МИНОБРНАУКИ РОССИИ САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

**ОТЧЕТ**

по лабораторной работе № 2

дисциплина «Алгоритмы и Структуры данных»

Тема: «Самобалансирующие двоичные деревья поиска»

Студент гр. 3351 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Фабер К.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Пестерев Д.О.

Санкт-Петербург

2024

**Цель лабораторной работы: реализация самобалансирующихся деревьев поиска и экспериментальная проверка оценок высоты данных деревьев.**

**Теоретическая часть.**

**АВЛ дерево.**

АВЛ дерево – двоичное дерево поиска, у которого для любого узла X: , где ( – высота левого поддерева X, – высота правого поддерева X).

Алгоритм вставки в АВЛ дерево: сравниваем значение ключа с ключом текущего узла, начиная с корня, в зависимости от результата сравнения идем либо в правый узел, либо в левый (если значение ключа нового элемента больше ключа текущего узла – вправо, иначе – влево), после полного прохождения дерева, когда текущей узел указывает на нулевой указатель, вставляем новый элемент и привязываем его к родителю левым или правым ребенком, в зависимости от значения ключа, далее, для того чтобы сохранялись свойства АВЛ дерева, необходимо произвести балансировку: берем самый нижний затронутый узел, вызываем от него один из поворотов при необходимости и рекурсивно повторяем от родителя.

Алгоритм удаления в АВЛ дереве: путём сравнения ключей, приходим к узлу дерева, который нужно удалить, после нахождения данного узла может быть 3 случая:

1. Найденный элемент не имеет детей (лист)
2. Найденный элемент имеет одного ребенка
3. Найденный элемент имеет двух детей

В первом случае необходимо просто удалить узел, во втором – перед удалением нужно произвести замену указателей: ребенок найденного узла становится ребенком родителя найденного узла, в 3 случае – ищем наименьший элемент больше данного (т.е. идем один раз вправо и влево до конца), в результате этот элемент имеет не больше одного поддерева, меняем местами данный элемент с удаляемым и вызываем функцию удаления для правого поддерева текущего элемента и получаем первый, либо второй случай. После удаления также необходимо произвести балансировку.

Верхняя оценка высоты АВЛ дерева:

Пусть - минимальное количество узлов в АВЛ дереве высоты h, тогда из определения АВЛ дерева можно получить, что ;

N(0) = 0;

N(1) = 1;

N(2) = 2;

Предположим, что где (h + 2) число Фибоначчи.

Докажем индукцией:  
База индукции: h = 0:

Предположение индукции:

Распишем для k + 1:

Ч.т.д.

Найдем верхнюю оценку высоты АВЛ дерева:

*,* где ;

Тогда и ;

Если n – количество элементов в дереве, то

*,* отсюда следует, что

**Красно-черное дерево.**

Красно-черное дерево – это двоичное дерево поиска, где каждый узел имеет свой цвет: красный или черный, определяемый свойствами.

Свойства красно-черного дерева:

1. Корень всегда черный
2. Потомки красного узла не могут быть красными
3. Существуют фиктивные узлы – черные листья
4. Путь из корня до любого листа содержит одинаковое количество черных вершин

Алгоритм вставки в красно-черное дерево: аналогично АВЛ дереву сначала спускаемся вниз, сравнивая ключи, после вставки нового элемента (вставляем всегда красный), к нему подвешиваются фиктивные черные листы. Далее требуется восстановить все свойства красно-черного дерева, если они нарушены. После вставки могут нарушиться следующие свойства: первое (корень всегда черный) – когда вставляем первый элемент и второе (потомки красного не могут быть красными) - когда родитель вставленного узла – красный. Для того, чтобы восстановить свойства красно-черного дерева необходимо произвести балансировку. Первое свойство исправляется путем перекрашивания вставленного узла (корня) в черный, если нарушено четвертое свойство, то вариант балансировки зависит от цвета дяди вставленного узла:

1. Дядя – красный.

Если дядя красный, производим обмен цветов: дед становится красным, а его дети – черными. Далее необходимо выше рекурсивно проделать те же действия от деда.

1. Дядя – черный.

Если дядя черный и вставленный элемент является левым сыном, необходимо произвести малый правый поворот и перекрасить узлы: родитель вставленного узла становится черным, старый дед становится красным. Если же вставленный элемент – правый сын, то большой правый поворот и перекрасить аналогично.

Алгоритм удаления из красно-черного дерева: аналогично АВЛ дереву, при удалении узла, будет 3 случая:

1. Найденный элемент не имеет детей (только фиктивные листы)
2. Найденный элемент имеет одного ребенка и фиктивный лист
3. Найденный элемент имеет двух детей

Если у элемента есть 2 ребенка, то необходимо найти минимальный элемент в правом поддереве удаляемого узла и заменить удаляемый узел на этот минимальный элемент, а затем его нужно удалить из правого поддерева. Таким образом мы пришли к первому или второму случаю, т.к. у максимального элемента нет правого сына. При удалении элемента с одним ребенком или без него, существует 3 случая:

1. Когда удаляемый узел – красный
2. Когда удаляемый узел – черный без детей
3. Когда удаляемый узел – черный с одним красным ребенком

В первом случае необходимо заменить узел на фиктивный лист, в третьем случае нужно перекрасить сына в черный и подвесить вместо отца, второй случай разбивается на несколько и сводится к общей задаче, когда было корректное красно-черное дерево и поддерево с черной вершиной x с черной высотой h, вместо него подвесили такое же дерево, но черная глубина на 1 меньше. Будем считать, что это дерево – левый сын своего родителя (если родителя нет, то все свойства сохраняются), если это дерево – правый сын, то действия будут симметричны. Пусть отец данного поддерева – a, брат вершины данного поддерева – b, тогда возможно несколько случаев:

1. Брат – черный:

В таком случае важны цвета детей брата:

* 1. Оба ребенка брата – черные:

В случае, когда поддерево x имеет 1 элемент, этими потомками могут быть только фиктивные листы, иначе имеем некорректное красно-черное дерево, в общем случае, дети брата могут быть полноценными поддеревьями. Здесь, нужно перекрасить брата в красный, а родителя в черный, если родитель был красным, то дерево отбалансировано, если же черным, то необходимо рекурсивно отбалансировать дерево от родителя.

