UNIVERSIDAD COOPERATIVA DE COLOMBIA (UCC)

ESTRUCTURA DE DATOS
(DOCUMENTACION DE EJERCICIO EVALUATIVO DE PILAS)

Jesús Rendón

SEMESTRE V

2025

Documentación: Algoritmo de Ordenamiento de Pilas

Índice

1. Introducción
2. Estructura de Datos: Clase Pila
3. Algoritmos de Ordenamiento
4. Funciones Auxiliares
5. Función Principal
6. Análisis de Complejidad
7. Casos de Uso y Ejemplos
8. Limitaciones y Posibles Mejoras

Introducción

Este programa implementa el ordenamiento de dos tipos de pilas:

- Una pila de números reales, ordenada en forma ascendente
- Una pila de caracteres, ordenada tanto en forma ascendente como descendente

El programa respeta la restricción de utilizar únicamente pilas auxiliares para el ordenamiento, sin emplear estructuras de datos adicionales. Utiliza un enfoque similar al algoritmo de ordenamiento por inserción.

Estructura de Datos: Clase Pila

La implementación se basa en una clase `Pila` que encapsula la estructura de datos y sus operaciones:

```python
class Pila:
    def __init__(self):
        self.items = []

    def apilar(self, item):
        #Agrega un elemento a la pila.
        self.items.append(item)

    def desapilar(self):
        #Elimina y devuelve el último elemento de la pila.
        if not self.esta_vacia():
            return self.items.pop()
        else:
            raise Exception("La pila está vacía")

    def esta_vacia(self):
        #Verifica si la pila está vacía.
        return len(self.items) == 0

    def ver_tope(self):
        #Devuelve el elemento en la cima sin eliminarlo.
        if not self.esta_vacia():
            return self.items[-1]
        else:
            raise Exception("La pila está vacía")

    def __len__(self):
        #Retorna la cantidad de elementos en la pila.
        return len(self.items)
```

Métodos de la clase Pila:

Método	Descripción	Complejidad
`__init__()`	Constructor que inicializa una pila vacía	O(1)
`apilar(item)`	Añade un elemento al tope de la pila	O(1)
`desapilar()`	Extrae y retorna el elemento del tope	O(1)
`esta_vacia()`	Verifica si la pila no tiene elementos	O(1)
`ver_tope()`	Retorna el elemento del tope sin eliminarlo	O(1)
`__len__()`	Retorna la cantidad de elementos en la pila	O(1)

Algoritmos de Ordenamiento

Ordenamiento Ascendente

```
def ordenar_pila_ascendente(pila):
    #Ordena la pila en orden ascendente usando una pila auxiliar.
    pila_auxiliar = Pila()

    while not pila.esta_vacia():
        elemento_actual = pila.desapilar()

        while (not pila_auxiliar.esta_vacia() and
               pila_auxiliar.ver_tope() > elemento_actual):
            pila.apilar(pila_auxiliar.desapilar())

        pila_auxiliar.apilar(elemento_actual)

    while not pila_auxiliar.esta_vacia():
        pila.apilar(pila_auxiliar.desapilar())

    return pila
```

Funcionamiento del ordenamiento ascendente:

1. **Inicialización**: Se crea una pila auxiliar vacía.
2. **Proceso principal**:
    - Se recorre la pila original elemento por elemento.
    - Para cada elemento extraído:
        - Se compara con los elementos de la pila auxiliar.
        - Si el elemento actual es menor que el tope de la pila auxiliar, se desapilan elementos de la auxiliar y se apilan en la original.
        - Se inserta el elemento actual en la posición correcta de la pila auxiliar.
3. **Restauración**: Finalmente, se transfieren todos los elementos ordenados de la pila auxiliar a la pila original.

Ordenamiento Descendente

```
def ordenar_pila_descendente(pila):
    #Ordena la pila en orden descendente usando una pila auxiliar.
    pila_auxiliar = Pila()

    while not pila.esta_vacia():
        elemento_actual = pila.desapilar()

        while (not pila_auxiliar.esta_vacia() and
               pila_auxiliar.ver_tope() < elemento_actual):
            pila.apilar(pila_auxiliar.desapilar())

        pila_auxiliar.apilar(elemento_actual)

    while not pila_auxiliar.esta_vacia():
        pila.apilar(pila_auxiliar.desapilar())

    return pila
```

Funcionamiento del ordenamiento descendente:

1. **Inicialización**: Se crea una pila auxiliar vacía.
2. **Proceso principal**:
   - Se recorre la pila original elemento por elemento.
   - Para cada elemento extraído:
     - Se compara con los elementos de la pila auxiliar.
     - Si el elemento actual es mayor que el tope de la pila auxiliar, se desapilan elementos de la auxiliar y se apilan en la original.
     - Se inserta el elemento actual en la posición correcta de la pila auxiliar.

3. **Restauración**: Finalmente, se transfieren todos los elementos ordenados de la pila auxiliar a la pila original.

Funciones Auxiliares

Impresión de Pila

```
def imprimir_pila(pila, nombre):
    #Muestra los elementos de la pila sin modificarlos.
    print(f"{nombre}: ", end="")
    pila_temporal = Pila()

    while not pila.esta_vacia():
        elemento = pila.desapilar()
        print(elemento, end=" ")
        pila_temporal.apilar(elemento)

    while not pila_temporal.esta_vacia():
        pila.apilar(pila_temporal.desapilar())

    print()
```

Funcionamiento de `imprimir_pila()`:

1. Muestra el nombre identificador de la pila.
2. Crea una pila temporal para almacenar elementos.
3. Mientras extrae y muestra cada elemento, lo guarda en la pila temporal.
4. Restaura la pila original con los elementos de la pila temporal.

Ingreso de Elementos

```
def ingresar_elementos_pila(tipo):
    #Permite al usuario ingresar números o caracteres en una pila.
    pila = Pila()

    if tipo == "numeros":
        print("\nIngrese números reales ('fin' para terminar):")
        while True:
            entrada = input("Número: ")
            if entrada.lower() == 'fin':
                break
            try:
                pila.apilar(float(entrada))
            except ValueError:
                print("Entrada inválida. Ingrese un número válido.")

    elif tipo == "caracteres":
        print("\nIngrese caracteres ('fin' para terminar):")
        while True:
            entrada = input("Caracter: ")
            if entrada.lower() == 'fin':
                break
            if len(entrada) == 1:
                pila.apilar(entrada)
            else:
                print("Ingrese solo un caracter a la vez.")

    return pila
```

Funcionamiento de `ingresar_elementos_pila()`:

1. Crea una pila vacía.
2. Según el tipo especificado ("numeros" o "caracteres"):
   - Solicita al usuario que ingrese elementos hasta que ingrese "fin".
   - Para números: convierte las entradas a flotantes con manejo de errores.
   - Para caracteres: verifica que se ingrese un solo caracter a la vez.
3. Retorna la pila con los elementos ingresados.

Función Principal

```python
def main():
    print("Ordenamiento de pilas")
    print("=====================")

    #Pila de números reales ordenada ascendentemente.
    pila_numeros = ingresar_elementos_pila("numeros")

    print("\nPila de números original:")
    imprimir_pila(pila_numeros, "Números")

    ordenar_pila_ascendente(pila_numeros)

    print("\nPila de números ordenada ascendentemente:")
    imprimir_pila(pila_numeros, "Números")

    #Pila de caracteres ordenada ascendente y descendente.
    pila_caracteres = ingresar_elementos_pila("caracteres")

    print("\nPila de caracteres original:")
    imprimir_pila(pila_caracteres, "Caracteres")

    #Crear copias de la pila original.
    pila_caracteres_ascendente= Pila()
    pila_caracteres_descendente= Pila()
    pila_temp = Pila()

    while not pila_caracteres.esta_vacia():
        pila_temp.apilar(pila_caracteres.desapilar())

    while not pila_temp.esta_vacia():
```

```
        caracter = pila_temp.desapilar()
        pila_caracteres.apilar(caracter)
        pila_caracteres_ascendente.apilar(caracter)
        pila_caracteres_descendente.apilar(caracter)

    ordenar_pila_ascendente(pila_caracteres_ascendente)

    print("\nPila de caracteres ordenada ascendentemente:")
    imprimir_pila(pila_caracteres_ascendente, "Caracteres")

    ordenar_pila_descendente(pila_caracteres_descendente)

    print("\nPila de caracteres ordenada descendentemente:")
    imprimir_pila(pila_caracteres_descendente, "Caracteres")
```

Flujo de la función `main()`:

1. **Primera parte - Pila de números reales**:
   - Solicita al usuario ingresar números reales.
   - Muestra la pila original.
   - Ordena la pila en orden ascendente.
   - Muestra la pila ordenada.
2. **Segunda parte - Pila de caracteres**:
   - Solicita al usuario ingresar caracteres.
   - Muestra la pila original.
   - Crea dos copias de la pila original:
     - Una para ordenar en forma ascendente.
     - Otra para ordenar en forma descendente.
   - Ordena cada copia según corresponda.

- Muestra ambas pilas ordenadas.

Análisis de Complejidad

Complejidad Temporal

Función	Complejidad	Explicación
`ordenar_pila_ascendente()`	$O(n^2)$	En el peor caso, cada elemento debe compararse con todos los demás
`ordenar_pila_descendente()`	$O(n^2)$	Similar al ordenamiento ascendente
`imprimir_pila()`	O(n)	Recorre todos los elementos una vez
`ingresar_elementos_pila()`	O(n)	Depende del número de elementos ingresados
Operaciones de la pila	O(1)	Las operaciones básicas son de tiempo constante

Complejidad Espacial

- **Ordenamiento**: O(n) adicional para la pila auxiliar
- **Impresión**: O(n) adicional para la pila temporal

Función principal: O(n) para las pilas auxiliares

Casos de Uso y Ejemplos

Ejemplo 1: Ordenamiento de números reales

Entrada:

```
3.5
1.2
4.7
2.1
fin
```

Salida:

```
Pila de números original:
Números: 3.5 1.2 4.7 2.1

Pila de números ordenada ascendentemente:
Números: 1.2 2.1 3.5 4.7
```

Ejemplo 2: Ordenamiento de caracteres

Entrada:

```
z
a
m
c
fin
```

Salida:

```
Pila de caracteres original:
Caracteres: z a m c

Pila de caracteres ordenada ascendentemente:
Caracteres: a c m z

Pila de caracteres ordenada descendentemente:
Caracteres: z m c a
```

Limitaciones y Posibles Mejoras

Limitaciones:

1. **Eficiencia**: El algoritmo de ordenamiento tiene complejidad $O(n^2)$, lo que puede ser ineficiente para grandes conjuntos de datos.
2. **Manejo de Errores**: El manejo de excepciones es básico.
3. **Tipos de Datos**: Solo se manejan números reales y caracteres individuales.

Posibles Mejoras:

1. **Algoritmo de Ordenamiento**: Implementar un algoritmo más eficiente adaptado a pilas.
2. **Interfaz de Usuario**: Mejorar la interacción con mensajes más claros y validaciones adicionales.
3. **Generalización**: Extender para manejar otros tipos de datos o comparaciones personalizadas.
4. **Opciones Adicionales**: Añadir funcionalidades como guardado/carga de datos o estadísticas.
5. **Optimización**: Reducir la cantidad de operaciones de apilado/desapilado para mejorar el rendimiento.
6. **Visualización**: Implementar una representación visual del proceso de ordenamiento.