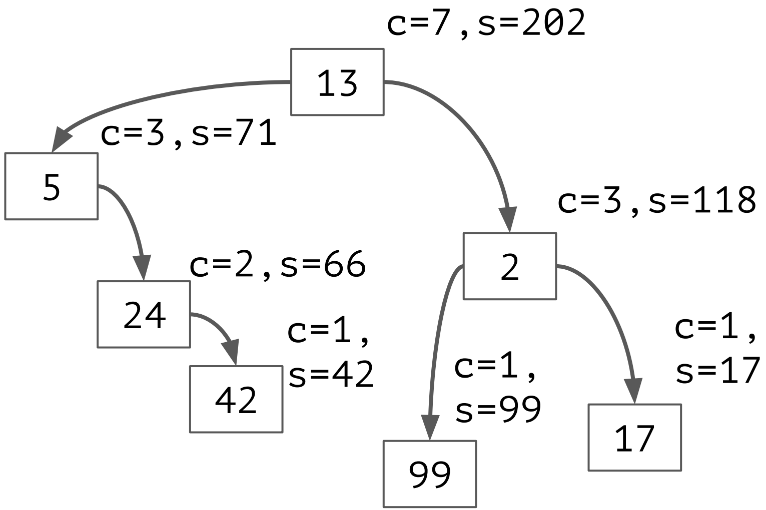
Мини-задача #42 (2 балла)

Реализовать неявное дерево с поддержкой функции sum(from, to), которая возвращает сумму элементов массива от from до to включительно.

Не забывайте о тестах для вашего решения!

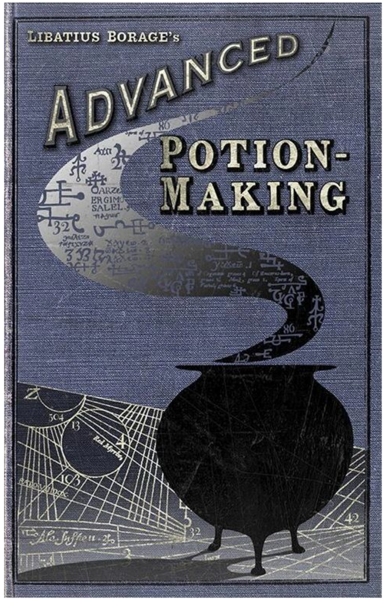


Алгоритмы и структуры данных

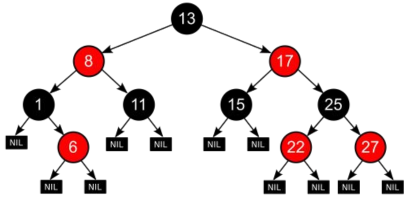
Декартовы деревья, неявные декартовы деревья

Что будем изучать?

1. Жадные алгоритмы
2. Динамическое программирование
3. Необычные структуры данных (новые виды деревьев, пирамид и не только)
4. … and beyond



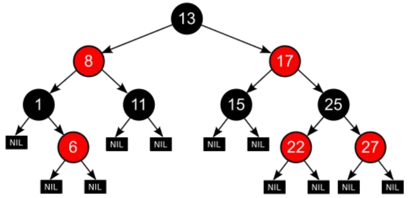
Сбалансированные деревья поиска



Операции:

```
1.  find(value)      -> O(logN)
2.  select(i)        -> O(logN)
3.  min/max          -> O(logN)
4.  pred/succ(ptr)   -> O(logN)
5.  rank(value)      -> O(logN)
6.  вывод в пор.     -> O(N)
    возрастания
-------------------------------

7.  insert(value)    -> O(logN)
8.  remove(value)    -> O(logN)
```

Сбалансированные деревья поиска

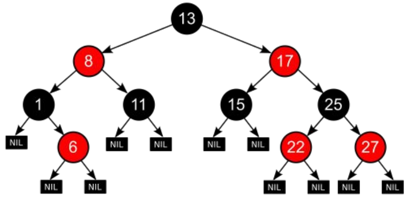


Операции:

```
1.  find(value)     -> O(logN)
2.  select(i)       -> O(logN)
3.  min/max         -> O(logN)
4.  pred/succ(ptr)  -> O(logN)
5.  rank(value)     -> O(logN)
6.  вывод в пор.    -> O(N)
    возрастания
-------------------------------

7.  insert(value)   -> O(logN)
8.  remove(value)   -> O(logN)
```

Пирамиды лучше здесь (по константам или даже по асимптотике)

Сбалансированные деревья поиска



Операции:

```
1.  find(value)     -> O(logN)
2.  select(i)       -> O(logN)
3.  min/max         -> O(logN)
4.  pred/succ(ptr)  -> O(logN)
5.  rank(value)     -> O(logN)
6.  вывод в пор.    -> O(N)
    возрастания
-------------------------------

7.  insert(value)   -> O(logN)
8.  remove(value)   -> O(logN)
```

Пирамиды лучше здесь (по константам или даже по асимптотике)

А еще АВЛ и красно-черные деревья сложно и неприятно писать! 😠

Декартово дерево

Пусть есть уникальный набор ключей.

Декартово дерево

Пусть есть уникальный набор ключей. Каждому ключу сопоставим уникальное случайное число и назовем его приоритетом.

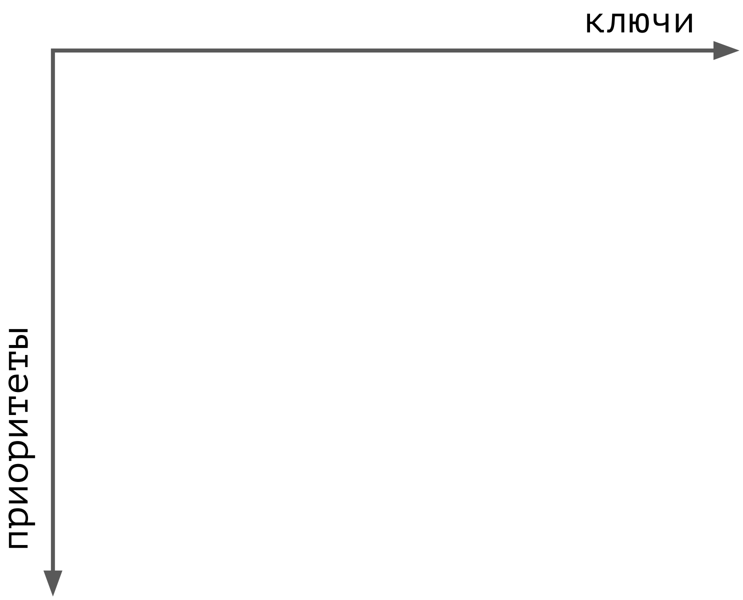
Декартово дерево

Пусть есть уникальный набор ключей. Каждому ключу сопоставим уникальное случайное число и назовем его приоритетом.

Декартово дерево — структура данных, являющаяся BST по ключам и минимальной пирамидой по приоритетам.

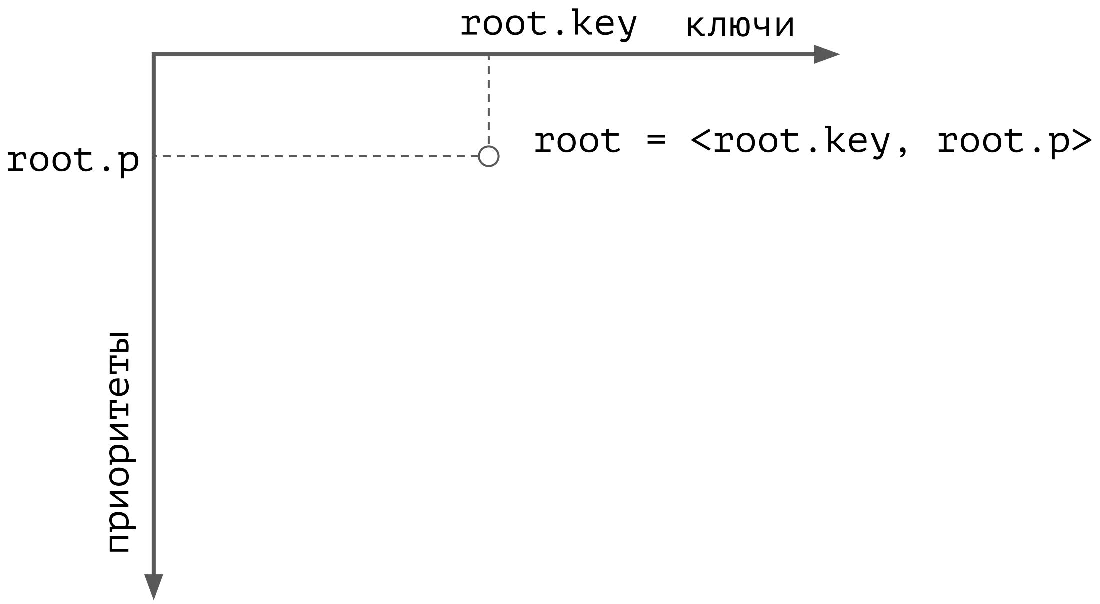
Декартово дерево

Декартово дерево — структура данных, являющаяся BST по ключам и минимальной пирамидой по приоритетам.



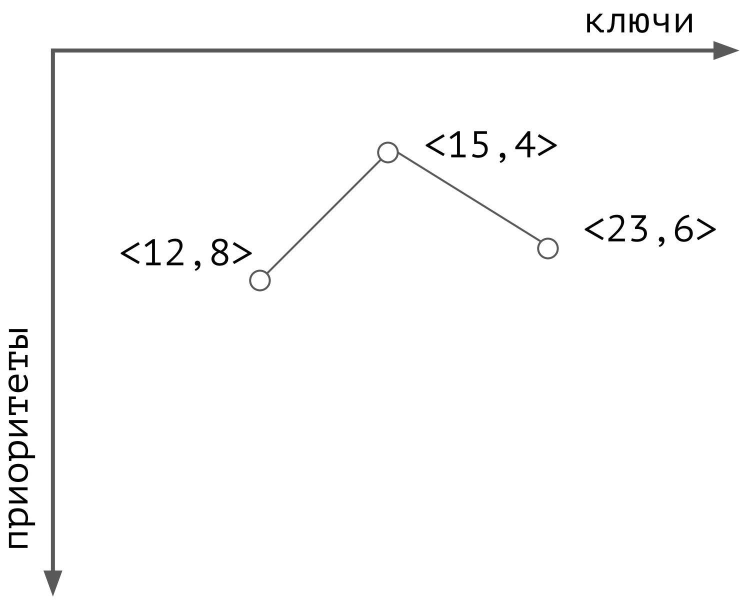
Декартово дерево

Декартово дерево — структура данных, являющаяся BST по ключам и минимальной пирамидой по приоритетам.



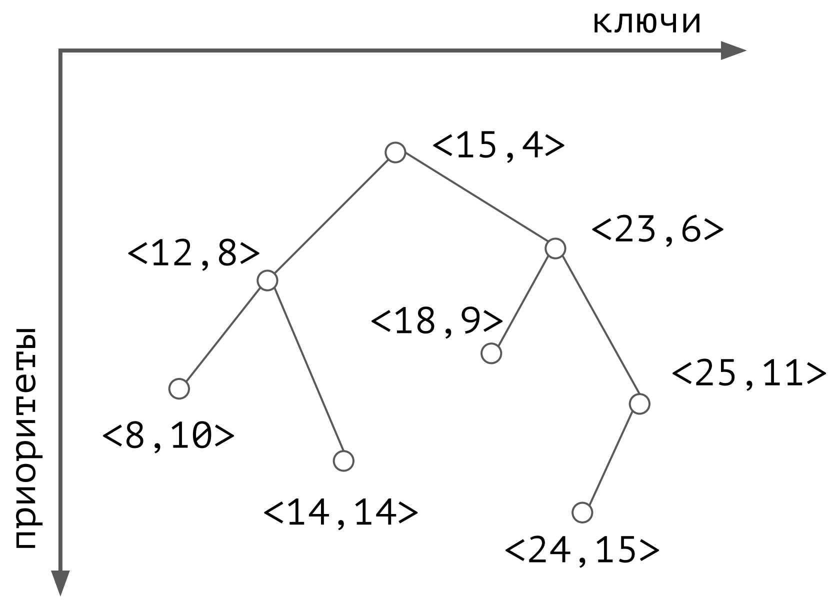
Декартово дерево

Декартово дерево — структура данных, являющаяся BST по ключам и минимальной пирамидой по приоритетам.



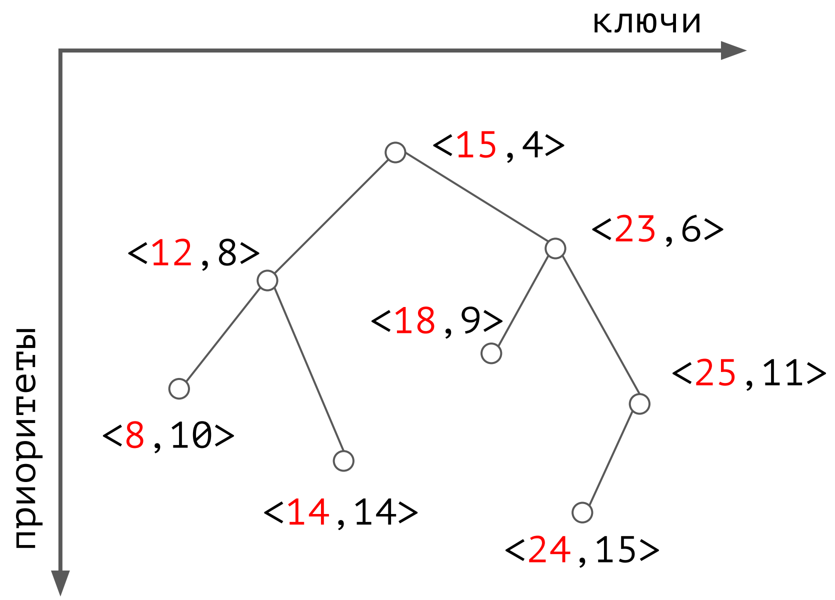
Декартово дерево

Декартово дерево — структура данных, являющаяся BST по ключам и минимальной пирамидой по приоритетам.



Декартово дерево

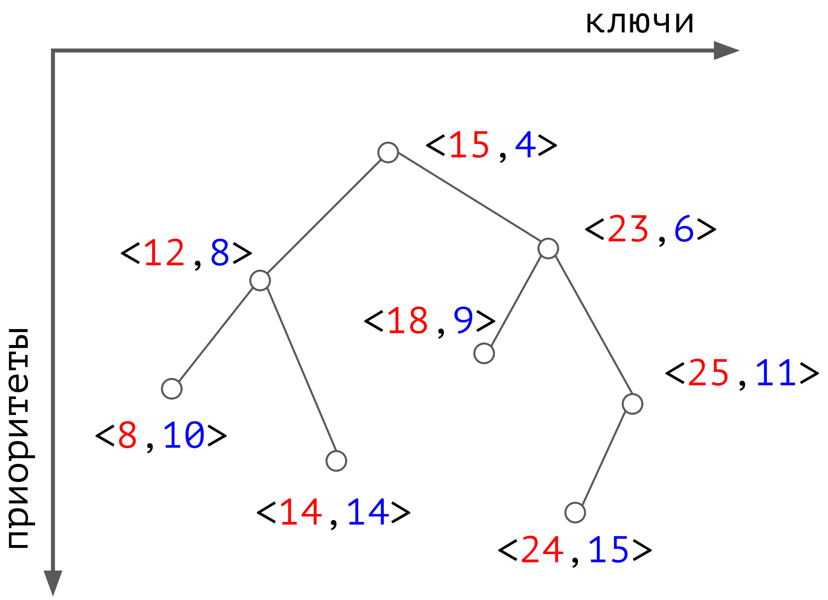
Декартово дерево — структура данных, являющаяся BST по ключам и минимальной пирамидой по приоритетам.



В левом поддереве все ключи меньше, в правом все ключи больше, ведь это BST.

Декартово дерево

Декартово дерево — структура данных, являющаяся BST по ключам и минимальной пирамидой по приоритетам.

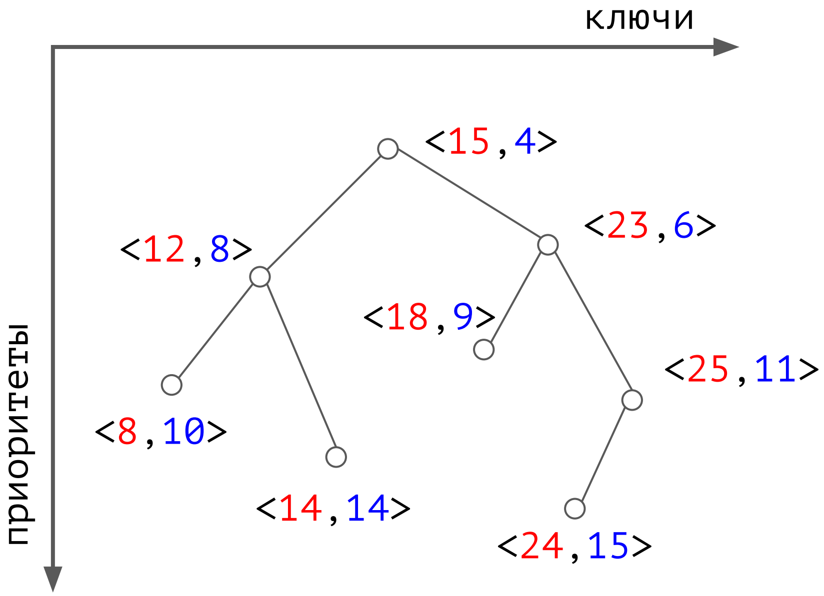


В левом поддереве все ключи меньше, в правом все ключи больше, ведь это BST.

В обоих поддеревьях все приоритеты выше, ведь это минимальная пирамида.

Декартово дерево

Декартово дерево — структура данных, являющаяся BST по ключам и минимальной пирамидой по приоритетам.



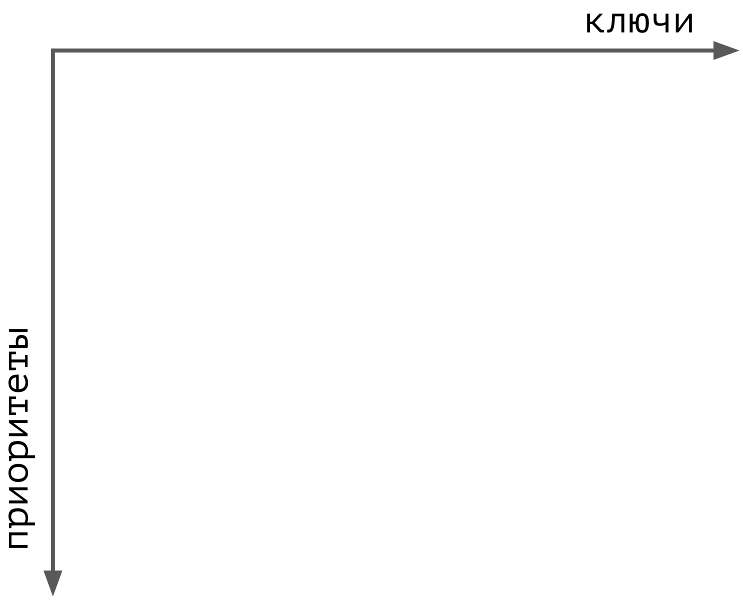
В левом поддереве все ключи меньше, в правом все ключи больше, ведь это BST.

В обоих поддеревьях все приоритеты выше, ведь это минимальная пирамида.

Альтернативные названия: treap (tree + heap), дермида (дерево + пирамида), …

Декартово дерево: построение (наивное)

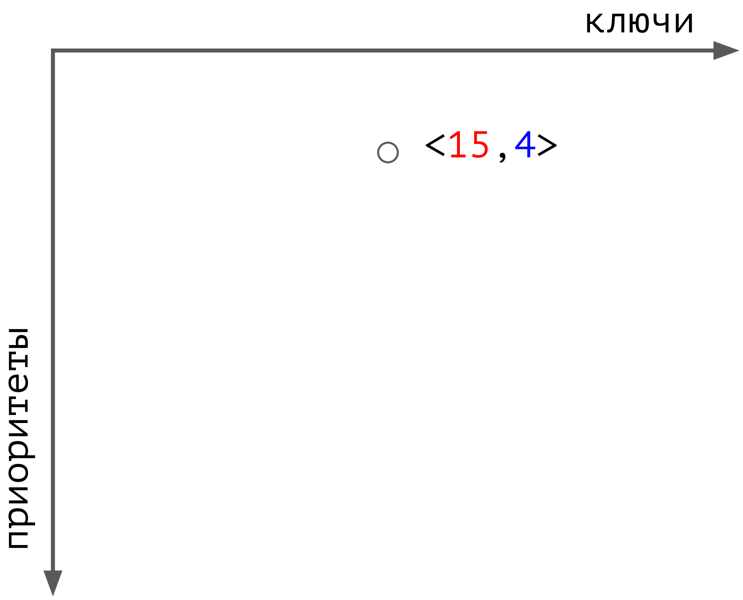
Пусть есть (отсортированный) набор ключей. Каждому дали случайный приоритет. Как построить дерево? Кто корень?



8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Декартово дерево: построение (наивное)

Пусть есть (отсортированный) набор ключей. Каждому дали случайный приоритет. Как построить дерево? Кто корень?

