Explicación de implementaciones y decisiones

aquí se añadirán las explicaciones y la decisiones que vayamos tomando en cuanto al código

Entrega número 1:

Explicación Carpeta

Esta clase **Carpeta** representa un depósito de mensajes, que va a permitir organizar mensajes de una forma ordenada, asegurando métodos para agregar, eliminar, realizar listas y también buscar mensajes de una manera ordenada, sin alterar una modificación directa de los datos internos. Usamos atributos privados para encapsular el estado interno y así evitar modificaciones accidentales, validamos las entradas para impedir datos incorrectos, también funciones que permiten búsquedas avanzadas y prevenir errores, para proteger la lista original.

Explicación Servidor

En el servidor decidimos, aparte del método de registrar usuario, hacer también los métodos que se pedían en la interfaz (**enviar mensaje, recibir mensaje y listar mensajes**). Para que sea el que gestione todo, y luego la interfaz hacerla en otro módulo aparte, haciendo que importe la clase **ServidorCorreo** y que solamente llame a los métodos que contiene. Creemos que es lo correcto para hacer una interfaz.

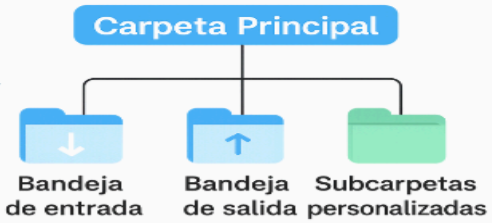
Para solucionar lo de llamar a la interfaz para bandeja de entrada o salida, decidimos hacer dos listar mensajes en **ServidorCorreo**, uno para bandeja de entrada y otro para bandeja de salida, para luego poder llamarlos en interfaz a cada uno por separado.

Corrección: Habíamos confundido una interfaz de menú (que ya removimos) con la interfaz abstracta requerida. Por lo que ya se integró eso en el servidor con todas las modificaciones correspondientes para que funcione el servidor con todo lo requerido.

Entrega número 2 :Infografia

ESTRUCTURA DEL ÁRBOL DE CARPETAS

CLIENTE DE CORREO



Concepto básico

- La clase Carpeta funciona como nodo y como árbol a la vez
- Cada usuario, al crearse, obtiene por defecto:
  - Una carpeta raíz: Carpeta Principal
  - Dos subcarpetas iniciales: Bandeja de Entrada y Bandeja de Salida

Características

- No es un árbol binario → cada nodo puede tener infinitos hijos
- Cada carpeta (nodo) contiene:
  - Una lista de mensajes
  - Un diccionario de subcarpetas (hijos)

INICIALIZACIÓN Y ESTRUCTURA DE LA CLASE CARPETA

La nueva clase Carpeta, implementada como un árbol general, se inicializa por defecto como nodo y árbol a la vez, para garantizar una mayor eficiencia en los métodos de búsqueda e inserción.
Cuenta con un diccionario para almacenar sus hijos, que representan las subcarpetas de la raíz.
Esta raíz se inicializa como "Carpeta principal" al momento de crear un usuario, junto con dos subcarpetas por defecto: "Bandeja de entrada" y "Bandeja de salida".
Cada nodo o subcarpeta tiene una lista donde almacena sus mensajes y un atributo con su nombre para identificarla.

FUNCIÓN DE LOS MÉTODOS

El metodo buscar_mensajes lo hace de forma recursiva con un algoritmo de busqueda en profundidad que primero comienza buscando en la raiz y luego va llamando y "analizando" recursivamente a sus subcarpetas en sus mensajes correspondientes que coincidan con el criterio de busqueda, como se requirio por asunto o remitente. Una vez que los encuentra mediante el metodo .extend() en cada llamada recursiva lo agrega a una lista de resultados.
A la que se retorna finalmente una vez que ya no hay mas recursion (o mensajes que coincidancon el criterio de busqueda).

Para el método mover_mensaje(), optamos por hacer tres métodos auxiliares antes para que sea más limpio el mover mensajes.
Los métodos auxiliares son: agregar_mensaje(), eliminar_mensaje(), que verifican que el mensaje a remover o eliminar sea una instancia de Mensaje, y mediante .append() y .remove(), agregan y eliminan el mensaje deseado.
Y el tercer método auxiliar, obtener_carpeta(), es para devolver la carpeta de origen donde está el mensaje a mover, para luego poder aplicar el método eliminar_mensaje() y reubicarlo en el destino deseado, o sea, moverlo.
Para encontrar la subcarpeta aplicamos la recursión, similar al método buscar_mensajes().

Para el método mover_mensaje() dimos por entendido que era necesario buscar recursivamente la carpeta de origen, pero también dimos por entendido que el usuario de antemano iba a proporcionar la carpeta de destino.
Sino, es imposible saber dónde quiere mover el mensaje.
Por lo tanto, la pasamos como parámetro junto al mensaje a mover en los parámetros del método mover_mensaje().
En el método mover_mensaje() primero verificamos que todos los datos sean instancias.
Después, obtenemos la carpeta de origen con obtener_carpeta(), luego eliminamos el mensaje de la lista de mensajes de la carpeta de origen y agregamos el mensaje al destino con agregar_mensaje(), en el destino proporcionado por el usuario.

Entrega número 3

Explicación nuevos métodos

Nuevo método en módulo Carpeta: `obtener_carpeta_padre()`

Lo que hace este método es primero mirar en las subcarpetas de la raíz; si no encuentra nada, llama recursivamente a los valores de las subcarpetas.
Busca la carpeta padre recursivamente; si encuentra algo que no es `None`, la retorna en la primera línea inmediatamente. Si recorre todos los subniveles y no encuentra nada, retorna `None`. En el caso límite, que no se encuentre, retorna `None`.

La recursión se ejecuta una y otra vez hasta encontrar el resultado. En el peor de los casos, recorre todo y no lo encuentra.
La complejidad es de **O(1)** en el mejor de los casos y **O(n)** en el peor, siendo *n* la cantidad de subcarpetas, ya que se recorre todo sin encontrar nada.

Utilizamos este método para poder implementar el método de crear una subcarpeta anidada en el módulo **Usuario** con el método: `crear_subcarpeta_anidada()`.
En este método, lo que se hace es primero verificar si la carpeta padre es la raíz, y luego, si no lo es, utilizar el método `obtener_carpeta_padre()` para encontrar la carpeta padre en el árbol, y ahí usar el método `agregar_subcarpeta()` e insertar la carpeta nueva en el lugar deseado.

Justificación y costo — Cola de prioridades

Elegimos utilizar el módulo **heapq** para implementar la cola de prioridades porque, a comparación de una lista común, al implementar un **heap binario** nos aseguramos de que las operaciones de inserción y extracción de mensajes tengan un costo/complejidad algorítmica de **O(log N)**.
Esto significa que si la lista es muy grande y sigue creciendo, el costo de las operaciones crece, pero a un valor muchísimo más bajo que lo haría con una lista normal, que tendría un costo mucho más elevado **O(N)**.

Para poder implementar la cola de prioridades en lugar de una lista normal de mensajes en cada carpeta, en la clase **Mensaje** tuvimos que agregarle un parámetro/atributo por defecto, que es la **prioridad**. Si es **1** (por defecto), es normal, y si es **0**, es urgente.

Esto hace que los mensajes urgentes se ubiquen más cerca del índice **0** en la nueva lista (cola de prioridad).

Después, en la clase **Servidor**, se determina que, si el asunto del mensaje contiene la palabra **“urgente”**, se le cambia la prioridad al mensaje, pasando a ser **prioridad = 0**; y si no contiene la palabra **“urgente”**, sigue como normal con **prioridad = 1**.

Con esto, sin importar en qué carpeta esté, si es urgente, siempre va a tener prioridad a la hora de ser ordenado en la cola de prioridades.

Después del filtrado, el mensaje va a ir a una carpeta específica según las reglas definidas por el usuario, y si no coincide con ninguna regla, va por defecto a la **bandeja de entrada**, donde igualmente seguirá ordenado porque es una instancia de **Carpeta**.

Implementación en Carpeta

En la clase **Carpeta**, en el método **agregar_mensaje()**, el mensaje se almacena como una tupla con su prioridad y el mensaje.
Hacemos que, si la prioridad es **0**, esté más cerca de la raíz, ya que el módulo **heapq** ordena a los hijos con prioridad más baja más cerca de la raíz.

Esto lo implementamos con **heapq.heappush()**, que realiza ese "filtrado" del mensaje con menor prioridad (numéricamente) hacia arriba en la raíz.
Como nosotros pusimos **0** en los urgentes, esos quedan más arriba, y los que no son urgentes (**1**, que es mayor a **0**) quedan más abajo.

En el método **listar_mensajes()**, primero se hace una copia de la lista de mensajes. Luego se itera sobre esa lista, y en cada iteración se aplica el algoritmo **heappop()**, que tiene un costo de **O(N log N)**.
Este algoritmo "saca" y coloca en la lista a mostrar el mensaje de mayor prioridad (**0**, urgente).

Cada vez que lo hace, reorganiza el heap para colocar como raíz al hijo de mayor prioridad, lo saca, vuelve a ordenar y así sucesivamente, con un costo de **O(log N)** por cada operación.
El coste total de la lista es **O(N log N)**.

Por último, se muestra la lista de mensajes ordenada por prioridad.
N es la cantidad de veces que "saca y reemplaza" el algoritmo, y **log N** es el costo de cada operación individualmente.

Explicación filtros

El objetivo de implementar el filtrado de mensajes que vamos a hacer es el siguiente:
Que el usuario reciba un mensaje y, si contiene una palabra clave en el asunto, vaya a una carpeta específica o no.
Y si no va a una carpeta en específico, se va directamente a la **bandeja de entrada por defecto**.
Todo esto siempre manteniendo la **prioridad** de un mensaje en la lista de mensajes de cada subcarpeta si es "urgente", ya que la **cola de prioridades** está implementada para todas las listas de mensajes de las subcarpetas que son objetos de la clase **Carpeta**.

La clase **Usuario** va a tener una lista vacía en el constructor, donde se van a guardar las tuplas que contienen *(asunto, destino)*.

Es decir, si el mensaje tiene determinada palabra en el asunto, va a esa carpeta.
En el **caso límite**, si la carpeta no existe o el mensaje recibido no aplica a ningún filtro, simplemente va a la **bandeja de entrada por defecto**.

Para crear la regla de filtrado hicimos el método **`creacion_regla_de_filtro()`**, que toma como parámetro una palabra "clave" del asunto y una carpeta de destino.
Se guarda como tupla y se almacena en la lista de filtros.

Para aplicar el filtro, hicimos un método auxiliar (**`aplicar_filtro()`**) que, si el mensaje a filtrar coincide con una regla de filtrado, **retorna el nombre** de la carpeta de destino para que otro método se encargue de la lógica de movimiento.
Si no coincide, **devuelve None** para que el otro método lo mueva por defecto a la bandeja de entrada.
Para hacer esto, se **itera sobre la lista de tuplas** (reglas de filtrado).

El **análisis de complejidad** de `aplicar_filtro()` es de **O(n)**, ya que para obtener la carpeta de destino dependiendo de la palabra clave en el asunto, depende de la cantidad de tuplas a recorrer que haya en la lista de reglas de filtrado.

Para filtrar el mensaje, en el método **`filtrar_mensaje()`** primero se aplica el método auxiliar `aplicar_filtro()` para que devuelva el nombre de la carpeta de destino o `None`.
Si devuelve el nombre de la carpeta de destino (previamente definida en la regla de filtrado), se aplica la **lógica de movimiento del mensaje**;
y si devuelve `None`, por defecto se guarda el mensaje en la **bandeja de entrada**.

También cambiamos en el **ServidorCorreo** el método **`enviar_mensaje()`**.
El único cambio que hicimos es que, para el destinatario, se aplique el filtro cuando recibe el mensaje.
Antes, lo recibía directamente en la bandeja de entrada.

Entrega número 4 (Final)

Pequeño cambio en la cola de prioridad

Hicimos un pequeño cambio en la cola de prioridad, ya que se implementa con el módulo **heapq** como una *minheap*, es decir, el de menor valor queda arriba, cerca de la raíz.
Habíamos puesto anteriormente que, cuando se enviaba un mensaje, por defecto fuera **0**, y que si contenía "urgente" en el asunto fuera **1**. Pero al hacer el reordenamiento de la lista de mensajes, siempre iba a quedar el mensaje normal arriba, ya que tenía menor valor.

Se nos había ocurrido cambiarle el signo a "urgente" al momento de agregar el mensaje, para que fuera menor que 0 y quedara más arriba en el ordenamiento.
Pero a último momento se nos ocurrió que simplemente **el mensaje normal sea 1 y el urgente 0**, sin necesidad de cambiar el signo. Es un cambio chico, pero deja el código más limpio y legible.

Explicación del grafo y del módulo redservidores

Para hacer el grafo, decidimos hacer otro módulo aparte que se llama **redservidores**. Lo decidimos hacer así porque nos pareció lo más prolijo y funcional. Y no encontramos manera de hacer que se integre dentro del módulo servidor, ya que este mismo lo vamos a utilizar solo para administrar los usuarios que pertenezcan a él y administrar el envío de mensajes.

Y la redservidores contiene todos los nodos (dominios), el mapa topográfico de la red de servidores (dominios) y el algoritmo BFS que vamos a usar para mostrar la simulación de la ruta que hace un mensaje para llegar de un dominio A a un dominio B. Una simulación del camino que hace.

En cuanto al grafo, decidimos hacerlo **un grafo dirigido** porque estuvimos investigando, a grandes rasgos, cómo funciona el envío de mensajes en realidad entre distintos dominios de mail y no siempre se asegura que si un dominio le envía un correo a otro dominio, este mismo tenga en esa arista la capacidad de responder.

Descubrimos que los **Registros MX** son entradas en el DNS (Sistema de Nombres de Dominio) que básicamente le dicen a un dominio que, si quiere enviarle un mensaje a otro dominio, debe conectarse primero a X servidor y recién ahí se hace la conexión/arista en este caso y puede enviarle el mensaje.

Pero esto no significa que pueda responder el receptor. Debe crear otra arista mediante el servidor externo para hacerlo. Por eso es nuestra decisión hacer un grafo dirigido.

En cuanto a la implementación, decidimos usar **la lista de adyacencia**, ya que para el caso de grafos dirigidos es más eficiente en el uso de memoria, ya que a diferencia de la matriz de adyacencia esta solamente almacena en memoria las aristas que tengan conexiones entre vértices.
Y la matriz de adyacencia no: tiene un montón de conexiones aunque no se utilicen, lo que no sería eficiente para este caso.

- **Complejidad lista de adyacencia:** O(n + m), donde n es el número de nodos y m es el número de aristas.

- **Matriz de adyacencia:** $O(n^2)$.

Y para el algoritmo de recorrido decidimos utilizar el **BFS** porque es el algoritmo que encuentra la ruta con menos aristas intermedias entre un nodo y otro, ya que al recorrer completamente nivel por nivel se asegura que si hay un camino entre un nodo A y un nodo B va a ser el que tenga menos aristas intermedias.

Para poder hacer el BFS en el grafo importamos el módulo **collections** para no tener que implementar manualmente una cola como una lista doblemente enlazada, ya que este módulo nos permite utilizarla sin escribir todo el código necesario manualmente. Y sus

operaciones tienen una eficiencia de O(1). Decidimos hacer esto para ganar tiempo, para que el código quede más legible y porque suponemos que está permitido, ya que para hacer la cola de prioridades nos permitió usar el módulo heapq.

En el constructor, las conexiones las hacemos como un diccionario, ya que el tiempo de búsqueda es de O(1) porque busca por nombre (clave) y no por valor, que sería un costo de O(n).

Costo y explicación de métodos

agregar_servidor():
Almacenamos los dominios como set en la lista de adyacencia. Porque el set se almacena como conjunto en el diccionario (lista de adyacencia), con dominio y conexiones. Es altamente eficiente porque mantiene la estructura del diccionario y no agrega conjuntos duplicados (ahorra espacio), manteniendo la eficiencia de búsqueda e inserción en O(1).

agregar_conexion():
Verifica primero que ambos dominios, origen y destino, existan en la red y luego accede a la lista de adyacencias buscando el origen y le agrega a su conexión con `.add()` el destino. Todo manteniendo la eficiencia O(1).

encontrar_ruta():
Utilizamos el algoritmo de búsqueda BFS, que lo que hace es asegurarse de visitar cada "nivel de conexiones" de los dominios antes de pasar al siguiente.
Primero crea una tupla con el dominio actual y la ruta actual visitada (el primero es el mismo). También se almacena en un set, para evitar repeticiones, los dominios ya visitados.

Luego, iterativamente, el algoritmo saca ese primer nodo y, en su lista de conexiones, busca el destino requerido. Si lo encuentra, retorna la ruta desde el origen hasta el destino.
Si no, sigue buscando en su lista de conexiones:

- se agrega al set de visitados la conexión actual para evitar un bucle infinito y repeticiones,
- se crea una nueva ruta con la ruta actual más la conexión actual,
- y finalmente se crea una nueva tupla con esa conexión actual y la ruta actual hasta el momento, y se guarda como una tupla en la cola.

Se realiza este proceso iterativamente hasta encontrar la ruta que coincida con el destino, y de esta manera se asegura que la ruta retornada sea la que tenga **menos vértices/nodos intermedios**. Esto se logra al explorar nivel por nivel antes de pasar al siguiente.

Implementación

Para modelar el grafo vamos a hacer que todo esté ya predefinido, intentando simular cómo se comporta en la vida real la red de servidores, ya que tiene que ser una estructura de datos fija.

El usuario **no puede definir las aristas/conexiones**; de eso se encarga el ingeniero o programador que esté a cargo para configurar la red (qué servidor puede enviar un mensaje a otro, etc.).

Al estar todo predefinido, el usuario simplemente se registra en un dominio en específico y envía mensajes (aparte de definir sus carpetas, etc.), nada más.

Con “predefinido” nos referimos a que los dominios van a estar ya definidos **antes de ejecutar la interfaz**, en un módulo de pruebas `main.py`, no predefinidos en el módulo redservidores.

Integración del grafo en la clase Servidor

Para poder hacer el grafo tuvimos que modificar la clase servidor. Ahora cada servidor va a tener dominio propio y en él cada usuario registrado obviamente va a pertenecer a ese dominio.
Hicimos que como parámetro todas las instancias de servidor tengan el mismo parámetro *red_servidores*. Esto lo creamos para que al momento de crear *main.py* todas las instancias de servidor utilicen el mismo mapa topográfico de la red de servidores y no cada una una instancia separada, y se pueda hacer la simulación del enrutamiento de mensajes.

El cambio que hicimos en el método *enviar_mensaje()* fue que se pueda enviar y recibir mensajes a nivel local, o sea en el mismo servidor. Pero que a nivel externo, en el enrutamiento, solo se pueda hacer la simulación de envío mostrando la ruta utilizando el método *encontrar_ruta()* de *RedServidores*.
No pudimos hacer que se envíen y guarden mensajes entre distintos servidores más allá de la simulación del enrutamiento. Lo logramos haciendo *.join()* con el BFS en el caso de que sea un envío de mensaje a un servidor externo.

Implementación del Menú y su Integración con la Red de Servidores

Para hacer el menú, lo que decidimos fue crear un módulo aparte con un menú de línea de comandos, que funciona mediante *inputs* del usuario. Decidimos hacerlo así porque usa como parámetro la red de servidores, que luego le será proporcionada en **main.py**, donde se importa la red (que maneja el mapa topográfico de servidores) y el servidor (que maneja toda la lógica de movimiento de mensajes, registros, carpetas, etc., utilizando funciones de la clase Usuario).

Para lograr “extraer” el objeto servidor y utilizar sus funcionalidades, hicimos una función auxiliar en el menú que realiza eso: **get_servidor()**, que revisa que el correo sea válido y, si lo es, extrae el dominio y luego el objeto servidor asociado a ese dominio. Lo obtiene de un diccionario que almacena los servidores asociados a los dominios.

El método **registrar_usuario()** toma los datos mediante inputs del usuario y lo que hace es registrar al usuario pero, en vez de usar *self.* como en la instancia individual del servidor, utiliza la instancia del servidor asociada a su dominio mediante la extracción de **get_servidor()** (que la obtiene del diccionario con las distintas instancias de servidor, cada una manejando su propio diccionario de usuarios, carpetas, etc.). Luego utiliza las funcionalidades de ese servidor en específico.

El método **enviar_mensaje()** aplica la lógica de *enviar_mensaje*, pero en vez de utilizar *self* para obtener al usuario del diccionario del servidor, nuevamente usa **get_servidor()** para extraer la instancia asociada al dominio. Para el destinatario, si es del mismo dominio, no es necesario un segundo *get_servidor()* porque ya se activó el método al detectar primero el remitente; y si no es el mismo dominio, se aplica la simulación de enrutamiento.

Para **listar_mensajes()**, tuvimos que modificar el método en la clase Servidor para que quede actualizado con los nuevos métodos de obtener subcarpeta y listar los mensajes ordenados por prioridad de Carpeta. La lógica es la misma que en los otros métodos: se piden entradas, se obtiene el servidor mediante **get_servidor()** y se llama al *listar_mensajes()* de esa instancia de Servidor.

Para **crear_subcarpeta_principal()**, primero creamos el método en Servidor para poder delegarlo a la interfaz. Utilizamos el método *agregar_subcarpeta()* de Carpeta en la carpeta principal del usuario (raíz), mediante las entradas del usuario para crear la subcarpeta de la carpeta principal. Siempre usando **get_servidor()** para obtener el servidor correspondiente al dominio del correo del usuario.

Para **crear_subcarpeta_anidada()**, también creamos el nuevo método previamente en Servidor, utilizando el método *crear_subcarpeta_anidada()* de Usuario. Mediante entradas del usuario y usando **get_servidor()**, se crea la subcarpeta anidada dentro de la carpeta que el usuario indique.

Para **mover_mensaje()**, con el método previamente creado en Servidor y usando **get_servidor()**, reutilizamos el método *mover_mensaje()* de Carpeta y *buscar_mensajes()* de Carpeta para obtener el primer mensaje cuyo asunto coincida con el solicitado, y *obtener_subcarpeta()* para obtener el objeto Carpeta especificado por el usuario.

El método **contador_mensaje()** reutiliza *recibir_mensaje()* de Servidor; es un simple contador de mensajes en la bandeja principal.

Para la ejecución de la interfaz gráfica de comandos, imprimimos en consola un menú con todas las opciones disponibles, y mediante inputs el usuario elige primero la opción y luego la función que quiera utilizar, incluyendo una opción para errores o para salir de la interfaz. Todo esto mediante un bucle *while*.

Para probar el programa hicimos un módulo de pruebas **main.py**, donde creamos las instancias de los servidores (dominios) y hacemos que todos utilicen la misma red gracias al parámetro del constructor de Servidor, ya que está pensado para que al crear una instancia compartan la misma variable; de lo contrario se crearía un mapa topográfico distinto para cada uno y no se podría hacer la simulación a nivel externo, solo a nivel local de cada servidor.

Solución en main.py

Al hacer el archivo de prueba nos dimos cuenta de que no funcionaba la lógica de mover mensajes porque estaba diseñada para mover el mensaje directamente (o sea, eliminarlo y moverlo a una nueva carpeta). Nos daba el error de que no encontraba el mensaje, ya que el objeto al que estábamos intentando acceder era una tupla. Al usar *heapq* para hacer la cola de prioridades, ahora el mensaje era una tupla con (valor, mensaje).

Para solucionarlo tuvimos que modificar varios métodos de la lógica de búsqueda y movimiento para acceder al objeto mensaje dentro de la tupla. El que más se nos complicó fue el de eliminar mensaje, ya que teníamos que iterar sobre la lista de mensajes —que ahora eran tuplas—, encontrar el objeto [1] de la tupla en la lista y ver si coincidía (el asunto). Si coincidía lo eliminábamos con *pop* y usábamos *heapq.heapify* para mantener la estructura de *heapq* (cola de prioridad) en la lista de mensajes de la clase Carpeta. Esto se usa cuando se quita “manualmente” un elemento de la estructura de la cola y es necesario que se reordene, ya que eso no ocurre automáticamente cuando se modifica manualmente; solo ocurre al usar *heappush* y *heappop*.

También se hicieron modificaciones en *buscar_mensaje* y *obtener_carpeta*, intentando ingresar al objeto [1] de la tupla y, si no se encontraba en un primer nivel, aplicar la búsqueda recursiva en subcarpetas. Utilizamos *enumerate* para convertir la lista en un iterable con posiciones, poder eliminar la tupla completa y luego reordenar con *heapify*.