Министерство образования и науки Российской Федерации  
Федеральное государственное автономное образовательное учреждение высшего образования

«ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

(ПНИПУ)

Электротехнический факультет

Кафедра Информационные технологии и автоматизированные системы

Лабораторная работа по

«Теория алгоритмов и структуры данных»

вариант № 15

за 2 семестр

Выполнил студент:

Тимолянов Григорий Констнатинович

Группа РИС-21-1бз

Проверила:

Доцент кафедры ИТАС

Полякова Ольга Андреевна

Пермь 2024

Лабораторная работа №8

**Бинарные деревья**

# Цель: получить практические навыки работы с бинарными деревьями.

###### **Постановка задачи**

###### Тип информационного поля char. Найти количество элементов дерева, начинающихся с заданного символа.

**Решение:**

Класс Search

Search.h

#pragma once

#include<vector>

class SearchTree

{

char data; // Данные типа Т

SearchTree\* left; // Указатель на узел слева

SearchTree\* right; // Указатель на узел справа

SearchTree\* parent; // Указатель на предка

public:

int node\_x;

int node\_y;

int text\_x;

int text\_y;

// Установить координаты для данного узла при рисовании

void drawTree(int, char\*\*, int, int, int, int); //рисовать дерево

//-------------Constructors-------------//

SearchTree(char); // Конструктор

~SearchTree(); // Деструктор

SearchTree(const SearchTree&); // Конструктор копирования

//

int getHeight(); // Получить высоту дерева (считает с текущего узла по направлению к листьям)

char getData() { return data; }; // Получить данные с узла

int getAmountOfNodes(); // Получить количество элементов дерева

SearchTree\* getLeft() { return left; } // Получить левый узел

SearchTree\* getRight() { return right; } // Получить правый узел

SearchTree\* getParent() { return parent; } // Получить родителя

int getLevel (SearchTree\* tree);

SearchTree\* getNodeByCoords (int x, int y, int R);

void setCoordsForNode( int window\_width, int window\_height, int shift, int ree\_width, int tree\_height, int x, int y, int R);

void setCoordsForText(int k, int shift); // Установить координаты для текста текущего узла при рисовании

SearchTree\* copyTree(); // Скопировать дерево

SearchTree\* replaceNULLforEmpty(); //Если дерево неполное, сделать его полным (недостающие узлы приобретут данные NULL)

SearchTree\* replace\_help(SearchTree\* node, int h);

void insertLeft(char); // Вставить узел слева

void insertRight(char); // Вставить узел справа

void addLeftTree(SearchTree\* tree) { left = tree; } // Добавить поддерево слева

void addRightTree(SearchTree\* tree) { right = tree; } // Добавить поддерево справа

void deleteSearchTree() { delete this; } // Удалить дерево

void deleteNode(char); // Удалить узел

void deleteLeft(); // Удалить поддерево слева

void deleteRight(); // Удалить поддерево справа

SearchTree\* ejectLeft(); // Извлечь поддерево слева

SearchTree\* ejectRight(); // Извлечь поддерево справа

void printSearchTree(int); // Горизонтальная печать дерева

void inOrder(SearchTree\*); // Симметричный обход дерева

void preOrder(SearchTree\*); // Вывести значения элементов (сверху вниз)

void postOrder(SearchTree\*); // Вывести значения элементов (cниз вверх)

void printTree(int); // Вывести значения элементов (горизонтально)

void printVTree(int); // Печать дерева вертикально

void setData(char dt) { data = dt; } // Установить данные для узла

SearchTree\* next(); // Найти следующий элемент

SearchTree\* prev(); // Найти предыдущий элемент

void insertNode(char); // Вставить узел

SearchTree\* findElement(char); // Найти элемент

SearchTree\* findMax(); // Найти максимум

SearchTree\* findMin(); // Найти минимум

size\_t number\_by\_symbol(const char&);

static SearchTree\* balancedTree(int n);// Построить идеально сбалансированное дерево поиска по данному количеству элементов

void findElement\_insertLeft(SearchTree\*, char, char); // Найти элемент и добавить к нему слева узел

void findElement\_insertRight(SearchTree\*, char, char); // Найти элемент и добавить к нему справа узел

void add(char dt); // Добавить элемент в дерево

std::vector<char> copy\_to\_vector(); // Копировать дерево в вектор

SearchTree\* ideal\_to\_search( std::vector<char>, int, int); // Преобразовать идеально сбалансированное дерево в дерево поиска

};

Search.cpp

#include<iostream>

#include<vector>

#include<fstream>

#include<queue>

#include"draw.h"

#include<algorithm>

#include"Search.h"

void SearchTree::insertLeft(char dt) //вставка узла слева

{

SearchTree\* node = new SearchTree(dt);

if (this->left != nullptr)

this->left->parent = node;// Сделать так, чтобы его предком был новый узел

node->left = this->left; //Теперь у нового узла левый потомок – это левый потомок вызывающего узла

this->left = node; //А вызывающий узел теперь имеет левого потомка – это новый узел

node->parent = this; //Указываем, что у нового узла теперь предком является текущий узел

