# B-деревья

B-деревья — это тип самобалансирующегося дерева, предназначенного для хранения данных в памяти внешних устройств, таких как жесткие диски. Они часто используются в базах данных и файловых системах. Вот основные концепции и характеристики B-деревьев:

1. **Структура**: B-дерево — это дерево поиска, в котором узлы могут иметь большее количество дочерних узлов по сравнению с двоичными деревьями. Каждый узел содержит ключи и указатели на дочерние узлы.
2. **Упорядочение**: Ключи в узле упорядочены. Все ключи в поддереве слева от ключа меньше этого ключа, а все ключи в поддереве справа — больше.
3. **Параметр B-дерева**: B-дерево определяется своим параметром *t*, который является минимальным количеством ключей, которые узел должен содержать. Каждый узел, кроме корня, должен содержать как минимум t-1 ключей. Корень должен содержать как минимум один ключ.
4. **Максимальное и минимальное количество ключей**: Каждый узел может содержать максимум 2t-1 ключей.
5. **Сбалансированность**: Все листовые узлы находятся на одном уровне, что обеспечивает сбалансированность и гарантирует, что операции поиска, вставки и удаления выполняются за логарифмическое время.
6. **Операции**:
   * **Поиск**: Похож на поиск в двоичном дереве поиска, но мы можем переходить сразу через несколько ключей в узле.
   * **Вставка**: Новый ключ вставляется в существующий узел. Если узел переполнен (имеет 2t ключей), он разделяется.
   * **Удаление**: Удаление ключа может требовать переструктуризации дерева для сохранения его свойств.
7. **Применение**: B-деревья чрезвычайно полезны в системах, где данные, с которыми происходят операции, не умещаются в основной памяти и должны часто читаться с или записываться на медленные внешние накопители.

B-деревья представляют собой эффективный и масштабируемый способ индексации данных, что делает их важным инструментом в области баз данных и файловых систем.

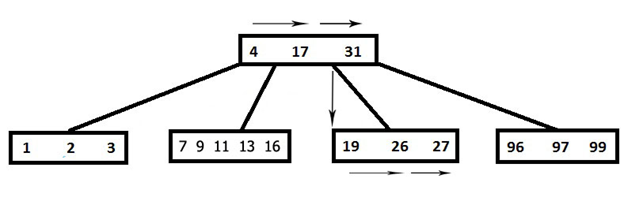
## Структура

При построении B-дерева применяется фактор t, который называется минимальной степенью. Каждый узел, кроме корневого, должен иметь, как минимум t – 1, и не более 2t – 1 ключей. Обозначается n[x] – количество ключей в узле x.  
  
Ключи в узле хранятся в неубывающем порядке. Если x не является листом, то он имеет n[x] + 1 детей. Если занумеровать ключи в узле x, как k[i], а детей c[i], то для любого ключа в поддереве с корнем c[i] (пусть k1), выполняется следующее неравенство – k[i-1] ≤k1≤k[i] (для c[0]: k[i-1] = -∞, а для c[n[x]]: k[i] = +∞). Таким образом, ключи узла задают диапазон для ключей их детей.  
  
Все листья B-дерева должны быть расположены на одной высоте, которая и является высотой дерева. Высота B-дерева с n ≥ 1 узлами и минимальной степенью t ≥ 2 не превышает logt(n+1). Это очень важное утверждение (почему – мы поймем чуть позже)!  
  
h ≤ logt((n+1)/2) — логарифм по основанию t.

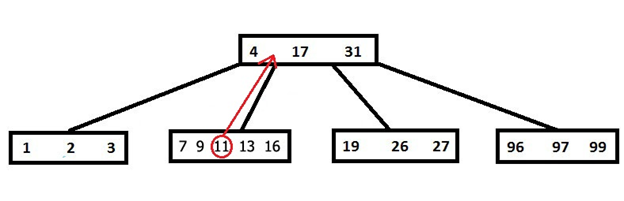
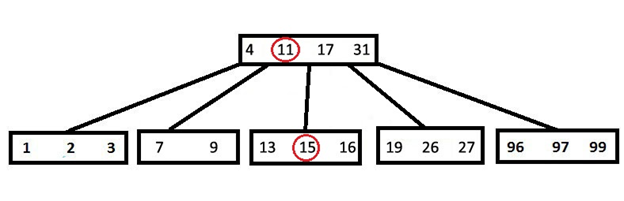
##### **Поиск**

Поиск в B-дереве очень схож с поиском в бинарном дереве, только здесь мы должны сделать выбор пути к потомку не из 2 вариантов, а из нескольких. В остальном — никаких отличий. На рисунке ниже показан поиск ключа 27. Поясним иллюстрацию (и соответственно стандартный алгоритм поиска):

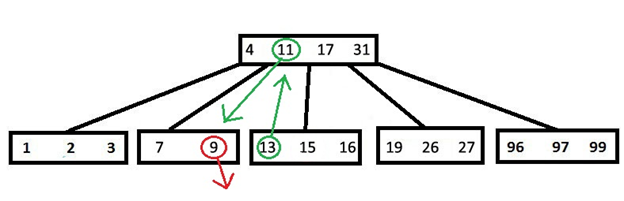
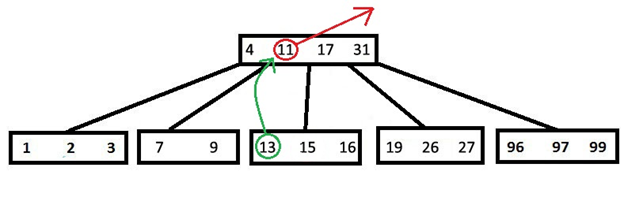
* Идем по ключам корня, пока меньше необходимого. В данном случае дошли до 31.
* Спускаемся к ребенку, который находится левее этого ключа.
* Идем по ключам нового узла, пока меньше 27. В данном случае – нашли 27 и остановились.

  
  
Операция поиска выполняется за время O(t logt n), где t – минимальная степень. Важно здесь, что дисковых операций мы совершаем всего лишь O(logt n)!

##### **Добавление**

В отличие от поиска, операция добавления существенно сложнее, чем в бинарном дереве, так как просто создать новый лист и вставить туда ключ нельзя, поскольку будут нарушаться свойства B-дерева. Также вставить ключ в уже заполненный лист невозможно => необходима операция разбиения узла на 2. Если лист был заполнен, то в нем находилось 2t-1 ключей => разбиваем на 2 по t-1, а средний элемент (для которого t-1 первых ключей меньше его, а t-1 последних больше) перемещается в родительский узел. Соответственно, если родительский узел также был заполнен – то нам опять приходится разбивать. И так далее до корня (если разбивается корень – то появляется новый корень и глубина дерева увеличивается). Как и в случае обычных бинарных деревьев, вставка осуществляется за один проход от корня к листу. На каждой итерации (в поисках позиции для нового ключа – от корня к листу) мы разбиваем все заполненные узлы, через которые проходим (в том числе лист). Таким образом, если в результате для вставки потребуется разбить какой-то узел – мы уверены в том, что его родитель не заполнен!  
  
На рисунке ниже проиллюстрировано то же дерево, что и в поиске (t=3). Только теперь добавляем ключ «15». В поисках позиции для нового ключа мы натыкаемся на заполненный узел (7, 9, 11, 13, 16). Следуя алгоритму, разбиваем его – при этом «11» переходит в родительский узел, а исходный разбивается на 2. Далее ключ «15» вставляется во второй «отколовшийся» узел. Все свойства B-дерева сохраняются!  
  
  
  
  
  
Операция добавления происходит также за время O(t logt n). Важно опять же, что дисковых операций мы выполняем всего лишь O(h), где h – высота дерева.

##### **Удаление**

Удаление ключа из B-дерева еще более громоздкий и сложный процесс, чем вставка. Это связано с тем, что удаление из внутреннего узла требует перестройки дерева в целом. Аналогично вставке необходимо проверять, что мы сохраняем свойства B-дерева, только в данном случае нужно отслеживать, когда ключей t-1 (то есть, если из этого узла удалить ключ – то узел не сможет существовать). Рассмотрим алгоритм удаления:  
1)Если удаление происходит из листа, то необходимо проверить, сколько ключей находится в нем. Если больше t-1, то просто удаляем и больше ничего делать не нужно. Иначе, если существует соседний лист (находящийся рядом с ним и имеющий такого же родителя), который содержит больше t-1 ключа, то выберем ключ из этого соседа, который является разделителем между оставшимися ключами узла-соседа и исходного узла (то есть не больше всех из одной группы и не меньше всех из другой). Пусть это ключ k1. Выберем ключ k2 из узла-родителя, который является разделителем исходного узла и его соседа, который мы выбрали ранее. Удалим из исходного узла нужный ключ (который необходимо было удалить), спустим k2 в этот узел, а вместо k2 в узле-родителе поставим k1. Чтобы было понятнее ниже представлен рисунок (рис.1), где удаляется ключ «9». Если же все соседи нашего узла имеют по t-1 ключу. То мы объединяем его с каким-либо соседом, удаляем нужный ключ. И тот ключ из узла-родителя, который был разделителем для этих двух «бывших» соседей, переместим в наш новообразовавшийся узел (очевидно, он будет в нем медианой).  
Рис. 1.  
  
  
2)Теперь рассмотрим удаление из внутреннего узла x ключа k. Если дочерний узел, предшествующий ключу k содержит больше t-1 ключа, то находим k1 – предшественника k в поддереве этого узла. Удаляем его (рекурсивно запускаем наш алгоритм). Заменяем k в исходном узле на k1. Проделываем аналогичную работу, если дочерний узел, следующий за ключом k, имеет больше t-1 ключа. Если оба (следующий и предшествующий дочерние узлы) имеют по t-1 ключу, то объединяем этих детей, переносим в них k, а далее удаляем k из нового узла (рекурсивно запускаем наш алгоритм). Если сливаются 2 последних потомка корня – то они становятся корнем, а предыдущий корень освобождается. Ниже представлен рисунок (рис.2), где из корня удаляется «11» (случай, когда у следующего узла больше t-1 ребенка).  
Рис.2.  
  
  
Операция удаления происходит за такое же время, что и вставка O(t logt n). Да и дисковых операций требуется всего лишь O(h), где h – высота дерева.  
  
Итак, мы убедились в том, что B-дерево является быстрой структурой данных (наряду с такими, как красно-черное, АВЛ). И еще одно важное свойство, которое мы получили, рассмотрев стандартные операции, – автоматическое поддержание свойства сбалансированности – заметим, что мы нигде не балансируем его специально.

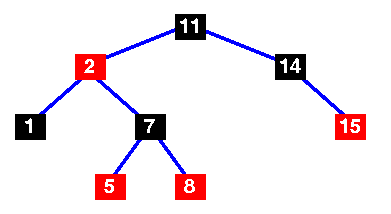
Главная особенность B-дерева - его способность хранить большое количество ключей в одном узле и минимизировать количество операций чтения/записи на внешнем устройстве. Это достигается за счет того, что B-деревья могут иметь много потомков у каждого узла, в отличие от, скажем, двоичного дерева поиска, у которого только два потомка на узел.

Зачем оно нужно? Когда дело доходит до работы с большими объемами данных, которые не помещаются в основной памяти компьютера и должны храниться на жестких дисках или других внешних накопителях, B-деревья становятся очень важными. Они позволяют эффективно организовать и быстро находить данные, существенно сокращая время доступа к ним. Это делает B-деревья идеальным выбором для систем, где производительность и скорость доступа к данным критически важны.

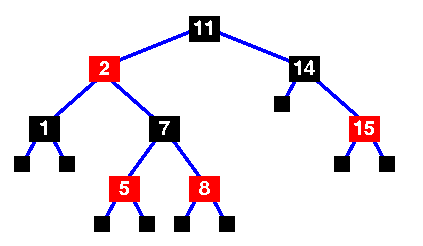
Source: https://habr.com/ru/articles/114154/

# Основные Концепции (красно черное дерево)

1. Определение и Свойства:
   * Красно-чёрное дерево - это самобалансирующееся двоичное поисковое дерево.
   * Каждый узел окрашен в красный или чёрный цвет.
   * Свойства:
     1. Каждый узел либо красный, либо чёрный.
     2. Корень всегда чёрный.
     3. Все листья (NULL) чёрные.
     4. Красный узел не может иметь красного потомка.
     5. Любой путь от узла до его потомков содержит одинаковое количество чёрных узлов.
2. Вставка и Балансировка:
   * Вставка узлов может нарушить свойства красно-чёрного дерева, поэтому после каждой вставки необходимо выполнять балансировку.
   * Балансировка включает в себя повороты и перекраску узлов.
3. Удаление и Балансировка:
   * Удаление также может нарушить баланс, поэтому требуется последующая балансировка.



красно-черное дерево



красно-черное дерево с добавленными сторожевыми узлами. Реализации алгоритмов красно-черного дерева обычно включают сторожевые узлы в качестве удобного средства маркировки того, что вы достигли конечного узла.

Это черные NULL-узлы свойства 3.

Операция левого поворота:

left\_rotate( Tree T, node x ) {

node y;

y = x->right;

/\* Turn y's left sub-tree into x's right sub-tree \*/

x->right = y->left;

if ( y->left != NULL )

y->left->parent = x;

/\* y's new parent was x's parent \*/

y->parent = x->parent;

/\* Set the parent to point to y instead of x \*/

/\* First see whether we're at the root \*/

if ( x->parent == NULL ) T->root = y;

else

if ( x == (x->parent)->left )

/\* x was on the left of its parent \*/

x->parent->left = y;

else

/\* x must have been on the right \*/

x->parent->right = y;

/\* Finally, put x on y's left \*/

y->left = x;

x->parent = y;

}

Операция вставки:

Вставка довольно сложна и включает в себя ряд случаев. Обратите внимание, что мы начинаем со вставки нового узла x в дерево так же, как и для любого другого двоичного дерева, используя функцию Tree\_insert. Этот новый узел помечен красным и, возможно, разрушает свойство «красный-черный». Основной цикл перемещается вверх по дереву, восстанавливая свойство красного-черного.

rb\_insert( Tree T, node x ) {

/\* Insert in the tree in the usual way \*/

tree\_insert( T, x );

/\* Now restore the red-black property \*/

x->colour = red;

while ( (x != T->root) && (x->parent->colour == red) ) {

if ( x->parent == x->parent->parent->left ) {

/\* If x's parent is a left, y is x's right 'uncle' \*/

y = x->parent->parent->right;

if ( y->colour == red ) {

/\* case 1 - change the colours \*/

x->parent->colour = black;

y->colour = black;

x->parent->parent->colour = red;

/\* Move x up the tree \*/

x = x->parent->parent;

}

else {

/\* y is a black node \*/

if ( x == x->parent->right ) {

/\* and x is to the right \*/

/\* case 2 - move x up and rotate \*/

x = x->parent;

left\_rotate( T, x );

}

/\* case 3 \*/

x->parent->colour = black;

x->parent->parent->colour = red;

right\_rotate( T, x->parent->parent );

}

}

else {

/\* repeat the "if" part with right and left

exchanged \*/

}

}

/\* Colour the root black \*/

T->root->colour = black;

}

|  |  |
| --- | --- |
|  | Вот оригинальное дерево..  Обратите внимание, что на следующих диаграммах черные сторожевые узлы опущены для простоты диаграмм. |
|  | Только что была вызвана процедура вставки дерева для вставки узла «4» в дерево.  Это уже не красно-черное дерево — на пути есть два последовательных красных узла.  11 – 2 – 7 – 5 – 4  Отметьте новый узел x и его наследника y.  y красный, поэтому у нас случай 1... |
|  | Измените цвета узлов 5, 7 и 8. |
|  | Переместите x вверх к его предку, 7.  Родитель x (2) по-прежнему красный, так что это еще не красно-черное дерево.  Отметьте наследника, y.  В данном случае наследника черный, поэтому имеем случай 2... |
|  | Переместите x вверх и поверните влево. |
|  | До сих пор не красно-черное дерево.. наследник черный, а родитель х слева.. |
|  | Поменяйте цвета 7 и 11 и поверните вправо.. |
|  | Теперь это красно-черное дерево, так что мы закончили!  O(logn) время! |

Красно-чёрные деревья применяются во многих областях, где требуется эффективное упорядоченное хранение и быстрый доступ к данным:

1. **Базы данных**: Красно-чёрные деревья используются во многих типах баз данных для управления индексами. Благодаря своей способности поддерживать сбалансированность, они обеспечивают быструю вставку, удаление и поиск данных, что является критически важным для производительности баз данных.
2. **Файловые системы**: В файловых системах, таких как NTFS, используются структуры, похожие на красно-чёрные деревья, для управления свободным пространством на диске, а также для организации файлов и каталогов.
3. **Контейнерные библиотеки**: В стандартной библиотеке шаблонов C++ (STL) и во многих других языковых библиотеках ассоциативные контейнеры, такие как **map**, **multimap**, **set**, и **multiset** часто реализуются с использованием красно-чёрных деревьев, предоставляя баланс между временем доступа и эффективностью изменений.
4. **Планировщики задач**: В некоторых системах планирования задач в операционных системах используются красно-чёрные деревья для управления приоритетами задач.
5. **Компьютерная графика**: Алгоритмы, такие как алгоритмы поиска столкновений и разбиение пространства, могут использовать красно-чёрные деревья для управления динамическими наборами объектов.

**Красно-чёрные деревья**

1. **Менее строгая балансировка**: Красно-чёрные деревья не требуют такой строгой балансировки, как AVL-деревья. Это означает, что они могут быть немного "выше" (иметь больше уровней), но всё равно остаются достаточно сбалансированными для эффективной работы.
2. **Проще в изменении**: При вставке или удалении элементов, красно-чёрные деревья часто требуют меньше изменений (перебалансировок). Это делает их более эффективными в ситуациях, где часто происходят вставки и удаления.
3. **Хороший компромисс**: Они представляют собой хороший компромисс между необходимостью поддержания баланса и затратами на перебалансировку при изменении дерева.

**AVL-деревья**

1. **Строгая балансировка**: AVL-деревья поддерживают более строгую балансировку, что делает их "ниже" (с меньшим количеством уровней) и более быстрыми для операций поиска.
2. **Больше перебалансировок**: Эта строгость означает, что при каждом добавлении или удалении элемента часто требуются сложные операции для поддержания баланса, что может быть менее эффективно при частых изменениях.
3. **Лучше для чтения**: Они идеально подходят для ситуаций, где операции чтения (поиска) происходят гораздо чаще, чем операции вставки или удаления.

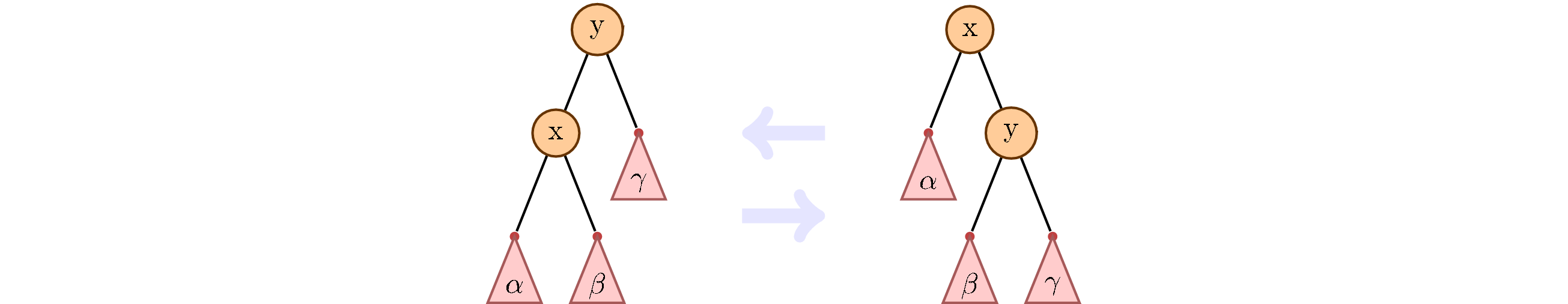
источник: https://www.cs.auckland.ac.nz/software/AlgAnim/red\_black.html

# Splay

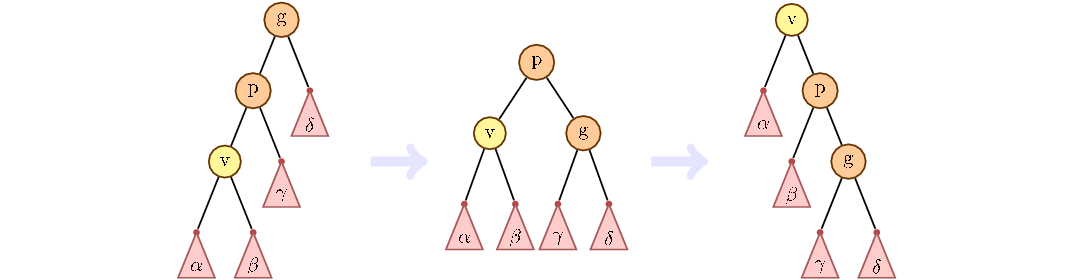
Splay-дерево – это самобалансирующееся дерево бинарного поиска. Основная особенность Splay-дерева заключается в операции Splay, которая перемещает доступный узел в корень дерева с помощью последовательности поворотов

1. **Структура**: Как и во всех бинарных деревьях поиска, каждый узел содержит ключ, левый и правый дочерние узлы.
2. **Операция Splay**: Это основная операция, придающая Splay-дереву его уникальные свойства. Когда происходит доступ к узлу (например, при поиске, вставке или удалении), этот узел "подтягивается" к корню дерева с использованием поворотов. Это улучшает время доступа к недавно использованным элементам.
3. **Повороты**: Повороты бывают нескольких типов: Zig (одиночный поворот), Zig-Zig, Zig-Zag. Они используются для перестройки дерева в процессе операции Splay.
4. **Вставка**: При вставке новый элемент сначала добавляется как в обычное дерево бинарного поиска, а затем выполняется операция Splay для этого элемента.
5. **Поиск**: При поиске элемента сначала выполняется стандартный поиск в бинарном дереве, а затем операция Splay для найденного узла (или последнего доступного узла, если элемент не найден).
6. **Удаление**: Удаляемый узел сначала перемещается в корень с помощью Splay, после чего удаляется. Если у удаляемого узла есть два дочерних узла, его заменяет ближайший узел.
7. **Преимущества**: Splay-дерево обеспечивает хорошее среднее время выполнения операций за счет локальности доступа и амортизированного анализа производительности.
8. **Недостатки**: В худшем случае операции могут занимать линейное время. Также Splay-деревья могут стать несбалансированными.
9. **Применение**: Splay-деревья полезны в приложениях, где доступ к элементам происходит неравномерно и часто осуществляется к недавно использованным элементам.

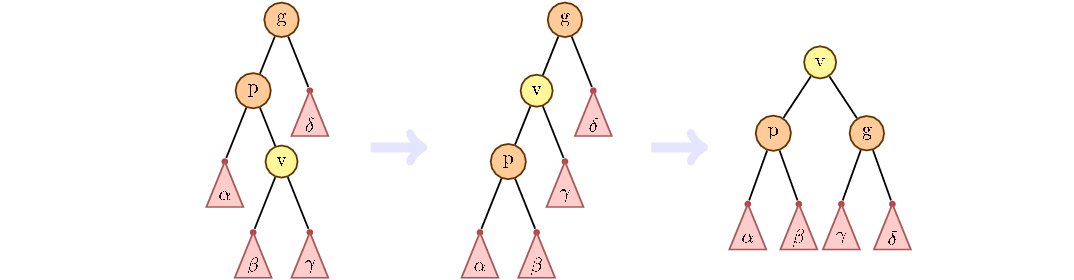
Основной метод move-to-root. После обращения к любой вершине, она поднимается в корень. Подъем реализуется через повороты вершин. За один поворот, можно поменять местами родителя с ребенком, как показано на рисунке ниже.



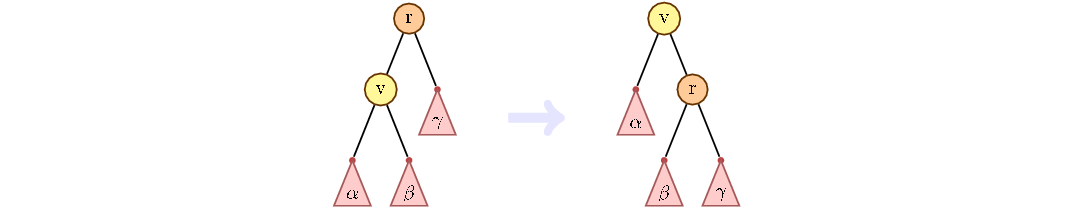
Хитрость splay-дерева в том, что при продвижении вершины вверх, расстояние до корня сокращается не только для поднимаемой вершины, но и для всех ее потомков в текущих поддеревьях. Для этого используется техника zig-zig и zig-zag поворотов.  
  
Основная идея zig-zig и zig-zag поворотов, рассмотреть путь от дедушки к ребенку. Если путь идет только по левым детям или только по правым, то такая ситуация называется zig-zig. Как ее обрабатывать показано на рисунке ниже. Сначала повернуть родителя, потом ребенка.



В противном случае, мы сначала меняем ребенка с текущим родителем, потом с новым.



Если у вершины дедушки нет, делаем обычный поворот:



Описанная выше процедура поднятия вершины с помощью zig-zig и zig-zag поворотов является ключевой для splay-дерева.

# Программная реализация

Функция **rotate** используется для выполнения поворота в Splay-дереве, что является ключевым элементом операции Splay. Эта функция помогает поддерживать баланс дерева при операциях вставки, удаления и поиска.

def rotate(parent, child):

# Получаем родителя родителя (дедушку) текущего узла

gparent = parent.parent

# Если дедушка существует, обновляем его соответствующего потомка (левого или правого)

if gparent != None:

# Если текущий родитель является левым потомком дедушки, то обновляем левого потомка дедушки

if gparent.left == parent:

gparent.left = child

# В противном случае, обновляем правого потомка

else:

gparent.right = child

# Если ребенок является левым потомком, выполняем правый поворот

if parent.left == child:

# При правом повороте левый потомок родителя становится правым потомком ребенка

parent.left, child.right = child.right, parent

# Если ребенок является правым потомком, выполняем левый поворот

else:

# При левом повороте правый потомок родителя становится левым потомком ребенка

parent.right, child.left = child.left, parent

# Обновляем родителя для обоих узлов

keep\_parent(child)

keep\_parent(parent)

# Устанавливаем родителя ребенка равным дедушке

child.parent = gparent

класс Node представляет собой узел для использования в Splay-дереве или любом другом дереве бинарного поиска. Каждый узел хранит ключ (key), который используется для выполнения операций поиска, вставки и удаления. Также узел содержит ссылки на левого и правого потомков (left и right соответственно) и на своего родителя (parent). Это позволяет выполнять различные операции с деревьями, такие как обход, повороты и балансировка.

class Node:

# Конструктор класса Node

def \_\_init\_\_(self, key, left=None, right=None, parent=None):

# Свойство left содержит ссылку на левого потомка узла

self.left = left

# Свойство right содержит ссылку на правого потомка узла

self.right = right

# Свойство parent содержит ссылку на родительский узел

self.parent = parent

# Свойство key хранит значение (ключ) узла

self.key = key

функции **set\_parent** и **keep\_parent** используются для управления и поддержания родительских связей в Splay-дереве или любом другом бинарном дереве поиска.

Функция set\_parent устанавливает родительский узел для данного дочернего узла. Она проверяет, существует ли дочерний узел (**child**), и если это так, то устанавливает его свойство **parent**.

def set\_parent(child, parent):

# Проверяем, существует ли узел child

if child != None:

# Устанавливаем родителя узла child равным parent

child.parent = parent

Функция keep\_parent обновляет родительские ссылки для обоих потомков (если они существуют) узла **v**. Она вызывает функцию **set\_parent** для левого и правого потомков, устанавливая текущий узел **v** в качестве их родителя.

def keep\_parent(v):

# Обновляем родительский узел для левого потомка узла v

set\_parent(v.left, v)

# Обновляем родительский узел для правого потомка узла v

set\_parent(v.right, v)

Эти функции важны для поддержания правильной структуры дерева, особенно после операций, которые изменяют структуру дерева, таких как повороты и Splay-операции. Они помогают обеспечивать, что все родительские и дочерние связи в дереве актуализированы и соответствуют текущему состоянию структуры.

Эта функция выполняет серию поворотов, чтобы переместить заданный узел **v** в корень дерева.

def splay(v):

# Проверяем, является ли узел v корнем дерева (или имеет ли он родителя)

if v.parent == None:

# Если узел v уже является корнем, просто возвращаем его

return v

# Получаем родителя и дедушку узла v

parent = v.parent

gparent = parent.parent

# Случай, когда узел v имеет родителя, но не имеет дедушки (т.е., parent является корнем)

if gparent == None:

# Выполняем одинарный поворот между v и его родителем

rotate(parent, v)

return v

else:

# Определяем, выполняется ли ситуация Zig-Zig

zigzig = (gparent.left == parent) == (parent.left == v)

if zigzig:

# Если да, то сначала выполняем поворот между gparent и parent, а затем между parent и v

rotate(gparent, parent)

rotate(parent, v)

else:

# В случае Zig-Zag сначала выполняем поворот между parent и v, а затем между gparent и v

rotate(parent, v)

rotate(gparent, v)

# Рекурсивно выполняем операцию Splay для узла v, пока он не станет корнем дерева

return splay(v)

В этой функции рассматриваются два основных случая:

1. **Zig (или одиночный поворот)**: когда узел **v** имеет родителя, но не имеет дедушку. В этом случае выполняется один поворот.
2. **Zig-Zig и Zig-Zag**: когда у узла **v** есть и родитель, и дедушка. В зависимости от положения узлов (Zig-Zig или Zig-Zag) выполняется пара поворотов.

Функция **splay** является рекурсивной, что означает, что она будет повторяться до тех пор, пока узел **v** не станет корнем дерева. Это обеспечивает, что недавно использованные или добавленные узлы остаются близко к корню, что улучшает среднее время доступа в дереве.

Начало формы

Процедура поиска в splay-дереве отличается от обычной только на последней стадии: после того, как вершина найдена, мы тянем ее вверх и делаем корнем через процедуру splay.

def find(v, key):

# Проверяем, существует ли текущий узел v

if v == None:

# Если узел не существует, возвращаем None, т.е. ключ не найден

return None

# Проверяем, совпадает ли ключ текущего узла v с искомым ключом

if key == v.key:

# Если да, выполняем операцию Splay для узла v и возвращаем его

return splay(v)

# Если искомый ключ меньше ключа узла v, и у узла v есть левый потомок

if key < v.key and v.left != None:

# Рекурсивно ищем ключ в левом поддереве

return find(v.left, key)

# Если искомый ключ больше ключа узла v, и у узла v есть правый потомок

if key > v.key and v.right != None:

# Рекурсивно ищем ключ в правом поддереве

return find(v.right, key)

# Если мы достигли этой точки, ключ не найден в потомках,

# но мы выполняем операцию Splay для ближайшего узла

return splay(v)Конец формы

Процедура split получает на вход ключ key и делит дерево на два. В одном дереве все значения меньше ключа key, а в другом — больше. Реализуется она просто. Нужно через find найти ближайшую к ключу вершину, вытянуть ее вверх и потом отрезать либо левое, либо правое поддерево (либо оба).

def split(root, key):

# Проверяем, существует ли корневой узел

if root == None:

# Если дерево пустое, возвращаем пару None, None

return None, None

# Находим узел с заданным ключом или ближайший к нему узел в дереве

root = find(root, key)

# Если ключ найденного узла совпадает с заданным ключом

if root.key == key:

# Отсоединяем левое и правое поддеревья от этого узла

set\_parent(root.left, None)

set\_parent(root.right, None)

# Возвращаем левое и правое поддеревья

return root.left, root.right

# Если ключ корня меньше заданного ключа

if root.key < key:

# Отсоединяем правое поддерево

right, root.right = root.right, None

set\_parent(right, None)

# Возвращаем корневое дерево (без правого поддерева) и отсоединенное правое поддерево

return root, right

else:

# Отсоединяем левое поддерево

left, root.left = root.left, None

set\_parent(left, None)

# Возвращаем отсоединенное левое поддерево и корневое дерево (без левого поддерева)

return left, root

Чтобы вставить очередной ключ, достаточно вызвать split по нему, а затем сделать новую вершину-корень, у которой поддеревьями будет результат split-а.

def insert(root, key):

# Используем функцию split для разделения дерева на два поддерева:

# одно с ключами меньше key и другое с ключами больше или равными key

left, right = split(root, key)

# Создаём новый узел с ключом key, где left и right являются его потомками

root = Node(key, left, right)

# Обновляем родительские ссылки для левого и правого поддеревьев

keep\_parent(root)

# Возвращаем новый узел, который теперь является корнем дерева

return root

Процедура merge получает на вход два дерева: левое left и правое right. Для корректной работы, ключи дерева left должны быть меньше ключей дерева right. Здесь мы берем вершину с наименьшим ключом правого дерева right и тянем ее вверх. После этого в качестве левого поддерева присоединяем дерево left.

def merge(left, right):

# Проверяем, пусто ли правое дерево

if right == None:

# Если правое дерево пусто, просто возвращаем левое дерево

return left

# Проверяем, пусто ли левое дерево

if left == None:

# Если левое дерево пусто, просто возвращаем правое дерево

return right

# Находим узел в правом дереве с минимальным ключом, большим или равным ключу в корне левого дерева

right = find(right, left.key)

# Устанавливаем левое дерево в качестве левого поддерева для найденного узла в правом дереве

right.left, left.parent = left, right

# Возвращаем объединенное дерево, корнем которого теперь является модифицированный узел из правого дерева

return right

Для того, чтобы удалить вершину, нужно поднять ее вверх, а потом слить ее левые и правые поддеревья.

def remove(root, key):

# Находим узел с заданным ключом и перемещаем его в корень дерева

root = find(root, key)

# Отсоединяем левое и правое поддеревья узла от его родителя

set\_parent(root.left, None)

set\_parent(root.right, None)

# Объединяем левое и правое поддеревья и возвращаем новое дерево

return merge(root.left, root.right)

Чтобы splay-дерево поддерживало повторяющиеся ключи, можно поступить двумя способами. Нужно либо каждому ключу сопоставить список, хранящий нужную доп. информацию, либо реализовать процедуру find так, чтобы она возвращала первую в порядке обхода LUR вершину с ключом, большим либо равным заданного.

# Сложность алгоритмов

Каждая из функций в реализации Splay-дерева имеет определенную сложность в терминах Big O, которая характеризует её производительность:

1. **rotate (Поворот)**: O(1). Поворот в Splay-дереве — это базовая операция, которая включает только несколько присваиваний и не зависит от размера дерева.
2. **splay (Splay-операция)**: Амортизированно O(log n). Операция Splay может потребовать нескольких поворотов, но благодаря свойству "амортизации", среднее время работы остаётся логарифмическим относительно размера дерева.
3. **find (Поиск)**: Амортизированно O(log n). Поиск включает в себя движение вниз по дереву, что обычно занимает логарифмическое время, и последующую операцию Splay.
4. **split (Разделение)**: Амортизированно O(log n). Разделение дерева основано на поиске и нескольких манипуляциях с указателями, что делает его эффективным по времени.
5. **insert (Вставка)**: Амортизированно O(log n). Вставка включает разделение дерева и создание нового узла, что в среднем занимает логарифмическое время.
6. **merge (Объединение)**: Амортизированно O(log n). Объединение двух деревьев включает поиск и несколько базовых операций с указателями.
7. **remove (Удаление)**: Амортизированно O(log n). Удаление включает поиск удаляемого узла, его удаление и объединение оставшихся поддеревьев.

В целом, операции в Splay-дереве обладают хорошей производительностью благодаря свойству амортизации, что делает их эффективными для большинства операций в среднем случае, даже если в худшем случае некоторые операции могут достигать O(n).