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.

Время работы. $k \sim \frac{m}{n}$. Рассмотрим вероятность ложноположительного срабатывания. Вероятность того, что клетка свободна — $(\frac{m-1}{m})^{kn}$. Тогда вероятность ложноположительного срабатывания это $(1 - (\frac{m-1}{m})^{kn})^k \sim (1 - e^{-\frac{kn}{m}})^k$.

Путем долгих вычислений получаем $k = \ln 2 \cdot \frac{m}{n}$.

Кроме того, ФБ поддерживает операции пересечения и объединения множеств.