## **RB tree**

Красно-чёрное дерево (англ. red-black tree, RB tree) — 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».

**Принципы организации (свойства) КЧД:**

1. Корень дерева **черный**.

2. Все листья, не содержащие данных, **черные**

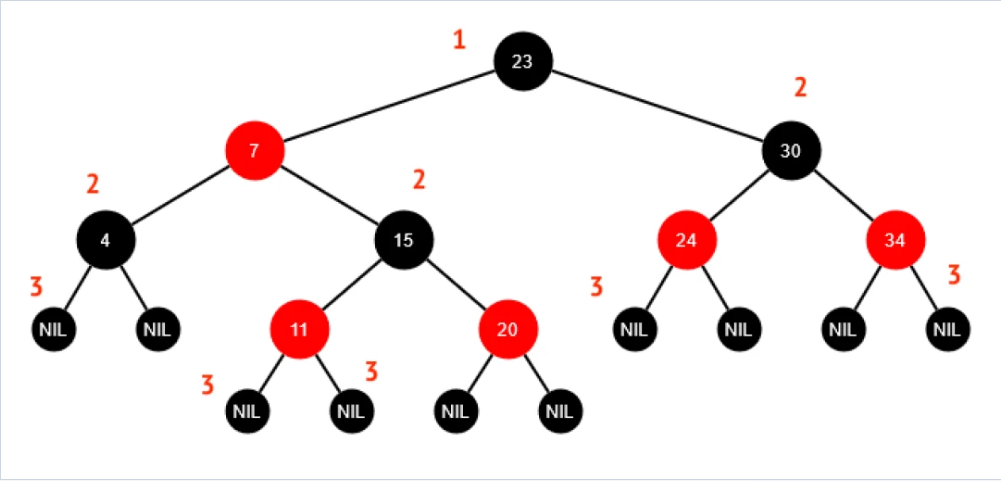
3. Оба потомка каждого красного узла - **черные**

4. Глубина в **черных** узлах одинаковая для любого поддерева

Листовые узлы красно-чёрных деревьев не содержат данных, благодаря

чему не требуют выделения памяти — достаточно записать в узле-предке

в качестве указателя на потомка нулевой указатель (NIL).



**Основные операции выполняемые над деревом это:**

1. Поиск (search)

2. Вставка элемента (insert)

3. Удаление элемента (delete)

Последние две операции очевидно приводят к изменению структуры дерева, а следовательно, их результатом может стать нарушение сбалансированности дерева.

Операции вставки, удаления и поиска для КЧД **составляют O(log n)**, где n - количество узлов в дереве, поскольку для их выполнения нам нужно дойти до нужного узла, на каждом шаге отметая одно из поддеревьев. В случае, когда вставка или удаление привели к нарушению свойств КЧД, необходимо выполнить перебалансировку. Балансировка состоит из 2-ух операций: перекраска O(1) и ротации O(1). Каждая операция балансировки занимает константное время, поскольку состоит из перезаписи ссылок у дочерних и родительских элементов, а также информации об их цвете. Однако при вставке или удалении элемента может возникнуть ситуация, при которой требуется балансировать дерево от самого нижнего узла вплоть до корня. Поскольку гарантируется, что максимальная высота КЧД, состоящего из n узлов не более 2log(n + 1), то в худшем случае перебалансировка может занять log(n) - операций. Затраты по **памяти** для вставки составляют **O(1)**, поскольку заключается только в создании нового узла, операции балансировки и перекраски дополнительной памяти не требуют.

***Вставка узла в дерево***

Чтобы вставить узел, мы сначала ищем в дереве место, куда его следует добавить. Новый узел всегда добавляется как лист, поэтому оба его потомка являются **NIL**-узлами и предполагаются черными. После вставки красим узел в красный цвет. После этого смотрим на предка и проверяем, не нарушается ли красно-черное свойство. Если необходимо, мы перекрашиваем узел и производим поворот, чтобы сбалансировать дерево.

1. Добавляем узел красного (R) цвета.

2. Если это корень, то он перекрашивается в черный цвет (B) и

свойства не нарушаются.

