МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МИРЭА – РОССИЙСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»
РТУ МИРЭА

Институт ИКБ

Специальность (направление): 09.03.02 (информационные системы и технологии)

Кафедра: КБ-3 «Разработка программных решений и системного программирования»

Дисциплина: «Алгоритмы и структуры данных»

Практическая работа на тему:
Программа по деревьям

Студент: 20.12.2024 Крашенинников М.В.

подпись *Дата* *инициалы и фамилия*

Группа: БСБО-16-23 Шифр: 23Б0107

Преподаватель: 20.12.2024 Филатов В.В.

подпись *дата* *инициалы и фамилия*

Москва 2024 г.

1. Задание (вариант 7)

7.	Дерево двоичного поиска	Список сыновей	$A=A \cup_{обр} B$	обратный обход

2. Термины

1. Дерево двоичного поиска (Binary Search Tree, BST):

- Это структура данных в виде дерева, где каждый узел содержит ключ, и для каждого узла выполняется следующее условие:
- Все элементы в левом поддереве узла меньше ключа этого узла.
- Все элементы в правом поддереве узла больше ключа этого узла.
- Дерево двоичного поиска позволяет эффективно искать, вставлять и удалять элементы.

2. Список сыновей (Children List):

- Это подход, при котором каждый узел дерева хранит список всех своих дочерних узлов (сыновей).
- Такой подход используется, например, в деревьях с несколькими дочерними элементами или в специализированных структурах, где каждый узел может иметь любое количество детей.
- В контексте двоичного дерева, это может означать, что каждый узел будет содержать список из двух элементов: для левого и правого поддерева.

3. Обратный обход:

- Обратный обход — это способ обхода дерева, при котором элементы посещаются в обратном порядке по сравнению с обычным обходом. Сначала с левого поддерева, плавно переходя к правому и заканчивая корнем.

3. Описание программы

Основные методы:

push(const T& key):

- Этот метод добавляет элемент в дерево. Он ищет подходящее место для нового элемента, сравнивая его с текущими значениями узлов.
- Вставка происходит через рекурсивный спуск влево или вправо в зависимости от значения нового элемента.

myOperation(Tree<T>& A, const Tree<T>& B):

- Осуществляет операцию объединения двух деревьев. Сначала выполняется обход дерева B в обратном порядке (post-order), и элементы добавляются в дерево A.
- Вариант решения использует рекурсивную функцию для обхода дерева B, либо можно использовать итератор для обхода.

PARENT(const Node* node):

- Возвращает родительский узел для переданного узла node. Для этого метод выполняет обход дерева и ищет родителя.

LEFT_CHILD(const Node* node):

- Возвращает указатель на левого ребенка узла, если он существует.

RIGHT_SIBLING(const Node* node):

- Возвращает правого брата для переданного узла, если таковой существует.

LABEL(const Node* node):

- Возвращает значение узла (или пустое значение типа T, если узел не существует).

ROOT():

- Возвращает корень дерева.

printTree(const std::string& treeName):

- Выводит дерево в консоль в виде текстовой схемы, показывая родительско-дочерние связи с помощью символов "/". Для этого используется рекурсивная функция buildTreeLines.

printPostOrderByIterator(const std::string& treeName):

- Выводит дерево в обходе post-order с использованием итератора.

printPostOrderByRecursive(const std::string& treeName):

- Рекурсивно выполняет обход дерева в post-order и выводит значения узлов.

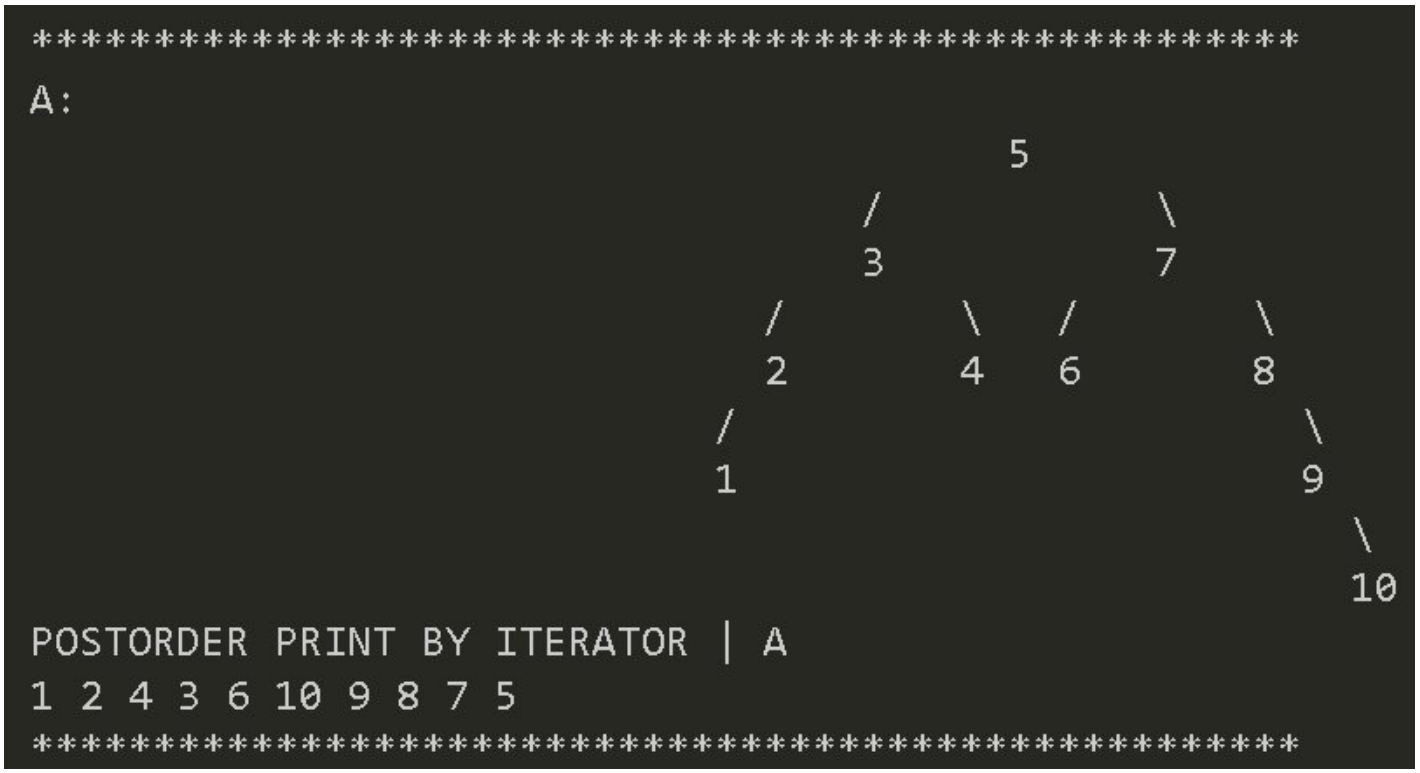
MAKENULL():

- Этот метод удаляет все узлы дерева с помощью итератора (или альтернативно с помощью рекурсивного обхода). Он очищает память, выделенную для каждого узла.

Class Iterator: Итератор используется для обхода дерева в post-order. Он работает через стек, чтобы хранить информацию о текущем узле и его потомках. Итератор позволяет проходить по дереву и получать значения узлов с помощью оператора разыменования operator*.

- **moveToNext()**: Основная логика перехода к следующему узлу в обходе. Он добавляет детей в стек для обхода и поочередно извлекает элементы.
- **operator++**: Переходит к следующему узлу в обходе.
- **operator!=**: Сравнивает два итератора, чтобы проверить, достигнут ли конец обхода.

4. Скриншот работы программы



```
**********************************************************
B:
                                        0
                                          \
                                           3
                                         /   \
                                        2.5   4
                                       /     /  \
                                      2     3.5 4.5
                                    /           \
                                   1.5           5
                                  /
                                 1
POSTORDER PRINT BY RECURSIVE | B
1 1.5 2 2.5 3.5 5 4.5 4 3 0
**********************************************************
```

```
**********************************************************
UNION A and B:
                                          5
                                        /   \
                                       3     7
                                     /   \ /   \
                                    2    4 6    8
                                  /  \  / \     \
                                 1   2.5 3.5 4.5  9
                               /  \                \
                              0   1.5               10
POSTORDER PRINT BY RECURSIVE | UNION A and B
0 1.5 1 2.5 2 3.5 4.5 4 3 6 10 9 8 7 5
**********************************************************
POSTORDER PRINT BY ITERATOR | UNION A and B
0 1.5 1 2.5 2 3.5 4.5 4 3 6 10 9 8 7 5
**********************************************************
```

5. Исходный код

```
/* Программа по деревьям

Группа: БСБО-16-23
```

```
Студент: Крашенинников М. В.

Вариант: 7

----------------------------------------

Название дерева: Дерево двоичного поиска

Реализация дерева: Список сыновей

Обход: обратный

Операция: A = A ∪обр. B

*/


#include <iostream>

#include <vector>

#include <stack>

#include <string>

#include <unordered_set>

#include <functional>

#include <map>

#include <sstream>


const std::string stars(52, '*');


template <typename T>

class Tree {

private:

    struct Node {

        std::vector<Node*> children;

        T value;


        Node(const T& v) : value(v) {}
```

```
    };

    Node *LAMBDA = nullptr, *root;

public:
    Tree() : root(nullptr) {}
    ~Tree() { MAKENULL(); }

    class Iterator {
    public:
        const Node* current;
        std::stack<const Node*> nodes;
        std::unordered_set<const Node*> visited;

        Iterator(const Node* root) : current(nullptr) {
            if (root) nodes.push(root);
            moveToNext();
        }

        T operator*() { return current->value; }

        Iterator& operator++() {
            moveToNext();
            return *this;
        }

        bool operator!=(const Iterator& other) const { return current != other.current; }

    private:
        void moveToNext() {
```

```
            while (!nodes.empty()) {

                const Node* node = nodes.top();


                if (visited.find(node) == visited.end()) {

                    visited.insert(node);


                    if (node->children.size() == 2 && node->children[1])
nodes.push(node->children[1]);

                    if (node->children.size() >= 1 && node->children[0])
nodes.push(node->children[0]);

                } else {

                    current = node;

                    nodes.pop();

                    return;

                }

            }


            current = nullptr;

        }

    };


    Iterator begin() const { return Iterator(root); }

    Iterator end() const { return Iterator(nullptr); }


    Node* PARENT(const Node* node) const {

        if (!root || !node || node == root) return LAMBDA;


        for (Iterator it = begin(); it != end(); ++it) {

            const Node* current = it.current;
```

```
            if (!current) continue;

            for (const Node* child : current->children) {
                if (child == node) {
                    return const_cast<Node*>(current);
                }
            }
        }

        return LAMBDA;
    }

    Node* LEFT_CHILD(const Node* node) const {
        return (!root || !node || node->children.empty()) ? LAMBDA : node->children[0];
    }

    Node* RIGHT_SIBLING(const Node* node) const {
        if (!root || !node) return LAMBDA;

        Node* parent = PARENT(node);
        if (!parent) return LAMBDA;

        if (parent->children.size() == 2 && parent->children[0] == node) {
            return parent->children[1];
        }

        return LAMBDA;
    }
```

```
T LABEL(const Node* node) const { return !node ? T() : node->value; }

static Tree<T> CREATE(Node*& node, Tree<T>& T1, Tree<T>& T2) {
    Tree<T> newTree;
    if (!node) return newTree;

    newTree.root = node;
    if (T1.root) node->children.push_back(T1.root);
    if (T2.root) node->children.push_back(T2.root);

    T1.root = T2.root = nullptr;

    return newTree;
}



Node* ROOT() const { return root; }

void push(const T& key) {
    if (!root) {
        root = new Node(key);
        return;
    }

    Node* current = root;

    while (true) {
        if (key < current->value) {
            if (current->children.empty()) current->children.resize(2, nullptr);
            if (!current->children[0]) {
```

```
                current->children[0] = new Node(key);
                break;
            } else {
                current = current->children[0];
            }
        } else if (key > current->value) {
            if (current->children.size() < 2) current->children.resize(2, nullptr);
            if (!current->children[1]) {
                current->children[1] = new Node(key);
                break;
            } else {
                current = current->children[1];
            }
        } else {
            break;
        }
    }
}

static void myOperation(Tree<T>& A, const Tree<T>& B) {
    std::function<void(const Node*)> postOrder = [&](const Node* current) {
        if (!current) return;

        if (!current->children.empty() ) postOrder(current->children[0]);
        if (current->children.size() == 2 ) postOrder(current->children[1]);

        A.push(current->value);
```

```
    };

    postOrder(B.ROOT());

    // альтернативное решение:
    // for (auto it = B.begin(); it != B.end(); ++it) A.push(*it);
}

void printPostOrderByIterator(const std::string& treeName) const {
    if (!root) {
        std::cout << treeName << " is empty :(\n";
        return;
    }

    std::cout << "POSTORDER PRINT BY ITERATOR | " << treeName << std::endl;

    for (auto it = begin(); it != end(); ++it) {
        std::cout << *it << ' ';
    }

    std::cout << std::endl << stars << std::endl;
}

void printPostOrderByRecursive(const std::string& treeName) const {
    if (!root) {
        std::cout << treeName << " is empty :(\n";
        return;
    }
```

```
        std::cout << "POSTORDER PRINT BY RECURSIVE | " << treeName << std::endl;

        std::function<void(const Node*)> printRec = [&](const Node* current) {
            if (!current) return;

            if (!current->children.empty() ) printRec(current->children[0]);
            if (current->children.size() == 2 ) printRec(current->children[1]);

            std::cout << current->value << ' ';
        };

        printRec(root);

        std::cout << std::endl << stars << std::endl;
    }

    void printTree(const std::string& treeName) const {
        std::cout << stars << std::endl;

        if (!root) {
            std::cout << treeName << " is empty :(\n";
            return;
        }

        std::cout << treeName << ":\n";

        std::function<void(const Node*, int, int, std::map<int, std::string>&)>
buildTreeLines =
            [&](const Node* node, int depth, int position, std::map<int, std::string>&
levels) {
                if (!node) return;
```

```
                std::ostringstream oss;
                oss << node->value;
                std::string value = oss.str();

                if (levels.count(depth) == 0) {
                    levels[depth] = std::string(position, ' ') + value;
                } else {
                    if (static_cast<int>(levels[depth].size()) < position) {
                        levels[depth] += std::string(position - levels[depth].size(), ' ')
+ value;
                    } else {
                        levels[depth] += value;
                    }
                }

                if (!node->children.empty()) {
                    int spacing = std::max(2, 6 - depth);
                    int leftPosition = position - spacing;
                    int rightPosition = position + spacing;

                    if (node->children.size() >= 1 && node->children[0]) {
                        if (static_cast<int>(levels[depth + 1].size()) < leftPosition) {
                            levels[depth + 1] += std::string(leftPosition - levels[depth +
1].size(), ' ') + "/";
                        } else {
                            levels[depth + 1] += "/";
                        }
                        buildTreeLines(node->children[0], depth + 2, leftPosition, levels);
```

```
                    }

                    if (node->children.size() >= 2 && node->children[1]) {
                        if (static_cast<int>(levels[depth + 1].size()) < rightPosition) {
                            levels[depth + 1] += std::string(rightPosition - levels[depth + 1].size(), ' ') + "\\";
                        } else {
                            levels[depth + 1] += "\\";
                        }
                        buildTreeLines(node->children[1], depth + 2, rightPosition, levels);
                    }
                }
            };

        std::map<int, std::string> levels;
        buildTreeLines(root, 0, 40, levels);

        for (const auto& [_, line] : levels) {
            std::cout << line << std::endl;
        }
    }

private:
    void MAKENULL() {
        for (auto it = begin(); it != end(); ) {
            const Node* current = it.current;
            ++it;
            delete current;
```

```
        }

        root = nullptr;

        // альтернативное решение через рекурсию обратным обходом
    }
};

int main() {
    Tree<double> A;
    const double valuesA[10] = {5.0, 3.0, 7.0, 2.0, 4.0, 6.0, 1.0, 8.0, 9.0, 10.0};
    for (int i = 0; i != 10; ++i) A.push(valuesA[i]);
    A.printTree("A");
    A.printPostOrderByIterator("A");

    Tree<double> B;
    const double valuesB[10] = {0.0, 3.0, 2.5, 4.0, 2.0, 1.5, 3.5, 4.5, 5.0, 1.0};
    for (int i = 0; i != 10; ++i) B.push(valuesB[i]);
    B.printTree("B");
    B.printPostOrderByRecursive("B");



    Tree<double>::myOperation(A, B);
    A.printTree("UNION A and B");
    A.printPostOrderByRecursive("UNION A and B");
    A.printPostOrderByIterator("UNION A and B");

    return 0;
```

6. Вывод

Эта практика помогла мне лучше понять, как работает дерево двоичного поиска и как его эффективно обходить с помощью разных методов, таких как рекурсивный и итеративный обход. Я научился реализовывать операцию объединения деревьев и понял, как важно сохранять структуру дерева при добавлении новых элементов. Также я понял, как использовать итераторы для удобного обхода и извлечения данных из дерева. Этот опыт улучшил мои навыки в управлении деревьями и их эффективном использовании в алгоритмах.

7. Литература

Кормен Т. Х., Лейзерсон Ч. Е., Ривест Р. L., Штайн К. — *Введение в алгоритмы*, стр. 539, Глава 12. Деревья поиска.

Хиршберг Д. С., Чьенг В. В. — *Деревья и алгоритмы на них*, стр. 186, Глава 7. Алгоритмы обхода и модификации деревьев.

Лекции и практики – преподаватель Филатов В. В.