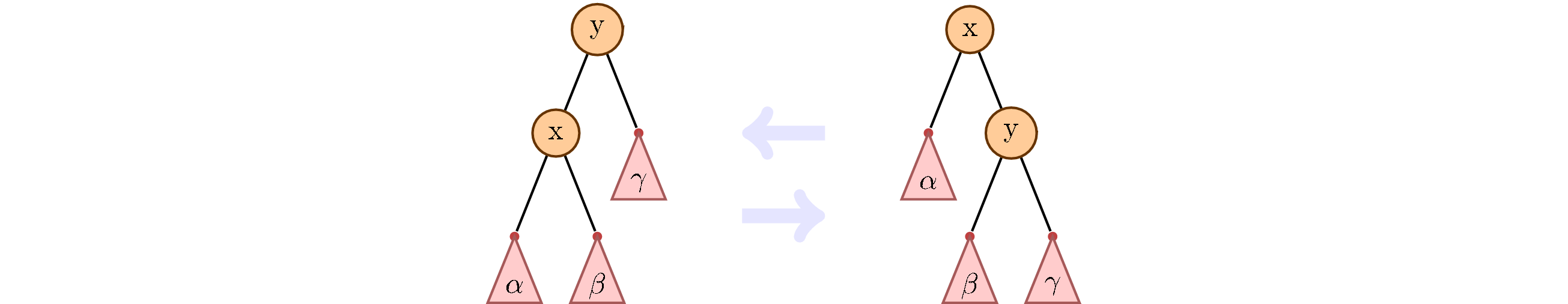
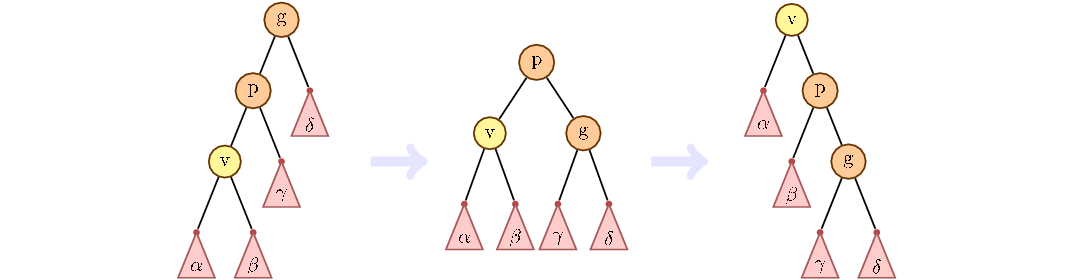
Splay-дерево – это самобалансирующееся дерево бинарного поиска. Основная особенность Splay-дерева заключается в операции Splay, которая перемещает доступный узел в корень дерева с помощью последовательности поворотов

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.

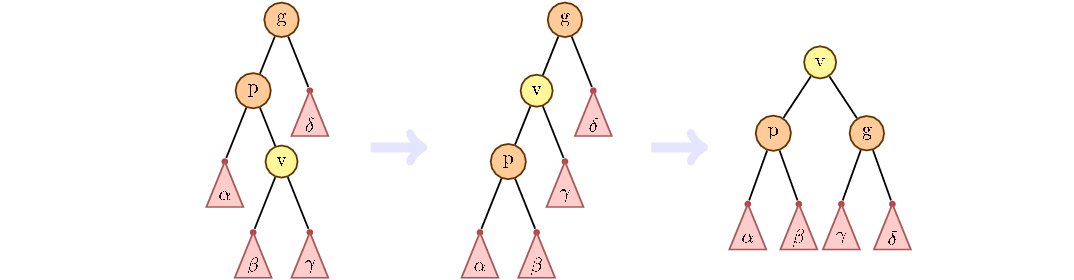
Основной метод move-to-root. После обращения к любой вершине, она поднимается в корень. Подъем реализуется через повороты вершин. За один поворот, можно поменять местами родителя с ребенком, как показано на рисунке ниже.



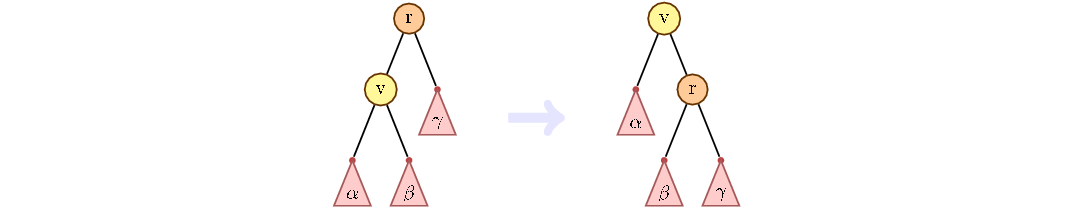
Хитрость splay-дерева в том, что при продвижении вершины вверх, расстояние до корня сокращается не только для поднимаемой вершины, но и для всех ее потомков в текущих поддеревьях. Для этого используется техника zig-zig и zig-zag поворотов.  
  
Основная идея zig-zig и zig-zag поворотов, рассмотреть путь 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).

*Source: https://habr.com/ru/companies/JetBrains-education/articles/210296/*