Министерство науки и высшего образования Российской Федерации федеральное государственное бюджетное образовательное учреждение высшего образования  
Московский авиационный институт  
(Национальный исследовательский университет)

Институт №3  
«Системы управления, информатика и электроэнергетика»  
Кафедра 304

**Отчёт по лабораторной работе**  
  
по учебной дисциплине «Бинарные деревья поиска»

Исполнитель:  
обучающиеся группы М3О-207Б-23  
Ильин А.А.  
Романов Д.И.

Проверили:  
Дмитриева Е.А.

Ивашенцев И. В.

Москва 2024

Оглавление

[Задание 3](#_Toc185187221)

[Код программы 4](#_Toc185187222)

[Результат работы 15](#_Toc185187223)

[Вывод 19](#_Toc185187224)

## Задание

**Задание**

1. Случайным образом сгенерировать массив размерностью 20-25 элементов, повторные значения не допустимы.
2. Реализовать функции вставки, поиска, удаления узла, обхода дерева, вывода дерева на экран, нахождения высоты дерева и количества узлов.
3. Реализовать дополнительно функцию в соответствии с вариантом: *T* – тип ключей, *D* – диапазон изменения значений ключей.
4. Для набора значений из пункта 1 построить рандомизированное дерево, сравнить высоты бинарного и рандомизированного дерева.

|  |  |  |  |
| --- | --- | --- | --- |
| № | *T* | *D* | Функция |
| 9 | **int** | [1; 90] | Определить два минимальных элемента, два максимальных элемента. |

## Код программы

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* КАФЕДРА № 304 2 КУРС \*

\*-----------------------------------------------------------------\*

\* Project Type : Win32 Console Application \*

\* Project Name : BinarySearchTrees \*

\* File Name : main.cpp \*

\* Language : C/C++ \*

\* Programmer(s) : Романов Д.И., Ильин А.А \*

\* Modified By : \*

\* Created : 20/09/2024 \*

\* Last Revision : 15/11/2024 \*

\* Comment(s) : Бинарные деревья поиска \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include "binary\_tree.h"

#include "helpers.h"

#include "random\_tree.h"

const int ARR\_LEN = 24;

const int ARR\_MIN = 1;

const int ARR\_MAX = 50;

int main() {

std::cout << "Лабораторная работа №2 - Линейные списки\n";

std::cout << " Вариант №9\n";

// auto vec = generate\_array(ARR\_LEN, ARR\_MIN, ARR\_MAX);

auto vec = {31, 33, 11, 5, 13, 39, 29, 23, 7, 27, 19, 41, 1, 3, 25, 47, 21, 35, 43, 45, 9, 15, 17, 37};

std::cout << "Изначальный массив: ";

for (auto val: vec) {

std::cout << val << ' ';

}

std::cout << '\n';

srand(1);

btree::node \*binary\_tree{nullptr};

for (auto val: vec) {

binary\_tree = btree::insert(binary\_tree, val);

btree::print\_tree(binary\_tree);

}

std::cout << "Бинарное дерево поиска. Высота: " << btree::height(binary\_tree) << "\n";

btree::print\_tree(binary\_tree);

srand(1);

rtree::node \*random\_tree{nullptr};

for (auto val: vec) {

random\_tree = rtree::insert(random\_tree, val);

}

std::cout << "Случайное бинарное дерево поиска. Высота: " << rtree::height(random\_tree) << "\n";

rtree::print\_tree(random\_tree);

return 0;

}

#include "binary\_tree.h"

#include <iostream>

#include <limits>

#include <stack>

#include <queue>

namespace btree {

node \*insert(node \*root, int key) {

node \*cur\_node = root;

node \*parent{};

// traverse down the tree finding a suitable place for a new node

while (cur\_node != nullptr) {

// save the parent to later link to it

parent = cur\_node;

if (key <= cur\_node->key) {

cur\_node = cur\_node->left;

} else if (key > cur\_node->key) {

cur\_node = cur\_node->right;

} else {

std::cout << "Duplicate key: " << key << "!\n";

return root;

}

}

if (parent == nullptr) {

// the tree was empty

return new node(key);

}

if (key <= parent->key) {

parent->left = new node(key);

} else {

parent->right = new node(key);

}

return root;

