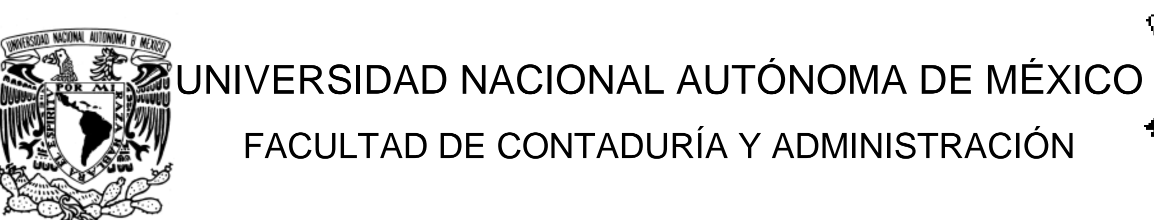


AUTOR: ISMAEL VICENTE HERNÁNDEZ CASTILLO

Informática II (Estructuras de datos estáticas y dinámicas en memoria principal)		Clave:	1265
Plan:	2005	Créditos:	12
Licenciatura:	Informática	Semestre:	2º
Área:	Informática (Desarrollo de sistemas)	Asesoría	
Requisitos:	Ninguno	Por semana:	6 h
Tipo de asignatura:	Obligatoria (x)	Optativa ( )	

Objetivo general de la asignatura

Al finalizar el curso, el alumno será capaz de entender la abstracción, e implantar en un lenguaje de programación las estructuras de datos más importantes.

Temario oficial (horas sugeridas 96)

1. Fundamentos de las estructuras de datos (16 h)
2. Estructuras de Datos Fundamentales (40 h)
3. Estructuras de datos avanzadas (40 h)

Introducción

Las Estructura más simples alojadas en la memoria principal se estudian por dos razones fundamentales: la primera porque de ellas se forman otras estructuras más complejas y la segunda por que varios compiladores actualmente tiene incluidos los tipos de datos estándar.

Las Estructuras de Datos son objetos con los cuales se representa el manejo y ubicación de la información en la memoria interna de la Computadora. Por lo cual se administra el espacio en dicha memoria.

TEMA 1. FUNDAMENTOS DE LAS ESTRUCTURAS DE DATOS

Objetivo particular

Al finalizar el tema el alumno conocerá las estructuras de datos simples, así como su importancia en la representación de la información en la memoria principal de la computadora.

Temario detallado

1.1 Definición de estructura de datos
1.2 Tipos de datos
1.3 Tipos de datos abstractos

Introducción

Las estructuras simples de datos son la base del manejo de información en la programación, conocer las operaciones y el manejo en memoria de éstas nos permite construir programas más eficaces y, sobre todo, empleado a los nuevos lenguajes, nos permite desarrollar aplicaciones más seguras y robustas.

1.1 Definición de estructuras de datos

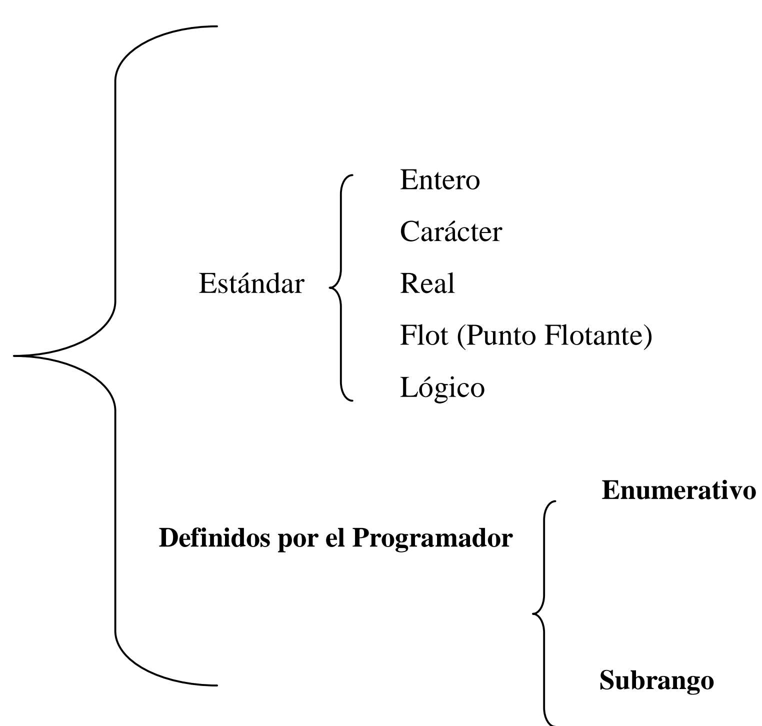
"Es la representación conceptual de la organización interna de los datos en la memoria interna de a computadora con el fin de optimizar el empleo del espacio de dicha memoria".[1] Los Tipos de Datos son objetos que representan

