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: АВЛ-дерево

Студент гр. 9304 Аксёнова Е.А.

Преподаватель Филатов Ар.Ю.

Санкт-Петербург
2020

Цель работы.

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

Задание.

Вариант 16.

Бинарное дерево поиска – АВЛ-дерево.

По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП или хеш-таблицу.

Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то удалить элемент e из структуры данных (первое обнаруженное вхождение). Предусмотреть возможность повторного выполнения с другим элементом

Описание алгоритм работы.

В программе объявляется строка, в которую считывается последовательность элементов. К ней применяем функцию, которая её проверяет. Далее данная строка объявляется стандартным потоком, и с неё считывает последовательность в вектор, игнорируя пробелы.

Далее создаем объект класса Tree. С помощью метода print() выводим дерево в консоль. Метод print() вызывает метод printElem(), который выводит элемент дерева с определенным числом символов табуляции в зависимости от «высоты» узла.

Далее программа заходит в цикл, который заканчивается, если был введен элемент не того типа или пользователь написал строку «exit». Программа просит вести элемент, который нужно удалить. Далее вызывается метод findAndDelete(), если введенный элемент был найден в дереве, то он удаляется и выводится дерево после удаления элемента, если нет, то программа выводит сообщение, что элемент не найден и выводится исходное дерево.

Метод findAndDelete() вызывает методы findElem(), если он возвращает true, то вызывается метод deleteElem() и возвращается true, если нет, то возвращает false.

Метод deleteElem() удаляет элемент, а потом в зависимости от наличия поддеревьев либо просто заменяет его на nullptr, либо находит минимальный элемент и балансирует дерево с помощью поворотов, которые вызываются в зависимости от фактора баланса. Затем дерево снова проверяется и балансируется. После чего выводится.

Разработанный программный код см. в приложении А.

Формат входных и выходных данных.

На вход программе подается строка из элементов АВЛ-дерево, которые разделены пробелами. Затем программа принимает элемент, который следует удалить из дерева.

Программа выводит АВЛ-дерево с удаленным элементом. Либо выводит сообщение о том, что удаляемого элемента нет в дереве, и выводит исходное дерево.

Описание основных структур данных и функций.

1. Class Node:

   Поля:

   T data – элемент, хранящийся в узле.

   std::shared_ptr<Node<T>> left – указатель на левое поддерево

   std::shared_ptr<Node<T>> right - указатель на правое поддерево

   int height – высота узла

2. Class Tree:

   Поля:

   std::shared_ptr<Node<T>> root – указатель на корень дерева

int height – высота всего дерева

Методы:

std::shared_ptr<Node<T>> copyTree(std::shared_ptr<Node<T>>) – метод для копирования дерева по узлам

void fixhigh(std::shared_ptr<Node<T>>) – метод для поиска уровня каждого узла

bool findAndDelete(T) – метод, которая запускает поиск и удаление элемента

bool print() – метод, для вывода дерева

std::shared_ptr<Node<T>> createNode(std::vector<T>) – метод для создания дерева

bool findElem(std::shared_ptr<Node<T>>, T) – метод, который ищет элемент по значению

std::shared_ptr<Node<T>> findMin(std::shared_ptr<Node<T>>) – метод, который ищет минимальный элемент дерева

std::shared_ptr<Node<T>> balanceRight(std::shared_ptr<Node<T>>,

std::shared_ptr<Node<T>>) – метод, который балансирует правое поддерево, если оно есть

std::shared_ptr<Node<T>> deleteElem(std::shared_ptr<Node<T>>, T) - метод, который удаляет элемент

std::shared_ptr<Node<T>> rotateLeft(std::shared_ptr<Node<T>>) – метод, реализующий левый поворот

std::shared_ptr<Node<T>> rotateRight(std::shared_ptr<Node<T>>) – метод, реализующий правый поворот

std::shared_ptr<Node<T>> balanceTree(std::shared_ptr<Node<T>>) – метод, который определяет нужно ли балансировать дерево и балансирует его

void printElem(std::shared_ptr<Node<T>>, int) – метод, который выводит дерево

3. Функции:

   void checkStr(std::string&) – функция для проверки введенной строки

Тестирование.

