

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

Кафедра информатики и программирования

**АНАЛИЗ АВЛ ДЕРЕВА**

**ОТЧЕТ**

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

## СОДЕРЖАНИЕ

1	Текст программы .....	3
2	Вставка в АВЛ дереве .....	13
3	Удаление в АВЛ дереве .....	14
4	Поиск в АВЛ дереве .....	15
5	Обход в АВЛ дереве .....	16
6	Расход памяти .....	17

## 1 Текст программы

// Структура дерева

```
struct node {  
    int key;  
    node* left;  
    node* right;  
    int height;  
    node(int k) : key(k), left(nullptr), right(nullptr), height(1) {}  
};  
//
```

//

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

// Высота

```
int height(node* n) {  
    return n ? n->height : 0;  
}  
//
```

// Вычисляет балансирующий фактор, нужно чтоб поддерживать дерево (и меня)

```
int balanceFactor(node* n) {  
    return n ? height(n->left) - height(n->right) : 0;  
}
```

// Обновляет высоту дерева, чтобы поддерживать высоту и корректность работы др

```
void updateHeight(node* n) {  
    if (n) {  
        n->height = 1 + max(height(n->left), height(n->right));  
    }  
}
```

```

}
//

// Вращение дерева, для балансирования дерева. Правое вращение балансирует, когда
// и дерево становится не сбалансированным
node* rotateRight(node* y) {
    node* x = y->left;
    node* T2 = x->right;

    x->right = y;
    y->left = T2;

    updateHeight(y);
    updateHeight(x);

    return x;
}
//

// Вращение дерева, для балансирования дерева. Левое вращение балансирует, когда
// и дерево становится не сбалансированным
node* rotateLeft(node* x) {
    node* y = x->right;
    node* T2 = y->left;

    y->left = x;
    x->right = T2;

    updateHeight(x);
    updateHeight(y);

    return y;
}

```

```
//
```

```
// Проверяет балансирующий фактор узла и выполняет необходимые вращения для
```

```
node* balance(node* n) {
```

```
    if (!n) return n;
```

```
    updateHeight(n);
```

```
    int bf = balanceFactor(n);
```

```
    if (bf > 1 && balanceFactor(n->left) >= 0)
```

```
        return rotateRight(n);
```

```
    if (bf > 1 && balanceFactor(n->left) < 0) {
```

```
        n->left = rotateLeft(n->left);
```

```
        return rotateRight(n);
```

```
    }
```

```
    if (bf < -1 && balanceFactor(n->right) <= 0)
```

```
        return rotateLeft(n);
```

```
    if (bf < -1 && balanceFactor(n->right) > 0) {
```

```
        n->right = rotateRight(n->right);
```

```
        return rotateLeft(n);
```

```
    }
```

```
    return n;
```

```
}
```

```
//
```

```
//
```

```

void max_height(node* x, short& max, short deepness = 1) {
    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 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 >> 1);
}

//

//

void print() {

```

```

if (root == NULL)
{
    cout << "Пусто\n";
}
else
{
    short max = 1;
    max_height(root, max);
    short width = 1 << max + 1, max_w = 128;
    if (width > max_w) width = max_w;
    while (!isSizeOfConsoleCorrect(width, max)) system("pause");
    for (short i = 0; i < max; ++i) cout << ' \n';
    GetConsoleScreenBufferInfo(outp, &csbInfo);
    COORD endPos = csbInfo.dwCursorPosition;
    print_helper(root, { 0, short(endPos.Y - max) }, width >> 1);
    SetConsoleCursorPosition(outp, endPos);
}
}
//

// Добавить узел
node* insert(node* root, int key) {
    if (!root) return new node(key);

    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    else
        return root; // Дубликаты не допускаются

    return balance(root);
}
//

```

```

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

// Удаление узла
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 || !root->right) {
            node* temp = root->left ? root->left : root->right;
            if (!temp) {
                temp = root;
                root = nullptr;
            }
            else {
                *root = *temp;
            }
            delete temp;
        }
        else {
            node* temp = findMin(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }
}

```



```

    }
}

if (!root) return root;

return balance(root);
}
//

// Поиск узла
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 симметричном порядке
void inorder(node* root) {
    if (root) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}
//

// Обход в прямом порядке
void preorder(node* root) {

    if (root == nullptr) {
        return;
    }
    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 узлы\n";
        cout << "2. Удалить узел\n";
        cout << "3. Вывести дерево\n";
        cout << "4. Поиск узла\n";
        cout << "5. Обход в прямом порядке\n";
        cout << "6. Обход в симметричном порядке\n";
        cout << "7. Обход в обратном порядке\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:
    cout << "-1 - ВЫХОД" << endl;
    while (true) {
        cin >> key;
        if (key == -1) break;
        root = insert(root, key);
        cout << key << " добавлен.\n";
    }
    break;

case 2:
    cout << "ключ для удаления: ";
    cin >> key;
    root = deleteNode(root, key);
    cout << key << " удалён.\n";
    break;

case 3:
    print();
    cout << endl;
    break;

case 4:
    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 5:

```

```

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

```

## **2 Вставка в AVL дереве**

В AVL-дереве высота гарантированно составляет  $O(\log n)$ , где  $n$  - количество элементов в дереве. Операция вставки включает:

- Поиск места для вставки:  $O(\log n)$
- Вставка нового узла:  $O(1)$
- Балансировка (до 2 поворотов):  $O(1)$

**Временная сложность одной операции вставки:  $O(\log n)$**

Для выполнения  $n$  операций вставки сложность составляет  $O(n \log n)$ .

### 3 Удаление в AVL дереве

Операция удаления в AVL-дереве сохраняет балансировку и включает: -

Поиск удаляемого узла:  $O(\log n)$

- Поиск минимального в правом поддереве (при необходимости):  $O(\log n)$

- Балансировка (до 2 поворотов):  $O(1)$

**Временная сложность одной операции удаления:  $O(\log n)$**

Для  $n$  операций удаления сложность составляет  $O(n \log n)$ .

#### **4 Поиск в AVL-дереве**

Поиск в AVL-дереве выполняется аналогично бинарному дереву поиска, но благодаря балансировке гарантируется логарифмическая высота:

- Худший случай:  $O(\log n)$
- Средний случай:  $O(\log n)$
- Лучший случай:  $O(1)$  - если искомый элемент в корне

Поиск выполняется рекурсивным спуском от корня к листьям, на каждом уровне отбрасывая половину оставшихся узлов

## **5 Обход в AVL-дереве**

Префиксный (preorder), постфиксный (postorder) и инфиксный (inorder) обходы занимают  $O(n)$  времени, так как каждый узел дерева должен быть посещен ровно один раз. Таким образом, общая временная сложность операций вставки, удаления, поиска и обходов в AVL-дереве составляет  $O(n \log n)$  для  $n$  элементов.



## 6 Расход памяти

### 1. Узел дерева

Для каждого узла дерева выделяется фиксированное количество памяти, состоящее из:

inf:  $O(1)$  памяти

left и right: указатели на другие узлы дерева, каждый из которых занимает  $O(1)$  памяти

height: переменная типа unsignedchar, занимающая  $O(1)$  памяти

Таким образом, общее количество памяти, выделенное под каждый узел дерева, составляет  $O(1)$ .

### 2. Локальные переменные и дополнительные данные

Локальные переменные, используемые в функциях, также требуют ограниченное количество памяти, которое не зависит от размера входных данных или высоты дерева. Поэтому расход памяти на локальные переменные можно считать  $O(1)$ .

### 3. Входные данные

Расход памяти на входные данные (значения, которые вставляются в дерево) также не зависит от размера дерева и может быть считан  $O(n)$ , где  $n$  – количество элементов.

### 4. Стек вызовов

Во время выполнения рекурсивных операций используется стек вызовов, чтобы хранить информацию о текущем состоянии выполнения каждой рекурсивной функции. Глубина стека вызовов зависит от высоты дерева и количества рекурсивных вызовов, что может быть  $O(\log n)$  в худшем случае для операций вставки, удаления и поиска. Таким образом, общий асимптотический анализ сложности расхода памяти составляет  $O(n + \log n)$ , где  $n$  - количество элементов в дереве.

Вращение происходит за время  $O(1)$ . Временная сложность всех функций равна  $O(\log N)$ , потому что дерево всегда сбалансированное.