3. Если родитель для вставляемого узла имеет цвет B, то свойства не

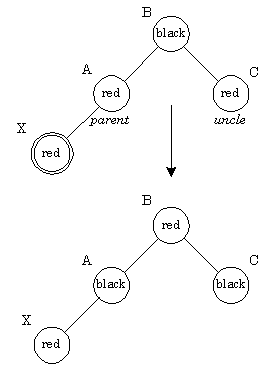
нарушаются.

4. Если родитель для вставляемого узла имеет цвет R, то свойства нарушены.

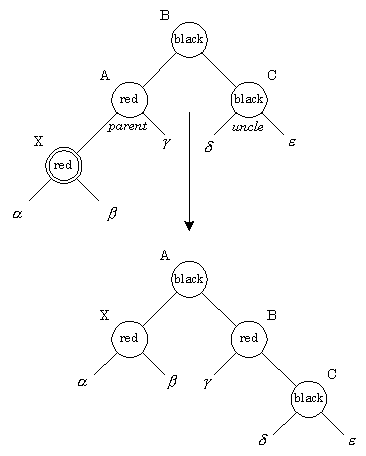
Вставив красный узел с двумя **NIL**-потомками, мы сохраняем свойство черной высоты (свойство 4). Однако, при этом может оказаться нарушенным свойство 3, согласно которому оба потомка красного узла обязательно черны. В нашем случае оба потомка нового узла черны по определению (поскольку они являются **NIL**-узлами), так что рассмотрим ситуацию, когда предок нового узла красный: при этом будет нарушено свойство 3. Достаточно рассмотреть следующие два случая:

* *Красный предок, красный "дядя"*: Ситуацию красный-красный иллюстрирует рис. 1. У нового узла **X** предок и "дядя" оказались красными. Простое перекрашивание избавляет нас от красно-красного нарушения. После перекраски нужно проверить "дедушку" нового узла (узел **B**), поскольку он может оказаться красным. Обратите внимание на распространение влияния красного узла на верхние узлы дерева. В самом конце корень мы красим в черный цвет корень дерева. Если он был красным, то при этом увеличивается черная высота дерева.
* *Красный предок, черный "дядя"*: На рис. 3.7 представлен другой вариант красно-красного нарушения - "дядя" нового узла оказался черным. Здесь узлы может понадобиться вращать, чтобы скорректировать поддеревья. В этом месте алгоритм может остановиться из-за отсутствия красно-красных конфликтов и вершина дерева (узел A) окрашивается в черный цвет. Обратите внимание, что если узел X был в начале правым потомком, то первым применяется левое вращение, которое делает этот узел левым потомком.

Каждая корректировка, производимая при вставке узла, заставляет нас подняться в дереве на один шаг. В этом случае до остановки алгоритма будет сделано 1 вращение (2, если узел был правым потомком). Метод удаления аналогичен.



**Рис. 1**: Вставка - Красный предок, красный "дядя



**Рис. 2**: Вставка - красный предок, черный "дядя"

## ***Удаление***

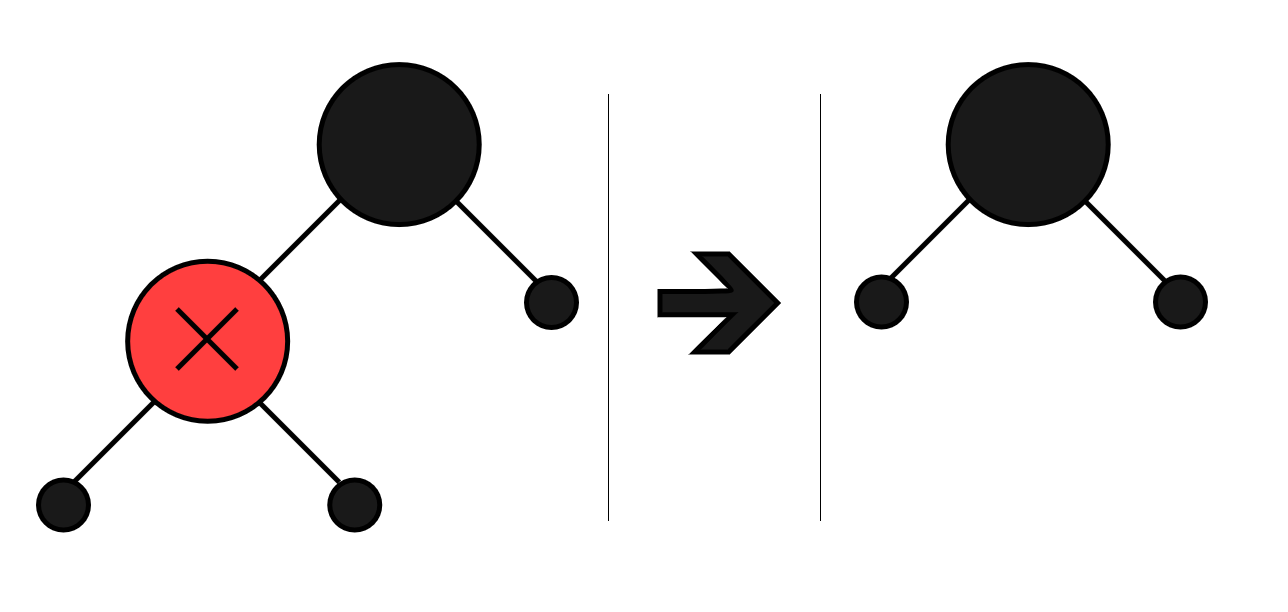
### Вот шаги, необходимые для удаления узла в красно-черном дереве:

1. Если у удаляемого узла нет дочерних элементов, просто удалите его и обновите родительский узел.
2. Если у удаляемого узла есть только один дочерний узел, замените узел на его дочерний узел.
3. Если у удаляемого узла есть два дочерних элемента, замените узел его преемником по порядку, который является крайним левым узлом в правом поддереве. Затем удалите узел-преемник по порядку, как если бы у него было не более одного дочернего узла.
4. После удаления узла свойства красно-черного могут быть нарушены. Чтобы восстановить эти свойства, в узлах дерева выполняются некоторые изменения цвета и повороты. Изменения аналогичны тем, которые выполняются при вставке, но с другими условиями.
5. Операция удаления в красно-черном дереве в среднем занимает O (log n) времени, что делает его хорошим выбором для поиска и удаления элементов в больших наборах данных.

Итак, ситуация: нам нужно удалить конкретную вершину из дерева. Сложность в том, чтобы соблюдать правила построения дерева, иначе красно-черное дерево перестанет быть таковым. То есть тот самый критерий "необходимости" балансировки -- нарушение правил красно-черного дерева при добавлении или удалении вершин.

**1) Удаление красной вершины с 0 детьми**

Для удобства удаляемую (или уже удаленную, понадобится в дальнейшем) вершину будем обозначать с крестом на ней.

NULL-вершины здесь и далее -- маленькие черные вершины

В этом случае мы просто заменяем данную вершину на NULL-вершину. Никакие пункты из вышеописанных правил не нарушены (та же черная высота не нарушается), дерево все еще корректно.

**2)Удаление красной или черной вершины с 2 детьми**

Да, тут мы уже рассматриваем не только *Красные* вершины, но и *Черные*. Все из-за метода, который здесь можно применить:

Так как у вершины есть два ребенка (а то и поддерева), чтобы сделать наиболее безболезненное удаление, стоит найти кого поставить в качестве *замены* на это же место. Эту замену мы можем найти в этих самых поддеревьях: из левого мы можем искать максимальное, а из правого -- минимальное:

Дерево для примера. 
В целом, у удаляемой вершины могут быть просто два ребенка, из которых нам надо выбрать.

Имея на руках двух кандидатов, нам нужно выбрать из них. Есть несколько вариантов:

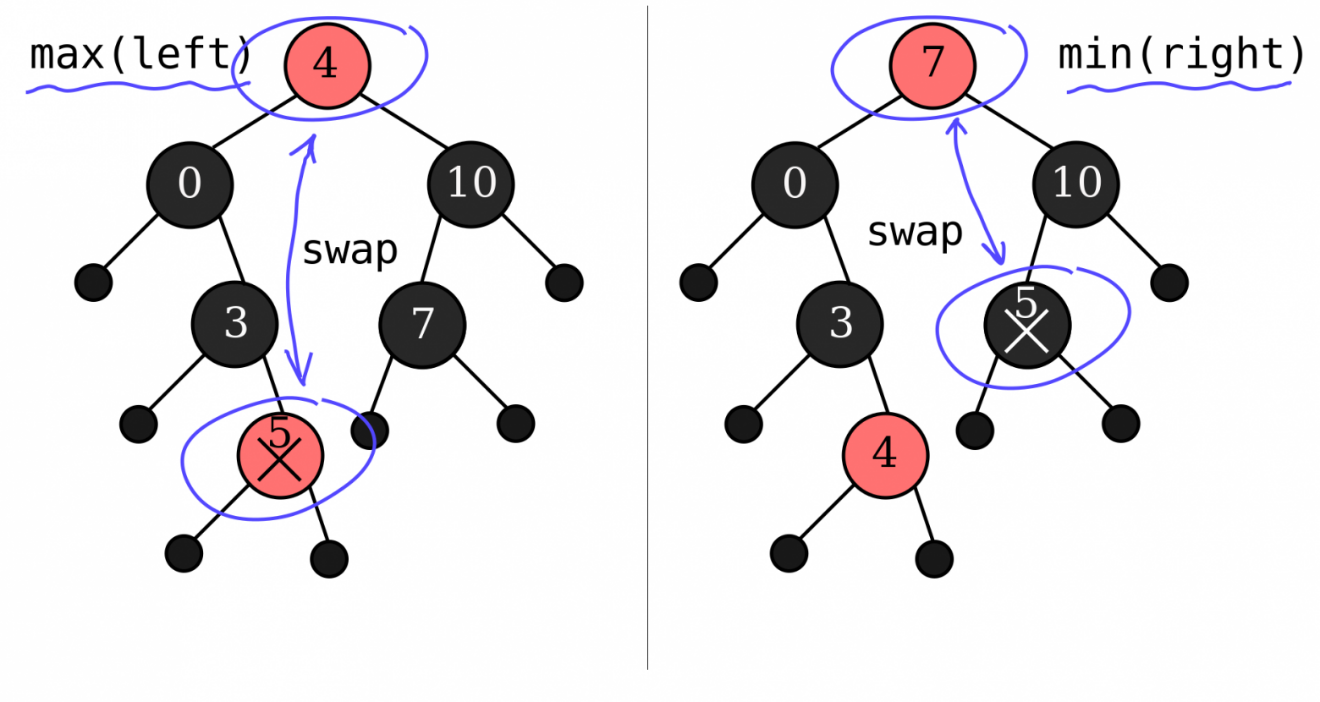
* Ориентироваться на цвет (красные удалять мы уже умеем, и, как мы узнаем далее, они проще)
* Ориентироваться на "дальность" -- на какой глубине относительно удаляемой вершины находится кандидат.
* Ориентироваться на модуль, то есть брать ближайшую по значению вершину.

**На самом деле** этот выбор очень условный и необязательный. Мы можем вовсе брать только минимальную вершину из правого поддерева, результат будет примерно тот же.

Наиболее оптимальный вариант из приведенных зависит от реализации, хочется ли быстрое удаление (предпочтение "красных" кандидатов), или равномерное дерево (выбирать самых дальних, чтобы не было длинных красно-черных ветвей), или что-то иное.

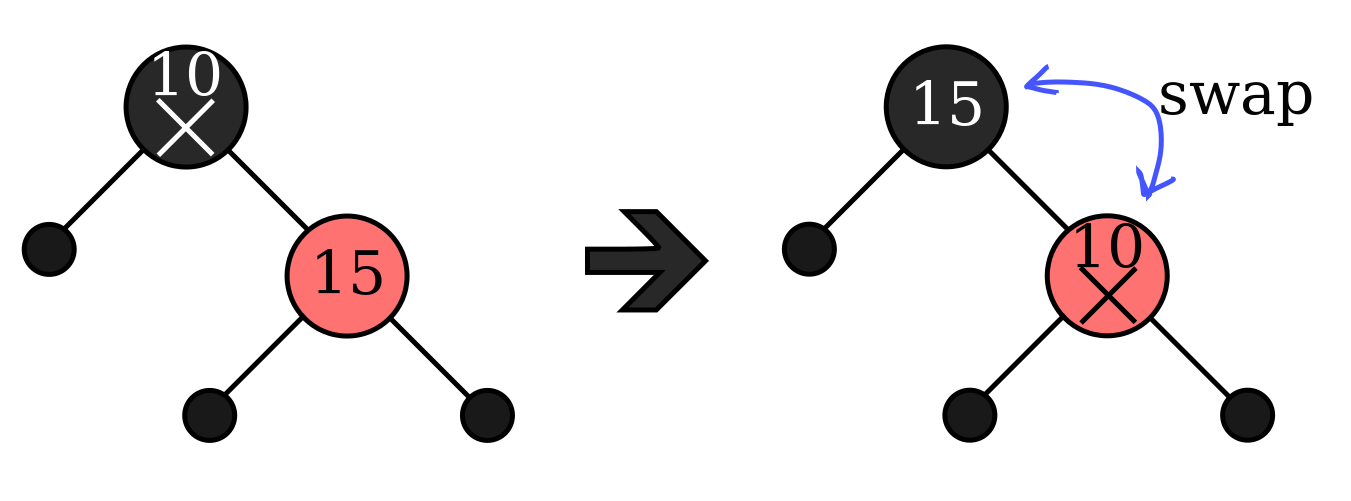
**Процесс замены**. Все же выбрав нужную вершину, нам надо сделать обмен **значений** с ней. Это очень важно, ведь если мы поменяем цвета, то у нас может нарушиться черная высота, а вот замена значений пройдет очень гладко (мы же специально выбрали максимальный или минимальный -- то есть порядок значений не нарушится).

Таким образом, нам останется удалить вершину, с которой мы произвели замену, а у нее уже точно **один**или **ноль** детей (иначе , то есть мы перешли к более простому варианту:



**3) Удаление черной вершины с 1 ребенком**

Итак, рассмотрим, как выглядит подобный вариант:

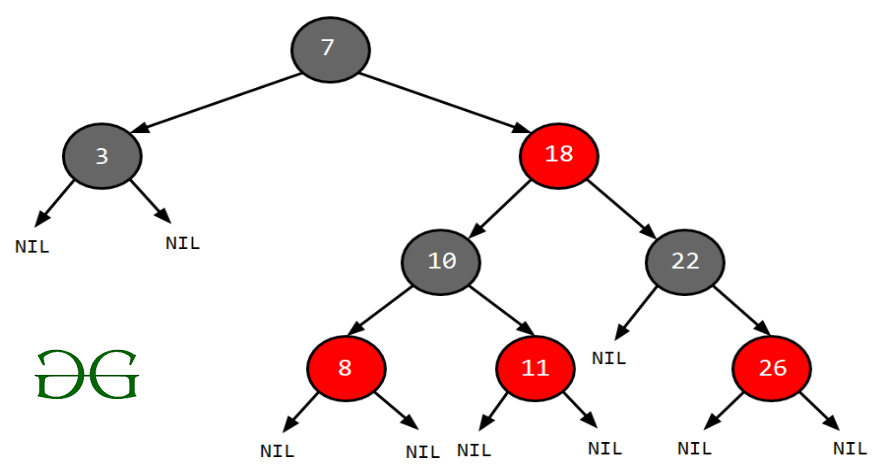


То есть опять же, мы можем поменять **значения** (не цвета!) и начать удалять вершину с **одним** или **нулем** детей. Черная высота (по крайней мере в пределах этой замены) остается той же.

***Поиск:***

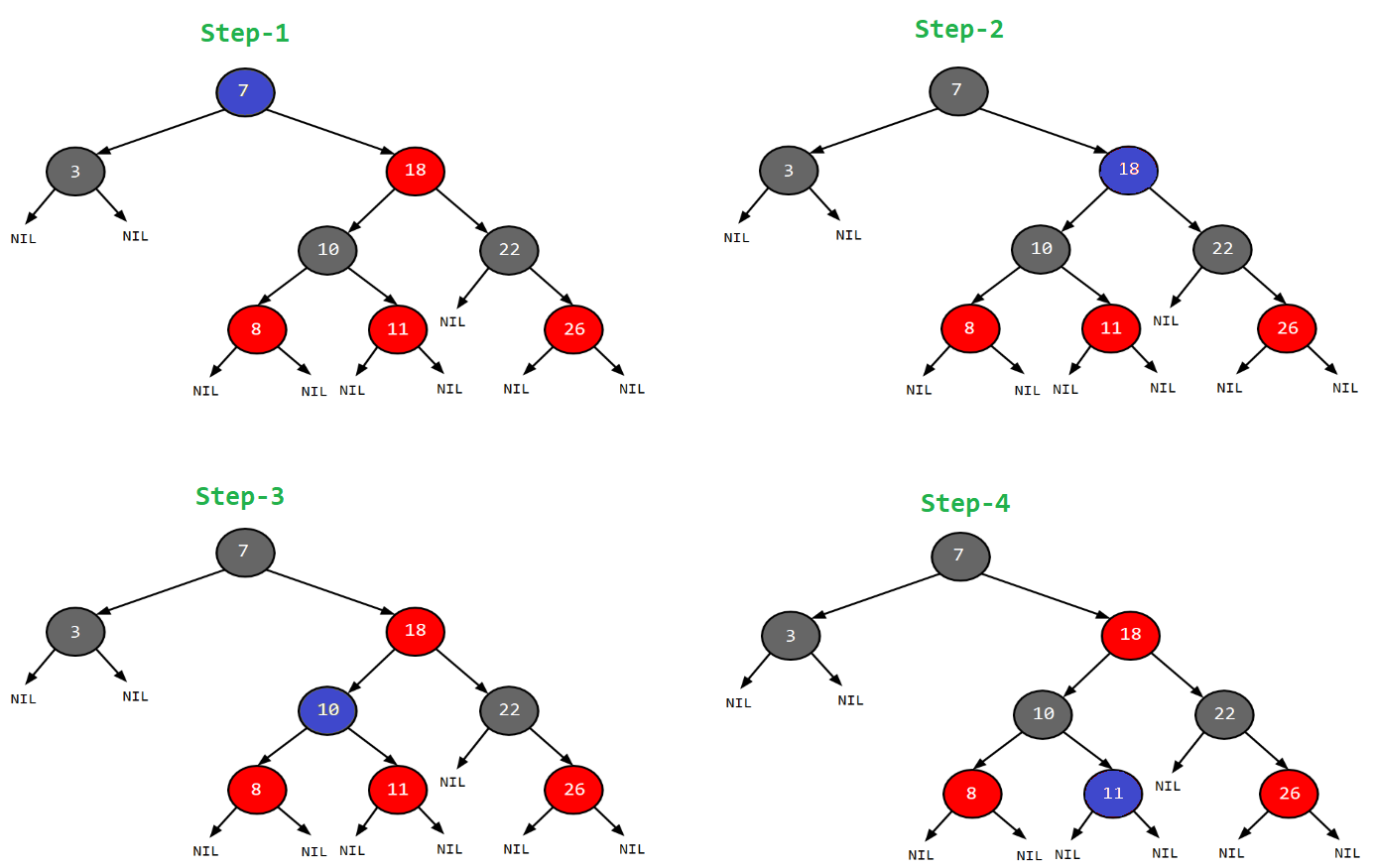
Поскольку каждое красно-черное дерево является частным случаем бинарного дерева, алгоритм поиска в красно-черном дереве аналогичен алгоритму поиска в бинарном дереве.

**Пример: Поиск 11 в следующем красно-черном дереве.** 



**Решение:**

1. Начните с корня.
2. Сравните вставляемый элемент с root, если меньше root, то выполните рекурсию для left, иначе рекурсию для right.
3. Если элемент для поиска найден где угодно, верните true, в противном случае верните false.



*Просто следуйте за синим пузырьком.*

Операция вставки в RBT соответствует некоторым правилам вставки в BST. Если дерево пустое, нам нужно вставить элемент и сделать его черным, потому что это будет корневой узел. Когда дерево не пустое, мы создаем новый узел и окрашиваем его в красный цвет.

Кроме того, **всякий раз, когда мы хотим вставить какой-либо элемент в дерево, цвет нового узла по умолчанию всегда будет красным.** Это означает, что цвет дочернего узла и родительского узла не должен быть красным. Нам только нужно убедиться, что рядом нет красных узлов.

После вставки элемента, если родительский узел вставленного элемента черный, нам не нужно выполнять никаких дополнительных процессов. Это будет сбалансированный RBT. Но если родительский элемент вставляемого элемента красного цвета, нам нужно проверить цвет узла-побратима родительского элемента. Следовательно, нам нужно проверить цвет узла, который находится на том же уровне, что и родительский узел.

Операция удаления в RBT такая же, как и в BST. Сначала мы должны пройти по дереву, пока не будет найден нужный узел. Как только мы находим узел, мы удаляем его из RBT.

**Если мы удалим любой красный узел, это не нарушит никаких условий RBT.** Следовательно, нам просто нужно удалить узел, как в BST. **Однако в случае удаления черного узла это может нарушить условие RBT.** Поэтому, после удаления любого черного узла, мы должны выполнить некоторые действия, такие как вращение и перекраска, чтобы сбалансировать RBT.

Источники:

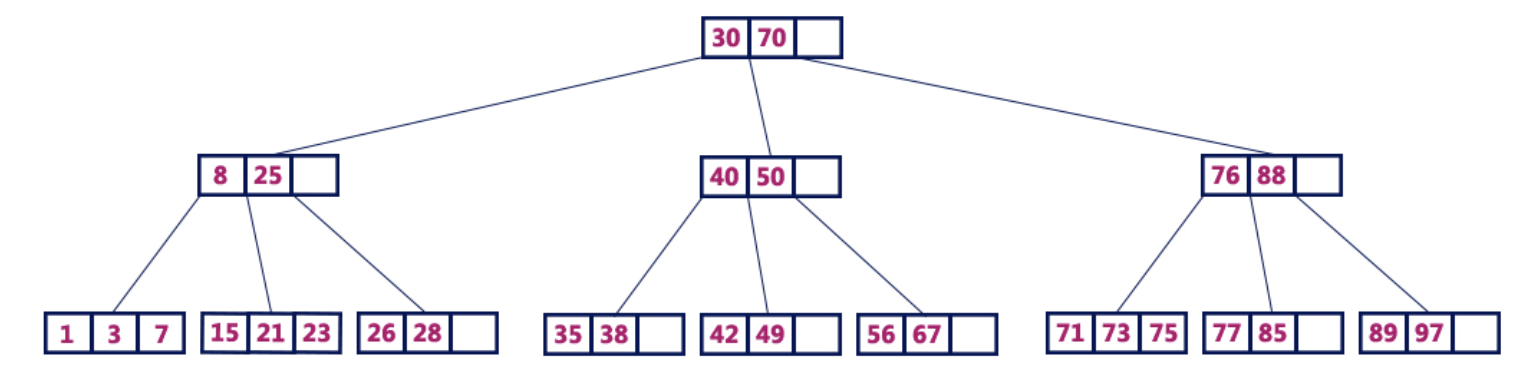
[Красночерное дерево (sonofgabe.github.io)](https://sonofgabe.github.io/4.html)

[Red-Black Tree vs. AVL Tree | Baeldung on Computer Science (turbopages.org)](https://translated.turbopages.org/proxy_u/en-ru.ru.88b7d4e4-6554bee5-c27af078-74722d776562/https/www.baeldung.com/cs/red-black-tree-vs-avl-tree)

[Deletion in Red-Black Tree - GeeksforGeeks (turbopages.org)](https://translated.turbopages.org/proxy_u/en-ru.ru.8ac946ca-6554597c-e2f3a34b-74722d776562/https/www.geeksforgeeks.org/deletion-in-red-black-tree/)

[Сравнение АВЛ и Красно-черного дерева (studylib.ru)](https://studylib.ru/doc/2595760/sravnenie-avl-i-krasno-chernogo-dereva?ysclid=lozrqj59tq696290093)

B-дерево

**B-дерево – это сбалансированное дерево поиска, в котором каждый узел содержит множество ключей и имеет более двух потомков.**  
  
Здесь количество ключей в узле и количество его потомков зависит от порядка B-дерева. Каждое B-дерево имеет порядок.  
  
B-дерево порядка **m** обладает следующими свойствами:  
  
*Свойство 1:* Глубина всех листьев одинакова.  
*Свойство 2:* Все узлы, кроме корня должны иметь как минимум ***(m/2) – 1*** ключей и максимум ***m-1*** ключей.  
*Свойство 3:* Все узлы без листьев, кроме корня (т.е. все внутренние узлы), должны иметь минимум ***m/2*** потомков.  
*Свойство 4:* Если корень – это узел не содержащий листьев, он должен иметь минимум **2** потомка.  
*Свойство 5:*Узел без листьев с **n-1** ключами должен иметь **n** потомков.  
*Свойство 6:* Все ключи в узле должны располагаться в порядке возрастания их значений.  
  
Например, B-дерево 4 порядка содержит максимум 3 значения ключа и максимум 4 потомка для каждого узла.  
  
  
*B-дерево 4 порядка*

**Операции над B-деревом**  
  
Над B-деревом можно проводить следующие операции:

1. Поиск
2. Вставка
3. Удаление

**Поиск по B-дереву**  
  
Поиск по B-дереву аналогичен поиску по двоичному дереву поиска. В двоичном дереве поиска поиск начинается с корня и каждый раз принимается двустороннее решение (пойти по левому поддереву или по правому). В В-дереве поиск также начинается с корневого узла, но на каждом шаге принимается n-стороннее решение, где n – это общее количество потомков рассматриваемого узла. В В-дереве сложность поиска составляет **O(log n)**. Поиск происходит следующим образом:  
  
*Шаг 1:* Считать элемент для поиска.  
*Шаг 2:* Сравнить искомый элемент с первым значением ключа в корневом узле дерева.  
*Шаг 3:* Если они совпадают, вывести: «Искомый узел найден!» и завершить поиск.  
*Шаг 4:* Если они не совпадают, проверить больше или меньше значение элемента, чем текущее значение ключа.  
*Шаг 5:* Если искомый элемент меньше, продолжить поиск по левому поддереву.  
*Шаг 6:* Если искомый элемент больше, сравнить элемент со следующим значением ключа в узле и повторять Шаги 3, 4, 5 и 6 пока не будет найдено совпадение или пока искомый элемент не будет сравнен с последним значением ключа в узле-листе.  
*Шаг 7:* Если последнее значение ключа в узле-листе не совпало с искомым, вывести «Элемент не найден!» и завершить поиск.  
  
**Операция вставки в B-дерево**  
  
В В-дереве новый элемент может быть добавлен только в узел-лист. Это значит, что новая пара ключ-значение всегда добавляется только к узлу-листу. Вставка происходит следующим образом:  
  
*Шаг 1:* Проверить пустое ли дерево.  
*Шаг 2:* Если дерево пустое, создать новый узел с новым значением ключа и его принять за корневой узел.  
*Шаг 3:* Если дерево не пустое, найти подходящий узел-лист, к которому будет добавлено новое значение, используя логику дерева двоичного поиска.  
*Шаг 4:* Если в текущем узле-листе есть незанятая ячейка, добавить новый ключ-значение к текущему узлу-листу, следуя возрастающему порядку значений ключей внутри узла.  
*Шаг 5:* Если текущий узел полон и не имеет свободных ячеек, разделите узел-лист, отправив среднее значение родительскому узлу. Повторяйте шаг, пока отправляемое значение не будет зафиксировано в узле.  
*Шаг 6:* Если разделение происходит с корнем дерева, тогда среднее значение становится новым корнем дерева и высота дерева увеличивается на единицу.