Сириус
IT-Колледж

Лекция 12. Деревья

Поиск, вставка, удаление, поиск следующего и предыдущего элемента за время, пропорциональное высоте. Декартово дерево: split, merge, реализация операций вставки и удаления через split и merge. Использование неявного ключа, rope.

Сириус
IT-Колледж

Поиск элемента за время, пропорциональное высоте

```
def search(node, key):
    if not node or node.data == key:
        return node

    if node.data > key:
        return search(node.left, key)

    return search(node.right, key)
```

Сириус IT-Колледж

Поиск следующего и предыдущего элемента за время, пропорциональное высоте

Найти элемент со следующим значением ключа, относительно ключа некоторого узла.

```
def searchNext(node, key):
    if not node:
        return None

    if  node.data == key:
        if node.right:
            current = node.right
            while current.left:
                current = current.left
            return current
        return None

    if node.data > key:
        return search(node.left, key)

    return search(node.right, key)
```

Поиск предыдущего элемента осуществляется аналогично.

Сириус
IT-Колледж

Вставка элемента за время, пропорциональное высоте

```
def insert(node, key):
    if node is None:
        return Node(key)

    if key < node.data:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)

    return node
```

Сириус
IT-Колледж

Удаление элемента за время, пропорциональное высоте

```
def deleteNode(root, key):
    if root is None:
        return root

    if key < root.data:
        root.left = deleteNode(root.left, key)
    elif(key > root.data):
        root.right = deleteNode(root.right, key)
    else:
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        temp = searchPrev(root, key)
        root.data = temp.data
        root.right = deleteNode(root.right, temp.data)
    return root
```

Сириус IT-Колледж

Декартово дерево

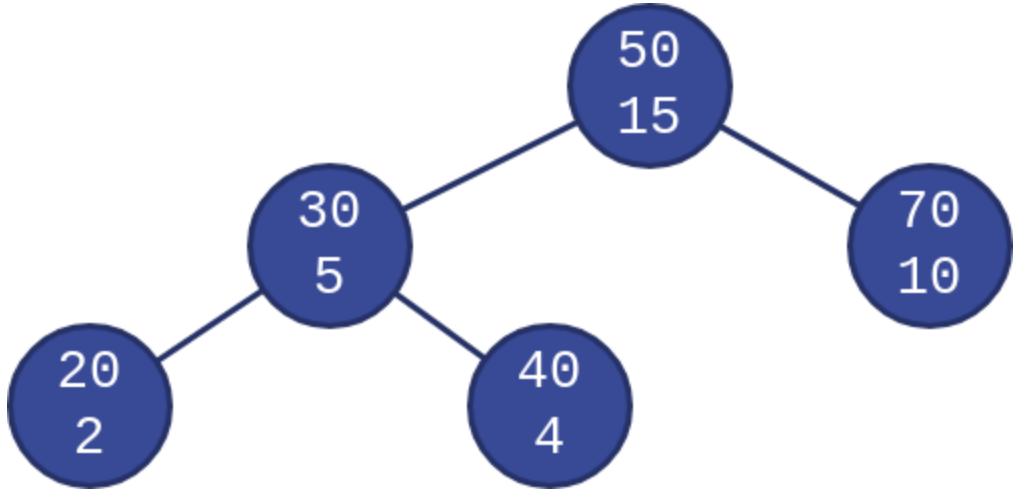
Декартово дерево или дерамида (Treap) — это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree + heap) и дерамида (дерево + пирамида), также существует название курево (куча + дерево).

Более строго, это бинарное дерево, в узлах которого хранятся пары (x, y), где x — это ключ, а y — это приоритет. Также оно является двоичным деревом поиска по x и пирамидой по y. Предполагая, что все x и все y являются различными, получаем, что если некоторый элемент дерева содержит (x_0, y_0), то у всех элементов в левом поддереве $x < x_0$, у всех элементов в правом поддереве $x > x_0$, а также и в левом, и в правом поддереве имеем: $y < y_0$.

Дерамиды были предложены Сиделем и Арагон в 1996 г.

Сириус
IT-Колледж

Декартово дерево




```
class Node:
    def __init__(self, value, prior):
        self.value = value
        self.prior = prior
        self.left = None
        self.right = None
```

Сириус IT-Колледж

Декартово дерево: split

Операция **split** (разрезать) позволяет сделать следующее: разрезать исходное дерево T по ключу k. Возвращать она будет такую пару деревьев $\langle T_1, T_2 \rangle$, что в дереве T_1 ключи меньше k, а в дереве T_2 все остальные: $split(T, k) \to \langle T_1, T_2 \rangle$.

Эта операция устроена следующим образом.

Рассмотрим случай, в котором требуется разрезать дерево по ключу, большему ключа корня. Посмотрим, как будут устроены результирующие деревья T_1 и T_2:

- T_1: левое поддерево T_1 совпадёт с левым поддеревом T. Для нахождения правого поддерева T_1 , нужно разрезать правое поддерево T на T_1^R и T_2^R по ключу k и взять T_1^R.
- T_2 совпадёт с T_2^R.

Случай, в котором требуется разрезать дерево по ключу, меньше либо равному ключа в корне, рассматривается симметрично.

Сириус IT-Колледж

Декартово дерево: split

```
def split(root, val):
    if root is None:
        return (None, None)
    elif root.value is None:
        return (None, None)
    else:
        if value < root.value:
            left, root.left = split(root.left, value)
            return (left, root)
        else:
            root.right, right = split(root.right, value)
            return (root, right)
```

Сириус IT-Колледж

Декартово дерево: merge

Рассмотрим вторую операцию с декартовыми деревьями — **merge** (слить).

С помощью этой операции можно слить два декартовых дерева в одно. Причём, все ключи в первом(левом) дереве должны быть меньше, чем ключи во втором(правом). В результате получается дерево, в котором есть все ключи из первого и второго деревьев: $merge(T_1, T_2) \to T$

Рассмотрим принцип работы этой операции. Пусть нужно слить деревья T_1 и T_2. Тогда, очевидно, у результирующего дерева T есть корень. Корнем станет вершина из T_1 или T_2 с наибольшим приоритетом y. Но вершина с самым большим y из всех вершин деревьев T_1 и T_2 может быть только либо корнем T_1, либо корнем T_2. Рассмотрим случай, в котором корень T_1 имеет больший y, чем корень T_2. Случай, в котором корень T_2 имеет больший y, чем корень T_1, симметричен этому.

Если y корня T_1 больше y корня T_2, то он и будет являться корнем. Тогда левое поддерево T совпадёт с левым поддеревом T_1. Справа же нужно подвесить объединение правого поддерева T_1 и дерева T_2.

Сириус
IT-Колледж

Декартово дерево: merge

```
def merge(left, right):
    if (not left) or (not right):
        return left or right
    elif left.prior < right.prior:
        left.right = merge(left.right, right)
        return left
    else:
        right.left = merge(left, right.left)
        return right
```

Сириус IT-Колледж

Декартово дерево: insert

Операция $insert(T, k)$ добавляет в дерево T элемент k, где $k.x$ — ключ, а $k.y$ — приоритет.

Представим что элемент k, это декартово дерево из одного элемента, и для того чтобы его добавить в наше декартово дерево T, очевидно, нам нужно их слить. Но T может содержать ключи как меньше, так и больше ключа $k.x$, поэтому сначала нужно разрезать T по ключу $k.x$.

- Реализация №1
  i. Разобьём наше дерево по ключу, который мы хотим добавить, то есть $split(T, k.x) \to \langle T_1, T_2 \rangle$.
  ii. Сливаем первое дерево с новым элементом, то есть $merge(T_1, k) \to T_1$.
  iii. Сливаем получившиеся дерево со вторым, то есть $merge(T_1, T_2) \to T$.

Сириус IT-Колледж

Декартово дерево: insert

- Реализация №2
  i. Сначала спускаемся по дереву (как в обычном бинарном дереве поиска по $k.x$), но останавливаемся на первом элементе, в котором значение приоритета оказалось меньше $k.y$.
  ii. Теперь вызываем $split(T, k.x) \to \langle T_1, T_2 \rangle$ от найденного элемента (от элемента вместе со всем его поддеревом)
  iii. Полученные T_1 и T_2 записываем в качестве левого и правого сына добавляемого элемента.
  iv. Полученное дерево ставим на место элемента, найденного в первом пункте.

В первой реализации два раза используется $merge$, а во второй реализации слияние вообще не используется.

Сириус IT-Колледж

Декартово дерево: remove

Операция $remove(T, x)$ удаляет из дерева T элемент с ключом x.

- Реализация №1
  i. Разобьём наше дерево по ключу, который мы хотим удалить, то есть $split(T, k.\, x) \to \langle T_1, T_2 \rangle$.
  ii. Теперь отделяем от первого дерева элемент x, то есть самого левого ребёнка дерева T_2.
  iii. Сливаем первое дерево со вторым, то есть $merge(T_1, T_2) \to T$.
- Реализация №2
  i. Спускаемся по дереву (как в обычном бинарном дереве поиска по x), и ищем удаляемый элемент.
  ii. Найдя элемент, вызываем $merge$ его левого и правого сыновей
  iii. Результат процедуры $merge$ ставим на место удаляемого элемента.

В первой реализации один раз используется $split$, а во второй реализации разрезание вообще не используется.

Сириус
IT-Колледж

Декартово дерево: реализация операций вставки и удаления через split и merge

```
def insert(root, value):
    node = Node(value)
    left, right = split(root, value)
    return merge(merge(left, node), right)


def remove(root, value):
    left, right = split(root, value - 1)
    _, right = split(right, value)
    return merge(left, right)
```

Сириус
IT-Колледж

Использование неявного ключа

В стандартной реализации структуру данных динамический массив мы умеем добавлять элемент в конец вектора, узнавать значение элемента, стоящего на определенной позиции, изменять элемент по номеру и удалять последний элемент. Предположим, что нам необходима структура данных с вышеуказанными свойствами, а также с операциями: добавить элемент в любое место (с соответствующим изменением нумерации элементов) и удалить любой элемент (также с соответствующим изменением нумерации). Такую структуру можно реализовать на базе декартового дерева, результат часто называют **декартово дерево по неявному ключу (Treap with implicit key)**.

Сириус
IT-Колледж

Использование неявного ключа

При реализации декартова дерева по неявному ключу модифицируем эту структуру. А именно, оставим в нем только приоритет Y, а вместо ключа X будем использовать следующую величину: **количество элементов в нашей структуре, находящихся левее нашего элемента**. Иначе говоря, будем считать ключом порядковый номер нашего элемента в дереве, уменьшенный на единицу.

Заметим, что при этом сохранится структура двоичного дерева поиска по этому ключу (то есть модифицированное декартово дерево так и останется декартовым деревом). Однако, с этим подходом появляется проблема: операции добавления и удаления элемента могут поменять нумерацию, и при наивной реализации на изменение всех ключей потребуется $O(n)$ времени, где n — количество элементов в дереве.

Сириус
IT-Колледж

Использование неявного ключа

Решается эта проблема довольно просто. Основная идея заключается в том, что такой ключ X сам по себе нигде не хранится. Вместо него будем хранить вспомогательную величину C: **количество вершин в поддереве нашей вершины** (в поддерево включается и сама вершина). Обратим внимание, что все операции с обычным декартовым деревом делались сверху. Также заметим, что если по пути от корня до некой вершины просуммировать все такие величины в левых поддеревьях, в которые мы не пошли, увеличенные на единицу, то придя в саму вершину и добавив к этой величине количество элементов в её левом поддереве, мы получим как раз ее ключ X.

Сириус IT-Колледж

Использование неявного ключа: split

Пусть процедура $split$ запущена в корне дерева с требованием отрезать от дерева k вершин. Также известно, что в левом поддереве вершины находится l вершин, а в правом r. Рассмотрим все возможные случаи:

- $l \geqslant k$. В этом случае нужно рекурсивно запустить процедуру $split$ от левого сына с тем же параметром k. При этом новым левым сыном корня станет правая часть ответа рекурсивной процедуры, а правой частью ответа станет корень.
- $l < k$ Случай симметричен предыдущему. Рекурсивно запустим процедуру $split$ от правого сына с параметром $k - l - 1$. При этом новым правым сыном корня станет левая часть ответа рекурсивной процедуры, а левой частью ответа станет корень.

Сириус
IT-Колледж

Использование неявного ключа: merge

Посмотрим любую из реализаций процедуры $merge$. Заметим, что в ней программа ни разу не обращается к ключу X. Поэтому реализация процедуры $merge$ для декартова дерева по неявному ключу вообще не будет отличаться от реализации той же процедуры в обычном декартовом дереве.

Сириус IT-Колледж

Использование неявного ключа: применение

Таким образом, описана структура, от которой можно отрезать слева часть произвольной длины и слить две любые части в одну в нужном порядке. Теперь мы имеем возможность:

- вставить элемент в любое место (отрежем нужное количество элементов слева, сольем левое дерево с деревом из одного добавленного элемента и результат — с правым деревом),
- переставить любой кусок массива куда угодно (сделаем нужные разрезы и слияния в правильном порядке),
- совершать групповые операции с элементами. Вспомним реализацию таких операций в дереве отрезков и поймем, что ничего не помешает нам сделать то же самое с описанным деревом. В групповые операции включается, естественно, и взятие функции от отрезка,
- сделав на одном исходном массиве два дерева из элементов разной четности, можно решить задачу про смену мест четных и нечетных на отрезке,

Rope

Rope (веревка) — структура данных для хранения строки, представляющая из себя двоичное сбалансированное дерево и позволяющая делать операции вставки, удаления и конкатенации с логарифмической асимптотикой.

Иногда при использовании строк нам нужны следующие свойства:

- Операции, которые часто используются на строках, должны быть более эффективными. Например: конкатенация, взятие подстроки.
- Также эти операции должны эффективно работать и с длинными строками. Не должно быть прямой зависимости от длины строк.
- Персистентность. Иногда необходимо при изменении строки сохранить ее состояние перед изменением и вернуться к нему, если необходимо.

В данном случае Rope удовлетворяет всем этим свойствам.

Сириус IT-Колледж

Rope

Для хранения **Rope** создадим структуру, похожую на декартово дерево по неявному ключу. В каждом листе будем хранить последовательную часть строки и ее длину, а в промежуточных вершинах будем хранить сумму длин всех листьев в поддереве. Изначально дерево состоит из одной вершины — самой строки. Используя информацию в промежуточных вершинах, можно получать символы строки по индексу.

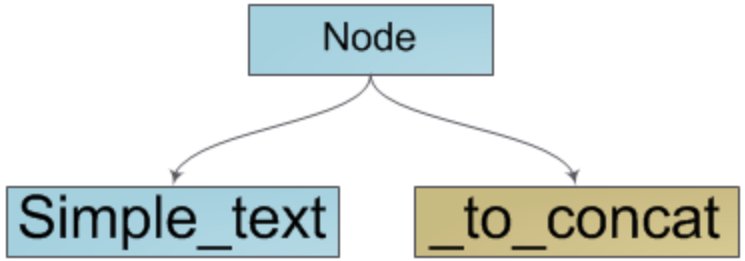
Также заметим, что для отметки листа не обязательно хранить дополнительную информацию: все внутренние вершины имеют ровно двух детей, а листы — ни одного. Поэтому для проверки вершины на то, что она является листом, достаточно проверить, есть ли у неё дети.

Сириус IT-Колледж

Rope: merge

Когда приходит запрос на конкатенацию с другой строкой, мы объединяем оба дерева, создав новый корень и подвесив к нему обе строки. Пример результата конкатенации двух строк:

Результат конкатенации двух строк.




```
def merge(n1, n2):
    return Node(n1, n2, n1.w + n2.w)
```

Асимптотика выполнения операции конкатенации двух строк, очевидно, $O(1)$.

Сириус
IT-Колледж

Rope: получение символа по индексу

Чтобы получить символ по некоторому индексу i, будем спускаться по дереву из корня, используя веса, записанные в вершинах, чтобы определить, в какое поддерево пойти из текущей вершины. Алгоритм выглядит следующим образом:

- Текущая вершина — не лист, тогда возможно два варианта:
  - Вес левого поддерева больше либо равен i, тогда идем в левое поддерево.
  - Иначе идем в правое поддерево и ищем там $i - w$ символ, где w вес левого поддерева.
- Текущая вершина — лист, тогда в этом листе хранится ответ; необходимо взять символ с соответствующим номером у строки, которая там хранится.

Сириус
IT-Колледж

Rope: получение символа по индексу

```
def get(i, node):
    if node.left:
        if node.left.w >= i:
            return get(i, node.left)
        else:
            return get(i - node.left.w, node.right)
    else:
        return node.s[i]
```

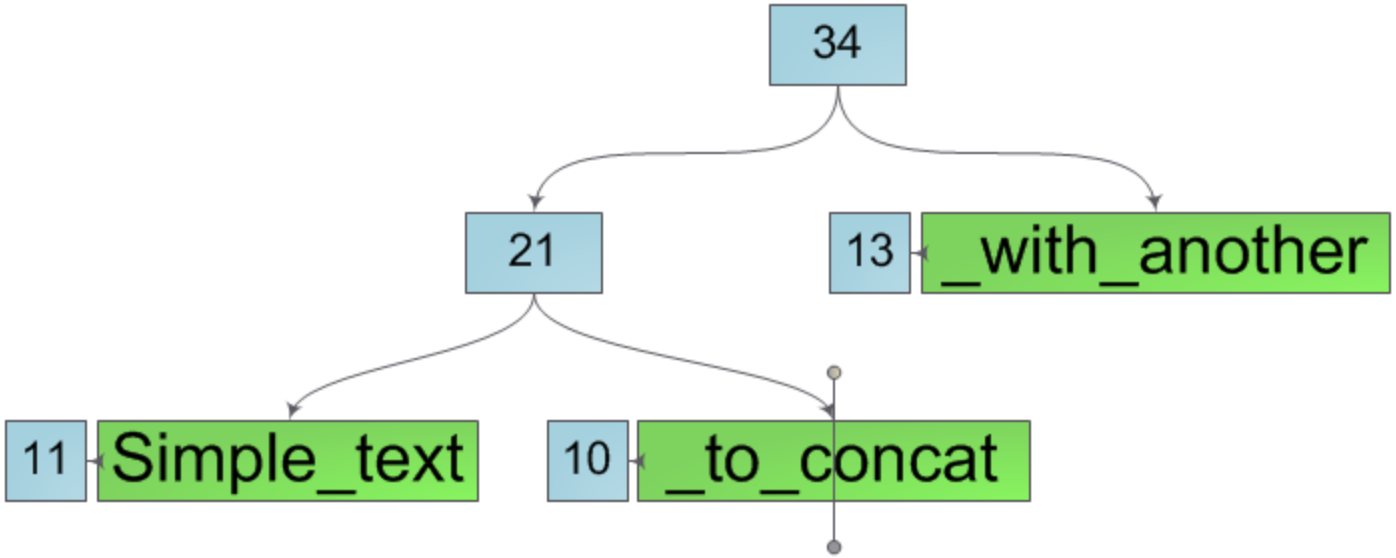
Асимптотика выполнения одного такого запроса, очевидно, $O(h)$, где h — высота дерева.

Сириус
IT-Колледж

Rope: split

Чтобы разбить строку на две по некоторому индексу i, необходимо, спускаясь по дереву (аналогично операции get), каждую вершину на пути поделить на две, каждая из которых будет соответствовать одной из половинок строк, при этом необходимо после деления пересчитать вес этих вершин.

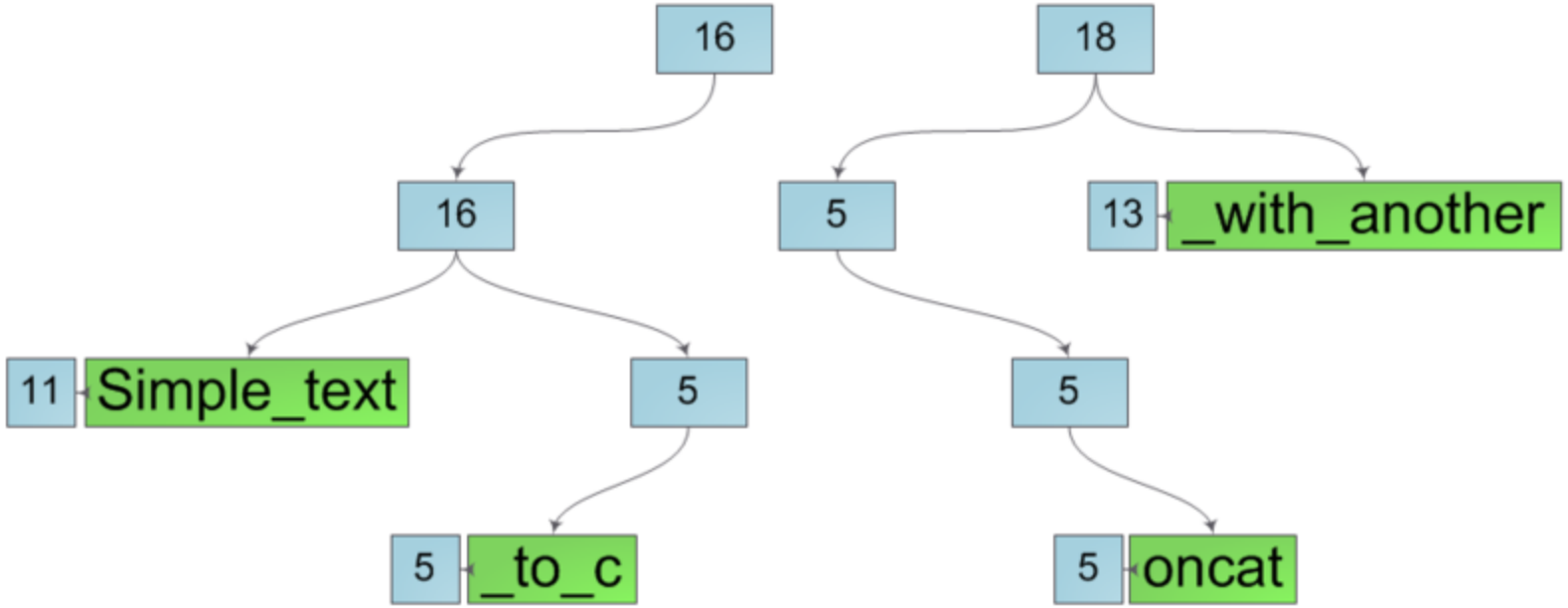
Пускай дано дерево:



Сириус
IT-Колледж

Rope: split

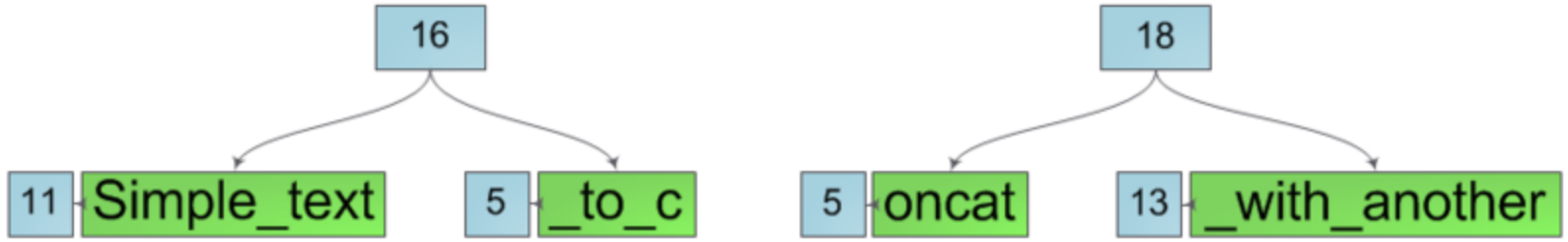
Тогда результатом выполнения операции $split$ по индексу 16 будет:



Сириус
IT-Колледж

Rope: split

Заметим, что появляются лишние вершины, у которых есть только один потомок. От них можно легко избавиться, просто заменив их на этого потомка. Тогда результатом той же операции $split$ будет:



Сириус IT-Колледж

Rope: split

```
def split(node, i):
    if node.left:
        if node.left.w >= i:
            res = split(node.left, i)
            tree1 = res[0]
            tree2.left = res[1]
            tree2.right = node.right
            tree2.w = tree2.left.w + tree2.right.w
        else:
            res = split(node.right, i - node.left.w)
            tree1.left = node.left
            tree1.right = res[0]
            tree1.w = tree1.left.w + tree1.right.w
            tree2 = res[1]
    else:
        tree1.s = node.s[:i+1]
        tree2.s = node.s.[i+1:]
        tree1.w = i
        tree2.w = len(node.s) - i
    return tree1, tree2
```

Сириус IT-Колледж

Rope: операции удаления и вставки

Нетрудно понять, что имея операции $merge$ и $split$, можно легко через них выразить операции $delete$ и $insert$ по аналогии с другими деревьями поиска.

Операция $delete$ удаляет из строки подстроку, начиная с индекса $beginIndex$ и заканчивая (не включая) индексом $endIndex$.

```
def delete(node, beginIndex, endIndex):
    tree1, tree2 = split(node, beginIndex)
    tree3 = split(tree2, endIndex - beginIndex)[1]
    return merge(tree1, tree3)
```

Сириус
IT-Колледж

Rope: операции удаления и вставки

Операция $insert$вставляет данную строку s в исходную, начиная с позиции $insertIndex$.

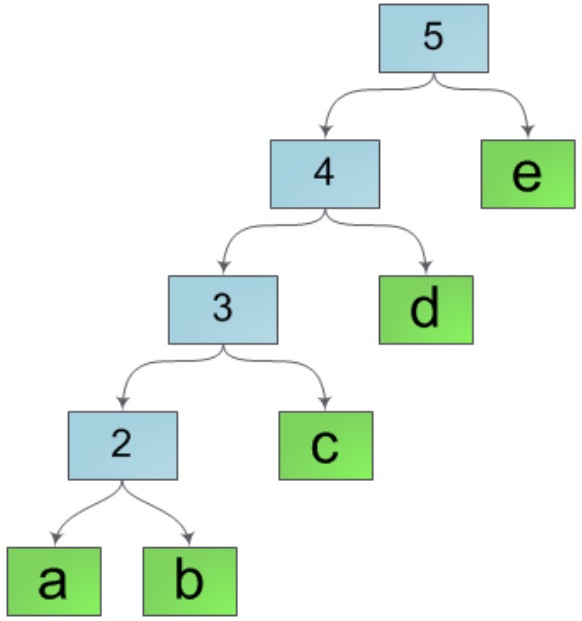
```
def insert(node, insertIndex, s):
    tree1, tree3 = split(node, insertIndex)
    tree2 = Node(s)
    return merge(merge(tree1, tree2), tree3)
```

Так как данные операции используют только $split$ и $merge$, то асимптотика времени их работы — $O(h)$, где h — высота дерева.

Сириус IT-Колледж

Rope: балансировка

Для того, чтобы дерево не превращалось в бамбук:



Предлагается хранить его как АВЛ-дерево и делать соответствующие балансировки. Тогда, так как высота АВЛ-дерева $h = \log n$, то асимптотика операций get,$split$,$delete$, $insert$,$merge$ будет равна $O(\log n)$, где n — количество сконкатенированных строк.