МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждения высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практической работы № 7.1

Тема:

«Балансировка дерева поиска»

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Туляшева А.Т.

Группа: ИКБО-42-23

Москва - 2024

Содержание

ЦЕЛЬ РАБОТЫ

Цель: получение умений и навыков разработки и реализаций операций над структурой данных бинарное дерево.

ЗАДАНИЕ 1

Ответьте на вопросы.

1. Что определяет степень дерева?

Степень дерева — это максимальное количество дочерних узлов, которые могут быть у любого узла в дереве.

2. Какова степень сильноветвящегося дерева?

Степень сильноветвящегося дерева — это степень дерева с наибольшим количеством дочерних узлов среди всех узлов дерева.

3. Что определяет путь в дереве?

Путь в дереве — это последовательность узлов, ведущая от корня дерева до заданного узла или между двумя узлами.

4. Как рассчитать длину пути в дереве?

Длина пути в дереве равна количеству ребер между начальным и конечным узлом пути.

5. Какова степень бинарного дерева?

Степень бинарного дерева — это 2, так как каждый узел может иметь максимум два дочерних узла (левый и правый).

6. Может ли дерево быть пустым?

Да, дерево может быть пустым, когда в нем нет ни одного узла.

7. Дайте определение бинарного дерева.

Бинарное дерево — это дерево, каждый узел которого имеет не более двух дочерних узлов.

8. Дайте определение алгоритму обхода.

Алгоритм обхода дерева — это метод последовательного посещения всех узлов дерева, например, обход в глубину (префиксный, инфиксный, постфиксный) или обход в ширину.

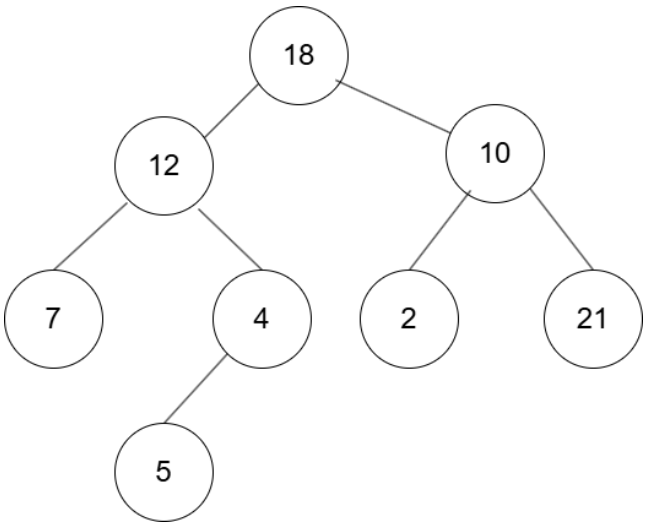
9. Что такое высота дерева? Приведите рекуррентную зависимость для вычисления высоты дерева.

Высота дерева — это максимальная длина пути от корня до самого глубокого узла. Рекуррентная формула:

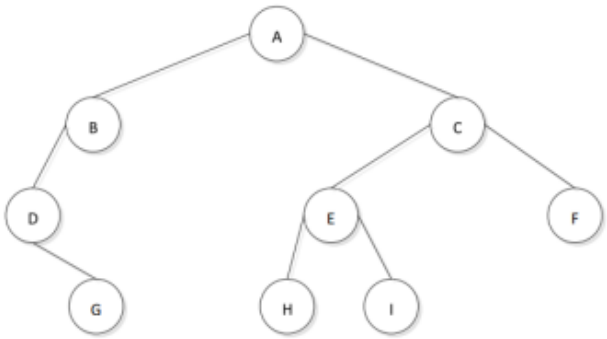
Высота дерева есть максимальная из высот правого и левого деревьев, увеличенная на единицу.

10. Изобразите бинарное дерево, корень которого имеет индекс 6, и которое представлено в памяти таблицей вида:

Индекс	*key*	*left*	*right*
1	12	7	3
2	15	8	NULL
3	4	10	NULL
4	10	5	9
5	2	NULL	NULL
6	18	1	4
7	7	NULL	NULL
8	14	6	2
9	21	NULL	NULL
10	5	NULL	NULL



11. Укажите путь обхода дерева по алгоритмы: прямой, обратный, симметричный



1. Прямой (Pre-order, NLR): Корень → Левый потомок → Правый потомок. Порядок обхода: A, B, D, G, C, E, H, I, F

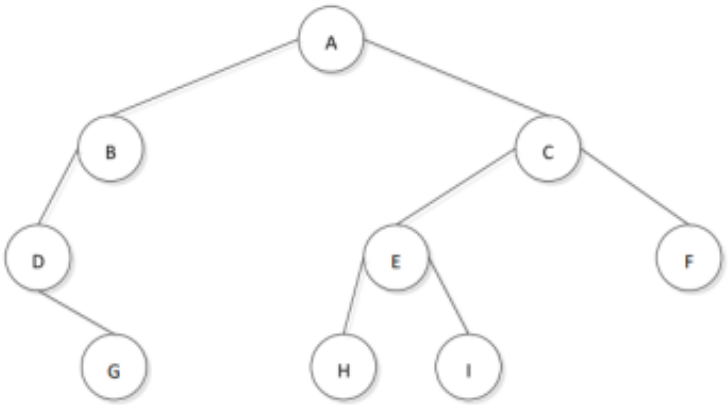
2. Обратный (Post-order, LRN): Левый потомок → Правый потомок → Корень. Порядок обхода: G, D, B, H, I, E, F, C, A

3. Симметричный (In-order, LNR): Левый потомок → Корень → Правый потомок. Порядок обхода: G, B, A, H, E, I, C, F

12. Какая структура используется в алгоритме обхода дерева методом в «ширину»?

Для обхода дерева в ширину (по уровням) используется очередь (queue). Этот метод предполагает обработку узлов на каждом уровне дерева слева направо.

13. Выведите путь при обходе дерева в «ширину». Продемонстрируйте использование структуры при обходе дерева.



Используя очередь, обход дерева по ширине будет следующим:

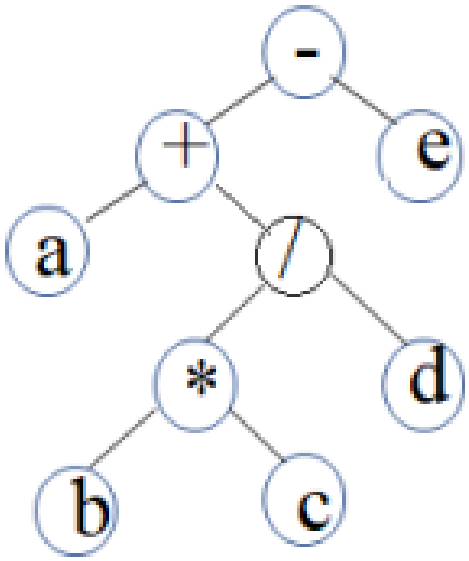
- Начинаем с корня A, добавляем его в очередь.

- Извлекаем A и добавляем его потомков (B и C) в очередь.
- Извлекаем B и добавляем его потомков (D) в очередь.
- Извлекаем C и добавляем его потомков (E и F) в очередь.
- Извлекаем D и добавляем его потомка (G) в очередь.
- Извлекаем E и добавляем его потомков (H и I) в очередь.
- Извлекаем F, затем G, H и I (у них нет потомков).
- Путь обхода: A, B, C, D, E, F, G, H, I.

14. Какая структура используется в не рекурсивном обходе дерева методом в «глубину»?

Для обхода в глубину без рекурсии используется стек (stack). Стек хранит узлы, которые нужно посетить, начиная с корня.

15. Выполните прямой, симметричный, обратный методы обхода дерева выражений.



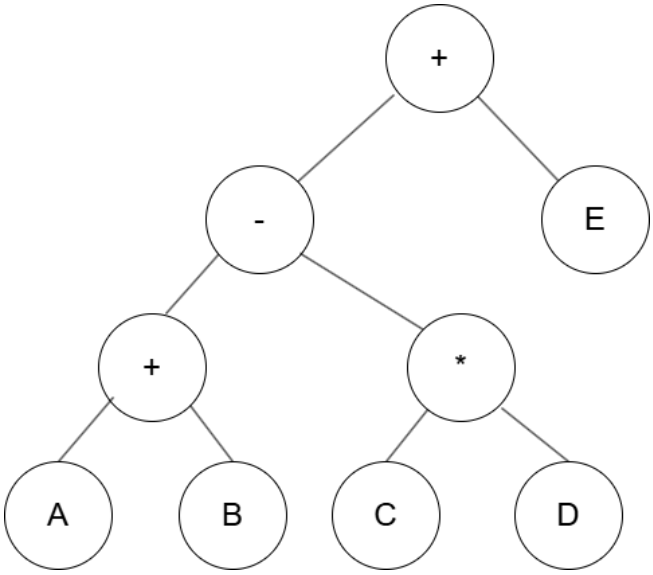
Прямой обход (Pre-order, NLR): - + a / * b c d e

Симметричный обход (In-order, LNR): a + b * c / d - e

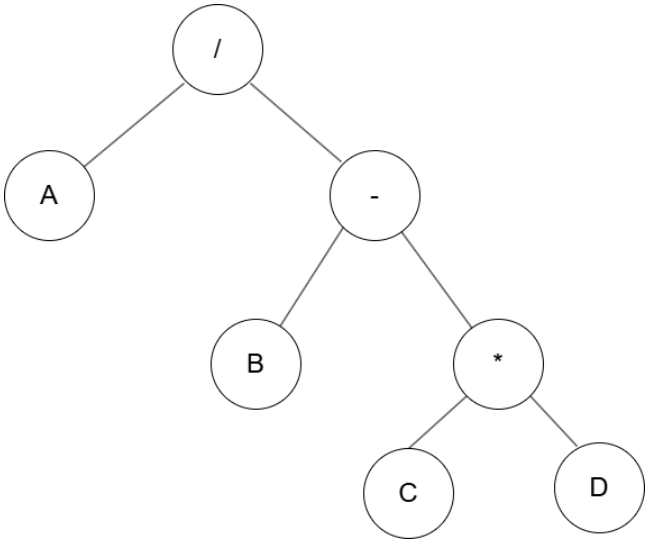
Обратный обход (Post-order, LRN): a b c * d / + e –

16. Для каждого заданного арифметического выражения постройте бинарное дерево выражений:

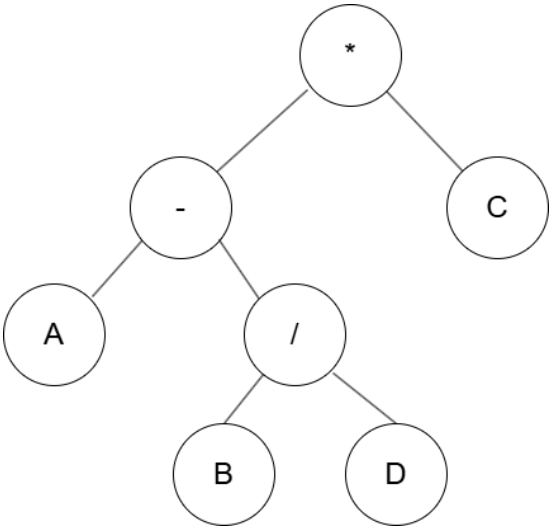
1) a+b-c*d+e



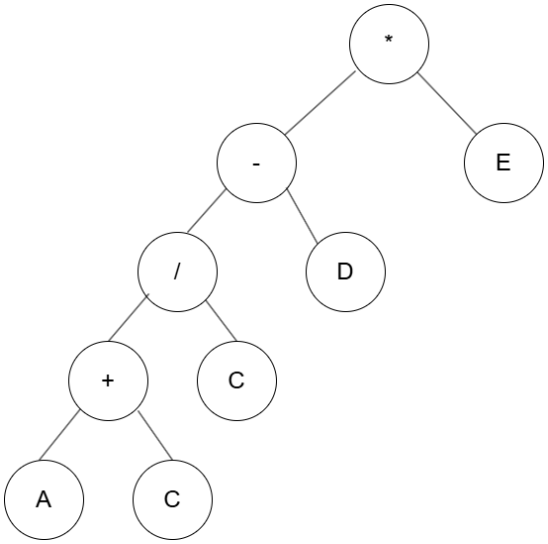
2) /a-b*c d



3) a b c d / - *



4) * - / + a b c d e



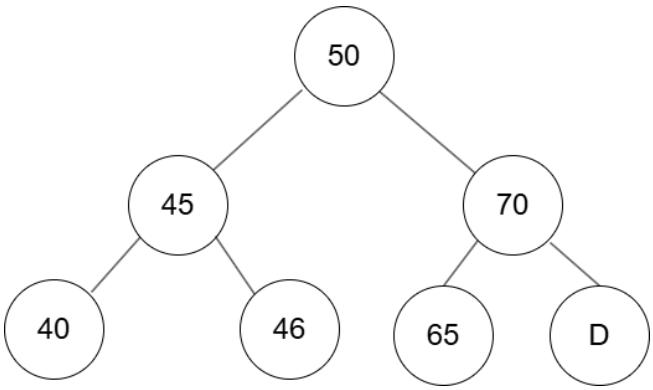
17. В каком порядке будет проходиться бинарное дерево, если алгоритм обхода в ширину будет запоминать узлы не в очереди, а в стеке?

Обход дерева в ширину обычно запоминает узлы в очереди, что приводит к посещению узлов уровня за уровнем слева направо. Если использовать стек вместо очереди, порядок обхода изменится, приближаясь к обходу в глубину, но с некоторыми особенностями.

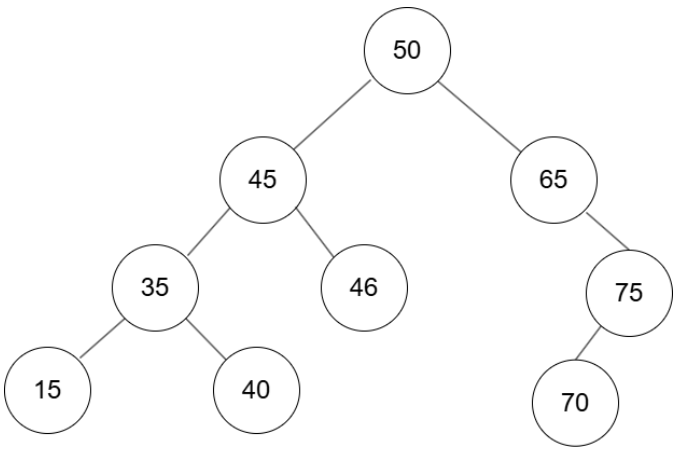
Использование стека для обхода в ширину приведет к следующему поведению:

1. Узлы будут извлекаться в обратном порядке их добавления.
2. При добавлении узлов в стек сначала добавляются правые потомки, затем левые, чтобы при извлечении они были посещены в порядке слева направо.

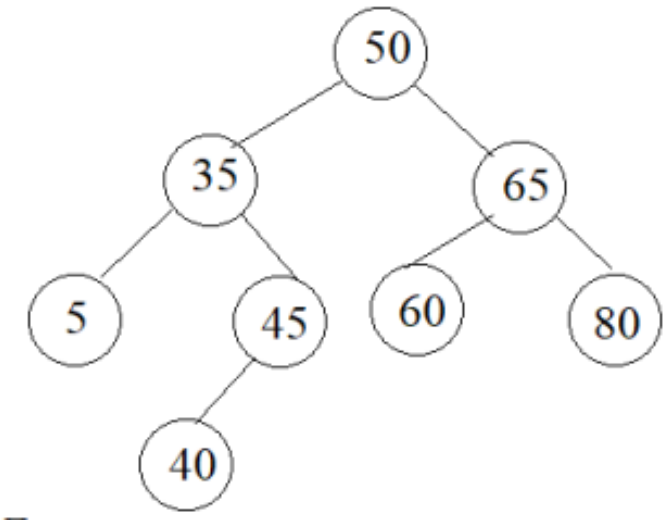
18. Постройте бинарное дерево поиска, которое в результате симметричного обхода дало бы следующую последовательность узлов: 40 45 46 50 65 70 75



