Хеш-таблицы

Абстрактный тип данных - ассоциативный массив

Словарь (ассоциативного массива (map, dictionary, associative array)) - абстрактная структура данных позволяющая хранить пары вида "ключ - значение" и поддерживающая операции добавления пары, а также поиска и удаления пары по ключу. Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами. Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа из определенного диапазона, но и значения других типов — например, строки. В Python словарь реализуется при помощи dict().

In [1]:
```
# Базовые операции словаря:

d1 = dict() # создание пустого словаря

d1['abc'] = 42 # put(key, val) добавление пары ключ-значение в словарь;
# если по данному ключу уже есть значение - то оно заменяется новым.

v = d1['abc'] # get(key) извлечение значения по ключу;
# в случае отсутсутствия значения по ключу возникает исключительная ситуация KeyError
print(v)

b = 'xyz' in d1 # возвращает True если ключ содержится в словаре, иначе возвращает False
print(b)

it = iter(d1) # итератор по умолчанию генерирует последовательность всех ключей хранящихся в словаре
print(list(it))

l = len(d1) # возвращает количество пар ключ-значение в словаре
print(l)

del d1['abc'] # удаляет из словаря пару ключ-значение с заданным ключом
```

```
42
False
['abc']
1
```

Далее мы рассмотрим различные решения задачи реализации словаря.

Таблица с прямой адресацией

Прямая адресация представляет собой простейшую технологию, которая хорошо работает для небольших множеств ключей. Предположим, что приложению требуется динамическое множество, каждый элемент которого имеет ключ из множества $U = \{0, 1, \dots, m-1\}$, где m не слишком велико. Кроме того, предполагается, что никакие два элемента не имеют одинаковых ключей.

Для представления динамического множества мы используем массив, или таблицу с прямой адресацией, каждая ячейка которого соответствует ключу из пространства ключей U.

Возможность прямой индексации элементов обычного массива обеспечивает доступ к произвольной позиции в массиве за время $O(1)$. Прямая индексация применима, если есть возможность выделить массив размера, достаточного для того, чтобы у каждого возможного значения ключа имелась своя ячейка.

In [2]:
```
# пример: хранение строки строчных латинских символов не более заданной длины
```

In [3]:
```
def to_key(s=None):
    '''Возвращает неотрицательный ключ по объекту. Если объект не задан, то возвращает максимальное значение ключа.
    '''
    if s is None:
        s = 'z'
    return ord(s) - ord('a')
```

In [4]:
```
to_key('a')
```

Out[4]: 0

In [5]:
```
to_key('c')
```

Out[5]: 2

In [6]:
```
to_key()
```

Out[6]: 25

In [8]:
```
def seq_to_key(seq):
    res = 0
    n = to_key() # количество допустимых символов
    for i, el in enumerate(seq):
        assert 0 <= to_key(el) <= n # проверка корректности рассматриваемого символа
        res += to_key(el) * n ** i
    return res
```

In [9]:
```
seq_to_key('a')
```

Out[9]:
```
0
```

In [10]:
```
seq_to_key('ab')
```

Out[10]:
```
25
```

In [11]:
```
seq_to_key('abc')
```

Out[11]:
```
1275
```

In [12]:
```
seq_to_key('aa') # проблема!
```

Out[12]:
```
0
```

In [13]:
```
def to_key(s=None):
    '''Возвращает ПОЛОЖИТЕЛЬНЫЙ ключ по объекту. Если объект не задан, то возвращает максимальное значение ключа.
    '''
    if s is None:
        s = 'z'
    return ord(s) - ord('a') + 1
```

In [16]:
```
[seq_to_key(x) for x in ['','a', 'z', 'aa', 'zz', 'aaa']]
```

Out[16]:
```
[0, 1, 26, 27, 702, 703]
```

In [17]:
```
['a']*10
```

Out[17]:
```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```

In [18]:
```
class DirectStrTable:
    def __init__(self, str_len):
        self._str_len = str_len # максимальная длина строки в таблице
        self._n = self._to_key() # количество допустимых символов
        self._none = object() # создаем пустой объект в качестве внутреннего аналога None
        total = 0
        for i in range(self._str_len + 1):
            total *= self._n
            total += 1
        self._table = [self._none] * total
        self._len = 0 # текущее количество элементов, хранящихся в таблице

    # internal method
    def _to_key(self, s=None):
        '''Возвращает ПОЛОЖИТЕЛЬНЫЙ ключ по символу. Если объект не задан, то возвращает максимальное значение клю
        '''
        if s is None:
            s = 'z'
        return ord(s) - ord('a') + 1

    def _from_key(self, k):
        return chr(k + ord('a') - 1)

    # internal method
    def _str_to_key(self, seq):
        res = 0
        assert len(seq) <= self._str_len
        for i, el in enumerate(seq):
            assert 0 <= to_key(el) <= self._n # проверка корректности рассматриваемого символа
            res += to_key(el) * self._n ** i
        return res

    def _key_to_str(self, k):
        s = ''
        while k > 0:
            k, m = divmod(k, self._n)
            if m == 0:
                s += self._from_key(self._n)
                k -= 1
            else:
                s += self._from_key(m)
#             s += self._from_key(self._n) if m == 0 else self._from_key(m)
        return s

    # len(dst)
    def __len__(self):
        return self._len

    # dst[str_key]
    def __getitem__(self, str_key):
        k = self._str_to_key(str_key)
        if self._table[k] is self._none:
            raise KeyError(f'Key Error: {repr(s)}') # по данному ключу значения в таблице нет
        else:
            return self._table[k]

    # str_key in dst
    def __contains__(self, str_key):
        k = self._str_to_key(str_key)
        if self._table[k] is self._none:
            return False
        else:
            return True

    # dst[str_key] = val
    def __setitem__(self, str_key, val):
        k = self._str_to_key(str_key)
        if self._table[k] is self._none:
            # в таблицу добавляется новый элемент
            self._table[k] = val
            self._len += 1
        else:
            # в таблице меняется значение существующего элемента
            self._table[k] = val

    # del dst[str_key]
    def __delitem__(self, str_key):
        k = self._str_to_key(str_key)
        if self._table[k] is self._none:
            raise KeyError(f'Key Error: {repr(s)}') # по данному ключу значения в таблице нет
        else:
            self._table[k] = self._none
        self._len -= 1

    # for x in dst
    def _raw_iter(self):
        for rk, val in enumerate(self._table):
            if val is not self._none:
                yield rk, val # self._key_to_str(rk)
```

```
    def __iter__(self):
        for rk, val in self._raw_iter():
            yield self._key_to_str(rk) #, val #

    def items(self):
        for rk, val in self._raw_iter():
            yield self._key_to_str(rk), val

    def values(self):
        for val in self._table:
            if val is not self._none:
                yield val

    def stat(self):
        return f'''Хранится элементов: {self._len};
размер таблицы: {len(self._table)};
доля используемых элементов таблицы: {self._len/len(self._table)}'''
```

In [19]:
```
dst1 = DirectStrTable(3)
```

In [20]:
```
len(dst1)
```

Out[20]: 0

In [21]:
```
dst1['ab'] = 11
```

In [22]:
```
len(dst1)
```

Out[22]: 1

In [23]:
```
dst1['ab']
```

Out[23]: 11

In [24]:
```
'ab' in dst1, 'a' in dst1
```

Out[24]: (True, False)

In [25]:
```
dst1['abc'] = 7
```

In [26]:
```
len(dst1)
```

Out[26]: 2

In [27]:
```
dst1['zz'] = 77
len(dst1)
```

Out[27]: 3

In [28]:
```
del dst1['zz']
len(dst1)
```

Out[28]: 2

In [29]:
```
dst1['zzz'] = 1
dst1['x'] = 5
dst1['z'] = 6
dst1['aa'] = 7
dst1['zz'] = 8
dst1['aaa'] = 9
```

In [31]:
```
list(dst1)
```

Out[31]: ['x', 'z', 'aa', 'ab', 'zz', 'aaa', 'abc', 'zzz']

In [32]:
```
list(dst1._raw_iter())
```

Out[32]:
```
[(24, 5),
 (26, 6),
 (27, 7),
 (53, 11),
 (702, 8),
 (703, 9),
 (2081, 7),
 (18278, 1)]
```

In [33]:
```
list(dst1.items())
```

Out[33]:
```
[('x', 5),
 ('z', 6),
 ('aa', 7),
 ('ab', 11),
 ('zz', 8),
 ('aaa', 9),
 ('abc', 7),
 ('zzz', 1)]
```

```
In [34]: list(dst1.values())
Out[34]: [5, 6, 7, 11, 8, 9, 7, 1]

In [35]: d1 = dict([('ab', 11), ('abc', 7)])

In [36]: list(d1)
Out[36]: ['ab', 'abc']

In [37]: list(d1.values())
Out[37]: [11, 7]

In [39]: print(dst1.stat())
```

```
Хранится элементов: 8;
размер таблицы: 18279;
доля используемых элементов таблицы: 0.00043766070353958096
```

Каждая из приведенных операций очень быстрая: время их работы равно $O(1)$. В некоторых приложениях элементы динамического множества могут храниться непосредственно в таблице с прямой адресацией.

Недостаток прямой адресации очевиден: если пространство ключей U велико, хранение таблицы T размером $|U|$ непрактично, а то и вовсе невозможно – в зависимости от количества доступной памяти и размера пространства ключей. Кроме того, множество K реально сохраненных ключей может быть мало по сравнению с пространством ключей U, а в этом случае память, выделенная для таблицы T, в основном расходуется напрасно.

Хеш-таблица

Хеш-таблица представляет собой эффективную структуру данных для **реализации словарей**. Хотя на поиск элемента в хеш-таблице может в наихудшем случае потребоваться столько же времени, что и в связанном списке, а именно $O(n)$, на практике хеширование исключительно эффективно. При вполне обоснованных допущениях **математическое ожидание времени поиска** элемента в хеш-таблице составляет $O(1)$.

Сравнение разынх методов реализации словарей: https://en.wikipedia.org/wiki/Associative_array (https://en.wikipedia.org/wiki/Associative_array)

Хеш-таблица (hash table) представляет собой обобщение обычного массива. Если количество реально хранящихся в массиве ключей мало по сравнению с количеством возможных значений ключей, эффективной альтернативой массива с прямой индексацией становится **хеш-таблица**, которая обычно использует массив с размером, пропорциональным количеству реально хранящихся в нем ключей. Вместо непосредственного использования ключа в качестве индекса массива, **индекс вычисляется по значению ключа**. Идея хеширования состоит в использовании некоторой частичной информации, полученной из ключа, т.е. вычисляется хеш-адрес $h(key)$, который используется для индексации в хеш-таблице.

Когда множество K хранящихся в словаре ключей гораздо меньше пространства возможных ключей U, хеш-таблица требует существенно меньше места, чем таблица с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до $\Theta(|K|)$, при этом время поиска элемента в хеш-таблице остается равным $O(1)$. Надо только заметить, что это граница среднего времени поиска, в то время как в случае таблицы с прямой адресацией эта граница справедлива для наихудшего случая.

Хеш-таблица это коллекция, хранящая проиндексированные элементы (значения). Каждая позиция в хеш-таблице - слот таблицы (slot, bucket) может содержать элемент адресованный целочисленным не отрицательным индексом внутри таблицы. Т.е. в таблице есть слот с индексом 0, 1 и т.д. При создании все слоты в хеш-таблице пустые. В Python хеш-таблицу мужно реализовать в виде списка заполненного значениями None или их заменителями (для случаев, если пользователю нужно хранить в хеш-таблице значения None).

В случае прямой адресации элемент с ключом k хранится в ячейке k. При хешировании этот элемент хранится в ячейке $h(k)$, т.е. мы используем хеш-функцию h для вычисления ячейки для данного ключа k. Функция h отображает пространство ключей U на ячейки хеш-таблицы $\mathrm{T}[0..m-1]$:

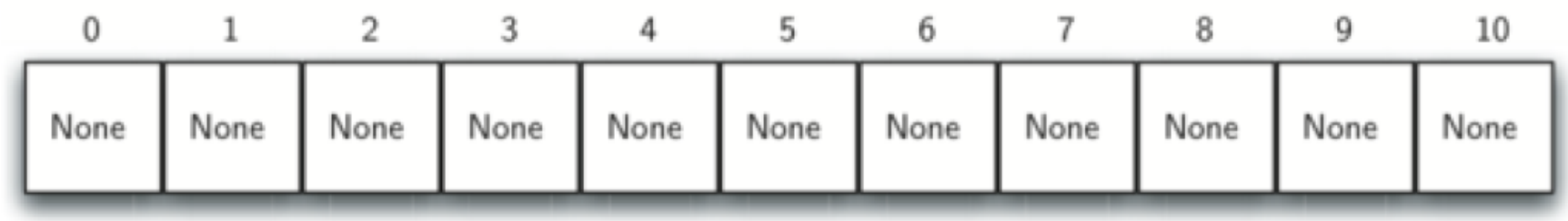
$$h : U \to \{0, 1, \ldots, m-1\}.$$

Мы говорим, что элемент с ключом k хешируется в ячейку $h(k)$, величина $h(k)$ называется хеш-значением ключа k.

Пример

$T[0, \ldots, 10]; h(k) = k \mod 11$

Пустая хеш-таблица:



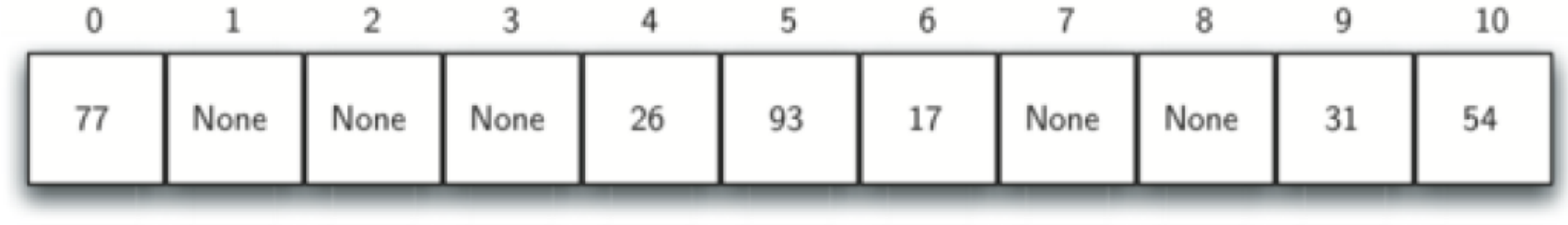
In [40]:
```
def h(k):
    return k % 11
```

In [41]:
```
keys1 = [54, 26, 93, 17, 77, 31]
```

In [42]:
```
[h(k) for k in keys1]
```

Out[42]:
```
[10, 4, 5, 6, 0, 9]
```

Хеш-таблица с 6ю элементами:



При построении хеш-таблиц есть одна проблема: два ключа могут быть хешированы одну и ту же ячейку. Такая ситуация называется **коллизией**.

In [43]:
```
# пример коллизии:
h(12), h(23)
```

Out[43]:
```
(1, 1)
```

Т.к. h является детерминистической и для одного и того же значения k всегда дает одно и то же хеш-значение $h(k)$, то поскольку $|U| > m$, должно существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Таким образом, полностью **избежать коллизий невозможно** в принципе.

Используются два подхода для борьбы с этой проблемой:

- выбор хеш-функции снижающей вероятность коллизии;
- использование эффективных алгоритмов разрешения коллизий.

Хеш-функция

Хеш-функция выполняет преобразование массива входных данных произвольной длины (ключа, сообщения) в (выходную) битовую строку установленной длины (хеш, хеш-код, хеш-сумму).

Хеш-функции применяются в следующих задачах:

- построение ассоциативных массивов;
- поиске дубликатов в сериях наборов данных;
- построение уникальных идентификаторов для наборов данных;
- вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
- сохранении паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
- выработке электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ»); и многих других.

Для решения различных задач требования к хеш-функциям могут очень существенно отличаться.

"Хорошая" хеш-функция должна удовлетворять двум свойствам:

- быстрое вычисление;
- минимальное количество коллизий.

Для обеспечения минимального количества коллизий хеш-функция удовлетворяет (приближенно) предположению простого **равномерного хеширования**: для каждого ключа равновероятно помещение в любую из m ячеек, независимо от хеширования остальных ключей. К сожалению, это условие обычно невозможно проверить, поскольку, как правило, распределение вероятностей, в соответствии с которым поступают вносимые в таблицу ключи, неизвестно; кроме того, вставляемые ключи могут не быть независимыми.

При построении хеш-функции хорошим подходом является подбор функции таким образом, чтобы она никак **не коррелировала с закономерностями**, которым могут подчиняться существующие данные. Например, мы можем потребовать, чтобы **"близкие" в некотором смысле ключи давали далекие хеш-значения** (например, хеш функция для подряд идущих целых чисел давала далекие хеш-значения). В некоторых приложениях хеш-функций требуется противоположное свойство - непрерывность (близкие ключи длолжны порождать близкие хеш-значения).

Обычно от хеш-функций ожидается, что значения хеш-функции находиятся в диапазоне от 0 до $m - 1$. Причём, часто удобно, есчли $m = 2^n$. Таким образом значение хеш-функции может, например, без преобразований хранится в машинном слове.

Метод деления

Построение хеш-функции методом деления состоит в отображении ключа k в одну из ячеек путем получения остатка от деления k на m, т.е. хеш-функция имеет вид $h(k) = k \mod m$.

При использовании данного метода мы обычно стараются избегать некоторых значений m. Например, m не должно быть степенью 2, поскольку если $m = 2^n$, то $h(k)$ представляет собой просто p младших битов числа k. Если только заранее неизвестно, что все наборы младших p битов ключей равновероятны, лучше строить хеш-функцию таким образом, чтобы ее результат зависел от всех битов ключа. Зачастую хорошие результаты можно получить, выбирая в качестве значения m простое число, достаточно далекое от степени двойки.

In [44]:
```
# пример хеш-функции для строк, построенной по методу деления:

def str_h(s, m=701):
    return sum(ord(symb) for symb in s) % m
```

In [46]:
```
s1 = 'Хеш-функция выполняет преобразование массива входных данных произвольной длины (ключа, сообщения) в (выходную) б
s1_cod = [(s, str_h(s)) for s in s1.split()]
s1_cod
```

Out[46]:
```
[('Хеш-функция', 391),
 ('выполняет', 671),
 ('преобразование', 403),
 ('массива', 550),
 ('входных', 596),
 ('данных', 201),
 ('произвольной', 397),
 ('длины', 516),
 ('(ключа,', 611),
 ('сообщения)', 3),
 ('в', 373),
 ('(выходную)', 375),
 ('битовую', 586),
 ('строку', 217),
 ('установленной', 64),
 ('длины', 516),
 ('(хеш,', 546),
 ('хеш-код,', 290),
 ('хеш-сумму).', 425)]
```

In [47]:
```
sorted(s[1] for s in s1_cod)
```

Out[47]:
```
[3,
 64,
 201,
 217,
 290,
 373,
 375,
 391,
 397,
 403,
 425,
 516,
 516,
 546,
 550,
 586,
 596,
 611,
 671]
```

Метод MAD

Хеш-функция multiply-add-and-divide (часто именуемая как MAD) преобразует целое число k по следующему алгоритму. У хеш-функци имеются следующие параметры: p - большое простое число, $a \in \{1, 2, \ldots, p-1\}$ и $b \in \{0, 1, \ldots, p-1\}$, m - количество значений в диапозоне значений хеш-функции.

$$h_{a,b}(k) = ((ak + b) \mod p) \mod m$$

Этот класс хеш-функций удобен тем, что размер m выходного диапазона произволен и не обязательно представляет собой простое число. Поскольку число, a можно выбрать $p - 1$ способом, и p способами – число b, всего в данном семействе будет содержаться $p(p-1)$ хеш-функций.

Данную хеш-функцию (семейство хеш-функций) можно использовать для **универсального хеширования**. Универсальным хешированием называется хеширование, при котором используется не одна конкретная хеш-функция, а происходит выбор хеш-функции из заданного семейства по случайному алгоритму. Семейство универсальных хеш-функций называется универсальным, если для любых двух допустимых ключей вероятность коллизии является наименьшей из возможных:

$$\Pr_{h \in \mathcal{H}} [h(x) = h(y)] = \frac{1}{m}, x \neq y$$

Нужно отметить, что это утверждение верно дложно выполняться для любых различающихся ключей, при этом случайно выбираются здесь именно функции хеширования h из всего класса универсальных хеш-функций $\mathcal{H}$.

Универсальное хеширование обычно отличается низким числом коллизий и применяется, например, при реализации хеш-таблиц и в криптографии. В частности, можно показать, что хеш-функции MAD для случано выбранных a и b являются классом универсальных хеш-функций (см. https://en.wikipedia.org/wiki/Universal_hashing#Hashing_strings) (https://en.wikipedia.org/wiki/Universal_hashing#Hashing_strings)).

In [48]:
```
import random
```

In [50]:
```
2**32 # количество целых чисел в 32 битовом слове
```

Out[50]:
```
4294967296
```

In [51]:
```
mad_p = 4294967311 # https://www.numberempire.com/primenumbers.php
```

In [53]:
```
class MadHash:
    def __init__(self, m):
        self.m = m
        self.p = mad_p
        assert self.p > self.m
        self.a = random.randint(1, self.p)
        self.b = random.randint(0, self.p)

    def h(self, k):
        return ((self.a * k + self.b) % self.p ) % self.m
```

In [54]:
```
mh = MadHash(2**8)
mh.a, mh.b
```

Out[54]:
```
(2901740422, 311557595)
```

In [57]:
```
hr1 = [mh.h(v) for v in range(128)]
hr1
```

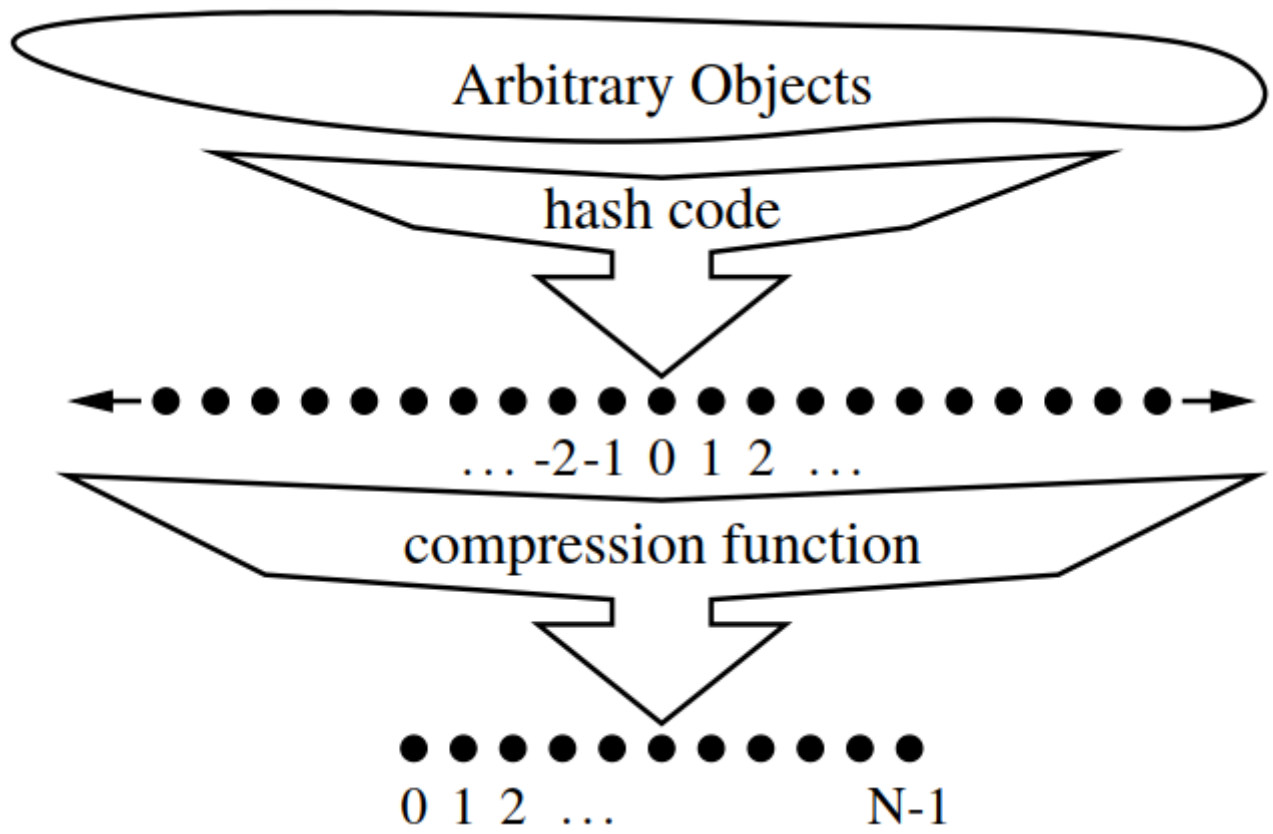
Out[57]:
```
[219,
 97,
 216,
 79,
 213,
 76,
 195,
 73,
 192,
 55,
 189,
 52,
 171,
 49,
 168,
 31,
 165,
 28,
 147,
```

In [58]:
```
hr1s = sorted(hr1)
list(zip(hr1s, [x0 == x1 for x0, x1 in zip(hr1s[:-1], hr1s[1:])]))
```

Out[58]:
```
[(0, False),
 (1, True),
 (1, False),
 (2, False),
 (3, False),
 (4, True),
 (4, False),
 (5, False),
 (6, False),
 (7, False),
 (22, False),
 (23, False),
 (24, False),
 (25, True),
 (25, False),
 (26, False),
 (27, False),
 (28, True),
 (28, False),
```

Зачастую расчет хеш-функции $h(k)$ можно представить в виде двух операций: рассчета хеш-кода который превращает ключ k в целое число и функции компрессии, которая преобразует хеш-код в целое число в заданном диапазоне $[0, m-1]$.

Схема построения хеш-функции на базе двух шагов:



Преимуществом разделения хеш-функции на две компоненты является то, что рассчет хеш-кода проивзодится независимо от того с каким размером хеш-таблицы нужно будет работать. Это **позволяет разрабатывать функции рассчета хеш-кодов для различных типов данных** не ориентируясь на размер хеш-таблицы, который важен только для функции компрессии. Это особенно удобно, т.к. **размер хеш таблицы может быть динамечески изменен в зависимости от количества элементов, хранимых в словаре**.

Функция MAD может использоваться и в качестве функции построения хеш-кода для целых чисел и в качестве функции компрессии для хеш-кодов, построенных с помощью других функций.

Полиномиальная хеш-функция

Приведенная ранее функция построения хеш-кода, основанная на суммировании (или опреации xor) хеш-кодов, плохо подходит для работы с символьными строками и другими объектами различной длины, которые могут быть представлены в виде кортежа $(x_0, x_1, \ldots, x_{n-1})$, в котором позиция элемента x_i имеет значение, т.к. такой хеш-код создает коллизии для строк (последовательностей) с одинаковым составом элементов.

In [59]:
```
# примеры коллизий для хеш-функций строк основанных на суммировании
s2 = ["stop", "tops", "pots", "spot"]
[(s, str_h(s)) for s in s2]
```

Out[59]:
```
[('stop', 454), ('tops', 454), ('pots', 454), ('spot', 454)]
```

Такого рода коллизии не будут возникать в хеш-функции, которая учитыват положение элементов в массиве входных данных. Примером такой хеш-функции является функция, использующая константу a $(a \neq 0, a \neq 1)$ при построении хеш-функции вида:

$$x_0 a^{n-1} + x_1 a^{n-2} + \ldots + x_{n-2} a + x_{n-1}.$$

Т.е. это полином, использующий элементы массива входных данных $(x_0, x_1, \ldots, x_{n-1})$ в качестве коэффициентов. Такая функция назвается полиниомиальным хеш-кодом. Для использования ее в качестве хеш-функции к ней необходимо только добавить функцию компресии в соответствующий диапозон значений.

Используя схему Горнера полиномиальный хеш-код можно эффективно вычислить по формуле:

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \ldots + a(x_2 + a(x_1 + a x_0)) \ldots))$$

Функция hash в Python

Стандартным способом для получения хеш-кода в Python является встроенная функция **hash(x)**. Она возвращает целочисленное значение для объекта x. Однако, в Python **только неизменяемые типы** данных могут возвращать значение хеш-кода. Это ограчиние гарантирует, что хеш-код для объека не изменится во время его жизни. Это свойство очень важно для корректной работы при использовании хеш-кодов объектов в хеш-таблицах, например в dict().

In [241]:
```
hash('Hello world!')
```

Out[241]:
```
5241752689838367761
```

In [60]:
```
hash(42)
```

Out[60]:
```
42
```

In [61]:
```
hash(3.141)
```

Out[61]:
```
325123864299130883
```

In [62]:
```
hash((1, 2))
```

Out[62]:
```
3713081631934410656
```

In [65]:
```
hash(None)
```

Out[65]:
```
122410684
```

In [67]:
```
hash(frozenset([1, 2]))
```

Out[67]:
```
-1834016341293975159
```

In [68]:
```
# ошибка:
hash([1, 2])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-68-2b47f0a42a13> in <module>()
      1 # ошибка:
----> 2 hash([1, 2])

TypeError: unhashable type: 'list'
```

Важным правилом реализации функции hash для классов является необходимость сохранять **консистентность между равенством (x == y) и равенством хеш-функций (hash(x) == hash(y))**. Для любых двух объектов из равенства x == y должно следовать hash(x) == hash(y) (из-за возможности коллизий у хеш-функций следствие в обратную сторону в общем случае не выполняется). Это необходимо для того, чтобы в случае использования объекта в качестве ключа в хеш таблицы для равных объектов (x == y) результат поиска в таблице (который ведется с ипользованием hash(x), hash(y)) был идентичен.

In [69]:
```
42 == 42.0
```

Out[69]:
```
True
```

In [70]:
```
hash(42), hash(42.0), hash(42.0000001)
```

Out[70]:
```
(42, 42, 230584303658)
```

In [71]:
```
# реализация hash для пользовательского типа данных:

class Color:
    def __init__(self, r, g, b):
        assert type(r) is int
        assert 0 <= r <= 255
        self.__red = r
        assert type(g) is int
        assert 0 <= g <= 255
        self.__green = g
        assert type(b) is int
        assert 0 <= b <= 255
        self.__blue = b

    @property
    def red(self):
        return self.__red

    @property
    def green(self):
        return self.__green

    @property
    def blue(self):
        return self.__blue

    def __hash__(self):
        return hash((self.__red, self.__green, self.__blue))

    def __eq__(self, other):
        return self.__red == other.red and\
    self.__green == other.green and self.__blue == other.blue
```

In [72]:
```
c1 = Color(2, 2, 115)
c1.red, c1.green, c1.blue
```

Out[72]:
```
(2, 2, 115)
```

In [73]:
```
hash(c1)
```

Out[73]:
```
3789703756491517098
```

In [74]:
```
c2 = Color(2, 2, 115)
c1 == c2
```

Out[74]:
```
True
```

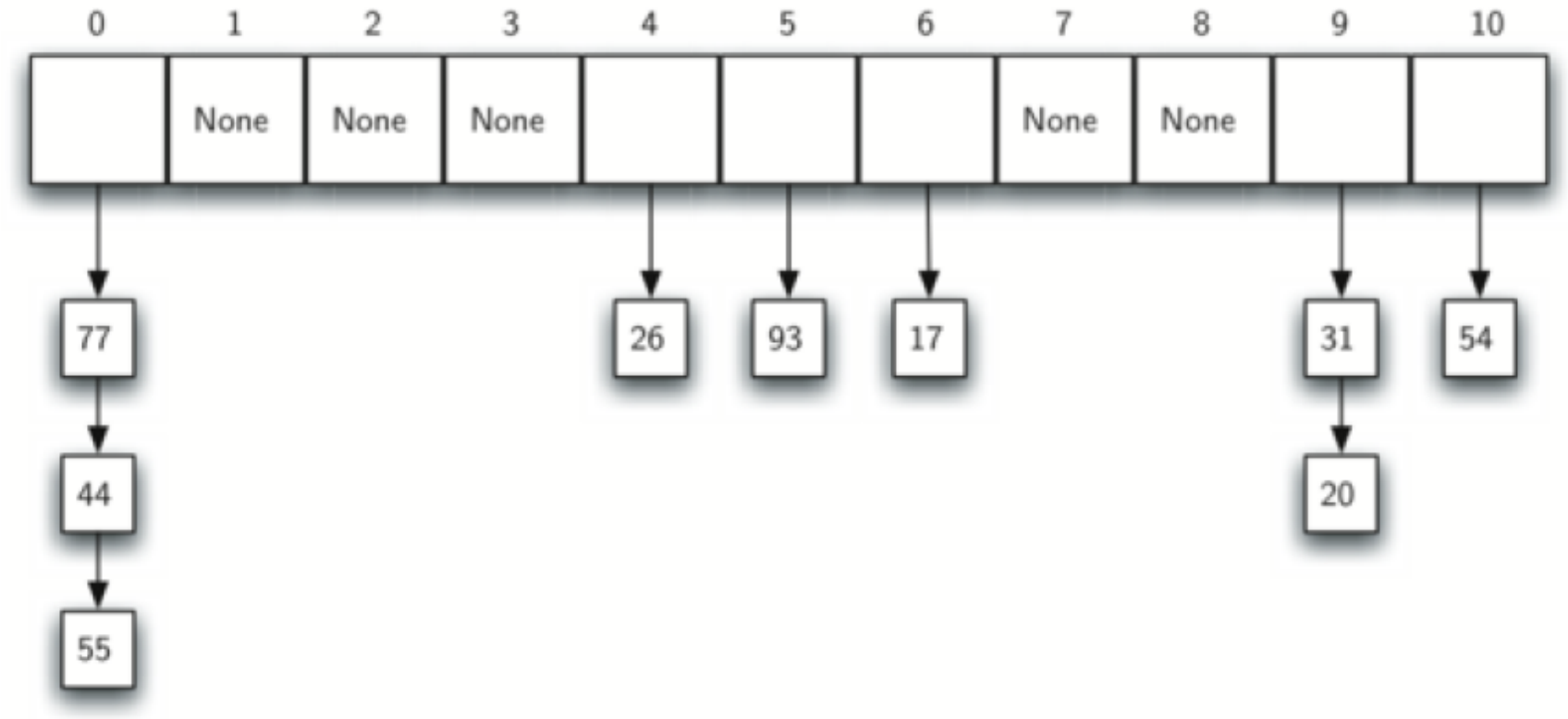
In [75]:
```
hash(c1) == hash(c2)
```

Out[75]:
```
True
```

Методы разрешения коллизий

Разрешение коллизий при помощи цепочек. При использовании данного метода все элементы, хешированные в одну и ту же ячейку, объединяются в связанный список, как показано на рис. Ячейка j или содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно j; если таких элементов нет, то ячейка содержит значение None.

Хеш-таблица использующая цепочки для разрешения коллизий:



In [154]:
```
def h(k):
    return k % 11
```

In [185]:
```
keys1 = [54, 26, 93, 17, 77, 31, 44, 20, 55]
```

In [186]:
```
[h(k) for k in keys1]
```

Out[186]:
```
[10, 4, 5, 6, 0, 9, 0, 9, 0]
```

Время, необходимое для вставки в наихудшем случае, равно $O(1)$. Процедура вставки выполняется очень быстро, поскольку предполагается, что вставляемый элемент отсутствует в таблице. При необходимости это предположение может быть проверено путем выполнения поиска перед вставкой. Время работы поиска в наихудшем случае пропорционально длине списка. Удаление элемента может быть выполнено за время $O(1)$.

Открытая адресация

При использовании метода открытой адресации все элементы хранятся непосредственно в хеш-таблице, т.е. каждая запись таблицы содержит либо элемент динамического множества, либо специальное значение (None или его заменитель).

При поиске элемента мы систематически проверяем ячейки таблицы до тех пор, пока не найдем искомый элемент или пока не убедимся в его отсутствии в таблице. Здесь, в отличие от метода цепочек, нет ни списков, ни элементов, хранящихся вне таблицы. Таким образом, в методе открытой адресации хеш-таблица может оказаться заполненной, делая невозможной вставку новых элементов.

Коэффициент заполнения таблицы (load factor):

$$\lambda = \frac{number\_of\_items}{table\_size}$$

В хеш-таблице с открытой адресацией коээфициент заполнения таблицы λ не может превышать 1.

Вместо того чтобы следовать по указателям, при открытой адресации мы вычисляем последовательность проверяемых ячеек. Дополнительная память, освобождающаяся в результате отказа от указателей, позволяет использовать хеш-таблицы большего размера при том же общем количестве памяти, потенциально приводя к меньшему количеству коллизий и более быстрой выборке.

Для выполнения вставки при открытой адресации мы последовательно проверяем, или исследуем (probe), ячейки хеш-таблицы до тех пор, пока не находим пустую ячейку, в которую помещаем вставляемый ключ.

Для определения исследуемых ячеек хеш-функция расширяется, в нее включается в качестве второго аргумента номер исследования (начинающийся с 0). В методе открытой адресации требуется, чтобы для каждого ключа k последовательность исследований $h(k,0), h(k,1), \ldots, h(k,m-1)$ представляла собой перестановку множества $\{0,1,\ldots,m-1\}$, чтобы в конечном счете могли быть просмотрены все ячейки хеш-таблицы.

```
% Псевдокод помоещения значения в хеш-таблицу с открытой адресацией:
HASH_INSERT(T, k)
1 i ← 0 \n
2 repeat j ← h(k, i)
3   if T[j] = NIL
4   then T[j] ← k
5     return j
6   else i ← i + 1
7 until i = m
8 error "Хеш-таблица переполнена"
```

```
% Псевдокод поиска значения в хеш-таблице с открытой адресацией:
HASH_SEARCH(T, k)
1 i ← 0
2 repeat j ← h(k,i)
3   if T[j] =k
4   then return j
5   i ← i + 1
6 until T[j] = NIL или i= m
7 return NIL
```

Процедура **удаления из хеш-таблицы** с открытой адресацией достаточно сложна. При удалении ключа из ячейки i мы не можем просто пометить ее значением NIL. Поступив так, мы можем сделать невозможным выборку ключа k, в процессе вставки которого исследовалась и оказалась занятой ячейка i. Одно из решений состоит в том, чтобы помечать такие ячейки специальным значением DELETED вместо NIL. При этом мы должны слегка изменить процедуру HASH_INSERT, с тем, чтобы она рассматривала такую ячейку, как пустую и могла вставить в нее новый ключ. В процедуре HASH_SEARCH никакие изменения не требуются, поскольку мы просто пропускаем такие ячейки при поиске и исследуем следующие ячейки в последовательности. Однако при использовании специального значения DELETED время поиска перестает зависеть от коэффициента заполнения.

Линейное исследование

Пусть задана обычная хеш-функция $h' : U \to \{0, 1, \ldots, m-1\}$, которую мы будем в дальнейшем именовать вспомогательной хеш-функцией (auxiliary hash function). Метод линейного исследования для вычисления последовательности исследований использует хеш-функцию

$$h(k, i) = (h'(k) + i) \mod m$$

Линейное исследование легко реализуется, однако с ним связана **проблема первичной кластеризации**, связанной с созданием длинных последовательностей занятых ячеек, что, увеличивает среднее время поиска. Кластеры возникают в связи с тем, что вероятность заполнения пустой ячейки, которой предшествуют i заполненных ячеек, равна $(i + 1)/m$. Таким образом, длинные серии заполненных ячеек имеют тенденцию к все большему удлинению, что приводит к увеличению среднего времени поиска.

Хеш-таблица с открытой адресацией и линейным исследованием:

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

```
In [185]: keys1 = [54, 26, 93, 17, 77, 31, 44, 20, 55]
```

```
In [186]: [h(k) for k in keys1]
```

```
Out[186]: [10, 4, 5, 6, 0, 9, 0, 9, 0]
```

Квадратичное исследование

Квадратичное исследование использует хеш-функцию вида:

$$h(k, i) = (h'(k) + c_1 i + c_2 i_2) \mod m$$

где h' – вспомогательная хеш-функция, c_1 и $c_2 \neq 0$ – вспомогательные константы, а i принимает значения от 0 до $m - 1$ включительно.

Начальная исследуемая ячейка - $T[h'(k)]$; остальные исследуемые позиции смещены относительно нее на величины, которые описываются квадратичной зависимостью от номера исследования i.

Этот метод работает существенно лучше линейного исследования, но для того, чтобы исследование охватывало все ячейки, необходим выбор специальных значений c_1, c_2 и m . Кроме того, если два ключа имеют одну и то же начальную позицию исследования, то одинаковы и последовательности исследования в целом.

Двойное хеширование

Двойное хеширование представляет собой один из наилучших способов использования открытой адресации, поскольку получаемые при этом перестановки обладают многими характеристиками случайно выбираемых перестановок. Двойное хеширование использует хеш-функцию вида

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m$$

где h_1 и h_2 – вспомогательные хеш-функции. Начальное исследование выполняется в позиции $\mathrm{T}[h_1(k))]$, а смещение каждой из последующих исследуемых ячеек относительно предыдущей равно $h_2(k)$ по модулю m.

В отличие от линейного и квадратичного исследования, в данном случае последовательность исследования зависит от клю- ча k по двум параметрам – в плане выбора начальной исследуемой ячейки и расстояния между соседними исследуемыми ячейками, так как оба эти параметра зависят от значения ключа. Производительность двойного хеширования достаточно близка к производительности "идеальной" схемы равномерного хеширования.

Перехеширование (rehashing)

In the hash table schemes described thus far, it is important that the load factor, λ = n/N , be kept below 1. With separate chaining, as λ gets very close to 1, the probability of a collision greatly increases, which adds overhead to our operations, since we must revert to linear-time list-based methods in buckets that have col- lisions.

Experiments and average-case analyses suggest that we should maintain λ < 0.9 for hash tables with separate chaining. With open addressing, on the other hand, as the load factor λ grows beyond 0.5 and starts approaching 1, clusters of entries in the bucket array start to grow as well.

If an insertion causes the load factor of a hash table to go above the specified threshold, then it is common to resize the table (to regain the specified load factor) and to reinsert all objects into this new table. Although we need not define a new hash code for each object, we do need to reapply a new compression function that takes into consideration the size of the new table. Each rehashing will generally scatter the items throughout the new bucket array. When rehashing to a new table, it is a good requirement for the new array's size to be at least double the previous size.

Type *Markdown* and LaTeX: α^2