Mémoria dinámica

Parte 1

Estructuras de Datos

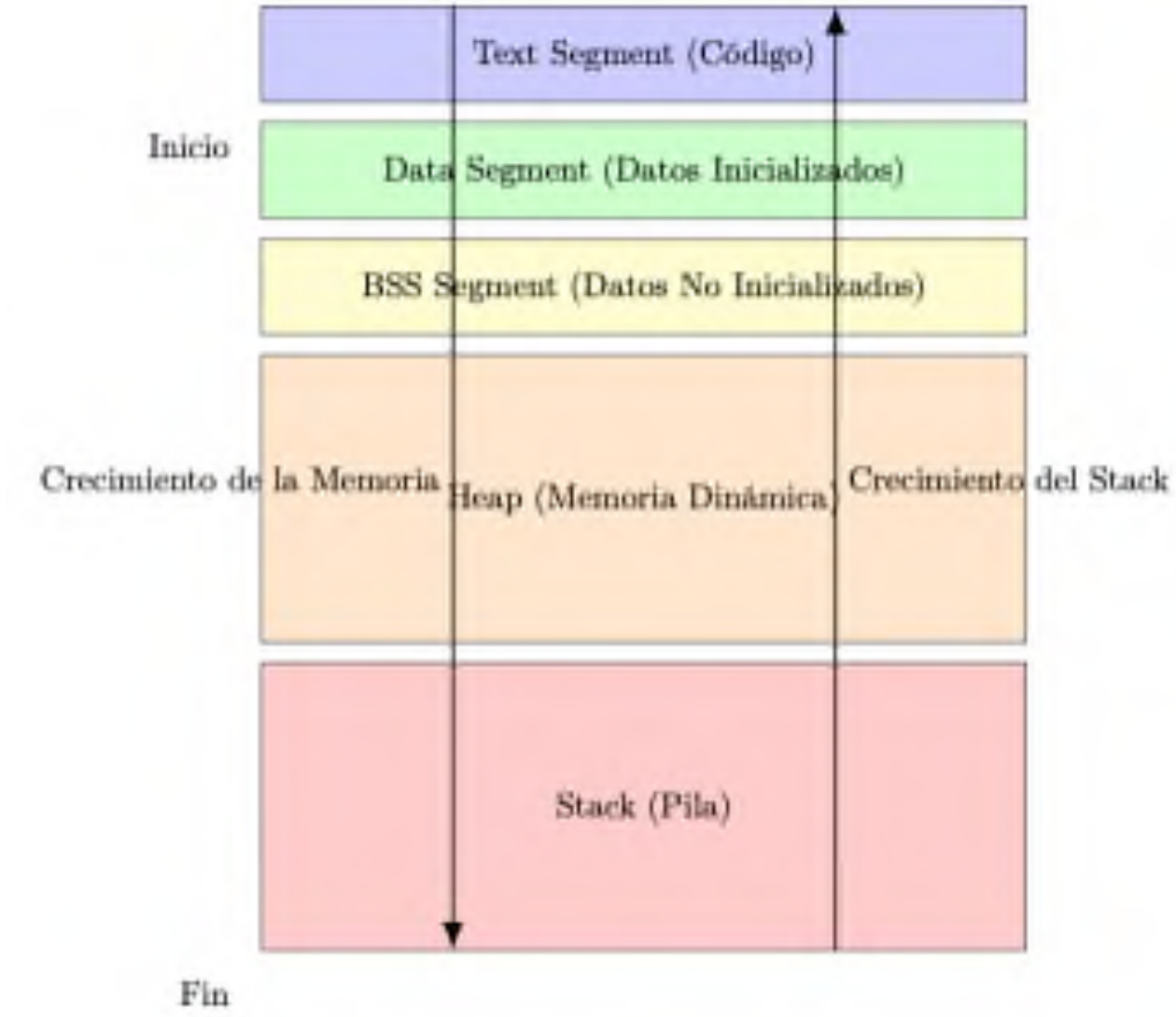
Dpto. Lenguajes y Ciencias de la Computación.

Universidad de Málaga

La memoria *No* dinámica

```c
#include <stdio.h>
int global_var = 10; // Sección de datos inicializados
int x;               // Sección de datos no inicializados
void contador(void){
    static int static_counter = 0; // Sección de datos inicializados
    static_counter++;
    x = static_counter;
    printf("Llamado %d veces\n", static_counter);
}
int main(void){
    short i; // Sección de datos Stack
    for (i = 0; i < global_var; i++)
    {
        contador();
    }
    printf("Valor de x %d\n", x);
    // error: use of undeclared identifier 'static_counter'
    //  printf("Valor de static_counter %d", static_counter);
    return 0;
}
```

Código



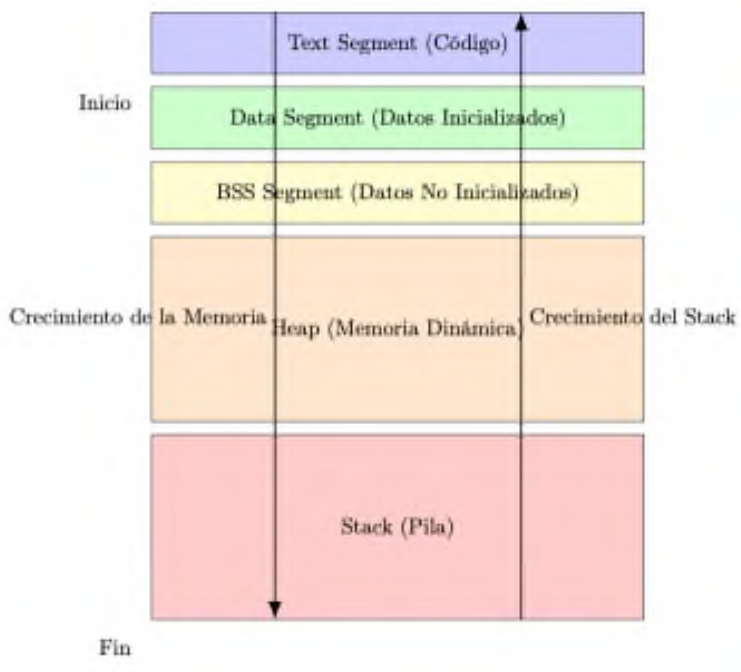
Solicitud memoria dinámica

- El tipo *size_t* indica un tamaño en bytes. Es una redefición de un entero sin signo.
- Todos devuelven un *void **, o *NULL* si no se puede asignar memoria. Se debe realizar *casting*.

```
//size: tamaño de bytes
void * malloc(size_t size);
```

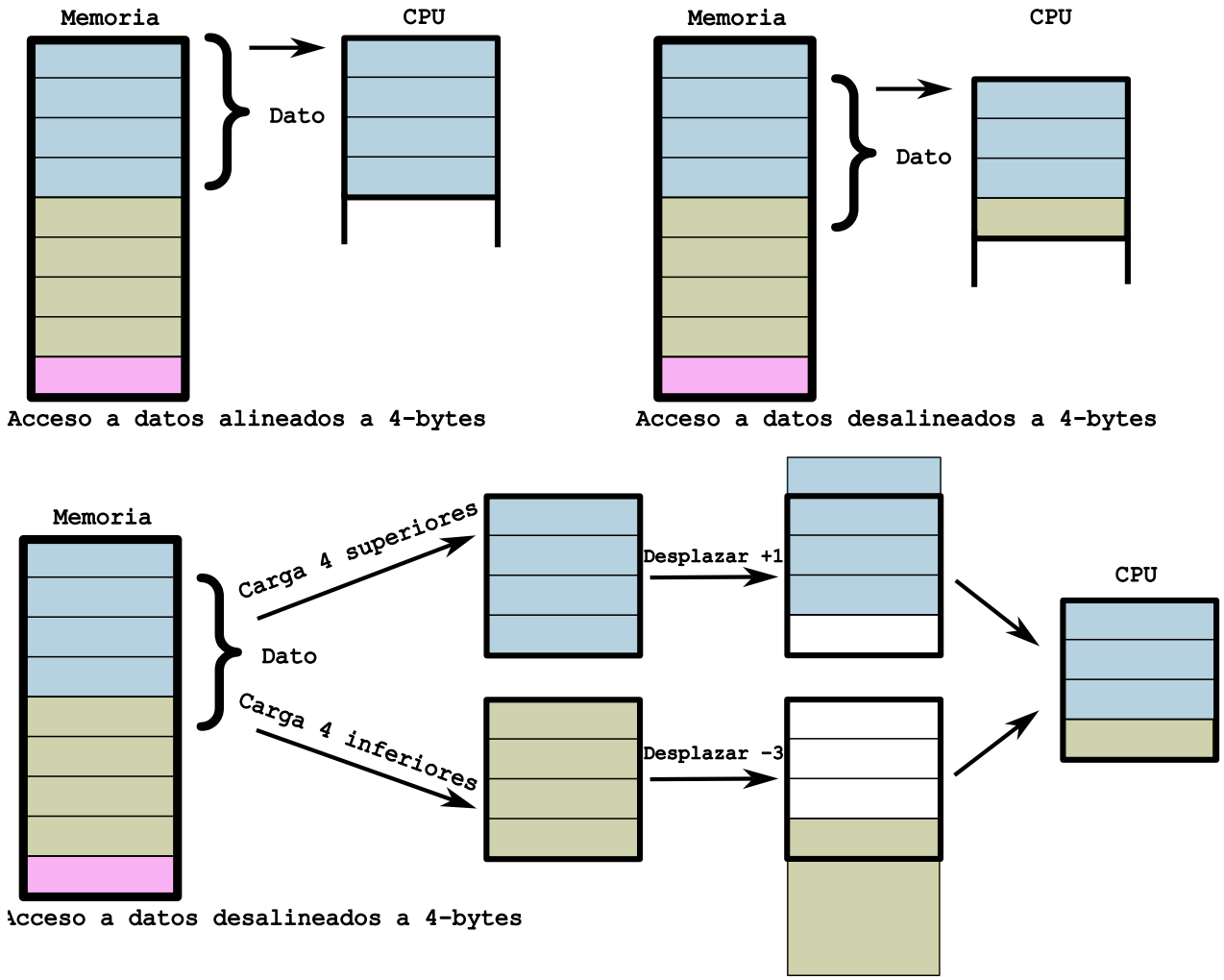
```
//count: número elementos; size: tamaño de bytes
void * calloc(size_t count, size_t size);
```

```
//ptr: zona de memoria; size: nuevo tamaño en bytes
void * realloc(void *ptr, size_t size);
```



Asignación de mémoria eficiente en tiempo

- La CPU no lee la memoria byte a byte. Lo hace en bloques de 2, 4, 8, etc. El motivo es el rendimiento en lectura/escritura.
- El alineamiento de datos implica que las direcciones de los datos han de ser divisibles por 1, 2, 4, etc.



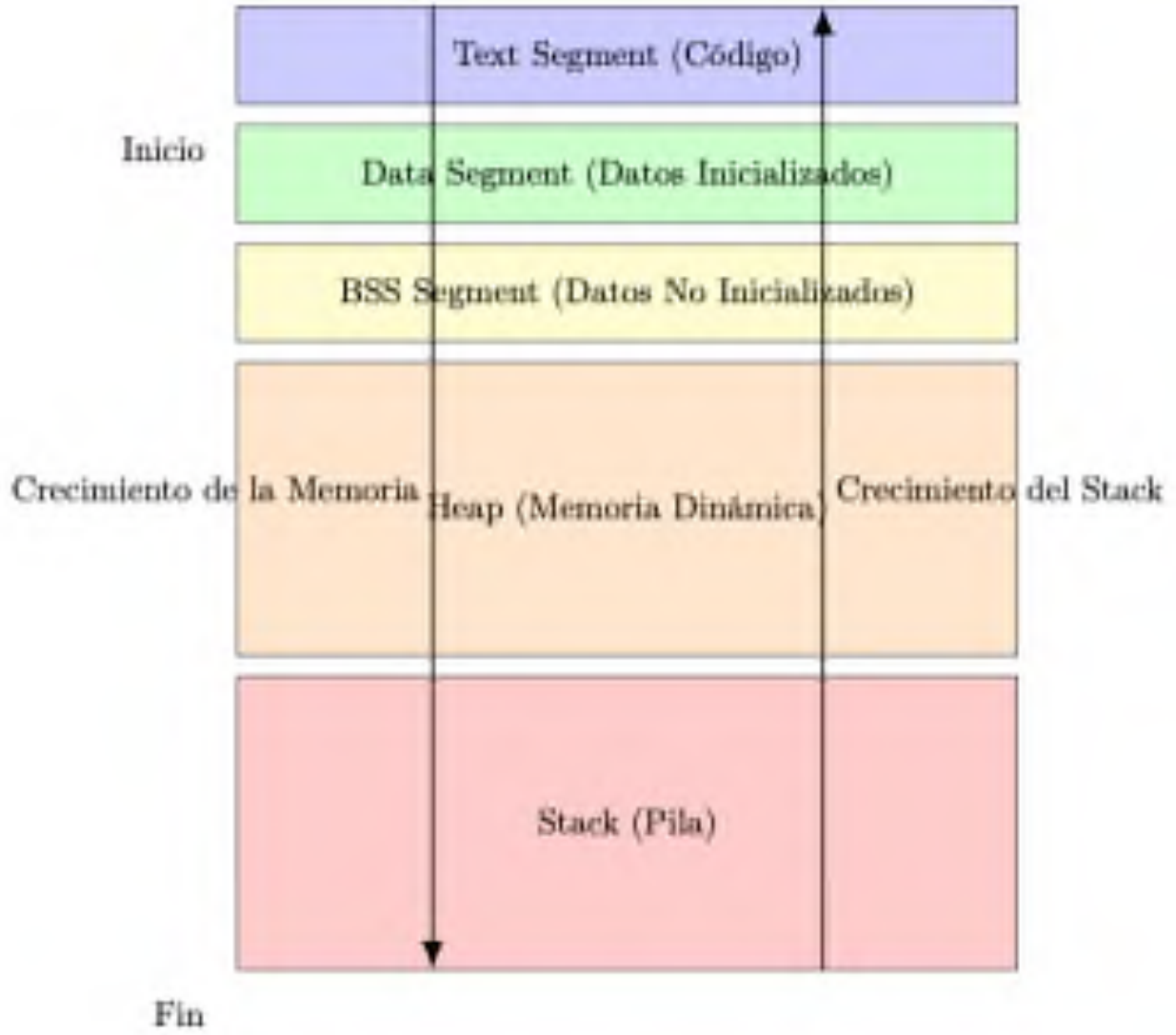
Asignación de memoria: *padding*

¿Cual de estas estructuras ocupa más? Usa *sizeof*

```
struct SinPadding
{
    char c;      // 1 byte
    int i;       // 4 bytes
    char c2;     // 1 byte
};

struct ConPadding
{
    char c;      // 1 byte
    char c2;     // 1 byte
    int i;       // 4 bytes
};
```

La memoria dinámica

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // Para srand y rand

int main()
{
    int numElementos;

    //Entrada teclado
    printf("Ingrese el número de enteros que desea generar: ");
    scanf("%d", &numElementos);

    //Petición de memoria
    int *arrayEnteros = (int *)malloc(numElementos * sizeof(int));

    //No se puede conceder memoria, nos salimos
    if (arrayEnteros == NULL){
        perror("Error: No se pudo reservar memoria.\n");
        exit(1);
    }

    srand(time(NULL)); // Semilla
    for (int i = 0; i < numElementos; i++){
        arrayEnteros[i] = rand() % (99 - 0 + 1);
    }

    printf("Array generado con %d elementos:\n", numElementos);
    for (int i = 0; i < numElementos; i++){
        printf("%d ", arrayEnteros[i]);
    }
    printf("\n");
    printf("Teclea enter para finalizar");
    getchar(); //Retorno anterior
    getchar();

    //Liberamos la memoria del montículo
    free(arrayEnteros);
    return 0;
```



Persistencia en el montículo

- Ejecuta el ejemplo y observa la cantidad de memoria que consume tu programa.
- ¿Cuánto consume cuando comentas la línea de //free(arrDin); ?
- ¿Cuánto consume cuando se descomenta?
- ¿Cuándo se libera esa memoria del montículo?

DevoradorMemoria.c

```
joaquin@pdi-131-199 EstructuraDatos % ps -la
  UID   PID  PPID       F CPU PRI NI       SZ    RSS WCHAN     S             ADDR TTY          TIME CMD
  501 51644 36103    4006   0  31  0 408253424   1280 -       Ss+              0 ttys008    0:00.01 /bin/zsh -i
  501 52703 36103    4006   0  31  0 408261616   1280 -       Ss+              0 ttys010    0:00.01 /bin/zsh -i
  501 53461 36103    4006   0  31  0 408441104   3360 -       Ss               0 ttys012    0:00.09 /bin/zsh -il
  501 55346 53461    4006   0  31  0 407980640   1232 -       R+               0 ttys012    0:12.24 ./a.out
    0 55370 55367    4106   0  31  0 408664848   9616 -       Ss               0 ttys013    0:00.02 login -pfl joaqu
  501 55371 55370    4006   0  31  0 408813568   4432 -       S                0 ttys013    0:00.02 -zsh
    0 55433 55371    4106   0  31  0 408626864   2048 -       R+               0 ttys013    0:00.00 ps -la
```

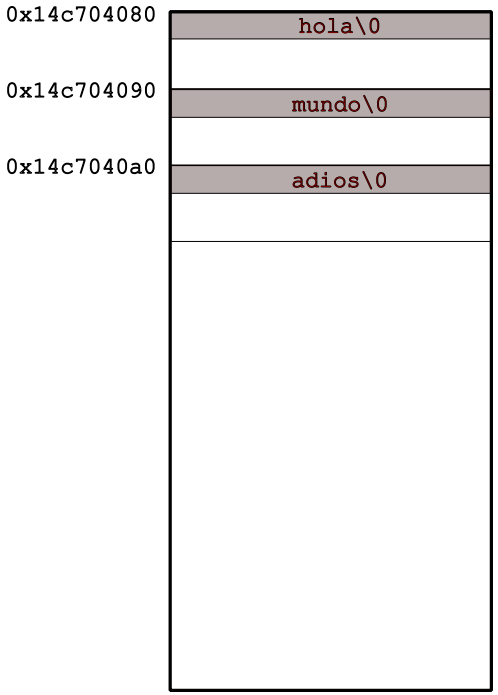
Liberar memoria

- Cuando se solicita memoria esta queda marcada como usada.
- Futuras peticiones, usarán otras zonas de memoria libres.
- Cuando se deja de utilizar una zona de memoria, **¡esta se DEBE liberar!**
- Cuando finaliza el programa, libera toda la memoria.

```
//ptr: dirección de memoria a liberar.
void free(void *ptr);
```

Ejemplo de fuga de memoria

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void){
    char *nombre = (char *)malloc(sizeof(char) * 6);
    char *apellido = (char *)malloc(sizeof(char) * 6);
    if (nombre == NULL || apellido == NULL){
        printf("Error al asignar memoria\n");
        return 1;
    }
    strcpy(nombre, "Hola");
    strcpy(apellido, "mundo");
    printf("%s %s en   %p y %p\n", nombre, apellido, (void *)nombre, (void *)apellido);

    nombre = (char *)malloc(sizeof(char) * 10);
    strcpy(nombre, "Adios");
    printf("%s %s en  %p y %p\n", nombre, apellido, (void *)nombre, (void *)apellido);

    free(nombre);
    free(apellido);
    return 0;

}
```



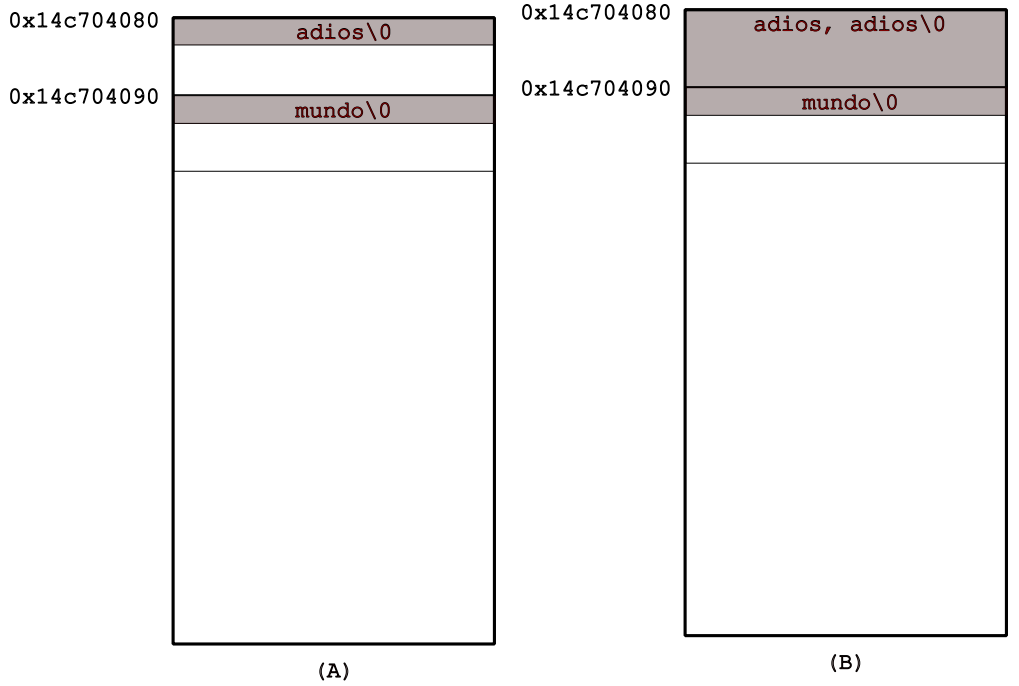
FugaMemoria

Fugas de memoria en C

- Una fuga de memoria (memory leak) ocurre cuando un programa reserva dinámicamente memoria pero no la libera y pierde la referencia a ella.
- El acceso a memoria no reservada (acceso fuera de rango), puede conllevar a errores en tiempo de ejecución.

Completa sin fuga de memoria

1. Modifica el ejemplo anterior para evitar la fuga de memoria (resultado final del montículo en (A)).
2. Usa realloc para modificar el tamaño de la memoria solicitada a nombre (16 caractéres) y asígnale el contenido "adios, adios" (resultado final del montículo en (B)).



¿Funciona este código?

```
//...
    // Size_t es un entero que se usa para represenar número de bytes
    // SIZE_MAX es el valor máximo de bytes de un objeto.
    size_t numBytes = SIZE_MAX;
    printf("Intento pedir %zu bytes\n", numBytes);

    int *arrDin = (int *)malloc(numBytes);
    arrDin[0] = 1;
    printf("Primer elemento %d en zona de memoria %p\n", arrDin[0], (void*) arrDin);

    // Libero memoria
    free(arrDin);
//...
```

PedirMemoria.c

Errores en tiempo de ejecución

- Acceso a memoria no asignada (e.g. fuera de rango, punteros que apuntan a direcciones no inicializadas, aritmética de punteros errónea)
- Uso incorrecto de punteros (desrefenciar punteros despues de liberar, o que apuntan NULL).
- Desborde de la pila (llamadas recursivas excesivas, estructuras de datos que ocupan mucho).
- Liberación de punteros

Encuentra los errores en los siguientes códigos

- Revisa bien cuándo y cómo se aplica cada función para la gestión de la memoria dinámica.
- Revisa los índices y tamaños que se definen y se usan.
- **¡Encuentralo el/la primer@!**

Ejemplo1.c, Ejemplo2.c, Ejemplo3.c

Estructuras enlazadas

```
struct Node {                   // Nuevo tipo: struct Nodo.
    int data;                   // Entero, 4 bytes
    struct Nodo* next;          // Puntero, 4/8 bytes. Forward declaration
};
```

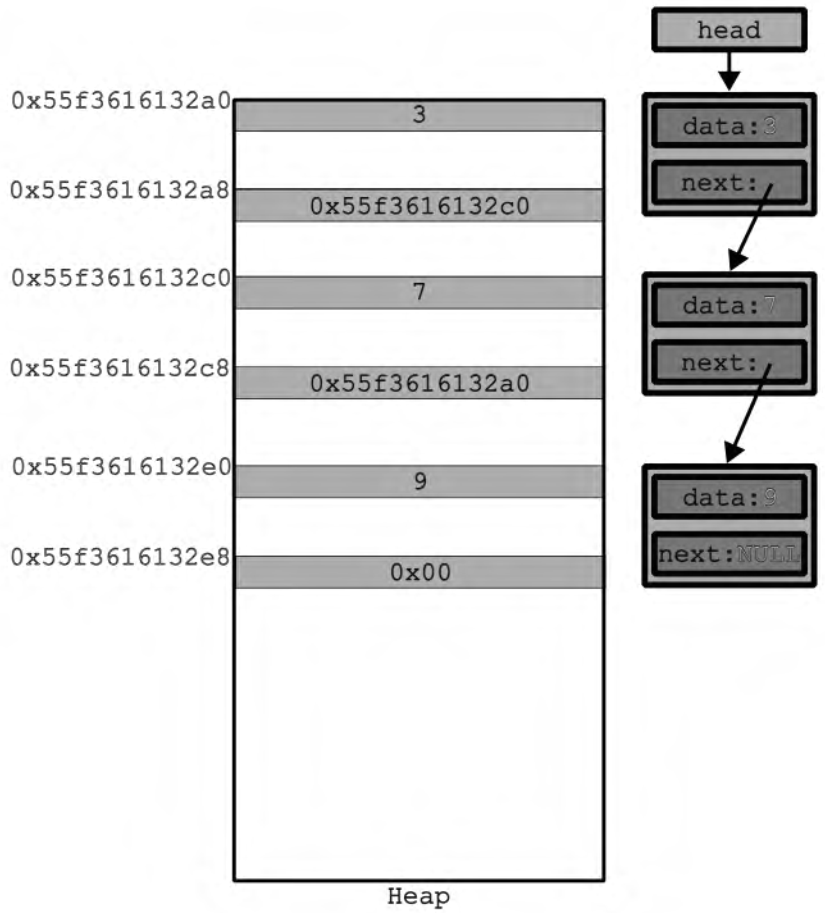
```
struct Node {                   // Nuevo tipo: struct Nodo.
    int data;                   // Entero, 4 bytes
    struct Nodo* left;          // Puntero, 4/8 bytes. Forward declaration
    struct Nodo* right;         // Puntero, 4/8 bytes. Forward declaration
};
```

```
struct Node {                   // Nuevo tipo: struct Nodo.
    int data;                   // Entero, 4 bytes
    struct Nodo* next;          // Puntero, 4/8 bytes. Forward declaration
    struct Nodo* prev;          // Puntero, 4/8 bytes. Forward declaration
};
```

Uso de typedef en estructuras

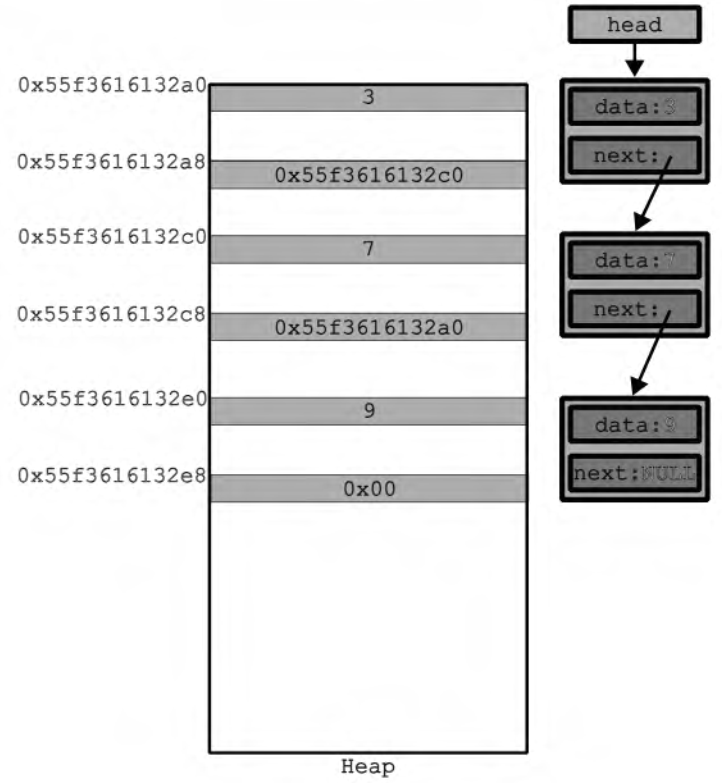
- Crea alias o nombres alternativos para tipos de datos existentes
- Hace las declaraciones más concesisas y fáciles de entender.
- Completa este ejercicio (*ListaEnlazada.c*):

```c
typedef struct Node *ptrNode; // Alias para un puntero a la estructura
typedef struct Node{
    int data;
    ptrNode next;
} Node; // Alias para la estructura

int main(void){
    // Declara una solo variable de tipo ptrNode llamada head.
    // Pide memoria para tres nodos y enlaza cada una de ellas para tener el valor 3->7->9->NULL.
    return 0;
}
```



Operaciones en una lista enlazada

- Iterar
- Insertar en la cabeza
- Insertar en la cola
- Insertar en una posición intermedia
- Eliminar en la cabeza
- Eliminar en la cola
- Eliminar en una posición intermedia
- Destruir lista

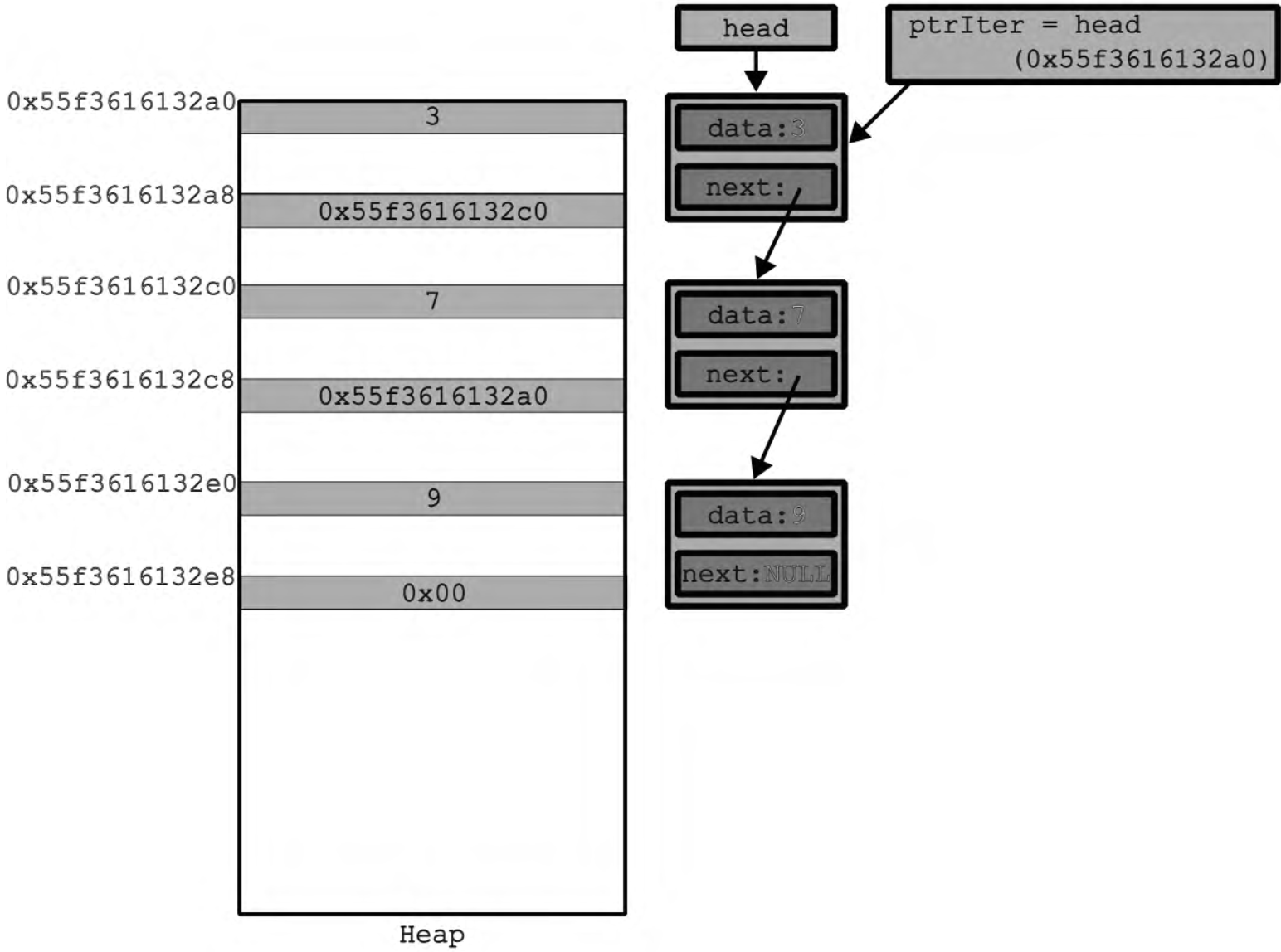


Iterar

```
//Añade este método a ListaEnlazada.c
void iterar(ptrNode head){
    //Imprime por pantalla la información de cada nodo: su data, la posición de memoria de data y el next.
}
...
```

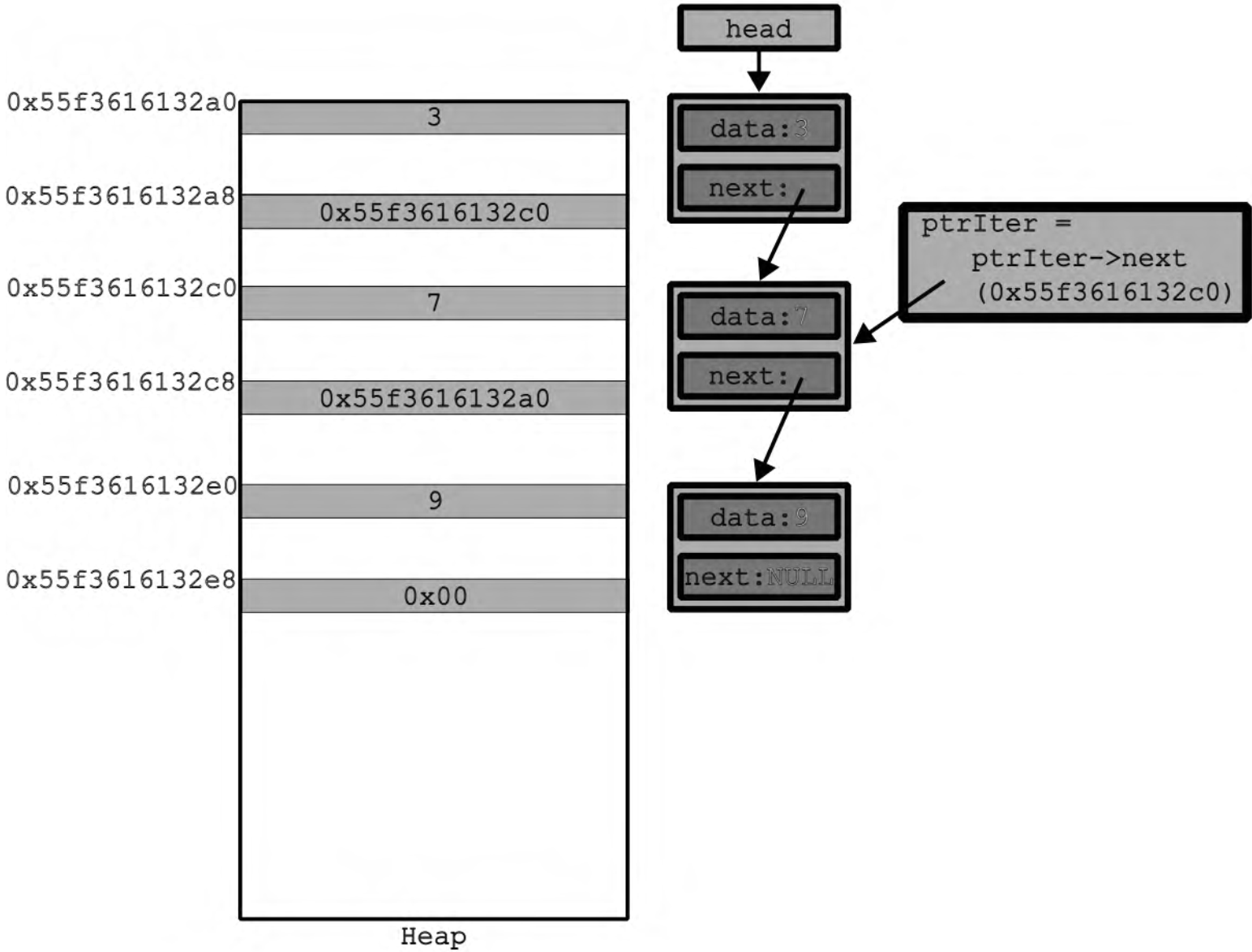
Iterar

- Crear una variable auxiliar *ptrIter* que apunte a *head*



Iterar

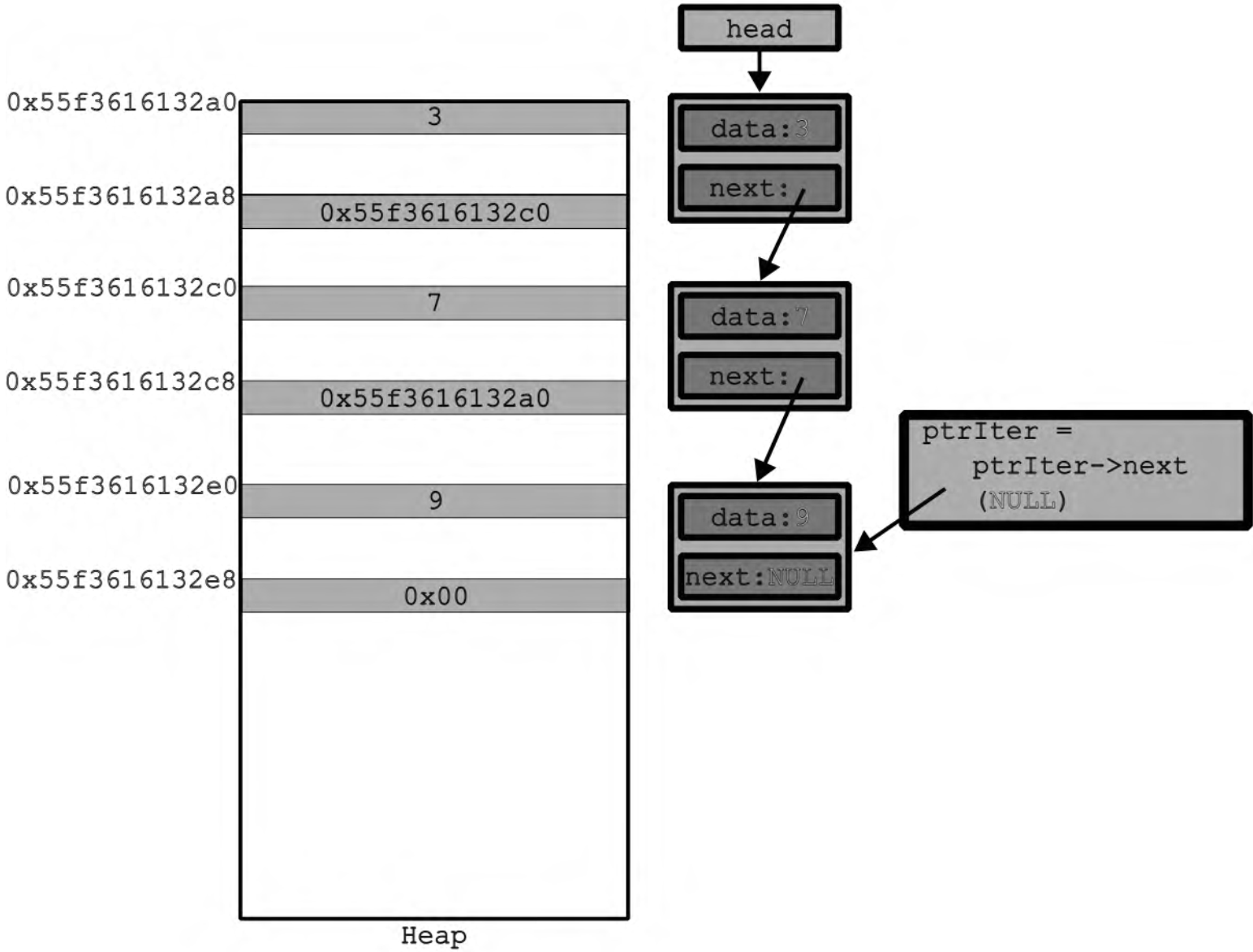
- Una vez mostramos el nodo actual. Modicamos el valor de la variable *ptrIter* para que apunte al *next*.



Iterar

- Terminamos de iterar cuando llegamos al final de la lista:

ptrIter->next* es *NULL*



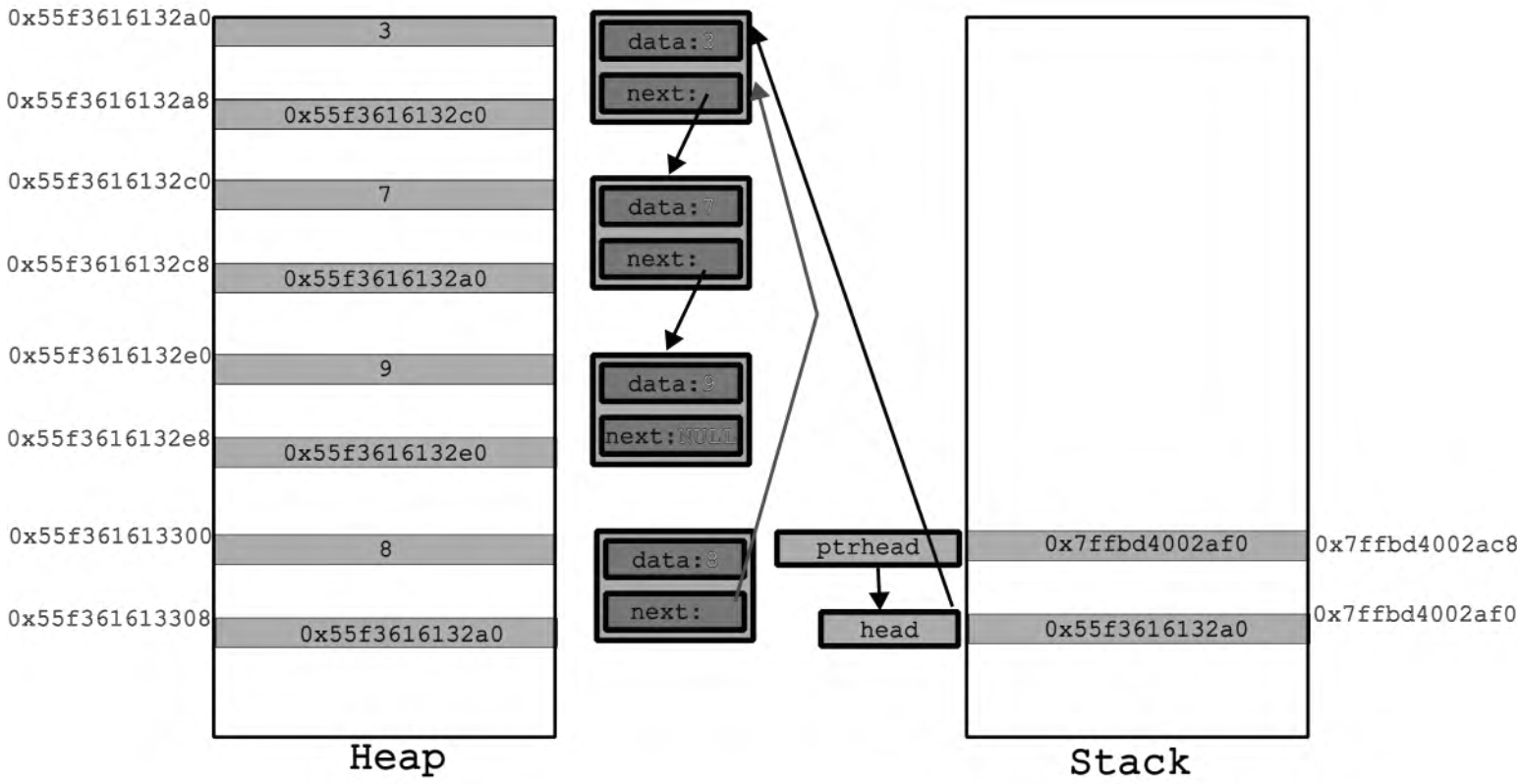
Insertar

- Depende dónde se inserte, se tendrá que modificar la cabeza de la lista *head*.
- La variable *head* es un puntero a la estructura *struct node* (tipo *ptrNode*), para poder modificarla, pasamos un puntero a ella *ptrhead*.
- (Recordatorio) Para modificar **cualquier** variable externa a un procedimiento/función, se debe pasar su dirección de memoria (puntero).

```
typedef struct Node *ptrNode;
typedef struct Node{
    int data;
    ptrNode next;
} Node;

//Añade este método a ListaEnlazada.c. ptrhead es de tipo puntero a ptrNode;
//ptrhead is de tipo puntero a puntero a struct Node (struct Node ** ptrhead).
bool insertar(ptrNode * ptrhead, int data);
```

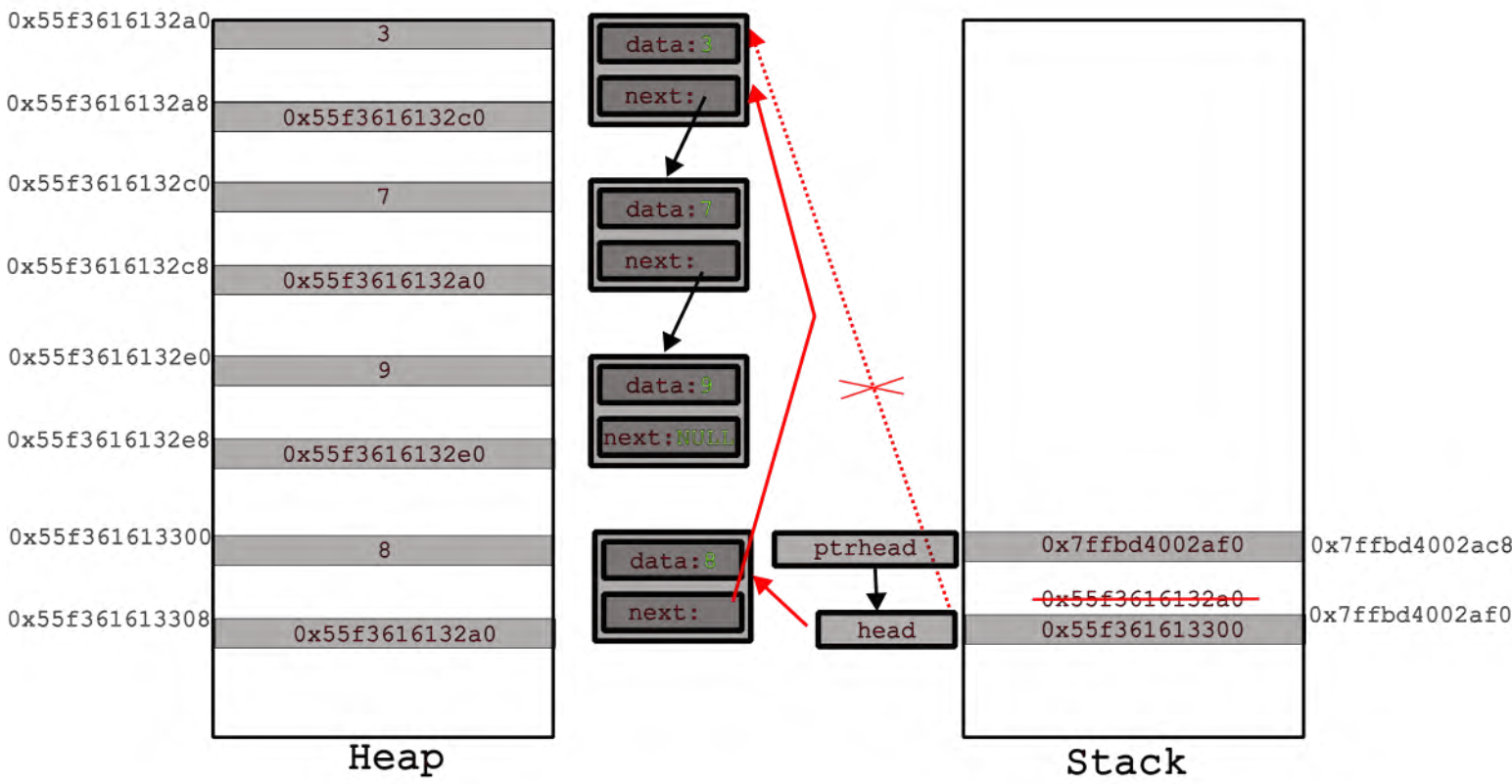
Insertar en cabeza 1/2

- El nuevo nodo apunta a *head*



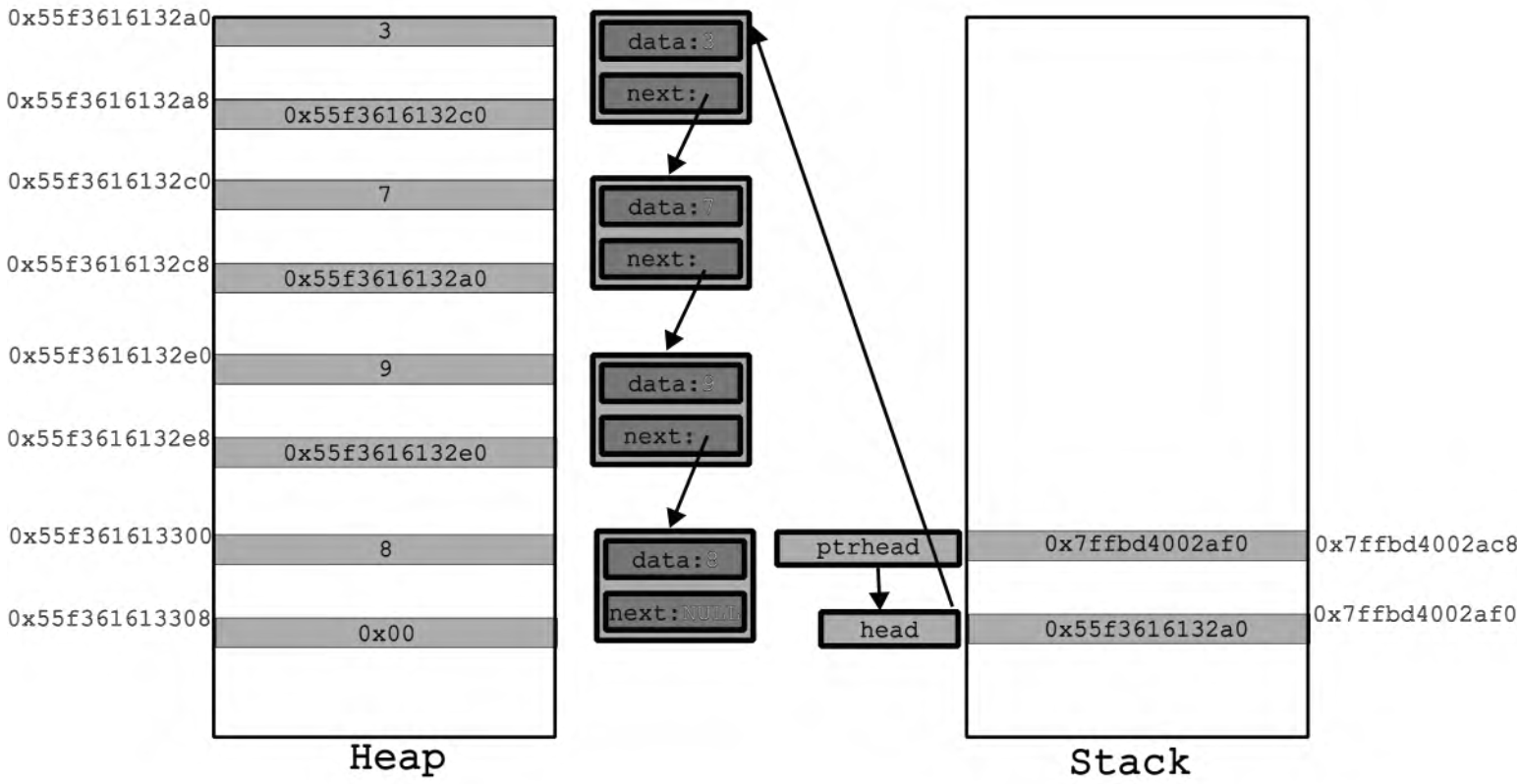
Insertar en cabeza 2/2

- *head* apunta al nuevo nodo ya enlazado.

