Dictionaries/Maps

Hash-Maps in Python

In computer science, a Hash table or a Hashmap is a type of data structure that maps keys to its value pairs (implement abstract array data types). Suppose we are given a string or a character array and asked to find the maximum occurring character. It could be quickly done using arrays.

- We can simply create an array of size 256.
- Initialize this array to zero.
- Traverse the array to increase the count of each character against its ASCII value in the frequency array.
- In this way, we will be able to figure out the maximum occurring character in the given string.

The above method will work fine for all the 256 characters whose ASCII values are known. But what if we want to store the maximum frequency string out of a given array of strings? It can't be done using a simple frequency array, as the strings do not possess any specific identification value like the ASCII values. For this, we will be using a different data structure called **hashmaps**.

In **hashmaps**, the data is stored in the form of keys against which some value is assigned. **Keys** and **values** don't need to be of the same data type.

If we consider the above example in which we were trying to find the maximum occurring string, using hashmaps, the individual strings will be regarded as keys, and the value stored against them will be considered as their respective frequency.

For example, The given string array is:

```
str[] = {"abc", "def", "ab", "abc", "def", "abc"}
```

Hashmap will look like follows:

Key (datatype : string)	Value (datatype : int)
“abc”	3
“def”	2
“ab”	1

From here, we can directly check for the frequency of each string and hence figure out the most frequent string among them all.

Note: *One more limitation of arrays is that the indices could only be whole numbers but this limitation does not hold for hashmaps.*

When it comes to Python, Hash tables are used via **dictionary** ie, the built-in data type.

Dictionaries

- A Dictionary is a Python’s implementation of a data structure that is more generally known as an associative array.
- A dictionary is an unordered collection of key-value pairs.
- Indexing in a dictionary is done using these **“keys”.**
- Each pair maps the key to its value.
- Literals of the type **dictionary** are enclosed within curly brackets.
- Within these brackets, each entry is written as a key followed by a colon ” : ”, which is further followed by a value. This is the value that corresponds to the given key.
- These keys must be unique and cannot be any immutable data type. **Eg-** string, integer, tuples, etc. They are always mutable.
- The values need not be unique. They can be repetitive and can be of any data type (Both mutable and immutable)
- Dictionaries are mutable, which means that the key-value pairs can be changed.

What does a dictionary look like?

This is the general representation of a dictionary. Multiple key-value pairs are enclosed within curly brackets, separated by commas.

```
myDictionary = {
    <key>: <value>,
    <key>: <value>,
      .
      .
    <key>: <value>
}
```

Creating A Dictionary

Way 1- The Basic Approach

Simply put pairs of key-value within curly brackets. Keys are separated from their corresponding values by a colon. Different key-value pairs are separated from each other by commas. Assign this to a variable.

```
myDict= {"red":"boy", 6: 4, "name":"boy"}
months= {1:"January", 2:"February", 3: "March"}
```

Way 2- Type Casting

This way involves type casting a list into a dictionary. To typecast, there are a few mandates to be followed by the given list.

- The list must contain only tuples.
- These tuples must be of length **2**.
- The first element in these tuples must be the key and the second element is the corresponding value.

This type casting is done using the keyword **`dict`.**

```
>>> myList= [("a",1),("b",2),("c",2)]
>>> myDictionary= dict(myList)
>>> print(myDictionary)
{"a":1,"b":2,"c":2}
```

Way 3- Using inbuilt method `.fromkeys()`

This method is particularly useful if we want to create a dictionary with variable keys and all the keys must have the same value. The values corresponding to all the keys are exactly the same. This is done as follows:

```
>>> d1= dict.fromkeys(["abc",1,"two"])
>>> print(d1)
{"abc":None ,1: None, "two": None}
# All the values are initialized to None if we provide only one argument
i.e. a list of all keys.
```

```
# We can initialise all the values to a custom value too. This is done by
providing the second argument as the desired value.
>>> d2= dict.fromkeys(["abc",1,"two"],6)
>>> print(d2)
["abc":6 ,1:6, "two":6]
```

How to access elements in a dictionary?

