75.41 Algoritmos y Programación II
Tda Lista y sus Derivados

Dr. Mariano Méndez[1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

10 de marzo de 2020

1. Introducción

Si los programadores se limitaran a utilizar únicamente los tipos de datos que los lenguajes de programación proveen en forma primitiva, estos lenguajes no serian muy útiles. Por ello mediante la ayuda de un concepto central, llamado *abstracción* que permite a partir de la combinación de cierras herramientas primitivas que ofrecen los lenguajes de programación tradicionales, crear nuevos tipos de datos. Estas herramientas son:

- Tipos Primitivos: normalmente numéricos, caracteres, booleanos, entre otros.
- Funciones: Permiten generar y encapsular nuevas acciones o funcionalidad sobre los datos.
- bibliotecas: Agrupan conceptualmente acciones y datos para generar nuevos tipos de datos.
- Caja negra: Técnica que permite centrarse en ¿Que? realiza cierta función y no en el ¿Cómo?.

A continuación se presentarán tipos de datos abstractos clásicos, extraídos de la siguiente bibliografía [3] [2] [1].

1.0.1. listas y sus derivados

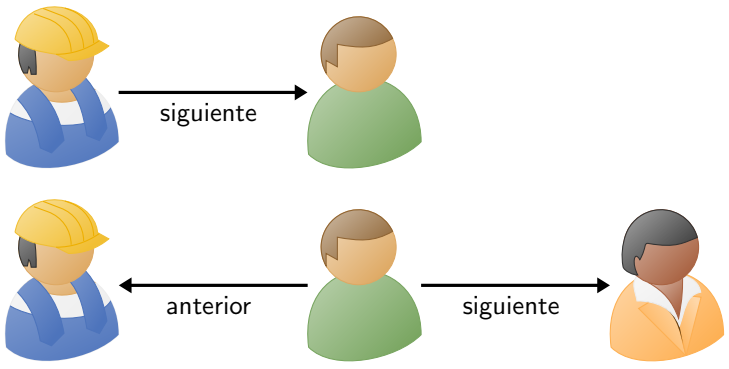
De la gran cantidad de tdas que existen hay un grupo muy común que son las listas y sus derivados:

- Pilas
- Colas
- Listas
- Listas Doblemente Enlazads
- Listas Circulares

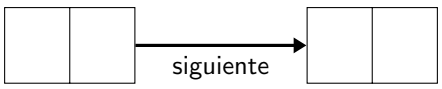
A partir de este punto se implementarán utilizando memoria dinámica. Cabe destacar que para cada tda *No existe una única implementación* y aqui se tomará como válida la de este apunte.

2. TDA Nodo Enlazado

El **nodo enlazado** es tal vez el **tipo de dato abstracto** más utilizado. La idea que abstrae un nodo enlazado es la propiedad de que una entidad pueda conocer a quien le sucede o antecede:



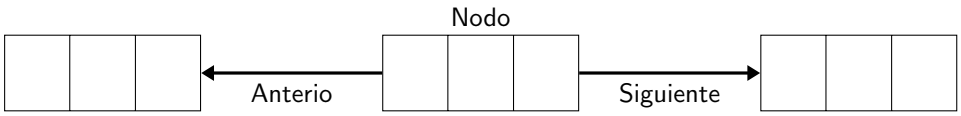
La idea principal es la de abstraer se, en este caso de personas, e intentar modelar o diseñar un tipo de dato que pueda almacenar en primera instancia el elemento siguiente, a partir de aquí a un elemento se lo llamara nodo. Desde lo más abstracto un nodo que puede enlazarse, se debe pensar de la siguiente forma:



Desde el punto de vista de la implementación, crear esta abstracción es muy sencilla:

```
1
2 typedef struct nodo{
3         dato_t elemento;
4         nodo_t* siguiente;
5 } nodo_t;
```

Otra variante es que el nodo conozca no sólo su siguiente sino también su antecesor:




```
1
2 typedef struct nodo{
3         dato_t elemento;
4         nodo_t* siguiente;
5         nodo_t* anterior;
6 } nodo_t;
```

3. TDA Pila

Una pila es una colección ordenada de elementos en la que pueden insertarse y eliminarse por un extremo, denominado **tope**, sus elementos [**?**].

Normalmente existe un conjunto de operaciones que se puede realizar sobre un tda. ¿Estas operaciones son estándares? No. Existe un pequeño conjunto que siempre debería estar y después las mismas variar según las necesidades del programador. Al conjunto que siempre debería estar se lo denominara **conjunto mínimo de operaciones**. Para el caso de una pila el conjunto mínimo de operaciones es:

- crear (create)
- poner (push)
- sacar (pop)
- tope (top)
- esta_vacia (is_empty)
- destruir (destroy)

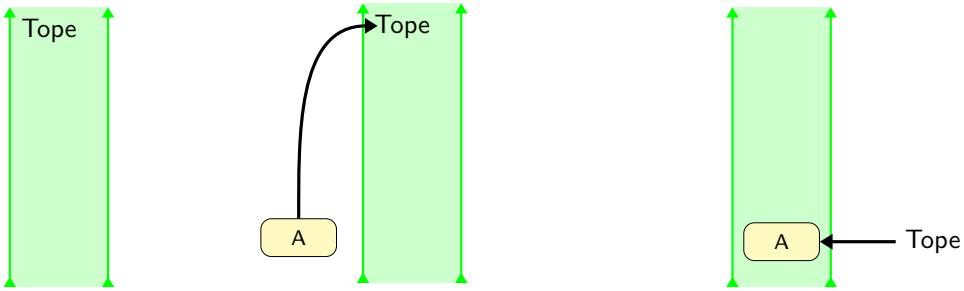
Una cosa interesante es que este conjunto mínimo también puede variar según quien implemente el tda.

3.1. Conjunto Mínimo de Operaciones

A continuación se explicaran algunas de las operaciones del conjunto mínimo para la pila.

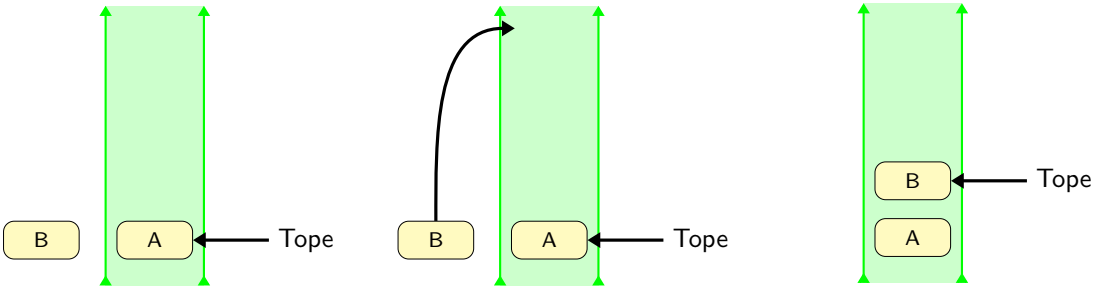
3.1.1. Poner

La operación poner o **push** es una de las dos operaciones más importantes de una pila. Esta operación 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:



Pila Vacia | Pila: Poner(A) | Pila

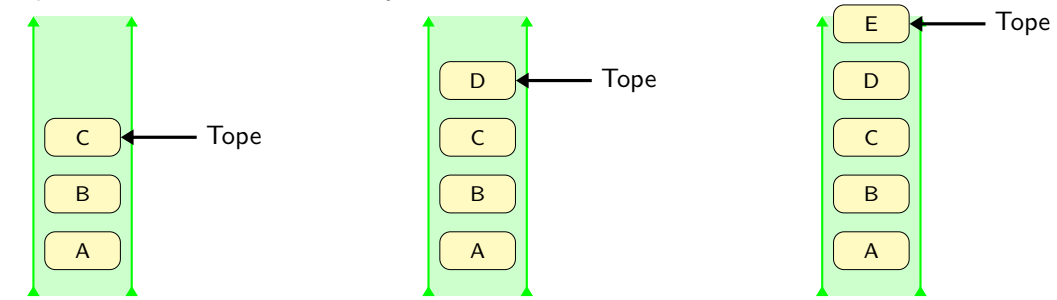
En la imagen anterior se puede ver como a una pila sin elementos se le inserta uno, por el tope, y el tope pasa a ser el elemento insertado. A continuación, se muestra un par de operaciones poner en la pila.



Pila: Poner(B) | Pila: Poner(B) | Pila: Poner(B)

Antes | Durante | Despues

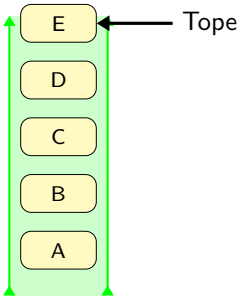
Sin más preámbulos se insertarán C, D, y E:



Pila: Poner(C) | Pila: Poner(D) | Pila: Poner(E)

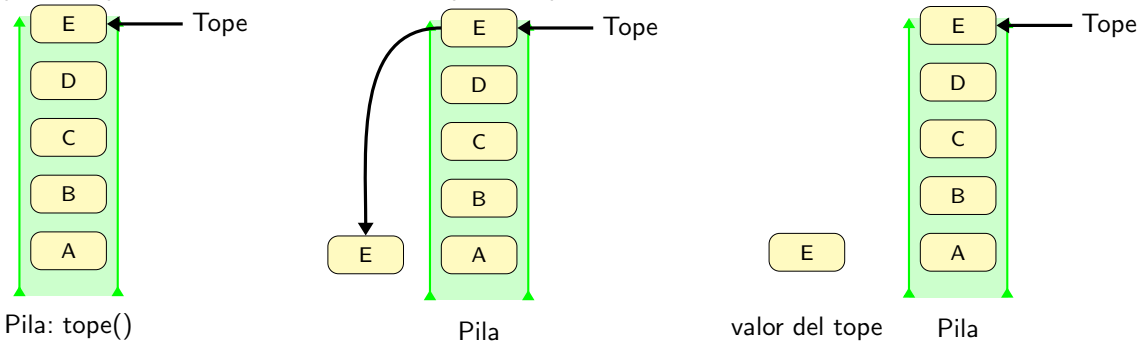
Puede verse como queda la pila desde su estado vacío= verdadero, tras haber ejecutado :

```
poner(A)
poner(B)
poner(C)
poner(D)
poner(E)
```



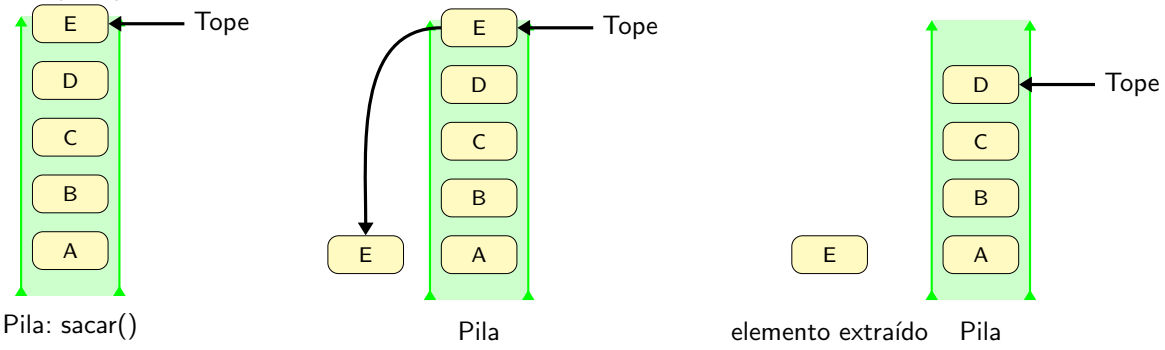
3.1.2. Tope

Esta operación permite observar el valor del tope de la pila.

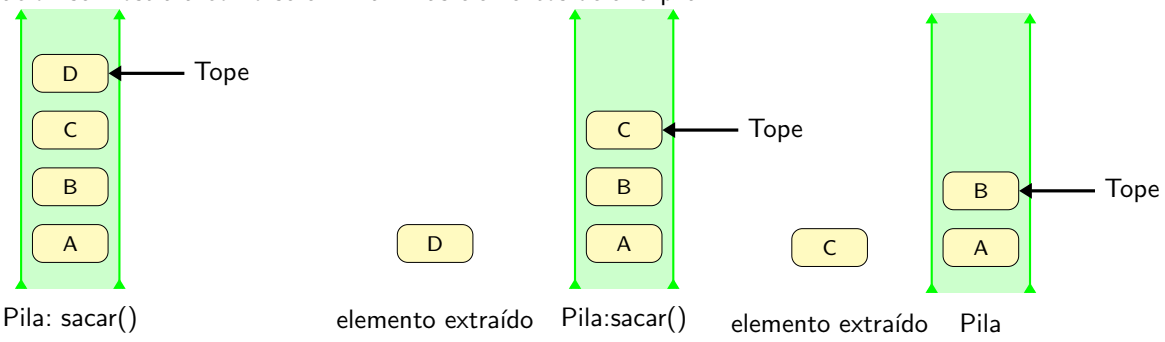


3.1.3. Sacar

La otra operación por antonomasia que se realiza sobre una pila es la operación sacar() o **pop**. Esta operación retira el elemento del tope de la pila y mueve el tope de la pila al elemento anterior al extraído, si el elemento extraído es el último la pila queda vaciá:



Notar que la operación sacar extra el elemento del tope de la pila, haciendo que esta tenga un elemento menos. A continuación se mostrará como se eliminan más elementos de una pila:



las operaciones poner permiten que la pila crezca o decrezca en la cantidad de elementos que esta posea apilados internamente.

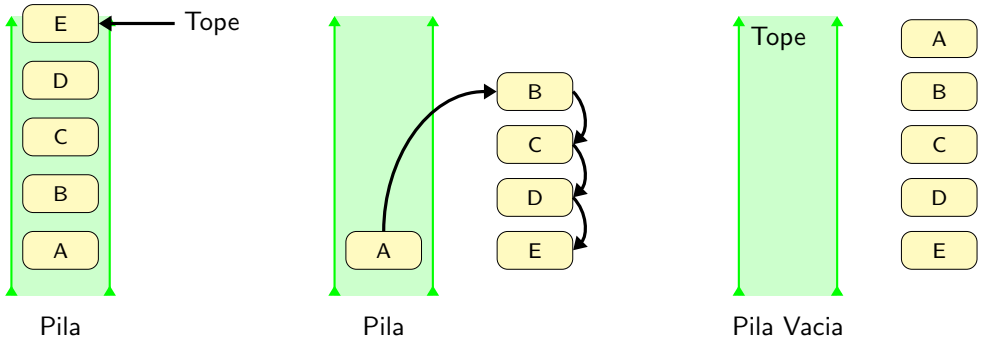
3.1.4. Esta_vacia

Esta operación permite determinar si una pila tiene o no tiene elementos. La operación **esta_vacia()** es una operación que devuelve el estado de la pila como:

- verdadero: si le pila no posee elemento apilados
- falso: si la pila posee elemento en su interior apilado

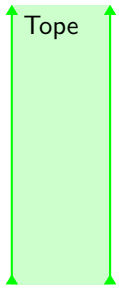
Esta operación muestra su importancia a la hora de querer vaciar una pila sin la necesidad de saber cuanto elementos está apilados en su interior. Para ello podria utilizarse el siguiente algoritmo:

```
while( !esta_vacia(pila) )
        sacar();
```



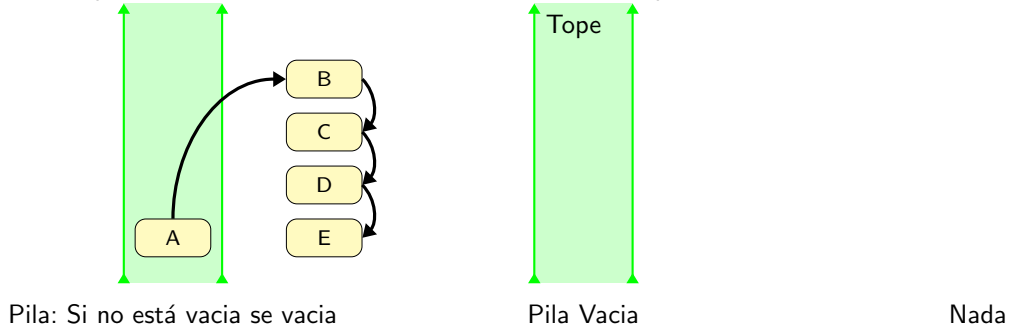
3.1.5. Crear

En la operación crear se realizan operaciones que tienen que ve con el tamaño de la pila, es decir la cantidad de elemento que la misma podrá albergar. Además se realizan las tareas de inicialización de la misma, como por ejemplo que el estado de la misma es vacía:



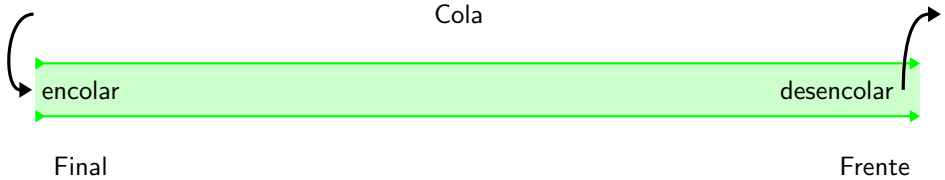
3.1.6. Destruir

La operación **destruir** se ocupa de liberar y limpiar todos los recursos que se utilizan para la creación de una pila. Si la misma posee elementos en su interior debe vaciarse en el proceso de destrucción:



4. TDA Cola

Otro tipo de dato abstracto muy utilizado ya que puede representar varias situaciones de la vida cotidiana, es el tipo de dato **Cola**. Una cola es una estructura que posee dos dos extremos por los que se realizan operaciones. Un extremo es el inicio o frente de la cola y el otro extremo es el final o rear de la cola. Este tipo de estructura sirve para modelar procesos como la cola de un colectivo, la cola de un supermercado, el despacho de combustible, entre muchos otros.



El **conjunto mínimo de operaciones** para el caso de una cola es:

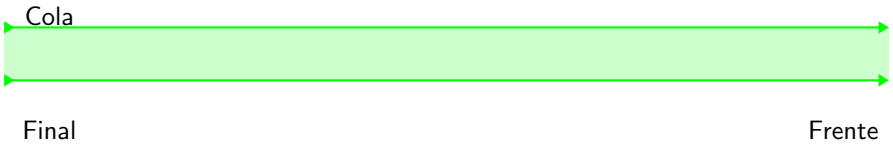
- crear (create)
- encolar (enqueue)
- desencolar (dequeue)
- primero (first)
- esta_vacia (is_empty)
- destruir (destroy)

4.0.1. Encolar

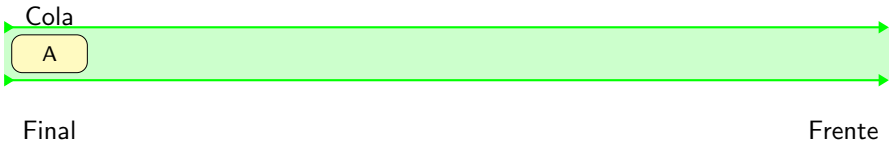
La operación **encolar** es una de las dos operaciones más destacables de este tipo de dato o estructura. la idea es que los elemento se encolan por el final de la cola. La idea es la misma que la cola de un autobús, uno llega a la parada y se coloca al final de la cola para respetar el orden de llegada de los pasajeros.

Final  Inicio

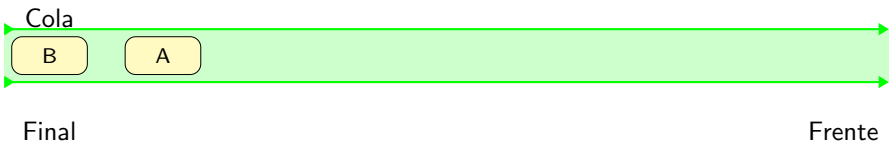
Para agilizar los ejemplos se utilizaran letras. A partir de una cola vaciá se encolaran varios elementos en la misma.



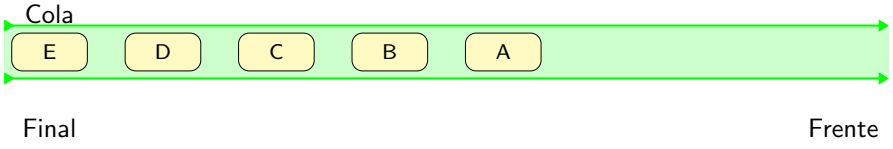

```
encolar(A)
```




```
encolar(B)
```



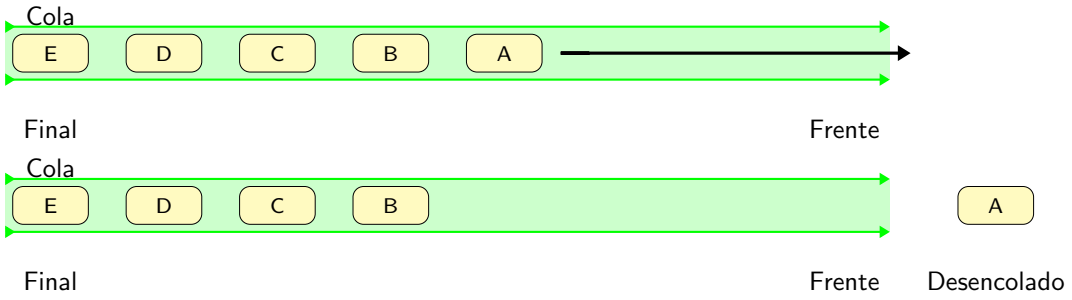

```
encolar(C)
encolar(D)
encolar(E)
```



4.0.2. Desencolar

la otra operación importante para la utilización de una cola es la operación **desencolar**. Esta operación, extrae del frente de la cola el elemento que se encuentra en él.

```
desencolar()
```

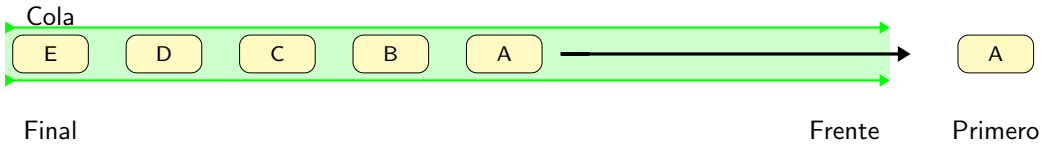


Esta operación no tiene más secretos. La orma en que opera una estructura como una cola se denomina **First In First Out** o **FIFO**. El primer elemento que entra en la cola es el primer elemento que sale de la misma, es decir, se respeta el orden de llegada de los mismos.

4.0.3. Primero

esta operación permite observar cual es el primer elemento del frente de la cola:

```
primero()
```



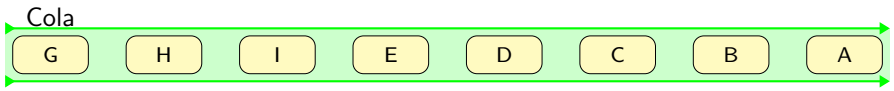
4.0.4. Esta_vacia

4.0.5. Esta_vacia

Esta operación permite determinar si una cola tiene o no tiene elementos. La operación **esta_vacia()** es una operación que devuelve el estado de la cola como:

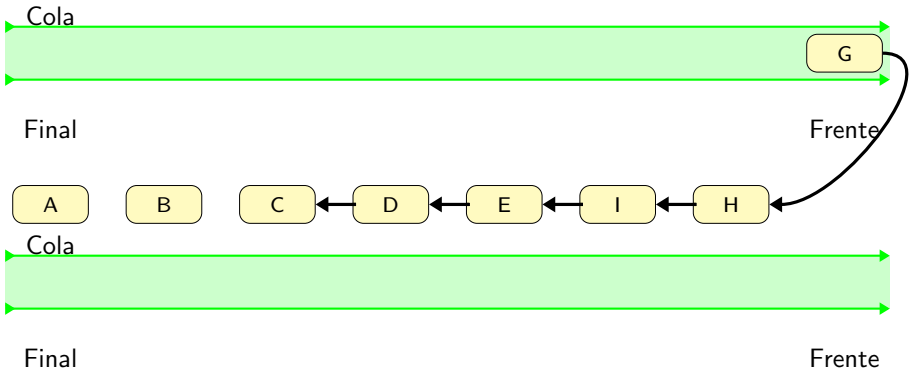
- verdadero: si le cola no posee elemento encolados
- falso: si la cola posee elementos en su interior encolados

Esta operación muestra su importancia a la hora de querer vaciar una cola (al igual que en la pila) sin la necesidad de saber cuanto elementos hay encolados en su interior. Para ello podría utilizar el siguiente algoritmo:



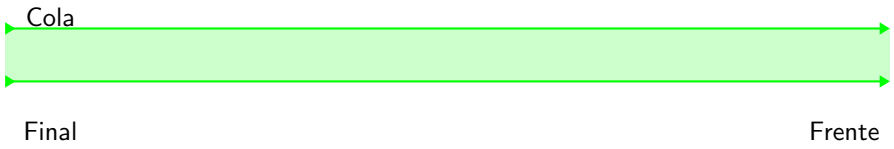

```
        while( !esta_vacia(cola) )
        sacar();

```



4.0.6. Crear

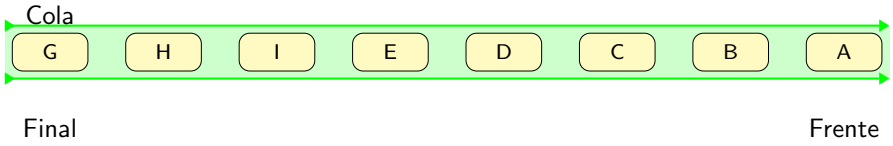
En la operación crear se realizan operaciones que tienen que ve con el tamaño de la cola, es decir la cantidad de elemento que la misma podrá albergar. Además se realizan las tareas de inicialización de la misma, como por ejemplo que el estado de la misma es vacía:



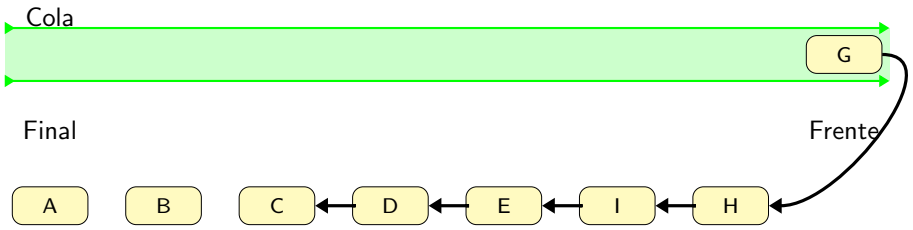
4.0.7. Destruir

La operación **destruir** se ocupa de liberar y limpiar todos los recursos que se utilizan para la creación de una cola. Si la misma posee elementos en su interior debe vaciarse en el proceso de destrucción:

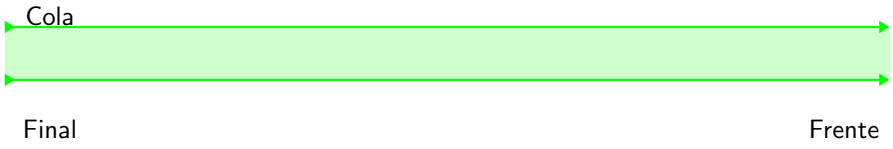
1. La cola tienen elementos



2. Se deben eliminar cada uno de los elementos de la cola, es decir vaciarla.



3. Una vez que la cola no posee ningún elemento



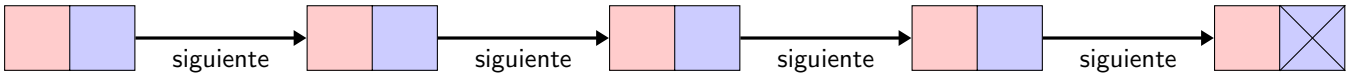
4. Se destruye liberando toda la memoria que la estructura ocupa:

   Nada

5. TDA Lista

Este tipo de dato está basado en los nodos enlazados. La idea del mismo es simple y reside en que un nodo al poder conocer su siguiente puede crear una lista de nodos, esta lista termina cuando el último nodo apunta a nada o a NULL.

Básicamente un lista compuesta por este tipo de nodos se denomina **lista enlazada**. Existen muchas variantes, según como se estructure su nodo (anterior y siguiente) y según su comportamiento (lineal o cíclica)



5.1. Lista Simplemente Enlazada

Una lista simplemente enlazada cada nodo conoce al nodo siguiente, de forma tal que es unidireccional su recorrido. Las operaciones básicas de una lista simplemente enlazada son:

- crear

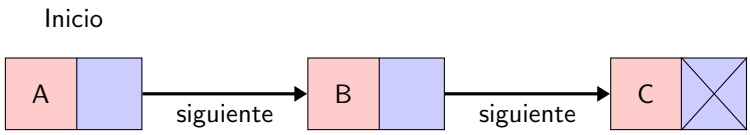
- insertar
- insertar_en_posicion
- borrar
- borrar_de_posicion
- elemento_en_posicion
- ultimo
- vacia
- elementos
- destruir

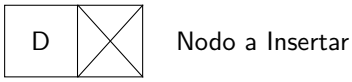
5.1.1. crear

Se debe inicializar la estructura lista_t, seteando todos sus campos en valores válidos. De realizarse su implementación con memoria dinámica se debe reservar la cantidad de memoria dinámica en el heap y posteriormente devolver la referencia a esa memoria reservada.

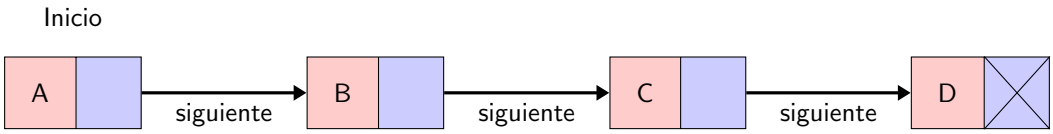
5.1.2. insertar

Esta operación inserta un elemento al final de la lista, devuelve 0 si pudo insertar o -1 si no pudo.





Una vez insertado el elemento:



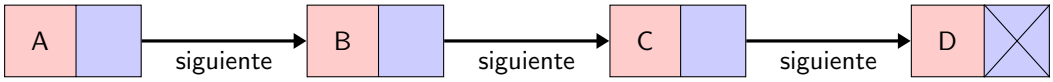
Notar que la inserción se realiza siempre al final de la lista.

