1.Find a value in a BST:

```
bool search(struct Node* root, int x)
{
   if(!root || root->data==x)return root;
   if(root->data>x)return search(root->left,x);
   if(root->data<x)return search(root->right,x);
}
```

Complexity:Time=space=O(height).

2.Deletion of a node in a BST:

```
int minvalue(Node* root)
{
   int min=root->data;
   while(root->left)
   {
      min=root->left->data;
      root=root->left;
   }
   return min;
}
struct Node* deleteNode(struct Node* root, int key)
{
   if(!root)return NULL;
   if(root->data>key)root->left=deleteNode(root->left,key);
   else if
(root->data<key)root->right=deleteNode(root->right,key);
   else
```

```
    {
        if(!root->left)return root->right;
        else if (!root->right)return root->left;
        else
        {
            root->data=minvalue(root->right);
            root->right=deleteNode(root->right,root->data);
        }
    }
    return root;
}
```

Complexity:Time=O(height)

3.Find min and max value in a BST:

```
int minValue(struct Node *root) {
    if(!root)return -1;
    int min=root->data;
    while(root->left)
    {
        min=root->left->data;
        root=root->left;
    }
    return min;
}
```

4.Find inorder successor and inorder predecessor in a BST:

Method 1:
Find inorder traversal of tree and then left and right elements of given key is pre and suc respectively.

complexity=O(N)   Space=O(N)

Method 2:

```
void findPreSuc(Node* root, Node*& pre, Node*& suc, int
key)
{
    if(!root)return;
    if(root->key==key)
    {
        if(root->left)
        {
            Node* t=root->left;
            while(t->right)t=t->right;
            pre=t;
        }
       if(root->right)
        {
            Node* t=root->right;
            while(t->left)t=t->left;
            suc=t;
        }
    }
```

```
    else if(root->key > key)
    {
        suc=root;
        findPreSuc(root->left,pre,suc,key);
    }
    else
    {
        pre=root;
        findPreSuc(root->right,pre,suc,key);
    }
}
```

5.Check for BST:

Method 1:Find min from right subtree and max from left subtree.

Complexity=Time=O(N^2)

Method 2:
Inorder traverse the whole tree and check if prev->data <root->data

```
class Solution
{
    public:
    Node* prev=NULL;
    bool isBST(Node* root)
    {
```

```
        if(root)
        {
            if(!isBST(root->left))return 0;

            if(prev && prev->data >= root->data)return 0;

            prev=root;

            return isBST(root->right);
        }
        return 1;
    }
};
```

Complexity:O(N)   Space=O(1)

Note:if we store whole inorder traversal then space complexity will increase—>O(N)

6.Populate Inorder successor of all nodes:

Given a Binary Tree, write a function to populate next pointer for all nodes. The next pointer for every node should be set to point to inorder successor.

```
class Solution
{
public:
    Node* prev=NULL;
    void populateNext(Node *root)
    {
```

```
        if(!root)return;
        populateNext(root->left);
        if(prev)prev->next=root;
        prev=root;
        populateNext(root->right);
    }
};
```

7.Find LCA of 2 nodes in a BST:

```
Node* LCA(Node *root, int n1, int n2)
{
   if(!root)return NULL;
   if(root->data < n1 && root->data < n2)return
LCA(root->right,n1,n2);
   if(root->data > n1 && root->data > n2)return
LCA(root->left,n1,n2);

    return root;
}
```

Complexity=O(height)

8.Construct BST from preorder traversal:

Method 1:
We can check one by one actual position of every element which will take O(N) fro every element if tree is skewed,total complexity

Time=O(N^2)

Method 2:Using sorting

We can sort the pre order traversal which will become inorder traversal which can be solved easily then…

Complexity:Time=O(NlognN)+O(N)

Method 3:Using bound concept

Left subtree—>use node->data as bound
Right subtree→use bound itself subtree
Upper bound concept

```
Node* construct_bst(int pre[],int i,int bound,int n)
{
   if(i==n || pre[i]>bound)return NULL;
   Node* node=new Node(pre[i++]);
   node->left=construct_bst(pre,i,node->data,n);
   node->right=construct_bst(pre,i,bound,n);
   return node;
```

```
}
struct Node *constructTree(int n, int pre[], char preLN[])
{
   return construct_bst(pre,0,INT_MAX,n);
}
```

Complexity;Time=O(N)

9.Binary Tree to BST:

```
class Solution{
  public:
    void inorder(Node* root,vector<int>& v)
    {
       if(!root)return;
       inorder(root->left,v);
       v.push_back(root->data);
       inorder(root->right,v);
    }
    void inorder_bst(Node* root,vector<int>& v,int &i)
    {
       if(!root)return ;
       inorder_bst(root->left,v,i);
       root->data=v[i++];
       inorder_bst(root->right,v,i);
    }
    Node *binaryTreeToBST (Node *root)
    {
       vector<int> v;
```

```
        inorder(root,v);
        sort(v.begin(),v.end());
        int i=0;
        inorder_bst(root,v,i);
        return root;
    }
};
```

Complexity:O(NlogN)

10.Convert a normal BST into a Balanced BST:

Method 1:
A Simple Solution is to traverse nodes in Inorder and one by one insert into a self-balancing BST like AVL tree. Time complexity of this solution is O(n Log n) and this solution doesn’t guarantee.

Method 2:
1.Traverse given BST in inorder and store result in an array. This step takes O(n) time. Note that this array would be sorted as inorder traversal of BST always produces sorted sequence.
2.Build a balanced BST from the above created sorted array using the recursive approach discussed here. This step also takes O(n) time as we traverse every element exactly once and processing an element takes O(1) time.

```
Node* get_BBST(vector<int>& in,int st,int ed)
{
```

```
    if(st>ed)return NULL;
    int mid=(st+ed)/2;
    Node* node=new Node(in[mid]);

    node->left=get_BBST(in,st,mid-1);
    node->right=get_BBST(in,mid+1,ed);

    return node;

}
void inorder(Node* root,vector<int>& in)
{
    if(!root)return;
    inorder(root->left,in);
    in.push_back(root->data);
    inorder(root->right,in);
}
Node* buildBalancedTree(Node* root)
{
        vector<int> in;
        inorder(root,in);
        return get_BBST(in,0,in.size()-1);
}
```

11.Merge two BST [V.V.V>IMP]:

Method 1: (Insert elements of the first tree to second):

Take all elements of first BST one by one, and insert them into the second BST. Inserting an element to a self balancing BST takes Logn time (See this) where n is size of the BST. So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1). The value of this expression will be between mLogn and mLog(m+n-1). As an optimization, we can pick the smaller tree as first tree.

Method 2 (Merge Inorder Traversals):

1.Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.
2.Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.
3.The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.
4.Construct a balanced tree from the merged array using the technique discussed in this post. This step takes O(m+n) time.

Time complexity of this method is O(m+n) which is better than method 1. This method takes O(m+n) time even if the input BSTs are not balanced.

12.Find Kth largest element in a BST:

Method 1:
We ca n store inorder traversal of tree in an array the return arr[n-k];

Complexity:Time=O(N)   Space=O(n)

Method 2:By using a index variable and when it becomes index==k then return root->data

```
class Solution
{
   public:
   int ans=-1;
   void get_kthLargest(Node* root,int k,int &idx)
   {
      if(!root)return ;
      get_kthLargest(root->right,k,idx);
      if(idx==k)
      {
         ans=root->data;
         idx++;
         return;
      }
      else idx++;
      get_kthLargest(root->left,k,idx);
   }
   int kthLargest(Node *root, int K)
   {
```

```
        int idx=1;
        get_kthLargest(root,K,idx);
        return ans;
    }
};
```

Complexity:Time=O(K+H)   Space=O(H)

13.Find Kth smallest element in a BST;

Method 1:
Same like above

Method 2:

```
class Solution
{
    public:
    int ans=-1;
    void get_kthLargest(Node* root,int k,int &idx)
    {
        if(!root)return ;
        get_kthLargest(root->left,k,idx);
        if(idx==k)
        {
            ans=root->data;
            idx++;
            return;
```

```
        }
        else idx++;
        get_kthLargest(root->right,k,idx);
    }
    int KthSmallestElement(Node *root, int K)
    {
        int idx=1;
        get_kthLargest(root,K,idx);
        return ans;
    }
```

Complexity:Time=O(K+height)   Space=O(1)

14.Count pairs from 2 BST whose sum is equal to given value "X":

Method 1:
For every element in BST_1,we will search an element in BST_2 and if found then increment counter otherwise continue:

Complexity:Time=O(N*height)   Space=O(1)

Method 2:Using Maps

```
class Solution
{
public:
    int c=0;
```

```
    void get_pair(Node* root,unordered_map<int,int>& m,int k)
    {
       if(!root)return;
       get_pair(root->left,m,k);
       if(m.find(k-root->data)!=m.end())c++;
       //else m[root->data]++;
       get_pair(root->right,m,k);

    }
    void inorder(Node* root,unordered_map<int,int>& m)
    {
       if(!root)return;
       inorder(root->left,m);
       m[root->data]++;
       inorder(root->right,m);
    }
    int countPairs(Node* root1, Node* root2, int x)
    {
       unordered_map<int,int> m;
       inorder(root2,m);
       get_pair(root1,m,x);
       return c;
    }
};
```

Complexity:Time=O(N)   space=O(n)

15.Find the median of BST :

Method 1:
Traverse the entire tree in orderly and store elements in an array and return median accordingly.

Foreven number elements–return((array[n/2]+array[n/2-1])/2);
For dd–return array[n/2];

Complexity:Time=O(N)   Space=O(N)

Method 2:Space=O(1)

```
void count_nodes(Node* root,int &c){
   if(!root) return;
   count_nodes(root->left,c);
   c++;
   count_nodes(root->right,c);
}

void func(Node* root,Node* &cur,Node* &prev,int &c,int k,int
&f){
   if(!root) return;
   func(root->left,cur,prev,c,k,f);
   if(prev==NULL){
      prev = root;
      c++;
   }
   else if(c==k){
      c++;
```

```
      cur = root;
      f = 1;
      return;
   }
   else if(f==0){
      c++;
      prev = root;
   }
   func(root->right,cur,prev,c,k,f);
}

float findMedian(struct Node *root)
{
     //Code here
     int n = 0;
     count_nodes(root,n);
     Node* cur = NULL;
     Node* prev = NULL;
     int c = 1;
     int x = (n/2)+1;
     int f = 0;
     func(root,cur,prev,c,x,f);
     if(n&1){
        float ans = (cur->data)*1.0;
        return ans;
     }
     else {
        float ans = ((cur->data+prev->data)*1.0)/(2*1.0);
        return ans;
     }
```

}

Time=O(n)   BUT   Space=O(1)

16.Count BST nodes that lie in a given range:

Method 1:
Use inorder traversal and check one by one.
Complexity ,time=O(N)

```
 void inorder(Node* root,int l,int h,int &c)
   {
       if(!root)return;
       inorder(root->left,l,h,c);
       if(root->data>=l && root->data<=h )c++;
       inorder(root->right,l,h,c);
   }
   int getCount(Node *root, int l, int h)
   {
     int c=0;
     inorder(root,l,h,c);
     return c;
   }
```

Method 2:

```
int getCount(Node *root, int l, int h)
   {
```

```
    if(!root)return 0;
    if(root->data==l && root->data==h)return 1;
    if(root->data>=l && root->data<=h)
    return
1+getCount(root->left,l,h)+getCount(root->right,l,h);
    if(root->data < l)return getCount(root->right,l,h);
    else return getCount(root->left,l,h);
  }
```

Complexity:
Time complexity of the above program is O(h + k) where h is height of BST and k is number of nodes in given range.

17.Replace every element with the least greater element on its right:

Method 1:
For every element we will find least element on right hand side it willl take O(N^2) time complexity.

Method 2:

We will build a BST by iterating the array from left to right

```
struct Node{
  int data;
  Node* left;
  Node* right;
  Node(int val)
  {
```

```
        data=val;
        left=right=NULL;
    }

};
class Solution{
    public:
    Node* insert(Node* root,Node* &suc,int val)
    {
        if(!root)return root=new Node(val);
        if(val<root->data)
        {
            suc=root;
            root->left=insert(root->left,suc,val);
        }
        else
if(val>=root->data)root->right=insert(root->right,suc,val);

        return root;
    }
    vector<int> findLeastGreater(vector<int>& arr, int n) {
       // int n=arr.size();

        Node* root=NULL;
        for(int i=n-1;i>=0;i--)
        {
            Node* suc=NULL;
            root=insert(root,suc,arr[i]);
            if(suc)arr[i]=suc->data;
            else arr[i]=-1;
```

```
        }
        return arr;
    }
};
```

Expected Time Complexity: O(N* log N)
Expected Auxiliary Space: O(N)

18.Preorder to Postorder :

Given an array arr[] of N nodes representing preorder traversal of BST. The task is to print its postorder traversal.

```
class Solution{
public:
    //Function that constructs BST from its preorder traversal.
    struct Node* insert(Node* root,int val)
    {
        if(!root)return root=newNode(val);
        if(root->data>val)root->left=insert(root->left,val);
        else if(root->data<=val)root->right=insert(root->right,val);
        return root;
    }
    Node* post_order(int pre[], int size)
    {
        Node* root=NULL;
        for(int i=0;i<size;i++)
        {
            root=insert(root,pre[i]);
        }
        return root;
    }
};
```

Expected Time Complexity: O(N).
Expected Auxiliary Space: O(N).

19.Check whether BST contains Dead end:

Here Dead End means, we are not able to insert any element after that node.

Method 1:
For every element check if ele-1 & ele+1 exists in BST,then return true.

Complexity:Time=O(N*height) Space=O(1)

Method 2:Using maps

We can store all elements in the map and then check for all BST elements whether ele-1 & ele+1 exists the return true.

Complexity:Time=O(N) space=O(N)

Method 3:Using upper bound and lower bound

```
bool deadend(Node* root,int lo,int up)
{
   if(!root)return 0;
   if(lo==up)return 1;
   return deadend(root->left,lo,root->data-1) or
deadend(root->right,root->data+1,up);
```

```
}
bool isDeadEnd(Node *root)
{
   return deadend(root,1,INT_MAX);
}
```

20.Largest BST in a Binary Tree [V.V.V.V.V IMP]:

We will create a vector for every node which contain four values (BST,count,lower bound,upper bound).

```
class Solution{
   public:
   vector<int> get_largestBST(Node* root)
   {
      if(!root)return {1,0,INT_MAX,INT_MIN};
      if(!root->left &&
!root->right)return{1,1,root->data,root->data};

      vector<int> l=get_largestBST(root->left);
      vector<int> r=get_largestBST(root->right);
      if(l[0] && r[0])
      {
         if(root->data > l[3]  && root->data <r[2])
         {
            int x=l[2];
            int y=r[3];
            if(x==INT_MAX)x=root->data;
            if(y==INT_MIN)y=root->data;
            return {1,l[1]+r[1]+1,x,y};
```

```
        }
        else return {0,max(l[1],r[1]),0,0};
    }
    else return {0,max(l[1],r[1]),0,0};
}
int largestBst(Node *root)
{
    vector<int> v=get_largestBST(root);
    return v[1];
}
};
```

For confusion refer:

Largest BST in a Binary Tree | BST | Love Babbar DSA She…

Expected Time Complexity: O(N).