Binary Tree

In [ ]:

```

```

Finding max depth of binary tree using recursion
[https://leetcode.com/problems/maximum-depth-of-binary-tree/ (https://leetcode.com/problems/maximum-depth-of-binary-tree/)](https://leetcode.com/problems/maximum-depth-of-binary-tree/)

Depth First Traversal TC: O(N) SC: Best=O(log N) Worst=O(N) = O(N) stack space

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == NULL) return 0;

        auto leftHeight = maxDepth(root->left);
        auto rightHeight = maxDepth(root->right);

        return 1 + (leftHeight>rightHeight ? leftHeight : rightHeigh
t)
    }
};
```

Level Order Traversal
TC: O(N)
SC: O(N/2) = O(N) ; because the last level has n/2 nodes

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
    if (root == NULL) return 0;

    std::queue<TreeNode*> q; // Use a Queue
    q.push(root);  // Push root node to q
    q.push(NULL);
    int level=1;
    while (!q.empty()) {
        TreeNode *n = q.front();
```

```
class Solution {
public:
    int maxDepth(TreeNode* root) {

        if (root == NULL) return 0;

        std::queue<TreeNode*> q;
        TreeNode *dummy = new TreeNode();

        q.push(root);
        q.push(dummy);

        int level = 1;
        while (!q.empty()) {
            TreeNode *n = q.front();
            q.pop();

            if(q.empty()) break;

            if (n == dummy) {
                q.push(dummy);
                level = level + 1;
                continue;
            } else {
                if (n->left) q.push(n->left);
                if (n->right) q.push(n->right);
            }
        }

        return level;
    }
};
```

```
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if root is None:
            return 0
        q = []
        q.append(root)
        q.append(None)
        maxDep = 0

        while q:
            el = q.pop(0)

            if el is None:
                maxDep += 1
                if len(q) != 0:
                    q.append(None)
                continue

            if el.left:
                q.append(el.left)
            if el.right:
                q.append(el.right)

        return maxDep
```

Level Order traversal without Dummy/Sentinal; using length of the queue

```
var maxDepth = function(root) {

    // using BFS

    if(!root) {
        return 0;
    }

    const q = [root];
    let maxDepth = 0;
```

In [ ]:

```

```

DIY
[https://leetcode.com/problems/minimum-depth-of-binary-tree/ (https://leetcode.com/problems/minimum-depth-of-binary-tree/)](https://leetcode.com/problems/minimum-depth-of-binary-tree/)

Java

```
public int minDepth(TreeNode root) {
      if(root == null) return 0;
        int left = minDepth(root.left);
        int right = minDepth(root.right);
        return (left == 0 || right == 0) ? left + right + 1: Math.min
(left,right) +1;
    }



public int minDepth(TreeNode root) {
        if(root == null) return 0;

        if(root.left == null && root.right==null) return 1;

        if(root.left == null){
            return 1 + minDepth(root.right);
        }
        if(root.right == null){
            return 1 + minDepth(root.left);
        }

        return 1 + Math.min(minDepth(root.left),minDepth(root.righ
t));

    }
```

In [ ]:

```

```

DIY

[https://leetcode.com/problems/same-tree/ (https://leetcode.com/problems/same-tree/)](https://leetcode.com/problems/same-tree/)

In [ ]:

~~~
```c++
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        bool left=false;
        bool right=false;
        if(p==NULL&&q==NULL) return true;
        if(p==NULL||q==NULL) return false;

        if(p->val==q->val){
          left = isSameTree(p->left,q->left);
          right = isSameTree(p->right,q->right);
        }
        return left && right;
    }
};
```
~~~

```
public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p == null && q != null){
            return false;
        }
        if(q == null && p!= null){
            return false;
        }
        if(p==null && q== null){
            return true;
        }
        if(p == null || q == null || p.val != q.val){
            return false;
        }
        return isSameTree(p.left,q.left) && isSameTree(p.right,q.righ
t);
    }
```

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (p == NULL && q == NULL) return true;
        if (p == NULL || q == NULL) return false;
        return (p->val == q->val) && isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};



public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p==null && q== null)
         return true;

         if(p.val == q.val) {
            return  isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
         }

         return false;
    }
```

Python

```python
class Solution:
    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        if p is None and q is None:
            return True
        if (p is None and q is not None) or (q is None and p is not None):
            return False

        leftSame = self.isSameTree(p.left, q.left)
        rightSame = self.isSameTree(p.right, q.right)

        return p.val == q.val and leftSame and rightSame
```

In [ ]:

Question

https://leetcode.com/problems/symmetric-tree/ (https://leetcode.com/problems/symmetric-tree/)

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        return isSymmetricNode(root->left,root->right);
    }
     bool isSymmetricNode(TreeNode* p,TreeNode* q) {
         bool left=false;m
        bool right=false;
        if(p==NULL&&q==NULL) return true;
        if(p==NULL||q==NULL) return false;

        if(p->val==q->val){
          left = isSymmetricNode(p->left,q->right);
          right = isSymmetricNode(p->right,q->left);
        }
        return left && right;
     }
};
```

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        return self.isSymHelper(root, root)

    def isSymHelper(self, p, q):
        if p is None and q is None:
            return True
        if (p is None and q is not None) or (q is None and p is not None):
            return False

        if p.val != q.val:
            return False
        return self.isSymHelper(p.left, q.right) and self.isSymHelper(p.right, q.left)
```

In [ ]:

```

```

In [ ]:

```

```

Breadth First / Level Order Traversal, BFS = Queue

```
struct Node {
    int data;
    Node *left;
    Node *right;
};

Node* newNode(int data, Node* left=NULL, Node* right=NULL) {
    Node *temp = new Node;

    temp->data = data;
    temp->left = left;
    temp->right = right;

    return temp;
}

void leveOrderTraversal(Node *root) {
    if (root == NULL) return;

    std::queue<Node*> q; // Use a Queue
    q.push(root);  // Push root node to q

    while (!q.empty()) {
        Node *n = q.front();
        q.pop();
        cout << n->data << " ";

        if (n->left) q.push(n->left);
        if (n->right) q.push(n->right);
    }
}

int main() {

     //     10
     //    /  \
     //   20  30
     //  / \
     // 40  50
    //        \
    //        60

    Node *root = newNode(10, newNode(20, newNode(40), newNode(50, NUL
L, newNode(60)))   , newNode(30) );
    leveOrderTraversal(root);
}
```

In [ ]:

Operations in a Tree

- Add
- Remove
- Traverse
- Search

In [ ]:

Recursive implementation of DFS

In [ ]:

In [ ]:

In [ ]: