9-3-2025

Damir Jiménez Arango

Actividad n° 2

Estimado alumno:

1. Elabore dos programas en C++ con dos ejemplos donde utilice árboles binarios. (50%)

Postes.cpp

#include <iostream>

using namespace std;

// Estructura para un nodo

struct Nodo {

    int valor;

    Nodo\* izquierdo;

    Nodo\* derecho;

};

// Función para crear un nodo

Nodo\* nuevoNodo(int valor) {

    Nodo\* nodo = new Nodo();

    nodo->valor = valor;

    nodo->izquierdo = nodo->derecho = nullptr;

    return nodo;

}

// Función para insertar un nodo en el árbol

Nodo\* insertar(Nodo\* raiz, int valor) {

    if (raiz == nullptr) {

        return nuevoNodo(valor);

    }

    if (valor < raiz->valor) {

        raiz->izquierdo = insertar(raiz->izquierdo, valor);

    } else {

        raiz->derecho = insertar(raiz->derecho, valor);

    }

    return raiz;

}

// Función para recorrer el árbol en orden (In-Order)

void recorridoInOrden(Nodo\* raiz) {

    if (raiz != nullptr) {

        recorridoInOrden(raiz->izquierdo);

        cout << raiz->valor << " ";

        recorridoInOrden(raiz->derecho);

    }

}

int main() {

    Nodo\* raiz = nullptr;

    // Insertando elementos

    raiz = insertar(raiz, 30);

    insertar(raiz, 20);

    insertar(raiz, 40);

    insertar(raiz, 10);

    insertar(raiz, 25);

    // Imprimir recorrido en orden

    cout << "Recorrido en orden: ";

    recorridoInOrden(raiz);  // Salida esperada: 10 20 25 30 40

    cout << endl;

    return 0;

}

**Explicación**

#include <iostream>

#include <map>

#include <vector>

using namespace std;

// Estructura para un nodo que representa un poste de electricidad

struct Poste {

int id;

int capacidad;

Poste\* izquierdo;

Poste\* derecho;

string zona; // Zona para los puntos cardinales

Poste(int id, int capacidad, string zona = "Centro") {

this->id = id;

this->capacidad = capacidad;

this->zona = zona;

izquierdo = derecho = nullptr;

}

};

// Función para insertar un poste en el árbol binario

Poste\* insertar(Poste\* raiz, int id, int capacidad, string zona = "Centro") {

if (raiz == nullptr) {

return new Poste(id, capacidad, zona);

}

if (id < raiz->id) {

// Si el ID es menor, va a la izquierda (Zona Sur u Oeste)

string nuevaZona = (raiz->zona == "Centro" || raiz->zona == "Norte") ? "Sur" : "Oeste";

raiz->izquierdo = insertar(raiz->izquierdo, id, capacidad, nuevaZona);

} else {

// Si el ID es mayor, va a la derecha (Zona Norte o Este)

string nuevaZona = (raiz->zona == "Centro" || raiz->zona == "Sur") ? "Norte" : "Este";

raiz->derecho = insertar(raiz->derecho, id, capacidad, nuevaZona);

}

return raiz;

}

// Función para imprimir el recorrido en orden de los postes

void imprimirRecorrido(Poste\* raiz) {

if (raiz != nullptr) {

imprimirRecorrido(raiz->izquierdo);

cout << "Poste ID: " << raiz->id << ", Capacidad: " << raiz->capacidad << " kW, Zona: " << raiz->zona << endl;

imprimirRecorrido(raiz->derecho);

}

}

// Función para clasificar los postes por zona (Norte, Sur, Este, Oeste)

void clasificarPorZona(Poste\* raiz, map<string, vector<int>>& zonas) {

if (raiz != nullptr) {

zonas[raiz->zona].push\_back(raiz->id);

clasificarPorZona(raiz->izquierdo, zonas);

clasificarPorZona(raiz->derecho, zonas);

}

}

int main() {

Poste\* redDePostes = nullptr;

int id, capacidad, numPostes;

cout << "Ingrese el número de postes que desea agregar: ";

cin >> numPostes;

// Insertar postes

for (int i = 0; i < numPostes; i++) {

cout << "Ingrese el ID del poste " << (i+1) << ": ";

cin >> id;

cout << "Ingrese la capacidad del poste " << (i+1) << " (en kW): ";

cin >> capacidad;

// Insertar el poste en la red

redDePostes = insertar(redDePostes, id, capacidad);

}

// Clasificar postes por zona

map<string, vector<int>> zonas;

clasificarPorZona(redDePostes, zonas);

// Imprimir los postes clasificados por zona

cout << "\nPostes clasificados por zona (Puntos Cardinales):\n";

for (const auto& zona : zonas) {

cout << zona.first << ": ";

for (int id : zona.second) {

cout << id << " ";

}

cout << endl;

}

// Imprimir el recorrido de los postes

cout << "\nRecorrido de la red de postes (en orden de ID):\n";

imprimirRecorrido(redDePostes);

return 0;

}

**Programa 1: Árbol binario con inserción y recorrido en orden**

Estructura del Nodo: Cada nodo tiene un valor y dos punteros, uno para el hijo izquierdo y otro para el hijo derecho.

Función insertar: Inserta un nuevo valor en el árbol binario manteniendo el orden.

Función recorridoInOrden: Imprime los valores de los nodos en orden ascendente (de menor a mayor).

Este programa construye un árbol binario con algunos valores y luego realiza un recorrido en orden, mostrando los valores en el orden correcto.

Postes.cpp

Explicación Resumida del Código

El programa simula una red de postes eléctricos utilizando un árbol binario de búsqueda (ABB). Cada nodo representa un poste con un ID, capacidad en kW y una zona geográfica (Norte, Sur, Este, Oeste o Centro).

**Funciones principales:**

insertar → Agrega un poste al árbol, asignándole una zona basada en su posición en el ABB.

Izquierda → Sur u Oeste

Derecha → Norte o Este

imprimirRecorrido → Muestra los postes en orden creciente de ID.

clasificarPorZona → Agrupa los postes por zona y los almacena en un mapa (map<string, vector<int>>).

main

El usuario ingresa la cantidad de postes y sus datos.

Se insertan en el árbol y se clasifican por puntos cardinales.

Se muestra la lista ordenada y la clasificación por zonas.

Este código organiza y muestra postes eléctricos con una estructura clara, permitiendo una fácil visualización de su ubicación.

2. Elabore dos programas en C++ con dos ejemplos donde utilice grafos. (50%)

Gafo.cpp

#include <iostream>

#include <list>

using namespace std;

class Grafo {

private:

    int V; // Número de vértices

    list<int>\* adj; // Lista de adyacencia

public:

    Grafo(int V); // Constructor

    void agregarArista(int v, int w); // Agregar arista

    void imprimirGrafo(); // Imprimir el grafo

};

Grafo::Grafo(int V) {

    this->V = V;

    adj = new list<int>[V]; // Crear la lista de adyacencia

}

void Grafo::agregarArista(int v, int w) {

    adj[v].push\_back(w); // Agregar w a la lista de adyacencia de v

    adj[w].push\_back(v); // Como es no dirigido, agregar v a la lista de adyacencia de w

}

void Grafo::imprimirGrafo() {

    for (int i = 0; i < V; i++) {

        cout << "Vértice " << i << ": ";

        for (int x : adj[i]) {

            cout << x << " ";

        }

        cout << endl;

    }

}

int main() {

    Grafo g(5); // Crear un grafo con 5 vértices

    g.agregarArista(0, 1);

    g.agregarArista(0, 4);

    g.agregarArista(1, 2);

    g.agregarArista(1, 3);

    g.agregarArista(1, 4);

    g.agregarArista(3, 4);

    cout << "Representación del grafo (lista de adyacencia):" << endl;

    g.imprimirGrafo();

    return 0;

}

**Resumen del Código**

Este programa implementa un grafo no dirigido usando listas de adyacencia.

Estructura:

Clase Grafo

V: Número de vértices.

adj: Lista de adyacencia.

**Métodos:**

agregarArista(v, w): Conecta v con w en ambas direcciones.

imprimirGrafo(): Muestra las conexiones de cada vértice.

**Flujo del programa:**

Se crea un grafo con 5 vértices.

Se agregan aristas (conexiones entre nodos).

Se imprime la lista de adyacencia.

Ejemplo de salida:

yaml

Copiar

Editar

Vértice 0: 1 4

Vértice 1: 0 2 3 4

Vértice 2: 1

Vértice 3: 1 4

Vértice 4: 0 1 3

Ciudad.cpp

#include <iostream>

#include <list>

#include <stack>

#include <unordered\_map>

using namespace std;

class Grafo {

private:

    unordered\_map<string, list<string>> adj; // Lista de adyacencia usando un mapa

    unordered\_map<string, bool> visitado; // Mapa para marcar los nodos visitados

public:

    void agregarRuta(const string& origen, const string& destino); // Agregar una ruta entre dos ciudades

    void imprimirRutas(); // Imprimir todas las rutas

    bool tieneRuta(const string& origen, const string& destino); // Verificar si existe una ruta

    void dfs(const string& nodo, const string& destino, bool& encontrado); // DFS para encontrar la ruta

};

void Grafo::agregarRuta(const string& origen, const string& destino) {

    adj[origen].push\_back(destino);

}

void Grafo::imprimirRutas() {

    for (const auto& ciudad : adj) {

        cout << ciudad.first << " -> ";

        for (const auto& ruta : ciudad.second) {

            cout << ruta << " ";

        }

        cout << endl;

    }

}

void Grafo::dfs(const string& nodo, const string& destino, bool& encontrado) {

    if (nodo == destino) {

        encontrado = true;

        return; // Si encontramos la ciudad de destino, terminamos

    }

    visitado[nodo] = true; // Marcamos el nodo actual como visitado

    // Recorremos todas las rutas del nodo actual

    for (const string& vecino : adj[nodo]) {

        if (!visitado[vecino]) {

            dfs(vecino, destino, encontrado);

        }

    }

}

bool Grafo::tieneRuta(const string& origen, const string& destino) {

    visitado.clear(); // Limpiar el mapa de visitados antes de comenzar el DFS

    bool encontrado = false;

    dfs(origen, destino, encontrado);

    return encontrado;

}

int main() {

    Grafo redDeRutas;

    // Agregar rutas entre ciudades

    redDeRutas.agregarRuta("CiudadA", "CiudadB");

    redDeRutas.agregarRuta("CiudadA", "CiudadC");

    redDeRutas.agregarRuta("CiudadB", "CiudadD");

    redDeRutas.agregarRuta("CiudadC", "CiudadD");

    redDeRutas.agregarRuta("CiudadD", "CiudadE");

    // Imprimir las rutas del grafo

    cout << "Rutas entre las ciudades:\n";

    redDeRutas.imprimirRutas();

    // Consultar si existe una ruta entre dos ciudades

    string origen, destino;

    cout << "\nIngrese la ciudad de origen: ";

    cin >> origen;

    cout << "Ingrese la ciudad de destino: ";

    cin >> destino;

    if (redDeRutas.tieneRuta(origen, destino)) {

        cout << "Sí, hay una ruta de " << origen << " a " << destino << ".\n";

    } else {

        cout << "No, no hay una ruta de " << origen << " a " << destino << ".\n";

    }

    return 0;

}

**Resumen del Código: Grafo de Ciudades**

Usa un grafo con lista de adyacencia (unordered\_map) para representar ciudades conectadas por rutas.

Funciones principales:

1️⃣ agregarRuta(origen, destino): Añade una conexión entre dos ciudades.

2️⃣ imprimirRutas(): Muestra todas las ciudades y sus conexiones.

3️⃣ tieneRuta(origen, destino): Usa DFS para verificar si hay un camino entre dos ciudades.

4️⃣ dfs(nodo, destino): Explora el grafo en profundidad hasta encontrar la ciudad de destino.

Ejemplo de Uso:

Conexiones: CiudadA → CiudadB → CiudadD → CiudadE

Entrada: Origen: CiudadA, Destino: CiudadE

Salida: "Sí, hay una ruta de CiudadA a CiudadE."

Aplicaciones: Modelar redes de transporte, carreteras o sistemas eléctricos.