Escuela de
Ciencia y Tecnología
ECyT_UNSAM

Programación (en C)

Primer Cuatrimestre 2025

programacionbunsam@gmail.com

Asignación dinámica de memoria

Asignación dinámica de memoria

→ Hasta ahora, la memoria se asignaba al momento de compilación.

Asignación dinámica de memoria

→ Hasta ahora, la memoria se asignaba al momento de compilación. Por este motivo, en muchas ocasiones, era necesario alocar memoria ‘extra’ por las dudas.

Asignación dinámica de memoria

→ Hasta ahora, la memoria se asignaba al momento de compilación. Por este motivo, en muchas ocasiones, era necesario alocar memoria ‘extra’ por las dudas. Por ejemplo, al leer un archivo de longitud desconocida, necesitábamos crear un vector de tamaño N, ya que no conocíamos de antemano la cantidad de información que el mismo contenía.

Asignación dinámica de memoria

→ Hasta ahora, la memoria se asignaba al momento de compilación. Por este motivo, en muchas ocasiones, era necesario alocar memoria 'extra' por las dudas. Por ejemplo, al leer un archivo de longitud desconocida, necesitábamos crear un vector de tamaño N, ya que no conocíamos de antemano la cantidad de información que el mismo contenía.

→ La asignación dinámica de memoria nos permite alocar memoria durante la ejecución del programa.

Asignación dinámica de memoria

→ Hasta ahora, la memoria se asignaba al momento de compilación. Por este motivo, en muchas ocasiones, era necesario alocar memoria 'extra' por las dudas. Por ejemplo, al leer un archivo de longitud desconocida, necesitábamos crear un vector de tamaño N, ya que no conocíamos de antemano la cantidad de información que el mismo contenía.

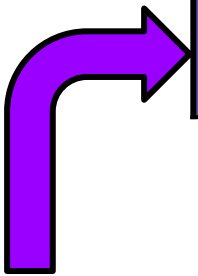
→ La asignación dinámica de memoria nos permite alocar memoria durante la ejecución del programa.

→ Cuando trabajamos de esta manera, estamos tratando con el *Heap* de la memoria

Estructura

Segmento de código	+ Código del programa - Solo lectura
Segmento de datos	+ Variables globales + Variables estáticas
Pile (pila)	+ Variables locales
Heap (montón)	+ Memoria dinámica - Administrable (malloc, free)

Estructura



Segmento de código	+ Código del programa - Solo lectura
Segmento de datos	+ Variables globales + Variables estáticas
Pile (pila)	+ Variables locales
Heap (montón)	+ Memoria dinámica - Administrable (malloc, free)

Funciones - malloc( )

def. malloc = memory allocation.

Esta función nos permite asignar un único bloque de memoria del tamaño deseado.

Funciones - malloc( )

def. malloc = memory allocation.

Esta función nos permite asignar un único bloque de memoria del tamaño deseado.

return.

→ Si el bloque es creado correctamente, nos devuelve un puntero al primer byte del bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

Funciones - malloc( )

def. malloc = memory allocation.

Esta función nos permite asignar un único bloque de memoria del tamaño deseado.

return.

→ Si el bloque es creado correctamente, nos devuelve un puntero al primer byte del bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

```
tipo* nombre = (tipo*) malloc(tamaño_en_bytes);
```

Funciones - malloc( )

def. malloc = memory allocation.

Esta función nos permite asignar un único bloque de memoria del tamaño deseado.

return.

→ Si el bloque es creado correctamente, nos devuelve un puntero al primer byte del bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

```
tipo* nombre = (tipo*) malloc(tamaño_en_bytes);
```

```
int* ptr = (int*) malloc(5 * sizeof(int));
```

Funciones - calloc( )

def. calloc = contiguous allocation.

Esta funciona asigna N bloques de tamaño M (bytes) cada uno. Además, inicializa todos los bloques con valor 0.

Funciones - calloc( )

def. calloc = contiguous allocation.

Esta funciona asigna N bloques de tamaño M (bytes) cada uno. Además, inicializa todos los bloques con valor 0.

return.

→ Si el bloque es creado correctamente, nos devuelve un puntero al primer bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

Funciones - calloc( )

def. calloc = contiguous allocation.

Esta funciona asigna N bloques de tamaño M (bytes) cada uno. Además, inicializa todos los bloques con valor 0.

return.

→ Si el bloque es creado correctamente, nos devuelve un puntero al primer bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

```
tipo* nombre = (tipo*) calloc(N, M);
```

Funciones - calloc( )

def. calloc = contiguous allocation.

Esta funciona asigna N bloques de tamaño M (bytes) cada uno. Además, inicializa todos los bloques con valor 0.

return.

→ Si el bloque es creado correctamente, nos devuelve un puntero al primer bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

```
tipo* nombre = (tipo*) calloc(N, M);
```

```
int* ptr = (int*) calloc(5, sizeof(int));
```

Funciones - malloc( ) vs calloc( )

	malloc	calloc
Asigna	1 bloque de tamaño N*M	N bloques de tamaño M
Valor inicial	Lo que hubiera en esa dirección de memoria (basura)	Inicializa todos los bloques en 0
Velocidad	rápida	lenta
Seguridad	menos segura	más segura

```
int* ptr = (int*) malloc(5 * sizeof(int));

int* ptr = (int*) calloc(5, sizeof(int));
```

Funciones - free( )

def. free = free memory.

Esta función libera un bloque de memoria **asignado dinámicamente**. Sabe automáticamente cuánto espacio de memoria debe liberar

Funciones - free( )

def. free = free memory.

Esta función libera un bloque de memoria **asignado dinámicamente**. Sabe automáticamente cuánto espacio de memoria debe liberar

return.

esta función no devuelve nada.

Funciones - free( )

def. free = free memory.

Esta función libera un bloque de memoria **asignado dinámicamente**. Sabe automáticamente cuánto espacio de memoria debe liberar

return.

esta función no devuelve nada.

```
free(ptr);
```

Funciones - free( )

def. free = free memory.

Esta función libera un bloque de memoria **asignado dinámicamente**. Sabe automáticamente cuánto espacio de memoria debe liberar

return.

esta función no devuelve nada.

```
free(ptr);
```

https://stackoverflow.com/questions/22481134/is-it-ever-ok-to-not-use-free-on-allocated-memory

Funciones - realloc( )

def. realloc = memory re-allocation.

Esta función modifica el tamaño del bloque/s asignado/s previamente a un único bloque de un nuevo tamaño.

Funciones - realloc( )

def. realloc = memory re-allocation.

Esta función modifica el tamaño del bloque/s asignado/s previamente a un único bloque de un nuevo tamaño.

return.

→ Si el bloque es modificado correctamente, nos devuelve un puntero al primer bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

Funciones - realloc( )

def. realloc = memory re-allocation.

Esta función modifica el tamaño del bloque/s asignado/s previamente a un único bloque de un nuevo tamaño.

return.

→ Si el bloque es modificado correctamente, nos devuelve un puntero al primer bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

```
nombre = (tipo*) realloc(nombre, new_size);
```

Funciones - realloc( )

def. realloc = memory re-allocation.

Esta función modifica el tamaño del bloque/s asignado/s previamente a un único bloque de un nuevo tamaño.

return.

→ Si el bloque es modificado correctamente, nos devuelve un puntero al primer bloque asignado.

→ Si no es posible crear el bloque, la función nos devuelve *NULL*.

```
nombre = (tipo*) realloc(nombre, new_size);
```

```
ptr = (int*) realloc(ptr, 10 * sizeof(int));
```

realloc( ) - consideraciones

realloc( ) - consideraciones

1. Cambia el tamaño del bloque/s original/es a un único bloque del tamaño que se le pasó como argumento a la función.

realloc( ) - consideraciones

1. Cambia el tamaño del bloque/s original/es a un único bloque del tamaño que se le pasó como argumento a la función.
2. Los espacios de memoria previamente asignados mantienen sus valores, pero los nuevos **no** son inicializados (contienen basura)

realloc( ) - consideraciones

1. Cambia el tamaño del bloque/s original/es a un único bloque del tamaño que se le pasó como argumento a la función.
2. Los espacios de memoria previamente asignados mantienen sus valores, pero los nuevos **no** son inicializados (contienen basura)
3. Al usar la función, pueden ocurrir 3 cosas

realloc( ) - consideraciones

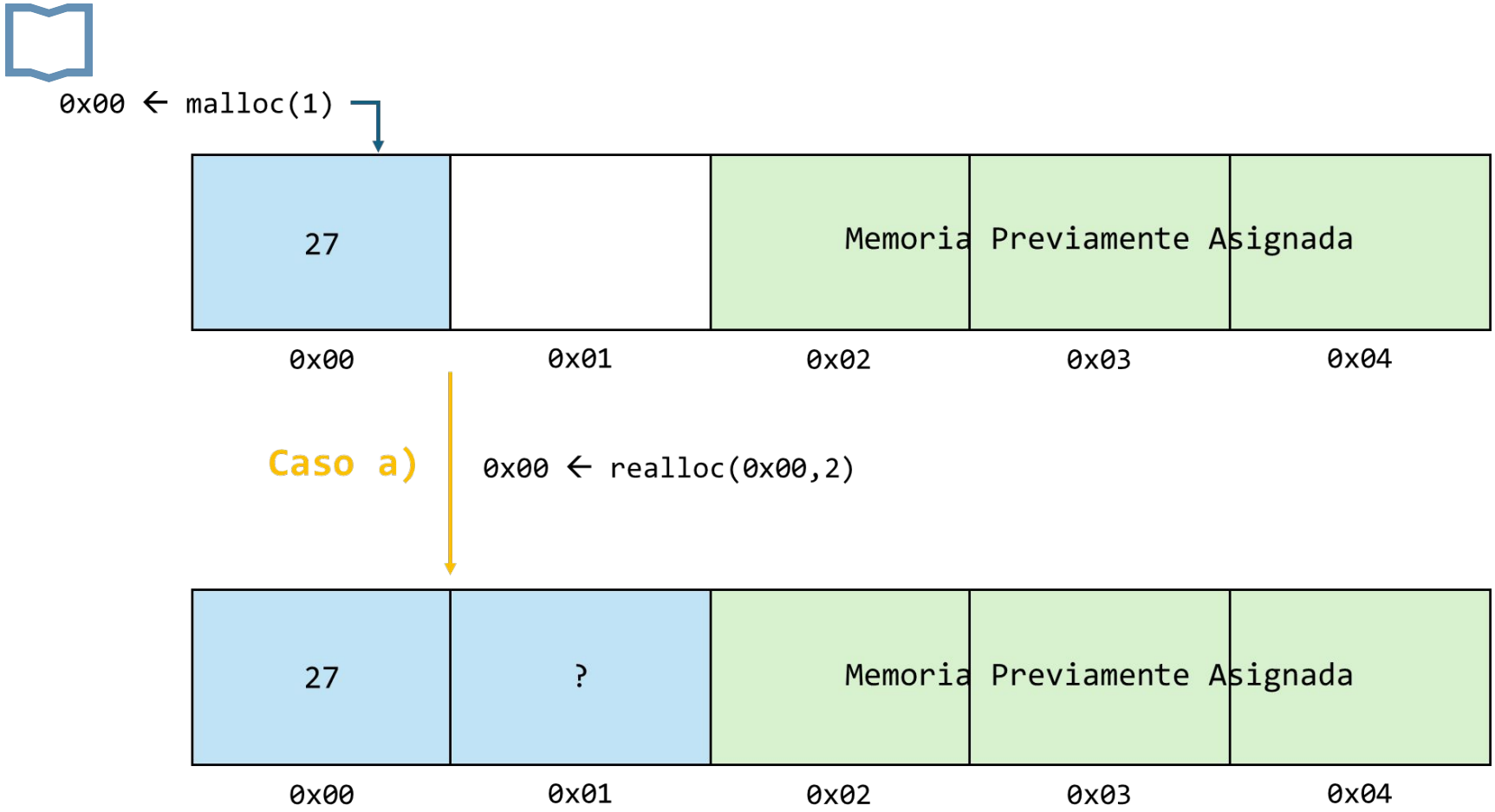
1. Cambia el tamaño del bloque/s original/es a un único bloque del tamaño que se le pasó como argumento a la función.
2. Los espacios de memoria previamente asignados mantienen sus valores, pero los nuevos **no** son inicializados (contienen basura)
3. Al usar la función, pueden ocurrir 3 cosas
    a. Se asigna la memoria sin movimientos: en este caso el puntero devuelto por la función coincidirá con aquel pasado como argumento.

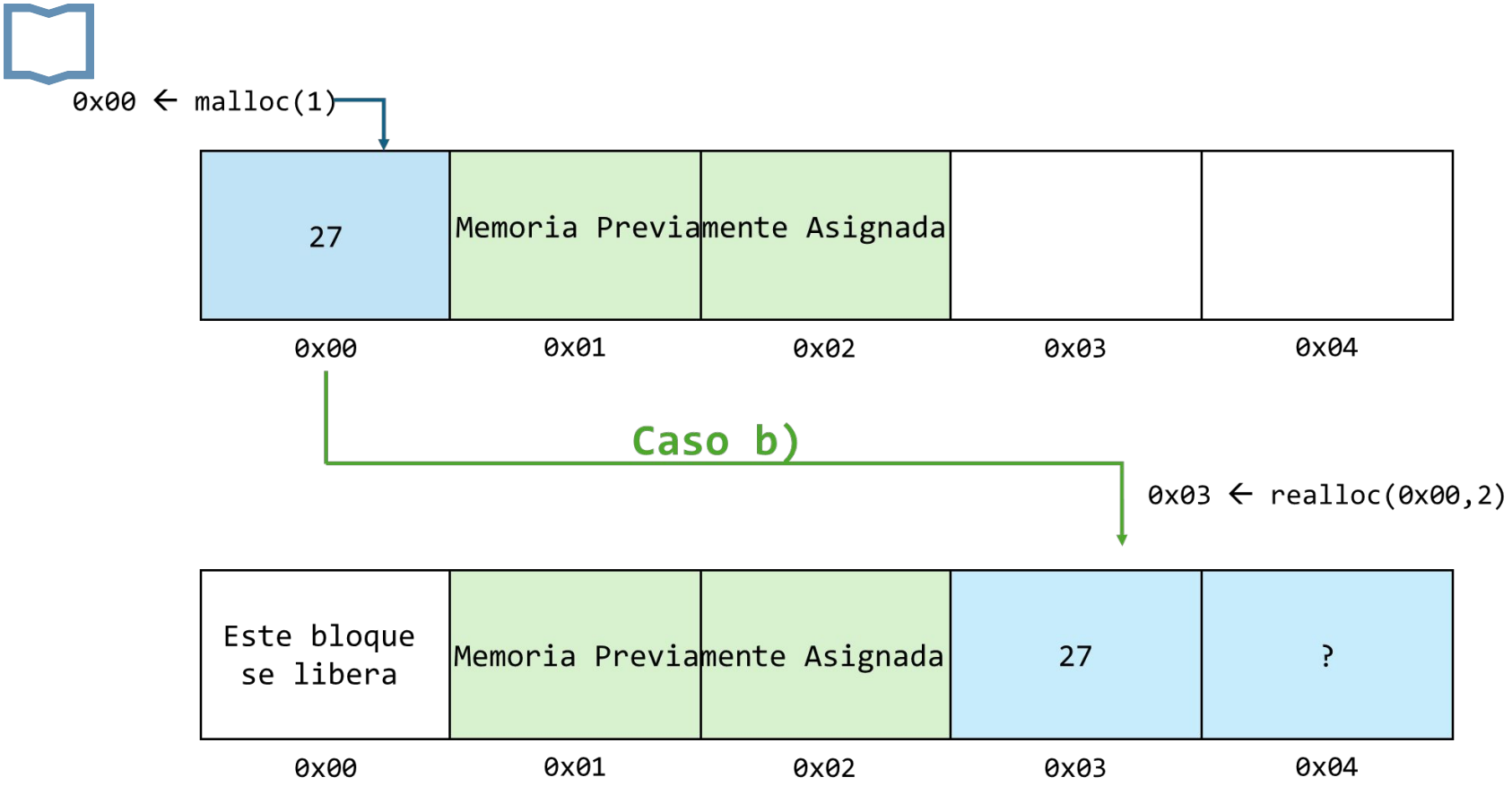
realloc( ) - consideraciones

1. Cambia el tamaño del bloque/s original/es a un único bloque del tamaño que se le pasó como argumento a la función.
2. Los espacios de memoria previamente asignados mantienen sus valores, pero los nuevos **no** son inicializados (contienen basura)
3. Al usar la función, pueden ocurrir 3 cosas
   a. Se asigna la memoria sin movimientos: en este caso el puntero devuelto por la función coincidirá con aquel pasado como argumento.
   b. Se asigna la memoria con movimientos: para poder crear el bloque de memoria es necesario trasladar los bloques previamente asignados a un nuevo lugar en la memoria. La memoria previa es liberada y no pierdo información.

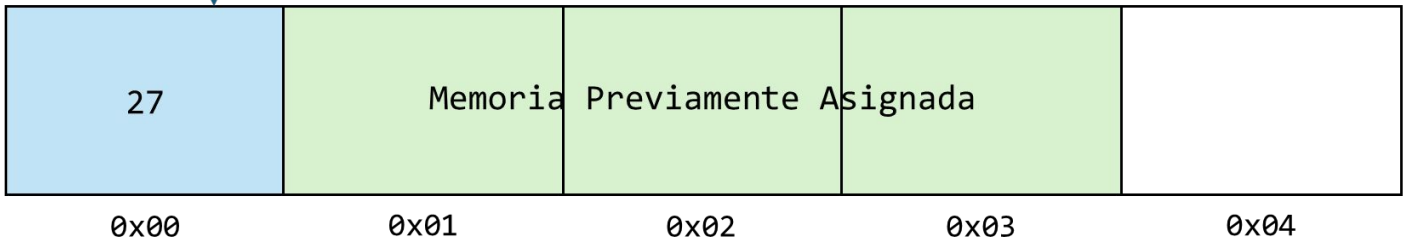
realloc( ) - consideraciones

1. Cambia el tamaño del bloque/s original/es a un único bloque del tamaño que se le pasó como argumento a la función.
2. Los espacios de memoria previamente asignados mantienen sus valores, pero los nuevos **no** son inicializados (contienen basura)
3. Al usar la función, pueden ocurrir 3 cosas
   a. Se asigna la memoria sin movimientos: en este caso el puntero devuelto por la función coincidirá con aquel pasado como argumento.
   b. Se asigna la memoria con movimientos: para poder crear el bloque de memoria es necesario trasladar los bloques previamente asignados a un nuevo lugar en la memoria. La memoria previa es liberada y no pierdo información.
   c. No es posible asignar un bloque de memoria del tamaño deseado, en cuyo caso la función devuelve NULL, pero no se borra la información original. Por este motivo es conveniente utilizar un puntero temporal junto con la función.


0x00 ← malloc(1)
27
Memoria Previamente Asignada
0x00
0x01
0x02
0x03
0x04
Caso a)
0x00 ← realloc(0x00,2)
27
?
Memoria Previamente Asignada
0x00
0x01
0x02
0x03
0x04


0x00 ← malloc(1)
27
Memoria Previamente Asignada
0x00
0x01
0x02
0x03
0x04
Caso b)
0x03 ← realloc(0x00,2)
Este bloque se libera
Memoria Previamente Asignada
27
?
0x00
0x01
0x02
0x03
0x04

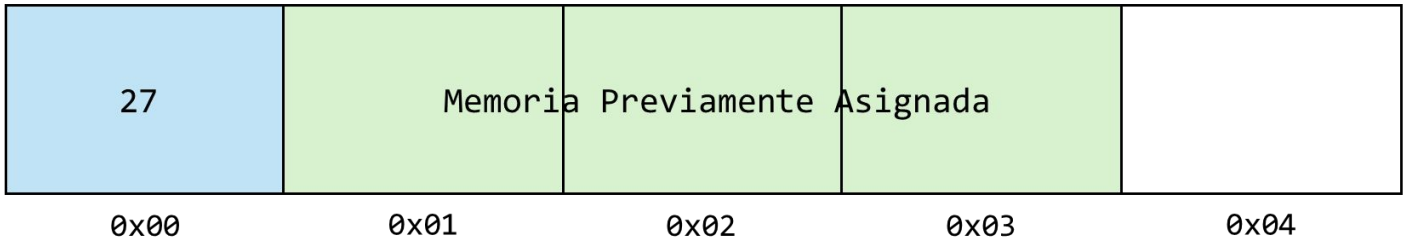
0x00 ← malloc(1)



Caso c)

NULL ← realloc(0x00,2)

Obs. Si bien en este caso no se borra la información, si no tengo cuidado con la asignación de punteros, puedo perderla



Estructuras dinámicas

def. Las estructuras dinámicas son aquellas cuyo tamaño no está definido, por lo que pueden agrandarse o achicarse a medida que avanza el programa.

Estructuras dinámicas

def. Las estructuras dinámicas son aquellas cuyo tamaño no está definido, por lo que pueden agrandarse o achicarse a medida que avanza el programa.

Su capacidad de adaptación las vuelve ideales para manejar grandes cantidades de información (i.e. videojuegos) o cuando directamente no conocemos la cantidad de información de antemano (i.e. archivos de tamaño desconocido, inputs de un usuario, etc.)

Estructuras dinámicas

def. Las estructuras dinámicas son aquellas cuyo tamaño no está definido, por lo que pueden agrandarse o achicarse a medida que avanza el programa.

Su capacidad de adaptación las vuelve ideales para manejar grandes cantidades de información (i.e. videojuegos) o cuando directamente no conocemos la cantidad de información de antemano (i.e. archivos de tamaño desconocido, inputs de un usuario, etc.)

¿Los vectores, como los usábamos previamente, son estructuras dinámicas?

Estructuras dinámicas vs estáticas

	Estática	Dinámica
Alocación de memoria	Fija al momento de compilación, no es flexible	Puede ser ajustada mientras el programa se ejecuta, es flexible
Eficiencia y performance	En general, son más rápidas y eficientes	El cambio de tamaño dinámico les quita velocidad y eficiencia
Implementación	En general con arrays (vectores)	Usando punteros y/o alocación de memoria dinámica
Uso	Cuando conozco el tamaño de la estructura o me importa más la velocidad que la memoria	Cuando necesito ajustar el tamaño en tiempo real o me importa más ahorrar memoria que velocidad

Listas enlazadas

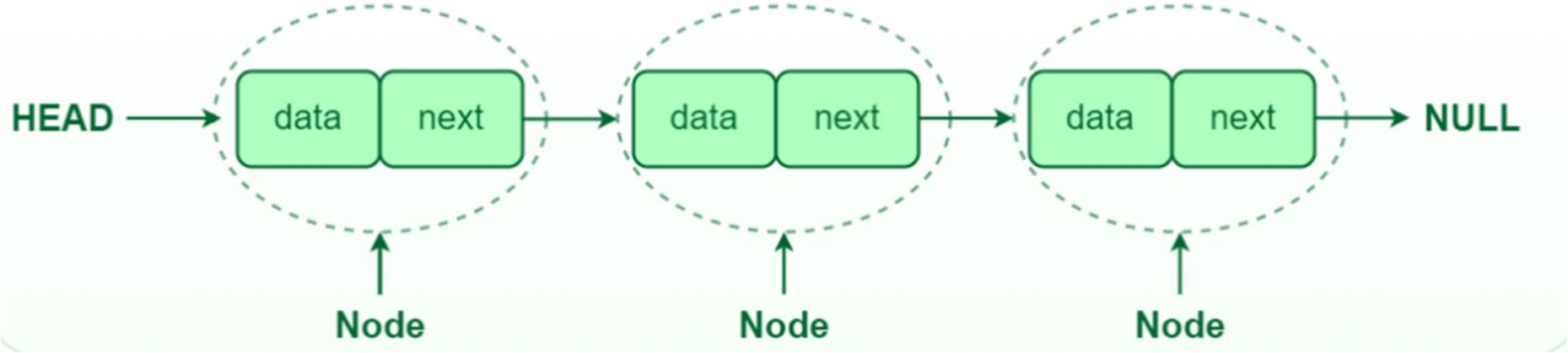
Las listas enlazadas son estructuras de datos lineales cuyos elementos no están almacenados en direcciones de memoria contiguas, sino que están enlazados mediante punteros.

Listas enlazadas

Las listas enlazadas son estructuras de datos lineales cuyos elementos no están almacenados en direcciones de memoria contiguas, sino que están enlazados mediante punteros.

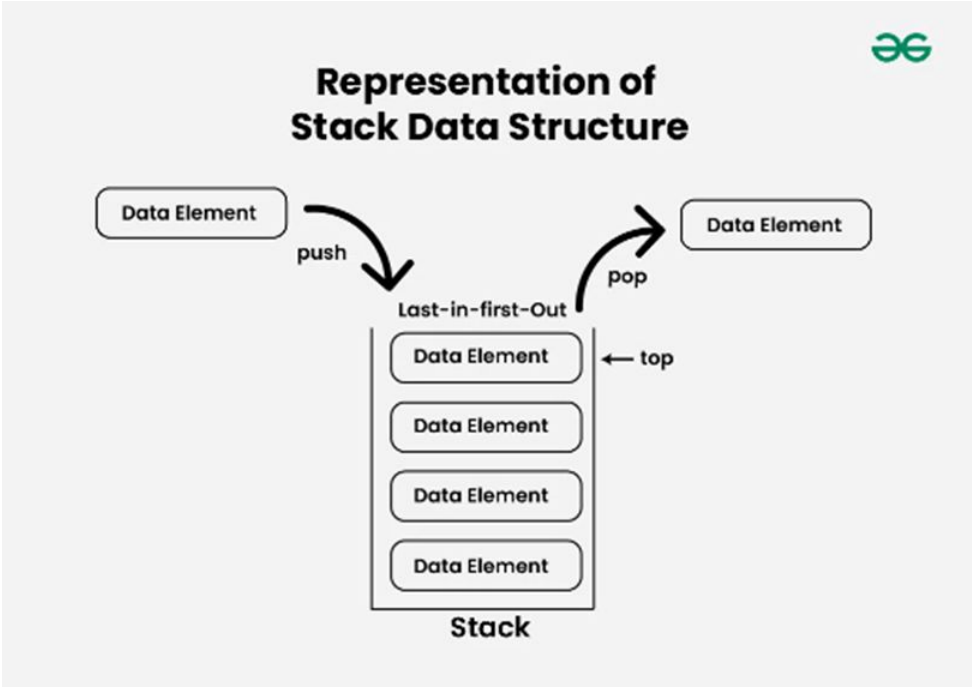
Nodo

- Data: contiene el valor o dato asociado al nodo
- 'next' (puntero o referencia): contiene la dirección de memoria (referencia) de algún otro nodo en la secuencia



Pilas (Stack)

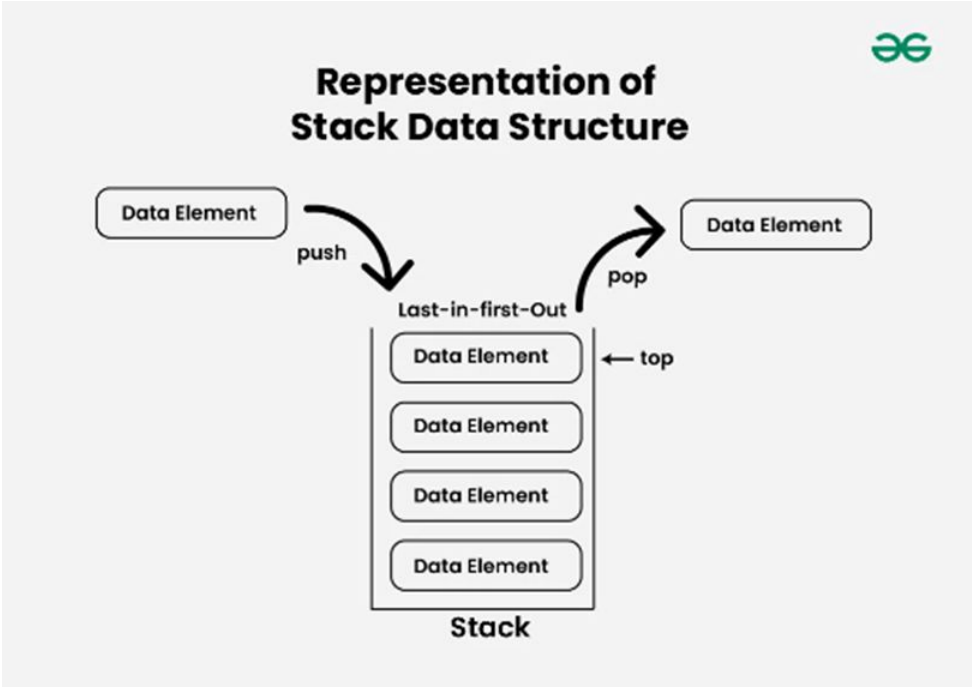
Las pilas o stacks son estructuras lineales que se caracterizan por seguir el principio **L**ast **I**n **F**irst **O**ut (**LIFO**).



Pilas (Stack)

Las pilas o stacks son estructuras lineales que se caracterizan por seguir el principio **L**ast **I**n **F**irst **O**ut (**LIFO**).

Todas las operaciones (agregado y quitado de elementos) ocurren en el mismo lugar, la cima (top) de la pila.

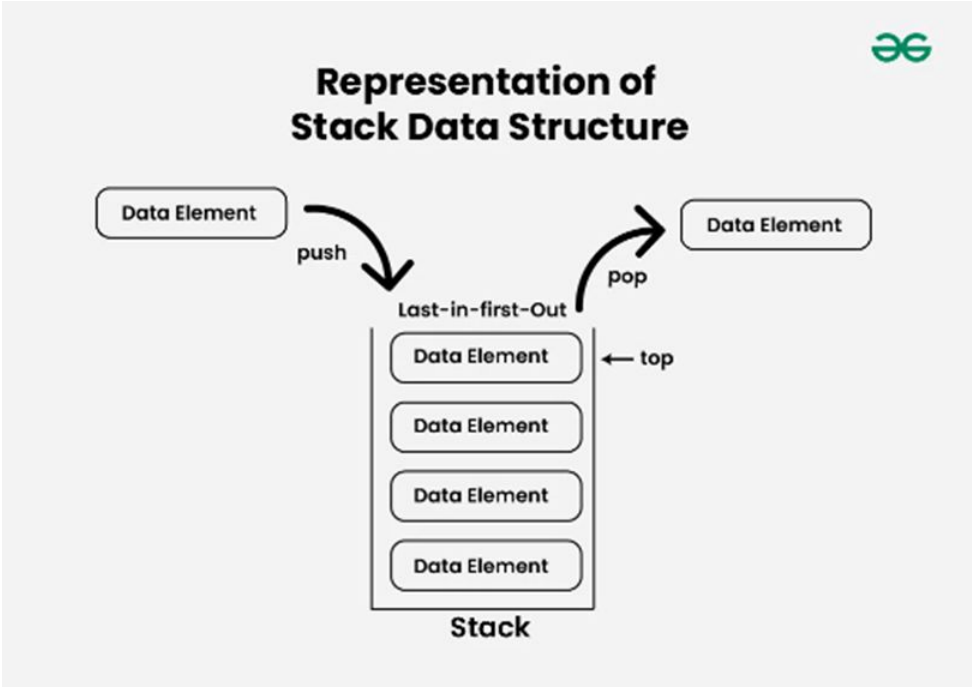


Pilas (Stack)

Las pilas o stacks son estructuras lineales que se caracterizan por seguir el principio **L**ast **I**n **F**irst **O**ut (**LIFO**).

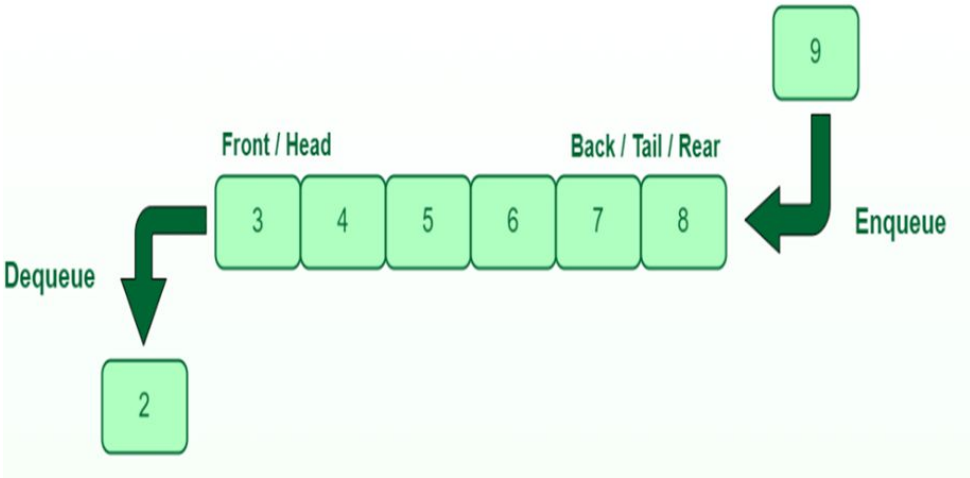
Todas las operaciones (agregado y quitado de elementos) ocurren en el mismo lugar, la cima (top - cabeza) de la pila.

Esto implica que para quitar el N-ésimo elemento, tengo que primero sacar (N-1) de ellos.



Colas (Queues)

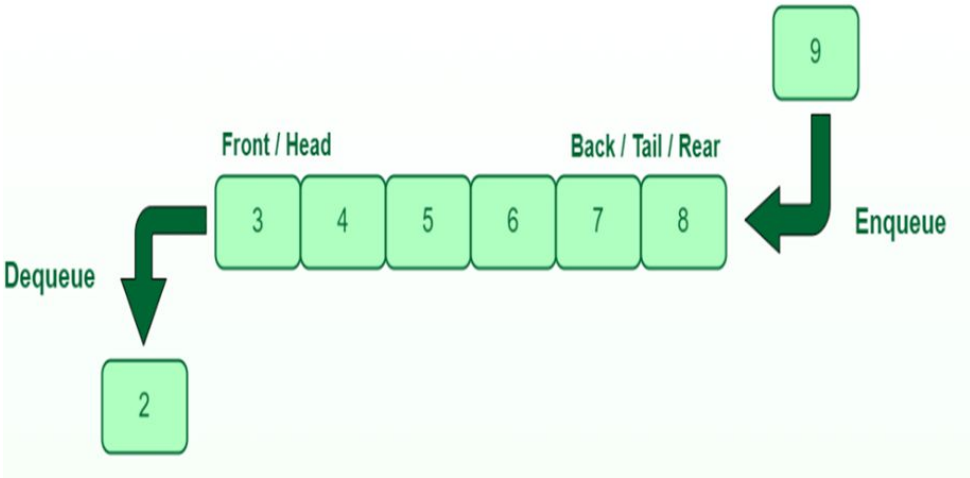
Las colas o queues son estructuras lineales que siguen el principio **F**irst **I**n **F**irst **O**ut (**FIFO**).



Colas (Queues)

Las colas o queues son estructuras lineales que siguen el principio **F**irst **I**n **F**irst **O**ut (**FIFO**).

Las inserción de elementos ocurre al final (back/tail/rear) de la cola, mientras que son eliminados en el frente (front / head) de la misma.



Colas (Queues)

Las colas o queues son estructuras lineales que siguen el principio **F**irst **I**n **F**irst **O**ut (**FIFO**).

Las inserción de elementos ocurre al final (back/tail/rear) de la cola, mientras que son eliminados en el frente (front / head) de la misma.

A diferencia de las pilas, tenemos 2 ‘puntos de acceso’ a la estructura.