Формула оценки: **0.25 дз + 0.3 контест + 0.15 кр + 0.3 экзамен + Бонус**
Домашние задания: сдавать устно(раз в неделю ассисты устраивают доп пару, запись онлайн) или latex, дедлайн 10-21 день.
Контесты: Длинные(код ревью), короткие(раз в 2 недели), неточные, бонусные(идет к бонусу). Штрафов нет.
Контрольные работы: раз в модуль, тестовые вопросы.
Бонусы: бонусные контесты, ACM, работа на семинаре.
Материалы:

- Кормен
- en.wikipedia
- викиконспекты
- e-maxx
- Корте-Фанен Коммбинаторная оптимизация

Теория вероятности. $(\Omega, 2^\Omega, P)$ - вероятностная пространство.
$A \subset \Omega$, $P(A) = \sum_{w \in A} P(w)$.
Def: A, B - события, $P(B) > 0$. **P(A|B)** - вероятность события A, если наступило событие B. Тогда $P(A|B) = \frac{\sum_{w \in A \cap B} P(w)}{\sum_{w \in B} P(w)} = \frac{P(A \cap B)}{P(B)}$.
Def: A и B независимые, если P(A|B) = P(A).
Тогда, если A и B независимые, то $P(A \cap B) = P(A) \cdot P(B)$.

$\xi : \Omega \to \mathbb{R}$ - случайная величина. $\xi(w)$ - значение, $w \in \Omega$.
Пример: Есть 5 марок автомобиля, их стоимости и их количества. A - 1000 - 100; B - 2000 - 5; C - 3000 - 5; D - 2000 - 20; E - 1500 - 30; Тогда нас интересуют $P(\xi = 1000) = \frac{100}{160}, P(\xi = 1500) = \frac{30}{160}, P(\xi = 2000) = \frac{25}{160}, P(\xi = 3000) = \frac{5}{160}$.

Матожидание $E(\xi) = \sum_{w \in \Omega} \xi(w) \cdot P(w) = \sum_x x \cdot P(\xi = x)$.

Индикаторная случайная величина: $I_A = \begin{cases} 1, w \in A \\ 0, w \notin A \end{cases}$ Тогда $E(I_A) = P(A)$.

Пусть есть 2 случайной величины ξ_1 и ξ_2. Тогда $E(\alpha\xi_1 + \beta\xi_2) = \alpha E(\xi_1) + \beta E(\xi_2)$.
$E(\alpha\xi_1 + \beta\xi_2) = \sum_{w \in \Omega}(\alpha\xi_1(w) \cdot P(w) + \beta\xi_2(w) \cdot P(w)) = \alpha \sum_{w \in \Omega} \xi_1(w) \cdot P(w) + \beta \sum_{w \in \Omega} \xi_2(w) \cdot P(w) = \alpha E(\xi_1) + \beta E(\xi_2)$

Две случайные величины называются независимые, если $\forall x, y : P(\xi_1 = x \text{ и } \xi_2 = y) = P(\xi_1 = x) \cdot P(\xi_2 = y)$.
n случайных величин называются попарно независимыми, если любые 2 величины независимы.
независимы в совокупности - см семинар

ξ_1 и ξ_2 - случайные независимые величины. Тогда $E(\xi_1\xi_2) = E(\xi_1)E(\xi_2)$.
$E(\xi_1\xi_2) = \sum_{w \in \Omega} \xi_1(w)\xi_2(w)P(w) = \sum_x x \cdot P(\xi_1\xi_2 = x) = \sum_{(u,v)} uv \cdot P(\xi_1 = u \text{ и } \xi_2 = v) = [\xi_1 \text{ и } \xi_2 \text{ независимы}] = \sum_{(u,v)} uv \cdot P(\xi_1 = u) \cdot P(\xi_2 = v) = (\sum_u u \cdot P(\xi_1 = u)) \cdot (\sum_v v \cdot P(\xi_2 = v)) = E(\xi_1)E(\xi_2)$.

Задача о назначениях. Есть n работников и n работ. Есть таблица, где a_{ij} - сколько i-ый работник берет за j-ую работу. Нужно распределить работников по работам так, чтобы суммарная плата за все работы была миниальна. Оценим матожидание затрат при случайном решении. A_{ij} - событие, когда i-ый работник делает j-ую работу. $\xi = \sum_{(i,j)} I_{A_{ij}} \cdot a_{ij}$. Тогда $E(\xi) = \sum_{(i,j)} E(I_{A_{ij}} = \sum_{(i,j)} a_{ij} P(Aij) = \sum_{(i,j)} a_{ij} \cdot \frac{1}{n}$.

Найти максимальный разрез в неориентированном невзвешанном графе.

Будем строить случайный разрез(каждую вершину либо в A, либо в $\bar{A}$). Тогда ξ - величина нашего разреза. $\xi = \sum_{e \in E(G)} I_{B_e}$, где B_e - событие, когда e лежит в разрезе. $P(e \in \text{разрез}) = \frac{1}{2}$. Тогда $E(\xi) = E(\sum_{e \in E(G)} I_{B_E}) = \sum E(I) = \frac{1}{2}|E(G)|$.

Есть перестановка $p_1 \dots p_n$. Алгоритм жадно набирает возрастающую подпоследовательность. Какое матожидание длины этой подпоследовательности?
Событие A_i - алгоритм возьмет p_i. $E(\xi) = E(\sum I_{A_i}) = \sum P(A_i)$. $P(A_i) = P(\forall j < i : p_j < p_i) = \frac{1}{i}$. Тогда $E(\xi) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(logn)$.

