Trees : Level 3
Lesson 5

1. fixed-two-nodes-of-a-bst
2. Sorted-list-to-bst
3. Merge-two-bst-s
4. Largest-bst

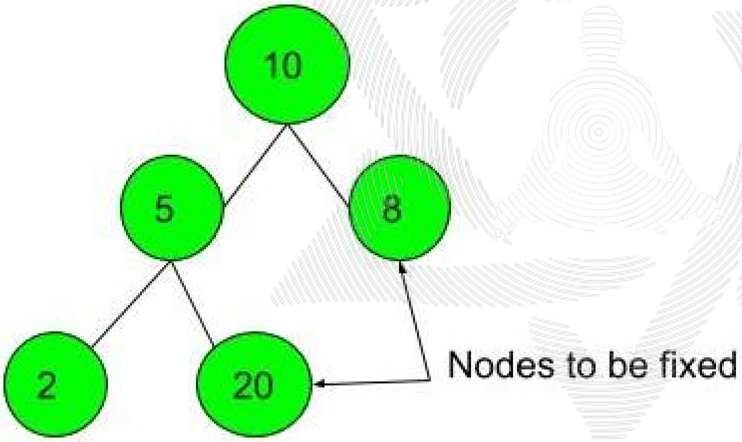
Fixing Two nodes of a BST

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST by swapping them back. Do not change the structure of the tree.
Note: It is guaranteed that the given input will form BST, except for 2 nodes that will be wrong. All changes must be reflected in the original linked list.

Example 1:
Input:

```
      10
     /    \
    5      8
   / \
  2   20
```

Output: 1
Explanation:



Example 2:

Input:

```
        11
       /    \
      3      17
       \    /
        4  10
```

Output: 1
Explanation:
By swapping nodes 11 and 10, the BST can be fixed.

Expected Time Complexity : O(n)
Expected Auxiliary Space : O(1)

Constraints:
1 <= Number of nodes <= 1000

```
void chk(Node* root, Node* &prev, Node* &node1, Node* &node2)
{
  if(!root)   return;
  chk(root->left,prev,node1,node2);
  if(prev && prev->data>root->data)
  {
     if(!node1)
     {
        node2=root;
        node1=prev;
     }
     else node2=root;
  }
  prev=root;
  chk(root->right,prev,node1,node2);
}
struct Node correctBST( struct Node root )
{
  // add code here.
  int min=INT_MIN,max=INT_MAX;
  Node* node1=NULL,*node2=NULL, *prev=NULL;
  chk(root,prev,node1, node2);
  swap(node1->data,node2->data);
  return root;
}
```

Sorted Link List to BST

Given a Singly Linked List which has data members sorted in ascending order. Construct a Balanced Binary Search Tree which has the same data members as the given Linked List.

Note: There might be nodes with the same value.

Example 1:

Input:
Linked List: 1->2->3->4->5->6->7
Output:
4 2 1 3 6 5 7
Explanation :
The BST formed using elements of the linked list is,

```
       4
     /   \
    2     6
  / \   / \
 1   3  5   7
```

Hence, preorder traversal of this tree is 4 2 1 3 6 5 7

Example 2:

Input:
Linked List : 1->2->3->4
Output:
3 2 1 4
Explanation:
The BST formed using elements of the linked list is,

```
     3
   /  \
  2    4
 /
1
```

Hence, the preorder traversal of this tree is 3 2 1 4
Expected Time Complexity: O(N), N = number of Nodes
Expected Auxiliary Space: O(N), N = number of Nodes

Constraints:
1 ≤ Number of Nodes ≤ 10^6
1 ≤ Value of each node ≤ 10^6

```
TNode* sortedListToBST(LNode *head)
{
  int n = countLNodes(head);
  return sortedListToBSTRecur(&head, n);
}

TNode* sortedListToBSTRecur(LNode **head_ref, int n)
{
  if (n <= 0)
     return NULL;

  TNode *left = sortedListToBSTRecur(head_ref, n/2);

  TNode *root = newNode((*head_ref)->data);
  root->left = left;

  *head_ref = (*head_ref)->next;

  root->right = sortedListToBSTRecur(head_ref, n - n / 2 - 1);
  return root;
}
```

Merge two BST 's

Given two BSTs, return elements of both BSTs in sorted form.

Example 1:

Input:

```
BST1:
       12
     /
    9
   / \
  6   11
BST2:

      8
    /  \
   5    10
  /
 2
```

Output: 2 5 6 8 9 10 11 12

Expected Time Complexity: O(M+N) where M and N are the sizes of the two BSTs.

Expected Auxiliary Space: O(Height of BST1 + Height of BST2).

Constraints:

1 <= Number of Nodes <= 100000

Method 1: Inorder traversal
Space=O(m+n)
Time=O(m+n)

Method 2: Iterative inorder traversal

1. First, push all the elements from root to the left-most leaf node onto a stack. Do this for both trees
2. Peek at the top element of each stack (if non-empty) and print the smaller one.
3. Pop the element off the stack that contains the element we just print
4. Add the right child of the element we just popped onto the stack, as well as all its left descendants.

Time Complexity: O(m+n)
Auxiliary Space: O(height of the first tree + height of the second tree)

```
void insertNodes(Node *root,stack<Node *> &s)
{
  while(root!=NULL)
  {
     s.push(root);
     root=root->left;
  }
}
void merge(Node *root1, Node *root2)
{
 stack<Node *> s1;
 stack<Node *> s2;

 insertNodes(root1,s1);
 insertNodes(root2,s2);

 while(!s1.empty() || !s2.empty())
 {
    int a,b;

    if(!s1.empty()) a=s1.top();
    else if(s1.empty()) a=INT_MAX;
```

```

        if(!s2.empty()) b=s2.top();
        else if(s2.empty()) b=INT_MAX;

        if(a<=b)
        {
            cout<<a<<" ";
            Node *temp=s1.top();
            s1.pop();
            insertNodes(temp->right,s1);
        }
        else
        {
            cout<<b<<" ";
            Node *temp=s2.top();
            s2.pop();
            insertNodes(temp->right,s2);
        }
    }
}
```

Largest BST

Given a binary tree. Find the size of its largest subtree that is a Binary Search Tree.

Note: Here Size is equal to the number of nodes in the subtree.

Example 1:

Input:
```
       1
     /   \
    4     4
  /   \
 6     8
```
Output: 1
Explanation: There's no sub-tree with size greater than 1 which forms a BST. All the leaf Nodes are the BSTs with size equal to 1.

Example 2:

Input: 6 6 3 N 2 9 3 N 8 8 2
```
          6
      /       \
     6         3
      \      /   \
       2    9     3
        \  /  \
         8 8    2
```
Output: 2
Explanation: The following sub-tree is a BST of size 2:
```
      2
    /   \
   N     8
```

Expected Time Complexity: O(N).

Expected Auxiliary Space: O(Height of the BST).

Constraints:
1 ≤ Number of nodes ≤ 10^5

1 ≤ Data of a node ≤ 10^6

```
int size(Node*root)
{
  if(root==NULL)
     return 0;
    return size(root->left)+size(root->right)+1;
}
int isBSTUtil(struct Node* node, int min, int max)
{
 if (node==NULL)
   return 1;

 if (node->data < min || node->data > max)
   return 0;

 return
  isBSTUtil(node->left, min, node->data-1) &&
  isBSTUtil(node->right, node->data+1, max);
}
int isBST(Node*root)
{
  return isBSTUtil(root,INT_MIN,INT_MAX);
}
int largestBst(Node *root)
{
  if (isBST(root))
   return size(root);
 else
  return max(largestBst(root->left), largestBst(root->right));
```

```
}
int bst(Node*root,int &min_val,int &max_val , int &ans , int&is_bst)
{
  if(!root)
  {
     is_bst=1;//empty tree
     return 0;//No node , hence 0 size
  }

  bool left_flag = false;
  bool right_flag = false;


  int min = INT_MAX;
  max_val = INT_MIN;

  int ls=bst(root->left,min_val,max_val,ans,is_bst);
  if (is_bst == 1 && root->data > max_val)
     left_flag = true;

  min = min_val;//temporary variable just to store the value of min_val
  min_val = INT_MAX;

  int rs=bst(root->right,min_val,max_val,ans,is_bst);
  if (is_bst == 1 && root->data < min_val)
     right_flag = true;



  if (min < min_val)
     min_val = min;

```

```
    if(root->data>max_val)
        max_val=root->data;
    if(root->data<min_val)
        min_val=root->data;


    if(left_flag && right_flag)
    {
        if (ls + rs + 1 > ans)
            ans = ls + rs + 1;
        return ls + rs + 1;
    }
    else
    {
        // Since this subtree is not BST,
        // change is_bst
        is_bst = 0;
        return 0;
    }
}
// Return the size of the largest subtree which is also a BST
int largestBst(Node *root)
{
    int min_val=INT_MAX,max_val=INT_MIN;
    int ans=0,is_bst=0;
    bst(root,min_val,max_val,ans,is_bst);
    return ans;
         //Your code here
}
```