We already know that the indexing in a dictionary is done with the keys from the various key-value pairs present within. Thus to access any value we need to use its index i.e. it's **key**.

Similar to the list and tuples, this can be done by the square bracket operator **[ ]**.

```
foo= {"a":1,"b":2,"c":3}
print(foo["a"])
--> 1
print(foo["c"])
--> 3
```

If we want the value corresponding to any key, we can even use an inbuilt dictionary method called **`get`**.

```
>>> foo= {"a":1,"b":2,"c":3}
```

```
>>> print(foo.get("a"))
1
>>> print(foo.get("c"))
3
```

A very unique feature about this method is that , in case the desired **key** is not present in the dictionary , it won't throw an error or an exception. It would simply return `None`.

We can make use of this feature in another way. Say we want the method to do the following action: If the key is present in the dictionary then return the value corresponding to the key. In case the key is not present, return a custom desired value (say 0).

This can be done as follows:

```
>>> foo= {"a":1,"b":2,"c":3}
>>> print(foo.get("a",0))
1
>>> print(foo.get("d",0))
0
```

Accessing all the available keys in the dictionary:

This can be done using the method `.keys()`. This method returns all the different keys present in the dictionary in the form of a list.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.keys()
dict_keys(["a","b","c"])
```

Accessing all the available values in the dictionary:

This can be done using the method `.values()`. This method returns all the different values present in the dictionary in the form of a list.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.values()
dict_values([1,2,3])
```

Accessing all the available items in the dictionary:

This can be done using the method `.items()`. This method returns all the different items (key-value pairs) present in the dictionary in the form of a list of tuples, with the first

element of the tuple as the key and the second element as the value corresponding to this key.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.items()
dict_items([("a",1),("b",2),("c",3)])
```

Checking if the dictionary contains a given key:

The keyword used is `in`. We can easily get a boolean output using this keyword. It returns True if the dictionary contains the given **key**, else the output is False. This checks the presence of the keys and not the presence of the values.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> "a" in foo
True
>>> 1 in foo
False
```

Iterating over a Dictionary:

To traverse through the dictionary, we can use a simple for loop. The loop will go through the keys of the dictionary one by one and do the required action.

```
bar= {2:1,3:2,4:3}
for t in bar:
      print(t)
Out[]:
2
3
4
# Here t is the key in the dictionary and hence when we print t in all
iterations then all the keys are printed.
```

```
for t in bar:
      print(t, bar[t])
Out[]:
2 1
3 2
```

```
4 3
# Here along with the keys, the values which are bar[t] are printed. In
this loop, the values are accessed using the keys.
```

```
for key,item in dict.items():
      print(key, item)
Out[]:
2 1
3 2
4 3
# Here along with the keys, the values are printed.
```

Adding Elements In a Dictionary

Since a dictionary is mutable, we can add or delete entries from the dictionary. This is particularly useful if we have to maintain a dynamic data structure. To assign a value corresponding to a given key *(This includes over-writing the value present in the key or adding a new key-value pair)*, we can use the square bracket operators to simply assign the value to a key.

If we want to update the value of an already existing key in the dictionary then we can simply assign the new value to the given key. This is done as follows:

```
>>> bar= {2:1,3:2,4:3}
>>> bar[3]=4
# This operation updates the value of the key 3 to a new value i.e. 4.
>>> print(bar)
{2:1,3:4,4:3}
```

Now if we want to add a new key-value pair to our dictionary, then we can make a similar assignment. If we have to add a key-value pair as `"man":"boy"`, then we can make the assignment as:

```
>>> bar["man"]="boy"
>>> print(bar)
{2:1,3:2,4:3,"man":"boy"}
```

Adding or concatenation of two dictionaries:

If we have 2 dictionaries and we want to merge the contents of both the dictionaries and form a single dictionary out of it . It is done as follows:

```
a= {1:2,2:3,3:4}
b= {7:2,10:3,6:4}
a.update(b)
print(a)
--> {1:2,2:3,3:4,7:2,10:3,6:4}
```

In this process, the second dictionary is unchanged and the contents of the second dictionary are copied into the first dictionary. The uncommon keys from the second dictionary are added to the first with their corresponding values. However, if these dictionaries have any common key, then the value of the common key present in the first dictionary is updated to the new value from the second.

