**JSPM Narhe Technical Campus**

**MCA Department**

**MCA-I(Sem-I)**

**(Data Structure and Algorithm)**

# Write a JS program for Graph implementation and DFS graph traversals

**ANS:**

// create a graph class class Graph {

// defining vertex array and

// adjacent list constructor(noOfVertices)

{

this.noOfVertices = noOfVertices; this.AdjList = new Map();

}

// functions to be implemented

// add vertex to the graph addVertex(v)

{

// initialize the adjacent list with a

// null array this.AdjList.set(v, []);

}

// add edge to the graph addEdge(v, w)

{

// get the list for vertex v and put the

// vertex w denoting edge between v and w

this.AdjList.get(v).push(w);

// Since graph is undirected,

// add an edge from w to v also this.AdjList.get(w).push(v);

}

// Prints the vertex and adjacency list printGraph()

{

// get all the vertices

var get\_keys = this.AdjList.keys();

// iterate over the vertices for (var i of get\_keys)

{

// great the corresponding adjacency list

// for the vertex

var get\_values = this.AdjList.get(i); var conc = "";

// iterate over the adjacency list

// concatenate the values into a string for (var j of get\_values)

conc += j + " ";

// print the vertex and its adjacency list console.log(i + " -> " + conc);

}

}

// Main DFS method dfs(startingNode)

{

var visited = {}; this.DFSUtil(startingNode, visited);

}

// Recursive function which process and explore

// all the adjacent vertex of the vertex with which it is called DFSUtil(vert, visited)

{

visited[vert] = true; console.log(vert);

var get\_neighbours = this.AdjList.get(vert);

for (var i in get\_neighbours) {

var get\_elem = get\_neighbours[i]; if (!visited[get\_elem])

this.DFSUtil(get\_elem, visited);

}

}

}

// Using the above implemented graph class var g = new Graph(6);

var vertices = [ 'A', 'B', 'C', 'D', 'E', 'F' ];

// adding vertices

for (var i = 0; i < vertices.length; i++) { g.addVertex(vertices[i]);

}

// adding edges g.addEdge('A', 'B');

g.addEdge('A', 'D');

g.addEdge('A', 'E');

g.addEdge('B', 'C');

g.addEdge('D', 'E');

g.addEdge('E', 'F');

g.addEdge('E', 'C');

g.addEdge('C', 'F');

// prints all vertex and

// its adjacency list

// A -> B D E

// B -> A C

// C -> B E F

// D -> A E

// E -> A D F C

// F -> E C

g.printGraph();

// prints// DFS// A B C E D F console.log("DFS");

g.dfs('A');

# Write a JS program for implementation of Hashing.

class HashTable{ constructor(size=50){ this.buckets = new Array(size) this.size = size

}

hash(key){

return key.toString().length % this.size;

}

// Insert data setItem(key,value){

let index = this.hash(key);

if(!this.buckets[index]){ this.buckets[index] = [];

}

this.buckets[index].push([key,value]) return index

}

// Search data getItem(key){

let index = this.hash(key); if(!this.buckets[index])return null

for(let bucket of this.buckets[index]){

// key

if(bucket [0] === key){

// value

return bucket [1]

}

}

}

}

const hashTable = new HashTable();

// Insert data to the hash table hashTable.setItem("bk101","Data structures algorithms"); hashTable.setItem("bk108","Data analytics");

hashTable.setItem("bk200","Cyber security"); hashTable.setItem("bk259","Business Intelligence"); hashTable.setItem("bk330","S/W Development");

// Search data from the hash table hashTable.getItem("bk101"); console.log(hashTable.getItem("bk101"));

# Write a JS program Rain water Trapping (Practical based on Brute Force technique)

// Javascript implementation of the approach

// Function to return the maximum

// water that can be stored function maxWater(arr, n)

{

// To store the maximum water

// that can be stored let res = 0;

// For every element of the array

// except first and last element for(let i = 1; i < n - 1; i++)

{

// Find maximum element on its left let left = arr[i];

for(let j = 0; j < i; j++)

{

left = Math.max(left, arr[j]);

}

// Find maximum element on its right let right = arr[i];

for(let j = i + 1; j < n; j++)

{

right = Math.max(right, arr[j]);

}

// Update maximum water value res += Math.min(left, right) - arr[i];

}

return res;

}

let arr = [ 0, 1, 0, 2, 1, 0,

1, 3, 2, 1, 2, 1 ];

let n = arr.length; console.log(maxWater(arr,n));

# Write a JS program Jump Game.( Practical based on Greedy Algorithm-)

var canJump = function(nums) { var len = nums.length;

if(len < 1)

return false; if(len == 1)

return true; var max = 0;

for (var i = 0; i < len; i++) { if (i > max) return false;

max = Math.max(max, i + nums[i]);

}

return true;

};

var nums1=[2,3,1,1,4]; //true var nums2 =[3,2,1,0,4]; //false console.log(canJump(nums1)); console.log(canJump(nums2));

# Write a JS program for Binary Search(Practical based on Divide and Conquer Technique-)

const binarySearch = (list, item) => { let low = 0

let high = list.length - 1

while (low <= high) {

const mid = Math.floor((low + high) / 2) const guess = list[mid]

if (guess === item) { return mid

}

if (guess > item) { high = mid - 1

} else {

low = mid + 1

}

}

return null //if not found

}

console.log(binarySearch([1, 2, 3, 4, 5], 1)) //0

console.log(binarySearch([1, 2, 3, 4, 5], 5)) //4

console.log(binarySearch([1, 2, 3, 4, 5], 6)) //null

# Write a JS program for Longest Common Subsequence LCS (Implementation of Dynamic Programming- )

function longestCommonSequenceLength(str1, str2) {

var matrix = Array(str1.length + 1).fill(Array(str2.length + 1).fill(0)), rowLength = str1.length + 1,

colLength = str2.length + 1, max = 0;

for (var row = 1; row < rowLength; row++) { for (var col = 1; col < colLength; col++) {

var str1Char = str1.charAt(row - 1), str2Char = str2.charAt(col - 1);

if (str1Char == str2Char) {

matrix[row][col] = matrix[row - 1][col - 1] + 1; max = Math.max(matrix[row][col], max);

}

}

}

return max;

}

//var lcs= longestCommonSequenceLength('abcd', 'bc');

var lcs= longestCommonSequenceLength('longest', 'stone'); console.log(lcs);

# Write a JS program for finding out Power Set (Practical based on backtracking)

function printPowerSet(set, set\_size)

{

/\*

\* set\_size of power set of a set with set\_size n is (2\*\*n -1)

\*/

var pow\_set\_size = parseInt(Math.pow(2, set\_size)); var counter, j;

/\*

\* Run from counter 000..0 to 111..1

\*/

for (counter = 0; counter < pow\_set\_size; counter++)

{

for (j = 0; j < set\_size; j++)

{

/\*

\* Check if jth bit in the counter is set If set then prvar jth element from set

\*/

if ((counter & (1 << j)) > 0) console.log(set[j]);

}

console.log(" ");

}

}

// Driver program to test printPowerSet var set = [ 'a', 'b', 'c' ]; printPowerSet(set, 3);

# Write a JS program BST

// Node class class Node

{

constructor(data)

{

this.data = data; this.left = null; this.right = null;

}

}

// Binary Search tree class class BinarySearchTree

{

constructor()

{

// root of a binary seach tree this.root = null;

}

// function to be implemented

// insert(data)

// remove(data)

// Helper function

// findMinNode()

// getRootNode()

// inorder(node)

// preorder(node)

// postorder(node)

// search(node, data)

// helper method which creates a new node to

// be inserted and calls insertNode insert(data)

{

// Creating a node and initailising

// with data

var newNode = new Node(data);

// root is null then node will

// be added to the tree and made root. if(this.root === null)

this.root = newNode; else

// find the correct position in the

// tree and add the node this.insertNode(this.root, newNode);

}

// Method to insert a node in a tree

// it moves over the tree to find the location

// to insert a node with a given data insertNode(node, newNode)

{

// if the data is less than the node

// data move left of the tree if(newNode.data < node.data)

{

// if left is null insert node here if(node.left === null)

node.left = newNode; else

// if left is not null recur until

// null is found this.insertNode(node.left, newNode);

}

// if the data is more than the node

// data move right of the tree else

{

// if right is null insert node here if(node.right === null)

node.right = newNode; else

// if right is not null recur until

// null is found this.insertNode(node.right,newNode);

}

}

// helper method that calls the

// removeNode with a given data remove(data)

{

// root is re-initialized with

// root of a modified tree.

this.root = this.removeNode(this.root, data);

}

// Method to remove node with a

// given data

// it recur over the tree to find the

// data and removes it removeNode(node, key)

{

// if the root is null then tree is

// empty if(node === null)

return null;

// if data to be delete is less than

// roots data then move to left subtree else if(key < node.data)

{

node.left = this.removeNode(node.left, key); return node;

}

// if data to be delete is greater than

// roots data then move to right subtree else if(key > node.data)

{

node.right = this.removeNode(node.right, key); return node;

}

// if data is similar to the root's data

// then delete this node else

{

// deleting node with no children if(node.left === null && node.right === null)

{

node = null; return node;

}

// deleting node with one children if(node.left === null)

{

node = node.right; return node;

}

else if(node.right === null)

{

node = node.left; return node;

}

// Deleting node with two children

// minumum node of the rigt subtree

// is stored in aux

var aux = this.findMinNode(node.right); node.data = aux.data;

node.right = this.removeNode(node.right, aux.data); return node;

}

}

// finds the minimum node in tree

// searching starts from given node findMinNode(node)

