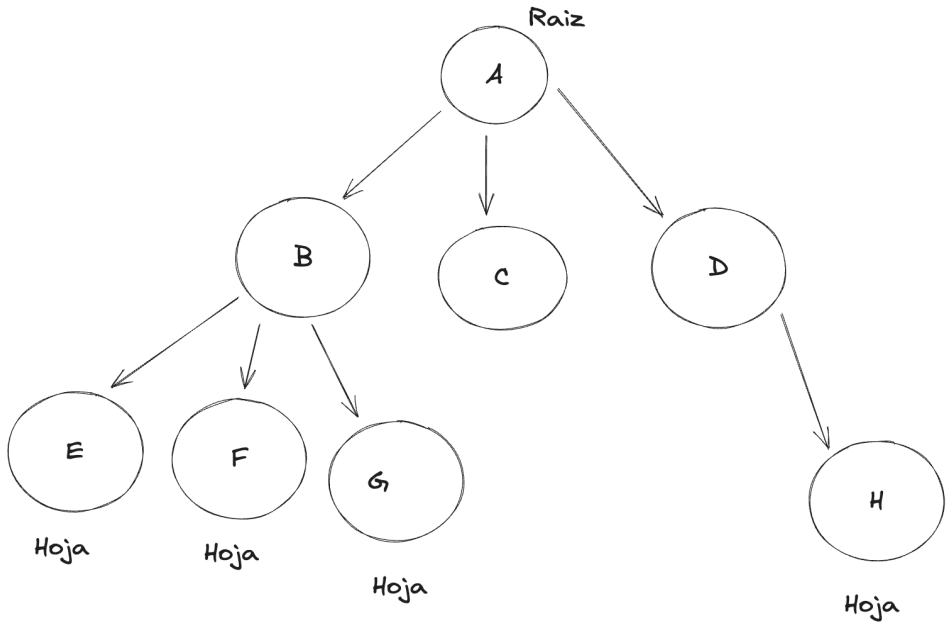
Estructuras arbóreas

Definición de Árbol

Un árbol es una 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.



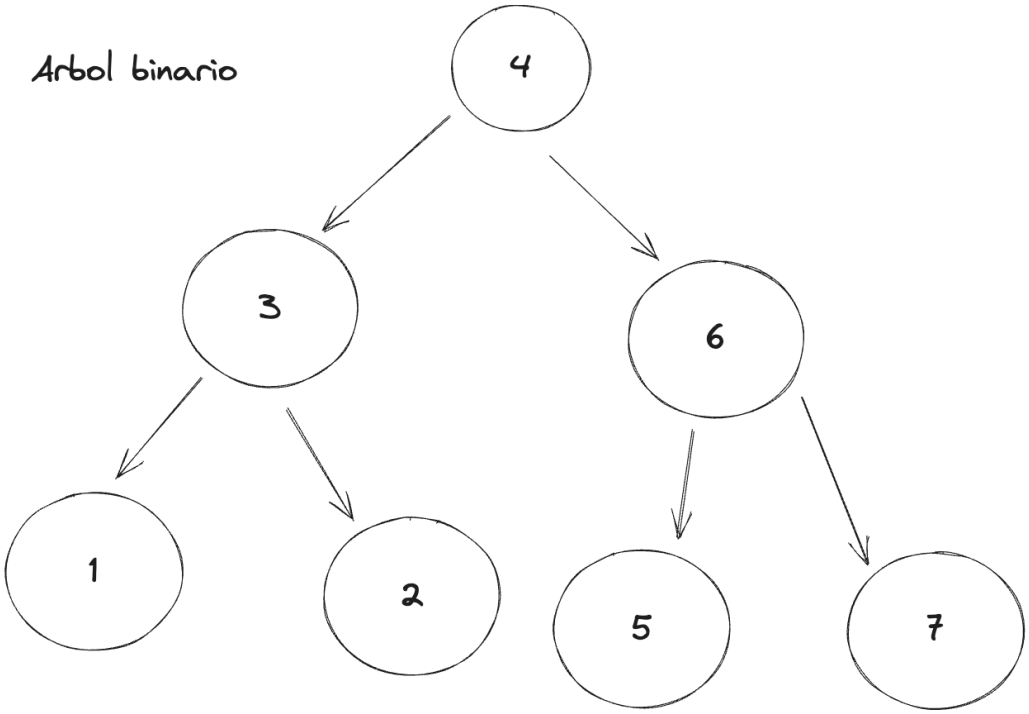
Árbol Binario de Búsqueda (BST)

Un árbol binario de búsqueda es un tipo especial de árbol en el que cada nodo tiene, como máximo, dos hijos. Además, los nodos siguen una propiedad importante: 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.

Árbol Balanceado

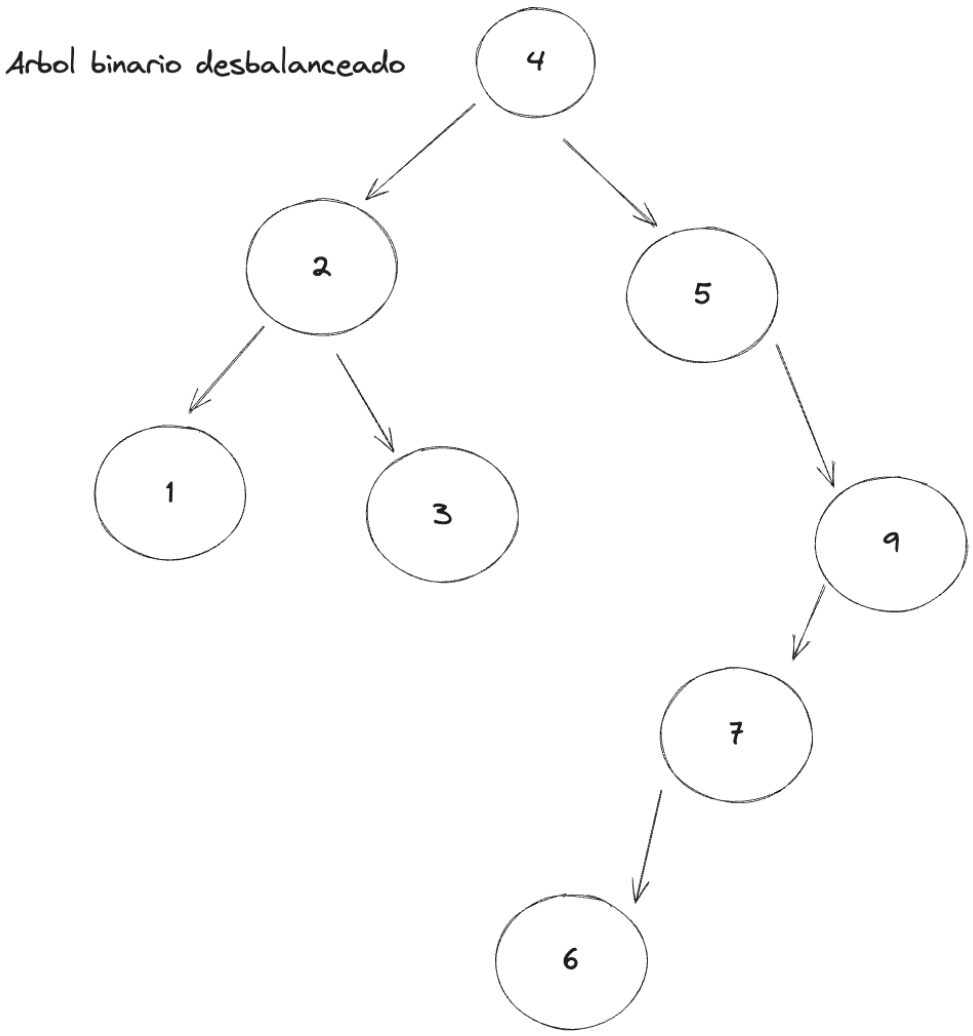
Los árboles binarios de búsqueda pueden volverse ineficientes si están desequilibrados, lo que puede llevar a operaciones costosas. Un árbol balanceado es un tipo de árbol

binario de búsqueda en el que se garantiza que la altura de los subárboles izquierdo y derecho de cualquier nodo difiere en, como máximo, 1. Esto asegura que las operaciones sean eficientes en términos de tiempo.



Árbol desbalanceado

Un árbol binario de búsqueda no balanceado no sigue la propiedad de altura equilibrada. Puede ser desequilibrado y tener un alto grado de profundidad en un lado, lo que hace que las operaciones de búsqueda sean ineficientes.



Código para saber si está balanceado

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

// Definición de un nodo de árbol binario de búsqueda
struct TreeNode {
    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 + std::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)) {
        std::cout << "El árbol está balanceado." << std::endl;
    } else {
        std::cout << "El árbol no está balanceado." << std::endl;
    }

    ...

    return 0;
}
```

Inserción en Árboles

```
#include <iostream>

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

// 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;
    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;
}
```

¿Cómo hago para insertarlo y garantizar que este balanceado?

Para garantizar que un árbol binario de búsqueda esté balanceado mientras se insertan elementos, es importante aplicar una estrategia de inserción que mantenga su equilibrio.

Se debe asegurar que la diferencia de altura entre los subárboles izquierdo y derecho de cualquier nodo sea como máximo 1. Aquí tienes un enfoque general para lograr esto durante la inserción:

1. Realiza la inserción del elemento como lo harías en un árbol binario de búsqueda normal.
2. Después de la inserción, verifica si la propiedad de equilibrio se ha roto en algún nodo. Puedes hacer esto verificando la diferencia de altura entre los subárboles izquierdo y derecho del nodo en cuestión.
3. Si la diferencia de altura es mayor que 1 (lo que indica un desequilibrio), realiza las rotaciones adecuadas para restaurar la propiedad de equilibrio. Las rotaciones comunes son las rotaciones simples y dobles (izquierda-izquierda, izquierda-derecha, derecha-derecha y derecha-izquierda).
4. Continúa verificando y ajustando el equilibrio hacia arriba en el árbol, hacia el nodo raíz, si es necesario.

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

// Definición de un nodo de árbol binario de búsqueda AVL
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 + std::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 da
do
    }

    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

En esta sección, exploraremos los tres tipos principales de recorrido de árboles: Preorden, Inorden y Postorden. Estos recorridos son fundamentales para explorar y procesar los elementos de un árbol de manera sistemática.

Tipos de Recorrido

Preorden

En el recorrido en preorden, primero visitamos el nodo raíz, luego el subárbol izquierdo y finalmente el subárbol derecho. Es decir, primero procesamos el nodo actual antes de explorar sus hijos.

Inorden

En el recorrido inorden, primero visitamos el subárbol izquierdo, luego el nodo raíz y finalmente el subárbol derecho. Esto significa que los nodos se visitan en orden ascendente en un árbol de búsqueda binaria.

Postorden

En el recorrido en postorden, primero visitamos el subárbol izquierdo, luego el subárbol derecho y finalmente el nodo raíz. Esto se utiliza a menudo para liberar la memoria de un árbol.

Implementación de Recorridos

Recorridos Recursivos

Los recorridos de árboles se pueden implementar de manera recursiva, lo que significa que una función se llama a sí misma para explorar los subárboles. Esto es especialmente útil para comprender los conceptos, pero puede ser menos eficiente para árboles muy grandes debido a la sobrecarga de llamadas a funciones.

Recorridos Iterativos

También es posible implementar recorridos de árboles de manera iterativa utilizando estructuras de datos como pilas o colas. Los enfoques iterativos a menudo son más eficientes en términos de espacio y velocidad, especialmente para árboles grandes.

Ejemplo de Código en C++

A continuación, proporcionaré ejemplos de código en C++ para realizar los tres tipos de recorrido en un árbol binario:

```
#include <iostream>
#include <stack>

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

    TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

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

// Recorrido en preorden de manera iterativa
void preordenIterativo(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    std::stack<TreeNode*> pila;
    pila.push(root);

    while (!pila.empty()) {
        TreeNode* actual = pila.top();
        pila.pop();

        std::cout << actual->data << " ";

        if (actual->right) {
            pila.push(actual->right);
        }
        if (actual->left) {
            pila.push(actual->left);
        }
    }
}

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

```
    std::cout << root->data << " ";
    inordenRecursivo(root->right);
}

// Recorrido en inorden de manera iterativa
void inordenIterativo(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    std::stack<TreeNode*> pila;
    TreeNode* actual = root;

    while (actual != nullptr || !pila.empty()) {
        while (actual != nullptr) {
            pila.push(actual);
            actual = actual->left;
        }
        actual = pila.top();
        pila.pop();
        std::cout << actual->data << " ";
        actual = actual->right;
    }
}

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

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);

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

;

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

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

    std::cout << "Recorrido en Inorden (Iterativo): ";
    inordenIterativo(root);
    std::cout << std::endl;

    std::cout << "Recorrido en Postorden (Recursivo): ";
    postordenRecursivo(root);
    std::cout << std::endl;

    return 0;
}
```

Este código implementa los tres tipos de recorrido en un árbol binario y muestra cómo se pueden realizar tanto de manera recursiva como iterativa.

Ejercicios

Ejercicio 1: Crear un Árbol de Clientes

Implementa un programa que permita agregar clientes a un árbol binario de búsqueda basado en su número de DNI. Cada cliente debe tener un nombre y un DNI. El programa debe proporcionar funciones para agregar clientes y mostrar la lista de clientes en orden (inorden) según su DNI.

Ejercicio 2: Buscar un Cliente por DNI

Amplía el programa anterior para que los clientes puedan buscar su información en el árbol por su número de DNI. Implementa una función que permita a los clientes ingresar su número de DNI y luego busque y muestre su nombre y DNI en el árbol. Si el cliente no se encuentra en el árbol, muestra un mensaje adecuado.

Ejercicio 3: Eliminar Clientes

Añade una función que permita eliminar clientes del árbol por su número de DNI. Los clientes deben poder proporcionar su DNI para solicitar la eliminación de su información del sistema.