Министерство образования и науки Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования

«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт вычислительной математики и информационных технологий

ОТЧЕТ

По научно-исследовательской работе

Дисциплина «Теория конечных графов»

Оптимальные деревья бинарного поиска

Выполнили:

Студенты группы 09-832

Гусев Виталий

Шептур Ангелина

Турдиев Мухамадовуд

Ларичев Никита

Проверил:

доц. к.ф.-м.н. Нурмеев Наиль Нуреевич

Казань

2020

Оглавление

[Введение. 2](#_Toc41039582)

[Бинарное дерево. 3](#_Toc41039583)

[Красно-черное дерево. 5](#_Toc41039584)

[АВЛ-дерево. 7](#_Toc41039585)

[Анализ проблемной области. 8](#_Toc41039586)

[Расчетная часть 9](#_Toc41039587)

[Реализация бинарного дерева поиска на C++ 9](#_Toc41039588)

[Реализация красно-черного дерева на C++. 13](#_Toc41039589)

[Реализация АВЛ-дерева на C++. 23](#_Toc41039590)

[Результаты тестов. 26](#_Toc41039591)

[Заключение. 27](#_Toc41039592)

[Список использованной литературы. 28](#_Toc41039593)

[Приложение. 29](#_Toc41039594)

[Справка о трудовом участии всех участников проекта 29](#_Toc41039595)

[Листинги программ 30](#_Toc41039596)

# Введение.

**Бинарные деревья** являются неотъемлемой частью компьютерных наук. Они применяются для быстрого поиска в базах данных, сортировки данных, кодирования и вычисления арифметических выражений.

Цель данного проекта – изучение понятия оптимального дерева бинарного поиска, формирование, особенности работы с памятью и доступа к данным, реализация алгоритмов обхода бинарных деревьев и анализ результатов.

В процессе будут изучены бинарные деревья поиска, красно-черные деревья и АВЛ-деревья.

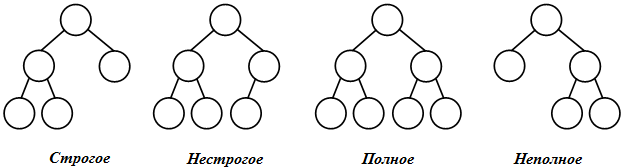
# Бинарное дерево.

**Бинарное (двоичное) дерево** – это динамическая структура данных, которая состоит из элементов, состоящие из информационного поля и не более двух ссылок на различные бинарные поддеревья. Данная структура представляет собой дерево, состоящее из вершин, содержащие не более двух потомков. На каждый элемент дерева имеется ровно одна ссылка.

Каждая *вершина* *бинарного дерева* является структурой, состоящей из четырех видов полей. Содержимым этих полей будут соответственно:

* информационное поле (ключ вершины);
* служебное поле (их может быть несколько или ни одного);
* указатель на *левое поддерево*;
* указатель на правое *поддерево*.

*По* *степени вершин* *бинарные деревья* делятся на:

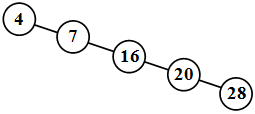


* *строгие* – вершины дерева имеют степень ноль (у листьев) или два (у узлов);
* *нестрогие* – вершины дерева имеют степень ноль (у листьев), один или два (у узлов).

В общем случае у *бинарного дерева* на k -м уровне может быть до 2k- 1 вершин. *Бинарное дерево* называется *полным*, если оно содержит только полностью заполненные уровни. В противном случае оно является *неполным*.

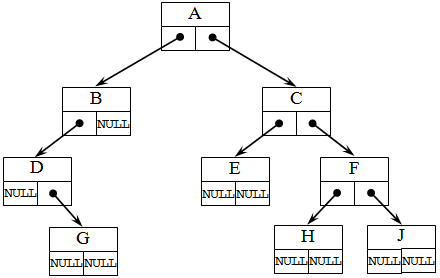
*Дерево* называется *сбалансированным*, если длины всех путей от корня к внешним вершинам равны между собой. *Дерево* называется *почти сбалансированным*, если длины всевозможных путей от корня к внешним вершинам отличаются не более, чем на единицу.

*Бинарное дерево* может представлять собой *пустое множество*. *Бинарное дерево* может выродиться в *список*.



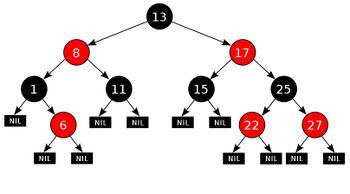
Список как частный случай бинарного дерева

Структура дерева отражается во входном потоке данных так: каждой вводимой пустой связи соответствует условный символ, например, '\*' (звездочка). При этом сначала описываются левые потомки, затем, правые. Для структуры *бинарного дерева*, представленного на следующем рисунке, *входной* *поток* имеет вид: ABD\*G\*\*\*CE\*\*FH\*\*J\*\*.

  
Адресация в бинарном дереве

# Красно-черное дерево.

**Красно-чёрное дерево** (англ. *red-black tree*) — это двоичное дерево поиска, в котором каждый узел имеет «красный» или «черный» цвет, за счет этого осуществляется баланс.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:RBT.jpg)

Пример красно-чёрного дерева.

Все листья, которые не содержат данных и являются фиктивными, относятся к дереву и имею черный цвет.

Фиктивные листья можно сделать одним общим фиктивным листом в целях экономии памяти.

Красно-чёрным называется бинарное поисковое дерево, у которого каждому узлу сопоставлен дополнительный атрибут — цвет и для которого выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
5. Чёрный узел может иметь чёрного родителя

В книге Кормена "Алгоритмы: построение и анализ" дается немного иное определение красно-черного дерева, а именно:

Двоичное дерево поиска является красно-чёрным, если обладает следующими свойствами:

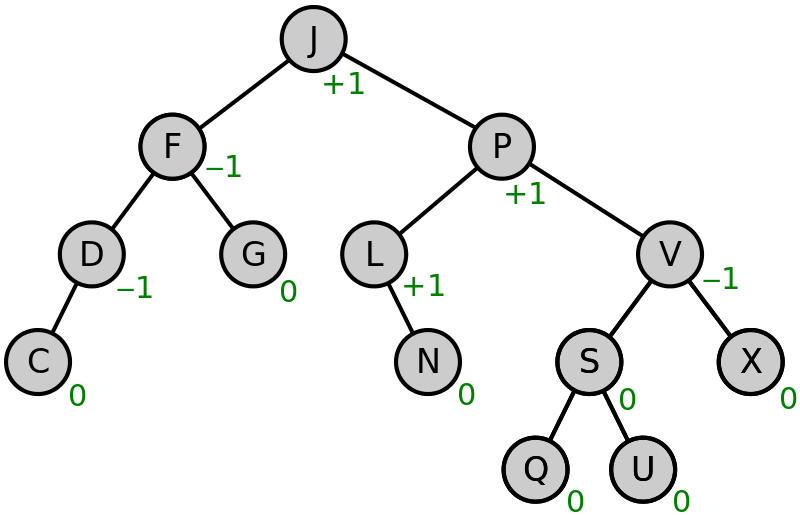
1. Каждая вершина — либо красная, либо черная
2. Каждый лист — черный
3. Если вершина красная, оба ее ребенка черные
4. Все пути, идущие от корня к листьям, содержат одинаковое количество черных вершин

То, что только черная вершина может иметь красных детей, совместно с 4-ым свойством говорит о том, что корень дерева должен быть черным, а значит определения можно считать эквивалентными.

# АВЛ-дерево.

**АВЛ-дерево** (англ. *AVL-Tree*) — сбалансированное двоичное дерево поиска, в котором баланс достигается за счет того, что у каждой вершины высота ее двух поддеревьев различается не более чем на 1.

АВЛ-деревья названы по первым буквам фамилий их изобретателей, Г. М. Адельсона-Вельского и Е. М. Ландиса, которые впервые предложили использовать АВЛ-деревья в 1962 году.



# Анализ проблемной области.

При создании проекта была поставлена задача реализации бинарного дерева, красно-черного дерева и АВЛ- дерева. Реализация алгоритмов обхода дерева, поиск минимального и максимального элемента, добавление элемента, удаление узла. Будут проведены сравнения эффективности алгоритмов между бинарным, красно-черным и АВЛ деревом на массивах данных. Эта задача не актуальна, так как все это реализовано и изучено программистами и учеными, однако очень важная, так как без подобных знаний не обходятся высококвалифицированные программисты и крупные компании учитывают подобные знания при принятии новых сотрудников. Реализация этого проекта позволит понять, как устроены деревья и научит с ними работать.

# Расчетная часть

## Реализация бинарного дерева поиска на C++

Бинарное дерево поиска имеет следующую структуру. Оно состоит из поля данных и двух указателей на левое и правое поддерево.

struct BSTnode {

int key; // поле данных

struct BSTnode\* left; // левый потомок

struct BSTnode\* right; // правый потомок

};

Существует три типа обхода дерева: прямой, симметричный и обратный. При прямом обходе идет обращение к корню дерева, потом к левому и правому поддереву. При симметричном обходе идет обращение к левому поддереву, корню и правому поддереву. В обратном обходе идет обращение к левому, правому поддереву и к корню дерева. Все обходы проходят рекурсивным способом.

В процессе выяснилось, что симметричный обход выводит узлы в отсортированном порядке. Таким образом, хорошо подходит для сортировки входных данных. При обратном обходе, можно заметить, что узлы выводятся, у которых нет детей, что хорошо подходит для удаления дерева и ни один узел не будет висеть в памяти. Прямой обход подходит для копирования дерева.

//префиксный вид

void treeprintpref(BSTnode\* tree) {

if (tree != NULL) { //Пока не встретится пустой узел

cout << tree->key << " "; //Отображаем корень дерева

treeprintpref(tree->left); //Рекурсивная функция для левого поддерева

treeprintpref(tree->right); //Рекурсивная функция для правого поддерева

}

}

//инфиксный вид

void treeprintinf(BSTnode\* tree) {

if (tree != NULL) { //Пока не встретится пустой узел

treeprintinf(tree->left); //Рекурсивная функция для левого поддерева

cout << tree->key << " "; //Отображаем корень дерева

treeprintinf(tree->right); //Рекурсивная функция для правого поддерева

}

}

//постфиксный вид

void treeprintpost(BSTnode\* tree) {

if (tree != NULL) { //Пока не встретится пустой узел

treeprintpost(tree->left); //Рекурсивная функция для левого поддерева

treeprintpost(tree->right); //Рекурсивная функция для правого поддерева

cout << tree->key << " "; //Отображаем корень дерева

}

}

Добавление узла достигается следующим образом. Если дерева нет, то формируется корень, которому задаются данные и инициализируются пустые правый и левый потомок. Далее, если корень есть, проверяется значение элемента с полем узла. Если добавляемый элемент меньше, он идет в левое поддерево, иначе в правое. В результате функции возвращается образованное дерево.

struct BSTnode\* addnode(int x, BSTnode\* tree) {

if (tree == NULL) { // Если дерева нет, то формируем корень

tree = new BSTnode; // память под узел

tree->key = x; // поле данных

tree->left = NULL;

tree->right = NULL; // ветви инициализируем пустотой

}

else if (x < tree->key) // условие добавление левого потомка

tree->left = addnode(x, tree->left);

else // условие добавление правого потомка

tree->right = addnode(x, tree->right);

return(tree);

}

При удалении поддерева передается ссылка на поддерево и рекурсивно вызывается в правые и левые поддеревья и удаляет узлы, пока не достигнет пустой ссылки.

void DeleteTree(BSTnode\* tree) {

if (tree != NULL) {

DeleteTree(tree->left);

DeleteTree(tree->right);

delete tree;

}

}

Для поиска минимального элемента достаточно идти влево по дереву до пустого элемента. Таким образов для поиска максимального элемента нужно двигаться вправо.

BSTnode\* MinValue(BSTnode\* Tree)

{

if (Tree->left != NULL) {

return MinValue(Tree->left);

}

else {

cout << Tree->key;

}

}

BSTnode\* MaxValue(BSTnode\* Tree)

{

if (Tree->right != NULL) {

return MaxValue(Tree->right);

}

else

{

cout << Tree->key;

}

}

Для подсчета числа узлов в дереве рекурсивно суммируются правые и левые поддеревья, прибавляя 1 при каждом узле.

int NumberOfNodes(BSTnode\* Tree)

{

if (Tree == NULL) return 0;

return NumberOfNodes(Tree->left) + 1 + NumberOfNodes(Tree->right);

}

Для вычисления высоты дерева рекурсивно вычисляется глубина правого и левого поддерева. Если левое поддерево больше, возвращается его глубина, иначе глубина правого поддерева.

int HeightBTree(BSTnode\* Tree)

{

int x = 0, y = 0;

if (Tree == NULL) return 0; //пустое дерево или дошли до листа

if (Tree->left) x = HeightBTree(Tree->left); //высота левого поддерева

if (Tree->right) y = HeightBTree(Tree->right); //высота правого поддерева

if (x > y) return x + 1; //+1 от корня к левому поддереву

else return y + 1; //+1 от корня к правому поддереву

}

Для поиска элемента в дереве сравнивается значение элемента и узел дерева. Если элемент меньше узла, поиск продолжается в левом поддереве, иначе в правом, пока не найдется элемент или не пойдет до пустого значения.

BSTnode\* Search(BSTnode\* Tree, int x)

{

if (Tree == NULL) return NULL;

if (Tree->key == x) return Tree;

if (x < Tree->key) return Search(Tree->left, x);

else

return Search(Tree->right, x);

}

Для удаления узла в дереве идет поиск элемента в дереве. В случае его нахождения создается вспомогательная ссылка. Если справа нет узла, то вспомогательному элементу присваивается левый узел, удаляется узел и возвращается эта вспомогательная ссылка. Иначе создается вторая вспомогательная ссылка на правый элемент. Если эта ссылка не имеет левого потомка, то этому левому потомку присваивается левый узел и первой вспомогательной ссылке присваивается вторая. Иначе создается третья вспомогательная ссылка на левый узел второй ссылки. Далее, пока третья вспомогательная ссылка не будет указывать на пустой элемент, второй ссылке присваивается третья, а третьей ее левый потомок. Далее левому потомку второй ссылки присваивается правый потомок третьей ссылки. Третьей ссылке правому и левому потомку присваивается правый и левый потомок узла и третья ссылка присваивается первой ссылке. В результате удаляется узел и возвращается первая ссылка.

