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. 9303 _____ Махаличев Н.А.

Преподаватель _____ Филатов Ар.Ю.

Санкт-Петербург
2020

Цель работы.

Написание бинарного дерева поиска в соответствие с вариантом задания, а также его функционала

Задание.

Вариант 15.

БДП: АВЛ-дерево; действие: 1+2а

1) По заданной последовательности элементов Elem 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 входит, то в скольких экземплярах. Добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

б) 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.

в) Записать в файл элементы построенного БДП в порядке их возрастания; вывести построенное БДП на экран в наглядном виде.

г) Другое действие.

Основные теоретические положения.

АВЛ-деревья – сбалансированные по высоте бинарные деревья поиска, имеющее следующее определение:

$$\text{Т – АВЛ-дерево} \Leftrightarrow \begin{cases} T: |h(T_L) - h(T_R)| \leq 1 \\ T_L \text{ и } T_R - \text{АВЛ} - \text{деревья} \end{cases},$$

где T_L и T_R – левое и правое поддерево соответственно, $h(T)$ – высота дерева.

Ключ любого узла АВЛ-дерева не меньше любого ключа в левом поддереве и не больше любого ключа в правом поддереве, а основной его особенностью является то, что оно является сбалансированным: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу.

При построении дерева используется балансировка – операция, которая в случае разницы высот левого и правого поддеревьев равна двум, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится ≤ 1. Балансировка осуществляется с помощью четырёх видов поворотов:

1) Малое левое вращение используется, когда $(h(b) - h(L)) = 2$ и $h(c) \leq h(R)$. Результат малого левого вращения представлен на рис. 1.

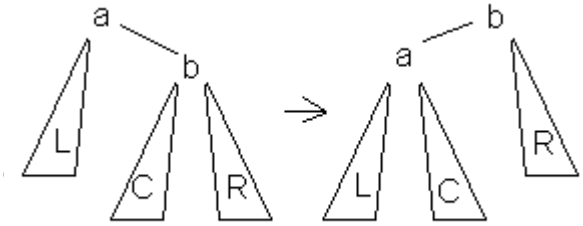

Рисунок 1 – Пример малого левого вращения

2) Большое левое вращение используется, когда $(h(b) - h(L)) = 2$ и $h(c) > h(R)$. Результат малого левого вращения представлен на рис. 2.

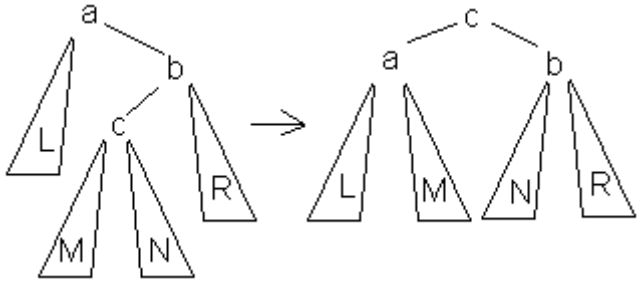

Рисунок 2 – Пример большого левого вращения

3) Малое правое вращение используется, когда $(h(b) - h(R)) = 2$ и $h(c) \leq h(L)$. Результат малого левого вращения представлен на рис. 3.

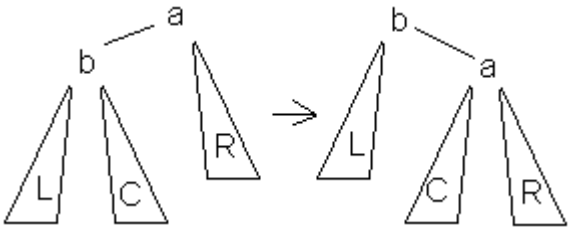

Рисунок 3 – Пример малого правого вращения

4) Большое правое вращение используется, когда $(h(b) - h(R)) = 2$ и $h(c) > h(L)$. Результат малого левого вращения представлен на рис. 4.

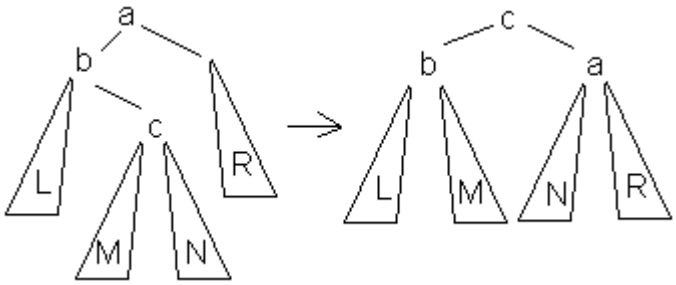

Рисунок 4 – Пример большого правого вращения

Сложность АВЛ-деревьев для поиска, вставки и удаления равна $O(\log_2 n)$.

