CS5800: Algorithms — Iraklis Tsekourakis

Homework 5 Name: Avinash R. Arutla

1. (20 points)

Solution:

```
ENQUEUE-SL(L,x):
    if L.tail == NIL:
        // There are no elements in the linkedlist
        L.head = x
        L.tail = x

    else:
        // We update the tail of the linkedlist
        L.tail.next = x
        L.tail = x

    x.next = NIL
    return L


DEQUEUE-SL(L):

    if L.head == NIL:
        //There is no element to remove from the linkedlist
        return "No element present"

    else if L.head == L.tail:
        // There is only a single element present in the LL
        x = L.head
        L.tail = NIL
        L.head = NIL
        return x

    else:
        // We remove the HEAD, and the next element is made the HEAD of the LL
        x = L.head
        L.head = L.head.next
        return x
```

2. (15 points)

Solution:

To address the challenge of implementing a direct address table, where the keys not be distinct is a problem becuase direct address table's concept is to directly index elements into an array so that it can provide constant time operations. When the keys here are not distinct it will pose a problema and increase the complexity of the operation.

To fix this, we can use the combination of direct address table with chaining. Now here, every slot in the direct address table instead of holding just one element contain an access to the HEAD of a linkedlist which will be used for chaining. Through this method, we can accomodate non unique keys and their satellite data.

The classic INSERT, SEARCH, DELETE operations wont be affected in any manner, and their approach is as follows

INSERT

To insert an element into the direct address table, we use the key of the element to compute the slot in the direct address table, we then add the element to the next available slot in the linkedlist. This operation is O(1) because it takes a single operation to insert the element.

SEARCH

To search for an element in the direct address table, we use the element to find the index in the table, and assuming that all the elements in the linkedlist are uniform or near uniform, we use simple traversal to find the element in O(1) time.

DELETE

For deletion here, we can use the pointer of the element, not the key and directly delete it from the linkedlist. Here, since the deletion operation is carried out by directly referencing the element's pointer, the complexity is just O(1) here.

3. (20 points)

Solution:

Assuming here, we are using a hash function h, to hash n distinct keys into an array T of length m.

Since here we are using independent uniform hash functions, there might be a chance that an element may be allocated an index position where another element already resides. This causes collisions with other elements in the hash table.

Number of ways to take 2 elements from a total of n elements is given by permutation of n_r^p. Since any element can be hashed into a hash table with uniform probability, the probability that two elements can be inserted into the same index of the hash table of length m is given by $\frac{1}{m}$.

$$E[X] = \frac{n}{2} \cdot \frac{1}{m}$$
$$E[X] = \frac{n(n-1)}{2} \cdot \frac{1}{m}$$
$$E[X] = \frac{n^2 - n}{2m}$$

Therefore the random variable $E[X]$, clarifies the probability that such an event might take place.

4. (18 points)

Solution:

Here we have a hash table with 9 slots. The hashing function that was given to us was, $h(k) = k|9|$, where the calculation are as follows.

- **Key 5:**

  $$h(5) = 5 \mod 9 = 5$$

  Key 5 is placed in slot 5.

- **Key 28:**

  $$h(28) = 28 \mod 9 = 1$$

  Key 28 is placed in slot 1.

- **Key 19:**

  $$h(19) = 19 \mod 9 = 1$$

  Key 19 collides with Key 28 and is chained in slot 1.

- **Key 15:**

  $$h(15) = 15 \mod 9 = 6$$

  Key 15 is placed in slot 6.

- **Key 20:**

  $$h(20) = 20 \mod 9 = 2$$

  Key 20 is placed in slot 2.

- **Key 33:**

  $$h(33) = 33 \mod 9 = 6$$

  Key 33 collides with Key 15 and is chained in slot 6.

- **Key 12:**

$$h(12) = 12 \mod 9 = 3$$

Key 12 is placed in slot 3.

- **Key 17:**

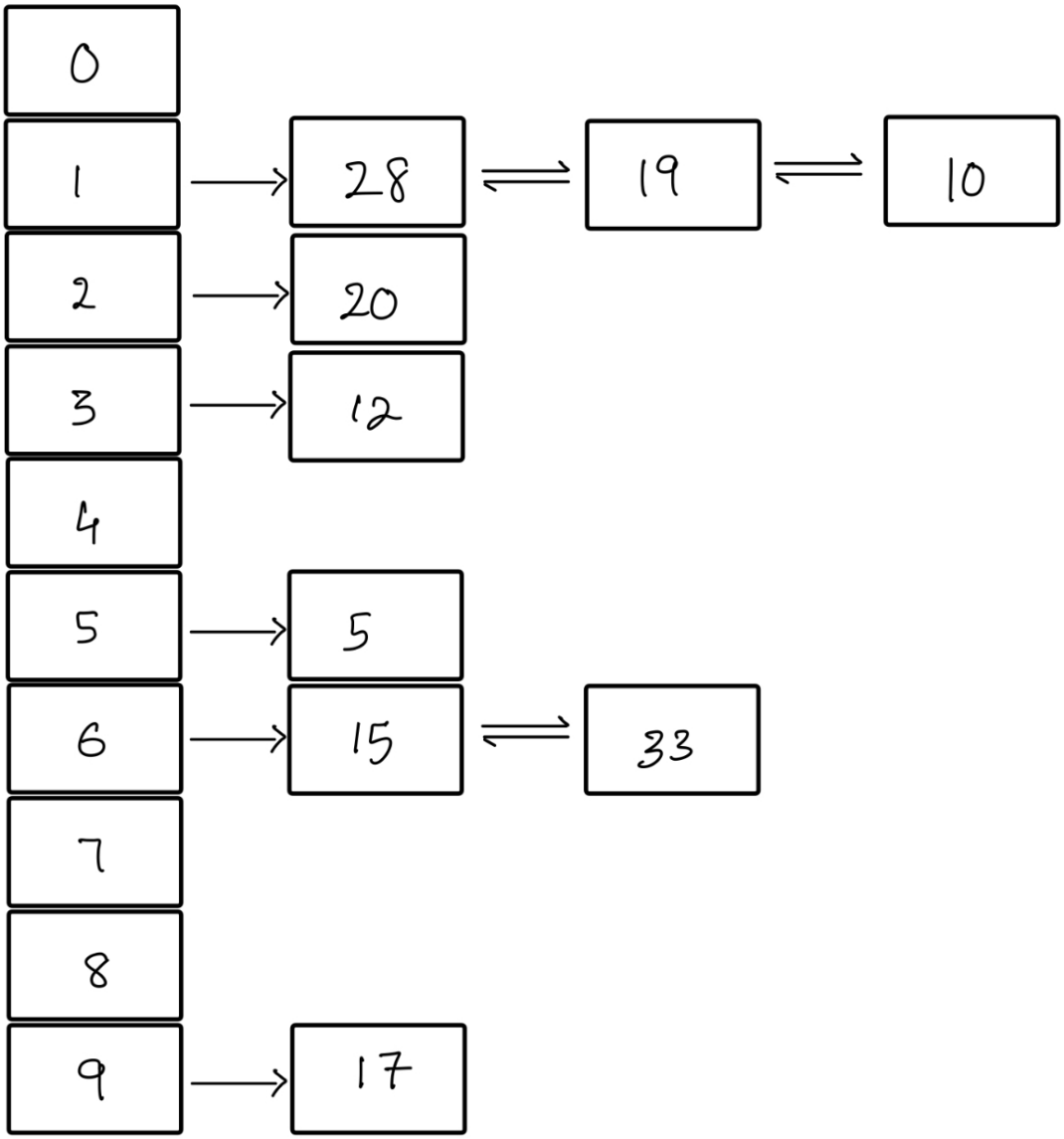
$$h(17) = 17 \mod 9 = 8$$

Key 17 is placed in slot 8.

- **Key 10:**

$$h(10) = 10 \mod 9 = 1$$

Key 10 collides with Keys 28 and 19, and is chained in slot 1.



5. (14 points)

Solution:

0	1	2	3	4	5	6	7	8	9
9	18		12	3	14	4	21		

The mod function given to us here is $h(k) = k|9|$.

for h(9), 9 mod 9 = 0. So we place the element in index 0 according to the rules, here the element is already present in index 0, so its safe to assume it was placed there because of the same reason

for h(18), 18 mod 9 = 0. Here we already know that 9 was occupying index 0, so in consideration of open addressing and linear probing, 18 was added to index 1 as seen from the diagram.

for h(12), 12 mod 9 = 3. So as we can see from the diagram, since the mod results in 3, we add 12 element to the index 3.

for h(3), 3 mod 9 = 3. We are already aware that 12 resides at index 3, so in considerating of open addressing and linear probing, we place element 3 at the next available slot which is slot 4.

for h(14), 14 mod 9 = 5. We can see that that slot 5 is empty, so in consideration we place element 14 at slot 5.

for h(4), 4 mod 9 = 4, as we already know that slot 4 is occupied by 3, and the next slot which is slot 5 is occupied by 14, so in consideration of open addressing and linear probing, we place element 4 in the next available slot which is slot 6.

for h(21), 21 mod 9 = 3, as we can see from the diagram that slot 3,4,5,6, are consecutively occupied by other elements, the next free slot is slot 7, so we place element 21 in slot 7 due to open addressing and linear probing.

6. (20 points)

Solution:

The non recursive solution for a inorder tree walk is as follows

```
// A stack will be used to track the nodes, so that the tree runs in O(n) order.

ITERATIVE-INORDER-TREE-WALK(root):
```

```
S = new_stack()
current = root() // a new node which is replicated for traversing

while not(S.isEmpty()) or current != NIL:
    // if current== nil, we reached the end of the node
    // if S.isEmpty() is true, that all the elements have been traversed.

    if current!= NIL:

        PUSH(s,current)
        current = current.left

    else:
        // we reached the end of tree path fdrom the traversal from the root,
        // and the element at the top of thestack is the lowest

        current = POP(s)
        print(current)

        // moving to the right to find the next smallest element.
        current = current.right
```