Выводы.

Была изучена структура данных АВЛ-дерево. Данная структура была реализована при помощи языка программирования C++.

Была разработана программа, которая создает АВЛ-дерево по заданному набору элементов и удаляет из него элемент. При реализации классов использовались шаблоны и умные указатели.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab5.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <memory>
#include <sstream>

template<typename T>
class Tree;

template<typename T>
class Node {
public:
    Node(T data) : left(nullptr), right(nullptr), data(data) {
        height = 1;
    }

    Node(std::shared_ptr<Node<T>> left, std::shared_ptr<Node<T>>
right, T data) : left(left), right(right), data(data) {
        height = 1;
    }

private:
    T data;
    std::shared_ptr<Node<T>> left, right;
    int height;

    friend class Tree<T>;

};

template<typename T>
class Tree {
public:
    Tree(std::vector<T> vec) {
        std::sort(vec.begin(), vec.end());
        root = createNode(vec);
        height = root->height;
    }

    std::shared_ptr<Node<T>> copyTree(std::shared_ptr<Node<T>> tree) {
        if (tree->left != nullptr && tree->right != nullptr) {
            std::shared_ptr<Node<T>> node(new
Node<T>(copyTree(tree->left), copyTree(tree->right), tree->data));
```

```
            return node;
        }
        if (tree->left != nullptr && tree->right == nullptr) {
            std::shared_ptr<Node<T>> node(new
Node<T>(copyTree(tree->left), nullptr, tree->data));
            return node;
        }
        if (tree->left == nullptr && tree->right == nullptr) {
            std::shared_ptr<Node<T>> node(new Node<T>(nullptr,
nullptr, tree->data));
            return node;
        }
    }

    void fixhigh(std::shared_ptr<Node<T>> cur) {
        if (cur->left) {
            fixhigh(cur->left);
        }
        if (cur->right) {
            fixhigh(cur->right);
        }
        if (cur->left == nullptr) {
            cur->height = 1;
        }
        else {
            if (cur->right == nullptr) {
                cur->height = cur->left->height + 1;
            }
            else {
                cur->height = cur->left->height > cur->right->height ?
cur->left->height : cur->right->height;
                cur->height++;
            }
        }
    }

    Tree(const Tree<T>& tree) {
        std::cout << "I'm a copy constructor!\n";
        root = copyTree(tree.root);
        fixhigh(root);
        height = root->height;
    }

    Tree(Tree<T>&& tree) {
        std::cout << "I'm a move constructor!\n";
        std::swap(tree.root, root);
    }

    Tree<T>& operator = (const Tree<T>& tree) {
```

```
        root = copyTree(tree.root);
        return *this;
    }

    Tree<T>& operator = (Tree<T>&& tree) {
        root = std::move(tree.root);
        return *this;
    }

    std::shared_ptr<Node<T>> getRoot() {
        return this->root;
    }

    bool findAndDelete(T e) {
        if (findElem(root, e)) {
            root = deleteElem(root, e);
            if (root == nullptr) {
                height = 0;
            }
            else {
                height = root->height;
            }
            return true;
        }
        else {
            return false;
        }
    }

    bool print() {
        if (!this->height) {
            std::cout << "The tree is empty\n";
            return 0;
        }
        std::cout<<"The AVL-Tree:"<<'\n';
        printElem(root, height);
        return 1;
    }

private:
    std::shared_ptr<Node<T>> createNode(std::vector<T> vec) {
        if (vec.size() == 0) {
            return nullptr;
        }
        else {
            int ind = vec.size() / 2;
            auto node = std::make_shared<Node<T>>(vec[ind]);
            std::vector<T> left, right;
            for (int i = 0; i < vec.size(); i++) {
```

