Ejercidos de árboles de búsqueda binaria, arboles rojos negros

Nombre: Kevin Andino Fecha:14/12/2025

1: ABB con 1000 números, barridos y altura

```
// Archivo: Ejercicio1_ABB_Completo.cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

struct Nodo {
   int dato;
   Nodo *izq, *der;
   Nodo(int d) : dato(d), izq(nullptr), der(nullptr) {}
};

class ArbolBinario {
   Nodo* raiz;

   void insertar(Nodo*& nodo, int valor) {
      if (!nodo) nodo = new Nodo(valor);
      else if (valor < nodo->dato) insertar(nodo->izq, valor);
      else if (valor > nodo->dato) insertar(nodo->der, valor);
   }

   void inOrden(Nodo* nodo) {
      if (!nodo) return;
      inOrden(nodo->izq);
      cout << nodo->dato << " ";
      inOrden(nodo->der);
   }

   void preOrden(Nodo* nodo) {
      if (!nodo) return;
      cout << nodo->dato << " ";
      preOrden(nodo->izq);
      preOrden(nodo->der);
   }

   void postOrden(Nodo* nodo) {
      if (!nodo) return;
      postOrden(nodo->izq);
      postOrden(nodo->der);
      cout << nodo->dato << " ";
   }

   int altura(Nodo* nodo) {
      if (!nodo) return -1;
      int altIzq = altura(nodo->izq);
      int altDer = altura(nodo->der);
      return (altIzq > altDer ? altIzq : altDer) + 1;
```

```
    }

    bool buscar(Nodo* nodo, int valor) {
        if (!nodo) return false;
        if (nodo->dato == valor) return true;
        if (valor < nodo->dato) return buscar(nodo->izq, valor);
        return buscar(nodo->der, valor);
    }

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

    void insertar(int valor) { insertar(raiz, valor); }

    void barridos() {
        cout << "In-Orden (parcial): "; inOrden(raiz); cout << "..." << endl;
        cout << "Pre-Orden (parcial): "; preOrden(raiz); cout << "..." << endl;
        cout << "Post-Orden (parcial): "; postOrden(raiz); cout << "..." << endl;
    }

    int obtenerAltura() { return altura(raiz); }
    bool existe(int valor) { return buscar(raiz, valor); }
};

int main() {
    srand(time(0));
    ArbolBinario abb;

    cout << "Insertando 1000 numeros aleatorios..." << endl;
    for(int i=0; i<1000; i++) {
        abb.insertar(rand() % 10000);
    }

    cout << "Altura del arbol: " << abb.obtenerAltura() << endl;

    // Mostramos solo un poco para no saturar la consola
    cout << "Realizando barridos..." << endl;
    abb.barridos(); // Nota: Modificado para imprimir, quitar '...' en implementacion real para
ver todo

    int buscarVal = 500;
    cout << "¿Existe el " << buscarVal << "? " << (abb.existe(buscarVal) ? "Si" : "No") <<
endl;

    return 0;
}
```

2. Árbol de Expresión Matemática

```
// Archivo: Ejercicio2_Expresion.cpp

#include <iostream>
```

```
#include <string>

using namespace std;

struct NodoExpr {
    string valor;
    NodoExpr *izq, *der;
    NodoExpr(string v) : valor(v), izq(nullptr), der(nullptr) {}
};

class ArbolExpresion {
public:
    NodoExpr* raiz;

    ArbolExpresion() : raiz(nullptr) {}

    int evaluar(NodoExpr* nodo) {
        if (!nodo->izq && !nodo->der) return stoi(nodo->valor);

        int valIzq = evaluar(nodo->izq);
        int valDer = evaluar(nodo->der);

        if (nodo->valor == "+") return valIzq + valDer;
        if (nodo->valor == "-") return valIzq - valDer;
        if (nodo->valor == "*") return valIzq * valDer;
```

```
        if (nodo->valor == "/") return valIzq / valDer;
        return 0;
    }

    // Barrido Post-Orden es el ideal para evaluar expresiones (RPN)
    void mostrarPostOrden(NodoExpr* nodo) {
        if (!nodo) return;
        mostrarPostOrden(nodo->izq);
        mostrarPostOrden(nodo->der);
        cout << nodo->valor << " ";
    }

    // Barrido In-Orden muestra la expresión matemática normal
    void mostrarInOrden(NodoExpr* nodo) {
        if (!nodo) return;
        if (nodo->izq) cout << "(";
        mostrarInOrden(nodo->izq);
        cout << " " << nodo->valor << " ";
        mostrarInOrden(nodo->der);
        if (nodo->der) cout << ")";
    }
};

int main() {
    ArbolExpresion expr;
```

```
    // Construimos manualmente la expresión: (10 + 5) * 2
    // Esto equivale a un árbol con raíz '*'

    expr.raiz = new NodoExpr("*");
    expr.raiz->der = new NodoExpr("2");
    expr.raiz->izq = new NodoExpr("+");
    expr.raiz->izq->izq = new NodoExpr("10");
    expr.raiz->izq->der = new NodoExpr("5");

    cout << "Expresion (In-Orden): ";
    expr.mostrarInOrden(expr.raiz);
    cout << endl;

    cout << "Barrido para evaluacion (Post-Orden): ";
    expr.mostrarPostOrden(expr.raiz);
    cout << endl;

    cout << "Resultado: " << expr.evaluar(expr.raiz) << endl;

    return 0;
}
```

3. Índice de Libro (Transformada de Knuth)

```
// Archivo: Ejercicio3_IndiceKnuth.cpp
#include <iostream>
#include <string>
```

```
using namespace std;

// Nodo para Árbol General transformado a Binario
// Hijo Izquierdo = Primer Hijo Original
// Hijo Derecho = Siguiente Hermano Original
struct NodoIndice {
    string titulo;
    int pagina;
    NodoIndice *hijo, *hermano;

    NodoIndice(string t, int p) : titulo(t), pagina(p), hijo(nullptr), hermano(nullptr) {}
};

class ArbolLibro {
    NodoIndice* raiz;

    NodoIndice* buscar(NodoIndice* nodo, string tituloPadre) {
        if (!nodo) return nullptr;
        if (nodo->titulo == tituloPadre) return nodo;

        NodoIndice* encontrado = buscar(nodo->hijo, tituloPadre);
        if (encontrado) return encontrado;
        return buscar(nodo->hermano, tituloPadre);
    }
```

```
public:
    ArbolLibro(string tituloLibro) {
        raiz = new NodoIndice(tituloLibro, 0);
    }

    void agregarCapitulo(string tituloPadre, string nuevoTitulo, int pag) {
        NodoIndice* padre = buscar(raiz, tituloPadre);
        if (!padre) {
            cout << "Padre no encontrado" << endl;
            return;
        }

        NodoIndice* nuevo = new NodoIndice(nuevoTitulo, pag);

        if (!padre->hijo) {
            padre->hijo = nuevo; // Primer hijo
        } else {
            NodoIndice* actual = padre->hijo;
            while (actual->hermano) actual = actual->hermano;
            actual->hermano = nuevo; // Siguiente hermano
        }
    }

    void mostrarIndice(NodoIndice* nodo, int nivel) {
```

```
        if (!nodo) return;
        for(int i=0; i<nivel; i++) cout << "  ";
        cout << "- " << nodo->titulo << " (pg. " << nodo->pagina << ")" << endl;

        mostrarIndice(nodo->hijo, nivel + 1);    // Hijos (más indentación)
        mostrarIndice(nodo->hermano, nivel);     // Hermanos (misma indentación)
    }

    NodoIndice* getRaiz() { return raiz; }
};

int main() {
    ArbolLibro libro("Estructuras de Datos");

    libro.agregarCapitulo("Estructuras de Datos", "Capitulo 1: Arboles", 10);
    libro.agregarCapitulo("Estructuras de Datos", "Capitulo 2: Grafos", 50);

    libro.agregarCapitulo("Capitulo 1: Arboles", "1.1 Binarios", 12);
    libro.agregarCapitulo("Capitulo 1: Arboles", "1.2 AVL", 20);

    libro.agregarCapitulo("Capitulo 2: Grafos", "2.1 Dijkstra", 55);

    cout << "INDICE DEL LIBRO:" << endl;
    libro.mostrarIndice(libro.getRaiz(), 0);
```

```
    return 0;
}
```

4. Obtener Hijos Izquierdo y Derecho

```
// Archivo: Ejercicio4_Hijos.cpp
#include <iostream>

using namespace std;

struct Nodo {
    int dato;
    Nodo *izq, *der;
    Nodo(int d) : dato(d), izq(nullptr), der(nullptr) {}
};

class ABB_Hijos {
    Nodo* raiz;

    void insertar(Nodo*& nodo, int valor) {
        if (!nodo) nodo = new Nodo(valor);
        else if (valor < nodo->dato) insertar(nodo->izq, valor);
        else insertar(nodo->der, valor);
    }

    Nodo* buscarNodo(Nodo* nodo, int valor) {
        if (!nodo || nodo->dato == valor) return nodo;
```

```
        if (valor < nodo->dato) return buscarNodo(nodo->izq, valor);
        return buscarNodo(nodo->der, valor);
    }

public:
    ABB_Hijos() : raiz(nullptr) {}
    void insertar(int v) { insertar(raiz, v); }

    void mostrarInfoHijos(int valor) {
        Nodo* encontrado = buscarNodo(raiz, valor);
        if (!encontrado) {
            cout << "El nodo " << valor << " no existe en el arbol." << endl;
            return;
        }
        cout << "Nodo: " << valor << endl;
        cout << " -> Hijo Izquierdo: " << (encontrado->izq ? to_string(encontrado->izq->dato) : "NULL") << endl;
        cout << " -> Hijo Derecho:   " << (encontrado->der ? to_string(encontrado->der->dato) : "NULL") << endl;
    }
};

int main() {
    ABB_Hijos arbol;
    arbol.insertar(10);
    arbol.insertar(5);
```

```
    arbol.insertar(15);
    arbol.insertar(3);

    cout << "Consultando hijos del nodo 10:" << endl;
    arbol.mostrarInfoHijos(10);

    cout << "\nConsultando hijos del nodo 5:" << endl;
    arbol.mostrarInfoHijos(5);

    return 0;
}
```

5. Superhéroes de Marvel

```
// Archivo: Ejercicio5_Marvel.cpp
#include <iostream>
#include <string>

using namespace std;

struct Personaje {
    string nombre;
    bool esVillano;

    // Operadores necesarios para comparar en el ABB
    bool operator<(const Personaje& p) const { return nombre < p.nombre; }
```

```
    bool operator>(const Personaje& p) const { return nombre > p.nombre; }
};

struct NodoP {
    Personaje dato;
    NodoP *izq, *der;
    NodoP(Personaje p) : dato(p), izq(nullptr), der(nullptr) {}
};

class ArbolMarvel {
public:
    NodoP* raiz;
    ArbolMarvel() : raiz(nullptr) {}

    void insertar(NodoP*& nodo, Personaje p) {
        if (!nodo) nodo = new NodoP(p);
        else if (p < nodo->dato) insertar(nodo->izq, p);
        else if (p > nodo->dato) insertar(nodo->der, p);
    }

    void insertar(Personaje p) { insertar(raiz, p); }

    void listarVillanos(NodoP* nodo) {
        if (!nodo) return;
        listarVillanos(nodo->izq);
```

```
        if (nodo->dato.esVillano) cout << "- " << nodo->dato.nombre << endl;
        listarVillanos(nodo->der);
    }

    int contarSuperheroes(NodoP* nodo) {
        if (!nodo) return 0;
        int c = (!nodo->dato.esVillano) ? 1 : 0;
        return c + contarSuperheroes(nodo->izq) + contarSuperheroes(nodo->der);
    }

    void modificarNombre(NodoP* nodo, string original, string correccion) {
        if(!nodo) return;
        if(nodo->dato.nombre == original) nodo->dato.nombre = correccion;
        modificarNombre(nodo->izq, original, correccion);
        modificarNombre(nodo->der, original, correccion);
    }
};

int main() {
    ArbolMarvel mcu;
    mcu.insertar({"Iron Man", false});
    mcu.insertar({"Thanos", true});
    mcu.insertar({"Capitan America", false});
    mcu.insertar({"Doctor Strnge", false}); // Error intencional
    mcu.insertar({"Loki", true});
```

```
    cout << "--- Villanos ---" << endl;
    mcu.listarVillanos(mcu.raiz);

    cout << "\n--- Cantidad de Superheroes ---" << endl;
    cout << mcu.contarSuperheroes(mcu.raiz) << endl;

    cout << "\n--- Corrigiendo Doctor Strange ---" << endl;
    mcu.modificarNombre(mcu.raiz, "Doctor Strnge", "Doctor Strange");
    // Listamos todo para verificar
    cout << "Arbol completo verificado (In-Orden):" << endl;
    // (Implementación rápida de listado para verificar)
    // En un caso real llamaríamos a un método 'mostrar'
    return 0;
}
```

6: Árbol de Jedis

```
// Archivo: Ejercicio6_Jedis.cpp
#include <iostream>
#include <string>

using namespace std;

struct Jedi {
    string nombre;
    int ranking;
```

```
    string especie;

    // Comparación por defecto (Nombre)
    bool operator<(const Jedi& j) const { return nombre < j.nombre; }
    bool operator>(const Jedi& j) const { return nombre > j.nombre; }
};

struct NodoJ {
    Jedi dato;
    NodoJ *izq, *der;
    NodoJ(Jedi j) : dato(j), izq(nullptr), der(nullptr) {}
};

class ArbolJedi {
public:
    NodoJ* raiz;
    ArbolJedi() : raiz(nullptr) {}

    void insertar(NodoJ*& nodo, Jedi j) {
        if (!nodo) nodo = new NodoJ(j);
        else if (j < nodo->dato) insertar(nodo->izq, j);
        else insertar(nodo->der, j);
    }

    void insertar(Jedi j) { insertar(raiz, j); }
```

```
    void buscarPorRanking(NodoJ* nodo, int rank) {
        if (!nodo) return;
        if (nodo->dato.ranking == rank)
            cout << "Encontrado: " << nodo->dato.nombre << " (" << nodo->dato.especie
<< ")" << endl;
        buscarPorRanking(nodo->izq, rank);
        buscarPorRanking(nodo->der, rank);
    }
};

int main() {
    ArbolJedi orden;
    orden.insertar({"Yoda", 1, "Desconocida"});
    orden.insertar({"Luke Skywalker", 2, "Humano"});
    orden.insertar({"Obi-Wan Kenobi", 3, "Humano"});
    orden.insertar({"Ahsoka Tano", 4, "Togruta"});

    cout << "Buscando Jedi con ranking 1:" << endl;
    orden.buscarPorRanking(orden.raiz, 1);

    return 0;
}
```

7: Sistema de Archivos (Transformada Knuth)

```
// Archivo: Ejercicio7_FileSystem.cpp
```

```
#include <iostream>

#include <string>



using namespace std;



struct NodoFS {

    string nombre;

    bool esCarpeta;

    int tamano;

    NodoFS *hijo, *hermano; // Punteros de Knuth



    NodoFS(string n, bool carpeta, int t) : nombre(n), esCarpeta(carpeta), tamano(t),
hijo(nullptr), hermano(nullptr) {}

};



class FileSystem {

    NodoFS* raiz;



    NodoFS* buscar(NodoFS* nodo, string nombre) {

        if (!nodo) return nullptr;

        if (nodo->nombre == nombre) return nodo;

        NodoFS* f = buscar(nodo->hijo, nombre);

        if (f) return f;

        return buscar(nodo->hermano, nombre);

    }
```

```
public:
    FileSystem(string root) { raiz = new NodoFS(root, true, 0); }

    void agregar(string carpetaPadre, string nombre, bool esCarpeta, int tam) {
        NodoFS* padre = buscar(raiz, carpetaPadre);
        if (!padre || !padre->esCarpeta) return;

        NodoFS* nuevo = new NodoFS(nombre, esCarpeta, tam);
        if (!padre->hijo) padre->hijo = nuevo;
        else {
            NodoFS* aux = padre->hijo;
            while(aux->hermano) aux = aux->hermano;
            aux->hermano = nuevo;
        }
    }

    void listar(NodoFS* nodo, int indent) {
        if (!nodo) return;
        for(int i=0; i<indent; i++) cout << "  ";
        cout << (nodo->esCarpeta ? "[D] " : "[F] ") << nodo->nombre << " (" << nodo->tamano << "kb)" << endl;
        listar(nodo->hijo, indent + 1);
        listar(nodo->hermano, indent);
    }
```

```
    NodoFS* getRaiz() { return raiz; }
};

int main() {
    FileSystem fs("/");
    fs.agregar("/", "home", true, 0);
    fs.agregar("/", "bin", true, 0);
    fs.agregar("home", "user", true, 0);
    fs.agregar("user", "doc.txt", false, 1500);
    fs.agregar("bin", "ls", false, 20);

    cout << "Estructura de Archivos:" << endl;
    fs.listar(fs.getRaiz(), 0);
    return 0;
}
```

8: Obtener Mínimo y Máximo

```
// Archivo: Ejercicio8_MinMax.cpp
#include <iostream>
using namespace std;

struct Nodo {
    int dato;
    Nodo *izq, *der;
    Nodo(int d) : dato(d), izq(nullptr), der(nullptr) {}
```

```
};

class ABB_MinMax {
    Nodo* raiz;

    void insertar(Nodo*& nodo, int v) {
        if(!nodo) nodo = new Nodo(v);
        else if(v < nodo->dato) insertar(nodo->izq, v);
        else insertar(nodo->der, v);
    }

public:
    ABB_MinMax() : raiz(nullptr) {}
    void insertar(int v) { insertar(raiz, v); }

    int obtenerMinimo() {
        if (!raiz) throw runtime_error("Arbol vacio");
        Nodo* aux = raiz;
        while(aux->izq) aux = aux->izq;
        return aux->dato;
    }

    int obtenerMaximo() {
        if (!raiz) throw runtime_error("Arbol vacio");
        Nodo* aux = raiz;
```

```
        while(aux->der) aux = aux->der;
        return aux->dato;
    }
};

int main() {
    ABB_MinMax abb;
    abb.insertar(20); abb.insertar(10); abb.insertar(30);
    abb.insertar(5); abb.insertar(40);

    cout << "Minimo: " << abb.obtenerMinimo() << endl;
    cout << "Maximo: " << abb.obtenerMaximo() << endl;
    return 0;
}
```

9: Compresión Huffman (Resistencia)

```
// Archivo: Ejercicio9_Huffman.cpp
#include <iostream>
#include <string>

using namespace std;

struct NodoH {
    char car;
    int freq;
    NodoH *izq, *der;
```

```
    NodoH *sig; // Para lista enlazada (Priority Queue)


    NodoH(char c, int f) : car(c), freq(f), izq(nullptr), der(nullptr), sig(nullptr) {}
    NodoH(int f, NodoH* i, NodoH* d) : car('\0'), freq(f), izq(i), der(d), sig(nullptr) {}
};


class PriorityQueue {
    NodoH* head;
public:
    PriorityQueue() : head(nullptr) {}
    void push(NodoH* n) {
        if (!head || n->freq < head->freq) { n->sig = head; head = n; }
        else {
            NodoH* temp = head;
            while(temp->sig && temp->sig->freq <= n->freq) temp = temp->sig;
            n->sig = temp->sig;
            temp->sig = n;
        }
    }
    NodoH* pop() {
        if (!head) return nullptr;
        NodoH* aux = head;
        head = head->sig;
        return aux;
    }
```

```
    int size() {
        int c=0; NodoH* t=head;
        while(t){ c++; t=t->sig; }
        return c;
    }
};

void imprimirCodigos(NodoH* raiz, string str) {
    if (!raiz) return;
    if (raiz->car != '\0') cout << raiz->car << ": " << str << endl;
    imprimirCodigos(raiz->izq, str + "0");
    imprimirCodigos(raiz->der, str + "1");
}

int main() {
    string msg = "que la fuerza te acompañe";
    int freq[256] = {0};
    for(char c : msg) freq[(int)c]++;

    PriorityQueue pq;
    for(int i=0; i<256; i++)
        if(freq[i] > 0) pq.push(new NodoH((char)i, freq[i]));

    while(pq.size() > 1) {
        NodoH *izq = pq.pop();
```

```
        NodoH *der = pq.pop();
        pq.push(new NodoH(izq->freq + der->freq, izq, der));
    }

    cout << "Codigos Huffman generados:" << endl;
    imprimirCodigos(pq.pop(), "");
    return 0;
}
```

10: Nodos por Nivel y Completitud

```
// Archivo: Ejercicio10_NivelCompleto.cpp
#include <iostream>
using namespace std;

struct Nodo {
    int dato;
    Nodo *izq, *der;
    Nodo(int d) : dato(d), izq(nullptr), der(nullptr) {}
};

class ABB_Nivel {
    Nodo* raiz;

    void insertar(Nodo*& n, int v) {
        if(!n) n = new Nodo(v);
        else if(v < n->dato) insertar(n->izq, v);
```

```
        else insertar(n->der, v);
    }

    int contarEnNivel(Nodo* n, int actual, int objetivo) {
        if (!n) return 0;
        if (actual == objetivo) return 1;
        return contarEnNivel(n->izq, actual+1, objetivo) + contarEnNivel(n->der,
actual+1, objetivo);
    }

public:
    ABB_Nivel() : raiz(nullptr) {}
    void insertar(int v) { insertar(raiz, v); }

    void analizarNivel(int nivel) {
        int cantidad = contarEnNivel(raiz, 0, nivel);
        int maxPosible = 1 << nivel; // 2^nivel
        cout << "Nivel " << nivel << ": " << cantidad << " nodos. ";
        cout << (cantidad == maxPosible ? "(Completo)" : "(Incompleto)") << endl;
    }
};

int main() {
    ABB_Nivel abb;
    abb.insertar(10); // Nivel 0
```

```
    abb.insertar(5);  // Nivel 1
    abb.insertar(15); // Nivel 1
    abb.insertar(2);  // Nivel 2

    abb.analizarNivel(0);
    abb.analizarNivel(1);
    abb.analizarNivel(2); // Deberia ser incompleto (falta 1 nodo)
    return 0;
}
```

11 (6.8 del libro): Árbol Rojinegro Insertar 10..100

```
// Archivo: Ejercicio6_8_RB.cpp
#include <iostream>
using namespace std;

enum Color { ROJO, NEGRO };

struct NodoRB {
    int dato;
    Color color;
    NodoRB *izq, *der, *padre;
    NodoRB(int v) : dato(v), color(ROJO), izq(nullptr), der(nullptr), padre(nullptr) {}
};

class ArbolRB {
    NodoRB* raiz;
```

```
void rotarIzq(NodoRB*& raizPtr, NodoRB* x) {
    NodoRB* y = x->der;
    x->der = y->izq;
    if(y->izq) y->izq->padre = x;
    y->padre = x->padre;
    if(!x->padre) raizPtr = y;
    else if(x == x->padre->izq) x->padre->izq = y;
    else x->padre->der = y;
    y->izq = x;
    x->padre = y;
}

void rotarDer(NodoRB*& raizPtr, NodoRB* y) {
    NodoRB* x = y->izq;
    y->izq = x->der;
    if(x->der) x->der->padre = y;
    x->padre = y->padre;
    if(!y->padre) raizPtr = x;
    else if(y == y->padre->izq) y->padre->izq = x;
    else y->padre->der = x;
    x->der = y;
    y->padre = x;
}
```

```
void arreglarInsercion(NodoRB*& raizPtr, NodoRB* z) {
    while(z->padre && z->padre->color == ROJO) {
        if(z->padre == z->padre->padre->izq) {
            NodoRB* y = z->padre->padre->der;
            if(y && y->color == ROJO) {
                z->padre->color = NEGRO;
                y->color = NEGRO;
                z->padre->padre->color = ROJO;
                z = z->padre->padre;
            } else {
                if(z == z->padre->der) {
                    z = z->padre;
                    rotarIzq(raizPtr, z);
                }
                z->padre->color = NEGRO;
                z->padre->padre->color = ROJO;
                rotarDer(raizPtr, z->padre->padre);
            }
        } else {
            NodoRB* y = z->padre->padre->izq;
            if(y && y->color == ROJO) {
                z->padre->color = NEGRO;
                y->color = NEGRO;
                z->padre->padre->color = ROJO;
                z = z->padre->padre;
```

```
            } else {
                if(z == z->padre->izq) {
                    z = z->padre;
                    rotarDer(raizPtr, z);
                }
                z->padre->color = NEGRO;
                z->padre->padre->color = ROJO;
                rotarIzq(raizPtr, z->padre->padre);
            }
        }
    }
    raizPtr->color = NEGRO;
}

public:
    ArbolRB() : raiz(nullptr) {}

    void insertar(int valor) {
        NodoRB* z = new NodoRB(valor);
        NodoRB* y = nullptr;
        NodoRB* x = raiz;
        while(x) {
            y = x;
            if(z->dato < x->dato) x = x->izq;
            else x = x->der;
```

```
        }
        z->padre = y;
        if(!y) raiz = z;
        else if(z->dato < y->dato) y->izq = z;
        else y->der = z;

        arreglarInsercion(raiz, z);
    }

    void inOrden(NodoRB* n) {
        if(!n) return;
        inOrden(n->izq);
        cout << n->dato << "(" << (n->color==ROJO?"R":"N") << ") ";
        inOrden(n->der);
    }

    void mostrar() { inOrden(raiz); cout << endl; }
};

int main() {
    ArbolRB rb;
    cout << "Insertando 10, 20... 100" << endl;
    for(int i=10; i<=100; i+=10) rb.insertar(i);

    cout << "Arbol resultante (InOrden con colores):" << endl;
```

```
    rb.mostrar();

    return 0;

}
```