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

ОТЧЕТ

по лабораторной работе
по дисциплине «Алгоритмы и Структуры данных»
Тема: Реализация и исследование АВЛ-деревьев

Студент гр. 2300 _____ Хидда А..

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2023

Цель работы.

Исследовать АВЛ-деревья.

Задание.

а) Реализовать функции удаления узлов: любого, максимального и минимального.

б) Сравнить время и количество операций, необходимых для реализованных операций, с теоритическими оценками на различных объёмах данных.

Выполнение работы.

Создан класс *Node* с его полями и методами:

int val – значение в вершине.

Node left* – указатель на левое поддерево.

Node right* – указатель на правое поддерево.

int height – высота текущего дерева.

Node(int val, Node left = nullptr, Node* right = nullptr)* – конструктор класса.

Реализованы функции:

cNode insert(int val, Node* node)* – вставка ключа k в дерево с корнем p.

Node Remove(Node* p, int k)* – удаление ключа k из дерева p.

Node RemoveMax(Node* p)* – удаление узла с максимальным ключом из p.

Node RemoveMin(Node* p)* – удаление узла с минимальным ключом из p.

int Height(Node p)* – функция высоты для опредленного узла.

int BFactor(Node p)* – данный balance-factor поможет нам в будущем при балансировке деревьев, так как именно по нему мы будем принимать решения того, что нам именно необходимо сделать с деревом.

void FixHeight(Node p)* – восстановить корректность высот в дереве p.

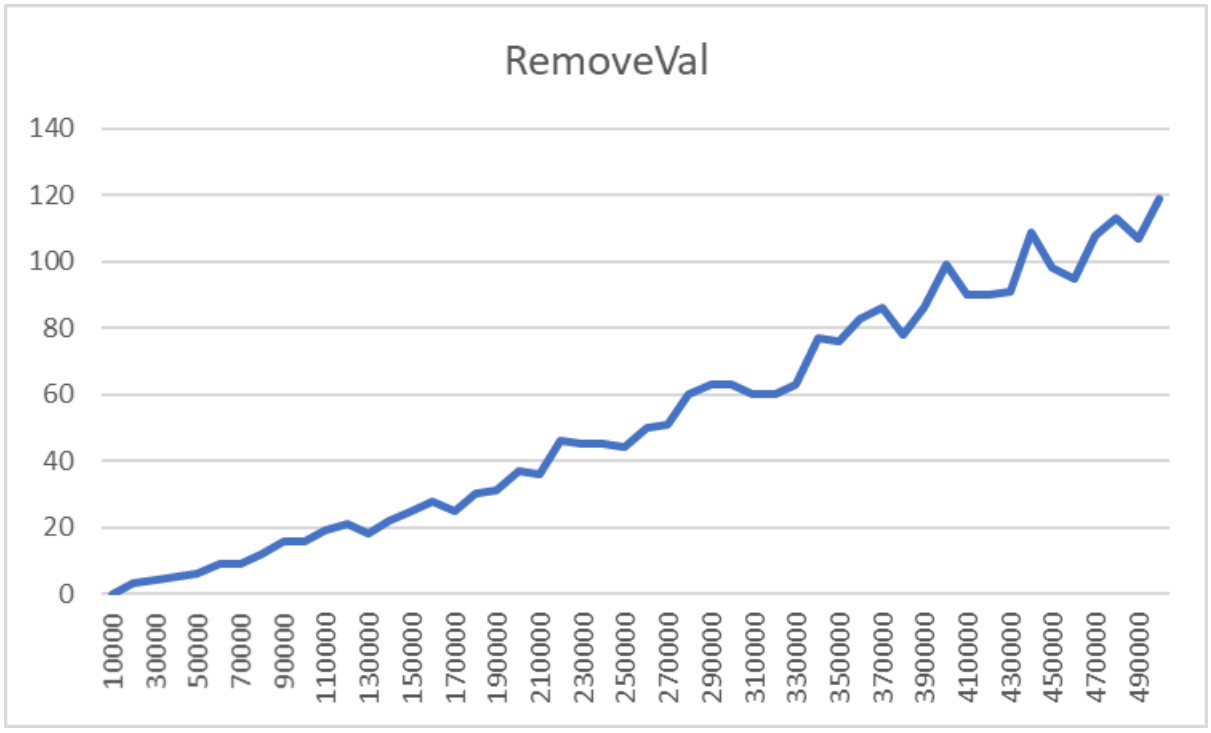
Node RotateRight(Node* p)* – правый поворот вокруг p.

Node RotateLeft(Node* q)* – левый поворот вокруг q.

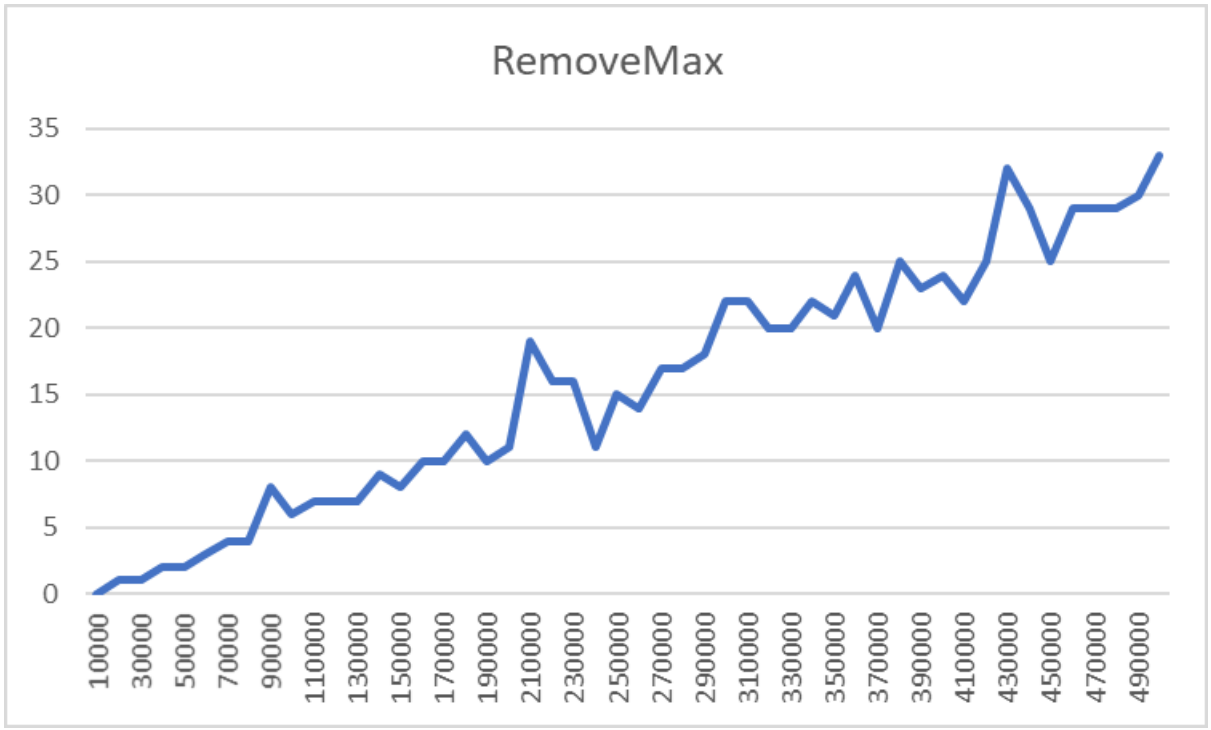
Node balance(Node* p)* – балансировка узла p.

Node FindMin(Node* p)* – поиск узла с минимальным ключом в дереве p.

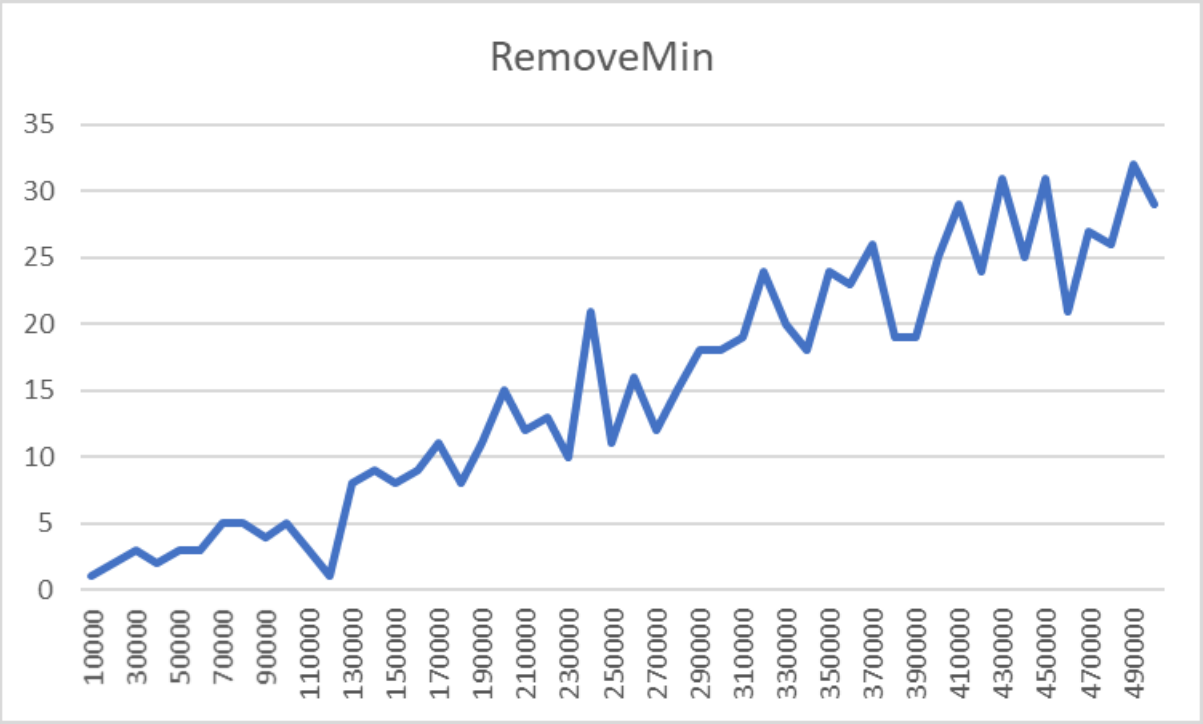
В результате тестировании функции Remove, получена зависимость времени работы от объёма данных:



