Birla Institute of Technology & Science, Pilani
2nd Semester 2016-17 - CS F211 – Data Structures and Algorithms

---

Lab 6 : 25th Feb 2017
Hash Tables (Rehashing)

Instructions:

- All input expressions should be read from stdin (scanf) and output should be printed on stdout (printf).
- Write code in different header (*.h) and source code (*.c) files. Name them properly, compile each of them using **–c** flag of gcc and combine all *.o files to create an executable file. You may create a Makefile for the above job as well.

Problem:

Compilers use a data structure known as Symbol Table to store list of identifiers and their type for each block of code. The symbol table is usually implemented as Hash Table to insure fast lookups. In case of collisions, linear chaining can be used to store the new value at the end of existing linked list with the same index. Performance of a hash table for lookups and optimization of memory required, can be controlled by keeping the load factor of the table within some bounds.

Load factor is defined as the ratio of number of entries in the table with number of buckets. A common approach to maintain load factor between **[minLoad, maxLoad]** is to rehash the table whenever *load factor* reaches these limits. *Rehashing* means creating a new table with larger/smaller bucket size and inserting all entries of old table into new table and in the end freeing up the old one. New bucket size is calculated by multiplying/dividing the existing size by a factor known as *resizing factor*. For example, if maxLoad is 0.75 and *resizing factor* is 2, then in case of rehashing the size of (number of buckets in) new table will be twice the old size.

Implementation:

Each identifier symbol can be represented by a key-value pair. Name (key) as well as its type (value) will be string (array of characters). Each identifier should be stored as a structure so that more information can be stored with it later. Let's call it, `symbol`. Read given input data in a dynamically allocated array of symbol structure (let's call it, `symbolList`).

Hashing Function:

`index = ((sum of ASCII value of characters in key) mod` 2^{16}`) mod size`

Hint: Think about a quick way to perform mod 2^K *operation.*

With a hash table, keep following additional fields:

- `entries`: total number of symbols in the table
- `size`: number of buckets
- `loadFactor`: entries/size
- `freeSlots`: number of empty buckets (should be equal to size in the beginning)
- `insertionTime`: total number of jumps done in any of the lists (chains) to insert the element at the end. Increment only in case of collision.
- `queryingTime`: total number of comparisons done in any of the chains during all lookups.

Note: The nodes of the table should store a pointer to actual symbol in `symbolList` array i.e. there should be only one copy of each symbol but accessible through different hash tables.

Input Format:
Implement following top-level functions to be called by driver based on given key.

1. Function **readSymbols**:
   - Key: 0
   - Format:

     0 N
     $name_1$ $type_1$

     $name_2$ $type_2$

     …..

     $name_N$ $type_N$
   - Description:
     - "0" shows start of a set of symbols.
     - "N" represents number of symbols to come.
     - Next N lines will have a two words (separated by a tab), in each line. First word will be name of the identifier and second word will be its type.
     - Store all symbols in a dynamic array of size N (symbolist).

2. Function **readQueries**:
   - Key: 1
   - Format: same as Function readSymbols
   - Description:
     - Store given symbols in another dynamic array named as queryList

3. Function **createHashTable**:
   - Key: 2
   - Format: 2 size minLoad maxLoad resizeFactor
   - Description: Use algorithm given below to create a hash table with given parameters and insert all symbols of symbolList

Algorithm `createHashTable (size, minLoad, maxLoad resizeFactor)`

```
H = createEmptyHashTable(size)
for (sym in symbolist)
     H = insert(H, sym)
     if (H.loadFactor > maxLoad)
          newSize = H.size * resizeFactor
          Hnew = createEmptyHashTable(newSize)
          Hnew.insertionTime = H.insetionTime
          Hnew = reinsert all symbols in H into Hnew
          delete(H)
          H = Hnew i.e. do remaining insertions in Hnew
if (H.loadFactor < maxLoad)
     newSize = H.size / resizeFactor
     Hnew = createEmptyHashTable(newSize)
     Hnew.insertionTime = H.insetionTime
     Hnew = reinsert all symbols in H into Hnew
     delete(H)
     H = Hnew
PrintTabSeparated(H.entries, H.size, H.loadFactor,
H.freeSlots, H.insertionTime)
```

Algorithm `insert(hashTable H, symbol s)`

```
H.entries = H.entries + 1
index = hashingFunction (s)
if (isSlotEmpty (index))
     H.freeSlots = H.freeSlots – 1
t = insertSymbol(index, s)
H.insertionTime  = H.insertionTime + t
H.loadFactor = H.entries / H.size
```

4. Function **lookupQueries**:
   - Key: 3
   - Format: 3
   - Description:
     - Lookup each symbol name of queryList in the last hash table. Keep updating queryingTime parameter of hash table.

- In the end, print number of symbols in queryList and queryingTime of the hash table (tab separated)

Test Case 1:

Sample Input	**Sample Output**
0 10 cristian int abbigail char jonathan int abdullah double cristian int adelaide float jonathan char prentice int juliette double kasandra float 1 5 jonathan int cristian int cristian int juliette double kasandra float 2 12 0.1 0.75 2 3 2 8 0.2 0.9 5 3 -1	