МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

АНАЛИЗ ДВОИЧНОГО ДЕРЕВА

ОТЧЁТ

студента 2 курса 251 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Карасева Вадима Дмитриевича

Проверено:
доцент, к. ф.-м. н. ______________ М. И. Сафрончик

Саратов 2025

СОДЕРЖАНИЕ

1 Пример программы

```
// Структура узла двоичного дерева поиска
struct Node {
    int key;
    Node* left;
    Node* right;

    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

// Класс двоичного дерева поиска
class BinarySearchTree {
private:
    Node* root;

    // Рекурсивная функция для добавления узла в дерево
    Node* insertRec(Node* root, int key) {
        if (root == nullptr) {
            return new Node(key);
        }
        if (key < root->key) {
            root->left = insertRec(root->left, key);
        }
        else if (key > root->key) {
            root->right = insertRec(root->right, key);
        }
        return root;
    }

    // Нахождение минимального узла в поддереве
    Node* findMin(Node* node) {
        Node* current = node;
        while (current && current->left != nullptr) {
            current = current->left;
```

```
    }
    return current;
}

// Рекурсивная функция для удаления узла из дерева
Node* deleteRec(Node* root, int key) {
    if (root == nullptr) {
        return root;
    }
    if (key < root->key) {
        root->left = deleteRec(root->left, key);
    }
    else if (key > root->key) {
        root->right = deleteRec(root->right, key);
    }
    else {
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = findMin(root->right);
        root->key = temp->key;
        root->right = deleteRec(root->right, temp->key);
    }
    return root;
}

// Рекурсивная функция для поиска узла по ключу
```

```
Node* searchRec(Node* root, int key) {
    if (root == nullptr || root->key == key) {
        return root;
    }
    if (key < root->key) {
        return searchRec(root->left, key);
    }
    return searchRec(root->right, key);
}

// Рекурсивная функция для обхода дерева в порядке inorder
void inorderRec(Node* root) {
    if (root != nullptr) {
        inorderRec(root->left);
        cout << root->key << " ";
        inorderRec(root->right);
    }
}

// Рекурсивная функция для обхода дерева в порядке preorder
void preorderRec(Node* root) {
    if (root != nullptr) {
        cout << root->key << " ";
        preorderRec(root->left);
        preorderRec(root->right);
    }
}

// Рекурсивная функция для обхода дерева в порядке postorder
void postorderRec(Node* root) {
    if (root != nullptr) {
        postorderRec(root->left);
        postorderRec(root->right);
        cout << root->key << " ";
```

```
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    // Метод для добавления узла в дерево
    void insert(int key) {
        root = insertRec(root, key);
    }

    // Метод для удаления узла из дерева
    void remove(int key) {
        root = deleteRec(root, key);
    }

    // Метод для поиска узла по ключу
    Node* search(int key) {
        return searchRec(root, key);
    }

    // Метод для симметричного обхода дерева
    void inorder() {
        inorderRec(root);
        cout << endl;
    }

    // Метод для прямого обхода дерева
    void preorder() {
        preorderRec(root);
        cout << endl;
    }

    // Метод для обратного обхода дерева
```

```cpp
    void postorder() {
        postorderRec(root);
        cout << endl;
    }
};

int main() {
    setlocale(LC_ALL, "RUS");
    srand(time(NULL));

    BinarySearchTree bst;

    int n = 10;
    while (n != 0) {
        int x = rand() % 20;
        if (!bst.search(x)) {
            bst.insert(x);
            n--;
            //cout << x << " ";
        }
    }

    //cout << endl;

    cout << "Симметричный обход: ";
    bst.inorder();

    cout << "Прямой обход: ";
    bst.preorder();

    cout << "Обратный обход: ";
    bst.postorder();

    cout << "Какой ключ хотите удалить?\n";
```

```
    int a;
    cin >> a;
    bst.remove(a);
    cout << "Симметричный обход после удаления " << a << " :";
    bst.inorder();

    cout << "Какой ключ хотите найти?\n";
    cin >> a;
    Node* searchResult = bst.search(a);
    if (searchResult) {
        cout << "Узел с ключом " << a << " найден." << endl;
    }
    else {
        cout << "Узел с ключом " << a << " не найден." << endl;
    }

    return 0;
}
```

2 Анализ случаев двоичного дерева

2.1 Лучший случай

Идеально сбалансированное дерево, то есть высота одного поддерева отличается от высоты другого не более чем на 1. Высота такого дерева равна $log_2 N$, где N — количество элементов в дереве.

2.2 Худший случай

Дерево не сбалансированное и имеет только одно поддерево. Высота такого дерева будет равна N.

2.3 Средний случай

Любое другое дерево бинарного поиска. Его высота также составит $log_2 N$.

3 Поиск в двоичном дереве

3.1 Лучший случай

Проходим через узлы один за другим. Если мы найдем элемент на втором уровне, то для этого мы сделаем 2 сравнения, если на третьем — 3 сравнения и так далее. Таким образом, на поиск ключа в дереве бинарного поиска мы затратим время, равное высоте дерева, то есть $log_2 N$, поэтому временная сложность поиска в лучшем случае составит O(logN).

3.2 Худший случай

Нужно пройти от корня до самого глубокого узла, являющегося листом, и в этом случае высота дерева становится равной N, где N — количество элементов в дереве, и затрачиваемое время совпадает с высотой дерева. Поэтому временная сложность в худшем случае составит O(N).

3.3 Средний случай

Пусть S(N) — среднее значение общей длины внутреннего пути. Докажем, что временная сложность в этом случае составит O(logN).

Очевидно, что для дерева с одним узлом S(1) = 0. Любое бинарное дерево с N узлами содержит i элементов в левом поддереве, $0 \leq$ i $\leq$ N – 1, а в правом поддереве n - i - 1. Для фиксированного i получим: S(N) = (n - 1) + S(i) + S(n - i - 1), где (n - 1) - сумма дополнительных шагов к каждому узлу, учитывая увеличение глубины всех узлов на 1; S(i) – это суммарный внутренний путь в левом поддереве; S(n - i - 1) – это суммарный внутренний путь в правом поддереве. После суммирования этих повторений для $0 \leq$ i $\leq$ N – 1 получим: $S(n) = n(n-1) + 2\sum\limits_{i=1}^{n-1} S(i)$.

Следовательно, S(N) $\in$ O(NlogN), и глубина узла S(N) $\in$ O(logN).

4 Вставка в двоичном дереве

4.1 Лучший случай

В идеально сбалансированном дереве нам так же будет необходимо сделать лишь максимум $log_2 N$ сравнений для поиска подходящего места для вставляемого узла, следовательно, временная сложность вставки в лучшем случае составит O(logN).

4.2 Худший случай

Нужно пройти от корня до последнего или самого глубокого листового узла, а максимальное количество шагов равно N (высота дерева). Таким образом, временная сложность составит O(N), так как поиск каждого узла один за другим до последнего листового узла займёт время O(N), а затем мы вставим элемент, что занимает константное время.

4.3 Средний случай

Проведя рассуждения, аналогичные тем, что были рассмотрены в среднем случае поиска элемента, получаем, что сложность операции вставки в среднем случае составит O(logN).

5 Удаление в двоичном дереве

5.1 Лучший случай

Снова максимальным значением количества проходов (сравнений) по дереву будет $log_2 N$ — высота дерева. Копирование содержимого и его удаление требуют константного времени. Поэтому общая временная сложность составит O(logN).

5.2 Худший случай

Процесс удаления займёт O(N) времени, так как максимальное количество проходов (сравнений) по дереву равно N .

5.3 Средний случай

Проведя рассуждения, аналогичные тем, что были рассмотрены в среднем случае поиска элемента, получаем, что сложность операции удаления в среднем случае составит O(logN).

6 Обходы дерева

Дерево бинарного поиска имеет 3 основных обхода: прямой, обратный и симметричный. Их разница заключается в том, в каком порядке мы обращаемся к элементам. Каждый из них будет иметь временную сложность O(N), так как процедура вызывается ровно два раза для каждого узла дерева.

7 Расход памяти

В двоичном дереве поиска каждый узел содержит значение и указатели на левого и правого потомков. Расход памяти в двоичном дереве поиска зависит от количества узлов и размера каждого узла. Предположим, что каждый узел имеет фиксированный размер, состоящий из:

— Значения элемента (например, целочисленное значение).
— Указателя на левого потомка (обычно 4 байта на 32-битной системе или 8 байт на 64-битной системе).
— Указателя на правого потомка (также 4 или 8 байт).

Следовательно, общий размер каждого узла составляет от 12 до 24 байтов в зависимости от архитектуры.

Память, затраченная на двоичное дерево поиска, зависит от количества узлов и их размера. Пусть n - количество узлов в дереве.

Тогда общий расход памяти для дерева будет составлять O(n), так как каждый узел требует фиксированного количества памяти и количество узлов напрямую пропорционально объему занимаемой памяти.