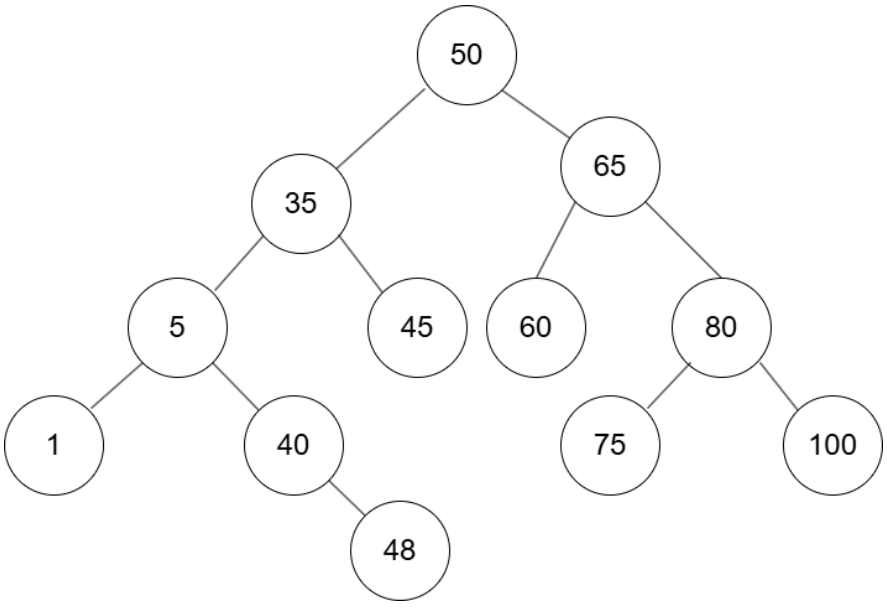
19. Приведенная ниже последовательность получена путем прямого обхода бинарного дерева поиска: 50 45 35 15 40 46 65 75 70. Постройте это дерево.



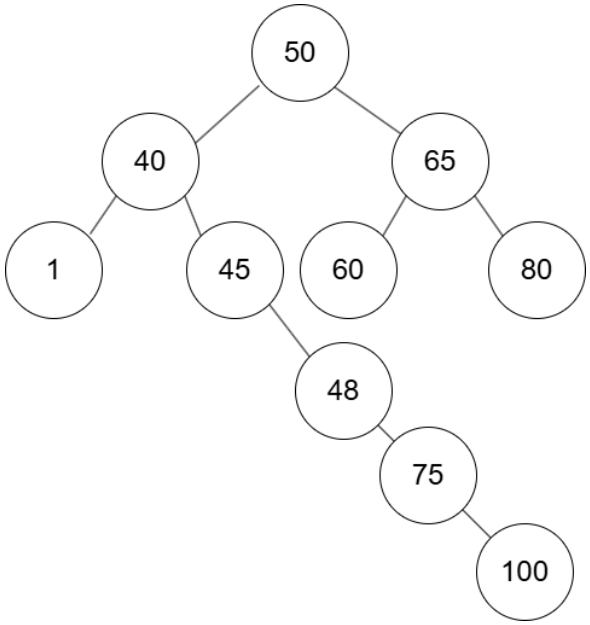
20. Дано следующее бинарное дерево поиска



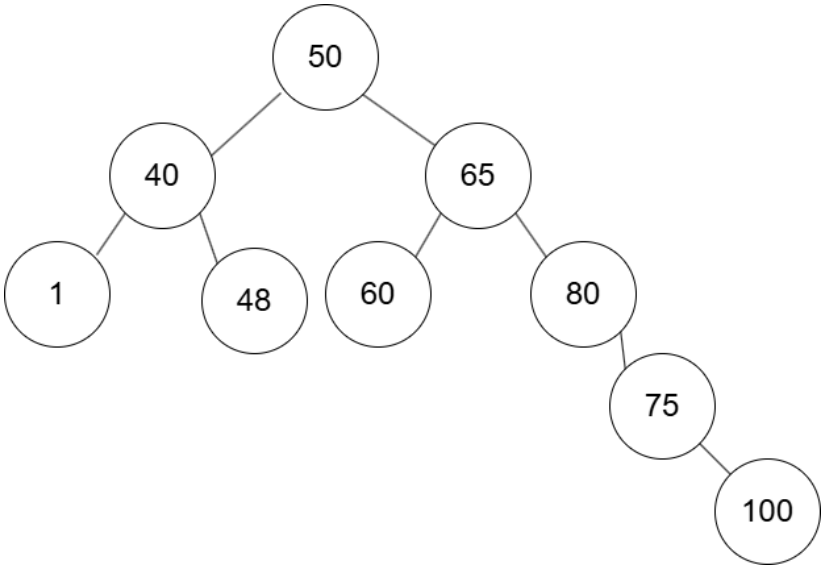
1) после включения узлов 1, 48, 75, 100



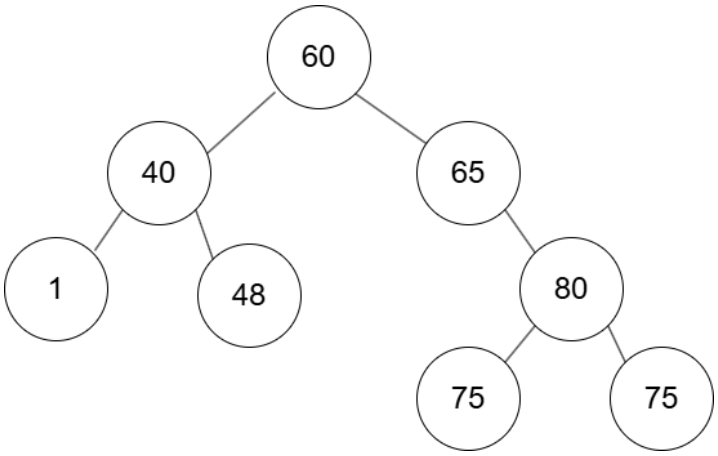
2) после удаления узлов 5, 35



3) после удаления узла 45

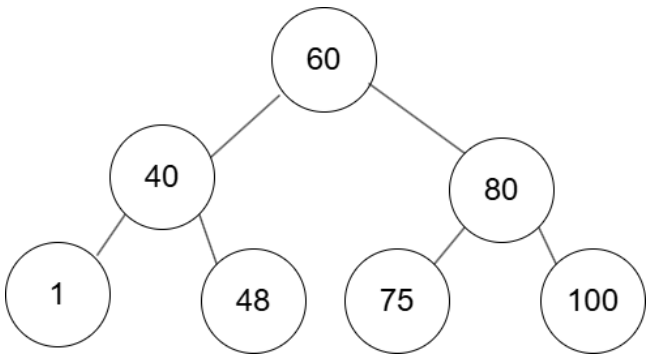


4) после удаления узла 50

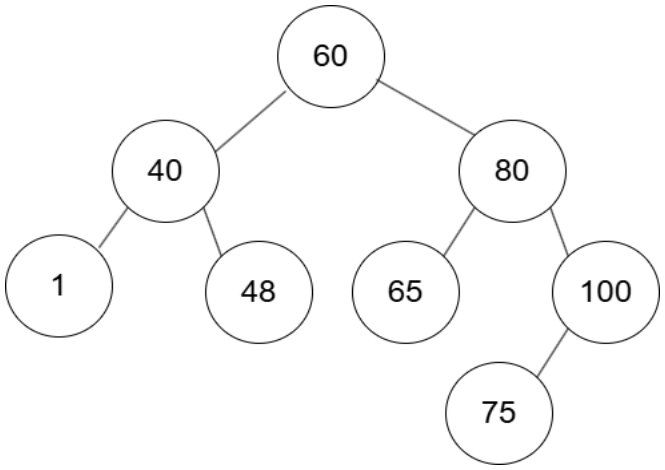


5) после удаления узла 65 и вставки его снова

1. Удаление узла 65



2. Возврат узла 65



ЗАДАНИЕ 2

Разработать программу бинарного дерева поиска и выполнить операции в соответствии с требованиями варианта. Методы, которые должны быть реализованы в независимости от варианта: включение элемента в дерево; поиск ключа в дереве; удаление ключа из дерева; отображение дерева. Выполните реализацию средствами ООП, операции будут методами класса.

Вариант – 1:

1	Целое число	Определить высоту дерева Определить длину пути дерева (количество ребер), используя алгоритм прямого обхода Вычисляет среднее арифметическое всех чисел в дереве.

