t(n, input) - время работы алгоритмы при входных данных input размера n. Тогда время работы алгоритма $t(n) = max_{input} t(n, input)$.

$t(n) = O(f(n)) \Leftrightarrow \exists c > 0,\ N > 0 : \forall n > N\ t(n) \leq c \cdot f(n)$.

$t(n) = o(f(n)) \Leftrightarrow \forall c > 0, \exists N : \forall n > N\ t(n) \leq c \cdot f(n)$.

$t(n) = \Omega(f(n)) \Leftrightarrow \exists c > 0,\ \forall N,\ \exists n > N,\ input : t(n, input) \geq c \cdot f(n)$.

$t(n) = \omega(f(n)) \Leftrightarrow \forall c > 0,\ \forall N,\ \exists n > N,\ input : t(n, input) \geq c \cdot f(n)$.

$t(n) = \theta(f(n)) \Leftrightarrow t(n) = \Omega(f(n)),\ t(n) = O(f(n))$.

Алгоритм является полиномиальным, если $t(input) = O(|input|^k)$. $|input|$ - битовая длина.

Сильно полиномиальный алгоритм - $t(n) = O(Poly(n))$ - string-poly

Слабо полиномиальный алгоритм - $O(Poly(n, \log C))$ - weak-poly

Псевдо полиномиальный алгоритм - $O(Poly(n, C))$ - pseudo-poly

- unit test (обычные, запускаем просто тесты)
- integration (разные компоненты программы нормально живут вместе)
- prod (тестирование от самого начала до конца)

Вот нам split и merge ДД рассказали.

Insert Генерируем вершину с нашим ключом и случайным приоритетом. Сплитим дерево по x, который хочется вставить. Мёрджим левое поддерево с вершиной, а потом полученное с правым поддеревом.

Erase Сплитим по x + 1, потом левое поддерево по x. Затем мёрджим два крайних дерева.

Обсудим, как удалить элемент, если нельзя инкрементировать, а можно только сравнивать.

1 способ. Пусть удаляемый ключ - A. Найдём следующий ключ в ДД - B. Сплитим по B, потом левое по A. Мёрджим два крайних дерева.

2 способ. Делаем два сплита (первый отправляет равные ключи влево, второй отправляет равные ключи вправо). Тогда, с помощью таких операций можно выразить удаление.

Ещё один способ вставки (на практике быстрый) Спускаемся по бин дереву особо не задумываясь. Идём вниз по бин дереву, пока не нарушаются условия для приоритетов. Когда условия нарушилось, берём это поддерево, сплитим его и ставим нашу вершину на это место, а левыми и правыми сыновьями делаем расспличенныедеревья.

Ещё один способ удаления (аналогичный) Ищем элемент. Мёрджим два поддерева дерева, а потом просто подвешиваем полученное дерево на место исходной вершины.

Подсчёт k-ой порядковую статистику. Для каждой вершины храним размер поддерева. Идём по дереву и смотрим на размер левого поддерева. Если он больше, чем текущий счётчик, то идём в левое поддерево, если равен, то мы уже там где надо, иначе уменьшаем счётчик на размер левого поддерева + 1 и идём вправо. Изначально счётчик равен k.

По неявному ключу. Хотим структуру: массив, к индексам которого можно обращаться за log и вставлять в середину за log.

Переделаем split: Делаем разрезы не по ключу, а по количеству (совметим идеи split и k-ой порядковой статистики). Теперь мы не пользуемся тем, что ключи отсортированны и мы можем ключи использовать как какой-нибудь мусор.

Итерироваться по такому массиву можно за линию (каждый переход по ребру туда и обратно занимает одну операцию).

Вращение в массиве: Вырезаем splitom первые k элементов, а потом меняем два поддерева местами и мёрджим.

Ещё всякие приколы: Умеем находить максимум на отрезке. В вершине будем хранить значение максимума в поддереве. Выспличиваем наш подотрезок и смотрим на значение в корне.

Умеем зеркально отражать подотрезки. Будем в каждой вершине хранить модификаторы и если надо - будем их проталкивать вниз. Когда разворачиваем поддерево вершины - просто говорим, что его модификатор = 1. Теперь, когда хотим проталкивать делаем вот что: убираем флажок в нашей вершине меняем ссылки на левого и правого сына и ксорим с единицей флажок детей. Теперь, вершина - абсолютно нормальная.

Построение декартача за линию, когда x уже отсорчены.

Пойдём слево направо. Когда вставляем новую вершину идём снизу вверх от самой большой вершины дерева к корню и смотрим, в какой момент мы можем вставить нашу вершину. Мы ее вставляем как корень соответствующего поддерева, а потом переподвешиваем её туда, куда нужно. Аммортизированно - линия. Потенциал - длина правого пути (упражнение))). При этом правый путь надо хранить стэком.

interleave (как merge, только ключи первого дерева не обязательно меньше). Тут трэш начался)) Если коротко, то мы огкусываем от первого дерева все элементы, которые меньше минимума во втором, потом отрезаем от второго дерева, ну и продолжаем так, пока оба дерева не пропадут. А потом просто мёрджишь по порядку.

Splay tree. Уже пройденные способы балансировки для BST, которые у нас были, это способы из ДД (четкая структура и ожидаемая глубина) и 2-3 дерева (гарантированная одинаковая глубина для всех вершин).

Новый способ балансировки — не напрягаться с балансированием лишний раз, и работать за амортизированное время.

Основной нашей операцией будет $expose(v)$, которая будет превращать вершину v в корень дерева. Условие BST на дереве сохраняется. Эта операция будет поддерживать два типа поворотов: левый и правый. Поворот вершины v возьмет ребро от v к предку u, и его «повернет» — если v было левым сыном u, то u станет правым сыном v. При этом правый сын v станет левым сыном u. Другой поворот делается симметрично. Вершина v станет ближе к корню дерева. Если применять повороты к v, пока можно, то в итоге v станет корнем.

expose будет выполняться после каждой операции, и работать за высоту дерева. Более того, поскольку все остальное тоже работает за высоту дерева, то мы будем оценивать только *expose*.

Каждый раз делать повороты нельзя, потому что будет работать за долго. У нас будет три операции, которые будут поднимать нашу вершину:

- zig
- zig-zig
- zig-zag

zig. Самая тупая операция — если мы сын предка, то делаем поворот. Оставшиеся операции будут поднимать нас сразу на 2.

zig-zig. Если наш дедушка от нас находится справа-справа или слева-слева, то выполним сначала поворот предка, а потом себя.

zig-zag. Иначе мы сначала сделаем поворот себя, а потом поворот предка.

Лучше порисовать картинки или погуглить визуализацию, потому что иначе будет очень непонятно, что сейчас произошло.

Также можно выразить *split* и *merge* через предыдущие операции.

Потенциал. Введем $\Phi(t) = \sum\limits_{v \in t} rank(v) = \sum\limits_{v \in t} \log_2 size(v)$. Тогда стоимость *expose* это

$$\tilde{t_e} = t_e + \Phi(t') - \Phi(t) \leq 3(rank(root) - rank(v)) + 1$$

Воспользуемся $rank(root) - rank(v) = (rank(root) - rank(u_1)) + (rank(u_1) - rank(u_2)) + \cdots + (rank(u_n) - rank(v))$ и разложим *expose* на три операции, и покажем, что они тоже оплачиваются потенциалом.

Формальные грамматики и языки. Обозначим алфавит символов за σ, а множество конечных строк за $L \subset 2^{\sigma^*}$, где σ^* — это все последовательности конечной длины.

Регулярные выражения. Тривиальные регулярные выражения: (пустое множество) ϵ (пустая строка), $x \in \sigma$. Остальные регулярные выражения определяются рекурсивно относительно них. А именно, мы также разрешаем $A+B$ (последовательная запись), $A|B$ (выбор из двух регулярок), $A*$ (повторение строки из языка A $0, 1, 2, \dots$ раз) Звездочка еще называется замыканием $Kleene$.

Например, $\{0|1|2|3\} * +\{0|1|2|3\}$ это множество всех непустых строк из символов $\sigma = \{0, 1, 2, 3\}$.

Способы задания языков. Регулярные выражения, ДКА, НКА, ϵ-НКА. Эти способы эквивалентны. Очевидно $L_{\text{ДКА}} \subset L_{\text{НКА}} \subset L_{\epsilon\text{-НКА}}$ Вложенность ϵ-НКА в ДКА можно показать, если рассмотреть алгоритм приведения одного автомата в другой. Например, можно выделить все подмножества вершин НКА как отдельные вершины ДКА (то есть, он будет иметь экспоненциальный размер). Тогда переход по ребру — это то же самое, что взять все вершины подмножества, и объединить переходы по ребрам из них. Еще стоит следить за переходами по пустому символу, потому что вершина сразу задает подмножество вершин, достижимых из нее по ϵ-ребрам.