Solution Mid Term Fall 2023

Question 1: MCQs

1. The answers is (c). 6,5,2,11,7,4,9,5,2

2. The answer is (c) O(n).

The worst-case scenario for locating the largest element in a list of N elements is when the elements are in reverse order. In this case, we would have to compare the first element to all N-1 remaining elements. This would take N comparisons, which is a linear time complexity of O(N).

The best-case scenario is when the largest element is the first element in the list. In this case, we would only need to make one comparison. However, the best-case scenario is not guaranteed, so we must consider the worst-case scenario when calculating the runtime complexity.

Therefore, the runtime complexity for locating the largest element in a list of N elements is O(N).

3. The answer is (b) It traverses in an increasing order.

Inorder traversal of a binary search tree visits all the nodes of the tree in an increasing order. This is because the left subtree of a node contains all the nodes that are less than the node, and the right subtree contains all the nodes that are greater than the node. Therefore, when we traverse the tree in inorder, we will visit the nodes in increasing order.

4. The worst-case time complexity of the function get_answer is O(n^2) (d)

The inner for loop iterates from i to len(a), which is a linear function of n. The outer for loop iterates over the entire list, so it also has a linear time complexity. Therefore, the overall time complexity is O(n * n) = O(n^2).

5. The answer is (c) O(n).

The function fA iterates over the list lst and performs the following operations:

1. Removes the first element from the list.
2. Appends the removed element to the end of the list.

The first operation takes constant time, since the first element of the list can be accessed directly. The second operation takes linear time, since the list must be traversed to find the first element. Therefore, the overall time complexity of the function is O(n).

6. In a doubly-linked list, removing the first element (by index) involves updating the pointers to reassign the connections of the nodes. The worst-case runtime complexity for removing the first element in a doubly-linked list of N elements is O(1) i.e. O(1) (option a)

7. The answer is (a) s.append('!').

The string s is immutable, so we cannot append an element to it. This will result in an TypeError exception being raised.

The other options will not raise any exceptions.

- Option (b) will delete the elements from the list l from index 2 to 4, inclusive.
- Option (c) will iterate over the tuple t.
- Option (d) will return a slice of the tuple t, starting from index 1.

8. The specialty about the inorder traversal of a binary search tree is: (b) It traverses in an increasing order.

9. The traversal of a Binary Search Tree that would result in a sorted list of its elements is: (a) Inorder

10. The traversal of a Binary Search Tree that visits the root node first, followed by its left and right children is: (b) Preorder

Question 2:

Function sum_of_n(n):

This function utilizes a loop to sum the numbers from 1 to n. The loop iterates through all numbers from 1 to n and accumulates the sum.

```
def sum_of_n(n):
    the_sum = 0
    for i in range(1, n+1):
        the_sum = the_sum + i
    return ("Sum of {} numbers is {}".format(n, the_sum))
```

The loop runs n times to calculate the sum. Therefore, the time complexity of this function is linearly proportional to the value of n. The running time can be expressed as **O(n)** because as n grows, the number of iterations directly increases linearly.

Function sum_of_n2(n):

This function employs a direct mathematical formula to find the sum of numbers from 1 to n. It uses the formula for the sum of an arithmetic series: (+1)22n·(n+1)

```
def sum_of_n2(n):
    sum = (n * (n + 1)) / 2
    return ("Sum of {} numbers is {}".format(n, sum))
```

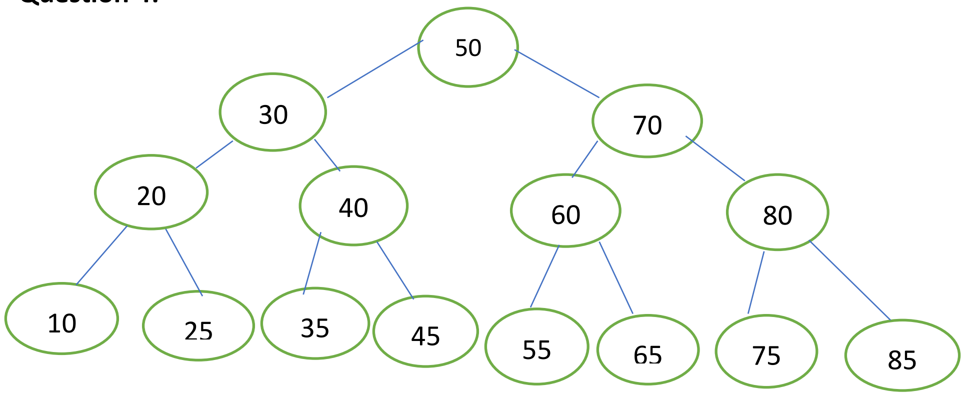
The time complexity of this function is constant, denoted as (1) **O(1).** Regardless of the value of n, the function uses a simple mathematical formula to calculate the sum. It doesn't involve any loops or iterative operations that increase with the input size. The time taken to execute remains constant, making it independent of the input size n.

Question 3. #

1. Define the function signature.

```
# 2. Check if the linked list is empty.

# 3. Start from the head node.

# 4. Loop through the linked list.

# 5. Check if the current node's data matches the target number.

# 6. If found, return True.

# 7. Move to the next node in the linked list.

# 8. If the end of the list is reached without finding the number, return False.

def has_number_in_doubly_linked_list(head, number):
    if head is None:  # Use the correct check for an empty list.
        return False  # Use the correct value to indicate the number is not found.
    current = head
    while current is not None:
        if current.data == number:
            return True
        current = current.next  # Move to the next node.
    return False
```

Question 4:



Question 5:

```
def _find_largest(self, node):
    current = node
    while current.right:
        current = current.right
    return current

def _find_second_largest(self, node):
    if node is None or (node.left is None and node.right is None):
        raise ValueError("BST must have at least two nodes.")

    current = node

    while current:
        # Case 1: If the largest node has a left subtree
        if current.left and not current.right:
            return self._find_largest(current.left)

        # Case 2: If the largest node is the parent of the largest
        if current.right and not current.right.left and not current.right.right:
            return current

        current = current.right

# Assigning the function to the class method
Node._find_second_largest = _find_second_largest

# Example usage:
# Assuming 'root' is the reference to the root node of the BST
# Call the function as follows:
# second_largest = root._find_second_largest(root)
```

This function first checks if the tree has at least two nodes, then traverses the tree to find the second-largest element. It covers various cases, including when the largest node might have a left subtree or when the largest node is the parent of the largest.

Question 6:

To find the Lowest Common Ancestor (LCA) of two nodes in a Binary Search Tree (BST), the following steps can be followed:

Step 1: Begin at the root of the BST.

Step 2: Compare the values of the two nodes with the current node value:

- If both nodes' values are less than the current node's value, move to the left child.
- If both nodes' values are greater than the current node's value, move to the right child.
- If one value is greater and the other is smaller than the current node's value, then the current node is the LCA.

Step 3: Repeat this process, traversing the tree from the root towards the required nodes until you find the LCA.

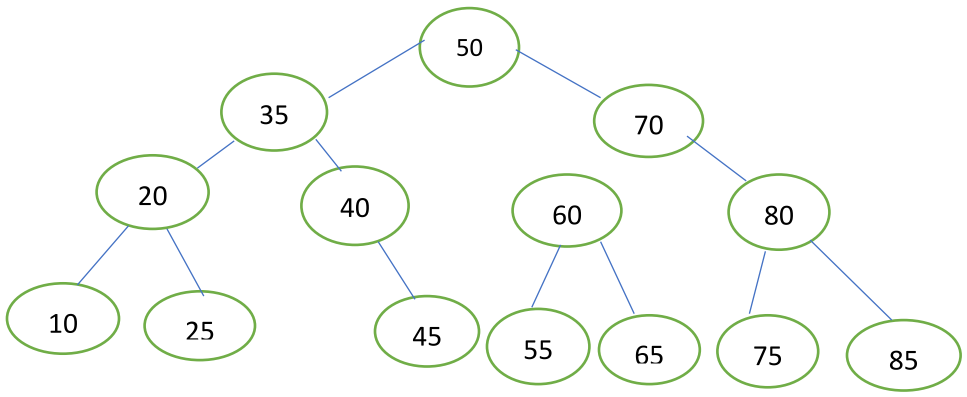
- If the values of the two nodes are in different subtrees (one in the left subtree, the other in the right), then the current node is the LCA.

Step 4: Continue the steps until the LCA is found, considering the relationship between the values of the two nodes and the current node's value.

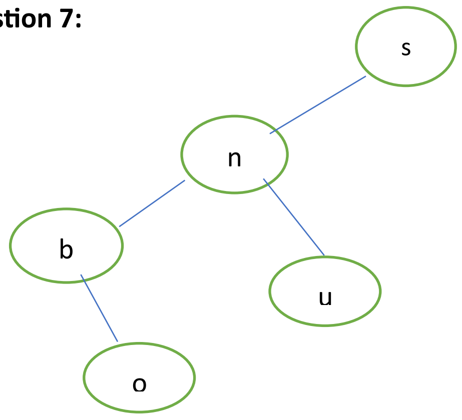
Step 5: The last node that follows the criteria of being the "split" node—where the values of the two nodes are in different subtrees—is the Lowest Common Ancestor.

Step 6: Return this identified node as the LCA of the given nodes in the BST

Delete the number 30 from the BST and draw the resulting tree.



Question 7:



Inorder: bonus

Postorder: obuns