Hashing

Dictionary Abstract Data Type

a set of items, each with a `key`

- insert item
- delete item
- search item with key

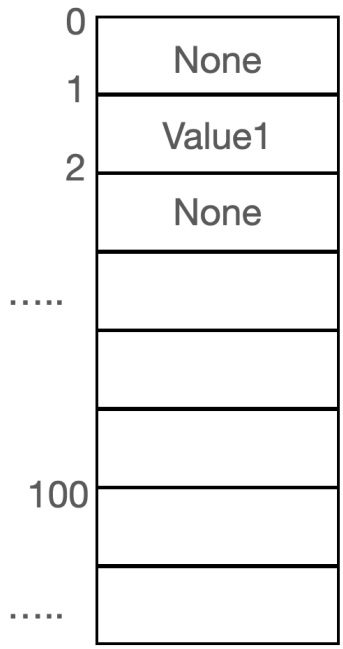
we can implement it in a AVL tree , each operation is $O(\log N)$, but we can do better, optimize them into $O(1)$.

Motivation

- document distance(similarities between two files)
- database
- build compilers and interpreters
- network router
- substring search
- string commonality
- file or directionary synchronization
- cryptography
- .....

Let's start with a simple approach : direct access table

store the values in an array indexed by keys



Problems:

- keys may not be integers
- gigantic memory hog

Solutions:

- prehashing
  - map keys to non-negative integers
    - theoretically, all things in computer world are bits, we just convert them into integers directly

```
>>> int('hello'.encode('ascii').hex() , 16)
448378203247
```

- Practically, Python has a complex implementation of hash, it has different function for different type
  - you can overwrite the hash functions behavior via redefining `__hash__`
  - for security reason, hash values are unique to each runtime

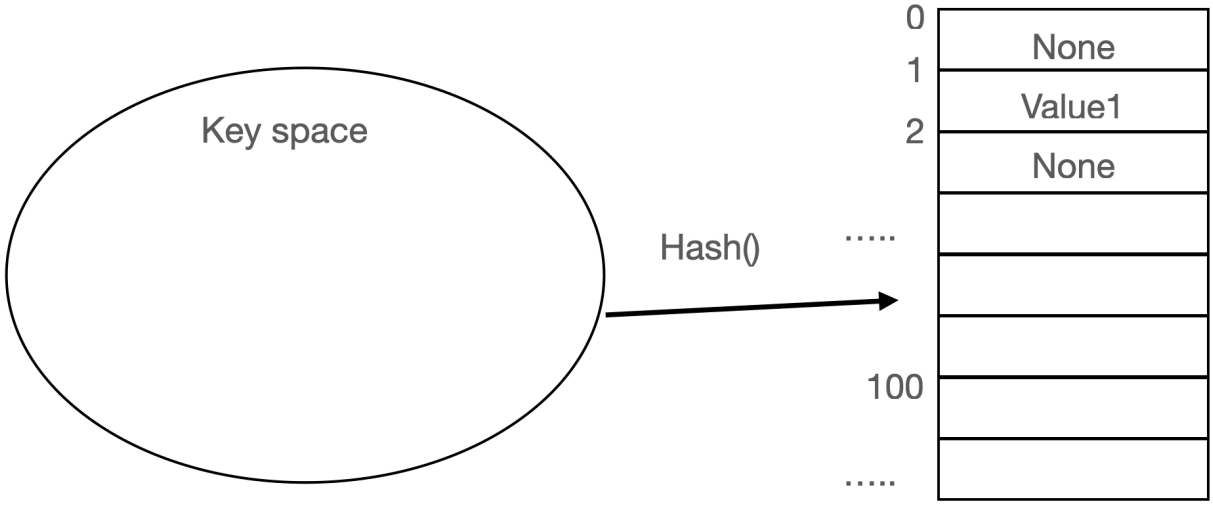
```
% for (( n = 0 ; n < 4 ; n++))
for> do
for> py -c "print(hash('hello'))"
for> done
4203706303205554004
858035038639559076 2
-1280904045718253363
-8726737701756394311
```

  - a classic collision

```
>>> hash(-1)==hash(-2)
True
>>>help(hash)
hash(obj, /)
    Return the hash value for the given
object.

    Two objects that compare equal must
also have the same hash value, but the
    reverse is not necessarily true.
```

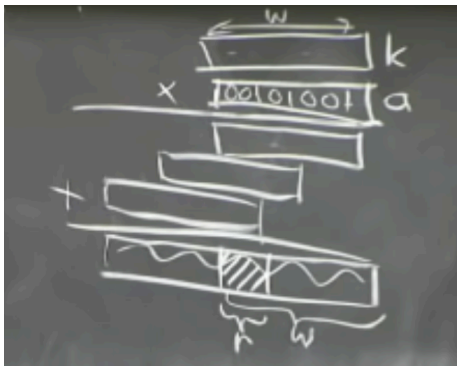
- really hashing
  - reduce universe of all keys(integers) down to reasonable size m for table where $m = O(n)$, n is the number of keys
  - we use a hash function



  - However, there are much more keys than our memory space, the collision is guaranteed.

Hash functions

- division method : $h(k) = k \mod m$
- multiplication method : $h(k) = [a \cdot k \mod 2^w] >> (w - r)$
  - k is w bits
  - a is a random number
  - length of hash table is 2^r
  - we take the left side of the right of the result(the shadowed part)



- universed hashing : $h(k) = [(ak + b) \mod p] \mod m$
  - a,b is random number in {0,1,2,...,p-1}
  - p is a big prime, which is larger than number of keys

How to choose the hashtable length m?

Goal:

- $O(m) = O(n)$

Method:

- start with a constant c
  - grow table if n > m
    - make a new table size of m', $m' = 2m$
    - build a new hash function(if we use the old hash function we will still map elements to m results rather than m' results)
    - rehashing

```
for item in oldTable:
    newTable.insert(item)
```

  - Shrink table if $n < \frac{m}{2}$
    - make a new table of $\frac{m}{2}$
    - build a new hash function
    - rehashing
  - if we perform insertion and deletion frequently, shrink when $n < \frac{m}{2}$ is not a good idea, insertion will make it grow again very soon, so $n < \frac{m}{t}, t \geq 3$ is better.

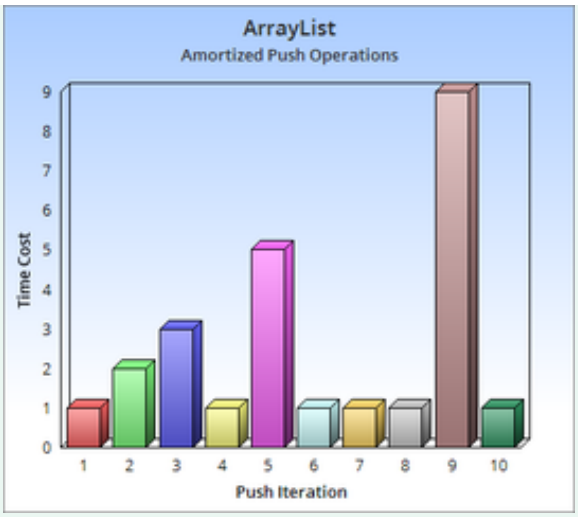
Amortization(分摊复杂度):

it is a consideration of average complexity

Consider a **dynamic array** that grows in size as more elements are added to it, such as `ArrayList` in Java or `std::vector` in C++.

If we started out with a dynamic array of size 4, we could push 4 elements onto it, and each operation would take **constant time**. Yet pushing a fifth element onto that array would take longer as the array would have to create a new array of double the current size (8), copy the old elements onto the new array, and then add the new element. The next three push operations would similarly take constant time, and then the subsequent addition would require another slow doubling of the array size.

In general if we consider an arbitrary number of pushes $n + 1$ to an array of size n, we notice that push operations take constant time except for the last one which takes $\Theta(n)$ time to perform the size doubling operation. Since there were $n + 1$ operations total we can take the average of this and find that pushing elements onto the dynamic array takes: $\frac{n\Theta(1)+\Theta(n)}{n+1} = \Theta(1)$, constant time.




```
operations = {op1 : cost1 , op2 : cost2 , .... , opn
: costn}
actual_operatons = [op1 , op1 , op3 , op6  , op3 ,...
, opn]
amortization = sum([operations[op] for op in
acutal_operations]) / len(acutal_operations)
```

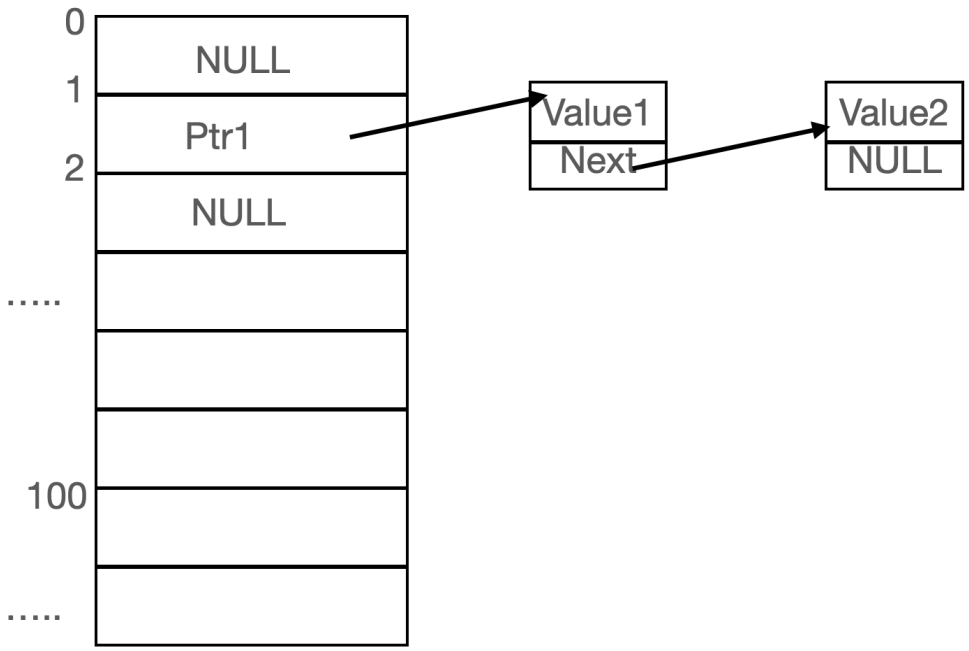
Actually, we can make the insertion/deletion operation works at a real $O(1)$ time, by moving some items to the new array along with insert a new item, then if we need to grow, we can switch to the new array immediately.

Always, always remeber, you are FREE to do any thing!!!

Approaches to collision

Chaining

- we just store collide values in a linked list



- However, in the worest case, we will have a list whose length is $O(n)$, we just build a plain linked list.
- 系统缓存机制几乎完全失效，动态申请到的内存未必连续分布

Open addressing

implement the hash table only using an array, no pointers

Idea: for each key we try to insert it into some place, but if there is already an element exist(collision happens), we compute a different hash value for it to be insert, we call this strategy ***Probing***

Insertion

- $\text{position } = hash(k, i)$, k is the key, i is the time we have tried(trial count)
  - in order to use all of the space, we require a permutation property here
  - Permutation: $\forall k, \{h(k,1), h(k,2), . . h(k,m)\} = \{0, 1, 2, . . , m-1\}$
- keep probing until an empty/"DeleteMe" slot is found, then insert the item

Searching

- keep probing until you find the key(return true) or find a "None" tag(return false), if you find a "DeleteMe" tag, you keep searching.

Deletion

- search for the target element, if it is found, mark it as "DeleteMe", otherwise return Not Found
- When you grow/shrink the table, clean all "DeleleMe"(just don't copy them into the new table)

Probing strategy

Linear probing

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

  h' Is the original hash function, m is the size of the table

- this functions satisfy the permutation property perfectly, but it will yield cluster, and the cluster keep growing.
- Once you have a cluster, the probobility of collision grows and you lose the $O(1)$ complexity for searching and inserting.

Double hashing

- $h(k,i) = h'_1(k) + h'_2(k) \cdot i \mod m$

  h'_1, h'_2 are ordinary hash functions

- if h'_2 is relative prime to m, we have the permutation
- uniform hash assumption tells us that the complexity is $O(1)$

String matching Problem

Given two string `s,t`, `s in t == True?`

This is how your `grep` and `Ctrl-F` work

simple algorithm

```
any([s == t[i:i+len(s)] for i in range(len(t) -
len(s)) ])
```

```
>>> s = "jefoiejfreg"
>>> t =
"ijeogjweprhgoqpirepotoieruogijfdnlsbmfdsjbopjisreoig
jeiorwjikew[optjjefoiejfregokfgoijewoprgjw"
>>> any([s == t[i:i+len(s)] for i in range(len(t) -
len(s)) ])
True
>>> t = "joweyrtuwoitupwpri[fdjskgmbkmslmgfgjkwpgr"
>>> any([s == t[i:i+len(s)] for i in range(len(t) -
len(s)) ])
False
```

- 

- it takes $O(|t| - |s|)$ iterations, and in each iteration, $O(|s|)$

time to compare `s == t?`, the overall complexity is $O(|t| \cdot |s|)$

algorithm using hashing

it is expensive to compare two string, but it is easy to compare two integer

However, compute the hash value for `t[i:i+len(s)]` can still be painful

Rolling hashing ADT

Note that `t[i:i+len(s)]` and `t[i+1 : i+1+len(s)]` share a lot of characters, we only need to remove the first character and append the new one.

```
from sympy import randprime
class RollingHash():
    def __init__(self , message : str):
        self.message = message
```

```
        self.p = randprime(len(message) ,
len(message)**2)

    def __prehash(self):
        return int(self.message.encode('ascii').hex()
, 16)

    def hashvalue(self):
        integer = self.__prehash()
        return integer % self.p

    def append(self , c:str):
        self.message += c

    def skip(self):
        self.message = self.message[1:]
```

- the `append` and `skip` function can be done with mathematical way

```
# modify the arithmetic according to your encoding
def append(s:str , c:str):
    sint = int(s.encode('ascii').hex() , 16)
    sint *= 2**8 # shift 8 bits to left
    sint += int(c.encode('ascii').hex() , 16)
    return str(sint.to_bytes(len(s)+1, 'big'))
print(append('hello', 'm')[2:-1])
# hellom
```

```
def skip(s:str):
    sint = int(s.encode('ascii').hex() , 16)
    sint %= 2**((len(s)-1)*8)
    return str(sint.to_bytes(len(s)-1, 'big'))
print(skip('hello')[2:-1])
# ello
```

Karp-Rabin algorithm

```
def match(s,t):
    rolling_s = RollingHash(s)
    rolling_t = RollingHash(t[:len(s)])
    for i in range(len(t) - len(s)):
        rolling_t  = RollingHash(t[i:i+len(s)])
        if rolling_s.hashvalue() ==
rolling_t.hashvalue():
            return rolling_t.message ==
rolling_s.message # beware of the collision
        else:
            rolling_t.append(t[i+len(s)])
            rolling_t.skip()
    return False

print(match('hello' , 'oiqjgojre helloworldfowqjf'))
# True
```

- the time complexity is considered to be

$$O(1) \cdot O(|t| - |s|) = O(|t| - |s|)$$

散列表/桶数组

bucket array == hash table

- `hash(key) = rand(key)`

桶排序/计数排序(bucket sorting)

- 复杂度不仅取决于数据的多少，还取决于数据的大小，$T = O(n + M)$, M为代排序数据中的最大元素
- 如果M<n，即待排序数据中存在大量重复，则有望在线性时间内完成排序
- 利用散列表排序key，每一个value都是频率计数器
- 利用频率累计值即可算出当前key所要存放的位置

```
def bucketsort(ls:list)->None:
    count = [0 for i in range(len(ls))]
    for i in ls: count[i]+=1
    accumulator = 0
    for i in range(len(ls)):
        last = accumulator
        accumulator+=count[i]
        for j in range(last , accumulator):
            ls[j] = i
```

```
from random import randint
ll = [randint(0, 5) for i in range(10)]
print(ll)
bucketsort(ll)
print(ll)
```