Diameter of a Binary Tree:

```
class Solution {
public:
    typedef TreeNode Node;
    int maxDiameter=0;

    int getDiameter(Node*root){
        if(root==NULL){
            return 0;
        }
        int leftHeight=getDiameter(root->left);
        int rightHeight=getDiameter(root->right);
        maxDiameter=max(maxDiameter,leftHeight+rightHeight);
        return 1+max(leftHeight,rightHeight);
    }
    int diameterOfBinaryTree(TreeNode* root) {
        int ans=getDiameter(root);
        return maxDiameter;
    }
};
```

Check if subtree:

```
class Solution
{
  public:
    //Function to check if S is a subtree of tree T.
    bool ans=false;
    string getSubtreeString(Node*S){
        if(S==NULL){
            return "";
        }

        string left=getSubtreeString(S->left);
        string right=getSubtreeString(S->right);
        return "("+left+to_string(S->data)+right+")";
    }
    string getTreeString(Node*T,string s){
         if(T==NULL){
            return "";
        }

        string left=getTreeString(T->left,s);
        string right=getTreeString(T->right,s);
```

```
        string final="("+left+to_string(T->data)+right+")";
        if(final==s){
            ans=true;
            return "";
        }
        return final;
    }
    bool isSubTree(Node* T, Node* S)
    {
      string s=getSubtreeString(S);
      string t=getTreeString(T,s);
      return ans;

    }
};
```

Left View of a Binary Tree:

```
vector<int> leftView(Node *root)
{
   vector<int>ans;
   if(root==NULL){
       return ans;
   }
   queue<pair<Node*,int>>q;
   map<int,int>mp;
   q.push({root,0});
   while(!q.empty()){
       pair<Node*,int>p=q.front();
       Node*node=p.first;
       int level=p.second;
       q.pop();

       if(mp.find(level)==mp.end()){
           mp[level]=node->data;
       }
       if(node->left!=NULL){
           q.push({node->left,level+1});
       }

        if(node->right!=NULL){
           q.push({node->right,level+1});
       }
   }
   for(auto it:mp){
       ans.push_back(it.second);
```

```
  }
  return ans;

}
```

Right View of a Binary Tree:

```
class Solution {
public:
typedef TreeNode Node;
    vector<int> rightSideView(TreeNode* root) {
        vector<int>ans;
   if(root==NULL){
       return ans;
   }
   queue<pair<Node*,int>>q;
   map<int,int>mp;
   q.push({root,0});
   while(!q.empty()){
       pair<Node*,int>p=q.front();
       Node*node=p.first;
       int level=p.second;
       q.pop();

       if(mp.find(level)==mp.end()){
           mp[level]=node->val;
       }
       if(node->right!=NULL){
           q.push({node->right,level+1});
       }

        if(node->left!=NULL){
           q.push({node->left,level+1});
       }
   }
   for(auto it:mp){
       ans.push_back(it.second);
   }
   return ans;

    }
};
```

Predecessor and Successor:

```
class Solution
{
    public:
    void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
    {
       pre=NULL;
       suc=NULL;

       Node*curr=root;
       while(curr!=NULL){
           if(curr->key>=key){
               curr=curr->left;
           }
           else{
               pre=curr;
               curr=curr->right;
           }
       }

       curr=root;

       while(curr!=NULL){
           if(curr->key<=key){
               curr=curr->right;
           }
           else{
               suc=curr;
               curr=curr->left;
           }
       }
    }
};
```

Find the Closest Element in BST:

```
class Solution
{
    public:
    //Function to find the least absolute difference between any node
      //value of the BST and the given integer.
    int minDiff(Node *root, int K)
    {
        Node*pre=NULL;
       Node*suc=NULL;

       Node*curr=root;
```

```
        while(curr!=NULL){
            if(curr->data>K){
                curr=curr->left;
            }
            else{
                pre=curr;
                curr=curr->right;
            }
        }

        curr=root;

        while(curr!=NULL){
            if(curr->data<K){
                curr=curr->right;
            }
            else{
                suc=curr;
                curr=curr->left;
            }
        }
    //   cout<<pre->data<<","<<suc->data<<endl;
        if(pre==NULL){
            return abs(K-suc->data);
        }
        else if(suc==NULL){
            return abs(pre->data-K);
        }
        else if(pre==NULL&&suc==NULL){
            return 0;
        }
        else if(abs(K-pre->data)<abs(suc->data-K)){
            return abs(pre->data-K);
        }
        else{
            return abs(suc->data-K);
        }
    }
};
```