1. Цель работы: исследование алгоритмов для работы с двоичным (троичным) деревом.

2. Задание на работу с деревьями.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 17 | Двоичное | Обратная | В глубину | Количество левых листьев |

1. Обоснование выбора способа представления деревьев в памяти ЭВМ. Здесь следует сделать ссылку на выводы в отчётах по темам 1 и 2.

Дерево являет собой частный случай графа – совокупности множеств V (вершин) и E соответственно. Пусть каждой вершине дерева будет присвоена уникальная метка (тег) в виде строчной латинской буквы a-z. Таким образом, мы ограничиваем 0 ≤ |V| ≤ 26.

Понятно, что в таком дереве будет существовать |V|-1 ребер (теорема о числе ребер дерева). Однако, поскольку мы не знаем, какие именно вершины будут соединены друг с другом, попытка хранения дерева в виде показавших наилучшие временные результаты предыдущем тесте массива битов требовало бы рассмотрение всех возможных пар вершин, т.е. n(n-1) ребер. При этом:

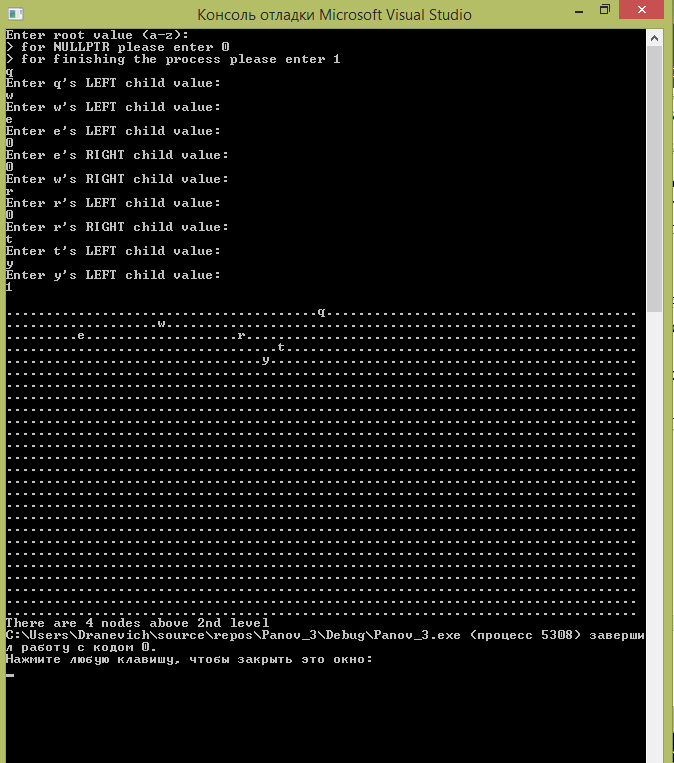
* в каждой строке матрицы смежности содержится не более трех единиц (родитель+двое потомков);
* реальное количество вершин может быть меньше мощности универсума, что, опять же, приводит к неоправданному расходу оперативной памяти.

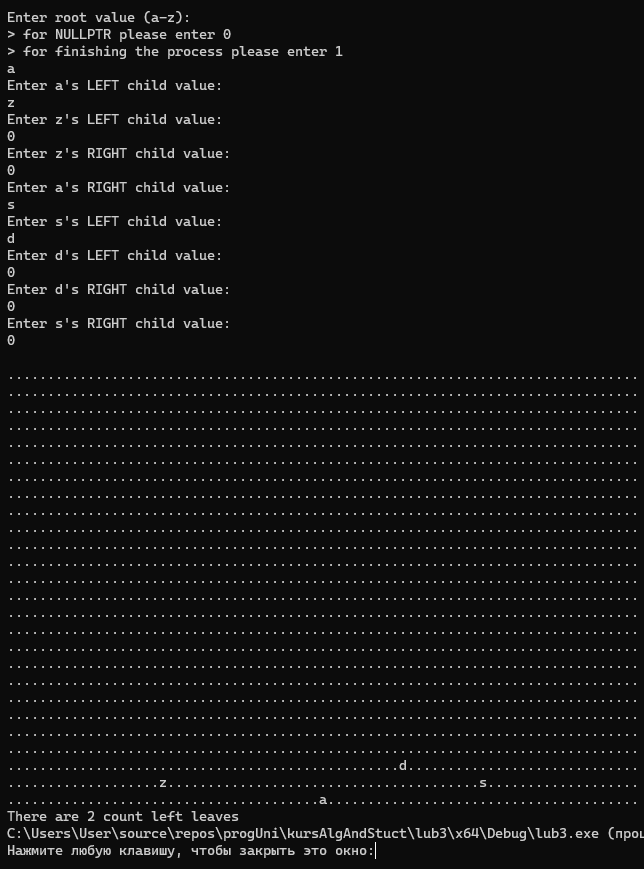
Те же недостатки делают не самым лучшим выбором и «обычный» массив.

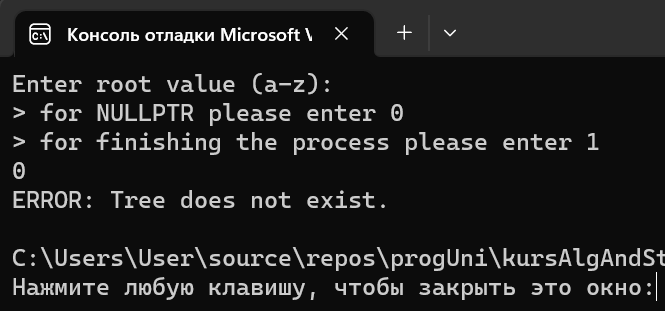
По этой причине наиболее обоснованной структурой данных мне кажется ветвящийся список, каждый элемент которого имеет по два указателя на своих потомков, если они существуют. Среди его достоинств имеет смысл упомянуть:

* Возможность динамического выделения памяти;
* Логическое соответствие структуре дерева как таковой;
* Удобство в использовании для рекурсивных алгоритмов.

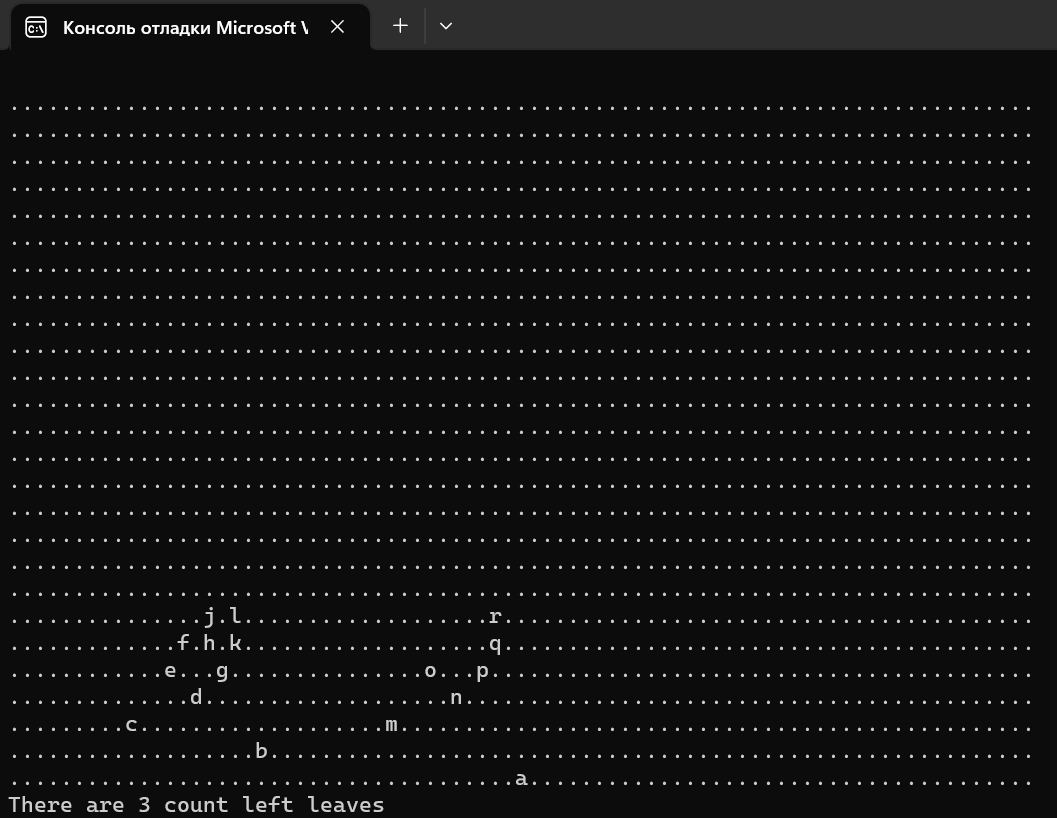
4. Тестовый пример: изображение дерева и порядок его ввода с клавиатуры.







