MANUAL DE USUARIO

KEVIN ANDRES ALVAREZ HERRERA

202203038

ESTRUCTURA DE DATOS

SECCION C

Social Structure

INDICE

Objetivos

Específicos

- Crear una red social para facilitar la interacción entre los usuarios a pesar de la distancia.
- Permitir la interacción de los usuarios y publicaciones realizadas por ellos y sus amigos.
- Desarrollar una plataforma de red social que permita a los usuarios conectarse y comunicarse con amigos y nuevos contactos, independientemente de su ubicación geográfica.

Generales

- Implementar funcionalidades que permitan a los usuarios realizar publicaciones, compartir contenido y participar en interacciones con otros usuarios de su red de contactos.
- Crear un sistema de gestión de usuarios que facilite el registro, autenticación y administración de perfiles, así como la búsqueda y conexión con otros usuarios.
- Diseñar mecanismos que promuevan la privacidad y seguridad de los usuarios, asegurando que la información

Especificación técnica

Requisitos de hardware

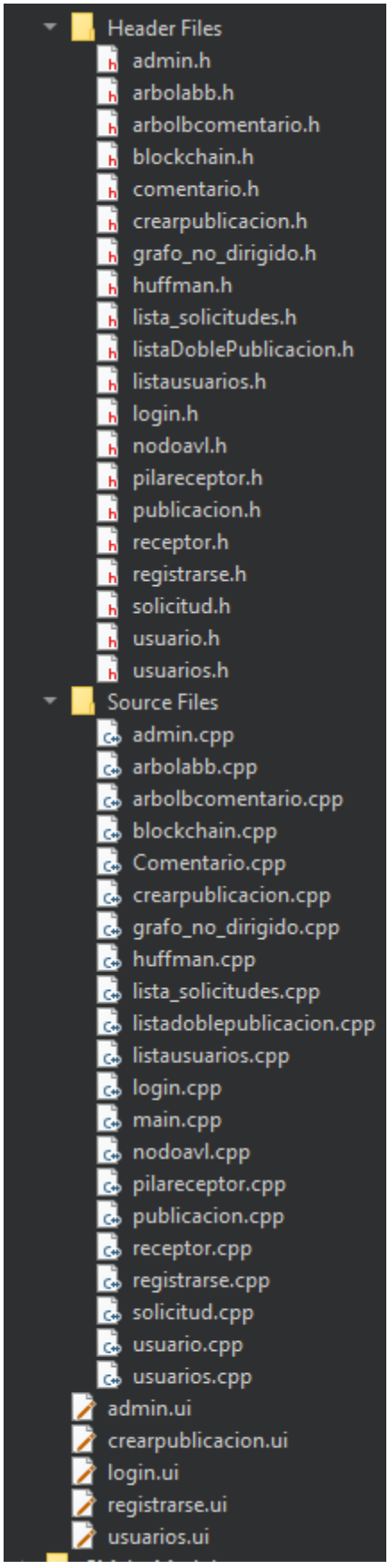
- Monitor
- Mouse
- Teclado
- CPU
- RAM

Memoria de almacenamiento

- Requisitos de software
- Sistema operativo Windows 10 o 11
- La herramienta Qt
- La última versión de c++ (13.2.0)

Lógica para la realización del programa

Clases utilizadas


Header Files
admin.h
arbolabb.h
arbolbcomentario.h
blockchain.h
comentario.h
crearpublicacion.h
grafo_no_dirigido.h
huffman.h
lista_solicitudes.h
listaDoblePublicacion.h
listausuarios.h
login.h
nodoavl.h
pilareceptor.h
publicacion.h
receptor.h
registrarse.h
solicitud.h
usuario.h
usuarios.h
Source Files
admin.cpp
arbolabb.cpp
arbolbcomentario.cpp
blockchain.cpp
Comentario.cpp
crearpublicacion.cpp
grafo_no_dirigido.cpp
huffman.cpp
lista_solicitudes.cpp
listadoblepublicacion.cpp
listausuarios.cpp
login.cpp
main.cpp
nodoavl.cpp
pilareceptor.cpp
publicacion.cpp
receptor.cpp
registrarse.cpp
solicitud.cpp
usuario.cpp
usuarios.cpp
admin.ui
crearpublicacion.ui
login.ui
registrarse.ui
usuarios.ui

Estructuras utilizadas

- Listas enlazadas
- Listas doblemente enlazadas
- Grafo no dirigido
- Lista de adyacencia
- Blockchain
- Árbol AVL
- Árbol BTree
- Árbol B de orden 5
- Pilas
- Colas

Explicación del codigo

Formato de los “.jsons”

Usuarios

- Nombres
- Apellidos
- Fecha_de_nacimiento
- Correo
- Contraseña

Solicitudes

- Emisor
- Receptor
- Estado

Publicaciones

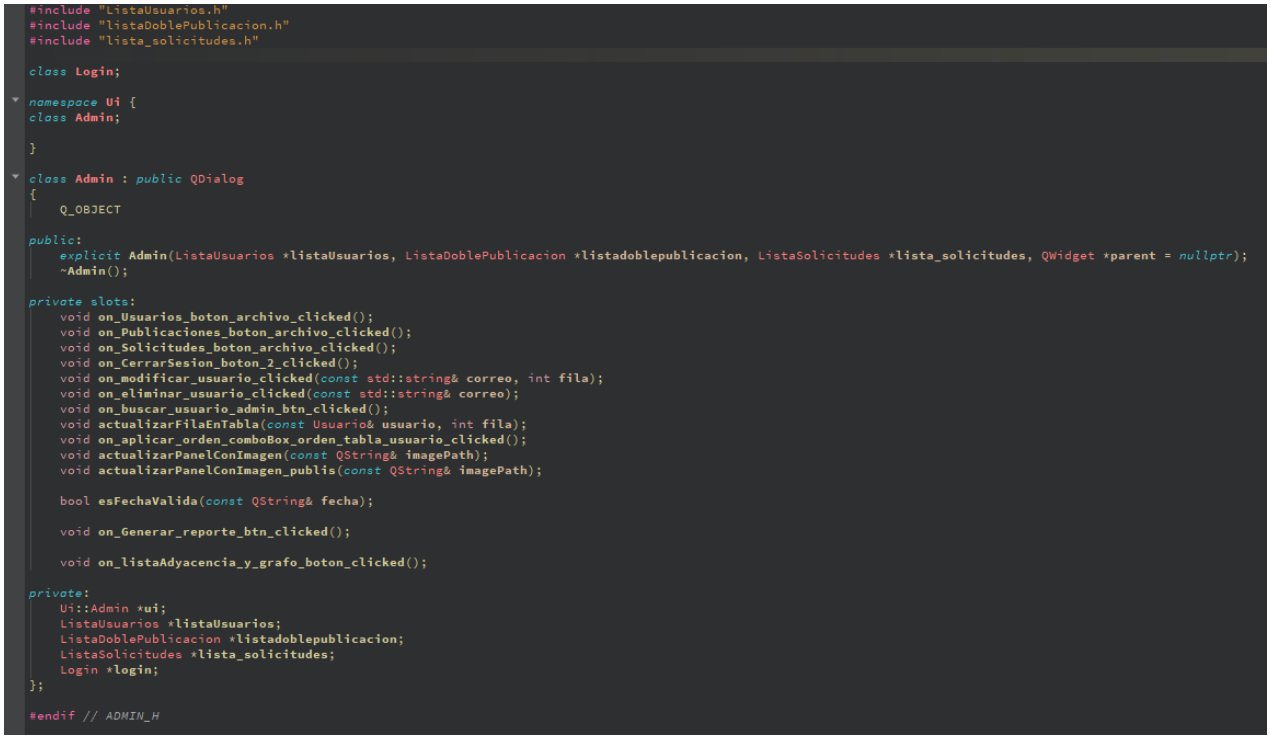
- Correo

- Contenido
- Fecha
- Hora
- Comentarios (este es una lista que contiene todos los comentarios de dicha publicacion)

Atributos de comentarios

- Correo
- Comentario
- Fecha
- Hora

Admin



```
#include "ListaUsuarios.h"
#include "listaDoblePublicacion.h"
#include "lista_solicitudes.h"

class Login;

namespace Ui {
class Admin;

}

class Admin : public QDialog
{
    Q_OBJECT

public:
    explicit Admin(ListaUsuarios *listaUsuarios, ListaDoblePublicacion *listadoblepublicacion, ListaSolicitudes *lista_solicitudes, QWidget *parent = nullptr);
    ~Admin();

private slots:
    void on_Usuarios_boton_archivo_clicked();
    void on_Publicaciones_boton_archivo_clicked();
    void on_Solicitudes_boton_archivo_clicked();
    void on_CerrarSesion_boton_2_clicked();
    void on_modificar_usuario_clicked(const std::string& correo, int fila);
    void on_eliminar_usuario_clicked(const std::string& correo);
    void on_buscar_usuario_admin_btn_clicked();
    void actualizarFilaEnTabla(const Usuario& usuario, int fila);
    void on_aplicar_orden_comboBox_orden_tabla_usuario_clicked();
    void actualizarPanelConImagen(const QString& imagePath);
    void actualizarPanelConImagen_publis(const QString& imagePath);

    bool esFechaValida(const QString& fecha);

    void on_Generar_reporte_btn_clicked();

    void on_listaAdyacencia_y_grafo_boton_clicked();

private:
    Ui::Admin *ui;
    ListaUsuarios *listaUsuarios;
    ListaDoblePublicacion *listadoblepublicacion;
    ListaSolicitudes *lista_solicitudes;
    Login *login;
};

#endif // ADMIN_H
```

Atributos:

- listaUsuarios: Apunta a una lista de usuarios (ListaUsuarios), que probablemente almacena los usuarios de la aplicación.
- listadoblepublicacion: Apunta a una lista doblemente enlazada de publicaciones (ListaDoblePublicacion), que almacena publicaciones de los usuarios.

- lista_solicitudes: Apunta a una lista de solicitudes de amistad (ListaSolicitudes).
- login: Un puntero a un objeto de la clase Login, que posiblemente gestiona el inicio de sesión.

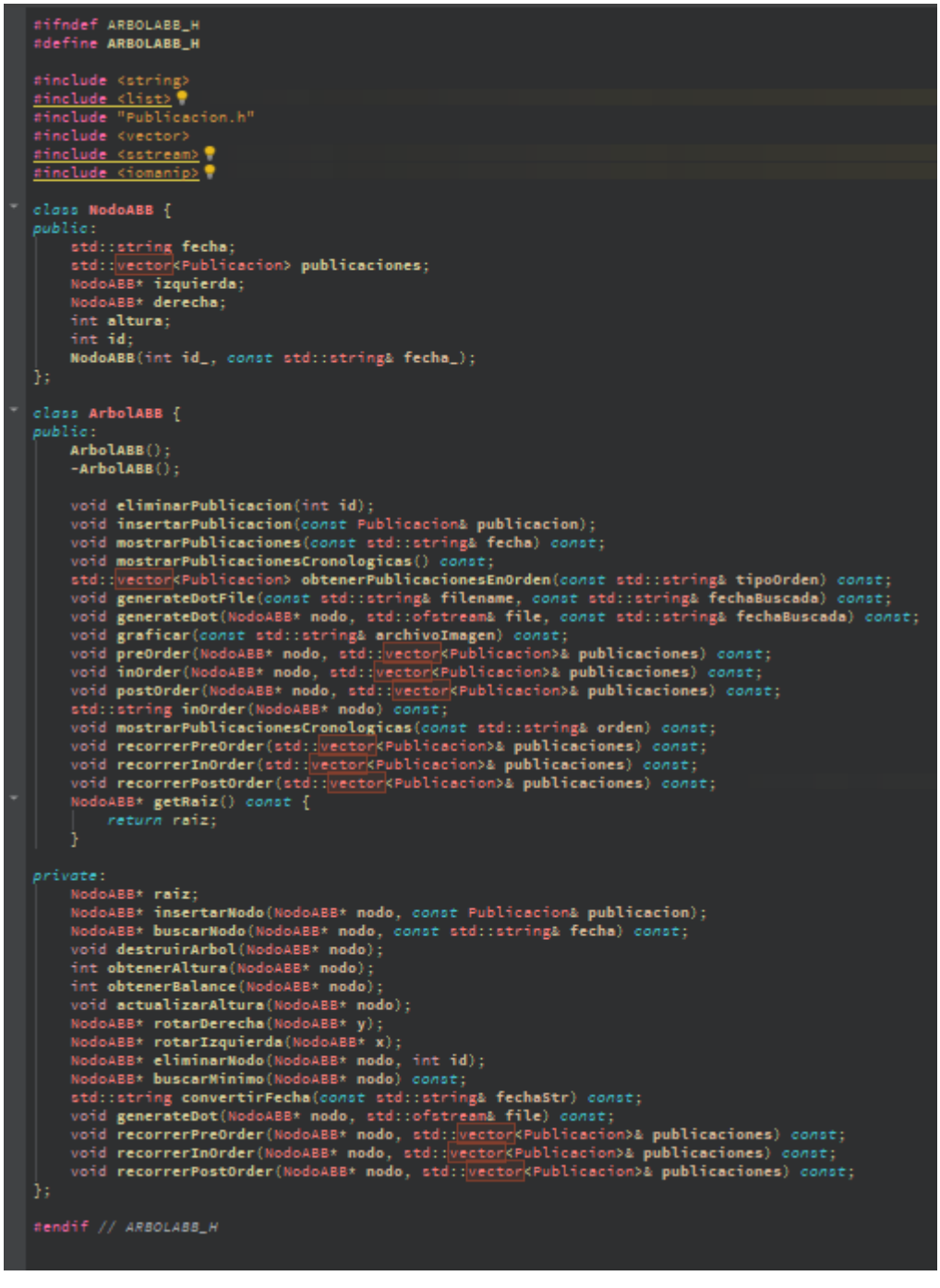
Constructor y Destructor:

- El constructor inicializa el diálogo de administración con listas de usuarios, publicaciones y solicitudes.
- El destructor se asegura de liberar los recursos asignados.

Slots (funciones relacionadas con la interfaz):

- **on_Usuarios_boton_archivo_clicked()**: Se ejecuta cuando el botón asociado a usuarios es clicado, probablemente para gestionar la carga o guardado de usuarios en un archivo.
- **on_Publicaciones_boton_archivo_clicked()**: Se ejecuta cuando el botón relacionado con publicaciones es clicado.
- **on_Solicitudes_boton_archivo_clicked()**: Similar a las anteriores, pero para las solicitudes.
- **on_CerrarSesion_boton_2_clicked()**: Maneja la acción de cerrar sesión.
- **on_modificar_usuario_clicked(const std::string& correo, int fila)**: Permite modificar un usuario identificado por su correo y fila en la tabla.
- **on_eliminar_usuario_clicked(const std::string& correo)**: Elimina un usuario específico usando su correo.
- **on_buscar_usuario_admin_btn_clicked()**: Busca un usuario.
- **actualizarFilaEnTabla(const Usuario& usuario, int fila)**: Actualiza los datos de un usuario en la tabla de administración.
- **on_aplicar_orden_comboBox_orden_tabla_usuario_clicked()**: Ordena los usuarios en la tabla según un criterio seleccionado.
- **actualizarPanelConImagen(const QString& imagePath)**: Actualiza un panel con una imagen, posiblemente del usuario.
- **actualizarPanelConImagen_publis(const QString& imagePath)**: Similar al anterior, pero para publicaciones.
- **esFechaValida(const QString& fecha)**: Valida una fecha en formato QString.
- **on_Generar_reporte_btn_clicked()**: Genera un reporte de algún tipo (usuarios, publicaciones, etc.).
- **on_listaAdyacencia_y_grafo_boton_clicked()**: Probablemente relacionado con la visualización de un grafo o lista de adyacencia, posiblemente de relaciones entre usuarios o publicaciones

ArbolABB



```
#ifndef ARBOLABB_H
#define ARBOLABB_H

#include <string>
#include <list>
#include "Publicacion.h"
#include <vector>
#include <sstream>
#include <iomanip>

class NodoABB {
public:
    std::string fecha;
    std::vector<Publicacion> publicaciones;
    NodoABB* izquierda;
    NodoABB* derecha;
    int altura;
    int id;
    NodoABB(int id_, const std::string& fecha_);
};

class ArbolABB {
public:
    ArbolABB();
    ~ArbolABB();

    void eliminarPublicacion(int id);
    void insertarPublicacion(const Publicacion& publicacion);
    void mostrarPublicaciones(const std::string& fecha) const;
    void mostrarPublicacionesCronologicas() const;
    std::vector<Publicacion> obtenerPublicacionesEnOrden(const std::string& tipoOrden) const;
    void generateDotFile(const std::string& filename, const std::string& fechaBuscada) const;
    void generateDot(NodoABB* nodo, std::ofstream& file, const std::string& fechaBuscada) const;
    void graficar(const std::string& archivoImagen) const;
    void preOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;
    void inOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;
    void postOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;
    std::string inOrder(NodoABB* nodo) const;
    void mostrarPublicacionesCronologicas(const std::string& orden) const;
    void recorrerPreOrder(std::vector<Publicacion>& publicaciones) const;
    void recorrerInOrder(std::vector<Publicacion>& publicaciones) const;
    void recorrerPostOrder(std::vector<Publicacion>& publicaciones) const;
    NodoABB* getRaiz() const {
        return raiz;
    }

private:
    NodoABB* raiz;
    NodoABB* insertarNodo(NodoABB* nodo, const Publicacion& publicacion);
    NodoABB* buscarNodo(NodoABB* nodo, const std::string& fecha) const;
    void destruirArbol(NodoABB* nodo);
    int obtenerAltura(NodoABB* nodo);
    int obtenerBalance(NodoABB* nodo);
    void actualizarAltura(NodoABB* nodo);
    NodoABB* rotarDerecha(NodoABB* y);
    NodoABB* rotarIzquierda(NodoABB* x);
    NodoABB* eliminarNodo(NodoABB* nodo, int id);
    NodoABB* buscarMinimo(NodoABB* nodo) const;
    std::string convertirFecha(const std::string& fechaStr) const;
    void generateDot(NodoABB* nodo, std::ofstream& file) const;
    void recorrerPreOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;
    void recorrerInOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;
    void recorrerPostOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;
};

#endif // ARBOLABB_H
```

Este código define una estructura y una clase para implementar un Árbol Binario de Búsqueda (ABB) que organiza y gestiona publicaciones según sus fechas.

NodoABB: Representa un nodo del árbol con una fecha y una lista de publicaciones. Incluye punteros a los nodos izquierdo y derecho, y métodos para insertar un nodo y mostrar publicaciones en orden.

ArbolABB: Administra el árbol binario. Contiene métodos para:

Insertar publicaciones por fecha.

- Mostrar publicaciones para una fecha específica o en orden cronológico.
- Obtener publicaciones en un orden específico (preorden, inorden o postorden).
- Generar un archivo DOT para visualizar el árbol.
- Destruir el árbol para liberar memoria.

Comentario

```cpp
#ifndef COMENTARIO_H
#define COMENTARIO_H

#include <string>
#include "json.hpp"

class Comentario {
public:
    // Constructor
    Comentario() : contenido_(""), idPublicacion_(0), correo_(""), fecha_(""), hora_("") {}

    Comentario(const std::string& contenido, int idPublicacion, const std::string& correo, const std::string& fecha, const std::string& hora);

    // Getters
    std::string getCorreo() const;
    std::string getComentario() const;
    std::string getFecha() const;
    std::string getHora() const;
    int getIdPublicacion() const;
    bool esVacio() const {
        return contenido_.empty();
    }

    std::string toJSON() const {
        nlohmann::json jsonData;
        jsonData["correo"] = this->correo_;
        jsonData["contenido"] = this->contenido_;
        jsonData["fecha"] = this->fecha_;
        jsonData["hora"] = this->hora_;
        return jsonData.dump();
    }
private:
    std::string contenido_;
    std::string correo_;
    int idPublicacion_;
    std::string fecha_;
    std::string hora_;
};

#endif // COMENTARIO_H
```

Esta clase es la encargada de guardar todos los atributos relacionados con los comentarios a modo de objeto

Árbol B de orden 5

```
#ifndef ARBOL_B_COMENTARIO_H
#define ARBOL_B_COMENTARIO_H

#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include "comentario.h"
#include <map>
#include "json.hpp"

#define ORDEN 5

class NodoArbolBComentario {
public:
    bool esHoja;
    std::vector<Comentario> comentarios;
    std::vector<NodoArbolBComentario*> hijos;

    NodoArbolBComentario(bool hoja);

    void obtenerComentariosPorPublicacion(int idPublicacion, std::vector<Comentario>& comentarios) const;
    void insertarNoLleno(const Comentario& comentario);
    void dividirHijo(int indice, NodoArbolBComentario* nodoLleno);
    void recorrer() const;
    NodoArbolBComentario* buscar(const std::string& fecha, const std::string& hora)const;
    static bool compararComentarios(const Comentario& c1, const Comentario& c2);
    void eliminarComentariosPorCorreo(const std::string& correo);

};

// Clase que define el Árbol B de comentarios
class ArbolBComentario {
public:
    ArbolBComentario();
    void eliminarComentariosPorCorreo(const std::string& correo);
    void insertar(const Comentario& comentario);
    std::vector<Comentario> getComentariosDePublicacion(int idPublicacion) const;
    void mostrarComentarios() const;
    NodoArbolBComentario* buscar(const std::string& fecha, const std::string& hora)const;
    void graficarArbolB(const std::string& nombreArchivo, int idPublicacion);
    void graficarNodo(NodoArbolBComentario* nodo, int& nodeCount, std::map<NodoArbolBComentario*, int>& nodeMap, std::ofstream& archivo, int idPublicacion);
    std::vector<Comentario> getAllComentarios() const;


private:
    NodoArbolBComentario* raiz;
    void obtenerComentariosRecursivos(NodoArbolBComentario* nodo, int idPublicacion, std::vector<Comentario>& comentarios) const;
    void obtenerComentariosRecursivos(NodoArbolBComentario* nodo, std::vector<Comentario>& comentarios) const ;
};

#endif // ARBOL_B_COMENTARIO_H
```

Con los comentarios almacenados en el objeto Comentario asignados a ciertas publicaciones ingresan a este árbol B de orden 5 el cual tiene diferentes métodos que permiten el funcionamiento de este de una manera eficiente.

Atributos:

- esHoja: indica si el nodo es una hoja.
- raiz: puntero al nodo raíz del árbol.
- comentarios: vector que almacena los comentarios en el nodo.
- hijos: vector de punteros a los hijos del nodo.

Métodos:

- obtenerComentariosPorPublicacion: obtiene los comentarios de una publicación específica.
- insertarNoLleno: inserta un comentario en un nodo que no está lleno.
- dividirHijo: divide un nodo hijo cuando está lleno.

- recorrer: recorre y muestra todos los comentarios en el nodo.
- buscar: busca un comentario según la fecha y la hora.
- compararComentarios: función estática para comparar dos comentarios.
- insertar: inserta un comentario en el árbol.
- getComentariosDePublicacion: obtiene los comentarios de una publicación específica.
- mostrarComentarios: muestra todos los comentarios en el árbol.
- buscar: busca un comentario según la fecha y la hora.

- graficarArbolB: genera un gráfico del árbol B.
- graficarNodo: función auxiliar para graficar cada nodo.
- obtenerComentariosRecursivos: obtiene comentarios de una publicación específica de forma recursiva.

CrearPublicacion

```
#ifndef CREARPUBLICACION_H
#define CREARPUBLICACION_H

#include <QDialog>
#include "listaDoblePublicacion.h"

namespace Ui {
class CrearPublicacion;
}
class Usuarios;

class CrearPublicacion : public QDialog
{
    Q_OBJECT

public:
    explicit CrearPublicacion(std::string correoUsuario, ListaDoblePublicacion *listaDoblePublicacion, QWidget *parent = nullptr);
    ~CrearPublicacion();

private slots:
    void on_crearPublicacion_boton_clicked();
    void on_seleccionarImagen_boton_clicked();
    void on_cancelar_boton_clicked();

private:
    Ui::CrearPublicacion *ui;
    ListaDoblePublicacion *listaDoblePublicacion;
    std::string correoActualUsuario_;
    Usuarios *usuarioWindow;
    int idPublicacionActual_;
};

#endif // CREARPUBLICACION_H
```

Esta clase permite la creación de nuevas publicaciones por parte del usuario y agregarlas a la listaDoblePublicaciones

Lista_solicitudes

```cpp
#ifndef LISTA_SOLICITUDES_H
#define LISTA_SOLICITUDES_H

#include "solicitud.h"
#include <string>
#include <vector>
#include "grafo_no_dirigido.h"
#include "arbolabb.h"

extern GrafoNoDirigido grafoNoDirigido;

class ListaSolicitudes {
private:
    class NodoSolicitud {
    public:
        Solicitud solicitud;
        NodoSolicitud* siguiente;

        NodoSolicitud(const Solicitud &solicitud);
    };

    NodoSolicitud* cabeza;

public:
    ListaSolicitudes();
    ~ListaSolicitudes();
    bool existeSolicitudEnEstado(const std::string& emisor, const std::string& receptor, const std::string& estado) const;

    void agregarSolicitud(const Solicitud &solicitud);
    void eliminarSolicitud(const std::string &emisor, const std::string &receptor);
    void mostrarSolicitudes() const;
    void cargarRelacionesDesdeJson(const std::string &filename);
    void enviarSolicitud(const std::string &emisor, const std::string &receptor);
    std::vector<std::string> obtenerSolicitudesEnviadas(const std::string &correoEmisor) const;
    bool existeSolicitudEnEstado(const std::string &emisor, const std::string &receptor, const std::string &estado);
    std::vector<Solicitud> obtenerSolicitudesPorReceptor(const std::string &correoReceptor) const;
    void buscarYApilarPendientes(const std::string &correo, const ListaSolicitudes &listaSolicitudes);
    bool actualizarEstadoSolicitud(const std::string& emisor, const std::string& receptor, const std::string& nuevoEstado);
    void agregarRelacionesAceptadasAMatriz(GrafoNoDirigido &grafoNoDirigido);
    std::vector<Solicitud> getSolicitudes() const {
        std::vector<Solicitud> solicitudes; // Vector para almacenar las solicitudes
        NodoSolicitud* actual = cabeza; // Comenzamos desde la cabeza
        while (actual != nullptr) { // Recorremos la lista
            solicitudes.push_back(actual->solicitud); // Agregamos la solicitud al vector
            actual = actual->siguiente; // Pasamos al siguiente nodo
        }
        return solicitudes; // Retornamos todas las solicitudes
    }
    void guardarSolicitudesEnviadas() const;
};

#endif // LISTA_SOLICITUDES_H
```

NodoSolicitud: Clase interna que representa un nodo de la lista, almacenando una solicitud y un puntero al siguiente nodo.

ListaSolicitudes:

Administra la lista enlazada de solicitudes.

Métodos principales:

- agregarSolicitud: Añade una solicitud a la lista.
- eliminarSolicitud: Elimina una solicitud específica entre un emisor y un receptor.
- mostrarSolicitudes: Muestra todas las solicitudes.
- cargarRelacionesDesdeJson: Carga relaciones desde un archivo JSON.
- enviarSolicitud: Envia una solicitud de un emisor a un receptor.

- obtenerSolicitudesEnviadas: Obtiene las solicitudes enviadas por un usuario.
- existeSolicitudEnEstado: Verifica si existe una solicitud con un estado específico.
- obtenerSolicitudesPorReceptor: Recupera todas las solicitudes para un receptor.

- buscarYApilarPendientes: Apila solicitudes pendientes para un usuario.
- actualizarEstadoSolicitud: Cambia el estado de una solicitud.
- agregarRelacionesAceptadasAMatriz: Agrega relaciones aceptadas a un grafo no dirigido.

ListaDoblePublicacion

```
#ifndef LISTADOBLEPUBLICACION_H
#define LISTADOBLEPUBLICACION_H

#include "publicacion.h"
#include <string>
#include "arbolabb.h"
#include "arbolbcomentario.h"
#include "comentario.h"
#include "grafo_no_dirigido.h"
extern GrafoNoDirigido grafoNoDirigido;
extern ArbolABB arbolABB;
extern ArbolBComentario arbolComentarios_;


class ListaDoblePublicacion
{
private:
    class NodoPublicacion
    {
    public:
        Publicacion publicacion;
        NodoPublicacion *siguiente;
        NodoPublicacion *anterior;

        NodoPublicacion(const Publicacion &publicacion);
    };

    NodoPublicacion *cabeza;
    NodoPublicacion *cola;
    int siguienteId;

    void crearPNG(const std::string &dotFilename, const std::string &pngFilename);
    NodoABB* raiz;
    void preOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;
    void inOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;
    void postOrder(NodoABB* nodo, std::vector<Publicacion>& publicaciones) const;

public:
    ListaDoblePublicacion();
    ~ListaDoblePublicacion();
    void generateDot(const std::string &filename);
    void cargarPublicacionesDesdeJson(const std::string &filename);
    void agregarPublicacion(const Publicacion &publicacion);
    void mostrarPublicacion(const Publicacion& publicacion) const;
    void mostrarPublicacionesPorUsuario(const std::string& correo) const;
    void mostrarTodasLasPublicaciones() const;
    void mostrarPublicacionesPorCorreo(const std::string& correo) const;
    std::vector<Publicacion> mostrarPublicacionesYAmigos(const std::string &correo, const GrafoNoDirigido &grafoNoDirigido, ArbolABB &arbol, const std::string &orden)
    int obtenerNuevoId() const;
    void mostrarPublicacionesOrden(const std::string& correoUsuario, const GrafoNoDirigido& grafoNoDirigido, ArbolABB& arbolABB, int orden) const;
    std::vector<Publicacion> obtenerPublicacionesEnOrden(const std::string& tipoOrden) const;
    void eliminarPublicacionPorId(int id);
    void eliminarPublicacionesPorCorreo(const std::string& correo);
    void guardarPublicacionesEnBloques(const std::string& directory);

};

#endif // LISTADOBLEPUBLICACION_H
```

NodoPublicacion: Clase interna que representa un nodo de la lista, almacenando una publicación y punteros a los nodos siguiente y anterior.

ListaDoblePublicacion:

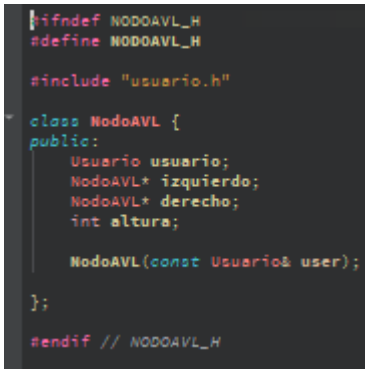
Estructura: Administra una lista doblemente enlazada con una cabeza, una cola y un contador de IDs para publicaciones.

Métodos principales:

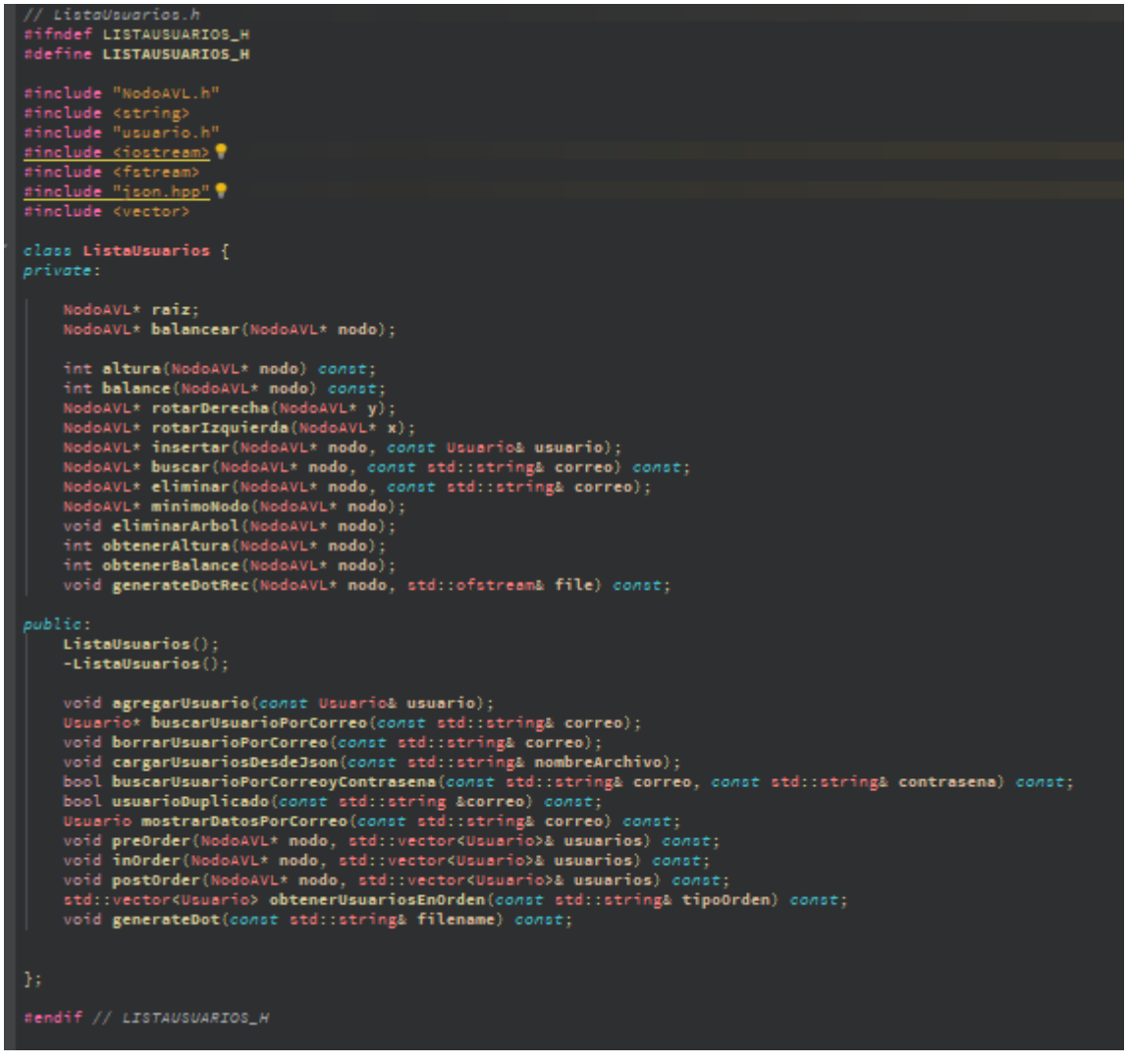
- agregarPublicacion: Añade una publicación a la lista.
- mostrarPublicacion: Muestra una publicación específica.
- mostrarPublicacionesPorUsuario: Muestra todas las publicaciones de un usuario.
- mostrarTodasLasPublicaciones: Muestra todas las publicaciones en la lista.
- mostrarPublicacionesYAmigos: Muestra publicaciones de un usuario y sus amigos utilizando una grafo no dirigido y un árbol.

- mostrarPublicacionesOrden: Muestra publicaciones en un orden específico (preorden, inorden, postorden).
- obtenerPublicacionesEnOrden: Recupera publicaciones ordenadas según un criterio.
- cargarPublicacionesDesdeJson: Carga publicaciones desde un archivo JSON.
- generateDot: Genera un archivo DOT para visualizar la lista.
- crearPNG: Convierte el archivo DOT en una imagen PNG.

ArbolAVL



```
#ifndef NODOAVL_H
#define NODOAVL_H

#include "usuario.h"

class NodoAVL {
public:
    Usuario usuario;
    NodoAVL* izquierdo;
    NodoAVL* derecho;
    int altura;

    NodoAVL(const Usuario& user);

};

#endif // NODOAVL_H
```



```h
// ListaUsuarios.h
#ifndef LISTAUSUARIOS_H
#define LISTAUSUARIOS_H

#include "NodoAVL.h"
#include <string>
#include "usuario.h"
#include <iostream>
#include <fstream>
#include "json.hpp"
#include <vector>

class ListaUsuarios {
private:

    NodoAVL* raiz;
    NodoAVL* balancear(NodoAVL* nodo);

    int altura(NodoAVL* nodo) const;
    int balance(NodoAVL* nodo) const;
    NodoAVL* rotarDerecha(NodoAVL* y);
    NodoAVL* rotarIzquierda(NodoAVL* x);
    NodoAVL* insertar(NodoAVL* nodo, const Usuario& usuario);
    NodoAVL* buscar(NodoAVL* nodo, const std::string& correo) const;
    NodoAVL* eliminar(NodoAVL* nodo, const std::string& correo);
    NodoAVL* minimoNodo(NodoAVL* nodo);
    void eliminarArbol(NodoAVL* nodo);
    int obtenerAltura(NodoAVL* nodo);
    int obtenerBalance(NodoAVL* nodo);
    void generateDotRec(NodoAVL* nodo, std::ofstream& file) const;

public:
    ListaUsuarios();
    ~ListaUsuarios();

    void agregarUsuario(const Usuario& usuario);
    Usuario* buscarUsuarioPorCorreo(const std::string& correo);
    void borrarUsuarioPorCorreo(const std::string& correo);
    void cargarUsuariosDesdeJson(const std::string& nombreArchivo);
    bool buscarUsuarioPorCorreoyContrasena(const std::string& correo, const std::string& contrasena) const;
    bool usuarioDuplicado(const std::string &correo) const;
    Usuario mostrarDatosPorCorreo(const std::string& correo) const;
    void preOrder(NodoAVL* nodo, std::vector<Usuario>& usuarios) const;
    void inOrder(NodoAVL* nodo, std::vector<Usuario>& usuarios) const;
    void postOrder(NodoAVL* nodo, std::vector<Usuario>& usuarios) const;
    std::vector<Usuario> obtenerUsuariosEnOrden(const std::string& tipoOrden) const;
    void generateDot(const std::string& filename) const;


};

#endif // LISTAUSUARIOS_H
```

NodoAVL: Se utiliza para almacenar los nodos del árbol AVL, cada uno contiene un usuario y sus datos.

ListaUsuarios:

Estructura: Utiliza un árbol AVL para almacenar usuarios, equilibrando el árbol automáticamente después de cada inserción o eliminación.

Métodos principales:

- agregarUsuario: Inserta un nuevo usuario en el árbol AVL.
- buscarUsuarioPorCorreo: Busca un usuario por su correo electrónico.
- borrarUsuarioPorCorreo: Elimina un usuario del árbol por su correo.
- cargarUsuariosDesdeJson: Carga usuarios desde un archivo JSON.

- buscarUsuarioPorCorreoyContrasena: Verifica si un usuario con correo y contraseña especificados existe.
- usuarioDuplicado: Comprueba si un correo ya está registrado.
- mostrarDatosPorCorreo: Muestra los datos de un usuario según su correo.
- preOrder, inOrder, postOrder: Recorridos del árbol para obtener usuarios en diferentes órdenes.
- obtenerUsuariosEnOrden: Retorna un vector de usuarios en un orden específico.
- generateDot: Genera un archivo DOT para visualizar el árbol.
- balancear y rotaciones: Métodos para equilibrar el árbol AVL.

Login

```h
#ifndef LOGIN_H
#define LOGIN_H

#include <QMainWindow>
#include "ListaUsuarios.h"
#include "listaDoblePublicacion.h"
#include "lista_solicitudes.h"

class Admin;
class Usuarios;
class Registrarse;

namespace Ui {
class Login;
}

class Login : public QMainWindow
{
    Q_OBJECT

public:
    explicit Login(ListaUsuarios *listaUsuarios, ListaDoblePublicacion *listadoblepublicacion, ListaSolicitudes *lista_solicitudes, QWidget *parent = nullptr);
    ~Login();
    void guardarUsuarios();
    void calcularFrecuencias(const std::string& nombreArchivo, int frecuencias[256]);
protected:
    void closeEvent(QCloseEvent *event) override;  // Sobrescribir el método closeEvent

private slots:
    void on_InicioSesion_btn_clicked();
    void on_Registrarse_btn_clicked();
signals:
    void cerrarSesion();
private:
    Ui::Login *ui;
    Admin *adminWindow;
    Usuarios *usuarioWindow;
    Registrarse *registrarseWindow;
    ListaUsuarios *listaUsuarios;
    ListaDoblePublicacion *listadoblepublicacion;
    ListaSolicitudes *lista_solicitudes;
};

#endif // LOGIN_H
```

Clase Login:

- Herencia: Deriva de QMainWindow, lo que la convierte en una ventana principal de la aplicación.

Objetos:

- listaUsuarios: Apunta a una instancia de la clase ListaUsuarios que gestiona los usuarios.

- listadoblepublicacion: Apunta a una instancia de la clase ListaDoblePublicacion, que gestiona publicaciones.

- lista_solicitudes: Apunta a una instancia de ListaSolicitudes, que maneja las solicitudes de amistad.
- adminWindow, usuarioWindow, registrarseWindow: Ventanas adicionales que se abrirán desde esta clase (posiblemente para administrar, usuarios o registrarse).

Métodos:

- on_InicioSesion_btn_clicked: Método que se ejecuta cuando se presiona el botón de inicio de sesión.
- closeEvent: este metodo permite trackear cuando se cierra el programa desde el login permitiendo la pertinencia de datos por medio de huffman para la compresión y descompresión de datos de los usuarios, amistades y solicitudes
- on_Registrarse_btn_clicked: Método que se ejecuta cuando se presiona el botón de registrarse.

NodoAVL

Esta clase únicamente tiene como objetivo la gestión dentro del árbol avl

PilaReceptor

```h
#ifndef PILARECEPTOR_H
#define PILARECEPTOR_H

#include "receptor.h"
#include <iostream>
#include <unordered_map>
#include <string>

class NodoReceptor {
public:
    Receptor receptor;
    NodoReceptor* siguiente;

    NodoReceptor(const Receptor& receptor_);
};

class PilaReceptor {
private:
    NodoReceptor* cima;

public:
    PilaReceptor();
    ~PilaReceptor();
    PilaReceptor(const PilaReceptor& otra);
    PilaReceptor& operator=(const PilaReceptor& otra);

    // Métodos de la pila
    bool estaVacia() const;
    void push(const Receptor& receptor);
    Receptor pop();
    Receptor peek() const;
    void mostrarPila() const;
    bool actualizarEstadoSolicitud(const std::string& emisor, const std::string& receptor, const std::string& nuevoEstado);
    NodoReceptor* getCima() const { return cima; }
    void setCima(NodoReceptor* nuevaCima) { cima = nuevaCima; }
};

// Declaración de la función global
PilaReceptor& obtenerPilaReceptor(const std::string &correoReceptor);

#endif // PILARECEPTOR_H
```

NodoReceptor:

Representa un nodo en la pila, que contiene un objeto Receptor y un puntero al siguiente nodo.

PilaReceptor:

Atributos:

- cima: Apunta al nodo en la cima de la pila.

Constructor y Destructor:

Maneja la inicialización y destrucción de la pila.

Métodos Principales:

- estaVacia: Verifica si la pila está vacía.
- push: Inserta un nuevo Receptor en la cima de la pila.
- pop: Elimina y devuelve el receptor en la cima.
- peek: Devuelve el receptor en la cima sin eliminarlo.

- mostrarPila: Muestra el contenido de la pila.

- actualizarEstadoSolicitud: Actualiza el estado de una solicitud entre dos usuarios (emisor y receptor).

Publicaciones

```cpp
#ifndef PUBLICACION_H
#define PUBLICACION_H

#include <string>
#include "Comentario.h"
#include "arbolbcomentario.h"
#include "json.hpp"
extern ArbolBComentario arbolComentarios_;

class Publicacion {
public:
    // Constructor
    Publicacion(int id, const std::string& correo, const std::string& contenido, const std::string& fecha, const std::string& hora);

    // Destructor
    ~Publicacion();

    // Métodos de acceso a los atributos
    int getId() const;
    std::string getCorreo() const;
    std::string getContenido() const;
    std::string getFecha() const;
    std::string getHora() const;
    ArbolBComentario& getArbolComentarios();

    // Modificar el contenido
    void setContenido(const std::string& nuevoContenido);

    // Métodos relacionados con los comentarios
    void agregarComentario(const Comentario& comentario);
    void mostrarComentarios() const; // Método para mostrar comentarios
    void limpiarComentarios();

    std::string toJSON() const {
        nlohmann::json jsonData;
        jsonData["\n id"] = this->id_;
        jsonData["\n correo"] = this->correo_;
        jsonData["\n contenido"] = this->contenido_;
        jsonData["\n fecha"] = this->fecha_;
        jsonData["\n hora"] = this->hora_;

        return jsonData.dump();
    }


private:
    int id_;
    std::string correo_;
    std::string contenido_;
    std::string fecha_;
    std::string hora_;
    ArbolBComentario arbolComentarios_; // Miembro de la clase
};

#endif
```

Atributos:

- id_: Identificador único de la publicación.
- correo_: Correo del autor de la publicación.
- contenido_: Texto del contenido de la publicación.
- fecha_ y hora_: Fecha y hora en que se creó la publicación.
- Constructores y Destructor:

- Constructor que inicializa los atributos de la publicación (ID, correo, contenido, fecha y hora).
- Destructor que se encarga de liberar la memoria asociada a los comentarios.

Métodos Principales:

- getId, getCorreo, getContenido, getFecha, getHora: Métodos para obtener los valores de los atributos.
- setContenido: Permite modificar el contenido de la publicación.

Gestión de Comentarios:

La clase contiene una estructura interna llamada NodoComentario para manejar una lista enlazada de **comentarios**.

Métodos:

- agregarComentario: Añade un comentario a la lista de comentarios.
- mostrarComentarios: Muestra todos los comentarios asociados a la publicación.
- limpiarComentarios: Elimina todos los comentarios.

Receptor

```cpp
#ifndef RECEPTOR_H
#define RECEPTOR_H

#include <string>

class Receptor {
private:
    std::string emisor;
    std::string receptor;
    std::string estado;

public:
    Receptor(const std::string& emisor_, const std::string& receptor_, const std::string& estado_);

    // Getters
    std::string getEmisor() const;
    std::string getReceptor() const;
    std::string getEstado() const;

    // Setters
    void setEmisor(const std::string& emisor_);
    void setReceptor(const std::string& receptor_);
    void setEstado(const std::string& estado_);
};

#endif // RECEPTOR_H
```

Esta clase tiene como objetivo la gestión del objeto Receptor, con atributos, emisor, receptor y estado

Registrarse

```h
#ifndef REGISTRARSE_H
#define REGISTRARSE_H

#include <QDialog>
#include "ListaUsuarios.h"
#include "listaDoblePublicacion.h"
#include "lista_solicitudes.h"

class Login;

namespace Ui {
class Registrarse;
}

class Registrarse : public QDialog
{
    Q_OBJECT

public:
    explicit Registrarse(ListaUsuarios *listaUsuarios, ListaDoblePublicacion *listadoblepublicacion, ListaSolicitudes *lista_solicitudes, QWidget *parent = nullptr);
    ~Registrarse();

private slots:
    void on_Registrar_boton_clicked();

    void on_cancelar_boton_clicked();

private:
    Ui::Registrarse *ui;
    ListaUsuarios *listaUsuarios;
    ListaDoblePublicacion *listadoblepublicacion;
    ListaSolicitudes *lista_solicitudes;
    Login *login;
};

#endif // REGISTRARSE_H
```

Esta clase tiene como objetivo registrar a usuarios dentro de la aplicación, guardándolos dentro del árbol avl, se usan los atributos, listaUsuarios, listaDoblePublicacion, lista_solicitudes

Solicitud

```h
#ifndef SOLICITUD_H
#define SOLICITUD_H

#include <string>
#include <iostream>
#include "grafo_no_dirigido.h"
extern GrafoNoDirigido grafoNoDirigido;
class Solicitud {
public:
    Solicitud(const std::string &emisor, const std::string &receptor, const std::string &estado);

    std::string getEmisor() const;
    void setEmisor(const std::string &emisor);

    std::string getReceptor() const;
    void setReceptor(const std::string &receptor);

    std::string getEstado() const;
    void setEstado(const std::string &estado);
    static Solicitud deserializar(const std::string& linea) {
        // Aquí asumimos que la línea tiene el formato "emisor,receptor,estado"
        size_t pos1 = linea.find(',');
        size_t pos2 = linea.find(',', pos1 + 1);

        std::string emisor = linea.substr(0, pos1);
        std::string receptor = linea.substr(pos1 + 1, pos2 - pos1 - 1);
        std::string estado = linea.substr(pos2 + 1);
        return Solicitud(emisor, receptor, estado);
    }
private:
    std::string emisor_;
    std::string receptor_;
    std::string estado_;
};

#endif // SOLICITUD_H
```

```cpp
#ifndef USUARIO_H
#define USUARIO_H

#include <string>
#include <sstream>

class Usuario
{
public:
    Usuario(const std::string &nombre, const std::string &apellido, const std::string &fecha_de_nacimiento,
            const std::string &correo, const std::string &contrasena);

    std::string getNombre() const;
    std::string getApellido() const;
    std::string getFechaDeNacimiento() const;
    std::string getCorreo() const;
    std::string getContrasena() const;

    void setNombre(const std::string &nombre);
    void setApellido(const std::string &apellido);
    void setFechaDeNacimiento(const std::string &fecha_de_nacimiento);
    void setCorreo(const std::string &correo);
    void setContrasena(const std::string &contrasena);
    void mostrarInformacion() const;

    std::string serializar() const;
    static Usuario deserializar(const std::string& datos) {
        std::istringstream stream(datos);
        std::string nombre, apellido, fecha_nacimiento, correo, contrasena;

        std::getline(stream, nombre, ',');
        std::getline(stream, apellido, ',');
        std::getline(stream, fecha_nacimiento, ',');
        std::getline(stream, correo, ',');
        std::getline(stream, contrasena, ',');

        return Usuario(nombre, apellido, fecha_nacimiento, correo, contrasena);
    }
private:
    std::string nombre_;
    std::string apellido_;
    std::string fecha_de_nacimiento_;
    std::string correo_;
    std::string contrasena_;
};

#endif // USUARIO_H
```

Esta lista permite la gestión de los objetos solicitud con atributo emisor, receptor y estado

Usuario

```cpp
#ifndef USUARIO_H
#define USUARIO_H

#include <string>
#include <sstream>

class Usuario
{
public:
    Usuario(const std::string &nombre, const std::string &apellido, const std::string &fecha_de_nacimiento,
            const std::string &correo, const std::string &contrasena);

    std::string getNombre() const;
    std::string getApellido() const;
    std::string getFechaDeNacimiento() const;
    std::string getCorreo() const;
    std::string getContrasena() const;

    void setNombre(const std::string &nombre);
    void setApellido(const std::string &apellido);
    void setFechaDeNacimiento(const std::string &fecha_de_nacimiento);
    void setCorreo(const std::string &correo);
    void setContrasena(const std::string &contrasena);
    void mostrarInformacion() const;

    std::string serializar() const;
    static Usuario deserializar(const std::string& datos) {
        std::istringstream stream(datos);
        std::string nombre, apellido, fecha_nacimiento, correo, contrasena;

        std::getline(stream, nombre, ',');
        std::getline(stream, apellido, ',');
        std::getline(stream, fecha_nacimiento, ',');
        std::getline(stream, correo, ',');
        std::getline(stream, contrasena, ',');

        return Usuario(nombre, apellido, fecha_nacimiento, correo, contrasena);
    }
private:
    std::string nombre_;
    std::string apellido_;
    std::string fecha_de_nacimiento_;
    std::string correo_;
    std::string contrasena_;
};

#endif // USUARIO_H
```

Esta clase permite la gestión del objeto usuario el cual posee como atributos, nombre, apellido, fecha_de_nacimiento, correo, contraseña

Usuarios

```h
#ifndef USUARIOS_H
#define USUARIOS_H

#include <QDialog>
#include "listausuarios.h"
#include "listaDoblePublicacion.h"
#include "lista_solicitudes.h"
#include "pilareceptor.h"
#include "crearpublicacion.h"
#include <QComboBox>
#include <QLabel>
#include "arbolabb.h"
#include "publicacion.h"
#include <QVBoxLayout>


class CrearPublicacion;

namespace Ui {
class Usuarios;
}
class Login;
class Usuarios : public QDialog
{
    Q_OBJECT

public:
    explicit Usuarios(std::string correoUsuario, ListaUsuarios *listaUsuarios, ListaDoblePublicacion *listadoblepublicacion, ListaSolicitudes *lista_solicitudes, QWidget *parent = nullptr);
    ~Usuarios();
    ListaSolicitudes& obtenerListaSolicitudesEnviadas();
    ListaSolicitudes& obtenerListaSolicitudesRecibidas();
    std::vector<std::string> obtenerAmigos() const;
private slots:
    void on_cerrar_sesion_btn_clicked();
    void on_buscar_correo_btn_clicked();
    void on_Eliminar_boton_clicked();
    void on_Modificar_boton_clicked();
    void on_btnEnviarSolicitud_clicked(const std::string& correo);
    void on_btnCancelar_clicked(const std::string& correo);
    void on_actualizar_tablas_clicked();
    void on_btnAceptar_clicked(const std::string& correo);
    void on_btnRechazar_clicked(const std::string& correo);
    void on_fecha_filtro_publis_boton_clicked();
    void on_crear_nueva_publi_boton_clicked();
    void on_aplicar_orden_publis_boton_clicked();
    void on_generar_bst_reporte_boton_clicked();
    void actualizarPanelConImagen(const QString& imagePath);
    void mostrarPublicacionesFiltradasPorFecha(const QString& fechaSeleccionada);
    void llenarComboBoxFechas();
    void llenarComboBoxFechas_BST();
    void mostrarComentariosDePublicacion(const Publicacion& publicacion);
    void llenarFechasCantidad(NodoABB* nodo, std::vector<std::pair<std::string, int>>& fechasCantidad);
    void obtenerCantidadComentariosDePublicaciones(NodoABB* nodo, std::vector<std::pair<int, int>>& publisConMasComentarios);
    void on_generar_reportes_usuario_boton_clicked();
    void obtenerDetallesComentariosDePublicaciones(NodoABB* nodo, std::vector<std::tuple<std::string, std::string, int>>& publisConMasComentarios);
    void on_actualizarTablaRecomendados_boton_clicked();

    void on_VerAmistadySugerncias_boton_clicked();
```

```h
private:
    Ui::Usuarios *ui;
    Login *login;
    std::string correoActualUsuario_;
    ListaUsuarios *listaUsuarios;
    ListaDoblePublicacion *listadoblepublicacion;
    ListaSolicitudes *lista_solicitudes;
    CrearPublicacion *CrearPublicacion;
    QFrame* publicaciones_frame;
};

class VentanaArbolComentarios : public QDialog {
    Q_OBJECT

public:
    VentanaArbolComentarios(QWidget *parent = nullptr) : QDialog(parent) {
        setWindowTitle("Árbol de Comentarios");
        setFixedSize(1280, 720); // Ajusta el tamaño según sea necesario

        // Layout para la ventana
        QVBoxLayout* layout = new QVBoxLayout(this);

        QLabel* imageLabel = new QLabel(this);
        layout->addWidget(imageLabel);

        // Cargar la imagen generada
        QPixmap pixmap("arbol_comentarios.png");
        imageLabel->setPixmap(pixmap);
        imageLabel->setScaledContents(true); // Para ajustar la imagen al tamaño del QLabel
    }
};

#endif // USUARIOS_H
```

Atributos:

- correoActualUsuario_: Correo electrónico del usuario actual que ha iniciado sesión.
- Punteros a otras clases como ListaUsuarios, ListaDoblePublicacion, ListaSolicitudes, y CrearPublicacion, que representan las estructuras para gestionar usuarios, publicaciones, solicitudes de amistad, y la creación de publicaciones.

- ui: Interfaz gráfica de Qt para el diálogo.
- Constructores y Destructor:
- Constructor que inicializa los atributos y las listas necesarias para la gestión de usuarios, publicaciones y solicitudes.
- Destructor para liberar los recursos al cerrar el diálogo.

Métodos:

- obtenerListaSolicitudesEnviadas y obtenerListaSolicitudesRecibidas: Devuelven las listas de solicitudes de amistad enviadas y recibidas.
- obtenerAmigos: Devuelve un vector con la lista de amigos del usuario actual.
- Slots (manejadores de eventos):

Varios métodos que manejan la interacción del usuario con la interfaz gráfica, como:

- on_cerrar_sesion_btn_clicked: Cierra la sesión del usuario.
- on_buscar_correo_btn_clicked: Busca a un usuario por correo.
- on_Eliminar_boton_clicked y on_Modificar_boton_clicked: Elimina o modifica un usuario.
- Gestión de solicitudes de amistad: Enviar, cancelar, aceptar o rechazar solicitudes.
- Filtros y creación de publicaciones.
- Generación de reportes en formato de árbol binario de búsqueda (BST).
- actualizarPanelConImagen: Actualiza un panel de la interfaz con una imagen.

Grafo no dirigido

```cpp
#ifndef GRAFO_NO_DIRIGIDO_H
#define GRAFO_NO_DIRIGIDO_H

#include <iostream>
#include <string>

class Nodo {
public:
    std::string nombre;
    Nodo** vecinos;  // Arreglo dinámico de punteros a nodos vecinos
    int numVecinos;  // Cantidad de vecinos actuales
    int capacidadVecinos;  // Capacidad actual del arreglo de vecinos
    Nodo(const std::string& nombre);
    ~Nodo();

    void agregarVecino(Nodo* vecino);
    bool tieneVecino(Nodo* nodo) const;
};

class GrafoNoDirigido {
private:
    Nodo** nodos;  // Arreglo dinámico de punteros a nodos
    int numNodos;  // Cantidad de nodos actuales en el grafo
    int capacidadNodos;  // Capacidad actual del arreglo de nodos

    Nodo* encontrarNodo(const std::string& nombre) const;
    void redimensionarNodos();
    void generarArchivoDOT(const std::string& nombreArchivo) const;

public:
    // Constructor y destructor
    GrafoNoDirigido();
    ~GrafoNoDirigido();

    // Métodos públicos
    bool existeNombre(const std::string& nombre) const;
    void insertarNombre(const std::string& nombre);
    void insertarRelacion(const std::string& nombre1, const std::string& nombre2);
    std::string* obtenerAmigos(const std::string& nombre, int& cantidadAmigos) const;
    void mostrarGrafo() const;
    void generarPNG(const std::string& nombreArchivo) const;
    void generarArchivoDOTEstilos(const std::string& nombreArchivo, const std::string& correoActualUsuario_) const;
    // Nuevo método para recomendar amigos
    std::string* recomendarAmigos(const std::string& nombre, int& cantidadRecomendaciones) const;
    void eliminarUsuario(const std::string& nombre);

    int obtenerAmigosEnComun(const std::string& usuario1, const std::string& usuario2) const;
    void generarArchivoDOTListaAdyacencia(const std::string& nombreArchivo) const;
    void generarPNG_ListaAdyacencia(const std::string& nombreArchivo) const;
    void guardarAmigos() const;
    void cargarAmigos();
};

#endif // GRAFO_NO_DIRIGIDO_H
```

Clase Nodo:

- Representa un nodo en el grafo (un usuario).
- **Atributos**:
    - nombre: Nombre del nodo (usuario).
    - vecinos: Un arreglo dinámico de punteros a otros nodos (los amigos de este nodo).
    - numVecinos: Cantidad de vecinos actuales del nodo.
    - capacidadVecinos: Capacidad actual del arreglo de vecinos (se redimensiona según sea necesario).
- **Métodos**:

- **Constructor y destructor**: Inicializan el nodo y gestionan la memoria del arreglo de vecinos.
- agregarVecino(Nodo* vecino): Añade un vecino (amistad).
- tieneVecino(Nodo* nodo) const: Verifica si un nodo es vecino (amigo).

2. Clase GrafoNoDirigido:

- **Administra el grafo no dirigido, permitiendo la inserción de nodos y relaciones (amistades) entre ellos.**

Métodos principales:

- **Gestión de nodos y relaciones:**
  - **insertarNombre(const std::string& nombre): Inserta un nuevo nodo (usuario) en el grafo.**
  - **insertarRelacion(const std::string& nombre1, const std::string& nombre2): Crea una relación (amistad) entre dos nodos.**
  - **eliminarUsuario(const std::string& nombre): Elimina un nodo (usuario) del grafo.**
- **Consultas y recomendaciones:**
  - **existeNombre(const std::string& nombre) const: Verifica si un nodo existe en el grafo.**
  - **obtenerAmigos(const std::string& nombre, int& cantidadAmigos) const: Obtiene los amigos de un usuario.**
  - **recomendarAmigos(const std::string& nombre, int& cantidadRecomendaciones) const: Recomienda amigos basándose en amigos en común.**
  - **obtenerAmigosEnComun(const std::string& usuario1, const std::string& usuario2) const: Calcula la cantidad de amigos en común entre dos usuarios.**
- **Generación de visualizaciones:**
  - **generarArchivoDOT(const std::string& nombreArchivo) const: Genera un archivo DOT para visualizar el grafo.**
  - **generarPNG(const std::string& nombreArchivo) const: Convierte el archivo DOT en una imagen PNG.**
  - **generarArchivoDOTEstilos(const std::string& nombreArchivo, const std::string& correoActualUsuario_) const: Genera un archivo DOT con un estilo personalizado.**
  - **generarArchivoDOTListaAdyacencia(const std::string& nombreArchivo) const: Genera un archivo DOT con la lista de adyacencia del grafo.**

- **generarPNG_ListaAdyacencia(const std::string& nombreArchivo) const: Convierte la lista de adyacencia a una imagen PNG.**

- **Persistencia:**
  - **guardarAmigos() const: Guarda los datos de amigos en un archivo.**
  - **cargarAmigos(): Carga los datos de amigos desde un archivo.**

Huffman

```cpp
#ifndef HUFFMAN_H
#define HUFFMAN_H

#include <iostream>
#include <cstdio>
#include <cstring>

// Estructura del nodo para el árbol de Huffman
struct NodoHuffman {
    char caracter;
    int frecuencia;
    NodoHuffman* izquierda;
    NodoHuffman* derecha;

    NodoHuffman(char c, int f) : caracter(c), frecuencia(f), izquierda(nullptr), derecha(nullptr) {}
};

// Estructura para la cola de prioridad manual
struct NodoPrioridad {
    NodoHuffman* nodo;
    NodoPrioridad* siguiente;
};

// Clase Huffman para compresión y descompresión
class Huffman {
public:
    // Constructor y destructor
    Huffman();
    ~Huffman();

    // Métodos para compresión
    void calcularFrecuencias(const char* datos, int longitud, int frecuencias[256]);
    NodoHuffman* construirArbol(int frecuencias[256]);
    void generarCodigos(NodoHuffman* raiz, char* codigoActual, int profundidad, char* codigos[256]);
    void codificarDatos(const char* datos, int longitud, char* codigos[256], char*& datosCodificados, int& longitudCodificados);
    void escribirBits(const char* datosCodificados, int longitudCodificados, const char* nombreArchivoSalida);
    bool comprimir(const char* nombreArchivoEntrada, const char* nombreArchivoSalida);

    // Métodos para descompresión
    void descomprimir(const char* nombreArchivoEntrada, NodoHuffman* arbolHuffman, const char* nombreArchivoSalida);

    // Métodos para manejar la cola de prioridad
    void insertarCola(NodoPrioridad*& cabeza, NodoHuffman* nodo);
    NodoHuffman* extraerMin(NodoPrioridad*& cabeza);

private:
    NodoHuffman* raiz_;  // Nodo raíz del árbol de Huffman
};

#endif // HUFFMAN_H
```

Estructura NodoHuffman:

- Representa un nodo en el árbol de Huffman.
- **Atributos**:
  - caracter: El carácter almacenado en el nodo.

- frecuencia: La frecuencia del carácter en los datos.
- izquierda y derecha: Punteros a los nodos hijos izquierdo y derecho.

2. Estructura NodoPrioridad:

- Representa un nodo en la cola de prioridad manual utilizada para construir el árbol de Huffman.
- **Atributos**:
  - nodo: Puntero a un nodo del árbol de Huffman.
  - siguiente: Puntero al siguiente nodo en la cola de prioridad.

3. Clase Huffman:

- Implementa la lógica de compresión y descompresión utilizando el algoritmo de Huffman.

Métodos principales:

Compresión:

- calcularFrecuencias(const char* datos, int longitud, int frecuencias[256]): Calcula la frecuencia de cada carácter en los datos.
- construirArbol(int frecuencias[256]): Construye el árbol de Huffman basado en las frecuencias calculadas.
- generarCodigos(NodoHuffman* raiz, char* codigoActual, int profundidad, char* codigos[256]): Genera los códigos de Huffman para cada carácter basado en el árbol.
- codificarDatos(const char* datos, int longitud, char* codigos[256], char*& datosCodificados, int& longitudCodificados): Codifica los datos de entrada utilizando los códigos de Huffman.
- escribirBits(const char* datosCodificados, int longitudCodificados, const char* nombreArchivoSalida): Escribe los datos codificados en un archivo binario.
- comprimir(const char* nombreArchivoEntrada, const char* nombreArchivoSalida): Comprime un archivo de entrada y guarda los datos comprimidos en un archivo de salida.

Descompresión:

- descomprimir(const char* nombreArchivoEntrada, NodoHuffman* arbolHuffman, const char* nombreArchivoSalida): Descomprime un archivo comprimido utilizando el árbol de Huffman.

Manejo de la cola de prioridad:

- insertarCola(NodoPrioridad*& cabeza, NodoHuffman* nodo): Inserta un nodo en la cola de prioridad (usada para construir el árbol de Huffman).

- extraerMin(NodoPrioridad*& cabeza): Extrae el nodo con la menor frecuencia de la cola de prioridad.

Block chain

```cpp
#ifndef BLOCKCHAIN_H
#define BLOCKCHAIN_H

#include <string>

class Block {
public:
    int index;
    std::string data;
    std::string timestamp;
    std::string previousHash;
    std::string hash;
    int nonce;

    Block(int idx, const std::string& data, const std::string& previousHash);

    // Función para calcular el hash del bloque
    std::string calculateHash() const;

    // Función para minar el bloque
    void mineBlock(int difficulty);
    void saveToFile(const std::string& directory) const;
    std::string toJSON() const;
private:
    // Obtener la fecha y hora actual como string
    std::string getCurrentTime() const;
};

class Blockchain {
private:
    struct BlockNode {
        Block block;        // Contendrá el bloque
        BlockNode* next;    // Apunta al siguiente bloque
    };

    BlockNode* head;        // Apuntador al primer bloque
    int chainSize;          // Tamaño de la cadena de bloques (cantidad de bloques)

public:
    Blockchain();           // Constructor
    ~Blockchain();          // Destructor
    int difficulty;         // Dificultad de la prueba de trabajo (cantidad de ceros requeridos en el hash)

    // Obtener el último bloque de la cadena
    Block getLastBlock() const;

    // Agregar un nuevo bloque (publicación/comentario)
    void addBlock(const std::string& data);

    // Mostrar la cadena completa
    void printBlockchain() const;

private:
    // Función auxiliar para liberar la memoria de los nodos
    void clear();
};

#endif // BLOCKCHAIN_H
```

Atributos:

- index: El índice del bloque en la cadena.
- data: Los datos que contiene el bloque.
- timestamp: Fecha y hora de creación del bloque.

- previousHash: Hash del bloque anterior, para garantizar la integridad de la cadena.
- hash: Hash del bloque actual, calculado usando sus atributos.
- nonce: Un número usado en el proceso de minería para ajustar el hash.

Métodos principales:

- calculateHash(): Calcula el hash del bloque combinando sus atributos (index, data, timestamp, previousHash, nonce).
- mineBlock(int difficulty): Realiza la prueba de trabajo, ajustando el nonce hasta que el hash cumpla con la dificultad especificada (número de ceros al inicio).
- saveToFile(const std::string& directory): Guarda los detalles del bloque en un archivo en un directorio especificado.
- toJSON(): Convierte los datos del bloque en formato JSON.
- getCurrentTime(): Obtiene la fecha y hora actual.

Estructura BlockNode:

- Contiene un bloque (Block) y un puntero al siguiente nodo (BlockNode* next), formando una lista enlazada de bloques.

Atributos:

- head: Puntero al primer bloque de la cadena.
- chainSize: Tamaño actual de la cadena (cantidad de bloques).
- difficulty: La dificultad de la prueba de trabajo (número de ceros requeridos en el hash).

Métodos principales:

- Blockchain(): Constructor que inicializa la blockchain.
- ~Blockchain(): Destructor que libera la memoria de los nodos.
- getLastBlock(): Retorna el último bloque de la cadena.
- addBlock(const std::string& data): Crea y agrega un nuevo bloque con los datos proporcionados a la cadena.
- printBlockchain(): Muestra la cadena completa de bloques.

Función auxiliar:

- clear(): Libera la memoria de los nodos al eliminar la blockchain.

Paradigmas utilizados

- POO: este se puede hacer visible gracias al uso de diferentes clases y métodos los cuales nos permitieron el fácil desarrollo y manejo de datos a la hora de programar la aplicación
- Funcional: se ha utilizado en algunas partes del codigo donde se realizan operaciones sobre colecciones de datos.