



ETAPA 3 — PROTOTIPO

Curso: Estructura de Datos

Docente: Yesenia Concha Ramos

Integrantes:

- Jhon Gustavo Ccarita Velasquez
- Rodrigo Sevillanos Tinco
- Andre Sebastian Espinoza Zea

Tema: ABR - prototipado

NRC: 59008

2025-02

Cusco - Perú

CAPÍTULO 1 - FASE DE IDEACIÓN.....	4
1. Descripción general del proyecto.....	4
2. Requerimientos del sistema.....	4
Requerimientos funcionales (RF).....	4
Requerimientos no funcionales (RNF).....	4
3. Respuestas a preguntas guía.....	5
1. Información almacenada en cada nodo.....	5
2. Inserción y eliminación sin romper el ABB.....	5
3. Métodos de recorrido.....	5
4. Determinar si pertenece a una rama.....	6
5. Balancear el árbol.....	6
CAPÍTULO 2 – PROTOTIPO.....	7
1. DESCRIPCIÓN DE LA ESTRUCTURA DE DATOS Y OPERACIONES.....	7
1.1 Estructura de cada nodo.....	7
1.2 Operaciones implementadas.....	7
2. ALGORITMOS PRINCIPALES (PSEUDOCÓDIGO).....	9
2.1 Pseudocódigo para crear un Árbol Binario de Búsqueda (Inserción).....	9
2.2 Pseudocódigo para realizar recorridos del árbol.....	10
2.3 Pseudocódigo: Crear un Árbol Binario (Insertar nodo en ABB).....	11
2.4 Pseudocódigo: eliminar un miembro del ABB.....	12
3. DIAGRAMAS DE FLUJO.....	13
DIAGRAMA 1 – Proceso de Inserción en un Árbol Binario de Búsqueda (ABB).....	13
DIAGRAMA 2 — Recorrido INORDEN (Izq – Raíz – Der).....	14
DIAGRAMA 3 – Eliminación en un Árbol Binario de Búsqueda (ABB).....	15
DIAGRAMA 4 – Búsqueda en ABB.....	16
DIAGRAMA 5 – Inserción en ABB (para referencia).....	17
4. AVANCE DEL CÓDIGO FUENTE.....	18
1.1 Definición del nodo.....	18
1.2 Función Insertar.....	18
1.3 Función Buscar.....	19
1.4 Función ELiminar.....	20
1.5 Implementación en ASCII del arbol.....	21
Capítulo 3 – Solución Final.....	22
1. Código limpio, bien comentado y estructurado.....	22
2. Capturas de pantalla de las ventanas de ejecución con las diversas pruebas de validación de datos.....	29
2.1 Validación de entradas insertar.....	29
2.2 Buscar miembros.....	30
2.3 Eliminar miembros.....	31
Capítulo 4 – Evidencias de Trabajo Colaborativo.....	32
1. Repositorio con Control de Versiones.....	32
Historial de ramas y fusiones si es aplicable.....	32

Evidencia por cada integrante del equipo.....	32
Enlace a la herramienta colaborativa.....	34

CAPÍTULO 1 - FASE DE IDEACIÓN

1. Descripción general del proyecto

Este proyecto propone el diseño e implementación de un Árbol Genealógico Mitológico, cuyo propósito es modelar la compleja jerarquía de dioses, titanes, héroes y criaturas de la mitología griega.

Para representar esta estructura se emplea un Árbol Binario de Búsqueda (ABB), tomando como clave principal el año simbólico de nacimiento de cada entidad mitológica.

El ABB permite:

- organizar la genealogía en un orden temporal coherente,
- consultar relaciones de antigüedad y descendencia,
- recorrer la estructura bajo distintos enfoques jerárquicos,
- realizar operaciones de búsqueda en tiempo eficiente,
- mantener el orden interno sin necesidad de estructuras externas.

Cada nodo representa a un ser mitológico y contiene su nombre, su rol jerárquico y su año simbólico, lo que permite reconstruir relaciones como:

- linajes divinos (Titanes → Olímpicos),
- descendencias célebres (Zeus → Atenea → héroes),
- etapas mitológicas (primigenios → olímpicos → héroes).

2. Requerimientos del sistema

Requerimientos funcionales (RF)

- RF1: Insertar dioses, titanes, héroes o criaturas en el árbol.
- RF2: Buscar miembros por año simbólico.
- RF3: Eliminar nodos conservando el orden del ABB.
- RF4: Mostrar recorridos genealógicos:
 - Inorden (línea temporal)
 - Preorden (jerarquía divina)
 - Postorden (generaciones)
- RF5: Representar líneas familiares dentro del árbol.

Requerimientos no funcionales (RNF)

- RNF1: Interfaz sencilla y compatible con evaluación académica.
- RNF2: Operaciones con eficiencia promedio $O(\log n)$.
- RNF3: Implementación en C++ bajo entorno Dev-C++.
- RNF4: Manejo robusto de entradas incorrectas.

3. Respuestas a preguntas guía

1. Información almacenada en cada nodo

Cada nodo del ABB representa un personaje mitológico.

Debe incluir:

- nombre (ej. Zeus, Hades, Cronos)
 - rol mitológico (Titán, Olímpico, Semidiós, Criatura)
 - año simbólico de nacimiento → clave del ABB
 - puntero izquierdo → entidades más antiguas
 - puntero derecho → entidades más recientes
- Esto permite mantener un árbol ordenado temporalmente y apto para consultas eficientes.

2. Inserción y eliminación sin romper el ABB

Inserción

Se compara el año nuevo con el nodo actual:

- menor → va al subárbol izquierdo
- mayor → va al subárbol derecho
- igual → se rechaza para evitar duplicados

El algoritmo es recursivo y garantiza la validez del ABB.

Eliminación

Se usa el método estándar del ABB:

1. Sin hijos: se elimina directamente
2. Un hijo: el hijo reemplaza al nodo
3. Dos hijos: se reemplaza con el sucesor inorden
(nodo más pequeño del subárbol derecho)

3. Métodos de recorrido

Tres recorridos clásicos permiten visualizar diferentes aspectos:

- Inorden (izq – raíz – der):
 - Lista cronológica de los seres mitológicos
- Preorden (raíz – izq – der):
 - Muestra jerarquía divina
- Postorden (izq – der – raíz):
 - Permite analizar generaciones completas

Cada uno responde a un tipo de consulta distinta.

4. Determinar si pertenece a una rama

Para saber si alguien pertenece a una línea ancestral:

1. buscar al ancestro (ej. Cronos, Zeus),
2. desde su nodo, recorrer preorden/inorden,
3. comprobar si aparece el miembro buscado.

También se pueden establecer ramas por rol:

- Titanes → primigenios
- Olímpicos → generación central
- Semidioses → descendientes
- Criaturas → derivados alternos

Si el miembro está dentro del subárbol de esa categoría, pertenece a la rama.

5. Balancear el árbol

Si se insertan años muy dispersos, el ABB puede volverse una lista.

Para evitarlo:

Métodos aplicables:

Reconstrucción balanceada (RECOMENDADO)

1. obtener nodos en inorden,
2. reconstruir tomando elemento medio como raíz,
3. subárboles con elementos anteriores/siguientes.

Complejidad: $O(n)$.

Rotaciones (AVL)

- izquierda
- derecha
- doble

CAPÍTULO 2 – PROTOTIPO

1. DESCRIPCIÓN DE LA ESTRUCTURA DE DATOS Y OPERACIONES

El prototipo usa un Árbol Binario de Búsqueda (ABB) para representar la genealogía de la mitología griega.

Cada nodo contiene información de un dios, titán, héroe u otro personaje, y queda ubicado en el árbol según una clave numérica única que deriva del año simbólico de nacimiento y el mes. Esta clave asegura que no haya duplicados exactos y mantiene el orden entre los nodos.

Los árboles de búsqueda binarios se construyen de forma que, para cada nodo:

- los valores menores se ubican en el subárbol izquierdo,
- los mayores en el subárbol derecho.

Esto permite operaciones eficientes —inserción, búsqueda y eliminación— siempre que el árbol se mantenga relativamente balanceado.

1.1 Estructura de cada nodo

Cada nodo almacena:

- int anio y int mes — datos cronológicos del personaje.
- int clave — combinación anio*100 + mes, que funciona como clave única para ordenar.
- string nombre — nombre mitológico.
- string rol — categoría o función, por ejemplo Titán, Olímpico, Héroe.
- Nodo* izq — puntero al subárbol con claves menores.
- Nodo* der — puntero al subárbol con claves mayores.

Este diseño permite que el árbol refleje un orden temporal y jerárquico, y que cada consulta o modificación sea guiada por la comparación de claves.

1.2 Operaciones implementadas

La siguiente tabla resume las operaciones fundamentales que implementa el prototipo del Árbol Binario de Búsqueda (ABB), explicando su propósito, comportamiento interno y relevancia dentro del sistema de genealogía mitológica:

Insertar

*Inserta un nuevo nodo en el ABB utilizando como clave compuesta anio*100 + mes. La función compara la clave del nuevo elemento con los nodos existentes y lo ubica recursivamente en el subárbol izquierdo (si es menor) o en el derecho (si es mayor). Si la clave ya existe, el algoritmo detiene la operación para preservar la propiedad fundamental del ABB (no duplicados).*

Buscar

Realiza una búsqueda exacta basada en la clave año+mes. Se emplea un recorrido recursivo que desciende por izquierda o derecha según la comparación con el nodo actual. La búsqueda concluye en tiempo proporcional a la altura del árbol, retornando el nodo encontrado o NULL.