5. Результаты прогона программы с генерацией случайного дерева (скриншоты).



6. Оценки временной сложности для каждой функции обхода дерева, использованной в программе: создание дерева, обработка, вывод.

|  |  |
| --- | --- |
| Tree::createNode() | O(N) |
| Tree::getUserInput() | O(N) |
| Tree::setTree() | O(N) |
| Tree::growTree() | O(N) |
| Tree::printTree() | O(1) |
| Tree::clear() | O(N2) |
| Tree::printNodes () | O(N) |
| DFS() | O(N(N-1)) |
| Tree::isEmpty() | O(1) |

7. Выводы о результатах испытания алгоритмов обхода деревьев.

Проведено исследование различных алгоритмов для обхода двоичного дерева.

При этом возможно три варианта обхода: прямой, внутренний и обратный, каждый из которых может иметь те или иные преимущества применительно к конкретной задаче.

К примеру, при вводе дерева с клавиатуры логично сначала присвоить значение родительскому узлу (тем более, что оно может оказаться nullptr), после чего приступить к обработке его потомков.

Использованные для решения поставленной задачи алгоритмы кажутся мне достаточно эффективными.

В реализации дерева использовалось два отдельных класса – один для узла и один для дерева в целом. Это удобно ввиду большого объема данных, характеризующего все дерево, и необходимости в использовании методов, обрабатывающих более одного узла.

8. Список использованной литературы.

1. Поздняков С. Н., Рыбин С. В. Дискретная математика: учебник для вузов. М.: Академия, 2008. – 448 с.
2. Хагерти Р. Дискретная математика для программистов. Изд. 2-е, испр. —

М.: Техносфера, 2012. — 400 с.;

1. Новиков Ф. А. Дискретная математика для программистов. — СПб.: Питер, 2000. — 304 с., ил.
2. Страуструп Б. Язык программирования С++. 2-е доп. изд. — М.: Биномпресс, 2001. – 1098 с.;

9. Приложение: исходный текст программы для работы с деревьями.

#include <iostream>

#include <conio.h>

#include <cstdlib>

#include <ctime>

using namespace std;

static const int WIDTH = 80;

static const int MAXPOWER = 26;

static bool unbrocken = 1;

template <class X> class STACK {

X\* x; int t;

public:

STACK(int maxPower) : x(new X[maxPower]), t(0) {}

bool isEmpty() const { return t == 0; }

void push(X el) { x[t++] = el; }

X pop() { return (t ? x[--t] : 0); }

};

template <class X> class QUEUE {

X\* x; int h, t, N;

public:

QUEUE(int maxQ) : h(0), t(0), N(maxQ), x(new X[maxQ + 1]) {}

bool isEmpty() const { return (h % N) == t; }

void push(X el) { x[t++] = el; t %= N; }

X pop() { h %= N; return x[h++]; }

};

class Node {

char tag;

Node\* l;

Node\* r;

int lvl;

public:

Node();

