Сириус
IT-Колледж

Лекция 14. Поиск подстроки в строке

Полиномиальный хеш, алгоритм Рабина-Карпа. Сортировка строк хешами. Хеш-таблицы. Коллизии. Открытая и закрытая адресация. Гипотеза равномерного хеширования.

Сириус
IT-Колледж

Хеширование

Хеширование (hashing) — класс методов поиска, идея которого состоит в вычислении хеш-кода, однозначно определяемого элементом с помощью хеш-функции, и использовании его, как основы для поиска (индексирование в памяти по хеш-коду выполняется за $O(1)$). В общем случае, однозначного соответствия между исходными данными и хеш-кодом нет в силу того, что количество значений хеш-функций меньше, чем вариантов исходных данных, поэтому существуют элементы, имеющие одинаковые хеш-коды — так называемые коллизии, но если два элемента имеют разный хеш-код, то они гарантированно различаются. Вероятность возникновения коллизий играет немаловажную роль в оценке качества хеш-функций. Для того чтобы коллизии не замедляли работу с таблицей существуют методы для борьбы с ними.

Сириус IT-Колледж

Полиномиальный хеш

Пусть дана строка $s[0..n-1]$. Тогда полиномиальным хешем (polynomial hash) строки s называется число $h = hash(s[0..n-1]) = p^0 s[0] + \ldots + p^{n-1} s[n-1]$, где p — некоторое простое число, а $s[i]$ – код i-ого символа строки s.

Пример:

$$s = \text{”sirius”}, p = 3$$

$$h = hash(s) = 3^0 115 + 3^1 105 + 3^2 114 + 3^3 105 + 3^4 117 + 3^5 115 =$$

$$= 115 + 315 + 1026 + 2835 + 9477 + 27945 = 41713$$

Проблему переполнения при вычислении хешей довольно больших строк можно решить так – считать хеши по модулю $r = 2^{64}$ (или 2^{32}), чтобы модуль брался автоматически при переполнении типов.

Для работы алгоритма поиска подстроки потребуется считать хеш подстроки $s[i..j]$. Делать это можно следующим образом:

Рассмотрим хеш $s[0..j]$:

$$hash(s[0..j]) = s[0] + ps[1] + \ldots + p^{i-1}s[i-1] + p^i s[i] + \ldots + p^{j-1}s[j-1] + p^j s[j]$$

Разобьем это выражение на две части:

$$hash(s[0..j]) = (s[0] + ps[1] + \ldots + p^{i-1}s[i-1]) + (p^i s[i] + \ldots + p^{j-1}s[j-1] + p^j s[j])$$

Вынесем из последней скобки множитель p^i:

$$hash(s[0..j]) = (s[0] + ps[1] + \ldots + p^{i-1}s[i-1]) + p^i(s[i] + \ldots + p^{j-i-1}s[j-1] + p^{j-i}s[j])$$

Выражение в первой скобке есть не что иное, как хеш подстроки $s[0..i-1]$, а во второй — хеш нужной нам подстроки $s[i..j]$. Итак, мы получили, что:

$$hash(s[0..j]) = hash(s[0..i-1]) + p^i hash(s[i..j])$$

Отсюда получается следующая формула для $hash(s[i..j])$:

$$hash(s[i..j]) = (1/p^i)(hash(s[0..j]) - hash(s[0..i-1]))$$

Сириус
IT-Колледж

Однако, как видно из формулы, чтобы уметь считать хеш для всех подстрок начинающихся с i, нужно предпосчитать все p^i для $i \in [0..n-1]$. Это займет много памяти. Но поскольку нам нужны только подстроки размером m – мы можем подсчитать хеш подстроки $s[0..m-1]$, а затем пересчитывать хеши для всех $i \in [0..n-m]$ за $O(1)$ следующим образом:

$$hash(s[i+1..i+m-1]) = (hash(s[i..i+m-1]) - p^{m-1}s[i]) \mod r$$

$$hash(s[i+1..i+m]) = (p \cdot hash(s[i+1..i+m-1]) + s[i+m]) \mod r$$

Получается : $hash(s[i+1..i+m]) = (p \cdot hash(s[i..i+m-1]) - p^i s[i] + s[i+m]) \mod r$

Сириус IT-Колледж

Алгоритм Рабина-Карпа

Алгоритм начинается с подсчета $hash(s[0..m-1])$ и $hash(p[0..m-1])$, а также с подсчета p^m, для ускорения ответов на запрос.

Для $i \in [0..n-m]$ вычисляется $hash(s[i..i+m-1])$ и сравнивается с $hash(p[0..m-1])$. Если они оказались равны, то образец p скорее всего содержится в строке s начиная с позиции i, хотя возможны и ложные срабатывания алгоритма. Если требуется свести такие срабатывания к минимуму или исключить вовсе, то применяют сравнение некоторых символов из этих строк, которые выбраны случайным образом, или применяют явное сравнение строк, как в наивном алгоритме поиска подстроки в строке. В первом случае проверка произойдет быстрее, но вероятность ложного срабатывания, хоть и небольшая, останется. Во втором случае проверка займет время, равное длине образца, но полностью исключит возможность ложного срабатывания.

Если требуется найти индексы вхождения нескольких образцов, или сравнить две строки − выгоднее будет предпосчитать все степени p, а также хеши всех префиксов строки s.

Сириус IT-Колледж

Алгоритм Рабина-Карпа

Алгоритм находит все вхождения строки w в строку s и возвращает массив позиций, откуда начинаются вхождения.

```
def rabinKarp (s, w):
    answer = []
    n = len(s)
    m = len(w)
    hashS = hash(s[:m])
    hashW = hash(w[:m])
    for i in range(n - m):
        if hashS == hashW:
            answer.add(i)
        hashS = (p * hashS - (p ** m) * hash(s[i]) + hash(s[i + m])) % r
    return answer
```

Новый хеш $hashS$ был получен с помощью быстрого пересчёта. Для сохранения корректности алгоритма нужно считать, что $s[n+1]$ — пустой символ.

Сириус IT-Колледж

Хеш-таблицы

Хеш-таблица (англ. hash-table) — структура данных, реализующая интерфейс ассоциативного массива. В отличие от деревьев поиска, реализующих тот же интерфейс, обеспечивают меньшее время отклика в среднем. Представляет собой эффективную структуру данных для реализации словарей, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Существует два основных вида 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. Хеш-код $i = h(key)$ играет роль индекса в массиве H, а зная индекс, мы можем выполнить требующуюся операцию (добавление, удаление или поиск).

Сириус IT-Колледж

Коллизии

Коллизия (collision): $\exists x \neq y : h(x) = h(y)$ - существует x не равный y, такой что h(x)=h(y).

Разрешение коллизий (collision resolution) в хеш-таблице, задача, решаемая несколькими способами: метод цепочек, открытая адресация и т.д. Очень важно сводить количество коллизий к минимуму, так как это увеличивает время работы с хеш-таблицами.

Сириус IT-Колледж

Разрешение коллизий с помощью цепочек

Каждая ячейка i массива H содержит указатель на начало списка всех элементов, хеш-код которых равен i, либо указывает на их отсутствие. Коллизии приводят к тому, что появляются списки размером больше одного элемента.

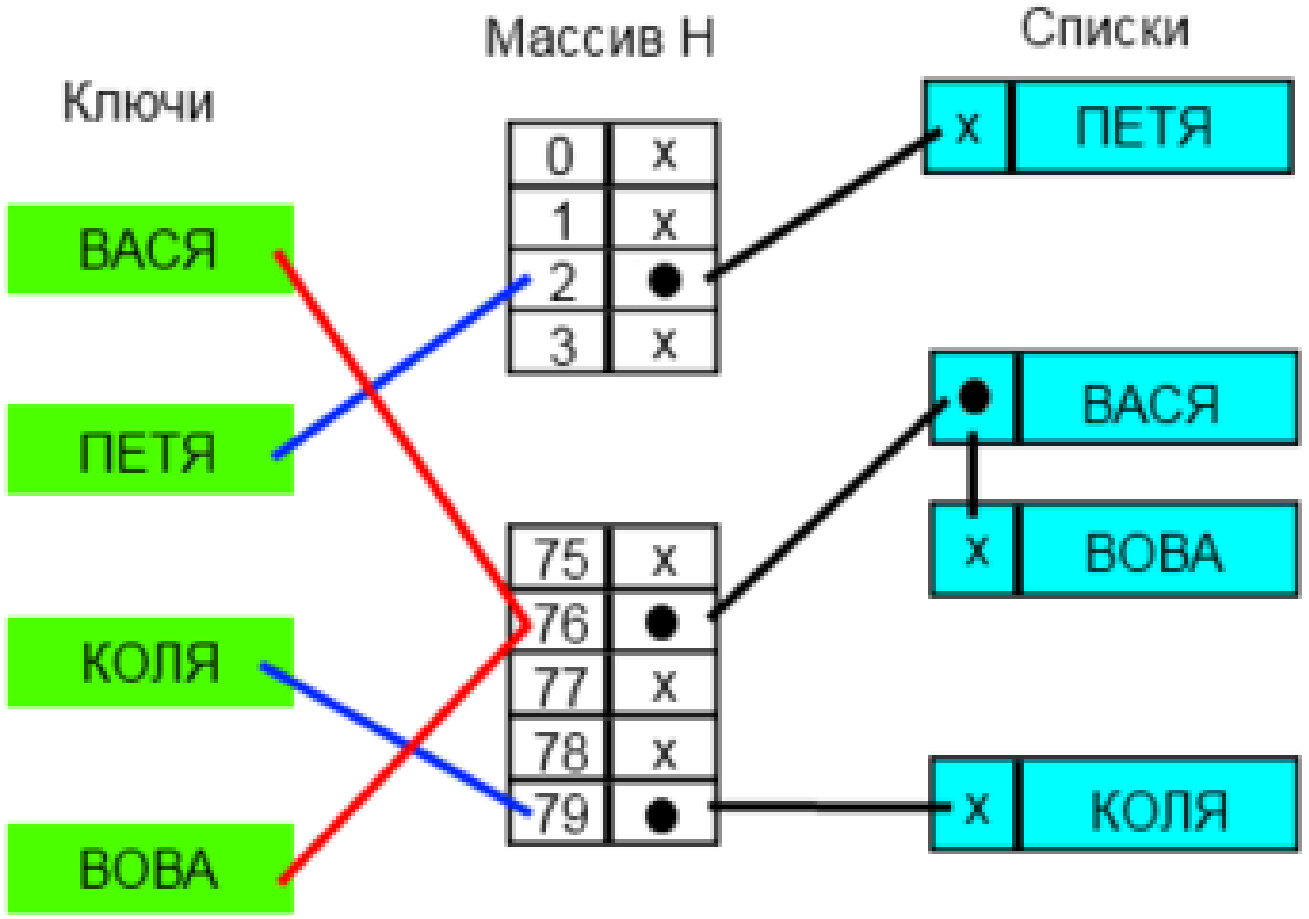
В зависимости от того нужна ли нам уникальность значений операция вставки у нас будет работать за разное время. Если не важна, то мы используем список, время вставки в который будет в худшем случае равна $O(1)$. Иначе мы проверяем есть ли в списке данный элемент, а потом в случае его отсутствия мы его добавляем. В таком случае вставка элемента в худшем случае будет выполнена за $O(n)$

Время работы поиска в наихудшем случае пропорционально длине списка, а если все n ключей захешировались в одну и ту же ячейку (создав список длиной n) время поиска будет равно $\Theta(n)$ плюс время вычисления хеш-функции, что ничуть не лучше, чем использование связного списка для хранения всех n элементов.

Удаления элемента может быть выполнено за $O(1)$, как и вставка, при использовании двухсвязного списка.

Сириус
IT-Колледж

Разрешение коллизий с помощью цепочек



Сириус
IT-Колледж

Открытая адресация

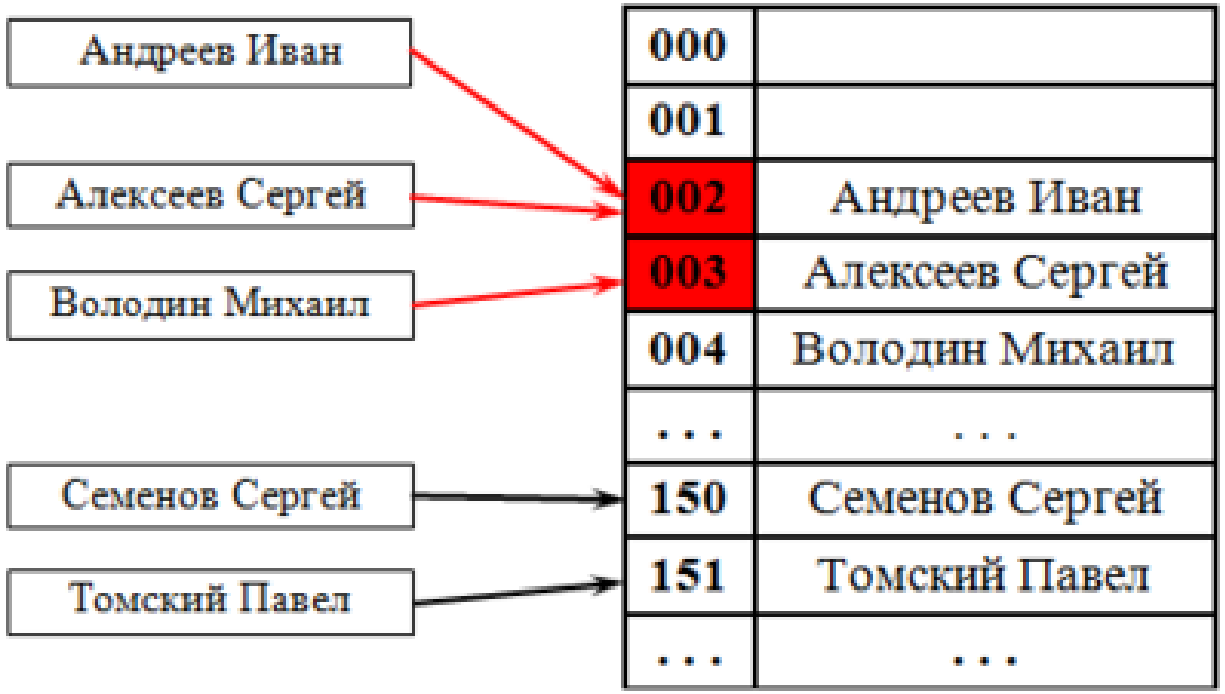
Все элементы хранятся непосредственно в хеш-таблице, без использования связных списков. В отличие от хеширования с цепочками, при использовании этого метода может возникнуть ситуация, когда хеш-таблица окажется полностью заполненной, следовательно, будет невозможно добавлять в неё новые элементы. Так что при возникновении такой ситуации решением может быть динамическое увеличение размера хеш-таблицы, с одновременной её перестройкой.

Сириус IT-Колледж

Открытая адресация

Последовательный поиск

При попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки $i+1, i+2, i+3$ и так далее, пока не найдём свободную ячейку. В неё и запишем элемент.



Сириус IT-Колледж

Открытая адресация

Линейный поиск

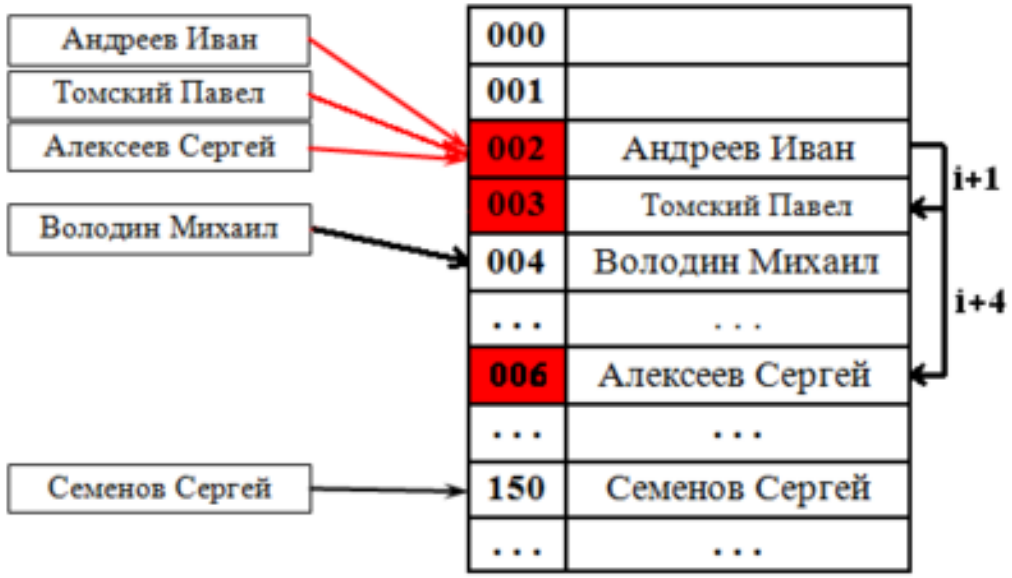
Выбираем шаг q. При попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки $i + (1 \cdot q), i + (2 \cdot q), i + (3 \cdot q)$ и так далее, пока не найдём свободную ячейку. В неё и запишем элемент. По сути последовательный поиск - частный случай линейного, где $q = 1$.

Сириус IT-Колледж

Открытая адресация

Квадратичный поиск

Шаг q не фиксирован, а изменяется квадратично: $q = 1, 4, 9, 16...$. Соответственно при попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки $i + 1, i + 4, i + 9$ и так далее, пока не найдём свободную ячейку.



Сириус
IT-Колледж

Гипотеза равномерного хеширования

Универсальное хеширование (Universal hashing) — это вид хеширования, при котором используется не одна конкретная хеш-функция, а происходит выбор из заданного семейства по случайному алгоритму. Такой подход обеспечивает равномерное хеширование: для очередного ключа вероятности помещения его в любую ячейку совпадают. Известно несколько семейств универсальных хеш-функций, которые имеют многочисленные применения в информатике, в частности в хеш-таблицах, вероятностных алгоритмах и криптографии.

Впервые понятие универсального хеширования было введено в статье Картера и Вегмана в 1979 году.

Сириус
IT-Колледж

Гипотеза равномерного хеширования

Созданный алгоритм универсального хеширования представлял собой случайный выбор хеш-функции из некоторого набора хеш-функций(называемого универсальным семейством хеш-функций), обладающих определёнными свойствами. Авторами было показано, что в случае универсального хеширования число обращений к хеш-таблице (в среднем по всем функциям из семейства) для произвольных входных данных оказывается очень близким теоретическому минимуму для случая фиксированной хеш-функции со случайно распределёнными входными данными.

Сириус IT-Колледж

Гипотеза равномерного хеширования

Пусть U — множество ключей, H — конечное множество хеш-функций, отображающих U во множество $\{0, 1, .., m-1\}$. Возьмем произвольные $h \in H$ и $x, y \in U$ и определим функцию коллизий $\delta_h(x, y)$:

$$\delta_h(x,y) = \begin{cases} 1, & \text{if } x \neq y \text{ and } h(x) = h(y) \\ 0, & \text{otherwise} \end{cases}$$

Если $\delta_h(x, y) = 1$, то говорят, что имеет место **коллизия**. Можно определить функцию коллизии не для отдельных элементов x, y, h, а для целого множества элементов — для этого надо произвести сложение функций коллизий по всем элементам из множества. Например, если H — множество хеш-функций, $x \in U$, $S \subset U$, то для функции коллизии $\delta_H(x, S)$ получим:

$$\delta_H(x, S) = \sum_{h \in H} \sum_{y \in S} \delta_h(x, y)$$

Причём порядок суммирования не имеет значения.

Сириус
IT-Колледж

Гипотеза равномерного хеширования

Семейство хеш-функций H называется **универсальным**, если

$$\forall x, y \in U \longrightarrow \delta_H(x, S) = \frac{|H|}{m}$$

Универсальные семейства хеш-функций для:

1. **Чисел**

   p - некоторое простое число

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$

Сириус IT-Колледж

Гипотеза равномерного хеширования

2. **Векторов**

   Пусть число m является простым. Пусть входные данные x представлены как последовательность $r+1$ элементов, принадлежащих $\{0, 1, .., p-1\}$, то есть $x = \langle x_0, x_1, \ldots, x_r \rangle$.

   Для всех последовательностей вида $a = \langle a_0, a_1, \ldots, a_r \rangle, a_i \in \mathbb{Z}_p, i = \overline{0, r}$ рассмотрим функцию h_a вида

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m$$

Сириус IT-Колледж

Гипотеза равномерного хеширования

3. **Строк**

$$h_a(\bar{x}) = h_a^{\text{int}}\left(\left(\sum_{i=0}^{\ell} x_i \cdot a^i \right) \bmod p \right),$$

где $h_a^{\text{int}} : \{0, 1, .., p-1\} \to \{0, 1, .., m-1\}$ является универсальной хеш-функцией для числовых аргументов.