Materia: Informática II

Unidad 1. Estructuras simples de datos

Temario

1.1. Tipos de datos estándar
- 1.1.1. Enteros
- 1.1.2. Reales
- 1.1.3. Carácter
- 1.1.4. Lógicos

1.2. Tipos definidos por el programador
- 1.2.1. De subrango
- 1.2.2. Enumerativos

Introducción

El estudio de la estructura de datos es importante para la definición de la organización lógica de los mismos y para estar en condiciones de administrar la memoria real y la de la computadora. La elección de la estructura de datos dependerá de la organización del compilador y de la aplicación a ejecutar. Por eso, al finalizar el estudio de esta asignatura, el alumno será capaz de implantar y manipular las estructuras de datos más importantes, así como aplicarlas en el manejo de la información**.**

Estructura es la relación que hay entre los elementos de un grupo. En informática, esta relación se llama estructura de datos. Uno de los problemas más serios con los que se enfrentará el programador es precisamente la estructuración y almacenamiento de información. Deberá estudiar con mucho cuidado la estructuración, almacenamiento y recuperación de los datos, comparándolos con la eficacia del algoritmo elegido para su manipulación. De no ser así, la solución dada a un problema puede resultar demasiado costosa en espacio de memoria o en tiempo de trabajo-máquina.

En esta unidad, se estudiarán las formas que pueden tomar los datos y las aplicaciones en que se utilizan las estructuras que se almacenarán en la memoria principal o interna, soportes externos de almacenamiento de información, memoria secundaria o externa, cintas, discos, etcétera.

Objetivo

El alumno conocerá las estructuras de datos simples y su definición; asimismo, analizará por qué no pueden contener más estructuras.

Esbozo

Uno de los fines al utilizar la computadora es manejar información o datos (cifras de ventas de un supermercado o calificaciones de una clase, por ejemplo). Un dato es la expedición general que describe los objetos con los cuales opera una computadora. La mayoría de las computadoras puede trabajar con varios tipos de datos. Los algoritmos y programas correspondientes operan sobre datos. La acción de las instrucciones ejecutables de las computadoras realiza cambios en los valores de las partidas de datos. Los datos de entrada son transformados por el programa, después de las etapas intermedias, en datos de salida.

En el proceso de solución de problemas, el diseño de la estructura de datos es tan importante como el del algoritmo y el del programa que se basa en el mismo.

Los lenguajes de alto nivel se fundamentan en abstracciones e ignoran los detalles de la representación interna. Asimismo, en ellos aparece el conjunto de tipo de datos y su representación. Hay datos simples (sin estructura) y compuestos (estructurados), que se representan de diferentes formas en la computadora. Un dato es un conjunto o secuencia de *bits* (dígitos 0 ó 1).

Tipos de datos simples:

Numéricos (integer, real)
Lógicos (boolean)
Carácter (char, string)

Hay lenguajes de programación –FORTRAN esencialmente– que admiten otros tipos de datos: complejos –permiten tratar números complejos–, y Pascal, que declara y define sus propios tipos de datos: enumerados (*enumerated*) y subrango (*subrange*).

1.1. Tipos de datos estándar

1.1.1. Enteros

El tipo entero es un subconjunto finito de los números enteros. Dentro de esta categoría están incluidas constantes, funciones y expresiones de tipo entero. Los enteros son números completos, no tienen componentes fraccionarios o decimales y pueden ser negativos o positivos.

Ejemplos de números enteros:

5	6
–15	4
20	17
1.340	26

En ocasiones, los enteros se denominan números de punto o coma fijo. Los números enteros máximos y mínimos de una computadora * suelen ser –32768 a +32767. Fuera de este rango, los enteros no se pueden representar como tales, sino como reales.

Colectivamente, los operadores que se usan para realizar operaciones de tipo numérico se denominan aritméticos. Hay seis operadores de este grupo que pueden aplicarse con otros de tipo entero. Cinco de ellos producirán un resultado entero (un resultante de tipo entero); el sexto, uno de tipo real:

Operador aritmético	Efecto	Tipo de operandos	Tipo resultantes
+	Adición	Entero	Entero
-	Sustracción	Entero	Entero
*	Multiplicación	Entero	Entero
/	División	Entero	Real
DIV	División truncada	Entero	Entero
MOD	Resto de la división	Entero	Entero

1.1.2. Reales

El tipo real consiste en un subconjunto de números reales. Incluye constantes, variables, funciones y expresiones de tipo real. Los números reales siempre tienen un punto decimal y pueden ser positivos o negativos. Un número real consta de un entero y una parte decimal.

Ejemplos de números reales:

0.08	3739.41
3.7452	-52.321
-8.12	3.0

En aplicaciones científicas, se requiere de una representación especial para manejar números muy grandes, como la masa de la Tierra, o muy pequeños, como la masa de un electrón. Una computadora sólo puede representar un número fijo de dígitos, que puede variar de una máquina a otra (ocho es el número típico). Este límite representará y almacenará números muy grandes o muy pequeños como los siguientes:

4867213432 0.00000000387

Hay un tipo de representación denominado notación exponencial o científica; se utiliza para números muy grandes o muy pequeños. Así,

367520100000000000000,

se representa en notación científica, descomponiéndolo en grupos de tres dígitos:

367 520 100 000 000 000 000,

y, posteriormente, en forma de potencias de 10:

3.675201*10 19.

Y, de modo similar:

.0000000000302579,

se representa como:

3.02579*10-11

La representación en coma flotante es una generalización de notación científica.

Las expresiones siguientes son equivalentes:

3.675201*10 19=.3675201*10 20=.03675201*10 21=,...,

=36.75201*10 18=367.5201*10 17=...

En estas expresiones se consideran la mantisa (parte decimal), el número real y el exponente (parte potencial), el de la potencia de 10:

36.75201 *mantisa* 18 *exponente*

1.1.3. Carácter

Es el conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato de este tipo contiene un solo carácter.

La mayoría de las computadoras reconoce los siguientes caracteres alfabéticos y numéricos:

- caracteres alfabéticos (A,B,C,...,Z)
- caracteres numéricos (1,2,...,9)
- caracteres especiales (+,-,*,/,~,.,;.<,>,$,...)

1.1.4. Lógicos

También denominados booleanos, son datos que sólo pueden tomar uno de dos valores:

Cierto o verdadero (*true*) y falso (*false*).

Este tipo de datos se emplea para representar las alternativas (sí/no) a determinadas condiciones. Por ejemplo, cuando se quiere saber si un valor entero es par, la respuesta será verdadera o falsa. Y las expresiones de tipo lógico se forman combinando operandos del mismo tipo mediante operadores relacionales, que son:

Operador relacional	Significado
=	igual a
<>	no igual a
<	menor que
<=	menor que o igual a
>	mayor que
>=	mayor que o igual a

Ejemplo:

Expresión	Valor
2=3	falso
2<3	verdadero
0.6>=1.5	falso
0.6>=-1.5	verdadero
-4<>4	verdadero
1.7<=-2.2	falso

1.2. Tipos de datos definidos por el programador

1.2.1. De subrango

Subrango es una porción del rango original de un tipo de dato simple. Los datos que lo integran son los individuales que se encuentran dentro de ese subrango, formando de este modo un subconjunto de datos ordenados y contiguos. Además, el tipo de datos original es el tipo soporte (se considera que cada uno de los datos del subrango es del tipo soporte).

El concepto de subrango puede aplicarse a cualquier conjunto de datos simples y ordenados, incluyendo los definidos enumerativamente y los tres tipos estándar: entero, char y booleano.

La forma general de una definición de tipo subrango es:

TYPE *nombre = primer dato...ultimo dato*;

donde *nombre* es el nombre del tipo; *primer dato*, el primero de los datos ordenados incluidos en el subrango; y *ultimo dato*, el último de los datos ordenados.

Turbo Pascal permite crear, de diversas maneras, nuevos tipos de datos. Un método que se emplea al respecto consiste en comenzar con un tipo de datos ordinal y restringir los que se encuentren dentro de un subrango particular. Por ejemplo, un subrango de enteros es:

1...10

Este subrango está formado por los enteros desde 1 hasta 10 (1,2,3,4,5,6,7,8,9,10).

El siguiente es un ejemplo de subrango para el tipo char:

´D´…´H´

Este subrango contiene los caracteres entre ´D´y ´H´ inclusive (‘D’, ‘E’, ‘F’, ‘G’, ‘H’).

El programador puede crear variables cuyos valores estén restringidos a un determinado subrango. Para ello, basta con escribir el subrango como un tipo de datos dentro de la declaración de la variable. Por ejemplo, consideremos la declaración siguiente:

```
var
num:1..10;
ch:’D..’H’;
```

Supongamos que una variable pertenece a un subrango. Si le damos un valor fuera del rango definido, habrá error durante la ejecución. Por ejemplo, si se intenta asignar a la variable num de la declaración anterior los valores 15 ó –1, se presentará error.

Cuando se emplean subrangos, es posible asignar valores de variables que corresponden a un tipo cuyo subrango sea mayor, siempre y cuando el valor pertenezca al subrango. Por ejemplo, si Num es una variable de tipo integer y Small pertenece al subrango 0…10, la asignación es:

```
Small : = Num;
```

Lo anterior es correcto si el valor de Num se encuentra dentro del rango 0...10. De otro modo, habría falla de ejecución.

También siempre es posible asignar a una variable del tipo más grande el valor de una de subrango.

El identificador de tipo Natural describe un tipo que es un subrango de INTEGER. Además es importante señalar que la definición de Natural define un rango de valores de este tipo. Igualmente, el identificador de tipo Digito describe un tipo que es subrango de CHAR.

NOTA: los valores que se asignan a las variables subrango no pueden estar fuera del subrango establecido en la definición de tipo. No es válido definir tipos subrango del tipo REAL.

	Definición de tipos	***TYPE***	
usando subrangos	Natural	=	O...MAXINT;
	Positivo	=	1...MAXINT;
	Caracteres Dígito	=	‘0’...’9’:

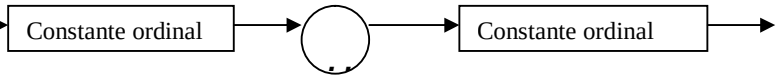
```
Caracteres Mayúsculas    =      'A'…'Z';
Caracteres Minúscula             =      'a'...'z';

VAR
Cuantas                  :      Natural;
Cuenta                   :      Positivo;
Digito                   :      Catacteres Dígito;
Caract Mayusculas        :      Carácter Mayúsculas;
MinusculaCh              :      Carácter Minúscula;
```

	Asignación a	***Enunciado Efecto***
Variable tipo subrango	Cuenta:=1;	se asigna a Cuenta el valor 1
	Cuenta:=0;	¡EL PROGRAMA SE TRABAJA!
	Cuantas:=0;	Se asigna a Cuantas el valor 0
	Cuantas:=Cuantas –1;	¡EL PROGRAMA SE TRABAJA!
	Digito:='7';	Se asigna a Digito el valor '7'
	Digito:=CHR(48);	Si se usa el código ASCII, se asigna a Digito el valor '0'; si se usa el código EBCDIC, ¡EL PROGRAMA SE TRABAJA!

Diagrama sintáctico

para definición de tipo subrangos



Definición de tipos usando enumeraciones y subrangos

```
TYPE

Color = (Rojo, Blanco, Azul, Amarillo, Verde, Naranja, Negro);
ColorPatriotico = Rojo...Azul;
ColorHalloween = Naranja...Negro;
Relacion = (Menor, Igual, Mayor);
MesDelAño = (Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov, Dic,);
DiaDeLaSemana = (Lun, Mar, Mie, Jue, Vie, Sab, Dom,);
Palo = (Basto, Diamantes, Corazones, Espadas);
Figura = (Dos, Tres, Cuatro, Cinco, Seis, Siete, Ocho, Nueve, Diez, Sota,
Reina, Rey, As);
CaraNaipes = Sota...Rey;

VAR
Dia : DiaDeLaSemana;
Mes : MesDelAño;
```

Prueba : Relación;
Tono : Color;
PaloNaipe : Palo;
FiguraNaipe : Figura

1.2.2. Enumerativos

Son una secuencia ordenada de identificadores que se interpretan como datos individuales, pero que, tomados colectivamente, se asocian a un nombre que sirve para identificar el tipo. La asociación entre el nombre del tipo y los datos individuales se establece mediante una definición *type.*

En términos generales, la definición de tipo se escribe:

TYPE *nombre* = (*dato 1, dato 2, . . . , dato n*),

donde *nombre* es la nominación del tipo de dato enumerativo y *dato 1, dato 2,* etcétera, son los datos concretos.

Turbo Pascal permite que el usuario defina tipos de datos ordinales que constituyan una lista de identificadores, definidos también por el usuario.

Un tipo de datos enumerados es un tipo ordinal, donde el orden está dado por la ubicación de los valores dentro de los paréntesis que delimitan la definición del tipo de datos. Por ejemplo, empleamos un tipo de datos formado por colores:

(red, green, blue)

El orden de los valores es:

red < green < blue

Para definir la variable Color, a la que se le pueden asignar los valores del tipo de datos anterior, debe emplearse una declaración de la forma:

Valor Color: (red, green, blue)

De acuerdo con lo anterior, las siguientes son proposiciones válidas que utilizan la variable Color:

```
Color:=Red;
If Color = Red
The
FillRegion;
For Color :=Red to Blue do
TestSize (Color)
```

Materia: Informática II

Unidad 2. Estructuras estáticas en memoria principal

Introducción

El enfoque de esta unidad es destacar la importancia del uso de tipos de datos abstractos en la solución de problemas.

Las estructuras de datos son esenciales en los sistemas de información. Para diseñar apoyos eficientes en los recursos de datos, se debe saber cómo representar y manipularlos. Los tipos de datos que se emplean en un programa ayudan a determinar cuáles son los requerimientos de almacenamiento y su tiempo de respuesta, los de proceso, los de entrada/salida, etcétera. Gran cantidad de estructuras de datos está disponible en primera instancia, puesto que diferentes tipos de datos se utilizan en diversas formas. Las que se utilizan actualmente definen las rutas de acceso entre los elementos de datos viables.

Estudiaremos los arreglos, bloques básicos para la construcción de estructuras de datos más complejas (veremos primero los simples y multidimensionales). Además analizaremos los registros, componentes elementales de los archivos y de las bases de datos. Y finalmente, desarrollaremos una aplicación que requiera de arreglos y registros.

Objetivo

El alumno conocerá y analizará los elementos que sirven para representar las estructuras estáticas, tanto físicas como lógicas; asimismo, su importancia en la solución de problemas.

Esbozo

Una estructura de datos es una colección de datos que pueden ser caracterizados por su organización y operaciones.

Las estructuras de datos son muy importantes en los sistemas de computadora. Los tipos de datos más frecuentes utilizados en los lenguajes de programación son:

- Datos simples estándar
  - Carácter
    - Entero (integer)
    - Real (real)
    - (char)
    - Lógico (boolean)
  - Definidos por el programador
    - Subrango (subrange)
    - Enumerativo (enumerated)
- Datos estructurados
  - Simples o estáticos
    - Arrays (vectores / matrices)
    - Registros
    - Ficheros
    - Conjuntos
    - Cadenas (string)
  - Compuestos o dinámicos
    - Listas (pilas / colas)
    - Lista enlazas
    - Árboles
    - Grafos

2.1. Tipos de datos abstractos

2.1.1. Definición

Los datos abstractos son el resultado de “empacar” un tipo de datos junto con sus operaciones, de modo que pueda considerarse en términos de su ejemplo, sin que el programador tenga que preocuparse por su representación en memoria o la instrucción de sus operaciones.

2.1.2. Importancia del uso de tipos de datos abstractos –simples o estáticos arrays (vectores / matrices), ficheros, conjuntos, cadenas (string)– en la solución de problemas

Los tipos de datos simples o primitivos no están compuestos por otras estructuras de datos. Los más frecuentes en casi todos los lenguajes son: enteros, reales y de

carácter (char). Los tipos lógicos, de subrango y enumerativos son propios de lenguajes estructurados como Pascal.

Las estructuras de datos estáticos son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute (no puede modificarse dicho tamaño durante la ejecución del programa). Estas estructuras están implementadas en casi todos los lenguajes: arrays (vectores/tablas–matrices), registros, ficheros. Los conjuntos son específicos del lenguaje Pascal.

La estrategia descendente también puede aplicarse en el diseño de datos. A esto se le conoce como abstracción de los datos (no se especifican los datos prácticos). Basta con determinar las funciones que el programa emplea para manipular estructuras de datos. En el siguiente refinamiento, se proporcionarán más detalles de la estructura de datos, como seudocódigo de las funciones. Y en el refinamiento final se especificarán los detalles de las funciones y la organización de memoria práctica para le estructura de datos.

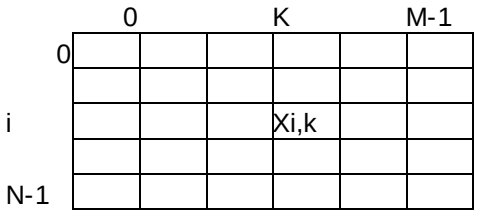
Si se aplica efectivamente la abstracción de los datos, es posible simplificar el proceso de resolución del problema de forma similar que cuando se elaboran algoritmos descendentes.

2.1.3. Representación de un objeto abstracto

Al comenzar el diseño de un TAD (Tipo Abstracto) es necesario tener una representación abstracta del objeto sobre el cual se quiere trabajar, sin establecer un compromiso con ninguna estructura de datos concreta ni tipo de dato del lenguaje de programación seleccionado. Esto va a permitir expresar las condiciones, relaciones y operaciones de los elementos modelados, sin restringirse a una representación interna concreta. En este orden, lo primero que se hace es dar nombre y estructura a los elementos a través de los cuales se puede modelar el estado interno de un objeto abstracto, utilizando algún formalismo matemático o gráfico.

Ejemplo:

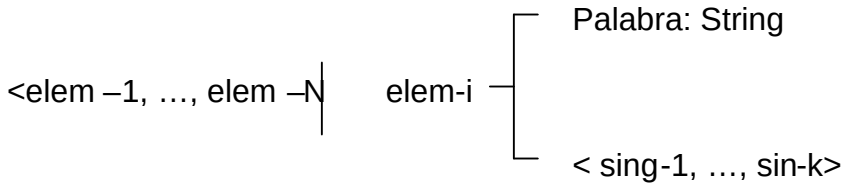
En el TAD Matriz, una manera gráfica de representar el objeto abstracto sobre el cual se va a trabajar es la siguiente:



Con esta notación, es posible hablar de los componentes de una matriz y sus dimensiones, de las nociones de fila y columna, de la relación de vecindad entre elementos, etcétera, sin necesidad de establecer estructuras de datos concretas para manejarlas.

Ejemplo:

Para el TAD diccionario, en el que cada palabra tiene uno o más significados asociados, el objeto abstracto puede representarse mediante el formalismo siguiente:



Dándole nombre a cada una de sus partes y relacionándolas, y ligando las palabras con sus significados se define claramente la estructura general del formalismo anterior. En este caso, se utiliza la notación <...> para expresar múltiples repeticiones y el símbolo de bifurcación para mostrar composición.

Otra manera de representar el mismo objeto abstracto es:

$$<[palabra_1, < s_{11},..., s_{1k} >],..., [palabra_N, < s_{N1},..., s_{Nk} >]>$$

Incluso podríamos hacer la siguiente representación gráfica:

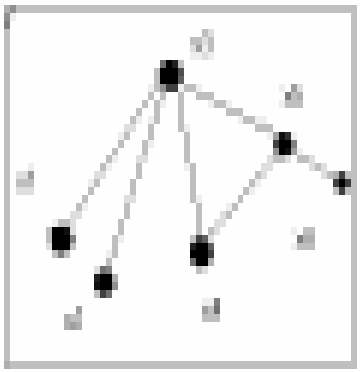
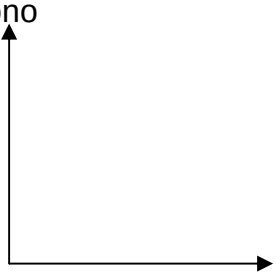
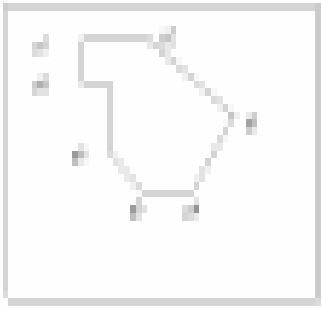
Palabra -1	s-11, ...,s-Nk
Palabra - N	s-11,... , s-Nk

Lo importante, en todos los casos, es que los componentes del objeto abstracto sean diferenciables, y que su estructura global se haga explícita.

Ejemplo:

Hay objetos abstractos con una representación gráfica natural. Veamos algunos ejemplos:

Conjunto	$\{x_1,...,x_N\}$
Cadena de caracteres	“$c_1c_2...c_N$”

Vector	0 1 N-1 .....
Polinomio	$C_0+c_1x+c_2x^2+...+c_Nx^N$
Red	
Lista	$<x_1,x_2,...,x_N>$
Polígono 	

Ejemplo:

Algunos elementos de la realidad se pueden modelar con una composición de atributos, los cuales representan las características importantes del objeto abstracto, definidos también en términos de otros elementos de la realidad. En el caso de una biblioteca, se puede tener el formalismo siguiente:

Biblioteca
- Fichero 1: Fichero Autor
- fichero 2: Fichero Título
- Libros: Bodega

Los atributos corresponden a objetos abstractos de los TAD Fichero Autor, Fichero Título y Bodega. En este caso, se dice que la Biblioteca es un cliente de dichos

TAD. Para claridad en la notación, los nombres de los TAD se escriben en mayúsculas. Y las nominaciones de los atributos tienen las características de cualquier variable.

2.1.4. Invariante de un dato abstracto

El invariante de un TAD establece una noción de validez, en términos de condiciones, para cada uno de los objetos abstractos, sobre su estructura interna y sus componentes. Es decir, define en qué casos un objeto abstracto modela un elemento posible del mundo del problema. Por ejemplo, para el TAD Conjunto y la notación $\{^{x}1,...^{x}N\}$ el invariante debe exigir que todos los $^{x}_{i}$ pertenezcan al mismo tipo de dato, y que sean diferentes entre sí. De este modo, el objeto abstracto estará modelando realmente un conjunto. Estructuralmente, el invariante está compuesto por condiciones que restringen el dominio de los componentes internos y sus relaciones.

Ejemplo:

El objeto abstracto del TAD Diccionario debe incluir tres condiciones en el lenguaje natural y en el formal:

- Las palabras estarán ordenadas ascendentemente y sin repetición:
  $Elem_i.palabra < elem_{i+1}.palabra$, $1 \leq i < N$
- Los significados estarán ordenados ascendentemente y sin repetición:
  $Elem_i.\ Sig_{r+1}$, $1 \leq i \leq N$, $1 \leq r < k$
- Toda palabra debe tener asociado por lo menos un significado:
  $Elem_i =$ palabra, $<sig_1,...,sig_k>$, $k>0$

Si un objeto TAD Diccionario no cumple cualquiera de las condiciones anteriores, no se encuentra modelando un diccionario real, de acuerdo con el modelaje que ya se ha hecho con anterioridad.

2.1.5. Especificación de un tipo de dato abstracto

Un TAD se define con un nombre, un formalismo para expresar un objeto abstracto, un invariante y un conjunto de operaciones sobre este objeto. En este tutorial, utilizamos el esquema siguiente:

TAD <nombre>
<Objeto abstracto>
<Invariante del TAD>
<Operaciones>

La especificación de las operaciones consta de dos partes. Primero se coloca la funcionalidad de cada una de ellas (dominio y condominio de la operación) y, luego, su comportamiento, mediante dos aserciones (precondición y poscondición)

que indican la manera como se va afectando el estado del objeto una vez ejecutada la operación:

- <operación 1> <dominio> → <condominio>
- ...
- <operación k>: <dominio> <condominio>

<prototipo operación 1>
/*Explicación de la operación */
{pre:...}
{post:...}

La precondición y la poscondición de una operación pueden referirse únicamente a los elementos que integran el objeto abstracto y a los argumentos que recibe. No incluye ningún otro tipo de elemento del contexto en el cual se va a ejecutar. En la especificación de las operaciones, precondición y poscondición, debe suponerse que el objeto abstracto sobre el cual se va a operar cumple el invariante. Lo anterior quiere decir que dichas aserciones sólo deben tener condiciones adicionales a las de validez del objeto. Si la precondición de una operación es TRUE, es decir, no impone ninguna restricción al objeto abstracto ni a los argumentos, se omite la especificación.

Es importante colocar una breve descripción de cada operación, de manera que el cliente pueda darse una idea rápida de los servicios que ofrece un TAD, sin necesidad de hacer una interpretación de su especificación formal, que está dirigida sobre todo al programador.

Al seleccionar los nombres de las operaciones, se debe tener en cuenta que no pueden existir dos operaciones con igual nominación en un programa, incluso si pertenecen a TAD diferentes. Por esta razón, conviene agregar un mismo sufijo a todas las operaciones de un TAD, de tal forma que las identifique. Además, es preferible que este sufijo tenga por lo menos tres caracteres.

Ejemplo:

Para definir el TAD Matriz de valores enteros, se puede utilizar la especificación siguiente:

TAD Matriz

0

k

M-1

0

I

Xi,k

N-1

{inv: N>0, M>0}

- crearMat: int x int ⟹ Matriz
- asignarMat Matriz x int x int x int Matriz

- infoMat: Matriz x int x int ⟹ int
- filasMat Matriz int
- columnasMat Matriz int

Matriz crearMat(int fil, int col)
/* Construye y retorna una matriz de dimensión [0... fil-1, 0... col-1], inicializada en 0 */

{pre: fil > 0, col > 0 }
{post: crearMat es una matriz de dimensión [0... fil-1, 0... col-1], $x_{ik} = 0$ }

Void asignarMat (Matriz mat, int fil, int col, int val)
/* Asigna a la casilla de coordenadas [fil, col] el valor val */

{pre: 0 $\leq$ fil < N, 0 $\leq$ col M}
{post: $X_{fil, col} = val$}

Int infoMat (Matriz mat, int fil, int col)
/* Retorna el contenido de la casilla de coordenadas [fil, col] */

{pre: 0 $\leq$ fil < N, 0 $\leq$ col M}
{post: infoMat = $X_{fil, col}$ }

Int filasMat (Matriz mat)
/* Retorna el número de filas de la matriz */

{post: filasMat = N}

Int columnasMat (Matriz mat)
/* Retorna el número de columnas de la matriz */

{post: columnasMat = M}

En el caso del TAD matriz, el invariante sólo establece una restricción para el número de filas y de columnas (es decir, coloca una limitante al dominio en el cual puede tomar valores). También cuenta con cinco operaciones para administrar un objeto del TAD: una para crearlo, otra para asignar un valor a una casilla, una más para tomar el valor de una casilla y dos para formar sus dimensiones. Con este conjunto de operaciones, y sin necesidad de seleccionar estructuras de datos específicas, es posible resolver cualquier problema que involucre una matriz.

Es importante anotar que cada elemento utilizado como parte del formalismo de un objeto abstracto puede emplearse directamente como parte de la especificación de una operación. Es el caso de los valores N M, utilizados como parte de la poscondición de las operaciones filasMat y columnasMat.

2.1.6. Tipos de operaciones

Las operaciones de un TAD se clasifican en tres grupos, según su función sobre el objeto abstracto:

- Constructora: crea elementos del TAD. En el caso típico, elabora el objeto abstracto más simple. Tiene la estructura siguiente:

Clase < constructora> (<argumentos>)
{pre: <condiciones de los argumentos>}
{post: <condiciones del objeto inicial, adicionales al invariante>}

En los anteriores ejemplos, las operaciones crearMat y crearDic son las constructoras de los TAD Matriz y Diccionario respectivamente. Un TAD puede tener múltiples constructoras.

- Modificadora: puede alterar el estado de un elemento del TAD. Su misión es simular una reacción del objeto. Su estructura típica es:

Void <modificadora> (<objetoAbstracto>, <argumentos>)
{pre: <condiciones del objeto adicionales al invariante, condiciones de los argumentos> }
{post: <condiciones del objeto adicional al invariante> }

En el ejemplo del TAD Matriz la única modificadora es la operación asignarMat, que altera el contenido de una casilla de la matriz. Otra modificadora posible de ese TAD sería una que alterara sus dimensiones. Al final de toda modificadora, se debe continuar cumpliendo el invariante.

- Analizadora: no altera el estado del objeto, sino que tiene como misión consultar su estado y retornar algún tipo de información. Su estructura es:

<tipo><analizadora>(objetoAbstracto>, <argumentos>)
{pre: <condiciones del objeto adicionales al invariante, condiciones de los argumentos>}
{post: <analizadora> = función (<estado del objetoAbstracto>)}

En el TAD Matriz, las operaciones infoMat, filasMat y columnasMat son analizadoras. A partir de ellas, es posible consultar cualquier aspecto del objeto abstracto. En la especificación del TAD es conveniente hacer explícito el tipo de operación al cual corresponden, por que, en el momento de hacer el diseño de manejo de error, es necesario tomar decisiones diferentes.

Además, hay varias operaciones interesantes que deben agregarse a un TAD para aumentar su portabilidad. Son casos particulares de las ya vistas, pero dada su importancia, merecen atención especial. Entre estas operaciones se pueden nombrar las siguientes:

- Comparación: analizadora que permite calcular la noción de igualdad entre dos objetos del TAD.
- Copia: modificadora que facilita alterar el estado de un objeto del TAD copiándolo a partir de otro.
- Destrucción: modificadora que se encarga de retornar el espacio de memoria dinámica ocupado por un objeto abstracto. Después de su ejecución, el objeto abstracto deja de existir, y cualquier operación que se aplique sobre él va a generar error. Sólo se debe llamar esta operación cuando un objeto temporal del programa ha dejado de ocuparse.
- Salida a pantalla: analizadora que permite al cliente visualizar el estado de un elemento del TAD. Esta operación, que parece más asociada con la interfaz que con el modelo del mundo, puede resultar una excelente herramienta de depuración en la etapa de pruebas del TAD.
- Persistencia: operación que permite salvar/leer el estado de un objeto abstracto de algún medio de almacenamiento en memoria secundaria. Esto permite a los elementos de un TAD sobrevivir a la ejecución del programa que los utiliza.

En general, las aplicaciones están soportadas por manejadores de bases de datos que se encargan de resolver los problemas de persistencia, concurrencia, coherencia, etcétera. Sin embargo, para aplicaciones pequeñas, puede ser suficiente un esquema de persistencia sencillo, en el cual cada TAD sea responsable de su propia administración.

2.1.7. Tipos de datos abstractos parametrizados

Desarrollar un TAD contenedor parametrizado tiene la ventaja de precisar qué puntos del diseño son dependientes del tipo de elemento que maneja. Lo ideal es poder reutilizar todo el software de una aplicación a otra, y no sólo el diseño, de tal forma que sea posible contar con librerías genéricas perfectamente portables, capaces de contener elementos de cualquier tipo. La sintaxis para especificar un TAD paramétrico se puede apreciar en el ejemplo siguiente.

Ejemplo:

Si se quiere definir un TAD Conjunto para cualquier tipo de elemento, puede emplearse un esquema como este:

TAD Conjunto [tipo]

	$\{x_1, x_2,...x_N\}$		
{ inv: $x_i!=x_k$, i!=k,x_i pertenece al TAD tipo			
• crearConjunto:		⇒	Conjunto
• insertarConjunto:	Conjunto x tipo		Conjunto
• eliminarConjunto:	Conjunto x tipo		Conjunto
• esteConjunto:	Conjunto x tipo		int
• vacioConjunto:	Conjunto		int

Void insertarConjunto (Conjunto conj, Tipo elem) {pre: conj = { $x_1, x_2, ...x_N$ }, elem ¡=x_i ¡} {post: conj = { $x_1, x_2, ...x_N$ }, elem }}

2.2. Arreglos

2.2.1. Definición

Los arreglos son estructuras de datos compuestos en las que se utilizan uno o más subíndices para identificar los elementos individuales almacenados, a los que es posible tener acceso en cualquier orden.