2.1 Математическая модель решения (описание алгоритма)

1. insert(int key) – добавление узла, в качестве аргумента передается ключ (целое число). Создает новый узел и находит подходящее место для его вставки. Если узел меньше текущего, происходит переход в левое поддерево, иначе — в правое.

2. search(int key) – поиск по ключу. Перемещение по дереву, пока ключ не будет найден, либо возврат -1.

3. remove(int key) – удаление по ключу. Рассматривается три варианта: когда узел не имеет поддеревьев, имеет одно поддерево и имеет два поддерева.

4. height() – высота дерева. Рассчитывается как наибольшая длина пути от корня дерева до его листа.

5. edgeLength() – длина пути (количество ребер). Реализовано через обход всего дерева.

6. average() – среднее арифметическое всех чисел в дереве. Использует очередь для уровня. При обходе увеличивает сумму и счетчик узлов.

7. print(Node* t, int u) (в качестве аргументов принимает принимая указатель на узел и текущий уровень) и display() – отображение дерева. Отображает все элементы дерева в порядке их обхода. Использует стек для обхода и печатает значения узлов.

2.2 Код программы

```
struct Node { //структура для узла дерева
    int key; //инфорационное поле - целое число
    Node* left; //левое поддерево
    Node* right; //правое поддерево
    Node(int val) : key(val), left(nullptr), right(nullptr) {} //конструктор
};
```

Рис. 1 – Структура узла

В классе BinarySearchTree для бинарного дерева поиска реализованы различные методы (рис. 2-5).

```cpp
class BinarySearchTree { //класс для бинарного дерева поиска
private:
    Node* root; //корень дерева

public:
    BinarySearchTree() : root(nullptr) {} //конструктор для создания пустого дерева

    void insert(int key) { //добавление узла
        Node* newNode = new Node(key); //создаем новый узел
        if (root == nullptr) { //если дерево пустое, то новый узел - корень
            root = newNode;
            return;
        }
        Node* current = root; //для перемещения по дереву
        Node* parent = nullptr; //родитель
        while (current != nullptr) { //ищем место для нового узла
            parent = current; //запоминаем родителя
            if (key < current->key) { current = current->left; } //переход в левое поддерево
            else { current = current->right; } //в правое поддерево
        }
        if (key < parent->key) { parent->left = newNode; } //добавляем в левое поддерево
        else { parent->right = newNode; } //добавляем в правое поддерево
    }

    int search(int key) { //поиск по ключу
        Node* current = root; //для перемещения по дереву
        while (current != nullptr) { //пока не достигли конца дерева
            if (current->key == key) { return current->key; } //ключ найден
            else if (key < current->key) { current = current->left; } //поиск слева
            else { current = current->right; } //поиск справа
        }
        return -1; //не найден ключ
    }
```

Рисунок 2 – Конструктор и методы insert и search

```cpp
void remove(int key) { //удаление
    Node* current = root; //для перемещения по дереву
    Node* parent = nullptr; //родитель
    while (current != nullptr && current->key != key) { //ищем узел для удаления
        parent = current; //запоминаем родителя
        if (key < current->key) current = current->left; //двигаемся влево
        else current = current->right; //двигаемся вправо
    }
    if (current == nullptr) return; //узел не найден
    //узел не имеет поддеревьев
    if (current->left == nullptr && current->right == nullptr) {
        if (current == root) root = nullptr; //узел - корень
        else if (parent->left == current) parent->left = nullptr; //узел - левый ребенок
        else parent->right = nullptr; //узел - правый ребенок
    }
    //узел имеет одно поддерево
    else if (current->left == nullptr || current->right == nullptr) {
        Node* child = (current->left != nullptr) ? current->left : current->right; //ребенок
        if (current == root) root = child; //если удаляем корень
        else if (parent->left == current) parent->left = child; //узел - левый ребенок
        else parent->right = child; //узел - правый ребенок
    }
    //узел имеет два поддерева
    else { //ищем наименьший узел в правом поддереве
        Node* successor = current->right; //начинаем с правого поддерева
        Node* successorParent = current; //родитель для узла-наследника
        while (successor->left != nullptr) { //ищем наименьший узел
            successorParent = successor;
            successor = successor->left;
        }
        current->key = successor->key; //копируем значение узла-наследника
        if (successorParent->left == successor) { //удаляем узел-наследник
            successorParent->left = successor->right; //подключаем правое поддерево
        }
        else successorParent->right = successor->right; //подключаем правое поддерево
        current = successor; //текущий узел указывает на узел-наследник
    }
    delete current; //удаляем узел
}
```

Рисунок 3 – Метод remove

```
int height() { //высота дерева
    if (root == nullptr) return -1; //дерево пустое
    int h = 0; //высота
    Node* queue[100]; //очередь для выполнения обхода
    int front = -1, rear = -1; //индексы вершины и конца очереди
    queue[++rear] = root; //добавляем корень в очередь
    while (front != rear) {
        int count = rear - front;
        h++;
        while (count--) {
            Node* node = queue[++front]; //извлекаем узел из очереди
            if (node->left != nullptr) queue[++rear] = node->left; //добавляем левого ребенка в очередь
            if (node->right != nullptr) queue[++rear] = node->right; //добавляем правого ребенка в очередь
        }
    }
    return h;
}

int edgeLength() { //длина пути (кол-во ребер)
    if (root == nullptr) return 0; //дерево пустое
    int length = 0; //длина
    Node* queue[100]; //очередь для обхода
    int front = -1, rear = -1; //индексы вершины и конца очереди
    queue[++rear] = root; //добавляем корень в очередь
    while (front != rear) {
        int count = rear - front;
        while (count--) {
            Node* node = queue[++front]; //извлекаем узел из очереди
            if (node->left != nullptr) {
                queue[++rear] = node->left; //добавляем левого ребенка в очередь
                length++; //увеличиваем длину
            }
            if (node->right != nullptr) {
                queue[++rear] = node->right; //добавляем правого ребенка в очередь
                length++; //увеличиваем длину
            }
        }
    }
    return length;
}
```

Рисунок 4 – Методы height и edgeLength

```
double average() { //среднее арифметическое
    if (root == nullptr) return 0; //дерево пустое
    int sum = 0; //для хранения суммы всех узлов
    int count = 0; //для хранения количества узлов
    Node* queue[100]; //очередь для обхода
    int front = -1; //индекс передней части очереди
    int rear = -1; //индекс задней части очереди
    queue[++rear] = root; //добавляем корень дерева в очередь
    while (front != rear) {
        Node* node = queue[++front]; //извлекаем узел из очереди
        sum += node->key; //добавляем значение узла к сумме
        count++; //увеличиваем счетчик узлов
        //добавляем детей узла в очередь
        if (node->left != nullptr) queue[++rear] = node->left; //левое поддерево
        if (node->right != nullptr) queue[++rear] = node->right; //правое поддерево
    }
    return count != 0 ? (double)sum / count : 0;
}

void print(Node* t, int u) {
    if (t == nullptr) return;
    print(t->left, u + 1); //обход левого поддерева
    for (int i = 0; i < u; ++i) { cout << "| "; }
    cout << t->key << endl;
    print(t->right, u + 1); //обход правого поддерева
}
void display() { //вывод
    if (root == nullptr) {
        cout << "Дерево пустое" << endl;
        return;
    }
    print(root, 0); //начало с корня
}
```

Рисунок 5 – Методы average, print и display

```cpp
int main() {
    setlocale(LC_ALL, "rus");
    BinarySearchTree tree;
    int n; //размер
    int r;
    cout << "Как заполнить дерево? Вручную - 1, рандомно - 2: ";
    cin >> r;
    cout << "\nВставка элементов в дерево\nВведите количество элементов: ";
    cin >> n;
    switch (r) {
    case 1:
        int q;
        cout << "Введите элементы для вставки:\n";
        for (int i = 0; i < n; i++) {
            cin >> q;
            tree.insert(q);
        }
        break;
    case 2:
        srand(time(0));
        for (int i = 0; i < n; ++i) {
            int randomValue = rand() % 1000;
            tree.insert(randomValue);
        }
        break;
    }
    cout << "Дерево:\n";
    tree.display();
    cout << "Среднее арифметическое всех чисел в дереве: " << tree.average() << endl;
    cout << "Длина пути дерева (количество ребер): " << tree.edgeLength() << endl;
    cout << "Высота дерева: " << tree.height() << endl;
    cout << "Поиск по ключу: " << tree.search(2);
    cout << "\nКакой элемент удалить? ";
    int w;
    cin >> w;
    tree.remove(w);
    cout << "Дерево после удаления:\n";
    tree.display();
}
```

Рисунок 6 – main

2.3 Тестирование программы

```
Как заполнить дерево? Вручную - 1, рандомно - 2: 1

Вставка элементов в дерево
Введите количество элементов: 5
Введите элементы для вставки:
4 9 10 2 1
Дерево:
| | 1
| 2
4
| 9
| | 10
Среднее арифметическое всех чисел в дереве: 5.2
Длина пути дерева (количество ребер): 4
Высота дерева: 3
Поиск по ключу: 2
Какой элемент удалить? 9
Дерево после удаления:
| | 1
| 2
4
| 10
```

Рисунок 7 – Тестирование при вводе значений с клавиатуры

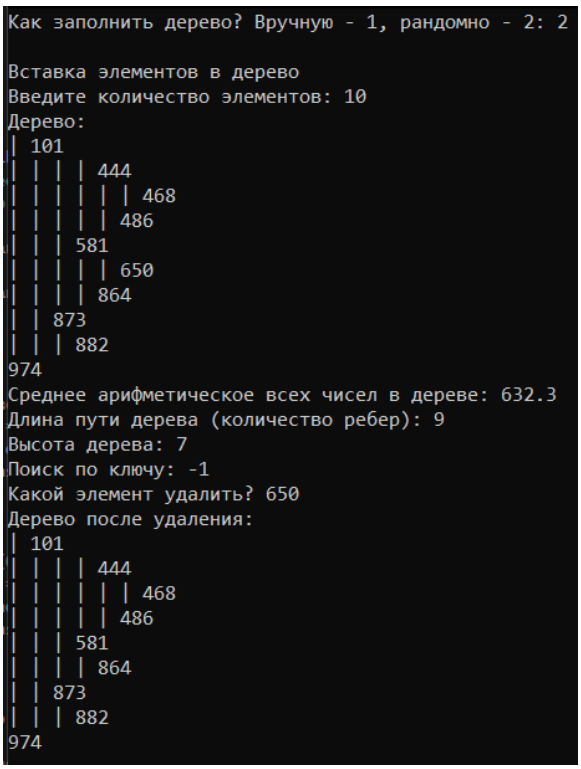


Рисунок 8 – Тестирование при заполнении дерева случайными числами

ЗАДАНИЕ 3

Разработать приложение, которое использует сбалансированное дерево поиска (СДП), и выполнить операции в соответствии с требованиями варианта.

Методы, которые должны быть реализованы в независимости от варианта:

• включение элемента в дерево;

• удаление ключа из дерева;

• поиск ключа в дереве с возвратом записи из файла;

• вывод дерева в форме дерева (с отображением структуры дерева).

Вариант – 1:

1	Красно-чёрное дерево	Строка – имя	Найти длину пути от корня до заданного значения. Найти высоту дерева, обойдя дерево симметричным обходом. Найти максимальное значение среди значений листьев дерева.

3.1 Математическая модель решения (описание алгоритма)

1. rotateRight(SplayTreeNode* node) и rotateLeft(Node*& node) – методы для вращения. Используются для поддержания свойств красно-черного дерева при вставке новых узлов.

2. fixViolation(Node*& pt) – метод для исправления нарушений свойств красно-черного дерева. Создаются указатели для хранения родителя и дедушки узла. Далее цикл, продолжающийся пока pt не является корнем, и цвета узла и его родителя — красный. Это условие указывает на нарушение свойств красно-черного дерева. Далее определяется, является ли pt_parent левым или правым потомком pt_grandparent, и применяются соответствующие действия для исправления структуры и цветов дерева, включая вращения и смену цветов.

3. insert(string name) – вставка нового узла. Вставляет новый узел в дерево и вызывает fixViolation для исправления возможных нарушений.

4. remove(const string& key) – удаление узла. Ищем узел для удаления с помощью метода search. После удаления узла проверяются свойства красно-черного дерева и в случае нарушения, исправляются (например, смена цвета, если удаляемый узел был черным).

5. search(Node* node, const string& key) – поиск узла по имени.

6. print(Node* t, int u) и display() – отображение дерева. Сначала рассматривается левое поддерево для вывода, потом корень и правое поддерево. Еще выводится цвет узлов (красный/черный).

7. inorder() – симметричный обход дерева. Он заключается в том, что сначала рассматривается левое поддерево, потом корень (или текущий узел), затем правое поддерево. Метод выводит имена всех узлов.

8. length(const string& key) – длина пути от корня до узла с заданным именем.

9. height() – высота дерева. Рассчитывается как наибольшая длина пути от корня дерева до его листа.

10. maxLeafValue() – максимальное значение среди листьев дерева. Сначала каждый узел дерева в процесса его обхода проверяем на наличие

дочерних узлов. Если их нет, то это листья. Добавляем листья в очередь, где постепенно рассматриваем каждый узел и сравниваем с другими. Максимальное значение возвращаем.

11. generateRandomName() – вспомогательная функция генерации случайных имен для заполнения дерева.

3.2 Код программы

Класс RedBlackTree содержит корень дерева (root) и методы для работы с деревом (рис. 9-15).

```
struct Node { //структура узла
    string name;
    bool color;;
    Node* left;  //указатель на левое поддерево
    Node* right; //указатель на правое поддерево
    Node* parent; //указатель на родителя
    Node(string name) : name(name), color(nullptr), left(nullptr), right(nullptr), parent(nullptr) {}
};
```

Рисунок 9 – Структура узла

```
class RedBlackTree {
private:
    Node* root; //корень дерева

    void rotateRight(Node*& node) { //поворот вправо
        Node* newParent = node->left; //указатель указывает на левый узел текущего узла
        node->left = newParent->right; //перенаправление левого ребенка на правого
        if (newParent->right != nullptr) //если правый ребненок есть
            newParent->right->parent = node; //его родитель - текущий узел
        newParent->parent = node->parent;
        if (node->parent == nullptr) root = newParent; //если node-корень, обновляем указатель на корень
        else if (node == node->parent->right) //если node-правый ребенок
            node->parent->right = newParent; //обновляем указатель родителя
        else node->parent->left = newParent; //если левый ребенок, обновляем указатель
        newParent->right = node;
        node->parent = newParent;
    }

    void rotateLeft(Node*& node) { //поворот налево
        Node* newParent = node->right; //указатель на правый дочерний узел текущего узла
        node->right = newParent->left; //далее аналогично
        if (newParent->left != nullptr) newParent->left->parent = node;
        newParent->parent = node->parent;
        if (node->parent == nullptr) root = newParent;
        else if (node == node->parent->left)
            node->parent->left = newParent;
        else node->parent->right = newParent;
        newParent->left = node;
        node->parent = newParent;
    }
```

Рисунок 10 – Методы rotateRight и rotateLeft

```
void fixViolation(Node*& pt) { //исправление нарушений свойств КЧД
    Node* pt_parent = nullptr;
    Node* pt_grandparent = nullptr;
    //текущий узел не корень и родитель красный
    while ((pt != root) && (pt->color == RED) && (pt->parent->color == RED)) {
        pt_parent = pt->parent; //родитель
        pt_grandparent = pt_parent->parent; //запоминаем деда
        if (pt_parent == pt_grandparent->left) { //родитель - левый потомок деда?
            Node* pt_uncle = pt_grandparent->right; //запоминаем дядю
            if (pt_uncle != nullptr && pt_uncle->color == RED) { //дядя красный
                pt_grandparent->color = RED; //дед становится красным
                pt_parent->color = BLACK; //родитель становится черным
                pt_uncle->color = BLACK; //дядя становится черным
                pt = pt_grandparent; //переход к деду
            }
            else {
                if (pt == pt_parent->right) { //тек узел - прый ребенок
                    rotateLeft(pt_parent); //поворот налево
                    pt = pt_parent; //обновляем текущий узел
                    pt_parent = pt->parent; //обновляем родителя
                }
                rotateRight(pt_grandparent); //поворот направо
                swap(pt_parent->color, pt_grandparent->color); //смена цвета
                pt = pt_parent; //переход к родителю
            }
        }
        else {
            Node* pt_uncle = pt_grandparent->left; //запоминаем дядю
            if ((pt_uncle != nullptr) && (pt_uncle->color == RED)) { //дядя красный
                pt_grandparent->color = RED; //дед становится красным
                pt_parent->color = BLACK; //родитель становится черным
                pt_uncle->color = BLACK; //дядя становится черным
                pt = pt_grandparent; //переход к дедушке
            }
            else {
                if (pt == pt_parent->left) { //тек узел - левый ребенок
                    rotateRight(pt_parent); //поворот направо
                    pt = pt_parent; //обновляем текущий узел
                    pt_parent = pt->parent; //обновляем родителя
                }
                rotateLeft(pt_grandparent); //поворот налево
                swap(pt_parent->color, pt_grandparent->color); //смена цвета
                pt = pt_parent; //переход к родителю
            }
        }
    }
    root->color = BLACK; //корень всегда черный
}
```

Рисунок 11 – Метод fixViolation

```
public:
    RedBlackTree() : root(nullptr) {}

    void insert(string name) { //вставка нового узла
        Node* newNode = new Node(name); //создание нового узла
        if (root == nullptr) {
            root = newNode; //дерево пустое -> новый узел - корень
            root->color = BLACK; //корень всегда черный
            return;
        }
        Node* parent = nullptr; //родитель
        Node* current = root; //начинаем с корня
        while (current != nullptr) { //поиск места для нового узла
            parent = current; //запоминаем родителя
            if (newNode->name < current->name) current = current->left; //переход к левому поддереву
            else current = current->right; //переход к правому поддереву
        }
        newNode->parent = parent; //родитель для новго узла
        if (newNode->name < parent->name)  parent->left = newNode; //устанавливаем как левое поддерево
        else parent->right = newNode; //устанавливаем как правое поддерево
        fixViolation(newNode); //исправляем возможные нарушения
    }
```

Рисунок 12 – Метод insert

```
void remove(const string& key) { //удаление
    Node* nodeToDelete = search(root, key); //ищем узел с данным ключом
    if (nodeToDelete == nullptr) { //не найден
        cout << "Ключ не найден: " << key << endl;
        return;
    }
    Node* toDelete = nodeToDelete; //указатель на узел, который будет удален
    Node* child = nullptr; //указатель для хранения дочернего узла удаляемого узла
    if (nodeToDelete->left == nullptr || nodeToDelete->right == nullptr) { //какого-то ребенка
        child = nodeToDelete->left ? nodeToDelete->left : nodeToDelete->right;
    } //присваиваем значение существующего ребенка
    else { //два ребенка
        toDelete = nodeToDelete->right; //удаляем правый узел
        //в правом поддереве ищем наименьший узел, чтобы заменить на удаляемый
        while (toDelete->left != nullptr) toDelete = toDelete->left;
        child = toDelete->left;
    }
    if (child != nullptr) { //есть дочерний узел
        child->parent = toDelete->parent; //устанавливаем его родителей todelete
    }
    if (toDelete->parent == nullptr) { //todelete-корень
        root = child; //обновляем корень
    }
    else if (toDelete == toDelete->parent->left) { //todelete-левый
        toDelete->parent->left = child;
    }
    else toDelete->parent->right = child; //todelete-правый
    if (toDelete != nodeToDelete) { //узел для удаления не является искомым узлом
        nodeToDelete->name = toDelete->name; //копируем имя
    }
    if (toDelete->color) { //исправляем дерево (если был черным todelete)
        fixViolation(child);
    }
    delete toDelete;
}

Node* search(Node* node, const string& key) {
    while (node != nullptr) {
        if (key == node->name) return node; //возвращаем указатель на узел
        else if (key < node->name) { //ключ < имени узла
            node = node->left; //переход влево
        }
        else node = node->right; //иначе вправо
    }
    return nullptr; //не найден
}
```

Рисунок 13 – Методы remove и search

```
void print(Node* t, int u) { //вывод
    if (t == nullptr) return;
    print(t->left, u + 1); //сначала левое поддерево
    string indent = "";
    for (int i = 0; i < u; ++i) indent += "|   ";
    string color = t->color ? "черный" : "красный";
    cout << indent  << t->name << " (" << color << ")" << endl;
    print(t->right, u + 1); //правое поддерево
}
void display() {
    if (root == nullptr) {
        cout << "Дерево пустое" << endl;
        return;
    }
    print(root, 0);
}

void inorder() { //симметричный обход дерева: левое поддерево -> корень (тек узел) -> правое поддерево
    Node* current = root;
    Node* stack[100]; //стек для хранения узлов
    int top = -1; //индекс стека
    while (current != nullptr || top != -1) {
        while (current != nullptr) { //переходим в самому левому узлу от текущего
            stack[++top] = current;
            current = current->left;
        }
        current = stack[top--]; //удаляем узел из стека
        cout << current->name << " "; //выводим имя узла
        current = current->right; //переход к правому поддереву
    }
    cout << endl;
}
```

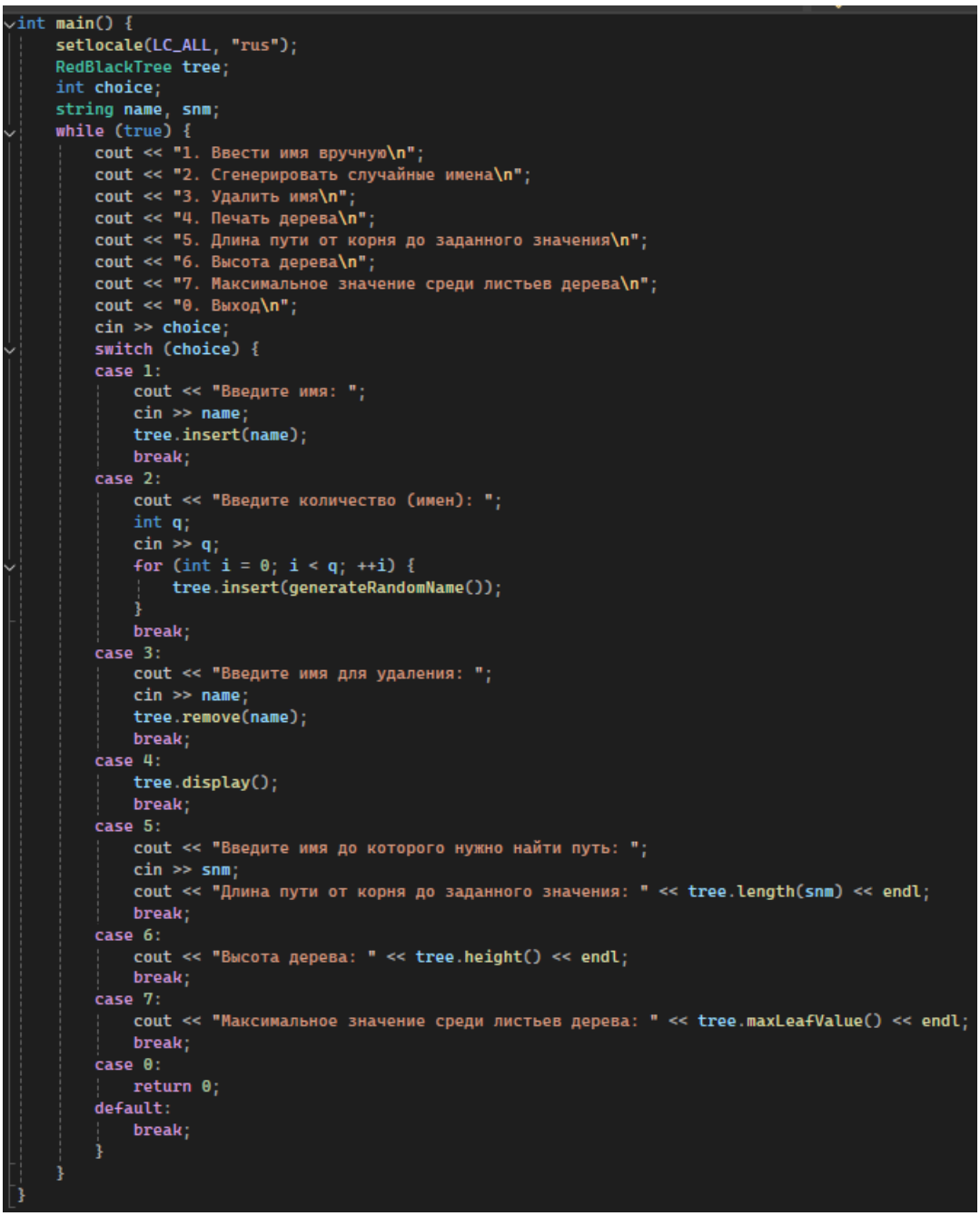
Рисунок 14 – Методы print, display и inorder

```
int length(const string& key) { //длина пути от корня до узла с заданным именем
    Node* current = root;
    int length = 0;
    while (current != nullptr) {
        if (current->name == key) {
            return length; // длина пути найдена
        }
        else if (key < current->name) {
            current = current->left;
        }
        else {
            current = current->right;
        }
        length++;
    }
    return -1; //не найдено
}
int height() { //высота дерева
    queue<pair<Node*, int>> nodeQueue; //очередь для хранения уровня
    nodeQueue.push(make_pair(root, 0)); //корень на уровне 0
    int maxHeight = -1;
    while (!nodeQueue.empty()) {
        auto nodePair = nodeQueue.front();
        nodeQueue.pop();
        Node* node = nodePair.first;
        int height = nodePair.second;
        if (node == nullptr) continue; //пропускаем пустые узлы
        maxHeight = max(maxHeight, height); //обновляем максимальную высоту
        nodeQueue.push(make_pair(node->left, height + 1)); //левое поддерево
        nodeQueue.push(make_pair(node->right, height + 1)); //правое поддерево
    }
    return maxHeight;
}
string maxLeafValue() { //макс значение среди листьев дерева
    if (root == nullptr) return ""; //дерево пустое
    queue<Node*> q; //очередь для обхода
    q.push(root);
    string maxLeaf = "";
    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        //является ли узел листом (no child)
        if (node->left == nullptr && node->right == nullptr) {
            if (node->name > maxLeaf) {
                maxLeaf = node->name; //обновляем максимальное значение
            }
        }
        else {
            if (node->left) q.push(node->left); //добавляем левое поддерево в очередь
            if (node->right) q.push(node->right); //добавляем правое поддерево в очередь
        }
    }
    return maxLeaf;
}
```

Рисунок 15 – Методы length, height и maxLeafValue

```
string generateRandomName() { //для генерации имен
    static const char alphanum[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    static const int name_length = 5;
    string name;
    for (int i = 0; i < name_length; ++i) name += alphanum[rand() % (sizeof(alphanum) - 1)];
    return name;
}
```

Рисунок 16 – Функция для генерации случайных имен



```cpp
int main() {
    setlocale(LC_ALL, "rus");
    RedBlackTree tree;
    int choice;
    string name, snm;
    while (true) {
        cout << "1. Ввести имя вручную\n";
        cout << "2. Сгенерировать случайные имена\n";
        cout << "3. Удалить имя\n";
        cout << "4. Печать дерева\n";
        cout << "5. Длина пути от корня до заданного значения\n";
        cout << "6. Высота дерева\n";
        cout << "7. Максимальное значение среди листьев дерева\n";
        cout << "0. Выход\n";
        cin >> choice;
        switch (choice) {
        case 1:
            cout << "Введите имя: ";
            cin >> name;
            tree.insert(name);
            break;
        case 2:
            cout << "Введите количество (имен): ";
            int q;
            cin >> q;
            for (int i = 0; i < q; ++i) {
                tree.insert(generateRandomName());
            }
            break;
        case 3:
            cout << "Введите имя для удаления: ";
            cin >> name;
            tree.remove(name);
            break;
        case 4:
            tree.display();
            break;
        case 5:
            cout << "Введите имя до которого нужно найти путь: ";
            cin >> snm;
            cout << "Длина пути от корня до заданного значения: " << tree.length(snm) << endl;
            break;
        case 6:
            cout << "Высота дерева: " << tree.height() << endl;
            break;
        case 7:
            cout << "Максимальное значение среди листьев дерева: " << tree.maxLeafValue() << endl;
            break;
        case 0:
            return 0;
        default:
            break;
        }
    }
}
```

Рисунок 17 – main

3.3 Тестирование программы

```
1. Ввести имя вручную
2. Сгенерировать случайные имена
3. Удалить имя
4. Печать дерева
5. Длина пути от корня до заданного значения
6. Высота дерева
7. Максимальное значение среди листьев дерева
0. Выход
Что хотите сделать?: 1
Введите количество (имен): 3
Введите имя: asd
Введите имя: glkfj
Введите имя: tewyw
1. Ввести имя вручную
2. Сгенерировать случайные имена
3. Удалить имя
4. Печать дерева
5. Длина пути от корня до заданного значения
6. Высота дерева
7. Максимальное значение среди листьев дерева
0. Выход
Что хотите сделать?: 4
|  asd (красный)
glkfj (черный)
|  tewyw (красный)
1. Ввести имя вручную
2. Сгенерировать случайные имена
3. Удалить имя
4. Печать дерева
5. Длина пути от корня до заданного значения
6. Высота дерева
7. Максимальное значение среди листьев дерева
0. Выход
Что хотите сделать?: 5
Введите имя до которого нужно найти путь: asd
Длина пути от корня до заданного значения: 1
1. Ввести имя вручную
2. Сгенерировать случайные имена
3. Удалить имя
4. Печать дерева
5. Длина пути от корня до заданного значения
6. Высота дерева
7. Максимальное значение среди листьев дерева
0. Выход
Что хотите сделать?: 6
Высота дерева: 1
```

