**Задание на курсовую работу:**

1. Реализовать класс «Красно-чёрное дерево»

2.1 Реализовать необходимые для класса функции:

* 1. Конструкторы
  2. Деструкторы

2.2 Реализовать необходимые для класса методы:

1. Добавление элемента
2. Удаление элемента
3. Поиск элемента по ключу
4. Поиск минимального элемента
5. Поиск максимального элемента

3. Добавить итераторы для перечисления элементов дерева

4. Оптимизировать построенный код по:

1. Производительности
2. Потреблению памяти

**Оглавление:**

1. Введение.
2. Описание задачи и иллюстрация оценок по памяти и производительности.
   1. Описание класса «Красно-чёрный узел».
   2. Описание класса «Красно-чёрное дерево».
   3. Описание класса «Итератор»
   4. Оптимизация программы: по памяти и по производительности.
3. Результаты запуска кода.
4. Заключение.
5. Список использованной литературы.
6. Приложение.

**Введение.**

Что такое красно-чёрное дерево и зачем оно нужно? Красно-чёрное дерево – это двоичное дерево поиска, в котором у каждого узла дерева есть дополнительный бит, отвечающий за «цвет» («красный» или «чёрный») узла. Данный бит используется, чтобы дерево оставалось сбалансированным при вставке в него элемента или удаления из него какого-либо элемента. Когда идёт процесс добавления нового узла, то новое дерево перестраивается и «перекрашивается» для того, чтобы сохранить свои свойства:

1. Каждый узел («вершина») — либо красный, либо черный.
2. Каждый конечный элемент дерева («лист») — черный.
3. Если вершина красная, оба ее ребенка черные.
4. Все пути, идущие от корня к листьям, содержат одинаковое количество черных вершин.

Красно-чёрные деревья важны в современном мире, так как на их основе реализуются часть ассоциативных контейнеров: (в STL языка С++ контейнер map реализован с помощью данного дерева). Также, красно-чёрное дерево используется для организации сравнимых данных, таких как фрагменты текста или числа.

**Описание класса «Красно-чёрный узел».**

Класс должен иметь:

1. Значение – ключ. Это число, которое имеет узел и благодаря которому, выстраивается иерархия узлов в дереве.
2. Цвет – красный или черный. Для реализации цвета, я воспользовался встроенным в C++ типом данных - «перечислением» («enum»), так как он удобен для ассоциации численного значения со цветом (0 – Чёрный, 1- Красный) и память для перечисления выделяется только после объявления переменной.
3. Указатели на левого и правого ребенка и на родителя - (по умолчанию данные элементы узла ни на что не указывают и инициализируются как «nullptr»)

enum NodeColor { Black, Red };

class RBNode

{

public:

RBNode(int value,

NodeColor color = Black,

RBNode\* left = nullptr,

RBNode\* right = nullptr,

RBNode\* parent = nullptr) :

\_value(value),

\_color(color),

\_parent(parent),

\_left(nullptr),

\_right(nullptr)

{ }

public:

int \_value;

NodeColor \_color; //base black

RBNode\* \_parent,

\* \_left,

\* \_right;

};

Код 1. Тип перечисления «NodeColor» и класс Красно-чёрный узел.

**Описание класса Красно-чёрное дерево.**

Класс должен иметь:

1. Конструкторы:
   1. Конструктор по умолчанию.
   2. Конструктор инициализации.
2. Деструктор
3. Методы:
   1. Добавление элемента
   2. Удаление элемента
   3. Поиск элемента по ключу
   4. Поиск минимального элемента
   5. Поиск максимального элемента
   6. Инициализация указателей для класса итератор

Конструктор инициализации использует функцию добавления элемента в дерево, которая, использует функцию балансировки.

Вставка, удаление и поиск узла в красно-чёрном дереве происходят за O(log(n)).

