Universidad Americana

Facultad de Ingeniería y Arquitectura

Algoritmos y Estructuras de Datos, Grupo 4

Manejo de Funciones, Clases y Paquetes

Realizado por:

Joaquín Alberto Pérez Zúñiga

Nicolás Rafael Laguna Vallejos

Solieth Valentina Trejos Pérez

Docente:

MSc. César Marín López

Implementación de la Clase LinkedList()

1. **node,py:**

```python
"""
Implementación de un nodo en una lista enlazada.
"""

from typing import (
    Generic,
    Optional,
    TypeVar
)

T = TypeVar("T")


class Node(Generic[T]):
    """
    Representa un nodo en una lista enlazada,
    con el dato que almacena, y el proximo elemento en la lista.
    """

    def __init__(self, data: T, next_node: Optional["Node"] = None):
        self.data = data
        self.next = next_node

```

2. **linked_list.py:**

```python
"""
Implementación personalizada de una lista enlazada.
"""

from typing import (
    Generic,
    Optional,
    TypeVar
)

from .node import Node

T = TypeVar("T")


class LinkedList(Generic[T]):
    """
    Implementación personalizada de una lista enlazada.
    """

    def __init__(self) -> None:
        self.head: Optional[Node] = None
        self.size = 0

    def __len__(self) -> int:
        return self.size

    def __repr__(self) -> str:
        return self.__str__()

```

```python
class LinkedList(Generic[T]):
    def __str__(self) -> str:
        self_str = "[ "

        current = self.head
        while current is not None:
            if current.next is not None:
                self_str += f"{str(current.data)}, "
            else:
                self_str += f"{str(current.data)} ]"
            current = current.next

        return self_str

    def is_empty(self) -> bool:
        """
        Determina si la lista esta vacía.
        """

        return self.head is None

    def add_at_front(self, data: T) -> None:
        """
        Agrega un nuevo elemento al inicio de la lista enlazada.
        """

        self.head = Node(data=data, next_node=self.head)
        self.size += 1

```

```python
class LinkedList(Generic[T]):
    def add_at_end(self, data: T) -> None:
        """
        Agrega un nuevo elemento al final de la lista enlazada.
        """

        new_node = Node(data)

        if self.is_empty():
            self.head = new_node
            self.size += 1
            return

        current = self.head
        while True:
            if current.next is not None:
                current = current.next
            else:
                current.next = new_node
                self.size += 1
                return

```

```python
class LinkedList(Generic[T]):
    def update_node(self, index: int, new_data: T) -> None:
        """
        Actualiza el valor de un nodo en la lista.
        """

        if self.is_empty():
            raise IndexError("Índice inválido: no hay elementos en la lista.")
        if index < 0:
            raise IndexError("Índice inválido: debe ser mayor o igual a cero.")
        if index >= self.size:
            raise IndexError(
                "Índice inválido: es fuera de límites; no corresponde a un nodo."
            )

        i = 0
        current = self.head
        while current is not None and i < index:
            i += 1
            current = current.next

        if current is not None:
            current.data = new_data

```

```python
class LinkedList(Generic[T]):
    def search(self, data: T) -> Optional[Node]:
        if self.is_empty():
            return None

        current = self.head
        while current is not None:
            if current.data == data:
                return current
            current = current.next

        return None

    def delete(self, data: T, count: int = 1) -> None:
        """
        Elimina nodos con el valor especificado.
        """

        if self.is_empty():
            return
        if self.size == 1:
            self.head = None

        deletions = 0
        previous = None
        current = self.head

        while current is not None and deletions < count:
            if current.data == data:
                if previous is None:
                    self.head = current.next
                else:
                    previous.next = current.next
                deletions += 1

            previous = current
            current = current.next
```

```python
class LinkedList(Generic[T]):
    def delete_first(self) -> None:
        """
        Elimina el primer nodo de la lista.
        """

        if self.is_empty():
            return

        if self.head.next is None:
            self.head = None
        else:
            self.head = self.head.next

    def delete_last(self) -> None:
        """
        Elimina el último nodo de la lista.
        """

        if self.is_empty():
            return
        if self.size == 1:
            self.delete_first()
            return

        previous = None
        current = self.head

        while True:
            if current.next is None:
                previous.next = None
                return
            previous = current
            current = current.next

```

Ejercicio 1

1. **estudiante.py:**

```python
"""
Implementación de la clase Estudiante.
"""

class Estudiante:
    """
    Representa los datos de un estudiante.
    """

    def __init__(
        self,
        carnet: str,
        nombres: tuple[str, str],
        apellidos: tuple[str, str],
        genero: str,
        peso: float,
        estatura: float,
        promedio: float
    ) -> None:
        self.carnet = carnet
        self.nombres = nombres
        self.apellidos = apellidos
        self.genero = genero
        self.peso = peso
        self.estatura = estatura
        self.promedio = promedio

```

2. **main.py:**

```python
"""
Una escuela de educación primaria requiere un algoritmo que muestre los datos de los
estudiantes de un salón de clase ordenados de forma ascendente, según un parámetro
indicado; este parámetro puede ser cualquiera de los siguientes campos:
carnet, nombres, apellidos, genero, peso, estatura, promedio.
"""

from typing import Union

from . import Estudiante, LinkedList


def input_estudiantes(lista: LinkedList[Estudiante]) -> None:
    """
    Guarda los datos de estudiantes ingresados en una lista enlazada.
    """

    i = 1

    while True:
        print("\nPara terminar, dejar todos los campos vacíos.\n")

        try:
            carnet = input(f"Ingrese el carnet del estudiante #{i}: ")
            nombres = input(f"Ingrese los nombres del estudiante #{i}: ")
            apellidos = input(f"Ingrese los apellidos del estudiante #{i}: ")
            genero = input(f"Ingrese el género del estudiante #{i}: ")

            peso = input(f"Ingrese el peso del estudiante #{i} (en kg): ")
            if peso != "" and float(peso) <= 0:
                raise ValueError

            estatura = input(f"Ingrese la estatura del estudiante #{i} (en m): ")
            if peso != "" and float(estatura) <= 0:
                raise ValueError

```

```
13  def input_estudiantes(lista: LinkedList[Estudiante]) -> None:
33              estatura = input(f"Ingrese la estatura del estudiante #{i} (en m): ")
34              if peso != "" and float(estatura) <= 0:
35                  raise ValueError
36
37              promedio = input(f"Ingrese el promedio del estudiante #{i}: ")
38              if peso != "" and float(promedio) <= 0:
39                  raise ValueError
40
41          except (TypeError, ValueError):
42              input(
43                  "\n¡Error! El peso, promedio y estatura del estudiante deben"
44                  + " ser un número real positivo, intente nuevamente..."
45              )
46
47              continue
48
49          if all(
50              x == ""
51              for x in (carnet, nombres, apellidos, genero, peso, estatura, promedio)
52          ):
53              break
54
55          i += 1
56          lista.add_at_end(Estudiante(
57              carnet=carnet,
58              nombres=tuple(nombres.split()),
59              apellidos=tuple(apellidos.split()),
60              genero=genero,
61              peso=float(peso),
62              estatura=float(estatura),
63              promedio=float(promedio)
64          ))
65
66      print()
67
68
```

```
69  def print_ordenado(lista: LinkedList[Estudiante], filtro: int) -> None:
75      lista_filtrada: list[tuple[str, Union[str, float]]] = []
76
77      # hay que extraer los nombres de todos los estudiantes de la lista
78      # y emparejarlos con el dato que se desea visualizar y ordenar
79
80      current = lista.head
81      while current is not None:
82          # construir un string con el primer nombre y primer apellido
83          key = f"{current.data.nombres[0]} {current.data.apellidos[0]}"
84
85          # extraer el atributo correspondiente al filtro especificado
86          match filtro:
87              case 1:
88                  atributo = "Carnet"
89                  value = current.data.carnet
90              case 2:
91                  atributo = "Nombres"
92                  value = current.data.nombres
93              case 3:
94                  atributo = "Apellidos"
95                  value = current.data.apellidos
96              case 4:
97                  atributo = "Género"
98                  value = current.data.genero
99              case 5:
100                 atributo = "Peso"
101                 value = current.data.peso
102             case 6:
103                 atributo = "Estatura"
104                 value = current.data.estatura
105             case 7:
106                 atributo = "Promedio"
107                 value = current.data.promedio
108
109         # guardar la entrada en la lista
110         lista_filtrada.append((key, value))
111         current = current.next
```

```python
def print_ordenado(lista: LinkedList[Estudiante], filtro: int) -> None:
        # guardar la entrada en la lista
        lista_filtrada.append((key, value))
        current = current.next

    # ordenar la lista de tuplas por el segundo elemento (el atributo del objeto)
    lista_filtrada.sort(key=lambda x: x[1])

    print()
    for i, estudiante in enumerate(lista_filtrada):
        print(f"{i + 1}. {atributo} de {estudiante[0]}: {estudiante[1]}")
    print()


def main() -> None:
    """
    Ejecución del programa.
    """

    lista: LinkedList[Estudiante] = LinkedList()
    print("\nPrimeramente, debe ingresar los datos de los estudiantes de la clase:")
    print("-----------------------------------------------------------------------")

    input_estudiantes(lista)
    print("\nAhora, seleccione el campo por el cual desea ordenar los datos:")
    print("1. Carnet\n2. Nombres\n3. Apellidos\n4. Genero\n5. Peso\n6. Estatura\n7. Promedio")

    seleccion = input("=> ")

    if seleccion.isnumeric():
        print_ordenado(lista, int(seleccion))
    else:
        input("\n¡Error! No seleccionó una opción válida, intente nuevamente...\n")



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

Ejercicio 2

1. ruta.py:

```python
class Ruta:
    """
    Representa una ruta en un mapa.
    """

    def __init__(self):
        # Diccionario que representa el grafo de estaciones
        # Cada estación tiene una lista de tuplas (estación_destino, tiempo_minutos)
        self.estaciones = {}

        # Inicializar el mapa con algunas estaciones de ejemplo
        # En una aplicación real, esto podría cargarse desde un archivo o base de datos
        self.inicializar_mapa_ejemplo()

    def inicializar_mapa_ejemplo(self):
        """
        Inicializa un mapa de ejemplo con estaciones y tiempos entre ellas.
        """

        # Formato: "estación": [(estación_destino, tiempo_en_minutos), ...]
        self.estaciones = {
            "Terminal Central": [("Plaza Mayor", 10), ("Parque Industrial", 15)],
            "Plaza Mayor": [
                ("Terminal Central", 10),
                ("Universidad", 8),
                ("Centro Comercial", 5),
            ],
            "Parque Industrial": [("Terminal Central", 15), ("Zona Residencial", 12)],
            "Universidad": [("Plaza Mayor", 8), ("Hospital", 6)],
            "Centro Comercial": [
                ("Plaza Mayor", 5),
                ("Zona Residencial", 7),
                ("Hospital", 9),
            ],
            "Zona Residencial": [("Parque Industrial", 12), ("Centro Comercial", 7)],
            "Hospital": [("Universidad", 6), ("Centro Comercial", 9)],
        }

```

```python
class Ruta:
    def agregar_estacion(self, nombre):
        """
        Agrega una nueva estación al mapa.
        """

        if nombre not in self.estaciones:
            self.estaciones[nombre] = []
            return True
        return False

    def conectar_estaciones(self, origen, destino, tiempo):
        """
        Conecta dos estaciones con un tiempo determinado en minutos.
        """

        if origen in self.estaciones and destino in self.estaciones:
            # Verificar si ya existe la conexión
            for i, (est, _) in enumerate(self.estaciones[origen]):
                if est == destino:
                    # Actualizar tiempo si ya existe la conexión
                    self.estaciones[origen][i] = (destino, tiempo)
                    return True

            # Agregar nueva conexión si no existe
            self.estaciones[origen].append((destino, tiempo))
            return True
        return False

```

```python
class Ruta:
    def obtener_conexiones(self):
        """
        Devuelve todas las conexiones del mapa.
        """

        conexiones = []
        for origen, destinos in self.estaciones.items():
            for destino, tiempo in destinos:
                conexiones.append((origen, destino, tiempo))
        return conexiones

    def calcular_ruta_mas_rapida(self, inicio, fin):
        """
        Implementa el algoritmo de Dijkstra para encontrar la ruta más rápida.
        """

        if inicio not in self.estaciones or fin not in self.estaciones:
            return None, None

        # Inicializar distancias con valores infinitos
        tiempos = {estacion: float("infinity") for estacion in self.estaciones}
        tiempos[inicio] = 0

        # Diccionario para almacenar el predecesor de cada estación en la ruta más corta
        predecesores = {estacion: None for estacion in self.estaciones}

        # Conjunto de nodos no visitados
        no_visitados = set(self.estaciones.keys())

        while no_visitados:
            # Encontrar la estación no visitada con el tiempo mínimo actual
            actual = min(no_visitados, key=lambda x: tiempos[x])

            # Si llegamos al destino o si el tiempo es infinito (no hay ruta)
            if actual == fin or tiempos[actual] == float("infinity"):
                break
```

```python
class Ruta:
    def calcular_ruta_mas_rapida(self, inicio, fin):
            # Si llegamos al destino o si el tiempo es infinito (no hay ruta)
            if actual == fin or tiempos[actual] == float("infinity"):
                break

            no_visitados.remove(actual)

            # Revisar vecinos de la estación actual
            for vecino, tiempo in self.estaciones[actual]:
                # Calcular nuevo tiempo a través de esta ruta
                nuevo_tiempo = tiempos[actual] + tiempo

                # Si encontramos una ruta más rápida, actualizamos
                if nuevo_tiempo < tiempos[vecino]:
                    tiempos[vecino] = nuevo_tiempo
                    predecesores[vecino] = actual

        # Reconstruir la ruta
        if tiempos[fin] == float("infinity"):
            return None, None  # No hay ruta

        ruta = []
        actual = fin
        while actual:
            ruta.insert(0, actual)
            actual = predecesores[actual]

        return ruta, tiempos[fin]

    def mostrar_estaciones(self):
        """
        Muestra todas las estaciones disponibles.
        """

        return list(self.estaciones.keys())
```

2. main.py:

```python
"""
Se requiere automatizar un mapa que contiene las estaciones de una
ruta previamente establecida para una aplicación que indique, a
partir de un punto de la ruta, el tiempo estimado para llegar
a un destino determinado de la misma.
"""

from . import Ruta


def main():
    """
    Ejecución del programa.
    """

    # Crear el mapa de la ruta
    mapa = Ruta()

    while True:
        print("\n=== SISTEMA DE CÁLCULO DE TIEMPO DE VIAJE ===")
        print("1. Ver estaciones disponibles")
        print("2. Agregar nueva estación")
        print("3. Conectar estaciones")
        print("4. Ver todas las conexiones")
        print("5. Calcular tiempo de viaje")
        print("6. Salir")

        opcion = input("\nSeleccione una opción (1-6): ")

        if opcion == "1":
            estaciones = mapa.mostrar_estaciones()
            print("\nEstaciones disponibles:")
            for i, estacion in enumerate(estaciones, 1):
                print(f"{i}. {estacion}")

        elif opcion == "2":
```

```python
def main():
        elif opcion == "2":
            nombre = input("Ingrese el nombre de la nueva estación: ")
            if mapa.agregar_estacion(nombre):
                print(f"Estación '{nombre}' agregada correctamente.")
            else:
                print(f"La estación '{nombre}' ya existe.")

        elif opcion == "3":
            estaciones = mapa.mostrar_estaciones()
            print("\nEstaciones disponibles:")
            for i, estacion in enumerate(estaciones, 1):
                print(f"{i}. {estacion}")

            try:
                idx_origen = (
                    int(input("\nSeleccione el número de la estación de origen: ")) - 1
                )

                idx_destino = (
                    int(input("Seleccione el número de la estación de destino: ")) - 1
                )

                tiempo = int(
                    input("Ingrese el tiempo en minutos entre las estaciones: ")
                )

                if (
                    0 <= idx_origen < len(estaciones)
                    and
                    0 <= idx_destino < len(estaciones)
                ):
                    origen = estaciones[idx_origen]
                    destino = estaciones[idx_destino]
```

```python
def main():
                if (
                    0 <= idx_origen < len(estaciones)
                    and
                    0 <= idx_destino < len(estaciones)
                ):
                    origen = estaciones[idx_origen]
                    destino = estaciones[idx_destino]

                    if mapa.conectar_estaciones(origen, destino, tiempo):
                        print(
                            f"Conexión de {origen} a {destino} ({tiempo} minutos) establecida."
                        )
                    else:
                        print("No se pudo establecer la conexión.")
                else:
                    print("Selección de estación inválida.")
            except ValueError:
                print("Entrada inválida. Ingrese números válidos.")

        elif opcion == "4":
            conexiones = mapa.obtener_conexiones()
            print("\nConexiones actuales:")
            for i, (origen, destino, tiempo) in enumerate(conexiones, 1):
                print(f"{i}. {origen} → {destino}: {tiempo} minutos")

        elif opcion == "5":
            estaciones = mapa.mostrar_estaciones()
            print("\nEstaciones disponibles:")
            for i, estacion in enumerate(estaciones, 1):
                print(f"{i}. {estacion}")

            try:
                idx_inicio = (
                    int(input("\nSeleccione el número de la estación de inicio: ")) - 1
                )
```

```python
def main():
            try:
                idx_inicio = (
                    int(input("\nSeleccione el número de la estación de inicio: ")) - 1
                )

                idx_fin = (
                    int(input("Seleccione el número de la estación de destino: ")) - 1
                )

                if 0 <= idx_inicio < len(estaciones) and 0 <= idx_fin < len(estaciones):
                    inicio = estaciones[idx_inicio]
                    fin = estaciones[idx_fin]

                    ruta, tiempo_total = mapa.calcular_ruta_mas_rapida(inicio, fin)

                    if ruta:
                        print(f"\nRuta más rápida de {inicio} a {fin}:")
                        print(" → ".join(ruta))
                        print(f"Tiempo total estimado: {tiempo_total} minutos")
                    else:
                        print(f"No se encontró una ruta de {inicio} a {fin}.")
                else:
                    print("Selección de estación inválida.")
            except ValueError:
                print("Entrada inválida. Ingrese números válidos.")

        elif opcion == "6":
            print("¡Gracias por usar el sistema de cálculo de tiempo de viaje!")
            break

        else:
            print("Opción inválida. Por favor, seleccione una opción del 1 al 6.")


if __name__ == "__main__":
    main()

```

Ejercicio 3

1. paciente.py:

```python
"""
Implementación de la clase Paciente.
"""

class Paciente:
    """
    Representa los datos de un paciente médico.
    """

    def __init__(
        self,
        nombre: str,
        edad: int,
        sintoma: str,
        prioridad: int
    ) -> None:
        self.nombre = nombre
        self.edad = edad
        self.sintoma = sintoma
        self.prioridad = prioridad

```

2. main.py:

```python
"""
Una clínica recibe pacientes en orden de llegada. Cada paciente debe ser ingresado al
sistema con los siguientes datos: nombre completo, edad, síntoma principal y prioridad
(de 1 a 5). El sistema debe permitir insertar nuevos pacientes, recorrer la lista
para mostrar el orden de atención, y eliminar a un paciente una vez atendido.
"""

from os import system

from . import LinkedList, Paciente


def agregar_paciente(lista: LinkedList[Paciente]) -> None:
    """
    Pide los datos del nuevo paciente y lo agrega a la lista.
    """

    print(f"\nPaciente {len(lista) + 1}:\n")

    try:
        nombre = input("Nombre: ")
        edad = int(input("Edad: "))
        if edad < 0:
            raise ValueError

        sintoma = input("Síntoma principal: ")
        prioridad = int(input("Prioridad (1-5): "))
        if prioridad < 1 or prioridad > 5:
            raise ValueError

    except (TypeError, ValueError):
        input(
            "\n¡Error! La edad debe ser mayor a cero, y la prioridad"
            + " debe ser un número entre 1 y 5, intente nuevamente..."
        )

        agregar_paciente(lista)
```

```python
def agregar_paciente(lista: LinkedList[Paciente]) -> None:
    lista.add_at_end(Paciente(
        nombre,
        edad,
        sintoma,
        prioridad
    ))


def consultar_pacientes(lista: LinkedList[Paciente]) -> None:
    """
    Muestra todos los pacientes en la lista, ordenados por su prioridad.
    """

    lista_filtrada: list[tuple[str, int]] = []

    current = lista.head
    while current is not None:
        lista_filtrada.append((current.data.nombre, current.data.prioridad))
        current = current.next

    # ordenar la lista de tuplas por el segundo elemento (el atributo del objeto)
    lista_filtrada.sort(key=lambda x: x[1])

    print()
    for i, paciente in enumerate(lista_filtrada):
        print(f"{i + 1}. Prioridad de {paciente[0]}: {paciente[1]}")
    print()
```

```python
def eliminar_paciente(lista: LinkedList[Paciente]) -> None:
    """
    Pide el nombre del paciente a eliminar despues de ser atendido.
    """

    nombre = input("\nIngrese el nombre del paciente a eliminar: ")

    previous = None
    current = lista.head

    while current is not None:
        if current.data.nombre == nombre:
            if previous is None:
                lista.head = current.next
            else:
                previous.next = current.next
        previous = current
        current = current.next


def menu_principal(lista: LinkedList[Paciente]) -> None:
    """
    Loop principal del programa que le muestra un menú de opciones al usuario.
    """

    opcion = ""
    while opcion != "4":
        system("cls || clear")
        print("\nGestión de Pacientes")
        print("-----------------------------------------------------\n")
        print("1. Agregar paciente")
        print("2. Consultar pacientes")
        print("3. Eliminar paciente")
        print("4. Salir")

        opcion = input("\n-> ")
```

```python
def menu_principal(lista: LinkedList[Paciente]) -> None:
        match opcion:
            case "1":
                system("cls || clear")
                agregar_paciente(lista)
            case "2":
                system("cls || clear")
                consultar_pacientes(lista)
                input("\nPresione 'Enter' para regresar al menú principal...")
            case "3":
                system("cls || clear")
                eliminar_paciente(lista)
            case "4":
                print("\nSaliendo del programa...\n")
            case _:
                input("\n¡Opción inválida, intente nuevamente!")


def main() -> None:
    """
    Ejecución del programa.
    """

    menu_principal(LinkedList())


if __name__ == "__main__":
    main()

```

Ejercicio 4

1. acciones_editor.py:

```python
"""
Implementa las clases que representan las acciones permitidas en la aplicación.
"""


class Accion:
    """
    Clase base que representa una acción genérica en el editor.
    """

    def __init__(self, descripcion):
        self.descripcion = descripcion

    def ejecutar(self, editor):
        """
        Método a implementar en las clases derivadas.
        """

    def deshacer(self, editor):
        """
        Método a implementar en las clases derivadas.
        """


class EscribirTexto(Accion):
    """
    Acción de escribir texto en una posición específica.
    """

    def __init__(self, posicion, texto):
        super().__init__(f"Escribir '{texto}' en posición {posicion}")
        self.posicion = posicion
        self.texto = texto

    def ejecutar(self, editor):
        editor.texto = (
            editor.texto[: self.posicion] + self.texto + editor.texto[self.posicion :]
        )
        editor.posicion_cursor = self.posicion + len(self.texto)

    def deshacer(self, editor):
        # Eliminar el texto que se escribió
        editor.texto = (
            editor.texto[: self.posicion]
            + editor.texto[self.posicion + len(self.texto) :]
        )
        editor.posicion_cursor = self.posicion

```

```python
class BorrarTexto(Accion):
    """
    Acción de borrar texto desde una posición específica.
    """

    def __init__(self, posicion, longitud):
        super().__init__(f"Borrar {longitud} caracteres desde posición {posicion}")
        self.posicion = posicion
        self.longitud = longitud
        self.texto_borrado = None  # Se guardará al ejecutar

    def ejecutar(self, editor):
        # Guardar el texto que se va a borrar para poder recuperarlo al deshacer
        self.texto_borrado = editor.texto[self.posicion : self.posicion + self.longitud]
        editor.texto = (
            editor.texto[: self.posicion]
            + editor.texto[self.posicion + self.longitud :]
        )
        editor.posicion_cursor = self.posicion

    def deshacer(self, editor):
        # Restaurar el texto borrado
        editor.texto = (
            editor.texto[: self.posicion]
            + self.texto_borrado
            + editor.texto[self.posicion :]
        )
        editor.posicion_cursor = self.posicion + len(self.texto_borrado)

```

```python
class CopiarTexto(Accion):
    """
    Acción de copiar texto al portapapeles.
    """

    def __init__(self, posicion, longitud):
        super().__init__(f"Copiar {longitud} caracteres desde posición {posicion}")
        self.posicion = posicion
        self.longitud = longitud
        self.texto_anterior_portapapeles = None  # Se guardará al ejecutar

    def ejecutar(self, editor):
        # Guardar el contenido anterior del portapapeles
        self.texto_anterior_portapapeles = editor.portapapeles
        editor.portapapeles = editor.texto[
            self.posicion : self.posicion + self.longitud
        ]

    def deshacer(self, editor):
        # Restaurar el portapapeles anterior
        editor.portapapeles = self.texto_anterior_portapapeles

```

```python
class PegarTexto(Accion):
    """
    Acción de pegar texto desde el portapapeles.
    """

    def __init__(self, posicion):
        super().__init__(f"Pegar texto en posición {posicion}")
        self.posicion = posicion

    def ejecutar(self, editor):
        if editor.portapapeles:
            editor.texto = (
                editor.texto[: self.posicion]
                + editor.portapapeles
                + editor.texto[self.posicion :]
            )
            editor.posicion_cursor = self.posicion + len(editor.portapapeles)

    def deshacer(self, editor):
        if editor.portapapeles:
            # Eliminar el texto que se pegó
            editor.texto = (
                editor.texto[: self.posicion]
                + editor.texto[self.posicion + len(editor.portapapeles) :]
            )
            editor.posicion_cursor = self.posicion

```

2. **editor.py:**

```python
"""
Implementación de la clase principal de la aplicación.
"""

from .acciones_editor import (
    EscribirTexto,
    BorrarTexto,
    CopiarTexto,
    PegarTexto
)


class EditorTexto:
    """
    Clase principal que simula un editor de texto con historial de acciones.
    """

    def __init__(self):
        self.texto = ""
        self.posicion_cursor = 0
        self.portapapeles = ""
        self.historial = []  # Lista de acciones realizadas
        self.posicion_actual = -1  # Índice de la última acción ejecutada (-1 significa ninguna)

    def ejecutar_accion(self, accion):
        """
        Ejecuta una nueva acción y la añade al historial.
        """

        # Si hay acciones en el historial después de la posición actual, se eliminan
        if self.posicion_actual < len(self.historial) - 1:
            self.historial = self.historial[:self.posicion_actual + 1]

        # Ejecutar la acción
        accion.ejecutar(self)

```

```
13  class EditorTexto:
25      def ejecutar_accion(self, accion):
37          # Añadir al historial
38          self.historial.append(accion)
39          self.posicion_actual += 1
40
41      def deshacer(self):
42          """
43          Deshace la última acción ejecutada.
44          """
45
46          if self.posicion_actual >= 0:
47              # Deshacer la acción actual
48              self.historial[self.posicion_actual].deshacer(self)
49              self.posicion_actual -= 1
50              return True
51          return False
52
53      def rehacer(self):
54          """
55          Rehace la siguiente acción en el historial.
56          """
57
58          if self.posicion_actual < len(self.historial) - 1:
59              # Avanzar a la siguiente acción
60              self.posicion_actual += 1
61              # Ejecutar esa acción
62              self.historial[self.posicion_actual].ejecutar(self)
63              return True
64          return False
65
```

```
13  class EditorTexto:
66      def escribir(self, texto):
67          """
68          Método para escribir texto en la posición actual del cursor.
69          """
70
71          accion = EscribirTexto(self.posicion_cursor, texto)
72          self.ejecutar_accion(accion)
73
74      def borrar(self, longitud):
75          """
76          Método para borrar texto desde la posición actual del cursor.
77          """
78
79          if self.posicion_cursor - longitud >= 0:
80              accion = BorrarTexto(self.posicion_cursor - longitud, longitud)
81              self.ejecutar_accion(accion)
82
83      def copiar(self, longitud):
84          """
85          Método para copiar texto desde la posición actual del cursor.
86          """
87
88          if self.posicion_cursor + longitud <= len(self.texto):
89              accion = CopiarTexto(self.posicion_cursor, longitud)
90              self.ejecutar_accion(accion)
91
92      def pegar(self):
93          """
94          Método para pegar el texto del portapapeles en la posición actual del cursor.
95          """
96
97          if self.portapapeles:
98              accion = PegarTexto(self.posicion_cursor)
99              self.ejecutar_accion(accion)
100
```

```
13  class EditorTexto:
101     def mostrar_historial(self):
102         """
103         Muestra el historial de acciones.
104         """
105 
106         print("\nHistorial de acciones:")
107         for i, accion in enumerate(self.historial):
108             marcador = "→" if i == self.posicion_actual else " "
109             print(f"{marcador} {i+1}. {accion.descripcion}")
110 
111     def mostrar_estado(self):
112         """
113         Muestra el estado actual del editor.
114         """
115 
116         print("\nEditor de Texto")
117         print("==============")
118         print(f"Texto: '{self.texto}'")
119         print(f"Posición del cursor: {self.posicion_cursor}")
120         print(f"Portapapeles: '{self.portapapeles}'")
121 
122         # Mostrar posición del cursor visualmente
123         print("\nRepresentación visual:")
124         cursor_visual = " " * self.posicion_cursor + "^"
125         print(self.texto)
126         print(cursor_visual)
127 
```

3. main.py:

```python
"""
Se desea implementar el historial de acciones realizadas por un usuario en un
editor de texto (como escribir, borrar, pegar, copiar). Cada acción debe
guardarse en orden y poder recorrerlas en ambas direcciones,
simulando las acciones de Deshacer y Rehacer.
"""

from . import EditorTexto


def menu_principal():
    """
    Función para mostrar el menú principal.
    """

    print("\n=== EDITOR DE TEXTO ===")
    print("1. Escribir texto")
    print("2. Borrar texto")
    print("3. Copiar texto")
    print("4. Pegar texto")
    print("5. Deshacer")
    print("6. Rehacer")
    print("7. Mover cursor")
    print("8. Ver historial")
    print("9. Salir")
    opcion = input("Seleccione una opción: ")
    return opcion

```

```python
def main():
    """
    Ejecución del programa.
    """

    editor = EditorTexto()

    while True:
        editor.mostrar_estado()
        opcion = menu_principal()

        if opcion == "1":  # Escribir
            texto = input("Texto a escribir: ")
            editor.escribir(texto)

        elif opcion == "2":  # Borrar
            try:
                longitud = int(input("Número de caracteres a borrar: "))
                editor.borrar(longitud)
            except ValueError:
                print("Por favor, ingrese un número válido.")

        elif opcion == "3":  # Copiar
            try:
                longitud = int(input("Número de caracteres a copiar: "))
                editor.copiar(longitud)
            except ValueError:
                print("Por favor, ingrese un número válido.")

        elif opcion == "4":  # Pegar
            editor.pegar()

        elif opcion == "5":  # Deshacer
            if not editor.deshacer():
                print("No hay más acciones para deshacer.")

```

```
30  def main():
66          elif opcion == "6":  # Rehacer
67              if not editor.rehacer():
68                  print("No hay más acciones para rehacer.")
69
70          elif opcion == "7":  # Mover cursor
71              try:
72                  nueva_posicion = int(
73                      input(f"Nueva posición del cursor (0-{len(editor.texto)}): ")
74                  )
75                  if 0 <= nueva_posicion <= len(editor.texto):
76                      editor.posicion_cursor = nueva_posicion
77                  else:
78                      print("Posición fuera de rango.")
79              except ValueError:
80                  print("Por favor, ingrese un número válido.")
81
82          elif opcion == "8":  # Ver historial
83              editor.mostrar_historial()
84
85          elif opcion == "9":  # Salir
86              print("¡Hasta luego!")
87              break
88
89          else:
90              print("Opción no válida. Intente de nuevo.")
91
92
93  if __name__ == "__main__":
94      main()
95
```