Practical Session 8- Hash Tables

Hash Function	A hash function ***h*** maps keys of a given type into integers in a fixed interval [0,m-1]
Uniform Hash	$\Pr[h(key)=i]=\frac{1}{m}$, where m is the size of the hash table.
Hash Table	A hash table for a given key type consists of: • Hash function ***h: keys-set →[0,m-1]*** • Array (called table) of size ***m***
Direct Addressing	K is a set whose elements' keys are in the range [0,m-1]. Use a table of size m and store each element x in index x.key. Disadvantage: when |K| << m → waste of space
Chaining	h(k) = k mod m (This is an example of a common hash function) If h(k) is occupied, add the new element in the head of the chain at index h(k)
Open Addressing	**Linear Probing:** $h(k,i) = (h'(k) + i) \bmod m \quad 0 \le i \le m-1$ h'(k) - common hash function First try h(k,0) = h'(k) , if it is occupied, try h(k,1) etc. ... Advantage: simplicity Disadvantage: clusters, uses **Θ(m) permutations** of index addressing sequences **Quadratic Probing:** $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad 0 \le i \le m-1$ c1 and c2 constants ≠0 Advantage: Works much better than Linear Probing. Disadvantage: Secondary clustering. **Θ(m) permutations** **Double Hashing:** $h(k,i) = (h1(k) + i \cdot h2(k)) \bmod m \quad 0 \le i \le m-1$ h1 – hash function h2 – step function First try h(k,0) = h1(k), if it is occupied, try h(k,1) etc. ... Advantage: less clusters , uses **Θ(m*m) permutations of index addressing sequences**

Load Factor **α**	$\alpha = \frac{n}{m}$ **,** Hash table with m slots that stores n elements (keys)
Average (expected) Search Time	**Open Addressing** unsuccessful search: O(1+1/(1-α)) successful search: O(1+ (1/α)ln(1/(1-α))) **Chaining** unsuccessful search: **Θ** (1 + α) successful search: **Θ** (1 + α/2) = **Θ** (1 + α)
Dictionary ADT	The dictionary ADT models a searchable collection of ***key-element*** items, and supports the following operations: • Insert • Delete • Search

Remarks:

1. The efficiency of hashing is examined in the average case, not the worst case.
2. Load factor defined above is the average number of elements that are hashed to the same value.
3. Let us consider chaining:

   3.1 The load factor is the average number of elements that are stored in a single linked list.

   3.2 If we examine the worst-case, then all n keys are hashed to the same cell and form a linked list of length n. Thus, running time of searching of an element in the worst case is (Θ(n)+ the time of computing the hash function).

So, it is clear that we are not using hash tables due to their worst case performance, and when analyzing running time of hashing operations we refer to the average case.

Question 1

Given a hash table with m=11 entries and the following hash function h_1 and step function h_2:

$h_1(key) = key \bmod m$
$h_2(key) = \{key \bmod (m-1)\} + 1$

Insert the keys {22, 1, 13, 11, 24, 33, 18, 42, 31} in the given order (from left to right) to the hash table using each of the following hash methods:

a. Chaining with $h_1 \Rightarrow h(k) = h_1(k)$
b. Linear-Probing with $h_1 \Rightarrow h(k,i) = (h_1(k)+i) \bmod m$
c. Double-Hashing with h_1 as the hash function and h_2 as the step function
   $\Rightarrow h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

Solution:

	Chaining	Linear Probing	Double Hashing
0	33 → 11→ 22	22	22
1	1	1	1
2	24 → 13	13	13
3		11	
4		24	11
5		33	18
6			31
7	18	18	24
8			33
9	31→ 42	42	42
10		31	

Question 2

a. For the previous h_1 and h_2, is it OK to use h_2 as a hash function and h_1 as a step function?
b. Why is it important that the result of the step function and the table size will not have a common divisor? That is, if h_{step} is the step function, why is it important to enforce $GCD(h_{step}(k),m) = 1$ for every k?

Solution:

a. No, since h_1 can return 0 and h_2 skips the entry 0. For example, key 11→ $h1(11) = 0$, and all the steps are of length 0, $h(11,i) = h2(11) + i*h1(11) = h2(11) = 2$. The sequence of index addressing for key 11 consists of only one index, 2.

b. Suppose $GCD(h_{step}(k),m) = d > 1$ for some k. The search after k would only address 1/d of the hash table, **as the steps will always get us to the same cells. If those cells are occupied we may not find an available slot even if the hast table if not full.**

Example: $m = 48$, $h_{step}(k) = 24$,$GCD(48,24) = 24$ ➔ only $\frac{48}{24} = 2$ cells of the table would be addressed.

There are two guidelines on choosing m and step function so that $GCD(h_{step}(k),m)=1$:

1. The size of the table m is a prime number, and $h_{step}(k) < m$ for every k. (An example for this is illustrated in Question1).
2. The size of the table m is 2^x, and the step function returns only odd values.

Question 3

Suggest how to implement Delete(k) function for a hash table, when using open addressing.

Solution: Each table entry will contain a special mark to indicate if it was deleted.
While searching for a key, if we encounter an entry which is marked as deleted, we continue to search. Note that when using this solution, the search time of an element is not dependent on the load-factor.

```
Delete (k)
  HT[find(k)]="deleted"

Insert(k)
  j:= ∞
  for (i:=0; i<n; i++)
    if  (HT[h(k,i)] = "empty" || HT[h(k,i)]="deleted")
      j:=i;
      break;
  if j = ∞
     print “hash table overflow”
  else
     HT[h(k,j)]:=k

Find(k)
  j:= ∞
  for (i:=0; i<n; i++)
    if  (HT[h(k,i)] = k)
      return i;
    if  (HT[h(k,i)] = "empty")
        return -1;
  if j = ∞
    return -1;
```

Question 4

Consider two sets of integers, $S = \{s_1, s_2, ..., s_m\}$ and $T = \{t_1, t_2, ..., t_n\}$, $m \leq n$.
a. Device an algorithm that uses a hash table of size m to test whether S is a subset of T.
b. What is the average running time of your algorithm?

Note that the below analysis gives an estimation of average run time

Solution: The naive solution is

```
Subset(T,S,n,m)
   T:=sort(T)
   for each s_j ∈ S
      found := BinarySearch(T, s_j)
      if (!found)
         return "S is not a subset of T"
   return "S is a subset of T"
```

Time Complexity: O(nlogn+mlogn)=O(nlogn)

The solution that uses a hash table with chaining of size m:

```
SubsetWithHashTable(T,S,n,m)
  for each t_i ∈T
    insert(HT, t_i)
  for each s_j ∈ S
    found = search(HT, s_j)
    if (!found)
      return "S is not a subset of T"
  return "S is a subset of T"
```

Time Complexity Analysis:

- Inserting all elements t_i to HT: nO(1) = O(n)
- If S $\subseteq$ T then there are m successful searches: $m(1+\frac{\alpha}{2}) = O(n+m)$
- If S $\not\subseteq$ T then in the worst case all the elements except the last one, s_m in S, are in T, so there are (m-1) successful searches and one unsuccessful search: $(m-1)(1+\frac{\alpha}{2}) + (1+\alpha) = O(n+m)$

Time Complexity: O(n+m) = O(n)

Question 5

In a hash table with double hashing (without deletions) assume that in every entry *i* in the table we keep a counter c_i where c_i is the number of keys k in the table such that $h_1(k) = i$.

a. How would you use this fact in order to reduce (usually) the number of accesses to the table entries during an unsuccessful search?

b. Show an example of an unsuccessful search in which the number of accesses to table entries is reduced from n to 2 (using your answer to item a).

c. Does this algorithm work for the linear probing method as well?

Solution:

a.

```
Search(HT,n,k,h)
  index=h(k,0)
  c=c_index
  i=0
  do
    if  c = 0
        return -1
    if  HT[h(k,i)] is empty
        return -1
    if  HT[h(k,i)] = k
        return h(k,i)
    else
        k'=HT[h(k,i)]
        if (h(k',0) = index)
           c = c-1
        i = i+1
  while (i<n)
```

In the worst case we might access all table till we find or fail to find the variable.

b. m= 7

$h_1(k) = k \bmod 7$

$h_2(k) = (k \bmod 6)+1$

Inserting the keys in the following order { -3, 3, 1, 8, 9, 12,21} results in the next hash table:

0	8	$c_0=1$
1	1	$c_1=2$
2	9	$c_2=1$
3	3	$c_3=1$
4	-3	$c_4=1$
5	12	$c_5=1$
6	21	$c_6=0$

Searching for key 29 takes 2 accesses to table entries instead of 7.

c. The algorithm will work for linear probing as well. Again, we might end up accessing almost over all table entries. The algorithm improves search time significantly if the table is sparse.

Question 6

In Moshe's grocery store, an automatic ordering system that uses a Queue (FIFO) is installed. Whenever a client places an order, the order's data (product, quantity, client's name) are being inserted to the back of the Queue. Moshe is extracting orders from the front of the queue, handling them one by one. In order to avoid a scenario where the store runs out of some product, every time Moshe extracts an order from the front of the Queue, he would like to know the total quantity of the currently extracted product, summed over all of this product's orders in the Queue.

Find a data structure that supports the following operations in the given time:

1. enqueue(r)-Inserting an order to the back of the queue, r = (product, quantity, client's name). Running time- O(1) on average.

2. dequeue()-Extracting an order from the front of the queue. Running time- O(1) on average.

3. Query(p)-Returns the total quantity of the product p in the Queue.

It is known that there are n products in the grocery. The Queue may hold at most m orders at any given time. We know that **m<n**. The data structure may use O(m) memory.

Solution

We will use a Queue and a hash table with chaining of size O(m). Each element (in the linked list) contains a key – the product's name and another field – its quantity.

1. enqueue(r) – Insert the order to the back of the queue. Search for r.product in the hash table. If r is in the table, add r.quantity to the quantity field of the appropriate element. If not, insert r to the hash table according to its key r.product.

2. dequeue() – Extract r from the front of the queue. Search for r.product in the hash table (it must be in it). Decrement the quantity field of the element by r.quantity. If the quantity is 0, remove the element from the hash table.

3. Query(p) – Look for p in the hash table. If it's in it, return its quantity; otherwise, return 0.

Notice that $\alpha = \frac{numberOfDifferent\,\mathrm{Pr}oductsInQueue}{SizeOfHashTable} \le \frac{m}{\Theta(m)} = O(1)$; therefore, the running time of the three operations is O(1) in average.