Estructuras de Dato Dinámicas

Los programas están formados por algoritmos y estructuras de datos. Un buen programa es aquel que utiliza los métodos correctos para "almacenar" y "recuperar" esos datos; por eso es necesario tener en cuenta cómo vamos a estructurar los datos a la hora de crear un programa.

Las estructuras de datos se clasifican en estáticas (arreglos y registros) y dinámicas (pilas, colas, listas y árboles). Las primeras reciben su nombre debido a que durante la compilación se les asigna espacio de memoria y permanecen inalterables durante la ejecución de un programa. Las segundas toman importancia cuando no se conoce por adelantado cuánta memoria se requerirá para un programa. Por lo que es conveniente disponer de un método para adquirir posiciones adicionales de memoria a medida que se necesiten durante la ejecución del programa y liberarlas cuando no se necesitan (por ejemplo, New() y Disponse() en Pascar).

Las variables que se crean y están disponibles durante la ejecución de un programa se llaman variables dinámicas. Estas variables se representan con un tipo de dato conocido como puntero. Las variables dinámicas se utilizan para crear estructuras dinámicas de datos que se pueden ampliar o comprimir a medida que se requieran durante la ejecución de un programa. Una estructura de datos dinámica es una colección de elementos denominados nodos de la estructura -normalmente de tipo registro- que son enlazados juntos.

El dinamismo de estas estructuras soluciona el problema de decidir cuál es la cantidad óptima de memoria que debe reservarse para un problema dado. Sin embargo no siempre pueden reemplazarse los arreglos por estructuras dinámicas. Existen casos en que la solución se complica si se utiliza este tipo de estructura en vez de arreglos.

Las estructuras dinámicas de datos se pueden dividir en 2 grandes grupos:

- LINEALES { LISTAS - PILAS - COLAS }
- NO LINEALES { ÁRBOLES - GRAFOS }

TAD LISTA

Una *lista enlazada* es una secuencia de nodos en el que cada nodo está enlazado o conectado con el siguiente.

La lista enlazada es una estructura de datos dinámica cuyos nodos suelen ser normalmente registros y que no tienen un tamaño fijo.

Una lista es una estructura que se utiliza para almacenar información del mismo tipo, con la característica que puede contener un número indeterminado de elementos, y que éstos elementos mantienen un orden explícito ya que cada elemento contiene en sí mismo la dirección del siguiente elemento.

Una lista es una secuencia de 0 a *n* elementos. A la lista de cero elementos llamaremos *lista vacía*.

Especificación formal del TAD Lista

Matemáticamente, una lista es una secuencia de cero o más elementos de un determinado tipo.

$(a_1, a_2, a_3, ..., a_n)$ donde n > = 0,
si n = 0 la lista es vacía.

Los elementos de la lista tienen la propiedad de que están ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que a_i precede a a_{i+1} para i = 1, 2, 3,..., n -1 y que a_i sucede a a_{i-1} para i = 2, 3,..., n.

Para formar el ***Tipo Abstracto de Datos LISTA*** a partir de la noción matemática de lista, se debe definir un conjunto de operaciones con objetos de tipo lista.

La decisión de qué operaciones serán las más utilizadas depende de las características del problema que se va a resolver. También dependerá del tipo de representación elegido para las listas. Este conjunto de operaciones lo definirán en la práctica.

Para representar las listas y las correspondientes implementaciones pueden seguirse dos alternativas:

- Utilización de la estructura estática de *arreglo* para almacenar los nodos de la lista.
- Utilización de estructuras dinámicas, mediante punteros y variables dinámicas.

Implementación del TAD Lista con estructuras estáticas

En la implementación de Listas mediante arreglos, los elementos se guardan en posiciones contiguas del arreglo.

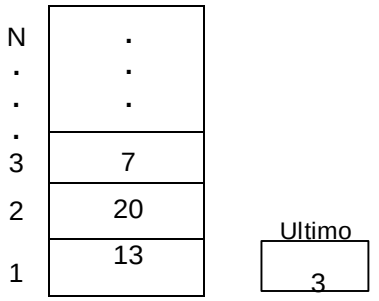
En esta realización se define el tipo *Lista* como un registro de dos campos: el primero, es el arreglo de elementos (tendrá un máximo número de elementos preestablecido); y el segundo, es un entero, que representa la posición que ocupa el último elemento de la lista. Las posiciones que ocupan los nodos en la lista son valores enteros; la posición *i-enésima* es el entero *i*.

Definición de Listas usando arreglos:

Max: ... valor constante
Tipo_elto: ... (Tipo del campo información de cada nodo)

Lista: Registro
Eltos: arreglo [1..Max] de Tipo_elto

Ultimo: entero
FRegistro

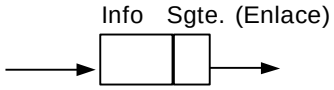


Representación de Listas mediante arreglos

Con ésta realización se facilita el recorrido de la lista y la operación de añadir elementos al final. Sin embargo, la operación de insertar un elemento en una posición intermedia de la lista obliga a desplazar en una posición a todos los elementos que siguen al nuevo elemento a insertar con el objeto de dejar un "hueco".

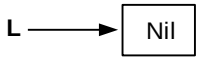
Implementación del TAD Lista mediante variables dinámicas

Este método de representación de listas enlazadas utiliza variables puntero que permiten crear variables dinámicas (Nodos), en las cuales se almacena el *campo de información*, que puede ser un registro, un entero, una cadena de caracteres, etc., y el *campo de enlace* al siguiente nodo de la lista.



Nodo de una Lista enlazada

A una lista enlazada se accede desde un puntero externo que contiene la dirección (referencia) del primer nodo de la lista. En el caso de la lista vacía, el puntero externo apunta a nulo.

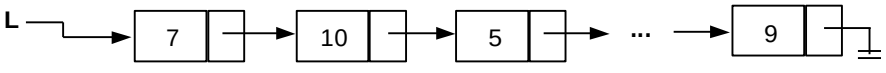


El campo de dirección o enlace del último elemento no debe apuntar a ningún elemento.

Definición de Lista mediante uso de punteros:

Tipo_elto: ... (Tipo del campo información de cada nodo)

Nodo: Registro
Elto: Tipo_elto
Sig: **hNodo**
FRegistro



Representación de Lista utilizando punteros

LISTA: hNodo

Con ésta implementación se ajusta la memoria ocupada al tamaño real de la lista. Ahora, cada vez que sea necesario crear un nuevo nodo se llama al procedimiento que capta la memoria necesaria de la pila de memoria libre, y se enlaza con la lista. El proceso es dinámico, la lista crece o decrece según las necesidades. Con ésta realización se evitan los desplazamientos al insertar y suprimir elementos.

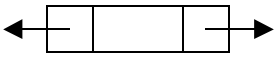
Lo malo de ésta representación es que cada vez que se quiera acceder a un nodo hay que recorrer la lista, a través de los enlaces hasta alcanzar su posición. También hay que tener en cuenta que en éste caso cada nodo ocupa la memoria adicional del campo enlace.

Con los punteros se puede construir toda clase de estructuras enlazadas (pilas, colas, etc.). Estas estructuras son dinámicas, es decir, que el número de elementos puede cambiar durante la ejecución del programa, y así se ajustan a las necesidades de cada instante, cosa que no ocurre con las estructuras estáticas como los arreglos.

Al trabajar con estructuras dinámicas hay que manejar dos clases diferentes de variables: *variables puntero* (direccionan un dato) y *variables de referencia* (son apuntadas).

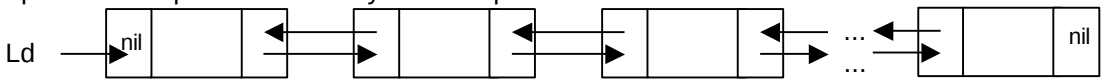
Listas Doblemente Enlazadas

En las listas simplemente enlazadas hay un solo sentido en el recorrido de la Lista. Puede resultar útil el poder avanzar en ambos sentidos, de tal forma que los términos predecesor y sucesor no tengan significado puesto que la lista es completamente simétrica. En cada nodo de éste tipo de listas existe dos enlaces, uno al siguiente nodo y otro al anterior.



Nodo de una Lista doblemente enlazada

Un nodo de una lista doblemente enlazada puede ser considerado como un registro con tres campos: un campo información y dos campos de enlace.



La implementación de listas doblemente enlazadas se puede realizar con estructuras estáticas (arreglos) o con estructuras dinámicas (punteros).

Implementación con punteros

Tipo_elto:… *(Tipo del campo información de cada nodo)*

Nodo: Registro
Elto: Tipo_elto;
Sig, Ant: **hNodo**;
FRegistro

Ld: hNodo

Implementación con arreglos

Max: ... valor constante

Tipo_elto: ... *(Tipo del campo información de cada nodo)*

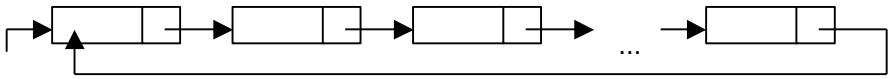
Ld: Registro
Eltos: arreglo [1..Max] de Tipo_elto;
Ultimo: entero;
FRegistro

Las operaciones que se pueden definir en este TAD pueden ser las mismas que para el TAD Listas, o pueden variar, dependiendo de para que se las utilice.

Lista Circular mediante variables dinámicas

En las listas lineales siempre hay un último nodo que tiene el campo de enlace a nil. Una lista circular, en cambio, por su naturaleza no tiene primero ni último nodo. Sin embargo, resulta útil establecer un primer y un último nodo; una convención es la de considerar que el puntero externo de la lista circular referencia al último nodo, y que el nodo siguiente sea el primer nodo de la lista.

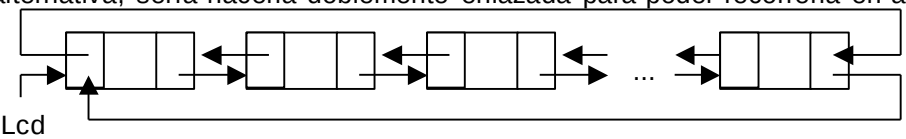
Todos los recorridos de la lista circular se hacen tomando como referencia el considerado último nodo.



Lc

Al igual que con las listas y las listas doblemente enlazadas, sobre una lista circular se pueden especificar una serie similar de operaciones, y así formar el TAD Lista Circular.

Otra alternativa, sería hacerla doblemente enlazada para poder recorrerla en ambos sentidos.



Lcd

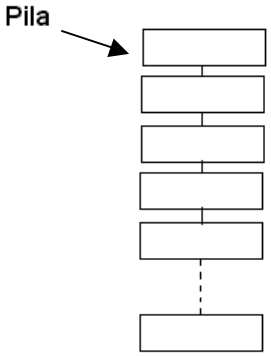
TAD Pila

Una pila es una estructura de datos en la que todas las inserciones y eliminaciones de elementos se realizan por un extremo denominado tope o cima de la pila.

La implementación de una pila se puede realizar mediante arreglos o con punteros. El inconveniente de la realización de una pila mediante arreglos es que su tamaño máximo se debe especificar en tiempo de compilación. Debido a esto, la implementación de pilas se realiza frecuentemente mediante punteros.

Especificación formal del TAD Pila

Una pila es una lista ordenada de elementos en el que todas las inserciones y eliminaciones se realizan por un mismo extremo de la lista. En una pila el último elemento añadido es el primero en salir de la pila. Por esa razón a las pilas se las denomina también listas LIFO (*last input, firts output*; último en entrar, primero en salir).



Las pilas crecen y decrecen dinámicamente, es una estructura de datos dinámica. En cuanto a la representación de una pila, existen varias alternativas. Una primera representación podría ser mediante la estructura estática de un arreglo, de la cual ya indicamos la desventaja.

Una segunda alternativa sería usando listas enlazadas. Las listas crecen y decrecen dinámicamente, al igual que ocurre con una pila. En esta representación dinámica se utilizan punteros y variables dinámicas. Las operaciones se definirán en la práctica.

Implementación del TAD Pila con arreglos

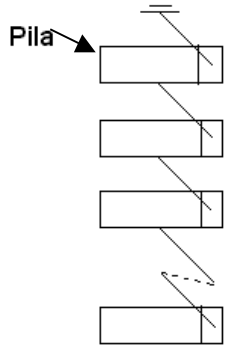
Un arreglo constituye el depósito de los elementos de la pila. El rango del arreglo debe ser lo suficientemente amplio para poder contener el máximo previsto de elementos de la pila. Un extremo del arreglo se considera el *fondo* de la pila, que permanecerá fijo. La parte superior de la pila, *tope* o *cima*, estará cambiando dinámicamente durante la ejecución del programa. Además del arreglo, una variable entera nos sirve para tener en todo momento el índice del arreglo que contiene el elemento *tope*. Las declaraciones, procedimientos y funciones para representar el TAD Pila, forman parte de la unidad pila.

```
Max_elem= ...(Dependerá de cada realización)
Tipo_elem=... (Tipo de los elementos de la pila)
Pila= Registro
        Elem= arreglo [1..Max_elem] de Tipo_elem;
        Tope= entero;
FinRegistro
```

Implementación del TAD Pila mediante variables dinámicas

La implementación dinámica de una pila se hace almacenando los elementos como nodos de una lista enlazada, con la particularidad de que siempre que se quiera meter o sacar un elemento se hará por el mismo extremo.

Esta realización tiene la ventaja de que el tamaño se ajusta exactamente a los elementos de la pila. Sin embargo, para cada elemento se necesita más memoria, ya que hay que guardar el campo de enlace. En la realización con arreglos hay que establecer un máximo de posibles elementos, aunque el acceso es más rápido, ya que se hace con una variable subindicada.




```
Tipo_elem=... (Tipo de los elementos de la pila)
Pila= ĥNodo;
Nodo= Registro
        Elem= Tipo_elem;
        Sig= ĥNodo;
FinRegistro
```

Aplicaciones de Pilas

La Pila son utilizadas para solucionar una amplia variedad de problemas. Se utilizan en compiladores, en sistemas operativos y en programas de aplicación. Los casos más representativos son:

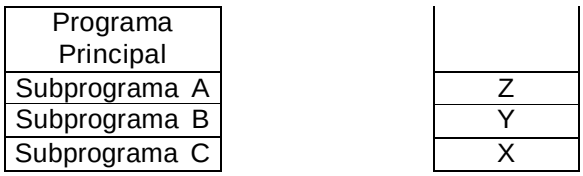
- Llamada a subprogramas
- Recursión
- Tratamiento de expresiones aritméticas
- Ordenación

Llamada a subprogramas o procedimientos

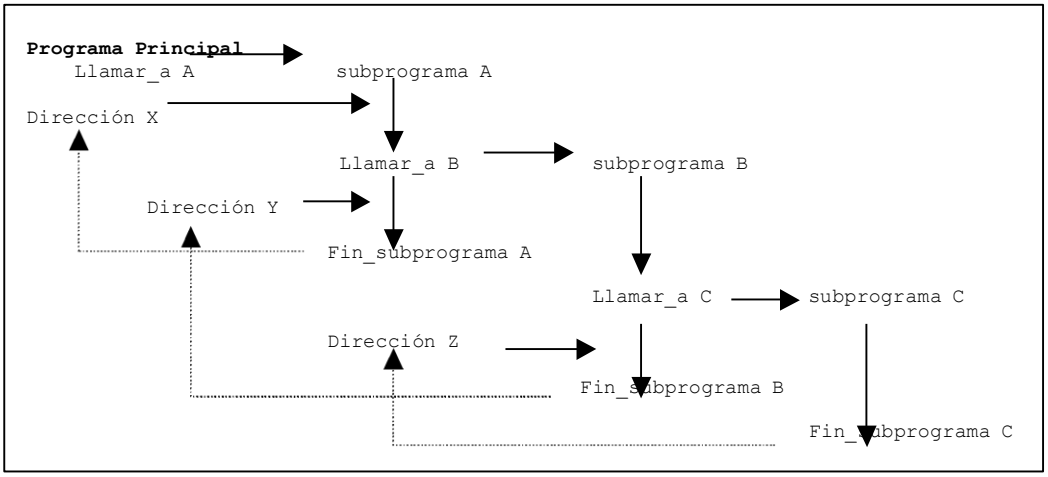
Cuando dentro de un programa se realizan llamadas a subprogramas, el programa principal debe recordar el lugar dónde se hizo la llamada, de modo que pueda retornar allí cuando el subprograma se haya terminado de ejecutar.

Supongamos que tenemos tres subprogramas, llamados A, B, C, y supongamos también que A invoca a B, y B invoca a C. Entonces B no terminará su trabajo hasta que C haya terminado y devuelto su control a B. De modo similar, A es el primero que arranca su ejecución, pero es el último que la termina, tras la terminación y retorno de B.

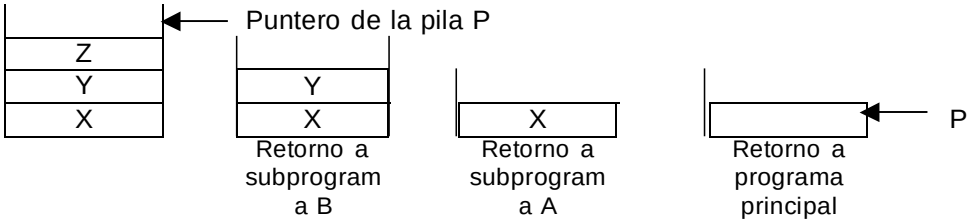
Esta operación se consigue disponiendo las direcciones de retorno en una pila.



Pila de direcciones de retorno



Cuando un subprograma termina, debe retornar a la dirección siguiente a la instrucción que le llamó (llamar_a). Cada vez que se invoca un subprograma, la dirección siguiente (x, y o z) se introduce en la pila. El vaciado o limpieza de la pila se realiza por los sucesivos retornos; decrementándose el puntero de pila queda libre siempre apuntando a la siguiente dirección de retorno.



Además de guardar las direcciones antes mencionadas, cuando se tiene un programa que llama a un subprograma, internamente se usan pilas para guardar el estado de las variables del programa en el momento que se hace la llamada. Así, cuando termina la ejecución del subprograma, los valores almacenados en la pila pueden recuperarse para continuar con la ejecución del programa en el punto en el que fue interrumpido.

Implementación de procedimientos recursivos mediante pilas

Un procedimiento o función contiene tanto variables locales como argumentos ficticios o parámetros. A través de los argumentos se transmiten datos en las llamadas a los subprogramas, o bien se devuelven valores al programa o subprograma invocante.

Además, el subprograma debe guardar la dirección de retorno al programa que realiza la llamada. En el momento en que termina la ejecución de un subprograma el control pasa a la dirección guardada.

Ahora el subprograma es recursivo, entonces además de la dirección de retorno, los valores actuales de las variables locales y argumentos deben de guardarse ya que se usarán de nuevo cuando el subprograma se reactive.

Supongamos que se ha llamado al subprograma P, que tiene llamadas a sí mismo, es decir, es recursivo. El funcionamiento del programa recursivo P será:

- Se crea una pila para cada argumento.
- Se crea una pila para cada variable local.
- Se crea una pila para almacenar la dirección de retorno.

Cada vez que se hace una llamada recursiva a P, los valores actuales de los argumentos y de las variables locales se meten en sus pilas para ser procesadas posteriormente. Asimismo, cada vez que hay un retorno a P procedente de una llamada recursiva anterior, se restauran los valores de variables locales y argumentos de la simas de la pilas.

Para la obtención de la dirección de retorno vamos a suponer que el procedimiento P contiene una llamada recursiva en la sentencia N. Entonces guarda en otra pila la dirección de retorno, que será la sentencia siguiente, la N+1. De tal forma que cuando el nivel de ejecución del procedimiento P actual termine, o sea alcance la sentencia *end* final, usará dicha pila de direcciones para volver al nuevo nivel de ejecución. De esta forma cuando la pila de direcciones se queda vacía volverá al programa que llamó al subprograma recursivo P.

