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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «АиСД»

ТЕМА: БДП: АВЛ-дерево

Студентка гр. 9382 _____ Балаева М.О.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Освоить операции вставки и удаления в АВЛ-деревьях.

Основные теоретические положения.

Задание

Вариант №16.

АВЛ-дерево— сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

АВЛ — аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса.

БДП: АВЛ-дерево; действие: 1+2б:

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

2.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.

Ход работы.

class AVL_tree -Класс представления АВЛ-дерева. Является классом-оберткой над Node , в то время когда в Node определены функции - вращения, балансировки. В классе AVL_tree определены функции вставки и исключения.

int diff_height() – Функция поиска разности между высотами поддеревьем элемента. Возвращает между высотами левого и правого поддерева

void update_height() - После каждой вставки/балансировки/удаления нужно обновлять высоту дерева

Node *rotate_right() – Правое вращение. Вовращает Node* p - новый корень полученного дерева

Node *rotate_left - Функция левого вращения. Возвращает новый корень дерева

Node *balance() - Функция балансировки АВЛ-дерева. Балансировка нужна в случае когда разница высот левого и правого поддеревьев становится равной |2|.Возвращает указатель на самого себя(узел).

AVL_tree - Конструктор АВЛ-дерева принимает корень.

void print_tree (Node *node, int level) - Служебная функция вывода дерева. Выводит дерево не сверху-вниз, а слева-направо. Принимает корень выводимого поддерева. Принимает уровень рекурсии для индентации.

Node *insert_node (Node *node, int value) - Вставка элемента. В конце необходимо балансировать. Принимает корень дерева, куда добавляем. Принимает ключ элемента. Возвращает корень сбалансированного дерева.

Node *remove_node (Node *node, int value) - Функция удаления элемента с заданным ключом находим узел p с заданным ключом value , в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min.

Принимает корень дерева, в котором происходит удаление элемента. Принимает value ключ для удаления.Возвращает ребалансированный корень дерева.

Node *find_min (Node *node) - Функция поиска минимального элемента в дереве или поддереве. Возвращает корень дерева, где ищется минимум. Возвращает указатель на элемент с наименьшим ключем.

Node *remove_min (Node *node)- Удаление минимального элемента из заданного дерева. По свойству АВЛ-дерева у минимального элемента справа либо подвешен узел, либо там пусто. В обоих случаях надо просто вернуть указатель на правый узел и при возвращении из рекурсии выполнить балансировку.
Принимает корень дерева или поддерева, где удаляется минимальный элемент.
Возвращает указатель на новый корень после балансировки.

Node *lets_insert_node (Node *root, int value) - Служебная функция-обертка над вставком для удобного вывода. Возвращает корень дерева или поддерева, куда вставляется элемент. Принимает ключ элемента для вставки. Возвращает корень поддерева.

Node *lets_remove_node (Node *root, int value)- Служебная функция-обертка над remove. Принимает поддерево или дервео, в котором удаляется элемент ,элемент для удаления. Возвращает корень дерева, где удаляли элемент.

Тестирование.

№	Входные данные	Выходные данные
1.	1 35	-!-!-!-!-!-!-!-!-!-!-!-!- 35 -!-!-!-!-!-!-!-!-!-!-!-!-
2.	1 11	-!-!-!-!-!-!-!-!-!-!-!-!- 35 11 -!-!-!-!-!-!-!-!-!-!-!-!-

3.	1
	10
		-!-!-!-!-!-!-!-!-!-!-!-!-
		  35
		11
		  10
		-!-!-!-!-!-!-!-!-!-!-!-!-

4.	1
	40
		-!-!-!-!-!-!-!-!-!-!-!-!-
		  40
		  35
		11
		  10
		-!-!-!-!-!-!-!-!-!-!-!-!-

5.	1 30	-!-!-!-!-!-!-!-!-!-!-!-!- 40 35 30 11 10 -!-!-!-!-!-!-!-!-!-!-!-!-
6.	1 6	-!-!-!-!-!-!-!-!-!-!-!-!- 40 35 30 11 10 6 -!-!-!-!-!-!-!-!-!-!-!-!-
7.	1 4	-!-!-!-!-!-!-!-!-!-!-!-!- 40 35 30 11 10 6 4 -!-!-!-!-!-!-!-!-!-!-!-!-
8.	2 10	-!-!-!-!-!-!-!-!-!-!-!-!- 40 35 30 11 10 6 -!-!-!-!-!-!-!-!-!-!-!-!-
9.	1 3	-!-!-!-!-!-!-!-!-!-!-!-!- 40 35 30 11 10 6 3 -!-!-!-!-!-!-!-!-!-!-!-!
10.	yh	Wrong input!