}

node \*erase(node \*root, int key) {

node \*cur\_node = root;

node \*parent{};

while (cur\_node != nullptr) {

if (key == cur\_node->key) {

break; // found the node we need to delete

}

parent = cur\_node;

if (key < cur\_node->key) {

cur\_node = cur\_node->left;

} else {

cur\_node = cur\_node->right;

}

}

if (cur\_node == nullptr) {

// not found

return root;

}

if (cur\_node->left != nullptr && cur\_node->right != nullptr) {

// there are two children. we'll replace current node with

// the smallest node in the right subtree, then delete it

// because it will have at most one child (left one is nullptr)

// , we can just use the code below to get rid of it

node \*successor\_parent = cur\_node;

node \*successor = cur\_node->right;

while (successor->left != nullptr) {

successor\_parent = successor;

successor = successor->left;

}

cur\_node->key = successor->key;

cur\_node = successor;

parent = successor\_parent;

}

// there is only one child (if there were two, we are now deleting its successor, which has at most one)

// we'll need to reconnect this child to the parent node

node \*replacement\_child;

if (cur\_node->left != nullptr) {

replacement\_child = cur\_node->left;

} else {

replacement\_child = cur\_node->right;

}

if (parent == nullptr) {

// the node to delete is root. make its child the new root

root = replacement\_child;

} else if (cur\_node->left == replacement\_child) {

// reconnect the parent's left node

parent->left = replacement\_child;

} else {

// reconnect the parent's right node

parent->right = replacement\_child;

}

delete cur\_node; // finally

return root;

}

void clear(node \*root) {

if (root == nullptr) {

return;

}

// copied from btree::print\_postorder. refer to that for comments

std::stack<node \*> stack{};

node \*cur\_node = root;

node \*last\_visited = nullptr;

while (cur\_node != nullptr || !stack.empty()) {

while (cur\_node != nullptr) {

stack.push(cur\_node);

cur\_node = cur\_node->left;

}

node \*parent\_node = stack.top();

if (parent\_node->right != nullptr && last\_visited != parent\_node->right) {

cur\_node = parent\_node->right;

} else {

delete parent\_node;

last\_visited = parent\_node;

stack.pop();

}

}

}

node \*search(node \*root, int key) {

while (root != nullptr) {

if (key < root->key) {

root = root->left;

} else if (key > root->key) {

root = root->right;

} else {

// key == root->key

return root;

}

}

return nullptr; // not found

}

int height(node \*root) {

if (root == nullptr) {

return 0;

}

// level-order traversal (breadth-first search)

std::queue<node \*> queue{};

queue.push(root);

int height = 0;

int nodes\_cur\_level = 1; // only root

int nodes\_next\_level = 0; // unknown

while (!queue.empty()) {

node \*cur\_node = queue.front();

queue.pop();

nodes\_cur\_level--; // processed a new node

if (cur\_node->left != nullptr) {

queue.push(cur\_node->left);

nodes\_next\_level++;

}

if (cur\_node->right != nullptr) {

queue.push(cur\_node->right);

nodes\_next\_level++;

}

if (nodes\_cur\_level == 0) {

// all nodes at current level are processed

height++;

// advance current level

nodes\_cur\_level = nodes\_next\_level;

nodes\_next\_level = 0;

}

}

return height;

}

int node\_count(node \*root) {

if (root == nullptr) {

return 0;

}

// preorder traversal (depth-first search)

std::stack<node \*> stack{};

stack.push(root);

int count = 0;

while (!stack.empty()) {

node \*cur\_node = stack.top();

stack.pop();

count++;

if (cur\_node->right != nullptr) {

stack.push(cur\_node->right);

}

if (cur\_node->left != nullptr) {

stack.push(cur\_node->left);

}

}

return count;

}

std::tuple<node \*, node \*, node \*, node \*> min\_max\_2(node \*root) {

if (root == nullptr) {

return {nullptr, nullptr, nullptr, nullptr};

}

node \*minimum = root, \*minimum\_2{}, \*maximum\_2{}, \*maximum = root;

// find minimal (leftmost) element and it's parent

while (minimum->left != nullptr) {

minimum\_2 = minimum;

minimum = minimum->left;

}

// the parent could be second to min, but

// if this minimal element has a right subtree

if (minimum->right != nullptr) {

// then the parent is bigger than the whole subtree

// we need to find this subtree's leftmost element

// it will be second to minimum

minimum\_2 = minimum->right;

while (minimum\_2->left != nullptr) {

minimum\_2 = minimum\_2->left;

}

}

// find maximal (rightmost) element and it's parent

while (maximum->right != nullptr) {

maximum\_2 = maximum;

maximum = maximum->right;

}

// the parent could be second to max, but

// if this maximal element has a left subtree

if (maximum->left != nullptr) {

// then the parent is smaller than the whole subtree

// we need to find this subtree's rightmost element

// it will be second to maximum

maximum\_2 = maximum->left;

while (maximum\_2->right != nullptr) {

maximum\_2 = maximum\_2->right;

}

}

return {minimum, minimum\_2, maximum\_2, maximum};

}

void print\_preorder(node \*root) {

if (root == nullptr) {

return;

}

// preorder traversal (depth-first search)

std::stack<node \*> stack{};

stack.push(root);

while (!stack.empty()) {

node \*cur\_node = stack.top();

stack.pop();

// do whatever you need with the node. here we're just printing it

std::cout << cur\_node->key << ' ';

// we're putting the right first since the next iteration grabs the last element (left subtree)

if (cur\_node->right != nullptr) {

stack.push(cur\_node->right);

}

if (cur\_node->left != nullptr) {

stack.push(cur\_node->left);

}

}

std::cout << '\n';

}

void print\_inorder(node \*root) {

if (root == nullptr) {

return;

}

// inorder traversal (depth-first search)

std::stack<node \*> stack{};

node \*cur\_node = root;

// reach the left subtree, then its parent, then the right subtree

while (cur\_node != nullptr || !stack.empty()) {

// push nodes onto stack until we reach the leftmost element

while (cur\_node != nullptr) {

stack.push(cur\_node);

cur\_node = cur\_node->left;

}

// get it

cur\_node = stack.top();

stack.pop();

// we've reached the leftmost element in this subtree that we haven't traversed before

// do whatever you need with the node. here we're just printing it

std::cout << cur\_node->key << ' ';

// next we're advancing into this element's right subtree (if it exists)

// if it doesn't exist cur\_node will be null, and we'll just pop a parent off the stack

cur\_node = cur\_node->right;

}

std::cout << '\n';

}

void print\_postorder(node \*root) {

if (root == nullptr) {

return;

}

// postorder traversal (depth-first search)

std::stack<node \*> stack{};

node \*cur\_node = root;

node \*last\_visited = nullptr; // we need it to determine from which subtree we're ascending to the parent

while (cur\_node != nullptr || !stack.empty()) {

// push nodes onto stack until we reach the leftmost element

while (cur\_node != nullptr) {

stack.push(cur\_node);

cur\_node = cur\_node->left;

}

node \*parent\_node = stack.top();

// here it gets tricky

// if right child exists, and we're ascending from the left child

if (parent\_node->right != nullptr && last\_visited != parent\_node->right) {

// then advance into this element's right subtree (if it exists)

cur\_node = parent\_node->right;

} else {

// else the right child doesn't exist, or we're ascending from the right child

// either way we need to visit this parent node and move to it's parent

// do whatever you need with the node. here we're just printing it

std::cout << parent\_node->key << ' ';

last\_visited = parent\_node;

stack.pop();

}

}

std::cout << '\n';

}

void print\_levelorder(node \*root) {

if (root == nullptr) {

return;

}

// level-order traversal (breadth-first search)

std::queue<node \*> queue{};

queue.push(root);

while (!queue.empty()) {

node \*cur\_node = queue.front();

queue.pop();

// do whatever you need with the node. here we're just printing it

std::cout << cur\_node->key << ' ';

if (cur\_node->left != nullptr) {

queue.push(cur\_node->left);

}

if (cur\_node->right != nullptr) {

queue.push(cur\_node->right);

}

}

std::cout << '\n';

}

void print\_tree(const node \*node, const std::string &prefix, bool is\_left, bool is\_end) {

if (node == nullptr)

return;

std::cout << prefix;

std::cout << (is\_end ? "└" : "├") << (is\_left ? "l" : "r") << "─";

// print the value of the node

std::cout << node->key << std::endl;

// enter the next tree level - left and right branch

print\_tree(node->left, prefix + (is\_end ? " " : "│ "), true, node->right == nullptr);

print\_tree(node->right, prefix + (is\_end ? " " : "│ "), false, true);

