## **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**

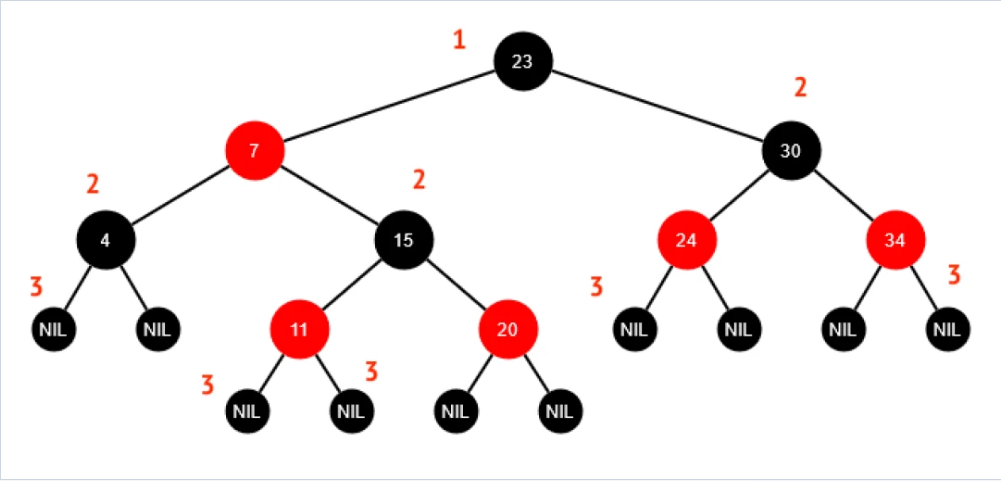
3. Оба потомка каждого красного узла - **черные**

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

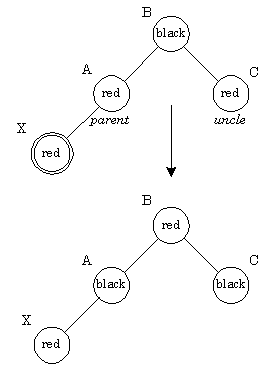
нарушаются.

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

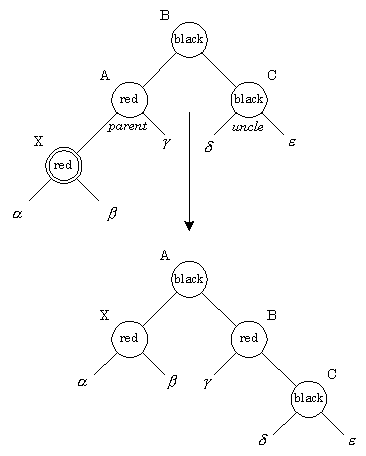
Вставив красный узел с двумя **NIL**-потомками, мы сохраняем свойство черной высоты (свойство 4). Однако, при этом может оказаться нарушенным свойство 3, согласно которому оба потомка красного узла обязательно черны. В нашем случае оба потомка нового узла черны по определению (поскольку они являются **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***

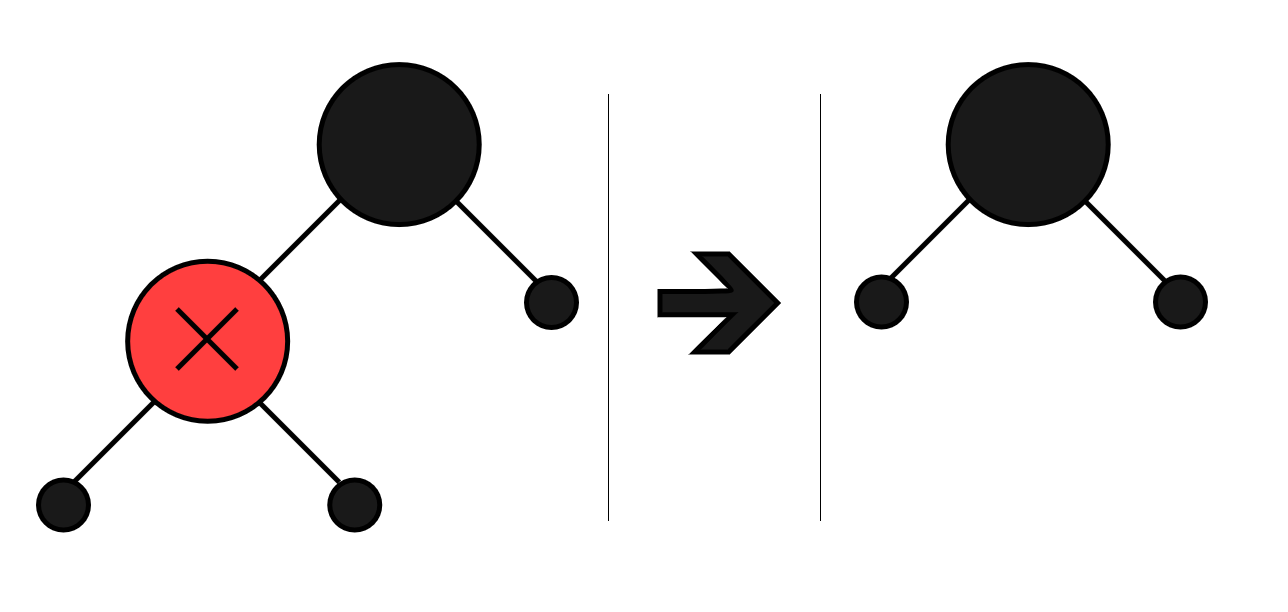
### Вот шаги, необходимые для удаления узла в красно-черном дереве:

