

**“Año de la recuperación y consolidación de la economía
peruana”**

ASIGNATURA:

Estructura de Datos
(Plan 2024)

NRC: 29901

NRO GRUPO: 4

PROYECTO GRUPAL:

Sistema de Gestión Genealógica con Árbol Binario

DOCENTE:

Rosario Delia Osorio Contreras

INTEGRANTES:

1. Mijael Joseph Bejarano Miche
2. Luis Enrique Cueva Barrera
3. Damián Javier Lopez Naula
4. Nelinho Capcha Hidalgo

Huancayo, 2025

ÍNDICE

Capítulo 1: Fase de Ideación	3
1.1. Descripción del Sistema.....	3
1.2. Requerimientos del Sistema	4
1.2.1. Requerimientos funcionales.....	4
1.2.2. Requerimientos no funcionales.....	5
1.4. Herramientas colaborativas.....	8
Capítulo 2: Prototipo	9
2.1. Descripción de estructuras de datos y operaciones	9
2.2. Algoritmos principales	11
2.3. Diagramas de Flujo.....	16
2.4. Avance del código fuente	19
Capítulo 3: Código Fuente	23
3.1. Código en C++	23
3.1.1. Definiciones preliminares.....	23
3.1.2. Funciones del sistema.....	24
3.1.3. Menú del sistema	28
3.2. Pruebas de validación	29
3.3. Manual de Usuario	31
Capítulo 4: Evidencias de Trabajo en Equipo	33
4.1. Control de Versiones	33
4.1.1. Registro de commits:	33
4.1.2. Evidencias adicionales	42
4.2. Plan de Trabajo y Roles Asignados	44
4.2.1. Descripción del rol semana 1:.....	44
4.2.2. Descripción de los roles semana 2:	45
4.2.3. Cronograma con fechas límite.....	47

Capítulo 1: Fase de Ideación

1.1. Descripción del Sistema

El **Sistema de Gestión Genealógica** es una aplicación desarrollada inicialmente en **PSelnt** y posteriormente implementada en **C++** usando **Dev-C++**, que simula un árbol genealógico familiar. Cada miembro puede tener un hijo izquierdo y un hijo derecho, siguiendo el modelo de un árbol binario no balanceado. El sistema contiene 3 funciones principales:

- **Insertar nuevos miembros:** El sistema permitirá la inserción dinámica de miembros familiares en un árbol binario, solicitando ID, nombre e ID del padre (si existe). Los hijos se incorporarán automáticamente, asignando el primero al subárbol izquierdo y el segundo al derecho. Para respetar las restricciones del árbol binario, se valida que cada nodo no tenga más de dos hijos, preservando la estructura jerárquica e integridad del árbol.
- **Mostrar tres recorridos diferentes:** Para procesar y mostrar un árbol binario, se emplean tres tipos de recorridos: **preorden**, **inorden** y **postorden**, cada uno con un propósito específico:
 - El recorrido **preorden** visita primero la raíz, seguido de los subárboles izquierdo y derecho; es útil cuando se necesita procesar la raíz antes que sus descendientes, como en un árbol genealógico donde se muestra primero al abuelo, luego hijos y nietos.
 - El recorrido **inorden** explora primero el subárbol izquierdo, luego la raíz y finalmente el derecho, ideal para una visualización ordenada, mostrando hijos, padres y luego nietos en secuencia.
 - El recorrido **postorden** recorre primero los hijos y luego la raíz, útil cuando se deben procesar descendientes antes del padre, como al analizar o eliminar miembros del árbol.
- **Eliminar un miembro:** La eliminación de miembros en este árbol genealógico solo es posible si no tienen hijos. Si el miembro a eliminar es la raíz sin descendencia, el árbol se vacía. En caso de tener hijos, no se permite su eliminación, ya que alteraría la estructura familiar. Por ejemplo, un abuelo con descendencia no puede eliminarse. Esta función mantiene la integridad del árbol.

1.2. Requerimientos del Sistema

1.2.1. Requerimientos funcionales

Los requerimientos funcionales del sistema tienen como objetivo garantizar el cumplimiento de las operaciones esenciales de un árbol genealógico digital, permitiendo representar, insertar, visualizar y eliminar miembros familiares de forma estructurada y lógica. Para ello, el sistema se apoya en una estructura de datos dinámica no lineal: el árbol binario, vista en la Unidad 4 del curso.

El sistema está compuesto por un solo módulo funcional, el Gestor del Árbol Genealógico, el cual debe implementar las siguientes funciones clave:

- **RF 1 [Inserción de miembros]:** Permite agregar un nuevo miembro al árbol con su ID, nombre y el ID del padre. El miembro se insertará automáticamente en el lado izquierdo si no hay otro hijo registrado; de lo contrario, se colocará en el lado derecho. Si el miembro es la raíz, se inserta directamente. Se valida que el padre exista en el árbol y que el lado correspondiente (izquierdo o derecho) esté disponible.
- **RF 2 [Estructura Binaria]:** Los miembros se organizan como nodos en un árbol binario, cada uno con un ID, nombre y punteros a sus hijos izquierdo y derecho. El nodo raíz es el primer miembro insertado, y los hijos se agregan automáticamente a la izquierda o derecha según disponibilidad. La estructura jerárquica se mantiene validando la existencia del padre y el lado disponible para insertar al nuevo miembro.
- **RF 3 [Recorrido Preorden]:** Muestra los miembros del árbol siguiendo el orden: primero la raíz, luego el hijo izquierdo y finalmente el hijo derecho. Esta lógica se implementa mediante la técnica de recursividad que imprime el nodo actual, luego llama a la función para recorrer el subárbol izquierdo y después el derecho. Esto se repite para cada nodo en el árbol hasta que todos los miembros sean mostrados.

- **RF 4 [Recorrido Inorden]:** Recorre primero el hijo izquierdo, luego la raíz y finalmente el hijo derecho. Se implementa mediante una función recursiva que primero llama a la función de recorrido para el subárbol izquierdo, luego imprime el nodo actual, y finalmente recorre el subárbol derecho.
- **RF 5 [Recorrido Postorden]:** Recorre primero ambos hijos y luego la raíz. La función recursiva primero llama a la función para recorrer el hijo izquierdo, luego el hijo derecho y, finalmente, imprime el nodo actual después de haber recorrido ambos subárboles.
- **RF 6 [Eliminación de miembros]:** Solo se pueden eliminar nodos que no tengan hijos (hojas). Si el nodo tiene descendientes, la operación es rechazada. En el código, se verifica si el nodo tiene hijos antes de eliminarlo, y si tiene descendientes, se muestra un mensaje de error.

1.2.2. Requerimientos no funcionales

Los requerimientos no funcionales del sistema deben asegurar una buena experiencia de usuario y detalles de calidad. A continuación, se detallan aspectos que el sistema debe cumplir:

- **RNF 1- [Interfaz sencilla]:** Los menús deben ser intuitivos y accesibles para usuarios sin experiencia técnica, permitiendo insertar, mostrar y eliminar miembros, y presentando información clara con validaciones para evitar errores.
- **RNF 2- [Validación de entradas]:** El sistema debe manejar eficientemente errores como IDs duplicados, referencias inválidas al buscar padres, exceso de hijos por nodo y la eliminación de nodos con descendencia, asegurando entradas válidas antes de ejecutar operaciones.
- **RNF 3 [Buen rendimiento]:** El sistema debe ejecutar todas las operaciones (inserción, eliminación y recorridos) en menos de 10 segundos, incluso con hasta 50 miembros en el árbol. Esto incluye la eficiencia en la búsqueda de nodos y la gestión de los hijos de cada nodo, garantizando un funcionamiento rápido.

- **RNF 4 [Código organizado]:** El código debe ser claro, bien comentado y modular, siguiendo buenas prácticas de programación para facilitar la comprensión y el mantenimiento. Cada función debe ser autónoma, con nombres descriptivos y comentarios que expliquen la lógica de inserción, eliminación y recorrido de los nodos.

1.3. Respuesta a las preguntas guías

Se plantearon preguntas clave que permiten comprender mejor el funcionamiento del sistema del árbol binario, proporcionando una visión más clara sobre la estructura y lógica del código a desarrollar para el reto del Árbol Genealógico.

a. ¿Qué información se debe almacenar en cada nodo del árbol?

El árbol genealógico se construye a partir de nodos, donde cada nodo **representa a un miembro de la familia**. Estos nodos están conectados entre sí mediante punteros que simulan las relaciones familiares directas.

```
struct Nodo {  
    int id;  
    string nombre;  
    Nodo* izquierdo;  
    Nodo* derecho;  
  
    Nodo(int i, string nom) {  
        id = i;  
        nombre = nom;  
        izquierdo = NULL;  
        derecho = NULL;  
    }  
};
```

Cada nodo tiene:

- **ID:** Un identificador único para distinguir a cada miembro del árbol.
- **Nombre:** El nombre del miembro, facilitando la identificación humana.
- **Puntero izquierdo:** Apunta al hijo ubicado en el lado izquierdo del nodo padre.
- **Puntero derecho:** Apunta al hijo ubicado en el lado derecho del nodo padre.

Esto permite representar las relaciones jerárquicas dentro del árbol, asegurando que cada miembro esté correctamente vinculado a sus descendientes.

b. ¿Cómo insertar y eliminar miembros del árbol sin romper su estructura?

Para insertar un miembro, primero se verifica si el ID ya existe. Si el miembro es la raíz (por ejemplo, el primer abuelo), se inserta directamente. Si no, se busca al padre del nuevo miembro, y si el lado izquierdo está libre, se coloca allí (por ejemplo, el primer hijo del abuelo). Si el lado izquierdo ya tiene un hijo, se coloca en el lado derecho (por ejemplo, el segundo hijo del abuelo).

Para eliminar un miembro, solo se puede eliminar si no tiene hijos. Si un hijo tiene descendencia (por ejemplo, un nieto), no se puede eliminar para no romper la estructura familiar. Por ejemplo, si intentamos eliminar a un hijo que tiene dos nietos, el sistema no permitirá la eliminación.

c. ¿Qué métodos permiten recorrer el árbol para visualizar la genealogía?

Para visualizar la genealogía, se utilizan tres métodos de recorrido:

- **Preorden:** Muestra primero la raíz, luego el hijo izquierdo y después el derecho. Este recorrido se implementará en la función *mostrarPreorden*.
- **Inorden:** Recorre primero el hijo izquierdo, luego la raíz y finalmente el hijo derecho, lo que proporciona un orden más jerárquico. Se implementará en la función *mostrarInorden*.
- **Postorden:** Recorre primero ambos hijos y luego la raíz, lo cual es útil para mostrar la descendencia antes que la ascendencia. Este recorrido estará en la función *mostrarPostorden*.

d. ¿Cómo determinar si un miembro pertenece a una rama específica?

Para saber si un miembro pertenece a una rama, **se busca al padre del miembro en el árbol**. Si encontramos al abuelo (nodo raíz), verificamos si el hijo que estamos buscando está en el lado izquierdo o derecho del abuelo. Por ejemplo, si el abuelo tiene dos hijos, uno a la izquierda (Hijo1) y otro a la derecha (Hijo2), al buscar a "Hijo1", sabremos que pertenece a la rama izquierda.

e. ¿Cómo balancear el árbol si se vuelve demasiado profundo?

El código no tiene balanceo automático, pero si el árbol se vuelve muy profundo, como un árbol con muchas ramas de un solo lado, **podemos reorganizarlo**. Por ejemplo, si tenemos un árbol que solo crece en el lado izquierdo (como un árbol inclinado), sería necesario reestructurarlo para equilibrar ambos lados. Esto podría resolverse con una técnica de balanceo, evitando que el árbol se alargue de un solo lado y afecte la eficiencia de operaciones como insertar o buscar miembros.

1.4. Herramientas colaborativas

En el desarrollo de nuestro proyecto final, hemos recurrido a diversas herramientas colaborativas que nos han permitido trabajar de manera sincronizada, optimizando tanto la codificación como la elaboración de presentaciones e informes. A continuación, detallo las funciones e importancias de las herramientas que hemos utilizado:

- **GitHub:**

La función de control de versiones en GitHub, a través de los *commits*, fue fundamental para realizar un seguimiento detallado de los cambios realizados en el proyecto. Cada *commit* permitió guardar un registro claro y específico de las modificaciones, lo que facilitó tanto la revisión de avances como la identificación de errores. Esta práctica también nos permitió revertir a versiones anteriores del código si era necesario, asegurando que cualquier cambio no deseado pudiera ser corregido de forma rápida y eficiente.

🔗 Link: https://github.com/Skadex2/E.D.D.-ABR_.git

- **Canva:**

Para diseñar y crear nuestras presentaciones de manera rápida y visualmente atractiva. Esta herramienta facilitó la creación de diapositivas con diseños profesionales, permitiendo que todos los miembros colaboraran en tiempo real para aportar ideas visuales y contenido gráfico.

🔗 Link: <https://acortar.link/tL4M5Q>

- **Google Meet:**

Las reuniones a través de Google Meet fueron esenciales para coordinar y mantener una comunicación constante entre los miembros del equipo. Nos permitió realizar llamadas de manera eficiente, compartir pantallas y discutir el progreso del proyecto en tiempo real.

- **Microsoft 365-Word (en línea):**

Para la elaboración del informe final, utilizamos Word como nuestra herramienta de trabajo colaborativo. Nos permitió escribir, editar y comentar el documento en tiempo real, lo que facilitó la integración de ideas de todos los miembros del equipo.

 Link: <https://acortar.link/N1Xf9z>

Capítulo 2: Prototipo

2.1. Descripción de estructuras de datos y operaciones

Para el prototipo del sistema de gestión genealógica con árboles binarios, se creó un pseudocódigo en PSeInt que utiliza diversas variables y estructuras de datos, principalmente arreglos, junto con la inicialización de valores, para dar forma al árbol genealógico.

A. Arreglos:

Los arreglos en PSEINT se utilizan para almacenar la información clave de los miembros del árbol genealógico y simular cómo se organizarían los nodos en un árbol binario de búsqueda. Si bien la solución final se implementará en un lenguaje como C++, donde la representación de un árbol binario se realiza mediante nodos dinámicos, en PSeInt no se dispone de tales estructuras. Por lo tanto, se opta por el uso de arreglos para emular esta funcionalidad,

- **padre_id [100]:** Este arreglo almacena el ID único de cada miembro del árbol. En un nodo típico de C++, cada nodo tendría un ID que lo identifica, y este array simula esa función. Cada elemento en el array corresponde a un miembro, y el ID se utiliza para acceder rápidamente a cada miembro durante las operaciones de inserción, eliminación o recorrido.