Deleting an entry:

To delete an entry corresponding to any key in a dictionary, we can simply pop the key from the dictionary. The method used here is `.pop()`. This method removes the key-value pair corresponding to any particular key and then returns the value of the removed key-value pair.

```
>>> c={1:2,2:(3,23,3),3:4}
>>> c.pop(2)
(3,23,3)
```

Deleting all the entries from the dictionary:

If we want to clear all the key-value pairs from the given dictionary and thus convert it into an empty dictionary we can use the `.clear()` method.

```
>>> c={1:2,2:(3,23,3),3:4}
>>> c.clear()
>>> print(c)
{}
```

Deleting the entire dictionary:

We can even delete the entire dictionary from the memory by using the `del` keyword. This would remove the presence of the dictionary. This is similar to tuples and lists.

Problem statement: Print all words with frequency k.

Approach to be followed:

First, we convert the given string of words into a list containing all the words individually. Some of these words are repetitive and to find all the words with a specific frequency, we convert this list into a dictionary with all the unique words as keys and their frequencies or the number of times they occur as their values.

To convert the string to a list, we use the `.split()` function. This gives us a list of words. Now, we run a loop through this list and keep making changes to the frequency in the dictionary. If the word in the current iteration already exists in the dictionary as a key, then we simply increase the value(or frequency) by 1. If the key does not exist, we create a new key-value pair with value as 1.

Now we have a dictionary with unique keys and their respective frequencies. Now we run another loop to print the keys with the frequency '**k**'.

Given below is a function that serves this purpose.

```
def printKFreqWords(string, k):
      # Converting the input string to a list
      myList= string.split()
      # Initialise an empty dictionary
      dict= {}
      # Iterate through the list in order to find frequency
      for i in myList:
            dict[i]=dict[i]+1
      else:
```

```
            dict[i]=1
      # Loop for printing the keys with frequency as k
      for t in dict:
            if dict[t]==k:
                  print(t)
```

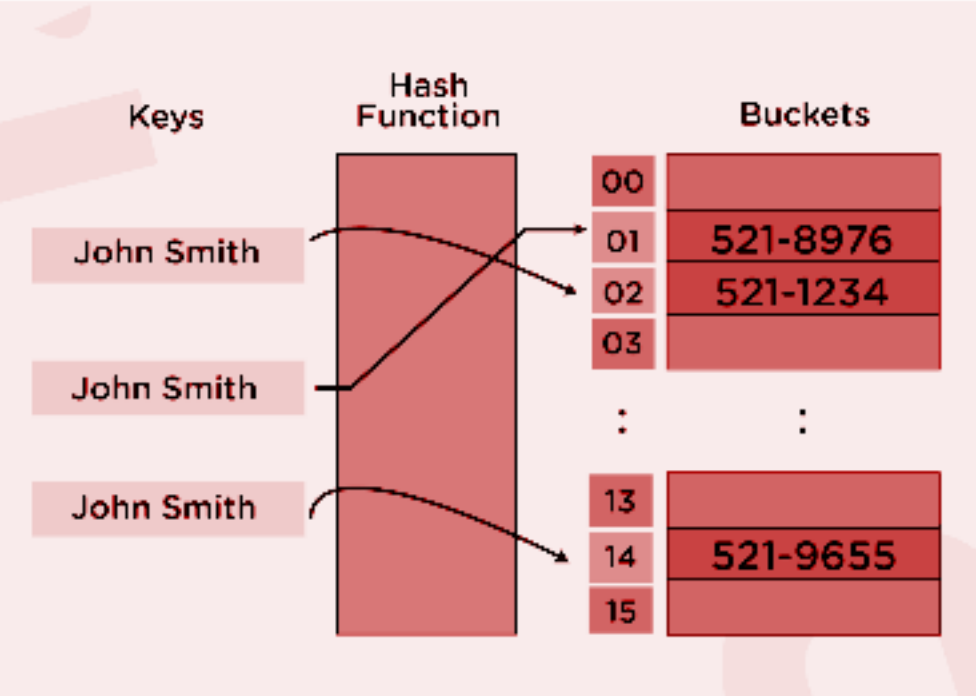
Implementing Our Own Hashmap

Bucket array and Hash function

Now, let's see how to perform insertion, deletion, and search operations using hash tables. Till now, we have seen that arrays are the fastest way to extract data as compared to other data structures as the time complexity of accessing the data in the array is O(1). So we will try to use them in implementing the hashmaps.

Now, we want to store the key-value pairs in an array, named **bucket array**. We need an integer corresponding to the key so that we can keep it in the bucket array. To do so, we use a **hash function**. A hash function converts the key into an integer, which acts as the index for storing the key in the array.

For example: Suppose, we want to store some names from the contact list in the hash table, check out the following image:



Suppose we want to store a string in a hash table, and after passing the string through the hash function, the integer we obtain is equal to 10593, but the bucket array's size is only 20. So, we can't store that string in the array as 10593, as this index does not exist in the array of size 20.

To overcome this problem, we will divide the hashmap into two parts:

- Hash code
- Compression function

The first step to store a value into the bucket array is to convert the key into an integer (this could be any integer irrespective of the size of the bucket array). This part is achieved by using a **hashcode**. For different types of keys, we will be having different kinds of hash codes. Now we will pass this value through the compression function, which will convert that value within the range of our bucket array's size. Now, we can directly store that key against the index obtained after passing through the compression function.

The compression function can be used as (`%bucket_size`).

One example of a hash code could be: (Example input: `"abcd"`)

```
"abcd" = ('a' * p^3) + ('b' * p^2) + ('c' * p^1) + ('d' * p^0)
#Where p is generally taken as a prime number so that they are well distributed.
```

But, there is still a possibility that after passing the key through from hash code, when we give the same through the compression function, we can get the same values of indices. For example, let s1 = “ab” and s2 = “cd”. Now using the above hash function for p = 2, h1 = 292 and h2 = 298. Let the bucket size be equal to 2. Now, if we pass the hash codes through the compression function, we will get:

```
Compression_function1 = 292 % 2 = 0
Compression_function2 = 298 % 2 = 0
```

This means they both lead to the same index 0.

This is known as a **collision**.

Collision Handling

We can handle collisions in two ways:

- Closed hashing (or closed addressing)
- Open addressing

In closed hashing, each entry of the array will be a linked list. This means it should be able to store every value that corresponds to this index. The array position holds the address to the head of the linked list, and we can traverse the linked list by using the head pointer for the same and add the new element at the end of that linked list. This is also known as **separate chaining**.
On the other hand, in open addressing, we will check for the index in the bucket array if it is empty or not. If it is empty, then we will directly insert the key-value pair

over that index. If not, then will we find an alternate position for the same. To find the alternate position, we can use the following:

$h_i(a) = hf(a) + f(i)$

Where **hf(a)** is the original hash function, and **f(i)** is the **i^{th}** try over the hash function to obtain the final position **$h_i(a)$**.

To figure out this **f(i)**, the following are some of the techniques:

1. **Linear probing:** In this method, we will linearly probe to the next slot until we find the empty index. Here, `f(i) = i`.
2. **Quadratic probing:** As the name suggests, we will look for alternate **i^2** positions ahead of the filled ones, i.e., **$f(i) = i^2$**.
3. **Double hashing:** According to this method, `f(i) = i * H(a)`, where H(a) is some other hash function.

In practice, we generally prefer to use **separate chaining** over **open addressing**, as it is easier to implement and is also more efficient.