BSTnode\* DeleteNode(BSTnode\* node, int x) {

if (node == NULL)

return node;

if (x == node->key) {

BSTnode\* help1;

if (node->right == NULL)

help1 = node->left;

else {

BSTnode\* help2 = node->right;

if (help2->left == NULL) {

help2->left = node->left;

help1 = help2;

}

else {

BSTnode\* pmin = help2->left;

while (pmin->left != NULL) {

help2 = pmin;

pmin = help2->left;

}

help2->left = pmin->right;

pmin->left = node->left;

pmin->right = node->right;

help1 = pmin;

}

}

delete node;

return help1;

}

else if (x < node->key)

node->left = DeleteNode(node->left, x);

else

node->right = DeleteNode(node->right, x);

return node;

}

## Реализация красно-черного дерева на C++.

Структура красно-черного дерева состоит из ссылки на корень, числа узлов дерева и структуры узла дерева. Структура узла состоит из ссылки на правое и левое поддерево, поля данных и булевской переменной, определяющей является ли узел красным.

class RBtree {

struct node\_st { node\_st\* p1, \* p2; int value; bool red; }; // структура узла

node\_st\* tree\_root; //!< корень

int nodes\_count; //!< число узлов дерева

private:

node\_st\* NewNode(int value); //!< выделение новой вешины

void DelNode(node\_st\*); //!< удаление вершины

void Clear(node\_st\*); //!< снос дерева (рекурсивная часть)

node\_st\* Rotate21(node\_st\*); //!< вращение влево

node\_st\* Rotate12(node\_st\*); //!< вращение вправо

void BalanceInsert(node\_st\*\*); //!< балансировка вставки

bool BalanceRemove1(node\_st\*\*); //!< левая балансировка удаления

bool BalanceRemove2(node\_st\*\*); //!< правая балансировка удаления

bool Insert(int, node\_st\*\*); //!< рекурсивная часть вставки

bool GetMin(node\_st\*\*, node\_st\*\*); //!< найти и убрать максимальный узел поддерева

bool Remove(node\_st\*\*, int); //!< рекурсивная часть удаления

public: // отладочная часть

enum check\_code { error\_balance, error\_struct, ok }; // код ошибки

void Show(); //!< вывод дерева

check\_code Check(); //!< проверка дерева

bool TreeWalk(bool\*, int); //!< обход дерева и сверка значений с массивом

private: // отладочная часть

void Show(node\_st\*, int, char); //!< вывод дерева, рекурсивная часть

check\_code Check(node\_st\*, int, int&);//!< проверка дерева (рекурсивная часть)

bool TreeWalk(node\_st\*, bool\*, int); //!< обход дерева и сверка значений с массивом (рекурсивная часть)

public:

RBtree();

~RBtree();

void Clear(); //!< снести дерево

bool Find(int); //!< найти значение

void Insert(int); //!< вставить значение

void Remove(int); //!< удалить значение

int GetNodesCount(); //!< узнать число узлов

};

RBtree::RBtree()

{

tree\_root = 0;

nodes\_count = 0;

}

RBtree::~RBtree()

{

Clear(tree\_root);

}

int RBtree::GetNodesCount()

{

return nodes\_count;

}

// выделение новой вешины

RBtree::node\_st\* RBtree::NewNode(int value)

{

nodes\_count++;

node\_st\* node = new node\_st;

node->value = value;

node->p1 = node->p2 = 0;

node->red = true;

return node;

}

// удаление вершины

void RBtree::DelNode(node\_st\* node)

{

nodes\_count--;

delete node;

}

// снос дерева (рекурсивная часть)

void RBtree::Clear(node\_st\* node)

{

if (!node) return;

Clear(node->p1);

Clear(node->p2);

DelNode(node);

}

// вывод дерева, рекурсивная часть

//! \param node узел

//! \param depth глубина

//! \param dir значёк

//! \code Show(root,0,'\*'); \endcode

void RBtree::Show(node\_st\* node, int depth, char dir)

{

int n;

if (!node) return;

for (n = 0; n < depth; n++) putchar(' ');

printf("%c[%d] (%s)\n", dir, node->value, node->red ? "red" : "black");

Show(node->p1, depth + 2, '-');

Show(node->p2, depth + 2, '+');

}

// вращение влево

//! \param index индеск вершины

//! \result новая вершина дерева

RBtree::node\_st\* RBtree::Rotate21(node\_st\* node)

{

node\_st\* p2 = node->p2;

node\_st\* p21 = p2->p1;

p2->p1 = node;

node->p2 = p21;

return p2;

}

// вращение вправо

//! \param index индеск вершины

//! \result новая вершина дерева

RBtree::node\_st\* RBtree::Rotate12(node\_st\* node)

{

node\_st\* p1 = node->p1;

node\_st\* p12 = p1->p2;

p1->p2 = node;

node->p1 = p12;

return p1;

}

// балансировка вершины

void RBtree::BalanceInsert(node\_st\*\* root)

{

node\_st\* p1, \* p2, \* px1, \* px2;

node\_st\* node = \*root;

if (node->red) return;

p1 = node->p1;

p2 = node->p2;

if (p1 && p1->red) {

px2 = p1->p2; // задача найти две рядом стоящие крастне вершины

if (px2 && px2->red) p1 = node->p1 = Rotate21(p1);

px1 = p1->p1;

if (px1 && px1->red) {

node->red = true;

p1->red = false;

if (p2 && p2->red) { // отделаемся перекраской вершин

px1->red = true;

p2->red = false;

return;

}

\*root = Rotate12(node);

return;

}

}

// тоже самое в другую сторону

if (p2 && p2->red) {

px1 = p2->p1; // задача найти две рядом стоящие крастне вершины

if (px1 && px1->red) p2 = node->p2 = Rotate12(p2);

px2 = p2->p2;

if (px2 && px2->red) {

node->red = true;

p2->red = false;

if (p1 && p1->red) { // отделаемся перекраской вершин

px2->red = true;

p1->red = false;

return;

}

\*root = Rotate21(node);

return;

}

}

}

bool RBtree::BalanceRemove1(node\_st\*\* root)

{

node\_st\* node = \*root;

node\_st\* p1 = node->p1;

node\_st\* p2 = node->p2;

if (p1 && p1->red) {

p1->red = false; return false;

}

if (p2 && p2->red) { // случай 1

node->red = true;

p2->red = false;

node = \*root = Rotate21(node);

if (BalanceRemove1(&node->p1)) node->p1->red = false;

return false;

}

unsigned int mask = 0;

node\_st\* p21 = p2->p1;

node\_st\* p22 = p2->p2;

if (p21 && p21->red) mask |= 1;

if (p22 && p22->red) mask |= 2;

switch (mask)

{

case 0: // случай 2 - if((!p21 || !p21->red) && (!p22 || !p22->red))

p2->red = true;

return true;

case 1:

case 3: // случай 3 - if(p21 && p21->red)

p2->red = true;

p21->red = false;

p2 = node->p2 = Rotate12(p2);

p22 = p2->p2;

case 2: // случай 4 - if(p22 && p22->red)

p2->red = node->red;

p22->red = node->red = false;

\*root = Rotate21(node);

}

return false;

}

bool RBtree::BalanceRemove2(node\_st\*\* root)

{

node\_st\* node = \*root;

node\_st\* p1 = node->p1;

node\_st\* p2 = node->p2;

if (p2 && p2->red) { p2->red = false; return false; }

if (p1 && p1->red) { // случай 1

node->red = true;

p1->red = false;

node = \*root = Rotate12(node);

if (BalanceRemove2(&node->p2)) node->p2->red = false;

return false;