[1] Cf. Luis Joyanes Aguilar, *Fundamentos de programación*, 2ª ed., México, McGraw Hill, 1996, pp. 439-440, 496.

tipos de información determinada almacenada y manejada en las páginas de memoria interna del ordenador los cuales se utilizarán para una aplicación.

1.2 Tipos de datos

Clasificación de las Estructuras de Datos



Cuadro 1.1 Clasificación general de las estructuras de datos simples

A los Datos Estándar también se les conoce como Datos Primitivos; y a los Definidos por el Programador como Datos Incorporados.

Tipos de dato estándar

Los tipos de dato estándar son aquellos que no presentan una estructura, son unitarios, y nos permiten almacenar un solo dato.

Entero

El tipo entero es un subconjunto de los números enteros, que dependiendo del lenguaje que estemos usando podrá ser mayor o menor a 16 bits

(2^16=32768), es decir se pueden representar desde el -32768 hasta el 32767. Para representar un número entero fuera de este rango se tendría que usar un tipo real.

Real

El tipo real define un conjunto de números que puede ser representado con la notación de punto-flotante, por lo que nos permite representar datos muy grandes o muy específicos.

Carácter

Cualquier signo tipográfico. Puede ser una letra, un número, un signo de puntuación o un espacio. Generalmente este tipo de dato está definido por el conjunto ASCII.

Lógicos

Este tipo de dato, también llamado booleano, permite almacenar valores de lógica booleana o binaria, es decir, representaciones de verdadero o falso.

Float

Este tipo de Datos se emplean en aquellos datos fraccionarios que requieren de cifras a la derecha del punto decimal o de entero con varios decimales. De estos se desprende la necesidad de crear el formato de representación científica (10 E +12).

Definidos por el programador

Este tipo de datos nos ayuda a delimitar los datos que podemos manejar dentro de un programa y nos sirve como una manera segura de validar la entrada/salida de este.

Subrango

En este tipo de datos nosotros delimitamos el rango, menor y mayor de los posibles valores que puede tomar una variable, de esta manera podemos asegurar que la entrada o salida de nuestro programa esté controlada.

Ejemplos: type Tipo1 = 1.. 30; type Tipo2 = 1.. 10;

Enumerativo

Los tipos de dato enumerativo son tipos que pueden tomar valores dentro de un rango en el que se especifica ordenadamente cada uno de dichos valores, al igual que el tipo de subrango nos permite restringir los valores de una variable.

Ejemplo: type Dias = lunes, martes, jueves, sábado;

