**1. Usa la recursividad**

La **recursividad** es un concepto donde una función se llama a sí misma para resolver problemas más pequeños del mismo tipo. Es útil en problemas que pueden dividirse en subproblemas idénticos, como cálculos matemáticos o exploración de estructuras como árboles.

**2. Define y crea la estructura Árbol**

Un **árbol** es una estructura de datos jerárquica compuesta de nodos. El nodo superior se llama raíz (**root**) y cada nodo puede tener nodos hijos (**children**). Es ideal para modelar relaciones jerárquicas como directorios de archivos o árboles genealógicos.

**3. Usa Algoritmos Pre-order Traversal**

El **pre-order traversal** es una forma de recorrer un árbol donde:

1. Visitas el nodo raíz.
2. Recorres el subárbol izquierdo.
3. Recorres el subárbol derecho.

**4. Usa Algoritmos Post-order Traversal**

El **post-order traversal** es otra forma de recorrer un árbol:

1. Recorres el subárbol izquierdo.
2. Recorres el subárbol derecho.
3. Visitas el nodo raíz.

**EJEMPLOS**

1. **Recursividad: Calcular el factorial de un número**

public class Recursividad {

public static int factorial(int n) {

if (n == 0 || n == 1) {

return 1; // Caso base

}

return n \* factorial(n - 1); // Llamada recursiva

}

public static void main(String[] args) {

int numero = 5;

System.out.println("El factorial de " + numero + " es: " + factorial(numero));

}

}

1. **Definir y crear la estructura Árbol**

class Nodo {

int valor;

Nodo izquierdo;

Nodo derecho;

Nodo(int valor) {

this.valor = valor;

this.izquierdo = null;

this.derecho = null;

}

}

public class Arbol {

Nodo raiz;

public Arbol(int valorRaiz) {

this.raiz = new Nodo(valorRaiz);

}

public void agregarIzquierdo(Nodo nodoPadre, int valor) {

nodoPadre.izquierdo = new Nodo(valor);

}

public void agregarDerecho(Nodo nodoPadre, int valor) {

nodoPadre.derecho = new Nodo(valor);

}

public static void main(String[] args) {

Arbol arbol = new Arbol(10);

arbol.agregarIzquierdo(arbol.raiz, 5);

arbol.agregarDerecho(arbol.raiz, 15);

System.out.println("Árbol creado con raíz: " + arbol.raiz.valor);

}

}

1. **Pre-order Traversal**

public class PreOrderTraversal {

public static void preOrder(Nodo nodo) {

if (nodo == null) return;

System.out.print(nodo.valor + " ");

preOrder(nodo.izquierdo);

preOrder(nodo.derecho);

}

public static void main(String[] args) {

Nodo raiz = new Nodo(10);

raiz.izquierdo = new Nodo(5);

raiz.derecho = new Nodo(15);

System.out.println("Recorrido Pre-order:");

preOrder(raiz);

}

}

1. **Post-order Traversal**

public class PostOrderTraversal {

public static void postOrder(Nodo nodo) {

if (nodo == null) return;

postOrder(nodo.izquierdo);

postOrder(nodo.derecho);

System.out.print(nodo.valor + " ");

}

public static void main(String[] args) {

Nodo raiz = new Nodo(10);

raiz.izquierdo = new Nodo(5);

raiz.derecho = new Nodo(15);

System.out.println("Recorrido Post-order:");

postOrder(raiz);

}

}

1. **Ejercicios**
2. Implementa una función recursiva que calcule la suma de los primeros n números naturales.
3. Define un árbol binario con los valores 20, 10, 30. Impleméntalo y muestra la raíz.
4. Realiza un recorrido pre-order en un árbol con los valores 50, 25, 75.
5. Implementa un árbol binario con un nodo raíz y dos hijos. Imprime sus valores.
6. Haz un recorrido post-order en un árbol con los valores 10, 5, 15.

**RETOS:**

1. Crea una función recursiva para encontrar el número máximo en un árbol binario.
2. Implementa un árbol binario con al menos 7 nodos y realiza un recorrido pre-order.
3. Implementa un algoritmo para contar la cantidad de nodos en un árbol binario utilizando recursividad.
4. Diseña un algoritmo recursivo que invierta los hijos izquierdo y derecho de un árbol binario.
5. Crea un árbol con nodos con valores arbitrarios y calcula la suma total de todos los valores utilizando recursión.