Выполнение работы.

Создан шаблонный класс Node – узел АВЛ-дерева, в котором определены следующие поля:

- T key – ключ текущего узла;
- int height – высота текущего поддерева;
- Node *left – указатель на левое поддерево;
- Node *right – указатель на правое поддерево.

Создан шаблонный класс AVLTree, в котором определено поле АВЛ-дерева root, а также созданы следующие методы для работы с АВЛ-деревом:

- void Read() – выполняет считывание АВЛ-дерева из терминала;
- Node<T> *MakeNode(Node<T> *tree, T elem) – добавляет узел с ключом elem в дерево tree;
- Node<T> *Balance(Node<T> *tree) – метод, выполняющий балансировку АВЛ-дерева;

- int BalanceFactor(Node<T> *tree) – возвращает разность высот левого и правого поддеревьев АВЛ-дерева tree;
- int Height(Node<T> *tree) – возвращает высоту АВЛ-дерева tree;
- void UpdateHeight(Node<T> *tree) – метод выполняет пересчёт высот поддеревьев дерева tree;
- Node<T> *RotateLeft(Node<T> *tree) – выполняет левый поворот дерева tree;
- Node<T> *RotateRight(Node<T> *tree) – выполняет правый поворот дерева tree;
- void Display(Node<T> *tree, int depth) – выводит АВЛ-дерево tree в терминал;
- void Find(Node<T> *tree, T elem) – выполняет функцию поиска elem среди ключей АВЛ-дерева tree;
- int ElemQuantity(Node<T> *tree, T elem) – метод считает, сколько ключей дерева tree равны значению elem.

Тестирование.

Тестирование программы представлено в приложении Б.

Выводы.

В процессе выполнения лабораторной работы было изучено понятие АВЛ-дерева, его особенности, определена его сложность.

