UNIVERSIDAD CONTINENTAL

FACULTAD DE INGENIERÍA

INFORME DE ESTRUCTURA DE DATOS

NRC: 29901

Asignatura: Estructura de datos
Docente: Osorio Contreras Rosario Delia

Tema

“ÁRBOL GENEALÓGICO - ABR”

INTEGRANTES:

- García Betancourt Israel Jedidias
- Leon Armas Luis Aram
- Ramírez Quillatupa Juan Diego
- Veliz Durand Vieri Del Piero

2025 - 10
Huancayo - Perú

ÍNDICE

ABR – IMPLEMENTACIÓN

Capítulo 1 - Fase de Ideación

1. Descripción del problema

Un grupo de arqueólogos ha descubierto los restos de una antigua civilización. Tras un análisis detallado, se revela que su historia y linajes pueden representarse mediante un árbol genealógico. Esta representación es fundamental para entender cómo estaban organizadas las familias, qué jerarquías existían, quiénes fueron sus ancestros y cómo se distribuyeron los linajes a lo largo del tiempo.

Sin embargo, la información obtenida se encuentra dispersa y debe organizarse de forma estructurada. Para ello, se requiere una aplicación informática que permita almacenar y organizar estos datos de forma eficiente, brindando la posibilidad de realizar consultas rápidas sobre ancestros, descendientes y relaciones familiares.

El reto principal radica en crear una solución tecnológica capaz de representar fielmente esta estructura genealógica, facilitando el acceso y manipulación de los datos históricos descubiertos.

2. Requerimientos del sistema

2.1 Funcionales

- **Registro de individuos:** El sistema debe permitir ingresar personas con atributos como nombre, fecha aproximada, linaje o generación.

- **Relaciones familiares:** Posibilidad de registrar vínculos como padre, madre, hijo(a), y hermanos

- **Consultas genealógicas:** Las consultas genealógicas permiten extraer información específica del árbol familiar de acuerdo con las necesidades del análisis. Entre las más comunes se encuentran la búsqueda de ancestros, tanto directos como indirectos, lo cual permite rastrear el origen familiar de una persona. También es posible consultar los descendientes de un individuo determinado, es decir, obtener información sobre sus hijos, nietos y generaciones posteriores. Asimismo, se pueden visualizar líneas genealógicas completas o parciales, ya sea ascendentes o descendentes, lo que facilita comprender la estructura y la extensión del árbol familiar en función del objetivo planteado.

- **Edición de datos:** Actualizar o eliminar relaciones o individuos registrados.

2.2. No funcionales

- **Rendimiento:** El sistema debe responder rápidamente incluso con árboles de gran tamaño (miles de registros).
- **Interfaz amigable:** Fácil de usar, con menús intuitivos para arqueólogos o investigadores.
- **Escalabilidad:** Capacidad para manejar ampliaciones del árbol conforme se descubren nuevos datos.
- **Seguridad:** Protección de los datos históricos almacenados para evitar pérdidas o alteraciones accidentales.

3. Respuestas a la pregunta guía

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

Cada nodo del árbol binario de búsqueda (ABB) representaría a una persona e incluiría información como su nombre completo, fecha de nacimiento y, en caso de haber fallecido, su fecha de defunción. Además, se asignaría un ID único que actuaría como clave para organizar y localizar eficientemente a los individuos dentro del árbol. El nodo también almacenaría relaciones familiares, como referencias al padre, la madre (si se permite) y los hijos, lo que facilitaría la representación de la estructura genealógica. Opcionalmente, se podrían incluir otros metadatos, como el género o el lugar de nacimiento, para enriquecer la información. Esta estructura no solo captura la identidad de cada persona, sino también su posición dentro de la genealogía, aprovechando las propiedades del ABB para garantizar un acceso y organización óptimos.

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

Para insertar un nuevo miembro en el árbol sin romper la estructura del Árbol Binario de Búsqueda (ABB), se utiliza un ID único como clave que permite mantener el orden. Si el ID del nuevo miembro es menor que el del nodo actual, se inserta en la rama izquierda; si es mayor, se coloca en la rama derecha. Las relaciones entre nodos (padre e hijos) se establecen de manera implícita según estas conexiones.

Para eliminar un miembro del árbol sin alterar su estructura, existen tres casos:

- Si el nodo es una hoja, se puede eliminar directamente.
- Si tiene un solo hijo, este hijo reemplaza al nodo eliminado.

- Si tiene dos hijos, se sustituye por su sucesor inorden (el nodo más pequeño del subárbol derecho) para mantener la propiedad del ABB.

En ambos casos, es fundamental actualizar correctamente las referencias entre nodos (padre, madre o hijos, si se usan explícitamente) para conservar la coherencia del árbol genealógico.

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

Para visualizar correctamente la genealogía en un árbol binario, se utilizan distintos métodos de recorrido que permiten interpretar las relaciones entre los miembros según el tipo de análisis que se quiera realizar. Los tres métodos más utilizados son:

- **Preorden (root-left-right):** Este recorrido comienza por la raíz del árbol (el ancestro común) y continúa hacia la izquierda y luego hacia la derecha. Es ideal para representar la genealogía desde el antecesor hacia sus descendientes, lo que permite construir un relato generacional desde el origen familiar. Además, es útil cuando se desea guardar o imprimir toda la estructura jerárquica respetando el orden generacional.

- **Inorden (left-root-right):** Recorre primero el subárbol izquierdo, luego el nodo actual y finalmente el derecho. Este método es especialmente útil cuando se quiere listar a los miembros de la genealogía en orden cronológico, siempre que el árbol utilice la fecha de nacimiento como clave. De esta manera, se puede obtener un registro ordenado de las generaciones, lo cual es valioso para análisis históricos o demográficos.

- **Postorden (left-right-root):** En este recorrido, se procesan primero los subárboles izquierdo y derecho, y finalmente la raíz. Es útil cuando se desea eliminar todos los nodos del árbol desde los más jóvenes hacia el ancestro, respetando una lógica de eliminación inversa a la generación. También se aplica en procesos como la liberación de memoria o la depuración del árbol genealógico en sistemas automatizados.

Cada uno de estos métodos cumple una función específica y se puede emplear dependiendo del tipo de información que se desea obtener del árbol. Por ejemplo, si se quiere responder a preguntas como “¿Cuáles son los descendientes de un antecesor específico?” se optaría por el recorrido en preorden. En cambio, si se busca analizar la genealogía desde una perspectiva temporal, el recorrido inorden es el más adecuado. Así, estos métodos no solo

permiten navegar el árbol, sino también entender la estructura familiar desde diferentes enfoques.

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

Para determinar si un miembro pertenece a una rama específica dentro de un árbol genealógico, se realiza una búsqueda recursiva a partir del nodo raíz o desde un antepasado determinado. Durante esta búsqueda, se analiza si el ID del miembro buscado aparece en el subárbol izquierdo o derecho del nodo actual. Si se encuentra, se concluye que dicho miembro pertenece a esa rama. Este procedimiento también puede llevarse a cabo siguiendo los punteros de los hijos y comparando sus identificadores con el ID objetivo.

Este método resulta útil para responder preguntas como: "¿Está esta persona dentro del linaje de esta familia?" o "¿Desciende de este antepasado?", ya que permite verificar de forma precisa si existe una conexión directa o indirecta entre dos nodos del árbol, manteniendo así la coherencia y trazabilidad de la genealogía representada.

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

Con el tiempo, un árbol genealógico basado en un árbol binario de búsqueda (ABB) podría desequilibrarse si los nodos se insertan en un orden específico, como por fechas o IDs secuenciales, lo que resultaría en una estructura similar a una lista lineal en uno de los lados. Este desbalance afectaría negativamente el rendimiento de las operaciones de búsqueda y recorrido. Para solucionar este problema, se pueden implementar estrategias como el uso de un Árbol AVL, que realiza ajustes automáticos en las alturas de las ramas después de cada inserción o eliminación, garantizando un equilibrio óptimo. Otra alternativa es emplear un Árbol Rojo-Negro, que ofrece mayor eficiencia en ciertos escenarios. Además, existe la opción de realizar un rebalanceo manual, recolectando todos los nodos en orden y reconstruyendo el árbol de manera equilibrada, insertando los elementos de forma estratégica. El objetivo principal es mantener un balance entre las ramas izquierda y derecha para asegurar que las operaciones sigan siendo eficientes, incluso cuando el árbol crezca con muchos miembros.

Capítulo 2 – Prototipo

1. Descripción de estructuras de datos y operaciones

1.1. Pseudocódigo para crear un árbol binario

Se define una estructura llamada “Nodo”, que representa a cada persona dentro del árbol genealógico.

Cada nodo contiene los siguientes campos:

- id: una cadena de texto que actúa como identificador único de la persona.
- nombre: el nombre de la persona.
- apellido: el apellido de la persona.
- padre: puntero al nodo que representa al padre (antecesor inmediato).
- izq: puntero al hijo izquierdo (primer hijo).
- der: puntero al hijo derecho (segundo hijo).

Esta estructura permite construir un árbol binario con jerarquía familiar, donde cada persona puede tener como máximo dos hijos directos y un único padre. Este modelo es útil para representar genealogías de manera estructurada y organizada.

1.2. Operaciones implementadas

1.1.1. Búsqueda por ID (buscarPorID)

- Permite localizar a una persona dentro del árbol usando su identificador único (id).
- Utiliza un recorrido recursivo en preorden (raíz, hijo izquierdo, hijo derecho).
- Si el nodo con el ID buscado se encuentra, se retorna su puntero; de lo contrario, se retorna NULL.

1.1.2. Verificación de ancestro (esAncestro)

- Determina si un nodo dado es ancestro directo de otro.
- Recorre hacia arriba utilizando el puntero padre desde el nodo objetivo hasta llegar a la raíz.
- Esta función es esencial para prevenir la creación de ciclos genealógicos al insertar nuevos miembros en el árbol.

1.1.3. Inserción de persona (insertarPersona)

- Permite agregar una nueva persona al árbol como hijo de un nodo padre ya existente.
- Se validan las siguientes condiciones antes de la inserción:
    - Que el padre exista en el árbol.
    - Que la persona nueva no tenga ya un padre asignado.
    - Que no se cree un ciclo genealógico (verificación mediante esAncestro).
    - Que el padre tenga espacio disponible (máximo dos hijos).
- Si todo es válido, la nueva persona se enlaza como hijo izquierdo (izq) o derecho (der) del padre.

1.1.4. Recorridos del árbol (Preorden, Inorden y Postorden)

- Preorden (preorden): primero muestra el nodo actual, luego recorre el hijo izquierdo y finalmente el derecho.
- Inorden (inorden): primero recorre el hijo izquierdo, luego muestra el nodo actual, y después recorre el hijo derecho.
- Postorden (postorden): recorre primero ambos hijos (izquierdo y derecho) y finalmente muestra el nodo actual.
- Estos recorridos se usan para visualizar el árbol en distintos órdenes según el análisis deseado.

1.1.5. Mostrar ascendencia detallada (mostrarAscendenciaDetallada)

- Muestra la línea ascendente de una persona desde el nodo seleccionado hasta la raíz.
- Se presenta el nivel jerárquico de cada ancestro encontrado.
- Además, se detectan y muestran los hermanos del nodo en cada nivel (si existen).
- Esta operación proporciona una vista clara y estructurada del linaje familiar.

1.1.6. Añadir persona (anadirPersona)

- Solicita datos básicos como ID, nombre, apellido e ID del padre.
- Verifica que el ID sea único.
- Si el árbol está vacío, la persona se convierte en la raíz del árbol.
- Si no, se llama a insertarPersona para añadirla bajo el padre indicado.

1.1.7. Eliminación de persona (eliminarPersona)

- Permite eliminar un nodo del árbol solo si no tiene hijos.

- Si el nodo a eliminar es la raíz y no tiene hijos, el árbol queda vacío.
- Si el nodo tiene hijos, la eliminación es rechazada para mantener la integridad del árbol genealógico.

1.1.8. Submenú de métodos de búsqueda (submenuMetodosBusqueda)

- Ofrece un menú para que el usuario pueda ejecutar cualquiera de los tres recorridos del árbol.

- Después de mostrar los resultados, permite volver al menú principal.

1.1.9. Búsqueda de persona y ascendencia

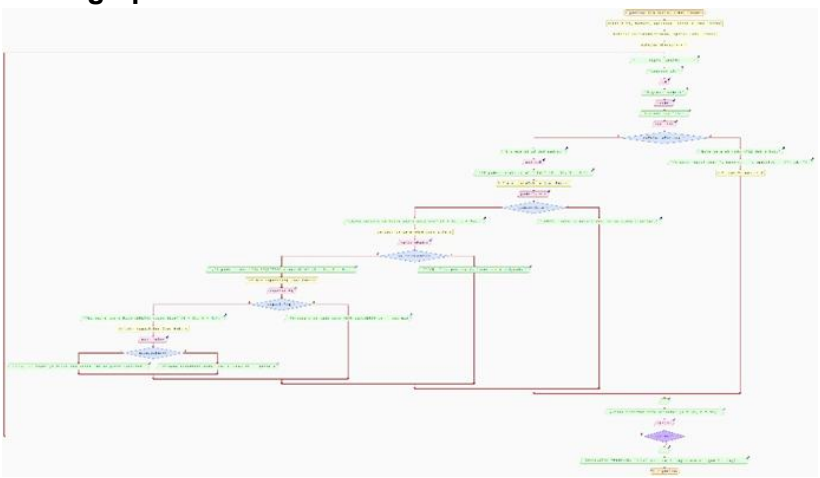
- Solicita un ID y utiliza buscarPorID para ubicar a la persona.
- Si se encuentra, se muestra la ascendencia detallada usando mostrarAscendenciaDetallada.

1.1.10. Menú principal (menu)

- Controla toda la interacción con el usuario mediante opciones por consola.
- Las opciones disponibles incluyen: agregar persona, eliminar persona, mostrar recorridos, buscar persona por ID y salir del programa.

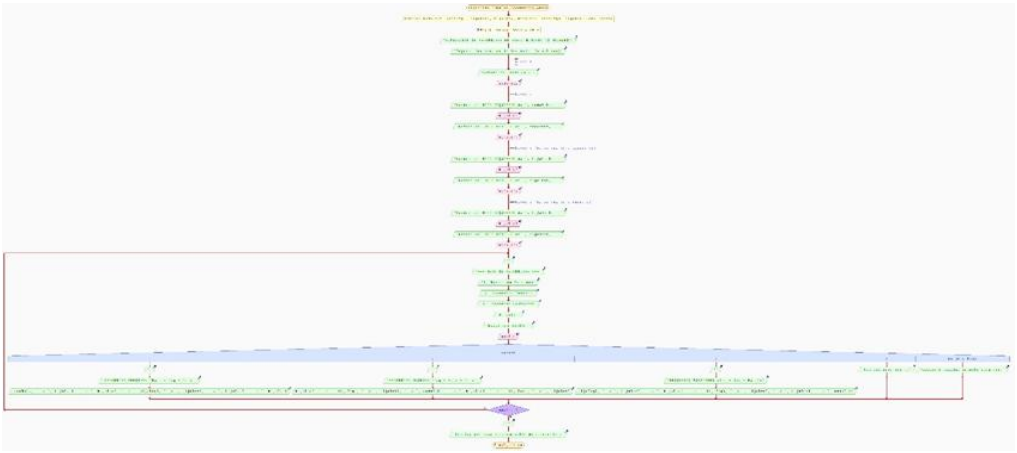
2. Algoritmos principales

2.1. Pseudocódigo para crear un árbol binario



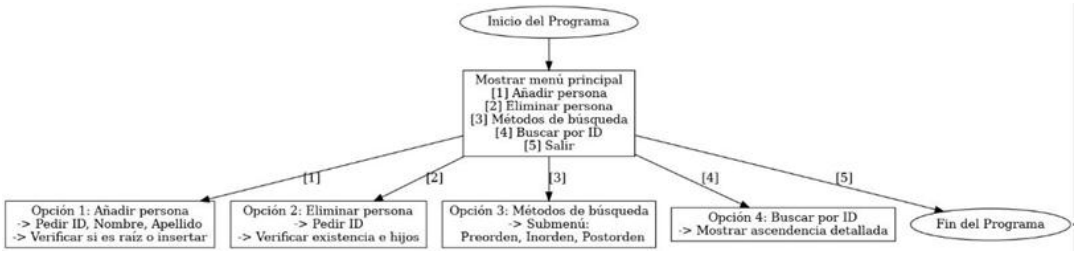
https://drive.google.com/drive/folders/1A_dJmSbkKeUvANEfRnhPoYXFRTfMkJqz?usp=drive_link

2.2. Pseudocódigo para realizar el recorrido de un árbol



https://drive.google.com/drive/folders/1A_dJmSbkKeUvANEfRnhPoYXFRTfMkJqz?usp=drive_link

3. Diagramas de Flujo



4. Avance del código fuente

A continuación, se muestran capturas del código fuente desarrollado hasta el momento para el prototipo del árbol genealógico. Estas incluyen la estructura del nodo, funciones de inserción, eliminación, búsqueda, recorridos y el menú principal.

4.1. Definición de estructuras

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

// Definición de la estructura Nodo para representar a una persona en un árbol genealógico
struct Nodo {
    string id;          // Identificador único de la persona
    string nombre;      // Nombre de la persona
    string apellido;    // Apellido de la persona
    Nodo* padre;        // Puntero al nodo del padre
    Nodo* izq;          // Hijo izquierdo (puede representar al primer hijo)
    Nodo* der;          // Hijo derecho (puede representar al segundo hijo)
};
```

4.2. Funciones principales del árbol

4.2.1. Función para buscar persona por ID

```cpp
// Función para buscar una persona en el árbol por su ID
Nodo* buscarPorID(Nodo* nodo, const string& id) {
    if (nodo == NULL) return NULL;              // Si el nodo es nulo, no se encontró
    if (nodo->id == id) return nodo;            // Si el ID coincide, se encontró
    Nodo* encontrado = buscarPorID(nodo->izq, id); // Buscar recursivamente en el subárbol izquierdo
    if (encontrado) return encontrado;          // Si se encontró en el subárbol izquierdo, devolverlo
    return buscarPorID(nodo->der, id);          // Si no, buscar en el subárbol derecho
}
```

4.2.2. Insertar persona al árbol

```cpp
// Función para insertar una nueva persona en el árbol bajo un padre especificado
bool insertarPersona(Nodo* raiz, Nodo* nueva, const string& padreID) {
    Nodo* padre = buscarPorID(raiz, padreID);     // Buscar el nodo del padre
    if (padre == NULL) {
        cout << "Error: Padre no encontrado.\n";  // Validar existencia del padre
        return false;
    }
    if (nueva->padre != NULL) {
        // Validar que la nueva persona no tenga ya un padre asignado
        cout << "Error: Esta persona ya tiene un padre asignado (" << nueva->padre->id << ").\n";
        return false;
    }
    if (esAncestro(nueva, padre)) {
        // Prevenir ciclos: no se puede insertar un ancestro como hijo de su descendiente
        cout << "Error: No se puede insertar (crear a un ciclo).\n";
        return false;
    }
    nueva->padre = padre;                         // Asignar el padre
    if (padre->izq == NULL) {
        padre->izq = nueva;                       // Insertar como hijo izquierdo si está disponible
        return true;
    } else if (padre->der == NULL) {
        padre->der = nueva;                       // Insertar como hijo derecho si está disponible
        return true;
    } else {
        cout << "Error: El padre ya tiene dos hijos.\n"; // Si ya tiene dos hijos, no se puede insertar
        return false;
    }
}
```

4.2.3. Recorridos del árbol

```cpp
// Recorrido en preorden: visita raíz, luego hijo izquierdo y luego derecho
void preorden(Nodo* nodo) {
    if (nodo != NULL) {
        cout << nodo->nombre << " " << nodo->apellido << " (" << nodo->id << ")\n";
        preorden(nodo->izq);  // Visitar hijo izquierdo
        preorden(nodo->der);  // Visitar hijo derecho
    }
}

// Recorrido en inorden: hijo izquierdo, luego raíz, luego derecho
void inorden(Nodo* nodo) {
    if (nodo != NULL) {
        inorden(nodo->izq);   // Visitar hijo izquierdo
        cout << nodo->nombre << " " << nodo->apellido << " (" << nodo->id << ")\n";
        inorden(nodo->der);   // Visitar hijo derecho
    }
}

// Recorrido en postorden: hijo izquierdo, hijo derecho, luego raíz
void postorden(Nodo* nodo) {
    if (nodo != NULL) {
        postorden(nodo->izq);  // Visitar hijo izquierdo
        postorden(nodo->der);  // Visitar hijo derecho
        cout << nodo->nombre << " " << nodo->apellido << " (" << nodo->id << ")\n";
    }
}
```

4.3. Operaciones del sistema

4.3.1. Añadir persona

```
145  // Permite añadir una nueva persona al árbol
146  void anadirPersona(Nodo*& raiz) {
147      Nodo* nueva = new Nodo();  // Crear nuevo nodo
148
149      cout << "ID unico: ";
150      getline(cin, nueva->id);
151
152      // Verificar si ya existe una persona con ese ID
153      if (raiz != NULL && buscarPorID(raiz, nueva->id) != NULL) {
154          cout << "Error: Ya existe una persona con ese ID.\n";
155          delete nueva; // Liberar memoria si ya existe
156          return;
157      }
158
159      // Ingresar nombre y apellido
160      cout << "Nombre: ";
161      getline(cin, nueva->nombre);
162      cout << "Apellido: ";
163      getline(cin, nueva->apellido);
164
165      // Inicializar punteros
166      nueva->izq = nueva->der = nueva->padre = NULL;
167
168      if (raiz == NULL) {
169          raiz = nueva; // Si el árbol está vacío, esta persona es la raíz
170          cout << "Anadido como raiz del arbol.\n";
171      } else {
172          // Solicitar ID del padre y tratar de insertar
173          string padreID;
174          cout << "ID del padre: ";
175          getline(cin, padreID);
176          if (!insertarPersona(raiz, nueva, padreID)) {
177              delete nueva; // Si falla, liberar memoria
178          } else {
179              cout << "Persona anadida al arbol.\n";
180          }
181      }
182  }
183
```

4.3.2. Eliminar persona

```
184  // Permite eliminar una persona del árbol si no tiene hijos
185  void eliminarPersona(Nodo*& raiz, const string& id) {
186      Nodo* objetivo = buscarPorID(raiz, id); // Buscar a la persona
187      if (objetivo == NULL) {
188          cout << "Error: Persona no encontrada.\n";
189          return;
190      }
191
192      // No se permite eliminar personas con hijos
193      if (objetivo->izq != NULL || objetivo->der != NULL) {
194          cout << "Error: No se puede eliminar. Esta persona tiene hijos.\n";
195          return;
196      }
197
198      // Si es la raíz y no tiene hijos, eliminarla y vaciar el árbol
199      if (objetivo == raiz) {
200          delete raiz;
201          raiz = NULL;
202          cout << "Raiz eliminada. Arbol vacio.\n";
203          return;
204      }
205
206      // Si no es la raíz, eliminarla del padre
207      Nodo* padre = objetivo->padre;
208      if (padre->izq == objetivo)
209          padre->izq = NULL;
210      else if (padre->der == objetivo)
211          padre->der = NULL;
212
213      delete objetivo; // Liberar memoria del nodo
214      cout << "Persona eliminada correctamente.\n";
215  }
```

4.4. Menú principal

```
void menu() {
    Nodo* raiz = NULL; // Árbol inicialmente vacío
    int op;

    do {
        // Mostrar opciones del menú principal
        cout << "*** MENU ARBOL GENEALOGICO ***\n";
        cout << "[1]: Anadir persona\n";
        cout << "[2]: Eliminar persona\n";
        cout << "[3]: Metodos de busqueda (Preorden, Inorden, Postorden)\n";
        cout << "[4]: Buscar persona por ID (ascendencia)\n";
        cout << "[5]: Salir\n";

        // Validar opción ingresada
        do {
            cout << "Elegir opcion: ";
            cin >> op;
            if (op < 1 || op > 5) {
                cout << "ERROR, opcion incorrecta\n";
            }
        } while (op < 1 || op > 5);

        cin.ignore(); // Limpiar buffer

        // Ejecutar opción elegida
        switch (op) {
            case 1:
                // Añadir nueva persona al árbol
                anadirPersona(raiz);
                break;
            case 2: {
                // Eliminar persona del árbol
                string id;
                cout << "ID de la persona a eliminar: ";
                getline(cin, id);
                eliminarPersona(raiz, id);
                break;
            }
            case 3:
                // Acceder a submenú de métodos de recorrido
                submenuMetodosBusqueda(raiz);
                break;
            case 4:
                // Buscar persona y mostrar su ascendencia
                buscarPersonaYAscendencia(raiz);
                break;
            case 5:
                // Salir del programa
                break;
        }

        // Pausar y limpiar pantalla si no se ha elegido salir
        if (op != 5) {
            system("pause");
            system("cls");
        }

    } while (op != 5);
}
```

Capítulo 3 - Solución Final

1. Código limpio, bien comentado y estructurado.

1.1. Definición de estructuras

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

// Definición de la estructura Nodo para representar a una persona en un árbol genealógico
struct Nodo {
    string id;          // Identificador único de la persona
    string nombre;      // Nombre de la persona
    string apellido;    // Apellido de la persona
    Nodo* padre;        // Puntero al nodo del padre
    Nodo* izq;          // Hijo izquierdo (puede representar al primer hijo)
    Nodo* der;          // Hijo derecho (puede representar al segundo hijo)
};
```

1.2. Funciones principales del árbol

1.2.1. Función para buscar persona por ID

```cpp
// Función para buscar una persona en el árbol por su ID
Nodo* buscarPorID(Nodo* nodo, const string& id) {
    if (nodo == NULL) return NULL;              // Si el nodo es nulo, no se encontró
    if (nodo->id == id) return nodo;            // Si el ID coincide, se encontró
    Nodo* encontrado = buscarPorID(nodo->izq, id); // Buscar recursivamente en el subárbol izquierdo
    if (encontrado) return encontrado;          // Si se encontró en el subárbol izquierdo, devolverlo
    return buscarPorID(nodo->der, id);          // Si no, buscar en el subárbol derecho
}

```

1.2.2. Verificación de ancestros

```cpp
// Función para verificar si un nodo es ancestro de otro (para evitar ciclos)
bool esAncestro(Nodo* posibleAncestro, Nodo* nodo) {
    while (nodo != NULL) {
        if (nodo == posibleAncestro)            // Si encontramos el nodo en la cadena ascendente, es ancestro
            return true;
        nodo = nodo->padre;                     // Subir al padre
    }
    return false;                               // Si se llega al final sin coincidencias, no es ancestro
}
```

1.2.3. Insertar persona al árbol

```cpp
// Función para insertar una nueva persona en el árbol bajo un padre especificado
bool insertarPersona(Nodo* raiz, Nodo* nueva, const string& padreID) {
    Nodo* padre = buscarPorID(raiz, padreID);     // Buscar el nodo del padre
    if (padre == NULL) {
        cout << "Error: Padre no encontrado.\n";  // Validar existencia del padre
        return false;
    }
    if (nueva->padre != NULL) {
        // Validar que la nueva persona no tenga ya un padre asignado
        cout << "Error: Esta persona ya tiene un padre asignado (" << nueva->padre->id << ").\n";
        return false;
    }
    if (esAncestro(nueva, padre)) {
        // Prevenir ciclos: no se puede insertar un ancestro como hijo de su descendiente
        cout << "Error: No se puede insertar (crear a un ciclo).\n";
        return false;
    }
    nueva->padre = padre;                         // Asignar el padre
    if (padre->izq == NULL) {
        padre->izq = nueva;                       // Insertar como hijo izquierdo si está disponible
        return true;
    } else if (padre->der == NULL) {
        padre->der = nueva;                       // Insertar como hijo derecho si está disponible
        return true;
    } else {
        cout << "Error: El padre ya tiene dos hijos.\n"; // Si ya tiene dos hijos, no se puede insertar
        return false;
    }
}
```

1.2.4. Recorridos del árbol

```cpp
// Recorrido en preorden: visita raíz, luego hijo izquierdo y luego derecho
void preorden(Nodo* nodo) {
    if (nodo != NULL) {
        cout << nodo->nombre << " " << nodo->apellido << " (" << nodo->id << ")\n";
        preorden(nodo->izq);  // Visitar hijo izquierdo
        preorden(nodo->der);  // Visitar hijo derecho
    }
}

// Recorrido en inorden: hijo izquierdo, luego raíz, luego derecho
void inorden(Nodo* nodo) {
    if (nodo != NULL) {
        inorden(nodo->izq);   // Visitar hijo izquierdo
        cout << nodo->nombre << " " << nodo->apellido << " (" << nodo->id << ")\n";
        inorden(nodo->der);   // Visitar hijo derecho
    }
}

// Recorrido en postorden: hijo izquierdo, hijo derecho, luego raíz
void postorden(Nodo* nodo) {
    if (nodo != NULL) {
        postorden(nodo->izq);  // Visitar hijo izquierdo
        postorden(nodo->der);  // Visitar hijo derecho
        cout << nodo->nombre << " " << nodo->apellido << " (" << nodo->id << ")\n";
    }
}
```

1.2.5. Mostrar ascendencia detallada

```cpp
void mostrarAscendenciaDetallada(Nodo* persona) {
    if (persona == NULL) {
        cout << "Persona no encontrada.\n"; // Verifica si la persona existe
        return;
    }

    cout << "\nPersona encontrada: " << persona->nombre << " " << persona->apellido << " (" << persona->id << ")\n";
    Nodo* camino[100];   // Arreglo para guardar la ruta hasta la raíz
    int nivel = 0;
    Nodo* actual = persona;
    while (actual != NULL) {
        camino[nivel++] = actual; // Guarda cada nodo del camino hacia arriba
        actual = actual->padre;
    }

    // Recorre el camino desde la raíz hasta la persona buscada
    for (int i = nivel - 1; i >= 0; i--) {
        actual = camino[i];
        cout << "\nNivel " << (nivel - i) << ":\n  ";

        if (actual->padre != NULL) {
            Nodo* hermano = NULL;

            // Determina si existe un hermano (el otro hijo del padre)
            if (actual->padre->izq == actual && actual->padre->der != NULL)
                hermano = actual->padre->der;
            else if (actual->padre->der == actual && actual->padre->izq != NULL)
                hermano = actual->padre->izq;

            // Imprime la información de la persona en ese nivel
            cout << actual->nombre << " " << actual->apellido << " (" << actual->id;
            if (i == nivel - 1) cout << ") (Raiz)";       // Es la raíz
            else if (i == 0) cout << ") (buscado)";       // Es la persona buscada
            else cout << ") (padre)";                     // Es un ancestro intermedio

            // Si hay hermano, mostrar su información también
            if (hermano != NULL) {
                cout << "    " << hermano->nombre << " " << hermano->apellido << " (" << hermano->id;
                if (i == 0) cout << ") (hermano)";        // Hermano del nodo buscado
                else cout << ")";                         // Hermano de un ancestro
            } else {
                cout << "\n"; // Nueva línea si no hay hermano
            }
        } else {
            // Es la raíz del árbol (no tiene padre)
            cout << actual->nombre << " " << actual->apellido << " (" << actual->id << ") (Raiz)\n";
        }
    }
    cout << "\n";
```

1.3. Operaciones del sistema

1.3.1. Añadir persona

```cpp
// Permite añadir una nueva persona al árbol
void anadirPersona(Nodo*& raiz) {
    Nodo* nueva = new Nodo();  // Crear nuevo nodo

    cout << "ID unico: ";
    getline(cin, nueva->id);

    // Verificar si ya existe una persona con ese ID
    if (raiz != NULL && buscarPorID(raiz, nueva->id) != NULL) {
        cout << "Error: Ya existe una persona con ese ID.\n";
        delete nueva; // Liberar memoria si ya existe
        return;
    }

    // Ingresar nombre y apellido
    cout << "Nombre: ";
    getline(cin, nueva->nombre);
    cout << "Apellido: ";
    getline(cin, nueva->apellido);

    // Inicializar punteros
    nueva->izq = nueva->der = nueva->padre = NULL;

    if (raiz == NULL) {
        raiz = nueva; // Si el árbol está vacío, esta persona es la raíz
        cout << "Anadido como raiz del arbol.\n";
    } else {
        // Solicitar ID del padre y tratar de insertar
        string padreID;
        cout << "ID del padre: ";
        getline(cin, padreID);
        if (!insertarPersona(raiz, nueva, padreID)) {
            delete nueva; // Si falla, liberar memoria
        } else {
            cout << "Persona anadida al arbol.\n";
        }
    }
}

```

1.3.2. Eliminar persona

```cpp
// Permite eliminar una persona del árbol si no tiene hijos
void eliminarPersona(Nodo*& raiz, const string& id) {
    Nodo* objetivo = buscarPorID(raiz, id); // Buscar a la persona
    if (objetivo == NULL) {
        cout << "Error: Persona no encontrada.\n";
        return;
    }

    // No se permite eliminar personas con hijos
    if (objetivo->izq != NULL || objetivo->der != NULL) {
        cout << "Error: No se puede eliminar. Esta persona tiene hijos.\n";
        return;
    }

    // Si es la raíz y no tiene hijos, eliminarla y vaciar el árbol
    if (objetivo == raiz) {
        delete raiz;
        raiz = NULL;
        cout << "Raiz eliminada. Arbol vacio.\n";
        return;
    }

    // Si no es la raíz, eliminarla del padre
    Nodo* padre = objetivo->padre;
    if (padre->izq == objetivo)
        padre->izq = NULL;
    else if (padre->der == objetivo)
        padre->der = NULL;

    delete objetivo; // Liberar memoria del nodo
    cout << "Persona eliminada correctamente.\n";
}
```

1.3.3. Submenú de recorridos

```cpp
void submenuMetodosBusqueda(Nodo* raiz) {
    int subopcion;
    do {
        cout << "\n--- METODOS DE BUSQUEDA ---\n";
        cout << "[1]: Mostrar Preorden\n";
        cout << "[2]: Mostrar Inorden\n";
        cout << "[3]: Mostrar Postorden\n";
        cout << "[4]: Volver al menu principal\n";

        // Validar que se elija una opción válida
        do {
            cout << "Elegir opcion: ";
            cin >> subopcion;
            if (subopcion < 1 || subopcion > 4) {
                cout << "ERROR, opcion incorrecta.\n";
            }
        } while (subopcion < 1 || subopcion > 4);

        cin.ignore(); // Limpiar buffer de entrada

        switch (subopcion) {
            case 1:
                // Mostrar recorrido preorden
                cout << "\n-- Recorrido Preorden --\n";
                preorden(raiz);
                break;
            case 2:
                // Mostrar recorrido inorden
                cout << "\n-- Recorrido Inorden --\n";
                inorden(raiz);
                break;
            case 3:
                // Mostrar recorrido postorden
                cout << "\n-- Recorrido Postorden --\n";
                postorden(raiz);
                break;
            case 4:
                // Volver al menú principal
                cout << "Regresando al menu principal...\n";
                break;
        }

        // Pausar y limpiar pantalla si no se sale
        if (subopcion != 4) {
            system("pause");
            system("cls");
        }

    } while (subopcion != 4);
```

1.3.4. Buscar por ID y mostrar ascendencia

```
// Permite buscar una persona por su ID y mostrar su ascendencia detallada
void buscarPersonaYAscendencia(Nodo* raiz) {
    string id;
    cout << "Ingrese el ID de la persona: ";
    getline(cin, id); // Leer el ID de la persona
    Nodo* persona = buscarPorID(raiz, id); // Buscar persona
    mostrarAscendenciaDetallada(persona);  // Mostrar su ascendencia
}
```

1.4. Menú y ejecución principal

1.4.1. Menú principal

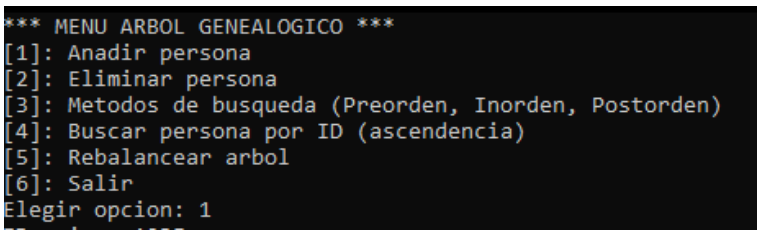
```
void menu() {
    Nodo* raiz = NULL; // Árbol inicialmente vacío
    int op;

    do {
        // Mostrar opciones del menú principal
        cout << "*** MENU ARBOL GENEALOGICO ***\n";
        cout << "[1]: Anadir persona\n";
        cout << "[2]: Eliminar persona\n";
        cout << "[3]: Metodos de busqueda (Preorden, Inorden, Postorden)\n";
        cout << "[4]: Buscar persona por ID (ascendencia)\n";
        cout << "[5]: Salir\n";

        // Validar opción ingresada
        do {
            cout << "Elegir opcion: ";
            cin >> op;
            if (op < 1 || op > 5) {
                cout << "ERROR, opcion incorrecta\n";
            }
        } while (op < 1 || op > 5);

        cin.ignore(); // Limpiar buffer

        // Ejecutar opción elegida
        switch (op) {
            case 1:
                // Añadir nueva persona al árbol
                anadirPersona(raiz);
                break;
            case 2: {
                // Eliminar persona del árbol
                string id;
                cout << "ID de la persona a eliminar: ";
                getline(cin, id);
                eliminarPersona(raiz, id);
                break;
            }
            case 3:
                // Acceder a submenú de métodos de recorrido
                submenuMetodosBusqueda(raiz);
                break;
            case 4:
                // Buscar persona y mostrar su ascendencia
                buscarPersonaYAscendencia(raiz);
                break;
            case 5:
                // Salir del programa
                break;
        }

        // Pausar y limpiar pantalla si no se ha elegido salir
        if (op != 5) {
            system("pause");
            system("cls");
        }

    } while (op != 5);
}
```

1.4.2. Función main

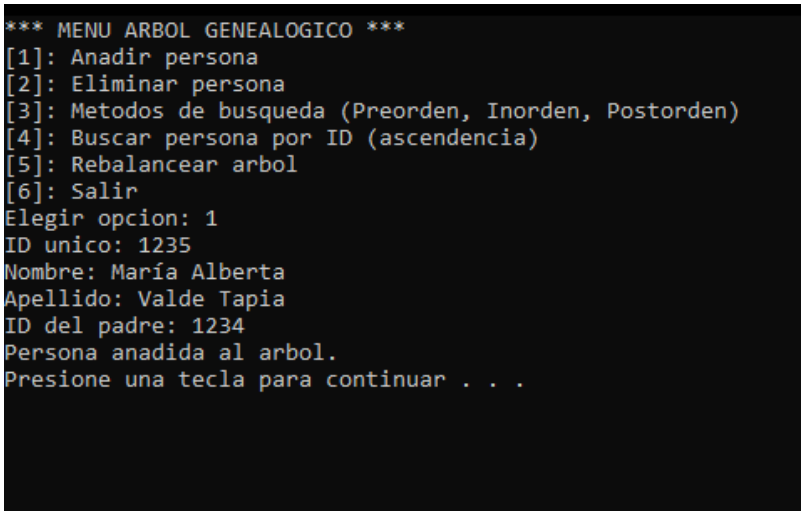
```
// Función principal del programa
int main() {
    setlocale(LC_ALL, "Spanish"); // Configura la codificación para admitir caracteres en español
    menu();                       // Llama al menú principal
    return 0;                     // Fin del programa
}
```

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

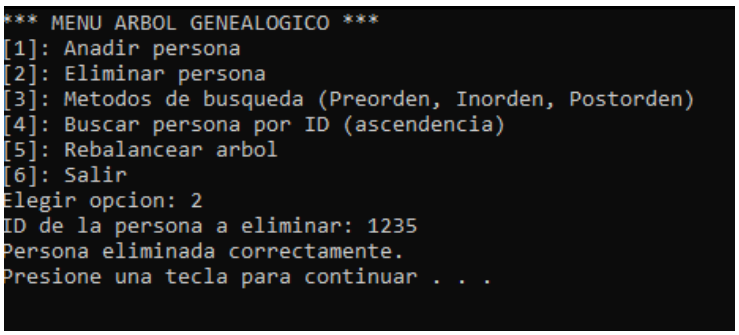
2.1. Menú



2.2. Añadir Persona



2.3. Eliminar Persona



2.4. Métodos de búsqueda

```
*** MENU ARBOL GENEALOGICO ***
[1]: Anadir persona
[2]: Eliminar persona
[3]: Metodos de busqueda (Preorden, Inorden, Postorden)
[4]: Buscar persona por ID (ascendencia)
[5]: Rebalancear arbol
[6]: Salir
Elegir opcion: 3

--- METODOS DE BUSQUEDA ---
[1]: Mostrar Preorden
[2]: Mostrar Inorden
[3]: Mostrar Postorden
[4]: Volver al menu principal
Elegir opcion: 1

-- Recorrido Preorden --
Juan Eduardo Rojas Rivas (1234)
Juan  Ramirez (1236)
Pedro Quillatupa (1239)
Presione una tecla para continuar . . .
```

```
--- METODOS DE BUSQUEDA ---
[1]: Mostrar Preorden
[2]: Mostrar Inorden
[3]: Mostrar Postorden
[4]: Volver al menu principal
Elegir opcion: 2

-- Recorrido Inorden --
Juan  Ramirez (1236)
Juan Eduardo Rojas Rivas (1234)
Pedro Quillatupa (1239)
Presione una tecla para continuar . . .
```

```
--- METODOS DE BUSQUEDA ---
[1]: Mostrar Preorden
[2]: Mostrar Inorden
[3]: Mostrar Postorden
[4]: Volver al menu principal
Elegir opcion: 3

-- Recorrido Postorden --
Juan  Ramirez (1236)
Pedro Quillatupa (1239)
Juan Eduardo Rojas Rivas (1234)
Presione una tecla para continuar . . .
```

2.5. Buscar por ID

```
*** MENU ARBOL GENEALOGICO ***
[1]: Anadir persona
[2]: Eliminar persona
[3]: Metodos de busqueda (Preorden, Inorden, Postorden)
[4]: Buscar persona por ID (ascendencia)
[5]: Rebalancear arbol
[6]: Salir
Elegir opcion: 4
Ingrese el ID de la persona: 1234

Persona encontrada: Juan Eduardo Rojas Rivas (1234)

Nivel 1:
  Juan Eduardo Rojas Rivas (1234) (Raiz)

Presione una tecla para continuar . . .
```

2.6. Rebalancear árbol

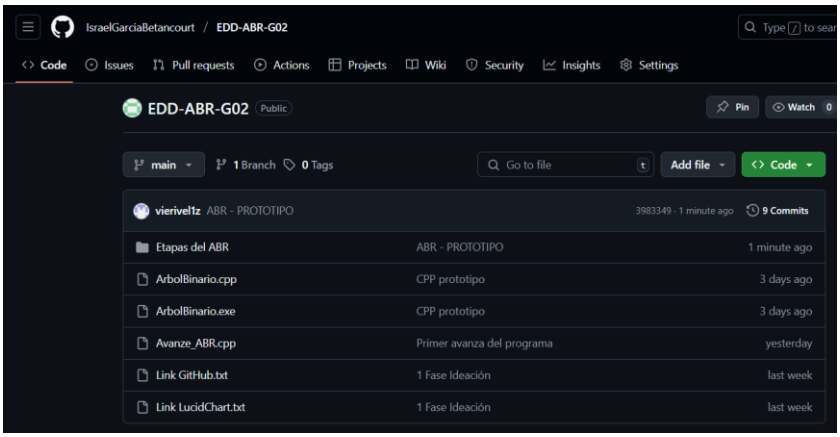
```
*** MENU ARBOL GENEALOGICO ***
[1]: Anadir persona
[2]: Eliminar persona
[3]: Metodos de busqueda (Preorden, Inorden, Postorden)
[4]: Buscar persona por ID (ascendencia)
[5]: Rebalancear arbol
[6]: Salir
Elegir opcion: 5
Arbol rebalanceado correctamente.
Presione una tecla para continuar . . .
```

2.7. Salir

```
*** MENU ARBOL GENEALOGICO ***
[1]: Anadir persona
[2]: Eliminar persona
[3]: Metodos de busqueda (Preorden, Inorden, Postorden)
[4]: Buscar persona por ID (ascendencia)
[5]: Rebalancear arbol
[6]: Salir
Elegir opcion: 6

--------------------------------
Process exited after 459.5 seconds with return value 0
Presione una tecla para continuar . . .
```

Capítulo 4 – Evidencias del trabajo colaborativo

1. Repositorio con Control de Versiones (Capturas de Pantalla)



Conclusiones

- La implementación del árbol genealógico mediante estructuras de datos tipo árbol binario de búsqueda permitió representar de forma clara y ordenada la información de una civilización ficticia. Este modelo estructural resultó ser altamente eficiente para reflejar jerarquías familiares, relaciones ascendentes y descendentes, y facilitar búsquedas dentro del sistema. La correcta selección de esta estructura no solo optimizó el acceso a los datos, sino que también proporcionó un soporte teórico robusto basado en los principios de organización y eficiencia algorítmica.

- A lo largo del desarrollo del sistema, se validaron y aplicaron operaciones fundamentales como la inserción y eliminación de nodos, los distintos tipos de recorridos (preorden, inorden, postorden) y la verificación de relaciones familiares. Estas operaciones fueron cuidadosamente diseñadas para garantizar la integridad del árbol genealógico y evitar conflictos como ciclos o datos duplicados. Asimismo, se logró que el sistema mantenga una coherencia lógica y funcional, incluso en escenarios de expansión o actualización de datos.

- El proyecto también reflejó la importancia del trabajo colaborativo, ya que a través de la división de responsabilidades, el uso de herramientas digitales como Lucidspark y GitHub, y una comunicación constante, el grupo logró construir una solución funcional, bien estructurada y alineada con los requerimientos del curso. Esta experiencia fortaleció las habilidades técnicas y blandas del equipo, resaltando el valor de la planificación, la documentación y la programación orientada a problemas reales.

- Este trabajo no solo permitió aplicar conocimientos teóricos en un contexto práctico, sino que también demostró cómo la informática puede ser una herramienta clave para preservar, organizar y analizar información histórica o cultural. La estructura desarrollada puede adaptarse a otros contextos donde se necesite representar relaciones complejas, mostrando la versatilidad y el alcance del uso de estructuras de datos en el desarrollo de soluciones tecnológicas.

Referencias

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.

Knuth, D. E. (1997). *The Art of Computer Programming: Volume 1 - Fundamental Algorithms* (3rd ed.). Addison-Wesley.

Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern Information Retrieval: The Concepts and Technology behind Search* (2nd ed.). Addison-Wesley.

Tavares, A. H., & Silva, J. P. (2017). Using Binary Trees for Genealogical Applications. *Procedia Computer Science, 121*, 410–417. https://doi.org/10.1016/j.procs.2017.11.056

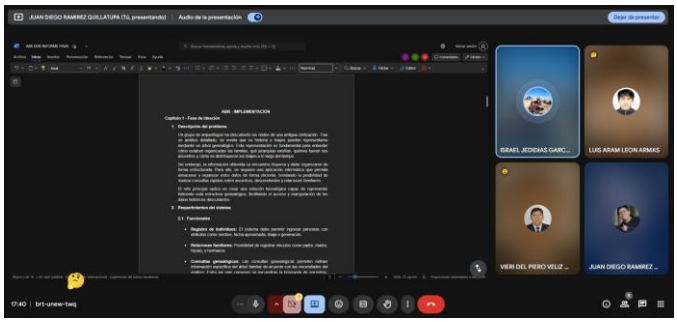
Silberschatz, A., Galvin, P. B., & Gagne, G. (2020). *Operating System Concepts* (10th ed.). Wiley.

Pressman, R. S., & Maxim, B. R. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill.

Shneiderman, B., Plaisant, C., Cohen, M., Jacobs, S., & Elmqvist, N. (2016). *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (6th ed.). Pearson.

Anexos

Anexo 1

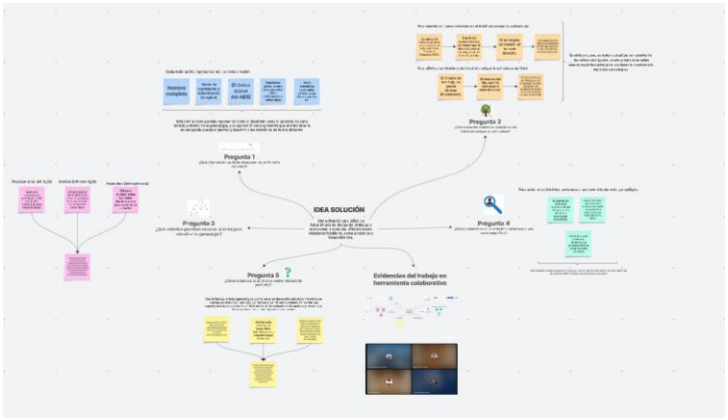


En esta imagen se observa la reunión que tuvo el grupo, para coordinar el ABR - Implementación

Anexo 2

En esta imagen se observa la coordinación en el aula de clases para coordinar el ABR – Implementación

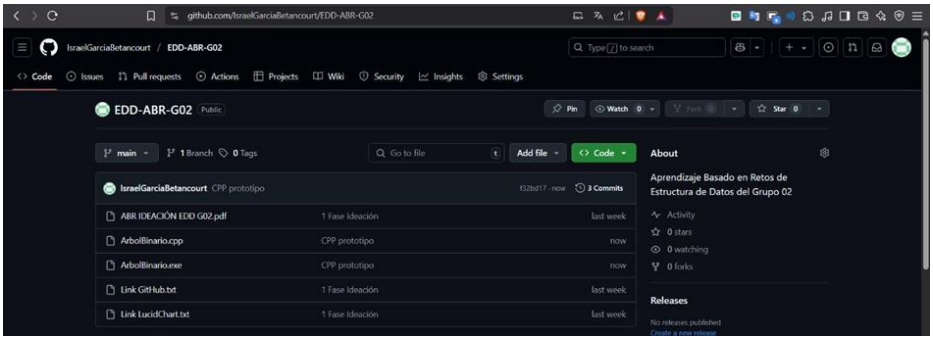
Anexo 3



En esta imagen se observa el material donde se trabajo la ideación incluyendo las preguntas guía, para efectuar la aclaración se añade el link del Lucidspark

https://lucid.app/lucidspark/f48f7979-c237-4677-939e-445a71f2edaf/edit?invitationId=inv_31a41481-1f90-4bcd-a7f2-bbc45b4195e6&page=0_0#

Anexo 4



En esta imagen se observa el repositorio donde se almacenará todos los archivos correspondientes al ABR

https://github.com/IsraelGarciaBetancourt/EDD-ABR-G02

Anexo 5

En esta imagen se observa el link de la herramienta Canva, donde se realizó las diapositivas respectivas
https://www.canva.com/design/DAGrT8oGiaU/pF9jufZwmNa0fBonk9zCrg/edit?utm_content=DAGrT8oGiaU&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton