Красно-черное дерево

**Красно-чёрное дерево** — один из видов самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

**Историческая справка**

Изобретателем красно-чёрного дерева считают немца Рудольфа Байера. Название «красно-чёрное дерево» структура данных получила в статье Л. Гимбаса и Р. Седжвика (1978). По словам Гимбаса, они использовали ручки двух цветов. По словам Седжвика, красный цвет лучше всех смотрелся на лазерном принтере.

**Принцип работы**

Красно-чёрное дерево — двоичное дерево поиска, 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 чёрного на красный;
3. Все листья, не содержащие данных — чёрные.
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 в худшем случае быть более эффективными, чем обычные двоичные деревья поиска.

[**Вставка**](https://www.geeksforgeeks.org/insertion-in-red-black-tree)

Вставка нового узла в красно-черное дерево включает в себя двухэтапный процесс: выполнение стандартной вставки двоичного дерева поиска (BST) с последующим исправлением любых нарушений свойств красно-черного дерева.

### Шаги вставки

1. **Вставка BST** : вставка нового узла, как в стандартном BST.
2. **Устранение нарушений** :
   * Если родительский элемент нового узла **черный** , то никакие свойства не нарушаются.
   * Если родительский элемент **красный** , дерево может нарушить свойство Red, что потребует исправлений.

### Исправление нарушений во время вставки

После вставки нового узла как **красного** узла мы можем столкнуться с несколькими случаями в зависимости от цветов родителя и дяди узла (брата родителя):

* **Случай 1: Дядя красный** : Перекрасьте родителя и дядю в **черный цвет** , а дедушку и бабушку в **красный цвет** . Затем поднимитесь по дереву, чтобы проверить наличие дальнейших нарушений.
* **Случай 2: Дядя — черный** :
  + **Подслучай 2.1: Узел является правым дочерним узлом** : выполнить левый поворот родительского узла.
  + **Подслучай 2.2: Узел является левым потомком** : выполнить поворот вправо на прародителе и перекрасить соответствующим образом.

**Устранение нарушений после вставки**

Когда вставляется новый узел, он всегда окрашивается в красный цвет. Это может привести к нарушениям свойств красно-черного дерева, в частности:

* Корень должен быть черным .
* Красные узлы не могут иметь красных дочерних узлов.

Анализ случая фиксации вставок:

* Случай 1: Перекрашивание и распространение вверх
  + Если родитель и дядя нового узла оба красные , перекрасить родителя и дядю в черный , а деда в красный . Затем рекурсивно применить исправление к деду.
* Случай 2: Вращение и перекрашивание
  + Если дядя нового узла черный , а новый узел является правым потомком левого потомка (или наоборот), выполните поворот, чтобы переместить новый узел вверх и выровнять его.
  + Если дядя нового узла черный , а новый узел является левым потомком левого потомка (или правым правым), выполните поворот и перекрасьте родителя и прародителя, чтобы исправить нарушение.



## [Удаление](https://www.geeksforgeeks.org/deletion-in-red-black-tree)

Удаление узла из красно-черного дерева также включает двухэтапный процесс: выполнение удаления BST с последующим исправлением любых возникающих нарушений.

### Шаги удаления

1. **Удаление BST** : удаление узла с использованием стандартных правил BST.
2. **Исправление двойного черного** :
   * При удалении черного узла может возникнуть состояние «двойного черного», требующее определенных исправлений.

### Исправление нарушений при удалении

При удалении черного узла мы решаем проблему двойного черного на основе цвета родственного узла и цветов его дочерних узлов:

* **Случай 1: Родной элемент красный** : поверните родительский элемент и перекрасьте родного и родного элемента.
* **Случай 2: Брат или сестра — черный** :
  + **Подслучай 2.1: дети брата или сестры черные** : перекрашиваем брата или сестру и распространяем двойной черный цвет вверх.
  + **Подслучай 2.2: По крайней мере один из детей брата или сестры красный** :
    - **Если дальний потомок родственного элемента красный** : выполните поворот родительского элемента и родственного элемента и перекрасьте его соответствующим образом.
    - **Если ближайший дочерний элемент родственного элемента красный** : поверните родственный элемент и его дочерний элемент, затем выполните действия, описанные выше.

**Устранение нарушений после удаления**

После удаления может потребоваться исправление дерева для восстановления свойств:

* При удалении черного узла или замене красного узла черным может возникнуть ситуация двойного черного.

Анализ случая исправления удалений:

* Случай 1: Брат/Сестра — Красный
  + Перекрасьте родной и родительский элементы и выполните поворот.
* Случай 2: брат или сестра — черные, у них есть черные дети
  + Перекрасьте родственного элемента в красный цвет и перенесите проблему на уровень родителя.
* Случай 3: брат или сестра — черные, у которого есть как минимум один ребенок красного цвета.
  + Поверните и перекрасьте, чтобы устранить проблему двойного черного цвета.

****

**Поиск**

Поиск узла в красно-черном дереве аналогичен поиску в стандартном двоичном дереве поиска (BST) . Операция поиска следует прямолинейному пути от корня к листу , сравнивая целевое значение со значением текущего узла и перемещаясь влево или вправо соответственно.

Шаги поиска

1. Начать с корня : начать поиск с корневого узла.
2. Пересечь дерево :
   * Если целевое значение равно значению текущего узла, узел найден.
   * Если целевое значение меньше значения текущего узла, переходим к левому дочернему узлу.
   * Если целевое значение больше значения текущего узла, переходим к правому дочернему узлу.
3. Повторить : продолжать этот процесс до тех пор, пока не будет найдено целевое значение или не будет достигнут узел NIL (указывающий на то, что значение отсутствует в дереве).

****

**Сложность основных операции:**

* Поиск – O(log n)
* Вставка - O(log n)
* Удаление – O(log n)

**Доказательство:**

Высота красно-черного дерева не превышает 2 \* log(n + 1). Это следует из того, что при фиксированной черной высоте, минимальное количество узлов в дереве экспоненциально растет относительно черной высоте. Таким образом, высота дерева ограничена O(log n), и все операции зависящие от глубины, имеют логарифмическую сложность.

## Преимущества красно-черных деревьев:

* **Сбалансированный:** Красно-черные деревья являются самобалансирующимися, то есть они автоматически поддерживают баланс между высотами левого и правого поддеревьев. Это гарантирует, что операции поиска, вставки и удаления займут O(log n) времени в худшем случае.
* **Эффективный поиск, вставка и удаление:** Благодаря сбалансированной структуре, красно-черные деревья предлагают эффективные операции. Поиск, вставка и удаление занимают O(log n) времени в худшем случае.
* **Простота реализации:** правила поддержания свойств красно-черного дерева относительно просты и понятны для реализации.
* **Широкое применение:** красно-черные деревья являются популярным выбором для реализации различных структур данных, таких как карты, наборы и очереди с приоритетами.

## Недостатки красно-черных деревьев:

* **Более сложные, чем другие сбалансированные деревья:** по сравнению с более простыми сбалансированными деревьями, такими как деревья AVL, красно-черные деревья имеют более сложные правила вставки и удаления.
* **Постоянные накладные расходы:** поддержание свойств красно-черного дерева добавляет небольшие накладные расходы к каждой операции вставки и удаления.
* **Не оптимальны для всех вариантов использования:** хотя красно-черные деревья эффективны для большинства операций, они могут оказаться не лучшим выбором для приложений, где требуются частые вставки и удаления, поскольку постоянные накладные расходы могут стать значительными.

**Литература**

* <https://www.geeksforgeeks.org/dsa/introduction-to-red-black-tree/#properties-of-redblack-trees>
* <https://javarush.com/groups/posts/4165-krasno-chjernoe-derevo-svoystva-principih-organizacii-mekhanizm-vstavki>
* <https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>

**Код программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Красно\_черное\_дерево

{

public class RedBlackTree

{

private RBTreeNode root;

private readonly RBTreeNode nullNode;

public RedBlackTree()

{

nullNode = new RBTreeNode(null) { Color = Color.Black };

}

public void Insert(string value)

{

var node = new RBTreeNode(value);

node.Left = nullNode;

node.Right = nullNode;

if (root == null)

{

root = node;

root.Color = Color.Black;

return;

}

RBTreeNode current = root;

RBTreeNode parent = null;

while (current != nullNode)

{

parent = current;

if (string.Compare(node.Value, current.Value) < 0)

current = current.Left;

else

current = current.Right;

}

node.Parent = parent;

if (string.Compare(node.Value, parent.Value) < 0)

parent.Left = node;

else

parent.Right = node;

FixInsert(node);

}

private void FixInsert(RBTreeNode node)

{

while (node != root && node.Parent.Color == Color.Red)

{

if (node.Parent == node.Parent.Parent.Left)

{

var uncle = node.Parent.Parent.Right;

if (uncle.Color == Color.Red)

{

node.Parent.Color = Color.Black;

uncle.Color = Color.Black;

node.Parent.Parent.Color = Color.Red;

node = node.Parent.Parent;

}

else

{

if (node == node.Parent.Right)

{

node = node.Parent;

RotateLeft(node);

}

node.Parent.Color = Color.Black;

node.Parent.Parent.Color = Color.Red;

RotateRight(node.Parent.Parent);

}

}

else

{

var uncle = node.Parent.Parent.Left;

if (uncle.Color == Color.Red)

{

node.Parent.Color = Color.Black;

uncle.Color = Color.Black;

node.Parent.Parent.Color = Color.Red;

node = node.Parent.Parent;

}

else

{

if (node == node.Parent.Left)

{

node = node.Parent;

RotateRight(node);

}

node.Parent.Color = Color.Black;

node.Parent.Parent.Color = Color.Red;

RotateLeft(node.Parent.Parent);

}

}

}

root.Color = Color.Black;

}

public bool Search(string value)

{

var current = root;

while (current != nullNode)

{

int cmp = string.Compare(value, current.Value);

if (cmp == 0)

return true;

else if (cmp < 0)

current = current.Left;

else

current = current.Right;

}

return false;

}

public void Delete(string value)

{

RBTreeNode node = root;

while (node != nullNode && string.Compare(node.Value, value) != 0)

{

if (string.Compare(value, node.Value) < 0)

node = node.Left;

else

node = node.Right;

}

if (node == nullNode)

return;

RBTreeNode x;

Color originalColor = node.Color;

RBTreeNode y = node;

if (node.Left == nullNode)

{

x = node.Right;

Transplant(node, node.Right);

}

else if (node.Right == nullNode)

{

x = node.Left;

Transplant(node, node.Left);

}

else

{

y = Minimum(node.Right);

originalColor = y.Color;

x = y.Right;

if (y.Parent == node)

x.Parent = y;

else

{

Transplant(y, y.Right);

y.Right = node.Right;

y.Right.Parent = y;

}

Transplant(node, y);

y.Left = node.Left;

y.Left.Parent = y;

y.Color = node.Color;

}

if (originalColor == Color.Black)

FixDelete(x);

}

private void FixDelete(RBTreeNode x)

{

while (x != root && x.Color == Color.Black)

{

if (x == x.Parent.Left)

{

var w = x.Parent.Right;

if (w.Color == Color.Red)

{

w.Color = Color.Black;

x.Parent.Color = Color.Red;

RotateLeft(x.Parent);

w = x.Parent.Right;

}

if (w.Left.Color == Color.Black && w.Right.Color == Color.Black)

{

w.Color = Color.Red;

x = x.Parent;

}

else

{

if (w.Right.Color == Color.Black)

{

w.Left.Color = Color.Black;

w.Color = Color.Red;

RotateRight(w);

w = x.Parent.Right;

}

w.Color = x.Parent.Color;

x.Parent.Color = Color.Black;

w.Right.Color = Color.Black;

RotateLeft(x.Parent);

x = root;

}

}

else

{

var w = x.Parent.Left;

if (w.Color == Color.Red)

{

w.Color = Color.Black;

x.Parent.Color = Color.Red;

RotateRight(x.Parent);

w = x.Parent.Left;

}

if (w.Right.Color == Color.Black && w.Left.Color == Color.Black)

{

w.Color = Color.Red;

x = x.Parent;

}

else

{

if (w.Left.Color == Color.Black)

{

w.Right.Color = Color.Black;

w.Color = Color.Red;

RotateLeft(w);

w = x.Parent.Left;

}

w.Color = x.Parent.Color;

x.Parent.Color = Color.Black;

w.Left.Color = Color.Black;

RotateRight(x.Parent);

x = root;

}

}

}

x.Color = Color.Black;

}

private void Transplant(RBTreeNode u, RBTreeNode v)

{

if (u.Parent == null)

root = v;

else if (u == u.Parent.Left)

u.Parent.Left = v;

else

u.Parent.Right = v;

v.Parent = u.Parent;

}

private RBTreeNode Minimum(RBTreeNode node)

{

while (node.Left != nullNode)

node = node.Left;

return node;

}

private void RotateLeft(RBTreeNode x)

{

var y = x.Right;

x.Right = y.Left;

if (y.Left != nullNode)

y.Left.Parent = x;

y.Parent = x.Parent;

if (x.Parent == null)

root = y;

else if (x == x.Parent.Left)

x.Parent.Left = y;

else

x.Parent.Right = y;

y.Left = x;

x.Parent = y;

}

private void RotateRight(RBTreeNode x)

{

var y = x.Left;

x.Left = y.Right;

if (y.Right != nullNode)

y.Right.Parent = x;

y.Parent = x.Parent;

if (x.Parent == null)

root = y;

else if (x == x.Parent.Right)

x.Parent.Right = y;

else

x.Parent.Left = y;

y.Right = x;

x.Parent = y;

}

}

}

public enum Color

{

Red,

Black

}

public class RBTreeNode

{

public string Value;

public Color Color;

public RBTreeNode Left, Right, Parent;

public RBTreeNode(string value)

{

Value = value;

Color = Color.Red;

}

}

class Program

{

static void Main()

{

Random rand = new Random();

List<int> sizes = new List<int>();

for (int i = 100; i <= 10000; i += 100)

sizes.Add(i);

foreach (var size in sizes)

{

double insertTime = 0, searchTime = 0, deleteTime = 0;

for (int trial = 0; trial < 5; trial++)

{

var words = GenerateRandomWords(size, rand);

RedBlackTree tree = new RedBlackTree();

// Вставка

DateTime start = DateTime.Now;

foreach (var word in words)

tree.Insert(word);

insertTime += (DateTime.Now - start).TotalMilliseconds;

// Поиск

start = DateTime.Now;

foreach (var word in words)

tree.Search(word);

searchTime += (DateTime.Now - start).TotalMilliseconds;

// Удаление

start = DateTime.Now;

foreach (var word in words)

tree.Delete(word);

deleteTime += (DateTime.Now - start).TotalMilliseconds;

}

Console.WriteLine($"{size} | {insertTime / 5:F2} | {searchTime / 5:F2} | {deleteTime / 5:F2}");

}

}

static List<string> GenerateRandomWords(int count, Random rand)

{

var words = new List<string>();

const string chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";

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

{

string word = new string(Enumerable.Repeat(chars, 5).Select(s => s[rand.Next(s.Length)]).ToArray());

words.Add(word);

}

return words;

}

}