Михаил Густокашин. Хеширование.

Допустим, перед нами стоит следующая задача: нам дается множество ключей (уникальных значений) и требуется уметь быстро проверять, входит ли ключ в наше множество. При этом множество ключей может изменяться, т.е. ключи могут добавляться и исключаться из множества.

Сейчас мы будем рассматривать все на числовых примерах. Допустим, нам дан набор целых чисел от 1 до 10000, а затем идет серия запросов вида «есть ли число X в множестве?». Мы можем создать булевский массив, в котором будем помечать, встречалось данное число или нет. При этом сложность одного запроса будет $O(1)$.

Если же чисел больше, то мы можем попробовать использовать для хранения признака наличия числа во множестве не 1 байт, а 1 бит, пользуясь битовыми функциями. Это даст нам возможность увеличить максимальный размер числа в 8 раз. Но и этого может не хватить. Например, если числа изменяются от 0 до 2^{31}, то такую таблицу невозможно разместить в памяти, которая дается нашему решению.

Эта задача имеет решение в некоторых частных случаях. Например, пусть максимальный размер множества равен 1000, а каждый элемент может быть в пределах от 0 до 2^{31}. Если мы будем создавать таблицу размером 2^{31} элементов, то будет использована меньше ее одной миллионной части.

Будем решать такой класс задач (когда количество чисел намного меньше максимального значения) с помощью так называемых хеш-таблиц.

Введем понятие хеш-функции, как функции, отображающей множество ключей (в нашем случае чисел от 0 до 2^{31}) в меньшее множество ключей (соизмеримое с максимальным количеством элементов — в нашем случае с 1000). Хеш-функция должна обладать двумя основными свойствами: быть быстрой и равномерно генерировать ключи (т.е. чтобы одному и тому же ключу в малом множестве соответствовало примерно равное количество ключей в большом множестве). Иногда от хеш-функции требуют неустойчивости (т.е. чтобы при близких значениях ключей большого множества она генерировала сильно отличающиеся ключи малого множества).

Размер таблицы (вообще говоря, одномерного массива) для эффективной работы должен быть больше, чем удвоенное количество элементов малого множества. Обозначим размер таблицы за N.

Будем использовать в качестве хеш-функции операцию взятия остатка от деления числа большого множества на `N` (`X % N`). Это достаточно хорошая функция: считается относительно быстро и распределена равномерно. Итак, мы считаем остаток от деления нашего числа `X` на `N` и записываем `X` в ячейку с индексом `X % N`. Затем, при проверке числа `Y`, мы просто смотрим на ячейку `Y % N` и, если число `Y` находится там, то возвращаем признак наличия. Сложность опять получается $O(1)$.

Однако возникает проблема — несколько чисел могут иметь одинаковый остаток от деления на N. Такая ситуация называется «коллизией». Существует несколько способов разрешения коллизий, мы рассмотрим способ со списками, каждый из которых соответствует одному значению остатка (одной ячейке хеш-таблицы).

Допустим, размер нашей хеш-таблицы равен 8 и в нее были внесены элементы $6, 19, 27, 11, 16, 22$. Тогда она будет выглядеть как на рисунке.

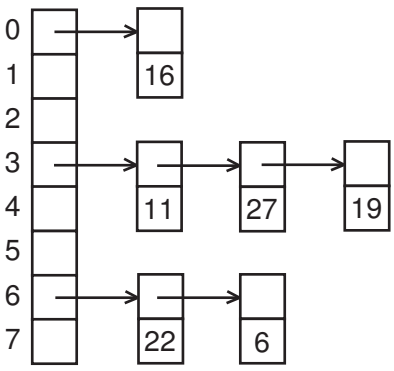

Рис. 1: Пример хеш-таблицы

Использование списков в данном случае уместно, т.к. количество элементов в каждом списке будет небольшим. Можно использовать и динамически расширяемые массивы, если значений в хеш-таблице достаточно много. Теперь для добавления элемента нам надо подсчитать его хеш-функцию и поместить в соответствующий этому значению список. Для проверки принадлежности элемента множеству нам также надо посчитать его хеш-функцию и попытаться найти его в нашем списке.

В среднем такая хеш-таблица будет работать за $O(1)$, т.к. коллизии будут встречаться редко. Ее можно «завалить» по времени только в случае, если точно знать размер таблицы, а при использовании набора тестов для проверки задачи это невозможно. Поэтому следует выбирать произвольный размер, больший $2 \times N$, где N — максимальное количество элементов, одновременно находящихся в хеш-таблице.

Функции работы с хеш-таблицей можно реализовать так:

```
typedef struct {
  struct -node *next;
  int data;
} node;

node **hashTable;
int hashTableSize;

int hash(int data) {
  return (data % hashTableSize);
}

node *insertnode(int data) {
```

```
  node *p, *p0;
  int bucket;
  bucket = hash(data);
  p = (node*) malloc(sizeof(node));
  p0 = hashTable[bucket];
  hashTable[bucket] = p;
  p->next = p0;
  p->data = data;
  return p;
}

void deletenode(int data) {
  node *p0, *p;
  int bucket;
  p0 = 0;
  bucket = hash(data);
  p = hashTable[bucket];
  while (p && (p->data != data)) {
    p0 = p;
    p = p->next;
  }
  if (!p) return;
  if (p0) p0->next = p->next;
  else hashTable[bucket] = p->next;
  free (p);
}

node *findnode (int data) {
  node *p;
  p = hashTable[hash(data)];
  while (p && (p->data != data)) p = p->next;
  return p;
}
```

Перед работой необходимо создать хеш-таблицу в динамической памяти и установить ее размер. Например:

```
hashTableSize = 20001;
hashTable = (node**) malloc(hashTableSize*sizeof(node*));
```

Кроме того, хеш-функции могут использоваться и в других ситуациях, например при сравнении сложных объектов. Мы заранее считаем хеш-функцию от каждого объекта, а затем, при сравнении, в первую очередь сравниваем значения хеш-функция и проводим сравнение для объектов полностью только в случае совпадения хешей.

Например, простейший способ подсчитать хеш-функцию от строки — сложить коды всех символов, входящих в строку. Такая хеш-функция будет генерировать одинаковые значения для строк, которые содержат одинаковые буквы, независимо от порядка.

Существуют и другие методы подсчета хеш-функции от строки, например подсчет полинома по какому-либо модулю.

Хеш-функция от строки обычно должна хорошо пересчитываться при удалении первого символа строки и добавлении нового символа — это позволяет использовать такое хеширование при поиске подстроки в строке (метод Рабина-Карпа).

В принципе, хеш-функция может быть введена для любых объектов.