**Красно-Черное Дерево**

**Красно-чёрное дерево** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Red-black tree*, *RB-Tree*) — один из видов самобалансирующихся [двоичных деревьев поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0), гарантирующих [логарифмический](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B0%D1%80%D0%B8%D1%84%D0%BC) рост [высоты дерева](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C_%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%BE%D0%B2_%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D0%B8_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2#%D0%92) от числа [узлов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C_%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%BE%D0%B2_%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D0%B8_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2#%D0%A3) и позволяющее быстро выполнять [основные операции](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)#%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8_%D0%B2_%D0%B4%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%BC_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%B5_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0) дерева поиска: добавление, удаление и поиск узла. [Сбалансированность](https://ru.wikipedia.org/wiki/%D0%90%D0%92%D0%9B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

Красно-чёрное дерево используется для организации сравнимых данных, таких как фрагменты текста или числа. [Листовые узлы](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)#%D0%9B%D0%B8%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B5_%D1%83%D0%B7%D0%BB%D1%8B) красно-чёрных деревьев не содержат данных, благодаря чему не требуют выделения памяти — достаточно записать в узле-предке в качестве указателя на потомка нулевой указатель. Однако, в некоторых реализациях для упрощения алгоритма могут использоваться явные листовые узлы.

Красно-чёрное дерево — [двоичное дерево поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0), в котором каждый узел имеет атрибут *цвета*. При этом:

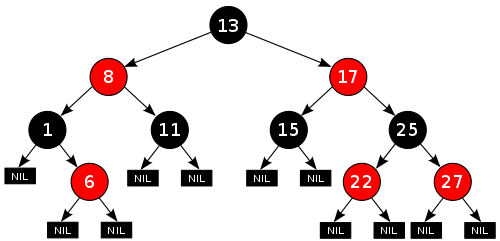
1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья — чёрные и не содержат данных.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Чтобы понять, как это работает, достаточно рассмотреть эффект свойств 4 и 5 вместе. Пусть для красно-чёрного дерева T число чёрных узлов от корня до листа равно B. Тогда кратчайший возможный путь до любого листа содержит B узлов и все они чёрные. Более длинный возможный путь может быть построен путём включения красных узлов. Однако, благодаря п.4 в дереве не может быть двух красных узлов подряд, а согласно пп. 2 и 3, путь начинается и кончается чёрным узлом. Поэтому самый длинный возможный путь состоит из 2B-1 узлов, попеременно красных и чёрных.

Если разрешить не листовому узлу иметь меньше двух потомков, а листовым — содержать данные, дерево сохраняет основные свойства, но алгоритмы работы с ним усложнятся. Поэтому в статье рассматриваются только «фиктивные листовые узлы», которые не содержат данных и просто служат для указания, где дерево заканчивается. Эти узлы могут быть опущены в некоторых иллюстрациях. Из п.5, также следует, что потомками красного узла могут быть либо два чёрных промежуточных узла, либо два чёрных листа, а с учётом п.3 и 4 — что если у чёрного узла один из потомков — листовой узел, то вторым должен быть либо тоже листовой, либо вышеописанная конструкция из одного красного и двух листовых.

Также в литературе встречается трактовка, в которой в красный/чёрный цвета раскрашивают не сами узлы, а ведущие к ним рёбра — но это не имеет большого значения для понимания принципа его работы.



**Анализ Красно-Черного дерева.**

Обходы

void preOrder(Tree\* node){ // Прямой

if (node != NULL){

cout << node->inf << " ";

preOrder(node->left);

preOrder(node->right);

}

}

void inOrder(Tree\* node){ // Симметричный

if (node != NULL){

inOrder(node->left);

cout << node->inf << " ";

inOrder(node->right);

}

}

void postOrder(Tree\* node){ // Обратный

if (node != NULL){

postOrder(node->left);

postOrder(node->right);

cout << node->inf << " ";

}

}

Симметричный обход используется когда необходимо обойти дерево в порядке, соответствующем значениям узлов. Обходим их от наименьшего до наибольшего. То есть от левой части к правой через корень.

Сложность: O(n)

Порядок обхода: 1, 2, 3, 4, 5, 6, 7, 8

При прямом обходе алгоритм получает значение текущего узла перед тем, как перейти сначала в левую часть, а затем в правую. Начиная от корня, сначала получаем значения корня, дальше таким же образом обходятся левый ребенок и его дети, затем правый ребенок и все его дети. Прямой обход обычно применяется для копирования дерева с сохранением его структуры.

Сложность: O(n)

Порядок обхода: 4, 2, 1, 3, 5, 7, 6, 8

При обратном обходе посещаем левое поддерево, правое поддерево, а потом, после обхода всех детей, переходим к самому узлу. Обратный обход часто используется для полного удаления дерева, так как в некоторых языках программирования необходимо убирать из памяти все узлы явно или для удаления поддерева. Поскольку корень в данном случае обрабатывается последним, таким образом уменьшаем работу, необходимую для удаления узлов.

Сложность: O(n)

Порядок обхода: 1, 3, 2, 6, 8, 7, 5, 4

В красно-черном дереве с черной высотой hb количество внутренних вершин не менее 2hb−1−1. Докажем по индукции по обычной высоте h(x), что поддерево любого узла xx с черной высотой hb(x) содержит не менее 2hb(x)−1−1 внутренних узлов. Здесь h(x) — кратчайшее расстояние от вершины xx до какого-то из листьев. Если высота узла xx равна 11, то xx — это лист, hb(x)=1, 21−1−1=0. Так как любая внутренняя вершина (вершина, у которой высота положительна) имеет двух потомков, то применим предположение индукции к ним — их высоты на единицу меньше высоты x. Тогда черные высоты детей могут быть hb(x) или hb(x)−1 — если потомок красный или черный соответственно.

Тогда по предположению индукции в каждом из поддеревьев не менее 2hb(x)−2−1 вершин. Тогда всего в поддереве не менее 2⋅(2hb(x)−2−1)+1=2hb(x)−1−1вершин (+1 — мы учли еще саму вершину x).

Переход доказан. Теперь, если мы рассмотрим корень всего дерева в качестве x, то получится, что всего вершин в дереве не менее 2hb−1−1 .

Следовательно, утверждение верно и для всего дерева. Красно-чёрное дерево с N ключами имеет высоту h=O(logN)

|  |
| --- |
|  |
| Рассмотрим красно-чёрное дерево с высотой h. Так как у красной вершины чёрные дети количество красных вершин не больше h2. Тогда чёрных вершин не меньше, чем h2−1.  По доказанной лемме, для количества внутренних вершин в дереве N выполняется неравенство:  N⩾2h/2−1  Прологарифмировав неравенство, имеем:  log(N+1)⩾h/2  2log(N+1)⩾h  h⩽2log(N+1)  Высота дерева, как сказано выше, есть величина h <= 2\*log2(n+1). Поэтому добавление будет осуществляться за O(h) или же за O(log(n)). Операции, которые выполняются за константное время нас не интересуют. |
|  |

void Push(int X) {

Tree\* tree = new Tree;

tree->parent = NULL;

tree->i = X;

tree->left = null;

tree->right = null;

tree->color = true;

Tree\* y = NULL;

Tree\* x = root;

while (x != null) {

y = x;

if (tree->i < x->i)

x = x->left;

else

x = x->right;

}

tree->parent = y;

if (y == NULL)

root = tree;

else if (tree->i < y->i)

y->left = tree;

else

y->right = tree;

if (tree->parent == NULL) {

tree->color = false;

return;

}

if (tree->parent->parent == NULL)

return;

Insert(tree);

}

Аналогично и для операции удаления. Операции внутри цикла while выполняются за константу, поэтому учитывать их не будем. Сам цикл идет в глубь дерева, поэтому, как сказано выше, будет пройдено h <= 2\*log2(n+1). Где h-высота дерева. Поэтому общее время выполнения будет составлять O(h) или же O(log(n)).

void Delete(int x) {

DelNode(root, x);

}

void DelNode(Tree\* tree, int x) {

Tree\* z = null;

Tree\* X;

while (tree != null) {

if (tree->i == x)

z = tree;

if (tree->i <= x)

tree = tree->right;

else

tree = tree->left;

}

if (z == null) {

cout << "\n\nТакой элемент не найден" << endl;

return;

}

Tree\* y;

y = z;

int c = y->color;

if (z->left == null) {

X = z->right;

Trans(z, z->right);

}

else if (z->right == null) {

X = z->left;

Trans(z, z->left);

}

else {

y = Last(z->right);

c = y->color;

X = y->right;

if (y->parent == z)

X->parent = y;

else {

Trans(y, y->right);

y->right = z->right;

y->right->parent = y;

}

Trans(z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

delete z;

if (c == false)

AutoDel(X);

}