Nombre y Apellido: Legajo:

Examen Parcial de Programación 2

Tiempo mínimo para el examen: 1 hora reloj.
Tiempo máximo para el examen: 2 horas reloj.

El examen se compone los siguientes ejercicios.

1. **Navegación en un navegador web**

   Los navegadores web permiten a los usuarios navegar hacia adelante y hacia atrás entre las páginas visitadas.

   Funcionamiento básico de un navegador:

   1. Cuando el usuario visita una *nueva página*, la *página actual* se guarda en el *historial* y la *nueva página* se conviene en la *página actual*. El *avance* se vacía.
   2. Al hacer clic en "Atrás", recupera la última página guardada en el *historial* (si la hay) y se convierte en la *página actual*. La *página actual* hasta ese momento pasa a formar parte del *avance*.
   3. Al hacer clic en "Adelante", recupera la última página guardada en el *avance* (si la hay) y se convierte en la *página actual*. La *página actual* hasta ese momento pasa a formar parte del *historial*.

   Considere que ya existen las implementaciones de:

   - una clase `Pila` que provee los métodos `__init__`, `apilar`, `desapilar` y `esta_vacia` que Ud. conoce;
   - una clase `Cola` que provee los métodos `__init__`, `encolar`, `desencolar` y `esta_vacia` que Ud. conoce; y
   - una clase `Pagina` que guarda los datos de la página web y sus métodos, incluyendo el método `__str__`.

   ¿Puede implementar la clase `Navegador`, que implemente las funcionalidades básicas explicadas más arriba? Para ello debe completar el código que se le propone a continuación. Puede hacer uso de las clases `Pila`, `Cola` y/o `Pagina` sin implementarlas.

```
class Navegador:
    def __init__(self):
        self.historial = ...          #considere que el historial comienza vacio
        self.avance = ...             #considere que el avance comienza vacio
        self.pagina_actual = None

    def visitar(self, pagina: 'Pagina'):    #considere que existe una clase Pagin
        pass                                #ya definida

    def atras(self):
        pass

    def adelante(self):
        pass

    def mostrar_página_actual(self) -> None
        '''
            Muestra en pantalla la información de la página actual
        '''
        pass
```

2. Considere la clase vista en clase `BinaryTree`:

```
class BinaryTree:
     def __init__(self, cargo=None, left=None, right=None):
         self.cargo = cargo
         self.left = left
         self.right = right

def cantidad_hojas(btree):
    ...

def numeros_entre(btree, l, r):
   ...
```

   1. Implemente una función que reciba un `BinaryTree` y devuelva la cantidad de hojas (i.e. nodos terminales) del árbol.
   2. Implemente una función que reciba un `BinaryTree` de números, y dos números `l` y `r`, devuelva una lista con todos los números entre `l` y `r` inclusive que se encuentran en el árbol.

3. Considere la siguiente implementación del TAD Grafo Dirigido:

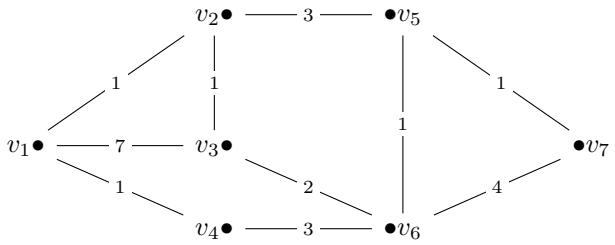
```
from typing import Any

class GrafoDirigido:
    def __init__(self) -> None:
        self.vertices = []
        self.vecinos = {}

    def add_node(self, vertice: Any) -> None:
        self.vertices.append(vertice)
        self.vecinos[vertice] = []

    def add_edge(self, vertice1: Any, vertice2: Any) -> None:
        self.vecinos[vertice1].append(vertice2)

    def get_adjacent(self, vertice: Any) -> Any:
        return self.vecinos[vertice]

    def get_nodes(self) -> list[Any]:
        return self.vertices

# Ejemplo de uso
grafo = GrafoDirigido()
grafo.add_node("A")
grafo.add_node("B")
grafo.add_node("C")
grafo.add_edge("A", "B")
grafo.add_edge("A", "C")
grafo.add_edge("B", "C")
print("Vértices:", grafo.get_nodes())
print("Vecinos de A:", grafo.get_adjacent("A"))
print("Vecinos de B:", grafo.get_adjacent("B"))
print("Vecinos de C:", grafo.get_adjacent("C"))
```

   Completa la implementación agregando los siguientes métodos:

   1. `remove_edge(x, y)`: Remueve la arista dirigida entre el nodo x y el nodo y (si existe).

2. `remove_node(x)`: Remueve el nodo x del grafo. Si había aristas que salían o llegaban a este nodo, también deben borrar del grafo.
3. `are_adjacent(x, y)`: Devuelve True si x apunta a y, False en caso contrario.
4. `get_outdegree(v)`: Devuelve el grado de salida (outdegree) del vértice, que es el número de aristas que salen de él.
5. `get_indegree(v)`: Devuelve el grado de salida (indegree) del vértice, que es el número de aristas que llegan a él.

4. Dado el siguiente grafo G



a) Encontrar la longitud del camino más corto entre $\mathbf{v_1}$ y $\mathbf{v_7}$ utilizando el algoritmo de Dijkstra.

b) Aplicar el algoritmo de Prim para encontrar el árbol de expansión mínimo del grafo comenzando por el nodo $\mathbf{v_3}$.