* 1. Хотя бы один ребенок брата – красный:
     1. Правый ребенок – красный (левый – любой):

В этом случае перекрашиваем брата в родительский цвет (цвет родителя не важен, т.к. он не повлияет на исходную черную высоту), самого родителя в – черный, а красного ребенка – в черный и выполняем левый поворот.

* + 1. Левый ребенок – красный (правый – черный):

Необходимо совершить обмен цветов красного ребенка и брата, тем самым свести ситуацию к случаю 1.2.1

1. Брат – красный:

Перекрашиваем родителя в красный, а брата в черный и применяем левый поворот относительно родителя и приходим к случаю, когда брат поддерева x – черный.

Верхняя оценка высоты красно-черного дерева:

Пусть x – произвольная вершина, bh(x) – “черная” высота поддерева x (количество черных элементов при любом пути от x до листа, не учитывая x), тогда докажем, что количество элементов (ключей) . Докажем по индукции по h(x) – высоте поддерева x:

База индукции: h(x) = 0, в этом случае x – фиктивный лист и не имеет ключа, а bh(x) = 0, подставляя в неравенство получим .

Далее возьмем вершину x на произвольной глубине и рассмотрим черную высоту ее потомков:

Высота левого потомка

Высота правого потомка

Тогда, по предположению индукции количество ключей в левом и правом поддереве .

Теперь найдем количество ключей в поддереве x: ч.т.д.

Найдем верхнюю оценку высоты красно-черного дерева:

Черная высота корня т.к. у красного узла нет красных потомков, и в случае чередования цветов черная высота равна половине всей высоты.

Применим неравенство для корня, тогда получим:

;

;

Тогда

**Практическая часть.**

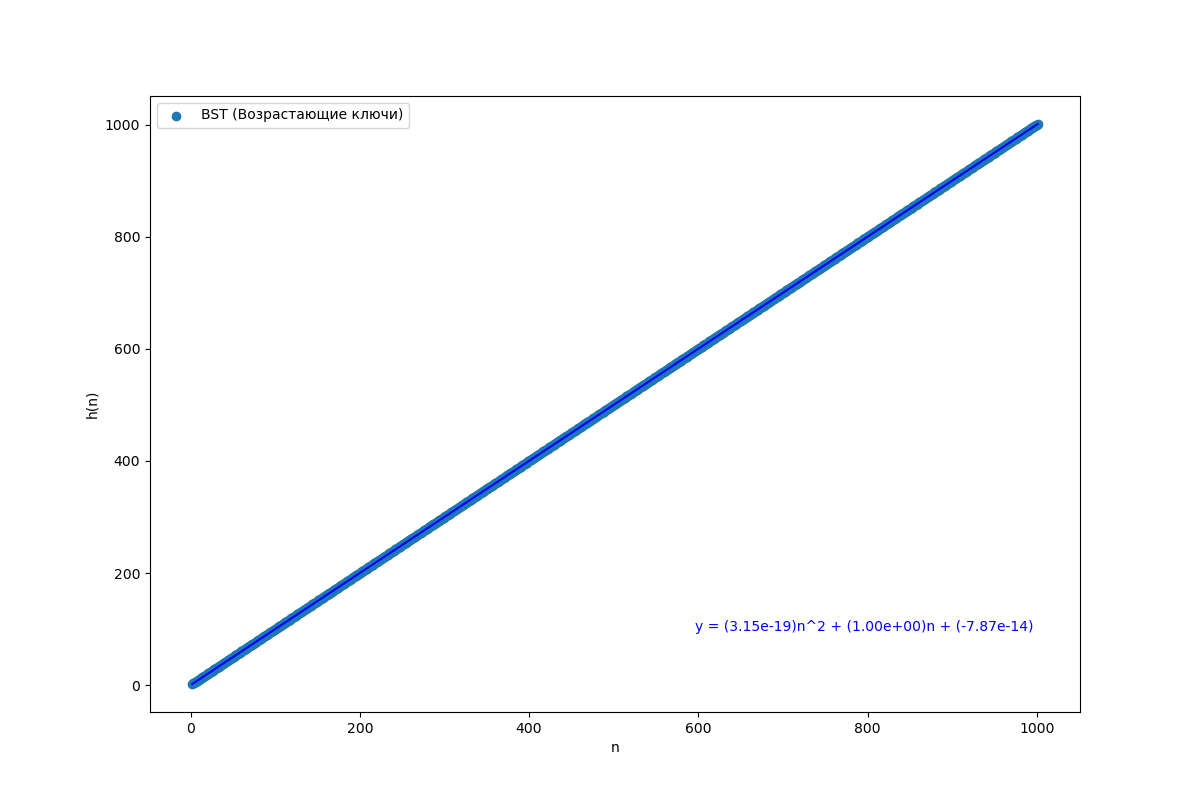
****

Рисунок 1 – График зависимости высоты h от количества элементов n для бинарного дерева поиска (при возрастающих ключах)