```
                if (i < ind) {
                    left.push_back(vec[i]);
                }
                if (i > ind) {
                    right.push_back(vec[i]);
                }
            }
            node->left = createNode(left);
            node->right = createNode(right);
            if (node->left == nullptr) {
                node->height = 1;
            }
            else {
                if (node->right == nullptr) {
                    node->height = node->left->height + 1;
                }
                else {
                    node->height = node->left->height >
node->right->height ? node->left->height : node->right->height;
                    node->height++;
                }
            }
            return node;
        }
    }

    bool findElem(std::shared_ptr<Node<T>> cur, T e) {
        if (cur == nullptr) {
            return false;
        }
        else if (cur->data > e) {
            return findElem(cur->left, e);
        }
        else if (cur->data < e) {
            return findElem(cur->right, e);
        }
        else {
            return true;
        }
    }

    std::shared_ptr<Node<T>> findMin(std::shared_ptr<Node<T>> cur) {
        if (cur == nullptr) {
            return nullptr;
        }
        else if (cur->left == nullptr) {
            return cur;
        }
        else {
```

```
            return findMin(cur->left);
        }
    }

    std::shared_ptr<Node<T>> balanceRight(std::shared_ptr<Node<T>>
cur, std::shared_ptr<Node<T>> min) {
        if (cur == nullptr) {
            return nullptr;
        }
        else if (cur->left == nullptr) {
            return cur->right;
        }
        else if (cur->left == min) {
            cur->left = nullptr;
            return balanceTree(cur);
        }
        else {
            cur->left = balanceRight(cur->left, min);
            return balanceTree(cur);
        }
    }

    std::shared_ptr<Node<T>> deleteElem(std::shared_ptr<Node<T>> cur,
T e) {
        if (cur->data > e) {
            cur->left = deleteElem(cur->left, e);
        }
        else if (cur->data < e) {
            cur->right = deleteElem(cur->right, e);
        }
        else {
            if (cur->right == nullptr) {
                return cur->left;
            }
            else {
                auto min = findMin(cur->right);
                min->right = balanceRight(cur->right, min);
                min->left = cur->left;
                return balanceTree(min);
            }
        }
        return balanceTree(cur);
    }

    std::shared_ptr<Node<T>> rotateLeft(std::shared_ptr<Node<T>> cur)
{
        auto right = cur->right;
        cur->right = right->left;
        right->left = cur;
```

```
        int lHeight = 0, rHeight = 0;
        if (cur->left != nullptr) {
            lHeight = cur->left->height;
        }
        if (cur->right != nullptr) {
            rHeight = cur->right->height;
        }
        cur->height = lHeight > rHeight ? lHeight + 1 : rHeight + 1;
        lHeight = rHeight = 0;
        if (right->left != nullptr) {
            lHeight = right->left->height;
        }
        if (right->right != nullptr) {
            rHeight = right->right->height;
        }
        right->height = lHeight > rHeight ? lHeight + 1 : rHeight + 1;
        return right;
    }

    std::shared_ptr<Node<T>> rotateRight(std::shared_ptr<Node<T>> cur)
{
        auto left = cur->left;
        cur->left = left->right;
        left->right = cur;
        int lHeight = 0, rHeight = 0;
        if (cur->left != nullptr) {
            lHeight = cur->left->height;
        }
        if (cur->right != nullptr) {
            rHeight = cur->right->height;
        }
        cur->height = lHeight > rHeight ? lHeight + 1 : rHeight + 1;
        lHeight = rHeight = 0;
        if (left->left != nullptr) {
            lHeight = left->left->height;
        }
        if (left->right != nullptr) {
            rHeight = left->right->height;
        }
        left->height = lHeight > rHeight ? lHeight + 1 : rHeight + 1;
        return left;
    }

    std::shared_ptr<Node<T>> balanceTree(std::shared_ptr<Node<T>> cur)
{
        int lHeight = 0, rHeight = 0;
        if (cur->left != nullptr) {
            lHeight = cur->left->height;
        }
```

