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 и алгоритмы сжатия»

Студент гр. 7381 Минуллин М.А.

Преподаватель Фирсов М.А.

Санкт-Петербург

2018

Задание.

17. БДП: АВЛ-дерево; действие: 1+2б. **5-я лабораторная работа** включает в себя задания на кодирование, декодирование или создание определённого БДП с последующим выполнением определённых действий с ним.

В вариантах заданий 1-ой группы (кодирование и декодирование) на вход подаётся файл с закодированным или незакодированным содержимым. Требуется раскодировать или закодировать содержимое файла определённым алгоритмом.

В вариантах заданий 2-ой группы (БДП) требуется:

1) По заданному файлу F (типа file of *Elem*), все элементы которого различны, построить БДП определённого типа;

2) Выполнить одно из следующих действий:

а) Для построенного БДП проверить, входит ли в него элемент *e* типа *Elem*, и если не входит, то добавить элемент *e* в дерево поиска.

б) Для построенного БДП проверить, входит ли в него элемент *e* типа *Elem*, и если входит, то удалить элемент *e* из дерева поиска.

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

г) Другое действие.

Пояснение задания:

На вход программе подаётся файл, содержащий как минимум один элемент заданного типа данных – элемент, с которым необходимо выполнить второе задание. Следом идёт произвольное количество значений заданного типа, с помощью которых строится дерево.

Описание алгоритма.

Первый элемент файла сохраняется в отдельную переменную. Все остальные читаются до конца файла и добавляются в дерево. Алгоритм добавления при это игнорирует элементы, уже добавленные в дерево, т.е. если в дерево добавить n раз один и тот же элемент, то в дереве он будет записан всего 1 раз.

Как только файл закончился, производится вывод в стандартный поток вывода информации о том, найден ли первый элемент из файла в дереве (и незамедлительно удаляется), о построенном дереве, позволяющей убедиться в правильности построения: высота дерева, количество элементов в дереве, минимальный и максимальный элементы дерева, элементы дерева по возрастанию (ЛКП обход) и наглядная структура дерева (КЛП обход). Чтобы не загромождать вывод, вывод элементов дерева производится, если их количество невелико.

Задание лабораторной работы удобно расписать на 3 алгоритма: алгоритм поиск и алгоритм удаления, алгоритм балансировки.

Алгоритм поиска:

1. Если корень равен NULL, возвращаем NULL.
2. Если значение в данном узле совпадает с искомым, возвращаем указатель на текущий узел.
3. Если значение в данном узле строго больше искомого:
    a. Если указатель на левое поддерево равен NULL, возвращаем NULL,
    b. Иначе переходим в левое поддерево, переходим к пункту 2.
4. Если значение в данном узле строго меньше искомого:
    a. Если указатель на правое поддерево равен NULL, возвращаем NULL,
    b. Иначе переходим в правое поддерево, переходим к пункту 2.

Алгоритм удаления:

1. Если алгоритм поиска вернул NULL, ничего не делаем.
2. Сохраняем указатели на левого и правого сыновей в переменные l и r соответственно.
3. Переходим в правое поддерево, затем спускаемся по левым поддеревьям, пока не дойдём до листа – это лист с минимальным элемент правого поддерева удаляемого элемента, записываем его в переменную m.
4. Поднимаем лист m на место удаляемого поддерева.
5. Балансируем поддерево m.

Алгоритм балансировки:

1. Если фактор балансировки равен 2, то
    a. Если фактор балансировки правого поддерева отрицательный, то делаем для него правый поворот.
    b. Делаем левый поворот.
2. Если фактор балансировки равен -2, то
    a. Если фактор балансировки левого поддерева положительный, то делаем для него левый поворот.
    b. Делаем правый поворот.

Описание функций и структур данных.

Для выполнения лабораторной работы были написаны два класса. Основной – AVLTree и вспомогательный, используемый первым – AVLNode.

Рассмотрим класс AVLNode:

– конструктор, выставляющий все поля созданного объекта по стандарту.

Принимаемые аргументы:

`const Type& value` – значение, хранимое в новой записи.
`size_t height = 1` – высота дерева.
`AVLNode *left = nullptr` – указатель на левое поддерево.

`AVLNode *right = nullptr` – указатель ан право поддерево.

Возвращаемое значение: конструктор ничего не возвращает.

`size_t size() const` – метод, возвращающий количество элементов в дереве.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: целое неотрицательное число – количество элементов.

`AVLNode *minimum()` – метод, возвращающий указатель на узел дерева с минимальным ключом. Поиск осуществляется спуском по левым сыновьям дерева до листа.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: указатель на узел с минимальным элементом.

`AVLNode *maximum()` – метод, возвращающий указатель на узел дерева с максимальным ключом. Поиск осуществляется спуском по правым сыновьям дерева до листа.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: указатель на узел с максимальным ключом.

`int balance_factor() const` – определение баланса текущего узла.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: целое знаковое число – значение баланса из промежутка [-2;2]

`void height_update()` – обновление высот после балансировки текущего узла.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: метод ничего не возвращает.

`AVLNode *rotate_left()` – малый левый поворот текущего узла.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: метод возвращает узел, вставший на место поворачиваемого.

`AVLNode *rotate_right()` – малый правый поворот текущего узла.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: метод возвращает указатель на узел, вставший на место поворачиваемого.

`AVLNode *balance()` – метод, балансирующий текущий узел.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: метод возвращает указатель на узел, вставший на место балансируемого.

`AVLNode *remove_minimum()` – метод, удаляющий узел с минимальным ключом из данного поддерева.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: метод возвращает указатель на текущий узел.

`AVLNode *insert(const Type& value)` – метод, вставляющий заданное значение в дерево.

Принимаемые аргументы:

`const Type& value` – вставляемое значение.

Возвращаемое значение: метод возвращает указатель на новый узел.

`AVLNode *search(const Type& value)` – метод, осуществляющий поиск заданного значения в дереве.

Принимаемые аргументы:

`const Type& value` – искомое значение.

Возвращаемое значение: указатель на узел с элементом, равным искомому, либо NULL, если узел на найден.

`AVLNode *remove(const Type& value)` – метод, удаляющий узел с заданным значением.

Принимаемые аргументы:

`const Type& value` – удаляемое значение.

Возвращаемое значение: указатель на узел вставший на место удаляемого элемента.

`void display_tree(size_t depth) const` – метод, выводящий дерево в древовидной форме в стандартный поток вывода.

Принимаемые аргументы:

`size_t depth` – глубина текущего вызова функции.

Возвращаемое значение: метод ничего не возвращает.

`void display_list() const` – метод, выводящий элементы, хранимые в дереве в порядке возрастания в стандартный поток вывода.

Принимаемые аргументы: метод ничего не принимает.

Возвращаемое значение: метод ничего не возвращает.

Рассмотрим отдельные функции:

`void print_tabs(size_t tabs_count)` – метод, выводящий в стандартный поток вывода заданное количество символов табуляции, необходимый для вывода дерева в удобочитаемом виде в консоль.

Принимаемые аргументы:

`size_t tabs_count` – количество символов табуляции.

Возвращаемое значение: функция ничего не возвращает.

Тестирование.

Для тестирования был использован bash-скрипт, использованный в первых 4 лабораторных работах с небольшими доработками. Данные тестирования представлены в таблице ниже. Ввиду большого объёма выводимой информации большая часть тестов искусственно урезана.

№	Входные данные	Выходные данные
1	42	default test: "empty.txt" AVL-Tree input: element: 42 values: empty (0) AVL-Tree task: element NOT found AVL-Tree information: Height: 0 Size: 0 Minimum: AVLTree::minimum() error: AVL-Tree is empty! Maximum: AVLTree::maximum() error: AVL-Tree is empty! Values: AVL-Tree is empty Scheme: AVL-Tree is empty
2	42 17 23 41 42	default test: "found.txt" AVL-Tree input: element: 42 values: 17, 23, 41, 42 (4) AVL-Tree task: element found

		AVL-Tree information: Height: 2 Size: 3 Minimum: 17 Maximum: 41 Values: 17 23 41 Scheme: 0 1 root: 23 left: 17 right: 41 0 1
3	0 -20 -18 -16 -14 -12 -10 -8 -6 -4 -2 0 2 4 6 8 10 12 14 16 18 20	default test: "many_values.txt" AVL-Tree task: element found AVL-Tree information: Height: 5 Size: 20 Minimum: -20 Maximum: 20
4	42 -7 0 7 13 43	AVL-Tree task: element NOT found
5	1 1 1 1 1 1 1 1	AVL-Tree task: element found
6	13 1 1 1 1 1 1 1 1 1 1 1 1 1	AVL-Tree task: element NOT found
7	7 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	AVL-Tree task: element found

Выводы.

В ходе выполнения лабораторной работы была изучена такая абстрактная структура данных как самобалансирующееся бинарное дерево поиска АВЛ-дерево. Был написан шаблонный класс АВЛ-дерева для работы с произвольным типом данных. В качестве языка разработки был использован C++, компилятор clang++, bash-скрипты.

ПРИЛОЖЕНИЯ

Приложение А. Код основной программы.

```
#include "AVLTree.hpp"

#include <iostream>
#include <typeinfo>
#include <algorithm>

typedef int Type;

int main() {
    const size_t max_count = 16;
    Type array[max_count];
    AVLTree<Type> avl_tree;

    Type element;
    std::cin >> element;

    Type value;
    size_t count = 0;
    while (!std::cin.eof()) {
        std::cin >> value;
        if (count < max_count)
            array[count] = value;
        ++count;

        avl_tree.insert(value);
    }

    std::cout << "AVL-Tree input:" << std::endl;
    std::cout << "\telement:\t" << element << std::endl;

    std::cout << "\tvalues:\t\t";
    if (count == 0)
        std::cout << "empty";
    if (count > 0)
        std::cout << array[0];
    for (size_t i = 1; i < std::min(count, max_count); ++i)
        std::cout << ", " << array[i];
    if (count > max_count)
        std::cout << "...";
    std::cout << " (" << count << ")" << std::endl;

    std::cout << "AVL-Tree task: " << std::endl;
    std::cout << "\telement\t\t" << (avl_tree.find(element) ? "" : "NOT ") <<
"found" << std::endl;
    avl_tree.remove(element);
    std::cout << "AVL-Tree information: " << std::endl;
```

```
    std::cout << "\tHeight:\t\t" << avl_tree.height() << std::endl;
    std::cout << "\tSize:\t\t" << avl_tree.size() << std::endl;

    try {
        std::cout << "\tMinimum:\t" << avl_tree.minimum() << std::endl;
    } catch (std::domain_error &e) {
        std::cout << e.what() << std::endl;
    }

    try {
        std::cout << "\tMaximum:\t" << avl_tree.maximum() << std::endl;
    } catch (std::domain_error &e) {
        std::cout << e.what() << std::endl;
    }

    if (avl_tree.size() <= max_count) {
        std::cout << "\tValues:\t\t";
        avl_tree.display_list();
        std::cout << "\tScheme:\t\t";
        avl_tree.display_tree();
    }

    std::cout << std::endl;

    return 0;
}
```

Приложение Б. Код заголовочного файла АВЛ-дерева.

```
#ifndef __AVLTREE_HPP__
#define __AVLTREE_HPP__

#include "AVLNode.hpp"

#include <algorithm>
#include <stdexcept>
#include <iostream>
#include <cstddef>

template <class Type>
class AVLTree {
public:
    AVLTree();
    ~AVLTree();

    Type minimum() const;
    Type maximum() const;

    size_t height() const;
    size_t size() const;
    bool empty() const;
```

```
    void insert(const Type& value);
    void remove(const Type& value);
    bool find(const Type& value);

    void display_tree() const;
    void display_list() const;
private:
    AVLNode<Type> *root;
};

template <class Type>
AVLTree<Type>::AVLTree() {
    root = nullptr;
}

template <class Type>
AVLTree<Type>::~AVLTree() {
    if (root)
        delete root;
}

template <class Type>
Type AVLTree<Type>::minimum() const {
    if (!root)
        throw std::domain_error("AVLTree::minimum() error: AVL-Tree is empty!");
    return root->minimum()->value;
}

template <class Type>
Type AVLTree<Type>::maximum() const {
    if (!root)
        throw std::domain_error("AVLTree::maximum() error: AVL-Tree is empty!");
    return root->maximum()->value;
}

template <class Type>
size_t AVLTree<Type>::height() const {
    return (root ? root->height : 0);
}

template <class Type>
size_t AVLTree<Type>::size() const {
    if (root)
        return root->size();
    else
        return 0;
}

template <class Type>
bool AVLTree<Type>::empty() const {
```

```
    return (!root);
}

template <class Type>
void AVLTree<Type>::insert(const Type& value) {
    if (root) {
        if (!root->search(value))
            root = root->insert(value);
    }
    else
        root = new AVLNode<Type>(value);
}

template <class Type>
void AVLTree<Type>::remove(const Type& value) {
    if (root)
        root = root->remove(value);
}

template <class Type>
bool AVLTree<Type>::find(const Type& value) {
    if (root)
        return (bool)root->search(value);
    return false;
}

template <class Type>
void AVLTree<Type>::display_tree() const {
    if (!root) {
        std::cout << "AVL-Tree is empty" << std::endl;
        return;
    }

    std::cout << std::endl;
    size_t h = height();
    for (size_t i = 0; i < h; ++i)
        std::cout << '\t' << i;
    std::cout << std::endl;

    std::cout << "\troot:\t";
    root->display_tree(1);

    for (size_t i = 0; i < h; ++i)
        std::cout << '\t' << i;
    std::cout << std::endl;
}

template <class Type>
void AVLTree<Type>::display_list() const {
    if (root)
        root->display_list();
```

```
    else
        std::cout << "AVL-Tree is empty";
    std::cout << std::endl;
}


#endif
```

Приложение В. Текст файла исходного кода АВЛ-дерева.

```
#include "AVLTree.hpp"
```

Приложение Г. Текст заголовочного файла записи АВЛ-дерева.

```
#ifndef __AVLNODE_HPP__
#define __AVLNODE_HPP__

#include "Additional.hpp"

#include <algorithm>
#include <iostream>
#include <cstddef>

template <class Type>
struct AVLNode {
    Type value;
    size_t height;
    AVLNode *left;
    AVLNode *right;
    AVLNode(const Type& value, size_t height = 1, AVLNode *left = nullptr,
AVLNode *right = nullptr) :
        value(value), height(height), left(left), right(right) { }

    size_t size() const;

    AVLNode *minimum();
    AVLNode *maximum();

    int balance_factor() const;
    void height_update();

    AVLNode *rotate_left();
    AVLNode *rotate_right();

    AVLNode *balance();

    AVLNode *remove_minimum();
    AVLNode *insert(const Type& value);
    AVLNode *search(const Type& value);
    AVLNode *remove(const Type& value);

    void display_tree(size_t depth) const;
```

```
    void display_list() const;
};

template <class Type>
size_t AVLNode<Type>::size() const {
    return 1 + (left ? left->size() : 0) + (right ? right->size() : 0);
}

template <class Type>
AVLNode<Type> *AVLNode<Type>::minimum() {
    return (left ? left->minimum() : this);
}
template <class Type>
AVLNode<Type> *AVLNode<Type>::maximum() {
    return (right ? right->maximum() : this);
}

template <class Type>
int AVLNode<Type>::balance_factor() const {
    return (right ? right->height : 0) - (left ? left->height : 0);
}

template <class Type>
void AVLNode<Type>::height_update() {
    height = std::max(right ? right->height : 0, left ? left->height : 0) + 1;
}

template <class Type>
AVLNode<Type> *AVLNode<Type>::rotate_left() {
    AVLNode<Type> *node = right;
    right = node->left;
    node->left = this;
    height_update();
    node->height_update();
    return node;
}

template <class Type>
AVLNode<Type> *AVLNode<Type>::rotate_right() {
    AVLNode<Type> *node = left;
    left = node->right;
    node->right = this;
    height_update();
    node->height_update();
    return node;
}

template <class Type>
AVLNode<Type> *AVLNode<Type>::balance() {
    height_update();
    if (balance_factor() == 2) {
```

```
        if (right->balance_factor() < 0)
            right = right->rotate_right();
        return rotate_left();
    }
    if (balance_factor() == -2) {
        if (left->balance_factor() > 0)
            left = left->rotate_left();
        return rotate_right();
    }
    return this;
}

template <class Type>
AVLNode<Type> *AVLNode<Type>::remove_minimum() {
    if (!left)
        return right;
    left = left->remove_minimum();
    return balance();
}

template <class Type>
AVLNode<Type> *AVLNode<Type>::insert(const Type& value) {
    if (value < this->value) {
        if (left)
            left = left->insert(value);
        else
            left = new AVLNode<Type>(value);
    }
    else {
        if (right)
            right = right->insert(value);
        else
            right = new AVLNode<Type>(value);
    }
    return balance();
}

template <class Type>
AVLNode<Type> *AVLNode<Type>::search(const Type& value) {
    if (value < this->value) {
        if (left)
            return left->search(value);
    }
    else if (value > this->value) {
        if (right)
            return right->search(value);
    }
    else
        return this;
    return nullptr;
}
```

```
template <class Type>
AVLNode<Type> *AVLNode<Type>::remove(const Type& value) {
    if (value < this->value) {
        if (left)
            left = left->remove(value);
    }
    else if (value > this->value) {
        if (right)
            right = right->remove(value);
    }
    else {
        AVLNode<Type>* l = left;
        AVLNode<Type>* r = right;
        delete this;
        if (!r)
            return l;
        AVLNode<Type>* m = r->minimum();
        m->right = r->remove_minimum();
        m->left = l;
        return m->balance();
    }
    return balance();
}

template <class Type>
void AVLNode<Type>::display_tree(size_t depth) const {
    std::cout << value << std::endl;
    if (left) {
        print_tabs(depth);
        std::cout << "\tleft:\t";
        left->display_tree(depth + 1);
    }
    if (right) {
        print_tabs(depth);
        std::cout << "\tright:\t";
        right->display_tree(depth + 1);
    }
}

template <class Type>
void AVLNode<Type>::display_list() const {
    if (left)
        left->display_list();
    std::cout << value << " ";
    if (right)
        right->display_list();
}
#endif
```

Приложение Д. Текст файла исходного кода записи АВЛ-дерева.

```
#include "AVLNode.hpp"
```

Приложение Е. Текст заголовочного файла дополнительных функций.

```
#include "Additional.hpp"

void print_tabs(size_t tabs_count) {
    for (size_t i = 0; i < tabs_count; ++i)
        std::cout << '\t';
}
```

Приложение Ж. Текст файла исходного кода дополнительных функций.

```
#ifndef __ADDITIONAL_HPP__
#define __ADDITIONAL_HPP__

#include <cstddef>
#include <iostream>

void print_tabs(size_t tabs_count);

#endif
```

Приложение З. Текст мэйк-файла.

```
CODE    = ./source/
OBJ     = main.o Additional.o AVLNode.o AVLTree.o
EXE     = avl_tree
CXX     = clang++
CFLAGS  = -std=c++11 -Wall -Wextra -c

all: $(OBJ)
    $(CXX) $(OBJ) -o $(EXE)

main.o: $(CODE)main.cpp $(CODE)AVLTree.hpp
    $(CXX) $(CFLAGS) $(CODE)main.cpp

Additional.o: $(CODE)Additional.cpp
    $(CXX) $(CFLAGS) $(CODE)Additional.cpp

AVLTree.o:  $(CODE)AVLTree.cpp $(CODE)AVLNode.hpp
    $(CXX) $(CFLAGS) $(CODE)AVLTree.cpp

AVLNode.o: $(CODE)AVLNode.cpp $(CODE)Additional.hpp
    $(CXX) $(CFLAGS) $(CODE)AVLNode.cpp

clean:
    rm $(OBJ)

cleanest:
    rm $(OBJ) $(EXE)
```

Приложение И. Текст bash-скрипта для тестирования программы.

```
#!/bin/bash
make
if [ -f "result.txt" ]; then
    rm result.txt
fi
touch result.txt
for i in $(ls tests/default); do
    echo "running test: \"tests/default/$i\" ";
    sleep 0.05s;
    echo "default test: \"$i\"" >>result.txt;
    ./avl_tree <tests/default/$i >>result.txt;
done;
for i in $(ls tests/random); do
    echo "running test: \"tests/random$i\" ";
    sleep 0.05s;
    echo "random test: \"$i\"" >>result.txt;
    ./avl_tree <tests/random/$i >>result.txt;
done;
make clean
```