DAY - (1-6)

Task 1: Linked List Middle Element Search
You are given a singly linked list. Write a function to find the middle element without using any extra space and only one traversal through the linked list.
Ans)
Code:-

```
package com.wipro.linear;
public class LinkedListMiddle {
  private Node head;
  private int length;
  class Node {
     int value;
     Node next;
     public Node(int value) {
        this.value = value;
     }
  }
  public void addNode(int value) {
     Node newNode = new Node(value);
     if (head == null) {
        head = newNode;
     } else {
        Node current = head;
        while (current.next != null) {
           current = current.next;
        }
        current.next = newNode;
     }
     length++;
  }
  public Node findMiddle() {
     if (head == null) {
```

```
            return null;
        }
        Node slowPtr = head;
        Node fastPtr = head;
        while (fastPtr != null && fastPtr.next != null) {
            fastPtr = fastPtr.next.next;
            slowPtr = slowPtr.next;
        }
        return slowPtr;
    }
public static void main(String[] args) {
        LinkedListMiddle linkedList = new LinkedListMiddle();
        linkedList.addNode(14);
        linkedList.addNode(26);
        linkedList.addNode(33);
        linkedList.addNode(45);
        linkedList.addNode(58);
        Node middle = linkedList.findMiddle();
        if (middle != null) {
            System.out.println("Middle element value: " +
middle.value);
        } else {
            System.out.println("The list is empty.");
        }
    }
}
```

OUTPUT:

```
Middle element value: 33
```

Task 2: Queue Sorting with Limited Space
You have a queue of integers that you need to sort. You can only use additional space equivalent to one stack. Describe the steps you would take to sort the elements in the queue.
Ans)

Code:

```
package com.wipro.non.liniear;
import java.util.Queue;
import java.util.Iterator;
import java.util.LinkedList;
public class QueueSorting {
  static int minIndex(Queue<Integer> q, int sortedIndex) {
      int min_index = -1;
      int min_val = Integer.MAX_VALUE;
      int n = q.size();
      for (int i = 0; i < n; i++) {
          int curr = q.poll();
          if (curr <= min_val && i <= sortedIndex) {
              min_index = i;
              min_val = curr;
          }
          q.add(curr);
      }
      return min_index;
  }
  static void insertMinToRear(Queue<Integer> q, int
min_index) {
      int min_val = q.poll();
      for (int i = 0; i < min_index; i++) {
          int curr = q.poll();
          q.add(curr);
      }
```

```
        q.add(min_val);
        for (int i = min_index + 1; i < q.size(); i++) {
            int curr = q.poll();
            q.add(curr);
        }
    }
    static Queue<Integer> sortQueue(Queue<Integer> q) {
        Queue<Integer> s = new LinkedList<Integer>();
        int n = q.size();
        for (int i = 0; i < n; i++) {
            int min_index = minIndex(q, i);
            insertMinToRear(q, min_index);
            s.add(q.poll());
        }
        return s;
    }
    public static void main(String[] args) {
        Queue<Integer> q = new LinkedList<Integer>();
        q.add(4);
        q.add(3);
        q.add(6);
        q.add(1);
        q.add(2);
        Queue<Integer> sortedQueue = sortQueue(q);
        System.out.println("Sorted Queue: ");
        Iterator<Integer> itr = sortedQueue.iterator();
        while (itr.hasNext()) {
            System.out.print(itr.next() + " ");
        }
    }
}
```

OUTPUT:-

Sorted Queue:
4 3 1 2 6

Task 3: Stack Sorting In-Place
You must write a function to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure such as an array. The stack supports the following operations: push, pop, peek, and isEmpty.

CODE:-

```
package com.wipro.linear;
public class Stack {
    private Node top;
    private int height;
    private int value;
    private Node newNode;
      class Node {
          int value;
          Node next;
          public Node(int value) {
              this.value = value;
          }
      }
      public Stack(int value) {
          Node newNode = new Node(value);
          top = newNode;
          height = 1;
      }
      //Print the height of stack.
      public void getHeight() {
          System.out.println("Height:"+height);
      }

      public void printList() {
```

```
            Node temp = top;
            getTop();
            getHeight();
            System.out.println("Items in Stack:");
            while (temp != null) {
                System.out.println("---> " + temp.value );
                temp = temp.next;
            }
        }
        // Print the element in the top .
        public void getTop() {
            System.out.println("Top:"+top.value);
        }
        // ADD element :
        public void push(int value) {
            Node newNode = new Node(value);
            if(height== 0) {
                top = newNode;
            }else {
                newNode.next=top;
                top=newNode;
            }
            height++;
        }
        public Node pop() {
            if(height== 0) {
                return null;
            }else {
                Node temp=top;
                top=top.next;
                temp.next= null;
                height--;
                return temp;
```

```
                    }
                }
                public boolean isEmpty() {
                    return height ==0;
                }
                public int peek() {
                    if (top== null) {
                        System.out.println("Stack is empty");
                        return -1;
                    }
                    return top.value;
                }
                public static void main(String[] args) {
                    Stack mystack = new Stack(11);
                    mystack.getHeight();
                    mystack.getTop();
                    mystack.push(3);
                    mystack.push(11);
                    mystack.push(45);
                    mystack.push(49);
                    mystack.printList();
                    System.out.println("Top Node Deleted:" +
mystack.pop().value);
                    mystack.printList();
                    System.out.println("\n The peek element is:" +
mystack.peek());
                    mystack.printList();
                    System.out.println("\n The Stack is
empty?"+mystack.isEmpty());
                    mystack.printList();
                }
}
```

```
____
____
```

OUTPUT:

```
Height:1
Top:11
Top:49
Height:5
Items in Stack:
---> 49
---> 45
---> 11
---> 3
---> 11
Top Node Deleted:49
Top:45
Height:4
Items in Stack:
---> 45
---> 11
---> 3
---> 11
The peek element is:45
Top:45
Height:4
Items in Stack:
---> 45
---> 11
---> 3
---> 11
The Stack is empty?false
Top:45
Height:4
```

```
Items in Stack:
---> 45
---> 11
---> 3
---> 11
```

Task 4: Removing Duplicates from a Sorted Linked List
A sorted linked list has been constructed with repeated elements. Describe an algorithm to remove all duplicates from the linked list efficiently.
Ans)

CODE:-

```
package com.wipro.non.liniear;
public class LinkedListDup {
  Node head;
  class Node {
      int data;
      Node next;
      Node(int d) {
          data = d;
          next = null;
      }
  }
  void removeDuplicates() {
      Node current = head;
      Node previous = head;
      while (current!= null) {
          if (current.data == previous.data) {
              previous.next = current.next;
          } else {
              previous = current;
          }
          current = current.next;
      }
```

```
    }
    void printList() {
        Node temp = head;
        while (temp!= null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        LinkedListDup list = new LinkedListDup();
        list.head = list.new Node(1);
        list.head.next = list.new Node(2);
        list.head.next.next = list.new Node(2);
        list.head.next.next.next = list.new Node(3);
        list.head.next.next.next.next = list.new Node(3);
        list.head.next.next.next.next.next = list.new Node(3);
        list.head.next.next.next.next.next.next = list.new
Node(4);
        list.head.next.next.next.next.next.next.next = list.new
Node(5);
        list.head.next.next.next.next.next.next.next.next =
list.new Node(5);
        System.out.println("Original Linked List:");
        list.printList();
        list.removeDuplicates();
        System.out.println("Linked List after removing
duplicates:");
        list.printList();
    }
}
```

OUTPUT:-

```
Original Linked List:
1 2 2 3 3 3 4 5 5
Linked List after removing duplicates:
1 2 3 4 5
```

Task 5: Searching for a Sequence in a Stack
Given a stack and a smaller array representing a sequence, write a function that determines if the sequence is present in the stack. Consider the sequence present if, upon popping the elements, all elements of the array appear consecutively in the stack.
ANS)

CODE:-

```
package com.wipro.linear;
public class Stack {
    private Node top;
    private int height;
    private int value;
    private Node newNode;
      class Node {
          int value;
          Node next;
          public Node(int value) {
              this.value = value;
          }
      }
      public Stack(int value) {
          Node newNode = new Node(value);
          top = newNode;
          height = 1;
      }
      //Print the height of stack.
      public void getHeight() {
          System.out.println("Height:"+height);
      }
```

```
    public void printList() {
        Node temp = top;
        getTop();
        getHeight();
        System.out.println("Items in Stack:");
        while (temp != null) {
            System.out.println("---> " + temp.value );
            temp = temp.next;
        }
    }

    public void getTop() {
        System.out.println("Top:"+top.value);
    }

    public void push(int value) {
        Node newNode = new Node(value);
        if(height== 0) {
            top = newNode;
        }else {
            newNode.next=top;
            top=newNode;
        }
        height++;
    }
    public Node pop() {
        if(height== 0) {
            return null;
        }else {
            Node temp=top;
            top=top.next;
            temp.next= null;
```

```
                height--;
                return temp;

            }
        }
        public boolean isEmpty() {
            return height ==0;
        }
        public int peek() {
            if (top== null) {
                System.out.println("Stack is empty");
                return -1;
            }
            return top.value;
        }
        public static void main(String[] args) {
            Stack mystack = new Stack(11);
            mystack.getHeight();
            mystack.getTop();
            mystack.push(3);
            mystack.push(11);
            mystack.push(45);
            mystack.push(49);
            mystack.printList();
            System.out.println("Top Node Deleted:" +
mystack.pop().value);
            mystack.printList();
            //mystack.printList();
            //System.out.println("\n The peek element is:" +
mystack.peek());
            //mystack.printList();
            System.out.println("\n The Stack is
empty?"+mystack.isEmpty());
```

```
            mystack.printList();
        }
}
```

OUTPUT:-

```
Height:1
Top:11
Top:49
Height:5
Items in Stack:
---> 49
---> 45
---> 11
---> 3
---> 11
Top Node Deleted:49
Top:45
Height:4
Items in Stack:
---> 45
---> 11
---> 3
---> 11
The Stack is empty?false
Top:45
Height:4
Items in Stack:
---> 45
---> 11
---> 3
---> 11
```

Task 6: Merging Two Sorted Linked Lists

You are provided with the heads of two sorted linked lists. The lists are sorted in ascending order. Create a merged linked list in ascending order from the two input lists without using any extra space (i.e., do not create any new nodes).
Ans)

```
package com.wipro.linear;
public class MergeSortedLinkedLists {

  static class Node {
     int value;
     Node next;
     public Node(int value) {
        this.value = value;
     }
  }
  public static Node mergeTwoLists(Node l1, Node l2) {
     if (l1 == null) return l2;
     if (l2 == null) return l1;

     Node dummy = new Node(0);
     Node tail = dummy;

     while (l1 != null && l2 != null) {
        if (l1.value <= l2.value) {
           tail.next = l1;
           l1 = l1.next;
        } else {
           tail.next = l2;
           l2 = l2.next;
        }
        tail = tail.next;
     }

     if (l1 != null) {
        tail.next = l1;
```

```
        } else {
            tail.next = l2;
        }
        return dummy.next;
    }

    public static void printList(Node head) {
        Node current = head;
        while (current != null) {
            System.out.print(current.value + " ");
            current = current.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {

        Node l1 = new Node(1);
        l1.next = new Node(3);
        l1.next.next = new Node(5);

        Node l2 = new Node(2);
        l2.next = new Node(4);
        l2.next.next = new Node(6);
        System.out.println("List 1: ");
        printList(l1);
        System.out.println("List 2: ");
        printList(l2);

        Node mergedList = mergeTwoLists(l1, l2);
        System.out.println("Merged List: ");
        printList(mergedList);
    }
}
```

OUTPUT:-

```
List 1:
1 3 5
List 2:
2 4 6
Merged List:
1 2 3 4 5 6
```

Task 7: Circular Queue Binary Search
Consider a circular queue (implemented using a fixed-size array) where the elements are sorted but have been rotated at an unknown index. Describe an approach to perform a binary search for a given element within this circular queue.
Ans)

```
package com.wipro.linear;
public class CircularQueueBinarySearch {
  public static int search(int[] arr, int target) {
      int low = 0;
      int high = arr.length - 1;
      while (low <= high) {
          int mid = low + (high - low) / 2;
          if (arr[mid] == target) {
              return mid;
          }
                  if (arr[low] <= arr[mid]) {
                          if (target >= arr[low] && target < arr[mid]) {
                  high = mid - 1;
              } else {                    low = mid + 1;
              }
          } else {
              if (target > arr[mid] && target <= arr[high]) {
                  low = mid + 1;
              } else {
                  high = mid - 1;
```

```
            }
        }
    }
        return -1;
  }
  public static void main(String[] args) {
      int[] circularQueue = {6, 7, 8, 1, 2, 3, 4, 5};
      int target = 4;
      int result = search(circularQueue, target);
      if (result != -1) {
          System.out.println("Element found at index: " +
result);
      } else {
          System.out.println("Element not found in the
array.");
      }
  }
}
```

OUTPUT:-

Element found at index: 6

To perform a binary search on a circular queue that is implemented using a fixed-size array and sorted but rotated at an unknown index, we need to modify the standard binary search algorithm to account for the rotation. Here is a step-by-step approach in Java:

1. Identify the middle element of the array.
2. Determine which half is sorted (since the array is sorted but rotated, at least one half will always be sorted).
3. Adjust the search range based on whether the target lies in the sorted half or the unsorted half.

Steps:

Binary Search Loop:

Calculate the middle index mid.

Check if the mid element is the target.

Determine if the left half is sorted:

If the element at low is less than or equal to the element at mid, then the left half is sorted.

Determine if the right half is sorted:

If the element at mid is less than or equal to the element at high, then the right half is sorted.

Adjust low and high based on the sorted half and the target value.

Explanation:

1. Initial Setup:

- low starts at the beginning of the array.
- high starts at the end of the array.

2. Binary Search Loop:

- Compute mid using low + (high - low) / 2 to avoid potential overflow.
- Check if the mid element is the target. If yes, return mid.
- Determine if the left half [low, mid] is sorted by checking if arr[low] <= arr[mid].
- If sorted, check if the target lies within this range. If yes, adjust high to mid-1.
- If not, adjust low to mid + 1.
- If the left half is not sorted, the right half [mid, high] must be sorted.
- Check if the target lies within this range. If yes, adjust low to mid + 1.
- If not, adjust high to mid-1.

3. Return Result:

- If the target is found, return its index.
- If the loop exits, return -1 indicating the target is not found.

This approach efficiently searches for the target in a rotated sorted array by leveraging binary search principles.