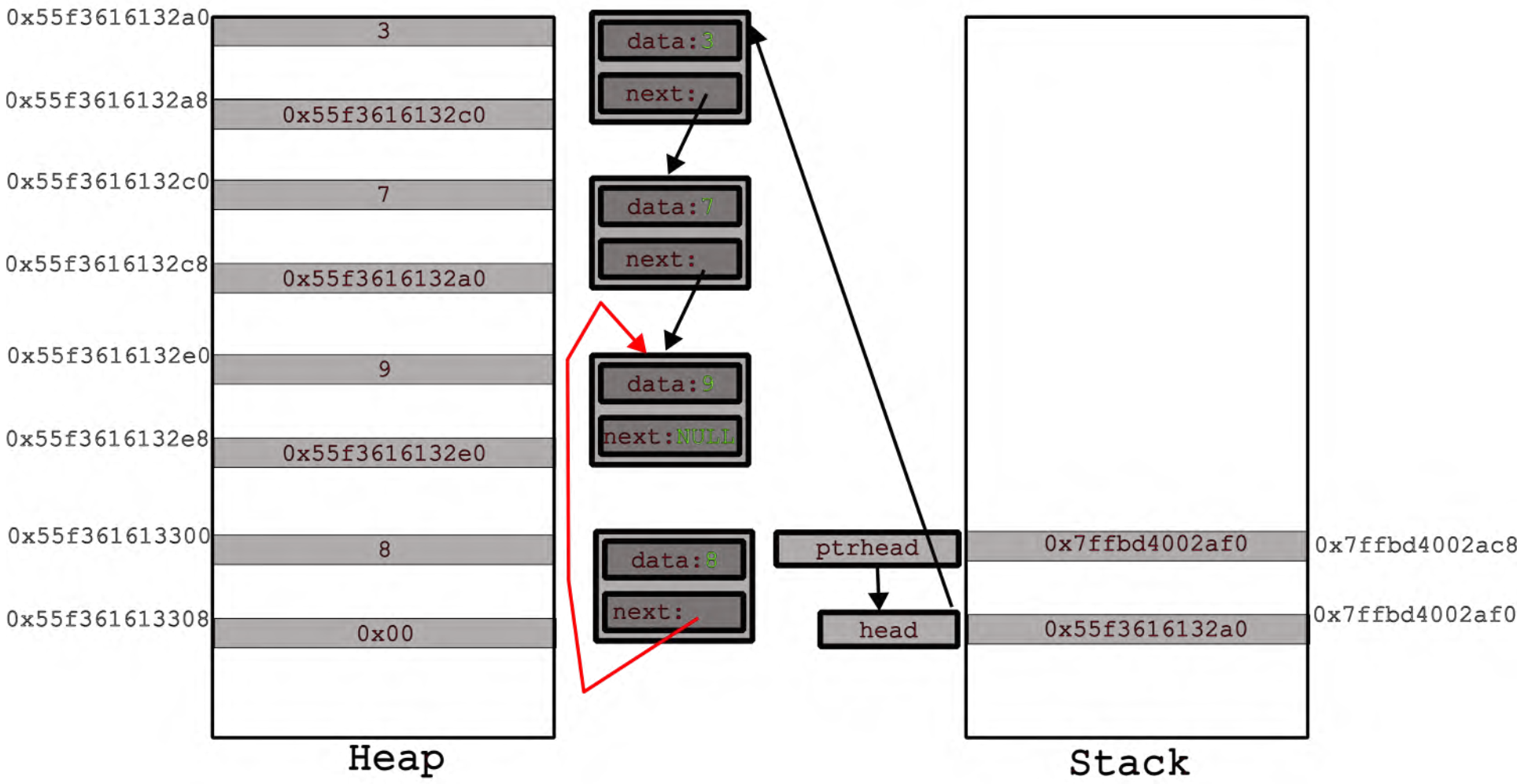


Insertar en cola

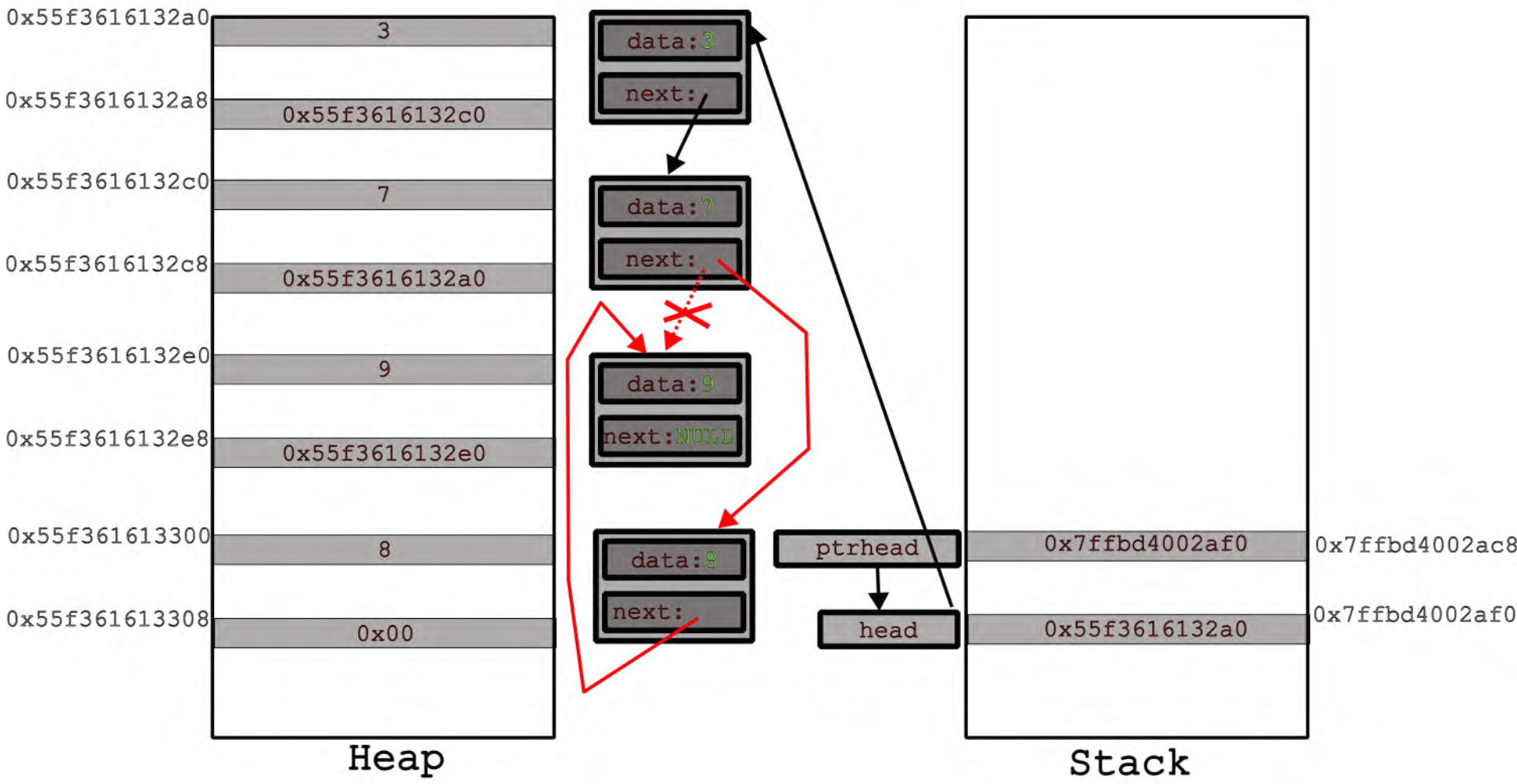
- Recorres hasta el último nodo, y su siguiente apunta al nuevo nodo.



Insertar en medio 1/2



Insertar en medio 2/2



Eliminar

- Depende dónde se elimine el nodo, se tendrá que modificar la cabeza de la lista *head*.
- La variable *head* es un puntero a la estructura *struct node* (tipo *ptrNode*), para poder modificarla, pasamos un puntero a ella *ptrhead*.
- Para eliminar correctamente, ¡debes liberar la memoria! Y luego, asignarle el valor *NULL*.

```
//ptr: dirección de memoria a liberar.
void free(void *ptr);
```

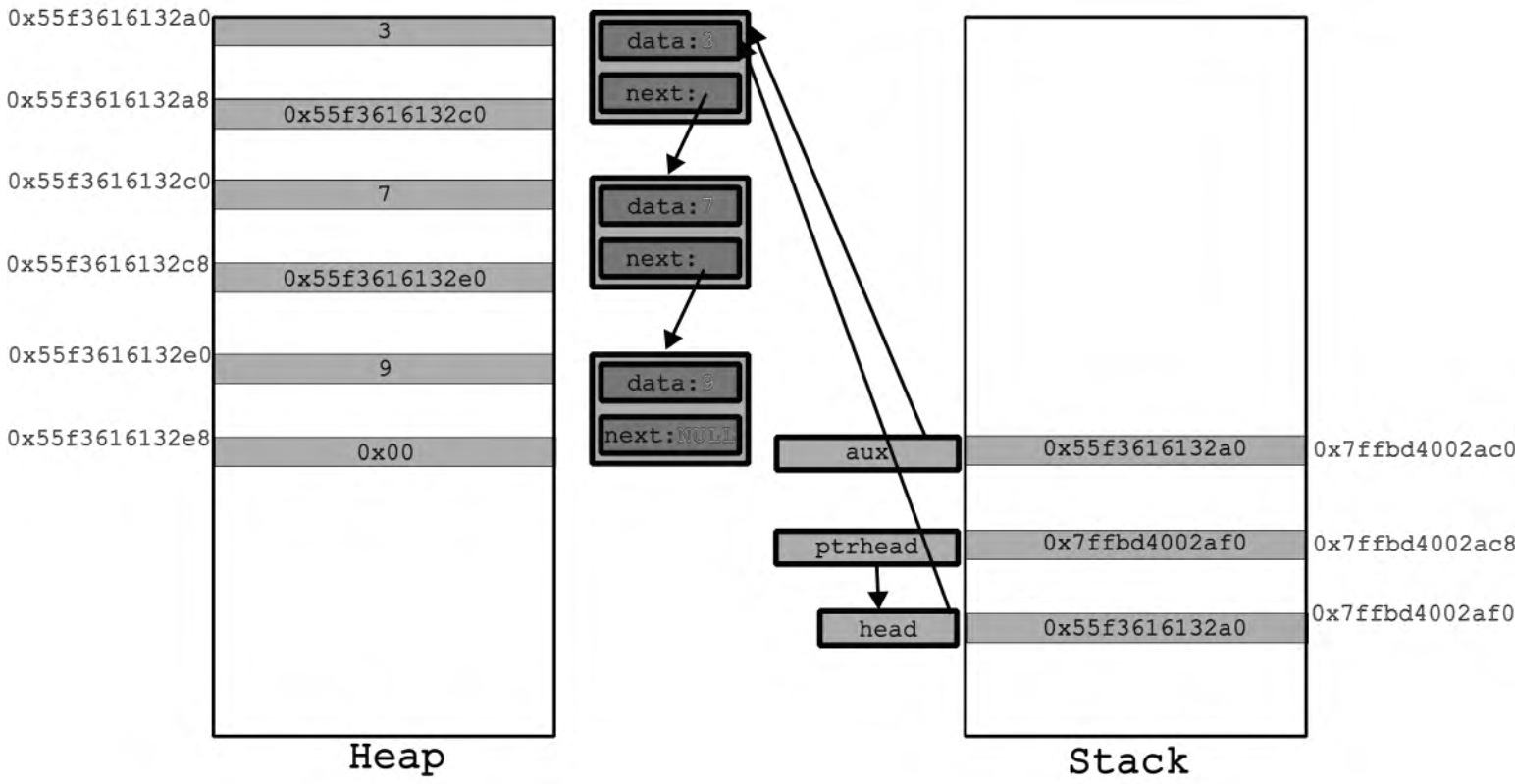
- (Importante) En estructuras en el que nodo tiene punteros a memoria dinámica (char *, int *, etc), debes liberar primero la memoria que tiene el nodo y luego el propio nodo.

Eliminar

```
typedef struct Node *ptrNode;
typedef struct Node{
    int data;
    ptrNode next;
} Node;

//Añade este método a ListaEnlazada.c. ptrhead es de tipo puntero a ptrNode;
//ptrhead is de tipo puntero a puntero a struct Node (struct Node ** ptrhead).
//data es el dato que contiene el nodo a eliminar
//en caso de haber más de uno, el primero encontrado.
bool eliminar(ptrNode * ptrhead, int data);
```

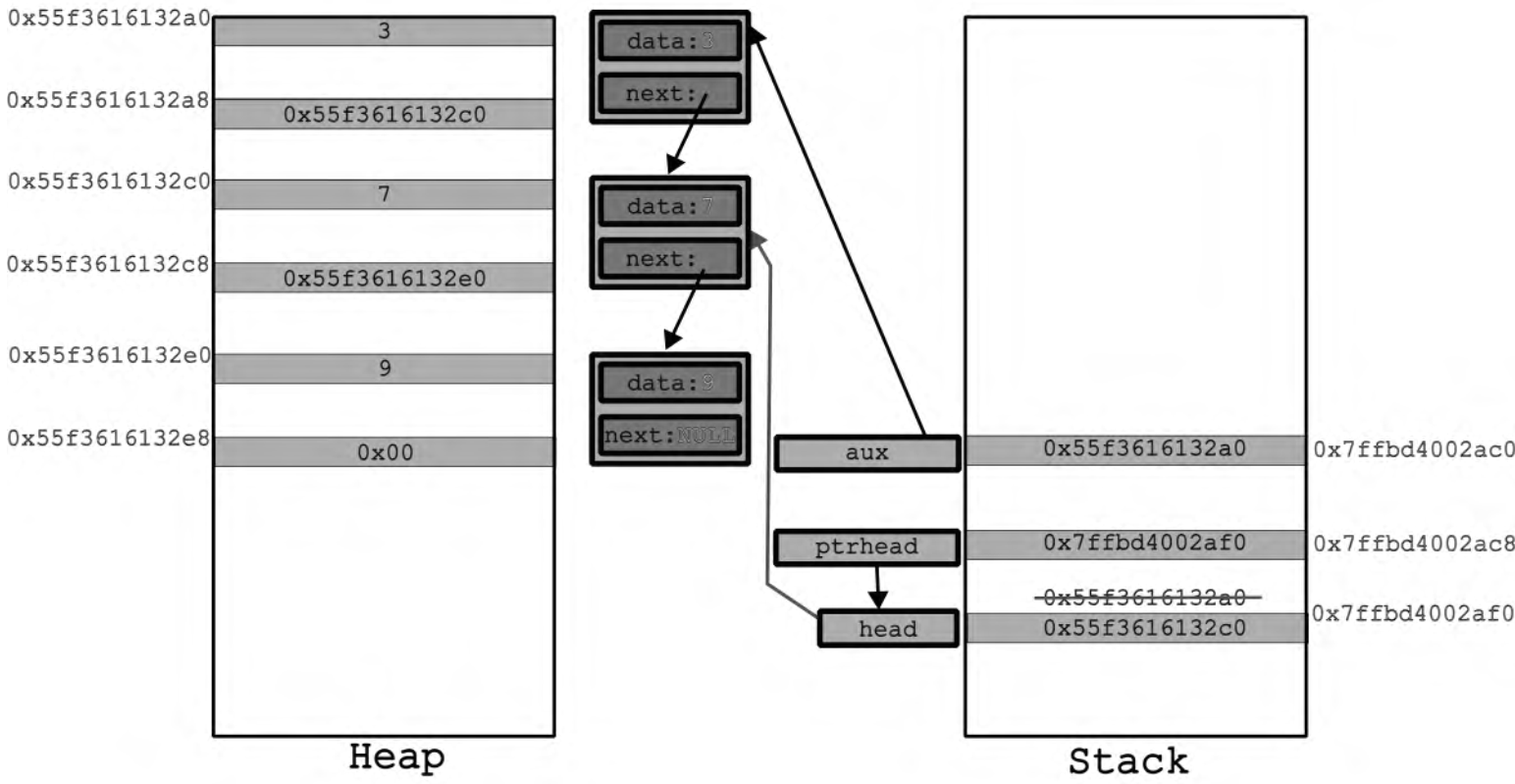
Elimina en cabeza 1/3

- Creamos un *aux* que apunta a *head*



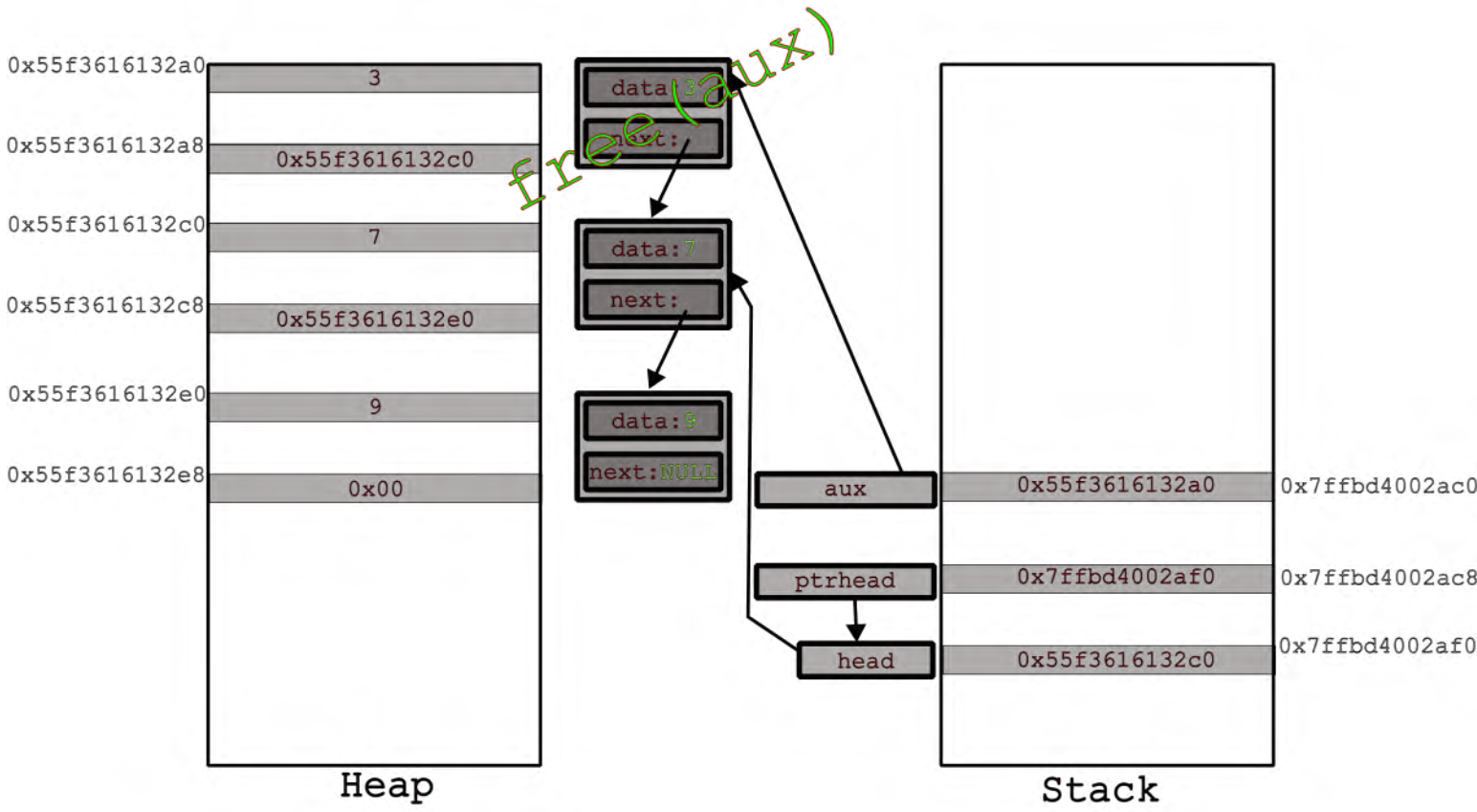
Elimina en cabeza 2/3

- Movemos *head* al siguiente elemento.



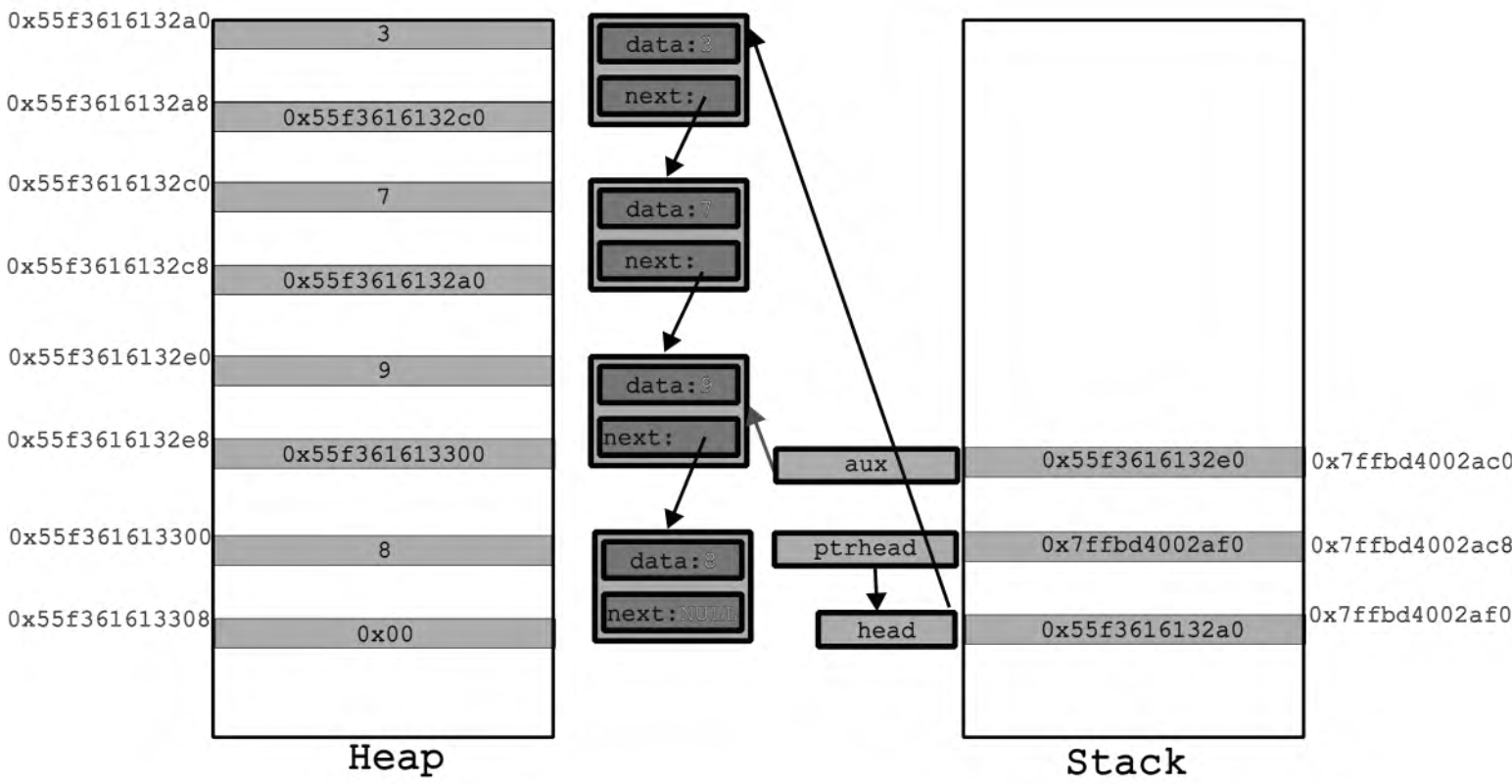
Elimina en cabeza 3/3

- Liberamos el primer elemento.



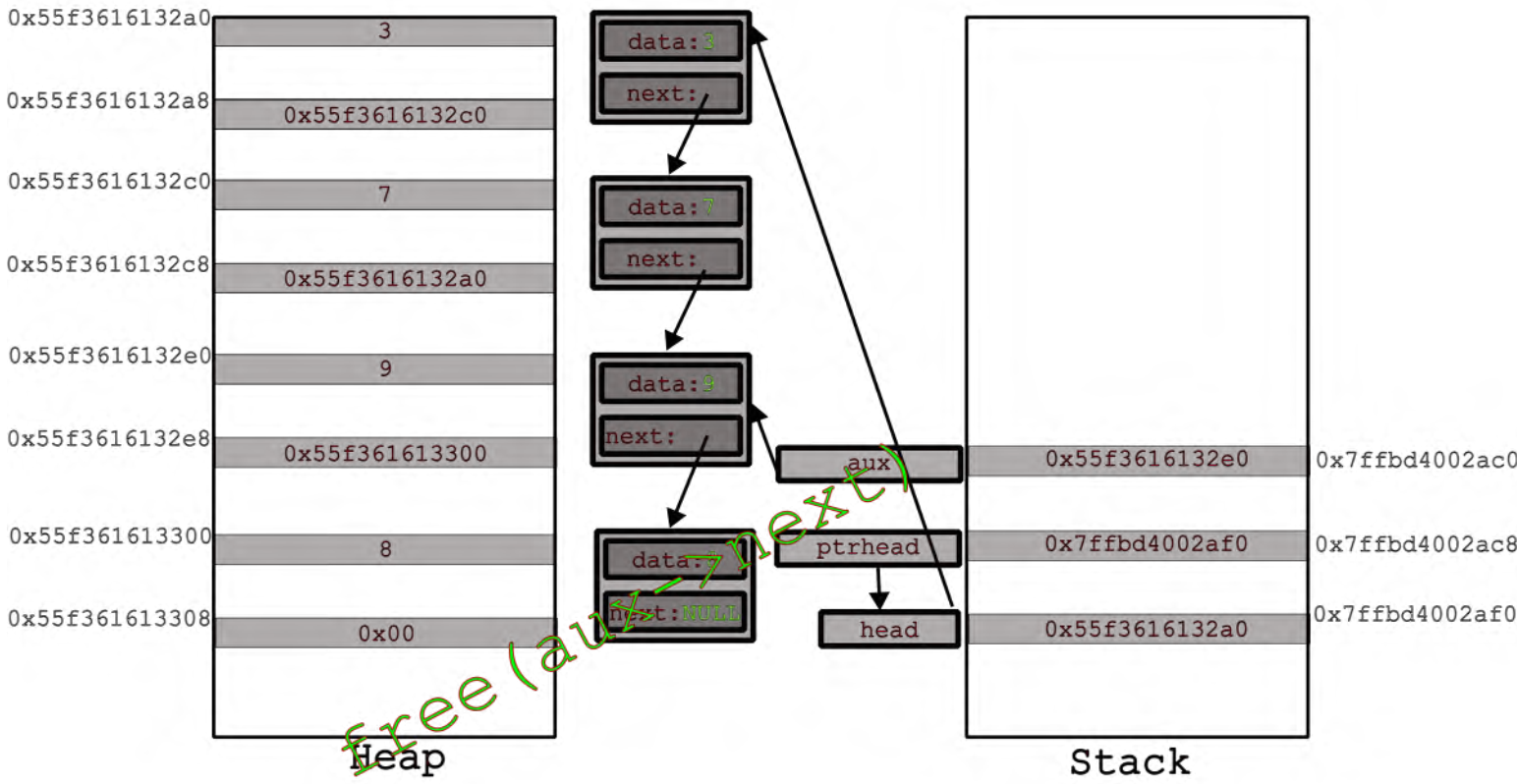
Elimina en cola 1/3

- Recorremos la lista y nos paramos en **el anterior al último**.



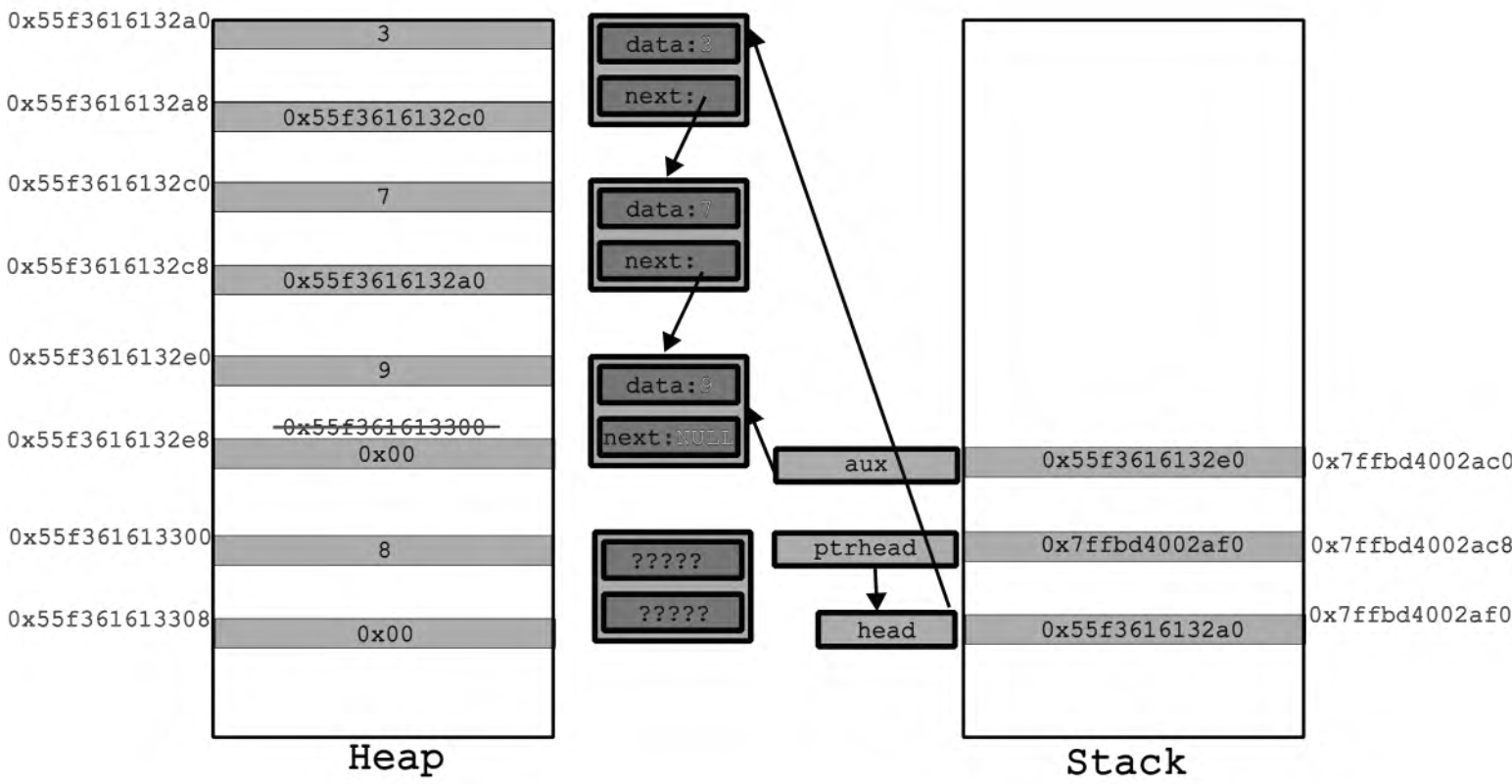
Elimina en cola 2/3

- Libera el aux cuando apunta al último.
- **Los punteros siguen apuntando a una dirección liberada** tras *free*



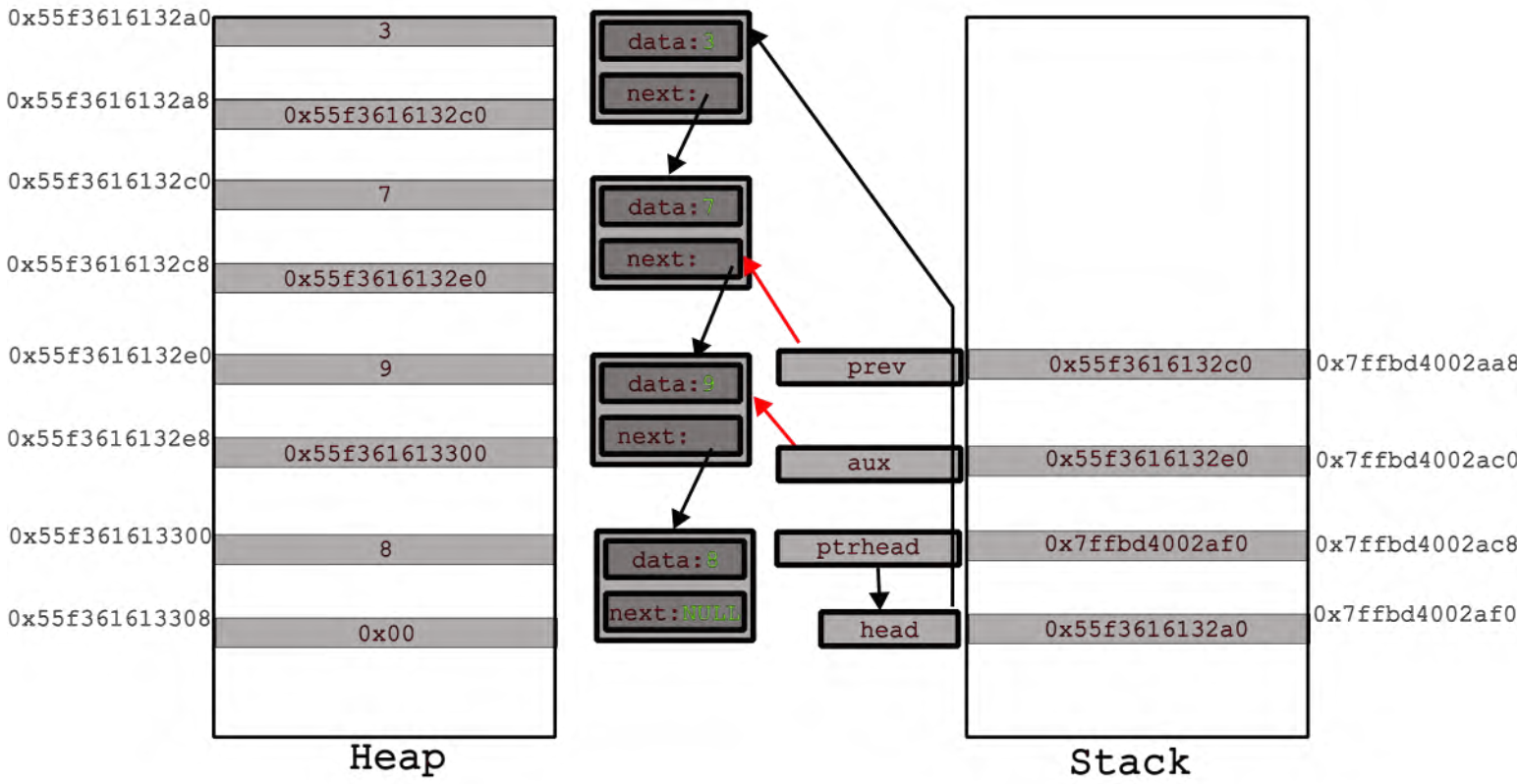
Elimina en cola 3/3

- Importancia de poner a *NULL*



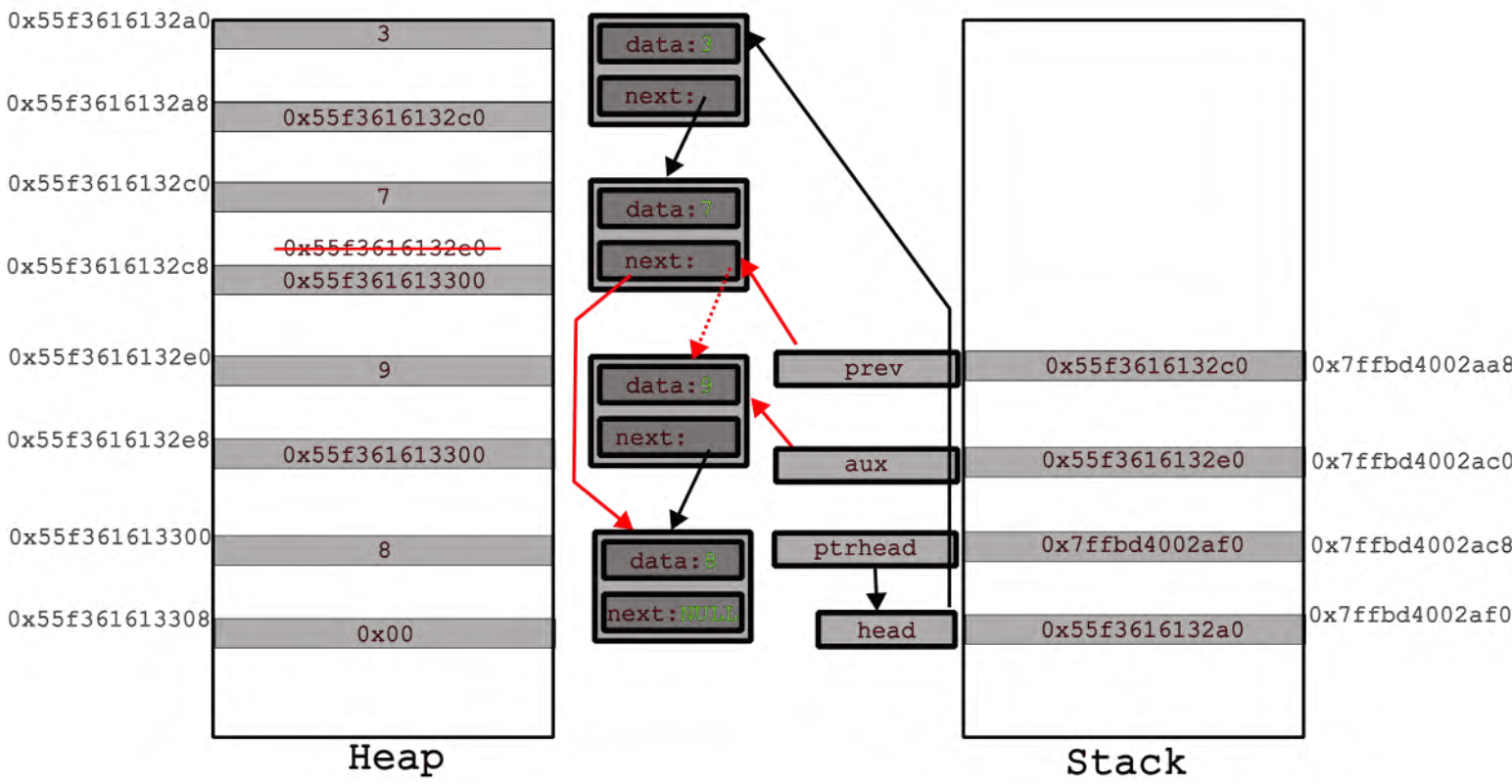
Elimina en medio 1/3

- Recorremos la lista y obtenemos un puntero **al anterior al elemento a eliminar**.



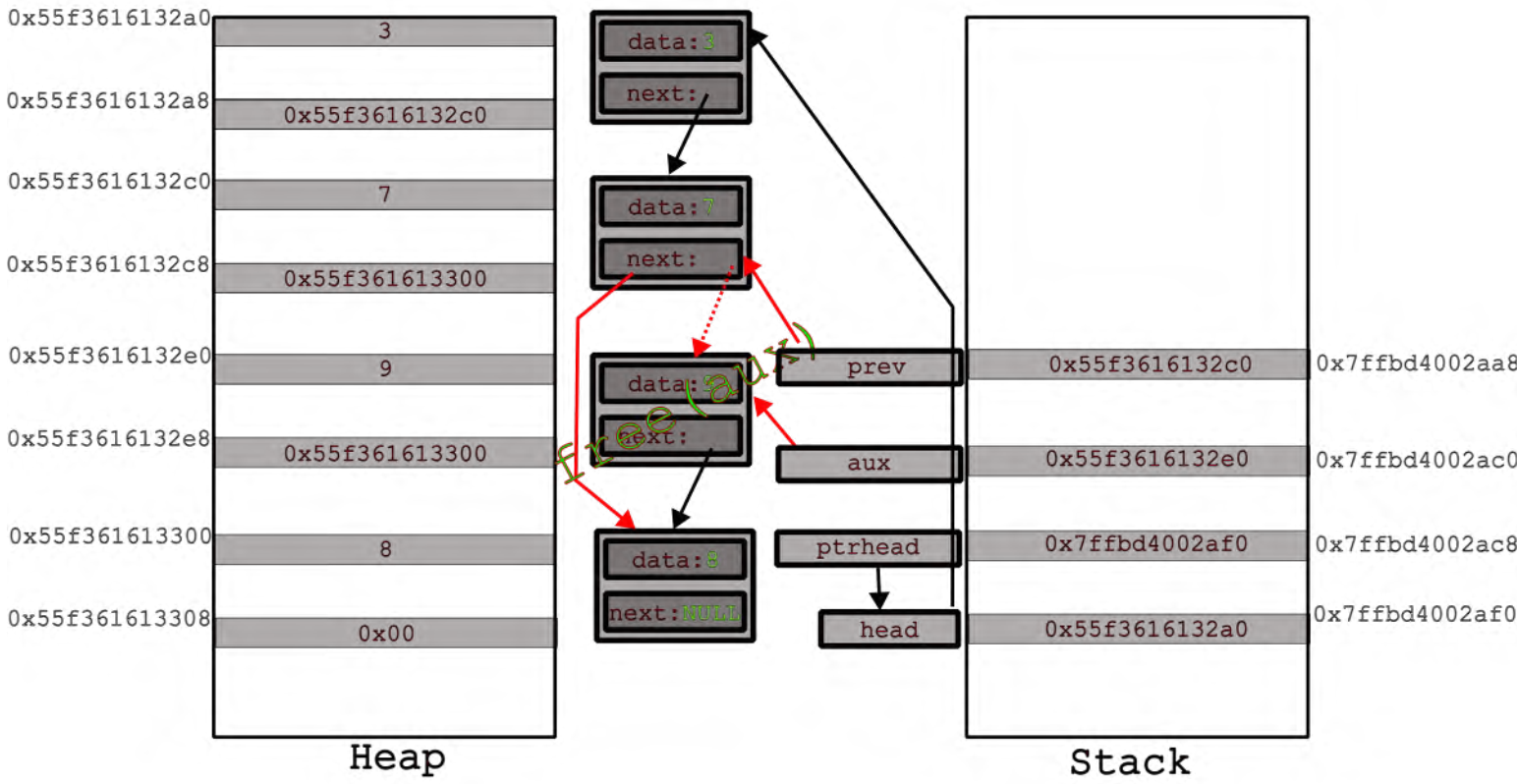
Elimina en medio 2/3

- El elemento anterior a borrar *prev* cambia su siguiente *next*



Elimina en medio 3/3

- Liberamos la memoria del nodo a eliminar. Si tuviera memoria dinámica el nodo, liberamos su contenido antes.



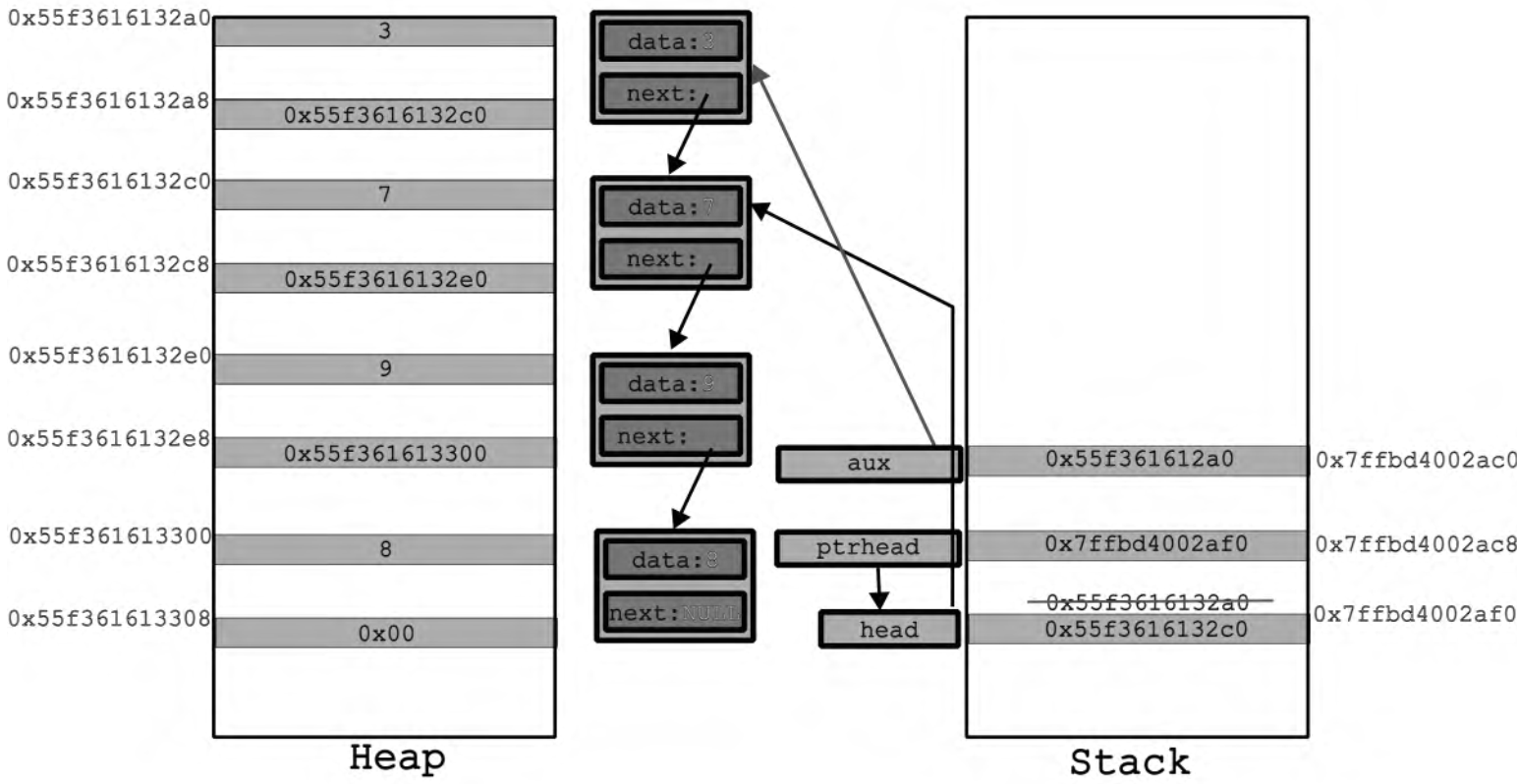
Destruir

- Cuando se destruye, *head* debe terminar valiendo *NULL*
- Se debe liberar toda la memoria dinámica:
  - Cada nodo ha sido creado con memoria dinámica y debe liberarse.
  - El contenido de cada nodo, **sólo si contiene memoria dinámica**

```
typedef struct Node *ptrNode;
typedef struct Node{
    int data;
    ptrNode next;
} Node;

//Añade este método a ListaEnlazada.c. ptrhead es de tipo puntero a ptrNode;
//ptrhead is de tipo puntero a puntero a struct Node (struct Node ** ptrhead).
//debe dejar ptrHead a NULL tras liberar
void destruir(ptrNode * ptrhead);
```

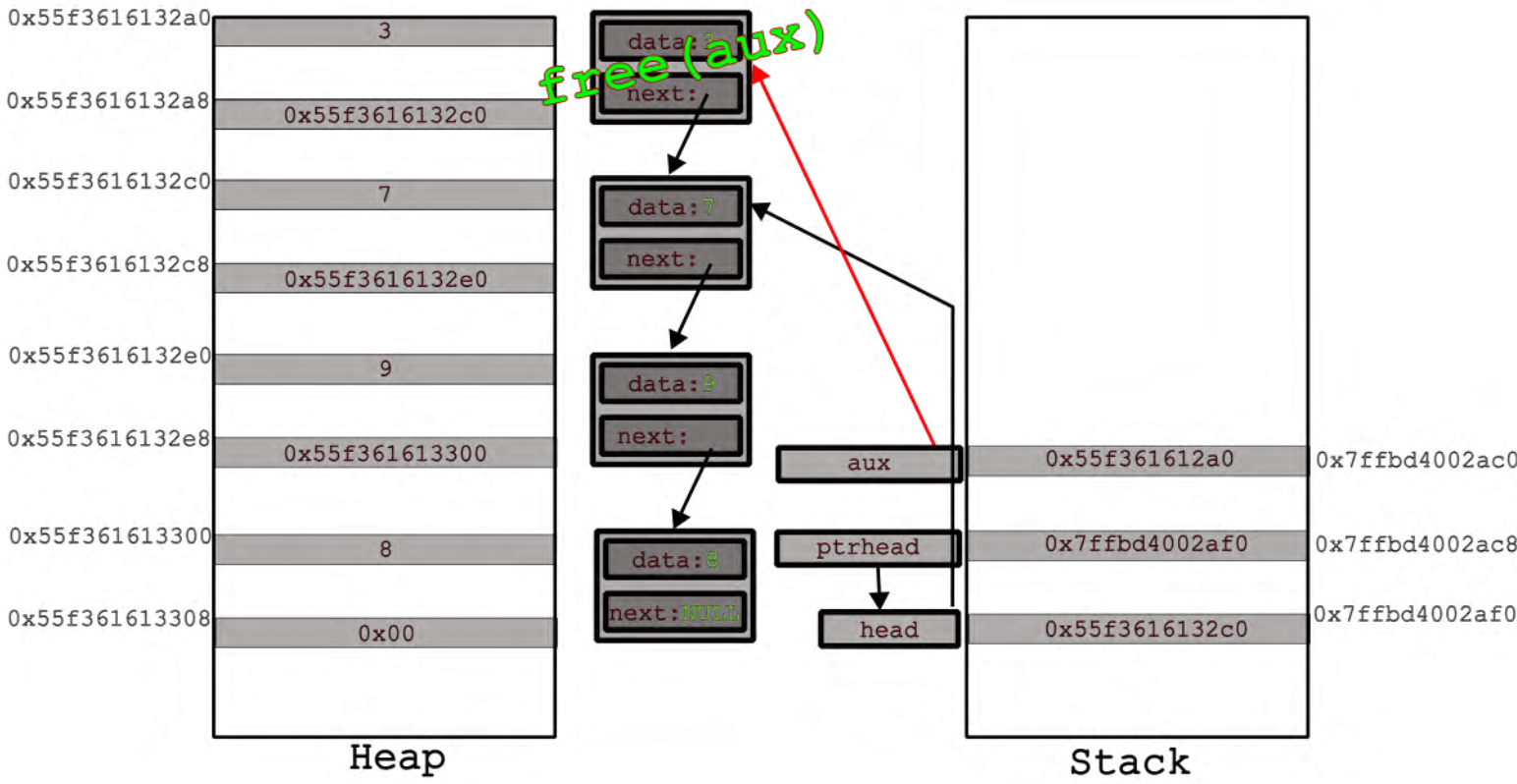
Destruir lista 1/3

- Usamos *head* para iterar.
- *aux* contendrá la dirección del elemento anterior.



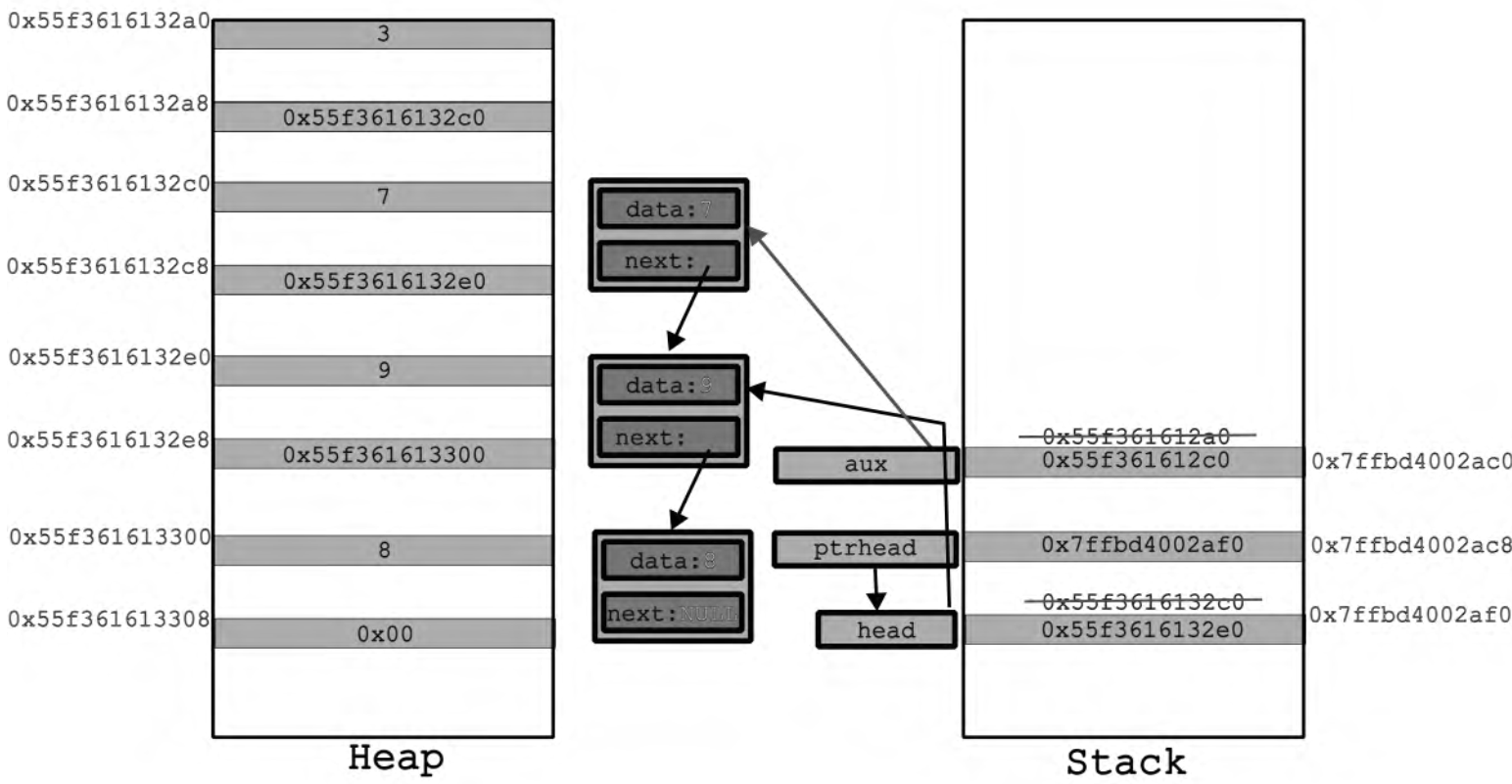
Destruir lista 2/3

- Liberamos *aux*



Destruir lista 3/3

- En cada iteración, almacenamos el nodo actual en *aux*, avanzamos *head* y ¡liberamos!+
- … y continuamos avanzando *head* hasta que sea *NULL*



Ejercicio con lista enlazada ordenada

- Inserción modificada:
  - Si el elemento es menor que *head* se inserta en cabeza.
  - Si el elemento es el mayor, se inserta en cola.
  - En otro caso, insertamos ordenado entre medio de los nodos que corresponda.
- Eliminar mejorado. Se puede parar la búsqueda si el elemento es menor que el nodo actual.
  Ejercicio: ListaEnlazadaOrdenada.c