Trees.

```
Node * buildTree()
  {
      int data;
      cin>>data;
      if (data==-1) return NULL;
    Node *root =  new node(data);
     root->left = buildTree();
     root->right =buildTree();
     return root;
  }
```

Level Order Traversal :

```
void  lvlOrderTraversal (Node*root)
 {
    queue<Node *> q;
    q.push(root);
     q.push(NULL);
    while (!q.empty())
    {      //nikal
     Node *front = q.front();
      q.pop();   if (front ==NULL){ cout<<endl;
                                   if(!q.empty())
                                     q.push(NULL);}
 else{cout<< front->data << " ";
     }          //leave children
    if (front->left !=NULL)
          q.push (front->left)
     if (front->right!= NULL)
          q.push(front->right)
    }}
```

~~[illegible]~~ ~~[illegible]~~ Height of BT -

```
int height (Node* root)
 {
   if (root == NULL)
    return 0;

   if (height->left == NULL &&
            height->right == NULL)
    return 0;

   int leftAns = height(root->left);
   int rightAns = height(root->right);

   return 1+max (leftAns, RightAns);
   }.
```

bcz we need edges not no of nodes

for max depth, remove this.

Diameter of Tree

```
int diameter (Treenode * root)
 {
  if (root == NULL) return 0;

  int op1 = diameter (root ->left);
  int op2 = diameter (root ->right);
  int op3 = 1 + height (root -> left) +
                   height( root -> right);
  return max(op1, max (op2, op3));
  }
```

diameter (root) -1;

Balanced Tree :

Not optimised - $O(N^2)$

```
bool isBalanced (TreeNode* root)
{
 if (root ==NULL)
     return true;

 bool leftAns = isBalanced (root->left);
 bool ~~left~~ rightAns =      "        "  right

 bool diff = abs(height(root->left) -
             height (root->right)) <=1;

 if (leftAns && rightAns && diff)
 return true;
  else
 return false;
  }
```

Optimised -

```
pair<int,bool> solve(Treenode * root) {
      if(root ==NULL)
           return make pair (0,true);

  pair<int,bool> leftAns = solve(root->left);
     "    "    "   rightAns = solve (root->right);
int leftHeight = leftAns. first;
int rightHeight = rightAns. first;

bool diff = abs (leftHeight - rightheight) <=1;

bool leftbalanced  = leftAns. second;
 bool right   "    = rightAns. second;

if (leftbalanced && rightbalanced && diff)
{
   return make_pair(max (leftHeight,
                    rightHeight) +1, true);
 }
 else
      ~~false~~
  return make_pair (max (leftHeight,
                 }  rightHeight) +1, false);
ans. second.
```

Path Sum.

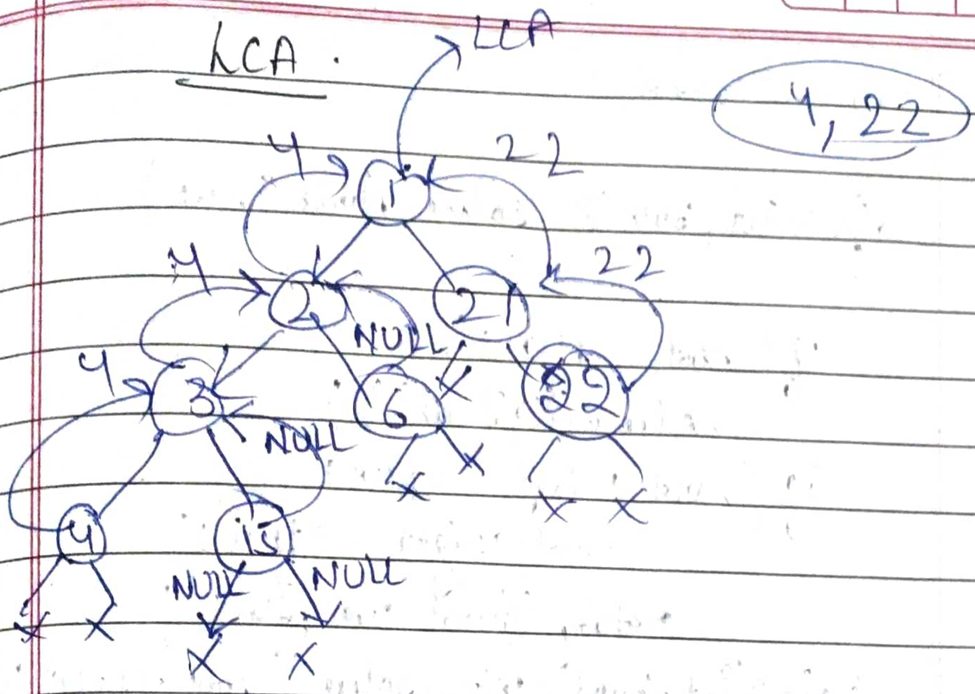
```
bool hasPathSum(Treenode* root,
                        int target)
  {
   if(root == NULL)
     return false;

  if(root->val == target &&
     root->left == NULL &&
     root->right == NULL)
   return true;

  bool leftSum = hasPathSum(root->left,
                          target - root->val);
       right
  bool ~~left~~Sum = '' ''      '' (root->right,
                          target - root->val);

   return
  ~~if~~( leftSum || rightSum)
      ~~return true~~;
  }.
```

LCA .



```
TreeNode *LCA( Treenode * root , Treenode * p, *q)
  {
  if (root ==NULL)
     return NULL;
  if (root ==p || root ==q)
     return ~~p~~root;
  else
   {
    Treenode *left = LCA(root->left, p, q);
    Treenode *right = LCA( "  ->right, p, q);
    if (leftAns != NULL && rightAns ~~!~~= NULL)
          return leftAns;
    ~~else~~ if (leftAns == NULL) && rightAns != NULL)
          return rightAns;
     if (leftAns != NULL && rightAns != NULL)
          return ~~or~~ root;
    }
```

⇒ Sum Tree

```
pair <int, bool>  solve(Node * root)
 {
    if( root == NULL)
         return({0, True});
    if (root → left == NULL &&
              root → right == NULL)
          return ({root → val, True}';
   pair <int, bool> left = solve (root → left);
    "     "     "  right =  "     "   right

   if (left.second && right.second &&
       root → val == left.first + right.first))
     return({2 * root → val, true})
   else
     return (root → val + left.first + right.
                       first, false })
  }
```

Path Sum 3

```
void solve(root ,int target, longlongint sum)
{
  if(root == NULL)
    return NULL;
  if(target ==sum)
    ~~return~~ count++;
  if(root->left)
     solve(root->left, target, sum+ root->left->val);
  if(root->right)
     solve(root->right, target, sum+ root->right->val);
 }
Int pathSum(root, targetSum)
    {
    if(root ==NULL)
       return 0;

  Solve (root, targetSum, root->val);
  pathSum(root->left, targetSum);
    "   "      "right
  return count;
  }
```

K^th Ancestor

BF -

```
~~void~~ int  solve( root, data, k, vector
                            <int> &ans)
   {
 if (root == NULL)
      return 0;
 if (root → val == data)
    { int index = ans.size - k;
   if (index < 0) return -1;
    return ans[ans.size() - k];
    }
 ans.pushback( root → val);
 solve ( root → left, data, k, ans);
   "        "      right    "      "    "
  ans. pop_back();
     }
  return ans;
```

Optimised -

```
void solve (root, data, k)
  {
   if (root == NULL)
      return;
   if (root → val == data)
   { found = true;
      if (k == 0)
        {    ~~root~~ ans = root → val;
             k = INT_MAX;
        }
   k--
    return; }
solve ( root → left, data, k);
solve ( root → right  "    " );

(If found = false)

if (k == 0 && found == true)
{
      ~~cout~~ ans << root → val ; k = INT_MAX ;
}
else if (found == true)
      k-- ;
}.
```

Wrong approach since it will do k-- ~~even in~~ 2 times ie. in left & right too. So wrong.

~~Binary tree~~

Build Tree from Inorder & P~~re~~ostorder

```
Node* buildPosterder (vector<int> in,
                      vector<int> post,
                      int &postOrderIndex,
                      int inStart, int inEnd)
{
   if (postOrderIndex < 0 || instart > inEnd)
   {     return null ;
   int element = post [postOrderIndex--] ;
   Node* root = new Node(element);

   int pos = position(in, element, inStart, inEnd);
root -> right = buildPostOrder (in, post,
                  postOrderIndex, pos+1, inEnd);
root -> left = buildPostOrder (in, post,
                  postOrderIndex, inStart, pos-1);
  return root;
 }
```

* Can be furthur optimised to find position of element using hashmap instead of using linear search.