El problema de las Torres de Hanoi resuelto sin recursividad

El problema de las Torres de Hanoi, cuya solución recursiva se ha descrito en la práctica, se puede resolver de forma no recursiva siguiendo esta estrategia.
Los argumentos conocidos son:

```
N: número de discos  A: Varilla A    B: Varilla B    C: Varilla C
```

Para cada uno de ellos se define una pila: `PilaN, PilaA, PilaB, PilaC`

Además, la pila de direcciones: `PilaDir`

Las etapas a considerar en la construcción del algoritmo HANOI son:

1. Crear Pilas.

```
Anular (PilaN); Anular (PilaA); Anular (PilaB); Anular (PilaC);
Anular (PilaDir);
```

2. Condición de fin.

```
Si N=1 entonces
       Escribir ("Mover disco", N, " de ", A, " a ", C);
       Ir a 5;
Fin-si.
```

3. Llamada recursiva.

   a) Guardar en pilas:

```
METE (N, PilaN);
METE (A, PilaA);
METE (B, PilaB);
METE (C, PilaC);
METE (<paso 3>, PilaDir);
```

b) Actualizar los argumentos:

```
N=N-1;
A=A;
Aux=B;
B=C;
C=Aux;
```
} *Intercambiar B y C*

c) Volver al paso 1.

4. Progreso.

```
Escribir ("Mover disco", N, " de ", A, " a ", C);
```

5. Segunda llamada recursiva.

a) Guardar en pilas:

```
METE (N, PilaN);
METE (A, PilaA);
METE (B, PilaB);
METE (C, PilaC);
METE (<paso 5>, PilaDir);
```

b) Actualizar los argumentos:

```
N=N-1;
C=C;
Aux=B;
B=A;
A=Aux;
```
} *Intercambiar B y A*

c) Volver al paso 1.

6. Retorno de las llamadas recursivas.

a) Si las pilas están vacías entonces fin de algoritmo: volver a la rutina llamadora.

b) Sacar de las pilas:

```
N= SACA (PilaN);
A= SACA (PilaA);
B= SACA (PilaB);
C= SACA (PilaC);
Dir= SACA (PilaDir);
```

c) Ir al paso que está guardado en `Dir`.

Evaluación de Expresiones Aritméticas mediante Pilas

Una de las aplicaciones más típicas del TAD Pila es almacenar los caracteres de una expresión aritmética con el fin de evaluar el valor numérico de dicha expresión.

Una expresión aritmética está formada por operandos y operadores.

La forma habitual de escribir una expresión aritmética (el operador en medio de los operandos) se conoce como *notación* ***infija***.

Conviene recordar que las operaciones tienen distintos niveles de precedencia:

Paréntesis	:	( )	Nivel mayor de prioridad
Potencia	:	^	
Multiplicación/División	:	* , /	
Suma/Resta	:	+ , -	Nivel menor de prioridad

También suponemos que a igualdad de precedencia son evaluados de izquierda a derecha.

Notación Prefija (Polaco) y Postfija (Polaco Inversa)

La forma habitual de escribir operaciones aritméticas es situar el operador entre sus dos operandos mediante la citada notación *infija*. Esta forma de notación obliga en muchas ocasiones a utilizar paréntesis para indicar el orden de evaluación.

A * B / (A + C) ó A * B / A + C

Estas expresiones por más que se parecen, llevan a resultados diferentes.

La notación en la que el operador se coloca delante de los operandos se conoce como *notación* ***prefija*** o ***Polaca*** (en honor al matemático polaco que la estudió).

A * B / (A + C) (infija) ⟹ / * A B + A C (polaca)

A * B / A + C (infija) ⟹ + / * A B A C (polaca)

Podemos observar que con la notación Polaca no es necesario el uso de paréntesis al escribir la expresión. La propiedad fundamental de la notación polaca es que el orden en el que se van a realizar las operaciones está determinado por las posiciones de los operadores y los operandos en la expresión.

Otra forma de escribir las operaciones es mediante la *notación* ***prostfija*** o ***polaco inversa***, que coloca el operador a continuación de sus operandos.

A * B / (A + C) (infija) ⟹ A B * A C + / (polaca inversa)

A * B / A + C (infija) ⟹ A B * A / C + (polaca inversa)

Algoritmos para la evaluación de una expresión aritmética

A la hora de evaluar una expresión aritmética escrita, normalmente, en notación infija la computadora sigue dos pasos:

1º Transforma la expresión infija en postfija

2º Evalúa la expresión en postfija

En el algoritmo para resolver cada paso es fundamental la utilización de pilas. Se parte de una expresión en notación infija que tiene operadores, operandos y puede o no tener paréntesis.

La transformación se realiza utilizando una pila en la que se almacenan los operadores y los paréntesis izquierdos. La expresión se va leyendo carácter a carácter, los operandos pasan directamente a formar parte de la expresión postfija.

Los operadores se meten en la pila siempre que ésta esté vacía, o bien siempre que tengan mayor prioridad que el operador *tope* de la pila(o bien igual si es la máxima prioridad). Si la prioridad es menor o igual se saca el elemento *tope* de la pila y se vuelve a hacer la comparación con el nuevo elemento *tope*.

Los paréntesis izquierdo siempre se meten en la pila con la mínima prioridad. Cuando se lee un paréntesis derecho, hay que sacar todos los operadores de la pila pasando a formar parte de la expresión postfija, hasta llegar a un paréntesis izquierdo, el cual simplemente se elimina, ya que

los paréntesis no forman parte de la expresión postfija. El algoritmo termina cuando no hay más items en la expresión y la pila está vacía.

