Vectores y Listas

Estructuras de Datos

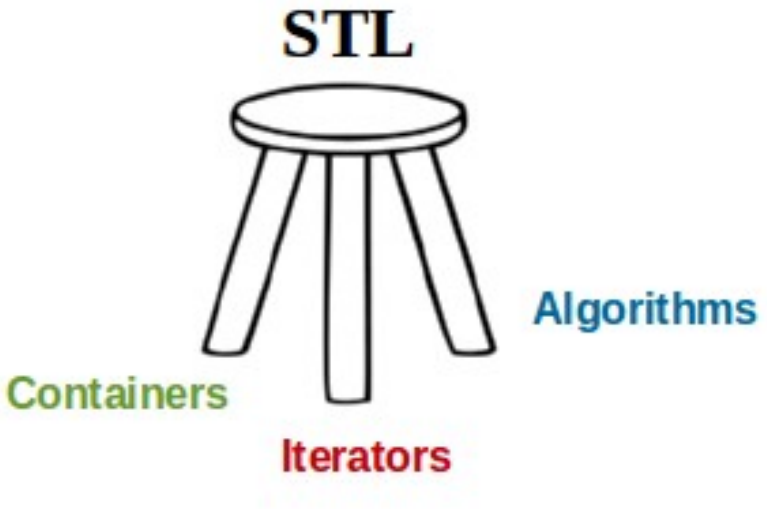
Andrea Rueda

Pontificia Universidad Javeriana
Departamento de Ingeniería de Sistemas

STL
Standard Template Library

STL (Standard Template Library)

- ¡Librería con “muchas cosas” genéricas!
- Provee un conjunto de clases comunes, usables con cualquier tipo de dato y con operaciones elementales.
- Tres componentes:
  - Contenedores (*containers*).
  - Algoritmos (*algorithms*).
  - Iteradores (*iterators*).



www.bogotobogo.com/
cplusplus/stl_vector_list.php

http://www.sgi.com/tech/stl

Contenedores STL

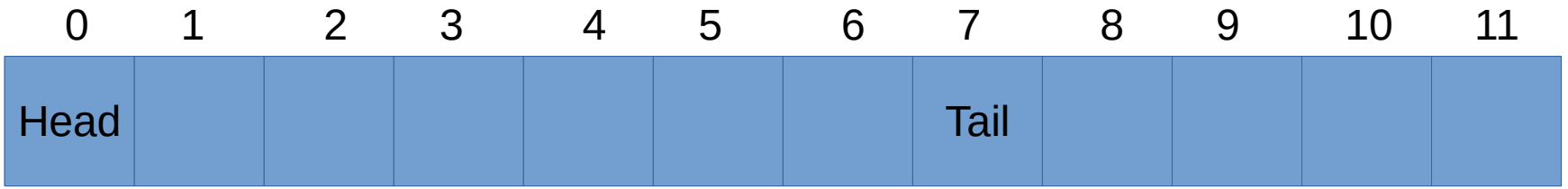
- Contenedores secuenciales estándar:
  - De acceso aleatorio:

    - **`vector`**: Arreglos dinámicos.
    (`std::vector` ↔ `#include <vector>`)

    - **`deque`**: Cola de doble cabeza.
    (`std::deque` ↔ `#include <deque>`)
  - De acceso iterativo:

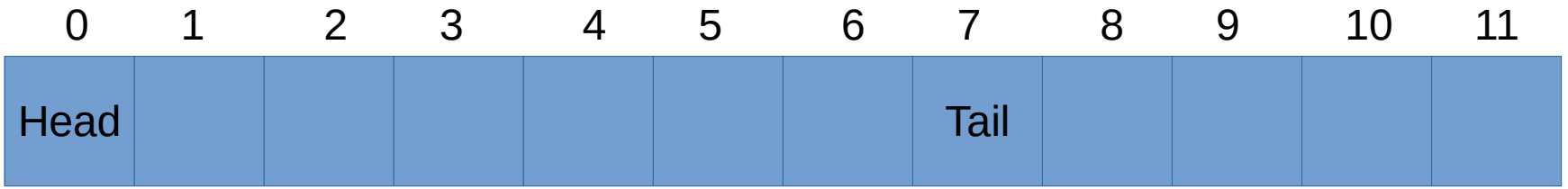
    - **`list`**: Doblemente encadenada.
    (`std::list` ↔ `#include <list>`)

vector<T>



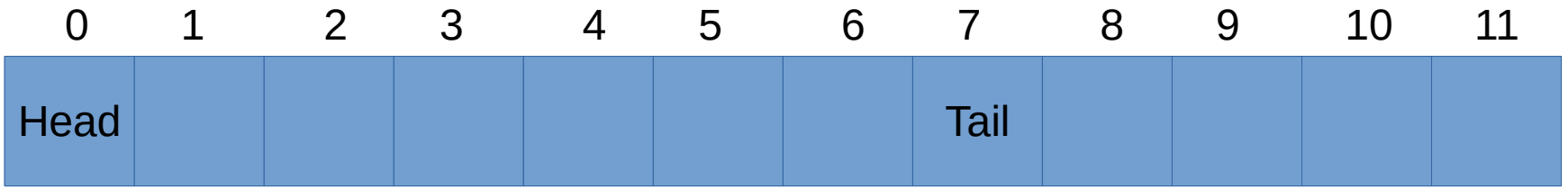
- Representa arreglos que cambian de tamaño (dinámicos)
- Posiciones contiguas de memoria
- Elementos de un mismo tipo de dato (plantilla)
- Crece en memoria por el final (cola, extremo derecho)

vector<T>



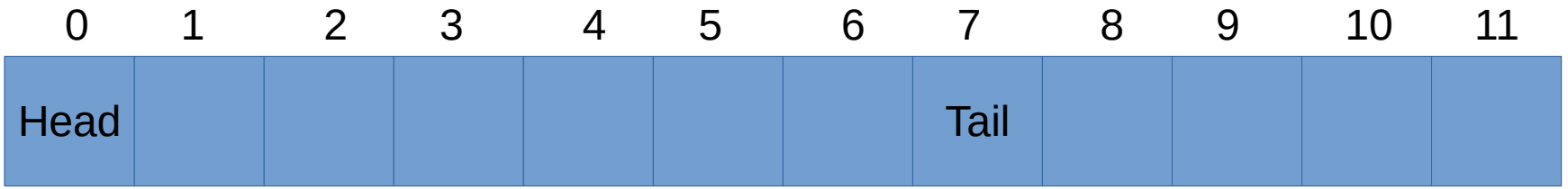
- Métodos soportados:
  - `size`: tamaño
  - `front`: elemento al frente
  - `clear`: vaciar vector
  - `pop_back`: eliminar en cola
  - `pop_front`: eliminar al frente
  - `insert`: insertar en posición
  - `empty`: vector está vacío?
  - `back`: elemento al final
  - `push_back`: insertar en cola
  - `push_front`: insertar al frente
  - `erase`: eliminar en posición

vector<T>



- ¿Orden de complejidad de los métodos?
  - `size`, `empty`.
  - `front`, `back`.
  - `clear`.
  - `push_back`, `pop_back`.
  - `push_front`, `pop_front`.
  - `insert`, `erase`.

vector<T>



- ¿Orden de complejidad de los métodos?
  - `size`, `empty`. → **O(1)**
  - `front`, `back`. → **O(1)**
  - `clear`. → **O(1)**
  - `push_back`, `pop_back`. → **O(1)**
  - `push_front`, `pop_front`. → **O(n)**
  - `insert`, `erase`. → **O(n)**

vector<T>

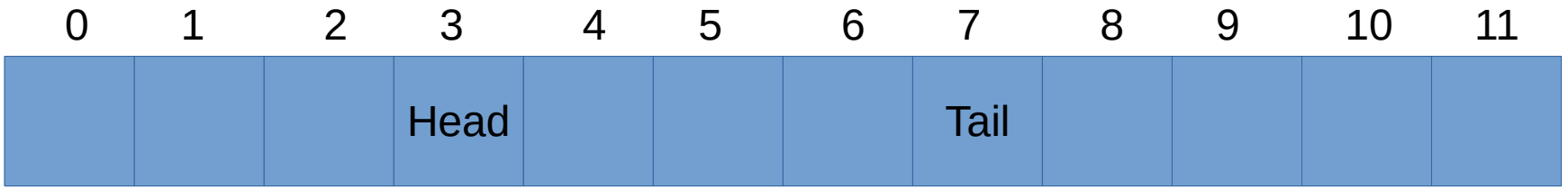
- Declaración
```
std::vector<int> miVec;
```

- Inserción de datos
```
miVec.push_back(1);
miVec.push_front(2);
```

- Acceso a datos
```
for (int i = 0; i < miVec.size(); i++)
   std::cout << miVec[i] << std::endl;
```

deque<T>

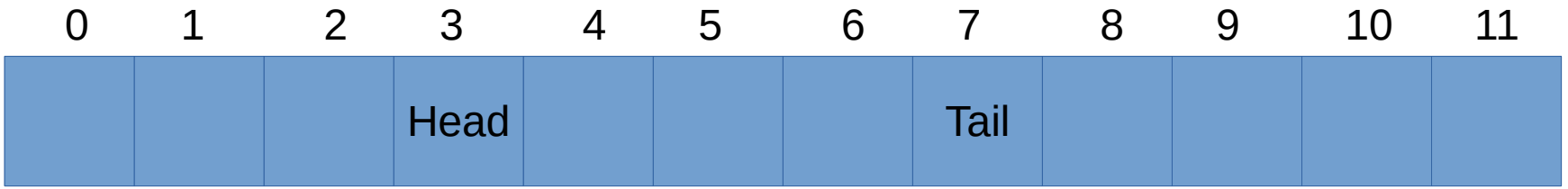
double-ended queue



- Representa arreglos que cambian de tamaño (dinámicos) en ambos extremos
- Posiciones contiguas de memoria (a trozos)
- Elementos de un mismo tipo de dato (plantilla)
- Crece en memoria por el principio y por el final (cabeza y cola, ambos extremos)

deque<T>

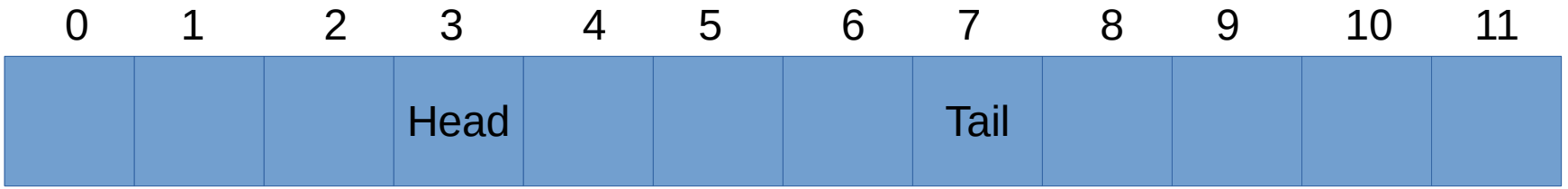
double-ended queue



- ¿Orden de complejidad de los métodos?
  - `size`, `empty`.
  - `front`, `back`.
  - `clear`.
  - `push_back`, `pop_back`.
  - `push_front`, `pop_front`.
  - `insert`, `erase`.

deque<T>

double-ended queue



- ¿Orden de complejidad de los métodos?
  - `size`, `empty`. → **O(1)**
  - `front`, `back`. → **O(1)**
  - `clear`. → **O(1)**
  - `push_back`, `pop_back`. → **O(1)**
  - `push_front`, `pop_front`. → **O(1)**
  - `insert`, `erase`. → **O(n)**

deque<T>

- Declaración
```
std::deque<int> miDeq;
```

- Inserción de datos
```
miDeq.push_back(1);
miDeq.push_front(2);
```

- Acceso a datos
```
for (int i = 0; i < miDeq.size(); i++)
   std::cout << miDeq[i] << std::endl;
```

Iteradores STL

- Objeto que puede recorrer un rango de elementos predefinido a través de ciertos operadores.

Iteradores STL

- Objeto que puede iterar entre un rango de elementos predefinido a través de operadores:
  - *Input iterator* : extrae datos, movimiento hacia adelante.
  - *Output iterator* : almacena datos, movimiento hacia adelante.
  - *Forward iterator* : almacena y extrae datos, movimiento hacia adelante.
  - *Bidirectional iterator* : almacena y extrae datos, movimiento hacia adelante y hacia atrás.
  - *Random-access iterator* : almacena y extrae datos, acceso a elementos en cualquier orden.

Iteradores STL

- Operaciones:

  - Operador *: (dereferenciación) funciona como un apuntador, retorna el contenido del iterador.

  - Operadores ++ y --: mueve el iterador a la siguiente posición o a la anterior posición.

  - Operadores == y !=: comparación de iteradores, si apuntan o no al mismo elemento.

  - Operador =: asigna una nueva posición al iterador (usualmente principio o fin del contenedor).

Iteradores STL

	Operaciones					
	*	* =	++	--	== , !=	=
Input iterator	✔	✘	✔	✘	✔	✔
Output iterator	✘	✔	✔	✘	✘	✔
Forward iterator	✔	✔	✔	✘	✔	✔
Bidirectional iterator	✔	✔	✔	✔	✔	✔
Random-access iterator *	✔	✔	✔	✔	✔	✔

* además soporta operaciones de acceso aleatorio: +n, -n, <, >, <=, >=, +=, -=, []

Contenedores e Iteradores STL

- Cada contenedor incluye funciones básicas para usar con el operador =
  - **`begin():`** iterador que representa el inicio de los elementos.
  - **`end():`** iterador que representa el elemento después del final.
  - **`rbegin():`** representa el inicio en la secuencia inversa.
  - **`rend():`** representa el elemento después del final en la secuencia inversa.

Contenedores e Iteradores STL

- Cada contenedor tiene varios tipos de iteradores:
  - ***`container`*`::iterator`**
    iterador de lectura/escritura (entrada/salida).
  - ***`container`*`::reverse_iterator`**
    iterador en secuencia inversa de lectura/escritura (entrada/salida).

Contenedores e Iteradores STL

- `vector` con iteradores:

```
std::vector<int> miVec;

for (int i = 0; i < 10; i++)
   miVec.push_back(i+1);

std::vector<int>::iterator miIt;

for (miIt = miVec.begin();
     miIt != miVec.end(); miIt++)
   std::cout << *miIt << std::endl;
```

Contenedores e Iteradores STL

- `deque` con iteradores:

```
std::deque<int> miDeq;

for (int i = 0; i < 5; i++)
   miDeq.push_front(i+1);
   miDeq.push_back(i+1);

std::deque<int>::iterator miIt;

for (miIt = miDeq.begin();
     miIt != miDeq.end(); miIt++)
   std::cout << *miIt << std::endl;
```

Iteradores inválidos

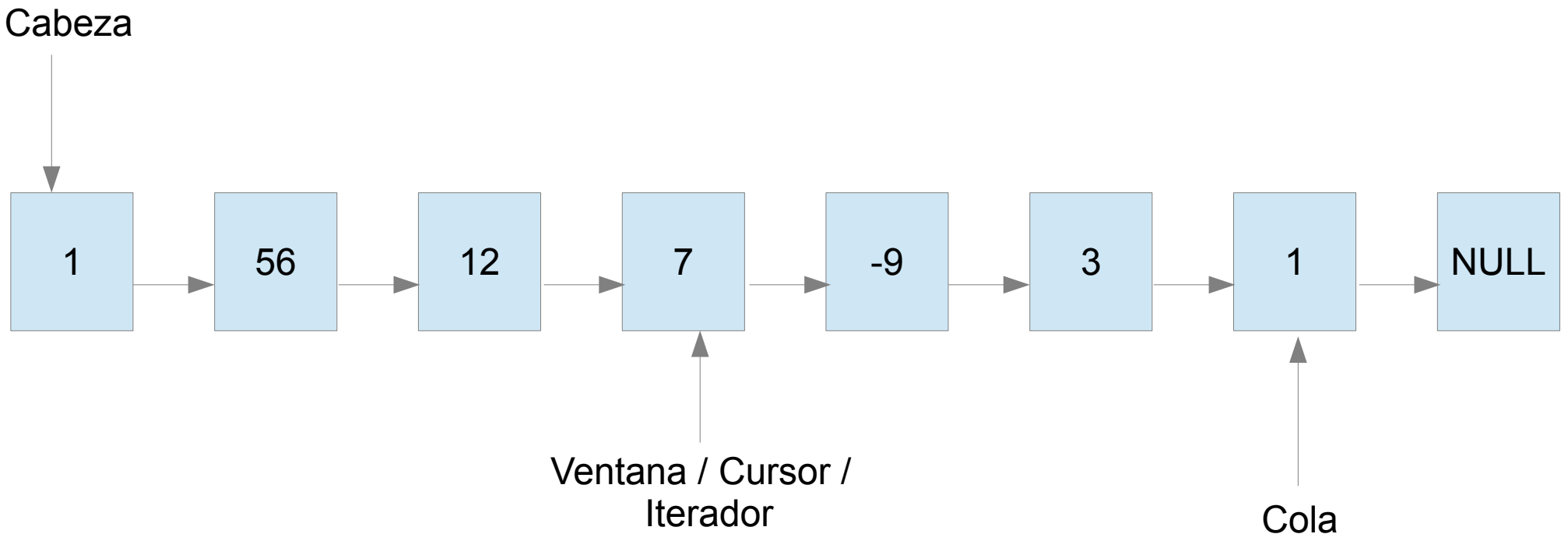
- Después de algunas operaciones en los contenedores, los iteradores pueden resultar inválidos:
  * Cambiar el tamaño o la capacidad.
  * Algunas inserciones y/o eliminaciones:
  - Iteradores singulares: sin asociar a un contenedor.
  - Iteradores después del final.
  - Iteradores fuera de rango.
  - Iterador colgante: apunta a un elemento no existente, en otra ubicación o no accesible.
  - Iteradores inconsistentes.

TAD Lista

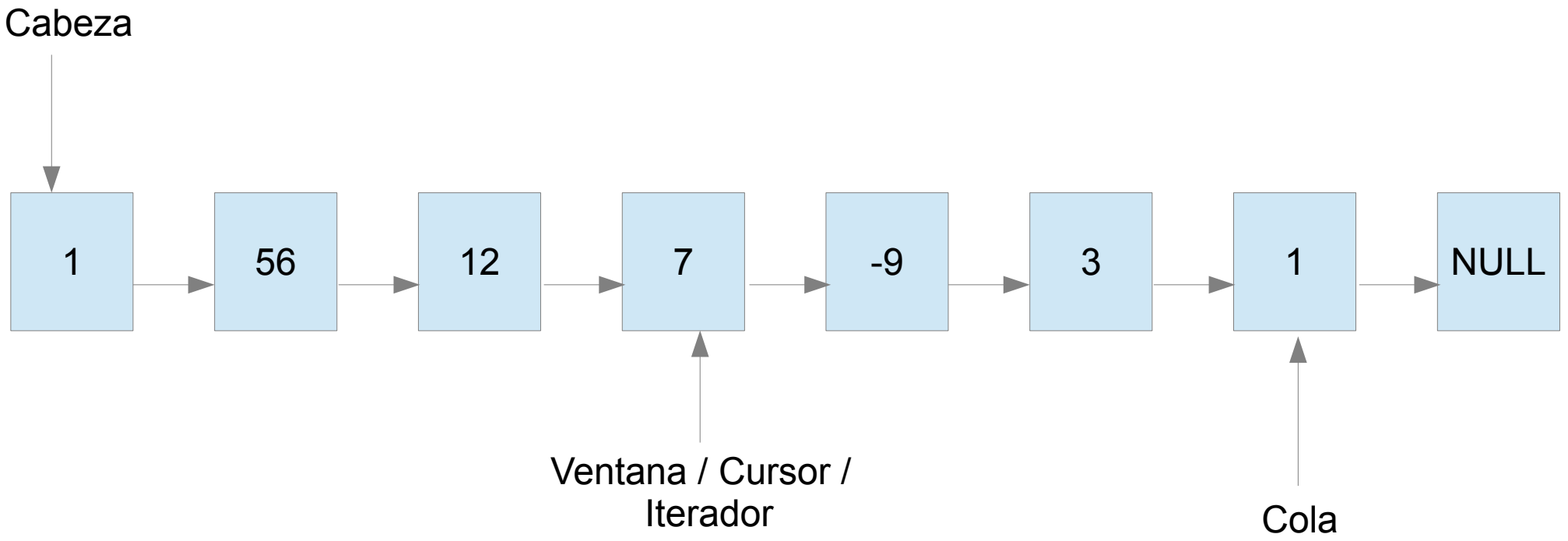
TAD Lista

- Secuencia finita de datos:
  - Primero, último.
  - Siguiente, anterior.
- Recorrido:
  - Primero → Último.
  - Último → Primero.
- Acceso aleatorio:
  - Restringido.
- Algoritmos:
  - Vacía, 1 elemento.
  - Cabeza, cola.
  - Intermedio.

TAD Lista



TAD Lista



¿Estado?

¿Interfaz?

TAD Lista

- Estado:
  - Cabeza.
  - Cola.
  - $\{E_i\}$ (¿importa el tipo?).

- Interfaz:
  - Creadoras.
  - Analizadoras: tamaño, acceso, cabeza, cola.
  - Modificadoras: insertar, eliminar.

TAD Lista

TAD Lista

Conjunto mínimo de datos:

- ...
- ...

Comportamiento (operaciones) del objeto:

- *...*
- ...
- ...
- ...
- ...
- ...

TAD Lista

TAD Lista

Conjunto mínimo de datos:

- cabeza, tipo plantilla, representa el punto de inicio.
- cola, tipo plantilla, representa el punto de finalización.

Comportamiento (operaciones) del objeto:

- esVacia(), indica si la lista está vacía.
- tamaño(), cantidad de elementos en la lista.
- cabeza(), retorna el elemento en la cabeza.
- cola(), retorna el elemento en la cola.
- insertarCabeza(v), inserta v en la cabeza.
- insertarCola(v), inserta v en la cola.

TAD Lista

TAD Lista

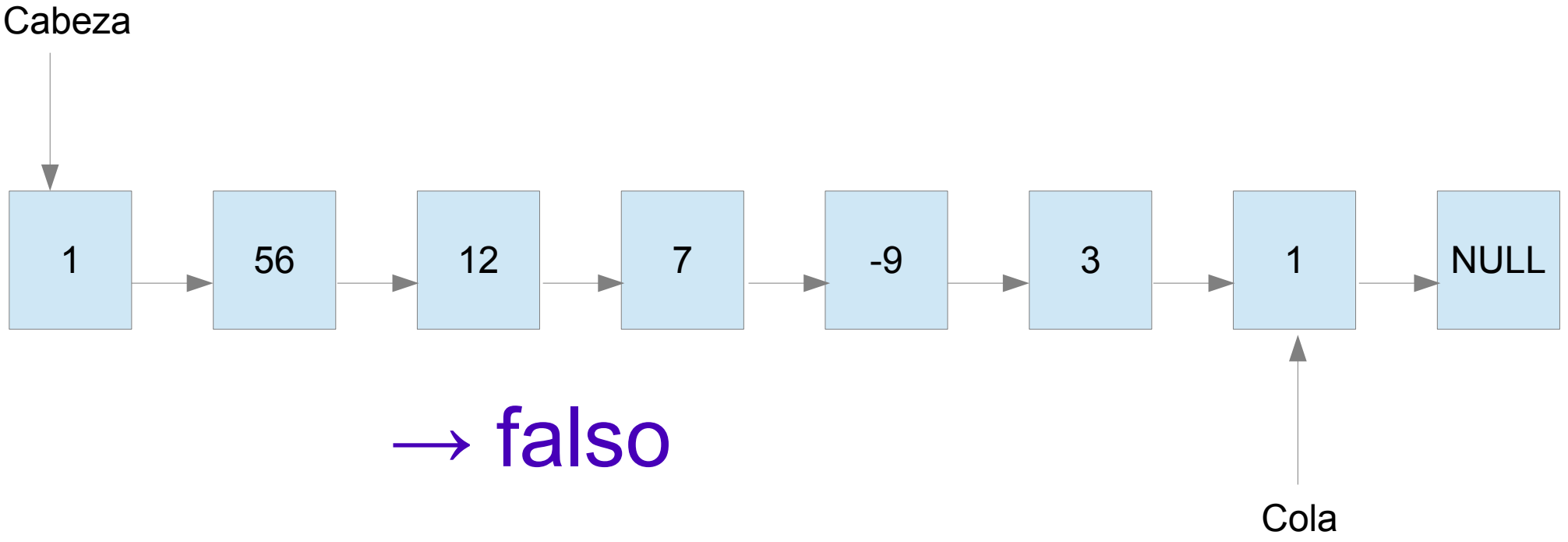
Conjunto mínimo de datos:

- cabeza, tipo plantilla, representa el punto de inicio.
- cola, tipo plantilla, representa el punto de finalización.

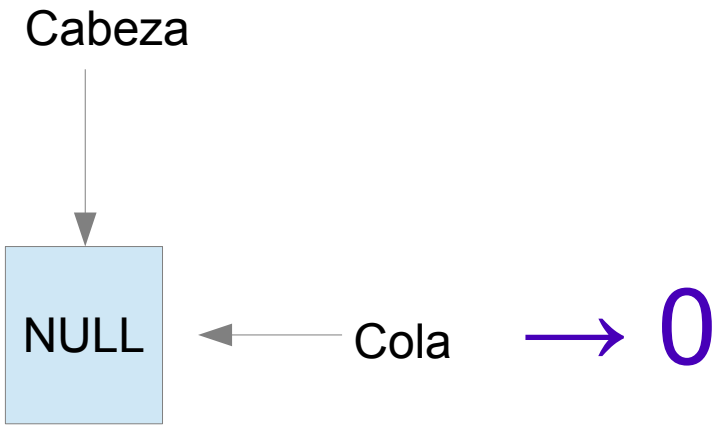
Comportamiento (operaciones) del objeto:

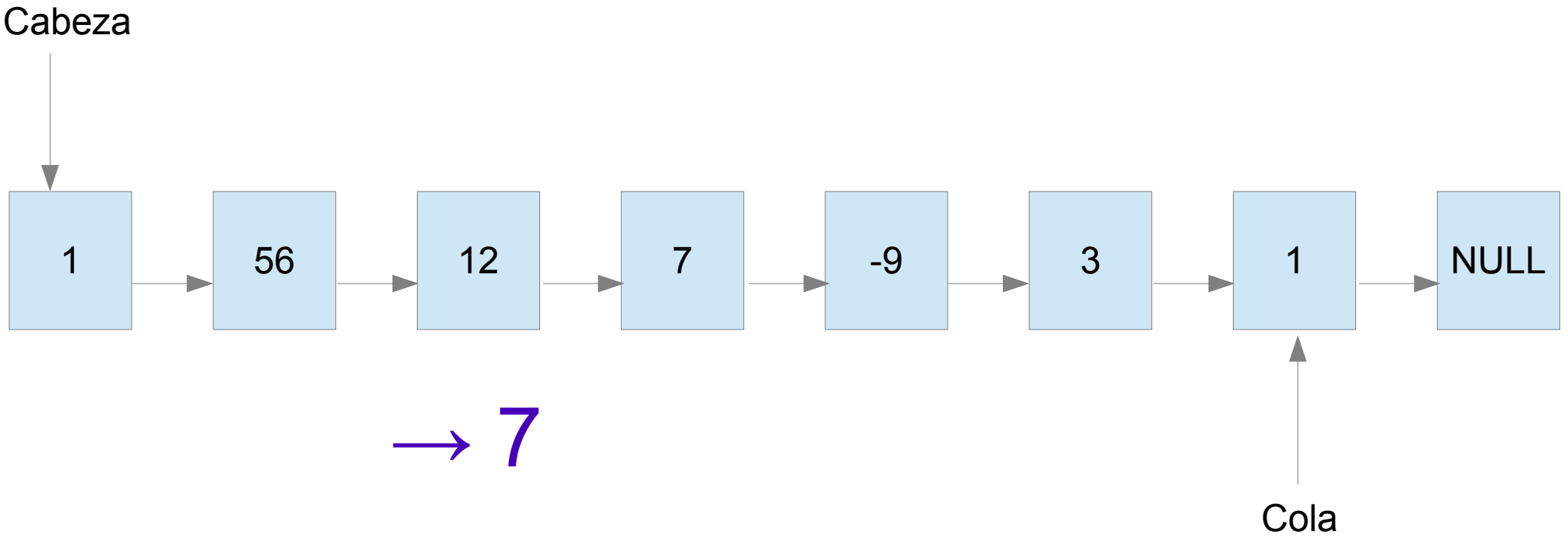
- eliminarCabeza(), elimina elemento de la cabeza.
- eliminarCola(), elimina elemento de la cola.
- insertar(pos,v), inserta v en la posición pos.
- eliminar(pos), elimina el elemento ubicado en pos.
- vaciar(), elimina todos los elementos de la lista.

TAD Lista: esVacia()

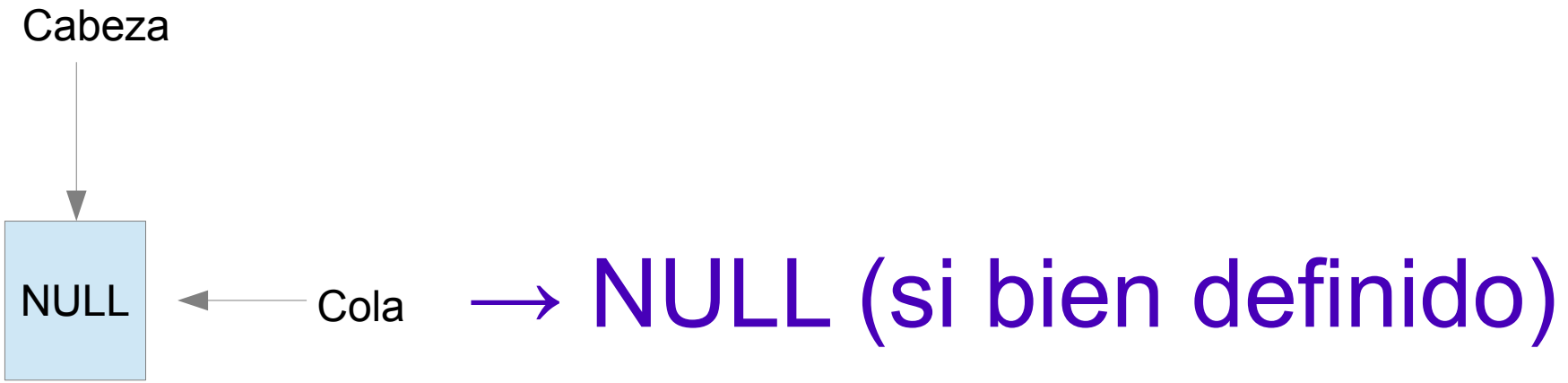


TAD Lista: tamaño()

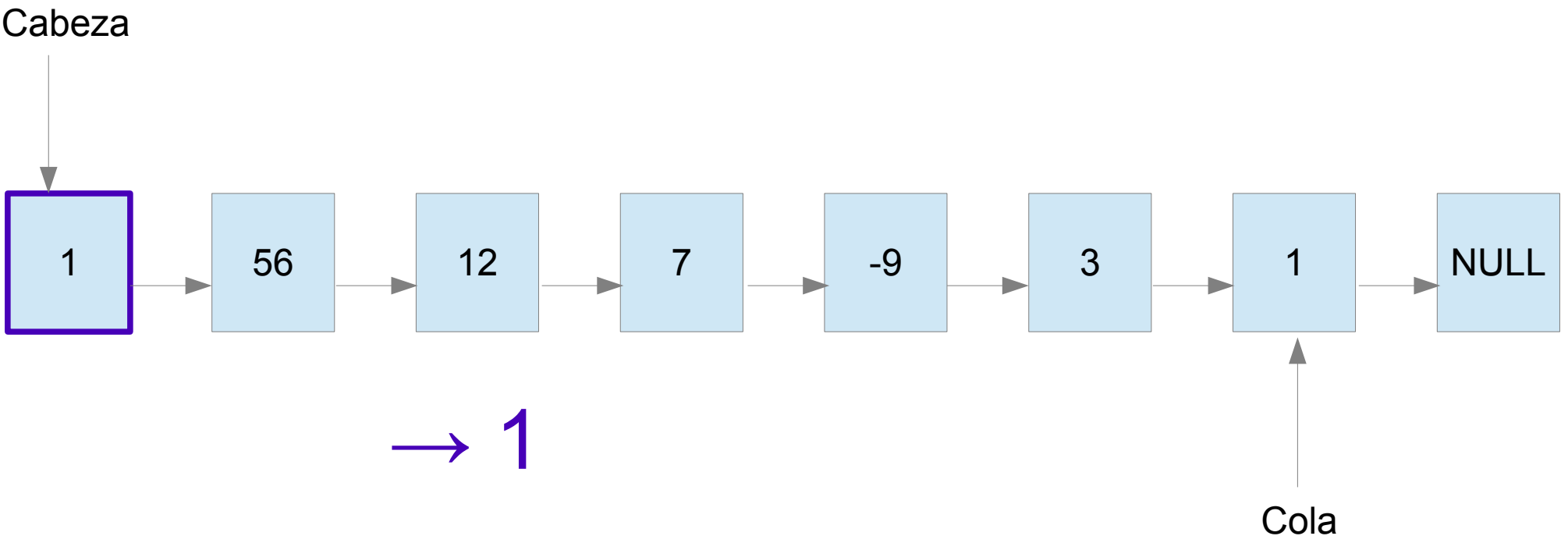




TAD Lista: cabeza()

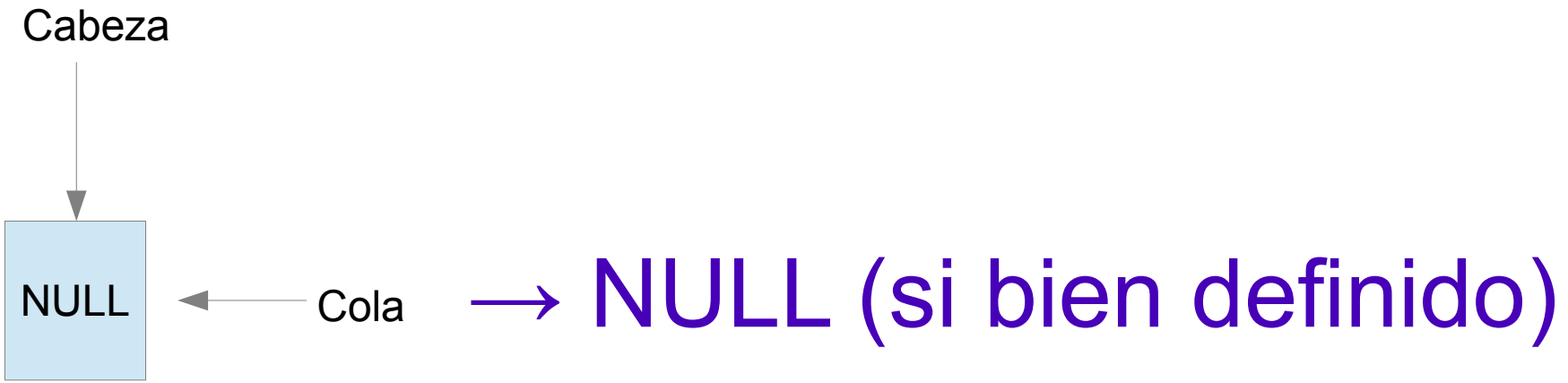


→ NULL (si bien definido)

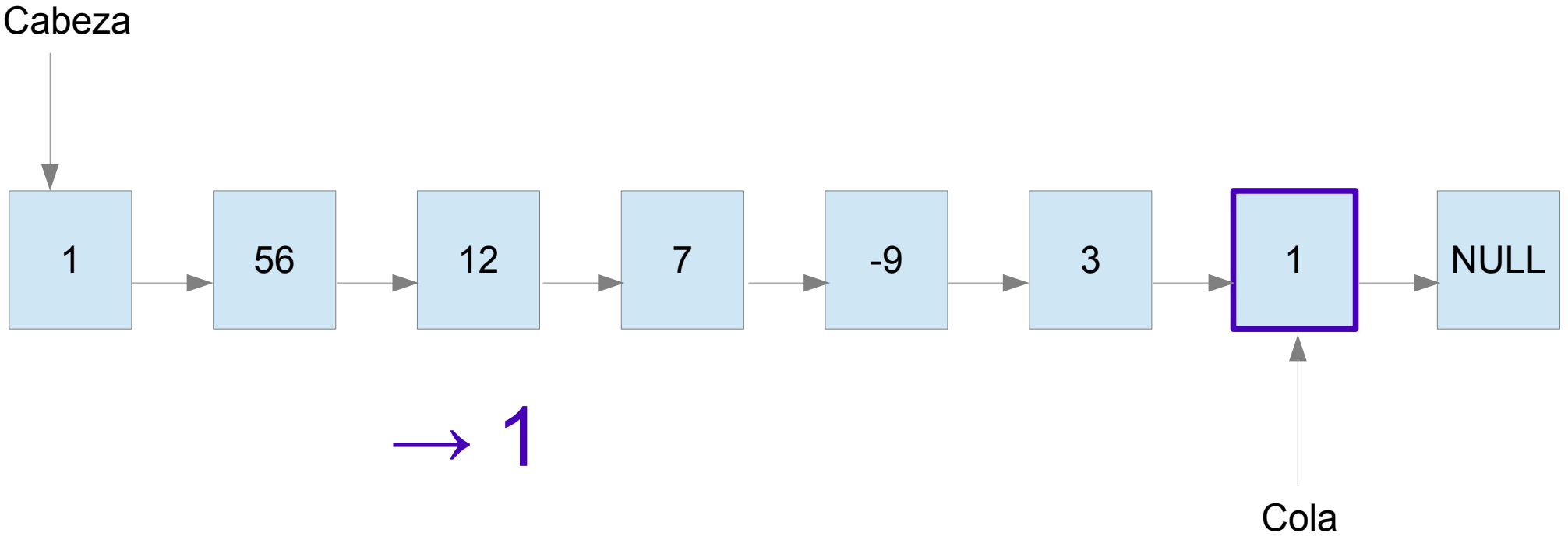


→ 1

TAD Lista: cola()

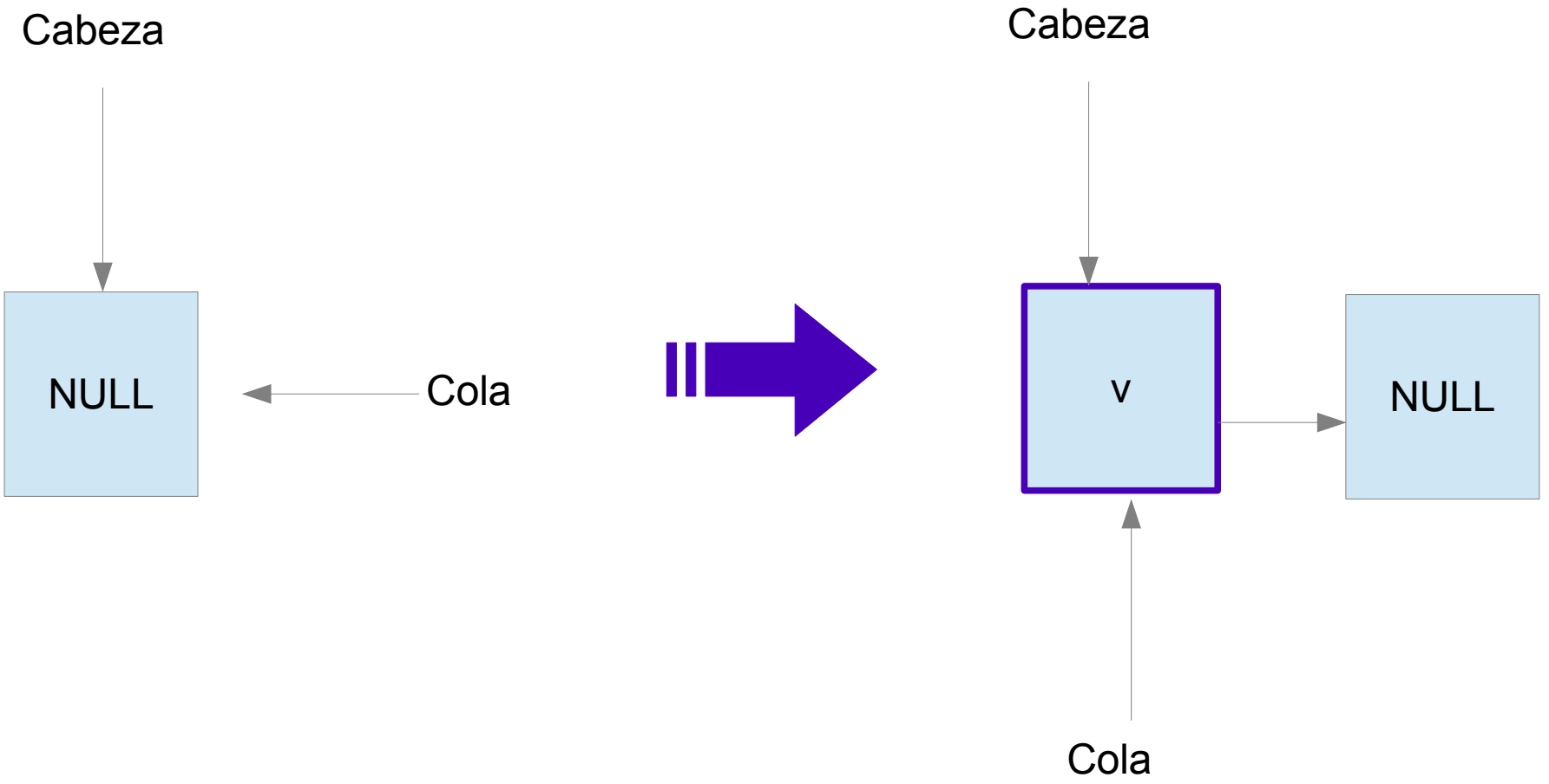


→ NULL (si bien definido)

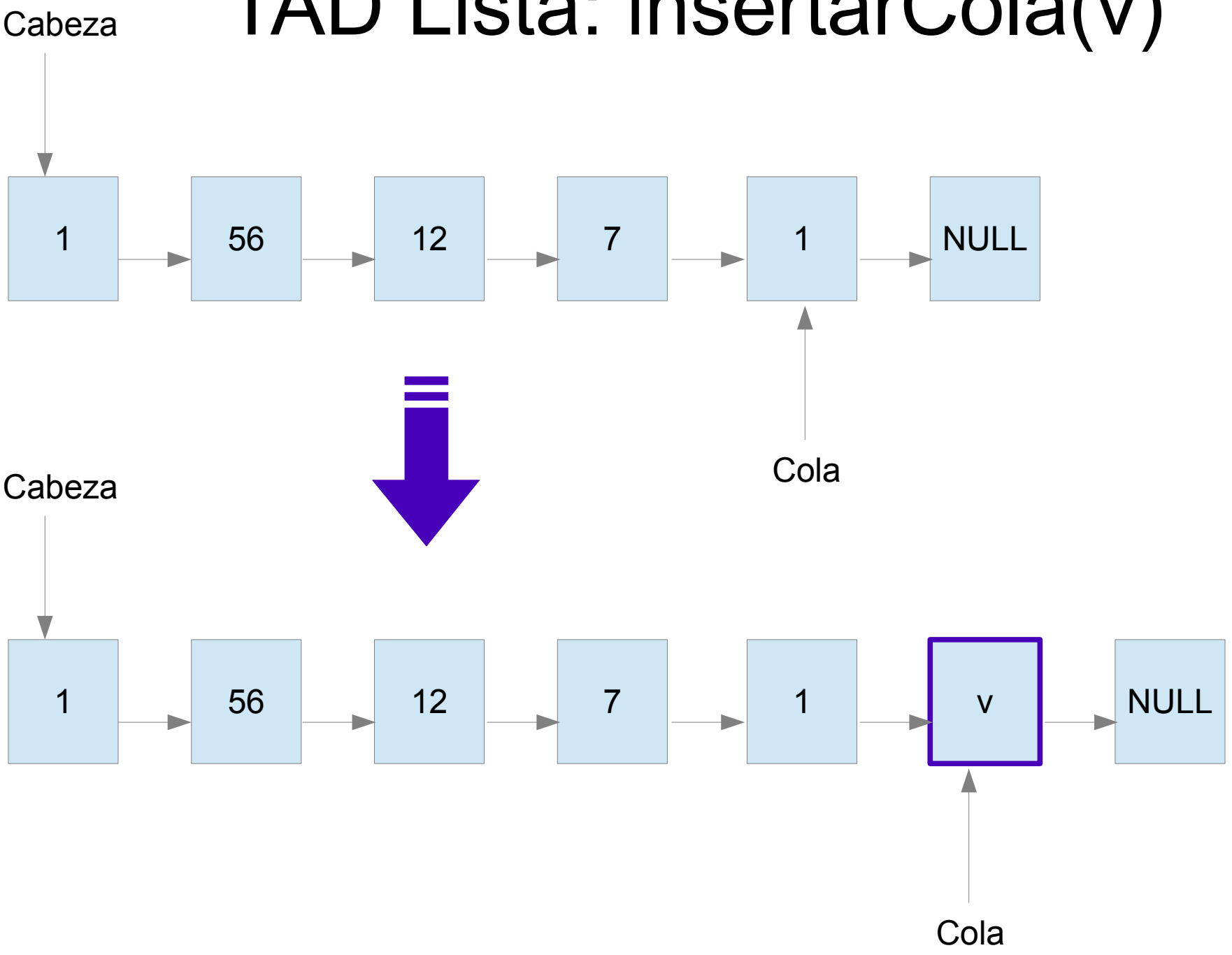


→ 1

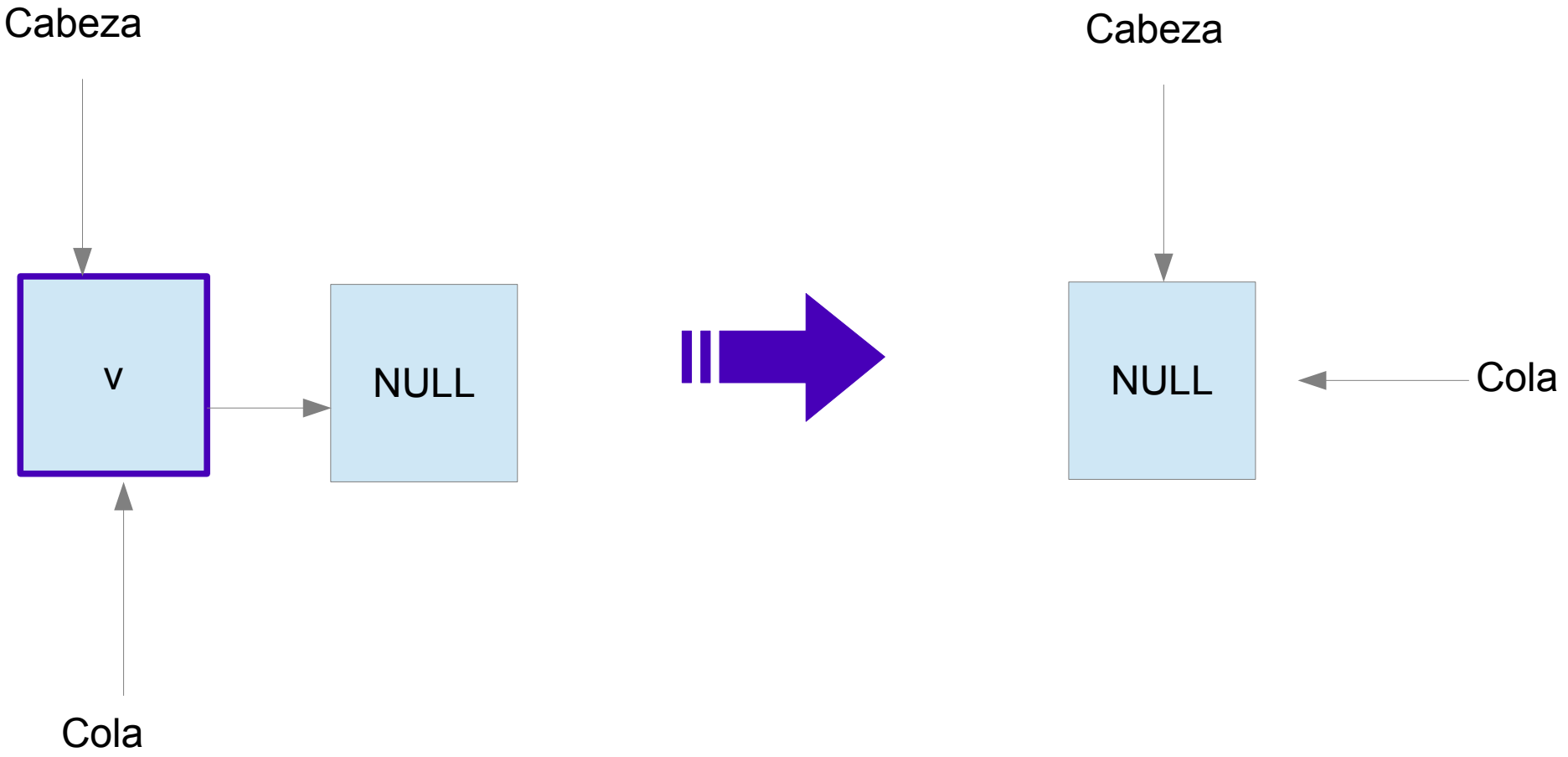
TAD Lista: insertarCola(v)



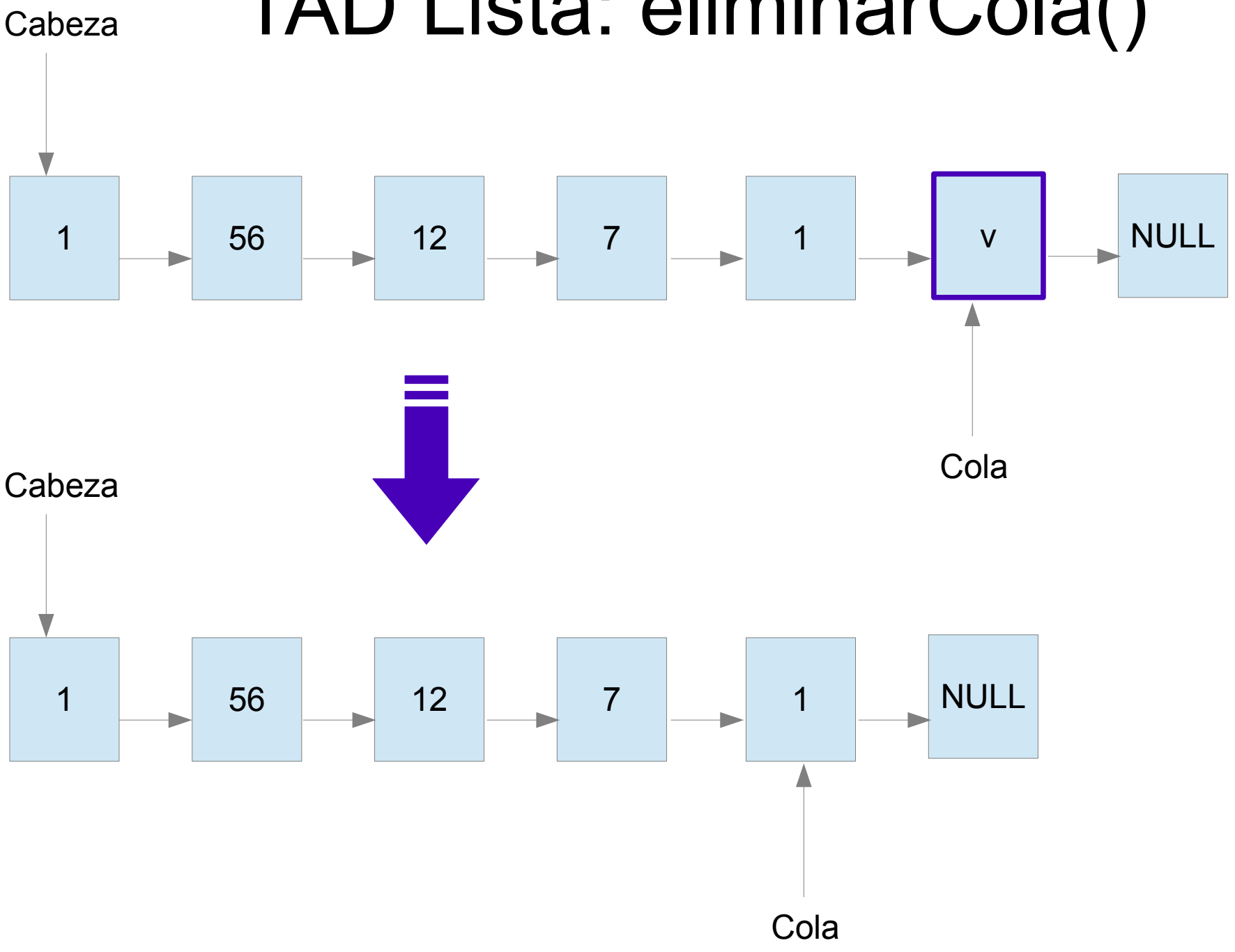
TAD Lista: insertarCola(v)



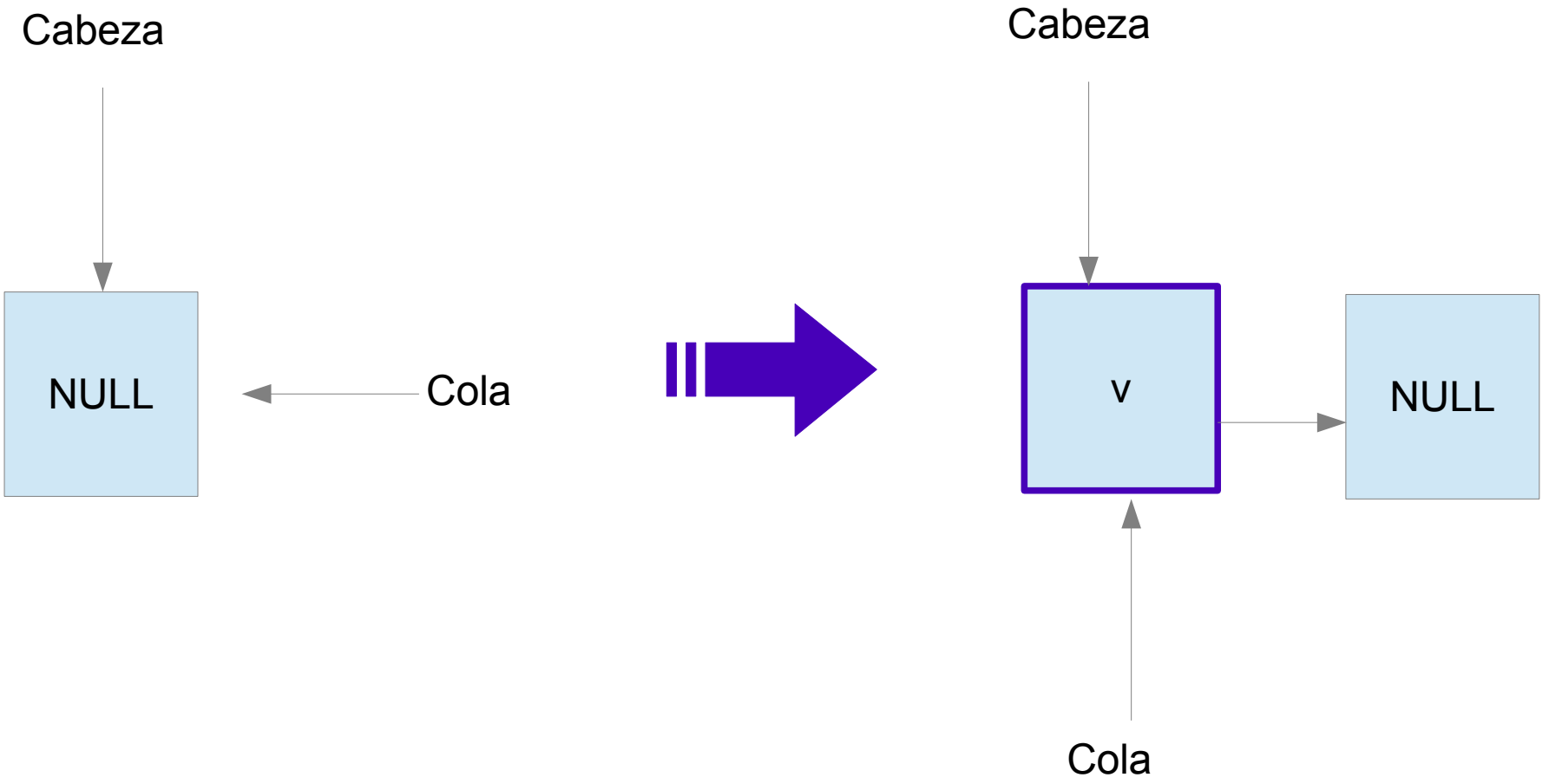
TAD Lista: eliminarCola()



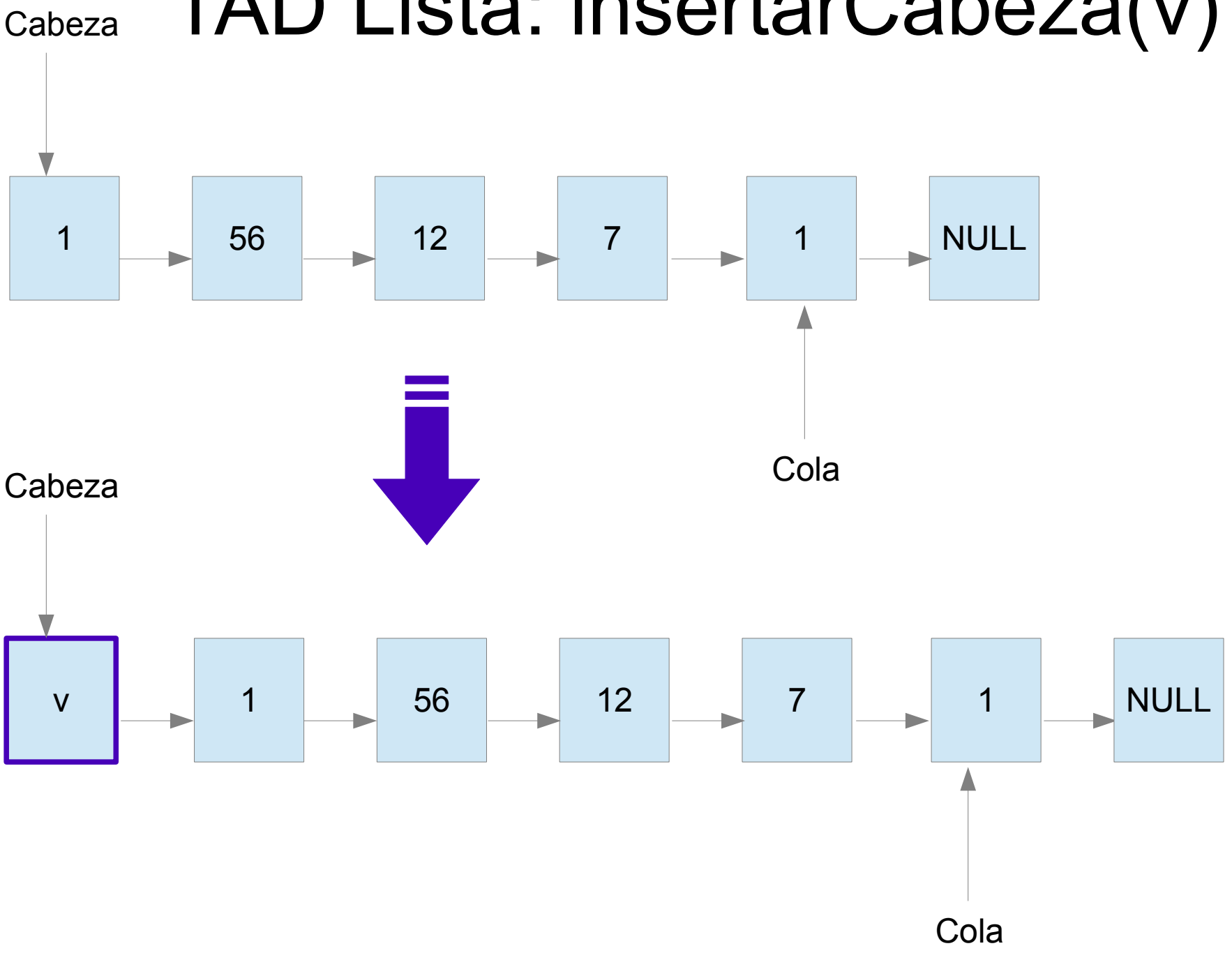
TAD Lista: eliminarCola()



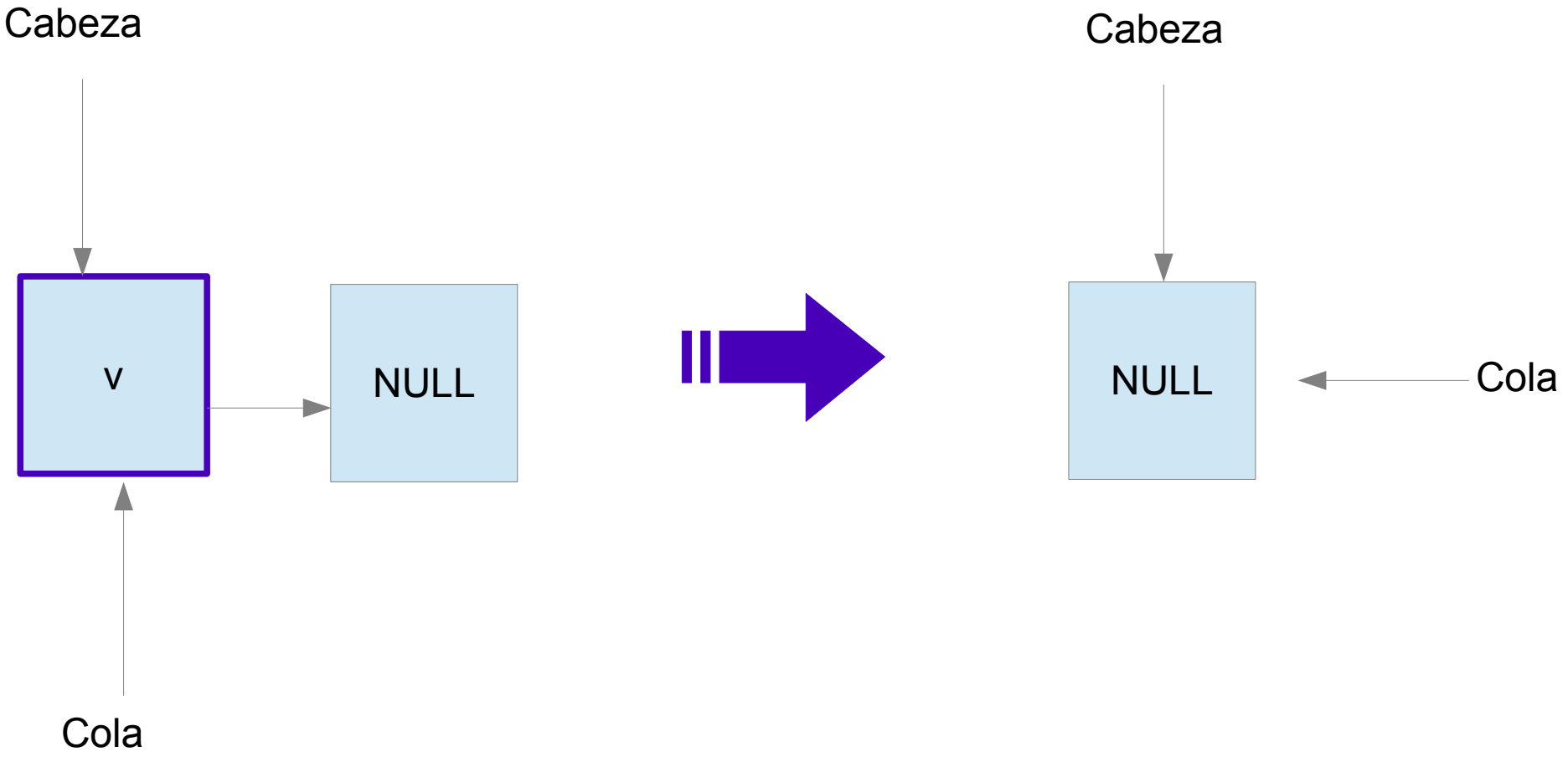
TAD Lista: insertarCabeza(v)



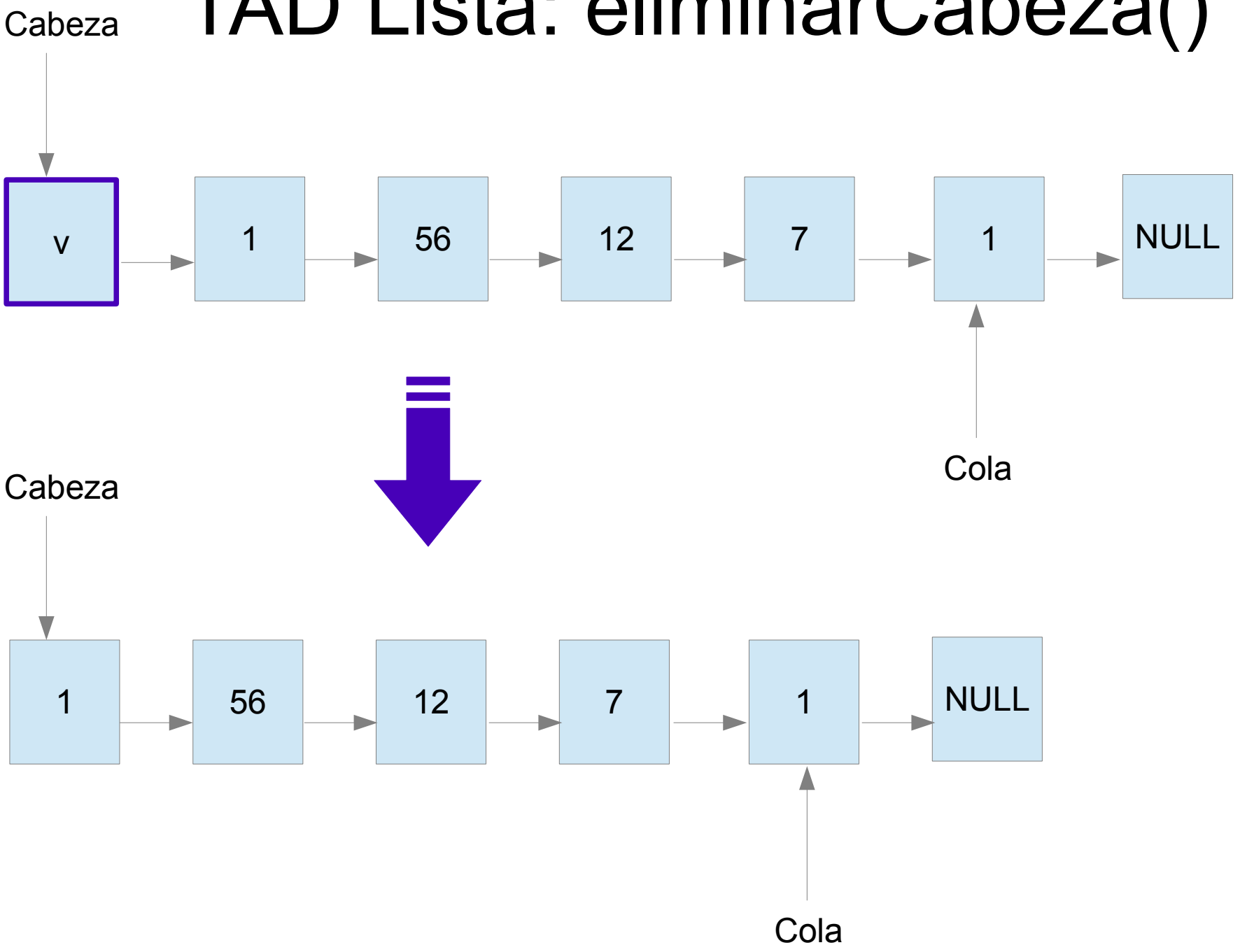
TAD Lista: insertarCabeza(v)



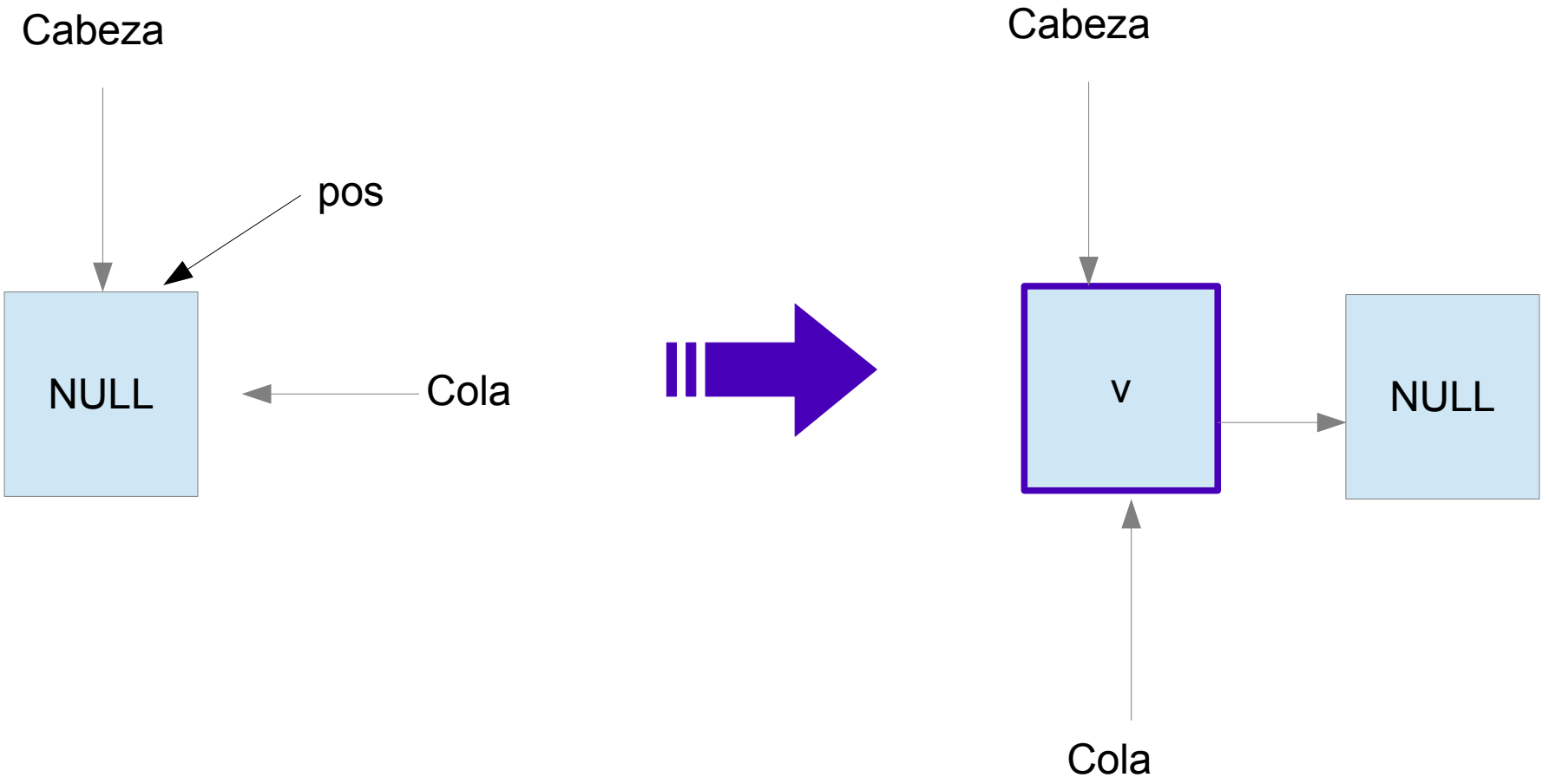
TAD Lista: eliminarCabeza()



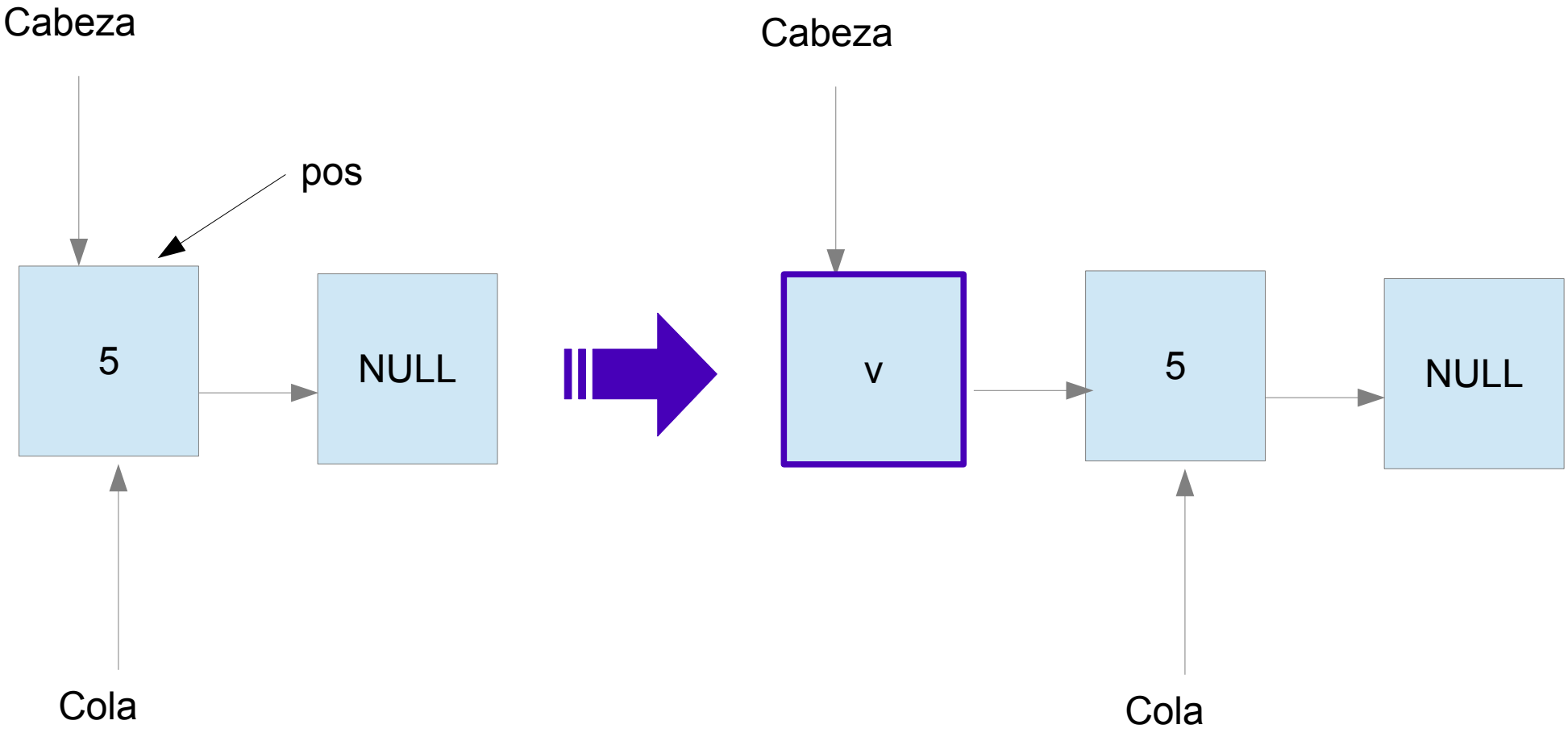
TAD Lista: eliminarCabeza()



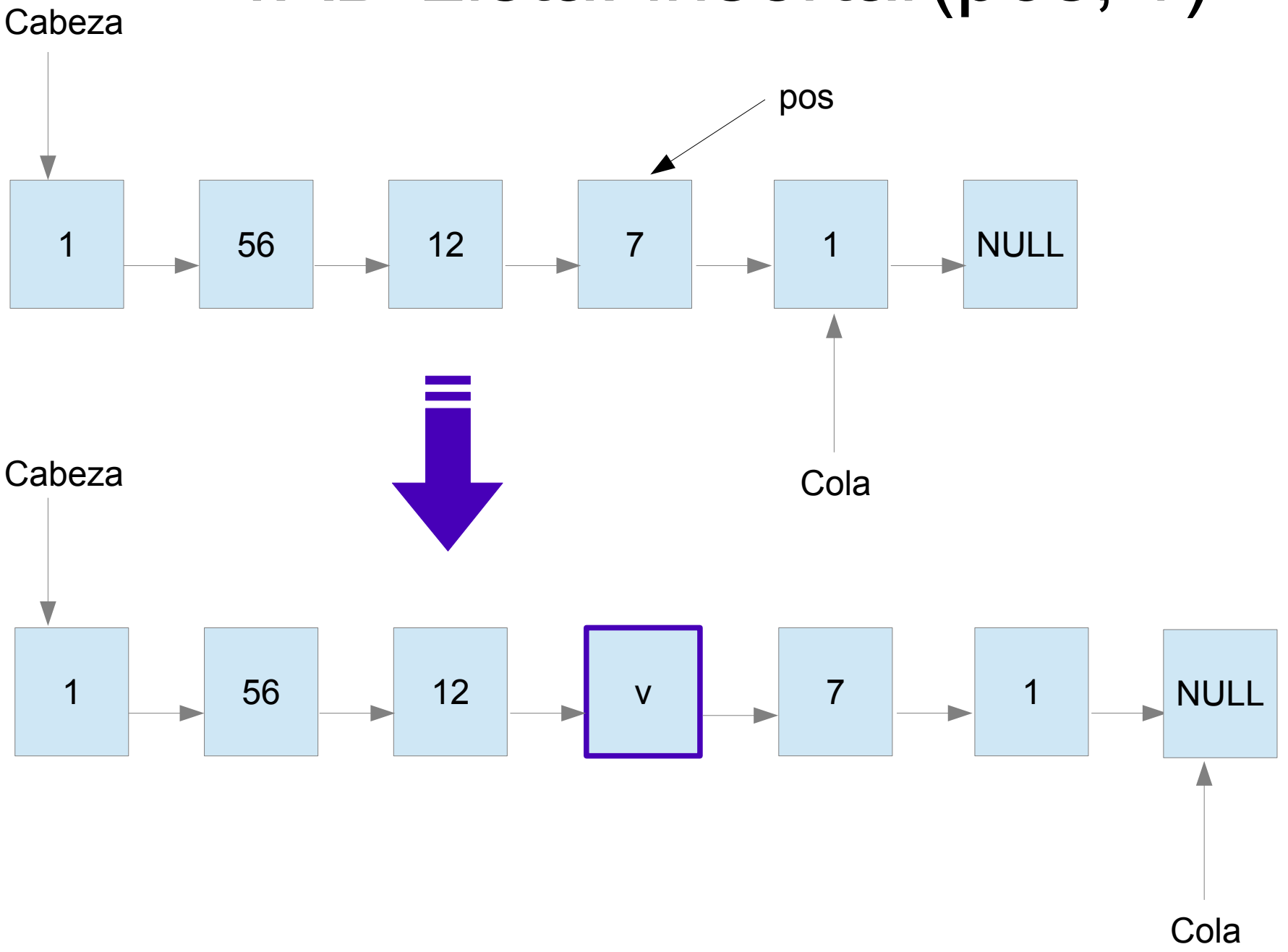
TAD Lista: insertar(pos, v)



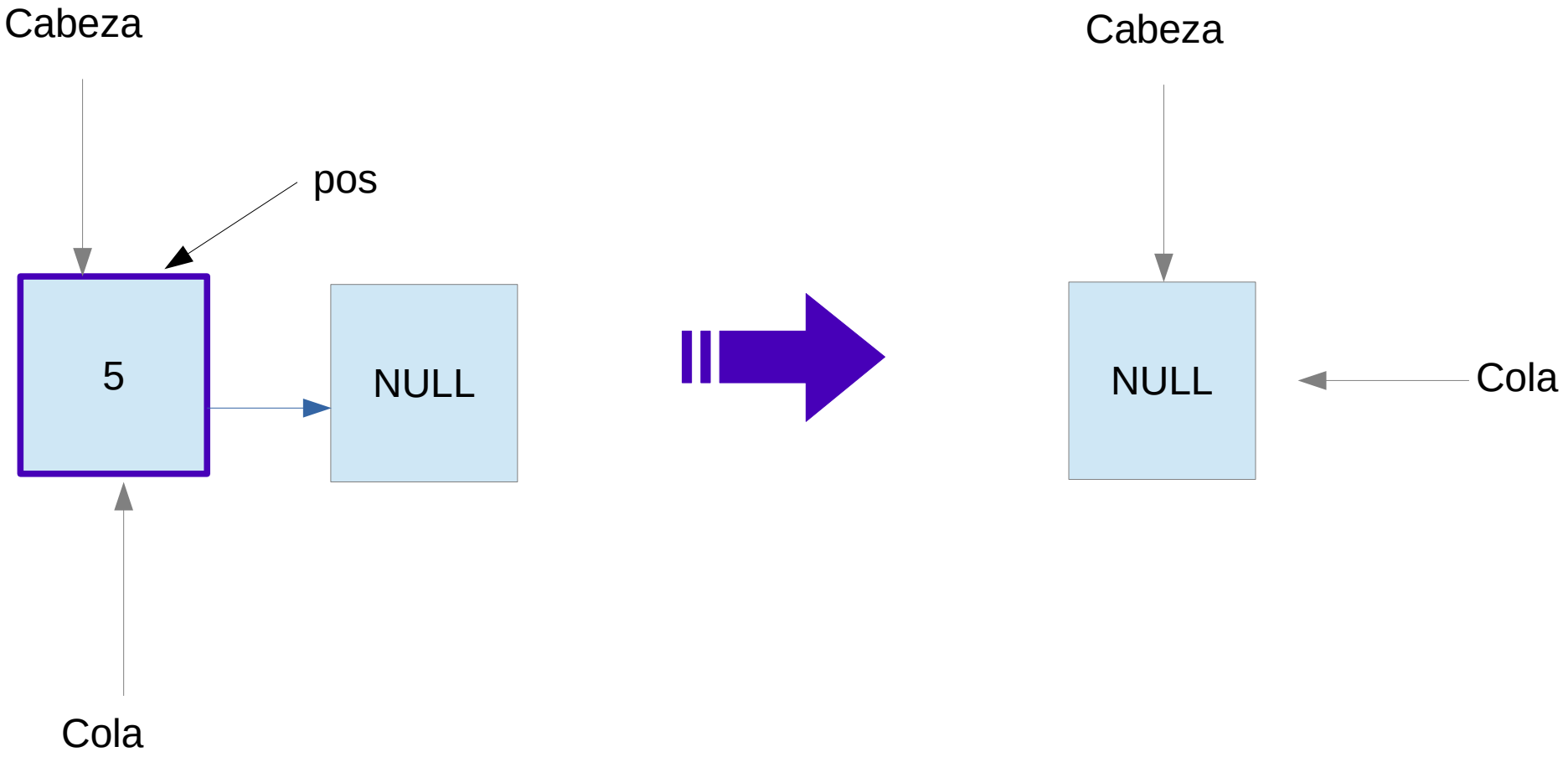
TAD Lista: insertar(pos, v)



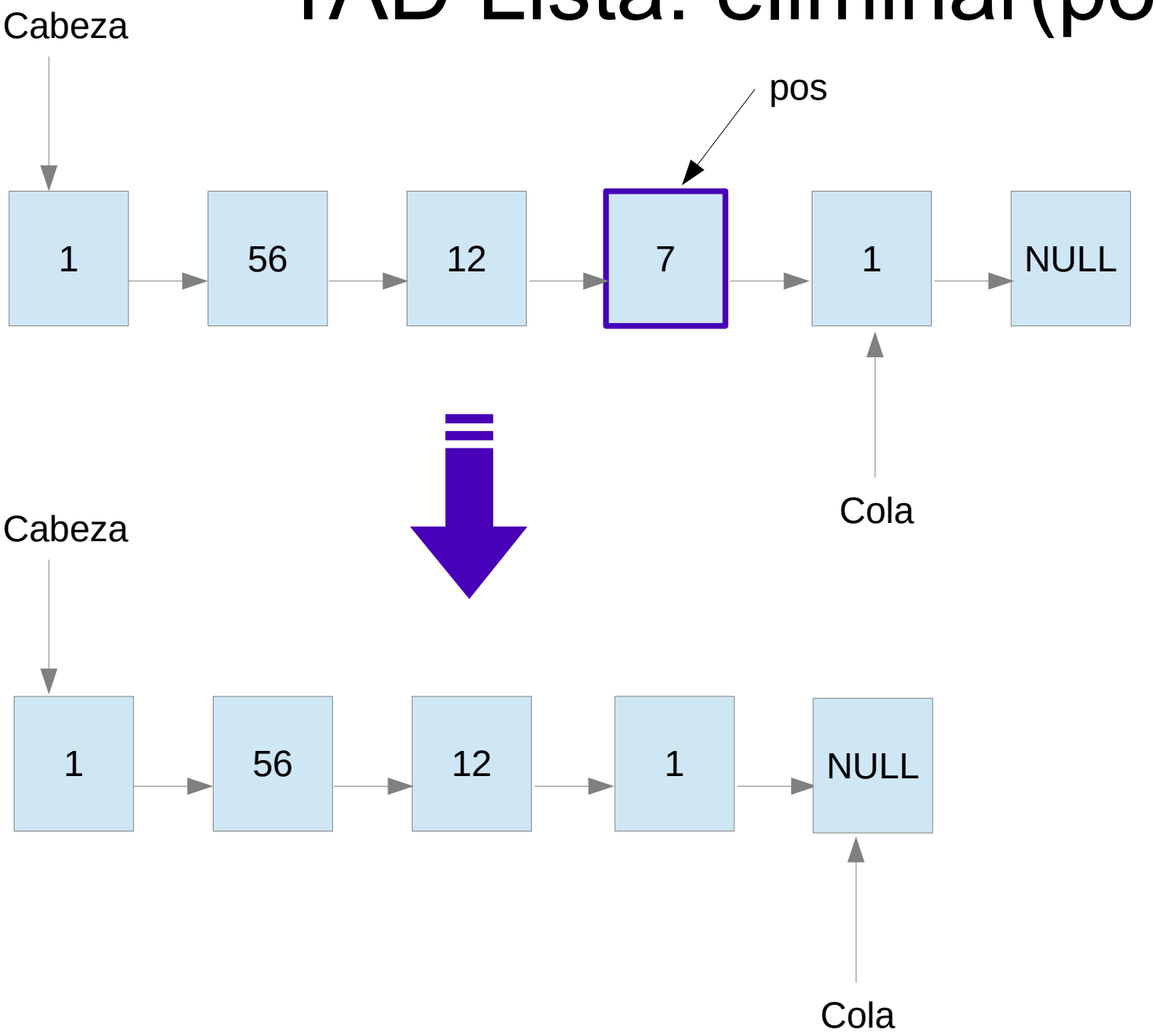
TAD Lista: insertar(pos, v)



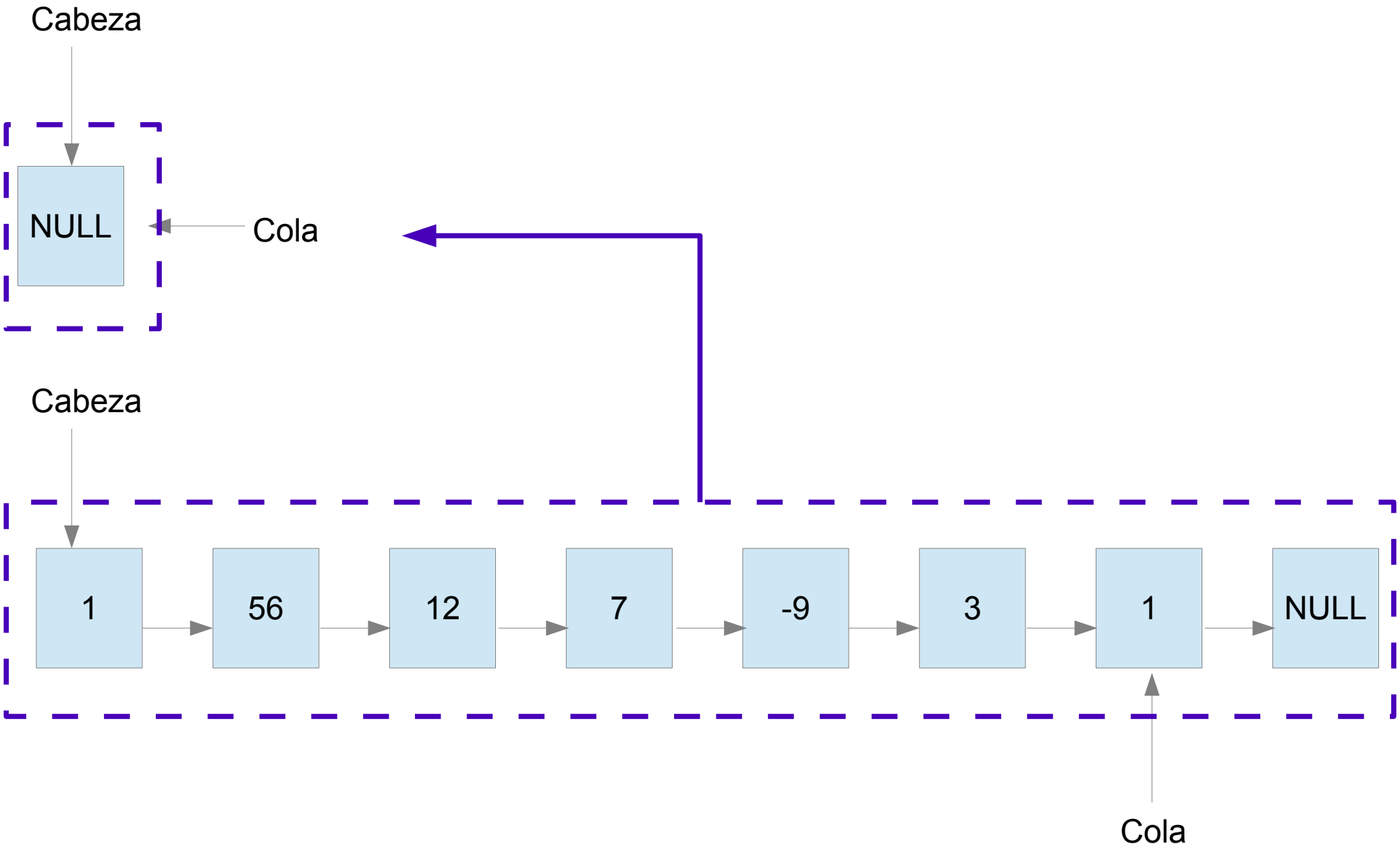
TAD Lista: eliminar(pos)



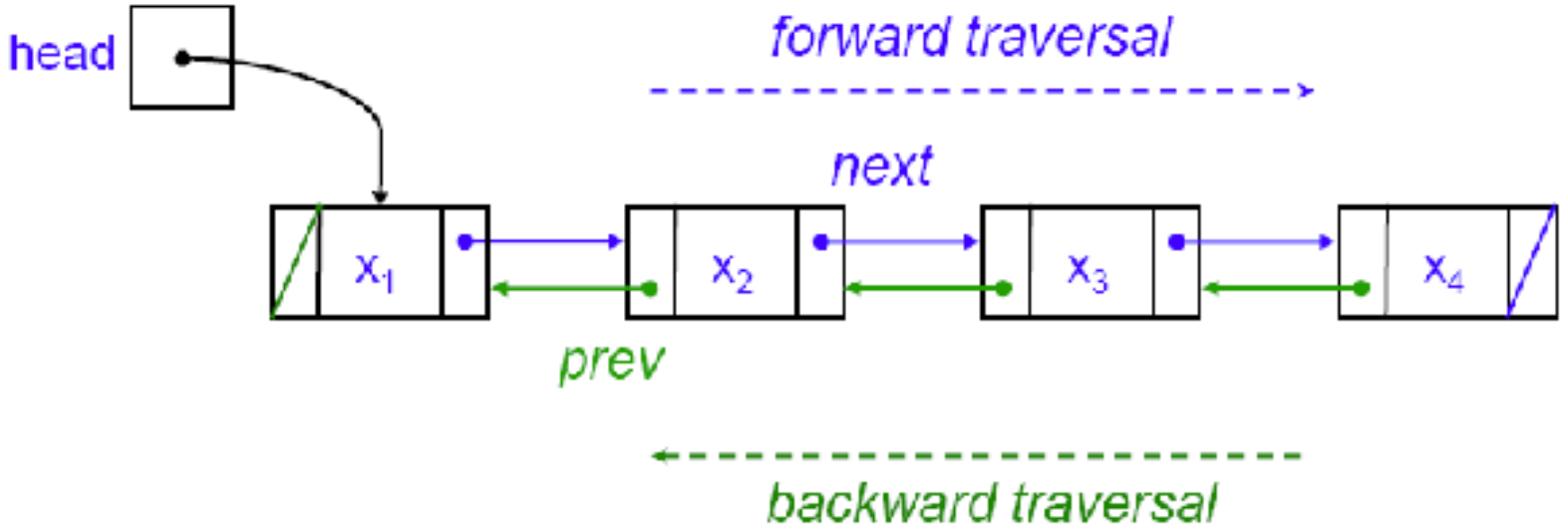
TAD Lista: eliminar(pos)



TAD Lista: vaciar()

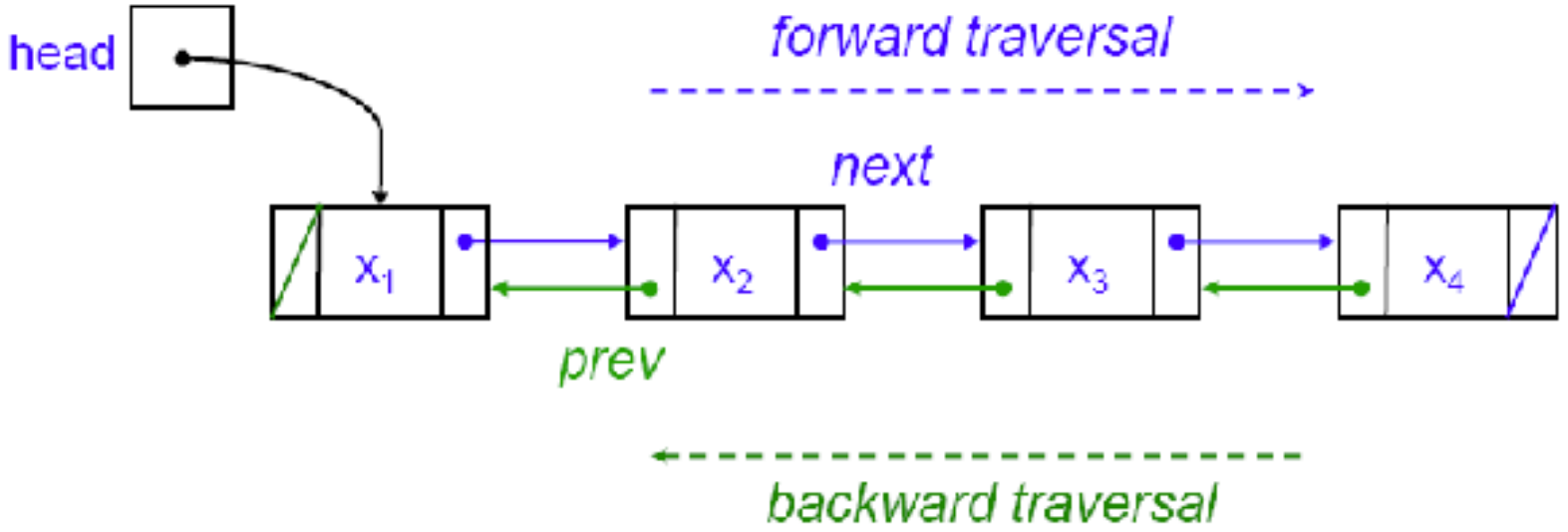


list<T>



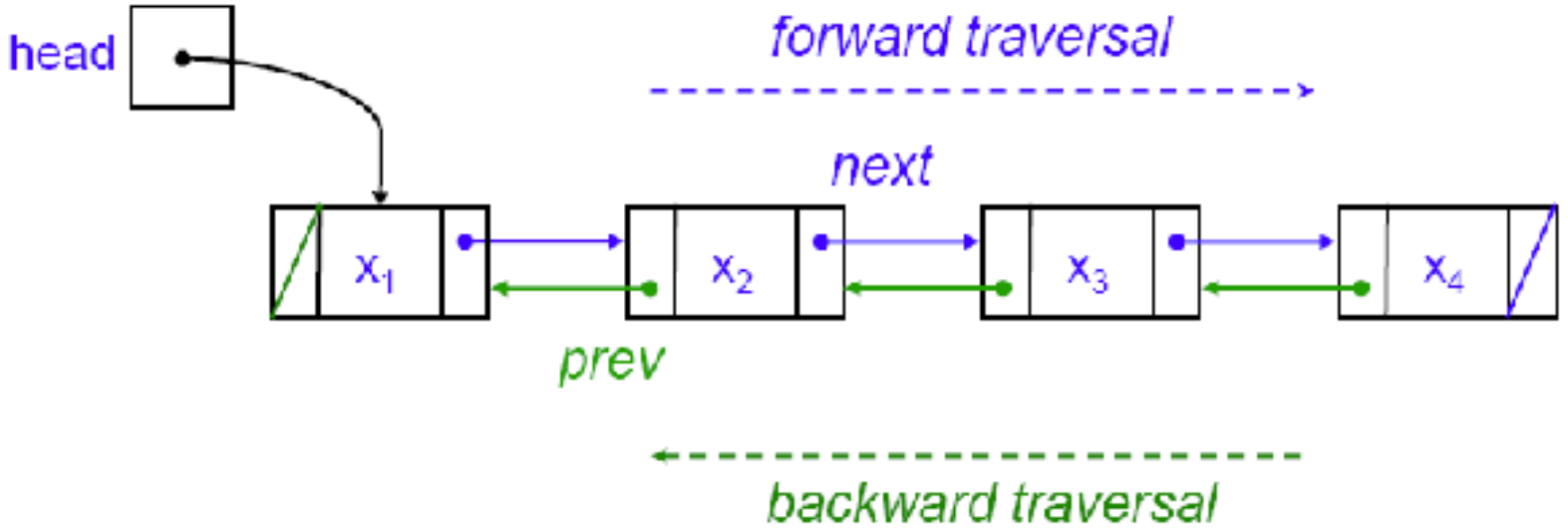
- Representa listas doblemente encadenadas (recorrido hacia adelante y hacia atrás)
- Posiciones separadas en memoria (cualquier parte)
- Elementos de un mismo tipo de dato (plantilla)
- Crece en memoria en cualquier punto de la secuencia

list<T>



- ¿Orden de complejidad de los métodos?
  - `size`, `empty`.
  - `front`, `back`.
  - `clear`.
  - `push_back`, `pop_back`.
  - `push_front`, `pop_front`.
  - `insert`, `erase`.

list<T>



- ¿Orden de complejidad de los métodos?
  - `size`, `empty`. → **O(1)**
  - `front`, `back`. → **O(1)**
  - `clear`. → **O(n)**
  - `push_back`, `pop_back`. → **O(1)**
  - `push_front`, `pop_front`. → **O(1)**
  - `insert`, `erase`. → **O(1)***

Órdenes de complejidad

	vector	deque	list
size	O(1)	O(1)	O(1)
clear	O(1)	O(1)	O(n)
empty	O(1)	O(1)	O(1)
push_front	O(n)	O(1)	O(1)
pop_front	O(n)	O(1)	O(1)
push_back	O(1)	O(1)	O(1)
pop_back	O(1)	O(1)	O(1)
insert	O(n)	O(n)	O(1)
erase	O(n)	O(n)	O(1)
¿Acceso aleatorio?	si	si	no

Criterios para escoger

- Evolución de la secuencia:
  - Estática:
    - ¿Inserción solo por final? ¿O por final y cabeza?
  - Dinámica:
    - ¿Inserción solo por final? ¿O por final y cabeza? ¿O en cualquier punto?
- ¿Memoria limitada?
- Tipo de acceso:
  - ¿Necesita acceso aleatorio?
  - ¿Predominan las iteraciones sobre toda la secuencia?

Tarea (para entregar)

- (Individual) en un archivo de código fuente:
  - Declarar un `vector`, una `deque` y una `list`.
  - Rellenar cada uno con:
    - `vector`, 25 datos aleatorios insertados por el final.
    - `deque` y `list` , 30 datos aleatorios (cada uno) insertados por ambos extremos (15 al inicio, 15 al final, intercalados).
  - Recorrer e imprimir cada contenedor en orden (con iterador normal) y en orden inverso (con iterador en reverso).
  - En cada contenedor, insertar un nuevo valor aleatorio en la posición 18, eliminar los valores ubicados en las posiciones 5 y 10, y volver a imprimir en orden normal.

Referencias

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms, 3rd edition. MIT Press, 2009.
- L. Joyanes Aguilar, I. Zahonero. Algoritmos y estructuras de datos: una perspectiva en C. McGraw-Hill, 2004.