Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

**ТЕСТИРОВАНИЕ**

По предмету “Основы алгоритмизации и программирования”

На тему: “Бинарное дерево поиска нерекурсивно”

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. TreeSearch
   2. VyvodTree
   3. PushStack
   4. PopStack
   5. VyvodStack
5. Добавим поиск максимального и минимальных значений
6. Добавим преобразование дерева в массив
7. Добавим удаление узла по значению
8. Заключение

## Введение

В данной работе рассматриваются алгоритмы обработки бинарных деревьев поиска, представляющие собой эффективные структуры данных для хранения и управления упорядоченными наборами данных. Особое внимание уделяется функциям обработки дерева, включая поиск узлов по заданному ключу (`TreeSearch`) и обход дерева в определенном порядке с целью вывода его содержимого на экран (`VyvodTree`). В реферате также рассматриваются вспомогательные функции работы со стеком (`PushStack`, `PopStack`, `VyvodStack`) и дополнительные функции (`CountNodes`, `ContainsValue`) для подсчета количества узлов в дереве и проверки наличия определенного значения в дереве. Все функции анализируются с точки зрения их роли в обеспечении эффективной работы с бинарными деревьями поиска.

## Исходный код с добавлением комментариев

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

using namespace std; // Использование пространства имен std

// Структура для представления узла дерева

struct node {

int Key; // Ключ (значение) узла

int Count; // Счетчик количества повторений ключа

node\* Left; // Указатель на левое поддерево

node\* Right; // Указатель на правое поддерево

};

// Структура для представления звена стека

struct no {

node\* elem; // Указатель на узел дерева

int ch; // Уровень вершины в дереве

no\* sled; // Указатель на следующее звено стека

};

// Класс для представления дерева

class TREE {

private:

node\* Tree; // Указатель на корень дерева

// Функция для помещения звена с элементами \*el и \*n в стек

void PushStack(no\*\*, node\*\*, int\*);

// Функция для извлечения из стека звена с элементами \*t и \*n

void PopStack(no\*\*, node\*\*, int\*);

// Функция для вывода содержимого стека на экран дисплея

void VyvodStack(no\*\*);

public:

TREE() {

// Инициализация корня дерева

Tree = new(node);

// Установка правого поддерева корня в NULL

(\*Tree).Right = NULL;

}

// Функция для получения правого поддерева дерева

node\* GetTreeRight() {

// Возвращение указателя на правое поддерево корня

return (\*Tree).Right;

}

// Функция для поиска вершины с информационным полем el в дереве

// Если вершина не найдена, включение ее в дерево

void TreeSearch(int el);

// Функция для вывода дерева на экран дисплея (нерекурсивный алгоритм)

void VyvodTree(node\* t);

};

// Функция main - точка входа в программу

void main() {

// Установка локализации на русский язык

setlocale(LC\_ALL, "Rus");

TREE A; // Создание экземпляра класса TREE

int el; // Переменная для ввода значений

// Вывод сообщения о вводе значений вершин

cout << "Вводите значения информационных полей вершин: " << endl;

// Ввод первого значения информационного поля вершины

cin >> el;

// Цикл для обработки всех введенных значений

while (el != 0) {

// Поиск или добавление узла в дерево

A.TreeSearch(el);

// Ввод следующего значения информационного поля вершины

cin >> el;

}

// Вывод дерева на экран дисплея

A.VyvodTree(A.GetTreeRight());

// Пауза перед завершением программы

cout << "\n";

system("PAUSE");

}

// Реализация функции TreeSearch

void TREE::TreeSearch(int el) {

node\* p1, \* p2; // Указатели на текущий и предыдущий узлы

int d; // Переменная для определения направления поиска

p2 = Tree; // Установка предыдущего узла на корень дерева

p1 = (\*p2).Right; // Установка текущего узла на правое поддерево корня

d = 1; // Начальное направление поиска

// Цикл для поиска узла с ключом el

while (p1 != NULL && d != 0) {

p2 = p1; // Перемещение предыдущего узла на текущий

if (el < (\*p1).Key) { // Поиск в левом поддереве

p1 = (\*p1).Left;

d = -1; // Направление поиска влево

}

else if (el > (\*p1).Key) { // Поиск в правом поддереве

p1 = (\*p1).Right;

d = 1; // Направление поиска вправо

}

else {

d = 0; // Узел найден

}

}

if (d == 0) {

// Если узел найден, увеличиваем счетчик повторений ключа

(\*p1).Count += 1;

}

else {

// Если узел не найден, создаем новый узел

p1 = new(node);

(\*p1).Key = el; // Установка ключа узла

(\*p1).Left = NULL; // Инициализация левого поддерева

(\*p1).Right = NULL; // Инициализация правого поддерева

(\*p1).Count = 1; // Инициализация счетчика повторений ключа

// Вставка нового узла в дерево

if (d < 0) {

(\*p2).Left = p1; // Вставка в левое поддерево

}

else {

(\*p2).Right = p1; // Вставка в правое поддерево

}

}

}

// Реализация функции VyvodTree

void TREE::VyvodTree(node\* t) {

no\* stk, \* stk1; // Указатели на стек

node\* u; // Временный указатель на узел

int i, n; // Переменные для отступа и уровня

stk = stk1 = NULL; // Инициализация стеков

n = 0; // Инициализация уровня

// Цикл для вывода дерева

while (t != NULL) {

// Помещение звена в стек

PushStack(&stk1, &t, &n);

// Если правое поддерево не пустое

if ((\*t).Right != NULL) {

// Если левое поддерево не пустое, помещаем его в стек

if ((\*t).Left != NULL) {

PushStack(&stk, &((\*t).Left), &n);

}

t = (\*t).Right; // Переход к правому поддереву

}

else {

// Если левое поддерево не пустое

if ((\*t).Left != NULL) {

if (stk1 != NULL) {

// Извлечение из стека и вывод значения

PopStack(&stk1, &u, &n);

for (i = 0; i <= n; i++) cout << " ";

cout << (\*u).Key << endl;

}

t = (\*t).Left; // Переход к левому поддереву

}

else if (stk == NULL) {

t = NULL; // Если стек пуст, завершение цикла

}

else {

// Цикл для извлечения и вывода значений из стека

while ((\*stk).elem != (\*((\*stk1).elem)).Left) {

PopStack(&stk1, &u, &n);

for (i = 0; i <= n; i++) cout << " ";

cout << (\*u).Key << endl;

}

// Извлечение из стека и вывод значения

PopStack(&stk1, &u, &n);

for (i = 0; i <= n; i++) cout << " ";

cout << (\*u).Key << endl;

// Переход к следующему узлу из стека

PopStack(&stk, &t, &n);

}

}

n = n + 1; // Увеличение уровня

}

VyvodStack(&stk1); // Вывод содержимого стека

}

// Реализация функции PushStack

void TREE::PushStack(no\*\* stk, node\*\* el, int\* n) {

// Создание нового звена стека

no\* q = new(no);

// Инициализация звена стека

(\*q).elem = \*el; (\*q).ch = \*n;

// Помещение звена в стек

(\*q).sled = \*stk;

\*stk = q;

}

// Реализация функции PopStack

void TREE::PopStack(no\*\* stk, node\*\* t, int\* n) {

no\* q; // Временное звено

// Если стек не пуст

if (\*stk != NULL) {

// Извлечение из стека звена с элементами \*t и \*n

\*t = (\*\*stk).elem;

\*n = (\*\*stk).ch;

// Удаление звена

q = \*stk;

\*stk = (\*\*stk).sled;

delete q;

}

}

// Реализация функции VyvodStack

void TREE::VyvodStack(no\*\* stk) {

node\* k; // Временный указатель на узел

int i, n; // Переменные для отступа и уровня

// Цикл для вывода содержимого стека

while (\*stk !=

NULL) {

k = (\*\*stk).elem; // Извлечение узла

n = (\*\*stk).ch; // Извлечение уровня

// Вывод узла с отступом

for (i = 0; i <= n; i++) cout << " ";

cout << (\*k).Key << endl;

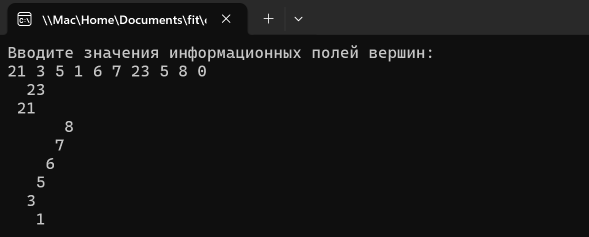
// Переход к следующему звену стека

\*stk = (\*\*stk).sled;

}

}

Результат работы программы:



## О работе кода

Этот код представляет собой реализацию бинарного дерева поиска и включает в себя операции по добавлению, поиску и отображению узлов дерева. Код использует классы и структуры для организации данных и функций, необходимых для работы с деревом и стеком.

Вот краткое описание каждой части кода:

1. Структура `node`:

Представляет узел бинарного дерева поиска.

Содержит ключ (`Key`), счетчик повторений (`Count`), и указатели на левое (`Left`) и правое (`Right`) поддеревья.

2. Структура `no`:

Представляет звено стека.

Содержит указатель на узел дерева (`elem`), уровень вершины в дереве (`ch`), и указатель на следующее звено стека (`sled`).

3. Класс `TREE`:

Класс для работы с бинарным деревом поиска.

Содержит частные члены: указатель на корень дерева (`Tree`), функции для работы со стеком (`PushStack`, `PopStack`, `VyvodStack`), и публичные члены: конструктор по умолчанию, методы для получения правого поддерева (`GetTreeRight`), поиска или добавления узла в дерево (`TreeSearch`), и вывода дерева на экран (`VyvodTree`).

4. Функция `main`:

Точка входа в программу.

Создает экземпляр класса `TREE` (`A`), читает входные значения от пользователя, добавляет их в дерево (`TreeSearch`), и отображает дерево на экран (`VyvodTree`).

Программа завершает работу после ввода значения `0`.

5. Функция `TREE::TreeSearch`:

Ищет узел с заданным значением в дереве.

Если узел не найден, добавляет новый узел в дерево и увеличивает счетчик повторений.

6. Функция `TREE::VyvodTree`:

Выводит дерево на экран дисплея.

Использует стеки для нерекурсивного обхода дерева.

7. Функция `TREE::PushStack`:

Добавляет звено в стек.

8. Функция `TREE::PopStack`:

Извлекает звено из стека.

9. Функция `TREE::VyvodStack`:

Выводит содержимое стека на экран дисплея.

Программа предоставляет базовую реализацию бинарного дерева поиска и различные методы для работы с деревом, включая добавление, поиск и вывод дерева на экран.

## TreeSearch

Функция в коде выполняет следующие задачи:

Поиск узла в дереве: Функция ищет узел в дереве по заданному ключу (`el`). Начинает поиск от корневого узла дерева и продолжает двигаться вниз по дереву, пока не находит узел с указанным ключом или не достигает конца дерева.

Добавление нового узла: Если узел с таким ключом не найден, функция добавляет новый узел с этим ключом в дерево. Если значение ключа (`el`) меньше ключа текущего узла, функция перемещается влево; если больше — вправо. Когда достигается пустое место в дереве, функция создает новый узел и вставляет его в дерево.

Обновление счетчика: Если узел с указанным ключом найден, функция увеличивает счетчик (`Count`) для этого узла на единицу, указывая на то, что это значение уже существует в дереве и появляется еще раз.

Детали работы функции:

1. Инициализация указателей: Функция инициализирует два указателя: `p1` и `p2`. `p2` указывает на корневой узел дерева (`Tree`), а `p1` указывает на правое поддерево корневого узла.

2. Цикл поиска: Функция запускает цикл, в котором она проверяет значение ключа (`el`) относительно ключа текущего узла (`p1`):

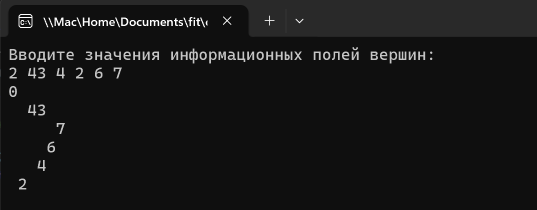
- Если `el` меньше ключа текущего узла, функция переходит к левому поддереву и устанавливает флаг `d` в -1.

- Если `el` больше ключа текущего узла, функция переходит к правому поддереву и устанавливает флаг `d` в 1.

- Если `el` равен ключу текущего узла, функция устанавливает флаг `d` в 0, прерывая цикл.

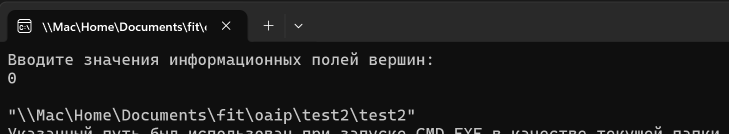
3. Обработка результатов поиска: Если флаг `d` равен 0, функция увеличивает счетчик (`Count`) для найденного узла. Если `d` не равен 0, функция создает новый узел, устанавливает его ключ и счетчик, и затем вставляет его в соответствующее место в дереве, ориентируясь на флаг `d`.

Таким образом, `TreeSearch` отвечает за поиск и добавление узлов в дерево, а также за обновление счетчиков существующих узлов, что позволяет дереву корректно обрабатывать повторяющиеся значения ключей.



### При вводе пустого дерева

При первом вводе значения `0` программа работает корректно. Она сразу завершает цикл ввода, пропуская добавление новых узлов в дерево. Таким образом, дерево остается пустым, и программа выводит пустое дерево, завершая свою работу без ошибок.



## VyvodTree

Функция `VyvodTree` выводит содержимое бинарного дерева поиска в консоль в структурированном виде. Она реализует обход дерева в порядке обхода слева направо (in-order traversal) для отображения структуры дерева и ключей узлов.

Процесс начинается с инициализации двух стеков (`stk` и `stk1`) и уровня `n`. Затем начинается цикл, который будет выполняться до тех пор, пока указатель на текущий узел (`t`) не станет равен `NULL`.

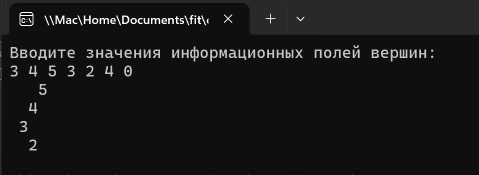
На каждой итерации цикла функция помещает текущий узел и его уровень в стек `stk1` с помощью функции `PushStack`. Если у текущего узла есть правое поддерево, функция проверяет наличие левого поддерева и помещает его в стек `stk` (если левое поддерево не пустое). Затем происходит переход к правому поддереву.

Если у текущего узла нет правого поддерева, функция проверяет наличие левого поддерева. Если левое поддерево не пустое, происходит вывод ключа узла из стека `stk1` с соответствующим отступом, определяемым уровнем `n`. Затем происходит переход к левому поддереву.

Если стек `stk` пуст, цикл завершается, и функция выводит содержимое стека `stk1` на консоль.

Если левое поддерево отсутствует и стек `stk` не пуст, происходит извлечение узлов из стека `stk1` и вывод их ключей с отступами, затем происходит переход к следующему узлу из стека `stk`.

После завершения цикла функция выводит содержимое стека `stk1` на консоль, завершая вывод содержимого бинарного дерева поиска.



## PushStack

Функция `PushStack` добавляет новый элемент в стек, который используется для временного хранения информации во время выполнения других операций. Она выполняет эту задачу с помощью следующих шагов:

1. Создание нового элемента стека: Функция выделяет память для нового элемента стека (`no`) с помощью оператора `new`.

2. Инициализация нового элемента стека: Функция инициализирует поля нового элемента стека:

- Поле `elem` устанавливается на указатель на узел дерева (`\*el`), переданный в функцию.

- Поле `ch` устанавливается на уровень узла (`\*n`), также переданный в функцию.

- Поле `sled` устанавливается на текущий верхний элемент стека (`\*stk`).

3. Обновление указателя на верх стека: После создания нового элемента стека и его инициализации, функция обновляет указатель на верх стека (`\*stk`), чтобы указывать на только что добавленный элемент. Таким образом, новый элемент стека становится верхним элементом стека.

В итоге, функция `PushStack` добавляет новый элемент в стек, сохраняя указатель на узел дерева и его уровень, и обновляет верх стека, чтобы указывать на только что добавленный элемент. Это позволяет управлять стеком в других функциях для реализации операций с деревом, таких как обход и вывод.

## PopStack

Функция `PopStack` выполняет извлечение элемента из стека и возвращает его информацию. Эта функция управляет стеком и используется для обработки временно сохраненных данных во время выполнения других операций. Функция работает следующим образом:

1. Проверка наличия элементов в стеке: Функция сначала проверяет, есть ли элементы в стеке (`\*stk != NULL`). Если стек пуст, функция ничего не делает.

2. Извлечение верхнего элемента стека: Если в стеке есть элементы, функция извлекает верхний элемент стека:

- Поле `elem` верхнего элемента стека (`\*\*stk`) копируется в указатель на узел дерева (`\*t`), переданный в функцию.

- Поле `ch` верхнего элемента стека (`\*\*stk`) копируется в переменную уровня (`\*n`), переданную в функцию.

3. Обновление указателя на верх стека: После извлечения верхнего элемента стека, функция обновляет указатель на верх стека (`\*stk`), чтобы указывать на следующий элемент в стеке (`(\*\*stk).sled`).

4. Освобождение памяти: После того, как верхний элемент стека извлечен, функция удаляет его (`delete q`), освобождая память.

Функция `PopStack` возвращает информацию о верхнем элементе стека (узел дерева и его уровень) и удаляет этот элемент из стека, освобождая память. Это позволяет использовать извлеченные данные для продолжения обработки дерева.

## VyvodStack

Функция `VyvodStack` отвечает за вывод содержимого стека на экран дисплея. Она выполняет следующие действия:

1. Инициализация: Функция принимает указатель на стек (`\*\*stk`) и начинает работу, проверяя, есть ли в стеке элементы.

2. Цикл: Пока в стеке есть элементы, функция выполняет следующий цикл:

- Извлекает верхний элемент стека (`\*stk`) и получает узел дерева (`k = (\*\*stk).elem`) и уровень отступа (`n = (\*\*stk).ch`).

- Использует переменную `i` для отслеживания уровня отступа и выводит нужное количество пробелов (`for (i = 0; i <= n; i++) cout << " ";`) перед выводом ключа узла (`(\*k).Key`).

- После этого `\*stk` обновляется на следующий элемент в стеке (`\*stk = (\*\*stk).sled`).

3. Повторение: Цикл продолжается, пока стек не будет пустым.

Функция последовательно выводит содержимое стека на экран, начиная с верхнего элемента и продвигаясь вниз по стеку. Каждый узел дерева выводится с соответствующим уровнем отступа, что создает визуальное представление дерева с иерархическим отношением между узлами.

## Добавим подсчет количества узлов

CountNodes(node\* root):

- Эта функция предназначена для подсчета количества узлов в дереве, начиная с узла, переданного в качестве аргумента.

- Если переданный узел пустой (`NULL`), функция возвращает `0`, так как в пустом дереве нет узлов.

- В противном случае функция считает текущий узел (`1`) и рекурсивно вызывает себя для подсчета узлов в левом и правом поддеревьях, добавляя их к сумме.

- В итоге возвращается общее количество узлов в дереве.

// Подсчет количества узлов в дереве

int count = A.CountNodes();

cout << "\nКоличество узлов в дереве: " << count << endl;

// Функция подсчета количества узлов в дереве

int CountNodes(node\* root) {

if (root == NULL) {

return 0; // Если узел пустой, возвращаем 0

}

// Считаем текущий узел (1) и рекурсивно подсчитываем потомков

return 1 + CountNodes(root->Left) + CountNodes(root->Right);

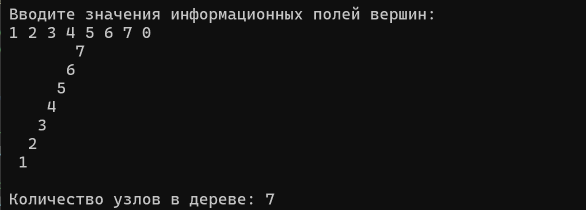
}

// Обертки для функций `CountNodes` и `ContainsValue`, которые работают с корнем дерева

int CountNodes() {

return CountNodes(Tree->Right); // Подсчет количества узлов в правом поддереве

}



## 

## Добавим поиск элемента

ContainsValue(int value, node\* root):

- Эта функция проверяет наличие заданного значения (`value`) в дереве, начиная с узла, переданного в качестве аргумента.

- Если переданный узел пустой (`NULL`), функция возвращает `false`, так как в пустом дереве нет узлов с заданным значением.

- Если значение узла совпадает с искомым значением, функция возвращает `true`, так как значение найдено.

- Если искомое значение меньше значения узла, функция рекурсивно вызывает себя для поиска в левом поддереве.

- Если искомое значение больше значения узла, функция рекурсивно вызывает себя для поиска в правом поддереве.

- Таким образом, функция рекурсивно проверяет все поддеревья, чтобы найти искомое значение, и возвращает `true`, если оно найдено, и `false`, если нет.

// Функция проверки наличия значения в дереве

bool ContainsValue(int value, node\* root) {

if (root == NULL) {

return false; // Если узел пустой, значение не найдено

}

// Если значение найдено, возвращаем true

if (root->Key == value) {

return true;

}

// Рекурсивно проверяем левое и правое поддеревья

if (value < root->Key) {

return ContainsValue(value, root->Left);

}

else {

return ContainsValue(value, root->Right);

}

}

bool ContainsValue(int value) {

return ContainsValue(value, Tree->Right); // Проверка наличия значения в правом поддереве

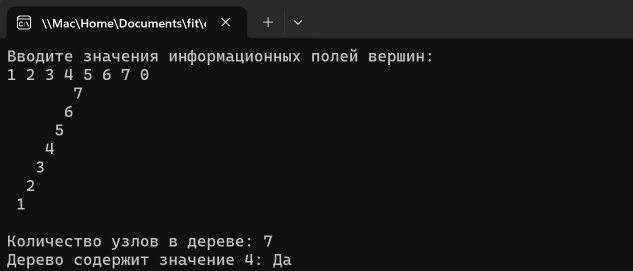
}

// Проверка наличия значения 4 в дереве

int valueToCheck = 4;

bool hasValue = A.ContainsValue(valueToCheck);

cout << "Дерево содержит значение " << valueToCheck << ": " << (hasValue ? "Да" : "Нет") << endl;



## Заключение

В ходе анализа реализованных функций обработки бинарного дерева поиска можно отметить, что каждая из них выполняет свою задачу, способствуя эффективному управлению данными в структуре. Функция `TreeSearch` обеспечивает быстрый поиск узлов по заданному ключу и добавление новых узлов при необходимости. Функция `VyvodTree` предоставляет удобный способ обхода дерева и вывода его содержимого в структурированном виде. Вспомогательные функции для работы со стеком (`PushStack`, `PopStack`, `VyvodStack`) позволяют эффективно управлять временно сохраненными данными во время обхода дерева. Дополнительные функции (`CountNodes`, `ContainsValue`) расширяют функциональность класса, позволяя считать количество узлов в дереве и проверять наличие заданного значения. В целом, реализованные функции обеспечивают эффективное и интуитивное управление бинарным деревом поиска.