**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**ОТЧЕТ**

**по лабораторной работе № 3**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Деревья»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 3311 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Сапронов К. Д.  Землякова С. А. |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Преподаватель |  | Манирагена Валенс |

Санкт-Петербург

2024

Цель работы: исследование алгоритмов для работы с двоичным деревом.

Задание (вариант 31):

- симметричная разметка;

- обход дерева «в глубину»;

- требуется вычислить количество вершин на глубине не более 2.

В памяти ЭВМ деревья представляются в виде объектов классов Node и Tree: с таким представлением удобно как организовать сложную структуру дерева, так и обеспечить работу с узлами дерева через методы, встроенные в эти классы.

Узлы дерева (Node) включают в себя данные узла d и указатели на потомков (lft и rgt), что обеспечивает возможность строить иерархическую структуру. Класс Tree инкапсулирует корень дерева (root), методы построения, вывода, обхода и удаления дерева, что делает структуру дерева понятной и удобной для пользователя программы. Пользователю не нужно напрямую работать с узлами — это делает сам класс Tree.

Класс Node имеет деструктор, который рекурсивно удаляет поддеревья, а класс Tree освобождает всю динамическую память, связанную с визуализацией (SCREEN). Благодаря этому дерево очищает память после завершения работы. Это минимизирует риск утечек памяти при создании и удалении узлов, так как узлы удаляются при вызове деструктора дерева.

Использование классов позволяет встроить в Tree разные методы обхода, такие как симметричный (внутренний), прямой и обратный обходы. В классе Tree организованы функции для создания дерева как случайным образом, так и вручную, что добавляет гибкости к способу построения дерева.

Наконец, с использованием классов становится легче реализовывать специфические задачи (напр., считать количество узлов на глубине до 2).

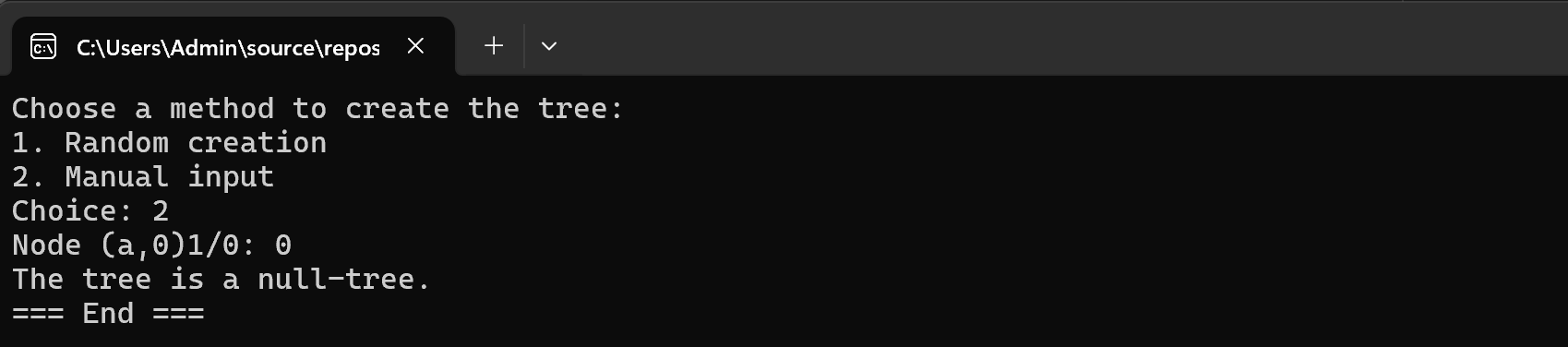
**Контрольные тесты.**

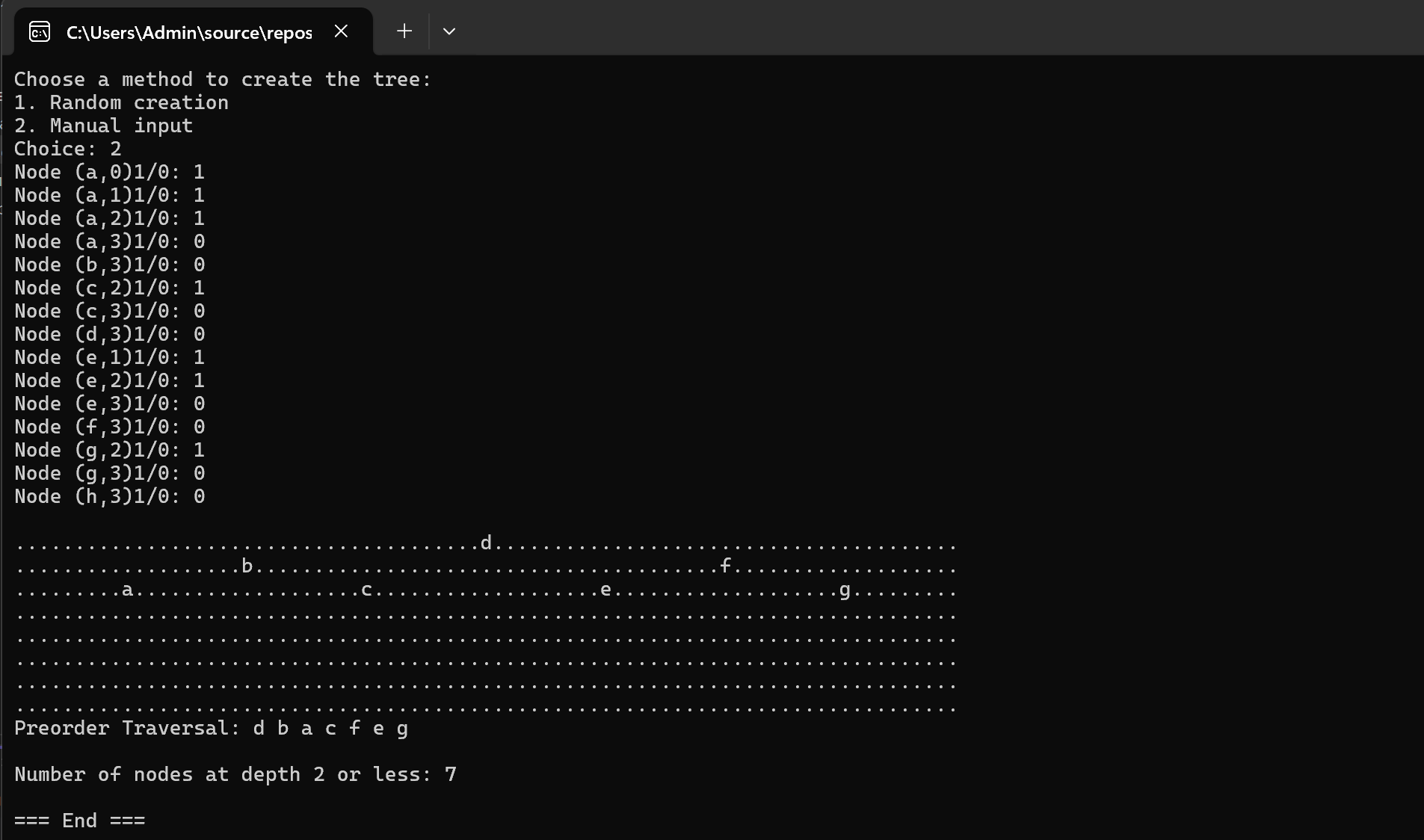
1. Случайная генерация



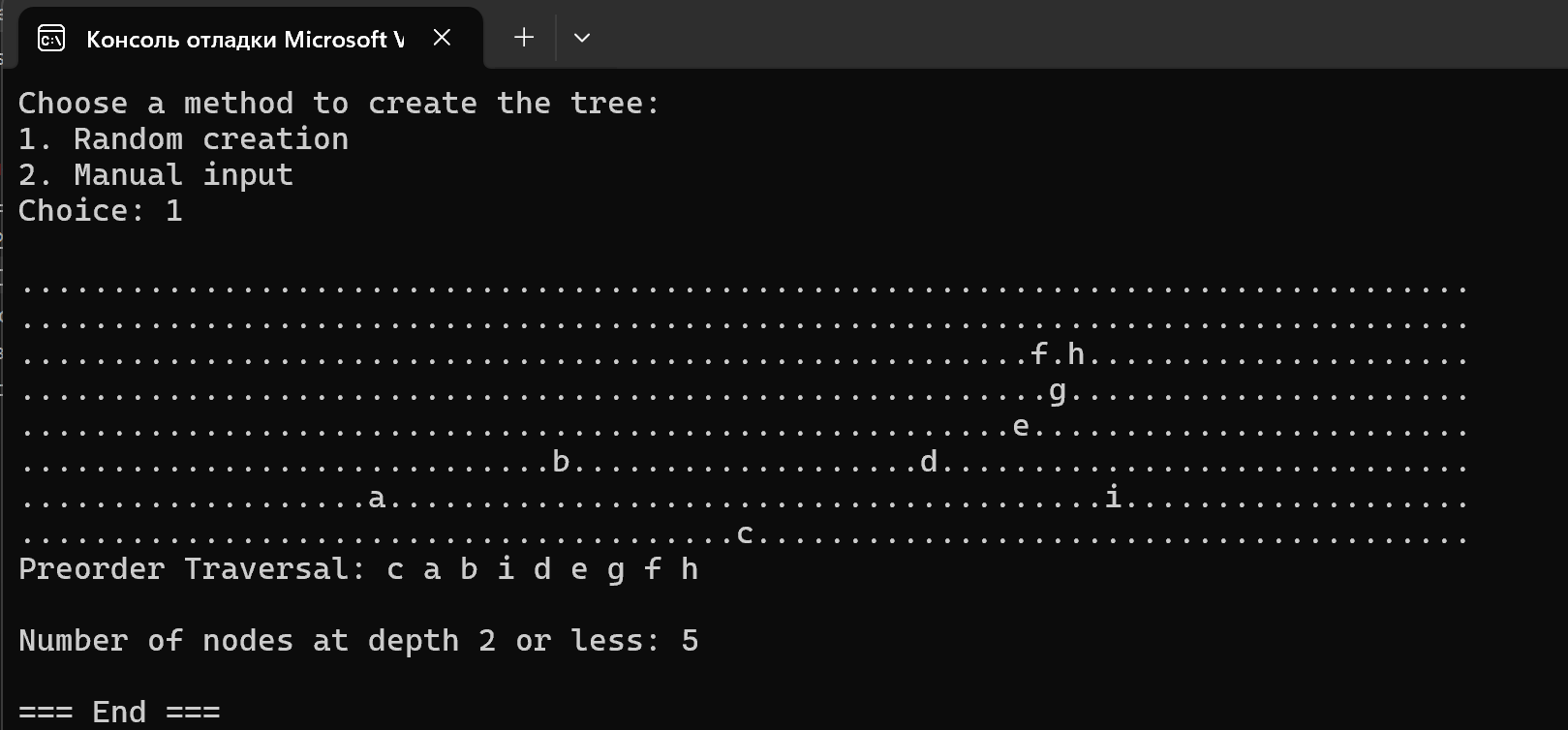


1. Ручной ввод

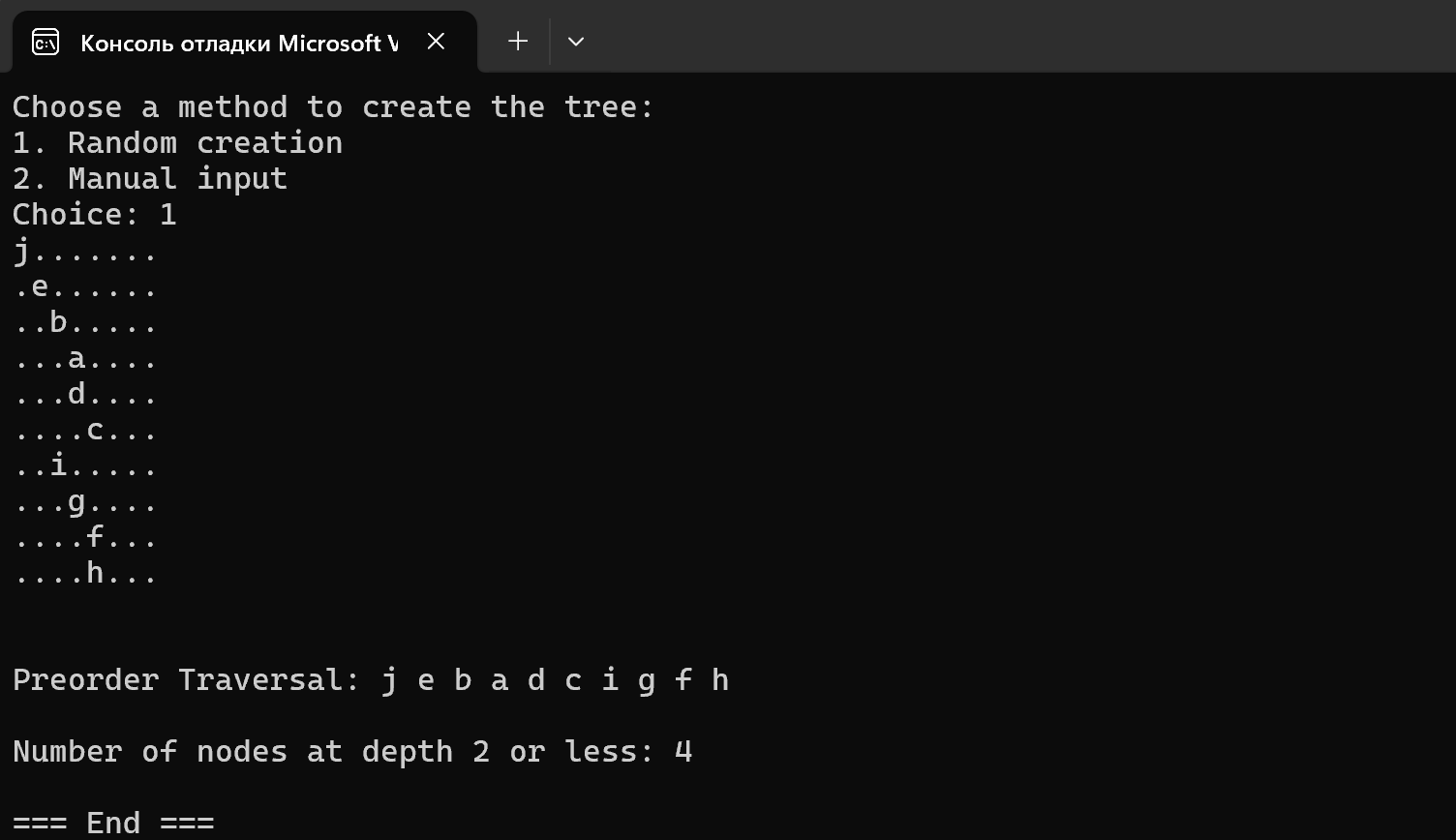




1. Вывод дерева сверху вниз (корень снизу)



1. Вывод дерева слева направо (корень слева сверху)



**Оценка временной сложности.**

1. Создание дерева: *MakeTree* и *MakeNode*

*MakeTree*: функция запускает создание дерева, вызывая *MakeNode* с корневым узлом.

*MakeNode*: функция рекурсивно создаёт узлы на определённой глубине, задавая для каждого узла левого и правого потомка. На каждом уровне создаются новые узлы, пока не будет достигнута максимальная глубина или пока случайное значение не вернёт ложь, что прерывает создание дерева.

Сложность: в худшем случае, где дерево является полным двоичным деревом, временная сложность создания дерева составляет *O(2d)*, где *d* — глубина дерева.

2. Обход дерева: *preorderTraversal*

*preorderTraversal* – рекурсивный метод для прямого обхода дерева (обход в глубину), который посещает каждый узел ровно один раз. Для каждого узла функция выполняет постоянное количество операций (печатает метку узла и вызывает себя для его потомков).

Сложность: Временная сложность обхода дерева — *O(n)*, где *n* — общее количество узлов в дереве.

3. Вывод дерева: *OutTree* и *OutNodes*

*OutTree*: функция очищает экран и затем вызывает *OutNodes* для вывода дерева, начиная с корневого узла.

*OutNodes*: рекурсивная функция, проходит по всем узлам дерева и размещает каждый узел в массиве SCREEN, отображая его на экране. Аналогично функции обхода, *OutNodes* посещает каждый узел ровно один раз, выполняя постоянное количество операций.

Сложность: Временная сложность вывода дерева составляет *O(n)*, где *n* — количество узлов в дереве.

4. Подсчёт узлов на заданной глубине: *countDepthTwo*

*countDepthTwo*: рекурсивно подсчитывает узлы на глубине не более 2. Она также посещает каждый узел только один раз до заданной глубины, суммируя узлы при достижении глубины 2 или менее. Для каждого узла выполняется конечное число операций.

Сложность: в худшем случае, когда дерево глубже 2 и все узлы до глубины 2 существуют, временная сложность этой функции составляет *O(n)*, где *n* — количество узлов в дереве, находящихся до указанной глубины.

Временная сложность большинства функций для обхода и обработки дерева линейна относительно количества узлов, что эффективно, так как каждая функция проходит по каждому узлу не более одного раза.

**Текст программы.**

#include <iostream>

#include <ctime>

#include <cstring>

using namespace std;

class Node {

char d;

Node\* lft;

Node\* rgt;

public:

Node() : lft(nullptr), rgt(nullptr) {}

~Node() {

if (lft) delete lft;

if (rgt) delete rgt;

}

friend class Tree;

};

class Tree {

Node\* root;

char num, maxnum;

int maxrow, offset;

char\*\* SCREEN;

void clrscr();

Node\* MakeNode(int depth, bool random);

void OutNodes(Node\* v, int r, int c);

void preorderTraversal(Node\* node);

int countDepthTwo(Node\* node, int depth);

Tree(const Tree&) = delete;

Tree(Tree&&) = delete;

Tree operator=(const Tree&) const = delete;

Tree operator=(Tree&&) const = delete;

public:

Tree() = delete;

Tree(char num, char maxnum, int maxrow);

~Tree();

void MakeTree(bool random) { root = MakeNode(0, random); }

bool exist() { return root != nullptr; }

void OutTree();

void OutPreorderTraversal();

int countDepthTwo();

};

// Destructor

Tree::~Tree() {

for (int i = 0; i < maxrow; ++i) delete[] SCREEN[i];

delete[] SCREEN;

delete root;

}

// Constructor

Tree::Tree(char nm, char mnm, int mxr) : num(nm), maxnum(mnm), maxrow(mxr), offset(40), root(nullptr), SCREEN(new char\*[maxrow]) {

for (int i = 0; i < maxrow; ++i)

SCREEN[i] = new char[80];

}

// Screen clearing

void Tree::clrscr() {

for (int i = 0; i < maxrow; i++)

memset(SCREEN[i], '.', 80);

}

// Node creation with symmetric filling

Node\* Tree::MakeNode(int depth, bool random) {

Node\* v = nullptr;

int Y;

if (random)

