1. **Write a program to create singly linked list**

**class Node {**

**constructor(data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**class LinkedList {**

**constructor() {**

**this.head = null;**

**}**

**append(data) {**

**const new\_node = new Node(data);**

**if (!this.head) {**

**this.head = new\_node;**

**return;**

**}**

**let current\_node = this.head;**

**while (current\_node.next) {**

**current\_node = current\_node.next;**

**}**

**current\_node.next = new\_node;**

**}**

**display() {**

**let current\_node = this.head;**

**while (current\_node) {**

**console.log(current\_node.data);**

**current\_node = current\_node.next;**

**}**

**}**

**}**

**2. Write a Program to insert an element at given position in singly linked list**

**class Node {**

**constructor(data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**class LinkedList {**

**constructor() {**

**this.head = null;**

**this.size = 0;**

**}**

**add(data) {**

**const node = new Node(data);**

**if (!this.head) {**

**this.head = node;**

**} else {**

**let current = this.head;**

**while (current.next) {**

**current = current.next;**

**}**

**current.next = node;**

**}**

**this.size++;**

**}**

**insertAt(data, position) {**

**if (position < 0 || position > this.size) {**

**return false;**

**} else {**

**const node = new Node(data);**

**let current = this.head;**

**let previous = null;**

**let index = 0;**

**if (position === 0) {**

**node.next = this.head;**

**this.head = node;**

**} else {**

**while (index < position) {**

**previous = current;**

**current = current.next;**

**index++;**

**}**

**previous.next = node;**

**node.next = current;**

**}**

**this.size++;**

**}**

**}**

**print() {**

**let current = this.head;**

**let result = "";**

**while (current) {**

**result += current.data + " ";**

**current = current.next;**

**}**

**console.log(result.trim());**

**}**

**}**

**const linkedList = new LinkedList();**

**linkedList.add(1);**

**linkedList.add(2);**

**linkedList.add(3);**

**linkedList.print(); // Output: "1 2 3"**

**linkedList.insertAt(4, 2);**

**linkedList.print();**

**// Output: "1 2 4 3"**

**3.Write a Program to insert an element at the end of singly linked list**

**class Node {**

**constructor(data) {**

**this.data = data;**

**this.next = null;**

**}**

**}**

**class SinglyLinkedList {**

**constructor() {**

**this.head = null;**

**this.tail = null;**

**this.length = 0;**

**}**

**insertAtEnd(data) {**

**const newNode = new Node(data);**

**if (!this.head) {**

**this.head = newNode;**

**this.tail = newNode;**

**} else {**

**this.tail.next = newNode;**

**this.tail = newNode;**

**}**

**this.length++;**

**}**

**}**

**4. Write a Program to insert an element in the beginning of singly linked list**

class Node {

constructor(data) {

this.data = data;

this.next = null;

}

}

class SinglyLinkedList {

constructor() {

this.head = null;

this.length = 0;

}

insertAtBeginning(data) {

const newNode = new Node(data);

if (!this.head) {

this.head = newNode;

} else {

newNode.next = this.head;

this.head = newNode;

}

this.length++;

}

display() {

let current = this.head;

let list = "";

while (current) {

list += current.data + " ";

current = current.next;

}

console.log(list);

}

}

const linkedList = new SinglyLinkedList();

linkedList.insertAtBeginning(5);

linkedList.insertAtBeginning(10);

linkedList.insertAtBeginning(15);

linkedList.display();

// Output: 15 10 5

**5. Write a program to create doubly linked list.**

class Node {

constructor(data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

// Add a new node to the end of the list

append(data) {

const newNode = new Node(data);

if (!this.head) {

this.head = newNode;

this.tail = newNode;

} else {

newNode.prev = this.tail;

this.tail.next = newNode;

this.tail = newNode;

}

this.length++;

}

// Insert a new node at a specific index

insertAt(data, index) {

if (index < 0 || index > this.length) {

throw new Error("Index out of bounds");

}

const newNode = new Node(data);

if (!this.head || index === 0) {

newNode.next = this.head;

this.head = newNode;

} else if (index === this.length) {

newNode.prev = this.tail;

this.tail.next = newNode;

this.tail = newNode;

} else {

let currentNode = this.head;

let currentIndex = 0;

while (currentIndex < index) {

currentNode = currentNode.next;

currentIndex++;

}

newNode.prev = currentNode.prev;

newNode.next = currentNode;

currentNode.prev.next = newNode;

currentNode.prev = newNode;

}

this.length++;

}

// Remove a node at a specific index

removeAt(index) {

if (index < 0 || index >= this.length) {

throw new Error("Index out of bounds");

}

let removedNode;

if (index === 0) {

removedNode = this.head;

if (this.length === 1) {

this.head = null;

this.tail = null;

} else {

this.head = removedNode.next;

this.head.prev = null;

}

} else if (index === this.length - 1) {

removedNode = this.tail;

this.tail = removedNode.prev;

this.tail.next = null;

} else {

let currentNode = this.head;

let currentIndex = 0;

while (currentIndex < index) {

currentNode = currentNode.next;

currentIndex++;

}

removedNode = currentNode;

currentNode.prev.next = currentNode.next;

currentNode.next.prev = currentNode.prev;

}

this.length--;

return removedNode.data;

}

// Traverse the list and execute a callback function on each node

traverse(callback) {

let currentNode = this.head;

while (currentNode) {

callback(currentNode);

currentNode = currentNode.next;

}

}

// Reverse the order of the list

reverse() {

let currentNode = this.head;

let prevNode = null;

while (currentNode) {

const nextNode = currentNode.next;

currentNode.next = prevNode;

currentNode.prev = nextNode;

prevNode = currentNode;

currentNode = nextNode;

}

this.tail = this.head;

this.head = prevNode;

}

}

**6. Write a Program to insert an element at given position in doubly linked list**

**class Node {**

**constructor(data) {**

**this.data = data;**

**this.prev = null;**

**this.next = null;**

**}**

**}**

**class DoublyLinkedList {**

**constructor() {**

**this.head = null;**

**this.tail = null;**

**this.length = 0;**

**}**

**insertAt(position, data) {**

**if (position < 0 || position > this.length) {**

**return false;**

**}**

**const newNode = new Node(data);**

**if (this.length === 0) {**

**this.head = newNode;**

**this.tail = newNode;**

**} else if (position === 0) {**

**newNode.next = this.head;**

**this.head.prev = newNode;**

**this.head = newNode;**

**} else if (position === this.length) {**

**newNode.prev = this.tail;**

**this.tail.next = newNode;**

**this.tail = newNode;**

**} else {**

**let current = this.head;**

**let previous = null;**

**let index = 0;**

**while (index < position) {**

**previous = current;**

**current = current.next;**

**index++;**

**}**

**newNode.prev = previous;**

**newNode.next = current;**

**previous.next = newNode;**

**current.prev = newNode;**

**}**

**this.length++;**

**return true;**

**}**

**}**

**7. Write a Program to insert an element at the end of doubly linked list**

class Node {

constructor(data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.size = 0;

}

// method to add a node to the end of the list

addNode(data) {

const newNode = new Node(data);

if (!this.head) {

this.head = newNode;

this.tail = newNode;

} else {

newNode.prev = this.tail;

this.tail.next = newNode;

this.tail = newNode;

}

this.size++;

}

// method to print the list

printList() {

let current = this.head;

let output = '';

while (current) {

output += current.data + ' <-> ';

current = current.next;

}

console.log(output);

}

}

// example usage

const list = new DoublyLinkedList();

list.addNode(1);

list.addNode(2);

list.addNode(3);

list.printList(); // output: 1 <-> 2 <-> 3 <->

list.addNode(4);

list.printList();

// output: 1 <-> 2 <-> 3 <-> 4 <->

**8. Write a Program to insert an element in the beginning of doubly linked list**

class Node {

constructor(value) {

this.value = value;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

// Method to insert a node at the beginning of the list

insertAtBeginning(value) {

const newNode = new Node(value);

if (this.length === 0) {

this.head = newNode;

this.tail = newNode;

} else {

this.head.prev = newNode;

newNode.next = this.head;

this.head = newNode;

}

this.length++;

}

// Method to print the linked list

printList() {

let current = this.head;

let result = "";

while (current !== null) {

result += current.value + " ";

current = current.next;

}

console.log(result);

}

}

// Example usage

const myList = new DoublyLinkedList();

myList.insertAtBeginning(10);

myList.insertAtBeginning(20);

myList.insertAtBeginning(30);

myList.insertAtBeginning(40);

myList.printList();

// Output: 40 30 20 10

**9. Write a program to reverse the nodes of a linked list.**

function reverseLinkedList(head) {

let prevNode = null;

let currentNode = head;

while (currentNode !== null) {

let nextNode = currentNode.next;

currentNode.next = prevNode;

prevNode = currentNode;

currentNode = nextNode;

}

return prevNode; // the new head of the list

}

**10. Write a program to find the count of nodes of singly linked list**

function countNodes(head) {

let count = 0;

let current = head;

while (current !== null) {

count++;

current = current.next;

}

return count;

}

**11. Write a program to find sum of values in linked list.**

// Define the Node class

class Node {

constructor(value) {

this.value = value;

this.next = null;

}

}

// Define the LinkedList class

class LinkedList {

constructor() {

this.head = null;

this.tail = null;

}

// Add a new node to the end of the list

addNode(value) {

const newNode = new Node(value);

if (this.head === null) {

this.head = newNode;

this.tail = newNode;

} else {

this.tail.next = newNode;

this.tail = newNode;

}

}

// Calculate the sum of values in the linked list

getSum() {

let sum = 0;

let current = this.head;

while (current !== null) {

sum += current.value;

current = current.next;

}

return sum;

}

}

// Test the program

const list = new LinkedList();

list.addNode(2);

list.addNode(3);

list.addNode(5);

list.addNode(7);

console.log(list.getSum());

// Output: 17

**12. Write a program to implement stack using Array with PUSH, POP operations**

// Initialize an empty array to represent the stack

let stack = [];

// Push an element onto the top of the stack

function push(element) {

stack.push(element);

}

// Remove and return the element at the top of the stack

function pop() {

return stack.pop();

}

**13. Write a Program to reverse a string using stack.**

function reverseString(str) {

// Create an empty stack

const stack = [];

// Push each character of the string onto the stack

for (let i = 0; i < str.length; i++) {

stack.push(str.charAt(i));

}

// Pop each character off the stack to create the reversed string

let reversedStr = '';

while (stack.length > 0) {

reversedStr += stack.pop();

}

return reversedStr;

}

// Example usage

console.log(reverseString('Hello, world!'));

// Output: !dlrow ,olleH

**14. Write a Program to check for balanced parentheses by using Stacks**

function isBalanced(str) {

const stack = [];

const openBrackets = ['(', '{', '['];

const closeBrackets = [')', '}', ']'];

for (let i = 0; i < str.length; i++) {

if (openBrackets.includes(str[i])) {

stack.push(str[i]);

} else if (closeBrackets.includes(str[i])) {

const matchingOpenBracket = openBrackets[closeBrackets.indexOf(str[i])];

if (stack.length === 0 || stack.pop() !== matchingOpenBracket) {

return false;

}

}

}

return stack.length === 0;

}

**15. Write a Program to Implement Stack using Linked List with PUSH, POP operations**

class Node {

constructor(data) {

this.data = data;

this.next = null;

}

}

class Stack {

constructor() {

this.top = null;

this.size = 0;

}

push(data) {

const newNode = new Node(data);

newNode.next = this.top;

this.top = newNode;

this.size++;

}

pop() {

if (this.isEmpty()) {

return null;

}

const poppedNode = this.top;

this.top = this.top.next;

this.size--;

return poppedNode.data;

}

isEmpty() {

return this.size === 0;

}

peek() {

if (this.isEmpty()) {

return null;

}

return this.top.data;

}

print() {

let currentNode = this.top;

while (currentNode !== null) {

console.log(currentNode.data);

currentNode = currentNode.next;

}

}

}

**16. Write a Program to perform Enqueue and Dequeue operations on linear Queue**

class Queue {

constructor() {

this.items = [];

}

// enqueue function adds an element to the end of the queue

enqueue(element) {

this.items.push(element);

}

// dequeue function removes an element from the front of the queue

dequeue() {

if (this.isEmpty()) {

return "Underflow";

}

return this.items.shift();

}

// front function returns the element at the front of the queue without removing it

front() {

if (this.isEmpty()) {

return "No elements in Queue";

}

return this.items[0];

}

// isEmpty function returns true if the queue is empty, otherwise false

isEmpty() {

return this.items.length == 0;

}

// printQueue function prints all the elements of the queue

printQueue() {

let str = "";

for (let i = 0; i < this.items.length; i++) {

str += this.items[i] + " ";

}

return str;

}

}

// Example usage of the Queue class

let queue = new Queue();

// Adding elements to the queue

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

// Printing the queue elements

console.log(queue.printQueue());

// Removing an element from the queue

queue.dequeue();

// Printing the queue elements after removing an element

console.log(queue.printQueue());

// Printing the element at the front of the queue

console.log(queue.front());

**17. Write a Program to create Circular Queue.**

class CircularQueue {

constructor(maxSize) {

this.maxSize = maxSize;

this.queue = new Array(maxSize);

this.front = -1;

this.rear = -1;

}

enqueue(data) {

if (this.isFull()) {

console.log("Queue is full.");

return false;

}

if (this.front == -1) {

this.front = 0;

}

this.rear = (this.rear + 1) % this.maxSize;

this.queue[this.rear] = data;

console.log(`${data} added to queue.`);

return true;

}

dequeue() {

if (this.isEmpty()) {

console.log("Queue is empty.");

return false;

}

let data = this.queue[this.front];

if (this.front == this.rear) {

this.front = -1;

this.rear = -1;

} else {

this.front = (this.front + 1) % this.maxSize;

}

console.log(`${data} removed from queue.`);

return data;

}

isFull() {

return (

(this.rear + 1) % this.maxSize == this.front &&

this.front != -1

);

}

isEmpty() {

return this.front == -1;

}

display() {

if (this.isEmpty()) {

console.log("Queue is empty.");

return;

}

let i = this.front;

while (i != this.rear) {

console.log(this.queue[i]);

i = (i + 1) % this.maxSize;

}

console.log(this.queue[this.rear]);

}

}

// Example usage:

let queue = new CircularQueue(5);

queue.enqueue(1);

queue.enqueue(2);

queue.enqueue(3);

queue.enqueue(4);

queue.enqueue(5);

queue.enqueue(6); // queue is full, cannot add

queue.display(); // prints 1 2 3 4 5

queue.dequeue(); // prints 1

queue.display(); // prints 2 3 4 5

queue.enqueue(6); // can add now, prints "6 added to queue."

queue.display(); // prints 2 3 4 5 6

**18. Write a Program to reverse stack using queue.**

// Stack class

class Stack {

constructor() {

this.items = [];

}

push(element) {

this.items.push(element);

}

pop() {

if (this.items.length == 0)

return "Underflow";

return this.items.pop();

}

isEmpty() {

return this.items.length == 0;

}

printStack() {

var str = "";

for (var i = 0; i < this.items.length; i++)

str += this.items[i] + " ";

return str;

}

}

// Queue class

class Queue {

constructor() {

this.items = [];

}

enqueue(element) {

this.items.push(element);

}

dequeue() {

if (this.isEmpty())

return "Underflow";

return this.items.shift();

}

isEmpty() {

return this.items.length == 0;

}

printQueue() {

var str = "";

for (var i = 0; i < this.items.length; i++)

str += this.items[i] + " ";

return str;

}

}

// ReverseStack function

function reverseStack(stack) {

var queue = new Queue();

while (!stack.isEmpty()) {

queue.enqueue(stack.pop());

}

while (!queue.isEmpty()) {

stack.push(queue.dequeue());

}

return stack;

}

// Example usage

var stack = new Stack();

stack.push(1);

stack.push(2);

stack.push(3);

stack.push(4);

stack.push(5);

console.log("Original Stack: " + stack.printStack());

reverseStack(stack);

console.log("Reversed Stack: " + stack.printStack());

**19. Write a program to create a binary search tree**

**class Node {**

**constructor(value) {**

**this.value = value;**

**this.left = null;**

**this.right = null;**

**}**

**}**

**class BinarySearchTree {**

**constructor() {**

**this.root = null;**

**}**

**insert(value) {**

**const newNode = new Node(value);**

**if (!this.root) {**

**this.root = newNode;**

**return this;**

**} else {**

**let currentNode = this.root;**

**while (true) {**

**if (value === currentNode.value) return undefined;**

**if (value < currentNode.value) {**

**if (!currentNode.left) {**

**currentNode.left = newNode;**

**return this;**

**}**

**currentNode = currentNode.left;**

**} else {**

**if (!currentNode.right) {**

**currentNode.right = newNode;**

**return this;**

**}**

**currentNode = currentNode.right;**

**}**

**}**

**}**

**}**

**find(value) {**

**if (!this.root) return undefined;**

**let currentNode = this.root;**

**while (currentNode) {**

**if (value === currentNode.value) return currentNode;**

**if (value < currentNode.value) {**

**currentNode = currentNode.left;**

**} else {**

**currentNode = currentNode.right;**

**}**

**}**

**return undefined;**

**}**

**}**

**20. Write a program to insert an element in Binary search trees**.

**class Node {**

**constructor(value) {**

**this.value = value;**

**this.left = null;**

**this.right = null;**

**}**

**}**

**class BinarySearchTree {**

**constructor() {**

**this.root = null;**

**}**

**insert(value) {**

**const newNode = new Node(value);**

**if (!this.root) {**

**this.root = newNode;**

**return this;**

**}**

**let current = this.root;**

**while (true) {**

**if (value === current.value) return undefined;**

**if (value < current.value) {**

**if (!current.left) {**

**current.left = newNode;**

**return this;**

**}**

**current = current.left;**

**} else {**

**if (!current.right) {**

**current.right = newNode;**

**return this;**

**}**

**current = current.right;**

**}**

**}**

**}**

**}**

To use this implementation, you can create a new instance of the **BinarySearchTree** class and call the **insert** function with the value you want to insert:

**const tree = new BinarySearchTree();**

**tree.insert(10);**

**tree.insert(5);**

**tree.insert(15);**

**tree.insert(2);**

**tree.insert(7);**

**tree.insert(12);**

**tree.insert(17);**

**21. Write a program to create BST and implement Pre-order traversal on BST**

**// Node class to represent each node in the BST**

**class Node {**

**constructor(data, left = null, right = null) {**

**this.data = data;**

**this.left = left;**

**this.right = right;**

**}**

**}**

**// BST class to represent the binary search tree**

**class BST {**

**constructor() {**

**this.root = null;**

**}**

**// method to add a new node to the BST**

**add(data) {**

**const newNode = new Node(data);**

**if (this.root === null) {**

**this.root = newNode;**

**} else {**

**this.addNode(this.root, newNode);**

**}**

**}**

**// helper method to add a new node to the BST**

**addNode(node, newNode) {**

**if (newNode.data < node.data) {**

**if (node.left === null) {**

**node.left = newNode;**

**} else {**

**this.addNode(node.left, newNode);**

**}**

**} else {**

**if (node.right === null) {**

**node.right = newNode;**

**} else {**

**this.addNode(node.right, newNode);**

**}**

**}**

**}**

**// method to perform pre-order traversal on the BST**

**preOrderTraversal(node) {**

**if (node !== null) {**

**console.log(node.data);**

**this.preOrderTraversal(node.left);**

**this.preOrderTraversal(node.right);**

**}**

**}**

**}**

**// create a new BST**

**const bst = new BST();**

**// add nodes to the BST**

**bst.add(10);**

**bst.add(5);**

**bst.add(15);**

**bst.add(3);**

**bst.add(7);**

**bst.add(12);**

**bst.add(20);**

**// perform pre-order traversal on the BST**

**bst.preOrderTraversal(bst.root);**

**22. Write a program to create BST and implement In-order traversal on BST**

// Define a class for the BST node

class Node {

constructor(value) {

this.value = value;

this.left = null;

this.right = null;

}

}

// Define a class for the BST

class BST {

constructor() {

this.root = null;

}

// Method to insert a new node into the BST

insert(value) {

const newNode = new Node(value);

if (this.root === null) {

this.root = newNode;

} else {

this.\_insertNode(this.root, newNode);

}

}

\_insertNode(node, newNode) {

if (newNode.value < node.value) {

if (node.left === null) {

node.left = newNode;

} else {

this.\_insertNode(node.left, newNode);

}

} else {

if (node.right === null) {

node.right = newNode;

} else {

this.\_insertNode(node.right, newNode);

}

}

}

// Method to perform in-order traversal

inOrderTraversal() {

const result = [];

this.\_inOrderTraversal(this.root, result);

return result;

}

\_inOrderTraversal(node, result) {

if (node !== null) {

this.\_inOrderTraversal(node.left, result);

result.push(node.value);

this.\_inOrderTraversal(node.right, result);

}

}

}

// Example usage

const bst = new BST();

bst.insert(50);

bst.insert(30);

bst.insert(70);

bst.insert(20);

bst.insert(40);

bst.insert(60);

bst.insert(80);

console.log(bst.inOrderTraversal()); // Output: [20, 30, 40, 50, 60, 70, 80]

**23. Write a program to create BST and implement Post-order traversal on BST.**

class Node {

constructor(data) {

this.data = data;

this.left = null;

this.right = null;

}

}

class BST {

constructor() {

this.root = null;

}

insert(data) {

const newNode = new Node(data);

if (!this.root) {

this.root = newNode;

} else {

this.insertNode(this.root, newNode);

}

}

insertNode(node, newNode) {

if (newNode.data < node.data) {

if (!node.left) {

node.left = newNode;

} else {

this.insertNode(node.left, newNode);

}

} else {

if (!node.right) {

node.right = newNode;

} else {

this.insertNode(node.right, newNode);

}

}

}

postOrderTraversal() {

const result = [];

this.postOrderTraversalHelper(this.root, result);

return result;

}

postOrderTraversalHelper(node, result) {

if (node) {

this.postOrderTraversalHelper(node.left, result);

this.postOrderTraversalHelper(node.right, result);

result.push(node.data);

}

}

}

// example usage

const bst = new BST();

bst.insert(10);

bst.insert(5);

bst.insert(15);

bst.insert(2);

bst.insert(7);

bst.insert(12);

bst.insert(17);

console.log(bst.postOrderTraversal());

// [ 2, 7, 5, 12, 17, 15, 10 ]

**24. Write a program to perform DFS traversal on Graph**

// Define Graph class

class Graph {

constructor() {

this.vertices = [];

this.adjacencyList = {};

}

addVertex(vertex) {

this.vertices.push(vertex);

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2) {

this.adjacencyList[vertex1].push(vertex2);

this.adjacencyList[vertex2].push(vertex1);

}

// DFS traversal method

dfs(startingVertex) {

const visited = {};

const result = [];

// Recursive DFS function

const dfsHelper = (vertex) => {

if (!vertex) return null;

visited[vertex] = true;

result.push(vertex);

this.adjacencyList[vertex].forEach((neighbor) => {

if (!visited[neighbor]) {

return dfsHelper(neighbor);

}

});

};

dfsHelper(startingVertex);

return result;

}

}

// Example usage

const graph = new Graph();

graph.addVertex("A");

graph.addVertex("B");

graph.addVertex("C");

graph.addVertex("D");

graph.addVertex("E");

graph.addEdge("A", "B");

graph.addEdge("A", "C");

graph.addEdge("B", "D");

graph.addEdge("C", "E");

graph.addEdge("D", "E");

console.log(graph.dfs("A")); // Output: ["A", "B", "D", "E", "C"]

**25. Write a program to perform BFS traversal on Graph**

// Define Graph class

class Graph {

constructor() {

this.vertices = [];

this.adjacencyList = {};

}

addVertex(vertex) {

this.vertices.push(vertex);

this.adjacencyList[vertex] = [];

}

addEdge(vertex1, vertex2) {

this.adjacencyList[vertex1].push(vertex2);

this.adjacencyList[vertex2].push(vertex1);

}

// BFS traversal method

bfs(startingVertex) {

const visited = {};

const result = [];

const queue = [startingVertex];

visited[startingVertex] = true;

while (queue.length) {

const currentVertex = queue.shift();

result.push(currentVertex);

this.adjacencyList[currentVertex].forEach((neighbor) => {

if (!visited[neighbor]) {

visited[neighbor] = true;

queue.push(neighbor);

}

});

}

return result;

}

}

// Example usage

const graph = new Graph();

graph.addVertex("A");

graph.addVertex("B");

graph.addVertex("C");

graph.addVertex("D");

graph.addVertex("E");

graph.addEdge("A", "B");

graph.addEdge("A", "C");

graph.addEdge("B", "D");

graph.addEdge("C", "E");

graph.addEdge("D", "E");

console.log(graph.bfs("A")); // Output: ["A", "B", "C", "D", "E"]

**26. Write a program to implement Hash table using array by simple hashing function.**

class HashTable {

constructor(size = 50) {

this.data = new Array(size);

}

\_hash(key) {

let hash = 0;

for (let i = 0; i < key.length; i++) {

hash = (hash + key.charCodeAt(i) \* i) % this.data.length;

}

return hash;

}

set(key, value) {

const address = this.\_hash(key);

if (!this.data[address]) {

this.data[address] = [];

}

this.data[address].push([key, value]);

}

get(key) {

const address = this.\_hash(key);

const currentBucket = this.data[address];

if (currentBucket) {

for (let i = 0; i < currentBucket.length; i++) {

if (currentBucket[i][0] === key) {

return currentBucket[i][1];

}

}

}

return undefined;

}

keys() {

const keysArray = [];

for (let i = 0; i < this.data.length; i++) {

if (this.data[i]) {

for (let j = 0; j < this.data[i].length; j++) {

keysArray.push(this.data[i][j][0]);

}

}

}

return keysArray;

}

}

**27. Write a program to implement Linear Search algorithm(Brute Force approach)**

function linearSearch(arr, x) {

for (let i = 0; i < arr.length; i++) {

if (arr[i] === x) {

return i;

}

}

return -1; // return -1 if element not found

}

// Example usage

const arr = [2, 5, 8, 10, 13, 17];

const elementToFind = 10;

const foundIndex = linearSearch(arr, elementToFind);

if (foundIndex === -1) {

console.log(`${elementToFind} not found in array`);

} else {

console.log(`${elementToFind} found at index ${foundIndex}`);

}

**28. Write a program to implement Binary Search algorithm(Divide and conquer approach)**

function binarySearch(arr, target) {

let left = 0;

let right = arr.length - 1;

while (left <= right) {

const mid = Math.floor((left + right) / 2);

if (arr[mid] === target) {

return mid;

} else if (arr[mid] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

**Here's how to use this function:**

const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const target = 7;

const result = binarySearch(arr, target);

if (result === -1) {

console.log("Element not found in array");

} else {

console.log(`Element found at index ${result}`);

}

**29. Write a program to solve recursive staircases problem using brute force approach.**

function countSteps(n) {

if (n === 0) {

return 1;

} else if (n < 0) {

return 0;

} else {

return countSteps(n-1) + countSteps(n-2) + countSteps(n-3);

}

}

console.log(countSteps(5));

// prints 13

This program uses a recursive function **countSteps(n)** to count the number of ways to climb **n** stairs. The function checks the base cases where **n** is either 0 or negative, and returns the appropriate value. Otherwise, it recursively calls itself with **n-1**, **n-2**, and **n-3**, and adds up the results.

Note that this approach can be slow for larger values of **n**, since it recomputes the same values over and over again. A more efficient approach would use memoization or dynamic programming to avoid recomputing values.

**30. Write a program to find maximum sum of a subarray among all subarrays of given array using brute force approach**

function maxSubarraySum(arr) {

let maxSum = -Infinity;

for (let i = 0; i < arr.length; i++) {

let sum = 0;

for (let j = i; j < arr.length; j++) {

sum += arr[j];

maxSum = Math.max(maxSum, sum);

}

}

return maxSum;

}

// example usage

const arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4];

console.log(maxSubarraySum(arr));

// output: 6

**31. Write a program to solve jump game problem using greedy method.**

function canJump(nums) {

let lastPos = nums.length - 1;

for (let i = nums.length - 2; i >= 0; i--) {

if (i + nums[i] >= lastPos) {

lastPos = i;

}

}

return lastPos === 0;

}

Here's an example usage of the function:

**console.log(canJump([2,3,1,1,4])); // true**

**console.log(canJump([3,2,1,0,4])); // false**

**32. Write a program to implement Prim’s algorithm to find minimum spanning tree using greedy method**

function prim(graph) {

const n = graph.length;

const visited = Array(n).fill(false); // keep track of visited vertices

const distances = Array(n).fill(Infinity); // initialize distances to infinity

const parent = Array(n).fill(null); // parent of each vertex in MST

// start with vertex 0 as root

distances[0] = 0;

// priority queue to keep track of next vertex to add to MST

const pq = new PriorityQueue((a, b) => distances[a] - distances[b]);

pq.enqueue(0);

while (!pq.isEmpty()) {

const u = pq.dequeue();

visited[u] = true;

// update distances and parent for all neighbors of u

for (const [v, weight] of graph[u]) {

if (!visited[v] && weight < distances[v]) {

distances[v] = weight;

parent[v] = u;

pq.enqueue(v);

}

}

}

// construct MST from parent array

const mst = [];

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

mst.push([parent[i], i, graph[parent[i]][i]]);

}

return mst;

}

// priority queue implementation

class PriorityQueue {

constructor(comparator) {

this.comparator = comparator || ((a, b) => a - b);

this.heap = [];

}

enqueue(item) {

this.heap.push(item);

this.bubbleUp(this.heap.length - 1);

}

dequeue() {

const item = this.heap[0];

const last = this.heap.pop();

if (this.heap.length > 0) {

this.heap[0] = last;

this.bubbleDown(0);

}

return item;

}

peek() {

return this.heap[0];

}

isEmpty() {

return this.heap.length === 0;

}

size() {

return this.heap.length;

}

bubbleUp(index) {

while (index > 0) {

const parent = Math.floor((index - 1) / 2);

if (this.comparator(this.heap[index], this.heap[parent]) < 0) {

const temp = this.heap[parent];

this.heap[parent] = this.heap[index];

this.heap[index] = temp;

index = parent;

} else {

break;

}

}

}

bubbleDown(index) {

while (true) {

const left = 2 \* index + 1;

const right = 2 \* index + 2;

let smallest = index;

if (left < this.heap.length && this.comparator(this.heap[left], this.heap[smallest]) < 0) {

smallest = left;

}

if (right < this.heap.length && this.comparator(this.heap[right], this.heap[smallest]) < 0) {

smallest = right;

}

if (smallest !== index) {

const temp = this.heap[smallest];

this.heap[smallest] = this.heap[index];

this.heap[index] = temp;

index = smallest;

} else {

break;

}

}

}

}

**33. Write a program to implement Euclidean Algorithm using divide and conquer method**

function gcd(a, b) {

// Base case

if (a === b) {

return a;

}

// If one of the numbers is 0, return the other number as GCD

if (a === 0) {

return b;

}

if (b === 0) {

return a;

}

// If both numbers are even, divide them by 2

if (a % 2 === 0 && b % 2 === 0) {

return 2 \* gcd(a/2, b/2);

}

// If a is even and b is odd, divide a by 2

if (a % 2 === 0 && b % 2 !== 0) {

return gcd(a/2, b);

}

// If a is odd and b is even, divide b by 2

if (a % 2 !== 0 && b % 2 === 0) {

return gcd(a, b/2);

}

// If both numbers are odd and a > b, subtract b from a

if (a > b) {

return gcd((a - b)/2, b);

}

// If both numbers are odd and b > a, subtract a from b

return gcd(a, (b - a)/2);

}

**34. Write a program to sort an array using Quick sort. (divide and conquer method)**

function quickSort(arr) {

if (arr.length <= 1) {

return arr;

}

const pivot = arr[arr.length - 1];

const leftArr = [];

const rightArr = [];

for (let i = 0; i < arr.length - 1; i++) {

if (arr[i] < pivot) {

leftArr.push(arr[i]);

} else {

rightArr.push(arr[i]);

}

}

return [...quickSort(leftArr), pivot, ...quickSort(rightArr)];

}

const arr = [5, 3, 8, 4, 2, 7, 1, 6];

const sortedArr = quickSort(arr);

console.log(sortedArr);

**35. Write a program to solve Tower of Hanoi problem using divide and conquer method**

function towerOfHanoi(n, fromRod, toRod, auxRod) {

if (n === 1) {

console.log(`Move disk 1 from ${fromRod} to ${toRod}`);

return;

}

towerOfHanoi(n - 1, fromRod, auxRod, toRod);

console.log(`Move disk ${n} from ${fromRod} to ${toRod}`);

towerOfHanoi(n - 1, auxRod, toRod, fromRod);

}

// Example usage

towerOfHanoi(3, "A", "C", "B");

**36. Write a program to find Fibonacci number using Dynamic programming**

function fibonacci(n) {

let memo = {};

function fib(n) {

if (n in memo) {

return memo[n];

}

if (n <= 2) {

return 1;

}

memo[n] = fib(n - 1) + fib(n - 2);

return memo[n];

}

return fib(n);

}

To use this program, simply call the **fibonacci** function with the index of the Fibonacci number you want to calculate:

**console.log(fibonacci(10)); // outputs 55**

**37. Write a program to find unique path using Dynamic programming.**

function uniquePaths(m, n) {

// Initialize a 2D array to store the number of unique paths

const dp = new Array(m).fill().map(() => new Array(n).fill(1));

// Fill the array using dynamic programming

for (let i = 1; i < m; i++) {

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

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

}

}

// Return the number of unique paths

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

}

// Test the function with some sample inputs

console.log(uniquePaths(3, 2)); // should output 3

console.log(uniquePaths(7, 3)); // should output 28

**38. Write a program to implement N-Queens Problem using Backtracking method**

function isSafe(board, row, col, n) {

// check the row

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

if (board[row][i] === 1) {

return false;

}

}

// check the upper diagonal

for (let i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board[i][j] === 1) {

return false;

}

}

// check the lower diagonal

for (let i = row, j = col; j >= 0 && i < n; i++, j--) {

if (board[i][j] === 1) {

return false;

}

}

return true;

}

function solveNQueens(board, col, n) {

if (col >= n) {

return true;

}

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

if (isSafe(board, i, col, n)) {

board[i][col] = 1;

if (solveNQueens(board, col + 1, n)) {

return true;

}

board[i][col] = 0;

}

}

return false;

}

function printBoard(board, n) {

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

let row = "";

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

row += board[i][j] + " ";

}

console.log(row);

}

}

function solve(n) {

let board = Array.from({ length: n }, () => Array.from({ length: n }, () => 0));

if (solveNQueens(board, 0, n) === false) {

console.log("Solution does not exist.");

} else {

printBoard(board, n);

}

}

solve(4); // output: 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0