МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет науки и технологий имени академика М.Ф. Решетнева»

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

Алгоритмы и структуры данных

Бинарные деревья

Руководитель _____ В. В. Тынченко
подпись, дата инициалы, фамилия

Обучающийся БПИ22-02, 221219040 _____ К.В. Трифонов
номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2023 г.

Цель работы:

изучение метода рекурсивной обработки данных;

знакомство с принципами организации и обработки бинарных деревьев.

Постановка задачи.

Выполнить задание в соответствии с вариантом. Для решения поставленной задачи разработать и использовать шаблон класса «Бинарное дерево поиска».

Для оценки «удовлетворительно».

Шаблон класса «Бинарное дерево поиска» должен включать в себя необходимый минимум методов, обеспечивающий полноценное функционирование объектов указанного класса, а именно:

- конструктор по умолчанию;
- конструктор копирования;
- деструктор;
- добавление узла;
- удаление узла;
- поиск узла с заданным значением ключа;

Используя механизм перегрузки стандартных операций, реализовать:

- вывод дерева на экран в удобном для восприятия человеком виде;
- ввод данных в дерево с клавиатуры;
- сравнение двух деревьев;
- присваивание данных одного дерева другому дереву;
- решение задачи согласно варианту задания.

Для оценки «хорошо».

Дополнить разработанный класс следующими функциональными возможностями:

- прямой обход дерева (корень, левое поддерево, правое поддерево);
- обратный обход дерева (левое поддерево, корень, правое поддерево);
- концевой обход дерева (левое поддерево, правое поддерево, корень).
- вывод дерева в файл в удобном для восприятия человеком виде;
- ввод данных в дерево из файла;

Для оценки «отлично».

На базе разработанного класса «Бинарное дерево поиска», реализовать производный класс «Сбалансированное бинарное дерево», добавив показатель

балансировки для узлов дерева и метод, выполняющий при необходимости повторную балансировку дерева.

Вариант №21.

Из входной последовательности целых чисел построить бинарное дерево поиска. Определить среднее арифметическое элементов самой длинной ветви. Если таких ветвей несколько, то взять для рассмотрения самую левую ветвь.

Код программы:

Описание класса узла поискового дерева Node:

```
// Узел поискового дерева
template<class T>
class Node
{
private:
    T _value;         // Значение узла
    Node* left;       // Указатель на левого потомка
    Node* right;      // Указатель на правого потомка

public:
    // Конструктор по умолчанию
    Node() : _value(T()), left(nullptr), right(nullptr) {}

    // Конструктор с параметрами
    Node(T value){
        _value = value;
        left = nullptr;
        right = nullptr;
    }

    // Методы для установки потомков и значения узла
    void SetLeft(Node* t) { left = t; }
    void SetRight(Node* t) { right = t; }
    void SetValue(T value) { _value = value; }

    // Методы для получения значения узла и указателей на потомков
    T GetValue() { return _value; }
    Node* GetLeft() { return left; }
    Node* GetRight() { return right; }
};
```

Описание класса поискового дерева

```
// Представление бинарного поискового дерева
template<class T>
class Tree
{
private:
    Node<T>* head;  // Указатель на корень дерева

private:
    bool search(T value, Node<T>* node);  // Поиск значения в дереве
    void add(Node<T>* node, T value);      // Добавление значения в дерево
    void delete_tree(Node<T>* t);          // Рекурсивное удаление дерева
```

```
    void print_unordered(Node<T>* t, ostream& out, int& c);  // Вывод дерева в
неупорядоченном порядке
    ostream& print_ordered(Node<T>* t, ostream& out);         // Вывод дерева в
упорядоченном порядке
    void print_koncevoe(Node<T>* t, ostream& out);            // Вывод кронцевого
обхода дерева
    Node<T>* RemoveNode(Node<T>* root, T x);                  // Удаление узла с
заданным значением
    void copyTree(Node<T>* newNode, Node<T>* otherNode);     // Копирование
дерева
    bool compareNodes(Node<T>* node1, Node<T>* node2);       // Сравнение узлов
деревьев

    // Определение среднего арифметического элементов самой длинной ветви
    int leng(Node<T>* node, T& maxSum, int& maxLength);

public:
    Tree();                                   // Конструктор по умолчанию
    Tree(T value);                            // Конструктор с корневым значением
    Tree(Node<T>* node);                      // Конструктор с заданным корневым
узлом
    Tree(const Tree& other);                  // Конструктор копирования
    ~Tree();                                  // Деструктор

    void Lenght(ostream& out);                // Вывод среднего арифметического
самой длинной ветви
    Node<T>* GetHead() { return head; }       // Получение указателя на корень
дерева
    void SetHead(Node<T>* head_) { head = head_; }  // Установка нового корня
дерева
    void print_unordered(ostream& out);       // Вывод дерева в неупорядоченном
порядке
    ostream& print_ordered(ostream& out);     // Вывод дерева в упорядоченном
порядке
    void print_koncevoe(ostream& out);        // Вывод кронцевого обхода дерева
    Tree* scan_file(string path);             // Создание дерева из файла

    Node<T>* RemoveNode(T x);                 // Удаление узла с заданным
значением
    void add(T value);                        // Добавление значения в дерево
    bool search(T value);                     // Поиск значения в дереве
    Tree<T>& operator = (const Tree<T>& other);  // Перегрузка оператора
присваивания
    bool operator == (const Tree<T>& other);      // Перегрузка оператора
сравнения
};
```

