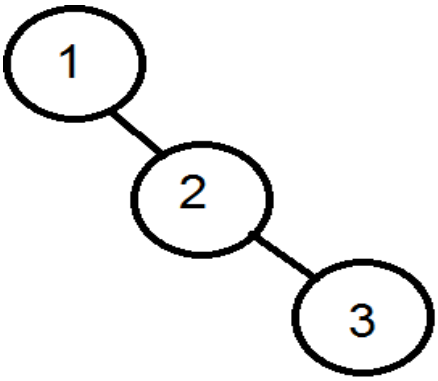


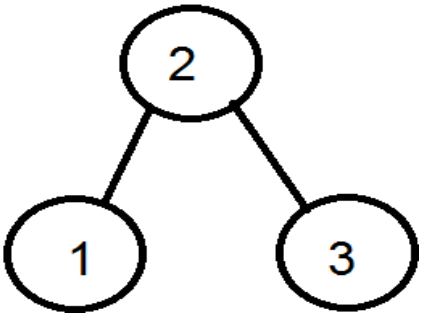
3) Balance == 2 right->balance == -1

Решается с помощью двух поворотов:

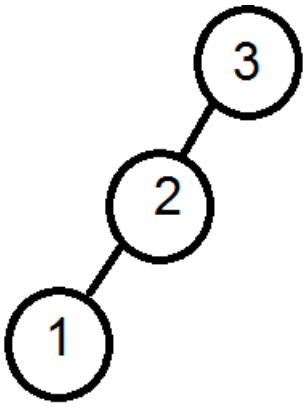
Сначала вокруг 3, затем вокруг1


1
3
2


1
2
3

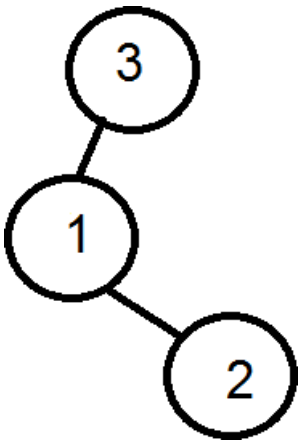

2
1
3

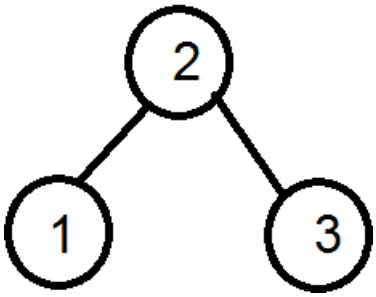
Balance == -2, left->balance ==1



Приведены некоторые ошибки, нуть при работе программы. Данные ошибки приведе

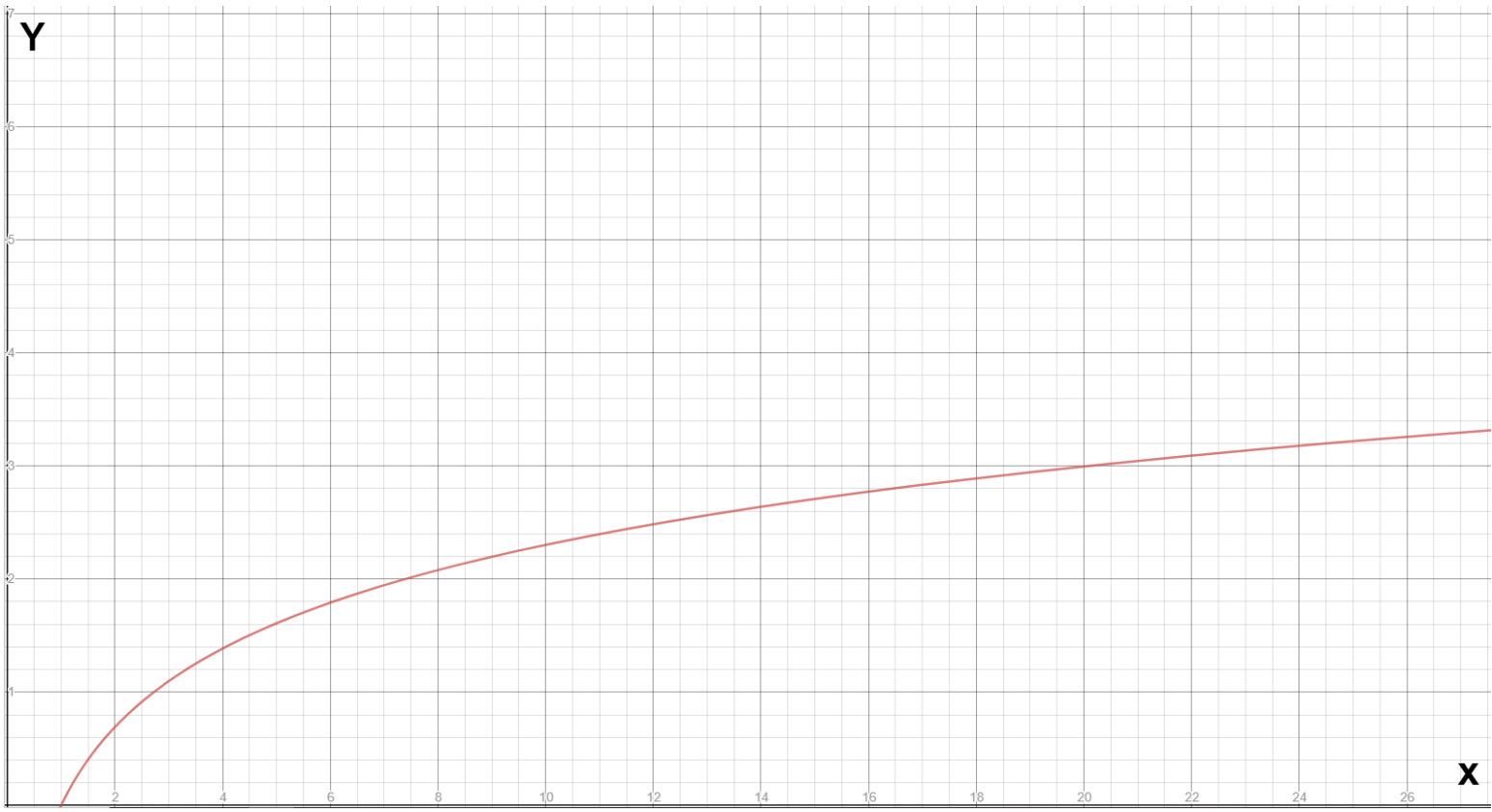
Таблица 1 – Реакция программы на н





ЗАКЛЮЧЕНИЕ

В работе изучена структура данных сбалансированное дерево, на примере АВЛ-дерева. Сбалансированные деревья позволяют выполнять вставку, удаление и поиск заданного элемента с логарифмической асимптотической сложностью. В отличие от деревьев. Маловероятно, что дерево из N элементов будет иметь высоту N и все же такая вероятность возможна. Тогда вставка, поиск и удаление элементов будет происходить с линейной сложностью, и такие деревья будут уступать по скорость элементарному линейному списку.



ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ФАЙЛА MAIN.CPP

```
  #include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <map>
#include <algorithm>
#include <memory>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cstring>
#include <QMainWindow>
#include <QGraphicsItem>
#include <QGraphicsView>
#include <QGraphicsEffect>
#include <QFileDialog>
#include <QStandardPaths>
#include <QtGui>
#include <QLabel>
#include <QString>
#include <QColorDialog>
#include <QInputDialog>
#include <QMainWindow>
#include <QPushButton>
#include <QMessageBox>
#include <QStringList>
#include <QTextEdit>
#include <stack>
 #include <QTextEdit>
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "Tree.h"
bool started = true;
std::string Name;
int action = 0;
typedef int Type;

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),

    ui(new Ui::MainWindow)
{

    ui->setupUi(this);

    scene = new QGraphicsScene;
    ui->graphicsView->setScene(scene);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButton_clicked()
```

```
{
    text_sreen = new QTextEdit;

    Head_AVL_Tree<Type> head;

        std::string main_str;
        std::ifstream file;

        file.open(Name);
        if (!file.is_open()) {
            std::cout << "Error! File isn't open" << std::endl;
            return;
        }
        getline(file, main_str);
        std::cout << "Input: " << main_str << std::endl;

        int count_num = 0;
        for(size_t i = 0; i < main_str.size(); i++){
            int temp = 0;
            bool is_num = false;
            for(size_t j = i; isdigit(main_str[j]); j++, i++){ // create tree from
string
                temp = temp*10 + main_str[j]-'0';
                is_num = true;
            }
            if(is_num){
                count_num++;
                head.insert(temp);
            }

        }

      head.print_tree1(scene);

        if(!count_num){
            std::cout << "no one element to tree. Program will end" << std::endl;
            exit(0);
        }

        std::cout << "add something? ";
        for( ; ; )
        {
        QApplication::processEvents();
        if(!started)break;

        }
        started = true;

        while(1){
            switch (action) {
                case 0:

                    exit (0);
                    break;
                case 1:
                    Type insert_tree_element;
                    QString str= ui->textEdit->toPlainText ();
                    insert_tree_element  = str.toInt();
```

```
                        std::cout << "add to tree new element: " <<
insert_tree_element  << std::endl;
                    /*if(std::cin.fail()){
                        std::cout << "input error. May be it`s not a number or it`s
very big number" << std::endl;
                        exit(0);
                    }*/

                    std::cout << std::endl << "find this element in AVL_tree: " <<
std::endl;

                    //pri(scene,head);

                    if(!head.is_contain(insert_tree_element)){
                        std::cout << "tree does not have this element" << std::endl
<< std::endl;
                        head.insert(insert_tree_element);
                        std::cout << "THREE AFTER ADD NEW ELEM:" << std::endl;

                        head.print_tree1(scene);
                    }
                    else
                        std::cout << "element already in tree" << std::endl <<
std::endl;

                       action = 0;
                       std::cout << "More action? ";

                       for( ; ; )
                       {
                       QApplication::processEvents();
                       if(!started)break;

                       }
                       started = true;

                   /*if(std::cin.fail()){
                        std::cout << "input error. May be it`s not a number or it`s
very big number" << std::endl;
                        exit(0);
                    }*/
                    break;
}
}
}

void MainWindow::on_pushButton_2_clicked()
{
    QString FileName = QFileDialog::getOpenFileName(this, "OpenDialog",
    QDir::homePath(), "*.txt;; *.*");
    Name = FileName.toStdString();
}

void MainWindow::on_pushButton_3_clicked()
{
    started=false;
    action = 1;
}
```

```
void MainWindow::on_pushButton_4_clicked(){
    started=false;
    action = 0;
}
```

Файл Tree.H

```
  #ifndef AVL_TREE_H //
   for include only one
   time
 #define AVL_TREE_H
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <map>
#include <algorithm>
#include <memory>
#include <iostream>
#include <fstream>
#include <sstream>
#include <QMainWindow>
#include <QGraphicsItem>
#include <QGraphicsView>
#include <QGraphicsEffect>
#include <QFileDialog>
#include <QStandardPaths>
#include <QtGui>
#include <QLabel>
#include <QColorDialog>
#include <QInputDialog>
#include <QMainWindow>
#include <QPushButton>
#include <QMessageBox>
#include <QStringList>
#include <QTextBrowser>
#include <QTextEdit>
#include <stack>

// вспомогательные функции для печати //
void make_deep(int deep, int flag){
  if(flag)
    for(int i = 0 ; i < deep ; i++)
      std::cout << "        ";
  else{
    for(int i = 0 ; i < deep-1 ; i++)
      std::cout << "        ";
    if(deep > 0)
      std::cout << "--------";
  }
}

void make_hight(int deep){
  make_deep(deep , 0);
  std::cout << '|' << std::endl;
  make_deep(deep , 1);
  std::cout << '|' << std::endl;
  make_deep(deep , 1);
}
```

```
//template <T> friend std::istream & operator>> <>(std::istream & is, Vector<T, N>
& v)
//friend const Vector<T, N> & operator+(const Vector<T, N> & v1, const Vector<T, N>
& v2);

 template <class Type>
 class Head_AVL_Tree;

 template <class Type>
 class Node_AVL_Tree{
 public:
     friend class Head_AVL_Tree<Type>; // дружим с головой
      //template <Type> friend int countDeep(Node_AVL_Tree<Type>* node);
      //template <Type> friend  QGraphicsScene *graphic(Node_AVL_Tree<Type>* head,
QGraphicsScene *&scene);
      //template <Type> friend int paint(QGraphicsScene *&scene,
Node_AVL_Tree<Type>* node, int width, int height, int w, int h, QPen &pen, QBrush
&brush, QFont &font, int depth);

     bool is_contain(Type, int);
     int set_height();
     int set_balance();
     void print_tree();
     void print_tree(int);

     class Node_AVL_Tree<Type>* insert(Type);
     class Node_AVL_Tree<Type>* remove(Type);
     class Node_AVL_Tree<Type>* remove_min();
     class Node_AVL_Tree<Type>* find_min();
     class Node_AVL_Tree<Type>* left_rotate();
     class Node_AVL_Tree<Type>* right_rotate();
     class Node_AVL_Tree<Type>* make_balance();
     class Node_AVL_Tree<Type>* get_r();
     class Node_AVL_Tree<Type>* get_l();
     Type get_d();
     int countDeep(Node_AVL_Tree<Type>*);
     Node_AVL_Tree();
     ~Node_AVL_Tree();

 private:

     int height;
     int balance;
     Type data;
     class Node_AVL_Tree<Type>* left;
     class Node_AVL_Tree<Type>* right;
 };

 template <class Type>
 class Head_AVL_Tree{
 public:
     Head_AVL_Tree();
     ~Head_AVL_Tree();
     void insert(Type);
     void print_tree();
     bool is_contain(Type);
     void remove(Type);
     void print_tree1(QGraphicsScene *&);
 private:
```

```
    class Node_AVL_Tree<Type>* head;
};

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::get_r(){
    return right;
}

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::get_l(){
    return left;
}

template <class Type>
Type Node_AVL_Tree<Type>::get_d(){
    return data;
}

template <class Type>
Node_AVL_Tree<Type>::Node_AVL_Tree(){
    left = nullptr;
    right = nullptr;
}

template <class Type>
Node_AVL_Tree<Type>::~Node_AVL_Tree(){ // очищаем память под дерево
    if(left)
        delete left;
    if(right)
        delete right;
}

template <class Type>
bool Node_AVL_Tree<Type>::is_contain(Type desired, int depth){
    if(data == desired)
        return true;
    if(left && data > desired){
        std::cout << "find in left" << std::endl;
        if(left->is_contain(desired, depth+1))
            return true;
    }
    if(right && data < desired){
        std::cout << "find in right" << std::endl;
        return right->is_contain(desired, depth+1);
    }
    return false;
}

template <class Type>
void Node_AVL_Tree<Type>::print_tree(){ // печатает информацию об узла дерева
    std::cout << data << " ";
    std::cout << height << " ";
    std::cout << balance << std::endl;
    if(left)
        left->print_tree();
    if(right)
        right->print_tree();
}
```

```
template <class Type>
void Node_AVL_Tree<Type>::print_tree(int deep){ // beautiful print tree
    make_hight(deep);
    std::cout << data << std::endl;
    if(right){
        right->print_tree(deep+1);
    }
    if(left){
        left->print_tree(deep+1);
    }
}

template <class Type>
int countDeep(Node_AVL_Tree<Type>* node)
{
    if (node == nullptr)
        return 0;
    int cl = countDeep(node->get_l());
    int cr = countDeep(node->get_r());
    return 1 + ((cl>cr)?cl:cr);
}

template <class Type>
QGraphicsScene *graphic(Node_AVL_Tree<Type>* head, QGraphicsScene *&scene)
{
    if (head == nullptr)
        return scene;
    scene->clear();
    QPen pen;
    pen.setWidth(5);
    QColor color;
    color.setRgb(192, 192, 192);
    pen.setColor(color);
    QBrush brush (color);
    QFont font;
    font.setFamily("Helvetica");

    int w_deep = static_cast<int>(pow(2, countDeep(head))+2);
    int h = 60;
    int w = 12;
    font.setPointSize(w);
    int width = (w*w_deep)/2;
    paint(scene, head, width/2, h, w, h, pen, brush, font, w_deep);
    return scene;
}

template <class Type>
void Head_AVL_Tree<Type>::print_tree1(QGraphicsScene *&scene){ // beautiful print
tree
     graphic(head, scene);
}

template <class Type>
int paint(QGraphicsScene *&scene, Node_AVL_Tree<Type>* node, int width, int
height, int w, int h, QPen &pen, QBrush &brush, QFont &font, int depth)
{

    if (node == nullptr)
        return 0;
    QString out;
```

```
     out = QString::number(node->get_d());
     QGraphicsTextItem *elem = new QGraphicsTextItem;
     elem->setPos(width, height);
     elem->setPlainText(out);
     elem->setFont(font);
     scene->addRect(width-w/2, height, w*5/2, w*5/2, pen, brush);
     if (node->get_l() != nullptr)
         scene->addLine(width+w/2, height+w, width-(depth/2)*w+w/2, height+h+w,
pen);
     if (node->get_r() != nullptr)
         scene->addLine(width+w/2, height+w, width+(depth/2)*w+w/2, height+h+w,
pen);
     scene->addItem(elem);
     paint(scene, node->get_l(), width-(depth/2)*w, height+h, w, h, pen, brush,
font, depth/2);
     paint(scene, node->get_r(), width+(depth/2)*w, height+h, w, h, pen, brush,
font, depth/2);
     return 0;
 }

 template <class Type>
 class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::remove(Type to_remove){
     if(data == to_remove){
         if(!left && !right){
             delete this;
             return nullptr;
         }
         if(!right){
             delete this;
             return left;
         }
         class Node_AVL_Tree<Type>* new_root;
         new_root = right->find_min();
         right = right->remove_min();
         new_root->left = left;
         new_root->right = right;
         new_root->height = set_height();
         new_root->balance = set_balance();
         return new_root->make_balance();
     }
     if(data < to_remove)
         right = right->remove(to_remove);
     if(data > to_remove)
         left = left->remove(to_remove);
     height = set_height();
     balance = set_balance();
     return make_balance();
}

 template <class Type>
 class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::find_min(){
     return left?left->find_min():this;
 }

 template <class Type>
 class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::remove_min(){
     if(!left){
         class Node_AVL_Tree<Type>* temp = right;
         this->right = nullptr;
         delete this;
```

```
            return temp;
        }
        left = left->remove_min();
        height = set_height();
        balance = set_balance();
        return make_balance();
    }

    template <class Type>
    class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::insert(Type value){ // вставка
        if(value >= data){
            if(!right){
                right = new Node_AVL_Tree<Type>; // находим нужный узел для вставки
как в обычном упорядоченном дереве
                right->data = value;
                right->height = 1;
            }
            else
                right = right->insert(value);
        }
        if(value < data){
            if(!left){
                left = new Node_AVL_Tree<Type>;
                left->data = value;
                left->height = 1;
            }
            else
                left = left->insert(value);
            }
        height = set_height();
        balance = set_balance();
        return make_balance(); // балансировке по возврату из рекурсии
    }

    template <class Type>
    class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::right_rotate(){
        std::cout << "right rotate around element: " << this->data << std::endl;
        Node_AVL_Tree<Type>* temp;
        temp = left;
        left = temp->right;
        this->height = this->set_height();
        this->balance = this->set_balance();
        if(temp->left){
            temp->left->height = temp->left->set_height();
            temp->left->balance = temp->left->set_balance();
        }
        temp->right = this;
        temp->height = temp->set_height();
        temp->balance = temp->set_balance();
        return temp;
    }

    template <class Type>
    class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::left_rotate(){
        std::cout << "left rotate around element: " << this->data << std::endl;
        Node_AVL_Tree<Type>* temp;
        temp = right;
        right = temp->left;
        this->height = this->set_height();
        this->balance = this->set_balance();
```

```
    if(temp->right){
        temp->right->height = temp->right->set_height();
        temp->right->balance = temp->right->set_balance();
    }
    temp->left = this;
    temp->height = temp->set_height();
    temp->balance = temp->set_balance();
    return temp;
}

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::make_balance(){
    Node_AVL_Tree<Type>* temp;
    temp = this;
    if(balance == 2){
        if(right->balance == -1)
            temp->right = right->right_rotate(); // случай право право
        temp = left_rotate(); // случай лево лево
    }
    if(balance == -2){
        if(left->balance == 1)
            temp->left = left->left_rotate(); // случай право лево
        temp = right_rotate(); // случай право право
    }
    return temp;
}

template <class Type>
int Node_AVL_Tree<Type>::set_height(){ // установка высоты
    if(!left && !right) // if sheet
         return 1;
    if(!left)
        return (right->height + 1);
    if(!right)
        return (left->height + 1);
    if(left->height >= right->height)
        return (1 + left->height);
    if(left->height < right->height)
        return (1 + right->height);
    return 0; // чтобы компилятор не ругался
}

template <class Type>
int Node_AVL_Tree<Type>::set_balance(){ // установка баланса в вершине
    if(!left && !right) // if sheet
         return 0;
    if(!left)
        return right->height;
    if(!right)
        return (left->height * (-1));
    return (right->height - left->height);
}

template <class Type>
Head_AVL_Tree<Type>::Head_AVL_Tree(){
    head = nullptr;
}

template <class Type>
Head_AVL_Tree<Type>::~Head_AVL_Tree(){
```

```
    delete head;
}

template <class Type>
void Head_AVL_Tree<Type>::print_tree(){
    if(!head){
        std::cout << "tree is empty" << std::endl;
        return;
    }
    head->print_tree(0);
}

template <class Type>
void Head_AVL_Tree<Type>::insert(Type value){
    if(!head){
        Node_AVL_Tree<Type>* temp = new Node_AVL_Tree<Type>;
        temp->data = value;
        temp->height = 1;
        head = temp;
        return;
    }
    head = head->insert(value);
}

template <class Type>
bool Head_AVL_Tree<Type>::is_contain(Type desired){
    if(!head)
        return false;
    if(head->data == desired){
   std::cout << "this element is root" << std::endl;
        return true;
    }
    if(head->left && head->data > desired){
        std::cout << "find in left " << std::endl;
        return head->left->is_contain(desired, 1);
    }
    if(head->right && head->data < desired){
        std::cout << "find in right " << std::endl;
        return head->right->is_contain(desired, 1);
    }
    return false;
}

template <class Type>
void Head_AVL_Tree<Type>::remove(Type to_remove){
    head = head->remove(to_remove);
}

#endif
```