Universidad San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Estructura de Datos

MANUAL TECNICO

Victor Eduardo Franco Suret

202001767

21 de Agosto de 2022

OBJETIVO DE ESTE MANUAL

El presente manual técnico tiene como finalidad describir aspectos técnicos y la forma en la que está estructurado, así como la interacción del usuario con este. El código elaborado para recrear una versión de escritorio que emula el funcionamiento de una red social en la cual se implementan diversas estructuras de datos para poder garantizar su funcionamiento óptimo.

DIRIGIDO

El uso de este manual es exclusivo para programadores o diseñadores que desean saber más sobre el funcionamiento técnico de la aplicación. Para la comprensión de este manual se debe tener conocimientos previos sobre el lenguaje de programación C++.

CONOCIMIENTOS PREVIOS

Los conocimientos mínimos que deben tener las que utilizaran este manual son:

- Conocimientos sobre el Lenguaje C++

-Uso de Apuntadores, Ciclos , Listas dinámicas

-Conocimientos Generales en el uso de las librerías Graphviz y Nlohmann(para archivos json)

ESPECIFICACIONES TECNICAS

Sistema Operativo: Windows 11

Lenguaje de Programación: c++.

IDE: Se utilizo Visual Studio Code

Tener instalado Graphviz y Nlomann.

INICIO

El menú principal de la aplicación esta localizado en el archivo Main.cpp, en este se incluye el archivo ListaUsuarios.cpp que es donde se trabaje la gran parte de la aplicación. Se cuenta con las opciones de Inicio de sesión, registro, información.

Por defecto existe el usuario administrador:
Usuario: admin@gmail.com

Contraseña: EDD2S2024

Al ingresar con este usuario se accederá al menú de Administrador.

Se declaran distintas variables globales que son instancia de estructuras de datos (Listas y Matriz).

Se hace uso de un ciclo Do-While para el menú.

SOLICITUDES DE AMISTAD

Para las solicitudes de amistad se maneja una Pila para el usuario receptor y una lista simple para el usuario emisor.

Tomando en cuenta que un usuario no puede enviar solicitud a uno que previamente envio solicitud.

Para el manejos de las solicitudes se cuenta con su Struct en el archivo Estructs.cpp:

```
struct Solicitud {
    Usuario* emisor;            // Usuario que envió la solicitud
    std::string correoReceptor;  // Correo del usuario que recibió la solicitud
    Solicitud* siguiente;        // Puntero a la siguiente solicitud en la lista

    Solicitud(Usuario* emisor, const std::string& correoReceptor)
        : emisor(emisor), correoReceptor(correoReceptor), siguiente(nullptr) {}
};
```

Contiene un puntero hacia el usuario que envia la solicitud (emisor) y el correo del receptor, también un puntero que apunta a la siguiente solicitud en una lista simple.

FLUJO DE TRABAJO PARA SOLICITUDES

Cuando un usuario envia una solicitud de amista a otro, se crea una instancia de estas estructura, la solicitu de agrega a la lista de solicitudes del usuario emisor y a la pila de solicitudes recibidas del receptor.

```
void enviarSolicitudAmistad(const std::string& correoEmisor, const std::string& correoReceptor) {
    Usuario* emisor = obtenerUsuario(correoEmisor);
    Usuario* receptor = obtenerUsuario(correoReceptor);

    if (!emisor || !receptor) {
        std::cout << "Usuario no encontrado.\n";
        return;
    }

    // Verificar si ya existe una solicitud pendiente en ambas direcciones
    if (solicitudPendiente(emisor, receptor) || solicitudPendiente(receptor, emisor)) {
        std::cout << "Ya existe una solicitud pendiente entre estos usuarios.\n";
        return;
    }
```

Un usuario puede revisar las solicitudes de amistad que ha recibido mediante una función que despliega las solicitudes almacenadas en la pila. se tiene la función de aceptar y rechazar esta solicitud.

```
void verSolicitudesAmistad(Usuario* usuario) {
    if (usuario->solicitudesRecibidas.empty()) {
        std::cout << "No tienes solicitudes de amistad.\n";
        return;
    }

    std::stack<Solicitud*> tempStack;
    int index = 1;
    std::cout << "Solicitudes de amistad recibidas:\n";
    while (!usuario->solicitudesRecibidas.empty()) {
        Solicitud* solicitud = usuario->solicitudesRecibidas.top();
        usuario->solicitudesRecibidas.pop();
        tempStack.push(solicitud);
        std::cout << index << ". De: " << solicitud->emisor->nombres << " " << solicitud->emisor->apellidos
                  << " (" << solicitud->emisor->correo << ")\n";
        index++;
    }

    // Aceptar o rechazar solicitud seleccionada
```

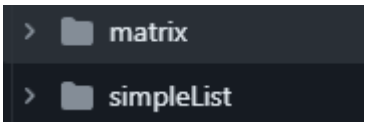
Las solicitudes se extraen de la pila una a una y se muestran al usuario.

Si el usuario decide aceptar una solicitud, se actualiza la matriz de amistades (se hablara mas delante de esta) para reflejar la nueva relación.

Si el usuario rechaza una solicitud, la solicitud se elimina. Esta también se elimina si decide aceptarla. Tanto de la lista como de la pila.

MATRIZ DE RELACIONES DE AMISTAD

Para el manejo de las relaciones de amistad se hizo uso de una Matriz dispersa (carpetas simpleList y Matrix)



La clase Matrix es la estructura principal que gestiona las relaciones de amistad entre usuarios. Utiliza dos listas enlazadas (SimpleList), una para las filas (rowHeader) y otra para las columnas (colHeader), donde cada nodo de estas listas representa un usuario. Dentro de la matriz, cada celda ocupada (que indica una relación de amistad) se representa mediante un nodo de la clase MatrixNode.

Cada MatrixNode representa una relación de amistad entre dos usuarios específicos. Tiene punteros (up, bottom, right, left) que lo conectan a otros nodos en la matriz, formando una estructura bidimensional dispersa.

ListNode representa cada nodo en las listas de filas y columnas, y tiene un puntero a un nodo de la matriz , que representa el primer nodo en esa fila o columna.

SimpleList maneja las listas de filas y columnas, permitiendo la inserción de nuevos nodos (usuarios) y manteniendo un enlace entre ellos.

Inserción de Relaciones de Amistad

La función insertarAmistad permite agregar una relación de amistad entre dos usuarios. Esta función llama a insert para colocar dos nodos en la matriz, uno en la posición [correo1][correo2] y otro en [correo2][correo1], dado que la amistad es una relación bidireccional(se apunta de ambos lados).

```
void Matrix::insertarAmistad(const std::string& correo1, const std::string& correo2) {
    int col = obtenerIndice(correo1);
    int row = obtenerIndice(correo2);

    insert(col, row, "Amistad");
    insert(row, col, "Amistad");
}
```

Inserción en la Matriz

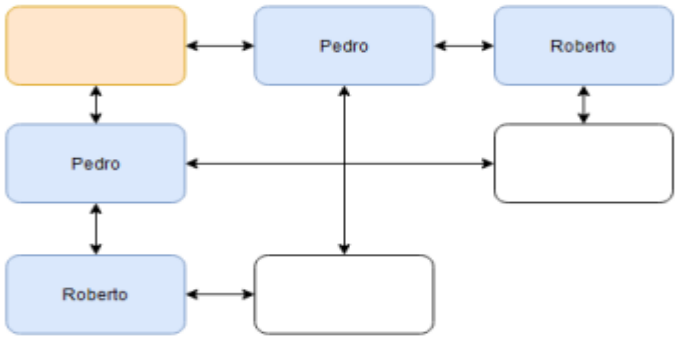
La función insert realiza la inserción de un MatrixNode en la matriz. Primero se asegura de que existan nodos en la lista de filas y columnas para las posiciones correspondientes. Luego, inserta el nodo de la matriz en la posición adecuada, manteniendo el orden correcto en las listas enlazadas.

```cpp
void Matrix::insert(int col, int row, const std::string& value) {
    ListNode *colNode = colHeader->insert(col);
    ListNode *rowNode = rowHeader->insert(row);

    MatrixNode *newMatrixNode = new MatrixNode(col, row, value);

    move_lr_pointers(rowNode, newMatrixNode);
    move_ub_pointers(colNode, newMatrixNode);
}
```

Manejo de Punteros

Las funciones move_lr_pointers y move_ub_pointers manejan los punteros para insertar el MatrixNode en el lugar correcto dentro de la lista enlazada. move_lr_pointers Inserta el nodo en la fila y move_ub_pointers Inserta el nodo en la columna

Estas funciones aseguran que la matriz dispersa esté organizada, y que mantenga la estructura consistente.



Verificación de Amistad

Para verificar si un usuario es amigo de otro es utiliza la fuicon sonAmigos verifica que 2 correos se encuentren apuntando uno hacia el otro.

```cpp
bool Matrix::sonAmigos(const std::string& correo1, const std::string& correo2) {
    int col = obtenerIndice(correo1);
    int row = obtenerIndice(correo2);

    MatrixNode* nodo = rowHeader->head->access;
    while (nodo != nullptr) {
        if (nodo->col == col && nodo->row == row) {
            return true;
        }
        nodo = nodo->right;
    }

    return false;
}
```

Generación de Grafico

Para la generación del Grafico de esta matriz se utiliza la función créate_dot esat genera un archivo .DOT que posteriormente se convierte en una imagen con la librería Graphviz.

```
void Matrix::create_dot() {
    string code = "digraph G{\n";
    code += "  node[shape=box];\n";
    code += "  MTX[ label = \"Matrix\", style = filled, fillcolor = firebrick1, group = 0 ];\n";

    code += get_content();
    code += "}\n";

    write_dot(code);
}
```

get_content genera el contenido que describe la estructura de la matriz, incluyendo los nodos y sus relaciones.

Destructor:

El destructor de Matrix y SimpleList se encargan de liberar la memoria dinámica asignada a los nodos de la matriz y las listas, evitando fugas de memoria.

```
Matrix::~Matrix() {
    delete rowHeader;
    delete colHeader;
}
```

PUBLICACIONES

Para el manejo de publicaciones se lleva a cabo utilizando dos clases principales: ListaPublicaciones y ListaCircularPublicaciones.

```
class ListaPublicaciones {
private:


public:
Publicacion* cabeza;
    Publicacion* cola;
    ListaPublicaciones() : cabeza(nullptr), cola(nullptr) {}

    void agregarPublicacion(const std::string& correo, const std::string& contenido) {
        Publicacion* nuevaPublicacion = new Publicacion(correo, contenido);
        if (cabeza == nullptr) {
            cabeza = cola = nuevaPublicacion;
        } else {
            cola->siguiente = nuevaPublicacion;
            nuevaPublicacion->anterior = cola;
            cola = nuevaPublicacion;
        }
    }
```

```
class ListaCircularPublicaciones {

public:
    NodoPublicacion* cabeza;
    ListaCircularPublicaciones() : cabeza(nullptr) {}

    void insertarPublicacion(Publicacion* pub) {
        NodoPublicacion* nuevoNodo = new NodoPublicacion(pub);
        if (cabeza == nullptr) {
            cabeza = nuevoNodo;
            cabeza->siguiente = cabeza;
            cabeza->anterior = cabeza;
        } else {
            NodoPublicacion* ultimo = cabeza->anterior;
            ultimo->siguiente = nuevoNodo;
            nuevoNodo->anterior = ultimo;
            nuevoNodo->siguiente = cabeza;
            cabeza->anterior = nuevoNodo;
        }
    }
```

La primera lista se encarga de almacenar todas las publicaciones del programa, en cambio la lista circular esta condicionada a que cuando un usuario se logee solo podrá visualizar las publicaciones propias y de su rama de amistad.

La función agergarPublicacion crea una nueva publicación y la agrega al final de la lista, Si la lista está vacía, la nueva publicación se convierte tanto en la cabeza como en la cola de la lista.

Si la lista ya contiene publicaciones, la nueva publicación se enlaza al final de la lista, ajustando los punteros siguiente y anterior de las publicaciones.

Para que el usuario logeado pueda ver las publicaciones filtradas se hace uso de la función verPublicaciones, esta se encarga de filtrar las publicaciones de listaPublicaciones y las muestra y recorre con la lista circular.

```
void filtrarPublicacionesParaUsuario(ListaPublicaciones& listaPublicaciones, Usuario* usu
    Publicacion* actual = listaPublicaciones.getCabeza();

    while (actual != nullptr) {
        // Verificar si la publicación pertenece al usuario logeado o a un amigo
        if (actual->correoUsuario == usuarioLogeado->correo || matrizAmistades.sonAmigos(
```

CARGA MASIVA:

La carga masiva se realiza con archivos .Json. para poder utilizar archivos Json en C++ utilizamos la librería nlohmann.

Se hace uso de distintas funciones de carga.

En estas se apretura el archivo, también se lee, se procesan los datos y dependiendo de el tipo de estructura se cargara un usuario, una relación de amistad o una publicación.

```
void cargarMasivaPublicaciones(const std::string& path) {
    std::ifstream archivo(path);
    if (!archivo.is_open()) {
        std::cerr << "No se pudo abrir el archivo " << path << std::endl;
        return;
    }

    nlohmann::json jsonData;
    archivo >> jsonData;

    for (const auto& item : jsonData) {
        std::string correo = item["correo"];
        std::string contenido = item["contendio"];
        std::string fecha = item["fecha"];
        std::string hora = item["hora"];

            Publicacion* nuevaPublicacion = new Publicacion(correo, contenido);
            nuevaPublicacion->fecha = fecha;
            nuevaPublicacion->hora = hora;
            listaPubli.agregarPublicacion(correo, contenido);
            std::cout << "Publicación de " << correo << " cargada exitosamente." << std::endl;

    }
}
```

Funcionamiento:

Apertura del Archivo JSON:

Intenta abrir el archivo JSON y, si falla, muestra un mensaje de error.

Lectura del Contenido :

Se carga el contenido del archivo JSON .

Procesamiento de Cada Solicitud:

La función recorre cada solicitud en el JSON, verificando que contenga los campos, dependiendo del tipo de dato que cargara variara.

Si faltan campos, se muestra un mensaje de error.

REPORTES

Para el apartado de reportes se hace uso de creación de grarficos mediante la generación de archivos Dot que luego se convierten en imágenes.

Se genera el archivo .Dot con los datos que son necesarios dependiendo del tipo de grafico.

Al crearse genera el archivo con un nombre predeterminado, al momento de generarse también se generara automáticamente la imagen conjunta para visualizar este archivo.

```
void graficarListaUsuarios(Usuario* cabeza) {
    std::ofstream file("lista_usuarios.dot");

    if (!file.is_open()) {
        std::cerr << "Error al abrir el archivo para escribir el grafo.\n";
        return;
    }

    file << "digraph G {\n";
    file << "    node [shape=record];\n";

    Usuario* actual = cabeza;
    int index = 0;

    while (actual != nullptr) {
        // Crear un identificador único para cada nodo basado en su posición en la lista
        std::string nodeName = "node" + std::to_string(index);
```