Конструкторы и деструкторы класса:

```
//Конструктор по умолчанию
template<class T>
Tree<T>::Tree(){
    head = nullptr;
}
//Конструктор с заданием значения
template<class T>
Tree<T>::Tree(T value){
    Node<T>* node = new Node<T>(value);
    head = node;
}
//Конструктор с заданием узла
template<class T>
Tree<T>::Tree(Node<T>* node) { head = node; }
//Конструктор копирования
template<class T>
Tree<T>::Tree(const Tree& other){
    if (other.head == nullptr){
        head = nullptr;
    }
    else{
        head = new Node<T>(other.head->GetValue());
        copyTree(head, other.head);
    }
}
//Деструктор
template<class T>
Tree<T>::~Tree(){
    delete_tree(head);
    head = nullptr;
}

//Рекурсия удаления дерева
template<class T>
void Tree<T>::delete_tree(Node<T>* t)
{
    if (t != NULL)
    {
        delete_tree(t->GetLeft());
        delete_tree(t->GetRight());
        delete t;
    }
}
```

Вывод на экран – (методы, вызывающие приватные рекурсивные методы:

```
//МЕТОДЫ ОБХОДА - вызов рекурсий
//Обратный (лево-корень-право)
template<class T>
void Tree<T>::print_unordered(ostream& out)
{
    if (&out == &cout)
        cout << "Обратный обход:\n"
            << "Формат: |шаг рекурсии|узел (Главный узел)\n";

    int c = 0;
    print_unordered(head, out, c);
}
//Прямой (корень-лево-право)
template<class T>
ostream& Tree<T>::print_ordered(ostream& out)
{
    if (&out == &cout)
        cout << "Прямой обход:\n";
    return print_ordered(head, out);
}
//Концевой (лево-право-корень)
template<class T>
void Tree<T>::print_koncevoe(ostream& out)
{
    if (&out == &cout)
        cout << "Концевой обход: " << endl;
    print_koncevoe(head, out);

    if (&out == &cout)
        cout << endl;
}
```

Рекурсии обходов:

```
//Рекурсии обходов:
//Рекурсия вывода обратным обходом
template<class T>
void Tree<T>::print_unordered(Node<T>* t, ostream& out, int &c)
{
    if (t != nullptr)
    {
        if (&out == &cout) {
            //Форматированный вывод с обозначением корня и глубины рекурсии
            int floor = c;
            c++;
            print_unordered(t->GetLeft(),out, c);
            if (floor == 0)
                cout << "(" << t->GetValue() << ") ";
```

```
            else
                cout << "|" << floor << "|" << t->GetValue() << " ";
            print_unordered(t->GetRight(), out, c);
        }
        else {
            print_unordered(t->GetLeft(), out, c);
            out << t->GetValue() << " ";
            print_unordered(t->GetRight(), out, c);
        }
    }
}
//Рекурсия вывода прямым обходом
template<class T>
ostream& Tree<T>::print_ordered(Node<T>* t, ostream& out)
{
    if (t != nullptr)
    {
        out << t->GetValue() << " ";
        print_ordered(t->GetLeft(), out);
        print_ordered(t->GetRight(), out);
    }
    return out;
}
//Рекурсия вывода концевым обходом
template<class T>
void Tree<T>::print_koncevoe(Node<T>* t, ostream& out)
{
    if (t != nullptr)
    {
        print_koncevoe(t->GetLeft(), out);
        print_koncevoe(t->GetRight(), out);
        out << t->GetValue() << " ";
    }
}
```

Ввод с файла:

```
//Чтение из файла
template<class T>
Tree<T>* Tree<T>::scan_file(string path)
{
    ifstream fin(path);
    T value;
    while (fin >> value)
    {
        this->add(value);
    }
    fin.close();
    return this;
}
```

Методы удаления:

```
//Вызов рекурсии удаления узла
template<class T>
Node<T>* Tree<T>::RemoveNode(T x){
    head = RemoveNode(head, x);
    return head;
}
//Рекурсивный поиск ноды и последующим её удалением
template<class T>
Node<T>* Tree<T>::RemoveNode(Node<T>* root, T x)
{
    if (root == NULL)
        return NULL;
    if (x == root->GetValue()){
        Node<T>* t;
        if (root->GetLeft() == NULL){
            t = root->GetRight();
            delete root;
            return t;
        }
        t = root->GetLeft();
        while (t->GetRight()){
            t = t->GetRight();
        }
        t->SetRight(root->GetRight());
        t = root;
        root = root->GetLeft();
        delete t;
        return root;
    }
    if (x < root->GetValue())
        root->SetLeft(RemoveNode(root->GetLeft(), x));
    else
        root->SetRight(RemoveNode(root->GetRight(), x));
    return root;
}
```

Методы добавления узла:

```
//Вызов добавления узла
template<class T>
void Tree<T>::add(T value){
    if (head == nullptr){
        Node<T>* newNode = new Node<T>(value);
        head = newNode;
    }
    else if (value >= head->GetValue()){
        if (head->GetRight() != nullptr)
            add(head->GetRight(), value);
        else{
            Node<T>* newNode = new Node<T>(value);
            head->SetRight(newNode);
        }
    }
    else{
        if (head->GetLeft() != nullptr)
            add(head->GetLeft(), value);
        else{
            Node<T>* newNode = new Node<T>(value);
            head->SetLeft(newNode);
        }
    }
}
```

```
//Добавление узла
template<class T>
void Tree<T>::add(Node<T>* node, T value){
    if (head == nullptr){
        Node<T>* newNode = new Node<T>(value);
        head = newNode;
    }
    else if (value >= node->GetValue()){
        if (node->GetRight() != nullptr)
            add(node->GetRight(), value);
        else{
            Node<T>* newNode = new Node<T>(value);
            node->SetRight(newNode);
        }
    }
    else{
        if (node->GetLeft() != nullptr)
            add(node->GetLeft(), value);
        else{
            Node<T>* newNode = new Node<T>(value);
            node->SetLeft(newNode);
        }
    }}}
```

Поиск узла:

```
//Вызов рекурсивного поиска
template<class T>
bool Tree<T>::search(T value){
    while (head != NULL){
        if (value == head->GetValue())
            return true;
        else{
            if (value <= head->GetValue())
                return search(value, head->GetLeft());
            else
                return search(value, head->GetRight());
        }
    }
    return false;
}
//Рекурсивный поиск
template<class T>
bool Tree<T>::search(T value, Node<T>* node){
    while (node != NULL){
        if (value == node->GetValue())
            return true;
        else{
            if (value <= node->GetValue())
                return search(value, node->GetLeft());
            else
                return search(value, node->GetRight());

        }
    }
    return false;
}
```

Перегрузки операторов для поискового дерева:

```
//Перегрузка оператора присваивания
template<class T>
Tree<T>& Tree<T>::operator=(const Tree<T>& other){
    if (this != &other){
        delete_tree(head);
        if (other.head == nullptr)
            head = nullptr;
        else
        {
            head = new Node<T>(other.head->GetValue());
            copyTree(head, other.head);
        }
    }
    return *this;
}
```

```cpp
//Перегрузка оператора равенства
template<class T>
bool Tree<T>::operator==(const Tree<T>& other)
{
    return compareNodes(head, other.head);
}
```

```cpp
//Перегрузка потоковых операторов:
template<class T>
ostream& operator << (ostream& out, Tree<T>& obj)
{
    int c = 0;
    obj.print_unordered(out);
    return out;
}
```

```cpp
template<class T>
istream& operator >> (istream& in, Tree<T>& obj)
{
    T value;
    int n;
    cout << "Введите кол-во узлов: ";
    cin >> n;
    for (int i = 0; i < n; i++) {
        in >> value;
        obj.add(value);
    }
    return in;
}
```

Описание класса узла AVL – дерева:

```
// Класс узла сбалансированного дерева
template<class T>
class BalancedNode : public Node<T>
{
private:
    int balance_factor; // Коэффициент балансировки
public:
    // Конструкторы
    BalancedNode() : Node<T>(), balance_factor(0) {}
    BalancedNode(T value) : Node<T>(value), balance_factor(0) {}

    // Методы для установки и получения коэффициента балансировки
    void SetBalanceFactor(int bf) { balance_factor = bf; }
    int GetBalanceFactor() { return balance_factor; }

    // Методы для установки потомков и значения узла
    void SetLeft(BalancedNode* t) { Node<T>::SetLeft(t); }
    void SetRight(BalancedNode* t) { Node<T>::SetRight(t); }
    void SetValue(T value) { Node<T>::SetValue(value); }

    // Методы для получения указателей на потомков
    BalancedNode* GetLeft() { return
static_cast<BalancedNode*>(Node<T>::GetLeft()); }
    BalancedNode* GetRight() { return
static_cast<BalancedNode*>(Node<T>::GetRight()); }
};
```

Описание класса AVL – дерева:

```
// Класс сбалансированного дерева
template<class T>
class BalancedTree : public Tree<BalancedNode<T>>
{
private:
    BalancedNode<T>* balancedHead; // Указатель на корень сбалансированного
дерева

    // Приватные методы для балансировки дерева
    BalancedNode<T>* LeftRotation(BalancedNode<T>* node);
    BalancedNode<T>* RightRotation(BalancedNode<T>* node);
    BalancedNode<T>* LeftRightRotation(BalancedNode<T>* node);
    BalancedNode<T>* RightLeftRotation(BalancedNode<T>* node);
    void UpdateBalanceFactor(BalancedNode<T>* node);
    BalancedNode<T>* BalanceTree(BalancedNode<T>* node);

    // Приватные методы для добавления элемента, вывода дерева и вычисления
высоты
    void Add(BalancedNode<T>* node, T value);
    void PrintTree(BalancedNode<T>* node, int& c);
```

```
    int GetHeight(BalancedNode<T>* node);

public:
    // Конструкторы и деструктор
    BalancedTree() : Tree<BalancedNode<T>>() {}
    BalancedTree(BalancedNode<T>* node) : Tree<BalancedNode<T>>(node) {}
    BalancedTree(const BalancedTree& other) : Tree<BalancedNode<T>>(other) {}
    ~BalancedTree() {}

    // Методы для установки и получения указателя на корень дерева
    void SetHead(BalancedNode<T>* node) { balancedHead = node; }
    BalancedNode<T>* GetHead() { return balancedHead; }

    // Методы для добавления элемента и вывода дерева
    void Add(T value);
    void PrintTree();
};
```

Повороты в сбалансированном дереве:

```
//ПОВОРОТЫ
//Малый левый
template<class T>
BalancedNode<T>* BalancedTree<T>::LeftRotation(BalancedNode<T>* node)
{
    BalancedNode<T>* temp = node->GetRight();
    node->SetRight(temp->GetLeft());
    temp->SetLeft(node);
    return temp;
}
//Малый правый
template<class T>
BalancedNode<T>* BalancedTree<T>::RightRotation(BalancedNode<T>* node)
{
    BalancedNode<T>* temp = node->GetLeft();
    node->SetLeft(temp->GetRight());
    temp->SetRight(node);
    return temp;
}
//Большой левый
template<class T>
BalancedNode<T>* BalancedTree<T>::LeftRightRotation(BalancedNode<T>* node){
    // Проверка наличия узла и его левого потомка
    if (node == nullptr || node->GetLeft() == nullptr){
        return node;
    }
    // Выполнение левого поворота для левого потомка
    node->SetLeft(LeftRotation(node->GetLeft()));
    // Выполнение правого поворота для исходного узла
    return RightRotation(node);
}
```

```
//Большой правый
template<class T>
BalancedNode<T>* BalancedTree<T>::RightLeftRotation(BalancedNode<T>* node){
    // Проверка наличия узла и его правого потомка
    if (node == nullptr || node->GetRight() == nullptr){
        return node;
    }
    // Выполнение правого поворота для правого потомка
    node->SetRight(RightRotation(node->GetRight()));
    // Выполнение левого поворота для исходного узла
    return LeftRotation(node);
}
```

Определение высот:

```
//Определение высоты
template<class T>
int BalancedTree<T>::GetHeight(BalancedNode<T>* node)
{
    if (node == nullptr)
        return 0;

    int leftHeight = GetHeight(node->GetLeft());
    int rightHeight = GetHeight(node->GetRight());

    return max(leftHeight, rightHeight) + 1;
}
```

Обновление высот:

```
//Обновление коэффициента балансировки
template<class T>
void BalancedTree<T>::UpdateBalanceFactor(BalancedNode<T>* node)
{
    // Вычисление высоты левого и правого поддеревьев узла
    int leftHeight = GetHeight(node->GetLeft());
    int rightHeight = GetHeight(node->GetRight());

    // Обновление коэффициента балансировки
    // (разница высот правого и левого поддеревьев)
    node->SetBalanceFactor(rightHeight - leftHeight);
}
```

Конструктор копий:

```
//Конструктор копирования
template<class T>
BalancedNode<T>* BalancedTree<T>::BalanceTree(BalancedNode<T>* node){
    UpdateBalanceFactor(node);
    int balanceFactor = node->GetBalanceFactor();
    if (balanceFactor > 1){
        if (node->GetRight()->GetBalanceFactor() < 0){
            return RightLeftRotation(node);
```

```
        }
        else{
            return LeftRotation(node);
        }
    }
    else if (balanceFactor < -1){
        if (node->GetLeft()->GetBalanceFactor() > 0){
            return LeftRightRotation(node);
        }
        else{
            return RightRotation(node);
        }
    }
    return node;
}
```

Добавление узла с балансировкой:

```
//Вызов рекурсивного добавления
template<class T>
void BalancedTree<T>::Add(T value)
{
    if (balancedHead == nullptr)
    {
        balancedHead = new BalancedNode<T>(value);
    }
    else
    {
        Add(balancedHead, value);
        balancedHead = BalanceTree(balancedHead);
    }
}

//Добавление узла
template<class T>
void BalancedTree<T>::Add(BalancedNode<T>* node, T value){
    if (node == nullptr){
        BalancedNode<T>* t = new BalancedNode<T>(value);
        node = t;
    }
    if (value < node->GetValue()){
        if (node->GetLeft() == nullptr){
            node->SetLeft(new BalancedNode<T>(value));
        }
        else{
            Add(node->GetLeft(), value);
            node->SetLeft(BalanceTree(node->GetLeft()));
        }
    }
    else if (value > node->GetValue()){
        if (node->GetRight() == nullptr){
```

```
            node->SetRight(new BalancedNode<T>(value));
        }
        else{
            Add(node->GetRight(), value);
            node->SetRight(BalanceTree(node->GetRight()));
        }
    }
}
```

Вывод в консоль:

```
//Вызов рекурсии вывода
template<class T>
void BalancedTree<T>::PrintTree()
{
    cout << "Обратный обход:\n"
        << "Формат: |шаг рекурсии|узел (Главный узел)\n";
    int c = 0;
    return PrintTree(balancedHead, c);
}

//Вывод с форматированием: |шаг рекурсии|узел (Главный узел) |шаг рекурсии|узел
template<class T>
void BalancedTree<T>::PrintTree(BalancedNode<T>* node,int& c){
    if (node != nullptr){
        //Форматированный вывод с обозначением корня и глубины рекурсии
        int floor = c;
        c++;
        PrintTree(node->GetLeft(), c);
        if (floor == 0)
            cout << "(" << node->GetValue() << ") ";
        else
            cout <<"|" << floor << "|" <<node->GetValue() << " ";
        PrintTree(node->GetRight(),c);
    }
}
```

Методы по варианту задания (для поискового дерева):

```
//Вызов рекурсии поиска суммы самой длинной ветки
template<class T>
void Tree<T>::Lenght(ostream& out) {

    int c = 0;
    T maxS = 0;
    int maxL = 0;
    leng(head, maxS, maxL);
    double avg = (maxS * 1.0) / maxL;
    if (&out == &cout)
        cout << "Среднее самой длинной ветки: ";
    cout << avg << "\n";
}
```

```
//Рекурсия поиска суммы
template<class T>
int Tree<T>::leng(Node<T>* node, T& maxSum, int& maxLength){
    if (node == nullptr) {
        maxSum = 0;
        maxLength = 0;
        return 0;
    }

    //Рекурсивно ищем в левом и правом поддеревьях
    T leftSum, leftLength, rightSum, rightLength;
    leftSum = leng(node->GetLeft(), maxLength, leftLength);
    rightSum = leng(node->GetRight(), maxLength, rightLength);

    //Максимальная сумма между левым и правым поддеревьями плюс текущий узел
    maxLength = max(leftLength, rightLength) + 1;

    if (leftLength >= rightLength) {
        maxSum = leftSum + node->GetValue();
    }
    else {
        maxSum = rightSum + node->GetValue();
    }

    return maxSum;
}
```

Рекурсия хранит переменные длины максимальной ветки и суммы максимальной ветки и постоянно их сравнивает через функцию max(), затем когда рекурсия достигает конца ветки, то полученные значения приходят обратно в функцию Length() - вызова рекурсивного метода, где происходит вычисление среднего путём деления суммы на количество.

Меню файла main.cpp:

```
cout << "Выберите тип дерева:\n"
        << "1) Поисковое дерево\n"
        << "2) AVL - дерево\n";
    int tr;
    cin >> tr;
    if (tr == 1) {
        //Поисковое дерево
        while (fl) {
            cout << "Главное меню\n"
                << "1) Ввод с консоли\n"
                << "2) Ввод из файла\n"
                << "3) Вывод в консоль\n"
                << "4) Вывод в файл\n"
                << "5) Добавление\n"
                << "6) Удаление\n"
                << "7) Поиск\n"
                << "8) Сравнение\n"
                << "9) Среднее самой длинной ветки\n"
                << "0) <<< Выход\n";
            cin >> ans;
            switch (ans){
```

Пользователю предлагается выбрать режим работы с поисковым деревом или с AVL – деревом, будут доступны для вызова методы соответствующего типа дерева.

```
Выберите тип дерева:
1) Поисковое дерево
2) AVL - дерево
```

Первый режим работы программы (работа с поисковым деревом):

Меню режима:

```
if (tr == 1) {
        //Поисковое дерево
        while (fl) {
            cout << "Главное меню\n"
                << "1) Ввод с консоли\n"
                << "2) Ввод из файла\n"
                << "3) Вывод в консоль\n"
                << "4) Вывод в файл\n"
                << "5) Добавление\n"
                << "6) Удаление\n"
                << "7) Поиск\n"
                << "8) Сравнение\n"
                << "9) Среднее самой длинной ветки\n"
                << "0) <<< Выход\n";
            cin >> ans;
```

```
Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
```

```
            switch (ans){
                //Ввод из консоли
                case 1:{
                    if (p.GetHead() != nullptr)
                        p.~Tree();
                    int size;
                    cout << "Введите искомое дерево (первый элемент считается
корнем):\n";
                    cin >> p;
                    break;
                }
                //Ввод из файла
                case 2:{
                    string filename;
                    if (p.GetHead() != nullptr)
                        p.~Tree();
                    cout << "Введите название файла ввода (первым элементом
должен быть корень):";
                    cin >> filename;
                    p.scan_file(filename);
                    break;
                }
                //Вывод в консоль
                case 3:{
                    cout << "\n" << p << "\n" << endl;
                    break;
                }
                //Вывод в файл
                case 4:{
                    int ans2;
                    cout << "Выберите вариант вывода в файл: " << endl
                        << "1) Прямой вывод в файл" << endl
                        << "2) Обратный вывод в файл" << endl
                        << "3) Концевой вывод в файл" << endl;
                    cin >> ans2;
                    string filename;
                    cout << "Введите название файла вывода: ";
                    cin >> filename;
                    switch (ans2){
                        case 1:{
```

```
                ofstream fout(filename);
                p.print_ordered(fout);
                fout.close();
                break;
            }
            case 2:{
                ofstream fout(filename);
                p.print_unordered(fout);
                fout.close();
                break;
            }
            case 3:{
                ofstream fout(filename);
                p.print_koncevoe(fout);
                fout.close();
                break;
            }
        }
        break;
    }
    //Добавление узла
    case 5:{
        int value;
        cin >> value;
        p.add(value);
        break;
    }
    //Удаление узла
    case 6:{
        int value;
        cin >> value;
        p.RemoveNode(value);
        break;
    }
    //Поиск узла
    case 7:{
        int value;
        cin >> value;
        if (p.search(value))
            cout << "\nЭлемент найден.\n" << endl;
        else
            cout << "\nЭлемент НЕ найден.\n" << endl;
        break;
    }
    //Сравнение деревьев

    case 8:{
        int inp;
        cout<<"Проверить оператор присваивания? 0 / 1\n";
        cin>>inp;
        if (inp){
```

```
                    Tree<int> p1 = p;
                    if (p1 == p)
                        cout<<"Успешное копирование\n";
                    else
                        cout<<"Ошибка.\n";
                }
                bool check = false;
                cout<<"Заполните второе дерево сравнения:\n";
                cin >> p1;
                if (p == p1){
                    cout << "Одинаковые\n";
                    check = true;
                }
                else{
                    cout << "Различные\n";
                }
                cout << endl;
                break;
            }
```

Пользователю предлагается выбрать запускать ли проверку оператора присваивания (для проверки работоспособности), затем выполняется заполнение второго дерева с последующим сравнением через перегруженный оператор равенства.

```
            //Среднее арифметическое самой длинной ветки
            case 9: {
                p.Lenght(cout);
                break;
            }
            case 0: {
                fl = false;
                break;
            }
        }
    }
}
```

По выбору пользователя выполняется нужная операция с вызовом соответствующего метода.

Второй режим работы программы (работа с AVL - деревом):

Меню режима:

```
else {
        //AVL - дерево
        while (fl) {
            cout <<"Главное меню\n"
                << "1) Ввод с консоли\n"
                << "2) Вывод в консоль\n"
                << "3) Добавление\n"
                << "0) <<< Выход\n";
            cin >> ans;
```

```
Главное меню
1) Ввод с консоли
2) Вывод в консоль
3) Добавление
0) <<< Выход
```

```
            switch (ans) {

    //Заполнение с консоли
    case 1: {
        int size;
        int value;
        cout << "Введите кол-во узлов: ";
        cin >> size;
        for (int i = 0; i < size; i++)
        {
            cout << "Введите значение узла: ";
            cin >> value;
            p3.Add(value);
        }
        break;
    }
```

Пользователь вводит количество узлов, затем заполняет узлы через пробел.

```
    //Вывод в консоль
    case 2: {
        p3.PrintTree();
        cout << endl;
        break;
    }
    //Добавление узла в дерево с балансировкой
    case 3: {
        int value;
        cout << "Введите элемент, который хотите добавить в дерево: ";
        cin >> value;
        p3.Add(value);
        break;
    }
```

```
    case 0: {
        fl = false;
        break;
    }
}
}
```

Тестирование дерева поиска

Пример заполнения дерева поиска длинной 5 корнем 50 и элементами: 100, 25, 70, 120 с последующим выводом обратным обходом на экран:

```
1
Введите искомое дерево (первый элемент считается корнем):
Введите кол-во узлов: 5
50
100
25
70
120
Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
3

Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|1|25 (50) |3|70 |2|100 |4|120
```

Добавление узла 50 в дерево: |2|1 |1|12 (25) |4|44 |3|70

```
Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|2|1 |1|12 (25) |4|44 |3|70

Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
5
50
Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
3

Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|2|1 |1|12 (25) |4|44 |5|50 |3|70
```

Удаление узла 12 из дерева: |2|1 |1|12 (25) |4|44 |5|50 |3|70:

```
Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|2|1 |1|12 (25) |4|44 |5|50 |3|70

Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
6
12
Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
3

Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|1|1 (25) |3|44 |4|50 |2|70
```

Вывод в файл ouput.txt дерева: |2|11 |1|50 |3|90 (125) |4|250:

```
Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|2|11 |1|50 |3|90 (125) |4|250

Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
4
Выберите вариант вывода в файл:
1) Прямой вывод в файл
2) Обратный вывод в файл
3) Концевой вывод в файл
1
Введите название файла вывода: out.txt
```

Содержимое файла out.txt при прямом выводе главный – лево - право:

```
out.txt
125 50 11 90 250
```

Содержимое файла out.txt при обратном выводе лево – главный - право:

```
out.txt
11 50 90 125 250
```

Содержимое файла out.txt при концевом выводе лево – право - главный:

```
out.txt
11 90 50 250 125
```

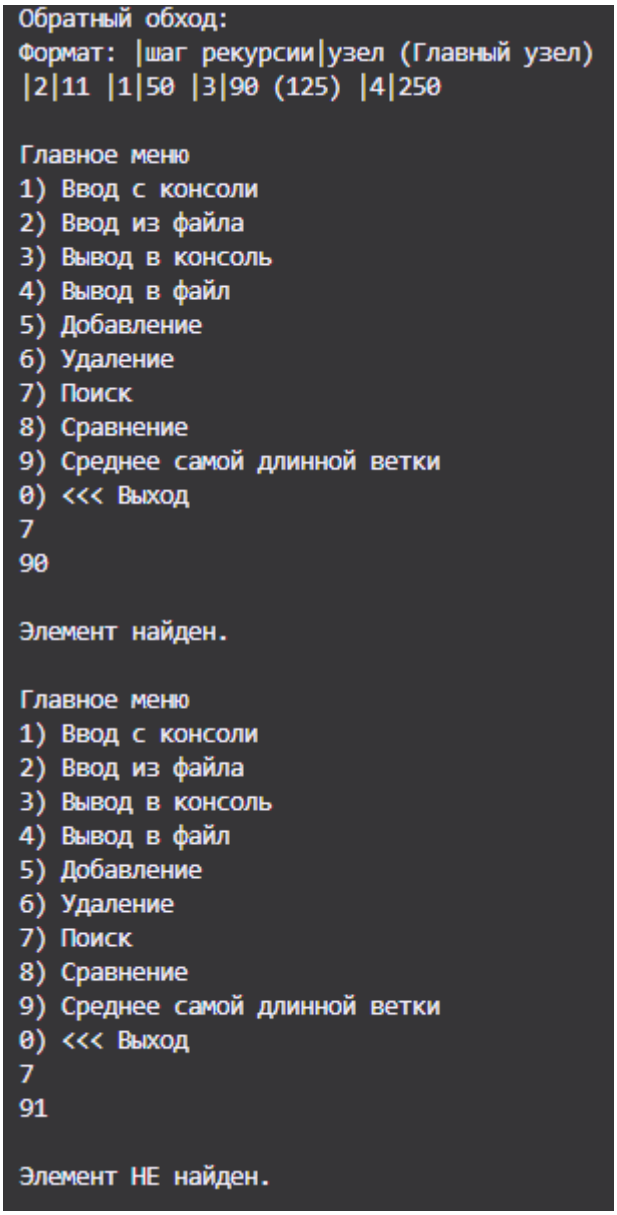
Чтение из файла out.txt (перезапись имеющихся данных):

```
Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|1|2 |2|5 (34) |3|75 |4|90

Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
2
Введите название файла ввода (первым элементом должен быть корень):out.txt
Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
3

Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|2|11 |1|50 |3|90 (125) |4|250
```

Пример работы поиска узлов в дереве:



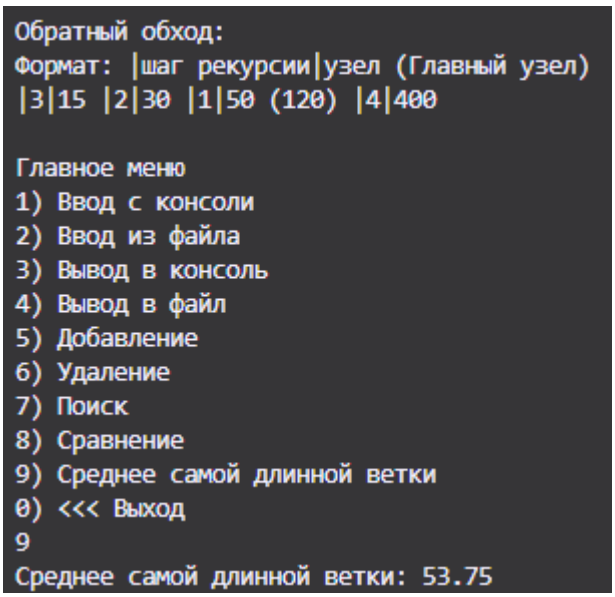
Пример работы проверки оператора присваивания и сравнения двух деревьев:

```
Введите искомое дерево (первый элемент считается корнем):
Введите кол-во узлов: 5
100 25 88 91 250
Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
3

Обратный обход:
Формат: |шаг рекурсии|узел (Главный узел)
|1|25 |2|88 |3|91 (100) |4|250

Главное меню
1) Ввод с консоли
2) Ввод из файла
3) Вывод в консоль
4) Вывод в файл
5) Добавление
6) Удаление
7) Поиск
8) Сравнение
9) Среднее самой длинной ветки
0) <<< Выход
8
Проверить оператор присваивания? 0 / 1
1
Успешное копирование
Заполните второе дерево сравнения:
Введите кол-во узлов: 5
100 25 88 91 250
Одинаковые
```

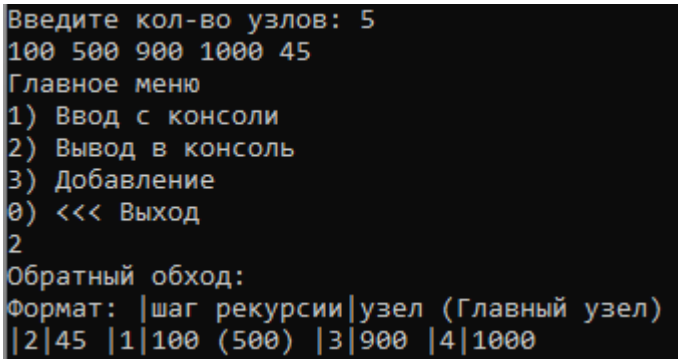
Работа функции по варианту на дереве: |3|15 |2|30 |1|50 (120) |4|400 – **определение среднего арифметического самой длинной ветки:**



Самой длинной веткой была определена левая (2 наследника имеет только центральный узел), соответственно среднее арифметическое для элементов дерева 120, 50, 30, 15 = 53,75.

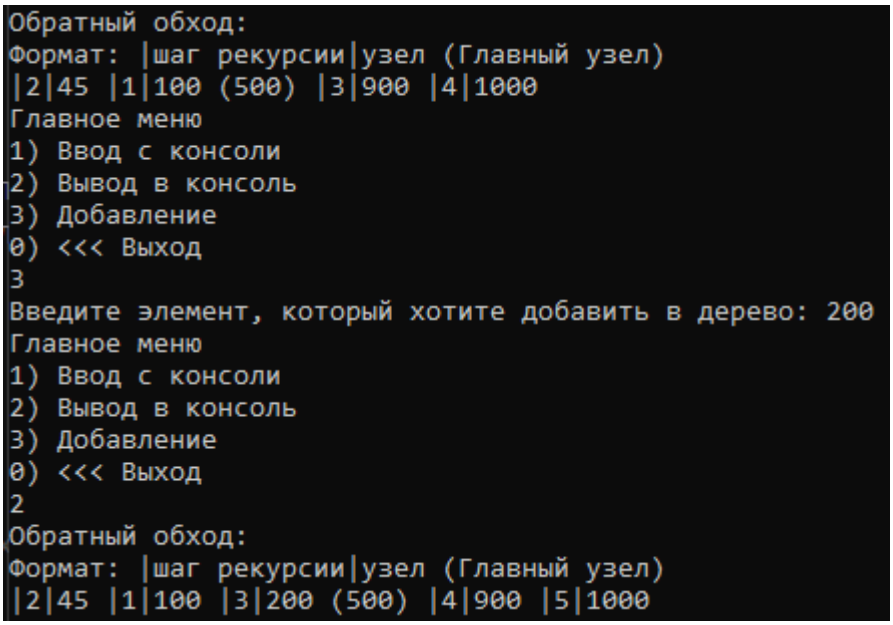
Тестирование сбалансированного дерева

Пример заполнения сбалансированного дерева элементами 100 500 900 1000 45:



Дерево сбалансировалось и главным узлом стал элемент “500”.

Пример добавления узла “200” к дереву: |2|45 |1|100 |3|200 (500) |4|900 |5|1000:



Ответы на контрольные вопросы

1. **Опишите общий принцип организации бинарного дерева поиска. В чем заключается преимущество подобной организации данных?**

Бинарное дерево поиска (BST) организовано так, что для каждого узла левое поддерево содержит значения меньшие, чем узел, а правое поддерево содержит значения большие. Преимущество заключается в том, что операции поиска, вставки и удаления выполняются в среднем за время O(log n), что делает BST эффективной структурой данных для упорядоченных данных.

2. **Опишите алгоритм построения бинарного дерева поиска?**

Алгоритм включает в себя поочередное добавление каждого элемента в дерево. Для каждого элемента проверяется, больше или меньше он текущего узла. Если больше, то идем вправо, если меньше - влево. Если соответствующего потомка нет, создаем новый узел. Процесс повторяется для каждого элемента.

3. **Как выполняется поиск данных в бинарном дереве?**

Поиск данных в бинарном дереве начинается с корня. Сравниваем искомое значение с текущим узлом. Если оно равно, поиск завершен. Если значение меньше, идем влево; если больше - вправо. Процесс повторяется до тех пор, пока не найден узел с искомым значением или не достигнут конец дерева.

4. **Опишите алгоритм удаления узла из бинарного дерева.**

    - Если у удаляемого узла нет потомков, удаляем его просто.
    - Если у узла есть один потомок, заменяем узел его потомком.
    - Если у узла два потомка, находим минимальный узел в правом поддереве (или максимальный в левом), заменяем удаляемый узел этим значением, а затем рекурсивно удаляем минимальный (максимальный) узел.

5. **Перечислите и охарактеризуйте различные способы обхода бинарного дерева.**

    - **Прямой (корень-лево-право):** Посещаем узел, затем рекурсивно обходим левое и правое поддеревья.
    - **Обратный (лево-корень-право):** Рекурсивно обходим левое поддерево, затем посещаем узел, и, наконец, рекурсивно обходим правое поддерево.

- **Концевой (лево-право-корень):** Рекурсивно обходим левое и правое поддеревья, затем посещаем узел.

Каждый из этих методов обхода предоставляет разные порядки посещения узлов, что может быть полезно в различных сценариях работы с данными в деревьях.

Вывод

Были получены навыки разработки и использования поисковых и AVL деревьев, с помощью рекурсивных методов обработки данных, а также умением расширять функциональность классов в ООП. В ходе тестирования была проверена корректность работы реализованных методов, включая добавление элементов, поиск, удаление и вывод дерева в различных порядках обхода, нахождение самой длинной ветки и вычисление её среднего арифметического.