**FPRT: CORE-CSE**

1. **List out different OOPS principles and explain with example**

There are 4 different principles in OOPS. They are as follows:

**\***Encapsulation

**\***Inheritance

**\***Abstraction

**\***Polymorphism

**Encapsulation:**

It is the process of binding together data and functions that manipulate the data. It helps to protect the data from outside interference and misuse.

**Example:**

class Account {

constructor(balance) {

this.\_balance = balance;

}

get balance() {

return this.\_balance;

}

set balance(amount) {

this.\_balance = amount;

}

}

const account = new Account(1000);

console.log(account.balance); // 1000

account.balance = 2000;

console.log(account.balance); // 2000

**Inheritance:**

It allows new class to inherit properties and behavior of an existing class.

**Example:**

class Shape {

constructor(width, height) {

this.width = width;

this.height = height;

}

getArea() {

return this.width \* this.height;

}

}

class Rectangle extends Shape {

constructor(width, height) {

super(width, height);

}

}

const rectangle = new Rectangle(10, 20);

console.log(rectangle.getArea()); // 200

**Abstraction:**

It is the process of hiding complex implementation details and showing only the necessary information to the user

**Example:**

class Vehicle {

startEngine() {

console.log('Engine started.');

}

}

class Car extends Vehicle {

drive() {

this.startEngine();

console.log('Driving...');

}

}

const car = new Car();

car.drive();

// Output:

// Engine started.

// Driving...

**Polymorphism:**

It allows objects to take on different forms

**Example:**

class Animal {

makeSound() {

throw new Error('You have to implement the method');

}

}

class Dog extends Animal {

makeSound() {

return 'Woof!';

}

}

class Cat extends Animal {

makeSound() {

return 'Meow!';

}

}

const dog = new Dog();

const cat = new Cat();

console.log(dog.makeSound()); // Woof!

console.log(cat.makeSound()); // Meow!

1. **Explain data structures that are mutable versus immutable**

A mutable data structure is a data structure that can be changed after it is created.**Ex:**Arrays, Objects

An immutable data structure is a data structure that cannot be changed after it is created. **Ex:** numbers, strings, Boolean

1. **Construct a binary tree using in-order and post-order traversal given below.**

**Inorder Traversal: 9, 3, 15, 20, 7**

**Post-Order Traversal: 9, 15, 7, 20, 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

1. **Construct a binary search tree using pre order traversal given below.**

**Pre order Traversal: 50 30 20 40 70 60 80**

**//to create a tree**

class BinaryTree{

constructor(data,left,right){

this.data = data;

this.left = left;

this.right = right;

}

}

const root = new BinaryTree (50,null,null);

node2 = new BinaryTree (30,null,null);

node3 = new BinaryTree (20,null,null);

root.left = node2;

root.right = node3;

node4 = new BinaryTree (40,null,null);

node5 = new BinaryTree (70,null,null);

node2.left = node4;

node2.right = node5;

node6 = new BinaryTree (60,null,null);

node7 = new BinaryTree (80,null,null);

node3.left = node6;

node3.right = node7;

console.log (root);

// after this our tree will look like

// 50

// / \

// 30 20

// / \ / \

// 40 70 60 80

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

function preorder (root){

//base condition

if (root == null) return;

//visit root node

console.log (root.data);

//visit left child

preorder(root.left);

//visit right child

preorder (root.right);

}

preorder (root); // **50,30,40,70,20,60,80**

1. What is the time complexity of

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

j=1;

while (j>n) {

console.log(i)

j=j\*2

}

}

The time complexity will be **O(nlogn);**

1. What is the time complexity of

i = 1;

while(i2<n) {

i+=1}

the time complexity will be **O(sqrt(n))**

1. What is the time complexity of

function bblSort(arr){

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

break;

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

if(arr[j] > arr[j+1]){

var temp = arr[j]

arr[j] = arr[j + 1]

arr[j+1] = temp

}

}

}

console.log(arr);

}

The time complexity will be **O(n^2)**