Lógica de programación I

Juan Pablo Restrepo Uribe

Ing. Biomedico

MSc. Automatización y Control Industrial

jprestrepo@correo.iue.edu.co

2023-2

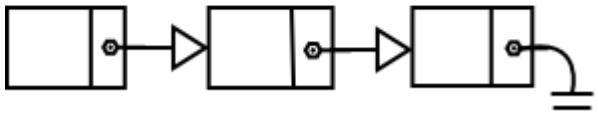
Institución Universitaria de Envigado

Nodos

Un nodo es un elemento básico que se utiliza para construir diversas estructuras de datos, como listas enlazadas, árboles, grafos, entre otros.

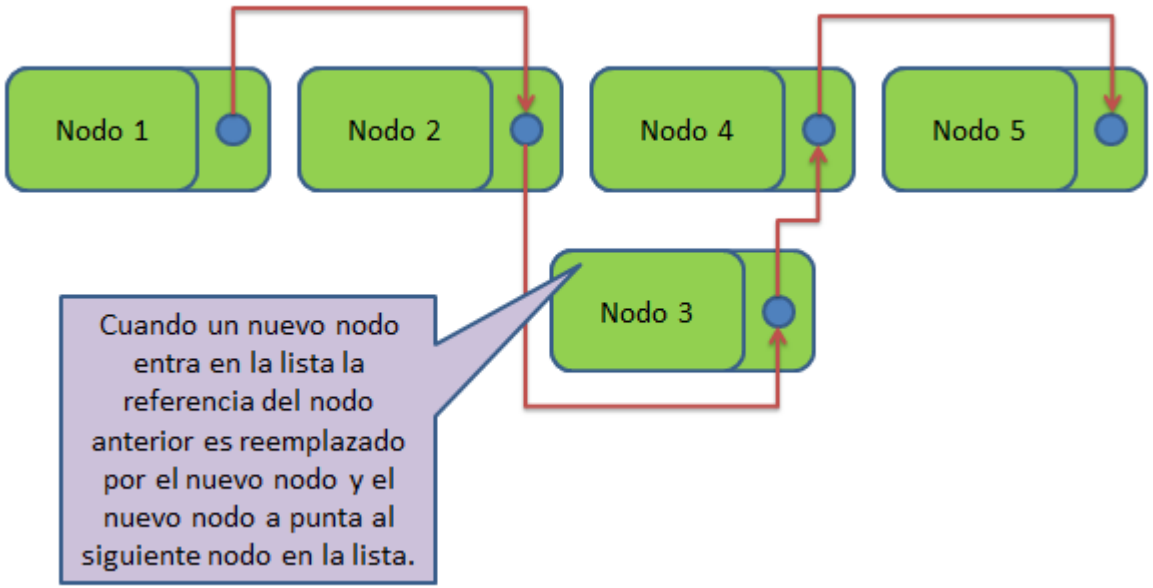
Un nodo generalmente contiene dos partes principales: los datos que se almacenan y una referencia (o puntero) que apunta al siguiente nodo en una estructura enlazada.

Estructura de un nodo



Lista enlazada

Nodos


Nodo 1
Nodo 2
Nodo 4
Nodo 5
Nodo 3
Cuando un nuevo nodo entra en la lista la referencia del nodo anterior es reemplazado por el nuevo nodo y el nuevo nodo a punta al siguiente nodo en la lista.

Nodos

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

# Ejemplo de uso
nodo1 = Nodo(10)
nodo2 = Nodo(20)
nodo3 = Nodo(30)

nodo1.siguiente = nodo2
nodo2.siguiente = nodo3

nodo_actual = nodo1
while nodo_actual is not None:
    print(nodo_actual.dato)
    nodo_actual = nodo_actual.siguiente

```

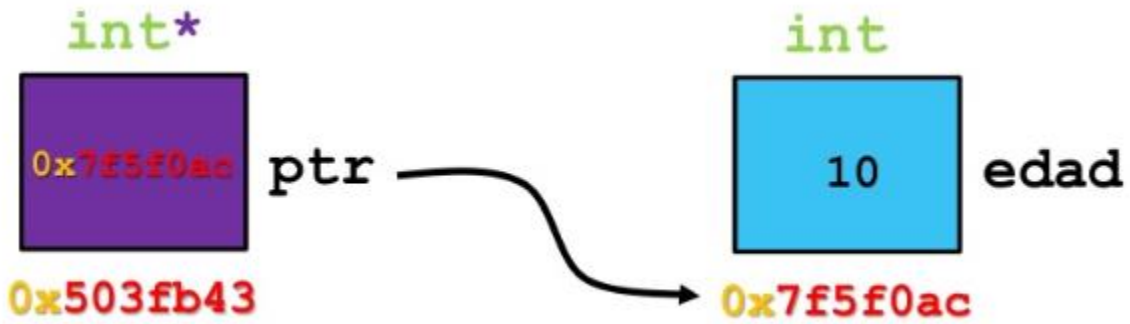
```
10
20
30
```

- Se define una clase llamada **Nodo**.
- El método **__init__** se utiliza como el constructor de la clase.
- El atributo **self.dato** almacena el dato
- El atributo **self.siguiente** es un puntero que apunta al siguiente nodo en la lista.

Nodos - Puntero

Un puntero es una variable que almacena la dirección de memoria de otra variable en un programa informático. En lugar de contener directamente datos, un puntero contiene la ubicación de memoria de donde se pueden recuperar o almacenar esos datos.

Un puntero "apunta" a una ubicación específica en la memoria del sistema.



Nodos - Puntero

```c
#include <stdio.h>

int main() {
    int num = 10; // Declaramos e inicializamos una variable entera
    int *ptr;     // Declaramos un puntero a entero

    ptr = &num;   // Asignamos la dirección de memoria de 'num' al puntero 'ptr'

    // Imprimimos el valor de 'num' y la dirección de memoria donde está almacenado
    printf("Valor de num: %d\n", num);
    printf("Dirección de memoria de num: %p\n", (void*)&num);

    // Imprimimos el valor apuntado por el puntero 'ptr' y la dirección de memoria que almacena
    printf("Valor apuntado por ptr: %d\n", *ptr);
    printf("Dirección almacenada en ptr: %p\n", (void*)ptr);

    return 0;
}
```

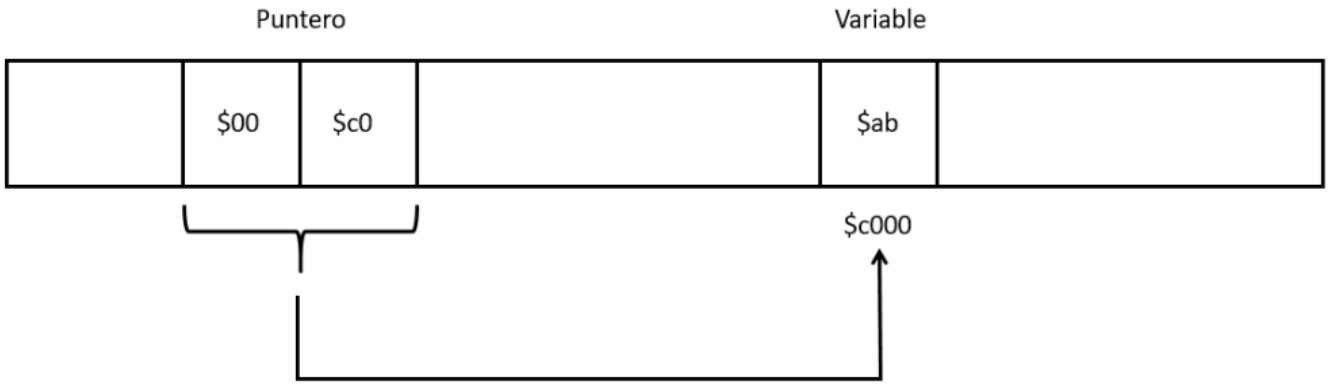
```
Valor de num: 10
Dirección de memoria de num: 0x7ffdea79f09c
Valor apuntado por ptr: 10
Dirección almacenada en ptr: 0x7ffdea79f09c
```

Nodos - Puntero

- **Gestión eficiente de la memoria:** Los punteros permiten trabajar directamente con la memoria del sistema, lo que permite una gestión más eficiente de los recursos.
- **Acceso a datos indirectos:** Los punteros permiten acceder a los datos indirectamente, lo que significa que puedes acceder a un dato a través de su dirección de memoria en lugar de copiarlo directamente.
- **Flexibilidad en estructuras de datos:** Los punteros facilitan la implementación de estructuras de datos dinámicas como listas enlazadas, árboles y grafos
- **Manipulación de direcciones de memoria:** Los punteros permiten manipular directamente las direcciones de memoria.

Nodos - Puntero

En Python, los punteros no funcionan de la misma manera que en lenguajes de programación como C o C++, donde los punteros son variables que almacenan direcciones de memoria. Python utiliza referencias en lugar de punteros. Las referencias son como punteros, pero con una capa de abstracción adicional que oculta la manipulación directa de direcciones de memoria.



Nodos - Puntero

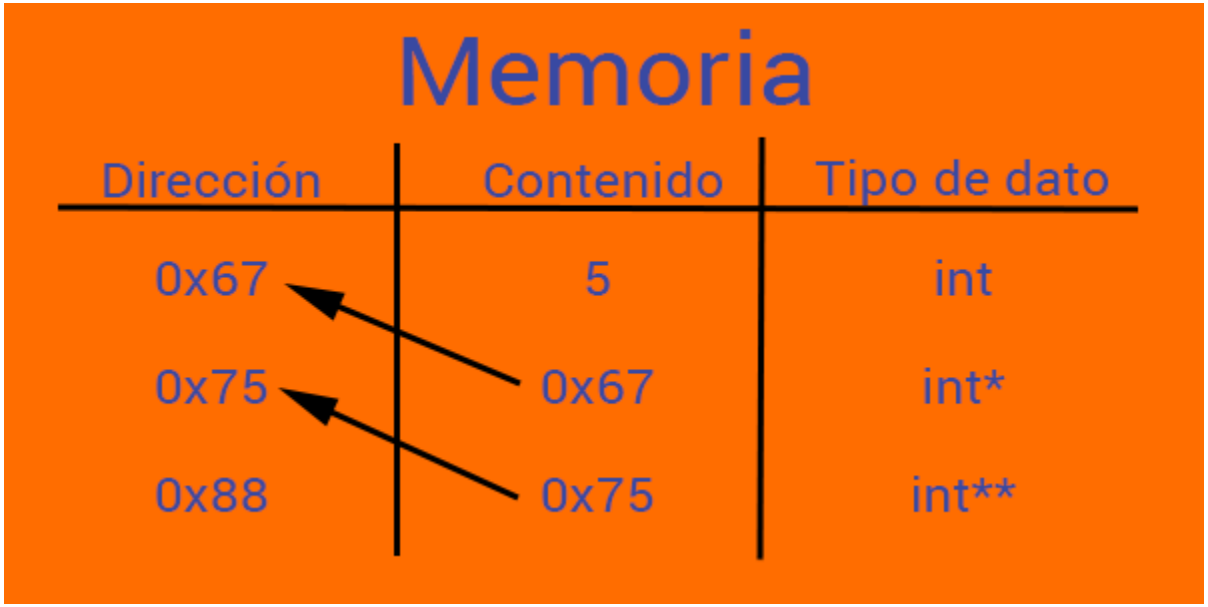
En Python, los punteros no funcionan de la misma manera que en lenguajes de programación como C o C++, donde los punteros son variables que almacenan direcciones de memoria. Python utiliza referencias en lugar de punteros. Las referencias son como punteros, pero con una capa de abstracción adicional que oculta la manipulación directa de direcciones de memoria.

```
[ ]   1 x = 5
      2 y = x
```

```
1 lista1 = [1, 2, 3]
2 lista2 = lista1
```

Nodos – Puntero (python)

Su ausencia se debe a que el uso explícito de punteros es una característica de lenguajes de más bajo nivel como el C. Lenguajes de alto nivel como Python lo evitan con el propósito de hacer más fácil y ágil su utilización, así como no tener que conocer detalles del modelo de datos.



Nodos – Puntero (python)

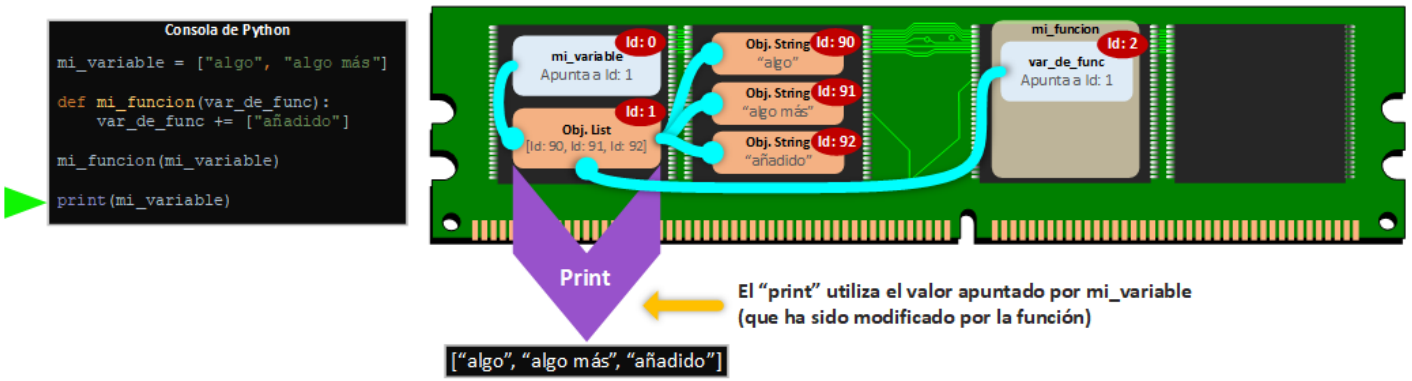
Que el programador de Python no tenga que lidiar con los punteros no quiere decir que el intérprete no haga uso de ellos. De hecho, los usa profusamente de forma implícita.

En Python todo es un objeto creado en la memoria dinámica. Cuando llamas a una función los argumentos son pasados mediante sus punteros.

Si asignas a = b, a lo que guarda es el puntero del objeto de b. Así que todas las variables son punteros a objetos, que son manejados implícitamente.

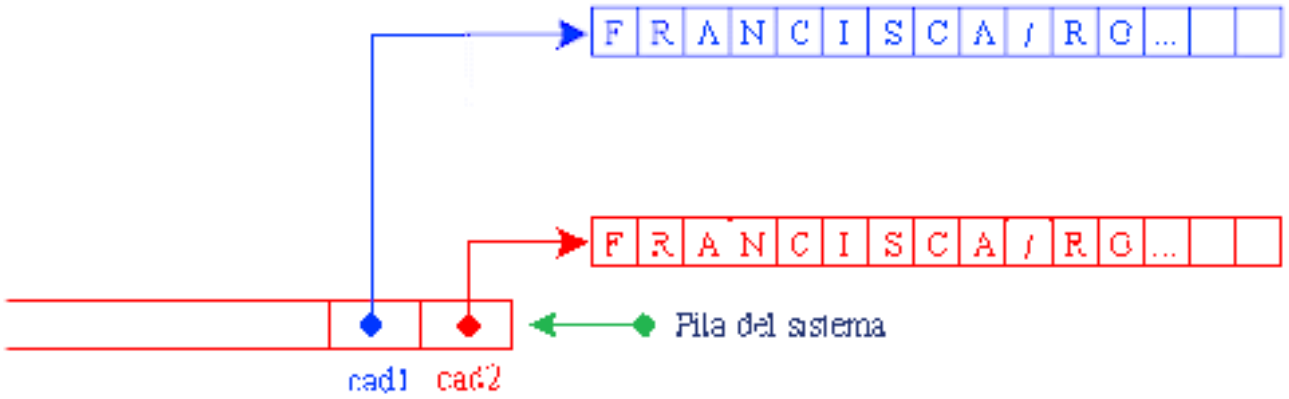
Nodos – Puntero (python)

- **Objetos inmutables:** son los números, las cadenas o las tuplas. Al asignar x = 2015 creará un objeto entero y x apuntará a él, pero el contenido de ese objeto no podrá ser modificado. Si luego asignas x = 2016 lo que hará internamente será crear un nuevo objeto con el nuevo contenido.
- **Objetos mutables:** son otros objetos como los diccionarios o las listas. En este caso dichos objetos sí podrán ser modificados. Si tienes v = [1] y luego llamas a v.append(2), v seguirá apuntando al mismo objeto, pero su contenido habrá cambiado.

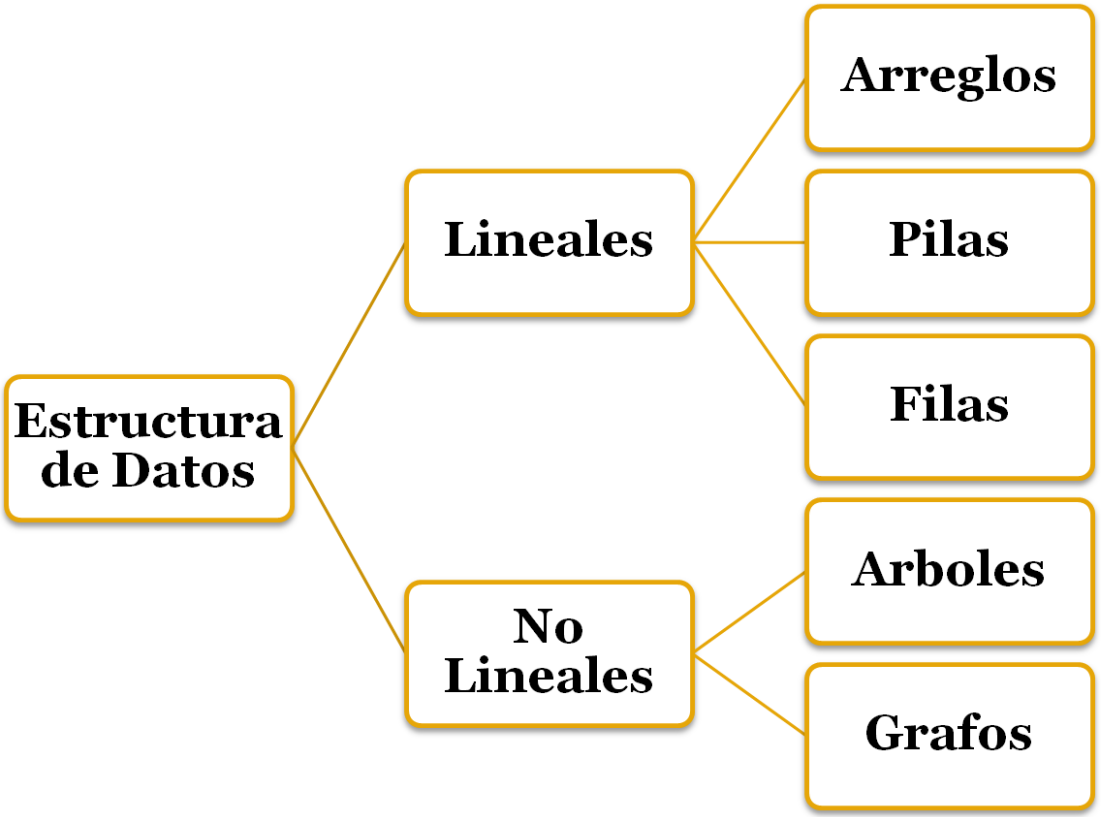


Memoria dinámica

Se refiere a aquella memoria que no puede ser definida ya que no se conoce o no se tiene idea del número de la variable a considerarse, la solución a este problema es la memoria dinámica que permite solicitar memoria en tiempo de ejecución, por lo que cuanta más memoria se necesite, más se solicita al sistema operativo.

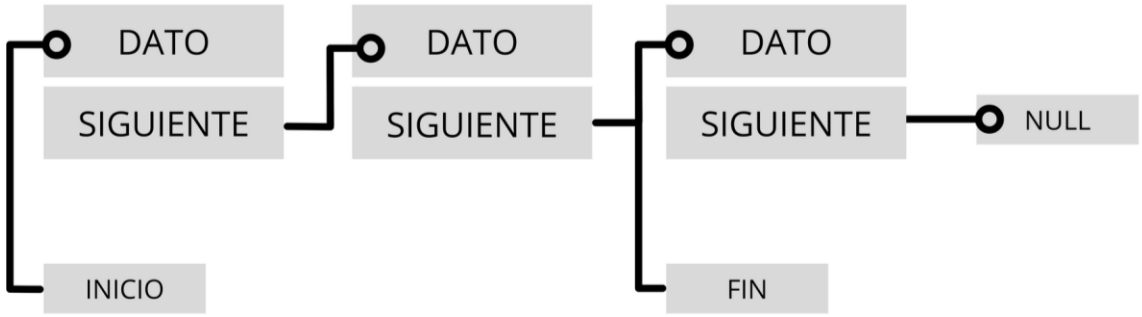


Nodos


Estructura de Datos
Lineales
No Lineales
Arreglos
Pilas
Filas
Arboles
Grafos

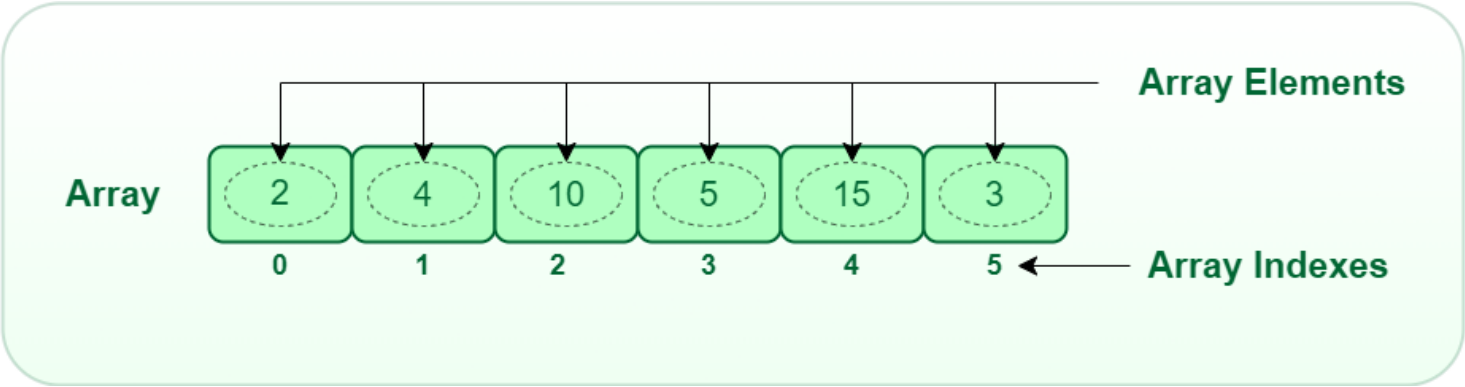
Estructuras Lineales

Las estructuras lineales son aquellas en las que los elementos están organizados de manera secuencial, es decir, uno tras otro, formando una línea o secuencia. No hay bifurcaciones en la estructura; cada elemento tiene exactamente un sucesor y, en el caso de la lista, uno o ningún predecesor.



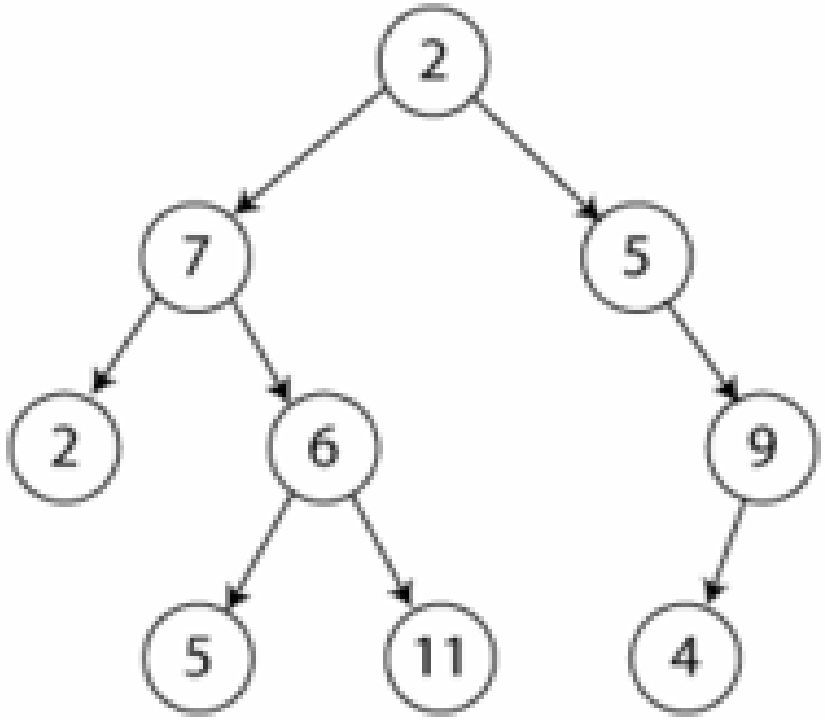
Estructuras Lineales

- Lista Enlazada (Simple o Doble)
- Pila (Stack)
- Cola (Queue)
- Vector (o Array)



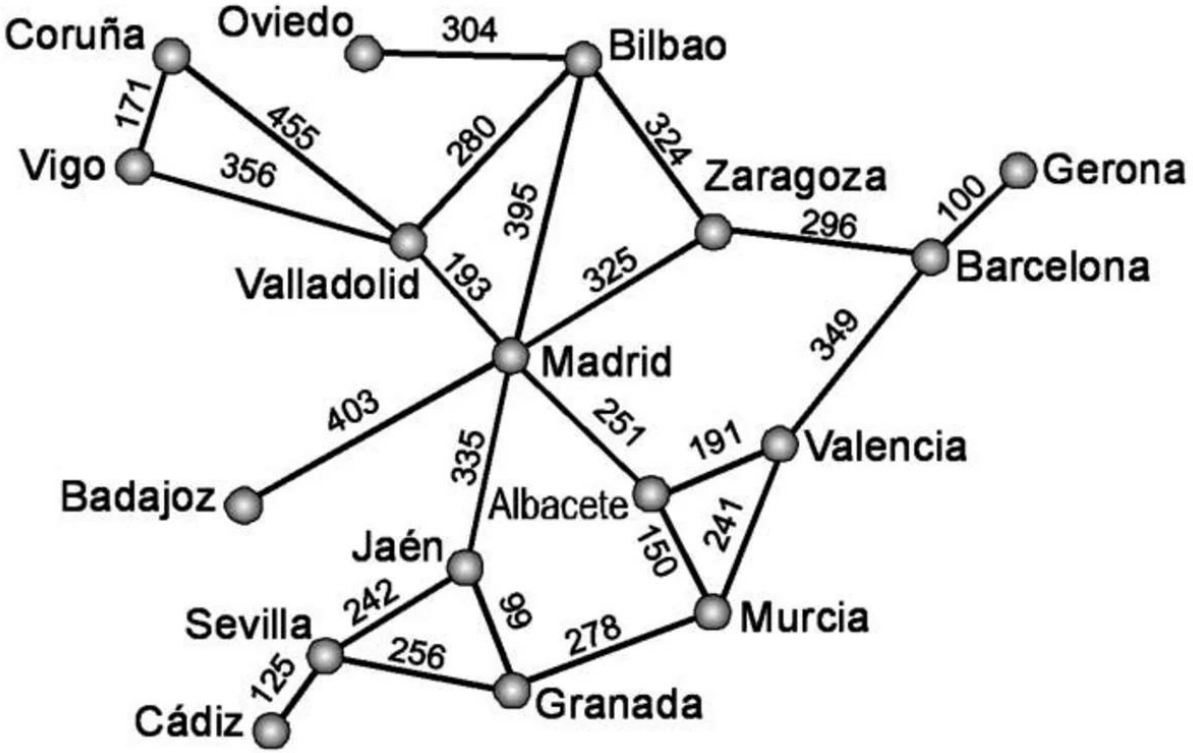
Estructuras No Lineales

Son aquellas en las que los elementos no están organizados secuencialmente, es decir, no siguen una estructura lineal como una lista, una pila o una cola. En cambio, los elementos en estructuras de datos no lineales pueden estar conectados de varias maneras, formando estructuras más complejas como árboles o grafos.



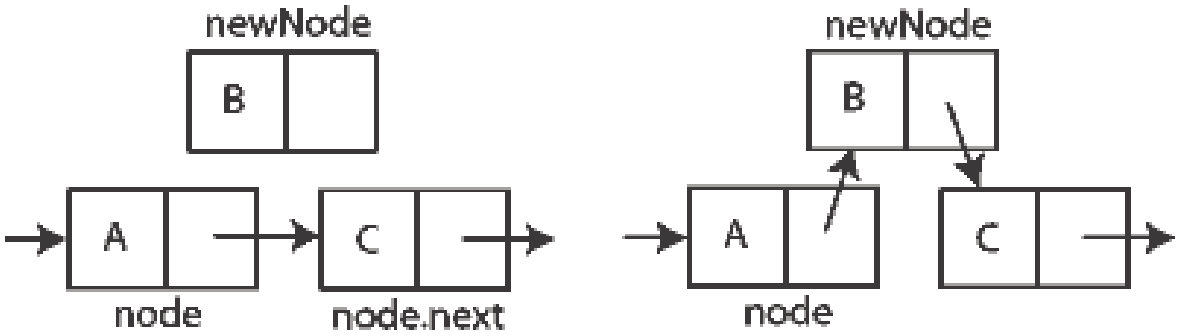
Estructuras No Lineales

- Árboles
- Grafos



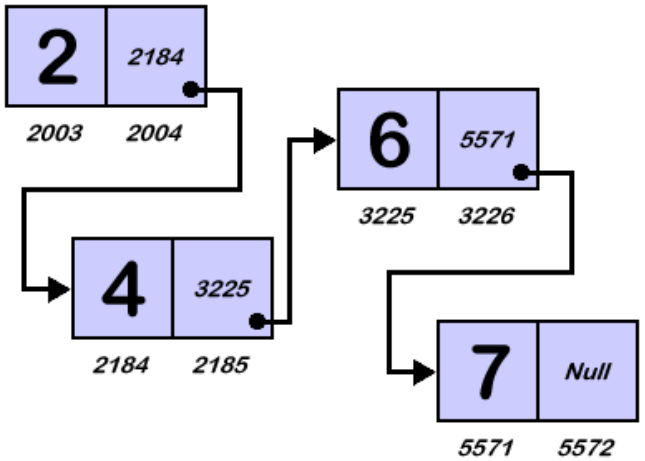
Listas enlazadas

Una lista enlazada es una de las estructuras de datos fundamentales, y puede ser usada para implementar otras estructuras de datos. Consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros al nodo anterior o posterior.



Listas enlazadas

Las listas enlazadas se encuentran entre las estructuras de datos más simples y comunes. Se pueden usar para implementar muchos otros tipos de datos abstractos comunes , incluyendo listas , pilas , colas , matrices asociativas y expresiones S , aunque no es raro implementar esas estructuras de datos directamente sin utilizar una lista vinculada como base.



Listas enlazadas

Ventajas

- Estructura de datos dinámica: LA lista puede crecer y reducirse en tiempo de ejecución mediante la asignación y des-asignación de memoria. **No hay necesidad de dar el tamaño inicial.**
- Sin desperdicio de memoria: en la lista vinculada, se puede lograr una utilización eficiente de la memoria ya que el tamaño de la lista vinculada aumenta o disminuye en el tiempo de ejecución.
- Operaciones de inserción y eliminación: Las operaciones de inserción y eliminación son bastante más fáciles en la lista enlazada. No es necesario cambiar elementos después de la inserción o eliminación de un elemento, solo se debe actualizar la dirección presente en el siguiente puntero.

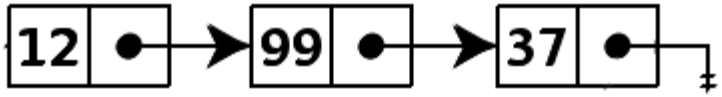
Listas enlazadas

Desventajas

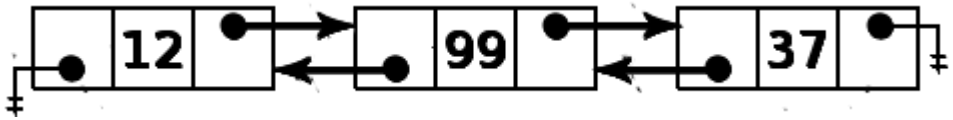
- Utilizan más memoria que las matrices debido al almacenamiento utilizado por sus punteros.
- Los nodos en una lista vinculada deben leerse en orden desde el principio, ya que las listas vinculadas son intrínsecamente de acceso secuencial .
- Los nodos se almacenan de forma incontinua, lo que aumenta en gran medida los períodos de tiempo necesarios para acceder a elementos individuales dentro de la lista.

Tipos de listas enlazadas

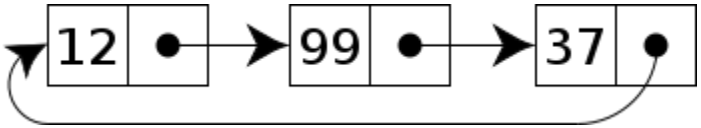
Listas enlazadas simples



Listas doblemente enlazadas

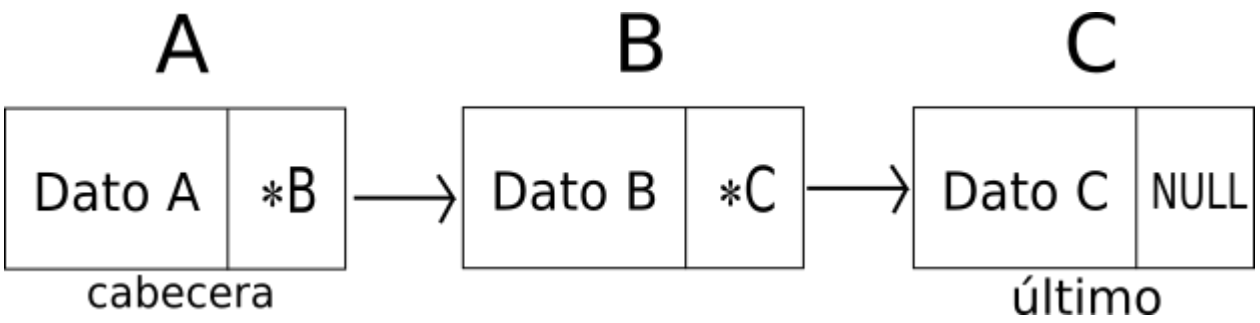


Listas enlazadas circulares



Listas simples

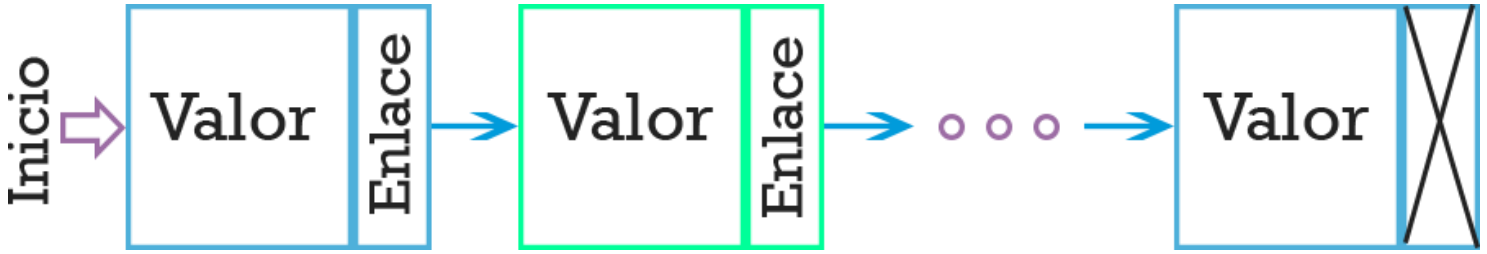
Una lista simple, también conocida como lista enlazada simple, es una estructura de datos que consta de una secuencia de nodos, donde cada nodo contiene un elemento de datos y un enlace (o puntero) al siguiente nodo en la secuencia.



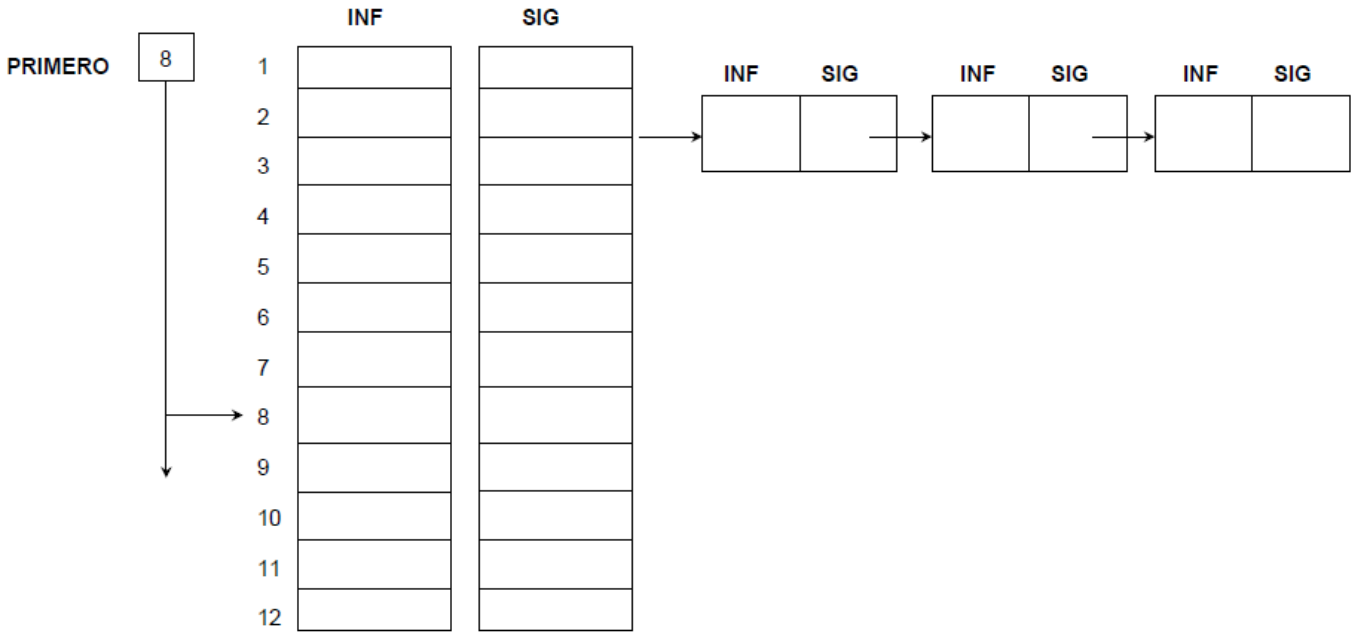
Listas simples

Es una estructura de datos que consta de una secuencia de nodos, donde cada nodo contiene un elemento de datos y un enlace (o puntero) al siguiente nodo en la secuencia.

En una lista simple, los nodos están conectados de forma secuencial, lo que significa que cada nodo apunta al siguiente nodo en la lista, formando una cadena lineal. El primer nodo de la lista se llama "cabeza" (head) y el último nodo puede tener un puntero nulo, indicando el final de la lista.



Listas simples



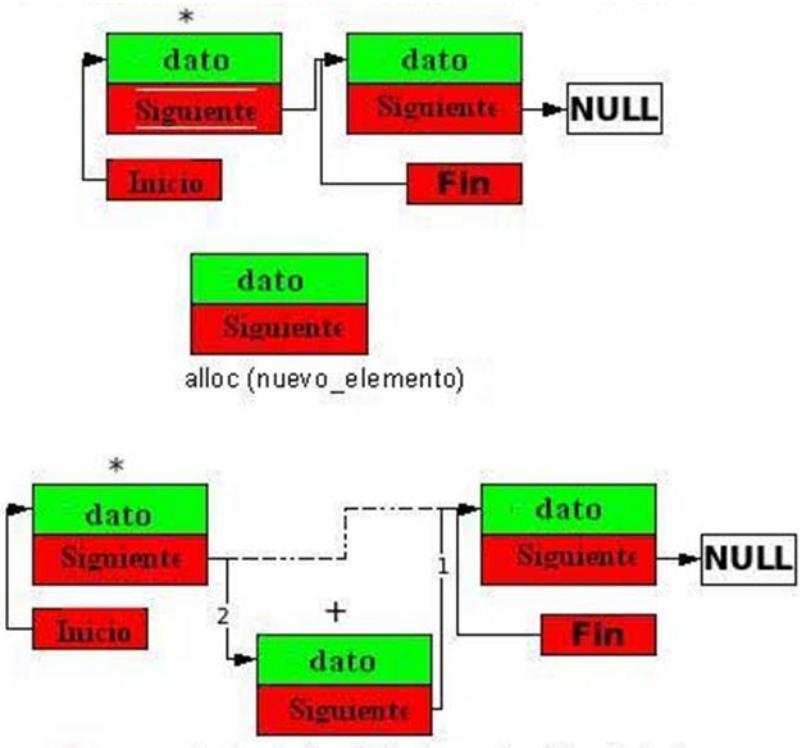
Listas simples

Las operaciones que m-+anejaremos con esta lista enlazada son:

- Agregar al inicio
- Agregar al final
- Saber si elemento existe
- Eliminar un elemento
- Obtener cabeza
- Obtener cola
- Recorrer lista

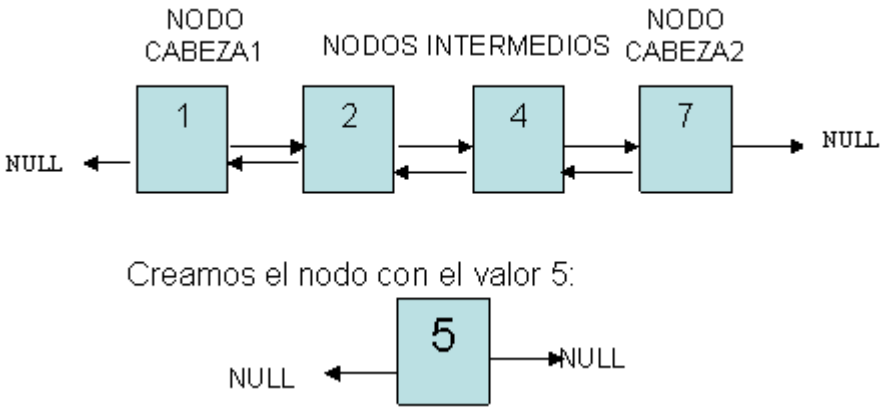
Inserción

Es una operación fundamental en estructuras de datos y se utiliza para agregar nuevos elementos a una estructura de datos existente en una posición específica.



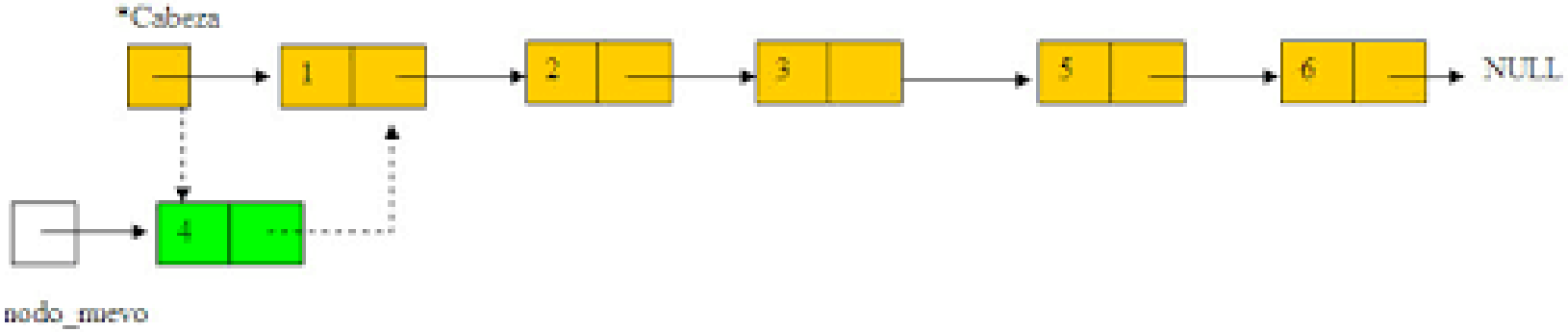
Inserción

Para insertar un nuevo elemento en una lista simple, primero se crea un nuevo nodo que contiene el valor que se desea insertar. Luego, se ajustan los enlaces para incluir este nuevo nodo en la lista. Dependiendo de la posición en la que se quiera insertar el nuevo elemento, los pasos pueden variar ligeramente:



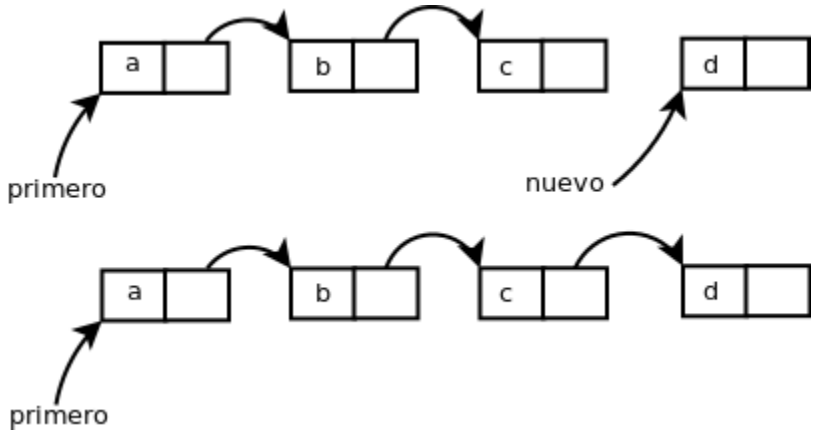
Inserción al principio de la lista

Si se quiere insertar un nuevo elemento al principio de la lista, el nuevo nodo se convierte en el primer nodo de la lista y su enlace apunta al nodo que anteriormente era el primero.



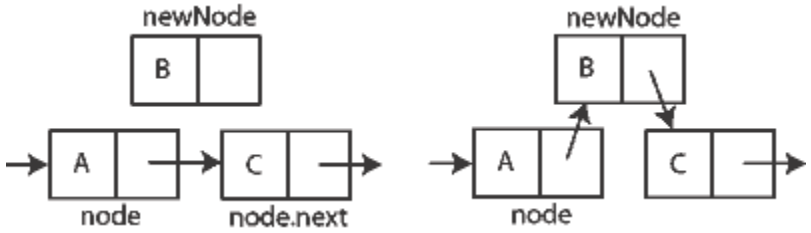
Inserción al final de la lista

Si se quiere insertar un nuevo elemento al final de la lista, se recorre la lista hasta llegar al último nodo y se ajustan los enlaces para que el nuevo nodo se convierta en el nuevo último nodo.



Inserción en el medio de la lista

Si se quiere insertar un nuevo elemento en alguna posición intermedia de la lista, se busca el nodo que actualmente ocupa esa posición y se ajustan los enlaces para que el nuevo nodo se inserte entre ese nodo y el siguiente.



Inserción: Ventajas

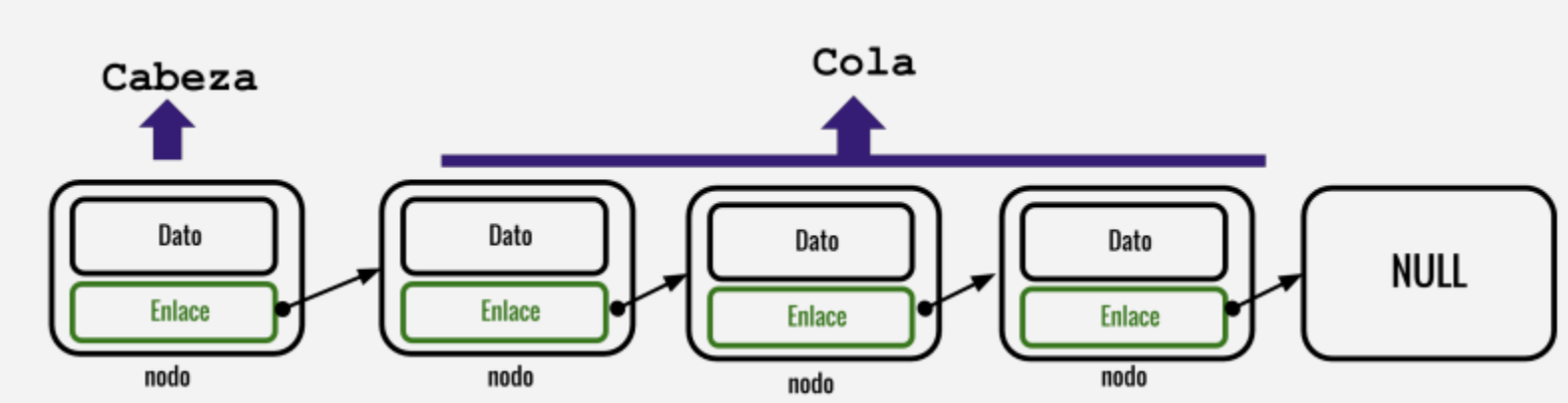
- **Flexibilidad en la modificación de datos:** Las listas permiten insertar y eliminar elementos en cualquier posición de manera eficiente. Esto es útil cuando se necesita manipular datos de forma dinámica, ya que no es necesario reorganizar la estructura completa de datos para realizar una inserción.
- **Eficiencia en la inserción y eliminación:** En comparación con estructuras como los arrays estáticos, donde agregar o eliminar elementos puede requerir reubicar todos los elementos adyacentes, las listas simplifican este proceso al solo requerir la actualización de los enlaces entre los nodos.
- **Reducción de la fragmentación de la memoria:** En estructuras de datos como los arrays, las inserciones y eliminaciones pueden llevar a la fragmentación de la memoria, especialmente si se necesita expandir o contraer el array. Las listas enlazadas no sufren de este problema, ya que cada elemento puede ubicarse en cualquier lugar de la memoria y solo requiere un enlace para acceder a los siguientes elementos.

Inserción: Ventajas

- **Capacidad de manejar tamaños dinámicos de datos:** Las listas permiten agregar elementos sin preocuparse por el tamaño máximo predefinido. Esto es especialmente útil cuando no se conoce de antemano la cantidad de elementos que se necesitarán almacenar.
- **Facilidad para implementar otras estructuras de datos:** Las listas enlazadas se utilizan comúnmente como base para implementar otras estructuras de datos más complejas, como pilas, colas o listas doblemente enlazadas. La flexibilidad en las operaciones de inserción y eliminación facilita la implementación de estas estructuras sobre una lista.

Cabeza

Se refiere al primer nodo de la lista. En una lista enlazada, cada nodo contiene algún tipo de dato y un enlace (o puntero) al siguiente nodo en la secuencia. La cabeza de la lista es el punto de entrada principal para acceder a los elementos de la lista, ya que proporciona el inicio de la secuencia de nodos enlazados.



Cabeza

La cabeza de la lista es crucial para realizar operaciones como:

- **Inserción**
- **Eliminación**
- **Búsqueda de elementos en la lista**

Si se pierde la referencia a la cabeza de la lista, se perderá el acceso a todos los elementos de la lista, ya que no habrá forma de acceder al resto de los nodos enlazados.

Cabeza

La cabeza de una lista enlazada se representa típicamente mediante una variable que contiene una referencia al primer nodo de la lista.

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

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

    def insertar_al_principio(self, dato):
        nuevo_nodo = Nodo(dato)
        nuevo_nodo.siguiente = self.cabeza
        self.cabeza = nuevo_nodo

# Ejemplo de uso
lista = ListaEnlazada()
lista.insertar_al_principio(5)
```

insertar_al_principio

- **nuevo_nodo = Nodo(dato):** Se crea un nuevo nodo utilizando la clase Nodo con el dato que se proporciona como argumento al método.
- **nuevo_nodo.siguiente = self.cabeza:** El siguiente del nuevo nodo se establece como la cabeza actual de la lista (self.cabeza). Esto significa que el nuevo nodo apunta al nodo que actualmente es el primero en la lista (o None si la lista está vacía).
- self.cabeza = nuevo_nodo: La cabeza de la lista (self.cabeza) se actualiza para ser el nuevo nodo creado. Esto significa que el nuevo nodo ahora se convierte en el primer nodo de la lista, ya que la cabeza es una referencia al primer nodo.

```
def insertar_al_principio(self, dato):
    nuevo_nodo = Nodo(dato)
    nuevo_nodo.siguiente = self.cabeza
    self.cabeza = nuevo_nodo
```

insertar_al_final

```
def insertar_al_final(self, dato):
    nuevo_nodo = Nodo(dato)
    if self.cabeza is None:
        self.cabeza = nuevo_nodo
        return

    actual = self.cabeza
    while actual.siguiente is not None:
        actual = actual.siguiente

    actual.siguiente = nuevo_nodo
```

insertar_al_final

- **nuevo_nodo = Nodo(dato):** Se crea un nuevo nodo utilizando la clase Nodo con el dato proporcionado como argumento al método.
- **if self.cabeza is None:** Se verifica si la lista está vacía. Si la cabeza (self.cabeza) es None, significa que la lista está vacía.
- **actual = self.cabeza:** Se asigna la cabeza de la lista a la variable actual. Esta variable se utilizará para recorrer la lista y encontrar el último nodo.
- **while actual.siguiente is not None:** Se itera sobre la lista mientras el siguiente nodo (actual.siguiente) no sea None.
- **actual = actual.siguiente:** Esto permite avanzar a través de la lista hasta llegar al último nodo.
- **actual.siguiente = nuevo_nodo:** Una vez que se alcanza el último nodo (cuando actual.siguiente es None), se actualiza el enlace siguiente del último nodo para que apunte al nuevo nodo creado, insertándolo al final de la lista.

insertar_al_final

- **nuevo_nodo = Nodo(dato):** Se crea un nuevo nodo utilizando la clase Nodo con el dato proporcionado como argumento al método.
- **if self.cabeza is None:** Se verifica si la lista está vacía. Si la cabeza (self.cabeza) es None, significa que la lista está vacía.
- **actual = self.cabeza:** Se asigna la cabeza de la lista a la variable actual. Esta variable se utilizará para recorrer la lista y encontrar el último nodo.
- **while actual.siguiente is not None:** Se itera sobre la lista mientras el siguiente nodo (actual.siguiente) no sea None.
- **actual = actual.siguiente:** Esto permite avanzar a través de la lista hasta llegar al último nodo.
- **actual.siguiente = nuevo_nodo:** Una vez que se alcanza el último nodo (cuando actual.siguiente es None), se actualiza el enlace siguiente del último nodo para que apunte al nuevo nodo creado, insertándolo al final de la lista.

elemento_existe

- **actual = self.cabeza:** Se inicializa una variable actual con la cabeza de la lista.
- **while actual is not None:** Se inicia un bucle while que se ejecutará mientras la variable actual no sea None.
- **if actual.dato == elemento:** En cada iteración del bucle, se compara el dato del nodo actual (actual.dato) con el elemento que se está buscando.
- **actual = actual.siguiente:** Después de comparar el dato del nodo actual con el elemento buscado, la variable actual se actualiza para apuntar al siguiente nodo en la lista.

```
def elemento_existe(self, elemento):
    actual = self.cabeza
    while actual is not None:
        if actual.dato == elemento:
            return True
        actual = actual.siguiente
    return False
```

eliminar

```
def eliminar(self, elemento):
    if self.cabeza is None:
        return

    if self.cabeza.dato == elemento:
        self.cabeza = self.cabeza.siguiente
        return

    anterior = self.cabeza
    actual = self.cabeza.siguiente
    while actual is not None:
        if actual.dato == elemento:
            anterior.siguiente = actual.siguiente
            return
        anterior = actual
        actual = actual.siguiente
```

eliminar

- **if self.cabeza is None:** Se verifica si la lista está vacía.
- **if self.cabeza.dato == elemento:** Se verifica si el elemento a eliminar se encuentra en la cabeza de la lista.
  - Si el dato de la cabeza (self.cabeza.dato) es igual al elemento que se desea eliminar, significa que el primer nodo de la lista contiene el elemento a eliminar. En este caso, la cabeza se actualiza para apuntar al siguiente nodo **(self.cabeza = self.cabeza.siguiente)** y luego la función retorna.
  - Si el elemento a eliminar no se encuentra en la cabeza de la lista, se recorre la lista para encontrar el nodo que contiene el elemento. Se utilizan dos punteros, anterior y actual, para recorrer la lista.
    - Anterior apunta al nodo anterior al nodo actual,
    - Actual apunta al nodo actual en cada iteración.

eliminar

- Mientras actual no sea **None**, se verifica si el dato del nodo actual (**actual.dato**) es igual al elemento que se desea eliminar. Si es así, significa que se ha encontrado el nodo que contiene el elemento a eliminar. En este caso, el enlace siguiente del nodo anterior se actualiza para que apunte al siguiente nodo después del nodo actual (**anterior.siguiente = actual.siguiente**). Esto "salta" el nodo actual y lo elimina de la lista.

Después de actualizar los enlaces para eliminar el nodo deseado, la función retorna.

Ejercicio

Considerando lo trabajado en clase, desarrolle el algoritmo para la realizar una intercesión en una posición especifica de la lista, puede basarse en los ejemplos trabajados en clase y adjuntos como material de estudio.