Carrera: Tecnicatura Universitaria en Programación
Materia: Estructura de datos 2025
Comisión: 5
Docente: Felipe Morales
Estudiantes: Igarzabal Valeria E. / Maximo Cruz

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Trabajo Práctico Final:
Tema: Gestión Integral de:

PASAJES DE AVIÓN

Trabajo Práctico Integrador – Gestión de Pasajes de Avión y Estructuras de Datos en Python

Hilo Conductor: "Gestión Integral de Pasajes de Avión"

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Modalidad y Entregas:

- Trabajo grupal (2 o 3 integrantes)
- El trabajo se desarrolla en 4 entregas parciales (una cada tres semanas)
- Cada entrega debe incluir: desarrollo teórico, implementación en Python, documentación y material adicional (presentación, video, infografía, etc.)
- Defensa oral individual y grupal al finalizar

Objetivo General:

Integrar y aplicar los conceptos de estructuras de datos y análisis de algoritmos en el contexto de la gestión, análisis y simulación de pasajes de avión, desarrollando tanto la fundamentación teórica como la implementación práctica.

Entrega 1: Modelado y Encapsulamiento (Unidades 1 y 2)

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Teoría:

- Explicar el concepto de encapsulamiento y su importancia en la programación orientada a objetos.
- Justificar la elección de atributos y métodos para modelar pasajeros, vuelos y reservas.
- Analizar ventajas y desventajas de distintas representaciones de listas y pilas para gestionar reservas y equipaje.

Práctica:

- Definir la clase Pasajero con atributos: nombre, documento, nacionalidad, historial de vuelos, equipaje, etc.
- Definir la clase Vuelo con atributos: código, origen, destino, fecha, lista de pasajeros, etc. Implementar interfaces para agregar/eliminar reservas y equipaje.
- Implementar una estructura recursiva para calcular el total de equipaje de un pasajero considerando conexiones y escalas.

Adicional obligatorio:

Presentación (diapositivas o video corto) explicando el diseño de las clases y la lógica de encapsulamiento.

Encapsulamiento

UNIVERSIDAD NACIONAL GUILLERMO BROWN

El encapsulamiento consiste en agrupar datos (atributos) y métodos (comportamiento) que operan sobre esos datos en una sola unidad conocida como “clase”.

características clave:

Ocultamiento de información: restringe el acceso directo a los datos internos de un objeto, solo los métodos de la propia clase tienen acceso para modificar sus atributos.

Interfaz pública: que incluye métodos específicos que otras clases pueden utilizar para interactuar con los datos de forma controlada y segura.

Seguridad: al ocultar los datos, se evita que sean modificados por un código externo de forma inesperada o no autorizada.

Modularidad: permite que los desarrolladores trabajen en distintas partes del código de forma independiente.

Flexibilidad: el encapsulamiento hace que el código sea más flexible y reutilizable.

Modelamos las clases con sus atributos y métodos:

UNIVERSIDAD NACIONAL GUILLERMO BROWN

```
Class Equipaje:
    def __init__(self, descripción, peso):
        self.descripcion = descripcion
        self. peso = peso
    def __str__(self):
        return f"{self.descripcion} {self.peso} kg."
```

El equipaje es un elemento que forma parte de una lista (equipajes), que luego se cargará (mediante un método) al equipaje del pasajero. Es por este motivo que lo vamos a modelar antes que pasajero.

Usamos el método "__str__" para definir la representación en cadena (string) "informal" o "legible para humanos" de un objeto (Equipaje). Se llama automáticamente cuando invocas mediante un "print()" sobre el objeto.

El formato f-string (o literal de cadena formateada) fue introducida en Python 3.6. Dentro de esta cadena se pueden incrustar expresiones de Python rodeandolas con llaves { }. Python evalúa la expresión dentro de las llaves, e inserta el valor en la ubicación correspondiente.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Ventajas y Desventajas de representación de Listas y Pilas

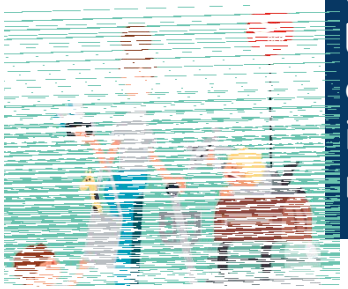
	Listas: Una lista permite el acceso, inserción y eliminación en cualquier posición. nos permite acceder a un elemento por su índice.	Pilas: Es una estructura de datos que sigue el principio LIFO (último en entrar (last - in), primero en salir (first - out). Es decir el último elemento añadido es el primero en ser eliminado. solo permite la inserción "push" y la eliminación "pop" en un extremo, la "cima".
Ventajas	Es fácil de iterar para encontrar un elemento específico y actualizar sus detalles.	Estas operaciones (añadir y eliminar), son extremadamente rápidas, ya que solo se manipula un extremo.
Desventajas	Para insertar o eliminar un elemento en el medio, es lento. ya que requiere mover todos los elementos posteriores y esto puede requerir reasignación de memoria más frecuente.	Tiene acceso restringido, ya que solo se puede acceder al elemento superior. acceder a otro elemento requiere la eliminación de todos los elementos que están por encima. solo es útil para situaciones que encajan estrictamente con el patrón LIFO.

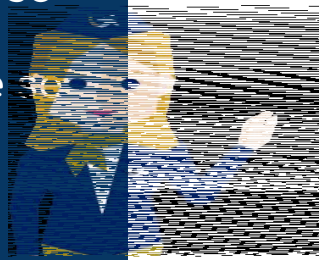
Modelamos las clases con sus atributos y métodos:

UNIVERSIDAD NACIONAL GUILLERMO BROWN

```
class Pasajero:
  def __init__(self, nombre,
documento, nacionalidad):
    self.nombre = nombre
    self.documento = documento
    self.nacionalidad = nacionalidad
    self.historial_de_vuelo = []
    self.equipajes = []
    self.equipajes_peso = 0
```

un pasajero tiene un nombre, apellido, documento y nacionalidad. y en los vuelos que hace se les agrega equipajes y un historial de vuelo.Estos últimos se definen como listas donde luego se van a ir cargando los elementos. A medida que le va incrementando el equipaje, si le irá incrementando el peso, mediante una función.





JUSTIFICACIÓN PARA LA REPRESENTACIÓN DE PILAS O LISTAS

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Analizando las ventajas y desventajas de la representación de pilas y listas:

Para agregar o eliminar “equipaje”, por ejemplo (en el caso que el pasajero decida no llevar alguno de ellos), no necesariamente va a ser el último que se cargó, puede llevar un bolso pequeño, una mochila y un bolso grande, cargado en ese orden, pero luego decide dejar la mochila, que se encontraba en la posición [1] de la lista, es necesario acceder al elemento por posición, por lo que se encontraría más útil la representación de listas.

Sin embargo, en “historial de vuelo”, en el caso de cancelar un vuelo, esté siempre sería el último, Por lo que no es necesario acceder por posición para agregar o eliminar, entonces resulta más práctico y eficiente la representación de pilas.

```
def mostrar_pasajero(self):
    print(f"Nombre: {self.nombre}")
    print(f"Documento: {self.documento}")
    print(f"Nacionalidad: {self.nacionalidad}")
    print(f"Equipajes: Peso Total:
{self.equipajes_peso} Kg")
    for e in self.equipajes:
      print(e)
```

Este método nos permite mostrar la información del pasajero de manera legible para el usuario. La cantidad de equipaje total del en Kg, y detalles del mismo.

```
def agregar_vuelo(self, vuelo):
    self.historial_de_vuelo.append(vuelo)
```

Nos permite agregar un vuelo al historial de vuelo de un pasajero.

```
 def agregar_equipaje(self, equipaje):
     self.equipajes.append(equipaje)
      self.equipajes_peso += equipaje.peso
 def quitar_equipaje(self, equipaje):
    self.equipajes.remove(equipaje)
    self.equipajes_peso -= equipaje.peso
```

agregar_equipaje y quitar_equipaje nos permite incrementar y disminuir el equipaje de un pasajero y a su vez calcular su peso.

```
class Vuelo:
  def __init__(self, codigo, origen, destino):
    self.codigo = codigo
    self.origen = origen
    self.destino = destino
    self.lista_de_pasajeros = []
```

self.lista_de_pasajeros = [] inicializa una lista vacía dentro de cada objeto Vuelo.

La clase Vuelo es una plantilla diseñada para modelar la información esencial de un vuelo individual dentro de un sistema de gestión.

Su lógica principal es encapsular los datos básicos del viaje y, fundamentalmente, mantener un registro de todos los pasajeros asociados a ese vuelo específico.

Propósito: Esta lista está diseñada para almacenar dinámicamente los datos de los pasajeros que reservan asientos en este vuelo específico. A medida que los pasajeros compran boletos, se añadirán elementos (probablemente objetos de la clase Reserva_Pasajes o Pasajero) a esta lista utilizando otros métodos que se definirán más adelante.

- self.codigo: Un identificador único para el vuelo (ej. "AR1240"). Se usa para diferenciar este vuelo de otros.
- self.origen: La ciudad o aeropuerto de salida (ej. "EZE").
- self.destino: La ciudad o aeropuerto de llegada (ej. "MAD").

```python
def agregar_pasajero(self, pasajero,reserva):
     if pasajero in self.lista_de_pasajeros:
        return f"el pasajero ya se encuentra"
     elif self.codigo != reserva.vuelo_asociado:
       return f"el vuelo no coincide"
     else:
        self.lista_de_pasajeros.append(pasajero)
```

El método agregar_pasajero primero verifica si el pasajero se encuentra ya en la lista_de_pasajeros, si el pasajeros se encuentra en la lista devuelve el mensaje “el pasajero ya se encuentra”.
Luego comprueba si el codigo de vuelo coincide con el de la reserva del pasajero, si no coincide devuelve el mensaje “el vuelo no coincide”.
Sino, agrega al pasajero a la lista_de_pasajeros.

```
def quitar_pasajero(self):
     if self.pasajero in self.lista_de_pasajeros:
       self.lista_de_pasajeros.remove(pasajero)
```

Comprueba si el pasajero se encuentra en la lista de pasajeros, y lo elimina.

```
 def mostrar_vuelo(self):
     return f"Codigo: {self.codigo}, Origen: {self.origen}, Destino:
{self.destino}"
```

Muestra los detalles del vuelo.

```python
class Reserva_Pasajes:
    def __init__(self, numero_reserva, pasajero, asiento, vuelo, precio_pasaje):
        self.numero_reserva = numero_reserva
        self.pasajero = pasajero
        self.asiento = asiento
        self.vuelo = vuelo
        self.precio_pasaje = precio_pasaje
        self.activa = True
```

- self.numero_reserva: Un identificador único para la reserva.
- self.pasajero: La información del pasajero (probablemente un nombre, un ID, o quizás otra clase Pasajero).
- self.asiento: El número o código del asiento asignado (ej. "14A").
- self.vuelo: El identificador del vuelo (ej. el número de vuelo).
- self.precio_pasaje: El costo de la reserva.
- self.activa = True: Este es un atributo predefinido que comienza siempre como True al crear la reserva.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

```
    def cancelar_reserva(self):
        if self.activa:
            self.activa = False
            return "La reserva ha sido cancelada"
        else:
            return "La reserva ya está cancelada"
```

El propósito de este método es gestionar el cambio de estado de una reserva, pasando de estar "activa" a estar "inactiva" (cancelada), evitando errores si se intenta cancelar dos veces la misma reserva.

Verificación Condicional (if self.activa:): El método primero comprueba el valor del atributo self.activa del objeto (que se inicializa como True en el __init__).

UNIVERSIDAD NACIONAL GUILLERMO BROWN

- Si la condición self.activa es True:
- Cambia el estado de la reserva, estableciendo self.activa a False.
- Devuelve un mensaje de confirmación ("La reserva ha sido cancelada").
- La ejecución del método termina aquí.

- Si la condición self.activa es False (porque ya se había cancelado anteriormente):
- El código dentro del else se ejecuta.
- Devuelve un mensaje indicando que no se requiere acción ("La reserva ya está cancelada").
- La ejecución del método termina aquí.

```python
  def actualizar_precio_pasaje(self, nuevo_precio):
        # La lógica correcta valida el 'nuevo_precio' (el parámetro de entrada)
        if nuevo_precio < 0:
            return "Error: El nuevo precio no puede ser negativo"
        else:
            # Si la validación es exitosa, actualiza el atributo
            self.precio_pasaje = nuevo_precio
            return f"Precio actualizado a: {self.precio_pasaje}"
```

El objetivo de esta función es actualizar el precio del pasaje (self.precio_pasaje) de la reserva actual a un nuevo_precio proporcionado por el usuario, pero solo si ese nuevo precio tiene sentido en el mundo real.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Verificación de la Condición (if nuevo_precio < 0:):

- El método recibe un argumento llamado nuevo_precio.
- La primera línea de lógica realiza una validación de datos. Comprueba si el valor que se intenta introducir es menor que cero (es decir, negativo).

- Si la condición nuevo_precio < 0 es True:
- La función detiene su ejecución inmediatamente.
- Devuelve un mensaje de error claro.
- Importante: El atributo self.precio_pasaje *no se modifica*. El precio original de la reserva permanece intacto.
- Si la condición es FALSE, el código salta a la sección Else:
- Se produce la actualización de estado, y el atributo (self.precio_pasaje) toma el nuevo valor.
- La función devuelve un mensaje de actualización y el valor nuevo.

El método especial __str__ (abreviatura de "string") es un método incorporado de Python que se llama automáticamente en las siguientes situaciones:

1. Cuando usas print(mi_reserva).
2. Cuando usas str(mi_reserva)

UNIVERSIDAD NACIONAL GUILLERMO BROWN

```
def __str__(self):
        """Representación legible del objeto."""
        estado = "Activa" if self.activa else "Cancelada"
        return f"Reserva {self.numero_reserva} | Pasajero: {self.pasajero} | Vuelo: {self.vuelo}
| Estado: {estado}"
```

Se utiliza un valor condicional para establecer el estado basándose en el valor booleano de self.activa:

- Si self.activa es True, la variable estado toma el valor "Activa".
- Si self.activa es False, la variable estado toma el valor "Cancelada".

- La función utiliza un f-string (cadena formateada con f al principio) para construir una cadena de texto legible y coherente.
- Combina los atributos clave del objeto (self.numero_reserva, self.pasajero, self.vuelo) con la variable estado que acabamos de definir, usando separadores (|, :) para una lectura clara.

Entrega 2: Árboles y Jerarquías (Unidades 3 y 4)

Teoría:

- Explicar la diferencia entre árboles binarios y árboles generales. Ejemplificar con jerarquías de vuelos y conexiones.
- Analizar la eficiencia de los recorridos y búsquedas en cada tipo de árbol.

Práctica:

- Implementar un árbol binario para organizar vuelos según fecha o número de vuelo.
- Modelar el historial de vuelos de un pasajero como un árbol general (cada nodo es un vuelo, ramas son conexiones o escalas).
- Implementar recorridos (preorden, inorden, postorden) y búsquedas en ambos árboles.

Adicional obligatorio:

- Infografía o póster digital mostrando la jerarquía de vuelos y el árbol de conexiones de al menos un pasajero.

Diferencia entre árboles binarios y árboles generales:

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Árbol Binario	Árbol General
• **Definición**: Cada nodo tiene como máximo dos hijos, típicamente llamados "hijo izquierdo" y "hijo derecho". • **Uso en jerarquías:** Es útil para estructuras simples y organizadas de forma binaria, como la división de datos en un árbol binario de búsqueda (BST) donde los valores menores van a la izquierda y los mayores a la derecha.	• **Definición:** Cada nodo puede tener un número variable de hijos. La aridad (número máximo de hijos) se puede definir según el problema. • **Uso en jerarquías:** Es más adecuado para representar estructuras jerárquicas complejas y dinámicas.

- Ejemplificar con jerarquías de vuelos y conexiones.
- Analizar la eficiencia de los recorridos y búsquedas en cada tipo de árbol.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Árbol Binario	Árbol General
• **Ejemplo de vuelos:** Un modelo de ruta aérea donde una ciudad central solo tiene dos posibles destinos de conexión (por ejemplo, una al este y otra al oeste), o dónde se clasifica la información de los vuelos de manera binaria para la búsqueda. • **Limitación:** No es adecuado para representar situaciones complejas con múltiples conexiones simultáneas, como el itinerario completo de una ciudad.	• **Ejemplo de vuelos:** Un aeropuerto principal (la raíz) con múltiples vuelos saliendo hacia diferentes ciudades (sus hijos). Cada ciudad hija puede tener a su vez múltiples vuelos saliendo a otras ciudades, formando una red de conexiones más compleja y realista. • **Ventaja:** Es más flexible y representa mejor las redes del mundo real, donde un nodo puede estar conectado a muchos otros, como en el caso de los aeropuertos.

Árbol General: Jerarquía con conexiones aéreas: Un árbol general es ideal para representar una estructura donde un punto de origen puede llevar a múltiples destinos directos, sin una limitación a sólo dos.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

- Escenario: Una compañía aérea quiere modelar todas las rutas directas que salen de un aeropuerto principal.
- Estructura:
- El nodo raíz es el aeropuerto principal (ej. Buenos Aires - EZE).
- Sus hijos son todos los destinos directos (ej. Madrid, Nueva York, Bogotá, Roma, Santiago, etc.).
- Cada uno de esos destinos puede, a su vez, ser el padre de sus propias conexiones directas (ej. desde Madrid, vuelos a Londres, París, Berlín, etc.).
- Flexibilidad: Cada aeropuerto puede tener un número variable de rutas directas, lo cual encaja perfectamente con la definición de un árbol general.

Árbol Binario: Sistema de Búsqueda de Vuelos por Precio: Un árbol binario es útil para organizar datos de una manera que facilite la búsqueda rápida, como en un sistema de comparación de precios de vuelos.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

- Escenario: Un sitio web de viajes organiza los precios de los vuelos para una ruta específica (ej. Buenos Aires a Madrid) para que los usuarios puedan encontrar fácilmente el más barato o el más caro. Se utilizará un árbol binario de búsqueda (BST), un tipo específico de árbol binario.
- Estructura:
- El nodo raíz tiene un precio (ej. $800).
- Todos los precios menores se organizan en su subárbol izquierdo.
- Todos los precios mayores se organizan en su subárbol derecho.
- Cada nodo subsiguiente sigue la misma regla: los precios menores van a la izquierda, los mayores a la derecha.
- Eficiencia: Esta organización permite encontrar el vuelo más barato o un precio específico de manera muy eficiente, descartando la mitad de las opciones en cada paso de la búsqueda.

Este árbol organiza los vuelos por su número de vuelo (clave). Los vuelos con números menores van a la izquierda, los mayores a la derecha, facilitando la búsqueda y ordenación.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

```python
class VueloBinario:
    def __init__(self, numero_vuelo, fecha, destino):
        self.numero_vuelo = numero_vuelo
        self.fecha = fecha
        self.destino = destino
        self.izquierdo = None
        self.derecho = None
```

```
class ArbolBinarioVuelos:
    def __init__(self):
        self.raiz = None
```

UNIVERSIDAD NACIONAL GUILLERMO BROWN

La clase ArbolBinarioVuelos representa el árbol binario completo. No contiene los datos de los vuelos individuales directamente, sino que gestiona el punto de entrada a esa estructura de datos.

Su propósito principal es:

- Mantener una referencia al primer nodo del árbol (la raíz).
- Proporcionar métodos (que aún no se han definido) para interactuar con el árbol completo.

```
 def insertar(self, numero_vuelo, fecha, destino):
        if self.raiz is None:
            self.raiz = VueloBinario(numero_vuelo, fecha,
destino)
        else:
            self._insertar_recursivo(self.raiz, numero_vuelo,
fecha, destino)
```

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Verificar si el Árbol está Vacío (if self.raiz is None)

- ¿El árbol no tiene raíz? Esto significa que es el primer vuelo que se introduce en todo el sistema.
- Acción: Crea una nueva instancia de VueloBinario con los datos proporcionados y la asigna directamente a self.raiz. El árbol ahora tiene un único nodo.
- ¿El árbol ya contiene nodos? No podemos simplemente sobrescribir la raíz existente. Entonces llama a la función auxiliar llamada _insertar_recursivo.

La función _insertar_recursivo (el guión bajo inicial, que indica que es un método interno/privado de la clase ArbolBinarioVuelos) compara el nuevo vuelo con los nodos existentes hasta encontrar un lugar vacío adecuado.
La función toma un nodo_actual como punto de referencia y lo compara con el nuevo numero_vuelo. Si es menor que el nodo_actual, debe ir a la izquierda y si es mayor debe ir a la derecha. Si es igual, muestra un mensaje y la inserción se detiene.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

```
def _insertar_recursivo(self, nodo_actual, numero_vuelo, fecha, destino):
        if numero_vuelo < nodo_actual.numero_vuelo:
            if nodo_actual.izquierdo is None:
                nodo_actual.izquierdo = VueloBinario(numero_vuelo, fecha, destino)
            else:
                self._insertar_recursivo(nodo_actual.izquierdo, numero_vuelo, fecha,
destino)
        elif numero_vuelo > nodo_actual.numero_vuelo:
            if nodo_actual.derecho is None:
                nodo_actual.derecho = VueloBinario(numero_vuelo, fecha, destino)
            else:
                self._insertar_recursivo(nodo_actual.derecho, numero_vuelo, fecha, destino)
        else:
            print(f"Vuelo con número {numero_vuelo} ya existe.")
```

Estas dos funciones implementan la lógica para encontrar un vuelo específico dentro del árbol, aprovechando la estructura ordenada del Árbol de Búsqueda Binaria (BST) para realizar la búsqueda de manera eficiente.

UNIVERSIDAD NACIONAL GUILLERMO BROWN

```
def buscar(self, numero_vuelo):
        return self._buscar_recursivo(self.raiz, numero_vuelo)

def _buscar_recursivo(self, nodo_actual, numero_vuelo):
    if nodo_actual is None:
        return None
    if nodo_actual.numero_vuelo== numero_vuelo:
        return nodo_actual
    elif numero_vuelo < nodo_actual.numero_vuelo:
        return self._buscar_recursivo(nodo_actual.izquierdo,
numero_vuelo)
    else:
        return self._buscar_recursivo(nodo_actual.derecho,
numero_vuelo)
```

Funcionamiento de buscar(self, numero_vuelo): Este es el método público que el usuario final llama para iniciar la búsqueda.

1. Punto de Entrada: Simplemente llama a la función auxiliar recursiva _buscar_recursivo, pasándole la raíz del árbol como punto de partida inicial y el número de vuelo que se está buscando.

2. Devolución: Retorna el resultado que le devuelve la función recursiva (ya sea el nodo del vuelo encontrado o None si no se encontró).

Buscar:

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Funcionamiento de: _buscar_recursivo(self, nodo_actual, numero_vuelo):
Esta es la función privada (auxiliar) que implementa la lógica recursiva y eficiente de la búsqueda.

Utiliza el mismo principio de ordenamiento que la función insertar: si el valor buscado es menor que el nodo actual, va a la izquierda; si es mayor, va a la derecha.

Caso base 1: Nodo No Encontrado (fin del camino):

- Condición: *if nodo_actual is None:* Si la función llega al punto donde el nodo_actual es None, significa que ha recorrido todo el camino posible sin encontrar el número de vuelo buscado. Retorna None, indicando que el número no existe en el árbol.

Caso base 2: Nodo Encontrado:

- Condición: *if nodo_actual.numero_vuelo == numero_vuelo:* Compara el numero_vuelo que estamos buscando con el numero_vuelo almacenado en el nodo_actual, si coincide, hemos encontrado el vuelo. Retorna el nodo_actual completo (que contiene la fecha y el destino).

Caso recursivo 1: Buscar a la izquierda:

- condición: *if numero_vuelo < nodo_actual.numero_vuelo:* Si el nodo_actual es menor al numero_vuelo, el algoritmos sabe (si el numero_vuelo existe) debe estar en la subrama izquierda. Realiza una llamada recursiva pero ahora con nodo_actual.izquierdo como punto de partida.

Caso recursivo 2: Buscar a la derecha:

- Condición: *if numero_vuelo > nodo_actual.numero_vuelo:* Si el nodo_actual es mayor al numero_vuelo, el algoritmo sabe(si el numero_vuelo existe) debe estar en la subrama derecha. Realiza una llamada recursiva pero ahora con nodo_actual.derecho como punto de partida.

Gracias a esta estructura, la búsqueda es muy rápida. En lugar de revisar todos los vuelos uno por uno, en cada paso de la recursión se descarta aproximadamente la mitad de los vuelos restantes.

```
 def inorden(self):
        resultado = []

self._inorden_recursivo(self.raiz,
resultado)
        return resultado

    def _inorden_recursivo(self, nodo,
resultado):
        if nodo:

self._inorden_recursivo(nodo.izquierdo,
resultado)

resultado.append(nodo.numero_vuelo)

self._inorden_recursivo(nodo.derecho,
resultado)
```

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Recorrido inorden:

- Estas funciones implementan el recorrido inorden de un Árbol de Búsqueda Binaria. La característica más importante de este recorrido es que visita los nodos en orden ascendente de su valor clave (en este caso, numero_vuelo).

inorden(self):

- Este es el método público que el usuario llama para obtener una lista ordenada de todos los vuelos.

_inorden_recursivo(self, nodo, resultado):

- Esta función privada implementa la lógica recursiva del recorrido inorden, el orden en que se ejecutan las tres líneas, dentro del if nodo: es fundamental.

Recorrido inorden:

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Inorden recursivo: *_inorden_recursivo(self, nodo, resultado):*

El algoritmo sigue este patrón estricto para cada nodo que visita:

1. **Ir completamente a la izquierda:** Se llama a sí misma repetidamente para explorar la rama izquierdo hasta llegar al nodo más a la izquierda posible (un nodo "hoja" o un None).
2. **Procesar el nodo actual:** Una vez que ya no puede ir más a la izquierda, "visita" el nodo actual (añadiendo su numero_vuelo a la lista resultado). Este es siempre el elemento más pequeño que no ha sido visitado todavía.
3. **Ir a la derecha:** Finalmente, intenta ir a la rama derecho de ese nodo para repetir el proceso (ir a la izquierda extrema de ese subárbol, visitarlo, ir a la derecha, etc.).

Resultado Final

Debido a que un Árbol de Búsqueda Binaria organiza los elementos de forma que todos los elementos a la izquierda de un nodo son menores y todos a la derecha son mayores, el recorrido Inorden garantiza que la lista resultado final contenga todos los números de vuelo en estricto orden ascendente.

```
def preorden(self):
        resultado = []
        self._preorden_recursivo(self.raiz,
resultado)
        return resultado

    def _preorden_recursivo(self, nodo,
resultado):
        if nodo:

resultado.append(nodo.numero_vuelo)

self._preorden_recursivo(nodo.izquierdo,
resultado)

self._preorden_recursivo(nodo.derecho,
resultado)
```

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Recorrido preorden:

- Estas funciones implementan un recorrido preorden de un Árbol de Búsqueda Binario.
- A diferencia del inorden, el preorden se utiliza comúnmente para crear una copia de la estructura del árbol o para mostrar los datos de una manera que preserve la jerarquía original del árbol (visitando la raíz antes que sus hijos).

preorden(self):

- Este es el método público que el usuario llama para obtener una lista de los vuelos en el orden de preorden.
- Crea una lista vacía llamada ***resultado***, llama a la función auxiliar ***_preorden_recursivo***, pasandole la raíz, (***self.raiz***) como punto de inicio y la lista resultado por referencia. Retorna la lista ***resultado*** completa, después de visitar todos los nodos.

_preorden_recursivo(self, nodo, resultado):

- Esta función privada, implementa la lógica recursiva del recorrido preorden. El orden de las operaciones es clave, primero procesa el nodo actual, luego a sus hijos.

Recorrido preorden:

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Preorden recursivo: _preorden_recursivo(self, nodo, resultado):

El algoritmo sigue este patrón estricto para cada nodo que visita:

1. Procesar el nodo actual (Raíz): Inmediatamente añade el numero_vuelo del nodo actual a la lista resultado.
2. Explorar la rama izquierda: Realiza una llamada recursiva para visitar el subárbol izquierdo, repitiendo el proceso (visitar la raíz izquierda, luego su izquierda, luego su derecha, y así sucesivamente).
3. Explorar la rama derecha: Una vez que todo el subárbol izquierdo ha sido procesado, realiza una llamada recursiva para visitar el subárbol derecho, siguiendo el mismo patrón.

Resultado final:

El recorrido preorden genera una lista donde la raíz del árbol aparece primero, seguida por todos los elementos de su subárbol izquierdo y finalmente todos los elementos de su subárbol derecho.

```
    def postorden(self):
        resultado = []

self._postorden_recursivo(self.raiz,
resultado)
        return resultado

    def _postorden_recursivo(self,
nodo, resultado):
        if nodo:

self._postorden_recursivo(nodo.izquierd
o, resultado)

self._postorden_recursivo(nodo.derecho,
resultado)

resultado.append(nodo.numero_vuelo)
```

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Recorrido postorden:

- Estas funciones implementan un recorrido postorden de un Árbol de Búsqueda Binario.
- En el recorrido postorden el nodo actual se visita después de que se han visitado sus dos subárboles,.

postorden(self):

- Este es el método público que el usuario llama para obtener una lista de los vuelos en el orden de postorden.
- Crea una lista vacía llamada ***resultado***, llama a la función auxiliar ***_postorden_recursivo***, pasandole la raíz, (***self.raiz***) como punto de inicio y la lista resultado por referencia.
- Retorna la lista ***resultado*** completa, después de visitar todos los nodos.

_postorden_recursivo(self, nodo, resultado):

- Esta función privada, implementa la lógica recursiva del recorrido preorden. El orden de las operaciones es clave, primero procesa los hijos, luego el padre.

Recorrido postorden:

UNIVERSIDAD NACIONAL GUILLERMO BROWN

postorden recursivo: _postorden_recursivo(self, nodo, resultado):

El algoritmo sigue este patrón estricto para cada nodo que visita:

1. Explora la rama izquierda: Se sumerge recursivamente en la rama izquierda, hasta llegar al nodo hoja más profundo.
2. Explorar la rama derecha: Una vez que ha explorado toda la rama izquierda, se sumerge por la rama derecha, hasta el nodo hoja más profundo.
3. Procesar el nodo_actual (Raíz/Padre): Sólo después de que ambos árboles (izquierdo y derecho) han sido completamente visitados y añadidos al resultado, el nodo actual se "visita" así mismo, añadiendo su numero_vuelo a la lista resultado.

Resultado final:

El recorrido postorden genera una lista donde los nodos hoja aparecen primero, y la raíz del árbol aparece siempre al final de la lista.

Arbol General para Historias de vuelo de pasajero: Este árbol modela el historial de un pasajero, donde cada nodo es un vuelo y sus hijos son las posibles conexiones o escalas subsiguientes en ese viaje. No hay un límite fijo de hijos por nodo.

```
class VueloGeneral:
    def __init__(self, numero_vuelo, fecha, origen, destino):
        self.numero_vuelo = numero_vuelo
        self.fecha = fecha
        self.origen = origen
        self.destino = destino
        self.conexiones = []
```

Esta clase, VueloGeneral, redefine el concepto del nodo que vimos anteriormente. Mientras que las clases VueloBinario estaban diseñadas específicamente para un Árbol Binario (con exactamente dos punteros: izquierdo y derecho), esta clase está diseñada para un tipo de estructura de datos diferente y más flexible, un Árbol General.
La clase representa un vuelo específico en una red o ruta. A diferencia del árbol binario, donde cada vuelo solo podía tener dos "hijos" ordenados, aquí un vuelo puede tener un número ilimitado de conexiones o escalas subsiguientes. Modela una relación de "muchos a muchos" o "uno a muchos".

- self.numero(identificador único de vuelo)
- Self.fecha (fecha programada del vuelo)
- self.origen(ciudad o aeropuerto de salida)
- self.destino(ciudad o aeropuerto de llegada)
- self.conexiones(Una lista que almacena referencias (punteros) a otros nodos (VueloGeneral) que representan las posibles escalas o vuelos de conexión desde este destino).

```
    def __str__(self):
        """Representación en
cadena para facilitar la
impresión."""
        return f"Vuelo
{self.numero_vuelo}
({self.origen} -> {self.destino})
en {self.fecha}"

    def agregar_conexion(self,
vuelo):

self.conexiones.append(vuelo)
```

Lógica y Funcionamiento:

- Propósito: Ofrece una interfaz clara y controlada para modificar la lista self.conexiones. En lugar de que el usuario acceda directamente a la lista (vuelo_a.conexiones.append(vuelo_b)), se le proporciona un método de clase.
- Parámetro: Recibe un objeto vuelo (que se espera que sea otra instancia de VueloGeneral).
- Implementación: Simplemente utiliza el método append() de Python para añadir el objeto vuelo proporcionado a la lista self.conexiones del objeto actual.
- Ejemplo de uso:
- vuelo_madrid = VueloGeneral(AV200, "...", "BOG", "MAD")
- vuelo_general.agregar_conexion(vuelo_roma)

```python
class HistorialVuelos:
    def __init__(self, vuelo_inicial):
        self.raiz = vuelo_inicial
```

Propósito de la Clase

La clase HistorialVuelos tiene una única función que es establecer y mantener la referencia al primer vuelo o al vuelo principal que inicia una serie de conexiones.

En el contexto de las clases VueloGeneral que hemos visto (que tienen una lista conexiones), esta clase HistorialVuelos define *dónde comienza* la red de vuelos.

```
    def recorrer_historial(self):
        resultado = []

self._recorrer_recursivo(self.raiz,
resultado)
        return resultado
```

Estas funciones implementan un algoritmo de recorrido en profundidad (DFS) para la estructura de árbol general que creamos con la clase VueloGeneral.

El objetivo es visitar cada vuelo en la secuencia en que están conectados, desde la raíz hasta las conexiones más profundas.

Función recorrer_historial(self):

Este es el método público que el usuario llama para obtener la secuencia completa del historial de vuelos o rutas.

Inicialización: Crea una lista vacía llamada resultado, que almacenará las descripciones de los vuelos visitados.

Llamada a la recursión: Inicia el proceso llamando a la función auxiliar, _recorrer_recursivo comenzando desde el self.raiz (el vuelo inicial del historial) y pasandole la lista resultado.

Devolución: Una vez completado el recorrido por toda la red de vuelos, devuelve la lista resultado completa.

```
    def _recorrer_recursivo(self,
nodo, resultado):
        if nodo:
            resultado.append(f"Vuelo
{nodo.numero_vuelo} de {nodo.origen}
a {nodo.destino}")
            for conexion in
nodo.conexiones:

self._recorrer_recursivo(conexion,
resultado)
```

Esta función privada, que implementa la lógica recursiva del recorrido en profundidad (DFS). Es diferente que los recorridos inorden, preorden, postorden, porque un nodo puede tener múltiples hijos (conexiones) no solo dos.

función: _recorrer_recursivo(self, nodo, resultado):

Recorrido en profundidad:

Procesa el nodo actual:

- Inmediatamente añade la información del nodo actual a la lista resultado.

Explorar en profundidad las conexiones:

- Itera sobre la lista nodo.conexiones, para cada conexion encontrada.
- llama recursivamente a _recorrer_recursivo usando esa conexion como el nuevo nodo actual.
- esta llamada recursiva, se sumerge en esa rama completamente antes de volver y pasar a la siguiente conexion de la lista.

Resultado:

El resultado es una lista que muestra la ruta principal y luego explora todas las rutas alternativas o escalas en profundidad.

```
# Búsqueda (DFS - Profundidad)
    def
buscar_vuelo_general(self,
numero_vuelo):
        return
self._buscar_vuelo_recursivo_gen
eral(self.raiz, numero_vuelo)
```

Esta función buscar_vuelo_general se añade a la clase HistorialVuelos y sirve como punto de entrada público para encontrar un vuelo específico dentro de la red de conexiones, utilizando un algoritmo de Búsqueda en profundidad (DFS - Depth-First Search).

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Función buscar_vuelo_general(self, numero_vuelo):

Este es el método principal que el usuario llamará para iniciar la búsqueda.

Inicia la búsqueda:

- Simplemente delega la tarea a una función auxiliar privada, _buscar_vuelo_recursivo_general.

Punto de partida:

- Inicia la búsqueda desde la raíz del historial (self.raiz), que es el primer vuelo registrado.

Resultado:

- Devuelve el resultado que retorna la función recursiva o bien el objeto VueloGeneral encontrado, o None si el vuelo no está en el historial.

```
    def
_buscar_vuelo_recursivo_general (self, nodo,
numero_vuelo):
        if nodo is None:
            return None
        if nodo.numero_vuelo  ==
numero_vuelo:
            return nodo
        for conexion in nodo.conexiones:
            resultado  =
self._buscar_vuelo_recursivo_general(conexi
on, numero_vuelo)
            if resultado:
                return resultado
        return None
```

Función _buscar_vuelo_recursivo_general(self, nodo, numero_vuelo):

La función utiliza la recursión para sumergirse en cada rama de la red de vuelo, buscando el número de vuelo deseado.

Caso base 1: El nodo actual es nulo (fin del camino):

- Condición: if nodo is None:
- Si la función es llamada con un nodo vacío, (lo que pasaría si la raíz original fuera None o si una conexión específica no llevará a ninguna parte), simplemente retorna None. Esto detiene esa rama de la búsqueda.

Caso base 2: El vuelo fue encontrado:

- Condición: if nodo.numero_vuelo == numero_vuelo:
- Comprueba si el nodo que estamos visitando actualmente es el vuelo que estamos buscando.
- Si coincide, retorna el objeto nodo completo, esto detiene toda la búsqueda y pasa el resultado hacia arriba en la cadena de llamadas recursivas.

Exploración recursiva (búsqueda en profundidad):

- Si el nodo actual no es el vuelo buscado, entra en un bucle for para explorar todas sus posibles conexiones.
  `for conexion in nodo.conexiones:`
- dentro del bucle se realiza la llamada recursiva.
  `resultado = self._buscar_vuelo_recursivo_general(conexion, numero_vuelo)`
- Esta línea hace que la función se sumerja en la primera conexión y explore toda esa sub-red antes de pasar a la siguiente conexión en la lista.

Caso exitoso:

- Si el llamado recursivo (resultado) devuelve algo que no es None (es decir, que encontró el vuelo en una rama más profunda), lo retorna inmediatamente hacia arriba.

```
if resultado:
    return resultado
```

Fracaso en la búsqueda:

- Si el bucle for termina de iterar sobre todas las conexiones del nodo actual sin encontrar el vuelo (todas las llamadas recursivas devolvieron None)indicando que el vuelo no se encontró en esta rama del árbol.

```
return None
```

UNIVERSIDAD NACIONAL GUILLERMO BROWN

Resumen

La función _buscar_vuelo_recursivo_general explora la red de vuelos de forma exhaustiva. Visita un nodo, y luego explora en profundidad la primera conexión, luego la segunda, y así sucesivamente, hasta encontrar el vuelo o agotar todas las rutas posibles.