Desarrollo de un ejemplo de pasar una expresión infija a postfija

Para mejor comprensión desarrollaremos el siguiente ejemplo:
Sea la expresión infija A * (B + C – (D / E ^ F) – G) – H, la expresión en postfija se va a ir formando con la siguiente secuencia:

Lista de Expresión Infija	Estado de la Pila	Acción	Lista de Expresión Postfija
A*(B+C–(D/E^F)–G)–H	Pila vacía	Inicialmente.	Lista vacía
*(B+C–(D/E^F)–G)–H		Carácter A a la expresión	A
(B+C–(D/E^F)–G)–H	*	Carácter * a la pila	
B+C–(D/E^F)–G)–H	(	Carácter (a la pila	
+C–(D/E^F)–G)–H	*	Carácter B a la expresión	AB
C–(D/E^F)–G)–H	+	Carácter + a la pila	
–(D/E^F)–G)–H	(	Carácter C a la expresión	ABC
(D/E^F)–G)–H	-	El nuevo carácter leído es -, que tiene igual prioridad que el elemento del *tope* de la pila +, por lo que se agrega el + a la expresión y el – a la pila.	ABC+
D/E^F)–G)–H	(	Carácter (a la pila	
/E^F)–G)–H	-	Carácter D a la expresión	ABC+D
E^F)–G)–H	/	Carácter / a la pila	
^F)–G)–H	(	Carácter E a la expresión	ABC+DE
F)–G)–H	^	Carácter ^ a la pila	
)–G)–H	/	Carácter F a la expresión	ABC+DEF
–G)–H	-	Carácter) provoca vaciar la pila hasta encontrar un (, al cual lo elimina, y a los elementos sacados de la pila los va agregando a la expresión en el orden en el que los lee	ABC+DEF^/
G)–H	-	Carácter – a la pila y se extrae y se agrega a la expresión el – del *tope* por tener igual prioridad	ABC+DEF^/-
)–H	-	Carácter G a la expresión	ABC+DEF^/-G
–H	*	Carácter) provoca vaciar la pila hasta encontrar un (, al cual lo elimina, y a los elementos sacados de la pila los va agregando a la expresión en el orden en el que los lee	ABC+DEF^/-G-
H	-	Carácter -, se saca de la pila * y se pone -	ABC+DEF^/-G-*
Lista vacía	-	Carácter H a la expresión	ABC+DEF^/-G-*H
Lista vacía	Pila Vacía	Lista de entrada vacía, por lo cual se vacía la pila y se va poniendo en la lista.	ABC+DEF^/-G-*H-

En el ejemplo desarrollado se observa que el paréntesis izquierdo tiene la máxima prioridad fuera de la pila, es decir, en la notación infija; sin embargo, cuando está adentro de la pila la prioridad es mínima. De igual forma, para tratar el hecho de que varios operadores de potenciación son evaluados de derecha a izquierda, este operador tendrá mayor prioridad cuando todavía no esté metido en la pila, que el mismo pero metido en la pila.

Las prioridades son determinadas según esta tabla:

Operador	Prioridad dentro de la pila	Prioridad fuera de la pila
^	3	4
*, /	2	2
+, -	1	1
(	0	5

Obsérvese que no se trata de paréntesis derecho ya que este provoca sacar operadores de la pila hasta encontrar un paréntesis izquierdo, el cual se deshecha.

Algoritmo para transformar una expresión en notación infija a notación postfija

El algoritmo de paso de notación infija a postfija sería:

1. Obtener caracteres de la expresión y repetir los pasos del 2 al 4 para cada carácter.
2. Si es operando, pasarlo a la expresión postfija. Volver al paso 1.
3. Si es operador:
   3.1 Si la pila está vacía, meterlo en la pila. Volver al paso 1.
   3.2 Si la pila no está vacía:
      3.2.1 Si la prioridad del operador leído es mayor que la prioridad del operador *tope* de la pila, meterlo en la pila. Volver al paso 1.
      3.2.2 Si la prioridad del operador leído es menor o igual que la prioridad del operador del *tope* de la pila, sacar el elemento *tope* de la pila y pasarlo a la expresión postfija, y volver al paso 3.
4. Si es paréntesis derecho:
   4.1 Sacar el elemento *tope* de la pila y pasarlo a la expresión postfija.
   4.2 Si el nuevo *tope* es el paréntesis izquierdo, suprimir el elemento *tope*.
   4.3 Si el elemento *tope* no es el paréntesis izquierdo, volver al paso 4.1.
   4.4 Volver al paso 1.
5. Si quedan elementos en la pila pasarlos a la expresión postfija.
6. Fin del algoritmo.

Evaluación de la expresión en postfija

En una lista o vector ha sido almacenada la expresión ya en notación postfija. El algoritmo de evaluación utiliza una pila de operandos (números). Al describir el algoritmo PF es la lista que contiene la expresión en postfija. El número de elementos que contiene la expresión es n=LONGITUD (PF).

1. Examinar la lista PF desde el primer elemento hasta el elemento n. Repetir los pasos 2 y 3 para cada elemento del la lista PF.
2. Si el elemento es un operando meterlo en la pila.
3. Si el elemento es un operador (lo designamos &), entonces:
   3.1 Sacar los dos elementos superiores de la pila (los llamamos X e Y respectivamente).
   3.2 Evaluar Y&X, y el resultado Z ponerlo en la pila.
   3.3 Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento *tope* de la pila.
5. Fin del algoritmo.

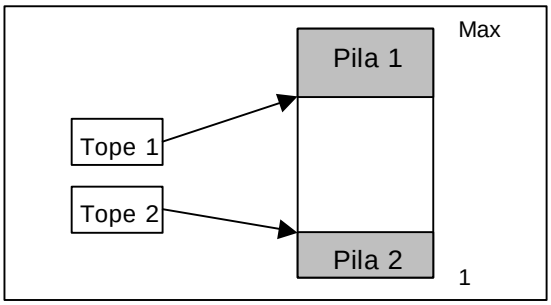
Ordenamiento

Otra aplicación de las pilas puede verse en el método de ordenación rápida o quicksort. Es posible aumentar la velocidad de ejecución del algoritmo de ordenación rápida eliminando las llamadas recursivas. La recursión es un instrumento muy poderoso, pero la eficiencia de ejecución es un factor muy importante en un proceso de ordenamiento que es necesario cuidar y administrar muy bien. Estas llamadas recursivas pueden sustituirse utilizando pilas, dando lugar a la iteratividad.

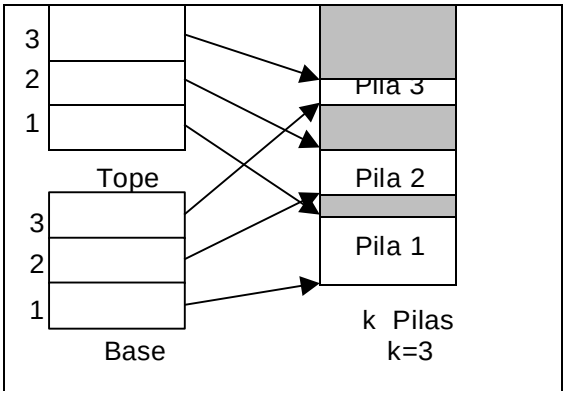
Ejemplo práctico de uso de pilas

Plantéese el siguiente ejercicio práctico de pilas:

Es posible guardar dos pilas en un solo arreglo, si una de ellas crece desde la posición 1 del arreglo y la otra lo hace desde la última posición. No se deben reportar desbordamientos en ninguna pila, al menos que no existan posiciones libres en el arreglo. Generar el TAD para dicha estructura.



Se pueden almacenar k pilas en un solo arreglo, como se muestra en el gráfico. Implemente rutinas para PONER y QUITAR elementos de cada pila. Considere que si la inserción en pila i hace que TOPE(i) iguale a BASE(i-1), antes de efectuarla hay que desplazar todas las pilas, de modo que quede una separación del tamaño apropiado entre cada par de pilas adyacentes. Por ejemplo, es posible hacer que las separaciones entre pilas sean todas iguales, o que la separación que queda encima de la pila i sea proporcional al tamaño actual de la pila i (en el supuesto de que las pilas más grandes tienen



mayor probabilidad de crecer antes, y se desea postergar la siguiente reorganización el mayor tiempo posible). Crear para ello un procedimiento REORGANIZA, al que se llama si las pilas colisionan.

TAD COLA

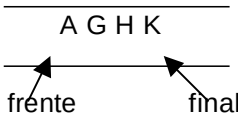
Una **cola** es una lista en la que todas las inserciones a la lista se realizan por un extremo, y todas las eliminaciones o supresiones de la lista se realizan por el otro extremo. Las colas se llaman también estructuras **FIFO** (*first-in, firs-out*; primero en entrar, primero en salir).

Las aplicaciones de las colas son numerosas en el mundo de la computación: colas de impresión, acceso a almacenamiento en disco, sistemas de tiempo compartido, uso de UCP.

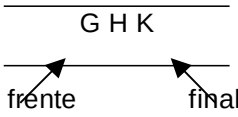
Especificación formal del TAD Cola

Una cola es una lista ordenada de elementos en la que las eliminaciones se realizan en un solo extremo, llamado *frente* o *principio de la cola*, y los nuevos elementos son añadidos por el otro extremo, llamado *fondo* o *final de la cola*.

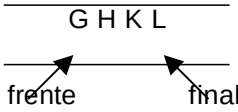
a) Estructura original



b) Estructura después de eliminar un elemento



c) Estructura después de agregar un elemento



Estructura de datos tipo cola

En esta estructura de datos el primer elemento que entra es el primero en salir. Las operaciones del TAD Cola se definirán en la práctica. Al igual que las pilas, la implementación de colas puede hacerse utilizando un arreglo o una lista enlazada y dos punteros a los extremos.

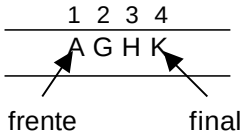
Implementación del TAD Cola con arreglos lineales

La forma más sencilla de representación de una cola es mediante arreglos lineales. Son necesarios dos índices para referenciar al elemento frente y al elemento final.

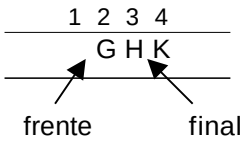
Definición de Cola usando arreglos lineales:

```
Tipo_elto: ... (Tipo del campo información de cada nodo)
Cola: Registro
    eltos: arreglo [1..N ] de Tipo_elto;
    frente: entero;
    final: entero;
Fregistro;
```

Un arreglo es una estructura estática, y por lo tanto tiene dimensiones finitas; por el contrario, una cola puede crecer y crecer sin límite, y en consecuencia se puede presentar la posibilidad de que ocurra un desbordamiento. Por esta razón se necesitaría incorporar al TAD la operación de verificación de si la cola está llena.
La operación de añadir un nuevo elemento a la cola comienza a partir de la posición 1 del arreglo.



Supongamos que la cola se encuentra en la situación anterior: el frente está fijo y el final de la cola es el que se mueve al añadir nuevos elementos. Si ahora se quita un elemento (evidentemente por el frente) la cola queda así:



Se ha dejado un hueco no utilizable, por lo que se desaprovecha memoria.

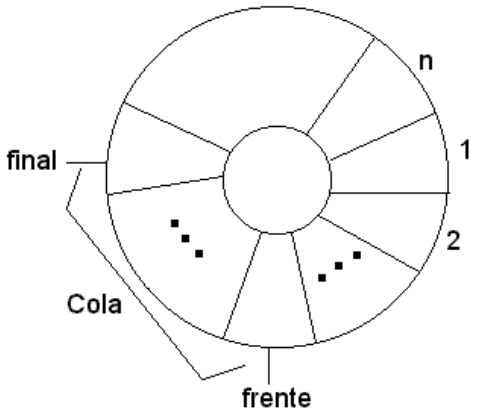
Una alternativa a esta situación es mantener fijo el frente de la cola al comienzo del arreglo: este hecho supone mover todos los elementos de la cola una posición cada vez que se quiera retirar un elemento de la cola.
Estos problemas se resuelven con los arreglos circulares.

Implementación del TAD Cola con arreglos circulares

La implementación de una cola mediante arreglos lineales es poco eficiente. Cada operación de supresión de un elemento supone mover todos los elementos restantes de la cola.

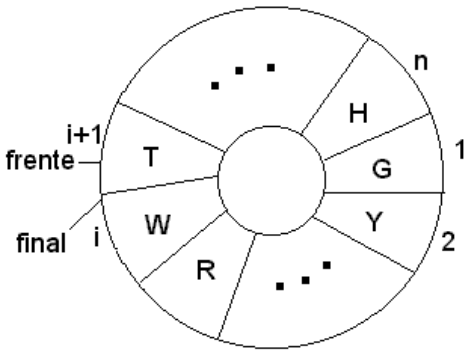
Para evitar este gasto, imaginaremos un arreglo como una estructura circular en la que al último elemento le sigue el primero.

Esta representación implica que aún estando ocupado el último elemento del arreglo, puede añadirse uno nuevo detrás de él, ocupando la primera posición del arreglo.



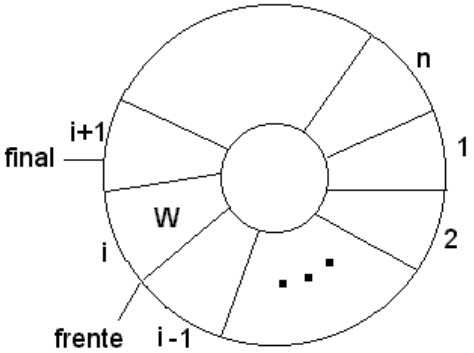
Arreglos circulares

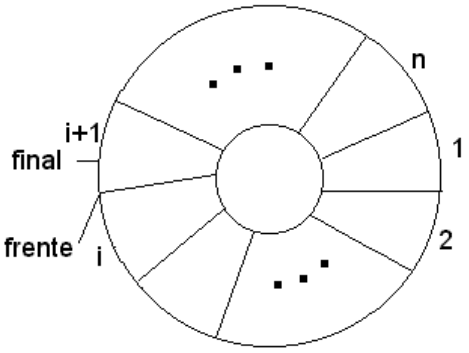
Para *añadir un elemento* en la cola, se mueve el índice *final* una posición en el sentido de las manecillas del reloj, y se asigna el elemento. Para *suprimir un elemento* es suficiente con mover el índice *frente* una posición en el sentido del avance de las manecillas del reloj. De esta manera, la cola se mueve en un mismo sentido, tanto si se realizan inserciones o supresiones de elementos. En este caso, la condición de cola vacía (*frente* = siguiente (*final*)) va a coincidir con una cola que ocupa el círculo completo, una cola que llene todo el arreglo.
Para resolver el problema, una primera tentativa sería considerar que el elemento *final* referencie una posición adelantada a la que realmente ocupa el elemento, en el sentido del avance del reloj. Teniendo presente esta consideración cuando la cola estuviera llena, el índice siguiente a *final* sería *frente*.