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:

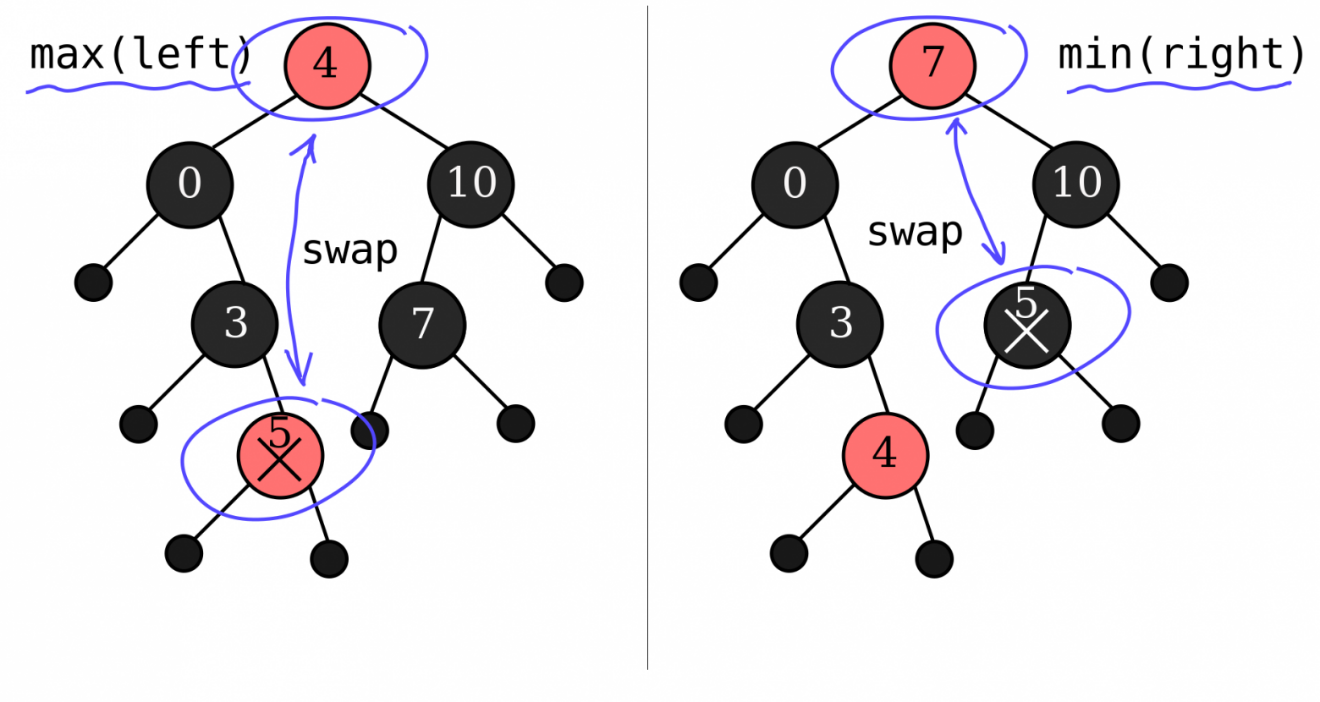
* Ориентироваться на цвет (красные удалять мы уже умеем, и, как мы узнаем далее, они проще)
* Ориентироваться на "дальность" -- на какой глубине относительно удаляемой вершины находится кандидат.
* Ориентироваться на модуль, то есть брать ближайшую по значению вершину.

**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.

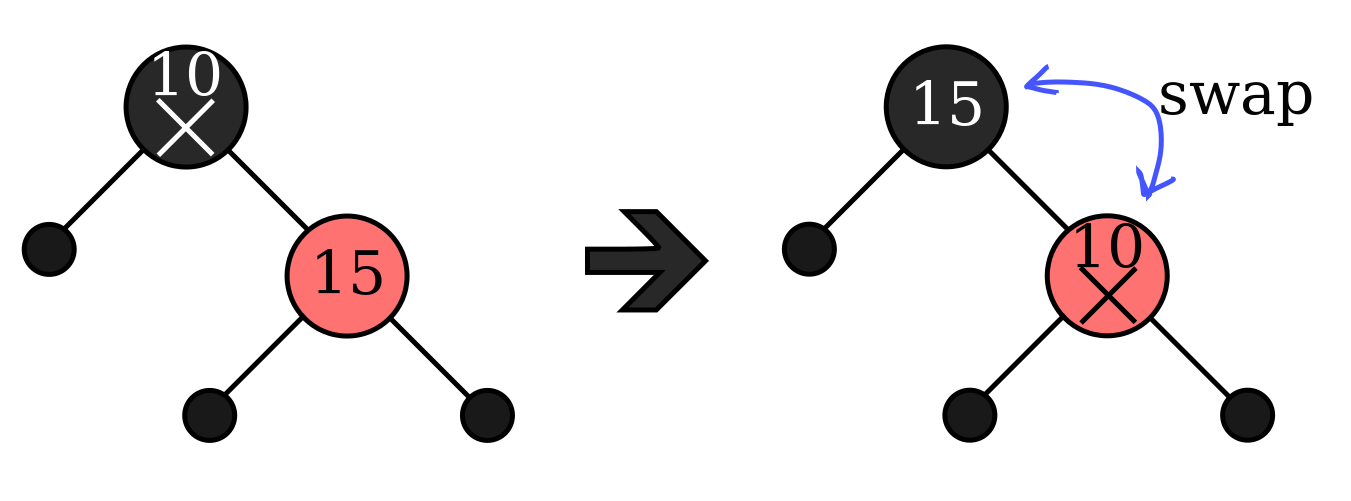
**Процесс замены**. Все же выбрав нужную вершину, нам надо сделать обмен **значений** с ней. Это очень важно, ведь если мы поменяем цвета, то у нас может нарушиться черная высота, а вот замена значений пройдет очень гладко (мы же специально выбрали максимальный или минимальный -- то есть порядок значений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

