Министерство высшего образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

**Пермский национальный исследовательский политехнический университет**

Кафедра «Информационные технологии и автоматизированные системы»

**ОТЧЕТ**

Дисциплина: «Основы алгоритмизации и программирования»

Тема: «Бинарное дерево»

Семестр 2

*Работу выполнил: Учащийся группы ИВТ - 22-2б: Цыбуцынин Фёдор Александрович*

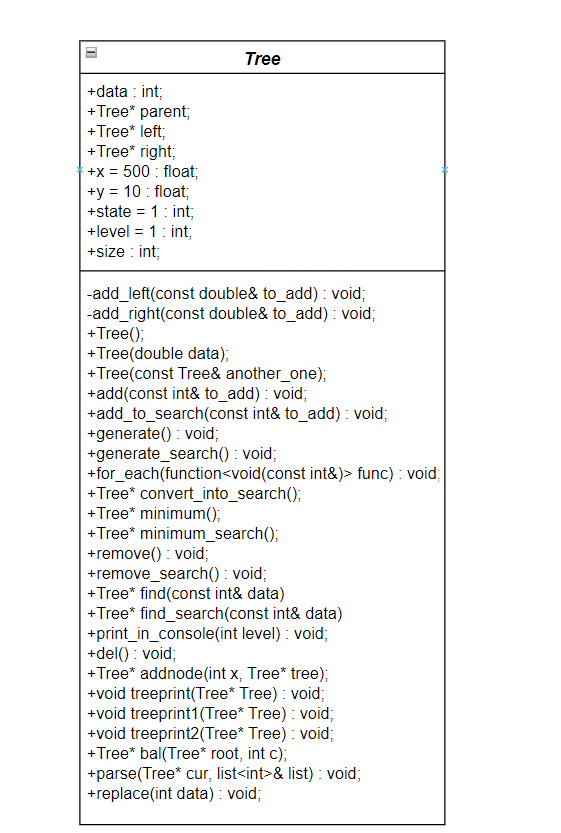
*Работу проверил: доцент кафедры ИТАС:*

*Полякова Ольга Андреевна*

Г. Пермь – 2023

**Постановка задачи**

Требуется реализовать алгоритмы для собственного варианта бинарного дерева поиска, имеющего не менее трёх уровней .  
Алгоритмы:  
1. Необходимо реализовать функции для редактирования дерева:  
- Вставка узла.  
- Удаление узла.  
- Поиск элемента по ключу.  
2. Реализовать алгоритмы обхода дерева:  
2.1 Прямой  
2.2 Симметричный  
2.3 Обратный  
3. Реализовать алгоритм балансировки дерева.  
4. Реализовать вертикальную и горизонтальную печать.

**UML-диаграмма**

**Рис. 1 – UML диаграмма**

**Код**

#include <iostream>

#include <functional>

#include <string>

#include <sstream>

#include <SFML/Graphics.hpp>

#pragma comment (lib, "opengl32.lib")

#pragma comment (lib, "glu32.lib")

#pragma comment(lib,"glut32.lib")

using namespace std;

using namespace sf;

const float RAD = 35;

class Tree

{

private:

void add\_left(const double& to\_add)

{

Tree\* new\_tree = new Tree;

new\_tree->data = to\_add;

new\_tree->parent = this;

left = new\_tree;

++left->size;

}

void add\_right(const double& to\_add)

{

Tree\* new\_tree = new Tree;

new\_tree->data = to\_add;

new\_tree->parent = this;

right = new\_tree;

++right->size;

}

public:

int data;

Tree\* parent;

Tree\* left;

Tree\* right;

int size;

float x = 500, y = 10;

int state, level = 1;

Tree()

{

parent = nullptr;

left = nullptr;

right = nullptr;

data = 0.0;

size = 0;

}

Tree(double data)

{

parent = nullptr;

left = nullptr;

right = nullptr;

data = data;

size = 1;

}

Tree(const Tree& another\_one)

{

parent = another\_one.parent;

left = another\_one.left;

right = another\_one.right;

data = another\_one.data;

size = another\_one.size;

}

void add(const int& to\_add)

{

if (size == 0)

data = to\_add;

else if (left == nullptr)

add\_left(to\_add);

else if (right == nullptr)

add\_right(to\_add);

else if (right->size < left->size)

right->add(to\_add);

else

left->add(to\_add);

++size;

}

void add\_to\_search(const int& to\_add)

{

if (size == 0)

{

data = to\_add;

size = 1;

}

else

{

if (to\_add > data)

{

++size;

if (right == nullptr)

{

Tree\* new\_tree = new Tree;

new\_tree->data = to\_add;

new\_tree->size = 1;

right = new\_tree;

new\_tree->parent = this;

}

else

{

right->add\_to\_search(to\_add);

}

}

else if (to\_add < data)

{

++size;

if (left == nullptr)

{

Tree\* new\_tree = new Tree;

new\_tree->data = to\_add;

new\_tree->size = 1;

left = new\_tree;

new\_tree->parent = this;

}

else

{

left->add\_to\_search(to\_add);

}

}

}

}

void generate()

{

cout << "Введите размер дерева: ";

int count;

cin >> count;

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

{

cout << "Введите число для добавления: ";

int to\_add;

cin >> to\_add;

add(to\_add);

}

}

void generate\_search()

{

cout << "Введите размер дерева поиска: ";

int count;

cin >> count;

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

{

cout << "Введите число для добавления: ";

double to\_add;

cin >> to\_add;

add\_to\_search(to\_add);

}

}

void for\_each(function<void(const int&)> func)

{

if (this != nullptr)

{

func(data);

if (left != nullptr)

left->for\_each(func);

if (right != nullptr)

right->for\_each(func);

}

}

Tree\* convert\_into\_search()

{

Tree\* new\_tree = new Tree;

for\_each([&](const double& data) { new\_tree->add\_to\_search(data); });

return new\_tree;

}

Tree\* minimum();

Tree\* minimum\_search()

{

if (left == nullptr)

return this;

else

return left->minimum\_search();

}

void remove()

{

Tree\* to\_delete = this;

Tree\* dad = parent;

if (to\_delete->left == nullptr and to\_delete->right == nullptr)

del();

else if (to\_delete->left == nullptr and to\_delete->right != nullptr)

{

to\_delete->data = to\_delete->right->data;

to\_delete->right->remove();

}

else if (to\_delete->left != nullptr and to\_delete->right == nullptr)

{

to\_delete->data = to\_delete->left->data;

to\_delete->left->remove();

}

else

{

to\_delete->data = to\_delete->left->data;

to\_delete->left->remove();

}

}

void remove\_search()

{

Tree\* to\_delete = this;

Tree\* dad = parent;

if (to\_delete->left == nullptr and to\_delete->right == nullptr)

del();

else if (to\_delete->left == nullptr and to\_delete->right != nullptr)

{

to\_delete->data = to\_delete->right->data;

to\_delete->right->remove\_search();

}

else if (to\_delete->left != nullptr and to\_delete->right == nullptr)

{

to\_delete->data = to\_delete->left->data;

to\_delete->left->remove\_search();

}

else

{

Tree\* replacer = to\_delete->right->minimum\_search();

to\_delete->data = replacer->data;

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

{

replacer->parent->left = replacer->right;

if (replacer->right != nullptr)

replacer->right->parent = replacer->parent;

}

else

{

replacer->parent->right = replacer->right;

if (replacer->right != nullptr)

replacer->right->parent = replacer->parent;

}

delete replacer;

}

}

Tree\* find(const int& data)

{

if (this == nullptr)

return nullptr;

if (this->data == data)

return this;

if (left->find(data) != nullptr)

return left->find(data);

return right->find(data);

}

Tree\* find\_search(const int& data)

{

if (this == nullptr)

return nullptr;

if (this->data == data)

return this;

if (data > this->data)

return right->find\_search(data);

return left->find\_search(data);

}

void print\_in\_console()

{

if (this != nullptr)

{

cout << data << " ";

left->print\_in\_console();

right->print\_in\_console();

cout << endl;

}

}

friend ostream& operator<<(ostream& stream, const Tree\* start)

{

if (start != nullptr)

{

stream << start->data;

}

return stream;

}

void del()

{

if (this != nullptr)

{

if (left != nullptr)

left->del();

if (right != nullptr)

right->del();

parent->size -= size;

if (this == parent->left)

parent->left = nullptr;

else

parent->right = nullptr;

delete this;

}

}

Tree\* addnode(int x, Tree\* tree) {

if (tree == NULL) { // Если дерева нет, то формируем корень

tree = new Tree; // память под узел

tree->data = x; // поле данных

tree->left = NULL;

tree->right = NULL; // ветви инициализируем пустотой

}

else if (x < tree->data) // условие добавление левого потомка

tree->left = addnode(x, tree->left);

else // условие добавление правого потомка

tree->right = addnode(x, tree->right);

return(tree);

}

void treeprint(Tree\* Tree) {

if (Tree != NULL) { //Пока не встретится пустой узел

cout << Tree->data << " "; //Отображаем корень дерева

treeprint(Tree->left); //Рекурсивная функция для левого поддерева

treeprint(Tree->right); //Рекурсивная функция для правого поддерева

}

}

void treeprint1(Tree\* Tree) {

if (Tree != NULL) { //Пока не встретится пустой узел

treeprint(Tree->left); //Рекурсивная функция для левого поддерева

cout << Tree->data << " "; //Отображаем корень дерева

treeprint(Tree->right); //Рекурсивная функция для правого поддерева

}

}

void treeprint2(Tree\* Tree) {

if (Tree != NULL) { //Пока не встретится пустой узел

treeprint(Tree->left); //Рекурсивная функция для левого поддерева

treeprint(Tree->right); //Рекурсивная функция для правого поддерева

cout << Tree->data << " "; //Отображаем корень дерева

}

}

Tree\* balanced(int count)

{

if (count <= 0)

{

return nullptr;

}

double a;

cout << "Input data to balanced tree: "; cin >> a;

Tree\* temp = new Tree(a);

temp->add\_left(balanced(count / 2));

temp->add\_right(balanced(count - count / 2 - 1));

return temp;

}

};

Tree\* min\_of\_three(Tree\* mas[3])

{

Tree\* min = new Tree;

min = mas[0];

if (mas[1]->data < min->data)

min = mas[1];

if (mas[2]->data < min->data)

min = mas[2];

return min;

}

Tree\* Tree::minimum()

{

Tree\* mas[3];

mas[0] = this;

if (left != nullptr)

mas[1] = left->minimum();

else

mas[1] = this;

if (right != nullptr)

mas[2] = right->minimum();

else

mas[2] = this;

return min\_of\_three(mas);

}

int depth;

void count\_levels(Tree\* root)

{

if (root != nullptr)

{

if (root->parent != nullptr)

root->level = root->parent->level + 1;

if (depth < root->level)

depth = root->level;

count\_levels(root->left);

count\_levels(root->right);

}

}

void calculate\_coords(Tree\* node)

{

if (node != nullptr)

{

if (node->parent != nullptr)

{

if (node->level == 2)

node->x = node->x + node->state \* (pow(2, depth - 1) / 2) \* 40;

else

node->x = node->parent->x + node->state \* (pow(2, depth - 1) / pow(2, node->level - 1)) \* 40;

node->y = node->parent->y + 80;

}

if (node->left != nullptr)

{

node->left->state = -1;

calculate\_coords(node->left);

}

if (node->right != nullptr)

{

node->right->state = 1;

calculate\_coords(node->right);

}

}

}

void draw\_lines(Tree\* node, RenderWindow& window)

{

if (node != nullptr)

{

if (node->parent != nullptr)

{

vector<Vertex> line =

{

Vertex(Vector2f(node->parent->x + RAD, node->parent->y + RAD), Color::Black),

Vertex(Vector2f(node->x + RAD, node->y + RAD), Color::Black)

};

window.draw(&line[0], line.size(), Lines);

}

if (node->left != nullptr)

draw\_lines(node->left, window);

if (node->right != nullptr)

draw\_lines(node->right, window);

}

}

void draw\_nodes(Tree\* node, RenderWindow& window)

{

if (node != nullptr)

{

CircleShape circle(RAD);

circle.setPosition(node->x, node->y);

circle.setFillColor(Color::White);

circle.setOutlineColor(Color::Black);

circle.setOutlineThickness(3);

Font font;

font.loadFromFile("ofont.ru\_a\_CampusOtl.ttf");

stringstream txt;

txt << node->data;

Text text(txt.str(), font, 35);

text.setPosition(node->x + RAD / 2, node->y + RAD / 2);

text.setFillColor(Color::Black);

window.draw(circle);

window.draw(text);

if (node->left != nullptr)

draw\_nodes(node->left, window);

if (node->right != nullptr)

draw\_nodes(node->right, window);

}

}

void draw(Tree\* tree)

{

RenderWindow MainWindow(VideoMode(1200, 500), "Our tree");

depth = 0;

count\_levels(tree);

calculate\_coords(tree);

while (MainWindow.isOpen())

{

Event event;

while (MainWindow.pollEvent(event))

{

if (event.type == Event::Closed)

{

MainWindow.close();

break;

}

}

MainWindow.clear(Color::White);

draw\_lines(tree, MainWindow);

draw\_nodes(tree, MainWindow);

MainWindow.display();

}

}

int main()

{

setlocale(LC\_ALL, "RUS");

Tree\* bush = new Tree;

bush->generate();

cout << "Наше дерево: " << endl;

draw(bush);

cout << "Прямой обход дерева: ";

bush->treeprint(bush);

cout << endl;

cout << "Симметричный обход дерева: ";

bush->treeprint1(bush);

cout << endl;

cout << "Обратный обход дерева: ";

bush->treeprint2(bush);

cout << endl;

bush->addnode(100, bush);

cout << "Дерево после добавления элемента: " << endl;

draw(bush);

bush->minimum()->remove();

cout << "Дерево после удаления минимума: " << endl;

draw(bush);

Tree\* search\_bush = bush->convert\_into\_search();

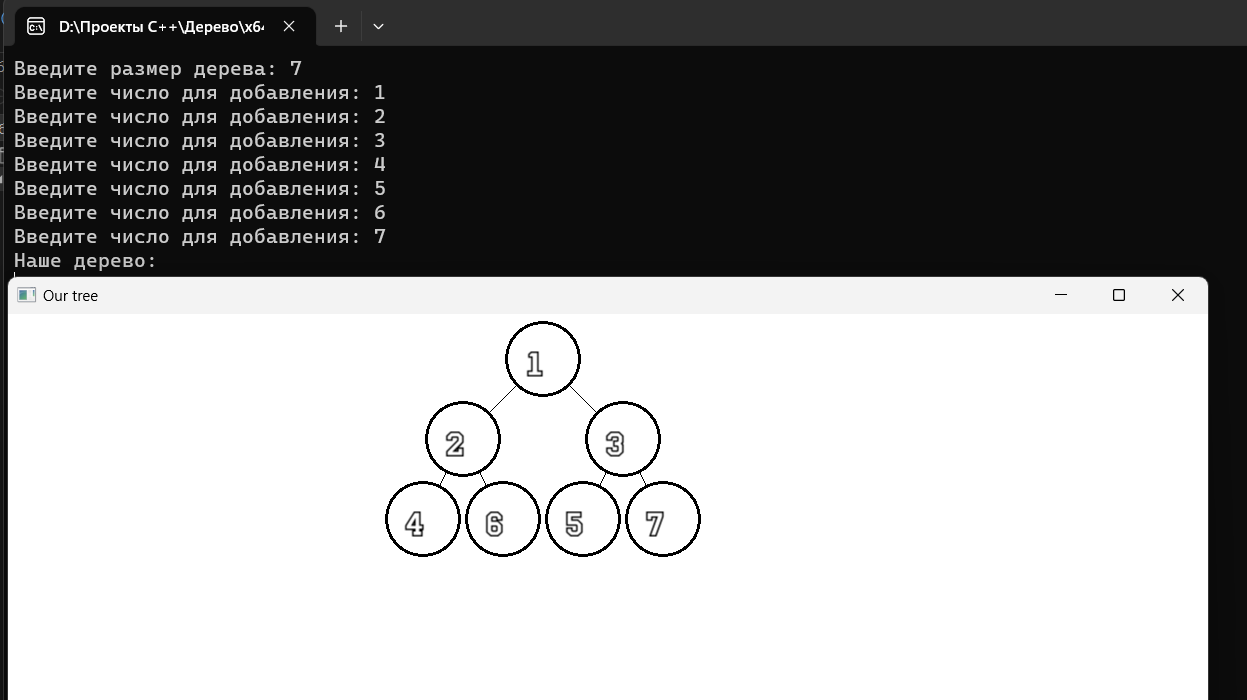
cout << "Преобразование дерева в БДП: " << endl;

draw(search\_bush);

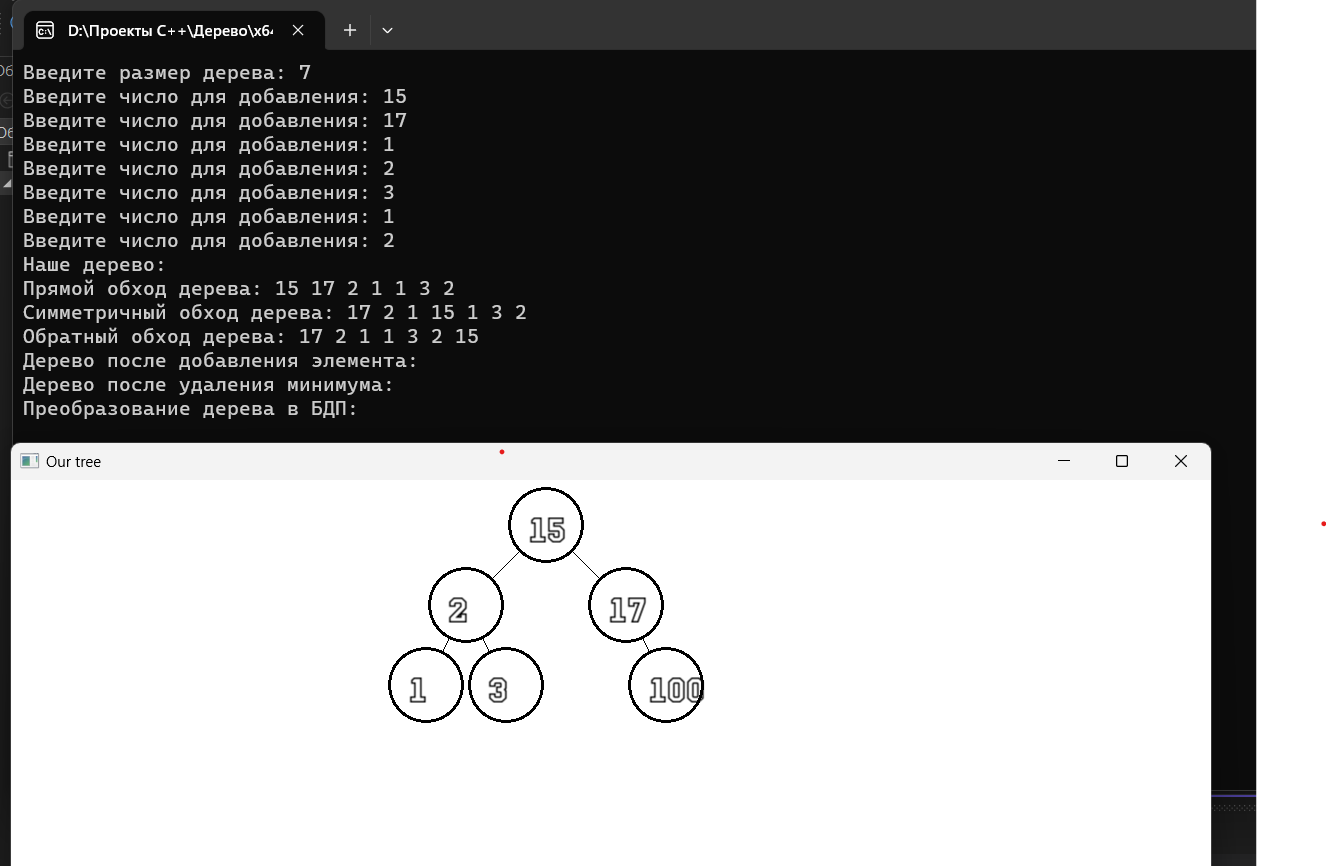
return 0;

}

**Результаты выполнения программы**



**Рис. 2 – Печать дерева с помощью SFML**



**Рис. 3 – Преобразованное дерево**