

1. (a) Истинное время работы `increment` в худшем случае составит k операций
 Это значение достигается когда массив a заполнен единицами.
- (b) Посмотрим сколько раз мы инвертировали бит с номером i .
 Нулевой бит инвертируется каждую операцию, первый бит - каждую вторую. Нетрудно заметить что i бит меняет значение только после 2^i операций `increment`. Итого получаем $n + \frac{n}{2} + \frac{n}{4} + \dots \leq 2n$
- (c) Рассмотрим массив в котором на $k - 1$ месте стоит единица, а остальные значения - нули.
 Будем чередуя выполнять операции `decrement` и `increment`.
 Нетрудно заметить что каждая будет выполняться за k простых операций.
 Следовательно время работы в худшем случае — $\mathcal{O}(nk)$.
- (d) Заведём дополнительную переменную `right`, изначально равную -1 , — самая правая единичка. После неё в массиве может храниться что угодно, но мы будем считать что там лежат нули.
 - i. Когда цикл в `increment` дойдет до `right + 1` мы должны остановиться так как помним о том что все элементы после `right` равны нулю. Ещё нужно подвинуть `right` в случае когда $i > \text{right}$, потому что самая правая единичка может подвинуться правее.
 - ii. `get(i)` нужно изменить так что-бы при $i > \text{right}$ она возвращала 0 .
 - iii. Теперь `setZero` будет просто ствить `right` в значение -1

```

1  increment():
2      i = 0
3      while i < k and i <= right and a[i] = 1:
4          a[i] = 0
5          i++
6      right = max(right, i)
7      if i < k:
8          a[i] = 1
9
10 get(i):
11     if i > r:
12         return 0;
13     return a[i]
14
15 setZero():
16     r = -1
17

```

2. Придумаем функцию потенциала:

n — количество элементов в векторе

c — фактический размер вектора

$$\frac{c}{4} \leq n \leq c$$

$$\Phi(n, c) = \begin{cases} c \leq 2n, 2n - c, \\ 2n < c, 2c - n. \end{cases}$$

$$\text{push} = \begin{cases} c \leq 2n, 1 + \Phi(n+1, c) - \Phi(n, c) = \mathcal{O}(1), \\ 2n < c, 1 + \Phi(n+1, c) - \Phi(n, c) = \mathcal{O}(1), \\ n = c, n + \Phi(n+1, 2c) - \Phi(n, c) = \\ = n + 2n + 2 - 2c - 2n + c = 3 + n - c = \mathcal{O}(1). \end{cases}$$

$$\text{pop} = \begin{cases} c \leq 2n, 1 + \Phi(n+1, c) - \Phi(n, c) = \mathcal{O}(1), \\ 2n < c, 1 + \Phi(n+1, c) - \Phi(n, c) = \mathcal{O}(1), \\ n = \frac{c}{4}, c + \Phi(n-1, \frac{c}{2}) - \Phi(n, c) = \\ = c + c - n + 1 - 2c + n = 1 = \mathcal{O}(1). \end{cases} \Rightarrow$$

\Rightarrow все операции выполняются в среднем за $\mathcal{O}(1)$

3. Пусть у нас есть неинициализированный массив a . Заведём дополнительно массив b на n элементов и стек на массиве s . s будет соответствовать элементам, которым мы уже что-нибудь присвоили. В ячейке стека будем хранить позицию в массиве, куда мы присваивали.

В a будем хранить последнее значение, либо мусор. Во b храним позицию в стеке нашего элемента, либо опять мусор.

Как это работает?

При запросе значения в ячейке i , будем смотреть на значение в b_i . Если $b_i \geq \text{size}(s)$, то очевидно в эту ячейку мы ничего не присваивали и нужно вернуть из $\text{get}(i)$ ноль.

Посмотрим на случай, когда $b_i < \text{size}(s)$.

Мы точно знаем что внутри стека нет мусора и каждый элемент соответствует инициализированному значению.

Это значит что $s_{b_i} = i \iff i$ — была проинициализирована ранее.

Осталось поддерживать эту структуру при добавлении элемента, это просто.

В $\text{set}(i, x)$ есть два случая.

a_i — проинициализирована, тогда нужно просто записать в a новое значение x .

a_i — не проинициализирована, тогда нужно записать в b_i значение $\text{size}(s)$. Затем добавить в s вершину со значением i и присвоить элементу a_i значение x .

Вот код для понимания.

```
1  a, b, s = [], [], []
2  sSize = 0
3  init(n):
4      a, b, s = [n], [n], [n]
5
6  isInit(i):
7      if b[i] < sSize and s[b[i]] == i:
8          return True
9      else:
10         return False
11
12  get(i):
13      if isInit(i):
14          return a[i]
15      else:
16          return 0
17
18  set(i, x):
19      if isInit(i):
20          a[i] = x
21      else:
22          a[i] = x
23          s[sSize] = i
24          b[i] = sSize
25          sSize++;
26
```

4. Будем составлять группы по k различных элементов. Если мы разобьем элементы на группы по k попарно не равных друг другу, то оставшиеся элементы - могут быть нашим ответом. Заведём массивы a и b на k элементов. a - массив значений кандидатов, b - количество таких элементов.

Будем идти по массиву и смотреть на элементы. Если очередной элемент - один из кандидатов, то увеличим счетчик этого элемента в a на 1, если не равен, то запишем его как нового кандидата. Если у нас собралась группа из k различных элементов, то уменьшим всем элементам счетчики на 1 (у нас собралась группа из k различных). Отсюда видно что мы всегда сможем добавить элемент в наш массив кандидатов (хотя бы 1 счетчик равен нулю, а значит этого элемента нет и можно вместо него записать текущий).

В конце у нас останется несколько элементов с ненулевыми счетчиками. Из них ответами будут те, у которых $b_i > \frac{n}{k}$.

5. Будем внутри нашей памяти хранить односвязный список не занятых блоков. Заведём указатель `head` на последний элемент списка. В вершине списка будем хранить одно число — указатель на предыдущий элемент, или -1 , если элемент является последним. Пусть для нас память представляет собой массив a длины n . Проинициализируем её так:

$$\forall i \in \{0, 1, \dots, n-1\}, a_i = i-1$$

$$\text{head} = n-1$$

Теперь очевидно. Когда нужно вернуть номер свободной ячейки,

мы возвращаем номер head и удаляем последний элемент списка. Когда нам говорят сделать free некоторой ячейки, мы добавляем эту вершинку в наш список неиспользованных.

```

1  head = -1
2  mem = []
3  init(n):
4      mem = [n]
5      for i in range(0, n):
6          mem[i] = i - 1
7      head = n - 1
8
9  malloc():
10     tmp = head
11     head = mem[head]
12     return tmp
13
14  free(p):
15     mem[p] = head
16     head = p

```

6. Найдем и докажем время работы push

Пусть $\{a_1, a_2, \dots, a_k\} = A$

Введём функцию потенциала:

$\Phi(A) = \sum_{i=0}^k (k-i)2^i$ Тогда рассмотрим что происходит при добавлении одного элемента в структуру.

Пусть в структуре первые k' массивов не пусты.

Тогда элемент, который мы вставляем в структуру, k' раз поучаствует в merge \Rightarrow на это мы потратим k' операций.

push работает за:

$$(1) k' + \Phi(A') - \Phi(A) = k' + (k - k' - 1)2^{k'+1} - \sum_{i=0}^{k'} (k-i)2^i$$

Рассмотрим сумму поближе:

$$(2) \sum_{i=0}^{k'} (k-i)2^i = k2^{k'+1} - \sum_{i=0}^{k'} i2^i$$

Ещё ближе, теперь уже на сумму $\sum_{i=0}^k i2^i$.

Докажем по индукции что она равна $2(k2^k - 2^k + 1)$

$$2(k2^k - 2^k + 1) + (k+1)2^{k+1} = k2^{k+1} - 2^{k+1} + 2 + 2^{k+1} + k2^{k+1} = k2^{k+2} + 2^{k+2} - 2^{k+1} + 2 = 2((k+1)2^{k+1} - 2^{k+1} + 1)$$

Доказали, круто, вернёмся слегка назад к (2).

$$k2^{k'+1} - \sum_{i=0}^{k'} i2^i = k2^{k'+1} - k'2^{k'+1} - 2^{k'+1} + 2 = 2^{k'+1}(k - k' - 1) + 2$$

Окэй, посчитали (2), тогда можем почитать и (1).

$$k' + 2^{k'+1}(k - k' - 1) - 2^{k'+1}(k - k' - 1) + 2 = \mathcal{O}(k)$$

Значит push работает в среднем за $\mathcal{O}(k)$.

C contains всё проще, потому что структуру данных эта функция не изменяет (разность потенциалов равна нулю). Значит просто считаем худший случай.

На i -й итерации запускается бинарный поиск на массиве размера 2^i .

contains работает за:

$$\log_2 2^0 + \log_2 2^1 + \log_2 2^2 + \dots + \log_2 2^k = \frac{k(k+1)}{2} = \mathcal{O}(k^2)$$