Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра информационных технологий и автоматизированных систем

**ОТЧЕТ**

**о работе по информатике**

Семестр: 2

Тема: Бинарные деревья в С++

Выполнил студент ИВТ-22-2б:

Якушев Михаил Витальевич

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата, подпись)

Проверил доцент кафедры ИТАС:

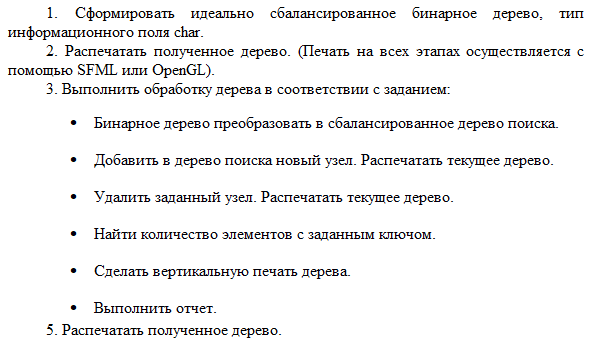
Полякова Ольга Андреевна

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата, подпись)

Пермь 2023

**1 Постановка задачи**

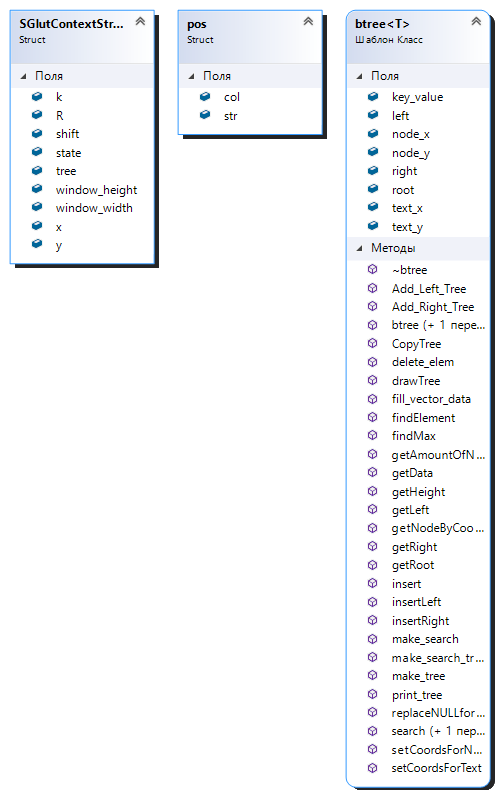
****

**2 Анализ задачи**

Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка.

В идеально сбалансированном дереве количество узлов справа и слева отличается не более чем на единицу.

**3 UML-диаграммы**

****

**4 Код программы**

#include <glut.h>

#include "Tree.h"

#include <Windows.h>

using namespace std;

SGlutContextStruct glutContext;

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

{

int x1 = 0;

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

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

int y2 = width - pow(2, height - 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;

}

template<class T>

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

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

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

glutCreateWindow("Бинарное дерево");

glutDisplayFunc(display<T>);

glutReshapeFunc(reshape);

glutMainLoop();

}

void reshape(int w, int h);

template <class T>

void display()

{

btree<T>\* tree = ((btree<T>\*)glutContext.tree)->CopyTree();

tree = tree->replaceNULLforEmpty();

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);

int width = 2 \* maxLeafs - 1;

int curLevel = 0;

int index = 0;

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

pos node;

vector<btree<T>\*> V;

vector<pos> Vi;

int R;

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;

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, 0.0, 1.0, 0.0);

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))

drawFillCircle(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);

}

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

{

srand(time(0));

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

setlocale(LC\_ALL, "rus");

btree<double>\* root = new btree<double>;

int size = 5 + rand() % 3;

cout << "SIZE: " << size << endl;

root = root->make\_tree(size);

double max = root->getData();

root->findMax(max);

cout << "Max element is: " << max << endl;

cout << endl;

root->print\_tree(1);

cout << endl << endl;

double del;

cout << "Delete element: ";

cin >> del;

cout << endl;

root = root->delete\_elem(del);

root->print\_tree(1);

cout << endl;

vector<double> vect;

root->fill\_vector\_data(vect);

btree<double>\* search\_root = new btree<double>();

search\_root->make\_search\_tree(vect);

search\_root->print\_tree(1);

cout << endl;

cout << "Insert element: ";

cin >> del;

search\_root->insert(del);

search\_root->print\_tree(1);

search\_root->drawTree(argc, argv, 800, 600, 10, 2);

return 0;

}

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();

}

#pragma once

#include <iostream>

#include <GL/glut.h>

#include <string>

#include <vector>

#include <ctime>

#include <algorithm>

#include <cmath>

using namespace std;

struct SGlutContextStruct

{

void\* tree;

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

};

bool myfunction(const double& i, const double& j) { return (i > j); }

template <class T>

class btree

{

public:

T key\_value;

btree\* left;

btree\* right;

btree\* root;

int node\_x;

int node\_y;

int text\_x;

int text\_y;

btree(T);

btree();

~btree();

void insert(T key);

void insertLeft(int key);

void insertRight(int key);

btree<T>\* search(T key, btree<T>\* leaf);

btree<T>\* findElement(T);

void findMax(double& max\_elem);

void Add\_Left\_Tree(btree<T>\* node)

{

left = node;

}

void Add\_Right\_Tree(btree<T>\* node)

{

right = node;

}

btree<T>\* delete\_elem(double& data);

void fill\_vector\_data(vector<double>& vect);

void make\_search\_tree(vector<double>& vect);

btree<T>\* make\_tree(int counter);

btree<T>\* search(T key);

void setCoordsForNode(int window\_width, int window\_height, int shift,

int tree\_width, int tree\_height, int x, int y, int R);

btree<T>\* getNodeByCoords(int x, int y, int R);

void setCoordsForText(int k, int shift);

void drawTree(

int argc, char\*\* argv,

int window\_width, int window\_height,

int shift, int k);

int getHeight();

btree<T>\* CopyTree();

btree<T>\* replaceNULLforEmpty();

btree<T>\* getLeft()

{

return left;

}

btree<T>\* getRight()

{

return right;

}

btree<T>\* getRoot()

{

return root;

}

int getData()

{

return key\_value;

}

void make\_search(vector<double> vect);

void print\_tree(int level);

int getAmountOfNodes();

};

template<class T>

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

template <class T>

void display();

void reshape(int w, int h);

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

extern SGlutContextStruct glutContext;

struct pos

{

int col;

int str;

};

template <class T>

void btree<T>::print\_tree(int level)

{

if (this != nullptr)

{

this->left->print\_tree(level + 1);

for (int i = 1; i < level; i++) cout << " ";

cout << this->key\_value << endl;

this->right->print\_tree(level + 1);

}

}

template <class T>

void btree<T>::fill\_vector\_data(vector<double>& vect)

{

if (this != nullptr)

{

vect.push\_back(this->key\_value);

this->left->fill\_vector\_data(vect);

this->right->fill\_vector\_data(vect);

}

}

template <class T>

void btree<T>::make\_search\_tree(vector<double>& vect)

{

vector<double> v = vect;

sort(v.begin(), v.end(), myfunction);

int mid = v.size() / 2;

this->key\_value = v[mid];

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

{

if (vect[i] != v[mid])

{

this->insert(vect[i]);

}

}

}

template <class T>

btree<T>\* btree<T>::make\_tree(int counter)

{

if (counter <= 0)

{

return nullptr;

}

double data = rand() % 50 + 1;

btree\* temp = new btree<T>(data);

temp->Add\_Left\_Tree(make\_tree(counter / 2));

temp->Add\_Right\_Tree(make\_tree(counter - counter / 2 - 1));

return temp;

}

template <class T>

btree<T>\* btree<T>::replaceNULLforEmpty()

{

btree<T>\* node = this->CopyTree();

int h = node->getHeight();

node = replace\_help(node, h);

return node;

}

template<class T>

btree<T>::btree(T key)

{

this->key\_value = key;

left = nullptr;

right = nullptr;

root = nullptr;

}

template<class T>

btree<T>::btree()

{

}

template<class T>

btree<T>::~btree()

{

}

template <class T>

btree<T>\* btree<T>::findElement(T key)

{

if ((this == NULL) || (key == this->key\_value))

return this;

if (key < this->key\_value) return this->left->findElement(key);

else return this->right->findElement(key);

}

template <class T>

void btree<T>::insert(T key)

{

if (key < -128 || key > 127)

{

cout << "Out of range" << endl;

return;

}

btree<T>\* tree = this;

btree<T>\* search = findElement(key);

while (tree != NULL)

{

if (key >= tree->key\_value)

{

if (tree->right != NULL)

{

tree = tree->right;

}

else

{

btree<T>\* t = new btree<T>(key);

t->root = tree;

tree->right = t;

break;

}

}

else if (key < tree->key\_value)

{

if (tree->left != NULL)

{

tree = tree->left;

}

else

{

btree<T>\* t = new btree<T>(key);

t->root = tree;

tree->left = t;

break;

}

}

}

}

template <class T>

void btree<T>::insertLeft(int key)

{

btree<T>\* node = new btree<T>();

if (this->left != NULL)

this->left->root = node;

node->left = this->left;

this->left = node;

node->root = this;

}

template<class T>

void btree<T>::insertRight(int key)

{

btree<T>\* node = new btree<T>();

if (this->right != NULL)

this->right->root = node;

node->right = this->right;

this->right = node;

node->root = this;

}

template <class T>

void btree<T>::make\_search(vector<double> vect)

{

if (this != nullptr)

{

sort(vect.begin(), vect.end(), myfunction);

this->left->make\_search(vect);

this->key\_value = vect[vect.size() - 1];

vect.pop\_back();

this->right->make\_search(vect);

}

}

template <class T>

void btree<T>::findMax(double& max\_elem)

{

if (this == nullptr)

{

return;

}

if (this->key\_value > max\_elem)

{

max\_elem = this->key\_value;

}

this->left->findMax(max\_elem);

this->right->findMax(max\_elem);

}

template <class T>

btree<T>\* btree<T>::delete\_elem(double& data)

{

if (this == nullptr)

{

return this;

}

this->left = this->left->delete\_elem(data);

if (data == this->key\_value && data != -1)

{

data = -1;

btree<T>\* tmp;

if (this->right == nullptr)

{

tmp = this->left;

}

else

{

btree<T>\* ptr = this->right;

if (ptr->left == nullptr)

{

ptr->left = this->left;

tmp = ptr;

}

else

{

btree<T>\* pmin = ptr->left;

while (pmin->left != nullptr)

{

ptr = pmin;

pmin = ptr->left;

}

ptr->left = pmin->right;

pmin->left = this->left;

pmin->right = this->right;

tmp = pmin;

}

}

delete this;

return tmp;

}

this->right = this->right->delete\_elem(data);

return this;

}

void drawLine(int x1, int y1)

{

glBegin(GL\_LINES);

glColor3f(0, 0, 0);

glVertex2i(x1, y1);

glEnd();

}

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

{

glBegin(GL\_LINES);

glColor3f(0, 0, 0);

glVertex2i(x1, y1);

glVertex2i(x2, y2);

glEnd();

}

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

{

glColor3f(0, 0.5, 0.5);

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();

}

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

{

glColor3f(0, 0, 0);

glPushMatrix();

glTranslatef(text\_x, text\_y, 0.0);

string s = to\_string(text);

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

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

}

template<class T>

btree<T>\* btree<T>::CopyTree()

{

btree<T>\* tree = new btree<T>(this->key\_value);

if (this->root != NULL)

tree->root = this->root;

if (this->left != NULL)

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

if (this->right != NULL)

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

return tree;

}

template <class T>

int btree<T>::getHeight()

{

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

if (this == NULL)return 0;

if (this->left != NULL) h1 = this->left->getHeight();

if (this->right != NULL) h2 = this->right->getHeight();

if (h1 >= h2) return h1 + 1;

else return h2 + 1;

}

template<class T>

int btree<T>::getAmountOfNodes()

{

if (this == NULL)return 0;

if ((this->left == NULL) && (this->right == NULL)) return 1;

int l = 0;

int r = 0;

if (this->left != NULL) l = this->left->getAmountOfNodes();

if (this->right != NULL) r = this->right->getAmountOfNodes();

return (l + r + 1);

}

template<class T>

btree<T>\* get\_help(btree<T>\* 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;

btree<T>\* 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;

}

template<class T>

void btree<T>::drawTree(int argc, char\*\* argv, int window\_width,

int window\_height, int shift, int k)

{

btree<T>\* 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<T>(argc, argv);

}

template<class T>

btree<T>\* btree<T>::getNodeByCoords(int x, int y, int R)

{

btree<T>\* node = this;

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

return node;

}

template <class T>

void btree<T>::setCoordsForText(int k, int R)

{

text\_x = node\_x - 3 \* R / 4;

text\_y = node\_y - 3 \* R / (4 \* k);

}

template <class T>

void btree<T>::setCoordsForNode(int window\_width, int window\_height, int shift, int tree\_width, int tree\_height, int x, int y, int R)

{

if (tree\_width != tree\_height)

{

int k\_x = (window\_width - 2 \* (shift + R)) / (tree\_width - 1);

int k\_y = (window\_height - 2 \* (shift + R)) / (tree\_height - 1);

node\_x = k\_x \* x + shift + R;

node\_y = window\_height - k\_y \* y - shift - R;

}

else

{

node\_x = window\_width / 2;

node\_y = window\_height / 2;

}

}

template <class T>

btree<T>\* btree<T>::search(T key)

{

if (root != NULL)

{

if (key == root->key\_value)

return root;

if (key < root->key\_value)

return search(key, root->left);

else

return search(key, root->right);

}

else return NULL;

}

template <class T>

btree<T>\* btree<T>::search(T key, btree<T>\* leaf)

{

if (leaf != NULL)

{

if (key == leaf->key\_value)

return leaf;

if (key < leaf->key\_value)

return search(key, leaf->left);

else

return search(key, leaf->right);

}

else return NULL;

}

template <class T>

btree<T>\* replace\_help(btree<T>\* node, int h)

{

int curLevel = getLevel(node);

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

{

node->insertLeft(NULL);

}

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

{

node->insertRight(NULL);

}

if (node->getLeft() != NULL) node->Add\_Left\_Tree(replace\_help(node->getLeft(), h));

if (node->getRight() != NULL) node->Add\_Right\_Tree(replace\_help(node->getRight(), h));

return node;

}

template<class T>

int getLevel(btree<T>\* node)

{

if (node->getRoot() == NULL)

{

return 0;

}

else

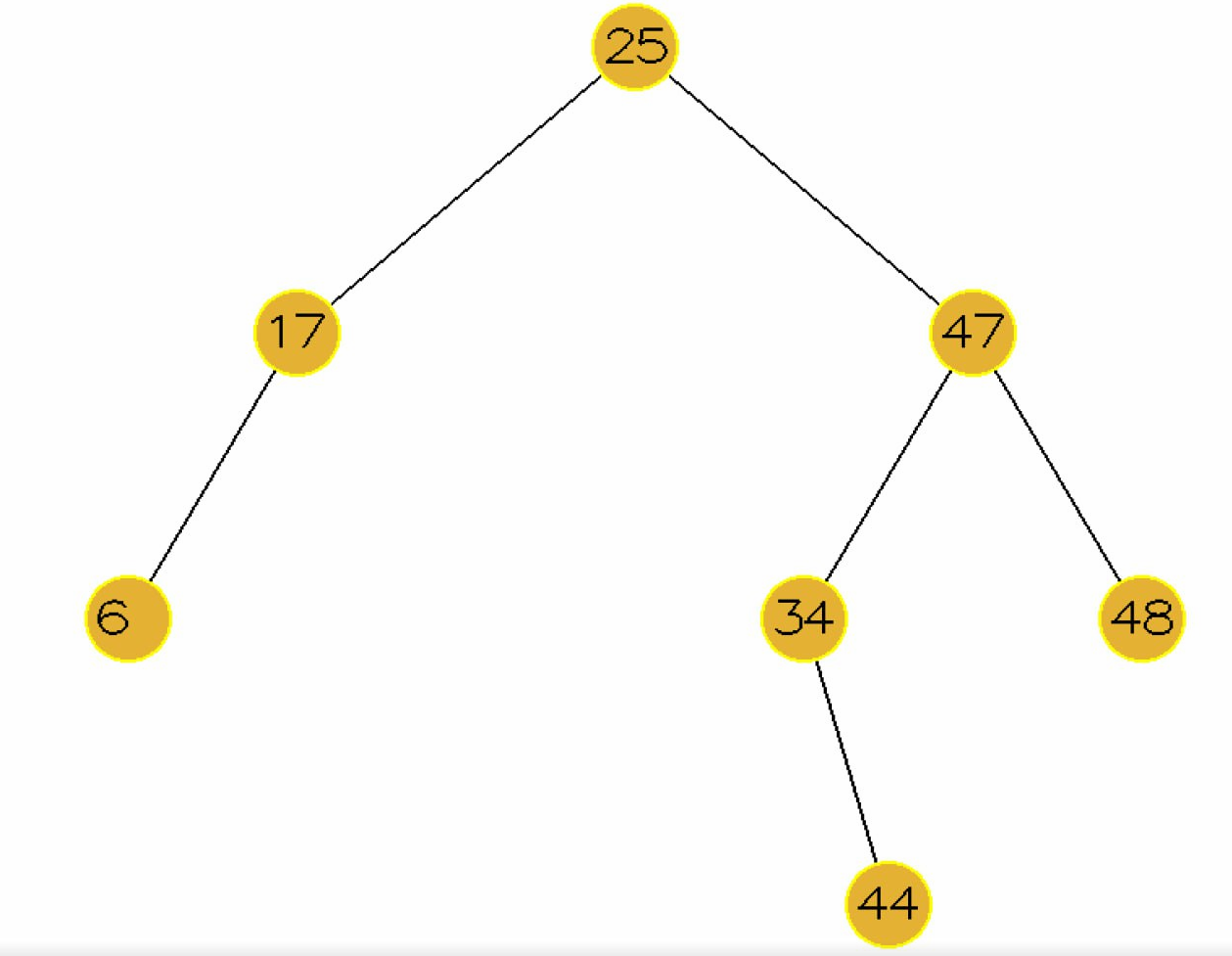
{

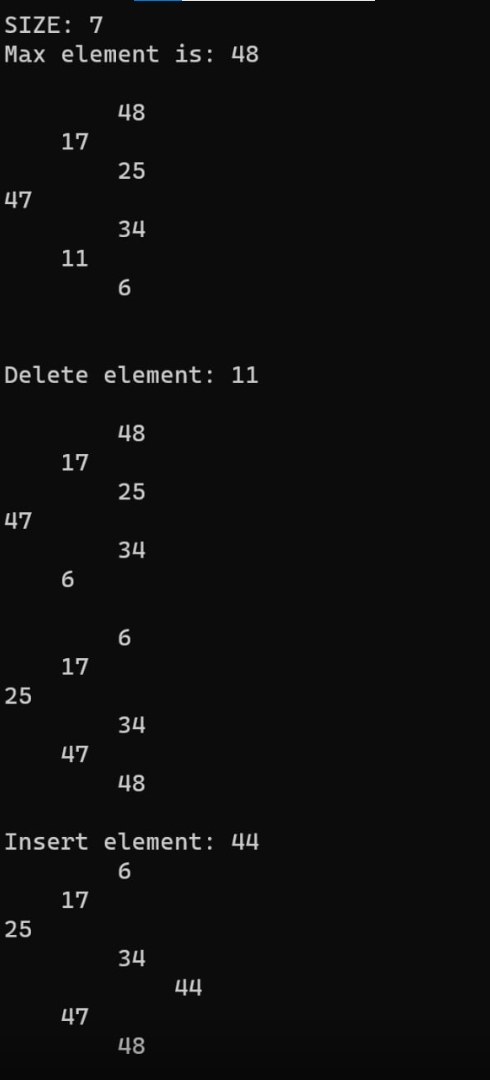
return getLevel(node->getRoot()) + 1;

}

}

**5 Результаты работы программы**

****

****