Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

**ТЕСТИРОВАНИЕ**

По предмету “Основы алгоритмизации и программирования”

На тему: “Бинарное дерево поиска”

Выполнил:

Студент 1 курса 9 группы

Павлович Ян Андреевич

Преподаватель: Белодед Н.И.

2024, Минск

## Содержание

1. Введение
2. Исходный код с добавлением комментариев
3. О работе кода
4. Тестирование методов
   1. BuildTree
   2. Vyvod
   3. ObhodLeft
   4. ObhodEnd
   5. ObhodBack
   6. CleanTree
5. Добавим поиск максимального и минимальных значений
6. Добавим преобразование дерева в массив
7. Добавим удаление узла по значению
8. Заключение

## Введение

Двоичное дерево поиска (БДП) является важной структурой данных в информатике, используемой для хранения и организации данных таким образом, чтобы обеспечить эффективный поиск, вставку и удаление элементов. БДП состоит из узлов, каждый из которых имеет ключевое значение, иерархически организованных так, чтобы все узлы левого поддерева имели ключи, меньшие чем ключ текущего узла, а все узлы правого поддерева имели ключи, большие чем ключ текущего узла. Это обеспечивает логичное и быстрое управление данными.

В данной работе представлен код, который реализует структуру данных в виде двоичного дерева поиска, а также включает в себя методы для управления этой структурой, такие как вставка элементов, обход дерева разными способами, определение его высоты и очистка. Особое внимание уделяется методу `CleanTree`, который отвечает за рекурсивную очистку дерева, освобождая память, выделенную для всех узлов дерева.

## Исходный код с добавлением комментариев

#include <iostream> // Подключение библиотеки ввода/вывода.

using namespace std; // Использование пространства имен std для упрощения кода.

struct node // Объявление структуры узла дерева.

{

    int Key; // Ключ вершины дерева.

    int Count; // Счетчик количества элементов с данным ключом.

    node\* Left; // Указатель на левое поддерево.

    node\* Right; // Указатель на правое поддерево.

};

class TREE // Объявление класса дерева.

{

private:

    node\* Tree; // Указатель на корень дерева.

    void Search(int, node\*\*); // Объявление функции поиска вершины с ключом int в дереве со вставкой.

public:

    TREE() { Tree = NULL; } // Конструктор класса, инициализирует дерево пустым.

    node\*\* GetTree() { return &Tree; } // Функция возвращает указатель на корень дерева.

    void BuildTree(); // Объявление функции для построения дерева.

    void CleanTree(node\*\*); // Объявление функции для очистки дерева.

    void ObhodEnd(node\*\*); // Объявление функции для концевого обхода дерева.

    void ObhodLeft(node\*\*); // Объявление функции для левостороннего обхода дерева.

    void ObhodBack(node\*\*); // Объявление функции для обратного обхода дерева.

    void Vyvod(node\*\*, int); // Объявление функции для вывода дерева на экран дисплея.

    int Height(node\*\*); // Объявление функции для определения высоты бинарного дерева.

};

void main()

// Главная функция программы, точка входа в программу.

{

    setlocale(LC\_ALL, "Rus"); // Установка локализации для вывода на русском языке.

    TREE A; // Создание объекта дерева типа TREE.

    A.BuildTree(); // Вызов функции для построения дерева.

    cout << "\nВывод дерева:\n"; // Вывод начального сообщения перед выводом дерева.

    A.Vyvod(A.GetTree(), 0); // Вызов функции для вывода дерева на экран.

    cout << "\nВысота дерева: " << A.Height(A.GetTree()) << endl; // Вывод высоты дерева.

    cout << "\nЛевосторонний обход дерева: "; // Вывод сообщения перед началом левостороннего обхода дерева.

    A.ObhodLeft(A.GetTree()); // Вызов функции для левостороннего обхода дерева.

    cout << "\nКонцевой обход дерева: "; // Вывод сообщения перед началом концевого обхода дерева.

    A.ObhodEnd(A.GetTree()); // Вызов функции для концевого обхода дерева.

    cout << "\nОбратный обход дерева: "; // Вывод сообщения перед началом обратного обхода дерева.

    A.ObhodBack(A.GetTree()); // Вызов функции для обратного обхода дерева.

    A.CleanTree(A.GetTree()); // Вызов функции для очистки дерева.

    cout << "\n"; // Вывод пустой строки.

    system("PAUSE"); // Ожидание нажатия клавиши для завершения программы.

}

void TREE::BuildTree()

// Построение дерева (рекурсивный алгоритм).

{

    int el; // Объявление переменной для хранения ключа элемента.

    cout << "Вводите ключи вершин дерева ...\n"; // Вывод сообщения о вводе ключей вершин.

    cin >> el; // Чтение ключа вершины дерева.

    while (el != 0) // Цикл выполняется, пока не введен 0.

    {

        Search(el, &Tree); // Вызов функции для поиска и вставки элемента в дерево.

        cin >> el; // Чтение следующего ключа.

    }

}

void TREE::Search(int x, node\*\* p)

// Поиск вершины с ключом x в дереве с вставкой (рекурсивный алгоритм).

{

    if (\*p == NULL) // Если указатель равен NULL, значит, вершина не найдена.

    {

        // Создание новой вершины дерева.

        \*p = new(node);

        // Инициализация нового узла.

        (\*\*p).Key = x; // Установка ключа узла.

        (\*\*p).Count = 1; // Установка счетчика на 1.

        (\*\*p).Left = NULL; // Установка левого поддерева на NULL.

        (\*\*p).Right = NULL; // Установка правого поддерева на NULL.

    }

    else if (x < (\*\*p).Key) // Если x меньше ключа текущей вершины, поиск продолжается в левом поддереве.

    {

        Search(x, &((\*\*p).Left)); // Рекурсивный вызов функции поиска в левом поддереве.

    }

    else if (x > (\*\*p).Key) // Если x больше ключа текущей вершины, поиск продолжается в правом поддереве.

    {

        Search(x, &((\*\*p).Right)); // Рекурсивный вызов функции поиска в правом поддереве.

    }

    else // Если x равен ключу текущей вершины, элемент уже существует.

    {

        (\*\*p).Count = (\*\*p).Count + 1; // Увеличение счетчика.

    }

}

void TREE::ObhodLeft(node\*\* w)

// Левосторонний обход дерева (рекурсивный алгоритм).

{

    if (\*w != NULL) // Если текущий узел не равен NULL.

    {

        cout << (\*\*w).Key << " "; // Вывод ключа текущего узла.

        ObhodLeft(&((\*\*w).Left)); // Рекурсивный вызов функции для обхода левого поддерева.

        ObhodLeft(&((\*\*w).Right)); // Рекурсивный вызов функции для обхода правого поддерева.

    }

}

void TREE::ObhodEnd(node\*\* w)

// Концевой обход дерева (рекурсивный алгоритм).

{

    if (\*w != NULL) // Если текущий узел не равен NULL.

    {

        ObhodEnd(&((\*\*w).Left)); // Рекурсивный вызов функции для обхода левого поддерева.

        ObhodEnd(&((\*\*w).Right)); // Рекурсивный вызов функции для обхода правого поддерева.

        cout << (\*\*w).Key << " "; // Вывод ключа текущего узла после обхода левого и правого поддеревьев.

    }

}

void TREE::ObhodBack(node\*\* w)

// Обратный обход дерева (рекурсивный алгоритм).

{

    if (\*w != NULL) // Если текущий узел не равен NULL.

    {

        ObhodBack(&((\*\*w).Left)); // Рекурсивный вызов функции для обхода левого поддерева.

        cout << (\*\*w).Key << " "; // Вывод ключа текущего узла.

        ObhodBack(&((\*\*w).Right)); // Рекурсивный вызов функции для обхода правого поддерева.

    }

}

void TREE::CleanTree(node\*\* w)

// Очистка дерева (рекурсивный алгоритм).

{

    if (\*w != NULL) // Если текущий узел не равен NULL.

    {

        // Очистка левого поддерева.

        CleanTree(&((\*\*w).Left));

        // Очистка правого поддерева.

        CleanTree(&((\*\*w).Right));

        // Удаление текущего узла.

        delete\* w;

    }

}

void TREE::Vyvod(node\*\* w, int l)

// Вывод дерева на экран дисплея (рекурсивный алгоритм).

{

    int i; // Объявление переменной для управления отступом.

    if (\*w != NULL) // Если текущий узел не равен NULL.

    {

        Vyvod(&((\*\*w).Right), l + 1); // Вызов функции для вывода правого поддерева с увеличением уровня отступа.

        // Вывод отступа для текущего уровня.

        for (i = 1; i <= l; i++) cout << "   ";

        // Вывод ключа текущего узла.

        cout << (\*\*w).Key << endl;

        Vyvod(&((\*\*w).Left), l + 1); // Вызов функции для вывода левого поддерева с увеличением уровня отступа.

    }

}

int TREE::Height(node\*\* w)

// Метод для определения высоты дерева.

{

    int h1, h2; // Переменные для высоты левого и правого поддеревьев.

    if (\*w == NULL) return -1; // Если дерево пустое, возвращаем -1.

    else

    {

        // Рекурсивный вызов для определения высоты левого поддерева.

        h1 = Height(&((\*\*w).Left));

        // Рекурсивный вызов для определения высоты правого поддерева.

        h2 = Height(&((\*\*w).Right));

        // Возвращаем большую высоту из двух поддеревьев + 1 для текущей вершины.

        if (h1 > h2)

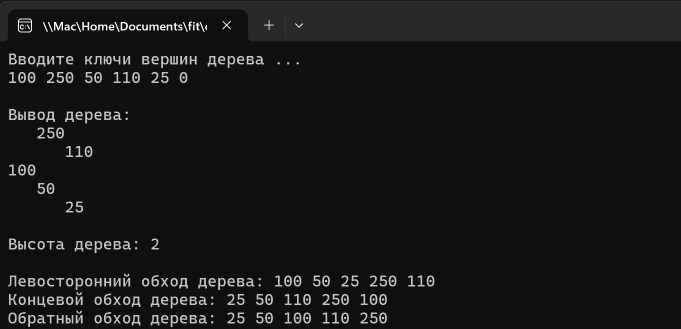
            return (1 + h1);

        else

            return (1 + h2);

    }

}

Результат работы программы:

## О работе кода

Код реализует структуру данных в виде двоичного дерева поиска (БДП) и включает в себя методы для управления этой структурой. Двоичное дерево поиска — это структура данных, в которой каждая вершина имеет максимум двух дочерних узлов, и ключи левого поддерева всегда меньше, чем ключи правого поддерева.

Работа кода:

1. Структура данных: Определяет структуру данных `node` для узла дерева, содержащего ключ `Key`, счетчик `Count` для подсчета количества повторений ключа, а также указатели `Left` и `Right` на левого и правого дочерних узлов.

2. Класс `TREE`: Определяет класс `TREE` для представления двоичного дерева поиска с указателем на корень дерева (`Tree`). В классе есть методы для следующих операций:

Построение дерева (`BuildTree`): Позволяет пользователю вводить ключи и добавлять их в дерево, пока не будет введен ключ 0.

Поиск и вставка (`Search`): Метод, который ищет ключ в дереве и, если ключ не найден, добавляет его в дерево.

Левосторонний обход (`ObhodLeft`): Выводит узлы дерева в порядке левостороннего обхода (корень, левое поддерево, правое поддерево).

Концевой обход (`ObhodEnd`): Выводит узлы дерева в порядке концевого обхода (левое поддерево, правое поддерево, корень).

Обратный обход (`ObhodBack`): Выводит узлы дерева в порядке обратного обхода (левое поддерево, корень, правое поддерево).

Очистка дерева (`CleanTree`): Освобождает память, выделенную для узлов дерева, рекурсивно очищая все поддеревья.

Вывод дерева (`Vyvod`): Выводит дерево на экран в структурированном виде, показывая расположение узлов на разных уровнях.

Высота дерева (`Height`): Определяет высоту дерева, рассчитывая максимальную глубину дерева от корня до листьев.

3. Главная функция (`main`): Создает объект дерева `TREE` и вызывает методы для построения, вывода и обхода дерева, определения его высоты и очистки. В конце выполняется ожидание нажатия клавиши для завершения программы.

## BuildTree

Метод `BuildTree()` в классе `TREE` отвечает за построение бинарного дерева поиска из входных ключей, вводимых пользователем. Процесс работы метода можно описать следующим образом:

1. Запуск метода: Метод `BuildTree()` начинается с вывода приглашения к пользователю о вводе ключей вершин дерева и обеспечивает возможность пользователю вводить данные.

2. Ввод ключевых значений: Пользователь вводит ключевые значения узлов дерева. Программа читает эти значения из ввода, начиная с первого ключа.

3. Проверка ввода: Если введенное значение равно `0`, метод завершает процесс построения дерева. Это означает, что пользователь закончил ввод данных.

4. Поиск или вставка ключа: Если введенное значение не равно `0`, программа вызывает метод `Search()` для поиска или вставки введенного ключа в дерево.

- Если дерево пустое, создается новый узел с введенным ключом, который становится корневым узлом дерева.

- Если дерево не пустое, программа рекурсивно ищет подходящее место для введенного ключа в дереве.

- Если ключ уже существует в дереве, счетчик `Count` у соответствующего узла увеличивается.

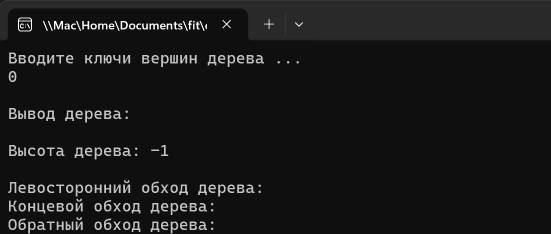
5. Продолжение ввода: После обработки каждого введенного ключа программа продолжает считывать следующие ключевые значения из ввода и повторяет шаги 3 и 4.

6. Завершение ввода: Когда пользователь вводит `0`, метод завершает ввод ключей, и процесс построения дерева заканчивается.

Метод `BuildTree()` создает дерево, используя рекурсивный подход, и размещает введенные ключи в бинарном дереве поиска в соответствии с их значениями. Дерево становится структурированным и может быть использовано для дальнейших операций, таких как обходы, вывод или поиск ключей.

### Проверка ввода пустого дерева

В текущем коде при вводе 0 программа не завершает свою работу, а продолжает выполнение дальнейших операций, таких как обход дерева, определение его высоты и очистка. Это может приводить к некорректному поведению программы.



Чтобы избежать таких ситуаций, добавим проверку на ввод первого 0 в главный метод (`main()`). Если первый ввод равен 0, программа завершит свою работу, а не будет продолжать выполнение оставшихся операций. Это позволит избежать некорректного поведения программы на пустом дереве и улучшит ее надежность.

### Добавим следующие строки в метод BuildTree:

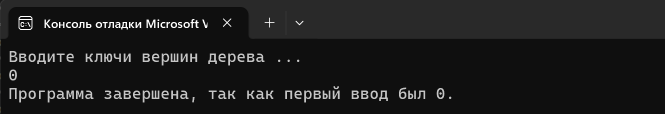
// Если первый ввод равен 0, сразу завершаем программу

if (el == 0) {

    cout << "Программа завершена, так как первый ввод был 0." << endl;

    exit(0); // Завершаем выполнение программы

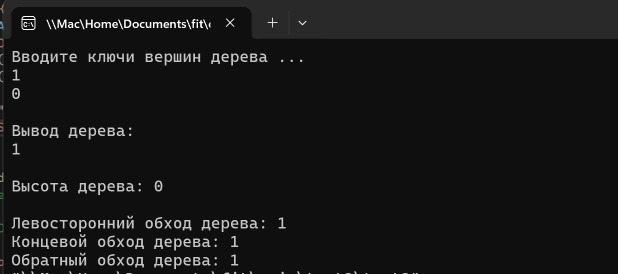
}



Таким образом, при вводе первого 0 программа завершит свою работу, что предотвратит потенциальные ошибки и упростит управление программой.

### Проверка ввода одного элемента

При вводе одного элемента в метод BuildTree() дерево работает корректно, так как программа добавляет единственный введенный ключевой элемент в дерево и завершает процесс построения дерева.



## Vyvod

Метод `Vyvod()` в классе `TREE` отвечает за отображение дерева на экране дисплея в виде визуального представления. Он рекурсивно обходит дерево и выводит узлы в определенном формате. Процесс работы метода можно описать следующим образом:

1. Аргументы: Метод принимает два аргумента:

    - Указатель на узел дерева (`node\*\* w`), с которого начинается вывод. Обычно это корневой узел.

    - Целое число (`int l`), обозначающее уровень узла в дереве (глубина рекурсии).

2. Рекурсивный обход: Метод работает рекурсивно, обходя дерево от корневого узла до самых нижних уровней:

    - Сначала метод рекурсивно вызывает сам себя для правого потомка текущего узла (`\*\*w`), увеличивая уровень (`l`) на 1. Это делает так, что правые узлы дерева выводятся выше по экрану.

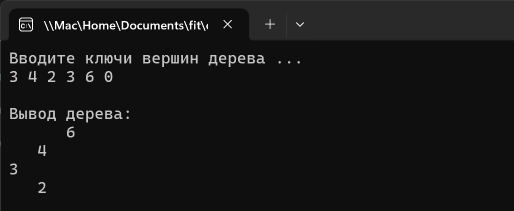
    - После обработки правого потомка метод выводит пробелы в количестве, равном текущему уровню (`l`), чтобы визуально сместить узлы вправо в зависимости от их уровня в дереве. Затем он выводит ключ (`(\*\*w).Key`) текущего узла.

3. Обработка левого потомка: Затем метод рекурсивно вызывает сам себя для левого потомка текущего узла (`\*\*w`), также увеличивая уровень (`l`) на 1. Это позволяет выводить левый узел ниже правого узла и корневого узла.

4. Продолжение рекурсии: Процесс рекурсивного обхода продолжается для всех узлов в дереве, сначала для правых, затем для текущих, и затем для левых узлов.

5. Формат вывода: Визуальное представление дерева на экране создается путем смещения узлов вправо в зависимости от их уровня (глубины) в дереве, что создает видимость дерева в виде графа.

Метод `Vyvod()` обеспечивает наглядное представление дерева на экране, позволяя пользователю увидеть структуру дерева, в том числе расположение узлов и их уровень.



## ObhodLeft

Метод `ObhodLeft()` в классе `TREE` отвечает за выполнение левостороннего обхода бинарного дерева поиска и вывода ключей узлов в процессе обхода. Левосторонний обход дерева также известен как прямой (preorder) обход дерева. Процесс работы метода можно описать следующим образом:

1. Аргумент: Метод принимает один аргумент:

Указатель на узел дерева (`node\*\* w`), с которого начинается обход. Обычно это корневой узел.

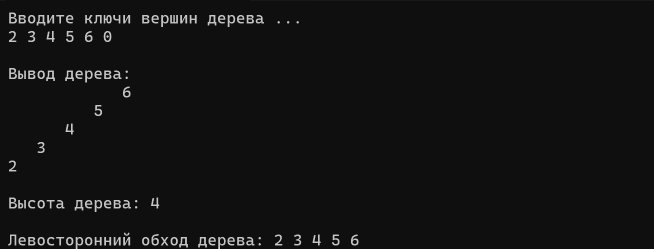
2. Обработка текущего узла: Метод сначала выводит ключ (`(\*\*w).Key`) текущего узла.

3. Рекурсивный обход левого потомка: После обработки текущего узла метод рекурсивно вызывает сам себя для левого потомка текущего узла (`\*\*w`). Это позволяет продолжить обход слева, начиная с левого потомка текущего узла.

4. Рекурсивный обход правого потомка: После обхода левого потомка метод рекурсивно вызывает сам себя для правого потомка текущего узла (`\*\*w`). Это позволяет продолжить обход справа, начиная с правого потомка текущего узла.

5. Продолжение рекурсии: Процесс рекурсивного обхода продолжается для всех узлов в дереве, сначала для текущего узла, затем для его левого потомка, и в заключение для правого потомка.

Метод `ObhodLeft()` выполняет левосторонний обход дерева и выводит ключи узлов в порядке: текущий узел, левый потомок, правый потомок. Это позволяет пользователю увидеть ключи узлов в порядке, соответствующем прямому обходу дерева, начиная с корневого узла.

****

## ObhodEnd

Метод `ObhodEnd()` в классе `TREE` отвечает за выполнение концевого обхода бинарного дерева поиска и вывода ключей узлов в процессе обхода. Концевой обход дерева также известен как обход в порядке (postorder) обход дерева. Процесс работы метода можно описать следующим образом:

1. Аргумент: Метод принимает один аргумент:

    - Указатель на узел дерева (`node\*\* w`), с которого начинается обход. Обычно это корневой узел.

2. Рекурсивный обход левого потомка: Метод сначала рекурсивно вызывает сам себя для левого потомка текущего узла (`\*\*w`). Это позволяет начать обход с левой части дерева.

3. Рекурсивный обход правого потомка: После обхода левого потомка метод рекурсивно вызывает сам себя для правого потомка текущего узла (`\*\*w`). Это позволяет продолжить обход с правой части дерева.

4. Обработка текущего узла: После обхода левого и правого потомков метод выводит ключ (`(\*\*w).Key`) текущего узла. Таким образом, текущий узел обрабатывается после своих потомков.

5. Продолжение рекурсии: Процесс рекурсивного обхода продолжается для всех узлов в дереве, сначала для левого потомка, затем для правого потомка, и в заключение для текущего узла.

Метод `ObhodEnd()` выполняет концевой обход дерева и выводит ключи узлов в порядке: левый потомок, правый потомок, текущий узел. Это позволяет пользователю увидеть ключи узлов в порядке, соответствующем обходу в порядке (postorder) дерева, начиная с корневого узла.

**Снимок экрана 2024-05-01 в 10.34.53**

## ObhodBack

Метод `ObhodBack()` в классе `TREE` отвечает за выполнение обратного обхода бинарного дерева поиска и вывода ключей узлов в процессе обхода. Обратный обход дерева также известен как симметричный (inorder) обход дерева. Процесс работы метода можно описать следующим образом:

1. Аргумент: Метод принимает один аргумент:

    - Указатель на узел дерева (`node\*\* w`), с которого начинается обход. Обычно это корневой узел.

2. Рекурсивный обход левого потомка: Метод сначала рекурсивно вызывает сам себя для левого потомка текущего узла (`\*\*w`). Это позволяет начать обход с левой части дерева.

3. Обработка текущего узла После обхода левого потомка метод выводит ключ (`(\*\*w).Key`) текущего узла. Таким образом, текущий узел обрабатывается между обходом левого и правого потомков.

4. Рекурсивный обход правого потомка: После обработки текущего узла метод рекурсивно вызывает сам себя для правого потомка текущего узла (`\*\*w`). Это позволяет продолжить обход с правой части дерева.

5. Продолжение рекурсии: Процесс рекурсивного обхода продолжается для всех узлов в дереве, начиная с левого потомка, затем текущего узла, и заканчивая правым потомком.

Метод `ObhodBack()` выполняет обратный обход дерева и выводит ключи узлов в порядке: левый потомок, текущий узел, правый потомок. Это позволяет пользователю увидеть ключи узлов в порядке, соответствующем симметричному (inorder) обходу дерева, начиная с корневого узла.

**Снимок экрана 2024-05-01 в 10.36.39**

## CleanTree

Метод `CleanTree` используется для очистки бинарного дерева. Он рекурсивно удаляет все узлы дерева, освобождая выделенную под них память. Метод принимает указатель на указатель корневого узла дерева (`node\*\* w`) и выполняет следующие действия:

1. Проверка на `NULL`: Если текущий узел (`\*w`) равен `NULL`, то дерево пустое, и функция завершает свою работу.

2. Очистка левого поддерева: Метод вызывает сам себя (`CleanTree`) рекурсивно для левого поддерева текущего узла (`&(\*\*w).Left`). Это продолжает процесс очистки слева.

3. Очистка правого поддерева: Аналогично, метод вызывает сам себя рекурсивно для правого поддерева текущего узла (`&(\*\*w).Right`). Это продолжает процесс очистки справа.

4. Удаление текущего узла: После очистки левого и правого поддеревьев, текущий узел (`\*w`) удаляется с помощью оператора `delete`.

В результате выполнения метода `CleanTree` вся память, выделенная под дерево, освобождается, и дерево становится пустым. Корневой узел должен быть установлен в `NULL`, чтобы указать, что дерево было очищено.

## Добавим поиск максимального и минимальных значений

`FindMax`:

    - Эта функция предназначена для поиска максимального значения в дереве.

    - Она начинает обход дерева с корня и переходит к самому правому узлу, так как в нем находится максимальное значение.

    - Функция возвращает ключ узла с максимальным значением в дереве.

`FindMin`:

    - Эта функция предназначена для поиска минимального значения в дереве.

    - Она начинает обход дерева с корня и переходит к самому левому узлу, так как в нем находится минимальное значение.

    - Функция возвращает ключ узла с минимальным значением в дереве.

// Функция для поиска максимального значения в дереве.

int TREE::FindMax(node\*\* w) {

    if (\*w == NULL) {

        return INT\_MIN; // Если дерево пустое, возвращаем наименьшее целое значение.

    }

    // Переходим к самому правому узлу, так как в нем находится максимальное значение.

    node\* current = \*w;

    while (current->Right != NULL) {

        current = current->Right;

    }

    return current->Key; // Возвращаем максимальный ключ.

}

// Функция для поиска минимального значения в дереве.

int TREE::FindMin(node\*\* w) {

    if (\*w == NULL) {

        return INT\_MAX; // Если дерево пустое, возвращаем наибольшее целое значение.

    }

    // Переходим к самому левому узлу, так как в нем находится минимальное значение.

    node\* current = \*w;

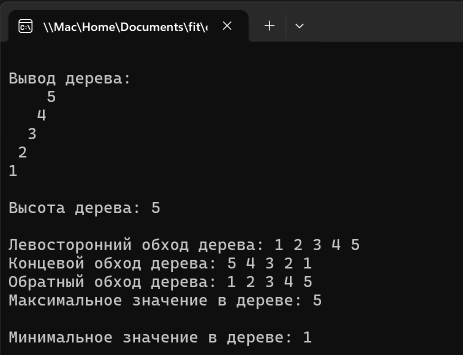
    while (current->Left != NULL) {

        current = current->Left;

    }

    return current->Key; // Возвращаем минимальный ключ.

}



## Добавим преобразование дерева в массив

`TreeToArray`:

    - Эта функция преобразует дерево в отсортированный массив, используя обход дерева по порядку.

    - Она рекурсивно обходит дерево, добавляя каждый узел в массив в порядке возрастания.

    - Функция принимает указатель на корень дерева, массив для записи значений и текущий индекс массива, который будет увеличиваться по мере добавления новых значений.

// Функция для преобразования дерева в массив.

void TREE::TreeToArray(node\*\* w, int arr[], int& index) {

    if (\*w != NULL) {

        // Рекурсивно обходим левое поддерево.

        TreeToArray(&((\*w)->Left), arr, index);

        // Добавляем текущий ключ в массив.

        arr[index] = (\*w)->Key;

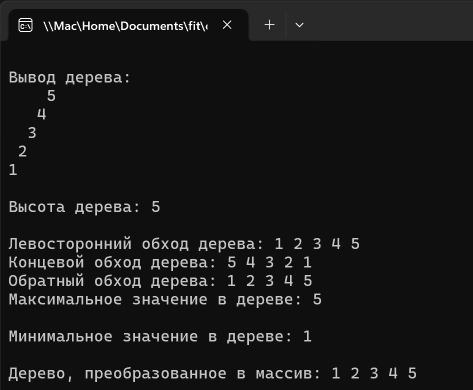
        index++; // Увеличиваем индекс массива.

        // Рекурсивно обходим правое поддерево.

        TreeToArray(&((\*w)->Right), arr, index);

    }

}



## Добавим удаление узла по значению

`DeleteNode`:

    - Эта функция удаляет узел из дерева по заданному значению ключа.

    - Функция сначала находит узел, соответствующий заданному значению, рекурсивно обходя дерево.

    - После нахождения узла, функция удаляет его и делает необходимые перестройки в дереве, чтобы сохранить структуру.

    - Если у удаляемого узла есть дети, функция находит и использует наименьший узел в правом поддереве для замены значения удаляемого узла.

// Функция для удаления узла по значению.

node\* TREE::DeleteNode(node\*\* w, int value) {

    if (\*w == NULL) {

        return NULL; // Узел не найден.

    }

    // Если значение меньше текущего ключа, ищем в левом поддереве.

    if (value < (\*w)->Key) {

        (\*w)->Left = DeleteNode(&((\*w)->Left), value);

    }

    // Если значение больше текущего ключа, ищем в правом поддереве.

    else if (value > (\*w)->Key) {

        (\*w)->Right = DeleteNode(&((\*w)->Right), value);

    }

    // Если значение равно текущему ключу, удаляем узел.

    else {

        // Если у узла нет детей или есть только один ребенок.

        if ((\*w)->Left == NULL) {

            node\* temp = (\*w)->Right;

            delete\* w;

            return temp;

        }

        else if ((\*w)->Right == NULL) {

            node\* temp = (\*w)->Left;

            delete\* w;

            return temp;

        }

        // У узла есть два ребенка, находим наименьший узел в правом поддереве.

        node\* temp = (\*w)->Right;

        while (temp && temp->Left != NULL) {

            temp = temp->Left;

        }

        // Копируем значение минимального узла в текущий узел.

        (\*w)->Key = temp->Key;

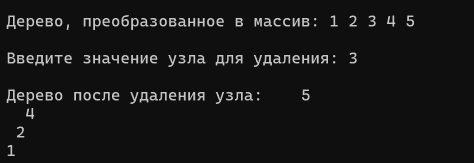
        // Удаляем минимальный узел в правом поддереве.

        (\*w)->Right = DeleteNode(&((\*w)->Right), temp->Key);

    }

    return \*w;

}

****

## Заключение

Исследование структуры данных в виде двоичного дерева поиска показало его значимость и эффективность при управлении данными в программировании. Представленный код наглядно демонстрирует реализацию основных методов для работы с этой структурой, включая вставку элементов, обход дерева разными способами, определение его высоты и очистку дерева.

Метод `CleanTree` является важным компонентом для поддержания чистоты и корректности дерева. Рекурсивный подход, используемый в этом методе, позволяет полностью очищать дерево, удаляя все узлы и освобождая выделенную память. Это гарантирует корректное функционирование программы и предотвращает утечки памяти.

Таким образом, рассмотренная реализация двоичного дерева поиска и методов работы с ним предоставляет прочное основание для эффективного управления данными, обеспечивая при этом надежность и эффективность программного кода.