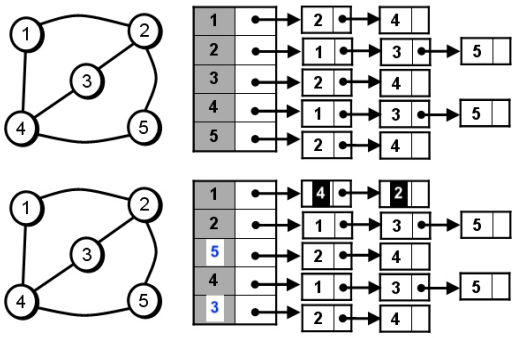
Tecnología de la Programación **Junio de 2022**
Duración: 2h:30' Evaluación Final.

NOTA IMPORTANTE: Aquellos ejercicios, que, a pesar de estar resueltos correctamente, no estén razonados por el alumno, se considerarán no válidos. **Comenta el código**. Controla los parámetros de entrada **sin usar assert**.

Apellidos y Nombre: .....

Ejercicio 1 (6 puntos + 1 punto extra) Una forma de representar el TDA Grafo es mediante **listas de adyacencias**. Es decir, mediante una lista de nodos y cada nodo tiene asociado a su vez la lista de sus nodos adyacentes.
En este ejercicio tendrás que implementar este TDA-Grafo utilizando la siguiente codificación:



- la lista de nodos se codificará mediante el tipo `list()` de Python, y
- la lista de nodos adyacentes se codificará mediante una lista enlazada.
- Además, ambas listas estarán ordenadas de forma **creciente** de acuerdo al identificador de cada nodo como se indica en la figura superior.

La representación de la figura inferior es incorrecta porque:

- la lista de nodos no respeta el orden creciente de los identificadores pues el 5 aparece antes que el 3, y
- la lista de adyacentes del nodo 1 no están ordenados de forma creciente pues el 4 aparece antes que el 2.

En nuestra representación trabajaremos con las siguientes estructuras

- Nodo cabecera son los que se almacenan en la lista de tipo `list()` de Python. Consta de los siguientes campos:
  - **id:** indica el identificador del nodo. Estos `id` son valores no-negativos.
    Resaltar que el **id** del nodo NO tiene por qué corresponderse con la posición que ocupe en la lista. P.e. para la siguientes lista de nodos identificados `lista=[N[id=20], N[id=50]]`, la posición del primer nodo es `lista.index(N[id=20])=0` pero su identificador es `20`. Del mismo modo el índice del segundo nodo es 1 pero su `id` es 50.
  - **valor:** contiene cualquier información del nodo del grafo. Por simplicidad se supondrá que es un simple string.
  - **right:** o es None, o apunta a un nodo de tipo `NodoAdyacente`.
- Nodo Adyacente son los elementos de la lista enlazada asociada a un nodo cabecera y representa a sus nodos adyacentes. Consta de los siguientes campos:
  - **id:** indica el identificador del nodo.
  - **right:** o es None, o apunta a un nodo de tipo `NodoElemento`..

La implementación se realizará en varias partes (módulos) y cada paso se desarrollará con el inicio de **un folio diferente**.
No está permitido ni el uso de **assert** ni de **list.sort()** en todo el programa
.

Parte/Folio 1: nodos.py (1.5 puntos) Especifica cómo se deben definir las **clases** `Nodo`, `NodoCabecera` y `NodoAdyacente`; así como sus constructores.
Recuerde redefinir estas clases, si fuera necesario, añadiendo cuantos métodos sean necesarios y en función de lo que se le pida en las partes siguientes.
NO escriba la interface Getter/Setter (se supone que ya estará implementada). Se asume, que para cada atributo de la forma `._atributo` que indiques en `.__init__()` existirá sus correspondientes propiedades `atributo`.

Parte/Folio 2: grafo.py (2 puntos) Implementa la clase `Grafo`, **usando solo las clases** `NodoCabecera` **y** `NodoElemento` y con los siguientes operadores:

- `Grafo(row: int, cols: int)`: Construye un grafo vacío (sin nodos). Permitirá trabajar con grafos que tengan un máximo de `row`-nodos y un máximo de `cols`-nodos adyacentes para cada nodo.
- `.add(self, id: int, valor: str = None, list_ady: list = None)` añade nodos a la estructura.
  - Si sólo se suministre el valor de `id`, se lanzará un error.
  - Si se suministra `id` y `valor` se construye un nodo cabecera y se añade a la estructura sin nodos adyacentes.
  - Si se suministra `id` y `list_ady` se añade `id` como nodo adyacente a cada uno de los nodos de la lista de identificadores `list_ady`. No es obligatorio que la lista de id's esté ordenada. Si la lista está vacía, `[]`, o si cualquiera de los identificadores no se corresponden con identificadores de nodos cabecera aún construidos se lanzará un error.
  - Si se suministran `id`, `valor` y `list_ady` se desarrollarán las dos acciones anteriores.
- Recuerda definir claramente los tipos de errores que generarás en esta parte en la parte 3.
- Añade aquí **otras clases** que te puedan ayudar para resolver el problema, si lo ves necesario.

Parte/Folio 3: errores.py (1.25 puntos) Especifica también las **clases** que usas para lanzar tus propios **errores**, y que veas necesarios en las demás partes.
Recuerde redefinir estas clases, si fuera necesario, añadiendo cuantos métodos y clases sean necesarias y en función de lo que se le pida en las demás partes.

Parte/Folio 4: main.py (1.25 punto) Construye un grafo **dirigido** a partir de la figura superior, controlando los errores.

Ten en cuenta lo siguiente:

- Implementarás explícitamente la inserción de un nodo en la lista enlazada, **sin recurrir a delegaciones de otros TDAs** distintos a los que aquí se indican.
- Justifique claramente las delegaciones que hace la clase `Grafo` a las clases de la jerarquía de nodos.
- Recuerda que las listas están ordenadas.

Ejercicio 2 (4 puntos + 1.5 punto extra)
Implementa las clases (atributos y métodos) que se piden en cada apartado, indicando de forma clara, y si fuera necesario, las clases abstractas, métodos abstractos, tipo de variables (de instancia o de clase), tipo de métodos (de instancia, de clase o estáticos).

1. **(0.5 puntos)** Todo guerrero tiene un nombre, pero no se pueden construir instancias de guerreros. Existen dos tipos de guerreros, los griegos y los troyanos, que sí pueden construirse. Construya las clases `Guerrero`, `Griego`, `Troyano`.

2. **(1 punto)** Escribe la clase `Caballo`. Consta de dos atributos de instancia. El primero hace referencia a su capacidad y el segundo hace referencia a una pila (stack) de ocupantes (que serán guerreros). Consta también del siguiente método:

   `.buscar(self, nombre: str)->bool` retorna `True` si se encontró a un guerrero con el nombre dado o `False` en otro caso.

   ⋆ Si necesitara los métodos de una pila, dispone de los métodos `.push()`, `.pop()`, ... (consulte las chuletas oficiales para conocer los métodos permitidos en un TDA de tipo Stack).

   ⋆ NO implemente/escriba ningún método del TDA Pila. Se asume que están disponibles e implementados.

3. **(1.5 puntos)** Existen dos tipos de caballos, `CaballoMamifero` y `CaballoTroya`, y que debes de construir:

   - Cualquier caballo mamífero tiene siempre una capacidad de hasta 2 jinetes y se construyen sin jinetes que lo monten.
   - Un caballo de Troya se construye suministrando su capacidad para n-jinetes junto con una lista de guerreros. Deberá de tener en cuenta que (a) la capacidad n no podrá superar una capacidad máxima fija establecida **para cualquier** caballo de Troya y (b) de la lista de guerreros dada solo pueden entrar en la pila los guerreros griegos de esa lista - se descarta cualquier troyano que venga dado en la lista de entrada.

4. **(1 punto)** Se te suministra la interface `IMontar` con los métodos abstractos `montar()` y `desmontar()` y cuyos parámetros y sus tipos debes deducir de lo que se te pide.

   - El método `desmontar` debe quitar al último ocupante que subió al caballo.
   - El método `montar` debe añadir un ocupante al caballo, siempre que no se supere su capacidad de ocupantes. Además deberá de tener en cuenta que en un caballo de Troya su parámetro de entrada será un griego (no se admiten troyanos) y en un caballo mamífero se admite cualquier guerrero (no importa si se mezclan troyanos con griegos).

   ¿Cómo y dónde deben implementarse estos métodos en la **jerarquía de clases de todos los caballos** teniendo en cuenta lo que se indica y sin que ninguno de los métodos use la sentencia **if** o **isinstance()**?

   En este apartado no copie todo lo que ya haya escrito sobre las clases de los apartados anteriores, **solo debe indicar** el nombre de la clase y la inclusión de lo que ahora realice.

5. **(0.5 puntos extras)** ¿Cuál debería ser la signatura - parámetros, retorno y sus tipos - sin que se viole el principio de sustitución de Liskov?

   En este apartado no copie todo lo que ya haya escrito sobre las clases de los apartados anteriores, **solo debe indicar** el nombre de la clase y la inclusión de lo que ahora realice.

6. **(1 punto extra)** Un `Ejercito` consta de un atributo que guarda una lista de guerreros (pueden ser griegos o troyanos). Todo ejército se **construye** invocando a su método `.load(fichero: str) -> List[Guerrero]` que consiste en leer el contenido de un fichero de texto. Cada línea del fichero tiene la secuencia `Griego nombreGuerreroGriego` o la secuencia `Troyano nombreGuerreroTroyano`, donde `nombreGuerreroXXX` es una secuencia de caracteres sin espacios. El método retorna una lista de guerreros, que puede ser vacía. Implementa dicho método controlando la excepción `FileNotFoundError`.

   ¿Cómo es el constructor de esta clase y su método `.load()`?