}

} // btree

#include "helpers.h"

#include <algorithm>

std::vector<int> generate\_array(int length, int min, int max) {

std::vector<int> vec(length);

int step = (max - min) / length;

for (int i = 0, num = min; i < length; i++, num += step) {

vec[i] = num;

}

std::shuffle(vec.begin(), vec.end(), std::default\_random\_engine { std::random\_device()() });

return vec;

}

#include "random\_tree.h"

#include <iostream>

#include <limits>

#include <stack>

#include <queue>

namespace rtree {

void fix\_size(node \*p) {

if (!p) return;

p->size = 1 + (p->left ? p->left->size : 0) + (p->right ? p->right->size : 0);

}

node \*rotate\_right(node \*old\_root)

{

node \*new\_root = old\_root->left;

if (!new\_root) return old\_root;

old\_root->left = new\_root->right;

new\_root->right = old\_root;

new\_root->size = old\_root->size;

fix\_size(old\_root);

fix\_size(new\_root);

return new\_root;

}

node \*rotate\_left(node \*old\_root)

{

node \*new\_root = old\_root->right;

if (!new\_root) return old\_root;

old\_root->right = new\_root->left;

new\_root->left = old\_root;

new\_root->size = old\_root->size;

fix\_size(old\_root);

fix\_size(new\_root);

return new\_root;

}

#pragma clang diagnostic push

#pragma ide diagnostic ignored "misc-no-recursion"

node \*insert\_root(node \*root, int k)

{

if (!root) return new node(k);

if (k < root->key) {

root->left = insert\_root(root->left, k);

return rotate\_right(root);

} else {

root->right = insert\_root(root->right, k);

return rotate\_left(root);

}

}

node \*insert(node \*root, int k)

{

if (!root) return new node(k);

if (rand() % (root->size + 1) == 0) {

return insert\_root(root, k);

}

if (k < root->key) {

root->left = insert(root->left, k);

}

else {

root->right = insert(root->right, k);

}

fix\_size(root);

return root;

}

#pragma clang diagnostic pop

void clear(node \*root) {

if (root == nullptr) {

return;

}

// copied from btree::print\_postorder. refer to that for comments

std::stack<node \*> stack{};

node \*cur\_node = root;

node \*last\_visited = nullptr;

while (cur\_node != nullptr || !stack.empty()) {

while (cur\_node != nullptr) {

stack.push(cur\_node);

cur\_node = cur\_node->left;

}

node \*parent\_node = stack.top();

if (parent\_node->right != nullptr && last\_visited != parent\_node->right) {

cur\_node = parent\_node->right;

} else {

delete parent\_node;

last\_visited = parent\_node;

stack.pop();

}

}

}

int height(node \*root) {

if (root == nullptr) {

return 0;

}

// level-order traversal (breadth-first search)

// copied from btree::height

std::queue<node \*> queue{};

queue.push(root);

int height = 0;

int nodes\_cur\_level = 1; // only root

int nodes\_next\_level = 0; // unknown

while (!queue.empty()) {

node \*cur\_node = queue.front();

queue.pop();

nodes\_cur\_level--; // processed a new node

if (cur\_node->left != nullptr) {

queue.push(cur\_node->left);

nodes\_next\_level++;

}

if (cur\_node->right != nullptr) {

queue.push(cur\_node->right);

nodes\_next\_level++;

}

if (nodes\_cur\_level == 0) {

// all nodes at current level are processed

height++;

// advance the current level

nodes\_cur\_level = nodes\_next\_level;

nodes\_next\_level = 0;

}

}

return height;

}

void print\_tree(const node \*node, const std::string &prefix, bool is\_left, bool is\_end) {

if (node == nullptr)

return;

std::cout << prefix;

std::cout << (is\_end ? "└" : "├") << (is\_left ? "l" : "r") << "─";

// print the value of the node

std::cout << node->key << ' ' << node->size << std::endl;

// enter the next tree level - left and right branch

print\_tree(node->left, prefix + (is\_end ? " " : "│ "), true, node->right == nullptr);

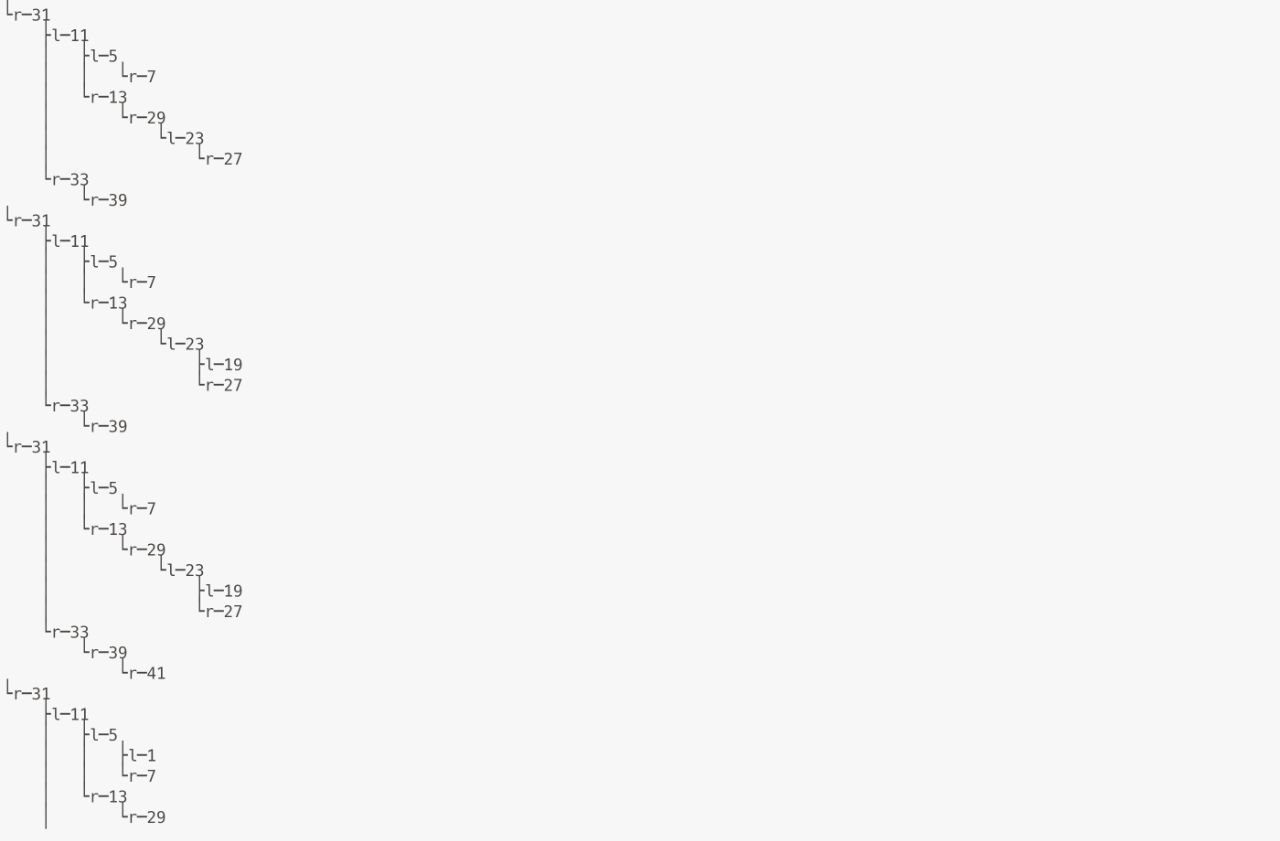
print\_tree(node->right, prefix + (is\_end ? " " : "│ "), false, true);

}

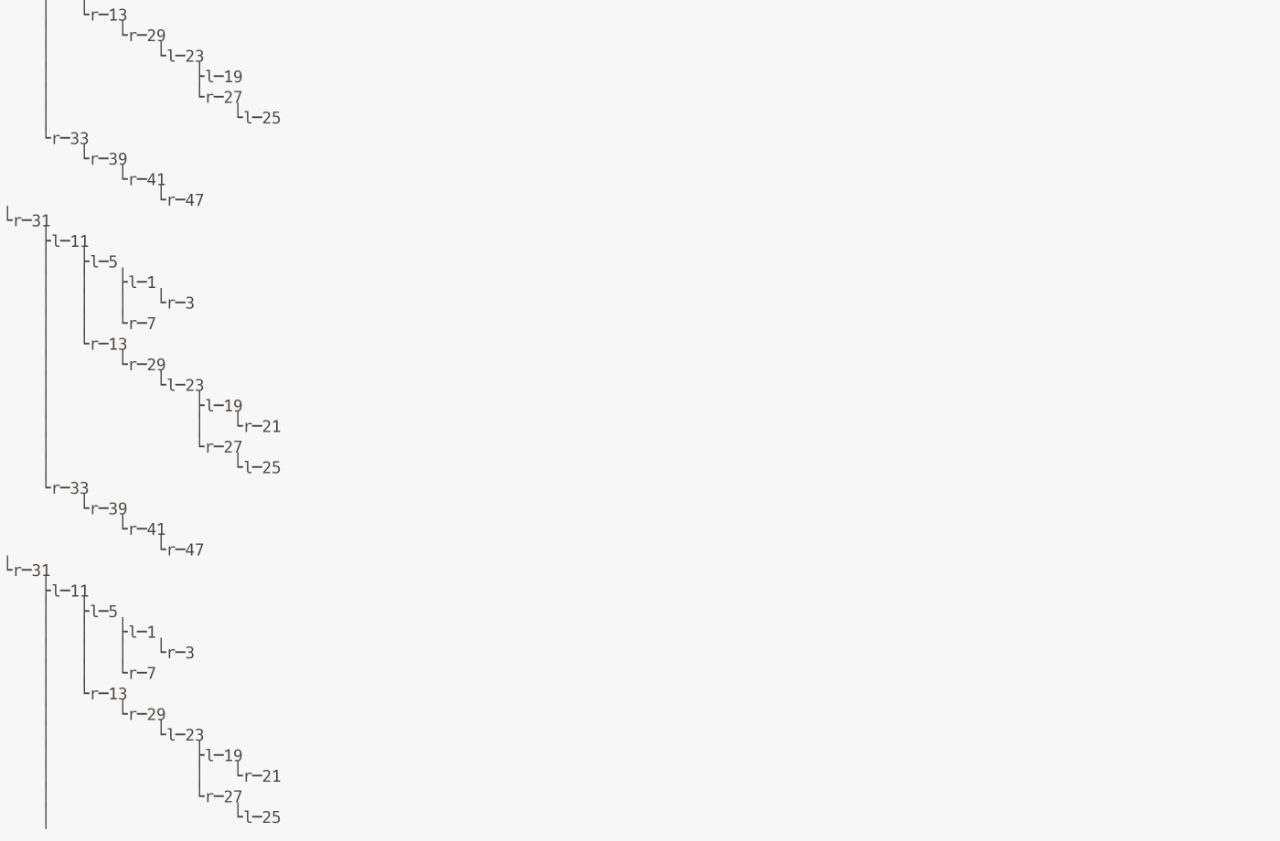
} // tree

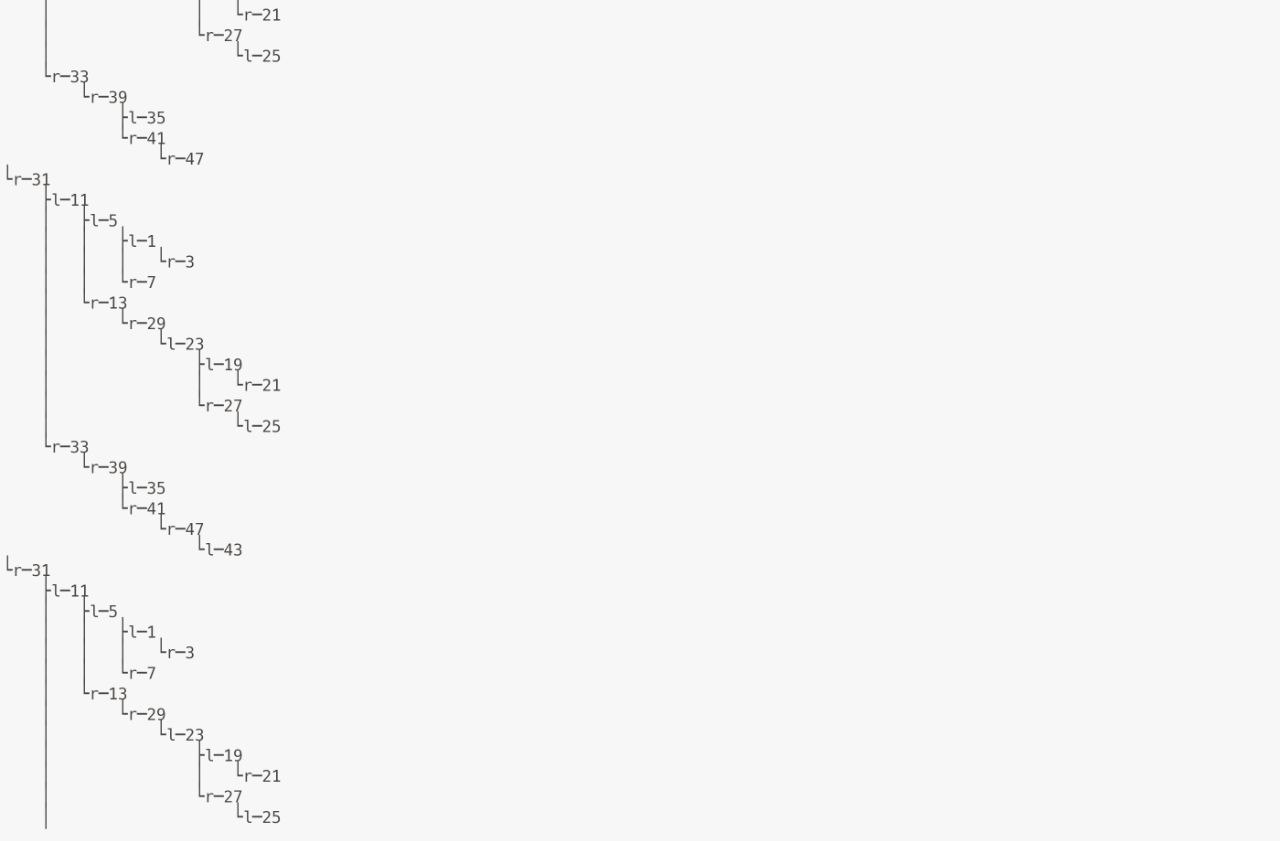
## Результат работы





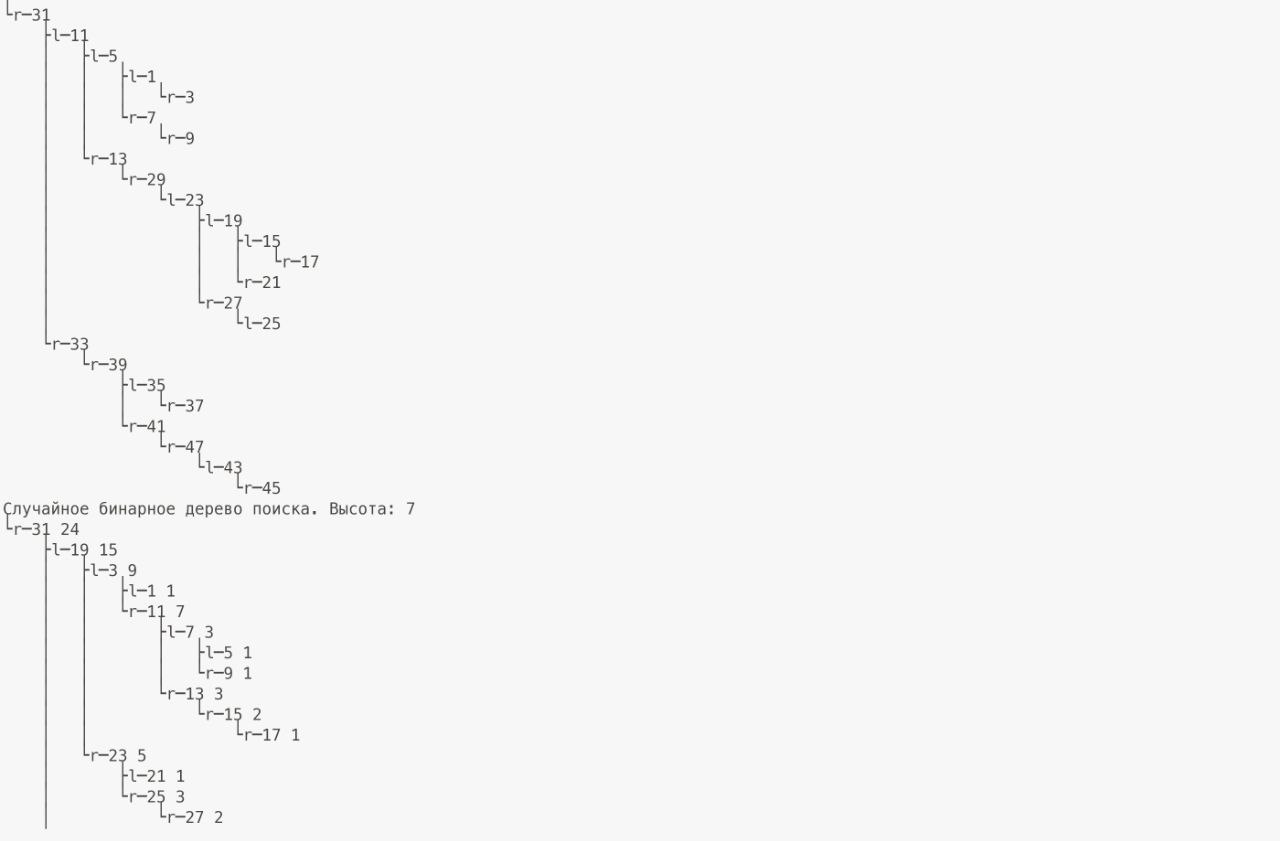
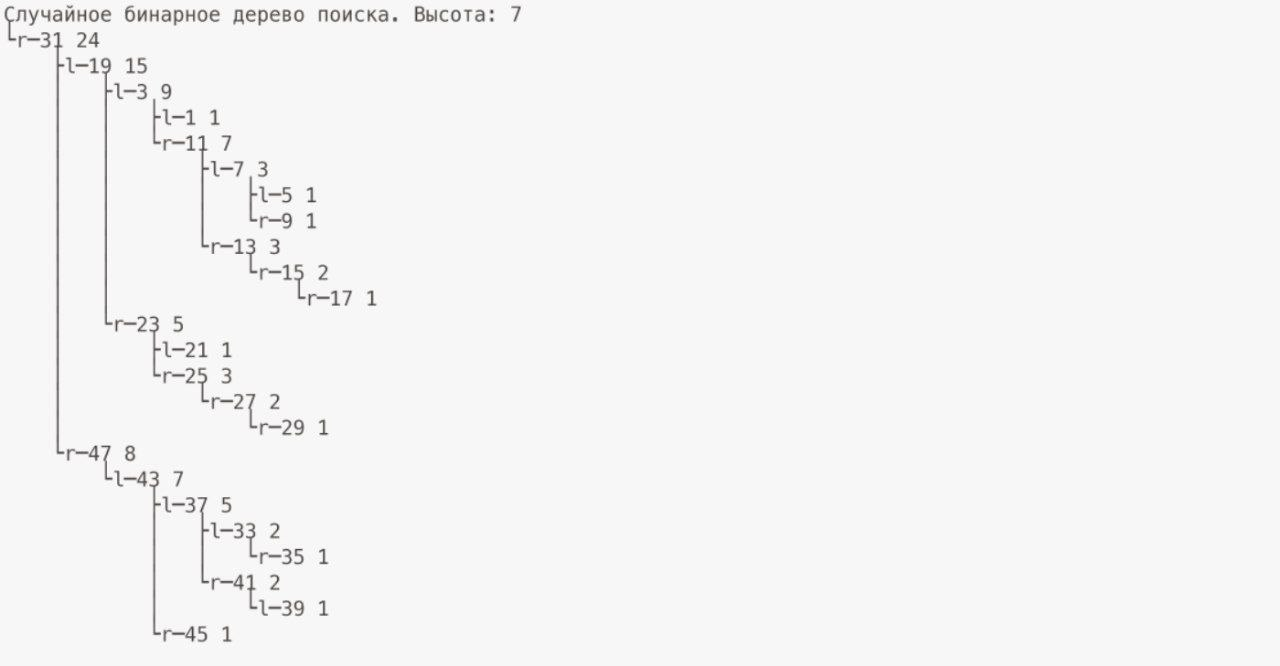










## Вывод

В ходе данной лабораторной работы мы на практике реализовали бинарное дерево поиска и случайное бинарное дерево поиска. Сравнив их высоты, выяснили, что у случайного дерева поиска она меньше (в тестовом случае).