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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Бинарные деревья поиска

Студент гр. 9384 _____ Соседков К.С.

Преподаватель _____ Ефремов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить и реализовать структуру данных АВЛ-дерево.

Задание. (Вариант № 16)

БДП: АВЛ-дерево

Для построенной структуры данных проверить, входит ли в нее элемент, и если входит, то удалить этот элемент из структуры данных(первое вхождение).

Выполнение работы.

Для выполнения работы был разработан класс *AVL_Tree.* Класс *AVL_Tree* является сбалансированным бинарным деревом поиска. Для реализации дерева были разработаны следующие методы:

insert(T data) — вставка элемента в дерево(*O(log n)*).

remove(T data) — удаление элемента из дерева(*O(log n))*.

Для поддержания баланса высоты в дереве были разработаны методы поворота дерева *rotate_left* и *rotate_right.* Они вызываются когда баланс дерева нарушается(*abs(left_tree.height-right_tree.height) >= 2*). Что бы узнавать текущий баланс, был написан метод *calculateBalanceFactor*.

Так же были реализованы вспомогательные методы, такие как:

getHeight — получение высоты дерева.

min/max — минимальный/максимальный элемент в дереве.

size — колличество узлов в дереве.

getLevel — получить текущий уровень(глубину) узла.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	1,2,3	2 --1 --3	Левый поворот
2.	1,2,3,4 remove(2)	3 --1 --4	Удаление + поворот
3.	1,2,3,4,5 remove(4)	2 --1 --5 ----3	Удаление

Выводы.

В ходе выполнения лабораторной работы была реализована структура данных АВЛ-дерево, а так же все необходимые методы для работы с ней(вставка, удаление, поиск).

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: avl_tree.h

```
#ifndef AVL_TREE_H
#define AVL_TREE_H
#include <QDebug>
template <class T>
class AVL_Tree
{
private:
    struct Node {
        Node(T data, int level=0) {
            this->data = data;
            this->level=level;
        };
        ~Node() {
            delete this->left;
            delete this->right;
        }

        AVL_Tree* left = nullptr;
        AVL_Tree* right = nullptr;
        T data;
        int level = 0;
    };

    Node* root = nullptr;
    int balance_factor=0;
    AVL_Tree(T data, int level) {
        this->root = new Node(data,level);
    }

    void rotate_left() {
        AVL_Tree<T>* new_root = new AVL_Tree(*root->right);
        AVL_Tree<T>* old_root = new AVL_Tree(*this);
        old_root->root->right = nullptr;
        if(new_root->root->left) {
            old_root->root->right = new_root->root->left;
        }
        else {
            old_root->root->right = nullptr;
        }
        new_root->root->left = nullptr;
        new_root->root->left = old_root;
        Node* tmp = this->root;
        this->root = new_root->root;
        delete tmp;
        this->updateLevel(root->left->root->level);
    }
```

```
    void rotate_right() {
        AVL_Tree<T>* new_root = new AVL_Tree(*root->left);
        AVL_Tree<T>* old_root = new AVL_Tree(*this);
        old_root->root->left = nullptr;
        if(new_root->root->right) {
            old_root->root->left = new_root->root->right;
        }
        else {
            old_root->root->left = nullptr;
        }
        new_root->root->right = nullptr;
        new_root->root->right = old_root;
        Node* tmp = this->root;
        this->root = new_root->root;
        delete tmp;
        this->updateLevel(root->right->root->level);
    }

    int calculateBalanceFactor() {
        int left_height=0;
        int right_height=0;
        if(this->leftChild())
            left_height= this->leftChild()->getHeight();
        if(this->rightChild())
            right_height = this->rightChild()->getHeight();
        return left_height-right_height;
    }

    void updateLevel(int level) {
        if(!root) return;
        this->root->level = level;
        if(this->root->left) {
            this->root->left->updateLevel(level+1);
        }
        if(this->root->right) {
            this->root->right->updateLevel(level+1);
        }
    }

public:
    AVL_Tree() {}

    AVL_Tree(const AVL_Tree & obj) {
        this->balance_factor = obj.balance_factor;
        if(obj.root) {
            this->root = new Node(obj.root->data, obj.root->level);
            if(obj.root->left) {
                this->root->left = new AVL_Tree(*obj.root->left);
            }
            if(obj.root->right) {
                this->root->right = new AVL_Tree(*obj.root->right);
            }
```

```
    }
}

AVL_Tree& operator=(const AVL_Tree & obj) {
    this->balance_factor = obj.balance_factor;
    if(obj.root) {
        Node* tmp = this->root;
        this->root = new Node(obj.root->data, obj.root->level);
        if(obj.root->left) {
            this->root->left = new AVL_Tree(*obj.root->left);
        }
        if(obj.root->right) {
            this->root->right = new AVL_Tree(*obj.root->right);
        }
        delete tmp;
    }
    return *this;
}

~AVL_Tree() {
    if(root) {
        delete root;
    }
    this->root = nullptr;
}
AVL_Tree* leftChild()  {
    if(this->root->left && !this->root->left->root) return nullptr;
    return this->root->left;
}

AVL_Tree* rightChild()  {
    if(this->root->right && !this->root->right->root) return nullptr;
    return this->root->right;
}

int getLevel() const {
    if(!root) return 0;
    return this->root->level;
}

T getData() const {
    return this->root->data;
}

bool isEmpty() {
    return this->root == nullptr;
}

void insert(T data, int level=0) {
    if(!root) {
        root = new Node(data);
        this->root->level = level;
```

```
    }
    if(data <  this->root->data) {
        if(!this->root->left) {
            this->root->left = new AVL_Tree(data, this->root->level+1);
        }
        root->left->insert(data, this->root->level+1);
    }
    else if(data > this->root->data) {
        if(!this->root->right) {
            this->root->right = new AVL_Tree(data, this->root->level+1);
        }
        root->right->insert(data, this->root->level+1);
    }
    this->balance();
}

void balance() {
    this->balance_factor = this->calculateBalanceFactor();
    if(this->balance_factor == 2) {
        if(this->leftChild()->calculateBalanceFactor() < 0) {
            //LR Rotation
            this->leftChild()->rotate_left();
        }
        //LL Rotation
        rotate_right();
    }
    else if(this->balance_factor == -2) {
        if(this->rightChild()->calculateBalanceFactor() > 0) {
            //RL Rotation
            this->rightChild()->rotate_right();
        }
        //RR Rotation
        rotate_left();
    }
}

int size() {
    int count = 0;
    if (this->leftChild()) {
        count += this->leftChild()->size();
     }
     if (this->rightChild()) {
         count += this->rightChild()->size();
     }
     return count;
}

void print(int offset=0) {
    if(!root) return;
    if(this->isEmpty()) return;
    qDebug() << QString(' ').repeated(offset)  << this->getData() << this->root->level;
    if(this->root->left)  this->root->left->print(offset+2);
```

```
    if(this->root->right)  this->root->right->print(offset+2);
}

int getHeight() {
    if(!root) return 0;
    int leftHeight = 0;
    int rightHeight = 0;
    if(this->leftChild()) {
        leftHeight = this->leftChild()->getHeight();
    }
    if(this->rightChild()) {
        rightHeight = this->rightChild()->getHeight();
    }
    if(leftHeight > rightHeight) {
        return leftHeight+1;
    }
    return rightHeight+1;
}

int getBalanceFactor() {
    return this->balance_factor;
}

void remove(T value) {
    if(!root) {
        return;
    }
    if(root->left && value < this->root->data) {
        this->root->left->remove(value);
    }
    else if(root->right && value > this->root->data) {
        this->root->right->remove(value);
    }
    else if(this->root->data == value){
        if(!this->leftChild() || !this->rightChild()) {
            AVL_Tree* tmp = this->leftChild()? this->leftChild(): this->rightChild();
            if(!tmp) {
                delete this->root;
                this->root = nullptr;
            }
            else {
                *this = *tmp;
                this->root->level--;
            }
        }
        else {
            AVL_Tree* tmp = this->rightChild()->min();
            this->root->data = tmp->root->data;
            this->root->right->remove(tmp->root->data);
            this->balance();
        }
    }
```

```
    }

    AVL_Tree* min() {
        return this->leftChild()? this->leftChild()->min(): this;

    }

    AVL_Tree* max() {
        return this->rightChild()? this->rightChild()->max(): this;
    }
};

#endif // AVL_TREE_H
```