Ejercicios

LISTAS

Ejercicio 1: Agregar un elemento al inicio de la lista enlazada.

Planteamiento: Se pide crear una función que agregue un elemento al inicio de la lista enlazada. La implementación de la clase Nodo en Python se puede ver en la siguiente estructura:

In [ ]:
```
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.siguiente = None
```

La implementación de la función para agregar un elemento al inicio de la lista enlazada se puede ver a continuación:

In [ ]:
```
def agregar_inicio(lista, valor):
    nuevo_nodo = Nodo(valor)
    nuevo_nodo.siguiente = lista
    return nuevo_nodo
```

La función recibe dos argumentos: la lista enlazada y el valor del nuevo nodo a agregar. Primero, se crea un nuevo nodo con el valor proporcionado. Luego, se establece el siguiente nodo del nuevo nodo como la lista enlazada original. Finalmente, se regresa el nuevo nodo, que se convierte en la nueva cabeza de la lista enlazada.

Ejercicio 2: Eliminar un elemento de la lista enlazada.

Planteamiento: Se pide crear una función que elimine un elemento de la lista enlazada. (Se utiliza la misma declaración de nodo)

In [ ]:
```python
def eliminar(lista, valor):
    if lista is None:
        return None
    if lista.valor == valor:
        return lista.siguiente
    nodo_actual = lista
    while nodo_actual.siguiente is not None:
        if nodo_actual.siguiente.valor == valor:
            nodo_actual.siguiente = nodo_actual.siguiente.siguiente
            return lista
        nodo_actual = nodo_actual.siguiente
    return lista
```

La función recibe dos argumentos: la lista enlazada y el valor del nodo a eliminar. Si la lista enlazada es nula, se regresa nulo. Si el valor del primer nodo es el valor a eliminar, se regresa el siguiente nodo. Si no, se recorre la lista enlazada hasta encontrar el nodo con el valor a eliminar. Luego, se establece el siguiente nodo del nodo actual como el siguiente del siguiente nodo, eliminando el nodo con el valor proporcionado. Finalmente, se regresa la lista enlazada.

Ejercicio 3: Recorrer la lista enlazada e imprimir los valores

Planteamiento: Recorrer la lista enlazada e imprimir los valores.

In [ ]:
```python
def imprimir(lista):
    nodo_actual = lista
    while nodo_actual is not None:
        print(nodo_actual.valor)
        nodo_actual = nodo_actual.siguiente
```

La función recibe un argumento: la lista enlazada. Se recorre la lista enlazada mientras el nodo actual no sea nulo. En cada iteración, se imprime el valor del nodo actual y se actualiza el nodo actual al siguiente nodo en la lista enlazada.

Ejercicio 4: Contar el número de nodos en una lista enlazada.

Planteamiento: Dada una lista enlazada, se te pide contar el número total de nodos que contiene.

Explicación: Para resolver este ejercicio, puedes recorrer la lista enlazada comenzando desde el primer nodo y contar el número de nodos hasta llegar al final de la lista.

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

def contar_nodos(lista):
    contador = 0
    nodo_actual = lista

    while nodo_actual is not None:
        contador += 1
        nodo_actual = nodo_actual.siguiente

    return contador
```

En este código, la función contar_nodos recibe el nodo inicial de la lista enlazada y utiliza un bucle while para iterar a través de los nodos. Se incrementa un contador en cada iteración y se actualiza el nodo actual al siguiente nodo. El bucle se detiene cuando el nodo actual es None, lo que indica que hemos llegado al final de la lista. Finalmente, se devuelve el contador con el número total de nodos.

Ejercicio 5: Buscar un elemento en una lista enlazada.

Planteamiento: Dada una lista enlazada y un valor objetivo, se te pide encontrar si el valor está presente en la lista y, en caso afirmativo, devolver True; de lo contrario, devolver False.

Explicación: Para resolver este ejercicio, puedes recorrer la lista enlazada nodo por nodo y comparar el valor de cada nodo con el valor objetivo.

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

def buscar_elemento(lista, objetivo):
    nodo_actual = lista

    while nodo_actual is not None:
        if nodo_actual.dato == objetivo:
            return True
        nodo_actual = nodo_actual.siguiente

    return False
```

En este código, la función buscar_elemento recibe el nodo inicial de la lista enlazada y el valor objetivo a buscar. Se utiliza un bucle while para iterar a través de los nodos y se compara el valor de cada nodo con el valor objetivo. Si se encuentra una coincidencia, se devuelve True. Si se recorre toda la lista sin encontrar el valor objetivo, se devuelve False.

Ejercicio 6: Eliminar duplicados de una lista enlazada.

Planteamiento: Dada una lista enlazada, se te pide eliminar todos los nodos duplicados, dejando solo un nodo con cada valor único.

Explicación: Para resolver este ejercicio, puedes utilizar un conjunto para realizar un seguimiento de los valores únicos encontrados hasta ahora. A medida que recorres la lista enlazada, si encuentras un valor que ya está en el conjunto, eliminas ese nodo de la lista enlazada. Si el valor no está en el conjunto, lo agregas al conjunto y continúas al siguiente nodo.

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

def eliminar_duplicados(lista):
    valores = set()
    nodo_actual = lista
    nodo_anterior = None

    while nodo_actual is not None:
        if nodo_actual.dato in valores:
            # Eliminar nodo duplicado
            nodo_anterior.siguiente = nodo_actual.siguiente
        else:
            # Agregar valor al conjunto y avanzar al siguiente nodo
            valores.add(nodo_actual.dato)
            nodo_anterior = nodo_actual

        nodo_actual = nodo_actual.siguiente
```

En este código, la función eliminar_duplicados recibe el nodo inicial de la lista enlazada. Se utiliza un conjunto llamado valores para realizar un seguimiento de los valores únicos encontrados hasta ahora. Se utiliza un bucle while para recorrer la lista enlazada y se comprueba si el valor del nodo actual está presente en el conjunto. Si es así, significa que es un nodo duplicado y se elimina conectando el nodo anterior con el siguiente nodo. Si el valor no está en el conjunto, se agrega al conjunto y se avanza al siguiente nodo. Al finalizar el bucle, todos los nodos duplicados habrán sido eliminados.

Estos son solo algunos ejemplos de ejercicios relacionados con listas enlazadas en Python. Hay muchas otras operaciones y problemas interesantes que se pueden abordar utilizando esta estructura de datos. Espero que estos ejercicios te resulten útiles para practicar y comprender mejor las listas enlazadas.

PILAS

Ejercicio 7: Verificar si una expresión de paréntesis está balanceada.

Planteamiento: Dada una cadena que representa una expresión matemática con paréntesis, se te pide verificar si los paréntesis están balanceados, es decir, si se abren y cierran correctamente.

Explicación: Para resolver este ejercicio, puedes utilizar una pila. Recorres la cadena de izquierda a derecha y, cada vez que encuentres un paréntesis de apertura, lo agregas a la pila. Si encuentras un paréntesis de cierre, compruebas si la pila está vacía o si el paréntesis en la

In [ ]:

```python
def verificar_balance(expresion):
    pila = []

    for caracter in expresion:
        if caracter == '(':
            pila.append(caracter)
        elif caracter == ')':
            if len(pila) == 0 or pila[-1] != '(':
                return False
            pila.pop()

    return len(pila) == 0
```

En este código, la función verificar_balance recibe la expresión como una cadena. Se utiliza una lista como pila. Se recorre la cadena de entrada y se realizan las operaciones descritas anteriormente. Si al finalizar la pila está vacía, se devuelve True, lo que indica que los paréntesis están balanceados. En caso contrario, se devuelve False.

Ejercicio 8: Convertir una expresión infija a una expresión posfija (notación polaca inversa). (Buscar en Google)

Planteamiento: Dada una expresión matemática en notación infija, se te pide convertirla a notación posfija utilizando una pila.

Explicación: Para resolver este ejercicio, puedes utilizar el algoritmo de conversión de expresiones infijas a posfijas utilizando una pila y el uso de operadores y paréntesis. El algoritmo implica recorrer la expresión de izquierda a derecha y realizar ciertas operaciones dependiendo del tipo de símbolo encontrado (operador, paréntesis, número). Al finalizar, la pila contendrá la expresión en notación posfija.

In [ ]:

```python
def infija_a_posfija(expresion):
    precedencia = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    pila = []
    resultado = ''

    for caracter in expresion:
        if caracter.isalnum():
            resultado += caracter
        elif caracter == '(':
            pila.append(caracter)
        elif caracter == ')':
            while pila and pila[-1] != '(':
                resultado += pila.pop()
            pila.pop()
        else:
            while pila and pila[-1] != '(' and precedencia[caracter] <= pre
                resultado += pila.pop()
            pila.append(caracter)

    while pila:
        resultado += pila.pop()

    return resultado
```

En este código, la función infija_a_posfija recibe la expresión infija como una cadena. Se define un diccionario precedencia que asigna una prioridad a cada operador. Se utiliza una lista como pila para realizar las operaciones necesarias. Se recorre la expresión de izquierda a derecha y se realizan diferentes operaciones dependiendo del tipo de símbolo encontrado. Si el símbolo es alfanumérico (un número o variable), se agrega directamente al resultado. Si es un paréntesis de apertura, se agrega a la pila. Si es un paréntesis de cierre, se sacan los elementos de la pila y se agregan al resultado hasta encontrar el paréntesis de apertura correspondiente. Si es un operador, se compara su precedencia con la del operador en la parte superior de la pila y se van sacando los operadores de la pila hasta que se cumplan las condiciones. Al finalizar, se sacan todos los elementos restantes de la pila y se agregan al resultado. Finalmente, se devuelve el resultado, que representa la expresión en notación posfija.

Ejercicio 9: Revertir una pila.

Planteamiento: Dada una pila, se te pide invertir su orden, es decir, el elemento que estaba en la parte superior debe quedar en la parte inferior, y viceversa.

Explicación: Para resolver este ejercicio, puedes utilizar una pila auxiliar. Recorres la pila original y vas sacando cada elemento y agregándolo a la pila auxiliar. Luego, vacías la pila original y vas sacando los elementos de la pila auxiliar y los agregas de nuevo a la pila original. Al finalizar, la pila original contendrá los elementos invertidos.

In [ ]:

```python
def invertir_pila(pila):
    pila_auxiliar = []

    while pila:
        pila_auxiliar.append(pila.pop())

    while pila_auxiliar:
        pila.append(pila_auxiliar.pop())

    return pila
```

En este código, la función invertir_pila recibe la pila original como parámetro. Se utiliza una lista pila_auxiliar como pila auxiliar para almacenar los elementos temporalmente. Se utiliza un bucle while para sacar elementos de la pila original y agregarlos a la pila auxiliar. Luego, se vacía la pila original y se utiliza otro bucle while para sacar elementos de la pila auxiliar y agregarlos nuevamente a la pila original. Finalmente, se devuelve la pila invertida.

Ejercicio 10: Implementar una pila que admita obtener el mínimo elemento en tiempo constante.

Planteamiento: Se te pide implementar una pila en la cual se pueda obtener el mínimo elemento en tiempo constante, es decir, O(1).

Explicación: Para resolver este ejercicio, puedes utilizar dos pilas. Una pila almacena los elementos en su orden normal, mientras que la otra pila se utiliza para realizar un seguimiento del mínimo elemento hasta el momento. Cada vez que se agrega un nuevo elemento a la pila, se compara con el elemento en la parte superior de la pila de mínimos. Si es menor o igual, se agrega a la pila de mínimos. Al realizar operaciones de apilado y desapilado, también se actualiza la pila de mínimos. De esta manera, puedes obtener el mínimo elemento en tiempo constante.

In [ ]:
```python
class MinPila:
    def __init__(self):
        self.pila = []
        self.pila_min = []

    def apilar(self, elemento):
        self.pila.append(elemento)
        if len(self.pila_min) == 0 or elemento <= self.pila_min[-1]:
            self.pila_min.append(elemento)

    def desapilar(self):
        elemento = self.pila.pop()
        if elemento == self.pila_min[-1]:
            self.pila_min.pop()
        return elemento

    def obtener_minimo(self):
        return self.pila_min[-1]
```

Ejercicio 11: Verificar si una cadena de caracteres es un palíndromo.

Planteamiento: Dada una cadena de caracteres, se te pide verificar si es un palíndromo, es decir, si se lee de la misma manera de izquierda a derecha y de derecha a izquierda.

Explicación: Para resolver este ejercicio, puedes utilizar una pila. Recorres la cadena de caracteres y apilas cada carácter en la pila. Luego, desapilas los caracteres de la pila y los comparas con los caracteres de la cadena en el orden inverso. Si todos los caracteres coinciden, la cadena es un palíndromo.

In [ ]:
```python
def es_palindromo(cadena):
    pila = Pila()
    for caracter in cadena:
        pila.apilar(caracter)

    for caracter in cadena:
        if caracter != pila.desapilar():
            return False

    return True
```

En este código, la función es_palindromo recibe la cadena de caracteres como parámetro. Se crea una instancia de la clase Pila y se apilan los caracteres de la cadena. Luego, se recorre la cadena nuevamente y se compara cada carácter con el resultado de desapilar de la pila. Si algún carácter no coincide, se devuelve False. Si todos los caracteres coinciden, se devuelve True.

Ejercicio 12: Convertir un número decimal a binario utilizando una pila.

Planteamiento: Dado un número decimal, se te pide convertirlo a su equivalente binario utilizando una pila.

Explicación: Para resolver este ejercicio, puedes utilizar una pila. Realizas divisiones sucesivas del número decimal entre 2 y apilas los residuos. Luego, desapilas los residuos y los concatenas en orden inverso para obtener la representación binaria del número.

In [ ]:
```
def decimal_a_binario(decimal):
    pila = Pila()

    while decimal > 0:
        residuo = decimal % 2
        pila.apilar(residuo)
        decimal //= 2

    binario = ''
    while not pila.esta_vacia():
        binario += str(pila.desapilar())

    return binario
```

En este código, la función decimal_a_binario recibe el número decimal como parámetro. Se crea una instancia de la clase Pila y se realizan divisiones sucesivas del número decimal entre 2. Los residuos se apilan en la pila. Luego, se desapilan los residuos y se concatenan en orden inverso para obtener el número binario.

COLAS

Ejercicio 13: Implementar una cola utilizando listas.

Planteamiento: Se te pide implementar una cola utilizando listas en Python.

Explicación: Para implementar una cola utilizando listas, puedes utilizar los métodos append() para encolar un elemento al final de la lista y pop(0) para desencolar el primer elemento de la lista.

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

    def encolar(self, elemento):
        self.items.append(elemento)

    def desencolar(self):
        if not self.esta_vacia():
            return self.items.pop(0)

    def esta_vacia(self):
        return len(self.items) == 0
```

En este código, la clase Cola implementa las operaciones básicas de una cola: encolar, desencolar y esta_vacia. La función encolar utiliza el método append() para agregar un elemento al final de la lista. La función desencolar utiliza el método pop(0) para eliminar y devolver el primer elemento de la lista.

Ejercicio 14: Implementar una cola con límite de capacidad.

Planteamiento: Se te pide implementar una cola que tenga un límite máximo de capacidad.

Explicación: Para implementar una cola con límite de capacidad, puedes utilizar una lista para almacenar los elementos y una variable para controlar el límite de capacidad. Al encolar un elemento, verificas si la cola ha alcanzado su capacidad máxima antes de agregar el elemento. Si la cola está llena, no se permite encolar más elementos.

In [ ]:
```python
class ColaConCapacidad:
    def __init__(self, capacidad):
        self.items = []
        self.capacidad = capacidad

    def encolar(self, elemento):
        if len(self.items) < self.capacidad:
            self.items.append(elemento)
        else:
            print("La cola ha alcanzado su capacidad máxima.")

    def desencolar(self):
        if not self.esta_vacia():
            return self.items.pop(0)

    def esta_vacia(self):
        return len(self.items) == 0
```

En este código, la clase ColaConCapacidad implementa una cola con un límite de capacidad especificado en el momento de su creación. La función encolar verifica si la cantidad de elementos en la cola es menor que la capacidad máxima antes de agregar un nuevo elemento. Si la cola está llena, se muestra un mensaje indicando que ha alcanzado su capacidad máxima.

Ejercicio 15: Invertir el orden de los elementos en una cola.

Planteamiento: Dada una cola, se te pide invertir el orden de sus elementos.

Explicación: Para invertir el orden de los elementos en una cola, puedes utilizar una pila auxiliar. Desencolas cada elemento de la cola y los apilas en la pila auxiliar. Luego, desapilas los elementos de la pila auxiliar y los encolas nuevamente en la cola. Al finalizar, la cola tendrá los elementos invertidos.

In [ ]:
```python
def invertir_cola(cola):
    pila_auxiliar = []

    while not cola.esta_vacia():
        pila_auxiliar.append(cola.desencolar())

    while pila_auxiliar:
        cola.encolar(pila_auxiliar.pop())

    return cola
```

En este código, la función invertir_cola recibe la cola original como parámetro. Se utiliza una lista pila_auxiliar como pila auxiliar para almacenar los elementos temporalmente. Se utiliza un bucle while para desencolar elementos de la cola original y apilarlos en la pila auxiliar. Luego, se desapilan los elementos de la pila auxiliar y se encolan nuevamente en la cola original. Finalmente, se devuelve la cola con los elementos invertidos.

Ejercicio 16: Verificar si una cadena de caracteres es un palíndromo utilizando una cola.

Planteamiento: Dada una cadena de caracteres, se te pide verificar si es un palíndromo utilizando una cola.

Explicación: Para resolver este ejercicio, puedes utilizar una cola. Recorres la cadena de caracteres y encolas cada carácter en la cola. Luego, desencolas los caracteres de la cola y los comparas con los caracteres de la cadena en el orden original. Si todos los caracteres coinciden, la cadena es un palíndromo.

In [ ]:
```python
def es_palindromo(cadena):
    cola = Cola()
    for caracter in cadena:
        cola.encolar(caracter)

    for caracter in cadena:
        if caracter != cola.desencolar():
            return False

    return True
```

En este código, la función es_palindromo recibe la cadena de caracteres como parámetro. Se crea una instancia de la clase Cola y se encolan los caracteres de la cadena. Luego, se recorre la cadena nuevamente y se compara cada carácter con el resultado de desencolar de la cola. Si

algún carácter no coincide, se devuelve False. Si todos los caracteres coinciden, se devuelve True.

Ejercicio 17: Implementar un sistema de atención en una tienda utilizando colas.

Planteamiento: Se te pide implementar un sistema de atención en una tienda utilizando una cola. Los clientes llegan a la tienda y se agregan a la cola de atención. El empleado de la tienda atiende a los clientes en orden de llegada.

Explicación: Para implementar este sistema, puedes utilizar la clase Cola y sus operaciones básicas. Los clientes se encolan en la cola de atención utilizando la operación encolar(). Cuando el empleado está listo para atender a un cliente, se utiliza la operación desencolar() para obtener al cliente siguiente en la cola. El empleado puede repetir este proceso hasta que no queden más clientes en la cola.

In [ ]:
```
def sistema_atencion():
    cola = Cola()
    continuar = True

    while continuar:
        opcion = input("¿Qué deseas hacer? (1: Llega un cliente, 2: Atender

        if opcion == "1":
            nombre = input("Ingresa el nombre del cliente: ")
            cola.encolar(nombre)
            print("Cliente", nombre, "agregado a la cola de atención.")

        elif opcion == "2":
            if not cola.esta_vacia():
                cliente = cola.desencolar()
                print("Atendiendo al cliente", cliente)
            else:
                print("No hay clientes en espera.")

        elif opcion == "3":
            continuar = False
            print("Saliendo del sistema de atención.")

        else:
            print("Opción inválida. Por favor, intenta nuevamente.")

sistema_atencion()
```

En este código, la función sistema_atencion implementa el sistema de atención en la tienda. Se utiliza un bucle while para mantener el sistema en funcionamiento hasta que se elija la opción de salir. Los clientes pueden llegar a la tienda agregándose a la cola de atención con la opción 1. El empleado puede atender a los clientes utilizando la opción 2, que desencola al siguiente cliente de la cola. También se manejan posibles casos, como cuando la cola está vacía o se elige una opción inválida.

Ejercicio 18: Interleaving de colas.

Planteamiento: Dadas dos colas, se te pide intercalar sus elementos en una tercera cola de forma alternada.

Explicación: Para resolver este ejercicio, puedes utilizar tres colas: una para cada cola original y otra para almacenar el resultado. Mientras ambas colas no estén vacías, desencolas un elemento de cada cola original y los encolas alternadamente en la tercera cola. Si alguna de las colas originales tiene elementos restantes después de intercalar todos los elementos

In [ ]:
```python
def interleaving_de_colas(cola1, cola2):
    cola_resultado = Cola()

    while not cola1.esta_vacia() and not cola2.esta_vacia():
        elemento1 = cola1.desencolar()
        elemento2 = cola2.desencolar()
        cola_resultado.encolar(elemento1)
        cola_resultado.encolar(elemento2)

    while not cola1.esta_vacia():
        cola_resultado.encolar(cola1.desencolar())

    while not cola2.esta_vacia():
        cola_resultado.encolar(cola2.desencolar())

    return cola_resultado
```

En este código, la función interleaving_de_colas recibe las dos colas originales como parámetros. Se crea una instancia de la clase Cola para almacenar el resultado. Se utiliza un bucle while para intercalar los elementos de las colas originales en la cola de resultado, alternando entre los elementos de cada cola. Luego, se verifica si alguna de las colas originales tiene elementos restantes y se encolan en la cola de resultado. Finalmente, se devuelve la cola de resultado.