**Write the answers to the following questions in Class Assignment 3 , Google Colab. Due Date: 26th September, 11:59 pm.**

**N:1**

Write a function named countLeftNodes that accepts a pointer to the root of a binary tree of integers. Your function should return the number of left children in the tree. A left child is a node that appears as the root of the left-hand subtree of another node. For example, if a pointer named root points to the root of the tree below, the call of countLeftNodes(root) should return 3 because the tree has 4 left children (the nodes storing the values 2, 15, and 14):

       11

       /  \

     2   20

           /  \

       15  28

      /

                              14

*Constraints:* You must implement your function recursively and without using loops. Do not construct any new BinaryTreeNode objects in solving this problem (though you may create as many BinaryTreeNode\* pointer variables as you like). Do not use any auxiliary data structures to solve this problem (no array, vector, stack, queue, string, etc). Your function should not modify the tree's state; the state of the tree should remain constant with respect to your function.

Assume that you are using the BinaryTreeNode structure as defined below:

struct BinaryTreeNode {

   int data;

   BinaryTreeNode\* left;

   BinaryTreeNode\* right;

};

**N:2**

Write a function named **range** that removes from a binary tree any elements that are not in a given range, inclusive. Your function accepts three parameters: a reference to a pointer to a BinaryTreeNode representing the root of a tree, and a minimum and maximum integer. You should remove from the tree any elements whose values are not in the range of [min .. max], inclusive. You should also return an integer count of the number of elements that were removed, if any.

For this function, assume that your tree is a binary search tree (BST) and that its elements are arranged in a valid BST order. Your function should maintain the BST ordering property of the tree. For example, suppose a variable named tree points to the root of a tree that stores the following elements:

              \_\_50\_\_\_\_\_\_\_\_\_\_\_\_

             /                 \

     \_\_\_\_\_\_38          \_\_\_\_\_\_\_\_\_90

    /        \        /

  14          42    54\_\_\_

 /  \                    \

8    20                   72

       \                 /  \

        26             61    83

The table below shows what the state of the tree would be if various calls were made. The calls shown are separate; it's not a chain of calls in a row. Also notice that each call returns the number of nodes that were removed.

|  |  |  |  |
| --- | --- | --- | --- |
| **range(tree,25,72)** | **range(tree,54,80)** | **range(tree,18,42)** | **range(tree,-3,6)** |
| \_\_\_50       /     \     38       54\_\_\_    /  \           \  26    42          72                   /                 61 | 54\_\_\_       \        72       /     61 | \_\_\_38    /     \  20       42    \     26 | (empty tree) |
| returns 7 | returns 3 | returns 4 | returns 0 |

Do not leak memory; if you remove a node from the tree, free the memory associated with that node. If the range is invalid (if the minimum exceeds the maximum), throw an integer exception and don't modify the tree.

*Constraints:* You must implement your function recursively and without using loops. Your solution should be at worst O(N) where N is the number of elements in the tree. Do not construct any new BinaryTreeNode objects in solving this problem (though you may create as many BinaryTreeNode\*pointer variables as you like). Do not use any auxiliary data structures to solve this problem (no array, vector, stack, queue, string, etc). You may define helper functions. You should not modify the tree passed in as the parameter. You also should not change the data of any nodes.

Assume that you are using the BinaryTreeNode structure as defined below:

**struct** **BinaryTreeNode** {

   int data;

   BinaryTreeNode\* left;

   BinaryTreeNode\* right;

};

**N:3**

Write a function named **hasPath** that interacts with a tree of BinaryTreeNode structures representing a binary search tree. The function accepts three parameters: a pointer to the root of the tree, and two integers *start* and *end*, and returns true if a path can be found in the tree from *start* down to *end*. In other words, both *start* and *end* must be element data values that are found in the tree, and *end* must be below *start*, in one of *start*'s subtrees; otherwise the function returns false. If *start* and *end* are the same, you are simply checking whether a single node exists in the tree with that data value. If the tree is empty, your function should return false.

For example, suppose a BinaryTreeNode pointer named tree points to the root of a tree storing the following elements. The table below shows the results of several various calls to your function:

           67

        /      \

    38            152

  /    \         /

16     42      99

      /

    40

|  |  |  |
| --- | --- | --- |
| **Call** | **Result** | **Reason** |
| hasPath(tree, 67, 99) | true | path exists 67 -> 152 -> 99 |
| hasPath(tree, 30, 40) | true | path exists 38 -> 42 -> 40 |
| hasPath(tree, 67, 67) | true | node exists with data of 67 |
| hasPath(tree, 16, 16) | true | node exists with data of 16 |
| hasPath(tree, 52, 99) | true | path exists 52 -> 99 |
| hasPath(tree, 99, 67) | false | nodes do exist, but in wrong order |
| hasPath(tree, 38, 99) | false | nodes do exist, but there is no path from 38 to 99 |
| hasPath(tree, 67, 100) | false | end of 100 doesn't exist in the tree |
| hasPath(tree, -1, 40) | false | start of -1 doesn't exist in the tree |
| hasPath(tree, 34, 64) | false | start/end of 34 and 64 both don't exist in the tree |

An empty tree does not contain any paths, so if the tree is empty, your function should return false.

*Constraints:* You must implement your function recursively and without using loops. Your function should not modify the tree's state; the state of the tree should remain constant with respect to your function. Do not construct any new BinaryTreeNode objects in solving this problem (though you may create as many BinaryTreeNode\* pointer variables as you like). Do not use any auxiliary data structures to solve this problem (no array, vector, stack, queue, string, etc).

Assume that you are using the BinaryTreeNode structure as defined below:

**struct** **BinaryTreeNode** {

   int data;

   BinaryTreeNode\* left;

   BinaryTreeNode\* right;

};

**N:26.**

Write a method **depthSum** that returns the sum of the values stored in a binary tree of integers weighted by the depth of each value. You should return the value at the overallRoot plus 2 times the values stored at the next level of the tree plus 3 times the values stored at the next level of the tree plus 4 times the values stored at the next level of the tree and so on. For example, in the tree below:

          +---+

          | 10 |

          +---+

         /     \

     +---+     +---+

     | 7 |     |16 |

     +---+     +---+

    /     \         \

+---+     +---+     +---+

| 3 |     | 9 |     |24 |

+---+     +---+     +---+

         /               \

     +---+               +---+

     | 8 |               |25 |

     +---+               +---+

The sum would be computed as:

1 \* 10 + 2 \* (7 + 16) + 3 \* (3 + 9 + 24) + 4 \* (8 + 25) = 296

Assume that you are using the BinaryTreeNode structure as defined below:

**struct** **BinaryTreeNode** {

   int data;

   BinaryTreeNode\* left;

   BinaryTreeNode\* right;

};

**N:27.**

Write a function named **sumCousin** that accepts a pointer to the root and a value of a binary tree of integers. Your function should return the sum of the cousins of the given value . The children of the same root is not considered as cousin.

        +---+

          |10 |

          +---+

         /     \

     +---+     +---+

     | 4 |     |16 |

     +---+     +---+

    /     \         \

+---+     +---+     +---+

| 3 |     | 9 |     |20 |

+---+     +---+     +---+

         /               \

     +---+               +---+

     | 7 |               |25 |

     +---+               +---+

    /      \           /       \

+---+     +---+     +---+      +--+

| 5 |     | 8 |     |22 |      |26|

+---+     +---+     +---+      +--+

For example in the figure above the sum of the cousin of 9 is 20 = 20. Sum of the cousin of 25 is 7 and the sum of the cousin of 22 is 5+8 = 13.

Assume that you are using the BinaryTreeNode structure as defined below:

**struct** **BinaryTreeNode** {

   int data;

   BinaryTreeNode\* left;

   BinaryTreeNode\* right;

};

**N:28.**

Write a function **getDistance** that takes the root of the binary search tree and two values a,b. Your function should return distance between the two values. If one of the values or two of the values are missing you should return -1.

          +---+

          |10 |

          +---+

         /     \

     +---+     +---+

     | 4 |     |16 |

     +---+     +---+

    /     \         \

+---+     +---+     +---+

| 3 |     | 9 |     |20 |

+---+     +---+     +---+

         /         /      \

     +---+      +---+   +---+

     | 7 |      |18 |   |25 |

     +---+      +---+   +---+

    /      \           /      \

+---+     +---+     +---+     +---+

| 5 |     | 8 |     |22 |     |26 |

+---+     +---+     +---+     +---+

For example distance between 3 and 9 is 2, there are two edges from 3 to 9, 3-4-9. Distance between 9 and 20 is 4, 9-4-10-16-20. Distance between 18 and 26 is 3, 18-20-25-26.

Assume that you are using the BinaryTreeNode structure as defined below:

**struct** **BinaryTreeNode** {

   int data;

   BinaryTreeNode\* left;

   BinaryTreeNode\* right;

};