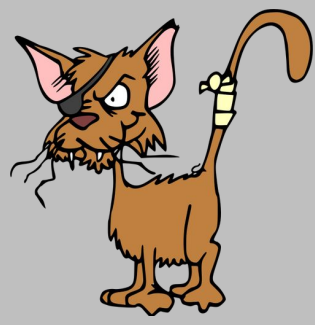
UNIVERSIDAD NACIONAL GUILLERMO BROWN

Tecnicatura Universitaria en

PROGRAMACIÓN

Pilas, colas , listas enlazadas , algoritmos de búsqueda y ordenamiento



Pilas y Colas

Las pilas y colas son estructuras de datos que se utilizan para gestionar elementos en una orden específica. A continuación, te explico cómo implementarlas en Python.

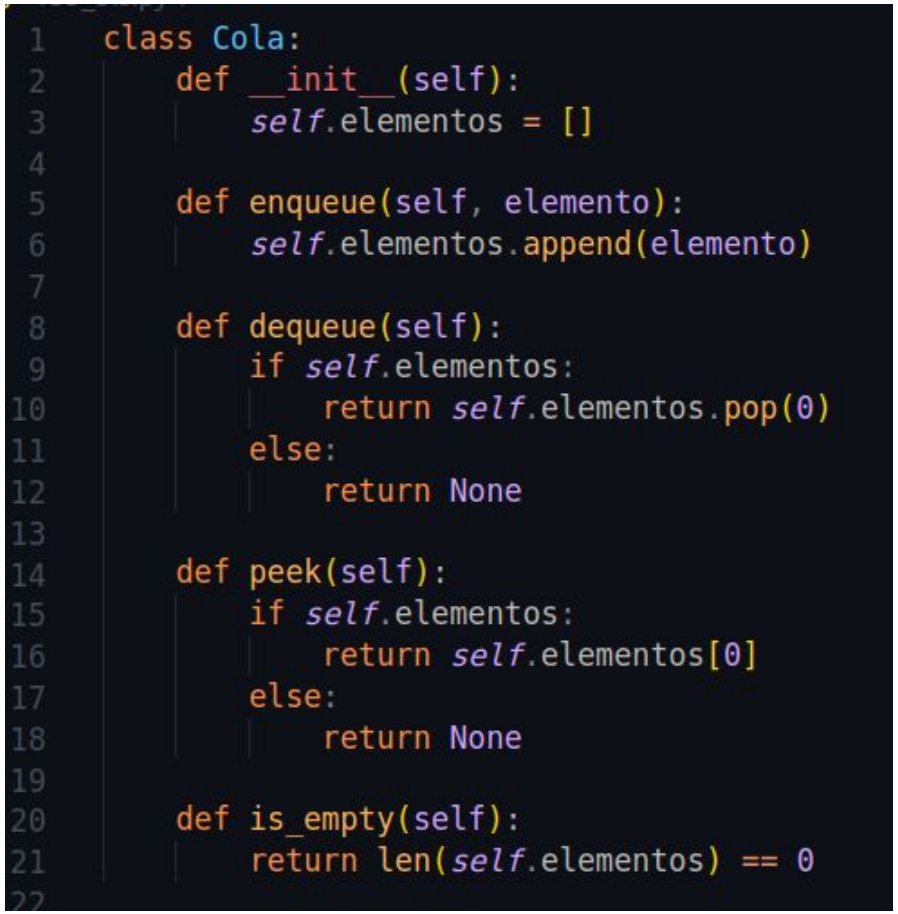
```python
class Pila:
    def __init__(self):
        self.elementos = []

    def push(self, elemento):
        self.elementos.append(elemento)

    def pop(self):
        if self.elementos:
            return self.elementos.pop()
        else:
            return None

    def peek(self):
        if self.elementos:
            return self.elementos[-1]
        else:
            return None

    def is_empty(self):
        return len(self.elementos) == 0

```

Pilas (Stacks)

Una pila es una estructura de datos que sigue el principio de "LIFO" (Last In, First Out), lo que significa que el elemento agregado más recientemente es el primero en ser eliminado. En Python, se puede implementar una pila utilizando una lista y métodos para agregar y eliminar elementos.

Implementación de una Pila en Python

Para implementar una pila en Python, puedes utilizar una lista y definir métodos para agregar y eliminar elementos. Aquí te muestro un ejemplo:



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

    def enqueue(self, elemento):
        self.elementos.append(elemento)

    def dequeue(self):
        if self.elementos:
            return self.elementos.pop(0)
        else:
            return None

    def peek(self):
        if self.elementos:
            return self.elementos[0]
        else:
            return None

    def is_empty(self):
        return len(self.elementos) == 0
```

Colas (Queues)

Una cola es una estructura de datos que sigue el principio de "FIFO" (First In, First Out), lo que significa que el elemento agregado primero es el primero en ser eliminado. En Python, se puede implementar una cola utilizando una lista y métodos para agregar y eliminar elementos.

Implementación de una Cola en Python

Para implementar una cola en Python, puedes utilizar una lista y definir métodos para agregar y eliminar elementos. Aquí te muestro un ejemplo:

enqueue:

La operación enqueue se utiliza para agregar un elemento al final de la cola.

Cuando se realiza una operación enqueue, el elemento se agrega al final de la cola.

La implementación típica de enqueue en Python utiliza el método append() de la lista para agregar el elemento al final de la cola.

dequeue:

La operación dequeue se utiliza para eliminar y devolver el elemento del frente de la cola.

Cuando se realiza una operación dequeue, el elemento del frente de la cola se elimina y se devuelve.

La implementación típica de dequeue en Python utiliza el método pop(0) de la lista para eliminar y devolver el primer elemento de la cola.

enqueue y dequeue no son palabras reservadas o comandos integrados en el lenguaje Python. Son términos generales utilizados en la teoría de estructuras de datos para describir las operaciones de agregar y eliminar elementos de una cola.

Ventajas y Desventajas

Ambas estructuras de datos tienen ventajas y desventajas:

Ventajas de las pilas:

Facilitan la implementación de algoritmos que requieren acceder a los elementos más recientes.

Son más eficientes para operaciones de inserción y eliminación en el final.

Desventajas de las pilas:

No permiten acceder directamente a elementos específicos.

No son adecuadas para operaciones que requieren acceder a elementos en una posición específica.

Ventajas y Desventajas

Ambas estructuras de datos tienen ventajas y desventajas:

Ventajas de las colas:

Permiten acceder directamente a los elementos más antiguos.

Son adecuadas para operaciones que requieren acceder a elementos en una posición específica.

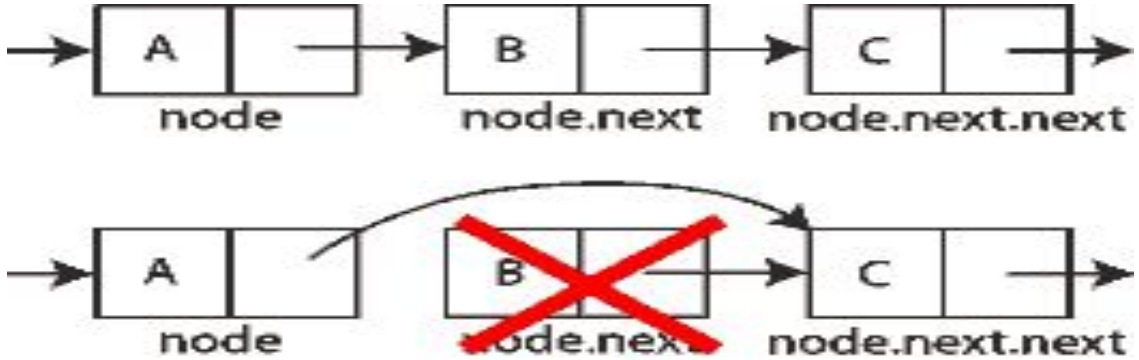
Desventajas de las colas:

No son tan eficientes para operaciones de inserción y eliminación en el final.

Requieren más memoria para almacenar los elementos.

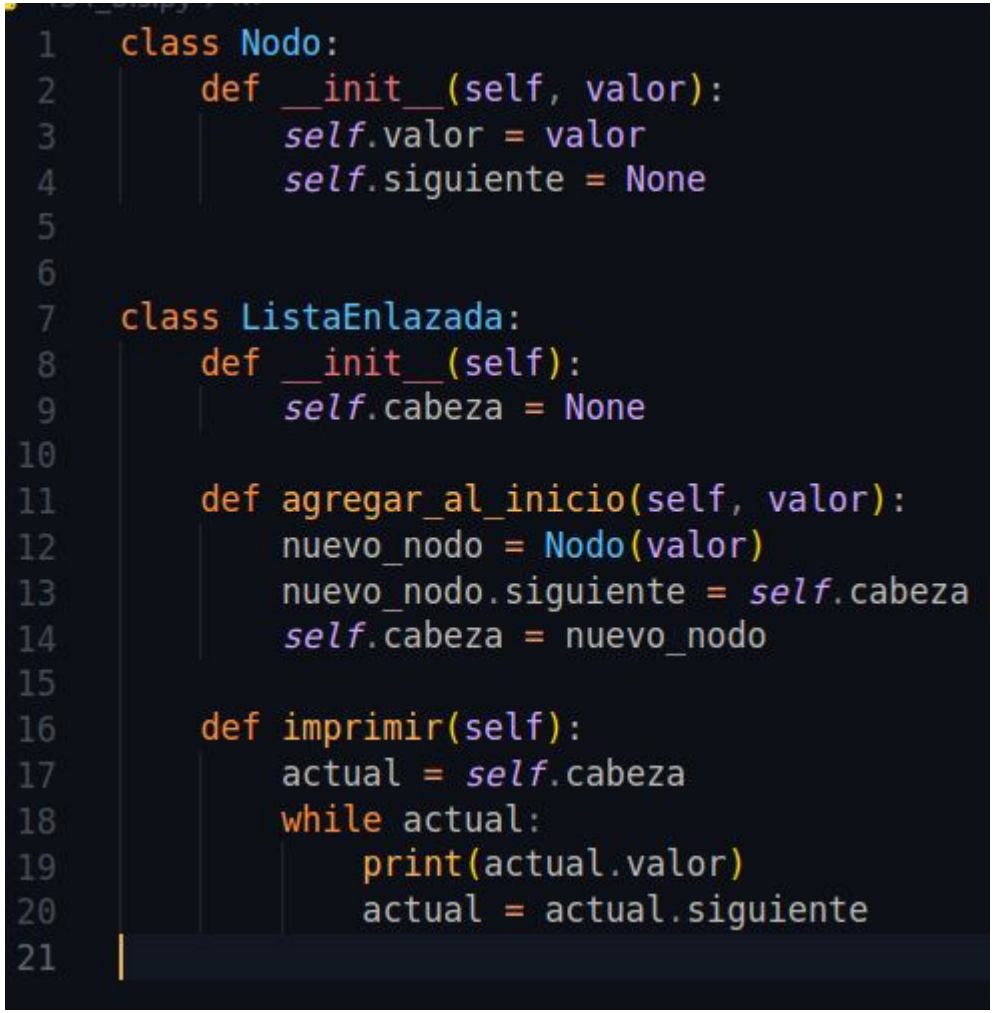
En resumen, las pilas son adecuadas para operaciones que requieren acceder a los elementos más recientes, mientras que las colas son adecuadas para operaciones que requieren acceder a los elementos más antiguos.

Listas Enlazadas



Las listas enlazadas en Python son estructuras de datos dinámicas que permiten almacenar y manipular elementos de manera eficiente.

Una lista enlazada es una estructura de datos que se compone de nodos, cada uno de los cuales contiene un valor y un puntero al siguiente nodo en la lista. Esta estructura permite agregar o eliminar elementos en cualquier posición de la lista sin necesidad de reordenar todos los elementos, lo que la hace especialmente útil para aplicaciones que requieren operaciones de inserción y eliminación frecuentes.



```python
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.siguiente = None


class ListaEnlazada:
    def __init__(self):
        self.cabeza = None

    def agregar_al_inicio(self, valor):
        nuevo_nodo = Nodo(valor)
        nuevo_nodo.siguiente = self.cabeza
        self.cabeza = nuevo_nodo

    def imprimir(self):
        actual = self.cabeza
        while actual:
            print(actual.valor)
            actual = actual.siguiente
```

Implementación de una lista enlazada en Python

Clase de nodo: Primero, se define una clase Nodo que contiene un valor y un puntero al siguiente nodo en la lista.

Clase de lista enlazada: Luego, se define una clase LinkedList que contiene una lista de nodos y métodos para insertar, eliminar y buscar elementos.

Ventajas y desventajas

Ventajas:

Dinámico: La cantidad de nodos en una lista enlazada no es fija y puede crecer o disminuir según sea necesario.

Eficiente: Las operaciones de inserción y eliminación son más rápidas que en matrices, ya que no se requiere reordenar todos los elementos.

Flexibilidad: Las listas enlazadas se pueden utilizar para implementar estructuras de datos como pilas y colas.

Ventajas y desventajas

Desventajas:

Memoria: Las listas enlazadas utilizan más memoria que las matrices debido al almacenamiento de los punteros.

Acceso secuencial: Los nodos en una lista enlazada deben leerse en orden desde el principio, lo que puede ser un problema en aplicaciones que requieren acceso directo a elementos individuales.

Dificultades en la reversa: Las listas enlazadas pueden ser engorrosas para navegar hacia atrás, especialmente si no se utiliza una lista doblemente enlazada.

Algoritmos de búsqueda

```python
def busqueda_lineal(lista, elemento):
    for i in range(len(lista)):
        if lista[i] == elemento:
            return i
    return -1
```

Búsqueda Lineal

La búsqueda lineal es el algoritmo más simple y básico.

Recorre la lista de elementos uno por uno, comparando cada elemento con el valor buscado.

Si encuentra el elemento, devuelve su posición. Si llega al final sin encontrarlo, devuelve -1.

Tiene una complejidad temporal O(n) en el peor caso.

Búsqueda Binaria

```python
def busqueda_binaria(lista, elemento):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == elemento:
            return medio
        elif lista[medio] < elemento:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1
```

La búsqueda binaria es más eficiente que la lineal.

Requiere que la lista esté ordenada previamente.

Compara el elemento buscado con el elemento del medio de la lista.

Si es menor, busca en la mitad inferior. Si es mayor, busca en la mitad superior.

Divide repetidamente la lista hasta encontrar el elemento o hasta que no queden más elementos.

Tiene una complejidad temporal O(log n) en el peor caso.

Búsqueda por Saltos (Jump Search)

```python
import math


def busqueda_por_saltos(lista, elemento):
    n = len(lista)
    paso = int(math.floor(math.sqrt(n)))
    bloque_anterior = 0
    while lista[min(paso, n) - 1] < elemento:
        bloque_anterior = paso
        paso += int(math.floor(math.sqrt(n)))
        if bloque_anterior >= n:
            return -1
    while lista[bloque_anterior] < elemento:
        bloque_anterior += 1
        if bloque_anterior == min(paso, n):
            return -1
    if lista[bloque_anterior] == elemento:
        return bloque_anterior
    return -1

```

.Combina características de la búsqueda lineal y binaria.

Divide la lista en bloques de tamaño √n.

Busca en qué bloque puede estar el elemento y luego hace una búsqueda lineal dentro de ese bloque.

Tiene una complejidad temporal O(√n) en el peor caso

Algoritmos de ordenamiento

Bubble Sort (Ordenamiento por Burbuja)

```
4
5   lista = []
6   for x in range(5):
7       sueldo = int(input("Ingrese sueldo :"))
8       lista.append(sueldo)
9   print(lista)
10
11  for x in range(4):
12      if lista[x] > lista[x+1]:
13          aux = lista[x]
14          lista[x] = lista[x+1]
15          lista[x+1] = aux
16
17
18  print("Lista ordenada)")
19  print(lista)
20
```

.El algoritmo de ordenamiento por burbuja es un método simple para ordenar una lista. Funciona comparando cada par de elementos y intercambiandolos si están en orden incorrecto. Esto se repite hasta que la lista esté completamente ordenada.

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)


def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result


# Ejemplo de uso
arr = [8, 2, 6, 4, 5]
print(merge_sort(arr))  # Output: [2, 4, 5, 6, 8]
```

Merge Sort (Ordenamiento por Mezcla)

.Es un algoritmo que divide la lista en dos mitades y las ordena recursivamente. Luego, fusiona las dos mitades ordenadas en una lista ordenada

¿PREGUNTAS?