Operaciones básicas con árboles

Bonilla Caiza David Alejandro

Andrade Chicaiza Julio Aaron

Quiroz Herrera María Laura

Tiupul Guamán Danny Alexander

Universidad de las Fuerzas Armadas ESPE

28436: Estructuras de Datos

Ing. Washington Loza H. Mgs.

10 de diciembre del 2025

Índice

Tabla de Ilustraciones

Introducción

El presente informe explica el tema de árboles, estructuras de datos muy importantes dentro de la ingeniería de software, ya que organizan la información de una manera jerárquica y eficiente, y ayuda en gran medida a la optimización de sistemas grandes el usarlos. El enfoque al que no dirigiremos será el análisis de las operaciones básicas de estas estructuras, entre estas operaciones tenemos:

- Creación
- Inserción
- Eliminación
- Recorridos Sistemáticos
- Balanceo

Esas operaciones son importantes para entender como se usar en una aplicación del mundo real. En el informe también se estudiará las características esenciales de estos, como se representan y también como se implementan en C++, las ventajas que tienen al igual que sus desventajas y adicional el grupo desarrolla un ejemplo en código para que se pueda observar correctamente el uso de este. De esta forma podremos aprender bien este tema con un desarrollo practico y simple.

Objetivos

Objetivo General

Aprender mediante una investigación las operaciones básicas que conlleva la estructura de datos llamada “Árbol”, así como si implementación en C++, y sus múltiples características.

Objetivos Específicos

Entender el concepto general de esta estructura y su definición.

Desarrollar un informe correctamente estructurado.

Realizar una correcta exposición del tema planteado en el informe.

Desarrollar un buen código con lo que se desea enseñar de forma estructurada y fácil de entender.

Desarrollo

Definiciones fundamentales

¿Qué es un Árbol?

Según Wikipedia un Árbol es “un TDA ampliamente usado que imita la estructura jerárquica de un árbol, con valor en la raíz y subárboles con un nodo padre, representado como un conjunto de nodos enlazados”. (Anónimo, 2025)

En palabras más simples un árbol es una estructura de datos, esta se compone de padres he hijos, diferentes niveles y con ellos un orden de jerarquía. Sirve principalmente para ordenar datos de forma más directa y fácil de ubicar.

Elementos de un Árbol

Realizando una investigación por la web, se encontró que en la página web VitaDemy Global se destacan principalmente los siguientes elementos de un árbol como los más importantes:

Node. - Unidad fundamental que contiene datos y enlaces

Root. - El nodo superior sin padre

Child. - Un nodo conectado directamente debajo de otro nodo

Parent. - El nodo directamente encima de un nodo secundario

Leaf. - Un nodo sin hijos

Edge. - Conexión entre dos nodos

Subtree. - Un nodo y todos sus descendientes

Depth. - Número de aristas desde la raíz hasta el nodo

Height. - Longitud del camino más largo desde un nodo hasta una hoja

Degree. - Número de hijos de un nodo (Anónimo, Tree, 2025)

¿Por qué usar Árboles?

Luego de analizar un poco la información encontrada puedo acotar lo siguiente, el usar árboles nos ayuda mucho cuando se necesita jerarquías (muy usado en este aspecto), o estructuras basadas en niveles, por ejemplo, pueden ser carpetas de archivos de un sistema, organización de empresas, o un ejemplo mucho más gráfico, árboles genealógicos. Además, es una estructura muy dinámica, puede crecer, o reducirse según las operaciones que realice el usuario, incluso optimiza mucho la búsqueda, inserción y eliminación, vuelve mucho mas eficiente estas acciones.

