Федеральное государственное автономное образовательное учреждение высшего образования «Пермский национальный исследовательский политехнический университет», ПНИПУ

ЛАБОРАТОРНАЯ РАБОТА

БИНАРНЫЕ ДЕРЕВЬЯ

Выполнил: студент группы РИС-23-3б

Артем Владимирович Швецов

Проверила: доцент кафедры ИТАС

Ольга Андреевна Полякова

Пермь 2024

**Постановка задачи**

1. Сформировать идеально сбалансированное бинарное дерево, тип информационного поля указан в варианте.

2. Распечатать полученное дерево.

3. Выполнить обработку дерева в соответствии с заданием, вывести полученный результат.

4. Преобразовать идеально сбалансированное дерево в дерево поиска.

5. Распечатать полученное дерево.

Для варианта 25:

### Тип информационного поля double. Найти минимальный элемент в дереве.

**Анализ задачи**

1. Для управления деревом (создание, изменение, просмотр ветвей дерева), реализуются функции добавления/удаления правых и левых поддеревьев, перехода к потомкам/предку, получения значения узла.
2. Для формирования идеально сбалансированного дерева и дерева поиска реализуются функции обхода и сохранения информации о значениях, а также построения нового дерева.
3. Обработка дерева строит дерево поиска и ищет самый левый элемент.
4. Печать дерева происходит в двух режимах: печать в консоль, отображение в графической среде.

**UML**

**Код**

Tree.h

#pragma once

#include <SFML/Graphics.hpp>

#include <list>

#include <vector>

#include <iostream>

#include <string>

#include <sstream>

#include <fstream>

#include <queue>

#include <iomanip>

template <typename T>

class Tree {

private:

Tree<T>\* left;

Tree<T>\* right;

Tree<T>\* parent;

T data;

Tree<T>\* build\_balanced\_bst(const std::vector<T>&, int, int, bool);

void DrawElement(Tree<T>\*, int, int, int, int, int, sf::RenderWindow&);

public:

Tree();

Tree(T);

~Tree();

T get\_data();

void set\_data(T);

void replace(T);

void delete\_left();

void delete\_right();

void insert\_left(T);

void insert\_right(T);

void view\_direct();

void view\_symmetric();

void view\_reverse();

void copy\_list(std::list<T>&);

void print\_tree(int);

void delete\_tree();

void insert(T);

void erase(T);

void set\_left(Tree<T>\*);

void set\_right(Tree<T>\*);

int getHeight();

int countNodes();

int obh(Tree<T>\* node);

void print\_vert();

void fillBlanks(int);

Tree<T>\* eject\_left();

Tree<T>\* eject\_right();

Tree<T>\* get\_left();

Tree<T>\* get\_right();

Tree<T>\* get\_parent();

Tree<T>\* copy();

Tree<T>\* search\_all(T);

Tree<T>\* search\_sort(T);

Tree<T>\* next();

Tree<T>\* prev();

Tree<T>\* find\_min();

Tree<T>\* find\_min\_helper();

Tree<T>\* find\_max();

Tree<T>\* balanced(int);

void in\_order\_traversal(std::vector<T>&);

Tree<T>\* create\_balanced\_tree();

Tree<T>\* create\_search\_tree();

void DrawT();

};

template <typename T>

Tree<T>::Tree() {

left = right = parent = nullptr;

data = 0;

}

template <typename T>

Tree<T>::Tree(T data) {

this->data = data;

left = right = parent = nullptr;

}

template <typename T>

Tree<T>::~Tree() {

delete this;

}

template <typename T>

T Tree<T>::get\_data() {

return data;

}

template <typename T>

void Tree<T>::set\_data(T dataNew)

{

data = dataNew;

}

template <typename T>

void Tree<T>::replace(T data) {

Tree<T>\* search\_tree = new Tree<T>();

std::list<T> list1;

this->copy\_list(list1);

for (auto data : list1) {

search\_tree->insert(data);

}

Tree<T>\* current = search\_tree->search\_sort(data);

if (current == nullptr) return;

if (current->left == nullptr && current->right == nullptr) {

current->data = -1;

return;

}

}

template <typename T>

void Tree<T>::delete\_left() {

if (left != NULL) {

left->delete\_left();

left->delete\_right();

delete left;

}

}

template <typename T>

void Tree<T>::delete\_right() {

if (right != NULL) {

right->delete\_left();

right->delete\_right();

delete right;

}

}

template <typename T>

void Tree<T>::insert\_left(T data) {

Tree<T>\* new\_node = new Tree(data);

if (this->left != nullptr) {

this->left->parent = new\_node;

new\_node->left = this->left;

}

this->left = new\_node;

new\_node->parent = this;

}

template <typename T>

void Tree<T>::insert\_right(T data) {

Tree<T>\* new\_node = new Tree(data);

if (this->right != nullptr) {

this->right->parent = new\_node;

new\_node->right = this->right;

}

this->right = new\_node;

new\_node->parent = this;

}

template <typename T>

void Tree<T>::view\_direct() {

if (this == nullptr) {

return;

}

else {

std::cout << this->get\_data() << ' ';

this->get\_left()->view\_direct();

this->get\_right()->view\_direct();

}

}

template <typename T>

void Tree<T>::view\_symmetric() {

if (this != nullptr) {

this->left->view\_symmetric();

std::cout << this->data << ' ';

this->right->view\_symmetric();

}

}

template <typename T>

void Tree<T>::view\_reverse() {

if (this != nullptr) {

this->left->view\_reverse();

this->right->view\_reverse();

std::cout << this->data << ' ';

}

}

template <typename T>

void Tree<T>::copy\_list(std::list<T>& list1) {

if (this == nullptr) {

return;

}

else {

list1.push\_back(this->get\_data());

this->get\_left()->view\_direct();

this->get\_right()->view\_direct();

}

}

template <typename T>

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

if (this != NULL) {

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

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

std::cout << this->get\_data() << std::endl;

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

}

}

template <typename T>

void Tree<T>::delete\_tree() {

this->delete\_left();

this->delete\_right();

delete this;

}

template <typename T>

void Tree<T>::insert(T data) {

Tree<T>\* current = this;

while (current != nullptr) {

if (data > current->data) {

if (current->right != nullptr) {

current = current->right;

}

else {

current->insert\_right(data);

return;

}

}

else if (data < current->data) {

if (current->left != nullptr) {

current = current->left;

}

else {

current->insert\_left(data);

return;

}

}

else return;

}

}

template <typename T>

void Tree<T>::erase(T data) {

Tree<T>\* to\_erase = this->search\_sort(data);

Tree<T>\* te\_parent = to\_erase->parent;

if (to\_erase->left == nullptr && to\_erase->right == nullptr) {

if (te\_parent->left == to\_erase) {

te\_parent->left == nullptr;

}

else {

te\_parent->right == nullptr;

}

delete to\_erase;

}

else if (to\_erase->left == nullptr && to\_erase->right != nullptr) {

if (te\_parent->left == to\_erase) {

te\_parent->left == to\_erase->right;

}

else {

te\_parent->right == to\_erase->right;

}

to\_erase->right->parent = te\_parent;

delete to\_erase;

}

else if (to\_erase->left != nullptr && to\_erase->right == nullptr) {

if (te\_parent->left == to\_erase) {

te\_parent->left == to\_erase->left;

}

else {

te\_parent->right == to\_erase->left;

}

to\_erase->left->parent = te\_parent;

delete to\_erase;

}

else {

Tree<T>\* next = to\_erase->next();

to\_erase->data = next->data;

if (next == next->parent->left) {

next->parent->left = next->right;

if (next->right != nullptr) {

next->right->parent = next->parent;

}

}

else {

next->parent->right = next->right;

if (next->right != nullptr) {

next->right->parent = next->parent;

}

}

delete next;

}

}

template <typename T>

void Tree<T>::set\_left(Tree<T>\* tree) {

left = tree;

}

template <typename T>

void Tree<T>::set\_right(Tree<T>\* tree) {

right = tree;

}

template <typename T>

int Tree<T>::getHeight() {

int h1 = 0, h2 = 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 <typename T>

int Tree<T>::countNodes() {

if (this == NULL) return 0;

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

int l = 0;

int r = 0;

if (this->left != nullptr) l = this->left->countNodes();

if (this->right != nullptr) r = this->right->countNodes();

return l + r + 1;