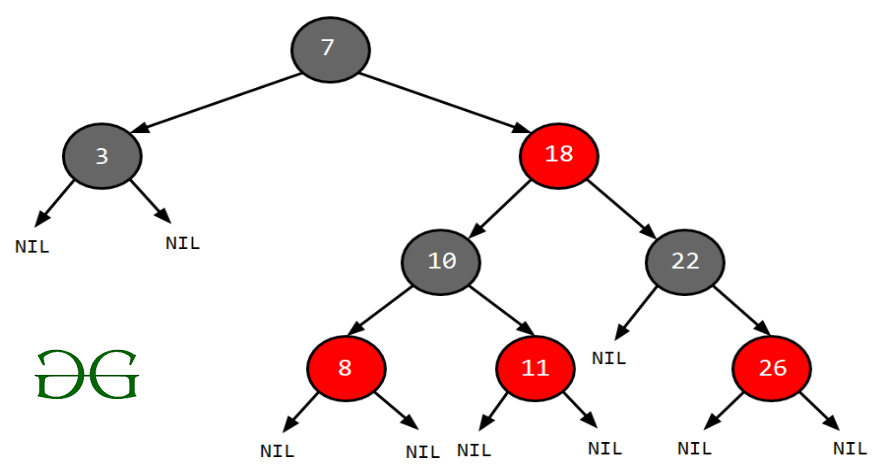


То есть опять же, мы можем поменять **значения** (не цвета!) и начать удалять вершину с **одним** или **нулем** детей. Черная высота (по крайней мере в пределах этой замены) остается той же.

***Поиск:***

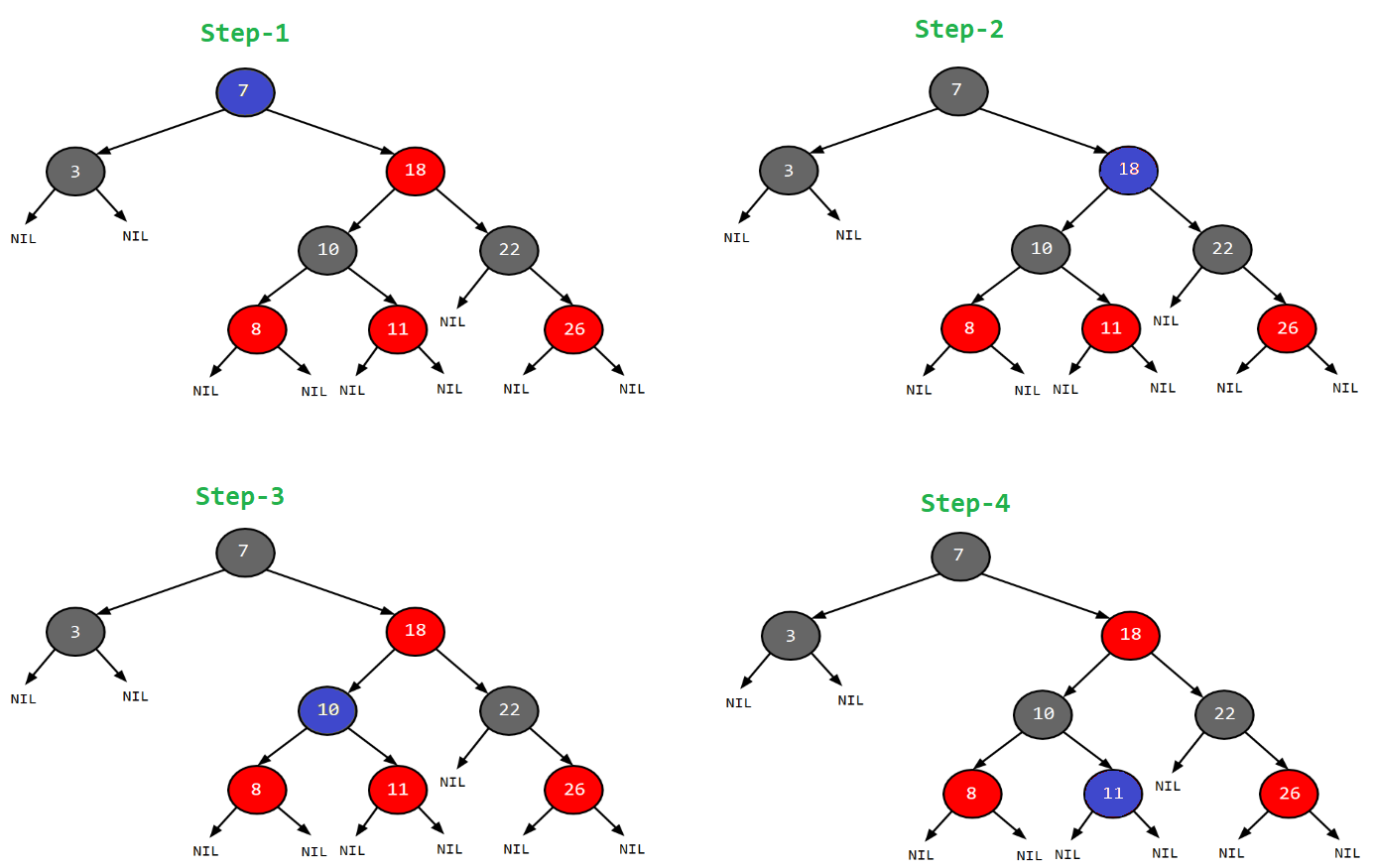
Поскольку каждое красно-черное дерево является частным случаем бинарного дерева, алгоритм поиска в красно-черном дереве аналогичен алгоритму поиска в бинарном дереве.