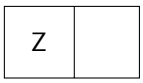
5.1.3. insertar_en_posicion

Inserta un elemento en la posición indicada, donde 0 es insertar como primer elemento y 1 es insertar luego del primer elemento y así sucesivamente. En caso de no existir la posición indicada, lo inserta al final. Devuelve 0 si pudo insertar o -1 si no pudo. hay que tener en cuenta que esta operación del tda lista es la mas compleja ya que pueden presentarse varios escenarios de inserción. Por ejemplo:

```
1 insertar_en_posicion(0,Z);
```

Inicio

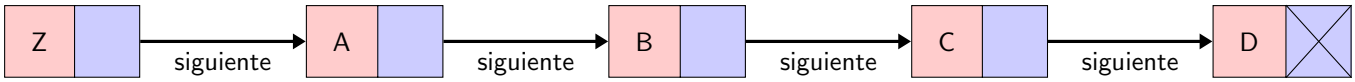




Nodo a Insertar

Una vez insertado el elemento:

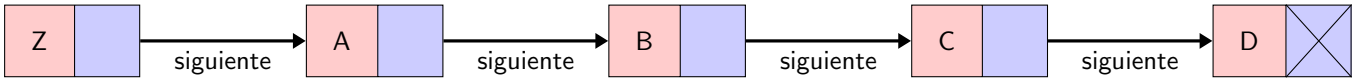
Inicio



5.1.4. borrar

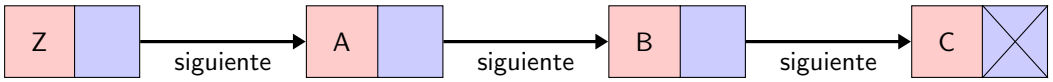
Esta operación quita de la lista el elemento que se encuentra en la última posición. Devuelve 0 si pudo eliminar o -1 si no pudo.

Inicio



Despues de ejecutarse la operación borrar():

Inicio

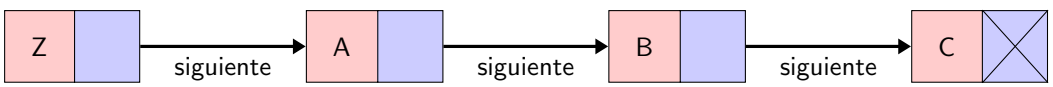


5.1.5. borrar_de_posicion

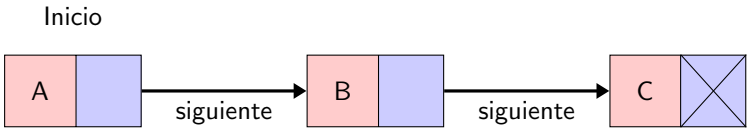
Esta operación quita de la lista el elemento que se encuentra en la posición indicada, donde 0 es el primer elemento. En caso de no existir esa posición se intentará borrar el último elemento. Devuelve 0 si pudo eliminar o -1 si no pudo.

```
1 Borrar_de_posicion(0);
```

Inicio



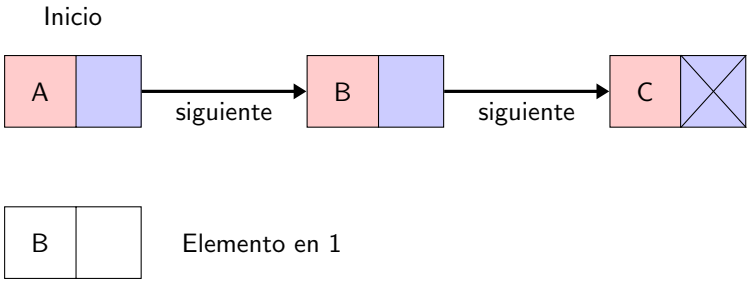
Despues de ejecutarse la operación:



5.1.6. elemento_en_posicion

Esta operación devuelve el elemento en la posición indicada, donde 0 es el primer elemento. Si no existe dicha posicion devuelve NULL.

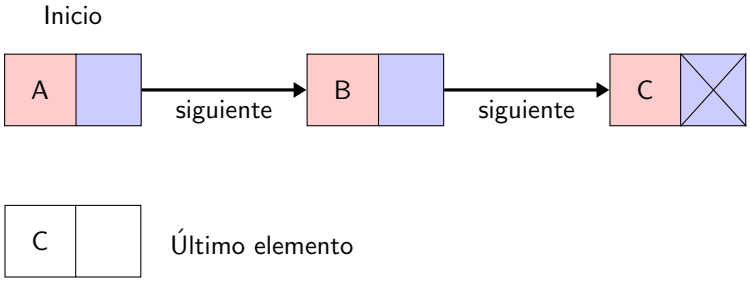
```
1 elemento_de_posicion(1);
```



5.1.7. ultimo

Devuelve el último elemento de la lista o NULL si la lista se encuentra vacía.

```
1 ultimo();
```



5.1.8. vacia

Devuelve true si la lista está vacía o false en caso contrario.

5.1.9. elementos

Devuelve la cantidad de elementos almacenados en la lista.

5.1.10. destruir

Libera la memoria reservada por la lista.

Referencias

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] Robert Kruse, CL Tondo, et al. *Data structures and program design in C*. Pearson Education India, 2007.

[3] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 199.