University of Waterloo
CS240, Winter 2010
Assignment 4

Due Date: Wednesday, March 17, at 5:00pm

Please read `http://www.student.cs.uwaterloo.ca/~cs240/w10/guidelines.pdf` for guidelines on submission. In particular, don't forget to make a **cover page** (`https://www.student.cs.uwaterloo.ca/~isg/makeCover?course=cs240`) and attach it to the front of any written assignment. There are 65 marks available. The assignment will be marked out of 60.

Problem 1 Extendible Hashing [6+6=12 marks]

Suppose we have an extendible hashing scheme with block size $S = 3$ and parameter $L = 5$. The universe of keys is non-negative integers with at most 8 bits, $U = \{0, 1, \ldots, 255\}$, and the hash function is

$$h(k) = \lfloor k/16 \rfloor + (k \bmod 16).$$

The dictionary is initially empty, with a single block B, pointed to by both entries in the directory, which has initial order $d = 1$.

(a). Show the result of inserting the following **keys** into the dictionary, in order. In each block location, you should write the key and the corresponding hash value. Indicate the order d and the local depth k_B of each block.

You don't need to draw out every single step, but you must indicate clearly every time a *block split* or *directory grow* occurs, and draw the resulting structure immediately following each such operation, and at the end of all the insertions.

The sequence of keys to insert is:

$$191,\ 142,\ 192,\ 248,\ 217,\ 95$$

(b). We say that the extendible hashing structure has "overflowed" when a block needs to be split, but its local depth k_B already equals the directory order d, which equals the width L of the hash function (so that a directory grow is not possible).

What is the *least* number of insertions into this table (with the parameters described above) before it could overflow? What is the *greatest* number of insertions into this table before it could overflow? Briefly explain your answers, and try to be as precise as possible.

Problem 2 Gallop Search [5+5+5=15 marks]

In this problem, we will examine some extensions of the galloping search idea. For the following, assume we are searching for an integer in a sorted list A of size n. If we search for the mth smallest number in this list (although of course we don't know m in advance), recall that the cost of normal galloping search is $\Theta(\log m)$.

(a). Suppose we know that the integers in A are roughly "evenly distributed". So we want to use the idea of interpolation search to make each "guess", rather than just doubling the index at each iteration.

Describe an algorithm which incorporates interpolation search into galloping search. You do not need to give an analysis, but the cost should be $O(\log \log m)$ if the integers in the array are "evenly distributed" as discussed in lecture.

(Hint: First repeatedly guess an index i such that $A[i]$ is approximately **twice as large** as k, then perform a normal interpolation search in the range $A[0], A[1], \ldots, A[i-1]$.)

(b). The idea of a skip list is to build links for a normal binary search over a linked list. Show how to perform a galloping search in a skip list. (You may **not** change the structure of the skip list.) Again, you do not need to give an analysis, but the cost should be $O(\log m)$, where m is the index of the item searched for in the skip list. (Hint: the idea is very similar to normal *skip-search*.)

(c). Normal galloping search doubles the search index at each iteration. **Hyper galloping** instead squares the index at each iteration. Specifically, the normal *gallop-search* algorithm on slide 5 in module 6 has line 2 changed to "$i \leftarrow 2$" and line 4 changed to "$i \leftarrow i^2$".

Show that normal galloping search costs $\Theta(\log m)$ in the **best case**, and that hyper galloping improves this best case cost to $\Theta(\log \log m)$. Also explain why the worst-case cost of hyper galloping is still $O(\log m)$. Here, "best case" and "worst case" mean when we are searching for the item at index m.

Problem 3 Searching for only a few values [6+6=12 marks]

Self-organizing search should be useful when some dictionary entries are accessed much more often than others. Here we will consider a special case, where *every* search is for one of only r elements in the dictionary.

Specifically, say a dictionary contains n items stored in a linked list L, initially in any order, and the r keys $k_1, k_2, \ldots, k_r$ are in the dictionary. We want to know the cost of a sequence of m searches in the dictionary, where each one is of the form *search*(k_i) with $1 \leq i \leq r$. (We can assume that $r \leq m$ and $r \leq n$.)

(a). Show that the worst-case cost for this sequence of searches is $\Theta(r(m+n))$, if **move-to-front** is used.

(b). Show that the worst-case cost for this sequence of searches is $\Theta(mn)$, if **transpose** is used.

Problem 4 Cuckoo Hashing (Programming) [8+12+6=26 marks]

You are implementing a server which must keep track of all users that are currently logged in. Specifically, you need to maintain a dictionary mapping IP addresses (keys) to usernames (values). When a new connection is made, you need to quickly determine whether that IP address is already tied to a specific user. So you decide to use a hash table with cuckoo hashing to store IP address-username pairs.

For extra efficiency, you try a slight variation of standard cuckoo hashing called **3-way cuckoo hashing**. In this version, three different hash functions are used (instead of only two). The *search* and *delete* operations now must look in (at most) three locations.

For *insert*, you should do the following when "kicking out" an item already in the hash table:

- If the item was in the location specified by the *first* hash function, put it in the location specified by the *second* hash function.
- If the item was in the location specified by the *second* hash function, put it in the location specified by the *third* hash function.
- If the item was in the location specified by the *third* hash function, put it in the location specified by the *first* hash function.

Each IP address (IPv4) is a sequence of four integers (a_3, a_2, a_1, a_0), each satisfying $0 \leq a_i \leq 255$. Each username is a character string with length at most 8.

There are three files to submit for this problem:

- `IP.h / IP.java`: Completed in part (a)
- `UserTable.h / UserTable.java`: Completed in part (b)
- `TestTable.h / TestTable.java`: Completed in part (c)

Starter code for each of these files is available for download on the assignemnts page. **Do not change the signatures (i.e. parameters, return value) of any functions in the skeleton code**, as we will use these for testing. You may add any class members or helper functions that you need.

You may not use any external libraries to help you solve these problems. In particular, Java programmers must not use any classes from `java.net` or `java.util` other than `Scanner`, and C++ programmers should not use any standard template library classes other than `string`.

(a). The `IP` class represents a single IPv4 address. Already completed are a constructor and the following methods (see the skeleton code for details):

- `setIPString`
- `getIPString`
- `sameIP`

You just need to complete the implementation of the three hash functions `hash1`, `hash2`, and `hash3`.

`hash1` should use the division method. `hash2` should use the multiplication method with constant $a = \varphi$ (already defined for you in the code).

For `hash3`, you should develop your own hash function. This can be whatever you want, but it must correctly map IP addresses to integers in the range $\{0, 1, \ldots, M-1\}$, and it should also follow the general principles we discussed on creating hash functions. Be sure to document your method well with comments.

(b). The `UserTable` class will represent a hash table of IP addresses and usernames. A constructor (and in C++, a destructor) have already been written for you, as well as a method `dump` which prints out all the contents of the hash table.

There is also an *private inner class* `User` with public fields `ipaddr` and `username`. In C++, there is an additional field `empty` to indicate that this table entry is empty. In Java, this is indicated by setting the table entry to `null`. If you haven't seen inner classes before, suffice it to say that the `User` class can be used from within `UserTable` just like any other class would be.

You need to complete the implementation of the three methods `search`, `insert`, and `remove` (we have to call it this instead of "delete" because `delete` is a keyword in C++). Use the method of 3-way cuckoo hashing described above, and the three hash functions in the `IP` class you wrote in part (a).

(c). Now you want to test how well your hash table performs. In particular, you want to see how many IP address–username pairs can be inserted when the size of the table is fixed (since you haven't implemented rehashing functionality).

The file `TestTable.cc` or `TestTable.java` will contain a main method to test your hash table implementation. The program should read a single integer from the command line to be the size M of the hash table. Then lines of the form

```
<IP address> <username>
```

should be read from standard in. For each pair, try to insert that user into the hash table of size M. Stop when an insertion fails, or when end-of-file is reached. Then print out the numeber of `User`s successfully inserted into the table, followed by a blank line, followed by the result of calling `dump()` on the table.

An outline of this `main` method has already been written for you. You just need to complete it. A file `sample.txt` with lines of the format specified is also provided with the starter code.