Cola llena en un arreglo circular

Consideremos ahora el caso en el que queda un solo elemento en la cola. Si en estas condiciones se suprime el elemento, la cola queda vacía, el elemento *frente* avanza una posición en el sentido de las agujas del reloj y va a referenciar a la misma posición que el siguiente al puntero *final*. Es decir, está exactamente en la misma posición relativa que ocuparía si la cola estuviera llena.





Supresión del último elemento en un arreglo circular. *Cola Vacía en un arreglo circular*

Una solución a este problema es sacrificar una posición del arreglo y dejar que *final* referencie a la posición realmente ocupada por el último elemento añadido. Si el arreglo tiene N posiciones, no se debe dejar que la cola crezca más que N-1.

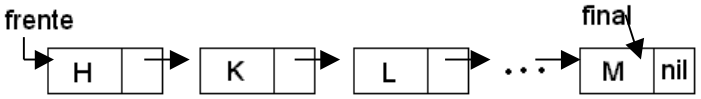
Teniendo en cuenta todo lo expuesto, la *definición de Cola usando arreglos circulares* sería:

Tipo_elto: ... (Tipo del campo información de cada nodo)

```
Cola: Registro
    eltos: arreglo [1..N ] de Tipo_elto;
    frente: entero;
    final: entero;
Fregistro;
```

Implementación del TAD Cola con listas enlazadas

Como ocurre con toda representación estática, una de las principales desventajas es que hay que prever un máximo de elementos, y de ese máximo no podemos pasarnos. La realización de una cola mediante una lista enlazada permite ajustarse exactamente al número de elementos de la cola.
Esta implementación con listas enlazadas utiliza dos punteros que acceden a la lista, el puntero *frente* y el puntero *final.*



Cola implementada con listas enlazadas

El puntero *frente* referencia al primer elemento que va a ser retirado. El puntero *final* referencia al último nodo que fue añadido. En esta representación no tiene sentido la operación que indica si la cola está llena. Al ser una estructura dinámica crece y decrece según las necesidades.

Las *definición de Cola usando listas enlazadas* serían:

Tipo_elto: ... (Tipo del campo información de cada nodo)

```
Nodo: Registro
    Elto: Tipo_elto;
    Sig: hNodo;
Fregistro;
```

```
Cola: Registro
    frente: hNodo;
    final: hNodo;
Fregistro;
```

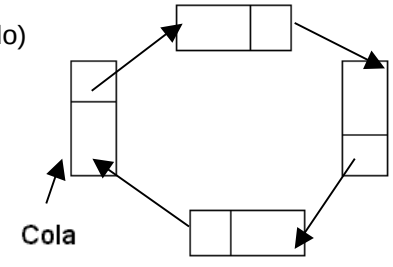
Implementación del TAD Cola con listas circulares

Esta representación es una variante de la realización con listas enlazadas. Al realizar una lista circular se estableció, por conveniencia, que el puntero al acceso a la lista referenciaba al último nodo, y que el nodo siguiente se considera el primero. Según este convenio, y teniendo en cuenta la definición de cola, las inserciones de nuevos elementos de la cola serán realizados por el nodo referenciado por el puntero *cola*, y la eliminación por el nodo siguiente.
Las *declaraciones de tipo* para esta representación es:

Tipo_elto: ... (Tipo del campo información de cada nodo)

Cola: h**Nodo;**

```
Nodo: Registro
    Elto: Tipo_elto;
    Sig: hNodo;
Fregistro;
```



Cola implementada con una lista circular con punteros.

Colas de Prioridad

El término *cola* sugiere la forma en que esperan ciertas personas u objetos la utilización de un determinado servicio. Por otro lado, el término *prioridad* sugiere que el servicio no se proporciona únicamente aplicando el concepto de cola (primero en llegar, primero en salir) sino que cada persona tiene asociada una prioridad basada en un criterio objetivo.
Un ejemplo típico de organización formando colas de prioridad, es el sistema de tiempo compartido necesario para mantener un conjunto de procesos que esperan servicio para trabajar. Los diseñadores de estos sistemas asignan cierta prioridad a cada proceso.
El orden en que los elementos son procesados y por lo tanto eliminados sigue estas reglas:

1. *Se elige la lista de elementos que tiene la mayor prioridad.*
2. *En la lista de mayor prioridad, los elementos se procesan según el orden de llegada; en definitiva, según la organización de una cola.*

Las colas de prioridad pueden implementarse de dos formas: mediante una única lista o mediante una lista de colas.

Implementación de Colas de Prioridad

Para realizar la implementación, se define en primer lugar el tipo de datos que representa un proceso.

```
Tipo_identif: ... (tipo del identificador del proceso)
Tipo_proceso: Registro
        Idt: Tipo_identif;
        Prioridad: entero;
Fin_registro
```

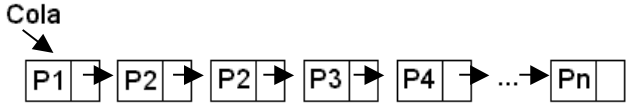
Implementación mediante una única lista

Cada proceso forma un nodo de la lista enlazada. La lista se mantiene ordenada por el campo prioridad.

La operación de añadir un nuevo nodo hay que hacerla siguiendo este 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. De esta manera la lista queda organizada de tal forma que un nodo X precede a un nodo Y si:

1. *Prioridad(X) > Prioridad(Y)*
2. *Ambos tienen la misma prioridad, pero X se añadió antes que Y*



Cola de prioridad con una lista enlazada

Los números de prioridad tienen el significado habitual: *a menor número mayor prioridad.*

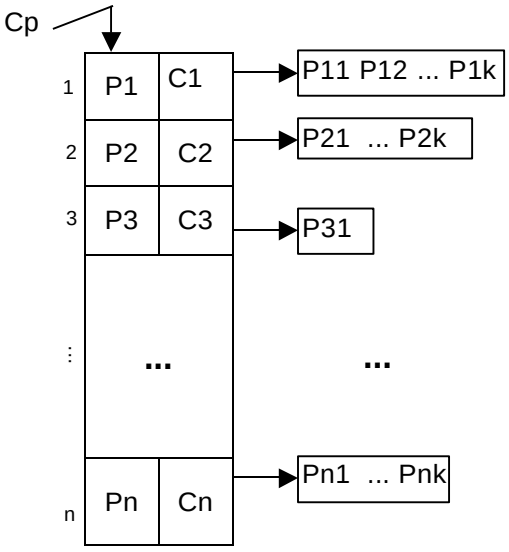
Esta realización presenta como principal ventaja que se determina el siguiente nodo a procesar de forma inmediata: siempre será el primero de la lista. Sin embargo, añadir un nuevo elemento supone encontrar la posición de inserción correcta dentro de la lista, según el criterio antes expuesto.

Implementación mediante una lista de n colas

Se utiliza una cola separada para cada nivel de prioridad. Cada cola puede representarse con un arreglo circular, mediante una lista enlazada, o bien mediante una lista circular, como se prefiera; en cualquier caso con su *frente* s su *final*. Para agrupar todas las colas se utiliza un arreglo de registros. Cada registro representa un nivel de prioridad, y tiene el *frente* y *final* de la cola correspondiente. La razón para utilizar un arreglo como lista radica en que los niveles de prioridad son establecidos de antemano, por lo cual no se requiere una lista dinámica.
La definición de los tipos de datos para esta realización son:

```
Max_prior: ... (máximo número de prioridades previsto)
Tipo_identif: ... (tipo del identificador del proceso)
Tipo_proceso: Registro
        Idt: Tipo_identif;
        Prioridad: entero;
Fin_registro
Nodo: Registro
        Elto: Tipo_proceso;
        Sig: hNodo;
Fregistro;
Cola: Registro
        Numprior: entero;
        frente, final: hNodo;
Fregistro;

Cp: arreglo [1...Max_prior] de Cola;
```



Cola de prioridad con lista de n Colas

El tipo Cp es el que define la lista de colas y cada cola representa una prioridad.
Las acciones más importantes al manejar una cola de prioridades son las de *añadir un nuevo elemento*, con una determinada prioridad al sistema, y la acción de *retirar un elemento para su procesamiento*.

Estas acciones se expresan en forma algorítmica en la realización de n colas de la siguiente manera:

Algoritmo para añadir un nuevo elemento:	***Algoritmo para retirar un elemento:***
Añade un elemento P que tiene prioridad m. 1. Buscar el índice en la tabla correspondiente a la prioridad m. 2. Si existe y es K, poner el elemento P como *final* de la cola de índice K. *3.* Si no existe, crear nueva cola y poner en ella el elemento P. *4.* Salir	Retira el elemento de frente de la cola que tiene máxima prioridad. 1. Buscar el índice de la cola de mayor prioridad. Cola K. 2. Retira y procesa el elemento frente de la Cola K. 3. Salir.

Colas de Simulación

El objetivo de un programa de simulación es modelizar algún *sistema físico* (por ej., un sistema meteorológico, un banco, una línea de montaje, etc.) de tal forma que podamos analizar y predecir su comportamiento. El sistema físico puede ser considerado como una colección de elementos interdependientes, o entidades, que operan e interactúan para conseguir cierta tarea. Se utiliza una colección de ***variables de estado*** para representar el estado del sistema físico. Bastante a menudo las variables de estado de un sistema se definen para representar los estados de las entidades involucradas.

> *Por ejemplo: en la simulación del tráfico aéreo de un aeropuerto, un avión dado sería una entidad del sistema que se está considerando, y su estado podría estar definido como en el aire, en tierra, aterrizando o despegando.*

Habitualmente, deseamos modelizar un sistema estocásticamente; en este caso, a una o más de las variables de estado se le asignan valores de acuerdo con una distribución de probabilidades.

> *En nuestro ejemplo: podríamos modelizar la probabilidad de que un avión sea incapaz de despegar debido a una avería de acuerdo a una distribución de probabilidad que se ajuste estrictamente a los datos observados.*

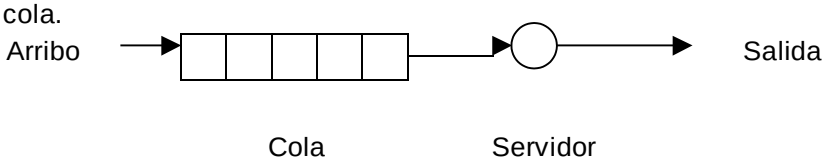
Además, los cambios en el estado del sistema son normalmente medidos en función del tiempo. En una ***simulación de sistema continua*** el estado del sistema está continuamente evolucionando con el correr del tiempo. Por el contrario, en una ***simulación de sistemas discreta*** a las variables de estado sólo se les permite cambiar en instantes discretos de tiempo.

Es importante distinguir entre el *tiempo simulado*, que es el tiempo en el sistema físico, y el tiempo de cómputo requerido para ejecutar el programa. Para sistemas complejos grandes pueden ser necesarias horas de tiempo de cómputo para simular unos pocos segundos de tiempo en el sistema físico. En otras situaciones, el tiempo de cómputo debería ser mucho menor que el tiempo simulado.

La principal dificultad que surge al implementar una simulación informática supone salvar la distancia conceptual que existe entre el sistema físico y su representación informática. Esto implica determinar qué estructuras de datos pueden ser utilizadas para modelizar las componentes del sistema físico.

Nosotros nos concentraremos en modelizar ***sistemas de colas***, esto es, sistemas con lista de espera. Un gran número de sistemas físicos puede ser modelizado de esta manera, por ejemplo: colas de banco, en comercios y supermercados, etc.. Los clientes que forman estas colas son atendidos considerando que el primero que llega es el primero en ser atendido. Esto respeta la propiedad FIFO de las colas. Lo mismo se puede decir de los vehículos esperando en una gasolinería, los aviones esperando permiso para despegar de la torre de control y las transmisiones de mensajes en una red informática de comunicaciones.

En su forma más simple un sistema de colas consiste en un único servidor que atiende a una única cola.



Con el fin de simular un sistema de esta clase se debe especificar la siguiente información:

1. La distribución de tiempos de llegada de clientes. Se asume que los clientes llegan de acuerdo a una distribución de probabilidad conocida A(x).(A: arribo)

2. La distribución del tiempo de atención. Se asume que el tiempo de atención de un cliente individual varía de acuerdo a una distribución de probabilidad conocida S(x) (S: servicio).

Dado este escenario, habitualmente estamos interesados en determinar la distribución de la longitud de la cola y de los tiempos de espera de los clientes, y la eficiencia del servidor.
Es fácil extender esta idea a escenarios más complicados, por ejemplo los siguientes: