TEMA 3.- TAD LINEALES.

3.0 Introducción

- Los TAD constituyen una forma de generalizar y encapsular los aspectos más importantes de la información que manejamos dentro del programa, olvidando hasta el momento de la implementación como se va a representar esa información dentro del ordenador.

- El TAD se puede ver como un conjunto de valores y operaciones que se definen sobre ellos independientes de su implementación

- Los TAD se dividen en:

  - **TAD Lineales:** Aquellas estructuras abstractas de datos en que cada elemento, salvo el primero y el último, tiene como mucho dos elementos adyacentes (posterior y/o anterior) y desde un elemento cualquiera sólo se puede acceder a uno de estos dos elementos como máximo.

  - **TAD no Lineales:** Aquellas estructuras cuyos elementos pueden tener más de dos elementos adyacentes, a los que pueden acceder directamente (no tiene sentido el concepto de anterior/siguiente), como los ***árboles o grafos***

- En este tema se estudia la primera gran familia de TADs, los **TAD lineales**, todos ellos derivados del concepto de *secuencia*: ***listas, pilas y colas.***

- Los diferentes TADs basados en este concepto se diferenciarán por las operaciones de acceso a los elementos y manipulación de la estructura.

- Las operaciones básicas para dichas estructuras son:

  1. **crear** la secuencia vacía
  2. **añadir** un elemento a la secuencia
  3. **borrar** un elemento de la secuencia
  4. **consultar** un elemento de la secuencia
  5. comprobar si la secuencia está **vacía**

- La diferencia entre las tres estructuras que se estudiarán vendrá dada por la posición del elemento a añadir, borrar y consultar:

  - **Listas**: las tres operaciones se realizan sobre una posición cualquiera de la secuencia.

  - **Pilas**: las tres operaciones actúan sobre un extremo de la secuencia.

  - **Colas**: se añade por un extremo y se borra y consulta por el otro.

3.1 TAD Lista

3.1.1. Definición y Concepto

> ***Una lista es un conjunto ordenado de elementos homogéneos en la que no hay restricciones de acceso, la introducción y borrado de elementos puede realizarse en cualquier posición de la misma.***

- Son TAD en los que los elementos están organizados siguiendo un orden secuencial
- Cada elemento tiene un anterior y un siguiente en la lista
- Las listas genéricas son estructuras muy flexibles, no existe una restricción en la localización de los elementos insertados o extraídos
- Las listas pueden crecer o acortarse.
- A los elementos de una lista se les suele llamar ***nodos*** (o *celdas*).

En la implementación del TAD lista de manera ***estática***, la eliminación o inserción de un elemento supondrá la reorganización del resto de elementos del vector.

La implementación ***dinámica*** es mucho más eficiente, y mediante ella podremos construir:

- *Listas Simplemente Enlazadas* (cada elemento tiene acceso al siguiente)
- *Listas Doblemente Enlazadas* (cada elemento tiene acceso al anterior y al siguiente)
- *Listas Circulares* (el último elemento tiene acceso al inicio de la lista)

3.1.2. Especificación.

<u>espec</u> listas

<u>usa</u> booleanos, naturales

<u>parámetro formal</u>

<u>género</u> elemento

<u>operaciones</u>

_ == _: elemento elemento → booleano

_ ≠ _: elemento elemento → booleano

<u>fpf</u>

<u>género</u> lista

<u>operaciones</u>

[ ]: → lista

+izq: elemento lista → lista

[_] : elemento → lista

_ & _ : lista lista → lista

+dch: lista elemento → lista

vacía?: lista → booleano

<u>parcial</u> -izq: lista → lista

<u>parcial</u> -dch: lista → lista

<u>parcial</u> izq: lista → elemento

<u>parcial</u> dch: lista → elemento

longitud: lista → natural

está?: elemento lista → booleano

parcial insertar: lista nat elemento → lista *{insertar elem. i-ésimo}*

parcial eliminar: lista nat → lista *{eliminar elem. i-ésimo}*

parcial modificar: lista nat elemento → lista *{modi. elem. i-ésimo}*

parcial _ [_] : lista nat → elemento *{elemento i-ésimo}*

parcial pos: elemento lista → natural *{posición del elemento}*

dominios de definición e: elemento; l: lista; i: natural

-izq (+izq (e, l))

-dch (+izq (e, l))

izq (+izq (e, l))

dch (+izq (e, l))

insertar (l, i, e) **está definido sólo si** $(1 \leq i) \wedge (i \leq long\ (l) + 1)$

eliminar (l, i) **está definido sólo si** $(1 \leq i) \wedge (i \leq long\ (l))$

modificar (l, i, e) **está definido sólo si** $(1 \leq i) \wedge (i \leq long\ (l))$

l[i] **está definido sólo si** $(1 \leq i) \wedge (i \leq long\ (l))$

pos (e, l) **está definido sólo si** está? (e, l)

ecuaciones e, e1, e2: elemento; l: lista; i: natural

[e] = +izq (e, [ ])

[ ] & l = l

+izq (e, l1) & l2 = +izq (e, l1 & l2)

longitud ([ ]) = 0

longitud (+izq (e, l)) = suc (long (l))

está? (e, [ ]) = falso

está? (e1, +izq (e2, l)) = e1 == e2 $\vee$ está? (e1, l)

insertar ([ ], i, e) = +izq (e, [ ]) *{ i solo puede valer 1. $1 \leq i \leq 1$ }*

insertar (+izq (e1, l), i, e2)) = ***si*** *i = 1* ***entonces*** *+izq (e2, +izq (e1, l))*
sino *+izq (e1, insertar (l, i -1, e2))*
fsi

eliminar (+izq (e, l), i) = ***si*** *i = 1* ***entonces*** *l { $1 \leq i \leq$ long (+izq (e, l)) }*
sino *+izq (e, eliminar (l, i -1))*
fsi

modificar (+izq (e1, l), i, e2) = ***si*** *i = 1* ***entonces*** *+izq (e2, l)*
{ $1 \leq i \leq$ long (+izq (e, l)) }
sino *+izq (e1, modificar (l, i-1, e2))*
fsi

+izq (e, l) [i] = ***si*** *i = 1* ***entonces*** *e*
sino *l [i-1]*
fsi

pos (e1, +izq (e2, l)) = ***si*** *e1 == e2* ***entonces*** *1*
sino *suc(pos (e1, l))*
fsi

+dch ([ ], e) = +izq (e, [ ])

+dch (+izq (e1, l), e2) = +izq (e1, +dch (l, e2))

-izq (+izq (e, l)) = l

-dch (+izq (e, l)) = ***si*** *vacía? (l)* ***entonces*** *[]*
sino *+izq (e, -dch (l))*
fsi

izq (+izq (e, l)) = e

dch (+izq (e, l)) = ***si*** *vacía? (l)* ***entonces*** *e*
sino *dch (l)*
fsi

fespec

3.1.3. Implementaciones.

Una lista podría implementarse utilizando un **array** (**vector**), almacenando en cada posición de la tabla un nodo de la lista.

VENTAJAS:

- Con esta implementación los elementos de la lista se almacenan en posiciones contiguas de memoria.
- Sencillo acceso a los nodos de la lista, ya que cada nodo sería directamente accesible mediante un índice.

INCONVENIENTES:

- La inserción y borrado requieren un desplazamiento de lugar de los elementos.
- Limitación del número de nodos de la lista → Estructura estática
- Las estructuras estáticas en las que su número de elementos y su disposición son permanentemente fijos, tienen ciertos problemas a la hora de añadir y eliminar elementos de la estructura.
- Así por ejemplo, si en una tabla de componentes ordenados:

1	8	10	40	100	…	…	…	…	…

se quiere intercalar en su lugar el valor 9, se tendrán que desplazar hacia la derecha los tres últimos elementos quedando:

1	8	9	10	40	100	…	…	…	…

Tipos de representación

Podemos pensar en varios tipos de representación usando un vector para almacenar los elementos de la lista:

1. **Primera idea**:

Guardar los elementos en el vector en el mismo orden y sin espacios vacíos.

lista = (*e*1,*e*2,*e*3,*e*4)

0	1	2	3	4	5	6	7
e_1	e_2	e_3	e_4				

2. **Segunda idea**:

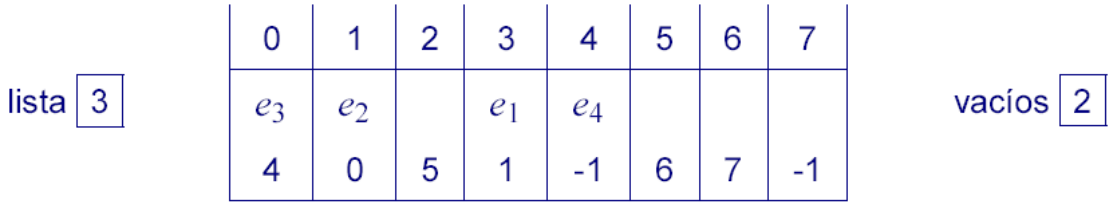
- No guardar los elementos en orden ni consecutivos.
- Junto a cada elemento guardar un índice al siguiente elemento.
- Otro índice, **lista**, indica la posición del primer elemento.
- Un valor especial (por ejemplo -1) indica fin de lista.

lista | 3 |

0	1	2	3	4	5	6	7
e_3	e_2		e_1	e_4			
4	0		1	-1			

3. **Lista de vacíos**:

   Las posiciones vacías podrían ser otra lista.



La implementación estática con índices es una primera versión de lista.

Inconveniente de todos los tipos de representación:

Siempre se está reservando toda la memoria ocupada por el vector independientemente del número de elementos que tenga la lista.

Implementación

- Implementamos en C++ una lista donde el tipo de los elementos es entero (elemento → int), haciendo uso de diseño modular y orientación a objetos.
- Al utilizar orientación a objetos el TAD lista será definido como una clase y sus operaciones serán métodos de la clase lista y por tanto tendrán que ser llamados desde un objeto de la clase.
- La operación de crear una lista vacía se convierte en el constructor de la clase lista.
- Los parámetros cuyo tipo es el que aparece en el **género** se convierte en el objeto sobre el que se aplica el método.

```
// FICHERO TADLista.h
# define MAX 30
# include <iostream>
using namespace std;

class lista {
    int  elementos[MAX];//elementos de la lista
    int  n;//nº de elementos que tiene la lista

    public:
        lista();
        lista(int e);
        bool esvacia();
        int longitud();
        void anadirIzq(int e);
        void anadirDch(int e);
        void eliminarIzq();
        void eliminarDch();
        int observarIzq();
        int observarDch();
        void concatenar(lista l);
        bool pertenece(int e);
        void insertar(int i, int e);
        void eliminar(int i);
        void modificar(int i, int e);
        int observar(int i);
        int posicion(int e);
};
```

//FICHERO TADLista.cpp

```
# include “TADLista.h”

lista::lista() {n=0;}
lista::lista(int e)
{
 n=1;
 elementos[0]=e;
}

void lista::insertar(int i, int e)
{
int pos,postabla;
postabla=i-1;
if (n < MAX){
 for (pos=n-1; pos>=postabla; pos--)
  elementos[pos+1]=elementos[pos];//Desplazamiento
 elementos[postabla]=e;
 n++;
}
}

void lista::eliminar(int i){
 int postabla;
 postabla=i-1;
 while (postabla<n-1)
  {
```

```
  elementos[postabla]=elementos[postabla+1];
//Desplazamiento
  postabla++;
  }
 n--;
}

void lista::modificar(int i, int e){
elementos[i-1]=e;}

int lista::observar(int i){
return(elementos[i-1]);}

bool lista::esvacia (){return (n == 0);}

int lista::posicion(int e){
int i=0;
while (  (elementos[i] != e)  &&  (i < n)  )
    i++;
if (elementos[i] == e)   return(i+1);
else return (-1);
}

int lista::longitud (){return n;}

void lista::anadirIzq(int e){insertar(1,e);}

void lista::anadirDch(int e){insertar(n+1,e);}
```

```
void lista::eliminarIzq()
 {
  for (int i=0;i<n-1;i++)
   elementos[i]=elementos[i+1];
  n--;
 }

void lista::eliminarDch(){  n--; }

int lista::observarIzq() {return(observar(1));}

int lista::observarDch(){ return(observar(n)); }

void lista::concatenar(lista l)
 {
  int lon=l.longitud();
  for (int i=1;i<=lon;i++)
   insertar(n+1,l.observar(i));
 }

bool lista::pertenece(int e)
 {
  return (posicion(e)==-1?false:true);
 }
```

3.2.4. Aplicaciones

- Existen multitud de ejemplos donde se hace uso de este tipo abstracto de dato. Si pensamos en una lista de la compra o en una guía telefónica estamos manejando listas.

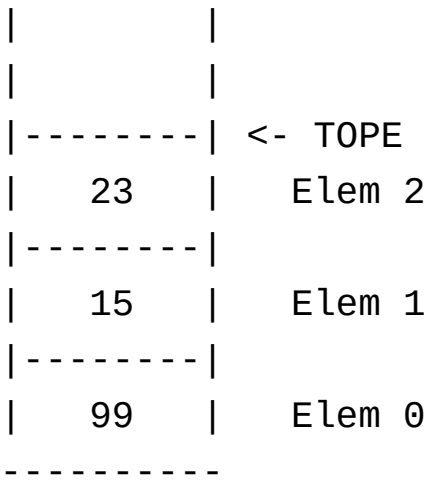
- Asimismo existen numerosas estructuras relacionadas directamente con las listas y distintas variantes que ofrecen diferentes propiedades de acceso. Es el caso de los TAD pila y cola que se verán a continuación y para los que veremos diversas aplicaciones con detalle.

3.2 TAD Pila (STACK)

3.2.1 . Definición y Concepto

> ***Una pila es un tipo especial de lista en la que la inserción y la eliminación de sus elementos se realizan sólo por un extremo que se denomina tope (cima, cabeza o cabecera).***

- Es un TAD que se caracteriza por el modo de acceso a sus elementos:



Ej: un montón de platos.

- La pila es una estructura con numerosas analogías en la vida real, su comportamiento es similar al de un conjunto de elementos apilados unos

sobre otros, p.e.: una pila de platos, una pila de monedas, una pila de libros, etc.,

- Estructuras donde los elementos sólo pueden eliminarse en orden inverso al que se insertan en la pila.

- En estos TAD, el último elemento que se pone en la pila es el primero que se puede sacar, por eso a estas estructuras se les conoce por el nombre "listas **LIFO**" (**L**ast **I**n, **F**irst **O**ut, o último en entrar, primero en salir).

- No existe un método de acceso directo a cualquier elemento de la pila, para acceder a uno de ellos es necesario desapilar los anteriores (los que estén "por encima" de éste).

- Llevan asociados una variable llamada *tope* que indica la posición del último elemento apilado.

- Los elementos se insertan de uno en uno (**apilar**)

- Se sacan en el orden inverso al cual se han insertado (**desapilar**)

- El único elemento que se puede observar dentro de la pila es el último insertado (**tope o cima**)

3.2.2. Especificación

<u>espec</u> pilas

<u>usa</u> booleanos, naturales

<u>parámetro formal</u>

género elemento

fpf

género pila

operaciones

creaPila: → pila

apilar: pila elemento → pila

parcial desapilar: pila → pila

parcial cima: pila → elemento

vacía?: pila → booleano

longitud: pila → natural

dominios de definición p: pila; e: elemento

desapilar (apilar (p, e))

cima (apilar (p, e))

ecuaciones p: pila; e: elemento

desapilar (apilar (p, e)) = p

cima (apilar (p, e)) = e

vacia? (creaPila) = verdad

vacia? (apilar (p, e)) = falso

longitud (creaPila) = 0

longitud (apilar (p, e)) = suc (longitud (p))

fespec

- *desapilar* y *cima* no están definidas para pila vacía
- Patrones necesarios para representar todas las posibles *pilas:*
  - ***creaPila***: representa la pila sin ningún elemento (pila vacía)

- ***apilar(p, e)***: representa cualquier pila con, al menos, un elemento

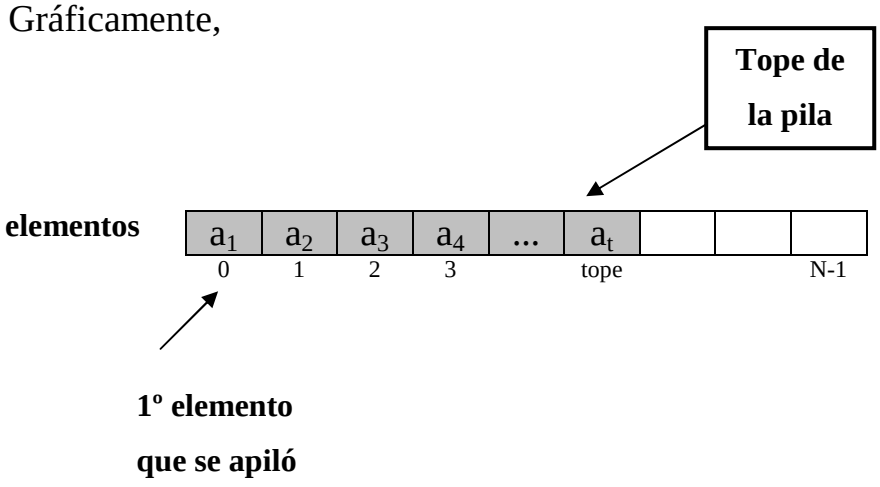
- Gen (pila) = {creaPila, apilar}
- Mod (pila) = {desapilar}
- Obs (pila) = {cima, vacía?,longitud}

3.2.3. Implementaciones

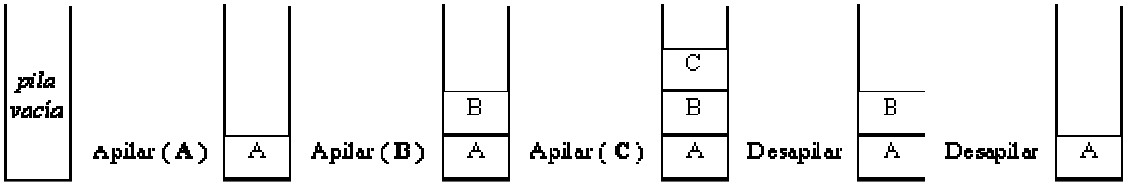
- Podemos pensar en una implementación de las pilas mediante un *array*, al igual que lo hemos visto para las listas.

- Sería una forma sencilla, pero presenta el principal inconveniente de las estructuras estáticas: *hemos de asignar un tamaño máximo a priori para la tabla que puede resultar excesivo o insuficiente.*

- Gráficamente,



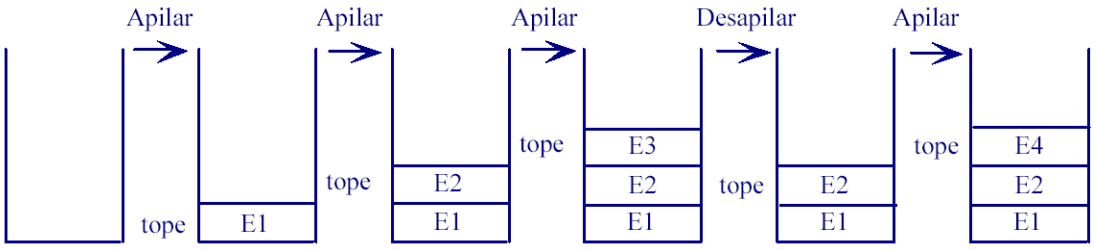
- Al poner un elemento en la pila se incrementará el tope y al sacar un elemento de la pila se decrementará el tope.



- Las operaciones aplicables a una pila son un subconjunto de las operaciones aplicables a una lista, considerando que las inserciones y

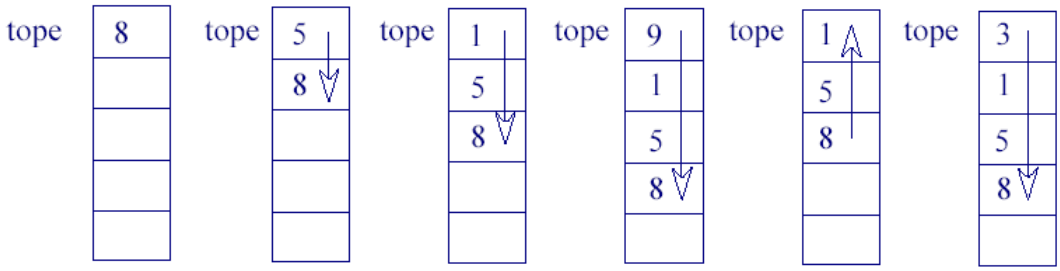
borrados tienen lugar siempre en la misma posición, posición que recibe el nombre de tope o *cabecera de la pila*, como hemos visto.

- Estas operaciones asociadas a las pilas y que son las más usuales, reciben generalmente los nombres de ***apilar*** y ***desapilar***.

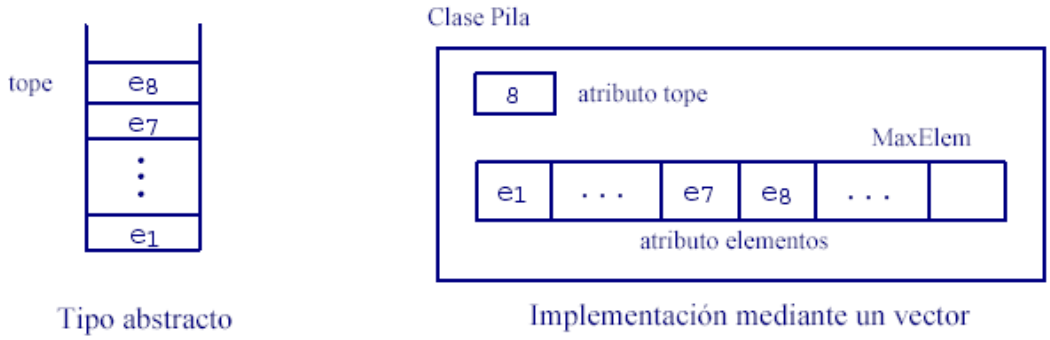


Implementación

Primera Opción - *Sin utilizar orientación a objetos*



Segunda Opción - *Utilizando orientación a objetos*



```
// FICHERO TADPila.h
# define MAX 30
# include <iostream>
using namespace std;
class pila {
    int  elementos[MAX]; //elementos de la pila
    int  tope ;//tope de la pila
    public:
    pila(); // constructor de la clase
    void apilar(int e);
    void desapilar();
    int cima();
    bool esvacia();
    int longitud();
    };
```

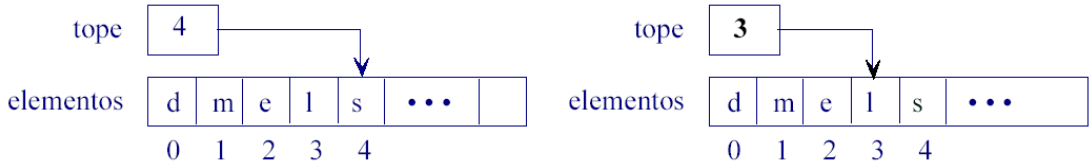
```
//FICHERO TADPila.cpp
# include "TADPila.h"

pila::pila(){tope=-1;}

void pila::apilar(int e){
if (tope+1<MAX){
tope++;
elementos[tope]=e;
}
}
```

```
int pila::longitud(){ return tope+1; }
void pila::desapilar(){tope--;}

/* El elemento no se "borra" del vector
```




```
*/

int pila::cima(){return(elementos[tope]);}

bool pila::esvacia (){return (tope == -1);
}
```

3.2.4. Aplicaciones

- Las pilas son utilizadas ampliamente para solucionar una gran variedad de problemas. Se utilizan en compiladores, sistemas operativos y en programas de aplicación.

- Algunas de estas aplicaciones son:

    - El gestor de programas del S.O. utiliza una pila para guardar momentáneamente los parámetros y dirección de retorno de la función que se está procesando actualmente.
    - Los editores de texto proporcionan normalmente un botón *deshacer* que cancela las operaciones de edición recientes y restablece el

estado anterior del documento. La secuencia de operaciones recientes se mantiene en una pila.

- Los navegadores permiten habitualmente volver hacia atrás en la secuencia de páginas visitadas. Las direcciones de los sitios visitados se almacenan en una pila.
- Estructuras auxiliares en numerosos algoritmos y esquemas de programación:
  - recorridos de árboles y grafos
  - evaluación de expresiones
  - conversión entre notaciones (postfija, prefija, infija)

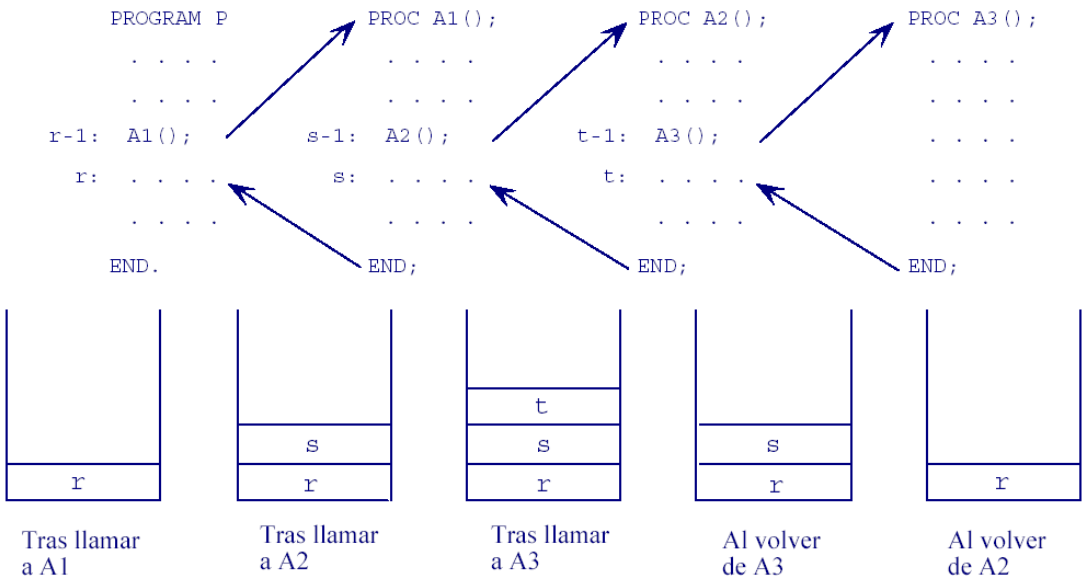
Veamos con más detalle una de sus aplicaciones más comunes:

Llamadas a subprogramas.

Cuando dentro de un programa se realizan llamadas a subprogramas, el programa principal debe de recordar el lugar desde donde se hizo la llamada, de modo que pueda retornar allí cuando el subprograma se haya terminado de ejecutar.

Ejemplo

Supongamos que tenemos tres subprogramas llamados A1, A2 y A3, y un programa principal P y que se realizan las llamadas que se muestran gráficamente. De esta manera A1 que es el primero que se ejecuta será el último en terminar y devolver el control al programa principal que sólo así podrá terminar. Esta operación se consigue disponiendo las direcciones de retorno en una pila.

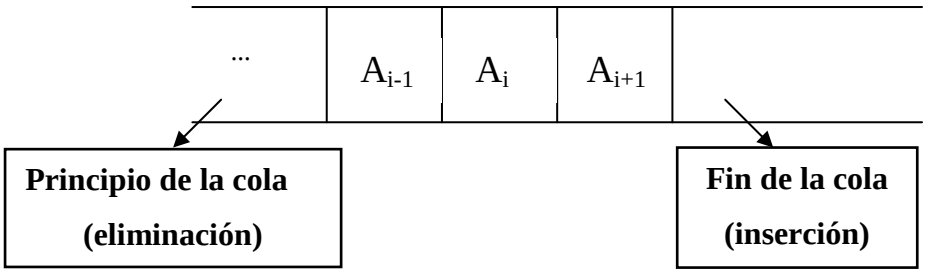


Cuando un subprograma termina debe retornar a la dirección siguiente a la instrucción que le llamó. Cada vez que se invoca a un subprograma, la dirección siguiente (r, s o t) se introduce en la pila. El vaciado de la pila se realizará por los sucesivos retornos, decrementándose el tope de la pila que queda siempre en la siguiente dirección de retorno.

3.3 TAD Cola (QUEUE)

3.3.1 . Definición y Concepto

Son TADs formados por una secuencia de elementos, caracterizados porque sus elementos se insertan por un extremo y se extraen por el extremo opuesto (el primer elemento en insertarse es el primero en extraerse).



- Las colas son estructuras lineales de datos, similar a las pilas, diferenciándose de ellas en el modo de insertar/eliminar elementos.

- Una cola es otro tipo especial de lista en la cual los elementos se insertan por un extremo – por el final de la lista - y se eliminan por el otro extremo –por el principio de la lista -.

- En las colas el elemento que entró el primero, sale también el primero, por ello se conocen como "listas **FIFO**" (**F**irst **I**nput, **F**irst **O**utput, o primero en entrar, primero en salir).

- La diferencia con las pilas reside en el modo de entrada/salida de datos; en las colas las inserciones se realizan al final de la lista, no al principio.

- Se denomina así porque el comportamiento de esta estructura de datos es similar al de una cola de personas en un cine, de coches en un atasco, etc., y por ello se usan para almacenar datos que necesitan ser procesados según el orden de llegada.

- El único elemento observable en todo momento es el primero que fue insertado.

3.3.2. Especificación

especificación colas

usa booleanos, naturales

parámetro formal

género elemento

fpf

género cola

operaciones

creaCola: → cola

encolar: cola elemento → cola

parcial desencolar: cola → cola

parcial primero: cola → elemento

vacía?: cola → booleano

longitud: cola → natural

dominios de definición c: cola; e: elemento

desencolar (encolar (c, e))

primero (encolar (c, e))

ecuaciones c: cola; e: elemento

desencolar (encolar (c, e)) = **si** vacia?(c) **entonces** creaCola
sino encolar (desencolar(c), e)
fsi

primero (encolar (c, e)) = **si** vacia?(c) **entonces** e
sino primero(c)
fsi

vacia? (creaCola) = verdad

vacia? (encolar (c, e)) = falso

longitud (creaCola) = 0

longitud (encolar (c, e)) = suc (longitud (c))

fespec

Gen (cola) = {creaCola, encolar}

Mod (cola) = {desencolar}

Obs (cola) = {primero, vacia?,longitud}

- Patrones necesarios para representar todas las posibles *colas:*
  - *creaCola*: representa la cola sin ningún elemento (cola vacía)
  - *encolar(c, e)*: representa cualquier cola con, al menos, un elemento

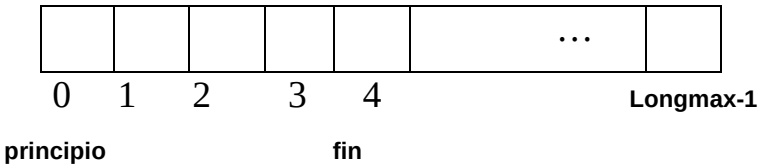
3.3.3. Implementaciones

- Podíamos pensar en una representación de las colas mediante un array (*vector),* lo que sería una forma sencilla, pero presenta el mismo inconveniente visto para las listas y pilas: hemos de asignar un tamaño máximo a priori para la tabla que puede resultar excesivo o insuficiente.

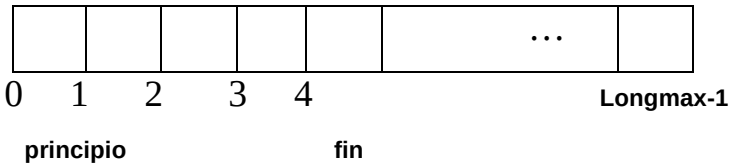
Veamos varias opciones para representar las colas con vectores:

1. Almacenamos los elementos de la cola en un array, considerando el **principio** de la cola *fijo* en la primera posición del array y el **fin** de la cola *variante* e igual al índice del último elemento, y vamos *insertando* en el **fin** de la cola hasta que lleguemos al límite de la tabla.

Problema: La operación de eliminar un elemento de la cola será ineficiente ya que supondrá el desplazamiento del resto de los elementos de la cola hacia delante.



2. Considerar el **principio** y el **fin** de la cola como *variantes* en el array.



Al poner un elemento en la cola se incrementará **fin** y al sacar un elemento de la cola se incrementará **principio**.

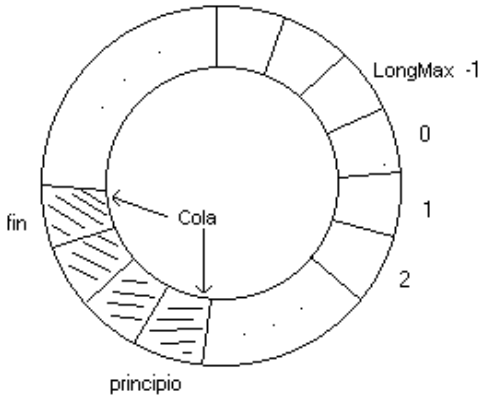
Problema: Tras varias operaciones de inserción sobre la cola llegará un momento en que **fin** llegará a **Longmax-1** y no podremos encolar más elementos pues se habrá alcanzado el final físico del array. Sin embargo es posible que la cola no tenga el número máximo de elementos que pueda tener puesto que aún puede haber espacio libre al comienzo del array.

Al eliminar elementos aumentaremos el principio de la cola, no pudiendo reaprovechar el espacio libre.

3. Para evitar esto se usan las **COLAS CIRCULARES**, un tipo especial de cola que actúa como si el vector fuese una estructura circular (es decir, cuando no existen más casillas libres del vector al final, sigue insertando elementos al principio).

Para *insertar* un elemento se escribe en la posición de **fin** y mueve **fin** una posición hacia adelante (circular). Para *eliminar* un elemento basta con desplazar el índice **principio** una posición hacia adelante, siempre en sentido circular.

Gráficamente*:*



Problema: saber si la cola está llena o está vacía. Es decir, los valores **fin** y **principio** son idénticos en las dos situaciones.

Soluciones:

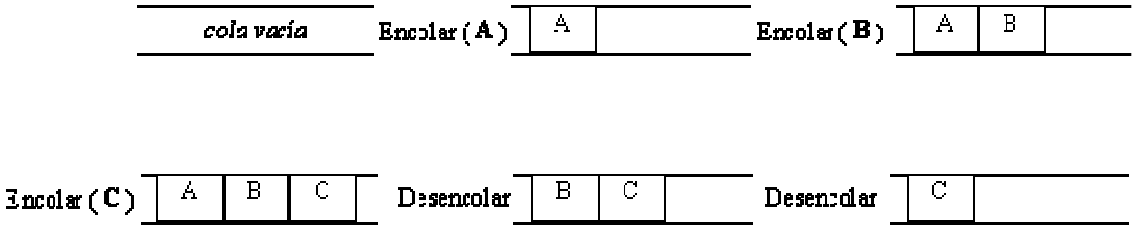
- añadir otro atributo, además de **principio** y **fin**, que cuente los elementos de la cola. Cuando el contador sea 0 la cola estará vacía y cuando el contador sea igual al número de elementos en la cola, la cola estará llena.
- No permitir que el array se llene, es decir, dejar siempre una celda del array circular vacía.

Problema: Al ser una estructura estática siempre tendremos la limitación del tamaño máximo de elementos a almacenar en la cola y la única solución posible será la utilización de una memoria dinámica que veremos en el próximo tema.

- Las operaciones aplicables a una cola son un subconjunto de las operaciones aplicables a una lista, considerando que las inserciones se

realizan por el final de la cola y las eliminaciones tienen lugar siempre por el otro extremo, posición que recibe el nombre de *cabeza, frente o principio de la cola*, como hemos visto.

- Estas operaciones asociadas a las colas y que son las más usuales, reciben generalmente los nombres de ***encolar*** y ***desencolar***.



Implementación

Vamos a implementar una cola con vectores mediante la primera opción de representación vista, almacenando los elementos de la cola en un array, considerando el **principio** *(inicio)* de la cola *fijo* en la primera posición del array y el **fin** *(fin)* de la cola *variante* e igual al índice del último elemento, y vamos *insertando* en el **fin** de la cola hasta que lleguemos al límite de la tabla.

```
// FICHERO TADCola.h
# define MAX 30
# include <iostream>
using namespace std;
class cola {
    int  elementos[MAX]; //elementos de la cola
    int  inicio, fin;//principio y fin de la cola
    public:
```

```
    cola(); // constructor de la clase
    void encolar(int e);
    void desencolar();
    int primero();
    bool esvacia();
    int longitud() ;
    };
```

//FICHERO TADCola.cpp

```
# include ”TADCola.h”
cola::cola(){
inicio=0;
fin=-1;
}
void cola::encolar(int e){
if (fin+1<MAX){
fin++;
elementos[fin]=e;
}
}

int cola:longitud() { return fin+1;}

void cola::desencolar(){
for(int i=inicio;i<fin;i++)
    elementos[i]=elementos[i+1]; //Desplazamiento
fin--;
}
```

```
int cola::primero(){
return(elementos[inicio]);
}

bool cola::esvacia (){
return (fin == -1);
}
```

3.3.4. Aplicaciones

- En informática existen numerosas aplicaciones de las colas:

  - Colas de trabajos a imprimir por una impresora
  - Asignación de tiempo de procesador a los procesos en un sistema multiusuario (sin prioridad)
  - Simulación por computadora de situaciones del mundo real: cajero automático, llamadas en espera,...

- Las colas se suelen utilizar en los sistemas operativos para controlar las Prioridades de los Procesos (Colas de Prioridad), o en las impresoras (cola de impresión donde se almacenan las peticiones de impresión de documentos que van llegando).

Ejemplo – Sistema de Tiempo Compartido

✓ En un sistema de tiempo compartido suele haber un procesador central y una serie de periféricos compartidos: discos, impresoras, etc.

- ✓ Los recursos se comparten por los diferentes usuarios y se utiliza una cola para almacenar las peticiones de los diferentes usuarios que esperan turno de ejecución.

- ✓ El procesador central atiende – normalmente – por riguroso orden de llamada; por tanto, todas las llamadas se almacenan en una cola y los usuarios deben esperar a que esté disponible el recurso que necesitan.

- Existe otra aplicación muy utilizada que se denomina *cola de prioridades*; en ella el procesador central no atiende por riguroso orden de llamada: aquí el procesador atiende prioridades asignadas por el sistema, o bien por el usuario, y sólo dentro de las peticiones de igual prioridad se producirá una cola.

Variante del TAD Cola: TAD Cola de Prioridad.

- El término *cola* sugiere la forma en que esperan ciertas personas u objetos la utilización de un determinado servicio.

- El término *prioridad* sugiere que el servicio no se proporciona únicamente aplicando el concepto de cola, sino que cada elemento tiene asociado una prioridad basada en un criterio objetivo.

El orden en que los elementos son eliminados sigue estas reglas:

– Se elige la lista de elementos que tiene mayor prioridad

– En la lista de mayor prioridad, los elementos se procesan según el orden de llegada

Implementación:

– Mediante una única **lista**

- Cada elemento se almacena en un nodo de la lista. La lista se mantiene ordenada por el campo prioridad.
- La operación insertar un nuevo nodo sigue el siguiente criterio: la posición de inserción es tal que la nueva lista ha de permanecer ordenada. A igualdad de prioridad se añade como último en el grupo de nodos de igual prioridad.

– Mediante una **lista de n colas**

- Se utiliza una cola separada para cada nivel de prioridad.
- Para agrupar todas las colas, se utiliza un array.
- Cada celda del array representa un nivel de prioridad y contiene a la cola correspondiente.