**1)** **Layers of TCP/IP Model**

**i)** Application layer

**ii)** Transport layer

**iii)** Internet layer

**iv)** Link layer

**APPLICATION LAYER:**

This layer represents the programs that interact with the network, such as web browsers, email clients, and file transfer utilities. It is responsible for providing services that enable users to access the network and exchange the data with other devices

**TRANSPORT LAYER:**

this layer provides end-to-end data transfer between devices. It is responsible for establishing connections, managing the flow of data, and ensuring data integrity. two of the most common protocols used in this layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

**INTERNET LAYER:**

This layer is responsible for routing data across the internet. It uses Internet Protocol (IP) to address packets and ensure they are delivered to correct destination. It also handles fragmentation and reassembly of packets as they travel across networks with different Maximum Transmission Units (MTUs).

**LINK LAYER:**

This layer is responsible for transmitting data between devices on the same physical network. It includes protocols such as Ethernet and Wi-Fi, which provide the physical means of data transfer.

---------------------------------------------------------------------------------------------------

**2)** **Creating Linked List:**

class LinkedListNode{

constructor(data,next){

this.data = data;

this.next = next;

}

}

function addNodeattheEnd(head,data) {

const newNode = new LinkedListNode (data,null);

let curr = head;

while (curr.next !== null) {

curr = curr.next;

}

curr.next = newNode;

}

function printLinkedList (head){

let curr = head;

while (curr!==null) {

console.log (curr.data);

curr = curr.next;

}

}

const head = new LinkedListNode (1,null);

addNodeattheEnd(head,2);

addNodeattheEnd(head,3);

addNodeattheEnd(head,4);

addNodeattheEnd(head,5);

printLinkedList(head)

**Reverse Linked List using Recursion:**

function revRec (head,prev){

if (head === null){

return prev;

}

let next = head.next;

head.next = prev;

return revRec (next,head);

}

const newhead = revRec (head,null);

printLinkedList(newhead);

----------------------------------------------------------------------------------------------------

**3)** //**to create a tree**

class BinaryTree{

constructor(data,left,right){

this.data = data;

this.left = left;

this.right = right;

}

}

const root = new BinaryTree (9,null,null);

node2 = new BinaryTree (3,null,null);

node3 = new BinaryTree (15,null,null);

root.left = node2;

root.right = node3;

node4 = new BinaryTree (20,null,null);

node5 = new BinaryTree (7,null,null);

node2.left = node4;

node2.right = node5;

console.log (root);

// after this our tree will look like

// 9

// / \

// 3 15

// / \

// 20 7

// **inorder traversal should be (left,root,right)**

function inorder (root){

//base condition

if (root == null) return;

//visit left child

inorder(root.left);

//visit root node

console.log (root.data);

//visit right child

inorder (root.right);

}

inorder (root); // 20,3,7.9,15

//**to create a tree**

class BinaryTree{

constructor(data,left,right){

this.data = data;

this.left = left;

this.right = right;

}

}

const root = new BinaryTree (9,null,null);

node2 = new BinaryTree (15,null,null);

node3 = new BinaryTree (7,null,null);

root.left = node2;

root.right = node3;

node4 = new BinaryTree (20,null,null);

node5 = new BinaryTree (3,null,null);

node2.left = node4;

node2.right = node5;

console.log (root);

// after this our tree will look like

// 9

// / \

// 15 7

// / \

// 20 3

//**postorder traversal should be (left,right,root)**

function postorder (root){

//base condition

if (root == null) return;

//visit left child

postorder(root.left);

//visit right child

postorder (root.right);

//visit root node

console.log (root.data);

}

postorder (root);// 20,3,15,7,9

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

4) Breadth First Search (BFS) and Depth First Search (DFS) are two common algorithms used to traverse graphs or trees. While both algorithms are used to traverse a graph, they differ in the way they traverse and the order in which they visit nodes.

**BFS:**

BFS traverse the graph level by level, visiting all the nodes at each level before moving on to the next level. It starts at the root node and then visits all the nodes at distance one before visiting the nodes at distance two, and so on. BFS is typically implemented using a queue data structure to keep track of the nodes to visit. The time complexity of BFS is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

**DFS:**

DFS explores as far as possible along each branch before backtracing. It starts at the root node and visits one of its neighbors, then visits one of the neighbors of that neighbor, and so on until it reaches a dead end. It then backtracks to the last node with an unvisited neighbor and continues the process until all nodes have been visited. DFS is typically implemented using a stack data structure to keep track of the nodes to visit. The time complexity of DFS is also **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**5)**

for(let i =0; i<n ;i++) {

j =1;

while(j<n){

console.log(i)

j= j\*2

}

}

Time Complexity: **O (n log n)**

The outer loop runs n times, where n is the value of the input parameter. The inner loop multiplies j by 2 in each iteration until it reaches a value greater than or equal to n. This means that the number of iterations in the inner loop is determined by the number of times j can be multiplied by 2 before it becomes greater than or equal to n (i.e) log2(n)

---------------------------------------------------------------------------------------------------

**6)**

i = 1;

while(i2<n){

i+=1

}

Time Complexity: **O(sqrt(n))**

The loop continues to execute as long as the condition i2 < n is true. In other words, the loop runs until i becomes greater than or equal to the square root of n.