Tema 1: Punteros

Puntero: variable que almacena la dirección de memoria de otra variable. *Agustín sabe de Tomás, y es Tomás el que sabe el número.*

Son esenciales porque nos permiten trabajar con memoría dinámica y estructuras de datos más avanzadas. También ahorra espacio, no tenes en memoria el archivo, solo la dirección del archivo.

Declaramos los punteros usando el operador `*`. Por ej, declarar un puntero a un entero: `int* punteroEntero;` En vez de declarar la variable, en algún lugar declaras la variable y en otro queda declarado el puntero a esa variable. Para acceder a un dato de la variable, utilizamos `->`. No se escribe `alumno.nombre`, sino se escribe `alumno->nombre`. Si cambias el dato desde la variable y le pedís al puntero que te lo muestre, te mostrará el cambiado.

La **memoria dinámica** nos permite crear y liberar memoria en tiempo de ejecución. Es necesario cuando no conocemos el tamaño de los datos con anticipación o cuando queremos utilizar datos que persistan más allá de la función que los crea.

Para asignar memoria dinámica usamos `new`, para liberarla `delete`. Hay que borrarla manual porque el sistema operativo ahora no se encarga, solo borra el puntero, no el pedazo de memoria que él apunta.

Cada vez que se inicializa el programa de nuevo, se le asigna una diferente posición de memoria a la información que apunta el puntero.

```
#include <iostream>
#include <string>

// Definición del struct Alumno

struct Alumno {
  string nombre;
  int edad;
  };

int main() {
    Alumno* alumnoDinamico; // Declaración de un puntero a un Alumno
    alumnoDinamico = new Alumno;  // Asignación dinámica de memoria para un
A                                    Alumno

   // Asignar valores al Alumno dinámico
    alumnoDinamico->nombre = "Juan";
    alumnoDinamico->edad = 20;

// Imprimir los valores del Alumno dinámico
  cout << "Nombre del Alumno: " << alumnoDinamico->nombre << endl;
  cout << "Edad del Alumno: " << alumnoDinamico->edad << " años" << endl;
    delete alumnoDinamico; // Liberar la memoria asignada dinámicamente
```

```
return 0;
}
```

Operadores de Dirección y Desreferenciación

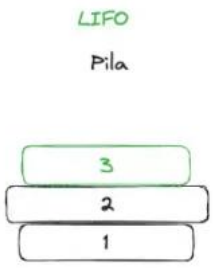
`&` : se utiliza para obtener la dirección de memoria de una variable.
`*` : acceder al valor almacenado en una dirección de memoria.

```
#include <iostream>
#include <string>

struct Alumno {
  string nombre;
  int edad;
};

int main() { // Declaración de un objeto Alumno y un puntero
  Alumno juan;
  juan.nombre = "Juan";
  juan.edad = 20;

  Alumno* punteroJuan = &juan;

   // Imprimimos la dirección de memoria y los valores almacenados en el objeto Alumno
  cout << "Dirección de memoria del objeto Alumno 'juan': " << punteroJuan << endl;
  cout << "Nombre del Alumno: " << punteroJuan->nombre << endl;
  cout << "Edad del Alumno: " << punteroJuan->edad << " años" << endl;

  delete punteroJuan;
return 0;
}
```

Tema 2: Estructuras Enlazadas

Son sucesiones de punteros.

Estructuras Lineales: los elementos ocupan lugares sucesivos en la estructura y cada uno de ellos tiene un sucesor y un predecesor. Sus elementos están ubicados uno al lado del otro, relacionados en forma lineal.



Estructura de Tipo Pila: sigue el principio LIFO ("Último en entrar, primero en salir"). Por ej: pila de libros. Se borra al leer, puede no hacerse pero no es la idea. El nuevo ingresado se acuerda del anterior, el tercero se acuerda del segundo, el segundo del primero, etc. El puntero apunta al último ingresado. Utilizamos `pop` al ingresar un elemento y `push` al sacarlo.

La idea es crear un push y pop original, e ir utilizándolos en otras funciones.

```
#include <iostream>

struct Nodo {  // Definición de la estructura Nodo
  int dato;
  Nodo* siguiente; // puntero a otro objeto de tipo nodo
};

Nodo* crearNodo(int valor) { // Función para crear un nuevo nodo
  Nodo* nuevoNodo = new Nodo;
  nuevoNodo->dato = valor;
  nuevoNodo->siguiente = nullptr;
  return nuevoNodo;
}

void push(Nodo*& pila, int valor) {  // Función para insertar un elemento
                                        en la pila
  Nodo* nuevoNodo = crearNodo(valor);
  nuevoNodo->siguiente = pila;
  pila = nuevoNodo;
}
int pop(Nodo*& pila) {  // Función para eliminar y obtener el elemento en
                           la cima de la pila
  if (pila == nullptr) {
      cout << "La pila está vacía." << endl;
      return -1; // Valor de error
   }

  int valor = pila->dato;
  Nodo* temp = pila;
  pila = pila->siguiente;
  delete temp;
  return valor;
}

bool isEmpty( Nodo* pila) {     // Función para verificar si la pila está
                                   vacía
   return pila == nullptr;
}

int main() {
```

```
  Nodo* pila = nullptr; // Inicializar la pila

  // Apilar elementos en la pila
  push(pila, 10);
  push(pila, 20);
  push(pila, 30);

  cout << "Elementos en la pila:" << endl;
  while (!isEmpty(pila)) {

       // Mostrar y eliminar el elemento en la cima de la pila
        int valor = pop(pila);
        cout << valor << " ";
   }
  cout << endl;
  return 0;
}
```

Para Push: `Nodo*& pila` es una referencia a un puntero a un Nodo. Permite que la función modifique el puntero pila original, lo que es necesario para agregar un nuevo elemento a la pila. El segundo parámetro, `int valor`, es el valor que deseas agregar a la pila.

`nuevoNodo->siguiente = pila;`: El puntero siguiente del nuevo Nodo se establece para que apunte al mismo Nodo al que apunta actualmente la pila. El nuevo Nodo se convierte en el nuevo elemento en la parte superior de la pila y su siguiente apunte al elemento anterior de la pila.

`pila = nuevoNodo`;: Finalmente, el puntero pila se actualiza para que apunte al nuevo Nodo que has creado. Esto significa que el nuevo Nodo ahora se encuentra en la parte superior de la pila y se ha convertido en el elemento más reciente agregado.

Para Pop: `if (pila == nullptr)`: verifica si el puntero pila es igual a `nullptr`, lo que significa que la pila no apunta a ningún Nodo. En otras palabras, esta condición comprueba si la pila está vacía y no contiene elementos.

Si quiero mostrar algo pero no consumirlo, solo le pregunto a la pila cuál es el siguiente.

```
void mostrar(Nodo* pila) {
  string proxima = pila -> nombre;
  cout << proxima << endl;
}
```

Si quiero invertir la pila:

```
void invertir(Nodo* pila) {
  Nodo* pilaAux = new pila;
  pila = null;
  while(pilaAux != null) {
       string valor = pop(pilaAux);
  push(pila,valor
  }
```

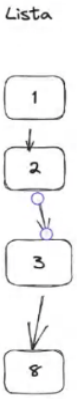
```
  pila = pilaAux;
}
```

Estructura Tipo Cola: sigue el principio FIFO ("Primero en entrar, primero en salir"). Por ej: cola de supermercado. El primero que ingresa se acuerda del segundo que ingresa, el puntero inicial apunta al primer ingresado. Se borra al leer.

```
#include <iostream>

struct Nodo {
  int dato;
  Nodo* siguiente;
};

Nodo* crearNodo(int valor) {
  Nodo* nuevoNodo = new Nodo;
  nuevoNodo->dato = valor;
  nuevoNodo->siguiente = nullptr;
  return nuevoNodo;
}

// Función para insertar un elemento en la cola
void encolar(Nodo*& frente, Nodo*& final, int valor) {
  Nodo* nuevoNodo = crearNodo(valor);
  if (final == nullptr) {
        frente = final = nuevoNodo;
  } else {
        final->siguiente = nuevoNodo;
        final = nuevoNodo;
    }
}

// Función para eliminar y obtener el elemento del frente de la cola
int desencolar(Nodo*& frente, Nodo*& final) {
  if (frente == nullptr) {
        cerr << "La cola está vacía." << endl;
        return -1; // Valor de error
   }

  int valor = frente->dato;
  Nodo* temp = frente;
  frente = frente->siguiente;
  if (frente == nullptr) {
        final = nullptr; // Si se elimina el último elemento, actualizar
'final'
```

```
    }
  delete temp;
  return valor;
}

// Función para verificar si la cola está vacía
bool isEmpty(Nodo* frente) {
   return frente == nullptr;
}

int main() {
  Nodo* frente = nullptr;
  Nodo* final = nullptr;
  // Encolar elementos
  encolar(frente, final, 10);
  encolar(frente, final, 20);
  encolar(frente, final, 30);

  cout << "Elementos en la cola:" << endl;
  while (!isEmpty(frente)) {
       // Desencolar y mostrar el elemento del frente de la cola
       int valor = desencolar(frente, final);
       cout << valor << " ";
   }
  cout << endl;
  return 0;
}
```

En resumen, la función `encolar` agrega un nuevo elemento al final de una cola implementada como una lista enlazada simple. Si la cola está vacía, crea el nuevo Nodo y hace que tanto `frente` como `final` apunten a él. Si la cola no está vacía, conecta el nuevo Nodo al final de la cola y actualiza `final` para que apunte al nuevo elemento, manteniendo así la estructura de la cola.



Estructura Tipo Lista: nodos que se conectan unos a otros para formar una secuencia. Cada elemento conoce al siguiente en la lista. Puede o no estar ordenada, no se borra al leer. Es más difícil de codear, pero también menos limitada. Sus diferencias con los arrays:

→ Para los arrays es necesario saber el valor de antemano y no se pueden ir agregando valores luego, en la lista siempre se pueden agregar.
→ Una lista ocupa más lugar, ya que es el valor del dato y el peso del puntero del siguiente.

→ En el array es mucho más rápido encontrar una posición en específico, en la lista se debe ir uno por uno.

```
#include <iostream>

struct Nodo {
  int dato;
  Nodo* siguiente;
};

Nodo* crearNodo(int valor) {
  Nodo* nuevoNodo = new Nodo;
  nuevoNodo->dato = valor;
  nuevoNodo->siguiente = nullptr;
  return nuevoNodo;
}

// Función para imprimir una lista enlazada simple
void imprimirListaSimple(Nodo* inicio) {
  Nodo* actual = inicio;
  while (actual != nullptr) {
       cout << actual->dato << " ";
       actual = actual->siguiente;
    }

   cout << endl;
}

int main() {
  // Crear una lista enlazada simple
  Nodo* listaSimple = crearNodo(10);
  listaSimple->siguiente = crearNodo(20);
  listaSimple->siguiente->siguiente = crearNodo(30);

  // Imprimir la lista enlazada simple
  cout << "Lista Simple: ";
  imprimirListaSimple(listaSimple);

  // Liberar la memoria al final del programa
  delete listaSimple->siguiente->siguiente;
  delete listaSimple->siguiente;
  delete listaSimple;
return 0;
}
```

Patrones de Operación en Listas

→ **Carga sin restricciones:** agregar nodos al final de la lista de acuerdo con la secuencia en la que se ingresaron los elementos.

```
Nodo* cargarLista(Nodo* inicio, int valor) {
  Nodo* nuevoNodo = crearNodo(valor);
  if (inicio == nullptr) {
       inicio = nuevoNodo;
  } else {
       Nodo* actual = inicio;
       while (actual->siguiente != nullptr) {
            actual = actual->siguiente;
        }
       actual->siguiente = nuevoNodo;
  }
   return inicio;
}
```

→ **Carga sin repetir:** verificar si el elemento ya existe antes de agregarlo a la lista.

```
bool existeElemento(Nodo* inicio, int valor) {
  Nodo* actual = inicio;
  while (actual != nullptr) {
       if (actual->dato == valor) {
            return true;
}
       actual = actual->siguiente;
  }
   return false;
}

Nodo* cargarSinRepetir(Nodo* inicio, int valor) {
  if (!existeElemento(inicio, valor)) {
       return cargarLista(inicio, valor);
    }

   return inicio;
}
```

→ **Búsqueda:** implica recorrer la lista hasta encontrar el valor deseado.

```
Nodo* buscarElemento(Nodo* inicio, int valor) {
  Nodo* actual = inicio;
  while (actual != nullptr) {
       if (actual->dato == valor) {
            return actual; // Devuelve el nodo que contiene el valor
}
       actual = actual->siguiente;
```

```
    }
   return nullptr; // El valor no se encontró en la lista
}
```

→ **Recorrido:** visitar cada elemento de la lista y realizar alguna acción, como mostrar su valor.

```
void recorrerLista(Nodo* inicio) {
  Nodo* actual = inicio;
  while (actual != nullptr) {
       cout << actual->dato << " ";
       actual = actual->siguiente;
}
```

→ **Eliminación de nodos:** eliminar un elemento específico de la lista.

```
Nodo* eliminarNodo(Nodo* inicio, int valor) {
  if (inicio == nullptr) {
       return nullptr; // Lista vacía, no hay nada que eliminar
    }

  if (inicio->dato == valor) {
       Nodo* temp = inicio;
       inicio = inicio->siguiente;
       delete temp;
       return inicio; // Devuelve la lista actualizada
  }
  Nodo* actual = inicio;
  while (actual->siguiente != nullptr) {
       if (actual->siguiente->dato == valor) {
            Nodo* temp = actual->siguiente;
            actual->siguiente = actual->siguiente->siguiente;
            delete temp;
            return inicio; // Devuelve la lista actualizada
        }

       actual = actual->siguiente;
  }
   return inicio; // Valor no encontrado en la lista
}
```

→ **Insertar ordenado:**

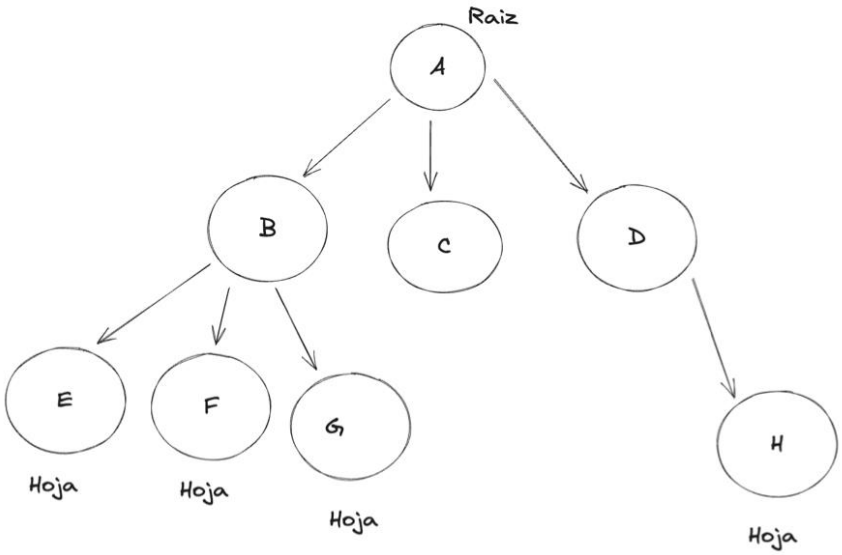
```
// Función para insertar un elemento de manera ordenada en la lista
Nodo* insertarOrdenado(Nodo* inicio, int valor) {
   Nodo* nuevoNodo = crearNodo(valor);

   // Caso especial: si la lista está vacía o el valor es menor que el
primer elemento
  if (inicio == nullptr || valor < inicio->dato) {
       nuevoNodo->siguiente = inicio;
```

```
        inicio = nuevoNodo;
        return inicio;
}
  Nodo* actual = inicio;
  while (actual->siguiente != nullptr && actual->siguiente->dato < valor) {
        actual = actual->siguiente;
  }
  nuevoNodo->siguiente = actual->siguiente;
  actual->siguiente = nuevoNodo;
  return inicio;
}
```

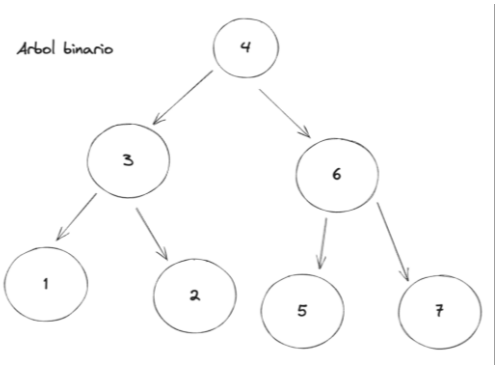
Tema 3: Estructuras Arbóreas

Árbol: estructura de datos jerárquica que consiste en nodos conectados por aristas. Cada nodo tiene un valor y puede tener cero o más nodos hijos. Un nodo sin hijos se llama nodo hoja, y el nodo desde el cual se origina una rama se llama nodo raíz.

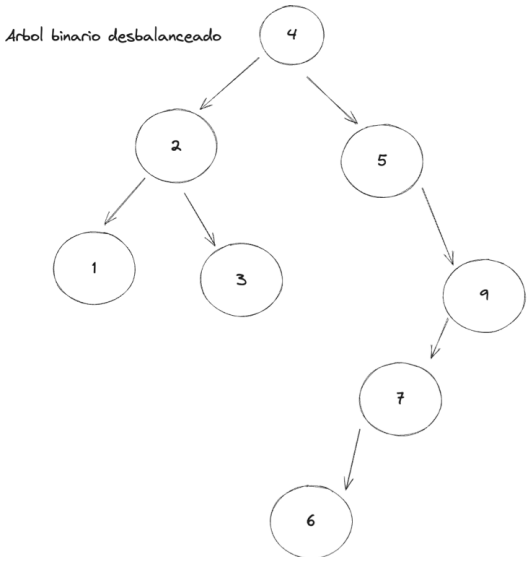
Root: nodo raíz.



Árbol Binario de Búsqueda (BST): Tipo especial de árbol en el que cada nodo tiene máximo dos hijos. En este caso, para cada nodo, todos los nodos en el subárbol izquierdo tienen valores menores que el nodo actual, y todos los nodos en el subárbol derecho tienen valores mayores. Se resuelve el problema de listas lentas, ya que el beneficio es hacer menos búsquedas.



Árbol Balanceado: Los árboles binarios de búsqueda se vuelven ineficientes si están desequilibrados. En un árbol balanceado se garantiza que la altura de los subárboles izquierdo y derecho de cualquier nodo difieren, como máximo, en 1. Esto asegura que las operaciones sean eficientes en términos de tiempo. Se traduce en menos búsquedas, pero también es costoso, depende del caso que es lo que hacemos. Se balancea a través de rotaciones simples o dobles, en cada una se modifica el nodo raiz.



Árbol Desbalanceado: Un árbol no balanceado no sigue la propiedad de altura equilibrada. Puede ser desequilibrado y tener un alto grado de profundidad de un lado, lo que vuelve a las operaciones ineficientes.

Código para saber si está balanceado:

```
#include <iostream>
#include <cstdlib> // para la función abs()

struct TreeNode {  // Definición de un nodo de árbol binario de búsqueda
  int data;
  TreeNode* left;
  TreeNode* right;
};

// Función para calcular la altura de un árbol
int calcularAltura(TreeNode* root) {
  if (root == nullptr) {
        return 0;
  }
```

```
  int alturaIzquierda = calcularAltura(root->left);
  int alturaDerecha = calcularAltura(root->right);
  return 1 + max(alturaIzquierda, alturaDerecha);
}

// Función para verificar si un árbol está balanceado
bool estaBalanceado(TreeNode* root) {
   if (root == nullptr) {
        return true; // Un árbol vacío se considera balanceado
}
   int alturaIzquierda = calcularAltura(root->left);
   int alturaDerecha = calcularAltura(root->right);

   int diferenciaAltura = alturaIzquierda - alturaDerecha;

   if (diferenciaAltura < 0)
        diferenciaAltura = diferenciaAltura * -1;

   if (diferenciaAltura <= 1 && estaBalanceado(root->left) &&
estaBalanceado(root->right)) {
        return true;
}
   return false;
}

int main() {
    TreeNode* root = .... // armar arbol

   // Verifica si el árbol está balanceado
    if (estaBalanceado(root)) {

       cout << "El árbol está balanceado." << endl;
    } else {
       cout << "El árbol no está balanceado." << endl;
    }
return 0; }
```

Inserción en Árboles

```
#include <iostream>

struct TreeNode {
  int data;
  TreeNode* left;
  TreeNode* right;
};
```

```
TreeNode* crearNodo(int valor) {
  TreeNode* nuevoNodo = new TreeNode;
  nuevoNodo->data = valor;
  nuevoNodo->left = nullptr;
  nuevoNodo->right = nullptr;
  return nuevoNodo;
}

// Función para insertar un nuevo elemento en el árbol binario de búsqueda
TreeNode* insertar(TreeNode* root, int valor) {
   if (root == nullptr) {
        return crearNodo(valor); // Si el nodo es nulo, crea un nuevo nodo
                                    con el valor dado

}
   // Si el valor es menor que el valor actual, inserta en el subárbol
izquierdo
    if (valor < root->data) {

       root->left = insertar(root->left, valor);
    }

   // Si el valor es mayor que el valor actual, inserta en el subárbol
derecho
    else if (valor > root->data) {
       root->right = insertar(root->right, valor);
    }
   return root; // Retorna el nodo raíz actual después de la inserción
}

int main() {
    TreeNode* root = nullptr; // Inicializa un árbol vacío

   // Inserta elementos en el árbol
    root = insertar(root, 4);
    root = insertar(root, 2);
    root = insertar(root, 6);

   root = insertar(root, 1);
    root = insertar(root, 3);
    root = insertar(root, 5);
    root = insertar(root, 7);

   // Tu árbol ahora contiene los elementos 1, 2, 3, 4, 5, 6, 7
return 0;
}
```

Insertar Balanceado: Se aplica una estrategia de inserción que mantenga su equilibrio, asegurando que la diferencia de altura entre los subárboles izquierdo y derecho de cualquier nodo sea como máximo 1. Por pasos:

1. Realiza la inserción del elemento normal.
2. Verificar si la propiedad de equilibrio se ha roto en algún nodo.
3. Si hay un desequilibrio, realiza las rotaciones necesarias para restaurarlo. Las rotaciones comunes son las rotaciones simples y dobles.
4. Continúa verificando y ajustando el equilibrio hacia arriba en el árbol, hasta el nodo raíz si es necesario.

```
#include <iostream>
#include <algorithm>

struct TreeNode {
  int data;
  TreeNode* left;
  TreeNode* right;
  int height; // Altura del nodo
};

// Función para obtener la altura de un nodo (teniendo en cuenta nodos
nulos)
int obtenerAltura(TreeNode* node) {
  if (node == nullptr) {
       return 0;
  }
   return node->height;
}

// Función para recalcular la altura de un nodo
void recalcularAltura(TreeNode* node) {

   if (node == nullptr) {
        return;
}
   node->height = 1 + max(obtenerAltura(node->left), obtenerAltura(node-
>right));
}

// Función para realizar una rotación simple a la derecha
TreeNode* rotacionDerecha(TreeNode* y) {
  TreeNode* x = y->left;
  TreeNode* T2 = x->right;
  x->right = y;
  y->left = T2;

  recalcularAltura(y);
  recalcularAltura(x);
  return x;
```

```
}

// Función para realizar una rotación simple a la izquierda
TreeNode* rotacionIzquierda(TreeNode* x) {
  TreeNode* y = x->right;
  TreeNode* T2 = y->left;
  y->left = x;
  x->right = T2;

  recalcularAltura(x);
  recalcularAltura(y);
  return y;
}

// Función para crear un nuevo nodo con el valor dado
TreeNode* crearNodo(int valor) {
  TreeNode* nuevoNodo = new TreeNode;
  nuevoNodo->data = valor;
  nuevoNodo->left = nullptr;
  nuevoNodo->right = nullptr;
  nuevoNodo->height = 1;
  return nuevoNodo;
}

// Función para insertar un nuevo elemento en el árbol AVL
TreeNode* insertar(TreeNode* root, int valor) {
    if (root == nullptr) {
       return crearNodo(valor); // Si el nodo es nulo, crea un nuevo nodo
                                con el valor dado
    }

   if (valor < root->data) {
        root->left = insertar(root->left, valor);
   }
    else if (valor > root->data) {
       root->right = insertar(root->right, valor);
    }
   else {
        return root; // Valor duplicado, no se permite en un AVL
   }

  recalcularAltura(root);
  int diferenciaAltura = obtenerAltura(root->left) - obtenerAltura(root-
>right);

   // Verificar y realizar rotaciones para mantener el equilibrio
    // Rotación a la derecha (simple o doble)
    if (diferenciaAltura > 1 && valor < root->left->data) {
       return rotacionDerecha(root);
```

```
    }

    // Rotación a la izquierda (simple o doble)
    if (diferenciaAltura < -1 && valor > root->right->data) {
        return rotacionIzquierda(root);
    }

    // Rotación izquierda-derecha (doble)
    if (diferenciaAltura > 1 && valor > root->left->data) {
        root->left = rotacionIzquierda(root->left);
        return rotacionDerecha(root);
    }

    // Rotación derecha-izquierda (doble)
    if (diferenciaAltura < -1 && valor < root->right->data) {
        root->right = rotacionDerecha(root->right);
        return rotacionIzquierda(root);
    }
    return root;
}
```

Recorrido de Árboles: Hay tres tipos principales de recorrido de árboles: Preorden, Inorden y Postorden. Son fundamentales para explicar y procesar los elementos de un árbol de manera sistemática.

Preorden: primero visitamos el nodo raíz, luego el subárbol izquierdo y luego el derecho. Es decir, primero procesamos el nodo actual antes de a sus hijos.

Inorden: primero se visita el subárbol izquierdo, luego el nodo raíz y finalmente el subárbol derecho. Los nodos se visitan en orden ascendente. De más chico a más grande.

Postorden: primero visitamos el subárbol izquierdo, luego el derecho y finalmente el raíz. Se utiliza, en gral, para liberar memoria de un árbol. Por pisos, empezando por el izquierdo.

Implementación de Recorridos

Recorridos Recursivos: una función se llama a sí misma para explorar los subárboles. Útil para comprender los conceptos, pero menos eficiente para árboles muy grandes debido a la sobrecarga de llamadas a funciones.

Recorridos Iterativos: a través de pilas o colas. Más eficientes en términos de espacio y velocidad, especialmente para árboles grandes.

Ejemplo con tipos de recorridos, de forma recursiva e iterativa:

```
#include <iostream>
```

```
struct TreeNode {
  int data;
  TreeNode* left;
  TreeNode* right;
};

// Función para crear un nuevo nodo
TreeNode* crearNodo(int val) {
  TreeNode* nuevoNodo = new TreeNode;
  nuevoNodo->data = val;
  nuevoNodo->left = nullptr;
  nuevoNodo->right = nullptr;
  return nuevoNodo;
}

// Recorrido en preorden de manera iterativa sin usar pila
void preordenIterativo(TreeNode* root) {
  TreeNode* actual = root;
  while (actual != nullptr) {
       cout << actual->data << " ";

       if (actual->left) {
            TreeNode* predecesor = actual->left;
            while (predecesor->right != nullptr && predecesor->right !=
actual) {
               predecesor = predecesor->right;
            }
           if (predecesor->right == nullptr) {
                predecesor->right = actual;
                actual = actual->left;
           } else {
                predecesor->right = nullptr;
                actual = actual->right;
           }
        } else {
           actual = actual->right;
        }
} }

// Recorrido en inorden de manera iterativa sin usar pila
void inordenIterativoSinPila(TreeNode* root) {
  TreeNode* actual = root;
  while (actual != nullptr) {
       if (actual->left == nullptr) {
            cout << actual->data << " ";
            actual = actual->right;
       } else {
            TreeNode* predecesor = actual->left;
```

```
            while (predecesor->right != nullptr && predecesor->right !=
actual) {
                predecesor = predecesor->right;
            }
            if (predecesor->right == nullptr) {
                predecesor->right = actual;
                actual = actual->left;
            } else {
                predecesor->right = nullptr;
                cout << actual->data << " ";
                actual = actual->right;
} }
} }

// Recorrido en preorden de manera recursiva
void preordenRecursivo(TreeNode* root) {
   if (root == nullptr) {
        return;
   }
    cout << root->data << " ";
    preordenRecursivo(root->left);
    preordenRecursivo(root->right);
}

// Recorrido en inorden de manera recursiva
void inordenRecursivo(TreeNode* root) {
   if (root == nullptr) {
        return;
   }
    inordenRecursivo(root->left);
    cout << root->data << " ";
    inordenRecursivo(root->right);
}

int main() {
    // Crear un árbol binario de ejemplo
    TreeNode* root = crearNodo(1);
    root->left = crearNodo(2);
    root->right = crearNodo(3);
    root->left->left = crearNodo(4);
    root->left->right = crearNodo(5);

   cout << "Recorrido en Preorden (Iterativo): ";
    preordenIterativo(root);
    cout << endl;

   cout << "Recorrido en Inorden (Iterativo sin Pila): ";
    inordenIterativoSinPila(root);
    cout << endl;
```

```
   cout << "Recorrido en Preorden (Recursivo): ";
    preordenRecursivo(root);
    cout << endl;

   cout << "Recorrido en Inorden (Recursivo): ";
    inordenRecursivo(root);
    cout << endl;

   // Liberar la memoria (se debe hacer al final)
    delete root->left->left;
    delete root->left->right;
    delete root->left;
    delete root->right;
    delete root;
return 0;
}
```