{

// if left of a node is null

// then it must be minimum node if(node.left === null)

return node; else

return this.findMinNode(node.left);

}

// Performs inorder traversal of a tree inorder(node)

{

if(node !== null)

{

this.inorder(node.left); console.log(node.data); this.inorder(node.right);

}

}

preorder(node)

{

if(node !== null)

{

console.log(node.data); this.preorder(node.left); this.preorder(node.right);

}

}

// Performs postorder traversal of a tree postorder(node)

{

if(node !== null)

{

this.postorder(node.left); this.postorder(node.right); console.log(node.data);

}

}

// returns root of the tree getRootNode()

{

return this.root;

}

// search for a node with given data search(node, data)

{

// if trees is empty return null if(node === null)

return null;

// if data is less than node's data

// move left

else if(data < node.data)

return this.search(node.left, data);

// if data is less than node's data

// move left

else if(data > node.data)

return this.search(node.right, data);

// if data is equal to the node data

// return node else

return node;

}

};

// create an object for the BinarySearchTree var BST = new BinarySearchTree();

// Inserting nodes to the BinarySearchTree BST.insert(15);

BST.insert(25); BST.insert(10);

BST.insert(7); BST.insert(22); BST.insert(17); BST.insert(13); BST.insert(5); BST.insert(9); BST.insert(27);

|  |  |
| --- | --- |
| //  // | 15  / \ |
| // | 10 25 |
| // | / \ / \ |
| // | 7 13 22 27 |
| // | / \ / |
| // | 5 9 17 |

var root = BST.getRootNode(); console.log("Printing Inorder Traversal");

// prints 5 7 9 10 13 15 17 22 25 27 BST.inorder(root);

// Removing node with no children BST.remove(5);

|  |  |
| --- | --- |
| //  // | 15  / \ |
| // | 10 25 |
| // | / \ / \ |
| // | 7 13 22 27 |
| // | \ / |
| // | 9 17 |

var root = BST.getRootNode();

console.log("Printing Inorder Traversal after removing 5");

// prints 7 9 10 13 15 17 22 25 27 BST.inorder(root);

// Removing node with one child BST.remove(7);

// 15

// / \

// 10 25

// / \ / \

// 9 13 22 27

// /

// 17

var root = BST.getRootNode();

console.log("Printing Inorder Traversal after removing 7");

// prints 9 10 13 15 17 22 25 27 BST.inorder(root);

// Removing node with two children BST.remove(15);

// 17

// / \

// 10 25

// / \ / \

// 9 13 22 27

var root = BST.getRootNode();

console.log("Printing Inorder Traversal after removing 15"); console.log("inorder traversal");

// prints 9 10 13 17 22 25 27 BST.inorder(root);

console.log("postorder traversal"); BST.postorder(root); console.log("preorder traversal"); BST.preorder(root);

# Write a JS program Practical based on backtracking- N Queen’s

**problems**

/\*\*

* @param {number} n
* @return {string[][]}

\*/

var solveNQueens = function(n) { var res = [];

if (n === 1 || n >= 4) dfs(res, [], n, 0); return res;

};

var dfs = function (res, points, n, index) { for (var i = index; i < n; i++) {

if (points.length !== i) return; for (var j = 0; j < n; j++) {

if (isValid(points, [i, j])) {

points.push([i, j]);

dfs(res, points, n, i + 1);

if (points.length === n) res.push(buildRes(points)); points.pop();

}

}

}

};

var buildRes = function (points) { var res = [];

var n = points.length; for (var i = 0; i < n; i++) { res[i] = '';

for (var j = 0; j < n; j++) {

res[i] += (points[i][1] === j ? 'Q' : '.');

}

}

return res;

};

var isValid = function (oldPoints, newPoint) { var len = oldPoints.length;

for (var i = 0; i < len; i++) {

if (oldPoints[i][0] === newPoint[0] || oldPoints[i][1] === newPoint[1]) return false;

if (Math.abs((oldPoints[i][0] - newPoint[0]) / (oldPoints[i][1] - newPoint[1])) === 1) return false;

}

return true;

};

var r =[] ;

r =solveNQueens(4); console.log(r);

# Write a JS program for findingUnique Paths.

function uniquePaths(m, n) { if(m==0 || n==0) return 0; if(m==1 || n==1) return 1; var dp = [];

for(var i=0; i<m; i++){ var temp = []; for(var j=0; j<n; j++){

temp.push(0);

}

dp.push(temp);

}

//left column

for(var i=0; i<m; i++){ dp[i][0] = 1;

}

//top row

for(var j=0; j<n; j++){ dp[0][j] = 1;

}

//fill up the dp table for(var i=1; i<m; i++){

for(var j=1; j<n; j++){

dp[i][j] = dp[i-1][j] + dp[i][j-1];

}

}

return dp[m-1][n-1];

}

var res=[]; res=uniquePaths(3,3); console.log(res);