Modelo de Parcial 2

Ejercicio 1: Considere trabajar con el **TAD Árbol Binario de Búsqueda**.
a) Defina el objeto de datos.
b) Especifique e implemente la operación que que imprima todos los nodos se encuentran en un nivel n ingresado por teclado.
(Considere el Nivel de la raíz 1)

```
# a) Defina el objeto de datos.
class Nodo:
    __der:object
    __izq:object
    __elem: str

    def __init__(self,x):
        self.__der = None
        self.__izq = None
        self.__elem = x
```

```
class Arbol:
    __raiz:Nodo

    def __init__(self):
        self.__raiz = None
```

```
def grado(self):
    g = 0
    if self.__izq is not None:
        g += 1
    if self.__der is not None:
        g += 1
    return g
```

```
# b) Especifique e implemente la operación que imprima todos los nodos que se encuentran
# en un nivel n ingresado por teclado. (Considere el Nivel de la raiz 1)

# Nombre: Nivel
# Funcion: Calcula el nivel del nodo con clave X
# Salida: Reporta el nivel del nodo con clave X si X pertenece A; Error en caso contrario

def nivel(self, nodo, n):
    if nodo is not None:
        if n == 1:
            print(nodo.getDato(), end=" ")
        elif n > 1:
            self.nivel(nodo.getIzq(), n - 1)
            self.nivel(nodo.getDer(), n - 1)
```

Ejercicio 2:
Considere trabajar con el **TAD Hashing**, con la política de manejo de colisiones: **Encadenamiento**.

a. Defina el objeto de datos, considerando aproximadamente 1000 claves.
b. Implemente la función hash a usar.
c. Especifique e implemente la operación Buscar_clave() y muestre la cantidad de intentos en que la encontró.

```
# a) Defina el objeto de datos. Considerando 1000 claves
class Hash:
    __tabla:np.ndarray
    __tamaño:int

    def __init__(self, tamaño=1000):
        self.__tamaño = self.getPrimo(tamaño/0.7)
        self.__tabla = np.empty(self.__tamaño, dtype=object)
        for i in range(self.__tamaño):
            self.__tabla[i] = ListaEnlazada()
    # b. Implemente la función hash a usar.
    def hash(self,valor):
        return valor % self.__tamaño
# c. Operación Buscar_clave y mostrar la cantidad de intentos en que encontró la clave.
def buscar(self, valor:int):
    i = self.hash(valor)
    aux = self.__tabla[i].getCab()
    intentos = 1
    encontrado = False

    while aux != None and aux.getElem() != valor:
        aux = aux.getSig()
        intentos += 1

    if aux != None:
        encontrado = True
        print(f"el valor {valor} está en el índice {i}")
        print(f"Se encontró en {intentos} intento(s)")
    else:
        print(f"el valor {valor} no se encontró después de {intentos} intento(s)")

    return encontrado
```

Ejercicio 3: : Considere el TAD **Digrafo**, formado por N vértices.
a) Defina el objeto de datos.
b) Implemente la operación que genere la Matriz de Adyacencia.

```python
# a) Defina el objeto de datos.
class Digrafo:
    __vertices: int
    __matriz: np.ndarray

    def __init__(self,N):
        self.__matriz = np.zeros((N,N),dtype=int)
        self.__vertices = N

    # b) Matriz de Adyacencia
    def agregarAristas(self,u,v):
        if self.__matriz[u][v] == 0:
            self.__matriz[u][v] = 1

    def matrizAdyacencia(self):
        print(self.__matriz)
```