Функция вставки работает следующим образом:

1. Получая ключ, мы выделяем память для нового узла и красим его в красный цвет, так как все новые элементы должны иметь красный цвет.
2. Для выбора места вставки мы идем от корня до самого нижнего указателя, пока его потомками не станут указатели в пустоту («nullptr»).
3. Вставляем в него новый элемент и делаем его листом (указатели на его потомков должны быть равны «nullptr»).
4. Проверяем балансировку.
   1. Если отец нового элемента чёрный, то ничего делать не нужно, так как свойства дерева не нарушаются.
   2. Если же он красный, то нужно рассмотреть 3 случая.
      1. Если он единственный элемент в дереве, то красим его в чёрный, так как он является корнем.
      2. Если у него есть красный «дядя», то перекрашиваем «отца» и «дядю» в чёрный, а «деда» - в красный.
      3. Если дядя чёрный, то нужно выполнить поворот, чтобы новый элемент стал левым потомком, а после этого закрашивать.

Код 2. Конструкторы класса Красно-чёрное дерево.

public:

RBTree() : root(nullptr) {} //конструктор по умолчанию

RBTree(std::vector<int>& v) { //инициализурующий конструктор

if (!v.empty()) //если массив не пустой

for (auto it : v) //последовательно инициализируем его числа

InsertNode(it);

}

void InsertNode(int value) {

RBNode\* t = new RBNode(value, Red); //выделяем память под новый элемент

RBNode\* p = root;

RBNode\* q = nullptr;

while (p) { // спускаемся вниз, пока не дойдем до листа

q = p;

if (t->\_value > p->\_value)

p = p->\_right;

else

p = p->\_left;

}

t->\_parent = q;

// добавляем новый элемент красного цвета

if (q) {

if (t->\_value > q->\_value)

q->\_right = t;

else

q->\_left = t;

}

else //в дереве не было элементов и новый элемент становится корнем

root = t;

t->\_color = Red;

fixInsertion(t); // проверяем, не нарушены ли свойства красно-черного дерева

}

Код 3. Функция добавления элемента.

void fixInsertion(RBNode\* node) {

RBNode\* parent = node->\_parent; //отец

while (node != RBTree::root && parent->\_color == Red){ //если есть красный отец

RBNode\* gparent = parent->\_parent; //дед

if (gparent->\_left == parent) // отец находится слева от деда

{

RBNode\* uncle = gparent->\_right; //дядя

if (uncle && uncle->\_color == Red) //если есть красный дядя //а цвет деда на красный

{ parent->\_color = Black;

uncle->\_color = Black;

gparent->\_color = Red;

node = gparent;

parent = node->\_parent;

}

else {

if (parent->\_right == node) //если выполнить перекрашивание выше, можно нарушить свойство дерева, поэтому

//если добавляемый элемент был правым потомком, то делаем левый поворот

{ TurnLeft(root, parent);

std::swap(node, parent);

}

TurnRight(root, gparent); //балансируем

gparent->\_color = Red;

parent->\_color = Black;

}

}

else { // отец находится справа от деда

RBNode\* uncle = gparent->\_left;

if (uncle && uncle->\_color == Red)

{ gparent->\_color = Red;

parent->\_color = Black;

uncle->\_color = Black;

node = gparent;

parent = node->\_parent;

} else {

if (parent->\_left == node){

TurnRight(root, parent);

std::swap(parent, node);

}

TurnLeft(root, gparent);

parent->\_color = Black;

gparent->\_color = Red;

}

}

}

root->\_color = Black; //корень всегда чёрный

}

Код 4. Функция балансировки дерева.

Деструктор класса построен рекурсивно при помощи вспомогательной функции «Destroy».

~RBTree() {

Destroy(root);

}

void Destroy(RBNode\* node) {

if (!node)

return void();

Destroy(node->\_left); //удаляем левое поддерево

Destroy(node->\_right);//удаляем правое поддерево

delete node;

node = nullptr;

}

Код 5. Деструктор и вспомогательная функция удаления.

Удаление элемента из красно-чёрного дерева может иметь несколько последствий (См. Приложение 1):

1. Если у вершины нет детей, то изменяем указатель на данный узел у родителя на nullptr.
2. Если у вершины один ребенок, то делаем у родителя ссылку на него вместо этой вершины.
3. Если у вершины два ребенка, то ищем вершину со следующим значением ключа, удаляем эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

После удаления элемента надо проверить балансировку дерева. Если мы удаляем красную вершину, то свойства дерева не нарушаются и не нужно выполнять восстановление; при удалении черного узла нужно восстанавливать порядок. Рассмотрим ребёнка удалённой вершины:

1. Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас вершина имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.
2. Если брат текущей вершины был чёрным, то получаем три случая:
   1. Оба ребёнка у брата чёрные. Красим брата в красный цвет и отца делаем черным.
   2. Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем поворот.
   3. Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение.
3. Продолжаем этот алгоритм, пока текущая вершина чёрная и мы не дошли до корня.

В функции поиска мы постоянно сравниваем ключи элементов с переданным в функцию значением.

RBNode\* Search(int value)

{

RBNode\* temp = root;

if (temp->\_value == value)

return temp;

while (temp)

{

if (temp->\_value > value)

temp = temp->\_left;

else if (temp->\_value < value)

temp = temp->\_right;

else

break;

}

return temp;

}

Код 6. Функция поиска элемента по ключу.

Функция поиска минимума и максимума. В зависимости от функции идём в левое дерево до конца или в правое.

RBNode\* Min()

{

RBNode\* temp = root;

if (!temp) return nullptr;

while (temp->\_left)

{

temp = temp->\_left;

}

return temp;

}

RBNode\* Max()

{

RBNode\* temp = root;

if (!temp) return nullptr;

while (temp->\_right)

{

temp = temp->\_right;

}

return temp;

}

Код 7. Функция поиска минимума и максимума.

Для работы итератора с красно-чёрным деревом – нужно в классе дерева объявить функции «begin()» и «end()», которые возвращают соответственно указатели на самый минимальный элемент дерева и на пустоту за последним элементом.

TreeIterator begin() { return TreeIterator(Min()); }

TreeIterator end() { return TreeIterator(); }

Код 8. Функции begin() и end().

**Описание класса «Итератор красно-чёрного дерева».**

Класс должен иметь:

1. Два перегруженных оператора («==» и «!=»).
2. Оператор разыменования.
3. Операторы декремента.
4. Операторы инкремент.

Оператор пре-инкремента и пре-декремента класса итератор работают следующим образом:

1. Если у узла есть наследник, то ищем максимальный/минимальный соответственно элемент этого поддерева и возвращаем указатель на него.
2. Если наследников нету, но есть родитель, то мы должны подняться на уровень вверх.
3. Если нет наследников и родителей, то наш элемент является корнем дерева.

class TreeIterator : public std::iterator<std::input\_iterator\_tag, RBNode> {

RBNode\* node;

public:

TreeIterator(RBNode\* node\_ = nullptr) : node(node\_) {}

TreeIterator(const TreeIterator& it) : node(it.node) {}

bool operator == (const TreeIterator& other) const { return node == other.node; }

bool operator != (const TreeIterator& other) const { return node != other.node; }

int& operator \* () { return node->\_value; }

const int& operator \* () const { return node->\_value; }

TreeIterator& operator++() {

if (node->\_right){

node = node->\_right;

while (node->\_left) {

node = node->\_left;

}

}else{

RBNode\* temp = node->\_parent;

while (temp && temp->\_right == node){

node = temp;

temp = node->\_parent;

}

// Если корневой узел дерева не имеет правого потомка и итератор начинается с корневого узла

if (node->\_right != temp)

node = temp;

}

return \*this;

}

TreeIterator operator++(int) {

TreeIterator old(node);

++\* this;

return old;}

TreeIterator& operator--() {

if (node->\_left){

node = node->\_left;

while (node->\_right){

node = node->\_right;

}

}else{

RBNode\* temp = node->\_parent;

while (temp && temp->\_left == node){

node = temp;

temp = node->\_parent;

}

node = temp;

}

return \*this;

}

TreeIterator operator--(int) {

TreeIterator old(node);

--\* this;

return old;}

};

Код 9. Класс красно-чёрный итератор.

Код 9. Класс Красно-чёрный итератор.

**Оптимизация программы по памяти и производительности.**

Оптимизация производилась с помощь встроенных в IDE MCVS 2022 Current инструментов. Вся компиляция программы производилась на Intel Core I7-7700hq (в зависимости от процессора, скорость на разных устройствах может варироваться).

Результаты различных настроек:

1. Максимальная оптимизация (приоритет скорости) (/O2), предпочитать скорость кода (/Ot) - время работы 8сек.
2. Максимальная оптимизация (приоритет скорости) (/O2), предпочитать краткость кода (/Os) – время работы 10сек.
3. Максимальная оптимизация (приоритет размера) (/O1), предпочитать краткость кода (/Os) – время работы 11сек.
4. Максимальная оптимизация (приоритет размера) (/O1), предпочитать скорость кода (/Ot) - время работы 10сек.
5. Оптимизация (приоритет скорости) (/Ox), предпочитать краткость кода (/Os) - время работы 9сек.
6. Оптимизация (приоритет скорости) (/Ox), предпочитать скорость кода (/Ot) - время работы 9сек.

**Результат запуска кода.**

int main()

{

unsigned int start\_time = clock();

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

std::vector<int> nums{ 10,50,30,60,80,70,20,40,90,100 };

RBTree tree1(nums); //инициализирующий конструктор

std::cout << "\nPrint\n"; //вывод при помощи итератора преинкремента

for (auto it = tree1.begin(); it != tree1.end(); ++it)

{

std::cout << \*it << ' ';

}

std::cout << '\n';

std::cout << '\n' << "Search(35)\t";

if (tree1.Search(35))

std::cout << std::endl << tree1.Search(35)->\_value << std::endl;

else

std::cout << "\nElement is not found\n";

std::cout << "\nSearch(100)\t";

std::cout << std::endl << tree1.Search(100)->\_value << std::endl;

std::cout << "\nRemove element 30\n";

tree1.Remove(30);

std::cout << "More intelligent print\n";

tree1.print();

std::cout << "Min = " << tree1.Min()->\_value << "\tMax = " << tree1.Max()->\_value << std::endl;

unsigned int end\_time = clock();

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

std::cout << "Working time is " << search\_time << std::endl;

return 0;

}

Код 10. Главная функция.

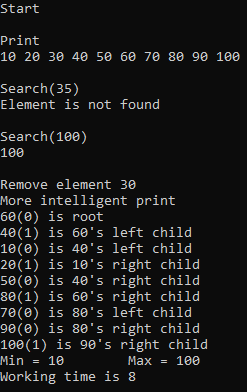


Рисунок 1. Результат запуска главной функции.

