UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTRUCTURA DE DATOS

ARBOLES AVL

NOMBRE: QUISHPE CHAVEZ DENISSE PAULINA

NRC: 29852

FECHA: 10/12/2025

DOCENTE: SOLIS ACOSTA EDGAR FERNANDO

INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

Contenido

INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

IMPLEMENTACIÓN DE ÁRBOL AVL

1. Introducción

1.1 objetivo del proyecto

Desarrollar un programa en c++ que implemente un árbol avl (adelson-velsky y landis) con la capacidad de almacenar valores duplicados, manteniendo las propiedades de balanceo automático y proporcionando una interfaz gráfica en modo consola dos.

1.2 alcance

El sistema permite realizar las cuatro operaciones fundamentales sobre árboles binarios de búsqueda balanceados:

- Inserción de valores (permitiendo duplicados)
- Edición de valores existentes
- Eliminación de valores (una ocurrencia a la vez)
- Visualización gráfica del árbol en consola

2. Marco teórico

2.1 ¿qué es un árbol avl?

Un árbol avl es un árbol binario de búsqueda auto-balanceable donde la diferencia de alturas entre los subárboles izquierdo y derecho de cualquier nodo (factor de balance) no puede ser mayor a 1.

Propiedades:

- Para todo nodo: |altura(subárbol_izquierdo) - altura(subárbol_derecho)| ≤ 1

- Las operaciones de inserción, eliminación y búsqueda tienen complejidad o(log n)
- Se balancea automáticamente mediante rotaciones

2.2 factor de balance

```
Balance(n) = altura(subárbol_izquierdo) - altura(subárbol_derecho)
Valores permitidos: -1, 0, 1
```

2.3 rotaciones avl

El árbol se balancea mediante cuatro tipos de rotaciones:

1. Rotación Simple Derecha (LL): Cuando el desbalance está en el hijo izquierdo del hijo izquierdo
2. Rotación Simple Izquierda (RR): Cuando el desbalance está en el hijo derecho del hijo derecho
3. Rotación Doble Izquierda-Derecha (LR): Desbalance en hijo derecho del hijo izquierdo
4. Rotación Doble Derecha-Izquierda (RL): Desbalance en hijo izquierdo del hijo derecho

2.4 Manejo de Duplicados

En un árbol AVL tradicional, los valores duplicados no se permiten. En esta implementación:

- Los valores duplicados se insertan en el subárbol derecho
- Se utiliza la regla: valor < nodo.dato → izquierda | valor ≥ nodo.dato → derecha
- Esto mantiene la propiedad BST y permite múltiples ocurrencias del mismo valor

3. Análisis del código

3.1 Estructura

```
class Nodo {
public:
    int dato;          // Valor almacenado
    Nodo* izq;         // Puntero al hijo izquierdo
    Nodo* der;         // Puntero al hijo derecho
    int altura;        // Altura del nodo en el árbol
};
```

3.2 Funciones Principales

Función de Inserción

```
Nodo* insertar(Nodo* nodo, int valor) {
    if (nodo == NULL) return new Nodo(valor);

    if (valor < nodo->dato)
        nodo->izq = insertar(nodo->izq, valor);
    else
        nodo->der = insertar(nodo->der, valor);  // Duplicados van aquí

    return balancear(nodo);
}
```

Complejidad: O(log n) en promedio

Función de Balanceo

```
Nodo* balancear(Nodo* nodo) {
    actualizarAltura(nodo);
    int balance = calcularBalance(nodo);
    // Detecta y corrige desbalances con rotaciones
    if (balance > 1 || balance < -1) {
        // Aplica rotaciones según el caso
    }
    return nodo;
}
```

Función de Eliminación

```
Nodo* eliminar(Nodo* raiz, int valor, bool& encontrado) {
    // Búsqueda del nodo
    // Eliminación según casos (hoja, un hijo, dos hijos)
```

```
    // Rebalanceo del árbol
    return balancear(raiz);
}
```

Casos de eliminación:

1. Nodo hoja: Se elimina directamente
2. Nodo con un hijo: Se reemplaza por su hijo
3. Nodo con dos hijos: Se reemplaza con el sucesor inorden (mínimo del subárbol derecho)

3.3 Visualización Gráfica

```
void dibujarArbol(Nodo* raiz, int x, int y, int espaciado) {
    // Dibuja el nodo en posición (x, y)
    // Dibuja líneas / y \ para conexiones
    // Recursión para subárboles izquierdo y derecho
}
```

3.4 Codigo completo

```
#include <iostream>
#include <cstdlib>
#include <windows.h>
#include <algorithm>

using namespace std;

// ============ CLASE NODO AVL ============
class Nodo {
public:
    int dato;
    Nodo* izq;
    Nodo* der;
    int altura;

    Nodo(int val) {
        dato = val;
        izq = NULL;
        der = NULL;
        altura = 1;
    }
};
```

```
// ============ UTILIDADES GRAFICAS ============
void posicionar(int x, int y) {
    HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);
    COORD pos;
    pos.X = x;
    pos.Y = y;
    SetConsoleCursorPosition(h, pos);
}

void limpiarPantalla() {
    system("cls");
}

// ============ FUNCIONES AVL ============

int obtenerAltura(Nodo* n) {
    if (n == NULL) return 0;
    return n->altura;
}

int calcularBalance(Nodo* n) {
    if (n == NULL) return 0;
    return obtenerAltura(n->izq) - obtenerAltura(n->der);
}

void actualizarAltura(Nodo* n) {
    if (n != NULL) {
        n->altura = 1 + max(obtenerAltura(n->izq), obtenerAltura(n->der));
    }
}

// ============ ROTACIONES ============

Nodo* rotarDerecha(Nodo* y) {
    Nodo* x = y->izq;
    Nodo* T2 = x->der;

    x->der = y;
    y->izq = T2;

    actualizarAltura(y);
    actualizarAltura(x);

    return x;
}
```

```
Nodo* rotarIzquierda(Nodo* x) {
    Nodo* y = x->der;
    Nodo* T2 = y->izq;

    y->izq = x;
    x->der = T2;

    actualizarAltura(x);
    actualizarAltura(y);

    return y;
}

// ============ BALANCEAR ============

Nodo* balancear(Nodo* nodo) {
    if (nodo == NULL) return NULL;

    actualizarAltura(nodo);
    int balance = calcularBalance(nodo);

    // Caso Izquierda-Izquierda
    if (balance > 1 && calcularBalance(nodo->izq) >= 0) {
        return rotarDerecha(nodo);
    }

    // Caso Izquierda-Derecha
    if (balance > 1 && calcularBalance(nodo->izq) < 0) {
        nodo->izq = rotarIzquierda(nodo->izq);
        return rotarDerecha(nodo);
    }

    // Caso Derecha-Derecha
    if (balance < -1 && calcularBalance(nodo->der) <= 0) {
        return rotarIzquierda(nodo);
    }

    // Caso Derecha-Izquierda
    if (balance < -1 && calcularBalance(nodo->der) > 0) {
        nodo->der = rotarDerecha(nodo->der);
        return rotarIzquierda(nodo);
    }

    return nodo;
}
```

```
// ============ INSERTAR (PERMITE DUPLICADOS SIN RESTRICCION) ============

Nodo* insertar(Nodo* nodo, int valor) {
    // Insercion normal BST
    if (nodo == NULL) {
        return new Nodo(valor);
    }

    // CLAVE: Los valores menores van a izquierda
    // Los valores MAYORES O IGUALES van a derecha (permitiendo duplicados)
    if (valor < nodo->dato) {
        nodo->izq = insertar(nodo->izq, valor);
    } else {
        // Aqui entran tanto valores mayores COMO IGUALES (duplicados)
        nodo->der = insertar(nodo->der, valor);
    }

    // Balancear el nodo
    return balancear(nodo);
}

// ============ BUSCAR MINIMO ============

Nodo* buscarMinimo(Nodo* nodo) {
    Nodo* actual = nodo;
    while (actual && actual->izq != NULL) {
        actual = actual->izq;
    }
    return actual;
}

// ============ ELIMINAR (Una ocurrencia) ============

Nodo* eliminar(Nodo* raiz, int valor, bool& encontrado) {
    if (raiz == NULL) {
        encontrado = false;
        return NULL;
    }

    // Buscar el nodo
    if (valor < raiz->dato) {
        raiz->izq = eliminar(raiz->izq, valor, encontrado);
    }
    else if (valor > raiz->dato) {
        raiz->der = eliminar(raiz->der, valor, encontrado);
    }
```

```
    else {
        // Nodo encontrado - eliminar
        encontrado = true;

        if (raiz->izq == NULL || raiz->der == NULL) {
            Nodo* temp = raiz->izq ? raiz->izq : raiz->der;

            if (temp == NULL) {
                temp = raiz;
                raiz = NULL;
            } else {
                *raiz = *temp;
            }
            delete temp;
        }
        else {
            Nodo* temp = buscarMinimo(raiz->der);
            raiz->dato = temp->dato;
            bool aux = false;
            raiz->der = eliminar(raiz->der, temp->dato, aux);
        }
    }

    if (raiz == NULL) return NULL;

    return balancear(raiz);
}

// ============ BUSCAR (verifica existencia) ============

bool existeValor(Nodo* raiz, int valor) {
    if (raiz == NULL) return false;
    if (raiz->dato == valor) return true;
    if (valor < raiz->dato) return existeValor(raiz->izq, valor);
    return existeValor(raiz->der, valor);
}

// ============ CONTAR OCURRENCIAS ============

int contarOcurrencias(Nodo* raiz, int valor) {
    if (raiz == NULL) return 0;
    int count = 0;
    if (raiz->dato == valor) count = 1;
    count += contarOcurrencias(raiz->izq, valor);
    count += contarOcurrencias(raiz->der, valor);
    return count;
}
```

```
// ============ DIBUJAR ARBOL ============

void dibujarArbol(Nodo* raiz, int x, int y, int espaciado) {
    if (raiz == NULL) return;

    // Dibujar el nodo actual
    posicionar(x, y);
    cout << "[" << raiz->dato << "]";

    // Dibujar hijo izquierdo
    if (raiz->izq != NULL) {
        posicionar(x - espaciado/2, y + 1);
        cout << "/";
        dibujarArbol(raiz->izq, x - espaciado, y + 2, espaciado/2);
    }

    // Dibujar hijo derecho
    if (raiz->der != NULL) {
        posicionar(x + espaciado/2, y + 1);
        cout << "\\";
        dibujarArbol(raiz->der, x + espaciado, y + 2, espaciado/2);
    }
}

// ============ RECORRIDOS ============

void inorden(Nodo* raiz) {
    if (raiz == NULL) return;
    inorden(raiz->izq);
    cout << raiz->dato << " ";
    inorden(raiz->der);
}

void preorden(Nodo* raiz) {
    if (raiz == NULL) return;
    cout << raiz->dato << " ";
    preorden(raiz->izq);
    preorden(raiz->der);
}

void postorden(Nodo* raiz) {
    if (raiz == NULL) return;
    postorden(raiz->izq);
    postorden(raiz->der);
    cout << raiz->dato << " ";
}
```

```
// ============ CALCULOS MATEMATICOS ============

int contarNodos(Nodo* raiz) {
    if (raiz == NULL) return 0;
    return 1 + contarNodos(raiz->izq) + contarNodos(raiz->der);
}

int calcularAltura(Nodo* raiz) {
    if (raiz == NULL) return 0;
    return 1 + max(calcularAltura(raiz->izq), calcularAltura(raiz->der));
}

int sumarNodos(Nodo* raiz) {
    if (raiz == NULL) return 0;
    return raiz->dato + sumarNodos(raiz->izq) + sumarNodos(raiz->der);
}

// ============ MENU PRINCIPAL ============

int main() {
    Nodo* raiz = NULL;
    int opcion, valor, nuevoValor;
    string mensaje = "";

    do {
        limpiarPantalla();

        // Titulo
        posicionar(0, 0);
        cout << "=== ARBOL AVL: PERMITE VALORES REPETIDOS ===";
        posicionar(0, 1);
        cout << "1. Insertar";
        posicionar(0, 2);
        cout << "2. Editar";
        posicionar(0, 3);
        cout << "3. Eliminar";
        posicionar(0, 4);
        cout << "4. Mostrar recorridos";
        posicionar(0, 5);
        cout << "5. Calculos matematicos";
        posicionar(0, 6);
        cout << "6. Salir";
        posicionar(0, 7);
        cout << "--------------------------------------";

        // Dibujar arbol
```

```
        if (raiz != NULL) {
            dibujarArbol(raiz, 40, 9, 20);
        } else {
            posicionar(0, 9);
            cout << "       (Arbol vacio)";
        }

        // Mensaje de estado
        posicionar(0, 20);
        cout << "--------------------------------------";
        posicionar(0, 21);
        cout << ">> ESTADO: " << mensaje;
        posicionar(0, 22);
        cout << "--------------------------------------";

        // Estadisticas
        posicionar(0, 23);
        cout << "Nodos totales: " << contarNodos(raiz) << " | Altura: " <<
calcularAltura(raiz);

        // Entrada
        posicionar(0, 25);
        cout << "Seleccione opcion: ";
        cin >> opcion;

        switch(opcion) {
            case 1:
                posicionar(0, 26);
                cout << "Valor a insertar: ";
                cin >> valor;
                raiz = insertar(raiz, valor);
                mensaje = "Valor insertado correctamente.";
                break;

            case 2:
                posicionar(0, 26);
                cout << "Valor a editar: ";
                cin >> valor;

                if (existeValor(raiz, valor)) {
                    int ocurrencias = contarOcurrencias(raiz, valor);
                    posicionar(0, 27);
                    cout << "Valor encontrado (" << ocurrencias << " ocurrencia(s))";
                    posicionar(0, 28);
                    cout << "Nuevo valor: ";
                    cin >> nuevoValor;
```

```
                bool encontrado = false;
                raiz = eliminar(raiz, valor, encontrado);
                raiz = insertar(raiz, nuevoValor);
                mensaje = "Valor editado (1 ocurrencia modificada).";
            } else {
                mensaje = "ERROR: Valor no encontrado.";
            }
            break;

        case 3:
            posicionar(0, 26);
            cout << "Valor a eliminar: ";
            cin >> valor;

            if (existeValor(raiz, valor)) {
                int ocurrencias = contarOcurrencias(raiz, valor);
                bool encontrado = false;
                raiz = eliminar(raiz, valor, encontrado);
                if (encontrado) {
                    mensaje = "Valor eliminado (1 ocurrencia). Quedan " +
to_string(ocurrencias - 1);
                }
            } else {
                mensaje = "ERROR: Valor no encontrado.";
            }
            break;

        case 4:
            limpiarPantalla();
            posicionar(0, 0);
            cout << "=== RECORRIDOS DEL ARBOL ===";

            posicionar(0, 2);
            cout << "INORDEN (Izq-Raiz-Der): ";
            inorden(raiz);

            posicionar(0, 4);
            cout << "PREORDEN (Raiz-Izq-Der): ";
            preorden(raiz);

            posicionar(0, 6);
            cout << "POSTORDEN (Izq-Der-Raiz): ";
            postorden(raiz);

            posicionar(0, 8);
            cout << "Presione ENTER para continuar...";
            cin.ignore();
```

```
            cin.get();
            mensaje = "Recorridos mostrados.";
            break;

        case 5:
            limpiarPantalla();
            posicionar(0, 0);
            cout << "=== CALCULOS MATEMATICOS ===";

            posicionar(0, 2);
            cout << "Total de nodos: " << contarNodos(raiz);

            posicionar(0, 3);
            cout << "Altura del arbol: " << calcularAltura(raiz);

            posicionar(0, 4);
            cout << "Suma de todos los valores: " << sumarNodos(raiz);

            if (contarNodos(raiz) > 0) {
                posicionar(0, 5);
                cout << "Promedio: " << (float)sumarNodos(raiz) / contarNodos(raiz);
            }

            posicionar(0, 6);
            cout << "Balance de la raiz: " << calcularBalance(raiz);

            posicionar(0, 8);
            cout << "Presione ENTER para continuar...";
            cin.ignore();
            cin.get();
            mensaje = "Calculos realizados.";
            break;

        case 6:
            mensaje = "Saliendo del programa...";
            break;

        default:
            mensaje = "Opcion invalida.";
    }

} while(opcion != 6);

limpiarPantalla();
posicionar(0, 0);
cout << "Gracias por usar el programa!" << endl;
```

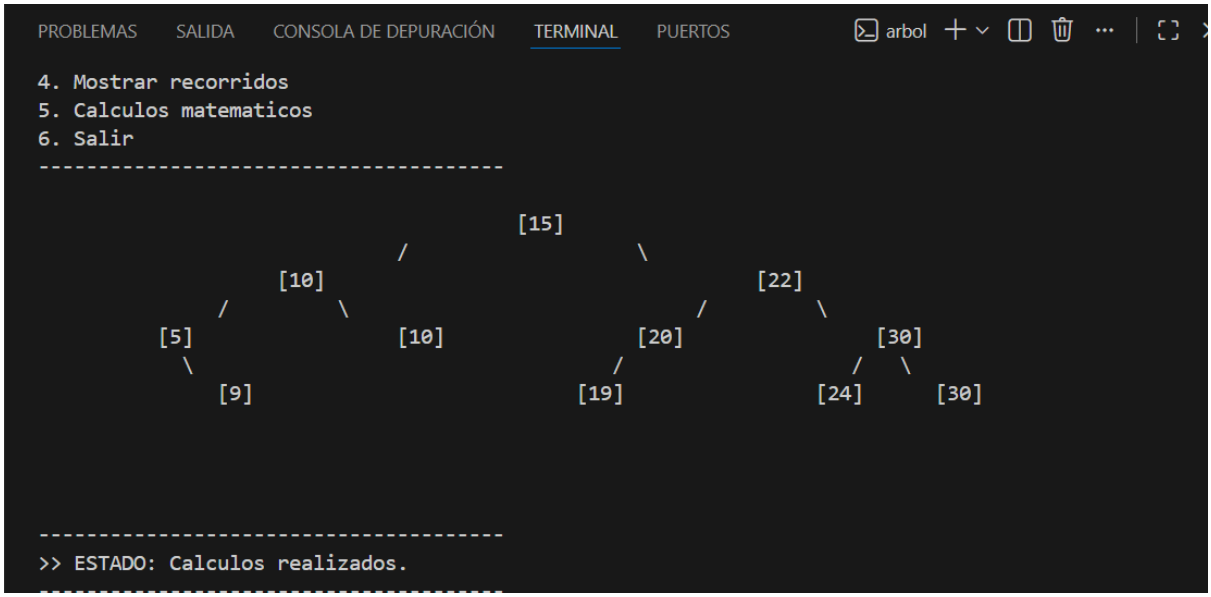
```
    return 0;
}
```

4. Análisis de caso de prueba

4.1 datos de entrada

Secuencia de valores insertados:

15, 10, 22, 5, 10, 20, 30, 9, 19, 24, 30



4.2 Verificación de Propiedades AVL

Balance de cada nodo:

- [15]: altura_izq=3, altura_der=3 → Balance = 0
- [10]: altura_izq=2, altura_der=1 → Balance = 1
- [22]: altura_izq=2, altura_der=2 → Balance = 0
- [5]: altura_izq=0, altura_der=1 → Balance = -1
- [20]: altura_izq=1, altura_der=0 → Balance = 1

- [30]: altura_izq=1, altura_der=1 → Balance = 0

4.3. Casos de prueba ejecutados

#	Operación	Entrada	Resultado Esperado	Estado
1	Insertar	15	Árbol con raíz [15]	Pasó
2	Insertar	10	[10] como hijo izquierdo	Pasó
3	Insertar	22	[22] como hijo derecho	Pasó
4	Insertar	10 (duplicado)	Segundo [10] en subárbol derecho	Pasó
5	Insertar	30 (duplicado)	Ambos [30] correctamente ubicados	Pasó
6	Verificar balance	-	Todos los nodos con balance ≤ 1	Pasó
7	Eliminar	10	Solo elimina una ocurrencia	Pasó
8	Editar	5→7	Elimina [5], inserta [7], mantiene balance	Pasó