PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

AYUDANTÍA 2

Estructuras de datos

Ayudante: Felipe Fuentes Porras

TUPLAS

- Son inmutables.
- Si contiene estructuras de datos mutables, estas se pueden modificar dentro de ella.

```
# Para crear una tupla vacia se usa tuple() sin ingresar elementos.
a = tuple()

# Se puede declarar explícitamente los elementos de la tupla ingresando los elementos entre paréntesis.
b = (0, 1, 2)

# La tupla puede ser creda con objetos de distito tipo. En las tuplas el uso de parentesis no es obligatorio cuando son creadas.
c = 0, 'mensaje'

print(b[0], b[1])
print(c[0], c[1])
```

LISTAS

- Son mutables
- Tienen orden

```
# lista vacía y agregar elementos incrementalmente. En este caso agregamos tuplas.
lista = []
lista.append((2015, 3, 14))
lista.append((2015, 4, 18))
print(lista)

# Tambien es posible agregar los objetos explicitamente al definirla por primera vez
lista = [1, 'string', 20.5, (23, 45)]
print(lista)

# Extraemos un el elemento usando el indice respectivo
print(lista[1])
```

LISTAS

lista1.extend(lista2)

```
canciones = ['Addicted to pain', 'Ghost love score', 'As I am']
print(canciones)

nuevas_canciones = ['Elevate', 'Shine', 'Cry of Achilles']
canciones.extend(nuevas_canciones)
print(canciones)
```

Lista.insert(posición, valor)

```
print(canciones)
canciones.insert(1, 'Sober')
print(canciones)
```

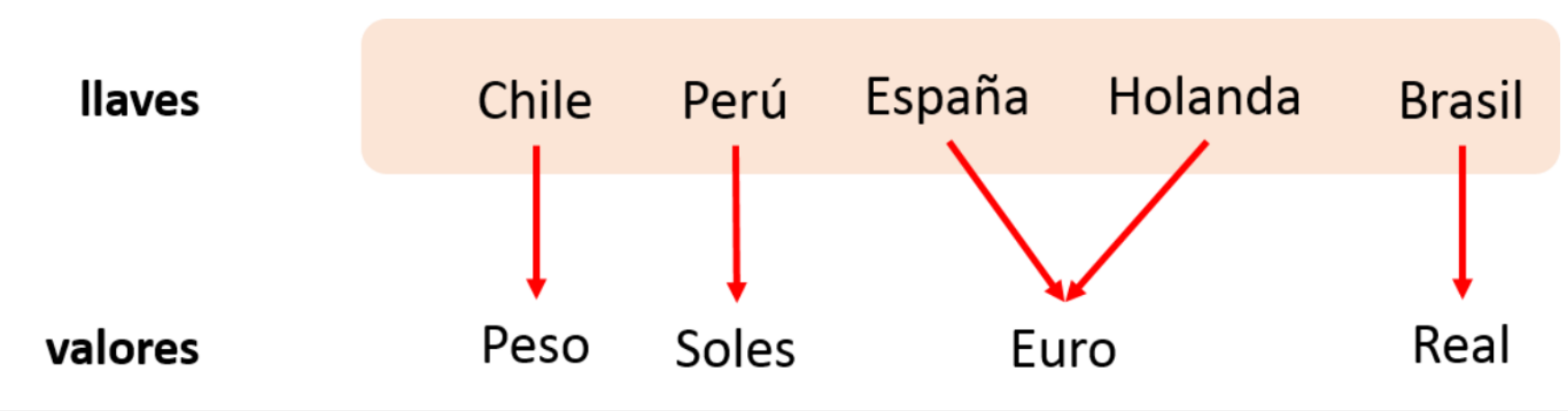
LISTAS

Slicing:

- `a[start : end]` : retorna los elementos desde `start` hasta `end-1` .
- `a[start:]` : retorna los elementos desde `start` hasta el final del arreglo.
- `a[:end]` : retorna los elementos desde el principio hasta `end-1` .
- `a[:]` : crea una copia (shallow) del arreglo completo. Es decir, el arreglo retornado está en una nueva dirección de memoria, pero los elementos en el arreglo están hace referencia a la dirección de memoria a los elemenos del arreglo original.
- `a[start : end : step]` : retorna los elementos desde `start` hasta no pasar `end` , en pasos de a `step` .
- `a[-1]` : retorna el último elemento en el arreglo.
- `a[-n:]` : # últimos `n` elementos en el arreglo.
- `a[:-n]` : retorna todos los elementos del arreglo menos los últimos `n` elementos.

DICCIONARIOS

- Mutables
- Se construyen a partir de par key-value.
- Las keys solo pueden ser datos inmutables



DICCIONARIO

Crear un dict():

```
vocales = dict() # Crea un diccionario
```

```
perros = {'bc': 'border-collie', 'lr': 'labrador retriever', 'pg': 'pug'}
```

Modificar:

```
perros['te'] = 'terrier'
```

Eliminar una key-value:

```
del perros['te']
```

Buscar si existe una llave (retorna True o False):

```
print('pg' in perros)
```

DICCIONARIO

```
print('Las llaves en el diccionario son las siguientes:')

for m in monedas.keys():
    print('{0}'.format(m))

print()
for m in monedas: # por defecto recorremos las llaves
    print('{0}'.format(m))
```

```
print('Los valores en el diccionario:')
for v in monedas.values():
    print('{0}'.format(v))
```

```
print('Los pares en el diccionario:')
for k, v in monedas.items():
    print('la moneda de {0} es {1}'.format(k,v))
```

DICCIONARIO ESPECIAL -> DEFAULTDICTS

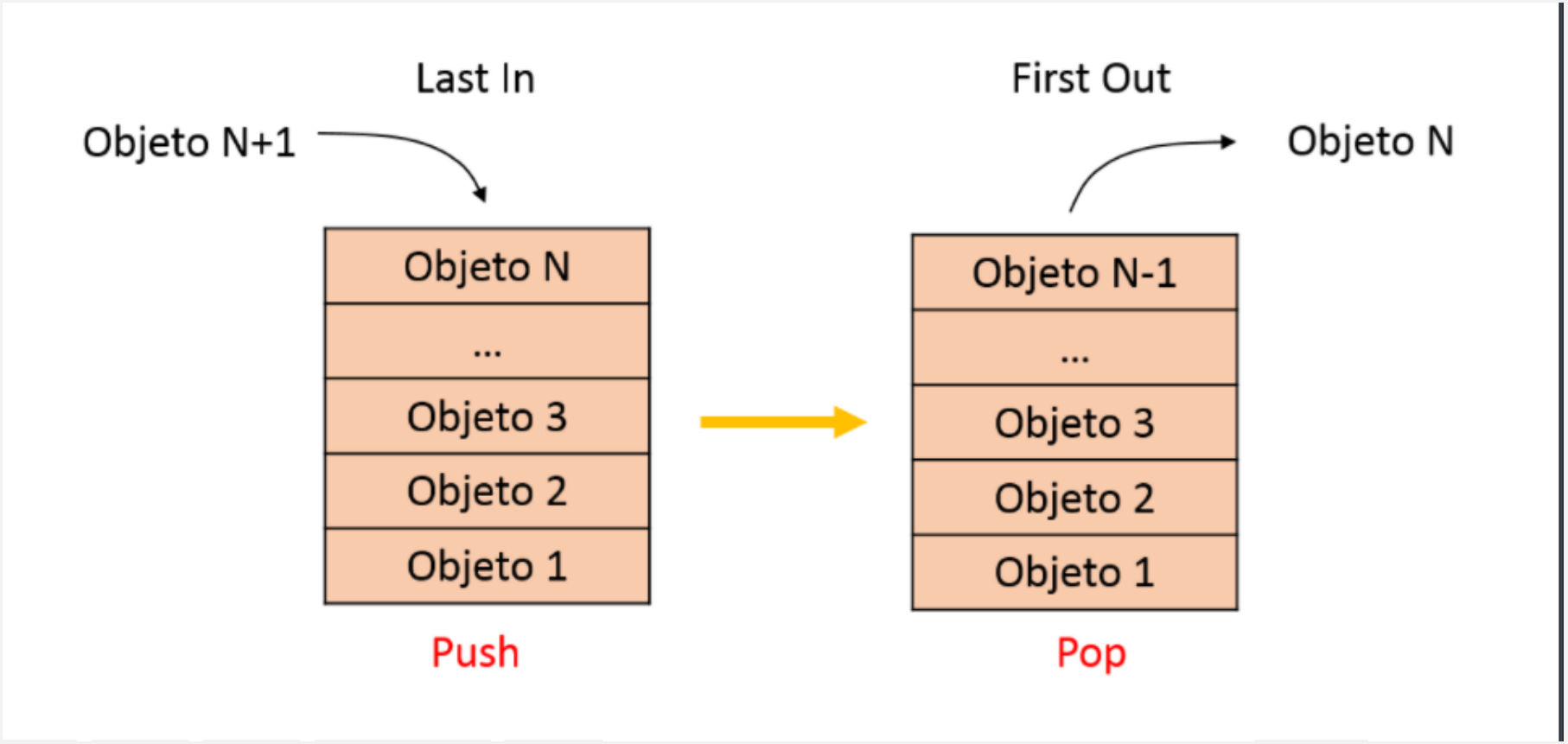
```
from collections import defaultdict

num_items = 0

def funcion_ej():
    global num_items
    num_items += 1
    return ([str(num_items)])

d = defaultdict(funcion_ej)
```

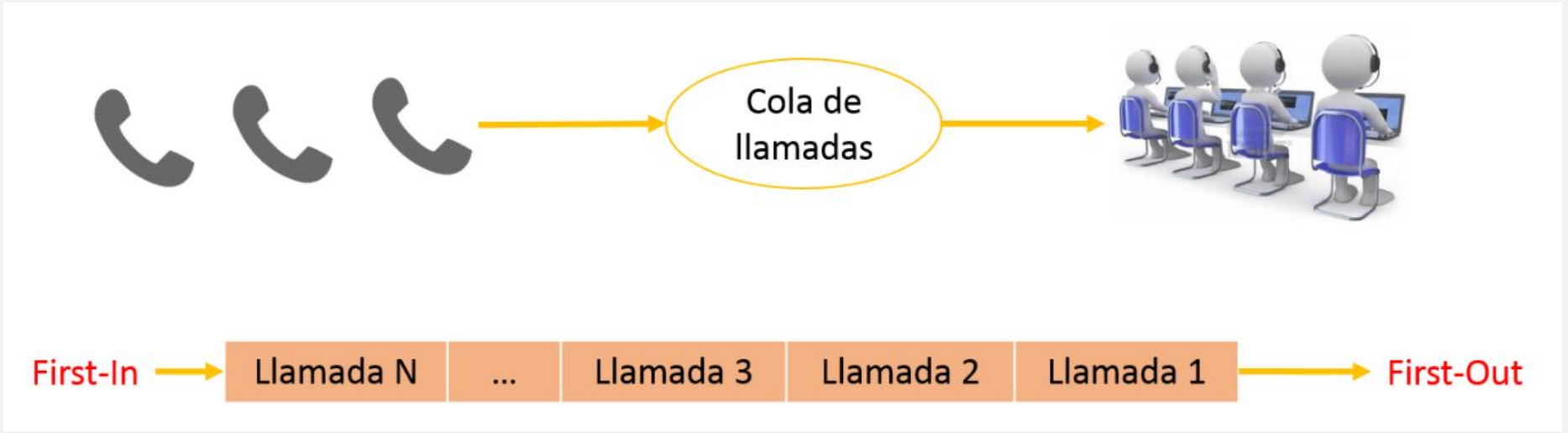
STACKS


Last In
Objeto N+1
Objeto N
...
Objeto 3
Objeto 2
Objeto 1
Push
First Out
Objeto N
Objeto N-1
...
Objeto 3
Objeto 2
Objeto 1
Pop

STACKS EN PYTHON

Métodos básicos en el stack	Implementación en Python	Descripción
Stack.push(elemento)	Lista.append(elemento)	Agrega secuencialmente elementos al stack
Stack.pop()	Lista.pop()	Retorna y extrae el último elemento agregado al stack
Stack.top()	Lista[-1]	Retorna el último elemento agregado sin extraerlo
len(Stack)	len(Lista)	Retorna la cantidad de elementos en el stack
Stack.is_empty()	len(Lista) == 0	Verifica si el stack está vacío

COLAS


Cola de
llamadas
First-In
Llamada N
...
Llamada 3
Llamada 2
Llamada 1
First-Out

DEQUE

Implementación en Python	Descripción
Deque.appendleft(item)	Agrega un objeto al comienzo del deque
Deque.append(item)	Agrega un objeto al final del deque
Deque.popleft()	Retorna y extrae el primer elemento del deque
Deque.pop()	Retorna y extrae el último elemento del deque
Deque[0]	Retorna el primer objeto del deque sin extraerlo
Deque[-1]	Retorna el último objeto del deque sin extraerlo
len(Deque)	Retorna el número de elementos del deque
len(Deque) == 0	Verifica si el deque está vació
Deque[j]	Acceso al ítem j del deque
Deque[j] = valor	Modificación del ítem j del deque
Deque.clear()	Borra todos los ítems
Deque.rotate(k)	Desplazamiento circular en k pasos
Deque.remove(e)	Remueve el primer ítem que sea igual a e
Deque.count(e)	Cuenta el número de ítems iguales a e

SETS

- No permite duplicado
- Los datos no presentan un orden

Crear un set():

```
artistas = set()
```

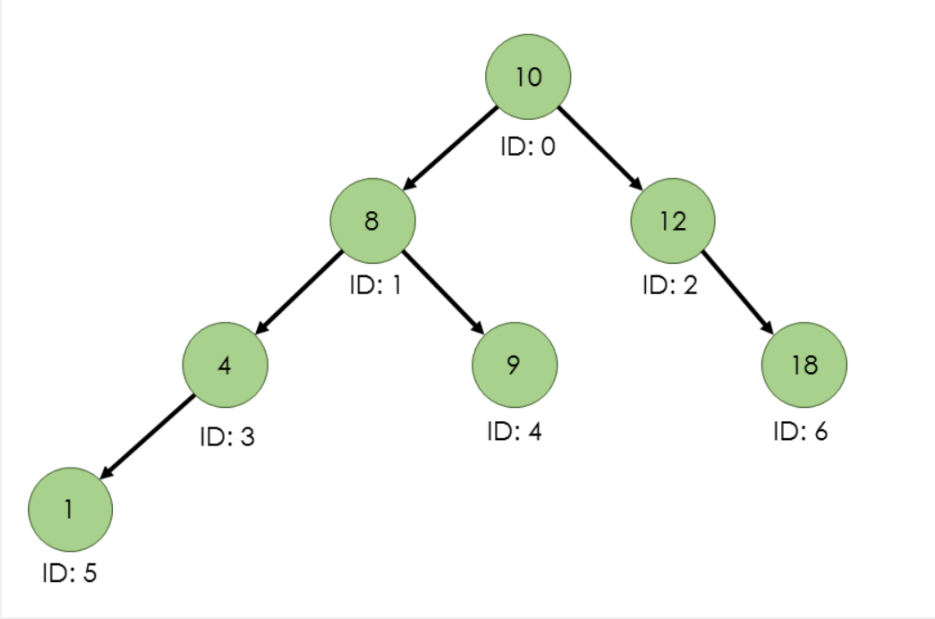
```
canciones = {'Style',
             'Uptown Funk',
             'Take Me To Church',
             'Sugar',
             'Thinking Out Loud',
             'Patterns In The Ivy',
             'Love Me Like You Do'}
```

Agregar a un set():

```
for cancion, artista in lista_canciones:
    artistas.add(artista)
```

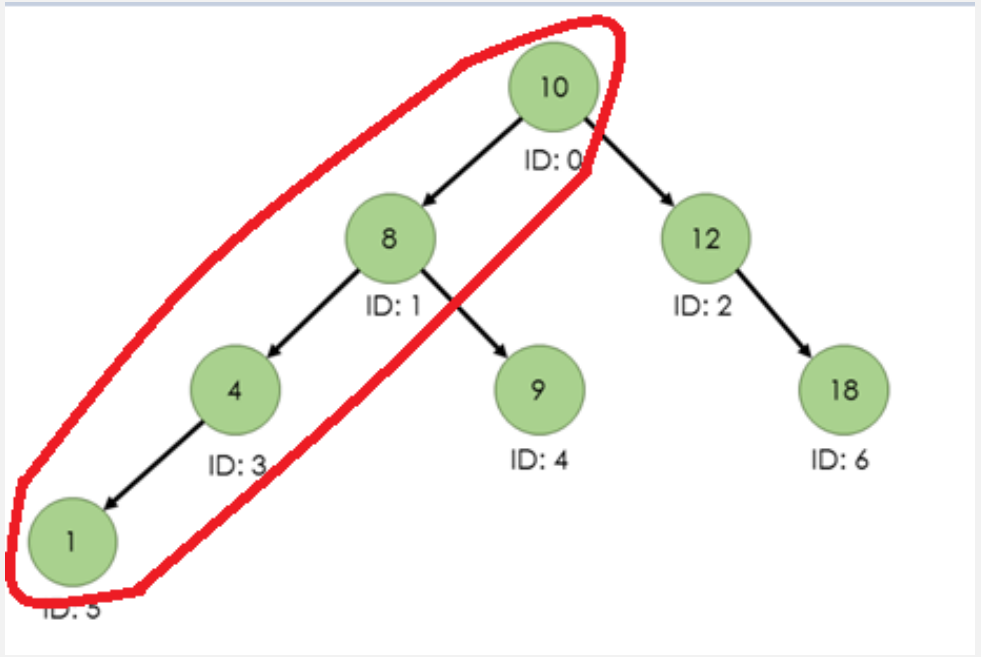
ARBOLES

- Estructura de datos con orden jerárquico
- Su unidad base es el nodo



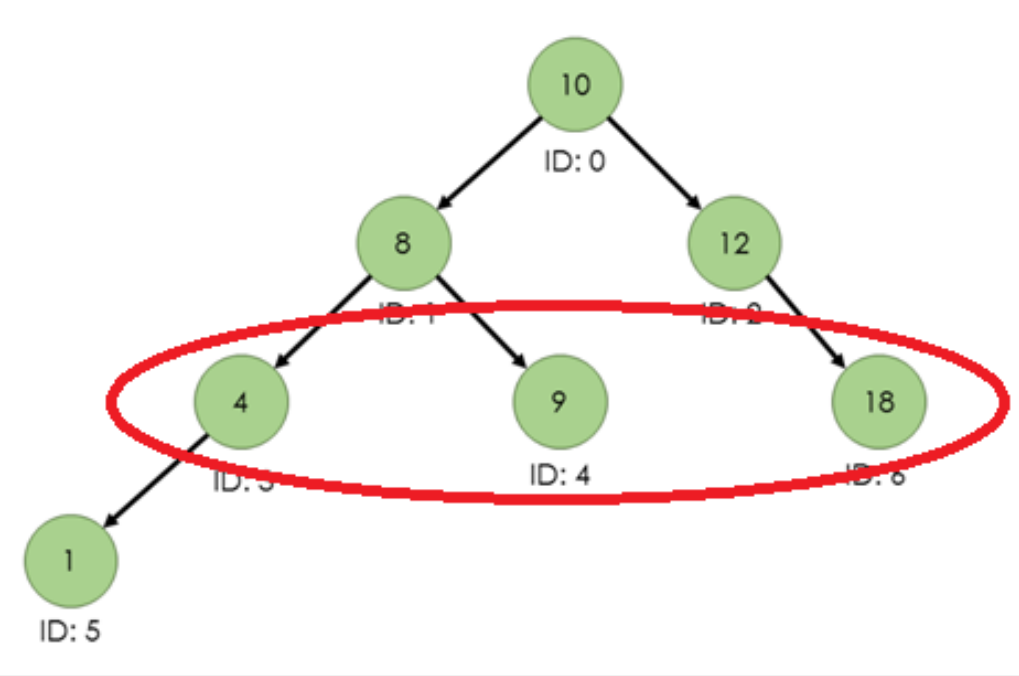
DFS

- Recorre el árbol por camino
- Usa stacks



BFS

- Recorre el árbol por nivel según jerarquía



EJERCICIO TALLER

Por algún extraño motivo, ud. se encuentra atrapado en una mansión embrujada y lógicamente debe encontrar la salida. Lamentablemente, la única manera de escapar es viajar a través de las habitaciones de la mansión, buscando alguna que tenga una puerta de salida. Lamentablemente, la mansión tiene una cantidad de habitaciones que en la práctica es infinita y además, cada una de ellas tiene entre 1 y 5 puertas (contando la de entrada), que llevan a otras habitaciones de la mansión.

En este escenario desolador, ud. afortunadamente descubre un libro en el suelo, donde se indica que cada habitación solo puede alcanzarse a través de una secuencia única de habitaciones. Además, este libro contiene un mapa parcial, que indica para solo algunas habitaciones, qué puerta se debe elegir.

CONSIDERACIONES DEL EJERCICIO

- La mansión con sus habitaciones debe ser construida de manera automática, asignando aleatoriamente tanto la cantidad de puertas en cada habitación, como la ubicación de las salidas.
- Cada habitación debe tener un identificador único, que no debe guardar relación con el lugar de la habitación en la mansión, ni con las habitaciones que la anteceden o siguen.
- La cantidad de habitaciones debe ser exponencialmente mayor que la de salidas.
- El mapa parcial solo podrá contener información para una cantidad de habitaciones que no debe exceder el logaritmo de la cantidad total de habitaciones.
- Dado todo lo anterior, su objetivo es construir un programa que encuentre una salida de la mansión lo antes posible. No es necesario que esta salida sea la más cercana a la entrada.

EJERCICIO 2A

Utilizando OOP y estructuras de datos básicas, modele un verificador de caminos en grafos no dirigidos de dos maneras distintas: i) utilizando listas y ii) utilizando diccionarios. Los nodos serán identificados con *strings* de largo 1, compuestos por letras mayúsculas. Para realizar la modelación, considere los siguientes instrucciones:

- Cree una clase para cada tipo de verificador (`Verificador_Lista` y `Verificador_Diccionario`). Las clases deben implementar al menos 2 métodos: `__init__(arcos)` y `existe_camino(camino)`. Puede implementar más métodos si lo prefiere o requiere.

EJERCICIO 2A

- Los arcos de los grafos serán entregados utilizando una lista de tuplas (una tupla por arco), donde cada tupla contendrá los 2 nodos que une el arco. Puede asumir que los arcos estarán bien formados y sin errores, pero no que siguen un orden específico.
- Los nodos de los caminos serán entregados como una lista de *strings*, donde cada *string* indica un nodo del camino. El orden de la lista indica el orden del camino. Puede asumir que cada uno de los nodos en el camino existe en el grafo, pero no puede asumir que el camino existe en el grafo o que el camino no tiene nodos o arcos repetidos.
- Para cada una de las clases, almacene los arcos en la estructura de datos correspondiente, es decir, lista o diccionario. Recuerde que el grafo es **no dirigido**.
- Para verificar la existencia de los caminos, itere por cada arco considerado en este y compruebe si está almacenado en la estructura de datos utilizada. El método debe retornar (no imprimir en pantalla) una tupla, indicando si el camino existe (`True` o `False`) y cuántas verificaciones de arcos debió realizar en la estructura almacenadora. Por ejemplo, si un camino no existe y el método iteró sobre 45 arcos en total para realizar la verificación, el retorno debe ser `(False, 45)`. **IMPORTANTE:** Para la clase basada en una lista, no está permitido utilizar métodos que realicen una búsqueda automática en esta (por ejemplo `index` o `in`), para verificar si existe un arco.

EJERCICIO 2A EJEMPLO EJECUCIÓN

Código

```
arcos = [("A","B"), ("A","E"),("A","C"), ("B","D"),("B","E"),
         ("C","F"),("C","G"), ("D","E"), ("E","F")]
camino = ["A","C","F","E","A","B"]
verificador = Verificador_Lista(arcos)
res = verificador.existe_camino(camino)
print(res)
verificador = Verificador_Diccionario(arcos)
res = verificador.existe_camino(camino)
print(res)
```

Salida

```
(True, 21)
(True, 10)
```

EJERCICIO 2B

Considere la siguiente situación: ud. recibe una clave secreta en forma de número natural, que debe ser utilizada para desactivar una bomba. Lamentablemente, por problemas en el canal de comunicación, la clave recibida tiene algunos errores. Después de realizar análisis estadísticos, se ha identificado que el valor recibido tiene dígitos duplicados. Escriba un programa que los elimine, manteniendo sólo la primera aparición de cada dígito y entregue como respuesta el número resultante. Un ejemplo de ejecución del algoritmo es el siguiente:

Código

```
def eliminar_duplicados(n):
    #solucion al problema


n = 1242241
n_ = eliminar_duplicados(n)
print(n_)
```

Salida

```
124
```