

# Manual Técnico

## Social Structure

Fecha: 26 de octubre de 2024

## Contenido

OBJETIVO.....	3
ALCANCE .....	3
DESARROLLO DEL MANUAL TÉCNICO .....	4
Características Principales .....	4
Tecnologías utilizadas .....	4
Requisitos del sistema.....	5
Instalación .....	5
Estructura del proyecto.....	5
Estructuras utilizadas .....	6
Listas Enlazadas .....	6
Lista doble .....	7
Pila .....	7
Árbol AVL.....	8
Árbol BST .....	8
Árbol B.....	9
Lista de adyacencia.....	9
Método de Huffman .....	10

## OBJETIVO

El manual técnico proporciona una guía detallada para desarrolladores, administradores de sistemas y otros usuarios técnicos sobre cómo instalar, configurar y utilizar Social Structure. Contiene información sobre la estructura del código, las dependencias del proyecto, las instrucciones de implementación, y la explicación de los diferentes componentes y su funcionamiento.

## ALCANCE

El proyecto "Social Structure" es un sistema de gestión de redes sociales que permite la administración y visualización de usuarios, solicitudes de amistad, y publicaciones. Está desarrollado en C++ y hace uso de varias estructuras de datos, como listas enlazadas simples, listas enlazadas dobles, pilas, árboles AVL, BST, B, lista de adyacencia y método de Huffman para organizar y gestionar la información de manera eficiente.

# DESARROLLO DEL MANUAL TÉCNICO

## Características Principales

- **Gestión de Usuarios:** Los usuarios pueden registrarse, iniciar sesión, y ver su perfil. Los administradores tienen la capacidad de cargar usuarios desde archivos JSON, así como eliminar usuarios existentes.
- **Solicitudes de Amistad:** Los usuarios pueden enviar y recibir solicitudes de amistad. Las solicitudes recibidas se almacenan en pilas asociadas a cada usuario receptor y se pueden gestionar desde su perfil. Asimismo, las solicitudes enviadas se almacenan en listas simples del usuario emisor.
- **Publicaciones:** Los usuarios pueden crear, visualizar, y filtrar publicaciones. Las publicaciones se gestionan utilizando listas enlazadas dobles y árboles binarios de búsqueda BST, lo que permite una organización eficiente y la posibilidad de reportes.
- **Reportes:** El sistema permite generar reportes en dentro de la interfaz gráfica para visualizar la estructura de datos de los usuarios, sus relaciones, y publicaciones. Además, los administradores pueden ver un Top 5 de los usuarios con más publicaciones.
- **Menú Administrativo:** Los administradores tienen acceso a un menú exclusivo donde pueden gestionar usuarios, relaciones, y publicaciones a través de cargas masivas desde archivos JSON.
- **Autenticación:** El sistema incluye un mecanismo de autenticación para garantizar que solo los usuarios registrados y el administrador tengan acceso a sus respectivas funciones.
- **Compresión y descompresión de usuarios:** El sistema permite la persistencia de usuarios cuando la aplicación se vuelve a inicializar.

## Tecnologías utilizadas

- **C++:** El proyecto está desarrollado principalmente en lenguaje C++, aprovechando las bibliotecas nativas para implementar la funcionalidad requerida.
- **Graphviz:** Se utiliza la biblioteca Graphviz para generar gráficos visuales que representan las mascotas y su estado.
- **Qt:** IDE utilizado para ejecutar la interfaz gráfica.

## Requisitos del sistema

- Compilador C++
- Sistema operativo compatible con C++
- Graphviz
- Qt

## Instalación

Debido a que es una aplicación que se ejecuta en consola, se debe clonar o descargar el repositorio del proyecto desde [https://github.com/angelygm03/-EDD-Proyecto\\_202210483.git](https://github.com/angelygm03/-EDD-Proyecto_202210483.git) para luego abrir la carpeta Fase 3 del proyecto en el IDE de su preferencia (se recomienda Qt) y correr la aplicación.

## Estructura del proyecto

El proyecto sigue una estructura de directorios típica de C++

Fase 1

Fase 2

- Main.cpp: Script principal para ejecutar la aplicación.
- Mainwindow: Ventana del login
- Adminwindow: Ventana de administrador
- Userwindow: Ventana de usuario
- AVLTree: Estructura para almacenar a los usuarios
- BinarySearchTree: Estructura que almacena las publicaciones de un usuario
- DoubleList: Estructura que almacena las publicaciones globalmente
- Stack: Estructura que almacena las solicitudes recibidas
- Solicitud List: Estructura que almacena las solicitudes enviadas
- Friend List: Estructura que almacena las relaciones de amistad
- Fecha List: Estructura que almacena las publicaciones por fecha

Fase 3

- Main.cpp: Script principal para ejecutar la aplicación.
- Mainwindow: Ventana del login
- Adminwindow: Ventana de administrador
- Userwindow: Ventana de usuario
- AVLTree: Estructura para almacenar a los usuarios
- BinarySearchTree: Estructura que almacena las publicaciones de un usuario

- DoubleList: Estructura que almacena las publicaciones globalmente
- Stack: Estructura que almacena las solicitudes recibidas
- Solicitud List: Estructura que almacena las solicitudes enviadas
- Adjacency List: Estructura que almacena las relaciones de amistad
- Huffman: Método que se encarga de comprimir y descomprimir la información de los usuarios.

## Estructuras utilizadas

### Listas Enlazadas

El proyecto ha implementado el uso de tres listas enlazadas simples, una para almacenar las publicaciones por fecha, otra para las relaciones de amistad y la última para las solicitudes de amistad enviadas.

```

1  class SolicitudNode {
2  public:
3      string emisor;
4      string receptor;
5      string estado;
6      SolicitudNode* next;
7
8      SolicitudNode(string e, string r, string s) : emisor(e), receptor(r), estado(s), next(nullptr) {}
9  };
10
11 class SolicitudList {
12 private:
13
14 public:
15     SolicitudNode* head;
16     int size;
17     SolicitudList();
18     void insert(string emisor, string receptor, string estado);
19     void print();
20     bool eliminar(const string& emisor, const string& receptor);
21     SolicitudNode* buscar(const string& emisor, const string& receptor);
22     void generateDotFile(const string& fileName);
23     bool eliminarPorEmisorYReceptor(const string& emisor, const string& receptor);
24 };
25

```

## Lista doble

Como se mencionó anteriormente, esta estructura almacena las publicaciones de forma global en todo el sistema, solamente el usuario administrador puede acceder a ella.

```
1 class DoubleList {
2 private:
3     NodeDoubleList* head;
4     NodeDoubleList* tail;
5     int size;
6
7 public:
8     DoubleList();
9
10    void insertAtBeginning(string correo, string contenido, string fecha, string hora, string imagenPath);
11    void insertAtEnd(const string& correo, const string& contenido, const string& fecha, const string& hora, const string& imagenPath);
12    void insertAtPosition(int givenPosition, string correo, string contenido, string fecha, string hora, string imagenPath);
13    void print() const;
14    bool deleteAtPosition(int position);
15    void printByUser(const string& correoUsuario) const;
16    bool deleteByUser(int position, const string& correoUsuario);
17    void generateDotFile();
18    void printTopUsersByPublications() const;
19 };
```

## Pila

La pila fue una buena elección para poder mostrarle las solicitudes pendientes al usuario, de esta manera cuando el usuario quiera ver y aprobar las solicitudes se le muestran primero las últimas solicitudes entrantes.

```
1 class StackNode {
2 public:
3     string emisor;
4     string receptor;
5     string estado;
6     StackNode* next;
7
8     // Constructor del nodo
9     StackNode(string emisor, string receptor, string estado);
10
11 };
```

## Árbol AVL

Este árbol almacena los usuarios del sistema teniendo como criterio el correo electrónico de la A-Z.

```
1 class Node {
2 public:
3     string nombres;
4     string apellidos;
5     string fechaNacimiento;
6     string correo;
7     string contrasena;
8     int factor;
9     Node* left;
10    Node* right;
11    Stack solicitudes; //pila de solicitudes recibidas
12    SolicitudoList solicitudListEnviadas; //lista de solicitudes enviadas
13    FriendList friends; //lista de amigos
14    BinarySearchTree* publicaciones; //árbol de publicaciones
15
16    // Constructor para inicializar el nodo
17    Node(string nombres, string apellidos, string fechaNacimiento, string correo, string contrasena);
18
19    // Destructor para liberar memoria
20    ~Node();
21 };
```

## Árbol BST

Esta estructura permite la eficiencia en la búsqueda de publicaciones, por lo que cada nodo tiene como criterio las fechas de las publicaciones y cada nodo tiene su propia lista enlazada con las publicaciones para esa fecha.

```
1 class NodeBST {
2 public:
3     string fecha; // Clave para el nodo del BST
4     FechaList publicaciones; // Lista de publicaciones para esa fecha
5
6     NodeBST* left;
7     NodeBST* right;
8
9     NodeBST(string fecha); // Constructor
10 };
```



## Árbol B

El árbol B se ha implementado para almacenar los comentarios de las publicaciones. Es un árbol de orden 5.

```
1 class NodoB {
2 public:
3     Comentario** comentarios; // Puntero a comentarios (hasta 4 comentarios por nodo)
4     NodoB** hijos;           // Puntero a hijos (hasta 5 hijos por nodo)
5     int numComentarios;      // Número actual de comentarios en el nodo
6     bool hoja;               // True si es un nodo hoja
7
8     NodoB(bool esHoja);
9     void insertarComentario(Comentario* comentario);
10    void dividirNodo(int i, NodoB* y);
11    void imprimir();
12    void graficar(std::ofstream &file);
13 };
14
15 class ArbolB {
16 private:
17     NodoB* raiz;
18
19 public:
20     ArbolB();
21     void insertar(Comentario* comentario);
22     void imprimir();
23     void graficar(const std::string &filename);
24 };
25
```

## Lista de adyacencia

Se ha implementado un grafo no dirigido representado por una lista de adyacencia que contiene las relaciones de amistad de los usuarios en la aplicación:

```
1 #ifndef ADJACENCYLIST_H
2 #define ADJACENCYLIST_H
3
4 #include <string>
5 #include <fstream>
6 #include "usernode.h"
7
8 class AdjacencyList {
9 private:
10
11 public:
12     UserNode* head; // Puntero a la cabeza de la lista
13     AdjacencyList();
14     ~AdjacencyList();
15     void insert(const std::string& userName);
16     void createConnection(const std::string& userName1, const std::string& userName2);
17     void graph(const std::string &filename);
18 };
19
20
21 #endif // ADJACENCYLIST_H
22
```

## Método de Huffman

Utilizado para comprimir la información de los usuarios, consiste en asignar a cada byte del archivo que vamos a comprimir un código binario compuesto por una cantidad de bits tan corta como sea posible.

```
1  #ifndef HUFFMAN_H
2  #define HUFFMAN_H
3
4  #include <string>
5  #include <unordered_map>
6  #include "simplelist.h"
7  #include "huffmannode.h"
8
9  class Huffman {
10 private:
11     HuffmanNode* root;
12     void generateCodes(HuffmanNode* node, std::string code, std::unordered_map<char, std::string> &huffmanCode);
13
14 public:
15     Huffman() : root(nullptr) {}
16     void buildHuffmanTree(std::string text);
17     std::unordered_map<char, std::string> getHuffmanCodes();
18     std::string compress(const std::string& text, std::unordered_map<char, std::string>& huffmanCodes);
19     std::string decompress(const std::string &compressedData, const std::unordered_map<std::string, char> &huffmanCodes);
20     std::unordered_map<char, int> countFrequencies(const std::string& text);
21 };
22
23 #endif // HUFFMAN_H
```

A su vez, este método necesita de una lista simplemente enlazada:

```
1  #ifndef SIMPLELIST_H
2  #define SIMPLELIST_H
3
4  #include "nodelist.h"
5
6  class SimpleList {
7  private:
8     NodeList* head;
9  public:
10     // Constructor por defecto
11     SimpleList();
12     int size;
13
14     void push(HuffmanNode* huffNode);
15     NodeList* pop();
16     void print();
17 };
18
19 #endif // SIMPLELIST_H
20
```