Solutions to Problem Set 4

Ekkapot Charoenwanit
Efficient Algorithms

Problem 4.1. Direct Addressing

1) Suppose that a dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

Solution I:
Assume that the values of elements have no relationship with their keys. Under this assumption, we need to search the entire table to look for the maximum element. Therefore, the running time is $\mathcal{O}(m)$. In other words, the worst-case running time is $\Theta(m)$.

Note that we first initialize the variable $maxValue$ to $-\infty$ to facilitate the first comparison; $T[1]$ is the first element so we need a dummy value to compare with $T[1]$. In the case that the table is empty, the algorithm returns 0.

Algorithm 1 implements searching for the maximum element in a direct-address table.

```
1: procedure DIRECT-ADDRESS-TABLE-MAX(T)
2:     maxValue = −∞
3:     maxIndex = 0
4:     for i = 1 → m do
5:         if T[i] ≠ NULL then
6:             if T[i] > maxValue then
7:                 maxValue = T[i]
8:                 maxIndex = i
9:     return maxIndex
```

Solution II:
Assume that the key of an element determines its value; the larger the key, the larger the value. Given a direct-address table T with m slots, the largest possible key must go to the last slot $T[m]$. Therefore, to find the element with the largest key, we must start our search from $T[m]$, work our way towards $T[1]$, and return the index of the first non-NULL slot. Otherwise, we return 0 if all the slots are NULL.

The worst case is when all the slots are NULL, i.e., the direct-address table T is empty. Hence, the while loop will execute for m iterations. Therefore, the running time of finding the maximum element is $\mathcal{O}(m)$. In other words, the worst case running time is $\Theta(m)$.

Algorithm 2 implements searching for the maximum element in a direct-address table.

```
1: procedure DIRECT-ADDRESS-TABLE-MAX(T)
2:     i = T.size
3:     while i > 0 do
4:         if T[i] ≠ NULL then
5:             return i
6:         i = i − 1
7:     return 0
```

Problem 4.2. Hashing

1) Demonstrate what happens when we insert the keys $< 14, 37, 10, 33, 47, 6, 21, 8, 55 >$ into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \mod 9$.

Solution: There are three collisions as illustrated in Figure 1; 10 collides with 37 in slot 1 ($10 \equiv 37 \equiv 1 \mod 9$)[1], 6 collides with 33 in slot 6 ($6 \equiv 33 \equiv 6 \mod 9$), and 55 collides with 10 and 37 in slot 1 ($55 \equiv 10 \equiv 37 \equiv 1 \mod 9$).

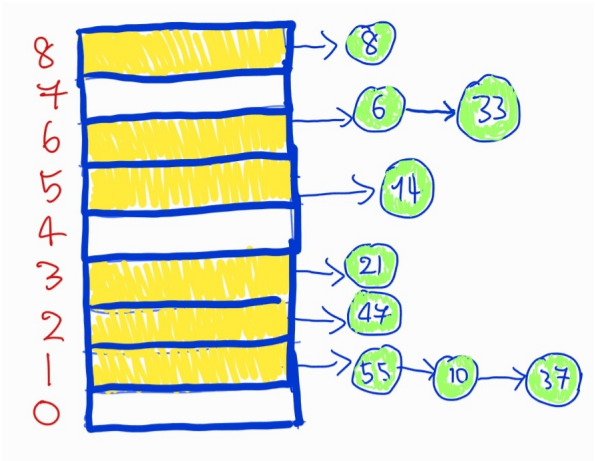

Figure 1: Inserting the keys $< 14, 37, 10, 33, 47, 6, 21, 8, 55 >$ into an initially empty hash table with chaining

2) Explain why the delete operation for hash tables does not work in $\mathcal{O}(1)$ time when separate chaining is implemented using singly-linked lists instead of doubly-linked lists.

Answer: Suppose there are three keys hashing to slot i as shown in Figure 2, and we would like to delete the element x from the hash table. To delete x, we must update $y.next$ to make it point to z as indicated by the dotted red arrow in Figure 2. Given a reference to the element x, we can access only z via $x.next$, but we have no direct access to y. Hence, it is necessary that we traverse the chain of slot i starting from the first element until the element immediately preceding x, which happens to be y in this particular example.

In the worst case, we need to traverse the entire chain to delete a given element x. Therefore, the running time of a deletion is $\mathcal{O}(n_i)$, where n_i is the length of the chain of slot i the element x hashes to, i.e., $h(x.key) = i$.

[1] $a \equiv b \mod n$ reads a is congruent to b modulo n; two integers a and b are said to be congruent modulo n if they have the same remainder when divided by n.

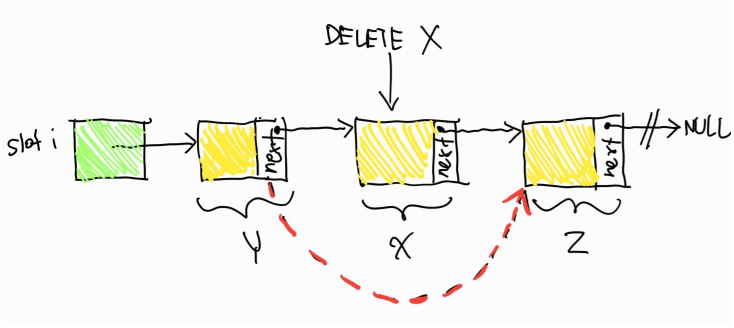

Figure 2: Deleting x from slot i

3) Suppose we come up with a hash function for computing the hash values of names using the following scheme.

$h(S) = \sum_{i=0}^{S.length-1} ASCII(S[i])$, where S is a string of characters of length $S.length$ and the function $ASCII$ returns the ASCII value of a character.

What is wrong with this hash function?

As a hint, consider the following situation when we use it to hash these three names,"Lee Chin Tan","Chen Le Tian" and "Chan Tin Lee".
Answer: The hash function does not take into account of the positions of the characters that appear in a name. Therefore, names that are made of the same set of characters, each of which is repeated for the same number of times, will hash to the same hash value as demonstrated by the three names provided above.

A better hash function would be:

Algorithm 3 implements a hash function for strings.

```
1: procedure HASH(S[1...n],m,c)
2:     sum = 0
3:     for i = 1 → n do
4:         sum = sum · c + S[i]
5:     return sum mod m
```

This hash function shifts the sum after each character by c[2].

Problem 4.3. Open Addressing

1) Suppose we want to delete an element with key k. What is wrong with deleting the element by simply marking the slot as empty? You may provide a counter example of situations when doing this will cause searching to behave incorrectly.

Answer: Suppose there is a hash table that uses open addressing to resolve collisions via **linear probing** and there are already elements x and y already present in slots i and $i+1$, respectively, as shown in Figure 4. Suppose further that we would like to insert a new element z into the hash table, provided that $h(x,0) = h(y,0) = h(z,0)$ for probe number 0. Therefore, the first and the second probe will result in a collision with x and y, respectively. On the third probe, z is successfully inserted into slot $i+2$ as it is marked as NULL, i.e., empty.

[2]Java's String.hashCode() uses 31 as the value of c.

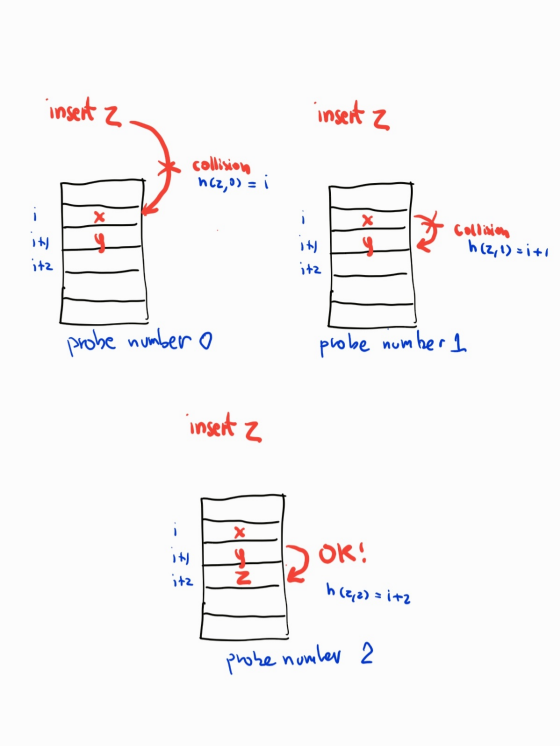

Figure 3: Inserting a new element z into a hash table with elements x and y already present where $h(z.key, p) = h(x.key, p) = h(y.key, p)$

Let us now consider what will happen if we delete y by marking slot $i + 1$ as NULL and search for z immediately afterwards. On the first probe, slot i is checked, but it is occupied by x. The second probe is performed at slot $i + 1$ and NULL is found, indicating z is not in the hash table, which is **incorrect** !!!.

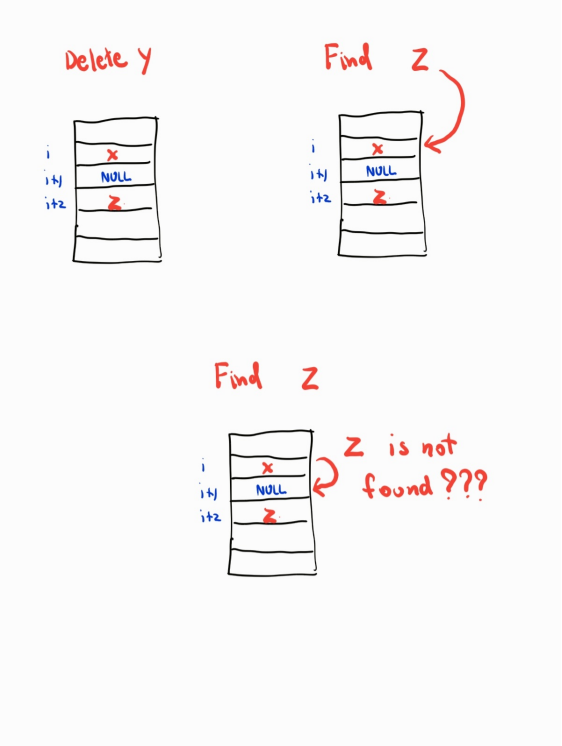

Figure 4: Deleting y by marking slot $i+1$ as empty and searching for z immediately afterwards

Generally speaking, deleting an element by simply marking the corresponding slot as empty can cause a search for an element that was inserted after the deleted element to fail incorrectly. Note that the element being searched for was inserted where it is as a result of collisions with all the existing elements that hash to the same slots on the earlier probes.

2) Write pseudocode for HASH-DELETE and modify HASH-INSERT to handle the special value DELETED.

Solution:

Algorithm 4 implements deletion via open addressing.

```
1:  procedure HASH-DELETE(T,k)
2:      p = 0
3:      do
4:          i = h(k, p)
5:          if T[i] == k then
6:              T[i] = DELETED
7:              return i
8:          else
9:              p = p + 1
10:     while T[i] == NULL ∨ p == m
11:     ERROR: KEY NOT FOUND
```

We need to modify HASH-INSERT so that it can insert a new element into a slot marked as NULL or DELETED as shown in lines $5-7$ in Algorithm 5, where any slot marked as DELETED is treated equally the same as one marked as NULL when it comes to inserting a new element.

Note that HASH-SEARCH does not need to be modified since the while loop will simply ignore any

Algorithm 5 implements insertion via open addressing.

```
1:  procedure HASH-INSERT(T,k)
2:      p = 0
3:      do
4:          i = h(k, p)
5:          if T[i] == NULL ∨ T[i] == DELETED then
6:              T[i] = k
7:              return i
8:          else
9:              p = p + 1
10:     while i == m
11:     Error: Hash Table Full
```

slot marked as DELETED.