**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**КУРСОВАЯ РАБОТА**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: АВЛ-деревья – вставка и поиск. Исследование.**

| Студент гр. 1303 |  | Чубан Д.В. |
| --- | --- | --- |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2022

**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ**](#_tqt6uoab75jn) **3**

[**РЕАЛИЗОВАННЫЕ КЛАССЫ И СТРУКТУРЫ**](#_22x59sfmuq21) **4**

[**ИССЛЕДОВАНИЕ**](#_1ph5goma4oee) **5**

[**ЗАКЛЮЧЕНИЕ**](#_ihi3v2ru7nhd) **10**

[**ПРИЛОЖЕНИЕ А**](#_u1x8fc8bgz5o) **11**

# **ВВЕДЕНИЕ**

**Цель работы:**

Исследовать вставку и поиск элементов в АВЛ-дерево.

**Основные задачи:**

АВЛ-деревья - вставка и поиск. Исследование.

Программа должна реализовать следующий функционал по работе с

деревьями:

* Вставка элемента в дерево
* Поиск элемента в дереве

Сравнение времени на вставку и поиск в разных случаях АВЛ-

деревьев

# **РЕАЛИЗОВАННЫЕ КЛАССЫ И СТРУКТУРЫ**

Реализована структура Node узла АВЛ-дерева, которая имеет поле для данных, высоту поддерева, указатели на левого и правого детей.

Реализован класс АВЛ-дерева, имеющий поле с указателем на корень дерева и методы для реализации задач:

* void insert(int x)
* bool find(int x)

Также реализованы приватные методы подзадач для выполнения поиска, вставки и балансировки дерева.

# **ИССЛЕДОВАНИЕ**

Всего было проведено 4 исследования АВЛ-деревьев.

При первом тестировании последовательно создавались деревья с количеством элементов от 1 до 2001 с шагом 10 (рис. 1) и последовательно находили каждый элемент в дереве (рис. 2).

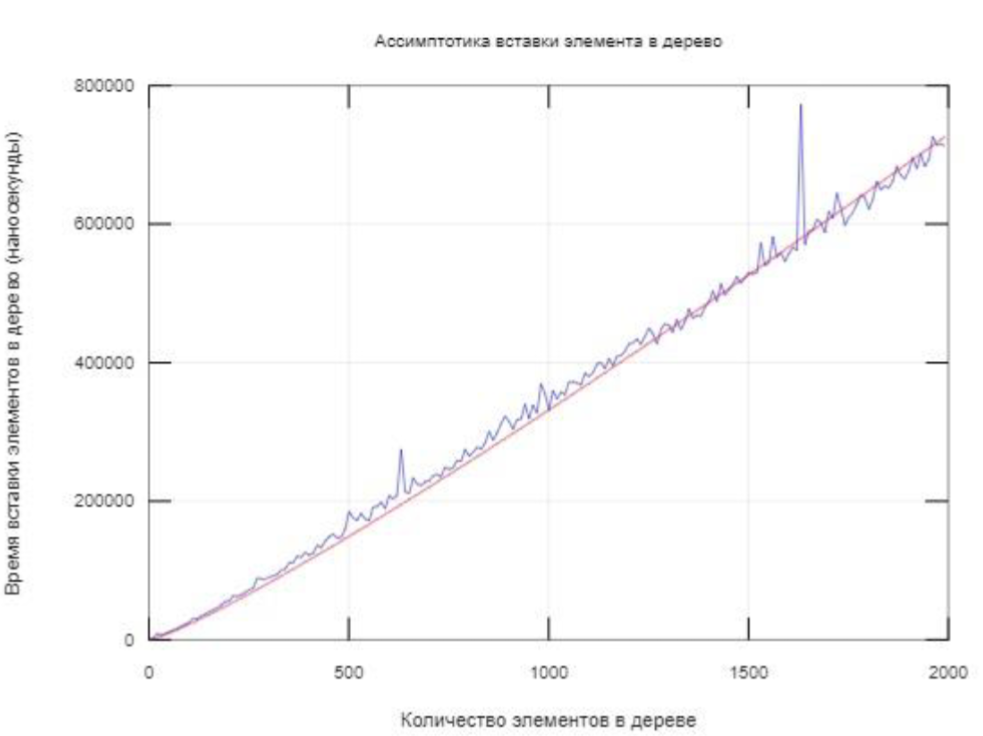


Рисунок 1 – Асимптотика вставки в АВЛ-дерево.

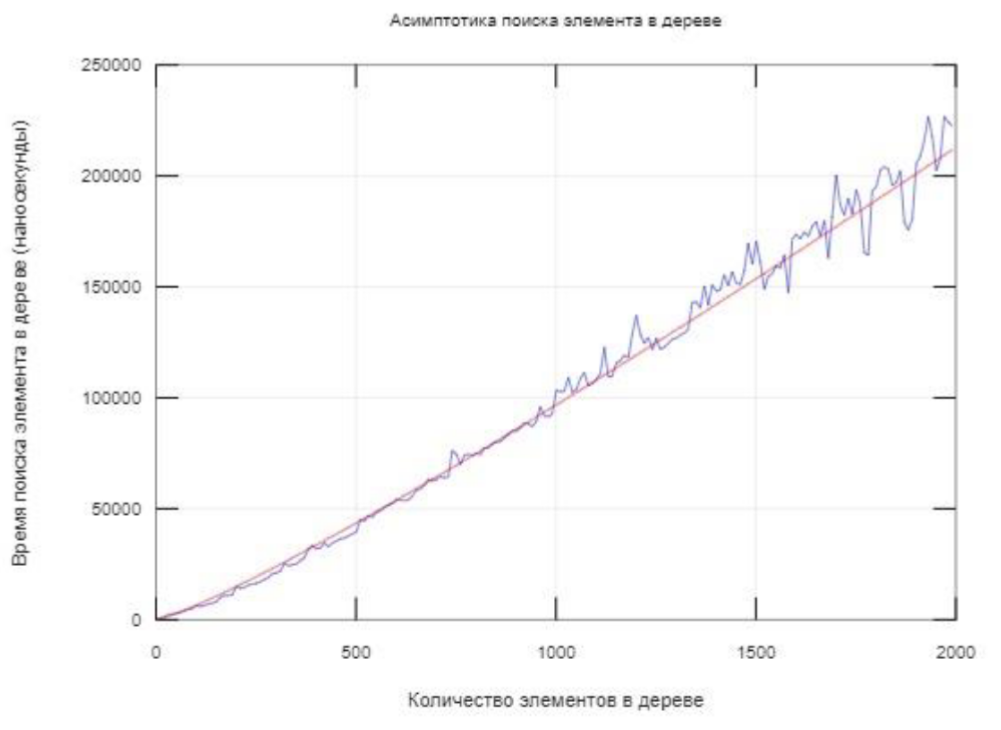


Рисунок 2 – Асимптотика поиска элемента в АВЛ-дереве.

Из графиков видно, что при добавлении и поиске элементов прослеживается теоретическая асимптотика O(log(n)). На графики также добавлена функция log(n) для демонстрации схожести теоретических и фактических результатов.

Далее были рассмотрены три возможных случая: средний, лучший и худший.

Для среднего случая случайно отсортированный массив чисел от 1 до 4096 заполняет дерево, затем в него добавляется (рис. 3) и сразу же ищутся (рис. 4) 500 новых элементов.

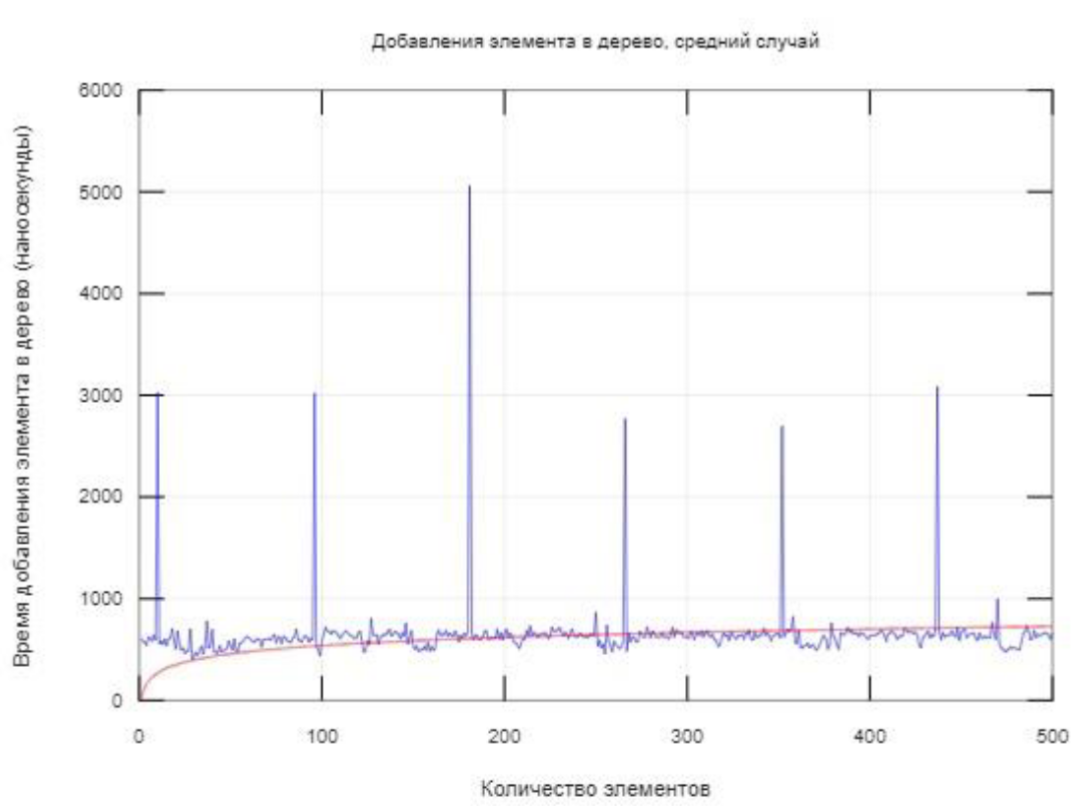


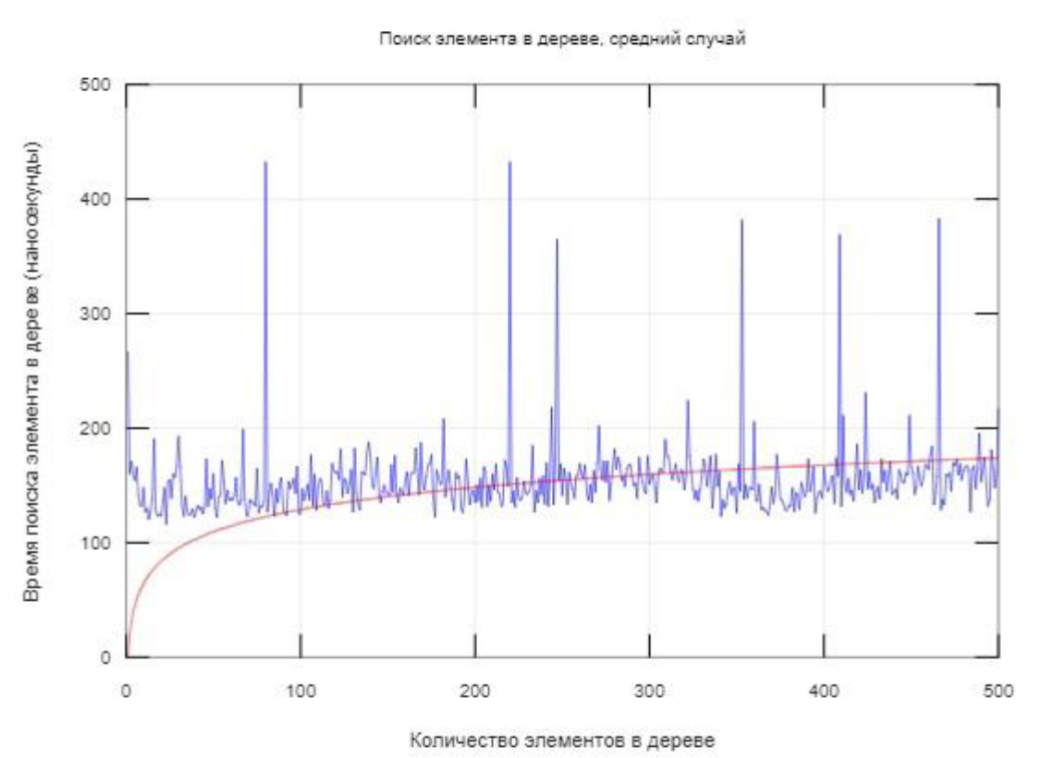
Рисунок 3 – добавление элементов, средний случай

Рисунок 4 – поиск элемента, средний случай.

Худшим случаем АВЛ-дерева является дерево Фибоначчи. Сначала дерево заполняется 4096 элементами, после чего происходит замер времени вставки (рис. 5) и поиска (рис. 6) 500 элементов.

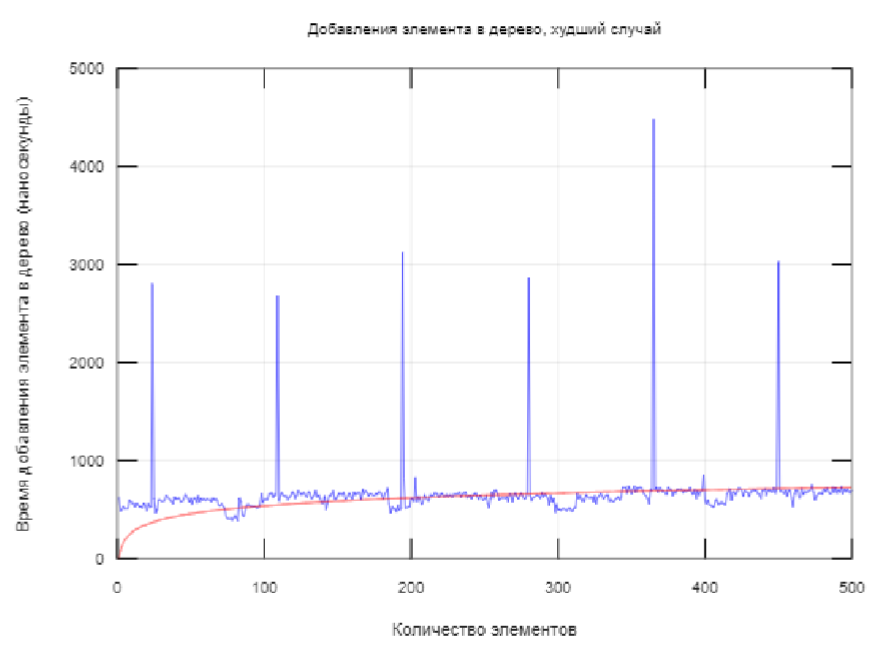


Рисунок 5 – добавление элементов, худший случай.

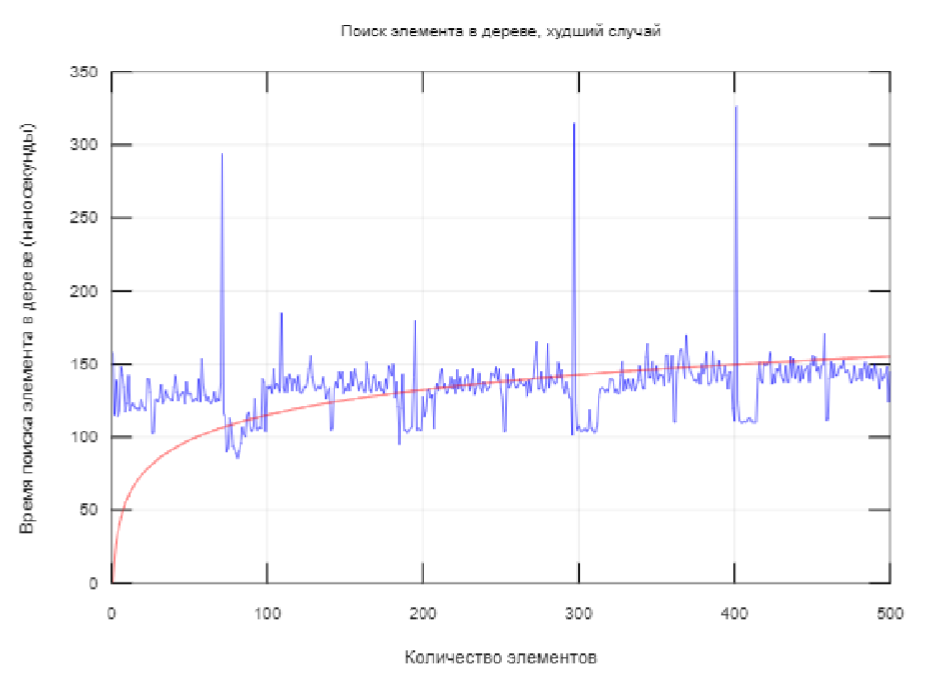


Рисунок 6 – поиск элементов, худший случай.

Лучший случай АВЛ-дерева – сбалансированное дерево. В дерево добавляется 4096 элементов, после чего производится вставка (рис. 7) и поиск (рис. 8) 500 новых элементов.

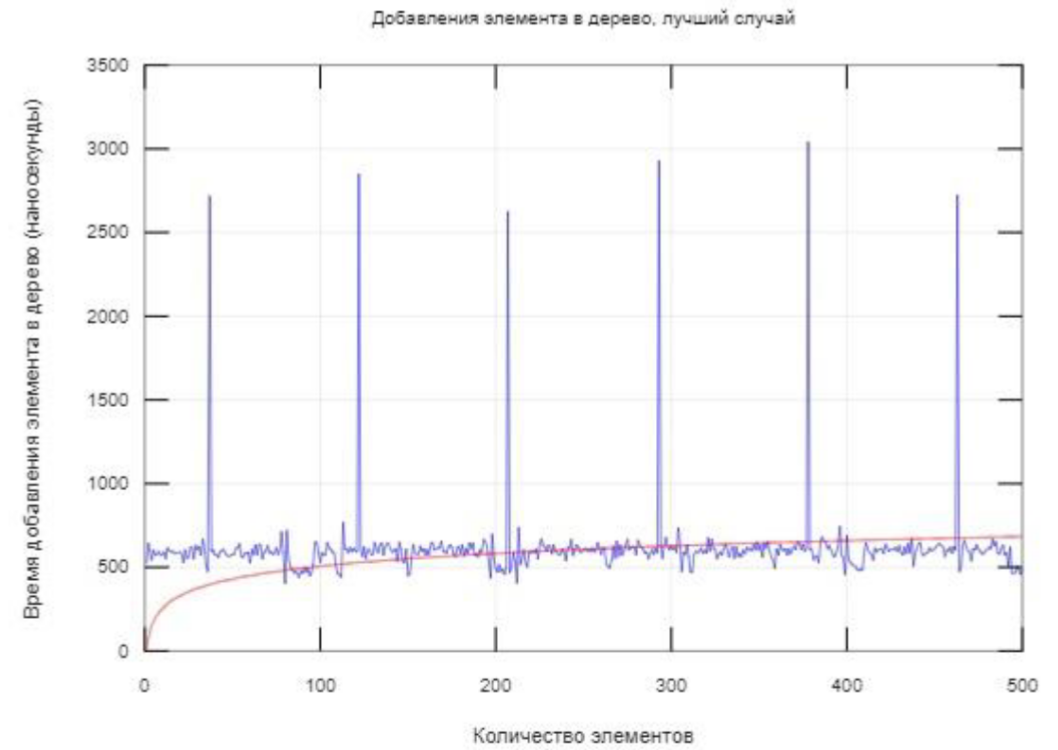


Рисунок 7 – добавление элементов, лучший случай.

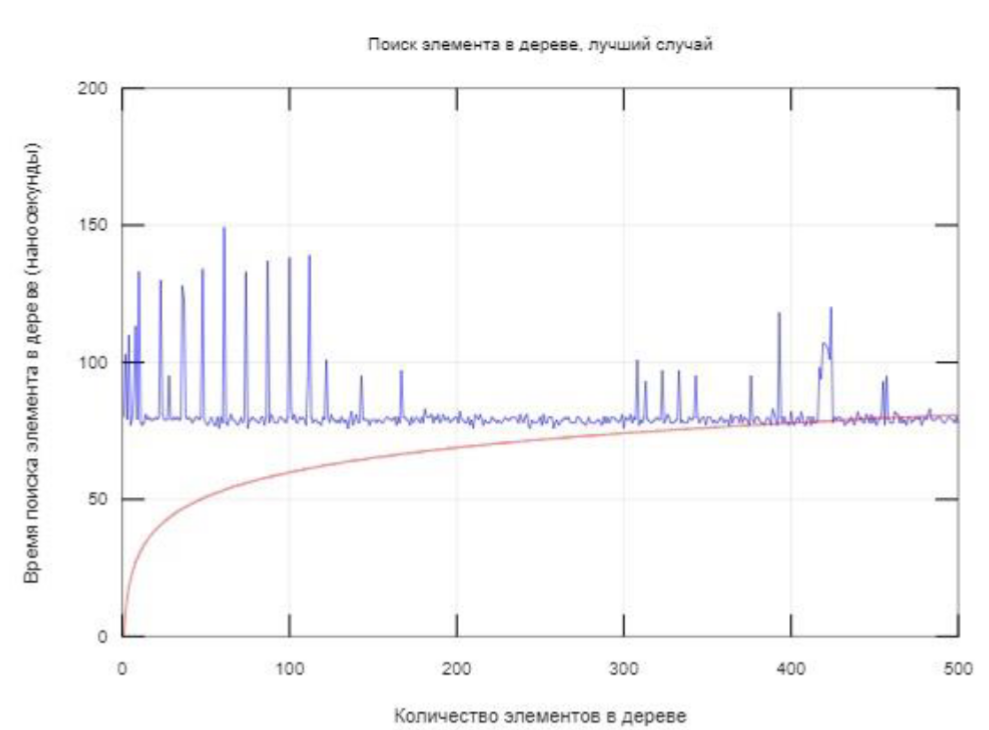


Рисунок 8 – поиск элементов, лучший случай.

# **ЗАКЛЮЧЕНИЕ**

Разработана программа для работы со вставкой и поиском элементов в АВЛ-деревьях.

Проведено исследование данных алгоритмов на различных видах деревьев, по итогам которого можно сделать вывод, что время вставки и поиска логарифмически зависит от наполненности дерева.

Наиболее быстро операции происходят в сбалансированном дереве, а наименее быстро в дереве Фибоначчи

# **ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла: cw.cpp

#include <string>

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <fstream>

#include <ctime>

#include <functional>

#include <algorithm>

#include <tuple>

#include <stdexcept>

#include <memory>

using namespace std;

using namespace chrono;

static std::string ch\_hor = "-", ch\_ver = "|", ch\_ddia = "/",

ch\_rddia = "\\", ch\_udia = "\\", ch\_ver\_hor = "|-", ch\_udia\_hor =

"\\-", ch\_ddia\_hor = "/-", ch\_ver\_spa = "| ";

class AVL

{

private:

struct node

{

int data;

node\* left;

node\* right;

int height;

};

node\* root;

void makeEmpty(node\* t)

{

if(t == nullptr)

return;

makeEmpty(t->left);

makeEmpty(t->right);

delete t;

}

node\* insert(int x, node\* t)

{

if(t == nullptr)

{

t = new node;

t->data = x;

t->height = 0;

t->left = t->right = nullptr;

}

else if(x < t->data)

{

t->left = insert(x, t->left);

if(height(t->left) - height(t->right) == 2)

{

if(x < t->left->data)

t = rightRotateSingle(t);

else

t = rightRotateDouble(t);

}

}

else if(x > t->data)

{

t->right = insert(x, t->right);

if(height(t->right) - height(t->left) == 2)

{

if(x > t->right->data)

t = leftRotateSingle(t);

else

t = leftRotateDouble(t);

}

}

t->height = max(height(t->left), height(t->right))+1;

return t;

}

node\* rightRotateSingle(node\* &t)

{

node\* u = t->left;

t->left = u->right;

u->right = t;

t->height = max(height(t->left), height(t->right))+1;

u->height = max(height(u->left), t->height)+1;

return u;

}

node\* leftRotateSingle(node\* &t)

{

node\* u = t->right;

t->right = u->left;

u->left = t;

t->height = max(height(t->left), height(t->right))+1;

u->height = max(height(t->right), t->height)+1 ;

return u;

}

node\* leftRotateDouble(node\* &t)

{

t->right = rightRotateSingle(t->right);

return leftRotateSingle(t);

}

node\* rightRotateDouble(node\* &t)

{

t->left = leftRotateSingle(t->left);

return rightRotateSingle(t);

}

// Finding an element

bool find(int x, node\* t) {

if (t==nullptr) {

return false;

}

else if (x < t->data) {

return find(x, t->left);

} else if (x > t->data) {

return find(x, t->right);

} else {

return true;

}

}

node\* findMin(node\* t)

{

if(t == nullptr)

return nullptr;

else if(t->left == nullptr)

return t;

else

return findMin(t->left);

}

node\* findMax(node\* t)

{

if(t == nullptr)

return nullptr;

else if(t->right == nullptr)

return t;

else

return findMax(t->right);

}

int height(node\* t)

{

return (t == nullptr ? -1 : t->height);

}

int getBalance(node\* t)

{

if(t == nullptr)

return 0;

else

return height(t->left) - height(t->right);

}

void treeSimplePrint(node\* t)

{

if(t == nullptr)

return;

treeSimplePrint(t->left);

cout << t->data << " ";

treeSimplePrint(t->right);

}

void treePrint(node\* node, std::string const & prefix = "", bool root = true, bool last = true) {

std::cout << prefix << (root ? "" : (last ? ch\_udia\_hor :ch\_ver\_hor)) << (node ? std::to\_string(node->data) : "") <<std::endl;

if (!node || (!node->left && !node->right))

return;

treePrint(node->left, prefix + (root ? "" : (last ? " " :ch\_ver\_spa)), false, false);

treePrint(node->right, prefix + (root ? "" : (last ? " ": ch\_ver\_spa)), false, true);

}

void treeComplexPrint(node\* node, bool high = true, std::vector<std::string> const & lpref = std::vector<std::string>(), std::vector<std::string> const & cpref = std::vector<std::string>(), std::vector<std::string> const & rpref = std::vector<std::string>(), bool root = true, bool left = true, std::shared\_ptr<std::vector<std::vector<std::string>>> lines = nullptr) {

if (!node) return;

typedef std::vector<std::string> VS;

auto VSCat = [](VS const & a, VS const & b){ auto r = a;

r.insert(r.end(), b.begin(), b.end()); return r; };

if (root) lines = std::make\_shared<std::vector<VS>>();

if (node->left)

treeComplexPrint(node->left, high, VSCat(lpref, high ? VS({" ", " "}) : VS({" "})), VSCat(lpref, high ? VS({ch\_ddia, ch\_ver}) : VS({ch\_ddia})), VSCat(lpref, high ? VS({ch\_hor, " "}) : VS({ch\_hor})), false, true, lines);

auto sval = std::to\_string(node->data);

size\_t sm = left || sval.empty() ? sval.size() / 2 : ((sval.size() + 1) / 2 - 1);

for (size\_t i = 0; i < sval.size(); ++i)

lines->push\_back(VSCat(i < sm ? lpref : i == sm ?

cpref : rpref, {std::string(1, sval[i])}));

if (node->right)

treeComplexPrint(node->right, high, VSCat(rpref, high ? VS({ch\_hor, " "}) : VS({ch\_hor})), VSCat(rpref, high ? VS({ch\_rddia, ch\_ver}) : VS({ch\_rddia})), VSCat(rpref, high ? VS({" ", " "}) : VS({" "})), false, false, lines);

if (root) {

VS out;

for (size\_t l = 0;;++l) {

bool last = true;

std::string line;

for (size\_t i = 0; i < lines->size(); ++i) {

if (l < (\*lines).at(i).size()) {

line += lines->at(i)[l];

last = false;

} else line += " ";

}

if (last) break;

out.push\_back(line);

}

for (size\_t i = 0; i < out.size(); ++i)

std::cout << out[i] << std::endl;

}

}

node\* fibonTree1 (int k, node\* t)

{

if (k <= 0) {

t = nullptr;

return t;

}

if ( k == 1 ) {

t = new node;

t->left = t->right = nullptr;

t->height = k-1;

return t;

} else {

node\* t = new node;

t->left = t->right = nullptr;

t->height = k-1;

t->left = fibonTree1(k-1,t->left);

t->right = fibonTree1(k-2,t->right);

return t;

}

}

int fibonTree2 (node\* t, int k)

{

if (t != nullptr) {

k = fibonTree2(t->left, k);

t->data = k;

k++;

k = fibonTree2(t->right, k);

}

return k;

}

public:

AVL()

{

root = nullptr;

}

void insert(int x)

{

root = insert(x, root);

}

bool find(int x)

{

return find(x, root);

}

void insertAndVerify(int x)

{

root = insert(x, root);

if (!find(x, root)) {

cout << "ERROR: Cannot find element " << x << " inside avl tree" << endl;

}

}

int getHeight()

{

return height(root);

}

int getMax()

{

node\* t = findMax(root);

return t->data;

}

int getMin()

{

node\* t = findMin(root);

return t->data;

}

void buildFibonTree(int k, int startElem)

{

root = fibonTree1(k, root);

fibonTree2(root, startElem);

}

void clear()

{

return makeEmpty(root);

}

void display()

{

treeComplexPrint(root);

cout << endl;

}

};

int main(){

srand((unsigned)time(0));

std::fstream out\_add\_n\_avg;

std::fstream out\_find\_n\_avg;

std::fstream out\_avg\_add;

std::fstream out\_avg\_find;

std::fstream out\_best\_add;

std::fstream out\_best\_find;

std::fstream out\_worst\_add;

std::fstream out\_worst\_find;

out\_add\_n\_avg.open("add\_n\_avg.txt", ios::out);

out\_find\_n\_avg.open("find\_n\_avg.txt", ios::out);

out\_avg\_add.open("add\_avg.txt", ios::out);

out\_avg\_find.open("avg\_find.txt", ios::out);

out\_best\_add.open("best\_add.txt", ios::out);

out\_best\_find.open("best\_find.txt", ios::out);

out\_worst\_add.open("worst\_add.txt", ios::out);

out\_worst\_find.open("worst\_find.txt", ios::out);

//Асимптотика

vector<int> amounts\_n;

int maxSize = 2001;

for (int size = 1; size < maxSize; size=size + 10) {

int a [size];

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

a[i] = i + 1;

}

AVL tree;

auto start = high\_resolution\_clock::now().time\_since\_epoch();

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

tree.insert(a[i]);

}

auto stop = high\_resolution\_clock::now().time\_since\_epoch();

auto duration = stop - start;

out\_add\_n\_avg << duration.count() << ", ";

out\_add\_n\_avg.flush();

start = high\_resolution\_clock::now().time\_since\_epoch();

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

tree.find(a[i]);

}

stop = high\_resolution\_clock::now().time\_since\_epoch();

duration = stop - start;

out\_find\_n\_avg << duration.count() << ", ";

out\_find\_n\_avg.flush();

tree.clear();

}

//исследование среднего случая

// Create tree with 4096 elements, started from 1 to 4096 in random order

maxSize = 4096;

int a [maxSize];

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

a[i] = i + 1;

}

// Change order

random\_shuffle(&a[0],&a[maxSize]);

AVL tree;

for (int i=0; i < 500; i++) {

auto start = high\_resolution\_clock::now().time\_since\_epoch();

tree.insert(a[i]);

auto stop = high\_resolution\_clock::now().time\_since\_epoch();

auto duration = stop - start;

out\_avg\_add << duration.count() << ", ";

out\_avg\_add.flush();

start = high\_resolution\_clock::now().time\_since\_epoch();

tree.find(a[i]);

stop = high\_resolution\_clock::now().time\_since\_epoch();

duration = stop - start;

out\_avg\_find << duration.count() << ", ";

out\_avg\_find.flush();

}

//рассмотрим худший случай (дерево Фибоначчи)

// Create fibon tree with heigth 17 (4180 elements), started from 1 to 4180

AVL fibon;

fibon.buildFibonTree(17, 1);

maxSize = fibon.getMax();

int add\_num = 500;

// Measure insert and find performance on worst case

for (int i=1; i <= add\_num; i++) {

fibon.insert(maxSize + i);

}

// Measure insert and find performance on worst case

for (int i=1; i <= add\_num; i++) {

auto start = high\_resolution\_clock::now().time\_since\_epoch();

fibon.insert(maxSize + i);

auto stop = high\_resolution\_clock::now().time\_since\_epoch();

auto duration = stop - start;

out\_worst\_add << duration.count() << ", ";

out\_worst\_add.flush();

start = high\_resolution\_clock::now().time\_since\_epoch();

fibon.find(maxSize + i);

stop = high\_resolution\_clock::now().time\_since\_epoch();

duration = stop - start;

out\_worst\_find << duration.count() << ", ";

out\_worst\_find.flush();

}

//рассмотрим лучший случай

// Create balance tree with heigth 11 (4096 elements), started from 1 to 4096

AVL balanceTree;

int treeSize = 4096;

int step = treeSize / 4;

int arr [treeSize];

arr[0] = treeSize / 2; //root key

balanceTree.insert(arr[0]);

int k = 0;

int iteration = 1;

int j = 1;

while (step >= 1)

{

for (int i=k; i < k + iteration; i++)

{

arr[j] = arr[i] - step;

arr[j+1] = arr[i] + step;

balanceTree.insert(arr[j]);

balanceTree.insert(arr[j+1]);

j = j + 2;

}

step = step / 2;

k = k + iteration;

iteration = iteration \* 2;

}

maxSize = treeSize;

// Measure insert and find performance on best case

for (int i=1; i <= add\_num; i++) {

auto start = high\_resolution\_clock::now().time\_since\_epoch();

balanceTree.insert(maxSize + i);

auto stop = high\_resolution\_clock::now().time\_since\_epoch();

auto duration = stop - start;

out\_best\_add << duration.count() << ", ";

out\_best\_add.flush();

start = high\_resolution\_clock::now().time\_since\_epoch();

balanceTree.find(maxSize + i);

stop = high\_resolution\_clock::now().time\_since\_epoch();

duration = stop - start;

out\_best\_find << duration.count() << ", ";

out\_best\_find.flush();

}

out\_add\_n\_avg.close();

out\_find\_n\_avg.close();

out\_avg\_add.close();

out\_avg\_find.close();

out\_best\_add.close();

out\_best\_find.close();

out\_worst\_add.close();

out\_worst\_find.close();

tree.clear();

fibon.clear();

balanceTree.clear();

return 0;

}