B06901100 EE4 ChunYu, Wei

Report

Problem Description

We all know that python has a standard container called **dictionary**, dealing with the mapping between keys and values. It is very useful when coding. The following is its operations and usage.

Insert/modify (dictionary[key] = value) : Given an key and a value, insert the key/value mapping into the container. If the key already exists, modify its value.

Get (dictionary[key]): Given an key, get the value that the key maps to in the container. If the key doesn't exist, return None.

Delete (del dictionary[key]) : Given an key, delete the key and its mapping value from the container.

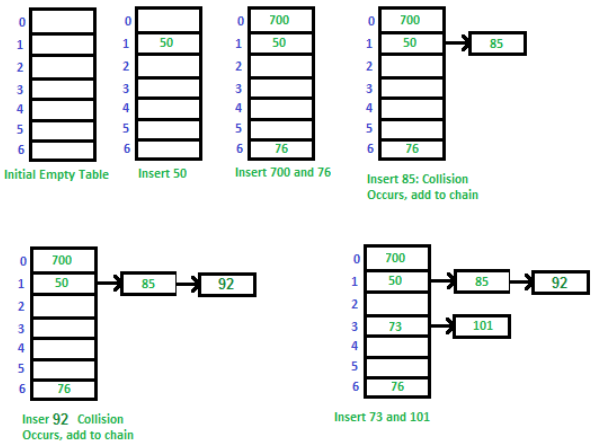
Usage example

```
dic = {}
dic["aaa"] = 1
dic["bbb"] = 3
print(dic["aaa"])      // 3
del dic["bbb"]
print(dic.get("bbb")) //None
```

Goal : I want to inplement an efficient dictionary container that supports the operations above, by using the data structures in the class. I restrict the key's type to string because it is the most general and useful data type when it comes to key of mapping.

Framework

Idea : To get a fast one-one mapping. I basically use chaining hash table. If the number of elements stored in the hash table doesn't exceed the table size too much, this type of data structure provide us with average time complexity O(1) per operation (Insert, find, delete). This data structure is really efficient corresponding to our purpose.



Methods : The chaining hash table contains several slots. Each slot stores several element by using singly linked list. Once we get a key, we use an hash function to map the key into an int as the index of slot in hash table. Then, triverse all the elements in the slot to get, insert, or delete the data related to the key. We make sure the operation finishing in O(1) time by limiting the number of elements in a slot being constant. Thus, we can fulfill efficient one-one mapping.

Data Structure Used

Hash – For mapping the keys into the index of slots

Singly linked list – For storing multiple elements in a single slot in hash table

Implementation Details

Class ListNode

Purpose

Used to construct linked list

Member data

- self.key - store the key and value in mapping
- self.value - store the key and value in mapping
- self.next - stores the next ListNode in a linked list.

```
class ListNode:
    def __init__(self, key, value, next = None):
        self.value = value
        self.key = key
        self.next = next
```

Class Map

Purpose

Manipulate the operations of dictionary container

Member data

- self.hashtable - store the hashtable with size HASHSIZE
- self.hashfunction - store the hashfunction used to hash a string into an int.

Member function

- set – set a correspond key's value in the hash table
- get – get a correspond key in the hash table
- delete – delete a correspond key in the hash table

```
class Map:
    def __init__(self, hashfunction = myhashfunction):
        self.hashtable = [None] * HASHSIZE
        self.hashfunction = hashfunction
```

def myhashfunction(x):

Purpose : Design a efficient and uniform hash function that maps string to int by myself. It takes a string x and returns a hash value (int).

Method : Get the fisrt n char of the string. Use ord() function to turn chars into ints by ASCII Code encoding. Then, calculate the value of the formula to get hash value, where COF and HASHSIZE are tuned parameters.

$$\text{hash value} = \left(\sum ord(x[i]) * COF^{i-1}\right) mod\ HASHSIZE$$

Time complexity : If we take constant char, it takes O(1) to calculate the formula

Space complexity : O(1)

def set (in class Map)

Purpose : Implement the set operation

Method : Use self.hashfunction to map input key to its slot index

```
pos = self.hashfunction(key)
curnode = self.hashtable[pos]
```

Triverse the linked list

```
while(curnode != None and curnode.key != key):
    prenode = curnode
    curnode = curnode.next
```

Create a node/Set the node's value

```
if(curnode != None):
    curnode.value = value
else:
    if(prenode == None):
        self.hashtable[pos] = self.ListNode(key, value)
    else:
        prenode.next = self.ListNode(key, value)
```

Time complexity : If we limit the number of elements being constant , it takes O(1) .

Space complexity : O(1)

def get (in class Map)

Purpose : Implement the get operation

Method

Similar to set. Return value instead of set value

```
if(curnode == None):
    return None
else:
    return curnode.value
```

Time complexity : If we limit the number of elements being constant , it takes O(1) .

Space complexity : O(1)

def delete (in class Map)

Purpose : Implement the delete operation

Method : Similar to set. Delete node instead of set value

```
while(curnode.next != None and curnode.next.key != key):
    curnode = curnode.next
if(curnode.next != None):
    tem = curnode.next
    curnode.next = curnode.next.next
    del tem
```

Time complexity : If we limit the number of elements being constant , it takes O(1) .

Space complexity : O(1)

Parameters

COF : The coefficient used in myhashfunction

HASHSIZE : The number of slots in hashtable

NUMBER OF CHAR TO HASH : The numbers of char choosed to calculate for the hash value

Choices Against Different Parameters

Experiment 1 - Number Of Chars Choosed To Hash

Control variables

# of elements inserted : 100000

HASHSIZE : 10091

COF: 500

Result

Figure1

X : the number of chars choosed to hash

Y : the variance of # elements of slots

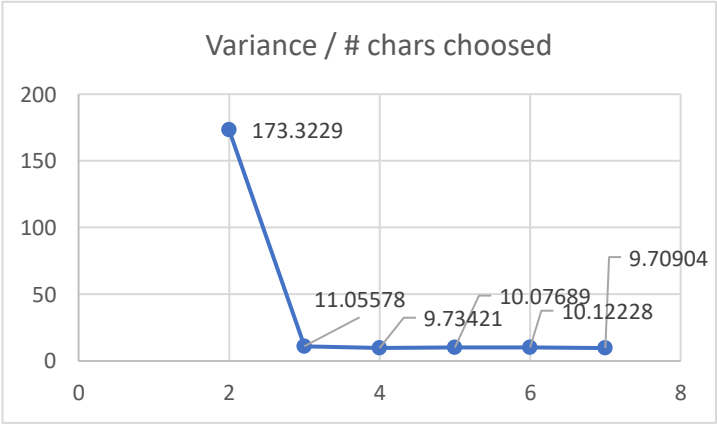
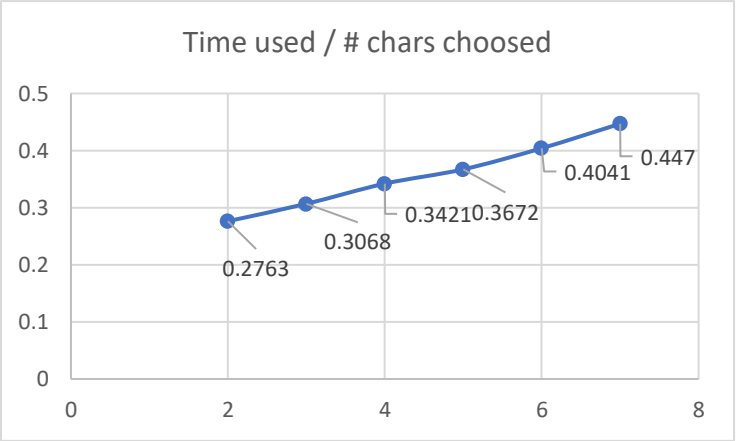


Figure 2

X : the number of chars choosed to hash

Y : time used (s)



Analysis

We evaluate the effect by calculate the variance of # element of all slots. If the variance is low, it means that the distribution of elements is more uniform.

From Figure 1, we can see that the variance getting low after x = 3. From Figure 2, we can see that the more number of chars we consider, the more time it use (with a linear relation).

4 chars get a good uniformity (better than 3 chars), and the uniformity becomes almost the same after 4. As a result, we choose to consider 4 chars to calculate the hash value.

Experiment 2 – COF

Control variables

# of elements inserted : 100000

HASHSIZE : 10091

# of chars choosed : 4

Result

Figure1

X : the COF value

Y : the variance of # elements of slots

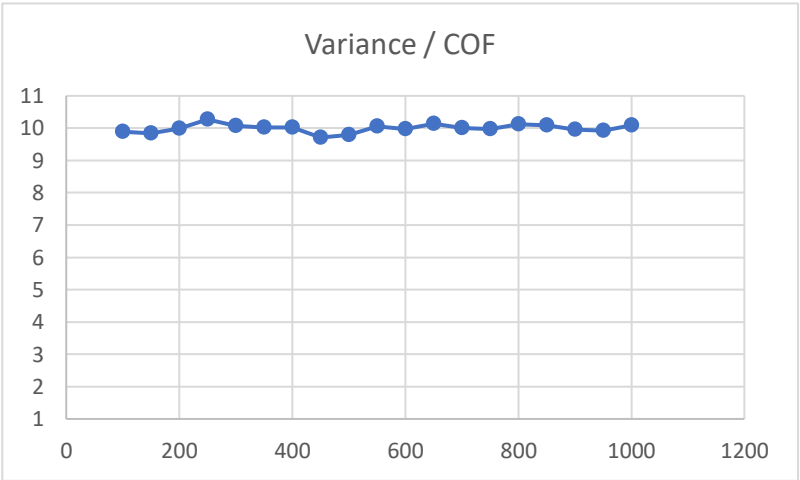
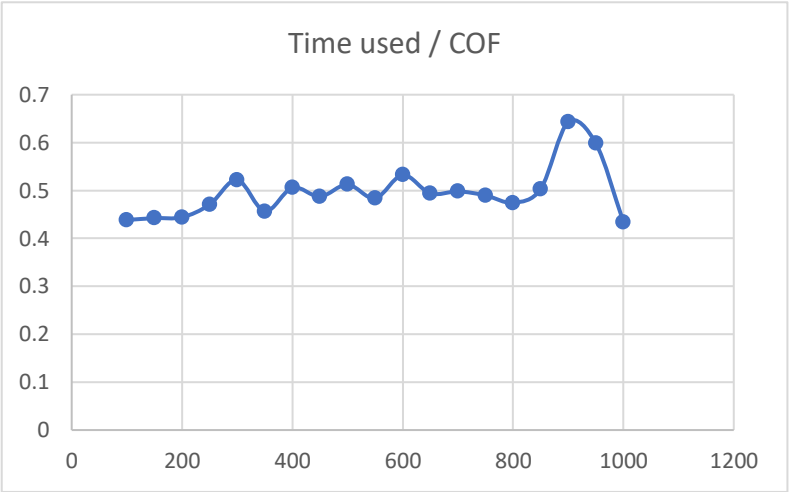


Figure 2

X : the COF value

Y : the variance of # elements of slots



Analysis : The COF value doesn't affect the effect.

Experiment 3 – HASHSIZE

Control variables

COF : 787

# of chars choosed : 4

Result

Figure 1

# of elements inserted : 100000

X : HASHSIZE ($\log_{10}$ scale)

Y : time used (sec)

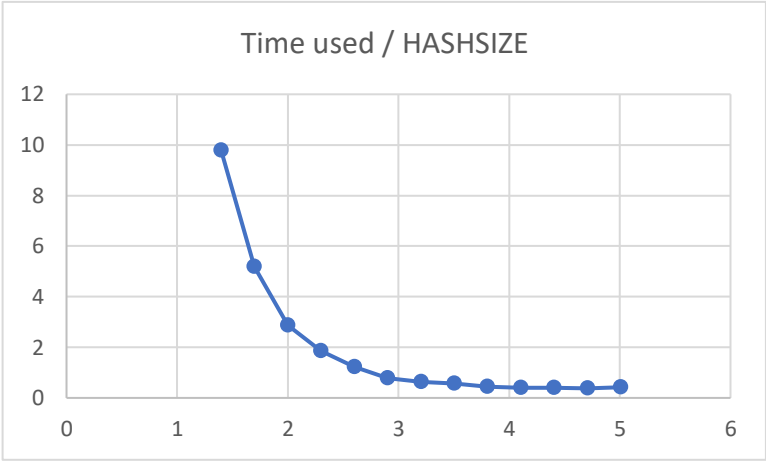
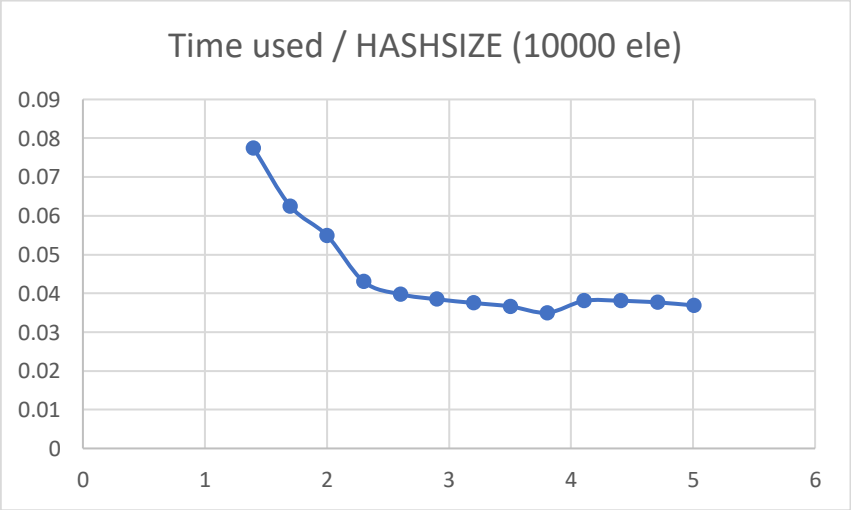


Figure 2

# of elements inserted : 10000

X : HASHSIZE ($\log_{10}$ scale)

Y : time used (sec)



Analysis

The larger HASHSIZE, the less time the hashtable cost to find a key. However, when the HASHSIZE exceed $\frac{(\#\text{ of elements inserted})}{10}$, the speed will reach saturation. As a result, I choosed the HASHSIZE to be $\frac{(\text{Expected \# of elements inserted})}{10}$.

Experiment 4 – Is prime number necessary for COF and HASHSIZE?

Control variables

# of elements inserted : 100000

# of chars choosed : 4

Experiment Result

Table 1

Column : COF

Row : HASHSIZE

Value : variance

HASHSIZE \ COF	500	787
10000	744.221	9.8318
787*14	11.18	1322.2

Table 2

HASHSIZE = 787*14 COF = 787

X : index of slot

Y : number of elements in the slot

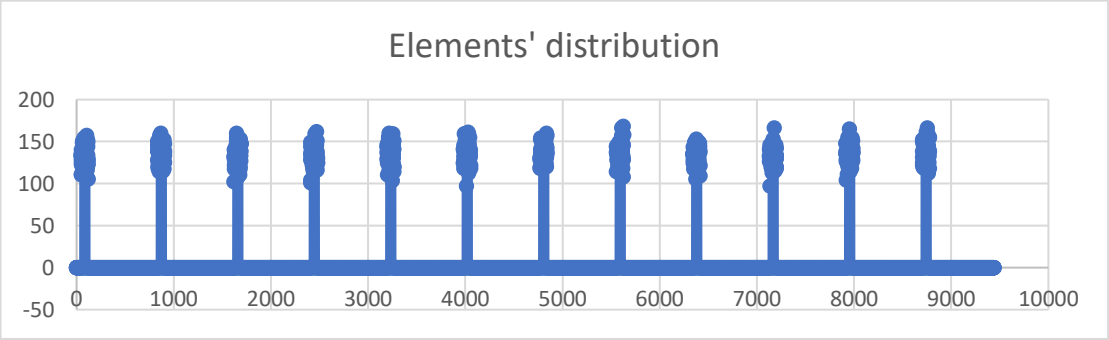
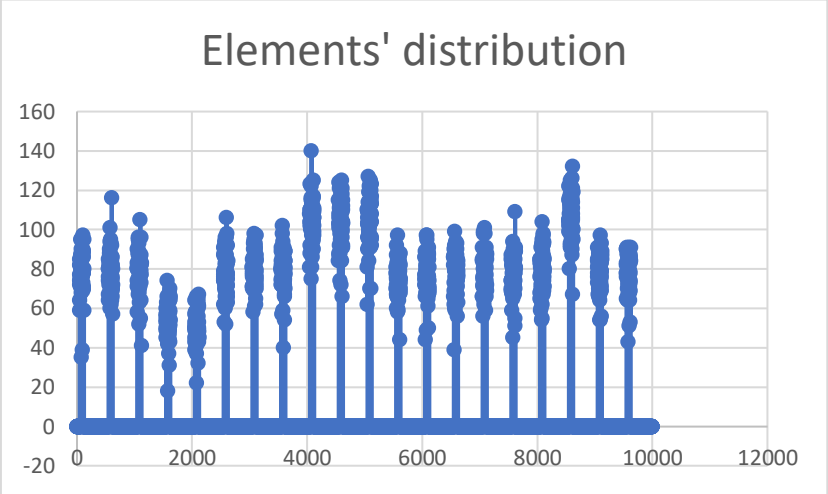


Table 3

HASHSIZE = 10000 COF = 500

X : index of slot

Y : number of elements in the slo



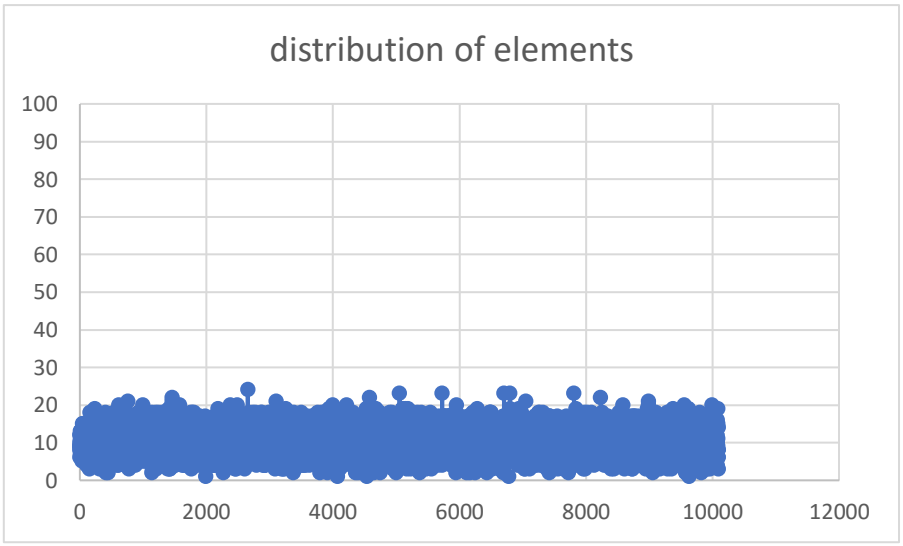
Analysis : From Table 1, we can see that the variance will be high if HASHSIZE and COF are not relatively prime. In Figure 2 and Figure 3, we can see the distribution is really scattered. As a result, I will choose both HASHSIZE and COF prime number for assurance.

Final parameters choosed

COF - 787

HASHSIZE - 10091

# of chars choosed - 4



Result

Correctness

I apply operations on both my Map and standard dictionary, check if there is diffierence between their outputs.

In the following testcases, the outputs are always equivalent. 10.in 100.in 200.in 400.in 800.in 1600.in 3200.in 6400.in 10000.in 12800.in 25600.in 51200.in 102400.in

These testcases are put under the folder /testcase.

Time&Space Complexity

# operations	Time used (s)	Memory used (MB)
100	0.00031	0.0167
200	0.00061	0.1155
400	0.001535	0.143
800	0.002263	0.1891
1600	0.004734	0.3013
3200	0.01093	0.4876
6400	0.01809	0.8609
12800	0.03676	1.6516
25600	0.0811	3.3094
51200	0.16029	6.5148
102400	0.236	7.949

We can see the time used is propotional to # of operations, so we can claim that the time complexity of every operaton is about O(1).

Compare With Standard Dictionary

Figure 1

X: number of operations

Y: Time used (s)

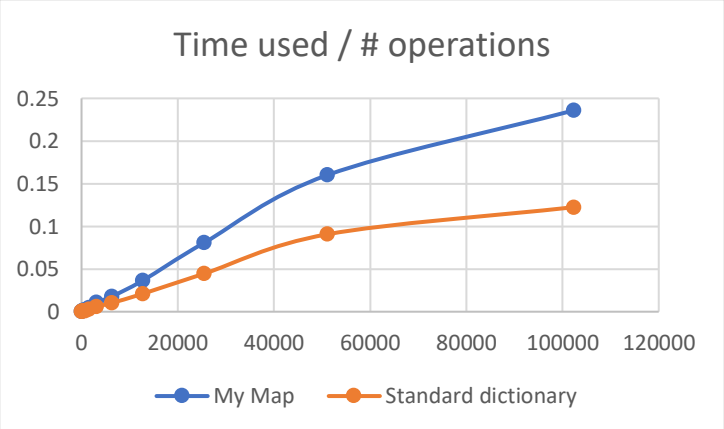


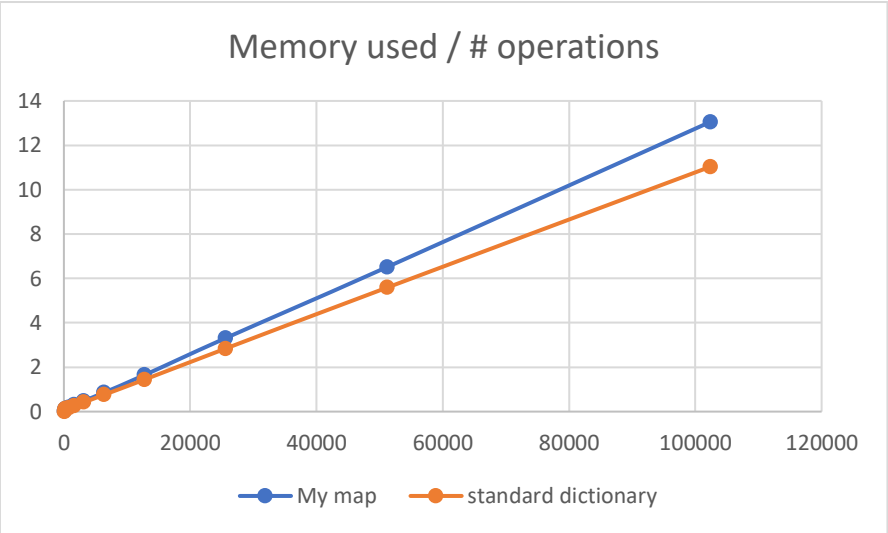
Figure 2

X: number of operations

Y: Memory used(MB)

Blue line: my implementation

Orange line: standard dictionary



Achievement

I wrapped my Map into an container that can be imported and used by users. It is efficient enough and can be extended with whatever you want. Here is the example of usage.

```
from Map import Map

map = Map()
map.set("aaa","bb")
print(map.get("aaa"))
```

Others

Reference : None

Work distribution : All done by B06901100

File Discription

*.xlsx – Experiment results

main.py – The code used to test and run experiments

Map.py – A container Map that can be imported by user

/testcase/* – testcases

genData – The program that generates testcase

test.py – The example of use Map