Student ID:
Name:
Signature:

COMP 203 DATA STRUCTURES AND ALGORITHMS

QUIZ 2 Date: 09.01.2024 10:30 am

Total: 100 points

1. Stack and SLL (Total: 55 points)

a. class Node: //2pt

```
   char data
   Node next

   constructor(data): //3pt
      this.data = data
      this.next = null
//end of node class

class SinglyLinkedList: //2pt
   Node head

   constructor()://3pt
      this.head = null
//end of sll

class SSLStack: //2pt
   SinglyLinkedList stack

   constructor(): //3pt
      this.stack = new SinglyLinkedList()
//end of sslstack
```

b. method SSLStack.push(addedValue): //Total= 10pt

```
   newNode = new Node(addedValue) // creating the new node to push 1 pt

   if this.stack.head is null: //check if it is null  and adding at the head as the first node 2pt
      this.stack.head = newNode
```

Student ID:
Name:
Signature:

```
        else:
            current = this.stack.head
            while current.next is not null: //traversing the stack 6pt
                current = current.next
            current.next = newNode

        return this.stack.head //returnin the head node 1pt
```

//if adds at the head only 2pt

c. Explanation of the Big-O complexity for the "Node push(char addedValue)" method: //5pt if there is no explanation only 2pt

- The time complexity of the push operation is O(n), where n is the number of elements in the stack. This is because it involves traversing the linked list to find the end before appending the new node.

d. method SSLStack.pop()://total= 10pt

```
    if this.stack.head is null: //check if Stack is empty  1pt
        return null
    else:
        if this.stack.head.next is null: //check if Stack is has only one node  and remove it 2pt
            poppedValue = this.stack.head.data
            this.stack.head = null
        else:
            current = this.stack.head
            while current.next.next is not null: //traverse the stack 4pt
                current = current.next
            poppedValue = current.next.data //remove from at the end 2pt
            current.next = null

        return poppedValue //returning the deleted value 1pt
```

Student ID:
Name:
Signature:

e. Explanation of the Big-O complexity for the "char pop()" method: //5pt if there is no explanation only 2pt

- The time complexity of the pop operation is O(n), where n is the number of elements in the stack. This is because it involves traversing the linked list to find the second-to-last node before removing the last node.

f. function testPushPop()://total=10pt

```
stack = new SSLStack() //creating a new stack 1pt

// Push elements onto the stack 3pt
stack.push('A')
stack.push('B')
stack.push('C')
stack.push('D') //testing push 3pt
// Print the stack after push operations
printStack(stack)

// Pop an element from the stack
poppedValue = stack.pop() //testing pop 3pt

// Print the stack after pop operation
printStack(stack)
// Print the popped value
print("Popped Value: " + poppedValue)
```

Student ID:
Name:
Signature:

2. Binary Search Tree and Node (Total: 45pt)

1. class Node: //2pt

```
    int data
    Node left
    Node right

    constructor(data): //3pt
        this.data = data
        this.left = null
        this.right = null
//end of Node class
```

2. class BinarySearchTree: //2pt

```
    Node root

    constructor(): //3pt
        this.root = null
```

3. method BinarySearchTree.insert(Node root, int value): //total=10pt

```
    if root is null: //check if it is null and create the first one 2pt
        return new Node(value)

    if value < root.data: //recursive part 7pt
        root.left = insert(root.left, value)
    else if value > root.data:
        root.right = insert(root.right, value)

    return root //returning root 1pt
```

3.1 Time Complexity of insert method: The time complexity of the insert method is O(h), where h is the height of the tree. In the worst case, for an unbalanced tree, the height can be n (the number of nodes), resulting in O(n) time complexity. However, for a balanced tree, the height is log(n), leading to O(log n) time complexity. //2pt

Student ID:
Name:
Signature:

4. method BinarySearchTree.inOrderTraversal(Node root): //Total: 10pt

```
if root is not null:  // base case 2pt
    inOrderTraversal(root.left) // recursive part 8 pt
    print(root.data)
    inOrderTraversal(root.right)
```

4.1 Time Complexity of inOrderTraversal method: The time complexity of the inOrderTraversal method is O(n), where n is the number of nodes in the tree. This is because every node needs to be visited once. //2pt

5. function main(): // creating the tree and testing the methods 5+6=11 pt

```
bst = new BinarySearchTree() // creating the bst 1pt
// Create the binary search tree 4pt
bst.root = insert(bst.root, 4)
insert(bst.root, 2)
insert(bst.root, 6)
insert(bst.root, 1)
insert(bst.root, 3)
insert(bst.root, 5)
insert(bst.root, 7) //testing insert 3pt

// Print the tree using in-order traversal
print("In-order Traversal:")
inOrderTraversal(bst.root) //testing inOrderTraversal  3pt
```