ESTRUCTURA DE DATOS FUNDAMENTALES Y ALGORITMOS
GUÍA DE LABORATORIO 08-09
GESTIONAR ASERCIONES – LISTAS ENLAZADAS SIMPLES, Y CIRCULARES

Asignatura	Datos del alumno	Fecha y Firma
Algoritmos y solución de problemas	Apellidos:	
	Nombre:	

Instrucciones:

Desarrollar las actividades que indica el docente en base a la guía de trabajo que se presenta.

1. Objetivos:

- Escribir algoritmos y codificar haciendo uso de aserciones.
- Escribir algoritmos y codificar haciendo uso de listas enlazadas simples y circulares.

2. Equipos, herramientas o materiales

- Computador, Software: Python, Algoritmos.

3. Fundamento teórico

3.1. Conceptos Clave

3.2. Aserciones en Python

Las aserciones en Python son una herramienta poderosa para verificar suposiciones sobre el comportamiento de su código. Son especialmente útiles para:

- **Depuración:**
  - Detectar errores en tiempo de ejecución.
  - Identificar la causa de un error más fácilmente.
  - Obtener información adicional sobre el estado del programa cuando se produce un error.
- **Documentación:**
  - Expresar las expectativas sobre el comportamiento del código.
  - Ayudar a otros programadores a comprender cómo funciona el código.
- **Pruebas:**
  - Escribir pruebas unitarias para verificar el comportamiento del código.
  - Asegurar que el código se comporte de la manera esperada.
- **Sintaxis:**
  - La sintaxis básica de una aserción es:

```
assert <expresión booleana>, <mensaje de error>
```

<expresión booleana>: La condición que se desea verificar.

<mensaje de error>: Un mensaje que se muestra si la condición es falsa.

- **Comportamiento:**
  - Si la expresión booleana es verdadera, la aserción se cumple y el programa continúa ejecutándose.
  - Si la expresión booleana es falsa, se genera una excepción AssertionError con el mensaje de error especificado.

Ejemplo 01:

```python
""" Calcula el factorial de un número."""

def factorial(numero):
  assert numero >= 0, "El número debe ser positivo"

  if numero == 0:
    return 1

  resultado = 1
  for i in range(1, numero + 1):
    resultado *= i

  return resultado

numero = -1
try:
  factorial(numero)
except AssertionError as error:
  print(error)

```

3.2.1. Gestionar aserciones en Python: Asignación, Secuencias, Condicional y Ciclos

Asignación:

- Asegurar el tipo de dato:

```python
edad = int(input("Ingrese su edad: "))
assert isinstance(edad, int), "La edad debe ser un número entero"
```

- Verificar el valor dentro de un rango

```python
calificacion = float(input("Ingrese su calificación: "))
assert 0 <= calificacion <= 100, "La calificación debe estar entre 0 y 100"
```

Secuencias

- Comprobar la longitud de una lista

```python
lista_nombres = ["Juan", "Ana", "Pedro"]
assert len(lista_nombres) == 3, "La lista debe tener 3 nombres"
```

- Validar un elemento dentro de una tupla

```python
coordenadas = (10.45, -67.53)
assert coordenadas[0] > 0, "La latitud debe ser positiva"
```

Condicional

- Verificar el resultado de una comparación

```python
numero1 = 5
numero2 = 10
assert numero1 < numero2, "El primer número debe ser menor que el segundo"
```

- Comprobar la igualdad de dos objetos

```python
objeto1 = {"nombre": "Juan", "edad": 30}
objeto2 = {"nombre": "Juan", "edad": 30}
assert objeto1 == objeto2, "Los dos objetos deben ser iguales"
```

Ciclos

- Asegurar que un ciclo while se ejecuta al menos una vez

```python
contador = 0
while contador < 10:
  contador += 1
assert contador == 10, "El ciclo while no se ejecutó al menos una vez"

```

- Validar el valor final de una variable en un ciclo for

```python
suma = 0
for numero in range(1, 11):
  suma += numero
assert suma == 55, "La suma de los números del 1 al 10 no es 55"

```

3.3. Listas Enlazadas Simples

Una lista enlazada simple es una estructura de datos que almacena una colección de elementos de forma ordenada. A diferencia de los arrays, las listas enlazadas no son contiguas en la memoria. En cambio, cada elemento, también conocido como nodo, se compone de dos partes:

Dato: El valor que se almacena en el nodo. Puede ser de cualquier tipo de dato, como un número, una cadena o un objeto.
Puntero: Un enlace al siguiente nodo de la lista. El último nodo de la lista tiene un puntero nulo para indicar el final de la lista.

- **Ventajas:**
  - Flexibilidad: Las listas enlazadas son dinámicas, lo que significa que su tamaño puede aumentar o disminuir sin necesidad de reservar memoria adicional por adelantado. Esto las hace ideales para situaciones en las que el número de elementos no se conoce de antemano.
  - Eficiencia en la inserción y eliminación: Insertar o eliminar un nodo en una lista enlazada simple es una operación relativamente sencilla, ya que no es necesario mover otros nodos en la memoria.
- **Desventajas:**
  - Acceso aleatorio lento: Acceder a un elemento específico en una lista enlazada simple requiere recorrer la lista desde el principio hasta encontrar el elemento deseado. Esto puede ser lento si la lista es grande.
  - Mayor uso de memoria: Las listas enlazadas requieren más memoria que los arrays, ya que cada nodo almacena un puntero al siguiente nodo.

- **Implementación de un nodo en Python:**

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None
```

- **Definir la clase ListaEnlazadaSimple:**

```python
class ListaEnlazadaSimple:
    def __init__(self):
        self.inicio = None

```

Operaciones básicas en listas enlazadas simples

Insertar un nodo al final de la lista:

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None

class ListaEnlazadaSimple:
    def __init__(self):
        self.inicio = None

    def insertar_al_principio(self, dato):
        nuevo_nodo = Nodo(dato)
        nuevo_nodo.siguiente = self.inicio
        self.inicio = nuevo_nodo

# Ejemplo de uso:
lista = ListaEnlazadaSimple()

# Insertar al principio
lista.insertar_al_principio(3)
lista.insertar_al_principio(2)
lista.insertar_al_principio(1)

# Mostrar la lista
actual = lista.inicio
while actual:
    print(actual.dato, end=" -> ")
    actual = actual.siguiente
```

Insertar un nodo al final de la lista:

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None


class ListaEnlazadaSimple:
    def __init__(self):
        self.inicio = None

    def insertar_al_final(self, dato):
        nuevo_nodo = Nodo(dato)
        if not self.inicio:
            self.inicio = nuevo_nodo
            return

        actual = self.inicio
        while actual.siguiente:
            actual = actual.siguiente

        actual.siguiente = nuevo_nodo

# Ejemplo de uso:
lista = ListaEnlazadaSimple()

# Insertar al final
lista.insertar_al_final(4)
lista.insertar_al_final(5)
lista.insertar_al_final(1)

# Mostrar la lista
actual = lista.inicio
while actual:
    print(actual.dato, end=" -> ")
    actual = actual.siguiente
```

- **Eliminar un nodo de la lista:**

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None


class ListaEnlazadaSimple:
    def __init__(self):
        self.inicio = None

    def insertar_al_principio(self, dato):
        nuevo_nodo = Nodo(dato)
        nuevo_nodo.siguiente = self.inicio
        self.inicio = nuevo_nodo

    def eliminar_nodo(self, dato):
        if not self.inicio:
            print("La lista está vacía.")
            return

        if self.inicio.dato == dato:
            self.inicio = self.inicio.siguiente
            return

        actual = self.inicio
        while actual.siguiente and actual.siguiente.dato != dato:
            actual = actual.siguiente

        if not actual.siguiente:
            print(f"No se encontró el nodo con el dato {dato}.")
            return

        actual.siguiente = actual.siguiente.siguiente


# Ejemplo de uso:
lista = ListaEnlazadaSimple()

# Insertar al principio
lista.insertar_al_principio(3)
lista.insertar_al_principio(2)
lista.insertar_al_principio(1)

# Eliminar un nodo
lista.eliminar_nodo(2)

# Mostrar la lista
actual = lista.inicio
while actual:
    print(actual.dato, end=" -> ")
    actual = actual.siguiente
```

3.4. Listas Enlazadas Circulares

Una lista enlazada circular es una estructura de datos que se compone de nodos, donde cada nodo contiene un valor y un puntero al siguiente nodo. El último nodo de la lista apunta al primer nodo, formando un ciclo.

- **Implementación de un nodo en Python:**

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None
```

- **Definiendo la clase ListaEnlazadaCircular:**

```python
class ListaEnlazadaCircular:
    def __init__(self):
        self.primero = None
```

- **Operaciones básicas en listas enlazadas circulares**
  - **Insertar un nodo al final de la lista y mostrar:**

```python
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.siguiente = None

class ListaEnlazadaCircular:
    def __init__(self):
        self.primero = None

    def esta_vacia(self):
        return self.primero is None

    def insertar_al_final(self, valor):
        nuevo_nodo = Nodo(valor)
        if self.esta_vacia():
            nuevo_nodo.siguiente = nuevo_nodo  # El nuevo nodo enlaza consigo mismo
            self.primero = nuevo_nodo
        else:
            nuevo_nodo.siguiente = self.primero
            nodo_actual = self.primero
            while nodo_actual.siguiente != self.primero:
                nodo_actual = nodo_actual.siguiente
            nodo_actual.siguiente = nuevo_nodo

    def mostrar_lista(self):
        if self.esta_vacia():
            print("La lista está vacía.")
        else:
            nodo_actual = self.primero
            while True:
                print(nodo_actual.valor, end=" -> ")
                nodo_actual = nodo_actual.siguiente
                if nodo_actual == self.primero:
                    break
            print()

# Ejemplo de uso
lista_circular = ListaEnlazadaCircular()
lista_circular.insertar_al_final(1)
lista_circular.insertar_al_final(2)
lista_circular.insertar_al_final(3)

print("Lista enlazada circular:")
lista_circular.mostrar_lista()
```

4. Desarrollo y Actividades

- **Ejercicio parte 01:**
  1. Validar la edad de un usuario.
  2. Verificar el tipo de dato de una variable.
  3. Validar el rango de una calificación.
  4. Asegurar que una lista no esté vacía.
  5. Validar la igualdad de dos objetos.
  6. Asegurar que un ciclo while se ejecuta al menos una vez.
  7. Asegurar que una función retorna un valor específico.
  8. Validar el estado de una variable después de una operación.
  9. Asegurar que un módulo se ha importado correctamente.

- **Ejercicios parte 02:**
  10. Desarrollar el código de **buscar nodo** en la lista enlazada simple.

  Suma de Nodos

  11. Implementa una función que sume todos los nodos de una lista enlazada simple.

  Longitud de la Lista

  12. Crea una función que devuelva la longitud de una lista enlazada simple.

  Concatenar Listas

  13. Implementa una función que concatene dos listas enlazadas simples.

  Eliminar Duplicados

  14. Crea una función que elimine los nodos duplicados de una lista enlazada simple.

  Invertir Lista

  15. Implementa una función que invierta el orden de una lista enlazada simple.