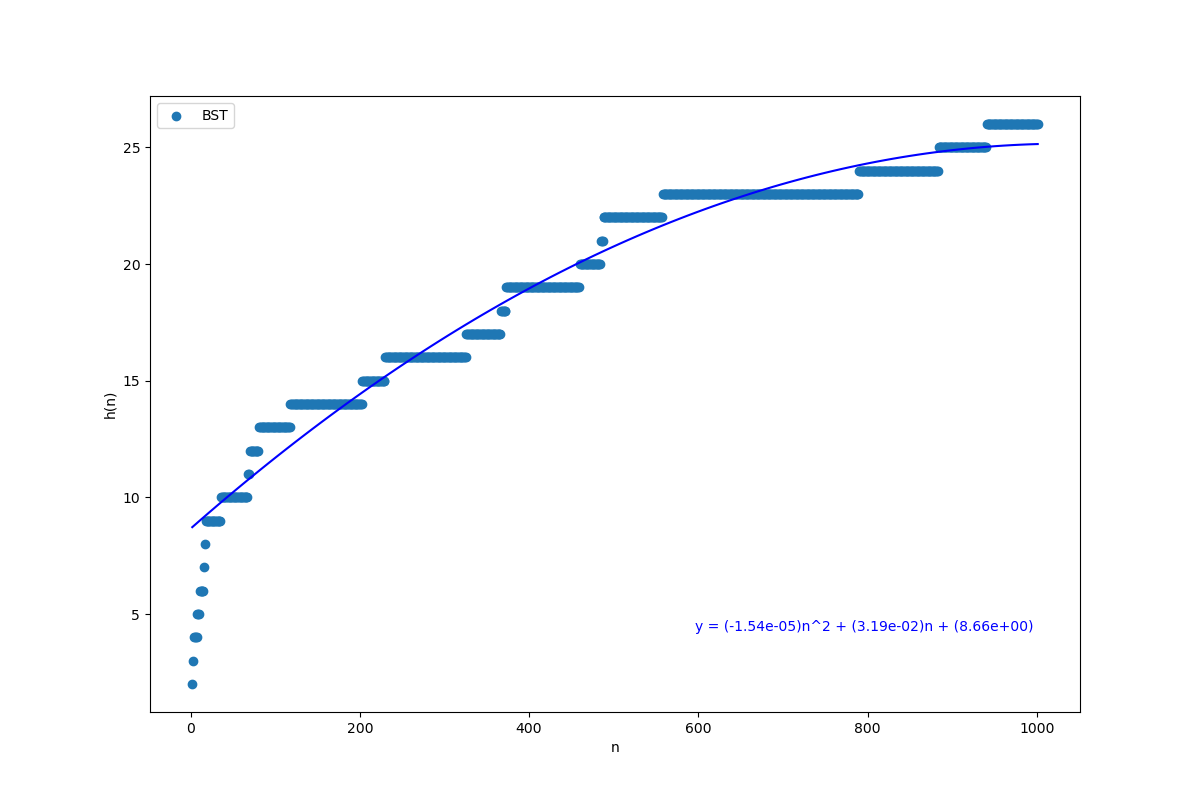
****

Рисунок 2 – График зависимости высоты h от количества элементов n для бинарного дерева поиска (при случайных ключах)

