Nombre: Isabela Zambrano, Santiago Gallardo **NRC:** 29852

Fecha: 17/12/2025

1. Para un ABB, realizar dos funciones tal que:
   a. Se encuentre el nodo más pequeño
   b. Se encuentre el nodo más grande

```
// Método para buscar el nodo con el valor mayor (extremo derecho)

template<typename T>

Nodo<T>* ABB<T>::buscarMayor() const {

    if (Raiz == nullptr) return nullptr;

    Nodo<T>* actual = Raiz;

    while (actual->getDerecha() != nullptr) {

        actual = actual->getDerecha();

    }

    return actual;

}

// Método para buscar el nodo con el valor menor (extremo izquierdo)

template<typename T>

Nodo<T>* ABB<T>::buscarMenor() const {

    if (Raiz == nullptr) return nullptr;

    Nodo<T>* actual = Raiz;

    while (actual->getIzquierda() != nullptr) {

        actual = actual->getIzquierda();

    }

    return actual;

}
```

2. Diseñar e implementar un árbol B+ con sus operaciones básicas:
    a. Insertar
    b. Buscar
    c. Imprimir

```
#include "ArbolBMas.hpp"
#include <iostream>

template<typename T>
ArbolBMas<T>::ArbolBMas(int orden) {
   this->raiz = nullptr;
   this->orden = orden;
}

template<typename T>
ArbolBMas<T>::~ArbolBMas() {
   // Implementación simple de destructor (idealmente recursivo)
   // Aquí se omite por brevedad, pero debería eliminar todos los nodos.
}

// Busca el nodo hoja donde debería estar la clave
template<typename T>
NodoB<T>* ArbolBMas<T>::buscarHoja(T clave) {
   if (raiz == nullptr) return nullptr;

   NodoB<T>* cursor = raiz;
   while (!cursor->esHoja) {
      int i = 0;
      while (i < cursor->n && clave >= cursor->claves[i]) {
         i++;
      }
      cursor = cursor->hijos[i];
   }
   return cursor;
}

template<typename T>
bool ArbolBMas<T>::buscar(T clave) {
   if (raiz == nullptr) return false;
   NodoB<T>* hoja = buscarHoja(clave);

   for (int i = 0; i < hoja->n; i++) {
      if (hoja->claves[i] == clave) return true;
   }
   return false;
}

template<typename T>
void ArbolBMas<T>::insertar(T clave) {
   // 1. Si el árbol está vacío
```

```
    if (raiz == nullptr) {
        raiz = new NodoB<T>(orden, true);
        raiz->claves[0] = clave;
        raiz->n = 1;
        return;
    }

    // 2. Buscar la hoja adecuada
    NodoB<T>* hoja = buscarHoja(clave);

    // 3. Insertar ordenado en la hoja
    if (hoja->n < orden - 1) {
        int i = 0;
        while (i < hoja->n && clave > hoja->claves[i]) i++;

        for (int j = hoja->n; j > i; j--) {
            hoja->claves[j] = hoja->claves[j - 1];
        }
        hoja->claves[i] = clave;
        hoja->n++;
    } else {
        // 4. La hoja está llena, dividir (Split)
        NodoB<T>* nuevaHoja = new NodoB<T>(orden, true);
        T buffer[orden + 1]; // Buffer temporal

        int i = 0;
        while (i < orden - 1 && clave > hoja->claves[i]) i++;

        // Copiar claves al buffer insertando la nueva
        for(int j=0; j<i; j++) buffer[j] = hoja->claves[j];
        buffer[i] = clave;
        for(int j=i; j<orden-1; j++) buffer[j+1] = hoja->claves[j];

        // Distribuir claves
        hoja->n = (orden) / 2;
        nuevaHoja->n = orden - (orden) / 2;

        for(int j=0; j<hoja->n; j++) hoja->claves[j] = buffer[j];
        for(int j=0; j<nuevaHoja->n; j++) nuevaHoja->claves[j] = buffer[j + hoja->n];

        // Enlazar hojas
        nuevaHoja->siguiente = hoja->siguiente;
        hoja->siguiente = nuevaHoja;

        // Insertar la clave media en el padre
        insertarEnPadre(hoja, nuevaHoja->claves[0], nuevaHoja);
    }
}

template<typename T>
```

```
void ArbolBMas<T>::insertarEnPadre(NodoB<T>* n, T clave, NodoB<T>* nuevoNodo) {
    if (n == raiz) {
        NodoB<T>* nuevaRaiz = new NodoB<T>(orden, false);
        nuevaRaiz->claves[0] = clave;
        nuevaRaiz->hijos[0] = n;
        nuevaRaiz->hijos[1] = nuevoNodo;
        nuevaRaiz->n = 1;
        raiz = nuevaRaiz;
        return;
    }

    NodoB<T>* padre = encontrarPadre(raiz, n);

    if (padre->n < orden - 1) {
        int i = 0;
        while (i < padre->n && clave > padre->claves[i]) i++;

        for (int j = padre->n; j > i; j--) {
            padre->claves[j] = padre->claves[j - 1];
            padre->hijos[j + 1] = padre->hijos[j];
        }
        padre->claves[i] = clave;
        padre->hijos[i + 1] = nuevoNodo;
        padre->n++;
    } else {
        // División de nodo interno
        NodoB<T>* nuevoInterno = new NodoB<T>(orden, false);
        T bufferClaves[orden + 1];
        NodoB<T>* bufferHijos[orden + 2];

        int i = 0;
        while (i < padre->n && clave > padre->claves[i]) i++;

        // Copiar datos al buffer
        for(int j=0; j<i; j++) bufferClaves[j] = padre->claves[j];
        bufferClaves[i] = clave;
        for(int j=i; j<padre->n; j++) bufferClaves[j+1] = padre->claves[j];

        for(int j=0; j<=i; j++) bufferHijos[j] = padre->hijos[j];
        bufferHijos[i+1] = nuevoNodo;
        for(int j=i+1; j<=padre->n; j++) bufferHijos[j+1] = padre->hijos[j];

        // Dividir
        int splitIndex = orden / 2; // Índice de la clave que sube
        T claveArriba = bufferClaves[splitIndex];

        padre->n = splitIndex;
        nuevoInterno->n = orden - splitIndex - 1; // -1 porque una clave sube

        for(int j=0; j<padre->n; j++) padre->claves[j] = bufferClaves[j];
```

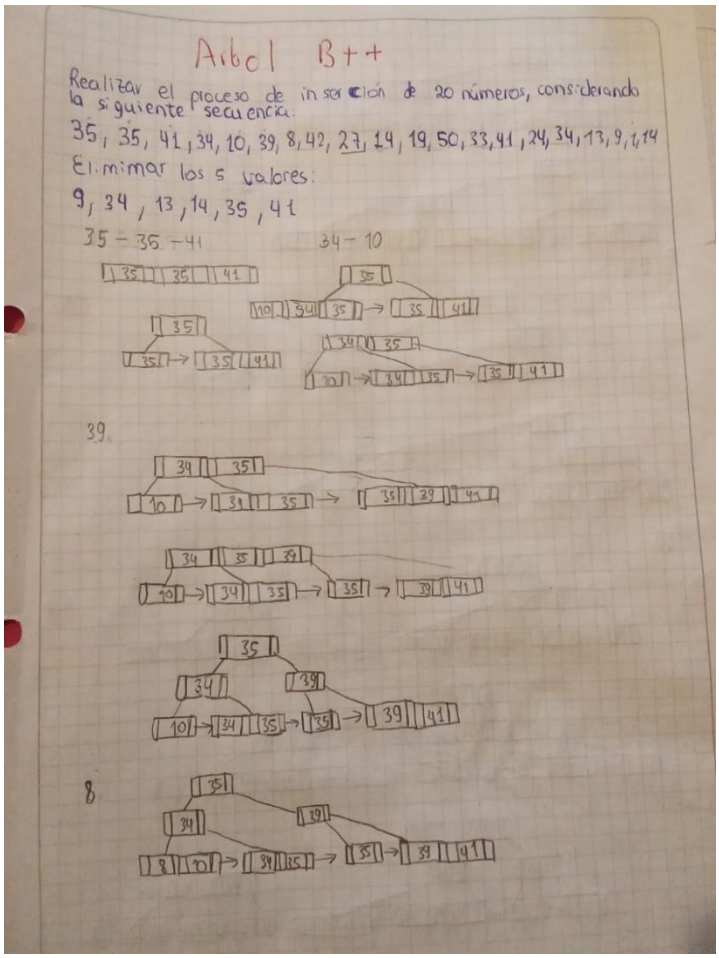
```
        for(int j=0; j<=padre->n; j++) padre->hijos[j] = bufferHijos[j];

        for(int j=0; j<nuevoInterno->n; j++) nuevoInterno->claves[j] = bufferClaves[j + splitIndex +
1];
        for(int j=0; j<=nuevoInterno->n; j++) nuevoInterno->hijos[j] = bufferHijos[j + splitIndex + 1];

        insertarEnPadre(padre, claveArriba, nuevoInterno);
    }
}

template<typename T>
NodoB<T>* ArbolBMas<T>::encontrarPadre(NodoB<T>* cursor, NodoB<T>* hijo) {
    if (cursor->esHoja || cursor->hijos[0]->esHoja) {
        // Si los hijos son hojas, verificamos si alguno es el hijo buscado
        // Pero en B+, navegamos por claves.
        // Manera simple: recorrer hijos
        for(int i=0; i<=cursor->n; i++){
            if(cursor->hijos[i] == hijo) return cursor;
        }
    }

    // Navegación estándar
    if (cursor->esHoja) return nullptr;

    for (int i = 0; i <= cursor->n; i++) {
        if (cursor->hijos[i] == hijo) return cursor; // Encontrado directo

        // Si no, buscar recursivamente en el hijo adecuado
        // Nota: Esta lógica simplificada asume que podemos bajar.
        // Para una implementación robusta de encontrarPadre se suele usar la clave del hijo.
        T claveHijo = hijo->claves[0];
        if (i == cursor->n || claveHijo < cursor->claves[i]) {
            NodoB<T>* res = encontrarPadre(cursor->hijos[i], hijo);
            if(res != nullptr) return res;
        }
    }
    return nullptr;
}

template<typename T>
void ArbolBMas<T>::imprimirListaEnlazada() {
    NodoB<T>* cursor = raiz;
    if(cursor == nullptr) return;
    while(!cursor->esHoja) cursor = cursor->hijos[0];

    std::cout << "Lista de Hojas: ";
    while(cursor != nullptr){
        for(int i=0; i<cursor->n; i++) std::cout << cursor->claves[i] << " ";
        std::cout << " -> ";
        cursor = cursor->siguiente;
```

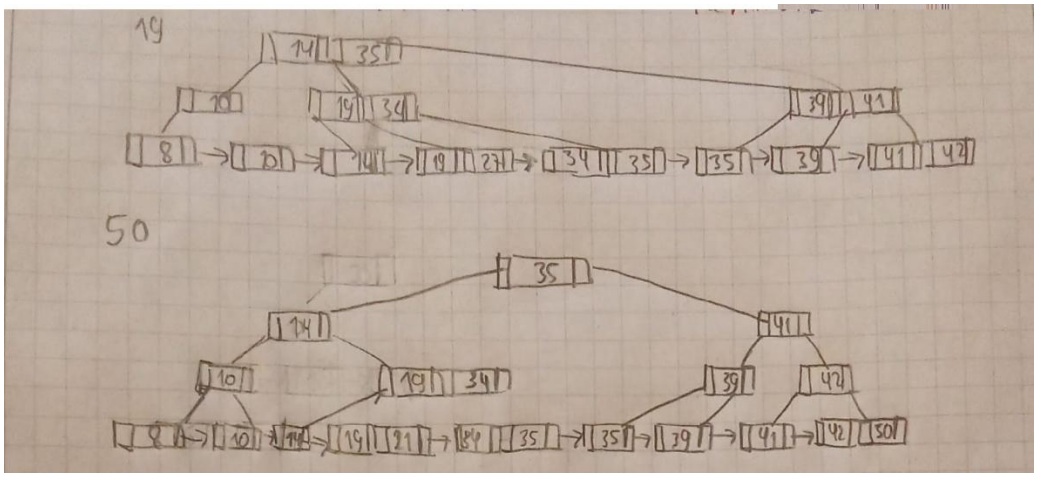
```
    }
    std::cout << "NULL" << std::endl;
}

template<typename T>
void ArbolBMas<T>::imprimir() {
    imprimirListaEnlazada();
}
```

3. Para un **árbol B+**, realizar el proceso de **inserción** de **20 números**, considerando la siguiente secuencia: **35, 35, 41, 34, 10, 39, 8, 42, 27, 14, 19, 50, 33, 41, 24, 34, 13, 9, 1, 14** Posteriormente, efectuar la **eliminación de 5 valores**, tomados del siguiente conjunto: **9, 34, 13, 14, 35, 41**

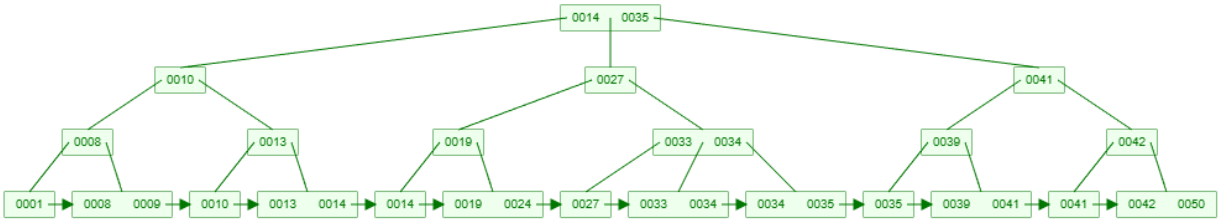
El desarrollo debe contemplar:

a. La **inserción paso a paso**, mostrando la estructura del árbol B+ después de cada operación, como se realizaría manualmente.
b. La **eliminación paso a paso**, indicando los casos de redistribución o fusión de nodos cuando sea necesario.
c. La **implementación en código** del árbol B+ que permita realizar dichas operaciones.

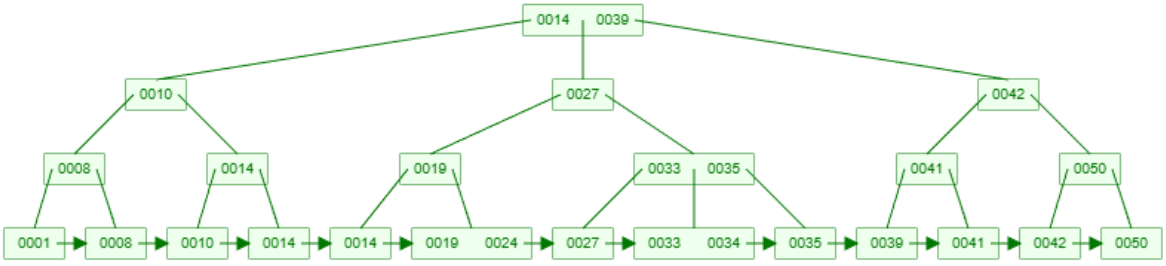




Resultado final:



Eliminación




```
#include <iostream>

#include <algorithm>


using namespace std;


const int ORDEN = 3;
```

```
/* ========================
   NODO DEL ÁRBOL B+
   ======================== */
class Nodo {
public:
    bool hoja;
    int claves[ORDEN + 1];
    Nodo* hijos[ORDEN + 2];
    int cuenta;
    Nodo* siguiente;

    Nodo(bool esHoja) {
        hoja = esHoja;
        cuenta = 0;
        siguiente = nullptr;
        for (int i = 0; i < ORDEN + 2; i++)
            hijos[i] = nullptr;
    }
};

/* ========================
   ÁRBOL B+
   ======================== */
class ArbolBPlus {
```

```
private:
    Nodo* raiz;


    /* Buscar hoja */
    Nodo* buscarHoja(int valor) {
        Nodo* actual = raiz;
        while (!actual->hoja) {
            int i = 0;
            while (i < actual->cuenta && valor >= actual->claves[i])
                i++;
            actual = actual->hijos[i];
        }
        return actual;
    }


    /* Insertar en hoja */
    void insertarEnHoja(Nodo* hoja, int valor) {
        int i = hoja->cuenta - 1;
        while (i >= 0 && hoja->claves[i] > valor) {
            hoja->claves[i + 1] = hoja->claves[i];
            i--;
        }
        hoja->claves[i + 1] = valor;
        hoja->cuenta++;
    }
```

```
/* Dividir hoja */
void dividirHoja(Nodo* hoja) {
    int mitad = (ORDEN + 1) / 2;
    Nodo* nuevaHoja = new Nodo(true);

    nuevaHoja->cuenta = hoja->cuenta - mitad;

    for (int i = 0; i < nuevaHoja->cuenta; i++)
        nuevaHoja->claves[i] = hoja->claves[mitad + i];

    hoja->cuenta = mitad;

    nuevaHoja->siguiente = hoja->siguiente;
    hoja->siguiente = nuevaHoja;

    insertarEnPadre(hoja, nuevaHoja->claves[0], nuevaHoja);
}

/* Insertar en nodo padre */
void insertarEnPadre(Nodo* izquierda, int clave, Nodo* derecha) {
    if (izquierda == raiz) {
        Nodo* nuevaRaiz = new Nodo(false);
        nuevaRaiz->claves[0] = clave;
        nuevaRaiz->hijos[0] = izquierda;
```

```
        nuevaRaiz->hijos[1] = derecha;
        nuevaRaiz->cuenta = 1;
        raiz = nuevaRaiz;
        return;
    }

    Nodo* padre = encontrarPadre(raiz, izquierda);

    int i = padre->cuenta - 1;
    while (i >= 0 && padre->claves[i] > clave) {
        padre->claves[i + 1] = padre->claves[i];
        padre->hijos[i + 2] = padre->hijos[i + 1];
        i--;
    }

    padre->claves[i + 1] = clave;
    padre->hijos[i + 2] = derecha;
    padre->cuenta++;

    if (padre->cuenta > ORDEN)
        dividirInterno(padre);
}

/* Dividir nodo interno */
void dividirInterno(Nodo* nodo) {
```

```
    int mitad = ORDEN / 2;
    Nodo* nuevo = new Nodo(false);

    int claveSube = nodo->claves[mitad];
    nuevo->cuenta = nodo->cuenta - mitad - 1;

    for (int i = 0; i < nuevo->cuenta; i++)
        nuevo->claves[i] = nodo->claves[mitad + 1 + i];

    for (int i = 0; i <= nuevo->cuenta; i++)
        nuevo->hijos[i] = nodo->hijos[mitad + 1 + i];

    nodo->cuenta = mitad;

    insertarEnPadre(nodo, claveSube, nuevo);
}

/* Encontrar padre */
Nodo* encontrarPadre(Nodo* actual, Nodo* hijo) {
    if (actual->hoja)
        return nullptr;

    for (int i = 0; i <= actual->cuenta; i++) {
        if (actual->hijos[i] == hijo)
            return actual;
```

```
            Nodo* res = encontrarPadre(actual->hijos[i], hijo);
            if (res != nullptr)
                return res;
        }
        return nullptr;
    }

public:
    ArbolBPlus() {
        raiz = new Nodo(true);
    }

    /* INSERCIÓN */
    void insertar(int valor) {
        Nodo* hoja = buscarHoja(valor);
        insertarEnHoja(hoja, valor);

        if (hoja->cuenta > ORDEN)
            dividirHoja(hoja);
    }

    /* ELIMINACIÓN (simplificada) */
    void eliminar(int valor) {
        Nodo* hoja = buscarHoja(valor);
```

```
    int i = 0;

    while (i < hoja->cuenta && hoja->claves[i] != valor)
        i++;

    if (i == hoja->cuenta)
        return;

    for (int j = i; j < hoja->cuenta - 1; j++)
        hoja->claves[j] = hoja->claves[j + 1];

    hoja->cuenta--;
}

/* MOSTRAR HOJAS */
void mostrar() {
    Nodo* actual = raiz;
    while (!actual->hoja)
        actual = actual->hijos[0];

    cout << "Hojas: ";
    while (actual != nullptr) {
        for (int i = 0; i < actual->cuenta; i++)
            cout << actual->claves[i] << " ";
        actual = actual->siguiente;
```

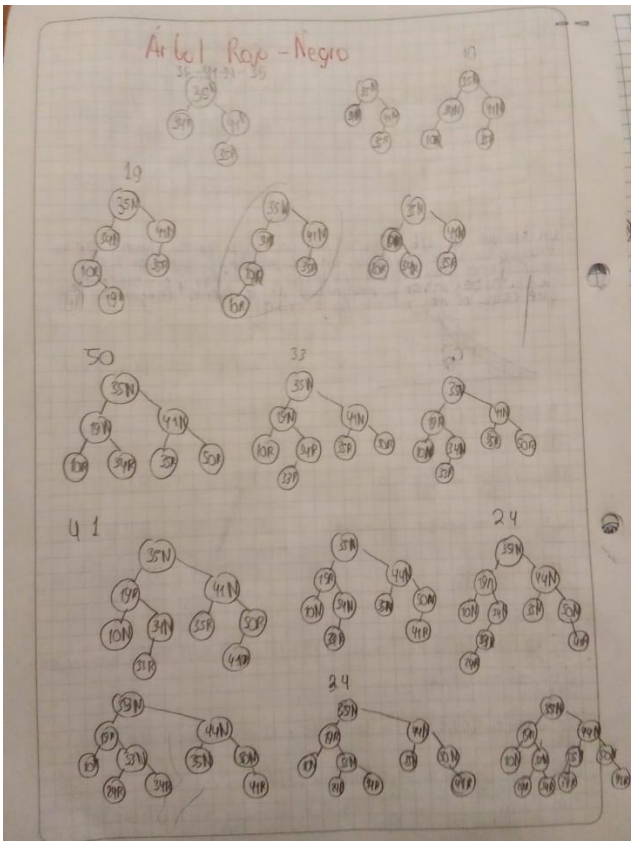
```
        }
        cout << endl;
    }
};

/* ========================
   PROGRAMA PRINCIPAL
   ======================== */
int main() {
    ArbolBPlus arbol;

    int insertarValores[20] = {
        35, 35, 41, 34, 10, 39, 8, 42, 27, 14,
        19, 50, 33, 41, 24, 34, 13, 9, 1, 14
    };

    int eliminarValores[5] = {9, 34, 13, 14, 35};

    cout << "INSERCIONES:\n";
    for (int i = 0; i < 20; i++) {
        arbol.insertar(insertarValores[i]);
        arbol.mostrar();
    }

    cout << "\nELIMINACIONES:\n";
```