- **padre_nombre [100]:** Este arreglo almacena los nombres de los miembros del árbol genealógico. De manera similar a como un nodo de C++ tendría un campo para almacenar el nombre del miembro, este array permite asociar un nombre a cada ID en el árbol. Es la información descriptiva de cada miembro del árbol, y se utiliza durante los recorridos del árbol (*preorden*, *inorden*, *postorden*) para mostrar el nombre de cada miembro, lo que facilita la comprensión del árbol.
- **padre_izq[100] y padre_der[100]:** Estos arreglos permiten **simular la estructura jerárquica** del árbol, donde cada miembro puede tener dos hijos (izquierdo y derecho). En un diseño basado en nodos de C++, cada nodo tendría punteros que apuntan a sus hijos izquierdo y derecho, pero como en PSEINT no existen punteros, se usa un índice en el arreglo para simular la relación entre los miembros.
- **pila[100], pila1[100], pila2[100]:** Estos arreglos se usan para simular pilas en los recorridos del árbol (*preorden*, *inorden* y *postorden*). Dado que PSEINT no permite estructuras dinámicas ni recursión eficiente, se recurre a este método para imitar el comportamiento de una pila de manera manual, usando arreglos y una variable tope.

B. Variables auxiliares

Estas variables son esenciales para gestionar y facilitar el acceso a los miembros del árbol en cada una de las acciones.

- **tope, actual, indiceActual:**
Estas variables gestionan el estado de las pilas durante los recorridos. La variable **tope** mantiene la posición actual en la pila, mientras que **actual** y **indiceActual** representan los índices de los miembros que deben ser procesados en cada operación de recorrido.

- **totalMiembros:**

Mantiene el **número total de miembros** en el árbol. Este contador es clave para controlar cuántos miembros se han insertado y evitar sobrepasar el límite de 100 miembros.

- **idNuevo, idPadre, idEliminar:**

Son los **IDs** del nuevo miembro a insertar, el padre del miembro que se está insertando, y el miembro que se desea eliminar, respectivamente. Estas variables facilitan la manipulación y gestión de los nodos en el árbol.

- **lado:**

Se utiliza para determinar si el nuevo miembro será colocado en el **lado izquierdo** o **lado derecho** del padre. Esto es necesario para mantener la estructura binaria del árbol, donde cada nodo puede tener hasta dos hijos, uno en cada lado.

2.2. Algoritmos principales

a. [Opción 1]: Inserta miembro

Para insertar un nuevo miembro en el árbol genealógico, el programa solicita al usuario el ID, nombre del nuevo miembro y el ID de su padre. Si **idPadre** es 0, el nuevo miembro se guarda como raíz. Antes de agregarlo, se verifica que **totalMiembros** no haya alcanzado el límite de 100.

Si hay espacio, se incrementa **totalMiembros** y se registran los datos en los arreglos **padre_id**, **padre_nombre**, **padre_izq** y **padre_der**. Si el miembro no es raíz, se busca el **idPadre** en **padre_id** y se obtiene su posición en **indicePadre**.

Luego, se solicita el lado de inserción (**lado**). Si el lado está disponible (**padre_izq** o **padre_der** en 0), se vincula al nuevo miembro. Si no, o si hay un error en la entrada, se cancela la inserción restando **totalMiembros**.

```

1: // Insertar miembro
Escribir "Ingrese ID del nuevo miembro:"
Leer idNuevo
Escribir "Ingrese nombre del nuevo miembro:"
Leer nombreNuevo
Escribir "Ingrese ID del padre (0 si es raíz):"
Leer idPadre

// Validar límite
Si totalMiembros ≥ 100 Entonces
    Escribir "Límite máximo de miembros alcanzado."
Sino
    // Agregar nuevo miembro temporalmente al final
    totalMiembros ← totalMiembros + 1
    padre_id[totalMiembros] ← idNuevo
    padre_nombre[totalMiembros] ← nombreNuevo
    padre_izq[totalMiembros] ← 0
    padre_der[totalMiembros] ← 0

    Si idPadre = 0 Entonces
        Escribir "Miembro raíz insertado."
    Sino
        // Buscar índice del padre antes de usarlo
        indicePadre ← 0
        Para i ← 1 Hasta totalMiembros - 1
            Si padre_id[i] = idPadre Entonces
                indicePadre ← i
            // Opcional, si quieres salir al encontrar padre
        FinSi
    FinPara
FinSi

```

```

Sino
    Escribir "Ingrese lado del hijo (I = Izquierdo, D = Derecho):"
    Leer lado

    Si lado = "I" Entonces
        Si padre_izq[indicePadre] = 0 Entonces
            padre_izq[indicePadre] ← totalMiembros
            Escribir "Miembro insertado como hijo izquierdo."
        Sino
            Escribir "El padre ya tiene hijo izquierdo."
            totalMiembros ← totalMiembros - 1
        FinSi
    Sino
        Si lado = "D" Entonces
            Si padre_der[indicePadre] = 0 Entonces
                padre_der[indicePadre] ← totalMiembros
                Escribir "Miembro insertado como hijo derecho."
            Sino
                Escribir "El padre ya tiene hijo derecho."
                totalMiembros ← totalMiembros - 1
            FinSi
        Sino
            Escribir "Opción de lado inválida."
            totalMiembros ← totalMiembros - 1
        FinSi
    FinSi
FinSi

```

b. [Opción 2]: Mostrar Preorden

Este algoritmo realiza el recorrido en *preorden* del árbol genealógico de forma iterativa. Primero, verifica si **totalMiembros** es 0; si lo es, el árbol está vacío. Si no, se inicializa la variable **tope** en 0 y coloca el índice de la raíz (**indiceActual**, que inicia en 1) en la pila simulada (**pila[tope]**).

Luego, se entra en un ciclo Mientras que continúa mientras haya elementos en la pila. En cada iteración, se toma el índice del nodo actual, se muestra su ID y nombre, y si tiene hijo derecho (**padre_der**) o izquierdo (**padre_izq**), se agregan a la pila. Primero se agrega el hijo derecho para que el izquierdo se procese antes, respetando el orden *preorden*: nodo, hijo izquierdo, hijo derecho.

```

2:
Si totalMiembros = 0 Entonces
    Escribir "Árbol vacío."
Sino
    tope ← 0
    indiceActual ← 1

    tope ← tope + 1
    pila[tope] ← indiceActual

    Mientras tope > 0 Hacer
        indiceActual ← pila[tope]
        tope ← tope - 1
        Escribir padre_id[indiceActual], " - ", padre_nombre[indiceActual]

        Si padre_der[indiceActual] ≠ 0 Entonces
            tope ← tope + 1
            pila[tope] ← padre_der[indiceActual]
        FinSi

        Si padre_izq[indiceActual] ≠ 0 Entonces
            tope ← tope + 1
            pila[tope] ← padre_izq[indiceActual]
        FinSi
    FinMientras
FinSi

```

c. [Opción 3]: Mostrar Inorden

Para el recorrido *inorden* del árbol genealógico, se aplica la lógica izquierda, raíz, derecho de forma iterativa. Primero, si **totalMiembros** es 0, se indica que el árbol está vacío. En caso contrario, se inicia desde la raíz (**actual**, que inicia en 1) usando la pila simulada (**pila[tope]**).

El recorrido avanza lo más posible hacia la izquierda (**padre_izq**), apilando cada nodo en **pila[tope]**. Al llegar al **extremo izquierdo**, se toma el nodo en la cima de la pila, se imprime su ID y nombre, y luego se pasa al subárbol derecho (**padre_der**). Este proceso se repite hasta recorrer todo el árbol.

```

3: // Mostrar Inorden
Si totalMiembros = 0 Entonces
    Escribir "Árbol vacío."
Sino
    tope ← 0
    actual ← 1

    Mientras tope > 0 O actual ≠ 0 Hacer
        Mientras actual ≠ 0 Hacer
            tope ← tope + 1
            pila[tope] ← actual
            actual ← padre_izq[actual]
        FinMientras

        actual ← pila[tope]
        tope ← tope - 1

        Escribir padre_id[actual], " - ", padre_nombre[actual]

        actual ← padre_der[actual]
    FinMientras
FinSi

```

d. (Opción 4): Mostrar Post-orden

Este algoritmo implementa el recorrido *postorden* del árbol genealógico (izquierdo, derecho, raíz) de forma iterativa. Si **totalMiembros** es 0, el árbol está vacío. En caso contrario, se utilizan dos arreglos tipo pila: **pila1** para recorrer y **pila2** para almacenar el orden final. Ambas pilas usan sus respectivas variables **tope1** y **tope2**.

Se inicia la **pila1** en 1. Luego, en un bucle, se extrae cada nodo, se guarda en **pila2** y se agregan sus hijos izquierdos y derecho a **pila1**. Esto invierte el orden natural. Al finalizar, **pila2** contiene los nodos en orden *postorden*. Se recorre **pila2** de atrás hacia adelante, mostrando de cada miembro del árbol.

```
4: // Mostrar Postorden
Si totalMiembros = 0 Entonces
    Escribir "Árbol vacío."
Sino
    tope1 ← 0
    tope2 ← 0

    tope1 ← tope1 + 1
    pila1[tope1] ← 1

    Mientras tope1 > 0 Hacer
        nodo ← pila1[tope1]
        tope1 ← tope1 - 1
        tope2 ← tope2 + 1
        pila2[tope2] ← nodo

        Si padre_izq[nodo] ≠ 0 Entonces
            tope1 ← tope1 + 1
            pila1[tope1] ← padre_izq[nodo]
        FinSi
        Si padre_der[nodo] ≠ 0 Entonces
            tope1 ← tope1 + 1
            pila1[tope1] ← padre_der[nodo]
        FinSi
    FinMientras

    Mientras tope2 > 0 Hacer
        nodo ← pila2[tope2]
        tope2 ← tope2 - 1
        Escribir padre_id[nodo], " - ", padre_nombre[nodo]
    FinMientras
FinSi
```

e. [Opción 5]: Eliminar miembro

Este algoritmo permite eliminar un miembro del árbol. Primero se verifica si **totalMiembros** es 0; si es así, se informa que no hay miembros para eliminar. Si hay datos, se solicita el id del miembro a eliminar y se busca su posición en el arreglo **padre_id**, guardando el resultado en **indiceEliminar**.

Si no se encuentra el ID, se muestra un mensaje. Si se encuentra, pero el miembro tiene hijos, no se permite eliminarlo. En caso de que no tenga hijos, se recorre el árbol para eliminar cualquier vínculo que lo conecte con su padre (asignando 0 en **padre_izq[i]** o **padre_der[i]**).

Si el miembro a eliminar no es el último, se reemplaza su información con la del último miembro (**totalMiembros**) y se actualizan las referencias en el árbol que apuntaban a ese último nodo, cambiándolas al nuevo índice (**indiceEliminar**). Finalmente, se reduce **totalMiembros** y se confirma la eliminación.

```
5: // Eliminar miembro
Si totalMiembros = 0 Entonces
    Escribir "No hay miembros para eliminar."
Sino
    Escribir "Ingrese ID del miembro a eliminar:"
    Leer idEliminar

    Definir indiceEliminar Como Entero
    indiceEliminar ← 0

    Para i ← 1 Hasta totalMiembros
        Si padre_id[i] = idEliminar Entonces
            indiceEliminar ← i
        FinSi
    FinPara

    Si indiceEliminar = 0 Entonces
        Escribir "Miembro no encontrado."
    Sino
        Si padre_izq[indiceEliminar] ≠ 0 O padre_der[indiceEliminar] ≠ 0 Entonces
            Escribir "No se puede eliminar un miembro con hijos."
        Sino
            // Buscar padre para desconectar al hijo eliminado
            Para i ← 1 Hasta totalMiembros
                Si padre_izq[i] = indiceEliminar Entonces
                    padre_izq[i] ← 0
                FinSi
                Si padre_der[i] = indiceEliminar Entonces
                    padre_der[i] ← 0
                FinSi
            FinPara
            FinSi
        FinPara
    FinSi
```

```
// Reemplazar el nodo eliminado con el último nodo si no es el mismo
Si indiceEliminar ≠ totalMiembros Entonces
    padre_id[indiceEliminar] ← padre_id[totalMiembros]
    padre_nombre[indiceEliminar] ← padre_nombre[totalMiembros]
    padre_izq[indiceEliminar] ← padre_izq[totalMiembros]
    padre_der[indiceEliminar] ← padre_der[totalMiembros]

    // Actualizar referencias que apuntaban al último nodo
    Para i ← 1 Hasta totalMiembros - 1
        Si padre_izq[i] = totalMiembros Entonces
            padre_izq[i] ← indiceEliminar
        FinSi
        Si padre_der[i] = totalMiembros Entonces
            padre_der[i] ← indiceEliminar
        FinSi
    FinPara
    FinSi

    totalMiembros ← totalMiembros - 1
    Escribir "Miembro eliminado correctamente."
FinSi
FinSi
```

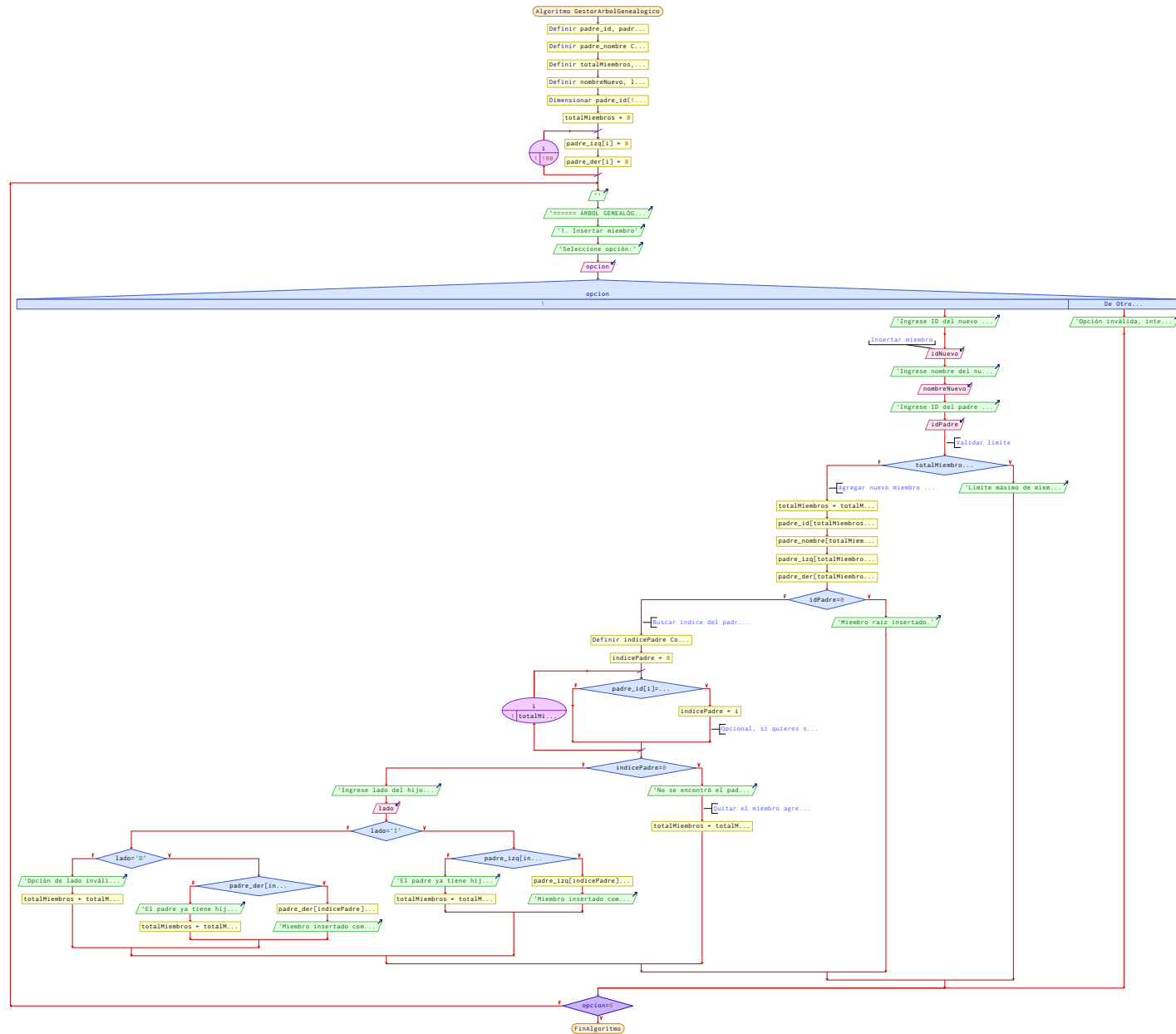
f. (Opción 6): Salir

Esta opción termina la ejecución del programa. Cuando el usuario selecciona "Salir", el algoritmo imprime un mensaje de despedida y finaliza la operación. La técnica de salida es importante porque permite al usuario cerrar el programa de manera controlada, sin que el sistema se cierre abruptamente.

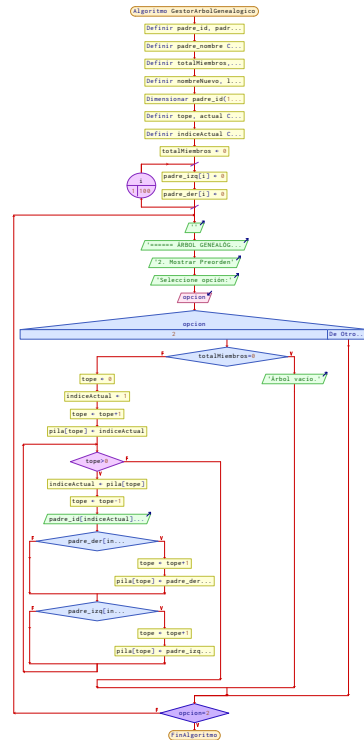
```
6:
    Escribir "Saliendo del programa."
```

2.3. Diagramas de Flujo

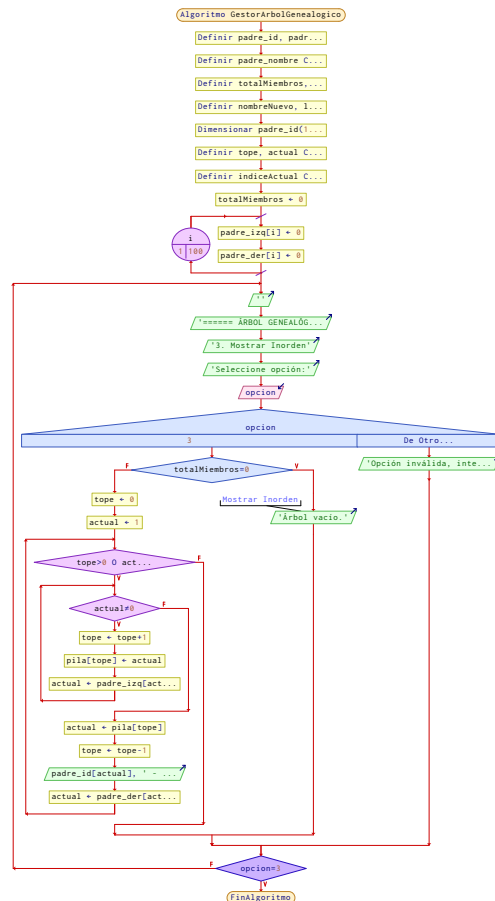
A. [Opción 1]-Insertar miembro:



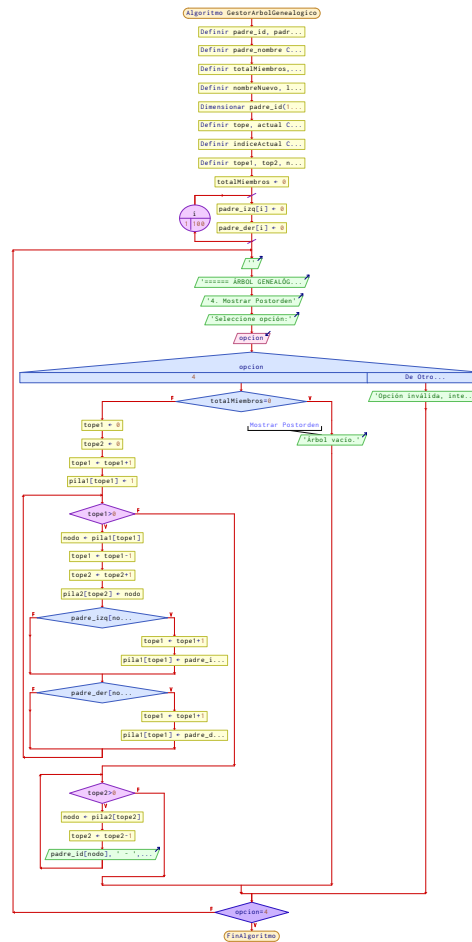
B. [Opción 2]-Mostrar Preorden:



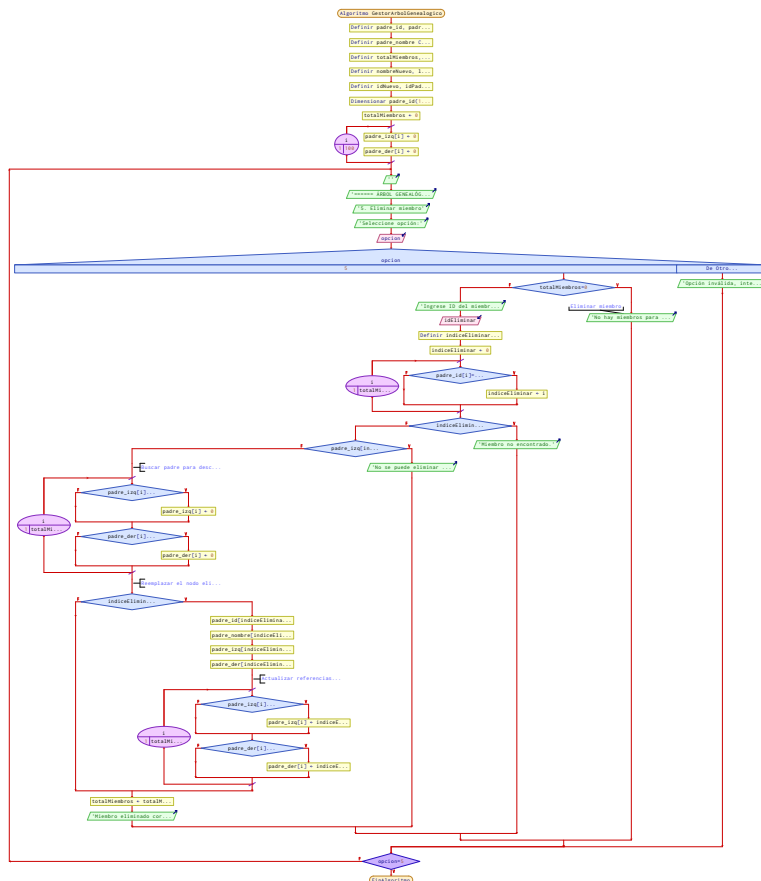
C. [Opción 3]-Mostrar Inorden:



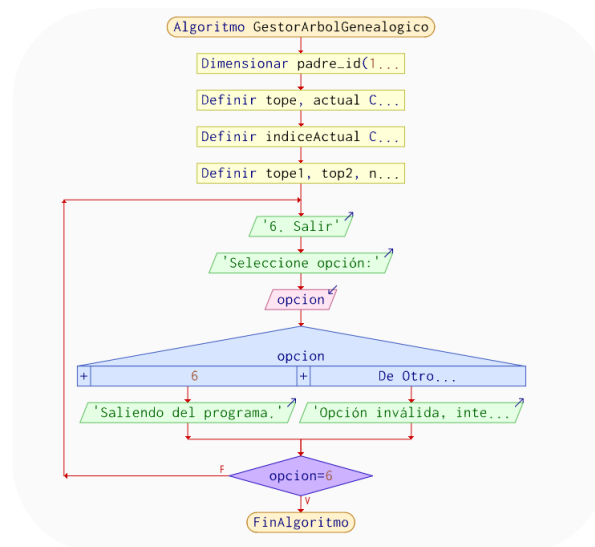
D. [Opción 4]-Mostrar Postorden:



E. [Opción 5]-Eliminar miembro:



F. [Opción 6]-Salir:



2.4. Avance del código fuente

A partir de este prototipo en forma de pseudocódigo, se realizó el primer avance del código fuente en C++. En esta implementación se aplicó la misma lógica desarrollada durante el prototipo, pero utilizando ahora nodos y punteros, elementos que no eran posibles de manejar directamente en PSeInt.

- **Variables y Estructuras de Datos:**

En esta parte del código se definen las estructuras base del gestor. Primero, se crea una estructura llamada **Nodo**, que representa a cada persona del árbol genealógico. Cada nodo guarda su ID, su nombre y tiene dos campos (izquierdo y derecho) que sirven para conectar con sus hijos. Al crear un nuevo nodo, esos campos empiezan vacíos (en **NULL**), porque todavía no tiene hijos. También se declara una variable global llamada raíz, que es donde empieza el árbol, o sea, el primer miembro que se agrega.

```

struct Nodo {
    int id;
    string nombre;
    Nodo* izquierdo;
    Nodo* derecho;

    Nodo(int i, string nom) {
        id = i;
        nombre = nom;
        izquierdo = NULL;
        derecho = NULL;
    }
};

Nodo* raiz = NULL;

```

Además, se crea la función **buscarPadre**, que permite buscar y verificar si un nodo con un ID específico ya se encuentra en el árbol genealógico. Esta función es muy útil e importante para evitar duplicados y asegurarse de que el nodo exista antes de realizar una inserción.

```

// Buscar nodo por ID (puntero)
Nodo* buscarPadre(Nodo* nodo, int id) {
    if (nodo == NULL) { return NULL; }
    if (nodo->id == id) { return nodo; }

    Nodo* izq = buscarPadre(nodo->izquierdo, id);
    if (izq != NULL) { return izq; }

    return buscarPadre(nodo->derecho, id);
}

```

- **Función Insertar Miembros:**

La función **insertarMiembro** permite agregar un nuevo nodo al árbol genealógico, ya sea como raíz o como hijo de otro nodo, según el ID del padre y el lado indicado. Para usar esta función, se creó **menuInsertarMiembro**, que sirve como una pequeña interfaz donde el usuario ingresa los datos del nuevo miembro antes de hacer la inserción.

```
// Insertar nodo
void insertarMiembro(int id, string nombre, int idPadre, char lado) {
    Nodo* nuevo = new Nodo(id, nombre);

    if (idPadre == 0) {
        if (raiz == NULL) {
            raiz = nuevo;
            cout << "Raiz insertada.\n";
        } else {
            cout << "Ya existe una raiz.\n";
            delete nuevo;
        }
        return;
    }

    Nodo* padre = buscarPadre(raiz, idPadre);
    if (padre == NULL) {
        cout << "Padre no encontrado.\n";
        delete nuevo;
        return;
    }

    if ((lado == 'I' || lado == 'i') && padre->izquierdo == NULL) {
        padre->izquierdo = nuevo;
        cout << "Insertado como hijo izquierdo.\n";
    } else if ((lado == 'D' || lado == 'd') && padre->derecho == NULL) {
        padre->derecho = nuevo;
        cout << "Insertado como hijo derecho.\n";
    } else {
        cout << "Ya hay un hijo en ese lado o el lado es invalido.\n";
        delete nuevo;
    }
}
```

```
// Función Menu
void menuInsertarMiembro() {
    int id, idPadre;
    string nombre;
    char lado;

    cout << "Ingrese ID del nuevo miembro: ";
    cin >> id;
    cout << "Ingrese nombre: ";
    cin >> nombre;
    cout << "Ingrese ID del padre (0 si es raiz): ";
    cin >> idPadre;

    if (idPadre != 0) {
        cout << "¿Lado del hijo? (I = Izquierdo, D = Derecho): ";
        cin >> lado;
    } else {
        lado = 'R'; // raiz
    }

    insertarMiembro(id, nombre, idPadre, lado);
}
```