Examinaremos varias estructuras de datos, que son parte importante del lenguaje Pascal, y su utilización. Son del tipo compuesto, y están integradas de tipos de datos simples que existen en el lenguaje como conjuntos ordenados, finitos de elementos homogéneos. Es decir, de un número específico de elementos grandes o pequeños, pero organizados de tal manera que hay primero, segundo, tercero, etcétera; además, los elementos deben ser del mismo tipo.

La forma de estructura de datos no se describe completamente, pero debemos especificar cómo se puede accesar a ella. Un arreglo tiene dos tipos de datos asociados. Las dos operaciones básicas que accesan un arreglo son extracción y alimentación. La de extracción acepta un arreglo *a* y un elemento de su tipo índice, que puede ser cualquier tipo ordinario o base. El elemento más pequeño de un arreglo del tipo índice es su limite inferior, y el más alto, su límite superior.

2.2.2. Operaciones con arreglos

Un arreglo unidimensional puede ser implementado fácilmente. Consideremos primero arreglos cuyos tipos de índice son subrangos de enteros.

Todos los elementos de un arreglo tienen el mismo tamaño fijado con anterioridad. Sin embargo, algunos lenguajes de programación permiten arreglos de objetos de dimensión variada.

Un método para hacer un arreglo de elementos de tamaño variable consiste en reservar un conjunto contiguo de posiciones de memoria, cada uno de los cuales contiene una dirección. Los contenidos de cada una de esas posiciones de memoria es la dirección del elemento del arreglo de longitud variable almacenando en otra posición de memoria. Otro método, similar al anterior, implica guardar todas las porciones de longitud fija de los elementos en el área de arreglo contiguo, y además almacenar todas las direcciones de la porción de longitud variable en el área contigua.

2.2.3. Arreglos bidimensionales

El tipo de un arreglo puede ser otro arreglo. Y un elemento de este arreglo puede ser accesado mediante la especificación de dos índices **–** los números de la fila y de la columna. Por ejemplo, en la fila 2, columna 4 –*a*[2][4] o [2,4]–, observamos que *a* [2] hace referencia a toda la fila 2, pero no hay forma equivalente para aludir a una columna completa:

COLUMNA 1	COLUMNA 2	COLUMNA 3	COLUMNA 4	COLUMNA 5

FILA 1
FILA 2
FILA 3

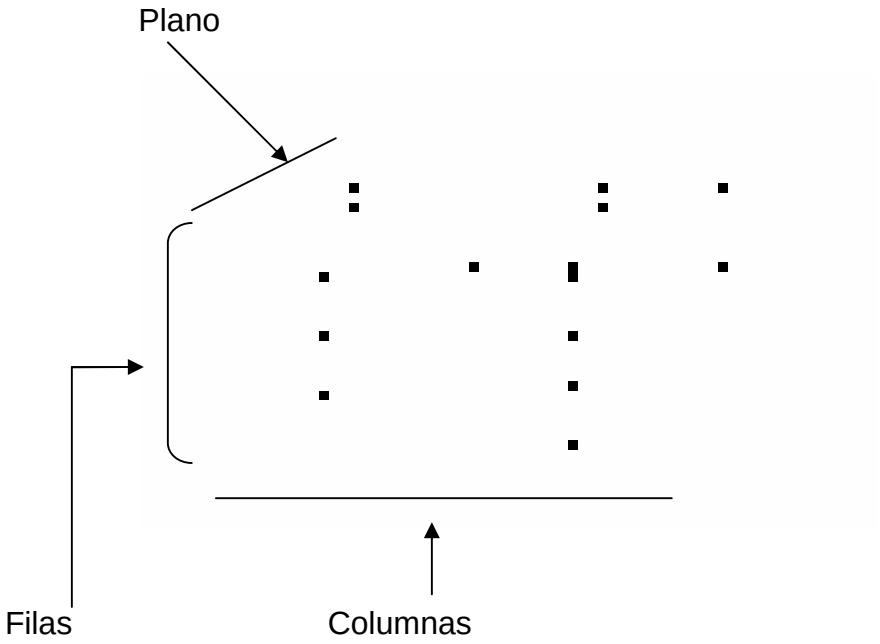
Un arreglo de dos **–** dimensiones ilustra claramente las diferencias lógica y física de un dato. Es una estructura de datos lógicos, útil en programación y en la solución de problemas. Sin embargo, aunque los elementos de dicho arreglo están organizados en un diagrama de dos dimensiones, el hardware de la mayoría de las computadoras no da este tipo de facilidad. El arreglo debe ser almacenado en la memoria de la computadora y dicha memoria es usualmente lineal.

Un método para mostrar un arreglo de dos dimensiones en memoria es la representación fila – mayor. Bajo esta representación, la primera fila del arreglo ocupa el primer conjunto de posiciones en memoria reservada para el arreglo; la segunda, el segundo, y así sucesivamente.

2.2.4. Arreglos multidimensionales

Puede haber arreglos de más de dos dimensiones. Por ejemplo, uno tridimensional, que está especificado por medio de tres subíndices: *c*[4] [1][3] o *c*[4,1,3]. El primer índice precisa el número del plano; el segundo, el de la fila; y el tercero, el de la columna. Este tipo de arreglo es útil cuando se determina un valor mediante tres entradas.

La representación de arreglos en la forma de fila – mayor puede extenderse a arreglos de más de dos dimensiones:



El último subíndice varía rápidamente y no aumenta hasta que todas las combinaciones posibles de los subíndices a su derecha hayan sido completadas.

2.3. Registros

2.3.1. Definición

Los registros son estructuras de datos heterogéneos por medio de las cuales se tiene acceso, por nombre, a los elementos individuales, llamados campos. En

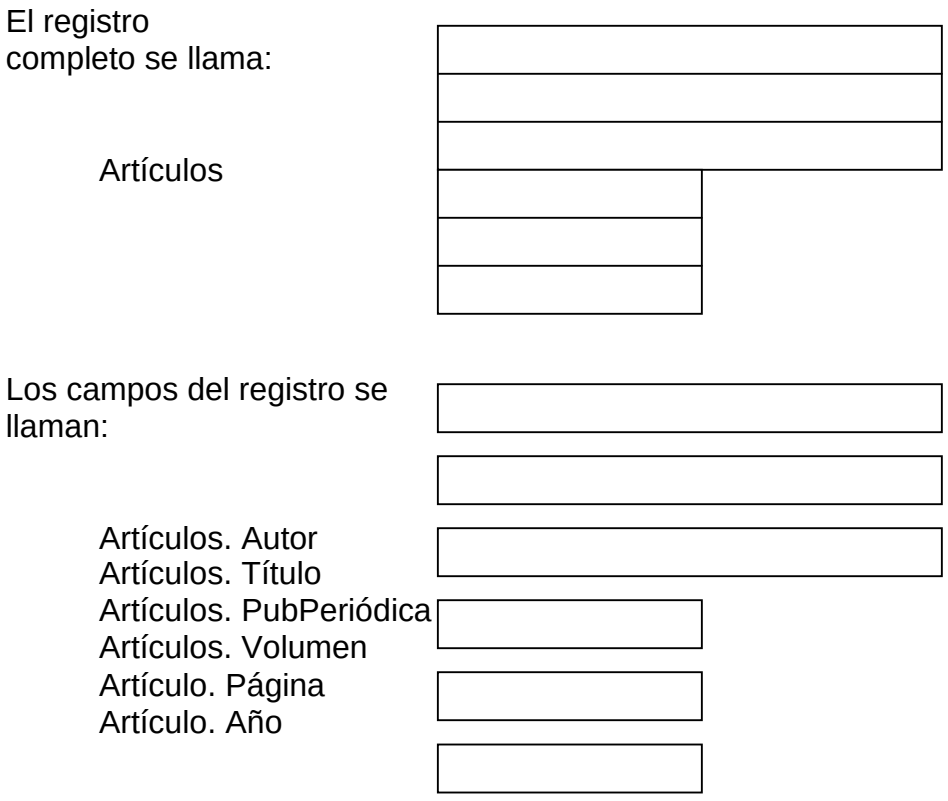
otras palabras, son un grupo de elementos en el que cada uno de ellos se identifica por medio de su propio campo.

2.3.2. Acceso a los campos de un registro

Tenemos una variable llamada Artículo: registro individual que consta de seis campos de registro, cada cual con su nombre o identificador, integrado a partir de la nominación del registro, un punto y el nombre del campo:

- Artículo. Autor es el nombre del primer campo, que es una cadena de 40 caracteres.
- Artículo. Título es el nombre del segundo campo, que también es una cadena de 40 caracteres.
- Artículo. PubPeriódica es el nombre del tercer campo, otra cadena de 40 caracteres.
- Artículo. Volumen es el nombre del cuarto campo, un entero.
- Artículo. Página es el nombre del quinto campo, un entero.
- Artículo. Año, es el nombre del sexto campo, un entero.

Podemos visualizar el registro y sus campos como se muestra en la figura siguiente:



Como sucede con los demás tipos de datos compuestos, con excepción de los arreglos de cadena, aquí no hay constantes de registro. Entonces, no es posible asignar un valor constante a una variable de registro; hay que dar valores individualmente a los diversos campos.

```
Artículo. Autor          := 'Dijkstra, E.                                    ';
Artículo. Título         := 'Go to statement cosidered harmful               ';
Artículo. PubPeriódica   := 'Comunications of the ACM                        ';
Artículo. Volumen        := 11;
Artículo. Página         := 147;
Artículo. Año            := 1968;
```

La única operación de registro completo que ofrece Pascal es el enunciado de asignación: si Artículo1 y Artículo2 son dos registros exactamente del mismo tipo (en este ejemplo, del tipo ArtículoPublicado), podemos asignar el valor de uno al otro:

```
Artículo1 := Artículo2;
```

Con este proceso, logramos lo mismo que si copiáramos todos los campos individualmente:

```
Artículo1. Autor          :=    Artículo2. Autor;
Artículo1. Título         :=    Artículo2. Título;
Artículo1. PubPeriódica   :=    Artículo2. PubPeriódica;
Artículo1. Página         :=    Artículo2. Página;
Artículo1. Año            :=    Artículo2. Año;
```

Como ocurre con otros tipos compuestos, excepto las cadenas, no podemos leer todo un registro a la vez; los procedimientos Read y ReadLn deben leer los registros uno por uno. Tal como ha sucedido en ocasiones anteriores, a continuación escribiremos nuestro propio procedimiento para leer un registro:

A. Lee del teclado datos bibliográficos de un artículo publicado. Se leen seis campos:

1. Nombre del autor o autores
2. Título del artículo
3. Nombre de la publicación
4. Número de volumen de la publicación
5. Número de página
6. Año

B. Otros procedimientos requeridos:

ReadLnCadena (lee un valor cadena 40)

```
PROCEDURE Leer ArtículoPublicado (VAR Artículo : ArtículoPublicado ) ;
BEGÍN

WriteLn ( ‘ Autor o Autores: (40 caracteres como máximo)’ );
ReadLnCadena (Artículo. Autor );

WriteLn (‘Título del artículo: 40 caracteres como máximo)’ );
ReadLnCadena (Artículo. Título );

WriteLn ( ‘Nombre de la publicación: (40 caracteres como máximo)’ );
ReadLnCadena ( Artículo. PubPeriódica ) ;
WriteLn (‘Número de volumen: (entero) ‘ ) ;
ReadLn (Artículo.Volumen ) ;

WriteLn ( ‘Número de página inicial: (entero)’ ) ;
ReadLn (Artículo. Página ) ;

WriteLn (‘Año: (entero de cuatro dígitos)’ );
ReadLn (Artículo.Año );

WriteLn

END; {fin de leerArtículoPublicado }
```

Como se indica en la documentación, este desarrollo da por hecho que contamos con un procedimiento ReadLnCadena para leer valores y colocarlos en una variable cadena 40. Podemos construir semejante procedimiento como parte de un TAD cadena 40.

Si deseamos escribir un registro, anotaremos los campos individualmente. Suponiendo que un registro ya recibió un valor por asignación, o mediante LeerArtículoPublicado, podremos exhibir su contenido con la ayuda de un procedimiento como el que mostramos a continuación (damos por hecho que ya disponemos de un procedimiento EscribirCadena para exhibir valores de tipo cadena 40):

A. Exhibe datos bibliográficos de un artículo publicado.

B. Otros procedimientos requeridos:

```
EscribirCadena (escribe un valor cadena 40)

PROCEDURE EscribirArtículoPublicado ( Artículo : ArtículoPublicado );
BEGIN

Write ( ‘ Autor(es) : ‘ ) ;
EscribirCadena (Artículo. Autor ) ;
```

```
WriteLn;

Write ( 'Título     :  ' ) ;
EscribirCadena (Artículo. Título );
WriteLn;

Write(' Publicación. ' ) ;
EscribirCadena (Artículo.PubPeriódica );
WriteLn,

Write ( 'Volumen   : ' ) ;
WriteLn( Artículo. Volumen: 4 ) ;

Write( ' Página          :   '  );
WriteLn ( Artículo. Página: 4 );

Writen( ' Año       :    '   );
WritenLn ( Artículo. Año: 4 ) ;
WriteLn

END; {fin de escribirArtículoPublicado}
```

Una muestra de este procedimiento se vería así:

```
Autor(es)     : Shaw. M
Título        : Abstracción Techniques in Mod Prog Langs
Publicación   : IEEE software
Volumen       : 1
Página        : 10
Año           : 1984
```

2.3.3. Combinaciones entre arreglos y registros

A menudo es útil construir arreglos cuyos elementos sean registros. Para ilustrar este caso, consideremos la creación de un arreglo bibliográfico que contenga referencias a publicaciones científicas del tipo que utilizamos en la sección pasada:

```
CONST
TamañoBiblio = 100;

TYPE
Cadena40 = PACKED ARRAY [1..40] of Char;

ArtículoPublicado =
```

```
RECORD
Autor     : cadena40;
Título    :cadena40;
PubPeriódica  : cadena40;
Volumen       : Integer;
Página         : Integer;
Año            : Integer;
END;

Bibliografía = ARRAY [1..TamañoBiblio ] OF ArtículoPublicado;
VAR
Biblio : Bibliografía;
```

Estas declaraciones describen un arreglo Biblio que contiene 100 registros del tipo ArtículoPublicado. Se trata de una declaración jerárquica porque el arreglo contiene registros que poseen arreglos y enteros. Los nombres de estos objetos reflejan la jerarquía:

Biblio es el nombre de todo el arreglo de registros.

Biblio [1] es el nombre del primer registro del arreglo Biblio.

Bibilio [1]. Autor es el nombre del primer campo del primer registro. Es una cadena de 40 caracteres.

Biblio [1]. Título es el nombre del segundo campo del primer registro. Es una cadena de 40 caracteres.

Biblio [1]. PubPeriódica es el nombre del tercer campo del primer registro. Es una cadena de 40 caracteres.

Biblio [1]. Volumen es el nombre del cuarto campo del primer registro. Es un entero.

Biblio [1]. Página es el nombre del quinto campo del primer registro. Es un entero.

Biblio [1]. Año es el nombre del sexto campo del primer registro. Es un entero.

Para llenar un arreglo de registros, debemos cubrir cada uno de ellos individualmente, lo que a su vez requiere llenar cada campo de registro en forma particular.

En Pascal, podemos crear toda clase de composiciones: registros que contienen otros registros, que a la vez tienen otros registros; arreglos de registros de arreglos de registros; y casi cualquier otra cosa que necesitemos. La única limitación es que un registro no puede contener registros de un mismo tipo, pues un tipo recursivo como ése nunca terminaría.

2.3.4. Arreglos de registros

Cuando se trabajó con las estructuras de datos, arreglos y registros, se hizo manipulando sus elementos individuales; nunca se laboró con el arreglo o con el registro completos, o con una sola operación. Por ejemplo, si se deseara buscar en un arreglo, se haría en cada uno de sus elementos individualmente, comparándolo con la clave deseada. Y si quisiéramos llenar un registro, ingresaríamos un valor a cada campo del registro por separado.

2.3.5. Registros anidados

Cuando se ejecuta una estructura cíclica como parte del cuerpo de otra, decimos que los ciclos están anidados. Hay dos clases de anidamiento: directo e indirecto. En el primer caso, la estructura cíclica interior es literalmente uno de los enunciados incluidos en el cuerpo de la estructura exterior. Y en el segundo caso, el cuerpo del ciclo exterior llama a un procedimiento o función que contiene la estructura cíclica interior.

