

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: AVL-деревья

Студент гр. 9304

Шуняев А.В.

Преподаватель

Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Изучить структуру данных АВЛ-дерево. Реализовать АВЛ-дерево на языке программирования C++.

Задание.

Вариант 15

БДП: АВЛ-дерево.

По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП.

Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то в скольких экземплярах. Добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Описание алгоритма работы программы.

На вход программы подается строка чисел, разделенных пробелами. Числа по очереди добавляются в дерево с помощью функции Insert. Добавление происходит следующим образом. Ключ нового элемента сравнивается с первым элементом. Если он меньше, то сравнивается с первым элементом левого поддерева, иначе с первым элементом правого поддерева. Далее все повторяется пока, пока первый элемент не окажется nullptr. В таком случае, создается на его месте новый элемент с нужным ключом. После этого выполняется балансировка дерева. Если находится элемент с таким же ключом, то его поле counter увеличивается на 1.

Балансировка происходит следующим образом. Пусть корневой элемент будет X, его левое поддерево L и правое P. Тогда, если $P > L$ на 2, то проверяться, если левое поддерево P больше его правого поддерева, то сначала делается поворот вправо относительно P, затем поворот влево относительно X. Если левое поддерево P меньше или равно правому, то просто делается поворот влево относительно X. В случае, если $L > P$ на 2, то проверяется, если левое поддерево

Л меньше правого, то производится производиться поворот влево относительно Л, затем поворот вправо относительно Х. В случае, если левое меньше или равно правому, производится поворот вправо относительно Х.

Поворот вправо осуществляется следующим образом. Левое поддерево копируется в Т (временная переменная), левому поддереву поворачиваемого узла приравнивается правое поддерево Т. Левому поддереву Т присваивается сам поворачиваемый элемент. Выставляются правильные высоты для Т и для поворачиваемого узла. Возвращается Т.

Поворот влево осуществляется следующим образом. Правое поддерево копируется в Т (временная переменная), правому поддереву поворачиваемого узла приравнивается левое поддерево Т. Правому поддереву Т присваивается сам поворачиваемый элемент. Выставляются правильные высоты для Т и для поворачиваемого узла. Возвращается Т.

Высоты выставляются следующим образом. Если высота левого поддерева больше правого, возвращается высота левого поддерева, иначе – правого.

Поиск элементов по заданному ключу осуществляется рекурсивно. Дерево также проверяется с самого начала, с помощью сравнения ключей. Если находится подходящий элемент, то возвращается значение его поля counter, которое хранит количество таких ключей в дереве.

Формат входных и выходных данных

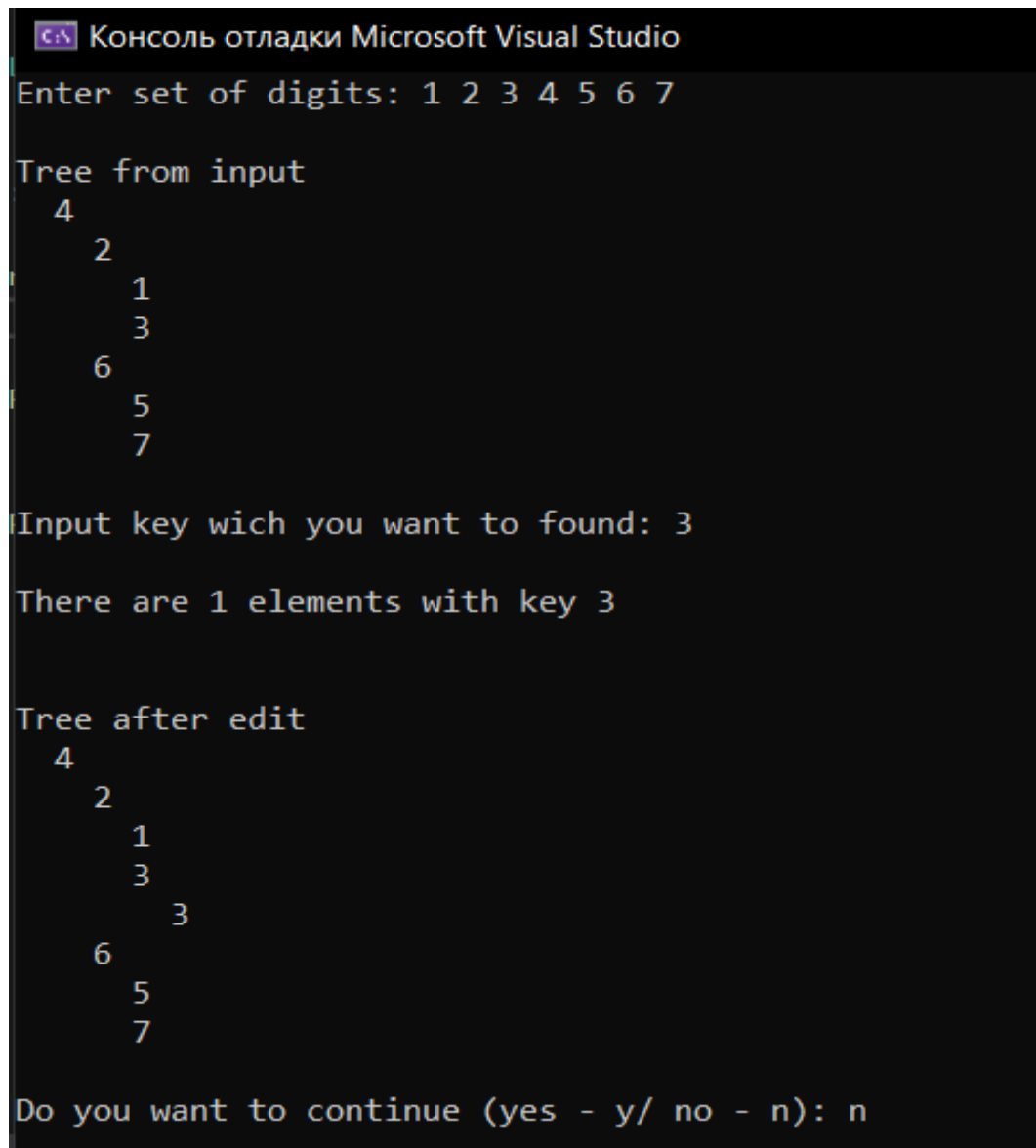
С консоли считывается строка чисел, разделенных пробелом.

Описание основных структур данных и функций (кратко).

- Класс Node – Реализация элемента списка
- Поле key_ класса Node – хранит ключ элемента

- Поле `height_` класса `Node` – хранит высоту дерева. Требуется для вычисления баланс фактора. Он нужен для балансировки дерева. Вычисляется путем вычитания высоты левого поддерева из правого.
- Поле `counter_` класса `Node` – хранит количество элементов с заданным ключом
- Поля `left_` и `right_` класса `Node` – хранят ссылки на левое и правое поддерева
- Класс `AVLTree` – реализация обертки над списком для более удобного использования
- Функция `Insert` – добавляет элемент в дерево
- Функция `Remove` – удаляет элемент из дерева
- Функция `FindMin` – находит наименьший элемент в поддереве (вспомогательная ф-ция для удаления)
- Функция `RemoveMin` – удаляется наименьший элемент из поддерева (вспомогательная функция для удаления)
- Функция `Find` – находит элемент по ключу
- Функция `NodeBalancing` – производит балансировку дерева
- Функция `RotateLeft` – поворачивает дерево влево (нужна в балансировке)
- Функция `RotateRight` – поворачивает дерево вправо (нужна в балансировке)
- Метод `AdaptHeight` – устанавливает правильное значение высоты в элементе
- Метод `BalanceFactor` – находит баланс фактор
- Метод `FindAll` – находит количество элементов с заданным ключом
- Метод `PrintTree` – отрисовывает дерево в консоль

Тестирование.



```
Консоль отладки Microsoft Visual Studio
Enter set of digits: 1 2 3 4 5 6 7

Tree from input
  4
   2
    1
    3
   6
    5
    7

Input key wich you want to found: 3

There are 1 elements with key 3

Tree after edit
  4
   2
    1
    3
   3
   6
    5
    7

Do you want to continue (yes - y/ no - n): n
```

Рисунок 1 – Тестирование программы

Выводы.

В ходе выполнения лабораторной работы была изучена и реализована структура данных АВЛ-дерево на языке программирования C++ с использованием умных указателей.

Была разработана программа, которая по заданной последовательности элементов создает и наглядно выводит АВЛ-дерево на экран, а также находит количество элементов с заданным ключом и добавляется еще один такой же элемент в дерево.

АВЛ-деревья оказались интересными. В данных деревьях высота логарифмически зависит от количества ключей. А так как операции поиска, вставки, удаления линейно зависят от высоты дерева, то получается гарантированная логарифмическая скорость выполнения этих операций.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ.

Файл mail.cpp:

```
#include "AVLTree.h"
#include "Node.h"

int main() {
    char c = 'y';
    int input = 0;
    AVLTree tree;

    std::cout << "\nTree from input\n";
    tree.PrintTree();
    std::cout << std::endl;

    while (c == 'y')
    {
        std::cout << "Input key wich you want to found: ";
        std::cin >> input;

        std::cout << "\nThere are " << tree.FindAll(input) << " elements with key " << input << "\n";
        std::cout << std::endl;

        tree.head_ = Insert(input, tree.head_);

        std::cout << "\nTree after edit\n";
        tree.PrintTree();

        std::cout << "\nDo you want to continue (yes - y/ no - n): ";
        std::cin >> c;
        std::cout << std::endl;
    }

    return 0;
}
```

Файл Node.h:

```
#pragma once
#include "Includes.h"

class Node
{
    friend class AVLTree;
    friend std::shared_ptr<Node> RotateRight(std::shared_ptr<Node> node);
}
```

```

friend std::shared_ptr<Node> RotateLeft(std::shared_ptr<Node> node);
friend std::shared_ptr<Node> NodeBalancing(std::shared_ptr<Node> node);

friend std::shared_ptr<Node> Remove(int key, std::shared_ptr<Node> node);
friend std::shared_ptr<Node> Insert(int key, std::shared_ptr<Node> head);
friend std::shared_ptr<Node> Find(int key, std::shared_ptr<Node> head);

friend std::shared_ptr<Node> FindMin(std::shared_ptr<Node> node);
friend std::shared_ptr<Node> RemoveMin(std::shared_ptr<Node> node);

public:
    Node(int key, std::shared_ptr<Node> left = nullptr, std::shared_ptr<Node> right = nullptr);

    unsigned char GetHeight();
    int GetKey();
    int BalanceFactor();
    void AdaptHeight();

private:
    unsigned char height_;
    int key_;
    std::shared_ptr<Node> left_;
    std::shared_ptr<Node> right_;
};

```

Файл Node.cpp:

```

#include "Node.h"
//Методы класса Node
Node::Node(int key, std::shared_ptr<Node> left, std::shared_ptr<Node> right)
{
    this->key_ = key;
    this->height_ = 1;
    this->left_ = left;
    this->right_ = right;
}

unsigned char Node::GetHeight()
{
    return this ? this->height_ : 0;
}

int Node::GetKey()
{
    return this->key_;
}

int Node::BalanceFactor()

```



```

{
    return (this->right_>GetHeight()) - (this->left_>GetHeight());
}

void Node::AdaptHeight()
{
    unsigned char left_height = this->left_>GetHeight();
    unsigned char right_height = this->right_>GetHeight();

    this->height_ = (left_height > right_height ? left_height : right_height) + 1;
}

////////////////////////////////////

//Удаление
std::shared_ptr<Node> Remove(int key, std::shared_ptr<Node> node)
{
    if (node == nullptr) {
        return nullptr;
    }

    if (key < node->key_) {
        node->left_ = Remove(key, node->left_);
    }
    else if (key > node->key_) {
        node->right_ = Remove(key, node->right_);
    }
    else {
        std::shared_ptr<Node> temp_left = node->left_;
        std::shared_ptr<Node> temp_right = node->right_;

        node.reset();

        if (temp_right == nullptr) {
            return temp_left;
        }
        else {
            std::shared_ptr<Node> min = FindMin(temp_right);

            min->right_ = RemoveMin(temp_right);
            min->left_ = temp_left;

            return NodeBalancing(min);
        }
    }
}

return NodeBalancing(node);

```

```

}

std::shared_ptr<Node> FindMin(std::shared_ptr<Node> node)
{
    return node->left_ ? FindMin(node) : node;
}

std::shared_ptr<Node> RemoveMin(std::shared_ptr<Node> node)
{
    if (node->left_ == nullptr) {
        return node->right_;
    }
    node->left_ = RemoveMin(node->left_);
    return NodeBalancing(node);
}

//Вставка
std::shared_ptr<Node> Insert(int key, std::shared_ptr<Node> head)
{
    if (head == nullptr) {
        return std::make_shared<Node>(key);
    }
    else if (key < head->key_) {
        head->left_ = Insert(key, head->left_);
    }
    else {
        head->right_ = Insert(key, head->right_);
    }

    return NodeBalancing(head);
}

//Поиск
std::shared_ptr<Node> Find(int key, std::shared_ptr<Node> head)
{
    if (head == nullptr) {
        return nullptr;
    }
    else if (key < head->key_) {
        return Find(key, head->left_);
    }
    else if (key > head->key_) {
        return Find(key, head->right_);
    }
    else {
        return head;
    }
}

```

```
}
```

```
////////////////////////////////////
```

```
//Дружественные вспомогательные функции
```

```
std::shared_ptr<Node> RotateRight(std::shared_ptr<Node> node)
```

```
{
```

```
    std::shared_ptr<Node> temp = node->left_;
```

```
    node->left_ = temp->right_;
```

```
    temp->right_ = node;
```

```
    node->AdaptHeight();
```

```
    temp->AdaptHeight();
```

```
    return temp;
```

```
}
```

```
std::shared_ptr<Node> RotateLeft(std::shared_ptr<Node> node)
```

```
{
```

```
    std::shared_ptr<Node> temp = node->right_;
```

```
    node->right_ = temp->left_;
```

```
    temp->left_ = node;
```

```
    node->AdaptHeight();
```

```
    temp->AdaptHeight();
```

```
    return temp;
```

```
}
```

```
std::shared_ptr<Node> NodeBalancing(std::shared_ptr<Node> node)
```

```
{
```

```
    node->AdaptHeight();
```

```
    if (node->BalanceFactor() == 2)
```

```
    {
```

```
        if (node->right_->BalanceFactor() < 0) {
```

```
            node->right_ = RotateRight(node->right_);
```

```
        }
```

```
        return RotateLeft(node);
```

```
    }
```

```
    else if (node->BalanceFactor() == -2)
```

```
    {
```

```
        if (node->left_->BalanceFactor() > 0) {
```

```
            node->left_ = RotateLeft(node->left_);
```

```
        }
```

```

        return RotateRight(node);
    }

    return node;
}

```

Файл AVLTree.cpp:

```

#include "AVLTree.h"
#include "Node.h"

```

```

AVLTree::AVLTree()
{
    int digit;
    std::string str;

    std::cout << "Enter set of digits: ";
    std::getline(std::cin, str);
    std::istringstream iss(str);

    while (iss >> digit) {
        this->head_ = Insert(digit, this->head_);
    }
}

```

```

void AVLTree::PrintTree(int tab, std::shared_ptr<Node> node)
{
    int temp = tab;
    std::string str = "";
    while (temp != 0) {
        str += " ";
        temp--;
    }

    if (node == nullptr) {
        std::cout << str << this->head_->key_ << '\n';
        this->PrintTree(tab + 1, this->head_->left_);
        this->PrintTree(tab + 1, this->head_->right_);
    }
    else {
        std::cout << str << node->key_ << '\n';
        if (node->left_ != nullptr) {
            this->PrintTree(tab + 1, node->left_);
        }
        if (node->right_ != nullptr) {
            this->PrintTree(tab + 1, node->right_);
        }
    }
}

```

```

}

//Задание лабораторной (подсчет количества элементов с ключом K)
int AVLTree::FindAll(int key)
{
    std::shared_ptr<Node> temp;

    temp = Find(key, this->head_);

    if (temp == nullptr) {
        return 0;
    }
    else {
        return 1 + this->FindAll(key, temp->left_) + this->FindAll(key, temp->right_);
    }
}

// Перегруженная функция FindALL (приватная)
int AVLTree::FindAll(int key, std::shared_ptr<Node> node)
{
    std::shared_ptr<Node> temp;

    temp = Find(key, node);

    if (temp == nullptr) {
        return 0;
    }
    else {
        return 1 + this->FindAll(key, temp->left_) + this->FindAll(key, temp->right_);
    }
}

```

Файл AVLTree.h"

```
#pragma once
```

```
#include "Includes.h"
```

```
class Node;
```

```
class AVLTree
```

```
{
```

```
public:
```

```
    std::shared_ptr<Node> head_ = nullptr;
```

```
    AVLTree();
```

```
    void PrintTree(int tab = 1, std::shared_ptr<Node> node = nullptr); //Сдлеать норм вывод
```

```
    int FindAll(int key);
```

```
private:
```

```
    int FindAll(int key, std::shared_ptr<Node> node);
```

```
};
```

Файл Includes.h:

```
#include <memory>
```

```
#include <string>
```

```
#include <sstream>
```

```
#include <iostream>
```