En el siguiente programa en Pascal 7.0 se muestra el empleo de los tipos de Datos Simples

```
program l'1 1;
{$APPTYPE CONSOLE}
{$R+} uses
SysUti1s,
Math;
type sub1000 = 1..1000;
type mOd2 .(par, impar); var
num1_int, nurn2_int:sub1000; res_int:integer; res_rea:real;
n1_mod,n2_mod,espar:mOd2; res_boo:boolean;
repetir_char:char;
function divlnt(i1_int, i2_int: integer}: real; begin
Result := i1 int / i2 int;
if ((i1_int mod i2_intJ > O) then
     raise EMathError.Create('La division de estos numeros
arrojaría un numero real') divlnt := i1_int div i2_int;
--~
```

```
     end;

procedure writeReal(il_int, i2_int:integer};
begin
res_real := il int / i2 int;
writeln('. pero en este caso tenemos que guardar el
resultado'};
writeln('de' numl_int. '/', num2_int, ' en un real:
',res_rea:2:2};
-',

end;

begin

espar:=par;
repeat
writeln(repetir_char) ;
write('Intoduce un entero positivo menor a 1000: ');
readln(num1_intl);
write('Intoduce otro entero positivo menor a 1000: ');
readln(num2_intl);
writeln('Tenemos 2 enteros'); n1_mod:=impar; n2 mod:=impar;
if (num1_int mod 2)=0) then nl_mod:=par;
 if (num2 int mod 2)=0) then n2 mod:=par;
writeln('Üsando tipos enumerados, veamos si el primer
entero `, num1_int);
res boo:= n1_mod = espar;
writeln('es un numero par:', res_boo);
writeln('y hagamos lo mismo con el segundo:', n2_mod E
espar);
writeln;
```

```
writeln('Ahora, a estos numeros los podemos Sumar: ',numl
int, '+',num2 int);
 res_int:=numl_int + nurn2_int;
 writeln('y el resultado guardarlo en un entero:
',res_int);
 writeln;
writeln('Los podemos restar:' ,numl int, '-',num2 int);
 res int:=numl int - num2 int;
writeln('y el resultado guardarlo en un entero: ',res_int);
writeln¡
writeln('Los podemos multiplicar: ',numl_int,
'*',num2_int);
res int:=numl int * num2 int;
writeln('y el-resultado guardarlo en un entero: ',res_int);
writeln;
writeln('Podemos tambien sacar el residuo de la divison:
',numl_int, , mod num2 int);
res int:=numl int mod num2 int¡
writeln('y el-resultado gUardarlo en un entero: ',res_int);
 writeln;
writeln('y podemos hacer una división entera: ',numl_int, ,
div ',num2_int); res_int:=numl_int div num2_int¡
writeln('y el resultado guardar lo en otro entero:
',res_int);
 writeln;
writeln('pero si los tratamos de dividir, el resultado
solo');
writeln('podria guardarse en un entero si la division');
write('fuera exacta');
try
     res int:=Math.floor(divInt(numl int, num2 int));
     writeln(' como en este caso en Que el resultado') ¡
```

```
     writeln('de `, numl_int, '/', num2_int, ' es entero:
     ',res_int) ¡
except
     On EMathError do writeReal(numl_int, num2_int);
     //writeln('como en este caso en que el resultado es
     entero: ',res_int);
      end¡
writeln;
writeln('para salir del programa presiona <q> y <enter>');
writeln('o presiona <enter> para volver a correr el
programa'}; read(repetir_char) ;
until (repetir_char='q') or (repetir_char.'Q');

end.
```

1.3 Tipos de datos abstractos (se verá en el tema siguiente)

Los Tipos de Datos Abstractos TDA son modelos con los cuales represento estructuras con propiedades relativas a un objeto general con el cual se desarrollará una aplicación en particular. Así como se hizo referencia en el punto 1.1 lo relativo a los Tipos de Datos, los TDA no hacen referencia a una aplicación específica sino que su finalidad es la generalización de la Definición del Objeto con propiedades establecidas para su modelo conceptual de representación de la aplicación.

Bibliografía y sitios web del tema 1

Hernández Castillo, Vicente. *Guía Didáctica de Informática II*, México, SUA-UNAM, AÑO

Joyanes Aguilar, Luis, *Estructura de Datos: Algoritmos, abstracción y objetos,* 3ª ed., McGraw-Hill, Madrid, 1999. [ISBN: 8448106032]

http://es.wikipedia.org/wiki/Estructura_de_datos.

"Estructura de datos, material en línea, disponible en: http://www.monografias.com/trabajos14/estruct-datos/estruct-datos.shtml

Actividades de Aprendizaje.

A.1.1. Investiga y elabora un cuadro comparativo con los diferentes tipos de Datos Simples manejados por los Compiladores de los Lenguajes de Programación en Pascal, C y C++. Compáralos con los Datos Primitivos que poseen estos compiladores.

A.1.2. Describe en un documento que Programas en los Lenguajes citados en la actividad anterior despliegan los tipos de datos simples tratados en el tema. En algunos libros los engloban como datos estándar.

