Name- Amit Kumar Parhi

Email-amitkparhi07@gmail.com

Day 1 to 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 wipro.com.assignment05;

class Node {
int data;
Node next;
     public Node(int data)
{
this.data = data;
this.next = null;
    }
}
class LinkedList {
    Node head;

public void add(int data) {
Node newNode = new Node(data);
if (head == null) {
head = newNode;
        }
else {

Node current = head;
while (current.next != null) {
current = current.next;
            }
            current.next = newNode;
        }
    }
public Node findMiddle() {
if (head == null) {
return null;
        }

        Node slowPointer = head;
        Node fastPointer = head;


while (fastPointer != null && fastPointer.next != null) {
slowPointer = slowPointer.next;
```

```
fastPointer = fastPointer.next.next;
        }
 return slowPointer;
    }
public void printList() {
Node current = head;
while (current != null) {

System.out.print(current.data + " -> ");
current = current.next;
        }
        System.out.println("null");
    }
public static void main(String[] args) {
LinkedList list = new LinkedList();
list.add(1);
list.add(2);
list.add(3);
list.add(4);
list.add(5);


System.out.print("Linked List: ");
list.printList();


Node middle = list.findMiddle();

if (middle != null) {

            System.out.println("The middle element is: " + middle.data);
        }
else {
            System.out.println("The list is empty.");
        }
    } }
```

Output:

```
Linked List: 1 -> 2 -> 3 -> 4 -> 5 -> null
The middle element is: 3
```

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.

```
package wipro.com.assignment05;
import java.util.LinkedList;
import java.util.Queue; import
java.util.Stack;

public class QueueSorter {
    public static void sortQueue(Queue<Integer> queue) {
        Stack<Integer> stack = new Stack<>();


while (!queue.isEmpty()) {
int x = queue.poll();

            // Place x in the correct position in the stack
while (!stack.isEmpty() && stack.peek() > x) {
queue.offer(stack.pop());
            }
stack.push(x);
        }

        // Transfer sorted elements back to the queue
while (!stack.isEmpty()) {
queue.offer(stack.pop());
        }
    }
public static void main(String[] args) {
Queue<Integer> queue = new LinkedList<>();
queue.offer(3);
queue.offer(1);
queue.offer(4);
queue.offer(1);
queue.offer(5);
queue.offer(9);
queue.offer(2);
queue.offer(6);
queue.offer(5);
queue.offer(3);
queue.offer(5);

        System.out.println("Original Queue: " + queue);
sortQueue(queue);
        System.out.println("Sorted Queue: " + queue);
    } }
```

Output:

```
Original Queue: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
Sorted Queue: [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
```

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 wipro.com.assignment05; import
java.util.Stack;

public class SortStack {

 public static void sortStack(Stack<Integer> stack) {
        Stack<Integer> tempStack = new Stack<>();


while        (!stack.isEmpty())        {
//    Pop    out    the    first    element
int current = stack.pop();

            // While temporary stack is not empty and top of tempStack is greater
than current

while (!tempStack.isEmpty() && tempStack.peek() > current) {
// Pop from tempStack and push it to the input stack
stack.push(tempStack.pop());
            }

            // Push current element to tempStack
tempStack.push(current);
        }

        // Transfer the sorted elements from tempStack back to the original stack
while (!tempStack.isEmpty()) {
stack.push(tempStack.pop());
        }
    }

    // Helper function to print the elements of the stack
public static void printStack(Stack<Integer> stack) {
for (Integer element : stack) {
System.out.print(element + " ");
        }
        System.out.println();
    }

    // Main method to test the sortStack function
public static void main(String[] args) {
Stack<Integer> stack = new Stack<>();
```

```
        // Push elements into the stack
stack.push(34);
stack.push(3);
stack.push(31);
stack.push(98);
stack.push(92);
stack.push(23);


System.out.println("Original stack:");
printStack(stack);

        // Sort the stack
sortStack(stack);

        System.out.println("Sorted stack:");
printStack(stack);
    } }
```

Output:

```
Original stack:
34 3 31 98 92 23
Sorted stack: 98
92 34 31 23 3
```

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.

```
package wipro.com.assignment05;
class
{
int data;
Node next;

    Node(int data) {
this.data = data;
this.next = null;
    }
}
public class RemoveDuplicates {

    // Function to remove duplicates from a sorted linked list
public static void removeDuplicates(Node head) {
 if (head == null) return;

        Node current = head;
```

```
while (current != null && current.next != null) {
if (current.data == current.next.data) {
                current.next = current.next.next; // Skip the duplicate node
            }
else {
                current = current.next; // Move to the next distinct element
            }
        }
    }

    // Helper function to print the linked list
public static void printList(Node head) {
Node temp = head;
 while (temp != null) {
            System.out.print(temp.data + " ");
temp = temp.next;
        }
        System.out.println();
    }

    // Main method to test the removeDuplicates function

public static void main(String[] args) {
        // Creating a sorted linked list with duplicates
Node head = new Node(1);
head.next = new Node(1);
head.next.next = new Node(2);
head.next.next.next = new Node(3);
head.next.next.next.next = new Node(3);
head.next.next.next.next.next = new Node(4);
head.next.next.next.next.next.next = new Node(4);
head.next.next.next.next.next.next.next = new Node(5);
        System.out.println("Original list:");
printList(head);

        // Remove duplicates from the linked list
removeDuplicates(head);


System.out.println("List after removing duplicates:");
printList(head);
    }
}
```

Output:

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

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.

```
package wipro.com.assignment05; import
java.util.Stack;
public class StackSequenceChecker {

    // Function to check if the sequence is present in the stack
public static boolean isSequencePresent(Stack<Integer> stack, int[] sequence) {
if (sequence.length == 0) return true; // An empty sequence is always considered
present


Stack<Integer> tempStack = new Stack<>();
int seqIndex = 0;

        // Iterate through the stack to find the sequence
while (!stack.isEmpty()) {
 int current = stack.pop();


 if (current == sequence[seqIndex]) {
seqIndex++;

if (seqIndex == sequence.length) {
                    // All elements of the sequence have been found consecutively
return true;
                }
            }
else {
             //Push element to temporary stack to preserve the stack's original order
tempStack.push(current);
            }
        }

        // Restore the original stack from the temporary stack
while (!tempStack.isEmpty()) {
stack.push(tempStack.pop());
        }
        return false; // Sequence was not found consecutively in the stack
}

    // Helper function to print the stack
```

```
    public static void printStack(Stack<Integer> stack) {
for (Integer element : stack) {
System.out.print(element + " ");
        }
        System.out.println();
    }

    // Main method to test the isSequencePresent function
    public static void main(String[] args) {
Stack<Integer> stack = new Stack<>();

        // Push elements into the stack
stack.push(3);
stack.push(2);
stack.push(1);
stack.push(4);
stack.push(6);
stack.push(5);
stack.push(4);
stack.push(3);
stack.push(2);
stack.push(1);


System.out.println("Original stack:");
printStack(stack);

        int[] sequence = {4, 6, 5};
        boolean result = isSequencePresent(stack, sequence);


System.out.println("Is the sequence " + arrayToString(sequence) + " present in the
stack? " + result);


System.out.println("Stack after checking for sequence:");
printStack(stack);
    }

    // Helper function to convert array to string
public static String arrayToString(int[] array) {
        StringBuilder sb = new StringBuilder();
        sb.append("[");

for (int i = 0; i < array.length; i++) {
sb.append(array[i]);
if (i < array.length - 1) {
sb.append(", ");
            }
}
        sb.append("]");
return sb.toString();
```

```
    }
}
```

Output:

```
Original stack:
3 2 1 4 6 5 4 3 2 1
Is the sequence [4, 6, 5] present in the stack? false
Stack after checking for sequence:
3 2 1 4 5 3 2 1
```

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).

```
package wipro.com.assignment05;
class Node {
int data;
Node next;

    Node(int data) {
this.data = data;
this.next = null;
    }
}
public class MergeSortedLists {

    // Function to merge two sorted linked lists
public static Node mergeTwoLists(Node l1, Node l2) {
// Create a dummy node to simplify the merge process
        Node dummy = new Node(0);
        Node current = dummy;

        // Traverse both lists and append the smaller node to the current node
while (l1 != null && l2 != null) {
if (l1.data <= l2.data) {
current.next = l1;

l1 = l1.next;             }
else {

current.next = l2;
l2 = l2.next;
            }
            current = current.next;
        }
```

```
        // If one list is exhausted, append the remaining elements of the other list
if (l1 != null) {
current.next = l1;
        }
else {
            current.next = l2;
        }

        // Return the merged list, starting from the node after the dummy node
return dummy.next;
    }

    // Helper function to print the linked list
public static void printList(Node head) {
Node temp = head;
while (temp != null) {
            System.out.print(temp.data + " ");
temp = temp.next;
        }
        System.out.println();
    }

    // Main method to test the mergeTwoLists function
public static void main(String[] args) {         //
Creating first sorted linked list
Node l1 = new Node(1);
l1.next = new Node(3);
l1.next.next = new Node(5);

        // Creating second sorted linked list
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);

        // Merge the two lists
        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 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.

```
package wipro.com.assignment05; public
class CircularQueueBinarySearch {

    // Function to perform binary search on a rotated sorted array
public static int search(int[] arr, int target) {
int left = 0;

int right = arr.length - 1;


while (left <= right) {

int mid = left + (right - left) / 2;

            // Check if mid is the target
if (arr[mid] == target) {
return mid;
            }

            // Determine which part is sorted
if (arr[left] <= arr[mid]) {
// Left part is sorted

if (arr[left] <= target && target < arr[mid]) {
right = mid - 1; // Target is in the left part
                }
else {
                    left = mid + 1; // Target is in the right part
                }
            }
else {
                // Right part is sorted

if (arr[mid] < target && target <= arr[right]) {
left = mid + 1; // Target is in the right part
                }
```

```
else {
                    right = mid - 1; // Target is in the left part
                }
            }
        }

        // Target not found
return -1;
    }

    // Main method to test the search function
public static void main(String[] args) {
        // Example of a rotated sorted array (circular queue)
int[] circularQueue = {4, 5, 6, 7, 0, 1, 2, 3};
int target = 6;

        // Perform binary search
        int index = search(circularQueue, target);

        // Output the result
if (index != -1) {
            System.out.println("Element " + target + " found at index " + index);
} else {
            System.out.println("Element " + target + " not found in the array");
}
    }

}
```

Output:

```
Element 6 found at index 2
```