

Universidad de San Carlos de Guatemala

Laboratorio de Estructura de Datos

Sección A

MANUAL TECNICO

Esdras Eliab Bautista Guerra


























202100301

El programa se desarrollo con el lenguaje de programación C++ , con el GUI QT Creator, para poder hacerlo mas interactivo para el usuario.

- Clases utilizadas:

- Clases .h:

- ▼ Header Files

-  admin.h
 -  arbolB.h
 -  buscar.h
 -  buscaradmin.h
 -  cargaradmin.h
 -  comentaru.h
 -  crearpublicacion.h
 -  crearuser.h
 -  GestionarSoli.h
 -  listaAmistades.h
 -  listaEnlazadaArbol.h
 -  listaPublicaciones.h
 -  listasoli_e.h
 -  listasoli_R.h
 -  mainwindow.h
 -  modificaradmin.h
 -  nodoMatriz.h
 -  nodoPublicacion.h
 -  perfil.h
 -  PilaSoli_R.h
 -  publicaciones.h
 -  reportesadmin.h
 -  SimplePublicacion.h
 -  solicitudes.h
 -  userint.h

- Clases .cpp:

- ▼ Source Files
 - admin.cpp
 - arbolB.cpp
 - buscar.cpp
 - buscaradmin.cpp
 - cargaradmin.cpp
 - comentaru.cpp
 - crearpublicacion.cpp
 - crearuser.cpp
 - GestionarSoli.cpp
 - listaAmistades.cpp
 - listaEnlazadaArbol.cpp
 - listaPublicaciones.cpp
 - listasoli_e.cpp
 - listasoli_R.cpp
 - main.cpp
 - mainwindow.cpp
 - modificaradmin.cpp
 - nodoMatriz.cpp
 - nodoPublicacion.cpp
 - perfil.cpp
 - PilaSoli_R.cpp
 - publicaciones.cpp
 - reportesadmin.cpp
 - SimplePublicacion.cpp
 - solicitudes.cpp
 - userint.cpp

- **Clase listaEnlazadaArbol.h:** Tenemos lo que es nuestro árbol binario de búsqueda (ABB o BST) en el cual vamos a guardar a todos los usuarios con el correo como llave, así mismo tenemos ciertos métodos los cuales ayudaran a que el árbol este equilibrado, en dado caso no lo este se realizan las rotaciones respectivas.

```
using namespace std;

class nodoArbol{
private:
    string nombres;
    string apellidos;
    string correo;
    string contrasena;
    string fechaNacimiento;
    nodoArbol *derecha, *izquierda;
    PilaSolicitudesRecibidas* pilaSolicitudesRecibidas; // Pila de solicitudes recibidas
    ListaSolicitudesEnviadas* listadeSolicitudesEnviadasUsuario; //Lista de solicitudes enviadas
    listaPublicaciones* listaPublicacionesU;
    listaAmistad* listaAmigos;
    ArbolBST* ArbolPublicacionesBST;

    listaNodoPub* listaTodasPubs;
    int altura;
public:
    nodoArbol(string n, string a, string c, string pwd, string fNac);
    nodoArbol();
    ~nodoArbol();
    int getAltura();
    void setAltura(int altura);
    string getNombres();
    string getApellidos();
    string getCorreo();
    string getContrasena();
    string getFechaNacimiento();

    void setNombres(string nombres);
    void setApellidos(string apellidos);
    void setCorreo(string correo);
    void setContrasena(string contrasena);
    void setFechaNacimiento(string fechaNacimiento);

    void setDerecha(nodoArbol *derecha);
    void setIzquierda(nodoArbol *izquierda);
    nodoArbol* getDerecha();
    nodoArbol* getIzquierda();
    PilaSolicitudesRecibidas* getPilaSolicitudesRecibidas(); // Getter para la pila de solicitudes
    ListaSolicitudesEnviadas* getListadeSolicitudesEnviadas(); // Getter para la lista de solicitudes que el usuario ha enviado
    listaPublicaciones* getListapublicacionesU();
    listaAmistad* getListamigos();
    ArbolBST* getArbolPublicacionesBST();

    listaNodoPub* getListatodasPubs();
};
```

En nuestro nodoArbol, tenemos ciertos atributos importantes como lo son sus datos principales y así mismo tenemos una pila y listas para cada nodo, para así poder manejar de forma más eficiente la información para cada nodo.

Y así mismo en la clase .cpp procedemos a desarrollar los métodos, los getters, setter, constructores y destructores.

- **Clase listaPublicaciones.h:** Tenemos lo que es nuestro AVL, en el cual guardaremos todas las fechas de publicaciones, y cada nodo tendrá una lista enlazada que tendrá las publicaciones echas en esa fecha y cada nodo de las publicaciones tendrá un árbol B que es el árbol donde se almacenan los comentarios de cada publicación.

```
#ifndef LISTAPUBLICACIONES_H
#define LISTAPUBLICACIONES_H

#include "SimplePublicacion.h"
#include <fstream>
#include <sstream>
#include <iostream>
#include <cstdlib>
#include <string>
#include <ctime>
#include <iomanip>
#include "nodoMatriz.h"
#include "PilaSoli_R.h"
#include "ListaSoli_E.h"
#include "Listasoli_R.h"

using namespace std;
class nodoBST{
private:
    string fecha;
    nodoBST *drcha, *izq;
    listaNodoPub *publicacionesBST;
public:
    nodoBST();
    nodoBST(string fecha);
    void setDrcha(nodoBST *der);
    void setgetIzq(nodoBST *izq);
    string getfecha();
    listaNodoPub* getPublicacionesBST();
    nodoBST* getDrcha();
    nodoBST* getIzq();
};
```

```
class ArbolBST{
private:
    nodoBST* raiz;
    nodoBST* agregarPublicacionBST(nodoBST* raiz, string correo, string contenido, string fecha, string hora, string imagen, int contador);
    bool verificarexistefecha(nodoBST* raiz, int contador, string fecha, string correo);
    void eliminarPublicacion(nodoBST* raiz, int contador, string fecha, string correo);
    nodoBST* buscarNodoPorFecha(nodoBST* raiz, string fecha);

    void postOrden(nodoBST *raiz, bool accion);
    void preOrden(nodoBST *raiz);
    void inOrden(nodoBST *raiz);
    void graph(nodoBST *raiz, std::ofstream &f);

    std::tm convertirFecha(string fecha) {
        std::tm tm = {};
        std::istringstream ss(fecha);
        ss >> std::get_time(&tm, "%d/%m/%Y"); // Formato d/m/y
        if (ss.fail()) {
            throw std::runtime_error("Error al convertir la fecha");
        }
        return tm;
    }

    void OrdenInorden(nodoBST *raiz, nodoSimplePub* &cabeza, nodoSimplePub* &actual, int &contador, int cantidadMaxima);
    void OrdenPostorden(nodoBST *raiz, nodoSimplePub* &cabeza, nodoSimplePub* &actual, int &contador, int cantidadMaxima);
    void OrdenPreorden(nodoBST *raiz, nodoSimplePub* &cabeza, nodoSimplePub* &actual, int &contador, int cantidadMaxima);
public:
    ArbolBST();
    ~ArbolBST();
    void agregarPublicacionBST(string correo, string contenido, string fecha, string hora, string imagen, int contador);
    bool verificarexistefecha(int contador, string fecha, string correo);
    void eliminarPublicacion(int contador, string fecha, string correo);

    void preOrden();
    void inOrden();
    void postOrden();
    void graph();
    nodoBST* getRaiz();
    nodoBST* buscarNodoPorFecha(string fecha);

    nodoSimplePub* OrdenInorden(int cantidadMax);
    nodoSimplePub* OrdenPostorden(int cantidadMax);
    nodoSimplePub* OrdenPreorden(int cantidadMax);

    bool compararFechas(const std::string& f1, const std::string& f2) {
        std::tm tm1 = convertirFecha(f1);
        std::tm tm2 = convertirFecha(f2);
        return std::difftime(std::mktime(&tm1), std::mktime(&tm2)) < 0;
    }
};
```

Aquí en la parte del árbol implementamos un método para poder pasar las fechas a un formato con el cual podamos compararla con otras fechas, y así poder ir insertando como se deben los nodos en nuestro árbol.

- **Clase SimplePublicacion.h:** Se implementó una lista doblemente enlazada, para poder ver los comentarios siguientes y anteriores al momento que el usuario vea su feed, este nodo tiene sus atributos necesarios, pero hay uno que resalta y es el árbol B, cada publicación tiene su propio árbol B, el cual es de comentarios.

```
#ifndef SIMPLEPUBLICACION_H
#define SIMPLEPUBLICACION_H

#include <fstream>
#include <sstream>
#include <iostream>
#include <string>
#include "arbolB.h"
using namespace std;

class nodoSimplePub{
private:
    string fecha;
    string correo;
    string contenido;
    string hora;
    string imagen;
    int id;
    nodoSimplePub* siguiente;
    nodoSimplePub* anterior;
    ArbolB *ArbolComentarios;

public:
    nodoSimplePub(string fecha,string correo,string contenido,string hora,string imagen,int id);
    nodoSimplePub();

    string getFechaL();
    string getCorreoL();
    string getContenidoL();
    string getHoral();
    string getImagen();
    int getIdL();
    void setSiguiente(nodoSimplePub* sig);
    void setAnterior(nodoSimplePub* ant);
    nodoSimplePub* getSiguiente();
    nodoSimplePub* getAnterior();

    ArbolB* getArbolComentarios();
};

class listaNodoPub{
private:
    nodoSimplePub* cabeza;

public:
    listaNodoPub();
    ~listaNodoPub();

    void setCabeza(nodoSimplePub* nuevaCabeza) {
        this->cabeza = nuevaCabeza;
    }
    void agregarPublicacionL(string fecha,string correo,string contenido,string hora,string imagen,int id);
    void mostrarPublicacionesL();
    bool verificarExistencia(string correo,int id);
    nodoSimplePub* getCabeza();
    void eliminarPub(string correo);
};

#endif // SIMPLEPUBLICACION_H
```

En nuestra lista implementamos los métodos necesarios para la inserción correcta de los nodos, el método verificarExistencia verifica que la publicación a agregar no este agregada ya anteriormente, esto solo para prevenir posibles problemas, el método agregarPublicacionL la cual agrega una publicación, un nodo, a la lista doblemente enlazada.

- Las clases pila, listasoli_e , nodoMatriz y nodoPublicacion siguen siendo las mismas con los mismos métodos que la fase 1.
- **Clase arbolB:** Se implemento un árbol B, en el cual se almacenan los comentarios de cada publicación

```

#ifndef ARBOLB_H
#define ARBOLB_H
#include <fstream>
#include <sstream>
#include <iostream>
#include <stdlib.h>
#include <string>
#include <ctime>
#include <iomanip>
using namespace std;
class Llave;
class Nodo;
class NodoB{
private:
    bool hoja;
    Llave *primero;
    int numeroLlaves;
    std::tm convertirFecha(string fecha) {
        std::tm tm = {};
        std::istringstream ss(fecha);
        ss >> std::get_time(&tm, "%d/%m/%Y"); // Formato d/m/y
        if (ss.fail()) {
            throw std::runtime_error("Error al convertir la fecha");
        }
        return tm;
    }
    std::tm convertirHora(const std::string& hora) {
        std::tm tm = {};
        std::istringstream ss(hora);
        ss >> std::get_time(&tm, "%H:%M"); // Formato H:M:S
        if (ss.fail()) {
            throw std::runtime_error("Error al convertir la hora");
        }
        return tm;
    }
public:
    NodoB();
    ~NodoB();
    void insertarLlave(Llave *llave);
    Llave* getPrimero();
    int getNumeroLlaves();
    bool esHoja();
    void setPrimero(Llave *llave);
    void setNumeroLlaves(int numeroLlaves);
    void setHoja(bool hoja);
    int compararFechas(const std::string& f1, const std::string& f2) { ... }
    int compararHoras(const std::string& h1, const std::string& h2) { ... }

class Llave{
private:
    string fecha;
    string hora;
    string correo;
    string contenido;

    Llave *prev;
    Llave *sig;
    Nodo *izq;
    Nodo *drcha;
public:
    Llave(string fecha,string hora,string correo,string contenido);
    ~Llave();

    bool tieneHijos();

    string getFecha();
    string getHora();
    string getCorreo();
    string getContenido();

    Llave* getPrev();
    Llave* getSig();
    Nodo* getIzq();
    Nodo* getDrcha();

    void setFecha(string fecha);
    void setHora(string hora);
    void setContenido(string contenido);
    void setCorreo(string correo);
    void setPrev(Llave*);
    void setSig(Llave*);
    void setIzq(Nodo*);
    void setDrcha(Nodo*);
};

```

```

class ArbolB {
private:
    Nodo *raiz;
    int orden;

    Llave* insertarEnHoja(string fecha,string hora,string correo,string contenido,Nodo *raiz);
    Llave* dividir(Nodo *nodo);

    std::tm convertirFecha(string fecha) { ... }
    std::tm convertirHora(const std::string& hora) { ... }

public:
    ArbolB();
    ~ArbolB();

    void insert(string fecha,string hora,string correo,string contenido);

    int compararFechas(const std::string& f1, const std::string& f2) { ... }
    int compararHoras(const std::string& h1, const std::string& h2) { ... }
};

#endif // ARBOLB_H

```

- Clase gestionarSoli: En esta clase se gestiona todos los procesos del usuario y unos del administrador:

```

#ifndef GESTIONARSOLI_H
#define GESTIONARSOLI_H
#include "listaEnlazadaArbol.h"

#include <QWidget>
#include <QLineEdit>
#include <QPlainTextEdit>
#include <QLabel>
#include <QPushButton>
#include <QTableWidget>

class gestionarSoli{
public:
    static void enviarSolicitud(listaEnlazadaArb &usuarios,string correoEmisor, string correoReceptor,QWidget *ventSolicitudes);
    static void aceptarSolicitud(listaEnlazadaArb &usuarios,string correoEmisor,string correoReceptor,QWidget *ventSolicitudes);
    static void rechazarSolicitud(listaEnlazadaArb &usuarios,string correoEmisor,string correoReceptor,QWidget *ventSolicitudes);
    static void aceptarSolicitudD(listaEnlazadaArb &usuario, string correoE, string correoR,QWidget *ventAdmin); // cuando es carga desde admin
    static void cancelarSolicitud(listaEnlazadaArb &usuario, string micorreo, string elotrocorreo,QWidget *ventSolicitudes);

    static void EliminarCuenta(listaEnlazadaArb &usuario, string correo);
    static void verAmigos(listaEnlazadaArb &usuario, string correo);

    static void agregarPublicacionesAmigos(listaEnlazadaArb &usuarios, string correo);
    static void agregarPublicacionesArbol(listaEnlazadaArb &usuarios, string correo);
    static nodoSimplePub* mostrarPublicacionesArbol(listaEnlazadaArb &usuarios, string correo, QLineEdit* txtUsuario, QLineEdit* txtFecha, QPlainTextEdit* textoPub, QLabel* lblImagen,string
    static nodoSimplePub* OrdenarPublicacionespor(listaEnlazadaArb &usuarios, string correo, QLineEdit* txtUsuario, QLineEdit* txtFecha, QPlainTextEdit* textoPub, QLabel* lblImagen,int ca

    static void agregarComentario(nodoSimplePub &publicaciones,string fecha,string hora,string contenido,string correoComento,QWidget *ventana);

    static void agregarPubDesdeAdmin(listaEnlazadaArb &usuarios,string correo,listaPublicaciones &listaPubs,string contenido,string fecha,string hora);
};

#endif // GESTIONARSOLI_H

```

Tenemos los métodos de enviar solicitud, aceptar solicitud, rechazar solicitud, cancelar solicitud, eliminar cuenta, ver amigos, agregar publicaciones de amigos, agregar publicaciones árbol, mostrar publicaciones árbol, ordenar publicaciones por, estas son los métodos que tiene esta clase para gestionar cada usuario y sus acciones o solicitudes, mientras que los métodos para gestión de administrador son aceptar solicitud D y agregar publicación admin.


```

void gestionarSoli::enviarSolicitud(listaEnlazadaArb &usuarios, string correoEmisor, string correoReceptor, QWidget *ventSolicitudes){
    try{
        nodoArbol* receptor = usuarios.buscarNodoPorCorreoArb(correoReceptor);
        nodoArbol* emisor = usuarios.buscarNodoPorCorreoArb(correoEmisor);
        listaAmistad* miAmigo = emisor->getListaAmigos();

        if(miAmigo->verificarAmistad(correoReceptor)){
            QString msg = QString::fromStdString("El usuario " + correoReceptor + " ya es tu amigo.");
            QMessageBox::warning(ventSolicitudes, "Error al enviar solicitud", msg);
        }else{
            if(receptor == nullptr){
                QString msg = QString::fromStdString("El usuario "+correoReceptor+" No existe!");
                QMessageBox::warning(ventSolicitudes, "Error al enviar solicitud", msg);
                return;
            }
            if (correoEmisor == correoReceptor) {
                QMessageBox::warning(ventSolicitudes, "Error al enviar solicitud", "No puedes enviarte solicitud a ti mismo!");
            }else{
                PilaSolicitudesRecibidas* pilaReceptor = receptor->getPilaSolicitudesRecibidas();
                ListaSolicitudesEnviadas* listaEmisor = emisor->getListaSolicitudesEnviadas();
                PilaSolicitudesRecibidas* pilaEmisor = emisor->getPilaSolicitudesRecibidas();

                if(pilaEmisor->existe(correoReceptor)){
                    QString msg = QString::fromStdString("No se pudo enviar la solicitud, ya existe una solicitud pendiente de parte de: "+correoReceptor);
                    QMessageBox::warning(ventSolicitudes, "Error al enviar solicitud", msg);
                }else{
                    if(!pilaReceptor->existe(correoEmisor) && !listaEmisor->existe(correoReceptor)){
                        pilaReceptor->push(correoEmisor);
                        listaEmisor->agregar(correoReceptor, "PENDIENTE");
                        QString msg = QString::fromStdString("Solicitud enviada de: "+ correoEmisor + " para: "+correoReceptor);
                        QMessageBox::information(ventSolicitudes, "Solicitud Enviada con Exito!", msg);
                    }else{
                        QMessageBox::warning(ventSolicitudes, "Error al enviar solicitud", "No se pudo enviar la solicitud, ya existe una solicitud pendiente");
                    }
                }
            }
        }
    }
}

}

void gestionarSoli::aceptarSolicitud(listaEnlazadaArb &usuarios, string correoEmisor, string correoReceptor, QWidget *ventSolicitudes){
    try{
        nodoArbol* receptor = usuarios.buscarNodoPorCorreoArb(correoReceptor); //yo
        nodoArbol* emisor = usuarios.buscarNodoPorCorreoArb(correoEmisor); //el que me envia la sol
        if(receptor == nullptr){
            QMessageBox::warning(ventSolicitudes, "Error", "Usuario no encontrado!!");
            return;
        }

        PilaSolicitudesRecibidas* pilaReceptor = receptor->getPilaSolicitudesRecibidas();
        ListaSolicitudesEnviadas* listaEmisor = emisor->getListaSolicitudesEnviadas();

        listaAmistad* listaAmigosEmisor = emisor->getListaAmigos();
        listaAmistad* listaAmigosReceptor = receptor->getListaAmigos();

        if (!pilaReceptor->estaVacia()) {
            pilaReceptor->eliminarElemento(correoEmisor);
            listaEmisor->eliminar(correoReceptor);
            QString msg = QString::fromStdString("Solicitud de: " +correoEmisor+ " Aceptada!");
            QMessageBox::information(ventSolicitudes, "Estado Solicitud",msg);

            //pilaReceptor->verPila();

            listaAmigosEmisor->agregarAmigo(correoReceptor);
            listaAmigosReceptor->agregarAmigo(correoEmisor);
        }
    }
}

}

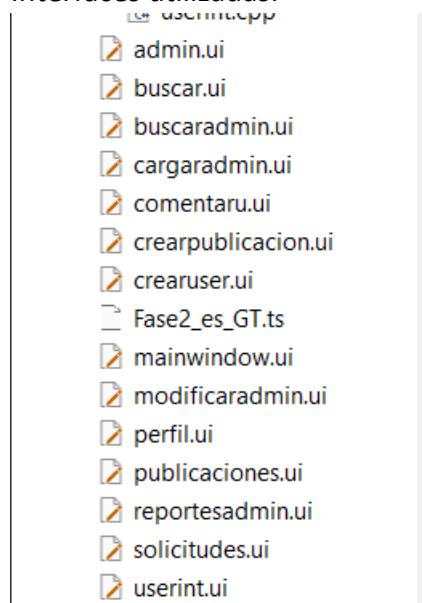
```

Métodos para enviar solicitud:

Este método evalúa primero si ya son amigos, si ya son amigos retorna un mensaje, de lo contrario si no son amigos, procede a evaluar si ese usuario al que se le quiere mandar solicitud no le ha enviado ya solicitud o si el no ha mandado ya la solicitud, ya que no se puede enviar dos veces una solicitud a una persona y no se le puede enviar solicitud a un usuario que ya te ha mandado solicitud.

Y se cumple eso retorna un mensaje de advertencia, y si no se cumple, procede a agregar a la pila de la persona a quien se le manda solicitud el correo y a nuestra lista de solicitudes el correo de ese usuario.

- Interfaces utilizadas:



Se utilizaron estas clases .ui las cuales son las ventanas de interfaz que el usuario visualizara, cada una con su respectivo nombre de lo que hace, la ventana principal es mainwindow.ui donde empezara todo el programa.

- Cada reporte fue realizado con graphviz y mostrado en un label tanto para el usuario como para el administrador, ejemplo de graphviz para mostrar el árbol de usuarios (Árbol ABB):

```
void listaEnlazadaArb::graph(){
    std::ofstream outfile ("AVL.dot");
    outfile << "digraph G {" << std::endl;

    if(raiz != nullptr){
        graph(this->raiz, outfile);
    }

    outfile << "}" << std::endl;
    outfile.close();

    int returnCode = system("dot -Tpng ./AVL.dot -o ./AVL.png");

    if(returnCode == 0){std::cout << "Command executed successfully." << std::endl; }
    else{std::cout << "Command execution failed or returned non-zero: " << returnCode << std::endl;}
}
```

```

void listaEnlazadaArb::graph(nodoArbol *raiz, std::ofstream &f){
    if(raiz != nullptr){
        std::stringstream oss;
        oss << raiz;
        std::string nombre = oss.str();

        f << "Nodo" + nombre + "[label = \"" + raiz->getCorreo() + "\"]" << std::endl;

        if(raiz->getIzquierda() != nullptr){
            oss.str("");
            oss << raiz->getIzquierda();
            std::string izquierda = oss.str();
            f << "Nodo" << nombre + "->Nodo" + izquierda << std::endl;
        }

        if(raiz->getDerecha() != nullptr){
            oss.str("");
            oss << raiz->getDerecha();
            std::string derecha = oss.str();
            f << "Nodo" << nombre + "->Nodo" + derecha << std::endl;
        }

        this->graph(raiz->getIzquierda(), f);
        this->graph(raiz->getDerecha(), f);
    }
}

```

Ya que es un árbol lo recorremos recursivamente, así en ese orden vamos a ir nombrando nodos y haciendo las conexiones, hacemos un recorrido PreOrden. Y al finalizar generamos un documento .dot y una imagen .png.

- **Relación Amistad**

En la clase de relacionAmistad.h tenemos lo siguiente:

```

1  #ifndef RELACIONAMISTAD_H
2  #define RELACIONAMISTAD_H
3
4  #include <iostream>
5  #include <cstdlib>
6  #include <string>
7  #include <fstream>
8  #include <sstream>
9
10 using namespace std;
11 class enodo;
12 //-----
13
14 //grafo vertice (emisor)
15 class vnodo{
16 private:
17     string correo;
18     vnodo *siguiente;
19     enodo *destinos; //el primero de la lista de destinos
20
21 public:
22     vnodo();
23     ~vnodo();
24
25     void insertarDestino(string);
26     void graficarAristas(std::ofstream&);
27     void graficarListaDestinos(std::ofstream&);
28
29     string getCorreoE();
30     vnodo* getSiguiete();
31     enodo* getDestinos();
32     void setCorreoE(string);
33     void setSiguiete(vnodo*);
34     void setDestinos(enodo*);
35 };
36
37

```

En esta parte tenemos la clase para el vértice del nodo, en donde contendrá de forma privada un string correo y dos apuntadores, uno de tipo vértice nodo(vnodo) el cual será “siguiente” y uno de destino de vértice destino (enodo).

De forma pública tenemos el constructor, y unos métodos los cuales nos ayudarán a graficar y a insertar vértice destino, y sus getter y setters.

```

//-----
// grafo destino (Receptor)
class enodo{
private:
    string correoDestino;
    enodo *siguiente;

public:
    enodo();

    string getDestino();
    enodo* getSiguiete();
    void setDestino(string);
    void setSiguiete(enodo*);
};
//

```

También tenemos la clase enodo, el cual es el destino del nodo, como el vértice destino, en donde de forma privada tenemos dos cosas, el correo del destino y un apuntador de siguiente de tipo enodo.

Así mismo de forma pública tenemos el constructor y los getter y setters correspondientes.

```
// Lista de Adyacencia (Creacion de conexiones)
class listaAdyacencia{
private:
    vnodo *cabeza;

public:
    listaAdyacencia();
    ~listaAdyacencia();

    void insert(string);
    bool verificarAmistadGrafo(string);
    void eliminarAmigoGrafo(string);
    void crearConexion(string, string);
    bool crearGrafo();
    bool crearGrafoLista();
    vnodo* getprimero();
};
```

También tenemos la clase listaAdyacencia, la cual como el mismo nombre lo dice es para crear la lista de adyacencia con respecto a las amistades que tenga cada usuario, en donde de forma privada tendremos únicamente la cabeza de tipo vnodo (vértice nodo) para poder empezar a recorrer la lista. Así mismo tenemos de forma pública lo que es el insert, la verificación de amistad, la eliminación del grafo, la creación de conexión y la graficación.

En la parte del .cpp tenemos:

Para la clase enodo: Inicializamos el constructor, así creamos los métodos de getter y setter

```
#include "../relacionAmistad.h"

//-----
enodo::enodo(){
    this->correoDestino = "";
    this->siguiente = nullptr;
}

string enodo::getDestino(){
    return correoDestino;
}

enodo* enodo::getSiguiente(){
    return siguiente;
}

void enodo::setDestino(string destino){
    this->correoDestino = destino;
}

void enodo::setSiguiente(enodo* siguiente){
    this->siguiente = siguiente;
}
//-----
```

Para la clase vnodes: tenemos la inicialización del constructor y de los getter y setter y así mismo del destructor.

```
vnodes::vnodes(){
    this->correo = "";
    this->destinos = nullptr;
    this->siguiente = nullptr;
}

string vnodes::getCorreoE(){
    return this->correo;
}

enodo* vnodes::getDestinos(){
    return this->destinos;
}

vnodes* vnodes::getSiguiente(){
    return this->siguiente;
}

void vnodes::setCorreoE(string data){
    this->correo = data;
}

void vnodes::setDestinos(enodo *destinos){
    this->destinos = destinos;
}

void vnodes::setSiguiente(vnodes *siguiente){
    this->siguiente = siguiente;
}

vnodes::~vnodes(){
    enodo *aux = this->destinos;
    enodo *temp;

    while(aux != nullptr){
        temp = aux->getSiguiente();
        delete aux;
        aux = temp;
    }
}
```

```

void vnodo::insertarDestino(string destino){
    enodo *nuevo = new enodo();
    nuevo->setDestino(destino);

    if(this->destinos == nullptr){
        this->destinos = nuevo;
    }else{
        enodo *aux = this->destinos;
        while(aux->getSiguiente() != nullptr){
            if(aux->getDestino() == destino){
                return;
            }
            aux = aux->getSiguiente();
        }

        if(aux->getDestino() != destino){
            aux->setSiguiente(nuevo);
        }
    }
}

void vnodo::graficarAristas(std::ofstream &f){
    enodo *aux = this->destinos;

    std::string nombre_origen = this->correo;
    while(aux != nullptr){
        if(this->correo < aux->getDestino()){
            std::string nombre_destino = aux->getDestino();
            f << nombre_origen + "->" + nombre_destino + "[dir = both];" << std::endl;
        }
        aux = aux->getSiguiente();
    }
}

```

También tenemos la implementación de los métodos, los cuales serían la inserción de un nodo destino en donde lo insertamos como si fuera una lista simplemente enlazada.

La graficación de las aristas, en donde mediante un ciclo while recorreremos toda nuestra lista de nodos y los vamos graficando, mientras lo recorremos, usando Graphviz.

```

void vnodo::graficarListaDestinos(std::ofstream &f){
    enodo *aux = this->destinos;
    std::ostringstream oss;
    std::string valor = this->correo;
    std::string nombre_origen = "Nodo" + valor;
    std::string rank = "{rank=same;" + nombre_origen;

    oss << this->destinos;
    std::string nombre = oss.str();

    f << nombre_origen + "->Nodo" + nombre + ";" << std::endl;

    while(aux != nullptr){
        oss.str("");
        oss << aux;
        std::string nombre = oss.str();

        std::string nombre_destino = "Nodo" + nombre;
        f << nombre_destino + "[label = \"" + aux->getDestino() + "\" fillcolor = \"white\"];\" << std::endl;
        rank+=";" + nombre_destino;

        if(aux->getSiguiente() != nullptr){
            oss.str("");
            oss << aux->getSiguiente();
            std::string nombre = oss.str();
            f << nombre_destino + "->Nodo" + nombre + ";" << std::endl;
        }
        aux = aux->getSiguiente();
    }
    rank+= "};";
    f << rank << std::endl;
}

```

Hacemos lo mismo para graficar los nodos destino.

Para la clase listaAdyacencia: inicializamos el constructor, implementamos los métodos del destructor, de la inserción (insert)

```
listaAdyacencia::listaAdyacencia(){
    this->cabeza = nullptr;
}

listaAdyacencia::~~listaAdyacencia(){
    vnodon *aux = this->cabeza;
    vnodon *temp;

    while(aux != nullptr){
        temp = aux->getSiguiente();
        delete aux;
        aux = temp;
    }
}

void listaAdyacencia::insert(string vertice){
    if(this->cabeza == nullptr){
        this->cabeza = new vnodon();
        this->cabeza->setCorreoE(vertice);
    }else{
        vnodon *nuevo = new vnodon();
        nuevo->setCorreoE(vertice);

        if(vertice < this->cabeza->getCorreoE()){
            nuevo->setSiguiente(cabeza);
            this->cabeza = nuevo;
        }else{
            vnodon *aux = this->cabeza;
            while(aux->getSiguiente() != nullptr){
                if(vertice < aux->getSiguiente()->getCorreoE()){
                    nuevo->setSiguiente(aux->getSiguiente());
                    aux->setSiguiente(nuevo);
                    break;
                }
                aux = aux->getSiguiente();
            }

            if(aux->getSiguiente() == nullptr){
                aux->setSiguiente(nuevo);
            }
        }
    }
}
```



```

void listaAdyacencia::crearConexion(string origen, string destino){
    v nodo *aux = this->cabeza;
    while(aux != nullptr){
        if(aux->getCorreoE() == origen){
            aux->insertarDestino(destino);
            break;
        }
        aux = aux->getSiguiente();
    }
}

bool listaAdyacencia::crearGrafo(){
    std::ofstream outfile("grafo.dot");
    if (!outfile.is_open()) {
        std::cerr << "Error al abrir el archivo grafo.dot" << std::endl;
        return false;
    }

    outfile << "digraph G {" << std::endl;

    v nodo *aux = this->cabeza;
    while(aux != nullptr) {
        std::string valor = aux->getCorreoE();
        std::string dec_nodo = valor + "[label = \"" + valor + "\"]";
        outfile << dec_nodo << std::endl;
        aux->graficarAristas(outfile);
        aux = aux->getSiguiente();
    }

    outfile << "}" << std::endl;
    outfile.close();

    int returnCode = system("dot -Tpng ./grafo.dot -o ./grafo.png");

    if (returnCode == 0) {
        std::cout << "Command executed successfully." << std::endl;
        return true;
    } else {
        std::cout << "Command execution failed or returned non-zero: " << returnCode << std::endl;
        return false;
    }
}

```

El método para crear conexión, nos sirve para claramente crear conexión entre un nodo y otro, hacer esa conexión mediante una arista(viéndolo desde un grafo).

El método crearGrafo prácticamente nos grafica mediante Graphviz la estructura de amistad de forma de un grafo, con sus vértices y nodos.

```

bool listaAdyacencia::crearGrafoLista(){
    std::ofstream outfile("grafoLista.dot");
    if (!outfile.is_open()) {
        std::cerr << "Error al abrir el archivo grafoLista.dot" << std::endl;
        return false;
    }

    outfile << "digraph G {" << std::endl;
    outfile << "node[shape = \"box\" style = \"filled\"]" << std::endl;

    v nodo *aux = this->cabeza;
    while(aux != nullptr) {
        std::string valor = aux->getCorreoE();
        std::string nombre = "Nodo" + valor + "[label = \"" + valor + "\" group = \"1\" fillcolor=\"lightgray\"]";
        outfile << nombre << std::endl;

        if(aux->getSiguiente() != nullptr) {
            outfile << "Nodo" + valor + "->Nodo" + aux->getSiguiente()->getCorreoE() + "[dir = none]";
        }

        aux->graficarListaDestinos(outfile);
        aux = aux->getSiguiente();
    }

    outfile << "}" << std::endl;
    outfile.close();

    int returnCode = system("dot -Tpng ./grafoLista.dot -o ./grafoLista.png");

    if (returnCode == 0) {
        std::cout << "Command executed successfully." << std::endl;
        return true;
    } else {
        std::cout << "Command execution failed or returned non-zero: " << returnCode << std::endl;
        return false;
    }
}

```

El método de crearGrafoLista es para crear el mismo grafo pero de una forma diferente, de una forma de una lista adyacente.

```

bool listaAdyacencia::verificarAmistadGrafo(string correoVerificar){
    if(cabeza == nullptr){
        return false;
    }

    vnod* actual = cabeza;
    while(actual != nullptr){
        if(actual->getCorreoE() == correoVerificar){
            return true;
        }
        actual = actual->getSiguiente();
    }

    return false;
}

vnodo* listaAdyacencia::getprimero(){
    if(cabeza != nullptr){
        return cabeza;
    }else{
        return nullptr;
    }
}

```

El método verificarAmistadGrafo es utilizado para ver si ya son amigos o no, en donde se recorre cada nodo vértice y si coinciden los correos retorna un true, de lo contrario retorna un false.