Hash-mapping with Open Addressing Collision Handling

CSX 3003 Data Structures & Algorithms

Section: 541

Group Members

Lynn Thit Nyi Nyi	ID: 6411271
Tanat Arora	ID: 6410381
Aung Cham Myae	ID: 6411325

Lecturer

Prof. Thitipong Tanprasert

What is hashing, and is it important?

Hashing is a process of mapping keys and values into a hash table, based on a single hash function. A hash table is a data structure that can be implemented in order to store data in a key-value format with instant direct access to all of its items in constant time. For each key that exists within the hash table, data occurs only once at most, meaning that the hash tables are *associative*, allowing things such as dictionaries or contact records to be easily implemented with direct access to store, retrieve, and delete data uniquely based on their individually unique key.

Hash tables are one of the most crucial data structures everyone in the computer science or IT field should master, since it single-handedly allows one to get a taste of algorithm design and understand that there's always more than one possible way to further improve and speed up your program. Hashing is undoubtedly one of the most helpful algorithms, foundational to standard tools and techniques behind caching and data indexing due to its remarkable speed and efficiency.

Why use hash tables?

The biggest strength hash tables have over most abstract data structures is its incredible speed to perform insertion, deletion, and search operations, at a time complexity of only O(1), the worst case taking a time complexity of O(n) which is extremely rare.

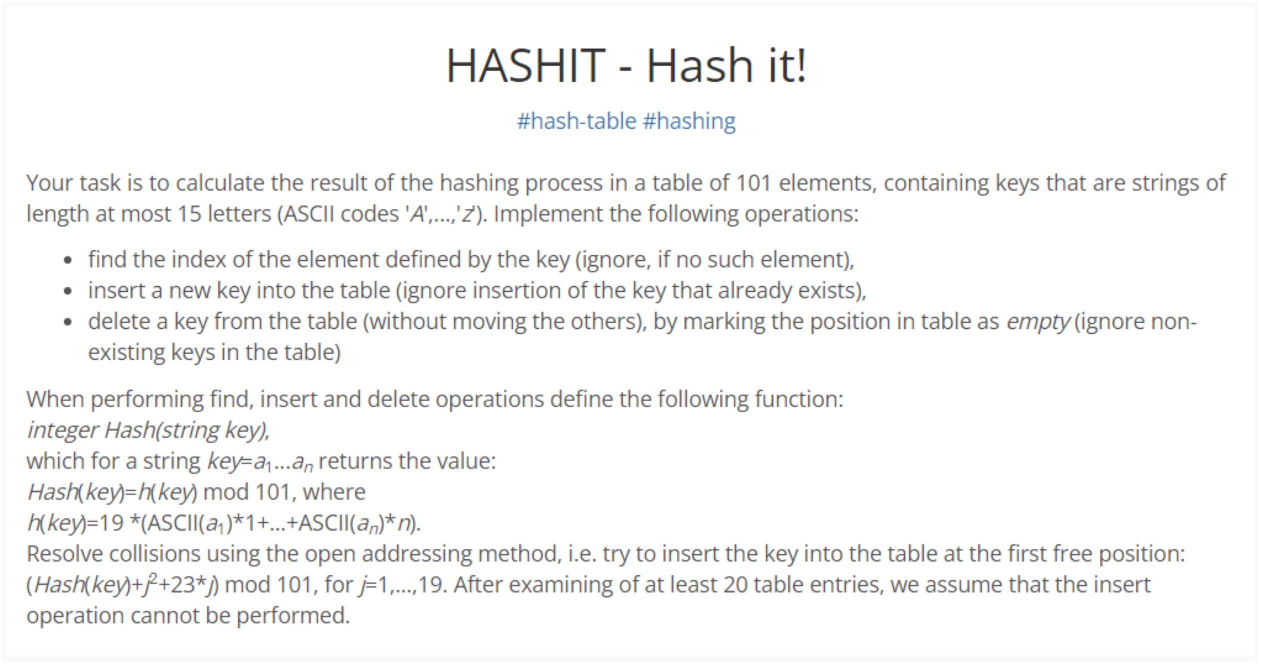
Open Addressing for Collision Handling

A collision occurs in an event when two or more keys that have the same hash value are present due to their hash function, competing for a single hash table slot that is available.

There are several methods to solve this issue, which is widely also known as collision handling. Two of the most popular collision handling methods are separate chaining, and open addressing. In this report, we will be focusing more on open addressing.

In open addressing, all elements are stored in the hash table itself, entirely based upon probing. It is also important to note that the size of the table must be greater than or equal to the total number of keys.
The example for open addressing are shown in test cases are the latter pages.

Problem Solving with Solution



HASHIT - Hash it!

#hash-table #hashing

Your task is to calculate the result of the hashing process in a table of 101 elements, containing keys that are strings of length at most 15 letters (ASCII codes '*A*',...,'*z*'). Implement the following operations:

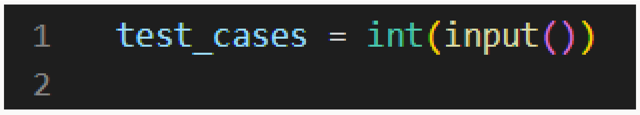
- find the index of the element defined by the key (ignore, if no such element),
- insert a new key into the table (ignore insertion of the key that already exists),
- delete a key from the table (without moving the others), by marking the position in table as *empty* (ignore non-existing keys in the table)

When performing find, insert and delete operations define the following function:
integer Hash(string key),
which for a string $key=a_1...a_n$ returns the value:
$Hash(key)=h(key)$ mod 101, where
$h(key)=19$ *(ASCII(a_1)*1+...+ASCII(a_n)*n).
Resolve collisions using the open addressing method, i.e. try to insert the key into the table at the first free position: ($Hash(key)+j^2+23*j$) mod 101, for j=1,...,19. After examining of at least 20 table entries, we assume that the insert operation cannot be performed.

https://www.spoj.com/problems/HASHIT/

And now, we will be explaining the process of hash mapping, step by step, while resolving collisions using the open addressing scheme.

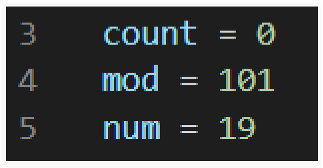
First of all, we start by accepting and storing input data (very simple):



```
test_cases = int(input())

```

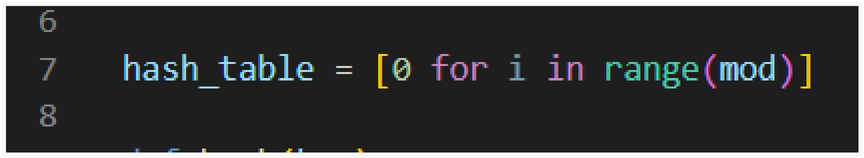
Then, we create variables for count, mod, and num as shown below; assigning them integer values according to the question, which in this case, count is initialized as zero, mod = 101, and num = 19. Note that the variable mod is the size of our hash table.



```
3   count = 0
4   mod = 101
5   num = 19
```

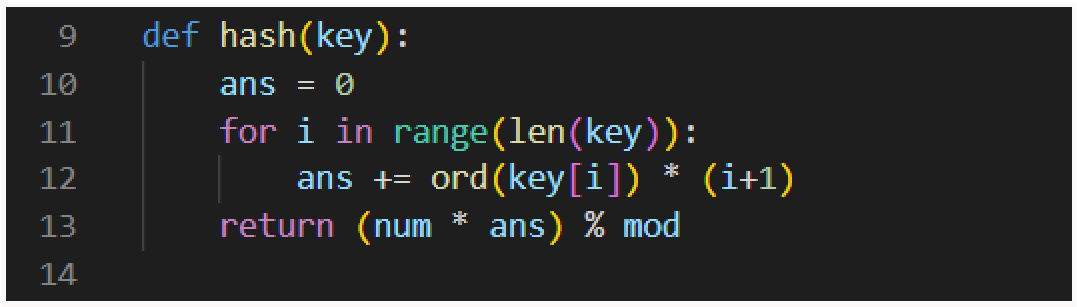
These values will be later used in our hash function; it is just there to keep the code tidy and organized.

Next, we create our hash table:



```
6
7   hash_table = [0 for i in range(mod)]
8
```

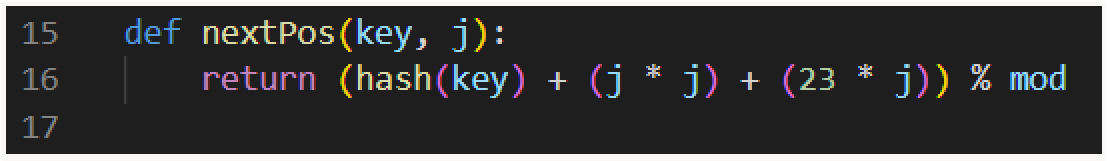
And now, we create our hash function, according to the question:



```
 9  def hash(key):
10      ans = 0
11      for i in range(len(key)):
12          ans += ord(key[i]) * (i+1)
13      return (num * ans) % mod
14
```

As seen in Line 12, we used ord function on each character of the string by looping through it to get the ASCII value, calculating and adding up as instructed in the question.

And then, we create a new function, not seen in hashing using separate chaining for collision handling, which we will call as nextPos:



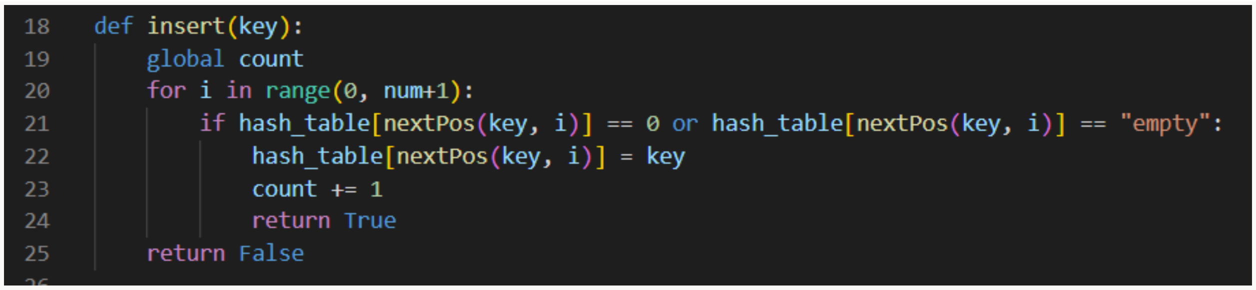
```
15  def nextPos(key, j):
16      return (hash(key) + (j * j) + (23 * j)) % mod
17
```

What this function does is that it simply allows us to jump from one slot to another in the hash table. The formulas used here are in accordance with the question. What makes this simple code so magical is how we implemented hash within nextPos itself. Therefore, when we call nextPos, the hash value is already automatically calculated.

Moreover, it will allow us to directly do implementations such as hash_table[nextPos(key, i)]. This will be seen in the later parts of our code.

Now we bring in the big guns, the three main functions essential in hash mapping; insert, exist (search), and delete.

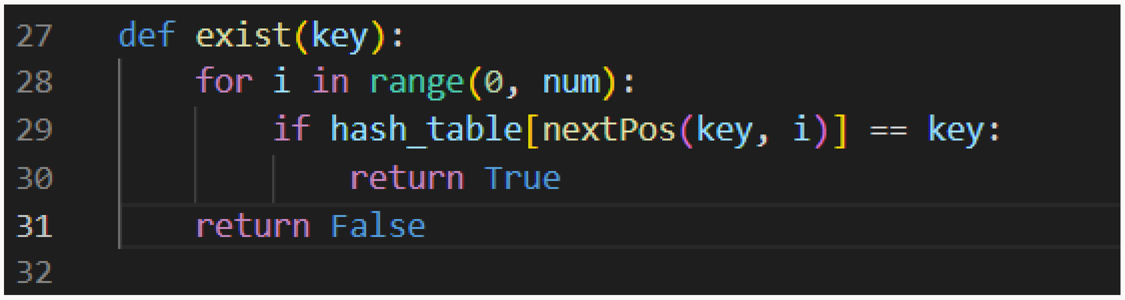
We will start with the insert function first. The code for it is as shown:



```
18  def insert(key):
19      global count
20      for i in range(0, num+1):
21          if hash_table[nextPos(key, i)] == 0 or hash_table[nextPos(key, i)] == "empty":
22              hash_table[nextPos(key, i)] = key
23              count += 1
24              return True
25      return False
```

Since the question instructed us to keep track of count, we will create a global variable of count here. Then, we will proceed to loop through 0 to num+1 (num=19), checking whether there is any free slot for the key to be inserted into; once found, the key will be *inserted* as demonstrated in Line 22, count will be incremented by 1, and we return True breaking the loop. If there is no available slot after 20 loops, we will conclude that the insertion has failed and return False.

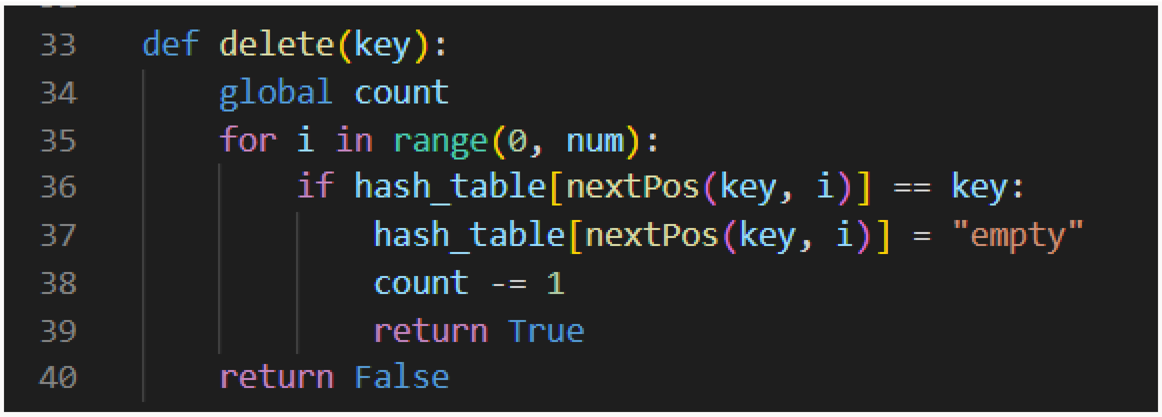
Now, we move on to the exist function, it's very self-explanatory The code is as show:



```
def exist(key):
    for i in range(0, num):
        if hash_table[nextPos(key, i)] == key:
            return True
    return False
```

Since this is very simple, we won't be talking much about it. This function returns True if the key exists in the hash table, and returns False if otherwise.

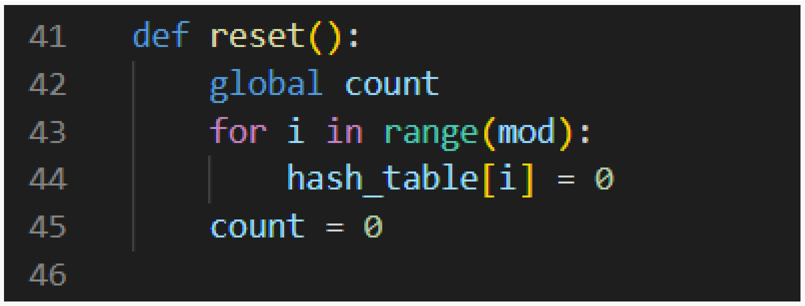
Now, for the delete function. This is what the code looks like:



```
def delete(key):
    global count
    for i in range(0, num):
        if hash_table[nextPos(key, i)] == key:
            hash_table[nextPos(key, i)] = "empty"
            count -= 1
            return True
    return False
```

Here, again, we create a global variable for count since we were instructed to keep count of it.

After all this, we still need one final touch to finish setting up all the functions, the reset function. This is how it looks like:

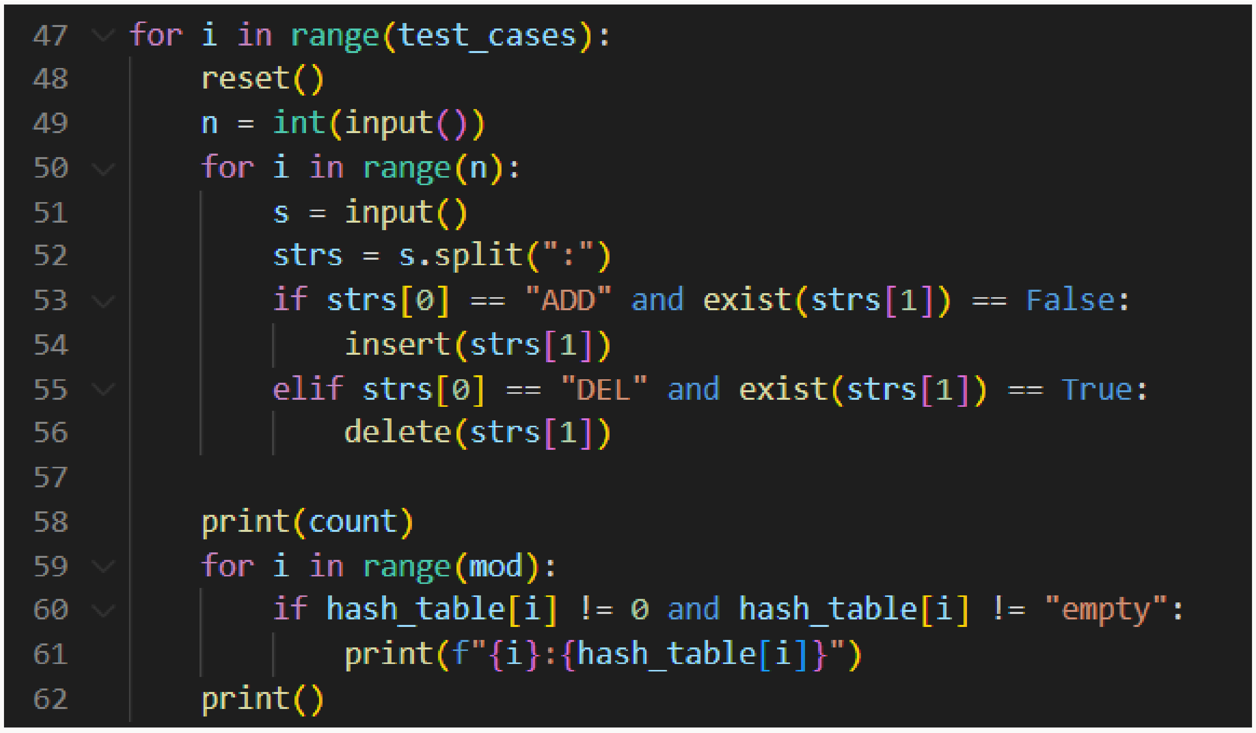


```
def reset():
    global count
    for i in range(mod):
        hash_table[i] = 0
    count = 0

```

Here, again, we create a global count. Then, looping from 0 to mod (mod=101), we set every single value in the hash table and count to zero. Straightforward!

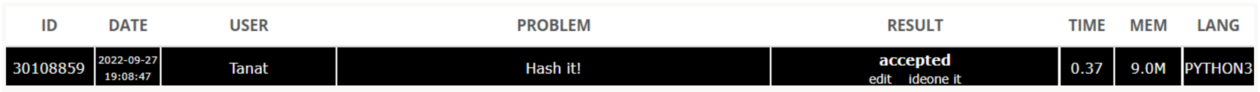
Now for the main, the driver:



```
47 for i in range(test_cases):
48     reset()
49     n = int(input())
50     for i in range(n):
51         s = input()
52         strs = s.split(":")
53         if strs[0] == "ADD" and exist(strs[1]) == False:
54             insert(strs[1])
55         elif strs[0] == "DEL" and exist(strs[1]) == True:
56             delete(strs[1])
57
58     print(count)
59     for i in range(mod):
60         if hash_table[i] != 0 and hash_table[i] != "empty":
61             print(f"{i}:{hash_table[i]}")
62     print()
```

First, we start by looping through the number of test cases input by the user previously. Then, we will use the reset function, cleaning up the hash table with every loop. After that, we will ask the user input for the number of command inputs, stored the value in n (Line 49). Moving on, we create a variable ‘s’ to store each command from the user (string inputs) and we proceed to do basic string manipulations to be able to process and interpret each input command. Using the 'split' function to split the input taken with ':', we then store it as a list in the strs variable. Then, we check if the first part of strs is 'ADD' and whether or not the key exists in the table. After that, we use the insert function to simply insert the key. In other cases where it is 'DEL' and the key exists in the table then we use the delete function to delete the key. Simple and straightforward! After all that, we apply the final touch to our entire code, and print out count, and then all the keys present in the hash table, alongside its respective indexes in the hashtable.

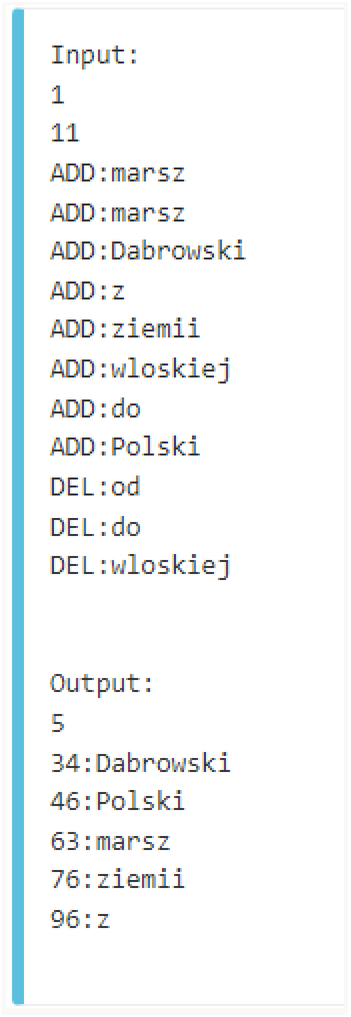
We have successfully solved the problem! We are very proud of how clean, organized, and straightforward our solution was in the end.



ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
30108859	2022-09-27 19:08:47	Tanat	Hash it!	accepted edit ideone it	0.37	9.0M	PYTHON3

Test Cases

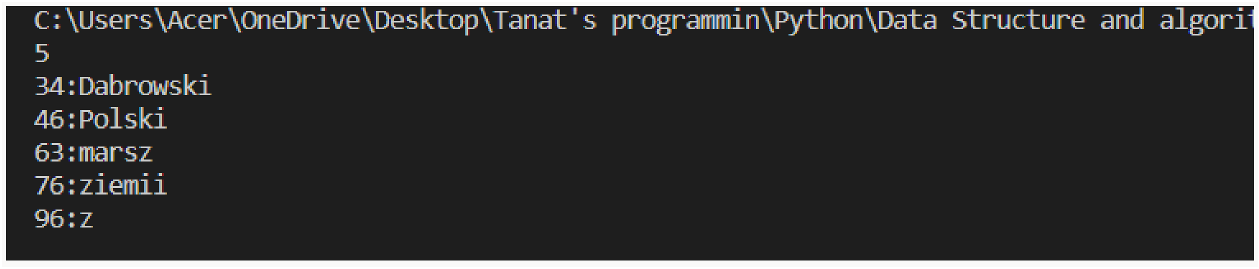
We got the test cases to test out program from https://www.spoj.com/problems/HASHIT/ There's one test case provided by the question itself and the other test cases are from the comment section of the website.



```
Input:
1
11
ADD:marsz
ADD:marsz
ADD:Dabrowski
ADD:z
ADD:ziemii
ADD:wloskiej
ADD:do
ADD:Polski
DEL:od
DEL:do
DEL:wloskiej


Output:
5
34:Dabrowski
46:Polski
63:marsz
76:ziemii
96:z
```

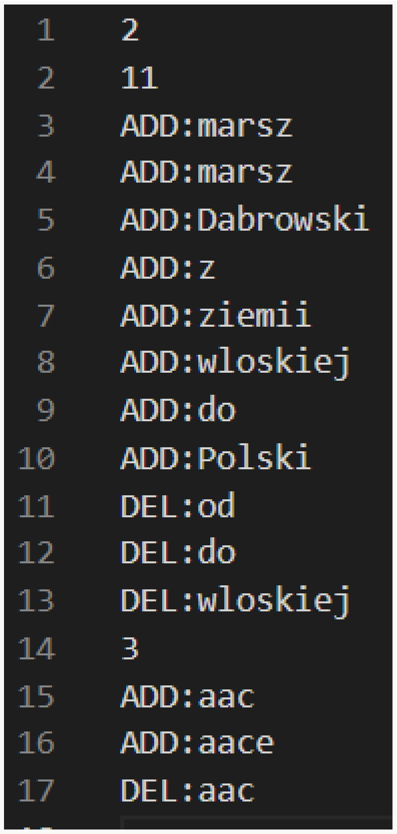
This is a test case provided by the question. As you can see theres Input given and an expected output.



```
C:\Users\Acer\OneDrive\Desktop\Tanat's programmin\Python\Data Structure and algori
5
34:Dabrowski
46:Polski
63:marsz
76:ziemii
96:z
```

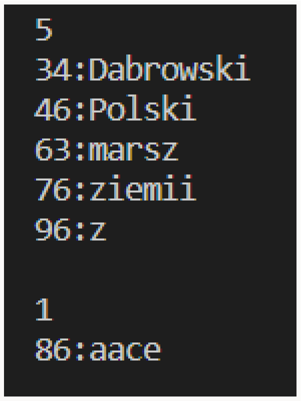
The process for getting the output is each time we add the key into the table, count is incremented by 1 and if the key already exists in the table then it will skip that key. In this case we can see that the key ‘marsz’ is recognized only once. Each time we delete a key from a table, count is decremented by 1.
So in this case marsz, dabrowski, ziemii, z, wloskiej, do and polski got added into the table which result in count having a value of 7, then we delete od, do, wloskiej. Note that od does not exist in the table so we ignore it and delete the other two from the table which result in count value of 5.

Some other input from the comment sections.



```
2
11
ADD:marsz
ADD:marsz
ADD:Dabrowski
ADD:z
ADD:ziemii
ADD:wloskiej
ADD:do
ADD:Polski
DEL:od
DEL:do
DEL:wloskiej
3
ADD:aac
ADD:aace
DEL:aac
```

This input is really good to test one condition of our program as it have two different test cases we need to go through. First case containing 11 keys and second case contain 3 keys.

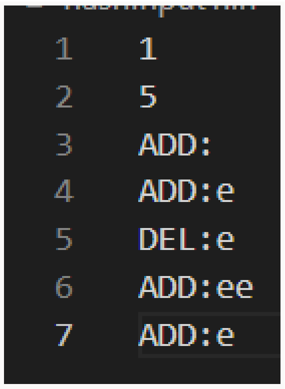


```
5
34:Dabrowski
46:Polski
63:marsz
76:ziemii
96:z

1
86:aace
```

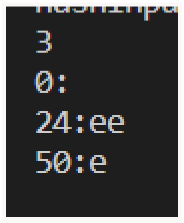
The first part of the test case is the same as the one provided by the question. Then we reset the table changing all the values in the table to be 0 and reset count to be 0. Then we add aac and aace to the table which result in count = 2. Then we proceed to delete acc which decrements 1 from count making count equal to 1 and only aace is left in the table.

This input is also from the comment section



```
1
5
ADD:
ADD:e
DEL:e
ADD:ee
ADD:e
```

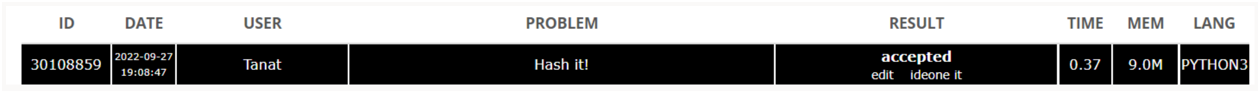
Now this is a very good input because we will be trying to add an empty string which should be put at an index 0. And the key that contains only ‘e’ like e, ee, eee, etc. will also have an index 0 at the hash table. As we are using an open addressing method it would be like first come first serve for index 0 position, so in this case, we first add an empty string which means at index 0 there's an empty string already. So when we add e which also has the same index value but because it's already taken it will continue forward finding an empty slot for it to fill.



```
3
0:
24:ee
50:e
```

So here after adding an empty string we add e and then remove it which leads to ee taking it place and then add e again at the last which makes it take place at 50 index because the index before that is already taken.

Now lastly we test our code on the website.



ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
30108859	2022-09-27 19:08:47	Tanat	Hash it!	accepted edit ideone it	0.37	9.0M	PYTHON3

Reference

https://www.chenguanghe.com/spoj-hashit-hash-it/

We took the idea of a solution from this website, but they provide the code in C language so we need to implement it in python, such as changing the hash table to contain 0 as its default instead of ‘’(empty string), to handle the condition when input in adding an empty string.