Алгоритмы и Структуры Данных. Лекция 7

03.04.2024

imkochelorov

Хэш-таблицы

Что нам нужно от хэш-таблиц?:

1. Структура данных `set` — множество уникальных элементов
   - `insert(x)` — добавить элемент в множество, если ранее в нём его не было
   - `remove(x)` — удалить элемент из множества, если он в нём присутствует
   - `contains(x)` — проверить, есть ли элемент в множестве
2. Структура данных `map` — отображение $k \to v$ по уникальному ключу k
   - `put(k, v)` — присвоить ключ соответствующее значение
   - `remove(k)` — удалить ключ
   - `get(k)` — получить значение, соответствующее ключу

Пусть ключ (значение, в случае `set`) `x` $\in$ `[0, u-1]`

Различные подходы к написанию:

1. `u` — маленькое

Например, хотим хранить ключи до 5. Создадим массив из 5 элементов:

```
a = [null] * 5
```

```
put(k, v):
  a[k] = v
```

```
get(k):
  return a[k]
```

```
remove(k):
  a[k] = null
```

Если известно, что все ключи не превосходят, например, миллион, то это очень хорошая реализация, каждая операция в которой работает за честную $O(1)$
Но, к сожалению, в жизни всё часто устроено иначе: не всегда u маленькая, а иногда хочется хранить элементы сложнее чисел, например, строки, которые уже непонятно как индексировать

2. `u` — большое

Предположим, что мы не можем позволить себе создать массив на u элементов. Например, `u` $= 10^{18}$
Всё ещё хочется хранить все наши элементы в массиве, но необходимо *пожать* наши числа, чтобы мы могли индексироваться по не очень большому массиву и хранить их

Создадим массив на m элементов
Придумаем функцию `h(x): [0, u-1]` $\to$ `[0, m-1]` — *называется хэш-функцией*
Первое что приходит в голову для функции `h(x) = x % m`

```
m #some const
a = [null] * m
```

```
h(x):
  return x % m
```

```
put(k, v):
  a[h(k)] = v
```

```
get(k):
  return a[h(k)]
```

```
remove(k):
  a[h(k)] = null
```

К сожалению, у этой реализации есть проблемы:

Пример:

```
m = 5
a = [null] * 5
h(x):
  return x % 5
put(12, 8) #a[2] = 8
put(17, 5) #перезаписываем a[2], затирая старое значение
get(12) #получим 5, хотя должны были получить 8
```

Коллизия:

$$\texttt{x} \neq \texttt{y} \quad \text{и} \quad \texttt{h(x) = h(y)}$$

Довольно мерзкая ситуация, с которой нужно как-то бороться. Но по принципу Дирихле следует, что полностью побороться с коллизиями не получится и необходимо как-то научиться жить с этим

3. `u` — большое, но теперь уживаемся с коллизиями (или почему хэш-*таблица* называется таблицей)

Пример:

```
m = 5
a = [null] * m

put(12, 8)
#a[2] = [(12, 8)]
put(17, 5)
#a[2] = [(12, 8),
#        (17, 5)]
get(12) #8
```

Массив `a` после инициализации состоит из `m` `null` значений
После добавления по ключу `12` значения `8`, `a[2] = 8`
Перед добавлением значения по ключу `17`, видим, что в `a[2]` уже лежит значение `5`
Исправим алгоритм и вместо хранения одного элемента в ячейке `a[2]`, будем хранить в ней список - *цепочку* ключ-значение `[(12, 8), (17, 5)]`
Выполняя `get(12)` получим не одно значение, а *цепочку* вида ключ-значение, пробежавшись по которому найдём искомое по ключу значение

Реализация:

```
put(k, v):
  h = h(k)
  for i in range(len(a[h])):
    if (a[i][0] == k):
      a[i][1] = y
  a.add((x, y))
```

```
get(k):
  for (x, y) in a[h(k)]:
    if (x == k):
      return y
  return null
```

```
remove(k):
  for i in range(len(a[h])):
    if (a[i][0] == k):
      a.remove(i)
```

Однако такая реализация будет плохо работать с серией значений `0, m, 2 * m, 3 * m, ...`
Теперь ассимптотика стала $O(n)$, что ничем не лучше обыкновенного массива

Добавим случайность в работу программы

`h(x)` — случайная функция: `[0, u-1]` $\to$ `[0, m-1]`. Существует m^u таких функций

Предположим, мы выбрали одну из них. Посчитаем теперь время математическое ожидание времени работы `get()`. Оно будет равно математическому ожиданию количества элементов во входных данных с одинаковым хэшом:

$$E(T(\texttt{get()})) = E(\text{количества элементов } \texttt{x: h(x) = h(k)})$$

Важный момент: случайность берётся только из выбора хэш-функции. Мы не считыаем, что входные данные случайны

Посчитаем вероятность того, что неравные элементы имеют одинаковый хэш (вероятность коллизий):

$$p(\texttt{h(x) = h(k)}) = \frac{1}{m}$$

По линейности математического ожидания посчитаем:

$$E(T(\texttt{get()})) = E(\text{количества элементов } \texttt{x: h(x) = h(k)}) = n \cdot p(\texttt{h(x) = h(k)}) = \frac{n}{m}$$

Теперь, если мы будем выбирать случайную хэш-функцию в начале работы программы, то вне зависимости от входных данных математическое ожидание времени работы `get()` будет равно $\frac{n}{m}$.
То есть, все n входных значений в среднем разбиваются поровну между m нашими ячейками

$$m \sim n: \quad E(T(\texttt{get()})) = O(1)$$

Отличная структура данных, которая хорошо работает, однако есть один нюанс. Всё это время мы опирались на то, что можем выбрать сдучайную хэш-функцию. Но на самом деле не очень понятно, как мы можем это сделать. Даже непонятно, как сохранить информацию об этой функции, когда она уже будет создана. Так как возможных функций m^k, число бит, необходимое, чтобы сохранить информацию об одной конкретной слишком велико:

$$\textit{необходимое число бит} = \log_2(m^u) = u \cdot \log_2(m)$$

Сохранить информацию о нашей функции уже проблема, не говоря о том, чтобы как-то её выбрать

Попробуем выбирать функцию не всех возможных существующих, а из множества поменьше.
Нам необходимо иметь возможность случайно её выбирать и хранить, но оставляя вероятность коллизии хэшей двух различных элементов неизменной

Наша хэш-функция будет выглядеть так (часто называется *Универсальное множество хэш-функций*):

$$\texttt{h(x) = ((a * x + b) \% p) \% m}$$

$$\texttt{a, b, p} \text{ — параметры, выбираемые случайно}$$

$$0 < \texttt{a} < \texttt{p} \qquad 0 \le \texttt{b} < \texttt{p} \qquad \texttt{p} \text{ – большое простое число (большее } m)$$

Теперь нашу функцию легко генерировать, рандомя 3 числа и также легко хранить. Осталось проверить, что она продолжает удовлетворять нашему свойству о вероятности коллизий:

$$\texttt{x} \neq \texttt{y} \quad p(\texttt{h(x) = h(y)})$$

$p(\texttt{h(x) = h(y)}) \Leftrightarrow$ `((a * x + b) % p) % m = ((a * y + b) % p) % m` $\Rightarrow$
`((a * x + b) % p - (a * y + b) % p)` $\vdots$ `m` $\Rightarrow$
`(a * x + b - a * y - b) % p` $\vdots$ `m` $\Rightarrow$
`(a * (x - y)) % p` $\vdots$ `m` $\Rightarrow$
`(a * (x - y)) % p = t * m`, $\quad 0 \le \texttt{t} \le \left\lfloor \frac{p}{m} \right\rfloor$
`a = t * m * (x - y)`$^{-1}$ `(mod p)` — относительно `a` это уравнение имеет ровно 1 решение, если `a` $\neq$ `0`
$\forall$ `t` $\in \left[0, \left\lfloor \frac{p}{m} \right\rfloor\right]$ $\exists!$ `a`, для которого равенство верно $\Rightarrow$

Для каждого `t` равенство верно с вероятностью $\frac{1}{p}$
Существует порядка $\frac{p}{m}$ значений `t`

$$\frac{p}{m} \cdot \frac{1}{p} = \frac{1}{m}$$

$$p(\texttt{h(x) = h(y)}) = \frac{1}{m}, \quad \text{для} \quad \texttt{x} \neq \texttt{y}$$

Теперь мы умеем выбирать случайную хэш-функцию, которая удовлетворяет необходимым свойствам.
Вне зависимости от дальнейших входных данных, наша таблица будет работать хорошо.
Более того, даже если вдруг вышло, что злоумышленник угадал нашу хэш-функцию и пытается этим пользоваться, мы можем это обнаружить.

Для этого в процессе работы хэш-таблицы будем поддерживать размер цепочек внутри ячеек. Если он станет сильно больше $\frac{n}{m}$, значит всё пошло совсем плохо, и мы можем сгенерировать новую хэш-функцию и перестроить таблицу

Преимущества данного подхода:

- Максимально простая реализация

Недостатки данного подхода:

- Всё плохо с точки зрения кэш-памяти: все наши цепочки случайно разбросаны в реальной памяти

4. *Хэш-таблица с открытой адресацией*

В прошлой реализации мы придумали цепочки для того, чтобы уживаться с коллизиями. Побробуем обойтись без них и с хорошим процентом кэш-хитов

Пример:

```
m = 5
a = [null] * m

put(12, 8)
#a[2] = (12, 8)
put(17, 5)
#a[2] = (12, 8)
#a[3] = (17, 5)
get(12) #8
```

Массив `a` после инициализации состоит из `m` `null` значений
После добавления по ключу `12` значения `8`, `a[2] = 8`
Перед добавлением значения по ключу `17`, видим, что в `a[2]` уже лежит значение `5`
Изменим предыдущий алгоритм и вместо составления *цепочки* ключ-значение в одной ячейке нашего массива, положим элемент в первую свободную ячейку справа

Реализация:

```
put(k, v):
i = h(k)
while (a[i] != null):
  i = (i + 1) % m
a[i] = (k, v)
```

```
get(k):
i = h(k)
while (a[i] != null):
  if (a[i].first == k):
    return a[i].second
  i = (i + 1) % m
return null
```

Функция `get()` работает, так как в ней мы проходим тот же самый путь, который проходили при добавлении конкретного элемента (если добавляли его), или чуть дальше, до первого `null` значения. Но если мы добавляли элемент, то `get()` его обязательно найдёт

Однако `remove()` реализуется не так легко, как кажется. Если создать его похожим на `get()`, то наша таблица сломается после удаления `12` в указанном примере, так как на попытку найти `17`, наша таблица скажет, что такого ключа не сущесвтует, когда он есть

Помашем руками:

Скажем, что мы знаем, сколько будет добавлений в таблицу и создадим её в 2 раза большую
Предположим, что элементы по нашей хэш-таблице расположены равномерно: после заполнения

$$\texttt{p(a[i] == null) =} \ \frac{1}{2}$$

Если это так, то с вероятностью $\frac{1}{2}$ `while` совершит одну итерацию, с вероятнотью $\frac{1}{4}$ две итерации и тд.

$$E(T(\texttt{while})) = \sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

Однако наше предположение, на самом деле редко выполняется на практике, что ломает нашу оценку
Но не смотря на это, таблица работает достаточно быстро

Хотя даже для решения этой проблемы существует модификация: будем переходить вправо не на 1

```
put(k, v):
i = h(k)
while (a[i] != null):
  i = (i + f(i)) % m
a[i] = (k, v)
```

```
get(k):
i = h(k)
while (a[i] != null):
  if (a[i].first == k):
    return a[i].second
  i = (i + f(i)) % m
return null
```

```
f(x):
  return a * x + b
```

Однако и здесь не без проблем. Всю эту реализацию мы придумали, чтобы табличка хорошо укладывалась в кэш, но данная модификация с *прыжками* по массиву вместо последовательного прохода усложняет хранение таблички в кэше
Если нам не нужно часто удалять, то это вполне хороший подход к созданию хэш-таблицы, который легко реализовывать. Но если от таблицы также нужны удаления, то, конечно, подход с цепочками является предпочтительным