Question 1

In a Juice shop, the shopkeeper sells Mango, Orange and Pineapple milkshakes. Group of customers comes and orders milkshakes. Your Juice machine can deliver two different milkshakes in 1 second or 1 milkshake in a second. Find out the minimum time required to deliver these milkshakes.

Test case 1:
Total number of orders for Mango milkshake
5
Total number of orders for Orange milkshake
4
Total number of orders for Pineapple milkshake
4

Minimum time needed to deliver all orders is: 7

Test case 2:
Total number of orders for Mango milkshake
3
Total number of orders for Orange milkshake
0
Total number of orders for Pineapple milkshake
0

Minimum time needed to deliver all orders is: 3

Test case 3:
Total number of orders for Mango milkshake
1
Total number of orders for Orange milkshake
4
Total number of orders for Pineapple milkshake
2

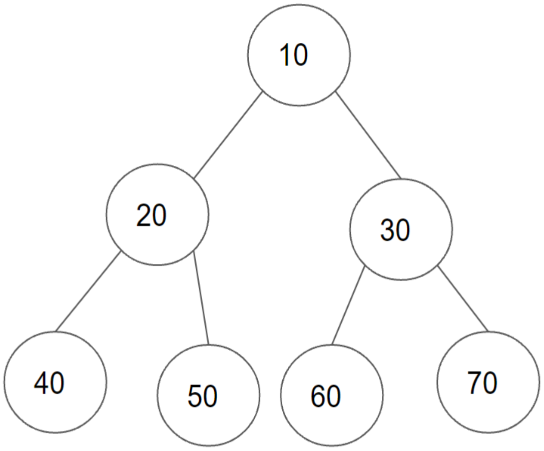
Minimum time needed to deliver all orders is: 4

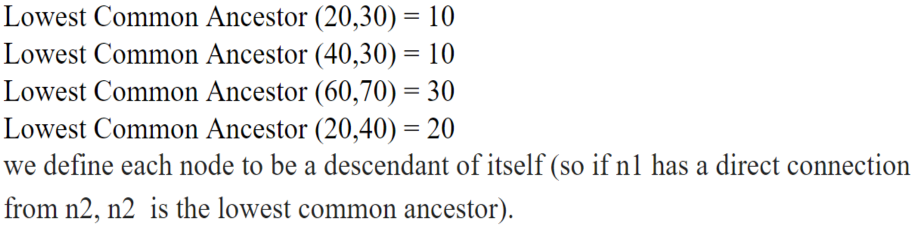
Solution

```
import java.util.Iterator;
import java.util.PriorityQueue;
import java.util.Scanner;

public class MinimumTime {
    // 1. Declare and initialize all variables and objects
    int seconds;
    Scanner sc = new Scanner(System.in);

    PriorityQueue<Integer> queue = new
PriorityQueue<>(java.util.Collections.reverseOrder());

    // 2. Get data from user and add to queue
    public void getData() {
        System.out.println("Total number of orders for Mango milkshake");
        queue.add(sc.nextInt());

        System.out.println("Total number of orders for Orange milkshake");
        queue.add(sc.nextInt());

        System.out.println("Total number of orders for Pineapple milkshake");
        queue.add(sc.nextInt());
    }

    // 3. Find the minimum time required to fill the cups
    public void findMinimumTime() {
        Iterator<Integer> list = queue.iterator();

        while(!queue.isEmpty()) {
            int val1 = 0, val2 = 0;

            //get top 2 priority values
            if(list.hasNext()) {
                val1 = queue.remove();
            }
            if(list.hasNext()) {
                val2 = queue.remove();
            }

            // check if there is 0 in any of the variables and take necessary steps
            if(val1 == 0 && val2 >0) {
                seconds += val2;
                break;
            }
            if(val1 > 0 && val2 == 0) {
                seconds += val1;
                break;
            }
            // if both the variables have value >0 then add a second and decrease
both the variable value by 1
            if(val1>0 && val2>0) {
                val1--;
                val2--;
                seconds++;
            }

            // If any of the variable has value >0 add then back into the queue
            if (val1>0) {
                queue.add(val1);
            }
            if(val2>0) {
                queue.add(val2);
            }
        }
    }

    public static void main(String[] args) {

        // 1. create object
        MinimumTime minimumTime = new MinimumTime();

        // 2. Implement the methods
        minimumTime.getData();
        minimumTime.findMinimumTime();
        System.out.println("\nMinimum time needed to deliver all orders is:
"+minimumTime.seconds);
    }
}
```

Question 2

Find the Lowest Common Ancestor in a Binary Tree for two nodes n1, n2.





Lowest Common Ancestor (20,30) = 10
Lowest Common Ancestor (40,30) = 10
Lowest Common Ancestor (60,70) = 30
Lowest Common Ancestor (20,40) = 20
we define each node to be a descendant of itself (so if n1 has a direct connection from n2, n2 is the lowest common ancestor).

Approach

Traverse tree twice and store path root to node1 and root to node 2. Traverse both paths till the values in arrays are the same. Now return the common element just before the mismatch.

Solution

```
package com.greatlearning.iiitr.mentoredSession3.lca;

import java.util.ArrayList;
import java.util.List;

// A Binary Tree node
            class Node {
                int data;
                Node left, right;
```

```
    Node(int value) {
        data = value;
        left = right = null;
    }
}

public class LeastCommonAncestor {


    Node root;
    private List<Integer> path1 = new ArrayList<>();
    private List<Integer> path2 = new ArrayList<>();

    // Finds the path from root node to given root of the tree.
    int findLCA(int node1, int node2) {
        path1.clear();
        path2.clear();
        return findLCAInternal(root, node1, node2);
    }

    private int findLCAInternal(Node root, int node1, int node2) {

        if (!findPath(root, node1, path1) || !findPath(root, node2, path2)) {
            System.out.println((path1.size() > 0) ? "node1 is present" : "node1 is missing");
            System.out.println((path2.size() > 0) ? "node2 is present" : "node2 is missing");
            return -1;
        }

        int i;
        for (i = 0; i < path1.size() && i < path2.size(); i++) {

        // System.out.println(path1.get(i) + " " + path2.get(i));
            if (!path1.get(i).equals(path2.get(i)))
                break;
        }

        return path1.get(i-1);
    }

    // Finds the path from root node to given root of the tree, Stores the
    // path in a vector path[], returns true if path exists otherwise false
    private boolean findPath(Node root, int n, List<Integer> path)
    {
        // base case
        if (root == null) {
            return false;
        }

        // Store this node . The node will be removed if
        // not in path from root to n.
        path.add(root.data);

        if (root.data == n) {
            return true;
        }

        if (root.left != null && findPath(root.left, n, path)) {
            return true;
        }

        if (root.right != null && findPath(root.right, n, path)) {
            return true;
        }

        // If not present in subtree rooted with root, remove root from
        // path[] and return false
        path.remove(path.size()-1);

        return false;
    }

    // Driver code
    public static void main(String[] args)
    {
            LeastCommonAncestor tree = new LeastCommonAncestor();
        tree.root = new Node(10);
        tree.root.left = new Node(20);
        tree.root.right = new Node(30);
        tree.root.left.left = new Node(40);
        tree.root.left.right = new Node(50);
        tree.root.right.left = new Node(60);
        tree.root.right.right = new Node(70);

        System.out.println("Least Common Ancestor(20, 30): " + tree.findLCA(20,30));
        System.out.println("Least Common Ancestor(40, 30): " + tree.findLCA(40,30));
        System.out.println("Least Common Ancestor(60, 70): " + tree.findLCA(60,70));
        System.out.println("Least Common Ancestor(20, 40): " + tree.findLCA(20,40));

    }
}
```

Question 3

Write a program to reverse a linked list without storing its data in any other data structure.

Example

1->2->3->4->NULL
4->3->2->1->NULL

Approach REFER : https://www.geeksforgeeks.org/reverse-a-linked-list/

1. Initialize three pointers prev as NULL, curr as head and next as NULL.
2. Iterate through the linked list. In loop, do following.

   // Before changing next of current,

   // store next node

   next = curr->next

   // Now change next of current

   // This is where actual reversing happens

   curr->next = prev

   // Move prev and curr one step forward

   prev = curr

   curr = next

Solution

```
package com.greatlearning.iiitr.mentoredSession3.ll;

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d)
        {
```

```
        data = d;
        next = null;
    }
}

/* Function to reverse the linked list */
Node reverse(Node node)
{
    Node prev = null;
    Node current = node;
    Node next = null;
    while (current != null) {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    node = prev;
    return node;
}

// prints content of double linked list
void printList(Node node)
{
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver Code
public static void main(String[] args)
{
    LinkedList list = new LinkedList();
    list.head = new Node(10);
    list.head.next = new Node(20);
    list.head.next.next = new Node(30);
    list.head.next.next.next = new Node(40);

    System.out.println("Given Linked list");
    list.printList(head);
    head = list.reverse(head);
    System.out.println("");
    System.out.println("Reversed linked list ");
    list.printList(head);
}
```

```
}
```

Learning Objectives:

1. Should be able to use Collection Framework classes and its methods.
2. Write less, efficient and better code with the help of collection framework concepts.
3. Implement and Understand non linear data structures with practical implementation.