# COMP 203 DATA STRUCTURES AND ALGORITHMS
# HOMEWORK 1 SOLUTION

**1. a.b.e.**
```
class Node<E> {
    E data;
    Node<E> next;

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

class SinglyLinkedList<E> {
    private Node<E> head;

    public SinglyLinkedList() {
        head = null;
    }

    public Node<E> addAfter(E v, E addedValue) {
        Node<E> newNode = new Node<>(addedValue);
        if (head == null) {
            return null; // List is empty, cannot add after a value.
        }

        Node<E> current = head;
        while (current != null && !current.data.equals(v)) {
            current = current.next;
        }

        if (current == null) {
            return head; // Value 'v' not found in the list, no
changes made.
        }

        newNode.next = current.next;
        current.next = newNode;
        return head;
    }

    public E deleteAfter(E n) {
        if (head == null) {
            return null; // List is empty, nothing to delete.
        }

        Node<E> current = head;
        while (current != null && !current.data.equals(n)) {
            current = current.next;
        }
        if (current == null || current.next == null) {
            return null; // Value 'n' not found in the list or no
node after 'n'.
        }

        Node<E> deletedNode = current.next;
        current.next = current.next.next;
        return deletedNode.data;
    }
}
```

**c.** For the **addAfter(E v, E addedValue)** method, the worst-case time complexity is O(n), where 'n' is the number of nodes in the singly linked list. This is because in the worst case, we may have to traverse the entire list to find the node with the value 'v'.

**f.** For the **deleteAfter(E n)** method, the worst-case time complexity is also O(n), where 'n' is the number of nodes in the singly linked list. This is because in the worst case, we may have to traverse the entire list to find the node with the value 'n'.

**g.**
```
public static void main(String[] args) {
    SinglyLinkedList<String> list = new SinglyLinkedList<>();
    list.head = new Node<>("A");
    list.head.next = new Node<>("B");
    list.head.next.next = new Node<>("C");
    list.head.next.next.next = new Node<>("D");

    System.out.println("Original List:");
    printList(list.head);

    System.out.println("\nAdding 'M' after 'C':");
    list.addAfter("C", "M");
    printList(list.head);

    System.out.println("\nDeleting after 'C':");
    String deletedValue = list.deleteAfter("C");
    System.out.println("Deleted: " + deletedValue);
    printList(list.head);
}

public static void printList(Node<String> head) {
    Node<String> current = head;
    while (current != null) {
        System.out.print(current.data + "->");
        current = current.next;
    }
    System.out.println("null");
}
```

//This code initializes a linked list, adds 'M' after 'C', and then deletes the node after 'C'. It prints the list after each operation to verify the correctness of the functions.

**2.a.b.e**

```
 class Node<E> {
    E data;
    Node<E> prev;
    Node<E> next;

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

class DoublyLinkedList<E> {
    private Node<E> header;
    private Node<E> trailer;

    public DoublyLinkedList() {
        header = new Node<>(null);
        trailer = new Node<>(null);
        header.next = trailer;
        trailer.prev = header;
    }

    public Node<E> addAfter(E n, E addedValue) {
        Node<E> newNode = new Node<>(addedValue);
        Node<E> current = header.next;

        while (current != trailer && !current.data.equals(n)) {
            current = current.next;
        }

        if (current == trailer) {
            return header; // Value 'n' not found in the list, no
changes made.
        }

        newNode.prev = current;
        newNode.next = current.next;
        current.next.prev = newNode;
        current.next = newNode;
        return header;
    }
    public E deleteAfter(E n) {
        Node<E> current = header.next;

        while (current != trailer && !current.data.equals(n)) {
            current = current.next;
        }

        if (current == trailer || current.next == trailer) {
            return null; // Value 'n' not found in the list or no
node after 'n'.
        }

        Node<E> deletedNode = current.next;
        current.next = current.next.next;
        current.next.prev = current;
        return deletedNode.data;
    }
}
```

**c.** For the **addAfter(E n, E addedValue)** method, the worst-case time complexity is O(n), where 'n' is the number of nodes in the doubly linked list. This is because in the worst case, we may have to traverse the entire list to find the node with the value 'n'.

**f.** For the **deleteAfter(E n)** method, the worst-case time complexity is also O(n), where 'n' is the number of nodes in the doubly linked list. This is because in the worst case, we may have to traverse the entire list to find the node with the value 'n'.

**d.g.**
```
public static void main(String[] args) {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addAfter(null, "A");
    list.addAfter("A", "B");
    list.addAfter("B", "C");
    list.addAfter("C", "D");

    System.out.println("Original List:");
    printList(list);

    System.out.println("\nAdding 'M' after 'C':");
    list.addAfter("C", "M");
    printList(list);

    System.out.println("\nDeleting after 'C':");
    String deletedValue = list.deleteAfter("C");
    System.out.println("Deleted: " + deletedValue);
    printList(list);
}

public static void printList(DoublyLinkedList<String> list) {
    Node<String> current = list.header.next;
    while (current != list.trailer) {
        System.out.print(current.data + "<=>");
        current = current.next;
    }
    System.out.println("trailer");
}

//This code initializes a doubly linked list, adds 'M' after 'C',
and then deletes the node after 'C'. It prints the list after each
operation to verify the correctness of the functions.
```

**3. a.**

```
public E[] addAfter(E[] myArray, E n, E addedValue) {
    int index = -1;

    // Find the index of the value 'n' in the array.
    for (int i = 0; i < myArray.length; i++) {
        if (myArray[i].equals(n)) {
            index = i;
            break;
        }
    }

    if (index == -1) {
        return myArray; // Value 'n' not found in the array, no
changes made.
    }

    // Create a new array with increased size to accommodate the
added value.
    E[] newArray = Arrays.copyOf(myArray, myArray.length + 1);

    // Shift elements after the 'n' index to make space for the
added value.
    for (int i = newArray.length - 1; i > index + 1; i--) {
        newArray[i] = myArray[i - 1];
    }

    // Insert the addedValue after 'n'.
    newArray[index + 1] = addedValue;

    return newArray;
}
```

**b.** The Big-O complexity of this code is O(n), where 'n' is the length of the input array **myArray**. This is because in the worst case, we may need to traverse the entire array to find the value 'n', and the array copying and shifting operations also take O(n) time in total.

**c.**
```java
public static void main(String[] args) {
    Integer[] myArray = {1, 2, 3, 4, 5, 6};

    System.out.println("Original Array: " +
Arrays.toString(myArray));

    Integer[] modifiedArray = addAfter(myArray, 3, 8);

    System.out.println("Modified Array: " +
Arrays.toString(modifiedArray));
}
```

**d.**
```java
public E[] deleteAfter(E[] myArray, E n) {
    int index = -1;

    // Find the index of the value 'n' in the array.
    for (int i = 0; i < myArray.length; i++) {
        if (myArray[i].equals(n)) {
            index = i;
            break;
        }
    }

    if (index == -1 || index == myArray.length - 1) {
        return myArray; // Value 'n' not found in the array or no
element after 'n'.
    }

    // Create a new array with the same size to accommodate the
modified array.
    E[] newArray = Arrays.copyOf(myArray, myArray.length - 1);

    // Copy elements up to the 'n' index.
    System.arraycopy(myArray, 0, newArray, 0, index + 1);

    // Copy elements after the 'n' index.
    System.arraycopy(myArray, index + 2, newArray, index + 1,
myArray.length - index - 2);

    return newArray;
}
```

**e.** The Big-O complexity of this code is O(n), where 'n' is the length of the input array **myArray**. Similar to the **addAfter** function, we may need to traverse the entire array to find the value 'n', and the array copying and shifting operations also take O(n) time in total.

**f.**
```
public static void main(String[] args) {
    Integer[] myArray = {1, 2, 3, 4, 5, 6};

    System.out.println("Original Array: " +
Arrays.toString(myArray));

    Integer[] modifiedArray = deleteAfter(myArray, 3);

    System.out.println("Modified Array: " +
Arrays.toString(modifiedArray));
}
```