Name :- Abhay Ashok Patil
E-mail :- abhya5050@gmail.com

DAY 1 – 6

Task 2: 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.

```
package com.wipro.linear;
public class LinkedList {
private Node head;
private Node tail;
private int length;
class Node {
int value;
Node next;
public Node(int value) {
super();
this.value = value;
}
}
public LinkedList(int value) {
super();
Node newNode = new Node(value);
// System.out.println("Node:"+ newNode);
head = newNode;
tail = newNode;
length = 1;
}
public void getHead() {
System.out.println("Head:" + head.value);
}
public void getTail() {
System.out.println("Tail: " + tail.value);
}
public void getLength() {
System.out.println("Length :" + length);
}
public void printList() {
Node temp = head;
System.out.println("\n");
getHead();
getTail();
getLength();
System.out.println("Items in list:");
while (temp != null) {
System.out.print("-->" + temp.value + "\t");
temp = temp.next;
}
}
public void append(int value) {
Node newNode = new Node(value);
if (length == 0) {
head = newNode;
```

```
tail = newNode;
} else {
tail.next = newNode;
tail = newNode;
}
length++;
}
public Node removeLast() {
if (length == 0) {
return null;
}
Node temp = head;
Node pre = head;
while (temp.next != null) {
pre = temp;
temp = temp.next;
}
tail = pre;
tail.next = null;
length--;
if (length == 0) {
head = null;
tail = null;
}
return temp;
}
public void prepend(int value) {
Node newNode = new Node(value);
if (length == 0) {
head = newNode;
tail = newNode;
} else {
newNode.next = head;
head = newNode;
}
length++;
}
public Node removeFirst() {
if (length == 0) {
return null;
}
Node temp = head;
head = head.next;
temp.next = null;
length--;
if (length == 0) {
tail = null;
}
return temp;
}
public Node get(int index) {
if (index < 0 || index >= length) {
return null;
}
Node temp=head;
```

```
for(int i=0;i<index;i++)
{
temp=temp.next;
}
return temp;
}
 public boolean set (int index, int value) {
 Node temp=get(index);
 if(temp != null) {
 temp.value=value;
 return true;
 }
 return false;
 }

 public boolean insert(int index,int value) {
 if(index <0 || index >=length) {
 return false;
 }
 if(index==0) {
 prepend(value);;
 return true;
 }
 if(index==length) {
 append(value);
 return true;
 }
 Node newNode=new Node(value);
 Node temp = get(index -1);
 newNode.next = temp.next;
 temp.next=newNode;
 length++;
 return true;
 }
 public Node findMiddleElement() {
 if(head==null)
 {
 return null ;
 }
 int count=0;
 Node temp=head;
 while(temp!=null)
 {
 count++;
 temp=temp.next;
 }
 temp=head;
 for(int i=0;i<count/2;i++)
 {
 temp=temp.next;
 }
 return temp;
 }
```

```
public static void main(String[] args) {
LinkedList myll = new LinkedList(11);
myll.printList();
myll.append(3);
myll.append(23);
myll.append(7);
myll.printList();
System.out.println("Removed :" + myll.removeLast().value);
myll.printList();
myll.prepend(1);
myll.printList();
System.out.println("Removed :" + myll.removeFirst().value);;
myll.printList();
System.out.println("Item at index 1 is "+ myll.get(1).value);
System.out.println("Replace item at index 1 :" +myll.set(1, 33));
 myll.printList();

 System.out.println("\n Insert between 1 and 2 :"+ myll.insert(2, 20));
 myll.printList();

 System.out.println("Middle element is: " + myll.findMiddleElement().value);

}
}
```

Task 3: 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.

Divide the Queue into Sub-Queues: Divide the queue into smaller sub-queues until each sub-queue contains only one element. You can achieve this by dequeuing elements from the original queue and enqueueing them into the stack.

Merge the Sub-Queues: Merge adjacent sub-queues into larger sorted sub-queues. To merge two sorted sub-queues, you'll use the stack to assist in sorting.

Repeat Merging: Repeat the merging process until all the sub-queues are merged into one sorted queue.

Input Queue: [4, 2, 5, 1, 3]

Step 1:

Queue => Stack: [4]
Queue => Stack: [2, 4]
Queue => Stack: [5, 2, 4]
Queue => Stack: [1, 5, 2, 4]
Queue => Stack: [3, 1, 5, 2, 4]

Step 2:

Stack => Queue: [1, 3, 5, 2, 4] (temporarily)
Queue => Stack: [1, 3, 5]
Stack => Queue: [1, 2, 4]
Stack => Queue: [1, 2, 3, 4, 5]

Resulting Sorted Queue: [1, 2, 3, 4, 5]

Initially, each element in the queue is considered as a separate sorted sub-queue.
Then, adjacent sub-queues are merged into larger sorted sub-queues until all elements are merged into one sorted queue.

Task 4: 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.

```
package com.wipro.linear;
import com.wipro.linear.LinkedList.Node;
public class Stack {
 private Node top;
 private int height;

 class Node{
 int value;
 Node next;

 public Node(int value) {
 super();
 this.value=value;
 }
 }
 public Stack(int value) {
 Node newNode = new Node(value);
 top = newNode;
 height=1;
 }
 public void getHeight() {
 System.out.println("Height :" + height);
 }
 public void getTop() {
 System.out.println("Top : "+ top.value);
 }

 public void printStack() {
Node temp = top;
System.out.println("\n");
getTop();
getHeight();
System.out.println("Items in Stack:");
while (temp != null) {
System.out.print("\n" + temp.value );
temp = temp.next;
}
}
 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;
}
Node temp = top;
top=top.next;
temp.next=null;
height--;
return temp;
}
 public int peek() {
 if(top==null) {
 System.out.println("Stack is empty");
 return -1;
 }
 return top.value;
 }

 public boolean isEmpty() {
 return height == 0;
 }
 public static void main(String[] args) {
 Stack mystack = new Stack(11);
 //mystack.getHeight();
 //mystack.getTop();
 mystack.printStack();

 mystack.push(3);
 mystack.push(23);
 mystack.push(7);
 mystack.printStack();

 System.out.println("\nTop Node pop out : " +mystack.pop().value);
 mystack.printStack();

 System.out.println("\nPeek element : "+ mystack.peek());
 System.out.println("\n stack is empty ? : "+ mystack.isEmpty());

 }
}
```

Task 5: 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.

To efficiently remove duplicates from a sorted linked list, you can use a simple algorithm that iterates through the list, comparing adjacent elements. If two adjacent elements are equal, you remove one of them. Since the list is sorted, duplicate elements will always be adjacent to each other.

the algorithm:

1. Initialize a pointer ‘current‘ to the head of the linked list.
2. Iterate through the linked list while ‘current‘ and ‘current.next‘ are not null.
3. Compare ‘current.data‘ with ‘current.next.data‘. If they are equal, remove ‘current.next‘ from the list by updating the next pointer of ‘current‘.
4. If ‘current.data‘ is not equal to ‘current.next.data‘, move ‘current‘ to ‘current.next‘.

5. Repeat steps 3-4 until ‘current.next‘ becomes null.
This algorithm has a time complexity of O(n), where n is the number of elements in the linked list, since we iterate through the list only once. Here’s a more detailed description:
plaintext
Input: 1 -> 1 -> 2 -> 3 -> 3 -> null
Step 1: current = head (1 -> 1 -> 2 -> 3 -> 3 -> null)
Step 2: current.data == current.next.data (1 == 1), so remove current.next (1 -> 2 -> 3 -> 3 -> null)
Step 3: current.data != current.next.data (1 != 2), move current to current.next (1 -> 2 -> 3 -> 3 -> null)
Step 4: current.data == current.next.data (2 == 3), so remove current.next (1 -> 2 -> 3 -> null)
Step 5: current.data != current.next.data (2 != 3), move current to current.next (1 -> 2 -> 3 -> null)
Step 6: current.next is null, stop.
After performing the above steps, the duplicates are removed from the linked list, and it becomes 1 -> 2 -> 3 -> null.

Task 6: 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.

```
import java.util.Stack;
public class SequenceInStack {
 public static boolean isSequenceInStack(Stack<Integer> stack, int[] sequence) {
 Stack<Integer> sequenceStack = new Stack<>();
 for (int i = sequence.length - 1; i >= 0; i--) {
 sequenceStack.push(sequence[i]);
 }
 while (!stack.isEmpty()) {
 if (stack.peek().equals(sequenceStack.peek())) {
 for (int value : sequence) {
 if (stack.isEmpty() || !stack.pop().equals(value)) {
 return false;
 }
 }
 return true;
 } else {
 stack.pop();
 }
 }
 return false;
 }
 public static void main(String[] args) {
 Stack<Integer> stack = new Stack<>();
 stack.push(1);
 stack.push(2);
 stack.push(3);
 stack.push(4);
 stack.push(5);
 stack.push(6);
 stack.push(7);
 stack.push(8);
```

```
 stack.push(9);
 int[] sequence = {5, 6, 7};
 System.out.println(isSequenceInStack(stack, sequence)); // Output: true
 }
}
```

Task 7: 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).

```
class ListNode {
 int val;
 ListNode next;
 ListNode(int val) {
 this.val = val;
 }
}
public class MergeSortedLinkedLists {
 public ListNode merge(ListNode l1, ListNode l2) {
 ListNode dummy = new ListNode(-1);
 ListNode current = dummy;
 while (l1 != null && l2 != null) {
 if (l1.val <= l2.val) {
 current.next = l1; l1 = l1.next; } else {
 current.next = l2; l2 = l2.next;
 }
 current = current.next }

 current.next = (l1 != null) ? l1 : l2;
 return dummy.next }
 public void printList(ListNode head) {
 ListNode current = head;
 while (current != null) {
 System.out.print(current.val + " ");
 current = current.next;
 }
 System.out.println();
 }
 public static void main(String[] args) {
 MergeSortedLinkedLists solution = new MergeSortedLinkedLists();
 ListNode l1 = new ListNode(1);
 l1.next = new ListNode(3);
 l1.next.next = new ListNode(5);
 ListNode l2 = new ListNode(2);
 l2.next = new ListNode(4);
 l2.next.next = new ListNode(6);
 ListNode mergedList = solution.merge(l1, l2);
 solution.printList(mergedList); // Output: 1 2 3 4 5 6
 }
}
```

Task 8: 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.

1. Initialize two pointers, ‘left‘ and ‘right‘, to the start and end indices of the circular queue, respectively.
2. While ‘left‘ is less than or equal to ‘right‘, do the following:
 - Calculate the middle index as ‘(left + right) / 2‘.
 - Check if the middle element is equal to the target element. If it is, return its index.
 - If the middle element is greater than or equal to the element at the ‘left‘ index, then the left half of the circular queue is sorted.
 - If the target element lies within the range of elements from ‘left‘ to ‘mid‘, update ‘right = mid - 1‘.
 - Otherwise, update ‘left = mid + 1‘.
 - If the middle element is less than or equal to the element at the ‘right‘ index, then the right half of the circular queue is sorted.
 - If the target element lies within the range of elements from ‘mid‘ to ‘right‘, update ‘left = mid + 1‘.
 - Otherwise, update ‘right = mid - 1‘.

By following this approach, you can perform a binary search on a rotated sorted array efficiently.

1. Initialize left = 0 and right = n - 1 (where n is the length of the circular queue).
2. While left <= right:
 a. Calculate mid = (left + right) / 2.
 b. If queue[mid] == target, return mid.
 c. If queue[left] <= queue[mid], then the left half is sorted.
 i. If queue[left] <= target <= queue[mid], search in the left half: right = mid - 1.
 ii. Otherwise, search in the right half: left = mid + 1.
 d. If queue[mid] <= queue[right], then the right half is sorted.
 i. If queue[mid] <= target <= queue[right], search in the right half: left = mid + 1.
 ii. Otherwise, search in the left half: right = mid - 1.
3. If the target is not found, return -1.

This approach ensures that the search is performed efficiently, even when the circular queue is rotated.