2.4. Desarrollo de una aplicación que requiera de arreglos y registros

En este ejemplo, se muestra el proceso completo de diseño de un TAD. Se va a utilizar como objeto abstracto un conjunto de valores naturales en un rango dado:

- Objeto abstracto: conjunto de números naturales es un rango dado, no vacío.
- Nombre: Conjunto (sufijo de las operaciones: Conj)
- Formalismo: inf : $\{x_1, x_2,...,x_N\}$: sup
- Invariante: inf $\leq x_i \leq$ sup /* todos los elementos están en el rango [inf...sup]*/
  x_i ¡= x_k, ¡!= k /*no hay elementos repetidos */
  1$\leq$ inf $\leq$ sup /* el rango es válido */

- Constructoras: únicamente se requiere una constructora, que permita crear conjuntos vacíos, dado un rango de enteros.
  Conjunto crearconj (int infer, int super);
- Manejo de error: retorno de un objeto inválido
  Crearconj {error: inf <1 v sup < inf, crearconj = NULL}

- Modificadoras: son necesarias dos operaciones para alterar el estado de un conjunto. Una para agregar elementos y otra para eliminarlos. Estas dos operaciones son suficientes para simular cualquier modificación posible de un conjunto.
  Int insertarConj (conjunto conj, int elem);
  Int eliminarConj(conjunto conj, int elem);

- Manejo de error: Informe de fallas por código de retorno. Se seleccionan las siguientes constantes y códigos de error.

  0 OK Operación con éxito
  1 RANGO Elemento fuera del rango
  2 INEXIS Elemento inexistente
  3 DUPLI Elemento ya presente

  insertarConj {error: (elem < inf v elem> sup, insertarConj= RANGO)
  v
  (Ei / x_1= elem, insertarConj= DUPLI)}
  eliminarConj {error: elem! = X_i A_i, eliminarConj= INEXIS

- Analizadoras: la operación básica de consultar un conjunto es verificar si un elemento pertenece a él; adicionalmente, es necesario permitirle al cliente revisar los límites del rango de enteros que puede contener. Con estas tres operaciones, es posible extraer toda la información del conjunto:
  Int estaConj (Conjunto conj, int elem);
  Int inferiorConj (Conjunto Conj);
  Int superiorConj (Conjunto conj);
- Manejo de error: ninguna analizadora puede fallar.

- Operaciones interesantes: se colocan operaciones para copiar, comparar, visualizar y destruir.
  int igualConj (conjunto c1, conjunto c2); /* informa si c1==c2*/
  int subConj (conjunto c1, conjunto c2); /*informa si __ c2*/
  void imprimirConj (conjunto conj); /*presenta los elementos del conj*/
  void destruirConj (Conjunto conj); /*Destruye un conjunto*/
  También se agregan operaciones de amplio uso por parte de los clientes:
  Int cardinalidadconj (Conjunto conj); /*número de elementos de un conj*/
  Void unirConjunto (conjunto c1, conjunto c2); /*c1= c1Uc2*/

- Persistencia: lo usual es colocar dos operaciones para la persistencia. Una, para salvar un objeto en disco y otra para cargarlo de nuevo en memoria.
  Conjunto cargarconj (FILE fp); /*lee un conjunto de disco*/
  Void salvarconj (Conjunto conj, FILE fp**);** /***salva un conjunto en disco*/
  El formato exacto de la persistencia en un archivo se define en el momento de diseñar las estructuras de datos. El cliente no necesita esta información.

- Especificación de las operaciones:

Conjunto crearconj (int infer, int super)
/* crea un conjunto vacío con rango de valores potenciales [infer....super] */

{pre: 1 ≤ infer ≤ super}
{post: crearconj = infer: {} : super}

Int insertarconj (conjunto conj, int elem)
/* inserta al conjunto un elemento válido */

{pre: conj = inf: {x1, x2,...xN}: sup, elem¡= X¡ A¡, inf ≤ elem ≤ sup}
{post: conj = inf: {x1,x2,..., xN, elem} : sup}

Int eliminarconj (conjunto conj, int elem)
/* elimina un elemento del conjunto */

{pre: conj = inf: {x1, x2,..., xN}: sup, x¡ = elem}
{post: conj = inf: {x1,..., x¡-1, x¡+1,…, xN}: sup}

Int estaconj (conjunto conj, int elem)
/* informa si un elemento se encuentra en el conjunto */

{post: estaconjunto = E¡ /x¡ = elem)}

Int inferiorconj (conjunto conj)
/* retorna el límite inferior del rango de valores válidos del conjunto */

{post: inferiorConj = inf}

Int superiorConj (conjunto conj)
/* retorna el límite superior del rango de valores válidos del conjunto */

{post: superiorConj = sup}

Para saber más de este tema, consulta la bibliografía mencionada; además, acude a las páginas siguientes:

- www.dgbiblio.unam.mx
- www.i68.es
- www.paisvirtual.com (/informatica/programación/franccy)
- www.civila.com (/informatica/areaweb)
- http://serpiente.dgsca.unam.mx/rectoria/htm/wwwunam.html
- http://www.internet2.unam.mx/seminario_nov99/ponencias/bibliodig/biblioUNAM/RAMIREZAlejandro.html

Bibliografía

Horrmar, Elliot B., *Introducción al lenguaje y resolución de problemas con programación estructurada,* México, Addison-Wesley, 1986.

Joyanes, Aguilar L., *Programación básica para microcomputadoras,* México, McGraw-Hill, 1987.

Perry, Paul J, *La biblia del Turbo Pascal para Windows,* México, Anaya Multimedia América, 1993.

Salmon, William I., *Introducción a la computación con Turbo Pascal (5.0/5.5/6.0/TPW), estructura y abstracciones,* México, Addison-Wesley Iberoamérica, S. A., 1993.

Schneider, Maichel, *Introducción a la programación y solución de problemas con Pascal*, México, Limusa, 1986.

Tenebaun, Aarón M. y Augesein, Moshe J., *Estructura de datos en Pascal*, México, Prentice-Hall Hispanoamericana, 1987.

Materia: Informática II

Unidad 3
Estructuras dinámicas en memoria principal

Temario

- 3.1. Listas
  - 3.1.1. Definición del tipo de dato abstracto lista
  - 3.1.2. Representaciones de listas
  - 3.1.3. Definición de las operaciones sobre listas (especificación algebraica)
    - 3.1.3.1. Operación para construir de una lista vacía
    - 3.1.3.2. Operación para insertar un elemento a una lista
    - 3.1.3.3. Operación para revisar si una lista es vacía o no
    - 3.1.3.4. Operación para obtener la cabeza de una lista
    - 3.1.3.5. Operación para eliminar la cabeza de una lista
  - 3.1.4. Definición de la semántica de las operaciones sobre lista
  - 3.1.5. Cálculo de la longitud de una lista
    - 3.1.5.1. Operación para pegar dos listas
    - 3.1.5.2. Operación para invertir los elementos de una lista
    - 3.1.5.3. Funciones de orden superior
  - 3.1.6. Implantación de las operaciones sobre listas
  - 3.1.7. Implantación dinámica (mediante el uso de apuntadores)
    - 3.1.7.1. Listas generalizadas
- 3.2. Definición del tipo de dato abstracto lista generalizada
  - 3.2.1. Definición de las operaciones sobre listas generalizadas
  - 3.2.2. Operación para construir una lista generalizada
    - 3.2.2.1. Operación para obtener la cabeza de una lista generalizada
    - 3.2.2.2. Operación para revisar si la cabeza de una lista generalizada es un átomo o una lista generalizada
    - 3.2.2.3. Definición de la semántica de las operaciones sobre listas generalizadas
  - 3.2.3. Colas
  - 3.2.4. Definición del tipo de dato abstracto cola
- 3.3. Definición de las operaciones sobre colas

Introducción

En la presente unidad, estudiamos los elementos que intervienen en las estructuras en memoria principal, pilas, listas y colas; cómo se construyen, su inserción y consulta, y la localización y borrado de elementos individuales. Una pila es un caso especial de una lista lineal en la cual las operaciones de inserción (meter) y supresión (sacar) son estrictamente efectuadas sobre un extremo de la pila (tope de la pila). Las pilas suelen emplearse en la solución de problemas que necesitan de una estructura de datos tipo último-que-entra-primero-que-sale, y se utilizan para conservar el registro de las direcciones en orden de las llamadas. Además, las pilas pueden alojarse en arreglos (este enfoque presenta algunas restricciones, pero se utiliza frecuentemente). La integridad de una pila debe mantenerse cuando está alojada en arreglos.

Una cola es un caso especial de una lista lineal en la que las inserciones son restringidas a ocurrir sólo por un extremo de la cola (fondo de la cola), al igual que las eliminaciones (frente de la cola). Las colas se usan ampliamente en operaciones de computadora (por ejemplo, en la selección del siguiente trabajo a atender, el siguiente archivo a imprimir en la impresora o el siguiente paquete a procesar en una línea de teleproceso). Las colas son utilizadas para diversos propósitos (en modelos de simulación de tráfico, almacenes comerciales, oficinas de correo, etcétera). Una cola puede usarse en la solución de cualquier problema que necesita estructuras de datos de tipo primero que entra-primero que-sale. A menudo, los inventarios se comportan de esta manera.

Una lista ligada es una forma, no secuencial, de representar una estructura de datos lineal. Cada nodo de una lista ligada contiene por lo menos un apuntador al siguiente nodo. Las listas pueden aparecer de forma continua (ligada) y contigua (secuencial), ya que hay listas enlazadas, doble y triplemente enlazadas, o circulares. A menudo, las listas ligadas se implantan con nodos cabeza. Para simplificar los algoritmos que ejecutan inserciones y supresiones de las listas ligadas, se utilizan apuntadores que apuntan tanto al nodo anterior como al nodo siguiente, dando como resultado listas doblemente ligadas. Cada tipo de lista presenta ventajas y desventajas (pero todos tienen un tope y un valor final); por lo tanto, es necesario elegir el adecuado para una

aplicación específica.. El costo básico al usar listas ligadas es el espacio que se requiere para los apuntadores, además de su manipulación y manejo.

Objetivo

El alumno diferenciará entre las estructuras de datos estructuradas y las no estructuradas; asimismo aplicará el tipo de dato abstracto y la lista en usos prácticos.

Esbozo

En esta unidad, se estudiarán el uso, la definición y la estructura de las listas (que pueden ser de pila –*stock*– o cola) asociadas a un nombre (que ayudará en la rapidez de los valores ligados a las listas). Además, en las listas debe definirse un índice para acceder a un elemento en particular.

Siempre hay que tomar en cuenta que, en una lista, el tamaño de la memoria principal se define desde el principio; como consecuencia, sus elementos constitutivos se acomodarán en los espacios reservados en el arreglo, según sean capturados. Al llegar al límite de las localidades definidas, y si continúan registrándose datos, el compilador puede marcar error de desbordamiento. Otro punto a considerar es la forma de acceso a los elementos constitutivos: si los elementos llegan por un extremo de la lista y por ahí mismo salen (pedimento), hablamos de una pila; si ingresan por un extremo y salen por otro, estamos ante una lista de tipo cola. En el caso de las pilas, se emplea la expresión "el último en llegar, primero en salir" (UEPS o LIFO); y en las colas, "el primero en llegar, primero en salir " (PEPS o FIFO). Por otro lado, el tipo de datos llamado puntero guarda el contenido de una dirección de memoria y se expresa como P^, a éste también podemos asignarle un número (dirección de memoria): P^: = 3600 (en Pascal); P* = 3600 (en lenguaje C). Al signar la dirección de memoria 3600, desconocemos el contenido de dicha localidad; entonces, necesitamos definir una variable para desplegar el contenido (valor) de la localidad de memoria referida.

Por definición, un tipo de dato puntero es entero ya que una dirección de memoria es un número entero.

Una vez vistos el manejo de las listas –ya sean pilas o colas– y la necesidad de utilizar el puntero en la asignación de las localidades de memoria, podremos analizar las

operaciones sobre las listas (construcción, inserción, consulta y localización de elementos individuales y el borrado de sus elementos).

Las estructuras dinámicas de datos "crecen" a medida que se ejecuta un programa. Una estructura dinámica de datos es una colección de elementos (nodos) que son normalmente registros. Además, son extremadamente flexibles y posibilitan añadir nueva información creando un nuevo nodo e insertándolo entre nodos ya existentes. (Una estructura dinámica de datos puede modificar su estructura, ampliar o limitar su tamaño mientras se ejecuta el programa).

3.1. Listas

Una lista lineal es un conjunto de elementos de un tipo dado que se encuentren ordenados (pueden variar en número). Los elementos de una lista se almacenan normalmente de manera contigua (un elemento detrás de otro) en posiciones consecutivas de la memoria.

Una lista enlazada es un conjunto de elementos que contienen la posición –o dirección– del siguiente. Cada elemento de una lista enlazada debe tener al menos dos campos: uno con el valor del elemento y otro (*link*) que contiene la posición del siguiente elemento o encadenamiento:

Dato (valor elemento)	Enlace

3.1.1. Definición del tipo de dato abstracto lista

Se define una lista como una secuencia de cero o más elementos de un mismo tipo. El formalismo escogido para representar este tipo de objeto abstracto es:

<e1,e2,.....,en>

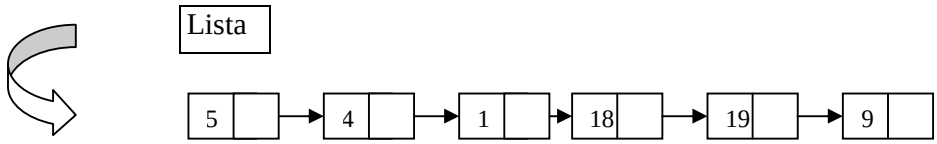
Cada ejemplo modela un elemento del agrupamiento. Así, *e1* es el primero de la lista; *en,* el último; y la lista formada por los elementos <e2,e3,...,*en* corresponde al resto de la lista inicial.

3.1.2. Representaciones de listas

Las listas enlazadas tienen una terminología propia que suelen utilizar normalmente. Los valores se almacenan en un nodo cuyos componentes se llaman campos. Un nodo tiene al menos un campo de dato o valor y un enlace (indicador o puntero) con el siguiente campo. El campo enlace apunta (proporciona la dirección de) al siguiente nodo de la lista. El último nodo de la lista enlazada, por un convenio, suele representarse por un enlace con la palabra reservada *nil* (nulo), una barra inclinada (/) y, en ocasiones, el símbolo eléctrico de la tierra o masa.

5	4	1	7	18	19	9	7	45	...		

(a) array representado por una lista



(b) lista enlazada representada por una lista de enteros

3.1.3. Definición de las operaciones sobre listas (especificación algebraica)

Las operaciones que pueden realizarse con listas lineales contiguas son:

1. Insertar, eliminar o localizar un elemento.
2. Determinar el tamaño de la lista.
3. Recorrer la lista para localizar un determinado elemento.
4. Clasificar los elementos de la lista en orden ascendente o descendente.
5. Unir dos o más listas en una sola.
6. Dividir una lista en varias sublistas.
7. Copiar la lista.
8. Borrar la lista.

Operaciones que normalmente se ejecutan con las listas enlazadas:

1. Recuperar información de un nodo especificado.
2. Encontrar un nodo nuevo que contenga información específica.
3. Insertar uno nodo nuevo en un lugar específico de la lista.
4. Insertar un nuevo nodo en relación con una información particular.
5. Borrar un nodo existente que contenga información específica.

3.1.3.1. Operación para construir una lista vacía

1. Leer longitud de la lista L;
2. Si L = 0, visualizar “error lista vacía”;
si no, comprobar si el elemento j-esimo está dentro del rango permitido de los elementos 1< = j<L; en este caso, asignar el valor del elemento P(j) a una variable B;
si el elemento j-esimo no está dentro del rango, visualizar un mensaje de error elemento solicitado no existe en la lista.

3. Fin

El pseudocódigo correspondiente es:

```
Inicio
LeerL                    (longitud de la lista)
Si L = 0
```

Entonces, visualizar la lista vacía.
Si no, si 1 < = j < = L.
Entonces, B ⟵ P (j)
Si no, visualizar elemento no existente.
Fin-si
Fin-si
Fin

Lista inicLista(void)
/ * Crea y retorna una lista vacía * /
{ post:inicLista =< > □ }

3.1.3.2. Operación para insertar un elemento a una lista

Sea Lista una lista enlazada. Se desea insertar en ella un nodo N que debe ocupar un lugar después del elemento X, o bien entre dos nodos, A y B. La inserción presente dos casos particulares:

1. Insertar el nuevo nodo en el frente –principio– de la lista.
2. Insertar el nuevo nodo al final de la lista, con lo que este nodo deberá contener el puntero nulo.

1. El algoritmo siguiente inserta un elemento (ELEMENTO) en el principio de la lista, apoyándose en un puntero nuevo.

Algoritmo inserción
Comprobación de desbordamiento
Si DISPO=NULO
Entonces, escribir “desbordamiento”
Si no:
NUEVO ⟵ DISPO
DISPO ⟵ P (DISPO)
INFO (NUEVO) ⟵ ELEMENTO
P (NUEVO) ⟵ PRIMERO
PRIMERO ⟵ NUEVO

2. Fin_si

3. Inserción a continuación de un nodo especifico

El algoritmo para insertar un elemento (por ejemplo, NOMBRE) en una lista enlazada a continuación del nodo específico P pasa por obtener un puntero auxiliar Q, y otro llamado NUEVO.

Algoritmo correspondiente:

1. NUEVO ← OBTENERNOMBRE
2. INFO(NUEVO) ← NOMBRE
3. Q ← SIG (P)
4. SIG (P) ← NUEVO
5. SIG (NUEVO) ← Q

3.1.3.3. Operación para revisar si una lista es vacía o no

Esta operación se realiza cuando utilizamos el algoritmo búsqueda. El algoritmo correspondiente de la búsqueda del elemento o nodo n de una lista se detalla a continuación.

Algoritmo búsqueda:

1. Si la lista está vacía, escribir un mensaje de error.
2. En caso contrario, la lista no está vacía y el algoritmo la seguirá recorriendo hasta encontrar algún elemento en la búsqueda.

El pseudocódigo correspondiente es:

```
Algoritmo búsqueda
Inicio
Leer PRIMERO
Si PRIMERO = nil
Entonces, escribir "lista vacía"


Si no:
Q ← 1
...
```

NOTA: una lista enlazada sin ningún elemento se llama lista vacía; es decir, la representación de una lista vacía si su puntero inicial o de cabecera tiene el valor nulo (*nul*).

3.1.3.4. Operación para obtener la cabeza de una lista

En este caso, se utiliza una variable (cabecera) en las listas enlazadas para apuntar el primer elemento.

El algoritmo siguiente inserta un elemento (ELEMENTO) al principio de la lista (CABECERA).

```
Algoritmo        inserción
{Comprobación de desbordamiento}
Si DISPO = NULO
Entonces, escribir "desbordamiento"
Si no:
NUEVO ← DISPO
```

DISPO ← P(DISPO)
INFO (NUEVO) ← ELEMENTO
P(NUEVO) ← PRIMERO
PRIMERO ← NUEVO

Fin_si

En una lista circular, el algoritmo para insertar un nodo X al frente de una lista circular es:

NUEVO ← NODO
INFO (NUEVO) ← X
SIG (NUEVO) ← SIG(CABECERA)
SIG (CABECERA) ← NUEVO

3.1.3.5. Operación para eliminar la cabeza de una lista

El algoritmo que elimina de la lista enlazada el elemento siguiente apuntado por P utiliza un puntero auxiliar Q, y se establece primero para apuntar al elemento que se elimina:
1. Q ← SIG (P)
2. SIG (P) ← SIG(Q)
3. LIBERAR NODO (Q)

3.1.4. Definición de la semántica de las operaciones sobre lista

Para procesar una lista enlazada se necesita conocer:

- Primer nodo (cabecera de la lista).
- El tipo de sus elementos.
- El tamaño de la lista (la definición de sus elementos o nodos).

Para definir las operaciones de una lista, se debe especificar la variable que contiene al primer nodo (cabecera), que en este caso llamaremos PRIMERO.

PRIMERO Valor del índice u orden del elemento de los *arrays*.

DATOS Que ocupa el primer lugar de la lista, es vacía.

I Variable que representa el índice de los *arrays*.

DATOS (i) Elemento i de la lista.

ENLACE (i) Valor del índice (orden) del *array*, donde se encuentra el siguiente elemento de la lista.

ENLACE (f) Último elemento de la lista (su valor es cero).

3.1.5. Cálculo de la longitud de una lista

La longitud de una lista se define como el número de elementos que la componen. Si no tiene ningún elemento, se encuentra vacía y su longitud es 0. Esta estructura se representa mediante la notación <>, y se considera simplemente como un caso particular de una lista con cero elementos.

La posición de un elemento dentro de una lista es el lugar que ocupa dentro de la secuencia de valores que componen la estructura. Es posible referirse sin riesgo de ambigüedad al elemento que ocupa la i-esima posición dentro de la lista, y hacerlo explícito en la representación mediante la notación:

1 2 i n
<e1, e2,...ei...,en>

Esta notación indica que e1 es el elemento que se encuentra en la posición i de la lista y que dicha lista consta de n elementos. Esta extensión del formalismo sólo se utiliza cuando se hace referencia a la relación entre un elemento y su posición (entero positivo, menor o igual al numero total de elementos de una lista).

3.1.5.1. Operación para pegar dos listas

```
#define INTGR        1
#define FLT          2
#define STRING       3
struct node {
        int etype: /*etype es igual a INTGR, FLT o STRING */
                   /*  dependiendo del tipo del                    */
                   /*   elemento correspondiente                   */
         union {
            int ival;
            float fval;
            char *pval; /*apuntador a una cadena  */
          }element;
           struct node *next;
};
```

3.1.5.2. Operación para invertir los elementos de una lista

Un algoritmo para invertir los elementos de una lista ligada implica pasar a través de cada nodo de la lista para cambiar todos los apuntadores, de tal manera que el último se convierta en el primer elemento, y el que era el primero se convierta en el último.
El algoritmo se muestra en la siguiente figura (en lenguaje COBOL):

```
01 APUNTADORES AUXILIARES.
      02 ESTE-NODO              PIC 9(3)
      02 NODO-ANTERIOR          PIC9(3)
      02 NODO-SIGUIENTE         PIC9(3)
INVERSION-DE-LA LISTA.
      IF PRIMER-NODO NOT =0
      THEN COMPUTE ESTE-NODO =PRIMER-NODO
```

```
        COMPUTE NODO-SIGUIENTE = SIG-NODO (ESTE-NODO)
        COMPUTE SIG-NODO (ESTE-NODO) =0
        PERFORM BUSCA-NODO UNTIL (NODO-SIGUIENTE =0)
        COMPUTE PRIMER-NODO = ESTE-NODO.
BUSCA-NODO.
        COMPUTE NODO-ANTERIOR = ESTE-NODO
        COMPUTE ESTE-NODO = NODO-SIGUIENTE
        COMPUTE NODO SIGUIENTE <0 SIG-NODO (ESTE-NODO)
         COMPUTE SIG-NODO (ESTE-NODO) = NODO-ANTERIOR.
```

3.1.5.3. Funciones de orden superior

Otras operaciones interesantes pueden enriquecerse con operaciones de manejo de persistencia y destrucción, de acuerdo con lo siguiente:

TAD Lista { tipL }

Void destruir Lista (Lista lst)
/ * destruye el objeto abstracto, retornando toda la memoria ocupada por éste * /
{ post: la lista 1st no tiene memoria reservada }

Lista cargar Lista (File *fp)
/ * construye una lista a partir de la información de un archivo * /

{ pre: el archivo está abierto y es estructuralmente correcto, de acuerdo con el esquema de persistencia }

{ post: se ha construido la lista que corresponde a la imagen de la información del archivo }

Void salvar Lista (Lista lst, FILE*fp)

/ * Salva la lista en un archivo * /

{ pre: el archivo está abierto }

{ post: se ha hecho para persistir la lista en el archivo, la ventana de la lista está indefinida }

Además, se puede traer de la memoria secundaria una lista, modificar su contenido eliminando todas las ocurrencias de un valor dado y hacer persistir de nuevo la lista resultante. La complejidad es 0 (n), donde n es la longitud de la lista. Se debe tener en

cuenta que la constante asociada con esta función es muy alta, dado el elevado costo en tiempo que tiene el acceso a la información en memoria secundaria.

```
Void   actualizar Lista (char nombre[], tipo L val)
FILE * fp= fopen (nombre, “r”)
Lista 1st=cargarLista(fp)
Fclose (fp) ;
ElimtodosLista(2st,val);
Fp=fopen (nombre, “w”);
SalvarLista(1st,fp);
Fclose (fp);
DestruirLista(1st);
```

Para cada uno de los objetos abstractos temporales que se utilicen en cualquier función, es necesario llamar la respectiva operación de destrucción.

3.1.6. Implantación de las operaciones sobre listas

Es una implantación con base en arreglos que permite visualizar_ lista y buscar en un tiempo lineal (la operación buscar_ K_ simo lleva un tiempo constante). La inserción y eliminación son costosas (no obstante, todas las instrucciones pueden implantarse simplemente con el uso de un arreglo). Por supuesto, en algunos lenguajes tiene que haberse declarado el tamaño del arreglo en tiempo de compilación; y en lenguajes que permiten que un arreglo sea asignado “al vuelo”, el tamaño máximo de la lista. Regularmente, esto exige una sobrevaloración, que consume espacio considerable.

3.1.7. Implantación dinámica (mediante el uso de apuntadores)

Los dos aspectos importantes en una implantación con apuntadores de las listas enlazadas son:

1. Los datos se almacenarán en una colección de registros. Cada registro contiene los datos y un apuntador al siguiente registro.
2. Se puede obtener un registro nuevo de la memoria global del sistema por medio de una llamada a *new* y liberar con una llamada *dispose.*

La forma lógica de satisfacer la condición 1 es tener un arreglo global de registros. Para cualquier celda de arreglo, puede usarse su índice en lugar de una dirección.

```
Type
Ap_nodo = integer
Nodo = record
Elemento  : tipo_elemento
Siguiente: ap_nodo

End;
LISTA = ap_nodo
Posición = ap_nodo
```

Var ESPACIO _CURSOR :array [0...TAMAÑO_ ESPACIO] of nodo.

Para escribir las funciones de una implantación por cursores de listas enlazadas, hay que pasar y devolver los mismos parámetros correspondientes a la implantación con apuntadores.

3.1.7.1. Lista generalizadas

Cuando las estructuras de datos se vuelven más complejas y proporcionan más posibilidades al usuario, las técnicas de manejo aumentan también su grado de dificultad. Así, la lista generalizada es un método de implantación de tipo de datos abstractos, en donde varía el tipo de elementos.

3.2. Definición del tipo de dato abstracto lista generalizada

Es posible ver una lista como un tipo de datos abstractos por derecho propio. Como tipo de dato abstracto, una lista sólo es una secuencia simple de objetos llamados elementos. Asociado a cada elemento, hay un valor. Hacemos una distinción muy especifica entre un elemento (el objeto como parte de una lista) y el valor del elemento (el objeto considerado de manera individual).

3.2.1 Definición de las operaciones sobre listas generalizadas

Definiremos ahora una serie de operaciones abstractas cuyas propiedades lógicas son las operaciones *head* y *tail*, que no están definidas si su argumento no es una lista. Una sublista de una lista R es una lista que resulta de aplicación de cero o más operaciones *tail* a l.

3.2.2. Operación para construir una lista generalizada

Debido a que los nodos de lista generales pueden contener elementos de datos simples de cualquier tipo, o apuntadores a otras listas, la forma más directa de declarar nodos de lista es utilizando uniones. La siguiente es una implementación posible:

```
#define INTGR1
#define CH 2
#define LST 3
Struct nodetype {
 Int utype;                  //utype es igual a INTGR, CH, o LST
 Union}
 Int intgrinfo;              //utype = INTGR
 Char charinfo;              //utype = CH
 Struct nodetype *lstinfo;   //utype = LST
 }info;
 struct nodetype *next;
```

```
};
typedef struct nodetype *NODEPTR;
```

3.2.2.1. Operación para obtener la cabeza de una lista generalizada

Para hacer este proceso, el algoritmo se vale de la operación *push* para agregar un nodo al frente de la lista.

```
Q = null ;
For   (p = list ; p ¡ = null & & x >
            Info (p) ;  p = next (p) )
  q = p ;
/* en este momento, un nodo que entrega x deberá insertarse if (q = = null) /* insertar x
a la cabeza de lista.
 push  (list, x)  ;
else
    insafter    (q, x)
```

3.2.2.2.Operación para revisar si la cabeza de una lista generalizada es un átomo una lista generalizada

Un nodo atómico no contiene apuntadores, sólo un elemento de datos simple**.** Existirían varios tipos diferentes de nodos atómicos, si cada uno de los elementos únicos de datos correspondiera a uno de los tipos de datos legales. Un nodo de lista comprende un apuntador a un nodo atómico y un indicador de tipo que señala la clase de nodo atómico a la que apunta (así como un apuntador al siguiente nodo de la lista, por supuesto). Cuando es necesario colocar un nodo nuevo en una lista, debe asignarse un nodo atómico del tipo apropiado y de su valor, y deben establecerse el campo de la información del nodo de lista para que apunte al nuevo nodo atómico y el tipo de campo correcto.

```
switch (typecode) {
case t1;     //hacer algo con node1
case t2;      //hacer algo con node2
          …
case t10;   //hacer algo con node10
}             //fin de switch
```

3.2.2.3. Definición de la semántica de las operaciones sobre listas generalizadas

Debido a que los nodos de lista generales pueden contener elementos de datos simples de cualquier tipo o apuntadores a otras listas, la forma más directa de declarar nodos es utilizando uniones. La siguiente es una implementación posible.

```
struct nodetype {
        int utype;        /* utype es igual a INTGR, CH, o LST */
```

```
        union {
        int intgrinfo;                  /* utype = INTGR */
        char charinfo;                  /* utype = CH    */
        struct nodetype *lstinfo;       /* utype = LST   */
}info;
struct nodetype *next;
};
typedef struct nodetype *NODEPTR;
```

3.2.3. Colas

Son otro tipo de estructura lineal de datos similares a las pilas, pero se diferencia de éstas en el modo de insertar y eliminar elementos.

3.2.4. Definición del tipo de dato abstracto cola

La definición de una cola puede simplificarse si se utiliza una lista enlazada circular, ya que, en este caso, sólo se necesita un puntero.

3.3. Definición de las operaciones sobre colas

Las operaciones que se pueden realizar con una cola son:

1. Acceder a su primer elemento.
2. Añadir un elemento al final.
3. Eliminar su primer elemento.
4. Vaciarla.
5. Verificar su estado.

3.3.1. Operación para construir una cola vacía

¿Cómo puede representarse una cola en PASCAL? Lo primero que haríamos es utilizar un arreglo que posea los elementos de la cola, y dos variables –frente y atrás– que contengan las posiciones del arreglo correspondiente al primero y al último elementos de la cola. Por tanto, una cola de enteros podría ser declarada por:

```
    const   maxqueue = 100;
    type    queue    =  record
    ítems: array [1.. maxqueue] of integer;
    front, rear: 0..maxqueue
  end;
```

Por supuesto, al emplear un arreglo para que contenga los elementos de una cola, es posible que se presente un sobreflujo (*overflow*) si ésta posee más elementos de los que fueron reservados en el arreglo. Ignorando la posibilidad de bajoflujo y sobreflujo por el

momento, la operación *insert* (q,x) se podría implementar mediante las declaraciones siguientes:

```
q.rear: = q.rear + 1;
q.ítems [q.rear]: = x
```

y la operación x: = remove (q) podría ser implementada por medio de:

```
x: = q.items [q.front];
q.front: = q.front + 1
```

Inicialmente, q.rear se ajusta a 0 y q.front se ajusta a 1; y se considera que la cola está vacía cada vez que q.rear < q.front. En la cola, el número de elementos en cualquier momento es igual al valor de q.rear – q.front + 1.

Examinemos ahora qué sucede al utilizar esta representación. La figura 1.2 ilustra un arreglo de cinco elementos utilizados para representar una cola (maxqueue = 5). Inicialmente, la cola está vacía (figura 1.2a). En la figura 1.2b, se han agregado o insertado los elementos A, B y C; en la 1.2c, retirado dos elementos; y en la 1.2d, se han insertado:

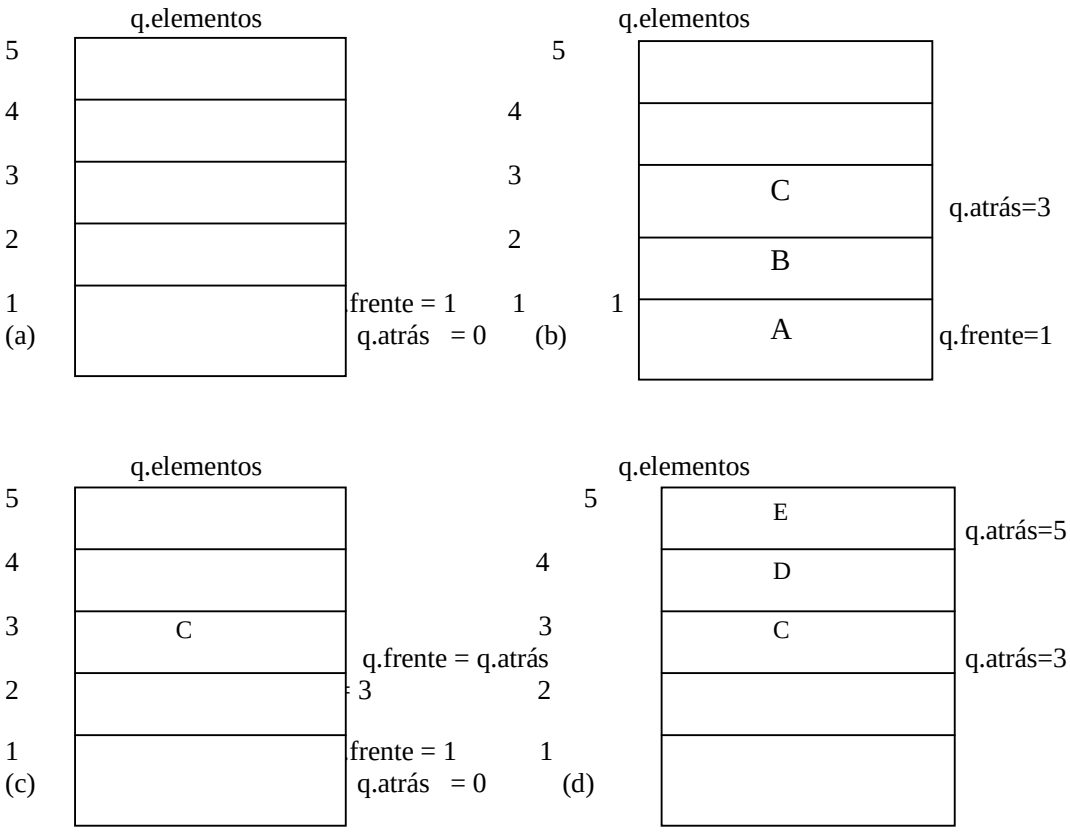


Figura 1.2

q.elementos q.elementos

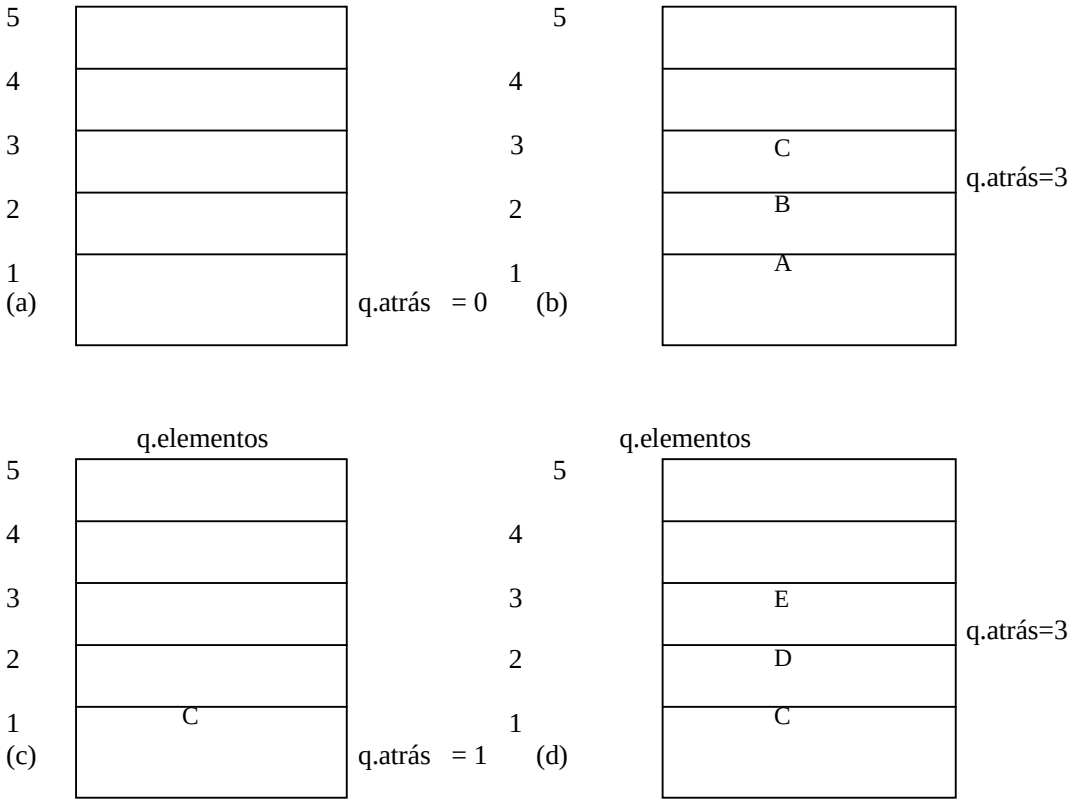


Figura 1.3

Tenemos dos elementos nuevos, D y E. El valor de q.front es 3 y el de q.rear, 5; así, hay solamente 5 – 3 + 1 = 3 elementos en la cola. Ya que el arreglo contiene cinco elementos, debe haber espacio para que la cola se expanda sin temor a un sobreflujo. Sin embargo, para insertar el elemento F, q.rear debe ser incrementado de 1 a 6 y q.items [6] debe ser ajustado al valor de F. Pero q.items es un arreglo de solamente 5 elementos: se puede hacer la inserción de este nuevo elemento. Es posible que la cola esté vacía, y sin embargo no se pueda agregar un nuevo elemento; entonces, debe estudiarse la posibilidad de realizar una secuencia de inserciones y eliminaciones hasta solucionarlo. Es claro que la representación del arreglo en la forma presentada anteriormente es inaceptable.

Una solución a este problema consiste en modificar la operación *remove* de tal manera que cuando se retire un elemento, se desplace toda la cola al principio del arreglo. La operación x: = *remove* (q) podría entonces ser modificada (nuevamente ignorando la posibilidad de bajoflujo) a:

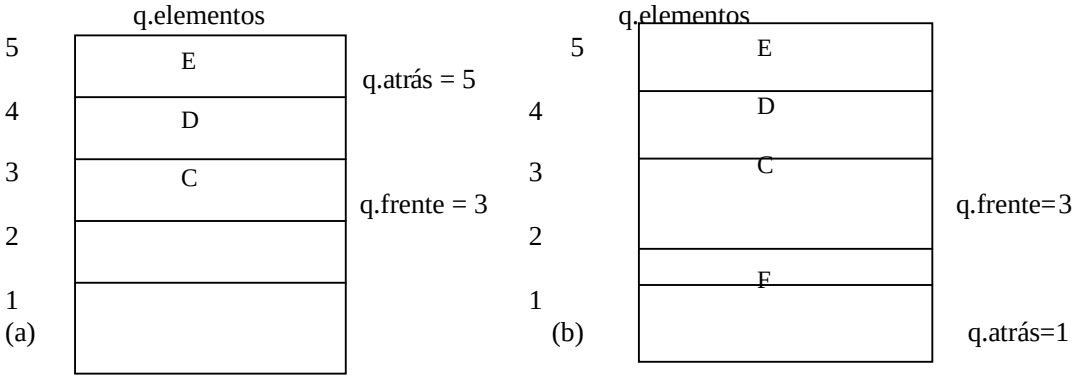
```
x: = q.items[1];
for i: 1 to q.rear – 1
    do q.items [i]: = q.items [i+1];
q.rear: = q.rear -1
```

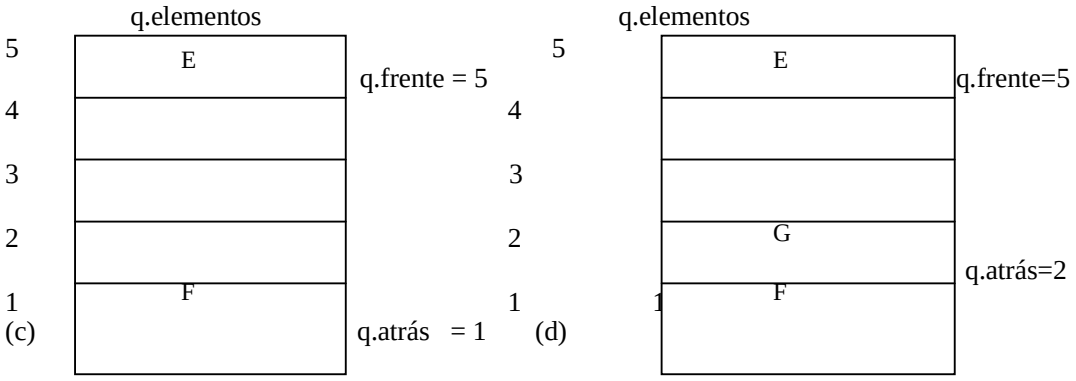
En este caso, el campo *front* (frente) ya no se especifica como parte de la cola, porque su frente es siempre el primer elemento del arreglo. La cola vacía está representada por

la cola, en la cual *rear* (atrás) es igual a 0. Con esta nueva representación, la figura 1.3 presenta la cola de la figura 1.2.

Sin embargo, el método anterior es poco satisfactorio. Cada retiro implica mover todos los elementos restantes de la cola. Es decir, si una cola contiene 500 ó 1.000 elementos, claramente se ve lo costoso de la operación. Además, el proceso de retirar un elemento de una cola lógicamente equivale a manipular solamente un elemento (el que actualmente se encuentra al frente de la cola). Ésta no debe incluir otras operaciones extrañas a ella.

Otra solución es considerar al arreglo que contiene la cola como un círculo en lugar de una línea recta. Es decir, nos podemos imaginar que el primer elemento del arreglo está inmediatamente después del último. Esto implica que, aun si el último elemento está ocupado, puede insertarse un nuevo valor detrás de éste como primer elemento del arreglo, siempre y cuando esté vacío.





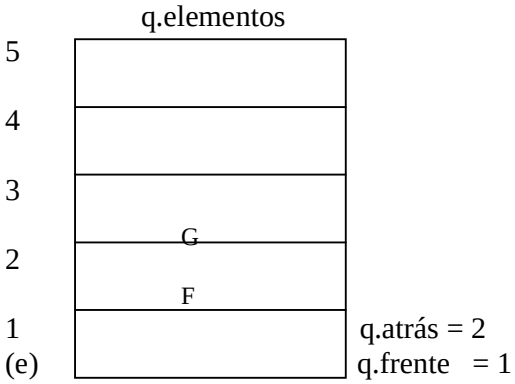


Figura 1.4

Consideremos un ejemplo. Asumamos que una cola contiene tres elementos en las posiciones 3, 4 y 5 de un arreglo de cinco elementos. Esta es la situación de la figura 1.2d, reproducida como la figura 1.4a. Aunque el arreglo no está completamente lleno, el último elemento del arreglo está ocupado. Si se intenta insertar el elemento F en la cola, éste puede colocarse en la posición 1 del arreglo, tal como se muestra en la figura 1.4b. El primer elemento de la cola está en q.items [3], que está seguido en la cola por q.items [4], q.items [5] y q.items [1]. Las figuras 1.4c-e muestran el estado de la cola cuando los dos primeros elementos, C y D, son retirados, G, insertado y, finalmente, E, retirado.

3.3.2. Operación para insertar un elemento a una cola

La operación *insert* se encarga del sobreflujo (ocurre cuando, a pesar de que todo el arreglo está ocupado por elementos de la cola, se intenta insertar otro elemento). Por ejemplo, consideremos el caso de la cola de la figura 1.5.a; en ésta hay tres elementos, C, D y E, en q.items [3], q.items [4] y q.items [5], respectivamente. Puesto que el último elemento de la cola ocupa la posición q.item [5], q.rear = 5. Y ya que el primer elemento de la cola está en q.item [3], entonces q.front es igual a 2. En las figuras 1.5.b y c, se presenta el caso en que los elementos F y G han sido adicionados a la cola y el valor de q.rear ha cambiado de acuerdo con esa nueva adición; en este momento, el arreglo está lleno y cualquier intento que se haga por insertar más elementos causará sobreflujo. Pero esto está marcado por el hecho de que q.front = q.rear (indicación de un bajoflujo). Es decir, bajo esta aplicación, aparentemente no hay forma de distinguir entre una cola vacía y una llena. Entonces, esta clase de situación no es satisfactoria.

Una solución a este problema consiste en sacrificar un elemento del arreglo y permitir que la cola crezca solamente hasta igualar el tamaño del arreglo menos 1. Es decir, que un arreglo de 10 elementos es declarado como una cola (ésta puede tener hasta 99 miembros). Cualquier intento que se haga por insertar un elemento 100 en la cola ocasionará sobreflujo. La rutina *insert* puede escribirse como sigue:

```
procedure insert (var q: queue; x: integer);
begin
```

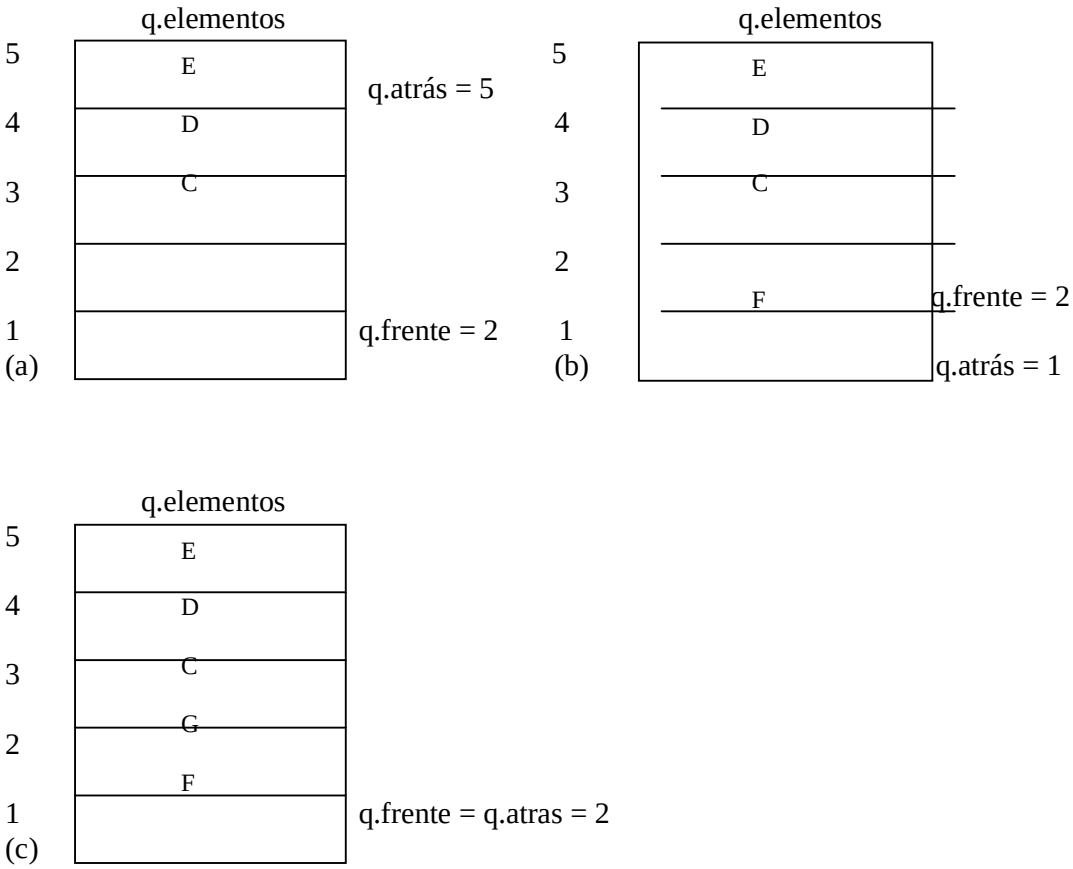


Figura 1.5

```
with q
    do begin
            if rear = maxqueue
               then rear: = 1
               else rear: = rear + 1;
             if rear = front
               then error ('sobreflujo')
               else items [rear]: = x
          end { termina with q do begin}
end { termina procedure insert}
```

La evaluación para sobreflujo en la rutina *insert* se presenta después de que se le ha asignado el valor a q.rear, mientras que la prueba para bajoflujo en la rutina de *remove* inmediatamente al entrar la rutina, antes de actualizar q-front[1].

La cola se implementará con un *array* global de dimensión n (n = longitud máxima).
La cola se define con el *array.*

Cola = cola1, cola2, cola n.
El algoritmo de inserción lo definimos como un procedimiento. Además, es preciso verificar que en la pila no se producirá error de desbordamiento.

3.3.2.1. Operación para revisar si una cola está vacía

Desafortunadamente, es difícil bajo esta representación determinar cuándo está vacía la cola. La condición q.rear < q.front ya no es válida para probar si la cola no es vacía (las figuras 1.4b-d ilustran las situaciones en las cuales la condición es verdadera, aun así, la cola no es vacía).

Una manera de resolver este problema consiste en establecer la convención en la cual el valor de q.front (q.frente) es el índice del elemento del arreglo inmediatamente que precede al primer elemento de la cola, en lugar de ser el índice del primer elemento en sí. Por tanto, puesto que crear (q.atrás) contiene el índice del ultimo elemento de la cola, la condición q.front = q.rear implica que la cola está vacía.

Una cola de enteros puede ser declarada e inicializada con:

```
const. mxqueue = 100;
type queue = record
                items: array[1..maxqueue] of integer;
                front,rear: 1..maxqueue
             end;
var   q: queue;
begin
      q.front: = maxqueue;
      q.rear :  = maxqueue
```

Observemos que q.front y q.rear son inicializados con el último índice del arreglo, en lugar de 0 ó 1, porque el último elemento del arreglo precede inmediatamente al primero dentro de la cola bajo esta representación. Puesto que q.rear = q.front., la cola está inicialmente vacía.

La función *empty* puede codificarse como:

```
function empty (q.queue): = bolean;
begin
      with q
            do if front = rear
                  then empty: = true
```

```
                    else empty: = false
    end {termina function empty};
```

La operación *remove* (q) puede ser codificada como:

```
    function remove (var q: queue): integer;
    begin
        if empty (q)
            then error(‘error(‘bajoflujo en la cola’)
            else with q
                do begin
                    if front = maxqueue
                        then front: = 1
                        else front: = front + 1;
                            remove: = items[front]
                        end {termina else with y do begin}
        end { termina function remove};
```

Observemos que q.front debe ser actualizado antes de extraer algún elemento.

Por supuesto, a menudo una condición de bajo flujo tiene significado y puede utilizarse como una señal dentro de la fase de procesamiento. Si queremos, podemos utilizar un procedimiento remvandtest que sería declarado por:

```
    Procederé remvandtest (var q: queue; var x: integer; var und: booleano);
```

Esta rutina asigna a und el valor de falso y x al elemento que ha sido removido de la cola, si la cola no está vacía y asigna a und el valor de verdadero si ocurre bajoflujo.

3.3.2.2. Operación para obtener el elemento que está al frente de la cola

```
Quitar (Quitar (X,Q))
Elimina y devuelve el frente de la cola.
Procedure Quitar (var x: Tipoelemento; var Q: (cola);
  Procedure desplazar;
  Var I: Posición;
   Begin
    End;
Begin
    X:= Q. Datos (Enfrente)
```

3.3.2.3. Operación para remover el elemento que está al frente de la cola

```
Tipo (info Cola (cola. Col)
/*Elimina el primer elemento de la cola*/
pre: n>0
post: col=x2...xn
```

3.3.2.4 Definición de la semántica de las operaciones sobre colas

Hay tres operaciones rudimentarias que pueden aplicarse a una cola: *insert(q, x)*, que inserta un elemento en el final de una cola *q*; *x=remove(q),* que borra un elemento del frente de la cola *q*; y *empty(q)*, que dará como resultado *false* o *true*, según si la cola tiene o no elementos.

Una cola es otra forma de lista de acceso restringido. A diferencia de las pilas, en las que tanto las inserciones como las eliminaciones tienen lugar en el mismo extremo, las colas restringen todas las inserciones a un extremo y todas las eliminaciones al extremo opuesto.

Para implantar una cola en la memoria de una computadora, podemos hacerlo con un bloque de celdas contiguas en forma similar a como almacenamos las pilas; pero dado que debemos efectuar operaciones en ambos extremos de la estructura, es mejor apartar dos celdas para usarlas como punteros, en lugar de una sola.

3.3.2.5. Implantación dinámica de las operaciones sobre colas

Una posibilidad para implantar colas consiste en usar un arreglo para guardar los elementos de la cola y emplear dos variables, front y rear, para almacenar las posiciones en el arreglo de último y el primer elementos de la cola dentro del arreglo. En este sentido, es posible declarar una cola de enteros mediante:

```
*define MARQUEVE 100.
Struct queve
        Int items (marqueve);
        Int front, rear;
```

3.3.3. Colas con prioridades

Tanto la pila como la cola son estructuras de datos cuyos elementos están ordenados con base en la secuencia en que se insertaron. La operación *pop* recupera el último elemento insertado, en tanto que *remove* toma el primer elemento que se introdujo. Si hay un orden intrínseco entre los elementos (por ejemplo, alfabético o numérico), las operaciones de la pila o la cola lo ignoran.

La cola de prioridad es una estructura de datos en la que el ordenamiento intrínseco de los elementos determina los resultados de sus operaciones básicas. Hay dos tipos de cola de prioridad: ascendente y descendente. La primera es una colección de elementos en la que pueden insertarse elementos de manera arbitraria y de la que puede eliminarse sólo el elemento menor (si *apq* es una cola de prioridad ascendente, la operación *pqinsert(apq, x)* introduce el elemento *x* dentro de *apq,* y *pqmindelete(apq)* elimina el elemento mínino de *apq* y regresa a su valor). La segunda es similar a la anterior, pero sólo permite la eliminación del elemento *mayor.* Las operaciones aplicables a una cola de este tipo, *dpq*, son *pqinsert(dpq, x)* y *pqmaxdelete(dpq). pqinsert(dpq,x)* inserta el elemento *x* en *dpq* y es idéntica desde un punto de vista lógico a *pqinsert* en el caso de

una cola de prioridad ascendente. Por último, *pqmaxdelete(dpq)* elimina el elemento máximo de *dpq* y regresa a su valor.

3.3.4. Definición del tipo de dato abstracto una cola con prioridades

La representación de una cola como un tipo de datos abstracto es directa. El *type* se usa para denotar el tipo de elementos de la cola y *parametrizar* el tipo de cola:

abstract typedef << eltype>> QUEUE (el type);

abstract empty (q)
QUEUE(el type) q;
Postcondition *empty==(len (q) == 0);*

Abstract eltype remove (q)
QUEUE (eltype) q;
precondition *empty (q) == FALSE;*
postcondition *remove == first (q');*
q == sub(q', 1, len(q') – 1);

abstract insert (q', elt)
QUEUE(eltype) q;
eltype elt;
postcondition *q == q' + <elt>*

Un ejemplo típico de programación formando colas de prioridades es el sistema de tiempo compartido necesario para mantener un conjunto de procesos que esperan servicio para trabajar. Los diseñadores de esta clase de sistemas asignan cierta prioridad a cada proceso.

El orden en que los elementos son procesados y, por tanto, eliminados sigue estas reglas:

1. Se elige la lista de elementos que tienen mayor prioridad.
2. En la lista de mayor prioridad, los elementos se procesan de mayor a menor, según el orden de llegada y de acuerdo con la organización de la cola.

3.4. Definición de las operaciones sobre una cola con prioridades

Un algoritmo usa una cola. Inicialmente, los trabajos se colocan al final de la cola. El planificador toma el primer trabajo de la cola, lo ejecutará hasta que termine o alcance su limite de tiempo y, si no termina, lo colocará al final de la cola. Por lo general, esta estrategia no es adecuada porque trabajos muy cortos en ejecución permanecerán lentos debido al tiempo de espera. No obstante, los trabajos cortos deben terminar tan pronto como sea posible, de modo que deberán tener preferencia sobre los que ya han estado en ejecución. Asimismo, los trabajos no cortos pero muy importantes deben tener prioridad.

Esta aplicación particular parece requerir una clase especial de cola, denominada cola de prioridad[1]. En ésta, el procesador central no atiende por riguroso orden de llamada, sino según prioridades asignadas por el sistema o el usuario, y sólo dentro de las peticiones de igual prioridad se producirá una cola[2].

Una cola de prioridad es una estructura de datos que permite al menos las siguientes dos operaciones: insertar, que hace la operación obvia; y eliminar_min, que busca, devuelve y elimina el elemento mínimo del montículo. La función insertar equivale a encolar; y eliminar_min, a desencolar.

La siguiente figura es un modelo básico de una cola de prioridad:

3.4.1. Operación para insertar un elemento en una cola con prioridades

Primero, estableceremos que un montículo es un árbol binario completo, lleno, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha.

Por otro lado, debemos mencionar que la propiedad de orden montículo nos permite efectuar rápidamente las operaciones. Aplicando esta lógica, llegamos a la propiedad de orden de montículo. (Todo trabajo implica asegurarse de que se mantenga esta propiedad).

Para insertar un elemento x en el montículo, creamos un hueco en la siguiente posición disponible; de lo contrario, el árbol no estaría completo. Ahora, si es posible colocar x en ese hueco sin violar la propiedad de orden de montículo, todo queda listo. De lo contrario, se desliza el elemento que está en el lugar del nodo padre del hueco, subiendo el hueco hacia la raíz. Seguimos este proceso hasta colocar x en el hueco. La anterior estrategia general se conoce como filtrado ascendente (el elemento nuevo se filtra en el montículo hasta encontrar su posición correcta).

A continuación explicamos el procedimiento para insertar en un montículo binario:

```
        (M. elementos "0" es un centinela)
        proceder insertar (x: tipo_elemento; var M: COLA_DEPRIORIDAD);
                var y: integer;
        begin
(1)             M. tamaño := MÁX_ELEMENTO then;
(2)                     error('El montículo está lleno')
```

[1] Mark Allen Weiss, *Estructura de datos y algoritmos,* México, Addison-Wesley Iberoamericana, 1995, p. 180.
[2] Luis Joyanes Aguilar, *Fundamentos de programación: algoritmos y estructura de datos,* México, Mc Graw-Hill, 1990, p. 407.

```
                else
                begin
(3)                     M. tamaño :=M.tamaño+1;
(4)                     i :=M.tamaño;

(5)                     while M.elemento(i div 2)>x do
                        begin
(6)                             M.elemento(i) := M.elemento(i div 2);

(7)                             i :=i div 2
                        end;
(8)                     M.elemento(i) := x;
                    end;
                end;
```

Podríamos implantar el filtrado en la rutina inserta realizando intercambios repetidos hasta establecer el orden correcto (recordemos que un intercambio requiere tres enunciados de asignación). Si un elemento es filtrado hacia arriba d niveles, el número de asignaciones efectuadas por los intercambios podría ser 3d. Así, nuestro método aplicado usa d + 1 asignaciones. Si el elemento que se ha de introducir es el nuevo mínimo, se llevará hasta arriba en el árbol en algún punto, y será 1, y se deseará romper el ciclo *white*. Esto se puede hacer con una comprobación explícita, pero hemos preferido poner un valor muy pequeño en la posición 0 con el fin de asegurar que el ciclo *white* termine. Se debe garantizar que este valor sea menor (o igual) que cualquier elemento del montículo –a dicho valor se le llama centinela–. Esta idea es semejante a usar nodos cabecera en las listas enlazadas.

El tiempo para hacer la inserción podría llegar a ser O(log n), si el elemento que se ha de insertar es el nuevo mínimo y es filtrado en todo el camino hacia la raíz. En término medio, el filtro termina pronto. Se ha demostrado que se requiere 2.607 comparaciones en promedio para realizar una inserción, en término medio, para subir un elemento 1.607 niveles.

3.4.2. Implantación dinámica de las operaciones sobre listas con prioridades

```
Int getnode(void)
{
int p;
        if(avail==-1) {
                printf("overflow\n");
                exit(1);
        }
        p=avail;
        avail=node[avail].next;
        return(p);
]       /*fin de getnode*/
```

3.4.2.1. Bicolas

Existe una variante de las colas simples, la doble cola o bicola. Ésta es una cola bidimensional en la que las inserciones y eliminaciones pueden realizarse en cualquiera de los dos extremos de la lista, pero no por la mitad. (El término *bicola* hace referencia a que la cola puede verse como una cola bidireccional).

Hay varias formas de representar una bicola en una computadora. A menos que se indique lo contrario, asumiremos que nuestras bicolas se mantienen en un *array* circular BICOLA con punteros IZQ y DER, que apuntará a los dos extremos de la bicola, y que los elementos se encuentran en los extremos izquierdo y derecho. El término *circular* hace referencia a que damos por hecho que la bicola [1] va detrás de bicola [N] en el *array*.

3.4.3. Definición del tipo de dato abstracto de bicola

Para representar una bicola, puede elegirse una representación estática con *arrays*, o una dinámica, con punteros. En ésta, la mejor opción es mantener la variable bicola con las variables puntero izquierdo y derecho, respectivamente.

3.5. Definición sobre las operaciones sobre bicolas

Las operaciones básicas que definen una bicola son:

Crear (Bq): inicializa una bicola sin elementos.
Esvacia(Bq): devuelve verdadero si la bicola no tiene elementos.
InserIzq (X,Bq): añade un elemento por el extremo izquierdo
InserDer(X,Bq): añade un elemento por el extremo derecho.
ElimnIzq(X,Bq): devuelve el elemento izquierdo y lo retira de la bicola.
EliminDer (X, Bq): devuelve el elemento derecho y lo retira de la bicola.

3.5.1. Operación para construir una bicola vacía

```
with unchecked_deallocation;
package body bicolas is

  procedure disponer is new unchecked_deallocation(unDato,ptUnDato);

  procedure creaVacia(b:out bicola) is
  begin
    b:=(null,null,0);
  end creaVacia;
```

3.5.2. Operación para revisar si una bicola es vacía o no

```
function observaIzq(b:in bicola) return elemto is
  begin
    return b.izq.dato;
  end observaIzq;
```

```
function observaDer(b:in bicola) return elemto is
begin
  return b.der.dato;
end observaDer;

function esVacia(b:in bicola) return boolean is
begin
  return b.n=0;
end esVacia;

function long(b:in bicola) return natural is
begin
  return b.n;
end long;
```

3.5.2.1. Operación para obtener el elemento que está al final de la bicola

```
procedure eliminaDer(b:in out bicola) is
  begin
    b.der:=b.der.ant;
    if b.der=null then disponer(b.izq); else disponer(b.der.sig); end if;
    b.n:=b.n-1;
  end eliminaDer;

  function observaIzq(b:in bicola) return elemto is
  begin
    return b.izq.dato;
  end observaIzq;

  function observaDer(b:in bicola) return elemto is
  begin
    return b.der.dato;
  end observaDer;

  function esVacia(b:in bicola) return boolean is
  begin
    return b.n=0;
  end esVacia;

  function long(b:in bicola) return natural is
  begin
    return b.n;
  end long;
```

3.5.2.2. Operación para obtener el elemento que está al inicio de la bicola

```
procedure eliminaIzq(b:in out bicola) is
  begin
    b.izq:=b.izq.sig;
    if b.izq=null then disponer(b.der); else disponer(b.izq.ant); end if;
    b.n:=b.n-1;
  end eliminaIzq;
```

3.5.2.3. Operación para insertar un elemento al final de la bicola

```
procedure agnadeDer(b:in out bicola; e:in elemto) is
```

```
  aux:ptUnDato;
 begin
  aux:=new unDato'(e,null,b.der);
  if b.izq=null then b.izq:=aux; b.der:=aux; else b.der.sig:=aux; b.der:=aux; end if;
  b.n:=b.n+1;
 end agnadeDer;
```

3.5.2.4. Operación para insertar un elemento al inicio de la bicola

```
procedure agnadeIzq(e:in elemto; b:in out bicola) is
  aux:ptUnDato;
 begin
  aux:=new unDato'(e,b.izq,null);
  if b.der=null then b.der:=aux; b.izq:=aux; else b.izq.ant:=aux; b.izq:=aux; end if;
  b.n:=b.n+1;
 end agnadeIzq;
```

3.5.2.5. Operación para remover un elemento que está al final de la bicola

```
procedure eliminaDer(b:in out bicola) is
 begin
  b.der:=b.der.ant;
  if b.der=null then disponer(b.izq); else disponer(b.der.sig); end if;
  b.n:=b.n-1;
 end eliminaDer;

 function observaIzq(b:in bicola) return elemto is
 begin
  return b.izq.dato;
 end observaIzq;

 function observaDer(b:in bicola) return elemto is
 begin
  return b.der.dato;
 end observaDer;

 function esVacia(b:in bicola) return boolean is
 begin
  return b.n=0;
 end esVacia;

 function long(b:in bicola) return natural is
 begin
  return b.n;
 end long;
```

3.5.2.6. Operación para remover un elemento que está al inicio de la bicola

```
procedure libera(b:in out bicola) is
  aux:ptUnDato;
 begin
  aux:=b.izq;
  while aux/=null loop
   b.izq:=b.izq.sig;
   disponer(aux);
   aux:=b.izq;
  end loop;
```

```
   creaVacia(b);
  end libera;
```

3.5.2.7. Definición de la semántica de las operaciones sobre bicolas

```
generic
  type elemto is private;
package bicolas is

  type bicola is limited private;

  procedure creaVacia(b:out bicola);
  procedure agnadeIzq(e:in elemto; b:in out bicola);
  procedure agnadeDer(b:in out bicola; e:in elemto);
  procedure eliminaIzq(b:in out bicola);
  procedure eliminaDer(b:in out bicola);
  function observaIzq(b:in bicola) return elemto;
  function observaDer(b:in bicola) return elemto;
  function esVacia(b:in bicola) return boolean;
  function long(b:in bicola) return natural;
  procedure asigna(bout:out bicola; bin:in bicola);
  procedure libera(b:in out bicola);
  function "="(b1,b2:in bicola) return boolean;

private

  type unDato;
  type ptUnDato is access unDato;
  type unDato is record dato:elemto; sig,ant:ptUnDato; end record;
  type bicola is record izq,der:ptUnDato; n:natural; end record;

end bicolas;
```

3.5.2.8. Implantación dinámica de una bicola

```
#include stdlib.h

int* ptr; /* puntero a enteros */

int* ptr2; /* otro puntero */

/* reserva espacio para 300 enteros */

ptr = (int*)malloc ( 300*sizeof(int) );

ptr[33] = 15; /* trabaja con el área de memoria */

rellena_de_ceros (10,ptr); /* otro ejemplo */

ptr2 = ptr + 15; /* asignación a otro puntero */

/* finalmente, libera la zona de memoria */

free(ptr);
```

3.6. Pilas

Una pila (*stack*) es un tipo especial de lista en la que la inserción y borrado de nuevos elementos se realiza sólo por un extremo que se denomina cima o tope.

3.6.1. Definición del tipo de dato abstracto pila

Una pila es una colección ordenada de elementos en la cual, en un extremo, pueden insertarse o retirarse otros, llamando la parte superior de la pila. Una pila permite la inserción y eliminación de elementos, por lo que realmente es un objeto dinámico que cambia constantemente.

3.6.2. Definición de las operaciones sobre pilas

Los dos cambios que pueden hacerse en una pila tienen nombres especiales. Cuando se agrega un elemento a la pila, éste es "empujando" (*pushed*) dentro de la pila. La operación *pop* retira el elemento superior y lo regresa como el valor de una función; la *empty* determina si la pila está o no vacía; y la *stacktop* determina el elemento superior de la pila sin retirarlo. (Debe retomarse el valor del elemento de la parte superior de la pila).

3.6.2.1. Operación para crear una pila vacía

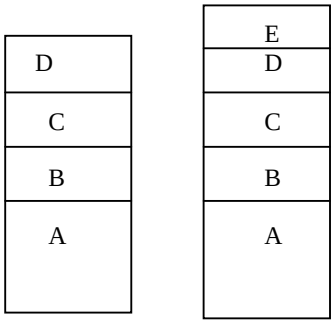
Pila inclinada (void)
/*crea una pila vacía*/
{Post; inclinada=0}

3.6.2.2. Operación para insertar un elemento a una pila

Los nuevos elementos de la pila deben colocarse en su parte superior –que se mueve hacia arriba para dar lugar a un nuevo elemento más alto–; además, los que están en este lugar pueden ser removidos (en este caso, la parte superior se desliza hacia abajo para corresponder al nuevo elemento más alto).

Ejemplo:

En una película en movimiento de una pila se agrega el elemento G a la pila. A medida que nuestra película avanza, puede verse que los elementos F, G y H han sido agregados sucesivamente a la pila. A esta operación se le llama lista empujada hacia abajo:



3.6.2.3. Operación para revisar si una pila es vacía o no

Una pila puede utilizarse para registrar los diferentes tipos de signos de agrupación. En cualquier momento que se encuentre un signo de éstos abriendo la expresión, es empujado hacia la pila, y cada vez que se detecte el correspondiente signo terminal, ésta se examina. Si la pila está vacía, quiere decir que el signo de la agrupación terminal no tiene su correspondiente apertura, por lo tanto, la hilera es inválida:

```
Fila vacía (vacía)
Inicio
        Si p=0
         Entonces VACIA ----Cierto
         Si no       VACIA ----Falso
        Fin - si
        Fin
```

3.6.2.4. Operación para obtener el último elemento insertado en la pila

La operación *pop* retira el último elemento superior y lo regresa como un valor de la función (en cada punto, se aleja el elemento superior, puesto que la operación sólo puede hacerse desde este lugar). El atributo más importante consiste en que el último elemento insertado en una pila es el primero en ser retirado.

3.6.2.5. Operación para remover el último elemento insertado en la pila

La operación *stackpop* determina el elemento superior de la pila SFR: basta con retirarlo y retomar el valor del elemento de la parte superior de la pila.

3.6.3. Definición de la semántica de las operaciones sobre pilas

Pilas
Las pilas son las estructuras de datos más utilizadas. Se trata de un caso particular de las estructuras lineales generales (secuencias o listas) que, debido a su amplio ámbito de aplicación, conviene sean estudiadas independientemente.

Fundamentos
La pila es una lista, o colección de elementos, que se distingue porque las operaciones de inserción y eliminación se realizan solamente en un extremo de la estructura. El extremo donde se llevan a cabo estas operaciones se denomina habitualmente cima o parte superior de la pila (*top*).

Las operaciones fundamentales en una pila son meter –que es equivalente a una inserción– y sacar de la pila –que elimina el elemento insertado más recientemente–. El elemento insertado más recientemente puede examinarse antes de realizar un *sacar* mediante la rutina cima. Un *sacar* o cima en una pila vacía suele considerarse como un

error en el TDA pila. Por otro lado, quedarse sin espacio al realizar un meter es un error de implantación y no de TDA.

Dada una pila P = (a, b, c,...k), se dice que a –elemento más inaccesible de la pila–, está en el fondo de la pila (*bottom*) y que k –el más accesible– está en la cima.

Las restricciones definidas para la pila implican que si una serie de elementos A, B, C, D y E se añaden –en este orden– a una pila, entonces, el primer elemento que se borre de la estructura deberá ser E. Por tanto, resulta que el último elemento que se inserta en una pila es el primero que se borra. Por esta razón, se dice que una pila es una lista LIFO (Last In First Out, es decir, el último en entrar es el primero en salir).

Un ejemplo típico de pila lo constituye un montón de platos. Cuando se quiere introducir un nuevo plato, éste se pone en la posición más accesible, encima del último. Al coger un nuevo plato, éste se extrae, igualmente, del punto más accesible, el último que se ha introducido.

Otro ejemplo natural de la aplicación de la estructura pila aparece durante la ejecución de un programa de ordenador, en la forma en que la máquina procesa las llamadas a los procedimientos. Cada llamada a un procedimiento (o función) hace que el sistema almacene toda la información que está asociada a éste (parámetros, variables, constantes, dirección de retorno, etcétera) de forma independiente a otros procedimientos y permitiendo que unos puedan invocar a otros distintos (o a sí mismos), y que toda esa información almacenada pueda ser recuperada convenientemente cuando corresponda. Como en un procesador sólo puede estarse ejecutando un procedimiento, es necesario que sean accesibles sus datos (el último activado que está en la cima). De ahí que la estructura pila sea muy apropiada para este fin.

En esta estructura, hay una serie de operaciones necesarias para su manipulación:
Crear la pila.
Comprobar si la pila está vacía (necesario para saber si es posible eliminar elementos).
Acceder al elemento situado en la cima.
Añadir elementos a la cima.
Eliminar elementos de la cima.

La especificación correcta de todas estas operaciones permitirá definir adecuadamente una pila.

Representación de las pilas

Los lenguajes de programación de alto nivel no suelen disponer de un tipo de datos pila. Por el contrario, los de bajo nivel (ensamblador) manipulan directamente alguna estructura pila propia del sistema. Por lo tanto, es necesario representar la estructura pila a partir de otros tipos de datos existentes en el lenguaje.

La forma más simple de representar una pila es mediante un vector unidimensional. Este tipo de datos permite definir una secuencia de elementos –de cualquier tipo– y posee un eficiente mecanismo de acceso a la información que contiene.

Al definir un *array*, debe determinarse el número de índices válidos y, por lo tanto, el número de componentes definidos. Así, la estructura pila representada por un *array* tendrá un número limitado de elementos posibles.

Se puede definir una pila como una variable:

Pila: array [1..n] de T

donde T es el tipo que representa la información contenida en la pila (enteros, registros…).

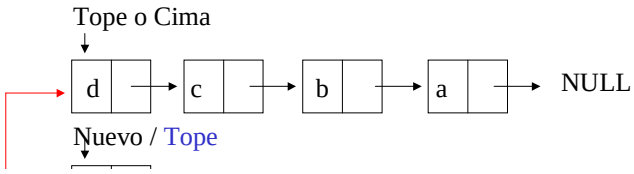
El primer elemento de la pila se almacenará en Pila[1], y será el fondo de la pila; el segundo, en Pila[2], y así sucesivamente. En general, el elemento i-úsimo estará almacenado en Pila[i]. Como todas las operaciones se realizan sobre la cima de la pila, es necesario tener correctamente localizada, en todo instante, esa posición. Asimismo, es indispensable una variable –cima– que apunte al último elemento (ocupado) de la pila.

3.6.4 Implantación dinámica de una pila

La pila incorpora la inserción y supresión de elementos, por lo que ésta es un objeto dinámico constantemente variable. La definición específica que de un extremo de la pila se designa como tope de la misma. Pueden agregarse nuevos elementos en el tope de la pila, o quitarse los elementos que están en el tope.

PILAS DINAMICAS

PUSH: insertar un elemento



PILAS DINAMICAS

Representaciòn grèfica de una pila o stack

PILAS DINAMICAS

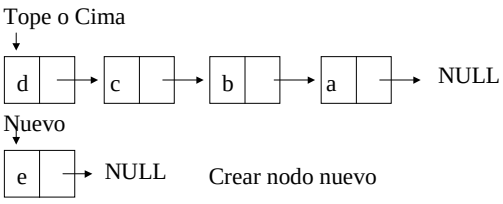
Función para insertar elementos - PUSH-

```
void push(PTRSTACK *Tope, int valor)
{
PTRSTACK Nuevo;

        Nuevo = new STACK;
        Nuevo -> dato = valor;
        Nuevo -> siguiente = *Tope;
        *Tope = Nuevo;
}
```

PILAS DINAMICAS

PUSH: insertar un elemento



PILAS DINAMICAS

Estructura para pilas dinámicas

```
struct stack {
        int dato;
        struct stack *siguiente;
};

typedef struct stack STACK;
typedef STACK *PTRSTACK;
```

Materia: Informática II

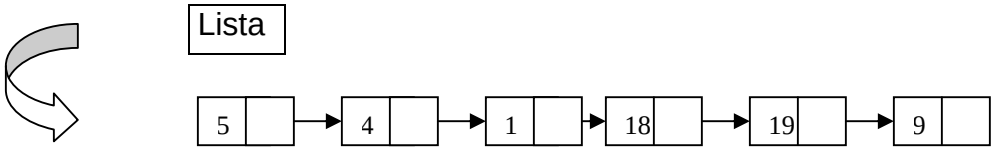
Unidad 4. Estructuras en memoria secundaria

Temario

4.1. Esquema de persistencia para una lista

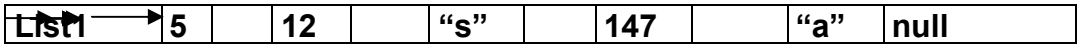
5	4	1	7	18	19	9	7	45	...		

(a) array representado por una lista



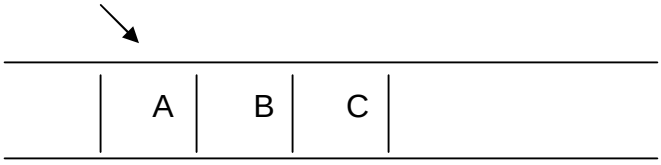
(b) lista enlazada representada por una lista de enteros

4.2. Esquema de persistencia para una lista generalizada

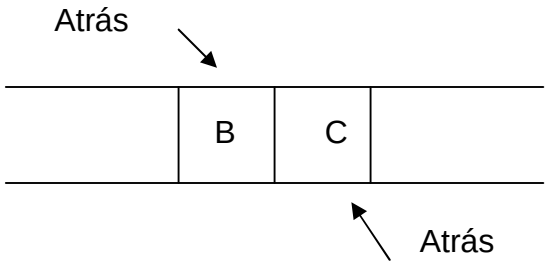


4.3. Esquema de persistencia para una cola y una cola con prioridades

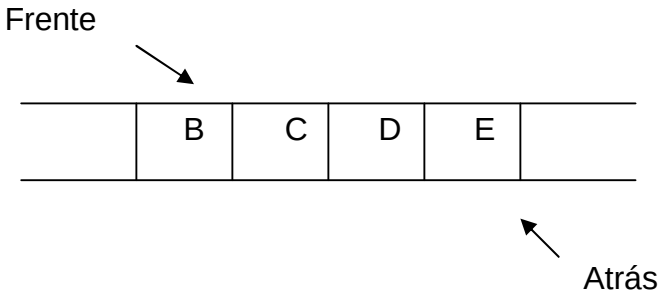
Frente



Cuando se elimina un elemento de la cola, se retira únicamente del frente de la misma. A menudo, este es utilizado para simular situaciones en el mundo real.



Cuando se agregan los elementos D y E, deben incluirse en la parte posterior de la cola:



4.4. Esquema de persistencia para una bicola

Hay dos variantes de la doble cola:

- Doble cola de entrada restringida: acepta intersecciones sólo al final de la cola.
- Doble cola de salida restringida: acepta eliminaciones sólo al frente de la cola.

BICOLA

IZQ 4			AAA	BBB	CCC	DDD		
DER 7	1	2	3	4	5	6	7	8

4.5. Esquema de persistencia para una pila

Símbolo de entrada	Pila (cabeza.....base)	Representación posfija
A	#	**A**
*****	***** **#**	**A**
B	***** **#**	**AB**
+	**+ #**	**AB***
(	**(+ #**	**AB***
C	**(+ #**	**AB*C**
-	**- (+ #**	**AB*C**
D	**- (+ #**	**AB*CD**
/	**/ - (+ #**	**AB*CD**
E	**/ - (+ #**	**AB*CDE**
)	**+ #**	**AB*CDE/-**
#		**AB*CDE/-+#**