}

unsigned int mask = 0;

node\_st\* p11 = p1->p1;

node\_st\* p12 = p1->p2;

if (p11 && p11->red) mask |= 1;

if (p12 && p12->red) mask |= 2;

switch (mask) {

case 0: // случай 2 - if((!p12 || !p12->red) && (!p11 || !p11->red))

p1->red = true;

return true;

case 2:

case 3: // случай 3 - if(p12 && p12->red)

p1->red = true;

p12->red = false;

p1 = node->p1 = Rotate21(p1);

p11 = p1->p1;

case 1: // случай 4 - if(p11 && p11->red)

p1->red = node->red;

p11->red = node->red = false;

\*root = Rotate12(node);

}

return false;

}

bool RBtree::Find(int value)

{

node\_st\* node = tree\_root;

while (node) {

if (node->value == value) return true;

node = node->value > value ? node->p1 : node->p2;

}

return false;

}

// рекурсивная часть вставки

//! \result true если изменений небыло или балансировка в данной вершине не нужна

bool RBtree::Insert(int value, node\_st\*\* root)

{

node\_st\* node = \*root;

if (!node) \*root = NewNode(value);

else {

if (value == node->value) return true;

if (Insert(value, value < node->value ? &node->p1 : &node->p2)) return true;

BalanceInsert(root);

}

return false;

}

// найти и убрать максимальный узел поддерева

//! \param root корень дерева в котором надо найти элемент

//! \retval res эелемент который был удалён

//! \result true если нужен баланс

bool RBtree::GetMin(node\_st\*\* root, node\_st\*\* res)

{

node\_st\* node = \*root;

if (node->p1) {

if (GetMin(&node->p1, res)) return BalanceRemove1(root);

}

else {

\*root = node->p2;

\*res = node;

return !node->red;

}

return false;

}

// рекурсивная часть удаления

//! \result true если нужен баланс

bool RBtree::Remove(node\_st\*\* root, int value)

{

node\_st\* t, \* node = \*root;

if (!node) return false;

if (node->value < value) {

if (Remove(&node->p2, value)) return BalanceRemove2(root);

}

else if (node->value > value) {

if (Remove(&node->p1, value)) return BalanceRemove1(root);

}

else {

bool res;

if (!node->p2) {

\*root = node->p1;

res = !node->red;

}

else {

res = GetMin(&node->p2, root);

t = \*root;

t->red = node->red;

t->p1 = node->p1;

t->p2 = node->p2;

if (res) res = BalanceRemove2(root);

}

DelNode(node);

return res;

}

return 0;

}

// вывод дерева

void RBtree::Show()

{

printf("[tree]\n");

Show(tree\_root, 0, '\*');

}

// функция вставки

void RBtree::Insert(int value)

{

Insert(value, &tree\_root);

if (tree\_root) tree\_root->red = false;

}

// удаление узла

void RBtree::Remove(int value)

{

Remove(&tree\_root, value);

}

// снос дерева

void RBtree::Clear()

{

Clear(tree\_root);

tree\_root = 0;

}

// проверка дерева (рекурсивная часть)

//! \param tree дерево

//! \param d текущая чёрная глубина

//! \param h эталонная чёрная глубина

//! \result 0 или код ошибки

RBtree::check\_code RBtree::Check(node\_st\* tree, int d, int& h)

{

if (!tree) {

// количество чёрных вершин на любом пути одинаковое

if (h < 0) h = d;

return h == d ? ok : error\_balance;

}

node\_st\* p1 = tree->p1;

node\_st\* p2 = tree->p2;

// красная вершина должна иметь чёрных потомков

if (tree->red && (p1 && p1->red || p2 && p2->red)) return error\_struct;

if (p1 && tree->value<p1->value || p2 && tree->value>p2->value) return error\_struct;

if (!tree->red) d++;

check\_code n = Check(p1, d, h); if (n) return n;

return Check(p2, d, h);

}

// проверка дерева

RBtree::check\_code RBtree::Check()

{

int d = 0;

int h = -1;

if (!tree\_root) return ok;

if (tree\_root->red) return error\_struct;

return Check(tree\_root, d, h);

}

// обход дерева и сверка значений с массивом (рекурсивная часть)

//! \param node корень дерева

//! \param array массив для сверки

//! \param size размер массива

bool RBtree::TreeWalk(node\_st\* node, bool\* array, int size)

{

if (!node) return false;

int value = node->value;

if (value < 0 || value >= size || !array[value]) return true;

array[value] = false;

return TreeWalk(node->p1, array, size) || TreeWalk(node->p2, array, size);

}

// обход дерева и сверка значений с массивом

//! \param array массив для сверки

//! \param size размер массива

bool RBtree::TreeWalk(bool\* array, int size)

{

if (TreeWalk(tree\_root, array, size)) return true;

for (int n = 0; n < size; n++) if (array[n]) return true;

return false;

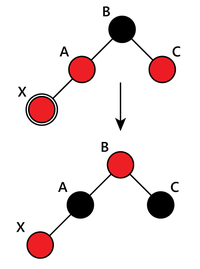
}

**Вставка вершины** происходит следующим способом.

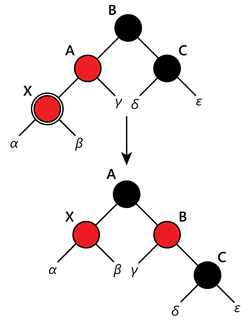
На картинках идет работа с узлом X.

Новый элемент вставляется вместо листа, поэтому для выбора места вставки идем от корня, пока указатель на следующего сына не станет null. Вставляется новый элемент с красным цветом вместо него. Если отец нового элемента черный, то свойства дерева не нарушены. Если он красный, то нарушается свойство «У красного узла родительский узел — чёрный», чтобы исправить нужно рассмотреть следующие случаи:

1. «Дядя» этого узла тоже красный. Поэтому для сохранения свойств «У красного узла родительский узел — чёрный» и «Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов», «дед» перекрашивается в красный цвет, а «отец» и «дядя» в черный цвет. Тогда черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин «отцы» черные. Далее проверяется нарушение балансировки дерева. Если в результате перекрашиваний дошли до корня, то в нем в любом случае нужно ставить черный цвет, чтобы дерево удовлетворяло свойству «Корень и конечные узлы (листья) дерева — чёрные».



2. «Дядя» черный. Если выполнить только перекрашивание, то нарушается постоянство черной высоты дерева по всем ветвям. Поэтому выполняется поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, свойство «У красного узла родительский узел — чёрный» и постоянство высоты сохраняются.



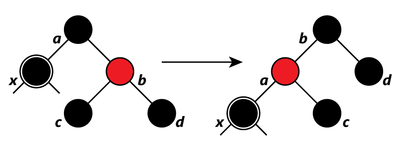
**Удаление вершины.**

Могут возникнуть три случая при удалении вершины в зависимости от количества её детей:

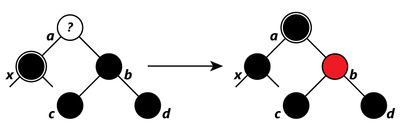
* Если у вершины нет детей, то меняется указатель на неё у родителя на null.
* Если у неё только один ребёнок, то делается у родителя ссылка на него вместо этой вершины.
* Если же имеются оба ребёнка, то идет поиск вершины со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины, и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами, сначала идет переход в правое поддерево, а после спуск вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляется данная вершина описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Проверяется балансировка дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

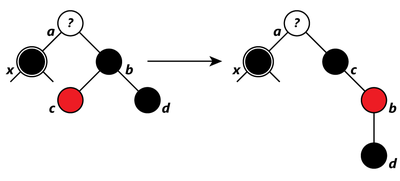
* Если брат этого ребёнка красный, то делается вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Он красится в чёрный, а отец — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас X имеет чёрного брата и красного отца. Таким образом, можно перейти к следующему шагу.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Untitled-3.png)

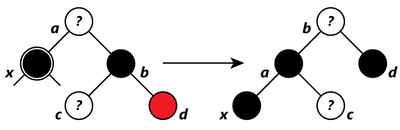
* Если брат текущей вершины был чёрным, то получается три случая:
  + Оба ребёнка у брата чёрные. Брат красится в красный цвет и рассматривается далее отец вершины. Он становится черным, это не повлияет на количество чёрных узлов на путях, проходящих через b, но добавит один к числу чёрных узлов на путях, проходящих через x, восстанавливая тем самым влияние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Untitled-4.png)

* + Если у брата правый ребёнок чёрный, а левый красный, то брат и его левый сын перекрашиваются и делается вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и далее рассматривается следующий случай. Ни x, ни его отец не влияют на эту трансформацию.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Untitled-5.png)

* + Если у брата правый ребёнок красный, то брат перекрашивается в цвет отца, его ребёнок и отец — в чёрный, делается вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Происходит выход из алгоритма.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Untitled-6.png)

Продолжается тот же алгоритм, пока текущая вершина чёрная и не дошли до корня дерева. Из рассмотренных случаев ясно, что при удалении выполняется не более трёх вращений.

## Реализация АВЛ-дерева на C++.

Структура узла состоит из поля значения, ссылки на правый и левый узел и высоты.

struct nodeavl // структура для представления узлов дерева

{

int key;

unsigned char height;

nodeavl\* left;

nodeavl\* right;

nodeavl(int k) { key = k; left = right = 0; height = 1; }

};

unsigned char height(nodeavl\* p)

{

return p ? p->height : 0;

}

int bfactor(nodeavl\* p)

{

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

}

void fixheight(nodeavl\* p)

{

unsigned char hl = height(p->left);

unsigned char hr = height(p->right);

p->height = (hl > hr ? hl : hr) + 1;

}

nodeavl\* rotateright(nodeavl\* p) // правый поворот вокруг p

{

nodeavl\* q = p->left;

p->left = q->right;

q->right = p;

fixheight(p);

fixheight(q);

return q;

}

nodeavl\* rotateleft(nodeavl\* q) // левый поворот вокруг q

{

nodeavl\* p = q->right;

q->right = p->left;

p->left = q;

fixheight(q);

fixheight(p);

return p;

}

nodeavl\* balance(nodeavl\* p) // балансировка узла p

{

fixheight(p);

if (bfactor(p) == 2)

{

if (bfactor(p->right) < 0)

p->right = rotateright(p->right);

return rotateleft(p);

}

if (bfactor(p) == -2)

{

if (bfactor(p->left) > 0)

p->left = rotateleft(p->left);

return rotateright(p);

}

return p; // балансировка не нужна

}

nodeavl\* insert(nodeavl\* p, int k) // вставка ключа k в дерево с корнем p

{

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

if (k < p->key)

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

else

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

return balance(p);

}

nodeavl\* findmin(nodeavl\* p) // поиск узла с минимальным ключом в дереве p

{

return p->left ? findmin(p->left) : p;

}

nodeavl\* removemin(nodeavl\* p) // удаление узла с минимальным ключом из дерева p

{

if (p->left == 0)

return p->right;

p->left = removemin(p->left);

return balance(p);

}

nodeavl\* remove(nodeavl\* p, int k) // удаление ключа k из дерева p

{

if (!p) return 0;

if (k < p->key)

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

else if (k > p->key)

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

else // k == p->key

{

nodeavl\* q = p->left;

nodeavl\* r = p->right;

delete p;

if (!r) return q;

nodeavl\* min = findmin(r);

min->right = removemin(r);

min->left = q;

return balance(min);

}

return balance(p);

}

void treeprint(nodeavl\* p) {

if (p != NULL) { //Пока не встретится пустой узел

cout << p->key << " "; //Отображаем корень дерева

treeprint(p->left); //Рекурсивная функция для левого поддерева

treeprint(p->right); //Рекурсивная функция для правого поддерева

}

}

После добавления узла сравниваются высоты правого и левого поддерева и, в случае разницы высот на 2, т.е. dif[i]=|h(L)−h(R)|=2, происходит процесс **балансировки вершины**. Меняются связи предок-потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева dif[i]=||h(L)−h(R)|⩽1, иначе не меняется ничего.

Для балансировки используются вращения в право или в лево.

Для **добавления ключа** **key** идет процесс спуска по дереву, как и при поиске этого ключа. Если находимся в вершине **a** и нужно идти в поддерево, которого нет, то ключ **key** делается листом, а вершина **a** его корнем. Далее нужно подняться вверх по пути поиска и пересчитывать баланс у вершин. Если поднялись в вершину **i** из правого поддерева, то **dif[i]** уменьшается на один, если из левого, то увеличивается на один. Если пришли в вершину и ее баланс стал равным нулю, это значит, что высота поддерева не изменилась и подъем останавливается. Если пришли в вершину и ее баланс стал равным -1 или 1, это значит, что высота поддерева изменилась и подъем продолжается. Если пришли в вершину и баланс стал равным -2 или 2, то делается одно из вращений и, если после вращения баланс стал равным нулю, то происходит остановка, иначе продолжается подъем.

**Алгоритм удаления**. Если вершина является листом, то она удаляется, иначе идет поиск самой близкой по значению вершины **a**, она перемещается на место удаляемой вершины и удаляется вершина **a**. От удаленной вершины нужно подняться к корню и пересчитать баланс у вершин. Если поднялись в вершину **i** из правого поддерева, то **dif[i]** увеличивается на один, если из левого, то уменьшается на один. Если пришли в вершину и ее баланс стал равным -1 или 1, то высота этого поддерева не изменилась и подъем останавливается. Если баланс равен -2 или 2, выполняется вращение и, если после вращения баланс вершины стал равен нулю, то подъем останавливается, иначе продолжается.

## Результаты тестов.

Тесты проводились на случайно сгенерированных массивах целых чисел разного диапазона значений из 100000 элементов. Для каждого диапазона проводилось 100 тестирований, генерируя новый массив. Время замерялось встроенным в язык программирования C++ таймером, отсчитывающий количество тактов процессора с начала выполнения функции до ее окончания. Количества тактов суммировались для каждого дерева и выводилось среднее арифметическое по окончанию теста. Замерялось время создания и формирования деревьев из элементов массива.

По результатам тестов с увеличением диапазона элементов время на формирование BST дерева уменьшалось, RBT дерева увеличивалось, а AVL дерева сильно не менялось.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Диапазон элементов |  | BST | RBT | AVL |
|  |  |  |  |  |
| 100 |  | 3940 | 27 | 584 |
| 1000 |  | 495 | 44 | 609 |
| 10000 |  | 157 | 74 | 603 |
| 100000 |  | 146 | 119 | 609 |
| 1000000 |  | 141 | 115 | 590 |

# Заключение.

В результате выполненной работы были изучены Бинарное дерево поиска, Красно-черное дерево и АВЛ-дерево. В процессе была проделана работа с данными структурами данных: реализация структуры дерева, обход, добавление узла, удаление узла и поиск элементов. Во всех тестах Красно-черное дерево показывало лучшую эффективность, чем АВЛ-дерево и Бинарное дерево поиска, поэтому его рекомендуется использовать для работы со структурами данных.

# Список использованной литературы.

1. https://www.intuit.ru/studies/courses/648/504/lecture/11458

2. https://ru.wikipedia.org/wiki/Красно-чёрное\_дерево

3. https://ru.wikipedia.org/wiki/АВЛ-дерево

# Приложение.

## Справка о трудовом участии всех участников проекта

Гусев Виталий 70%.

Составление графика, поиск информации, реализация деревьев и тесты алгоритмов, написание отчета и создание презентации.

Ларичев Никита 20%.

Составление графика и поиск информации.

Шептур Ангелина 5%.

Составление графика.

Турдиев Мухамадовуд 5%.

Составление графика.

## Листинги программ

#include <iostream>

#include <string>

#include <string.h>

#include <cmath>

#include <fstream>

#include <stack>

#include <ctime>

#include <cstdlib>

#include <queue>

using namespace std;

const int N = 100000;

const int M = 100;

//Узел дерева можно описать как структуру :

struct BSTnode {

int key; // поле данных

struct BSTnode\* left; // левый потомок

struct BSTnode\* right; // правый потомок

};

//префиксный вид

void treeprintpref(BSTnode\* tree) {

if (tree != NULL) { //Пока не встретится пустой узел

cout << tree->key << " "; //Отображаем корень дерева

treeprintpref(tree->left); //Рекурсивная функция для левого поддерева

treeprintpref(tree->right); //Рекурсивная функция для правого поддерева

}

}

//инфиксный вид

void treeprintinf(BSTnode\* tree) {

if (tree != NULL) { //Пока не встретится пустой узел

treeprintinf(tree->left); //Рекурсивная функция для левого поддерева

cout << tree->key << " "; //Отображаем корень дерева

treeprintinf(tree->right); //Рекурсивная функция для правого поддерева

}

}

//постфиксный вид

void treeprintpost(BSTnode\* tree) {

if (tree != NULL) { //Пока не встретится пустой узел

treeprintpost(tree->left); //Рекурсивная функция для левого поддерева

treeprintpost(tree->right); //Рекурсивная функция для правого поддерева

cout << tree->key << " "; //Отображаем корень дерева

}

}

//Добавление узлов в дерево

struct BSTnode\* addnode(int x, BSTnode\* tree) {

if (tree == NULL) { // Если дерева нет, то формируем корень

tree = new BSTnode; // память под узел

tree->key = x; // поле данных

tree->left = NULL;

tree->right = NULL; // ветви инициализируем пустотой

}

else if (x < tree->key) // условие добавление левого потомка

tree->left = addnode(x, tree->left);

else // условие добавление правого потомка

tree->right = addnode(x, tree->right);

return(tree);

}

//Удаление поддерева

void DeleteTree(BSTnode\* tree) {

if (tree != NULL) {

DeleteTree(tree->left);

DeleteTree(tree->right);

delete tree;

}

}

BSTnode\* MinValue(BSTnode\* Tree)

{

if (Tree->left != NULL) {

return MinValue(Tree->left);

}

else {

cout << Tree->key;

}

}

BSTnode\* findmin(BSTnode\* node) // поиск узла с минимальным ключом в дереве p

{

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

}

//Так как в бинарном дереве поиска для каждого узла справедливо, что left < right,

//то соответственно для нахождения наибольшего элемента

//надо топать от корня по правым веткам до упора - там и будет наибольший.

BSTnode\* MaxValue(BSTnode\* Tree)

{

if (Tree->right != NULL) {

return MaxValue(Tree->right);

}

else

{

cout << Tree->key;

}

}

int NumberOfNodes(BSTnode\* Tree)

{

if (Tree == NULL) return 0;

return NumberOfNodes(Tree->left) + 1 + NumberOfNodes(Tree->right);

}

//Высота(максимальная глубина) дерева определяется количеством уровней,

//на которых располагаются его вершины.

//Высота пустого дерева равна нулю, высота дерева из одного корня – единице.

//На первом уровне дерева может быть только одна вершина – корень дерева,

//на втором – потомки корня дерева, на третьем – потомки потомков корня дерева и т.д.

int HeightBTree(BSTnode\* Tree)

{

int x = 0, y = 0;

if (Tree == NULL) return 0; //пустое дерево или дошли до листа

if (Tree->left) x = HeightBTree(Tree->left); //высота левого поддерева

if (Tree->right) y = HeightBTree(Tree->right); //высота правого поддерева

if (x > y) return x + 1; //+1 от корня к левому поддереву

else return y + 1; //+1 от корня к правому поддереву

}

//поиск элемента в бинарном дереве поиска

BSTnode\* Search(BSTnode\* Tree, int x)

{

if (Tree == NULL) return NULL;

if (Tree->key == x) return Tree;

if (x < Tree->key) return Search(Tree->left, x);

else

return Search(Tree->right, x);

}

BSTnode\* DeleteNode(BSTnode\* node, int x) {

if (node == NULL)

return node;

if (x == node->key) {

BSTnode\* help1;

if (node->right == NULL)

help1 = node->left;

else {

BSTnode\* help2 = node->right;

if (help2->left == NULL) {

help2->left = node->left;

help1 = help2;

}

else {

BSTnode\* pmin = help2->left;

while (pmin->left != NULL) {

help2 = pmin;

pmin = help2->left;

}

help2->left = pmin->right;

pmin->left = node->left;

pmin->right = node->right;

help1 = pmin;

}

}

delete node;

return help1;

}

else if (x < node->key)

node->left = DeleteNode(node->left, x);

else

node->right = DeleteNode(node->right, x);

return node;

}

//=================================================================================

class RBtree {

struct node\_st { node\_st\* p1, \* p2; int value; bool red; }; // структура узла

node\_st\* tree\_root; //!< корень

int nodes\_count; //!< число узлов дерева

private:

node\_st\* NewNode(int value); //!< выделение новой вешины

void DelNode(node\_st\*); //!< удаление вершины

void Clear(node\_st\*); //!< снос дерева (рекурсивная часть)

node\_st\* Rotate21(node\_st\*); //!< вращение влево

node\_st\* Rotate12(node\_st\*); //!< вращение вправо

void BalanceInsert(node\_st\*\*); //!< балансировка вставки

bool BalanceRemove1(node\_st\*\*); //!< левая балансировка удаления

bool BalanceRemove2(node\_st\*\*); //!< правая балансировка удаления

bool Insert(int, node\_st\*\*); //!< рекурсивная часть вставки

bool GetMin(node\_st\*\*, node\_st\*\*); //!< найти и убрать максимальный узел поддерева

bool Remove(node\_st\*\*, int); //!< рекурсивная часть удаления

public: // отладочная часть

enum check\_code { error\_balance, error\_struct, ok }; // код ошибки

void Show(); //!< вывод дерева

check\_code Check(); //!< проверка дерева

bool TreeWalk(bool\*, int); //!< обход дерева и сверка значений с массивом

private: // отладочная часть

void Show(node\_st\*, int, char); //!< вывод дерева, рекурсивная часть

check\_code Check(node\_st\*, int, int&);//!< проверка дерева (рекурсивная часть)

bool TreeWalk(node\_st\*, bool\*, int); //!< обход дерева и сверка значений с массивом (рекурсивная часть)

public:

RBtree();

~RBtree();

void Clear(); //!< снести дерево

bool Find(int); //!< найти значение

void Insert(int); //!< вставить значение

void Remove(int); //!< удалить значение

int GetNodesCount(); //!< узнать число узлов

};

RBtree::RBtree()

{

tree\_root = 0;

nodes\_count = 0;

}

RBtree::~RBtree()

{

Clear(tree\_root);

}

int RBtree::GetNodesCount()

{

return nodes\_count;

}

// выделение новой вешины

RBtree::node\_st\* RBtree::NewNode(int value)

{

nodes\_count++;

node\_st\* node = new node\_st;

node->value = value;

node->p1 = node->p2 = 0;

node->red = true;

return node;

}

// удаление вершины

void RBtree::DelNode(node\_st\* node)

{

nodes\_count--;

delete node;

}

// снос дерева (рекурсивная часть)

void RBtree::Clear(node\_st\* node)

{

if (!node) return;

Clear(node->p1);

Clear(node->p2);

DelNode(node);

}

// вывод дерева, рекурсивная часть

//! \param node узел

//! \param depth глубина

//! \param dir значёк

//! \code Show(root,0,'\*'); \endcode

void RBtree::Show(node\_st\* node, int depth, char dir)

{

int n;

if (!node) return;

for (n = 0; n < depth; n++) putchar(' ');

printf("%c[%d] (%s)\n", dir, node->value, node->red ? "red" : "black");

Show(node->p1, depth + 2, '-');

Show(node->p2, depth + 2, '+');

}

// вращение влево

//! \param index индеск вершины

//! \result новая вершина дерева

RBtree::node\_st\* RBtree::Rotate21(node\_st\* node)

{

node\_st\* p2 = node->p2;

node\_st\* p21 = p2->p1;

p2->p1 = node;

node->p2 = p21;

return p2;

}

// вращение вправо

//! \param index индеск вершины

//! \result новая вершина дерева

RBtree::node\_st\* RBtree::Rotate12(node\_st\* node)

{

node\_st\* p1 = node->p1;

node\_st\* p12 = p1->p2;

p1->p2 = node;

node->p1 = p12;

return p1;

}

// балансировка вершины

void RBtree::BalanceInsert(node\_st\*\* root)

{

node\_st\* p1, \* p2, \* px1, \* px2;

node\_st\* node = \*root;

if (node->red) return;

p1 = node->p1;

p2 = node->p2;

if (p1 && p1->red) {

px2 = p1->p2; // задача найти две рядом стоящие крастне вершины

if (px2 && px2->red) p1 = node->p1 = Rotate21(p1);

px1 = p1->p1;

if (px1 && px1->red) {

node->red = true;

p1->red = false;

if (p2 && p2->red) { // отделаемся перекраской вершин

px1->red = true;

p2->red = false;

return;

}

\*root = Rotate12(node);

return;

}

}

// тоже самое в другую сторону

if (p2 && p2->red) {

px1 = p2->p1; // задача найти две рядом стоящие крастне вершины

if (px1 && px1->red) p2 = node->p2 = Rotate12(p2);

px2 = p2->p2;

if (px2 && px2->red) {

node->red = true;

p2->red = false;

if (p1 && p1->red) { // отделаемся перекраской вершин

px2->red = true;

p1->red = false;

return;

}

\*root = Rotate21(node);

return;

}

}

}

bool RBtree::BalanceRemove1(node\_st\*\* root)

{

node\_st\* node = \*root;

node\_st\* p1 = node->p1;

node\_st\* p2 = node->p2;

if (p1 && p1->red) {

p1->red = false; return false;

}

if (p2 && p2->red) { // случай 1

node->red = true;

p2->red = false;

node = \*root = Rotate21(node);

if (BalanceRemove1(&node->p1)) node->p1->red = false;

return false;

}

unsigned int mask = 0;

node\_st\* p21 = p2->p1;

node\_st\* p22 = p2->p2;

if (p21 && p21->red) mask |= 1;

if (p22 && p22->red) mask |= 2;

switch (mask)

{

case 0: // случай 2 - if((!p21 || !p21->red) && (!p22 || !p22->red))

p2->red = true;

return true;

case 1:

case 3: // случай 3 - if(p21 && p21->red)

p2->red = true;

p21->red = false;

p2 = node->p2 = Rotate12(p2);

p22 = p2->p2;

case 2: // случай 4 - if(p22 && p22->red)

p2->red = node->red;

p22->red = node->red = false;

\*root = Rotate21(node);

}

return false;

}

bool RBtree::BalanceRemove2(node\_st\*\* root)

{

node\_st\* node = \*root;

node\_st\* p1 = node->p1;

node\_st\* p2 = node->p2;

if (p2 && p2->red) { p2->red = false; return false; }

if (p1 && p1->red) { // случай 1

node->red = true;

p1->red = false;

node = \*root = Rotate12(node);

if (BalanceRemove2(&node->p2)) node->p2->red = false;

return false;

}

unsigned int mask = 0;

node\_st\* p11 = p1->p1;

node\_st\* p12 = p1->p2;

if (p11 && p11->red) mask |= 1;

if (p12 && p12->red) mask |= 2;

switch (mask) {

case 0: // случай 2 - if((!p12 || !p12->red) && (!p11 || !p11->red))

p1->red = true;

return true;

case 2:

case 3: // случай 3 - if(p12 && p12->red)

p1->red = true;

p12->red = false;

p1 = node->p1 = Rotate21(p1);

p11 = p1->p1;

case 1: // случай 4 - if(p11 && p11->red)

p1->red = node->red;

p11->red = node->red = false;

\*root = Rotate12(node);

}

return false;

}

bool RBtree::Find(int value)

{

node\_st\* node = tree\_root;

while (node) {

if (node->value == value) return true;

node = node->value > value ? node->p1 : node->p2;

}

return false;

}

// рекурсивная часть вставки

//! \result true если изменений небыло или балансировка в данной вершине не нужна

bool RBtree::Insert(int value, node\_st\*\* root)

{

node\_st\* node = \*root;

if (!node) \*root = NewNode(value);

else {

if (value == node->value) return true;

if (Insert(value, value < node->value ? &node->p1 : &node->p2)) return true;

BalanceInsert(root);

}

return false;

}

// найти и убрать максимальный узел поддерева

//! \param root корень дерева в котором надо найти элемент

//! \retval res эелемент который был удалён

//! \result true если нужен баланс

bool RBtree::GetMin(node\_st\*\* root, node\_st\*\* res)

{

node\_st\* node = \*root;

if (node->p1) {

if (GetMin(&node->p1, res)) return BalanceRemove1(root);

}

else {

\*root = node->p2;

\*res = node;

return !node->red;

}

return false;

}

// рекурсивная часть удаления

//! \result true если нужен баланс

bool RBtree::Remove(node\_st\*\* root, int value)

{

node\_st\* t, \* node = \*root;

if (!node) return false;

if (node->value < value) {

if (Remove(&node->p2, value)) return BalanceRemove2(root);

}

else if (node->value > value) {

if (Remove(&node->p1, value)) return BalanceRemove1(root);

}

else {

bool res;

if (!node->p2) {

\*root = node->p1;

res = !node->red;

}

else {

res = GetMin(&node->p2, root);

t = \*root;

t->red = node->red;

t->p1 = node->p1;

t->p2 = node->p2;

if (res) res = BalanceRemove2(root);

}

DelNode(node);

return res;

}

return 0;

}

// вывод дерева

void RBtree::Show()

{

printf("[tree]\n");

Show(tree\_root, 0, '\*');

}

// функция вставки

void RBtree::Insert(int value)

{

Insert(value, &tree\_root);

if (tree\_root) tree\_root->red = false;

}

// удаление узла

void RBtree::Remove(int value)

{

Remove(&tree\_root, value);

}

// снос дерева

void RBtree::Clear()

{

Clear(tree\_root);

tree\_root = 0;

}

// проверка дерева (рекурсивная часть)

//! \param tree дерево

//! \param d текущая чёрная глубина

//! \param h эталонная чёрная глубина

//! \result 0 или код ошибки

RBtree::check\_code RBtree::Check(node\_st\* tree, int d, int& h)

{

if (!tree) {

// количество чёрных вершин на любом пути одинаковое

if (h < 0) h = d;

return h == d ? ok : error\_balance;

}

node\_st\* p1 = tree->p1;

node\_st\* p2 = tree->p2;

// красная вершина должна иметь чёрных потомков

if (tree->red && (p1 && p1->red || p2 && p2->red)) return error\_struct;

if (p1 && tree->value<p1->value || p2 && tree->value>p2->value) return error\_struct;

if (!tree->red) d++;

check\_code n = Check(p1, d, h); if (n) return n;

return Check(p2, d, h);

}

// проверка дерева

RBtree::check\_code RBtree::Check()

{

int d = 0;

int h = -1;

if (!tree\_root) return ok;

if (tree\_root->red) return error\_struct;

return Check(tree\_root, d, h);

}

// обход дерева и сверка значений с массивом (рекурсивная часть)

//! \param node корень дерева

//! \param array массив для сверки

//! \param size размер массива

bool RBtree::TreeWalk(node\_st\* node, bool\* array, int size)

{

if (!node) return false;

int value = node->value;

if (value < 0 || value >= size || !array[value]) return true;

array[value] = false;

return TreeWalk(node->p1, array, size) || TreeWalk(node->p2, array, size);

}

// обход дерева и сверка значений с массивом

//! \param array массив для сверки

//! \param size размер массива

bool RBtree::TreeWalk(bool\* array, int size)

{

if (TreeWalk(tree\_root, array, size)) return true;

for (int n = 0; n < size; n++) if (array[n]) return true;

return false;

}

//================================================================

struct nodeavl // структура для представления узлов дерева

{

int key;

unsigned char height;

nodeavl\* left;

nodeavl\* right;

nodeavl(int k) { key = k; left = right = 0; height = 1; }

};

unsigned char height(nodeavl\* p)

{

return p ? p->height : 0;

}

int bfactor(nodeavl\* p)

{

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

}

void fixheight(nodeavl\* p)

{

unsigned char hl = height(p->left);

unsigned char hr = height(p->right);

p->height = (hl > hr ? hl : hr) + 1;

}

nodeavl\* rotateright(nodeavl\* p) // правый поворот вокруг p

{

nodeavl\* q = p->left;

p->left = q->right;

q->right = p;

fixheight(p);

fixheight(q);

return q;

}

nodeavl\* rotateleft(nodeavl\* q) // левый поворот вокруг q

{

nodeavl\* p = q->right;

q->right = p->left;

p->left = q;

fixheight(q);

fixheight(p);

return p;

}

nodeavl\* balance(nodeavl\* p) // балансировка узла p

{

fixheight(p);

if (bfactor(p) == 2)

{

if (bfactor(p->right) < 0)

p->right = rotateright(p->right);

return rotateleft(p);

}

if (bfactor(p) == -2)

{

if (bfactor(p->left) > 0)

p->left = rotateleft(p->left);

return rotateright(p);

}

return p; // балансировка не нужна

}

nodeavl\* insert(nodeavl\* p, int k) // вставка ключа k в дерево с корнем p

{

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

if (k < p->key)

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

else

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

return balance(p);

}

nodeavl\* findmin(nodeavl\* p) // поиск узла с минимальным ключом в дереве p

{

return p->left ? findmin(p->left) : p;

}

nodeavl\* removemin(nodeavl\* p) // удаление узла с минимальным ключом из дерева p

{

if (p->left == 0)

return p->right;

p->left = removemin(p->left);

return balance(p);

}

nodeavl\* remove(nodeavl\* p, int k) // удаление ключа k из дерева p

{

if (!p) return 0;

if (k < p->key)

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

else if (k > p->key)

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

else // k == p->key

{

nodeavl\* q = p->left;

nodeavl\* r = p->right;

delete p;

if (!r) return q;

nodeavl\* min = findmin(r);

min->right = removemin(r);

min->left = q;

return balance(min);

}

return balance(p);

}

void treeprint(nodeavl\* p) {

if (p != NULL) { //Пока не встретится пустой узел

cout << p->key << " "; //Отображаем корень дерева

treeprint(p->left); //Рекурсивная функция для левого поддерева

treeprint(p->right); //Рекурсивная функция для правого поддерева

}

}

int HeightBTree(nodeavl\* Tree)

{

int x = 0, y = 0;

if (Tree == NULL) return 0; //пустое дерево или дошли до листа

if (Tree->left) x = HeightBTree(Tree->left); //высота левого поддерева

if (Tree->right) y = HeightBTree(Tree->right); //высота правого поддерева

if (x > y) return x + 1; //+1 от корня к левому поддереву

else return y + 1; //+1 от корня к правому поддереву

}

unsigned int testbst(int array[])

{

unsigned int start\_time = clock();

struct BSTnode\* root = NULL;

for (int i = 0; i < N; i++)

{

root = addnode(array[i], root);

}

unsigned int end\_time = clock(); // конечное время

unsigned int search\_time = end\_time - start\_time;

return search\_time;

}

unsigned int testrbt(int array[])

{

unsigned int start\_time = clock();

RBtree tree;

for (int i = 0; i < N; i++)

{

tree.Insert(array[i]);

}

unsigned int end\_time = clock(); // конечное время

unsigned int search\_time = end\_time - start\_time;

return search\_time;

}

unsigned int testavl(int array[])

{

unsigned int start\_time = clock();

struct nodeavl\* root = NULL;

for (int i = 0; i < N; i++)

root = insert(root, i);

unsigned int end\_time = clock(); // конечное время

unsigned int search\_time = end\_time - start\_time;

return search\_time;

}

int main()

{

setlocale(LC\_ALL, "Russian");

int array[N];

srand(time(0));

unsigned int timeBST = 0;

unsigned int timeRBT = 0;

unsigned int timeAVL = 0;

for (int i = 0; i < M; i++)

{

for (int i = 0; i < N; i++)

array[i] = -500000 + rand() % 1000000;

timeBST += testbst(array);

timeRBT += testrbt(array);

timeAVL += testavl(array);

}

cout << timeBST / M << endl;

cout << timeRBT / M << endl;

cout << timeAVL / M << endl;

}