Балтийский государственный технический университет  
«ВОЕНМЕХ» им. Д. Ф. Устинова

Кафедра О7 «Информационные системы и программная инженерия»

**Практическая работа №3**по дисциплине «Структуры и организация данных»  
на тему «Оценка эффективности алгоритмов»  
часть 2 «Алгоритмы поиска»  
  
вариант 12

Выполнил:  
Студент Костров Г. Ю.  
Группа 0712Б  
  
Преподаватель:  
Палехова О. А.

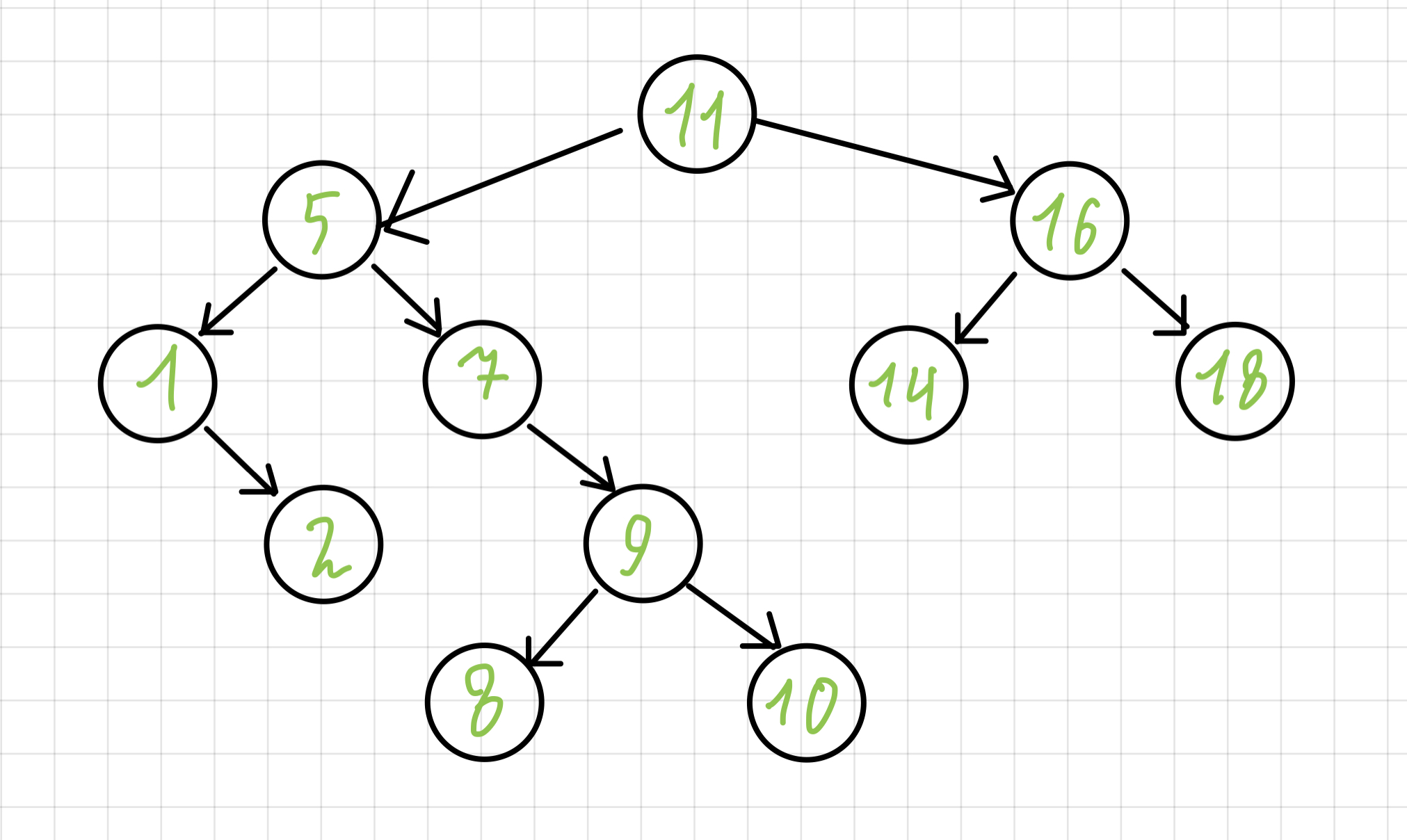
Санкт-Петербург  
2022 г.

Произвести сравнение указанных в вариативной части структур данных по пространственной сложности и вычислительной сложности поиска.