В результате тестировании функции RemoveMax, получена зависимость времени работы от объёма данных:



В результате тестировании функции RemoveMin, получена зависимость времени работы от объёма данных:



Выводы.

Разработаны требуемые функции. Время их работы напоминает линейную зависимость, это говорит о том, что они подтверждают ожидаемую теоретическую логарифмическую зависимость т.к. она схожа с линейным ростом при определённым значении констант.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: avl_tree.h

```
#pragma once

class Node{
    public:
        int val;
        Node* left;
        Node* right;
        int height;

        Node(int val, Node* left = nullptr, Node* right = nullptr) :
val(val), left(left), right(right), height(1)
        {}
};

Node* insert(int val, Node* node);

Node* Remove(Node* p, int k);

Node* RemoveMax(Node* p);

Node* RemoveMin(Node* p);

int Height(Node* p);

int BFactor(Node* p);

void FixHeight(Node* p);

Node* RotateRight(Node* p);

Node* RotateLeft(Node* q);

Node* balance(Node* p);

Node* FindMin(Node* p);
```

Название файла: avl_tree.cpp

```
#pragma once

#include "avl_tree.h"
#include <iostream>

Node* insert(int val, Node* node){
    if (node == nullptr)
        return new Node(val);
    if (val < node->val)
        node->left = insert(val, node->left);
    else
        node->right = insert(val, node->right);

    return balance(node);
```

```
}

Node* Remove(Node* p, int k){
    if (p == nullptr)
        return 0;
    if (k < p->val){
        p->left = Remove(p->left, k);
    }
    else
        if (k > p->val){
            p->right = Remove(p->right, k);
        }
        else{
            Node* left = p->left;
            Node* right = p->right;
            delete p;
            if (right == nullptr)
                return left;
            Node* min = FindMin(right);
            min->right = RemoveMin(right);
            min->left = left;
            return balance(min);
        }
    return balance(p);
}

Node* RemoveMax(Node* p){
    if (p->right == nullptr)
        return p->left;
    p->right = RemoveMax(p->right);
    return balance(p);
}

Node* RemoveMin(Node* p){
    if (p->left == nullptr)
        return p->right;
    p->left = RemoveMin(p->left);
    return balance(p);
}

int Height(Node* p){
    return p ? p->height : 0;
}

int BFactor(Node* p){
    return Height(p->right) - Height(p->left);
}

void FixHeight(Node* p){
    int h_left = Height(p->left);
    int h_right = Height(p->right);
    p->height = (h_left > h_right ? h_left : h_right) + 1;
}


Node* RotateRight(Node* p){
    Node* q = p->left;
    p->left = q->right;
```

```
    q->right = p;
    FixHeight(p);
    FixHeight(q);
    return q;
}

Node* RotateLeft(Node* q){
    Node* p = q->right;
    q->right = p->left;
    p->left = q;
    FixHeight(q);
    FixHeight(p);
    return p;
}

Node* balance(Node* p){
    FixHeight(p);
    if (BFactor(p) == 2){
        if (BFactor(p->right) < 0)
            p->right = RotateRight(p->right);
        return RotateLeft(p);
    }
    if (BFactor(p) == -2){
        if (BFactor(p->left) > 0)
            p->left = RotateLeft(p->left);
        return RotateRight(p);
    }
    return p;
}

Node* FindMin(Node* p){
    return p->left ? FindMin(p->left) : p;
}
```