**BITS Pilani, Pilani Campus**
**2nd Sem. 2019-20**
**CS F211 Data Structures & Algorithms**

**================================**

# Lab IX

**==========================================**

**Topics**: Hash tables: Creation and Querying along with performance analysis

Hash tables are used for faster querying on large datasets. Their creation and querying performance depends upon many factors including size of hash table and hashing function.

### Exercise 1: Identification of best hash function for a given input domain

a) **Hash function**: Implement a hash function which takes a `string`, a `baseNumber`, and a `tableSize` variables as inputs and returns a number in range [0, `tableSize`). Hashing should be done as:
```
= (((sum of ASCII values of characters in string) mod baseNumber) mod
tableSize)
```
b) **Collision count**: A collision is said to occur whenever two inputs are mapped to same value by the hash function. Implement a function which takes an array of strings, a `baseNumber`, and a `tableSize` as inputs. It should call the hash function in 1(a) for all the strings in input array and return the number of times collision will occur if the given hash function is used for hashing.

c) **Parsing**: Implement an input parser which reads a text file in UTF-8 format and returns an array of all strings in the file which follows certain rule. For now, assume that the rule is, “Any white space separated sequence of only English characters”. Use the provided *“Project Gutenberg’s Alice’s Adventures in Wonderland, by Lewis Carroll”* file as the test case. The function should also print the total number of valid strings before returning the array of strings (let’s call it, `book`).

d) **Profiling**: Assuming that the metric for selection of best hash function is the one with minimal collisions, implement a function to identify the best value of `baseNumber` and `tableSize` from following range of parameters, for the `book` (input array of strings returned from parser in 1(c)):

   a. Possible choices for `baseNumber`: 3 prime numbers between `tableSize` and `2*tableSize` and 3 prime numbers larger than `1000*tableSize` (6 choices, hardcoding allowed)

   b. Possible choices for `tableSize`: 5000, 50000, and 500000 (3 choices, hardcoding allowed)

   Implement the profiling function such that it prints the collision values of all 18 choices and then prints the indices of best parameters in `baseNumbers` and `tableSize` arrays.

### Exercise 2: Creating and using a hash table

One of the strategies to create a hash table is with separate chaining such that collided values are inserted at the end of list. A hash table may additionally store a few fields with it, like:

- `elementCount`: total number of elements (strings) in the table
- `insertionCost`: total number of jumps done in any of the lists (chains) to insert the element at the end. Increment only in case of collision.
- `queryingCost`: total number of comparisons done in any of the chains during all lookups

Notes:

1. Implement the chain as a linked list of records containing:
   a. The index of the first occurrence of the string in the `book` array (i.e. there should be only one copy of each string, even if there are multiple hash tables).
   b. The count of occurrences of that string
2. . Also, hard code the hash function to use best values of `baseNumber` and `tableSize` and only take a string as input.

End of notes.

a) **Creation**: Implement a function to create an empty hash table with best value of tableSize returned from Exercise 1(d). All the fields of hash table should be appropriately initialized.

b) **Insert**: Implement a function which takes a hash table, an array of strings, say A, an index into the array, say j, and:
   - inserts the string at given index i.e. A[j] in the hash table if the string at Aj] is not present;
   - update the count if it is already present
     Note that only the index and count are stored in the Hashtable. i.e. if `index=5` and `book[5] contains the first occurrence of the string "alice"`, then the value `5` should be added in the hash table in the chain identified by hashing `"alice"` along with a count of 1. If `index = 237` and `book[237]` contains a subsequent occurrence of "alice" then the count of the record – already inserted in the hashtable – should be incremented. It should also update `insertionCost` variable of the hashtable. Any new value should be inserted at the end of chain.

c) **InsertAll**: Implement a function which inserts all strings of the `book` array in an empty hash table passed as argument. Once all values are inserted return the value of `insertionCost`.

d) **Lookup**: Implement a function to take a hash table and a string as input and return the corresponding record. It should also update `queryingCost`.

e) **LookupAll**: Implement a function to take a hash table, an array of strings and a real number `m` as input. It should lookup all strings in the given hash table which appear in first `m%` indices of the array of strings. E.g. `m = 0.05` should trigger querying of first 5% entries of the input array to be looked up in the given hash table. It should reset `queryingCost` in the beginning and return it in the end.

### Exercise 3: Profiling and Optimizing hash table for an input domain

a) **Profiling**: Implement a function which calls LookupAll with varying number of percentages from 10% to 200% with a step of 10%, to determine the percentage where `queryingCost` overtakes `insertionCost` for this input `book` and the given parameters.
   **Cleanup**: Given a list of stop words (given in a separate file), delete all entries of stop words from the hashtable. Call profiling function on the updated hash table to identify the percentage where `queryingCost` overtakes `insertionCost` for this input `book` and the given parameters.

Solution to be uploaded as a part of this lab: Solve Exercise-1 and upload its solution.