Manual De Técnico

26/10/2024

Samuel Nehemias Coyoy Perez

202200198

Contenido

Objetivos

Específicos

- Utilización de estructura de datos

Generales

- Creación de Usuarios
- Creación de solicitudes de amistad
- Creación de publicaciones

Especificación Técnica

Requisitos Hardware

- Mouse
- Teclado
- Monitor

Requisitos Software

- Sistema Operativo: Windows 10
- Herramientas: Visual Studio Code y Qt creator
- Lenguaje de Programación: C++

Manual Técnico: Red Social Tails

Main.cpp

La función principal es int main(int argc, char *argv[]), donde se inicia la ejecución de la aplicación Qt. Los parámetros argc y argv permiten manejar los argumentos de la línea de comandos si es necesario.

Se crea un objeto de la clase QApplication llamado a, que es el encargado de gestionar el loop de eventos de la aplicación Qt y otros recursos globales. Este objeto es esencial para manejar las interacciones de la interfaz gráfica de usuario.

Se instancia la clase MainWindow, que es la ventana principal de la aplicación. Esta clase está definida en el archivo mainwindow.h.

La función show() se utiliza para hacer visible la ventana principal en pantalla. Sin esta llamada, la ventana no se mostraría al usuario.

La llamada a a.exec() es la que inicia el ciclo de eventos de Qt. Este ciclo se mantiene activo hasta que el usuario cierre la aplicación, permitiendo que la interfaz responda a las interacciones del usuario.

MainWindow.cpp

Este archivo contiene la implementación del constructor y destructor de la clase MainWindow, así como los métodos que gestionan los eventos de los botones de login y registro en la interfaz gráfica. En el constructor, se inicializa la interfaz llamando a ui->setupUi(this), mientras que el destructor se encarga de liberar la memoria asignada a ui. El método on_loginbtn_clicked() se activa cuando el usuario presiona el botón de login; aquí se obtienen las credenciales ingresadas y se comparan con las credenciales del administrador o las almacenadas en el árbol AVL de la aplicación. Si el login es exitoso, dependiendo de si es administrador o usuario común, se oculta la ventana de login y se muestra la ventana correspondiente. En caso de que las credenciales no sean correctas, se imprime un mensaje en la consola. Por otro lado, el método on_registrobtn_clicked() se ejecuta al presionar el botón de registro, ocultando la ventana de login y mostrando la ventana de registro si aún no ha sido creada. Este archivo es fundamental para la gestión del inicio de sesión y el flujo entre diferentes ventanas en la aplicación.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    // Inicializa la interfaz de usuario para la ventana principal.
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    // Libera la memoria utilizada por la interfaz de usuario.
    delete ui;
}

void MainWindow::on_loginbtn_clicked()
{
    // Definición de credenciales de administrador.
    std::string userAd = "admin@gmail.com";
    std::string passAd = "EDD2S2024";

    // Obtención de las credenciales ingresadas por el usuario.
    QString usuario = ui->userline->text();
    QString password = ui->passline->text();

    // Verificación de administrador.
    if(userAd == usuario.toStdString() && passAd == password.toStdString()) {
        if(!ventanaAdmin) {
            ventanaAdmin = new admin(this);
```

```
        }
        this->hide(); // Oculta la ventana de login.
        ventanaAdmin->show(); // Muestra la ventana de administración.
    }
    // Verificación de usuario común en el árbol AVL.
    else
if(AppData::getInstance().getAVLTree().verifyCredentials(usuario.toStdString(),
password.toStdString())) {
        if(!ventanaUser) {
            ventanaUser = new user(this);
        }
        this->hide(); // Oculta la ventana de login.
        ventanaUser->show(); // Muestra la ventana de usuario.
    }
    // Verificación de credenciales de usuario "user".
    else if(usuario.toStdString() == "user" && password.toStdString() == "user") {
        if(!ventanaUser) {
            ventanaUser = new user(this);
        }
        this->hide(); // Oculta la ventana de login.
        ventanaUser->show(); // Muestra la ventana de usuario.
    }
    else {
        // Muestra un mensaje de error si las credenciales son incorrectas.
        qDebug() << "Usuario o contraseña incorrecta";
    }
}


void MainWindow::on_registrobtn_clicked()
```

```
{
    // Muestra la ventana de registro y oculta la ventana de login.
    if(!ventanaRegistro) {
        ventanaRegistro = new registro(this);
    }
    this->hide(); // Oculta la ventana de login.
    ventanaRegistro->show(); // Muestra la ventana de registro.
}
```

Admin.cpp

Descripción General

El archivo `admin.cpp` define la clase `admin`, que es una ventana de diálogo en la aplicación de administración. Esta clase permite gestionar usuarios mediante una interfaz gráfica, incluyendo la carga de datos, visualización, modificación y eliminación de usuarios.

Funcionalidades Clave

1. **Carga de Usuarios desde JSON (`on_cargarUserbtn_clicked`)**
   - Permite seleccionar un archivo JSON que contiene datos de usuarios.
   - Los datos se leen y se insertan en el árbol AVL global gestionado por `AppData`.

2. **Visualización de Usuarios (`on_showUserbtn_clicked`)**
   - Realiza un recorrido en orden del árbol AVL y muestra los datos en una tabla.

3. **Mostrar Datos en Tabla (`mostrarDatosEnTabla`)**
   - Limpia la tabla y llena filas con los datos de los usuarios del árbol AVL.

- Agrega botones para modificar y eliminar usuarios, con funcionalidades asociadas.

4. **Eliminar Usuario (`eliminarUsuario`)**

- Elimina un usuario del árbol AVL y actualiza la tabla.

5. **Buscar Usuario (`on_buscarbtn_clicked`)**

- Busca un usuario en el árbol AVL basado en un correo electrónico ingresado.

- Muestra los datos del usuario encontrado en la tabla.

6. **Recorridos de Árbol (`on_ordenbtn_clicked`)**

- Permite seleccionar diferentes métodos de recorrido (InOrder, PreOrder, PostOrder) para visualizar usuarios en la tabla.

Detalles de Implementación

- **Interacción con el Diálogo de Modificación**

- Se oculta la ventana principal cuando se abre el diálogo para modificar un usuario.

- Se actualizan los datos en el árbol AVL y en la tabla después de la modificación.

- **Conexión de Señales y Slots**

- Se utilizan `QPushButton` para modificar y eliminar usuarios. Las señales de clic se conectan a las respectivas funciones de modificación y eliminación.

```
admin::admin(QWidget *parent)
    : QDialog(parent)
    , ui(new Ui::admin)
{
    ui->setupUi(this);
```

```
    avlTemporal = new AVL();  // Inicializa un AVL temporal
}

admin::~admin()
{
    delete ui;
    delete avlTemporal;  // Libera la memoria del AVL temporal
}

void admin::on_cargarUserbtn_clicked()
{
    // Cargar usuarios desde un archivo JSON
    AppData& appData = AppData::getInstance();
    QString ruta = QFileDialog::getOpenFileName(this, "Open file", "/", "Text Files
(*.json);;All Files (*,*)");

    if (!ruta.isEmpty()) {
        QFile file(ruta);
        if (file.open(QFile::ReadOnly)) {
            QByteArray jsonData = file.readAll();
            file.close();
            QJsonDocument doc = QJsonDocument::fromJson(jsonData);

            if (doc.isArray()) {
                QJsonArray root = doc.array();
                foreach (const QJsonValue &v, root) {
                    QJsonObject obj = v.toObject();
                    // Extrae los datos del JSON y los inserta en el AVL
                    QString correo = obj.value("correo").toString();
```

```
                appData.getAVLTree().insert(correo.toStdString(), /* otros datos */);
            }
            QMessageBox::information(this, "Éxito", "Los datos se han cargado exitosamente.");
        } else {
            QMessageBox::warning(this, "Error", "El archivo JSON no es un array.");
        }
    } else {
        QMessageBox::warning(this, "Error", "No se pudo abrir el archivo.");
    }
}

void admin::on_cargarSolibtn_clicked()
{
    // Función no implementada
}

void admin::on_cargarPublibtn_clicked()
{
    // Función no implementada
}

void admin::on_pushButton_6_clicked()
{
    // Cerrar sesión y mostrar la ventana principal
    MainWindow *ventana = new MainWindow(this);
    this->close();
    ventana->show();
```

```
}

void admin::on_showUserbtn_clicked()
{
    // Mostrar usuarios en consola usando el recorrido in-order del AVL
    AppData& appData = AppData::getInstance();
    appData.getAVLTree().inorder();
}

void admin::mostrarDatosEnTabla()
{
    // Mostrar datos en una tabla
    ui->tableBuscar->setRowCount(0);
    int row = 0;
    AppData& appData = AppData::getInstance();

    auto inorderToTable = [&](shared_ptr<Node> node, auto&& inorderToTableRef) -
> void {
        if (!node) return;

        inorderToTableRef(node->left, inorderToTableRef);

        ui->tableBuscar->insertRow(row);
        // Llenar datos en la tabla y agregar botones de modificar y eliminar
        ui->tableBuscar->setItem(row, 0, new
QTableWidgetItem(QString::fromStdString(node->nombre)));
        // Agregar botones con funcionalidades
        QPushButton *btnModificar = new QPushButton("Modificar");
        QPushButton *btnEliminar = new QPushButton("Eliminar");
```

```
        connect(btnModificar, &QPushButton::clicked, [this, node, &appData]() {
            // Lógica para modificar usuario
            this->hide();
            DialogModificar dialog;
            // Establecer datos en el diálogo y actualizar AVL
            int resultado = dialog.exec();
            if (resultado == QDialog::Accepted) {
                // Actualizar usuario en AVL
                appData.getAVLTree().deleteNode(node->email);
                appData.getAVLTree().insert(/* nuevos datos */);
                mostrarDatosEnTabla();
                QMessageBox::information(this, "Modificado", "El usuario ha sido
modificado con éxito.");
            }
        });

        connect(btnEliminar, &QPushButton::clicked, [this, node]() {
            // Lógica para eliminar usuario
            this->eliminarUsuario(node);
        });

        ui->tableBuscar->setCellWidget(row, 4, btnModificar);
        ui->tableBuscar->setCellWidget(row, 5, btnEliminar);

        row++;
        inorderToTableRef(node->right, inorderToTableRef);
    };
```

```
    inorderToTable(appData.getAVLTree().getRoot(), inorderToTable);
}

void admin::eliminarUsuario(shared_ptr<Node> node)
{
    // Eliminar usuario del AVL y actualizar la tabla
    AppData& appData = AppData::getInstance();
    appData.getAVLTree().deleteNode(node->email);
    mostrarDatosEnTabla();
    QMessageBox::information(this, "Eliminado", "El usuario ha sido eliminado.");
}

void admin::on_buscarbtn_clicked()
{
    // Buscar usuario por correo y mostrar en la tabla
    AppData& appData = AppData::getInstance();
    QString correo = ui->searchLine->text();
    std::string email = correo.toStdString();
    ui->tableBuscar->setRowCount(0);
    int row = 0;

    shared_ptr<Node> nodo = appData.getAVLTree().getNode(email);
    if (nodo) {
        ui->tableBuscar->insertRow(row);
        // Llenar datos en la tabla y agregar botones
        QPushButton *btnModificar = new QPushButton("Modificar");
        QPushButton *btnEliminar = new QPushButton("Eliminar");
```

```
        connect(btnModificar, &QPushButton::clicked, [this, nodo]() {
            // Lógica para modificar usuario
            this->hide();
            DialogModificar dialog;
            // Establecer datos en el diálogo y actualizar AVL
            int resultado = dialog.exec();
            if (resultado == QDialog::Accepted) {
                AppData& appData = AppData::getInstance();
                appData.getAVLTree().deleteNode(nodo->email);
                appData.getAVLTree().insert(/* nuevos datos */);
                mostrarDatosEnTabla();
                QMessageBox::information(this, "Modificado", "El usuario ha sido
modificado con éxito.");
            }
        });

        connect(btnEliminar, &QPushButton::clicked, [this, nodo]() {
            // Lógica para eliminar usuario
            this->eliminarUsuario(nodo);
        });

        ui->tableBuscar->setCellWidget(row, 4, btnModificar);
        ui->tableBuscar->setCellWidget(row, 5, btnEliminar);

        QMessageBox::information(this, "Búsqueda exitosa", "El usuario ha sido
encontrado.");
    } else {
        QMessageBox::warning(this, "Error", "Usuario no encontrado en el AVL
global.");
```

```
    }
}

void admin::on_ordenbtn_clicked()
{
    // Mostrar datos en la tabla según el recorrido seleccionado
    QString ordenSeleccionado = ui->ordenBox->currentText();
    ui->tableBuscar->setRowCount(0);
    int row = 0;
    AppData& appData = AppData::getInstance();

    auto addToTable = [&](shared_ptr<Node> node) {
        ui->tableBuscar->insertRow(row);
        ui->tableBuscar->setItem(row, 0, new
QTableWidgetItem(QString::fromStdString(node->nombre)));
        // Agregar botones
        QPushButton *btnModificar = new QPushButton("Modificar");
        QPushButton *btnEliminar = new QPushButton("Eliminar");

        connect(btnModificar, &QPushButton::clicked, [this, node]() {
            // Lógica para modificar usuario
            this->hide();
            DialogModificar dialog;
            // Establecer datos en el diálogo y actualizar AVL
            int resultado = dialog.exec();
            if (resultado == QDialog::Accepted) {
                AppData& appData = AppData::getInstance();
                appData.getAVLTree().deleteNode(node->email);
                appData.getAVLTree().insert(/* nuevos datos */);
```

```
            mostrarDatosEnTabla();
            QMessageBox::information(this, "Modificado", "El usuario ha sido modificado con éxito.");
        }
    });

    connect(btnEliminar, &QPushButton::clicked, [this, node]() {
        // Lógica para eliminar usuario
        this->eliminarUsuario(node);
    });

    ui->tableBuscar->setCellWidget(row, 4, btnModificar);
    ui->tableBuscar->setCellWidget(row, 5, btnEliminar);
    row++;
};

// Ejecutar el recorrido seleccionado
if (ordenSeleccionado == "InOrder") {
    appData.getAVLTree().inorderTraversal([&](shared_ptr<Node> node) { addToTable(node); });
} else if (ordenSeleccionado == "PreOrder") {
    appData.getAVLTree().preorderTraversal([&](shared_ptr<Node> node) { addToTable(node); });
} else if (ordenSeleccionado == "PostOrder") {
    appData.getAVLTree().postorderTraversal([&](shared_ptr<Node> node) { addToTable(node); });
}
}
```

Registro.cpp

Descripción General

El archivo `dialogmodificar.cpp` define la implementación de la clase `DialogModificar`, que es un diálogo para modificar los datos de un usuario. Este diálogo permite al usuario ingresar y actualizar información como nombre, apellido, correo electrónico, fecha de nacimiento y contraseña.

Funcionalidades Clave

1. **Constructor (`DialogModificar::DialogModificar`)**

   - **Propósito**: Inicializa el diálogo y configura la interfaz gráfica.

   - **Implementación**: Llama al constructor base de `QDialog` y luego configura la interfaz de usuario usando `setupUi`.

2. **Destructor (`DialogModificar::~DialogModificar`)**

   - **Propósito**: Libera los recursos asociados con la interfaz de usuario cuando el diálogo se destruye.

   - **Implementación**: Elimina el puntero `ui` que gestiona los elementos gráficos del diálogo.

3. **Setters**: Métodos para establecer valores en los campos de entrada del diálogo.

   - **`setNombre(const QString &nombre)`**: Establece el texto del campo de nombre (`lineEditNombre`) a `nombre`.

   - **`setApellido(const QString &apellido)`**: Establece el texto del campo de apellido (`lineEditApellido`) a `apellido`.

   - **`setEmail(const QString &email)`**: Establece el texto del campo de correo electrónico (`lineEditEmail`) a `email`.

   - **`setFechaNacimiento(const QString &fecha)`**: Establece el texto del campo de fecha de nacimiento (`lineEditFechaNacimiento`) a `fecha`.

- **`setPassword(const QString &password)`**: Establece el texto del campo de contraseña (`lineEditPassword`) a `password`.

4. **Getters**: Métodos para obtener los valores de los campos de entrada del diálogo.

- **`getNombre() const`**: Devuelve el texto del campo de nombre (`lineEditNombre`).

- **`getApellido() const`**: Devuelve el texto del campo de apellido (`lineEditApellido`).

- **`getEmail() const`**: Devuelve el texto del campo de correo electrónico (`lineEditEmail`).

- **`getFechaNacimiento() const`**: Devuelve el texto del campo de fecha de nacimiento (`lineEditFechaNacimiento`).

- **`getPassword() const`**: Devuelve el texto del campo de contraseña (`lineEditPassword`).

CODIGO COMENTADO

```
#include "dialogmodificar.h"      // Incluye la definición de la clase DialogModificar
#include "ui_dialogmodificar.h"  // Incluye la definición de la interfaz de usuario generada

// Constructor de la clase DialogModificar
DialogModificar::DialogModificar(QWidget *parent)
    : QDialog(parent)         // Llama al constructor base de QDialog
    , ui(new Ui::DialogModificar)  // Inicializa el puntero ui con la instancia de la interfaz
{
    ui->setupUi(this);  // Configura la interfaz de usuario del diálogo
}
```

```
// Destructor de la clase DialogModificar
DialogModificar::~DialogModificar()
{
    delete ui;  // Libera la memoria ocupada por el puntero ui
}

// Métodos Setters para establecer valores en los campos del diálogo

// Establece el texto del campo de nombre
void DialogModificar::setNombre(const QString &nombre) {
    ui->lineEditNombre->setText(nombre);  // Asigna el valor a lineEditNombre
}

// Establece el texto del campo de apellido
void DialogModificar::setApellido(const QString &apellido) {
    ui->lineEditApellido->setText(apellido);  // Asigna el valor a lineEditApellido
}

// Establece el texto del campo de correo electrónico
void DialogModificar::setEmail(const QString &email) {
    ui->lineEditEmail->setText(email);  // Asigna el valor a lineEditEmail
}

// Establece el texto del campo de fecha de nacimiento
void DialogModificar::setFechaNacimiento(const QString &fecha) {
    ui->lineEditFechaNacimiento->setText(fecha);  // Asigna el valor a
lineEditFechaNacimiento
}
```

```
// Establece el texto del campo de contraseña
void DialogModificar::setPassword(const QString &password) {
    ui->lineEditPassword->setText(password);  // Asigna el valor a lineEditPassword
}

// Métodos Getters para obtener valores de los campos del diálogo

// Devuelve el texto del campo de nombre
QString DialogModificar::getNombre() const {
    return ui->lineEditNombre->text();  // Retorna el texto del campo lineEditNombre
}

// Devuelve el texto del campo de apellido
QString DialogModificar::getApellido() const {
    return ui->lineEditApellido->text();  // Retorna el texto del campo lineEditApellido
}

// Devuelve el texto del campo de correo electrónico
QString DialogModificar::getEmail() const {
    return ui->lineEditEmail->text();  // Retorna el texto del campo lineEditEmail
}

// Devuelve el texto del campo de fecha de nacimiento
QString DialogModificar::getFechaNacimiento() const {
    return ui->lineEditFechaNacimiento->text();  // Retorna el texto del campo
lineEditFechaNacimiento
}

// Devuelve el texto del campo de contraseña
```

```
QString DialogModificar::getPassword() const {
    return ui->lineEditPassword->text();  // Retorna el texto del campo lineEditPassword
}
```

USER.CPP

El archivo user.cpp es parte de una aplicación Qt que parece gestionar usuarios, publicaciones y solicitudes de amistad en una red social o sistema similar. La clase user hereda de QDialog, lo que indica que maneja la interfaz de un cuadro de diálogo donde se pueden ver y modificar los detalles del usuario, buscar otros usuarios, ver publicaciones y administrar solicitudes de amistad.

Funciones principales que realiza el archivo:

1. **Inicialización del diálogo (user::user)**:
    - Se inicializan los elementos de la interfaz y se obtienen los detalles del usuario desde una instancia de AppData que almacena los datos del sistema.
    - Muestra usuarios en una tabla, solicitudes recibidas, enviadas y guarda publicaciones en un árbol binario.
2. **Botón de búsqueda (on_buscarbtn_clicked)**:
    - Permite buscar usuarios por correo electrónico y muestra los detalles del usuario encontrado.
3. **Cerrar sesión (on_logoutbtn_clicked)**:
    - Elimina las publicaciones del árbol binario y regresa a la ventana principal de la aplicación.
4. **Modificar los datos del usuario (on_modificarBtn_clicked)**:
    - Modifica la información del usuario y actualiza su correo si es necesario.
5. **Eliminar usuario (on_eliminarBtn_clicked)**:
    - Elimina al usuario del árbol AVL (estructura de datos) y regresa a la ventana principal.
6. **Crear una publicación (on_creaPubliBtn_clicked)**:

- Abre una nueva ventana donde el usuario puede crear publicaciones.

7. **Mostrar publicaciones por fecha (on_fechaBtn_clicked)**:
    - Muestra las publicaciones en un QScrollArea.

Método que está explicando: on_buscarbtn_clicked

```
void user::on_buscarbtn_clicked()
{
    AppData& appData = AppData::getInstance();
    QString search = ui->searchLine->text();


    string nombre, apellido, fechaDeNacimiento, password;
    if (appData.getAVLTree().getUserDetails(search.toStdString(), nombre, apellido, fechaDeNacimiento, password)) {
        ui->nameLine->setText(QString::fromStdString(nombre));
        ui->apellidoLine->setText(QString::fromStdString(apellido));
        ui->fechaLine->setText(QString::fromStdString(fechaDeNacimiento));
        ui->correoLine->setText(QString::fromStdString(search.toStdString()));
    } else {
        qDebug() << "Usuario no encontrado.";
    }
}
```

Explicación de on_buscarbtn_clicked:

1. **Propósito**:
    - Este método es llamado cuando el usuario hace clic en el botón de búsqueda. Permite buscar un usuario en el sistema por correo electrónico y muestra sus detalles en la interfaz.
2. **Detalles del funcionamiento**:

- Se obtiene el texto de la línea de búsqueda (ui->searchLine->text()).
- Se accede a los datos de la aplicación a través de la instancia singleton AppData y se busca al usuario en un árbol AVL usando el método getUserDetails().
- Si el usuario se encuentra, se muestran sus detalles en los campos de texto correspondientes: nombre, apellido, fecha de nacimiento y correo electrónico.
- Si no se encuentra el usuario, muestra un mensaje en la consola de depuración (qDebug() << "Usuario no encontrado.";).

Explicación del siguiente método: on_logoutbtn_clicked

```
void user::on_logoutbtn_clicked()
{
    AppData& appData = AppData::getInstance();

    // Eliminar el árbol de publicaciones antes de hacer logout
    appData.getArbolDePublicaciones().eliminarArbol();

    MainWindow *ventana = new MainWindow(this);

    // Mostrar la ventana principal y ocultar la ventana actual
    this->hide();
    ventana->show();
}
```

Explicación de on_logoutbtn_clicked:

1. **Propósito**:

- Este método se ejecuta cuando el usuario hace clic en el botón "Cerrar sesión" (logout). Realiza las tareas necesarias para cerrar la sesión de usuario.

2. **Detalles del funcionamiento**:
    - Se accede a AppData y se elimina el árbol de publicaciones mediante el método eliminarArbol() del árbol binario que almacena publicaciones.
    - Se crea una nueva instancia de MainWindow, que es la ventana principal de la aplicación.
    - La ventana actual (diálogo de usuario) se oculta y se muestra la ventana principal.

Este método ayuda a limpiar la memoria de publicaciones almacenadas antes de cambiar de vista en la aplicación.

Método: on_modificarBtn_clicked

```
void user::on_modificarBtn_clicked()
{
    AppData& appData = AppData::getInstance();
    AVL& usuarios = appData.getAVLTree();

    QString nombre = ui->nameEdit->text();
    QString apellido = ui->apellidoEdit->text();
    QString correo = ui->correoEdit->text();
    QString password = ui->passEdit->text();
    QString fecha = ui->fechaEdit->text();

    usuarios.modifyUser(userCorreo.toStdString(), correo.toStdString(),
nombre.toStdString(), apellido.toStdString(), fecha.toStdString(),
password.toStdString());

    userCorreo = correo;
```

```
    this->close();


    MainWindow *ventana = new MainWindow(this);
    ventana->show();
}
```

Explicación de on_modificarBtn_clicked:

1. **Propósito**:
    - Permite modificar los datos del usuario (nombre, apellido, correo, contraseña, y fecha de nacimiento) cuando el usuario presiona el botón de modificar.
2. **Detalles del funcionamiento**:
    - Obtiene los nuevos valores de los campos de texto en la interfaz de usuario (ui->nameEdit, ui->apellidoEdit, ui->correoEdit, etc.).
    - Utiliza el método modifyUser() del árbol AVL para modificar los datos del usuario en el sistema. Aquí se pasan los valores nuevos convertidos a std::string.
    - Actualiza la variable userCorreo con el nuevo correo, si es que ha cambiado.
    - Cierra la ventana actual y abre la ventana principal (MainWindow).

Método: on_eliminarBtn_clicked

```
void user::on_eliminarBtn_clicked()
{
    AppData& appData = AppData::getInstance();
    AVL& usuarios = appData.getAVLTree();


    usuarios.deleteNode(userCorreo.toStdString());
```

```
    this->close();


    MainWindow *ventana = new MainWindow(this);
    ventana->show();
}
```

Explicación de on_eliminarBtn_clicked:

1. **Propósito**:
    - Elimina al usuario actual del sistema y cierra la sesión.
2. **Detalles del funcionamiento**:
    - Se accede al árbol AVL que contiene los usuarios a través de la instancia AppData.
    - El método deleteNode() elimina el usuario actual del árbol usando su correo electrónico (userCorreo).
    - Luego, se cierra la ventana actual y se regresa a la ventana principal de la aplicación (MainWindow).

Método: on_creaPubliBtn_clicked

```
void user::on_creaPubliBtn_clicked()
{
    crearPublicacion *ventana = new crearPublicacion(this, userCorreo);
    // No se ocultará la ventana actual
    ventana->show();
}
```

Explicación de on_creaPubliBtn_clicked:

1. **Propósito**:

- Abre una nueva ventana para crear una publicación, cuando se hace clic en el botón "Crear publicación".

2. **Detalles del funcionamiento**:
   - Se crea una nueva instancia de crearPublicacion, que es una ventana para que el usuario pueda realizar una publicación. Se pasa el correo del usuario (userCorreo) como argumento para identificar al usuario que está creando la publicación.
   - La ventana de crearPublicacion se muestra, pero la ventana actual (ventana de usuario) no se oculta.

Método: on_fechaBtn_clicked

```
void user::on_fechaBtn_clicked()
{
    mostrarPublicacionesEnScrollArea();
}
```

Explicación de on_fechaBtn_clicked:

1. **Propósito**:
   - Llama al método que muestra las publicaciones organizadas por fecha en un área de desplazamiento (QScrollArea), cuando se hace clic en el botón correspondiente.
2. **Detalles del funcionamiento**:
   - Invoca el método mostrarPublicacionesEnScrollArea(), que muestra las publicaciones almacenadas en el árbol binario de publicaciones. Este método es útil para que los usuarios puedan ver todas las publicaciones ordenadas por fecha.

Método: mostrarPublicacionesEnScrollArea

```
void user::mostrarPublicacionesEnScrollArea() {
```

```
    QVBoxLayout* layout = new QVBoxLayout();


    AppData& appData = AppData::getInstance();
    ArbolBinario& arbolBinario = appData.getArbolDePublicaciones();


    arbolBinario.recorrerPublicaciones([=](const string& fecha, NodoPublicacion*
publicacion) {
        while (publicacion) {
            QLabel* correoLabel = new QLabel(QString::fromStdString("Correo: " +
publicacion->correo));
            layout->addWidget(correoLabel);


            QLabel* contenidoLabel = new QLabel(QString::fromStdString("Contenido:
" + publicacion->contenido));
            layout->addWidget(contenidoLabel);


            QLabel* fechaHoraLabel = new QLabel(QString::fromStdString("Fecha: " +
fecha + " - Hora: " + publicacion->hora));
            layout->addWidget(fechaHoraLabel);


            if (!publicacion->imagen.empty()) {
                QLabel* imagenLabel = new QLabel();
                QPixmap pixmap(QString::fromStdString(publicacion->imagen));
                imagenLabel->setPixmap(pixmap.scaled(100, 100));
                layout->addWidget(imagenLabel);
            }


            publicacion = publicacion->siguiente;
        }
    });
```

```
    QWidget* contentWidget = new QWidget();
    contentWidget->setLayout(layout);
    ui->scrollPubli->setWidget(contentWidget);
}
```

Explicación de mostrarPublicacionesEnScrollArea:

1. **Propósito**:
    - Muestra las publicaciones almacenadas en un árbol binario en un área de desplazamiento (QScrollArea), organizadas por fecha y con los detalles de cada publicación (correo, contenido, fecha, hora, e imagen).
2. **Detalles del funcionamiento**:
    - Se crea un QVBoxLayout para organizar los elementos visuales.
    - Se accede al árbol binario de publicaciones mediante AppData.
    - El método recorrerPublicaciones() del árbol binario recorre todas las publicaciones almacenadas y para cada una:
        - Se crea un QLabel que muestra el correo del usuario que realizó la publicación.
        - Otro QLabel muestra el contenido de la publicación.
        - Un tercer QLabel muestra la fecha y la hora de la publicación.
        - Si la publicación contiene una imagen, se muestra utilizando un QPixmap en otro QLabel.
    - Finalmente, el layout con los QLabel se asigna a un QWidget, que a su vez se asigna al área de desplazamiento ui->scrollPubli.

Métodos: mostrarUsuariosEnTabla, enviarSolicitud, mostrarSolicitudesRecibidas, aceptarSolicitud, rechazarSolicitud, mostrarSolicitudesEnviadas, cancelarSolicitud, guardarPublicacionesEnArbol

Estos métodos gestionan la visualización y gestión de usuarios y solicitudes de amistad. Aquí están sus explicaciones clave:

1. mostrarUsuariosEnTabla:

- Muestra todos los usuarios en una tabla (QTableWidget), exceptuando al usuario actual y aquellos a los que ya les ha enviado una solicitud.
- Añade un botón para enviar una solicitud de amistad, y se conecta a la función enviarSolicitud.

2. enviarSolicitud:

- Envía una solicitud de amistad a otro usuario, añade la solicitud a la lista de amistades y a la pila de solicitudes.
- Actualiza la tabla de usuarios y solicitudes enviadas.

3. mostrarSolicitudesRecibidas:

- Muestra todas las solicitudes de amistad recibidas, permitiendo aceptar o rechazar cada solicitud.
- Los botones "Aceptar" y "Rechazar" están conectados a los métodos aceptarSolicitud y rechazarSolicitud.

4. aceptarSolicitud:

- Acepta una solicitud de amistad, eliminándola de la pila de solicitudes y de la lista de amistades.

5. rechazarSolicitud:

- Rechaza una solicitud de amistad, eliminándola de la pila y la lista.

6. mostrarSolicitudesEnviadas:

- Muestra todas las solicitudes de amistad enviadas por el usuario y permite cancelarlas mediante un botón conectado al método cancelarSolicitud.

7. cancelarSolicitud:

- Cancela una solicitud de amistad eliminándola de la lista de solicitudes y la pila.

8. guardarPublicacionesEnArbol:

- Transfiere las publicaciones de una lista doblemente enlazada al árbol binario de publicaciones para su almacenamiento.

AVL.H

1. Clase Node

La clase Node representa un nodo individual del árbol AVL. Cada nodo contiene:

- **Datos del usuario**: correo, nombre, apellido, fecha de nacimiento y contraseña.
- **Altura** del nodo: utilizada para controlar el balance del árbol.
- **Punteros a los hijos izquierdo y derecho**: estos son punteros inteligentes (shared_ptr) para gestionar automáticamente la memoria.

2. Clase AVL

Esta clase implementa las operaciones del árbol AVL, como inserciones, eliminaciones, búsqueda y recorridos. Se utiliza un puntero root para referenciar la raíz del árbol.

Funciones internas (privadas)

a) Métodos de Rotación

Para mantener el balance del árbol, se utilizan dos tipos de rotaciones:

- **Rotación hacia la derecha** (rightRotate): Se utiliza cuando un subárbol izquierdo tiene mayor altura de lo permitido.
- **Rotación hacia la izquierda** (leftRotate): Se usa cuando el subárbol derecho es más alto de lo permitido.

b) Altura y balance

- **height(node)**: Devuelve la altura de un nodo.
- **getBalance(node)**: Calcula el **factor de balance** de un nodo, que es la diferencia entre la altura de sus subárboles izquierdo y derecho. Un valor fuera de -1 a 1 indica que el árbol está desequilibrado.

c) Inserción

- **insertRec**: Inserta un nuevo nodo en el árbol de manera recursiva, comparando los correos electrónicos. Si al insertar el nodo se desequilibra el árbol, se aplican las rotaciones necesarias para mantener el balance del árbol.

d) Eliminación

- **deleteNodeRec**: Elimina un nodo basado en el correo electrónico. Este método sigue las reglas tradicionales de un árbol binario de búsqueda, y si el árbol queda desbalanceado tras la eliminación, se aplican rotaciones.

e) Búsqueda

- **searchRec**: Busca un nodo por correo electrónico de manera recursiva. Si encuentra el correo, devuelve un puntero al nodo.

f) Recorridos

El código proporciona varios tipos de recorridos:

- **Inorder** (inorderHelper): Recorre el árbol en orden creciente según el correo electrónico.
- **Preorder** (preorderHelper) y **Postorder** (postorderHelper): Recorren el árbol en preorden y postorden respectivamente, útiles para diferentes tipos de visualización o procesamiento de los nodos.

Funciones públicas

a) Inserción pública (insert)

Método público que inserta un nuevo nodo en el árbol llamando a la función recursiva insertRec.

b) Búsqueda de usuarios (search y getNode)

Permite buscar si un usuario existe mediante su correo electrónico. El método getNode devuelve un puntero al nodo encontrado, y search simplemente retorna true o false.

c) Verificar credenciales (verifyCredentials)

Comprueba si el correo y la contraseña proporcionados coinciden con los almacenados en un nodo.

d) Eliminación pública (deleteNode)

Elimina un nodo llamando a la función deleteNodeRec.

e) Recorridos públicos

Proporciona métodos públicos para realizar recorridos como:

- **inorder()**: Recorre el árbol y muestra el correo, nombre, apellido, fecha de nacimiento y contraseña de cada nodo usando qDebug().

f) Generación de archivo DOT (generateDot)

Este método genera un archivo .dot para visualizar el árbol AVL utilizando **Graphviz**. También crea una carpeta llamada reportes para almacenar el archivo de salida y genera una imagen .png del árbol.

Modificación de un nodo

- **modifyUser**: Permite modificar los datos de un usuario. Si se encuentra el usuario, se elimina su nodo y se inserta nuevamente con los datos actualizados. Si no se encuentra el nodo, muestra un mensaje de error usando QMessageBox.

MATRIX.H

Estructuras y clases principales

1. **NodeMatrix**: Es una estructura que representa cada nodo de la matriz. Cada nodo tiene:
    - i: la fila del nodo.
    - j: la columna del nodo.
    - value: el valor almacenado en la celda (puede ser, por ejemplo, una conexión entre usuarios).
    - Enlaces a los nodos adyacentes (arriba, abajo, izquierda y derecha).
2. **Matrix**: Esta es la clase principal que gestiona la matriz dispersa y tiene como propiedades:
    - root: el nodo raíz de la matriz, que sirve como origen para los encabezados de filas y columnas.
    - currentFriend: un puntero que se usa para iterar sobre las conexiones (como amigos) de un usuario en la matriz.

Métodos clave

1. **insert(i, j, value)**: Inserta un nuevo nodo en la matriz en la posición (i, j) con un valor. Si la fila o la columna no existen, crea encabezados de fila o columna. El nuevo nodo es conectado adecuadamente dentro de su fila y columna.
2. **searchRow(i)** y **searchColumn(j)**: Estos métodos buscan un encabezado de fila o columna correspondiente a i o j.

3. **nodeExists(newNode)**: Verifica si ya existe un nodo en la posición especificada. Si existe, actualiza su valor.
4. **insertRowHeader(i)** y **insertColumnHeader(j)**: Insertan nuevos encabezados de fila o columna si no existen.
5. **insertInRow(newNode, rowHeader)** y **insertInColumn(newNode, columnHeader)**: Estos métodos colocan un nuevo nodo en la posición adecuada dentro de su fila o columna.
6. **print()**: Imprime la matriz en un formato de tabla donde se muestran los valores almacenados y los encabezados de filas y columnas.
7. **generateDot(filename)**: Genera un archivo en formato DOT que describe la estructura de la matriz y luego usa Graphviz para crear una imagen (PNG) visualizando la matriz.
8. **startFriendIteration(person)**: Comienza una iteración sobre los "amigos" (conexiones) de un usuario dado. Esto se utiliza para recorrer las conexiones de una persona en la matriz.
9. **getNextFriend()**: Devuelve el siguiente amigo (conexión) en la iteración iniciada por startFriendIteration.
10. **printTopUsersWithLeastFriends(topN)**: Imprime los usuarios con menos conexiones o amigos, ordenando de menor a mayor número de conexiones. Esto se puede usar para identificar usuarios menos conectados en una red social.
11. **printTopUsersWithMostPosts(topN)**: Similar al método anterior, pero en lugar de amigos, imprime los usuarios con más publicaciones (datos) en la matriz.
12. **deleteNode(correo)**: Elimina un nodo de la matriz dado un identificador (correo). Si después de eliminar el nodo no queda nada en la fila o columna, también elimina los encabezados correspondientes.

Doublylinkedlist.h

Estructuras de los nodos

1. **Estructura node_comment (para comentarios)**:
    - Representa un nodo que contiene un comentario.
    - Campos:

        - id: Identificador único del comentario.
        - correo: Correo del autor del comentario.
        - comentario: El texto del comentario.
        - fecha: Fecha del comentario.
        - hora: Hora del comentario.
        - next: Puntero al siguiente comentario en la lista.
2. **Estructura node_publi (para publicaciones)**:
    - Representa una publicación en la lista.
    - Campos:
        - id: Identificador único de la publicación.
        - correo: Correo del autor de la publicación.
        - contenido_correo: Texto de la publicación.
        - fecha: Fecha de la publicación.
        - hora: Hora de la publicación.
        - imagenPath: Ruta a una imagen asociada con la publicación (si existe).
        - next: Puntero al siguiente nodo de publicación.
        - prev: Puntero al nodo anterior de la lista.
        - commentsHead: Puntero a la lista enlazada de comentarios asociados a esta publicación.

Clase DoublyLinkedList

Esta clase gestiona las publicaciones y sus comentarios. Veamos los elementos clave:

Atributos privados:

- head: Puntero al primer nodo de la lista.
- tail: Puntero al último nodo de la lista.
- nextId: Contador que asigna un ID único a cada nueva publicación.
- nextCommentId: Contador que asigna un ID único a cada nuevo comentario.

Métodos importantes:

1. **Constructor (DoublyLinkedList)**:
    - Inicializa la lista con head, tail como nullptr y los contadores de IDs en 0.
2. **append**:
    - Añade una nueva publicación al final de la lista.
    - Actualiza los punteros prev y next para mantener el doble enlace entre nodos.
3. **addComment**:
    - Añade un comentario a una publicación específica.
    - Busca la publicación por su id y agrega el comentario al final de la lista de comentarios de esa publicación.
4. **print**:
    - Imprime todas las publicaciones junto con sus comentarios en la consola. Si una publicación tiene una imagen asociada, también muestra la ruta de la imagen.
5. **generateDot**:
    - Genera un archivo .dot que representa gráficamente la lista doblemente enlazada y luego lo convierte a un archivo PNG mediante Graphviz.
    - Llama a generateDotHelper que recorre la lista y escribe las conexiones entre nodos en formato DOT.
6. **agregarPublicacionesAlArbol**:
    - Agrega las publicaciones del usuario actual a un árbol binario (de la clase ArbolBinario). Recorre la lista y agrega solo las publicaciones cuyo correo coincida con el del usuario.
7. **Destructor (~DoublyLinkedList)**:
    - Libera la memoria asociada a la lista de publicaciones y sus comentarios al finalizar el uso de la lista.

LISTASIMPLE.H

Estructura del nodo node_amistad

El nodo node_amistad contiene la información sobre una solicitud de amistad:

- **Campos:**
    - emisor: Correo de la persona que envió la solicitud.
    - receptor: Correo de la persona que recibe la solicitud.
    - next: Puntero al siguiente nodo en la lista (el próximo nodo de solicitud de amistad).

El constructor de la estructura inicializa estos campos.

Clase ListaAmistad

Esta clase administra la lista simple de solicitudes de amistad, proporcionando diversas funcionalidades.

Atributos privados:

- head: Puntero al primer nodo de la lista, es decir, la cabeza de la lista.

Métodos:

1. **Constructor (ListaAmistad)**:
    - Inicializa la lista poniendo head como nullptr, indicando que la lista está vacía inicialmente.
2. **append**:
    - Añade una nueva solicitud de amistad al final de la lista.
    - Si la lista está vacía, el nuevo nodo se convierte en la cabeza.
    - Si no está vacía, se recorre la lista hasta el final y se inserta el nuevo nodo allí.
3. **display**:
    - Recorre la lista e imprime en la consola todas las solicitudes de amistad en el formato "Emisor -> Receptor".
4. **deleteNode**:
    - Elimina un nodo de la lista que coincida con una solicitud de amistad específica (según el emisor y receptor).
    - Si la solicitud está en el primer nodo, se actualiza la cabeza de la lista.
    - Si la solicitud está en medio o al final de la lista, se actualizan los punteros de los nodos para eliminar el nodo deseado.
5. **dot_solicitudes_enviadas**:

- Genera un archivo DOT que representa gráficamente las solicitudes enviadas por un usuario específico.
- El archivo DOT se convierte automáticamente en una imagen PNG usando Graphviz.
- Solo las solicitudes en las que el emisor coincide con el usuario buscado se incluyen en la visualización.

6. **existeSolicitud**:
    - Verifica si ya existe una solicitud de amistad entre dos correos específicos (emisor y receptor).
    - Se realiza una búsqueda en la lista para encontrar cualquier coincidencia.
    - Retorna true si encuentra una solicitud, false si no.

grafoNdirigido.h

Clase SubNodoAdyacencia

Esta clase representa un nodo en la lista de amigos de un usuario.

- **correoAmigo**: almacena el correo del amigo.
- **siguiente**: apunta al siguiente nodo en la lista de amigos.
- **SubNodoAdyacencia(const string& correo)**: constructor que inicializa el correoAmigo.

Clase NodoAdyacencia

Representa un usuario y su lista de amigos.

- **correoUsuario**: correo del usuario.
- **siguiente y anterior**: punteros al siguiente y anterior NodoAdyacencia, formando una lista doblemente enlazada.
- **listaAmigos**: puntero a la lista de amigos del usuario.

Métodos:

- **agregarAmigo(const string& correoAmigo)**: agrega un amigo a listaAmigos.
- **existeAmigo(const string& correoAmigo)**: verifica si un correo ya existe en listaAmigos.

- **imprimirAmigos()**: imprime la lista de amigos del usuario.

Clase NodoSugerencia

Este nodo representa una sugerencia de amistad.

- **email**: correo del amigo sugerido.
- **amigosEnComun**: número de amigos en común.
- **siguiente**: apunta al siguiente nodo en ListaSugerencia.

Clase ListaSugerencia

Es una lista enlazada ordenada por número de amigos en común.

- **head**: puntero al primer nodo.

Métodos:

- **insertarOrdenado(const string& email, int amigosEnComun)**: inserta una sugerencia ordenada de mayor a menor por amigosEnComun.
- **mostrar()**: muestra todas las sugerencias.
- **obtenerSugerencias()**: devuelve una cadena con las sugerencias.

Clase ListaAdyacencia

Representa el grafo de usuarios.

Atributos

- **cabeza y cola**: apuntan al inicio y final de la lista de usuarios.

Métodos

- **buscarNodo(const string& correoUsuario)**: busca un NodoAdyacencia por correoUsuario.
- **sugerirAmigos(const string& correoUsuario)**: sugiere amigos de amigos para el usuario especificado.
    - Recorre los amigos directos del usuario.
    - Busca amigos de los amigos y los agrega como sugerencias si no son amigos directos.
- **generarCadenaAmigos()**: genera una cadena con todos los usuarios y sus amigos en formato usuario: amigo1, amigo2, ...
- **esAmigoDirecto(const string& correoUsuario, const string& correoAmigo)**: verifica si dos usuarios son amigos directos.

- **esAmigoDeUsuario(const string& correoUsuario, const string& correoAmigo)**: revisa si un amigo ya está en la lista del usuario.
- **buscarSugerencia(ListaSugerencia& lista, const string& correoAmigo)**: busca una sugerencia existente.
- **agregarRelacion(const string& correo1, const string& correo2)**: agrega una relación de amistad entre dos usuarios. Si uno o ambos usuarios no existen, los crea y luego agrega la relación en ambas direcciones.
- **imprimirRelaciones()**: imprime todas las relaciones de amistad.
- **graph()**: genera un archivo .dot para visualizar el grafo y lo convierte a imagen usando dot.
- **graph_listaEnlazada()**: genera el archivo .dot de la lista enlazada de usuarios y amigos y lo convierte en imagen.
- **graph_user(const string& correoUsuario)**: genera un grafo específico para un usuario y sus amigos (con colores para diferenciar los niveles de relación).

Este código es útil para crear una red social básica, mostrar las relaciones y sugerir nuevos amigos en base a amigos de amigos.

BlockChain.h

1. Estructura de la Clase Block

La clase Block representa cada bloque de la cadena y contiene:

- **Atributos**:
    - index: Índice del bloque en la cadena.
    - timestamp: Marca de tiempo del bloque cuando se crea.
    - data: Datos que contiene el bloque (específicamente publicaciones en este caso).
    - previousHash: Hash del bloque anterior.
    - hash: Hash propio del bloque.
    - nonce: Contador para cumplir con la dificultad de minería.
- **Métodos**:

- **Constructor**: Recibe el índice, los datos, y el hash del bloque anterior. Si es el bloque génesis (índice 1), se ajusta el previousHash a "0000". También calcula el hash inicial del bloque con calculateHash().
- **getCurrentTime**: Obtiene la marca de tiempo en formato DD-MM-YYYY::HH:MM:SS y la devuelve como un string.
- **saveToJSON**: Guarda los detalles del bloque en un archivo JSON. Crea una carpeta llamada bloques si no existe y guarda cada bloque como block_index.json con sus datos (índice, marca de tiempo, nonce, hash previo y hash propio).
- **calculateHash**: Calcula el hash SHA-256 del bloque, incluyendo index, timestamp, data, previousHash, y nonce mediante QCryptographicHash.
- **mineBlock**: Ejecuta la minería del bloque buscando un hash que comience con una cantidad de ceros igual a la dificultad (proceso de "proof of work"). Incrementa el nonce hasta encontrar un hash válido.

2. Clase Blockchain

La clase Blockchain actúa como contenedor de los bloques, enlazándolos en una cadena mediante una lista enlazada.

- **Estructura Interna (Node)**: Cada nodo contiene un puntero a un Block y un puntero al siguiente nodo.
- **Atributos**:
  - head y tail: Apuntan al primer y último nodo de la lista enlazada.
  - difficulty: Define la dificultad para la minería de bloques (es decir, el número de ceros que debe tener el hash al inicio).
- **Métodos**:
  - **Constructor**: Inicializa una cadena creando el bloque génesis (bloque inicial con previousHash = "0000") y llama a mineBlock con la dificultad especificada.
  - **addBlock**: Agrega un bloque nuevo a la lista enlazada. Si es el primer bloque, head y tail se apuntan a este nodo. Si no, el bloque se añade al final, y tail se actualiza.
  - **createNewBlock**: Genera un nuevo bloque. Calcula el índice y el hash del bloque previo, crea el bloque, lo mina y lo guarda en un archivo JSON antes de añadirlo a la cadena.

- **Destructor**: Libera la memoria asignada a cada bloque y nodo, recorriendo la lista enlazada desde el primer bloque hasta el último.

Flujo General del Código

1. **Inicialización**: La Blockchain crea un bloque génesis y lo mina.
2. **Creación de Bloques**: Con createNewBlock, cada bloque se genera con datos específicos y se enlaza en la cadena.
3. **Minería**: Cada bloque se mina, ajustando el nonce hasta que el hash cumple con el requisito de dificultad.
4. **Almacenamiento**: Cada bloque minado se guarda como archivo JSON dentro de una carpeta llamada bloques.
5. **Destrucción**: Cuando el objeto Blockchain se elimina, libera todos los recursos de los bloques y nodos almacenados en memoria.

Huffman.h

1. Clases Internas y Atributos

Clase Nodo

La clase Nodo representa un nodo en el árbol de Huffman:

- **Atributos**:
  - caracter: El carácter en el nodo (si es un nodo hoja).
  - frecuencia: Frecuencia del carácter en el texto.
  - siguiente: Puntero al siguiente nodo en la lista enlazada.
  - izquierdo y derecho: Hijos izquierdo y derecho en el árbol.

Clase ListaEnlazadaHuff

La clase ListaEnlazadaHuff organiza los nodos en una lista enlazada ordenada según la frecuencia de cada carácter.

- **Atributos**:
  - cabeza: Puntero al primer nodo de la lista.
- **Métodos**:
  - insertarEnOrden: Inserta un nodo en la lista en orden de frecuencia.

- extraerMinimo: Extrae el nodo con la frecuencia más baja.
- soloUno: Devuelve true si solo queda un nodo en la lista (indica que el árbol de Huffman está completo).
- obtenerCabeza: Retorna el nodo cabeza de la lista.

Atributos de la Clase Huffman

- root: Puntero a la raíz del árbol de Huffman.
- tablaCaracteres[256]: Tabla de códigos binarios generados para cada carácter.
- tamanioCodigo[256]: Tamaño del código binario para cada carácter.

2. Métodos Principales de la Clase Huffman

Método construirArbol

Construye el árbol de Huffman a partir del texto dado:

- **Paso 1**: Cuenta la frecuencia de cada carácter en el texto.
- **Paso 2**: Crea nodos para cada carácter con frecuencia > 0 y los inserta en una lista enlazada (ListaEnlazadaHuff).
- **Paso 3**: Combina nodos con frecuencias más bajas para formar el árbol de Huffman.
- **Paso 4**: Al finalizar, el único nodo en la lista enlazada es la raíz del árbol de Huffman (root).

Método generarCodigos

Genera códigos binarios para cada carácter a partir del árbol de Huffman, almacenando cada código en tablaCaracteres y su tamaño en tamanioCodigo.

- Recorre el árbol de manera recursiva.
- Al llegar a un nodo hoja (carácter), guarda el código correspondiente y su tamaño.

Método codificar

Convierte el texto en su versión comprimida usando los códigos de Huffman.

- Recorre cada carácter del texto y lo reemplaza con su código binario correspondiente en tablaCaracteres.

Método decodificar

Reconstruye el texto original a partir de una secuencia de bits.

- Recorre los bits del texto codificado, navegando el árbol de Huffman desde la raíz. Si llega a un nodo hoja, añade el carácter al texto decodificado.

3. Funciones Públicas

comprimir

Este método principal recibe un texto, construye el árbol de Huffman, genera los códigos, y devuelve el texto comprimido.

descomprimir

Convierte el texto comprimido de regreso a su forma original usando el árbol de Huffman ya construido.

guardarArchivoComprimido

Guarda el texto comprimido y la tabla de códigos en un archivo:

- Escribe en el archivo cada carácter y su código binario.
- Luego añade el texto comprimido como una secuencia de bits.

descomprimirArchivo

Lee el archivo comprimido, extrae el contenido y lo descomprime.

- Lee el contenido del archivo en contenidoArchivo y lo pasa al método comprimir.

4. Manejo de Memoria

- **Destructores**: Se liberan todos los nodos del árbol y de la lista enlazada al final de la ejecución.

Flujo General del Código

1. **Compresión**:
    - comprimir crea un árbol de Huffman y genera códigos para cada carácter.
    - guardarArchivoComprimido guarda los códigos y el texto comprimido en un archivo.
2. **Descompresión**:
    - descomprimirArchivo lee el archivo comprimido y restaura el texto original.