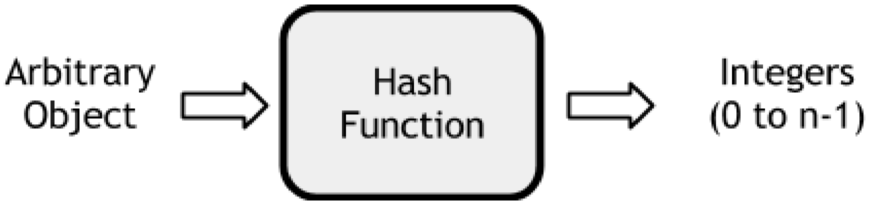
Assignment 7: Hashing

For this assignment, you have to implement the operations related to dictionary data structure. The operations are as follows -

➔ Search for an element stored in data structure.
➔ Insert new element.
➔ Delete an existing element.

All this operations can be implemented using the Hashing mechanism and thus can have expected time complexity of O(1).

The first step to the assignment is to design a hash function. The purpose of a hash function is to take an arbitrary object and map it to some integer.



Objects are stored using an array and we are basically computing an index to the array (0 to n-1 for an array of size **n**). For this assignment you will be processing `String` objects. An example of a simple hash function for strings is as follows -

```
Compute the sum of ASCII code of characters in the String
and store it in s.

Compute s % n where n=size of the array and return
```

Some examples are as follows -

```
s for the String “abc” = 97 + 98 + 99 = 294
Given n = 15, the result is 294 % 15 = 9

s for the String “Hello” = 72 + 101 + 108 + 108 + 111= 500
Given n = 15, the result is 500 % 15 = 5
```

The return value can be used as an index for storing the object

If we give “abc” and “bbb” as input, our hash function computes same result. So two different strings are eligible for same position in the array. This circumstance is known as Collision. As part of the data structure, you have to implement the code to handle such situations. An straightforward rude way is not to allow “bbb” to be inserted once “abc” has been already there and vice versa. This process is called Collision Resolution. Ways that are used for collision resolution in practice are -

- ➔ Linear Probing/Open addressing
- ➔ Separate Chaining

See the reference books of the course to get an idea how each of them works.

Direction

Create a classes named as `HashTableLP and HashTableSC.` Both of them will contain the following methods.

Method: `public boolean add (String str)`

adds `str` to the HashTable if it is not there already and returns `true`, otherwise does nothing and returns `false`.

Method: `public boolean delete (String str)`

deletes `str` from HashTable if it is already there and returns `true`, `false` otherwise.

Method: `public boolean contains (String str)`

checks whether `str` is present in the HashTable, returns `true` if str is in there, `false` otherwise.

Method: `public double loadFactor()`

returns the *load factor* of the HashTable. See references for the definition of load factor.

Method: `public int count ()`

returns how many strings are there in the HashTable.

Method: `public int size ()`

returns the size of the HashTable (length of the array)

The class `HashTableLP` will use linear probing for collision resolution. Separate Chaining will be used by `HashTableSC` class. Both of them will have a constructor through which we can specify the initial size of the hash table. The table must be resized when necessary and the existing items must be re-hashed to place in the new array. The size of the hash table should be prime in order to get better performance. The size user gives may not be a prime number. You need to compute by yourself. What you do is just find the smallest prime greater than user given size and create an array of that length. Use the following modified version of the example hash function.

```
Compute the sum of ASCII code of characters times their
position in the String and store it in s.

Compute s % n where n=size of the array and return
```

Examples given below

```
s for the String “abc” = 97 * 1 + 98 * 2 + 99 * 3 = 590
Given n = 15, the result is 590 % 15 = 5

s for the String “bbb” = 98 * 1 + 98 * 2 + 98 * 3 = 588
Given n = 15, the result is 588 % 15 = 3
```

Resizing the table:

The table must be resized if the load-factor is high. For linear probing the threshold is 0.75. That means if the load factor crosses 0.75, the table size must be doubled. For separate chaining, this threshold is 3.

Key Point on deletion for Linear Probing:

Reconsider the algorithm through which a String is located in the table (if exists). First it computes the hash value for the String. Beginning at that position, it looks for a match until a null (or empty place) is found. Now suppose our hash-table is as follows. Assume leftmost position as 0.

Hello	Bigg	Dark		Long		Fellow

Now let us insert “`Ticket`” into our hash table and assume `hash(“ticket”)` generates 1. That position is occupied by “`Bigg`”. So according to Linear Probing, “`Ticket`” will be placed as follows

Hello	Bigg	Dark	**Ticket**	Long		Fellow

Now lets delete “`Dark`” from the table. If nothing else is done, the table will look as follows

Hello	Bigg		**Ticket**	Long		Fellow

Now lets search for “`Ticket`”. As earlier, `hash(“Ticket”)` evaluates to 1. So we try to match it with “`Bigg`”. After failing we hit an empty place (or null) and our search algorithm terminates reporting “`Ticket`” is not there. Is it correct? How can we make our search algorithm work properly? Think Yourself.

Sample Input/Output

Test case will contain several lines. each lines will have the following format

```
command argument
```

The `command` part will indicate which operation is requested. `argument` will be present if the `command` requires. All commands you are required to handle are as follows -

`create <N>`	create a new Hash Table with `N` as argument to the constructor. The commands that follows will be applicable to this hashtable. Print the string “Created” at output.
`add <str>`	adds `str` to the hash table and prints the return value
`del <str>`	deletes `str` from hash table and prints the return value
`has <str>`	checks if the hash table contains `str` and prints the return value
`count`	prints number of strings currently in there.
`size`	prints the size of the hash table
`loadF`	prints the load factor of the hash table
`print`	prints the content of hash table in a formatted way. See below.

Sample IO Dataset:

Sample Input

```
create 16
add hello
has world
add world
del helo
has world
count
size
add friend
loadF
print
```

Sample Output

```
Created
true
false
true
false
true
2
17
true
0.17647058823529413
Index        String
-----        ------
0        --> world
1        -->
2        --> hello
3        -->
4        -->
5        -->
6        --> friend
7        -->
8        -->
9        -->
10       -->
11       -->
12       -->
13       -->
14       -->
15       -->
16       -->
```

This is the output when `HashTableLP` class is used. For `HashTableSC` class only change in output will be for the `print` command. In that case, index will be the bucket numbers and each bucket can contain several strings. Print them in one line separated by “comma followed by space” (i.e “, ”)

***Notes on copying other’s code*: If anybody found copying solution code from other student(s), all of them will be penalized. Penalty includes**

- ➢ ***Assigning ZERO as mark for the solution submitted***
- ➢ ***Assigning ZERO as mark for the best submission among all other submissions. (assigned at the end of semester)***

Copying solutions from the Internet will also incur similar penalties.

Submission & Contact

Submission Type	Individual, in-person during lab classes/consultation hours
Submission Deadline (HARD):	August 15, 2014

Students are encouraged to use Google Group for contacting to resolve their issues (instead of personal mails.)

Group Address: https://groups.google.com/forum/#!forum/algorithmfiesta2014