Node(char ch);

~Node();

char getTag() { return tag; }

friend class Tree;

};

Node::Node() : l(nullptr), r(nullptr), lvl(0), tag('0') {}

Node::Node(char ch) : l(nullptr), r(nullptr), lvl(0), tag(ch) {}

Node::~Node() {

if (l) delete l;

if (r) delete r;

}

class Tree {

Node\* root;

int curPower;

char\*\* SCREEN;

int curHeight;

int shift;

void clear();

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

Tree(const Tree&);

Tree(Tree&&);

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

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

public:

Tree();

~Tree();

void growTree();

void setTree();

void printTree();

Node\* createNode(int lvl);

Node\* getUserInput(int lvl);

int DFS();

bool isEmpty() { return (curPower == 0); }

};

int Tree::DFS() {

int count = 0;

STACK <Node\*> S(MAXPOWER);

S.push(root);

while (!S.isEmpty()) {

Node\* node = S.pop();

if (node->l) {

S.push(node->l);

if (node->l->l == nullptr && node->l->r == nullptr) ++count;

}

if (node->r) S.push(node->r);

}

return count;

}

Tree::Tree() :root(nullptr), SCREEN(new char\* [MAXPOWER]), curHeight(0), curPower(0), shift(WIDTH / 2) {

for (int i = 0; i < MAXPOWER; i++) SCREEN[i] = new char[WIDTH];

}

Tree::~Tree() {

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

delete[]SCREEN;

delete root;

}

void Tree::growTree() { root = createNode(0); }

void Tree::setTree() { root = getUserInput(0); }

Node\* Tree::createNode(int lvl) {

Node\* node = nullptr;

if (curPower < 26 && curHeight < MAXPOWER && (lvl < rand() % 11 + 1)) {

node = new Node;

node->lvl = lvl;

node->tag = ('a' + curPower++);

node->l = createNode(lvl + 1);

node->r = createNode(lvl + 1);

}

return node;

}

Node\* Tree::getUserInput(int lvl) {

Node\* node = nullptr;

char ch = '0';

if (curPower < 1) {

cout << "Enter root value (a-z):" << endl;

cout << "> for NULLPTR please enter 0" << endl;

cout << "> for finishing the process please enter 1" << endl;

}

cin >> ch;

if (ch == '1') { unbrocken = 0; }

if (isalpha(ch) && curHeight < MAXPOWER && curPower < MAXPOWER && unbrocken) {

node = new Node;

node->lvl = lvl;

node->tag = (ch); curPower++;

if (curPower == MAXPOWER || (!unbrocken)) return node;

cout << "Enter " << ch << "'s LEFT child value:" << endl;

node->l = getUserInput(lvl + 1);

if (curPower == MAXPOWER || (!unbrocken)) return node;

cout << "Enter " << ch << "'s RIGHT child value:" << endl;

node->r = getUserInput(lvl + 1);

}

return node;

}

void Tree::printTree() {

if (root == nullptr) {

cout << "ERROR: Tree does not exist." << endl;

return;

}

clear();

printNodes(root, 1, shift);

for (int i = MAXPOWER-1; i >= 0; --i) {

SCREEN[i][WIDTH - 1] = 0;

cout << endl;

cout << SCREEN[i];

}

cout << endl;

}

void Tree::printNodes(Node\* node, int x, int s) {

if (x && s && (s < WIDTH)) SCREEN[x - 1][s - 1] = node->tag;

if (x < MAXPOWER) {

if (node->l) printNodes(node->l, x + 1, s - (shift >> x));

if (node->r) printNodes(node->r, x + 1, s + (shift >> x));

}

}

void Tree::clear() {

for (int i = 0; i < MAXPOWER; i++) memset(SCREEN[i], '.', WIDTH);

}

int main()

{

srand(time(0));

Tree tree;

tree.growTree();

//tree.setTree();

tree.printTree();

if (!tree.isEmpty()) {

cout << "There are ";

cout << tree.DFS() << " count left leaves";

}

}