- **Función Eliminar Miembros:**

La función **eliminarMiembro** permite borrar un nodo del árbol, siempre que no tenga hijos. Recorre el árbol buscando el ID indicado, y si encuentra un nodo sin descendencia, lo elimina. Para utilizar esta lógica se creó **menuEliminarMiembro**, que funciona como interfaz: solicita el ID al usuario y gestiona el caso especial si se intenta eliminar la raíz.

```
// Eliminar miembro si no tiene hijos
bool eliminarMiembro(Nodo* nodo, int id) {
    if (nodo == NULL) { return false; }

    if (nodo->izquierdo != NULL && nodo->izquierdo->id == id) {
        if (nodo->izquierdo->izquierdo == NULL && nodo->izquierdo->derecho == NULL) {
            delete nodo->izquierdo;
            nodo->izquierdo = NULL;
            return true;
        } else {
            cout << "No se puede eliminar:  tiene hijos.\n";
            return false;
        }
    }

    if (nodo->derecho != NULL && nodo->derecho->id == id) {
        if (nodo->derecho->izquierdo == NULL && nodo->derecho->derecho == NULL) {
            delete nodo->derecho;
            nodo->derecho = NULL;
            return true;
        } else {
            cout << "No se puede eliminar:  tiene hijos.\n";
            return false;
        }
    }

    return eliminarMiembro(nodo->izquierdo, id) || eliminarMiembro(nodo->derecho, id);
}
```

```
void menuEliminarMiembro() {
    if (raiz == NULL) {
        cout << "Arbol vacio.\n";
        return;
    }

    int idEliminar;
    cout << "Ingrese ID a eliminar: ";
    cin >> idEliminar;

    if (raiz->id == idEliminar) {
        if (raiz->izquierdo == NULL && raiz->derecho == NULL) {
            delete raiz;
            raiz = NULL;
            cout << "Raiz eliminada.\n";
        } else {
            cout << "No se puede eliminar la raiz con hijos.\n";
        }
    } else {
        if (eliminarMiembro(raiz, idEliminar)) {
            cout << "Miembro eliminado.\n";
        } else {
            cout << "No se encontro el miembro o no se puede eliminar.\n";
        }
    }
}
```

- **Recorridos del árbol binario:**

Los tres recorridos (**mostrarPreorden**, **mostrarInorden** y **mostrarPostorden**) permiten mostrar los nodos del árbol genealógico en distintos órdenes:

- **Preorden:** muestra primero el nodo actual, luego recorre el hijo izquierdo y después el derecho. Es útil para ver la jerarquía desde la raíz hacia abajo.
- **Inorden:** recorre primero el hijo izquierdo, luego muestra el nodo actual y después el derecho. Se usa comúnmente para obtener un orden lógico de los datos.
- **Postorden:** recorre primero los hijos (izquierdo y derecho) y al final muestra el nodo actual. Sirve para procesar los nodos después de visitar sus descendientes.

Todas usan recursión y muestran el id y nombre de cada miembro.

```
// Mostrar recorridos
void mostrarPreorden(Nodo* nodo) {
    if (nodo == NULL) { return; }
    cout << nodo->id << " - " << nodo->nombre << endl;
    mostrarPreorden(nodo->izquierdo);
    mostrarPreorden(nodo->derecho);
}

void mostrarInorden(Nodo* nodo) {
    if (nodo == NULL) { return; }
    mostrarInorden(nodo->izquierdo);
    cout << nodo->id << " - " << nodo->nombre << endl;
    mostrarInorden(nodo->derecho);
}

void mostrarPostorden(Nodo* nodo) {
    if (nodo == NULL) { return; }
    mostrarPostorden(nodo->izquierdo);
    mostrarPostorden(nodo->derecho);
    cout << nodo->id << " - " << nodo->nombre << endl;
}
```

Capítulo 3: Código Fuente

3.1. Código en C++

3.1.1. Definiciones preliminares

A. Inclusión de Bibliotecas

El inicio de nuestro código en C++ incluye 2 bibliotecas esenciales: **<iostream>** para manejar la entrada y salida estándar mediante objetos como *cin* y *cout* y **<string>** para emplear la clase *string*, necesario para almacenar y mostrar nombres de las personas en árbol genealógico. Además, la línea **using namespace std;** nos permite escribir el código de forma más corta, ya que no es necesario poner *std::* delante de palabras como *cout* o *string*.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
```

B. Estructuras y Variables Globales

- **Estructura Nodo (Árbol Binario):** La estructura *Nodo* representa a cada miembro del árbol genealógico como un nodo de un árbol binario. Contiene un identificador único (**id**), el nombre del miembro (**nombre**) y dos punteros (**izquierdo** y **derecho**) que señalan a sus posibles hijos. Estos punteros se inicializan en **NULL**. Además, incluye un constructor que asigna el id y el nombre, dejando los hijos como nulos. Esta estructura permite modelar relaciones familiares de forma jerárquica y dinámica.

```
struct Nodo {
    int id;
    string nombre;
    Nodo* izquierdo = NULL;
    Nodo* derecho = NULL;
    Nodo(int i, string nom) : id(i), nombre(nom) {}
};
```

- **Variables globales:** Se usa una variable llamada raíz de tipo nodo que representa el nodo raíz del árbol genealógico, esta variable es el punto de acceso principal a recorrer o modificar la estructura del árbol.

```
19
20  Nodo* raiz = NULL;
21
```

C. Funciones auxiliares

- **Validación de existencia de un nodo:** La función **existeID** verifica de forma recursiva si un identificador (id) ya existe dentro del árbol genealógico. Recibe como parámetros un puntero al nodo actual y el ID a buscar. Si el nodo es nulo, devuelve false, lo que indica que no se encontró el ID en esa rama. En caso contrario, compara el ID del nodo actual y continúa la búsqueda en sus hijos izquierdo y derecho. Esta función es fundamental para evitar duplicación de miembros en el árbol.

```
bool existeID(Nodo* nodo, int id) {
    if (!nodo) return false;
    return nodo->id == id || existeID(nodo->izquierdo, id) || existeID(nodo->derecho, id);
}
```

3.1.2. Funciones del sistema

A. Insertar miembro (sistema):

Esta función corresponde a la opción 1 del menú principal y permite insertar un nuevo miembro en el árbol genealógico. Primero verifica que el ID no exista ya en el árbol para evitar duplicados. Si el **idPadre** es 0 y no hay una raíz definida, el nuevo nodo se inserta como raíz. En caso contrario, se busca el nodo padre recorriendo el árbol de forma iterativa. Si se encuentra, el nuevo miembro se inserta como hijo izquierdo o derecho, dependiendo de la disponibilidad. Si ambos lados del padre ya están ocupados, o si no se encuentra el padre, la inserción se cancela y el nuevo nodo es eliminado para mantener la integridad del árbol.


```

bool insertarNodo(int id, string nombre, int idPadre) {
    if (existeID(raiz, id)) {
        cout << "Error: ID ya existente.\n";
        return false;
    }

    Nodo* nuevo = new Nodo(id, nombre);

    if (idPadre == 0) {
        if (!raiz) {
            raiz = nuevo;
            cout << "Raiz insertada.\n";
            return true;
        } else {
            cout << "Ya existe una raiz.\n";
            delete nuevo;
            return false;
        }
    }

    // Búsqueda e inserción
    Nodo* stack[100];
    int top = -1;
    stack[++top] = raiz;
    while (top >= 0) {
        Nodo* actual = stack[top--];
        if (actual->id == idPadre) {
            if (!actual->izquierdo) {
                actual->izquierdo = nuevo;
                cout << "Insertado como hijo izquierdo.\n";
                return true;
            } else if (!actual->derecho) {
                actual->derecho = nuevo;
                cout << "Insertado como hijo derecho.\n";
                return true;
            } else {
                cout << "El padre ya tiene dos hijos.\n";
                delete nuevo;
                return false;
            }
        }
        if (actual->derecho) stack[++top] = actual->derecho;
        if (actual->izquierdo) stack[++top] = actual->izquierdo;
    }

    cout << "Padre no encontrado.\n";
    delete nuevo;
    return false;
}

```

B. Insertar miembro (Interfaz):

Esta función solicita al usuario los datos necesarios para agregar un nuevo miembro al árbol genealógico: el ID, el nombre y el ID del padre (ingresando 0 si se trata de la raíz). Una vez capturada la información, esta se envía a la función **insertarNodo**, que se encarga del proceso de validación e inserción dentro del árbol según las reglas establecidas.

```

void menuInsertar() {
    int id, idPadre;
    string nombre;
    cout << "ID: "; cin >> id;
    cout << "Nombre: "; cin >> nombre;
    cout << "ID del padre (0 si es raíz): "; cin >> idPadre;
    insertarNodo(id, nombre, idPadre);
}

```

C. Mostrar Preorden:

Esta función corresponde a la opción número 2 del menú principal y realiza un recorrido en *preorden* del árbol genealógico. Primero visita y muestra el nodo actual, luego recorre el hijo izquierdo y finalmente el hijo derecho. Este tipo de recorrido permite visualizar la jerarquía familiar desde la raíz hacia los descendientes, destacando primero a los ancestros.

```
void preorden(Nodo* nodo) {  
    if (!nodo) return;  
    cout << nodo->id << " - " << nodo->nombre << "\n";  
    preorden(nodo->izquierdo);  
    preorden(nodo->derecho);  
}
```

D. Mostrar Inorden:

Esta función corresponde al recorrido inorden y muestra primero el subárbol izquierdo, luego el nodo actual y finalmente el subárbol derecho. Es útil cuando se desea presentar el árbol genealógico en un orden más organizado, ya sea alfabético o numérico, especialmente si el orden está determinado por el ID de cada miembro.

```
void inorden(Nodo* nodo) {  
    if (!nodo) return;  
    inorden(nodo->izquierdo);  
    cout << nodo->id << " - " << nodo->nombre << "\n";  
    inorden(nodo->derecho);  
}
```

E. Mostrar Postorden:

Contiene la visualización en el orden Subárbol izquierdo - Subárbol derecho - Nodo actual. Este recorrido es ideal para operaciones que requieren procesar primero a los descendientes antes que, a los padres, como liberar memoria o realizar cálculos acumulativos desde las hojas hasta la raíz. En el contexto del árbol genealógico, es útil para examinar y analizar las generaciones más jóvenes antes de subir hacia los ancestros.

```
void postorden(Nodo* nodo) {  
    if (!nodo) return;  
    postorden(nodo->izquierdo);  
    postorden(nodo->derecho);  
    cout << nodo->id << " - " << nodo->nombre << "\n";  
}
```

F. Eliminar miembro (bool):

Esta función realiza la eliminación de un nodo hijo en el árbol genealógico buscando recursivamente el nodo con el ID especificado. Solo permite eliminar nodos que no tengan hijos, es decir, nodos hoja. Si encuentra un nodo hijo sin descendientes con el ID solicitado, lo elimina y devuelve true; de lo contrario, continúa la búsqueda o retorna false si no puede eliminarlo.

```
bool eliminarNodo(Nodo*& nodo, int id) {
    if (!nodo) return false;

    if (nodo->izquierdo && nodo->izquierdo->id == id) {
        if (!nodo->izquierdo->izquierdo && !nodo->izquierdo->derecho) {
            delete nodo->izquierdo;
            nodo->izquierdo = NULL;
            return true;
        }
    }
    if (nodo->derecho && nodo->derecho->id == id) {
        if (!nodo->derecho->izquierdo && !nodo->derecho->derecho) {
            delete nodo->derecho;
            nodo->derecho = NULL;
            return true;
        }
    }
    return eliminarNodo(nodo->izquierdo, id) || eliminarNodo(nodo->derecho, id);
}
```

G. Eliminar miembro (interfaz):

Esta función permite al usuario eliminar un miembro del árbol genealógico a partir de su ID. Primero verifica si el árbol está vacío, en cuyo caso muestra un mensaje y termina la operación. Si el árbol contiene nodos, solicita al usuario el ID del miembro a eliminar. En caso de que el ID corresponda a la raíz, solo podrá eliminarse si esta no tiene hijos; de lo contrario, se notifica al usuario que la operación no es posible. Para otros nodos, se llama a la función **eliminarNodo**, que intenta realizar la eliminación de acuerdo con las reglas definidas. Según el resultado, se informa si el miembro fue eliminado correctamente o si no pudo eliminarse, ya sea porque no existe o porque tiene descendencia.

```

void menuEliminarMiembro() {
    if (raiz == NULL) {
        cout << "Arbol vacio.\n";
        return;
    }

    int idEliminar;
    cout << "Ingrese ID a eliminar: ";
    cin >> idEliminar;

    if (raiz->id == idEliminar) {
        if (raiz->izquierdo == NULL && raiz->derecho == NULL) {
            delete raiz;
            raiz = NULL;
            cout << "Raiz eliminada.\n";
        } else {
            cout << "No se puede eliminar la raiz con hijos.\n";
        }
    } else {
        if (eliminarMiembro(raiz, idEliminar)) {
            cout << "Miembro eliminado.\n";
        } else {
            cout << "No se encontro el miembro o no se puede eliminar.\n";
        }
    }
}

```

3.1.3. Menú del sistema

La función *main* implementa el menú principal del sistema para gestionar los miembros del árbol genealógico, ofreciendo seis opciones al usuario. El programa ejecuta un ciclo *do-while* que mantiene activo el menú hasta que se seleccione la opción 6 para salir. En cada iteración, se muestra el menú con las opciones disponibles y se solicita al usuario ingresar su elección mediante la consola. La opción ingresada se evalúa con una estructura *switch* que llama a la función correspondiente según la selección. Si el usuario introduce un valor fuera del rango esperado, el sistema muestra un mensaje de error y vuelve a solicitar la opción.

```

switch (op) {
    case 1: menuInsertar(); break;
    case 2: preorden(raiz); break;
    case 3: inorden(raiz); break;
    case 4: postorden(raiz); break;
    case 5: menuEliminar(); break;
    case 6: cout << "Saliendo...\n"; break;
    default: cout << "Opción inválida.\n";
}

```

```

int main() {
    int opcion;

    do {
        // Menú de opciones
        cout << "\n===== ÁRBOL GENEALÓGICO =====\n";
        cout << "1. Insertar miembro\n";
        cout << "2. Mostrar Preorden\n";
        cout << "3. Mostrar Inorden\n";
        cout << "4. Mostrar Postorden\n";
        cout << "5. Eliminar miembro (si no tiene hijos)\n";
        cout << "6. Salir\n";

        // Validar opción
        do {
            cout << "Seleccione opción: ";
            cin >> opcion;
            if (opcion < 1 || opcion > 6) {
                cout << "Opción inválida. Intente de nuevo.\n";
            }
        } while (opcion < 1 || opcion > 6);
    } while (opcion < 1 || opcion > 6);
}

```

3.2. Pruebas de validación

A. Insertar miembro (raíz):

La validación fue exitosa, al ingresar el ID numérico, un nombre como cadena de texto y se estableció el ID 0 como padre, indicó el nuevo nodo será la raíz, el sistema reconoció correctamente esta condición y asigna el nodo como raíz del árbol.

```

Seleccione opcion: 1
Ingrese ID del nuevo miembro: 1650
Ingrese nombre: pablo
Ingrese ID del padre (0 si es raiz): 0
Raiz insertada.

```

B. Insertar miembro (Hijo Izquierdo):

La función validó que el ID del nuevo miembro no existiera previamente, buscó al nodo padre indicado y encontró que el puntero izquierdo estaba libre, por lo que insertó correctamente el nuevo nodo como hijo izquierdo del padre.

```

Opcion: 1
ID: 2
Nombre: Luis
ID del padre (0 si es raiz): 1
Insertado como hijo izquierdo.

```

C. Intento de inserción a más de 2 hijos:

La función buscó el nodo padre y verificó que ambos hijos, izquierdo y derecho, ya estaban ocupados; por lo tanto, rechazó la inserción del nuevo nodo para evitar que el padre tenga más de dos hijos.

```
Opcion: 1
ID: 4
Nombre: Jean
ID del padre (0 si es raiz): 1
El padre ya tiene dos hijos.
```

D. Eliminar miembros con hijos:

La validación fue aceptada ya que al insertar previamente un miembro que no tenía descendientes, el sistema detectó que podía eliminarse y lo hizo correctamente.

```
Opcion: 5
ID a eliminar: 3
Miembro eliminado.
```

E. Eliminar un nodo con hijos:

Al intentar eliminar un nodo que tenía descendientes el sistema detectó y rechazó el proceso para mantener la estructura del árbol la cual su validación fue exitosa.

```
Opcion: 5
ID a eliminar: 1
No se puede eliminar la raiz con hijos.
```

F. Recorridos validados tras inserciones:

Luego de insertar nodos adicionales, se verificaron los distintos tipos de recorrido y el sistema imprimió el orden correctamente la cuales siguió las funciones de **Preorden, Inorden y Postorden**, estas funciones permiten visualizar la estructura y relaciones jerárquicas.

```
Opcion: 2
1 - Pablo
2 - Luis
3 - Pedro
```

```
Opcion: 3
2 - Luis
1 - Pablo
3 - Pedro
```

```
Opcion: 4
2 - Luis
3 - Pedro
1 - Pablo
```

3.3. Manual de Usuario

Este sistema presenta una interfaz en consola que permite gestionar un árbol genealógico binario de manera interactiva, a continuación, se describirán las funciones disponibles y de qué manera se interactúa con el sistema, acompañadas de ejemplos reales de ejecución

Al iniciar, accederás a un menú con 6 opciones, cada una acompañada de una descripción clara e intuitiva para facilitar su uso.

```
***** ARBOL GENEALOGICO *****
1. Insertar miembro
2. Mostrar Preorden
3. Mostrar Inorden
4. Mostrar Postorden
5. Eliminar miembro (si no tiene hijos)
6. Salir
Seleccione opcion: |
```

A. Insertar miembro (raíz = 0):

La opción 1 permite agregar un nuevo miembro al árbol, si se trata del primer nodo, se define cómo raíz o si es otro nodo se define como 0.

```
Seleccione opcion: 1
Ingrese ID del nuevo miembro: 1
Ingrese nombre: Ana
Ingrese ID del padre (0 si es raiz): 0
Raiz insertada.
```

B. Insertar miembro (Izquierda / Derecha)

Una vez ingresada la raíz o el nodo padre, al agregar un nuevo miembro el sistema lo inserta automáticamente como hijo izquierdo si ese espacio está disponible; solo si el izquierdo ya está ocupado, se colocará como hijo derecho. No es necesario especificar manualmente el lado.

```
Opcion: 1
ID: 2
Nombre: Luis
ID del padre (0 si es raiz): 1
Insertado como hijo izquierdo.
```

C. Mostrar Preorden:

Al ingresar esta opción número 2 se muestra la jerarquía desde el nodo raíz hacia los descendientes en orden de: nodo actual - hijo izquierdo - hijo derecho.

```
Seleccione opcion: 2
Preorden:
1 - Ana
2 - Luis
3 - Pedro
```

D. Mostrar Inorden:

Al ingresar esta opción número 3 se muestra que el árbol recorre primero el hijo izquierdo, luego el nodo actual y por último el hijo derecho.

```
Seleccione opcion: 3
Inorden:
2 - Luis
1 - Ana
3 - Pedro
```

E. Mostrar Postorden:

Al ingresar esta opción número 4 se muestra que el árbol recorre los primeros hijos y luego el nodo actual, es decir que primero hijo izquierdo - hijo derecho y luego el nodo actual o raíz.

```
Seleccione opcion: 4
Postorden:
2 - Luis
3 - Pedro
1 - Ana
```

F. Eliminar miembros (si no tiene hijos)

En esta opción número 5 se permite eliminar un miembro que no tenga descendientes, si el nodo tiene hijos, la operación se cancela para salvaguardar el orden del árbol.

CON HIJOS:

```
Seleccione opcion: 5
Ingrese ID a eliminar: 1
Raiz eliminada.
```

SIN HIJOS:

```
Seleccione opcion: 5
Ingrese ID a eliminar: 1
No se puede eliminar la raiz con hijos.
```


G. Salir:

Al ingresar la opción número 6 el programa finaliza exitosamente.

```
6. Salir
Seleccione opcion: 6
Saliendo...

-----
Process exited after 2511 seconds with return value 0
Presione una tecla para continuar . . .
```

RECOMENDACIONES DE USO:

- Se recomienda insertar los ID de los nodos de manera ordenada es decir siguiendo una sucesión ascendente (1 - 2 -3; 12 - 13 - 14, A - B - C).
- Es recomendable ingresar nombres válidos.

Capítulo 4: Evidencias de Trabajo en Equipo

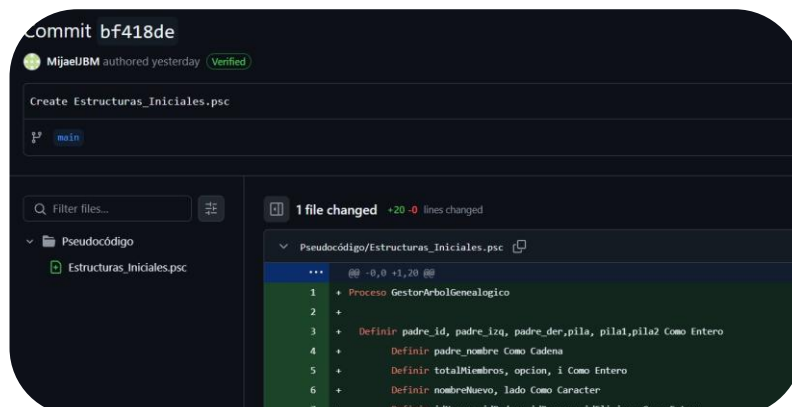
4.1. Control de Versiones

4.1.1. Registro de commits:

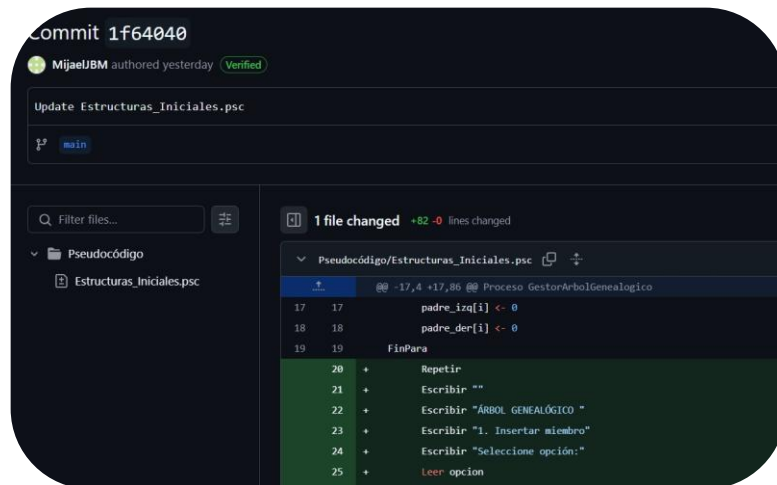
- Componentes del Pseudocódigo

Integrante: Mijael Joseph Bejarano Miche

El primer integrante subió en el repositorio de Github la primera parte del pseudocódigo correspondiente a las estructuras iniciales a las 8:00 p.m del día 17/06/2025. Comenzó creando las variables necesarias y los arreglos que almacenarán los datos de los miembros.



En un segundo commit, a las 9:35 p.m del día 17/06/2025, el integrante agregó un menú para insertar miembros al árbol genealógico. Se ingresan los datos del nuevo miembro y del padre, se valida su existencia y se asigna como hijo izquierdo o derecho.



Commit 1f64040

MijaeUBM authored yesterday (Verified)

Update Estructuras_Iniciales.psc

main

Filter files...

Pseudocódigo

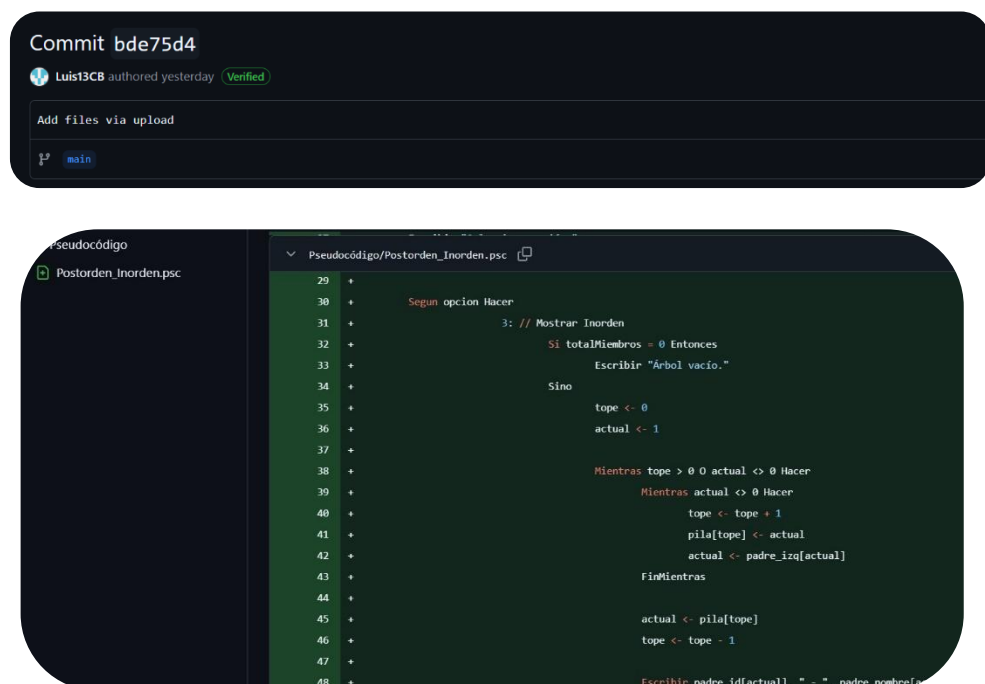
Estructuras_Iniciales.psc

1 file changed +82 -0 lines changed

```
@@ -17,4 +17,86 @@ Proceso GestorArbolGenealogico
17 17     padre_izq[i] <- 0
18 18     padre_der[i] <- 0
19 19     FinPara
20 +     Repetir
21 +         Escribir ""
22 +         Escribir "ÁRBOL GENEALÓGICO "
23 +         Escribir "1. Insertar miembro"
24 +         Escribir "Seleccione opción:"
25 +         Leer opcion
```

Integrante: Luis Enrique Cueva Barrera

El segundo integrante subió a las 9:26 p.m. del día 17/06/2025 las funciones correspondientes a los recorridos *inorden* y *postorden* del árbol binario, utilizando arreglos que simulan pilas mediante un índice de control (tope). Estas estructuras permiten recorrer el árbol sin utilizar recursividad. También se añadieron las opciones al menú y una validación para verificar si el árbol está vacío antes de mostrar los datos.



Commit bde75d4

Luis13CB authored yesterday (Verified)

Add files via upload

main

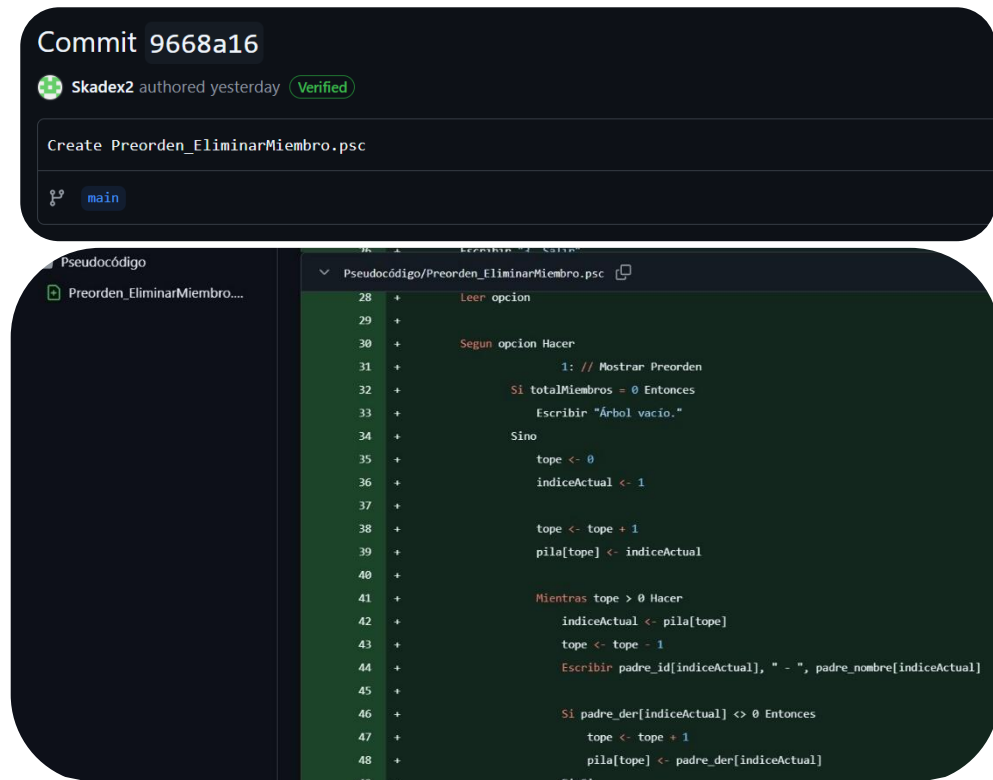
Pseudocódigo

Postorden_Inorden.psc

