Глава I

Лекция от 9 сентября

I.1. Структуры данных

1. Online.
   На поступающий запрос сразу даётся ответ. Говоря о времени работы, подразумеваем время, затрачиваемое на обработку одного запроса.

2. Offline.
   Запросы поступают «пачками», ответы даются сразу на все запросы. Оценивается время, затрачиваемое на обработку всех запросов.

Существует другая классификация:

1. Статические (Static Data).
   Можем отвечать на запросы вида «Присутствует ли элемент x в множестве A?», но не можем изменять само множество A.

2. Динамические (Dynamic Data).
   Можем отвечать на запросы «Присутствует ли элемент x в множестве A?» и, в отличие от Static Data, можем добавлять и удалять элементы A.

Также важным разделением является:

1. Амортизированное время — среднее за все запросы.

$$O(n) \text{ за } n \text{ запросов } = O_{\text{аморт}}(1) \text{ на запрос}$$

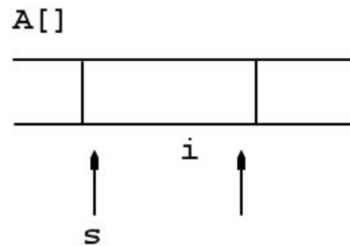
2. Real Time — максимум из времён каждого запроса. Важнее для больших сервисов (социальные сети, сайты...).

I.2. Стек, очередь, дек

𝔇𝔢𝔣: Интерфейс — заданная функциональность, но не заданная реализация.

Интерфейсы стека, очереди и дека мы знаем. Дек (Deque) является очередью (Queue) и стеком (Stack).

Реализуем дек на массиве. Дек — отрезок на этом массиве, а также указатели на начало и конец отрезка. Поддерживаются операции добавления и удаления элементов в начало и в конец дека. Дополнительно можно реализовать операцию доступа к i-ому элементу (это элемент $head + i$ в массиве). Время выполнения каждой из операций $O(1)$. (Стек с доступом к любому элементу за $O(1)$ называется вектор).

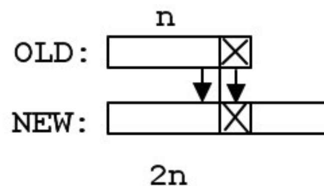


Реализация: у нас есть отрезок выделенной памяти и указатели на его начало и конец. В таком случае несложно превратить дек в дек с индексацией за $O(1)$ $(A[s+i])$.

Однако память у нас конечная, поэтому в некоторый момент нам может потребоваться больше, чем мы имеем, и тогда нам придётся перевыделить память и скопировать имеющиеся элементы в неё. В случае дека (пусть в данный момент его длина равна n), разумно будет выделить память размером $3n$ (имеющиеся ячейки, n ячеек в начало и n ячеек в конец). В этом случае нам удастся добиться добавления нового элемента за амортизированное $O(1)$, так как $\frac{O(n)}{n} = O(1)$, где $O(n)$ — время, затрачиваемое на выделение, и n — количество шагов доследующего перевыделения.

Теперь мы хотим научиться добавлять новый элемент не за амортизированное, а за реальное $O(1)$. Посмотрим, как это можно сделать, на примере вектора.

Итак, предположим, что мы исчерпали свободную память, и к нам пришёл новый элемент на добавление. Мы, как и в прошлый раз, перевыделяем память (увеличим её размер вдвое), однако копировать в неё все старые элементы сразу не будем, а поступим следующим образом: каждый раз, когда к нам будет приходить элемент на добавление, будем добавлять в стек его и один элемент из «старого» стека.



Таким образом, к моменту, когда новая память заполнится, в неё уже будут скопированы все элементы из старой.

Как выделять и освобождать память?

```
int *a;
a = new int[size]; // добавление
delete[] a; // освобождение НЕ ЗАБЫВАТЬ КВАДРАТНЫЕ СКОБКИ!!!
```

Итого, наш вектор перевыделяет память так:

```
n *= 2
delete[] Old;
Old = New;
New = new int[n];
```

Подобную идею можно использовать не только в векторе.

Можно реализовать дек, стек и очередь двусвязным (или односвязным для стека и очереди) списком. Этот способ проще описанного выше, однако теряется возможность обращения к элементу по индексу.

У стека последний элемент лежит в начале списка и указывает на более глубокий. У очереди добавление идёт в конец списка, нужно также хранить указатель на конец списка. В языке C++ `std::deque` — дек с индексацией, а `std::queue` и `std::stack` — без индексации.

I.3. Хеш-множество

Хеш-множество (HashSet): Мы хотим иметь множество, уметь добавлять и удалять элементы, а также быстро искать их.

Научимся делать хеш-множество для маленьких чисел. Это можно сделать массивом `bool`, размер которого будет равен максимальному из хранимых в нём элементов.

REM: Из C++: быстрый массив из N бит есть `std::bitset<N>`. Используется эквивалентно массиву, плюс умеет делать побитовые операции (`&, |, ^`). Устроен как массив целых чисел, только памяти использует меньше.

Вспомним, что есть просто список (List): он умеет то же самое, но всё, кроме добавления, работает $O(n)$.

Однако N может принимать достаточно большие значения($N \leqslant 10^{18}$). В таком случае мы хотим хранить наши числа в массиве (массиве списков, так как возможны коллизии) длины M, где $M \leqslant 10^6$. Каждый элемент $x \in [0, N)$ будет лежать в списке, хранящемся в ячейке $i \in [0, M)$, вычисляемой по некоторому «хорошему» правилу. Если i будут распределены равномерно, то примерное количество чисел в одной ячейке составит $\frac{n}{M}$ штук, где n - количество элементов x.

Какое «хорошее» правило можно взять? Утверждается, что следующее правило окажется вполне подходящим:

$$\begin{cases} i = x \bmod M \\ M \text{ — простое} \end{cases}$$

Для того, чтобы время поиска элемента множества оставалось $O(1)$, будем постоянно поддерживать выполнение условия $\frac{n}{M} \leqslant C$, и если в какой-то момент количество элементов n станет слишком большим и условие перестанет выполняться, то изменим число M примерно в 2 раза(при этом M должно остаться простым).

Как же найти новый простой модуль? Идём вперёд и увеличиваем, пока не простое. Время работы данного алгоритма составит $O(\sqrt{n} logn)$,

так как среднее расстояние между простыми числами равно логарифму.

В C++ и Java стандарнтые хешмножества удваиваются долго, за линию.

I.4. Хеш-таблица

Раньше мы хранили x — элементы множества. Теперь будем хранить пары (x, y), где x — ключ, а y — значение, которое мы сможем по этому ключу находить. Получилась хеш-таблица (HashTable). Как её сделать:

```
bitset<N> u; //присутствует ли ключ x в множестве
int y[N];    //значение по ключу, если есть
```

Можно обойтись только вторым массивом и фиктивными элементами.

I.5. Частичные суммы

```
sum[0] = 0;
for (int i(0); i < n; ++i)
    sum[i + 1] = sum[i] + a[i];
```

$$\sum_{i=L}^{R-1} a_i = Sum[R] - Sum[L]$$

I.6. Бинарный поиск

Пусть есть отсортированный массив a. Научимся быстро понимать каждую из трёх следующих вещей: присутствует ли элемент в массиве, на какой позиции в массиве находится данный элемент(поиск), на какой позиции в массиве находится первый элемент, больший либо равный заданному. Для защиты можно в конец добавить фиктивный элемент, гарантированно больший всех. Каждая следующая задача решает все предыдущие; более того, код будет практически таким же. При этом первые два варианта реализуются хеш-таблицей ($Map[A[x]] = x \Leftrightarrow$ x присутствует).

Для первых двух задач:

```
l = 0;
r = n - 1;
while (l <= r)
{
    m = (l + r) / 2;
    if (a[m] == x)
        return 1 // m;
    if (a[m] > x)
        r = m - 1;
    else
        l = m + 1;
}
```

Для последней:

```
l = 0;
r = n - 1;
while (l <= r)
{
    m = (l + r) / 2;
    if (a[m] > x)
        r = m - 1;
    else
        l = m + 1;
}
return l;
```

I.7. Задачи

1. Количество различных чисел [static]

    - HashSet. Запихнём все, пробежимся. $O(n)$.
    - Sort. Посортируем, ищем количество различных соседей и прибавить единицу. $O(n \log n)$.

2. Количество различных чисел [dynamic, online]

- HashMap. Для каждого числа храним его количество $count$. Также храним количество разных чисел $result$. Если нам поступил запрос Add и $count[x] = 0$, то увеличим $result$. Del реализуем зеркально. $O(1)$.

3. Stack, Deque: [online] с удвоением, [offline] заранее выделяем, сколько надо.

4. Выделение памяти (Allocation). Нужны функции:

   - `void* new(int size)`
   - `void delete(void *what)`

   Есть полезные подслучаи:

   - $n = const$ — стек свободных ячеек. Возьмём себе место под стек, а оставшееся место режем по n байт. Стек содержит указатели на все эти ячейки. Когда у нас просят new, возвращаем значение на вершине стека, $delete$ — кладём указатель на освободившийся участок обратно в стек.
     В современной реальности есть следующие размеры кешей: $4KB < 4MB < 4GB < 4TB$ (кеши процессора, оперативка, жёсткая память). При понижении уровня скорость падает в 10–20 раз.
     Если списком: храним список из кусков $n + размерУказателя$ байт. Теперь всё шустрее с точки зрения кеширования, т.к. все операции происходят только в начале памяти.
   - Можно хранить указатель на начало свободной памяти, а при выделении памяти сдвигать его (на столько, сколько памяти у нас попросили). В таком случае можно запоминать, на сколько мы сдвинулись в последний раз, и тогда мы сможем освобождать последний выделенный участок памяти.