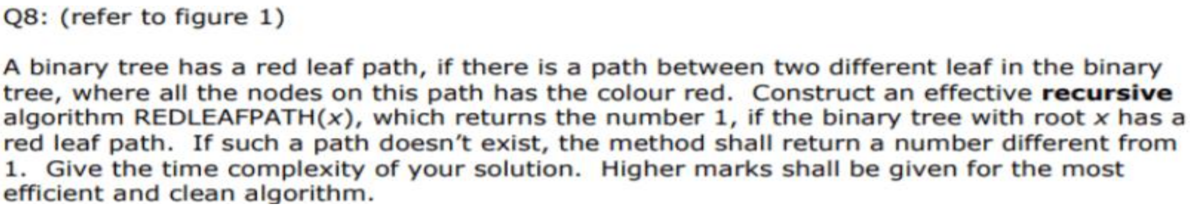

Q8: (refer to figure 1)

A binary tree has a red leaf path, if there is a path between two different leaf in the binary tree, where all the nodes on this path has the colour red. Construct an effective **recursive** algorithm REDLEAFPATH(*x*), which returns the number 1, if the binary tree with root *x* has a red leaf path. If such a path doesn't exist, the method shall return a number different from 1. Give the time complexity of your solution. Higher marks shall be given for the most efficient and clean algorithm.

- To find whether there is a red leaf path in a binary tree, we should first find all the leaves that are in our binary tree (the nodes without children).

Algorithm:

```
Let Leaves[] be an array of all the leaves we have.
Let's create a Graph from the first traversal of the binary tree. Graph[x] = [node1,nod2…]
Function Explore(x):
Start with root x:

    -   If x has no children:
        > Append it to Leaves.
        > Return.
    -   Else:
        > If x.left != null:
            ➢ Graph[x.left].add(x)
            ➢ Graph[x].add(x.left)
            ➢ Explore(x.left)

        > If x.right != null:
            >  Graph[x.right].add(x)
            >  Graph[x[.add(x.right)
            >  Explore(x.right)
```

At this point, we have all the leaves in Leaves and have the Graph created (all nodes with their neighbors cached)

Now, we start from Leaves and try to find if there exists a red leaf path connecting two leaves. For every node in Leaves do the following…

```
REDLEAFPATH(x):

    -   For every neighbor of x stored in Graph[x]:
            ➢ If neighbor is red:
                1) If not visited
                    a) If has no children then it's a leaf so return 1 and break.
                    b) If no leaf then mark visited and REDLEAFPATH(neighbor).
```

If the algorithm didn't return 1, then no red leaf path exists so we return 0.

Code in Java

```
int Leaves[];
Hashmap<Node,Array[]> Graph;
Hashmap<Node,Boolean> Visited;

void explore(x){
     if ( x.left == null && x.right == null )
          Leaves.Add(x);
     if ( x.left != null ){
          Graph[x].add(x.left);
          Graph[x.left].add(x);
          explore(x.left);
     }
     if ( x.right != null ){
          Graph[x].add(x.right);
          Graph[x.right].add(x);
          explore(x.right);
     }
}
// at this point we filled Graph and Leaves.

int REDLEAFPATH(x){
     for ( node neighbor : Graph[x] ){
          if ( ! Visited[neighbor)) ) {
               Visited[neighbor] = True;
               if ( neighbor.left == null && neighbor.right == null ) {
                    return 1;
               }
               else REDLEAFPATH(neighbor);
          }
     }
     return 0;
}

void solve(node x){
     explore(node root); // call helper function
     for ( node leaf : Leaves ){
          if ( REDLEAFPATH(leaf) == 1 ){
               System.out.println(“Red Leaf Path exists”);
          }
     }
}
///////// inside main
solve(x);
/////////
```

Helper function – explore(node x)

Main Function – REDLEAFPATH(node x)

Time complexity

The binary tree is traversed at most two times, and each node is traversed at most once at each tree traversal, so there is a total of O(n+n) time complexity, or **O(n)**.