Экзаменационные билеты
Алгоритмы. 5 курс. Весенний семестр.

Горбунов Егор Алексеевич

15 июня 2016 г.

1 Декартово дерево

Определение 1. Декартово дерево – это бинарное дерево, в каждой вершине которого хранится пара (k, p), причём декартово дерево является деревом поиска по ключу k и кучей по приоритету p.

Лемма 1. Пускай нам дан набор пар $(k_1, p_1), (k_2, p_2), \ldots, (k_n, p_n)$, причём все p_i различны. Тогда существует единственное декартово дерево, построенное по этому набору пар.

Доказательство. Будем рекурсивно (по индукции) строить декартово дерево. Все приоритеты p_i различны, а значит среди пар можно выбрать *единственную*, у которой приоритет $p_i = p_{max}$ максимальный. Относительно этого приоритета все пары мы можем однозначно разбить на две группы: в «левую» группу отправим пары у которых $p_i < p_{max}$, а в правую те, у которых $p_i \geq p_{max}$. Таким образом разделили исходную задачу размера n на более мелкие две. Построим соответственно декартово дерево на «левой» и «правой» группе и присоединим к корневой паре с приоритетом p_{max}. Дерево построено, значит существует. На каждом уровне выбор корня делается единственным способом (тут используем единственность приоритетов), а значит дерево единственно. □

Замечание 1. Алгоритм, описанный в доказательстве леммы отработает за $\mathcal{O}(n^2)$.

1.1 Построение за линейное время

Пускай нам дан *уже отсортированный* по ключам в порядке возрастания набор пар $(k_1, p_1), \ldots, (k_n, p_n)$. Построим по ним декартово дерево быстро – за $\mathcal{O}(n)$. Декартово дерево – бинарное дерево поиска по ключам k_i, а это значит, что при добавлении ключей в порядке возрастания, добавляемая вершинка должна оказаться самой правой (у нас в левой веточке ключи $< k$, а в правой $\geq k$). Заметим таким образом, что мы всегда можем добавлять новую вершину x в правую ветвь декартова дерева:

1. В самой правой ветви найти такие две вершины a и b (b – непосредственный сын a), что $a.p > x.p$ и $x.p > b.p$ (всякие граничные случаи где $a = NIL$ или $b = NIL$ не берём в голову, сейчас пофиг)

2. Перекинуть ссылки на сынишек: $x.l = b,\ a.r = x$

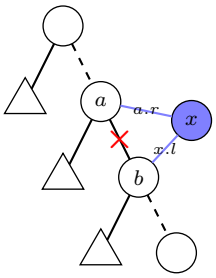


Рис. 1: Вставка вершины x при построении декартова дерева

Как находить такие вершины a и b? Можно спускаться от корня вниз и вправо. Но тогда представим себе такую ситуацию: на вход подали набор пар $(1,1),(2,2),(3,3),\ldots,(n,n)$. Пары отсортированы как надо. На каждой итерации алгоритма построения дерева новая вершина будет добавляться в самый конец пути вправо, таким образом для нахождения вершины a мы каждый раз будем спускаться вниз проходя по i вершин на i-ой итерации. Это как-то квадратично, а мы хотим за $\mathcal{O}(n)$.

Будем искать место для вставки с конца правой ветки. Почему это разумно? А потому, что веточка снизу ломается влево каждый раз при добавлении новой вершины там, куда её добавили. Это наводит на разные амортизационные мысли. Имеем дерево T, которое строится.

$$\Phi_i(T) = \text{длина правой ветки дерева перед } i\text{-ой итерацией}$$

Тогда амортизационная стоимость итерации (добавления одной вершинки) равна (тут t — это то, сколько мы прошли по правой ветки снизу в поисках места вставки)

$$\tilde{c_i} = t + \Delta_i\Phi = t + ((l - t + 1) - l) = t - t + 1 = 1 = \mathcal{O}(1)$$

Таким образом амортизационная оценка на построение всего дерева: $\mathcal{O}(n)$!

1.2 Операции вставки и удаления

Для вставки и удаления введём 2 дополнительные операции: `Split` и `Merge`

- *Процедура* `Split` принимает на вход дерево T и ключик k, а возвращает 2 декартовых дерева: в первом дереве все ключики $< k$, а во втором $\geq k$.

```
def Split(T, k):
    if T == nil: return (nil, nil) # nil is empty tree or something like this...
    if T.k < k:
        (T.r, R) = Split(T.r, k)
        return (T, R)
    else:
        (L, T.l) = Split(T.l, k)
        return (L, T)
```

Логика проста: если ключ в корне дерева меньше k, то этот корень уже находится в искомой «левой» половине разбиения (сплита, разреза, как хотите!), а значит «правую» половину разбиения мы берём из рекурсивного разбиения правого сына, а левые остатки присоединяем к итоговой левой части.

- *Процедура* `Merge` принимает на вход два дерева L и R, причём таких, что ключи в первом дереве меньше либо равны ключам во втором и возвращает декартово дерево содержащее ключи обоих деревьев. Опять просто и рекурсивно:

```
def Merge(L, R):
        if size(L) == 0: return R
        if size(R) == 0: return L
        if L.p > R.p:
            L.r = Merge(L.r, R)
            return L
        else:
            R.l = Merge(L, R.l)
            return R
```

Тут мы просто уменьшаем задачу валидным способом...

- *Вставка в дерево*: процедура `Insert`. Процедура будет принимать дерево и новую вершину, а возвращать изменённое дерево.

```
def Insert(T, node)
        L, R = Split(T, node.k)
        return Merge(Merge(L, node), R)
```

Заметим, что мы используем тут целых 2 слияния и 1 сплит. Можно чуть лучше. Если у `node` приоритет выше, чем у `T`, то достаточно посплитить `T`, а потом присоединить результат к `node`. Если же приоритет меньше, то мы можем спуститься по `T` как по дереву поиска до той вершины, где её приоритет будет выше.

```
def Insert(T, node):
    if node.p > T.p:
        node.l, node.r = Split(T, node.k)
        return node
    if node.k < T.k:
        T.l = Insert(T.l, node)
    else:
        T.r = Insert(T.r, node)
    return T # root of result tree is T
```

- *Удаление из дерева*: процедура `Remove`. Принимаем дерево и ключ, а возвращаем дерево, их которого вершинка с ключиком была удалена.

```
def Remove(T, k)
        L, R = Split(T, k)
        D, R = Split(R, k)
        return Merge(L, R)
```

Тут делается 2 сплита и 1 слияние, но можно лучше. Ясно, что если `T.k == k`, то нам просто нужно вернуть слитых левого и правого сына `T`. Этим и воспользуемся:

```
def Remove(T, k)
    if T.k == k:
        return Merge(T.l, T.r)
    if k < T.k:
        T.l = Remove(T.l, k)
    else:
        T.r = Remove(T.r, k)
    return T
```

Замечание 2. Легко видеть, что время работы операций `Split` и `Merge` — $\mathcal{O}(h)$, где h — высота дерева. А значит аналогично и время работы вставки и удаления.

1.3 Дуча (Treap, Дерамида)

Определение 2. Дуча — это декартово дерево, приоритеты в котором случайные и равномерно распределённые.

Лемма 2. Математическое ожидание высоты дучи равно $\mathcal{O}(\log n)$, где n — число вершин в ней.

Доказательство. Обратимся к рекурсивному (тот, что за $\mathcal{O}(n^2)$) алгоритму построения декартова дерева. Заметим, что в силу случайности приоритетов мы равновероятно выбираем любой ключ при построении дерева, относительно которого все остальные ключи будут разбиты по группам $<$ или $\geq$. Это в точности та же процедура, которая происходит при применении быстрой сортировки. И дерево рекурсии для этих задач идентичны. Умеем доказывать, что дерево рекурсии `QuickSort` будет иметь высоту, в среднем (из-за случайности процесса, а не по входам), логарифмическую! □

Выводы: таким образом `Treap` – это дерево поиска, операции на котором будут работать за $\mathcal{O}(\log n)$.

2 Задачи RMQ и LCA

Определение 3. Дан массив A. Задача **RMQ** или `Range Minimum Query` заключается в том, что по запросу $[l, r]$, $l \leq r$, $l, r \in \mathbb{N}$ нужно найти $\min_{i \in [l,r]} A[i]$.

Определение 4. Пусть дано дерево T. Задача **LCA** или `Least Common Ancestor` заключается в том, что по запросу (v, u), где v и u есть вершины дерева T, нужно найти вершину $p \in T$ такую, что p — общий предок v и u и при этом он самый близкий к ним, т.е. нету вершины p', которая общий предок v и u и при этом её глубина больше глубины p.

Задачи `RMQ` и `LCA` можно решать в разных условиях: когда исходный массив или дерево могут меняться (динамическая задача) или когда они даны и неизменны (статическая задача).

Замечание 3. Заранее скажем, что существует круговорот RMQ и LCA в природе, а именно: RMQ сводится к LCA, а LCA сводится к RMQ± 1, что есть частный случай задачи RMQ.

Замечание 4. В лоб задачи LCA и RMQ решаются за $\mathcal{O}(n)$ (ответ на один запрос, на дерево никаких ограничений в духе сбалансированности нет). Но мы научимся (возможно, описано тут некоторое не будет) решать быстрее:

	Дерево отрезков	Полный предподсчёт	Sparse Table
предподсчёт	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$
запрос	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
динамически?	да	нет	нет

Таблица 1: Задача RMQ

	Двоичные подъёмы	Полный предподсчёт	Ladder decomposition
предподсчёт	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$
запрос	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
динамически?	нет	нет	нет

Таблица 2: Задача LCA, если решать именно как LCA :) Под Ladder Decomposition понимается разложения дерева на «длиннейшие» пути, а потом наворачивание поверх этого двоичных подъёмов (самый обычных), в этом документе про это точно не будет.

Так же мы научимся сводить RMQ к LCA и наоборот за линейное $\mathcal{O}(n)$ время, так что методы решения одной задачи можно использовать для решения другой.

2.1 Динамический RMQ и RSQ. Дерево отрезков

Пускай у нас есть ассоциативная операция $\circ$, например $\min$ или $+$. Заметим тогда, что при вычислении $A[l] \circ A[l+1] \circ A[l+2] \circ \ldots \circ A[r]$ мы можем в силу ассоциативности расставлять скобки как хотим. Построим такую структуру данных, которая нам будет разбивать отрезок запроса на некоторое число *непересекающихся* подотрезков, на которых ответ уже посчитан. Тогда, в силу ассоциативности, мы можем получить ответ на всё отрезке. Далее в качестве операции будем рассматривать, например, $+$.

Определение 5. *Дерево отрезков* — это такое бинарное дерево, что его корень ассоциирован со всем массивом $A[1..n]$, а дети вершины, которой соответствует кусок массива $A[l..r]$, ассоциированы с кусками массива $A[l..m]$ и $A[m+1..r]$, где $m = \frac{l+r}{2}$. См. рисунок 2

В каждой вершине такого дерева мы можем хранить ответ на запрос на отрезке, которому эта вершина соответствует.

2.1.1 Построение дерева отрезков

Построить дерево можно за линию. Считаем, что длина исходного массива $n = 2^k$. Всего вершин в дереве $2n$, т.к. это полное бинарное дерево на n листьях (можешь доказать это через лемму о рукопожатиях, например). Само дерево можно хранить в массиве `tree[2*n]` так, что `tree[1]` — это корень дерева отрезков, который соответствует всему массиву. А дети `i`-ой вершины — это вершины `tree[2*i]` и `tree[2*i+1]`. Т.е. отец вершины `tree[j]` — это `tree[j / 2]`.

```
def build(arr):
    tree = [0] * (2 * len(arr) + 1)
    tree[len(arr) + 1:] = list(arr)
    for i in reversed(range(1, len(arr)+1)):
        tree[i] = tree[2 * i] + tree[2 * i + 1]
    return tree
```

Ясно, что строим за $\mathcal{O}(n)$.

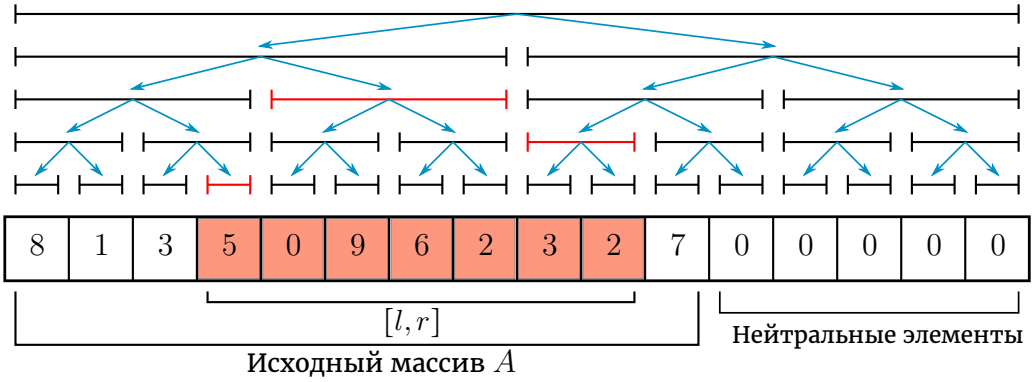


Рис. 2: Дерево отрезков для данного массива A. Если размер A не степень двойки, то добьём до степени двойки нейтральными элементами относительно той операции, которую хотим считать на отрезках массива.

2.2 Ответ на запрос и оценка на время работы

Отвечать на запрос о сумме (минимуме, ...) на отрезке будем декомпозируя искомый отрезок на куски, которые представляют из себя вершины дерева отрезков.

```
def sum(v, l, r, ql, qr):
    """
    v - индекс вершины дерева отрезков в массиве tree
    l, r - границы отрезка, за которые вершина v отвечает
    ql, qr - текущие границы запроса, ql >= l и qr <= r
    """
    if ql > qr:
        return 0
    if l == ql and r == qr:
        return tree[v]
```

```
11      m = (l + r) / 2
12      return sum(2*v, l, m, ql, min(m, qr)) + sum(2*v+1, m+1, r, max(m+1, ql), qr)
```

Нужно понять, как быстро это работает?

Лемма 3. Вызов `sum(1, 0, n-1, l, r)` отрабатывает за $\mathcal{O}(\log n)$.

Доказательство. Заметим следующую вещь. Всего может быть 2 случая:

1. Если какая-то граница отрезка `[l, r]` совпала с какой-то границей отрезка `[0, n-1]`. Видим

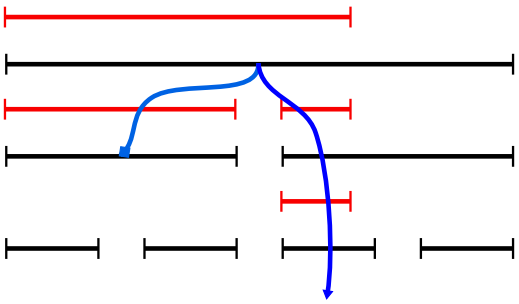


Рис. 3: Хотя бы одна из границ совпала.

   тогда, что у нас один из ветвей рекурсии сразу же остановится и останется ещё всего лишь одна. Т.е. можно считать, что число ветвей рекурсии не увеличивается.

2. Если у нас ни одна из границ не совпала и при этом отрезок `[l,r]` не пересекает середину отрезка, соответствующего вершине, то тогда у нас опять же число ветвей рекурсии на этом шаге не увеличивается.

3. Если же ни одна из границ не совпала, а отрезок `[l,r]` пересекает середину, то ситуация получается такой: Видим тогда, что у нас увеличилось число ветвей рекурсии. Их стало две. Но

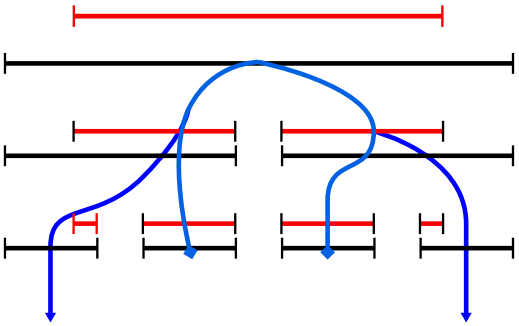


Рис. 4: Отрезок `[l, r]` пересекает середину отрезка вершины

   заметим тогда, что обе эти ветви рекурсии удовлетворяют случаю 2.. А значит, что ни одна из порождённых ветвей рекурсии не расплодится (больше чем на один шаг).

Получили таким образом, что максимальное число ветвей рекурсии равно 4 (это видно на рисунке 4), но это значит, что время работы ответа на запрос пропорционально высоте дерева отрезков, а высота этого дерева, очевидно, $\log n$. Вот и получили $\mathcal{O}(\log n)$ на запрос. □

Замечание 5. Из леммы так же следует, что мы можем любой отрезок разбить на $\mathcal{O}(\log n)$ отрезков из дерева (канонических отрезков). Иначе мы бы не могли получить такой асимптотики.

2.2.1 Обновление элемента массива

Тут всё просто. Каждый элемент массива покрывается ровно $\log n$ каноническими отрезками. Это, к слову, хорошо видно из рисунка 2. Таким образом нам нужно просто обновить значение на всех этих отрезках. Это делается либо проходом сверху вниз, либо снизу вверх (тут нам помогает наш способ хранения дерева через массив и удобная адресация к родителю вершины).

```
def update(v, l, r, idx, val):
    if l == r and l == idx:
        tree[l] = val
    else:
        m = (l + r) / 2
        if idx <= m:
            update(2*v, l, m, idx, val)
        else:
            update(2*v+1, m+1, r, idx, val)
        tree[v] = tree[2*v] + tree[2*v+1] # or min(tree[2*v], tree[2*v+1])
```

Тут совсем ясно, что мы тратим ровно высоту времени, т.е. $\mathcal{O}(\log n)$

Замечание 6. Быстрее делать обе операции (запрос и изменение) нельзя. Если бы было можно, то и массив бы мы отсортировали быстрее, чем за $\mathcal{O}(n \log n)$ (достаём минимум, заменяем элемент по его индексу на ∞), а это теоретически невозможно.

2.3 Статический RMQ. Sparse table

Пускай у нас теперь исходный массив фиксирован и не меняется. Нужно лишь уметь отвечать на запросы минимум на отрезке `[l, r]`.

1. Полный предподсчёт. Можно просто взять и для всех пар подсчитать ответ за $\mathcal{O}(n^2)$, после чего отвечать на каждый запрос за $\mathcal{O}(1)$.

2. **`Sparse Table`** или Разреженная таблица. В данной ситуации нам важно, что операция, которую мы хотим считать на массиве, *идемпотентна*. Заметим такую вещь: если исходный запрос $[l, r]$ мы разобъём на 2 *пересекающихся* подотрезка $[l, l+2^k]$ и $[r-2^k, r]$, то $\min_{[l,r]} = \min(\min_{[l,l+2^k]}, \min_{[r-2^k,r]})$. Тут 2^k максимальное такое, что $r - l + 1 \geq 2^k$. Суть в том, что такие 2 отрезка гарантированно покроют весь отрезок $[l, r]$. Суть метода `Sparse Table` в том, что мы предподсчитываем ответы на запросы на всех отрезках, длина которых — степень двойки. Вплоть до длины $\log n$. Таким образом предподсчёт у нас займёт $\mathcal{O}(n \log n)$. Идемпотентность нам нужна, т.к. отрезки, на которые мы разбиваем запрос, могут пересекаться.

3. Для операций, у которых есть обратная операция, мы можем предподсчитать за $\mathcal{O}(n)$ результаты на префиксах массива, а потом отвечать за $\mathcal{O}(1)$.

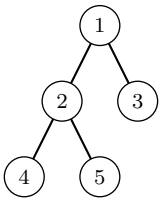
2.4 LCA. Эйлеровы обходы. Сведение к RMQ и наоборот

2.4.1 Сведение RMQ к LCA

Пусть у нас есть массив $A[1..n]$. Рассмотрим набор пар $(A[i], i)$ и построим по нему декартово дерево, где i (индексы в массиве) будут ключами, а приоритетами будут $A[i]$. Ясно, что т.к. полученное дерево будет деревом поиска по индексам и кучей по элементам массива, то наибольший общий предок вершины $(A[l], l)$ и вершины $(A[r], r)$ — это такая вершина $(A[i], i)$, что $l \le i \le r$ и $A[i] \le A[l], A[r]$. Это ясно по построению и свойствам дерева.

2.4.2 Эйлеров обход

Эйлеровы обходы бывают разные. Пускай у нас есть дерево: Эйлеров обход — это своеобразный про-



токол обхода в глубину данного дерева. Эйлеровы обходы бывают следующие:

(a) *Обход по рёбрам.* Обходим дерево начиная с корня по рёбрам слева направо. Проходя по ребру добавляем его к обходу.

$$(1,2),\ (2,4),\ (4,2),\ (2,5),\ (5,2),\ (2,1),\ (1,3),\ (3,1)$$

(b) *Обход по вершинам с повторениями.*

$$1, 2, 4, 2, 5, 2, 1, 3, 1$$

(c) *Обход по вершинам*, где вершина добавляется в обход при входе в её поддерево и при выходе.

$$1, 2, 4, 4, 5, 5, 2, 3, 3, 1$$

Будем эйлеровым обходом называть обход по вершинам с повторениями. Заметим следующую вещь. Пронумеруем все вершины в порядке обхода *в ширину*, т.е. чтобы у вершины с меньшей глубиной всегда был больший номер. Теперь пусть мы хотим найти LCA вершин a и b. Тогда посмотрим на индекс i первого вхождения a в эйлеровом обходе и индекс r первого вхождения j в эйлеровом обходе. Не умаляя общности пускай $i < j$. Что может находится между позициями i и j в эйлеровом обходе?

В силу устройства эйлерова обхода, если b не находится в поддереве a, то между i и j может лежать единственный общий предок a и b и причём он будет наименьшим (`least common ancestor`) и он будет иметь наименьший номер. Если же b лежит в поддереве a, то a и будет ответом на запрос. Заметим, что в силу нумерации получается, что для ответа на запрос нам нужно найти минимум на отрезке $[i, j]$ в эйлеровом обходе. Таким образом будем хранить массив $first[1..n]$ первых вхождений каждой вершины. Если потребуется найти $lca(a, b)$, то нужно спрашивать $rmq(first[a], first[b])$. Свели.

Замечание 7. Какова длина эйлерова обхода? Если смотреть на то, как мы ходим по рёбрам, то можно заметить, что в обход мы добавляем конец каждого ребра (когда идём вних) и начало каждого ребра (когда поднимаемся наверх), плюс ещё корень, который появляется вначале, потому что с него начинаем (очевидно). Итого $2(n-1)+1 = 2n-1$. А значит сведение работает за $\mathcal{O}(n)$.

Замечание 8. Существует алгоритм решения `RMQ`± 1 за $\mathcal{O}(n)$ предподсчёт и $\mathcal{O}(1)$ на запрос. А т.к. мы всё ко всему умеем сводить, то для LCA и RMQ в общем случае тоже можно получить такую асимптотику.

3 Хеширование 1

Довольно часто для решения тех или иных задач нам нужно хранить объекты в множестве, возможно с дополнительной информацией (в таком случае это будет уже словарь или `map`). Абстрактная структура данных **«множество»** должно уметь 3 вещи:

- `Insert` или вставка объекта
- `Delete` или удаление объекта
- `Find` или поиск объекта

Заметим, что если над объектами есть операция сравнения, а точнее линейный порядок, то можно в качестве структуры данных заиспользовать сбалансированное дерево поиска и получить $\mathcal{O}(\log n)$ время работы на всех трёх операциях.

Замечание 9. Сейчас вас ждёт выжимка из Кормена (почти).

Обозначаем U — множество объектов, которые хотим хранить в множестве.

3.1 Прямая адресация. Хеширование. Коллизии

Определение 6. Пускай у нас множество U представляет из себя подмножество целых положительных чисел. Тогда *таблица с прямой адресацией* — это такой массив T размера $|U|$, что $T[k] = NIL$, если элемента k в «множестве» нет, а иначе есть (если что-то в этой ячейке лежит, это могут быть любые сопутствующие данные).

Операции с таблицей с прямой адресацией все работают за $\mathcal{O}(1)$. Минус — размер множества U может быть велик, что плохо. На помощь приходит хеширование.

Определение 7. *Хеш-функция* — это функция из множества объектов U в множество $\{0, 1, \ldots, m-1\}$:

$$h : U \to \{0, 1, \ldots, m-1\}$$

Будем теперь хранить таблицу размера m и для вычисления индекса объекта x в таблице T будем дёргать функцию $h(x)$. Теперь для хранения таблицы нам нужно $\mathcal{O}(m)$ памяти. Такая таблица уже, внезапно, называется **хеш-таблицей**! Но есть проблемы, ведь если $|U| > m$, то по принципу Дирихле (sic!) хотя бы для двух объектов хеши совпадут.

Определение 8. *Коллизия* — это когда $x, y \in U$, $x \neq y$, но $h(x) = h(y)$

3.2 Гипотеза равномерного хеширования

Прежде чем идти рассуждать про разрешения коллизий введём гипотезу.

Определение 9. Пускай есть хеш-функция $h : U \to \{0, 1, \ldots, m-1\}$. Тогда *гипотеза равномерного хеширования* гласит, что для любого объекта $x \in U$ вероятность того, что $h(x)$ равна конкретному значению $i \in \{0, 1, \ldots, m-1\}$, равна $\frac{1}{m}$.

$$\forall i \in \{0, \ldots, m\} \ \ (P\{h(x) = i\} = \frac{1}{m})$$

Т.е. хеш-функция равномерно разбрасывает все объекты по индексам.

3.3 Разрешение коллизий цепочками

Будем в каждой ячейке $T[i]$ хеш-таблицы хранить связный список, тогда операции немного изменятся:

- `Find(x)`. Вычисляем $h(x)$ и линейным поиском ищем x в списке $T[h(x)]$
- `Insert(x)`. Делаем `Find(x)`, если он ничего не нашёл, то добавляем в конец списка.
- `Delete(x)`. Аналогично! За линию от длины цепочки удаляем из неё элемент.

Давайте теперь анализировать... Пускай в хеш-табличке у нас лежит n элементов, а при этом различных индексов (т.е. цепочек) m.

Определение 10. *Коэффициент заполнения* α — это средняя длина цепочки в хеш-таблице.

$$\alpha = \frac{n}{m}$$

Заметим, что все операции у нас работают столько же, сколько и `Find(x)` (почти). Достаточно его

проанализировать, а потом уж если что замечания сделаем.

Теорема 1. Пускай **верна гипотеза равномерного хеширования**. Тогда *среднее* время неуспешного поиска `Find(x)` равно $\mathcal{O}(1+\alpha)$

Доказательство. Ясно, что среднее время неуспешного поиска в пропорционально средней длине цепочки плюс ещё время на вычисление хеша, а значит равно:

$$E[\text{время неуспешного поиска}] = 1 + E[\text{длина цепочки}] = 1 + \alpha$$

□

Замечание 10. Заметим, что среднее время нужное на `Insert(x)`, если x ещё не был вставлен в таблицу, пропорционально $1+\alpha$, т.е. равно $\mathcal{O}(1+\alpha)$ (единичка на саму вставку!).

Теорема 2. Пускай **верна гипотеза равномерного хеширования**. Тогда *среднее* время *успешного* поиска `Find(x)` равно $\mathcal{O}(1+\alpha)$

Доказательство. Заметим следующее: время успешного поиска равно в точности времени неуспешного поиска при вставке, т.к. вставляем мы в конец. Вставить элемент мы могла на одной из n вставок, чтобы найти среднее время успешного поиска нужно усреднять по вставкам!

$$\begin{aligned}
E[\text{время успешного поиска}] &= \frac{1}{n}\sum_{i=1}^{n} E[\text{неуд. поиск при } i-1 \text{ размере таблицы}] = \\
&= \frac{1}{n}\sum_{i=1}^{n} 1 + \frac{i-1}{m} = \\
&= 1 + \sum_{i=1}^{n} \frac{i-1}{m} = [i-1 \text{ т.к. вставка номер } i \\
&= 1 + \frac{1}{nm}\frac{(n-1)n}{2} = 1 + \frac{n-1}{2m} = 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \mathcal{O}(1+\alpha)
\end{aligned}$$

□

Замечание 11. Видно, что в среднем, при верности гипотезы равномерного хеширования, операции с хеш-таблицей работают за $\mathcal{O}(1+\alpha)$.

3.4 Разрешение коллизий пробами

Разрешать коллизии можно по-другому, без затрат на хранение указателей на списки, как это было при использовании цепочек. Для этого расширим понятие хеш-функции.

Определение 11. Хеш-функция: $h : U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}$. Тут второй аргумент хеш-функции — это номер *пробы*.

При выполнении операции вставки мы обращаемся в ячейку по хешу $h(x, 0)$, если эта ячейка занята, то смотрим на $h(x, 1)$ и так далее.

Замечание 12. Ясно, что по-хорошему нужно, чтобы пробы охватывали всю хеш-таблицу T из m элементов, т.е. чтобы $\langle h(x, 0), h(x, 1), \ldots, h(x, m-1)\rangle$ было перестановкой множества $\{0, 1, \ldots, m-1\}$, для любого x.

Замечание 13. Гипотеза равномерного хеширования в данном случае тоже немного меняется, а именно нам хочется, чтобы расширенная хеш-функция $h(x, i)$ выдавала перестановки множества $[0, m)$ равновероятно (перестановок всего $m!$). И вообще было бы круто, чтобы она выдавала все перестановки.

3.4.1 Линейные пробы

Пусть у нас есть хеш-функция $h : U \to \{0, \ldots, m-1\}$. Тогда в методе линейных проб используется расширенная хеш функция:

$$h(x, i) = (h(x) + i) \bmod m$$

Ясно как это работает. Просто последовательно исследуем табличку в случае занятой ячейки.

Замечание 14. Проблема — кластеризация. Пускай есть какая-то длинная цепочка подряд идущих занятых ячеек, тогда в силу вероятности, она имеет тенденцию увеличиваться ещё сильнее, что в конечном итоге приводит к замедлению работы.

Метод линейных проб позволяет получить лишь m перестановок (циклических сдвигов, на самом деле) множества $\{0, \ldots, m-1\}$.

3.4.2 Квадратичные пробы

Аналогично, пусть есть функция $h : U \to \{0, \ldots, m-1\}$, тогда в данном методе используется:

$$h(x, i) = (h(x) + c_1 i + c_2 i^2) \bmod m$$

Тут нужно аккуратно подобрать c_1, c_2, чтобы пробы охватили всю таблицу. В данном случае *кластеризация* будет меньшей, она будет вторичной. Тут опять мы покроем лишь m перестановок, но это уже будут не циклические сдвиги, а что-то чуть более хитрое.

3.4.3 Двойное хеширование

Тут у нас пусть есть две функции: $h_1(x)$ и $h_2(x)$. Обе как-то хешируют объекты множества U. Тогда двойное хеширование использует следующую расширенную функцию:

$$h(x, i) = (h_1(x) + i h_2(x)) \bmod m$$

Чтобы добиться того, чтобы пробы $h(x,i)$ составляли перестановку множества $\{0,\dots m-1\}$ нужно заметить:

$$|\{h_1(x), h_1(x)+h_2(x), \dots, h_1(x)+(m-1)h_2(x)\}| = |\{0, h_2(x), \dots, (m-1)h_2(x)\}|$$

Т.е. нас интересует, чтобы каждое уравнение $i \cdot h_2(x) = j \bmod m$ (для каждого j) имело единственное решение. Это достигается только тогда, когда $gcd(h_2(x), m) = 1$.

Замечание 15. Математику можно подробнее посмотреть в главе 31 Кормена.

3.4.4 Оценки на время работы

Замечание 16. Тут мы оцениваем время работы для некоего сферического в вакууме метода проб. У него всё хорошо и он умеет генерировать *все* перестановки равновероятно.

Замечание 17. Будем считать что верна гипотеза равномерного хеширования, это будет значить при добавлении элемента, вероятность того, что мы просмотрим i-ую ячейку (по количеству), при условии что уже просмотрели $i-1$ ячейку равна:

$$\frac{\text{число ещё не просмотренных заполненных элементов}}{\text{число всех ещё не просмотренных элементов}} = \frac{n-i+1}{m-i+1}$$

Считаем, что у нас везде $n < m$, т.е. таблица не забита до отказа.

Теорема 3. При гипотезе равномерного хеширования время работы неудачного поиска равно $\frac{1}{1-\alpha}$.

Доказательство. X — случайная величина равна *числу проб* проделанных до того, как наткнулись на пустую ячейку. A_i — событие, что мы сделали i-ую пробу. Тогда посчитаем такую вероятность:

$$\begin{aligned}
P\{X \geq i\} &= P\{A_1 \cdot A_2 \cdot \dots \cdot A_{i-1}\} = \\
&= P\{A_{i-1} | A_1 \cdot \dots \cdot A_{i-2}\} P\{A_1 \cdot \dots \cdot A_{i-2}\} = \\
&\vdots \\
&= P\{A_1\} P\{A_2|A_1\} P\{A_3|A_1 \cdot A_2\} \dots P\{A_{i-1}|A_1 \cdot \dots \cdot A_{i-2}\} = \\
&\text{применяем замечание 17} \\
&= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} = \\
&\text{легко видеть, что } \frac{n-i}{m-i} \leq \frac{n}{m} \text{ если } n < m \\
&\leq (\frac{n}{m})^{i-1} = \alpha^{i-1}
\end{aligned}$$

Нас интересует математическое ожидание числа проб, т.е. X:

$$E[X] = \sum_{i=0}^{\infty} iP\{X=i\} = \sum_{i=1}^{\infty} i(P\{X \geq i\} - P\{X \geq i+1\}) = \sum_{i=1}^{\infty} P\{X \geq i\} = \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

Вот собственно и всё, ибо это и есть среднее время при неудачном поиске — среднее число неуспешных проб (проб по занятым ячейкам)! □

Это же время равно времени вставки в таблицу, если элемента там ещё нет.

Теорема 4. Среднее число исследований при успешном поиске, в предположении равномерного хеширования, равно:

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Доказательство. Среднее время при успешном поиске равно усреднённому времени вставки по всем вставкам, ибо элемент, который мы ищем, был когда-то вставлен! Т.е. среднее число исследований равно:

$$E[\text{число проб при успешном поиске}] = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-\frac{i}{m}} = \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{m}} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} =$$
$$= \frac{1}{\alpha}\sum_{i=m-n+1}^{m}\frac{1}{i} \le \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Воспользовались ограничением суммы интегралом, это норм. □

Ну вот как-то так.

4 Хеширование 2

Плохой чел может взять и подобрать последовательность ключей, которые все неким образом упадут в одну цепочку или выстроятся как-нибудь в линию при использовании метода проб. Чтобы как-то это пресечь можно от запуска к запуску выбирать случайную хеш-функцию. Будем об этом говорить.

4.1 Универсальное семейство хеш-функций

Определение 12. *Универсальное семейство хеш-функций* $\mathcal{H}$ — это такое множество хеш-функций $h : U \to \{0, \dots, m\}$, что для любых $x, y \in U$ и *случайно выбранной* хеш-функции $h \in \mathcal{H}$ верно:

$$P\{h(x) = h(y)\} \le \frac{1}{m}$$

4.2 Универсальное семейство для целочисленных ключей

Оказывается можно легко построить универсальное семейство, если U — это целые числа.

Теорема 5. Семейство хеш-функций:

$$\mathcal{H}_{p,m} = \left\{h(x) = ((ax+b) \bmod p) \bmod m \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p^+\right\}$$

является универсальным. Тут у нас p — простое, а $\mathbb{Z}_p^*, b \in \mathbb{Z}_p^+$ — это, соответственно, мультипликативная и аддитивная группы по модулю p, а в силу его простоты это просто множества $\{1, \ldots, p-1\}$ и $\{0, \ldots, p-1\}$.

Доказательство. Рассматриваем два числа x и y. В силу простоты p получается, что $ax + b \bmod p$ не равно $ay + b \bmod p$ (рассматриваем разность этих тождеств). Т.е. коллизий пока не случилось, а все коллизии, которые могут случиться случаются лишь при взятии по модулю m. Т.е. вероятность коллизии — это на самом деле вероятность того, что два числа r и s таких, что $r \neq s \bmod p$, равны по модулю m, т.е. $r = s \bmod m$. Как r так и s могут принимать p возможных значений. Но т.к. $r \neq s$, то для конкретного r существует всего $p-1$ возможных s. А число таких s, что будет верно $r - s = 0 \bmod m$ уж всяко не больше (т.к. это число всех чисел, которые делятся на m меньше p) $\lceil \frac{p}{m} \rceil - 1 \leq \frac{p+m-1}{m} - 1 = \frac{p-1}{m}$. (Тут, как я понимаю, мы имеем право закрепить r и относительно него смотреть сколько s таких, каких нам нужно, ибо нам изначально дано 2 каких-то числа из которых всё и выходит...). А значит, вероятность того, что произойдёт коллизия $\leq \frac{(p-1)/m}{p-1} = \frac{1}{m}$. □

4.3 Универсальное семейство и разрешение коллизий цепочками

Пускай хеш-функция h выбирается из универсального семейства $\mathcal{H}$. Тогда покажем что-нибудь про время на успешный и неуспешный поиск. Помним, что $\alpha = \frac{n}{m}$ — коэффициент загруженности хеш-таблицы.

Введём случайную величину:

$$X_{xy} = \begin{cases} 1, & h(x) = h(x) \\ 0, & h(x) \neq h(y) \end{cases}$$

Тут x, y — объекты, положенные в хеш-таблицу T. Тогда следующая случайная величина — это размер цепочки, в которой лежит объект x.

$$Y_x = \sum_{y \in T, y \neq x} X_{xy}$$

Заметим, что X_{xy} — индикаторная случайная величина, а значит $E[X_{xy}] = P\{h(x) = h(y)\} \leq \frac{1}{m}$ в силу того, что h из универсального семейства.

Теорема 6. $h \in \mathcal{H}$. Тогда математическое ожидание времени, необходимого на:

(a) успешный поиск равно $\mathcal{O}(1 + \alpha)$

(b) неуспешный поиск равно $\mathcal{O}(\alpha)$

Доказательство. Математическое время на поиск у нас пропорционально математическому ожиданию размера цепочки, в которой находится x. Обозначим размер этой цепочки за $E[n_{h(x)}]$

(a) x у нас в таблице есть, но Y_x его не учитывает. А $|T \setminus x| = |T| - 1 = n - 1$

$$E[n_{h(x)}] = 1 + E[Y_x] = 1 + \frac{n-1}{m} = 1 + \alpha - \frac{1}{m} < 1 + \alpha$$

(b) Самого x в таблице нет, т.е. $|T \setminus x| = |T| = n$, а значит:

$$E[n_{h(x)}] = E[Y_x] = \sum_{y \in T} E[X_{xy}] \leq \sum_{y \int T} \frac{1}{m} = \frac{n}{m} = \alpha$$

□

Во дела! В среднем, выходит, всё хорошо.

4.4 Идеальное хеширование

Иногда бывает (тут Корменовский пример о таблице зарезервированном символов в компиляторе), что хеш-таблица строится один раз, а потом уже не изменяется. Тогда, оказывается, что можно сделать её такой, чтобы она работала за $\mathcal{O}(1)$ в **худшем** случае! Как это сделать:

- Идея: в каждой ячейке хеш-таблицы T хранится ещё одна хеш-таблица (второго уровня). Т.е. у нас одна хеш-таблица верхнего уровня и m хеш-таблиц второго уровня
- Хеш-функция на верхнем уровне: $h \in \mathcal{H}_{p,m}$ (т.е. из того универсального семейства, которое мы умеем строить)
- Сначала строим обычную хеш-таблицу с использованием функции h и разрешаем коллизии цепочками. Получаем m цепочек, каждая из которых размера n_i. А далее каждую цепочку превращаем в хеш-таблицу...
- Хеш-функции на втором уровне: $h_i \in \mathcal{H}_{p,m_i}$
- Причём $m_i = n_i^2$ (кажется, что дорого, но в среднем это линия, докажем ниже)

Покажем, что при таком построении мы получим, что вероятность возникновения коллизий КРАЙНЕ МАЛА! (для каждой из внутренних хеш-таблиц (второго уровня))

Теорема 7. Вероятность возникновения коллизии при хешировании n_i в хеш-таблицу размера n_i^2, при использовании случайной хеш-функции из универсального семейства, меньше $\frac{1}{2}$

Доказательство. Всего возможно $\binom{n_i}{2}$ коллизий, каждая с вероятностью $\frac{1}{n_i^2}$ в силу универсального семейства. Тогда:

$$E[\text{число коллизий}] = \binom{n_i}{2}\frac{1}{n_i^2} = \frac{n_i(n_i-1)}{2n_i^2} = \frac{1}{2} - \frac{1}{2n_i} < \frac{1}{2}$$

Теперь вспомним про неравенство Маркова:

$$P\{X \geq t\} \leq \frac{E[X]}{t}$$

Применив его получим, что всё доказано. □

Таким образом нам понадобится пару раз подобрать хеш-функцию из семейства и коллизий не будет. Теперь увидим, что много место вся эта конструкция не займёт, а именно если $m = n$ и мы сохраняем в хеш-таблицу размера m n ключей:

$$\begin{aligned} E\left[\sum n_j^2\right] &= E\left[\sum n_j + 2\binom{n_j}{2}\right] = \\ &= E\left[\sum n_j\right] + 2E\left[\sum \binom{n_j}{2}\right] = \\ &= n + 2E[\text{общее число коллизий}] = \\ &< n + 2\binom{n}{2}\frac{1}{m} = n + n - 1 = 2n - 1 \leq 2n \end{aligned}$$

Круто, типа. В конце пользовались тем, что $n = m$.

Замечание 18. Мне не совсем понятно, зачем нам такое нужно, если при доказательстве мы опираемся на то, что $n = m$. Чем это лучше, чем просто таблица с прямой адресацией?

5 Числовые алгоритмы

5.1 Арифметика

Пусть у нас есть число N. Тогда длина этого числа — это то, сколько бит нужно, чтобы его уместить в памяти компьютера, т.е. это $\beta = \log N$. Все сложности будем мерять именно исходя из того, что размер входа — это β.

- Сложение. Обычное сложение в столбик: $\mathcal{O}(\beta)$.
- Умножение. Умножение с использование сдвига (деления на 2) (или в столбик): $\mathcal{O}(\beta^2)$:

```
def mul(a, b):
    # длины a и b O(β)
    if b == 0: return 0
    res = mul(a, b >> 1) << 1 # время на сдвиги O(β), длина b уменьшается на 1
    if b % 2 == 1:
        res += a
```

  Ещё есть алгоритм Карацубы за $\mathcal{O}(\beta^{\log_2 3})$ и быстрое преобразование Фурье за $\mathcal{O}(n \log n)$.

5.2 Модульная арифметика

5.3 Проверка чисел на простоту

5.4 Генерация случайных простых чисел

5.5 RSA