Хеширование и хеш-таблицы

1 Хеширование

Хеширование - процесс преобразования массива входных данных произвольной длины в выходной массив данных установленного размера, выполняемое определенным алгоритмом. Функция, воплощающая такой алгоритм называется **хеш-функцией**, а результат преобразования - **хешем** или **хеш-суммой**. Хеш-функция обладает следующими свойствами:

- одинаковые данные дают одинаковый хеш;
- разные данные "почти всегда" дают разный хеш.

Ремарка "почти всегда" во втором свойстве возникает из-за того, что хеши имеют установленный размер, в то время как входные данные этим не ограничены. В результате мы получаем, что хеш-функция делает отображение из множества входных данных во множество хешей, мощность которого будет сильно меньше. По принципу Дирихле на один хеш будет приходиться несколько различных массивов данных. Такие совпадения называются **коллизией**. Если в какой-либо задаче входные данные ограничены, можно подобрать такое множество хешей, что его мощность будет превышать мощность множества входных данных. В таком случае мы можем построить хеш-функцию, задающую инъективное отображение (идеальное хеширование). Однако, в общем случае возникновение коллизий неизбежно. Вероятность возникновения коллизий используется для оценки качества хеш-функции. Хорошая хеш-функция ведет себя следующим образом:

- весь доступный диапазон хешей используется по максимуму;
- даже небольшое изменение входных данных должно давать совершенно другой хеш, коллизии должны возникать только для совершенно разных входных данных.

Само по себе хеширование напоминает сопоставление объекту случайной величины. В результате первого свойства хеши должны вести себя как равномерно распределенные случайные величины, что обеспечит использование всего диапазона, что может быть полезно, например, при построении хеш-таблицы.

1.1 Полиномиальный хеш

Рассмотрим простой, но эффективный алгоритм хеширования. Определим нашу хеш-функцию следующим образом:

$$h(s) = \sum_{i=0}^{N} b^{N-i} \cdot code(s_i) \tag{1}$$

или

$$h(pref_i) = b \cdot h(pref_{i-1}) + code(s_{i-1}), \tag{2}$$

где $N = len(s) - 1$, $pref_i$ - префикс длины i, b - основание, $code(s_i)$ - код символа. Если развернуть формулу 1, то получится полином порядка N (отсюда и название). Формула 2 задает хеш в рекурсивном виде и будет использована при написании кода.

Стоит обратить внимание на код символа и основание, ибо от кодов будет зависеть выбор основания. Кодом может быть как код символа в таблице ASCII, так и просто порядковый номер в алфавите. Например, если в задаче гарантируется, что любая строка состоит только из строчных букв английского алфавита, то порядковый номер - хороший вариант для кодов символов. Основание должно превышать максимальный код какого-либо из возможных символов в строке, и обычно выбирается простое число (хотя строгих требований к простоте числа я не встречал). Например, для строк из строчных английских букв подойдут основания 31, 37 и т.д.

Однако, стоит заметить, что мы ни чем не ограничиваем хеш, что противоречит определению хеширования. В таком случае есть два возможных выхода: использовать операцию деления по модулю или использовать длинную арифметику.

Первый вариант широко используется в языках, где нет встроенной длинной арифметики. Более того, целочисленный тип данных, где хранится хеш, автоматически выполняет это деление (в результате переполнения типа лишние биты автоматически теряются). В результате мы получаем ограниченное множество хешей, но снова появляется риск возникновения коллизий. Так же возникает возможность “взлома” полиномиального хеша (см. 1.3).

Второй вариант имеет меньший шанс возникновения коллизий. Однако поддержка большего множества хешей обойдется дополнительной памятью и временем, которое необходимо на сравнение двух хешей, что все равно быстрее, нежели простое сравнение данных.

В языке Python уже реализована длинная арифметика, чем мы и воспользуемся при написании хешей. Для примера будем считать, что строка состоит из строчных английских букв. Основанием возьмем число 37.

Листинг 1: Python

```
def poly_hash(s):
    h = 0
    base = 37
```

```
    for c in s:
        h = h * base + ord(c) - ord('a') + 1
    return h
```

Листинг 2: C++

```
typedef unsigned long long int uint64;

uint64 poly_hash(std::string const& s) {
    uint64 h = 0;
    uint64 base = 37;
    for (auto c : s) {
        h = h * base + static_cast<uint64>(c - 'a' + 1);
    }
    return h;
}
```

1.1.1 Упражнение №1

Напишите программу, которая для заданной строки считает и печатает полиномиальный хеш. Не ограничивайтесь одним набором символов и одним основанием, пробуйте разные варианты.

Пример

Вход:	Выход:
hello	15263440

1.2 Алгоритм Рабина-Карпа

Алгоритм позволяет искать вхождения строки S в текст T при помощи хешей. Заключается он в следующем: считаем хеши для всех префиксов текста T, перебираем все подстроки длины |S| в T и сравниваем с S. Оценим время работы такого алгоритма. Время, необходимое для хеширования строки составляет O(|S|). Вспомним, что подсчет хеша для строки выражался рекурсивно через хеш строки меньшей длины. Таким образом, считая хеш для строки, мы попутно получаем хеши для всех ее префиксов, а значит для строки T нам надо O(|T|) времени. Перебор всех подстрок фиксированной длины требует линейное количество времени от длины строки T. Сравнение двух строк мы будем выполнять за O(1) благодаря хешам. Итого асимптотика алгоритма - O(|S| + |T|).

Для реализации алгоритма изменим функцию вычисления хеша:

Листинг 3: Python

```
def poly_hash(s):
    h = [0]
```

```
    p = [1]
    base = 37
    for c in s:
        h.append(h[-1] * base + ord(c) - ord('a') + 1)
        p.append(p[-1] * base)
    return (h, p)
```

Листинг 4: C++

```
typedef unsigned long long int uint64;

void poly_hash(std::string const& s,
        std::vector<uint64> &h, std::vector<uint64> &p) {
    h = {0};
    p = {1};
    uint64 base = 37;
    for (auto c : s) {
        h.push_back(*h.rbegin() * base +
            static_cast<uint64>(c - 'a' + 1));
        p.push_back(*p.rbegin() * base);
    }
}
```

Теперь мы храним хеши для всех префиксов и степени базы до len(s)-ой включительно. Для удобства при написании кода в нулевой ячейке массива мы храним хеш для префикса длины 0. А массив степеней позволит сэкономить время.

Теперь научимся вычислять хеш для произвольной подстроки. Пусть мы хотим вычислить хеш подстроки $[l, r)$. Заметим, что хеш нашей подстроки будет отличаться от хеша префикса длины r на первые l членов полинома. $h[l]$ как раз и будет содержать в себе только эти первые l членов, но меньшей степени. Домножив $h[l]$ на $base^{r-l}$, мы получим те самые l членов полинома с нужной нам степенью. Тогда хеш подстроки вычисляется как $h[r] - h[l] * p[r - l]$.

1.2.1 Упражнение №2

Дана строка S и текст T, состоящие из маленьких латинских букв. Требуется найти все вхождения строки S в текст T.

Пример

Вход:	Выход:
abc	7 13
adbcbccabcbbaabcc	

1.3 Антихеш

Как упоминалось ранее, при написании полиномиальных хешей с делением по модулю возникает вероятность коллизий. Обычно пользуются переполнением типа вместо явного деления. Например, используя в C++ тип `unsigned long long int`, по факту считают хеш по модулю 2^{64}. Именно в случае модуля степени двойки и возникает возможно подбора строки-антихеша, в которой возникает большое количество коллизий. Строка имеет вид `ABBABAABBAABABBABAABABBAABBABAAB...` и называется строкой Туэ-Морса. Про данный факт подробно написал участник соревнований на сайте codeforces.com под ником Zlobober в своем блоге. Хорошим решением этой проблемы является не использование переполнений (решение при помощи длинной арифметики). Потенциальное решение - деление по модулю простого числа. Однако в таком случае мы получаем меньший диапазон хешей, т.к. модуль должен быть сильно меньше 2^{64}, а значит сильно возрастает вероятность коллизий в общем.

2 Хеш-таблица

Хеш-таблица - структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и поддерживает три типа операций:

1. добавление новой пары;
2. поиск по ключу,
3. удаление пары по ключу.

Существуют два основных варианта хеш-таблиц: с цепочками и открытой адресацией. Хеш-таблица содержит некоторый массив H, элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица с цепочками). Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение $i = hash(key)$ играет роль индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива H[i].

Однако, не стоит забывать про коллизии. Такие события не так уж и редки — например, при вставке в хеш-таблицу размером 365 ячеек всего лишь 23-х элементов вероятность коллизии уже превысит 50% (если каждый элемент может равновероятно попасть в любую ячейку) — см. парадокс дней рождения. Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы. Способ открытой адресации тут мы рассматривать не будем.

Рассмотрим хеш-таблицу с цепочками. Каждый элемент массива H будет представлять собой связный список пар (ключ, значение), соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому,

что появляются цепочки длиной более одного элемента. Операции поиска или удаления элемента требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления элемента нужно добавить элемент в конец или начало соответствующего списка, и, в случае, если **коэффициент заполнения** (кол-во хранимых элементов, делённое на размер массива H) станет слишком велик, увеличить размер массива H и перестроить таблицу.

Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время O(1). Но при этом не гарантируется, что время выполнения отдельной операции мало.

2.1 Реализация

Многие современные языки имеют свою реализацию ассоциативного массива. Так в Python это set и dict, в C++ - std::set и std::map. Множества хоть и хранят только уникальные элементы, но реализованы на основе хеш-таблицы. Напишем свой довольно простой вариант хеш-таблицы. Заведем массив, размер которого будет являться большим числом (например, $10^5 + 7$). В случае открытой адресации некоторые методы требуют простоты числа для размера таблицы. Каждый элемент массива будет проинициализирован пустым списком. Определим нашу хеш-функцию следующим образом: остаток от деления полиномиального хеша для нашего ключа на длину нашего массива.

Листинг 5: Python

```
def get_hash(key):
    h = poly_hash(key)
    return (h % TABLE_SIZE, h)
```

Листинг 6: C++

```
void get_hash(std::string const& key,
        uint64 &pos, uint64 &h) {
    h = poly_hash(key);
    pos = h % TABLE_SIZE;
}
```

Первое число, возвращенное функцией, будет определять ячейку, где должна храниться пара. Второе число - полиномиальный хеш, который будем использовать как ключ для нашей пары. Определив ячейку, где должен храниться элемент, мы находим его в списке, сравнивая ключи. При добавлении нового элемента так же стоит проверять на наличие пары с таким же ключом, т.к. все ключи в хеш-таблице уникальны.

2.1.1 Упражнение №3

Реализуйте функции поиска, добавления и удаления элементов в множество строк. функции могут иметь следующую сигнатуру:

Листинг 7: Python

```python
def find(table: list, key: str) -> Optional[dict]:
    pass

def add(table: list, key: str):
    pass

def delete(table: list, key: str):
    pass
```

Листинг 8: C++

```cpp
struct Entry {
    uint64 hash = 0;
    std::string key = "";
};

typedef std::vector<std::list<Entry>> HashTable;

void find(HashTable const& table, std::string const& key,
          Entry *entry);

void add(HashTable & table, std::string const& key);

void delete(HashTable & table, std::string const& key);
```

2.1.2 Упражнение №4

Используя реализованное множество, напишите программу, которая будет обрабатывать запросы к множеству. Каждый запрос пишется с новой строки и имеет вид “command string”. Возможные команды:

- add - добавить элемент;
- find - найти элемент;
- delete - удалить элемент.

На запрос типа find нужно вывести Yes или No. Строки содержат только латинские строчные буквы. Также есть запрос “exit”, который просто завершает программу.

Пример

Вход:	Выход:
add hello	Yes
add bye	No
find bye	Yes
delete bye	
find bye	
find hello	
exit	

2.1.3 Упражнение №5

Переделайте ваше множество в обычный словарь, где ключами будут строки, а значениями - числа. Так же появился запрос на изменение значения. Запросы имеют следующий вид:

- add string number
- find string
- edit string number
- delete string

На запрос типа find нужно вывести хранимое число или No. На запрос типа edit нужно вывести Query error в случае отсутствия ключа. Строки содержат только латинские строчные буквы, значения представлены целыми числами. Также есть запрос “exit”, который просто завершает программу.

Пример

Вход:	Выход:
add hello 10	16
add bye 16	Query error
find bye	10
delete bye	8
edit bye 5	
find hello	
edit hello 8	
find hello	
exit	