

Examen Parcial de Programación 2

Tiempo mínimo para el examen: 1 hora reloj.

Tiempo máximo para el examen: 2 horas reloj.

1. Sistema de Gestión de Atención de Clientes en un Banco

El sistema permitirá gestionar tres tipos de operaciones en un banco (atención por caja, atención comercial y atención de préstamos) utilizando colas de prioridad. La prioridad de los clientes se establecerá según el tipo de cuenta que posean.

Tipos de cuentas y prioridades:

- **Cuenta Premium:** Prioridad 1 (más alta)
- **Cuenta Plata:** Prioridad 2
- **Cuenta Básica:** Prioridad 3 (más baja)

Funcionamiento básico del sistema:

Cuando llegue un cliente al banco, deberá indicar qué tipo de operación desea realizar (caja, comercial o préstamo). Teniendo en cuenta el tipo de operación que desea realizar y la prioridad de su cuenta, se agregará al cliente a la cola de prioridad que corresponda.

Implementación del sistema:

Deberá completar la implementación de las siguientes clases, que ayudarán al desarrollo del sistema de gestión del banco:

1. Clase ColaPrioridad:

- **Atributos:**
 - `elementos (list[Any, int])`: Lista que almacena los elementos en la cola, junto a su prioridad, ordenados por su prioridad.
- **Métodos:**
 - `agregar_elemento`: Agrega un elemento y su prioridad a la cola de prioridad.
 - `atender_elemento`: Remueve y devuelve al elemento el primer elemento de la cola.
 - `__str__`: Devuelve una representación en cadena de la cola.

2. Clase Cliente:

- **Atributos:**
 - `dni (str)`: DNI del cliente.
 - `nombre (str)`: Nombre del cliente.
 - `tipo_cuenta (str)`: Tipo de cuenta (premium, plata, básica).
- **Métodos:**
 - `__str__`: Devuelve una representación en cadena del cliente.

3. Clase SistemaGestionBanco:

- **Atributos:**
 - `cola_caja (ColaPrioridad)`: Cola para atención por caja.
 - `cola_comercial (ColaPrioridad)`: Cola para atención comercial.
 - `cola_prestamos (ColaPrioridad)`: Cola para atención de préstamos.
- **Métodos:**
 - `agregar_cliente`: Agrega un cliente a la cola correspondiente según el tipo de operación que desea realizar.
 - `atender_cliente`: Atiende al cliente de la cola correspondiente según el tipo de operación.
 - `mostrar_colas`: Muestra los clientes en cada cola de prioridad.

```

class ColaPrioridad:
    def __init__(self):
        self.elementos: list[Any, int] = []

    def agregar_elemento(self, dato: Any, prioridad: int) -> None:
        # Añade el dato y su prioridad como una tupla
        self.elementos.append((dato, prioridad))
        # Ordena la lista de elementos por prioridad
        # La sgte línea lo que hace es ordenar las tuplas de menor a mayor
        # en función del valor de la segunda componente de la tupla
        # Recordar que un valor entero menor implica mayor prioridad
        self.elementos.sort(key=lambda x: x[1])

    def quitar_elemento(self) -> Any:
        # Debe devolver el dato de mayor prioridad de la cola
        # o None en caso de estar vacía
        # completar
        pass

    def __str__(self) -> str:
        #completar
        pass

class Cliente:
    def __init__(self, dni: str, nombre: str, tipo_cuenta: str):
        self.dni = dni
        self.nombre = nombre
        self.tipo_cuenta = tipo_cuenta

    def __str__(self) -> str:
        return f"Cliente: {self.nombre}, DNI: {self.dni}, \
                Tipo de cuenta: {self.tipo_cuenta}"

class SistemaGestionBanco:
    def __init__(self):
        self.cola_caja = ColaPrioridad() # Cola para atención por caja
        self.cola_comercial = ColaPrioridad() # Cola para atención comercial
        self.cola_prestamos = ColaPrioridad() # Cola para atención de
                                                # préstamos

    def calcular_prioridad(self, tipo_cuenta: str) -> int:
        # calcula la prioridad según el tipo de cuenta
        # completar
        pass

    def agregar_cliente(self, cliente: Cliente, tipo_operacion: str) -> None:
        # Agrega al cliente a la cola de espera que corresponda,
        # según el tipo de operación que desea realizar
        # Recuerde calcular la prioridad usando el método correspondiente
        # completar
        pass

```

```

def atender_cliente(self, tipo_operacion: str) -> None:
    # Quita a un cliente de una de las colas de espera
    # teniendo en cuenta el tipo de operación
    # Debe mostrar en pantalla el nombre del cliente que está
    # siendo atendido o indicar que no hay clientes en la cola
    # completar
    pass

def mostrar_colas(self) -> None:
    # Muestra en pantalla los clientes en cada cola
    # completar
    pass

# Ejemplo de uso
sistema_banco = SistemaGestionBanco()

# Agregar clientes a las diferentes colas
sistema_banco.agregar_cliente(Cliente("12345678", "Juan Pérez", "premium"),
                               "caja")
sistema_banco.agregar_cliente(Cliente("87654321", "María López", "plata"),
                               "comercial")
sistema_banco.agregar_cliente(Cliente("23456789", "Carlos Gómez", "basica"),
                               "prestamo")
sistema_banco.agregar_cliente(Cliente("34567890", "Ana Torres", "plata"),
                               "caja")
sistema_banco.agregar_cliente(Cliente("45678901", "Luis Fernández", "premium"),
                               "prestamo")

# Mostrar los clientes en cada cola
sistema_banco.mostrar_colas()
# Caja:
# Cliente: DNI: 12345678, Nombre: Juan Pérez, Tipo Cuenta: premium
# Cliente: DNI: 34567890, Nombre: Ana Torres, Tipo Cuenta: plata
# Comercial:
# Cliente: DNI: 87654321, Nombre: María López, Tipo Cuenta: plata
# Préstamo:
# Cliente: DNI: 45678901, Nombre: Luis Fernández, Tipo Cuenta: premium
# Cliente: DNI: 23456789, Nombre: Carlos Gómez, Tipo Cuenta: basica

# Atender a un cliente de cada cola
print("\nAtendiendo clientes...")
sistema_banco.atender_cliente("caja")
# Atendido: Juan Pérez
sistema_banco.atender_cliente("comercial")
# Atendido: María López
sistema_banco.atender_cliente("prestamo")
# Atendido: Luis Fernández

# Mostrar las colas después de atender algunos clientes
sistema_banco.mostrar_colas()
# Caja:
# Cliente: DNI: 34567890, Nombre: Ana Torres, Tipo Cuenta: plata
# Comercial:
# Sin clientes
# Préstamo:
# Cliente: DNI: 23456789, Nombre: Carlos Gómez, Tipo Cuenta: basica

```

2. Suponga que existe ya una implementación de la clase `Usuario` que sirve para representar un usuario en una red social. Implemente una clase `RedSocial` que represente las conexiones de amistad entre diferentes usuarios de una red social como un grafo no dirigido:

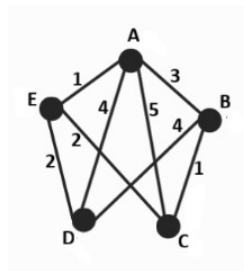
```
class RedSocial:
    def __init__(self):
        self.usuarios = []
        self.amistades = {}
```

Métodos a implementar:

1. `agregar_usuario(self, usuario: Usuario) -> None`: Agrega un nuevo usuario a la red.
2. `agregar_amistad(self, usuario1: Usuario, usuario2: Usuario) -> None`: Establece una amistad entre dos usuarios.
3. `eliminar_usuario(self, usuario: Usuario) -> None`: Elimina el usuario de la red social (tenga en cuenta que los vínculos de amistad de este usuario deben dejar de existir).
4. `get_amistades(self, usuario: Usuario) -> list[Usuario]`: Devuelve una lista de amigos de un usuario.

Realice todos los chequeos necesarios en su implementación para que el sistema funcione sin errores. Ejemplo, en el método `agregar_amistad` deben existir los dos usuarios para poder conectarlos.

3. Dado un grafo G de la siguiente figura que representa una red de transporte público:



encuentre el camino más corto desde la estación **A** a la estación **D** utilizando el algoritmo de Dijkstra.