}

template <typename T>

int Tree<T>::obh(Tree<T>\* tree) {

std::ofstream f("print.txt");

int countNodes = tree->countNodes();

std::queue<Tree<T>\*> q;

q.push(tree);

while (!q.empty()) {

Tree<T>\* temp = q.front();

q.pop();

f << temp->data << std::endl;

if (temp->left)

q.push(temp->left);

if (temp->right)

q.push(temp->right);

}

f.close();

return countNodes;

}

template <typename T>

void Tree<T>::fillBlanks(int i) {

if (i == 1) {

return;

}

if (!this->right) {

this->insert\_right(-1);

}

if (!this->left) {

this->insert\_left(-1);

}

this->right->fillBlanks(i - 1);

this->left->fillBlanks(i - 1);

}

template <typename T>

void Tree<T>::print\_vert() {

Tree<T>\* tree = this->copy();

int height = tree->getHeight();

tree->fillBlanks(height);

int amount = obh(tree);

std::ifstream f("print.txt");

T\* mas = new T[amount];

for (int i = 0; i < amount; i++) {

char str[255];

f.getline(str, 255);

mas[i] = atoi(str);

}

f.close();

int count = 0;

int\* spaces = new int[height];

spaces[0] = 0;

for (int i = 1; i < height; i++) {

spaces[i] = spaces[i - 1] \* 2 + 1;

}

if (height == 1) {

std::cout << std::setw(5) << mas[0] << std::endl;

}

else

{

for (int i = 0, l = height - 1; i < height - 1; i++, l--) {

for (int j = 0, k = 0; j < pow(2, i); j++, k++) {

if (k == 0) {

for (int u = 0; u < spaces[l]; u++) {

std::cout << std::setw(5) << ' ';

}

}

else {

for (int u = 0; u < spaces[l + 1]; u++) {

std::cout << std::setw(5) << ' ';

}

}

if (mas[count] != -1)

std::cout << std::setw(5) << mas[count++];

else

std::cout << std::setw(5) << ' ';

}

std::cout << std::endl;

}

T\* last\_str = new T[pow(2, height - 1)];

for (int i = 0; i < pow(2, height - 1); i++) {

last\_str[i] = 2000000000;

}

int sch1 = 0;

int sch2 = spaces[height - 2] + 1;

for (int i = count; i < amount; i += 2) {

if (i <= amount - 1) {

last\_str[sch1] = mas[i];

sch1 += 2;

}

if (i + 1 <= amount - 1) {

last\_str[sch2] = mas[i + 1];

sch2 += 2;

}

if (sch1 >= pow(2, height - 1) || sch2 >= pow(2, height - 1)) {

sch1 = 1;

sch2 = spaces[height - 2] + 2;

}

}

for (int i = 0; i < pow(2, height - 1); i++)

if (last\_str[i] != 2000000000) last\_str[i] = mas[count++];

for (int i = 0; i < pow(2, height - 1); i++)

if (last\_str[i] != 2000000000 && last\_str[i] != -1) std::cout << std::setw(5) << last\_str[i] << std::setw(5) << ' ';

else std::cout << std::setw(5) << ' ' << std::setw(5) << ' ';

std::cout << std::endl;

delete[] last\_str;

}

delete[]mas;

delete[]spaces;

}

template <typename T>

Tree<T>\* Tree<T>::eject\_left() {

Tree<T>\* temp = left;

left = nullptr;

if (temp != nullptr) {

temp->parent = nullptr;

}

return temp;

}

template <typename T>

Tree<T>\* Tree<T>::eject\_right() {

Tree<T>\* temp = right;

right = nullptr;

if (temp != nullptr) {

temp->parent = nullptr;

}

return temp;

}

template <typename T>

Tree<T>\* Tree<T>::get\_left() {

return left;

}

template <typename T>

Tree<T>\* Tree<T>::get\_right() {

return right;

}

template <typename T>

Tree<T>\* Tree<T>::get\_parent() {

return parent;

}

template <typename T>

Tree<T>\* Tree<T>::copy() {

Tree<T>\* new\_tree = new Tree<T>(data);

if (left != nullptr) {

new\_tree->left = left->copy();

new\_tree->left->parent = new\_tree;

}

if (right != nullptr) {

new\_tree->right = right->copy();

new\_tree->right->parent = new\_tree;

}

return new\_tree;

}

template <typename T>

Tree<T>\* Tree<T>::search\_all(T key) {

if (data == key) {

return this;

}

if (left != nullptr) {

Tree<T>\* result = left->search\_all(key);

if (result != nullptr) {

return result;

}

}

if (right != nullptr) {

Tree<T>\* result = right->search\_all(key);

if (result != nullptr) {

return result;

}

}

return nullptr;

}

template <typename T>

Tree<T>\* Tree<T>::search\_sort(T data) {

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

return this;

}

else if (data > this->data) {

return this->right->search\_sort(data);

}

else {

return this->left->search\_sort(data);

}

}

template <typename T>

Tree<T>\* Tree<T>::next() {

Tree<T>\* current = this;

if (current->right != nullptr) {

return current->right->find\_min();

}

Tree<T>\* temp = current->parent;

while (temp != nullptr && current == temp->right) {

current = temp;

temp = current->parent;

}

return temp;

}

template <typename T>

Tree<T>\* Tree<T>::prev() {

if (left != nullptr) {

Tree<T>\* temp = left;

while (temp->right != nullptr) {

temp = temp->right;

}

return temp;

}

else {

Tree<T>\* temp = this;

while (temp->parent != nullptr && temp->parent->left == temp) {

temp = temp->parent;

}

return temp->parent;

}

}

template <typename T>

Tree<T>\* Tree<T>::find\_min() {

Tree<T>\* buffer = this->create\_search\_tree();

return buffer->find\_min\_helper();

}

template <typename T>

Tree<T>\* Tree<T>::find\_min\_helper() {

if (this->left == nullptr) {

return this;

}

return this->left->find\_min();

}

template <typename T>

Tree<T>\* Tree<T>::find\_max() {

if (this->right == nullptr) {

return this;

}

return this->right->find\_max();

}

template <typename T>

Tree<T>\* Tree<T>::balanced(int count) {

if (count <= 0) {

return nullptr;

}

T data;

std::cout << "Enter value: ";

std::cin >> data;

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

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

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

return temp;

}

template <typename T>

Tree<T>\* Tree<T>::build\_balanced\_bst(const std::vector<T>& vect, int start, int end, bool prevMidR) {

if (start > end) {

return nullptr;

}

int mid = start + (end - start) / 2;

if (prevMidR && (mid + (end - start) % 2 <= end)) mid += (end - start) % 2;

//if (prevMidR) std::cout << "L " << vect[mid] << std::endl;

//else std::cout << "R " << vect[mid] << std::endl;

Tree<T>\* new\_node = new Tree<T>(vect[mid]);

new\_node->left = build\_balanced\_bst(vect, start, mid - 1, true);

new\_node->right = build\_balanced\_bst(vect, mid + 1, end, false);

return new\_node;

}

template <typename T>

void Tree<T>::in\_order\_traversal(std::vector<T>& vect) {

if (left != nullptr) {

left->in\_order\_traversal(vect);

}

vect.push\_back(data);

if (right != nullptr) {

right->in\_order\_traversal(vect);

}

}

template <typename T>

Tree<T>\* Tree<T>::create\_balanced\_tree() {

std::vector<T> sorted\_data;

this->in\_order\_traversal(sorted\_data);

return build\_balanced\_bst(sorted\_data, 0, sorted\_data.size() - 1, false);

}

template <typename T>

Tree<T>\* Tree<T>::create\_search\_tree() {

std::vector<T> sorted\_data;

this->in\_order\_traversal(sorted\_data);

sort(sorted\_data.begin(), sorted\_data.end());

return build\_balanced\_bst(sorted\_data, 0, sorted\_data.size() - 1, false);

}

template <typename T>

void Tree<T>::DrawT()

{

int height = getHeight();

int radius = 80 / height;

int heightDifference = (radius \* 2) + 10;

sf::RenderWindow window(sf::VideoMode(pow(2, height) \* (radius + 10) \* 2 ,

(height \* heightDifference) + radius), "Binary Tree");

window.clear(sf::Color(128, 128, 128));

DrawElement(this, pow(2, height) \* (radius + 10), 1, height, radius \* 3, radius, window);

window.display();

while (window.isOpen())

{

sf::Event ev;

while (window.pollEvent(ev))

{

if (ev.type == sf::Event::Closed)

window.close();

}

}

}

template <typename T>

void Tree<T>::DrawElement(Tree<T>\* tree, int xpos, int curLevel, int totalLevel, int shift, int radius, sf::RenderWindow& window)

{

sf::Vector2f position = { static\_cast<float>(xpos), static\_cast<float>((radius \* 2) + (radius \* 2 + 10) \* (curLevel - 1)) };

sf::CircleShape circle(static\_cast<float>(radius));

circle.setFillColor(sf::Color::White);

circle.setOutlineColor(sf::Color::Black);

circle.setOutlineThickness(3);

circle.setPosition(position.x - static\_cast<float>(radius), position.y - static\_cast<float>(radius));

T element = tree->get\_data();

sf::Text text;

std::ostringstream buffer;

buffer << std::fixed << std::setprecision(1) << element;

sf::Font font;

font.loadFromFile("Arial.TTF");

text.setFont(font);

text.setString(buffer.str());

text.setFillColor(sf::Color::Black);

text.setOutlineColor(sf::Color::White);

text.setCharacterSize(radius \* 0.75);

sf::FloatRect textRect = text.getLocalBounds();

text.setOrigin(textRect.left + textRect.width / 2.0f, textRect.top + textRect.height / 2.0f);

text.setPosition(sf::Vector2f(position.x, position.y));

window.draw(circle);

window.draw(text);

if (tree->left != nullptr) {

sf::VertexArray line(sf::Lines, 2);

position.x -= radius;

line[0].position = position;

position.x += radius;

line[1].position = { static\_cast<float>(xpos - shift \* (totalLevel - curLevel)), static\_cast<float>((radius)+(radius \* 2 + 10) \* (curLevel)) };

window.draw(line);

DrawElement(tree->left, xpos - shift \* (totalLevel - curLevel), curLevel + 1, totalLevel, shift, radius, window);

}

if (tree->right != nullptr) {

sf::VertexArray line(sf::Lines, 2);

position.x += radius;

line[0].position = position;

position.x -= radius;

line[1].position = { static\_cast<float>(xpos + shift \* (totalLevel - curLevel)), static\_cast<float>((radius)+(radius \* 2 + 10) \* (curLevel)) };

window.draw(line);

DrawElement(tree->right, xpos + shift \* (totalLevel - curLevel), curLevel + 1, totalLevel, shift, radius, window);

}

}

Main.cpp

#include "Tree.h"

#include <iostream>

int main() {

int intBuff;

int command = 1;

Tree<double>\* operTree = new Tree<double>();

Tree<double>\* bufferTree = operTree;

do

{

std::cout << std::endl;

std::cout << "Current possition value: " << bufferTree->get\_data() << std::endl;

std::cout << "Current parrent: " << bufferTree->get\_parent() << std::endl;

std::cout << "Current left subtree: " << bufferTree->get\_left() << std::endl;

std::cout << "Current right subtree: " << bufferTree->get\_right() << std::endl;

std::cout << std::endl;

std::cout << "1. Change current value" << std::endl;

std::cout << "2. Swap to parrent" << std::endl;

std::cout << "3. Swap to left subtree" << std::endl;

std::cout << "4. Swap to right subtree" << std::endl;

std::cout << "5. Return to root" << std::endl;

std::cout << "6. Insert left subtree" << std::endl;

std::cout << "7. Insert right subtree" << std::endl;

std::cout << "8. View tree" << std::endl;

std::cout << "9. Change to balanced tree" << std::endl;

std::cout << "10. Change to search tree" << std::endl;

std::cout << "11. Find element with lowest value" << std::endl;

std::cout << "12. Visual interpretation" << std::endl;

std::cout << "0. Stop program execution" << std::endl;

std::cout << "Choose command to execute: ";

std::cin >> command;

switch (command)

{

case 1:

std::cout << "Enter count of values for binary tree: ";

std::cin >> intBuff;

bufferTree->set\_data(intBuff);

break;

case 2:

if (bufferTree->get\_parent() != nullptr) bufferTree = bufferTree->get\_parent();

break;

case 3:

if (bufferTree->get\_left() != nullptr) bufferTree = bufferTree->get\_left();

break;

case 4:

if (bufferTree->get\_right() != nullptr) bufferTree = bufferTree->get\_right();

break;

case 5:

bufferTree = operTree;

break;

case 6:

std::cout << "Enter value to insert as left subtree: ";

std::cin >> intBuff;

bufferTree->insert\_left(intBuff);

break;

case 7:

std::cout << "Enter value to insert as right subtree: ";

std::cin >> intBuff;

bufferTree->insert\_right(intBuff);

break;

case 8:

std::cout << "Direct view: ";

operTree->view\_direct();

std::cout << std::endl;

std::cout << "Symetric view: ";

operTree->view\_symmetric();

std::cout << std::endl;

std::cout << "Reverse view: ";

operTree->view\_reverse();

std::cout << std::endl;

break;

case 9:

operTree = operTree->create\_balanced\_tree();

bufferTree = operTree;

break;

case 10:

operTree = operTree->create\_search\_tree();

bufferTree = operTree;

break;

case 11:

std::cout << "Adress of element: " << operTree->find\_min() << std::endl << "Element value: " << operTree->find\_min()->get\_data() << std::endl;

break;

case 12:

operTree->print\_vert();

operTree->DrawT();

break;

default:

break;

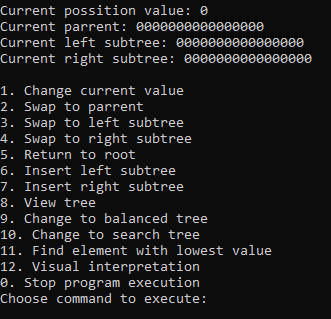
}

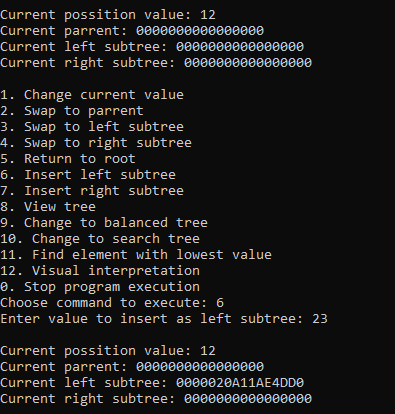
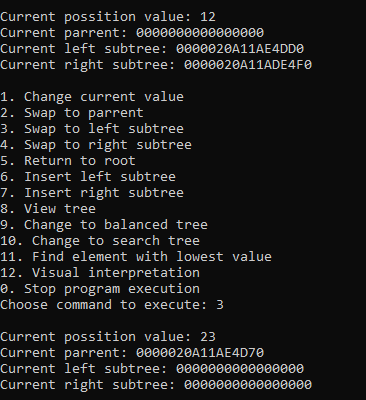
} while (command != 0);

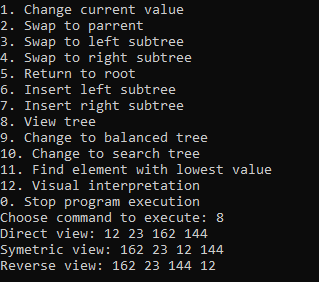
return 0;

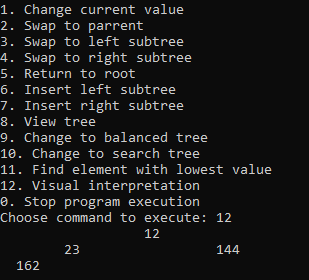
}

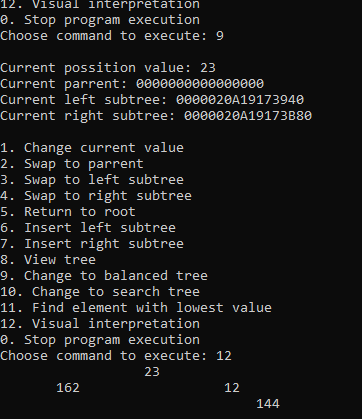
**Решение**

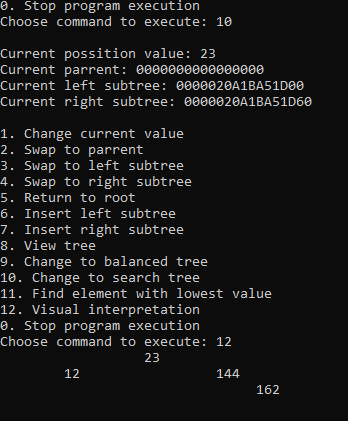


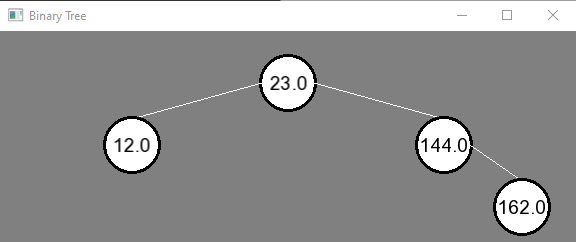
 







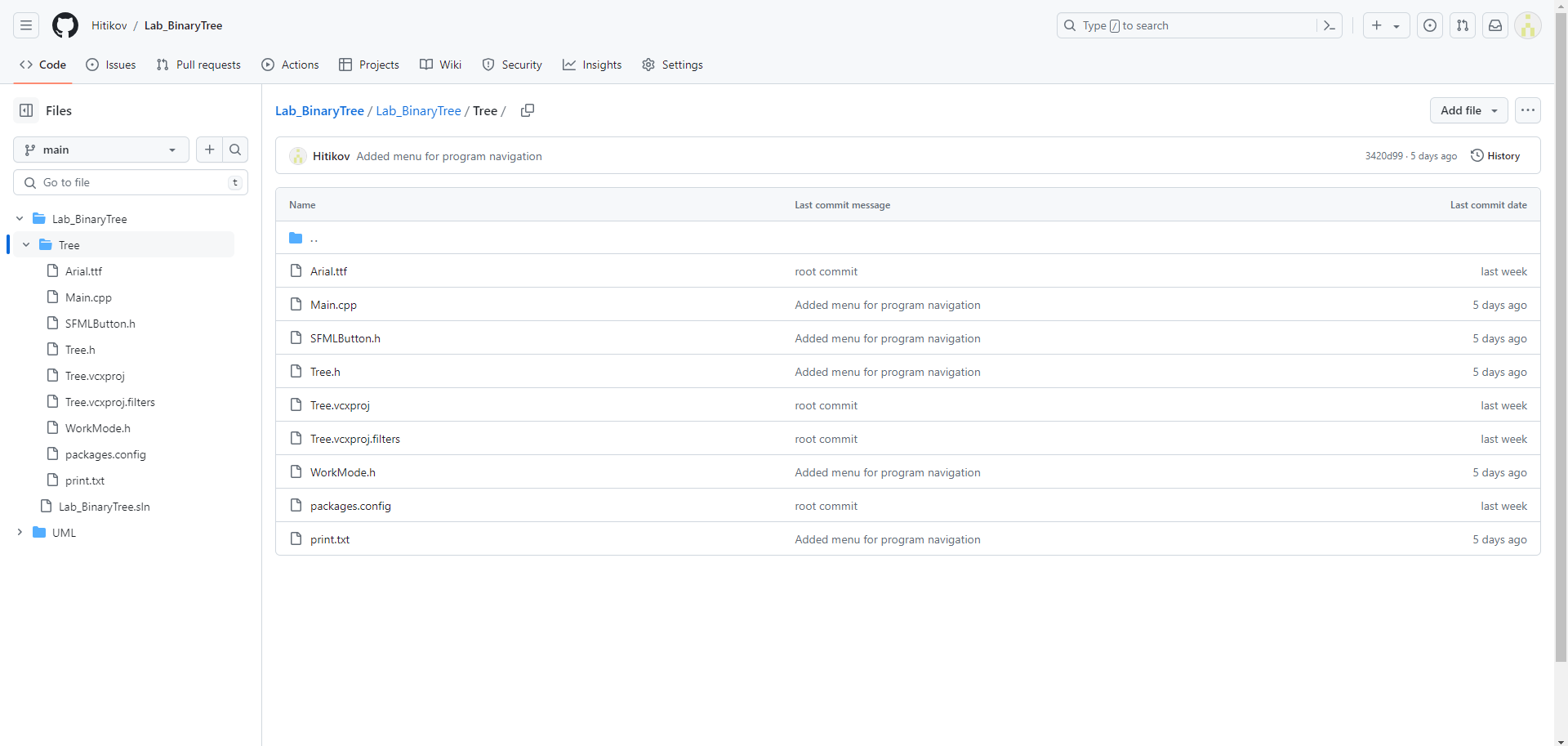




**Выводы**

В ходе выполнения работы были изучены бинарные деревья, их разновидности и методы их обработки.

**Github**



<https://github.com/Hitikov/Lab_BinaryTree>