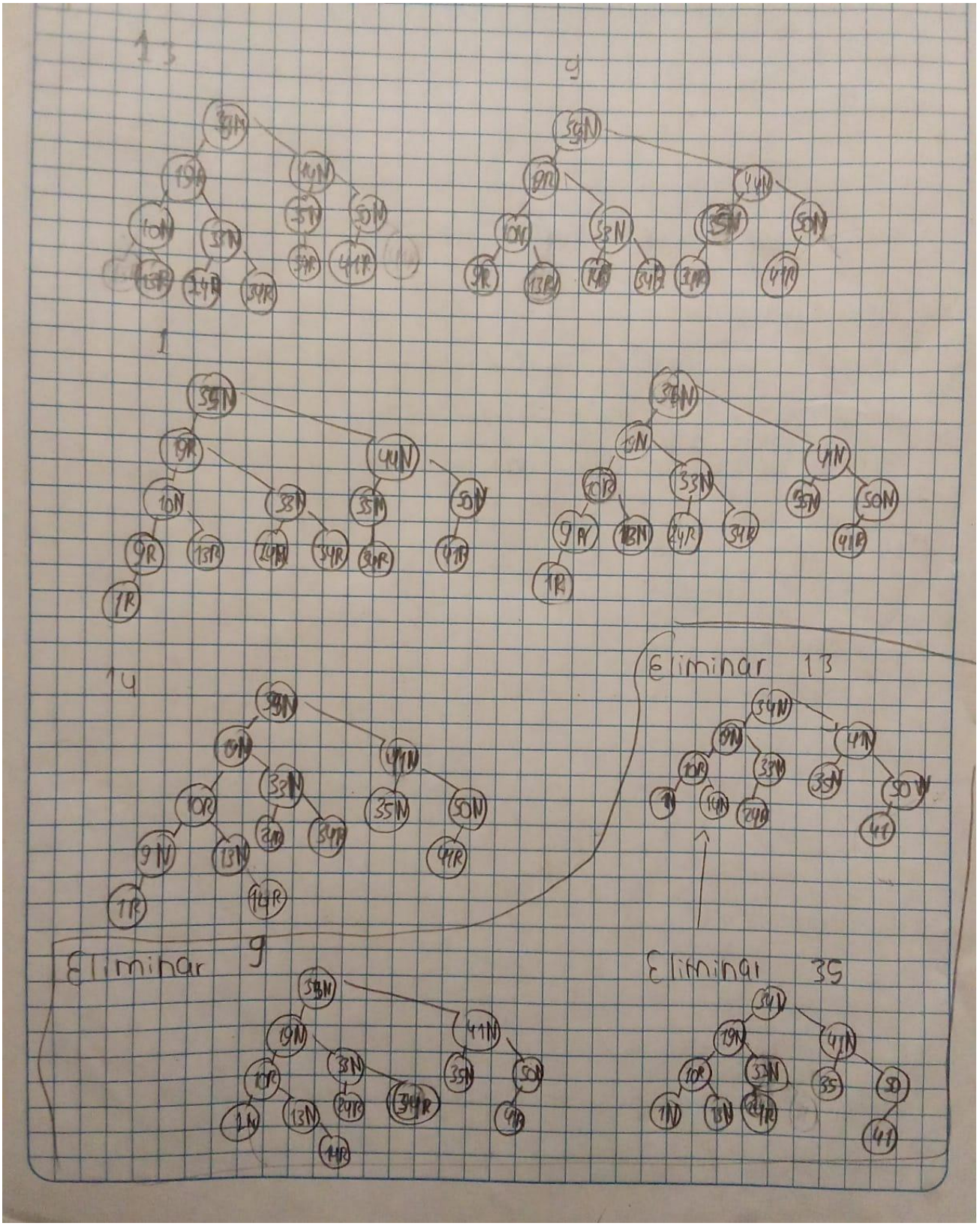
```
    for (int i = 0; i < 5; i++) {

        arbol.eliminar(eliminarValores[i]);

        arbol.mostrar();

    }



    return 0;

}
```

4. Para un **árbol rojo y negro**, realizar el proceso de **inserción** de **15 números**, considerando la siguiente secuencia: **35, 41,34,35, 10,19, 50, 33, 41, 24, 34, 13, 9, 1, 14** Posteriormente, efectuar la **eliminación de 3 valores**, tomados del siguiente conjunto: **9, 34, 13**

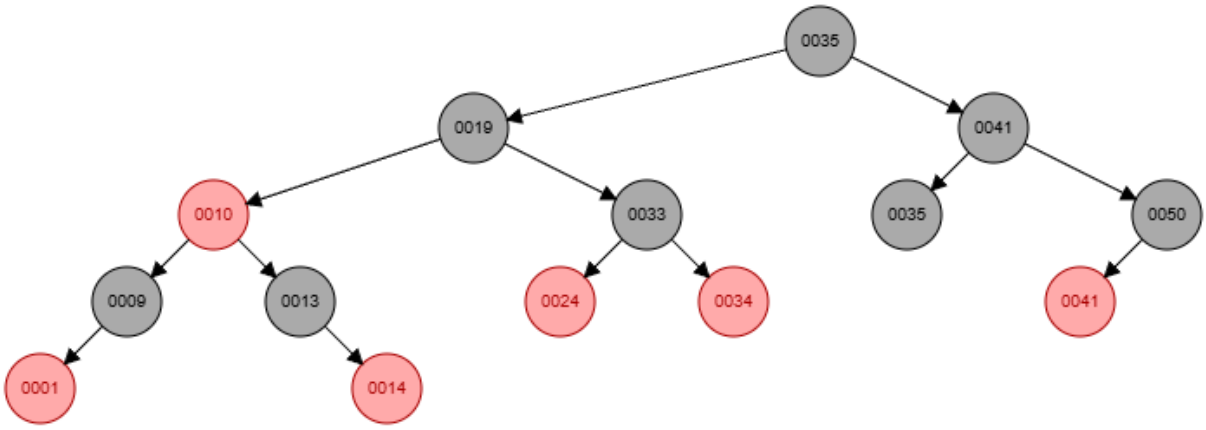
El desarrollo debe contemplar:

a. La **inserción paso a paso**, mostrando la estructura del árbol B+ después de cada operación, como se realizaría manualmente.
b. La **eliminación paso a paso**, indicando los casos de redistribución o fusión de nodos cuando sea necesario.

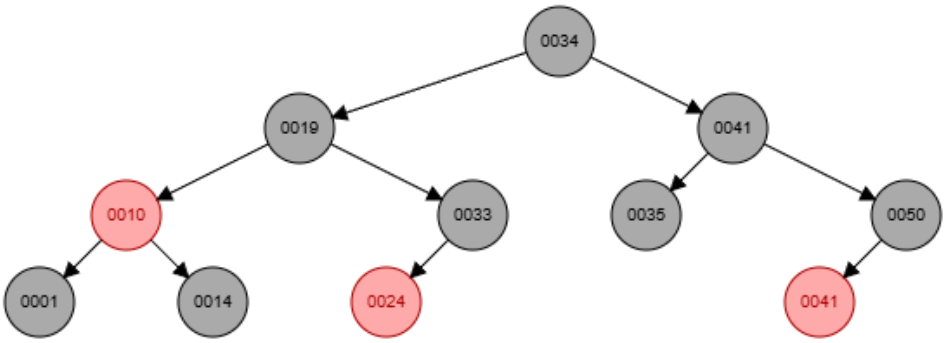



13
9
1
14
Eliminar 13
Eliminar 9
Eliminar 35

INSERTAR FINAL:



ELIMINAR FINAL:



5. Se cuenta con un archivo directo llamado ALUMNOS. DAT con información de alumnos de una universidad. El archivo se abre para lectura y se indexa por medio de su clave primaria (#legajo). El índice se mantiene en memoria principal. Se ofrece la posibilidad al usuario de buscar alumnos por legajo. La búsqueda binaria se aplica sobre el índice y se accede en forma directa (seek) al archivo para recuperar el registro. Se asume que el archivo está ordenado de manera ascendente por su clave primaria (#legajo). Si éste no fuese el caso, el índice debería ordenarse antes de aplicar una búsqueda binaria sobre él. Aclaración: no se provee el archivo ALUMNOS. DAT (debido a que es binario). Deberá ser generado antes de la ejecución de este programa)

```
#include <iostream>
#include <fstream>
using namespace std;

struct Alumno {
```

```
    int legajo;
    char nombre[50];
    char carrera[30];
    float promedio;
};

struct RegistroIndice {
    int legajo;
    streampos posicion;
};

void generarArchivo(const char* nombreArchivo) {
    fstream archivo(nombreArchivo, ios::binary | ios::out);

    if (!archivo) {
        cout << "Error al crear archivo.\n";
        return;
    }

    // Datos de prueba (ordenados por legajo)
    Alumno a1 = {101, "Ana_Garcia", "Ingenieria", 8.5};
    Alumno a2 = {105, "Luis_Perez", "Arquitectura", 7.2};
    Alumno a3 = {204, "Maria_Lopez", "Medicina", 9.1};
    Alumno a4 = {300, "Carlos_Ruiz", "Derecho", 6.8};
    Alumno a5 = {450, "Sofia_Mendieta", "Sistemas", 9.5};

    archivo.write((char*)&a1, sizeof(Alumno));
    archivo.write((char*)&a2, sizeof(Alumno));
    archivo.write((char*)&a3, sizeof(Alumno));
    archivo.write((char*)&a4, sizeof(Alumno));
    archivo.write((char*)&a5, sizeof(Alumno));

    archivo.close();
    cout << "[INFO] Archivo generado correctamente.\n";
}

// -----------------
// BÚSQUEDA BINARIA
// -----------------
int busquedaBinaria(RegistroIndice* base, int cantidad, int legajoBuscado) {

    int lo = 0;
    int hi = cantidad - 1;

    while (lo <= hi) {
        int mid = (lo + hi) / 2;

        int leg = (*(base + mid)).legajo;

        if (leg == legajoBuscado)
```

```
            return mid;

        if (leg < legajoBuscado)
            lo = mid + 1;
        else
            hi = mid - 1;
    }

    return -1;
}

int main() {

    const char* ARCHIVO = "ALUMNOS.DAT";

    generarArchivo(ARCHIVO);


    fstream archivo(ARCHIVO, ios::binary | ios::in);
    if (!archivo.is_open()) {
        cout << "Error al abrir archivo.\n";
        return 1;
    }

    archivo.seekg(0, ios::end);
    long bytes = archivo.tellg();
    int cantidad = bytes / sizeof(Alumno);

    archivo.seekg(0, ios::beg);

    cout << "\nRegistros encontrados: " << cantidad << "\n";


    RegistroIndice* indice = new RegistroIndice[cantidad];

    Alumno temp;
    for (int i = 0; i < cantidad; i++) {
        streampos posActual = archivo.tellg();

        archivo.read((char*)&temp, sizeof(Alumno));

        (*(indice + i)).legajo = temp.legajo;
        (*(indice + i)).posicion = posActual;
    }

    archivo.clear();
    cout << "\n--- BUSQUEDA DE ALUMNOS POR LEGAJO ---\n";
    int legajoBuscado;

    cout << "Ingrese legajo a buscar (0 para salir): ";
```

```
    while (cin >> legajoBuscado && legajoBuscado != 0) {

        int posIndice = busquedaBinaria(indice, cantidad, legajoBuscado);

        if (posIndice == -1) {
            cout << "[INFO] No existe ese legajo.\n\n";
        } else {
            // Recuperar posición del archivo
            streampos offset = (*(indice + posIndice)).posicion;

            archivo.seekg(offset);

            Alumno encontrado;
            archivo.read((char*)&encontrado, sizeof(Alumno));

            cout << "\n===== REGISTRO ENCONTRADO =====\n";
            cout << "Legajo:   " << encontrado.legajo << "\n";
            cout << "Nombre:   " << encontrado.nombre << "\n";
            cout << "Carrera:  " << encontrado.carrera << "\n";
            cout << "Promedio: " << encontrado.promedio << "\n";
            cout << "================================\n\n";
        }

        cout << "Ingrese otro legajo (0 para salir): ";
    }

    delete[] indice;
    archivo.close();

    return 0;
}
```

6. Localice un entero en un arreglo usando una versión recursiva del algoritmo de búsqueda binaria. El número que se debe buscar (clave) se debe ingresar como argumento por línea de comandos. El arreglo de datos debe estar previamente ordenado (requisito de la búsqueda binaria). Comparar esta versión recursiva de la búsqueda binaria con la versión iterativa dada en el capítulo 5.

```
#include <iostream>
#include <cstdlib>
using namespace std;

// --------------------------
// BÚSQUEDA BINARIA RECURSIVA
// --------------------------
int busquedaBinariaRec(int* inicio, int* fin, int clave) {
```

```
    if (inicio > fin)
        return -1;

    int* medio = inicio + (fin - inicio) / 2;

    if (*medio == clave)
        return medio - inicio;

    if (*medio > clave)
        return busquedaBinariaRec(inicio, medio - 1, clave);

    return busquedaBinariaRec(medio + 1, fin, clave);
}

// --------------------------
// BÚSQUEDA BINARIA ITERATIVA
// --------------------------
int busquedaBinariaIter(int* inicio, int* fin, int clave) {

    int* lo = inicio;
    int* hi = fin;

    while (lo <= hi) {

        int* mid = lo + (hi - lo) / 2;

        if (*mid == clave)
            return mid - inicio;

        if (*mid < clave)
            lo = mid + 1;
        else
            hi = mid - 1;
    }

    return -1;
}

int main(int argc, char* argv[]) {

    if (argc < 2) {
        cout << "Uso: programa <clave_a_buscar>\n";
        return 1;
    }

    int clave = atoi(argv[1]);

    // Arreglo ORDENADO
    int datos[] = {1, 7, 10, 15, 20, 35, 45, 69, 71, 89};
    int n = sizeof(datos) / sizeof(int);
```

```
    int* inicio = datos;
    int* fin = datos + (n - 1);

    // ------------------
    // Búsqueda recursiva
    // ------------------
    int posRec = busquedaBinariaRec(inicio, fin, clave);

    // -------------------
    // Búsqueda iterativa
    // -------------------
    int posIter = busquedaBinariaIter(inicio, fin, clave);

    cout << "\nClave buscada: " << clave << "\n";

    if (posRec != -1)
       cout << "Recursiva: encontrado en la posicion " << posRec << "\n";
    else
       cout << "Recursiva: no encontrado.\n";

    if (posIter != -1)
       cout << "Iterativa: encontrado en la posicion " << posIter << "\n";
    else
       cout << "Iterativa: no encontrado.\n";

    return 0;
}
```

7. Diseñe una variación de Búsqueda Binaria (algoritmo 1.4) que efectúe sólo una comparación binaria (es decir, la comparación devuelve un resultado booleano) de K con un elemento del arreglo cada vez que se invoca la función. Pueden hacerse comparaciones adicionales con variables de intervalo. Analice la corrección de su procedimiento. Sugerencia: ¿Cuándo deberá ser de igualdad (==) la única comparación que se hace?

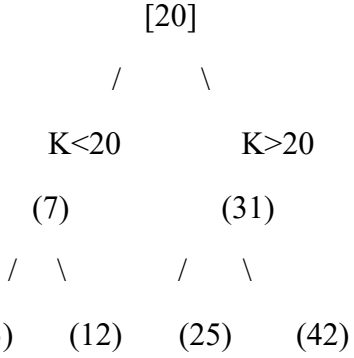
```
#include <iostream>
    using namespace std;


    int busquedaEspecial(int* E, int primero, int ultimo, int K) {

       if (ultimo < primero)
          return -1;

       int* pPrimero = E + primero;
       int* pUltimo  = E + ultimo;
```

```
    if (K < *pPrimero || K > *pUltimo)
        return -1;

    int medio = (primero + ultimo) / 2;
    int* pMedio = E + medio;

    if (K == *pMedio)
        return medio;

    if (K <= *pMedio)
        return busquedaEspecial(E, primero, medio - 1, K);
    else
        return busquedaEspecial(E, medio + 1, ultimo, K);
}

int main() {

    int E[] = {1, 7, 9, 14, 19, 24, 31, 42};
    int n = sizeof(E) / sizeof(E[0]);

    int K;
    cout << "Ingrese el valor a buscar: ";
    cin >> K;

    int indice = busquedaEspecial(E, 0, n - 1, K);

    if (indice == -1)
        cout << "No encontrado\n";
    else
        cout << "Encontrado en posicion: " << indice << "\n";

    return 0;
}
```

8. ¿Cómo podría modificar Búsqueda Binaria (¿algoritmo 1.4) para eliminar trabajo innecesario si se tiene la certeza de que K está en el arreglo? Dibuje un árbol de decisión para el algoritmo modificado con n = 7. Efectúe análisis de comportamiento promedio y de peor caso. (Para el promedio, puede suponerse que n = 2 - 1 para alguna k.)

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

int busquedaAsumida(int* A, int n, int K) {
    int low = 0;
    int high = n - 1;

    while (true) {
        int mid = (low + high) / 2;
```

```
        int valor = *(A + mid);

        if (valor == K)
            return mid;

        if (valor < K)
            low = mid + 1;
        else
            high = mid - 1;
    }
}

int main() {
    int datos[] = {3, 7, 12, 20, 25, 31, 42};
    int n = sizeof(datos) / sizeof(datos[0]);

    int K;
    cout << "Ingrese un valor que SI esté en el arreglo: ";
    cin >> K;

    int pos = busquedaAsumida(datos, n, K);

    cout << "Encontrado en la posicion: " << pos << endl;
    cout << "Valor verificado: " << *(datos + pos) << endl;

    return 0;
}
```



9. Encontrar el algoritmo de búsqueda binaria para encontrar un elemento K en una lista de elementos X_1, X_2, ..., X_n, previamente clasificados en orden ascendente.
   El array o vector X se supone ordenado en orden creciente si los datos son numéricos, o alfabéticamente si son caracteres. Las variables BAJO, CENTRAL, ALTO indican los límites inferior, central y superior del intervalo de búsqueda.
   algoritmo busqueda_binaria
   //declaraciones
   inicio

```
 //1lenar (X, N)
//ordenar (X, N)
leer(K)
//inicializar variables
BAJO← 1
ALTO← N
CENTRAL ← ent ((BAJO + ALTO) / 2)
 mientras (BAJO < ALTO) y (X[CENTRAL] <> K) hacer
si K<X[CENTRAL] entonces
ALTO ←CENTRAL-1
si_no
BAJO ← CENTRAL+1
fin_si
CENTRAL←ent ((BAJO + ALTO) / 2)
fin_mientras
si K = X[CENTRAL] entonces
escribir('Valor encontrado en', CENTRAL)
si_no
escribir('Valor no encontrado')
fin_si
fin
```

Pseudocódigo corregido

```
algoritmo BUSQUEDA_BINARIA
   // X es un arreglo ordenado de tamaño N
   leer(K)

   BAJO ← 1
   ALTO ← N
   CENTRAL ← truncar( (BAJO + ALTO) / 2 )

   mientras BAJO ≤ ALTO y X[CENTRAL] ≠ K hacer
      si K < X[CENTRAL] entonces
         ALTO ← CENTRAL - 1
      si_no
         BAJO ← CENTRAL + 1
      fin_si

      CENTRAL ← truncar( (BAJO + ALTO) / 2 )
   fin_mientras

   si X[CENTRAL] = K entonces
```

```
        escribir("Valor encontrado en ", CENTRAL)
    si_no
        escribir("Valor no encontrado")
    fin_si
fin_algoritmo
```

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

int busquedaBinaria(int* X, int N, int K) {
    int BAJO = 0;
    int ALTO = N - 1;
    int CENTRAL = (BAJO + ALTO) / 2;

    // mientras BAJO <= ALTO y *(X + CENTRAL) != K
    while (BAJO <= ALTO && *(X + CENTRAL) != K) {

        if (K < *(X + CENTRAL)) {
            ALTO = CENTRAL - 1;
        } else {
            BAJO = CENTRAL + 1;
        }

        CENTRAL = (BAJO + ALTO) / 2;
    }

    if (BAJO <= ALTO && *(X + CENTRAL) == K)
        return CENTRAL;
    else
        return -1;
}

int main() {
    int X[] = {3, 8, 12, 17, 25, 30, 45};
    int N = sizeof(X) / sizeof(X[0]);

    int K;
    cout << "Ingrese el valor a buscar: ";
    cin >> K;

    int pos = busquedaBinaria(X, N, K);

    if (pos != -1)
        cout << "Valor encontrado en la posicion " << pos << endl;
    else
        cout << "Valor no encontrado" << endl;

    return 0;
}
```

10. Se cuenta con una lista de Pokémons, de cada uno de estos se sabe su nombre, número y tipo (solo considerar uno el principal), con los cuales deberá resolver las siguientes tareas: determinar si existe el Pokémon Cobalion y mostrar toda su información.

```
#include <iostream>
#include <cstring>
using namespace std;

class Pokemon {
private:
   char nombre[50];
   int numero;
   char tipo[30];

public:
   Pokemon() {
      nombre[0] = '\0';
      tipo[0] = '\0';
      numero = 0;
   }

   void ingresar() {
      cout << "Nombre: ";
      cin >> nombre;
      cout << "Numero: ";
      cin >> numero;
      cout << "Tipo: ";
      cin >> tipo;
   }

   const char* getNombre() { return nombre; }
   int getNumero() { return numero; }
   const char* getTipo() { return tipo; }

   void mostrar() {
      cout << "---------------------------------\n";
      cout << "Nombre : " << nombre << "\n";
      cout << "Numero : " << numero << "\n";
      cout << "Tipo   : " << tipo << "\n";
      cout << "---------------------------------\n";
   }
};

class ListaPokemons {
private:
   Pokemon lista[100];
```

```
    int tam;

    // Ordenar alfabéticamente por nombre (burbuja simple)
    void ordenar() {
        for (int i = 0; i < tam - 1; i++) {
            for (int j = i + 1; j < tam; j++) {
                if (strcmp(lista[i].getNombre(), lista[j].getNombre()) > 0) {
                    Pokemon temp = lista[i];
                    lista[i] = lista[j];
                    lista[j] = temp;
                }
            }
        }
    }

public:
    ListaPokemons() { tam = 0; }

    void ingresarPokemons(int n) {
        for (int i = 0; i < n; i++) {
            cout << "\nIngresando Pokemon #" << (i + 1) << "\n";
            lista[i].ingresar();
        }
        tam = n;
        ordenar(); // Ordenar después de ingresar
    }

    // Búsqueda binaria por nombre
    int buscarCobalion() {
        int low = 0;
        int high = tam - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int cmp = strcmp(lista[mid].getNombre(), "Cobalion");
            if (cmp == 0)
                return mid; // encontrado
            else if (cmp < 0)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return -1; // no encontrado
    }

    void mostrarCobalion() {
        int pos = buscarCobalion();
        if (pos != -1) {
            cout << "\n*** Cobalion ENCONTRADO ***\n";
            lista[pos].mostrar();
        } else {
```

```
            cout << "\nCobalion NO se encuentra en la lista.\n";
        }
    }
};

int main() {
    ListaPokemons pokedex;

    int n;
    cout << "Cuantos Pokemons desea ingresar? ";
    cin >> n;

    pokedex.ingresarPokemons(n);
    pokedex.mostrarCobalion();

    return 0;
}
```

11. Se dispone de la lista de superhéroes y villanos de la saga de Marvel Cinematic Universe (MCU) de los que contamos con la información de nombre del personaje y año de la primera película en la que apareció; a partir de estos resolver las siguientes actividades: indicar quien fue el primer y el último personaje en aparecer en una película sin realizar un recorrido de la lista (podrían ser más de uno tanto el primero como el último).

```
#include <iostream>
#include <cstring>
using namespace std;

class Personaje {
private:
    char nombre[50];
    int anio;

public:
    Personaje() {
        nombre[0] = '\0';
        anio = 0;
    }

    void ingresar() {
        cout << "Nombre del personaje: ";
        cin >> nombre;
        cout << "Año de primera aparición: ";
        cin >> anio;
    }

    int getAnio() { return anio; }
    const char* getNombre() { return nombre; }
};
```

```
class MCU {
private:
    Personaje lista[100];
    char primeros[100][50];
    char ultimos[100][50];
    int n, cantPrimeros, cantUltimos;
    int anioMin, anioMax;

public:
    MCU() {
        n = 0;
        cantPrimeros = 0;
        cantUltimos = 0;
        anioMin = 9999;
        anioMax = -1;
    }

    void ingresarPersonajes(int cantidad) {
        n = cantidad;

        for (int i = 0; i < n; i++) {
            cout << "\nPersonaje #" << (i + 1) << "\n";
            lista[i].ingresar();

            int anio = lista[i].getAnio();
            const char* nombre = lista[i].getNombre();

            // Actualizar primeros
            if (anio < anioMin) {
                anioMin = anio;
                cantPrimeros = 0;
                strcpy(primeros[cantPrimeros++], nombre);
            } else if (anio == anioMin) {
                strcpy(primeros[cantPrimeros++], nombre);
            }

            // Actualizar últimos
            if (anio > anioMax) {
                anioMax = anio;
                cantUltimos = 0;
                strcpy(ultimos[cantUltimos++], nombre);
            } else if (anio == anioMax) {
                strcpy(ultimos[cantUltimos++], nombre);
            }
        }
    }

    void mostrarResultados() {
        cout << "\n===== Primeros personajes del MCU =====\n";
```

```
        for (int i = 0; i < cantPrimeros; i++)
            cout << primeros[i] << " (Año " << anioMin << ")\n";

        cout << "\n===== Últimos personajes del MCU =====\n";
        for (int i = 0; i < cantUltimos; i++)
            cout << ultimos[i] << " (Año " << anioMax << ")\n";
    }
};

int main() {
    MCU universo;
    int n;

    cout << "¿Cuántos personajes desea ingresar? ";
    cin >> n;

    universo.ingresarPersonajes(n);
    universo.mostrarResultados();

    return 0;
}
```

12. Se dispone de una lista de películas con la siguiente información: título, año de estreno, recaudación y valoración del público (de 1 a 5), los cuales debemos procesar contemplando las siguiente tareas: determinar si la película “Avengers: Infinity War” está en la lista y mostrar toda su información; indicar en qué posición se encuentra la película “Star Wars: The Return of Jedi”.

```
#include <iostream>
#include <string>
using namespace std;

struct Pelicula {
    string titulo;
    int anio;
    double recaudacion;
    double valoracion;
};

// Función de búsqueda binaria por título
int busquedaBinaria(Pelicula* lista, int n, string objetivo) {
    int low = 0;
    int high = n - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (lista[mid].titulo == objetivo)
            return mid; // encontrado
```

```
        else if (lista[mid].titulo < objetivo)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // no encontrado
}

int main() {
    // Lista ordenada alfabéticamente por título
    Pelicula lista[] = {
        {"Avengers: Infinity War", 2018, 2048.4, 4.8},
        {"Iron Man", 2008, 585.2, 4.5},
        {"Star Wars: The Return of Jedi", 1983, 475.1, 4.7}
    };
    int n = sizeof(lista)/sizeof(lista[0]);

    // Para búsqueda binaria, debemos **ordenar la lista alfabéticamente**
    // (aquí ya está ordenada manualmente para el ejemplo)

    // 1️⃣ Buscar "Avengers: Infinity War"
    int posAvengers = busquedaBinaria(lista, n, "Avengers: Infinity War");
    if (posAvengers != -1) {
        cout << "Avengers: Infinity War encontrada:\n";
        cout << "Título: " << lista[posAvengers].titulo << "\n";
        cout << "Año: " << lista[posAvengers].anio << "\n";
        cout << "Recaudación: " << lista[posAvengers].recaudacion << "\n";
        cout << "Valoración: " << lista[posAvengers].valoracion << "\n";
    } else {
        cout << "Avengers: Infinity War no encontrada.\n";
    }

    // 2️⃣ Buscar "Star Wars: The Return of Jedi"
    int posStarWars = busquedaBinaria(lista, n, "Star Wars: The Return of Jedi");
    if (posStarWars != -1) {
        cout << "\nStar Wars: The Return of Jedi está en la posición: " << posStarWars + 1 << "\n";
    } else {
        cout << "Star Wars: The Return of Jedi no encontrada.\n";
    }

    return 0;
}
```

13. Un vector T tiene cien posiciones, 0.100. Supongamos que las claves de búsqueda de los elementos de la tabla son enteros positivos (por ejemplo, número del DNI).

Una función de conversión h debe tomar un número arbitrario entero positivo x y convertirlo en un entero en el rango 0.100, esto es, h es una función tal que para un entero positivo x.

h(x)= n, donde n es entero en el rango 0.100
El método del módulo, tomando 101, será
h(x) = x mod 101
Si se tiene el DNI número 234661234, por ejemplo, se tendrá la posición 56:
234661234 mod 101 = 56

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

int main() {
   long long dni;
   long long posicion;

   cout << "Ingrese un DNI (entero positivo): ";
   cin >> dni;

   if (dni < 0) {
      cout << "El DNI debe ser un entero positivo." << endl;
      return 0;
   }

   posicion = dni % 101;

   cout << "La posicion hash para el DNI es: " << posicion << endl;

   return 0;
}
```

14. Un entero de ocho dígitos se puede dividir en grupos de tres, tres y dos dígitos, los grupos se suman juntos y se truncan si es necesario para que estén en el rango adecuado de índices.

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

int main() {
   long long dni;
   cout << "Ingrese un DNI de 8 digitos: ";
   cin >> dni;

   long long* p = &dni;

   long long temp = *p;
   long long* q = &temp;
```

```
        int g3 = (int)(*q % 100);
        *q /= 100;

        int g2 = (int)(*q % 1000);
        *q /= 1000;

        int g1 = (int)(*q);

        int suma = g1 + g2 + g3;

        int direccion = suma % 1000;

        cout << "Direccion: " << direccion << endl;

        return 0;
    }
```

15. Desarrollar un algoritmo que permita implementar una tabla hash que represente un diccionario que permita resolver las siguientes actividades:
    a. Agregar una palabra y su significado al diccionario;
    b. Determinar si una palabra existe y mostrar su significado;
    c. Borrar una palabra del diccionario;
    d. La tabla debe tener 28 posiciones y manejar las colisiones con lista enlazadas;
    e. Mejorar el rendimiento de la tabla utilizando árboles binarios de búsqueda.

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

#define TAM 28

/* ========================
   NODO LISTA ENLAZADA
   ======================== */
class NodoLista {
public:
    std::string palabra;
    std::string significado;
    NodoLista *siguiente;

    NodoLista(std::string p, std::string s) {
        palabra = p;
        significado = s;
        siguiente = nullptr;
    }
};

/* ========================
   NODO ARBOL BINARIO
```

```
   ========================= */
class NodoABB {
public:
    std::string palabra;
    std::string significado;
    NodoABB *izq;
    NodoABB *der;

    NodoABB(std::string p, std::string s) {
        palabra = p;
        significado = s;
        izq = nullptr;
        der = nullptr;
    }
};

/* ========================
   BUCKET DE LA TABLA
   ========================= */
class Bucket {
public:
    NodoLista *lista;
    NodoABB *arbol;
    bool usarArbol;

    Bucket() {
        lista = nullptr;
        arbol = nullptr;
        usarArbol = false;
    }
};

/* ========================
   TABLA HASH
   ========================= */
class HashTable {
private:
    Bucket *tabla;

    int funcionHash(std::string palabra) {
        int suma = 0;
        for (char c : palabra)
            suma += c;
        return suma % TAM;
    }

    NodoABB* insertarABB(NodoABB *raiz, std::string p, std::string s) {
        if (raiz == nullptr)
            return new NodoABB(p, s);
```

```
        if (p < raiz->palabra)
            raiz->izq = insertarABB(raiz->izq, p, s);
        else if (p > raiz->palabra)
            raiz->der = insertarABB(raiz->der, p, s);

        return raiz;
    }

    void buscarABB(NodoABB *raiz, std::string p) {
        if (raiz == nullptr) {
            std::cout << "Palabra no encontrada\n";
            return;
        }

        if (p == raiz->palabra) {
            std::cout << "Significado: " << raiz->significado << std::endl;
            return;
        }

        if (p < raiz->palabra)
            buscarABB(raiz->izq, p);
        else
            buscarABB(raiz->der, p);
    }

public:
    HashTable() {
        tabla = new Bucket[TAM];
    }

    /* a) Insertar palabra */
    void insertar(std::string palabra, std::string significado) {
        int i = funcionHash(palabra);
        Bucket *b = tabla + i;

        if (b->usarArbol) {
            b->arbol = insertarABB(b->arbol, palabra, significado);
            return;
        }

        NodoLista *nuevo = new NodoLista(palabra, significado);
        nuevo->siguiente = b->lista;
        b->lista = nuevo;

        int contador = 0;
        NodoLista *aux = b->lista;
        while (aux != nullptr) {
            contador++;
            aux = aux->siguiente;
        }
```

```
    if (contador > 3) {
        aux = b->lista;
        while (aux != nullptr) {
            b->arbol = insertarABB(b->arbol, aux->palabra, aux->significado);
            aux = aux->siguiente;
        }
        b->usarArbol = true;
    }
}

/* b) Buscar palabra */
void buscar(std::string palabra) {
    int i = funcionHash(palabra);
    Bucket *b = tabla + i;

    if (b->usarArbol) {
        buscarABB(b->arbol, palabra);
        return;
    }

    NodoLista *aux = b->lista;
    while (aux != nullptr) {
        if (aux->palabra == palabra) {
            std::cout << "Significado: " << aux->significado << std::endl;
            return;
        }
        aux = aux->siguiente;
    }
    std::cout << "Palabra no encontrada\n";
}

/* c) Eliminar palabra (lista enlazada) */
void eliminar(std::string palabra) {
    int i = funcionHash(palabra);
    Bucket *b = tabla + i;

    NodoLista *aux = b->lista;
    NodoLista *ant = nullptr;

    while (aux != nullptr) {
        if (aux->palabra == palabra) {
            if (ant == nullptr)
                b->lista = aux->siguiente;
            else
                ant->siguiente = aux->siguiente;

            delete aux;
            std::cout << "Palabra eliminada\n";
            return;
```

```
            }
            ant = aux;
            aux = aux->siguiente;
        }
        std::cout << "Palabra no encontrada\n";
    }
};

/* ========================
   PROGRAMA PRINCIPAL
   ======================== */
int main() {
    HashTable diccionario;

    diccionario.insertar("casa", "Lugar para vivir");
    diccionario.insertar("perro", "Animal domestico");
    diccionario.insertar("sol", "Estrella");
    diccionario.insertar("luna", "Satelite natural");

    diccionario.buscar("perro");
    diccionario.eliminar("casa");
    diccionario.buscar("casa");

    return 0;
}
```

16. Dado un árbol con nombre de los superhéroes y villanos de la saga Marvel Cinematic Universe (MCU), desarrollar un algoritmo que contemple lo siguiente:
    a. Además del nombre del superhéroe en cada nodo del árbol se almacena un campo booleano que indica si es un héroe o un villano, True y False respectivamente;
    b. Listar los villanos ordenados alfabéticamente;
    c. Mostrar todos los superhéroes que empiezan con C;
    d. Determinar cuántos superhéroes hay en el árbol.

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

/* ========================
   NODO DEL ÁRBOL
   ======================== */
class Nodo {
public:
    std::string nombre;
    bool esHeroe;   // true = héroe, false = villano
    Nodo *izq;
    Nodo *der;

    Nodo(std::string n, bool h) {
        nombre = n;
```

```
        esHeroe = h;
        izq = nullptr;
        der = nullptr;
    }
};

/* ========================
   ÁRBOL BINARIO DE BÚSQUEDA
   ======================== */
class ArbolMCU {
private:
    Nodo *raiz;

    /* Inserción ABB */
    Nodo* insertar(Nodo *r, std::string nombre, bool esHeroe) {
        if (r == nullptr)
            return new Nodo(nombre, esHeroe);

        if (nombre < r->nombre)
            r->izq = insertar(r->izq, nombre, esHeroe);
        else if (nombre > r->nombre)
            r->der = insertar(r->der, nombre, esHeroe);

        return r;
    }

    /* b) Villanos en orden alfabético */
    void listarVillanos(Nodo *r) {
        if (r == nullptr) return;

        listarVillanos(r->izq);

        if (!r->esHeroe)
            std::cout << r->nombre << std::endl;

        listarVillanos(r->der);
    }

    /* c) Héroes que empiezan con C */
    void heroesConC(Nodo *r) {
        if (r == nullptr) return;

        heroesConC(r->izq);

        if (r->esHeroe && r->nombre[0] == 'C')
            std::cout << r->nombre << std::endl;

        heroesConC(r->der);
    }
```

```
    /* d) Contar héroes */
    int contarHeroes(Nodo *r) {
        if (r == nullptr) return 0;

        int contador = contarHeroes(r->izq) + contarHeroes(r->der);

        if (r->esHeroe)
            contador++;

        return contador;
    }

public:
    ArbolMCU() {
        raiz = nullptr;
    }

    /* Inserción pública */
    void insertarPersonaje(std::string nombre, bool esHeroe) {
        raiz = insertar(raiz, nombre, esHeroe);
    }

    /* b) Mostrar villanos */
    void mostrarVillanos() {
        std::cout << "\nVillanos (orden alfabético):\n";
        listarVillanos(raiz);
    }

    /* c) Mostrar héroes que empiezan con C */
    void mostrarHeroesConC() {
        std::cout << "\nHeroes que empiezan con C:\n";
        heroesConC(raiz);
    }

    /* d) Total de héroes */
    void totalHeroes() {
        std::cout << "\nCantidad de heroes: "
                  << contarHeroes(raiz) << std::endl;
    }
};

/* ========================
   PROGRAMA PRINCIPAL
   ======================== */
int main() {
    ArbolMCU arbol;

    /* Inserción de personajes MCU */
    arbol.insertarPersonaje("Iron Man", true);
    arbol.insertarPersonaje("Captain America", true);
```

```
    arbol.insertarPersonaje("Thor", true);
    arbol.insertarPersonaje("Hulk", true);
    arbol.insertarPersonaje("Doctor Strange", true);

    arbol.insertarPersonaje("Thanos", false);
    arbol.insertarPersonaje("Loki", false);
    arbol.insertarPersonaje("Ultron", false);
    arbol.insertarPersonaje("Red Skull", false);

    /* Resultados */
    arbol.mostrarVillanos();
    arbol.mostrarHeroesConC();
    arbol.totalHeroes();

    return 0;
}
```

17. De un ABB realizar funciones que permita:
    a. Contar el total de nodos
    b. Contar todas las hojas
    c. Imprimir hojas
    d. Obtener el padre un nodo
    e. Obtener Altura

```
// Método para buscar el nodo con el valor mayor (extremo derecho)
template<typename T>
Nodo<T>* ABB<T>::buscarMayor() const {
    if (Raiz == nullptr) return nullptr;

    Nodo<T>* actual = Raiz;
    while (actual->getDerecha() != nullptr) {
        actual = actual->getDerecha();
    }
    return actual;
}

// Método para buscar el nodo con el valor menor (extremo izquierdo)
template<typename T>
Nodo<T>* ABB<T>::buscarMenor() const {
    if (Raiz == nullptr) return nullptr;

    Nodo<T>* actual = Raiz;
    while (actual->getIzquierda() != nullptr) {
        actual = actual->getIzquierda();
    }
    return actual;
}

template<typename T>
int ABB<T>::contarNodos(){
```

```
    return contarNodosAux(Raiz);
}

template<typename T>
int ABB<T>::contarNodosAux(Nodo<T>* nodo){
    if(nodo == nullptr) return 0;
    return 1 + contarNodosAux(nodo->getIzquierda()) + contarNodosAux(nodo->getDerecha());
}

template<typename T>
int ABB<T>::contarHojas(){
    return contarHojasAux(Raiz);
}

template<typename T>
int ABB<T>::contarHojasAux(Nodo<T>* nodo){
    if(nodo == nullptr) return 0;
    if(nodo->getIzquierda() == nullptr && nodo->getDerecha() == nullptr){
        return 1;
    }
    return contarHojasAux(nodo->getIzquierda()) + contarHojasAux(nodo->getDerecha());
}

template<typename T>
void ABB<T>::imprimirHojas(){
    imprimirHojasAux(Raiz);
    std::cout << std::endl;
}

template<typename T>
void ABB<T>::imprimirHojasAux(Nodo<T>* nodo){
    if(nodo == nullptr) return;

    // Si es hoja, imprimir
    if(nodo->getIzquierda() == nullptr && nodo->getDerecha() == nullptr){
        std::cout << nodo->getDato() << " ";
        return;
    }

    imprimirHojasAux(nodo->getIzquierda());
    imprimirHojasAux(nodo->getDerecha());
}

template<typename T>
Nodo<T>* ABB<T>::obtenerPadre(Nodo<T>* hijo){
    // Si el hijo es nulo, o es la raíz (no tiene padre), retornamos nullptr
    if(hijo == nullptr || Raiz == nullptr || hijo == Raiz) return nullptr;

    Nodo<T>* actual = Raiz;
    T valorHijo = hijo->getDato();
```

```
    while(actual != nullptr){
        if(valorHijo < actual->getDato()){
            if(actual->getIzquierda() == hijo) return actual;
            actual = actual->getIzquierda();
        }
        else if(valorHijo > actual->getDato()){
            if(actual->getDerecha() == hijo) return actual;
            actual = actual->getDerecha();
        }
        else {
            // Encontró el valor pero no es hijo directo (caso raro si el puntero es válido)
            return nullptr;
        }
    }
    return nullptr;
}

template<typename T>
int ABB<T>::obtenerAltura(){
    return obtenerAlturaAux(Raiz);
}

template<typename T>
int ABB<T>::obtenerAlturaAux(Nodo<T>* nodo){
    if(nodo == nullptr) return 0;

    int alturaIzq = obtenerAlturaAux(nodo->getIzquierda());
    int alturaDer = obtenerAlturaAux(nodo->getDerecha());

    return 1 + (alturaIzq > alturaDer ? alturaIzq : alturaDer);
}
```

18. Implementar un árbol AVL:

```
#include "AVL.hpp"
#include <iostream>

template<typename T>
AVL<T>::AVL() {
    raiz = nullptr;
}

template<typename T>
int AVL<T>::obtenerAltura(NodoAVL<T>* nodo) {
    if (nodo == nullptr) return 0;
    return nodo->getAltura();
}

template<typename T>
int AVL<T>::maximo(int a, int b) {
```

```
    return (a > b) ? a : b;
}

template<typename T>
int AVL<T>::obtenerBalance(NodoAVL<T>* nodo) {
    if (nodo == nullptr) return 0;
    return obtenerAltura(nodo->getIzquierda()) - obtenerAltura(nodo->getDerecha());
}

// Rotación a la derecha (Caso Izquierda-Izquierda)
template<typename T>
NodoAVL<T>* AVL<T>::rotacionDerecha(NodoAVL<T>* y) {
    NodoAVL<T>* x = y->getIzquierda();
    NodoAVL<T>* T2 = x->getDerecha();

    // Realizar rotación
    x->setDerecha(y);
    y->setIzquierda(T2);

    // Actualizar alturas
    y->setAltura(maximo(obtenerAltura(y->getIzquierda()), obtenerAltura(y->getDerecha())) + 1);
    x->setAltura(maximo(obtenerAltura(x->getIzquierda()), obtenerAltura(x->getDerecha())) + 1);

    // Retornar nueva raíz
    return x;
}

// Rotación a la izquierda (Caso Derecha-Derecha)
template<typename T>
NodoAVL<T>* AVL<T>::rotacionIzquierda(NodoAVL<T>* x) {
    NodoAVL<T>* y = x->getDerecha();
    NodoAVL<T>* T2 = y->getIzquierda();

    // Realizar rotación
    y->setIzquierda(x);
    x->setDerecha(T2);

    // Actualizar alturas
    x->setAltura(maximo(obtenerAltura(x->getIzquierda()), obtenerAltura(x->getDerecha())) + 1);
    y->setAltura(maximo(obtenerAltura(y->getIzquierda()), obtenerAltura(y->getDerecha())) + 1);

    // Retornar nueva raíz
    return y;
}

template<typename T>
void AVL<T>::insertar(T dato) {
    raiz = insertarAux(raiz, dato);
}
```

```
template<typename T>
NodoAVL<T>* AVL<T>::insertarAux(NodoAVL<T>* nodo, T dato) {
    // 1. Inserción normal de BST
    if (nodo == nullptr)
        return new NodoAVL<T>(dato);

    if (dato < nodo->getDato())
        nodo->setIzquierda(insertarAux(nodo->getIzquierda(), dato));
    else if (dato > nodo->getDato())
        nodo->setDerecha(insertarAux(nodo->getDerecha(), dato));
    else // Claves duplicadas no permitidas
        return nodo;

    // 2. Actualizar altura de este nodo ancestro
    nodo->setAltura(1 + maximo(obtenerAltura(nodo->getIzquierda()), obtenerAltura(nodo->getDerecha())));

    // 3. Obtener el factor de balance para verificar si se desbalanceó
    int balance = obtenerBalance(nodo);

    // 4. Casos de rotación

    // Caso Izquierda Izquierda
    if (balance > 1 && dato < nodo->getIzquierda()->getDato())
        return rotacionDerecha(nodo);

    // Caso Derecha Derecha
    if (balance < -1 && dato > nodo->getDerecha()->getDato())
        return rotacionIzquierda(nodo);

    // Caso Izquierda Derecha
    if (balance > 1 && dato > nodo->getIzquierda()->getDato()) {
        nodo->setIzquierda(rotacionIzquierda(nodo->getIzquierda()));
        return rotacionDerecha(nodo);
    }

    // Caso Derecha Izquierda
    if (balance < -1 && dato < nodo->getDerecha()->getDato()) {
        nodo->setDerecha(rotacionDerecha(nodo->getDerecha()));
        return rotacionIzquierda(nodo);
    }

    return nodo;
}
template<typename T>
void AVL<T>::inorden() {
    inordenAux(raiz);
    std::cout << std::endl;
}
template<typename T>
```

```
void AVL<T>::inordenAux(NodoAVL<T>* nodo) {
    if (nodo != nullptr) {
        inordenAux(nodo->getIzquierda());
        std::cout << nodo->getDato() << " ";
        inordenAux(nodo->getDerecha());
    }
}
```

19. Diseñar e implementar un árbol rojo y negro con sus operaciones básicas:

    a. Insertar
    b. Buscar
    c. Imprimir
    d. Eliminar por un elemento a escoger.

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

enum Color { ROJO, NEGRO };

/* ============================
   NODO ÁRBOL ROJO-NEGRO
   ============================ */
class Nodo {
public:
    int dato;
    Color color;
    Nodo* izquierdo;
    Nodo* derecho;
    Nodo* padre;

    Nodo(int valor) {
        dato = valor;
        color = ROJO;
        izquierdo = derecho = padre = nullptr;
    }
};

/* ============================
   ÁRBOL ROJO-NEGRO
   ============================ */
class ArbolRojoNegro {
private:
    Nodo* raiz;

    /* ROTACIÓN IZQUIERDA */
    void rotarIzquierda(Nodo* x) {
        Nodo* y = x->derecho;
        x->derecho = y->izquierdo;

        if (y->izquierdo != nullptr)
```

```
        y->izquierdo->padre = x;

    y->padre = x->padre;

    if (x->padre == nullptr)
        raiz = y;
    else if (x == x->padre->izquierdo)
        x->padre->izquierdo = y;
    else
        x->padre->derecho = y;

    y->izquierdo = x;
    x->padre = y;
}

/* ROTACIÓN DERECHA */
void rotarDerecha(Nodo* y) {
    Nodo* x = y->izquierdo;
    y->izquierdo = x->derecho;

    if (x->derecho != nullptr)
        x->derecho->padre = y;

    x->padre = y->padre;

    if (y->padre == nullptr)
        raiz = x;
    else if (y == y->padre->izquierdo)
        y->padre->izquierdo = x;
    else
        y->padre->derecho = x;

    x->derecho = y;
    y->padre = x;
}

/* CORREGIR INSERCIÓN */
void corregirInsercion(Nodo* z) {
    while (z != raiz && z->padre->color == ROJO) {
        Nodo* abuelo = z->padre->padre;

        if (z->padre == abuelo->izquierdo) {
            Nodo* tio = abuelo->derecho;

            if (tio != nullptr && tio->color == ROJO) {
                z->padre->color = NEGRO;
                tio->color = NEGRO;
                abuelo->color = ROJO;
                z = abuelo;
            } else {
```

```
            if (z == z->padre->derecho) {
                z = z->padre;
                rotarIzquierda(z);
            }
            z->padre->color = NEGRO;
            abuelo->color = ROJO;
            rotarDerecha(abuelo);
        }
    } else {
        Nodo* tio = abuelo->izquierdo;

        if (tio != nullptr && tio->color == ROJO) {
            z->padre->color = NEGRO;
            tio->color = NEGRO;
            abuelo->color = ROJO;
            z = abuelo;
        } else {
            if (z == z->padre->izquierdo) {
                z = z->padre;
                rotarDerecha(z);
            }
            z->padre->color = NEGRO;
            abuelo->color = ROJO;
            rotarIzquierda(abuelo);
        }
    }
  }
  raiz->color = NEGRO;
}

/* BUSCAR NODO */
Nodo* buscarNodo(Nodo* actual, int valor) {
  if (actual == nullptr || actual->dato == valor)
    return actual;

  if (valor < actual->dato)
    return buscarNodo(actual->izquierdo, valor);
  else
    return buscarNodo(actual->derecho, valor);
}

/* MÍNIMO */
Nodo* minimo(Nodo* nodo) {
  while (nodo->izquierdo != nullptr)
    nodo = nodo->izquierdo;
  return nodo;
}

/* TRANSPLANTAR */
void transplantar(Nodo* u, Nodo* v) {
```

```
        if (u->padre == nullptr)
            raiz = v;
        else if (u == u->padre->izquierdo)
            u->padre->izquierdo = v;
        else
            u->padre->derecho = v;

        if (v != nullptr)
            v->padre = u->padre;
    }

    /* ELIMINAR NODO */
    void eliminarNodo(Nodo* z) {
        if (z == nullptr) return;

        if (z->izquierdo == nullptr)
            transplantar(z, z->derecho);
        else if (z->derecho == nullptr)
            transplantar(z, z->izquierdo);
        else {
            Nodo* y = minimo(z->derecho);
            z->dato = y->dato;
            eliminarNodo(y);
            return;
        }
        delete z;
    }

    /* IMPRIMIR INORDEN */
    void imprimirInorden(Nodo* nodo) {
        if (nodo != nullptr) {
            imprimirInorden(nodo->izquierdo);
            cout << nodo->dato << "(" << (nodo->color == ROJO ? "R" : "N") << ") ";
            imprimirInorden(nodo->derecho);
        }
    }

public:
    ArbolRojoNegro() {
        raiz = nullptr;
    }

    /* INSERTAR */
    void insertar(int valor) {
        Nodo* nuevo = new Nodo(valor);
        Nodo* padre = nullptr;
        Nodo* actual = raiz;

        while (actual != nullptr) {
            padre = actual;
```

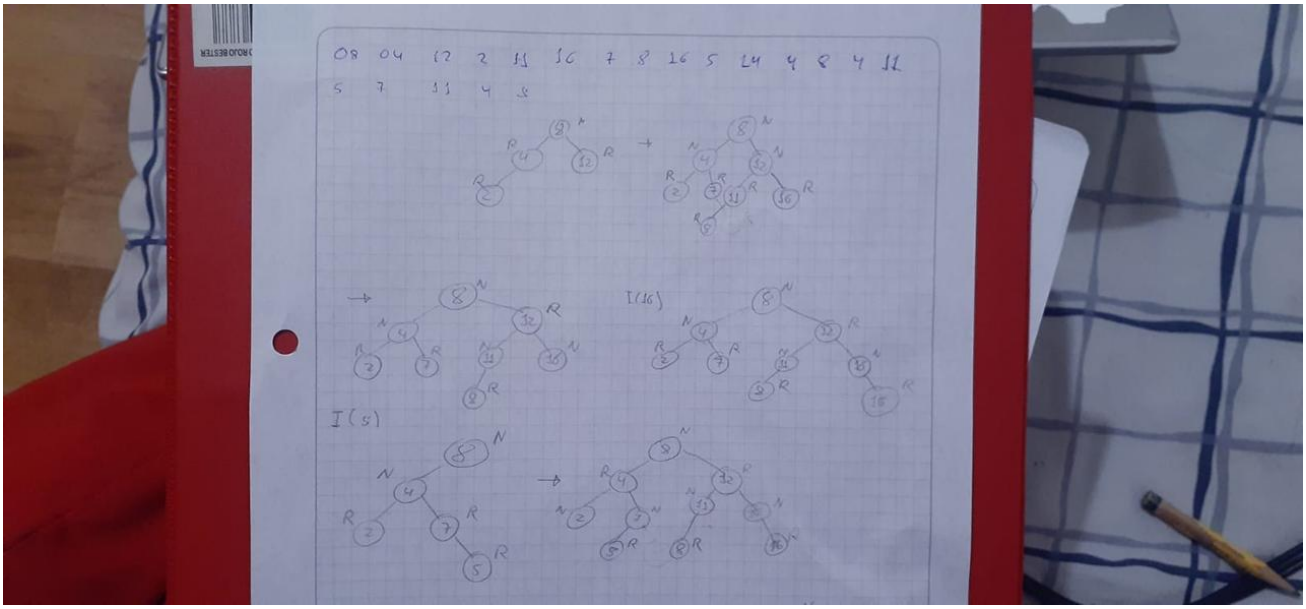
```
            if (valor < actual->dato)
                actual = actual->izquierdo;
            else
                actual = actual->derecho;
        }

        nuevo->padre = padre;

        if (padre == nullptr)
            raiz = nuevo;
        else if (valor < padre->dato)
            padre->izquierdo = nuevo;
        else
            padre->derecho = nuevo;

        corregirInsercion(nuevo);
    }

    /* BUSCAR */
    bool buscar(int valor) {
        return buscarNodo(raiz, valor) != nullptr;
    }

    /* ELIMINAR */
    void eliminar(int valor) {
        Nodo* nodo = buscarNodo(raiz, valor);
        eliminarNodo(nodo);
    }

    /* IMPRIMIR */
    void imprimir() {
        cout << "Inorden: ";
        imprimirInorden(raiz);
        cout << endl;
    }
};

/* ==========================
   MAIN
   ========================== */
int main() {
    ArbolRojoNegro arbol;
    int opcion, valor;

    do {
        cout << "\n--- ARBOL ROJO-NEGRO ---\n";
        cout << "1. Insertar\n";
        cout << "2. Buscar\n";
        cout << "3. Imprimir\n";
        cout << "4. Eliminar\n";
```

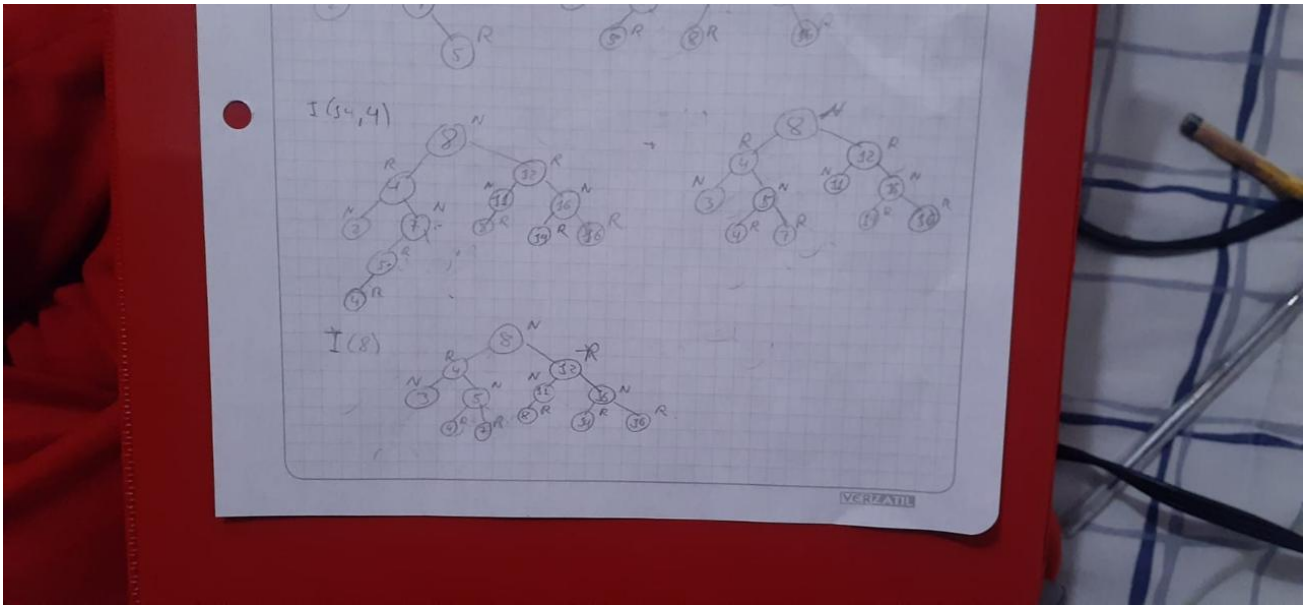
```
        cout << "0. Salir\n";
        cout << "Opcion: ";
        cin >> opcion;

        switch (opcion) {
        case 1:
            cout << "Valor a insertar: ";
            cin >> valor;
            arbol.insertar(valor);
            break;

        case 2:
            cout << "Valor a buscar: ";
            cin >> valor;
            if (arbol.buscar(valor))
                cout << "Elemento encontrado\n";
            else
                cout << "Elemento NO encontrado\n";
            break;

        case 3:
            arbol.imprimir();
            break;

        case 4:
            cout << "Valor a eliminar: ";
            cin >> valor;
            arbol.eliminar(valor);
            break;
        }
    } while (opcion != 0);

    return 0;
}
```

1. Para un **árbol rojo y negro**, realizar el proceso de **inserción** de **15 números**, considerando la siguiente secuencia: 8,4,12,2,11,16,7,8,16, 5,14,4,8,4,11 Posteriormente, efectuar la **eliminación de 3 valores**, tomados del siguiente conjunto: **5,7,11**

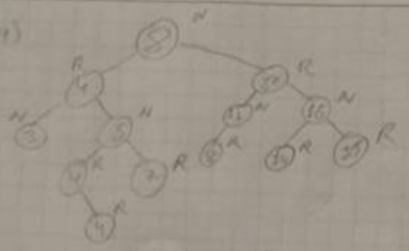
El desarrollo debe contemplar:

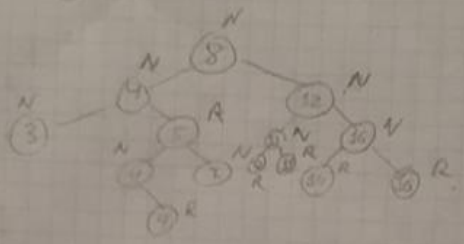
b. La **inserción paso a paso**, mostrando la estructura del árbol B+ después de cada operación, como se realizaría manualmente.
b. La **eliminación paso a paso**, indicando los casos de redistribución o fusión de nodos cuando sea necesario.


08 04 12 2 11 16 7 8 16 5 14 4 8 4 11
5 7 11 4 8
I(16)
I(5)

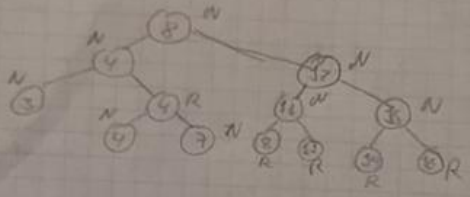

I(14,4)
I(8)
VERSATIL

I(4)

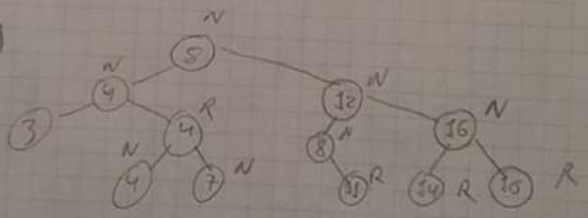


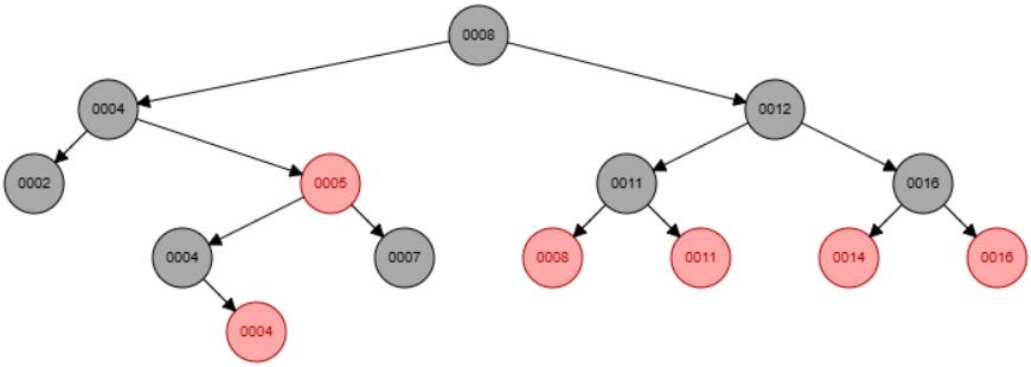


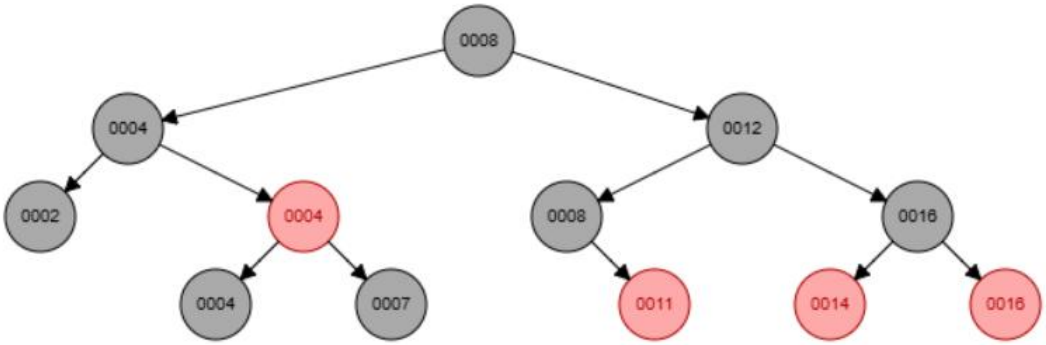
E(5,7)



E(31)




0008
0004
0012
0002
0005
0011
0016
0004
0007
0008
0011
0014
0016
0004


0008
0004
0012
0002
0004
0008
0016
0004
0007
0011
0014
0016