HASHING STUDY QUESTIONS Solutions
Fundamental Algorithms, Fall 2004, Professor Yap

Due: Do not submit.
SOLUTION PREPARED BY Instructor and T.A.s Ariel Cohen and Vikram Sharma

INSTRUCTIONS:

- This Study Question is in lieu of Homework 6, and focuses on hashing (Lecture XI). This should be your priority in studying Lecture XI.

---

1. Coalesced Hashing
As usual, we have a hash table $T[0..m-1]$ with m slots. We assume a hash function $h : U \to \mathbb{Z}_m = \{0, \ldots, m-1\}$. We want to implement coalesced hashing, but a little differently than described in the Lecture Notes. The approach described here is more[1] straight forward because we will *explicitly* introduce a variable to remember the "state" of each slot. As usual, we assume that the keys stored in the hash table are unique. The i-th slot $T[i]$ has four fields:

   (a) $T[i]$.`Key` which stores a key (element of U).

   (b) $T[i]$.`next` which stores an element of $\mathbb{Z}_m$.

   (c) $T[i].\mathit{State}$ stores a value in $\{-2, -1, 0, 1, 2\}$ where $\mathit{State} = 0$ indicates the ORIGINAL state, $|\mathit{State}| = 1$ indicates OCCUPIED, $|\mathit{State}| = 2$ indicates DELETED. Initially, $\mathit{State} = 0$ but once a slot has been used, it never revert to this state again. Moreover, if $\mathit{State} > 0$, indicates this slot as EOC (=END_OF_CHAIN); $\mathit{State} < 0$ means it is not EOC.

   (d) $T[i]$.`Data` which stores associated data. This is important in practice, but as usual, we ignore this field in our algorithms.

   (i) Describe scenarios where we need to use all the 5 possible state values in our data structure.
   (ii) Consider the operation FINDKEY(k) which returns $i \in \mathbb{Z}_m$ where $T[i]$ is a slot in the chain that begins at slot $T[h(k)]$. Moreover, one of three properties hold: (a) $|T[i].\mathit{State}| = 1$ and $T[i].\mathit{Key} = k$. (b) k is not in the current hash table, and $|T[i].\mathit{State}| \neq 1$. (c) k is not in the current hash table, and $T[i].\mathit{State} = 1$ (thus $T[i]$ is OCCUPIED and is EOC). Implement this algorithm.
   (iii) Consider the operation INSERT(k) which inserts k into the table if the table is not already full and does not contain k. Implement INSERT(k) with the help of FINDKEY(k). Assume a global integer variable N which remembers the number of keys currently in the hash table. If $N = m$, INSERT(k) returns an ERROR condition. Otherwise, it returns the $i \in \mathbb{Z}_m$ where k is stored. It is important to ensure that you do not create cycles during INSERTION.
   (iv) Implement DELETION(k), with the obvious meaning: if k is in the table, it will be deleted.
   (v) We want to prove a fundamental property of your solution in parts (iii) and (iv). SHOW that a sequence of INSERTION and DELETION operations does not introduce a cycle in our linked lists. Assume that we start from an empty hash table where $T.\mathit{State}[i] = 0$ for all i.

   SOLUTION:

   (i) Initially, we need state 0 (ORIGINAL state). We also need to indicate a slot as OCCUPIED, after it is first used for the first time. If an OCCUPIED slot is deleted, we must indicate it as DELETED (state 2) and not return it to the ORIGINAl state because this slot might be used in some chain. Now, if we were search for some key k along a given chain, it is necessary to know when we have reached the end of the chain. Hence, each slot must indicate whether it is EOC or not. Moreover, EOC is independent of the DELETED/OCCUPIED states. Hence we need 2 versions of DELETED (states -2 or 2) and 2 versions of OCCUPIED (states -1 and 1). We choose the positive states to indicate EOC.

   (ii) The algorithm for FINDKEY(k) goes as follows: first compute $h(k)$ and see if $T[h(k)].\mathit{State} = 0$. If so, return $h(k)$. Otherwise, we follow the chain looking for the key k; if k is found, we return its

[1] In the lecture notes, we used special values of `next` to encode this state information. Actually, we will also need a special value of `Key` (say `Key` = 0) to help us in this encoding – hence the write up in the Lecture Notes is buggy.

slot. Otherwise we will reach the end of chain. At this point, we return the first deleted slot (which we took care to remember) that we found. If there were no deleted slot, we return the current slot.

```
FINDKEY($k$)
1.   $i \leftarrow h(k)$.
2.   if ($T[i].State = 0$) return($i$).
3.   EMPTY$\leftarrow -1$.
4.   while (1)
     4.1  if ($|T[i].State| = 1$ and $T[i].\texttt{Key} = k$) $\triangleleft$ $k$ is found
                    and $T[i].\texttt{Key} = k$) return($i$).
     4.2  if ($|T[i].State| = 2$ and EMPTY $= -1$) $\triangleleft$ Found first deleted slot
                    EMPTY$\leftarrow i$.
     4.3  if ($T[i].State > 0$) $\triangleleft$ Reached end of chain (EOC)
                    if (EMPTY$\neq -1$) return(EMPTY)
                    else return($i$)
     4.4  $i \leftarrow T[i].\texttt{next}$
```

(iii) The insertion algorithm is based on FINDKEY:

```
INSERT($k$)
1.   $slot \leftarrow$FINDKEY($k$).
2.   if (($T[slot].\texttt{Key} = k$) and ($|T[slot].State| = 1$)) $\triangleleft$ Key $k$ already present
               return(ERROR).
3.   if (Table is full) return(ERROR).
4.   if ($|T[slot].State| \neq 1$) $\triangleleft$ If slot is a deleted entry
               $T[slot].\texttt{Key} \leftarrow k$ and return(SUCCESS).
5.   if ($T[slot].State = 1$) $\triangleleft$ End of Chain
     5.1       $j \leftarrow slot++(\mathbf{mod}\, m)$.
     5.2       while ($j \neq slot$)
     5.2.2         if ($T[j].State = 0$)
                       $T[j].\texttt{Key} \leftarrow k$, $T[slot].\texttt{next} \leftarrow j$, return(SUCCESS)
     5.2.3         if ($|T[j].State| = 2$)
                       $\ell \leftarrow j$.
                       while ($T[\ell].State < 0$), $\ell \leftarrow T[\ell].\texttt{next}$.
                       if ($\ell \neq slot$)
                           $T[j].\texttt{Key} \leftarrow k$, $T[slot].\texttt{next} \leftarrow j$, return(SUCCESS)
     5.2.4         $j++(\mathbf{mod}\, m)$
     5.3       return(ERROR).
```

(iv) Implementation of DELETE: This is easy, omitted.

(v) No Cycles: clearly, deletion cannot create cycles, since we do not change the links. For insertion, the new links into slots whose $State = 0$ is safe. The only potential danger comes from linking into slots with $|State| = 2$. The only new links in our INSERTION code above occurs inside (Line 5). We ensure that $T[slot].\texttt{next}$ only points to a slot j that does not lead back to $T[slot]$ (so no cycles is created).

2. Universal Hashing
The key result about how to use Universal Hash Functions is represented by Theorem 5 (p. 12, Lecture XI). Through this exercise, we want you to be familiar with a particular class of universal hash sets, namely the ones described in Lecture XI in §5 (p. 15-16). By a "finite field" F, you may assume that we mean[2] a set of the form $F = \mathbb{Z}_q = \{0, \ldots, q-1\}$ where q is a prime number. The four arithmetic

[2]There are other finite fields besides these, namely those with $|F|$ a power of prime. But the arithmetic here is more complicated, and you need not know about them.

operations in F are just the usual ones, but always modulo q. The most important thing you need to know about F is that the operation of *inverse* is defined. That is, for each $x \in F$, if $x \neq 0$ then there is a unique element $y \in F$ such that $xy = 1$. We call y the *inverse* of x and denote it by x^{-1}. The Example and Solution on page 16 should be mastered.
(i) What is the simplest example of a finite field?
(ii) In the finite field $\mathbb{Z}_{13}$, find the inverses of $x = 1, 2, 3, 4, 5, 6$. REMARK: there is an algorithm based on Euclid's algorithm for computing inverses. But you just need to find inverses by brute force search.
(iii) As a compiler designer, you want to construct a hash table to store all the user-defined variables that might be encountered in a compiled program. Assume each variable name (i.e., *key* for hashing) comes from the set $U = \Sigma^{30}$ where $\Sigma = \{\sqcup, a, b, \ldots, z, 0, 1, \ldots, 9\}$. So $|\Sigma| = 37$ and each key has length exactly 30. (NOTE: if a key has length less than 30, we assume you pad it with $\sqcup$ until it is 30.) You want to create a hash table $T[0..m-1]$ where $1000 < m < 2000$, and resolve collision using separate chaining. Show how to choose a hash function so that the expected number of keys that collide with any given key is at most 1.

HINT: First, choose a universal hash set $H \subseteq [U \to \mathbb{Z}_m]$ for an appropriate m. Remember that there are lots of primes[3] but for our purposes perhaps it is enough to know that smallest prime larger than $37^2 = 1369$ is $q = 1373$.
(iv) For a program with at most 1000 variable names, what is an upper bound on the expected length of any chain in your hash table in part (ii)?

SOLUTION:

(i) The simplest finite field is $F = \mathbb{Z}_2 = \{0, 1\}$; this may be familiar to you.

(ii) Choose $F = \mathbb{Z}_q$ where $q = 1373$. Treat $\Sigma^2 = \Sigma \times \Sigma$ as a subset of F because each pair $(a, b) \in \Sigma^2$ can be viewed as the number between 0 and 1368.

Also view $U = (\Sigma^2)^{15}$ as the set F^{15}. We use the hash function

$$h_a(x) = (a_0 + \sum_{i=1}^{15} a_i x_i) \,\mathbf{mod}\, q$$

where $a = (a_0, a_1, \ldots, a_{15}) \in F^{16}$ and $x = (x_1, \ldots, x_{15})$ If we randomly pick the constants $a \in F^{16}$, then h_a is a universal hash function. Thus $m = 1373$ is our table size.

(iii) With $n \leq 1000$, we have $\alpha \leq 1000/1373 < 0.73$. According to Theorem 5 in the Notes, the expected length of a non-empty chain is $1 + \alpha < 1.73$.

[3] You can easily look up some standard mathematical table for primes up to 2000.