# Estructuras de Datos Lineales

MSC. FRANCI SUNI LOPEZ
FSUNILO@UNSA.EDU.PE

CIENCIA DE LA COMPUTACIÓN - UNSA

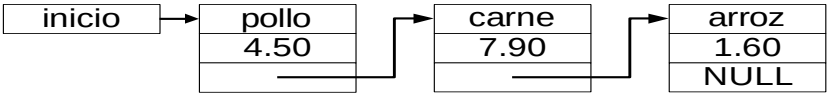


## EL TAD LISTA

El **T**ipo **A**bstracto de Datos **LISTA** es un estructura de datos básica que permite crear, insertar, modificar, buscar y eliminar datos o registros a una lista. Una lista en esencia es una secuencia de datos sucesivos.

$$a_1, a_2, a_3, \ldots, a_n$$

Donde el primer elemento es $a_1$ y el último es $a_n$. Adicionalmente su tamaño es $n$. Una lista es finita y además es flexible porque puede crecer y acortarse a voluntad.




El gráfico muestra una lista enlazada con tres elementos o nodos donde cada nodo tiene tres componentes: un campo que es una cadena de caracteres, un campo que es un flotante y un tercer campo que es una dirección o link para el siguiente nodo. La lista gráfico puede declararse como sigue:

```
struct listaNodo{
       char articulo[15];
       float precio;
       struct listaNodo *sig; // es recursiva
}
typedef listaNodo *inicio;
inicio -> articulo = strcpy("pollo","papas");
inicio -> precio = 0.80;
```



# Operaciones básicas con el TDA LISTA

1) Crear nodo.
2) Insertar nodo.
3) Eliminar nodo.
4) Buscar.
5) Recorrer nodo.
6) Lista vacía.

## Crear Nodo

Al construir una lista enlazada lo primero que debe hacerse es crear una lista vacia. Esto se logra estableciendo el primer nodo a NULL.

```
Algoritmo crearLista()
{
        inicio=NULL;
}
```

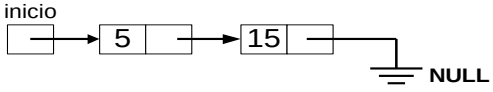
# Insertar Nodo

Para adicionar un nodo al TAD lista se hará al comienzo de la lista (puede hacerse en cualquier lugar). Los pasos a seguir son:

1) Crear un nuevo nodo.
2) Llenar el nodo con los datos que se va ha almacenar.
3) Hacer que el nuevo nodo apunte al primer nodo de la lista.
4) Hacer que el primer nodo puente al nuevo nodo.

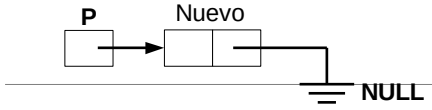
**Ejemplo grafico:**

Sea la lista de enteros 5, 15 y se desea insertar el entero 10.



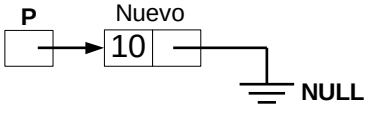



1) Crear nuevo nodo.

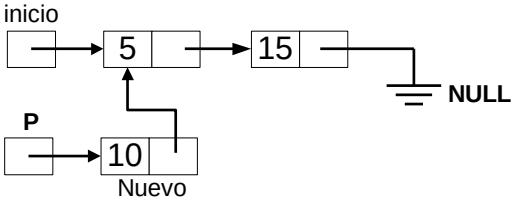



//P es un apuntador temporal a la lista de Nodos.

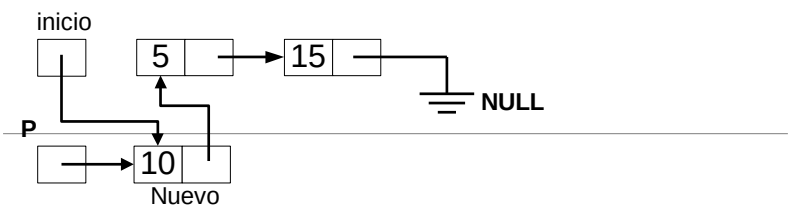
2) Llenar el nodo.