Рисунок 18 – Тестирование при вводе вручную

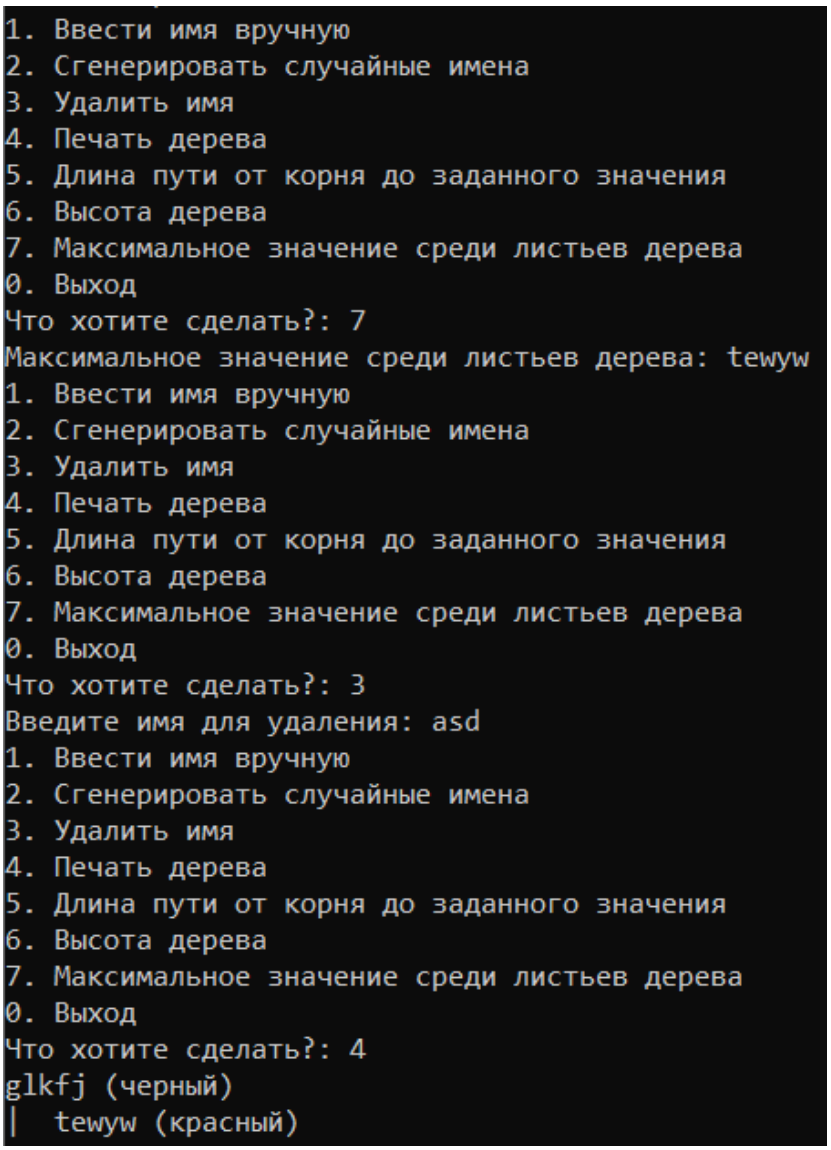


Рисунок 19 – Тестирование при вводе вручную

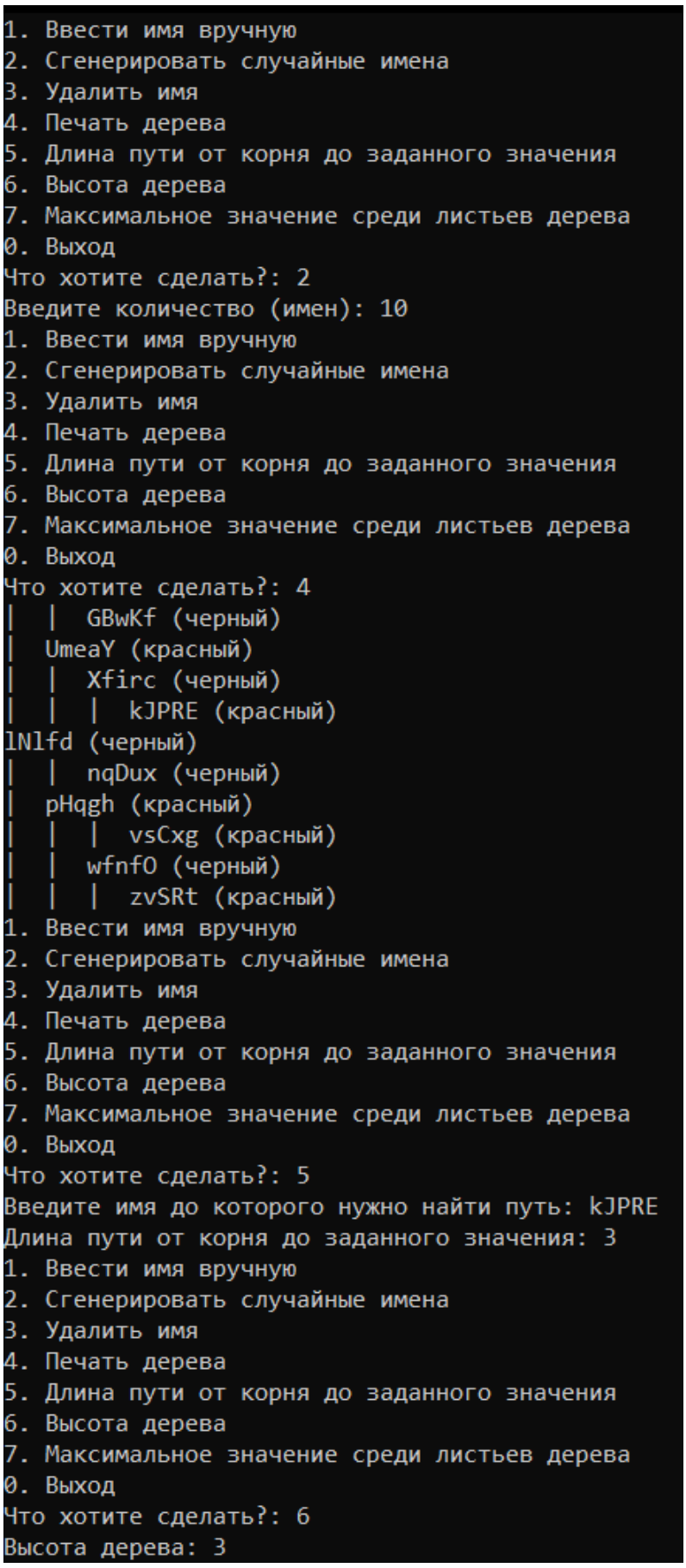


Рисунок 20 – Тестирование при заполнении дерева случайными именами

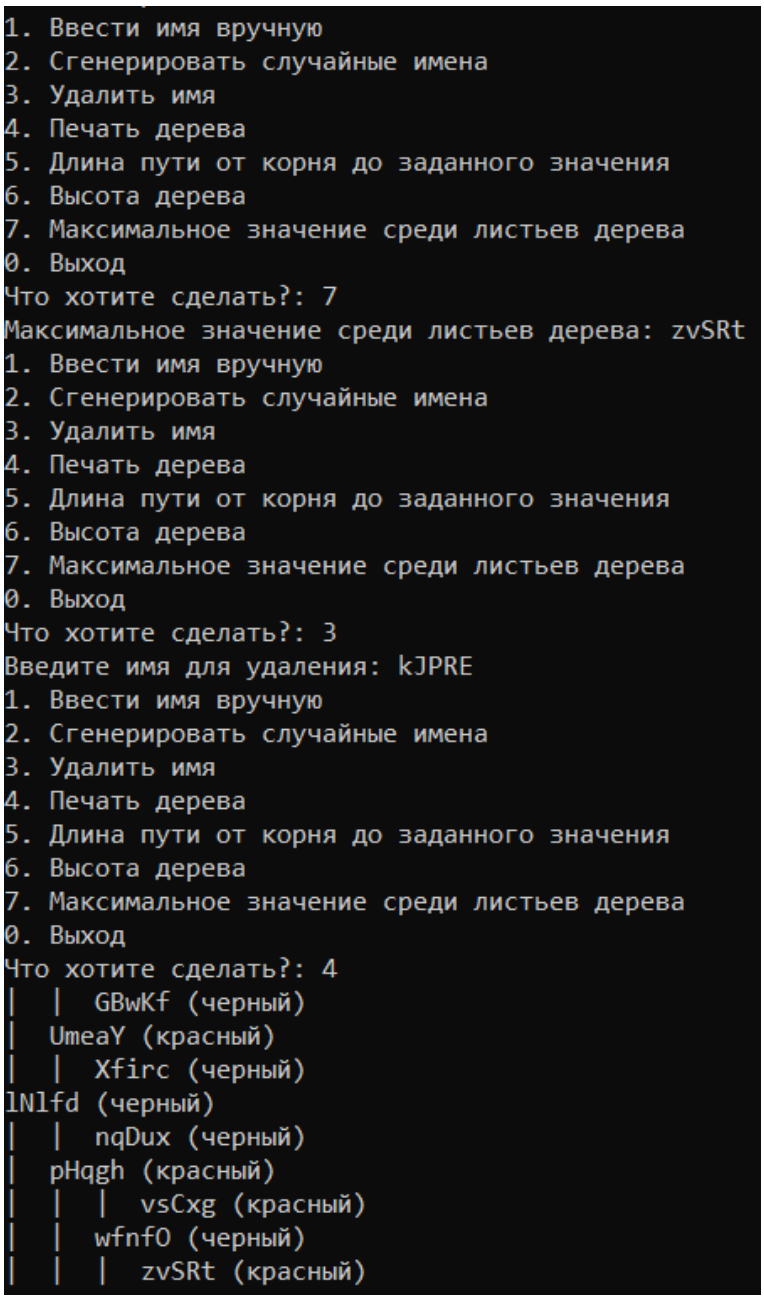


Рисунок 21 – Тестирование при заполнении дерева случайными именами

ВЫВОД

В рамках практической работы были разработаны бинарное и сбалансированное деревья поиска. Обычные бинарные деревья просты в реализации и хорошо подходят для работы с равномерно распределенными данными. В отличие от них, косые деревья оптимальны для ситуаций с неравномерным доступом, когда определенные узлы используются чаще других. Такие деревья обеспечивают амортизированную эффективность, ускоряя доступ к элементам, которые запрашивались недавно.

ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона.
2. Практические занятия Сорокин А.В. - РТУ МИРЭА 2024
3. Лекции Бузыкова Ю.С. – РТУ МИРЭА 2024
4. Практические занятия преподавателя Муравьевой Е. А., 2024.
5. Лекции Лозовский В.В. – РТУ МИРЭА 2024