Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Информационная безопасность»

ОТЧЕТ

по домашнему заданию №2-3

по учебной дисциплине «Алгоритмические языки»

на тему: «Двоичное дерево поиска»

Вариант 23

Выполнил:

Студент 1 курса, гр. ИУ8-24

Спиридонов Олег

2024 г.

Цель работы:

Изучить принципы работы двоичного дерева поиска и самостоятельно создать его упрощённую версию.

Условие задачи:

Необходимо реализовать класс BSTree (двоичное дерево поиска). За основу для реализации элементов дерева написать структуру Node. Структура должна содержать указатели на левый и правый элемент дерева, поле данных типа int, а также опционально указатель на предыдущий элемент.

Класс должен содержать указатель на корень с модификатором доступа private, остальные методы и поля могут быть с модификатором доступа public (при защите необходимо будет обосновать). Класс должен содержать два конструктора:

BSTree(); // конструктор по умолчанию.

BSTree(initializer_list<int> list); // конструктор с параметром.

Класс должен содержать следующие функции:

bool add_element(int value); // функция добавления

bool delete_element(int value); // функция удаления

bool find_element(int value); // функция поиска элемента

void print(); // функция вывода дерева в консоль

bool save_to_file(const std::string& path); // функция сохранения в файл

bool load_from_file(const std::string& path); // функция загрузки из файла

~BSTree(); //деструктор.

Выполнение работы:

Файл **input.txt:**

4 2 1 3 6 5 7

Файл **Binary tree.cpp:**

```
#include <iostream>
```

```cpp
#include <fstream>
#include <string>
using namespace std;

struct Node {
    int data;
    Node* left;   // smaller
    Node* right;  // bigger
    Node* previous = nullptr;
};

class BSTree {
private:
    Node* root = nullptr;
public:
    BSTree();
    BSTree(initializer_list<int> list);
    ~BSTree();
    bool add_element(int value);
    bool delete_element(int value);
    bool find_element(int value);
    void print();
    bool save_to_file(const string& path);
    bool load_from_file(const string& path);
};

BSTree::BSTree() {
}

BSTree::BSTree(initializer_list<int> list) {
    for (auto i: list) {
        add_element(i);
    }
}

BSTree::~BSTree() {
    Node* current = root;
    while (current != nullptr) {
        while (not(current->left == nullptr and current->right == nullptr)) {
            while (current->left != nullptr) {
                current = current->left;
            }
            while (current->right != nullptr) {
                current = current->right;
            }
        }
        Node* to_be_deleted = current;
        current = current->previous;
        if (current != nullptr and current->right == to_be_deleted) {
            current->right = nullptr;
        }
        else if (current != nullptr and current->left == to_be_deleted) {
            current->left = nullptr;
        }
        delete to_be_deleted;
    }
}

bool BSTree::add_element(int data) {
    Node* new_node = new Node{data, nullptr, nullptr, nullptr};
    if (root == nullptr) {
        root = new_node;
        return true;
    }
    else {
```

```
        Node* current = root;
        while (not((new_node->data > current->data and current->right == nullptr) or
(new_node->data < current->data and current->left == nullptr))) {
            if (new_node->data > current->data) {
                current = current->right;
            }
            else if (new_node->data < current->data) {
                current = current->left;
            }
            else {
                return false;
            }
        }
        if (new_node->data > current->data and current->right == nullptr) {
            current->right = new_node;
        }
        else {
            current->left = new_node;
        }
        new_node->previous = current;
        return true;
    }
}

bool BSTree::delete_element(int data) {
    Node* current = root;
    while (not(current == nullptr)) {
        if (data > current->data) {
            current = current->right;
        }
        else if (data < current->data) {
            current = current->left;
        }
        else {
            current->right->previous = current->previous;
            if (current->previous != nullptr) {
                if (current == current->previous->right) {
                    current->previous->right = current->right;
                }
                else {
                    current->previous->left = current->right;
                }
            }
            Node* left_edge = current->right;
            while (left_edge->left != nullptr) {
                left_edge = left_edge->left;
            }
            current->left->previous = left_edge->left;
            left_edge->left = current->left;
            delete current;
            return true;
        }
    }
    return false;
}

bool BSTree::find_element(int data) {
    Node* current = root;
    while (not(current == nullptr)) {
        if (data > current->data) {
            current = current->right;
        }
        else if (data < current->data) {
            current = current->left;
        }
```

```
        else {
            return true;
        }
    }
    return false;
}

string print_recursion(Node* current){
    if (current == nullptr) {
        return " ";
    }
    if (current->left == nullptr and current->right == nullptr) {
        return to_string(current->data);
    }
    return to_string(current->data) + '(' + print_recursion(current->left) + ';' +
print_recursion(current->right) + ')';
}

string print_to_file_recursion(Node* current) {
    if (current == nullptr) {
        return " ";
    }
    if (current->left == nullptr and current->right == nullptr) {
        return to_string(current->data);
    }
    return to_string(current->data) + ' ' + print_to_file_recursion(current->left) +
' ' + print_to_file_recursion(current->right);
}

void BSTree::print() {
    cout << print_recursion(root) << endl;
}

bool BSTree::save_to_file(const string& path) {
    ofstream file_output("output.txt");
    if (file_output.is_open()) {
        file_output << print_to_file_recursion(root) << endl;
        file_output.close();
        return true;
    }
    else {
        cout << "Output file didn't open." << endl;
        return false;
    }
}

bool BSTree::load_from_file(const string& path) {
    ifstream file_input("input.txt");
    if (file_input.is_open()) {
        string data;
        while (file_input >> data) {
            this->add_element(stoi(data));
        }
        file_input.close();
        return true;
    }
    else {
        cout << "Input file didn't open." << endl;
        return false;
    }
}

int main() {
    ofstream file_output("output.txt");
    file_output.close();
```

```
    ifstream file_input("input.txt");
    BSTree drevo;
    drevo.load_from_file("input.txt");
    /*initializer_list<int> list = {4, 2, 6, 1, 3, 5, 7};
    BSTree drevo(list);*/
    drevo.print();
    drevo.save_to_file("output.txt");
}
```

Результат работы программы:

4(2(1;3);6(5;7))

Вывод:

В ходе выполнения домашнего задания были изучены принципы работы двоичного дерева поиска и была реализована его упрощённая версия.