

# Ejercicios Pila

## 1. Implementación básica de una pila

- a) Crea una clase Pila con métodos push, pop, peek, is\_empty y size.
- b) Implementa la pila usando listas de Python.
- c) Modifica la pila para que tenga capacidad máxima y lance una excepción al superarla.
- d) Añade un método para imprimir la pila sin modificar su contenido.
- e) Implementa una función que invierta el contenido de una pila usando solo operaciones de pila.

## 2. Balanceo de paréntesis

- a) Verifica si una cadena de paréntesis (), [] y {} está balanceada.
- b) Extiende el ejercicio para incluir comentarios anidados /\* \*/.
- c) Devuelve la posición del primer paréntesis desbalanceado.
- d) Maneja etiquetas HTML como

y

.

- e) Permite otros delimitadores como < > y " ".

## 3. Conversión de expresiones

- a) Convierte una expresión infija a postfija (Notación Polaca Inversa).
- b) Evalúa una expresión en notación postfija.
- c) Maneja operadores unarios (ej: -5 en notación postfija).
- d) Extiende para soportar funciones matemáticas (sin, cos, log).
- e) Optimiza la conversión para expresiones con múltiples niveles de paréntesis.

## 4. Editor de texto con deshacer/rehacer

- a) Implementa un editor que permita insertar y borrar caracteres.
- b) Añade funcionalidad de "deshacer" usando una pila.
- c) Añade funcionalidad de "rehacer" usando una segunda pila.
- d) Limita el historial a los últimos 50 operaciones.
- e) Permite deshacer/rehacer operaciones en lote (ej: palabras completas).

## 5. Recorrido de laberinto

- a) Usa una pila para implementar DFS en un laberinto 2D.
- b) Encuentra todos los caminos posibles entre dos puntos.
- c) Marca el camino solución en el laberinto.
- d) Extiende a laberintos 3D (usando coordenadas x,y,z).
- e) Optimiza el algoritmo para evitar ciclos.

# 1. Implementación de una Pila (Stack)

La pila es el fundamento de todo lo que sigue. Es una estructura **LIFO** (*Last In, First Out*).

## Lógica

Imagina un tubo de ensayo: solo puedes añadir cosas por arriba y solo puedes sacar lo que está en la superficie. Si intentas sacar algo de un tubo vacío, rompes la lógica (*Underflow*); si intentas meter algo en un tubo lleno, se desborda (*Overflow*).

## Pseudocódigo

Plaintext

```
CLASE Pila:
    ATRIBUTOS: lista items, entero capacidad

    INICIALIZAR(limite):
        items = lista vacía
        capacidad = limite

    PROCEDIMIENTO push(valor):
        SI longitud(items) >= capacidad:
            LANZAR ERROR "Pila Llena"
        AGREGAR valor AL FINAL DE items

    FUNCION pop():
        SI items ESTÁ VACÍA:
            LANZAR ERROR "Pila Vacía"
        RETORNAR eliminar_ultimo(items)

    FUNCION peek():
        RETORNAR items[ultimo_indice]
```

## Código Python Comentado

Python

```
1 class Pila:
2     def __init__(self, capacidad_maxima=None):
3         # Inicializamos el contenedor de datos como una lista privada
4         self.items = []
5         # Guardamos el límite; si es None, la pila es "infinita"
6         # (limitada por RAM)
7         self.capacidad_maxima = capacidad_maxima
8
9     def is_empty(self):
```

```

9         # Evaluamos si la cantidad de elementos es igual a cero
10        return len(self.items) == 0
11
12    def push(self, item):
13        # Verificamos si existe un límite definido antes de insertar
14        if self.capacidad_maxima is not None:
15            # Si el tamaño actual ya llegó al tope, lanzamos una
            # excepción
16            if len(self.items) >= self.capacidad_maxima:
17                raise Exception("Error de Desbordamiento: Capacidad
                máxima superada")
18        # El método append coloca el elemento al final, que es nuestro
        # "Tope"
19        self.items.append(item)
20
21    def pop(self):
22        # Antes de extraer, debemos validar que existan elementos
23        if self.is_empty():
24            raise IndexError("Error de Subdesbordamiento: No hay
            elementos para extraer")
25        # pop() sin índice elimina y devuelve el último elemento
        # añadido
26        return self.items.pop()
27
28    def peek(self):
29        # Retorna el elemento en el tope sin modificar la estructura
30        if not self.is_empty():
31            # El índice -1 en Python siempre apunta al último elemento
32            return self.items[-1]
33        return None
34
35    def size(self):
36        # Devuelve la cantidad de elementos presentes mediante la
        # función len()
37        return len(self.items)
38
39    def __str__(self):
40        # Método especial para que al imprimir el objeto se vea el
        # contenido
41        # Invertimos la lista para que el tope sea lo primero que se
        # lea
42        return f"Pila: {self.items[::-1]}"