}

void SearchTree::insertRight(char dt) //вставка узла справа

{

SearchTree\* node = new SearchTree(dt);

if (this->right != nullptr)

this->right->parent = node;

node->right = this->right;

this->right = node;

node->parent = this;

}

int SearchTree::getHeight()

{

int h1 = 0, h2 = 0, hadd = 0;

if (this == nullptr)

return 0;

if (this->left != nullptr)

h1 = this->left->getHeight();

if (this->right != nullptr)

h2 = this->right->getHeight();

if (h1 >= h2 && this->data)

return h1 + 1;

else if (this->data)

return h2 + 1;

else

return 0;

}

int SearchTree::getAmountOfNodes()

{

if (this == nullptr)

return 0;

if ((this->left == nullptr) && (this->right == nullptr))

return 1;

int l = 0;

int r = 0;

if (this->left != nullptr)

l = this->left->getAmountOfNodes();

if (this->right != nullptr)

r = this->right->getAmountOfNodes();

return (l + r + 1);

}

int SearchTree::getLevel(SearchTree\* tree)

{

return tree->getParent() == nullptr ? 0 : getLevel(tree->getParent()) + 1;

}

SearchTree\* SearchTree::replaceNULLforEmpty()

{

SearchTree\* node = this->copyTree();

int h = node->getHeight();

node = replace\_help(node, h);

return node;

}

SearchTree\* SearchTree::copyTree()

{

SearchTree\* tree = new SearchTree(this->data);

if (this->parent != nullptr)

tree->parent = this->parent;

if (this->left != nullptr)

tree->left = this->left->copyTree();

if (this->right != nullptr)

tree->right = this->right->copyTree();

return tree;

}

SearchTree\* SearchTree::replace\_help(SearchTree\* node, int h)

{

int curLevel = getLevel(node);

if ((node->getLeft() == nullptr) && (curLevel != (h - 1)))

node->insertLeft(NULL);

if ((node->getRight() == nullptr) && (curLevel != h - 1))

node->insertRight(NULL);

if (node->getLeft() != nullptr)

node->addLeftTree(replace\_help(node->getLeft(), h));

if (node->getRight() != nullptr)

node->addRightTree(replace\_help(node->getRight(), h));

return node;

}

SearchTree\* SearchTree::findElement(char dt)

{

if ((this == nullptr) || (dt == this->data))

return this;

if (dt < this->data)

return this->left->findElement(dt);

else

return this->right->findElement(dt);

}

SearchTree\* SearchTree::findMin()

{

if (this->left == nullptr)

return this;

return this->left->findMin();

}

SearchTree\* SearchTree::findMax()

{

if (this->right == nullptr)

return this;

return this->right->findMax();

}

SearchTree\* SearchTree::next()

{

SearchTree\* tree = this;

if (tree->right != nullptr)

return tree->right->findMin();

SearchTree\* t = tree->parent;

while ((t != nullptr) && (tree == t->right))

{

tree = t;

t = t->parent;

}

return t;

}

SearchTree\* SearchTree::prev()

{

SearchTree\* tree = this;

if (tree->left != nullptr)

return tree->left->findMax();

SearchTree\* t = tree->parent;

while ((t != nullptr) && (tree == t->left))

{

tree = t;

t = t->parent;

}

return t;

}

void SearchTree::insertNode(char dt)

{

SearchTree\* tree = this; //создаётся элемент равный текущему

while (tree != NULL) //проверяется существование элемента

{

if (dt >= tree->data) //проверяется значение элемента

{

if (tree->right != NULL) //проверяется существование элемента справа

{

tree = tree->right; //переход к элементу справа

}

else

{

SearchTree\* temp = new SearchTree(dt); //создаётся новый элемент

temp->parent = tree; //задаётся новому элементу значение элемента предка

tree->right = temp; //задаётся элементу значение элемента справа

break;

}

}

else if (dt < tree->data) //проверяется значение элемента

{

if (tree->left != NULL) //проверяется существование элемента слева

{

tree = tree->left; //переход к элементу слева

}

else

{

SearchTree\* temp = new SearchTree(dt); //создаётся новый элемент

temp->parent = tree; //задаётся новому элементу значение элемента предка

tree->left = temp; //задаётся элементу значение элемента слева

break;

}

}

}

}

void SearchTree::deleteNode(char dt)

{

SearchTree\* e = this->findElement(dt);

SearchTree\* p = e->parent; //предок удаляемого элемента

if ((e->left == nullptr) && (e->right == nullptr)) // Первый случай: удаляемый элемент не имеет потомков

{

if (p->left == e) p->left = nullptr;

if (p->right == e) p->right = nullptr;

}

else if ((e->left == NULL) || (e->right == NULL)) // Второй случай: удаляемый элемент имеет одного потомка

{

if (e->left == NULL)

{

if (p->left == e) p->left = e->right;

else p->right = e->right;

e->right->parent = p;

}

else

{

if (p->left == e) p->left = e->left;

else p->right = e->left;

e->left->parent = p;

}

}

// Третий случай: удаляемый элемент имеет двух потомков

else

{

SearchTree\* s = e->next();

// Следующий элемент за удаляемым

e->data = s->data;

if (s->parent->left == s)

{

s->parent->left = s->right;

if (s->right != NULL)

s->right->parent = s->parent;

}

else

{

s->parent->right = s->right;

if (s->right != NULL)

s->right->parent = s->parent;

}

}

}

inline SearchTree::SearchTree(char dt)

{

data = dt; //задаётся значение элемента

left = NULL; //задаётся значение элемента слева

right = NULL; //задаётся значение элемента справа

parent = NULL; //задаётся значение элемента предка

}

SearchTree::~SearchTree()

{

delete this->left; //удаляется элемент слева

delete this->right; //удаляется элемент справа

}

inline SearchTree::SearchTree(const SearchTree& temp) :data(temp.data), left(temp.left), right(temp.right), parent(temp.parent)

{

}

void SearchTree::printSearchTree(int level)

{

if (this != NULL) //проверяется существование элемента

{

this->right->printSearchTree(level + 1); //переход к элементу справа

for (int i = 1; i < level; i++) //выводятся пробелы

{

std::cout << " ";

}

std::cout << this->data << std::endl; //выводится значение элемента

this->left->printSearchTree(level + 1); //переход к элементу слева

}

}

void SearchTree::preOrder(SearchTree\* temp)

{

if (temp != NULL) //проверяется существование элемента

{

std::cout << temp->data << " "; //выводится значение элемента

preOrder(temp->left); //переход к элементу слева

preOrder(temp->right); //переход к элементу справа

}

}

void SearchTree::inOrder(SearchTree\* temp)

{

if (temp != NULL) //проверяется существование элемента

{

inOrder(temp->left); //переход к элементу слева

std::cout << temp->data << " "; //выводится значение элемента

inOrder(temp->right); //переход к элементу справа

}

}

void SearchTree::postOrder(SearchTree\* temp)

{

if (temp != NULL) //проверяется существование элемента

{

postOrder(temp->left); //переход к элементу слева

postOrder(temp->right); //переход к элементу справа

std::cout << temp->data << " "; //выводится значение элемента

}

}

void SearchTree::printTree(int level)

{

if (this != NULL) //проверяется существование элемента

{

this->right->printTree(level + 1); //переход к элементу справа

for (int i = 1; i < level; i++) //выводятся пробелы

{

std::cout << " ";

}

std::cout << this->data << std::endl; //выводится значение элемента

this->left->printTree(level + 1); //переход к элементу слева

}

}

void SearchTree::printVTree(int k)

{

int height = this->getHeight(); //Максимальное число листов на нижнем уровне(нумерация с нуля)

int maxLeafs = pow(2, height - 1); //Минимальная ширина дерева для печати (не конечная, но необходимая)

int width = (2 \* maxLeafs - 1);

int curLevel = 0; // Номер строки (на выводе)

int index = 0; // Номер элемента в строке (нумерация с нуля)

int factSpaces = getPos(index, width, curLevel, height - 1); //Позиция корня (число пробелов перед ним)

pos node;

std::vector<SearchTree\*> V;

std::vector<pos> Vi;

SearchTree\* t = this->copyTree();

t = t->replaceNULLforEmpty();

SearchTree\* p = t;

V.push\_back(p);

node.col = factSpaces;

node.str = curLevel;

Vi.push\_back(node);

for (int i = 0; i < t->getAmountOfNodes(); i++)

{

if (pow(2, curLevel) <= index + 1)

{

index = 0;

curLevel++;

}

if (V.at(i)->left != nullptr)

{

V.push\_back(V.at(i)->left);

factSpaces = getPos(index, width, curLevel, height - 1);

node.col = factSpaces;

node.str = curLevel;

Vi.push\_back(node);

index++;

}

if (V.at(i)->right != nullptr)

{

V.push\_back(V.at(i)->right);

factSpaces = getPos(index, width, curLevel, height - 1);

node.col = factSpaces;

node.str = curLevel;

Vi.push\_back(node);

index++;

}

}

/\* Редактируем позиции в строчках (теперь они обозначают количество пробелов перед данным символом начиная с предыдущего символа):

до этого эти значения представляли собой координаты (как х) \*/

for (int i = V.size() - 1; i >= 0; i--)

{

if (i != 0)

{

if (Vi.at(i - 1).str == Vi.at(i).str) Vi.at(i).col = Vi.at(i).col - Vi.at(i - 1).col - 1;

}

}

int flag = 0; // Следит за тем, что y меняется

for (int i = 0; i < V.size(); i++)

{

node = Vi.at(i);

curLevel = node.str; //Переход на новую строчку будет, когда y1 станет меньше y (слежка за изменением y)

if (flag < curLevel)

{

flag = curLevel;

std::cout << std::endl;

}

factSpaces = node.col;

int realSpaces = k \* factSpaces;

for (int j = 0; j < realSpaces; j++)

std::cout << " ";

if (V.at(i)->getData() != NULL)

std::cout << V.at(i)->getData();

else for (int j = 0; j < k; j++)

std::cout << " ";

}

std::cout << std::endl;

}

size\_t SearchTree::number\_by\_symbol(const char& value)

{

size\_t n\_l{}, n\_r{};

if (this == nullptr)

{

return 0;

}

if (this->left != nullptr)

{

n\_l = this->left->number\_by\_symbol(value);

}

if (this->right != nullptr)

{

n\_r = this->right->number\_by\_symbol(value);

}

if (this->data == value)

{

return (n\_l + n\_r + 1);

}

return (n\_l + n\_r);

}

SearchTree\* SearchTree::balancedTree(int n)

{

if (!n)

return nullptr;

std::cout << "data=";

char dt;

std::cin >> dt;

int nLeft = n / 2;

int nRight = n - n / 2 - 1;

SearchTree\* tree = new SearchTree(dt);

tree->addLeftTree(balancedTree(nLeft));

tree->addRightTree(balancedTree(nRight));

return tree;

}

void SearchTree::findElement\_insertLeft(SearchTree\* node, char dt1, char dt2)

{

if (node != nullptr)

{

findElement\_insertLeft(node->getLeft(), dt1, dt2);

if (dt1 == node->getData()) node->insertLeft(dt2);

findElement\_insertLeft(node->getRight(), dt1, dt2);

}

}

void SearchTree::findElement\_insertRight(SearchTree\* node, char dt1, char dt2)

{

if (node != nullptr)

{

findElement\_insertRight(node->getLeft(), dt1, dt2);

if (dt1 == node->getData()) node->insertRight(dt2);

findElement\_insertRight(node->getRight(), dt1, dt2);

}

}

void SearchTree::add(char dt)

{

if (this == nullptr)

return;

if (this->data <= dt)

{

if (this->right == nullptr)

{

this->insertRight(dt);

}

else

{

this->right->add(dt);

}

}

if (this->data > dt)

{

if (this->left == nullptr)

{

this->insertLeft(dt);

}

else

{

this->left->add(dt);

}

}

}

SearchTree\* SearchTree::ejectLeft() // Извлечение

{

if (this->left != nullptr)

{

SearchTree\* temp = this->left;

this->left = nullptr;

return temp;

}

return nullptr;

}

void SearchTree::deleteLeft() // Удаление

{

SearchTree\* temp = this->ejectLeft();

delete temp;

}

SearchTree\* SearchTree::ejectRight()

{

if (this->right != nullptr)

{

SearchTree\* temp = this->right;

this->right = nullptr;

return temp;

}

return nullptr;

}

void SearchTree::deleteRight()

{

SearchTree\* temp = this->ejectRight();

delete temp;

}

std::vector<char> SearchTree::copy\_to\_vector()

{

static std::vector<char> v;

if (this != nullptr)

{

v.push\_back(this->data);

this->left->copy\_to\_vector();

this->right->copy\_to\_vector();

}

return v;

}

SearchTree\* SearchTree::ideal\_to\_search(std::vector<char> v, int start, int end)

{

if (!v.size() || start > end)

{

return nullptr;

}

int average = (start + end) / 2;

SearchTree\* tree = new SearchTree(v[average]);

tree->right = ideal\_to\_search(v, average + 1, end);

tree->left = ideal\_to\_search(v, start, average - 1);

return tree;

}

draw.h

#pragma once

#include"Search.h"

void initWindow(int, char\*\*);

void display(void);

static void reshape(int, int);

static void drawLine(int, int, int, int);

static void drawFillCircle(int, int, int);

static void drawBlueCircle(int, int, int);

template<class T>

static void drawText(T , void\*, int, int, int, int);

static void mouseMove(int, int);

SearchTree\* get\_help(SearchTree\*, int, int, int);

int getPos(int, int, int, int);

struct pos

{

int col; // Столбец (х)

int str; // Строка (у)

};

struct SGlutContextStruct

{

/\* window\_width – ширина окна;

\* window\_height – высота окна;

\* shift – отступ от краев (решено сделать его одинаковым с двух сторон);

\* k - коэффициент ширины данных;

\* R – радиус круга;

\* x, y - координаты чего-либо;

\* state - переменная состояния при работе с мышью

\*/

void\* tree;

int window\_width, window\_height, shift, k, R, x, y, state;

};

Draw.cpp

#include "draw.h"

#include <math.h>

#include<string>

#include<vector>

#include<gl/glut.h>

SGlutContextStruct glutContext;

int getPos(int index, int width, int curLevel, int maxLevel)

{

int x1 = 0;

int x2 = pow(2, curLevel) - 1;

int y1 = width / pow(2, curLevel + 1);

int y2 = width - pow(2, maxLevel - curLevel);

if (x1 == x2)

return y1;

double k = (y2 - y1) / (x2 - x1);

double m = -x1 \* k + y1;

int y = (int)(k \* index + m);

return y;

}

void SearchTree::setCoordsForNode(int window\_width, int window\_height, int shift, int tree\_width, int tree\_height, int x, int y, int R)

{

// Это условие не выполняется, когда дерево состоит из одного элемента

if (tree\_width != tree\_height)

{

int k\_x = (window\_width - 2 \* (shift + R)) / (tree\_width - 1); // Коэффициент пропорциональности по оси Ох

int k\_y = (window\_height - 2 \* (shift + R)) / (tree\_height - 1); // Коэффициент пропорциональности по оси Оy

node\_x = k\_x \* x + shift + R; // x-координата узла

node\_y = window\_height - k\_y \* y - shift - R; // у-координата узла

}

else

{

node\_x = window\_width / 2; // x-координата узла

node\_y = window\_height / 2; // у-координата узла

}

}

void SearchTree::setCoordsForText(int k, int R)

{

text\_x = node\_x - 3 \* R / 4; // х-координата первого символа текста

text\_y = node\_y - 3 \* R / (4 \* k); // у-координата первого символа текста

}

SearchTree\* SearchTree::getNodeByCoords(int x, int y, int R)

{

SearchTree\* node = this;

node = get\_help(node, x, y, R);

return node;

}

SearchTree\* get\_help(SearchTree\* node, int x, int y, int R)

{

if (pow(x - node->node\_x, 2) + pow(y - node->node\_y, 2) <= pow(R, 2))

return node;

SearchTree\* temp = NULL;

if (node->getLeft() != NULL)

temp = get\_help(node->getLeft(), x, y, R);

if (temp != NULL)

return temp;

if (node->getRight() != NULL)

temp = get\_help(node->getRight(), x, y, R);

return temp;

}

//

void initWindow(int argc, char\*\* argv)

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(glutContext.window\_width, glutContext.window\_height);

glutCreateWindow("DrawTree");

glutDisplayFunc(display);

glutReshapeFunc(reshape);

glutPassiveMotionFunc(mouseMove);

glutMainLoop();

}

static void reshape(int w, int h)

{

glViewport(0, 0, (GLsizei) w, (GLsizei) h);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, (GLsizei) w, 0, (GLsizei) h);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glutContext.window\_width = w;

glutContext.window\_height = h;

glutPostRedisplay();

}

static void drawLine(int x1, int y1, int x2, int y2)

{

glBegin(GL\_LINES);

glColor3f(0.0, 0.0, 0.0);

glVertex2i(x1, y1);

glVertex2i(x2, y2);

glEnd();

}

static void drawFillCircle(int x, int y, int R)

{

glColor3f(1.0, 1.0, 1.0);

float x1, y1;

glBegin(GL\_POINTS);

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

{

for (int t = 0; t <= 360; t++)

{

x1 = i \* sin(t) + x;

y1 = i \* cos(t) + y;

glVertex2f(x1, y1);

}

}

glEnd();

glColor3f(0.0, 0.0, 0.0);

glBegin(GL\_POINTS);

for (int i = R - 1; i <= R; i++)

{

for (int t = 0; t <= 360; t++)

{

x1 = R \* sin(t) + x;

y1 = R \* cos(t) + y;

glVertex2f(x1, y1);

}

}

glEnd();

}

static void drawBlueCircle(int x, int y, int R)

{

glColor3f(0.0, 0.0, 1.0);

float x1, y1;

glBegin(GL\_POINTS);

for (int i = R - 1; i <= R; i++)

{

for (int t = 0; t <= 360; t++)

{

x1 = R \* sin(t) + x;

y1 = R \* cos(t) + y;

glVertex2f(x1, y1);

}

}

glEnd();

}

template <class T>

static void drawText(T text, void\* font, int text\_x, int text\_y, int R, int k)

{

glColor3f(0.0, 0.0, 0.0);

glPushMatrix();

glTranslatef((text\_x + text\_x \* 0.03), text\_y, 0.0);

std::string s;

if (sizeof(text) == 1)

s = text;

else

s = std::to\_string(text).substr(0, 3);

char\* s1 = new char[s.size() + 1];

for (int i = 0; i < s.size(); i++)

s1[i] = s.at(i);

s1[s.size()] = 0;

char\* c;

int max\_char\_width = 0;

int char\_width = 0;

for (c = s1; \*c != '\0'; c++)

{

char\_width = glutStrokeWidth(font, \*c);

if (max\_char\_width < char\_width)

max\_char\_width = char\_width;

}

float expand\_x = (float)1.5 \* R / (float)(k \* max\_char\_width);

float expand\_y = (float)1.5 \* R / (float)(k \* 100);

glScalef(expand\_x, expand\_y, 1.0);

for (c = s1; \*c != '\0'; c++)

glutStrokeCharacter(font, \*c);

glPopMatrix();

}

void display(void)

{

SearchTree\* tree = (SearchTree\*)glutContext.tree;

int k = glutContext.k;

int window\_width = glutContext.window\_width;

int window\_height = glutContext.window\_height;

int shift = glutContext.shift;

int height = tree->getHeight();

int maxLeafs = pow(2, height - 1); // Максимальное число листов на нижнем уровне (нумерация с нуля)

//int maxLeafs = 10;

int width = 2 \* maxLeafs - 1; // Минимальная ширина дерева для печати (не конечная, но необходимая)

int curLevel = 0; // Номер строки (на выводе)

int index = 0; //номер элемента в строке (нумерация с нуля)

int factSpaces = getPos(index, width, curLevel, height - 1); // Позиция корня (число пробелов перед ним)

pos node;

std::vector<SearchTree\*> V;

std::vector<pos> Vi;

int R; // Радиус круга

R = (window\_width - 2 \* shift) / (2 \* width);

if (2 \* R \* height > (window\_height - 2 \* shift)) R = (window\_height - 2 \* shift) / (2 \* height);

glutContext.R = R;

// Установили координаты корня при рисовании

tree->setCoordsForNode(window\_width, window\_height,

shift, width,

height, factSpaces, curLevel, R);

V.push\_back(tree);

node.col = factSpaces;

node.str = curLevel;

Vi.push\_back(node);

glClearColor(1.0, 1.0, 1.0, 1.0);

glClear(GL\_COLOR\_BUFFER\_BIT);

glLineWidth(2);

glEnable(GL\_POINT\_SMOOTH);

for (int i = 0; i < tree->getAmountOfNodes(); i++)

{

if (pow(2, curLevel) <= index + 1)

{

index = 0;

curLevel++;

}

if (V.at(i)->getLeft() != NULL)

{

V.push\_back(V.at(i)->getLeft());

factSpaces = getPos(index, width, curLevel, height - 1);

node.col = factSpaces;

node.str = curLevel;

Vi.push\_back(node);

index++;

V.at(i)->getLeft()->setCoordsForNode(window\_width,

window\_height,

shift, width, height, factSpaces, curLevel, R);

if (V.at(i)->getLeft()->getData() != NULL)

{

int x1 = V.at(i)->node\_x;

int y1 = V.at(i)->node\_y;

int x2 = V.at(i)->getLeft()->node\_x;

int y2 = V.at(i)->getLeft()->node\_y;

drawLine(x1, y1, x2, y2);

}

}

if (V.at(i)->getRight() != NULL)

{

V.push\_back(V.at(i)->getRight());

factSpaces = getPos(index, width, curLevel, height - 1);

node.col = factSpaces;

node.str = curLevel;

Vi.push\_back(node);

index++;

V.at(i)->getRight()->setCoordsForNode(window\_width,

window\_height, shift, width, height, factSpaces,

curLevel, R);

if (V.at(i)->getRight()->getData() != NULL)

{

int x1 = V.at(i)->node\_x;

int y1 = V.at(i)->node\_y;

int x2 = V.at(i)->getRight()->node\_x;

int y2 = V.at(i)->getRight()->node\_y;

drawLine(x1, y1, x2, y2);

}

}

if (V.at(i)->getData() != NULL)

{

if (glutContext.state == 0)

{

drawFillCircle(V.at(i)->node\_x, V.at(i)->node\_y, R);

}

else

{

drawFillCircle(V.at(i)->node\_x, V.at(i)->node\_y, R);

if ((tree->getNodeByCoords(glutContext.x, glutContext.y, R)->node\_x == V.at(i)->node\_x) & (tree->getNodeByCoords(glutContext.x, glutContext.y, R)->node\_y == V.at(i)->node\_y))

drawBlueCircle(V.at(i)->node\_x, V.at(i)->node\_y, R);

}

V.at(i)->setCoordsForText(k, R);

drawText(V.at(i)->getData(), GLUT\_STROKE\_ROMAN, V.at(i)->text\_x, V.at(i)->text\_y, R, k);

}

}

glutSwapBuffers();

glDisable(GL\_POINT\_SMOOTH);

}

static void mouseMove(int x, int y)

{

SearchTree\* tree = (SearchTree\*)glutContext.tree;

int R = glutContext.R;

SearchTree\* node = tree->getNodeByCoords(x,

glutContext.window\_height - y, R);

if (node != NULL)

{

glutContext.x = x;

glutContext.y = glutContext.window\_height - y;

glutContext.state = 1;

glutPostRedisplay();

}

else

{

glutContext.state = 0;

glutPostRedisplay();

}

}

void SearchTree::drawTree(int argc, char\*\* argv, int window\_width, int window\_height, int shift, int k)

{

SearchTree\* temp = this->copyTree();

temp = temp->replaceNULLforEmpty();

glutContext.tree = temp;

glutContext.window\_width = window\_width;

glutContext.window\_height = window\_height;

glutContext.shift = shift;

glutContext.k = k;

initWindow(argc, argv);

}

//

functions.h

#pragma once

#include<iostream>

#include"Search.h"

#include<vector>

#include<conio.h>

short Checking\_For\_Valid\_Input(); //проверка ввода для типа данных short

char Checking\_For\_Valid\_Input\_Char(); //проверка ввода для типа данных char

//------functions-for-tree------

SearchTree\* create\_ideal\_tree(); //создать идеальное дерево

SearchTree\* create\_array\_search\_tree(); //создать дерево поиска из массива

void show\_tree(SearchTree\*); //печать дерева поиска

void delete\_left(SearchTree\*); //удалить поддерево слева

void delete\_right(SearchTree\*); //удалить поддерево справа

void number\_of\_elements(SearchTree\*); //кол-во элементов дерева

void height(SearchTree\*); //высота дерева

SearchTree\* create\_ideal\_search(SearchTree\*); // Преобразовать идеальное дерево в дерево поиска

void print\_OpenGL(SearchTree\*, int, char\*\*); // Печать дерева поиска в OpenGL

void number\_by\_symbol(SearchTree\* tree);

//------main-menu------

void Selection(int argc, char\*\* argv);

functions.cpp

#include<iostream>

#include"Search.h"

#include<vector>

#include<conio.h>

#include"functions.h"

#include<algorithm>

short Checking\_For\_Valid\_Input() //проверка ввода для типа данных short

{

short value;

do

{

if (!std::cin)

{

std::cout << "Syntax error" << std::endl;

std::cin.clear();

std::cin.ignore(1024, '\n');

}

std::cin >> value;

} while (!std::cin);

return value;

}

char Checking\_For\_Valid\_Input\_Char() //проверка ввода для типа данных char

{

char symbol;

do

{

if (!std::cin)

{

std::cout << "Syntax error" << std::endl;

std::cin.clear();

std::cin.ignore(1024, '\n');

}

std::cin >> symbol;

} while (!std::cin);

return symbol;

}

//------functions-for-tree------

SearchTree\* create\_ideal\_tree() //создать идеально сбалансированное дерево

{

int n;

std::cout << "Введите количество Элементов: ";

std::cin >> n;

std::cout << std::endl;

SearchTree\* tree = SearchTree::balancedTree(n);

std::cout << std::endl;

return tree;

}

SearchTree\* create\_array\_search\_tree() //Создать дерево поиска из массива

{

std::vector<char> arr = { 'A', 'B', 'D', 'G', 'E', 'H', 'C', 'F', 'I', 'J' }; //Создать контейнер значений элементов дерева

SearchTree\* tree = new SearchTree(arr.at(0)); //Создать первый элемент дерева

for (int i = 1; i < arr.size(); i++)

{

tree->insertNode(arr.at(i)); //Добавить элемент с текущим номером в дерево

}

return tree;

}

void show\_tree(SearchTree\* tree) //печать дерева поиска

{

std::cout << "Прямой обход: ";

tree->preOrder(tree); //Вывод значений элементов (сверху вниз)

std::cout << std::endl << std::endl;

std::cout << "Симметричный обход: ";

tree->inOrder(tree); //Вывод значений элементов (слева направо)

std::cout << std::endl << std::endl;

std::cout << "Обратный обход: ";

tree->postOrder(tree); //Вывод значений элементов (cниз вверх)

std::cout << std::endl << std::endl;

std::cout << "Горизонтальная печать дерева: " << std::endl;

tree->printTree(12); //Вывод значений элементов (слева направо)

std::cout << std::endl;

std::cout << "Вертикальная печать дерева: " << std::endl;

tree->printVTree(1); //Вывод значений элементов (слева направо)

std::cout << std::endl;

}

void add(SearchTree\* tree) //добавить элемент

{

char new\_elem{};

std::cout << std::endl << "Введите значение нового элемента: ";

std::cin >> new\_elem;

tree->add(new\_elem);

}

void delete\_left(SearchTree\* tree) //удалить поддерево слева

{

tree->deleteLeft();

}

void delete\_right(SearchTree\* tree) //удалить поддерево справа

{

tree->deleteRight();

}

void number\_of\_elements(SearchTree\* tree)

{

std::cout << "Количество элементов в дереве: " << tree->getAmountOfNodes() << std::endl << std::endl;

}

void height(SearchTree\* tree)

{

std::cout << "Высота дерева: " << tree->getHeight() << std::endl << std::endl;

}

void number\_by\_symbol(SearchTree\* tree) //найти количество элементов дерева поиска, начинающихся с заданного символа

{

char symbol{};

std::cout << "Введите символ:\t";

std::cin >> symbol;

std::cout << "Кол-во элементов, начинающихся с \'" << symbol << "\': " << tree->number\_by\_symbol(symbol) << std::endl;

}

SearchTree\* create\_ideal\_search(SearchTree\* tree)

{

std::vector<char> v = tree->copy\_to\_vector();

std::sort(v.begin(), v.end());

SearchTree\* temp;

temp = tree->ideal\_to\_search(v, 0, v.size() - 1);

return temp;

}

void print\_OpenGL(SearchTree\* tree, int argc, char\*\* argv)

{

tree->drawTree(argc, argv, 800, 600, 1, 2);

}

//------main-menu------

void Selection(int argc, char\*\* argv)

{

SearchTree\* tree = new SearchTree('X');

SearchTree\* seartch\_tree = new SearchTree('X');

bool boolean = true;

while (boolean)

{

std::cout << "Выберите нужный пункт меню:" << std::endl;

std::cout << "1. Создать идеально сбалансированное дерево." << std::endl;

std::cout << "2. Преобразовать сбалансированное дерево в дерево поиска." << std::endl;

std::cout << "3. Добавить новый элемент." << std::endl;

std::cout << "4. Удалить поддерево слева." << std::endl;

std::cout << "5. Удалить поддерево справа." << std::endl;

std::cout << "6. Показать кол-во элементов в дереве." << std::endl;

std::cout << "7. Показать высоту дерева." << std::endl;

std::cout << "8. Печать дерева поиска." << std::endl;

std::cout << "9. Печать дерева поиска в OpenGL." << std::endl;

std::cout << "10. Найти количество элементов дерева поиска, начинающихся с заданного символа." << std::endl;

std::cout << "0. Закрыть программу." << std::endl;

std::cout << "///////////////////////////////////////////////////////////////////////////////" << std::endl;

switch (Checking\_For\_Valid\_Input())

{

case 1:

tree = create\_ideal\_tree();

break;

case 2:

seartch\_tree = create\_ideal\_search(tree);

break;

case 3:

add(seartch\_tree);

break;

case 4:

delete\_left(seartch\_tree);

break;

case 5:

delete\_right(seartch\_tree);

break;

case 6:

number\_of\_elements(seartch\_tree);

break;

case 7:

height(seartch\_tree);

break;

case 8:

show\_tree(seartch\_tree);

break;

case 9:

print\_OpenGL(seartch\_tree, argc, argv);

break;

case 10:

number\_by\_symbol(seartch\_tree);

break;

case 0:

boolean = false;

break;

default:

std::cout << "Ошибка. Неверный пункт меню." << std::endl;

break;

}

system("pause");

system("cls");

}

}

Main.cpp

#include"functions.h"

int main(int argc, char\*\* argv)

{

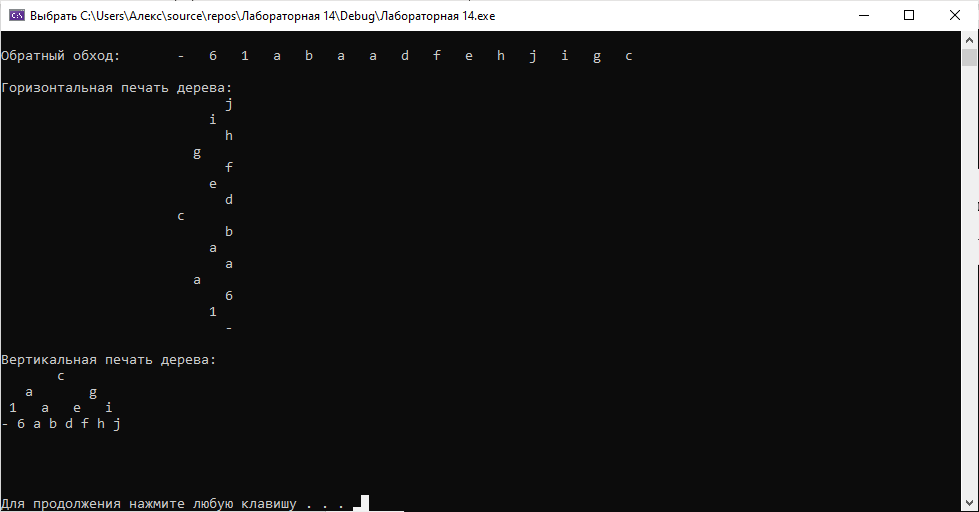
setlocale(LC\_ALL, "Ru");

Selection(argc, argv);

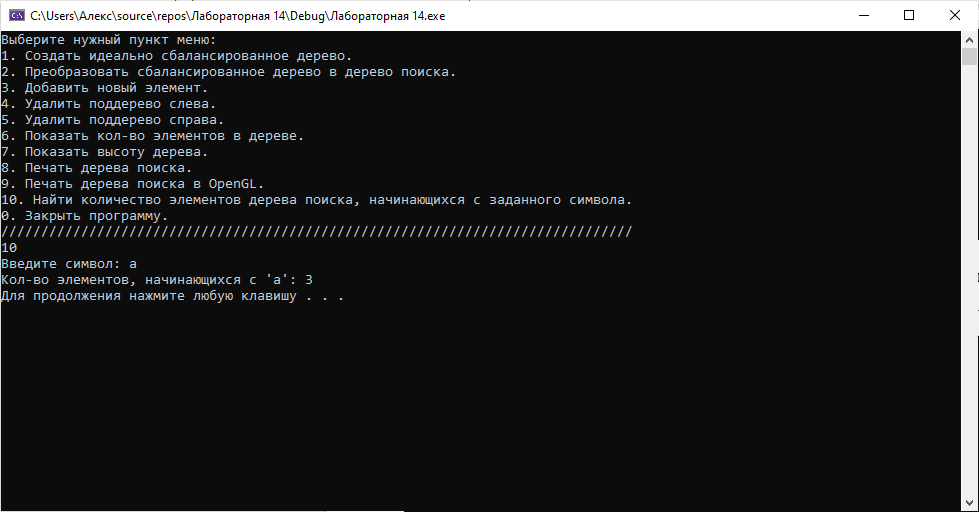
}

**Тесты:**

Создаётся сбалансированное дерево из 15 узлов и конвертируется в дерево поиска:



Находится количество элементов дерева, начинающихся с данного символа:



Печать дерева в OpenGL:

