Tema 5.
Tipos Abstractos de Datos

Diseño de Algoritmos.

E.U. Politécnica
I.T.Industrial (Electricidad)

M. Carmen Aranda Garrido
Despacho: I-307

Tipos Abstractos de Datos

Contenidos:

1. Introducción.
2. Listas
3. Pilas
4. Colas

1. Introducción.

¿Qué es un Tipo Abstracto de Datos?

- Consta de:
  - Especificación o definición de los Datos e identificación de su Conjunto de Operaciones
  - Implementación de las estructuras de datos y las operaciones descritas en la definición.

- Puede implementarse de diversas formas siempre que cumplan su definición.

1. Introducción.

- Compilación Separada: En general, es recomendable que cuando se crea un **TAD**, se implemente en un **módulo separado**, aprovechando así las ventajas de la **compilación separada.**

- Independencia de Datos: Una característica importante de un programa que utilice un **TAD** creado por el programador es que si se cambia cualquier aspecto de la **estructura de datos** que alberga el **TAD** o de la **implementación** de sus operaciones, el programa debería seguir funcionando correctamente.

1. Introducción.

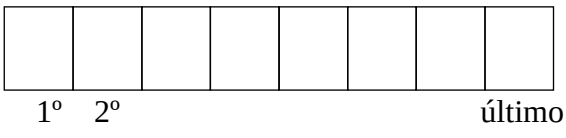
- Tipos Abstractos de Datos típicos:

1) Datos organizados por Posición ⇨ Pilas , Colas y Listas

2) Datos organizados por Valor ⇨ Arboles Binarios

2. Listas. Definición

- **Lista:** Colección de elementos homogéneos (del mismo tipo con una relación LINEAL establecida entre ellos. Pueden estar ordenadas o no con respecto a algún valor de los elementos.
- **Acceso a la Lista:** se puede acceder a cualquier elemento de la lista.



2. Listas. Operaciones

```
// Crea y devuelve una lista vacía
TipoLista Crear();
// Imprime todos los elementos de una lista
void Imprimir(TipoLista lista);
// Devuelve 1 sólo si la lista está vacía
int ListaVacia(TipoLista lista);
// Inserta un nuevo elemento en la lista. Habría que especificar posición
void Insertar(TipoLista lista, TipoElemento elem);
// Elimina un elemento de la lista. Habría que especificar la posición
void Eliminar(TipoLista lista, TipoElemento elem);
......
```


2. Listas. Implementación

1) Con un Array

```
/* TIPOS para un array estático*/
const int MaxLista = 100; /* Dimension estimada */
   typedef int TipoElemento; /* cualquiera */
   typedef struct{
      TipoElemento elementos[MaxLista];
          int num_elem; /* Número de elementos */
   }TipoLista;

/* TIPOS para un array dinámico*/
typedef int TipoElemento; /* U otro cualquiera */
typedef struct{
      TipoElemento *elementos;
      unsigned num_elem; /* num_elem: [0..] */
}TipoLista;
```

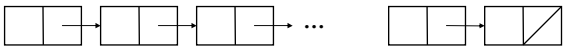
2. Listas. Implementación

2) Con una lista enlazada con punteros

```
/* Declaración de tipos de una lista simplemente
   enlazada sin cabecera */
struct element {
  long num;
  struct element *sig;
};

typedef struct element *lista;
```

Por supuesto, en vez de un único número cada elemento puede incluir cualquier cantidad de datos.



2. Lista simplemente enlazada

Funciones primitivas importantes:

```
/* Inicializa la lista */
void inicializar_lista(lista& list){
  list=NULL;
}

/* Devuelve TRUE si la lista está vacía. */
int lista_vacia(lista list){
  if (list) return 0;
  return 1;
}
```

2. Lista simplemente enlazada

```
/* Devuelve la longitud de la lista
(núm. de elems).  */
long long_lista(lista list){
  long i=0;

  while (list) {
    list=list->sig;
    i++;
  }
  return i;
}
```


2. Lista simplemente enlazada

```
/*********************************************************/
/* Devuelve en e, el elemento de la posición pos de list.*/
/* Si no existe esa posición, devuelve -1. En otro caso 0*/
int leer_element (lista list, long pos, struct element& e){
  long i=1;
  if (lista_vacia(list) || pos<1)
     return -1;

  while (list && i<pos) {
    list=list->sig;
    i++;
  }
  if (list){
     e=*list;
     return 0;
  }
  return -1;
}
```

2. Lista simplemente enlazada

```
/* Imprime un elemento e por la Salida estándar.      */
void Imprimir_elemento (struct element e){
  printf("%li",e.num);
}
/*****************************************************/
/* Muestra todos los elementos de list por su orden.  */
void mostrar_lista(lista list){
  struct element e;
  unsigned i=1,tama=long_lista(list);
  while (i<=tama) {
    printf("\nElemento %u: ",i);
    leer_element(list,i++,e);
    Imprimir_elemento(e);
  }
}
```


2. Lista simplemente enlazada

```
/*****************************************************/
/* Actualiza la posición pos de la lista list con e    */
/* Devuelve -1 si no existe esa posición.              */
int actualiza_lista(lista list, struct element e,
                    long pos){
  long i=1;
  if (pos<1) return -1;   /* Posición no válida   */
  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return -1; /* Posición no existe */
  list->num=e.num;      /* Actualización */
  return 0;
}
```

2. Lista simplemente enlazada

```
/* Inserta el elemento e en la posición pos de list.    */
/* Si pos<=1 inserta en la primera posición.            */
/* Si pos>longitud_lista, inserta en la última posición.*/
/* Devuelve -1 si no hay memoria suficiente.            */
int insertar_pos (lista& list, struct element e, long pos){
  lista p, anterior, L=list;
  long i=1;
  if ((p=(struct element *)malloc(sizeof(struct element)))
       == NULL)
     return -1;
  *p=e;
  if (pos<=1 || lista_vacia(list)){
    /* Hay que insertar en la posición 1 */
    list=p;
    p->sig=L;
    return 0;
  }
  while (L && i<pos){ /* Buscar la posición pos */
    anterior=L;
    L=L->sig;
    i++; }
  /* Insertar elemento apuntado por p, entre anterior y L*/
  anterior->sig=p;
  p->sig=L;
  return 0;
}
```

2. Lista simplemente enlazada

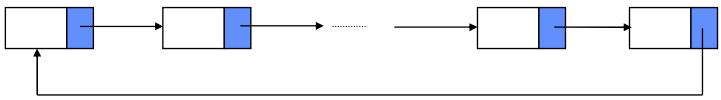
```
/* Borra el elemento en la posición pos de la lista list.  */
/* Si la pos=1 borra el primer elemento.                   */
/* Devuelve -1 si no existe la posición:                   */
/*                        pila vacía o pos>long            */
int borrar_elemento(lista& list,long pos){
  lista aux, anterior, L=list;
  long i=1;
  if (lista_vacia(list) || pos<1)
     return -1;
  if (pos==1) { /* Tratamiento para borrar elemento 1 */
     list = list->sig;
     free(L);
     return 0;
  }
  while (L && i<pos){/* Buscar la posición pos */
    anterior=L;
    L=L->sig;
    i++;
  }
  if (L){ /* OJO: Aqui no vale decir: if (i==pos)... */
     /* Borrar elemento apuntado por L,
        teniendo un puntero al anterior */
     anterior->sig=L->sig;
     free(L);
     return 0;
  }
  return -1; /* La lista tiene menos de pos elementos */
}
```

2. Lista simplemente enlazada

```
/* Libera la memoria ocupada por toda la lista.           */
void liberar_lista(lista& list){
  while (list) {
    borrar_elemento (list,1);
  }
}
/*********************************************************/
/* Devuelve posición de la primera ocurrencia del elemento*/
/* e en la lista list, a partir de posición pos(inclusive)*/
/* Esta ocurrencia será considerada por el campo num.     */
/* Devuelve 0 si no ha sido encontrada.                   */
/* Cambiando pos, podremos encontrar TODAS                */
/* las ocurrencias de un elemento en la lista.            */
long posic_lista(lista list, struct element e, long pos){
  long i=1;
  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return 0;
  while (list && e.num!=list->num) {
    /* Intentar encontrar el elemento */
    list=list->sig;
    i++;
  }
  if (!list) return 0; /* No existe */
  return i;            /* Existe en la posición i */
}
```

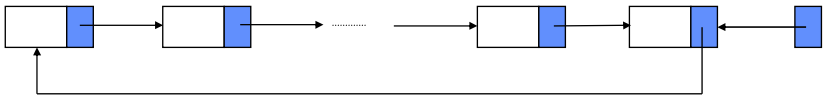
2. Listas enlazadas circulares

- Implementación estática o dinámica.
- El campo de enlace del último nodo apunta al primer nodo de la lista, en lugar de tener el valor **NULO.**
- No existe ni primer ni último nodo. Tenemos un anillo de elementos enlazados unos con otros.



2. Listas enlazadas circulares

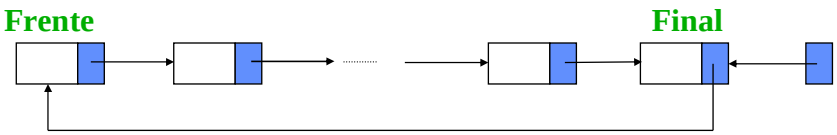
- Es conveniente, aunque no necesario, tener un enlace (puntero o índice) al último nodo lógico de la lista. Así podemos acceder facilmente a ambos extremos de la misma.



- Una lista circular vacía vendrá representada por un valor **NULO.**

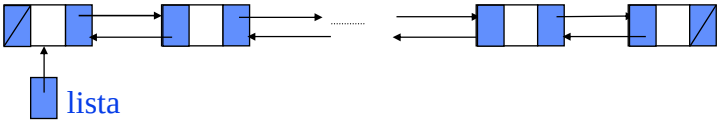
2. Listas enlazadas circulares

- Con una lista enlazada circular es muy fácil implementar una Cola, sin tener que disponer de un registro con dos campos para el frente y para el final.



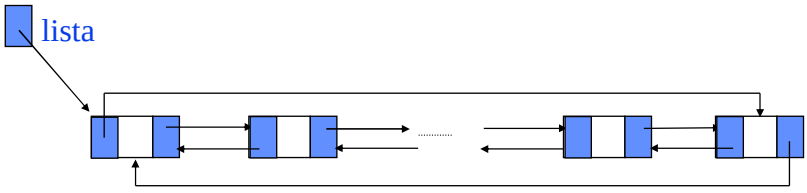
2. Listas doblemente enlazadas

- Es una lista enlazada en la que cada nodo tiene al menos tres campos:
  - Elemento. El dato de la lista.
  - Enlace al nodo anterior.
  - Enlace al nodo siguiente.
- Los algoritmos para las operaciones sobre listas doblemente enlazadas son normalmente más complicados.
- Pueden ser recorridas fácilmente en ambos sentidos.



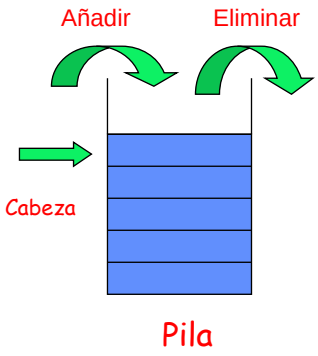
2. Listas doblemente enlazadas

- Una lista doblemente enlazada puede modificarse para obtener una estructura circular de la misma



3. Pilas. Definición

- **Pila:** Grupo Ordenado, (de acuerdo al tiempo que llevan en la pila) de Elementos Homogéneos (todos del mismo tipo).
- **Acceso a la Pila:** añadir y eliminar elementos, SÓLO a través de la CABEZA de la Pila
- Estructura **LIFO** (**L**ast **I**nput **F**irst **O**utput)



3. Pilas. Operaciones

```
TipoPila Crear(void);
/* Crea una pila vacía */
int PilaVacia(TipoPila pila);
/* Comprueba si la pila está vacía */
int PilaLlena(TipoPila pila);
/* Comprueba si la pila está llena */
void Sacar(TipoPila pila,TipoElemento& elem);
/* Saca un elemento. No comprueba antes si la pila está vacía */
void Insertar(TipoPila pila,TipoElemento elem);
/* Inserta un elemento en la pila. No comprueba si está llena. */
```

3. Pilas. Implementación

1) Con un Array

Array estructura adecuada ⇨ Elementos Homogéneos

Elementos almacenados de forma Secuencial

```
   const int MaxPila = 100; /* Dimension estimada */
/* TIPOS */
   typedef int TipoElemento; /*cualquiera */
   typedef struct{
      int Cabeza; /* Indice */
      TipoElemento elementos[100];
   }TipoPila;
```


3. Pilas. Implementación

Sólo es posible acceder a la Cabeza de la Pila

⇩

¿ Cómo es posible conocer la posición de la cabeza?

1) Variable entera "cabeza" ⇨ Inconveniente: se ha de pasar como parámetro adicional a todas las operaciones sobre la pila

2) Extender el array, en pila[0] almacenaremos el índice del elemento que ocupa la cabeza actual

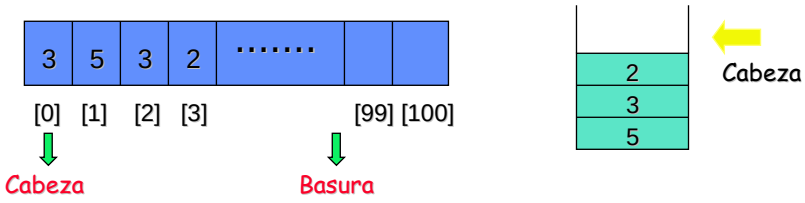
3. Pilas. Implementación

Constantes

```
Cabeza=0; MaxPila=100;
```

Tipos

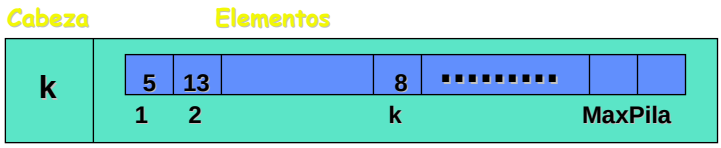
```
TipoElemento TipoPila[MaxPila]
```



3. Pilas.Implementación

- Inconveniente: Solo es posible implementar una pila de enteros (no de cualquier otro tipo de datos)
- Solución:

```
typedef int MaxPila=100;
typedef int TipoElemento; /* o cualquiera */
typedef struct{
    int Cabeza;
    TipoElemento Elementos[MaxPila];
} TipoPila;
```

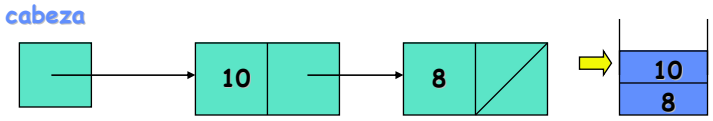


3. Pilas.Implementación

2) Con una Lista Enlazada de Punteros

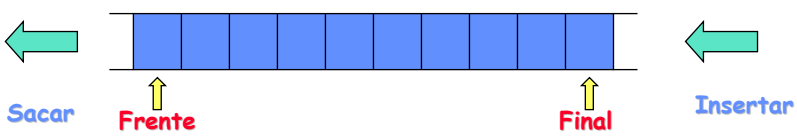
- Comienzo de una lista enlazada ⇨ Cabeza de la Pila

```
/* TIPOS */
      typedef int TipoElemento; /*o cualquiera */
      typedef struct nodo{
            TipoElemento dato;
            struct nodo *sig;
      }TipoNodo;
      typedef TipoNodo *TipoPuntero;
      typedef TipoPuntero TipoPila;
```



4. Colas. Definición

- Cola: es un grupo ordenado (con respecto al tiempo que llevan en él) de elementos homogéneos (todos del mismo Tipo)
- Acceso: los elementos se añaden por un extremo (final) y se sacan por el otro extremo (frente)
- Estructura **FIFO** (**F**irst **I**nput **F**irst **O**utput)



4. Colas. Operaciones

```
TipoCola Crear();
/* Crea y devuelve una cola vacía */
int ColaVacia(Tipocola Cola);
/* Devuelve 1 sólo si "cola" está vacía */
int ColaLlena(TipoCola Cola);
/* Devuelve 1 sólo si "cola" está llena */
int Sacar(Tipocola Cola, TipoElemento& elem);
/* Saca el siguiente elemento del frente y lo pone en "elem" */
int Insertar(TipoCola Cola,TipoElemento elem);
/* Inserta "elem" al final de la cola */
```

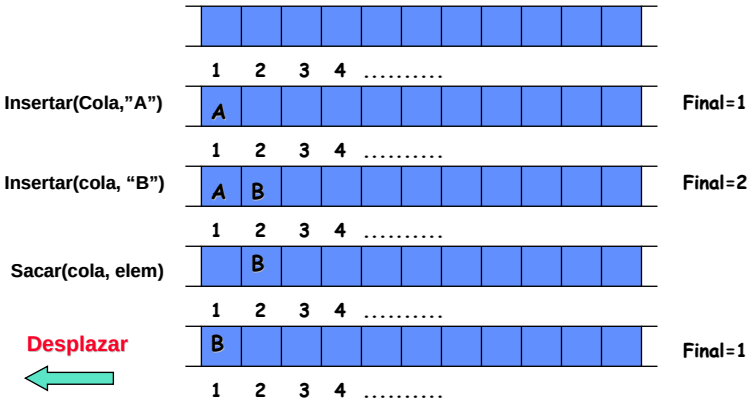

4. Colas. Implementación

1) Con un Array

- Se deja fijo el frente de la cola y se mueve el final a medida que se añaden nuevos elementos (Idem Pila)
- Las operaciones Insertar, Crear, ColaVacia y ColaLlena se implementan de una forma similar a sus análogas en Pilas
- La operación de Sacar es mas complicada: cada vez que saquemos un elemento de la cola se han de desplazar el resto una posición en el array, para hacer coincidir el frente con la primera posición del array
- Ventaja ⟹ Simplicidad
- Inconveniente ⟹ Ineficiente (colas con muchos elementos o elementos grandes)

4. Colas. Implementación

- Ejemplo:



4. Colas. Implementación

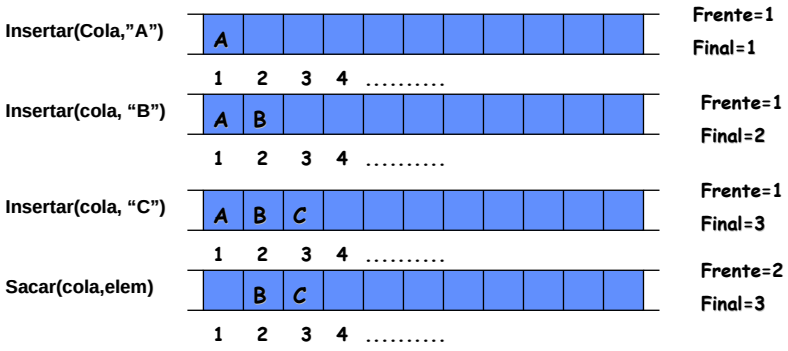
Solución:

- Utilizar un índice para el frente y otro para el final y permitir que ambos fluctúen por el array

- Ventaja: operación Sacar más sencilla
- Inconveniente: Es posible que final sea igual a Maxcola (última casilla del array) y que la cola no esté llena

4. Colas. Implementación

- Ejemplo:

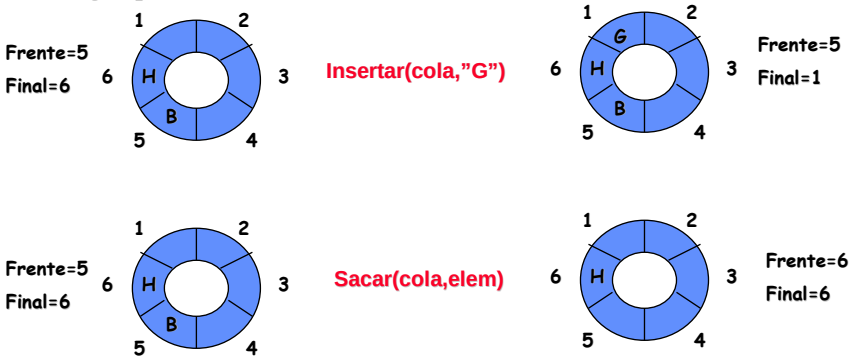


4. Colas. Implementación

- Solución:
- Tratar al array como una **Estructura Circular**, donde la última posición va seguida de la primera ⇨ Evitamos que el final de la cola alcance el final físico del array y no esté llena
- Operación Insertar ⇨ Añade elementos a las posiciones del array e incrementa el índice final
- Operación Sacar ⇨ Más sencilla. Sólo se incrementa el índice frente a la siguiente posición

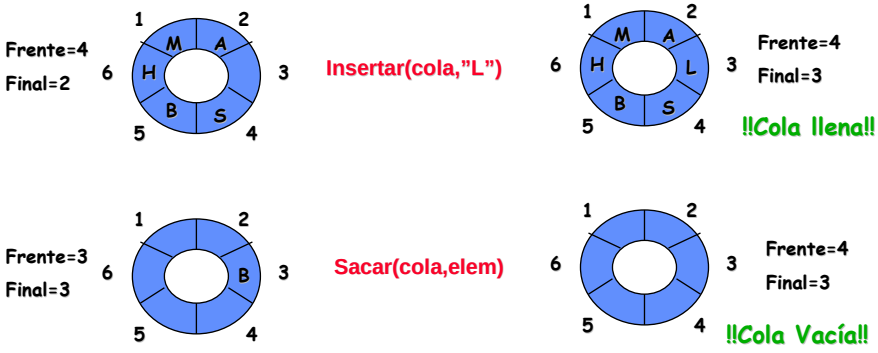
4. Colas. Implementación

- **Ejemplo:**



4. Colas. Implementación

¿Como sabemos si la cola está vacía o llena?



4. Colas. Implementación

- Solución:
- 1) Disponer de otra variable ⇨ Contabilizará los elementos almacenados en la cola

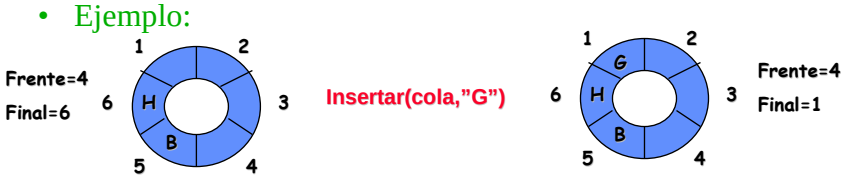
  Variable=0 ⇨ Cola vacía

  Variable=MaxCola ⇨ Cola llena

  Inconveniente: añade más procesamiento a las operaciones Insertar y Sacar
- 2) Frente apunte a la casilla del array que precede a la del elemento frente de la cola ⇨ Solución elegida

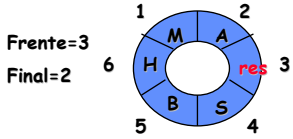
4. Colas. Implementación

- Ejemplo:



¿Como saber si la cola está llena? Es necesario que la posición a la que apunta frente en cada momento este Reservada

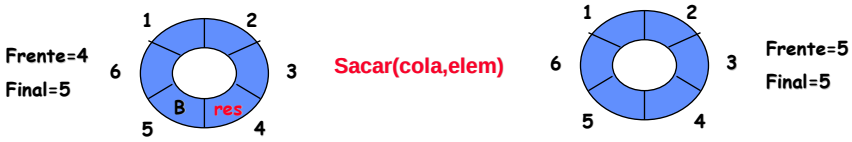
Cola Llena: final+1 =frente



4. Colas. Implementación

¿Como saber si la cola está vacía?

Cola Vacía: Frente = Final

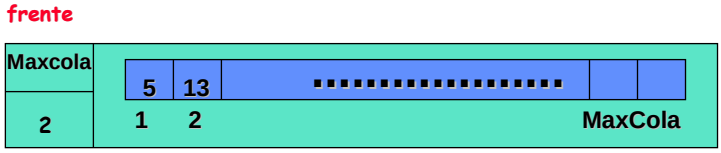


- Crear la cola (inicializarla vacía): frente =Maxcola (índice del array que precede al elemento frente de la cola) y final=Maxcola ⟹ Cola Vacía correcto (frente = final)

4. Colas. Implementación

- Agrupamos en una estructura los índices frente y final, junto con el array que contendrá los elementos de la cola

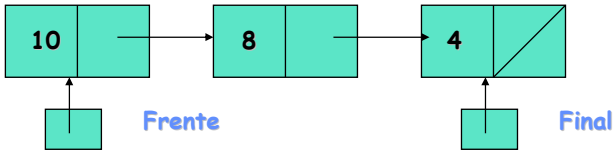
```
typedef int TipoElemento /* U otro cualquiera */
typedef struct{
      TipoElemento elementos[MaxCola];
      unsigned frente; /* Frente: [0..MaxCola-1] */
      int final; /* Final: [0..MaxCola-1] */
}TipoCola;
```



4. Colas. Implementación

2) Con listas enlazadas con Punteros

- Usamos dos variables de tipo puntero, frente y final, que apunten a los nodos que contienen los elementos frente y final



- ¿Que sucedería si intercambiáramos las posiciones de frente y final en la lista enlazada?

4. Colas. Implementación

- Agrupamos las variables frente y final en una estructura

```
typedef int TipoElemento; /* O cualquiera */
typedef struct nodo{
           TipoElemento valor;
           struct nodo *sig;
  }TipoNodo;
typedef TipoNodo *TipoPuntero;
typedef struct{
           TipoPuntero frente;
           TipoPuntero final;
  }TipoCola;
```