BST TREE CONSTRUCTION

```
class Node {
   constructor(value){
      this.left = null;
      this.right = null;
      this.value = value
   }
}

class BinarySearchTree {
   constructor() {
      this.root = null
   }

   insert(value){
      let current = this.root;
      const nodeToInsert = new Node(value);
      if (current === null){
         this.root = nodeToInsert;
      } else {
         while (true){
            if (current.value >= value){
                if (current.left){
                   current = current.left
                   continue
                }
               current.left = nodeToInsert;
               return this;
            } else {
                if (current.right){
                   current =  current.right
                   continue
                }
               current.right = nodeToInsert;
               return this;
            }
         }
      }
   }

   lookup(value){
      let current = this.root;
      while(current) {
         if (current.value === value) {
            return current
         } else if (current.value > value) {
            current = current.left;
```

```
        } else {
            current = current.right;
        }
    }
    return false;
}

breadthFirstSearch() {
    let currentNode = this.root;
    let list = []
    let queue = []
    queue.push(currentNode)

    while(queue.length) {
        currentNode = queue.shift()
        list.push(currentNode.value)
        if (currentNode.left){
            queue.push(currentNode.left)
        }
        if (current.right){
            queue.push(currentNode.right)
        }
    }
    return list;
}

breadthFirstSearchR(queue, list) {
    if(!queue.length) {
        return list;
    }
    let currentNode = queue.shift()
    list.push(currentNode.value)
    if (currentNode.left){
        queue.push(currentNode.left)
    }
    if (current.right){
        queue.push(currentNode.right)
    }

    return this.breadthFirstSearchR(queue, list)
}

DFSPreorderIterative() {
    let currentNode = this.root;
    let list = []
    let stack = []
    stack.push(currentNode)
```

```
    while(stack.length) {
      currentNode = stack.shift()
      list.push(currentNode.value)
      if (current.right){
        stack.unshift(currentNode.right)
      }
      if (currentNode.left){
        stack.unshift(currentNode.left)
      }
    }
    return list;
  }


  DFSInorder() {
    return traverseInorder(this.root, [])
  }

  DFSPreorder() {
    return traversePreorder(this.root, [])
  }

  DFSPostorder() {
    return traversePostorder(this.root, [])
  }
}

/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
// Iterative solution for inordertraversal
var inorderTraversal = function(root) {
  var stack = []
  var list = []
  if (!root) {
    return stack
  }
  var current = root.left
  stack = [root]
  while (current || stack.length){
```

```
    while(current){
      stack.push(current)
      current = current.left
    }

    current = stack.pop()
    list.push(current.val)
    current = current.right
  }

  return list
}


function traverseInorder(node, list){
  if (node.left){
    traverseInorder(node.left, list);
  }
  list.push(node.value)
  if (node.right){
    traverseInorder(node.right, list);
  }
  return list
}

function traversePreorder(node, list){
  list.push(node.value)
  if (node.left){
    traversePreorder(node.left, list);
  }
  if (node.right){
    traversePreorder(node.right, list);
  }
  return list
}

function traversePostorder(node, list){
  if (node.left){
    traversePostorder(node.left, list);
  }
  if (node.right){
    traversePostorder(node.right, list);
  }
  list.push(node.value)
  return list
}
```

```
function traverse(node) {
  const tree = { value: node.value };
  tree.left = node.left === null ? null : traverse(node.left);
  tree.right = node.right === null ? null : traverse(node.right);
  return tree;
}

/// LEET CODE SOLUTION
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var inorderTraversal = function(root) {
    if(!root)
       return [];
   else
       return traverseInorder(root, []);
};

function traverseInorder(node, list){
   if (node.left){
       traverseInorder(node.left, list);
   }
   list.push(node.val)
   if (node.right){
       traverseInorder(node.right, list);
   }

   return list
}
```

MEMOIZED FIB

```
function slowFib(n){
  if (n<2){
    return n
  }
  return fib(n-1) + fib(n-2)
}

function memoizedFib(){
  let cache = {}
  return function fib(n) {
    if (n in cache){
      return cache[n];
    } else {
      cache[n] = slowFib(n)
      return cache[n]
    }
  }
}
```

FROM LAST

```
function fromLast(list, n) {
  let slow = list.head;
  let fast = list.head;
  while(n>0){
    fast = fast.next
    n--
  }

  while(fast.next){
    fast = fast.next;
    slow = slow.next;
  }

  return slow
}
```

LEVEL WIDTH

```
// --- Directions
// Given the root node of a tree, return
// an array where each element is the width
// of the tree at each level.
// --- Example
// Given:
//     0
//   / |  \
// 1   2   3
// |       |
// 4       5
// Answer: [1, 3, 2]

function levelWidth(root) {
   let counter = [0]
   let array = [root, 's']

   while (array.length > 1){
      const node = array.shift();
      if(node === 's'){
         array.push(node)
         counter.push(0)
      } else {
         array.push(...node.children)
         counter[counter.length - 1]++
      }
   }
   return counter
}

module.exports = levelWidth;
```

LINKED LIST

```
// --- Directions
// Implement classes Node and Linked Lists
// See 'directions' document

class Node {
  constructor(data, next = null) {
    this.data = data;
    this.next = next;
  }
}

class LinkedList {
  constructor() {
    this.head = null;
  }

  insertFirst(data){
    const node = new Node(data, this.head);
    this.head = node;
  }

  size() {
    let current = this.head;
    let counter = 0;
    while (current){ // or while (current !== null)
      current = current.next;
      counter++;
    }
    return counter;
  }

  getFirst() {
    return this.head;
  }

  getLast() {
    let current = this.head;
    if (!current){
      return current;
    }
    while(current.next){
      current = current.next;
    }
    return current;
  }
```

```
clear() {
  this.head = null;
}

removeFirst() {
  if (this.head){
    this.head = this.head.next;
  }
}

removeLast() {
  if (!this.head){
    return;
  }

  if (!this.head.next){
    this.head = null;
    return;
  }

  let previous = this.head
  let node = this.head.next;
  while (node.next){
    previous = node;
    node = node.next;
  }
  previous.next = null;
}

insertLast(record) {
  // My Implementation
  // const newNode = new Node(record);
  // if (!this.head){
  //    this.head = newNode;
  //    return;
  // }

  // if(!this.head.next){
  //    this.head.next = newNode;
  //    return;
  // }
  // let node = this.head.next;
  // while(node.next){
  //    node = node.next;
  // }
  // node.next = newNode;

  // Solution 2
```

```
    const last = this.getLast();
    if (last){
        last.next = new Node(record);
    } else {
        this.head = new Node(record);
    }
}

getAt(index){
    // if(!this.head){ // not required since it is bieng taken care of at the bottom.
    //     return null;
    // }
    let node = this.head;
    let counter = 0;
    while(node) {
        if (counter === index){
            return node;
        }
        node = node.next;
        counter++;
    }
    return null;
}

removeAt(index){
    if (!this.head){
        return;
    }

    if (index === 0){
        this.head = this.head.next;
        return;
    }

    const previous = this.getAt(index-1);
    if (!previous || !previous.next) {
        return;
    }
    previous.next = previous.next.next;
}

insertAt(data, index){
    if (!this.head) {
        this.head = new Node(data);
        return;
    }
    if (index === 0) {
        this.head = new Node(data, this.head);
```

```
            return;
        }
        const previous = this.getAt(index - 1) || this.getLast();
        // if (!previous || !previous.next){
        //     const last = this.getLast()
        //     last.next = new Node(data);
        //     return;
        // }
        previous.next = new Node(data, previous.next)
    }

    forEach(fn){
        if (!this.head){
            return;
        }
        let node = this.head;
        while(node){
            fn(node);
            node = node.next
        }
    }
}

module.exports = { Node, LinkedList };
```

SPIRAL MATRIX

```
// --- Directions
// Write a function that accepts an integer N
// and returns a NxN spiral matrix.
// --- Examples
//  matrix(4)
//    [[1,   2,  3, 4],
//    [12, 13, 14, 5],
//    [11, 16, 15, 6],
//    [10,  9,  8, 7]]

function matrix(n) {
   var results = [];
   for (let i = 0; i < n; i++){
      results.push([]);
   }

   let counter = 1
   let startColumn = 0;
   let endColumn = n - 1;
   let startRow = 0;
   let endRow = n - 1;

   while(startColumn <= endColumn && startRow <= endRow){
      for (let i = startColumn;i <= endColumn; i++){
         results[startRow][i] = counter;
         counter++;
      }
      startRow++;
      for (let i = startRow;i <= endRow; i++){
         results[i][endColumn] = counter;
         counter++;
      }
      endColumn--;
      for (let i = endColumn;i >= startColumn; i--){
         results[endRow][i] = counter;
         counter++;
      }
      endRow--;
      for (let i = endRow;i >= startRow; i--){
         results[i][startColumn] = counter;
         counter++;
      }
      startColumn++;
   }
   return results
}
module.exports = matrix;
```

MAX BALANCED BINARY TREE

```
 /**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
// O(n2) runtime, O(n) stack space – Brute force top-down recursion:
var isBalanced = function(root) {
   if (!root){
      return true
   }

   let leftDepth = visit(root.left, 1)
   let rightDepth = visit(root.right, 1)
   if (Math.abs(leftDepth - rightDepth) <= 1 && isBalanced(root.left) && isBalanced(root.right)){
      return true
   }
   return false
};

const visit = (node, depth) => {
   if (!node){
      return depth
   }
   return Math.max(visit(node.left, depth + 1), visit(node.right, depth + 1))
}

/// O(n) runtime, O(n) stack space – Bottom-up recursion:
var isBalanced = function(root) {
   return visit(root) !== -1
};

const visit = (node) => {
  if (!node){
     return 0
  }
   let leftHeight = visit(node.left)
   if (leftHeight === -1){
      return -1
   }
   let rightHeight = visit(node.right)
   if (rightHeight === -1){
      return -1
   }
   return Math.abs(leftHeight - rightHeight) > 1 ? -1 : Math.max(leftHeight,rightHeight) + 1
}
```

MAX DEPTH OF THE TREE

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */

 // Iterative Approach
var maxDepth = function(root) {
   let depth = 0

   if (!root) {
   return 0;
   }

   let array = [root, 's']

   while (array.length > 1){
      const node = array.shift()
      if (node === 's'){
         array.push(node)
         depth++
      } else {
         if (node.left){
            array.push(node.left)
         }
         if (node.right) {
            array.push(node.right)
         }
      }
   }
    return depth+1;
};

// Recursive Approach
var maxDepth = function(root) {
   return visit(root, 0)
};

function visit(node, depth){
   if (node === null){
      return depth
   }
   depth++
   return Math.max(visit(node.left, depth), visit(node.right, depth))
}
```

MIN DEPTH OF THE TREE

```
// Time complexity O(N)
// Space complexity O(N) if unbalanced else O(log(n)) if balanced
var minDepth = function(root) {
   return visit(root, 0)
};

function visit(node, depth){
   if (node === null){
      return depth
   }
   if (node.left === null){
      return visit(node.right, depth) + 1
   }
   if (node.right === null){
      return visit(node.left, depth) + 1
   }

   depth++
   return Math.min(visit(node.left, depth), visit(node.right, depth))
}

// BFS Iterartion
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
var minDepth = function(root) {
   if (!root){
      return 0
   }
   let queue = [{node: root, height: 1}]
   while (queue.length) {
      let currentNode = queue.shift();
      if (!currentNode.node.left && !currentNode.node.right){
         return currentNode.height;
      }
      if (currentNode.node.left){
         queue.push({node :currentNode.node.left, height: currentNode.height + 1})
      }
      if (currentNode.node.right){
         queue.push({node :currentNode.node.right, height: currentNode.height + 1})
      }
   }
};
```

PYRAMID

```
// --- Directions
// Write a function that accepts a positive number N.
// The function should console log a pyramid shape
// with N levels using the # character.  Make sure the
// pyramid has spaces on both the left *and* right hand sides
//   pyramid(1)
//       '#'
//   pyramid(3)
//       '  #  '
//       ' ### '
//       '#####'

// Iterative approach
// function pyramid(n) {
//     const midpoint = Math.floor((2*n -1)/2)
//     for (let row = 0; row < n; row++){
//         let stair='';
//         for (let column = 0; column < 2*n - 1; column++){
//             if(midpoint - row <= column && midpoint + row >= column) {
//                 stair += '#';
//             } else {
//                 stair += ' ';
//             }
//         }
//         console.log(stair)
//     }
// }

// Recursive approach
function pyramid(n, row = 0, stair = '') {
    if(row === n){
        return;
    }

    if ((2*n - 1) === stair.length){
        console.log(stair);
        return pyramid(n, row+1)
    }
    const midpoint = Math.floor((2*n - 1)/2);
    if ((midpoint-row) <= stair.length && (midpoint+row) >= stair.length){
        stair += '#';
    } else {
        stair += ' ';
    }
    return pyramid(n, row, stair);
}
module.exports = pyramid;
```

QUEUE FROM STACKS

```
const Stack = require('./stack');

class Queue {
   constructor() {
      this.stack1 = new Stack();
      this.stack2 = new Stack();
   }

   add(record) {
      while (this.stack1.peek()){
         this.stack2.push(this.stack1.pop());
      }
      this.stack1.push(record)
      while (this.stack2.peek()){
         this.stack1.push(this.stack2.pop());
      }
   }

   remove() {
      return this.stack1.pop();
   }

   peek() {
      return this.stack1.peek();
   }
}

module.exports = Queue;
```

MERGE SORT

```
function mergeSort(arr) {
   if (arr.length === 1){
      return arr;
   }

   const center = Math.floor(arr.length/2);
   const left = arr.slice(0, center);
   const right = arr.slice(center);
   return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
   let results = []
   while (left.length && right.length){
      const leftFirstElement = left[0]
      const rightFirstElement = right[0]
      if (leftFirstElement < rightFirstElement){
         results.push(left.shift())
      } else {
         results.push(right.shift())
      }
   }
   // if (left.length) {
   //     results.push(...left)
   // }
   // if (right.length) {
   //     results.push(...right)
   // }
   results = [...results, ...left, ...right]
   return results
}
```

VALIDATE BST

```
// --- Directions
// Given a node, validate the binary search tree,
// ensuring that every node's left hand child is
// less than the parent node's value, and that
// every node's right hand child is greater than
// the parent

function validate(node, min = null, max = null) {
  if (max !== null && node.data > max) {
    return false;
  }

  if (min !== null && node.data < min) {
    return false;
  }

  if (node.left && !validate(node.left, min, node.data)){
    return false
  }

  if (node.right && !validate(node.right, node.data, max)){
    return false
  }

  return true
}
```