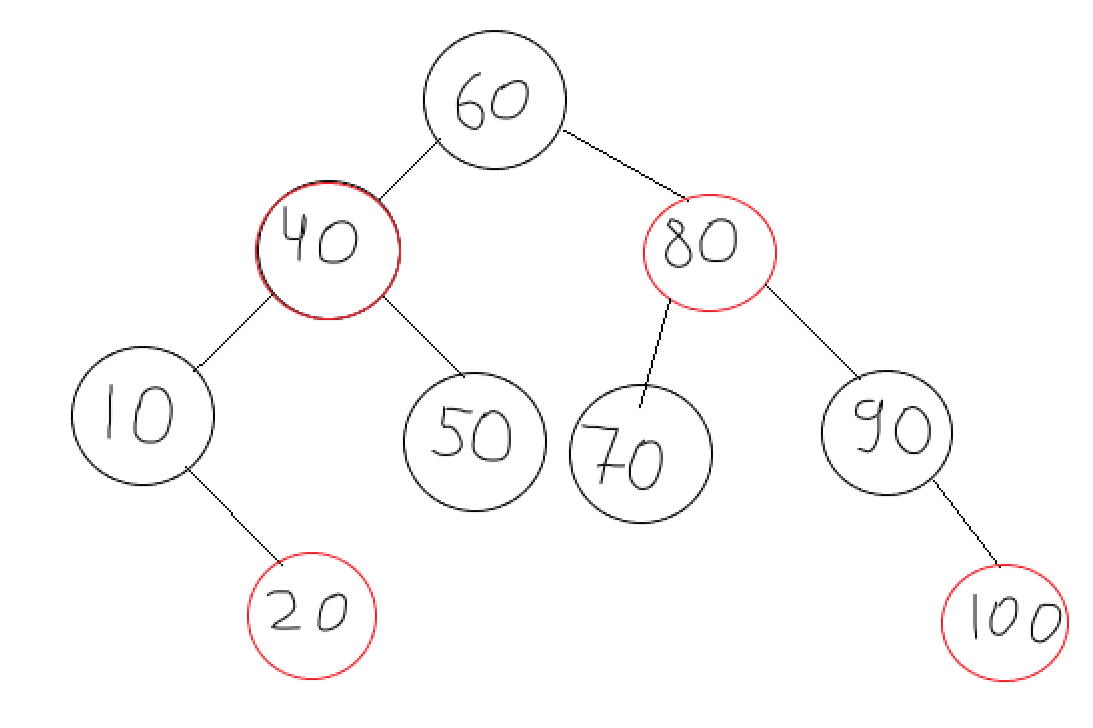


Рисунок 2. Визуальный вид более умного вывода дерева.

**Заключение.**

В данной работе, я создал и продемонстрировал работу структуру данных Красно-чёрное дерево, итератора по этому дереву. Лучше изучил внутреннюю структуру бинарных типов. Получил опыт в разработке собственных структур данных.

**Список использованной литературы.**

1. <https://neerc.ifmo.ru/wiki/index.php?title=%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D0%B5%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>
2. <https://habr.com/ru/post/330644/>
3. <https://habr.com/ru/post/265491/>

**Приложение 1.**

#include <iostream>

#include <vector>

class TreeIterator;

enum NodeColor { Black, Red };

class RBNode

{

public:

RBNode(int value,

NodeColor color,

RBNode\* left = nullptr,

RBNode\* right = nullptr,

RBNode\* parent = nullptr) :

\_value(value),

\_color(color),

\_parent(parent),

\_left(nullptr),

\_right(nullptr)

{ }

public:

int \_value;

NodeColor \_color; //base black

RBNode\* \_parent,

\* \_left,

\* \_right;

};

class TreeIterator : public std::iterator<std::input\_iterator\_tag, RBNode> {

RBNode\* node;

public:

TreeIterator(RBNode\* node\_ = nullptr) : node(node\_) {}

TreeIterator(const TreeIterator& it) : node(it.node) {}

bool operator == (const TreeIterator& other) const { return node == other.node; }

bool operator != (const TreeIterator& other) const { return node != other.node; }

int& operator \* () { return node->\_value; }

const int& operator \* () const { return node->\_value; }

TreeIterator& operator++() {

if (node->\_right)

{

node = node->\_right;

while (node->\_left)

{

node = node->\_left;

}

}

else

{

RBNode\* temp = node->\_parent;

while (temp && temp->\_right == node)

{

node = temp;

temp = node->\_parent;

}

// Если корневой узел дерева не имеет правого потомка и итератор начинается с корневого узла

if (node->\_right != temp)

node = temp;

}

return \*this;

}

TreeIterator operator++(int) {

TreeIterator old(node);

++\* this;

return old;

}

TreeIterator& operator--() {

if (node->\_left)

{

node = node->\_left;

while (node->\_right)

{

node = node->\_right;

}

}

else

{

RBNode\* temp = node->\_parent;

while (temp && temp->\_left == node)

{

node = temp;

temp = node->\_parent;

}

node = temp;

}

return \*this;

}

TreeIterator operator--(int) {

TreeIterator old(node);

--\* this;

return old;

}

};

class RBTree

{

public:

RBTree() : root(nullptr) {} //конструктор по умолчанию

RBTree(std::vector<int>& v) //инициализурующий конструктор

{

if (!v.empty()) //если массив не пустой

{

for (auto it : v) //последовательно инициализируем его числа

InsertNode(it);

}

}

void InsertNode(int value) {

RBNode\* t = new RBNode(value, Red); //выделяем память под новый элемент

RBNode\* p = root;

RBNode\* q = nullptr;

while (p) { // спускаемся вниз, пока не дойдем до листа

q = p;

if (t->\_value > p->\_value)

p = p->\_right;

else

p = p->\_left;

}

t->\_parent = q;

// добавляем новый элемент красного цвета

if (q) {

if (t->\_value > q->\_value)

q->\_right = t;

else

q->\_left = t;

}

else //в дереве не было элементов и новый элемент становится корнем

root = t;

t->\_color = Red;

fixInsertion(t); // проверяем, не нарушены ли свойства красно-черного дерева

}

void fixInsertion(RBNode\* node) {

RBNode\* parent = node->\_parent; //отец

while (node != RBTree::root && parent->\_color == Red) //если есть красный отец

{

RBNode\* gparent = parent->\_parent; //дед

if (gparent->\_left == parent) // отец находится слева от деда

{

RBNode\* uncle = gparent->\_right; //дядя

if (uncle && uncle->\_color == Red) //если есть красный дядя, то меняем цвет отца и дяди на черный,

//а цвет деда на красный

{

parent->\_color = Black;

uncle->\_color = Black;

gparent->\_color = Red;

node = gparent;

parent = node->\_parent;

}

else

{

if (parent->\_right == node) //если выполнить перекрашивание выше, можно нарушить свойство дерева, поэтому

//если добавляемый элемент был правым потомком, то делаем левый поворот

{

TurnLeft(root, parent);

std::swap(node, parent);

}

TurnRight(root, gparent); //балансируем

gparent->\_color = Red;

parent->\_color = Black;

}

}

else // отец находится справа от деда

{

RBNode\* uncle = gparent->\_left;

if (uncle && uncle->\_color == Red)

{

gparent->\_color = Red;

parent->\_color = Black;

uncle->\_color = Black;

node = gparent;

parent = node->\_parent;

}

else

{

if (parent->\_left == node)

{

TurnRight(root, parent);

std::swap(parent, node);

}

TurnLeft(root, gparent);

parent->\_color = Black;

gparent->\_color = Red;

}

}

}

root->\_color = Black; //корень всегда чёрный

}

~RBTree() {

Destroy(root);

}

void TurnLeft(RBNode\*& a, RBNode\* b) {

RBNode\* y = b->\_right;

b->\_right = y->\_left;

if (y->\_left)

y->\_left->\_parent = b;

y->\_parent = b->\_parent;

if (!b->\_parent)

a = y;

else {

if (b == b->\_parent->\_left)

b->\_parent->\_left = y;

else

b->\_parent->\_right = y;

}

y->\_left = b;

b->\_parent = y;

}

void TurnRight(RBNode\*& a, RBNode\* b) {

RBNode\* y = b->\_left;

b->\_left = y->\_right;

if (y->\_right)

y->\_right->\_parent = b;

y->\_parent = b->\_parent;

if (!b->\_parent)

a = y;

else {

if (b == b->\_parent->\_right)

b->\_parent->\_right = y;

else

b->\_parent->\_left = y;

}

y->\_right = b;

b->\_parent = y;

}

void Destroy(RBNode\* node) {

if (!node)

return void();

Destroy(node->\_left);

Destroy(node->\_right);

delete node;

node = nullptr;

}

void Remove(int value) {

RBNode\* deleteNode = Search(value);

if (!deleteNode) //такого узла нет, выходим из удаления

return void();

RBNode\* child

, \* parent;

NodeColor color;

if (deleteNode->\_left && deleteNode->\_right) // Левый и правый узлы удаленного узла не пусты (не конечные узлы)

{

RBNode\* replace = deleteNode;

// Найти узел-преемник (самый нижний левый узел правого поддерева текущего узла)

replace = deleteNode->\_right;

while (replace->\_left)

{

replace = replace->\_left;

}

// Случай, когда удаленный узел не является корневым узлом

if (deleteNode->\_parent)

{

if (deleteNode->\_parent->\_left == deleteNode)

deleteNode->\_parent->\_left = replace;

else

deleteNode->\_parent->\_right = replace;

}

// Ситуация с корневым узлом

else

root = replace;

// child - это правильный узел, который заменяет узел и является узлом, который требует последующей корректировки

// Поскольку замена является преемником, он не может иметь левого дочернего узла

// Аналогично, у узла-предшественника не может быть правого дочернего узла

child = replace->\_right;

parent = replace->\_parent;

color = replace->\_color;

// Удаленный узел является родительским узлом замещающего узла (repalce)

if (parent == deleteNode)

parent = replace;

else

{

// Существование дочернего узла

if (child)

child->\_parent = parent;

parent->\_left = child;

replace->\_right = deleteNode->\_right;

deleteNode->\_right->\_parent = replace;

}

replace->\_parent = deleteNode->\_parent;

replace->\_color = deleteNode->\_color;

replace->\_left = deleteNode->\_left;

deleteNode->\_left->\_parent = replace;

if (color == Black)

RemoveFix(child, parent);

delete deleteNode;

return;

}

// Когда в удаленном узле только левый (правый) узел пуст, найдите дочерний узел удаленного узла

if (deleteNode->\_left)

child = deleteNode->\_left;

else

child = deleteNode->\_right;

parent = deleteNode->\_parent;

color = deleteNode->\_color;

if (child)

{

child->\_parent = parent;

}

// Удаленный узел не является корневым узлом

if (parent)

{

if (deleteNode == parent->\_left)

parent->\_left = child;

else

parent->\_right = child;

}

// Удаленный узел является корневым узлом

else

RBTree::root = child;

if (color == Black)

{

RemoveFix(child, parent);

}

delete deleteNode;

}

void RemoveFix(RBNode\* node, RBNode\* parent ) {

if (root) {

RBNode\* othernode;

while ((!node) || node->\_color == Black && node != RBTree::root)

{

if (parent->\_left == node)

{

othernode = parent->\_right;

if (othernode->\_color == Red)

{

othernode->\_color = Black;

parent->\_color = Red;

TurnLeft(root, parent);

othernode = parent->\_right;

}

else

{

if (!(othernode->\_right) || othernode->\_right->\_color == Black)

{

othernode->\_left->\_color = Black;

othernode->\_color = Red;

TurnRight(root, othernode);

othernode = parent->\_right;

}

othernode->\_color = parent->\_color;

parent->\_color = Black;

othernode->\_right->\_color = Black;

TurnLeft(root, parent);

node = root;

break;

}

}

else

{

othernode = parent->\_left;

if (othernode->\_color == Red)

{

othernode->\_color = Black;

parent->\_color = Red;

TurnRight(root, parent);

othernode = parent->\_left;

}

if ((!othernode->\_left || othernode->\_left->\_color == Black) && (!othernode->\_right || othernode->\_right->\_color == Black))

{

othernode->\_color = Red;

node = parent;

parent = node->\_parent;

}

else

{

if (!(othernode->\_left) || othernode->\_left->\_color == Black)

{

othernode->\_right->\_color = Black;

othernode->\_color = Red;

TurnLeft(root, othernode);

othernode = parent->\_left;

}

othernode->\_color = parent->\_color;

parent->\_color = Black;

othernode->\_left->\_color = Black;

TurnRight(root, parent);

node = root;

break;

}

}

}

if (node)

node->\_color = Black;

}

else

return void();

}

void print() {

if (!root)

std::cout << "empty RBtree\n";

else

print(root);

}

void print(RBNode\* node) const {

if (!node)

return void();

if (!node->\_parent)

std::cout << node->\_value << "(" << node->\_color << ") is root" << std::endl;

else if (node->\_parent->\_left == node)

{

std::cout << node->\_value << "(" << node->\_color << ") is " << node->\_parent->\_value << "'s " << "left child" << std::endl;

}

else

{

std::cout << node->\_value << "(" << node->\_color << ") is " << node->\_parent->\_value << "'s " << "right child" << std::endl;

}

print(node->\_left);

print(node->\_right);

}

RBNode\* Search(int value)

{

RBNode\* temp = root;

if (temp->\_value == value)

return temp;

while (temp)

{

if (temp->\_value > value)

temp = temp->\_left;

else if (temp->\_value < value)

temp = temp->\_right;

else

break;

}

return temp;

}

RBNode\* Min()

{

RBNode\* temp = root;

if (!temp) return nullptr;

while (temp->\_left)

{

temp = temp->\_left;

}

return temp;

}

RBNode\* Max()

{

RBNode\* temp = root;

if (!temp) return nullptr;

while (temp->\_right)

{

temp = temp->\_right;

}

return temp;

}

TreeIterator begin() { return TreeIterator(Min()); }

TreeIterator end() { return TreeIterator(); }

private:

RBNode\* root;

friend TreeIterator;

};