1 Терминология

• *Offline* структура данных — все K запросов даны заранее. Можно обработать их одновременно.

• *Online* структура данных — следующий запрос можно узнать только после того, как структура данных ответит на предыдущий.

• *Real время* работы — максимальное время обработки одного запроса (например, *Real время* работы операции `get` в `Системе непересекающихся множеств` $= O(\log N)$.

• *Аморатизационное время* работы — максимальное время обработки **первых** K запросов деленное на K (например, *аморатизационное время* работы операции `get` в `Системе непересекающихся множеств` $= O($`обратной функции Аккермана`$)$.

2 Persistent Data Structures

Основной принцип: никакой уже созданный объект (например, вершина дерева) никогда в будущем не поменяется.

То, что мы хотим достигнуть: теперь если у нас есть, например, *Persistent Array*, то операция `a[i] := x` создаст новый массив. Т.е. в результате у нас будет два массива, отличающихся только в i-м элементе.

• *Persistent Search Tree* — Вопрос: как обычно добавляются вершины в дерево поиска (здесь и далее я говорю о самом простом, несбалансированном дереве поиска)? Ответ: спускаемся вниз по дереву и в конце создаем новый лист. Чтобы лист был частью дерева, нужно сказать, что у вершины, за которую крепится лист поменялась ссылка на левого или на правого сына. В нашем случае так делать нельзя. Нельзя менять уже существующие вершины, в частности — ссылки на левого и правого сына. Сделаем так: создадим копии всех вершин, на пути от корня до листа. Свежесозданные вершины, естественно, будут иметь левых и правых сыновей из старых вершин. Корень свежесозданного дерева — копия корня старого дерева.
Теперь мы можем написать код добавления в *Persistent Search Tree*:

```
Add(t, x) {
.     if t is empty return new tree(x, null, null)
.     else if t.x < x return new tree(t.x, t.l, Add(t.r, x))
.     else return new tree(t.x, Add(t.l, x), t.r)
}
```

• *Persistent Treap* (treap = декартово дерево) — декартово дерево основано на двух основных операциях `Split` и `Merge`. Чтобы декартово дерево стало персистентным достаточно по ходу рекурсии создавать новые вершины и не менять старые. **Внимание:** чтобы не было проблем с `y`-ми, используйте `RBST` вместо `treap` (см. тему `RBST`).

• *Persistent Дерево Отрезков* — если все операции с деревом отрезков реализованы спуском сверху вниз, то опять же нужно просто создавать новые вершины и не менять старые. Тут стоит заметить, что классическая реализация дерева отрезков считает, что вершина i имеет детей $2i$ и $2i+1$. Теперь так делать нельзя. У каждой вершины придется хранить две ссылки — на левого сына и на правого.

• *Persistent Array* — можно хранить массив в дереве отрезков (или в декартовом дереве по неявному ключу :-). Чтобы массив стал персистентным, достаточно дерево отрезков, в котором мы его храним, сделать персистентным.

• *ScanLine* (сканирующая прямая) — рассмотрим решение классической задачи *даны N точек на плоскости и M прямоугольников, нужно для каждого прямоугольника посчитать количество точек в внутри.* Будем отвечать на запросы в *Offline*. Ответ для прямоугольника $[x_1..x_2] \times [y_1..y_2] = F(x_2, [y_1..y_2]) - F(x_1 - 1, [y_1..y_2])$ где F — количество точек на полоске $[y_1..y_2]$, у которых x не больше чем данный. Чтобы посчитать F, отсортируем точки по x и будем в таком порядке для каждой точки (x, y) делать `sum[y]++`. В тот момент, когда мы добавили все точки с x не больше некоторого X, значение функции $F(X, [y_1..y_2])$ — это сумма на отрезке $y_1..y_2$. Чтобы посчитать эту сумму мы храним массив `sum`, как дерево отрезков с операцией "сумма на отрезке". Это было решение классической задачи. Теперь перейдем к вопросу "причем здесь персистентность?". А при том, что теперь после перебора всех точек нам доступны все промежуточные деревья отрезков, и на запрос-прямоугольник мы можем отвечать в `Online`.

• **Время и память** — Сбалансированные деревья поиска и дерево отрезков все еще работают за $O(\log N)$ на запрос. При этом K запросов выделяют $K \log N$ памяти. Операции с массивом (`get(i)` и `set(i, x)` теперь работают за $O(\log N)$.

• **Все структуры могут стать персистентными** — мир состоит из массивов, а персистентный массив — это персистентное дерево отрезков. Т.е. **любую** структуру данных вы уже можете сделать персистентной.

• *Garbage Collection* (сборка мусора) — Сейчас на примере персистентного дерева мы научимся оптимизировать память. После выполнения K запросов у нас появилась $K+1$ копия дерева, которые вместе занимают целых $O(N + K \log N)$ ячеек памяти, что печально. Печально это в том случае, если из всех $K+1$ созданных деревьев нужны нам только 3 (т.е. среди $O(N + K \log N)$ вершин, только $O(N)$ являеются нужными). Итак, **сборка мусора**: для каждой вершины будем хранить `count` — количество ссылок вида "родитель $\to$ ребенок"на эту вершину. Также для каждой версии увеличим `count` корня дерева на 1. Теперь, если какое то дерево с корнем `root` нам больше не нужно, то достаточно вызвать следующую функцию:

```
Del(t) {
.     t.count--;
.     if (t.count == 0) {
.         Del(t.l); Del(t.r); delete t;
.     }
}
```

• **Замечание** — все что сказано выше относится к *Online* задачам и структурам. В *Offline* задачи на персистентные структуры решаются гораздо проще. Для этого достаточно представить себе все дерево версий. Например, чтобы обработать все запросы к *Persistent Array* достаточно обойти дерево версий `dfs`-ом.

3 SQRT-decomposition

`SQRT-decomposition` = `корневая эвристика`. Этот подход многолик. Сейчас вы увидите несколько его проявлений в теории алгоритмов.

• **Простая** — Рассмотрим задачу "нужно с массивом делать 2 операции `a[i] := x` и `getSum(L..R)`". Пусть мы не знаем структуру данных дерево отрезков. Давайте хранить массив `a` и суммы `a[0..K-1]`, `a[K..2K-1]` и т.д. Чтобы сделать операцию `a[i] := x` достаточно двух действий — поменять массив и поменять одну из сумм. Чтобы посчитать `getSum(L..R)`, нам достаточно $O(K + \frac{N}{K})$ времени (отрезок $[L..R]$ делится на два хвоста и несколько уже посчитанных сумм `a[i·K..i·K+K-1]`. Если выбрать K равным $\sqrt{N}$, то время обработки запроса `getSum(L..R)` будет $\sqrt{N}$. Конец :-)

• **Почти дерево отрезков** — Если вы знаете дерево отерков, структуру описанную выше, можно рассматривать как двухуровневое дерево, в котором у каждой вершины $\sqrt{N}$ детей. Давайте добавим третий уровень, тогда у каждой вершины будет $\sqrt[3]{N}$ детей, и у нас получится структура данных, которая умеет запрос `getSum(L..R)` обрабатывать за $O(\sqrt[3]{N})$, а запрос `a[i] := x` за $O(1)$. Это *Real Time*.

• **Split & Rebuild** — Научимся решать новую, более сложную задачу. Запросы: `reverse(L..R)`, `getSum(L..R)`. Сперва для исходного массива предподсчитаем суммы на префиксах (массив `sum`). Теперь, если мы не пользуемся операцией `reverse(L..R)`, то `getSum(L..R) = sum[R+1] - sum[L]`.

Опишем структуру данных, которая умеет делать `reverse`. В каждый момент времени наш новый массив — это последовательность кусков $[L_1..R_1], [L_2..R_2], \cdots, [L_k..R_k]$ старого массива. Каждый из кусков может быть перевернут, т.е. мы помним флажки $isReversed_i$. На запрос `getSum(L..R)` можно отвечать за время $O(k)$. Все, что нам нужно — поддерживать массив в таком виде. Изначально пусть у нас есть один кусок $[1..N]$ и $isReversed_1 = 0$. Вопрос: что происходит при операции `reverse`? Ответ: какие-то два куска $[L_i..R_i]$ и $[L_j..R_j]$ разобьются на два более маленьких. Также некоторый отрезок кусков теперь перевернется. Каждый кусок на этом отрезке сам по себе тоже перевернется, т.е. $isReversed_i$ поменяется на противоположный. При этом количество кусков k каждый раз увеличивается на 2. Чтобы k не росло бесконечно, когда k станет больше $\sqrt{N}$, мы перестроим всю структуру данных за $O(N)$ и количество кусков опять станет равно 1. *Амортизационное время* на один запрос получилось $\sqrt{N}$.

• **Split & Merge** — а еще для решения той же задачи можно поддерживать инвариант, что размер каждого куска от K до $2K$. Чтобы его поддерживать, нам нужны операции `Split` и `Merge` для кусков. При $K = \sqrt{N}$ мы получаем ту же асимптотику. Константа же получится заметно хуже.

• **As Treap** — Если вы помните, что декартово дерево удобно тем, что, чтобы посчитать любую $F(L..R)$, нам достаточно высплитить отрезок $[L..R]$, вернуть F, хранящуюся теперь прямо в корне, и вмерджить $[L..R]$ обратно, то вам будет приятно узнать следующее. Если для изучаемой нами сейчас структуры реализовать такие же операции `GlobalSplit` и `GlobalMerge` (для этого нам понадобится реализовать корневую именно так, как предлагается в предыдущем пункте). то появится тот же эффект, что и в декартовом дереве.

• **2D корневая** — Пусть у нас есть N точек на плоскости с координатами от 1 до N. Все x-ы и y-ы различны. Плоскость можно делить на $\sqrt{N} \times \sqrt{N}$ ячеек. Для каждой ячейки посчитать количество точек в ней и на матрице ячеек посчитать двумерные частичные суммы (т.е. научиться отвечать на запрос "сколько точек в прямоугольнике ячеек"за $O(1)$). Тогда, мы за $O(\sqrt{N})$ умеем отвечать на запрос "сколько точек в прямоугольнике?". Для этого нужно хвосты (не более $\sqrt{N}$ точек слева, справа, сверху, снизу) обработать отдельно и получить прямоугольник-запрос из цельных ячеек, ответ на который можно узнать с помощью предподсчитанных частичных сумм за $O(1)$.

• **Отложенные операции** — Еще раз научимся обрабатывать все те же операции `a[i] := x` и `getSum(L..R)`". Сперва для исходного массива предподсчитаем суммы на префиксах (массив `sum`). Теперь, если мы не пользуемся операцией `a[i] := x`, то `getSum(L..R) = sum[R+1] - sum[L]`. Пусть произошло k операций вида $a[i_j] := x_j$. Сперва преобразуем их к виду $a[i_j]$+=y_j. Для этого на момент выполнения операции нужно помнить содержимое ячейки $a[i_j]$. Теперь запрос `getSum(L..R)` обрабатывается почти так же: изначально результат = `sum[R+1] - sum[L]`. Затем мы перебираем k изменений исходного массива, и, если изменение относится к нужному нам отрезку $[L..R]$, меняем ответ. Время работы $O(k)$. k постоянно растет. Применим уже известный нам прием: если k стало больше $\sqrt{N}$, построим частичные суммы для нового массива, k станет равным 0.

• **Длины строк всегда малы** — Пусть есть задача вида *даны N строк суммарной длины L*, тогда иногда полезно использовать тот факт, что различных длин будет $O(\sqrt{L})$ (пусть у нас есть строки длин $1, 2, \cdots, L$, тогда их суммарная длина $\frac{L(L+1)}{2}$. Пример. Пусть мы умеем с помощью хэшей искать в тексте T строку S за время $O(|T|+|S|)$, вернее $O(|T|)$ т.к. если текст короче строки, то правильный ответ "не найдена". Решение такое: посчитаем хэш строки и хэши всех подстрок текста нужной длины. Теперь пусть нам дали более сложную задачу *дан текст T и N строк S_i, нужно найти эти строки в тексте.* Если мы применим уже известный алгоритм, то получим решение за $O(|T|N)$. Теперь сперва научимся решать за $O(|T|)$ задачу в случае, если все длины строк одинаковы: сложим хэши всех сторок в хэш-таблицу и пройдемся один раз по хэшам подстрок текста. А, если решение этой задачи понятно, то общий случай мы умеем решать за $O(|T|\sqrt{N})$ т.к. различных длин, как мы и говорили в самом начале, мало.

• **Обобщение метода двух указателей** — Научимся в *Offline* решать такую задачу: нужно для отрезков $[L_i..R_i]$ массива a посчитать $\sum\limits_x x \cdot count(x)$. Сумма берется по всем числам на отрезке, а $count(x)$ — сколько раз x встречается на отрезке. Предположим, что числа в массиве целые от 1 до 10^6. Тогда для одного отрезка мы умеем вычислять нужную сумму за $O(R-L)$. Пусть мы знаем сумму для `[L..R]`, получим сумму для `[L..R+1]`. Мы знаем `a[R+1]`, мы знаем `count(a[R+1])`, мы умеем менять `count(a[R+1])`, этого нам должно хватить :-) Таким же образом можно делать не только операцию `R++`, но и `L--`, `L++`, `R--`. Теперь вернемся к исходной задаче. Если мы знаем, что $L_i \leqslant L_{i+1}, R_i \leqslant R_{i+1}$, то ответить на все запросы мы можем за суммарное время $O(N)$. Это и есть классический метод двух указателей. Вопрос: что же делать, если отрезки произвольные? Ответ: давайте все отрезки разобьем на группы: i-я группа — все отрезки, левый конец которых лежит в диапазоне `[iK..(i+1)K-1]`. Если в i-й группе m_i отрезков, то ее можно обработать за время $O(N + Km_i)$. Для этого нужно отсортировать по возрастанию R все отрезки, посчитать сумму для первого, а к следующему переходить с помощью операций `L--`, `L++`, `R++`. Операция `R++` за все время случится не более N раз. Операции `L--` и `L++` вызываются не более K раз для перехода к следующему отрезку (т.к. все левые концы "близки"). Просуммируем полученное время по всем i при $K = \sqrt{N}$. Получим $O(N\sqrt{N})$.

4 2D-деревья

• **КД-дерево** — Пусть есть N точек на плоскости, и пусть вы знаете, что такое дерево отрезков и умеете с помощью него обрабатывать запросы на отрезке вида `+= на отрезке` и `min на отрезке` за $O(\log N)$ спуском сверху вниз. Утверждается, что запросы на прямоугольнике обрабатываются аналогично, но не за $O(\log N)$, а за время $O(\sqrt{N})$.

Для этого посмотрим на процедуру построения дерева отрезков чуть по-другому: у нас есть множество из N точек на прямой, на каждом уровне мы делим это множество на две части по x и строим от половинок структуру рекурсивно. Построение `КД-дерева` отличается лишь тем, что на нечетном уровне мы точки будем делить пополам по x, а на четном по y. Запрос к `КД-дереву` от запроса к дереву отрезков ничем не отличается.

Полученная нами структура строится за $O(N \log N)$, занимает $O(N)$ памяти и на один запрос отвечает за real time $O(\sqrt{N})$. Почему $O(\sqrt{N})$? Представьте, что у нас есть большой прямоугольник-запрос и маленькая прямоугольник-вершина дерева, которая пересекается ровно с одной из четырех сторон запроса. Тогда каждый второй раз мы будем уходить в обе ветки рекурсии, а каждый второй раз из одной сразу отсекаться. Т.е. ширина дробления $2^{\frac{\log_2 N}{2}} = \sqrt{N}$.

• **Квадро-дерево** — эта идея чаще всего применяется в геометрии. Суть в следующем — разделить плоскость на 4 части некоторым образом (например, пополам по x и пополам по y) и продолжить построение рекурсивно. Строить, пока мы не получим элементарные куски плоскости. Что такое элементарный кусок? Если мы делим грид $W \times H$, то это одна клетка. Если на плоскости есть множество из N точек, то кусок, содержащий не более одной точки. Если мы приближенно считаем площади фигур, то кусок площади не более ϵ. В общем, зависит от задачи. Не путайте *квадро-дерево* с *КД-деревом*!.

• **Сколько точек в полуплоскости?** — Даны N точек на плоскости. Сейчас мы научимся в *Online* отвечать на запрос "сколько точек в полуплоскости"за время лучшее, чем $O(N)$. Давайте предположим для простоты, что никакие три точки не лежат на одной прямой. Сперва за $O(N \log N)$ научимся искать две перпендикулярных друг другу прямых, которые делят плоскость на четыре части, в каждой из которых ровно $\frac{N}{4}$ точек (естественно, с точностью до округления). Утверждается, что если зафиксировать угол α, то можно построить две такие перпендикулярные прямые, соответственно с углами нормалей α и $\alpha + \frac{\Pi}{2}$, что каждая из них делит множество точек на две равные части. Проблема в том, что в соседних четвертях может оказаться A и B точек ($A \neq B$). Рассмотрим $F(\alpha) = A - B$. Заметим, что $F(\alpha) = -F(\alpha + \frac{\Pi}{4})$. Значит, есть корень, найдем его бинпоиском. Все :-) Теперь самое простое. Если есть разбиение плоскости таким образом, то какой бы запрос-полуплоскость нам не пришла, прямая, образующая полуплоскость пересекает не более трех из четырех частей плоскости. А четвертая или целиком внутри, или целиком снаружи. Осталось построить квадро дерево. Время обработки одной полуплоскости будет $3^{log_4 N} = N^{\frac{log_2 3}{2}}$, что примерно равно $N^{0.792}$.

• **Простая реализация 2D-дерева** — Когда говорят страшные слова *двухмерное дерево отрезков*, большинство людей представляют себе длинный и сложный код. Это не всегда правда. Рассмотрим такую задачу: дан массив, нужно в *Online* отвечать на запрос "сколько точек на отрезке `[i..j]` имеют значения от L до R?". Решение: построим дерево отрезков, в каждой вершине которого будем хранить отсортированный массив (т.е. все числа этого отрезка в отсортированном порядке). Для того, чтобы код был покороче, реализуем дерево

отрезков снизу.
Самая простая процедура построения нашей структуры такова:

1. каждое число x массива положим циклом `for` в нужные вершины дерева отрезков
2. отсортируем массивы всех вершин дерева отрезков

Самая простая процедура ответа на запрос теперь такова:

1. поднимаемся по дереву снизу, какие-то вершины дерева отрезков покрывают наш отрезок `[i..j]`
2. для каждой такой вершины бинпоиском найдем R и $L-1$, разность полученных индексов нужно прибавить к ответу

Для тех, кто пишет на C++: вам достаточно функций `sort` и `lower_bound`, но для более быстрого построения вместо `sort` можно также использовать функцию `merge` от уже посчитанных детей.

• **Чуть более сложная задача** — пусть даны произвольные N точек на плоскости, опять же нужно научиться считать количество точек в прямоугольнике в *Online*. Давайте отсортируем наши точки по x-ам. i-я точка имеет теперь координаты x_i, y_i и $x_i \leqslant x_{i+1}$. Тогда прямоугольный запрос $[x_1..x_2] \times [y_1..y_2]$ превращается в запрос на отрезке отсортированного массива `[i..j]` "сколько элементов имеют y от y_1 до y_2?". Числа i и j предлагается найти бинпоиском.

• **Замена координат** — пусть даны произвольные N точек на плоскости, нужно научиться считать что-нибудь на прямоугольнике в *Online*. Если вас смущают равные x или y, можно сделать замену координат $x, y \to z = (x, y)$ и $t = (y, x)$ (z и t — это пары чисел, сравниваются на больше-меньше они именно как пары). Тогда прямоугольный запрос $[x_1..x_2] \times [y_1..y_2]$ перейдет в $[z_1..z_2] \times [t_1..t_2]$, где $z_1 = (x_1, y_1)$ и т.д.

5 Классификация BST

• **Не сбалансированное дерево поиска** — чтобы освоить все последующие деревья, нужно хорошо знать, как добавляется и удаляются вершины в обычном несбалансированном дереве. **Добавление:** спуститься вниз, создать новый лист. **Удаление:** найти нужную вершину p, если у p есть только один сын — очевидно. Если у p два сына, перейти к левому, от него спуститься вправо до упора, получить вершину q, поменять местами ключи p и q и удалить вершину q (а у нее, мы точно знаем, нет правого сына). Теперь, когда мы умеем строить обычное дерево поиска, осталось изучить методы и инварианты для балансировки.

• B, B^+ — инвариант: все листья имеют одинаковую глубину, а каждая вершина имеет степень от k до $2k-1$. Корню разрешается иметь степень меньше k. В B^+ дереве ключи хранятся во всех вершинах, в B дерево только в листьях. В B^+ дереве в вершине степени d хранится $d-1$ ключ (между каждой парой ссылок вниз один ключ). Как делать операции добавления/удаления я описывать не буду. Это можно или придумать самостоятельно, или прочитать в википедии.

• **2-3-4, 2-3** — оба дерева это вариации B^+ дерева. 2-3 — B^+ для $k = 2$, в 2-3-4 мы дополнительно разрешили вершинам иметь степень 4. Про 2-3 деревья интересно то, что быстрая реализация такого дерева работает быстрее многих других (`AVL`, красно-черное, `set` из `C++/STL`, `treap`).

• **Red-Black, AA** — дерево называется красно-черным, если все вершины раскрашены в два цвета (красный и черный), и выполняются следующие инварианты:

1. У каждой вершины количество детей или 2 или 0.
2. Все листья черные
3. На пути от корня до любого из листьев одинаковое количество черных вершин
4. Отцом красной вершины не может быть красная вершина

Теперь заметим интересный момент: если каждую красную вершину стянуть с ее отцом в одну большую вершину, получится 2-3-4 дерево. Про 2-3-4 дерево мы помним, что без числа 4 на самом деле можно обойтись. А теперь попробуем 2-3 дерево обратным преобразованием переделать в красно-черное. Мы получим особое красно-черное дерево, в котором все красные деревья являются левыми детьми. Такое дерево проще пишется (меньше случаев) и имеет особое название — `AA`-дерево.

Еще один момент: B^+ дерево, а значит и 2-3 дерево просты. Просты тем, что там нет случаев. Процедура добавления состоит только из одной операции вида "если степень слишком большая, делай так". Поэтому, чтобы запомнить красно-черное дерево (в реализации которого случаи как раз имеются в большом количестве), по-моему, проще представлять себе красно-черное дерево, как 2-3 дерево.

• **Вращения** — следующая группа деревьев использует для перебалансировки так называемые малые и большие вращения. Малое вращение: пусть есть дерево `(a)x((b)y(c))`, тогда из него можно сделать новое `((a)x(b))y(c)`. И обратно. Это малый поворот вокруг ребра `x-y`. Большое вращение выглядит так: `((a)x((b)y(c))z(d)` $\to$ `((a)x(b))y((c)z(d))`, вершина y при этом прыгает на две ступени вверх и становится корнем. Для реализации полезно знать, что большое вращение выражается через малое: мы сперва повернули вокруг ребра `x-y`, затем вокруг ребра `y-z`.

• **Pre-Splay** — работает не за логарифм, но хорошо илюстрирует общую идею. Давайте сделаем добавление также, как в несбалансированное. А теперь с помощью малых вращений переместим только что добавленную вершину в корень. Как это сделать? С помощью одного малого вращения мы можем поднять вершину на единицу вверх по дереву.

• **Splay** — Модифицируем описанную выше идею. Будем поднимать вершину не на один, а на два (возможно, из-за четности, в самом конце придется сделать один подъем на единицу). При подъеме на два вверх есть два случая: `zig-zag` (дерево устроено как при большом вращении, тогда сделаем большое вращение) и `zig-zig` (дерево устроено так `(((a)x(b))y(c))z(d)`, тогда мы дерево преобразуем так: `(a)x((b)y((c)z(d)))`). Заметим, что `zig-zig` не выражается через *малое вращение*. *Амoратизированное время* обработки одного запроса в `Splay` дереве $= O(\log N)$. Чтобы это было так нужно всегда после того, как мы спускались до вершины v и тратили на это свое драгоценное время после этого не забывать поднимать ее в корень. Этой перебалансировкой мы аморатизируем время спуска.

• **AVL** — Инвариант `AVL`: $|height_{left} - height_{right}| \leqslant 1$. Добавление в `AVL` дерево происходит так: сперва добавляем также, как в несбалансированное. Теперь поднимаемся вверх, если в текущей вершине нарушен инвариант, малое и большое вращения нам в руки, все получится.

• **ChinaTree** — Инвариант `ChinaTree`: $|\frac{size_{left}}{size_{right}}| \leqslant 2$ и наоборот. Также допускается случай, что один из размеров 1, а другой 0. Добавление в `ChinaTree` дерево происходит также, как в `AVL`, только функция балансировки проще. Утверждается, что если вращать (понятно в какую сторону) малыми вращениями все, что не сбалансированно, то, во-первых, процесс сойдется, во-вторых, аморатизированное время работы будет $O(\log N)$.

• **Treap, RBST** — Здесь предполагается, что читатель уже знаком с операциями `Split` и `Merge`. Инвариант декартового дерева (`treap`): у каждой вершины есть свой y. Это случайное число. По величине y вершины образуют кучу (в корне минимум). Из определения видно, что, если все y различны, корень определяется единственным способом, и все оставшиеся вершины однозначно делятся на левое поддерево и правое поддерево. Т.е. получается, что корень — случайная из N вершин. Тут нужно как раз сказать, что такое `RBST`. Random Balanced Search Tree. Это такое дерево, в котором корнем является случайная вершина. Разница в том, что y-ов в `RBST` нет, вместо этого в операции `Merge`, чтобы выбрать новый корень, в соответствии с инвариантом, корнем с вероятностью $\frac{L_{size}}{L_{size}+R_{size}}$ становится корень левого дерева. Иначе, корень правого дерева. Плюс `RBST` перед декартовым: не нужны y-и, благодаря этому получается персистентность. С точки зрения памяти: y-ки хранить не нужно, `size` хранить нужно. Т.е. `RBST` по памяти никогда не хуже, а иногда даже лучше. С точки зрения скорости: `RBST` в два раза медленнее. Все потому, что рандом по ходу выполнения генерировать слишком долго.

• **Выражение операций друг через друга** — Для всех деревьем есть операции `Add` и `Del`. Для декартовых деревьев мы видели операции `Split` и `Merge`, в Splay-дереве видели на самом деле операцию `MakeRoot` — сделать вершину корнем дерева. Давайте поймем, какая из выше описанных операций (какой подход) полезнее, мощнее.
Утверждения:

1. `Add` и `Del` выражаются через `Split` и `Merge`
2. `Split` и `Merge` выражаются через `MakeRoot`.
3. Операции `Split` и `Merge` позволяют использовать возможности *дерева по неявному ключу*
4. Операция `MakeRoot` позволяет сэкономить время в том случае, если к одной и той же вершине мы обращаемся много раз.

Обоснования:

1. `Add(x)` : Split T by x to L and R. Merge L, (x), R.
2. `Split T by x` : MakeRoot(x). L = T.L and (T.x), R = T.R
3. Здесь для тех, кто не знает, что такое *возможности дерева по неявному ключу*, я объясню это. Если массив хранить, как дерево по неявному ключу, то появляются новые операции: поменять два куска массива местами, вставить в середину массива, удалить из середины массива. При этом обращение по индексу все еще доступно.
4. Пусть в `Splay` дереве мы всегда обращаемся только к одному элементу. Тогда первым же запросом он поднимется в корень, далее обращения к нему будут быстрее. Если мы обращаемся только к каким-то k элементам утверждается, что *аморатизированное время* обработки одного запроса будет $O(\log k)$.

6 Преобразование операций

• **Add → Build** — чтобы построить дерево из N элементов, можно все их по очереди добавить.

• **Merge → Add** — если мы умеем `Merge`-ить, то чтобы добавить, нужно с-`Merge`-ить с одноэлементным множеством. Например, так устроены биномиальные (сливаемые) кучи.

• **Build, Add → Merge** — изначально все структуры построены операцией `Build`. Пусть теперь мы хотим какие-то две слить в одну. Перекинем меньшую в большую поэлементно операцией `Add`. Утверждается, что, каждый элемент будет добавлен $O(\log N)$ раз, т.к. после очередного добавления размер структуры, в которой он живет увеличился хотя бы в два раза.

• **Build → Merge** — что делать, если мы не можем перекидывать по одному элементу, если у нас нет операции `Add`? Придумаем новую идею на примере задачи *даны N точек на плоскости, нужно делать какой-то запрос на прямоугольнике.* Т.е. по сути мы сейчас хотим научиться *сливать 2D-деревья.* Будем хранить текущие K точек как $\log K$ 2D-деревьев, каждое из которых состоит из 2^x точек. Все x-ы различны. Теперь есть две структуры такого вида, мы их можем сложить как числа в двоичной системе счисления. Т.е. когда мы видим два множества из одинакового количества точек — 2^x, мы порождаем новое дерево из уже 2^{x+1} точек. Как мы это делаем? Просто вызываем процедуру `Build`. Почему это быстро работает? Потому что, когда точка участвует в очередном `Build`-е, размер множества, в котором она живет, удвоился. Т.е. каждая точка будет участвовать в `Build`-ах не более $\log N$ раз. Мы получили, что суммарное время на все `Build`-ы равно $Build(N \log N)$. За сколько работает `Get` в новой структуре? Мы вызовем старый `Get` от каждой из $O(\log N)$ частей, т.е. за $O(Get \cdot \log N)$.

• **Build → Merge, Add** — мы уже умеем делать `Build` → `Merge` и `Merge` → `Add`. Так что все ок.

• **Add = Change + Offline** — Все, о чем мы говорили выше работало в *Online*, теперь предположим, что все запросы добавления даны нам в *Offline*. Рассмотрим задачу "запрос = посчитать количество точек в прямоугольнике". Можно сказать, что у каждой точки есть значение, и изначально все не добавленные точки имеют значение 0, а добавление точки = поменять ее значение на 1. Запрос теперь превратился в найти сумму значений точек в прямоугольнике. Зато добавлять новые точки (что для 2D-дерева операция не простая) уже не нужно.

• **Del = Find** — Чтобы удалить элемент, например, в сбалансированном дереве, часто достаточно найти его и сделать специальную пометку *нет тебя!* Таким образом, если вы умеете добавлять в `AVL`, но не помните, как удалять. Ничего страшного, просто не удаляйте.

• **Del = do not Del** — На разборе вы узнаете более мощный способ, как можно не удалять, если все запросы даны в *Offline*. Тут удаление будет рассматриваться, как `отмена добавления`, все запросы будут разбиты на пары (добавить-удалить). Т.е. просто каждое добавление будет действовать на отрезке `[L..R]`. Собственно способ можно прочесть в разборе задачи `A`. А сейчас реклама: дан граф, добавляются и удаляются ребра, нужно говорить в каждый момент времени, сколько компонент связности. Запросов 10^5, `TL` $= 0.5$ секунд.