UNIVERSIDAD AMERICANA

Facultad de Ingeniería y Arquitectura

Algoritmos y Estructuras de Datos

Documentacion

Estudiantes:

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

Docente:

Silvia Gigdalia Ticay Lopez

Expresiones aritméticas:

En este ejercicio se convierte una expresión en notación infija a notación postfija utilizando una pila. Se define la clase Stack con los métodos básicos para apilar y desapilar elementos. Luego, la clase InfixToPostfixConverter establece la prioridad y asociatividad de los operadores (+, -, *, /, ^) como base para el algoritmo. Esta estructura permite reorganizar correctamente los operadores sin necesidad de paréntesis, facilitando su evaluación posterior.

```python
class Stack:
    """Implementación sencilla de una pila."""
    def __init__(self):
        self._items = []

    def is_empty(self):
        return not self._items

    def push(self, item):
        """Inserta un elemento en la pila."""
        self._items.append(item)

    def pop(self):
        """Elimina y devuelve el elemento tope. Lanza IndexError si está vacía."""
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """Devuelve el elemento tope sin quitarlo, o None si está vacía."""
        return self._items[-1] if self._items else None


class InfixToPostfixConverter:
    """Convierte expresiones infijas a notación postfija."""
    def __init__(self):
        self.precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
        self.assoc = {'+': 'L', '-': 'L', '*': 'L', '/': 'L', '^': 'R'}
```

La clase InfixToPostfixConverter continúa con dos métodos importantes. El primero, tokenize, convierte la cadena de la expresión en una lista de tokens (números, variables, operadores y paréntesis), ignorando espacios y validando caracteres permitidos. El segundo método, convert, es el que aplica el algoritmo de conversión de notación infija a postfija. Utiliza una pila para manejar los operadores y genera una lista ordenada con los operandos y operadores en el orden correcto. También valida que los paréntesis estén bien balanceados. Estos métodos permiten procesar correctamente expresiones de distintos niveles de complejidad.

```python
class InfixToPostfixConverter:
    def tokenize(self, expr):
        tokens, i = [], 0
        while i < len(expr):
            c = expr[i]
            if c.isspace():
                i += 1; continue
            if c.isalnum() or c == '.':
                j = i
                while j < len(expr) and (expr[j].isalnum() or expr[j] == '.'):
                    j += 1
                tokens.append(expr[i:j]); i = j
            elif c in '()+-*/^':
                tokens.append(c); i += 1
            else:
                raise ValueError(f"Token no válido: {c}")
        return tokens

    def convert(self, expr):
        output, stack = [], Stack()
        for token in self.tokenize(expr):
            if token.replace('.', '', 1).isdigit() or token.isalpha():
                output.append(token)
            elif token == '(':
                stack.push(token)
            elif token == ')':
                while not stack.is_empty() and stack.peek() != '(':
                    output.append(stack.pop())
                if stack.is_empty(): raise ValueError("Paréntesis desbalanceados")
                stack.pop()
```

En esta sección se completa el método convert, que organiza los operadores respetando su prioridad y asociatividad. Al final, extrae todos los operadores restantes de la pila y devuelve la expresión convertida a notación postfija como una cadena.

Además, se define la clase PostfixEvaluator, que permite evaluar directamente expresiones postfijas. Usa un diccionario llamado operators que asocia cada símbolo matemático con una función lambda para realizar la operación correspondiente. Esta estructura hace posible ejecutar operaciones aritméticas de manera dinámica según los símbolos encontrados en la expresión.

```
            else:
                # operador
                while (not stack.is_empty() and stack.peek() != '(' and
                       ((self.assoc[token]=='L' and self.precedence[token]<=self.precedence.get(stack.peek(),0)) or
                        (self.assoc[token]=='R' and self.precedence[token]<self.precedence.get(stack.peek(),0)))):
                    output.append(stack.pop())
                stack.push(token)
        while not stack.is_empty():
            if stack.peek() == '(':
                raise ValueError("Paréntesis desbalanceados")
            output.append(stack.pop())
        return ' '.join(output)


class PostfixEvaluator:
    """Evalúa expresiones en notación postfija."""
    def __init__(self):
        self.operators = {
            '+': lambda a,b: a+b,
            '-': lambda a,b: a-b,
            '*': lambda a,b: a*b,
            '/': lambda a,b: a/b,
            '^': lambda a,b: a**b,
        }
```

La función evaluate de la clase PostfixEvaluator se encarga de evaluar una expresión en notación postfija. Para ello, utiliza una pila. Recorre cada token de la expresión: si es un número, lo convierte a float y lo apila; si es un operador, extrae los dos últimos valores de la pila, aplica la operación y vuelve a apilar el resultado. Si encuentra un token inválido, o si hay errores en la estructura de la expresión (como operandos de más o de menos), lanza un error. Al finalizar, retorna el resultado final. Esta función permite ejecutar directamente expresiones postfijas de forma segura y controlada.

```
    def evaluate(self, postfix_expr):
        stack = Stack()
        for token in postfix_expr.split():
            if token.replace('.', '', 1).isdigit():
                stack.push(float(token))
            elif token in self.operators:
                b = stack.pop(); a = stack.pop()
                stack.push(self.operators[token](a,b))
            else:
                raise ValueError(f"Token inválido: {token}")
        if stack.is_empty(): raise ValueError("Expresión inválida")
        result = stack.pop()
        if not stack.is_empty(): raise ValueError("Operandos sobrantes")
        return result
```

Main:

La función main() permite al usuario ingresar expresiones en notación infija y ver su conversión a notación postfija, además de evaluar su resultado. Utiliza dos objetos: uno de la clase InfixToPostfixConverter para hacer la conversión, y otro de PostfixEvaluator para calcular el resultado. El programa se ejecuta en un ciclo hasta que el usuario escribe "salir", y muestra paso a paso la expresión convertida y su evaluación final. También maneja errores para evitar que el programa se detenga por entradas incorrectas.

```python
from funciones import InfixToPostfixConverter, PostfixEvaluator


def main():
    converter = InfixToPostfixConverter()
    evaluator = PostfixEvaluator()
    print("Convertir y evaluar expresiones infija->postfija")
    while True:
        expr = input("Ingresa expresión infija (o 'salir'): ")
        if expr.lower() == 'salir':
            break
        try:
            postfija = converter.convert(expr)
            print(f"Postfija: {postfija}")
            resultado = evaluator.evaluate(postfija)
            print(f"Resultado: {resultado}\n")
        except Exception as e:
            print(f"Error: {e}\n")


if __name__ == '__main__':
    main()
```

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.

```python
"""
Módulo con la lógica para invertir el orden de las palabras de una frase usando POO y pilas.
Contiene:
- Stack: estructura de datos LIFO genérica.
- WordInverter: clase que usa Stack para invertir frases.
"""

class Stack:
    """Implementación sencilla de una pila."""
    def __init__(self):
        self._items = []

    def is_empty(self):
        return not self._items

    def push(self, item):
        """Añade un elemento al tope de la pila."""
        self._items.append(item)

    def pop(self):
        """Extrae y devuelve el tope de la pila. Error si está vacía."""
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """Devuelve el elemento tope sin extraerlo, o None si está vacía."""
        return self._items[-1] if self._items else None

```

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.

```python


class WordInverter:
    """Invierte el orden de las palabras de una frase usando una pila."""
    def __init__(self, stack_class=Stack):
        # Permite inyección de dependencia para pruebas
        self.stack_class = stack_class

    def invert(self, sentence: str) -> str:
        """Devuelve la frase con el orden de palabras invertido."""
        stack = self.stack_class()
        # Separar por espacios, conservar palabras no vacías
        words = [w for w in sentence.strip().split(' ') if w]
        for word in words:
            stack.push(word)

        inverted = []
        while not stack.is_empty():
            inverted.append(stack.pop())

        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.

```python
from funciones_inversion import WordInverter


def main():
    inverter = WordInverter()  # creamos el invertidor
    print("Invertidor de palabras usando pilas")
    print("Escribe 'salir' para terminar.")
    while True:
        frase = input("Ingresa una frase: ") # pide la frase
        if frase.lower() == 'salir':
            print("¡Hasta luego!")
            break
        try:
            resultado = inverter.invert(frase)  # invierte y muestra
            print(f"Invertido: {resultado}\n")
        except Exception as e:
            print(f"Error: {e}\n")

if __name__ == '__main__':
    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.

```python
class Pila:
    """
    Clase que implementa una estructura tipo pila (stack).
    Sigue el principio LIFO: el ultimo en entrar es el primero en salir.
    """

    def __init__(self):
        self.elementos = []  # Lista que almacena los datos

    def push(self, valor):
        """
        Agrega un elemento al tope de la pila.
        """
        self.elementos.append(valor)

    def pop(self):
        """
        Elimina y retorna el elemento del tope de la pila.
        Si la pila esta vacia, retorna None.
        """
        if not self.is_empty():
            return self.elementos.pop()
        return None

    def peek(self):
        """
        Retorna el elemento del tope sin eliminarlo.
        Si la pila esta vacia, retorna None.
        """
        if not self.is_empty():
            return self.elementos[-1]
        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.

```python
    def peek(self):
        """
        Retorna el elemento del tope sin eliminarlo.
        Si la pila esta vacia, retorna None.
        """
        if not self.is_empty():
            return self.elementos[-1]
        return None

    def is_empty(self):
        """
        Retorna True si la pila esta vacia, False en caso contrario.
        """
        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 ((, [, {).

```python
# Problema 2: Verificación de paréntesis balanceados
# Version: 1.0
# Fecha: 21/05/2025
# Autor: Diego Urbina, Julio Delgadillo, Emmanuel Aguilar

from Pilaejercicio2 import Pila  # Importamos la estructura de pila

def esta_balanceado(cadena):
    """
    Verifica si los parentesis (), {}, [] en una cadena estan balanceados.
    Utiliza una pila para controlar las aperturas y cierres.
    """
    pila = Pila()
    pares = {')': '(', ']': '[', '}': '{'}  # Diccionario que relaciona cierres con aperturas

    for caracter in cadena:
        if caracter in "({[":
            # Si es un parentesis de apertura, lo apilamos
            pila.push(caracter)
        elif caracter in ")}]":
            # Si encontramos un cierre y la pila esta vacia, esta desbalanceado
            if pila.is_empty():
                return False
            # Si el ultimo abierto no corresponde con el cierre, esta mal emparejado
            if pila.pop() != pares[caracter]:
                return False

    # Si al final la pila esta vacia, todos los parentesis fueron cerrados correctamente
    return pila.is_empty()


```

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.

```python

def main():
    print("=== Verificador de parentesis balanceados ===")
    expresion = input("Ingrese una expresion: ").strip()

    if not expresion:
        print("→ La expresion esta vacia.")
    else:
        if esta_balanceado(expresion):
            print("→ Los parentesis estan balanceados.")
        else:
            print("→ Los parentesis NO estan balanceados.")


if __name__ == "__main__":
    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.

```python
class Cancion:
    """
    Clase que representa una cancion en la lista de reproduccion.
    Tiene nombre y enlaces al siguiente y anterior nodo (lista doblemente enlazada).
    """
    def __init__(self, nombre):
        self.nombre = nombre
        self.siguiente = None
        self.anterior = None

    def __str__(self):
        return self.nombre


class ListaReproduccion:
    """
    Clase que simula una lista de reproduccion de musica.
    Permite recorrer hacia adelante o atras, agregar y eliminar canciones.
    """
    def __init__(self):
        self.primera = None
        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.

```
    def agregar_cancion(self, nombre):
        """
        Agrega una cancion al final de la lista.
        """
        nueva = Cancion(nombre)
        if self.primera is None:
            self.primera = nueva
            self.actual = nueva
        else:
            temp = self.primera
            while temp.siguiente:
                temp = temp.siguiente
            temp.siguiente = nueva
            nueva.anterior = temp
        print(f"-> Cancion '{nombre}' agregada a la lista.")

    def eliminar_cancion_actual(self):
        """
        Elimina la cancion que esta siendo reproducida.
        """
        if self.actual is None:
            print("-> No hay cancion para eliminar.")
            return

        nombre = self.actual.nombre
        anterior = self.actual.anterior
        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.

```
        # Reconectar nodos
        if anterior:
            anterior.siguiente = siguiente
        else:
            self.primera = siguiente

        if siguiente:
            siguiente.anterior = anterior

        # Mover el puntero actual
        self.actual = siguiente if siguiente else anterior
        print(f"-> Cancion '{nombre}' eliminada.")

    def siguiente_cancion(self):
        """
        Mueve el puntero a la siguiente cancion si existe.
        """
        if self.actual and self.actual.siguiente:
            self.actual = self.actual.siguiente
            print(f"-> Reproduciendo: {self.actual.nombre}")
        else:
            print("-> No hay siguiente cancion.")
```

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.primera is None:
90              print("-> Lista de reproduccion vacia.")
91              return
92
93          print("Lista de reproduccion:")
94          temp = self.primera
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.

```python
class PriorityQueue:
    """Cola de prioridad con prioridades enteras (menor valor => mayor prioridad)."""
    def __init__(self):
        self._items = []

    def is_empty(self) -> bool:
        """Devuelve True si la cola está vacía."""
        return not self._items

    def enqueue(self, item: str, priority: int) -> None:
        """
        Agrega un elemento con su prioridad.
        priority: entero, menor = mayor prioridad.
        """
        # Insertar y mantener orden por prioridad ascendente
        self._items.append((priority, item))
        self._items.sort(key=lambda x: x[0])

    def dequeue(self) -> tuple:
        """
        Elimina y devuelve el elemento con mayor prioridad (menor valor numérico).
        Lanza IndexError si está vacía.
        Devuelve: (item, priority)
        """
        if self.is_empty():
            raise IndexError("dequeue from empty priority queue")
        pr, itm = self._items.pop(0)
        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.

```python
    def peek(self) -> tuple:
        """
        Devuelve el próximo elemento sin extraerlo.
        Si está vacía, devuelve None.
        """
        if self.is_empty():
            return None
        pr, itm = self._items[0]
        return itm, pr

    def __len__(self) -> int:
        """Número de elementos en la cola."""
        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.

```python
# Interfaz de usuario para la cola de prioridad
def main():
    pq = PriorityQueue()  # crear cola
    print("--- Cola de Prioridad Interactiva ---")
    while True:
        # Mostrar menú
        print("\nOpciones:")
        print("1. Encolar elemento")
        print("2. Desencolar siguiente")
        print("3. Ver siguiente (peek)")
        print("4. Chequear si está vacía")
        print("5. Salir")
        opcion = input("Selecciona una opción: ").strip()

        if opcion == '1':
            # Pedir datos y encolar
            item = input("Elemento (nombre): ").strip()
            try:
                pr = int(input("Prioridad (menor = más urgente): ").strip())
                pq.enqueue(item, pr)
                print(f"Agregado '{item}' con prioridad {pr}")
            except ValueError:
                print("Prioridad inválida. Debe ser un número entero.")

        elif opcion == '2':
            # Desencolar elemento más urgente
            try:
                item, pr = pq.dequeue()
                print(f"Desencolado '{item}' con prioridad {pr}")
            except IndexError as e:
                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.

```python
        elif opcion == '3':
            # Ver siguiente elemento
            res = pq.peek()
            if res:
                print(f"Siguiente en cola: '{res[0]}' con prioridad {res[1]}")
            else:
                print("La cola está vacía.")

        elif opcion == '4':
            # Chequear estado de la cola
            estado = "vacía" if pq.is_empty() else f"no está vacía (tamaño={len(pq)})"
            print(f"La cola {estado}.")

        elif opcion == '5':
            # Salir del programa
            print("¡Hasta luego!")
            break

        else:
            # Opción no válida
            print("Opción no reconocida. Intenta otra vez.")

if __name__ == '__main__':
    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.

```python
class Node:
    # Nodo que almacena un dato y apunta al siguiente nodo
    def __init__(self, data):
        self.data = data  # valor guardado
        self.next = None  # referencia al siguiente nodo, None si no hay

    def __repr__(self):
        # Permite ver el contenido al imprimir el nodo
        return f"Node({self.data})"

class LinkedList:
    # Lista enlazada simple, permite append y búsqueda
    def __init__(self):
        self.head = None  # inicio de la lista

    def append(self, data):
        # Crea un nuevo nodo al final con el dato dado
        new_node = Node(data)
        if not self.head:
            # Si la lista está vacía, head apunta al nuevo nodo
            self.head = new_node
            return
        current = self.head
        # Recorrer hasta llegar al último nodo
        while current.next:
            current = current.next
        # Enlazar el nuevo nodo al final
        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.

```python
    def search(self, target):
        # Busca `target` y devuelve su posición (0-based)
        current = self.head
        index = 0
        # Recorrer nodos hasta encontrar o llegar al final
        while current:
            if current.data == target:
                return index  # encontrado
            current = current.next
            index += 1
        return -1  # no encontrado en toda la lista

    def __str__(self):
        # Devuelve los valores de la lista separados por flechas
        values = []
        current = self.head
        while current:
            values.append(str(current.data))
            current = current.next
        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.

```python
def main():
    print("--- Búsqueda en Lista Enlazada ---")
    # Leer elementos separados por espacio
    datos = input("Ingresa elementos separados por espacios: ").strip().split()
    ll = LinkedList()
    for d in datos:
        ll.append(d)  # construir la lista

    print(f"Lista creada: {ll}")
    while True:
        # Pedir valor a buscar
        objetivo = input("Valor a buscar (o 'salir'): ")
        if objetivo.lower() == 'salir':
            print("Fin de la búsqueda. ¡Adiós!")
            break
        pos = ll.search(objetivo)
        if pos >= 0:
            print(f"'{objetivo}' encontrado en posición {pos}.")
        else:
            print(f"'{objetivo}' no está en la lista.")
        print()  # línea en blanco para legibilidad

if __name__ == '__main__':
    main()
```