Алгоритмы и структуры данных

Лекция 8

Декартовы деревья

Декартово дерево

Это бинарное дерево, в узлах которого хранятся пары (*x*, *y*), где *x* - это ключ, а *y*- это приоритет.

Оно является:

- двоичным деревом поиска по *x*,
- пирамидой по *y*.

Терминология:

Treap = tree + heap

Дуча = дерево + куча

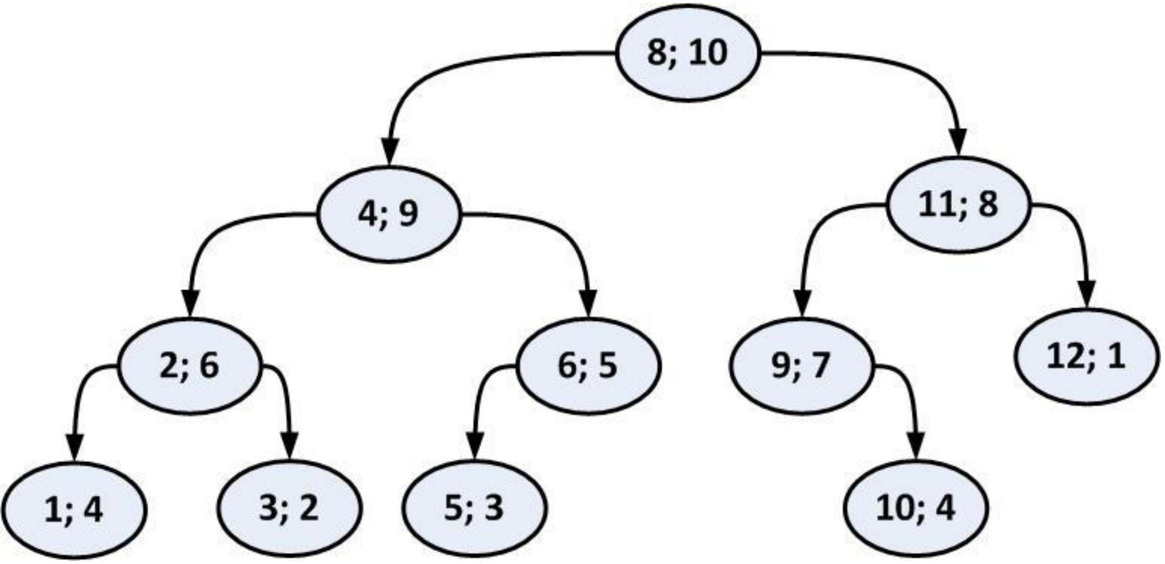
Дерамида = дерево + пирамида

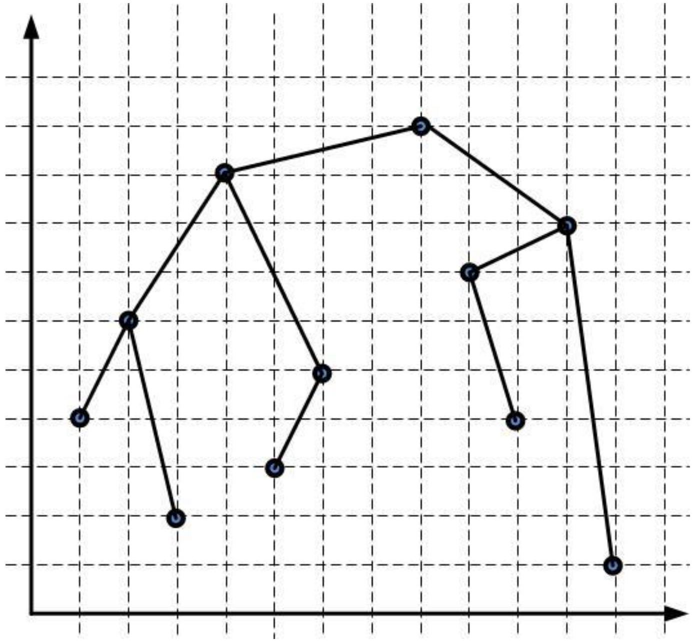
Курево = куча + дерево

Дерамиды были предложены Сиделем (Siedel) и Арагон (Aragon) в 1989 г.

1996 г. – дучи с рандомизированными приоритетами

Пример декартова дерева



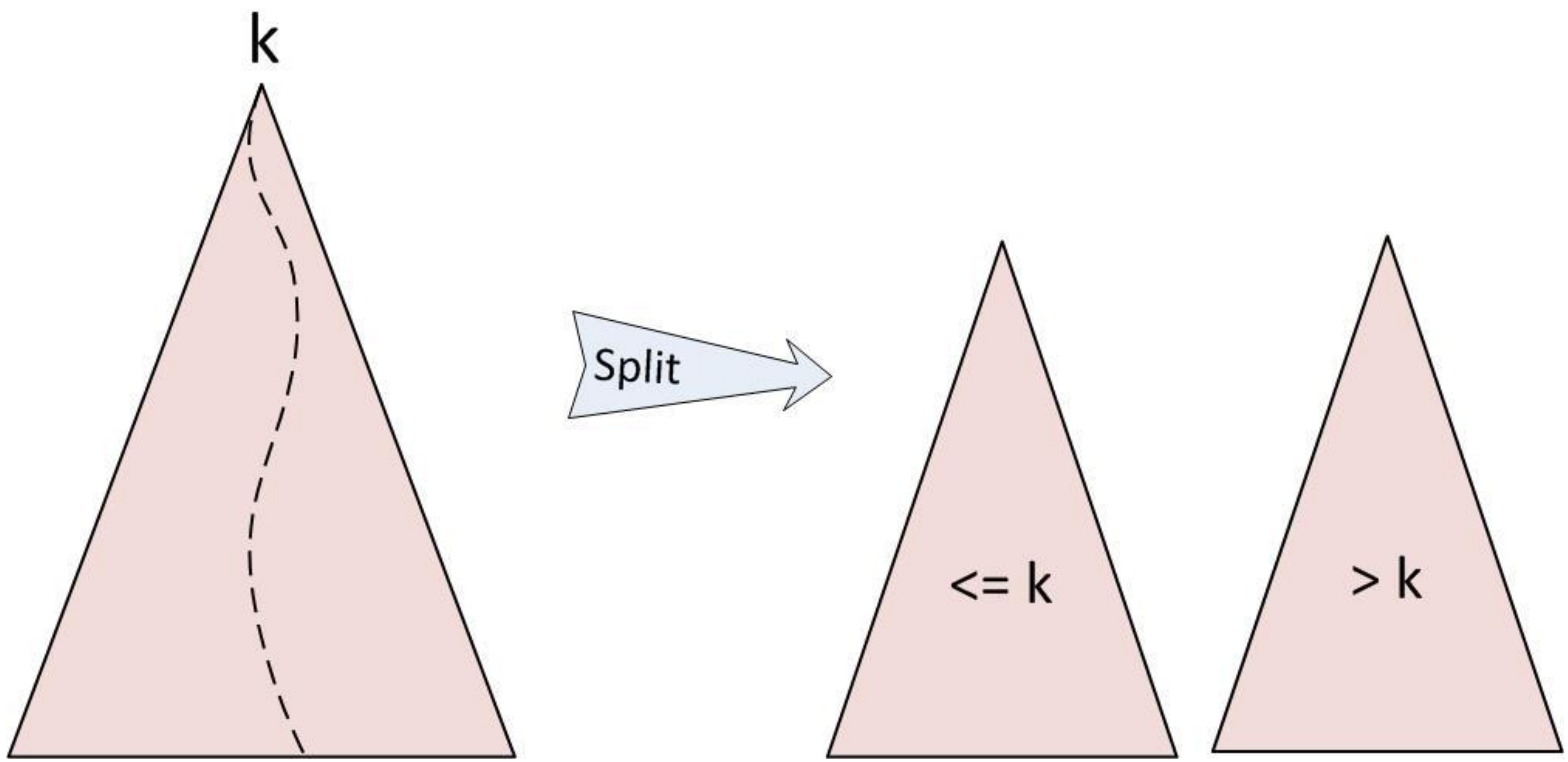


Операции в декартовом дереве

Split

Операция Split (*разрезать*) позволяет сделать следующее:

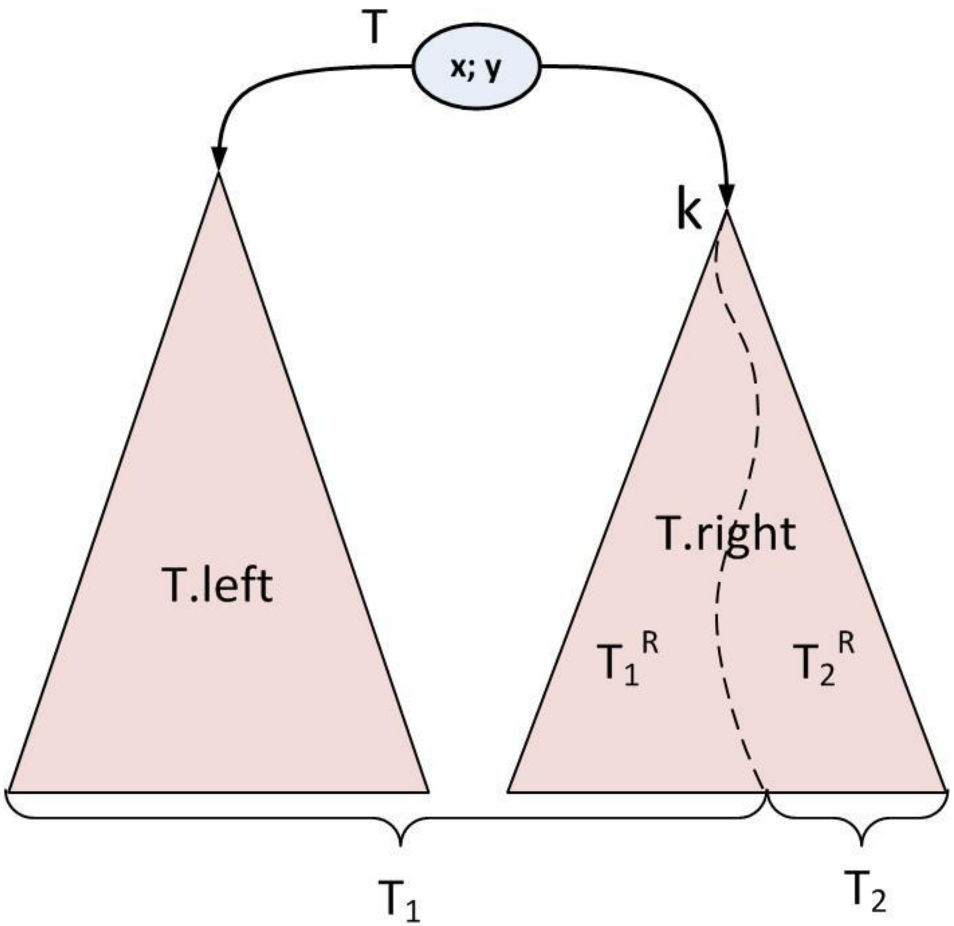
- разрезать декартово дерево T по ключу *k*
- и получить два других декартовых дерева: T_1 и T_2,
- причем в T_1 находятся все ключи дерева T, не большие k,
- а в T_2 — большие k.



Split (T, k) → { T_1, T_2}

Рассмотрим случай, в котором требуется разрезать дерево по ключу, большему ключа корня.

- Левое поддерево T_1 совпадёт с левым поддеревом T.
- Для нахождения правого поддерева T_1, нужно разрезать правое поддерево T на T_1^R иT_2^R по ключу k и взять T_1^R.
- T_2 совпадёт с T_2^R.



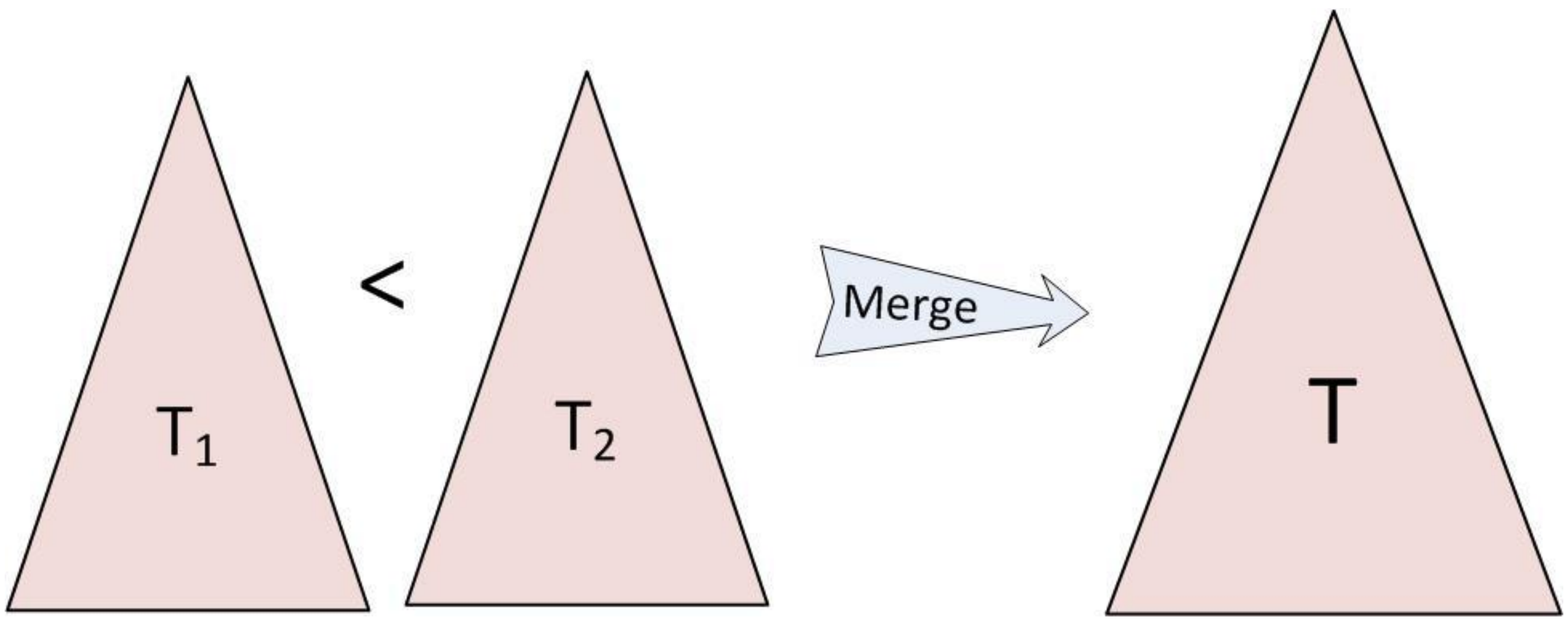
Случай, в котором требуется разрезать дерево по ключу, меньше либо равному ключа в корне, рассматривается симметрично.

Операции в декартовом дереве

Merge (*слить*)

С помощью этой операции можно слить два декартовых дерева в одно.

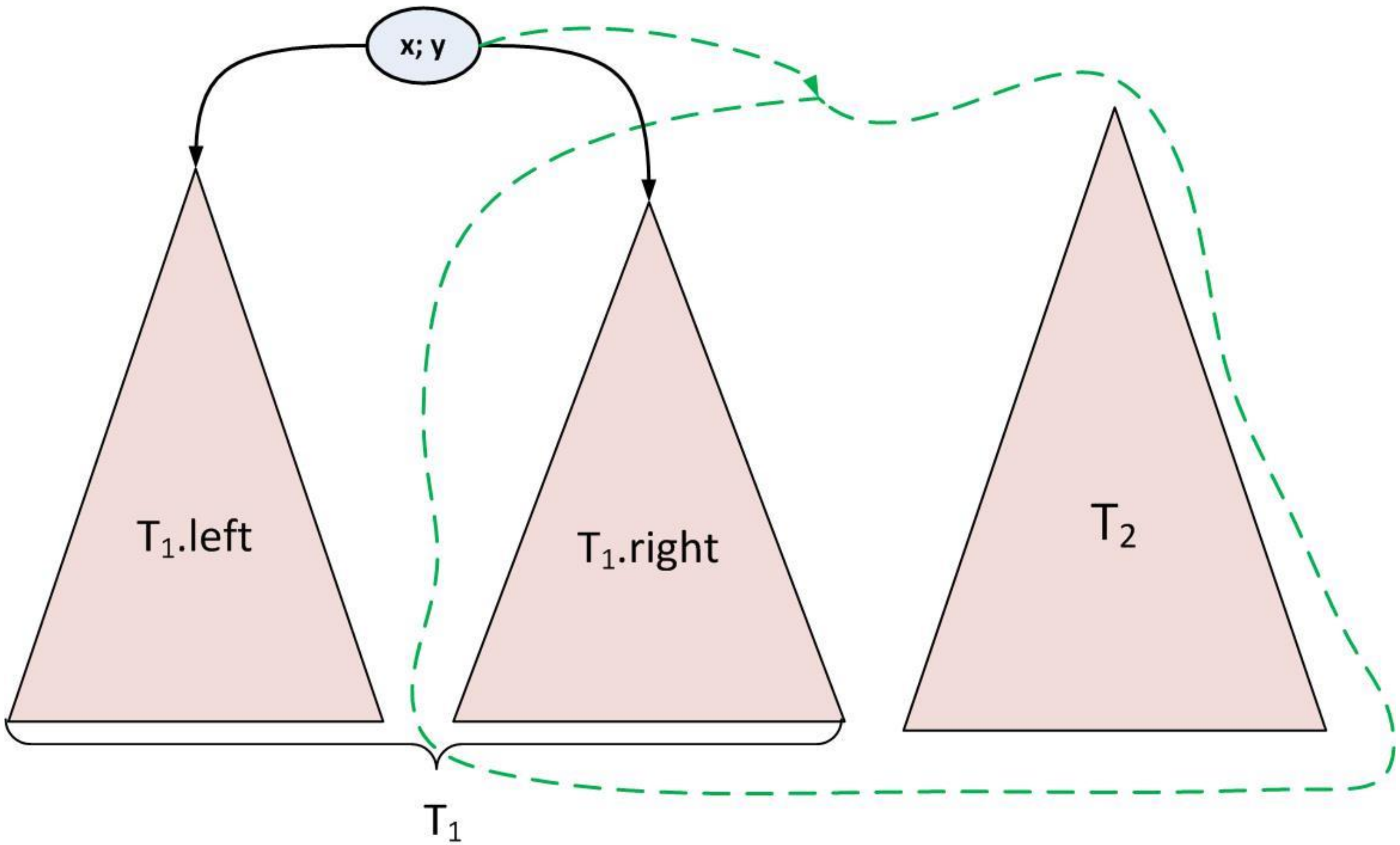
- Причем, все ключи в первом(*левом*) дереве должны быть меньше, чем ключи во втором(*правом*).
- В результате получается дерево, в котором есть все ключи из первого и второго деревьев.



Merge (T_1, T_2) → { T }

Корнем станет вершина из T_1 или T_2 с наибольшим ключом y.
Это либо корень T_1, либо корень T_2.
Пусть корень T_1 имеет больший y, чем корень T_2.

- Корень T_1 станет корнем T.
- Тогда левое поддерево T совпадёт с левым поддеревом T_1.
- Справа же нужно подвесить объединение правого поддерева T_1 и дерева T_2.



Операции в декартовом дереве

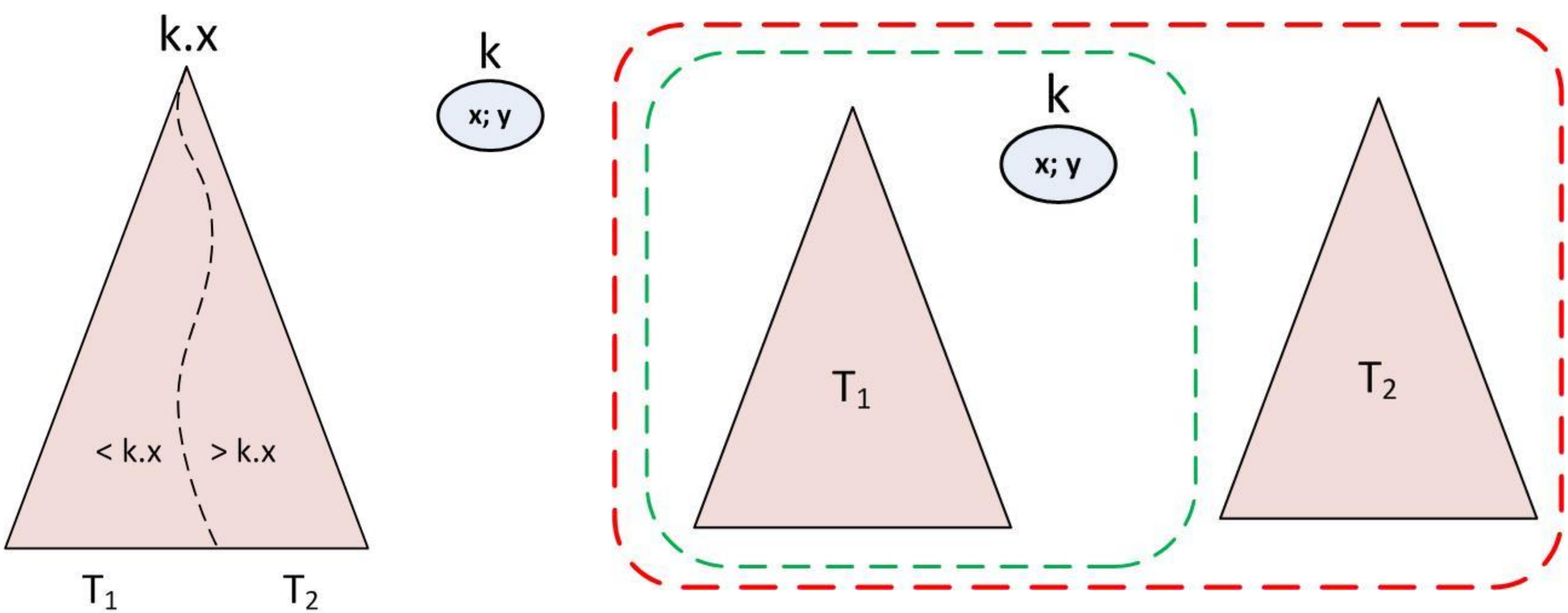
Insert

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

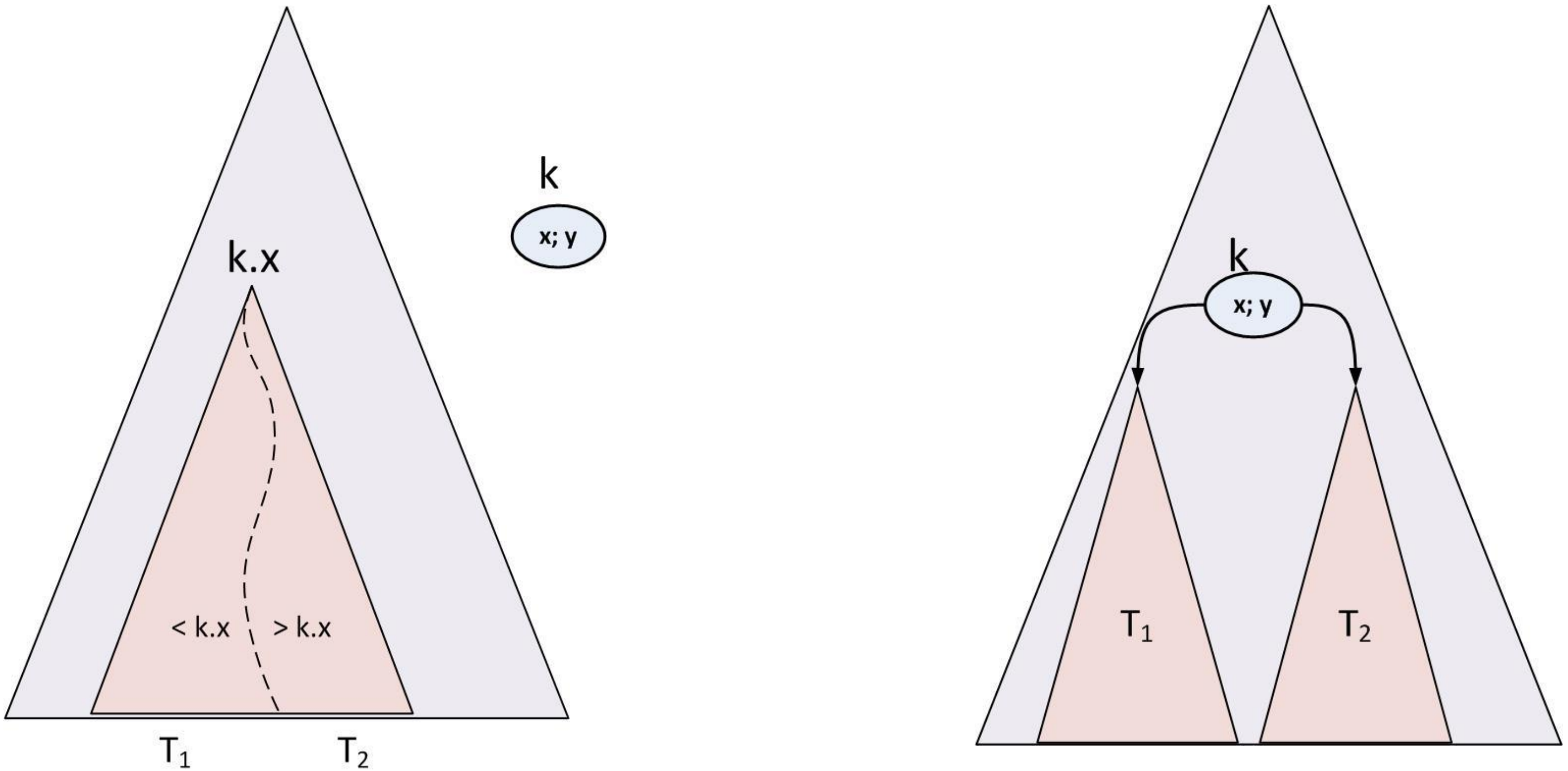
1. Разобьём наше дерево по ключу, который мы хотим добавить: Split (T, k.x) $\to$ { T_1, T_2}.
2. Сливаем первое дерево с новым элементом: Merge (T_1, k) $\to$ { T_1 } .
3. Сливаем получившиеся дерево со вторым: Merge (T_1, T_2) $\to$ { T } .



Insert

Реализация №2

1. 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.
2. Теперь вызываем Split (T, k.x) $\rightarrow$ { T_1, T_2} от найденного элемента (от элемента вместе со всем его поддеревом)
3. Полученные T_1 и T_2 записываем в качестве левого и правого сына добавляемого элемента.
4. Полученное дерево ставим на место элемента, найденного в первом пункте.



Операции в декартовом дереве

Remove

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

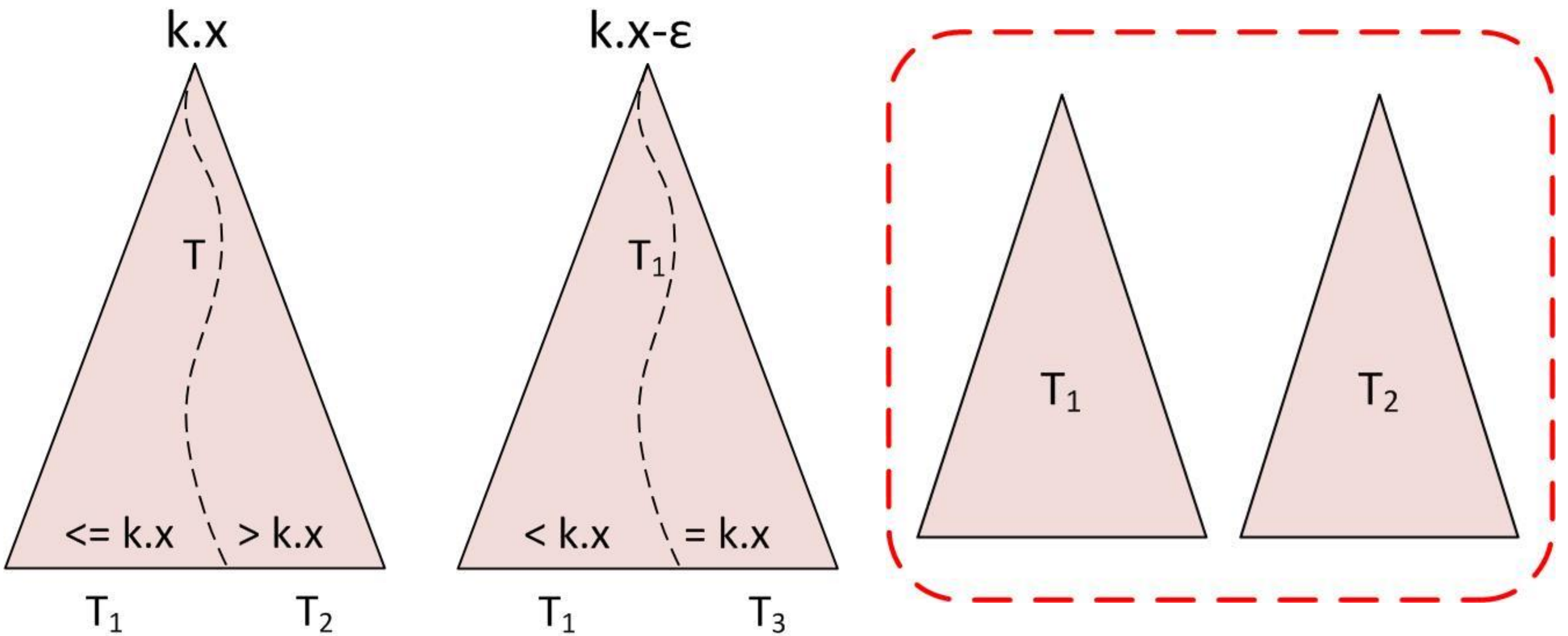
Реализация №1

1. Разобьём наше дерево по ключу, который мы хотим удалить:

$\text{Split}(T, k.x) \to \{T_1, T_2\}$.

2. Теперь отделяем от первого дерева элемент x, опять таки разбивая по ключу x: $\text{Split}(T_1, k.x - \varepsilon) \to \{T_1, T_3\}$.

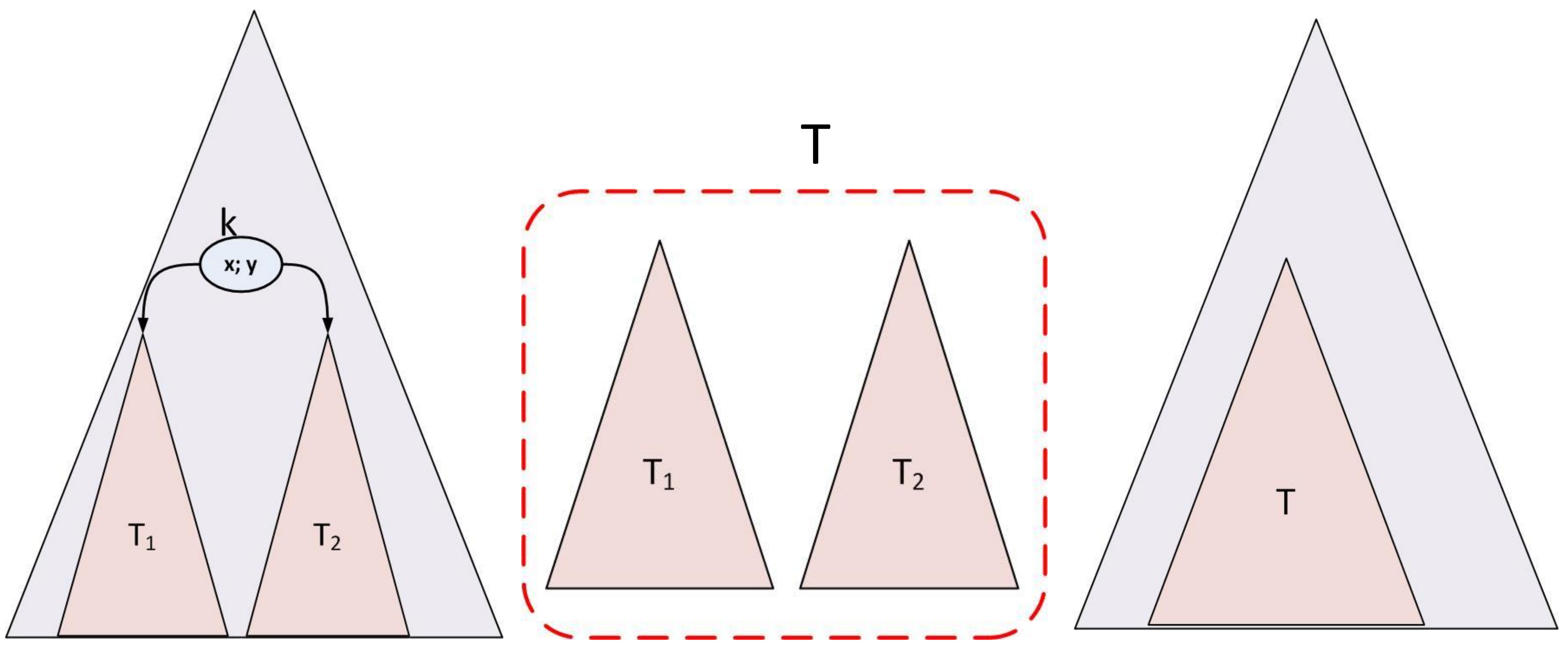
3. Сливаем первое дерево со вторым: $\text{Merge}(T_1, T_2) \to \{T\}$.



Remove

Реализация №2

1. Спускаемся по дереву (как в обычном бинарном дереве поиска по x), ища удаляемый элемент.
2. Найдя элемент, вызываем Merge его левого и правого сыновей
3. Результат процедуры Merge ставим на место удаляемого элемента.



Построение декартова дерева

Пусть нам известно, из каких пар (x_i, y_i) требуется построить декартово дерево, причем также известно, что $x_1 < x_2 < x_3 < \dots < x_n$.

Рекурсивный алгоритм

Рассмотрим приоритеты $y_1, y_2, y_3, \dots, y_n$ и выберем максимум среди них, пусть это будет y_k, и сделаем (x_k, y_k) корнем дерева.

Проделав то же самое с $y_1, y_2, y_3, \dots, y_{k-1}$ и $y_{k+1}, y_{k+2}, \dots, y_n$, получим соответственно левого и правого сына (x_k, y_k).

Такой алгоритм работает за $O(n^2)$.

=> Единственность представления декартова дерева

Построение декартова дерева

Алгоритм за O(n)

Будем строить дерево слева направо, то есть начиная с (x_1, y_1), по (x_n, y_n), при этом помнить последний добавленный элемент (x_k, y_k). Он будет самым правым, так как у него будет максимальный ключ, а по ключам декартово дерево представляет собой двоичное дерево поиска.

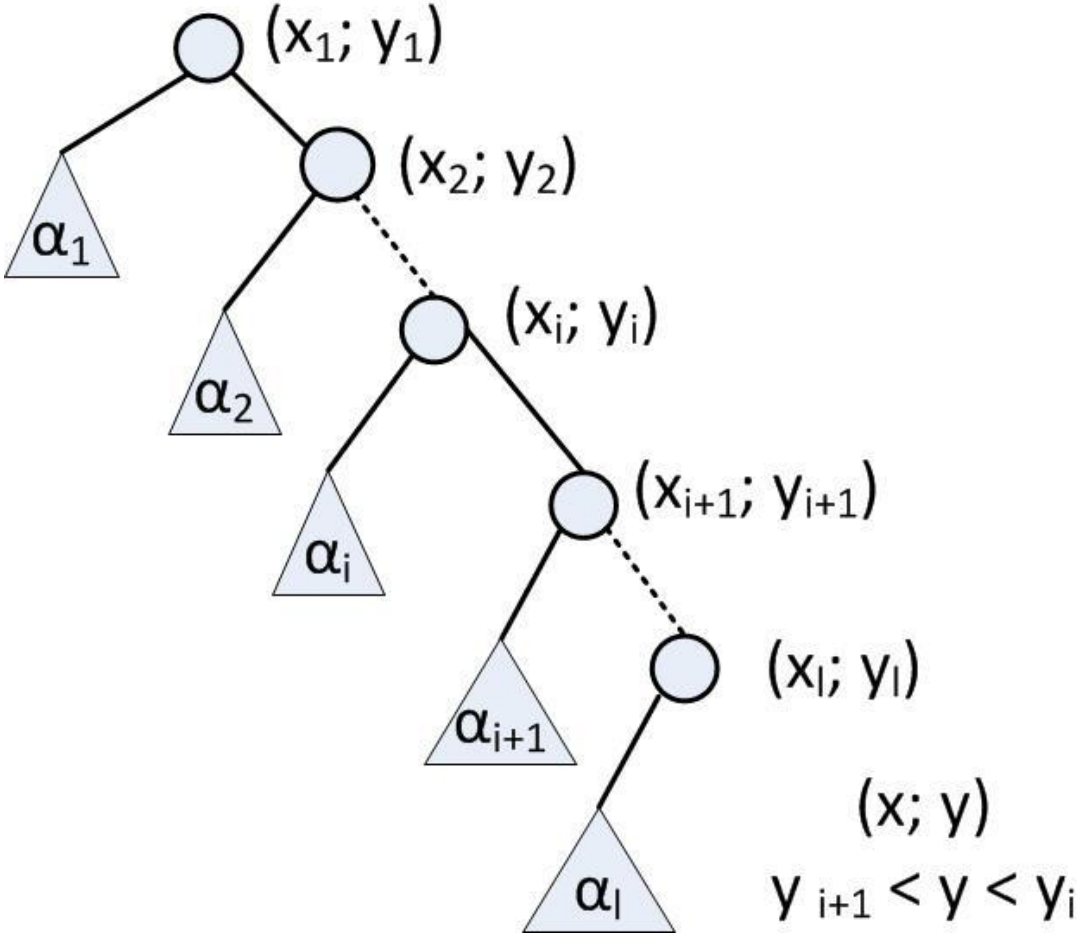
При добавлении (x_{k+1}, y_{k+1}), пытаемся сделать его правым сыном (x_k, y_k), это следует сделать, если $y_k > y_{k+1}$, иначе делаем шаг к предку последнего элемента и смотрим его значение y.

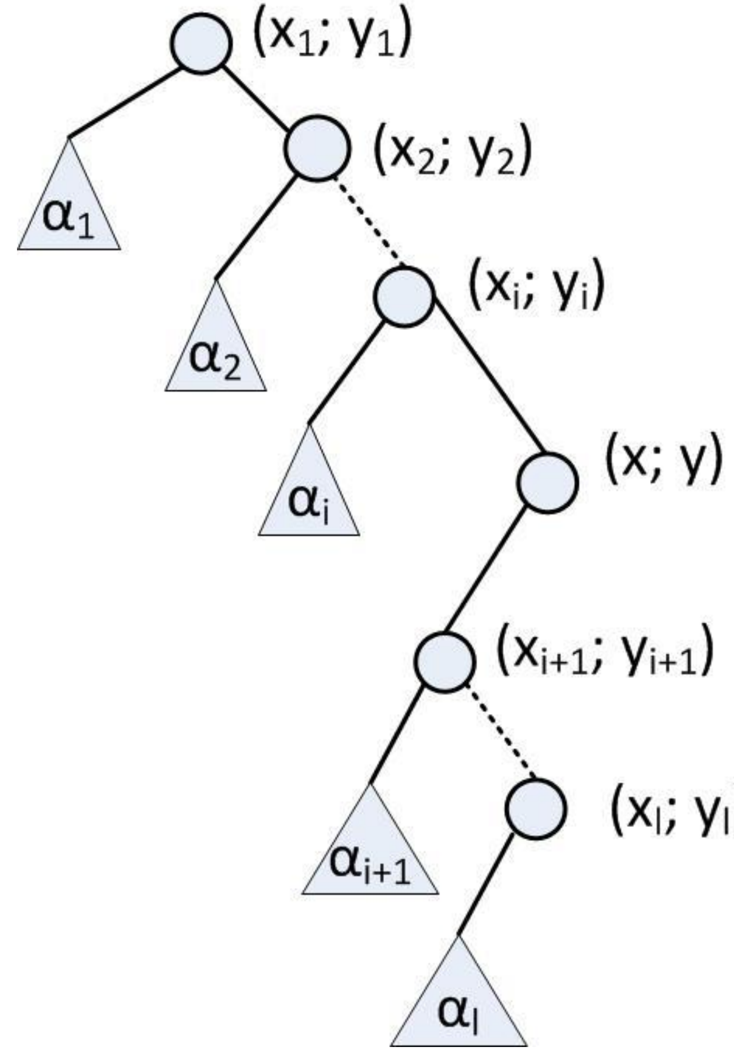
Поднимаемся до тех пор, пока приоритет в рассматриваемом элементе меньше приоритета в добавляемом, после чего делаем (x_{k+1}, y_{k+1}), его правым сыном, а предыдущего правого сына делаем левым сыном (x_{k+1}, y_{k+1}).

Каждую вершину мы посетим максимум дважды: при непосредственном добавлении и, поднимаясь вверх.

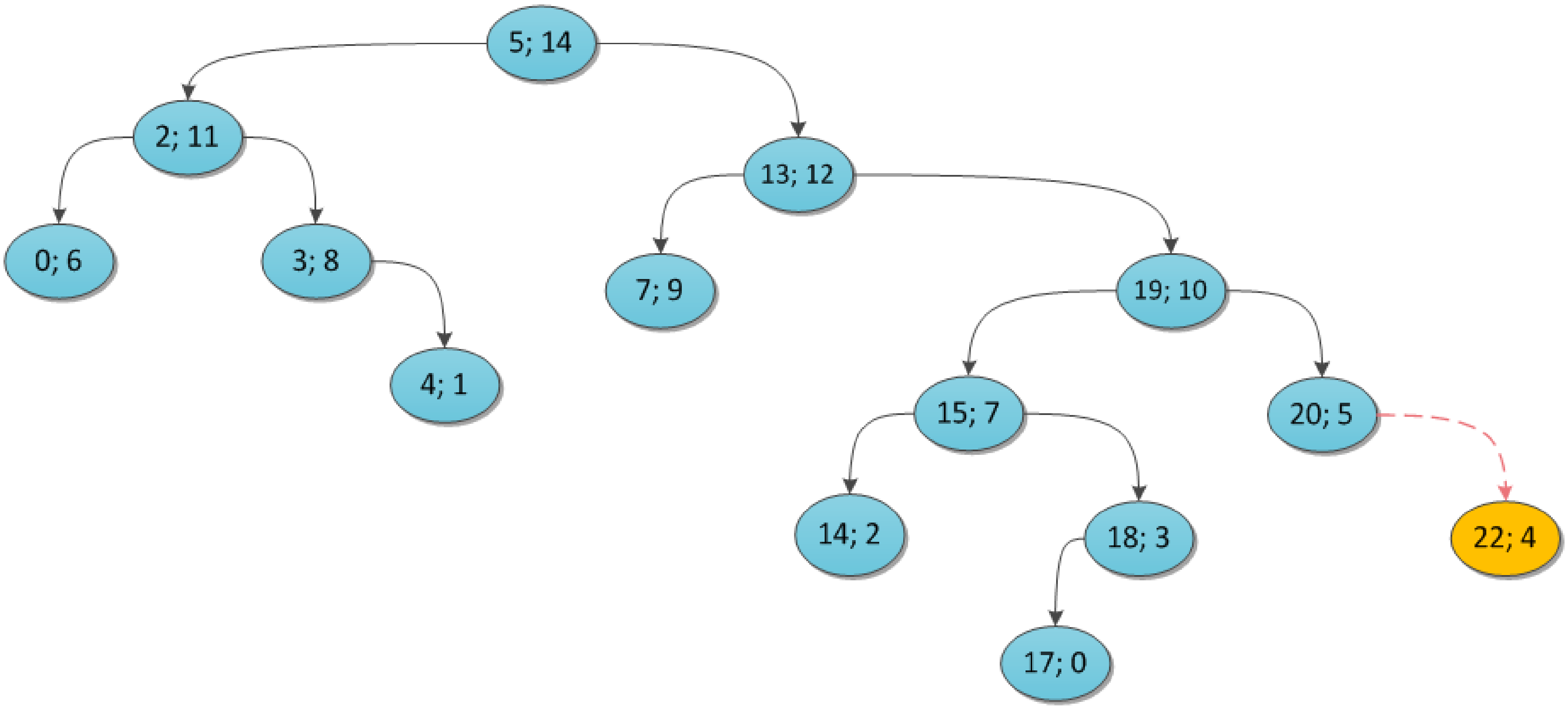
Из этого следует, что построение происходит за O(n).

Алгоритм за O(n)

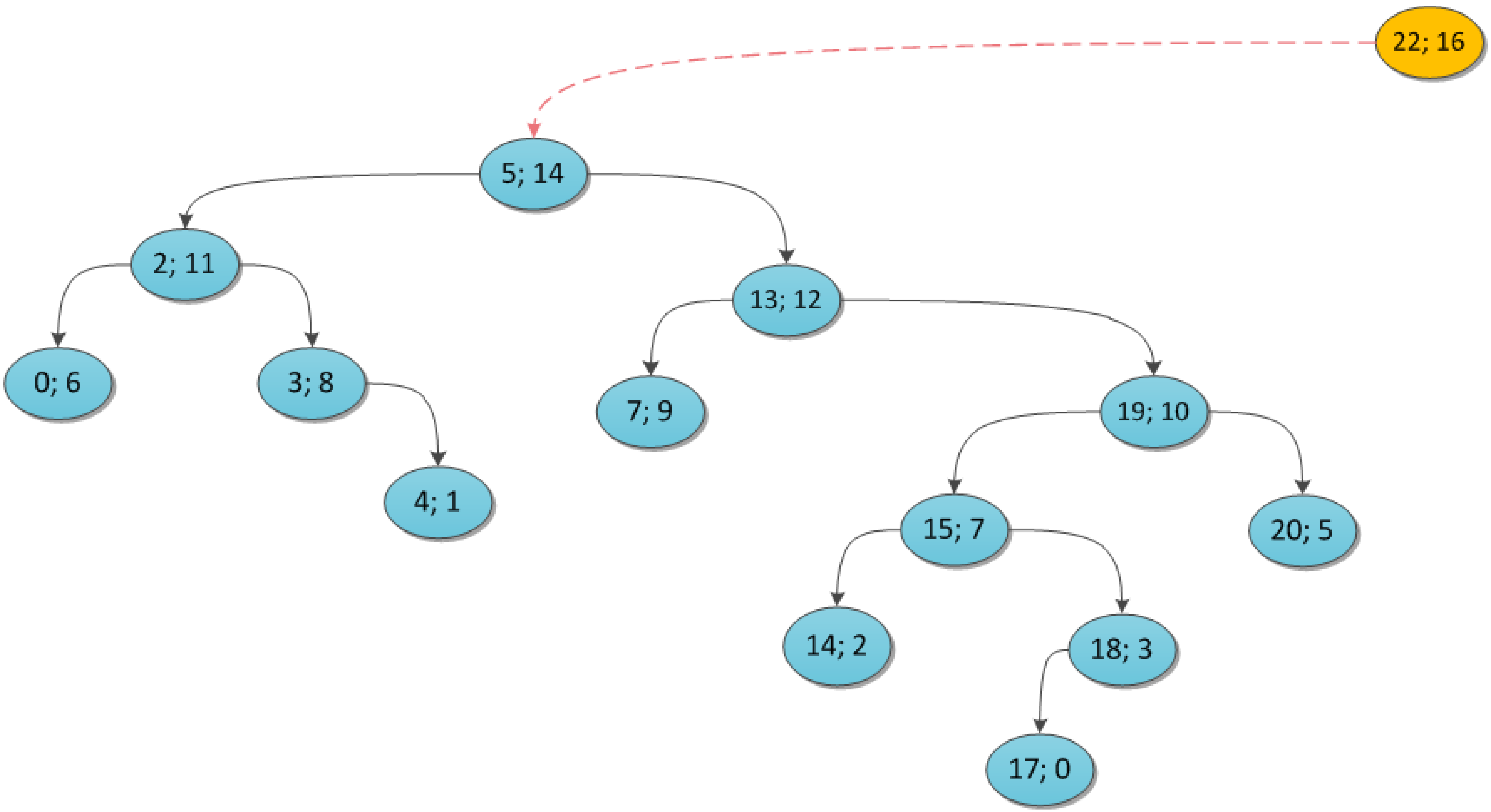




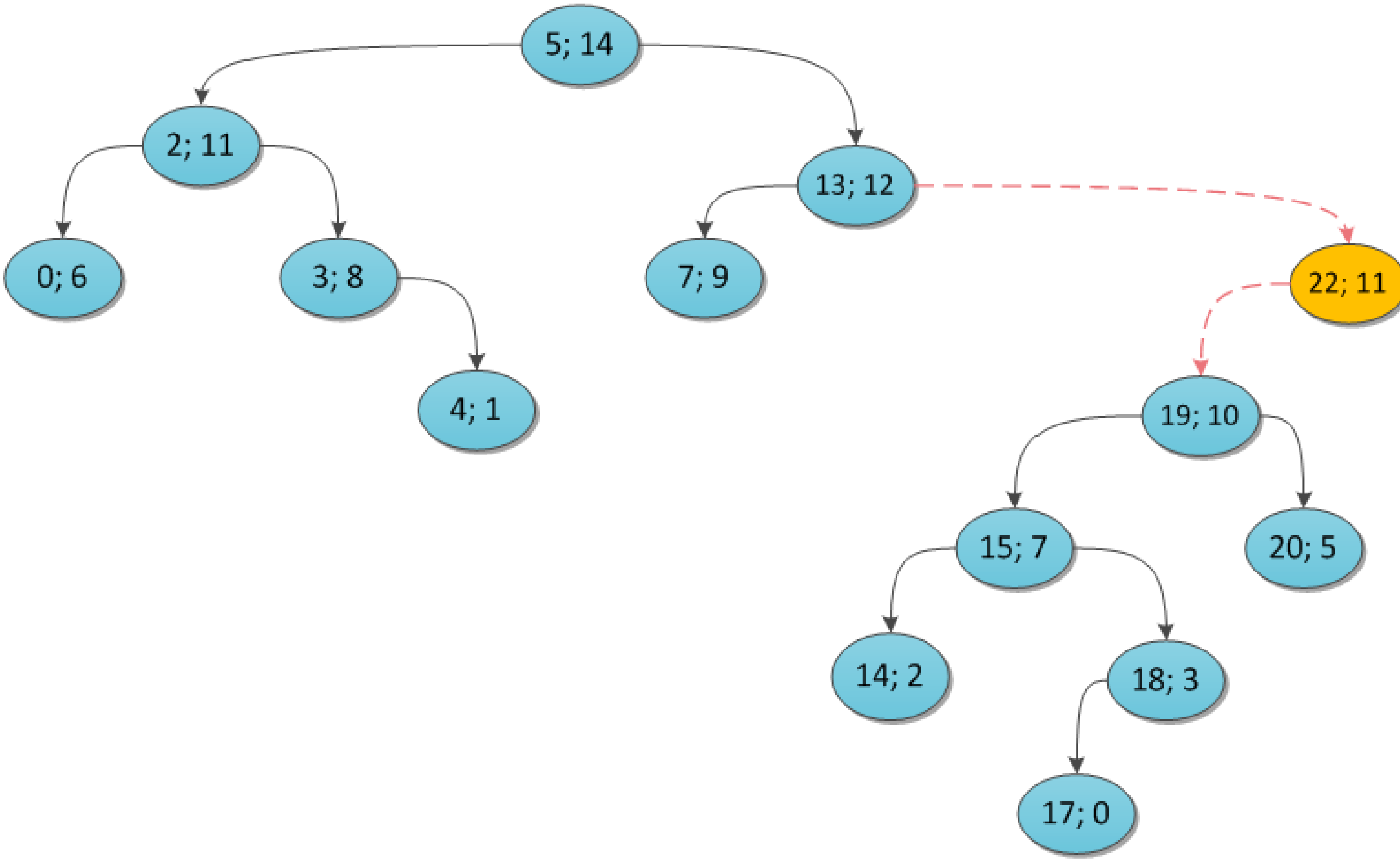
Пример (y = 4)



Пример (y = 16)



Пример (y = 11)



Рандомизация приоритетов

Теорема

В декартовом дереве из n узлов, приоритеты у которого являются случайными величинами с равномерным распределением, средняя глубина вершины $O(\log_2 n)$.

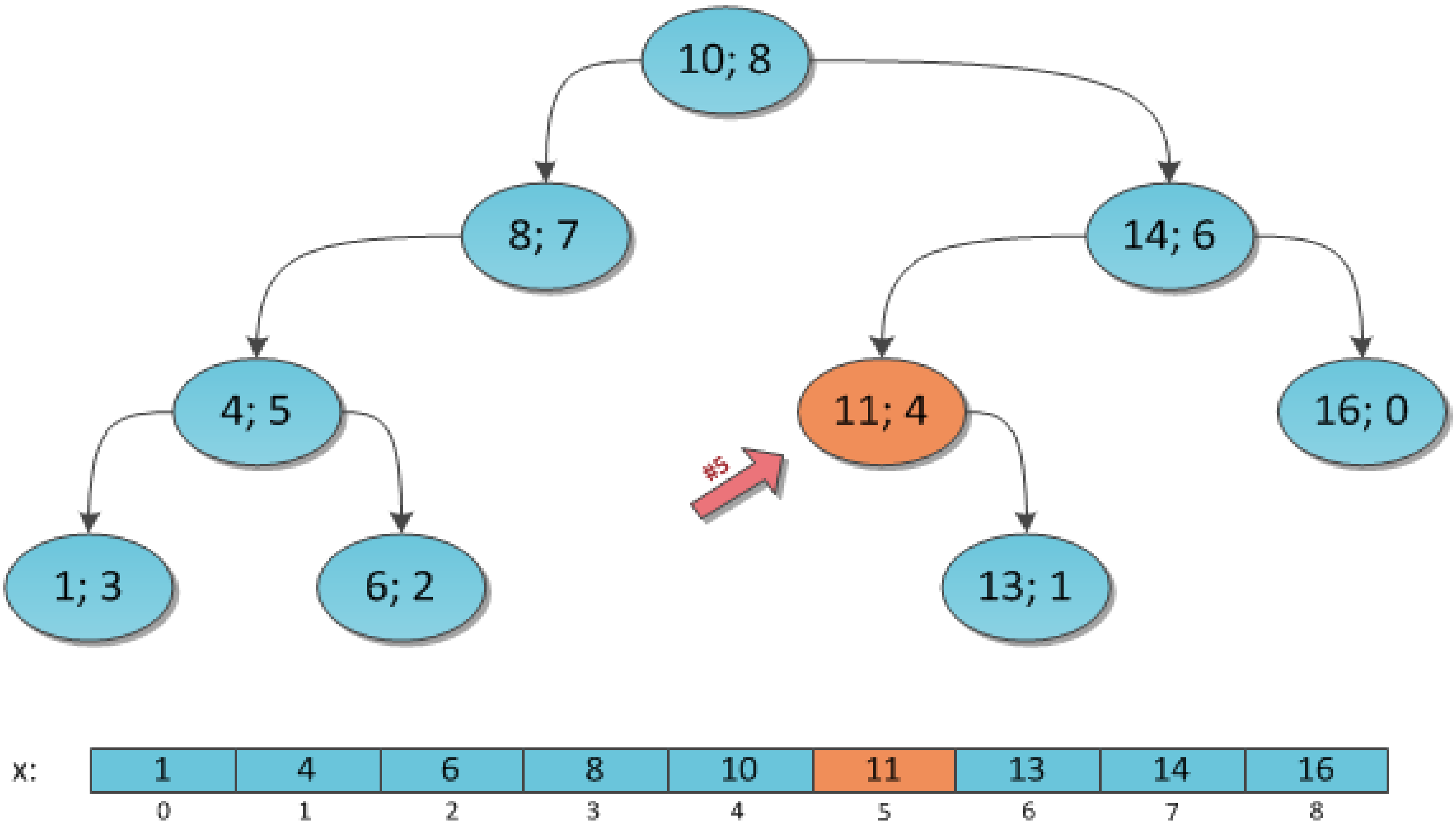
Доказательство:

http://neerc.ifmo.ru/wiki/index.php?title=Декартово_дерево

Свойства декартова дерева:

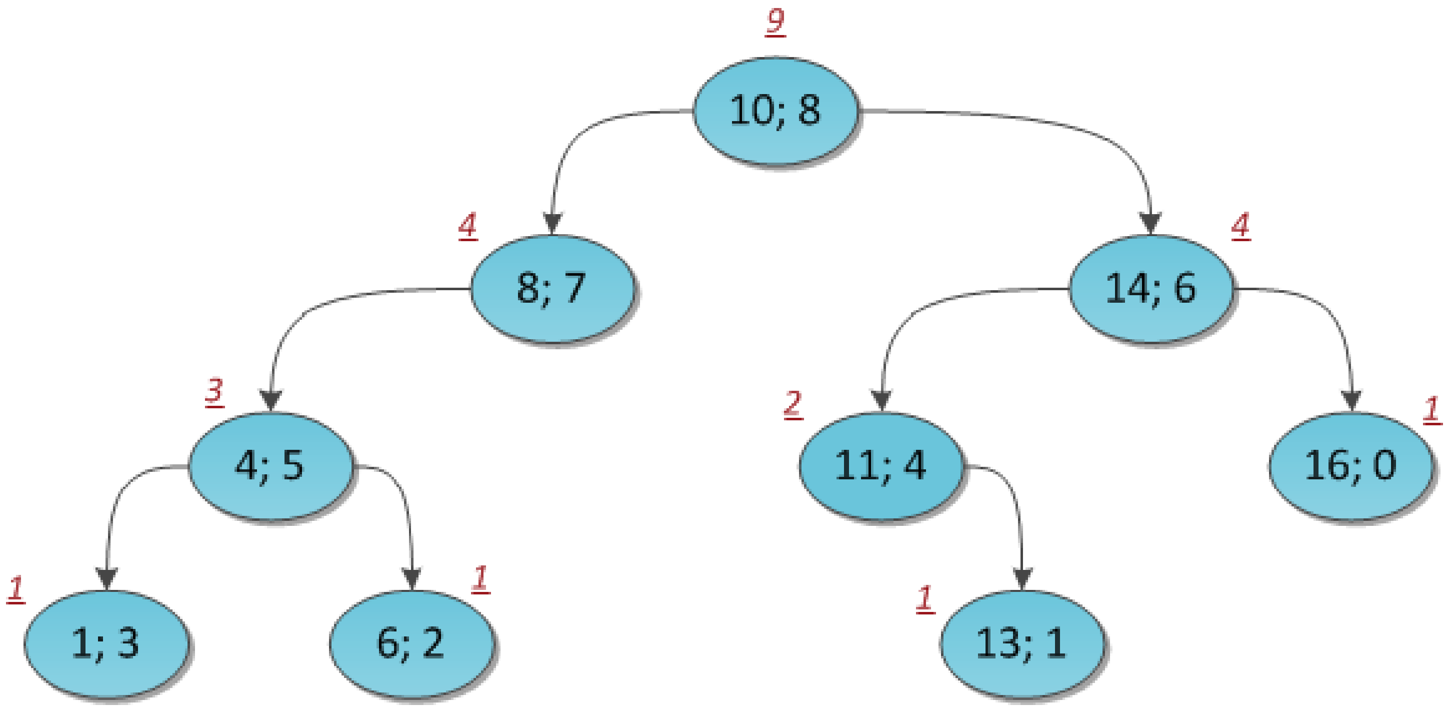
- обладает почти гарантированно логарифмической высотой относительно количества своих вершин;
- позволяет за логарифмическое время искать любой ключ в дереве, добавлять его и удалять;
- исходный код всех её методов не превышает 20 строк, они легко понимаются и в них крайне сложно ошибиться;
- содержит некоторый overhead по памяти, сравнительно с истинно самобалансирующимися деревьями, на хранение приоритетов.

К-я порядковая статистика, или индекс в дереве



К-я порядковая статистика или индекс в дереве

В каждой вершине будем хранить размер поддерева (количество вершин)



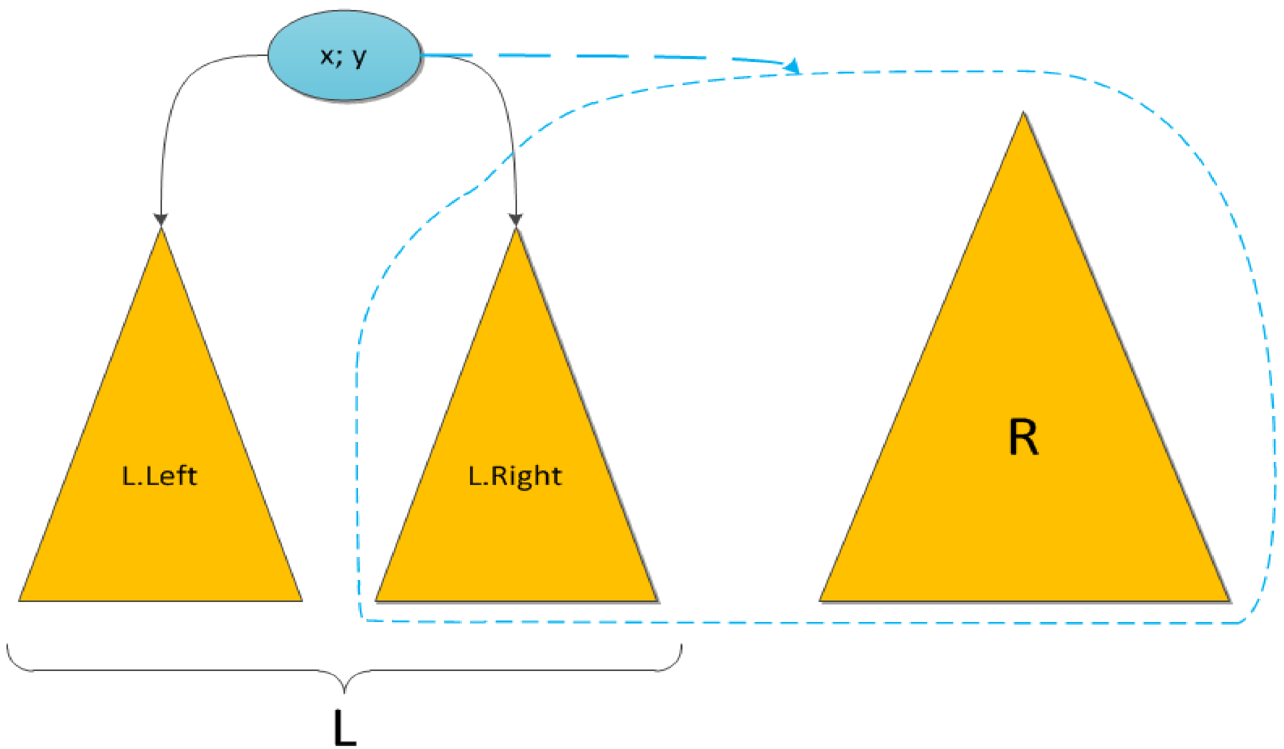
Алгоритм : смотрим в корень дерева и на размер его левого поддерева **S_L**.

Если S_L = K, то искомый элемент мы нашли, и это — корень.

Если S_L > K, то искомый элемент находится где-то в левом поддереве, спускаемся туда и повторяем процесс.

Если S_L < K, то искомый элемент находится где-то в правом поддереве. Уменьшим K на число S_L+1, чтобы корректно реагировать на размеры поддеревьев справа, и повторим процесс для правого поддерева.

Пересчет размеров поддеревьев: Merge



Индукционное предположение: пускай после выполнения Merge на поддеревьях в них все уже подсчитано верно.

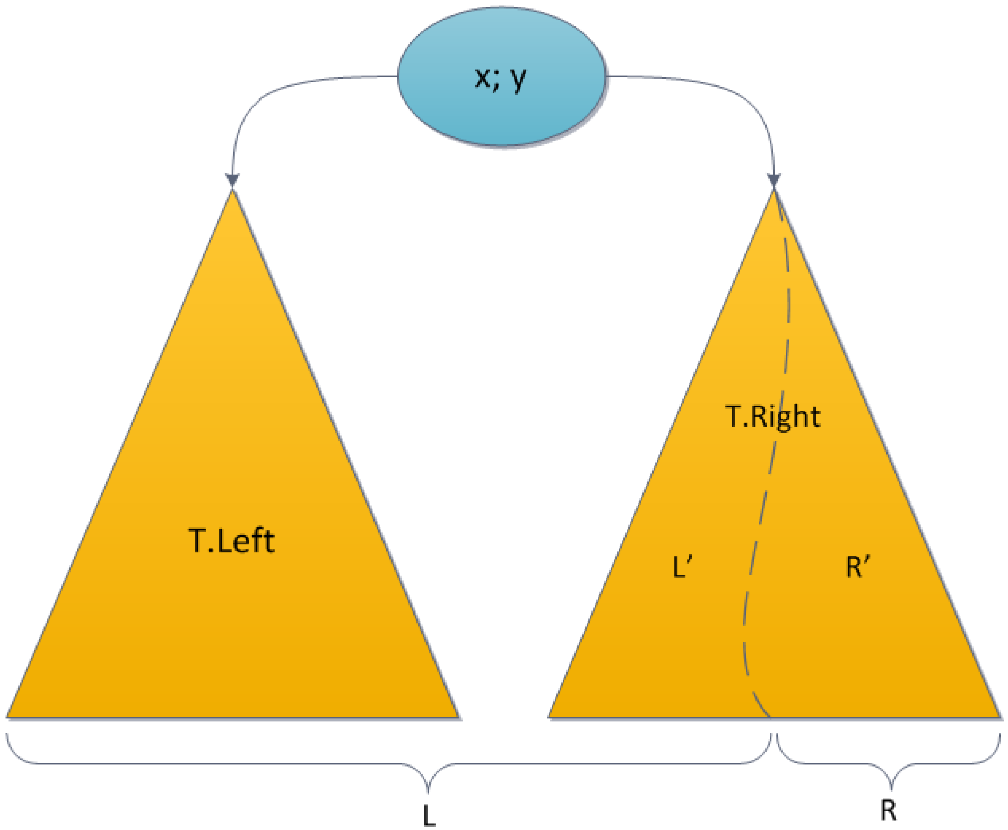
Тогда имеем :

- в левом поддереве размеры подсчитаны верно, т.к. его никто не трогал;
- в правом тоже подсчитаны верно, т.к. это результат работы Merge.

Осталось посчитать только в самом корне нового дерева!

$$size = L.left.size + Merge(L.right, R).size + 1.$$

Пересчет размеров поддеревьев: Split



Индукционное предположение — пускай рекурсивные вызовы Split все подсчитали верно.

Тогда размеры в T.Left корректны — их никто не трогал;

размеры в L' корректны — это левый результат Split;

размеры в R' корректны — это правый результат Split.

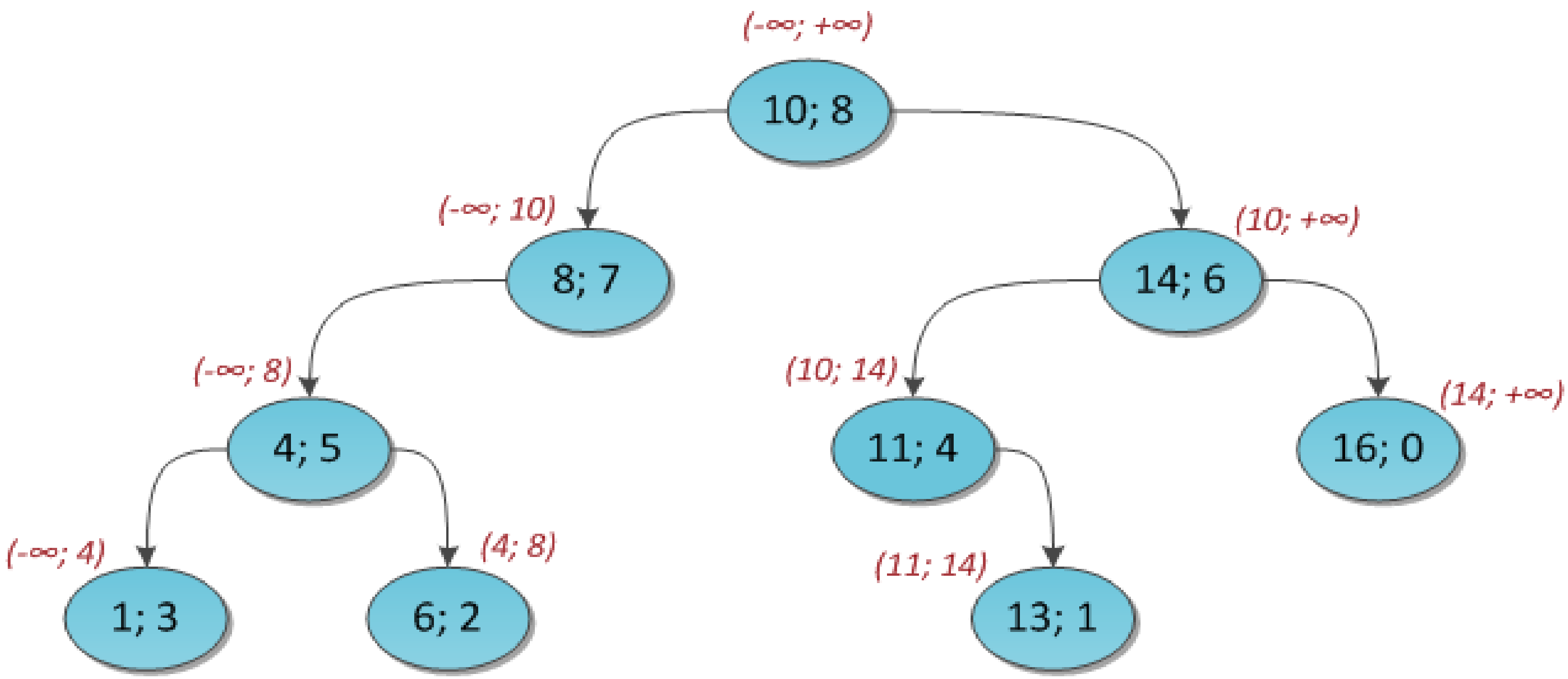
Перед завершением нужно посчитать значение в корне (x; y) будущего дерева L.

L.size = T.Left.size + L'.size + 1.

Нахождение максимума на отрезке

Пусть на вход постоянно поступают (а порою удаляются) ключи x, и с каждым из них связана соответствующая цена — Cost.

Необходимо поддерживать быстрые запросы на *максимум* цены на множестве таких элементов, где **A** ≤ x < **B**.




```
MaxCostOn( T, A, B)
        Split(T, A – 1) → { l,  r};
        Split(r, B) → {m,  r};
        return CostOf(m);
```

Поддержка *множественных операций*

Добавление константы на отрезке

Пусть у нас есть декартово дерево T, в каждой его вершине хранится пользовательская информация Cost.

И мы хотим к каждому значению Cost в дереве (или поддереве)

прибавить какое-то одно и то же число A**.**

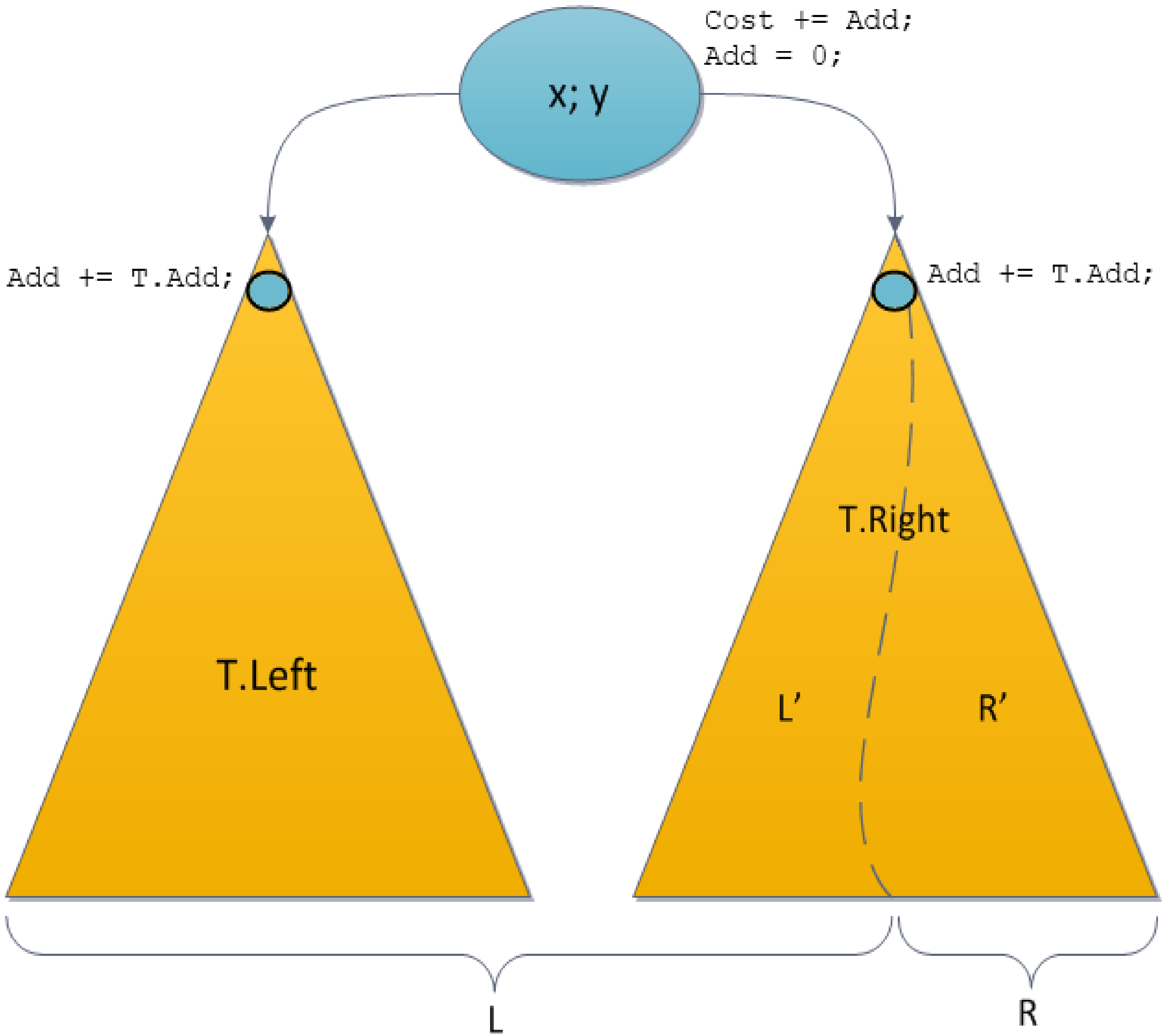
Заведем в каждой вершине дополнительный параметр Add. Он будет сигнализировать о том, что всему поддереву, растущему из данной вершины, *полагается* добавить константу, лежащую в Add.

Cost (T) = T.Cost + T.Add - реальная цена корня

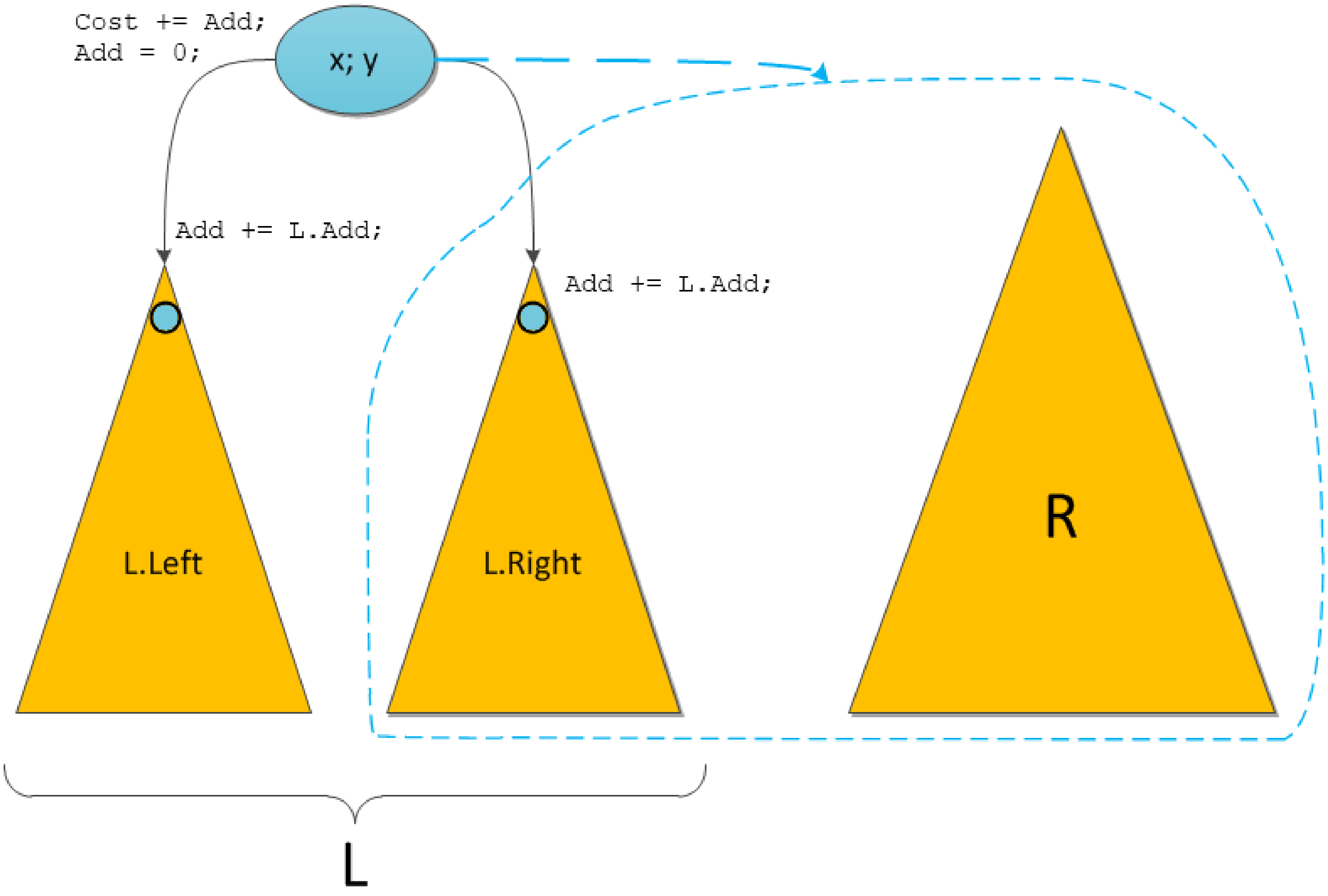
Сумма цен в дереве T:

SumTreeCost (T) = T.SumTreeCost + T.Add * T.Size

Split



Merge



Множественные операции:

- прибавление константы на отрезке;
- на отрезке «красить» — устанавливать всем элементам булев параметр,
- изменять — устанавливать все значения Cost на отрезке в одно значение,
- и т. д.

Главные условия на операцию:

- ее можно за O(1) протолкнуть вниз от корня к потомкам, передав отложенное обещание чуть ниже по дереву;
- информация должна легко восстанавливаться из обещания во время запроса.