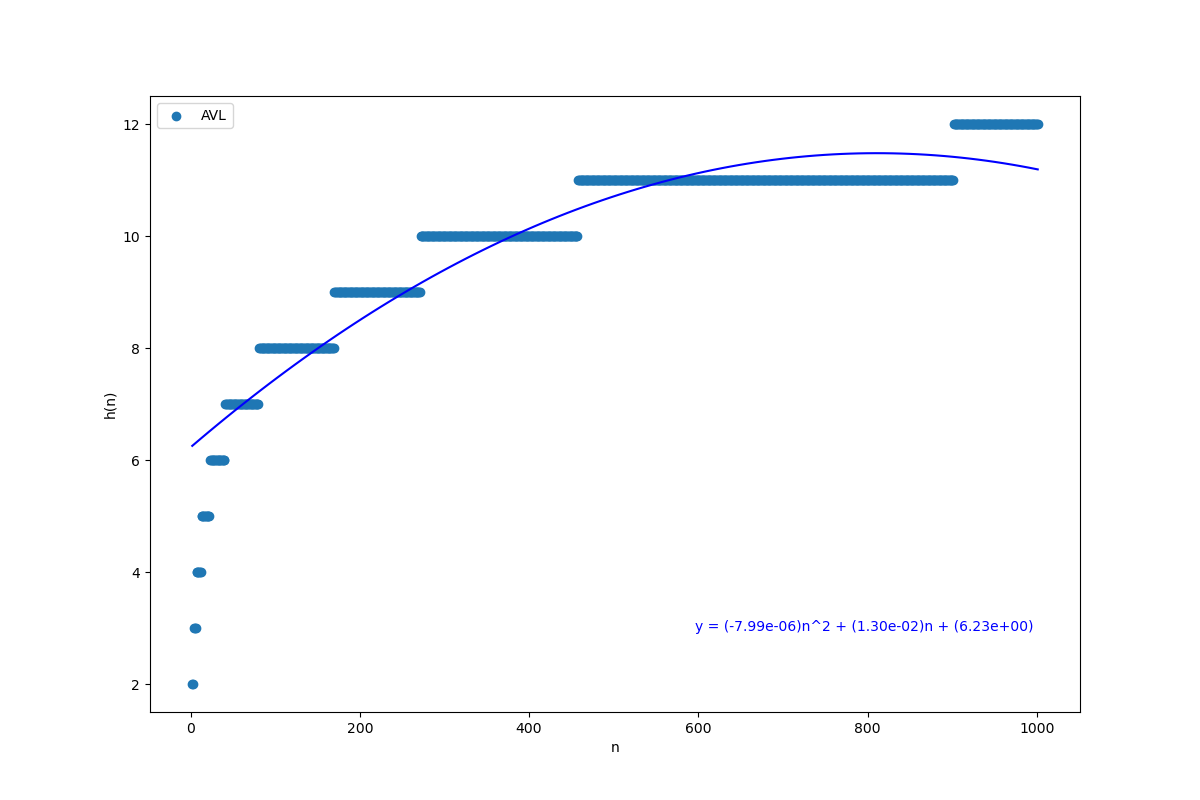
****

Рисунок 3 – График зависимости высоты h от количества элементов n для АВЛ дерева

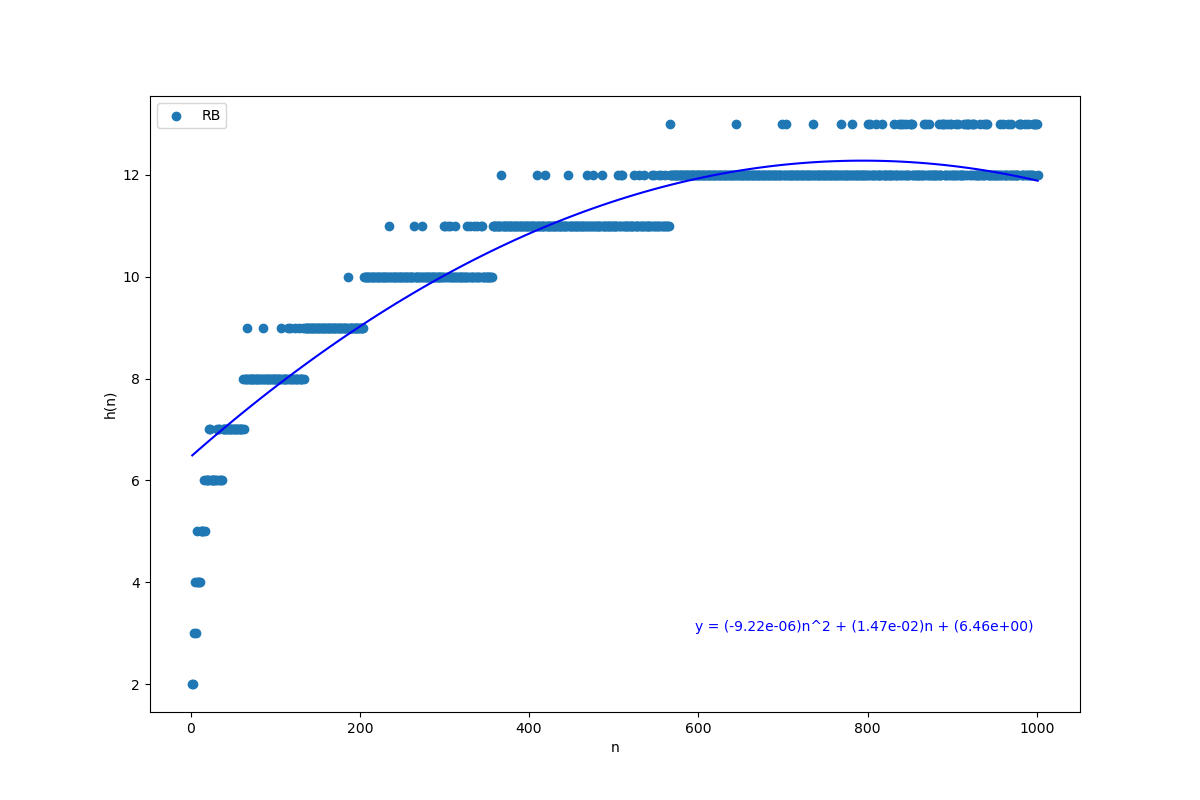


Рисунок 4 – График зависимости высоты h от количества элементов n для красно-черного дерева

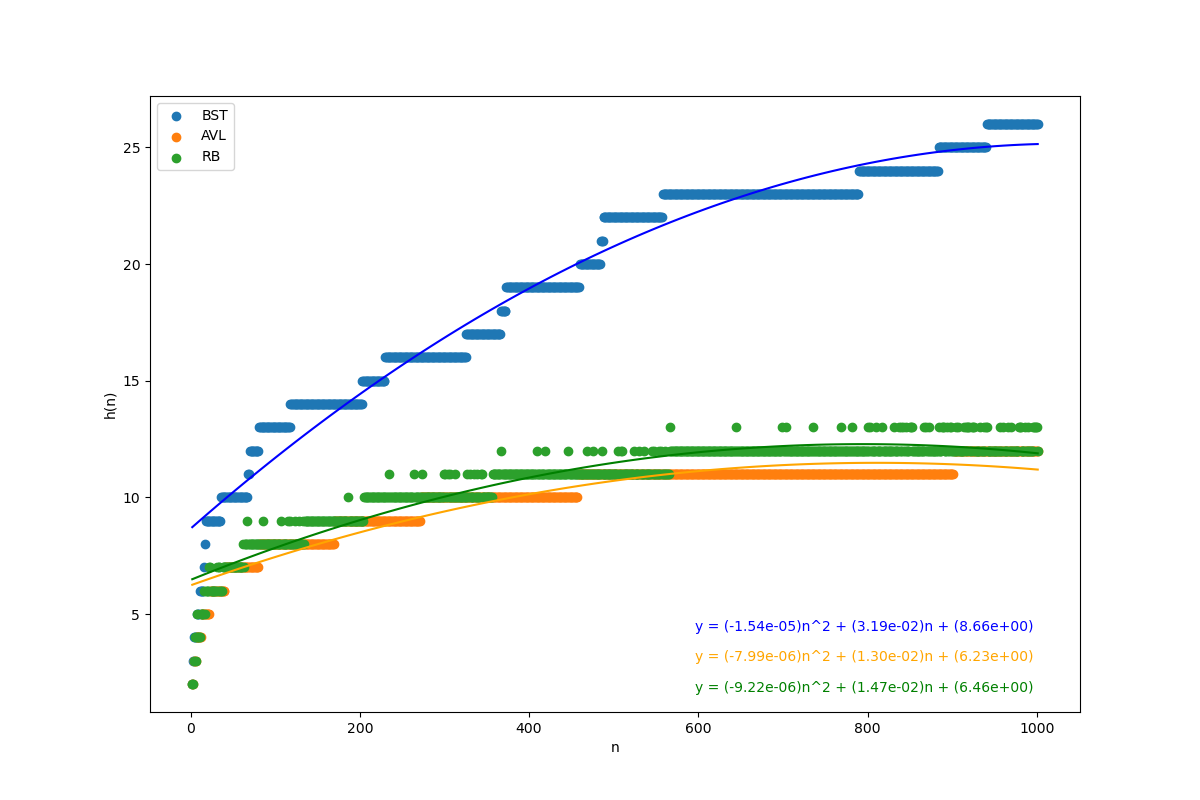
****

Рисунок 5 – График зависимости высоты h от количества элементов n для всех деревьев

Сравнивая теоретические данные, с практическими, можно заметить, что асимптотика высоты действительно логарифмическая. Высота бинарного дерева поиска растет быстрее, это объясняется тем, что бинарное дерево поиска никак не балансируется при вставке нового ключа, в отличии от АВЛ дерева и красно-черного дерева. Также на графике высота красно-черного дерева выше, чем высота АВЛ дерева, что также согласуется с теорией.

**Обходы дерева в глубину и ширину.**

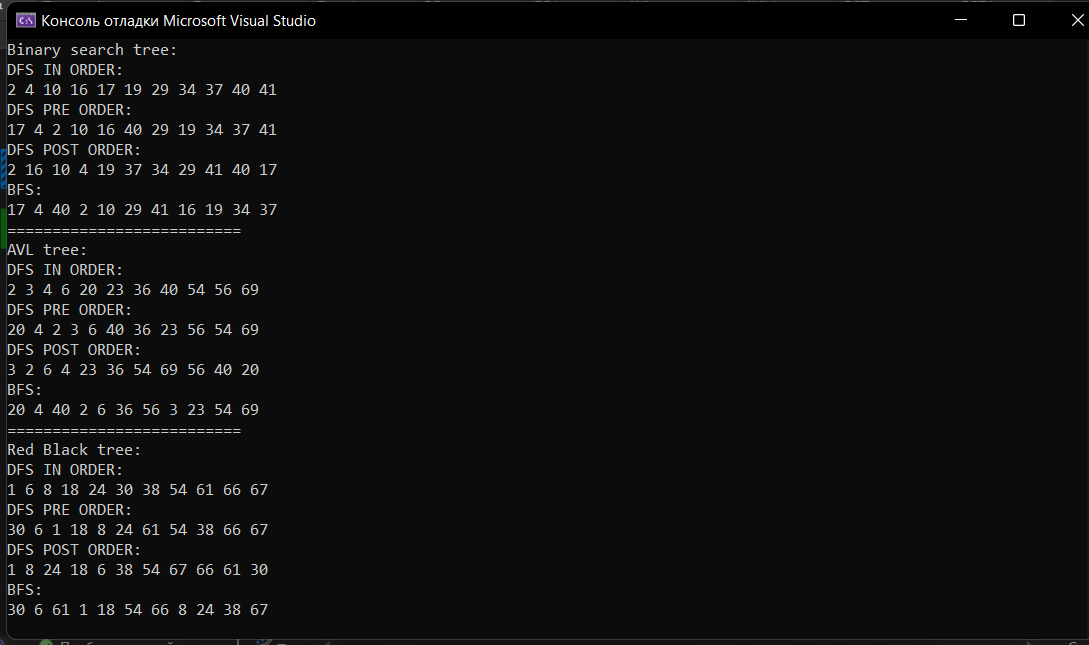


Рисунок 5 – Пример работы обходов

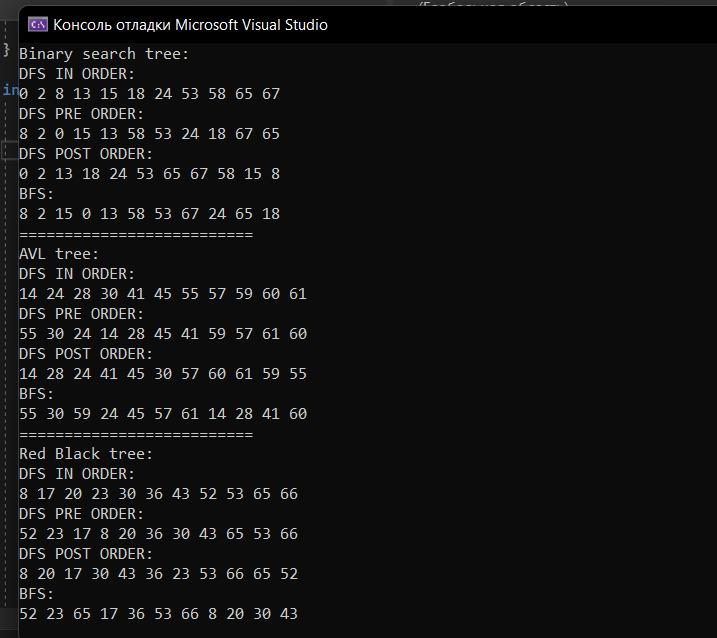


Рисунок 6 – Пример работы обходов

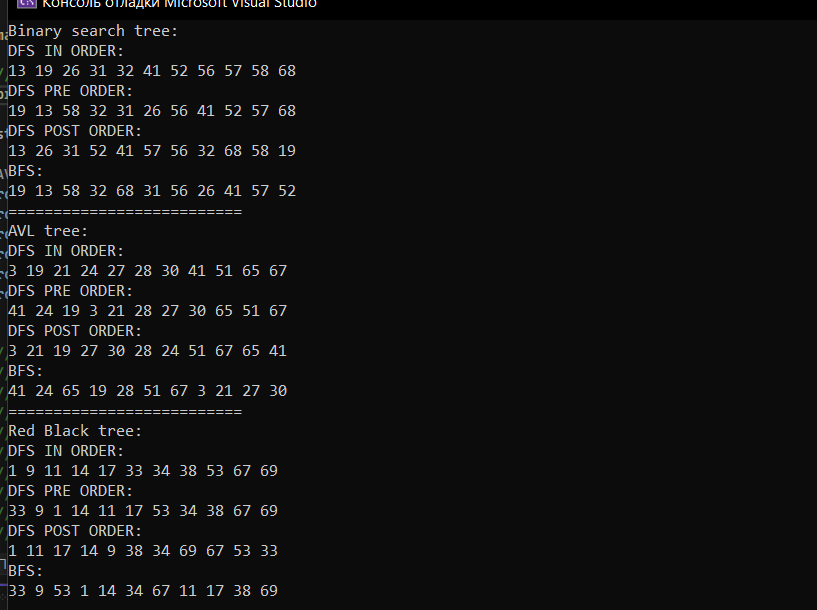


Рисунок 7 – Пример работы обходов

**Ссылки:**

Репозиторий GitHub: <https://github.com/KIRILLFABER/AICD2>

**Код:**

**main.cpp**

#include <iostream>

#include "BST.h"

#include "AVL.h"

#include "RB.h"

#include "Data.h"

#include "Traversal.cpp"

const int MAX\_VAL = 70;

const int SIZE = 10;

void printTraversals() {

srand(time(NULL));

BST::Node\* BSTtree = new BST::Node(rand() % MAX\_VAL);

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

int val = rand() % MAX\_VAL;

while (BST::search(BSTtree, val)) {

val = rand() % MAX\_VAL;

}

BSTtree = BST::insert(BSTtree, val);

}

AVL::Node\* AVLtree = new AVL::Node(rand() % MAX\_VAL);

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

int val = rand() % MAX\_VAL;

while (AVL::search(AVLtree, val)) {

val = rand() % MAX\_VAL;

}

AVLtree = AVL::insert(AVLtree, val);

}

RB::Node\* RBtree = new RB::Node(rand() % MAX\_VAL);

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

int val = rand() % MAX\_VAL;

while (RB::search(RBtree, val)) {

val = rand() % MAX\_VAL;

}

RBtree = RB::insert(RBtree, val);

}

std::cout << "Binary search tree:\n";

std::cout << "DFS IN ORDER:\n";

inOrder(BSTtree);

std::cout << "\nDFS PRE ORDER:\n";

preOrder(BSTtree);

std::cout << "\nDFS POST ORDER:\n";

postOrder(BSTtree);

std::cout << "\nBFS:\n";

breadth(BSTtree);

std::cout << "\n==========================\nAVL tree:\n";

std::cout << "DFS IN ORDER:\n";

inOrder(AVLtree);

std::cout << "\nDFS PRE ORDER:\n";

preOrder(AVLtree);

std::cout << "\nDFS POST ORDER:\n";

postOrder(AVLtree);

std::cout << "\nBFS:\n";

breadth(AVLtree);

std::cout << "\n==========================\nRed Black tree:\n";

std::cout << "DFS IN ORDER:\n";

inOrder(RBtree);

std::cout << "\nDFS PRE ORDER:\n";

preOrder(RBtree);

std::cout << "\nDFS POST ORDER:\n";

postOrder(RBtree);

std::cout << "\nBFS:\n";

breadth(RBtree);

}

int main() {

fillDataFile("DATA.csv");

printTraversals();

return 0;

}

**Data.cpp**

#include "Data.h"

#include <random>

const int MAX\_VAL = 100;

const int SIZE = 1e3;

void fillDataFile(std::string filename) {

std::ofstream data\_file;

data\_file.open(filename);

// Проверка файла на открытие

if (!data\_file.is\_open()) {

std::cout << "ERROR\n";

return;

}

data\_file << "Tree;n;h(n)\n";

srand(time(0));

// BST

BST::Node\* BSTroot = new BST::Node(rand() % MAX\_VAL);

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

BSTroot = BST::insert(BSTroot, rand() % MAX\_VAL);

data\_file << "BST;" << BST::Node::cnt << ";" << BST::height(BSTroot) << "\n";

}

// AVL

AVL::Node\* AVLroot = new AVL::Node(rand() % MAX\_VAL);

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

AVLroot = AVL::insert(AVLroot, rand() % MAX\_VAL);

data\_file << "AVL;" << AVL::Node::cnt << ";" << AVL::height(AVLroot) << "\n";

}

// RB

RB::Node\* RBroot = new RB::Node(rand() % MAX\_VAL);

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

RBroot = RB::insert(RBroot, rand() % MAX\_VAL);

data\_file << "RB;" << RB::Node::cnt << ";" << RB::height(RBroot) << "\n";

}

data\_file.close();

}

**Data.h**

#pragma once

#include <string>

#include <fstream>

#include <iostream>

#include "BST.h"

#include "AVL.h"

#include "RB.h"

void fillDataFile(std::string filename);

**BST.cpp**

#include "BST.h"

#include <iostream>

using namespace BST;

int Node::cnt = 0;

Node::Node(int key) {

this->key = key;

this->left = nullptr;

this->right = nullptr;

this->cnt++;

this->h = 1;

}

Node::~Node() {

Node::cnt--;

}

void BST::fixHeight(Node\* node) {

size\_t h\_left = height(node->left);

size\_t h\_right = height(node->right);

node->h = (h\_left > h\_right ? h\_left : h\_right) + 1;

}

size\_t BST::height(Node\* node) {

return node ? node->h : 0;

}

Node\* BST::insert(Node\* node, int key) {

if (node == nullptr) return new Node(key);

if (key < node->key) {

node->left = insert(node->left, key);

}

else {

node->right = insert(node->right, key);

}

fixHeight(node);

return node;

}

void BST::erase(Node\* root, int key) { // íå ðàáîòàåò ïðè óäàëåíèè óçëà ãäå îäèí èç äåòåé - ëèñò

Node\* curr = root;

Node\* parent = nullptr;

while (curr && curr->key != key) {

parent = curr;

if (key > curr->key) {

curr = curr->right;

}

else {

curr = curr->left;

}

}

if (!curr) return;

// Åñëè óçåë èìååò íå áîëüøå îäíîãî ðåáåíêà

if (curr->left == nullptr) {

if (parent && parent->left == curr) parent->left = curr->right;

else if (parent && parent->right == curr) parent->right = curr->right;

delete curr;

return;

}

if (curr->right == nullptr) {

if (parent && parent->left == curr) parent->left = curr->left;

else if (parent && parent->right == curr) parent->right = curr->left;

delete curr;

return;

}

// Åñëè ó óçëà åñòü 2 ðåáåíêà

if (curr->left != nullptr && curr->right != nullptr) {

Node\* newCurr = curr->right;

while (newCurr->left != nullptr) newCurr = newCurr->left;

curr->key = newCurr->key;

erase(curr->right, newCurr->key);

}

}

Node\* BST::search(Node\* root, int key) {

Node\* curr = root;

while (curr && key != curr->key) {

if (key > curr->key) {

curr = curr->right;

}

else if (key < curr->key) {

curr = curr->left;

}

}

return curr;

}

**BST.h**

#pragma once

namespace BST {

class Node {

public:

static int cnt;

size\_t h;

int key;

Node\* left;

Node\* right;

Node(int key);

~Node();

};

size\_t height(Node\* node);

void fixHeight(Node\* node);

Node\* insert(Node\* node, int key);

void erase(Node\* root, int key);

Node\* search(Node\* root, int key);

}

**AVL.cpp**

#include "AVL.h"

#include <iostream>

using namespace AVL;

int Node::cnt = 0;

Node::Node(int key) {

this->key = key;

this->left = nullptr;

this->right = nullptr;

this->h = 1;

this->cnt++;

}

Node::~Node() {

this->cnt--;

}

size\_t AVL::height(Node\* node) {

return node ? node->h : 0;

}

int AVL::bFactor(Node\* node) {

return node ? height(node->right) - height(node->left) : 0;

}

void AVL::fixHeight(Node\* node) {

size\_t h\_left = height(node->left);

size\_t h\_right = height(node->right);

node->h = (h\_left > h\_right ? h\_left : h\_right) + 1;

}

Node\* AVL::rotateRight(Node\* a) {

Node\* b = a->left;

a->left = b->right;

b->right = a;

fixHeight(a);

fixHeight(b);

return b;

}

Node\* AVL::rotateLeft(Node\* a) {

Node\* b = a->right;

a->right = b->left;

b->left = a;

fixHeight(a);

fixHeight(b);

return b;

}

Node\* AVL::balance(Node\* node) {

fixHeight(node);

if (bFactor(node) == 2) {

if (bFactor(node->right) < 0) {

node->right = rotateRight(node->right);

}

return rotateLeft(node);

}

if (bFactor(node) == -2) {

if (bFactor(node->left) > 0) {

node->left = rotateLeft(node->left);

}

return rotateRight(node);

}

return node;

}

Node\* AVL::insert(Node\* node, int key) {

if (node == nullptr) return new Node(key);

if (key < node->key) {

node->left = insert(node->left, key);

}

else {

node->right = insert(node->right, key);

}

return balance(node);

}

Node\* AVL::findMin(Node\* node) {

return node->left ? findMin(node->left) : node;

}

Node\* AVL::eraseMin(Node\* node)

{

if (node->left == 0) {

return node->right;

}

node->left = eraseMin(node->left);

return balance(node);

}

Node\* AVL::erase(Node\* node, int key) {

if (node == nullptr) return nullptr;

if (key < node->key) {

node->left = erase(node->left, key);

}

else if (key > node->key) {

node->right = erase(node->right, key);

}

else {

Node\* left = node->left;

Node\* right = node->right;

delete node;

if (right == nullptr) return left;

Node\* min = findMin(right);

min->right = eraseMin(right);

min->left = left;

return balance(min);

}

return balance(node);

}

Node\* AVL::search(Node\* root, int key) {

Node\* curr = root;

while (curr && key != curr->key) {

if (key > curr->key) {

curr = curr->right;

}

else if (key < curr->key) {

curr = curr->left;

}

}

return curr;

}

**AVL.h**

#pragma once

namespace AVL {

class Node

{

public:

static int cnt;

size\_t h;

int key;

Node\* left;

Node\* right;

Node(int key);

~Node();

};

size\_t height(Node\* node);

int bFactor(Node\* node);

void fixHeight(Node\* node);

Node\* rotateRight(Node\* a);

Node\* rotateLeft(Node\* a);

Node\* balance(Node\* node);

Node\* insert(Node\* root, int key);

Node\* erase(Node\* node, int key);

Node\* findMin(Node\* node);

Node\* eraseMin(Node\* node);

Node\* search(Node\* root, int key);

}

**RB.cpp**

#include "RB.h"

#include <iostream>

using namespace RB;

int Node::cnt = 0;

Node::Node(int key) {

this->key = key;

this->left = nullptr;

this->right = nullptr;

this->parent = nullptr;

this->cnt++;

this->color = RED;

this->h = 1;

}

Node::~Node() {

this->cnt--;

}

size\_t RB::height(Node\* node) {

return node ? node->h : 0;

}

void RB::fixHeight(Node\* node) {

size\_t h\_left = height(node->left);

size\_t h\_right = height(node->right);

node->h = std::max(h\_left, h\_right) + 1;

}

void RB::rotateLeft(Node\* root, Node\* node) {

Node\* rightChild = node->right;

node->right = rightChild->left;

if (rightChild->left != nullptr) {

rightChild->left->parent = node;

}

rightChild->parent = node->parent;

if (node->parent == nullptr) {

root = rightChild;

}

else if (node == node->parent->left) {

node->parent->left = rightChild;

}

else {

node->parent->right = rightChild;

}

rightChild->left = node;

node->parent = rightChild;

// Обновляем высоты

fixHeight(node);

fixHeight(rightChild);

}

void RB::rotateRight(Node\* root, Node\* node) {

Node\* leftChild = node->left;

node->left = leftChild->right;

if (leftChild->right != nullptr) {

leftChild->right->parent = node;

}

leftChild->parent = node->parent;

if (node->parent == nullptr) {

root = leftChild;

}

else if (node == node->parent->right) {

node->parent->right = leftChild;

}

else {

node->parent->left = leftChild;

}

leftChild->right = node;

node->parent = leftChild;

fixHeight(node);

fixHeight(leftChild);

}

void RB::fixInsert(Node\* root, Node\* node) {

while (node != root && node->parent != nullptr && node->parent->parent != nullptr && node->parent->color == RED) {

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

Node\* uncle = node->parent->parent->right;

if (uncle != nullptr && uncle->color == RED) {

node->parent->color = BLACK;

uncle->color = BLACK;

node->parent->parent->color = RED;

node = node->parent->parent;

}

else {

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

node = node->parent;

rotateLeft(root, node);

}

node->parent->color = BLACK;

node->parent->parent->color = RED;

rotateRight(root, node->parent->parent);

}

}

else {

Node\* uncle = node->parent->parent->left;

if (uncle != nullptr && uncle->color == RED) {

node->parent->color = BLACK;

uncle->color = BLACK;

node->parent->parent->color = RED;

node = node->parent->parent;

}

else {

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

node = node->parent;

rotateRight(root, node);

}

node->parent->color = BLACK;

node->parent->parent->color = RED;

rotateLeft(root, node->parent->parent);

}

}

}

root->color = BLACK;

}

Node\* RB::insert(Node\* root, int key) {

Node\* node = new Node(key);

Node\* parent = nullptr;

Node\* current = root;

while (current != nullptr) {

parent = current;

if (key < current->key) {

current = current->left;

}

else {

current = current->right;

}

}

node->parent = parent;

if (parent == nullptr) {

root = node;

}

else if (key < parent->key) {

parent->left = node;

}

else {

parent->right = node;

}

fixInsert(root, node);

return root;

}

Node\* RB::search(Node\* root, int key) {

Node\* curr = root;

while (curr && key != curr->key) {

if (key > curr->key) {

curr = curr->right;

}

else if (key < curr->key) {

curr = curr->left;

}

}

return curr;

}

Node\* RB::minimum(Node\* node) {

while (node->left != nullptr) {

node = node->left;

}

return node;

}

void RB::fixDelete(Node\* root, Node\* node) {

while (node != root && node->color == BLACK) {

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

Node\* sibling = node->parent->right;

if (sibling->color == RED) {

sibling->color = BLACK;

node->parent->color = RED;

rotateLeft(root, node->parent);

sibling = node->parent->right;

}

if ((sibling->left == nullptr || sibling->left->color == BLACK) &&

(sibling->right == nullptr || sibling->right->color == BLACK)) {

sibling->color = RED;

node = node->parent;

}

else {

if (sibling->right == nullptr || sibling->right->color == BLACK) {

sibling->left->color = BLACK;

sibling->color = RED;

rotateRight(root, sibling);

sibling = node->parent->right;

}

sibling->color = node->parent->color;

node->parent->color = BLACK;

if (sibling->right != nullptr) {

sibling->right->color = BLACK;

}

rotateLeft(root, node->parent);

node = root;

}

}

else {

Node\* sibling = node->parent->left;

if (sibling->color == RED) {

sibling->color = BLACK;

node->parent->color = RED;

rotateRight(root, node->parent);

sibling = node->parent->left;

}

if ((sibling->left == nullptr || sibling->left->color == BLACK) &&

(sibling->right == nullptr || sibling->right->color == BLACK)) {

sibling->color = RED;

node = node->parent;

}

else {

if (sibling->left == nullptr || sibling->left->color == BLACK) {

sibling->right->color = BLACK;

sibling->color = RED;

rotateLeft(root, sibling);

sibling = node->parent->left;

}

sibling->color = node->parent->color;

node->parent->color = BLACK;

if (sibling->left != nullptr) {

sibling->left->color = BLACK;

}

rotateRight(root, node->parent);

node = root;

}

}

}

node->color = BLACK;

}

Node\* RB::deleteNode(Node\* root, int key) {

Node\* nodeToDelete = search(root, key);

if (nodeToDelete == nullptr) {

return root;

}

Node\* y = nodeToDelete;

Node\* x;

Node\* xParent;

if (nodeToDelete->left == nullptr || nodeToDelete->right == nullptr) {

y = nodeToDelete;

}

else {

y = minimum(nodeToDelete->right);

}

if (y->left != nullptr) {

x = y->left;

}

else {

x = y->right;

}

if (x != nullptr) {

x->parent = y->parent;

}

if (y->parent == nullptr) {

root = x;

}

else if (y == y->parent->left) {

y->parent->left = x;

}

else {

y->parent->right = x;

}

if (y != nodeToDelete) {

nodeToDelete->key = y->key;

}

if (y->color == BLACK) {

fixDelete(root, x);

}

delete y;

return root;

}

**RB.h**

#pragma once

enum Color { RED, BLACK };

namespace RB {

class Node {

public:

static int cnt;

int key;

Node\* left;

Node\* right;

Node\* parent;

Color color;

size\_t h;

size\_t bh;

Node(int key);

~Node();

};

size\_t height(Node\* node);

void fixHeight(Node\* node);

void rotateLeft(Node\* root, Node\* node);

void rotateRight(Node\* root, Node\* node);

void fixInsert(Node\* root, Node\* node);

Node\* insert(Node\* node, int key);

Node\* search(Node\* root, int key);

Node\* minimum(Node\* node);

void fixDelete(Node\* root, Node\* node);

Node\* deleteNode(Node\* root, int key);

}