```
29 +
30 +     Segun opcion Hacer
31 +         3: // Mostrar Inorden
32 +         Si totalMiembros = 0 Entonces
33 +             Escribir "Árbol vacío."
34 +         Sino
35 +             tope <- 0
36 +             actual <- 1
37 +
38 +             Mientras tope > 0 0 actual <> 0 Hacer
39 +                 Mientras actual <> 0 Hacer
40 +                     tope <- tope + 1
41 +                     pila[tope] <- actual
42 +                     actual <- padre_izq[actual]
43 +                 FinMientras
44 +
45 +                 actual <- pila[tope]
46 +                 tope <- tope - 1
47 +
48 +                 Escribir padre_id[actual], " - ", padre_nombre[actual]
```

Integrante: Damián Javier López Naula

El tercer integrante envió al repositorio de GitHub a las 10:40 p.m. del 17/06/2025 la opción de recorrido en *preorden* del árbol genealógico, utilizando una pila. También agregó la funcionalidad para eliminar miembros sin hijos, actualizando el árbol cuando es necesario. Se implementó una validación para comprobar si el árbol está vacío antes de ejecutar las acciones.

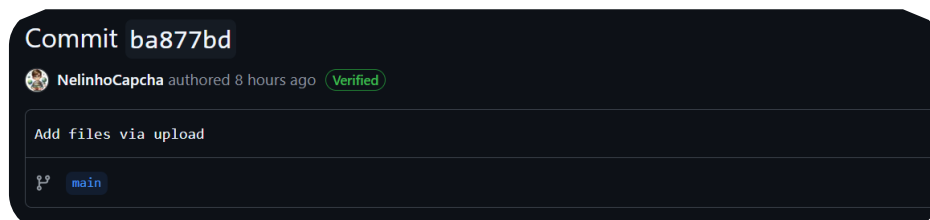


The image shows a GitHub commit interface and a code editor. The commit is titled "Commit 9668a16" and was authored by "Skadex2" yesterday. The commit message is "Create Preorden_EliminarMiembro.psc". The code editor displays the pseudocode for the "Preorden_EliminarMiembro.psc" file. The code is written in a mix of Spanish and pseudocode, using a stack to perform a pre-order traversal and eliminate members without children.

```
28 + Leer opcion
29 +
30 + Segun opcion Hacer
31 + 1: // Mostrar Preorden
32 + Si totalMiembros = 0 Entonces
33 +   Escribir "Árbol vacío."
34 + Sino
35 +   tope <- 0
36 +   indiceActual <- 1
37 +
38 +   tope <- tope + 1
39 +   pila[tope] <- indiceActual
40 +
41 +   Mientras tope > 0 Hacer
42 +     indiceActual <- pila[tope]
43 +     tope <- tope - 1
44 +     Escribir padre_id[indiceActual], " - ", padre_nombre[indiceActual]
45 +
46 +     Si padre_der[indiceActual] <> 0 Entonces
47 +       tope <- tope + 1
48 +       pila[tope] <- padre_der[indiceActual]
```

Integrante: Nelinho Capcha Hidalgo

El cuarto integrante subió el archivo final del pseudocódigo al repositorio a las 12:22 p.m. del 18/06/2025. Integró todas las funcionalidades desarrolladas anteriormente en un solo proceso. Unificó las opciones de insertar, mostrar recorridos (*preorden*, *inorden* y *postorden*), eliminar miembros y salir del programa, estructurando así el menú completo del árbol genealógico.



The image shows a GitHub commit interface. The commit is titled "Commit ba877bd" and was authored by "NelinhoCapcha" 8 hours ago. The commit message is "Add files via upload". The code editor shows the "main" branch.

```
Pseudocódigo
Gestor_Arbol_Genealógico.psc

11 + Definir indiceActual Como Entero
12 + Definir top1, top2, nodo Como Entero
13 +
14 + totalMiembros <- 0
15 +
16 + Para i <- 1 Hasta 100
17 +     padre_izq[i] <- 0
18 +     padre_der[i] <- 0
19 + FinPara
20 +
21 + Repetir
22 +     Escribir ""
23 +     Escribir "===== ÁRBOL GENEALÓGICO ====="
24 +     Escribir "1. Insertar miembro"
25 +     Escribir "2. Mostrar Preorden"
26 +     Escribir "3. Mostrar Inorden"
27 +     Escribir "4. Mostrar Postorden"
28 +     Escribir "5. Eliminar miembro"
29 +     Escribir "6. Salir"
30 +     Escribir "Seleccione opción:"
31 +     Leer opcion
32 +
```

- **Avance de código Fuente en C++**

Integrante: Damián Javier Lopez Naula

El primer integrante subió la primera parte del código en C++ al repositorio el 18/06/2025 a las 5:25 p.m. Creando la estructura básica del árbol genealógico utilizando nodos con punteros a hijos izquierdo y derecho. Añadió la función para insertar miembros, incluyendo validaciones para evitar duplicar la raíz o insertar en lados ya ocupados.

```
Commit 250ad96
Skadex2 authored 3 hours ago Verified

Create EstructurasIniciales_Insertarmiembro.cpp

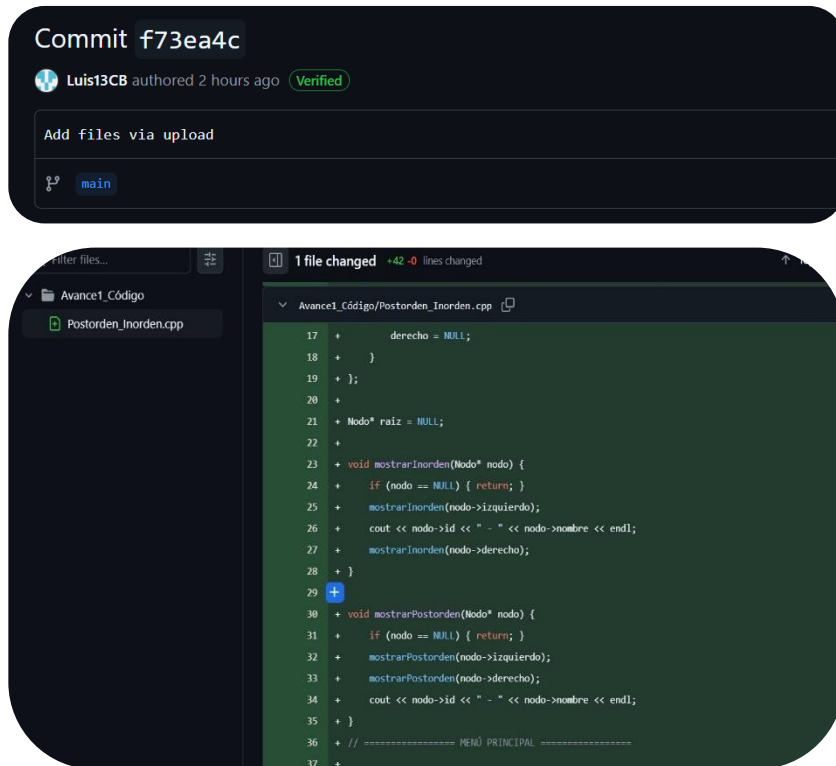
main

1 file changed +92 -0 lines changed

Avance1_Código/EstructurasIniciales_Insertarmiembro.cpp
... @@ -0,0 +1,92 @@
1 + #include <iostream>
2 + #include <string>
3 + using namespace std;
4 +
5 + struct Nodo {
6 +     int id;
7 +     string nombre;
8 +     Nodo* izquierdo;
```

Integrante: Luis Enrique Cueva Barrera

El segundo integrante subió el código en C++ al repositorio el 18/06/2025 a las 6:43 p.m., en el cual implementó las funciones de recorrido *inorden* y *postorden* para el árbol binario.

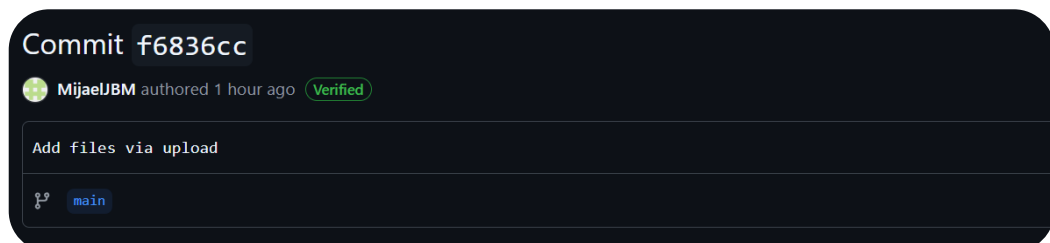


The screenshot shows a GitHub commit interface for commit `f73ea4c` by user `Luis13CB`, authored 2 hours ago. Below the commit information, there is a section for adding files via upload, with a file named `main` listed. The main part of the image is a code editor showing the file `Postorden_Inorden.cpp` in the `Avance1_Código` directory. The code implements functions for in-order and post-order traversal of a binary tree. The `mostrarInorden` function (lines 23-28) prints the node's ID and name, then recursively traverses the left and right subtrees. The `mostrarPostorden` function (lines 30-35) recursively traverses the left and right subtrees before printing the node's ID and name. Both functions include a base case for `NULL` nodes. The code is written in C++ and uses `cout` for output.

```
17 +     derecho = NULL;
18 + }
19 + };
20 +
21 + Nodo* raiz = NULL;
22 +
23 + void mostrarInorden(Nodo* nodo) {
24 +     if (nodo == NULL) { return; }
25 +     mostrarInorden(nodo->izquierdo);
26 +     cout << nodo->id << " - " << nodo->nombre << endl;
27 +     mostrarInorden(nodo->derecho);
28 + }
29 +
30 + void mostrarPostorden(Nodo* nodo) {
31 +     if (nodo == NULL) { return; }
32 +     mostrarPostorden(nodo->izquierdo);
33 +     mostrarPostorden(nodo->derecho);
34 +     cout << nodo->id << " - " << nodo->nombre << endl;
35 + }
36 + // ===== MENÚ PRINCIPAL =====
37 +
```

Integrante: Mijael Joseph Bejarano Miche

El tercer integrante envió al repositorio el 18/06/2025 a las 7:15 p.m. el código correspondiente al recorrido *preorden* del árbol binario, Además, implementó la función de eliminación de miembros, permitiendo borrar solo aquellos nodos que no tienen hijos (es decir, hojas). También incluyó una validación especial para la raíz.



The screenshot shows a GitHub commit interface for commit `f6836cc` by user `MijaelJBM`, authored 1 hour ago. Below the commit information, there is a section for adding files via upload, with a file named `main` listed.

```
Avance1_Código
Preorden_EliminarMiembro...

Avance1_Código/Preorden_EliminarMiembro.cpp
57 +
58 + void menuEliminarMiembro() {
59 +     if (raiz == NULL) {
60 +         cout << "Arbol vacio.\n";
61 +         return;
62 +     }
63 +
64 +     int idEliminar;
65 +     cout << "Ingrese ID a eliminar: ";
66 +     cin >> idEliminar;
67 +
68 +     if (raiz->id == idEliminar) {
69 +         if (raiz->izquierdo == NULL && raiz->derecho == NULL) {
70 +             delete raiz;
71 +             raiz = NULL;
72 +             cout << "Raiz eliminada.\n";
73 +         } else {
74 +             cout << "No se puede eliminar la raiz con hijos.\n";
75 +         }
76 +     } else {
77 +         if (eliminarMiembro(raiz, idEliminar)) {
78 +             cout << "Miembro eliminado.\n";
```

Integrante: Nelinho Capcha Hidalgo

El cuarto integrante subió el 18/06/2025 a las 7:42 p.m. el archivo final en C++ al repositorio. En esta versión integró todas las funciones anteriores (insertar, recorridos y eliminar miembros) y añadió un menú completo con validaciones, permitiendo al usuario interactuar con el árbol genealógico de forma sencilla.

Commit 37d4c55



NelinhoCapcha authored 1 hour ago

Verified

Add files via upload



main

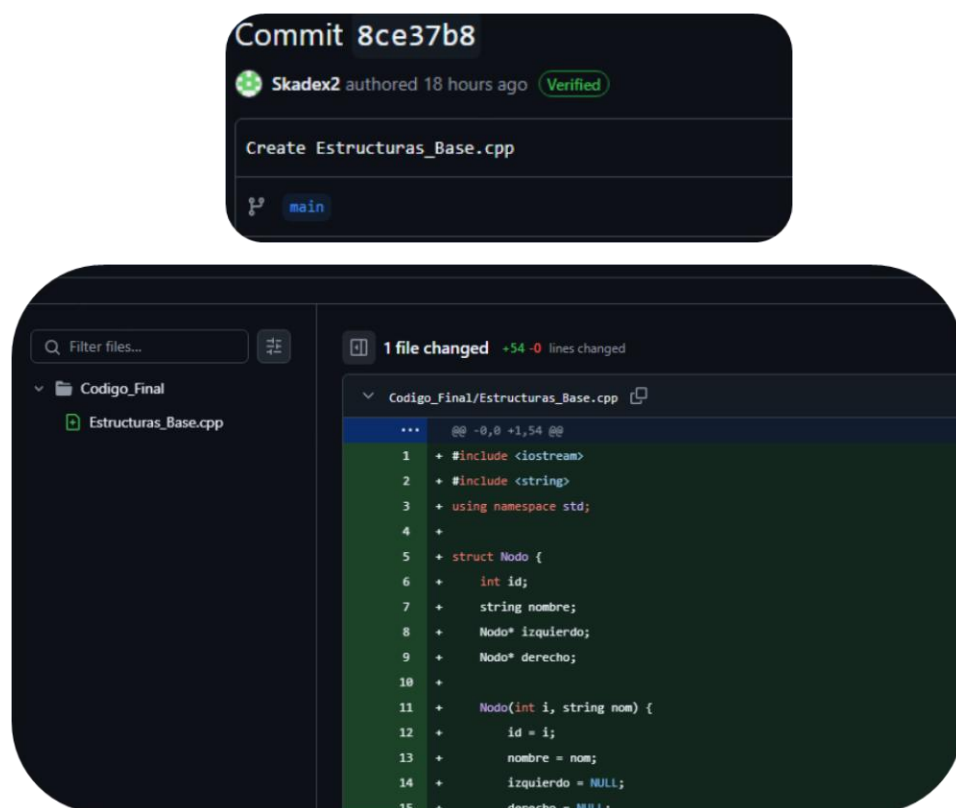
```
Avance1_Código
Arbol_Genealogico.cpp

Avance1_Código/Arbol_Genealogico.cpp
168 + int main() {
169 +     int opcion;
170 +
171 +     do {
172 +         cout << "\n***** ARBOL GENEALOGICO *****\n";
173 +         cout << "1. Insertar miembro\n";
174 +         cout << "2. Mostrar Preorden\n";
175 +         cout << "3. Mostrar Inorden\n";
176 +         cout << "4. Mostrar Postorden\n";
177 +         cout << "5. Eliminar miembro (si no tiene hijos)\n";
178 +         cout << "6. Salir\n";
179 +
180 +         do {
181 +             cout << "Seleccione opcion: ";
182 +             cin >> opcion;
183 +             if (opcion < 1 || opcion > 6) {
184 +                 cout << "Opcion invalida. Intente de nuevo.\n";
185 +             }
186 +         } while (opcion < 1 || opcion > 6);
187 +
188 +
```

- **Código Final en C++:**

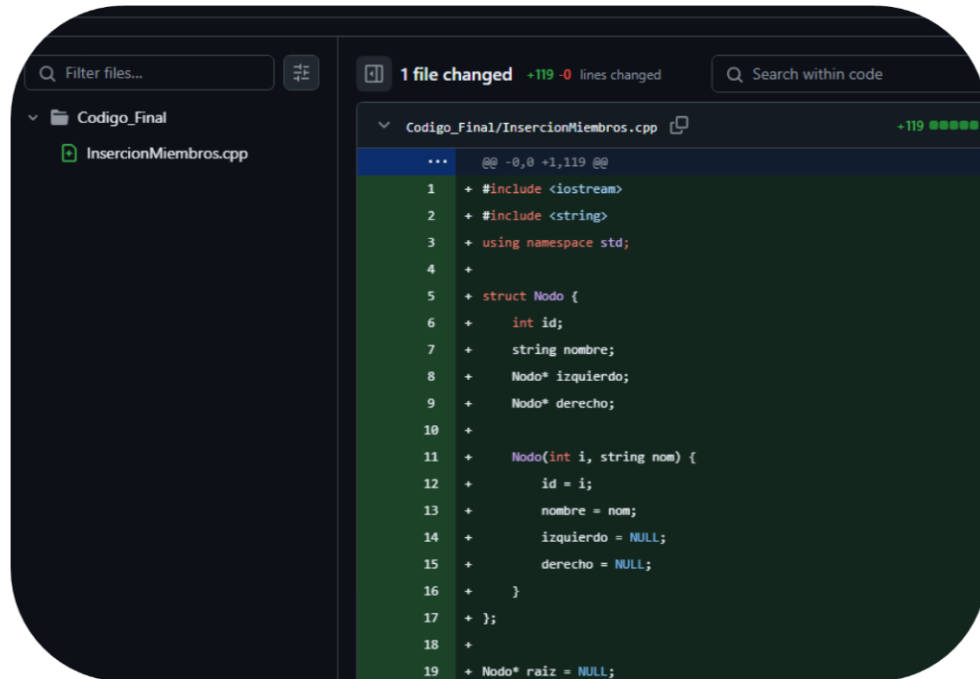
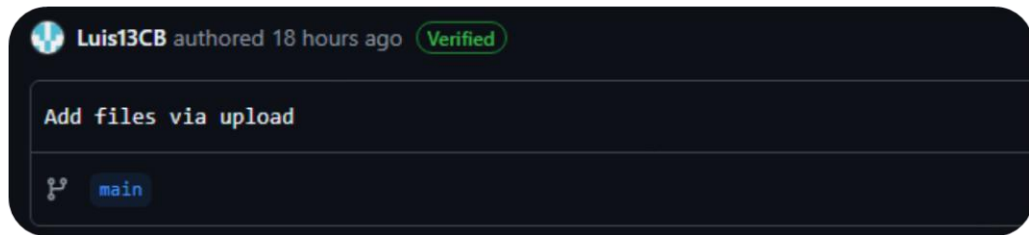
Integrante: Damian Javier Lopez Naula

El primer integrante subió en el repositorio de Github la primera parte del código correspondiente a las estructuras iniciales a las 11:00 p.m del día 23/06/2025. Diseñó la estructura principal del árbol genealógico mediante nodos enlazados con punteros a los hijos izquierdo y derecho, luego implementó la función para agregar nuevos miembros, incorporando verificaciones que impiden que la raíz se duplique o evitar que los nodos se coloquen en posiciones ocupadas.



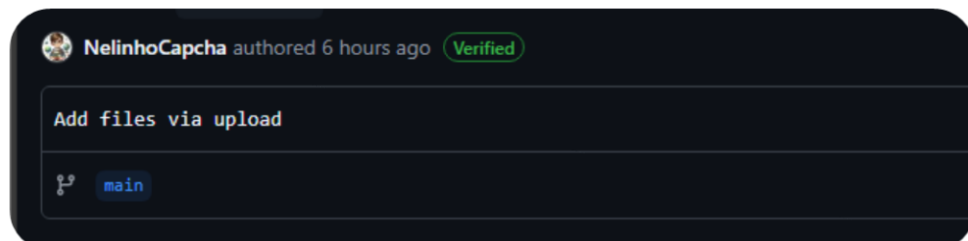
Integrante: Luis Enrique Cueva Barrera

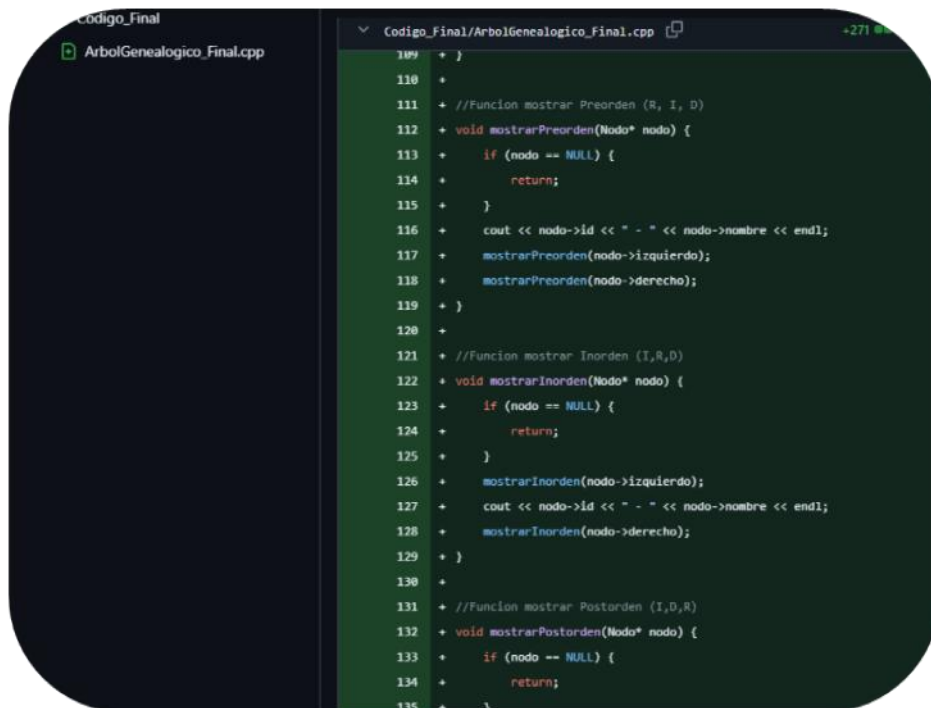
El segundo integrante subió en el repositorio de Github la segunda parte del código correspondiente a las inserciones a las 11:40 p.m del día 23/06/2025. Definió las estructuras nodo para presentar miembros del árbol genealógico con punteros a hijos izquierdos y derecho. Se implementaron funciones para verificar si un ID ya existe en el árbol genealógico con **existeID**, además se añadió un nodo padre (**buscarPadre**) y luego insertar automáticamente un nuevo nodo como hijo izquierdo o derecho si hay espacio disponible, después se desarrolló la función de **insertarMiembro** con validaciones para evitar duplicar la raíz o insertar IDs repetidos.



Integrante: Nelinho Capcha Hidalgo

El tercer integrante subió en el repositorio de Github la tercera parte del código correspondiente a los recorridos a las 11:41 a.m. del día 24/06/2025. Implementó las tres principales funciones para el recorrido del árbol genealógico la cual son de **mostrarPreorden**, **mostrarInorden** y **mostrarPostorden**, permitiendo visualizar la estructura del árbol en diferentes órdenes de visita. Además, integro las demás funciones trabajadas por los miembros del equipo en menú.

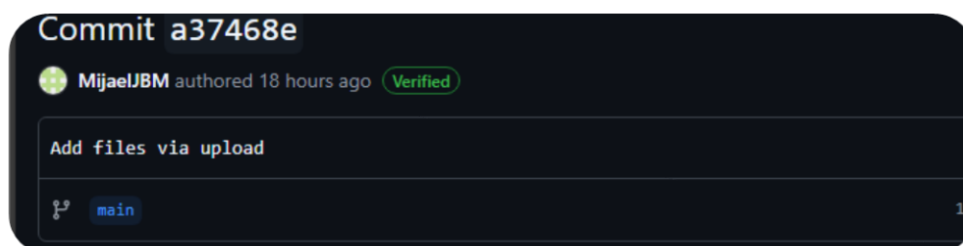


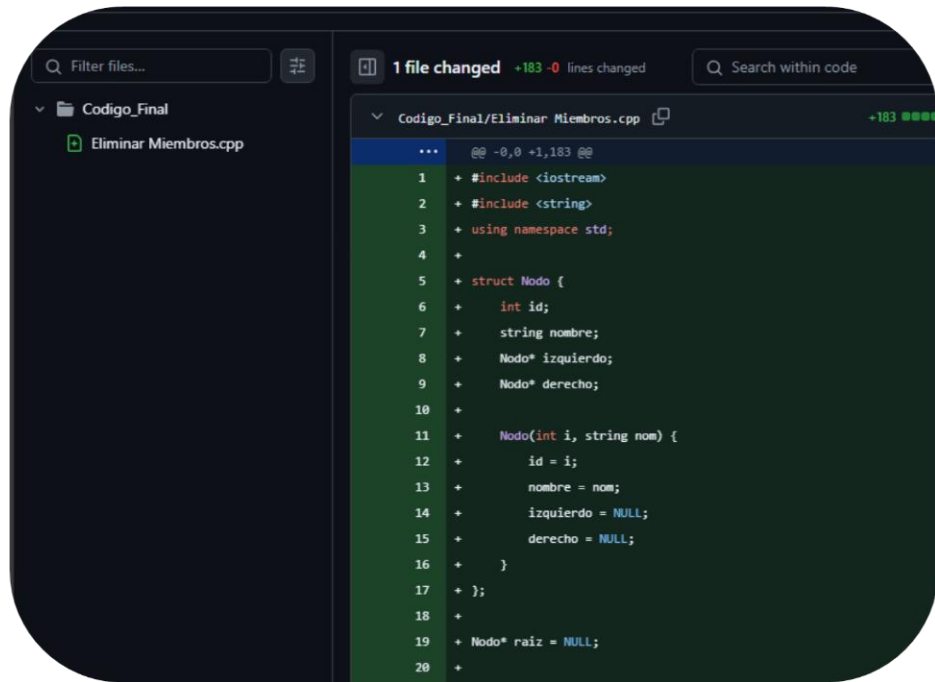


```
109 + }
110 +
111 + //Funcion mostrar Preorden (R, I, D)
112 + void mostrarPreorden(Nodo* nodo) {
113 +     if (nodo == NULL) {
114 +         return;
115 +     }
116 +     cout << nodo->id << " - " << nodo->nombre << endl;
117 +     mostrarPreorden(nodo->izquierdo);
118 +     mostrarPreorden(nodo->derecho);
119 + }
120 +
121 + //Funcion mostrar Inorden (I,R,D)
122 + void mostrarInorden(Nodo* nodo) {
123 +     if (nodo == NULL) {
124 +         return;
125 +     }
126 +     mostrarInorden(nodo->izquierdo);
127 +     cout << nodo->id << " - " << nodo->nombre << endl;
128 +     mostrarInorden(nodo->derecho);
129 + }
130 +
131 + //Funcion mostrar Postorden (I,D,R)
132 + void mostrarPostorden(Nodo* nodo) {
133 +     if (nodo == NULL) {
134 +         return;
135 +     }
```

Integrante: Mijael Joseph Bejarano Miche

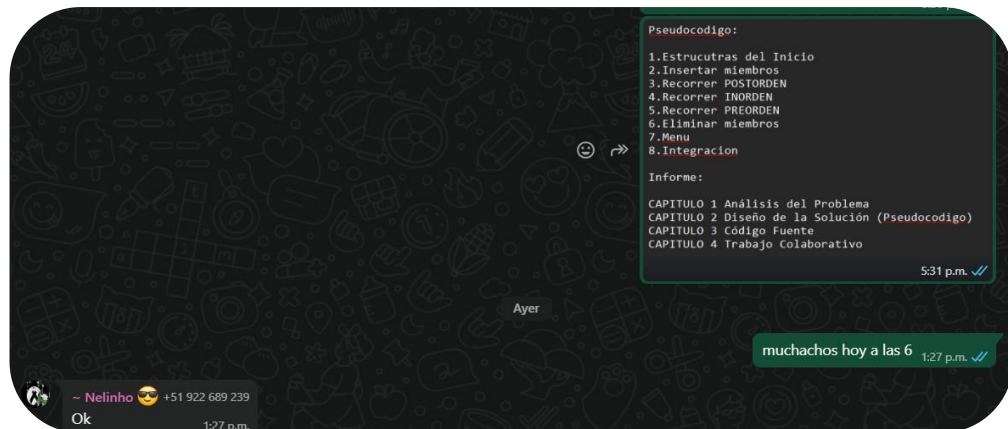
El cuarto integrante subió en el repositorio de Github la tercera parte del código correspondiente a la eliminación de miembros en el árbol genealógico a las 12:52 a.m. del día 24/06/2025. Implementó la funcionalidad para eliminar miembros del árbol genealógico mediante la función **eliminarMiembro**, la cual permite eliminar o borrar un nodo que no tenga hijos o que no sea raíz, también agregando el menú de **menuEliminarMiembro**, la cual solicita al usuario el ID del miembro a eliminar y maneja los casos especiales, como la eliminación de la raíz solo si no tiene descendencia.



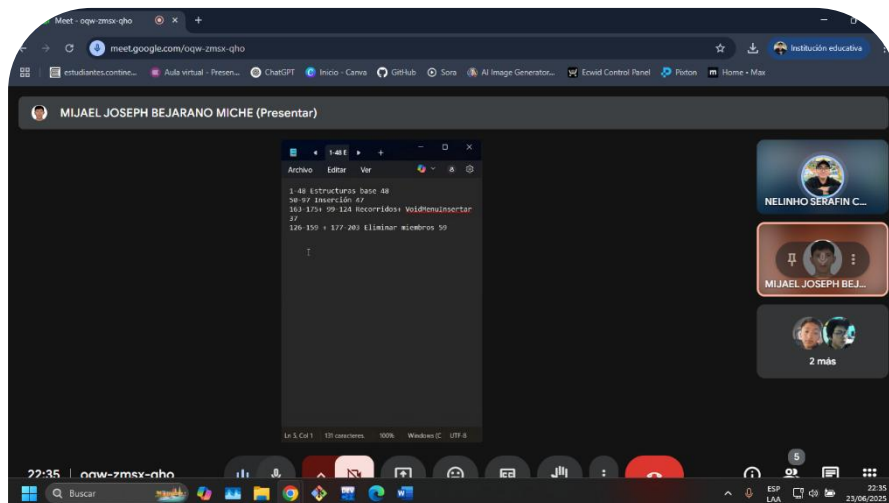
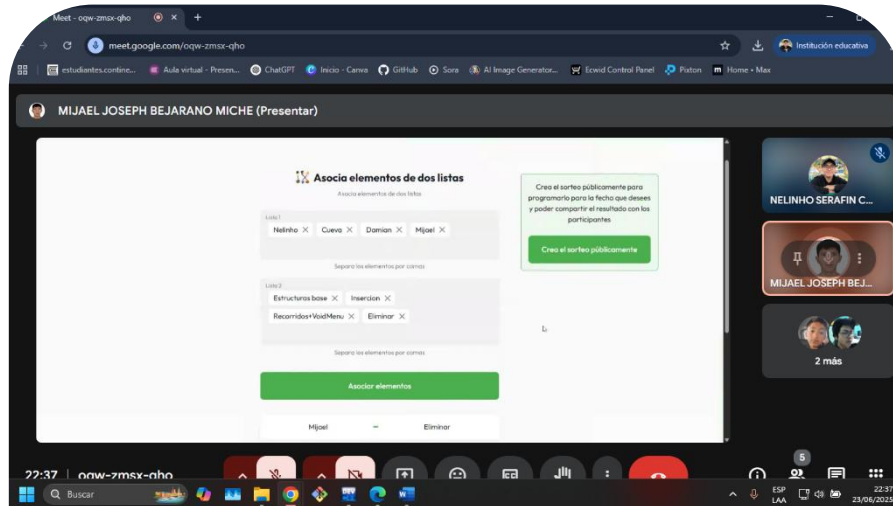
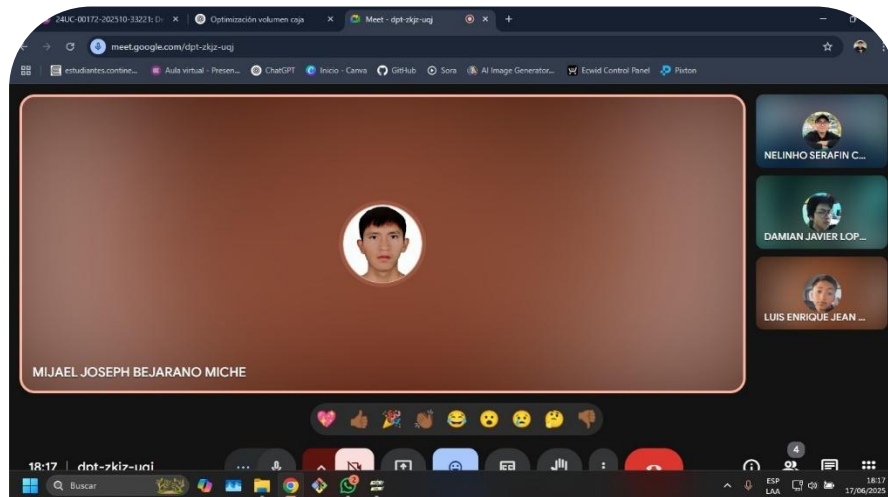


```
... @@ -0,0 +1,183 @@
1 + #include <iostream>
2 + #include <string>
3 + using namespace std;
4 +
5 + struct Nodo {
6 +     int id;
7 +     string nombre;
8 +     Nodo* izquierdo;
9 +     Nodo* derecho;
10 +
11 +     Nodo(int i, string nom) {
12 +         id = i;
13 +         nombre = nom;
14 +         izquierdo = NULL;
15 +         derecho = NULL;
16 +     }
17 + };
18 +
19 + Nodo* raiz = NULL;
20 +
```

4.1.2. Evidencias adicionales



- Se utilizó **WhatsApp** como medio principal para coordinar las reuniones y resolver cualquier duda relacionada con el desarrollo del trabajo. Además, se enviaron imágenes con los roles que fueron asignados previamente durante las llamadas.



- Mediante reuniones en **Google Meet** se definieron los roles de cada integrante, se estableció el plan de trabajo para la semana y se coordinaron las fechas de entrega, así como la forma en que se integraría el pseudocódigo y código en C++ al repositorio.

4.2. Plan de Trabajo y Roles Asignados

4.2.1. Descripción del rol semana 1:

Mediante la colaboración del equipo a través de GitHub, se decidió que un integrante diseñe las estructuras iniciales y el menú base del pseudocódigo en PSeInt, con el objetivo de facilitar la integración de las demás partes al final. Luego, cada integrante trabajó de forma individual en la función que le correspondía, y finalmente se integraron los aportes en un único programa funcional. A continuación, se presentan los roles de cada integrante.

A. Supervisor (Bejarano Mijael):

- En su **rol de supervisor**, se encargó de verificar y gestionar que los demás miembros del equipo cumplieran con sus funciones asignadas.
- **Como codificador**, fue responsable de desarrollar las estructuras iniciales del árbol binario y la función de inserción de miembros en el pseudocódigo (PSeInt). En C++, se encargó de programar el recorrido en *preorden* del árbol y la función para eliminar miembros.

B. Analista de requerimientos (Cueva Luis):

- Como **analista de requerimientos**, definió el propósito del programa, sus requerimientos funcionales y no funcionales, y las estructuras a utilizar.
- En su rol de **codificador**, implementó los recorridos *inorden* y *postorden* del árbol binario tanto en el pseudocódigo como en el código en C++.

C. Codificador principal (Capcha Nelinho):

- Como **codificador principal**, se encargó de revisar y analizar las partes del código enviadas por el equipo. Además, integró todas las funciones en un solo pseudocódigo en PSeInt y estructuró el menú completo en C++ para unir las funcionalidades desarrolladas. Además, elaboró gran parte de la explicación del pseudocódigo y código en el informe.

D. Documentador (Lopez Damián):

- Como **documentador**, se encargó de redactar y organizar gran parte del informe. Registró las funciones de cada integrante, detalló el funcionamiento del pseudocódigo y del programa en C++, y describió el proceso de trabajo colaborativo. Además, revisó la ortografía y coherencia del documento antes de su entrega.

4.2.2. Descripción de los roles semana 2:

Durante la segunda semana, el equipo continuó con la colaboración mediante GitHub para mejorar el sistema de árbol genealógico y complementar sus funcionalidades, luego se revisaron los aportes iniciales para proceder con la optimización del código, corrigiendo los errores y mejorando la organización del menú y ampliando la documentación, el trabajo se dividió en funciones específicas para cada integrante del equipo, manteniendo sesiones virtuales para coordinar avances. A continuación, se detallarán los roles y responsabilidades asignadas a cada miembro del equipo.

A. Supervisor (Bejarano Mijael):

- Como **supervisor**, se encargó de monitorear y verificar el progreso del equipo, asegurándose de que se cumplieran los plazos de entrega asignados y que cada integrante del equipo cumpliera con su tarea asignada.
- Como **codificador**, se encargó de diseñar mejoras en la función de eliminación de miembros, agregando validaciones para evitar errores en nodos con hijos, además apoyó en la verificación del correcto funcionamiento de los recorridos.

B. Analista de requerimientos (Capcha Nelinho):

- Como **analista**, definió los ajustes necesarios en el sistema tras evaluar los resultados de las pruebas iniciales, además documentó las nuevas condiciones funcionales, cómo la estructura binaria en el pseudocódigo.

- Como **codificador**, revisó y corrigió algunos detalles en los recorridos de *inorden*, *postorden* y *preorden* asegurándose que presentaran correctamente los datos del árbol.

C. Codificador principal (Cueva Luis)

- Se encargó de integrar las funciones actualizadas en el menú principal del programa en **c++**. Rediseño la estructura de navegación para hacerla más intuitiva y clara, además organizó y limpió el código fuente para facilitar la lectura y mantenimiento, así como también actualizó parte de la explicación técnica para incluir los cambios de esta semana.

D. Documentador (Lopez Damián)

- Se encargó de **redactar el registro de los cambios realizados en la semana**, detallando las mejoras hechas en el código y también explico los roles de cada integrante de la semana, así como también **amplió la documentación del programa en C++** incluyendo las descripciones para las funciones nuevas o modificadas.

4.2.3. Cronograma con fechas límite

Para hacer que el producto sea más organizado, se generó un cronograma de entrega de trabajos a cada integrante del equipo.

A. Primera Semana de Desarrollo del Sistema:

PLAN DE TRABAJO				
DÍA	Analista >	Codificador >	Supervisor >	Documentador
1	<ul style="list-style-type: none">Analiza el propósito del programa.Define requisitos funcionales y no funcionales.	<ul style="list-style-type: none">Propone las estructuras iniciales del pseudocódigo.Recibe su parte del código a desarrollar.	<ul style="list-style-type: none">Coordina la reunión por Google Meet.Asigna roles y verifica comprensión del plan.	<ul style="list-style-type: none">Anota acuerdos y define formato del informe.Crea estructura base del documento.
2	<ul style="list-style-type: none">Evalúa las estructuras definidas.Revisa si los requisitos están siendo cumplidos.	<ul style="list-style-type: none">Desarrolla su parte en PSeInt.Integra el pseudocódigo con observaciones a los miembros.	<ul style="list-style-type: none">Verifica que los códigos estén subidos a tiempo.Da feedback en caso de errores.	<ul style="list-style-type: none">Redacta descripciones de funciones.
3	<ul style="list-style-type: none">Revisa que los requisitos estén reflejados en el código final.Realiza el análisis del problema en el informe.	<ul style="list-style-type: none">Integra el primer avance del código fuente con un menú de opciones (C++).Realiza las explicaciones del pseudocódigo en el informe.	<ul style="list-style-type: none">Supervisa que todo esté listo para entrega.Elabora la explicación del código en C++ y funcionamiento final en el informe.	<ul style="list-style-type: none">Redacta y corrige el informe final.Documenta todo el proceso (commits y evidencias).


A. Segunda Semana de Desarrollo del Sistema:

PLAN DE TRABAJO				
DÍA	Analista >	Codificador >	Supervisor >	Documentador
1	<ul style="list-style-type: none"> Define el objetivo del programa, sus funciones y restricciones Analiza el sistema anterior 	<ul style="list-style-type: none"> Propone las estructuras iniciales en C++ Implementa las estructuras de datos 	<ul style="list-style-type: none"> Coordina la reunión por Google Meet. Asigna roles y verifica comprensión del plan. 	<ul style="list-style-type: none"> Anota acuerdos y define formato del informe. Redacta los objetivos, funciones y roles asignados en el documento
2	<ul style="list-style-type: none"> Programa los recorridos en Preorden, Inorden y Postorden Revisa si los requisitos están siendo cumplidos. 	<ul style="list-style-type: none"> Desarrolla su parte en C++ Integra las funciones al menú principal en C++ 	<ul style="list-style-type: none"> Verifica la estructura de los códigos para luego unirlos Da feedback en caso de errores. 	<ul style="list-style-type: none"> Redacta explicación de funciones implementadas y realiza pruebas básicas
3	<ul style="list-style-type: none"> Revisa que los requisitos estén reflejados en el código final. Revisa que el programa cumpla con los objetivos definidos 	<ul style="list-style-type: none"> Integra el primer avance del código fuente con un menú de opciones (C++). Limpia el código para luego organizar el menú final y hace ajustes para la presentación 	<ul style="list-style-type: none"> Supervisa que todo esté listo para entrega. Verifica el correcto funcionamiento del menú y corrige los errores detectados 	<ul style="list-style-type: none"> Finaliza el informe corrigiendo la ortografía y prepara la versión lista Documenta todo el proceso (commits y evidencias).

- LINK DEL GITHUB:

 https://github.com/Skadex2/E.D.D.-ABR_.git

- LINK DEL CANVA:

 https://www.canva.com/design/DAGrT0F5GXQ/Fslam3SQw75K_l-aEwNwBw/edit?utm_content=DAGrT0F5GXQ&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton