Задача A. Следующий

В этой задаче требовалось написать какое-то из балансированных деревьев, после чего сделать по нему спуск. Самым простым вариантом было построить декартово дерево, сделать в нем split таким образом, чтобы в правом разрезе оказались все ключи не меньше, чем i. В таком разрезе нам нужно найти минимальный элемент (если дерево оказалось пустым, ответом будет -1). А как найти минимальный элемент? Из свойств декартова дерева левый сын вершины v всегда имеет меньший ключ, чем v. Значит, надо просто идти по левым сыновьям до самого левого.

```
def find_min(v):
    if v.left is None:
        return v.x
    return find_min(v.left)

def add(v, key):
    l, r = split(v, key)
    node = Node(key, random())
    new_root = merge(l, node)
    new_root = merge(node, r)
    return new_root


def ans(root, val):
    l, r = split(root, val)
    if r is None:
        ans = -1
        last = -1
    else:
        last = find_min(sp.second)
        ans = last

    merge(l, r)
```

Задача B. Два-Три-Де... Дерево

Задача требовала написать добавление в 2-3 дерево.

Чтобы было чуть более понятно, давайте разобьем процесс на две части: «добавление листа» и «починка инварианта».

Для того, чтобы добавить лист, мы делаем поиск в 2-3 дереве и спускаемся до первого листа, со значением, большим x. Для этого мы храним максимум в каждом поддереве, и если максимум слева больше x, рекурсивно добавляем влево. Иначе, если центр есть и максимум в нем больше x, рекурсивно добавляем в центр; в третьем случае добавляем вправо.

Когда мы спустились до самого конца, мы можем создать новый лист и добавить его к тому же предку, что и первый лист больше x. Единственная плохая возможная ситуация в этот момент - у вершины появилось 4 сына. Тогда ее надо раздвоить, добавить еще одного сына предку и передать "починку"на уровень выше.

```
def fix(self) -> None:
    if (len(self.children) <= 3):
        return

    part1, part2 = self.children[:2], self.children[2:]
    self.children = part1
    newNode = Node()
    newNode.children = part2
```

```
    for i in part2:
        i.parent = newNode

    if self.parent is not None:
        newNode.parent = self.parent
        self.parent.children.append(newNode)
        self.parent.children.sort(key=lambda node: node.max())
        self.parent.fix()
    else:
        newRoot = Node()
        newRoot.children = [self, newNode]
        newRoot.children.sort(key=lambda node: node.max())
        self.parent = newRoot
        newNode.parent = newRoot

def find_node_to_add(x):
    node = root
    while len(node.children) > 0:
        for i in node.children:
            if i.max() > x:
                node = i
                break
        else:
            node = node.children[-1]
    return node.parent
```

Задача C. K-ый максимум

Пишем декартач (или любое другое дерево поиска), храним в каждой вершине размер поддерева. При поиске k-ой статистики смотрим на размер правого поддерева, если он больше k, то идём в него, если ровно k, то текущая вершина — ответ, если меньше k, то идём в левого сына и уменьшаем k на размер правого дерева плюс один.

Основной сложностью этой задачи было поддерживать размеры в поддереве. Для этого можно хранить счетчик, и каждый раз, когда функция merge или split модифицирует вершину, пересчитывать у нее этот счетчик

```
def merge(l, r):
    # ...
    if l.priority > r.priority:
        r.l = merge(l, r.l)
        r.update_count()
        return r

def split(node, val):
    if node is None:
        return None, None
    if node.val < val:
        node.r, r = Treap.split(node.r, val)
        node.update_count()
        r.update_count()
        return node, r

def update_count(self):
    if self is None:
```

```
        return
    self.count = 1
    if self.l:
        self.count += self.l.count
    if self.r:
        self.count += self.r.count

def get_max(self, k):
    if self is None:
        return -1
    r_count = 0
    if self.r:
        r_count = self.r.count
    if r_count == k:
        return self.val
    elif r_count > k and self.r:
        return self.r.get_max(k)
    elif r_count < k and self.l:
        return self.l.get_max(k - r_count - 1)
    return -1
```

Задача D. И снова сумма...

В этой задаче требовалось сделать декартово дерево, которое бы хранило сумму в поддереве, а на запрос суммы на отрезке с помощью двух split-ов вырезала нужный подотрезок и брала в нем предподсчитанную сумму.

В целом, реализация очень похожа на задачу C:

```
def sum(node):
    if node is None:
        return 0
    return node.x + sum(node.l) + sum(node.r)

def get(root, l, r):
    l_tree, m_tree, r_tree = None, None, None
    l_tree, r_tree = split(root, l)
    m_tree, r_tree = split(r_tree, r+1)
    res = sum(m_tree)
    root = merge(l_tree, merge(m_tree, r_tree))
    return root, res
```