Algorithms

Assignment 5

Ashwin Sathiyanarayanan
002227609

Question 1:

```
def Enqueue(L, newNode):
    node = null
    if L.head == Null:
        L.head = newNode
    else:
        node = L.tail
        node.next = newNode
    newNode.next = Null
    L.tail = newNode

def Dequeue(L):
    if L.head == None:
        error "Queue Underflow"
    node = L.head.next
    delNode = L.head
    if L.head == L.tail:
        L.tail = Null
    L.head = node
    delNode.next = Null
    return delNode.key
```

Explanation:

Enqueue Method:

In order to implement a queue's Enqueue method using a Linked list, first we have to check whether the queue is empty or not. If the queue is empty, we just place the new node at the head of the queue, set the next of the new node to null, and set the tail of the queue to the new node. If the queue is not empty, then we set the auxiliary node variable to the tail of the queue. Then we set the next of the auxiliary node(tail of the queue) to the new node. Then, whatever the queue’s state is, we set the next of the new node to null and set the tail of the queue to the new node. This method takes only O(1) time as there are no loops or recursions.

Dequeue Method:

In order to implement a queue's Dequeue method using a Linked list, first we have to check if the queue has elements or not. If the queue is empty, then we throw a Queue Underflow error. If the queue is not empty, we proceed to remove the head of the queue. We first set an auxiliary node variable to the next node of the head of the queue. Then we set a new variable named delNode, the node to be deleted, to the head of the queue. Then we check if the head of the queue is the same as the tail of the queue, if it is, then we set the tail of the queue to null. Then we set the head of the Linked List (queue) to the auxiliary node (next to the head of the queue). Then we set the next of the delNode to null and return the delNode. This method takes only O(1) time as there are no loops or recursions.

Question 2:

In order to implement a direct-address table in which the keys of the stored elements do not need to be distinct and the elements can have satellite data, instead of using the traditional key and data pair, we can have a pair of keys and a doubly linked list to store satellite data. Here, even if the keys are the same, the elements are added to the doubly linked list of that particular key. When it comes to the basic dictionary operations, Insertion takes O(1) time because whenever we get an element to insert, we just add the element to the start of the doubly linked list of that particular key of the element we receive. So Insertion takes O(1) time. Deletion operation also takes O(1) time because whenever we want to delete an element, we just delete the element from the doubly linked list as we will get the pointer to the element to be deleted as an argument. This whole deletion process only takes O(1) time. Now, regarding the Search method, this also takes O(1) time. This is because, whenever we want to get an element with a particular key, we just return the first element from the doubly linked list of that particular key. This is because when we want to search for an element with a particular key, we just want to find any element that has the key we want. So we can return any element from the doubly linked list of that particular key. Therefore, we can just return the first element from the doubly linked list, which takes O(1) time. Therefore, by using a key and a doubly linked list as a pair, we can store elements with duplicate keys, while also preserving the constant running time of the basic dictionary operations.

Question 3:

If we use a hash function h to hash n distinct keys into an array T of length m using independent uniform hashing, we can find the expected number of collisions using the theorem 11.2 in the textbook. Under uniform hashing, we assume that any given element is equally likely to hash into any of the m slots in the hash table. This means that an element may collide with all the other elements except itself. Using this, let us find the expected number of collisions:

Let $k_i = x_i.key$ and $k_j = x_j.key$.
If the two keys collide with each other, it implies that:
$hash(k_i) = hash(k_j)$
Therefore, the probability of two keys colliding with each other is:
$Pr\{hash(k_i) = hash(k_j)\}$

We know that under simple uniform hashing, the probability of two keys colliding is $1/m$ where m is the total number of elements in the array.

Therefore,
$Pr\{hash(k_i) = hash(k_j)\} = 1/m$
But as mentioned above, in simple uniform hashing, we assume that any given element is equally likely to hash into any of the m slots in the hash table.

Therefore, $hash(k_i)$ can collide with all the other keys in the hash table. So, the probability of collision $hash(k_i)\ with\ hash(k_j)$ is multiplied by $n - i$. We multiply by $n - i$ because n is the number of distinct keys and when considering the number of keys that may collide with the key at position i, only the keys that come after i, (i.e.) j > i, can collide with k_i.

Therefore, the probability of one collision occurring is given as:

$$Probability\ of\ 1\ collision = (n - i)/m$$

Now we have found the probability of 1 collision happening, (i.e.)

$(n - i)/m$

But, to find the expected number of collisions while hashing n distinct keys into an array, we have to find the summation of the probability of 1 collision from $i = 1 \ to\ n$, where n is the number of distinct keys.

Therefore, the total number of expected collisions is,

$$Expected\ Number\ of\ Collisions = \sum_{i=1}^{n} (n - i)/m$$

$$= (1/m) \sum_{i=1}^{n} (n - i)$$

$$= (1/m).[n^2 - ((n)(n + 1))/2]$$

$$= (1/m).[(2n^2 - n^2 - n)/2]$$

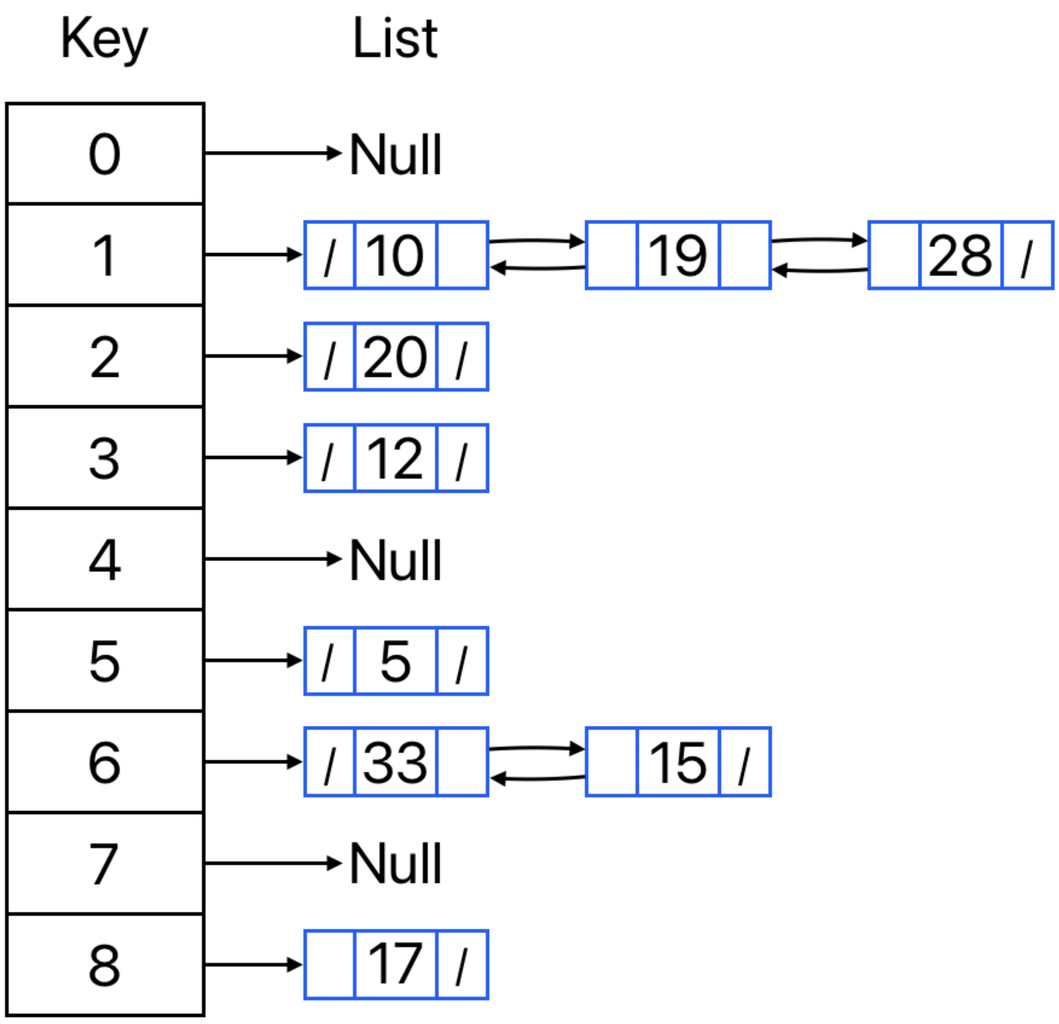
$$= (1/m).[(n^2 - n)/2]$$

$$= (n^2 - n)/2m$$

Therefore, we have found the expected number of collisions when we use a hash function h to hash n distinct keys into an array T of length m using independent uniform hashing as,

$$expected\ number\ of\ collisions = (n^2 - n)/2m$$

Question 4:



When inserting the keys 5, 28, 19, 15, 20, 33, 12, 17, and 10 using the hash function h(k) = k mod 9, the following takes place for each element:

I. First, let's see what the hash function does. The hash function takes in the element and calculates the mod of that element with 9 to calculate the key to which the element should be segregated.
II. Now when we first get 5 as the input, we calculate the hash function of 5. 5 mod 9 will give us 5. Therefore, element 5 will be inserted at the front of the doubly linked list of key 5.

III. Now when we get 28 as the input, 28 mod 9 will give us 1. Therefore, element 28 will be inserted at the front of the doubly linked list of key 1.
IV. Now when we get 19 as the input, 19 mod 9 will give us 1. Therefore, element 19 will be inserted at the front of the doubly linked list of key 1. Note that we already have 28 as the head of the doubly linked list at key 1. So, 19 will be prepended into the doubly linked list(in front of 28).
V. Now when we get 15 as the input, we calculate the hash function of 15. 15 mod 9 will give us 6. Therefore, element 15 will be inserted at the front of the doubly linked list of key 6.
VI. Now when we get 20 as the input, we calculate the hash function of 20. 20 mod 9 will give us 2. Therefore, element 20 will be inserted at the front of the doubly linked list of key 2.
VII. Now when we get 33 as the input, 33 mod 9 will give us 6. Therefore, element 33 will be inserted at the front of the doubly linked list of key 6. Note that we already have 15 as the head of the doubly linked list at key 6. So, 33 will be prepended into the doubly linked list(in front of 15).
VIII. Now when we get 12 as the input, we calculate the hash function of 12. 12 mod 9 will give us 3. Therefore, element 12 will be inserted at the front of the doubly linked list of key 3.
IX. Now when we get 17 as the input, we calculate the hash function of 17. 17 mod 9 will give us 8. Therefore, element 17 will be inserted at the front of the doubly linked list of key 8.
X. Now when we get 10 as the input, 10 mod 9 will give us 1. Therefore, element 10 will be inserted at the front of the doubly linked list of key 1. Note that we already have 19 and 28 in the doubly linked list at key 1. So, 10 will be prepended into the doubly linked list(in front of both 19 and 28).
XI. All the keys, that do not have any elements mapped to them, just point to the Null value.

Question 5:

The final hash table after inserting all the elements is:

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

When using open addressing, the elements are added directly to the hash table without any pointers to the actual data.
Linear probing is a method to overcome collisions in a hash table. This method works as:

1. First calculates the key where the element should be added using the hash function.
2. If that key is empty, the element is just added there.
3. If that is already occupied, then the element is added to the next empty space found.

Now, in our problem, the elements must be added in the following order for the final hash table to appear as given in the question,

Order of insertion: 9, 18, 12, 3, 14, 4, 21

Let's prove this by seeing the state of the hash table after each insertion.

I. The initial state of the hash table:

0	1	2	3	4	5	6	7	8

II. Element to be inserted: 9
To calculate the key of the element, we use the hash function h(k) = k mod 9. 9 mod 9 is 0. Therefore, 9 should be inserted at index 0. Since index 0 is empty, there is no collision. So we can directly insert 9 at index 0.

0	1	2	3	4	5	6	7	8
9								

III. Element to be inserted: 18
To calculate the key of the element, we use the hash function h(k) = k mod 9. 18 mod 9 is 0. Therefore, 18 should be inserted at index 0. But the index 0 is already occupied by element 9. So according to linear probing, we insert 18 at the next empty space we find, in this case, index 1. Therefore, collision is avoided here.

0	1	2	3	4	5	6	7	8
9	18							

IV. Element to be inserted: 12
To calculate the key of the element, we use the hash function h(k) = k mod 9. 12 mod 9 is 3. Therefore, 12 should be inserted at index 3. Since index 3 is empty, there is no collision. So we can directly insert 12 at index 3.

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

V. Element to be inserted: 3
To calculate the key of the element, we use the hash function h(k) = k mod 9. 3 mod 9 is 3. Therefore, 3 should be inserted at index 3. But index 3 is already occupied by element 12. So according to linear probing, we insert 3 at the next empty space we find, in this case, index 4. Therefore, collision is avoided here.

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

VI. Element to be inserted: 14
To calculate the key of the element, we use the hash function h(k) = k mod 9. 14 mod 9 is 5. Therefore, 14 should be inserted at index 5. Since index 5 is empty, there is no collision. So we can directly insert 14 at index 5.

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

VII. Element to be inserted: 4
To calculate the key of the element, we use the hash function h(k) = k mod 9. 4 mod 9 is 4. Therefore, 4 should be inserted at index 4. But index 4 is already occupied by element 3. So according to linear probing, we insert 4 at the next empty space we find, in this case, index 6. Therefore, collision is avoided here.

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

VIII. Element to be inserted: 21
To calculate the key of the element, we use the hash function h(k) = k mod 9. 21 mod 9 is 3. Therefore, 21 should be inserted at index 3. But index 3 is already occupied by element 12. So according to linear probing, we insert 21 at the next empty space we find, in this case, index 7. Therefore, collision is avoided here.

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

Finally, we have attained the hash table given to us in the question. Therefore, the order of insertion of elements will be **9, 18, 12, 3, 14, 4, 21**.

Question 6:

```
Iterative_Inorder_Tree_Walk(root):
    x = root
    stack = [ ]    // Let this be our auxiliary stack
    while True:
        if x $\neq$ Null:
            stack.push(x)
            x = x.left
        else if len(stack) $>$ 0:
            x = stack.pop()
            print x.key
            x = x.right
        else:
            break
```

Explanation:

1. To convert the traditional recursive in-order tree walk algorithm into an iterative one, we can use a stack. This is because, during an in-order traversal, we must print the entire left lineage of the tree before moving to the right tree. A stack allows us to do this as we can push the elements into the stack till we reach the end of the left lineage and pop the elements from there. The LIFO property of the stack ensures that we traverse the tree in the in-order method.
2. First, we get the root of the tree as an argument for our method. Since we start from the root of our tree and go down, we will assign the root node to x. We will also declare the stack that we will use to keep track of the elements.
3. Now, we will start a while loop which runs infinitely until broken explicitly using a break statement that occurs inside the loop when we have traversed through all the elements of the tree.
4. Inside the loop, we will first check if the current node x is not equal to null. If node x is not null, we push the element in the node x inside our stack and set node x to the node left of x. In this way, we traverse

all the elements on the left side of the tree first before we see any of the elements that have a right-child relationship.

5. Then, if x is none, it means we have reached the bottom of the left side of that subtree. So now we check if our stack has elements in them. If true, it means that we should start popping the elements from our stack, print the popped element, and move to the right side of the node simultaneously. The stack's LIFO property takes care of the printing of elements in an in-order way.
6. Finally, if both the above conditions are false (i.e.) current node x is null, and also our stack is empty, it means that we have traversed through all the elements in our tree. So now we will break the loop and stop the function.