

Implementación

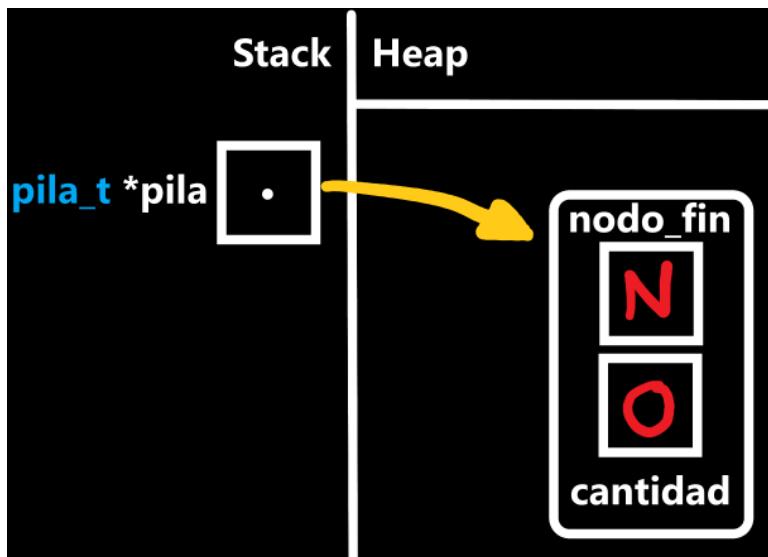
TDA Pila

```
struct _pila_t {
    nodo_t *nodo_fin;
    size_t cantidad;
};
```

Usé esta estructura para la pila porque no necesitaba un puntero al nodo_inicio. No castié la lista a una pila en ninguna de las funciones, hice todas de cero. Me dejó más libertad para organizar los nodos de otras maneras y todas las funciones de pila me quedaron en $O(1)$, excepto pila_destruir.

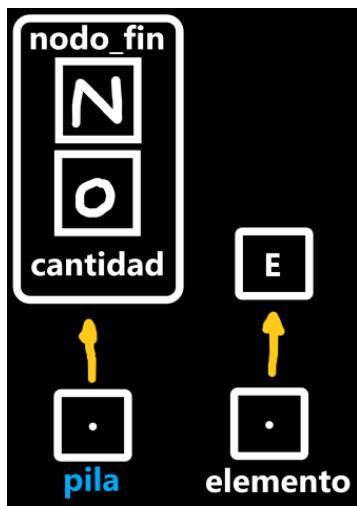
```
pila_t *pila_crear();
```

Reserva memoria con `calloc()` para el tamaño de una pila y devuelve un puntero a esa pila.

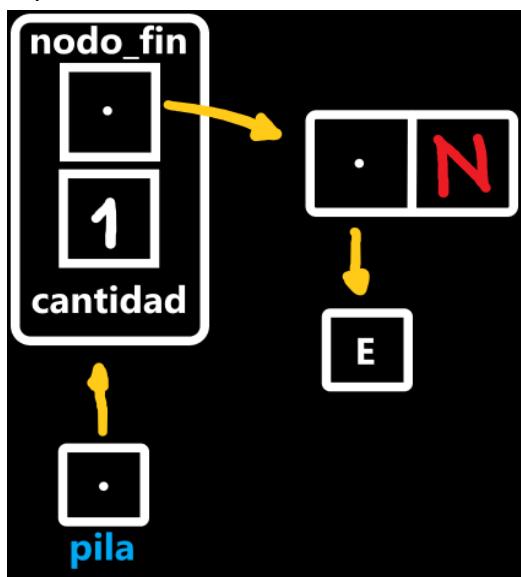


```
pila_t *pila_apilar(pila_t *pila, void *elemento);
```

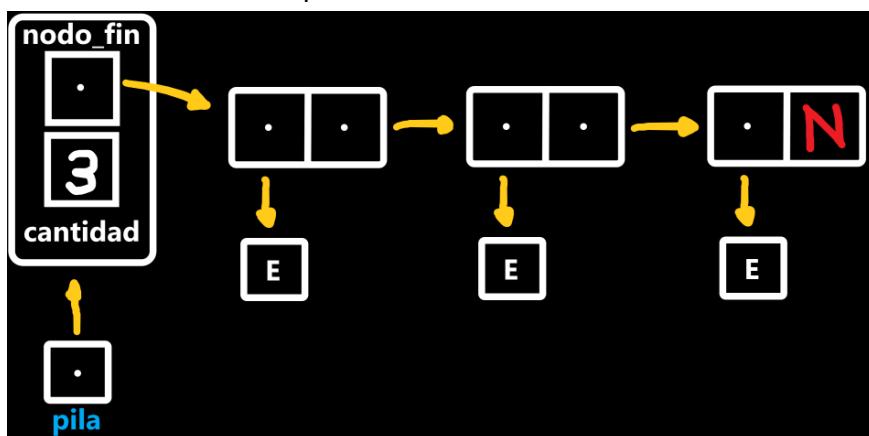
Esta función recibe un puntero a la pila creada por `pila_crear` y un elemento a introducir a la pila. Los punteros de pila y elemento se encuentran en el stack.



La función reserva memoria en el heap para almacenar una estructura del tamaño de un `nodo_t`. Apunta `pila->elemento` a la dirección del elemento pasado por parámetro, apunta `nodo->siguiente` a donde esté apuntando `nodo_fin` (siendo el primer nodo agregado sería `NULL`) y luego conecta el puntero `lista->nodo_fin` a ese nodo nuevo. Por último incrementa la `pila->cantidad` en 1.



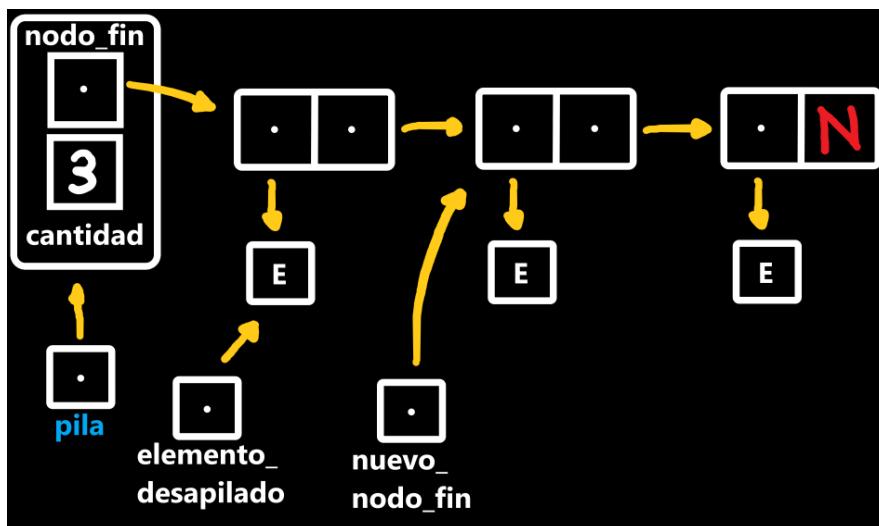
En esta foto se ve una pila con 3 elementos



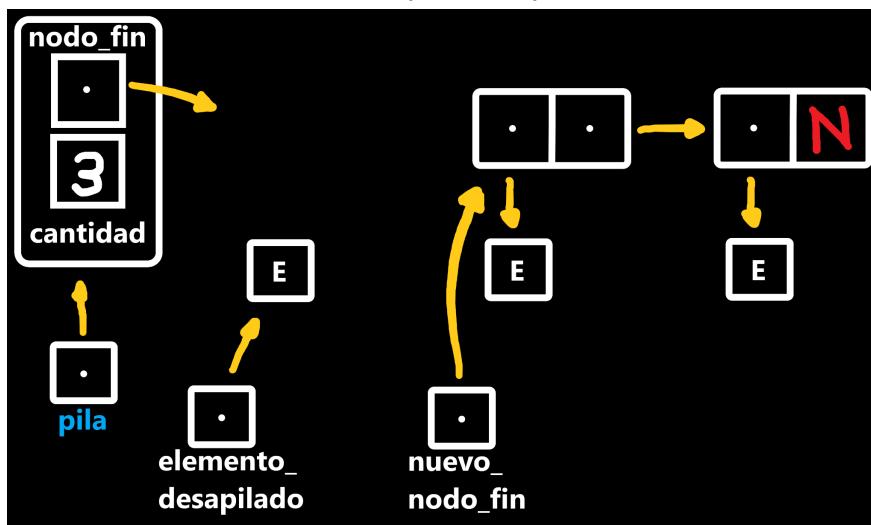
Esta función devuelve un puntero a la pila luego de ser alterada. Yo ordene la pila de manera que los nodos apunten a sus antecesores, esto ayuda a la hora de hacer cambios en la pila porque el nodo que quiero alterar es siempre el nodo apuntado por `nodo_fin`.

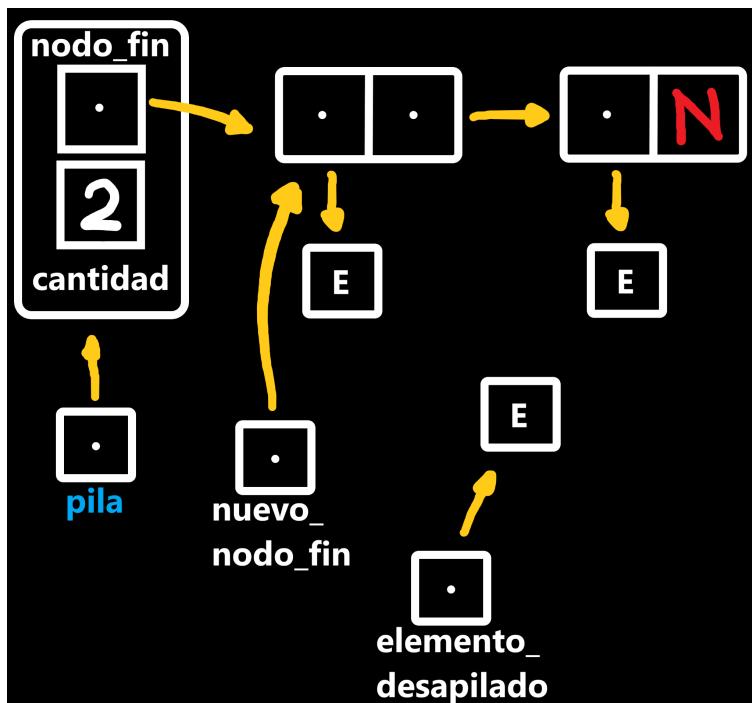
```
void *pila_desapilar(pila_t *pila);
```

Recibe por parámetro un puntero a la pila, primero verifica que la pila no esté vacía, de ser el caso devuelve NULL. Crea un puntero a nodo que apunta al nodo siguiente a `nodo_fin` y crea también un puntero void para apuntar al elemento que será borrado y necesita devolver.



Hace `free(pila->nodo_fin)`, esto borra el último nodo de la pila. Apunta `pila->nodo_fin` a donde apunta `nuevo_nodo_fin` y disminuye la cantidad en 1.





```
void *pila_tope(pila_t *pila);
```

Devuelve un puntero al elemento al que apunta el último nodo de la pila, sería el elemento al que apunta `pila->nodo_fin->elemento`.

```
size_t pila_tamanio(pila_t *pila);
```

Devuelve el estado actual de `pila->cantidad`.

```
bool pila_vacia(pila_t *pila);
```

Devuelve true si `pila->cantidad == 0` o si la pila no existe, sino false.

```
void pila_destruir(pila_t *pila);
```

Esta función libera el espacio reservado en el heap por `pila_crear` y `pila_apilar`. Mientras `pila->nodo_fin` no apunte a NULL se itera sobre la pila para liberar todos los nodos en orden, empezando por el último nodo y terminando por el primero. Hace `free(pila->nodo_fin)` y luego apunta `pila->nodo_fin` al siguiente nodo con la ayuda de un puntero auxiliar. Al terminar de borrar todos los nodos libera la memoria de la pila. Básicamente funciona como `pila_desapilar` pero no devuelve nada y no edita `pila->cantidad` al borrar los nodos. La complejidad de la función es de $O(n)$, $n =$ la cantidad de nodos que tenga la pila.

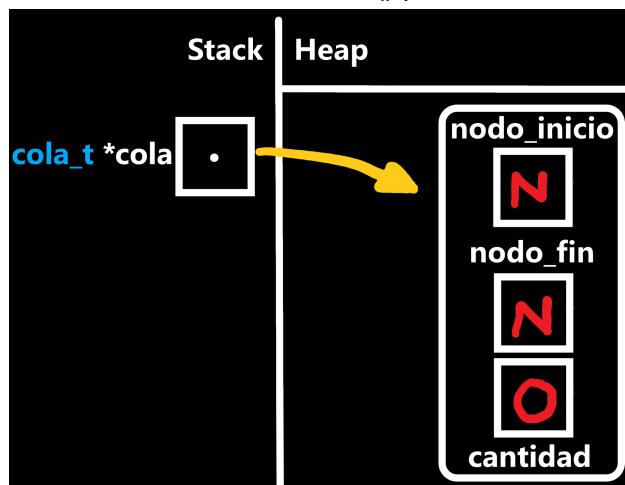
TDA Cola

```
struct _cola_t {
    nodo_t *nodo_fin;
    nodo_t *nodo_inicio;
    size_t cantidad;
};
```

Use esta estructura para la cola, es igual a la estructura de lista. No castie la lista a una cola en ninguna de las funciones, hice todas de cero. La complejidad de todas las funciones quedaron en $O(1)$ exceptuando `cola_destruir`.

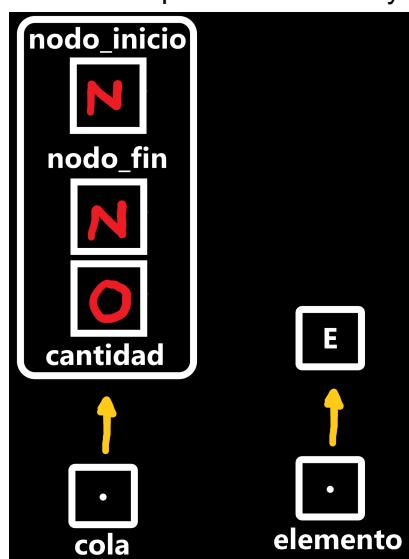
```
cola_t *cola_crear();
```

Reserva memoria con `calloc()` para el tamaño de una cola y devuelve un puntero a esa cola.

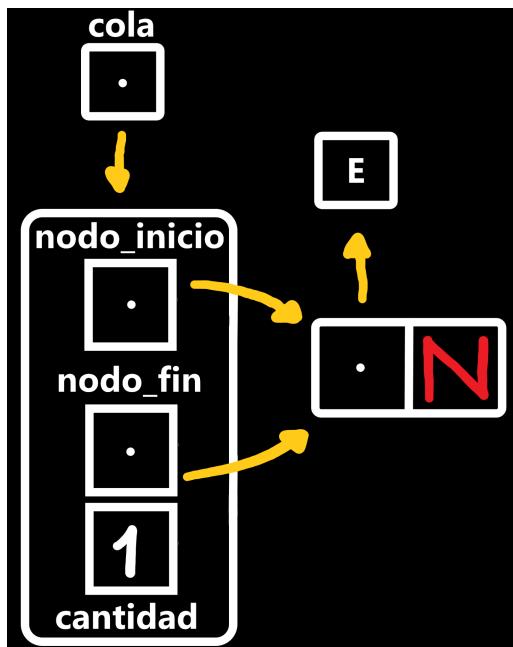


```
cola_t *cola_encolar(cola_t *cola, void *elemento);
```

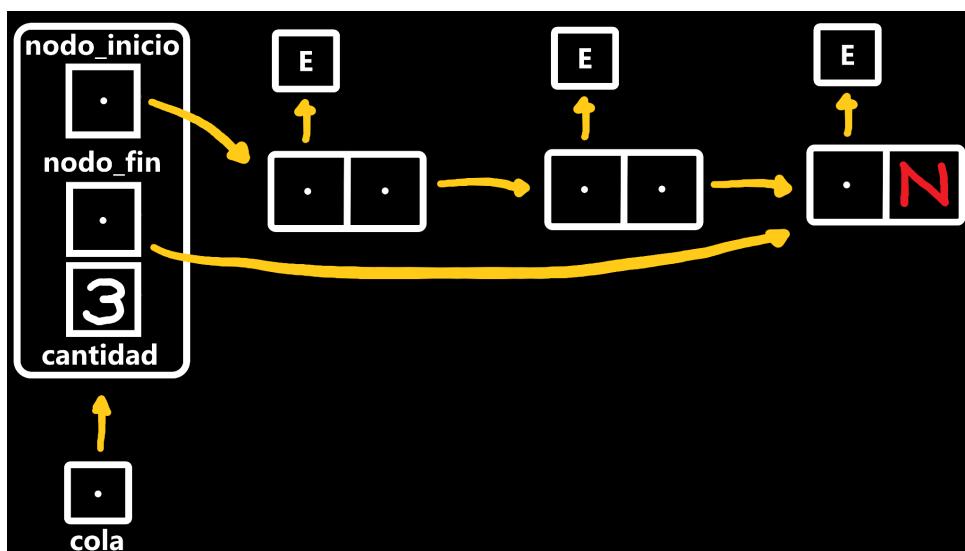
Esta función recibe un puntero a la cola creada por `cola_crear` y un elemento a introducir a la cola. Los punteros de cola y elemento se encuentran en el stack.



Se crea un puntero a un nodo reservado con `calloc` y se le asigna el elemento pasado por parámetro, si es el primer nodo de la cola entonces `cola->nodo_inicio` apuntará a este nuevo nodo. Sino, el siguiente al último nodo apunta al nuevo nodo (se conecta a la cola). Por último `cola->nodo_fin` apuntará a este nuevo nodo y se incrementa `cola->cantidad` en 1.



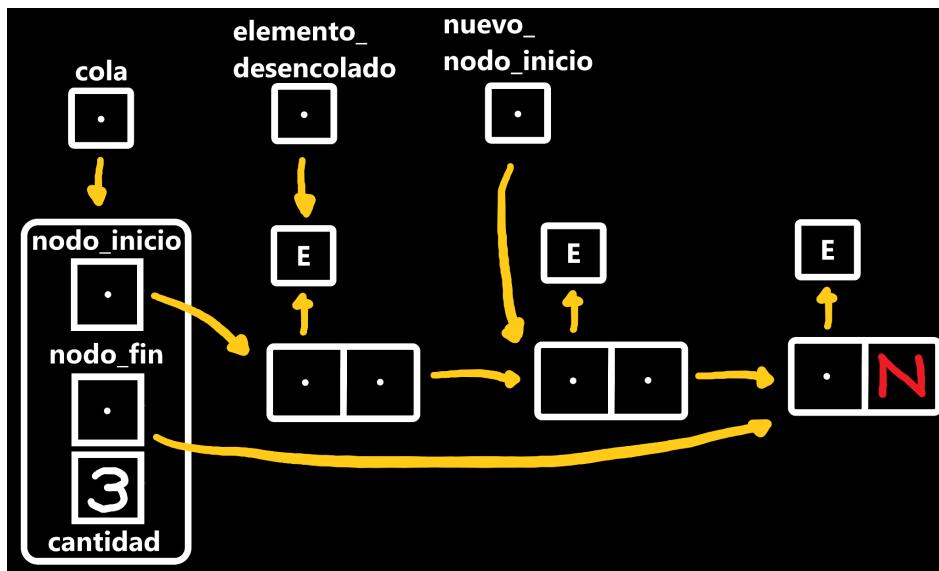
En esta foto se ve una cola con 3 elementos.



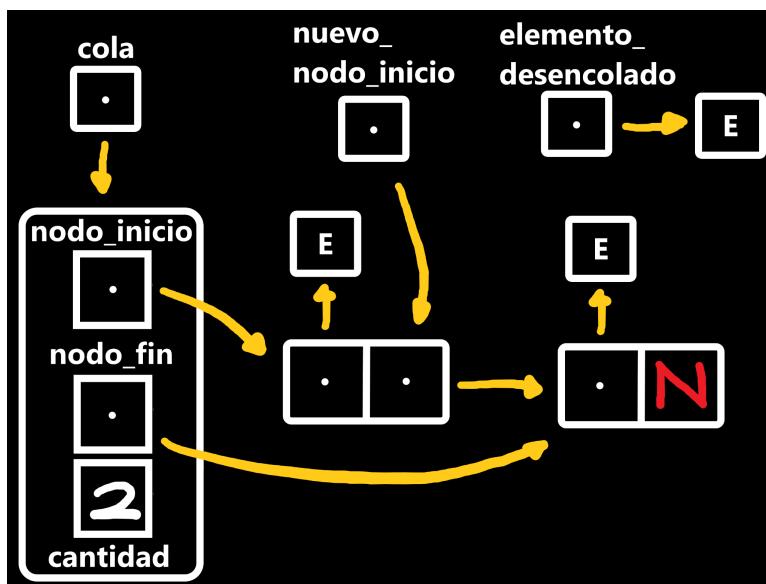
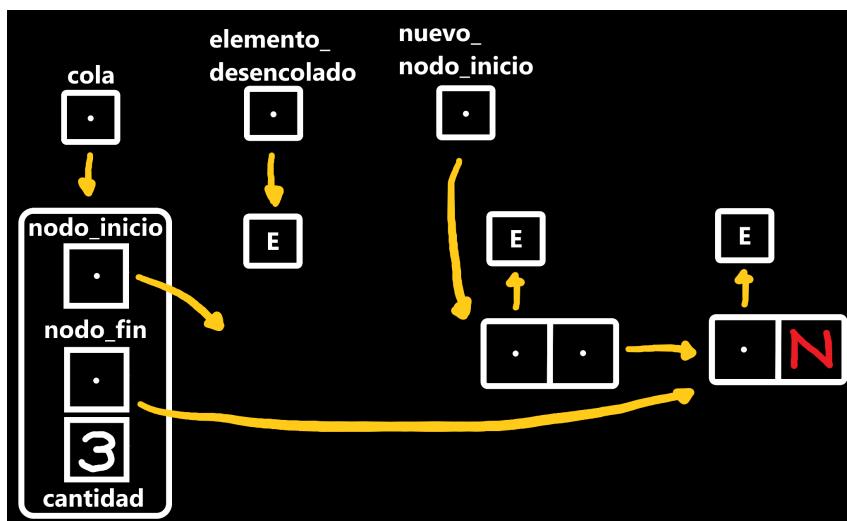
Esta función devuelve un puntero a la cola luego de ser alterada. A diferencia de mi TDA pila, organízale la cola para que cada nodo apunte a su sucesor, esto hace que liberar nodos sea $O(1)$.

```
void *cola_desencolar(cola_t *cola);
```

Recibe por parámetro un puntero a la cola, primero verifica que la cola no esté vacía, de ser el caso devuelve NULL. Crea un puntero a nodo que apunta al nodo siguiente a `nodo_inicio` y crea también un puntero void para apuntar al elemento que será borrado y necesita devolver.



Esta función funciona muy parecido a pila_desapilar, libera el nodo apuntado por `cola->nodo_inicio`, apunta `nodo_inicio` a `nuevo_nodo_inicio` y disminuye `cola->cantidad` en 1.



```
void *cola_frente(cola_t *cola);
```

Devuelve un puntero al primer elemento de la cola, sería un puntero a `cola->nodo_inicio->elemento`.

```
size_t cola_tamano(cola_t *cola);
```

Devuelve `cola->cantidad`, de ser NULL el puntero a `cola` devuelve 0.

```
bool cola_vacia(cola_t *cola);
```

Devuelve true si `cola->cantidad == 0` o si el puntero a `cola` es NULL, devuelve false si no se cumplen ninguna de ellas.

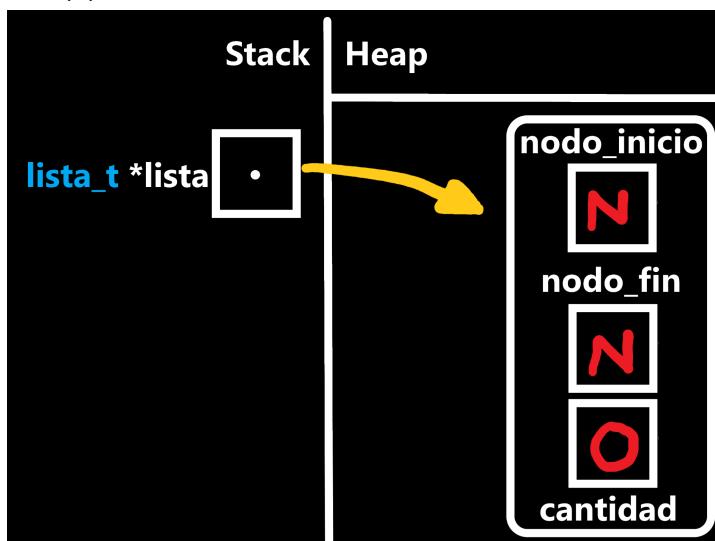
```
void cola_destruir(cola_t *cola);
```

Esta función libera el espacio reservado en el heap por `cola_crear` y `cola_encolar`. Mientras `cola->nodo_inicio` no apunte a NULL itera sobre la cola para liberar todos los nodos en orden, empezando por el primer nodo y terminando por el último. Hace `free(cola->nodo_inicio)` y luego apunta `cola->nodo_inicio` al siguiente nodo con la ayuda de un puntero auxiliar. Al terminar de borrar todos los nodos libera la memoria de la cola. La complejidad de la función es de $O(n)$, $n =$ la cantidad de nodos que tenga la cola.

TDA Lista

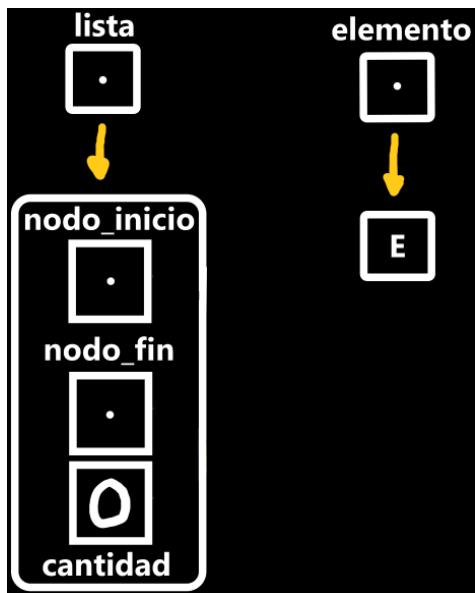
```
lista_t *lista_crear();
```

Reserva memoria con `calloc()` para el tamaño de una lista y devuelve un puntero a esa lista. Es $O(1)$.

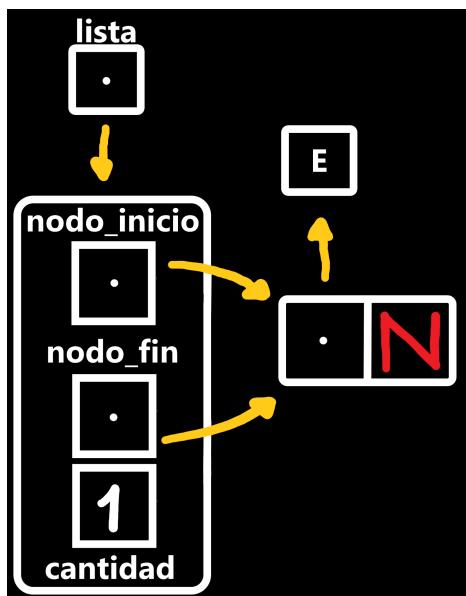


```
lista_t *lista_insertar(lista_t *lista, void *elemento);
```

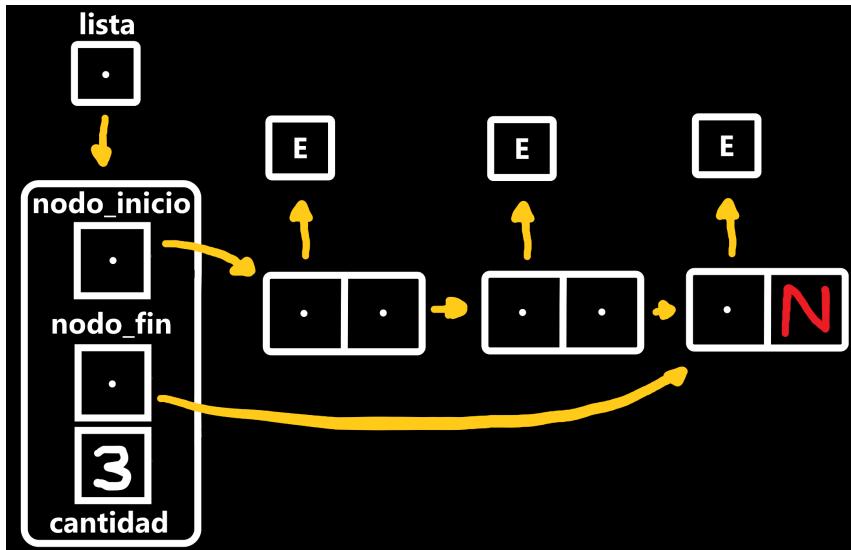
Esta función recibe un puntero a la lista creada por `lista_crear` y un elemento a introducir a la lista. Los punteros de `lista` y `elemento` se encuentran en el stack. Es $O(1)$.



Se crea un puntero a un nodo reservado con malloc y se le asigna el elemento pasado por parámetro, si es el primer nodo de la lista entonces lista->nodo_inicio apuntará a este nuevo nodo. Sino, el siguiente al último nodo apunta al nuevo nodo (se conecta a la lista). Por último lista->nodo_fin apuntará a este nuevo nodo y se incrementa lista->cantidad en 1.



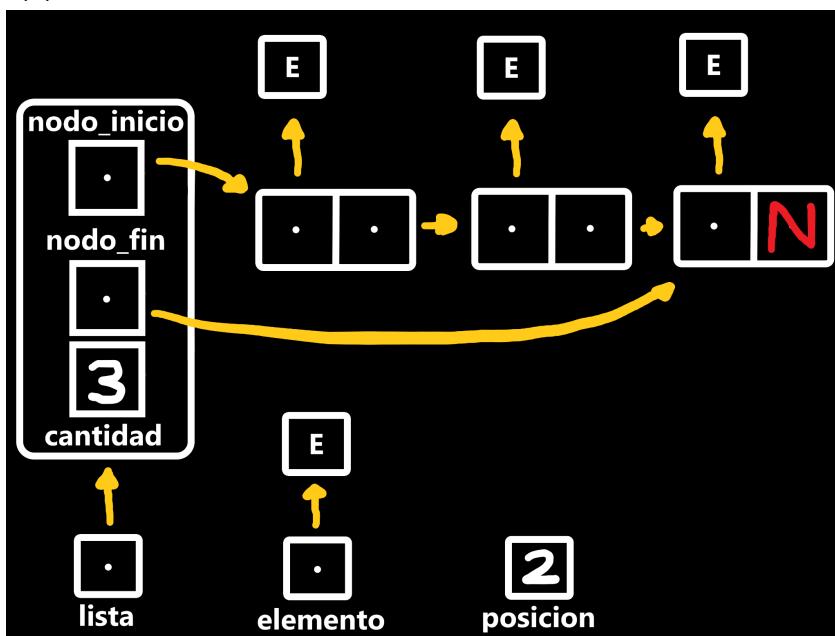
En esta foto hay una lista con 3 elementos.



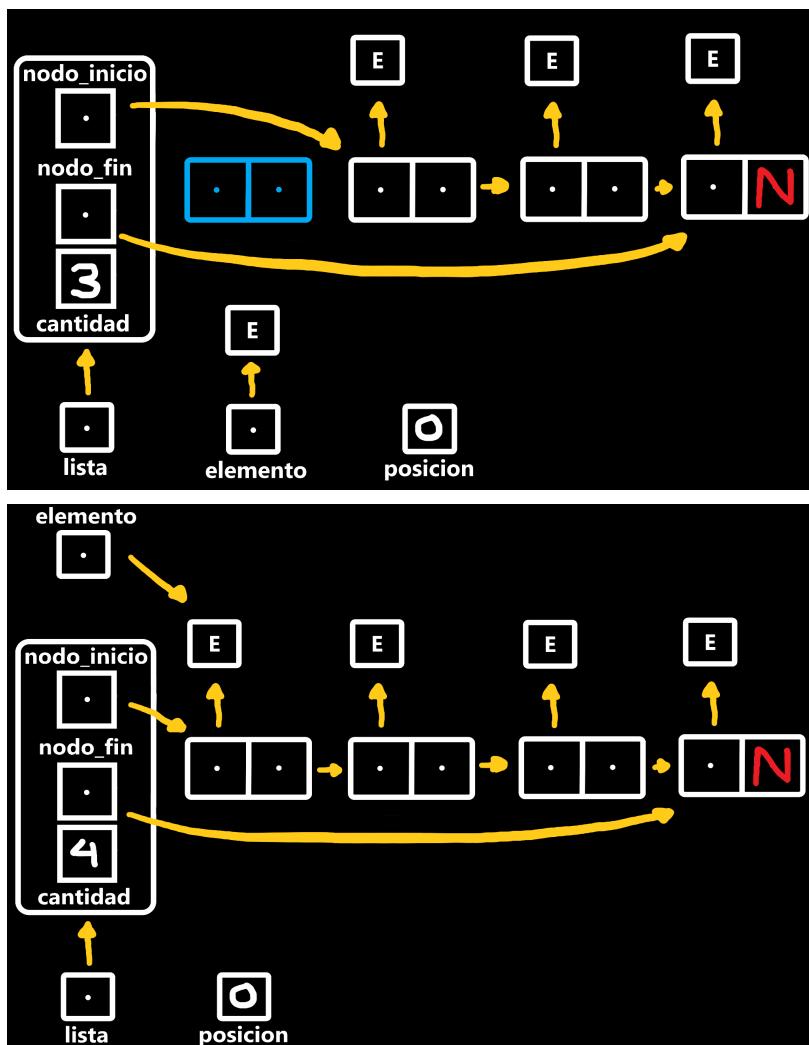
Esta función devuelve un puntero a la lista luego de ser alterada. Implemente la función de manera que organiza la lista de manera que cada nodo apunte a su sucesor. Algo que me pasó fue que en principio organice la lista como organice la pila, cada nodo apuntando a su antecesor. Esto me dejaba borrar el último elemento de la lista con una complejidad de $O(1)$ pero la razón por la que cambie la implementación fue que más tarde a la hora de usar el iterador externo y querer navegar los nodos, este recorrería la lista al revés.

```
lista_t *lista_insertar_en_posicion(lista_t *lista, void *elemento,
                                    size_t posicion);
```

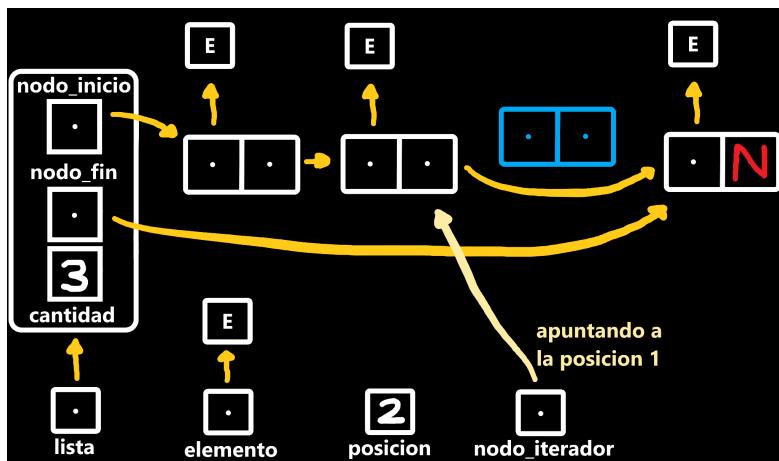
Esta función recibe un puntero a la lista, otro puntero al elemento que se quiere añadir a la lista y un número que representa la posición en la que quiero agregar este elemento. Es $O(n)$.



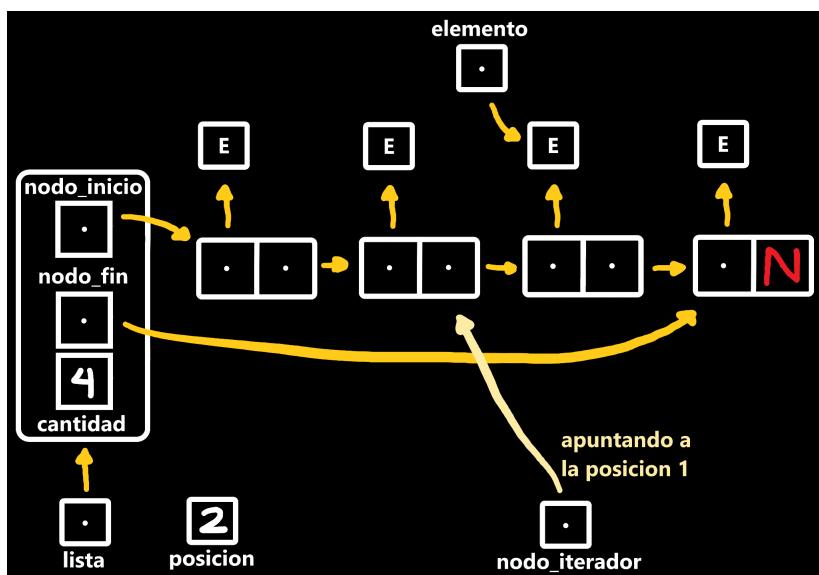
La función primero se fija que la posición enviada no sea equivalente a agregar un elemento al final de la lista, en ese caso llama a la función lista_insertar y devuelve la lista ya editada. En caso de que la posición sea una número que sea en este caso < 3, reserva memoria con malloc para agregar un nodo nuevo y recibe un puntero a ese nodo, le asigna el elemento pasado por parámetro y luego pueden pasar 2 cosas. Si la posición enviada es 0 (el usuario quiere agregar el elemento al principio de la lista) entonces apunta el nodo nuevo al primer nodo de la lista, hace apuntar nodo_inicio al nodo nuevo y luego suma 1 a la cantidad total de elementos en la lista.



Si la posición esta en el medio de la lista, entonces se crea un puntero a `nodo_t` que va a iterar en la lista hasta encontrarse apuntando al nodo anterior de donde quiero agregar el nuevo nodo.



Luego hago apuntar el nuevo nodo al siguiente del nodo apuntado por `nodo_iterador` (en este caso `nodo_nuevo` apunta al último nodo de la lista), apunta el nodo apuntado por `nodo_iterador` al nuevo nodo (el nuevo nodo fue agregado a la lista) y por último suma 1 a `lista->cantidad`.

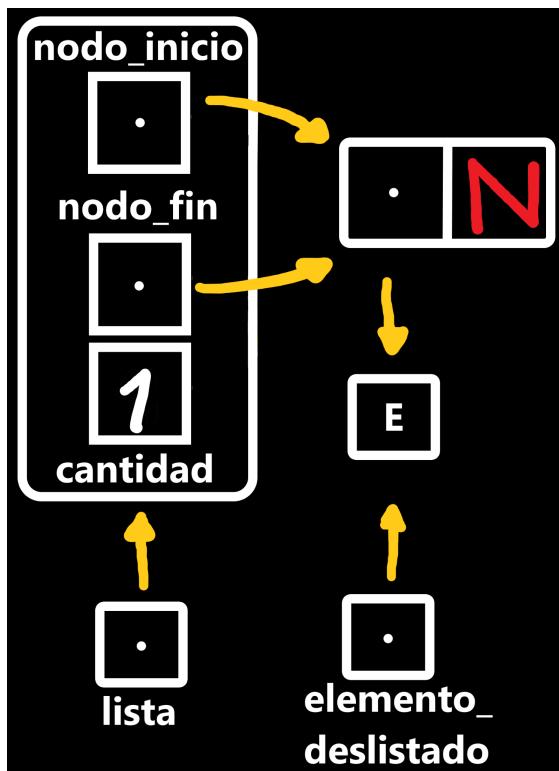


La complejidad es $O(n)$ porque para agregar un elemento en cualquier lugar de la lista es necesario iterar de alguna manera y recorrer la lista n veces, siendo $n = \text{posición} - 1$.

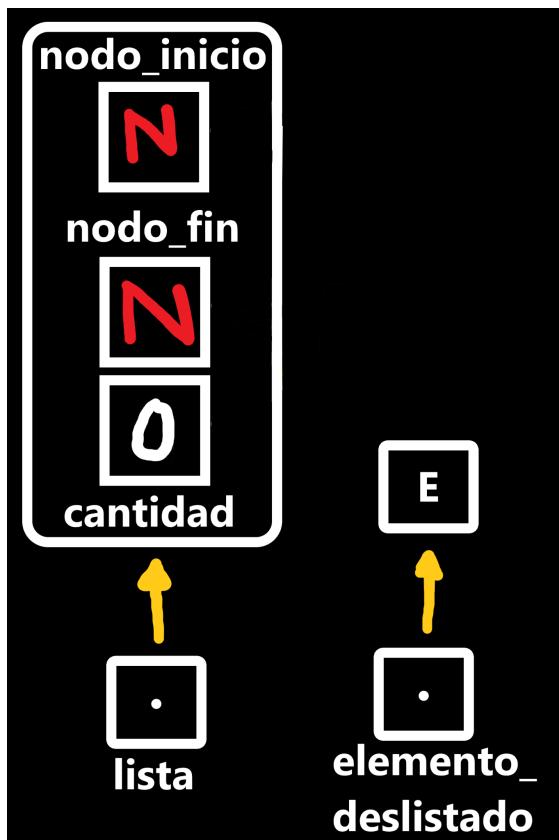
```
void *lista_quitar(lista_t *lista);
```

Recibe por parámetro un puntero a la lista, primero verifica que la lista no esté vacía, de ser el caso devuelve NULL. Si la lista tiene un elemento en total crea un puntero void y lo apunta al único elemento existente, libera el nodo y luego apunta los punteros `nodo_inicio` y `nodo_fin` a NULL, por último disminuye la cantidad de la lista por 1 (queda en 0).

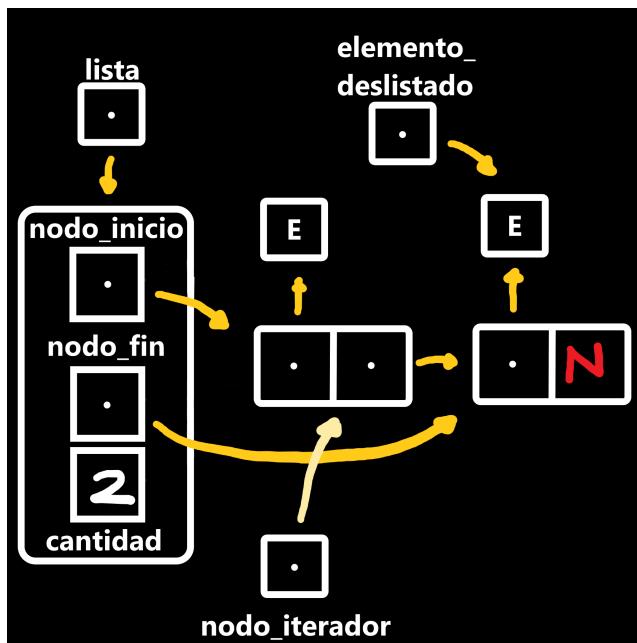
En este caso hay un solo elemento en la lista.



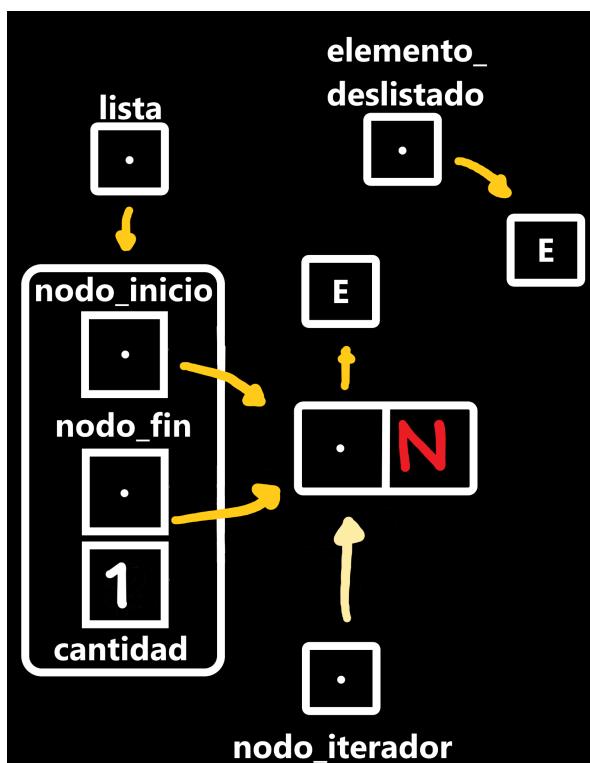
En esta foto el nodo ya fue eliminado. Se devuelve elemento_deslistado.



Ahora un ejemplo que muestre cómo quitar un nodo de la lista cuando hay más de un nodo.



En este caso la función crea un puntero iterador e irá moviéndose hasta llegar al ante último nodo, en este caso es el nodo de la posición 0. Luego se guarda el elemento que se va a devolver en `elemento_deslistado` y se libera la memoria ocupada por el nodo. Por último `nodo_fin` apunta al nodo anterior, el que es ahora el último nodo apunta a NULL y se disminuye la `lista->cantidad` en 1.



La función tiene una complejidad de $O(n)$ porque en una lista con más nodos debo iterar hasta encontrar el ante último nodo, $n = \text{lista}->\text{cantidad} - 2$. Es posible hacer este procedimiento con una complejidad de $O(1)$, la implementé y tuve que cambiarla como explique antes ya que el iterador externo no hubiera funcionado acordemente. Posiblemente lo podría haber hecho funcionar pero recorrer la lista en orden al estar dada vuelva hubiera

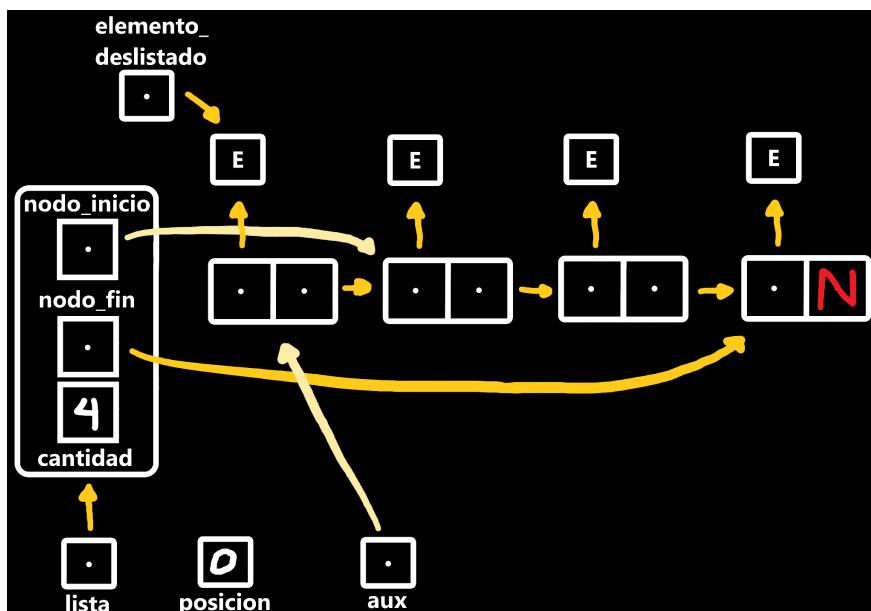
hecho que recorrer la lista con el iterador tenga una complejidad de $O(n!)$ y pierde la gracia de utilizar un iterador externo.

```
void *lista_quitar_de_posicion(lista_t *lista, size_t posicion);
```

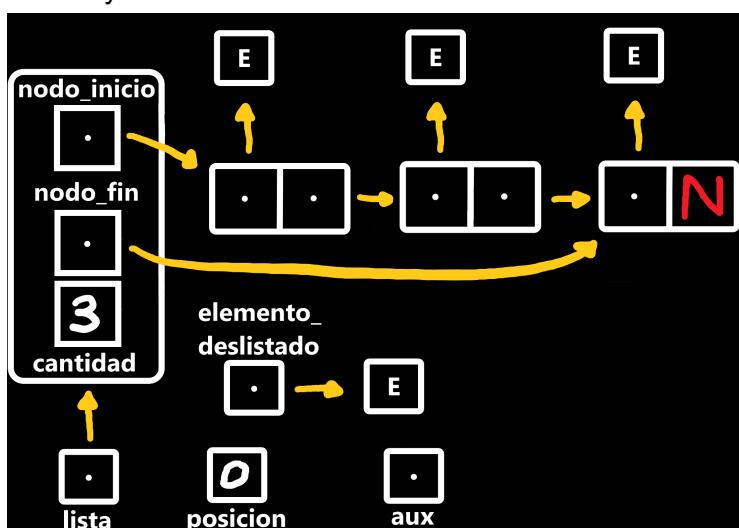
Esta función recibe un puntero a la lista y la posición del nodo que se quiere borrar, devuelve un puntero al elemento del cual el nodo fue eliminado. Tiene una complejidad de $O(n)$.

Si la lista está vacía la función devuelve NULL. Si la posición que se quiere borrar es la última o más grande la función llama a lista_quitar y devuelve la lista.

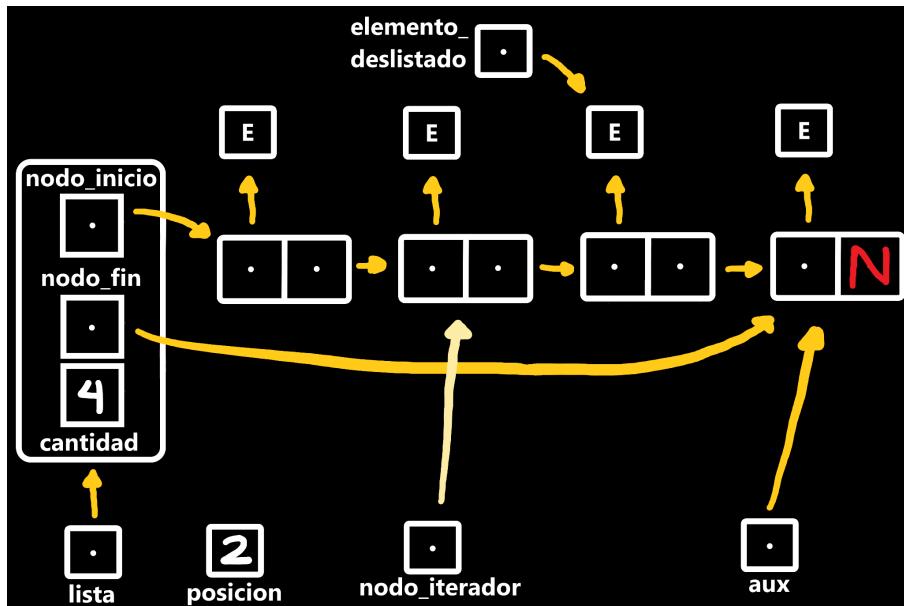
Si posición = 0, se crea un puntero al elemento del cual el nodo se va a borrar, con un puntero auxiliar se apunta al primer nodo y nodo_inicio ahora apuntará al segundo nodo de la lista.



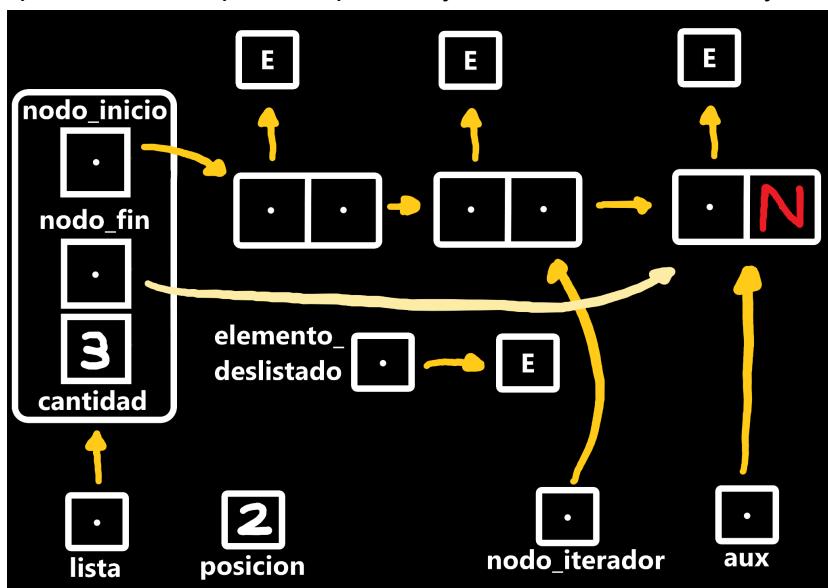
Luego se libera la memoria reservada del nodo y la cantidad de elementos en la lista se disminuye en 1.



En caso de querer borrar un elemento en el medio de la lista, se crea un puntero a nodo llamado `nodo_iterador` que irá recorriendo los nodos uno por uno hasta llegar al nodo anterior al que se quiere borrar, para este ejemplo voy a borrar el nodo en la posición 2, se crea un puntero al elemento que será devuelto por la función y se crea un puntero a nodo llamado `aux` que servirá para apuntar al nodo siguiente al que se borrara.



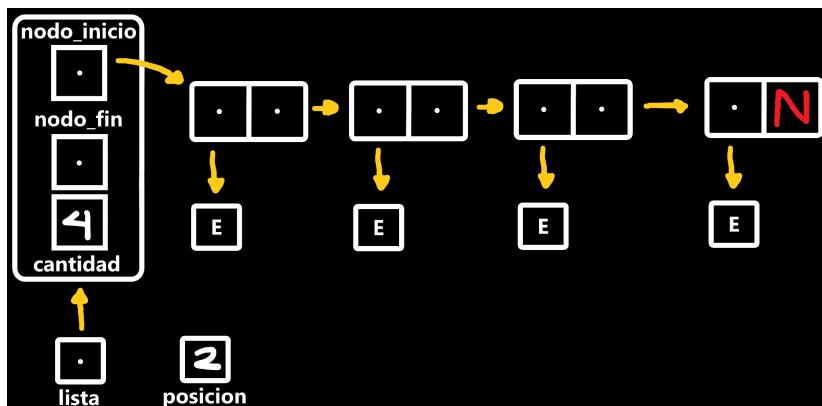
Luego se libera la memoria reservada por el nodo, el nodo apuntado por `nodo_iterador` se apunta al nodo apuntado por `aux` y `lista->cantidad` disminuye en 1.



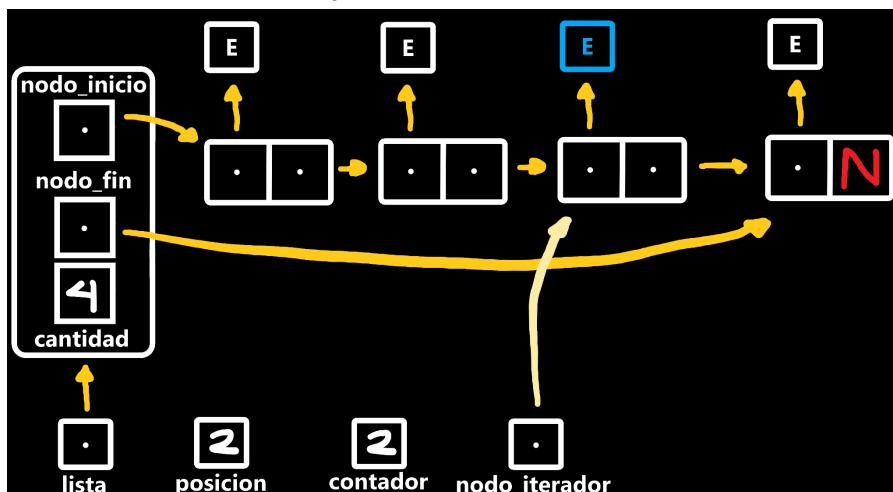
Esta función tiene una complejidad de $O(n)$ porque debe iterar hasta encontrar el nodo anterior al nodo que se quiere borrar ($n = \text{posición} - 1$).

```
void *lista_elemento_en_posicion(lista_t *lista, size_t posicion);
```

Esta función recibe un puntero a la lista y una posición, devuelve el elemento que ocupa la posición recibida.

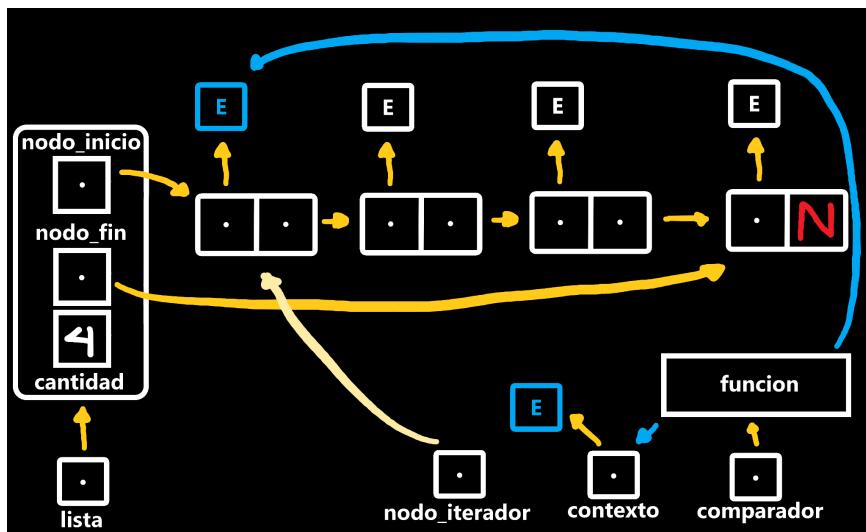


En este caso lista_elemento_en_posicion recibió posición = 2. Se crea un puntero a nodo_t llamado nodo_iterador y un size_t contador = 0. Se itera sobre la lista hasta que el contador sea igual a la posición recibida por parámetro y se devuelve el elemento apuntado por nodo_iterador. La complejidad de la función es de $O(n)$, siendo $n = \text{posición}$.

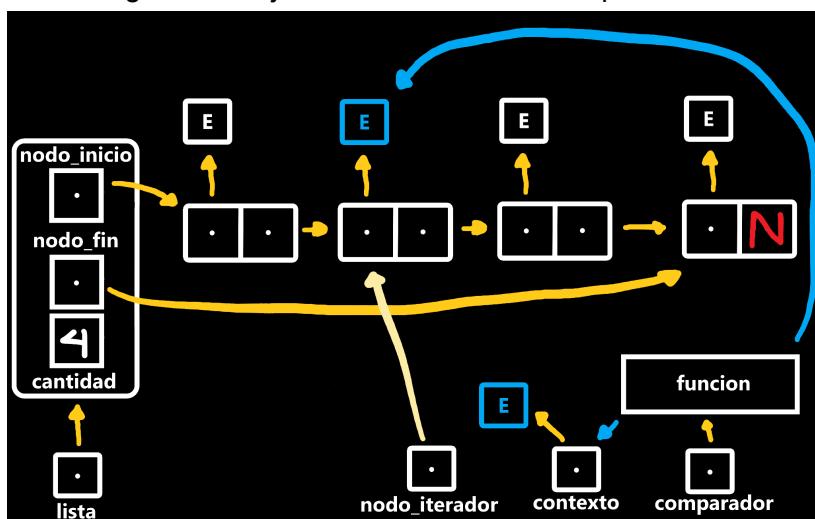


```
void *lista_buscar_elemento(lista_t *lista, int (*comparador)(void *,
void *), void *contexto);
```

Se recibe un puntero a la lista, un puntero a función y un puntero a un “contexto”. El contexto sirve para comparar entre los elementos en la lista y esta variable. Se crea un puntero a nodo_t llamado nodo_iterador, la función de este puntero es iterar la lista hasta que se acabe la lista o hasta que la función comparador devuelva un valor = 0. Finalmente devuelve el elemento del cual comparador devuelve 0 o en caso de recorrer toda la lista y no encontrar este elemento devolver NULL.



En la imagen de abajo el iterador avanzó una posición.



Esta función tiene una complejidad de $O(n)$ siendo $n =$ la cantidad de iteraciones hasta encontrar el elemento que devuelve 0 en la función comparador.

```
void *lista_primeros(lista_t *lista);
```

Esta función recibe la lista y devuelve el primer elemento de la lista, devuelve `lista->nodo_inicio->elemento`. En caso de error devuelve NULL. Tiene complejidad de $O(1)$.

```
void *lista_ultimo(lista_t *lista);
```

Esta función recibe la lista y devuelve el último elemento de la lista, devuelve `lista->nodo_fin->elemento`. En caso de error devuelve NULL. Tiene complejidad de $O(1)$.

```
bool lista_vacia(lista_t *lista);
```

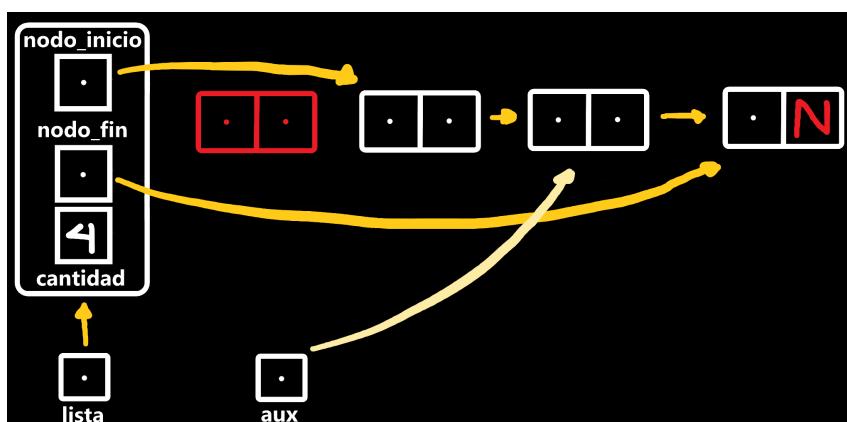
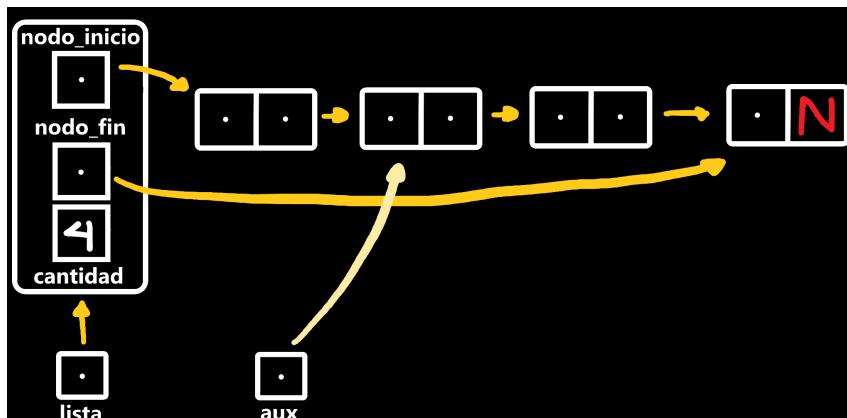
Devuelve true si `lista->cantidad = 0` o si el puntero a lista es NULL, en caso contrario devuelve false. Tiene complejidad de $O(1)$.

```
size_t lista_tamano(lista_t *lista);
```

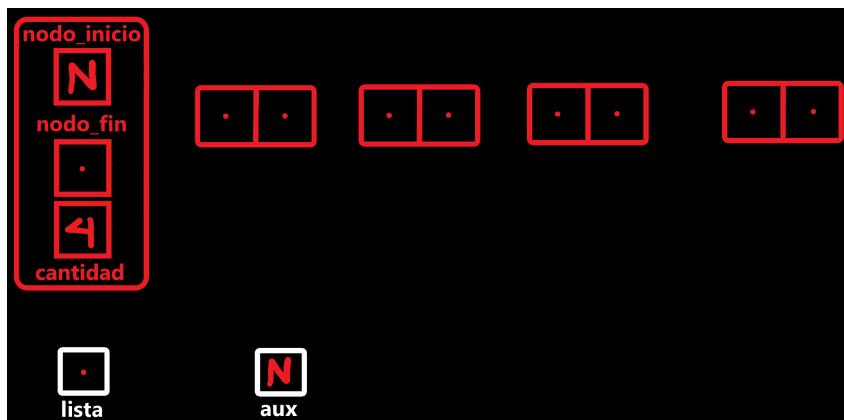
Devuelve el valor actual de lista->cantidad. En caso de que el puntero sea NULL devuelve 0. Tiene complejidad de o(1).

```
void lista_destruir(lista_t *lista);
```

Esta función recibe un puntero a la lista y libera la memoria reservada por los mallocs y callocs utilizados para llenar la lista. Itera sobre la lista junto a un puntero nodo_t auxiliar para no perder la referencia al resto de la lista mientras de borra el nodo apuntado por lista->nodo_inicio. Va liberando la memoria de los nodos 1 por 1 hasta borrar todos los nodos y por último hace free a la memoria apuntada por el puntero de lista. lista_destruir no libera los elementos, solo libera los nodos. Para eso se requiere utilizar la función lista_destruir_todo.



La iteración de la lista ocurre hasta que aux apunte a NULL, esto ocurrirá cuando al llegar al último elemento quiera ir al siguiente que en ese caso es NULL, borrara el nodo y luego libera la lista. A medida que borra nodos, la función no disminuye la lista->cantidad.

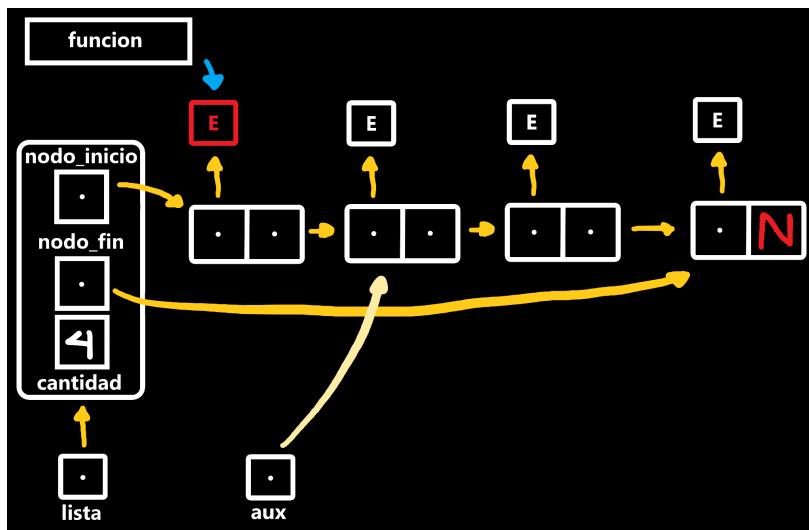


La complejidad de la función es $O(n)$, $n = \text{lista} \rightarrow \text{cantidad}$.

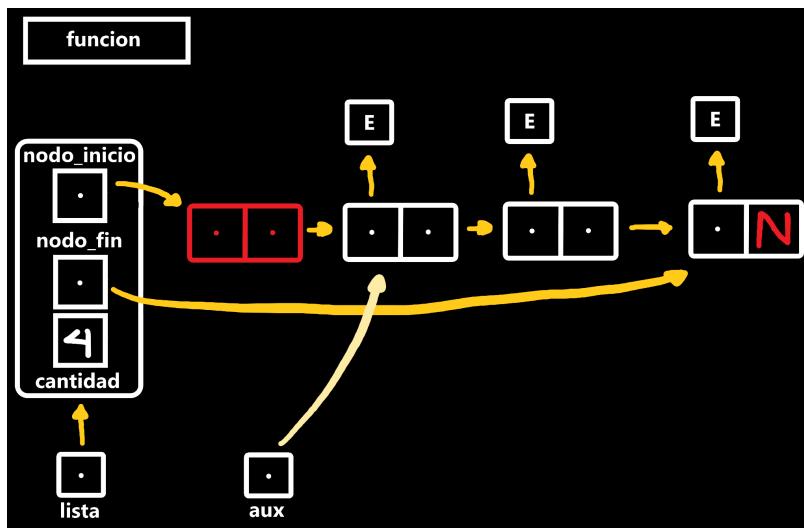
```
void lista_destruir_todo(lista_t *lista, void (*funcion)(void *));
```

`lista_destruir_todo` funciona de una manera parecida a `lista_destruir` pero antes de liberar el nodo aplica una función a su elemento. En el archivo de ejemplo utiliza `free()`. Sirve para liberar el espacio de memoria utilizado por el elemento en el caso de haber reservado memoria para él con `malloc()`.

Se fija que el puntero a función no sea `NULL`, en caso de serlo llama a la función `lista_destruir` y tira `return`. En caso de que los punteros recibidos por parámetro no sean `NULL`, itera sobre la lista con la ayuda de un puntero a `nodo_t` auxiliar de la misma manera que lo hace `lista_destruir` y aplica la función dada por parámetro al elemento apuntado por `lista->inicio`, luego hace `free` del nodo y mueve `nodo_inicio` al siguiente nodo. Al finalizar libera la memoria reservada al crear la lista.



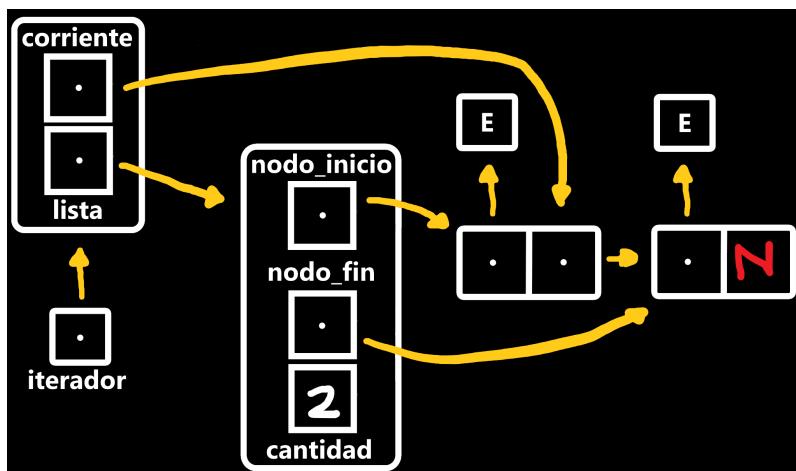
En esta foto la función ya borro el elemento del primer nodo y ahora se hará `free(lista->nodo_inicio)`.



Luego de borrar el nodo, `nodo_inicio` apuntará al siguiente, `aux` al siguiente y la función recibe el elemento de ese nodo. Se repite el proceso hasta terminar con la lista. La complejidad de la función es $O(n)$ siendo $n = \text{lista} \rightarrow \text{cantidad}$.

```
lista_iterador_t *lista_iterador_crear(lista_t *lista);
```

Esta función recibe el puntero a una lista y se la guarda en un struct donde aparte de tener la lista habrá un iterador llamado corriente. El puntero corriente será apuntado al primer nodo de la lista.



La función tiene una complejidad de $O(1)$.

```
bool lista_iterador_tiene_siguiente(lista_iterador_t *iterador);
```

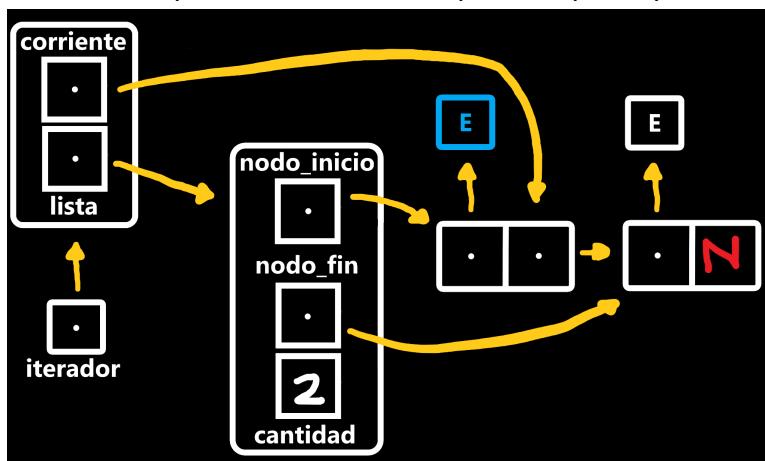
La función devuelve true a menos que el puntero corriente esté apuntando a NULL. Corriente sólo apunta a NULL si el iterador leyó toda la lista o si no está inicializado. Tiene una complejidad de $O(1)$.

```
bool lista_iterador_avanzar(lista_iterador_t *iterador);
```

La función avanza el iterador al siguiente nodo y devuelve true si avanza el iterador. Si avanza el iterador pero el puntero corriente ahora apunta a NULL la función devuelve false. Tiene una complejidad de o(1).

```
void *lista_iterador_elemento_actual(lista_iterador_t *iterador);
```

Devuelve un puntero al elemento apuntado por el puntero corriente.



La función tiene complejidad o(1).

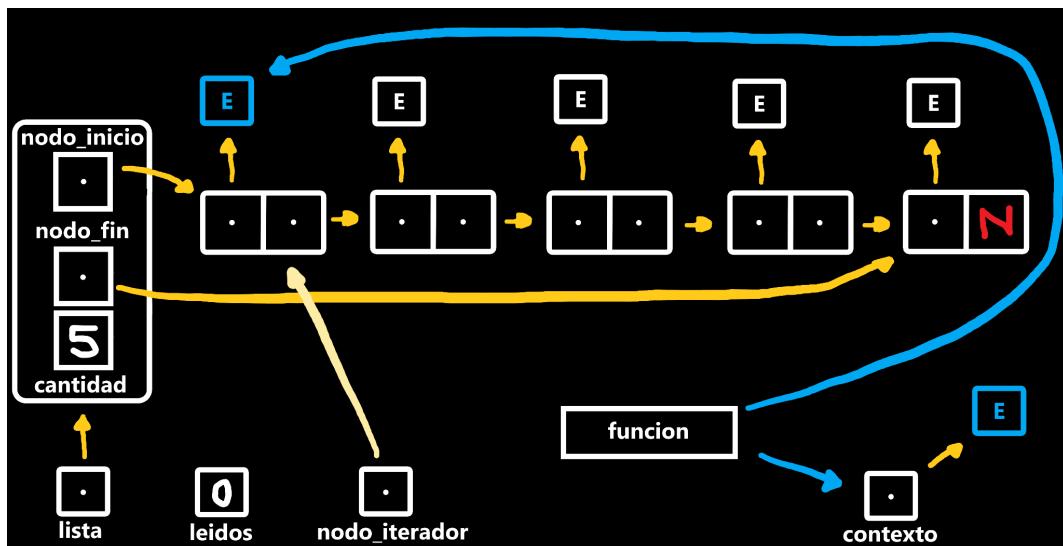
```
void lista_iterador_destruir(lista_iterador_t *iterador);
```

Libera la memoria utilizada por el iterador utilizando un free. La complejidad es o(1).

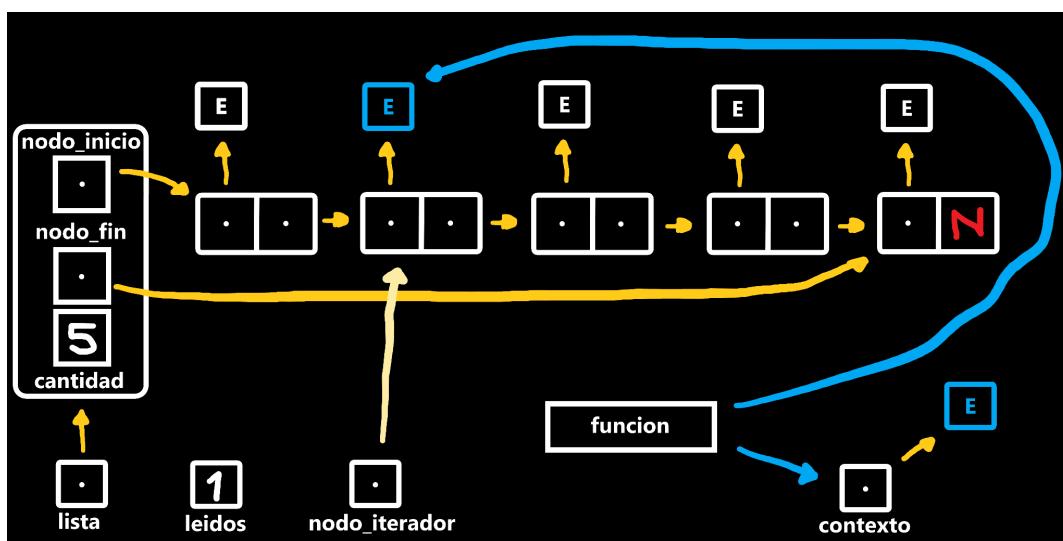
```
size_t lista_con_cada_elemento(lista_t *lista, bool (*funcion)(void *, void *), void *contexto);
```

Esta función navega la lista con un iterador interno. Es muy parecido a la iteración que hacían lista_agregar_en_posicion o lista_quitar_de_posicion. Recibe un puntero a la lista, un puntero a función y un puntero a un “contexto”. lista_con_cada_elemento va a navegar la lista de principio a fin y va a mandar a la función pasada por parámetro el elemento del nodo actual y el puntero a contexto y repetirá con todos los nodos hasta que función devuelve false. Cuando termine la iteración, lista_con_cada_elemento devolverá la cantidad de nodos iterados.

En esta imagen la función compara nodo_iterador->elemento y contexto.



Una vez que termina de correr la comparación entre los 2 elementos el iterador se mueve al siguiente nodo y se suma 1 a leidos.



De ser esta la vez que función devuelve false, a leidos se le suma 1 y la iteración termina y lista_con_cada_elemento devuelve el total de leídos. La complejidad del algoritmo es de $O(n)$, $n = \text{leidos}$.

Preguntas teóricas

TDA Pila:

La pila es un tipo de dato abstracto que permite guardar datos de una manera particular. Tiene una estructura LIFO que significa “last in, first out”, esto es lo que lo separa de los demás TDAs.

En una pila los datos a medida que van entrando se apilan, en el sentido de que uno no puede acceder a datos que están en el medio porque hay datos que están encima de él. La

pila no permite que el usuario acceda a esos datos sin antes quitar los datos que fueron introducidos posteriormente a él.

TDA Cola:

La cola es parecida a la pila en el sentido de que uno entra datos en la estructura desde el final. La diferencia entre los dos es que los datos de la cola no pueden ser accedidos si existe un dato más “viejo” a él, así como la pila solo le dejaba al usuario interactuar con el dato más reciente, la cola solo le dejará interactuar con el dato más antiguo. Tiene una estructura FIFO que significa “first in, first out”, esta es la cualidad única de la cola.

En una cola los datos nuevos entran por el final como en la pila, pero en vez de imaginarlos todos apilados uno los imagina uno al lado del otro. Esto ayuda a ver el primer dato de la cola (siendo el más antiguo) estando todavía accesible a ser eliminado y donde no está siendo restringido por los datos más nuevos como en la pila.

TDA Lista:

La lista es el TDA más libre y dinámico de los tres. No tiene reglas o restricciones como la pila y la cola, en ese sentido el usuario puede hacer lo que quiera con la lista. Es posible acceder a información del medio, agregar datos donde uno quiera y quitar de cualquier posición. Básicamente la pila y la cola son listas con restricciones de uso como solo poder agregar datos desde el final y quitarlos de cierta manera dependiendo de cual estemos hablando.

TDA Pila con nodos simplemente enlazados:

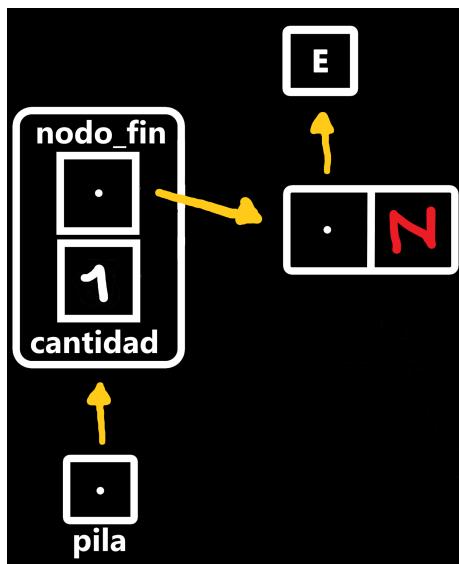
A la hora de crear una pila con nodos simplemente enlazados necesitamos crear una estructura para ella. También necesitamos reservar memoria ya que esta estructura necesita vivir en el heap de la RAM.

Para esta implementación necesitamos definir que es un nodo, yo usare el que utilice más arriba cuando explique la otra implementación de pila. Cada nodo va a guardar un puntero al elemento y un puntero a su siguiente, en este caso cada nodo va a apuntar su antecesor.



Creamos la pila y asignamos el puntero a NULL y la cantidad en 0.

Para agregar elementos vamos a primero necesitamos reservar memoria en el heap para un nodo y apuntar su puntero a elemento al elemento que quiera guardar el usuario y apuntar el siguiente del nodo a donde esté apuntando **nodo_fin** (NULL). Luego apuntar el puntero **nodo_fin** al nodo y sumar 1 a la cantidad de elementos en la pila.



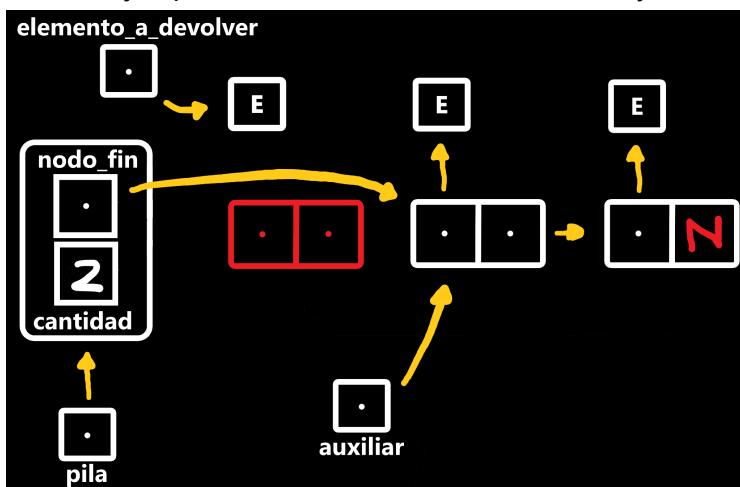
Lo bueno de este algoritmo es que sirve para agregar nodos en todos los contextos, ya sea agregar un nodo por primera vez o agregarlo una segunda o tercera.

Para quitar un elemento de la pila es bastante simple también ya que ordenamos la pila de manera de que los nodos apuntan a sus antecesores, de ser al revés quitar nodos de la pila quedaría con una complejidad de $O(n)$.

Quitar un elemento implica primero guardar la dirección de memoria del elemento del cual nodo será eliminado de la pila, apuntar con la ayuda de un puntero auxiliar al resto de nodos para cuando liberemos la memoria del último nodo no perdamos la pila entera. Liberar la

memoria, conectar nodo_fin al nodo antecesor y posteriormente restarle 1 a la cantidad de elementos de la pila.

En este ejemplo tenía una cola con 3 elementos y eliminé 1.



Estas son las acciones básicas que necesita hacer una pila, poder agregar y borrar elementos. También podemos implementar funciones como para saber cual es el último elemento de la pila o una función que devuelva la cantidad de elementos que guarda.

Por ser C necesitamos liberar la memoria que utilizamos en nuestro programa. Para esto necesitaremos liberar nodo por nodo y finalmente liberar el bloque de memoria al que apunta nuestro puntero pila. Todas estas funciones tienen una complejidad de $O(1)$ excepto la de liberar la memoria de la pila siendo $O(n)$.

La ventaja de crear una pila con este método es que a diferencia de utilizar un vector, la RAM no necesita reservar bloques de memoria todos juntos. Cuando tenemos una estructura almacenando pocos elementos no hay problema pero uno puede tener un inconveniente cuando quiere almacenar muchos elementos y usando nodos uno puede arreglar ese problema ya que la memoria puede reservar bytes por todos lados sin necesidad de que estén todos los bytes de corridos.

TDA Cola de vector estático (cola circular):

```
#define MAX_ELEMENTOS 8

int main()
{
    void *cola[MAX_ELEMENTOS];

    int inicio = 0;
    int tope = 0;
    int final = MAX_ELEMENTOS - 1;

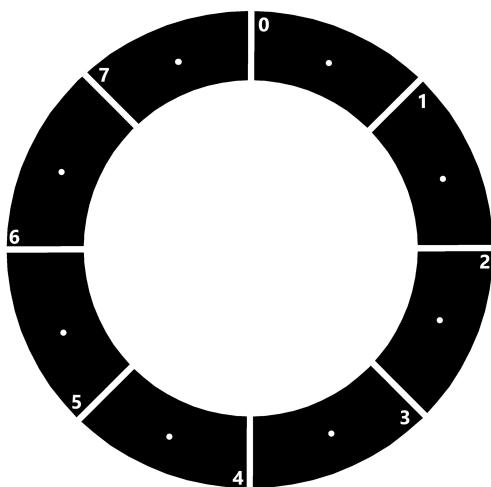
    return 0;
}
```

{}

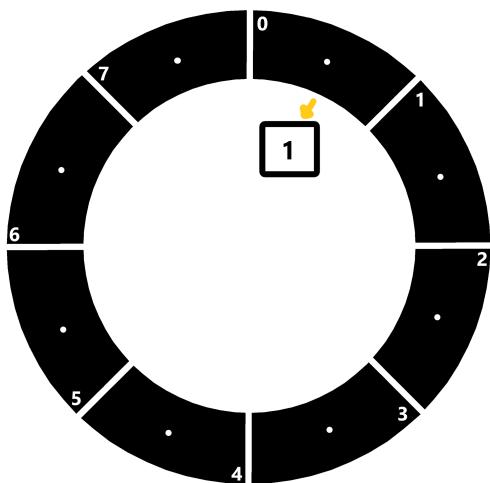
Esta es la definición de la cola que se me ocurrió hacer. Es un vector estático que guarda punteros a void. En este caso la cola solo podrá almacenar hasta 8 elementos. Al ser un vector estático no se utilizará memoria dinámica para el uso de la cola. Tengo una duda y es si necesito apuntar los punteros a NULL o si al definir la cola los punteros ya vienen con un valor de 0.

Definí la variable inicio que marca el principio de la cola, esta variable es necesaria porque estamos usando una cola circular, de ser “lineal” uno como programador no necesita guardar este dato ya que el principio siempre será el comienzo del vector. La variable tope guarda la posición del vector donde está guardado el último dato y la variable final guarda la última posición del vector.

En un diagrama la cola se vería algo así:



Para agregar un elemento a la cola el procedimiento sería algo así como crear una variable como por ejemplo “int elemento = 1”, asignar su dirección de memoria a `cola[tope]` y al terminar incrementar el tope en 1 (excepto la primera vez, porque tope ya está posicionado en 0 que es donde está el último dato). El problema viene cuando uno quiere agregar elementos cuando 1: la cola está llena y 2: inicio > 0 y tope es = 7.

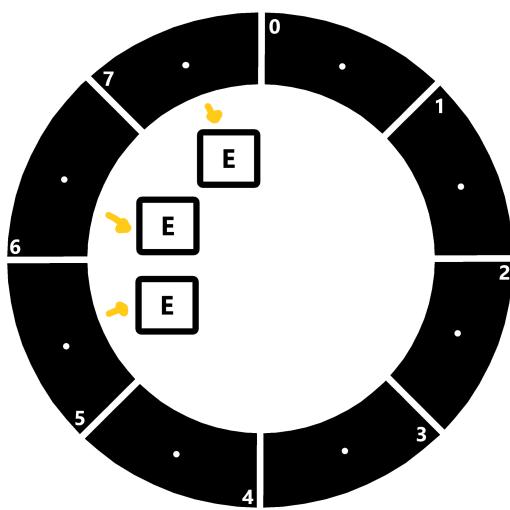


Si la cola está llena se pueden hacer 2 cosas, no agregar el elemento y devolver el error o eliminar el primer elemento de la cola y reemplazarlo con el nuevo. Me gusta mucho más la primera alternativa porque no perdemos información pero quería nombrar la otra porque dependiendo de cual sea el “contrato” de la cola, podría ser una opción viable o incluso la mejor. Antes de querer agregar un elemento deberíamos preguntar siempre si la cola está llena así que voy a agregar esta variable:

```
bool cola_llena = false;
```

cola_llena será false siempre y cuando (tope != inicio - 1) o en caso de que inicio sea 0 usare otro valor que será el de MAX_ELEMENTO.

Para solucionar el otro problema imaginemos que la cola tiene inicio = 5 y tope = 7. Eso se vería algo así:



En este caso queremos agregar un elemento a la cola, como tenemos que agregar al final y el final es en la posición 0 del vector tenemos que averiguar cómo “decirle” al programa que queremos guardar el elemento ahí.

Sabemos que:

```
inicio = 5
tope = 7
final = 7
cola_llena = false
```

Primero preguntamos si la cola está llena, la respuesta es no así que podemos agregar un elemento a la cola. Segundo preguntamos si tope es igual a final, esto es un caso particular que requiere una solución aparte a la solución que di para agregar un elemento al vector antes. En esta situación deberíamos decirle específicamente que apunte el puntero de la posición 0 al elemento que se quiere agregar (`cola[0] = &elemento`) y poner la variable tope en 0.

Para quitar un elemento necesitamos preguntar si la cola está vacía porque de estar vacía necesitamos devolver un error especificando que no quedan elementos en la cola. Voy a definir otra variable:

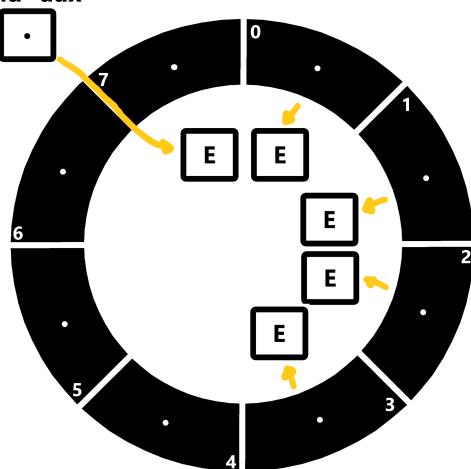
```
bool cola_vacia = true;
```

Esta variable será true siempre en cuanto inicio y tope tengan el mismo valor.

Luego debemos crear un puntero auxiliar a void que guarde el elemento que vamos a borrar, esto sirve para devolverle el elemento al usuario. Después tenemos que apuntar el puntero `cola[inicio]` a NULL y sumarle 1 a inicio. Por último chequear que inicio no sea 8, de ser el caso debemos cambiar su valor a 0.

En esta foto dibuje cómo quedaría la cola luego de quitar el elemento de la posición 7.

`void* aux`



En caso de querer saber cual es el elemento que sea encuentra en el inicio, podemos por ejemplo devolver `cola[inicio]` o un puntero a el elemento.

Para saber cual es el tamaño de la cola podemos agregar otra variable:

```
int tamanio = 0;
```

y sumarle o restarle 1 dependiendo de la acción que tomemos en la cola. Ahora que agregamos un contador, podemos saber si la cola está vacía sí tamaño = 0 y podemos saber si está llena si tamaño = 8.

Al estar usando solo la memoria del stack, en este caso no necesitamos liberar memoria. Todas las operaciones realizadas tienen una complejidad de $O(1)$.

También quería agregar que lo mejor sería tener todos estos procesos encapsulados en su propia función y que probablemente sea mejor tener todas estas variables en una variable estructurada solo para tener todo más organizado.

La ventaja de hacer la cola con un vector estático es que consume menos memoria y es más simple que hacerlo con un vector dinámico. Aparte al hacerla circular los procesos tienen todos los procesos complejidad $O(1)$, si fuera una cola “lineal” borrar un elemento cambiaría la complejidad a $O(n)$ porque habría que correr todos los elementos del vector una posición a la izquierda.

La desventaja es que estamos limitados a la hora de insertar elementos, podemos encolar x cantidad de elementos y x estaría definida al momento de crear la cola. Aparte si tenemos un vector de 1000 posiciones y solo usamos 100 estaríamos reservando memoria que no se utilizaría (no es eficiente).

TDA Lista con nodos simplemente enlazados:

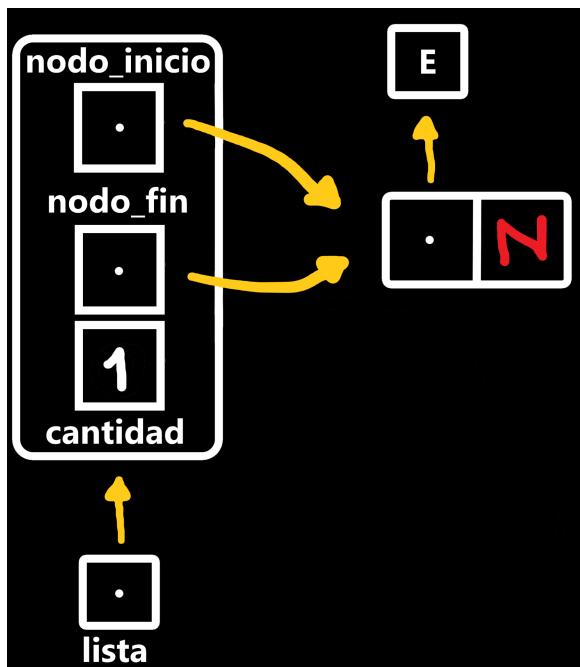
Para usar una lista con nodos simplemente enlazados necesitamos definir una estructura para ella, en este caso voy a usar la que use para este TP. Como vamos a estar trabajando con nodos vamos a hacer uso del heap. Voy a utilizar la misma estructura de nodos que utilice para mi otra implementación de lista.

Para crear la lista solo necesitamos reservar memoria del tamaño de nuestra estructura de lista.



Esta implementación que voy a explicar es distinta a la del TP. Es muy parecida a como agregamos y quitamos elementos de la pila. Deja la complejidad de agregar y quitar elementos al final de la lista en $O(1)$. Después meterlos y quitarlos del medio es un poco más engorroso.

Para agregar un elemento vamos a primero reservar memoria para un nodo y vamos a apuntar el puntero de `nodo->elemento` al elemento que queremos guardar, vamos a apuntar el siguiente del nodo a donde esté apuntando `nodo_fin`. Posteriormente vamos a apuntar `nodo_inicio` al nodo nuevo solo si la cantidad de elementos en la lista es 0 y finalmente incrementar la cantidad de la lista en 1.

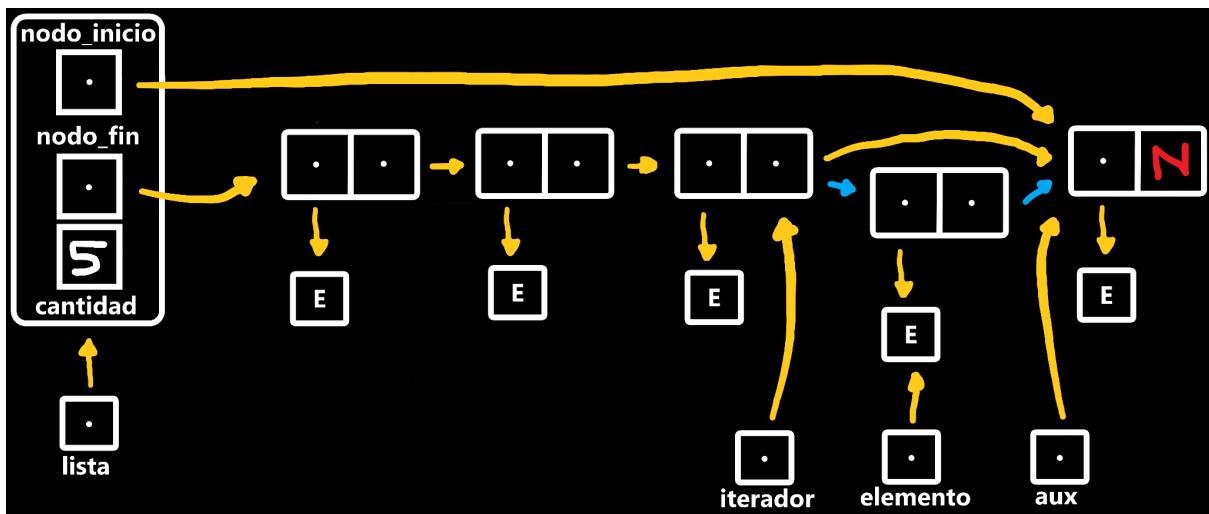


Este método hace que los nodos apunten a sus antecesores entonces agregar y quitar el último elemento es más fácil que si fuera al revés.

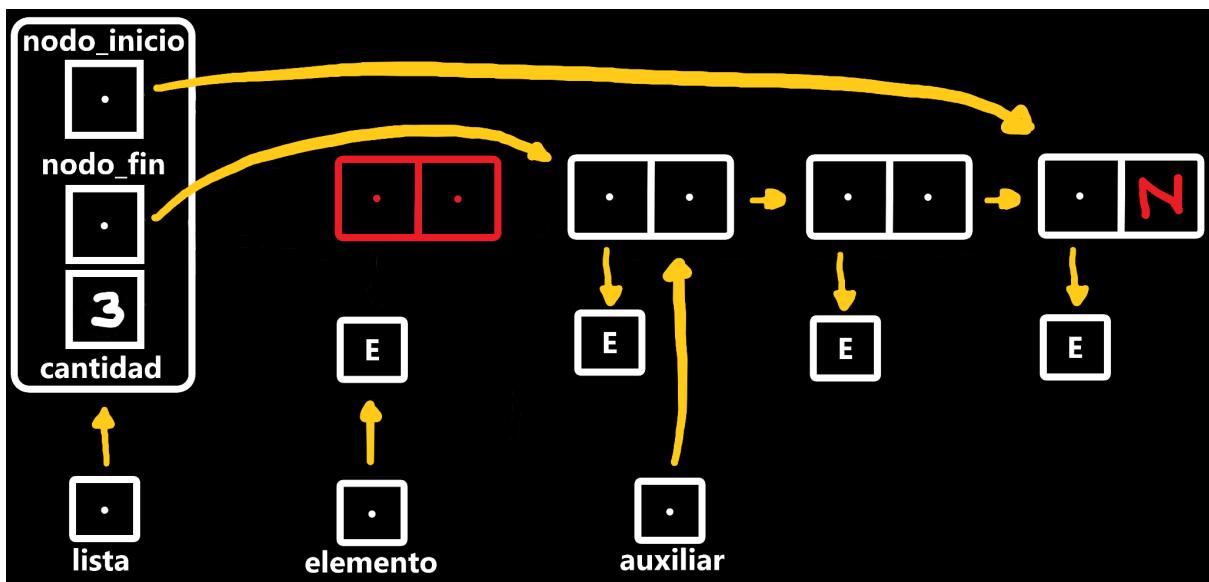
Si queremos agregar un elemento en el medio de la lista tenemos que reservar memoria para un nodo nuevo y apuntar su puntero a elemento al elemento que queremos almacenar. Tenemos que calcular una nueva posición en la lista ya que al estar “dada vuelta” tenemos que decirle que si por ejemplo:

```
lista->cantidad = 4
posición = 1
```

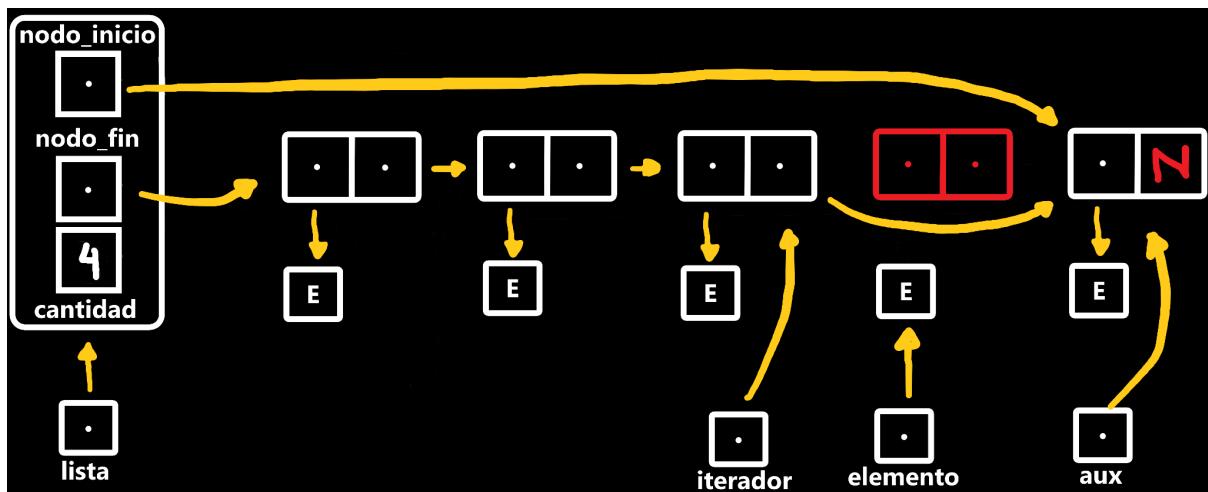
Vamos a querer meter el nuevo nodo en la posición 3. Para calcular la posición real necesitamos hacer $\text{lista}->\text{cantidad} - \text{posición}$. Luego debemos iterar con un iterador interno hasta encontrar el nodo anterior a 3, con la ayuda de un puntero auxiliar guardarnos el resto de la lista apuntando al nodo de la posición 4. Luego apuntar el nodo 2 al nodo 3 y el nodo 3 a donde apunta el puntero auxiliar (nodo 4), por último incrementar el contador de elementos en la lista en 1. Este algoritmo siempre va a tener una complejidad de $O(n)$.



Para quitar el último nodo de la lista es igual que cómo quitar un nodo de la implementación de lista que hice arriba. Podemos apuntar con un puntero al elemento del cual nodo vamos a borrar y con la ayuda de un nodo auxiliar apuntar al nodo siguiente para no perder la lista, hacer free del nodo y disminuir en 1 lista->cantidad.



Para quitar un elemento en el medio de la lista tenemos que utilizar casi el mismo cálculo que explique antes. En una lista con 5 elementos en la que queremos eliminar el nodo de la posición 1 vamos a cambiar la posición a 3 que es igual a `lista->cantidad - posición - 1`. Vamos a iterar con la ayuda de un iterador interno hasta encontrar el nodo anterior a 3 y vamos a guardarnos lo siguiente al nodo 3 con un puntero auxiliar. Luego guardamos en otro puntero el elemento que queremos devolver, liberamos la memoria del nodo y conectamos el nodo anterior con el posterior al borrado. Por último disminuimos el contador de elementos de la lista en 1.



Estas son las funciones más importantes de una lista, también podemos buscar por la lista hasta encontrar un elemento y devolver su posición o devolver el elemento que se encuentra último o primero en la lista. La iteración es la misma que use para navegar la lista al querer agregar y quitar un elemento. Hay que tener en cuenta que con la implementación que explique ahora hay que siempre calcular la posición nueva.

Las funciones de agregar y quitar elementos al final de la lista tienen complejidad de $O(1)$, las de editar la lista en el medio tienen complejidad $O(n)$.

Como último requisito hay que liberar la memoria utilizada por la lista. Primero se liberan los nodos en orden y luego el bloque de memoria apuntado por el puntero lista. Una función para esto funcionaría parecido a quitar el último nodo de la lista exceptuando la parte de devolver los elementos de ella. Tendría una complejidad de $O(n)$.