**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:

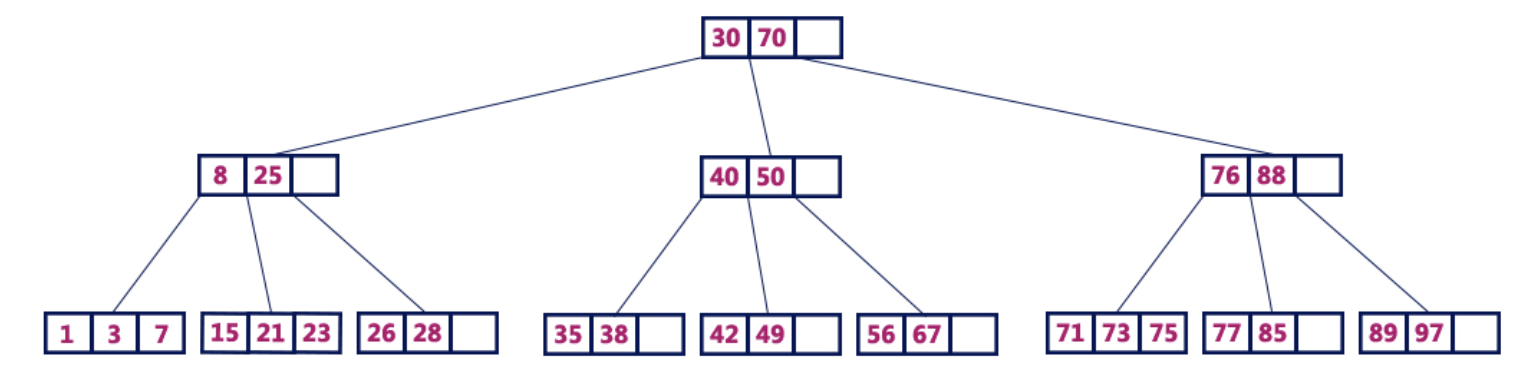
[Красночерное дерево (sonofgabe.github.io)](https://sonofgabe.github.io/4.html)

[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*

### **Применение B-дерева**

B-дерево используется для индексации данных и обеспечивает быстрый доступ к фактическим данным, хранящимся на дисках, так как доступ к значению, хранящемуся в большой базе данных, которая хранится на диске, является очень трудоемким процессом.

Поиск в неиндексированной и несортированной базе данных, содержащей n значений ключей, в худшем случае требует O(n) времени выполнения. Однако, если мы используем B-дерево для индексации этой базы данных, то в худшем случае поиск будет производиться за O(log n) времени.

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

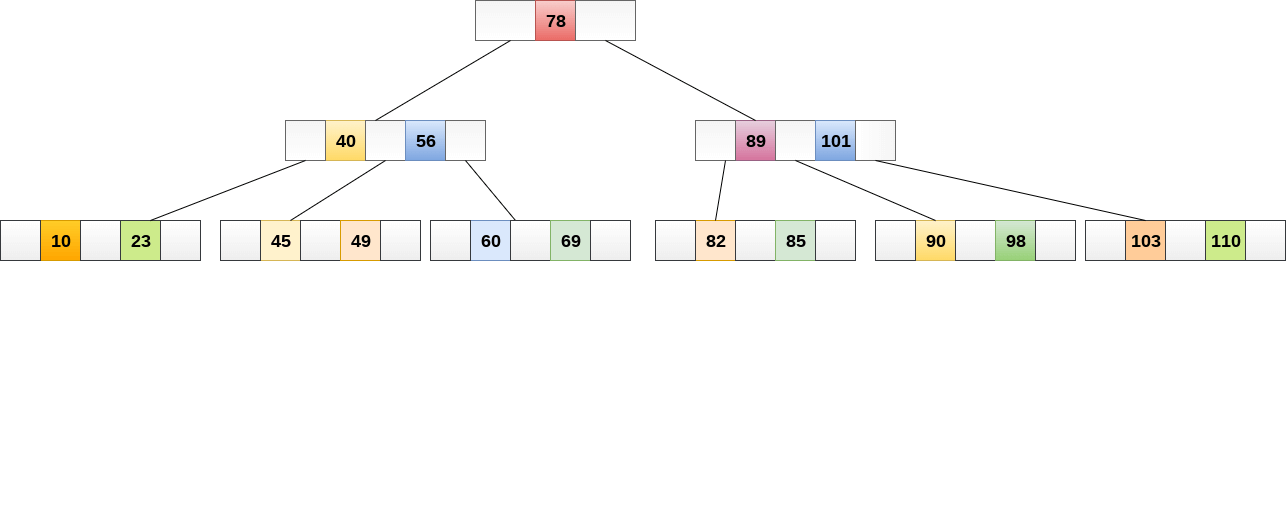
1. Поиск
2. Вставка
3. Удаление

## **Операции с B-деревом**

### **Поиск**

Поиск в B-деревьях аналогичен поиску в бинарном дереве. Например, если мы ищем элемент 49 в следующем B-дереве. Процесс будет выглядеть примерно так:

* Сравните пункт 49 с корневым узлом 78. Начиная с 49 < 78, двигайтесь к его левому поддереву.
* Так как 40<49<56 пройдите по правому поддереву 40.
* 49>45, двигайтесь вправо. Сравните 49.
* Совпадение найдено, возвращается.

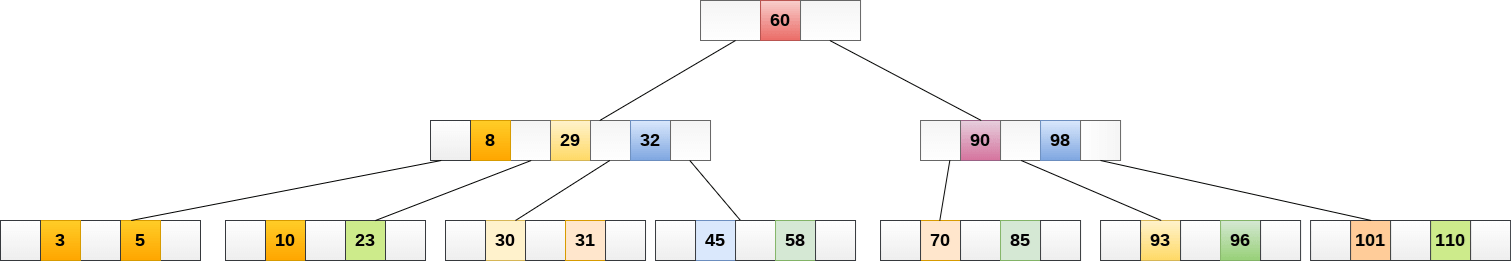
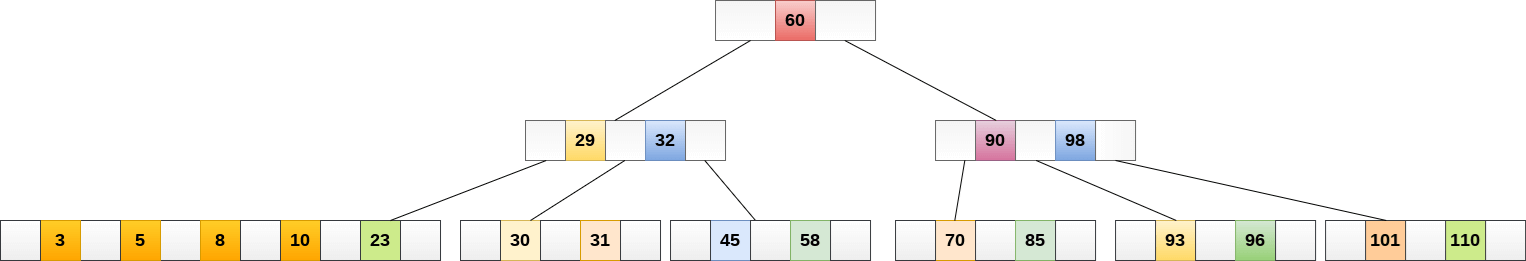
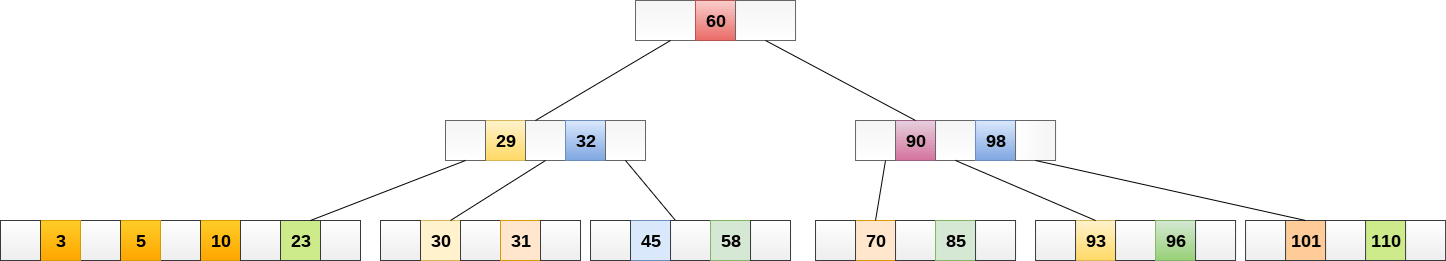
Поиск в B-дереве зависит от высоты дерева. Алгоритму поиска требуется O(log n) времени для поиска любого элемента в B-дереве.

### **Вставка**

Вставки выполняются на уровне листового узла. Для того, чтобы вставить элемент в B-дерево, необходимо следовать следующему алгоритму.

* Пройдитесь по B-дереву, чтобы найти подходящий конечный узел, в который можно вставить этот узел.
* Если конечный узел содержит менее m-1 ключей, то вставляйте элемент в порядке возрастания.
* В противном случае, если конечный узел содержит ключи m-1, выполните следующие действия.
  + Вставьте новый элемент в порядке возрастания элементов.
  + Разделите узел на два узла в медиане.
  + Подтолкните медианный элемент к его родительскому узлу.
  + Если родительский узел также содержит m-1 количество ключей, то разделите и его, выполнив те же действия.

Пример:

Вставьте узел 8 в дерево B порядка 5, показанное на следующем рисунке. 8 будет вставлена справа от 5, поэтому вставьте 8. Узел теперь содержит 5 ключей, что больше, чем (5 -1 = 4 ) ключей. Поэтому отделите узел от медианы, т.е. 8, и подтолкните его к родительскому узлу, как показано ниже.

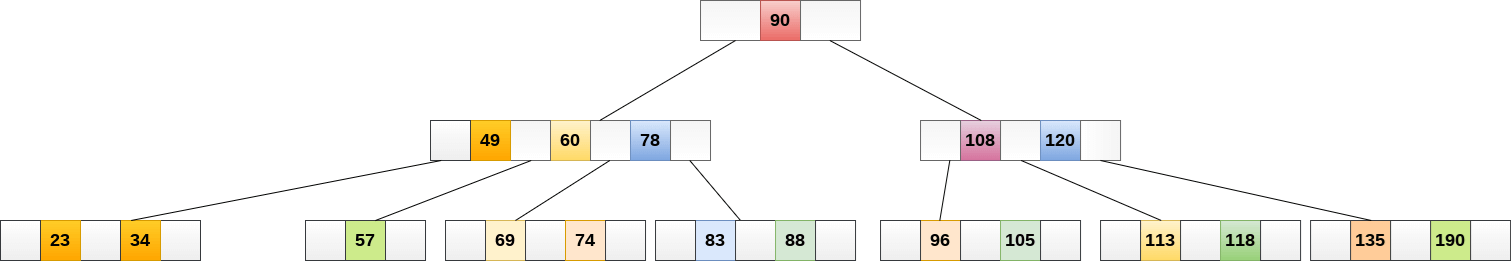
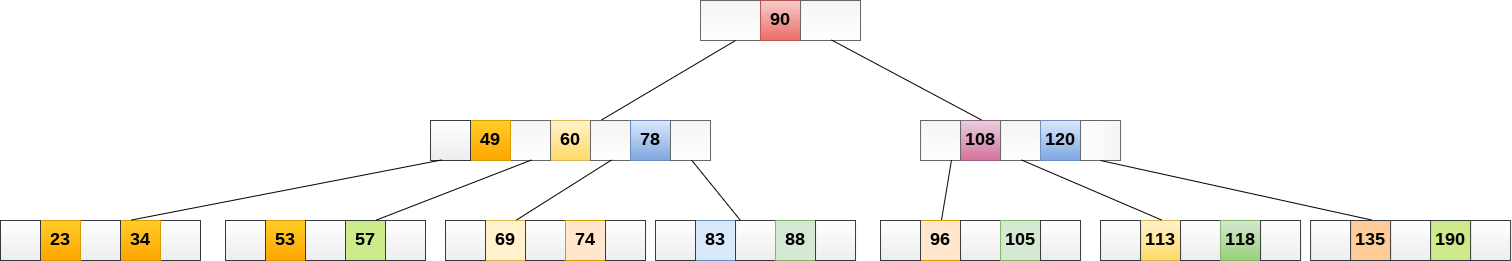
### **Удаление**

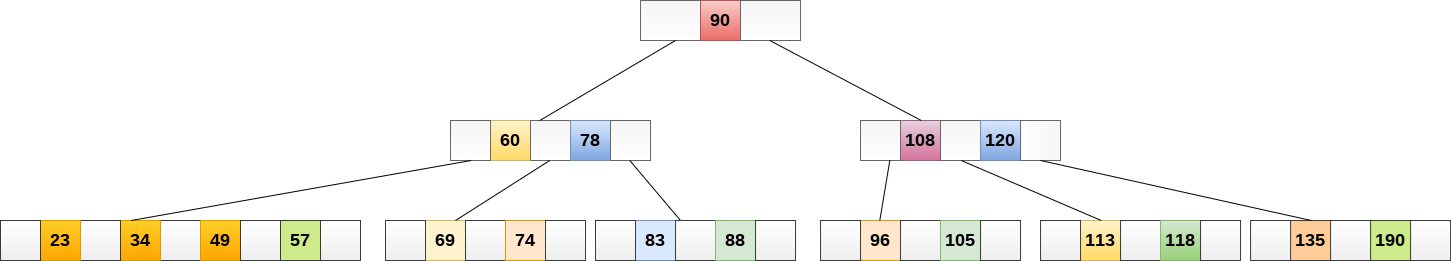
Удаление также выполняется в листовых узлах. Удаляемый узел может быть как конечным, так и внутренним. Для того, чтобы удалить узел из B-дерева, необходимо следовать следующему алгоритму.

* Найдите листовой узел.
* Если в конечном узле более m/2 ключей, удалите нужный ключ из узла.
* Если конечный узел не содержит ключей m/2, то завершите ключи, взяв элемент из восьми или левого брата.
  + Если левый одноуровневый элемент содержит более m/2 элементов, то переместите его самый большой элемент вверх к его родителю и переместите промежуточный элемент вниз к узлу, где удаляется ключ.
  + Если правый одноуровневый элемент содержит более m/2 элементов, то переместите его наименьший элемент вверх к родительскому элементу и переместите промежуточный элемент вниз к узлу, где удаляется ключ.
* Если ни один из одноуровневых узлов не содержит более m/2 элементов, создайте новый конечный узел, соединив два листовых узла и промежуточный элемент родительского узла.
* Если у родителя осталось менее m/2 узлов, то примените описанный выше процесс и к родителю.

Если удаляемый узел является внутренним, замените его последовательным преемником или предшественником. Поскольку преемник или предшественник всегда будет находиться на конечном узле, следовательно, процесс будет аналогичен удалению узла из конечного узла.

Пример:

Удалите узел 53 из дерева B порядка 5, показанного на следующем рисунке. 53 присутствует в правом потомке элемента 49. Удалите его. Теперь 57 - единственный элемент, который остался в узле, минимальное количество элементов, которое должно присутствовать в дереве B порядка 5, равно 2. Он меньше, элементы в его левом и правом поддереве также недостаточны, поэтому объедините его с левым братом и промежуточным элементом родителя, т.е. 49.

Итоговое B-дерево показано следующим образом.

Источники:

[B-дерево — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE)

[Структура данных - B-дерево @ https://jojozhuang.github.io](https://jojozhuang.github.io/algorithm/data-structure-b-tree/)

## Splay-дерево

Splay-дерево — это самобалансирующееся бинарное дерево поиска. Дереву не нужно хранить никакой дополнительной информации, что делает его эффективным по памяти.

Вместо этого после каждого обращения, даже поиска, splay-дерево меняет свою структуру, благодаря splay operation, частью которых являются вращения.

Представим, что мы уже построили дерево на https://habrastorage.org/getpro/habr/post_images/c7f/092/d48/c7f092d481acb49c8a0f96178ceb3119.gif ключах и теперь нам нужно отвечать на запросы, лежит ли заданный ключ в дереве. Может так оказаться, что пользователя интересует в основном один ключ, и остальные он запрашивает только время от времени. Если ключ лежит далеко от корня, то  запросов могут отнять https://habrastorage.org/getpro/habr/post_images/076/63d/c3c/07663dc3c790b8c5e111597f71b68abc.gif времени.

Эти деревья не являются перманентно сбалансированными и на отдельных запросах могут работать даже линейное время. Однако, после каждого запроса они меняют свою структуру, что позволяет очень эффективно обрабатывать часто повторяющиеся запросы. Более того, амортизационная стоимость обработки одного запроса у них http://habrastorage.org/getpro/habr/post_images/84c/07b/cc9/84c07bcc99d5fc8ab9086ace521ed96a.gif, что делает splay-деревья хорошей альтернативой для перманентно сбалансированных собратьев.

* Пример использования - сетевой маршрутизатор. Сетевой маршрутизатор получает сетевые пакеты с высокой скоростью от входящих подключений и должен быстро решить, по какому исходящему проводу отправлять каждый пакет, на основе IP-адреса в пакете. Маршрутизатору нужна большая таблица (карта), которую можно использовать для поиска IP-адреса и определения того, какое исходящее соединение использовать. Если IP-адрес использовался один раз, он, вероятно, будет использоваться снова, возможно, много раз. В этой ситуации Splay-деревья могут обеспечить хорошую производительность.

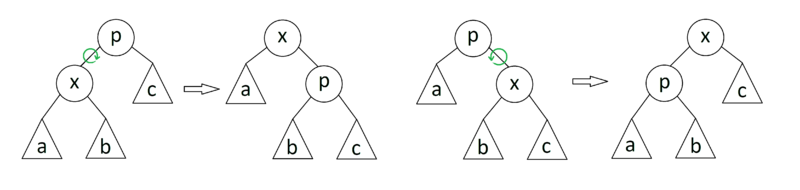
**Операции со splay-деревом**

### Splay (расширение)

Основная операция дерева. Заключается в перемещении вершины в корень при помощи последовательного выполнения трёх операций: Zig, Zig-Zig и Zig-Zag. Обозначим вершину, которую хотим переместить в корень за *x*, её родителя — *p*, а родителя *p* (если существует) — *g*.

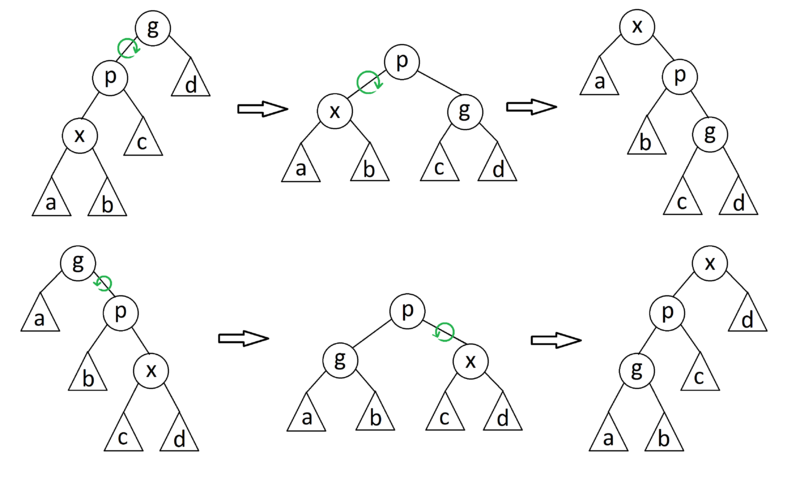
#### **zig**

Если p — корень дерева с сыном x, то совершаем один поворот вокруг ребра (x,p), делая x корнем дерева. Данный случай является крайним и выполняется только один раз в конце, если изначальная глубина x была нечетной.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:%D0%97%D0%B8%D0%B3.png)

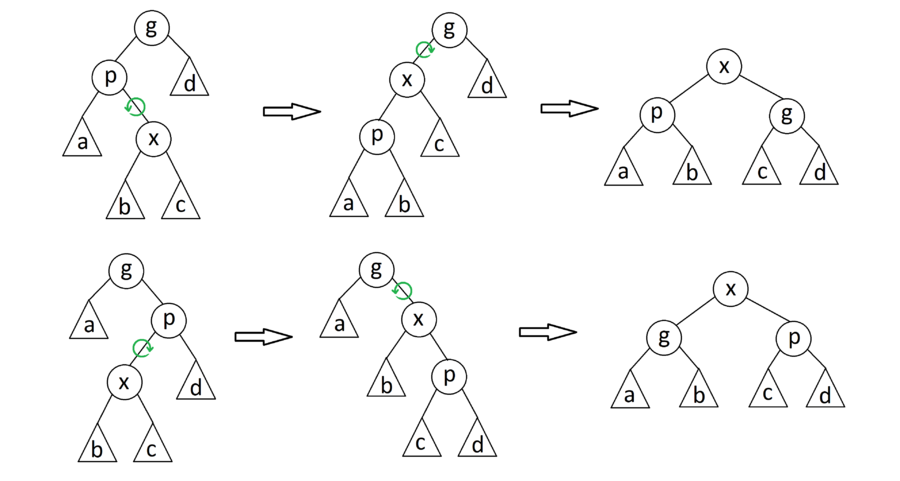
#### **zig-zig**

Если p — не корень дерева, а x и p — оба левые или оба правые дети, то делаем поворот ребра (p,g), где g отец p, а затем поворот ребра (x,p).

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:%D0%97%D0%B8%D0%B3_%D0%B7%D0%B8%D0%B3.png)

#### **zig-zag**

Если p — не корень дерева и x — левый ребенок, а p — правый, или наоборот, то делаем поворот вокруг ребра (x,p), а затем поворот нового ребра (x,g), где g — бывший родитель p.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:%D0%97%D0%B8%D0%B3_%D0%B7%D0%B0%D0%B32.png)

Данная операция занимает O(d) времени, где d— длина пути от x до корня.

1. **Search (поиск элемента)** Поиск выполняется как в обычном двоичном дереве поиска. При нахождении элемента запускаем Splay для него.
2. **Split (разделение дерева на две части)** Для разделения дерева найдем наименьший элемент, больший или равный x, и сделаем для него Splay. После этого отрезаем у корня левого ребёнка и возвращаем 2 получившихся дерева.
3. **Merge (объединение двух деревьев)** У нас есть два дерева tree1 и tree2, причём подразумевается, что все элементы первого дерева меньше элементов второго. Запускаем splay от самого большого элемента в дереве tree1 (пусть это элемент i). После этого корень tree1 содержит элемент i, при этом у него нет правого ребёнка. Делаем tree2 правым поддеревом i и возвращаем полученное дерево.
4. **Insert (добавление элемента)** Вставка происходит как в обычном бинарном дереве поиска, после, запускаем Split() от добавляемого элемента и подвешиваем получившиеся деревья за него.
5. **Remove (удаление элемента)** Находим элемент в дереве, делаем Splay для него, делаем текущим деревом Merge его детей.

* Сложность операций и требование по памяти:
* Затраты по памяти - O(n)
* Splay - O(log(n))
* Search - O(log(n))
* Split - O(log(n))
* Merge - O(log(n))
* Insert - O(log(n))
* Delete - O(log(n))

Расширяющееся дерево предоставляет самоизменяющуюся структуру — структуру, характеризующуюся тенденцией хранить узлы, к которым часто происходит обращение, вблизи верхушки дерева, в то время как узлы, к которым обращение происходит редко, перемещаются ближе к листьям. Таким образом время обращения к часто посещаемым узлам будет меньше, а время обращения к редко посещаемым узлам — больше среднего.

Расширяющееся дерево не обладает никакими явными функциями балансировки, но процесс скоса узлов к корню способствует поддержанию дерева в сбалансированном виде.

Источники:

[Splay-дерево — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/Splay-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE)

[Splay-деревья / Хабр (habr.com)](https://habr.com/ru/companies/JetBrains-education/articles/210296/)

[Algorithms-and-Data-Structures-2021/semester-work-splay-tree (github.com)](https://github.com/Algorithms-and-Data-Structures-2021/semester-work-splay-tree?ysclid=lp9ppqm4cg52559515)

[Splay-дерево — Викиконспекты (ifmo.ru)](https://neerc.ifmo.ru/wiki/index.php?title=Splay-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE)