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

Кафедра информатики и программирования

АНАЛИЗ ДВОИЧНОГО ДЕРЕВА ПОИСКА

ОТЧЕТ

студентки 2 курса 211 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета компьютерных наук и информационных технологий
Никитенко Яны Валерьевны

Саратов 2025

СОДЕРЖАНИЕ

1 Текст программы

```
//
struct node {
 int key;
 node* left;
 node* right;
 node(int k) : key(k), left(nullptr), right(nullptr) {}
};

//

HANDLE outp = GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO csbInfo;
node* root = nullptr;

void max_height(node* x, short& max, short deepness = 1)
{ // требует проверки на существование корня
 if (deepness > max) max = deepness;
 if (x->left) max_height(x->left, max, deepness + 1);
 if (x->right) max_height(x->right, max, deepness + 1);
}
bool isSizeOfConsoleCorrect(const short& width, const short& height) {
 GetConsoleScreenBufferInfo(outp, &csbInfo);
 COORD szOfConsole = csbInfo.dwSize;
 if (szOfConsole.X < width && szOfConsole.Y < height)
    cout << "Please increase the height and width of the terminal. ";
 else if (szOfConsole.X < width) cout << "Please increase the width of the terminal. ";
 else if (szOfConsole.Y < height) cout << "Please increase the height of the terminal. ";
 if (szOfConsole.X < width || szOfConsole.Y < height) {
 cout << "Size of your terminal now: " << szOfConsole.X << ' ' << szOfConsole.Y
   << ". Minimum required: " << width << ' ' << height << ".\n";
  return false;
 }
```

```
  return true;
}

void print_helper(node* x, const COORD pos, const short offset) {
 SetConsoleCursorPosition(outp, pos);
 cout << setw(offset + 1) << x->key;
 if (x->left) print_helper(x->left, { pos.X, short(pos.Y + 1) }, offset >> 1);
 if (x->right) print_helper(x->right, { short(pos.X + offset), short(pos.Y + 1) }, offset >>
}

void print() {
 if (root) {
  short max = 1;
  max_height(root, max);
  short width = 1 << max + 1, max_w = 128; // вычисляем ширину вывода
  if (width > max_w) width = max_w;
  while (!isSizeOfConsoleCorrect(width, max)) system("pause");
 for (short i = 0; i < max; ++i) cout << '\n'; // резервируем место для вывода
  GetConsoleScreenBufferInfo(outp, &csbInfo); // получаем данные
  COORD endPos = csbInfo.dwCursorPosition;
  print_helper(root, { 0, short(endPos.Y - max) }, width >> 1);
  SetConsoleCursorPosition(outp, endPos);
  SetConsoleTextAttribute(outp, 7);
 }
}



// Добавить узел
void insert(node*& root, int key) {
    if (!root) {
        root = new node(key);
    }
    else if (key < root->key) {
```

```
        insert(root->left, key);
    }
    else {
        insert(root->right, key);
    }
}
//

// Найти
node* findMin(node* root) {
    while (root && root->left) {
        root = root->left;
    }
    return root;
}
//

// Удалить узел
node* deleteNode(node* root, int key) {
    if (!root) return root;
    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    }
    else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    }
    else {
        if (!root->left) {
            node* temp = root->right;
            delete root;
            return temp;
        }
        else if (!root->right) {
            node* temp = root->left;
```

```
            delete root;
            return temp;
        }
        node* temp = findMin(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
//

// Поиск
node* search(node* root, int key) {
    if (!root || root->key == key) return root;
    if (key < root->key) return search(root->left, key);
    return search(root->right, key);
}
//

// Обход в прямом порядке
void inorder(node* root) {
    if (root) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);

    }
}
//

// Обход в симметричном порядке
void preorder(node* root) {
    if (root) {
        cout << root->key << " ";
```

```
        preorder(root->left);
        preorder(root->right);
    }
}
//

// Обход в обратном порядке
void postorder(node* root) {
    if (root) {
        postorder(root->left);
        postorder(root->right);
        cout << root->key << " ";
    }
}
//

// Меню
void menu() {
    cout << "1. Добавить узел\n";
    cout << "2. Удалить узел\n";
    cout << "3. Поиск узла\n";
    cout << "4. Обход в прямом порядке\n";
    cout << "5. Обход в симметричном порядке\n";
    cout << "6. Обход в обратном порядке\n";
    cout << "7. Вывести дерево\n";
    cout << "0. Выход\n";
}
//

int main() {
    setlocale(LC_ALL, "Russian");
    int choice, key;
    while (true) {
        menu();
```

```
cout << "Выберите действие: ";
cin >> choice;

switch (choice) {
case 1:
    while (true) {
        cout << "-1 - выход из цикла" << endl;
        cin >> key;
        if (key == -1) break; // Выход из цикла добавления
        insert(root, key);
    }
    break;

case 2:
    cout << "ключ для удаления: ";
    cin >> key;
    root = deleteNode(root, key);
    break;
case 3:
    cout << "ключ для поиска: ";
    cin >> key;
    if (search(root, key)) {
        cout << "узел с ключом " << key << " найден.\n";
    }
    else {
        cout << "узел с ключом " << key << " не найден.\n";
    }
    break;
case 4:
    cout << "обход в прямом порядке: ";
    preorder(root);
    cout << endl;
    break;
case 5:
```

```
            cout << "обход в симметричном порядке: ";
            inorder(root);
            cout << endl;
            break;
        case 6:
            cout << "обход в обратном порядке: ";
            postorder(root);
            cout << endl;
            break;
        case 7:
            print();
            break;
        case 0:
            return 0;
        default:
            cout << "\n";
        }
    }
    return 0;
}
```

2 Анализ случаев

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.

Худший случай

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

Средний случай

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

3 Вставка в двоичном дереве

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).

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.

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).

4 Удаление в двоичном дереве

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).

Худший случай

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

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).

5 Поиск в двоичном дереве

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).

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).

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:

$S(n) = n(n-1) + \sum_{i=0}^{n-1} S(i)$

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

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.