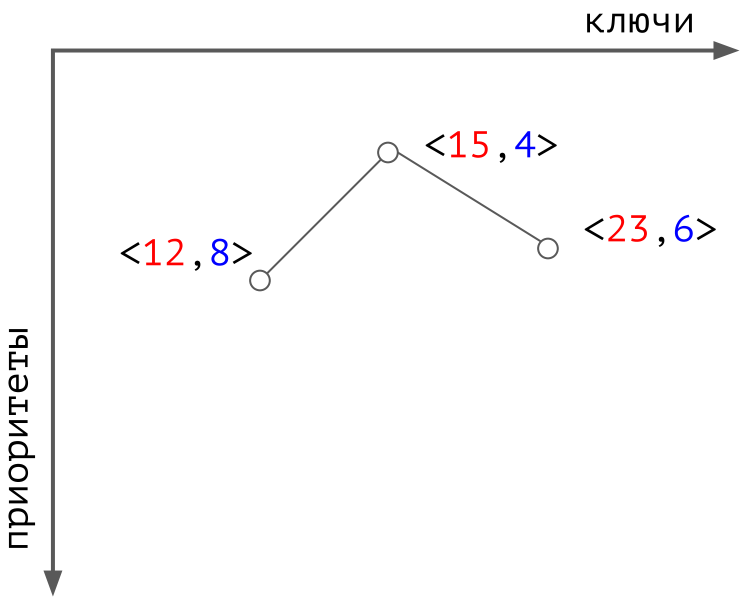


8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Корень - значение с наименьшим приоритетом.

Декартово дерево: построение (наивное)

Пусть есть (отсортированный) набор ключей. Каждому дали случайный приоритет. Как построить дерево? Кто корень?



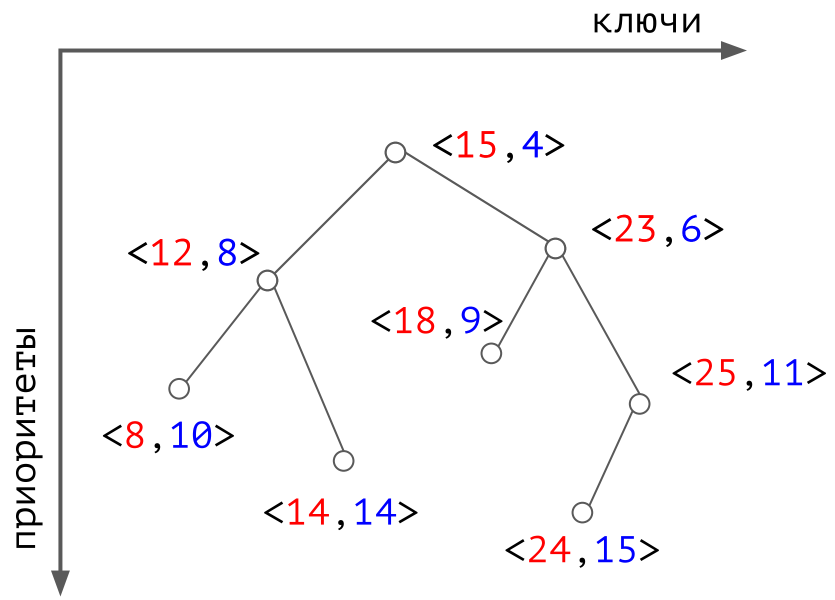
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Корень - значение с наименьшим приоритетом.

Элементы слева и справа - декартовы поддеревья, повторяем для них рекурсивно.

Декартово дерево: построение (наивное)

Пусть есть (отсортированный) набор ключей. Каждому дали случайный приоритет. Как построить дерево? Кто корень?



8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

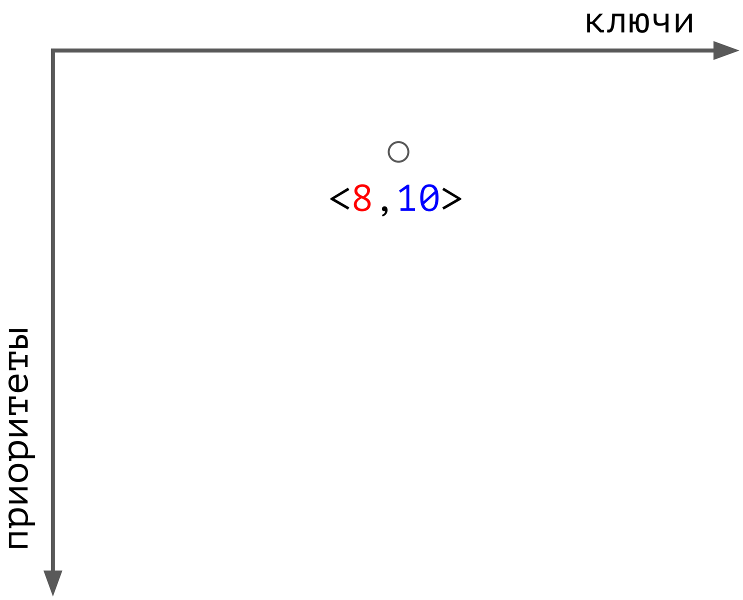
Корень - значение с наименьшим приоритетом.

Элементы слева и справа - декартовы поддеревья, повторяем для них рекурсивно.

Неэффективный алгоритм, можем сделать лучше, за линейное время.

Декартово дерево: построение линейное

Пусть есть (отсортированный) набор ключей. Каждому дали случайный приоритет. Как построить дерево? Кто корень?

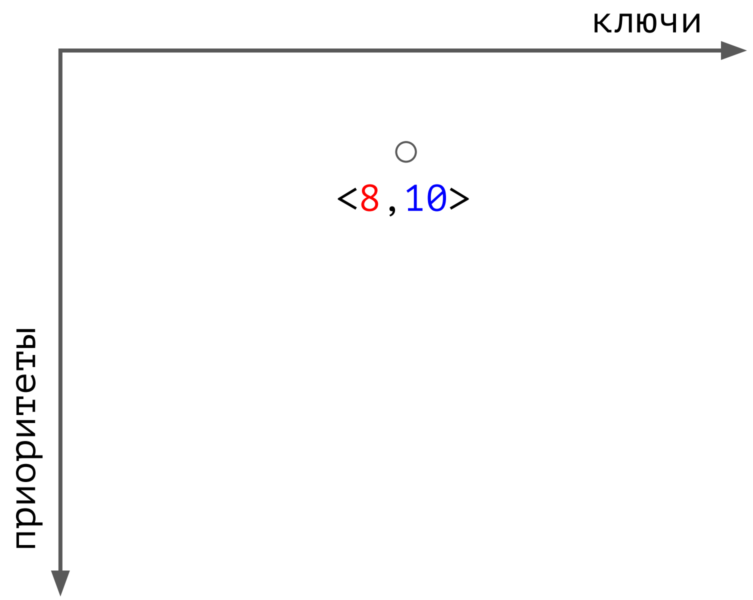


8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо, смотрим на i-ую пару ключ значение.

Декартово дерево: построение линейное

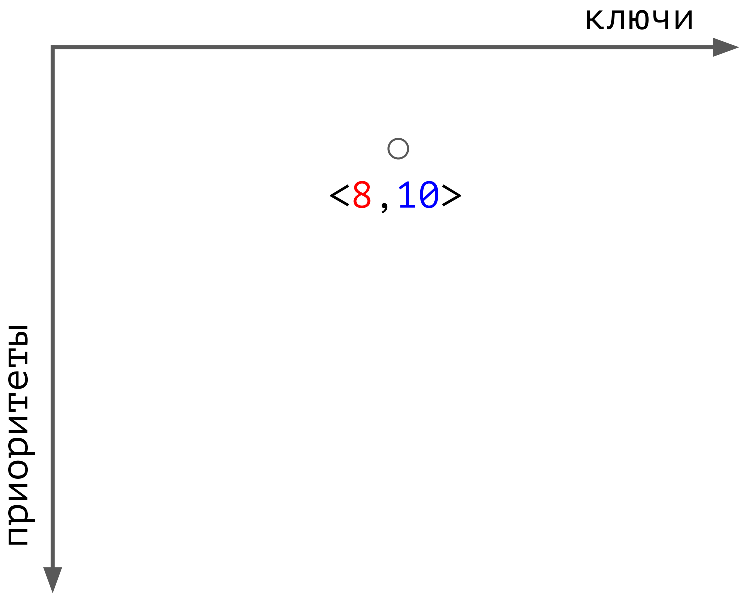
Пусть есть (отсортированный) набор ключей. Каждому дали случайный приоритет. Как построить дерево? Кто корень?



8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Декартово дерево: построение линейное

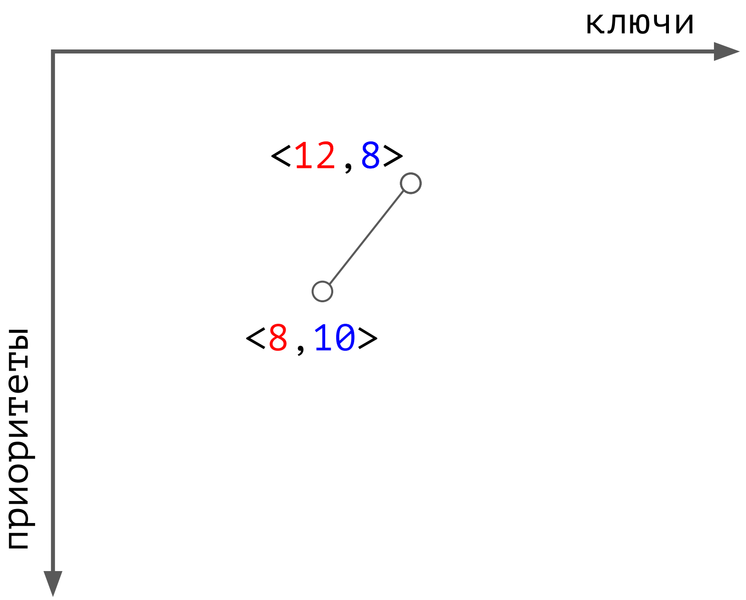


8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

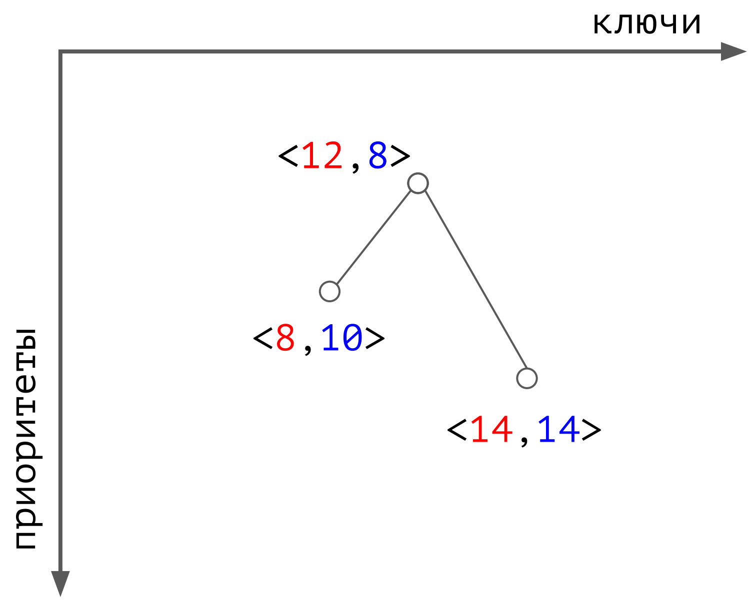


8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

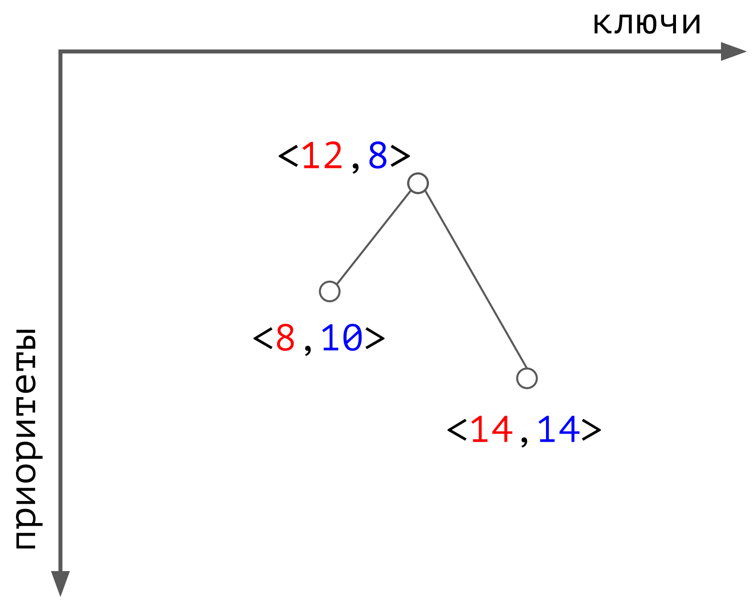


	i-1	i					
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

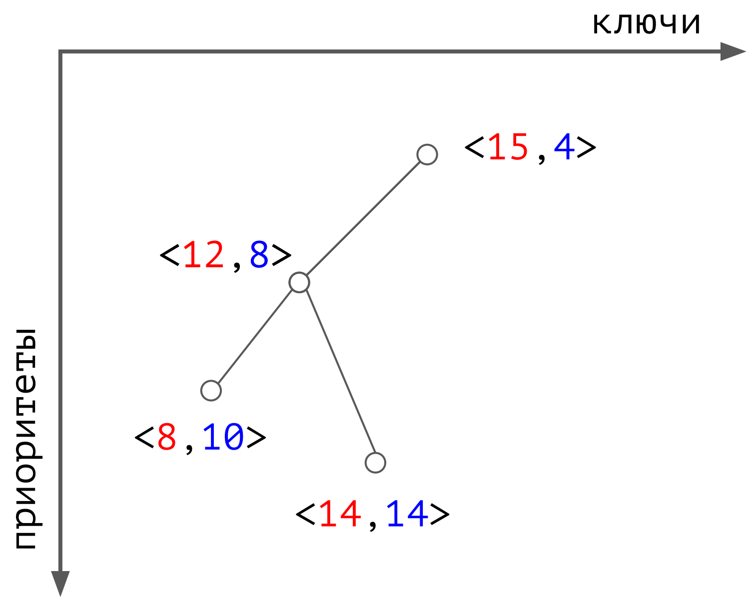


		i-1	i				
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

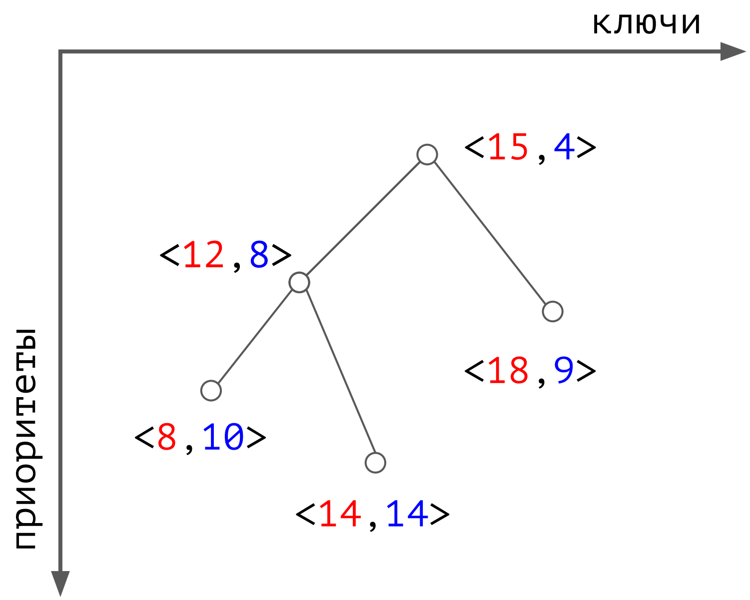


		i-1	i				
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

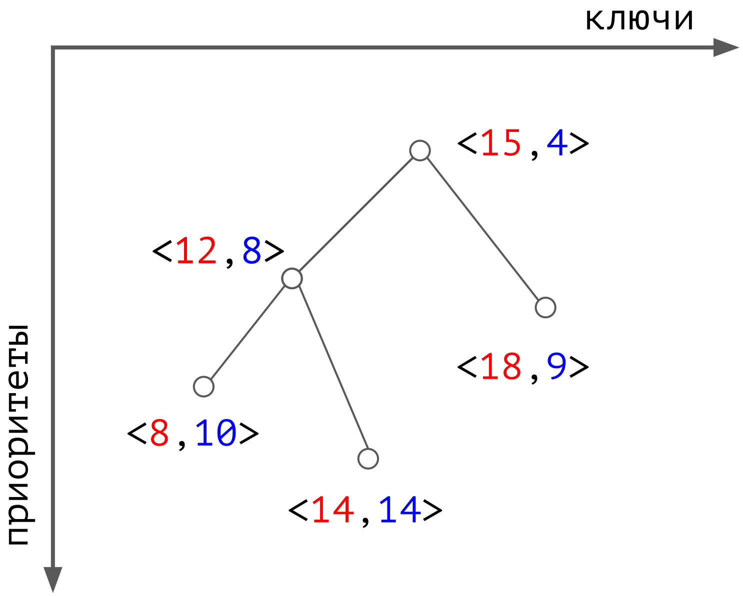


			i-1	i			
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

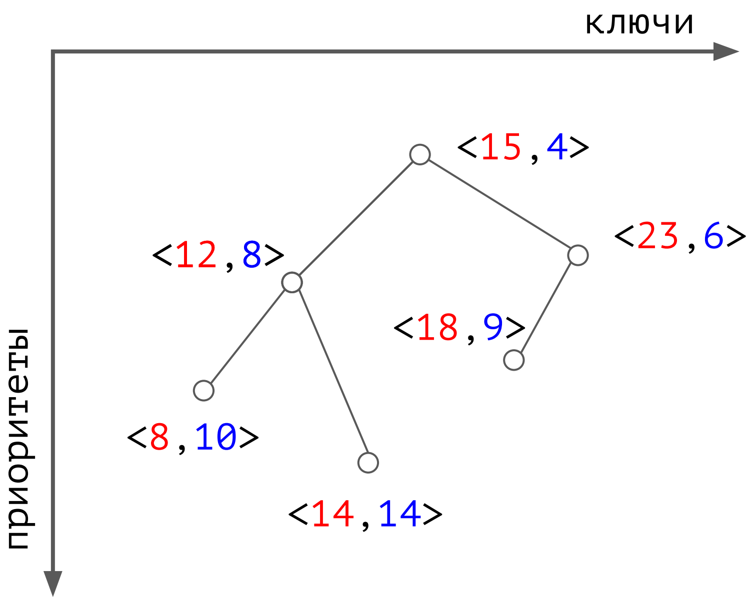


				i-1	i		
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

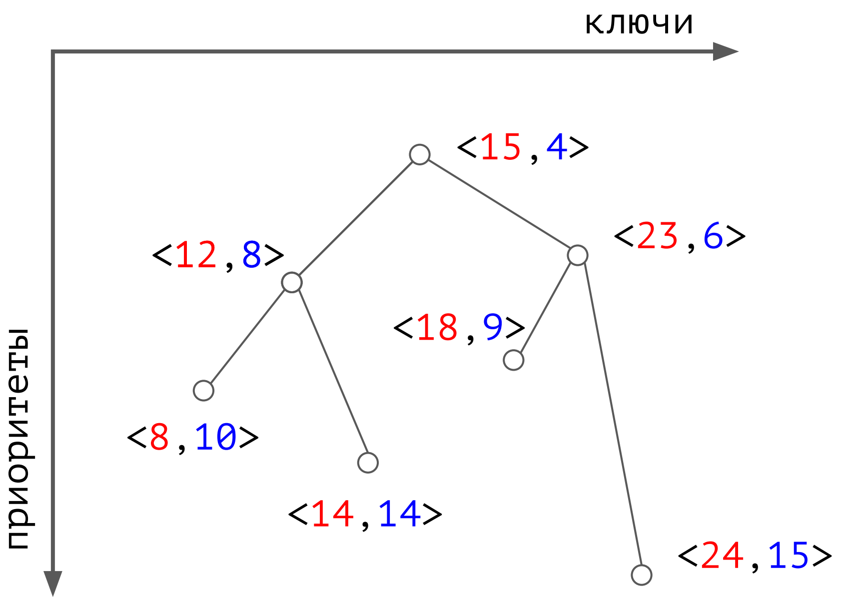


				i-1	i		
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

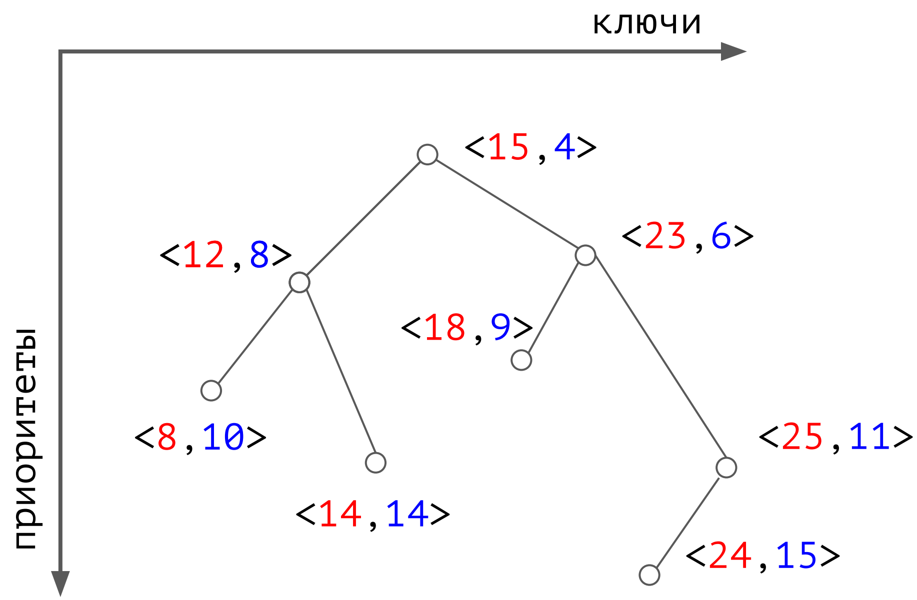


					i-1	i	
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное

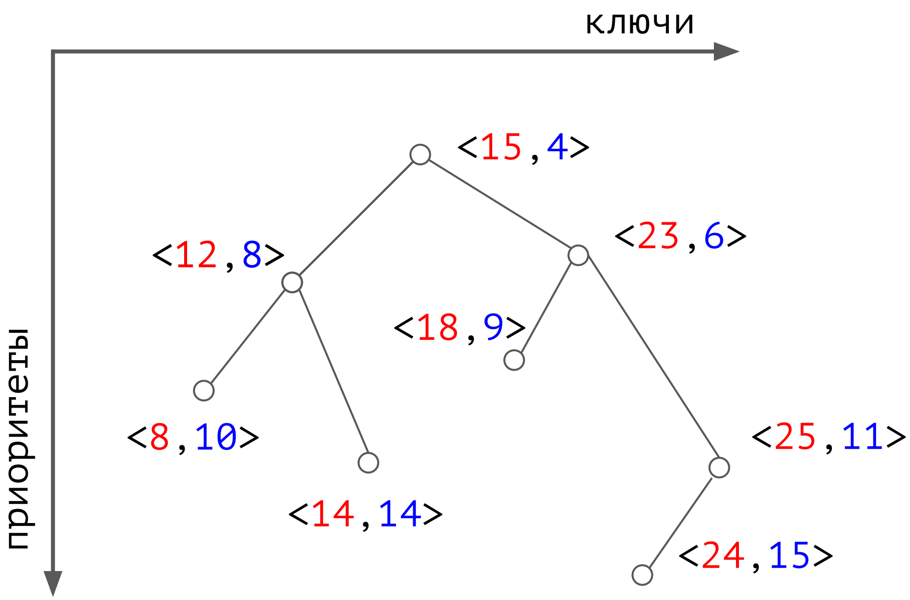


						i-1	i
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

Декартово дерево: построение линейное



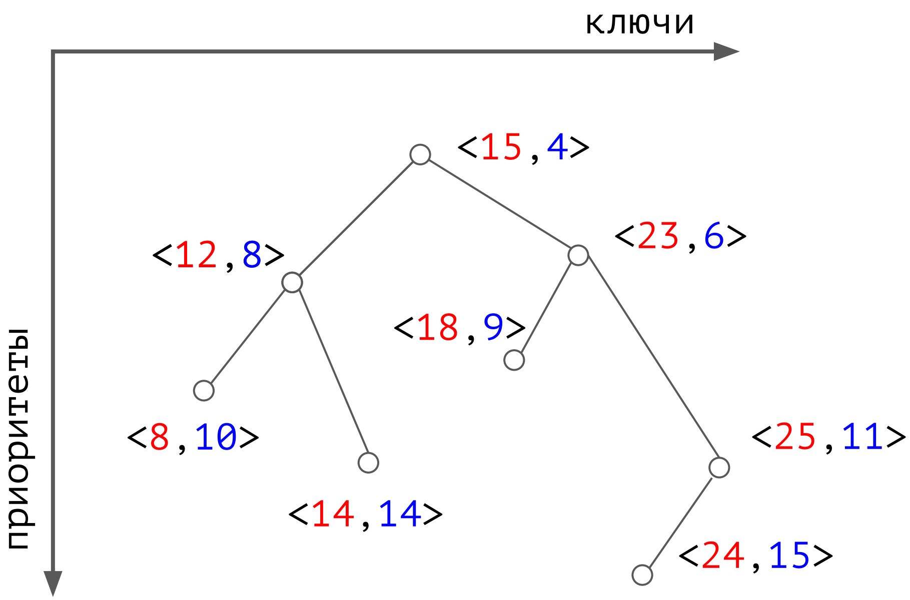
						i-1	i
8	12	14	15	18	23	24	25
10	8	14	4	9	6	15	11

Идем слева направо (по списку), смотрим на i-ую пару ключ значение, пытаемся добавить ее в качестве правого сына к (i-1)-ой паре.

Если не получается, идем от (i-1)-ого вверх, пока не найдем подходящую пару (или до конца). Делаем i-ую вершину - правым сыном того, кого нашли (или корнем), а то, откуда пришли - левым сыном i-ой вершины.

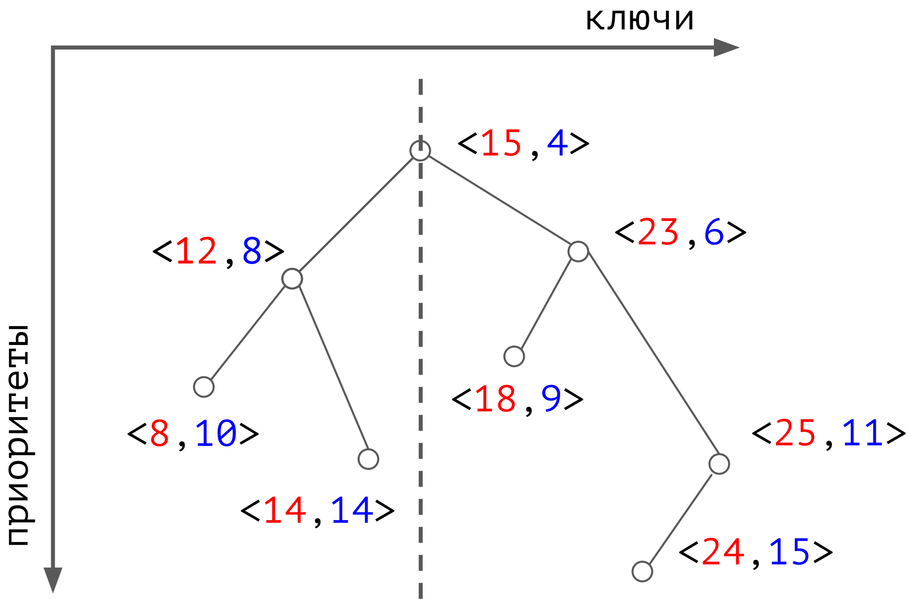
Сложность: каждую вершину посещаем не больше двух раз - при добавлении и при "переходе через нее" при поиске места вставки. Дальше она уходит в левое дерево. Итого: O(N)

Декартово дерево: операции



1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

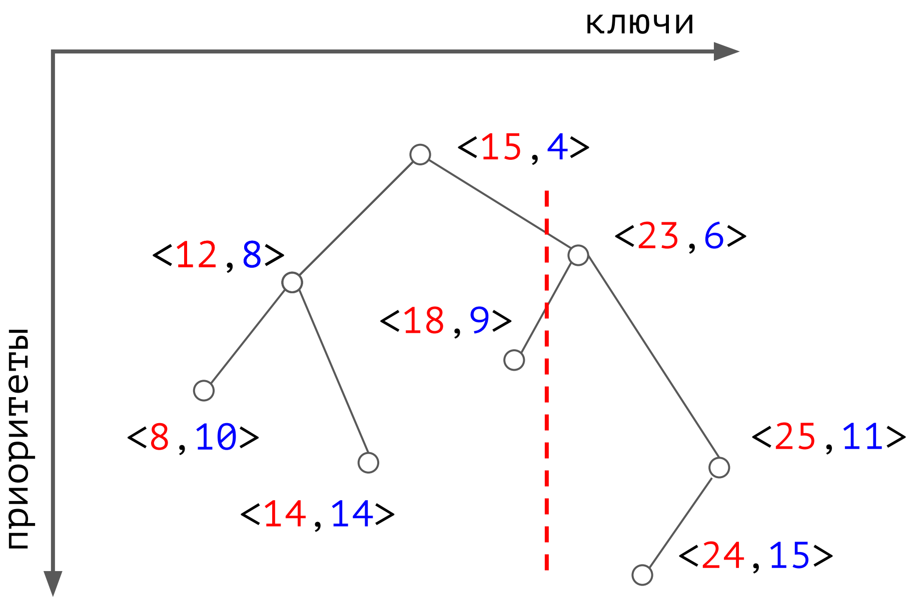
Декартово дерево: операции



1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

Со split(15) все понятно

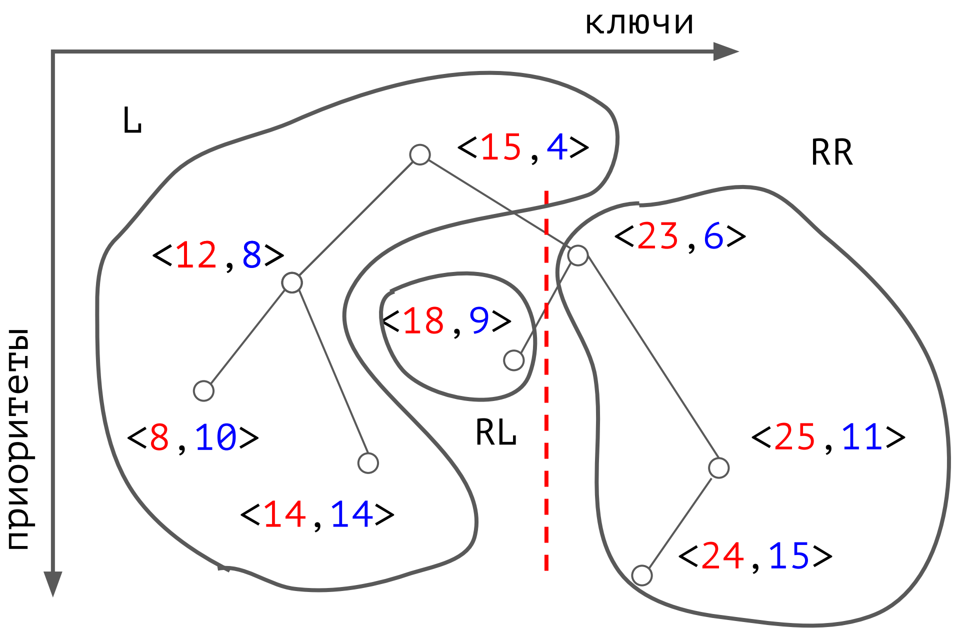
Декартово дерево: операции



1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

Co split(15) все понятно, но вот split(20) уже интереснее.

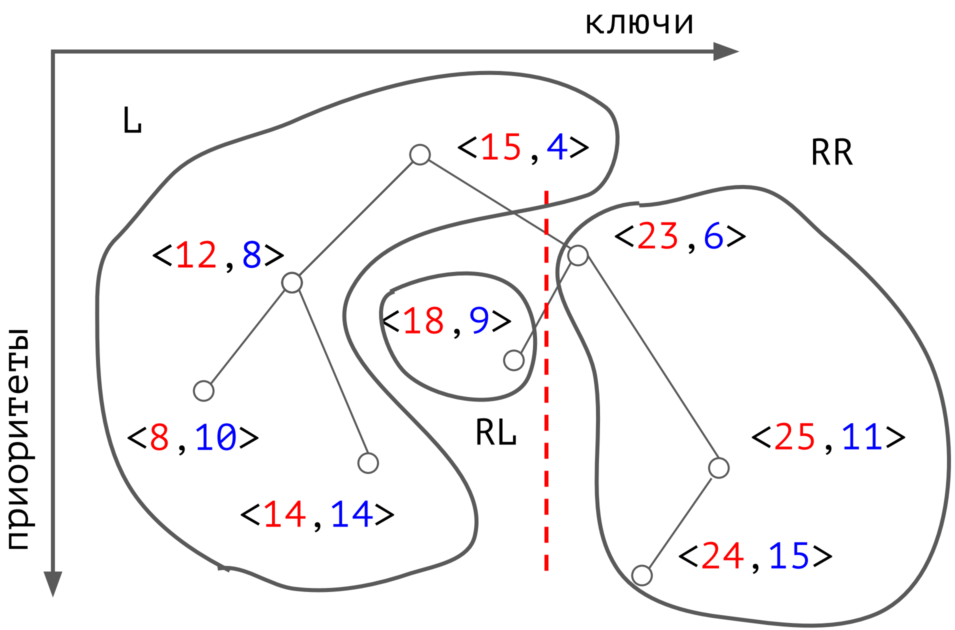
Декартово дерево: операции



1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

Со split(15) все понятно, но вот split(20) уже интереснее.

Декартово дерево: операции

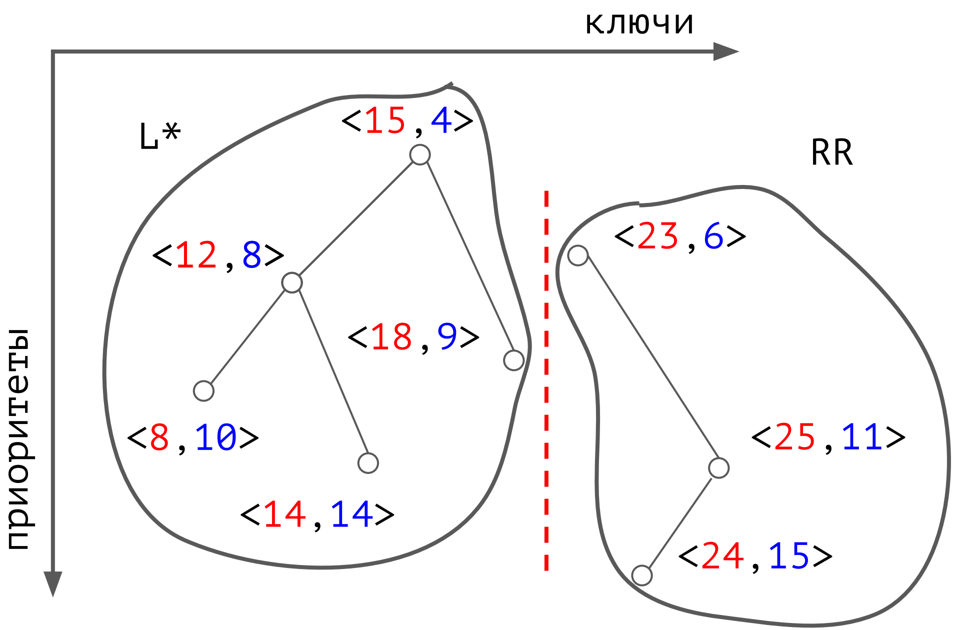


1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

   Если разрез прошел правее root.key (т.е. k > root.key), то вызываем рекурсивно split для правого поддерева, получаем <RL, RR>.

Со split(15) все понятно, но вот split(20) уже интереснее.

Декартово дерево: операции



Со split(15) все понятно, но вот split(20) уже интереснее.

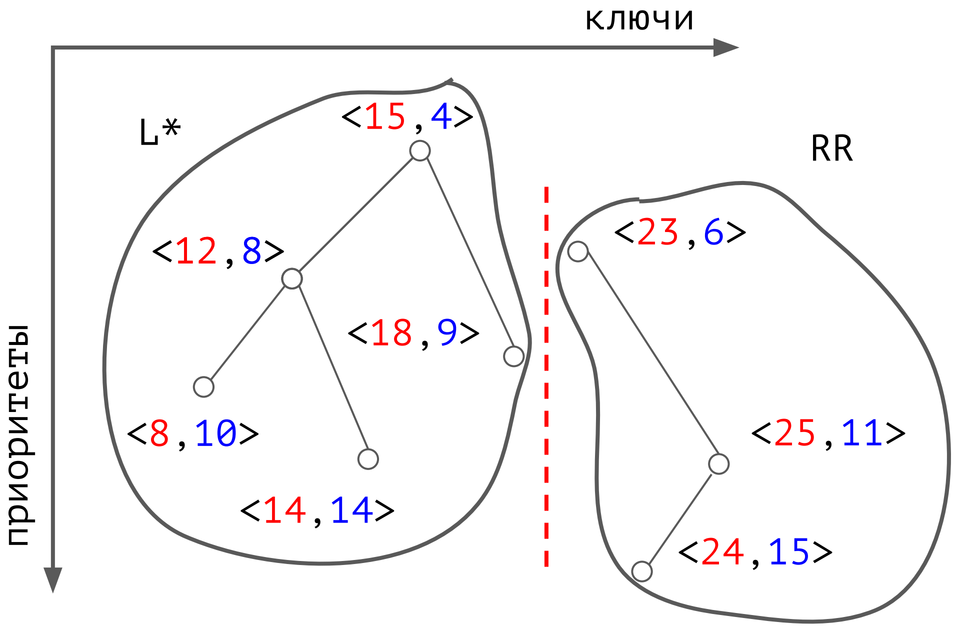
1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

   Если разрез прошел правее root.key (т.е. k > root.key), то вызываем рекурсивно split для правого поддерева, получаем <RL, RR>.

   L* = L с подвязанным в качестве правого сына корня поддеревом RL.

   Ответ в таком случае: <L*, RR>.

Декартово дерево: операции

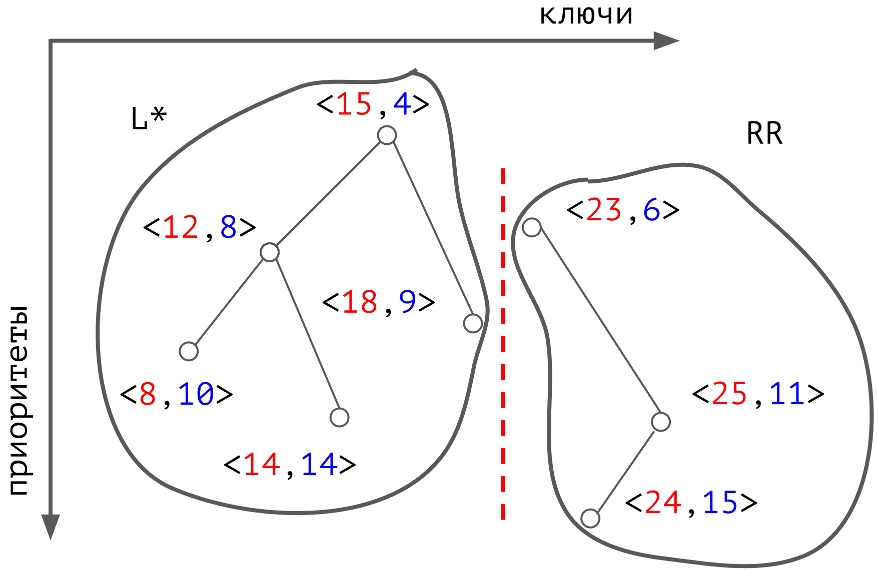


1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

   Если разрез прошел правее root.key (т.е. k > root.key), то вызываем рекурсивно split для правого поддерева, получаем <RL, RR>.

   L* = L с подвязанным в качестве правого сына корня поддеревом RL.

   Ответ в таком случае: <L*, RR>. Симметрично для случая k <= root.key.



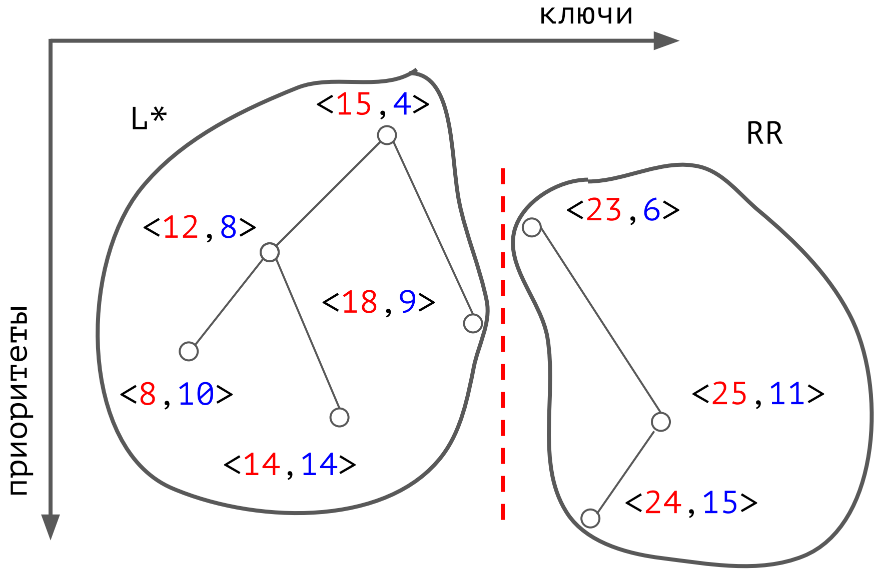

```
def split(T: Treap, k) -> (Treap, Treap):
  if k > T.root.key:
    RL, RR = split(T.root.right, k)
    T.right = RL
    return (T, RR)
  else
    LL, LR = split(T.root.left, k)
    T.left = LR
    return (LL, T)
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

   Если разрез прошел правее root.key (т.е. k > root.key), то вызываем рекурсивно split для правого поддерева, получаем <RL, RR>.

   L* = L с подвязанным в качестве правого сына корня поддеревом RL.

   Ответ в таком случае: <L*, RR>. Симметрично для случая k <= root.key.



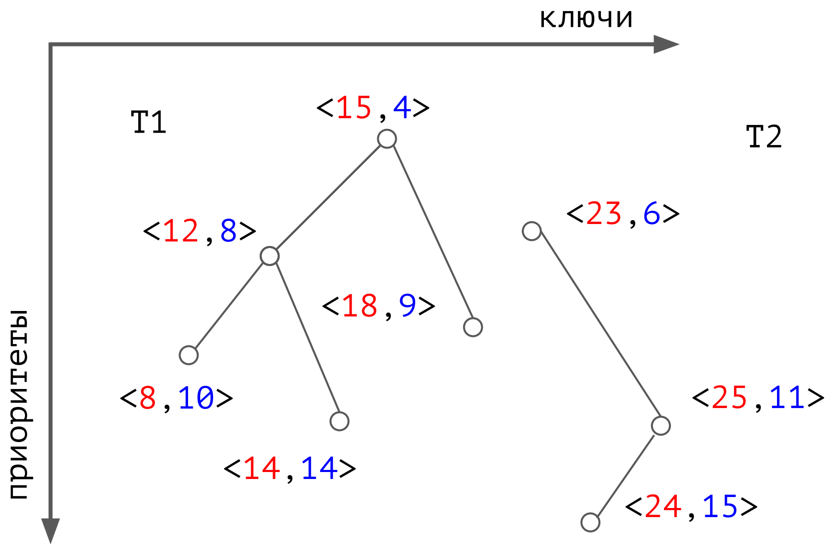

```
def split(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)
  if k > T.root.key:
    RL, RR = split(T.root.right, k)
    T.right = RL
    return (T, RR)
  else
    LL, LR = split(T.root.left, k)
    T.left = LR
    return (LL, T)
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

   Если разрез прошел правее root.key (т.е. k > root.key), то вызываем рекурсивно split для правого поддерева, получаем <RL, RR>.

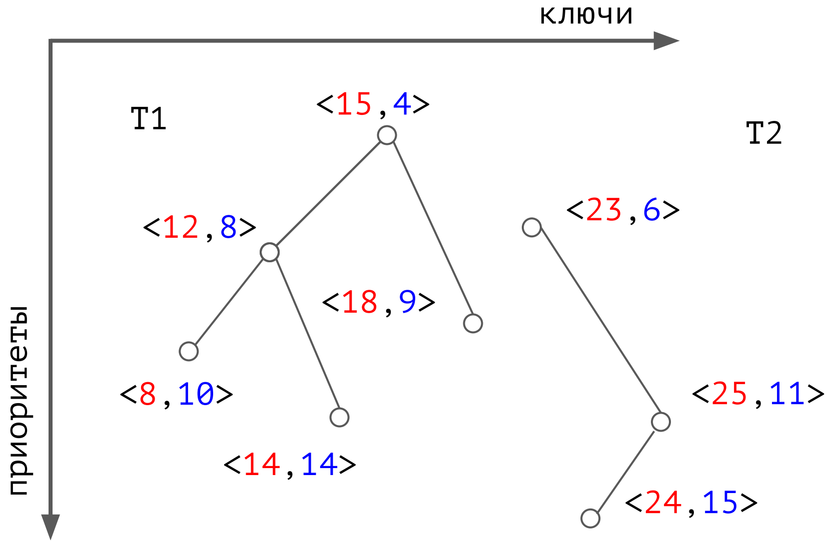
   L* = L с подвязанным в качестве правого сына корня поддеревом RL.

   Ответ в таком случае: <L*, RR>. Симметрично для случая k <= root.key.



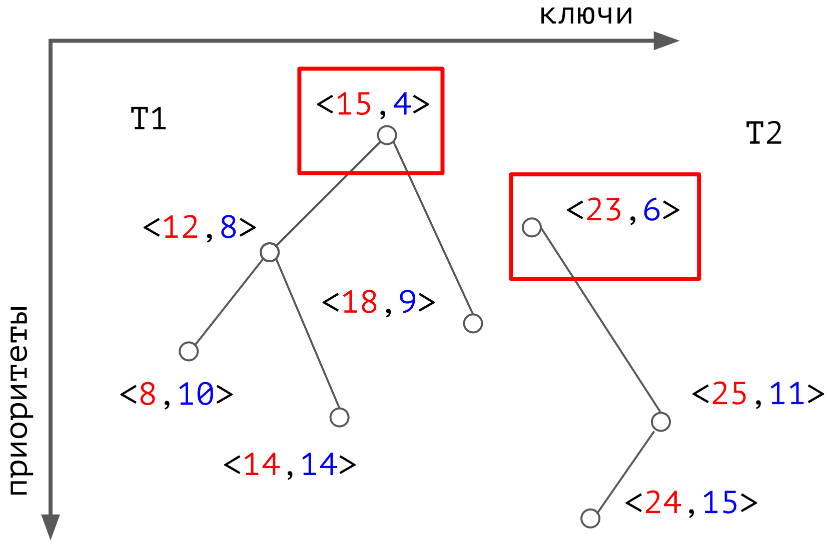
1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2.



1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

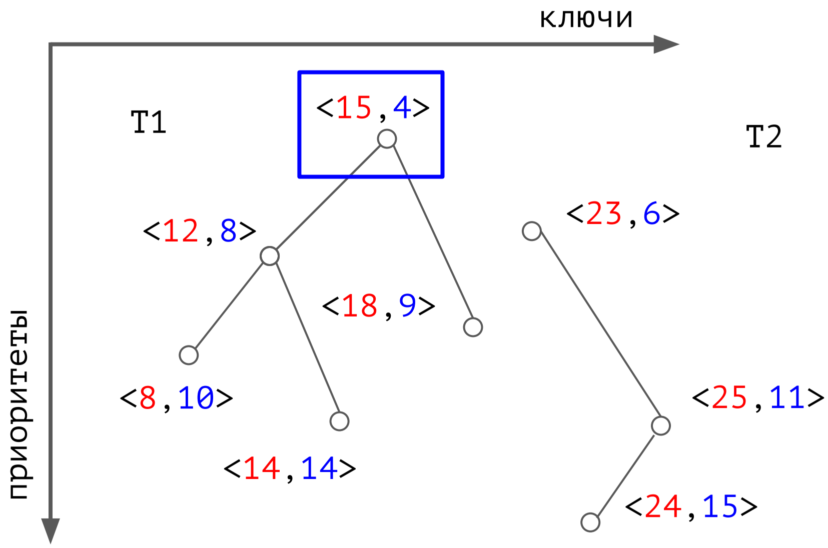
2. операция `merge(T1, T2)`: даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.



1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

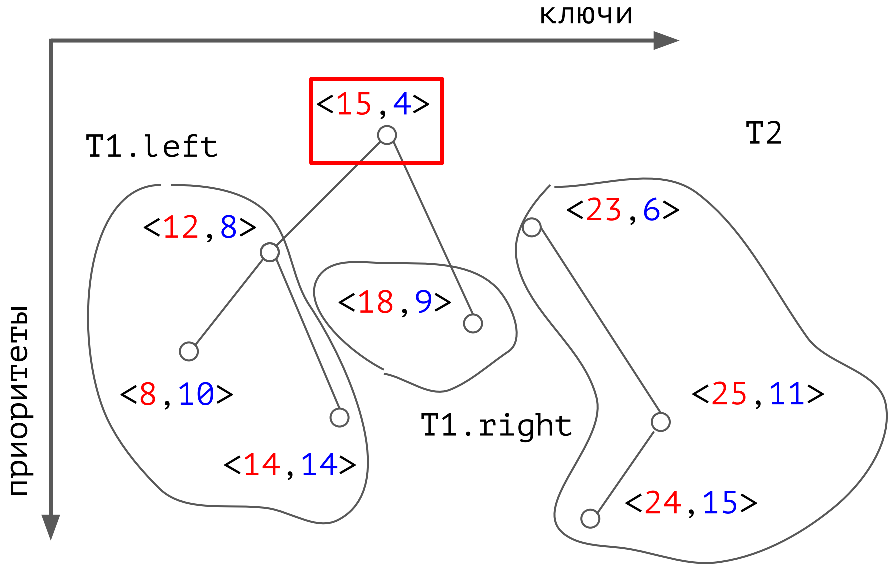
   Корнем нового дерева станет либо корень T1 либо корень T2.



1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

   Корнем нового дерева станет либо корень T1 либо корень T2. По определению берем корнем того, у кого приоритет меньше (т.к. на этом примере T1.root.p < T2.root.p, то берем в качестве нового рута T1.root).

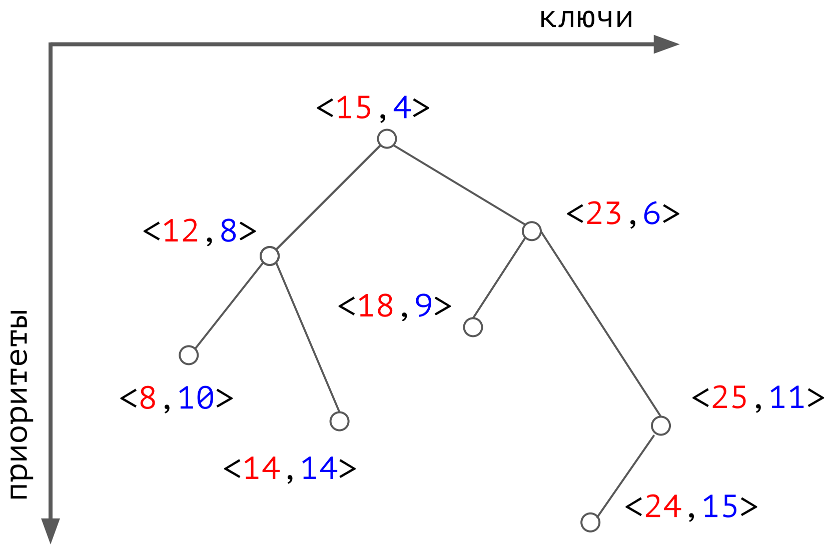


1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

   Корнем нового дерева станет либо корень T1 либо корень T2. По определению берем корнем того, у кого приоритет меньше.

   В нашем случае в качестве левого поддерева берем T1.left, в качестве правого merge(T1.right, T2)

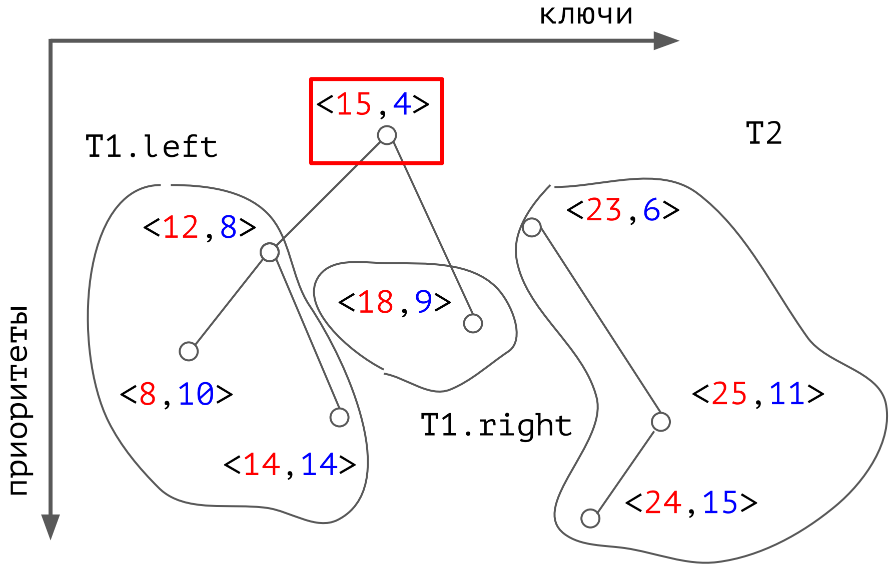


1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция `merge(T1, T2)`: даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

   Корнем нового дерева станет либо корень T1 либо корень T2. По определению берем корнем того, у кого приоритет меньше.

   В нашем случае в качестве левого поддерева берем T1.left, в качестве правого merge(T1.right, T2)



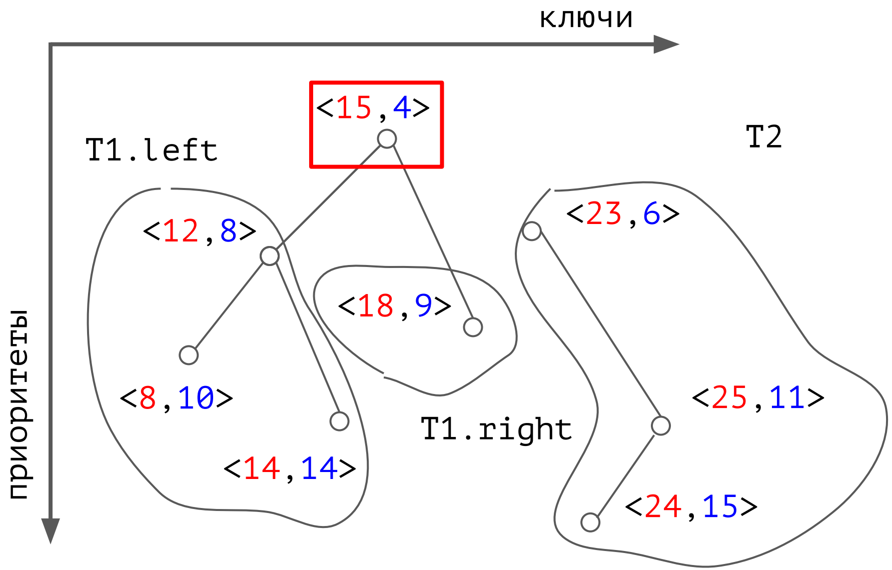

```
def merge(T1, T2: Treap) -> Treap:
  if not T1: return T2
  if not T2: return T1
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

   Корнем нового дерева станет либо корень T1 либо корень T2. По определению берем корнем того, у кого приоритет меньше.

   В нашем случае в качестве левого поддерева берем T1.left, в качестве правого merge(T1.right, T2)



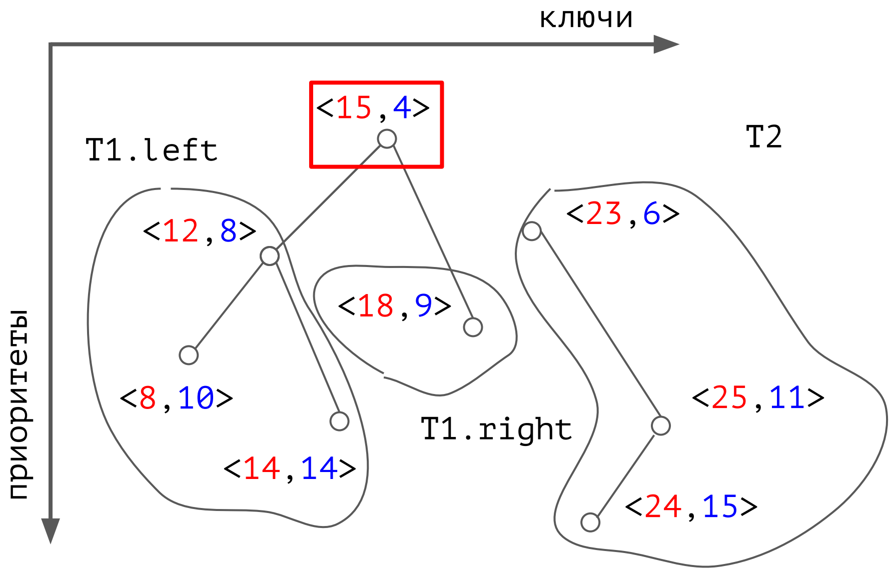

```python
def merge(T1, T2: Treap) -> Treap:
  if not T1: return T2
  if not T2: return T1

  if T1.root.p < T2.root.p:
    T1.right = merge(T1.right, T2)
    return T1
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

   Корнем нового дерева станет либо корень T1 либо корень T2. По определению берем корнем того, у кого приоритет меньше.

   В нашем случае в качестве левого поддерева берем T1.left, в качестве правого merge(T1.right, T2)



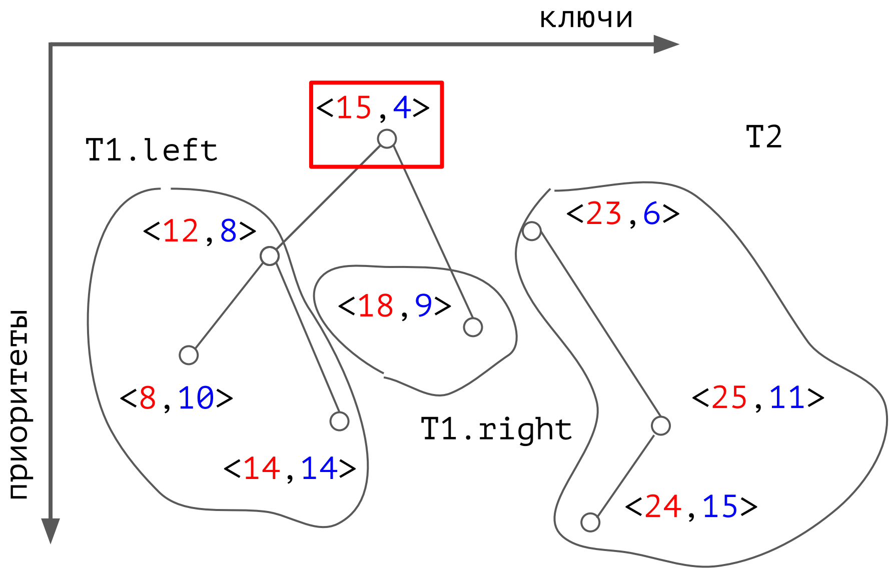

```
def merge(T1, T2: Treap) -> Treap:
  if not T1: return T2
  if not T2: return T1

  if T1.root.p < T2.root.p:
    T1.right = merge(T1.right, T2)
    return T1
  else:
    T2.left = merge(T1, T2.left)
    return T2
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

   Корнем нового дерева станет либо корень T1 либо корень T2. По определению берем корнем того, у кого приоритет меньше.

   В нашем случае в качестве левого поддерева берем T1.left, в качестве правого merge(T1.right, T2)

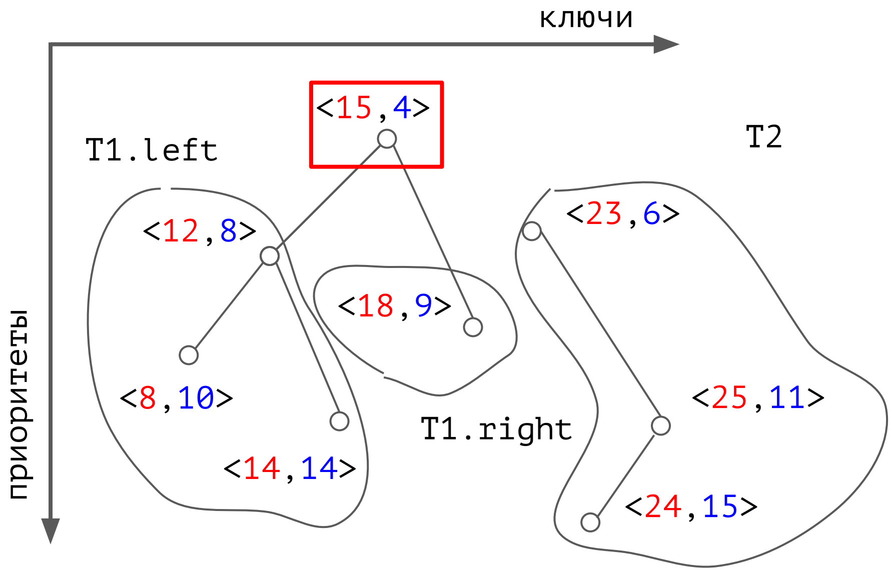



```
def merge(T1, T2: Treap) -> Treap:
  if not T1: return T2
  if not T2: return T1

  if T1.root.p < T2.root.p:
    T1.right = merge(T1.right, T2)
    return T1
  else:
    T2.left = merge(T1, T2.left)
    return T2
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

Сложность split и merge?




```
def merge(T1, T2: Treap) -> Treap:
  if not T1: return T2
  if not T2: return T1

  if T1.root.p < T2.root.p:
    T1.right = merge(T1.right, T2)
    return T1
  else:
    T2.left = merge(T1, T2.left)
    return T2
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

   Сложность split и merge?

   Всегда идем в рекурсию только налево или только направо, сделать это сможем столько раз, сколько высота дерева, т.е. сложность O(height).

Бинарные деревья поиска

Операции:

1. find(value) -> O(height)
2. select(i) -> O(height)
3. min/max -> O(height)
4. pred/succ(ptr) -> O(height)
5. rank(value) -> O(height)
6. вывод в пор. возрастания -> O(N)

---

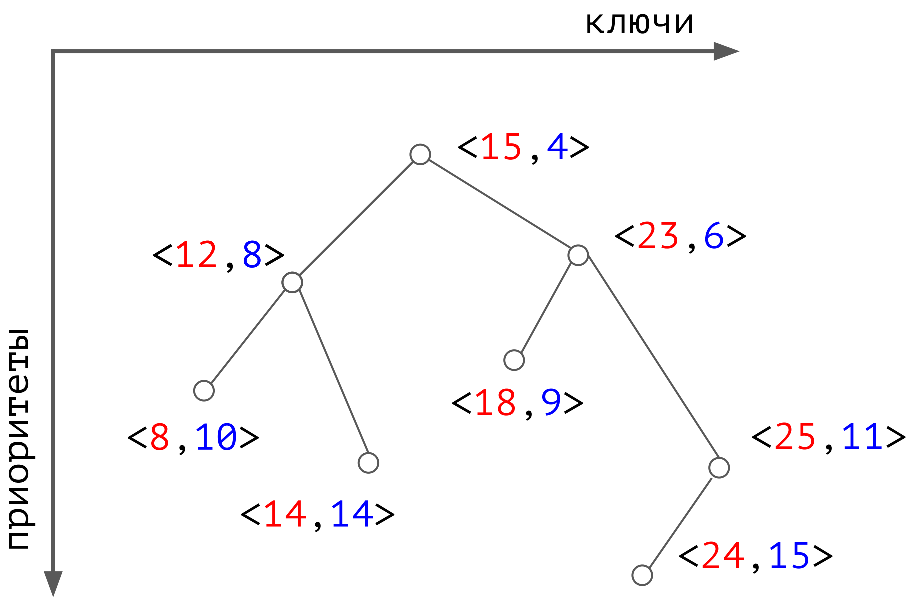
7. insert(value) -> O(height)
8. remove(value) -> O(height)

Назревает проблема!

Обещали логарифм, а дают какой-то O(height)

Значит нам нужны такие BST, чтобы высота у дерева всегда была logN

Декартово дерево: вставка

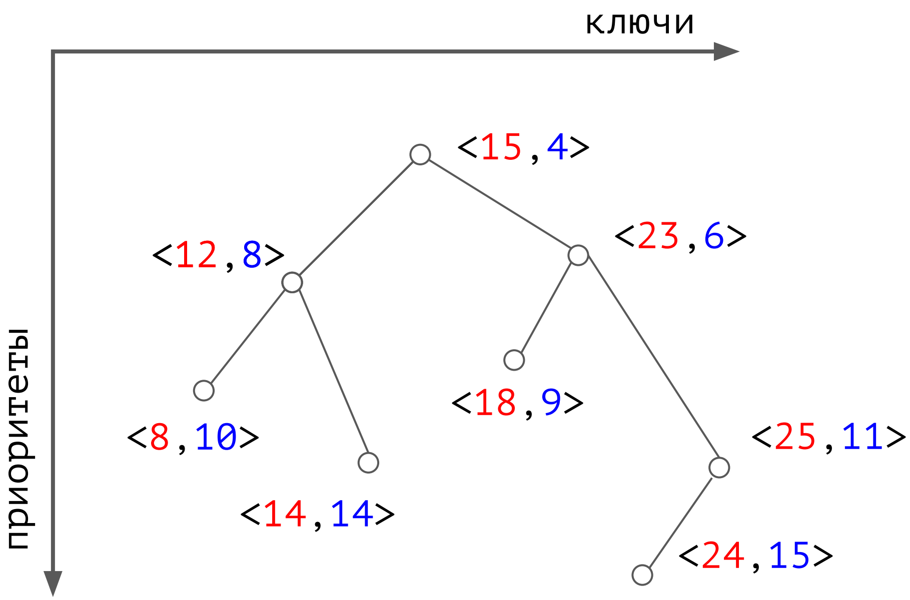


1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция `merge(T1, T2)`: даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция `insert(T, x)`: вставить пару с ключом x.

Декартово дерево: вставка

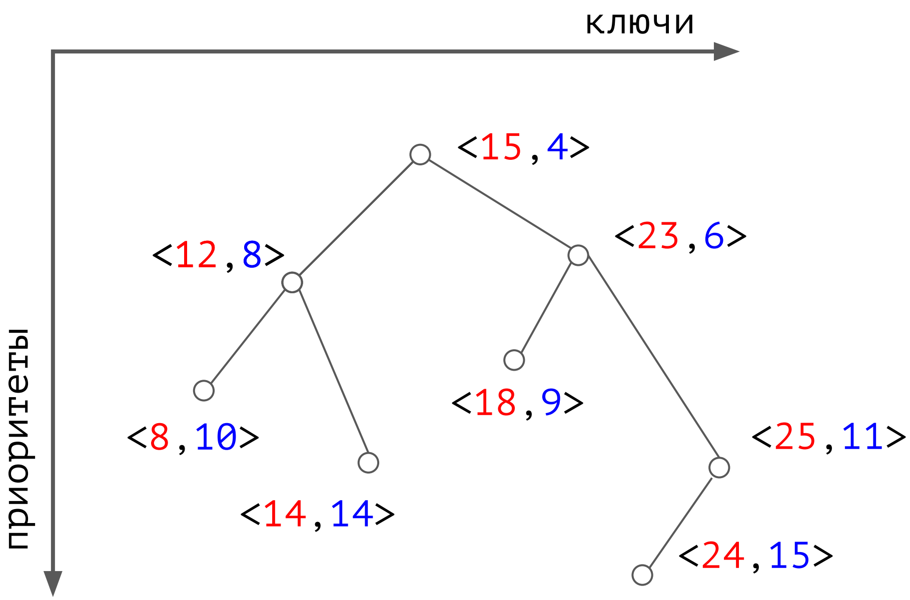


1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция `merge(T1, T2)`: даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция `insert(T, x)`: вставить пару с ключом x. Сгенерируем для него уникальный приоритет y. Как вставить <x, y>?

Декартово дерево: вставка



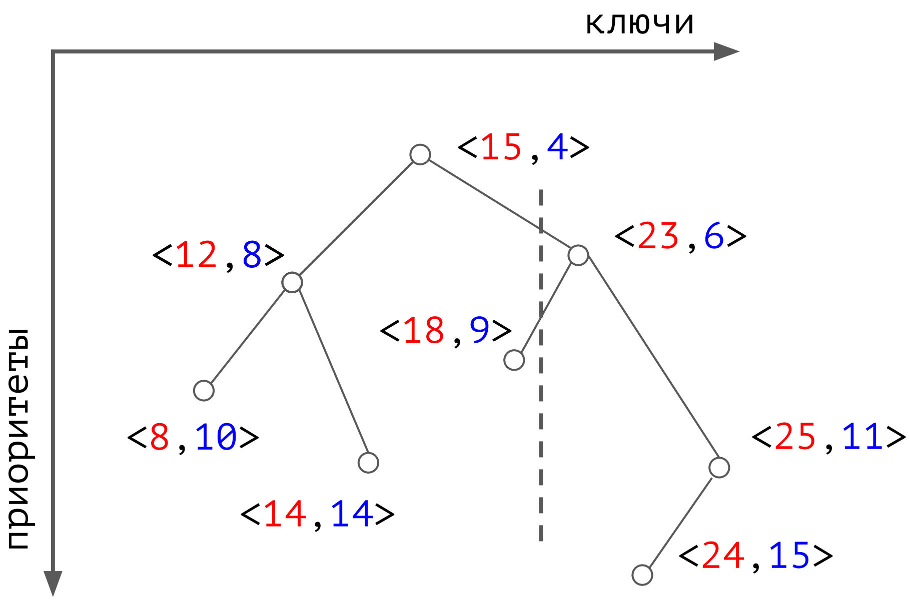
`insert(T, <20, 7>)?`

1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция `merge(T1, T2)`: даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция `insert(T, x)`: вставить пару с ключом x. Сгенерируем для него уникальный приоритет y. Как вставить <x, y>?

Декартово дерево: вставка



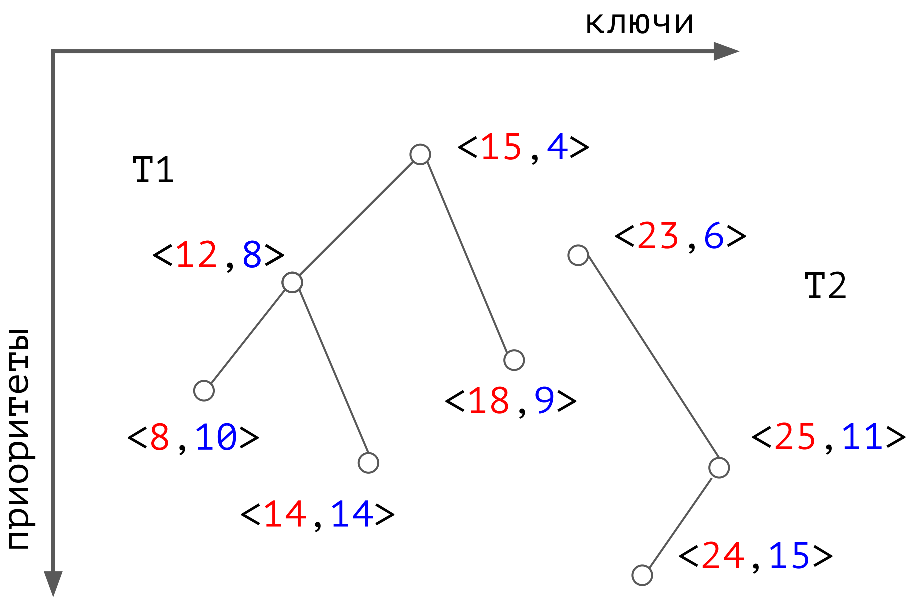

```
insert(T, <20, 7>) ->

1.  split(T, 20)
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x. Сгенерируем для него уникальный приоритет y. Как вставить <x, y>?

Декартово дерево: вставка



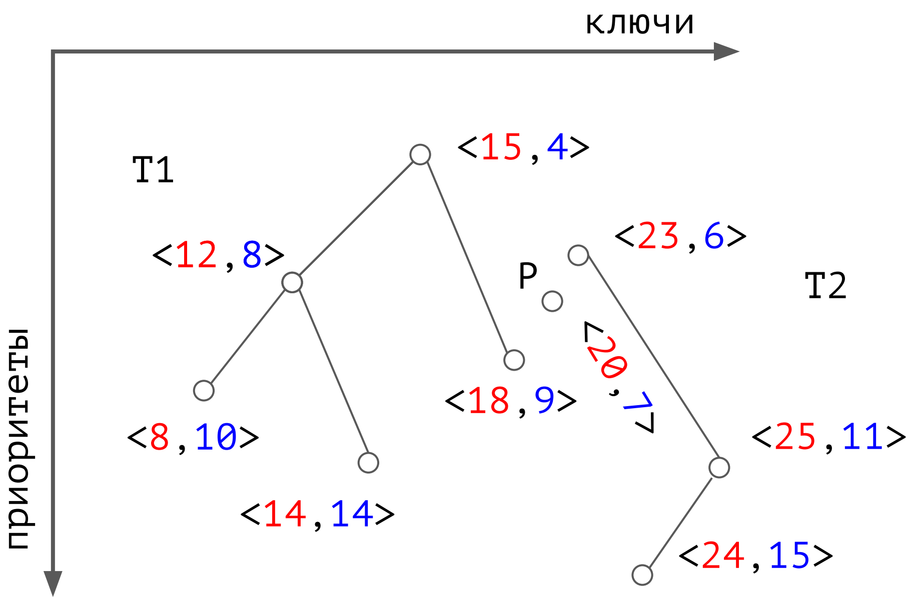

```
insert(T, <20, 7>) ->

1.  T1, T2 = split(T, 20)
```

1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция `merge(T1, T2)`: даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция `insert(T, x)`: вставить пару с ключом x. Сгенерируем для него уникальный приоритет y. Как вставить <x, y>?

Декартово дерево: вставка



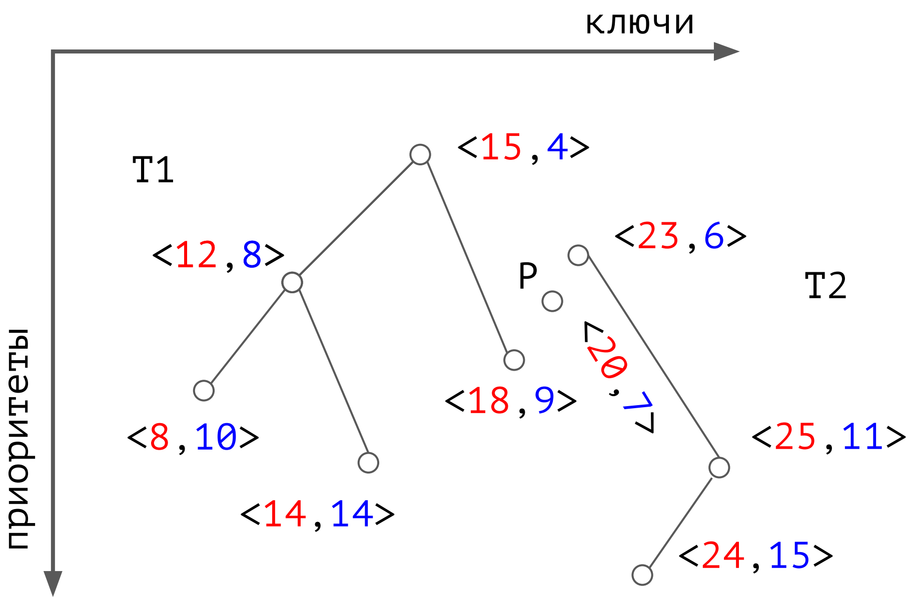

```
insert(T, <20, 7>) ->

1.  T1, T2 = split(T, 20)
2.  P = {<20, 7>}
```

1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция `merge(T1, T2)`: даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция `insert(T, x)`: вставить пару с ключом x. Сгенерируем для него уникальный приоритет y. Как вставить <x, y>?

Декартово дерево: вставка



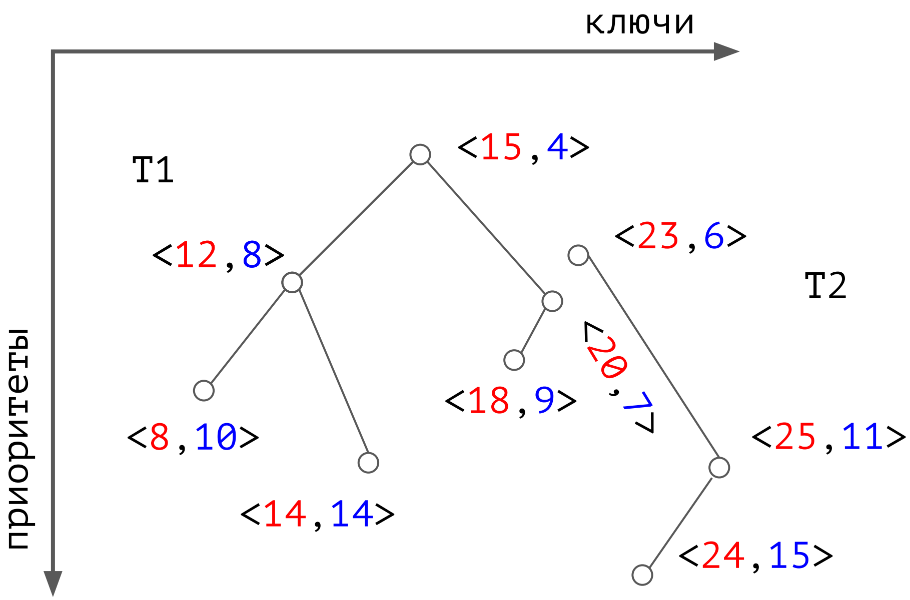

```
insert(T, <20, 7>) ->

1.   T1, T2 = split(T, 20)
2.   P = {<20, 7>}
3.   return merge(merge(T1, P), T2)
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x. Сгенерируем для него уникальный приоритет y. Как вставить <x, y>?

Декартово дерево: вставка



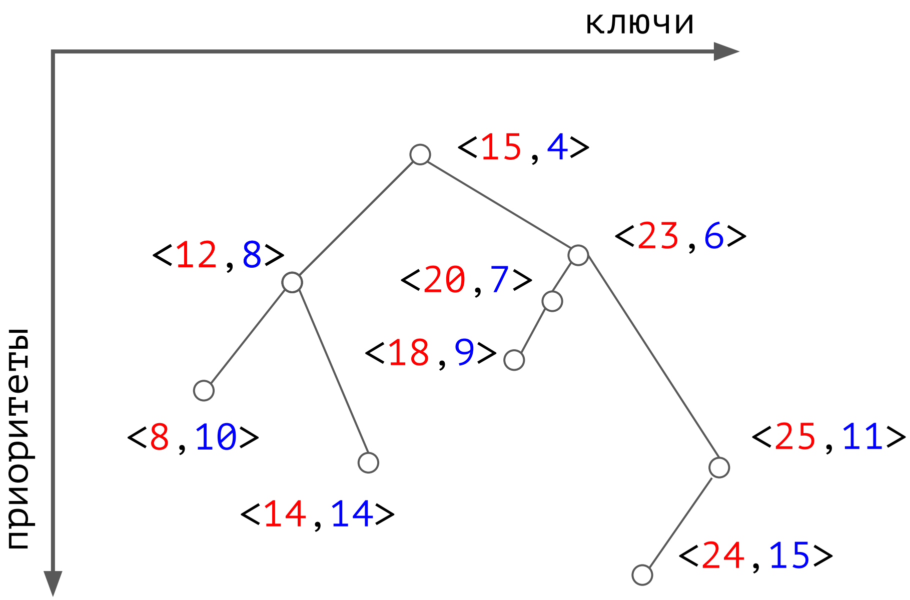

```
insert(T, <20, 7>) ->

1.  T1, T2 = split(T, 20)
2.  P = {<20, 7>}
3.  return merge(merge(T1, P), T2)
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x. Сгенерируем для него уникальный приоритет y. Как вставить <x, y>?

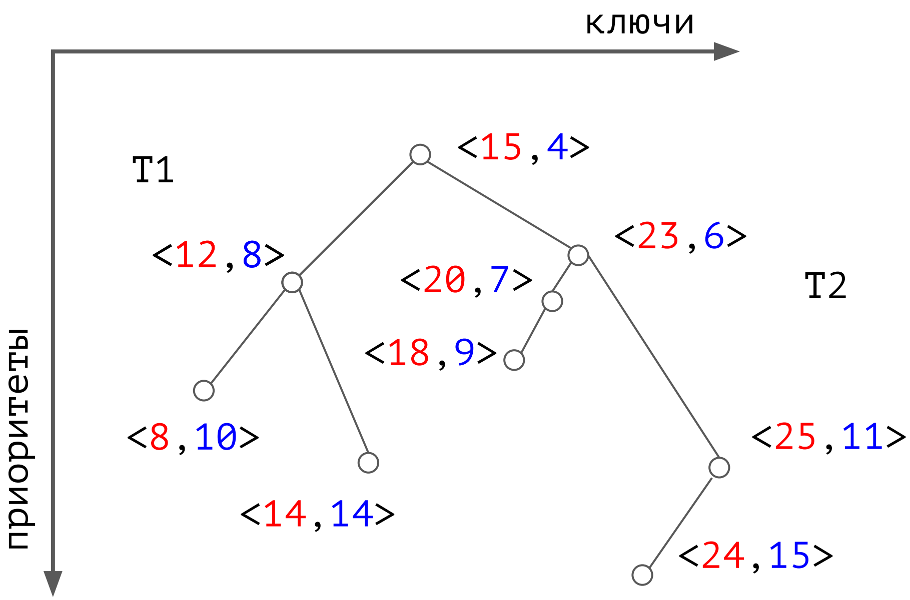
Декартово дерево: вставка




```
insert(T, <20, 7>) ->

1.   T1, T2 = split(T, 20)
2.   P = {<20, 7>}
3.   return merge(merge(T1, P), T2)
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.
2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.
3. операция insert(T, x): вставить пару с ключом x. Сгенерируем для него уникальный приоритет y. Как вставить <x, y>?

Декартово дерево: удаление



remove(T, 23)?

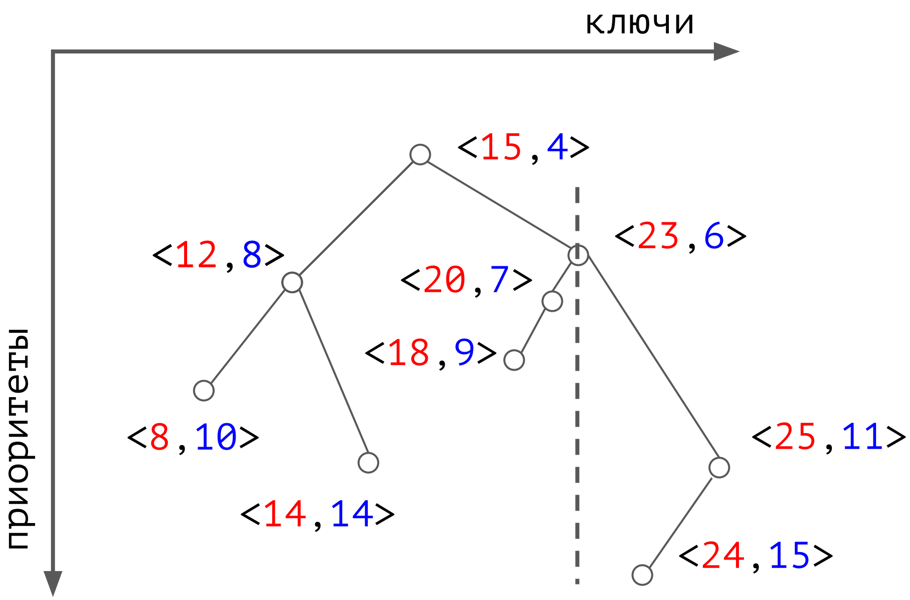
1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x.

4. операция remove(T, x): удалить элемент с ключом x.

Декартово дерево: удаление




```
remove(T, 23) ->

1.  T1, T2 = split(23)
```

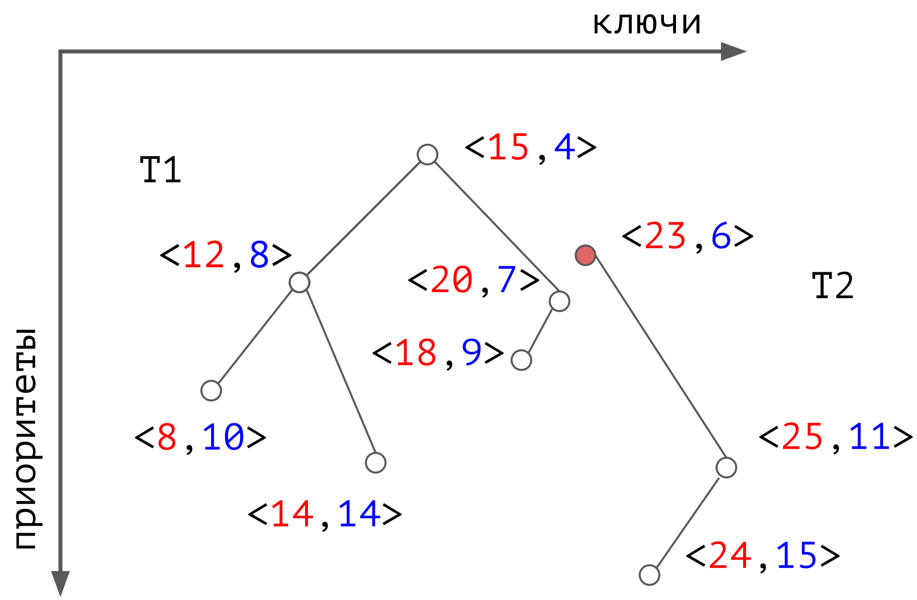
1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x.

4. операция remove(T, x): удалить элемент с ключом x.

Декартово дерево: удаление



remove(T, 23) ->

1. T1, T2 = split(23)

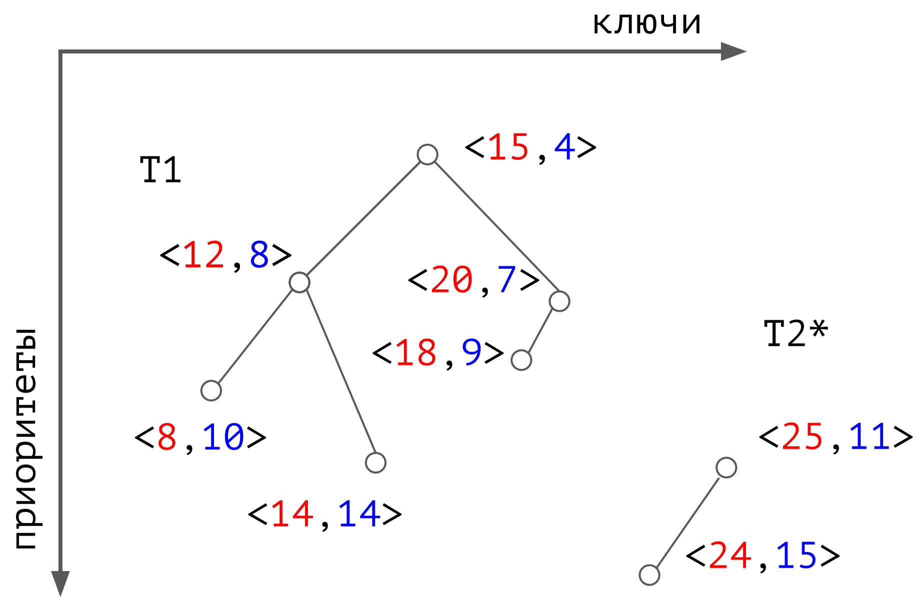
1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x.

4. операция remove(T, x): удалить элемент с ключом x.

Декартово дерево: удаление




```
remove(T, 23) ->

1.  T1, T2 = split(23)
2.  T2* = T2 / <23, 6>
```

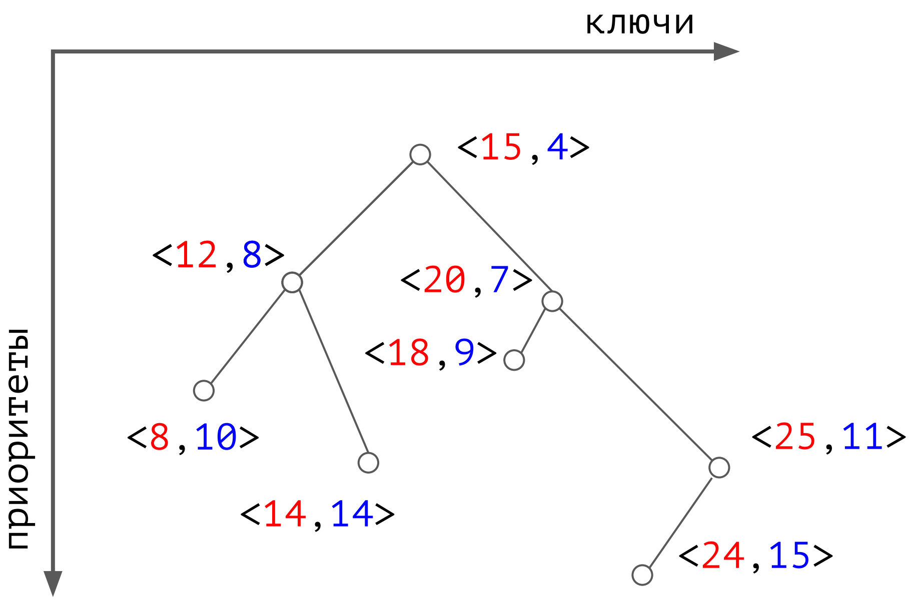
1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x.

4. операция remove(T, x): удалить элемент с ключом x.

Декартово дерево: удаление




```
remove(T, 23) ->

1.  T1, T2 = split(23)
2.  T2* = T2 / <23, 6>
3.  return merge(T1, T2*)
```

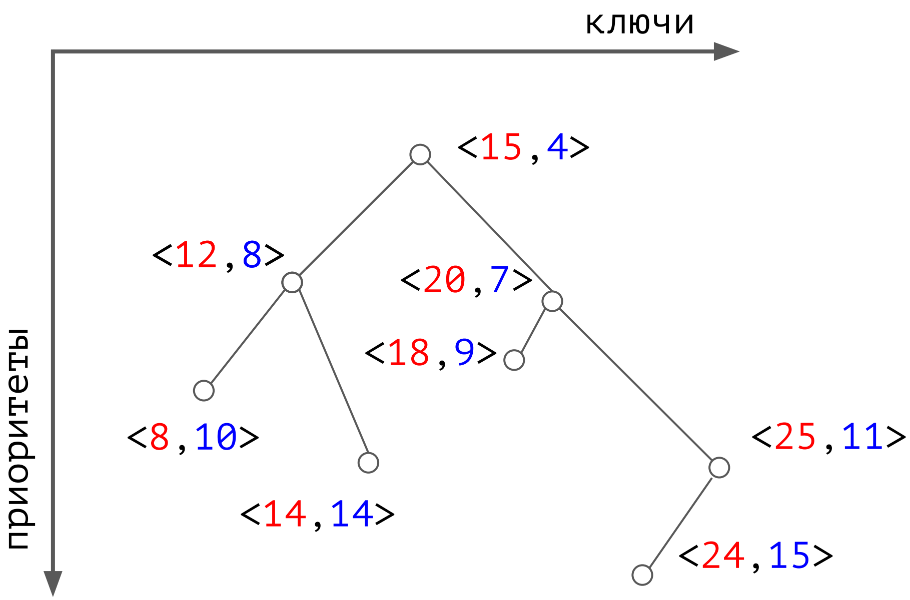
1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x.

4. операция remove(T, x): удалить элемент с ключом x.

Декартово дерево: удаление



1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x.

4. операция remove(T, x): удалить элемент с ключом x.

Итого: все операции снова зависят от O(heigh)

Декартово дерево

Есть ли какие-нибудь гарантии на высоту дерева?

Декартово дерево

Теорема: пусть есть декартово дерево из N элементов, в котором приоритеты выбраны случайным образом*. Тогда высота дерева в среднем составляет O(logN).

*точнее: приоритеты - случайный величины с равномерным распределением.

Декартово дерево

Теорема: пусть есть декартово дерево из N элементов, в котором приоритеты выбраны случайным образом*. Тогда высота дерева в среднем составляет O(logN).

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \text{ — предок } x_j \\ 0, & \text{иначе} \end{cases}$$

$d(v)$ - глубина вершины v;

*точнее: приоритеты - случайный величины с равномерным распределением.

Декартово дерево

Теорема: пусть есть декартово дерево из N элементов, в котором приоритеты выбраны случайным образом*. Тогда высота дерева в среднем составляет O(logN).

Док-во: считаем, что все приоритеты различны. Обозначим:
x_k - вершина с k-ым по порядку ключом;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \ - \text{ предок } x_j \\ 0, \text{ иначе} \end{cases}$$

$d(v)$ - глубина вершины v;

Тогда: $d(x_k) = \sum\limits_{i=1}^{N} A_{i,k}$.

*точнее: приоритеты - случайный величины с равномерным распределением.

Декартово дерево

Теорема: пусть есть декартово дерево из N элементов, в котором приоритеты выбраны случайным образом*. Тогда высота дерева в среднем составляет O(logN).

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \;-\; \text{предок } x_j \\ 0, & \text{иначе} \end{cases}$$

$d(v)$ - глубина вершины v;

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

*точнее: приоритеты - случайный величины с равномерным распределением.

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \text{ — предок } x_j \\ 0, & \text{иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \text{ — предок } x_k] = ?$$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \ - \text{ предок } x_j \\ 0, & \text{иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \ - \text{ предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \text{ — предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \text{ — предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \dots, x_k\}$ или $\{x_k, x_{k+1}, \dots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \text{ — предок } x_j \\ 0, & \text{иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \text{ — предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Док-во: если x_i - корень дерева, то его приоритет наименьший из всех в дереве, а следовательно и из всех в $X_{i,k}$ (и он точно предок x_k).

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \text{ — предок } x_j \\ 0, & \text{иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \text{ — предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Док-во: если x_i - корень дерева, то его приоритет наименьший из всех в дереве, а следовательно и из всех в $X_{i,k}$ (и он точно предок x_k).
Если x_k - корень, то у x_i приоритет больше и он точно не предок.

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \ - \text{ предок } x_j \\ 0, & \text{иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \ - \text{ предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Док-во: если x_i - корень дерева, то его приоритет наименьший из всех в дереве, а следовательно и из всех в $X_{i,k}$ (и он точно предок x_k).

Если x_k - корень, то у x_i приоритет больше и он точно не предок.

Если корень - один из элементов из $X_{i,k}$ между i и k, то x_i и x_k находятся в разных поддеревьях и не являются предками друг друга (у x_i не мин. приор)

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \; - \text{ предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \; - \text{ предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Док-во: наконец, если корень дерева вообще не входит в $X_{i,k}$, то x_i и x_k находятся в одном из поддеревьев выходящих из этого корня.

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \text{ — предок } x_j \\ 0, & \text{иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \text{ — предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Док-во: наконец, если корень дерева вообще не входит в $X_{i,k}$, то x_i и x_k находятся в одном из поддеревьев выходящих из этого корня. Тогда повторим рассуждения для поддерева (гарантированно меньшего размера).

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \ - \text{ предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \ - \text{ предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Док-во: наконец, если корень дерева вообще не входит в $X_{i,k}$, то x_i и x_k находятся в одном из поддеревьев выходящих из этого корня. Тогда повторим рассуждения для поддерева (гарантированно меньшего размера). В крайнем случае дойдем до поддерева состоящего только из $X_{i,k}$. В нем x_i является предком x_k только тогда, когда, x_i корень (с наим. приорит.) $\square$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, & \text{если } x_i \text{ — предок } x_j \\ 0, & \text{иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \text{ — предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \ - \text{ предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \ - \text{ предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Раз приоритеты распределены равномерно, то вероятность того, что в $X_{i,k}$ именно x_i имеет наименьший приоритет равна $\frac{1}{|X_{i,k}|}$, и $i > k \Rightarrow |X_{i,k}| = i - k + 1$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \ - \text{ предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k}] = Pr[x_i \ - \text{ предок } x_k] = ?$$

Введем: $X_{i,k} = \{x_i, x_{i+1}, \ldots, x_k\}$ или $\{x_k, x_{k+1}, \ldots, x_i\}$ (в зависимости от того, что больше i или k).

Лемма: для любых $i \neq k$ верно: x_i является предком x_k тогда и только тогда, когда x_i имеет наименьший приоритет в $X_{i,k}$

Раз приоритеты распределены равномерно, то вероятность того, что в $X_{i,k}$ именно x_i имеет наименьший приоритет равна $\dfrac{1}{|X_{i,k}|}$, и $i > k \Rightarrow |X_{i,k}| = i - k + 1$

Тогда:

$$Pr[A_{i,k} = 1] = \begin{cases} \frac{1}{i-k+1}, \text{ если } i > k, \\ 0, \text{ если } i = k, \\ \frac{1}{k-i+1}, \text{ если } i < k \end{cases}$$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \ - \text{ предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k} = 1] = \begin{cases} \frac{1}{i-k+1}, \text{ если } i > k, \\ 0, \text{ если } i = k, \\ \frac{1}{k-i+1}, \text{ если } i < k \end{cases}$$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \ - \text{ предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k} = 1] = \begin{cases} \frac{1}{i-k+1}, \text{ если } i > k, \\ 0, \text{ если } i = k, \\ \frac{1}{k-i+1}, \text{ если } i < k \end{cases}$$

$$\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} Pr[A_{i,k} = 1] = \sum_{i=1}^{k} \frac{1}{k-i+1} + \sum_{i=k+1}^{N} \frac{1}{i-k+1}$$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \ - \text{ предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

$$Pr[A_{i,k} = 1] = \begin{cases} \frac{1}{i-k+1}, \text{ если } i > k, \\ 0, \text{ если } i = k, \\ \frac{1}{k-i+1}, \text{ если } i < k \end{cases}$$

после замены это k-ое и n-k-ое гармонические числа, а для них известно, что они ограничены ln(x) + 1

$$\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} Pr[A_{i,k} = 1] = \sum_{i=1}^{k} \frac{1}{k-i+1} + \sum_{i=k+1}^{N} \frac{1}{i-k+1} \le ln(k) + ln(N-k) + 2$$

Док-во: считаем, что все приоритеты различны. Обозначим:

x_k - вершина с k-ым по порядку ключом;

$d(v)$ - глубина вершины v;

$$A_{i,j} = \begin{cases} 1, \text{ если } x_i \text{ — предок } x_j \\ 0, \text{ иначе} \end{cases}$$

Тогда: $d(x_k) = \sum_{i=1}^{N} A_{i,k}$. И тогда $\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} \mathbb{E}[A_{i,k}] = \sum_{i=1}^{N} Pr[A_{i,k} = 1]$

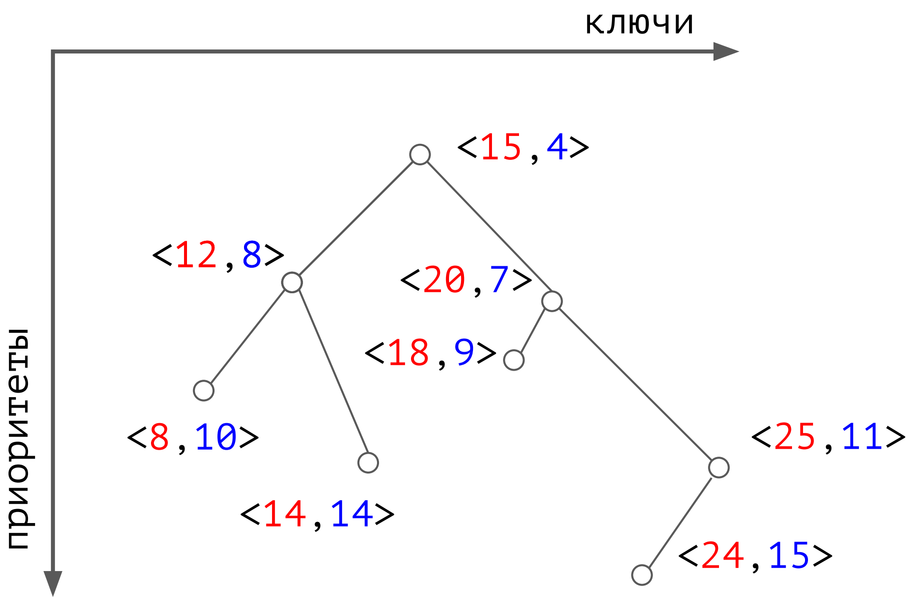
$$Pr[A_{i,k} = 1] = \begin{cases} \frac{1}{i-k+1}, \text{ если } i > k, \\ 0, \text{ если } i = k, \\ \frac{1}{k-i+1}, \text{ если } i < k \end{cases}$$

после замены это k-ое и n-k-ое гармонические числа, а для них известно, что они ограничены ln(x) + 1

$$\mathbb{E}[d(x_k)] = \sum_{i=1}^{N} Pr[A_{i,k} = 1] = \sum_{i=1}^{k} \frac{1}{k-i+1} + \sum_{i=k+1}^{N} \frac{1}{i-k+1} \leq ln(k) + ln(N-k) + 2$$

$$\mathbb{E}[height(T)] = \mathbb{E}[\max_k d(x_k)] = O(logN) \quad \square$$

Декартово дерево: операции



Итого: все операции снова зависят от O(heigh)

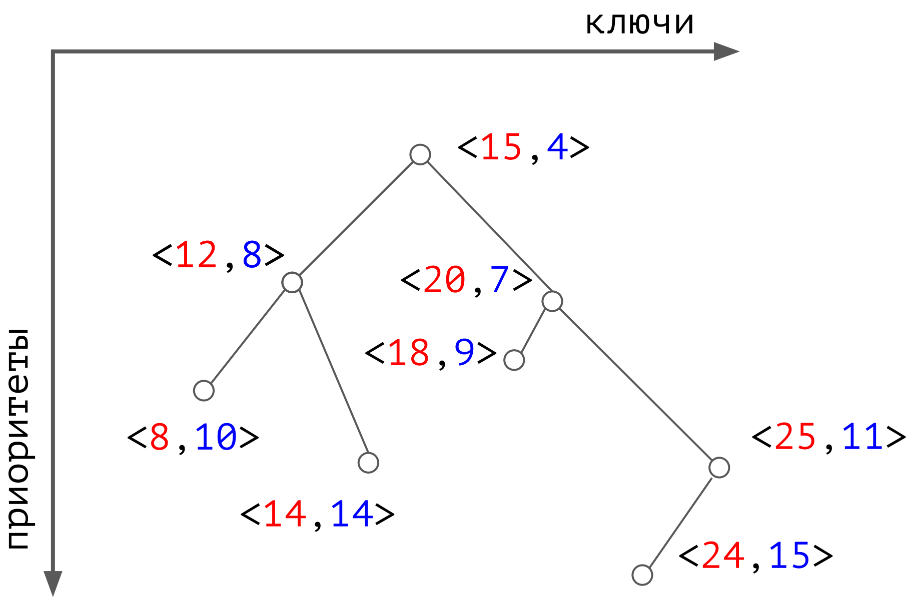
1. операция `split(T, k)`: разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция `merge(T1, T2)`: даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция `insert(T, x)`: вставить пару с ключом x.

4. операция `remove(T, x)`: удалить элемент с ключом x.

Декартово дерево: операции



1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

3. операция insert(T, x): вставить пару с ключом x.

4. операция remove(T, x): удалить элемент с ключом x.

Итого: все операции работают за O(logN) в среднем.

Декартово дерево: выводы

Минусы декартовых деревьев:

Декартово дерево: выводы

Минусы декартовых деревьев:

- Сложность в среднем (бывают худшие случаи)
- Потребление памяти (как минимум под приоритеты)

Декартово дерево: выводы

Минусы декартовых деревьев:

- Сложность в среднем (бывают худшие случаи)
- Потребление памяти (как минимум под приоритеты)

Плюсы декартовых деревьев:

Декартово дерево: выводы

Минусы декартовых деревьев:

- Сложность в среднем (бывают худшие случаи)
- Потребление памяти (как минимум под приоритеты)

Плюсы декартовых деревьев:

- Очень просто писать!

Декартово дерево: выводы

Минусы декартовых деревьев:

- Сложность в среднем (бывают худшие случаи)
- Потребление памяти (как минимум под приоритеты)

Плюсы декартовых деревьев:

- Очень просто писать!
- На базе декартовых можно делать продвинутые структуры данных.

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

$$a_1, a_2, a_3, \ldots, a_n$$

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

$$a_1, a_2, a_3, \dots, a_n$$

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

$$a_1, a_2, a_3, \ldots, a_{(pos-1)}, a_{pos}, \ldots, a_n$$

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. sum(from, to) - посчитать сумму элементов с a_from до a_to

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. sum(from, to) - посчитать сумму элементов с a_from до a_to

---

Реализовать можно по-разному, но мы подумаем про декартово дерево.

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

$$a_1, a_2, a_3, \dots, a_{(pos-1)}, a_{pos}, \dots, a_n$$

Пусть каждому элементу массива соответствует вершина в декартовом дереве.

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

$$a_1,\ a_2,\ a_3,\ \ldots,\ a_{(pos-1)},\ a_{pos},\ \ldots,\ a_n$$

Пусть каждому элементу массива соответствует вершина в декартовом дереве. При этом: приоритеты генерируются случайно, как обычно,

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

$$a_1,\ a_2,\ a_3,\ \dots,\ a_{(pos-1)},\ a_{pos},\ \dots,\ a_n$$

Пусть каждому элементу массива соответствует вершина в декартовом дереве. При этом: приоритеты генерируются случайно, как обычно, а вот ключи - это порядковые номера элементов в массиве.

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

Пусть каждому элементу массива соответствует вершина в декартовом дереве. При этом: приоритеты генерируются случайно, как обычно, а вот ключи - это порядковые номера элементов в массиве.

При наивной реализации придется обновлять ключи после вставки.

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

Пусть каждому элементу массива соответствует вершина в декартовом дереве. При этом: приоритеты генерируются случайно, как обычно, а вот ключи - это порядковые номера элементов в массиве.

При наивной реализации придется обновлять ключи после вставки. Линейный проход себе позволить не можем.

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

$$a\_1,\ a\_2,\ a\_3,\ \ldots,\ a\_(pos-1),\ val,\ a\_pos,\ \ldots,\ a\_n$$

Пусть каждому элементу массива соответствует вершина в декартовом дереве. При этом: приоритеты генерируются случайно, как обычно, а вот ключи - это порядковые номера элементов в массиве.

При наивной реализации придется обновлять ключи после вставки. Линейный проход себе позволить не можем. Поэтому мы вообще не будем явно хранить ключи!

$a_1, a_2, a_3, \ldots, a_{(pos-1)}, a_{pos}, \ldots, a_n$

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_{pos}
2. Приоритет: y

$$a_1, a_2, a_3, \dots, a_{(pos-1)}, a_{pos}, \dots, a_n$$

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_{pos}
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_pos
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C

as: 5 24 42 13 99 2 17

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_pos
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C

as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10 ← сгенерированные приоритеты

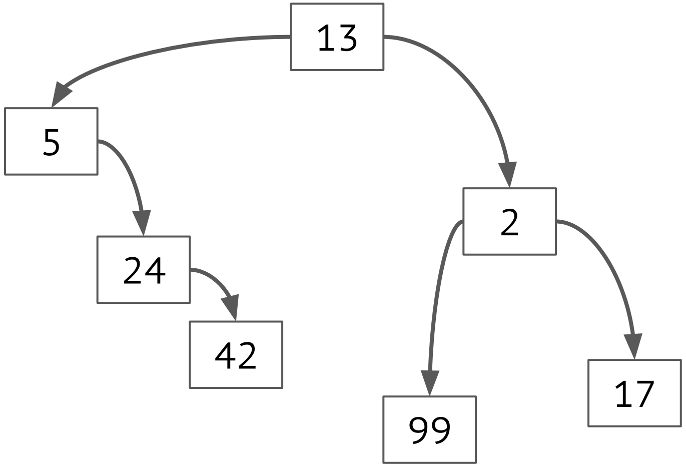
a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_pos
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C

as: 5 24 42 13 99 2 17

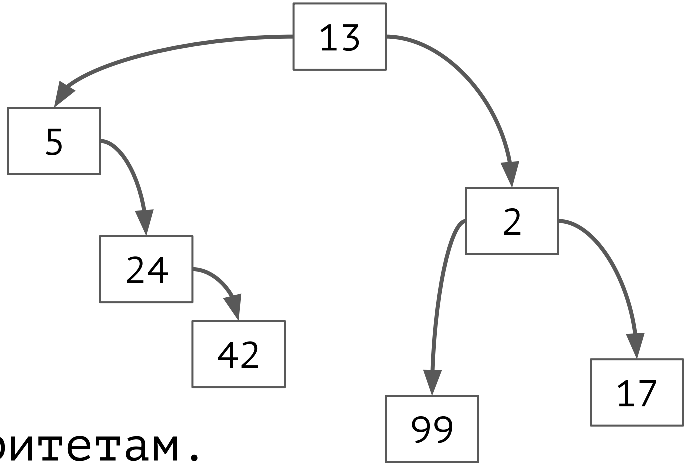
ys: 6 8 9 4 11 7 10



a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_pos
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C



as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

Т.е. по x-ам стоят в том же порядке,
что и в массиве, а по y-ам - по приоритетам.

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

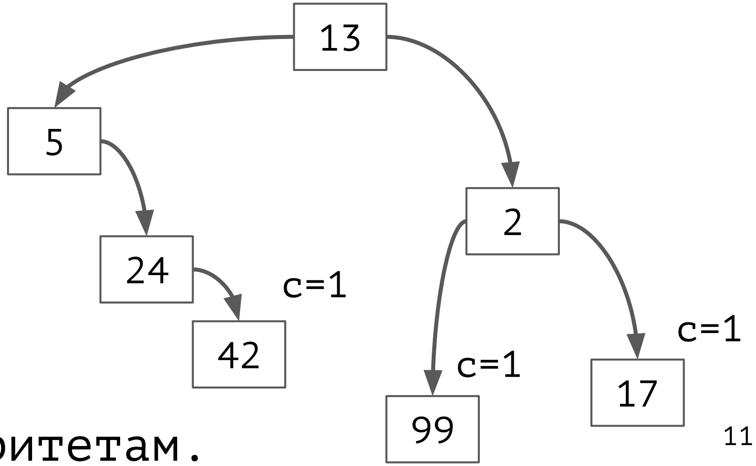
Каждая вершина v дерева хранит:

1. Значение элемента массива: a_pos
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C

as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

Т.е. по x-ам стоят в том же порядке, что и в массиве, а по y-ам - по приоритетам.



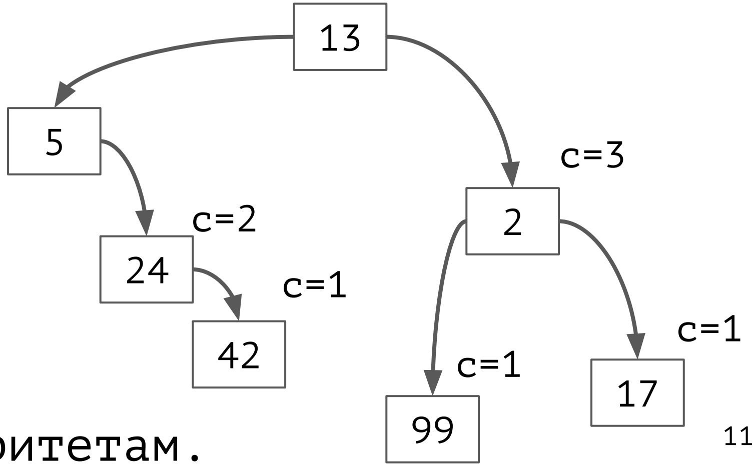
a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_pos
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C

as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

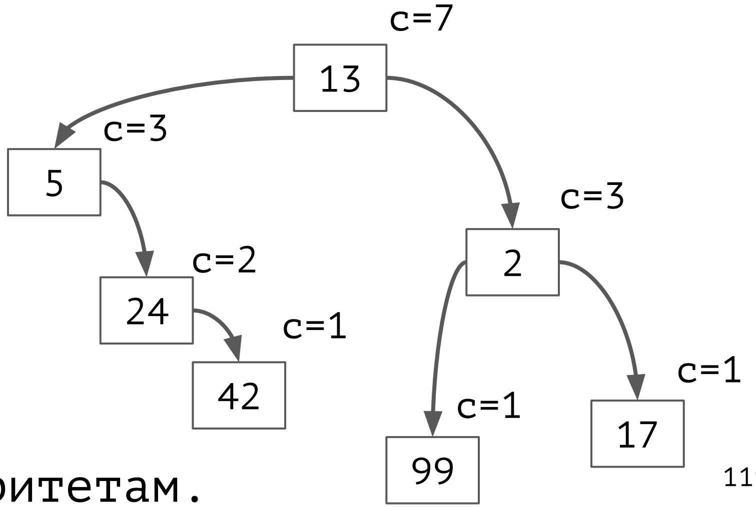


Т.е. по x-ам стоят в том же порядке, что и в массиве, а по y-ам - по приоритетам.

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_pos
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C



as: 5 24 42 13 99 2 17

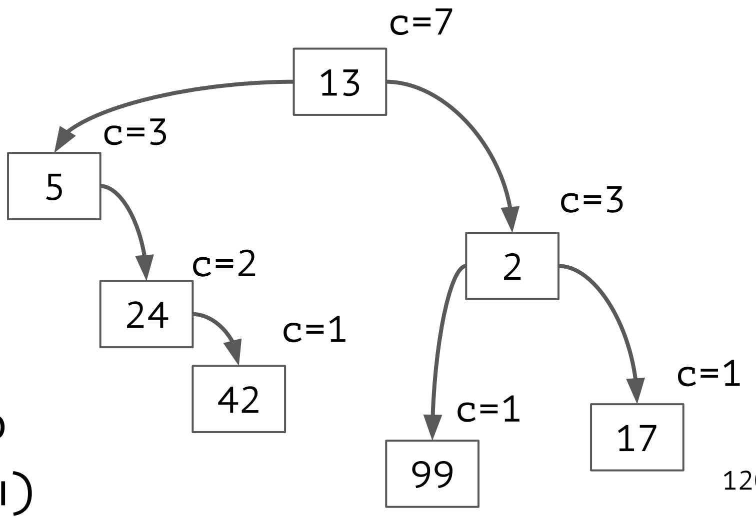
ys: 6 8 9 4 11 7 10

Т.е. по x-ам стоят в том же порядке, что и в массиве, а по y-ам - по приоритетам.

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Каждая вершина v дерева хранит:

1. Значение элемента массива: a_pos
2. Приоритет: y
3. Количество элементов в поддереве (включая вершину): C



as: 5 24 42 13 99 2 17

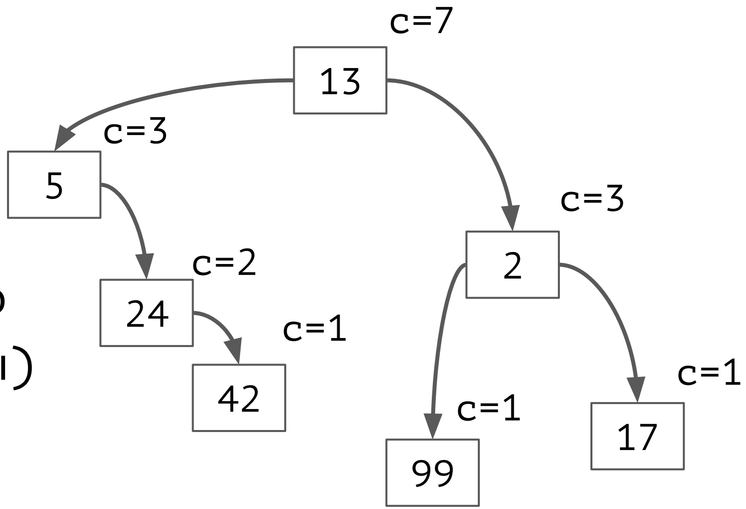
ys: 6 8 9 4 11 7 10

Ключ явно не хранится (хотя его можно получить проходом от корня до вершины)

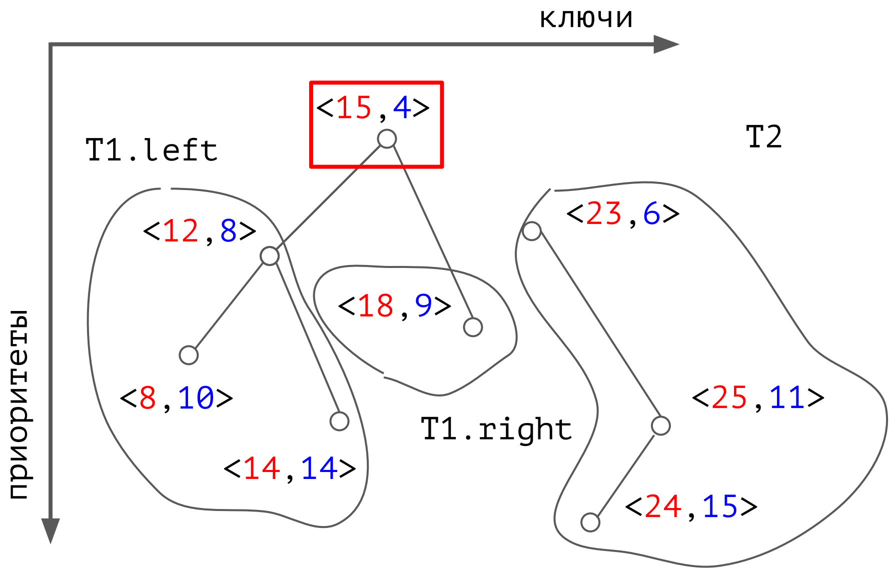
as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

Ключ явно не хранится (хотя его можно получить проходом от корня до вершины)



Как это влияет на операции с декартовым деревом?




```python
def merge(T1, T2: Treap) -> Treap:
  if not T1: return T2
  if not T2: return T1

  if T1.root.p < T2.root.p:
    T1.right = merge(T1.right, T2)
    return T1
  else:
    T2.left = merge(T1, T2.left)
    return T2
```

1. операция split(T, k): разрезать дерево на два так, чтобы в левом ключи были меньше k, а в правом - больше либо равны.

2. операция merge(T1, T2): даны два декартова дерева T1, T2, при этом известно, что все ключи T1 меньше всех ключей T2. Объединить их в одно декартово дерево.

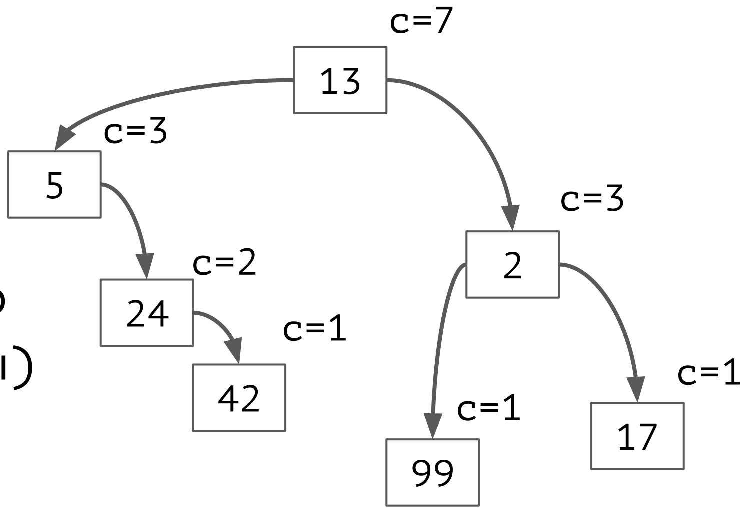
   Корнем нового дерева станет либо корень T1 либо корень T2. По определению берем корнем того, у кого приоритет меньше.

   В нашем случае в качестве левого поддерева берем T1.left, в качестве правого merge(T1.right, T2)

as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

Ключ явно не хранится (хотя его можно получить проходом от корня до вершины)



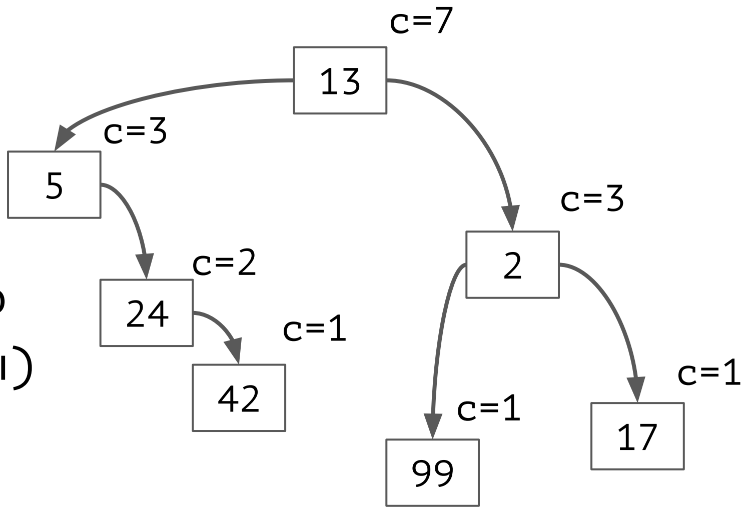
Как это влияет на операции с декартовым деревом?

1. merge вообще не смотрит на ключи, только на приоритеты! Поэтому в нем изменений нет.

as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

Ключ явно не хранится (хотя его можно получить проходом от корня до вершины)



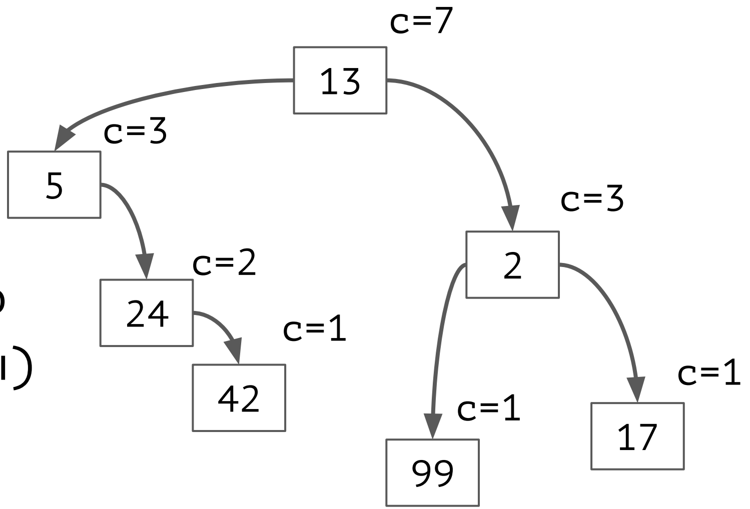
Как это влияет на операции с декартовым деревом?

1. merge вообще не смотрит на ключи, только на приоритеты! Поэтому в нем изменений нет (кроме обновлений c).

as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

Ключ явно не хранится (хотя его можно получить проходом от корня до вершины)



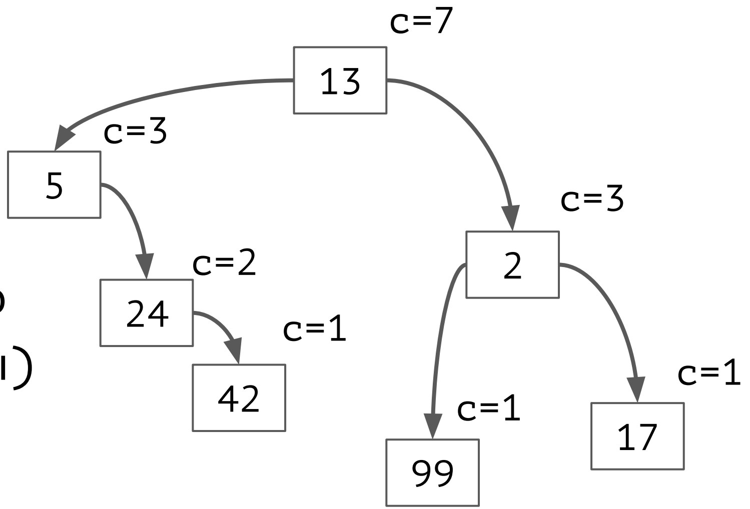
Как это влияет на операции с декартовым деревом?

1. merge - как в обычном декартовом.

as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

Ключ явно не хранится (хотя его можно получить проходом от корня до вершины)



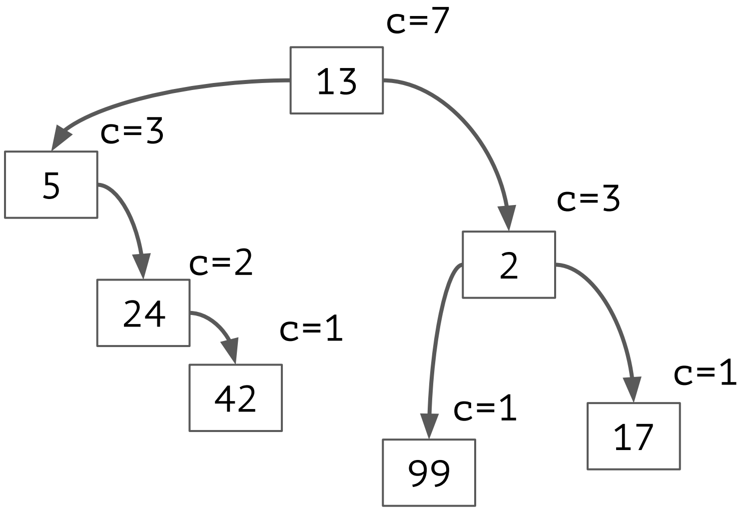
Как это влияет на операции с декартовым деревом?

1. merge - как в обычном декартовом.

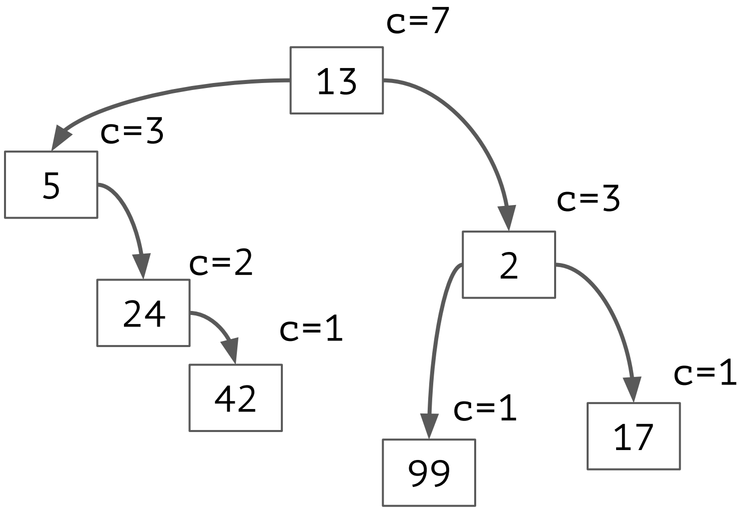
2. split заменяется на splitBySize(T, k): выделить первые k элементов в первое дерево, что осталось - унести во второе.

as: 5 24 42 13 99 2 17

ys: 6 8 9 4 11 7 10

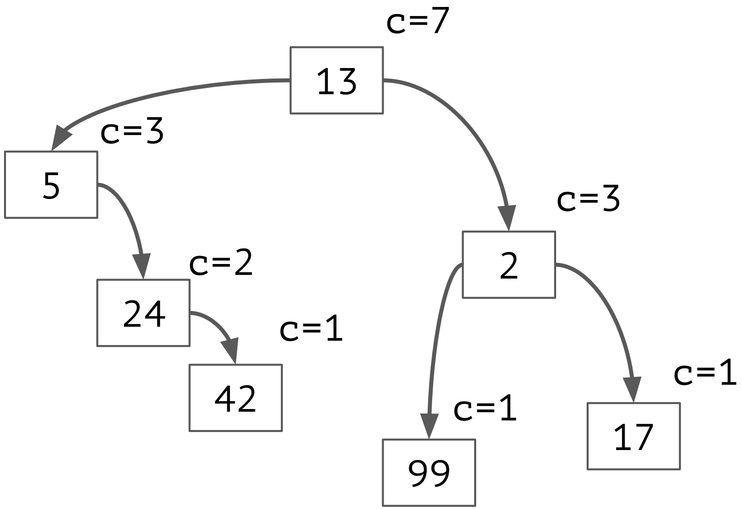


```
as: 5 24 42 13 99 2 17

ys: 6  8  9  4 11 7 10

splitBySize(2)?
```



```
as: 5 24    42 13 99 2 17

ys: 6  8     9  4 11 7 10
```

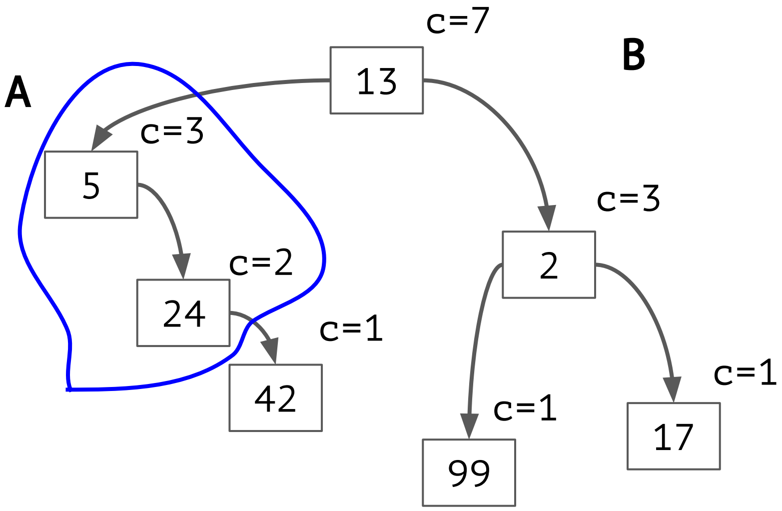
splitBySize(2)?



```
as: 5 24   42 13 99 2 17

ys: 6  8    9  4 11 7 10
```

splitBySize(2)?

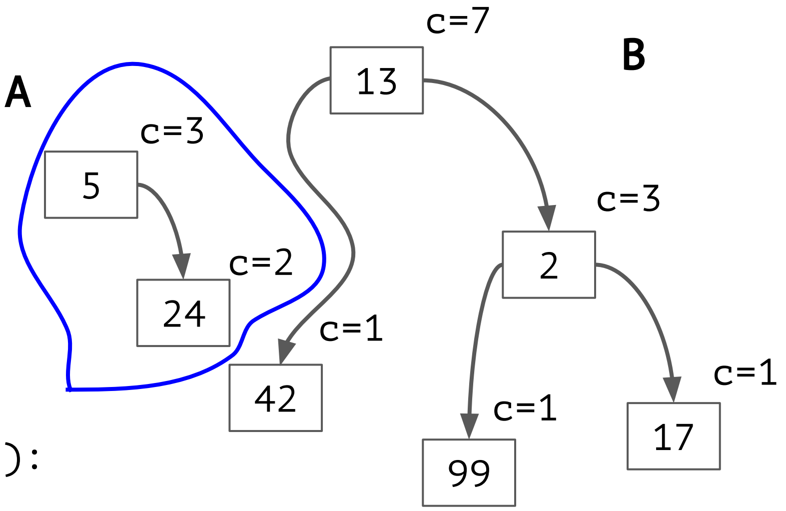
```python
def split(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)
  if k > T.root.key:
    RL, RR = split(T.root.right, k)
    T.right = RL
    return (T, RR)
  else
    LL, LR = split(T.root.left, k)
    T.left = LR
    return (LL, T)
```



as: 5 24] [42 13 99 2 17

ys: 6 8] [9 4 11 7 10

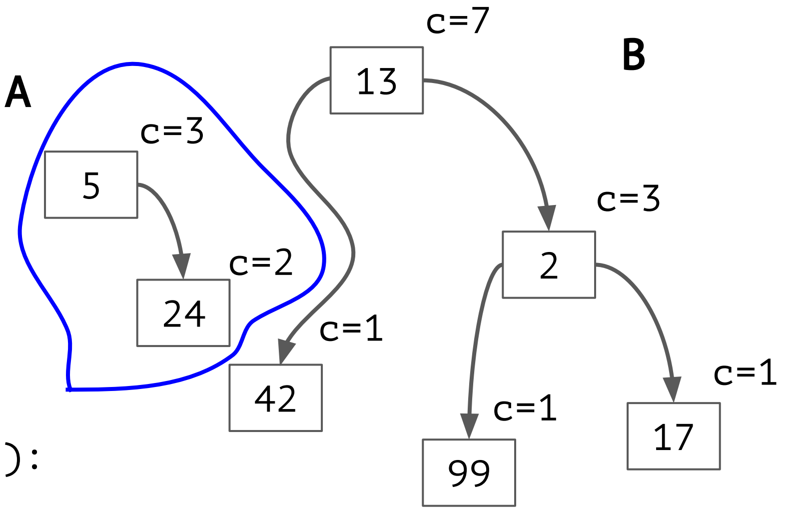
splitBySize(2)?




```
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
```

as:	5	24	42	13	99	2	17
ys:	6	8	9	4	11	7	10

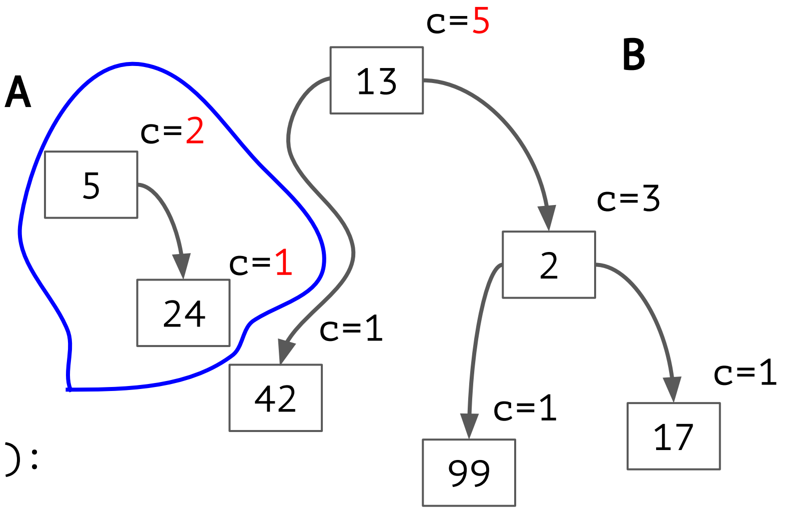
splitBySize(2)?




```
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
    update(T)
```

```
as: 5 24     42 13 99 2 17

ys: 6  8      9  4 11 7 10
```

splitBySize(2)?

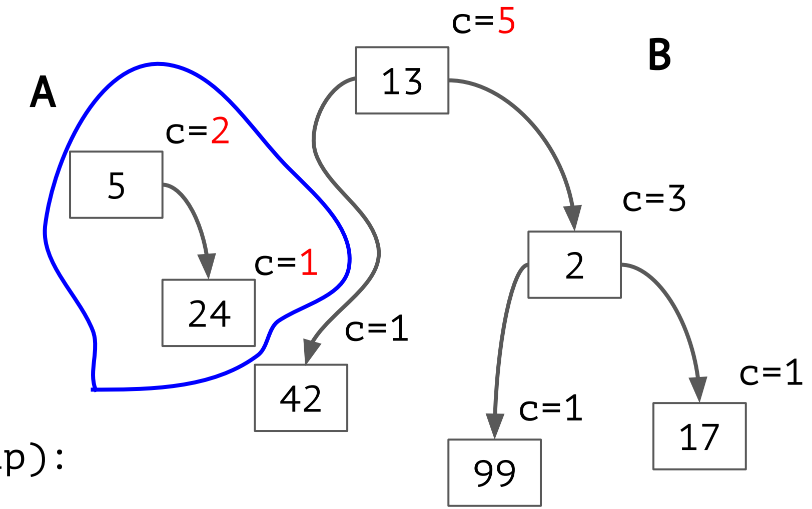



```
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
    update(T)
```

```
def update(T: Treap):
  // обработка краевых случаев
  T.root.c = 1 + T.left.c + T.right.c
```

```
as: 5 24]   [42 13 99 2 17

ys: 6  8]   [ 9  4 11 7 10
```

splitBySize(2)?

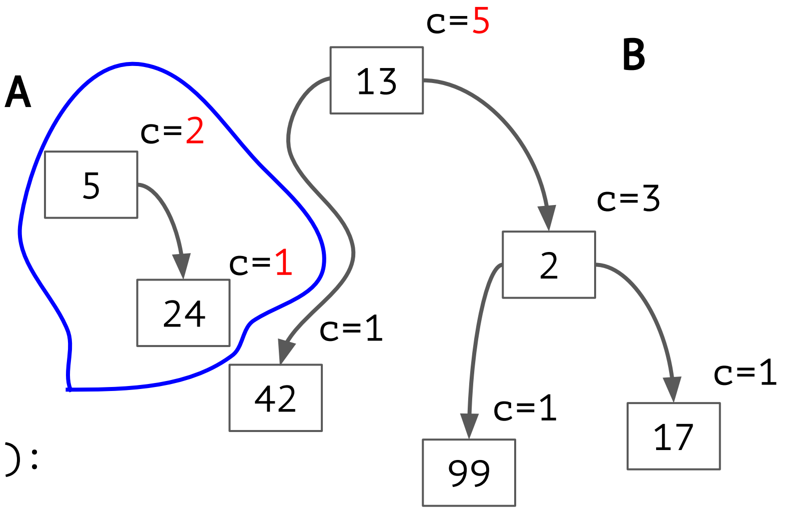



```python
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
    update(T)
    return LL, T
```

```python
def update(T: Treap):
  // обработка краевых случаев
  T.root.c = 1 + T.left.c + T.right.c
```

```
as: 5 24     42 13 99 2 17

ys: 6  8      9  4 11 7 10
```

splitBySize(2)?

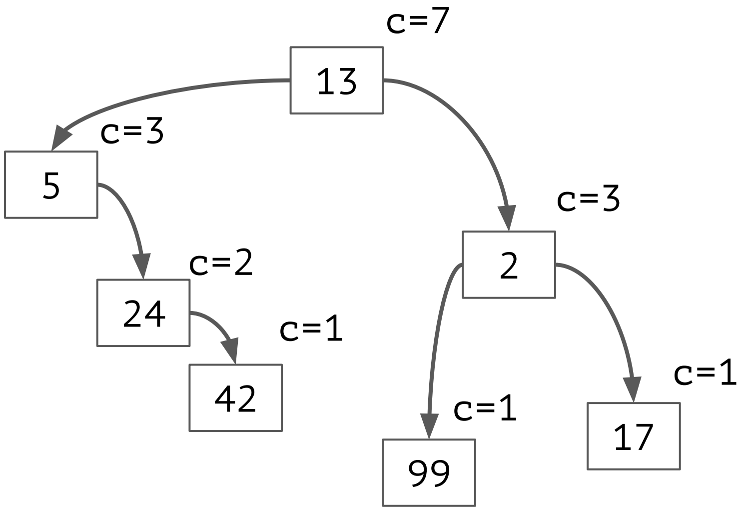



```
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
    update(T)
    return LL, T
  else:
```

```
def update(T: Treap):
  // обработка краевых случаев
  T.root.c = 1 + T.left.c + T.right.c
```

```
as: 5 24 42 13 99 ]   [2 17

ys: 6  8  9  4 11 ]   [7 10
```

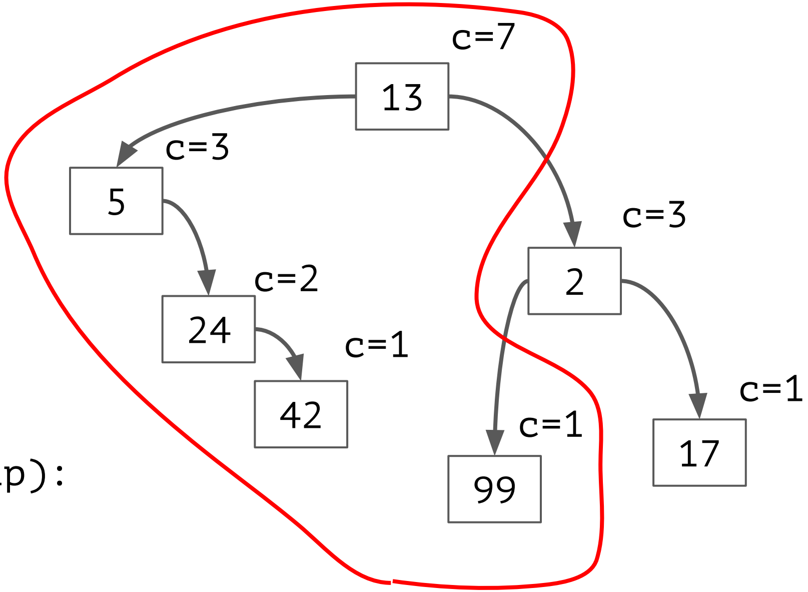
splitBySize(5)?




```
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
    update(T)
    return LL, T
  else:
```

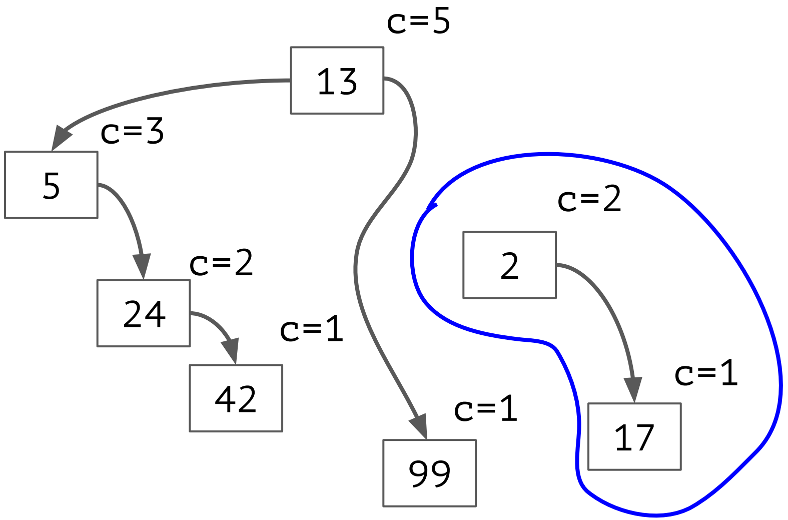
```
as: 5 24 42 13 99 ]  [ 2 17

ys: 6  8  9  4 11 ]  [ 7 10
```

splitBySize(5)?




```python
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
    update(T)
    return LL, T
  else:
```

```
as: 5 24 42 13 99 ]  [ 2 17
ys: 6  8  9  4 11 ]  [ 7 10
```



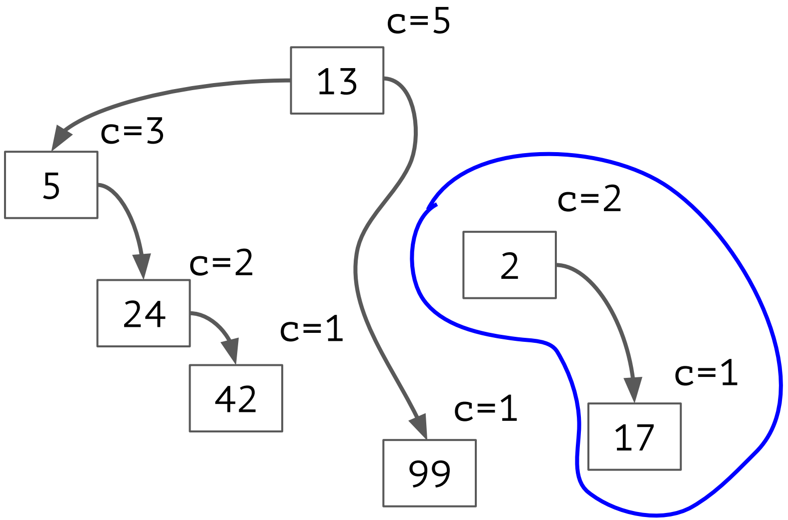
splitBySize(5)?

```
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
    update(T)
    return LL, T
  else:
    RL, RR = splitBySize(T.right, k - T.left.c - 1)
    T.right = RL
    update(T)
```

as: 5 24 42 13 99] [2 17

ys: 6 8 9 4 11] [7 10

splitBySize(5)?




```python
def splitBySize(T: Treap, k) -> (Treap, Treap):
  if not T: return (None, None)

  if k <= T.root.left.c:
    LL, LR = splitBySize(T.left, k)
    T.left = LR
    update(T)
    return LL, T
  else:
    RL, RR = splitBySize(T.right, k - T.left.c - 1)
    T.right = RL
    update(T)
    return T, RR
```

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. sum(from, to) - посчитать сумму элементов с a_from до a_to

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. sum(from, to) - посчитать сумму элементов с a_from до a_to

Получили структуру данных, в которой можно:

1. Выделять префикс нужной длины в новое дерево
2. Сливать два массива/дерева

Неявное декартово дерево

Задача: пусть есть динамический массив (с доступом к i-ому элементу, добавлением и удалением элементов из конца).

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

Необходимо поддержать новые операции следующего вида:

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. sum(from, to) - посчитать сумму элементов с a_from до a_to

Получили структуру данных, в которой можно:
1. Выделять префикс нужной длины в новое дерево
2. Сливать два массива/дерева (в любом порядке!!!)

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos:

Неявное декартово дерево

```
a_1, a_2, a_3, …, a_(pos-1) ]   [a_pos, …, a_n
```

1. `insert(val, pos)` - вставка элемента на позицию pos:

   a) `splitBySize(T, pos - 1)`

Неявное декартово дерево

```
a_1, a_2, a_3, …, a_(pos-1) ]  val  [ a_pos, …, a_n
```

```
1. insert(val, pos) - вставка элемента на позицию pos:

   a) splitBySize(T, pos - 1)
   b) добавили новое неявное дерево из {val}
```

Неявное декартово дерево

```
a_1, a_2, a_3, …, a_(pos-1) val] [a_pos, …, a_n
```

1. `insert(val, pos)` - вставка элемента на позицию pos:

   a) `splitBySize(T, pos - 1)`
   b) добавили новое неявное дерево из `{val}`
   c) merge x2

Неявное декартово дерево

```
a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n
```

1. insert(val, pos) - вставка элемента на позицию pos:

   a) splitBySize(T, pos - 1)
   b) добавили новое неявное дерево из {val}
   c) merge x2

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos:

```
a) L,R = splitBySize(T, pos - 1)
b) E,RR = splitBySize(R, 1)
c) merge(L, RR)
```

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos

Неявное декартово дерево

```
a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n
```

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos

```
a) L,R = splitBySize(T, pos - 1)
b) RL,RR = splitBySize(R, count)
c) merge(L, RR)
```

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos

4. sum(from, to)?

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos

4. sum(from, to)?

Здесь чуть хитрее: в каждой вершине нужно хранить еще и сумму всех элементов поддерева. Соответственно ее обновлять во всех операциях.

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos

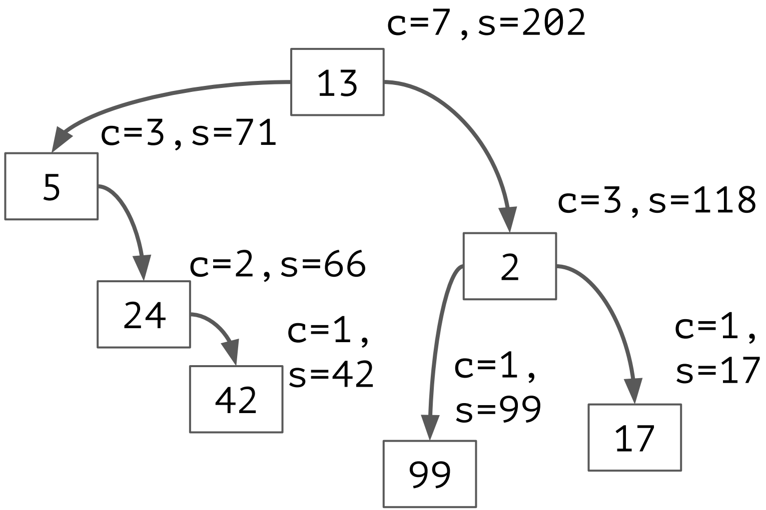
4. sum(from, to):

   a) L,R = splitBySize(T, from - 1)
   b) RL,RR = splitBySize(R, to - from + 1)
   c) res = RL.sum
   d) собираем дерево обратно через merge
   e) return res

Мини-задача #42 (2 балла)

Реализовать неявное дерево с поддержкой функции sum(from, to), которая возвращает сумму элементов массива от from до to включительно.

Не забывайте о тестах для вашего решения!



Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos
4. sum(from, to)

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos
4. sum(from, to), другие операции на интервалах

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos
4. sum(from, to), другие операции на интервалах
5. циклический сдвиг массива на k!
6. …

Неявное декартово дерево

a_1, a_2, a_3, …, a_(pos-1), val, a_pos, …, a_n

1. insert(val, pos) - вставка элемента на позицию pos
2. erase(pos) - удаление элемента с позиции pos
3. erase(pos, count): удалить регион из count элементов, начиная с pos
4. sum(from, to), другие операции на интервалах
5. циклический сдвиг массива на k!
6. …

И все это за O(logN) в среднем!

Takeaways

- Сбалансированные бинарные деревья поиска - не единственный способ решать проблему поиска/вставки/удаления элемента

- Вероятностный подход в декартовых деревьях => сложность только в среднем, зато просто писать

- Неявные декартовы деревья для операций с интервалами в массивах