Алгоритмы и структуры данных

Деревья поиска II

CS Center, Новосибирск

Рандомизированное BST

- Цель: получить дерево сбалансированное в среднем

Рандомизированное BST

- Цель: получить дерево сбалансированное в среднем
- Назначим вершинам случайно выбранные приоритеты

Рандомизированное BST

- Цель: получить дерево сбалансированное в среднем
- Назначим вершинам случайно выбранные приоритеты

```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

Рандомизированное BST

- Цель: получить дерево сбалансированное в среднем
- Назначим вершинам случайно выбранные приоритеты

```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

Дуча | Treap

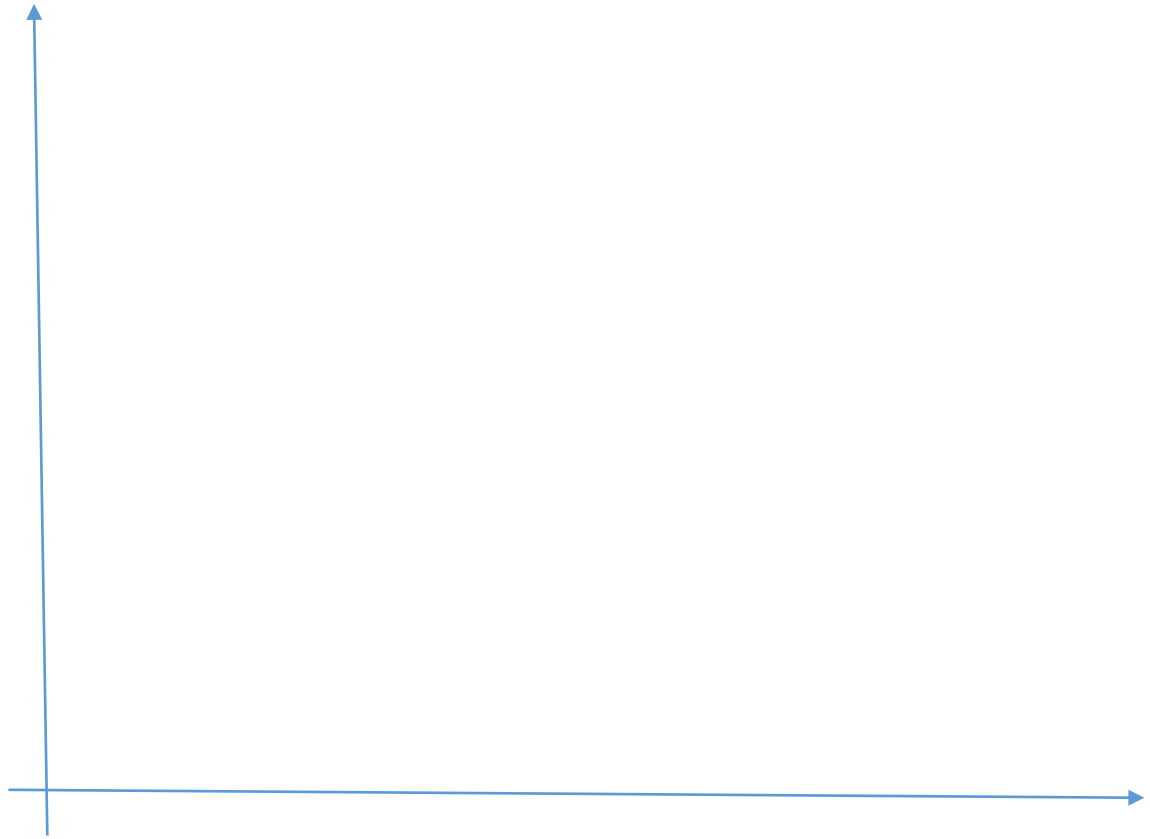
```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

- Декартово дерево
  - BST по x
  - Куча по y

Дуча | Treap

```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

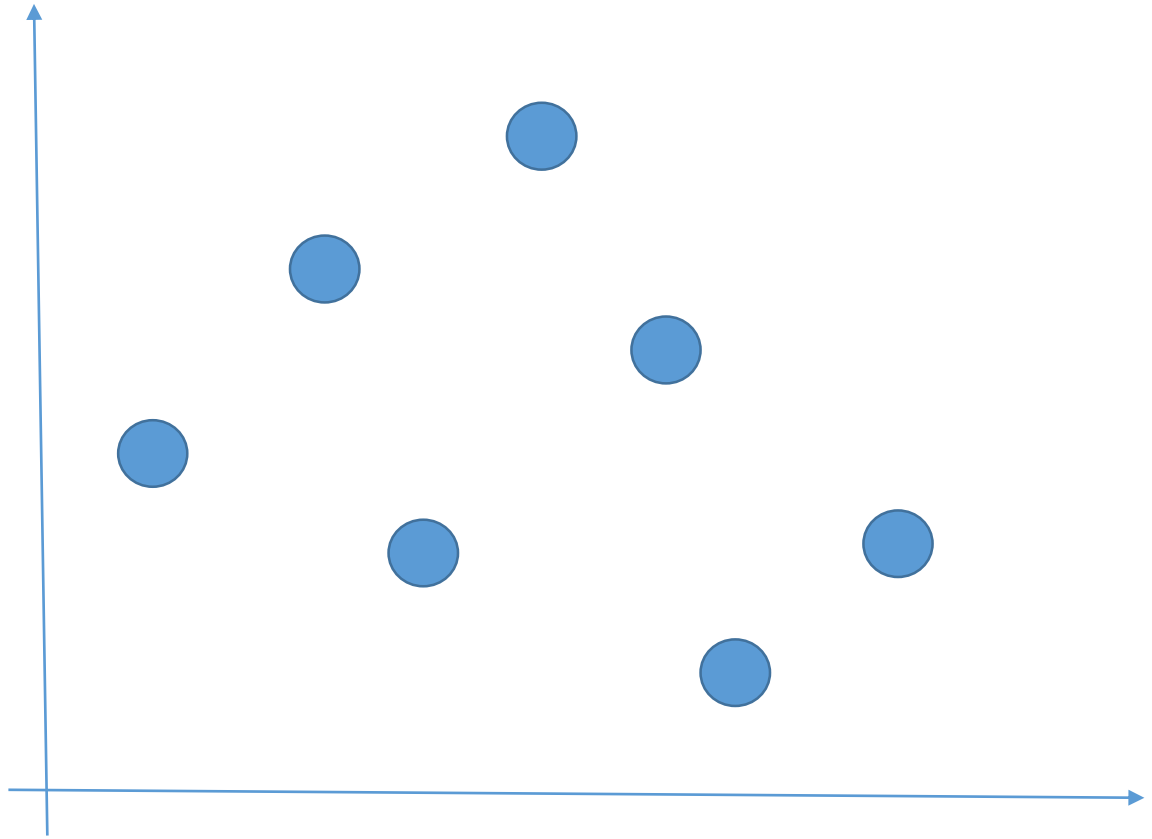
- Декартово дерево
  - BST по x
  - Куча по y



Дуча | Treap

```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

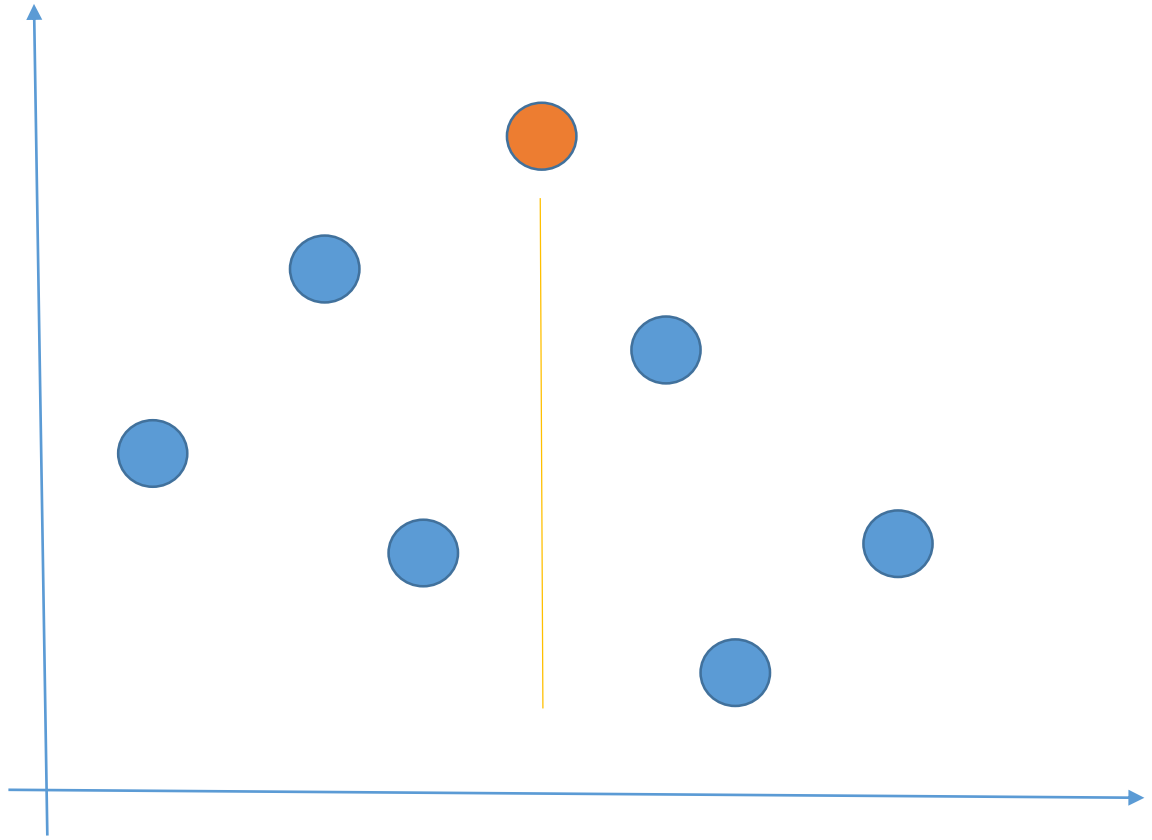
- Декартово дерево
  - BST по x
  - Куча по y



Дуча | Treap

```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

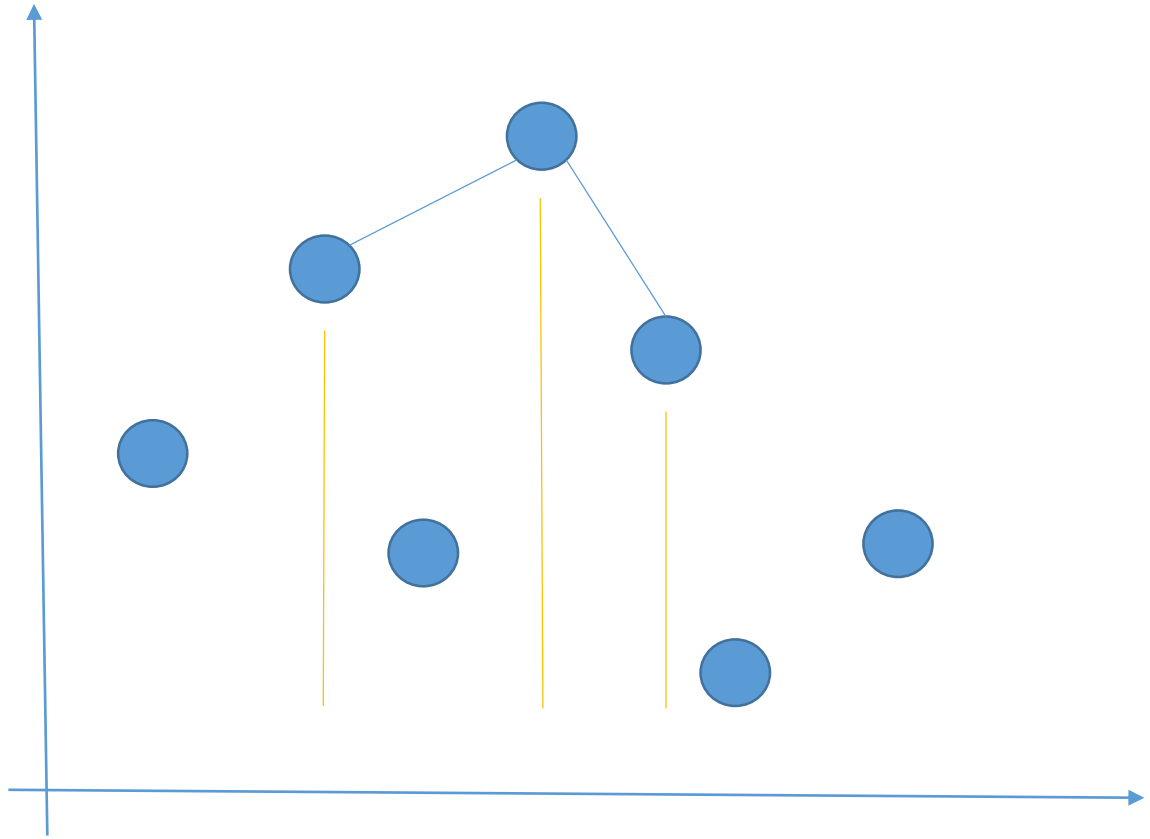
- Декартово дерево
  - BST по x
  - Куча по y



Дуча | Treap

```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

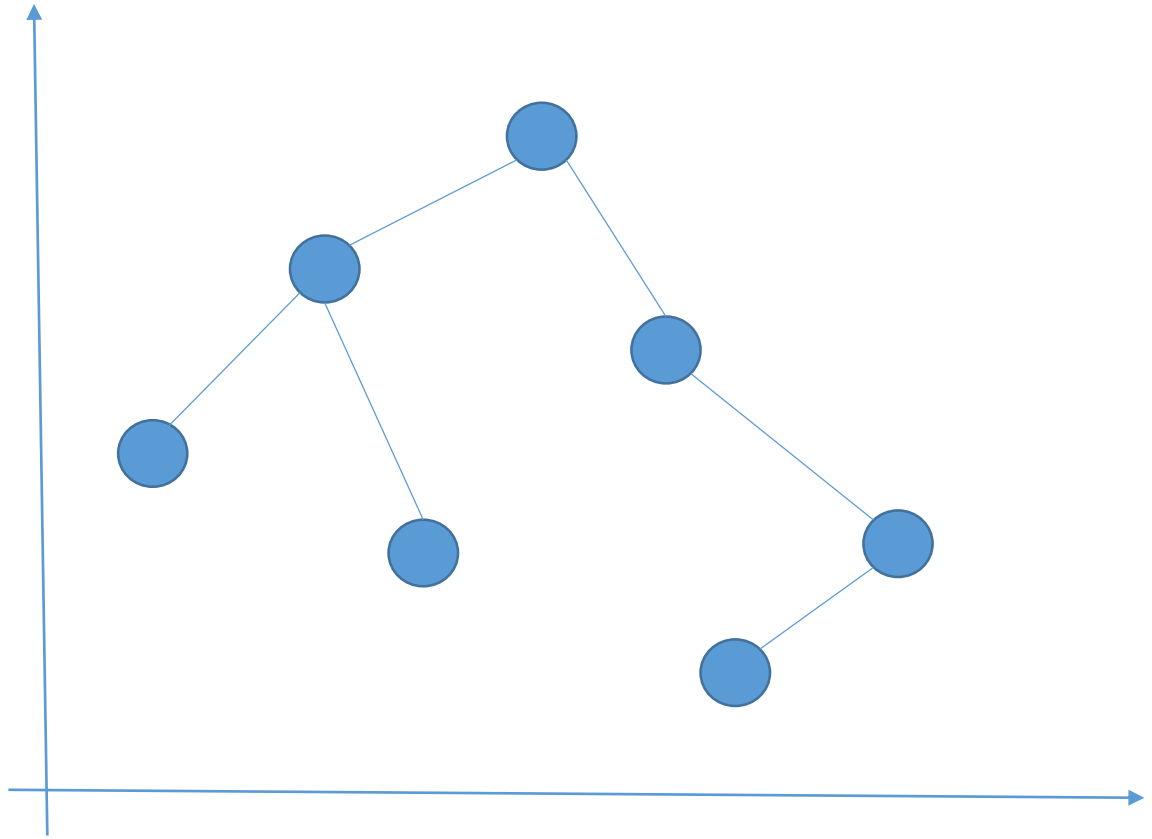
- Декартово дерево
  - BST по x
  - Куча по y



Дуча | Treap

```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

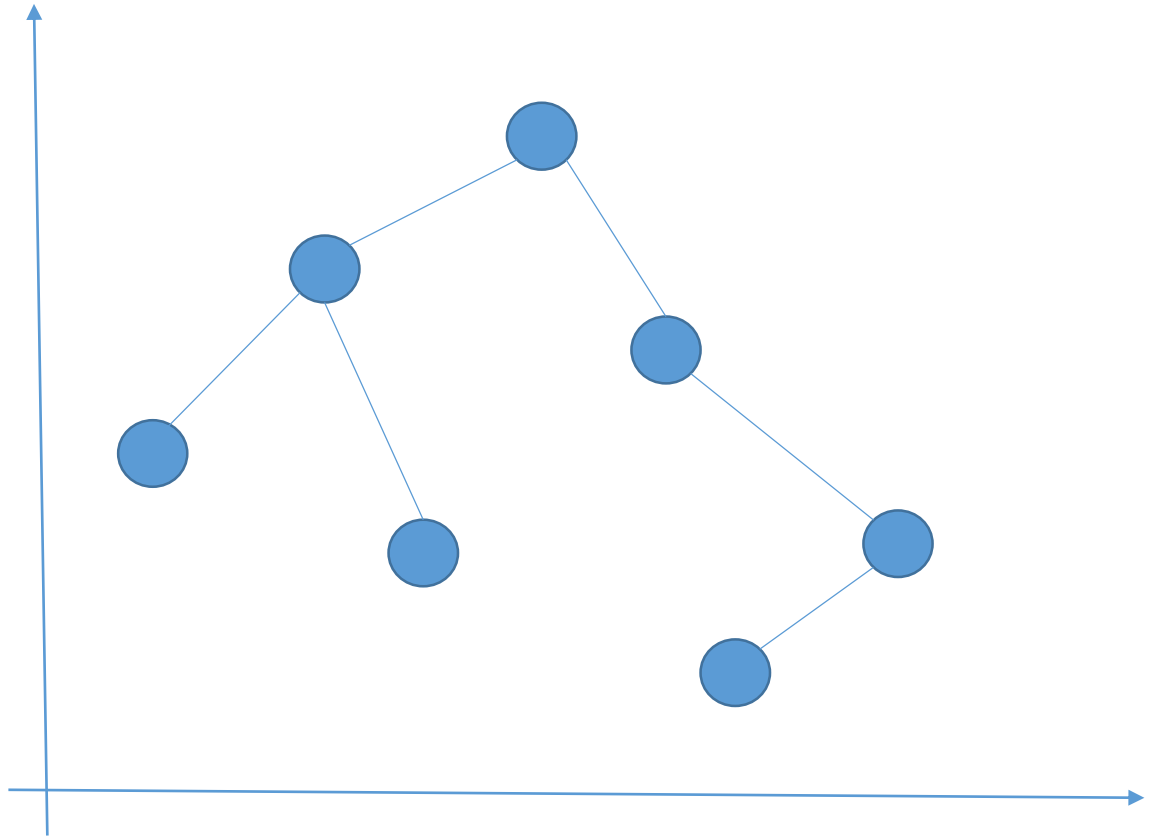
- Декартово дерево
  - BST по x
  - Куча по y



Дуча | Treap

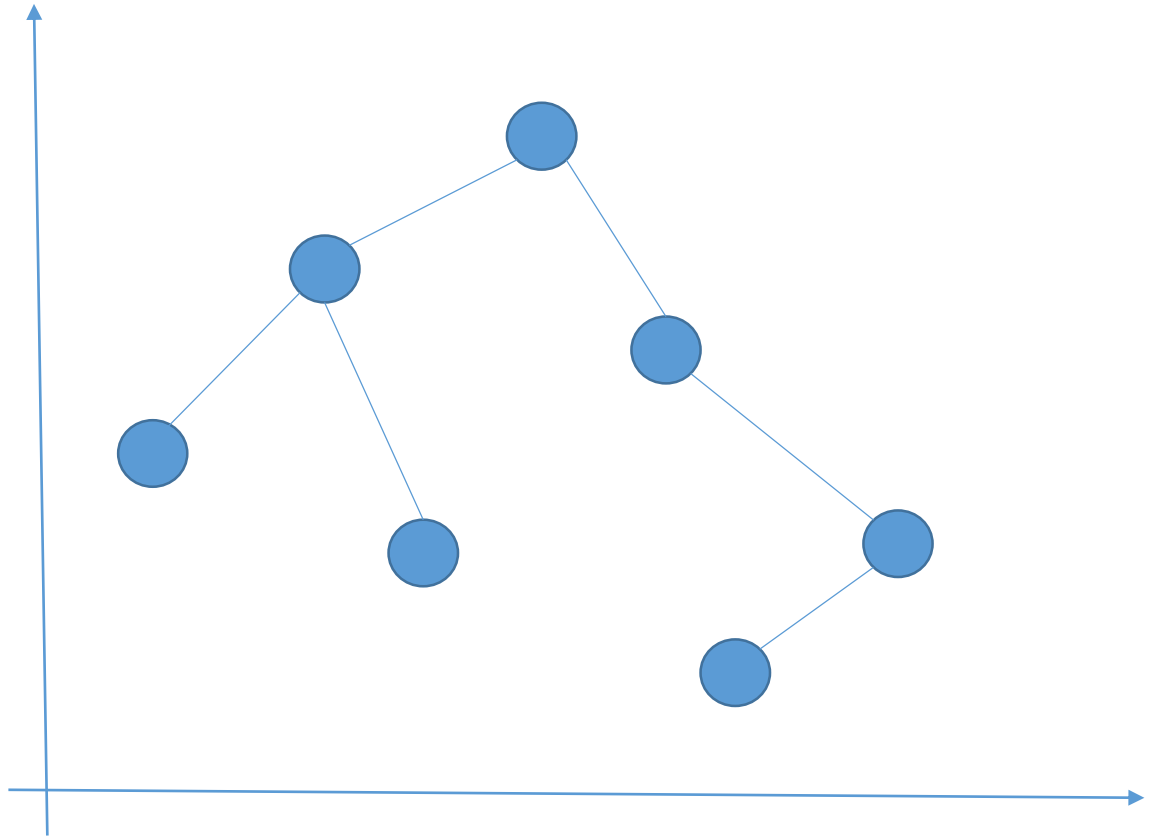
```
Struct node {
    Key x;
    Priority y;
    Value v;
    Node *l;
    Node *r;
}
```

- Декартово дерево
  - BST по x
  - Куча по y
- Всегда существует?



Дуча | Treap

- Декартово дерево
  - BST по x
  - Куча по y
- Если все ключи и приоритеты уникальны для них существует единственное дерево



Дуча | Treap: оценка высоты

```
build_treap( keys ) {
    nodes = make_nodes_with_rand_prior( keys )
    return build( nodes )
}

build( nodes ) {
    if (nodes == {}) return null
    root = extract_highest( nodes )
    (nodes_l, nodes_r) = partition( nodes, root )
    root.l = build( nodes_l )
    root.r = build( nodes_r )
    return root
}
```

Дуча | Treap: оценка высоты

```
build_treap( keys ) {
    nodes = make_nodes_with_rand_prior( keys )
    return build( nodes )
}

build( nodes ) {
    if (nodes == {}) return null
    root = extract_highest( nodes )
    (nodes_l, nodes_r) = partition( nodes, root )
    root.l = build( nodes_l )
    root.r = build( nodes_r )
    return root
}
```

С учётом случайно назначенных приоритетов равномерно выбираем случайный элемент

Дуча | Treap: оценка высоты

```
build_treap( keys ) {
    nodes = make_nodes_with_rand_prior( keys )
    return build( nodes )
}

build( nodes ) {
    if (nodes == {}) return null
    root = extract_highest( nodes )
    (nodes_l, nodes_r) = partition( nodes, root )
    root.l = build( nodes_l )
    root.r = build( nodes_r )
    return root
}
```

С учётом случайно назначенных приоритетов

равномерно выбираем случайный элемент

Высота дерева аналогична глубине рекурсии в QuickSort

Дуча | Treap: методы

- Как реализовать insert и erase?

Дуча | Treap: методы

- Как реализовать insert и erase?
- Определим методы split и merge

Дуча | Treap: методы

- Как реализовать insert и erase?
- Определим методы split и merge
  - split (T, x_0) -> (Tl, Tr)
  - merge(Tl, Tr) -> T

Дуча | Treap: методы

- Как реализовать insert и erase?
- Определим методы split и merge
  - split (T, x_0) -> (Tl, Tr), в Tl все x < «x_0», в Tr все x > «x_0»
  - merge(Tl, Tr) -> T, максимум x в Tl < минимум в Tr

Дуча | Treap: методы

- Как реализовать insert и erase?
- Определим методы split и merge
  - split (T, x_0) -> (Tl, Tr), в Tl все x < «x_0», в Tr все x > «x_0»
  - merge(Tl, Tr) -> T, максимум x в Tl < минимум в Tr

```
insert(T, x) {
   (Tl, Tr) = split(T, x)
   return merge( merge(Tl, node(x)), Tr )
}
```

Дуча | Treap: методы

- Как реализовать insert и erase?
- Определим методы split и merge
  - split (T, x_0) -> (Tl, Tr), в Tl все x < «x_0», в Tr все x > «x_0»
  - merge(Tl, Tr) -> T, максимум x в Tl < минимум в Tr

```
insert(T, x) {
    (Tl, Tr) = split(T, x)
    return merge( merge(Tl, node(x)), Tr )
}
```

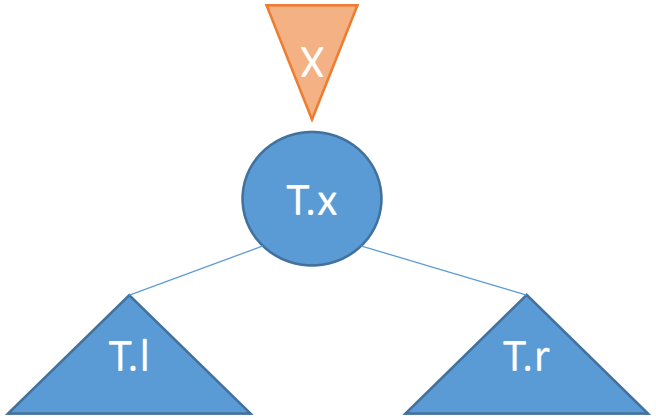
```
erase(T, x) {
    (Tl, Tr) = split(T, x)
    return merge( Tl, Tr )
}
```

Дуча | Treap: split

```
split(T, x) {
    if (T == null) return (null, null)
```

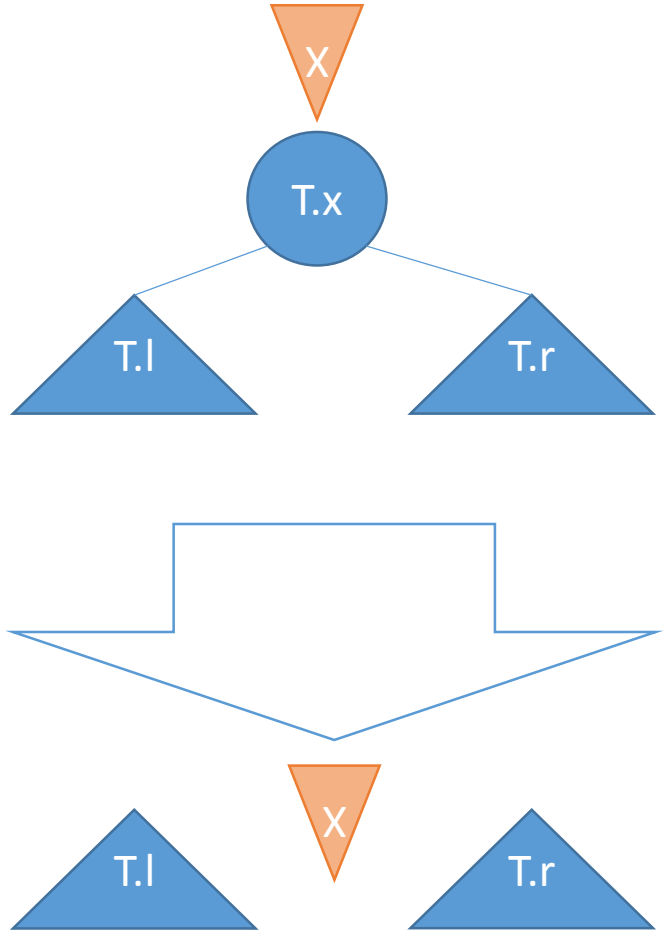
Дуча | Treap: split

```
split(T, x) {
    if (T == null) return (null, null)
    if(T.x == x) return (T.l, T.r)
```



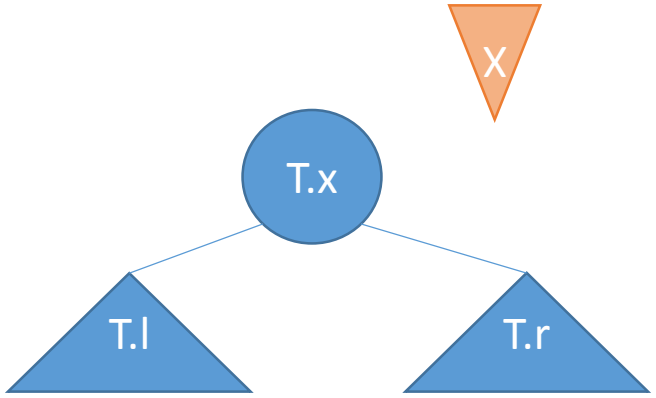
Дуча | Treap: split

```
split(T, x) {
    if (T == null) return (null, null)
    if(T.x == x) return (T.l, T.r)
```



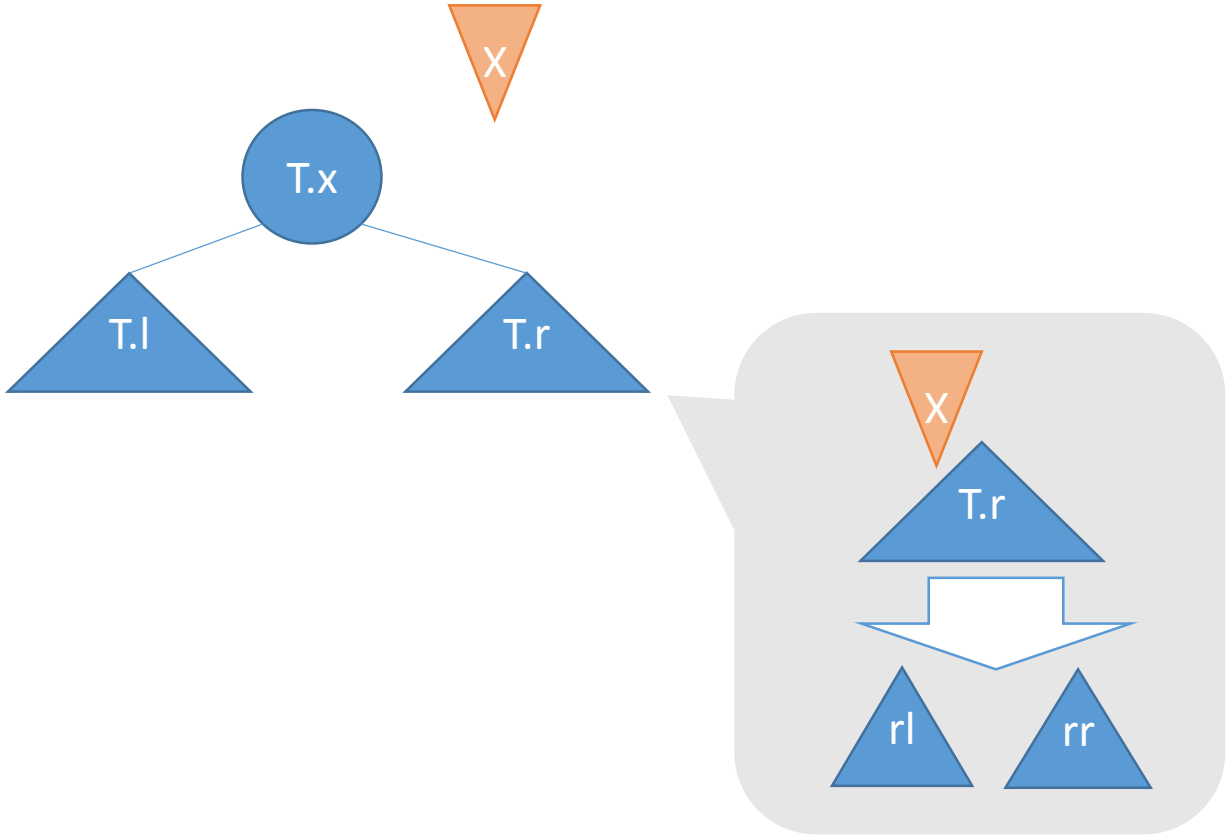
Дуча | Treap: split

```
split(T, x) {
    if (T == null) return (null, null)
    if(T.x == x) return (T.l, T.r)
    if (T.x < x) {
```



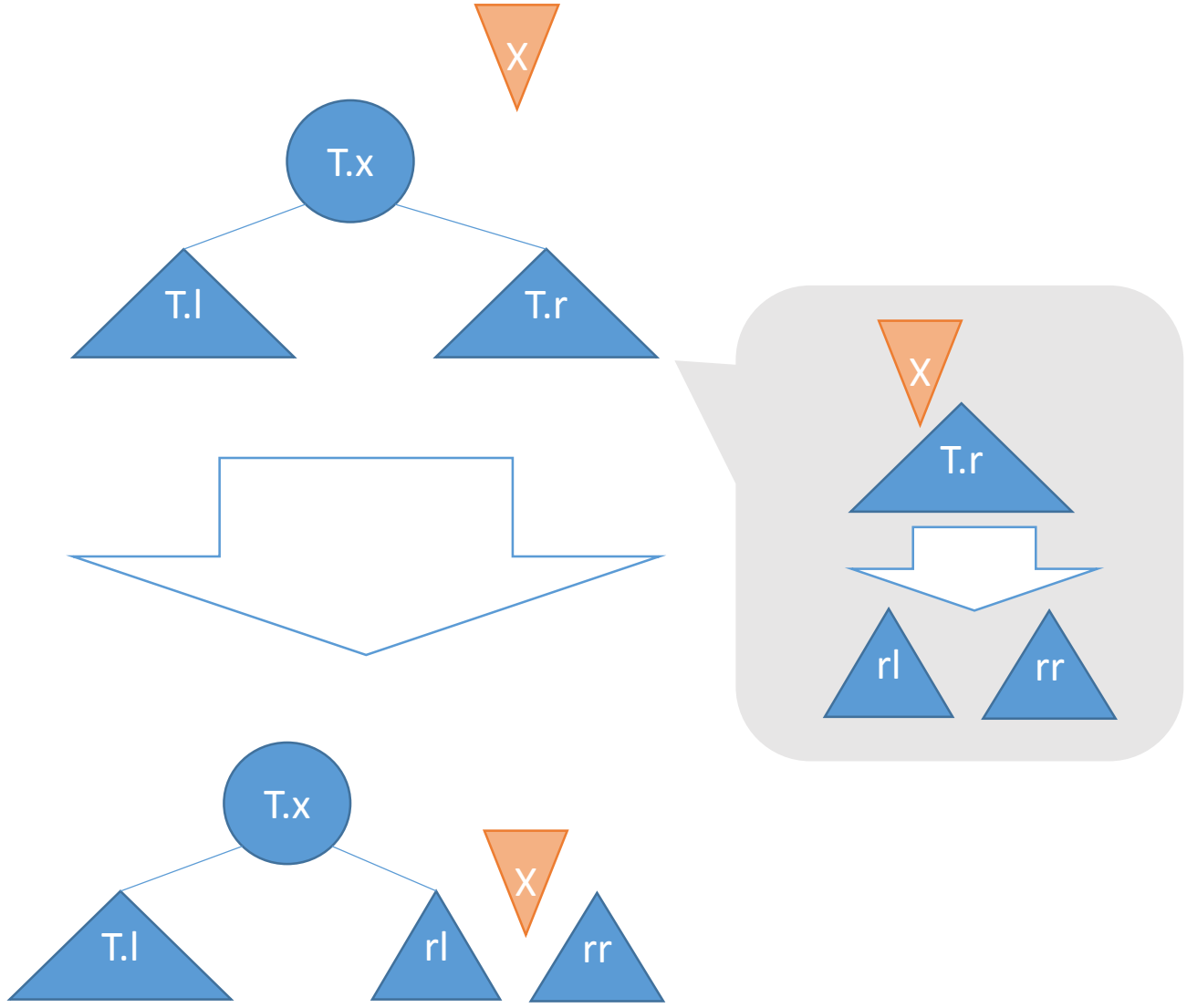
Дуча | Treap: split

```
split(T, x) {
    if (T == null) return (null, null)
    if(T.x == x) return (T.l, T.r)
    if (T.x < x) {
        (rl, rr) = split(T.r, x)
```



Дуча | Treap: split

```
split(T, x) {
    if (T == null) return (null, null)
    if(T.x == x) return (T.l, T.r)
    if (T.x < x) {
        (rl, rr) = split(T.r, x)
        T.r = rl
        return (T, rr)
    }
```



Дуча | Treap: split

```
split(T, x) {
    if (T == null) return (null, null)
    if(T.x == x) return (T.l, T.r)
    if (T.x < x) {
        (rl, rr) = split(T.r, x)
        T.r = rl
        return (T, rr)
    } else {
        (ll, lr) = split(T.l, x)
        T.l = lr
        return (ll, T)
    }
}
```

Дуча | Treap: merge

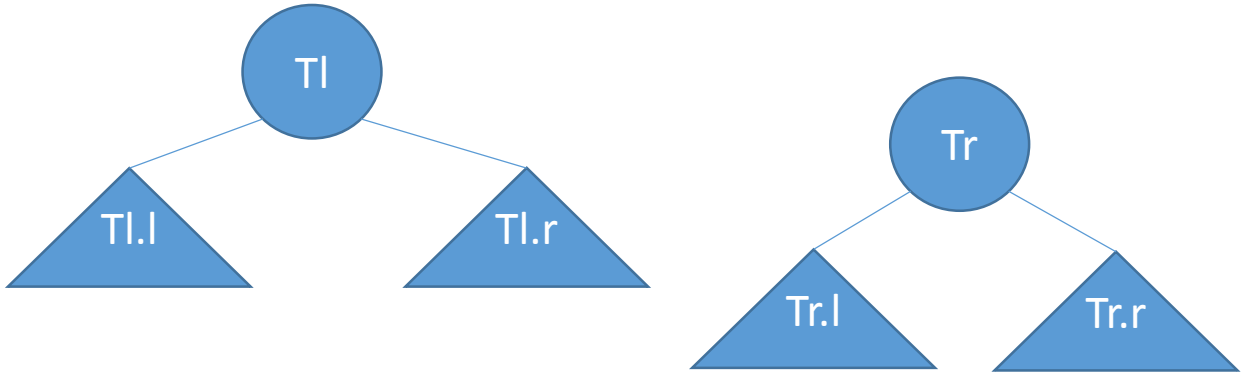
```
merge(Tl, Tr) {
    if (Tl == null) return Tr
```

Дуча | Treap: merge

```
merge(Tl, Tr) {
    if (Tl == null) return Tr
    if (Tr == null) return Tl
```

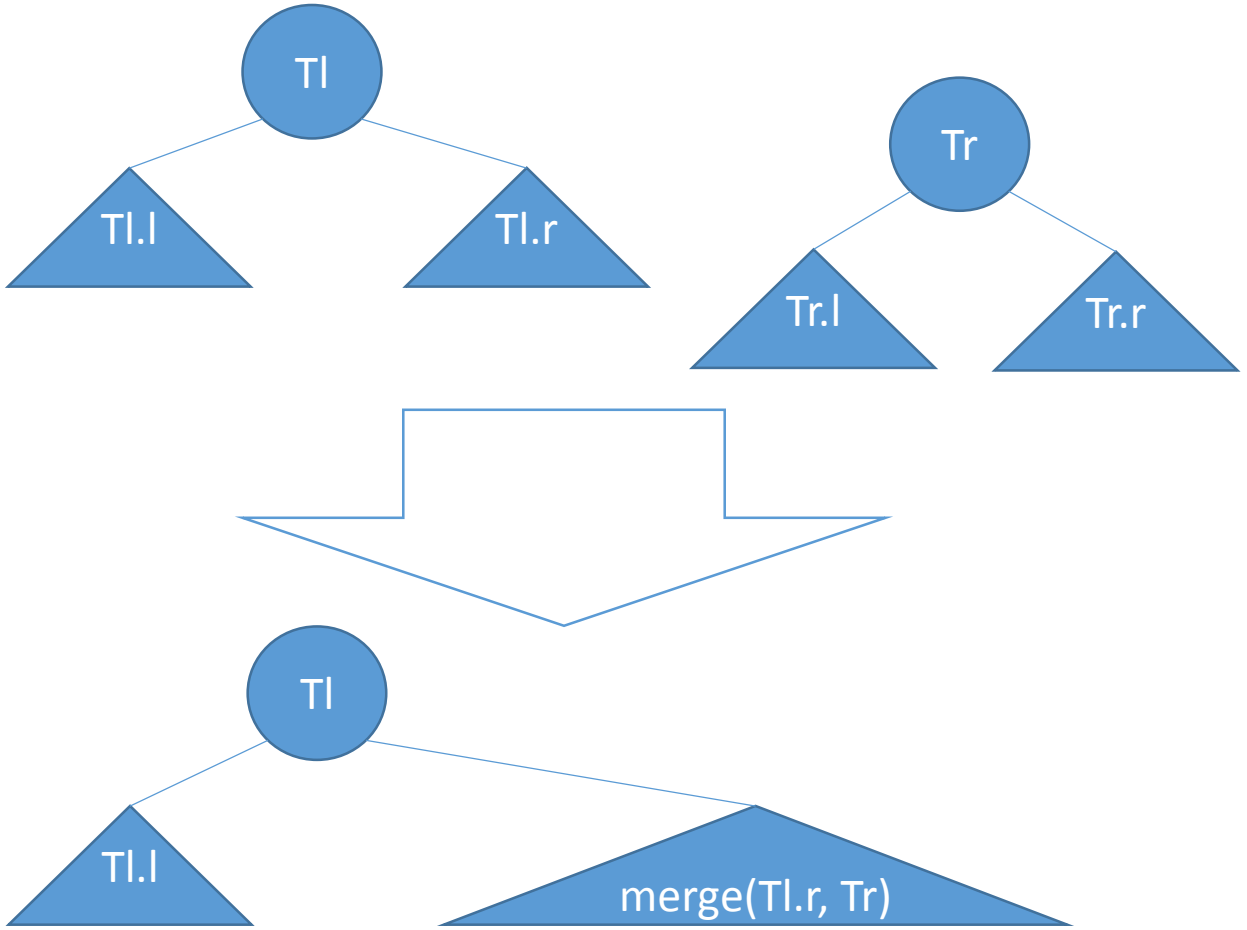
Дуча | Treap: merge

```
merge(Tl, Tr) {
    if (Tl == null) return Tr
    if (Tr == null) return Tl
    if (Tl.y > Tr.y) {
```



Дуча | Treap: merge

```
merge(Tl, Tr) {
    if (Tl == null) return Tr
    if (Tr == null) return Tl
    if (Tl.y > Tr.y) {
        Tl.r = merge(Tl.r, Tr)
        return Tl
    }
```



Дуча | Treap: merge

```
merge(Tl, Tr) {
    if (Tl == null) return Tr
    if (Tr == null) return Tl
    if (Tl.y > Tr.y) {
        Tl.r = merge(Tl.r, Tr)
        return Tl
    } else {
        Tr.l = merge(Tl, Tr.l)
        return Tr
    }
}
```

Дуча | Treap: сложность

- $T_{split} = O(h)$, $T_{merge} = O(h)$
- $T_{insert} = O(h)$, $T_{erase} = O(h)$
- $T_{find} = O(h)$
- В среднем $h = O(\log N)$, независимо от набора ключей

Дуча | Treap: построение для сортированных ключей

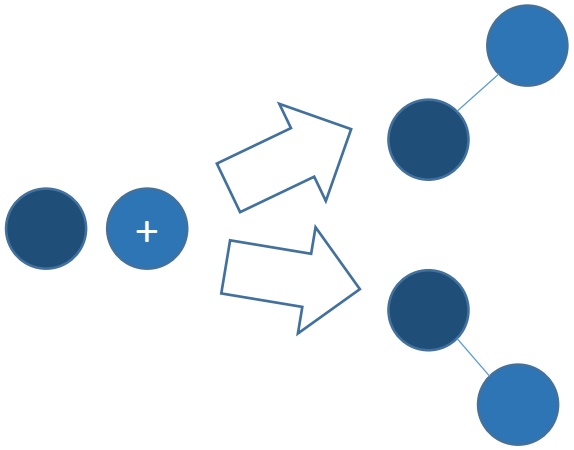
- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T - ?

Дуча | Treap: построение для сортированных ключей

- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T = O(N)
  - Добавляем вершины по одной в порядке увеличения X

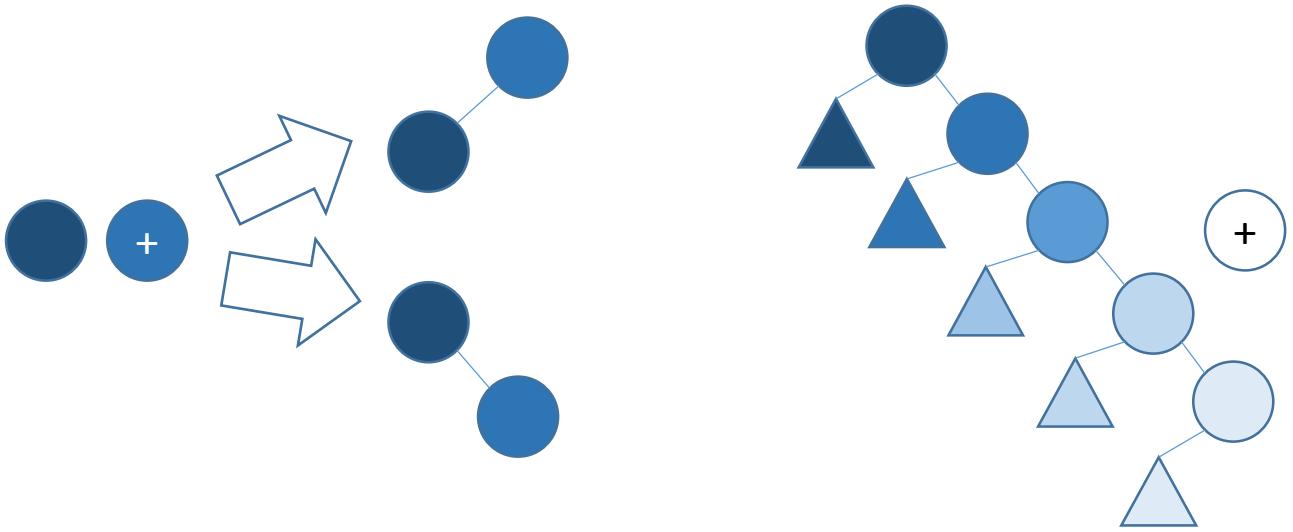
Дуча | Treap: построение для сортированных ключей

- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T = O(N)
  - Добавляем вершины по одной в порядке увеличения X



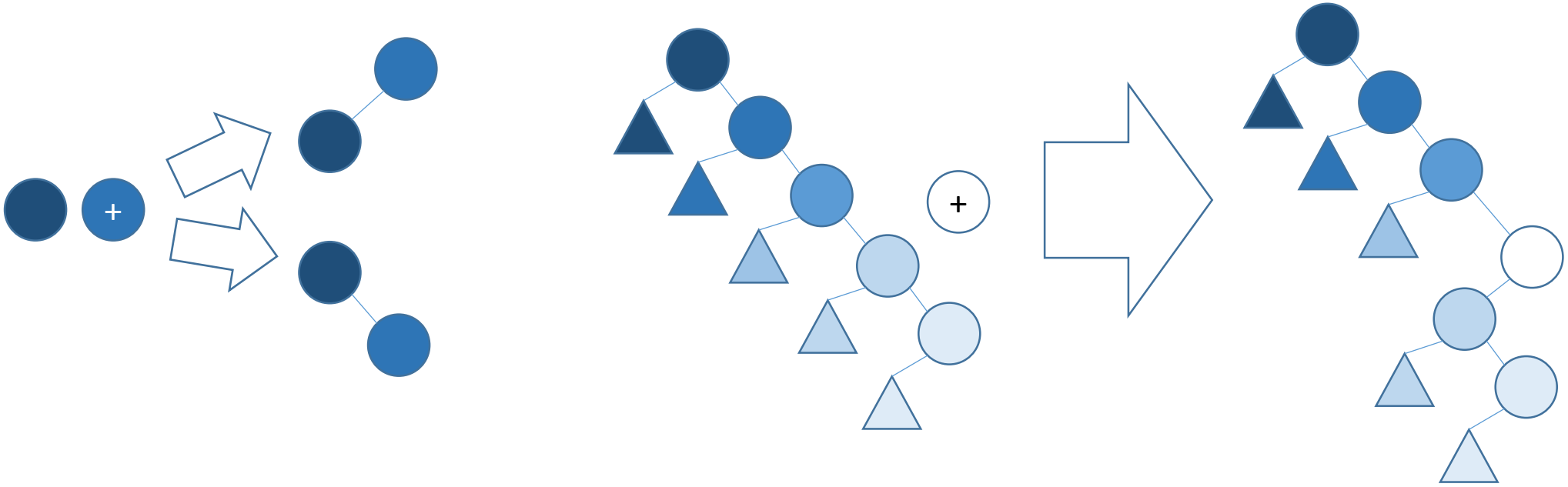
Дуча | Treap: построение для сортированных ключей

- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T = O(N)
  - Добавляем вершины по одной в порядке увеличения X



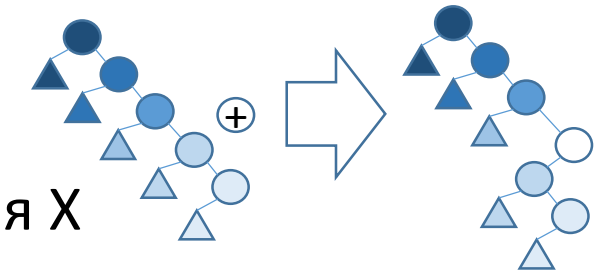
Дуча | Treap: построение для сортированных ключей

- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T = O(N)
  - Добавляем вершины по одной в порядке увеличения X



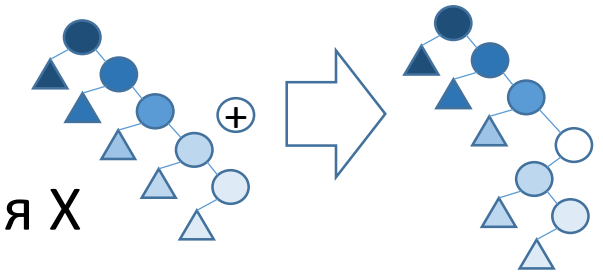
Дуча | Treap: построение для сортированных ключей

- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T = O(N)
  - Добавляем вершины по одной в порядке увеличения X
  - Новая вершина попадает на правый путь



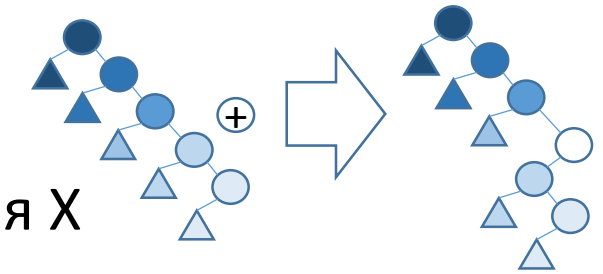
Дуча | Treap: построение для сортированных ключей

- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T = O(N)
  - Добавляем вершины по одной в порядке увеличения X
  - Новая вершина попадает на правый путь
    - Она становится самой глубокой на правом пути



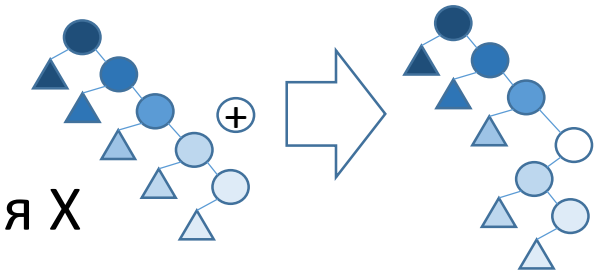
Дуча | Treap: построение для сортированных ключей

- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T = O(N)
  - Добавляем вершины по одной в порядке увеличения X
  - Новая вершина попадает на правый путь
    - Она становится самой глубокой на правом пути
  - Её место находим, поднимаясь вверх из вершины добавленной на прошлом шаге



Дуча | Treap: построение для сортированных ключей

- Даны пары (x_i, y_i), отсортированные по x
- Построить treap, T = O(N)
  - Добавляем вершины по одной в порядке увеличения X
  - Новая вершина попадает на правый путь
    - Она становится самой глубокой на правом пути
  - Её место находим, поднимаясь вверх из вершины добавленной на прошлом шаге
  - Потенциал = длина правого пути



B+ деревья

- Небинарные деревья поиска

B+ деревья

- Небинарные деревья поиска
- Все ключи в листьях

B+ деревья

- Небинарные деревья поиска
- Все ключи в листьях
  - В каждом листе один ключ

B+ деревья

- Небинарные деревья поиска
- Все ключи в листьях
  - В каждом листе один ключ
  - В промежуточных вершинах копии некоторых ключей

B+ деревья

- Небинарные деревья поиска
- Все ключи в листьях
  - В каждом листе один ключ
  - В промежуточных вершинах копии некоторых ключей
  - Число ключей в промежуточной вершине = (число её детей) - 1

B+ деревья

- Небинарные деревья поиска
- Все ключи в листьях
  - В каждом листе один ключ
  - В промежуточных вершинах копии некоторых ключей
  - Число ключей в промежуточной вершине = (число её детей) – 1
- Все листья на одной глубине h

B+ деревья

- Небинарные деревья поиска
- Все ключи в листьях
  - В каждом листе один ключ
  - В промежуточных вершинах копии некоторых ключей
  - Число ключей в промежуточной вершине = (число её детей) – 1
- Все листья на одной глубине h
- Число детей промежуточных вершин от a до b ($a < b/2$)

B+ деревья

- Небинарные деревья поиска
- Все ключи в листьях
  - В каждом листе один ключ
  - В промежуточных вершинах копии некоторых ключей
  - Число ключей в промежуточной вершине = (число её детей) – 1
- Все листья на одной глубине h
- Число детей промежуточных вершин от a до b (a < b/2)
  - Кроме корня, у него от 2 до b детей

B+ деревья

- Небинарные деревья поиска
- Все ключи в листьях
  - В каждом листе один ключ
  - В промежуточных вершинах копии некоторых ключей
  - Число ключей в промежуточной вершине = (число её детей) – 1
- Все листья на одной глубине h
- Число детей промежуточных вершин от a до b ($a < b/2$)
  - Кроме корня, у него от 2 до b детей
- $h = \log_a N$

B+ деревья: insert

- Найти к какой вершине добавить новый лист

B+ деревья: insert

- Найти к какой вершине добавить новый лист
  - После вставки число детей ≤ b => Ok
  - После вставки число детей = b + 1 => ?

B+ деревья: insert

- Найти к какой вершине добавить новый лист
  - После вставки число детей ≤ b => Ok
  - После вставки число детей = b + 1 => сделать split

B+ деревья: insert

- Найти к какой вершине добавить новый лист
  - После вставки число детей ≤ b => Ok
  - После вставки число детей = b + 1 => сделать split
- Split:
  - Разделим вершину на две, по половине детей в каждую

B+ деревья: insert

- Найти к какой вершине добавить новый лист
  - После вставки число детей ≤ b => Ok
  - После вставки число детей = b + 1 => сделать split
- Split:
  - Разделим вершину на две, по половине детей в каждую
  - Возможно, конфликт возник на уровне выше

B+ деревья: insert

- Найти к какой вершине добавить новый лист
  - После вставки число детей ≤ b => Ok
  - После вставки число детей = b + 1 => сделать split
- Split:
  - Разделим вершину на две, по половине детей в каждую
  - Возможно, конфликт возник на уровне выше
    - Продолжаем решать проблему итеративно

B+ деревья: insert

- Найти к какой вершине добавить новый лист
  - После вставки число детей ≤ b => Ok
  - После вставки число детей = b + 1 => сделать split
- Split:
  - Разделим вершину на две, по половине детей в каждую
  - Возможно, конфликт возник на уровне выше
    - Продолжаем решать проблему итеративно
    - Если дошли до корня, увеличим высоту дерева, добавив новый корень

B+ деревья: erase

- Найти у какой вершины удаляем лист

B+ деревья: erase

- Найти у какой вершины удаляем лист
  - После удаления число детей ≥ a => Ok
  - После удаления число детей = a - 1 => ?

B+ деревья: erase

- Найти у какой вершины удаляем лист
  - После удаления число детей ≥ a => Ok
  - После удаления число детей = a - 1 => сделать share или fuse

B+ деревья: erase

- Найти у какой вершины удаляем лист
  - После удаления число детей ≥ a => Ok
  - После удаления число детей = a - 1 => сделать share или fuse
- Share:
  - Условие: есть брат числом детей c > a

B+ деревья: erase

- Найти у какой вершины удаляем лист
  - После удаления число детей ≥ a => Ok
  - После удаления число детей = a - 1 => сделать share или fuse
- Share:
  - Условие: есть брат числом детей c > a
  - Раздадим вершине и её брату по половине от (c + a – 1) детей

B+ деревья: erase

- Найти у какой вершины удаляем лист
  - После удаления число детей ≥ a => Ok
  - После удаления число детей = a - 1 => сделать share или fuse
- Share:
  - Условие: есть брат числом детей c > a
  - Раздадим вершине и её брату по половине от (c + a – 1) детей
- Fuse:
  - Объединим вершину с братом

B+ деревья: erase

- Найти у какой вершины удаляем лист
  - После удаления число детей ≥ a => Ok
  - После удаления число детей = a - 1 => сделать share или fuse
- Share:
  - Условие: есть брат числом детей c > a
  - Раздадим вершине и её брату по половине от (c + a – 1) детей
- Fuse:
  - Объединим вершину с братом
  - Итеративно решаем возможную проблему на уровне выше