Representaciones gráficas

Los arboles se pueden representan visualmente de varias formas, entre estas tenemos:

Árbol *binario* balanceado

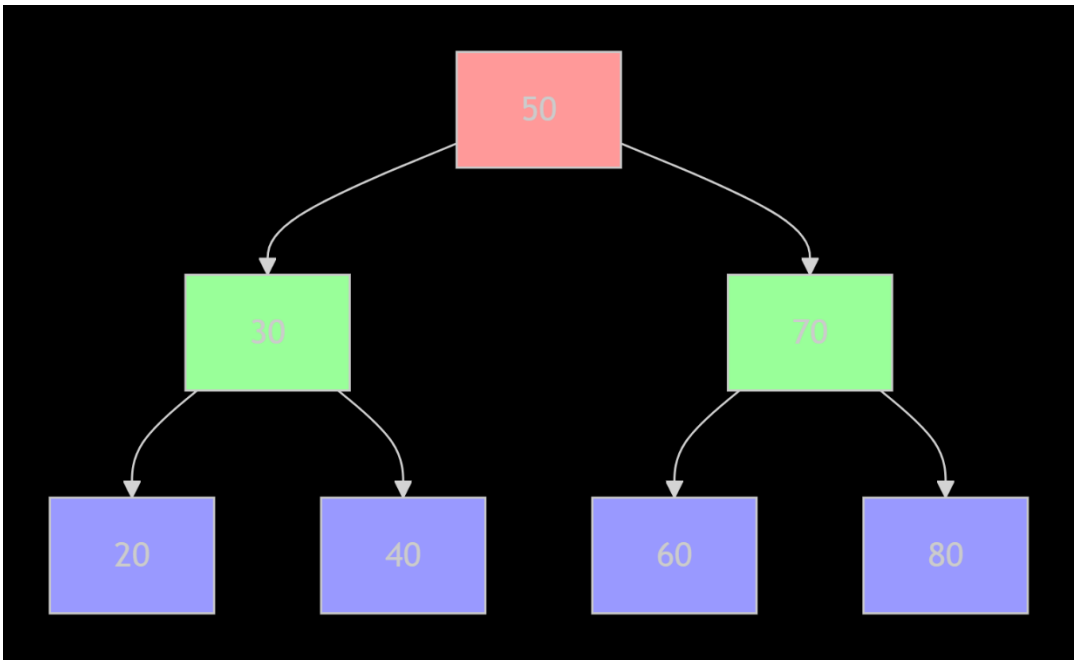

Ilustración 1. Árbol de Binario Balanceado

Árbol Binario No Balanceado

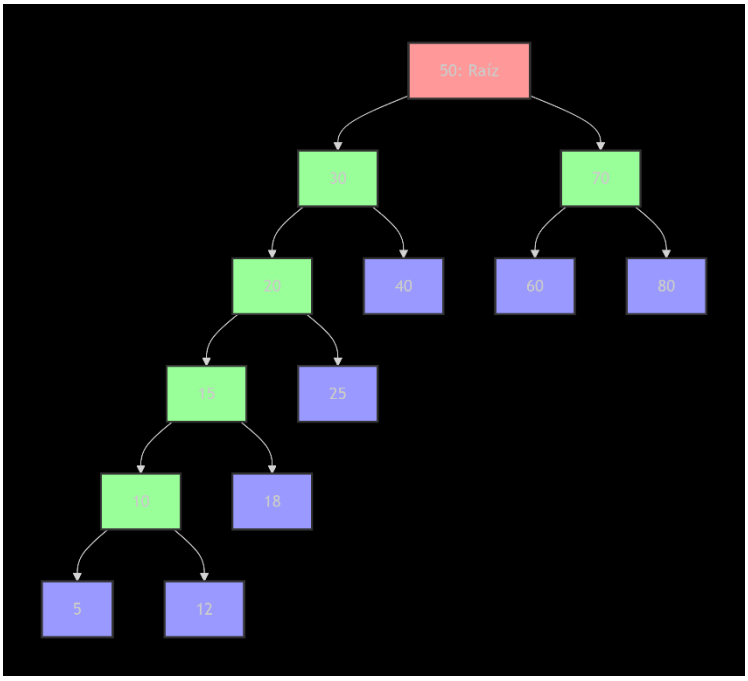

Ilustración 2. Árbol Binario No Blanceado

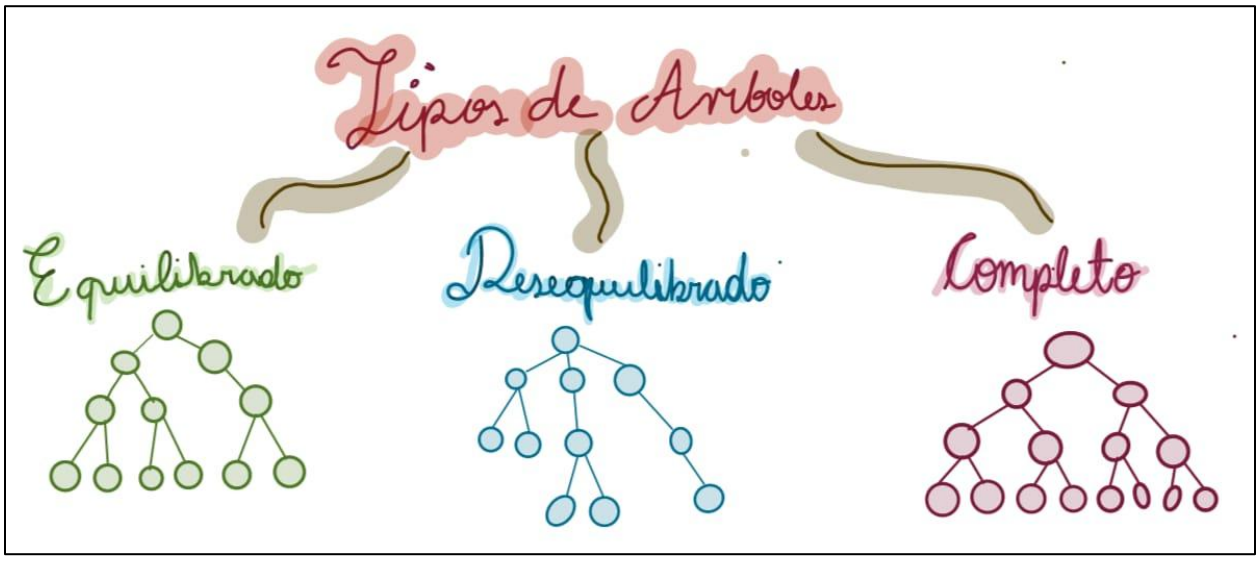

Ilustración 3. Tipos de Arboles

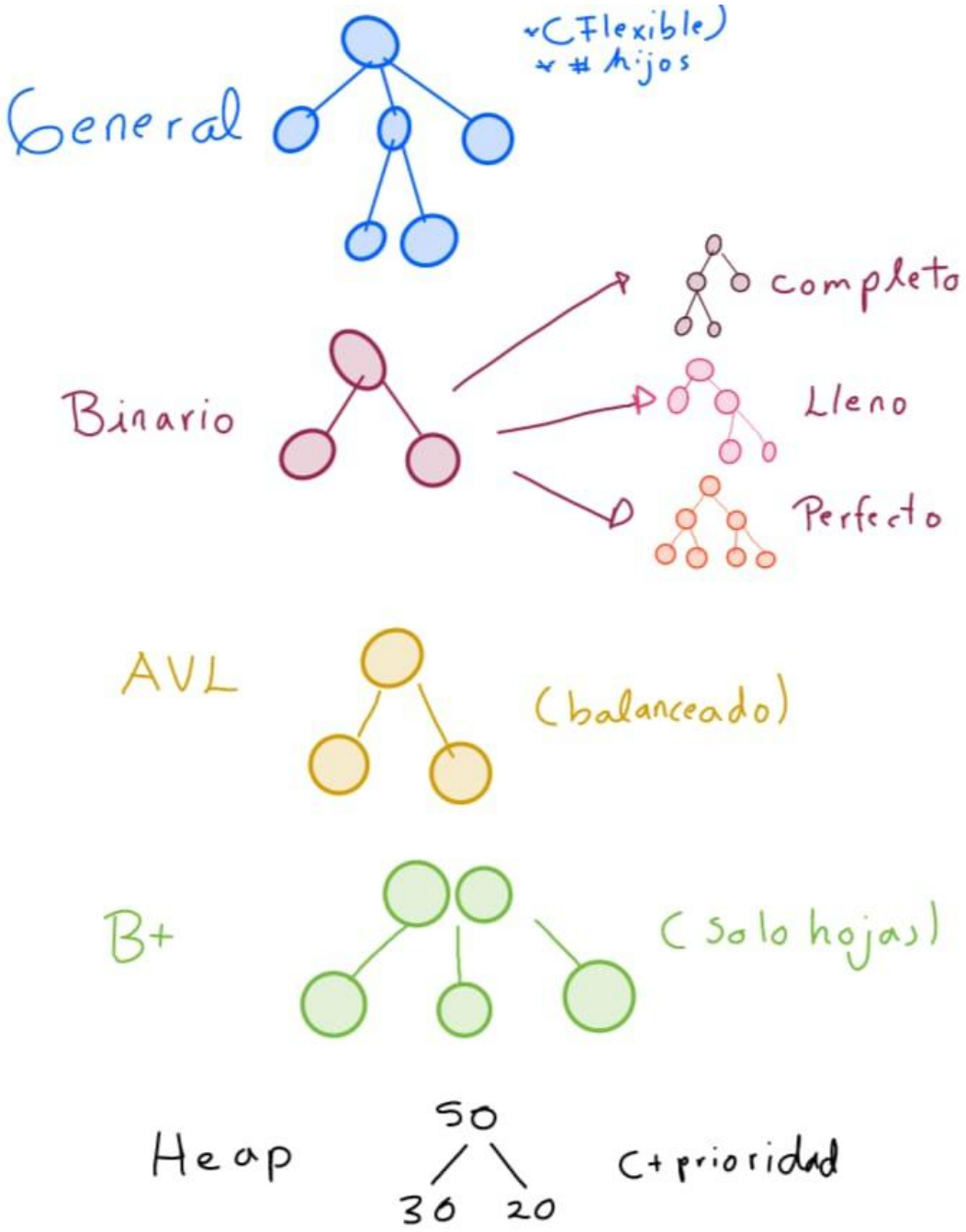

Ilustración 4. Tipos de Arboles

Comparación

Característica	Árbol Básico	Árbol Balanceado
Estructura	No tiene control de equilibrio	Con equilibrio automático
Altura	Se puede crear hasta n numero	Se mantiene en O(log n)
Velocidad de búsqueda	Se puede degradarse a o(n)	Se mantiene en O(log n)
Velocidad de inserción	Rápido	Se mantiene en O(log n)
Velocidad de eliminación	Puede causar desbalance	Se autobalancea al eliminar
Complejidad de implementación	Simple	compleja
Uso	Pequeños datos	Bases de datos
Riego de daño	Alto	Bajo
Consumo de memoria	Menor	Mayor(mayor datos)
Riesgo en transformarse en lista	Alto	Muy bajo

Tabla 1. Comparación Arboles Básicos VS Arboles Balanceados

Sintaxis y estructura en C++

Estructura Básica de un Nodo

```
struct Nodo {
    int dato;          // Valor almacenado en el nodo
    Nodo* izq;        // Puntero al hijo izquierdo
    Nodo* der;        // Puntero al hijo derecho

    // Constructor
    Nodo(int valor) {
        dato = valor;
        izq = nullptr;  // Inicialmente sin hijos
        der = nullptr;
    }
};
```

Clase ArbolBinario – Estructura Principal

```
class ArbolBinario {
private:
    Nodo* raiz;  // Puntero al nodo raíz del árbol

    // Funciones auxiliares privadas (recursivas)
    Nodo* insertarRec(Nodo* nodo, int valor);
    Nodo* eliminarRec(Nodo* nodo, int valor);
    Nodo* minimo(Nodo* nodo);
    void inorden(Nodo* nodo);
    void preorden(Nodo* nodo);
    void postorden(Nodo* nodo);
    int altura(Nodo* nodo);

public:
    // Constructor
    ArbolBinario() {
        raiz = nullptr;  // Árbol vacío al inicio
    }

    // Métodos públicos
    void insertar(int valor);
    void eliminar(int valor);
    void mostrarInorden();
    void mostrarPreorden();
    void mostrarPostorden();
    void mostrarNiveles();
    void obtenerBalance();
};
```

Explicación Simple del Código:

- Nodo: Unidad básica que almacena un valor y referencias a sus hijos.
- Raíz: Puntero principal que referencia al primer nodo del árbol.
- Funciones Recursivas: Se implementan de forma recursiva debido a la naturaleza jerárquica del árbol.
- Modificadores de Acceso:
  - private: Funciones internas de manejo de nodos.
  - public: Interfaz para el usuario final.

Operaciones básicas: creación, inserción, eliminación, recorridos y balanceo

Creación

La creación de un árbol binario inicia con un nodo raíz. Se comienza con un árbol vacío (raíz = nullptr) y se insertan nodos según sea necesario. La creación establece la estructura jerárquica inicial donde cada nuevo nodo se posiciona respetando la propiedad del árbol binario de búsqueda: valores menores a la izquierda, mayores a la derecha.

Inserción

La inserción añade nuevos nodos manteniendo la propiedad del árbol. Para insertar un valor:

- Si el árbol está vacío, se crea el nodo raíz
- Si el valor es menor que el nodo actual, se avanza al subárbol izquierdo
- Si el valor es mayor, se avanza al subárbol derecho
- Se repite hasta encontrar una posición vacía donde insertar el nuevo nodo

Eliminación

La eliminación de nodos considera tres casos:

- **Caso 1**: Nodo hoja (sin hijos) → Se elimina directamente
- **Caso 2**: Nodo con un hijo → El hijo reemplaza al nodo eliminado
- **Caso 3**: Nodo con dos hijos → Se encuentra el sucesor inorder (mínimo del subárbol derecho), se copia su valor al nodo a eliminar, y luego se elimina el sucesor

Recorridos Sistemáticos

- **Preorden**: Visita raíz → subárbol izquierdo → subárbol derecho
- **Inorden**: Visita subárbol izquierdo → raíz → subárbol derecho (produce secuencia ordenada)
- **Postorden**: Visita subárbol izquierdo → subárbol derecho → raíz
- **Por niveles**: Visita nodos nivel por nivel, de izquierda a derecha

Balanceo

El balanceo mantiene la diferencia de altura entre subárboles izquierdo y derecho en máximo 1. Se logra mediante rotaciones:

- **Rotación simple derecha**: Para desbalance izquierda-izquierda
- **Rotación simple izquierda**: Para desbalance derecha-derecha

- **Rotaciones dobles**: Para casos izquierda-derecha o derecha-izquierda

Ventajas y desventajas

Ventajas

- **Búsqueda, inserción y eliminación eficientes (arboles balanceados):** Cuando se trabaja con árboles balanceados se conserva el orden de complejidad O(log n) lo cual lo convierte en una estructura eficiente en comparación a otras a la hora de trabajar con grandes volúmenes de datos.

- Dinamicidad de tamaño: **Los árboles nos permiten crearlos o reducirlos (inserción, eliminación de nodos) de forma dinámica a diferencia de las estructuras de datos estáticas como los array cuyo tamaño es fijo.**

- Facilidad para operaciones complejas y/o recursivas: **Por la naturaleza recursiva de este tipo de estructura operaciones como: corridos, búsquedas, ordenamientos, operaciones con subárboles pueden ser implementadas de forma clara y elegante.**

Desventajas

- Sobrecarga de memoria: **Cuando se está trabajando con nodos cuyos punteros apuntan hacia sus nodos hijos existe un mayor consumo de memoria en comparación a otro tipo de estructuras más simples como los arrays o las listas enlazadas.**

- Rendimiento degradado: **Esto sucede si un árbol no está balanceado y se requiere hacer alguna acción como insertar o eliminar, su complejidad original es O(log n) sin embargo al no estar balanceado en el peor de los casos esta se puede convertir en O(n).**

- No siempre es la mejor opción: **Si un problema requiere una estructura que tenga acceso directo o datos estáticos lo más adecuado que se puede hacer es usar otro tipo de estructura de datos como arrays o tablas hash las cuales resultan ser más eficientes que el árbol.**

Limitaciones y cuándo no usarlos.

- Consumo adicional de memoria: **cada nodo almacena referencias ya sea de sus hijos, padre o ambos lo cual afecta al consumo de memoria cuando un árbol es muy grande.**

- Complejidad de implementación y mantenimiento: **Cuando se trabaja con árboles balanceados se requiere un proceso extra para realizar los algoritmos de balanceo, rotación, mantenimiento de invariantes, etc. Lo cual a la larga hace más difícil su mantenimiento aumentando la posibilidad de errores en el software.**

- **Degradación de rendimiento si no están balanceados:** En arboles de búsqueda binaria si los datos se insertan en un orden desfavorable el árbol terminaría luciendo como una lista enlazada lo cual convierte su orden de complejidad de O(log n) a O(n) haciéndolo menos eficiente.
- **No son ideales si no necesitas orden ni jerarquía**: si los datos con los que estás trabajando no requieren orden natural ni consultas ordenadas de alguna forma en particular o rangos, usar la estructura de árbol se convierte en algo innecesario.
- **Posible ineficiencia en algunos tipos de consultas**: ciertos tipos de consultas, por ejemplo: acceso por índice, búsquedas directas sin orden y búsquedas por clave arbitraria pueden ser más rápidos con otras estructuras.

Código desarrollado por el grupo

```
#include <iostream>
#include <queue>
#include <algorithm>
using namespace std;

class ArbolBinario {
private:
    // Estructura interna para representar un nodo
    struct Nodo {
        int dato;
        Nodo* izq;
        Nodo* der;

        Nodo(int valor) : dato(valor), izq(nullptr), der(nullptr) {}
    };

    Nodo* raiz;  // Punto de entrada al árbol

    // --- FUNCIONES AUXILIARES PRIVADAS ---

    // Inserta un valor manteniendo la propiedad ABB
    Nodo* insertarRec(Nodo* nodo, int valor) {
        if (!nodo) return new Nodo(valor);  // Caso base: crear nodo

        // Propiedad ABB: izquierda < raíz < derecha
        if (valor < nodo->dato)
            nodo->izq = insertarRec(nodo->izq, valor);
        else if (valor > nodo->dato)
            nodo->der = insertarRec(nodo->der, valor);

        return nodo;  // Retorna el nodo (sin cambios si ya existe)
    }

    // Encuentra el nodo con valor mínimo (más a la izquierda)
    Nodo* minimo(Nodo* nodo) {
        while (nodo && nodo->izq)
            nodo = nodo->izq;
        return nodo;
    }

    // Elimina un nodo específico
    Nodo* eliminarRec(Nodo* nodo, int valor) {
        if (!nodo) return nullptr;  // Valor no encontrado

        // Buscar el nodo a eliminar
        if (valor < nodo->dato)
            nodo->izq = eliminarRec(nodo->izq, valor);
        else if (valor > nodo->dato)
            nodo->der = eliminarRec(nodo->der, valor);
        else {
            // CASO 1: Nodo sin hijos (hoja)
            if (!nodo->izq && !nodo->der) {
                delete nodo;
                return nullptr;
            }
            // CASO 2: Nodo con un hijo
            else if (!nodo->izq || !nodo->der) {
                Nodo* temp = nodo->izq ? nodo->izq : nodo->der;
```

```
                delete nodo;
                return temp;
            }
            // CASO 3: Nodo con dos hijos
            else {
                // Encontrar sucesor inorder (mínimo del subárbol derecho)
                Nodo* sucesor = minimo(nodo->der);
                nodo->dato = sucesor->dato;  // Copiar valor
                nodo->der = eliminarRec(nodo->der, sucesor->dato);
            }
        }
        return nodo;
    }

    // --- RECORRIDOS RECURSIVOS ---

    // Inorden: Izquierda - Raíz - Derecha (resultado ordenado)
    void inorden(Nodo* nodo) {
        if (!nodo) return;
        inorden(nodo->izq);
        cout << nodo->dato << " ";
        inorden(nodo->der);
    }

    // Preorden: Raíz - Izquierda - Derecha
    void preorden(Nodo* nodo) {
        if (!nodo) return;
        cout << nodo->dato << " ";
        preorden(nodo->izq);
        preorden(nodo->der);
    }

    // Postorden: Izquierda - Derecha - Raíz
    void postorden(Nodo* nodo) {
        if (!nodo) return;
        postorden(nodo->izq);
        postorden(nodo->der);
        cout << nodo->dato << " ";
    }

    // Calcula la altura del árbol/subárbol
    int altura(Nodo* nodo) {
        if (!nodo) return 0;
        return 1 + max(altura(nodo->izq), altura(nodo->der));
    }

public:
    // Constructor
    ArbolBinario() : raiz(nullptr) {}

    // --- INTERFAZ PÚBLICA ---

    void insertar(int valor) {
        raiz = insertarRec(raiz, valor);
        cout << "Valor " << valor << " insertado.\n";
    }

    void eliminar(int valor) {
        raiz = eliminarRec(raiz, valor);
        cout << "Valor " << valor << " eliminado.\n";
```

```
    }

    void mostrarInorden() {
        cout << "Recorrido Inorden: ";
        inorden(raiz);
        cout << endl;
    }

    void mostrarPreorden() {
        cout << "Recorrido Preorden: ";
        preorden(raiz);
        cout << endl;
    }

    void mostrarPostorden() {
        cout << "Recorrido Postorden: ";
        postorden(raiz);
        cout << endl;
    }

    // Recorrido por niveles (BFS)
    void mostrarNiveles() {
        if (!raiz) {
            cout << "Árbol vacío.\n";
            return;
        }

        cout << "Recorrido por Niveles: ";
        queue<Nodo*> cola;
        cola.push(raiz);

        while (!cola.empty()) {
            Nodo* actual = cola.front();
            cola.pop();

            cout << actual->dato << " ";

            if (actual->izq) cola.push(actual->izq);
            if (actual->der) cola.push(actual->der);
        }
        cout << endl;
    }

    // Calcula factor de balance simple
    void obtenerBalance() {
        if (!raiz) {
            cout << "Árbol vacío.\n";
            return;
        }

        int alturaIzq = altura(raiz->izq);
        int alturaDer = altura(raiz->der);
        int factor = alturaIzq - alturaDer;

        cout << "Factor de Balance: " << factor;
        cout << " (Izq: " << alturaIzq << ", Der: " << alturaDer << ")\n";

        if (abs(factor) > 1)
            cout << "ADVERTENCIA: Árbol desbalanceado.\n";
        else
```

```
            cout << "Árbol relativamente balanceado.\n";
    }
};

// --- FUNCIÓN PRINCIPAL DEMOSTRATIVA ---
int main() {
    ArbolBinario arbol;

    cout << "=== DEMOSTRACIÓN DE ÁRBOL BINARIO DE BÚSQUEDA ===\n\n";

    // 1. CREACIÓN E INSERCIÓN
    cout << "1. Insertando valores: 50, 30, 70, 20, 40, 60, 80\n";
    int valores[] = {50, 30, 70, 20, 40, 60, 80};
    for (int v : valores) arbol.insertar(v);
    cout << endl;

    // 2. RECORRIDOS
    cout << "2. Recorridos del árbol:\n";
    arbol.mostrarInorden();    // Muestra en orden ascendente
    arbol.mostrarPreorden();   // Muestra en orden raíz primero
    arbol.mostrarPostorden();  // Muestra en orden raíz último
    arbol.mostrarNiveles();    // Muestra por niveles
    cout << endl;

    // 3. ANÁLISIS DE BALANCE
    cout << "3. Análisis de balance inicial:\n";
    arbol.obtenerBalance();
    cout << endl;

    // 4. ELIMINACIÓN
    cout << "4. Eliminando nodos: 20, 30, 50\n";
    arbol.eliminar(20);
    arbol.eliminar(30);
    arbol.eliminar(50);
    cout << endl;

    // 5. ESTADO FINAL
    cout << "5. Estado final del árbol:\n";
    arbol.mostrarInorden();
    arbol.mostrarNiveles();
    arbol.obtenerBalance();

    cout << "\n=== FIN DE LA DEMOSTRACIÓN ===\n";

    return 0;
}
```

Aplicación práctica o caso de uso

Un ejemplo en donde los árboles son muy útiles es al momento de organizar archivos de un sistema operativo. Pongamos un contexto más gráfico, supongamos que queremos representar carpetas como nodos padres, estos contienen los archivos o subcarpetas (nodos hijos). En este caso los datos de los archivos se vendrían a observar como nodos hoja (sin hijos), y la raíz de este árbol vendría a representar el directorio principal del sistema.

Gracias a manejar esta estructura el sistema puede organizar jerárquicamente los archivos y carpetas, lo que mantiene mas ordenado todo el sistema, además las búsquedas en el mismo serán mucho más rápidas gracias a que se ubica rápidamente en rutas, el insertar o eliminar archivos de forma dinámica adaptándose así al usuario también será mucho más rápido gracias a usar esta estructura.

Conclusiones

Para concluir, en este informe se ha analizado la estructura de datos conocida como árbol, se explicó su orden de complejidad, su sintaxis, elementos que componen el árbol ((nodos, raíz, hijos, hojas, ramas, niveles, profundidad/altura, etc.)), operaciones básicas que puede realizar (crear, insertar y eliminar) y su implementación practica en C++ con un ejemplo de la vida real.

Ademas, los árboles son estructuras jerárquicas fundamentales que optimizan operaciones como búsqueda, inserción y eliminación cuando están balanceados, ofreciendo complejidad O(log n). Sin embargo, requieren gestión de memoria y balanceo para evitar degradación del rendimiento.

Y su aplicación es ideal en escenarios que demandan orden y jerarquía, como sistemas de archivos o bases de datos. Para casos que no requieren relaciones jerárquicas, otras estructuras como arrays o tablas hash pueden resultar más eficientes y sencillas de implementar.

Bibliografía

Anónimo. (10 de Marzo de 2025). *Árbol (informática)*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/%C3%81rbol_%28inform%C3%A1tica%29

Anónimo. (2025). *Tree*. Obtenido de VitaDemy: https://vitademyglobal.com/computer-science/tree/

Anónimo. (23 de Julio de 2025). Advantages and Disadvantages of Tree*(informática)*. Obtenido de GeeksforGeeks: https://es.wikipedia.org/wiki/%C3%81rbol_%28inform%C3%A1tica%29