



Algoritmos y Estructuras de Datos

Documentacion

Estudiantes:

- Diego Armando Urbina Aviles
- Julio Cesar Delgadillo Pineda
- Emmanuel Leonardo Aguilar Novoa

Docente:

Silvia Gigdalia Ticay Lopez

Ejercicio 1:

En esta parte del programa se crea una pila utilizando programación orientada a objetos en Python. La clase Stack define los métodos necesarios para trabajar con esta estructura: push para agregar elementos, pop para sacarlos, peek para ver el último sin eliminarlo, y is_empty para saber si la pila está vacía. Internamente usa una lista y sigue la lógica LIFO, es decir, el último que entra es el primero que sale.

```
1  """
2  Módulo con la lógica para invertir el orden de las palabras de una frase usando POO y pilas.
3  Contiene:
4  - Stack: estructura de datos LIFO genérica.
5  - WordInverter: clase que usa Stack para invertir frases.
6  """
7
8  class Stack:
9      """Implementación sencilla de una pila."""
10     def __init__(self):
11         self._items = []
12
13     def is_empty(self):
14         return not self._items
15
16     def push(self, item):
17         """Añade un elemento al tope de la pila."""
18         self._items.append(item)
19
20     def pop(self):
21         """Extrae y devuelve el tope de la pila. Error si está vacía."""
22         if self.is_empty():
23             raise IndexError("pop from empty stack")
24         return self._items.pop()
25
26     def peek(self):
27         """Devuelve el elemento tope sin extraerlo, o None si está vacía."""
28         return self._items[-1] if self._items else None
29
```

A continuación, se presenta la clase WordInverter, que hace uso de la pila previamente definida para invertir el orden de las palabras en una frase. En su constructor se permite pasar la clase Stack como parámetro, lo cual facilita la reutilización y pruebas del código. El método invert() se encarga de realizar todo el proceso: primero elimina espacios innecesarios y separa la frase en palabras individuales, luego las apila una a una. Posteriormente, extrae cada palabra desde la pila (en orden inverso al original) y las almacena en una nueva lista. Finalmente, junta las palabras con espacios y devuelve la frase invertida.

```
29
30
31 class WordInverter:
32     """Invierte el orden de las palabras de una frase usando una pila."""
33     def __init__(self, stack_class=Stack):
34         # Permite inyección de dependencia para pruebas
35         self.stack_class = stack_class
36
37     def invert(self, sentence: str) -> str:
38         """Devuelve la frase con el orden de palabras invertido."""
39         stack = self.stack_class()
40         # Separar por espacios, conservar palabras no vacías
41         words = [w for w in sentence.strip().split(' ') if w]
42         for word in words:
43             stack.push(word)
44
45         inverted = []
46         while not stack.is_empty():
47             inverted.append(stack.pop())
48
49         return ' '.join(inverted)
```

Main:

Por último, se incluye una función `main()` que permite interactuar con el programa desde la consola. Al ejecutarse, crea una instancia del inversor de palabras y solicita al usuario que ingrese frases. Si el usuario escribe 'salir', el programa finaliza. En caso contrario, la frase ingresada se procesa utilizando el método `invert()` de la clase `WordInverter`, y se muestra el resultado invertido. También se maneja cualquier posible error mediante un bloque `try-except`, asegurando que el programa sea robusto ante entradas inesperadas.

```
1  from funciones_inversion import WordInverter
2
3
4  def main():
5      inverter = WordInverter() # creamos el invertidor
6      print("Invertidor de palabras usando pilas")
7      print("Escribe 'salir' para terminar.")
8      while True:
9          frase = input("Ingresa una frase: ") # pide la frase
10         if frase.lower() == 'salir':
11             print("¡Hasta luego!")
12             break
13         try:
14             resultado = inverter.invert(frase) # invierte y muestra
15             print(f"Invertido: {resultado}\n")
16         except Exception as e:
17             print(f"Error: {e}\n")
18
19  if __name__ == '__main__':
20      main()
```

Ejercicio 2:

Este ejercicio implementa la clase Pila, que representa una estructura tipo stack siguiendo el principio LIFO (Last In, First Out), donde el último elemento en entrar es el primero en salir.

La clase incluye los métodos básicos: push para agregar elementos al tope, pop para eliminarlos y devolverlos, y peek para consultar el elemento del tope sin retirarlo. Además, se valida si la pila está vacía antes de realizar operaciones, retornando None en caso de que no haya elementos.

```
1  class Pila:
2      """
3      Clase que implementa una estructura tipo pila (stack).
4      Sigue el principio LIFO: el ultimo en entrar es el primero en salir.
5      """
6
7      def __init__(self):
8          self.elementos = [] # Lista que almacena los datos
9
10     def push(self, valor):
11         """
12         Agrega un elemento al tope de la pila.
13         """
14         self.elementos.append(valor)
15
16     def pop(self):
17         """
18         Elimina y retorna el elemento del tope de la pila.
19         Si la pila esta vacia, retorna None.
20         """
21         if not self.is_empty():
22             return self.elementos.pop()
23         return None
24
25     def peek(self):
26         """
27         Retorna el elemento del tope sin eliminarlo.
28         Si la pila esta vacia, retorna None.
29         """
30         if not self.is_empty():
31             return self.elementos[-1]
32         return None
```

La clase Pila se completa con el método `is_empty`, que permite saber si la pila está vacía, retornando `True` cuando no contiene elementos. Esta verificación es clave para evitar errores al intentar retirar o consultar elementos inexistentes. Gracias a estos métodos, se puede utilizar esta pila como base para resolver el problema de verificar si una expresión tiene los paréntesis correctamente balanceados, simulando el proceso manual de abrir y cerrar símbolos conforme se recorren.

```
25  def peek(self):
26      """
27      Retorna el elemento del tope sin eliminarlo.
28      Si la pila esta vacia, retorna None.
29      """
30      if not self.is_empty():
31          return self.elementos[-1]
32      return None
33
34  def is_empty(self):
35      """
36      Retorna True si la pila esta vacia, False en caso contrario.
37      """
38      return len(self.elementos) == 0
```

Main:

Con la clase Pila ya definida, se implementa la función `esta_balanceado()`, encargada de verificar si los paréntesis en una cadena están correctamente emparejados. Para ello, se utiliza una pila para ir almacenando cada símbolo de apertura que se encuentra (`(`, `[`, `{`).

```
1  # Problema 2: Verificación de paréntesis balanceados
2  # Version: 1.0
3  # Fecha: 21/05/2025
4  # Autor: Diego Urbina, Julio Delgadillo, Emmanuel Aguilar
5
6  from Pilaejercicio2 import Pila # Importamos la estructura de pila
7
8  def esta_balanceado(cadena):
9      """
10     Verifica si los parentesis (), {}, [] en una cadena estan balanceados.
11     Utiliza una pila para controlar las aperturas y cierres.
12     """
13     pila = Pila()
14     pares = {'(': ')', '[': ']', '{': '}'} # Diccionario que relaciona cierres con aperturas
15
16     for caracter in cadena:
17         if caracter in "([{":
18             # Si es un parentesis de apertura, lo apilamos
19             pila.push(caracter)
20         elif caracter in ")]}":
21             # Si encontramos un cierre y la pila esta vacia, esta desbalanceado
22             if pila.is_empty():
23                 return False
24             # Si el ultimo abierto no corresponde con el cierre, esta mal emparejado
25             if pila.pop() != pares[caracter]:
26                 return False
27
28     # Si al final la pila esta vacia, todos los parentesis fueron cerrados correctamente
29     return pila.is_empty()
30
31
```

Finalmente, se define una función `main()` que actúa como punto de entrada para probar el verificador de paréntesis. El programa solicita al usuario una expresión y valida si está vacía. En caso contrario, llama a la función `esta_balanceado()` y muestra un mensaje indicando si los paréntesis están correctamente balanceados o no.

```
30
31
32 def main():
33     print("=== Verificador de parentesis balanceados ===")
34     expresion = input("Ingrese una expresion: ").strip()
35
36     if not expresion:
37         print("→ La expresion esta vacia.")
38     else:
39         if esta_balanceado(expresion):
40             print("→ Los parentesis estan balanceados.")
41         else:
42             print("→ Los parentesis NO estan balanceados.")
43
44
45 if __name__ == "__main__":
46     main()
```

Ejercicio 3:

En este ejercicio se implementa una lista de reproducción de música utilizando una estructura de lista doblemente enlazada. Primero se define la clase `Cancion`, que representa cada nodo de la lista. Cada canción tiene un nombre, así como referencias al nodo siguiente y al anterior, lo que permite recorrer la lista en ambas direcciones. Luego, la clase `ListaReproduccion` gestiona la lista completa. Su constructor inicializa dos punteros: uno a la primera canción (primera) y otro a la canción actualmente seleccionada (actual), lo que permitirá navegar entre canciones, agregar nuevas y eliminarlas dinámicamente.

```
1 class Cancion:
2     """
3     Clase que representa una cancion en la lista de reproduccion.
4     Tiene nombre y enlaces al siguiente y anterior nodo (lista doblemente enlazada).
5     """
6     def __init__(self, nombre):
7         self.nombre = nombre
8         self.siguiente = None
9         self.anterior = None
10
11     def __str__(self):
12         return self.nombre
13
14
15 class ListaReproduccion:
16     """
17     Clase que simula una lista de reproduccion de musica.
18     Permite recorrer hacia adelante o atras, agregar y eliminar canciones.
19     """
20     def __init__(self):
21         self.primera = None
22         self.actual = None
```

La clase `ListaReproduccion` incluye el método `agregar_cancion`, que permite añadir una nueva canción al final de la lista. Si la lista está vacía, la nueva canción se convierte en la primera y actual. De lo contrario, se recorre la lista hasta el último nodo y se enlaza la nueva canción al final, ajustando también la referencia al nodo anterior.

También se implementa el método `eliminar_cancion_actual`, que elimina la canción que está siendo reproducida en ese momento. Para ello, actualiza correctamente los enlaces de los nodos anterior y siguiente, y muestra un mensaje si no hay ninguna canción seleccionada.

```
24 def agregar_cancion(self, nombre):
25     """
26     Agrega una cancion al final de la lista.
27     """
28     nueva = Cancion(nombre)
29     if self.primer_a is None:
30         self.primer_a = nueva
31         self.actual = nueva
32     else:
33         temp = self.primer_a
34         while temp.siguiente:
35             temp = temp.siguiente
36         temp.siguiente = nueva
37         nueva.anterior = temp
38         print(f"-> Cancion '{nombre}' agregada a la lista.")
39
40 def eliminar_cancion_actual(self):
41     """
42     Elimina la cancion que esta siendo reproducida.
43     """
44     if self.actual is None:
45         print("-> No hay cancion para eliminar.")
46         return
47
48     nombre = self.actual.nombre
49     anterior = self.actual.anterior
50     siguiente = self.actual.siguiente
```

Una vez que se identifica la canción a eliminar, se reconectan los nodos anterior y siguiente para que la lista siga funcionando correctamente. Luego, se actualiza el puntero actual a la siguiente canción disponible; si no hay una siguiente, se pasa a la anterior.

```

51
52     # Reconectar nodos
53     if anterior:
54         anterior.siguiente = siguiente
55     else:
56         self.primer_a = siguiente
57
58     if siguiente:
59         siguiente.anterior = anterior
60
61     # Mover el puntero actual
62     self.actual = siguiente if siguiente else anterior
63     print(f"-> Cancion '{nombre}' eliminada.")
64
65     def siguiente_cancion(self):
66         """
67         Mueve el puntero a la siguiente cancion si existe.
68         """
69         if self.actual and self.actual.siguiente:
70             self.actual = self.actual.siguiente
71             print(f"-> Reproduciendo: {self.actual.nombre}")
72         else:
73             print("-> No hay siguiente cancion.")
74

```

También se implementa el método `siguiente_cancion`, que permite avanzar en la lista. Si existe una canción siguiente, el puntero se actualiza y se muestra el nombre de la nueva canción en reproducción. Si ya no hay más canciones, se indica con un mensaje. Esta funcionalidad hace que la lista de reproducción se comporte como un reproductor real, donde podemos avanzar y gestionar canciones fácilmente.

Además de avanzar, la lista de reproducción también permite retroceder mediante el método `anterior_cancion`, que mueve el puntero a la canción anterior si existe. Si no hay una canción previa, se muestra un mensaje informando al usuario.

```
75  def anterior_cancion(self):
76      """
77      Mueve el puntero a la cancion anterior si existe.
78      """
79      if self.actual and self.actual.anterior:
80          self.actual = self.actual.anterior
81          print(f"-> Reproduciendo: {self.actual.nombre}")
82      else:
83          print("-> No hay cancion anterior.")
84
85  def mostrar_lista(self):
86      """
87      Muestra toda la lista de canciones, marcando la que esta en reproduccion.
88      """
89      if self.primeras is None:
90          print("-> Lista de reproduccion vacia.")
91          return
92
93      print("Lista de reproduccion:")
94      temp = self.primeras
95      while temp:
96          if temp == self.actual:
97              print(f"-> {temp.nombre} [REPRODUCIENDO]")
98          else:
99              print(f"    {temp.nombre}")
100         temp = temp.siguiente
```

Finalmente, el método `mostrar_lista` imprime toda la lista de canciones desde el inicio, indicando cuál está siendo reproducida en ese momento. Si la lista está vacía, se notifica con un mensaje. Esta función permite visualizar el estado completo de la lista y es útil para saber qué canciones están cargadas y cuál está activa, simulando el comportamiento de un reproductor musical real.

Main:

La función `main()` actúa como menú interactivo para el usuario. Dentro de un ciclo `while`, permite ejecutar distintas acciones sobre la lista de reproducción, como agregar una nueva canción, eliminar la actual, avanzar, retroceder, mostrar la lista completa o salir del programa. Cada opción se activa según el número ingresado por el usuario, y se valida que la entrada sea correcta. De esta forma, se simula el funcionamiento básico de un reproductor musical, donde el usuario puede controlar fácilmente las canciones mediante un menú de texto claro y ordenado.

```

20 def main():
21     lista = ListaReproduccion()
22
23     while True:
24         mostrar_menu()
25         opcion = input("Selecciona una opcion (1-6): ").strip()
26
27         if opcion == "1":
28             nombre = input("Nombre de la cancion: ").strip()
29             if nombre:
30                 lista.agregar_cancion(nombre)
31             else:
32                 print("-> Nombre invalido.")
33         elif opcion == "2":
34             lista.eliminar_cancion_actual()
35         elif opcion == "3":
36             lista.siguiente_cancion()
37         elif opcion == "4":
38             lista.anterior_cancion()
39         elif opcion == "5":
40             lista.mostrar_lista()
41         elif opcion == "6":
42             print("-> Saliendo del reproductor.")
43             break
44         else:
45             print("-> Opcion invalida. Intenta de nuevo.")
46
47     print("-" * 40)
48
49 if __name__ == "__main__":
50     main()

```

Ejercicio 4:

En este ejercicio se implementa la clase `PriorityQueue`, que representa una cola de prioridad donde los elementos se atienden según su nivel de prioridad (siendo menor número = mayor prioridad). El método `enqueue` permite agregar elementos junto con su prioridad y se asegura de mantener el orden correcto usando `sort`. El método `dequeue` extrae y devuelve el elemento con mayor prioridad, es decir, el que tenga el valor numérico más bajo. Además, se incluye `is_empty` para verificar si la cola está vacía.

```

1  class PriorityQueue:
2      """Cola de prioridad con prioridades enteras (menor valor => mayor prioridad)."""
3      def __init__(self):
4          self._items = []
5
6      def is_empty(self) -> bool:
7          """Devuelve True si la cola está vacía."""
8          return not self._items
9
10     def enqueue(self, item: str, priority: int) -> None:
11         """
12         Agrega un elemento con su prioridad.
13         priority: entero, menor = mayor prioridad.
14         """
15         # Insertar y mantener orden por prioridad ascendente
16         self._items.append((priority, item))
17         self._items.sort(key=lambda x: x[0])
18
19     def dequeue(self) -> tuple:
20         """
21         Elimina y devuelve el elemento con mayor prioridad (menor valor numérico).
22         Lanza IndexError si está vacía.
23         Devuelve: (item, priority)
24         """
25         if self.is_empty():
26             raise IndexError("dequeue from empty priority queue")
27         pr, itm = self._items.pop(0)
28         return itm, pr

```

La clase se completa con el método `peek`, que permite consultar el próximo elemento con mayor prioridad sin retirarlo de la cola. Si la cola está vacía, devuelve `None`. También se implementa el método especial `__len__`, que retorna la cantidad de elementos presentes en la cola.

```
30 def peek(self) -> tuple:
31     """
32     Devuelve el próximo elemento sin extraerlo.
33     Si está vacía, devuelve None.
34     """
35     if self.is_empty():
36         return None
37     pr, itm = self._items[0]
38     return itm, pr
39
40 def __len__(self) -> int:
41     """Número de elementos en la cola."""
42     return len(self._items)
```

Main:

La función `main()` proporciona una interfaz interactiva para utilizar la cola de prioridad. Dentro de un menú en bucle, el usuario puede elegir entre varias opciones: encolar un nuevo elemento con su prioridad, desencolar el elemento más urgente, consultar el próximo elemento (`peek`), verificar si la cola está vacía o salir del programa. El ingreso de datos incluye validaciones para evitar errores, como asegurarse de que la prioridad sea un número entero.

```
10 # Interfaz de usuario para la cola de prioridad
11 def main():
12     pq = PriorityQueue() # crear cola
13     print("--- Cola de Prioridad Interactiva ---")
14     while True:
15         # Mostrar menú
16         print("\nOpciones:")
17         print("1. Encolar elemento")
18         print("2. Desencolar siguiente")
19         print("3. Ver siguiente (peek)")
20         print("4. Chequear si está vacía")
21         print("5. Salir")
22         opcion = input("Selecciona una opción: ").strip()
23
24         if opcion == '1':
25             # Pedir datos y encolar
26             item = input("Elemento (nombre): ").strip()
27             try:
28                 pr = int(input("Prioridad (menor = más urgente): ").strip())
29                 pq.enqueue(item, pr)
30                 print(f"Agregado '{item}' con prioridad {pr}")
31             except ValueError:
32                 print("Prioridad inválida. Debe ser un número entero.")
33
34         elif opcion == '2':
35             # Desencolar elemento más urgente
36             try:
37                 item, pr = pq.dequeue()
38                 print(f"Desencolado '{item}' con prioridad {pr}")
39             except IndexError as e:
40                 print(f"Error: {e}")
```

Las últimas opciones del menú permiten consultar el próximo elemento en cola sin retirarlo, mediante el método `peek`, así como verificar si la cola está vacía y mostrar su tamaño actual. También se incluye la opción para salir del programa de forma segura, y un mensaje de advertencia si se ingresa una opción no válida.

```
41
42     elif opcion == '3':
43         # Ver siguiente elemento
44         res = pq.peak()
45         if res:
46             print(f"Siguiente en cola: '{res[0]}' con prioridad {res[1]}")
47         else:
48             print("La cola está vacía.")
49
50     elif opcion == '4':
51         # Chequear estado de la cola
52         estado = "vacía" if pq.is_empty() else f"no está vacía (tamaño={len(pq)})"
53         print(f"La cola {estado}.")
54
55     elif opcion == '5':
56         # Salir del programa
57         print("¡Hasta luego!")
58         break
59
60     else:
61         # Opción no válida
62         print("Opción no reconocida. Intenta otra vez.")
63
64 if __name__ == '__main__':
65     main()
```

Ejercicio 5:

En este ejercicio se implementa una lista enlazada simple. La clase `Node` representa cada nodo de la lista, almacenando un dato y una referencia al siguiente nodo. Luego, la clase `LinkedList` administra la estructura general de la lista. En su constructor, la lista comienza vacía (`head = None`), y a través del método `append` se pueden agregar elementos al final. Si la lista está vacía, el nuevo nodo se asigna como cabeza; de lo contrario, se recorre hasta el último nodo y se enlaza con el nuevo.

```

1  class Node:
2      # Nodo que almacena un dato y apunta al siguiente nodo
3      def __init__(self, data):
4          self.data = data # valor guardado
5          self.next = None # referencia al siguiente nodo, None si no hay
6
7      def __repr__(self):
8          # Permite ver el contenido al imprimir el nodo
9          return f"Node({self.data})"
10
11 class LinkedList:
12     # Lista enlazada simple, permite append y búsqueda
13     def __init__(self):
14         self.head = None # inicio de la lista
15
16     def append(self, data):
17         # Crea un nuevo nodo al final con el dato dado
18         new_node = Node(data)
19         if not self.head:
20             # Si la lista está vacía, head apunta al nuevo nodo
21             self.head = new_node
22             return
23         current = self.head
24         # Recorrer hasta llegar al último nodo
25         while current.next:
26             current = current.next
27         # Enlazar el nuevo nodo al final
28         current.next = new_node

```

La clase `LinkedList` se complementa con el método `search`, que permite buscar un valor específico dentro de la lista. La función recorre nodo por nodo comparando el dato almacenado con el valor buscado. Si lo encuentra, retorna su posición (basada en cero); si no lo encuentra, retorna -1. También se incluye el método especial `__str__`, que genera una representación visual de la lista mostrando los valores unidos por flechas (`->`), lo cual facilita entender la estructura y el orden de los elementos.

```

30 def search(self, target):
31     # Busca 'target' y devuelve su posición (0-based)
32     current = self.head
33     index = 0
34     # Recorrer nodos hasta encontrar o llegar al final
35     while current:
36         if current.data == target:
37             return index # encontrado
38         current = current.next
39         index += 1
40     return -1 # no encontrado en toda la lista
41
42 def __str__(self):
43     # Devuelve los valores de la lista separados por flechas
44     values = []
45     current = self.head
46     while current:
47         values.append(str(current.data))
48         current = current.next
49     return ' -> '.join(values)

```

Main:

La función `main()` actúa como interfaz para construir y explorar la lista enlazada. Primero, solicita al usuario que ingrese elementos separados por espacios, los cuales se agregan a la lista usando el método `append`. Luego muestra la lista completa y permite al usuario buscar valores dentro de ella. Si el valor se encuentra, se indica su posición; de lo contrario, se muestra un mensaje diciendo que no está presente. El proceso continúa hasta que el usuario escribe "salir", lo que finaliza la búsqueda.

```
7  def main():
8      print("--- Búsqueda en Lista Enlazada ---")
9      # Leer elementos separados por espacio
10     datos = input("Ingresa elementos separados por espacios: ").strip().split()
11     ll = LinkedList()
12     for d in datos:
13         ll.append(d) # construir la lista
14
15     print(f"Lista creada: {ll}")
16     while True:
17         # Pedir valor a buscar
18         objetivo = input("Valor a buscar (o 'salir'): ")
19         if objetivo.lower() == 'salir':
20             print("Fin de la búsqueda. ¡Adiós!")
21             break
22         pos = ll.search(objetivo)
23         if pos >= 0:
24             print(f'{objetivo} encontrado en posición {pos}.')
25         else:
26             print(f'{objetivo} no está en la lista.')
27         print() # línea en blanco para legibilidad
28
29     if __name__ == '__main__':
30         main()
```