Y = (depth < rand() % 6 + 1) && (num <= 'z');

else {

cout << "Node (" << num << ',' << depth << ")1/0: ";

cin >> Y;

}

if (Y) {

v = new Node;

v->lft = MakeNode(depth + 1, random);

v->d = num++;

v->rgt = MakeNode(depth + 1, random);

}

return v;

}

/\* ================= Top-down variant ================= \*/

// void Tree::OutNodes(Node\* v, int r, int c) {

// if (v == nullptr || r >= maxrow) return;

// if (r && c && (c < 80))

// SCREEN[r - 1][c - 1] = v->d;

// if (r < maxrow) {

// if (v->lft)

// OutNodes(v->lft, r + 1, c - (offset >> r));

// if (v->rgt)

// OutNodes(v->rgt, r + 1, c + (offset >> r));

// }

// }

// void Tree::OutTree() {

// clrscr();

// OutNodes(root, 1, offset);

// for (int i = 0; i < maxrow; i++) {

// SCREEN[i][79] = 0;

// cout << '\n' << SCREEN[i];

// }

// }

/\* ================= Bottom-up variant ================= \*/

//void Tree::OutNodes(Node\* v, int r, int c) {

// if (v == nullptr || r >= maxrow) return;

// if (r && c && (c < 80))

// SCREEN[maxrow - r][c - 1] = v->d;

// if (r < maxrow) {

// if (v->lft)

// OutNodes(v->lft, r + 1, c - (offset >> r));

// if (v->rgt)

// OutNodes(v->rgt, r + 1, c + (offset >> r));

// }

//}

//void Tree::OutTree() {

// clrscr();

// OutNodes(root, 1, offset);

// for (int i = 0; i < maxrow; i++) {

// SCREEN[i][79] = 0;

// cout << '\n' << SCREEN[i];

// }

//}

/\* ================= Left-to-right variant ================= \*/

void Tree::OutNodes(Node\* v, int depth, int indent) {

if (depth >= maxrow) return;

// Initialize a line with dots

string line(maxrow, '.');

if (v != nullptr) {

line[indent] = v->d;

cout << line << endl;

}

if (v->lft) OutNodes(v->lft, depth + 1, indent + 1);

if (v->rgt) OutNodes(v->rgt, depth + 1, indent + 1);

}

void Tree::OutTree() {

clrscr();

OutNodes(root, 0, 0);

cout << '\n';

}

// Preorder traversal

void Tree::preorderTraversal(Node\* node) {

if (node == nullptr)

return;

cout << node->d << " ";

preorderTraversal(node->lft);

preorderTraversal(node->rgt);

}

void Tree::OutPreorderTraversal() {

cout << "\nPreorder Traversal: ";

preorderTraversal(root);

cout << endl;

}

// Count nodes at depth <= 2

int Tree::countDepthTwo(Node\* node, int depth) {

if (node == nullptr || depth > 2)

return 0;

return 1 + countDepthTwo(node->lft, depth + 1) + countDepthTwo(node->rgt, depth + 1);

}

int Tree::countDepthTwo() {

return countDepthTwo(root, 0);

}

int main() {

srand(static\_cast<unsigned int>(time(0))); // Initialize random number generator

Tree Tr('a', 'z', 8);

int userOption;

cout << "Choose a method to create the tree:\n";

cout << "1. Random creation\n";

cout << "2. Manual input\n";

cout << "Choice: ";

cin >> userOption;

if (userOption == 1) Tr.MakeTree(true);

else if (userOption == 2) Tr.MakeTree(false);

else {

cout << "Invalid option.";

return 1;

}

if (Tr.exist()) {

Tr.OutTree();

Tr.OutPreorderTraversal();

int count = Tr.countDepthTwo();

cout << "\nNumber of nodes at depth 2 or less: " << count << endl;

}

else {

cout << "The tree is a null-tree.";

}

cout << "\n=== End ===";

return 0;

}