Trabajo Práctico n — TDA LISTA

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

Alumno:	Dall'acqua,Denise
Número de padrón:	108645
Email:	ddallacqua@fi.uba.ar

1. Introducción

En esta entrega se verá reflejada la implementación de los TDA lista, pila y cola. Estos tienen como objetivo darle al usuario/cliente la biblioteca .h y el ejecutable .o para que pueda utilizar nuestras funciones y que no vea los bloques de código de las mismas.

2. ¿Qué es un TDA?

Antes de pasar a los tipos de TDA y la implementación que he hecho para cada función vamos hablar de qué es y para que sirve. TDA es la sigla de "tipo de dato abstracto" , un modelo del que se define una estructura o se referencia un concepto y sus operaciones. La abstracción permite de algún modo "simplificar" la realidad mediante la combinación de ciertas herramientas que nos frecen los lenguajes de programación, y poder crear nuevos tipos de datos. Los TDA están definidos por por las operaciones que pueden realizarse sobre ellos. La visión de aquel que utiliza un TDA es exclusivamente de caja negra. Cualquier detalle de implementación sobre el TDA es innecesario para aquel que hace uso del TDA.

2.1. TDA LISTA

El TDA lista esta basado en nodos enlazados. La estructura de este nodo esta compuesto por su elemento y un puntero que apunte al siguiente (si es una lista doblemente enlazada puede tener otro puntero que apunte al nodo anterior). La estructura de la lista esta compuesta por un puntero que apunte al primer nodo de la lista (nodo_inicio) y al nodo final (nodo_fin). Por medio de estos deberíamos poder movernos sobre ella y poder hacer determinadas operaciones, que por medio de algunas funciones que se han implementado podemos hacerlas.

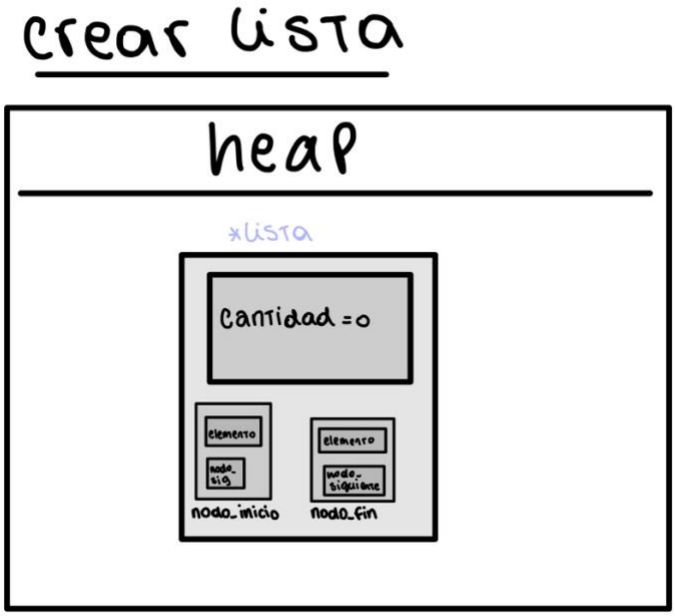
```
typedef struct nodo {
    void *elemento;
    struct nodo *siguiente;
} nodo_t;

typedef struct lista {
    nodo_t *nodo_inicio;
    nodo_t *nodo_fin;
    size_t cantidad;
} lista_t;
```

FIGURA.1 ESTRUCTURAS DE TIPO DE DATO NODO_T Y LISTA_T

A CONTINUACIÓN SE DESCRIBIRÁ CADA UNA DE LAS OPERACIONES IMPLEMENTADAS CON SUS ESPECÍFICOS DIAGRAMAS Y COMPLEJIDAD COMPUTACIONAL.

2.1.1 Crear_lista()

En esta función simplemente crearemos la lista almacenando, por medio de un *calloc(),* almacenando memoria del heap y que ya quede inicializada. Osea sus punteros a NULL y "cantidad" que sea cero. Si este falla por algún motivo, va a retornar NULL.
Complejidad computacional -> O(1).



2.1.2 insertar_lista()

En esta función tiene la finalidad es poder insertar un nodo nuevo en la lista, ósea poder agregar un elemento mas en ella. Para eso Lo único de debemos hacer es seguir una serie de pasos para que ningún puntero sea olvidado y que luego no podamos liberar ese espacio de memoria o no poder llegar a él: Primero, verificamos si la lista no es NULL(sino se retorna NULL) , si esta esta creada correctamente se guarda memoria del heap para el nuevo nodo, asignándole el elemento que queremos agregar y su nodo siguiente lo inicializamos a NULL; luego, si hay mas nodos en la lista, primero el nodo que era "nodo_fin" , su "siguiente" ahora es el nodo nuevo . Si todavía no hay ningún nodo en la lista , el nodo_inicio va a apuntar al nodo nuevo; y por último, cambiamos el nodo_fin de la lista al nuevo nodo y sumamos la cantidad. Esto nos devolvería la lista actualizada.
Complejidad computacional -> O(1), ya que no estamos iterando nada, solo creando variables y inicializándolas.

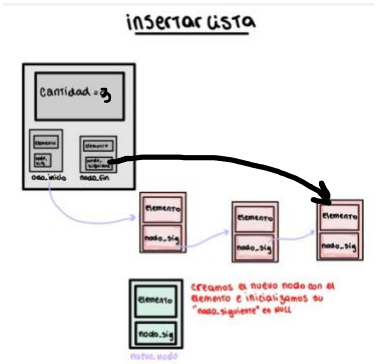


FIGURA 2. PASO 1 DE INSERTAR LISTA CON NODO

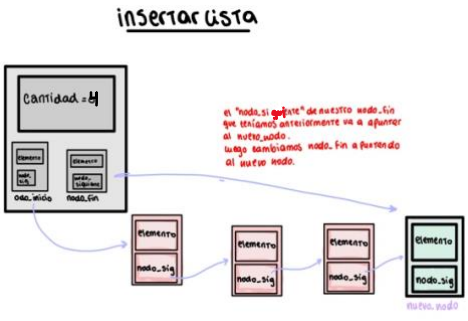


FIGURA3. PASO2 DE INSERTAR LISTA CON NODOS

2.1.3 Insertar_lista_posicion()

Esta función nos va a ayudar a poder agregar un nodo en la posición que queramos. Si la posición que deseamos acceder es mayor a la cantidad de nodos que tiene, este se va a añadir al final de la lista.

Si la posición es mayor a la cantidad o la cantidad es cero, se invoca a lista_insertar para que lo añada al final de la lista. Sino creamos el nodo nuevo, si la posición es cero lo va añadir al principio de la lista primero el nodo nuevo apuntando al nodo que anteriormente era nodo_inicio en la lista y luego hacemos que nodo_inicio apunte al nodo nuevo y subimos la cantidad. Pero si la posición es otro, con ayuda de un auxiliar y un bucle while nos vamos moviendo por la lista y cuando haya encontrado la posición deseada , hacemos que otro nodo auxiliar, en mi caso denominado auxiliar_2, apunte al auxiliar , el auxiliar ahora es su siguiente , y ahí podemos hacer el cambio para añadir el nodo nuevo: El siguiente de auxiliar_2 va a puntar al nodo nuevo , el siguiente del nodo agregado va a ser el auxiliar , incrementamos la cantidad de la lista y la devolvemos actualizada.

Complejidad computacional ->O(N) si la posición es menor a la cantidad total de la lista. Si la posición es mayor o la lista esta vacía, entonces es O(1) ya que estaríamos llamando a la función insertar_lista que es O(1).

FIGURA 4. PASO1 INSERCION EN POSICIÓN

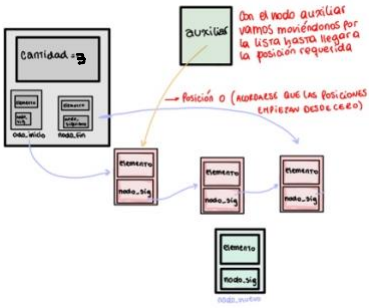


FIGURA 5. PASO2 INSERCION EN POSICIÓN

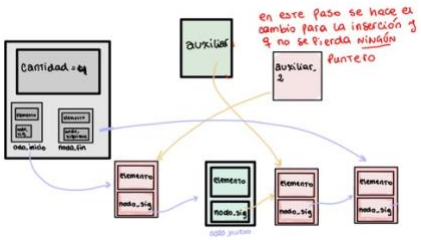
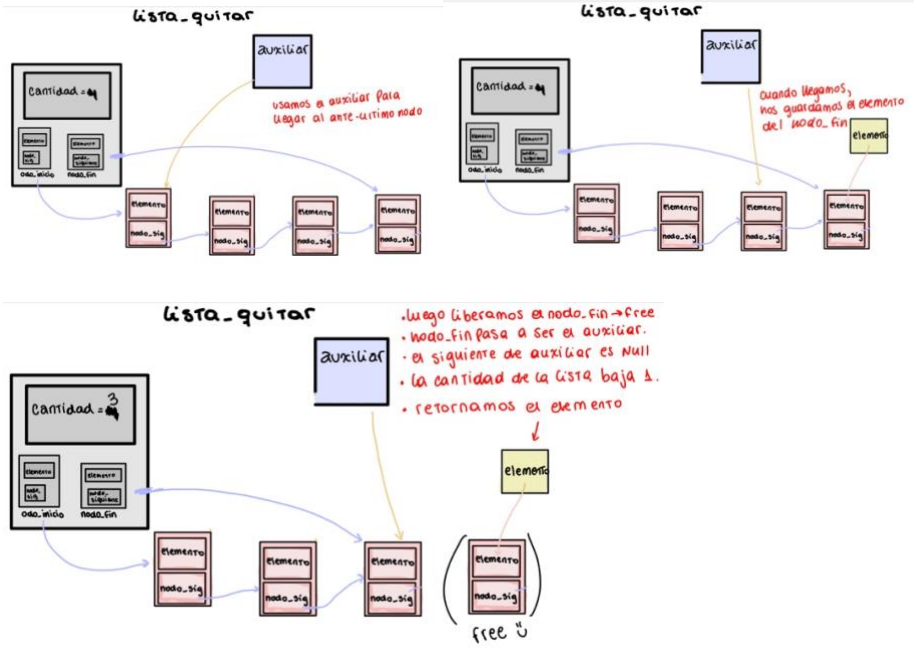


FIGURA 6. PASO 3 DE INSERTAR EN POSICION

2.1.4 *Quitar_lista()*

Para quitar lista usamos la misma lógica que en insertar. Queremos eliminar el ultimo no que hay en nuestra lista. Si la lista es NULL o la cantidad de esta es cero, retornamos con NULL. Si queda un solo elemento, nos quedamos con el elemento guardándolo en un puntero, le damos free al nodo e inicializamos en NULL los punteros nodo_inicio y nodo_fin, retornamos el elemento quitado. Si tiene mas de un elemento, se crea un auxiliar que nos ayude a encontrar el nodo que esta ante-ultimo para poder hacer el cambio. Cuando ya lo encontramos, nos guardamos el elemento de nodo_fin, le damos free y luego apuntamos como el final de la lista al auxiliar, le bajamos a la cantidad de la lista y retornamos el elemento eliminado.

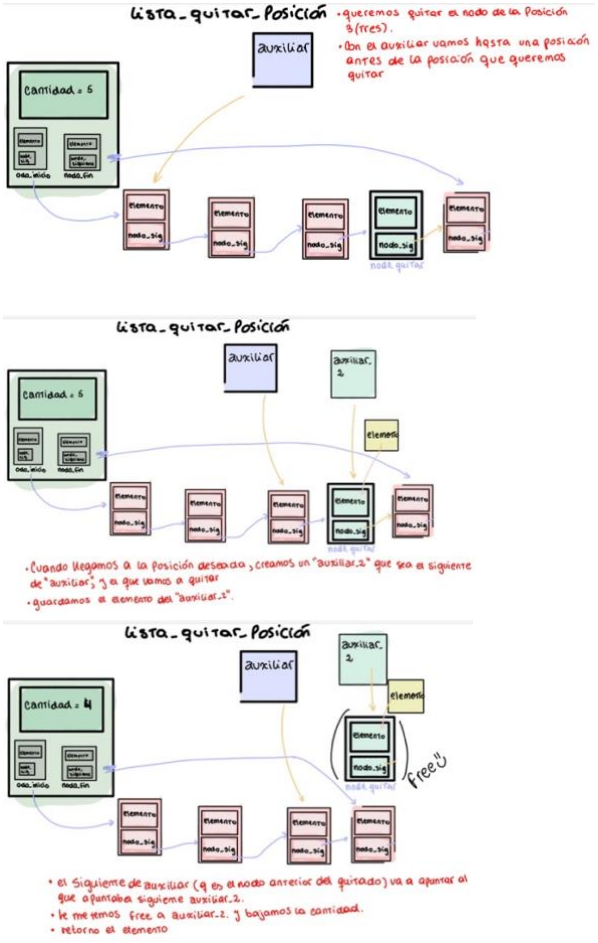
Complejidad computacional ->O(N) . Por que el auxiliar debe iterar por "n" elementos hasta llegar al ante-ultimo lugar y poder hacer el cambio de nodo_fin.



2.1.5 Quitar_lista_posicion()

Para quitar un nodo de una posición arbitraria, lo que primero verificamos es si la lista es null o no hay nodos en ella, en estos casos retornamos NULL. Si la posición a la que queremos acceder es mayor o igual a la cantidad de la lista -1 (ya que las posiciones empiezan desde el cero y la cantidad desde 1) , llamamos a la función quitar_lista para que haga el trabajo. Si la posición es 0(Cero) guardamos el elemento en un puntero, nos creamos un auxiliar que apunte al nodo inicio, luego apuntamos a nodo_inicio que sea el siguiente de auxiliar , le metemos free al auxiliar , bajamos la cantidad de la lista y retornamos el elemento eliminado. Si es otra posición, con ayuda de este auxiliar mencionado anteriormente y un bucle, nos paseamos por la lista hasta encontrar la posición deseada y hacemos el intercambio: Nos guardamos el elemento en un puntero,nos creamos otros auxiliar , en mi caso se denomina auxiliar_2, que se guardará el siguiente del auxiliar. El siguiente de auxiliar apuntará al nodo que se encuentra en auxiliar_2 , se da free a auxiliar_2 , se baja la cantidad de la lista y luego retornamos el elemento eliminado.

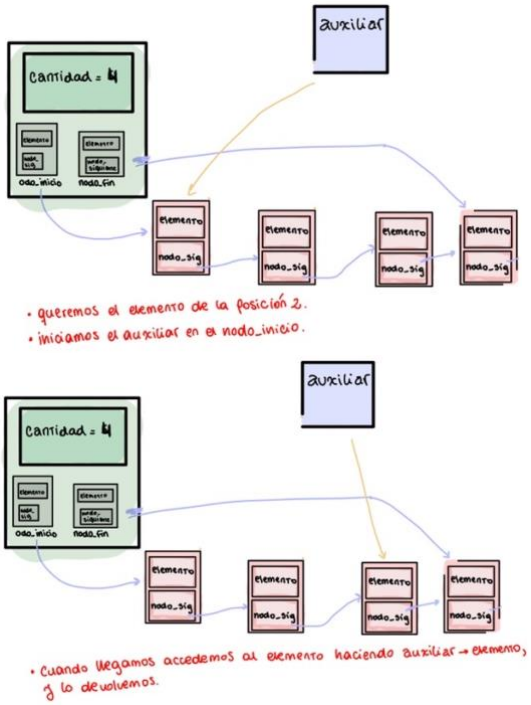
Complejidad computacional -> O(N) si la posición es menor a la cantidad total de la lista. Si la posición es mayor, entonces es O(1) ya que estaríamos llamando a la función insertar_lista que es O(1).



2.1.6 Lista_elemento_en_posicion()

En esta funcion nos va a dar el elemento a partir de la posición que queramos acceder. Si la lista es NULL o la posición es mayor a la cantidad, esta retornará NULL. Si no, por medio de un auxiliar y un bucle accederemos a la posición y retornaremos su elemento.

Complejidad computacional->O(N) ya que hay que iterar linealmente, por los n elementos hasta llegar a la posición que deseamos. Si la posición que nos pasan es mayor a la cantidad seria O(1) ya que no tiene que realizar la iteración, y no hay complejidad.



2.1.7 Lista_buscar_elemento()

Esta funcion nos va dar el elemento a partir de un puntero a una función llamado comparador. Si la lista o la función "comparador" son NULL, esta función retornará NULL. Si no, se creará un auxiliar que apunte al nodo_inicio de la lista y luego en un bucle podamos acceder a cada nodo de esta, y si cumple con la condición de la función "comparador" entonces retornará el elemento, si no cumple retorna NULL.

Complejidad computacional->O(N).

2.1.8 Lista_primero()

En esta funcion se volverá el elemento de la primera posicion si la lista es válida y su cantidad tiene más de 0, sino retornará NULL.

Complejidad computacional->O(1)

2.1.9 Lista_tamanio()

En esta función se devolverá la cantidad de nodos que tiene una lista válida. Por lo contrario, devolverá 0 (cero).
Complejidad computacional ->O(1)

2.1.10 Lista_ultimo()

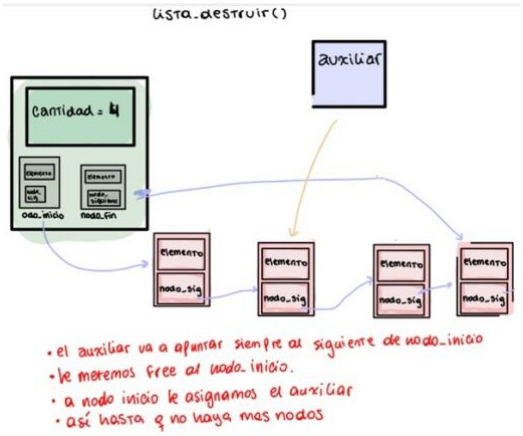
Esta funcion devolverá el ultimo element de una lista válida con mas de 0 nodos. Por el contrario, retornará NULL.
Complejidad computacional -> O(1)

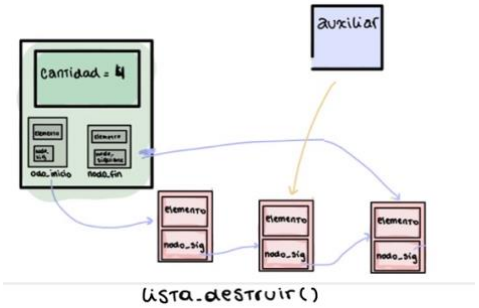
2.1.11 Lista_vacia()

Esta función booleana nos va retornal true si la lista es vacía (ósea con cantidad = 0) o una lista NULL. Por el contrario es false.
Complejidad computacional -> O(1)

2.1.12 Lista_destruir() y lista_destruir_todo()

Estas dos funciones van a ayudarnos a poder liberar de la memoria que previamente reservamos en el heap, pero la diferencia es que lista_destruir_todo tiene una función que nos permitirá liberar el bloque de memoria del elemento (si lo tuvimos que crear en el heap) , y lista_destruir solo se encarga de los nodos de la lista y de esta.
Para lista_destruir_todo nos fijamos si lo que nos pasan es una lista valida. Si la cantidad es cero, entonces solo hacemos free a la lista. Si no, dentro de un bucle(que va a iterar hasta que nodo_inicio sea válido) se crea un auxiliar que se queda con el siguiente del nodo_inicio, luego si la funcion no es NULL, se implementa el puntero a función como parámetro el elemento del nodo_inicio , se hace free al nodo_inicio , y se pone como nodo_inicio al auxiliar. Cuando el bucle termina se hace free a la lista.
Para lista_destruir , llamo a lista_destruir_todo pasandole como NULL el parámetro del puntero a función, así no escribo código de mas.
Complejidad computacional -> O(n)





2.2 Iteradores

En esta sección se explicará como fue la implementación de los iteradores para la lista de nodos enlazados. Un iterador se refiere al objeto que permite al programador recorrer un contenedor, particularmente listas. Hay dos tipos: El interno , en el cual no vemos la implementación de este recorrido, y el externo el cual por medio de tres funciones el usuario va a poder hacer lo que quiera con él.

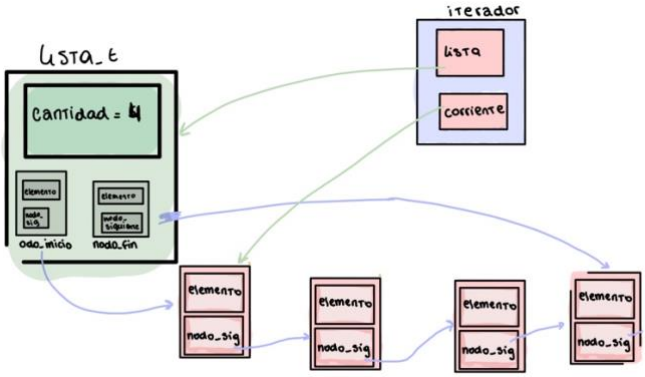
```c
typedef struct lista_iterador {
    nodo_t *corriente;
    lista_t *lista;
} lista_iterador_t;
```

STRUCT DE LISTA_ITERADOR

2.2.1 *Iterador externo*

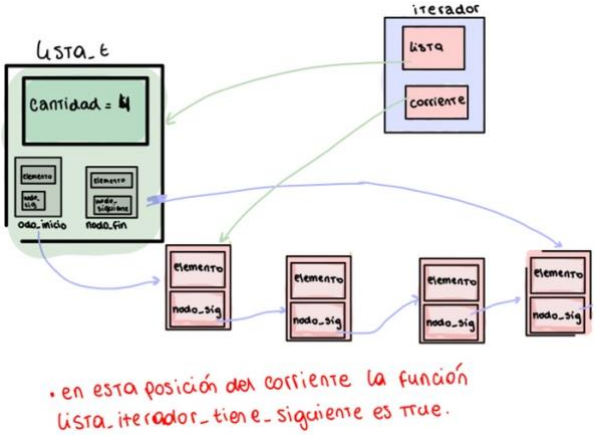
2.2.1.1 lista_iterador_crear()

En esta función se va a crear nuestro iterador si se le pasa una lista válida, sino retornará NULL. Corriente va a apuntar al nodo_inicio e lista a la lista.
Complejidad computacional ->O(1)



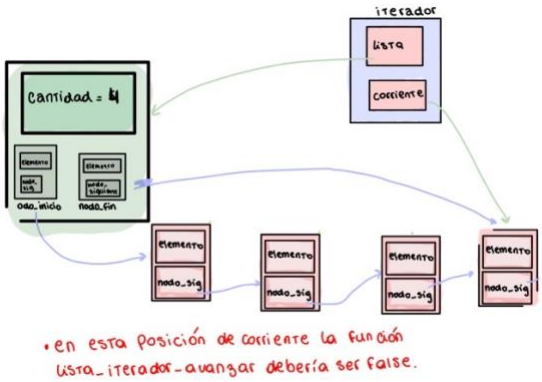
2.2.1.2 lista_iterador_tiene_siguiente()

En esta función booleana nos va a devolver true si hay corriente, si no hay corriente o el iterador es NULL nos va a retornar NULL.
Complejidad computacional -> O(1)



2.2.1.3 lista_iterador_avanzar()

En esta función booleana va a retornar true si luego de avanzar al siguiente nodo, no fue NULL. En caso contrario o por algún error, da false.
Complejidad computacional -> O(1)

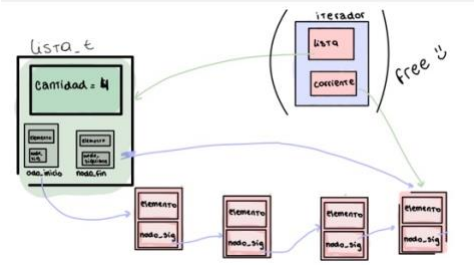


2.2.1.4 lista_iterador_elemento_actual()

En esta función si el iterador y/o su corriente no son NULL, va a devolver el elemento del corriente actual.
Complejidad computacional ->O(1)

2.2.1.5 iterador_destruir()

Esta función aplica free al iterador para liberar la memoria reservada previamente.
Complejidad computacional -> O(1).



2.2.2 *Iterador interno ->lista_con_cada_elemento*()

En el iterador interno, que la llamamos lista_con_cada_elemento, el usuario va a poder pasarle por parámetro la funcion booleana que desee y que le retornara cuantas veces tuvo que iterar hasta llegar a que se cumpla la condición de la función en relación a los elementos de la lista.
Si la lista, la función o el tamanio de la lista es cero , esta retornara automáticamente cero. Pero si no, con ayuda de un auxiliar recorremos la lista dentro de un bucle que tenga como

condición que auxiliar no sea NULL y que el booleano que he creado "avanzar" sea true siga iterando; si la función se cumple, la variable avanzar va a ser false y va a dejar de entrar en el bucle. Cada vez que itere, el contador va a ir sumando uno. Cuando salga de este, la función va a retornar el contador.
Complejidad computacional -> O(N).

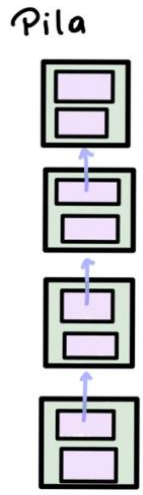
3. Pila y Cola

Para las funciones de pila y cola he reutilizado las funciones de la lista, casteando al tipo de dato que se pedía.
Los tipos de datos tanto pila_t y cola_t son estructuras opacas, ósea no sabemos que hay dentro de ellas. Pero podemos utilizarlas con ayuda de las funciones hechas y no es necesario saber que tienen dentro.

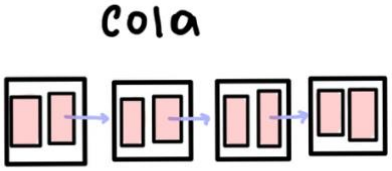
```
typedef struct _cola_t cola_t;
typedef struct _pila_t pila_t;
```

La complejidad computacional de cada función tanto de pila y cola, son las mismas que en la de TDA LISTA.

La pila y cola con nodos enlazados, como es en este caso podemos imaginarlos de la siguiente manera.



Este grafico de la pila en vertical es una ilustración para entender el tda, no es que la computadora sabe que es de abajo hacia arriba.
El único que podremos "ver" es el nodo_fin.

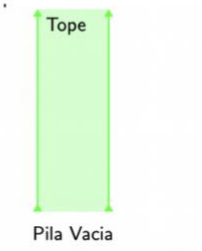


Esta es la presentación de la cola con nodos enlazados.

TDA PILA

- Es una coleccion ordenada de elemtnos en la que pueden insertarse y eliminarse por un extremo, denominda tope, sus elemtnos [?].
- Conjunto Minimo de operadores:

-**Crear** → Se realizan operaciones que tienen que ver con el tamaño de la pila, es decir, la cantidad de elementos que podrá tener. Ademas, se realiza la inicializacion de la misma(ejemplo, el estado de la misma es vacia).



Pila Vacia

-**Push**→Pone un elemento en la pila por el tope de la misma, haciendo que el tope de la pila pase a ser el nuevo elemento introducido.

-**tope**→Esta operacion permite observar el valor del tope de la pila.

-**pop**→Retira el elemtno del tope de la pila y mueve el tope de la pila al elemento anterior al extraido, si el elemento extraído es el ultimo la pila queda vacía.

-**esta_vacia**→Permite determinar si una pila tiene o no elementos. devuelve verdadero si la pila no posee elemtnos apilados y false si los posee. Es importante a la hora de querer vaciar una pila sin la necesidad de saber cuantos elementos estan apilados.

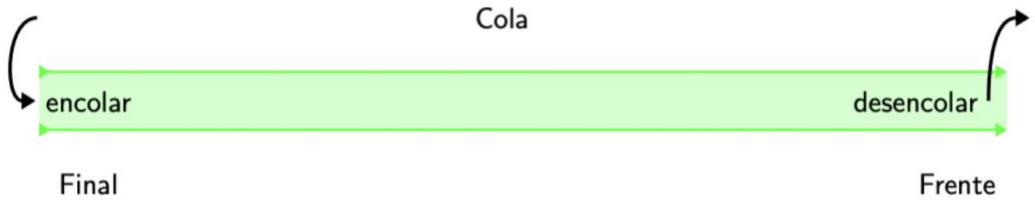
-**Destruir** → Se ocupa de liberar y limpiar todos los recursos que se utilizan para la creacion de una pila. Si la misma posee elementos en su interior debe vaciarse en el proceso de destruccion.

Las pilas las debemos pensar como una pila de cajas que contienen "cosas", a la única caja que podemos acceder es a la ultimo que apilamos, porque si queremos acceder a alguna que esté en el medio, toda nuestra pila se caería.

En esta implementación se utilizaron las funciones ya creadas del TDA lista, ya que las implementaciones de las operaciones mínimas de la pila son equivalentes, y el usuario no se va a concentrar en **cómo** están hechas las funciones, sino solamente en poder utilizarlas y punto. Entonces lo que se hice fue castear estas funciones al tipo de dato correspondiente , pila_t.

TDA COLA

- Es una estructura que posee dos extremos por los que se realiza operaciones. Un extremo es el inicio de la cola y el otro extremo es el final.
- Sirve para modelar procesos como la cola de un colectivo, supermercado, etc.



- Conjunto minimo de operaciones:

-**Encolar** → La idea es que los elementos se encolan por el final de la cola.

-**Desencolar**→Extrae del frente de la cola el elemento que se encuentra en él. Su estructura es el "First in, First out" o FIFO. El primer elemento que entra en la cola es el primer elemento que sale de la misma. Se respeta el orden de llegada.

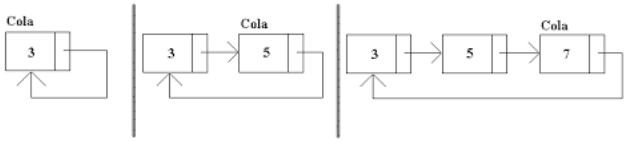
-Primero→Permite observar cual es el primer elemento del frente de la cola.

-**esta_vacia**→Permite determinar si una cola tiene o no elementos. Devuelve verdadero si la cola no posee elementos encolados, falso si paso lo contrario. Es muy importante a la hora de vaciar una cola sin la necesidad de saber cuantos elementos hay encolados.

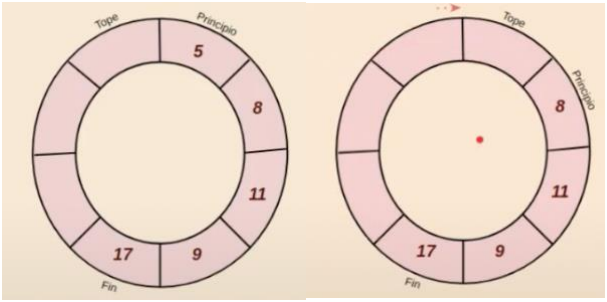
-**Crear**→ Se realizan las tareas de inicializacion de la cola y se detrmina la cantidad de elementos que la misma pueda tener.

-**Destruir** → Se ocupa de liberar y limpiar todos los recursos que se utilizan para la creacion de una cola. Si la misma posee elementos en su interior debe vaciarse en el proceso de destruccion.

COLA CIRCULAR VECTOR ESTÁTICO



En una cola circular en una vector estático en vez de que tope(ósea la capacidad máxima del vector) sea el final del vector, este va a estar siempre persiguiendo a "principio" , lo podemos pensar como un círculo. Si queremos desencolar usando una cola circular, tenemos que sacar el elemento que estaba en "principio" , que esta sea el elemento que anteriormente estaba en segundo lugar, y como el tope siempre persigue a principio este va a guardarse el lugar que anteriormente estaba el primer principio. Este proceso tiene como complejidad computacional O(1).



IMÁGENES SACADAS DE LA PRESENTACIÓN DE LA CLASE "PILA, COLA, LISTA" DONDE ESTÁ DESENCOLANDO EL ELEMENTO 5.