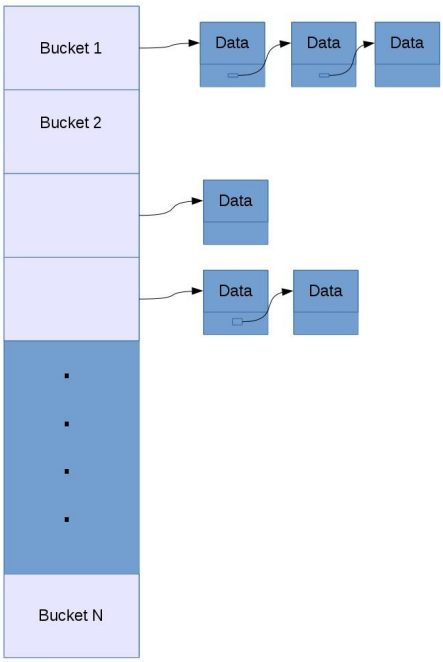
Hash tables

Objective: the objective of this laboratory class is to investigate the design issues of a hash table.

Laboratory work:

For this particular laboratory class, in a nutshell, you will implement **your own** hash table and see how it perform for different inputs. Pictorial view of the hash table you are expected to implemented is given below:



You will be using a method called open or external hashing for your implementation. In this particular case your hash function will take a key who's data type is *String* and would selected one of the N possible *buckets*. In addition to other information, each bucket would have a link to a linked list which contains the actual key/value (called data in the diagram) stored.

When storing a key/value pair, the hash function would select a bucket and you would add a new node to the list with the new key/value pair. For this implementation the value is the number of times the key was inserted. For example if one insert the word “the” 100 times the value should be 100.

When searching, one would select a bucket by using the same hash function and would search the key in the linked list. For searching the linked list one should use linear search.

The Hash function: an ideal hash function should distribute the keys evenly for all the buckets. With an ideal hash function your search time would be *M/N* where M is the number of keys and N is the number of buckets. Furthermore you can make the search fast by changing N.

Your main task is to develop a suitable hash function that would, as much as possible distribute the keys evenly. Note that keys are strings taken from a paragraph.

The interface to your hash table should include the following;

1. A method to create a *HashTable* were one can specify the number of buckets needed. This can be done with the constructor itself.
2. A method called void *insert(String key)* which would insert the given key to the *HashTable*.
3. A method called *int search(String key)* which would return the number of times the given key has appeared in the text.

In addition to the above interface, you should add other functions so that you can provide the requested details below. You should use the given skeleton code for the implementation. You may add other files to it.

What to submit: You are given two sample texts (with the skeleton code). Read the words in these

files and construct two hash tables. Note that you have to use only the words (“, : (etc. should be removed). Then see how the words are distributed. For this one might use things like maximum and minimum number of entries in buckets, average and standard deviation etc.

You **should submit** a report on what is the best hash function for this particular purpose. Your report should include graph to depict how the buckets are filled for different number of buckets, for different number of hash functions and different text files. You can decide how to show this (for example, maximum and minimum number of entries in buckets, average and standard deviation etc. can be used).

Once you have fine tuned the hash function for *sample-text1.txt* try the same with *sample-text2.txt*. Discuss whether the distribution is the same or not. Again you can use suitable graphs to demonstrate this.

Marks: Marks will be allocated for the design of your hash function and the good coding practices. Majority of the marks will be allocated for the report.

Deadline: 5th August 2019

Notes: Following might be useful
– The *String.split* function is very powerful when splitting strings. It can be used with regular expressions to extract only the words.
– There are ample resources on regular expressions in the web.