Eliminar

Elimina un nodo del ABB manteniendo su estructura correcta. La función identifica tres situaciones: nodo hoja (se elimina directamente), nodo con un solo hijo (el hijo reemplaza al nodo), y nodo con dos hijos (se reemplaza por el sucesor inorden obtenido con la función mínima). Este procedimiento garantiza que la propiedad de ordenamiento del ABB se conserve tras la eliminación.

Recorridos

Implementa los recorridos clásicos del ABB: Inorden (Izq–Raíz–Der), que produce una secuencia cronológica ascendente; Preorden (Raíz–Izq–Der), que destaca jerarquía y estructura; y Postorden (Izq–Der–Raíz), empleado para visualizar descendencia completa. Cada recorrido responde a un patrón específico de análisis de la genealogía mitológica.

Visualización ASCII del árbol

Genera una representación estructural del ABB mediante caracteres ASCII. Esta visualización ubica la raíz en la parte superior y despliega ramas, nodos internos y hojas con indentación gradual. Permite inspeccionar la forma del árbol después de cada inserción, eliminación o búsqueda, facilitando su interpretación sin herramientas gráficas.

Validación de entradas

2. ALGORITMOS PRINCIPALES (PSEUDOCÓDIGO)

2.1 Pseudocódigo para crear un Árbol Binario de Búsqueda (Inserción)

```
● ● ●

PROCEDIMIENTO Insertar(raiz, anio, mes, nombre, rol)
    clave ← anio * 100 + mes           // Se genera clave compuesta única

    SI raiz ES NULL ENTONCES
        raiz ← NUEVO Nodo(anio, mes, nombre, rol)
        RETORNAR raiz
    FIN SI

    SI clave < raiz.clave ENTONCES
        raiz.izq ← Insertar(raiz.izq, anio, mes, nombre, rol)
    SINO SI clave > raiz.clave ENTONCES
        raiz.der ← Insertar(raiz.der, anio, mes, nombre, rol)
    SINO
        MOSTRAR "Clave duplicada: no se inserta"
    FIN SI

    RETORNAR raiz
FIN PROCEDIMIENTO
```

El proceso de inserción crea y posiciona un nuevo nodo dentro del Árbol Binario de Búsqueda comparando su clave con los nodos existentes. El algoritmo recorre el árbol de forma recursiva hasta encontrar la posición correcta, asegurando que los valores menores se ubiquen en el subárbol izquierdo y los mayores en el subárbol derecho. En caso de que la clave ya exista, la inserción se detiene para evitar duplicados y mantener la estructura válida del ABB.

2.2 Pseudocódigo para realizar recorridos del árbol

```
● ● ●

PROCEDIMIENTO Inorden(nodo)
    SI nodo ES NULL ENTONCES
        RETORNAR
    FIN SI

    Inorden(nodo.izq)                                // Visitar subárbol izquierdo
    MOSTRAR nodo                                     // Procesar nodo actual
    Inorden(nodo.der)                               // Visitar subárbol derecho
FIN PROCEDIMIENTO

PROCEDIMIENTO Preorden(nodo)
    SI nodo ES NULL ENTONCES
        RETORNAR
    FIN SI

    MOSTRAR nodo                                // Procesar nodo primero
    Preorden(nodo.izq)                           // Subárbol izquierdo
    Preorden(nodo.der)                           // Subárbol derecho
FIN PROCEDIMIENTO

PROCEDIMIENTO Postorden(nodo)
    SI nodo ES NULL ENTONCES
        RETORNAR
    FIN SI

    Postorden(nodo.izq)                           // Subárbol izquierdo
    Postorden(nodo.der)                           // Subárbol derecho
    MOSTRAR nodo                                // Procesar al final
FIN PROCEDIMIENTO
```

Los recorridos del árbol permiten analizar la estructura desde diferentes enfoques. El recorrido Inorden lista los nodos en orden ascendente según la clave; Preorden muestra primero la raíz y luego sus descendientes, ideal para analizar jerarquías; mientras que Postorden explora primero los hijos y finaliza con el nodo actual, permitiendo interpretar generaciones completas. Los tres recorridos se implementan mediante procedimientos recursivos simples y directos.

Recorrido INORDEN (Izquierda → Raíz → Derecha)

El recorrido Inorden visita primero el subárbol izquierdo, luego procesa el nodo actual y finalmente recorre el subárbol derecho.

En un Árbol Binario de Búsqueda, este recorrido siempre produce los elementos en orden ascendente según la clave, que en este sistema corresponde al año simbólico de nacimiento más el mes.

Aplicado a la genealogía mitológica, permite generar una línea temporal perfectamente ordenada, mostrando a los seres más antiguos primero y a los más recientes al final. Por ello, es el recorrido ideal para elaborar cronologías, etapas mitológicas y secuencias históricas dentro del árbol.

Recorrido PREORDEN (Raíz → Izquierda → Derecha)

El recorrido Preorden procesa primero la raíz, luego desciende por el subárbol izquierdo y finalmente explora el derecho.

Este método permite observar la jerarquía natural del árbol, ya que se inicia siempre desde el nodo más importante o más reciente de la genealogía (raíz del ABB inicial o cargado). En el contexto del árbol mitológico, el Preorden ofrece una vista estructurada de líneas familiares, mostrando a cada dios o titán junto a sus descendientes inmediatos. Es útil para representar organigramas mitológicos, árboles jerárquicos y estructuras donde el rol dominante o el ancestro principal debe visualizarse primero.

Recorrido POSTORDEN (Izquierda → Derecha → Raíz)

El Postorden recorre primero los subárboles izquierdo y derecho, procesando el nodo actual únicamente al final.

Este enfoque revela la estructura desde los elementos más específicos hasta los más generales, es decir, desde las generaciones más jóvenes hacia los progenitores.

Es particularmente adecuado para analizar descendencias completas, estudiar ramas familiares específicas o preparar la eliminación segura de nodos (ya que garantiza que todos los descendientes se procesen antes que el nodo padre).

En genealogía mitológica, el Postorden permite identificar líneas de sangre completas y visualizar cómo se organizan las distintas familias dentro del árbol.

2.3 Pseudocódigo: Crear un Árbol Binario (Insertar nodo en ABB)

```
● ● ●

ALGORITMO Insertar(raiz, nacimiento, nombre, rol)
    SI raiz ES NULO ENTONCES
        // Crear un nuevo nodo con la información proporcionada
        nuevoNodo ← CrearNodo(nacimiento, nombre, rol)
        RETORNAR nuevoNodo
    FIN SI

    // Si la fecha de nacimiento es menor que la raíz, insertamos en el subárbol izquierdo
    SI nacimiento < raiz.nacimiento ENTONCES
        raiz.izq ← Insertar(raiz.izq, nacimiento, nombre, rol)

    // Si la fecha de nacimiento es mayor que la raíz, insertamos en el subárbol derecho
    SINO SI nacimiento > raiz.nacimiento ENTONCES
        raiz.der ← Insertar(raiz.der, nacimiento, nombre, rol)

    // Si la fecha de nacimiento es igual a la raíz, mostramos un mensaje de error
    SINO
        MOSTRAR "Clave duplicada. No se puede insertar."
    FIN SI

    // Retornamos la raíz (sin cambios si no se insertó nada)
    RETORNAR raiz
FIN ALGORITMO
```

La función *Insertión* coloca cada nuevo ser mitológico en el lugar correcto del ABB comparando su año simbólico de nacimiento con el de los nodos existentes.

2.4 Pseudocódigo: eliminar un miembro del ABB

```
ALGORITMO Eliminar(raiz, nacimiento)
    SI raiz ES NULO ENTONCES
        // No se encontró la clave
        RETORNAR NULL
    FIN SI

    // Buscar el nodo a eliminar
    SI nacimiento < raiz.nacimiento ENTONCES
        raiz.izq ← Eliminar(raiz.izq, nacimiento)

    SINO SI nacimiento > raiz.nacimiento ENTONCES
        raiz.der ← Eliminar(raiz.der, nacimiento)

    SINO
        // Caso 1: Nodo sin hijos (hoja)
        SI raiz.izq ES NULL Y raiz.der ES NULL ENTONCES
            eliminar raiz
            RETORNAR NULL
        FIN SI

        // Caso 2: Nodo con un solo hijo (derecho)
        SI raiz.izq ES NULL ENTONCES
            temp ← raiz.der
            eliminar raiz
            RETORNAR temp
        FIN SI

        // Caso 2: Nodo con un solo hijo (izquierdo)
        SI raiz.der ES NULL ENTONCES
            temp ← raiz.izq
            eliminar raiz
            RETORNAR temp
        FIN SI

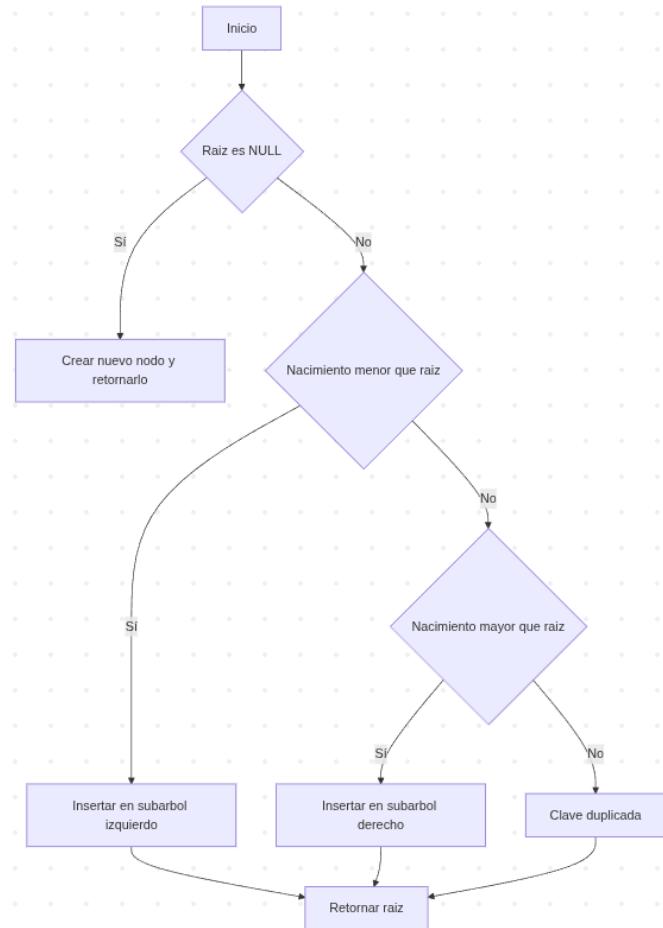
        // Caso 3: Nodo con dos hijos
        sucesor ← Minimo(raiz.der)
        copiar datos de sucesor a raiz
        raiz.der ← Eliminar(raiz.der, sucesor.nacimiento)
    FIN SI

    RETORNAR raiz
FIN ALGORITMO
```

Este algoritmo mantiene la estructura del árbol al eliminar un nodo, usando el sucesor inorden si es necesario.

3. DIAGRAMAS DE FLUJO

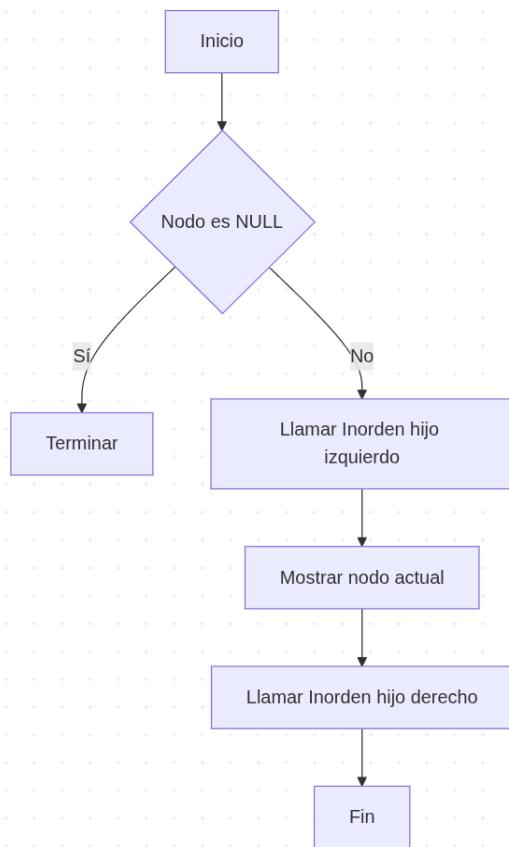
DIAGRAMA 1 – Proceso de Inserción en un Árbol Binario de Búsqueda (ABB)



Este diagrama muestra cómo el algoritmo decide en qué parte del árbol insertar un nuevo miembro basado en su año simbólico.

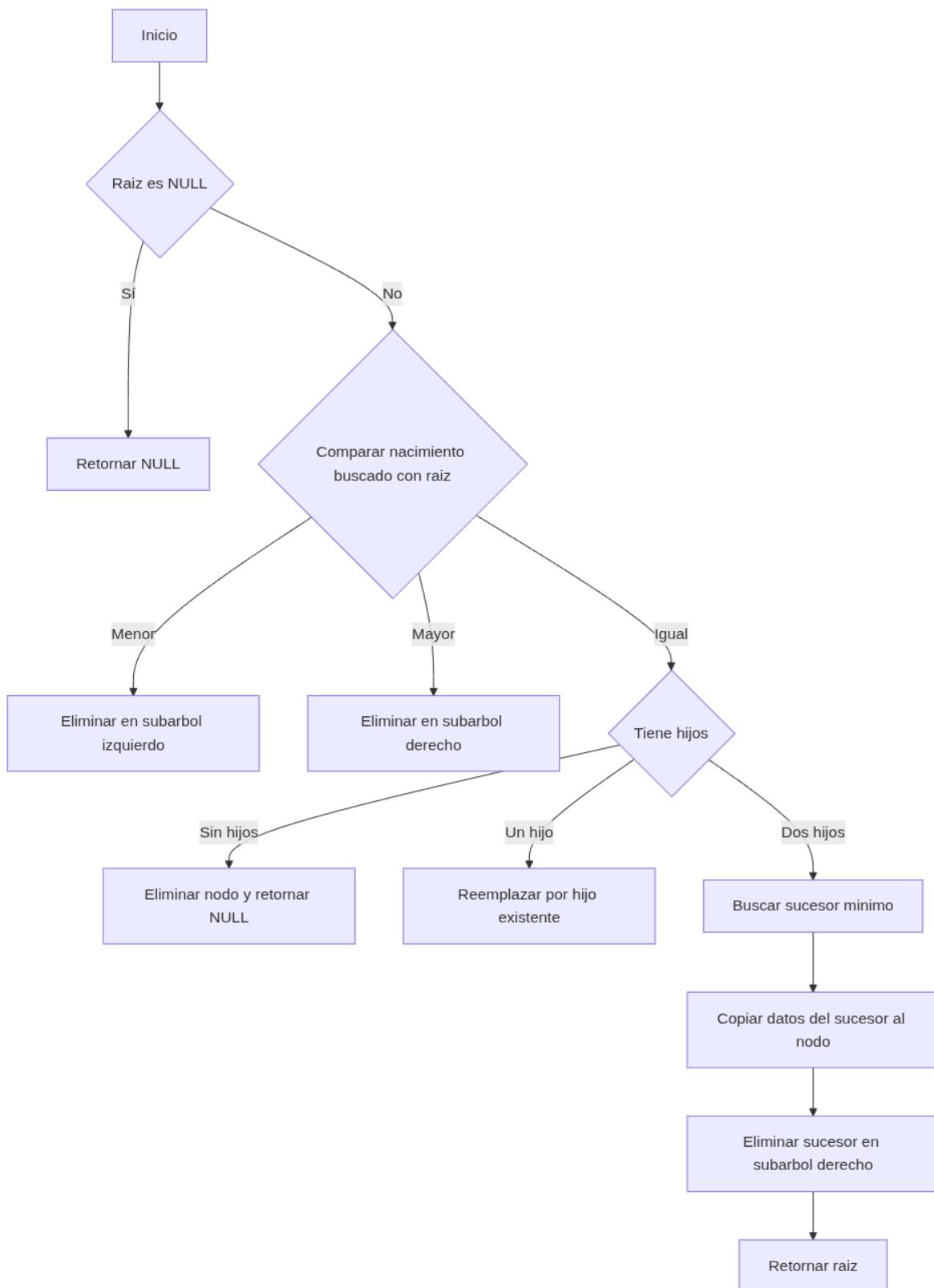
Si el árbol está vacío, crea la raíz; si el año es menor avanza por el subárbol izquierdo, si es mayor avanza por el derecho; y si es igual, detecta duplicados.

DIAGRAMA 2 — Recorrido INORDEN (Izq – Raíz – Der)



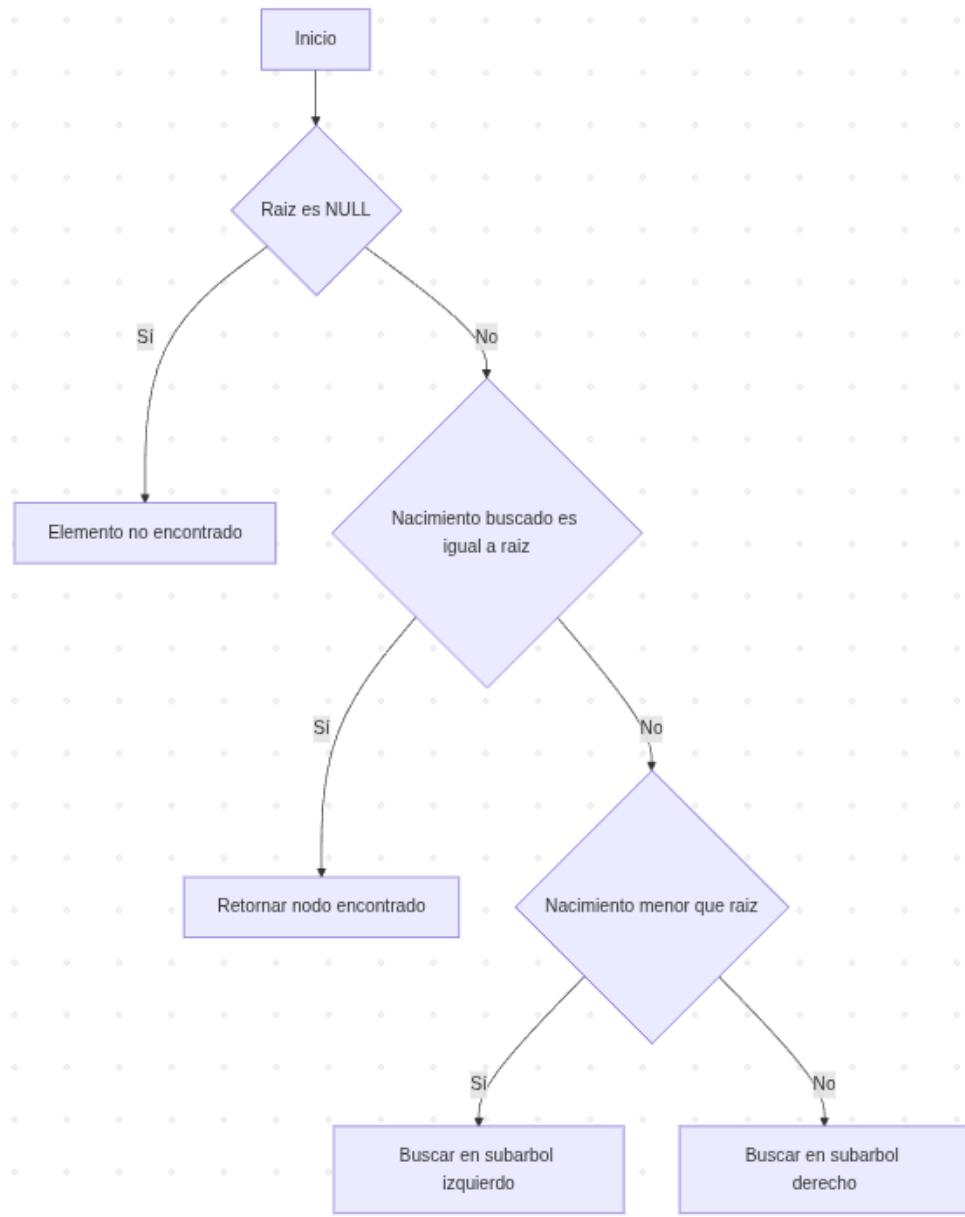
Este recorrido muestra los elementos del árbol en orden ascendente según su año simbólico. Primero explora el subárbol izquierdo (miembros más antiguos), luego muestra el nodo actual y finalmente el subárbol derecho (miembros más recientes).

DIAGRAMA 3 – Eliminación en un Árbol Binario de Búsqueda (ABB)



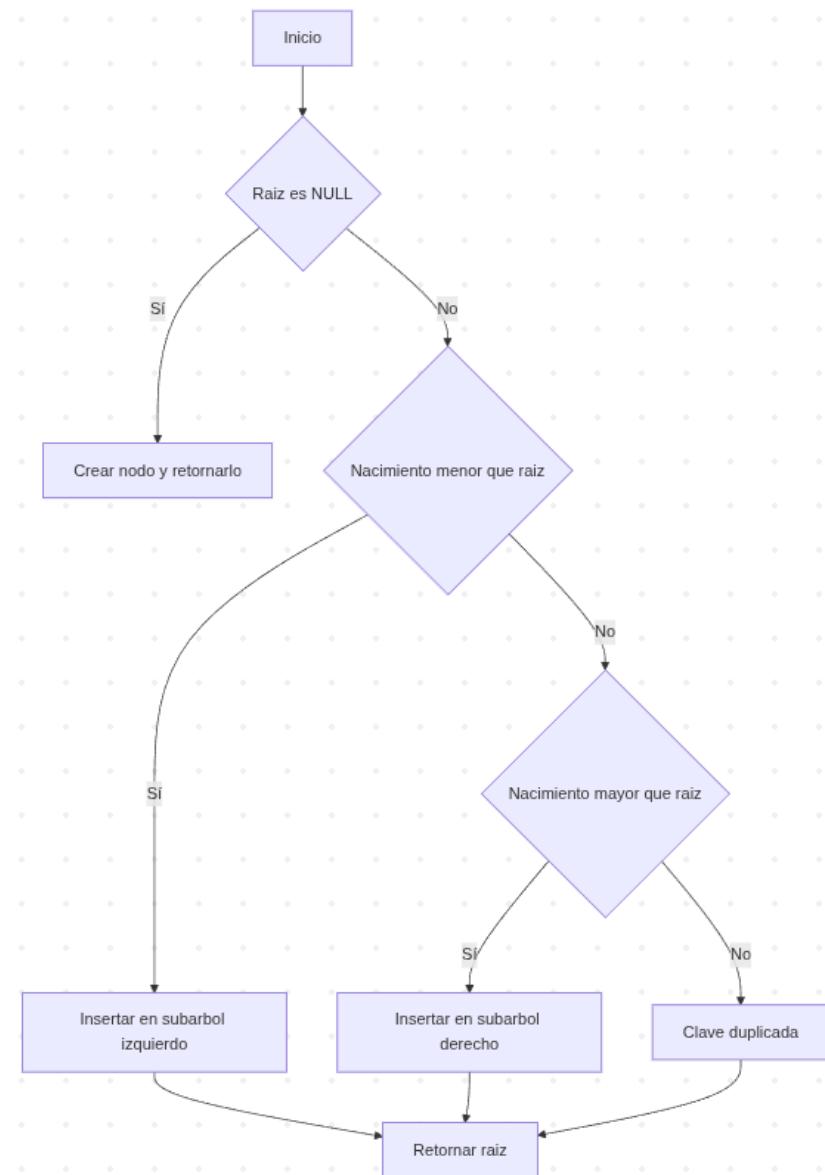
Este diagrama explica el proceso para eliminar un nodo de un ABB manteniendo el orden. El algoritmo distingue entre nodos sin hijos, con un hijo, o con dos hijos (donde es necesario reemplazar con el sucesor mínimo del subárbol derecho).

DIAGRAMA 4 – Búsqueda en ABB



El algoritmo de búsqueda compara la clave buscada con el nodo actual y decide si explorar izquierdo, derecha o finalizar si encuentra el valor.

DIAGRAMA 5 – Inserción en ABB (para referencia)



El proceso inserta un nuevo nodo respetando el orden del ABB: menores a la izquierda, mayores a la derecha, evitando duplicados.

4. AVANCE DEL CÓDIGO FUENTE

1.1 Definición del nodo

```
● ● ●  
struct Nodo {  
    int anio;  
    int mes;  
    int clave;  
    string nombre;  
    string rol;  
    Nodo *izq;  
    Nodo *der;  
  
    Nodo(int a, int m, const string &nom, const string &r) {  
        anio = a;  
        mes = m;  
        clave = a * 100 + m;  
        nombre = nom;  
        rol = r;  
        izq = NULL;  
        der = NULL;  
    }  
};
```

Este bloque define la estructura fundamental del Árbol Binario de Búsqueda.

Cada nodo representa a un ser mitológico y almacena su año y mes (para formar una clave única), nombre, rol y los punteros a sus hijos izquierdo y derecho. Esta estructura permite organizar a los personajes siguiendo un orden temporal.

1.2 Función Insertar

```
● ● ●  
Nodo *insertar(Nodo *raiz, int anio, int mes, const string &nombre,  
                const string &rol) {  
  
    int clave = anio * 100 + mes;  
  
    if (raiz == NULL)  
        return new Nodo(anio, mes, nombre, rol);  
  
    if (clave < raiz->clave)  
        raiz->izq = insertar(raiz->izq, anio, mes, nombre, rol);  
    else if (clave > raiz->clave)  
        raiz->der = insertar(raiz->der, anio, mes, nombre, rol);  
    else  
        cout << "Aviso: ya existe un miembro con ese anio y mes.\n";  
  
    return raiz;  
}
```

Esta función inserta nuevos personajes en el ABB respetando el orden definido por la clave año–mes. Si la clave es menor, va al subárbol izquierdo; si es mayor, al derecho. Si

coincide, se detecta un duplicado. Esto garantiza que la genealogía se mantenga ordenada cronológicamente.

1.3 Función Buscar

```
● ● ●  
Nodo *buscarExacto(Nodo *raiz, int anio, int mes) {  
    int clave = anio * 100 + mes;  
  
    if (raiz == NULL)  
        return NULL;  
  
    if (clave == raiz->clave)  
        return raiz;  
  
    if (clave < raiz->clave)  
        return buscarExacto(raiz->izq);  
  
    return buscarExacto(raiz->der);  
}
```

Este algoritmo localiza un nodo exacto en el ABB comparando la clave buscada con la del nodo actual. Según el valor, continúa la búsqueda por la izquierda o la derecha. Su eficiencia proviene del orden del ABB, permitiendo localizar personajes rápidamente.

1.4 Función ELiminar

```
● ● ●

Nodo *eliminarMiembro(Nodo *raiz, int anio, int mes) {
    if (raiz == NULL)
        return raiz;

    int clave = anio * 100 + mes;

    if (clave < raiz->clave)
        raiz->izq = eliminarMiembro(raiz->izq, anio, mes);
    else if (clave > raiz->clave)
        raiz->der = eliminarMiembro(raiz->der, anio, mes);
    else {
        if (raiz->izq == NULL) {
            Nodo *temp = raiz->der;
            delete raiz;
            return temp;
        }
        if (raiz->der == NULL) {
            Nodo *temp = raiz->izq;
            delete raiz;
            return temp;
        }
        Nodo *suc = minimo(raiz->der);
        raiz->anio = suc->anio;
        raiz->mes = suc->mes;
        raiz->clave = suc->clave;
        raiz->nombre = suc->nOMBRE;
        raiz->rol = suc->rol;
        raiz->der = eliminarMiembro(raiz->der, suc->anio, suc->mes);
    }
    return raiz;
}
```

La función `eliminar` gestiona correctamente los tres casos clásicos del ABB: nodos sin hijos, con un solo hijo y con dos hijos. En este último caso utiliza el sucesor inorden para mantener el árbol ordenado. Este método asegura la integridad estructural del árbol tras cada eliminación.

1.5 Implementación en ASCII del arbol

```
● ● ●

void imprimirArbolAscii(Nodo *raiz, string prefix, bool esUltimo) {
    if (raiz == NULL)
        return;

    cout << prefix << (esUltimo ? "└── " : "├── ")
    << raiz->nombre << " [" << raiz->anio << "-" << raiz->mes
    << "] (" << raiz->rol << ")\n";

    string nuevoPrefix = prefix + (esUltimo ? "      " : "|      ");

    if (raiz->izq != NULL && raiz->der != NULL) {
        imprimirArbolAscii(raiz->izq, nuevoPrefix, false);
        imprimirArbolAscii(raiz->der, nuevoPrefix, true);
    } else if (raiz->izq != NULL) {
        imprimirArbolAscii(raiz->izq, nuevoPrefix, true);
    } else if (raiz->der != NULL) {
        imprimirArbolAscii(raiz->der, nuevoPrefix, true);
    }
}
```

Este procedimiento genera una representación visual del ABB en formato ASCII, mostrando jerarquía y ramas del árbol. Gracias a los prefijos y caracteres especiales, el usuario puede visualizar claramente las relaciones genealógicas entre los personajes mitológicos.

Capítulo 3 – Solución Final

1. Código limpio, bien comentado y estructurado.

A continuación se incluye el código fuente final del sistema ABB mitológico. Este código está ordenado en módulos lógicos (modelo del nodo, validaciones, operaciones del ABB, carga inicial y menú). Se añadieron comentarios breves y técnicos que explican únicamente los puntos más relevantes del funcionamiento, tal como se solicita en informes académicos:

```
14 // Constructor del nodo
15 Nodo(int a, int m, const string &nom, const string &r) {
16     anio = a;
17     mes = m;
18     clave = a * 100 + m;
19     nombre = nom;
20     rol = r;
21     izq = NULL;
22     der = NULL;
23 }
24 };
25
26 // PROTOTIPOS
27 Nodo *insertar(Nodo *raiz, int anio, int mes, const string &nombre,
28                 const string &rol);
29 Nodo *eliminarMiembro(Nodo *raiz, int anio, int mes);
30 Nodo *buscarExacto(Nodo *raiz, int anio, int mes);
31 Nodo *minimo(Nodo *nodo);
32 void inorden(Nodo *raiz);
33 void preorden(Nodo *raiz);
34 void postorden(Nodo *raiz);
35 void imprimirArbolAscii(Nodo *raiz, string prefix, bool esUltimo);
36 bool leerIntRango(const string &prompt, int &valor, int minValue, int maxValue);
37
```

*En este bloque se define la estructura fundamental del Árbol Binario de Búsqueda: el nodo. Cada nodo almacena la información de un personaje mitológico junto con los punteros que permiten enlazarlo con sus descendientes dentro del árbol. El nodo contiene cinco datos principales: el año y mes simbólicos de nacimiento, una clave numérica única calculada como año * 100 + mes, el nombre del personaje y su rol mitológico. Además, incorpora dos punteros (izq y der) que representan a los nodos situados en el subárbol izquierdo y derecho respectivamente, permitiendo así organizar los personajes de forma ordenada. Finalmente, se implementa un constructor, encargado de inicializar todos los campos del nodo y asignar nulos a los punteros hijos al momento de crear un nuevo registro.*

```

38 // INSERTAR
39 /*
40 Inserta un nuevo miembro en el ABB según la clave anio*100 + mes.
41 Si la raíz es NULL, crea un nuevo nodo.
42 Si la clave es menor va a la izquierda, si es mayor va a la derecha.
43 Si la clave ya existe, muestra un aviso.
44 */
45 Nodo *insertar(Nodo *raiz, int anio, int mes, const string &nombre,
46                 const string &rol) {
47     int clave = anio * 100 + mes;
48
49     // Caso base: insertar en árbol vacío
50     if (raiz == NULL) {
51         return new Nodo(anio, mes, nombre, rol);
52     }
53
54     // Inserción recursiva en izquierda o derecha
55     if (clave < raiz->clave) {
56         raiz->izq = insertar(raiz->izq, anio, mes, nombre, rol);
57     } else if (clave > raiz->clave) {
58         raiz->der = insertar(raiz->der, anio, mes, nombre, rol);
59     } else {
60         cout << "Aviso: ya existe un miembro con ese anio y mes.\n";
61     }
62
63     return raiz;
64 }
```

En este bloque se implementa la función responsable de insertar un nuevo nodo en el Árbol Binario de Búsqueda (ABB). Primero se genera la clave única combinando el año y el mes. Si el árbol está vacío, la función crea y retorna un nuevo nodo como raíz. En caso contrario, se ejecuta un proceso recursivo para decidir si el dato debe ubicarse en el subárbol izquierdo (cuando la clave es menor) o en el derecho (cuando la clave es mayor). Si la clave ya existe, se evita duplicar información mostrando un aviso. Finalmente, se retorna la raíz del árbol actualizada, garantizando que la estructura mantenga el orden propio de un ABB.

```

66 // BUSCAR
67 /*
68 Busca un nodo exacto en el ABB según su clave.
69 Retorna un puntero al nodo o NULL si no existe.
70 */
71 Nodo *buscarExacto(Nodo *raiz, int anio, int mes) {
72     int clave = anio * 100 + mes;
73
74     if (raiz == NULL)
75         return NULL;
76
77     if (clave == raiz->clave)
78         return raiz;
79
80     if (clave < raiz->clave)
81         return buscarExacto(raiz->izq, anio, mes);
82
83     return buscarExacto(raiz->der, anio, mes);
84 }
85
86 // MÍNIMO
87 /*
88 Encuentra el nodo con la clave mínima.
89 Se usa para encontrar el sucesor al eliminar un nodo.
90 */
91 Nodo *minimo(Nodo *nodo) {
92     while (nodo != NULL && nodo->izq != NULL)
93         nodo = nodo->izq;
94     return nodo;
95 }
96

```

En estas líneas se implementan dos funciones clave para la gestión del Árbol Binario de Búsqueda. La primera, `buscarExacto()`, realiza una búsqueda recursiva para localizar un nodo utilizando su clave única formada por año y mes. La función compara la clave buscada con la del nodo actual y decide avanzar hacia el subárbol izquierdo si la clave es menor, o hacia el derecho si es mayor, retornando el nodo encontrado o NULL si no existe.

La segunda función, `minimo()`, recorre sucesivamente el subárbol izquierdo hasta encontrar el nodo más pequeño del ABB. Esta operación es esencial durante la eliminación de nodos, ya que permite identificar el sucesor inorder cuando se reemplaza un nodo con dos hijos.

```

97 // ELIMINAR
98 /*
99 Elimina un nodo del ABB respetando las reglas:
100 - Caso 1: nodo sin hijos → se elimina directamente.
101 - Caso 2: nodo con un hijo → se reemplaza por ese hijo.
102 - Caso 3: nodo con dos hijos → se reemplaza con su sucesor inorden.
103 */
104 Nodo *eliminarMiembro(Nodo *raiz, int anio, int mes) {
105     if (raiz == NULL)
106         return raiz;
107
108     int clave = anio * 100 + mes;
109
110    // Buscar nodo a eliminar
111    if (clave < raiz->clave) {
112        raiz->izq = eliminarMiembro(raiz->izq, anio, mes);
113    } else if (clave > raiz->clave) {
114        raiz->der = eliminarMiembro(raiz->der, anio, mes);
115    } else {
116        // Nodo encontrado: manejar casos de eliminación
117
118        // Caso 1: sin hijo izquierdo
119        if (raiz->izq == NULL) {
120            Nodo *temp = raiz->der;
121            delete raiz;
122            return temp;
123        }
124
125        // Caso 2: sin hijo derecho
126        else if (raiz->der == NULL) {
127            Nodo *temp = raiz->izq;
128            delete raiz;
129            return temp;
130        }
131
132        // Caso 3: dos hijos → tomar sucesor
133        Nodo *suc = minimo(raiz->der);
134
135        // Copiar datos del sucesor
136        raiz->anio = suc->anio;
137        raiz->mes = suc->mes;
138        raiz->clave = suc->clave;
139        raiz->nombre = suc->nombre;
140        raiz->rol = suc->rol;
141
142        // Eliminar sucesor
143        raiz->der = eliminarMiembro(raiz->der, suc->anio, suc->mes);
144    }
145
146    return raiz;
147 }

```

En este bloque se implementa la función `eliminarMiembro()`, responsable de retirar un nodo del Árbol Binario de Búsqueda manteniendo sus reglas estructurales. Primero, la función localiza el nodo comparando la clave objetivo con la del nodo actual y desplazándose recursivamente hacia la izquierda o derecha. Una vez encontrado, se gestionan los tres casos clásicos de eliminación en un ABB:

- *Nodo sin hijo izquierdo, donde se reemplaza directamente por su subárbol derecho.*
 - *Nodo sin hijo derecho, reemplazándolo por su subárbol izquierdo.*
- *Nodo con dos hijos, donde se obtiene el sucesor inorden usando la función `mínimo()`, se copian sus datos y luego se elimina recursivamente ese sucesor.*

Este proceso garantiza que el árbol conserve su ordenación y estructura tras la eliminación.

```

149 // RECORRIDOS
150 /*
151 Imprime árbol en INORDEN (izq - raíz - der)
152 Muestra miembros ordenados por fecha.
153 */
154 void inorder(Nodo *raiz) {
155     if (raiz) {
156         inorder(raiz->izq);
157         cout << raiz->nombre << "(" << raiz->anio << "-" << raiz->mes << ") - "
158             << raiz->rol << endl;
159         inorder(raiz->der);
160     }
161 }
162
163 /*
164 PREORDEN (raíz - izq - der)
165 */
166 void preorden(Nodo *raiz) {
167     if (raiz) {
168         cout << raiz->nombre << "(" << raiz->anio << "-" << raiz->mes << ") - "
169             << raiz->rol << endl;
170         preorden(raiz->izq);
171         preorden(raiz->der);
172     }
173 }
174
175 /*
176 POSTORDEN (izq - der - raíz)
177 */
178 void postorden(Nodo *raiz) {
179     if (raiz) {
180         postorden(raiz->izq);
181         postorden(raiz->der);
182         cout << raiz->nombre << "(" << raiz->anio << "-" << raiz->mes << ") - "
183             << raiz->rol << endl;
184     }
185 }
186

```

En este bloque se implementan los tres métodos clásicos de recorrido de un Árbol Binario de Búsqueda (ABB): `inorden`, `preorden` y `postorden`. Cada función utiliza recursividad para visitar sistemáticamente cada nodo del árbol siguiendo un orden específico.

- *Inorden procesa primero el subárbol izquierdo, luego el nodo y finalmente el derecho, lo que permite mostrar los personajes ordenados por fecha (clave).*
- *Preorden imprime el nodo antes de visitar sus hijos, útil para mostrar la jerarquía general del árbol o exportar su estructura.*
- *Postorden recorre primero ambos hijos y deja el nodo al final, apropiado para operaciones de borrado o análisis estructural.*

Estas funciones permiten visualizar el árbol desde distintas perspectivas y son esenciales para recorrer y procesar su contenido.

```

185
186 // ASCII TREE
187 /*
188 * Imprime el árbol de forma visual tipo diagrama.
189 * prefix → conforma el "nivel" visual
190 * esUltimo → decide entre └ o ┌
191 */
192
193 void imprimirArbolAscii(Nodo *raiz, string prefix, bool esUltimo) {
194     if (raiz == NULL)
195         return;
196
197     cout << prefix;
198     cout << (esUltimo ? "└ " : "┌ ");
199     cout << raiz->nombre << "[" << raiz->anio << "-" << raiz->mes << "] (" << raiz->rol << ")\n";
200
201     string nuevoPrefix = prefix + (esUltimo ? "    " : " |  ");
202
203     if (raiz->izq != NULL && raiz->der != NULL) {
204         imprimirArbolAscii(raiz->izq, nuevoPrefix, false);
205         imprimirArbolAscii(raiz->der, nuevoPrefix, true);
206     } else if (raiz->izq != NULL) {
207         imprimirArbolAscii(raiz->izq, nuevoPrefix, true);
208     } else if (raiz->der != NULL) {
209         imprimirArbolAscii(raiz->der, nuevoPrefix, true);
210     }
211 }
212
213
214 // VALIDACIÓN
215 /*
216 Lee un entero validado dentro de un rango específico.
217 Solo acepta dígitos.
218 */
219 bool leerIntRango(const string &prompt, int &valor, int minValue, int maxValue) {
220     string linea;
221     while (true) {
222         cout << prompt;
223         cin >> linea;
224
225         // Validar solo números
226         bool ok = true;
227         for (char c : linea)
228             if (!isdigit(c))
229                 ok = false;
230
231         if (!ok) {
232             cout << "Entrada inválida. Solo dígitos.\n";
233             continue;
234         }
235
236         valor = stoi(linea);
237
238         if (valor < minValue || valor > maxValue) {
239             cout << "Fuera de rango.\n";
240             continue;
241         }
242
243     return true;
244 }
245
246

```

Este bloque implementa `imprimirArbolAscii`, una función encargada de generar una representación visual del árbol binario usando caracteres ASCII. El procedimiento utiliza recursividad y un sistema de prefijos para mostrar la estructura jerárquica del árbol, indicando ramas, niveles y nodos hijos de manera clara.

El parámetro `prefix` mantiene la sangría acumulada según la profundidad, mientras que `esUltimo` determina si un nodo se dibuja con “└” (último hijo) o “┌” (aún hay nodos hermanos). La función imprime el contenido del nodo —nombre, fecha y rol— y luego continúa con el subárbol izquierdo y/o derecho, preservando el formato visual.

Este módulo facilita comprender la forma real del ABB, permitiendo observar cómo crecen las ramas, cómo se distribuyen los nodos y cómo se equilibran (o no) las inserciones en el árbol.

```

247 // CARGAR MITOLOGÍA
248 /*
249 Inserta 15 dioses/héroes/titanes predefinidos para comenzar con un árbol
250 grande.
251 */
252 Nodo *cargarMitologia(Nodo *raiz) {
253     raiz = insertar(raiz, 10, 6, "Zeus", "Rey Olimpico"); // raiz inicial
254
255     // Subárbol izquierdo
256     raiz = insertar(raiz, 5, 3, "Cronos", "Titan");
257     raiz = insertar(raiz, 3, 2, "Oceano", "Titan");
258     raiz = insertar(raiz, 2, 1, "Ouranos", "Primigenio");
259     raiz = insertar(raiz, 4, 5, "Rea", "Titanide");
260     raiz = insertar(raiz, 7, 8, "Hades", "Olimpico");
261     raiz = insertar(raiz, 6, 4, "Hestia", "Olimpica");
262     raiz = insertar(raiz, 8, 9, "Poseidon", "Olimpico");
263
264     // Subárbol derecho
265     raiz = insertar(raiz, 15, 7, "Atenea", "Olimpica");
266     raiz = insertar(raiz, 13, 2, "Apolo", "Olimpico");
267     raiz = insertar(raiz, 12, 1, "Artemisa", "Olimpica");
268     raiz = insertar(raiz, 14, 5, "Hermes", "Olimpico");
269     raiz = insertar(raiz, 18, 3, "Heracles", "Heroe");
270     raiz = insertar(raiz, 17, 9, "Perseo", "Heroe");
271     raiz = insertar(raiz, 20, 4, "Teseo", "Heroe");
272
273     return raiz;
274 }
275
276 // PROGRAMA PRINCIPAL
277 int main() {
278     Nodo *raiz = NULL;
279
280     // Carga inicial de 15 personajes
281     raiz = cargarMitologia(raiz);
282
283     cout << "\n==== Árbol genealógico mitológico cargado (15 miembros) ===\n\n";
284     imprimirArbolAscii(raiz, "", true);
285
286     int opcion;
287
288     // Menú principal
289     do {
290         cout << "\nMENU:\n";
291         cout << "1. Insertar\n";
292         cout << "2. Buscar (anio+mes)\n";
293         cout << "3. Eliminar\n";
294         cout << "4. Inorden\n";
295         cout << "5. Preorden\n";
296         cout << "6. Postorden\n";
297         cout << "7. Ver arbol\n";
298         cout << "0. Salir\n";
299         cout << "Opcion: ";
300         cin >> opcion;
301

```

La primera parte de este bloque implementa la función leerIntRango, que se encarga de validar entradas numéricas para impedir que el usuario ingrese caracteres inválidos o valores fuera de un rango permitido. Para lograrlo, la función captura la entrada como cadena, verifica que todos sus caracteres sean dígitos, convierte el texto a entero y luego comprueba si el valor está dentro de los límites establecidos; en caso contrario, solicita la entrada nuevamente, garantizando así que el programa nunca falle por datos mal ingresados. Después de esa validación aparece el inicio de la función cargarMitología, encargada de poblar automáticamente el Árbol Binario de Búsqueda con un conjunto inicial de dioses, titanes y héroes. Este bloque prepara la inserción de los 15 personajes que conforman el árbol base, permitiendo que el programa arranque con una estructura grande, ordenada y lista para pruebas de búsqueda, eliminación y recorridos.

2. Capturas de pantalla de las ventanas de ejecución con las diversas pruebas de validación de datos

2.1 Validación de entradas insertar

```
> ./main

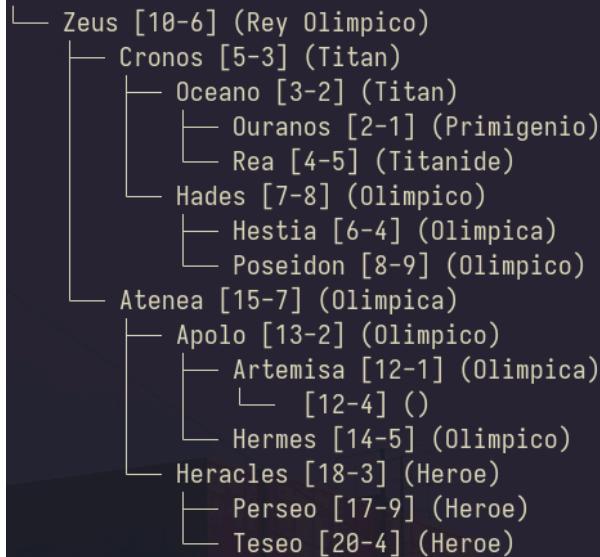
== Arbol genealógico mitológico cargado (15 miembros) ==

└── Zeus [10-6] (Rey Olimpico)
    ├── Cronos [5-3] (Titan)
    │   ├── Oceano [3-2] (Titan)
    │   │   ├── Ouranos [2-1] (Primigenio)
    │   │   └── Rea [4-5] (Titanide)
    │   └── Hades [7-8] (Olimpico)
    │       ├── Hestia [6-4] (Olimpica)
    │       └── Poseidon [8-9] (Olimpico)
    └── Atenea [15-7] (Olimpica)
        ├── Apolo [13-2] (Olimpico)
        │   ├── Artemisa [12-1] (Olimpica)
        │   └── Hermes [14-5] (Olimpico)
        └── Heracles [18-3] (Heroe)
            ├── Perseo [17-9] (Heroe)
            └── Teseo [20-4] (Heroe)

MENU:
1. Insertar
2. Buscar (anio+mes)
3. Eliminar
4. Inorden
5. Preorden
6. Postorden
7. Ver arbol
0. Salir
Opcion: 1
Anio: adsfas
Entrada invalida. Solo digitos.
Anio: 12
Mes (1-12): 13
Fuerza de rango.
Mes (1-12): 4
```

El programa implementa un sistema de validación robusto para asegurar que todos los datos ingresados por el usuario sean correctos antes de operar con el Árbol Binario de Búsqueda. La función leerIntRango verifica primero que la entrada esté compuesta únicamente por dígitos, evitando errores comunes como letras o símbolos que podrían detener la ejecución. Luego convierte la cadena a entero y comprueba que el valor esté dentro del rango permitido (por ejemplo, meses entre 1 y 12), forzando al usuario a corregir cualquier dato fuera de los límites establecidos. Este mecanismo garantiza que todas las operaciones del árbol (insertar, buscar, eliminar) reciban valores válidos, evitando nodos corruptos, claves inválidas o fallos inesperados. Gracias a esta validación estricta, el sistema mantiene su integridad y funcionamiento estable incluso ante entradas erróneas del usuario.

2.2 Buscar miembros



MENU:

1. Insertar
 2. Buscar (anio+mes)
 3. Eliminar
 4. Inorden
 5. Preorden
 6. Postorden
 7. Ver arbol
 0. Salir
- Opcion: 2
Anio: 1212
Mes: 12
No existe.

La función `buscarExacto` implementa una búsqueda recursiva dentro del Árbol Binario de Búsqueda utilizando la clave compuesta $\text{anio} * 100 + \text{mes}$. La función compara la clave solicitada con la del nodo actual y decide si continuar por el subárbol izquierdo o derecho según el orden del ABB. Si encuentra coincidencia, retorna el nodo; si llega a un nodo nulo, significa que el elemento no existe. Esta operación garantiza eficiencia $O(\log n)$ en árboles equilibrados y mantiene coherencia con el criterio de ordenamiento usado en las inserciones.

2.3 Eliminar miembros

```
MENU:  
1. Insertar  
2. Buscar (anio+mes)  
3. Eliminar  
4. Inorden  
5. Preorden  
6. Postorden  
7. Ver arbol  
0. Salir  
Opcion: 3  
Anio: 4312  
Mes: 12312  
Fuera de rango.  
Mes: 12  
  
└── Zeus [10-6] (Rey Olimpico)  
    ├── Cronos [5-3] (Titan)  
    │   ├── Oceano [3-2] (Titan)  
    │   └── Ouranos [2-1] (Primigenio)  
    ├── Rea [4-5] (Titanide)  
    ├── Hades [7-8] (Olimpico)  
    │   ├── Hestia [6-4] (Olimpica)  
    │   └── Poseidon [8-9] (Olimpico)  
    ├── Atenea [15-7] (Olimpica)  
    │   ├── Apolo [13-2] (Olimpico)  
    │   ├── Artemisa [12-1] (Olimpica)  
    │   │   └── [12-4] ()  
    │   └── Hermes [14-5] (Olimpico)  
    ├── Heracles [18-3] (Heroe)  
    │   ├── Perseo [17-9] (Heroe)  
    │   └── Teseo [20-4] (Heroe)
```

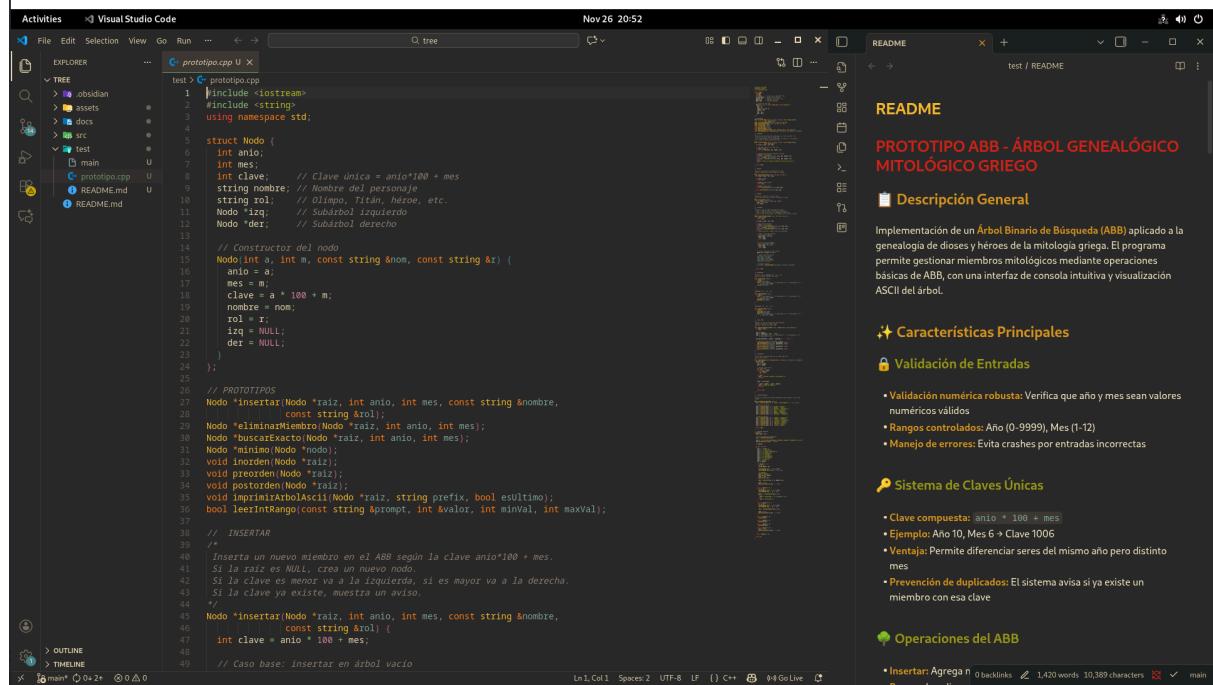
La función eliminarMiembro implementa el procedimiento estándar de eliminación en un Árbol Binario de Búsqueda usando la clave compuesta anio*100 + mes. Primero localiza el nodo a eliminar siguiendo el orden del ABB y, una vez encontrado, maneja tres escenarios: si el nodo no tiene hijo izquierdo se reemplaza directamente por su subárbol derecho; si no tiene hijo derecho se reemplaza por su subárbol izquierdo; y si posee ambos hijos se selecciona su sucesor inorden mediante la función minimo, copiando sus datos para mantener el orden del árbol y eliminando luego al sucesor del subárbol derecho. Esta operación conserva la estructura del ABB y garantiza consistencia en las relaciones jerárquicas del árbol genealógico.

Se probó la función de búsqueda ingresando claves válidas e inválidas. El sistema retorna el nodo correspondiente o un mensaje indicando que no existe.

Capítulo 4 – Evidencias de Trabajo Colaborativo

1. Repositorio con Control de Versiones

Para el desarrollo de este proyecto utilicé un [repositorio en GitHub](#) con el nombre de tree, lo cual me permitió organizar mi trabajo, llevar un seguimiento claro de cada modificación y registrar los avances de forma profesional durante todo el proceso. Gracias al control de versiones pude revisar qué cambios realizaba en cada etapa y corregir errores sin perder el progreso anterior.



The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER:** Shows the project structure with files: prototipo.cpp, test, README.md, and README.md.
- CODE EDITOR:** Displays the content of prototipo.cpp. The code implements a binary search tree (ABB) for managing mythological members based on birth year and month. It includes a struct for nodes, insertion logic, and search functions.
- RIGHT SIDE:** The README file is open, providing a brief description of the project: "PROTOTIPO ABB - ÁRBOL GENEALÓGICO MITOLÓGICO GRIEGO". It also lists "Características Principales" (Characteristics), "Validación de Entradas" (Input Validation), and "Sistema de Claves Únicas" (Unique Key System). It includes notes about robust numerical validation, date ranges, error handling, and unique key composition.

captura de pantalla de el entorno de trabajo compartido de vscode como ide para editar código y obsidian para la documentación

Historial de ramas y fusiones si es aplicable.

Solo se trabajó en una única rama principal

Evidencia por cada integrante del equipo.

1. Evidencia de Jhon Gustavo

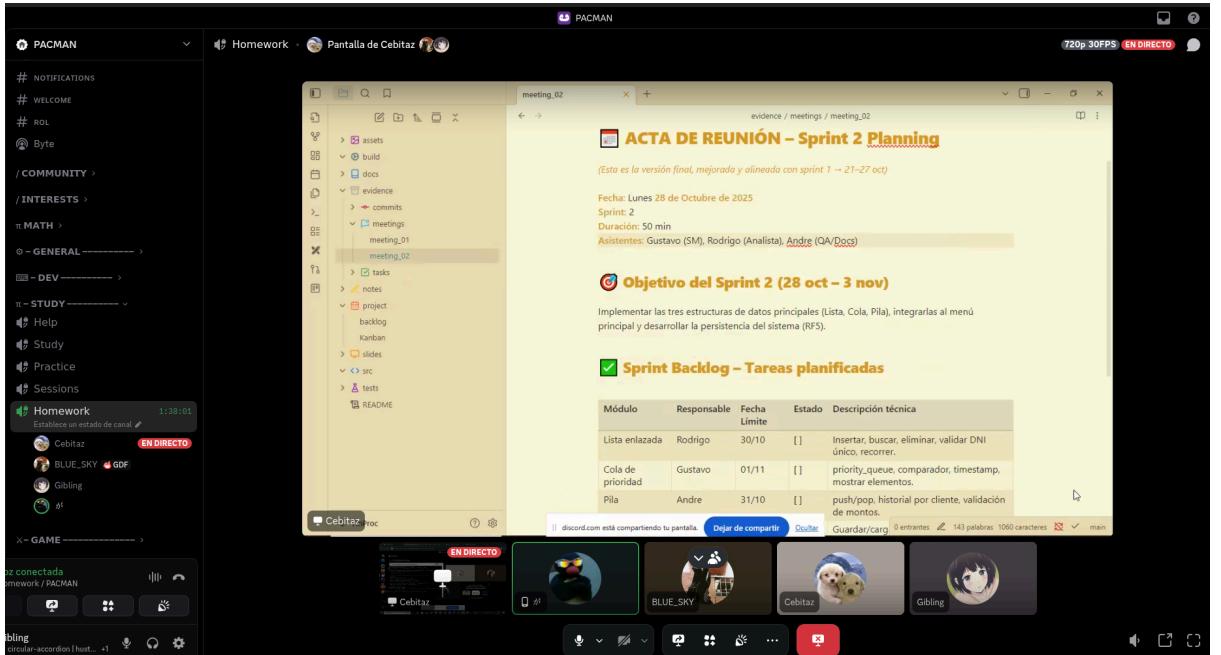
```

Thursday, 06 Nov 2025 06:47
hustavo@pirate:~/Github/hustavoJhon/FinProc 0/6
assets Dockerfile evidence LICENSE notes README.md
build docs include Makefile project slides tests
> ll
total 64K
drwxr-xr-x 5 hustavo hustavo 4.0K Oct 26 23:25 assets
drwxr-xr-x 2 hustavo hustavo 4.0K Oct 31 07:10 build
-rw-r--r-- 1 hustavo hustavo 195 Oct 31 07:14 Dockerfile
drwxr-xr-x 5 hustavo hustavo 4.0K Oct 26 23:25 docs
drwxr-xr-x 2 hustavo hustavo 4.0K Nov 5 20:33 evidence
drwxr-xr-x 2 hustavo hustavo 4.0K Nov 5 20:33 include
-rw-r--r-- 1 hustavo hustavo 12K Oct 26 21:28 LICENSE
-rw-r--r-- 1 hustavo hustavo 316 Oct 26 21:39 Makefile
drwxr-xr-x 2 hustavo hustavo 4.0K Oct 31 07:33 notes
drwxr-xr-x 2 hustavo hustavo 4.0K Oct 31 06:23 project
-rw-r--r-- 1 hustavo hustavo 2.0K Oct 31 07:44 README.md
drwxr-xr-x 2 hustavo hustavo 4.0K Nov 5 20:15 slides
drwxr-xr-x 2 hustavo hustavo 4.0K Nov 5 21:42 src
drwxr-xr-x 2 hustavo hustavo 4.0K Oct 31 07:38 tests

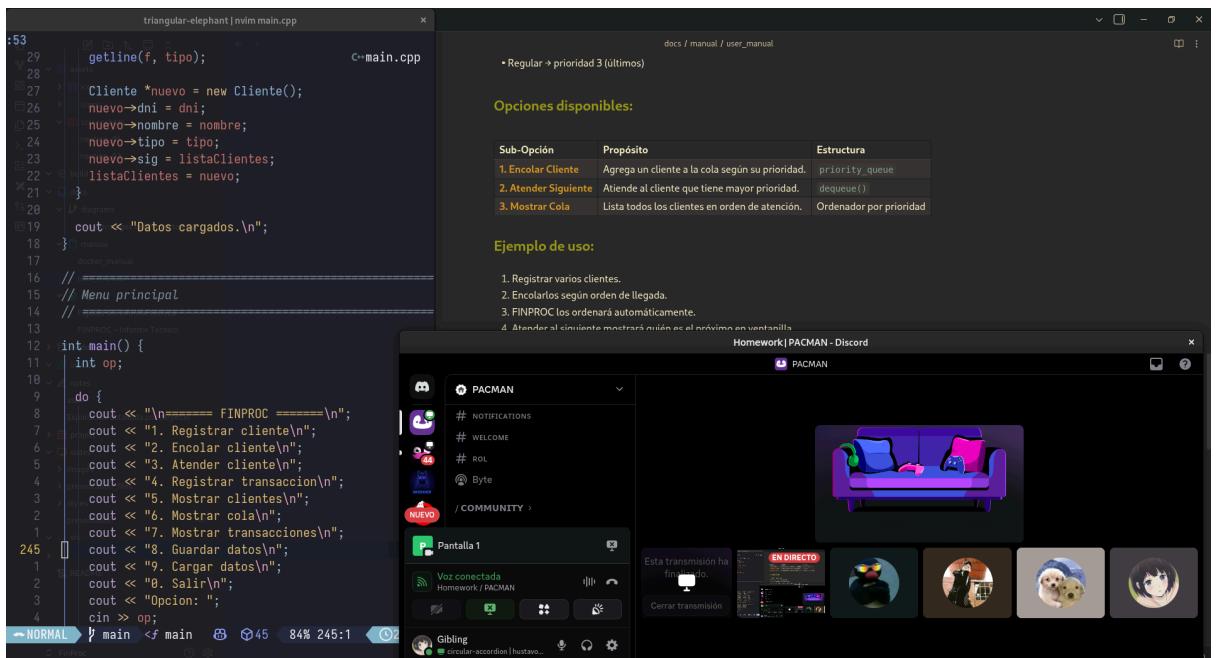
```

The terminal shows a directory listing of files and sub-directories within the `FinProc` repository. The files include `Dockerfile`, `Makefile`, and several `README` files.

2. Evidencia de Rodrigo



3. Evidencia de Sebastian



Enlace a la herramienta colaborativa

[GitHub](#) (repositorio):

[Obsidian](#): usado como gestor local de notas y documentación

[Discord](#): canal de comunicación para retroalimentación

[Git](#): herramienta principal de control de versiones utilizada en todo el flujo de trabajo