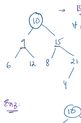
1a)

→ BST ?

left node < root.data < right node

for all nodes this condition should be satisfied

2a)

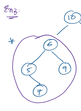


→ BST

∀ all nodes = [All Nodes in left subtree < Node.data < All Nodes in right subtree]

∀ for all Node = [All nodes in left subtree < Node.data < All nodes in right subtree] → 2

Eg:



∀ for [max element in LST < Node.data < min element in RST] → 1

Left Sub tree — Right Sub tree

1 : LST < root.data < RST

Inorder : LST, root.data, RST

Recall if this Inorder is BST & stored in ascending

2b) Given a BST search for a key.

↳ Ans: If present return T/else return False

```
bool search(Node *root, int k){
    if(root == NULL) {return False;}
    if(root->data == k)
        return true;
    if(root->data > k){
        // All elements on right are greater > k.
        return search(root->left, k)
    }
    else{
        return search(root->right, k)
    }
}
```

k = 9 : True.

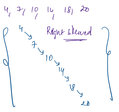
TC: O(H)

H is height of tree

min : log N : [Balanced Binary Search Tree] → AVL Tree, Red black Tree

max : N-1 : [Skewed Tree]

Eg: 4, 7, 10, 14, 18, 20



Right skewed

2c) Given a BST check if it is balanced or not?

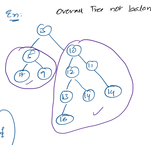
for every node:

|H(LST) − H(RST)| ≤ 1

```
bool Balanced(Node root){
    if(root == NULL) {return True}
    bool l = Balanced(root->left)
    bool r = Balanced(root->right)
    int lh = Height(root->left)
    int rh = Height(root->right)
    if( l && r && abs(lh-rh) ≤ 1){
        return True;
    }
    else return False;
}
```

O(N)

Eg:



Overall Tree not balanced

We are doing it for all nodes : TC: O(N²N)

For every node :

→ LST have to balanced

→ RST have to balanced

→ Current Node also have to be balanced

TC: O(N³)

Note: If all Node are balanced then BT is balanced.

→ Diameter

→ Max sum path

} Similar kind of problems

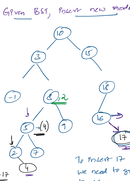
[Recursive inside recursion]

```
bool ans = True;
int helpHt(Node *root){
    if (root == NULL) return -1;
    int l = helpHt(root->left)
    int r = helpHt(root->right)
    // Check if current node balanced or not
    if( abs(l-r) > 1){ // Node not balanced // Entire BT not balanced
        ans = False;
    }
    Overall height
    return max(l,r)+1
}
```

TC: O(N) SC: O(H)

We have to traverse on all nodes, there's why: O(N)

3) Given BST, insert new node k.



K=17

K=4

To insert 4 we need to go to 2

To insert 17 we need to go to 16

K=8 (cannot insert)

```
Node* insert(Node *root, int k){
    if(root == NULL){
        return new Node(k);
    }
    Node * temp = root;
    while( root != NULL && root->data != k)
        if(root->data > k){
            // To the left of root
            if(root->left != NULL){ root = root->left}
            else { root->left = new Node(k);
            }
        else if(root->data < k){
            // To the right of root
            if(root->right != NULL) { root = root->right}
            else { root->right = new Node(k)
            }
    return temp;
}
```

Cond1 → Not necessary

Cond2

log N

Inorder

Tree Map/Ordered Map → BST →

Key Value → Based on this we compare data & do other left or right

→ This can be anything. All these node values/ information we are storing

→ Inorder traversal :

4 6 7 8 8 8 10

(Value) condition.

== (for all scenario) goto left)

or

== (for all scenario) goto right)

4a) Given BT check BST or not?

Solutions

1) Inorder sorted or not

→ Store entire inorder

→ check true or not

TC: O(N) SC: O(N)

2) To check within for a node

→ current node value we compare with prev

→ take a extra variable : prev

if(prev > Node->data){

ans = False;

}

prev = Node->data

Add this block of data in inorder traversal.

prev: Global variable

prev: Initialize = INT_MIN

TC: O(N) SC: O(1)

LST in range, RST in range: T or in range

LST in range, RST in range, T in range

[l, r]

root.data

[l, root.data-1] [root.data+1, r]

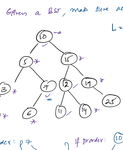
Narrowing range to each step

Sol 3

```
bool CheckBST(Node *root, int l, int r){
    if(root == NULL) { return True}
    bool BL = check BST(root->left, l, root->data-1);
    // bool BR = check BST (root->right, root->data+1, r);
    if((BL)&&(BR)&&(l <= root->data) && (root->data <= r)){
        return True;
    }
    else {
        return False;
    }
}
```

TC: O(N) SC: O(H)

4b) Given a BST, trim BST all nodes are within given range [l, r]

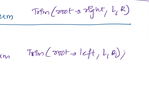


L = [7, 12]

Final BST

If Inorder :

If Preorder :

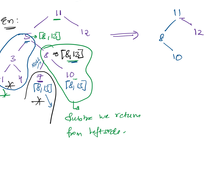


Eg: Do preorder traversal if a node data is within the range then insert that data in a new BST.

```
Node* Trim(Node *root, int l, int r){
    if(root == NULL) return NULL
    Ans: Given a root node, trim all node outside range &
    return root node of new BST.
    if( l <= root->data && root->data <= r){
        // root is passed prev
        root->left = Trim(root->left, l, r)
        root->right = Trim(root->right, l, r)
        return root;
    }
    else if( root->data < l){
        // root node & entire LST < l Not needed
        return Trim(root->right, l, r)
    }
    else {
        return Trim(root->left, l, r);
    }
}
```

root ≠ new LST

[l = 7, r = 12]



Solution we return from leftmost.

TC: O(N)

SC: O(H)

5a)

→ Sorted DLL → BBST

→ Deletion to BST

→ 2 nodes are swapped in BST find them

→ No of permutations of given data will give same BST structure

→ Convert BST → BST with same structure.

→ (Fun more questions)

5b) Given k find kth + k th in BST

→ greatest element < k to BST

```
int kth(Node *root, int k){
    int ans = INT_MIN;
    while( root != NULL){
        if( root->data == k)
            return k;
        else if( root->data < k){
            ans = root->data;
            root = root->right // update ans here for better result
        }
        else {
            root = root->left; // goto left for ans
        }
    }
    return ans;
}
```

→ Inorder

→ Fn : optional

Left

Inorder

→ Sorted → BST