

MANUAL DE USUARIO

KEVIN ANDRES ALVAREZ HERRERA

202203038

ESTRUCTURA DE DATOS

SECCION C

## **Social Structure**

## INDICE

Objetivos .....	4
Específicos .....	4
Generales .....	4
Especificación técnica .....	4
Requisitos de hardware .....	4
Memoria de almacenamiento .....	4
Lógica para la realización del programa .....	5
Clases utilizadas .....	5
Estructuras utilizadas .....	5
Explicación del código .....	6
Formato de los “.jsons” .....	6
Usuarios .....	6
Solicitudes .....	6
Publicaciones .....	6
Atributos de comentarios .....	6
Admin .....	7
ArbolABB .....	10
Comentario .....	11
Árbol B de orden 5 .....	12
CrearPublicacion .....	14
Lista_solicitudes .....	15
ListaDoblePublicacion .....	16
ArbolAVL .....	17
Login .....	18
MatrizDispersa .....	19
NodoAVL .....	20
NodoMatrizDispersa .....	20
PilaReceptor .....	21
NodoReceptor .....	21
PilaReceptor .....	21
Publicaciones .....	22
Receptor .....	23

Registrarse .....	24
Solicitud .....	24
Usuario .....	25
Usuarios .....	26
Paradigmas utilizados .....	27

# 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

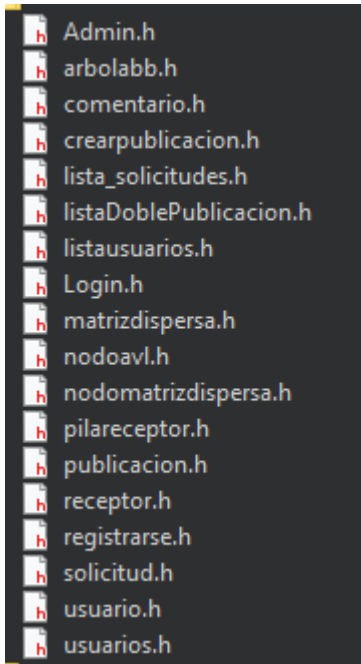
- Monitor
- Mouse
- Teclado
- CPU
- RAM

## 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



## Estructuras utilizadas

- Listas enlazadas
- Listas doblemente enlazadas
- Matriz dispersa
- Árbol AVL
- Árbol BB
- Árbol B de orden 5
- Árbol BST
- Pilas
- Colas

# Explicación del código

## 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 publicación)

### Atributos de comentarios

- Correo
- Comentario
- Fecha
- Hora

## Admin

```
#ifndef ADMIN_H
#define ADMIN_H

#include <QDialog>
#include <QVBoxLayout>
#include "ListaUsuarios.h"
#include "ListaDoblePublicacion.h"
#include "ListaSolicitudes.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_publicis(const QString& imagePath);

    bool esFechaValida(const QString& fecha);
    void on_Generar_reporte_btn_clicked();

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

#endif // ADMIN_H
```

La clase admin es la encargada de toda la parte lógica para el administrador cuenta con archivos .cpp y .ui que le permiten no solo la parte lógica sino también la parte visual gracias a Qt, posee como atributos la listaUsuarios, listaDoblePublicacion, lista\_solicitudes, estos atributos le permiten la gestión de los usuarios, publicación y las solicitudes a su vez posee varios métodos que al presionar botones le permiten hacer diferentes acciones, ejemplo:

Los métodos

```
void on_Usuarios_boton_archivo_clicked();
```

```
void on_Publicaciones_boton_archivo_clicked();
```

```
void on_Solicitudes_boton_archivo_clicked();
```

le permiten al código tener la lógica de que cuando se cliquea en los respectivos botones, buscar archivos .json y ingresarlos en las diferentes estructuras

```

void Admin::on_Usuarios_boton_archivo_clicked()
{
    QString filename = QFileDialog::getOpenFileName(this, "Seleccionar archivo JSON", "", "Archivos JSON (*.json)");

    if (!filename.isEmpty())
    {
        std::ifstream archivo(filename.toStdString());

        if (archivo.is_open())
        {
            listaUsuarios->cargarUsuariosDesdeJson(filename.toStdString());
            QMessageBox::information(this, "Cargar usuarios", "Usuarios cargados exitosamente.");
        }
        else
        {
            QMessageBox::warning(this, "Error", "No se pudo abrir el archivo.");
        }
    }
    else
    {
        QMessageBox::warning(this, "Error", "No se seleccionó ningún archivo.");
    }
}

```

```

void Admin::on_Solicitudes_boton_archivo_clicked()
{
    QString filename = QFileDialog::getOpenFileName(this, "Seleccionar archivo JSON", "", "Archivos JSON (*.json)");

    if (!filename.isEmpty())
    {
        std::ifstream archivo(filename.toStdString());

        if (archivo.is_open())
        {
            lista_solicitudes->cargarRelacionesDesdeJson(filename.toStdString());
            lista_solicitudes->agregarRelacionesAceptadasAMatriz(matrizDispersa);
            matrizDispersa.mostrarMatriz();
            QMessageBox::information(this, "Cargar usuarios", "Usuarios cargados exitosamente.");
        }
        else
        {
            QMessageBox::warning(this, "Error", "No se pudo abrir el archivo.");
        }
    }
    else
    {
        QMessageBox::warning(this, "Error", "No se seleccionó ningún archivo.");
    }
}

```

```

void Admin::on_Publicaciones_boton_archivo_clicked()
{
    QString filename = QFileDialog::getOpenFileName(this, "Seleccionar archivo JSON", "", "Archivos JSON (*.json)");
    std::cout << "Intentando leer el archivo..." << std::endl;

    if (!filename.isEmpty())
    {
        std::ifstream archivo(filename.toStdString());

        if (archivo.is_open())
        {
            std::cout << "Archivo abierto correctamente." << std::endl;

            try {
                listadoblepublicacion->cargarPublicacionesDesdeJson(filename.toStdString());
                listadoblepublicacion->mostrarTodasLasPublicaciones();
                QMessageBox::information(this, "Cargar publicaciones", "Publicaciones cargadas exitosamente.");
            } catch (const std::exception& e) {
                std::cerr << "Error al cargar las publicaciones: " << e.what() << std::endl;
                QMessageBox::critical(this, "Error", "Hubo un error al cargar el archivo JSON.");
            }

            archivo.close();
        }
        else
        {
            std::cerr << "No se pudo abrir el archivo." << std::endl;
            QMessageBox::warning(this, "Error", "No se pudo abrir el archivo.");
        }
    }
    else
    {
        std::cerr << "No se seleccionó ningún archivo." << std::endl;
        QMessageBox::warning(this, "Error", "No se seleccionó ningún archivo.");
    }
}

```



```

void Admin::on_aplicar_orden_comboBox_orden_tabla_usuario_clicked()
{
    // Obtener el texto seleccionado en el combo box
    QString criterioOrden = ui->comboBox_orden_tabla_usuario->currentText();

    // Obtener el vector de usuarios en el orden seleccionado
    std::vector<Usuario> usuariosOrdenados = listaUsuarios->obtenerUsuariosEnOrden(criterioOrden.toString());

    // Limpiar la tabla actual antes de actualizarla
    ui->tabla_buscar_admin->setRowCount(0); // Resetear las filas de la tabla

    // Actualizar la tabla con los usuarios ordenados
    for (const Usuario& usuario : usuariosOrdenados) {
        int row = ui->tabla_buscar_admin->rowCount(); // Obtener el número de filas actuales
        ui->tabla_buscar_admin->insertRow(row); // Insertar una nueva fila

        // Insertar los datos del usuario en las columnas correspondientes
        ui->tabla_buscar_admin->setItem(row, 0, new QTableWidgetItem(QString::fromStdString(usuario.getNombre())));
        ui->tabla_buscar_admin->setItem(row, 1, new QTableWidgetItem(QString::fromStdString(usuario.getApellido())));
        ui->tabla_buscar_admin->setItem(row, 2, new QTableWidgetItem(QString::fromStdString(usuario.getCorreo())));
        ui->tabla_buscar_admin->setItem(row, 3, new QTableWidgetItem(QString::fromStdString(usuario.getFechaDeNacimiento())));

        // Crear botones de Modificar y Eliminar
        QPushButton* btnModificar = new QPushButton("Modificar");
        QPushButton* btnEliminar = new QPushButton("Eliminar");

        // Añadir los botones a las columnas correspondientes
        ui->tabla_buscar_admin->setCellWidget(row, 4, btnModificar);
        ui->tabla_buscar_admin->setCellWidget(row, 5, btnEliminar);

        // Conectar los botones a sus respectivos slots
        connect(btnModificar, &QPushButton::clicked, [this, usuario, row]() {
            this->on_modificar_usuario_clicked(usuario.getCorreo(), row);
        });
        connect(btnEliminar, &QPushButton::clicked, [this, usuario]() {
            this->on_eliminar_usuario_clicked(usuario.getCorreo());
        });
    }
}

```

Este código permite la aplicación de los diferentes ordenes PreOrder, InOrder, PostOrder a el árbol avl y lo devuelve en la tabla\_buscar\_admin

## ArbolABB

```
#include <ostream>
#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

```
#ifndef COMENTARIO_H
#define COMENTARIO_H

#include <string>

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();
    }

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

```
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include "comentario.h"
#include <map>
#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);
};

// Clase que define el Árbol B de comentarios
class ArbolBComentario {
public:
    ArbolBComentario();

    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 graficarArbol(const std::string& nombreArchivo, int idPublicacion);
    void graficarNodo(NodoArbolBComentario* nodo, int& nodeCount, std::map<NodoArbolBComentario*, int>& nodeMap, std::ofstream& archivo, int idPublicacion);

private:
    NodoArbolBComentario* raiz;
    void obtenerComentariosRekursivos(NodoArbolBComentario* nodo, int idPublicacion, 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.
- obtenerComentariosRekursivos: 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

```
#ifndef LISTA_SOLICITUDES_H
#define LISTA_SOLICITUDES_H

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

extern MatrizDispersa matrizDispersa;

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

        NodoSolicitud(const Solicitud &solicitud);
    };

    NodoSolicitud* cabeza;

public:
    ListaSolicitudes();
    ~ListaSolicitudes();

    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 buscarVApilarPendientes(const std::string &correo, const ListaSolicitudes &listaSolicitudes);
    bool actualizarEstadoSolicitud(const std::string& emisor, const std::string& receptor, const std::string& nuevoEstado);
    void agregarRelacionesAceptadasAMatriz(MatrizDispersa &matriz);
};

#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: Envía 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 una matriz dispersa.

## ListadoDoblePublicacion

```

#ifndef LISTADOBLEPUBLICACION_H
#define LISTADOBLEPUBLICACION_H

#include "publicacion.h"
#include <string>
#include "matrizdispersa.h"
#include "arbolabb.h"
#include "arbolbcomentario.h"
#include "comentario.h"
extern MatrizDispersa matrizDispersa;
extern ArbolABB arbolABB;
extern ArbolBComentario arbolComentarios_;

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

        NodoPublicacion(const Publicacion &publicacion);
    };

    NodoPublicacion *cabecera;
    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:
    ListadoDoblePublicacion();
    ~ListadoDoblePublicacion();
    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 MatrizDispersa &matriz, ArbolABB &arbol, const std::string &orden);
    int obtenerNuevoId() const;
    void mostrarPublicacionesOrden(const std::string &correoUsuario, const MatrizDispersa &matrizDispersa, ArbolABB &arbolABB, int orden) const;
    std::vector<Publicacion> obtenerPublicacionesEnOrden(const std::string &tipoOrden) const;
    void eliminarPublicacionPorId(int id);
};

#endif // LISTADOBLEPUBLICACION_H

```

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

**ListadoDoblePublicacion:**

**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 matriz dispersa 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

```
#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 buscarUsuarioPorCorreoYContraseña(const std::string& correo, const std::string& contraseña) 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

```
#ifndef LOGIN_H
#define LOGIN_H

#include <QMainWindow>
#include "ListaUsuarios.h"
#include "ListaDoblePublicacion.h"
#include "ListaSolicitudes.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();

private slots:
    void on_InicioSesion_btn_clicked();
    void on_Registrarse_btn_clicked();

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.
- on\_Registrarse\_btn\_clicked: Método que se ejecuta cuando se presiona el botón de registrarse.

## MatrizDispersa

```
#ifndef MATRIZ_DISPERSA_H
#define MATRIZ_DISPERSA_H

#include <iostream>
#include <fstream>
#include <vector>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
#include "nodomatrizdispersa.h"

class MatrizDispersa {
private:
    std::vector<std::string> nombres;
    NodoMatrizDispersa* cabeza;
    NodoMatrizDispersa* buscarFila(const std::string& fila) const;
    NodoMatrizDispersa* buscarColumna(NodoMatrizDispersa* filaNodo, const std::string& columna) const;

    std::string escapeXml(const std::string& input) const;

public:
    // Constructor
    MatrizDispersa();

    // Métodos públicos
    bool existeNombre(const std::string& nombre) const;
    void insertarNombre(const std::string& nombre);
    void insertarRelacion(const std::string& nombreFila, const std::string& nombreColumna);

    std::vector<std::string> obtenerAmigos(const std::string& correo) const;
    void mostrarMatriz() const;
};

#endif // MATRIZ_DISPERSA_H
```

Atributos Privados:

- nombres: Vector que almacena los nombres (o identificadores) de las filas y columnas.
- cabeza: Apunta al nodo principal de la matriz (NodoMatrizDispersa).

Métodos de búsqueda:

- buscarFila: Busca un nodo de fila según el nombre.
- buscarColumna: Busca un nodo de columna dentro de una fila específica.

escapeXml: Método que convierte caracteres especiales en formato XML.

Métodos Públicos:

- existeNombre: Verifica si un nombre ya existe en la matriz.
- insertarNombre: Añade un nuevo nombre a la matriz.
- insertarRelacion: Inserta una relación entre una fila (usuario) y una columna (amigo).
- obtenerAmigos: Devuelve los amigos (relaciones) de un usuario específico.
- mostrarMatriz: Muestra la matriz completa en consola.

## NodoAVL

```
#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
```

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

## NodoMatrizDispersa

```
#ifndef NODOMATRIZDISPERSA_H
#define NODOMATRIZDISPERSA_H

#include <string>

class NodoMatrizDispersa
{
public:
    std::string nombreFila;
    std::string nombreColumna;
    NodoMatrizDispersa *derecha;
    NodoMatrizDispersa *abajo;
    NodoMatrizDispersa *arriba;
    NodoMatrizDispersa *izquierda;

    NodoMatrizDispersa(const std::string& fila, const std::string& columna);
};

#endif // NODOMATRIZDISPERSA_H
```

Esta clase tiene como objetivo la gestión dentro de la matriz dispersa

## PilaReceptor

```
#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

```
#ifndef PUBLICACION_H
#define PUBLICACION_H

#include <string>
#include "Comentario.h"
#include "ArbolComentario.h"
extern ArbolComentario 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;
    ArbolComentario& 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();

private:
    int id_;
    std::string correo_;
    std::string contenido_;
    std::string fecha_;
    std::string hora_;
    ArbolComentario 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

```
#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

```
#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

```
#ifndef SOLICITUD_H
#define SOLICITUD_H

#include <string>
#include <iostream>

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);

private:
    std::string emisor_;
    std::string receptor_;
    std::string estado_;
};
```

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



## Usuario

```
#ifndef USUARIO_H
#define USUARIO_H

#include <string>

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;

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

```
#ifndef USUARIOS_H
#define USUARIOS_H

#include <QDialog>
#include "ListaUsuarios.h"
#include "ListaDoblePublicacion.h"
#include "ListaSolicitudes.h"
#include "PilaReceptor.h"
#include "CrearPublicacion.h"
#include <QComboBox>
#include <QLabel>

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_public_boton_clicked();
    void on_crear_nueva_public_boton_clicked();
    void on_aplicar_orden_public_boton_clicked();
    void on_generar_bst_reporte_boton_clicked();
    void actualizarPanelConImagen(const QString& imagePath);

private:
    UI::Usuarios *ui;
    Login *login;
    std::string correoActualUsuario_;
    ListaUsuarios *listaUsuarios;
    ListaDoblePublicacion *listadoblepublicacion;
    ListaSolicitudes *lista_solicitudes;
    CrearPublicacion *CrearPublicacion;
};

#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.

## 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 código donde se realizan operaciones sobre colecciones de datos.