Дисперсия $D(\xi) = \sum_{w \in \Omega} P(w)(\xi(w) - E(\xi))^2$.
Свойства:

- $D(\xi_1 + \xi_2) = D(\xi_1) + D(\xi_2)$, ξ_1 и ξ_2 независимы
- $D(\lambda\xi_1) = \lambda^2 D(\xi_1)$

Неравенство Маркова. $\xi : \Omega \to \mathbb{R}_+$. $P(\xi(w) \geq E(\xi) \cdot k) \leq \frac{1}{k}$.

Неравенство Чебышева. $\xi : \Omega \to \mathbb{R}$. $P(|\xi - E(\xi)| \leq \alpha) \leq \frac{D(\xi)}{\alpha^2}$.

Модели
RAM-модель (Random Access Machine) Вопросы, возникающие при создании модели

1. адресация
2. какие инструкции
3. рекурсия
4. где лежат инструкции
5. размер данных
6. кол-во памяти
7. случайность

Адресация Есть ячейки, в которых можно хранить целые числа (ограничения на $MAXC$ разумные, и на них введена неявная адресация

Замечание. Явная адресация — при создании элемента получаем адрес и можем пользоваться только этим адресом. Неявно — можем получать адреса каким-то своим образом, к примеру, $ptr + 20$.

Кол-во памяти Неявное соглашение RAM — время работы не меньше памяти. По дефолту считаем, что мы его инициализируем мусором

Где инструкции Хранить инструкции можно в памяти и где-то снаружи. Мы будем хранить снаружи (внутри — RASP-модель). Иначе говоря, инструкции и данные отделены.

Какие инструкции В нашей модели есть инструкции следующих типов:

- работа с памятью
- ветвление
- передача управления ($=goto$),
- арифметика (at least $a + b, a - b, \frac{a}{b}, \cdot, mod, \lfloor \frac{a}{b} \rfloor$)
- сревнения (at least $a < b, a > b, a \leq b, a \geq b, a = b, a \neq b$)
- логические (at least $\wedge, \vee, \oplus, \neg$)
- битовые операции ($>>, <<, \&, |, \sim, \oplus$)
- математические функции (опять-таки, в рамках разумного)
- rand

Все инструкции работают от конечного разумного числа операндов (не умеем в векторные операции)

Размер данных $\exists\, C, k \ : \ C \cdot A^k \cdot n^k$ — верхнее ограничение на величины промежуточных вычислений.

Рекурсия Рекурсия всегда линейна по памяити относительно глубины.

Случайность Мы считаем, что у нас есть абсолютно рандомная функция. Будем полагать, что у нас есть источник энтропии, выдающий случайности в промежутке $[0, 1]$.

Время работы.

- наихудшее — $t = \max\limits_{input, random} t(input, random)$
- наилучшее — $t = \max\limits_{input} \min\limits_{random} t(input, random)$
- ожидаемое — $E\ t = \max\limits_{input} Average_{random} t(input, random)$
- на случайных данных — $t = Average_{input} Average_{random} t(input, random)$

Алгоритмы

Методы доказательства корректности алгоритма.

1. индукция
2. инвариант
3. от противного

Способы оценки времени работы:

- Прямой учет
- Рекурсивная оценка
- Амортизационный анализ

Прямой учет Время работы строчки — произведение верхних оценок по всем строчкам-предкам нашей.

К примеру

```
while (!is_sorted()) { // O(# inversions) = O(n^2)
    for (int i = 0; i + 1 < n; i++) { // O(n) * O(parent) = O(n^3)
        if (a[i] > a[i + 1]) {
            swap(a[i], a[i + 1]); // O(n^3)
        }
    }
}
```

Рекурсивная оценка Пример — сортировка слиянием

Нас интересует две вещи: инвариант и переход. Для оценки времени используем рекурренту вида $T(n) = O(f(n)) + \sum\limits_{n' \in calls} T(n')$

При этом если мы доказываем время работы, то показываем $T(n) \leq c \cdot f(n)$, зная, что для n' $\exists c$: $T(n') \leq c \cdot f(n)$

Важно, что c глобальное и не должно увеличиваться в ходе доказательства

stable sort Делает сортировку, не меняя порядок равных элементов относительно исходной последовательности. Merge-sort стабилен.

inplace-algorithm Не требует дополнительной памяти и делает все прямо на данной памяти (у нас есть $\log n$ памяти на рекурсию). Quick-sort inplace.

Время работы qsort

$$T(n) = \max_{input} average_{rand} t(input, rand) = \max_{|input|=n} Et(input)$$

$$T(n) \leq \Theta(n) + \frac{1}{n}\sum_{k=0}^{n-1}(T(k+1) + T(n-k)) \leq \Theta(n) + \frac{2}{n} \cdot \sum_{k=1}^{n} T(k-1) \leq a \cdot n + \frac{2}{n}\sum_{k=1}^{n} c \cdot (k-1) \cdot \log(k-1) \leq$$

$$\leq a \cdot n + \frac{2}{n}\sum_{k=1}^{\frac{n}{2}} c \cdot (k-1) \cdot \log n - \frac{2}{n}\sum_{k=1}^{\frac{n}{2}} c \cdot (k-1) + \frac{2}{n}\sum_{k=\frac{n}{2}+1}^{n} c \cdot (k-1) \cdot \log n \leq$$

$$\leq a \cdot n + \frac{2}{n} \cdot n^2 \cdot \log n - \frac{2c}{n} \cdot \frac{(n-2)^2}{4} \leq a \cdot n + cn \log n - \frac{c(n-2)}{4} \leq cn \log n$$

$$\frac{c(n-2)}{4} \geq a \cdot n$$

$$c \cdot n - 2c \geq 4 \cdot a \cdot n$$

$$c \geq \frac{4 \cdot a \cdot n}{(n-2)}$$

, что верно для достаточно больших n.

Ограничение на число сравнений в сортировке Бинарные сравнения на меньше.

Рассмотрим дерево переходов. Для перестановки есть хотя бы один лист — листьев котя бы $n!$

$$L(T) \leq 2^x, d(T) \leq x$$

, если x — ответ

$$L(T) \geq n!$$

$$d(T) \geq \log n! \geq \log \frac{n}{2}^{\frac{n}{2}} = \log 2^{(\log \frac{n}{2}) \cdot \frac{n}{2}} = \frac{n}{2} \cdot \log \frac{n}{2} = \Omega(n \log n)$$

1 Сортировки основанные на внутреннем виде данных

Имеем n чисел $[0, U-1]$, $U = 2^w$, числа укладываются в RAM-модель

Сортировка подсчетом Заводим массив $cnt[U]$, $cnt[x] = |\{i : a_i = x\}|$. Дальше переводим $count \to pref$, $pref[x] = pref[x-1] + count[x]$

$O(n+U)$

Поразрядная сортировка b_{ij}—j-й бит i-го числа. (Сортируем бинарные строки длины w)

Поочередно сортируем строки, разбивая их на классы эквивалентности по $iter$ последним символам. После чего мы стабильно сортируем по $(iter+1)$-му символу.

$O(n \log U)$

Bucket sort Разбиваем множество на корзины, каждой корзине соответствует отрезок. В каждой корзине запускаемся рекурсивно.

$O(n \log U)$

В продакшне используют первую пару итераций, чтобы сильно снизить размерность на реальных данных.

Пусть мы хотим отсортить равновероятные числа из $[0, 1]$. В каждом бакете отсортируем за квадрат. Получим $O(n)$.

$$t(n) \le \sum_{i=1}^{n} c \cdot (1 + E(cnt_i)^2) \le c \cdot n + c \cdot \sum_{i=1}^{n} E(cnt_i^2) =$$

$$= c \cdot n + c \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} EI_{A_{ij}} = c \cdot n + c \cdot n^2 \cdot \frac{1}{n} \le 2 \cdot c \cdot n$$

2 Иерархия памяти

Нас интересует задержка (latency), пропускная способность (throughput). Подгрузка x данных занимает $l + \frac{x}{t}$

От долгой к быстрой

1. external machine / internet
2. HDD
3. SSD
4. RAM
5. L3
6. L2
7. L1
8. registers

3 Алгоритмы во внешней памяти

n — размер задачи
M — размер RAM
B — блок данных
$B \ll M \ll n$
$\log n \ll B$
$B < \sqrt{M}$ (но это неявно и не факт)

Mergesort во внешней памяти Обычный mergesort, но три типа событий в $merge$:

1. Кончился первый буфер — подгружаем новый
2. Кончился второй — аналогично
3. Кончился буфер для слияния — выписываем обратно в RAM и сбрасываем

$O(\frac{n}{B} \log n) \to O(\frac{n}{B} \log \frac{n}{B}) \to O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$
+2 идеи:

1. Дошли до размера M — явно посортим в RAM
2. Можем сливать сразу $\frac{M}{B}$ массивов

1 Простые структуры данных

Требования к структуре данных:

- От СД мы хотим обработку каких-то наших запросов.
- online-offline. Бывает, что мы знаем все запросы, бывает, что мы узнаем запрос только тогда, когда отвечаем на предыдущий
- При обсуждении времени работы отделяется время на препроцессинг и на последующие ответы на запросы (query).

1.1 Data structure / interface

Структура данных — это какой-то математический объект, который умеет отвечать на наши запросы конкретным способом. Красно-черное дерево — это структура данных.
Интерфейс — это объект, с которым может взаимодействовать пользователь, который каким-то образом умеет отвечать на наши запросы (пользователю все равно, как, его волнует только то, что интерфейс реализует, и за какое время(память) он это делает). $std :: set$ — это интерфейс.
Итератор — это специальный объект, отвечающий непосредственно за ячейку в структуре данных. Для того, чтобы удалить элемент, мы должны иметь итератор на этот элемент. Т.е. удаление по ключу работает за $O(erase)$, а удаление по значению за $O(find) + O(erase)$

list Списки бывают двусвязными, односвязными, циклическими. У каждого элемента есть ссылка на следующий (и иногда на предыдущий), а также есть отдельный глобальный указатель на начало списка.

stack Стек — структура данных, которая умеет делать добавление в конец, удаление из конца, взятие последнего элемента, за $O(1)$.

queue Очередь — структура данных, которая умеет делать добавление в конец, удаление из начала, взятие первого элемента, за $O(1)$.

deque Двусторонняя очередь — структура данных, которая умеет делать добавление, удаление и взятие элемента с любого конца последовательности, за $O(1)$.

priority_queue Очередь с приоритетами ака куча — структура данных, которая умеет делать добавление, удаление, и быстрые операции с минимумом, представляющая из себя дерево с условием $parent(u) = v \to value(v) \leq value(u)$.

Все эти структуры реализуются как на массиве (храним последовательную память и указатели на начало/конец), так и на списках (что на самом деле является тем же самым, что и на массиве, просто ссылка вперед эквивалентна $a_i \to a_{i+1}$ в терминологии массивов, *прим. автора*).

Структура данных на массиве кратно быстрее аналогичной на ссылках, потому что массив проходится по кэшу и не требует дополнительной памяти.

data structure	add	delete	pop	find	top	build	min	get by index
stack	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	-
dynamic array	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$
queue	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	-
deque	$O(1)$	-	$O(1)$	-	$O(1)$	$O(n)$	$O(n)$	$O(1)$
linked list	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
sorted array	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(n \log n)$	$O(1)$	$O(1)$
priority_queue	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(n)$	$O(1)$	-	

1.2 Двоичная куча

Реализация двоичной кучи на массиве — создаем массив размера sz, и создаем ребра $i \to 2 \cdot i,\ i \to 2 \cdot i + 1$.

От такой кучи мы хотим:

- insert(x)
- get_min()
- extract_min()
- erase
- change = {decrease_key, increase_key}

Для такой кучи мы реализуем $sift\_up(x),\ sift\_down(x)$ — просеивание вниз и вверх. Процедура должна устранить конфликты с элементом x. Остальные операции умеют реализовываться через нее.

$$insert = add\_leaf + sift_up$$

$$extract\_min = swap(root, last) + last- = 1 + sift_down(root)$$

$$erase = decrease\_key(-\infty) + extract\_min$$

Отдельно отметим построение кучи за $O(n)$ — sift_down поочередно для всех элементов $n,\ n-1,\ \ldots,\ 1$.

```
void sift_up(int v) { // v >> 1 <=> v / 2
    if (key[v] < key[v >> 1]) {
        swap(key[v], key[v >> 1]);
        sift_up(v >> 1);
    }
}

void sift_down(int v, int size) { // indexes [1, size]
    if (2 * v > size) {
        return;
    }
    int left = 2 * v;
    int right = 2 * v + 1;
    int argmin = v;
```

```
    if (key[v] > key[left]) {
        argmin = left;
    }
    if (right <= size && key[right] < key[argmin]) {
        argmin = right;
    }
    if (argmin == v) {
        return;
    }
    swap(key[v], key[argmin]);
    sift_down(argmin, size);
}
```

Какое-то сегодняшнее дополнение про бинарную кучу есть в прошлом конспекте

К-чная куча Куча на полном К-чном дереве. Эту кучу можно так же хранить в массиве, с 0-индексацией.

- $sift\_up$ такой же, $O(\log_k n)$
- $sift\_down$ ищет минимум среди k элементов на каждом шаге, поэтому работает за $O(k \cdot \log_k n)$.

	$2-heap$	$k-heap$
$insert$	$O(\log n)$	$O(\log_k n)$
$extract\_min$	$O(\log n)$	$O(k \log_k n)$
$decreasekey$	$O(\log n)$	$O(\log_k n)$
$increasekey$	$O(\log n)$	$O(k \log_k n)$

Дейкстра на K-ной куче Алгоритм Дейкстры достает минимум n раз, и улучшает ключ m раз

$$a \cdot \log_k n = b \cdot k \cdot \log_k n, a = m, b = n$$

Отсюда $k = \frac{a}{b} = \frac{m}{n}$ в случае Дейкстры. Еще отметим, что $k \geq 2$.

В случае когда $a = b^q,\ q > 1$, то k-куча структура работает за $O(1)$ (вроде бы этот факт мы докажем в домашке), причем с хорошей константой, поэтому применимо на практике (привет, фибоначчиева куча!).

Амортизационный анализ Идея в том, что мы хотим оценить суммарное число операций, а не на каждом шаге работы. То есть вполне может быть итерация алгоритма за $O(n)$, но нам важно, что суммарное число $O(n \log n)$

Так что есть $t_{real} = t,\ t_{amortized} = \tilde{t}$.

Метод кредитов Элементам структуры сопоставляем сколько-то монет. Этими монетами элемент «расплачивается» за операции. Также мы накидываем сколько-то монет на операцию. Запрещаем отрицательное число монет. Начинаем с нулем везде.

Обозначим состояния структуры за $S_0, S_1, \ldots, S_n$. Каждый переход стоил t_i, $t_i \geq |operations|$, Где t_i — это сколько мы потратили. Также на i-м шаге мы вбрасываем в систему $\tilde{t}_i$ монет. Тогда

$$\sum t_i \leq \sum \tilde{t}_i \leq A \to O(A)$$

Стек с минимумом

- min_stack
- push
- pop
- get_min

$m_i = \min(m_{i-1}, a_i)$ — поддерживаем минимумы. Операции тривиальны

Очередь с минимумом на двух стеках Храним два стека с минимумом, один из которых мысленно наращиваем в одну сторону, а другой в другую, при этом очередь выглядит как бы как склеенные стеки. То есть мы добавляем элемент в первый стек, а извлекать хотим из второго.

$$X \to a_n, a_{n-1}, \ldots, a_1, \ | \ , b_1, b_2, \ldots, b_n \to Y$$

Тогда единственная сложная операция — если мы хотим извлечь минимум, а второй стек пустой. Тогда мы все элементы из первого перекинем во второй по очереди с помощью «извлеки-добавь»

Почему это работает за $O(1)$ на операцию амортизированно? Представим каждому элементу при рождении 2 монеты, одну из которых мы потратим на добавление в первый стек, а вторую на удаление через второй.

set для бедных Хотим не делать $erase$, только $insert$, $find$, get_min. Храним $\log n$ массивов, $|a_i| = 2^i$, каждый из которых по инварианту будет отсортирован. Тогда get_min рабтает за $O(\log n)$ — просто берем минимум по всем массивам. Аналогично $find$ делается бинпоисками за $O(\log^2 n)$

А как добавлять за $\tilde{O}(\log n)$? Каждый элемент при добавлении в структуру получает $\log n$ монет. Когда мы добавляем элемент, мы создаем новый массив ранга 0. Если было два массива ранга 0, сольем их в новый массив ранга 1 за суммарный размер (и заберем монетку у всех элементов во время слияния), и так далее, пока не создадим уникальный массив для текущего ранга.

Метод потенциалов $\Phi(S_i)$ — потенциал, который зависит только от состояния структуры (**не от** последовательности действий, которая к такому состоянию привела).

Опять вводим t_i, $\sum t_i = O(f(n))$. Определим амортизированное время работы:

$$\tilde{t_i} = t_i + (\Phi(S_{i+1}) - \Phi(S_i))$$

$$t_i = \tilde{t_i} + \Phi(S_i) - \Phi(S_{i+1})$$

Пусть мы показали $\tilde{t_i} \le f(n)$. Тогда

$$\sum t_i \le n \cdot f(n) + \Phi(0) - \Phi(n)$$

Нормальный потенциал — такой, что из неравенства выше все еще можно показать O-оценку на $\sum t_i = O(n \cdot f(n))$.

deque для богатых Хотим deque с поддержкой минимума.

Храним два стека как для обычной очереди. Все операции хорошо работают как на очереди, кроме перестройки структуры. В случае с очередью надо было переливать стеки только в одну сторону, а теперь иногда нужно туда-сюда.

Теперь мы будем перекидывать только половину элементов. Тогда нам понадобитсяя 3 стека, один из которых будет вспомогательным для перестройки (там иначе стеки развернуты).

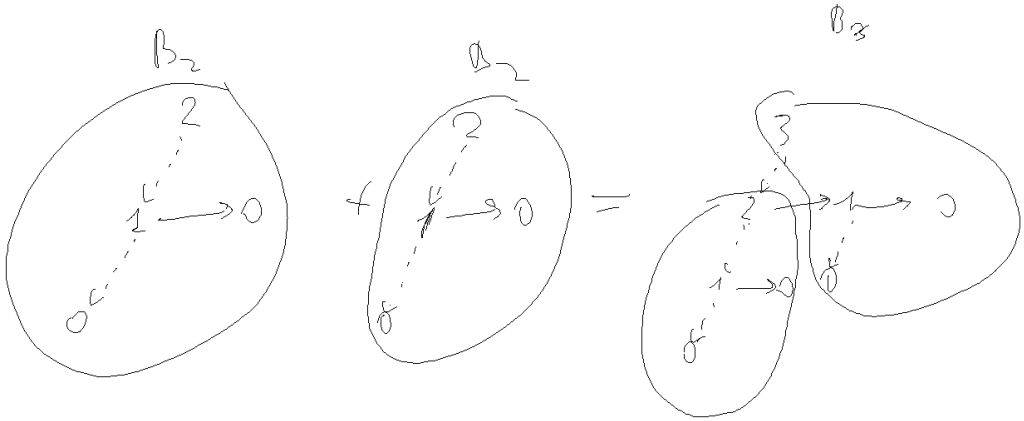
$\Phi(S_i) = |Size_1 - Size_2|$

Куча: insert, extract_min, decrease_min, decrease_key, increase_key, merge. Хотим добавить операцию merge - объединить 2 кучи. Считаем, что можем делать операции амортизировано (не разделяем кучи и кучи не персистентные). Меньшую кучу будем добавлять в большую ("переливать"в большую).

Хотелось бы раздать каждой вершине по $\log n$ монет и говорить, что, когда мы "переливаем"кучу, все элементы этой кучи платят по монете. Тогда монет хватит, так как при каждом переливании размер кучи, где находится вершина увеличивается вдвое, значит каждая вершина перельется не более $\log n$ раз. Но все ломается из-за того, что мы можем удалять вершины. Можно ввести потенциалы (подумать), а можно сказать, что у каждого элемента есть ранг (r(i)) и при каждом переливании все ранги меньшей кучи увеличиваются на 1. Тогда амортизировано merge будет работать за $O(\log^2 n)$.

Биномиальная куча:

Вершин в биномиальной кучи 2^n, на k-ом слое C_n^k вершин. Из каждой вершины храним ребро в старшего сына, в предка и в следующего брата. $B_k = \text{merge}(B_{k-1}, B_{k-1})$. Заметим, что merge деревьев одного ранга работает за O(1).



Если у нас есть n чисел, то как мы их поместим в кучу размера 2^k? $n = 2^{k_1} + 2^{k_2} \cdots + 2^{k_m}$. Тогда можем хранить все элементы как кучи рангов $k_1, \ldots k_m$. Тогда при merge нужно объединять два списка биномиальных куч (будем делать это 2 указателями по 2 массивам), будем объединять кучи одинаковых рангов

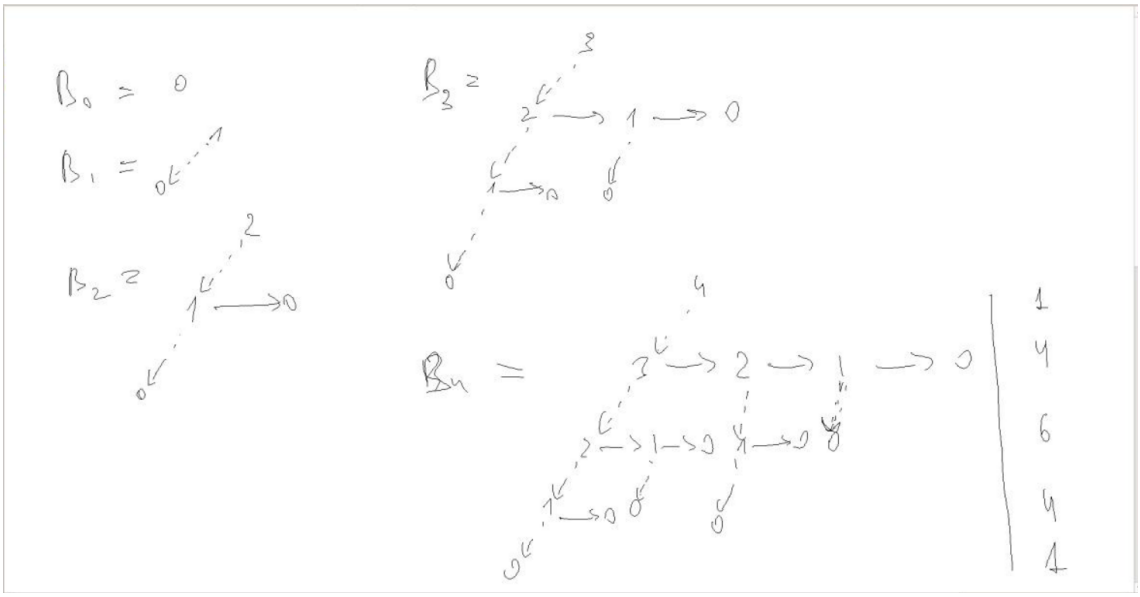
insert - создаем кучу ранга 0 с нашим элементом и делаем merge за $O(\log n)$.

Для поиска минимума будем просто поддерживать глобальный минимум, изменяя его за $O(\log n)$ (пробегаясь по всем кучам) при каждом запросе изменения.

decrease_key

extract_min

increase_key: decrease_key(v, $-\infty$) $\to$ extract_min $\to$ insert(x). Работает за $O(\log n)$.



Фибоначчиева куча: (чет я устал техать, чекайте у Кости)

Операции	binary heap	binomial heap	fibonacci heap
insert	$O(\log n)$	$O(\log n)$	$O(1)$
extract_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
decrease_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(1)$
increase_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
merge	$\tilde{O}(\log^2 n)$	$O(\log n)$	$O(1)$
get_min	$O(1)$	$O(1)$	$O(1)$

Больше куч!

	binaryheap	*binomialheap*	*fibonacciheap*
insert	$O(\log n)$	$O(\log n)$	$O(1)$
extract_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
decrease_key	$O(\log n)$	$O(\log n)$	$\tilde{O}(1)$
increase_key	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
merge	$\tilde{O}(\log^2 n)$	$O(\log n)$	$O(1)$
get_min	$O(\log n)$ *or* $O(1)$	$O(1)$	

Биномиальная куча

Храним биномиальные деревья. Каждому дереву сопоставим ранг. Ранг дерева полностью определяет его структуру. Дерево ранга 0 — одна вершина. Дерево ранга 1 — одно ребро. В общем случае, дерево ранга n содержит корень и полное двоичное дерево размера 2^{n-1}. Дерево ранга $n+1$ — это два слитых вместе дерева ранга n — корень второго указывает на корень первого, а корень первого теперь будет указывать на оба бинарных дерева.

Биномиальная куча — это набор из логарифма биномиальных куч.

Слияние двух деревьев мы научились делать за $O(1)$ — меньшая вершина по ключу становится новым корнем, а дальше перекидываем указатели.

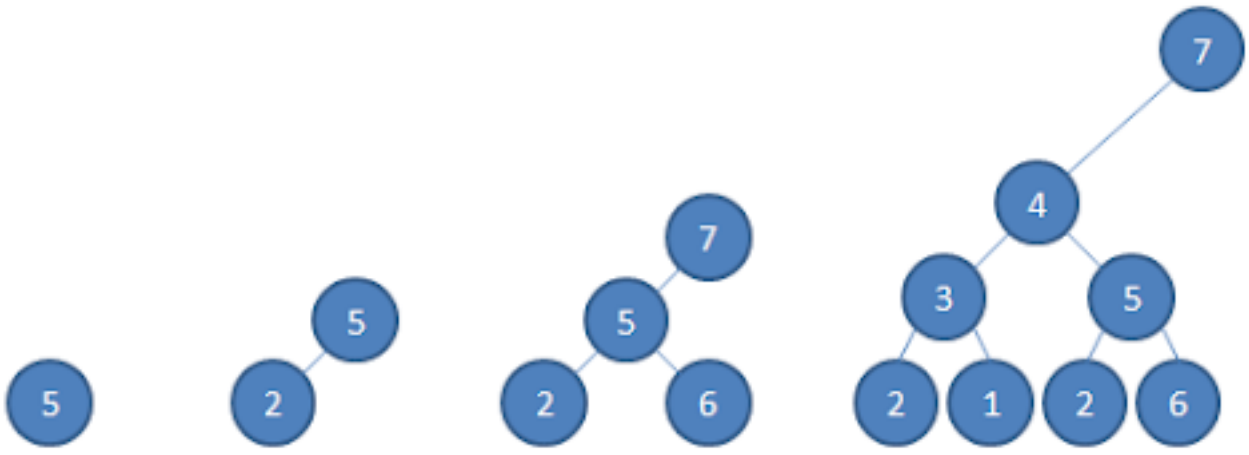
Слияние двух куч — это алгоритм сложения двоичных чисел — при сливании двух деревьев ранга n мы «переносим» прибавление кучи ранга $n+1$

Как добавлять элемент? Создать кучу на 0 элементов и слить их вместе.

Уменьшение ключа — напишем $sift\_up$ на нашей новой куче

Удаление минимума — Заметим, что если в правом дереве пройти по правым детям и обозначим их за корни, а их левых детей за полные бинарные деревья, то мы получим набор деревьев рангов $0, 1, \ldots, n-1$. Обозначим их за новую кучу, и сольем все вместе.

Увеличение ключа — удалим соответствующий элемент и добавим другой.



Фибоначчиева куча

Хотим сделать биномиальную кучу с послаблаблениями — делать операции в самый последний момент, менее четкую структуру, etc

Есть деревья, их корни храним в двусвязном закольцованном списке.

Всех детей для всех вершин храним в двусвязном закольцованном списке.

Новый ранг — это количество вершин в списке детей.

На каждом дереве выполнена куча, а также поддерживаем глобальный минимум.

Улучшение ключа делается так — удаляем вершину из своего списка, вместе с поддеревом. Добавляем в корневой список. Если мы удалили уже вторую вершину в поддереве родителя, то делаем каскадное вырезание — прыгаем по предкам с $mark = 1$, и вырезаем их в корневой список, причем все вырезания делаются по очереди.

Удаление делается так — мы приписываем всех детей к корневому списку, а потом вызываем *compact*, которая должна спасти наше дерево и навести порядок.

Псевдокод тупых операций:

```
struct Node{
    Node *child;
    Node *left;
    Node *right;
    Node *parent;
    int rank;
    bool mark;
    int value;
}

list<Node *> roots;

void insert(int x) {
    Node *node = new Node(x);
    roots.insert(node);
}

void merge(list<Node *> a, list<Node *> b) {
    merge(a, b); // O(1) haha super easy
}

int getmin() {
    return argmin->value;
}
```

compact

- Сбрасываем пометки корневого списка в 0
- Переводим дерево в состояние, где все ранги разлиичны
- Храним ранги, мерджим одинаковые
- Как мерджим? Берем меньший корень, и записываем в его детей второй корень

Обозначим $R = \max rank$, $t(H) = root\ list\ size$, $m(H) = \sum_v mark(v)$
compact работает за $O(R + t(H))$.

Анализ времени работы $extract\_min$ &$increase\_key$ — $\tilde{O}(R)$.
$\Phi(H) = t(H) + 2m(H) < 3n$
Пусть каскадное вырезание сделало t_i действий. $m: \ 1 \to 0, \ t: \ +1$. Тогда $\Phi'(H) = \Phi(H) - 1$ за каждое вырезание. Тогда амортизированно вырезание работает за $O(1)$.
Compact:
$t(H) \le R$, $t'(H) = t(H) - R$
Амортизациованно работает за $2R + 1$
Хотим показать $R = O(\log n)$.

$$\forall\ v \in H\ sz(v) \ge A^{rank(v)},\ A > 1$$

Тогда

$$r(v) \le \log_A s(v) \le c \log n$$

Возьмем

$$s(v) \ge \phi^{rank(v)-2}$$

<Оффтоп про числа Фибоначчи>

1. $F_n = F_{n-1} + F_{n-2},\ F_0 = F_1 = 1$
2. $F_n \ge \phi^{n-2}$
3. $\forall\ i \ge 2: \ F_i = \sum_{j=0}^{i-2} F_j + 1$

</Оффтоп про числа Фибоначчи>

Почему инвариант на размеры сохраняется? Возьмем вершину v с k детьми. Рассмотрим детей в том порядке, в котором их склеивал компакт. Тогда на момент добавления i-й вершины в структуру, ее ранг совпадал с рангом v. Тогда из условия на удаление не более чем одного сына ($mark$) следует, что $rank(i) \ge i - 1$. Тогда мы доказываем по индукции, что у нас все размеры — хотя бы числа фибоначчи, соответствующие рангу. Тогда $s(v) = \sum_{u \in g(v)} s(u) + 1 \ge \sum_{u \in g(v)} F_rank(u) + 1 \ge \sum_{i=0}^{rank(v)-2} F_i + 1 \ge F_{rank(v)-2}$

Хэширование Есть задача сравнения объектов:

$$U = \{objects\},\ u, v \in U :\ u \neq v?$$

Введем функцию $h :\ U \to \mathbb{Z}_m$, такую что $\forall\ u, v \in U :\ u \sim v \to h(v) = h(u)$. Обычно $m \ll |U|$.
Зачем юзать хэши, а не наивное сравнение?

- Бывает много сравнений, и мы не хотим дублировать вычисления
- Безопасность
- Для некоторых хешей верно, что $h(f(v, u)) = g(h(v), h(u))$. То есть можно вычислить хэш от некоего объекта, пользуясь уже посчитанными хэшами для других объектов.
- Иногда мы сравниваем объекты не на равенство, а на изоморфность.
- Сравнивать объекты бывает дорого

Требования к хэшу:

- вычислим за линейное время
- детерменирован
- семейство хэш-функций $\mathbb{H}$ (и можем взять сколько угодно оттуда)
- равномерность $\forall_{v \neq u} v,\ u :\ p_{h \in \mathbb{H}}(h(u) = h(v)) \simeq \frac{1}{m}$
- масштабируемость
- необратимость
- (*optional*) лавинный эффект (при маленьком изменении объекта хэш меняется сильно)

Важно, что мы хотим брать случайную функцию из семейства $\mathbb{H}$ на момент старта программы, потому что псевдослучайная функция на самом деле нам дает детерменированный алгоритм, который неверен.

Идеальная хэш-функция Так как объектов счетно, то отсортируем их, далее для каждого объекта запомним случайную величину от 1 до m (или даже можем запоминать ее лениво!).

Полиномиальный хэш Сводим объекты к строкам и хэшируем строки:

$$s = c_0 c_2 c_3 \dots c_{n-1},\ c_i > 0$$

$$h_b(s) = \sum_{i=0}^{n-1} c_i \cdot b^i\ (mod\ m)$$

$$p_b(h_b(s_1) = h_b(s_2)) \leq \frac{n}{m} \text{ для фиксированного m}$$

Доказательство: Рассмотрим функцию как многочлен, теперь для равных функций смотрим на то, равна ли разность многочленов нулю для какого-то b. У многочлена от b степень равна n, а вероятность попасть в конкретный модуль $\frac{1}{m}$.

$$h(s_1 + s_2) = h(s_1) + h(s_2) \cdot b^{|s_1|}$$

Хэш-таблица

Key-value storage: три типа операций — $set(x, y)$, $get(x)$, $has(x)$. Хэш-таблица умеет выполнять такие запросы за $O(1)$.

Хотим делать индексацию не по ключу, а по хэшу от ключа $x_0 \to h(x_0)$.

К сожалению, бывает так, что в одну и ту же ячейку попало много элементов (матожидание числа коллизий порядка $\frac{n^2}{m}$, где m — размер хэш-таблицы). Мы будем называть это коллизиями.

Коллизии можно решать двумя способами, соответствующие хэш-таблицы имеют **открытую** или **закрытую** адресацию.

Закрытая адресация (или решение коллизии методом цепочек) — в каждой ячейке храним список, в который будем добавлять соответствующие элементы. Матожидание длины списка будет порядка $\frac{n}{m}$.

Хэш-таблица работает линейно от числа элементов, которые в ней когда-либо были, поэтому динамическая хэш-таблица работает за амортизированное время.

«stop-the-world»-концепция — если внутренние параметры системы бьют тревогу, сделаем глобальное изменение. В случае хэш-таблицы, если в таблице сейчас m элементов, создадим новую хэш-таблицу удвоенного размера, в которую перехэшируем оставшиеся элементы.

Замечание. Очень часто разработчик не хочет амортизированное время работы, потому что боится внезапного «stop-the-world», потому что это выключает систему на длительное время. Пример — финал TI8.

Открытая адресация

Делаем вид, будто коллизий не бывает (то есть, в каждой ячейке храним только одно значение). Кроме того, рядом с ячейкой храним ключ элемента (или -1, если там пусто). Тогда если мы хотим найти элемент x, мы смотрим в ячейку $h(x)$, и идем от нее вправо до тех пор, пока не встретим x или -1.

Ожидаемое время — это $\frac{1}{1-\alpha}$, где $\alpha = \frac{n}{m}$. На практике все считают, что время — константа.

Удаление с открытой адресацией — нетривиальная задача, потому что удаление элемента рушит цепочки.

Удаление без «stop-the-world» — удаляем все элементы от нашего элемента до ближайшей -1, но потом вернем их обратно с помощью добавлений

Удаление с «stop-the-world» — кроме пометки -1 делаем пометку «зарезервирована». Тогда при удалении элемента мы ставим в его ячейку пометку «зарезервирована». Тогда при линейном проходе мы делаем вид, что резерв — это настоящий элемент, а при добавлении мы можем записать в резерв. Резерв включается в параметр α, поэтому нам понадобится делать перестройки.

Кроме линейного сканирования можно делать что-то типа «прыжкам по хешу» $(h(x) + æh'(x))$, но на практике линейное сканирование — наш бро.

Совершенное хэширование

Нам изначально дано сколько-то ключей, мы хотим делать $get(x)$, $set(x, y)$ за гарантированные $O(1)$ с ожидаемым $O(n)$ на преподсчет.

Сделаем хэш-таблицу с закрытой адресацией. В каждой ячейке ожидаемое $O(1)$ элементов. Сделаем новую хэш-функцию, которая переносит все элементы из l_i в ячейки размера l_i^2. Вероятность коллизии при таком хэшировании меньше $\frac{1}{2}$, поэтому мы будем просто рандомить хэш-функцию, пока не получим отсутствие коллизий, что займет у нас ожидаемое $O(1)$.

Фильтр Блума

insert, *get*

Запросы *get* работают необычным образом — No $\to$ No; Yes $\to$ Yes$(1-p)$/No(p). То есть, если структура говорит, что элемент в ней лежит, то он в ней точно не лежит. Дальше минимизируется p.

Фильтр Блума является массивом из битов длины m. Фильтр выбирает k хэш-функций, элементу сопоставляется k значений хэша.

insert — в каждое из k значений ставим 1

get — проверяем, что во всех k значениях стоит 1.