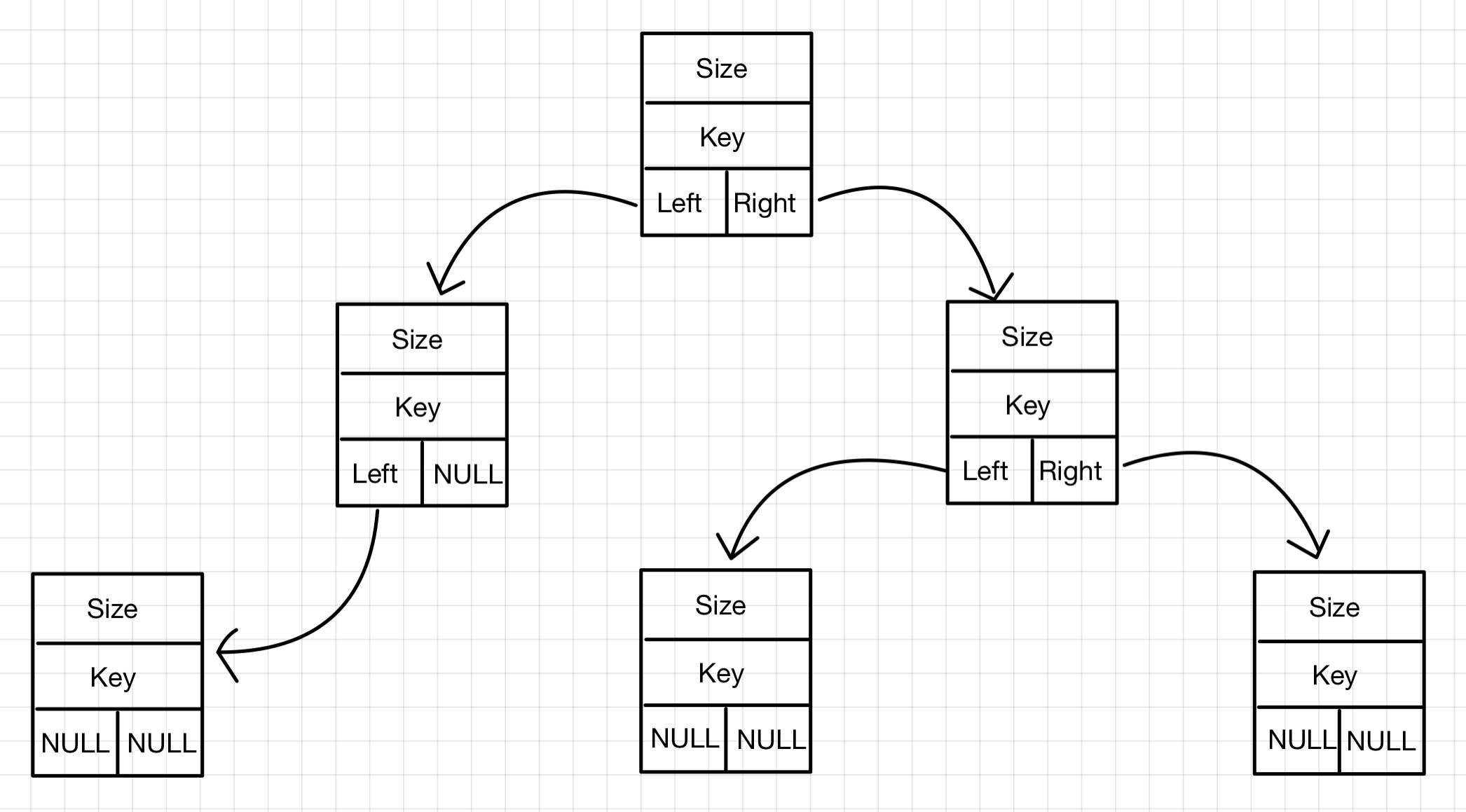
Выполнить теоретические расчеты сложности поиска и занимаемого структурой объема памяти. Сформировать корректный тестовый набор из 100 ключей. Реализовать две указанные структуры данных, заполнив их значениями из приложенного файла *test\_numbers.txt*. Выполнить поиск 100 ключей в указанных структурах данных, для каждого ключа выводить сообщение о том, найден он или нет, и количество выполненных при поиске сравнений ключей, в конце программы вывести среднее количество сравнений, пришедшееся на один ключ. Произвести анализ полученных теоретических и практических результатов, сравнить поисковые структуры по вычислительной и пространственной сложности и выбрать наиболее эффективную из них.

Вариант № 12 Рандомизированное бинарное дерево поиска, красно-черное дерево.

Схематичное изображение рандомизированного бинарного дерева



Схематичное изображение структуры хранения, использованной в программе для программирования рандомизированного бинарного дерева:

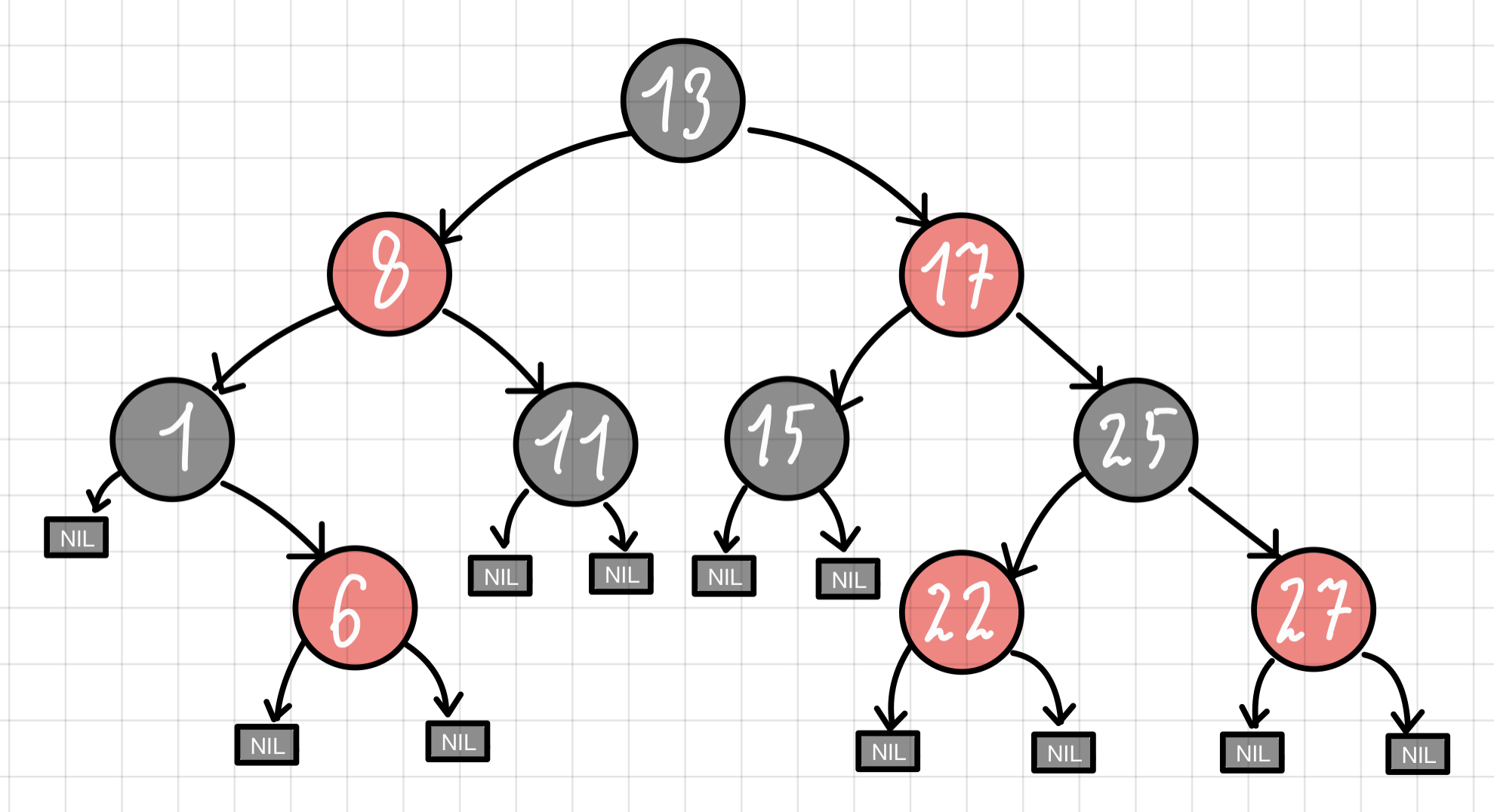


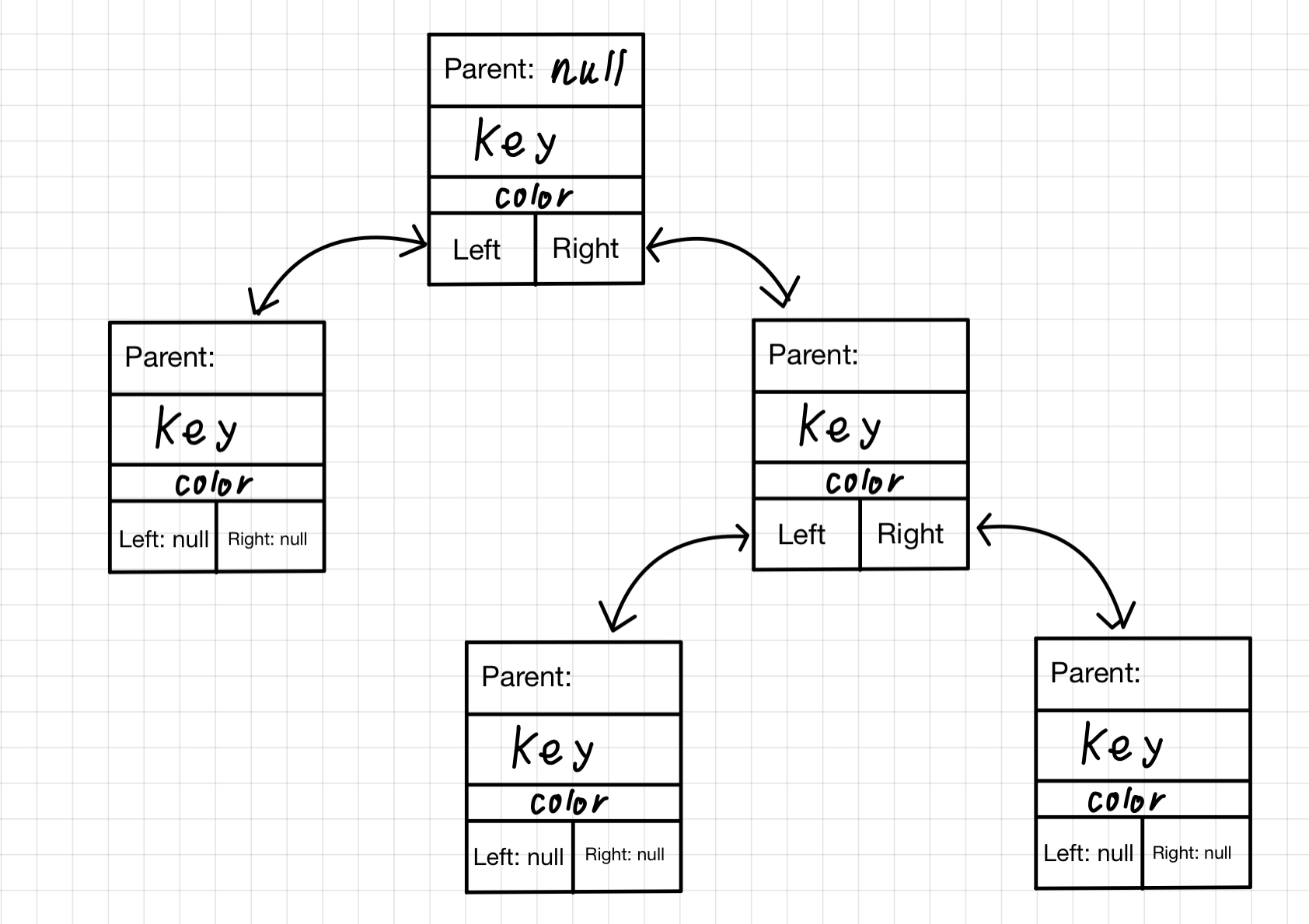
Трудоемкость поиска:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответствующая случаю** | **Обоснование** | **Ожидаемое число сравнений при поиске** | **Асимптотическая оценка сложности поиска** |
| наилучший | искомый ключ находится в корне дерева | требуется выполнить два сравнение | 2 | Ω(1) |
| наихудший | искомый ключ отсутствует в дереве | требуется осуществить перебор ключей на одной ветке дерева  три сравнения на ключ умножить на высоту дерева | 3\*(4log2N) | O(log2N) |
| наиболее вероятный | искомый ключ присутствует в дереве | требуется осуществить перебор ключей на одной ветке дерева  три сравнения на ключ умножить на среднюю высоту дерева | 3\*(2log2N) | O(log2N) |

Требуемый объем памяти (2 \*sizeof(int)+2\*sizeof(pointer))\*N байт.

Схематичное изображение красно-черного дерева:



Схематичное изображение структуры хранения, использованной в программе для программирования красно-черного дерева 

Трудоемкость поиска:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Случай** | **Ситуация, соответствующая случаю** | **Обоснование** | **Ожидаемое число сравнений при поиске** | **Асимптотическая оценка сложности поиска** |
| наилучший | искомый ключ находится в корне дерева | требуется выполнить два сравнение | 2 | Ω(1) |
| наихудший | искомый ключ отсутствует в дереве | требуется осуществить перебор ключей на одной ветке дерева  три сравнения на ключ умножить на высоту дерева | 3 \* 2 \* log2 (N + 1) | O(log2N) |
| наиболее вероятный | искомый ключ присутствует в дереве | требуется осуществить перебор ключей на одной ветке дерева  три сравнения на ключ умножить на среднюю высоту дерева | 3 \* log2(N + 1) | O(log2N) |

Требуемый объем памяти 2\*sizeof(int) + 3\*sizeof(pointer))\*N байт.

Тестовый набор ключей:

присутствующие в файле:

97192338, 90229668, 67343291, 70258238, 83246999, 37801902, 30349033, 19445301, 88278989, 36406281, 35966130, 35557396, 84580155, 83130037, 42315857, 15916697, 52779136, 32051522, 39771580, 20138140, 42947553, 95905677, 62667096, 66541010, 30617946, 36600568, 99020052, 55743512, 83673374, 42882723, 56804407, 66912047, 97986428, 56718265, 27691681, 87238974, 98425454, 34568631, 51935268, 85068842, 84258720, 64275429, 50492010, 24892362, 27648480, 55750064, 52695354, 46375986, 52779136, 28831849

Отсутствующие ключи в файле:

1. 2 числа, которые меньше 10 000 000:

3514288, 6146302

1. 46 чисел, которые входят в промежуток [10 000 000; 100 000 000)

40757147, 47978500, 26971124, 59567120, 11749387, 21470545, 16239572, 36984808, 13615544, 59042088, 18341522, 21564522, 55016433, 13987482, 30090226, 10617624, 74041604, 72908328, 37655783, 88159989, 17379060, 60530496, 40835680, 51597008, 16923810, 10342100, 96546364, 44012332, 62071040, 82755056, 69646416, 82695451, 93557676, 59942653, 47583462, 68791610, 19031778, 23930380, 76547830, 29375581, 18773940, 76607780, 18402970, 19622200, 68026284, 16697010

1. 2 числа, которые больше 100 000 000:

114295659, 106180358

Текст программы:

// randomTree.h

#include "iostream"

#include <stdio.h>

#include <stdlib.h>

#include <iostream>

#include <io.h>

#include <fcntl.h>

#include <locale>

#include <fstream>

using namespace std;

struct element // структура для представления узлов дерева

{

element\* left;

element\* right;

int key;

int size;

element(int k) { key = k; left = right = 0; size = 1; }

};

class randomizeTree{

public:

element \*root;

int comparing = 0;

int totalMemory = 0;

int getComparing() {return comparing;}

int getTotalMemory() {return totalMemory;}

randomizeTree(int \_root) {

this->root = new element(\_root);

}

element\* insert(int);

bool find(int k);

void remove(int k);

};

// обертка для поля size, работает с пустыми деревьями (t=NULL)

int getsize(element\* p) {

if( !p ) return 0;

return p->size;

}

// установление корректного размера дерева

void fixsize(element\* p) {

p->size = getsize(p->left)+getsize(p->right)+1;

}

// правый поворот вокруг узла p

element\* rotateright(element\* p) {

element\* q = p->left;

if( !q ) return p;

p->left = q->right;

q->right = p;

q->size = p->size;

fixsize(p);

return q;

}

// левый поворот вокруг узла q

element\* rotateleft(element\* q) {

element\* p = q->right;

if( !p ) return q;

q->right = p->left;

p->left = q;

p->size = q->size;

fixsize(q);

return p;

}

// вставка нового узла с ключом k в корень дерева p

element\* insertroot(element\* p, int k) {

if( !p ) return new element(k);

if( k<p->key )

{

p->left = insertroot(p->left,k);

return rotateright(p);

}

else

{

p->right = insertroot(p->right,k);

return rotateleft(p);

}

}

// объединение двух деревьев

element\* join(element\* p, element\* q) {

if( !p ) return q;

if( !q ) return p;

if( rand()%(p->size+q->size) < p->size )

{

p->right = join(p->right,q);

fixsize(p);

return p;

}

else

{

q->left = join(p,q->left);

fixsize(q);

return q;

}

}

// рандомизированная вставка нового узла с ключом k в дерево p

element\* \_insert(element\* p, int k) {

if( !p ) return new element(k);

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

p = insertroot(p,k);

return p;

}

if( p->key >k )

p->left = \_insert(p->left,k);

else

p->right = \_insert(p->right,k);

fixsize(p);

return p;

}

// рандомизированная вставка нового узла с ключом k в дерево p

element\* randomizeTree::insert(int k) {

return this->root = \_insert(this->root,k);

}

// поиск ключа k в дереве p

int Main\_second = 0;

element\* \_find(element\* p, int k) {

if( ++Main\_second && !p ) return 0;

if( ++Main\_second && k == p->key ) return p;

if( ++Main\_second && k < p->key ) {

return \_find(p->left,k);

} else{

return \_find(p->right,k);

}

}

// поиск ключа k в дереве p

bool randomizeTree::find(int k) {

bool flag = \_find(this->root,k) != NULL;

this->comparing=Main\_second;

Main\_second=0;

return flag;

}

// удаление из дерева p первого найденного узла с ключом k

element\* \_remove(element\* p, int k) {

if( !p ) return p;

if( p->key==k ) {

element\* q = join(p->left,p->right);

delete p;

return q;

}

else if( k<p->key )

p->left = \_remove(p->left,k);

else

p->right = \_remove(p->right,k);

return p;

}

// удаление из дерева p первого найденного узла с ключом k

void randomizeTree::remove(int k) {

\_remove(this->root,k);

}

// rbTree.h

#include "iostream"

#include "iomanip"

using namespace std;

// структура узла красно черного дерева

struct Node {

Node \*parent;

Node \*left;

Node \*right;

int color; // 1 -> красный, 0 -> черный

int data;

};

typedef Node \*NodePtr;

class RBTree {

private:

NodePtr root;

NodePtr TNULL;

int comparing;

int totalMemory;

NodePtr searchTreeHelper(NodePtr node, int key) {

if ((++comparing && (node == TNULL)) || (++comparing && (key == node->data))) {

return node;

}

if (++comparing && (key < node->data)) {

return searchTreeHelper(node->left, key);

} else {

return searchTreeHelper(node->right, key);

}

}

// восстановление красно-черного дерева после удаления элемента

void fixDelete(NodePtr x) {

NodePtr s;

while (x != root && x->color == 0) {

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

s = x->parent->right;

if (s->color == 1) {

s->color = 0;

x->parent->color = 1;

leftRotate(x->parent);

s = x->parent->right;

}

if (s->left->color == 0 && s->right->color == 0) {

s->color = 1;

x = x->parent;

} else {

if (s->right->color == 0) {

s->left->color = 0;

s->color = 1;

rightRotate(s);

s = x->parent->right;

}

s->color = x->parent->color;

x->parent->color = 0;

s->right->color = 0;

leftRotate(x->parent);

x = root;

}

} else {

s = x->parent->left;

if (s->color == 1) {

s->color = 0;

x->parent->color = 1;

rightRotate(x->parent);

s = x->parent->left;

}

if (s->right->color == 0 && s->right->color == 0) {

s->color = 1;

x = x->parent;

} else {

if (s->left->color == 0) {

s->right->color = 0;

s->color = 1;

leftRotate(s);

s = x->parent->left;

}

s->color = x->parent->color;

x->parent->color = 0;

s->left->color = 0;

rightRotate(x->parent);

x = root;

}

}

}

x->color = 0;

}

void rbTransplant(NodePtr u, NodePtr v){

if (u->parent == nullptr) {

root = v;

} else if (u == u->parent->left){

u->parent->left = v;

} else {

u->parent->right = v;

}

v->parent = u->parent;

}

void deleteNodeHelper(NodePtr node, int key) {

// ищем ключ с нужным значением

NodePtr z = TNULL;

NodePtr x, y;

while (node != TNULL){

if (node->data == key) {

z = node;

}

if (node->data <= key) {

node = node->right;

} else {

node = node->left;

}

}

// если не удалось найти узел

if (z == TNULL) {

cout<<"Не удалось найти узел в дереве"<<endl;

return;

}

y = z;

int y\_original\_color = y->color;

if (z->left == TNULL) {

x = z->right;

rbTransplant(z, z->right);

} else if (z->right == TNULL) {

x = z->left;

rbTransplant(z, z->left);

} else {

y = minimum(z->right);

y\_original\_color = y->color;

x = y->right;

if (y->parent == z) {

x->parent = y;

} else {

rbTransplant(y, y->right);

y->right = z->right;

y->right->parent = y;

}

rbTransplant(z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

delete z;

if (y\_original\_color == 0){

fixDelete(x);

}

}

// правка красно черного дерева

void fixInsert(NodePtr k){

NodePtr u;

// если узел красный

while (k->parent->color == 1) {

if (k->parent == k->parent->parent->right) {

u = k->parent->parent->left; // получаем дядю

// если дядя красный то перекрашиваем его

if (u->color == 1) {

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

} else { // иначе делаем правый поворот

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

k = k->parent;

rightRotate(k);

}

k->parent->color = 0;

k->parent->parent->color = 1;

leftRotate(k->parent->parent);

}

} else {

u = k->parent->parent->right; // получаем дядю

if (u->color == 1) {

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

} else {

if (k == k->parent->right) {

k = k->parent;

leftRotate(k);

}

k->parent->color = 0;

k->parent->parent->color = 1;

rightRotate(k->parent->parent);

}

}

if (k == root) {

break;

}

}

root->color = 0;

}

public:

RBTree() {

TNULL = new Node;

TNULL->color = 0;

TNULL->left = nullptr;

TNULL->right = nullptr;

root = TNULL;

comparing = 0;

totalMemory = 0;

totalMemory += sizeof(Node) + sizeof(root);

}

// поиск узла по ключу

NodePtr searchTree(int k) {

return searchTreeHelper(this->root, k);

}

// ищем узел с минимальным значением, для этого просто спускаемся влево

NodePtr minimum(NodePtr node) {

while (node->left != TNULL) {

node = node->left;

}

return node;

}

// ищем узел с максимальным значением, для этого просто спускаемся вправо

NodePtr maximum(NodePtr node) {

while (node->right != TNULL) {

node = node->right;

}

return node;

}

// левый поворот

void leftRotate(NodePtr x) {

NodePtr y = x->right;

x->right = y->left;

if (y->left != TNULL) {

y->left->parent = x;

}

y->parent = x->parent;

if (x->parent == nullptr) {

this->root = y;

} else if (x == x->parent->left) {

x->parent->left = y;

} else {

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

// правый поворот

void rightRotate(NodePtr x) {

NodePtr y = x->left;

x->left = y->right;

if (y->right != TNULL) {

y->right->parent = x;

}

y->parent = x->parent;

if (x->parent == nullptr) {

this->root = y;

} else if (x == x->parent->right) {

x->parent->right = y;

} else {

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

// вставка ключа и восстановление требований красно-черного дерева

void insert(int key) {

NodePtr node = new Node;

node->parent = nullptr;

node->data = key;

node->left = TNULL;

node->right = TNULL;

node->color = 1;

NodePtr y = nullptr;

NodePtr x = this->root;

totalMemory += sizeof(Node);

while (x != TNULL) {

y = x;

if (node->data < x->data) {

x = x->left;

} else {

x = x->right;

}

}

node->parent = y;

if (y == nullptr) {

root = node;

} else if (node->data < y->data) {

y->left = node;

} else {

y->right = node;

}

// если узел корень то просто возвращаем его

if (node->parent == nullptr){

node->color = 0;

return;

}

if (node->parent->parent == nullptr) {

return;

}

fixInsert(node);

}

NodePtr getRoot(){

return this->root;

}

int getComparing() {

int i = comparing;

comparing = 0;

return i;

}

int getTotalMemory(){

return totalMemory;

}

// удаление узла из дерева

void deleteNode(int data) {

deleteNodeHelper(this->root, data);

}

};

// main.cpp

#include "stdlib.h"

#include "fstream"

#include "rbTree.h"

#include "randomTree.h"

int main() {

// создание деревьев

RBTree rbTree;

ifstream test\_numbers, my\_keys;

test\_numbers.open("test\_numbers.txt");

if (!test\_numbers)

exit(-1);

// запись ключей в деревья

long int current\_key;

int totalMemRand = 0;

test\_numbers >> current\_key;

randomizeTree randomTree(current\_key);

rbTree.insert(current\_key);

while (!test\_numbers.eof()) {

test\_numbers >> current\_key;

totalMemRand += sizeof(element);

rbTree.insert(current\_key);

randomTree.insert(current\_key);

}

test\_numbers.close();

// вывод значений корней деревьев

cout << "rb tree root: " << rbTree.getRoot()->data << endl;

cout << "random tree root: " << randomTree.root->key << endl;

my\_keys.open("my\_100\_numbers.txt");

if (!my\_keys)

exit(-1);

// переменные нужны для создания статистики о работе поиска и деревьев

bool isRedBlackFind;

bool isRandomFind;

int comparingRedBlack;

int comparingRandom;

int sumComparingRedBlack;

int sumComparingRandom;

sumComparingRedBlack = 0;

sumComparingRandom = 0;

cout

<< " KEY | Red-Black tree | randomTree | Number of RB tree comparisons | Number of Random tree comparisons |"

<< endl;

while (!my\_keys.eof()) {

my\_keys >> current\_key;

// ищем только что считанное число из файла

isRedBlackFind = rbTree.searchTree(current\_key)->data == current\_key;

isRandomFind = randomTree.find(current\_key);

// получаем количество сравнений для одной операции поиска

comparingRedBlack = rbTree.getComparing();

comparingRandom = randomTree.getComparing();

cout << setw(9) << current\_key << " |";

cout << setw(24) << isRedBlackFind << " |";

cout << setw(17) << isRandomFind << " |";

cout << setw(32) << comparingRedBlack << " |";

cout << setw(37) << comparingRandom << " |";

cout << endl;

sumComparingRedBlack += comparingRedBlack;

sumComparingRandom += comparingRandom;

isRedBlackFind = false;

isRandomFind = false;

comparingRedBlack = 0;

comparingRandom = 0;

}

my\_keys.close();

// считаем среднее арифметическое сравнений на один ключ для каждой структуры

cout << "Arithmetic Mean of Red-Black Tree Comparisons: " << sumComparingRedBlack / 100 << endl;

cout << "Arithmetic Mean of Random Tree Comparisons: " << sumComparingRandom / 100 << endl;

cout<<endl;

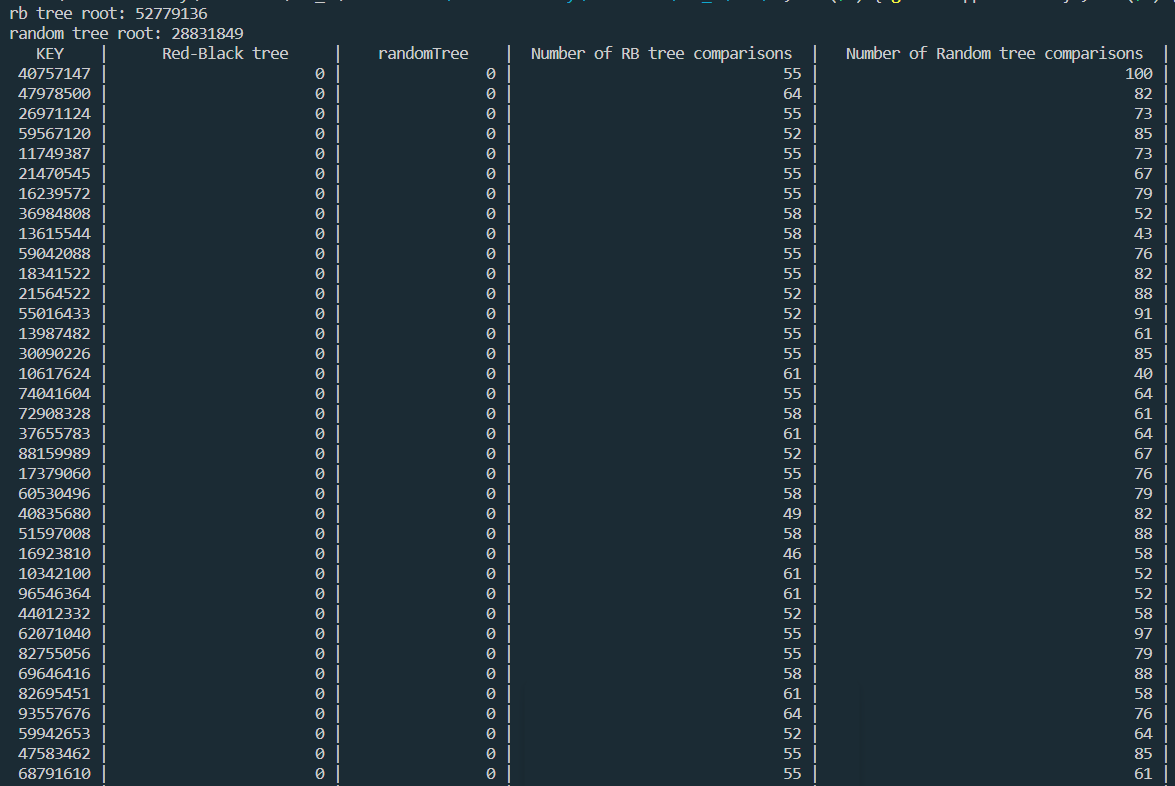
// получаем количество выделенной памяти для хранения каждой структуры

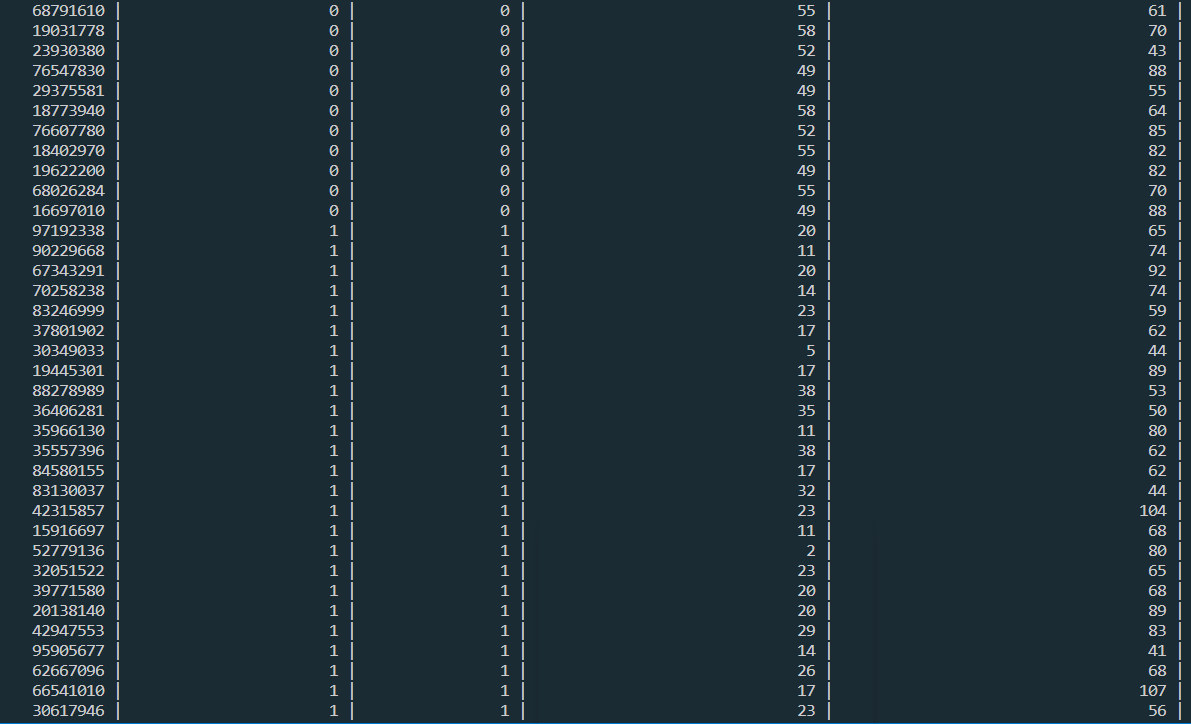
cout << "Dedicated memory for red-black tree: " << rbTree.getTotalMemory() << endl;

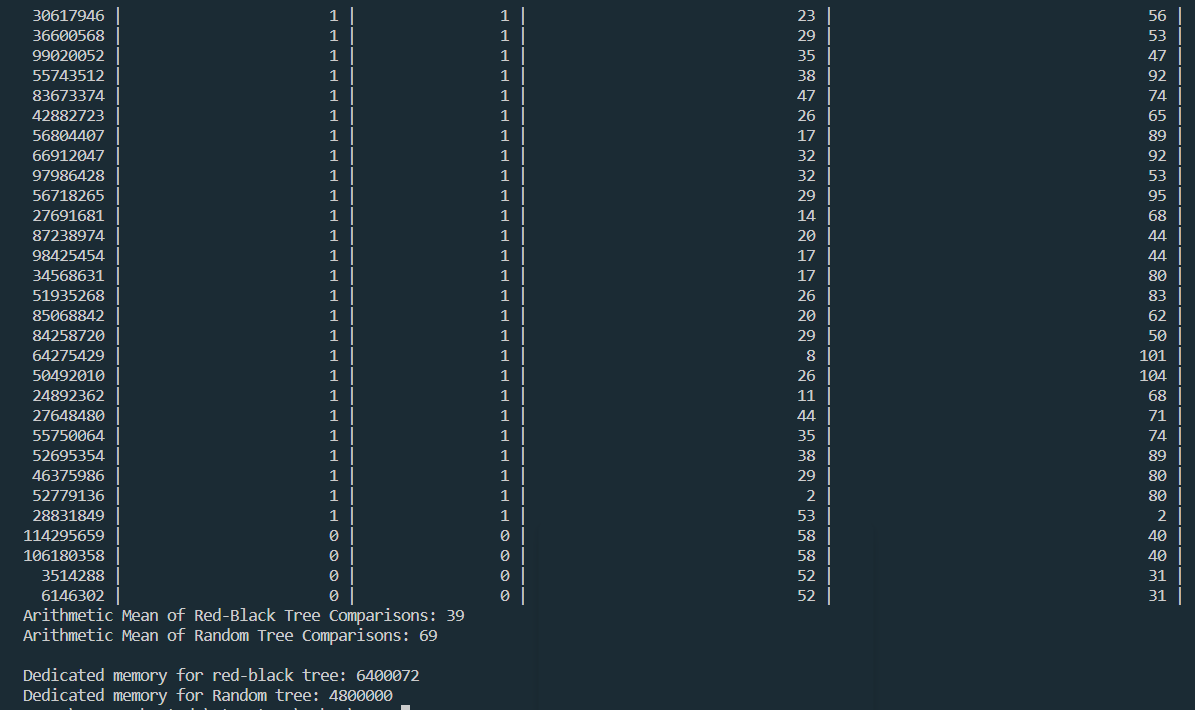
cout << "Dedicated memory for Random tree: " << totalMemRand << endl;

return 0;

}

Результаты работы программы:





Выводы:

По асимптотике средний вариант у обеих структур от логарифма, что не определяет однозначно какая структура лучше.

По результатам работы программы можно сделать вывод, что красно-чёрное дерево быстрее находит значение в 1.75 раза быстрее, этому предшествует несколько фактов:

во-первых, в пространственной сложности рандомизированное дерево легче красно-чёрного +– на 25%(узел randomTree занимает 24 байта, узел RBTree занимает 32 байта), что вполне закономерно – жертвуешь памятью – получаешь быстрый поиск, уменьшаешь количество необходимой памяти – получаешь менее быстрый поиск.

во-вторых, хотелось бы отметить, что оба дерева перестраиваются(не всегда) во время добавления нового узла, что требует дополнительных временных затрат. Этот фактор влияет на высоту дерева, и соответственно на количество сравнений при поиске.

Обе структуры отлично справляются со своей задачей, у каждой есть свои плюсы и минусы, так что явного фаворита и аутсайдера нет.