Let's now implement the hashmap of our own.

Hashmap Implementation - Insert

As discussed earlier, we will be implementing separate chaining. We will be using value as a template and key as a string as we are required to find the hash code for the key. Taking the key as a template will make it difficult to convert it using hash code.

Let's look at the code for the same.

```
class MapNode:
      def __init__  (self,key,value):
            self.key = key #to store key
            self.value = value #to store value
```

```
            self.next = None #to the next pointer

class Map:

      def  __init__ (self):
            self.bucketSize = 10#Buckets for compression function
            # Store the head pointers
            self.buckets = [None for i in range(self.bucketSize)]
            self.count = 0 #To store the size

      def size(self):#Return the size of the map
            return self.count

      def getBucketIndex(self,hc):#Index using hash function
            return (abs(hc)%(self.bucketSize))

     def insert(self,key,value):
            hc = hash(key)
            index = self.getBucketIndex(hc)
            head = self.buckets[index]
            while head is not None:
                  if head.key == key:
                        head.value = value
                        return
                  head = head.next

            newNode = MapNode(key,value)
            newNode.next  = head
            self.buckets[index] = newNode
            self.count+=1
```

NOTE:

- The **hash()** method returns the hash value of an object if it has one. Hash values are just integers that are used to compare dictionary keys during a dictionary lookup quickly.

- Internally, **hash()** method calls **__hash__()** method of an object which is set by default for any object.
- The syntax of **hash()** method is:

```
hash(object)
```

Hashmap Implementation - Search and Remove

Go through the given implementation of searching for a key in a map and removing it:

```
def remove(self,key):
      hc = hash (key) #Getting hashcode
      index= self.getBucketIndex(hc) #Finding index corresponding to hc
      head = self.buckets[index] #Finding head pointer
      prev= None
      while head is not None:
            if head.key == key:#If found
                  if prev is None: #Removing
                        self.buckets[index] = head.next
                  else:
                        prev.next= head.next
                  return head.value
            prev =head
            head =head.next
      return None
```

Time Complexity and Load Factor

Let's define a few specific terms before moving forward:

1. **n** = Number of entries in our map.
2. **l** = length of the word (in case of strings)

3. **b** = number of buckets. On average, each box contains **(n/b)** entries. This is known as **load factor** (This means **b** boxes contain **n** entries). We also need to ensure that the load factor is always less than 0.7, i.e., **`(n/b) < 0.7(This will ensure that each bucket does not contain too many entries in it).`**
4. To make sure that load factor < 0.7, we can't reduce the number of entries, but we can increase the bucket size comparatively to maintain the ratio. This process is known as **Rehashing**.

This ensures that time complexity is on an average **O(1)** for insertion, deletion, and search operations each.

Rehashing

Now, we will try to implement the rehashing in our map. After inserting each element into the map, we will check the load factor. If the load factor's value is greater than 0.7, then we will rehash. Refer to the code below for better understanding.

```
def rehash(self):
      temp= self.buckets #To store the old bucket
      self.buckets = [None for i in range(2*self.bucketSize))
      self.bucketSize = 2*self.bucketSize #doubling the size
      self.count =0
      for head in temp:#inserting each value of old bucket to new one
            while head is not None:
                  self.insert(head.key,head.value)
                  head = head.next
```

```
      def insert(self,key,value):#Modify the insert function

            hc = hash(key)
            index = self.getBucketIndex(hc)
            head = self.buckets[index]
            while head is not None:
```

```
                if head.key == key:
                        head.value = value
                        return
                head = head.next

        newNode = MapNode(key,value)
        newNode.next  = head
        self.buckets[index] = newNode
        self.count+=1
        # Now we will check the load factor after insertion.
        loadFactor = self.count/self.bucketSize
        if loadFactor>=0.7:
                self.refash()
```

Practice problems:

- https://www.codechef.com/problems/STEM
- https://codeforces.com/problemset/problem/525/A
- https://www.spoj.com/problems/ADACLEAN/