**«Санкт-Петербургский государственный электротехнический университет**

**«ЛЭТИ» им. В.И.Ульянова (Ленина)»**

**(СПбГЭТУ «ЛЭТИ»)**

Кафедра вычислительной техники

Отчет по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

|  |  |
| --- | --- |
| Студенты гр. 8307 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Никулин Л. |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Зуб В. |
|  |  |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Колинько П.Г. |

Тема: **ДЕРЕВЬЯ**

# Цель работы

# Исследование алгоритмов обхода дерева

# Задание

*8 вариант*

Вид дерева: Троичное

Разметка: Обратная

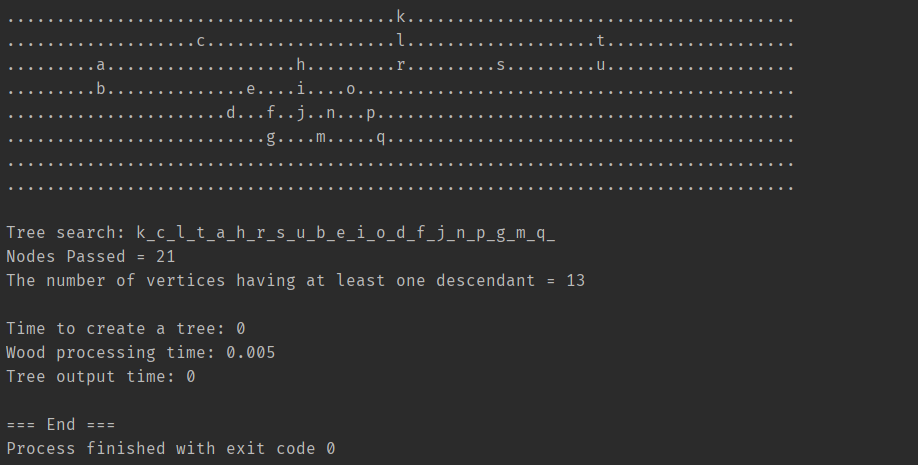
Способ обхода: В ширину

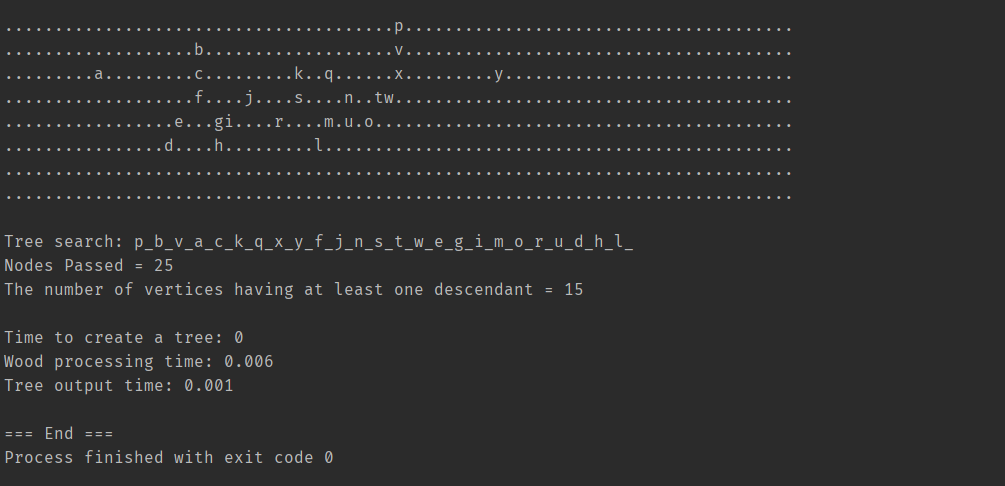
Что надо вычислить: Количество вершин, имеющих хотя бы одного потомка

**Обоснование выбора способа представления деревьев в памяти ЭВМ**

Был выбран прямой способ представления дерева в памяти с помощью разветвлённого списка, а обход производится с помощью очереди.

# Контрольные тесты



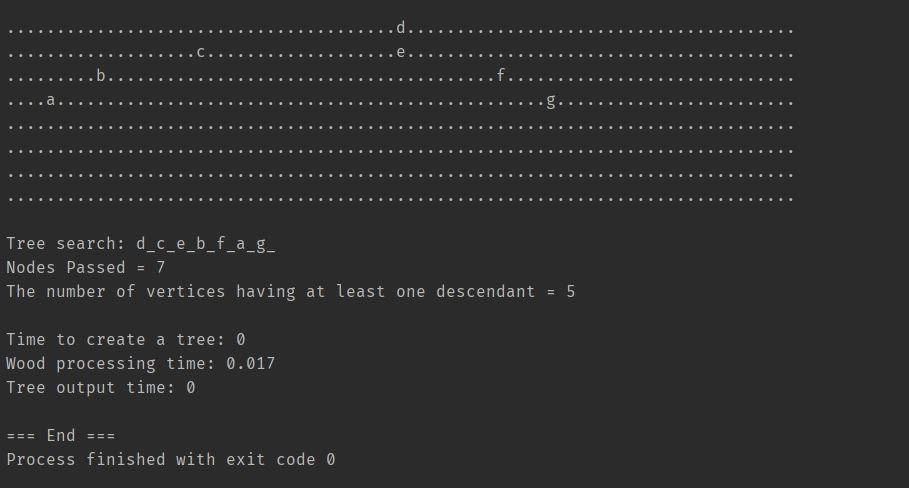




# Временная сложность

|  |  |  |
| --- | --- | --- |
| Функция обхода дерева | Ожидаемая временная сложность | Фактическая временная сложность |
| Создание | О(1) | О(1) |
| Обработка | О(n) | О(n) |
| Вывод | О(n) | О(n) |

**Последовательность битов, вводом которой можно построить дерево, полученное в результате генерации:**



1110100001100001

# Выводы:

Составили программу для работы с деревьями. Представили узел дерева и дерево в целом объектами соответствующих классов, а обходы дерева — функциями-членами этого класса. Использование классов дает возможность создавать сложные типы данных. Организация кода как объектов, значительно облегчает восприятие содержания программы, ее поддержку и тестируемость.

# Список использованных источников

1. Колинько П.Г. Алгоритмы и структуры данных. Часть 1: Методические указания к самостоятельной работе на ПЭВМ и курсовому проектированию. Вып. 1909. – СПб.: СПбГЭТУ «ЛЭТИ», 2019. – 69 с.

# Текст программы

#include <iostream>

#include <windows.h>

#include <time.h>

#include <chrono>

#include <queue>

using namespace std;

class Node {

char d;

Node\* lft;

Node\* mdl;

Node\* rgt;

public:

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

~Node() {

if (lft) delete lft;

if (mdl) delete mdl;

if (rgt) delete rgt;

}

friend class Tree;

};

class Tree

{

Node\* root;

char num{}, maxnum{};

int maxrow{}, offset{};

int count = 0;

int depth1 = 0;

char\*\*SCREEN{};

void clrscr()

{

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

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

}

Node\* MakeNode(int depth)

{

Node\* v = nullptr;

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

if (Y) {

v = new Node;

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

v->d = num++;

v->mdl = MakeNode(depth + 1);

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

}

return v;

}

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

{

int i = 1;

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

{

if (SCREEN[r - 1][c - 1] == '.')

{

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

}

else

{

while ((SCREEN[r - 1][c - i] != '.') || (SCREEN[r - 1][c + i] != '.'))

i++;

if (SCREEN[r - 1][c - i] != '.')

{

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

}

else

{

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

}

}

}

if (r < maxrow) {

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

if (v->mdl) OutNodes(v->mdl, r + 1, c);

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

}

}

Tree(const Tree& B)

{

root = new Node;

root->d = B.root->d;

root->lft = B.root->lft;

root->mdl = B.root->mdl;

root->mdl = B.root->mdl;

B.root->lft = nullptr;

B.root->mdl = nullptr;

B.root->rgt = nullptr;

}

Tree operator = (const Tree& B)

{

if (this != &B)

{

root = new Node;

root->d = B.root->d;

root->lft = B.root->lft;

root->mdl = B.root->mdl;

root->mdl = B.root->mdl;

B.root->lft = nullptr;

B.root->mdl = nullptr;

B.root->rgt = nullptr;

}

return \*this;

}

public:

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];

}

~Tree() {

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

delete[]SCREEN; delete root;

}

void MakeTree()

{

root = MakeNode(0);

}

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

Node\* Out\_root()

{

return root;

}

void result()

{

cout << "Nodes Passed = " << count << endl;

cout << "The number of vertices having at least one descendant = " << depth1 << endl;

}

std::string BFS(Node\* root1)

{

std::queue<Node\*> q;

std::string st;

q.push(root1);

int less\_then\_two\_vassals = 0;

while (!q.empty()) {

count++;

int current\_vassals = 0;

Node \*tmp = (Node\*) q.front();

q.pop();

st = st + tmp->d + "\_";

if (tmp->lft) {

current\_vassals++;

q.push(tmp->lft);

}

if (tmp->mdl) {

current\_vassals++;

q.push(tmp->mdl);

}

if (tmp->rgt) {

current\_vassals++;

q.push(tmp->rgt);

}

if (current\_vassals >= 1) {

depth1++;

}

}

cout<<st<<endl;

}

void OutTree()

{

clrscr();

OutNodes(root, 1, offset);

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

{

SCREEN[i][79] = 0;

cout << "\n" << SCREEN[i];

}

cout << "\n";

}

};

int main()

{

unsigned long begin\_t, end\_t, begin\_t1, end\_t1, begin\_t2, end\_t2;

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

srand(time(nullptr));

auto begin\_t4 = chrono::high\_resolution\_clock::now();

Tr.MakeTree();

auto end\_t4 = chrono::high\_resolution\_clock::now();

auto time = chrono::duration\_cast<chrono::nanoseconds>(end\_t4-begin\_t4);

if (Tr.exist()) {

begin\_t = clock();

Tr.OutTree();

end\_t = clock();

cout << "\n" << "Tree search: ";

begin\_t1 = clock();

cout<< Tr.BFS(Tr.Out\_root());

end\_t1 = clock();

Tr.result();

cout << endl;

cout << "Time to create a tree: " << time.count()<< endl;

cout << "Wood processing time: " << (float)(end\_t - begin\_t) / CLOCKS\_PER\_SEC << endl;

cout << "Tree output time: " << (float)(end\_t1 - begin\_t1) / CLOCKS\_PER\_SEC << endl;

}

else cout << "Tree is empty!";

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

return 0;

}