```

## 2. Balanceo de Paréntesis y Delimitadores

Este es el corazón de cómo los compiladores revisan tu código.

## Lógica

Cada vez que abres un paréntesis, creas una "deuda". Esa deuda solo se paga con el paréntesis de cierre **correspondiente**. Si cierras un corchete `]` cuando el último que abriste fue un paréntesis `(`, el código está mal formado. La pila nos permite recordar el orden exacto de las aperturas.

## Pseudocódigo

Plaintext

```
FUNCION verificar(cadena):
    pila = NUEVA Pila
    PARA cada caracter EN cadena:
        SI caracter ES DE APERTURA:
            pila.push(caracter)
        SINO SI caracter ES DE CIERRE:
            SI pila esta vacía: RETORNAR Error (Cierre huérfano)
            ultimo = pila.pop()
            SI ultimo NO COINCIDE con caracter: RETORNAR Error (Cruce de
símbolos)
    RETORNAR pila.is_empty()
```

## Código Python Comentado

Python

```
1  def validador_completo(cadena):
2      # Creamos nuestra instancia de Pila
3      pila = Pila()
4      # Mapa de correspondencia: la llave es el cierre, el valor es la
apertura esperada
5      diccionario_pares = {')': '(', ']': '[', '}': '{', '>': '<'}
6
7      # Recorremos la cadena usando enumerate para rastrear el índice
(la posición)
8      for indice, caracter in enumerate(cadena):
9          # Si el caracter es uno de los valores del diccionario
(aperturas)
10         if caracter in diccionario_pares.values():
11             # Lo guardamos en la pila para esperar su cierre posterior
12             pila.push((caracter, indice))
13
```

```

14         # Si el caracter es una de las llaves del diccionario
           (cierres)
15     elif caracter in diccionario_pares:
16         # Error 1: Si encontramos un cierre pero no hay nada
           abierto en la pila
17         if pila.is_empty():
18             return f"Error: Símbolo '{caracter}' en pos {indice}
               no tiene apertura."
19
20         # Sacamos el último símbolo que se abrió
21         ultimo_abierto, pos_abierta = pila.pop()
22
23         # Error 2: Si el cierre actual no corresponde al tipo del
           último abierto
24         if ultimo_abierto != diccionario_pares[caracter]:
25             return f"Error: El símbolo '{caracter}' en pos
               {indice} cierra incorrectamente a '{ultimo_abierto}'
               de pos {pos_abierta}."
26
27         # Al terminar el ciclo, si la pila no está vacía, quedaron
           símbolos sin cerrar
28         if not pila.is_empty():
29             simbolo, pos = pila.pop()
30             return f"Error: El símbolo '{simbolo}' de la posición {pos}
               nunca se cerró."
31
32     return "¡Cadena balanceada correctamente!"

```

### 3. Conversión de Expresiones (Infija a Postfija)

Transformar `3 + 4` en `3 4 +`.

#### Lógica

Los humanos usamos notación **Infija** (operador en el medio), pero es ambigua sin paréntesis. La **Postfija** (RPN) no necesita paréntesis porque el orden de los operandos y operadores define la jerarquía. El algoritmo usa la pila para retener los operadores hasta que sea su turno de actuar según su precedencia.

#### Pseudocódigo

Plaintext

PARA cada token EN expresion:  
 SI token ES numero: AGREGAR a salida

```
SINO SI token ES '(': push a pila
SINO SI token ES ')': pop de pila a salida HASTA encontrar '('
SINO (es operador):
    MIENTRAS tope de pila TENGA mayor/igual prioridad:
        pop de pila a salida
    push token a pila
```

## Código Python Comentado

Python

```
1  def infija_a_postfija(expresion):
2      # Definimos la jerarquía matemática (Precedencia)
3      # A mayor número, más prioridad tiene el operador
4      prioridad = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3, 'sin': 4,
5                  'cos': 4}
6      pila = Pila()
7      salida = []
8      # Dividimos la cadena por espacios para separar números de
9      operadores
10     tokens = expresion.split()
11
12     for t in tokens:
13         # Si el token es un número (detectamos si es dígito)
14         if t.replace('.', '', 1).isdigit():
15             # Los números pasan directo a la lista de salida
16             salida.append(t)
17         elif t == '(':
18             # El paréntesis de apertura siempre se apila
19             pila.push(t)
20         elif t == ')':
21             # Al cerrar, sacamos todo de la pila hasta llegar al de
22             apertura
23             while not pila.is_empty() and pila.peek() != '(':
24                 salida.append(pila.pop())
25             # Eliminamos el '(' de la pila
26             pila.pop()
27         else:
28             # Si es un operador, comparamos su prioridad con el que ya
29             está en el tope
30             while (not pila.is_empty() and pila.peek() != '(' and
31                   prioridad.get(pila.peek(), 0) >= prioridad.get(t,
32                           0)):
33                 # Si el de la pila es más importante, sale primero a
34                 la salida
35                 salida.append(pila.pop())
```

```

30         # Finalmente, el nuevo operador entra a la pila
31         pila.push(t)
32
33     # Al terminar, vaciamos lo que haya quedado en la pila a la salida
34     while not pila.is_empty():
35         salida.append(pila.pop())
36
37     return " ".join(salida)

```

## 4. Editor de Texto (Undo/Redo)

### Lógica

Mantenemos el estado actual del texto.

- **Deshacer (Undo):** Guardamos el estado "viejo" antes de cambiarlo.
- **Rehacer (Redo):** Si deshicimos algo, ese estado se guarda en otra pila por si el usuario se arrepiente de haber deshecho.

### Pseudocódigo

Plaintext

```

PROCEDIMIENTO escribir(nuevo_texto):
    PilaUndo.push(texto_actual)
    VACIAR PilaRedo // Nueva acción rompe el historial de rehacer
    texto_actual = nuevo_texto

PROCEDIMIENTO deshacer():
    PilaRedo.push(texto_actual)
    texto_actual = PilaUndo.pop()

```

### Código Python Comentado

Python

```

1  class Editor:
2      def __init__(self):
3          self.texto = ""
4          # Pila para retroceder en el tiempo (Undo)
5          self.pila_undo = Pila(capacidad_maxima=50)
6          # Pila para avanzar en el tiempo (Redo)
7          self.pila_redo = Pila(capacidad_maxima=50)
8

```

```

9     def realizar_accion(self, nuevo_contenido):
10         # Guardamos el texto actual en el historial de deshacer
11         self.pila_undo.push(self.texto)
12         # Importante: Si el usuario escribe algo nuevo, la pila de
13         # Redo se limpia
14         # porque hemos creado un "presente" diferente
15         while not self.pila_redo.is_empty():
16             self.pila_redo.pop()
17         self.texto = nuevo_contenido
18         print(f"Estado Actual: '{self.texto}'")
19
20     def undo(self):
21         if not self.pila_undo.is_empty():
22             # Guardamos lo que vamos a borrar en Redo
23             self.pila_redo.push(self.texto)
24             # Restauramos el estado anterior
25             self.texto = self.pila_undo.pop()
26             print(f"Undo realizado: '{self.texto}'")
27
28     def redo(self):
29         if not self.pila_redo.is_empty():
30             # Volvemos a guardar el estado actual en Undo
31             self.pila_undo.push(self.texto)
32             # Recuperamos lo que habíamos deshecho
33             self.texto = self.pila_redo.pop()
34             print(f"Redo realizado: '{self.texto}'")

```

## 5. Laberinto DFS (Depth First Search)

### Lógica

El DFS es como entrar a una cueva con un hilo: vas lo más profundo que puedes por un camino. Si llegas a una pared, regresas por el hilo hasta la última intersección y pruebas otro camino. La pila guarda ese "hilo" de coordenadas.

### Pseudocódigo

Plaintext

```

pila.push(coordenada_inicio)
MIENTRAS pila NO VACÍA:
    actual = pila.pop()
    SI actual == meta: RETORNAR camino
    PARA cada vecino DE actual:
        SI vecino ES CAMINO Y NO VISITADO:

```

```
marcar vecino como visitado
pila.push(vecino)
```

## Código Python Comentado

Python

```
1  def resolver_laberinto_3d(laberinto, inicio, fin):
2      # laberinto[z][y][x] -> 0 es pasillo, 1 es pared
3      pila = Pila()
4      # Guardamos rutas completas en la pila: [(0,0,0)], [(0,0,0),
5          (0,0,1)]
6      pila.push([inicio])
7      visitados = {inicio} # Set para búsqueda rápida O(1)
8
9      while not pila.is_empty():
10         # Obtenemos la ruta que estamos explorando actualmente
11         ruta = pila.pop()
12         # Nuestra posición actual es el último elemento de esa ruta
13         z, y, x = ruta[-1]
14
15         # Condición de victoria: llegamos al objetivo
16         if (z, y, x) == fin:
17             return ruta
18
19         # Definimos las 6 direcciones posibles (3D: arriba, abajo, N,
20             S, E, O)
21         for dz, dy, dx in [(1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1),
22             (0,0,-1)]:
23             nz, ny, nx = z + dz, y + dy, x + dx
24
25             # 1. Validar que la nueva coordenada esté dentro del mapa
26             if (0 <= nz < len(laberinto) and
27                 0 <= ny < len(laberinto[0]) and
28                 0 <= nx < len(laberinto[0][0])):
29
30                 # 2. Validar que sea un espacio transitable y no
31                     hayamos pasado por ahí
32                 if laberinto[nz][ny][nx] == 0 and (nz, ny, nx) not in
33                     visitados:
34                     # Marcamos como visitado para no entrar en bucles
35                         infinitos
36                     visitados.add((nz, ny, nx))
37                     # Creamos una nueva ruta extendida
38                     nueva_ruta = list(ruta)
39                     nueva_ruta.append((nz, ny, nx))
```

```
34         # La metemos a la pila para explorarla después
35         pila.push(nueva_ruta)
36
37     return "No se encontró salida"
```

Claro que sí. Si las Pilas eran como un tubo de ensayo, las **Colas (Queues)** son como la fila del supermercado: el primero que llega es el primero en ser atendido. Este principio se llama **FIFO** (*First In, First Out*).

Aquí tienes la guía definitiva, explicada paso a paso, con lógica, pseudocódigo y código Python ultra-comentado.

---

## Ejercicios Cola

1. Implementación básica de cola
  - a) Crea una clase Cola con métodos enqueue, dequeue, front, is\_empty y size.
  - b) Implementa una cola usando listas de Python.
  - c) Implementa una cola circular con arreglo de tamaño fijo.
  - d) Añade un método para invertir la cola.
  - e) Implementa una cola doblemente terminada (deque).
2. Simulación de banco
  - a) Simula una fila de banco con un solo cajero.
  - b) Extiende a múltiples cajeros con una cola única.
  - c) Calcula tiempo promedio de espera de clientes.
  - d) Implementa prioridad para clientes preferenciales.
  - e) Genera reportes de métricas de eficiencia.
3. Búsqueda en anchura (BFS)
  - a) Implementa BFS en un grafo no dirigido.
  - b) Encuentra el camino más corto entre dos nodos.
  - c) Extiende BFS para grafos ponderados (peso uniforme).
  - d) Cuenta el número de componentes conexos.
  - e) Encuentra nodos a distancia k de un nodo origen.
4. Buffer de mensajes
  - a) Implementa un buffer FIFO para mensajes.
  - b) Añade prioridad a mensajes urgentes.
  - c) Permite consultar mensajes sin eliminarlos.
  - d) Implementa expiración de mensajes antiguos.
  - e) Sincroniza el buffer para acceso concurrente.
5. Impresión de documentos
  - a) Simula una cola de impresión con múltiples impresoras.

- b)Asigna trabajos a la impresora menos ocupada.
- c)Permite pausar/reanudar la cola de impresión.
- d)Implementa cancelación de trabajos pendientes.
- e)Genera reportes de trabajos completados.

#### 6. Sistema de turnos

- a)Simula una fila virtual para asignación de turnos.
- b)Permite ingresar/eliminar personas de la cola.
- c)Notifica cuando el turno de una persona está próximo.
- d)Implementa turnos prioritarios (embarazadas, discapacitados).
- e)Genera códigos QR para turnos electrónicos.

## 1. Implementación Básica de una Cola

### Lógica Conceptual

Una cola estándar añade elementos por el final (*Rear*) y los elimina por el frente (*Front*).

### Pseudocódigo (Cola Circular)

Plaintext

```
CLASE ColaCircular:
    INICIALIZAR(tamaño):
        arreglo = [Nulo] * tamaño
        frente = 0, final = -1, cantidad = 0

    PROCEDIMIENTO enqueue(valor):
        SI cantidad == tamaño: LANZAR ERROR "Llena"
        final = (final + 1) MODULO tamaño
        arreglo[final] = valor
        cantidad = cantidad + 1

    FUNCION dequeue():
        SI cantidad == 0: LANZAR ERROR "Vacía"
        valor = arreglo[frente]
        frente = (frente + 1) MODULO tamaño
        cantidad = cantidad - 1
        RETORNAR valor
```

### Código Python Comentado

Python

```
1 | class ColaCircular:
```

```

2     def __init__(self, capacidad):
3         # Arreglo de tamaño fijo inicializado con None
4         self.capacidad = capacidad
5         self cola = [None] * capacidad
6         # Punteros para rastrear el inicio y el fin
7         self.frente = 0
8         self.final = -1
9         self.tamaño_actual = 0
10
11    def enqueue(self, dato):
12        # Verificamos si la cola está llena
13        if self.tamaño_actual == self.capacidad:
14            raise OverflowError("La cola circular está llena")
15
16        # Movemos el puntero final de forma circular usando el
17        # operador módulo %
18        # Ejemplo: (4 + 1) % 5 = 0 (vuelve al inicio)
19        self.final = (self.final + 1) % self.capacidad
20        self.col[self.final] = dato
21        self.tamaño_actual += 1
22
23    def dequeue(self):
24        # Verificamos si hay algo que sacar
25        if self.tamaño_actual == 0:
26            raise IndexError("La cola está vacía")
27
28        dato = self.col[self.frente]
29        # Limpiamos el espacio (opcional) y movemos el frente
30        # circularmente
31        self.col[self.frente] = None
32        self.frente = (self.frente + 1) % self.capacidad
33        self.tamaño_actual -= 1
34        return dato
35
36    def invertir(self):
37        # Para invertir una cola, la forma más elegante es usar una
38        # Pila auxiliar
39        pila_auxiliar = []
40        while self.tamaño_actual > 0:
41            pila_auxiliar.append(self.dequeue())
42        while pila_auxiliar:
43            self.enqueue(pila_auxiliar.pop())

```

## 2. Simulación de Banco (Múltiples Cajeros)

## Lógica Conceptual

Necesitamos medir el tiempo. Cada cliente tiene un `tiempo_llegada` y un `tiempo_servicio`. Si hay 3 cajeros, el cliente va al primero que esté libre. El tiempo de espera es:

$$\text{Espera} = \text{Tiempo Inicio Servicio} - \text{Tiempo Llegada}$$

## Pseudocódigo

Plaintext

```
MIENTRAS haya clientes O cajeros ocupados:
  SI cliente llega:
    AGREGAR a ColaUnica
  PARA cada cajero:
    SI cajero libre Y ColaUnica no vacía:
      cliente = ColaUnica.dequeue()
      tiempo_espera = tiempo_actual - cliente.llegada
      cajero.ocupado_hasta = tiempo_actual + cliente.servicio
```

## Código Python Comentado

Python

```
1  import collections
2
3  class Cliente:
4      def __init__(self, id, llegada, servicio, es_preferencial=False):
5          self.id = id
6          self.llegada = llegada
7          self.servicio = servicio
8          self.es_preferencial = es_preferencial
9
10 def simular_banco(lista_clientes, num_cajeros):
11     # Separamos en dos colas para manejar la prioridad
12     cola_prioridad = collections.deque([c for c in lista_clientes if
13                                         c.es_preferencial])
14     cola_normal = collections.deque([c for c in lista_clientes if not
15                                     c.es_preferencial])
16
17     # Lista que guarda cuándo se libera cada cajero (inicialmente en
18     # tiempo 0)
19     cajeros_libres_en = [0] * num_cajeros
20     tiempos_espera = []
```

```

19     # Mientras existan clientes en cualquiera de las dos colas
20     while cola_prioridad or cola_normal:
21         # Atendemos primero a los preferenciales
22         actual = cola_prioridad.popleft() if cola_prioridad else
            cola_normal.popleft()
23
24         # Buscamos el cajero que se libere más pronto
25         indice_cajero =
            cajeros_libres_en.index(min(cajeros_libres_en))
26
27         # El servicio inicia cuando el cajero está libre 0 cuando el
            cliente llega (lo que sea último)
28         inicio_servicio = max(actual.llegada,
            cajeros_libres_en[indice_cajero])
29         espera = inicio_servicio - actual.llegada
30         tiempos_espera.append(espera)
31
32         # El cajero ahora estará ocupado hasta que termine este
            servicio
33         cajeros_libres_en[indice_cajero] = inicio_servicio +
            actual.servicio
34
35         promedio = sum(tiempos_espera) / len(tiempos_espera)
36         return f"Tiempo promedio de espera: {promedio:.2f} min"

```

### 3. Búsqueda en Anchura (BFS)

#### Lógica Conceptual

El BFS explora un grafo por **niveles**. Empieza en el nodo A, luego visita todos los vecinos directos de A, luego los vecinos de los vecinos, etc. La cola garantiza que no visitemos un nodo lejano antes que uno cercano.

#### Pseudocódigo

Plaintext

```

FUNCION BFS(inicio, fin):
    cola.enqueue([inicio]) # Guardamos rutas
    visitados = {inicio}
    MIENTRAS cola no vacía:
        ruta = cola.dequeue()
        nodo = ruta.ultimo
        SI nodo == fin: RETORNAR ruta
        PARA cada vecino DE nodo:

```

```
SI vecino no visitado:
    visitados.add(vecino)
    cola.enqueue(ruta + vecino)
```

## Código Python Comentado

Python

```
1  def bfs_camino_corto(grafo, inicio, objetivo):
2      # La cola guarda caminos completos para recuperar la ruta al final
3      cola = collections.deque([[inicio]])
4      visitados = {inicio}
5
6      while cola:
7          # Sacamos el camino más antiguo (el primero que entró)
8          camino = cola.popleft()
9          # Tomamos el último nodo de ese camino para explorar sus
            vecinos
10         nodo_actual = camino[-1]
11
12         if nodo_actual == objetivo:
13             return camino # Retornamos la lista de nodos que forman la
                ruta
14
15         # Exploramos los vecinos del nodo actual en el grafo
16         for vecino in grafo.get(nodo_actual, []):
17             if vecino not in visitados:
18                 visitados.add(vecino)
19                 # Creamos un nuevo camino extendido y lo encolamos
20                 nuevo_camino = list(camino)
21                 nuevo_camino.append(vecino)
22                 cola.append(nuevo_camino)
23         return None
```

---

## 4. Buffer de Mensajes (FIFO con Expiración)

### Lógica Conceptual

Imagina un chat. Los mensajes llegan y se encolan. Pero si un mensaje tiene más de  $X$  segundos, ya no es relevante y debe borrarse. Usamos marcas de tiempo (*timestamps*).

## Código Python Comentado

Python

```

1  import time
2
3  class MessageBuffer:
4      def __init__(self, tiempo_vida):
5          self.buffer = collections.deque()
6          self.ttl = tiempo_vida # Time To Live en segundos
7
8      def enviar_mensaje(self, contenido, prioridad=0):
9          # Guardamos el mensaje con el tiempo actual del sistema
10         # Prioridad alta (1) va al frente, normal (0) al final
11         msg = {'texto': contenido, 'tiempo': time.time(), 'prio':
12               prioridad}
13         if prioridad == 1:
14             self.buffer.appendleft(msg) # Cola Doble (Deque)
15             # permitiendo entrar por delante
16         else:
17             self.buffer.append(msg)
18
19     def limpiar_expirados(self):
20         ahora = time.time()
21         # Mientras el mensaje más antiguo haya superado el tiempo de
22         # vida
23         while self.buffer and ahora - self.buffer[0]['tiempo'] >
24             self.ttl:
25             # Lo eliminamos silenciosamente
26             self.buffer.popleft()
27
28     def leer_siguiente(self):
29         self.limpiar_expirados() # Mantenemos el buffer fresco
30         return self.buffer.popleft() if self.buffer else None

```

## 5. Cola de Impresión

### Lógica Conceptual

Múltiples impresoras son como múltiples cajeros. La estrategia aquí es **Balanceo de Carga**: asignar el nuevo documento a la impresora que tenga menos páginas pendientes por imprimir.

### Código Python Comentado

Python

```

1  class Impresora:
2      def __init__(self, nombre):

```

```

3         self.nombre = nombre
4         self.trabajos = collections.deque()
5         self.esta_pausada = False
6
7     def calcular_carga(self):
8         # Sumamos el total de páginas de todos los trabajos en su cola
9         return sum(t['paginas'] for t in self.trabajos)
10
11 class SistemaImpresion:
12     def __init__(self, lista_nombres):
13         self.impresoras = [Impresora(n) for n in lista_nombres]
14
15     def enviar_a_imprimir(self, doc_id, num_paginas):
16         # Buscamos la impresora con menor carga de trabajo acumulada
17         destino = min(self.impresoras, key=lambda imp:
18             imp.calcular_carga())
19         destino.trabajos.append({'id': doc_id, 'paginas':
20             num_paginas})
21         print(f"Documento {doc_id} enviado a {destino.nombre}")

```

## 6. Sistema de Turnos Virtuales

### Lógica Conceptual

El sistema debe permitir que alguien "saque un número". Si es una persona mayor o embarazada, su "número" se salta la fila. Usamos dos estructuras: una cola para la lógica y un generador de identificadores únicos.

### Código Python Comentado

Python

```

1 class SistemaTurnos:
2     def __init__(self):
3         self.cola = collections.deque()
4         self.contador = 1
5
6     def sacar_turno(self, nombre, prioridad=False):
7         # Formateamos el código del turno
8         codigo = f"{'P' if prioridad else 'G'}-{self.contador:03d}"
9         self.contador += 1
10
11         datos = {"nombre": nombre, "codigo": codigo}
12
13         if prioridad:

```

```
14         # Los turnos prioritarios se insertan al principio para
           ser atendidos de inmediato
15         self.cola.appendleft(datos)
16     else:
17         self.cola.append(datos)
18     return codigo
19
20     def atender_siguiente(self):
21         if not self.cola:
22             return "No hay nadie en espera."
23         atendido = self.cola.popleft()
24         # Notificación al siguiente (simulada)
25         if self.cola:
26             print(f"AVISO: {self.cola[0]['nombre']}, prepare su código
                {self.cola[0]['codigo']}")
27         return f"Atendiendo ahora a: {atendido['nombre']}
                ({atendido['codigo']})"
```