COMP 203 HOMEWORK 2 SOLUTION

3. Submission: Submit your homework in **a single pdf**. Also, submit your code as **2 (two) .java files. No other file types will be accepted. You will submit only 3 files, 1 .pdf and 2 java files. DON'T USE ZIP/RAR etc. In these cases, your points will be deducted by 30%.**

(1.a. 5x3=15, b.10, c.5, d.5, e.5, f.5, g.5 =50 points)

1.

```
// Node class {//10
class Node<E> {
   E data;
   Node<E> next;

   public Node(E data) {
      this.data = data;
      this.next = null;
   }
}

// SinglyLinkedList class //5
class SinglyLinkedList<E> {
   Node<E> head;

   public SinglyLinkedList() {
      this.head = null;
   }
}

// SSLQueue class //5
class SSLQueue<E> {
   SinglyLinkedList<E> linkedList;

   public SSLQueue() {
      this.linkedList = new SinglyLinkedList<>();
   }

   // Enqueue function{  //10
   public Node<E> enqueue(E addedValue)
      Node<E> newNode = new Node<>(addedValue);

      if (linkedList.head == null) {
         linkedList.head = newNode;
      } else {
         Node<E> current = linkedList.head;
         while (current.next != null) {
            current = current.next;
            }
            current.next = newNode;
        }

        return linkedList.head;
    }

    // Dequeue function
    public E dequeue() {//5
        if (linkedList.head == null) {
            return null; // or throw an exception for an empty queue
        }

        E deletedValue = linkedList.head.data;
        linkedList.head = linkedList.head.next;

        return deletedValue;
    }
}

public class Main {
    public static void main(String[] args) { //5+5=10 (enqueue and dequeue tests)
        // Test case for enqueue
        SSLQueue<String> queue = new SSLQueue<>();
        queue.enqueue("A");
        queue.enqueue("B");
        queue.enqueue("C");
        queue.enqueue("D");
         printQueue(queue.linkedList.head);
        System.out.print("After enqueue(M): ");
        queue.enqueue("M");
        printQueue(queue.linkedList.head);

        // Test case for dequeue
        String deletedValue = queue.dequeue();
        System.out.println("\nAfter dequeue(): " + deletedValue);
        printQueue(queue.linkedList.head);
    }

    // Helper function to print the queue
    private static <E> void printQueue(Node<E> head) {
        Node<E> current = head;
        while (current != null) {
            System.out.print(current.data + "->");
            current = current.next;
        }
        System.out.println("null");
    }
}
```

c. The Big-O complexity of the **enqueue** function is O(n), where n is the number of elements in the queue. This is because, in the worst case, the function has to traverse the entire linked list to find the last node before adding a new one. //5
f. The Big-O complexity of the **dequeue** function is O(1), constant time. This is because the function only involves updating the head pointer to the next node, and it doesn't depend on the size of the queue. //5

(2. a. 5x3=15, b.10, c.5, d.5, e.5, f.5, g.5 =50 points)

```
2. // Node class //5
class Node<E> {
    E data;
    Node<E> next;

    public Node(E data) {
        this.data = data;
        this.next = null;
    }
}

// SinglyLinkedList class //5
class SinglyLinkedList<E> {
    Node<E> head;

    public SinglyLinkedList() {
        this.head = null;
    }
}

// SSLStack class //5
class SSLStack<E> {
    SinglyLinkedList<E> linkedList;

    public SSLStack() {
        this.linkedList = new SinglyLinkedList<>();
    }

    // Push function
    public Node<E> push(E addedValue) {//10
        Node<E> newNode = new Node<>(addedValue);

        if (linkedList.head == null) {
            linkedList.head = newNode;
        } else {
            Node<E> current = linkedList.head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }

        return linkedList.head;
    }

    // Pop function
    public E pop() {//5
        if (linkedList.head == null) {
            return null; // or throw an exception for an empty stack
        }

        Node<E> current = linkedList.head;
        Node<E> prev = null;

        while (current.next != null) {
            prev = current;
            current = current.next;
        }

        if (prev != null) {
            prev.next = null;
        } else {
            linkedList.head = null;
        }

        return current.data;
    }
}

public class Main {//5 +5=10 (pop and push tests)
    public static void main(String[] args) {
        // Test case for push
        SSLStack<String> stack = new SSLStack<>();
        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        printStack(stack.linkedList.head);
        stack.push("M");
        System.out.print("After push(M): ");
        printStack(stack.linkedList.head);

        // Test case for pop
        String poppedValue = stack.pop();
        System.out.println("\nAfter pop(): " + poppedValue);
        printStack(stack.linkedList.head);
    }

    // Helper function to print the stack
    private static <E> void printStack(Node<E> head) {
        Node<E> current = head;
        while (current != null) {
            System.out.print(current.data + "->");
            current = current.next;
        }
        System.out.println("null");
    }
}
```

c. The Big-O complexity of the **push** function is O(n), where n is the number of elements in the stack. This is because, in the worst case, the function has to traverse the entire linked list to find the last node before adding a new one. //5

f. The Big-O complexity of the **pop** function is O(n), where n is the number of elements in the stack. This is because, in the worst case, the function has to traverse the entire linked list to find the second-to-last node before removing the last one. //5