A **Tree** is a hierarchical data structure widely used in computer science for organizing data. It consists of **nodes**, where each node contains a value and references to its children (other nodes). The most common type of tree is a **binary tree**, but there are many other variations.

Memory updated

A **Tree** is a hierarchical data structure widely used in computer science for organizing data. It consists of **nodes**, where each node contains a value and references to its children (other nodes). The most common type of tree is a **binary tree**, but there are many other variations.

Basic Terminology

1. **Node**: The fundamental part of the tree. Each node contains data and references to its children.
2. **Root**: The top node of the tree, where the hierarchy starts.
3. **Parent**: A node that has references to its child nodes.
4. **Child**: A node that is a descendant of another node.
5. **Leaf**: A node that has no children.
6. **Subtree**: A portion of the tree that forms another smaller tree.
7. **Depth**: The number of edges from the root to the node.
8. **Height**: The number of edges on the longest path from a node to a leaf.
9. **Binary Tree**: A tree where each node has at most two children.

Types of Trees

1. **Binary Tree**: Each node has at most two children, commonly referred to as the left and right children.
2. **Binary Search Tree (BST)**: A binary tree where the left child of a node contains only values less than the node, and the right child contains only values greater than the node.
3. **Skewed Tree**

A **skewed tree** is a tree where all nodes have only one child. It can either be left-skewed or right-skewed:

- **Left-skewed Tree**: All nodes have only a left child.
- **Right-skewed Tree**: All nodes have only a right child.

4.**Degenerate Tree**

A **degenerate tree** is a special case of a skewed tree where each parent node has only one child. It can be considered a worst-case scenario for a binary search tree (BST), where it behaves like a linked list.

5. **Complete Tree**

A **complete tree** is a binary tree where all levels are completely filled except possibly the last level, which is filled from left to right.

Traversals:

Preorder-Traversal: https://leetcode.com/problems/binary-tree-preorder-traversal/description/

```
function getPreorderTraversal(root,preorder){

    if(root==null){

        return;

    }

    preorder.push(root.val);

    getPreorderTraversal(root.left,preorder);

    getPreorderTraversal(root.right,preorder);

}

var preorderTraversal = function(root) {

    var preorder=[];

    getPreorderTraversal(root,preorder);

    return preorder;

};
```

Inorder-Traversal:

Problem-Link: https://leetcode.com/problems/binary-tree-inorder-traversal/

```
function getInorderTraversal(root,inorder){

    if(root==null){

        return;

    }
```

```
    getInorderTraversal(root.left,inorder);
    inorder.push(root.val);
    getInorderTraversal(root.right,inorder);
}
var inorderTraversal = function(root) {
    var inorder=[];
    getInorderTraversal(root,inorder);
    return inorder;
};
```

Postorder-Traversal:

Problem-Link: https://leetcode.com/problems/binary-tree-postorder-traversal/

```
function getPostorderTraversal(root,postorder){
    if(root==null){
        return;
    }
    getPostorderTraversal(root.left,postorder);
    getPostorderTraversal(root.right,postorder);
    postorder.push(root.val);
}
var postorderTraversal = function(root) {
    var postorder=[];
    getPostorderTraversal(root,postorder);
    return postorder;
};
```

Lowest Common Ancestor:

Problem Link:
https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/description/

```
var lowestCommonAncestor = function(root, p, q) {

    var curr=root;

    while(curr!=null){

        if(p.val<curr.val&&q.val<curr.val){

            curr=curr.left;

        }

        else if(p.val>curr.val&&q.val>curr.val){

            curr=curr.right;

        }

        else{

            return curr;

        }

    }

    return null;

};
```

Maximum Depth of a Binary Tree:

Problem-Link:
https://leetcode.com/problems/maximum-depth-of-binary-tree/description/

```
function maximumDepth(root){

    if(root==null){

        return 0;
```

```
    }
    var leftDepth=maximumDepth(root.left);
    var rightDepth=maximumDepth(root.right);
    return 1+Math.max(leftDepth,rightDepth);
}
var maxDepth = function(root) {
    return maximumDepth(root);
};
```

Level-Order Traversal:

Problem-Link: https://leetcode.com/problems/binary-tree-level-order-traversal/description/

```
class myQueue {
  constructor() {
    this.queue = [];
  }

  // Enqueue operation (Add element to the end of the queue)
  push(element) {
    this.queue.push(element);
    console.log(`${element} added to the queue`);
  }

  // Dequeue operation (Remove element from the front of the queue)
```

```
pop() {
  if (this.isEmpty()) {
    console.log('Queue is empty, cannot dequeue');
    return;
  }
  const removedElement = this.queue.shift();
  console.log(`${removedElement} removed from the queue`);
  return removedElement;
}

// Peek operation (View the element at the front of the queue)
peek() {
  if (this.isEmpty()) {
    console.log('Queue is empty');
    return;
  }
  return this.queue[0];
}

// Check if the queue is empty
isEmpty() {
  return this.queue.length === 0;
}
```

```
  // Get the size of the queue
  size() {
    return this.queue.length;
  }

  // Print the queue
  printQueue() {
    console.log('Queue:', this.queue.join(', '));
  }
}

var levelOrder = function(root) {
     var levelOrderElements=[];
    if(root==null){
        return levelOrderElements;
    }
    var q=new myQueue();
    q.push(root);
    while(!q.isEmpty()){
        var n=q.size();
        var arr=[];
        for(var i=0;i<n;i++){
            var node=q.pop();
        arr.push(node.val);
```

```
            if(node.left!=null){
                q.push(node.left);
            }
            if(node.right!=null){
                q.push(node.right);
            }
            }
            levelOrderElements.push(arr);
        }
        return levelOrderElements;
    };
```