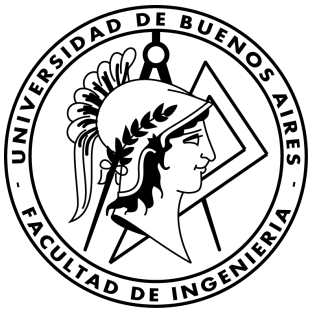

Trabajo Práctico 2 — TDA Lista

[7541/9515] Algoritmos y Programación II
Primer cuatrimestre de 2022

Alumno:	Gómez, Federico
Número de padrón:	109159
Email:	fedgomez@fi.uba.ar

Índice

1. Introducción

Para el TP de TDA Lista se pidió implementar los TDAs lista, cola y una pila mediante nodos simplemente enlazados. Para implementar dichos TDAs, se contó con las firmas de las primitivas y del TDA lista, así como un archivo de ejemplo a modo de, valga la redundancia, ejemplo de uso de los TDAs anteriormente mencionados. Junto a estos archivos, además, se brindó un makefile, para poder compilar y correr el archivo sin necesidad de escribir los comandos para gcc y Valgrind, con sus respectivos flags, una y otra vez.
Adicionalmente, se solicito hacer uso de la metodología orientada a pruebas, desarrollando tests de la implementación mediante el framework pa2mm o pa2m, a fin de poder tener una forma de comprobar el correcto funcionamiento de las primitivas de los TDAs cada vez que se compila el programa, así como también lograr una detección rápida de bugs y leaks en la memoria.

2. Teoría

A continuación se presenta información sobre qué es, el funcionamiento y distintos tipos de implementación de los TDAs Lista, Pila y Cola:

1. ¿Qué es una Pila?
   Una pila es una estructura que agrupa elementos y cuya principal característica es que solo se puede acceder al primer elemento. Esto se conoce como Last In-First Out (LIFO), y esto significa que se pueden apilar tantos elementos como se desee, pero siempre el ultimo elemento que fue apilado será al que tengamos accesso y podamos ver, y cuando queramos acceder a otro elemento, primero tendremos que desapilar los elementos que estan antes hasta llegar a él.
   Este TDA posee seis primitivas, más allá de cualquier otra que se le pueda agregar, las cuales son: crear, apilar, desapilar, tope, vacia y destruir. De aquí, apilar y desapilar nos permitiran insertar y quitar elementos de nuestra pila, mientras que tope nos servira para consultar el elemento que esta en lo que sería el principio de la pila.

   El TDA Pila puede implementarse de varias formas:
   - Mediante un vector estático: Esta implementación es bastante sencilla, se tiene un vector estático y dos variables, una para indicar la posicion del tope y otra para saber cuantos elementos pueden guardarse, sin embargo se paga un precio por la sencillez de manjeo de esta implementación, ya que dispondremos de una cantidad limitada de elementos. Cada vez que se apilan/desapilan elementos, se lo hace como un vector normal, con la diferencia de que para apilar pondremos el elemento en la posicion siguiente a la del tope, y cuando desapilemos un elemento cambiaremos el valor de la posicion del tope a otro indicando que fue desapilado, y ademas moveremos la posicion del tope a la posición anterior.
   Cabe destacar que para esta implementación todas las operaciones resultan O(1), ya que nunca hace falta mover todos los elementos a un nuevo espacio de memoria ni iterarlos.

   - Mediante un vector dinámico: Esta es una implementación muy parecida a la anterior, con la gran diferencia de que la cantidad de elementos que se pueden guardar en la pila ya no estará limitada por un vector estático, sino que se podrá solicitar más memoria cuando el vector dinámico se llene de elementos, a fin de expandirlo y poder continuar agregando elementos a la pila. De aquí, el gran problema sería que cada vez que se necesite más espacio, habrá que reallocar todos los elementos que estan en la pila, ya que se necesita que los bloques de memoria esten contiguos. En cuanto a las operaciones, se trabajan de la misma forma que la anterior implementación mencionada.
   En este caso, destruir, apilar y desapilar podrían llegar a tener una complejidad algorítmica O(n), ya que se tendría que, en el peor de los casos, reallocar todos los elementos a un nuevo espacio de memoria, pero el resto de las operaciones resultan O(1).

- Mediante nodos simplemente enlazados: En esta implementación se tienen Nodos, los cuales son un 'mini' TDA. Cada Nodo tiene referencias a un elemento y un Nodo siguiente, de modo que cada vez que se quiera apilar un elemento se debe crear un nodo, reservando memoria solo para el nodo, con la referencia al elemento y el nodo que lo procede. De esta forma, el único limitante para la cantidad de elementos es la cantidad de memoria disponible, y resulta algo más sencillo de manejar en estos terminos, ya que para una Pila implementada como vector dinámico se tenía que realocar un bloque entero cada vez que se quedaba sin espacio, mientras que aquí tendremos las referencias a los nodos y podrán estar en cualquier parte de la memoria sin la necesidar de estar contiguos.
Algo a destacar, además, es que las operaciones de este TDA serán O(1), excepto destruir, ya que en el peor de los casos deberán destruirse todos los nodos, mientras que para destruir las anteriores implementaciones, con vector dinámico o estático, esta primitiva era O(1).

2. ¿Qué es una Cola?
Una cola es una estructura que agrupa elementos (al igual que las anteriores), pero se trata de una estructura algo similar a una Pila. La salvedad aquí esta en que, mientras que la Pila es una estructura que funciona como LIFO, la Cola funciona como First In-First Out (FIFO), es decir que el primer elemento en ingresar a la cola será el primero en salir, algo sospechosamente similar a la Cola de un supermercado.
Este TDA posee seis primitivas, las cuales son crear, destruir, vacia, encolar, desencolar y ver frente. Encolar nos permite insertar elementos al final de la cola, desencolar nos permite quitar el primer elemento de la cola y por ultimo ver frente nos permite acceder al primer elemento de la cola, es decir, el elemento que fue insertado primero.

   El TDA Cola puede implementarse de varias formas:
   - Mediante un vector estático: Aquí la realidad es que resulta muy similar a una Pila, pero la trabajamos al revés de la pila. Tendremos una variable que nos indica la posición del elemento que está al frente de la cola, una variable para el final de la cola y una variable para la cantidad de elementos que podemos almacenar. Cada vez que encolemos, estaremos encolando en las posiciones siguientes al fin, mientras que, cuando desencolemos, estaremos 'quitando' el elemento que esta en la posición del frente, así como reacomodando dicha posición a la siguiente.
   En esta implementación, todas sus operaciones resultan O(1), pero si se desplazan los elementos cuando se desencola, tendríamos una complejidad O(n) para desencolar. Otra implementación con un vector estático sería una cola circular, de modo que el tope estaría 'persiguiendo' al principio, esto es, tendríamos en lugar de una variable de cantidad, un tope, que estaría siempre en la posición anterior a el principio.

   - Mediante un vector dinámico: De nuevo, en esta implementación se realizan las operaciones de la misma forma que se detallo anteriormente, con la diferencia de que ahora podremos realocar el vector para almacenar más elementos cuando nos quedemos sin espacio en él. O sea que viene a ser igual a la implementación anterior, pero además necesitaremos reservar memoria cada vez que nos quedemos sin espacio en el vector o si simplemente nos sobran demasiados espacios.
   En esta implementación, todas sus operaciones resultan O(1), excepto encolar y desencolar, ya que se tendría un peor caso en el que habrá que reallocar el vector, ya sea para achicarlo o agrandarlo, siendo estas primitivas O(n).

   - Mediante nodos simplemente enlazados: Esta implementación trabaja con la misma idea de Nodos enlazados que la Pila, y de hecho funciona casi, nótese el casi, de la misma forma que una Pila. ¿Por qué casi como una Pila? Es casi como una Pila, porque crearemos Nodos, reservando memoria para ellos, y cada uno tendra las referencias al nodo siguiente y su elemento, pero como estamos trabajando sobre las primitivas de Cola, ahora tendremos que el frente estará apuntando al primer nodo, el fin al último nodo, y cada vez que queramos

encolar lo haremos al final de la Cola, mientras que cuando desencolamos debemos liberar la memoria del primer nodo y devolver el elemento correspondiente. También pueden usarse nodos doblemente enlazados, con los cuales tendríamos una referencia al anterior, ademas del siguiente y el elemento.
Y por último, además de ser nodos doblemente o simplemente enlazados, podría ser circular, donde el siguiente del último sería el frente, y el anterior al frente sería el último.
En esta implementación, todas sus operaciones resultan O(1), excepto destruir, ya que se tendría un peor caso en el que habrá que destruir cada uno de los nodos de una lista, siendo esta primitiva O(n).

3. ¿Qué es una Lista?
Una Lista, similarmente a una Pila o una Cola, agrupa elementos, pero no tiene un protocolo particular a seguir como LIFO o FIFO, sino que se puede acceder a, insertar y quitar elementos en cualquier posición de la misma. Las primitivas de este TDA son: crear, destruir, vacia, insertar, quitar, insertar en posicion, quitar de posicion y ver elemento. Algo a resaltar de estas primitivas es que insertar inserta un elemento al final de la lista, y quitar quita el ultimo elemento de la lista.
Las formas de implementar una Lista son variadas, y son las siguientes:
- Mediante un vector estático: En esta implementación se trabaja con tres variables, una para saber la posicion del inicio, una para el final y otra para la cantidad de elementos que se pueden tener en el vector. En esta implementación, cuando se inserta, se lo hace en la posicion siguiente al final y cuando se quita, se lo hace de la posición del final, cambiando dicha posición a la anterior.

- Mediante un vector dinámico: Funciona de manera similar a la anterior, con la salvedad de que se podrá redimensionar el vector cuando no nos quede más espacio y queramos insertar, o nos sobre espacio del mismo luego de quitar elementos.

- Mediante nodos simplemente enlazados: En esta implementación se trabaja con la misma idea de las implementaciones anteriores mediante nodos enlazados. Se tienen variables para indicar un Nodo inicio y un Nodo fin, marcando estos el inicio y final de la lista. Se pueden agregar algunas cosas extras, por ejemplo cantidad, para saber cuantos elementos hay en la lista, pero la idea original no incluye esto.
Se reserva memoria cada vez que se ingresa un elemento a la lista, y se libera dicha memoria cuando se quita un elemento. Esa memoria es reservada para crear un nodo nuevo cada vez que se inserta. Al igual que antes, se tienen Nodos con referencias al elemento y al nodo siguiente, pero la gran diferencia entre este TDA y los mencionados anteriormente radica en que se puede acceder a cualquier elemento de la Lista, esto es, no se adhiere a ninguna política del estilo LIFO o FIFO. En cuanto a operaciones y complejidad de cada una, tendremos que tanto crear, vacia e insertar son O(1) en todas las implementaciones. Insertar en posicion, quitar de posicion y ver elemento resultan O(n), ya que en el peor de los casos se debe llegar hasta el último elemento de la lista (o el anterior al último en el caso de buscar, ya que si se busca el último o el primero resultaría O(1)); Quitar al final también resulta en algo similar, ya que, en la forma más básica de la Lista, no tenemos ningún modo de acceder al Nodo anterior al final, por lo que deberemos iterar hasta dicho anterior a fin de actualizar el nodo final de la lista con ese anterior, y establecer como siguiente del nodo fin NULL (o el equivalente en otro lenguaje). Además, cada vez que se quita un nodo de la lista, se retorna el elemento referenciado por él, ya que sino el dato se perdería por completo.
En el caso de destruir, también tendremos una operacion O(n), ya que en el peor y más usual de los casos, deberemos destruir todos los nodos de la Lista, mientras que en los casos anteriores destruir resultaba O(1).

Además, y para concluir, existen algunas variaciones cuando se trabaja con una Lista de

nodos enlazados:
- Lista doblemente enlazada: La idea es igual a nodos simplemente enlazados, con la diferencia de que aquí los nodos tendrán tambien una referencia al nodo anterior, por lo que cambiará la complejidad algorítmica de quitar (al final) a O(1), ya que no nos hará falta llegar hasta el anterior iterando, sino que podremos acceder a él mediante el nodo fin. También podremos recorrer la lista de una forma algo distinta, ya que tener referencias al anterior nos permitirá ir y volver entre los nodos de la lista, excepto el primero, que no tiene anterior. El resto de las operaciones se mantienen con la misma complejidad.

- Lista circular: Partimos de la idea de lista simplemente enlazada, pero ahora además tendremos que el nodo siguiente al nodo fin será el principio de la Lista, generando así la Lista Circular. Implementar este tipo de lista resulta algo un poco más complejo, ya que se debe mantener este 'circulo', es decir, siempre que se inserte al final se debe asegurar que el nuevo fin tenga como siguiente al incio, y cada vez que se inserta en el inicio, se debera actualizar el siguiente del nodo fin. En cuanto a sus operaciones, no hay grandes cambios de complejidad algorítmica.

- Lista circular doblemente enlazada: Combinando las dos implementaciones mencionadas arriba de ésta, obtenemos la Lista Circular Doblemente Enlazada. En principio, la gran diferencia aquí será que tendremos una Lista Circular, cuyos nodos tendrán, además de un elemento y un nodo siguiente, ahora un nodo anterior, como la lista doblemente enlazada. Esto provoca que se pueda ir y volver entre nodos, incluso entre el final y el principio, pero no hay otras grandes diferencias más allá de este hecho. Las complejidades algorítmicas no sufren grandes cambios, siendo lo más distinctivo que quitar (al final) será O(1), mientras el resto de las operaciones se mantienen con una complejidad parecida a las demás.

3. Detalles de implementación

En la implementación de estos tres TDAs, a mi modo de verlo, no hubo grandes decisiones de diseño tomadas, ya que habia un camino bastante claro para el desarrollo del mismo. Algunas cosas a resaltar en estos términos es que se aprovechó la implementación del TDA Lista (para ahorrar tiempo y trabajo) para los TDAs Cola y Pila, ya que pueden funcionar como una Lista si se le impone ciertas restricciones. Dichas restricciones radican en que, tanto para la pila como para la cola, se utilizó el nodo inicio de la lista como el tope/frente. De este modo, cuando se apila se estará insertando al principio de la lista, y cuando se desapila se quita del principio de la lista. En cuanto a la cola, cuando encolamos se insertará al final de la lista, mientras que cuando desencolamos se quitará del principio de la lista. En cuanto a ver el tope/ver el frente, será O(1) porque aprovechamos la referencia al nodo inicio de la lista.
Destruir será la única operación O(n), operación para la cual se utiliza el destructor de la lista.
Para acceder al elemento del frente/tope de la cola/pila, entonces, se utiliza lista_ver_primero, y para saber si esta vacia simplemente se utiliza lista_vacia.

En lo que respecta al iterador externo, el funcionamiento de lista_iterador_avanzar se encuentra ejemplificado en la sección '4.6 Avanzar iterador externo de una lista'. Este iterador no tiene gran complejidad sobre su implementación, simplemente se decidió apuntar corriente al nodo a recorrer, en lugar de tomar otro camino que implicaba reservar memoria para corriente, y así copiar los contenidos del nodo que se esta recorriendo al nodo corriente.

3.1. Sobre insertar_en_posicion

Para implementar esta función hubo que tomar en cuenta ciertos casos borde, los cuales eran insertar en una lista vacia, insertar en una posicion más grande que la lista o incluso insertar en el final de la lista. Con ésta última, simplemente se utiliza lista_insertar, ya que es la función

encargada de insertar al final de la lista.
Al querer insertar en cualquier posción, distinta del principio (0) o el final, lo que hace esta función es llegar al nodo anterior a la posición a insertar, establecer como siguiente del nuevo nodo el siguiente de este anterior (el cual resulta estar en la posción en la que se quiere insertar) y actualizar el siguiente del nodo anterior al nuevo nodo. De esta forma, se habrá insertado efectivamente un nodo en una posición distinta del principio y del fin.

En cuanto a insertar al fin, se hace uso de lista_insertar, lo cual crea un nuevo nodo con el elemento deseado y siguiente NULL, y cambia el siguiente del nodo fin a este nuevo nodo, actualizando también el nodo fin de la lista al nuevo nodo.
Por último, insertar al principio de la lista crea un nuevo nodo, el cuál tendrá como siguiente al nodo inicio de la lista, y luego simplemente se actualiza el nodo inicio de la lista al nuevo nodo.
Los detalles de esta función pueden verse claramente en la sección de diagramas correspondiente a insertar e insertar en posición.

3.2. Sobre quitar_de_posicion

Quitar de posición tenía ciertos casos bordes algo similares a los de la funcion anterior, como quitar de posicion de una lista NULL, quitar de posición mayor al tamaño de la lista y quitar de posición de una lista de 1 elemento.
Para el funcionamiento de esta función, la logica es llegar al nodo anterior al que se quiere eliminar, para establecer como siguiente de dicho nodo, el siguiente del nodo a eliminar, guardando el nodo a eliminar en un nodo auxiliar para posteriormente poder devolver el elemento y no perder la referencia a él. Además, si la posición a eliminar es el tamaño de la lista menos uno, esta función actua como lista_quitar, la cual itera la lista hasta llegar al nodo anterior a nodo fin, para luego establecer el nodo fin como ese nodo anterior y liberar el nodo fin antiguo (almacenado en una variable auxiliar) pero no sin antes guardar en una variable el elemento que se debe devolver mediante el return.

Nuevamente, todo esto puede verse algo más en detalle en la subsección correspondiente en la sección de diagramas.

3.3. Sobre lista_destruir

La función de destrucción de lista primero debe verificar si no nos estan pasando una lista NULL, para luego verificar si la lista tiene nodos, y de ser así, itera toda la lista liberando cada uno de los nodos mediante un bucle while. En este caso se utilizó el bucle while simplemente por una cuestión de gustos, ya que podría utilizarse su alternativa en el bucle for con el mismo fin, iterando con un indice que deberá ser menor a lista_tamanio.
Por último, se libera el puntero a la lista y se sale de esta función.

El funcionamiento de esta función se encuentra detallado en la sección siguiente bajo el apartado '4.5 Destrucción de la lista'.

4. Diagramas

Con el fin de aclarar un poco el funcionamiento de las funciones mencionadas anteriormente, veamos los siguientes diagramas:

4.1. Insertar elemento al final de una lista

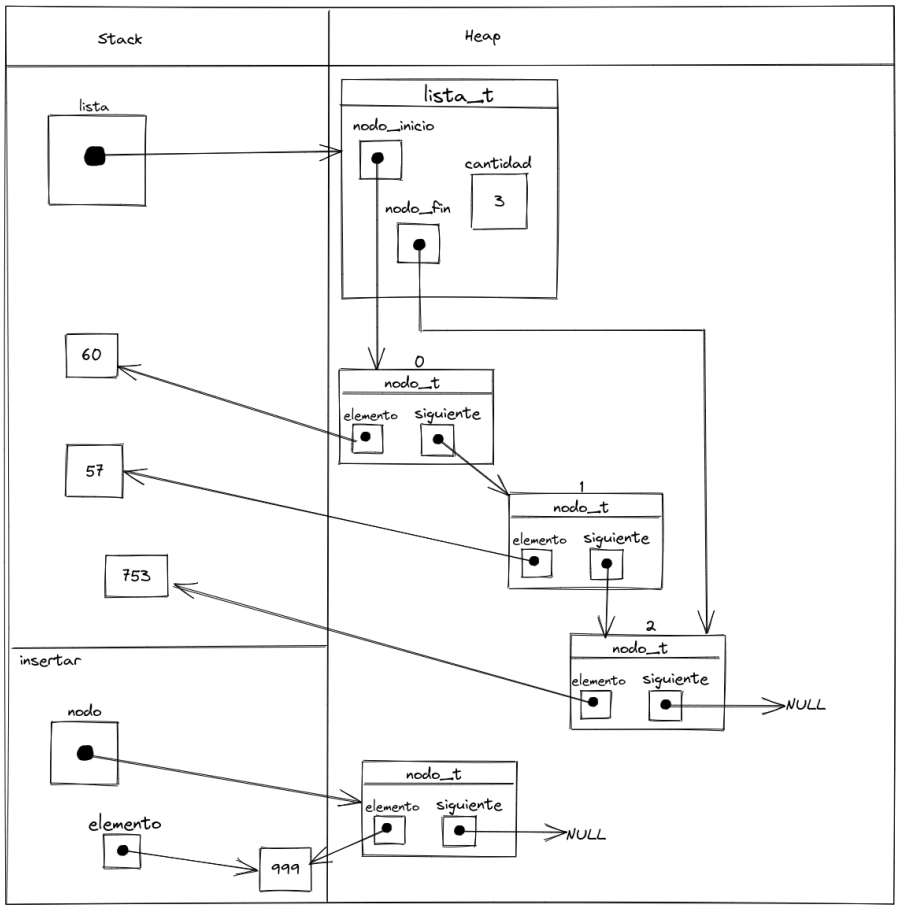

Figura 1: Creación del nodo a insertar al final

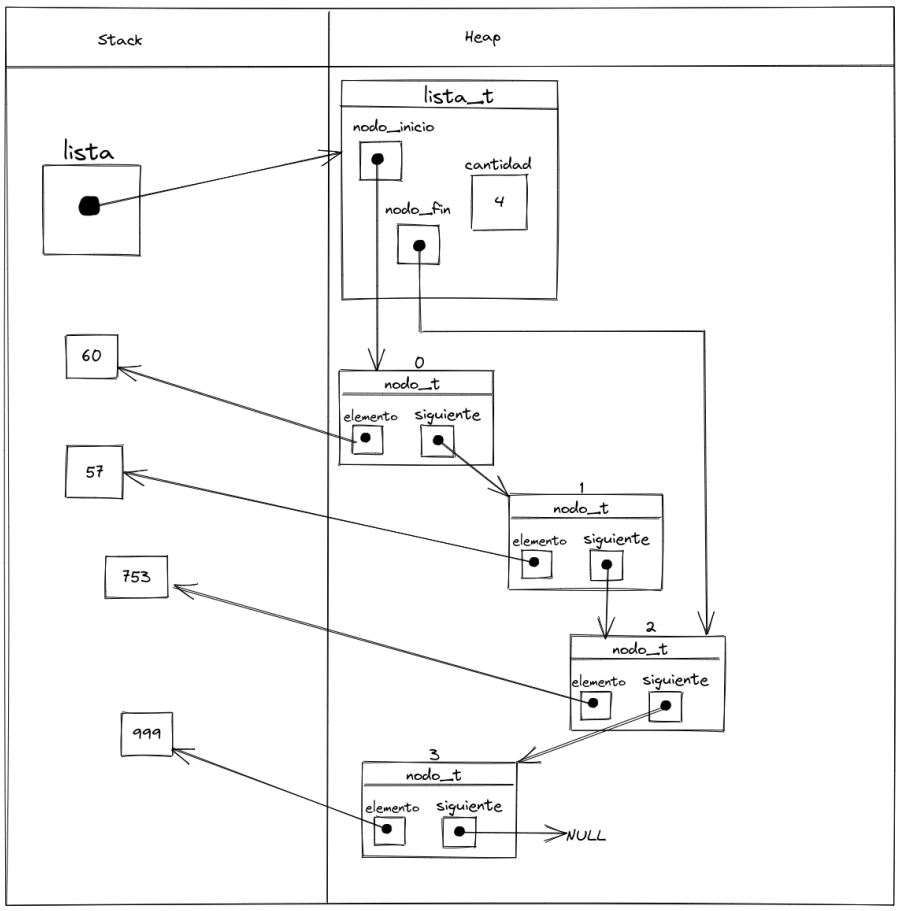

Figura 2: Estado de la Lista luego de insertar un nodo al final

4.2. Quitar un elemento del final de una lista

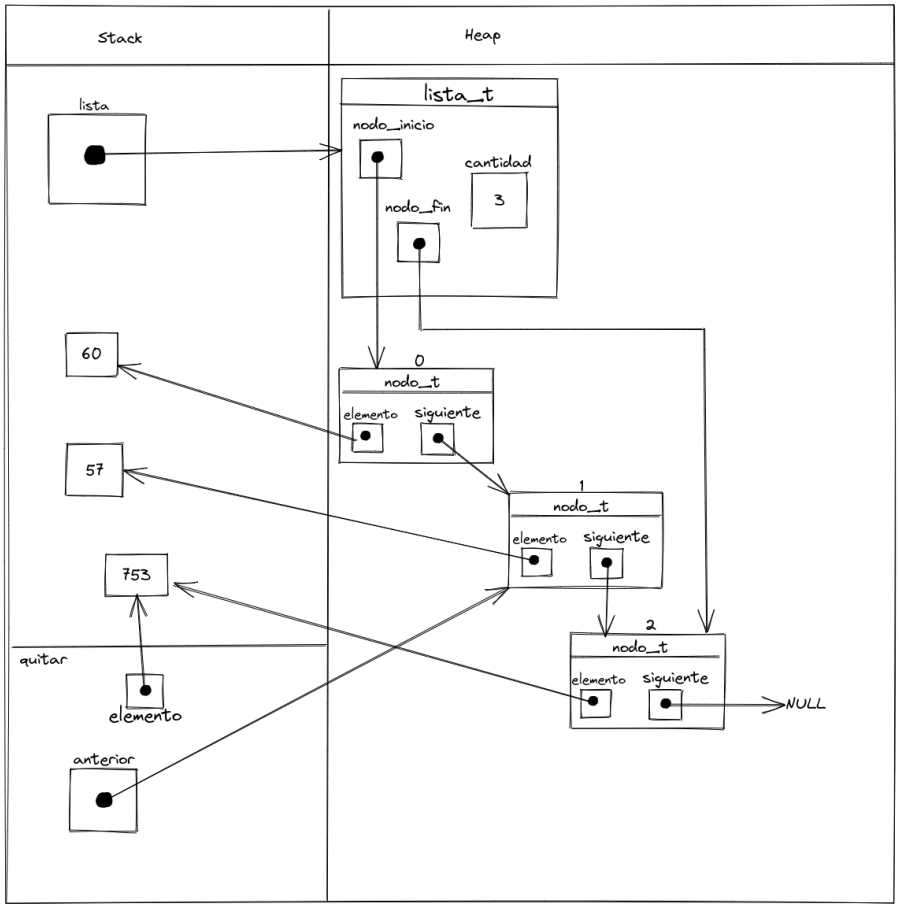

Figura 3: Diagrama de memoria en el contexto de lista_quitar

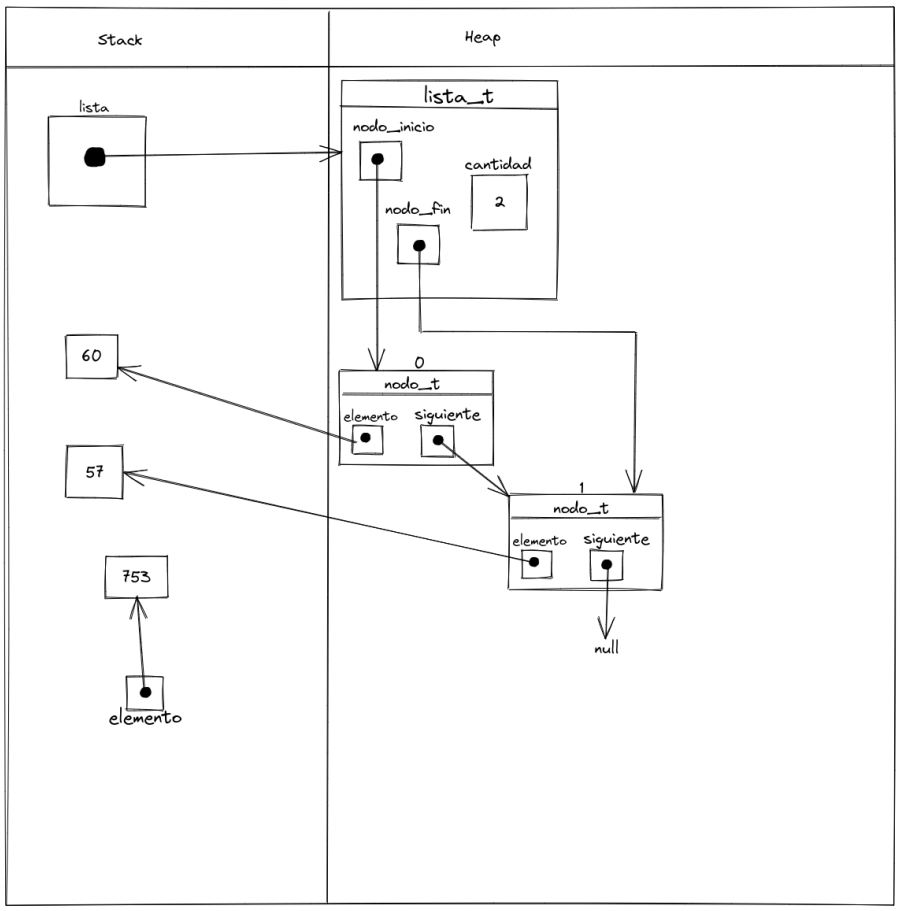

Figura 4: Diagrama de memoria luego de quitar un nodo al final de la lista y retornar el elemento

4.3. Insetar en el medio de una lista mediante insertar en posicion

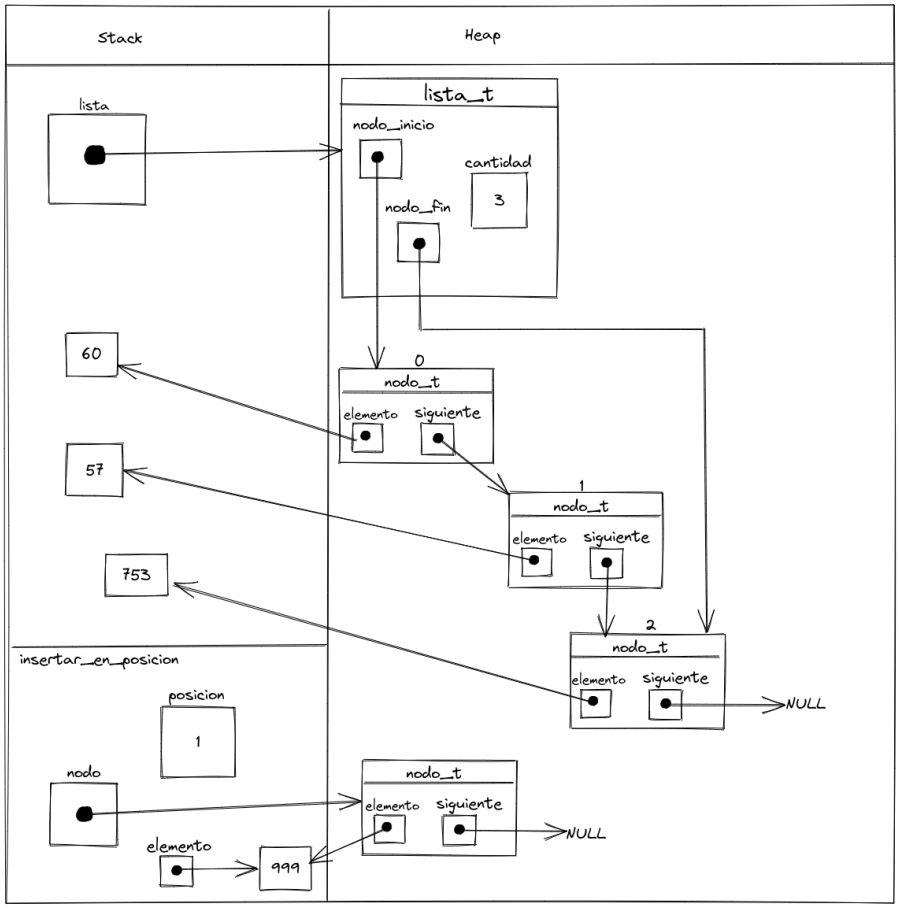

Figura 5: Creación del nodo a insertar en una posición

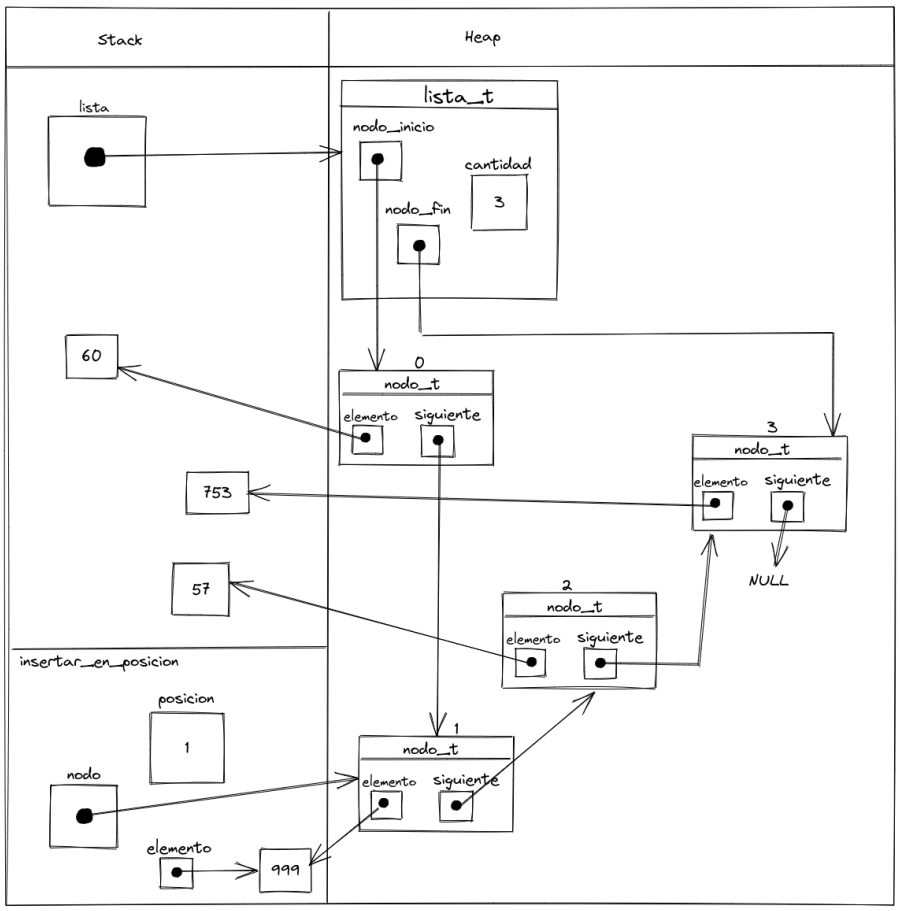

Figura 6: Diagrama de memoria luego de actualizar el siguiente del nodo en la posición anterior a la deseada

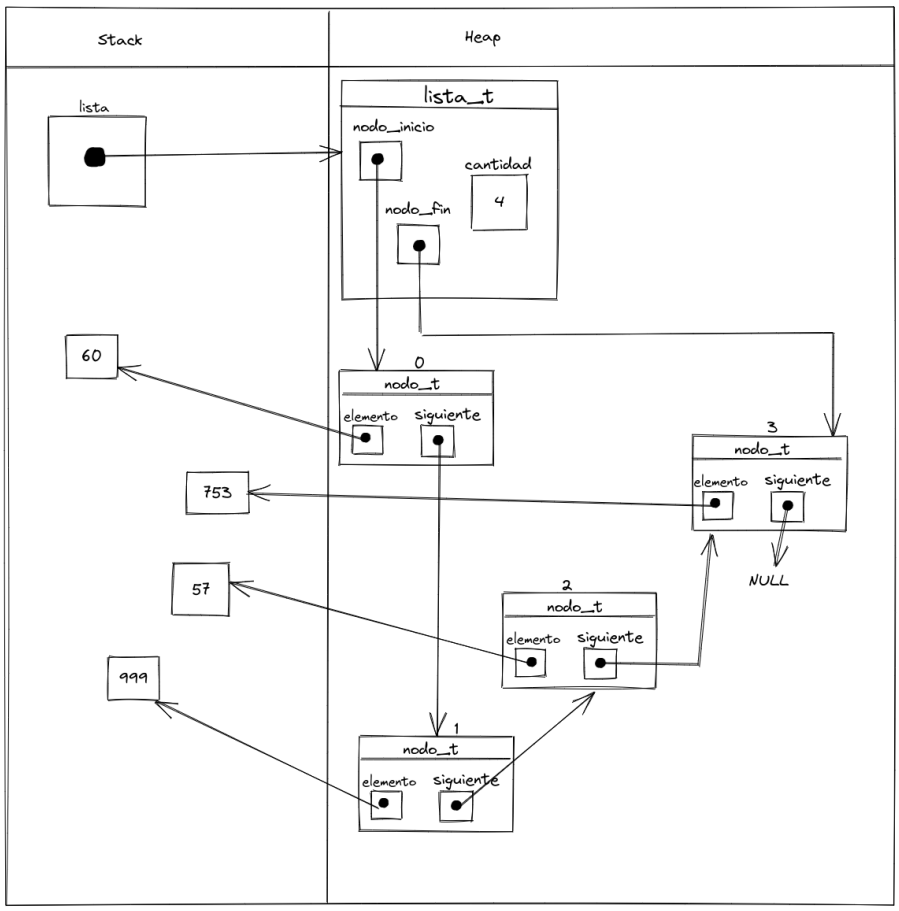

Figura 7: Diagrama de memoria luego de insertar un nodo en el medio de la lista

4.4. Quitar de una posición del medio

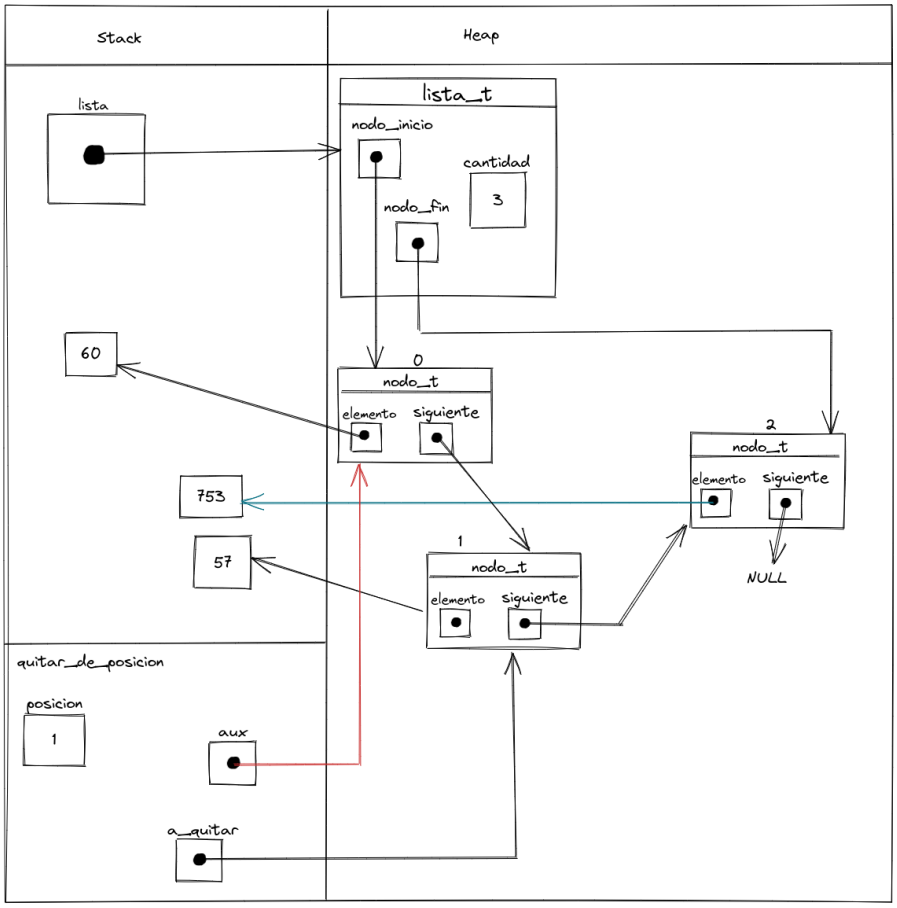

Figura 8: Primer paso al quitar de una posición (en este caso del medio)

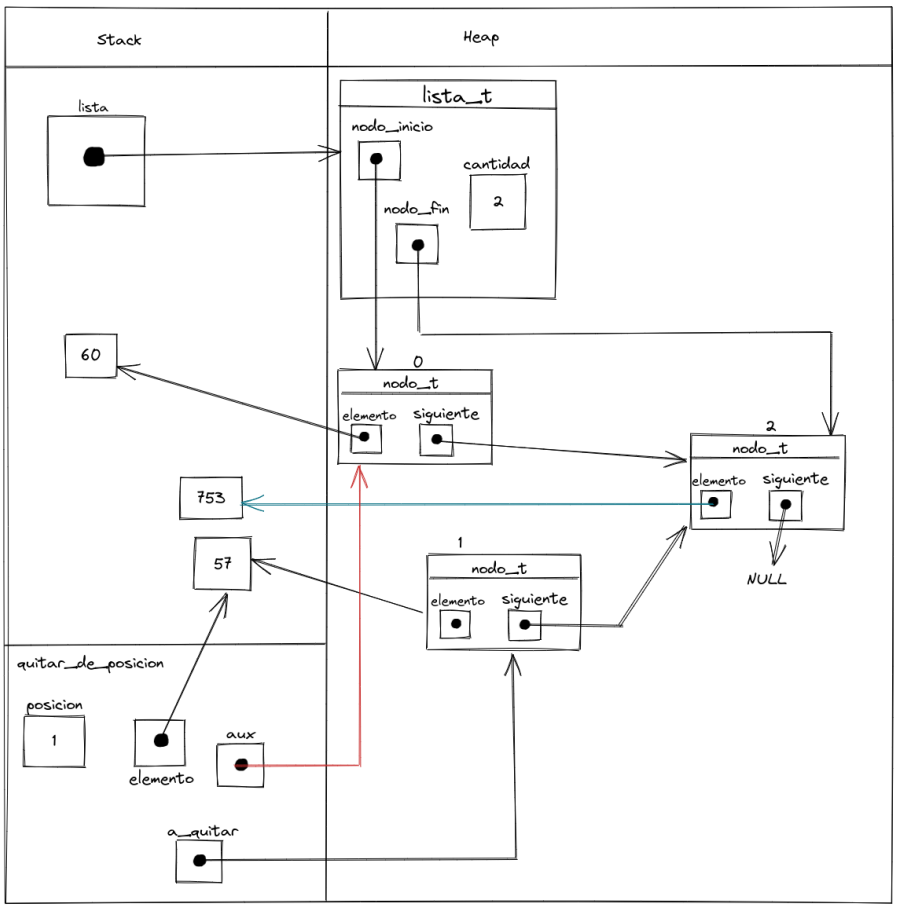

Figura 9: Diagrama de memoria luego de cambiar el siguiente del nodo aux (anterior de a_quitar) al siguiente del nodo a_quitar

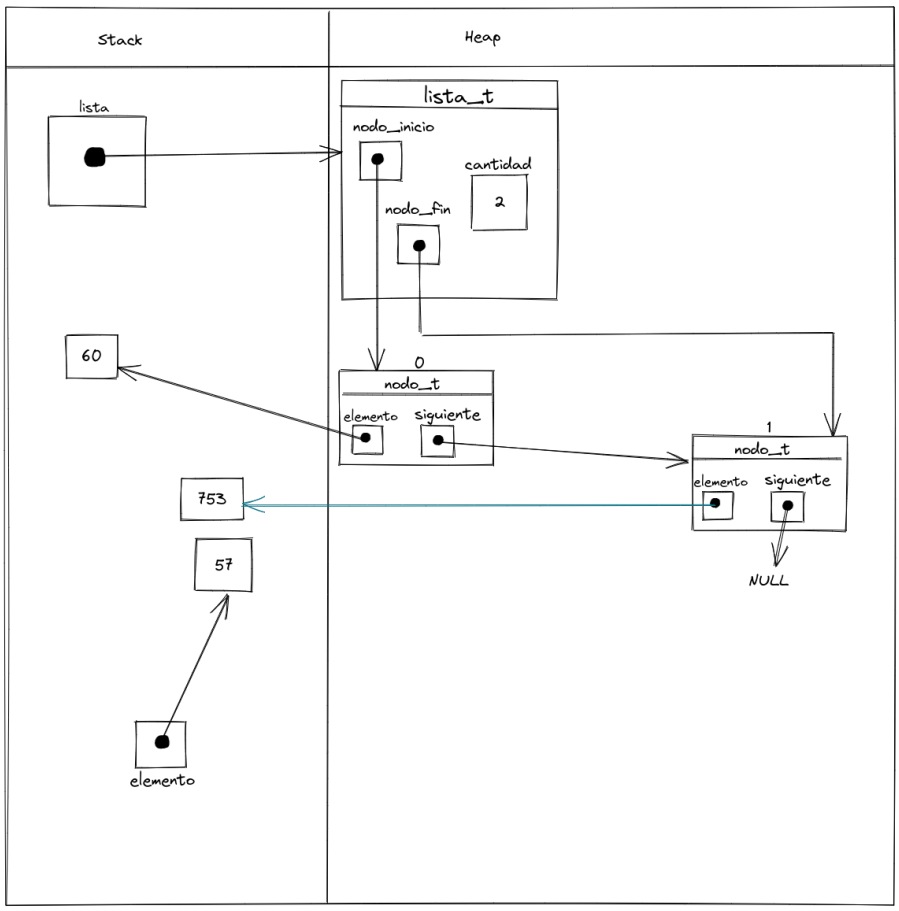

Figura 10: Diagrama de memoria luego de quitar el elemento

4.5. Destrucción de la lista

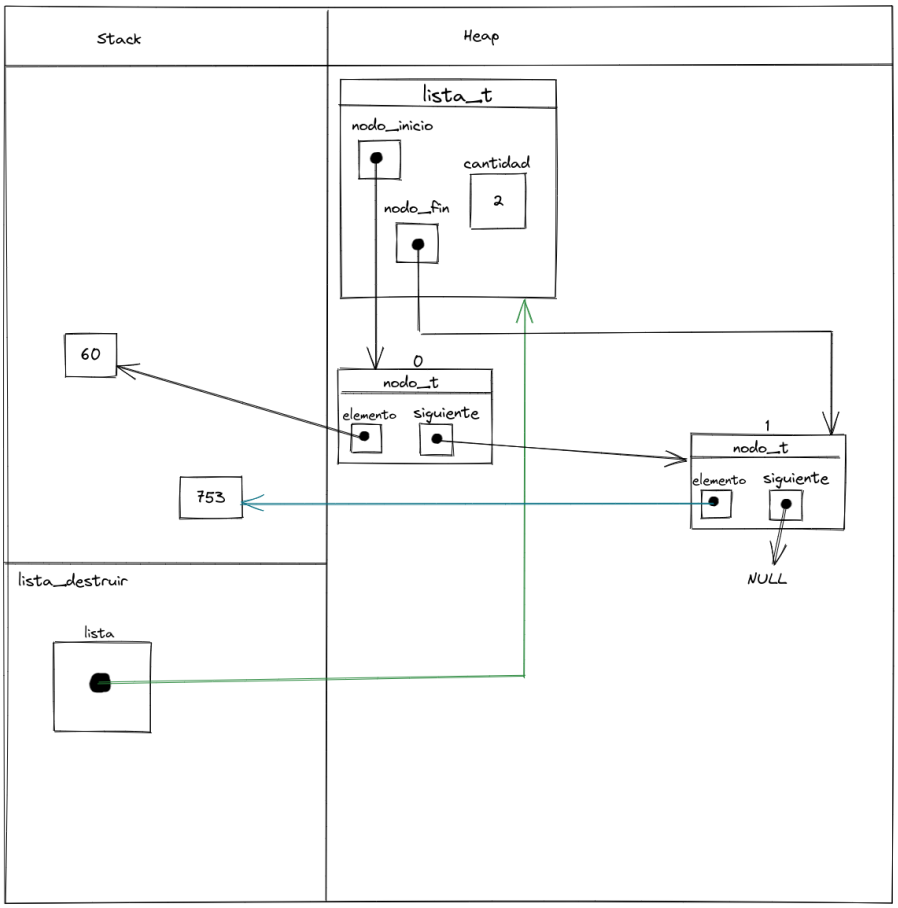

Figura 11: Diagrama de memoria al ingresar en el marco de ejecución de lista_destruir

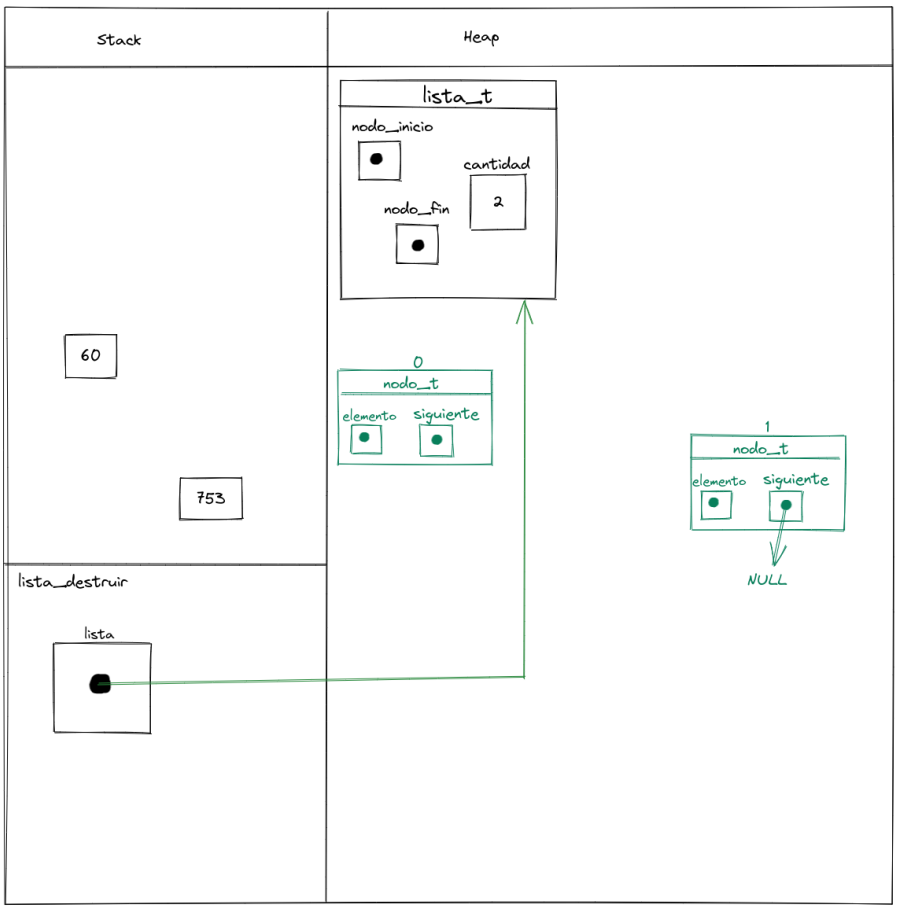

Figura 12: Diagrama de memoria luego de liberar los nodos de la lista

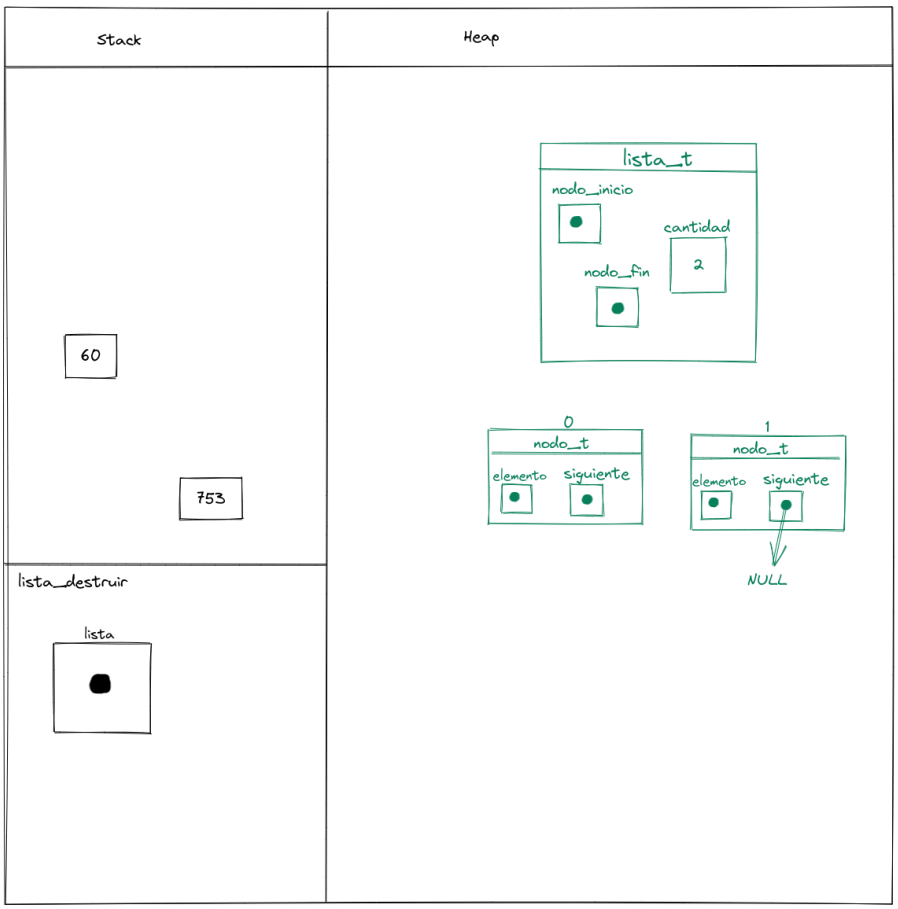

Figura 13: Diagrama de memoria tras liberar toda la memoria reservada por la lista

4.6. Avanzar iterador externo de una lista

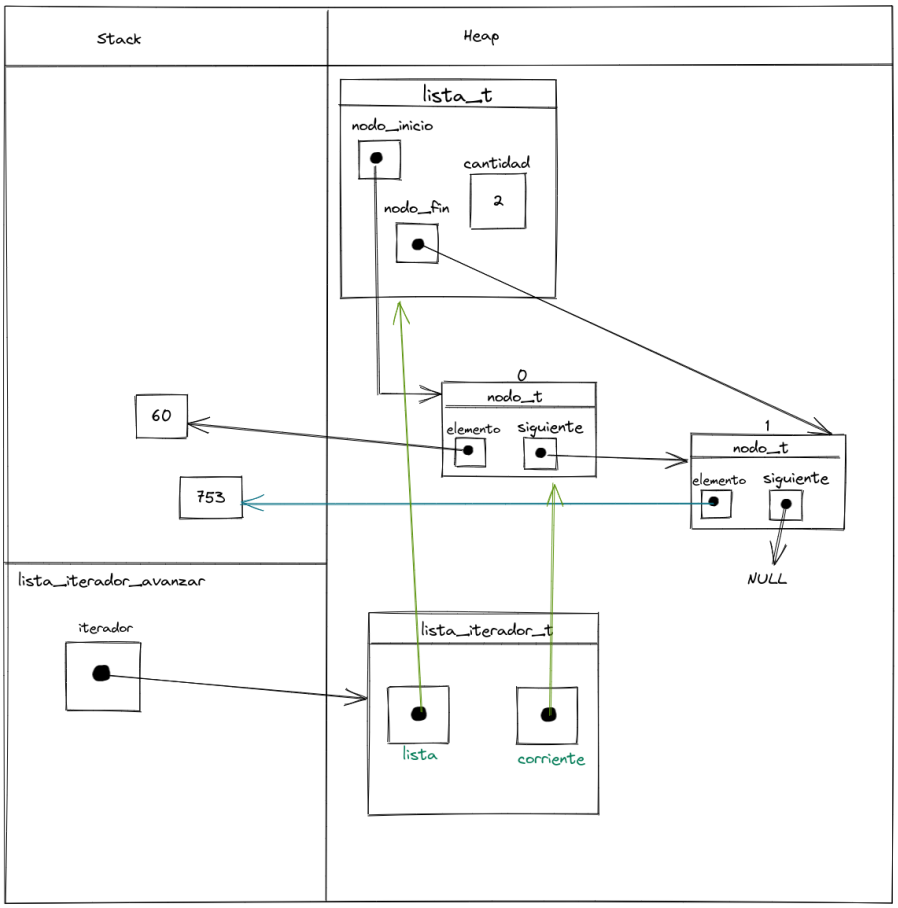

Figura 14: Diagrama de memoria para avanzar el iterador externo (recién creado, parado sobre el primer elemento)

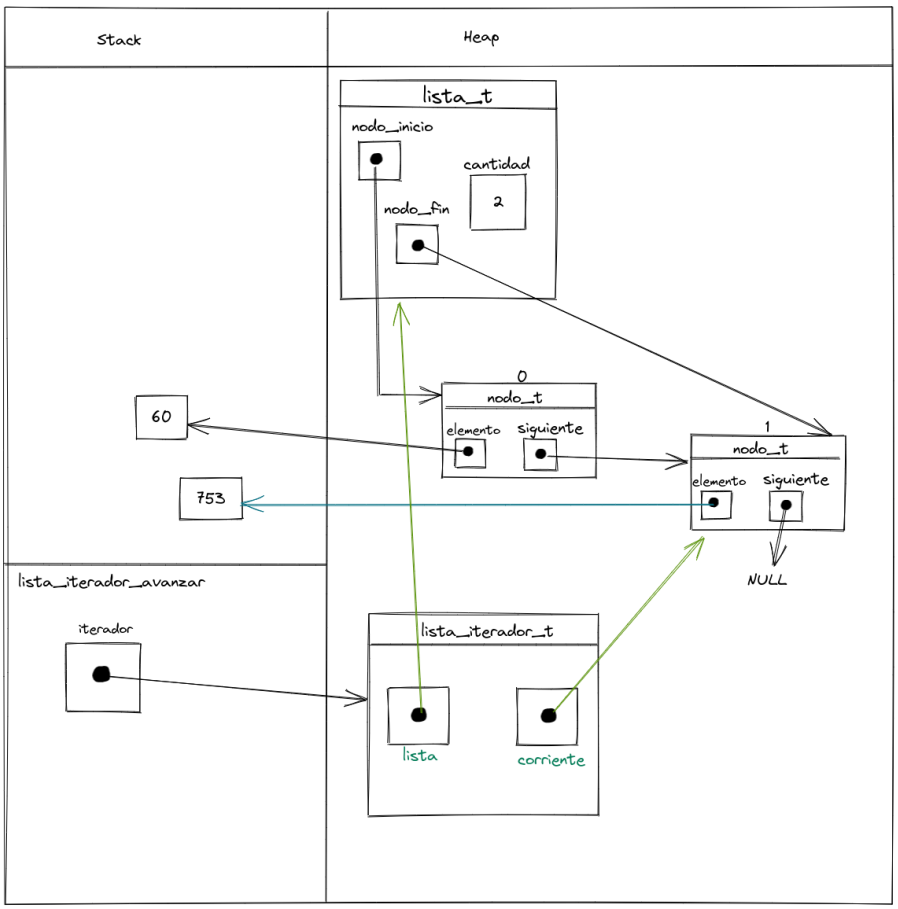

Figura 15: Diagrama de memoria tras avanzar el iterador

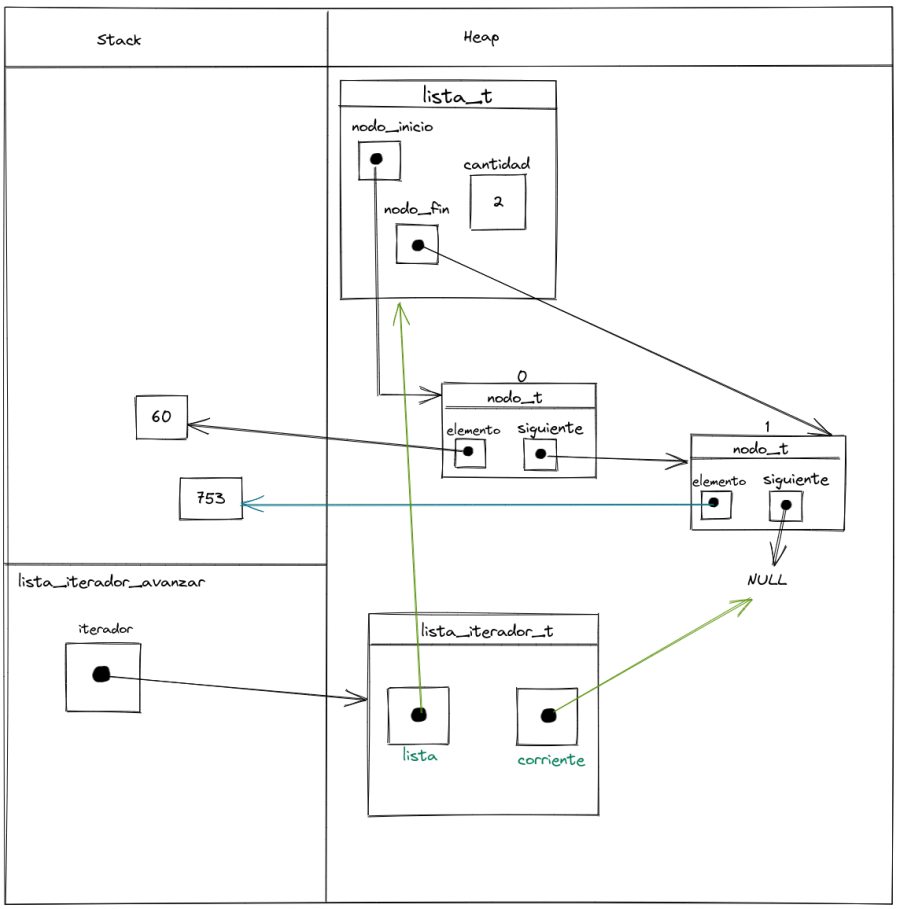

Figura 16: Diagrama de memoria tras llegar al ultimo elemento con el iterador externo

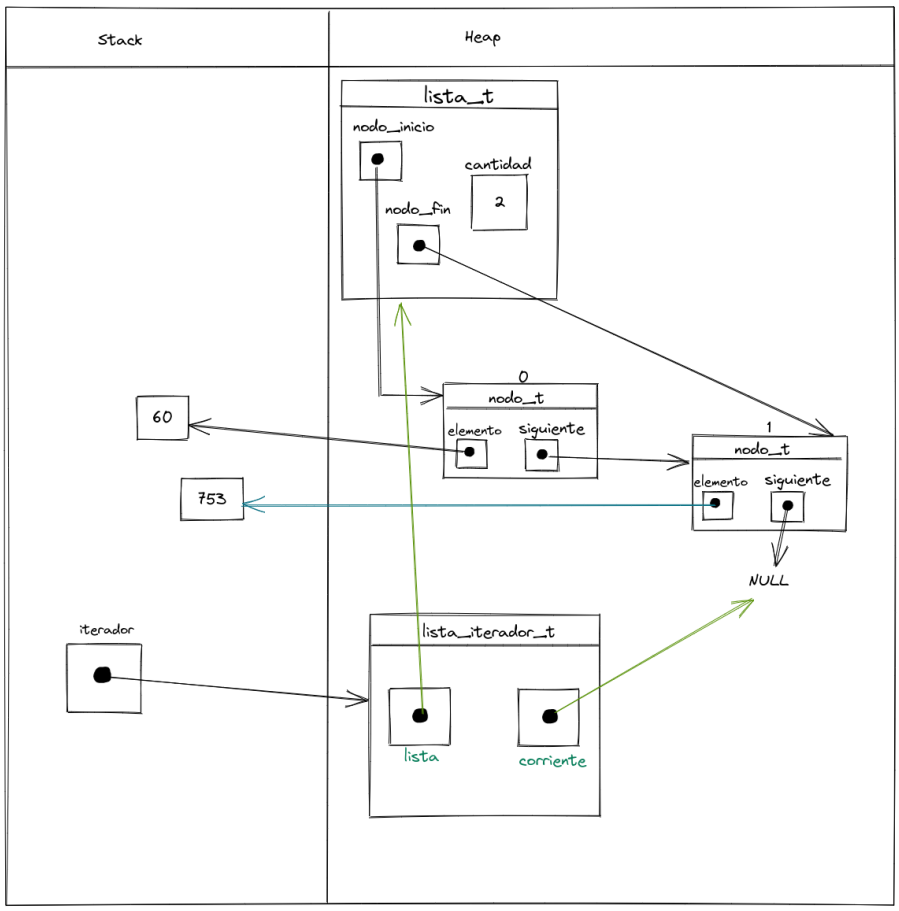

Figura 17: Diagrama de memoria tras finalizar la última ejecución de avanzar el iterador