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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: АВЛ-деревья

Студент гр. 9303 _____ Махаличев Н.А.

Преподаватель _____ Филатов Ар.Ю.

Санкт-Петербург

2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Махаличев Никита

Группа 9303

Тема работы: АВЛ-деревья

Исходные данные:

Язык программирования C++, разработка программы под Linux.

Содержание пояснительной записки:

«Содержание», «Введение», «Основные теоретические положения», «Описание классов программы», «Описание интерфейса пользователя», «Тестирование и демонстрация», «Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:

Не менее 19 страниц.

Дата выдачи задания: 06.11.2020

Дата сдачи реферата: 25.12.2020

Дата защиты реферата: 25.12.2000

Студент _____ Махаличев Н.А,

Преподаватель _____ Филатов Ар.Ю.

АННОТАЦИЯ

В данной курсовой работе была разработана программа, строящая АВЛ-дерево и выполняющая функции вставки и удаления элементов в полученное дерево. В программе происходит пошаговая демонстрация шагов во время выполнения операций.

SUMMARY

In this course work, a program was developed that builds an AVL-tree and performs the functions of inserting and removing elements into the resulting tree. The program shows a step-by-step demonstration of steps during operations.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью работы является построение АВЛ-дерева из исходного набора элементов (ключей) с реализацией возможности вставки и удаления элемента в текущее АВЛ-дерево, а также демонстрация происходящих с АВЛ-деревом процессов (вывод в терминал и в файл).

Для достижения поставленной цели необходимо выполнить следующее:

1. Изучить понятие и структуру АВЛ-дерева;
2. Программно реализовать АВЛ-дерево;
3. Реализовать возможность вставки и удаления элементов;
4. Добавить вывод комментариев при выполнении программы.

1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

1.1. АВЛ-дерево

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 больше любого ключа в левом поддереве и меньше 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.

1.2. Балансировка АВЛ-дерева

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.

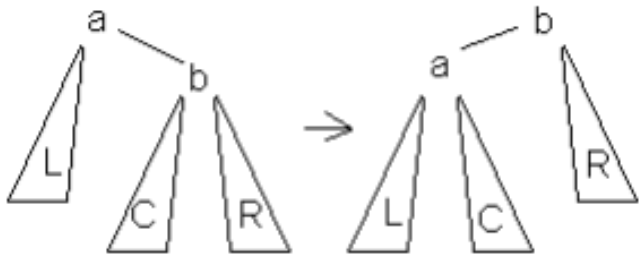

Рисунок 1 – Пример малого левого вращения

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

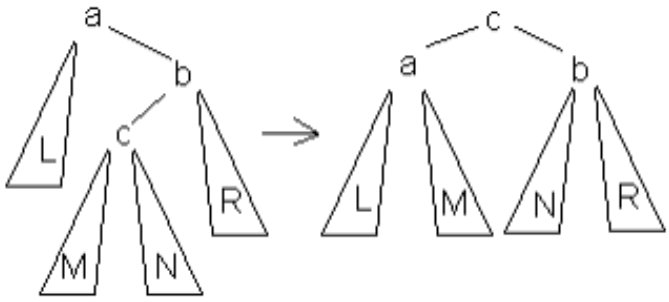

Рисунок 2 – Пример большого левого вращения

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.

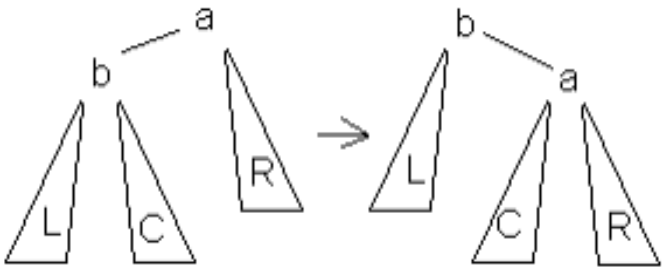

Рисунок 3 – Пример малого правого вращения

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

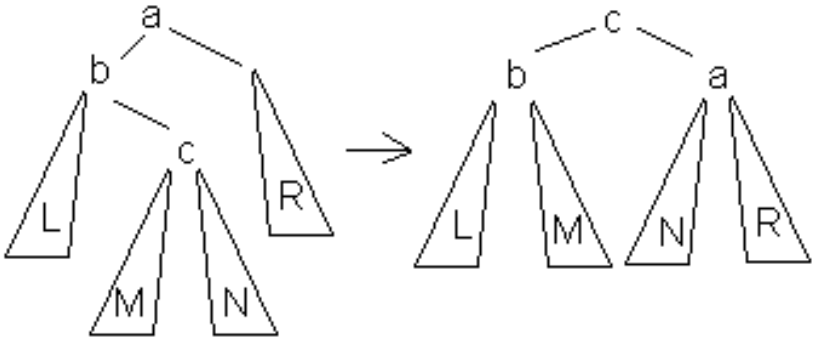

Рисунок 4 – Пример большого правого вращения

1.3. Вставка элемента в АВЛ-дерево

При добавлении элемента в дерево выполняется следующий алгоритм:

1. Спускаемся от корня дерева к листьям по правилу – если элемент меньше ключа текущего дерева, то переходим в левое поддерево, если больше – в правое;
2. Если элемент уже находится в дереве, то элемент не добавляется в дерево, а увеличивается счетчик для данного ключа;
3. Если при спуске по дереву дошли до его конца – создаем лист дерева с ключом, равным данному элементу;
4. Производим балансировку полученного дерева.

1.4. Удаление элемента из АВЛ-дерева

При удалении элемента из дерева выполняется следующий алгоритм:

1. Если существует несколько узлов с данным ключом, уменьшаем счетчик на единицу;
2. Если узел только один, и он является листом – удаляем лист;
3. Если узел с данным ключом только один и он не является листом – ищем ближайший по значению ключ АВЛ-дерева, удалим узел из АВЛ-дерева, заменив его ближайшим по значению ключом;
4. Выполняем балансировку.

2. ОПИСАНИЕ КЛАССОВ ПРОГРАММЫ

В программе реализованы классы Node и AVLTree.

Класс Node создает экземпляр узла и содержит следующие поля:

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

Класс AVLTree является классом с определёнными в нём методами для работы с АВЛ-деревом. Данный класс содержит следующие поля:

- Node<T> *root – указатель на корень АВЛ-дерева;
- ofstream output – файл вывода;
- bool file_output_ – логическая переменная, указывающая, нужен ли вывод в файл или нет;
- map<T, int> counter – счётчик конкретных ключей АВЛ-дерева.

В классе AVLTree определены следующие методы для работы с АВЛ-деревом:

- AVLTree() – конструктор класса AVLTree. Присваивает полю root значение NULL;
- int Height(Node<T> *tree) – возвращает высоту дерева tree;
- int BalanceFactor(Node<T> *tree) – возвращает баланс фактор дерева tree;
- void ReadTree() – метод производит изначальное считывание элементов АВЛ-дерева из терминала и вставляет их в текущее дерево;
- void UpdateHeight(Node<T> *tree) – метод выполняет перерасчёт высоты дерева tree;
- void Find(Node<T> *tree, T elem) – метод показывает, сколько узлов с ключом elem находятся в АВЛ-дереве tree;
- void Display(Node<T> *tree, int depth) – рекурсивный вывод текущего АВЛ-дерева;

- void CombinedOutput(const char *message) – метод выполняет двойной вывод строки в терминал и в файл (при необходимости);
- Node<T> *Balance(Node<T> *tree) – метод выполняет балансировку дерева tree;
- Node<T> *RotateLeft(Node<T> *tree) – малое левое вращение;
- Node<T> *RotateRight(Node<T> *tree) – малое правое вращение;
- Node<T> *MakeNode(Node<T> *tree, T elem) – метод создает узел дерева с ключом, равным elem
- Node<T> *FindMinimal(Node<T> *tree) – находит узел с минимальным ключом в дереве;
- Node<T> *RemoveMinimal(Node<T> *tree) – убирает наименьший элемент из дерева;
- Node<T> *Remove(Node<T> *tree, T elem) – выполняет удаление элемента из дерева tree.

Исходный код программы см. в приложении А.

3. ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

При запуске программа спрашивает у пользователя, нужен ли вывод данных в файл или нет. При вводе «y» все данные, выводящиеся в терминал, будут также выведены в файл output.txt. Пользователь задает начальные ключи АВЛ-дерева через пробел, затем выбирает одну из двух операций: 1 – вставка элемента, 2 – удаление элемента. После выбора операции программа запрашивает ввод элемента, с которым необходимо произвести выбранную операцию. После её выполнения, программа узнает у пользователя, вывести ли ему текущее дерево или нет («y» - вывести), и спрашивает, продолжать ли выполнение операций («y», если продолжить).

Выполнение программы выполняется до тех пор, пока пользователь на последнем шаге не введет значение, отличное от «y».

4. ТЕСТИРОВАНИЕ И ДЕМОНСТРАЦИЯ

Все тестирования представлены и изначально заданными ключами 1, 2, 3, 4, 5, 6, 7, 8.

4.1. Построение АВЛ-дерева

Построение АВЛ-дерева представлено на рис. 5.

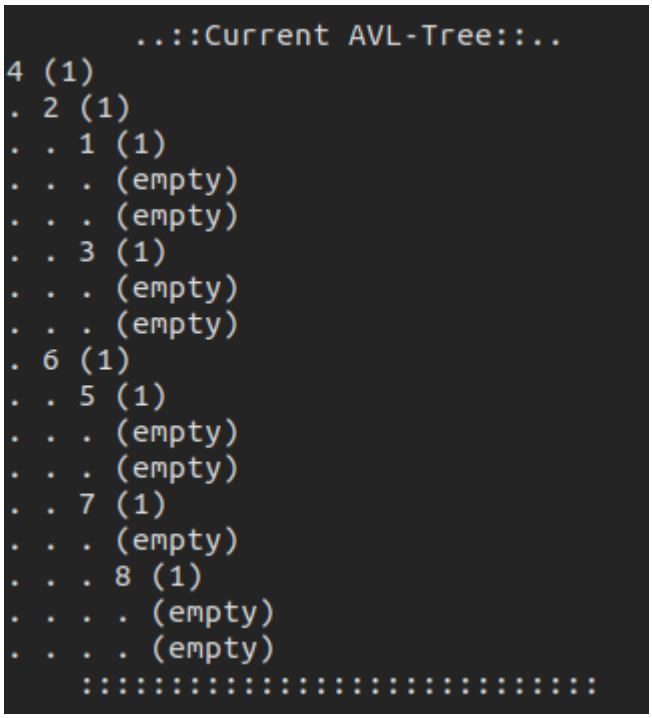

Рисунок 5 – Построение АВЛ-дерева

4.2. Вставка элемента

Вставка элемента в АВЛ-дерево представлена на рис. 6-7.

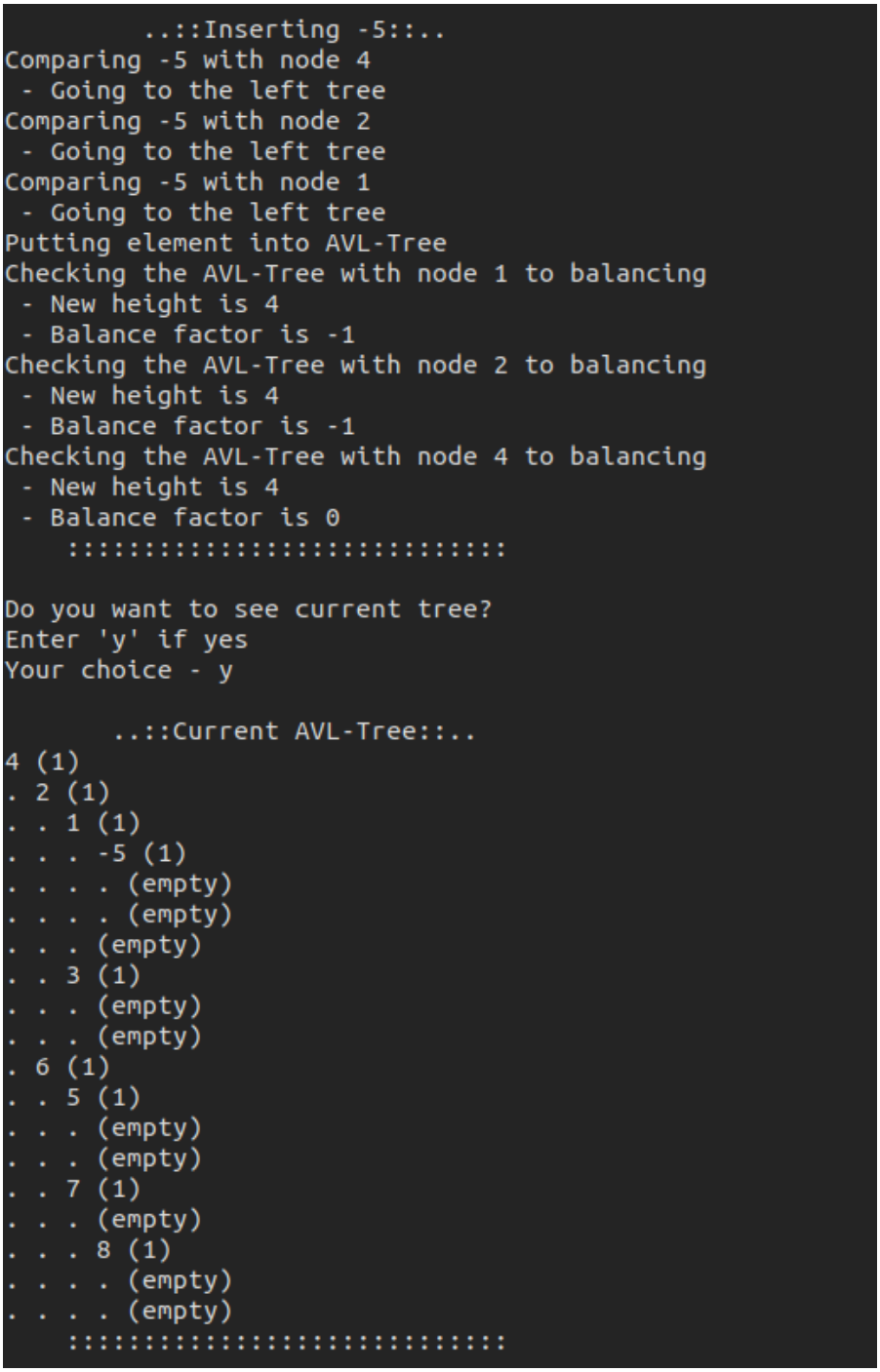

Рисунок 6 – Вставка элемента не содержащегося в АВЛ-дереве (элемент равен -5)

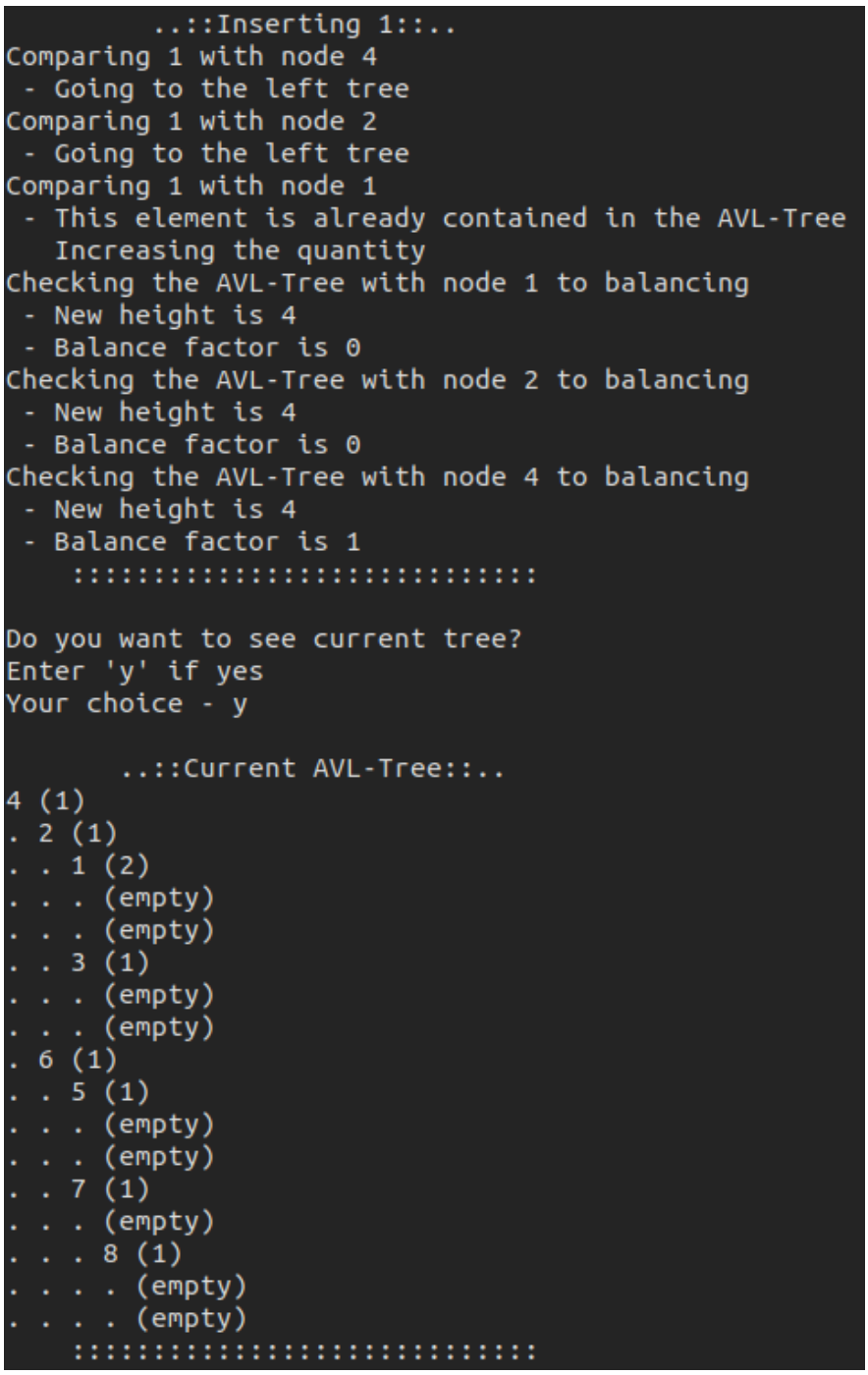

Рисунок 7 – Вставка элемента содержащегося

В АВЛ-дереве (элемент равен 1)

4.3. Удаление элемента

Удаление элемента из АВЛ-дерево представлена на рис. 8-10.

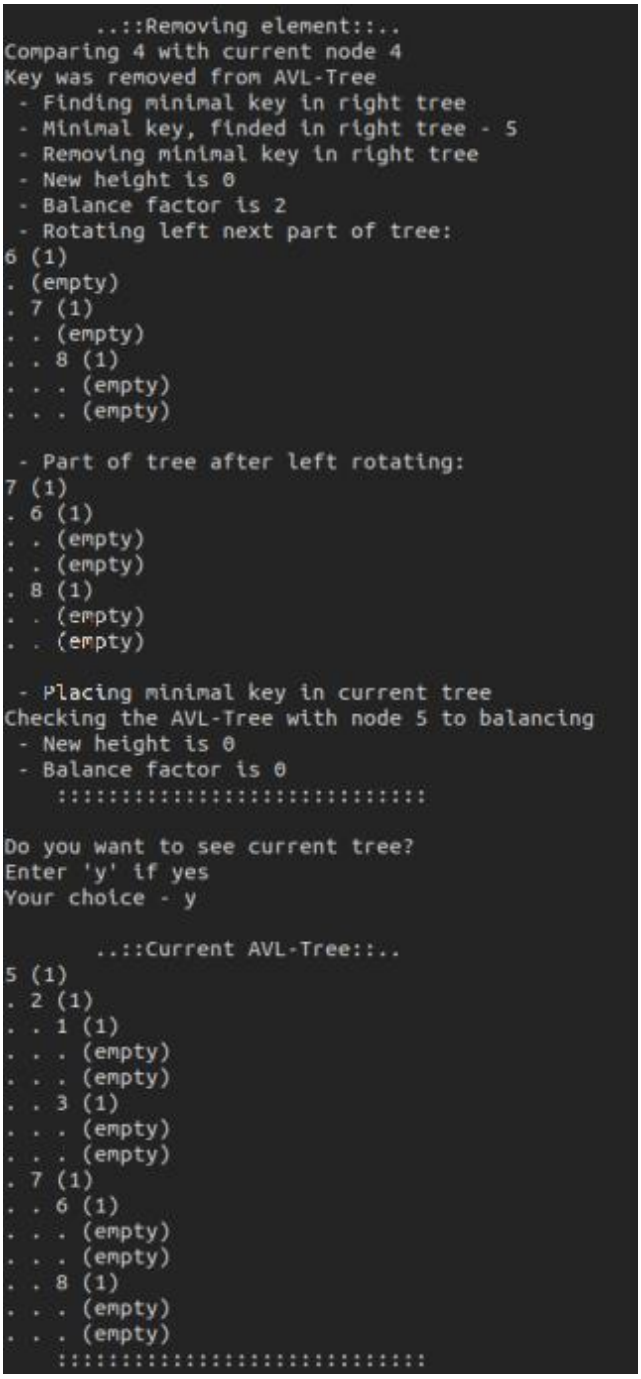

Рисунок 8 – Удаление элемента содержащегося в АВЛ-дереве один раз (элемент равен 4)

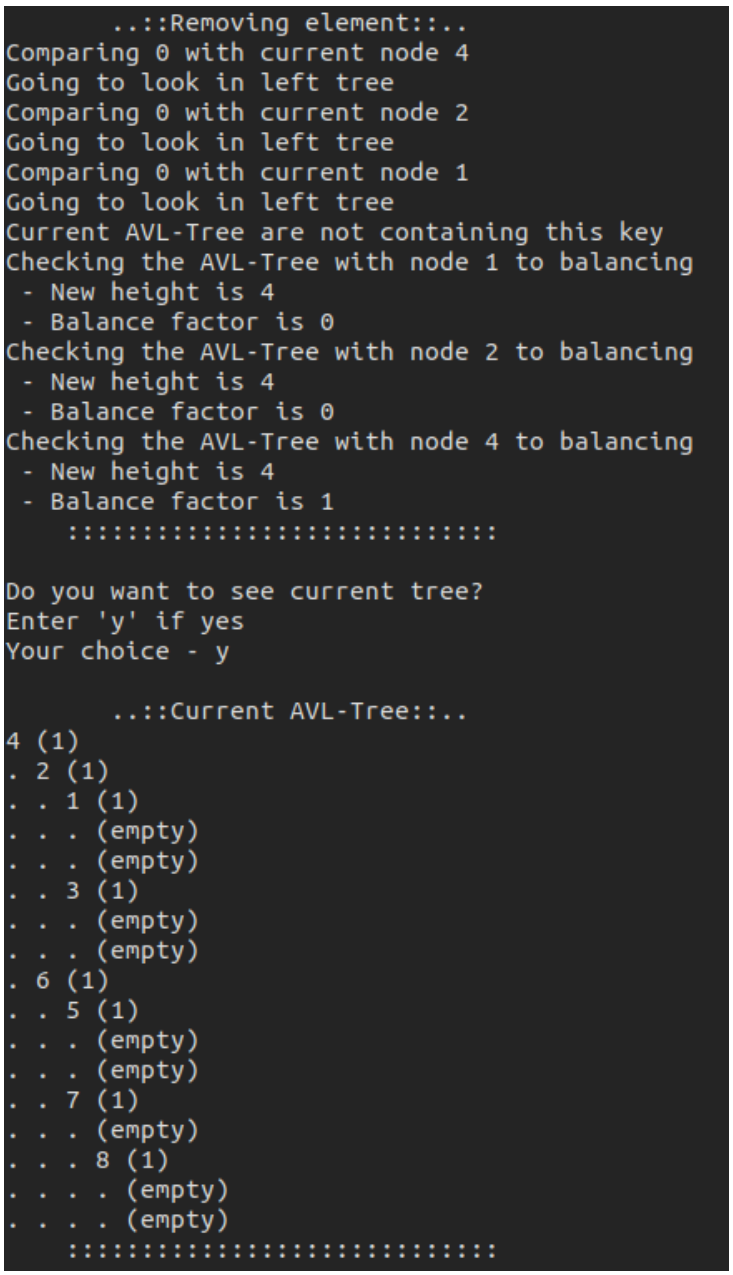

Рисунок 9 – Удаление элемента не содержащегося в АВЛ-дереве (элемент равен 0)

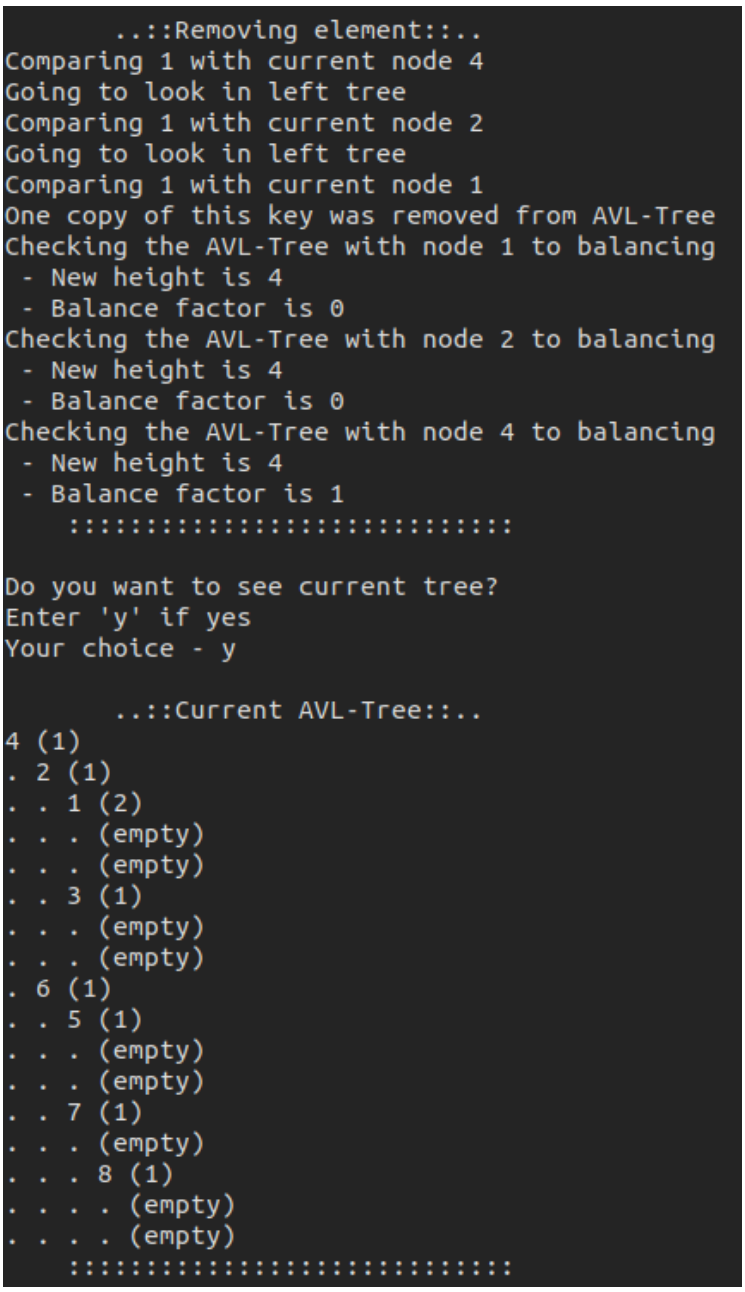

Рисунок 10 – Удаление элемента содержащегося в АВЛ-дереве три раза (элемент равен 1)

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы было изучено понятие АВЛ-дерева, его особенности.

Была разработана программа, строящая АВЛ-дерево из потока данных, а также выполняющая операцию вставки и удаления ключа из текущего АВЛ-дерева. Программа также комментирует и демонстрирует процессы, происходящие с АВЛ-деревом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. АВЛ-дерево – Википедия: https://ru.wikipedia.org/wiki/АВЛ-дерево
2. АВЛ-дерево – Хабр: https://habr.com/ru/post/150732/
3. C++ reference: https://en.cppreference.com/w/

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp

```
#include <map>
#include <string>
#include <sstream>
#include <fstream>
#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:
    Node<T> *root;
    ofstream output;
    bool file_output_;
    map<T, int> counter;
    AVLTree();
    int Height(Node<T> *tree);
    int BalanceFactor(Node<T> *tree);
    void ReadTree();
    void UpdateHeight(Node<T> *tree);
    void Find(Node<T> *tree, T elem);
    void Display(Node<T> *tree, int depth);
```

```
    void CombinedOutput(const char *message);
    Node<T> *Balance(Node<T> *tree);
    Node<T> *RotateLeft(Node<T> *tree);
    Node<T> *RotateRight(Node<T> *tree);
    Node<T> *MakeNode(Node<T> *tree, T elem);
    Node<T> *FindMinimal(Node<T> *tree);
    Node<T> *RemoveMinimal(Node<T> *tree);
    Node<T> *Remove(Node<T> *tree, T elem);
};

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

template<typename T>
void AVLTree<T>::ReadTree(){
    CombinedOutput("  ..::Reading the starting tree::..  \n");
    TYPE elem = 0;
    string str;
    CombinedOutput("Please, enter tree elements: ");
    getline(cin, str);
    if (file_output_){
        output << str << "\n";
    }
     istringstream stream(str);
    while (stream >> elem){
        cout << "\n         ..::Inserting " << elem << "::..        \n";
        if (file_output_){
            output << "\n       ..::Inserting " << elem << "::..       \n";
        }
        root = MakeNode(root, elem);
        CombinedOutput("    ::::::::::::::::::::::::::::    \n");
        CombinedOutput("\n       ..::Current AVL-Tree::..       \n");
        Display(root, 0);
        CombinedOutput("    ::::::::::::::::::::::::::::    \n");
    }
}

template<typename T>
Node<T> *AVLTree<T>::MakeNode(Node<T> *tree, T elem){
     if (tree == NULL) {
        cout << "Putting element into AVL-Tree" << "\n";
        counter[elem] = 1;
        return new Node<T>(elem);
     } else {
        cout << "Comparing " << elem << " with node " << tree->key << "\n";
        if (file_output_){
            output << "Comparing " << elem << " with node " << tree->key
<< "\n";
```

```
        }
        if (elem < tree->key){
            CombinedOutput(" - Going to the left tree\n");
           tree->left = MakeNode(tree->left, elem);
     } else if (elem > tree->key){
            CombinedOutput(" - Going to the right tree\n");
           tree->right = MakeNode(tree->right, elem);
     } else {
            CombinedOutput(" - This element is already contained in the
AVL-Tree\n");
            CombinedOutput("   Increasing the quantity\n");
            counter.at(elem)++;
        }
    }
    cout << "Checking the AVL-Tree with node " << tree->key << " to
balancing\n";
    if (file_output_){
        output << "Checking the AVL-Tree with node " << tree->key << " to
balancing\n";
    }
    return Balance(tree);
}

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

template<typename T>
Node<T> *AVLTree<T>::RotateLeft(Node<T> *tree){
    CombinedOutput(" - Rotating left next part of tree:\n");
    Display(tree, 0);
    Node<T> *newtree = tree->right;
    tree->right = newtree->left;
    newtree->left = tree;
    UpdateHeight(tree);
    UpdateHeight(newtree);
```

```
    CombinedOutput("\n - Part of tree after left rotating:\n");
    Display(newtree, 0);
    CombinedOutput("\n");
    return newtree;
}

template<typename T>
Node<T> *AVLTree<T>::RotateRight(Node<T> *tree){
    CombinedOutput(" - Rotating right next part of tree:\n");
    Display(tree, 0);
    Node<T> *newtree = tree->left;
    tree->left = newtree->right;
    newtree->right = tree;
    UpdateHeight(tree);
    UpdateHeight(newtree);
    CombinedOutput("\n - Part of tree after right rotating:\n");
    Display(newtree, 0);
    CombinedOutput("\n");
    return newtree;
}

template<typename T>
Node<T> *AVLTree<T>::Balance(Node<T> *tree){
    UpdateHeight(tree);
    int balance_factor = BalanceFactor(tree);
    cout << " - Balance factor is " << balance_factor << "\n";
    if (file_output_){
        output << " - Balance factor is " << balance_factor << "\n";
    }
     if (balance_factor <= -2){
          if (BalanceFactor(tree->left) > 0)
               tree->left = RotateLeft(tree->left);
          return RotateRight(tree);
     }
     if (balance_factor => 2){
          if (BalanceFactor(tree->right) < 0)
               tree->right = RotateRight(tree->right);
          return RotateLeft(tree);
     }
     return tree;
}

template<typename T>
void AVLTree<T>::Display(Node<T> *tree, int depth){
    for (int i = 0; i < depth; i++){
        CombinedOutput(". ");
    }
    if (tree == NULL){
        CombinedOutput("(empty)\n");
    }
```

```
    if (tree != NULL){
        cout << tree->key << " (" << counter[tree->key] << ")\n";
        if (file_output_){
            output << tree->key << " (" << counter[tree->key] << ")\n";
        }
        Display(tree->left, depth + 1);
        Display(tree->right, depth + 1);
    }
}

template<typename T>
void AVLTree<T>::Find(Node<T> *tree, T elem){
    if (counter[elem]){
        cout << " - This element is contained " << counter[elem] << "
time(s)\n";
        if (file_output_){
            output << " - This element is contained " << counter[elem] <<
" time(s)\n";
        }
    } else {
        cout << " - There is no element " << elem << "\n";
        if (file_output_){
            output << " - There is no element " << elem << "\n";
        }
    }
    cout << "\n         ..::Inserting " << elem << "::..        \n";
    if (file_output_){
        output << "\n         ..::Inserting " << elem << "::..        \n";
    }
    root = MakeNode(root, elem);
}

template<typename T>
Node<T> *AVLTree<T>::FindMinimal(Node<T> *tree){
    while (tree->left != NULL) {
        tree = tree->left;
    }
    return tree;
}

template<typename T>
Node<T> *AVLTree<T>::RemoveMinimal(Node<T> *tree){
    if(tree->left == NULL){
        return tree->right;
    }
    tree->left = RemoveMinimal(tree->left);
    return Balance(tree);
}

template<typename T>
```

```
Node<T>* AVLTree<T>::Remove(Node<T> *tree, T elem){
    if(tree == NULL){
        CombinedOutput("Current AVL-Tree are not containing this key\n");
        return NULL;
    }
    cout << "Comparing " << elem << " with current node " << tree->key <<
"\n";
    if (file_output_){
        output << "Comparing " << elem << " with current node " << tree-
>key << "\n";
    }
     if(elem < tree->key){
        CombinedOutput("Going to look in left tree\n");
           tree->left = Remove(tree->left, elem);
    } else if (elem > tree->key){
        CombinedOutput("Going to look in right tree\n");
           tree->right = Remove(tree->right, elem);
    } else {
        if (!(--counter[tree->key])){
            Node<T>* left = tree->left;
           Node<T>* right = tree->right;
           delete tree;
            CombinedOutput("Key was removed from AVL-Tree\n");
           if(right == NULL){
                return left;
            }
            CombinedOutput(" - Finding minimal key in right tree\n");
           Node<T>* min = FindMinimal(right);
            cout << " - Minimal key, finded in right tree - " << min->key
<< "\n";
            if (file_output_){
                output << " - Minimal key, finded in right tree - " <<
min->key << "\n";
            }
            CombinedOutput(" - Removing minimal key in right tree\n");
           min->right = RemoveMinimal(right);
            CombinedOutput(" - Placing minimal key in current tree\n");
           min->left = left;
            cout << "Checking the AVL-Tree with node " << min->key << " to
balancing\n";
            if (file_output_){
                output << "Checking the AVL-Tree with node " << min->key
<< " to balancing\n";
            }
           return Balance(min);
        } else {
            CombinedOutput("One copy of this key was removed from AVL-
Tree\n");
        }
     }
```

```
    cout << "Checking the AVL-Tree with node " << tree->key << " to
balancing\n";
    if (file_output_){
        output << "Checking the AVL-Tree with node " << tree->key << " to
balancing\n";
    }
    return Balance(tree);
}

template<typename T>
void AVLTree<T>::CombinedOutput(const char *message){
    cout << message;
    if (file_output_){
        output << message;
    }
}

int main(){
    system("clear");
    AVLTree<TYPE> tree;
    TYPE elem = 0;
    cout << "Do you need output to a file?\n";
    cout << "Enter \'y\' if yes\n";
    cout << "Your choice - ";
    string choice, working;
    cin >> choice;
    cin.ignore();
    switch(choice[0]){
        case 'y':
            tree.file_output_ = true;
            tree.output.open("output.txt");
            break;
        default:
            tree.file_output_ = false;
            break;
    }
    system("clear");
    if (tree.output.is_open() || (!tree.file_output_)){
        tree.ReadTree();
        do{
            tree.CombinedOutput("\nWhat's next?\n");
            tree.CombinedOutput("Find and insert element - 1, remove
element - 2\n");
            tree.CombinedOutput("Your choice - ");
            choice = "";
            cin >> choice;
            cin.ignore();
            if (tree.file_output_){
                tree.output << choice[0] << "\n";
            }
```

```
            tree.CombinedOutput("Enter the element you want to operate -
");
            cin >> elem;
            if (tree.file_output_){
                tree.output << elem << "\n";
            }
            switch(choice[0]){
                case '1':
                    tree.CombinedOutput("\n             ..::Finding   and
inserting::..    \n");
                    tree.Find(tree.root, elem);
                    tree.CombinedOutput("   ::::::::::::::::::::::::::::
\n");
                    break;
                case '2':
                    tree.CombinedOutput("\n      ..::Removing element::..
\n");
                    tree.root = tree.Remove(tree.root, elem);
                    tree.CombinedOutput("   ::::::::::::::::::::::::::::
\n");
                    break;
                default:
                    tree.CombinedOutput("Wrong operation\n");
                    break;
            }
            cout << "\nDo you want to see current tree?\n";
            cout << "Enter \'y\' if yes\n";
            cout << "Your choice - ";
            choice = "";
            cin >> choice;
            cin.ignore();
            if (choice[0] == 'y'){
                tree.CombinedOutput("\n          ..::Current AVL-Tree::..
\n");
                tree.Display(tree.root, 0);
                tree.CombinedOutput("       ::::::::::::::::::::::::::::
\n");
            }
            cout << "\nDo you want to continue?\n";
            cout << "Enter \'y\' if yes\n";
            cout << "Your choice - ";
            working = "";
            cin >> working;
            cin.ignore();
        } while(working[0] == 'y');
        tree.CombinedOutput("\n         ..::Final Result::..        \n");
        tree.Display(tree.root, 0);
        tree.CombinedOutput("    ::::::::::::::::::::::::::    \n");
    } else {
```

```
        cout << "Cannot open file \"output.txt\". The program is shutting
down.\n";
    }
    return 0;
}
```