```
    if (cur->right != nullptr) {
        rHeight = cur->right->height;
    }
    cur->height = lHeight > rHeight ? lHeight + 1 : rHeight + 1;
    int diff = rHeight - lHeight;
    if (diff == 2) {
        int diffRight = 0;
        if (cur->right->left != nullptr) {
            diffRight -= cur->right->left->height;
        }
        if (cur->right->right != nullptr) {
            diffRight += cur->right->right->height;
        }
        if (diffRight < 0) {
            cur->right = rotateRight(cur->right);
        }
        return rotateLeft(cur);
    }
    else if (diff == -2) {
        int diffLeft = 0;
        if (cur->left->left != nullptr) {
            diffLeft -= cur->left->left->height;
        }
        if (cur->left->right != nullptr) {
            diffLeft += cur->left->right->height;
        }
        if (diffLeft > 0) {
            cur->left = rotateLeft(cur->left);
        }
        return rotateRight(cur);
    }
    else {
        return cur;
    }
}

void printElem(std::shared_ptr<Node<T>> cur, int level) {
    if (cur->left != nullptr) {
        printElem(cur->left, level - 1);
    }
    for (int i = 0; i < level; i++) {
        std::cout << '\t';
    }
    std::cout << cur->data << '\n';
    if (cur->right != nullptr) {
        printElem(cur->right, level - 1);
    }
}
```

```
    std::shared_ptr<Node<T>> root;
    int height;

};

void checkStr(std::string& str) {
    for (int i = 0; i < str.size(); i++) {
        if (!isdigit(str[i]) && str[i] != ' ') {
            str.erase(i, 1);
            i -= 1;
        }
    }
}

typedef int elem;

int main() {
    std::vector<elem> vec;
    std::string str;
    std::getline(std::cin, str);
    checkStr(str);
    std::stringstream ss(str);
    elem value;
    while (ss >> value) {
        vec.push_back(value);

        if (ss.peek() == ' ') {
            ss.ignore();
        }
    }
    Tree<elem> tree(vec);
    Tree<elem> tree1(tree);
    if (!tree.print()) {
        return 0;
    }
    std::cout <<"This is copied tree:\n";
    if (!tree1.print()) {
        return 0;
    }
    std::string toDelete;
    while (true) {
        std::cout << '\n';
        std::cout << "Input element, that you want to delete:\n";
        std::cin >> toDelete;
        if (toDelete == "exit") {
            break;
        }
        checkStr(toDelete);
        if (!toDelete.size()) {
```

```
            std::cout << "Wrong input\n";
            return 0;
        }
        elem num = std::stoi(toDelete);
        std::cout << '\n';
        if (tree.findAndDelete(num)) {
            if (!tree.print()) {
                return 0;
            }
        }
        else {
            std::cout << num << " is not in tree\n";
            if (!tree.print()) {
                return 0;
            }
        }
    }
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Результаты тестирования представлены в таблице Б.1

Таблица Б.1 — Результаты тестирования

№ п/п	Входные данные	Выходные данные	Результат проверки
1.	1 2 3 4 5 1 2 3	The AVL-Tree: 1 2 3 4 5 The AVL-Tree: 2 3 4 5 The AVL-Tree: 3 4 5	Finished right

		The AVL-Tree: 4 5	
2.	7 5 9 1 5 3 7 5 1 9 5 3 2 4 8 5 4 5	The AVL-Tree: 1 1 2 3 3 4 5 5 5 5 7 7 8 9 9 The AVL-Tree: 1 1 2 3 3	Finished right

		4 5 5 5 7 7 8 9 9 The AVL-Tree: 1 1 2 3 3 5 5 5 7 7 8 9 9 The AVL-Tree: 1 1	

		2 3 3 5 5 7 8 9 9	
3.	3 4 5 6 6 3 2 4 3 exit	The AVL-Tree: 2 3 3 3 4 4 5 6 6	Finished right
4.	6 " 7 * 7 2 9 6 4 3 5 4 1 exit	The AVL-Tree: 2 3 4 4 5 6 6 7 7 9	Finished right

		1 is not in tree The AVL-Tree: 2 3 4 4 5 6 6 7 7 9	
5.	1 1 1 1 1 1 1 1 1 1 1 1 1 1	The AVL-Tree: 1 1 1 1 1 1 1 1 1 1 The AVL-Tree: 1 1 1 1 1 1 1 1 1 1 The AVL-Tree: 1 1 1 1 1 1 1 1	Finished right

		1 The AVL-Tree: 1 1 1 1 1 1 1 The AVL-Tree: 1 1 1 1 1 1	
6.	6	The tree is empty	Finished right