Была разработана программа, строящая АВЛ-дерево из потока данных, а также, в соответствии с вариантом, считающая количество ключей равных входному элементу и добавляющая входной элемент в АВЛ-дерево.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp

```
#include <string>
#include <sstream>
#include <iostream>
#include <stdlib.h>
#include <algorithm>
#define TYPE int

using namespace std;

template<typename T>
class Node{
public:
    Node(T elem);
    T key;
    int height;
    Node *left;
    Node *right;
};

template<typename T>
Node<T>::Node(T elem){
    this->key = elem;
    this->height = 1;
    this->left = nullptr;
    this->right = nullptr;
}

template<typename T>
class AVLTree{
public:
    AVLTree();
    Node<T> *root;
    void Read();
    Node<T> *MakeNode(Node<T> *tree, T elem);
    Node<T> *Balance(Node<T> *tree);
    int BalanceFactor(Node<T> *tree);
    int Height(Node<T> *tree);
    void UpdateHeight(Node<T> *tree);
    Node<T> *RotateLeft(Node<T> *tree);
    Node<T> *RotateRight(Node<T> *tree);
    void Display(Node<T> *tree, int depth);
    void Find(Node<T> *tree, T elem);
    int ElemQuantity(Node<T> *tree, T elem);
};

template<typename T>
AVLTree<T>::AVLTree(){
    root = NULL;
}

template<typename T>
void AVLTree<T>::Read(){
```

```
    TYPE elem = 0;
    string str;
    cout << "Please, enter tree elements: ";
    getline(cin, str);
     istringstream stream(str);
    while (stream >> elem){
        cout << "\n-Inserting " << elem << "...\n" << endl;
        root = MakeNode(root, elem);
        Display(root, 0);
    }
}

template<typename T>
Node<T> *AVLTree<T>::MakeNode(Node<T> *tree, T elem){
     if (tree == NULL) {
        return new Node<T>(elem);
     } else if (elem < tree->key){
           tree->left = MakeNode(tree->left, elem);
     } else {
           tree->right = MakeNode(tree->right, elem);
     }
    return Balance(tree);
}

template<typename T>
Node<T> *AVLTree<T>::Balance(Node<T> *tree){
    UpdateHeight(tree);
     if (BalanceFactor(tree) == -2){
           if (BalanceFactor(tree->left) > 0)
                tree->left = RotateLeft(tree->left);
           return RotateRight(tree);
     }
     if (BalanceFactor(tree) == 2){
           if (BalanceFactor(tree->right) < 0)
                tree->right = RotateRight(tree->right);
           return RotateLeft(tree);
     }
     return tree;
}

template<typename T>
Node<T> *AVLTree<T>::RotateLeft(Node<T> *tree){
    cout << "--Rotating left next part of tree:\n" << endl;
    Display(tree, tree->height);
    Node<T> *newtree = tree->right;
    tree->right = newtree->left;
    newtree->left = tree;
    UpdateHeight(tree);
    UpdateHeight(newtree);
    cout << "\n--Part of tree after left rotating:\n" << endl;
    Display(newtree, newtree->height);
    cout << "\n-----------------------------------\n\n";
    return newtree;
}

template<typename T>
Node<T> *AVLTree<T>::RotateRight(Node<T> *tree){
```

```
    cout << "--Rotating right next part of tree:\n" << endl;
    Display(tree, tree->height);
    Node<T> *newtree = tree->left;
    tree->left = newtree->right;
    newtree->right = tree;
    UpdateHeight(tree);
    UpdateHeight(newtree);
    cout << "\n--Part of tree after right rotating:\n" << endl;
    Display(newtree, newtree->height);
    cout << "\n-----------------------------------\n\n";
    return newtree;
}

template<typename T>
int AVLTree<T>::BalanceFactor(Node<T> *tree){
    return Height(tree->right) - Height(tree->left);
}

template<typename T>
int AVLTree<T>::Height(Node<T> *tree){
    if (tree != NULL){
        return tree->height;
    }
    return 0;
}

template<typename T>
void AVLTree<T>::UpdateHeight(Node<T> *tree){
     tree->height = max(Height(tree->left), Height(tree->right))+1;
}

template<typename T>
void AVLTree<T>::Display(Node<T> *tree, int depth){
    if (tree != NULL){
        for (int i = 0; i < depth; i++){
            cout << ". ";
        }
        cout << tree->key << endl;
        Display(tree->left, depth + 1);
        Display(tree->right, depth + 1);
    }
}

template<typename T>
void AVLTree<T>::Find(Node<T> *tree, T elem){
    int quantity = ElemQuantity(tree, elem);
    if (quantity){
        cout << "-This element is contained " << quantity << " time(s)"
<< std::endl;
    } else {
        cout << "-There is no element = " << elem << std::endl;
    }
}

template<typename T>
int AVLTree<T>::ElemQuantity(Node<T> *tree, T elem){
    if (tree != NULL){
```

```
        if (tree->key == elem){
            return 1 + ElemQuantity(tree->left, elem) +
           ElemQuantity(tree->right, elem);
        } else {
            return ElemQuantity(tree->left, elem) + ElemQuantity(tree-
           >right, elem);
        }
    }
    return 0;
}

int main(){
    AVLTree<TYPE> tree;
    tree.Read();
    TYPE elem = 0;
    string continue_;
    string show_;
    do{
        cout << "\nDo you want to see current tree? ('n' if you don't) -
";
        getline(cin, show_);
        if (show_[0] != 'n'){
            cout << "\n__CURRENT_TREE__" << endl;
            tree.Display(tree.root, 0);
        }
        cout << "\nDo you want to continue? ('n' if you don't) - ";
        getline(cin, continue_);
        if (continue_[0] != 'n'){
            cout << "Please, enter element what you want to find - ";
            cin >> elem;
            cin.ignore();
            tree.Find(tree.root, elem);
            cout << "-Inserting " << elem << "..." << endl;
            tree.root = tree.MakeNode(tree.root, elem);
        }
    } while(continue_[0] != 'n');
    cout << "\n__FINAL_RESULT__" << endl;
    tree.Display(tree.root, 0);
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ ПРОГРАММЫ

Результаты тестирования представлены в табл. 1.

Таблица 1 – Тестирование программы

№ п/п	Входные данные	Выходные данные	Комментарии
1	1 2 3 4 5 1	-There is no element = 4 -Inserting 4... -There is no element = 5 -Inserting 5... -This element is contained 1 time(s) -Inserting 3... __FINAL_RESULT__ 3 . 2 . . 1 . 4 . . 3 . . 5	Получен ожидаемый вывод
2	 1 1	-There is no element = 1 -Inserting 1... -This element is contained 1 time(s) -Inserting 1... __FINAL_RESULT__ 1 . 1	Получен ожидаемый вывод

Продолжение табл. 1

3	10 -2 3 7 -11 14 -11 0 110	-This element is contained 1 time(s) -Inserting -11... -There is no element = 0 -Inserting 0... -There is no element = 110 -Inserting 110... __FINAL_RESULT__ 3 . -11 . . -11 . . -2 . . . 0 . 10 . . 7 . . 14 . . . 110	Получен ожидаемый вывод
4	5 5 5 5 5 5 5 5 5 5 5	-This element is contained 11 time(s) -Inserting 5... __FINAL_RESULT__ 5 . 5 . . 5 . . . 5 . . . 5 . . 5 . . . 5 . . . 5	Получен ожидаемый вывод

		. 5 . . 5 . . 5 . . . 5	