PRACTICAL EXAMINATION (2021-2022)

Data Structures and Algorithms

Name: S.Priyanka

Class: III CSE B

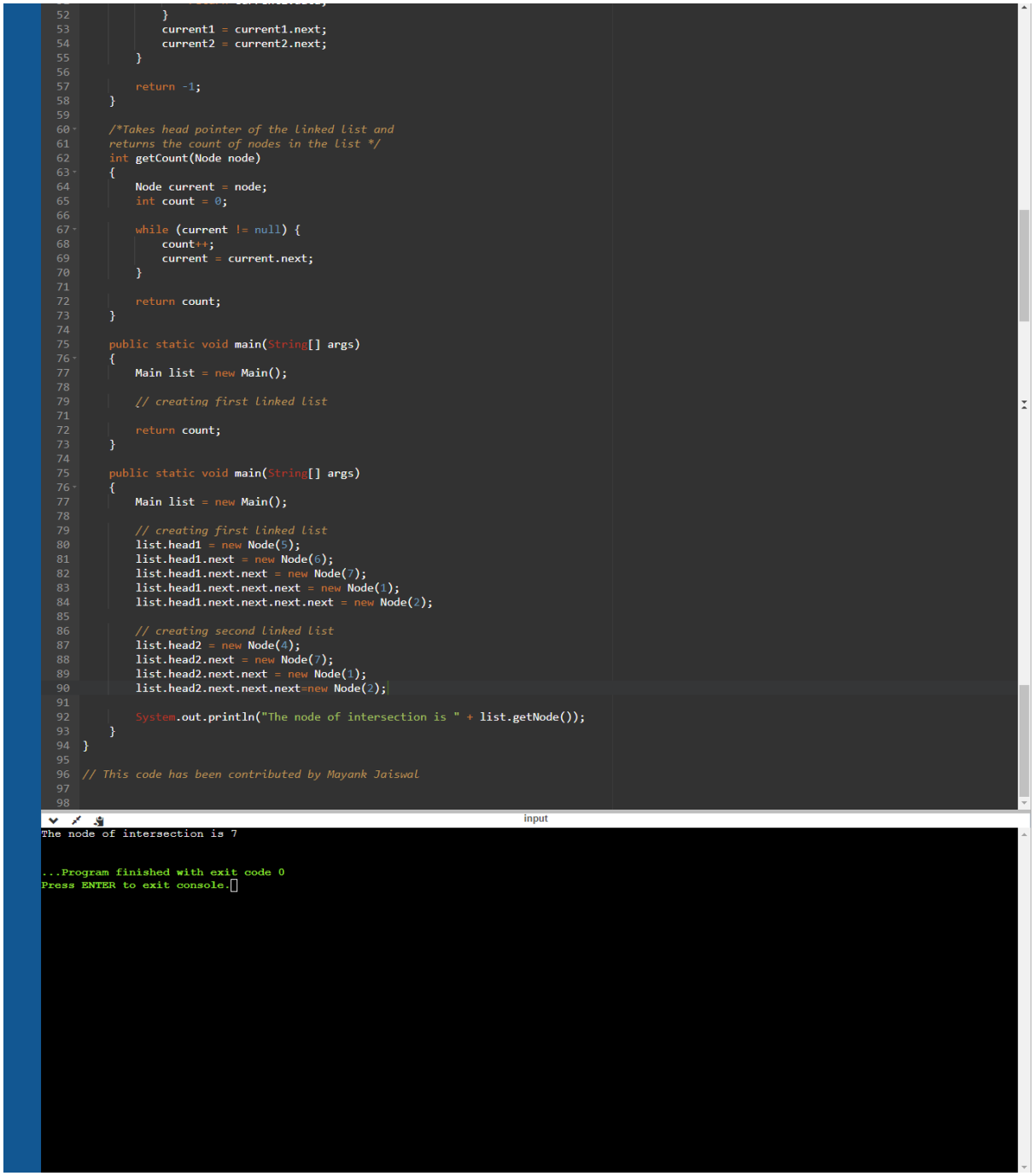
I.

1) Algorithm for finding the merging point of two linked lists

- Get count of the nodes in the first list, let count be c1.
- Get count of the nodes in the second list, let count be c2.
- Get the difference of counts **d = abs(c1 – c2)**
- Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes
- Then we can traverse both the lists in parallel till we come across a common node.

Implementation of the above algorithms

```
class Main {

    static Node head1, head2;

    static class Node {

        int data;
        Node next;

        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /*function to get the intersection point of two linked
    lists head1 and head2 */
    int getNode()
    {
        int c1 = getCount(head1);
        int c2 = getCount(head2);
        int d;

        if (c1 > c2) {
            d = c1 - c2;
            return _getIntesectionNode(d, head1, head2);
        }
        else {
            d = c1 - c2;
            return _getIntesectionNode(d, head1, head2);
        }
        else {
            d = c2 - c1;
            return _getIntesectionNode(d, head2, head1);
        }
    }

    /* function to get the intersection point of two linked
    lists head1 and head2 where head1 has d more nodes than
    head2 */
    int _getIntesectionNode(int d, Node node1, Node node2)
    {
        int i;
        Node current1 = node1;
        Node current2 = node2;
        for (i = 0; i < d; i++) {
            if (current1 == null) {
                return -1;
            }
            current1 = current1.next;
        }
        while (current1 != null && current2 != null) {
            if (current1.data == current2.data) {
                return current1.data;
            }
            current1 = current1.next;
            current2 = current2.next;
```



```
            }
            current1 = current1.next;
            current2 = current2.next;
        }

        return -1;
    }

    /*Takes head pointer of the linked list and
    returns the count of nodes in the list */
    int getCount(Node node)
    {
        Node current = node;
        int count = 0;

        while (current != null) {
            count++;
            current = current.next;
        }

        return count;
    }

    public static void main(String[] args)
    {
        Main list = new Main();

        // creating first linked list

        return count;
    }

    public static void main(String[] args)
    {
        Main list = new Main();

        // creating first linked list
        list.head1 = new Node(5);
        list.head1.next = new Node(6);
        list.head1.next.next = new Node(7);
        list.head1.next.next.next = new Node(1);
        list.head1.next.next.next.next = new Node(2);

        // creating second linked list
        list.head2 = new Node(4);
        list.head2.next = new Node(7);
        list.head2.next.next = new Node(1);
        list.head2.next.next.next=new Node(2);

        System.out.println("The node of intersection is " + list.getNode());
    }
}

// This code has been contributed by Mayank Jaiswal
```

input

```
The node of intersection is 7


...Program finished with exit code 0
Press ENTER to exit console.
```

2)Yes. This can be solved by using sorting technique. We can combine two techniques to get a solution to the problem. This can be done as:-

1. Create an array and store all the addresses of the nodes.
2. Now sort this array.
3. For each element in the second list, **from the beginning** search for the address in the array. We can use a very efficient search algorithm like Binary Search which gives us the result in O(log n).
4. If we find a same memory address, that means that is the merging point of the 2 lists.

Time Complexity: Time for sorting + Time for searching each element = O(Max(m*log(m), n*log(n)))
Space Complexity: O(Max(m,n))

3)yes . we can solve this using hashing table. The following steps are involved in it:-

1. Select a List, that has fewer number of elements. We can get the number of elements, by a single scan on both the lists. If both the lists have same number of elements, select any list at random.
2. Create a hash table using the list with fewer elements. Creating a hash table means storing the address of each of the nodes of the smaller list in a separate data structure such as an array.
3. Now, traverse the other list and compare the address of each of the node with the values in the hash table.
4. If there exists an intersection point, certainly we will find a match in the hash table and we will obtain the intersection point.

Time Complexity: Time for creating hash table + Time for scanning the list
Space Complexity: O(m) or O(n), depending upon the smaller size list.

4)Yes . we can use stack for solving this problem. The steps involved in solving the problem using stacks are:-

1. Create 2 different stacks for both the lists.
2. Push all the elements of both the lists in the 2 stacks.
3. Now start POPing the elements from both the stacks at once.
4. Till both the lists are merged, we will get the same value from both the stacks.
5. As soon as both the stacks return different value, we know that the last popped element was the merging point of the lists.
6. Return the last popped element from the stack.

5) Other way : BRUTE – FORCE APPROACH

This is the easiest method, and in this method we compare each node of List 1 with each node of List 2. If the address of both is same, we return the node.

The problem here is that the time complexity is very high. If list1 is of length 'm' and list2 of length 'n', then
Time Complexity:- O(m*n)
Space Complexity:- O(1)

6) FURTHER IMPROVING THE COMPLEXITY (THE BEST APPROACH)

This approach is the most efficient approach and utilizes a little brain power and a little maths trick, which can really give us a fast solution to the problem. The steps involved are:-

1. Find the length of both the lists. Let 'm' be the length of List 1 and 'n' be the length of List 2.
2. Find the difference in length of both the lists. $d = m - n$
3. Move ahead *'d'* steps in the longer list.
4. This means that we have reached a point after which, both of the lists have same number of nodes till the end.
5. Then we can traverse both the lists in parallel till we come across a common node.

II.

a) Algorithm for finding the size of a binary tree without recursion

Iterative method to Calculate Size of a tree.The idea is to use Level Order Traversing .

1) Create an empty queue q

2) temp_node = root /*start from root*/

3) Loop while temp_node is not NULL

a) Enqueue temp_node's children (first left then right children) to q

b) Increase count with every enqueuing.

c) Dequeue a node from q and assign it's value to temp_node

Implementation of the above algorithm

Main.java

```java
// Java programm to calculate
// Size of a tree without recursion
import java.util.LinkedList;
import java.util.Queue;

class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class Main
{
    Node root;

    public int size()
    {
        if (root == null)
            return 0;

        // Using level order Traversal.
        Queue<Node> q = new LinkedList<Node>();
```

Main.java

```java
        // Using level order Traversal.
        Queue<Node> q = new LinkedList<Node>();
        q.offer(root);

        int count = 1;
        while (!q.isEmpty())
        {
            Node tmp = q.poll();

            // when the queue is empty:
            // the poll() method returns null.
            if (tmp != null)
            {
                if (tmp.left != null)
                {
                    // Increment count
                    count++;

                    // Enqueue left child
                    q.offer(tmp.left);
                }
                if (tmp.right != null)
                {
                    // Increment count
                    count++;

                    // Enqueue left child
                    q.offer(tmp.right);
```

Main.java

```
                    // Enqueue left child
                    q.offer(tmp.right);
                }
            }
        }

        return count;
    }

    public static void main(String args[])
    {
        /* creating a binary tree and entering
        the nodes */
        Main tree = new Main();
        tree.root = new Node(6);
        tree.root.left = new Node(3);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(5);
        tree.root.right = new Node(9);
        tree.root.right.left = new Node(7);
        tree.root.right.right = new Node(11);

        System.out.println("The size of binary tree" +
                           " is : " + tree.size());
    }
}
```

input

```
The size of binary tree is : 7


...Program finished with exit code 0
Press ENTER to exit console.
```