3)



4)




Los 4 pasos anteriores se resumen en el siguiente pseudocódigo

```
Algoritmo Insertar Nodo
{
        p = new Nodo
        info(p) = Elemento de datos a insertar
        sig(p) = inicio
        inicio = p
}
```



La clase básica es la siguiente:

```
struct Nodo {
        T data;
        Nodo *sig;
};
class Lista {
        Nodo *inicio;
public:
        Lista();
        ~Lista();
        void insertarNodo(T dato);
        void eliminarNodo(T dato);
        void recorrerLista();
        int listaVacia();
}
```

```
// el constructor de la lista
Lista :: Lista()
{
        inicio = NULL;
}
// destructor de la lista
Lista :: ~Lista()
{       Nodo *p;
        Nodo *temp;
        p = inicio;
        while(p != NULL) /*eliminar nodo a nodo */
        {
                temp = p -> sig;
                delete p;
                p = temp;
        }
}
```





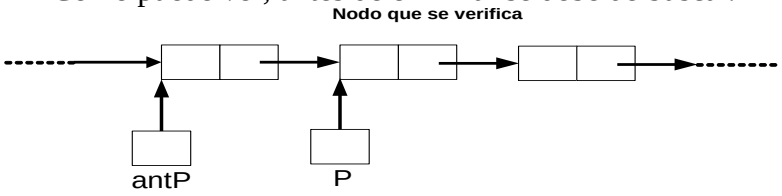
```
// operación insertar nodo
void Lista :: insertarNodo(T dato)
{
        Nodo *p;
        p = new Nodo;           // paso 1
        p -> info = dato;       // paso 2
        p -> sig = inicio;      // paso 3
        inicio = p;             // paso 4
}
```

## Operación eliminar Nodo

El algoritmo básico es el siguiente:

```
EliminarNodo
{
        Buscar en la lista el elemento a eliminar.
        Ajusta los punteros de la lista para eliminar el nodo que
        contenga el elemento que se va a eliminar.
}
```

Como puede ver; antes de eliminar se debe de buscar.






La búsqueda de un elemento se realiza Nodo a Nodo, de principio a fin, es decir, es secuencial.
El algoritmo para buscar es el siguiente:

```
Buscar
{       if lista vacia
                escribir("lista vacia...")
        else    // Ubicarse al comienzo de la lista
                p = inicio
                antp = NULL
                encuentra = false    //encuentra es una bandera
                while(NOT encuentra  AND  p  != NULL)
                {       if info(p) == dato
                                encuentra = true
                        else
                        {       //continuar busqueda
                                antp = p
                                p = sig(p)
                        }
                }
}
```

Después de utilizar el algoritmo de la búsqueda, se usan los valores de encuentra p y antp para proceder a eliminar el nodo requerido.
El algoritmo es el siguiente:

```
Eliminar
{       if(encuentra)
        {
                if(antp == NULL) //caso del 1er nodo
                        inicio = sig(p)
                else
                        sig(antp) = sig(p) //eliminacion
        }
        else
                escribir("No se encuentra en la lista...")
}
```

Ahora se puede ajustar la búsqueda con eliminar para escribir un solo algoritmo: Eliminar Nodo.

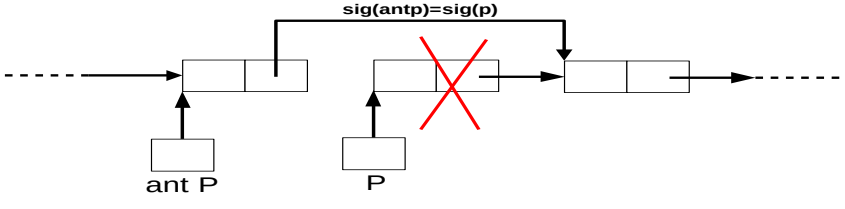


13

El algoritmo básico es el siguiente:

```
EliminarNodo
{
        Buscar en la lista el elemento a eliminar.
        Ajusta los punteros de la lista para eliminar el nodo que
        contenga el elemento que se va a eliminar.
}
```

Como puede verse, antes de eliminar se debe de buscar.






14

El algoritmo básico es el siguiente:

```
Algoritmo EliminarNodo
{
        Algoritmo Buscar
        Algoritmo Eliminar
}
```

El código:

```
// Operación EliminarNodo()
void Lista :: eliminarNodo(int dato)
{
        int encuentra = false;
        Nodo *p, *antP;
        p = inicio;
        antP = NULL;
}
```



```
        // buscar elemento a eliminar
        if( listaVacia() )
                cout<<"Lista vacia!…"<<endl;
        else
        {       while(!encuentra && p!=NULL)
                {       if (p -> info == dato)
                                encuentra = true;
                        else
                        {       antP = p;
                                p = p-> sig;
                        }
                }
                //eliminar nodo si se encuentra
                if(encuentra)
                {       if(antP == NULL)
                        {       inicio = p-> sig;
                                delete p;
                        }
                        else
                        {       antP -> sig = p -> sig;
                                delete p;
                        }
                }
                else
                        cout<<"El dato"<<dato<<"no se encuentra"<<endl;
        } // fin de 1er if
}
```

## Recorrer Lista

El recorrido de la lista es secuencial y se realiza de prinicipio a fin. El objetivo del recorrido es visitar cada nodo y enviar su(s) dato(s) hacia el flujo de salida. El seudocódigo es:

```
RecorrerLista
{       if(lista No vacia)
        {       p = inicio
                while( p !== NULL)
                {       mostrar info(p)
                        p= sig(p) //ir al siguiente nodo
                }
        }
        else
                escribir ("Lista vacia")
}
```



17

### El código

```
// Operación recorrer lista()

void Lista :: recorrerLista()
{       Nodo *p;
        p = inicio;
        if(!listaVacia())
        {       while(p!=NULL)
                {       cout<<p -> info<<"-> ";
                        p = p -> sig;
                }
                cout<<"NULO"<<endl;
        }
        else
                cout<<"lista Vacia…!"<<endl;
}
```



18

## Lista Vacía

Consiste en averiguar si la lista tiene o no elementos. Para ello se devuelve un valor booleano.

```
Lista Vacia
{       if (inicio = NULL)
                return TRUE
         else
                return FALSE
}
```

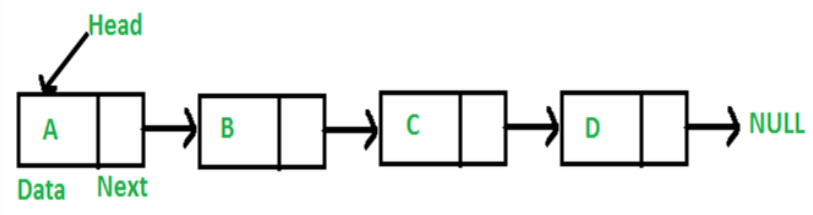
```
// código operación listaVacia()
int lista :: listaVacia()
{     if (inicio == NULL)
            return true;
       else
            return false;
}
```





¿Cómo optimizar la búsqueda?

¿Es eficiente usar una búsqueda binaria?




21

# Pilas

22

# EL TAD PILA ( STACK)

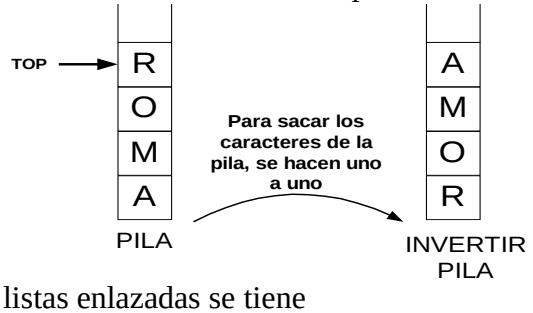
Una pila es una versión restringida de una lista enlazada. A una pila se le pueden añadir y retirar nodos únicamente por su extremo superior. Por esta razón el TDA Pila se le conoce como estructura de datos del tipo LIFO (Last In First Out) esto quiere decir; último en entrar, primero en salir. Una pila se referencia mediante un apuntador al extremo superior de la misma. El último nodo de la pila se define a NULL para indicar que se trata del último elemento de la estructura (parte inferior de la pila).



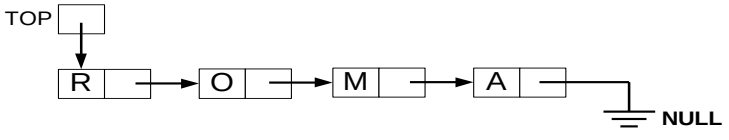
23

## Representación Gráfica

Sea el ingreso de cadena de caracteres “AMOR” los cuales se almacenan uno a uno en una pila.




Mediante listas enlazadas se tiene






24

Al puntero al extremo superior de la pila se le denomina también cima o tope

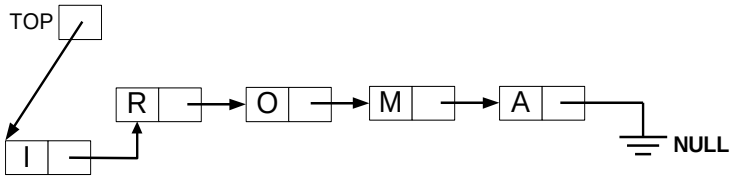
## Operaciones Básicas en una pila

1. **Operación Push o Apilar**
2. **Operación Pop o Desapilar**
3. **Operación Pila Vacía o Empty**
4. **Operación Stacktop**

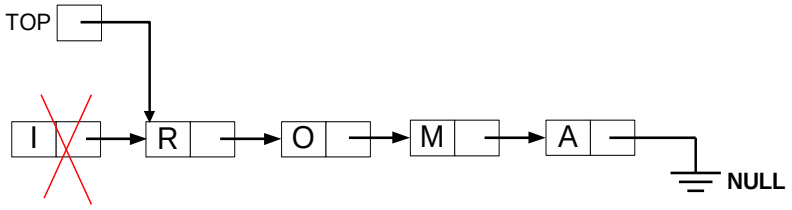


## Operaciones Básicas

**1 Operación Push o Apilar.** Significa ingresar un nuevo nodo a la pila. Esto se realiza por el extremo superior o tope.
Ejm: Insertar el carácter "I" a la pila del gráfico: push (p, carácter)



**2 <u>Operación Pop o Desapilar.</u>** Significa retirar un nodo de la pila. Esto se realiza también por el extremo superior o tope.
Ejm: Retirar un nodo de la pila anterior: pop(p)






## Operaciones Básicas

**3 <u>Operación Pila Vacía o Empty.</u>** La operación empty(p) devuelve un valor de verdad si la pila ***p*** está vacía o no. La operación **push** se puede aplicar a cualquier pila, pero la operación **pop** no puede aplicarse a una pila vacía pues no habrían elementos que remover.

El intento de aplicar **pop** a una pila vacía ocasiona un error conocido como "*underflow*" o "*bajo flujo*" por ello antes de hacer **pop** a una pila se debe verificar si ella está vacía o no.

**4 <u>Operación Stacktop.</u>** Permite averiguar que elemento está en la parte superior de la pila sin removerlo. La operación **stacktop**(p) consta de dos operaciones en secuencia:

1. Pop(p) y
2. Push(p,i).

Esto se resume en lo siguiente:

```
i = stacktop(p);
```

lo cual equivale a hacer las dos lineas siguientes:

```
i = pop (p);
push(p,i);
```

Al igual que con pop, la operación stacktop también debe evitar el subdesbordamiento o underflow.



29

## IMPLEMENTACION DEL TDA PILA

La implementación del TAD pila se puede hacer con:

1) Arreglos

2) Listas enlazadas

Programa que utiliza una clase llamada stack para almacenar caracteres en un arreglo.

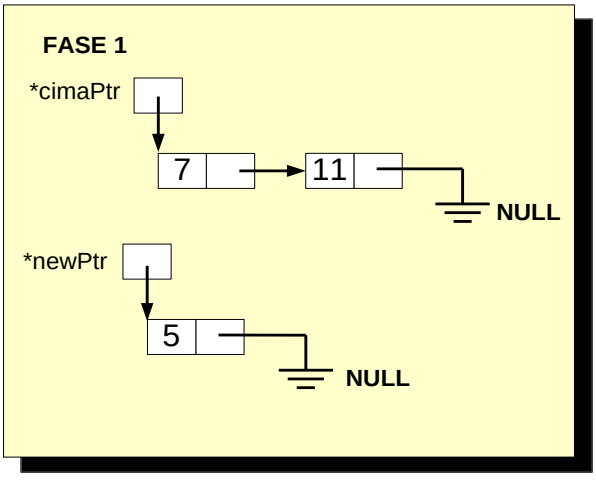
```
#include<iostream.h>
#include<stdlib.h>
const int TAM = 10;
```



30

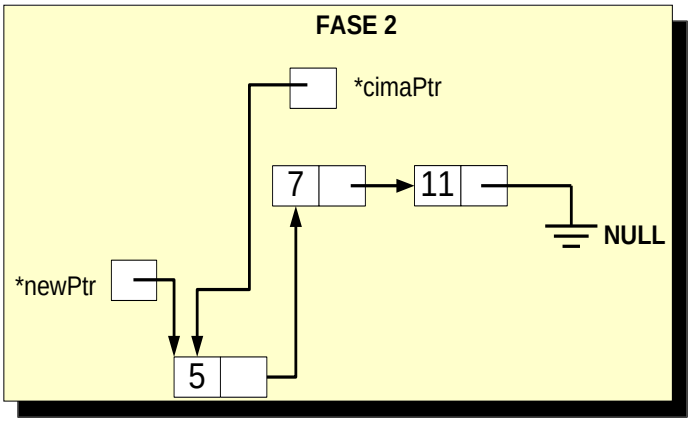
## Operación Push.

**Representación gráfica.-** Sea la pila con datos: 11, 7. Se desea insertar el 5.






35

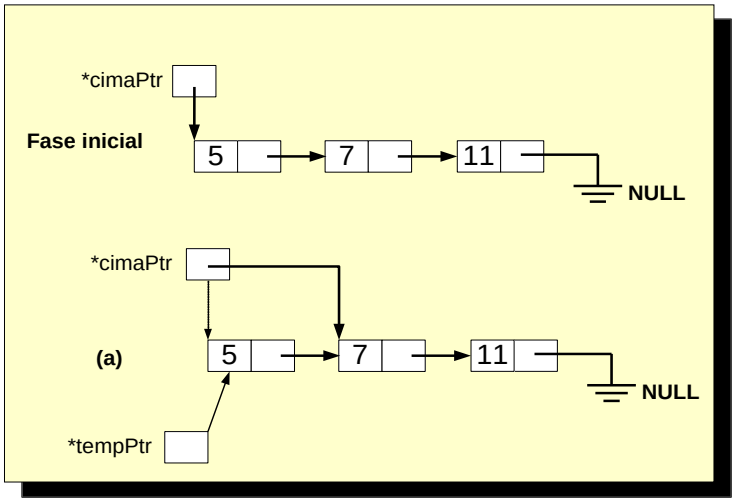





36

## Operación Pop.

**Representación gráfica.-** Hacer pop a la pila anterior.






# APLICACIONES PRACTICAS DEL TAD PILA

En computación son diversos las aplicaciones del TDA Pila.

1) Validación de expresiones.
2) Evaluación de expresiones en sus formas: infija, prefija y posfija.
3) Modelación de la memoria por parte del sistema operativo.
4) Eliminar la recursividad para lograr una versión iterativa de algunos algoritmos recursivos.

## ALGORITMO DE LA PILA
## PARA COMPROBAR SIGNOS DE COLECCION

Antes de evaluar una expresión, el compilador debe examinar si dicha expresión es válida o no. Para hacerlo procede a analizar los símbolos de colección como {, }, [, ], ( y ) o similares.

Sea la expresión:

**7-((x*((x+y)/(5-3))+y)/(4-2.5))**



Para comprobar si dicha expresión es correcta se realiza lo siguiente:

1) Averiguar si hay la misma cantidad de paréntesis izquierdos y derechos.

2) Cada paréntesis derecho esta precedido por un paréntesis izquierdo.

Volviendo a la expresión:

**7 - ( ( x * ( ( x + y ) / ( 5 - 3 ) ) + y ) / ( 4 - 2.5 ) )**

0 0 122 2 34 4 4 4 3344 4 43 2 2 21 1 2 2 2 2 1 0

contador = 0

Por lo que la expresión es correcta!.

**Pseudocódigo para el algoritmo?**



43

## NOTACION POLACA o REVERSE POLISH NORM (RPN)

La expresión en posfija se le conoce como Notación Polaca o Reverse Polish Norm (RPN).

El algoritmo para evaluar expresiones posfija, consiste en lo siguiente:

Cada operador en una cadena posfija hace referencia a los dos operandos anteriores de la cadena. Suponga que cada vez que se lee un operando lo agregamos a la pila, cuando se llega a un operador, los operandos serán los dos elementos superiores de la pila. Después se remueve estos dos elementos, se ejecuta la operación indicada sobre ellos y se agrega el resultado a la pila para que esté disponible como operando del operador siguiente.



45

**Ejemplo:** Evaluar la expresión posfija utilizando el algoritmo anterior.

6 2 3 + - 3 8 2 / + * 2 ^ 3 +

**Solución**

| simbolo | oprid1 | oprid2 | valor | pila opridstk |
|---|---|---|---|---|
| 6 | | | | 6 |
| 2 | | | | 6,2 |
| 3 | | | | 6,2,3 |
| + | 2 | 3 | 5 | 6,5 |
| - | 6 | 5 | 1 | 1 |
| 3 | 6 | 5 | 1 | 1,3 |
| 8 | 6 | 5 | 1 | 1,3,8 |
| 2 | 6 | 5 | 1 | 1,3,8,2 |
| / | 8 | 2 | 4 | 1,3,4 |
| + | 3 | 4 | 7 | 1,7 |
| * | 1 | 7 | 7 | 7 |
| 2 | 1 | 7 | 7 | 7,2 |
| ^ | 7 | 2 | 49 | 49 |
| 3 | 7 | 2 | 49 | 49,3 |
| + | 49 | 3 | 52 | 52 |



Puede realizarse el cambio de notación en las expresiones. Por ejemplo, si se quiere cambiar A+B/C a su forma posfija, se debe tener en cuenta la precedencia de los operadores. En el cado de + y / se ejecuta primero / (en ausencia de paréntesis).

| Infija | Posfija | Comentario |
|---|---|---|
| A+B/C | A+(B/C) | Se añaden paréntesis. |
| | A+(BC/) | Se pasa a posfija dentro del paréntesis. |
| | A(BC/)+ | El paréntesis y su contenido se tratan como un operando. |
| | ABC/+ | Se eliminan los paréntesis. |

Si la expresión es (A+B)/C se tiene

| Infija | Posfija | Comentario |
|---|---|---|
| (A+B)/C | (A+B)/C | |
| | (AB+)/C | Pasando a posfija dentro del paréntesis. |
| | (AB+)C/ | tomando el paréntesis como un operando y pasando a posfija |
| | AB+C/ | Quitando paréntesis. |

## REGLA

Cuando se examina operadores de la misma procedencia, se supone que el orden es de izquierda a derecha, excepto en el caso de la exponenciación en donde el orden se de derecha a izquierda. Por lo que A+B+C significa (A+B)+C, en tanto que A^B ^C significa A ^(B ^C). Observe que el uso de paréntesis ayuda a eliminar la precedencia predeterminada.

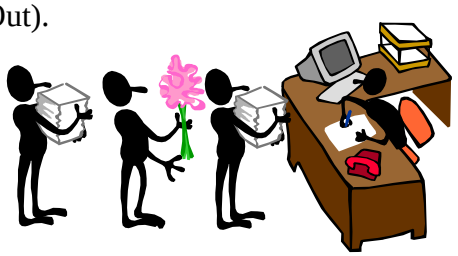




# Colas

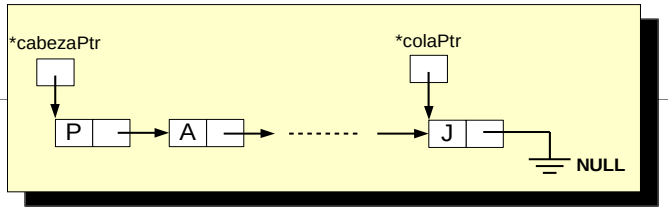
# EL TAD COLA

Una cola es un caso especial de una lista, donde sólo se permite la inserción de elementos al final y la eliminación (sacar elementos de la cola) se realiza únicamente por el frente. A las colas también se les conoce como estructuras dinámicas de datos del tipo "primero en entrar, primero en salir" FIFO (First In First Out).





**Representación Gráfica del TAD COLA**




**Operaciones en una COLA**

Las operaciones básicas definidas para la cola son:

1) **Frente o front**.- Averiguar por el elemento que está al frente.

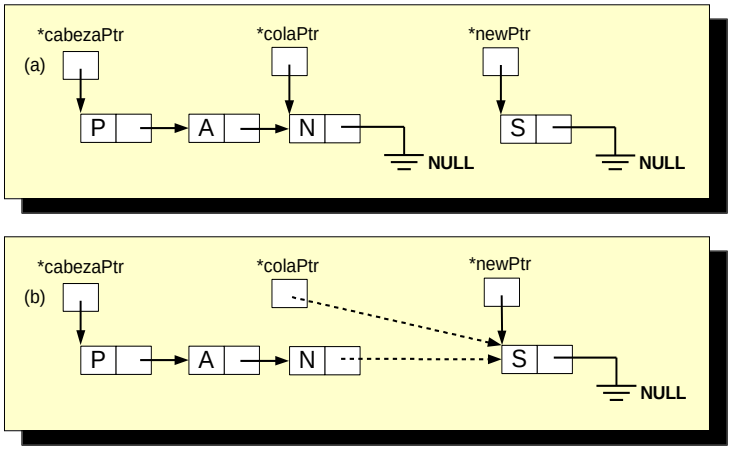
2.) Poner en cola o encolar (**enqueue**); es decir colocar elemento al final.

3) Sacar de cola o desencolar (**dequeue**); es decir sacar el elemento que está al frente.

4) Cola vacía o **empty**.

## Operación poner en cola; enqueue o encolar

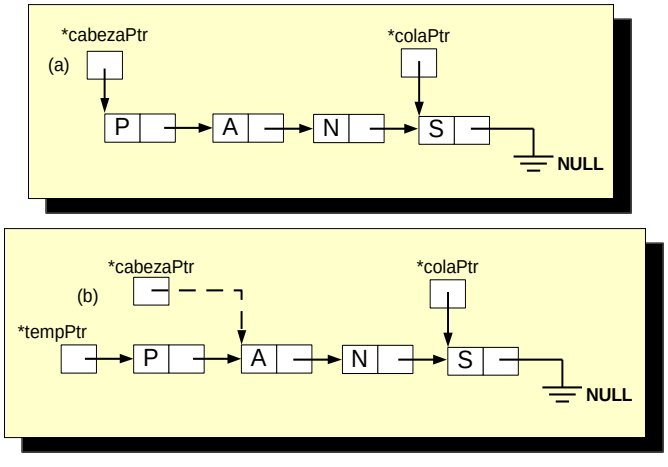
Supóngase que se tienen en cola los caracteres P, A, N y se quiere poner en la cola el carácter S. Esta situación se representa gráficamente como:





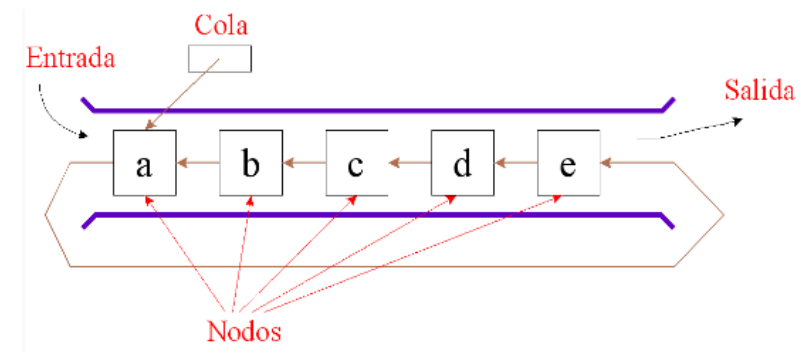

## Operación sacar de cola; dequeue o desencolar

Supóngase ahora que se quiere sacar un elemento de la cola actual P, A, N, S; el carácter que sale es P. Esta situación se representa en la figura siguiente:

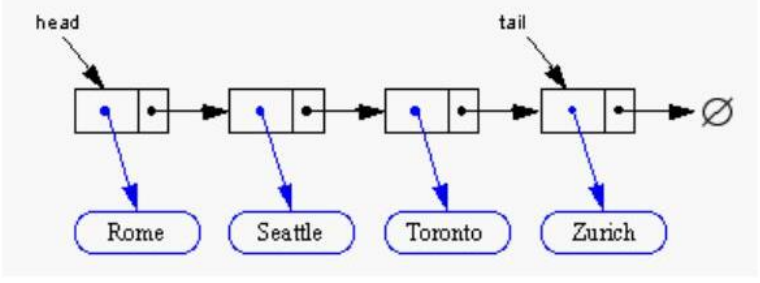




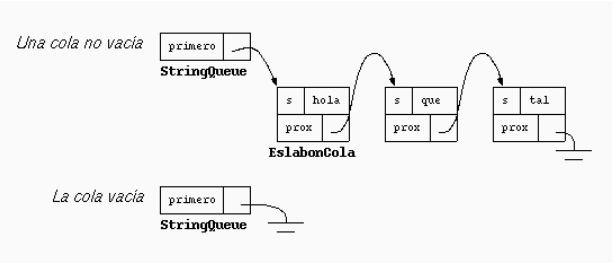

56


Cola
Entrada
Salida
a
b
c
d
e
Nodos

62

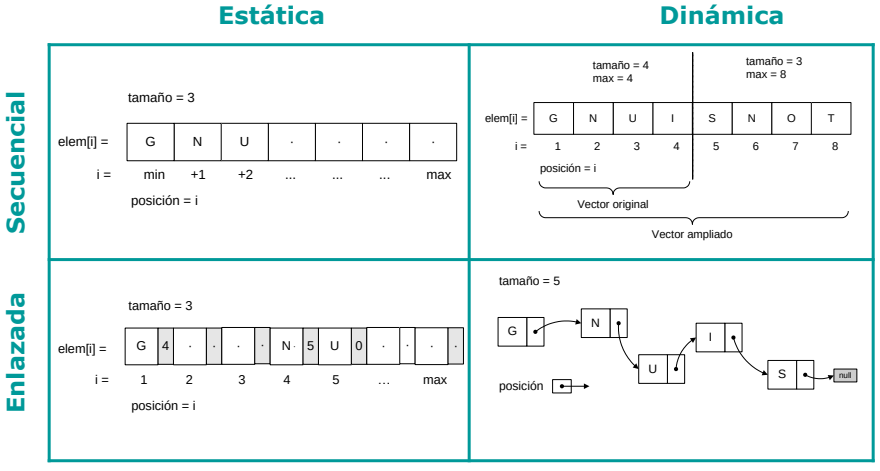

head
tail
Rome
Seattle
Toronto
Zurich

63




64

# Implementaciones




65

# Thanks!

## Questions?

M.Sc. Franci Suni Lopez
fsunilo@unsa.edu.pe

66