Universidad Nacional Abierta
Vicerrectorado Académico
Área de Ingeniería
Carrera Ingeniería de Sistemas

Trabajo práctico sustitutivo

Asignatura: Computación II Código: 324

Fecha de devolución: A más tardar el **12-03-2024** (Sin prórroga)

Nombre del Estudiante: **César Antonio Torres Chang**

Cédula de Identidad: **V-20246713**

Centro Local / Unidad de Apoyo: **Metropolitano**

Correo electrónico: cesarchang23@gmail.com Teléfono celular: +51 928 650 463

Carrera: **Ingeniería de Sistemas (cod. 236)**

Número de originales:

Lapso: 2024-1

Resultados de Corrección

	Objetivos			
Logrado: 1	**1**	**2**	**3**	**4**
No logrado: 0				

ESPECIFICACIONES DEL TRABAJO PRÁCTICO SUSTITUTIVO

M: 1, U: 1, O: 1 **C/D: 1/1**

1. Realice un TAD en C++ para verificar si dado 2 números complejos su suma y multiplicación son iguales.

/*

Los números complejos tienen una parte real y una parte imaginaria, ejemplo: 2 + 3i
i = sqrt(-1)

Suma de dos (2) números complejos:

Dados Z_1 = 1 + i y Z_2 = 2 + 2i

Se suma (parte real de Z_1 + parte real de Z_2) + (parte imaginaria de Z_1 + parte imaginaria de Z_2)

1 + i + 2 + 2i = 3 + 3i

Multiplicación de dos (2) números complejos:

Dados Z_1 = 1 + i y Z_2 = 2 + 2i

(a+bi) * (c+di) = (a*c-b*d) + (a*d+b*c)i

(1 + i) * (2+2i) = 1*2 - 2 + (2+2)i

= (2-2) + (4)i

= 0 + 4i

= 4i

Si el resultado de la suma es A y el producto es B nos piden una demostración donde se compruebe si A == B

*/

```
#include <iostream>

using namespace std;

// Definición de la estructura para generar números complejos
struct numerosComplejos{
   float real, imaginario;
} z_1, z_2;

//Prototipos de funciones
void ingresarDatos();
numerosComplejos suma(const numerosComplejos& z_1, const numerosComplejos& z_2);
numerosComplejos producto(const numerosComplejos& z_1, const numerosComplejos& z_2);
bool sonIguales(const numerosComplejos& z_1, const numerosComplejos& z_2);
void mostrarResultado(const numerosComplejos& a, const numerosComplejos& b);

int main(){
   char continuar;

// Bucle principal para permitir al usuario realizar múltiples operaciones
   do{
      ingresarDatos(); // Solicitar al usuario que ingrese los números complejos

 // Calcular la suma y el producto de los números complejos ingresados
      numerosComplejos a = suma(z_1, z_2);
      numerosComplejos b = producto(z_1, z_2);

      mostrarResultado(a,b);

      cout<<"\n¿Desea ingresar nuevos valores? (s/n): "; cin >> continuar;
   } while (continuar == 'S' || continuar == 's');
   return 0;
}
```

```
void ingresarDatos(){
    cout<<"Ingresa los datos del primer número complejo"<< endl;
    cout<<"Parte real: "; cin>>z_1.real;
    cout<<"Parte imaginaria: "; cin>>z_1.imaginario;

    cout<<"\n\nIngresa los datos del segundo número complejo"<< endl;
    cout<<"Parte real: "; cin>>z_2.real;
    cout<<"Parte imaginaria: "; cin>>z_2.imaginario;
}

numerosComplejos suma(const numerosComplejos& z_1, const numerosComplejos& z_2){
    // (a+bi) + (c+di) = (a + c) + (b + d)i

    numerosComplejos resultadoSuma;
    resultadoSuma.real = z_1.real + z_2.real;
    resultadoSuma.imaginario = z_1.imaginario + z_2.imaginario;
    return resultadoSuma;
}

numerosComplejos producto(const numerosComplejos& z_1, const numerosComplejos& z_2){
    // (a+bi) * (c+di) = (a*c-b*d) + (a*d+b*c)i

    numerosComplejos resultadoProducto;
    resultadoProducto.real = z_1.real * z_2.real - z_1.imaginario * z_2.imaginario;
    resultadoProducto.imaginario = z_1.real * z_2.imaginario + z_1.imaginario * z_2.real;
    return resultadoProducto;
}

// Función para comparar resultados de suma y multiplicación
bool sonIguales(const numerosComplejos& z1, const numerosComplejos& z2){
    return (z1.real == z2.real) && (z1.imaginario == z2.imaginario);
}
```

```
void mostrarResultado(const numerosComplejos& a, const numerosComplejos& b){
    cout << "Suma: " << a.real << " + " << a.imaginario << "i" <<endl;
    cout << "Producto: " << b.real << " + " << b.imaginario << "i" <<endl;

    if (sonIguales(a,b)){
        cout << "Los resultados de la suma y el producto son iguales." <<endl;
    } else {
        cout << "Los resultados de la suma y el producto no son iguales." <<endl;
    }
}
```

M: 1, U: 2, O: 2 **C/D: 1/1**

2.Realice un programa en C++ que construya una función **imprimeInverso** que imprima los elementos de una lista enlazada de enteros en orden inverso a partir de una posición p.

Este programa permite al usuario interactuar con una lista enlazada. Una lista enlazada es una estructura de datos donde los elementos están dispuestos en una secuencia y cada elemento está vinculado al siguiente mediante un puntero. El programa le permite al usuario realizar las siguientes operaciones:

- Inserción de elementos: El usuario puede agregar elementos a la lista. El programa solicitará al usuario que ingrese la cantidad de elementos que desea agregar y luego le pedirá que ingrese los valores de cada elemento.
- Impresión en orden inverso: Una vez que se ingresan los elementos, el programa los imprimirá en orden inverso. Esto se logra mediante la recursión: el programa recorre la lista desde el último elemento hasta el primero e imprime los valores en este orden.
- Reinicio de la lista: Después de imprimir los elementos en orden inverso, el programa le preguntará al usuario si desea reiniciar la lista. Si el usuario elige reiniciar la lista, todos los elementos se eliminarán y el programa volverá al paso de inserción de elementos. Si el usuario decide no reiniciar la lista, el programa termina.

El programa está diseñado para ser robusto y maneja errores de entrada del usuario, asegurando que solo se ingresen números enteros cuando se solicite. Esto se logra mediante el uso de bucles y verificaciones de entrada.

```
#include <iostream>
#include <limits>
using namespace std;

// Definición de la clase Nodo
class Nodo {
public:
    int dato;       // Dato almacenado en el nodo
    Nodo* siguiente; // Puntero al siguiente nodo en la lista

    // Constructor
    Nodo(int valor) : dato(valor), siguiente(nullptr) {}
};

// Definición de la clase Lista
class Lista {
private:
    Nodo* inicio;   // Puntero al primer nodo de la lista
    Nodo* fin;      // Puntero al último nodo de la lista

public:
    // Constructor
    Lista() : inicio(nullptr), fin(nullptr) {}

    // Destructor
    ~Lista() {
        reiniciarLista();
    }

    // Método para verificar si la lista está vacía
    bool esVacia() {
```

```
        return inicio == nullptr;
    }

    // Método para insertar un elemento al final de la lista
    void insertarAlFinal(int dato) {
        Nodo *nuevo = new Nodo(dato); // Crear un nuevo nodo con el dato proporcionado
        if(esVacia()) { // Si la lista está vacía
            inicio = fin = nuevo; // Establecer el nuevo nodo como inicio y fin de la lista
        } else { // Si la lista no está vacía
            fin->siguiente = nuevo; // El último nodo apunta al nuevo nodo
            fin = nuevo; // El nuevo nodo se convierte en el último nodo de la lista
        }
    }

    // Método para imprimir en orden inverso los elementos de la lista
    void imprimeInverso() {
        imprimeInversoRecursivo(inicio); // Llamar al método recursivo para imprimir en orden
inverso
    }

    // Método para reiniciar la lista completamente
    void reiniciarLista() {
        Nodo* actual = inicio;
        while (actual != nullptr) { // Recorrer la lista mientras haya nodos
            Nodo* temp = actual; // Guardar el nodo actual en una variable temporal
            actual = actual->siguiente; // Avanzar al siguiente nodo
            delete temp; // Eliminar el nodo guardado en la variable temporal
        }
        inicio = fin = nullptr; // Establecer inicio y fin como nullptr para indicar una lista vacía
    }
```

```
private:
    // Método privado para imprimir en orden inverso los elementos de la lista
(implementación recursiva)
    void imprimeInversoRecursivo(Nodo* p) {
        if (p != nullptr) { // Si el nodo actual no es nullptr
            imprimeInversoRecursivo(p->siguiente); // Llamar recursivamente con el siguiente
nodo
            cout << p->dato << " "; // Imprimir el dato del nodo actual
        }
    }
};

int main() {
    Lista lista;

    do {
        // Iteración para ingresar elementos en la lista
        do {
            // Solicitar al usuario que ingrese los elementos de la lista
            int numElementos;
            cout << "Ingrese el número de elementos de la lista: ";
            while (!(cin >> numElementos)) { // Validar la entrada del usuario
                cout << "Error: Solo se permiten números enteros. Intente de nuevo: ";
                cin.clear(); // Limpiar el estado de error del flujo de entrada
                cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Descartar la entrada
incorrecta
            }

            cout << "Ingrese los elementos de la lista:\n";
            for (int i = 0; i < numElementos; ++i) {
                int elemento;
```

```
            cout << "Elemento " << i+1 << ": ";
            while (!(cin >> elemento)) { // Validar la entrada del usuario
                cout << "Error: Solo se permiten números enteros. Intente de nuevo: ";
                cin.clear(); // Limpiar el estado de error del flujo de entrada
                cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Descartar la entrada
incorrecta
            }
            lista.insertarAlFinal(elemento); // Insertar el elemento en la lista
        }

        cout << "Elementos en orden inverso: ";
        lista.imprimeInverso(); // Imprimir los elementos de la lista en orden inverso
        cout << endl;

        // Preguntar al usuario si desea ingresar más elementos
        char respuesta;
        cout << "¿Desea ingresar más elementos? (s/n): ";
        cin >> respuesta;
        if (respuesta != 's' && respuesta != 'S') {
            break; // Salir del bucle si la respuesta no es 's' o 'S'
        }

    } while (true); // Bucle infinito para ingresar elementos en la lista

    // Preguntar al usuario si desea reiniciar la lista
    char respuesta;
    cout << "¿Desea reiniciar la lista? (s/n): ";
    cin >> respuesta;
    if (respuesta == 's' || respuesta == 'S') {
        lista.reiniciarLista(); // Reiniciar la lista eliminando todos los elementos
    } else {
```

```
            cout << "¡Hasta Luego!" << endl; // Mensaje de despedida si el usuario decide no
reiniciar la lista
            break; // Salir del bucle principal si el usuario decide no reiniciar la lista
        }

    } while (true); // Bucle infinito para manejar múltiples listas

    return 0;
}
```

M: 2, U: 3, O: 3 **C/D: 1/1**

3. Se dispone de un compilador que únicamente tiene implementada la estructura de datos pila, y las siguientes operaciones asociadas. -
void Inicializar (Tpila *pila);

- void Apilar (Tpila *pila,Telem elem);
- void Sacar (Tpila *pila);
- Telem Cima (Tpila Pila, Telem Elemento);
- BOOLEAN Vacia (Tpila Pila);

Se pide implementar una función que invierta una pila utilizando las operaciones descritas anteriormente. Se podrán utilizar aquellas estructuras auxiliares que tengan implementadas el compilador. Se invertirá el contenido de la pila utilizando una pila auxiliar.

Juego de la Torre (Basado en el concepto de Pila)

¡Bienvenido al emocionante juego de la Torre! En este juego, tendrás la oportunidad de experimentar cómo funciona una pila a través de la analogía de una torre. La torre en este juego se comporta de manera similar a una pila en la programación, lo que te permitirá comprender y familiarizarte con los conceptos fundamentales de esta estructura de datos.

¿Qué es una torre?

En este juego, la "torre" se representa como una estructura vertical en la que puedes apilar diferentes elementos uno encima del otro. Cada elemento que agregues se colocará en la parte superior de la torre, y cuando desapiles un elemento, se retirará el último elemento apilado (el que está en la cima).

Similitudes con una Pila:

Apilar (Push): Agregar un elemento a la torre se asemeja a la operación de "apilar" en una pila. Al apilar un elemento, este se coloca en la parte superior de la torre, ocupando el lugar más alto.

Desapilar (Pop): La acción de retirar un elemento de la torre es similar a la operación de "desapilar" en una pila. Al desapilar un elemento, se elimina el último elemento apilado, que es el que se encuentra en la cima de la torre.

Cima de la Torre (Top): Consultar el elemento en la cima de la torre es equivalente a obtener el elemento superior de una pila. Esto te permite conocer el elemento más recientemente apilado.

Torre Vacía: Al igual que una pila, la torre puede estar vacía, lo que significa que no hay elementos apilados en ella. Esta condición se verifica antes de realizar operaciones como desapilar o consultar la cima de la torre para evitar errores.

Invertir la Torre:

Una de las características interesantes de este juego es la capacidad de invertir la torre. Esta acción se realiza moviendo todos los elementos de la torre en el orden opuesto al que se apilaron originalmente. Utilizando una técnica similar a la de una pila auxiliar, la torre se invierte sin alterar los elementos individuales.

Objetivo del Juego:

El objetivo del juego es experimentar con las operaciones de la torre (equivalentes a las operaciones de una pila) y comprender cómo se comportan los elementos al ser apilados, desapilados, consultados y cómo se puede invertir el orden de la torre. Practica y domina estas operaciones para convertirte en un maestro de la torre y un experto en el concepto de pila.

```
/*
    Utilizando la teoría de la pila del enunciado del ejercicio, hice un juego sencillo de la Torre.
/*

#include <iostream>
#include <stack> // Incluye la biblioteca para utilizar la estructura de datos pila
#include <stdexcept>
#include <limits> // Para usar numeric_limits

using namespace std;

class Torre {
private:
    stack<string> elementos; // Contenedor para los elementos de la torre

public:
    // Método para agregar un elemento a la torre
    void Apilar(const string& elem) {
        elementos.push(elem); // Añade un elemento a la pila
    }

    // Método para remover un elemento de la torre
    void Sacar() {
```

```
        if (elementos.empty()) {
            throw runtime_error("No se puede sacar de una torre vacía"); // Si la pila está vacía, lanza una excepción
        }
        elementos.pop(); // Saca un elemento de la pila
    }

    // Método para obtener el elemento en la cima de la torre sin removerlo
    string Cima() {
        if (elementos.empty()) {
            throw runtime_error("No se puede acceder a la cima de una torre vacía"); // Si la pila está vacía, lanza una excepción
        }
        return elementos.top(); // Devuelve el elemento en la cima de la pila
    }

    // Método para verificar si la torre está vacía
    bool Vacia() {
        return elementos.empty(); // Verifica si la pila está vacía
    }

    // Método para invertir el orden de los elementos en la torre
    void Invertir() {
        if (elementos.empty()) {
            cout << "La torre ya está vacía, no se puede invertir." << endl; // Si la pila está vacía, muestra un mensaje
            return;
        }

        stack<string> torre_aux; // Torre auxiliar para invertir la torre
```

```
        // Movemos los elementos de la torre original a la torre auxiliar en el orden deseado
        while (!elementos.empty()) {
            torre_aux.push(elementos.top());
            elementos.pop();
        }

        stack<string> torre_invertida; // Torre para almacenar la torre invertida

        // Transferimos los elementos de la torre auxiliar a la torre invertida en el orden
invertido
        while (!torre_aux.empty()) {
            torre_invertida.push(torre_aux.top());
            torre_aux.pop();
        }

        // Copiamos la torre invertida de vuelta a la torre original
        while (!torre_invertida.empty()) {
            elementos.push(torre_invertida.top());
            torre_invertida.pop();
        }

        cout << "La torre se ha invertido exitosamente." << endl; // Muestra un mensaje de
éxito
    }

    // Método para mostrar los elementos de la torre
    void MostrarTorre() {
        stack<string> torre = elementos; // Creamos una copia de la torre original para no
modificarla
        cout << "Torre: ";
        while (!torre.empty()) {
```

```
            cout << torre.top() << " "; // Muestra los elementos de la pila
            torre.pop();
        }
        cout << endl;
    }

    // Método para limpiar la torre
    void Limpiar() {
        while (!elementos.empty()) {
            elementos.pop(); // Saca todos los elementos de la pila
        }
        cout << "La torre ha sido limpiada." << endl; // Muestra un mensaje de éxito
    }
};

int main() {
    Torre torre; // Creamos un objeto de la clase Torre

    cout << "¡Bienvenido al juego de la torre!" << endl;

    int cantidad_maxima_elementos;
    cout << "Ingrese la cantidad máxima de elementos que puede tener la torre: ";
    while (!(cin >> cantidad_maxima_elementos) || cantidad_maxima_elementos <= 0) {
        // Limpiar el búfer de entrada en caso de error
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cout << "Error. Por favor ingrese un número entero mayor que cero: ";
    }
```

```
    // Bucle principal que controla la iteración del juego
    while (true) {
        cout << "Seleccione una acción:" << endl;
        cout << "1. Apilar elemento" << endl;
        cout << "2. Desapilar elemento" << endl;
        cout << "3. Mostrar cima de la torre" << endl;
        cout << "4. Invertir la torre" << endl;
        cout << "5. Mostrar la torre" << endl;
        cout << "6. Limpiar la torre" << endl;
        cout << "7. Salir del juego" << endl;

        int opcion;
        cout << "Opción: ";
        while (!(cin >> opcion) || opcion < 1 || opcion > 7) {
            // Limpiar el búfer de entrada en caso de error
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            cout << "Error. Por favor seleccione una opción válida: ";
        }

        switch (opcion) {
            case 1: {
                string elemento;
                cout << "Ingrese el elemento a apilar: ";
                cin >> elemento;
                torre.Apilar(elemento); // Apila un elemento en la pila
                cout << "Elemento apilado correctamente." << endl; // Muestra un mensaje de
éxito
                break;
            }
```

```
        case 2: {
            try {
                torre.Sacar(); // Desapila un elemento de la pila
                cout << "Elemento desapilado correctamente." << endl; // Muestra un mensaje
de éxito
            } catch (const runtime_error& e) {
                cout << e.what() << endl; // Muestra un mensaje de error si la pila está vacía
            }
            break;
        }
        case 3: {
            try {
                cout << "Cima de la torre: " << torre.Cima() << endl; // Muestra la cima de la
pila
            } catch (const runtime_error& e) {
                cout << e.what() << endl; // Muestra un mensaje de error si la pila está vacía
            }
            break;
        }
        case 4:
            torre.Invertir(); // Invierte el orden de los elementos en la pila
            break;
        case 5:
            torre.MostrarTorre(); // Muestra todos los elementos de la pila
            break;
        case 6:
            torre.Limpiar(); // Limpia la pila
            break;
        case 7:
            cout << "Gracias por jugar. ¡Hasta luego!" << endl; // Muestra un mensaje de
despedida y sale del programa
```

```
            return 0;
        }
    }

    return 0;
}
```

M: 2, U: 4, O: 4 **C/D: 1/1**

4. El recorrido en preorden de un determinado árbol binario es: GEAIBMCLDFKJH y en inorden IABEGLDCFMKHJ. Resolver:

A) Dibujar el árbol binario.

B) Dar el recorrido en postorden.

C) Hacer una función en C++ para dar el recorrido en postorden dado el recorrido en preorden e inorden y hacer un programa en C++ para comprobar el resultado del apartado anterior.

ÁRBOL BINARIO

Generaremos el árbol binario a través de la siguiente tabla.

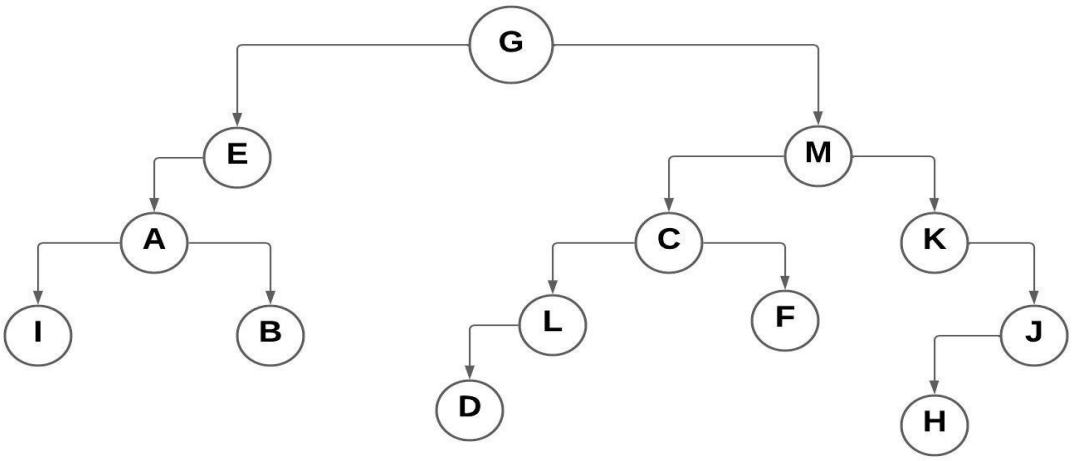
Cada letra es un nodo:

1- De izquierda a derecha se escribe el recorrido in-orden.

2- De arriba hacia abajo se escribe el recorrido pre-orden.

Unimos los puntos de coincidencia en cada renglón para hallar la forma del árbol binario.

	I	A	B	E	G	L	D	C	F	M	K	H	J
G					O								
E				O									
A		O											
I	O												
B			O										
M										O			
C								O					
L						O							
D							O						
F									O				
K											O		
J													O
H												O	



Recorrido en post-orden:

IBAEDLFCHJKMG

El programa implementa la construcción de un árbol binario a partir de dos tipos de recorridos: preorden e inorden. Luego, encuentra el recorrido en postorden del árbol construido y lo imprime.

Definición de las clases Nodo y ArbolBinario:

Nodo: Representa un nodo en el árbol binario. Contiene un valor (val) y punteros a sus hijos izquierdo (left) y derecho (right).

ArbolBinario: Representa un árbol binario. Contiene un puntero a la raíz del árbol y métodos para construir el árbol a partir de los recorridos en preorden e inorden, así como para obtener el recorrido en postorden del árbol.

Método de construcción del árbol (**construirArbol**):

Este método construye el árbol binario de manera recursiva. Recibe los recorridos en preorden e inorden, junto con otros parámetros que indican los índices de los elementos actuales en ambos recorridos.

Utiliza un enfoque dividir y conquistar: encuentra la raíz del subárbol actual, divide los recorridos en subárboles izquierdo y derecho, y luego llama recursivamente al método para construir cada subárbol.

Constructor de ArbolBinario:

Este constructor inicializa un objeto ArbolBinario a partir de los recorridos en preorden e inorden dados. Primero, verifica si los recorridos están vacíos o si tienen diferentes longitudes. Si alguno de estos casos se cumple, establece la raíz del árbol como nula y retorna.

Luego, crea un mapeo de los valores del recorrido en inorden a sus índices para facilitar la búsqueda del índice de la raíz en el recorrido en inorden. Después, llama al método **construirArbol** para construir el árbol binario a partir de los recorridos dados.

Método recorridoPostOrden:

Este método realiza un recorrido en postorden del árbol binario. Utiliza una ***función lambda*** recursiva para realizar el recorrido.

El recorrido en postorden consiste en visitar primero el subárbol izquierdo, luego el subárbol derecho y finalmente la raíz. En este caso, el valor de cada nodo se agrega a una cadena que representa el recorrido en postorden.

Función principal main:

- Define los recorridos en preorden e inorden del árbol binario.

- Crea un objeto ArbolBinario utilizando los recorridos en preorden e inorden.

- Llama al método recorridoPostOrden para obtener el recorrido en postorden del árbol binario.

- Imprime el recorrido en postorden obtenido.

```
#include <iostream>
#include <unordered_map>
#include <functional>
using namespace std;

// Definición de un nodo en el árbol binario
class Nodo {
public:
    char val; // Valor del nodo
    Nodo* left; // Puntero al hijo izquierdo
    Nodo* right; // Puntero al hijo derecho

    // Constructor para inicializar un nodo con un valor dado y punteros a nulo
    Nodo(char x) : val(x), left(nullptr), right(nullptr) {}
};
class ArbolBinario {
private:
    Nodo* raiz; // Puntero a la raíz del árbol binario

 // Función auxiliar para construir el árbol recursivamente
      Nodo* construirArbol(const string& preOrden, const string& enOrden, int preInicio, int enInicio, int enFinal, unordered_map<char, int>& mapaIndices) {
```

```
    // Caso base: si el índice de inicio del recorrido en preorden es mayor o igual al tamaño del recorrido en preorden, o si el índice de inicio del recorrido en inorden es mayor que el índice final
    if (preInicio >= preOrden.size() || enInicio > enFinal)
        return nullptr; // Retornar un nodo nulo

    // Valor de la raíz del árbol es el primer elemento del recorrido en preorden
    char valorRaiz = preOrden[preInicio];
    // Crear un nuevo nodo con el valor de la raíz
    Nodo* nodoRaiz = new Nodo(valorRaiz);
    // Encontrar el índice de la raíz en el recorrido en inorden
    int indiceEnOrden = mapaIndices[valorRaiz];

    // Construir recursivamente el subárbol izquierdo con los elementos antes de la raíz en el recorrido en preorden y en el recorrido en inorden
        nodoRaiz->left = construirArbol(preOrden, enOrden, preInicio + 1, enInicio, indiceEnOrden - 1, mapaIndices);
    // Construir recursivamente el subárbol derecho con los elementos después de la raíz en el recorrido en preorden y en el recorrido en inorden
        nodoRaiz->right = construirArbol(preOrden, enOrden, preInicio + 1 + (indiceEnOrden - enInicio), indiceEnOrden + 1, enFinal, mapaIndices);
    return nodoRaiz; // Retornar el nodo raíz del subárbol construido
}
```

```
public:
    // Constructor que construye el árbol binario a partir de los recorridos en preorden e inorden
   ArbolBinario(const string& preOrden, const string& enOrden) {
        // Verificar si alguno de los recorridos está vacío o si tienen diferente longitud
            if (preOrden.empty() || enOrden.empty() || preOrden.size() != enOrden.size()) {
            raiz = nullptr; // Si se cumple alguna de estas condiciones, la raíz del árbol es nula
            return; // Salir del constructor
       }

       // Crear un mapeo de los valores del recorrido en inorden a sus índices
       unordered_map<char, int> mapaIndices;
       for (int i = 0; i < enOrden.size(); ++i)
           mapaIndices[enOrden[i]] = i;

       // Construir el árbol binario recursivamente
            raiz = construirArbol(preOrden, enOrden, 0, 0, enOrden.size() - 1, mapaIndices);
   }
```

```
    // Función para obtener el recorrido en postorden del árbol binario
    string recorridoPostOrden() {
        // Cadena para almacenar el recorrido en postorden
        string resultadoPostOrden = "";
        // Función recursiva para recorrer el árbol en postorden
        function<void(Nodo*)> postOrden = [&](Nodo* nodo) {
            // Si el nodo es nulo, no hay nada que hacer
            if (nodo == nullptr)
                return;
            // Recorrer recursivamente el subárbol izquierdo
            postOrden(nodo->left);
            // Recorrer recursivamente el subárbol derecho
            postOrden(nodo->right);
            // Agregar el valor del nodo al recorrido en postorden
            resultadoPostOrden += nodo->val;
        };
        // Llamar a la función de recorrido en postorden desde la raíz del árbol
        postOrden(raiz);
        // Devolver el recorrido en postorden resultante
        return resultadoPostOrden;
    }
};
```

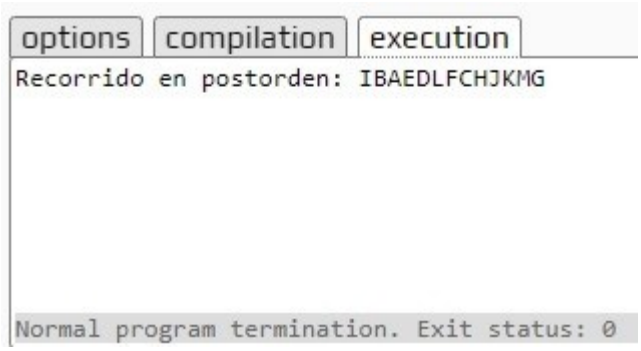
```
int main() {
    // Recorridos en preorden e inorden del árbol binario
    string preOrden = "GEAIBMCLDFKJH";
    string enOrden = "IABEGLDCFMKHJ";

    // Construir el árbol binario a partir de los recorridos dados
    ArbolBinario arbol(preOrden, enOrden);

    // Obtener el recorrido en postorden del árbol binario
    string postOrden = arbol.recorridoPostOrden();

    // Imprimir el recorrido en postorden
    cout << "Recorrido en postorden: " << postOrden << endl;

    return 0;
}
```



FIN DEL TRABAJO PRÁCTICO