Выводы.

В ходе работы была освоена реализация работы операций вставки и исключения для АВЛ деревьев.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <bits/stdc++.h>

using namespace std;
static int op_count = 0;
static int rot_count = 0;
class Node {
protected:
    int value;
    int height;
    Node *lt;
    Node *rt;
    Node *rotate_right();
    Node *rotate_left();

public:
    Node(int k) : value(k), lt(nullptr), rt(nullptr), height(1) {}
    int get_height();
    int diff_height();
    void update_height();
    Node *balance();
    Node *get_right();
    Node *get_left();
    void set_left(Node *node);
    void set_right(Node *node);
    void set_value(int value);
    int get_value();
};

int Node::get_height() {
    return this ? this->height : 0;
}

int Node::diff_height() {
    return this->rt->get_height() - this->lt->get_height();
}

void Node::update_height() {
    int hl = this->lt->get_height();
    int hr = this->rt->get_height();
    this->height = max(hl, hr) + 1;
}

Node *Node::rotate_right() {
    Node *new_root = this->lt;
    this->lt = new_root->rt;
```

```
    new_root->rt = this;
    this->update_height();
    new_root->update_height();
    return new_root;
}

Node *Node::rotate_left() {
    Node *new_root = this->rt;
    this->rt = new_root->lt;
    new_root->lt = this;
    this->update_height();
    new_root->update_height();
    return new_root;
}

Node *Node::balance() {
    rot_count++;
    this->update_height();
    int diff = this->diff_height();
    if (diff == 2) {
        if (this->rt->diff_height() < 0) this->rt = this->rt-
>rotate_right();
        return this->rotate_left();
    } else if (diff == -2) {
        if (this->lt->diff_height() > 0) this->lt = this->lt->rotate_left();
        return this->rotate_right();
    }
    return this;
}

Node *Node::get_right() {
    return this ? this->rt : nullptr;
}

Node *Node::get_left() {
    return this ? this->lt : nullptr;
}

int Node::get_value() {
    return this ? this->value : 0;
}

void Node::set_left(Node *node) {
    if (this)
        this->lt = node;
}

void Node::set_right(Node *node) {
    if (this)
        this->rt = node;
}
```

```
void Node::set_value(int value) {
    if (this)
        this->value = value;
}

class AVL_tree {
public:
    Node *root;
    AVL_tree() : root(nullptr) {};
    AVL_tree(int k);
    void print_tree(Node *node, int level);
    Node *insert_node(Node *node, int value);
    Node *find_min(Node *node);
    Node *remove_min(Node *node);
    Node *remove_node(Node *node, int value);
    Node *lets_insert_node(Node *root, int value);
    Node *lets_remove_node(Node *root, int value);
};

AVL_tree::AVL_tree(int k) {
    cout << "[Created avl tree| root:" << k << "]\n\n";
    this->root = new Node(k);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!-" << endl;
    this->print_tree(this->root, 0);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!-" << endl;
}

void AVL_tree::print_tree(Node *node, int level) {
    if (node) {
        print_tree(node->get_right(), level + 1);
        for (int i = 0; i < level; i++) cout << "    ";
        cout << node->get_value() << endl;
        print_tree(node->get_left(), level + 1);
    }
}

Node *AVL_tree::insert_node(Node *node, int value) {
    op_count++;
    if (node == nullptr) return new Node(value);
    if (value < node->get_value()) {
        node->set_left(insert_node(node->get_left(), value));
    } else if (value > node->get_value()) {
        node->set_right(insert_node(node->get_right(), value));
    }
    return node->balance();
}

Node *AVL_tree::remove_node(Node *node, int value) {
    op_count++;
    if (node == nullptr) {
        return nullptr;
    }
```

```
    if (value < node->get_value()) {
        node->set_left(remove_min(node->get_left()));
    } else if (value > node->get_value()) {
        node->set_right(remove_min(node->get_right()));
    } else {
        Node *rt = node->get_right();
        Node *lt = node->get_left();
        delete node;
        if (!rt) return lt;
        Node *min = find_min(rt);
        min->set_right(remove_min(rt));
        min->set_left(lt);
        return min->balance();
    }
    return node->balance();
}

Node *AVL_tree::find_min(Node *node) {
    return node->get_left() ? find_min(node->get_left()) : node;
}

Node *AVL_tree::remove_min(Node *node) {
    op_count++;
    if (node->get_left() == nullptr) {
        return node->get_right();
    }
    node->set_left(remove_min(node->get_left()));
    return node->balance();
}

Node *AVL_tree::lets_insert_node(Node *root, int value) {
    cout << "[Insert element:" << value << "]\n\n";
    root = this->insert_node(root, value);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!-" << endl;
    this->print_tree(root, 0);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!-" << endl;
    return root;
}

Node *AVL_tree::lets_remove_node(Node *root, int value) {
    cout << "[Remove element:" << value << "]\n\n";
    root = this->remove_node(root, value);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!-" << endl;
    this->print_tree(root, 0);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!-" << endl;
    return root;
}

void print_menu() {
    cout << "1.Insert element\n"
            "2.Remove element\n"
            "3.Exit\n\n";
```

```
}

AVL_tree *process_user_input(AVL_tree *tree) {
    int f, user_value;
    print_menu();
    cin >> f;
    switch (f) {
        case 1:
            cout << "Enter element: \n";
            cin >> user_value;
            if (tree) {
                tree->root = tree->lets_insert_node(tree->root, user_value);
            } else {
                tree = new AVL_tree(user_value);
            }
            break;
        case 2:
            if (tree) {
                cout << "Enter element: \n";
                cin >> user_value;
                tree->root = tree->lets_remove_node(tree->root, user_value);
            } else cout << "Tree is empty! \n";
            break;
        case 3:
            exit(0);
    }
    return tree;
}

class Research {
    int input_size;
public:
    unordered_set<int> input;
    Research(int v = 10000) : input_size(v) {};
    void generate_ascendance();

    void generate_random(int lower, int upper);

    void run_add(AVL_tree *tree);

    void run_delete(AVL_tree *tree);

};

void Research::generate_ascendance() {
    for(int i = 1; i <= input_size; i++) {
        input.insert(i);
    }
}

void Research::generate_random(int lower, int upper) {
    auto now = std::chrono::high_resolution_clock::now();
```

```
    std::mt19937 gen;
    gen.seed(now.time_since_epoch().count());
    std::uniform_int_distribution<> distribution(lower, upper);
    while(input.size() < input_size) {
        input.insert(distribution(gen));
    }
}


void Research::run_add(AVL_tree *tree) {
    int tree_size = 0;
    ofstream out;
    out.open("../research_add.csv");
    out << "tree_size," << "op_count," << "rot_count" << endl;
    for(auto x : this->input) {
        op_count = 0;
        rot_count = 0;
        tree_size++;
        tree->root = tree->insert_node(tree->root, x);
        out << tree_size << ',' << op_count << "," << rot_count << "\n";
    }
    out.close();
}

void Research::run_delete(AVL_tree *tree) {
    ofstream out;
    int tree_size = input_size;
    out.open("../research_delete.csv");
    out << "tree_size," << "op_count," << "rot_count" << endl;
    for(auto index : input) {
        op_count = 0;
        rot_count = 0;
        tree->root = tree->remove_node(tree->root, tree->root->get_value());
        out << tree_size << ',' << op_count  << "," << rot_count << "\n";
        tree_size--;
    }
    out.close();

}

int main() {
    AVL_tree *tree = new AVL_tree();
    while (true) {
        tree = process_user_input(tree);
    }

    return 0;
}
```