A.1.3. Elabora un Programa en Lenguaje C, C++, Pascal o en Visual Basic donde se imprimen los tipos de datos estándar del compilador correspondiente. Consulta los Manuales y los llamados *Reference Guide* de estos Lenguajes.

Cuestionario de autoevaluación

1. Anota la definición de Tipo de Dato Simple.
2. Cuáles son los tipos de datos simples.
3. Los Tipos de Datos son Objetos _______________ para la aplicación.
4. Qué es un tipo de dato float y cuál es su utilidad.
5. Para enlistar los Estados de la República Mexicana, ¿Qué tipo de dato se emplearía?
6. Si reúno varios caracteres, ¿Qué tipo de dato obtengo?
7. ¿Qué tipo de dato necesito para representar el valor del promedio de estaturas de un conjunto de personas?

8. La letra “A” se puede emplear en operaciones de comparación si le asigno un ___________, el cual también tiene un valor ya asignado en el Código _________.
9. Enlista los tipos de Datos Simples empleados por los compiladores Pascal, C y C++.
10. Anota la instrucción en Lenguaje Pascal y en Lenguaje C++ para definir un tipo de dato subrango.

Examen de autoevaluación

1.	El tipo de Dato Float es una afinación del tipo de dato entero	F	V
2.	Un conjunto de caracteres origina una cadena	F	V
3.	Todo carácter tiene un valor en el Código ASCII	F	V
4.	El cero es un valor entero	F	V
5.	El valor π es tipo de dato float	F	V
6.	Una letra se puede desplazar a lo largo de la pantalla del monitor gracias a una instrucción	F	V
7.	La operación A = 2 es de asignación	F	V
8.	La operación A = 2 es de acumulación	F	V
9.	La instrucción A = texto es correcta	F	V
10.	El tipo de dato double es real.	F	V

TEMA 2. ESTRUCTURAS DE DATOS FUNDAMENTALES

Objetivo Particular

Al finalizar el tema el alumno conocerá y comprenderá a los *tipos de datos complejos*, no solo como una integración de los tipos de datos simples sino también como una representación abstracta para desarrollar aplicaciones.

Temario detallado

2.1. Introducción a los tipos de datos abstractos

2.2. Arreglos

2. 2.1 Unidimensionales

2.2.2 Multidimensionales

2.2.3 Operaciones sobre arreglos

2.3.- Pilas

2.3.1.-Definición del tipo de dato abstracto pila

2.3.2.-Definición de las operaciones sobre pilas

2.3.3.-Implantación de una pila

2.4.-Colas

2.4.1.-Definición del tipo de dato abstracto cola

2.4.2.-Definición de las operaciones sobre colas

2.4.3.-Bicolas

2.4.4.-Implantación de una cola

2.5.-Listas

2.5.1.-Definición del tipo de dato abstracto lista

2.5.2.-Definición de las operaciones sobre listas

2.5.3.-Implantación de una lista

2.6.-Tablas de dispersión, funciones hash

Introducción

Los Tipos de Datos Complejos se integran de varios tipos de Datos Simples ya sean del mismo tipo o de varios, según las necesidades, pero aquellos se dividen en Estáticos y Dinámicos. En esta Unidad nos enfocaremos en los Estáticos, es decir, aquellos cuyo tamaño de memoria requerido se mantiene fijo a largo de la aplicación de desarrollar.

Cuadro 2.1 Clasificación de los Tipos de Datos Complejos

2.1 Introducción a los tipos de datos abstractos

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 una representación en memoria o la instrucción de sus operaciones.

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). Esas 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 pseudocódigo de las funciones. Y en el refinamiento final se especificarán los detalles de las funciones y la organización de memoria práctica para la 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.

Representación de un objeto abstracto

Al comenzar el diseño de un TDA (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 y el 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 TDA matriz, una manera gráfica de representar el modelo abstracto sobre el cual se va a trabajar es la siguiente.

COLUMNAS		A	B	……	J
REN	A				
GLO	:				
NES	:				
	J				X(I, J)

Figura 2.1. Un Elemento en una matriz definida [A..J, 0..5]

Con esta anotación, 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, etc., sin necesidad de establecer estructuras de datos concretas para manejarlas.

Ejemplo:

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

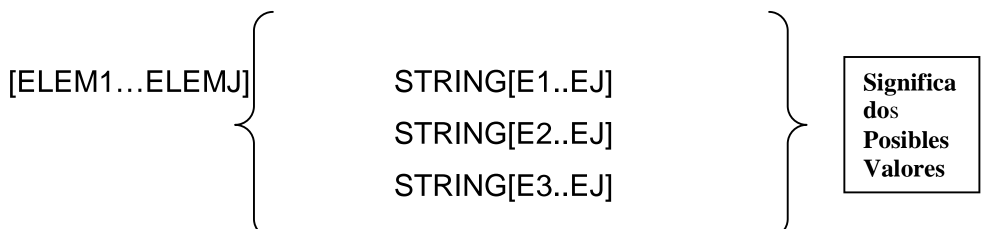


Figura 2.2. TDA de Significados de una cadena

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

Otra manera de representar el mismo objeto abstracto es:

<[palabra1,<si1,...,s1k>],..., [palabraN,<sNi,...,sNk>]>

Incluso podríamos hacer la siguiente representación gráfica.

STRING…I	S1…SN
STRING…I+1	S2..SN2
STRING…N	SN….N

Figura 2.3. Representación gráfica de un conjunto de cadenas

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

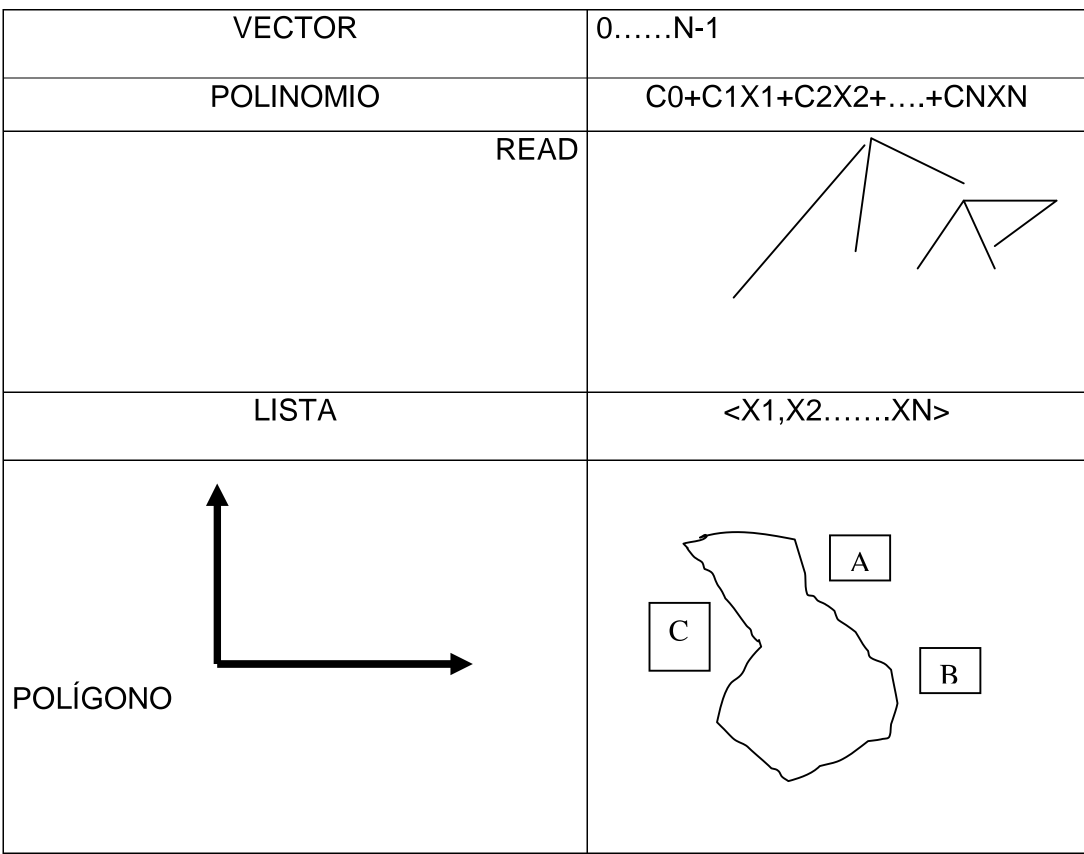
Ejemplo:
Hay objetos abstractos en una representación gráfica natural. Veamos algunos ejemplos

CONJUNTO DE ELEMENTOS	E1….EN
STRING1…STRINGN	C1….CN
C1 = [E1..EN] C2 = [E2..EM] C3 = [E3..ER]	[C1,C2,C3,C4…CN]

Figura 2.4 Integración de caracteres en cadenas

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:

Los atributos corresponden a objetos abstractos de los TDA Ficheros Autor, Fichero Título y Bodega. En este caso, se dice que la Biblioteca es un cliente de dichos TDA. Para claridad en la anotación, los nombres TDA se describen en mayúsculas. Y las nominaciones de los atributos tienen las características de cualquier variable.



VECTOR	0……N-1
POLINOMIO	C0+C1X1+C2X2+….+CNXN
READ	
LISTA	<X1,X2…….XN>
POLÍGONO	A C B

Figura 2.5 Representación Gráfica de Objetos Abstractos

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:

TDA: FICHERO AUTOR	ATRIBUTOS
USUARIO: BIBLIOTECA	Nombre
APLICACIONES:	Titulo
Altas	Genero
Bajas	Año
Prestamos.	

FICHERO TITULO	ATRIBUTOS Materia Año Volumen Paginas Editorial Impresión ISBN
FICHERO BODEGA	ATRIBUTOS Localización Capacidad Razón Social

Figura 2.6. TDA LIBRO

Los atributos corresponden a objetos abstractos de los TDA Ficheros Autor, Fichero Título y Bodega. En este caso, se dice que la Biblioteca es un cliente de dichos TDA. Para claridad en la notación, los nombres TDA se describen en mayúsculas. Y las nominaciones de los atributos tienen las características de cualquier variable.

Siempre hay que tomar en cuenta que un TDA es un modelo abstracto para resolver un problema en particular y generar el programa correspondiente. Su objetivo es poderse aplicar de forma general a los problemas semejantes al originalmente planteado, por tanto, es independiente del compilador al cual se le encargará interpretar el programa fuente que creó el programador. El TDA lo creó el Diseñador o el Programador.

Invariante de un dato abstracto

El invariante de un TDA establece una noción de validez, en términos de condiciones, para cada uno de los objetos abstractos, sobre una 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 *TDA Conjunto* y la notación {x1,...xN} asignada para todos sus ítem (elementos) por fila; el invariante debe exigir que todos los xi pertenezcan al mismo tipo de dato con iguales propiedades, y que sean diferentes entre sí en su contenido. 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, los cuales solo son válidos para ese TDA en particular; pudiéndose generalizar sus propiedades posteriormente.

Ejemplo:

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

- Las palabras estarán ordenadas ascendentemente y sin repetición
  $Elem_i.palabra<elem_{i+1}.palabra,\ 1<_i<N$
- Los significados estarán ordenados ascendentemente y sin repetición:
  $Elem_i.Sig_{r+1},\ 1<_i<N,\ 1<_r<_k$
- Toda palabra debe tener asociado por lo menos un significado:
  $Elem_i$ es igual a palabra,$<sig1,...,sig_k>\ k>0$

Si un objeto TDA 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.

Especificación de un tipo de dato abstracto

Un TDA se define con un nombre, un formalismo para expresar un objeto abstracto, o un invariante o un conjunto de operaciones sobre este objeto. Veamos el siguiente esquema:

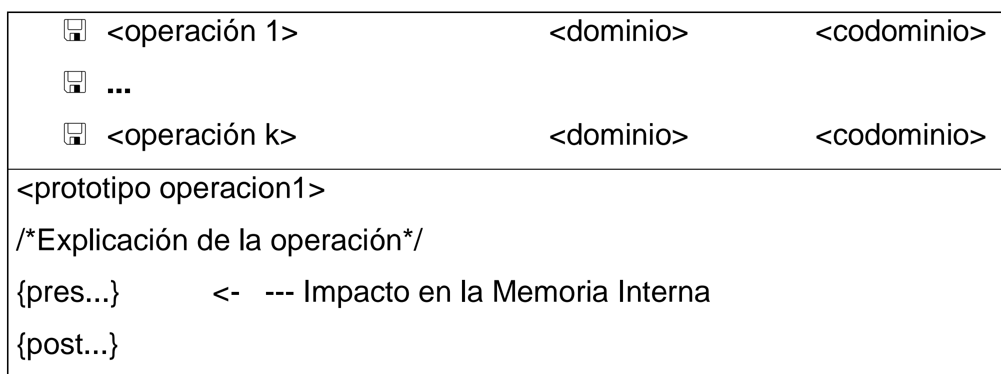
TDA <nombre>
<Objeto abstracto1>
<Invariante del TDA>

<Operaciones>
<Objeto abstracto2>
<Invariante del TDA>
<Operaciones>

Figura 2.7 Estructura de un TDA

La especificación de las operaciones consta de dos partes. Primero se coloca la funcionalidad de cada una de ellas (dominio y codominio 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:

Aplicando Terminología de Conjuntos:



<operación 1>	<dominio>	<codominio>
...		
<operación k>	<dominio>	<codominio>

<prototipo operacion1>
/*Explicación de la operación*/
{pres...} <- --- Impacto en la Memoria Interna
{post...}

Figura 2.8. Definición Operativa de TDA

La precondición y la poscondición de una operación pueden referirse únicamente a los elementos que integran el objeto abstracto y a los argumentos que reciben. No incluyen ningún otro tipo de elemento de 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 de las de validez del objeto. Si la precondición de una operación es TRUE, es decir, no imponen 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 TDA, 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 deben existir dos operaciones con igual nominación en un programa, incluso si pertenecen a TDA diferentes. Por esta razón, conviene agregar un mismo sufijo a todas las operaciones de un TDA, de tal forma que las identifique. Además, es preferible que este sufijo tenga por lo menos tres caracteres.

Ejemplo:
Para definir el TDA Matriz de los valores internos, se puede utilizar la especificación siguiente.

TDA BIBLIOTECA			
Object_Libro			
Titulo <A…Z,a..z,0..9>	Numero de Casos		
Materia<A..Z,a..z>	Numero de Casos		
Autor<A..Z,a..z>	Numero de Casos		
ISBN<0..9>	Numero de Casos		
Operaciones:			
Altas	Bajas	Prestamos	adquisición
Vistas			

Figura 2.9. TDA Biblioteca y su Estructura

En el caso del TDA 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 TDA: 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 específicos, es posible resolver cualquier problema que involucre una matriz.

Es importante mencionar 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.

Tipos de operaciones

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

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

```
Clase<constructura> (<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 TDA Matriz y Diccionario respectivamente. Un TDA puede tener múltiples constructoras.

- Modificadora: puede alterar el estado de un elemento del TDA. 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:<analizadora>=función(<estado del
objetoAbstracto>)}
```

En el ejemplo del TDA Matriz la única modificadora es la operación asignarMat, que altera el contenido de una casilla de la matriz. Otra modificadora posible de ese TDA 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>(<objeto abstracto>,
<argumentos>)
{pre:<condiciones del objeto adicionales al
invariante, condiciones de
los argumentos>}
{post:<analizadora>=function (<estado del objeto
abstracto>)}
```

En el TDA 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 TDA 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 TDA 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 TDA.
- Copia: modificadora que facilita alterar el estado de un objeto del TDA 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 en 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 TDA. Esta operación, que parece más sólida con la interfaz que con el modelo del mundo, puede resultar una excelente herramienta de depuración en la etapa de pruebas del TDA.
- 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 TDA 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 TDA sea responsable de su propia administración.

Tipos de datos abstractos parametrizados

El desarrollar un TDA 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 solo el diseño, de tal forma que sea posible contar con librerías genéricas perfectamente portátiles, capaces de contener elementos de cualquier tipo. La sintaxis para especificar un TDA paramétrico se puede apreciar en el ejemplo siguiente.

Ejemplo:

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

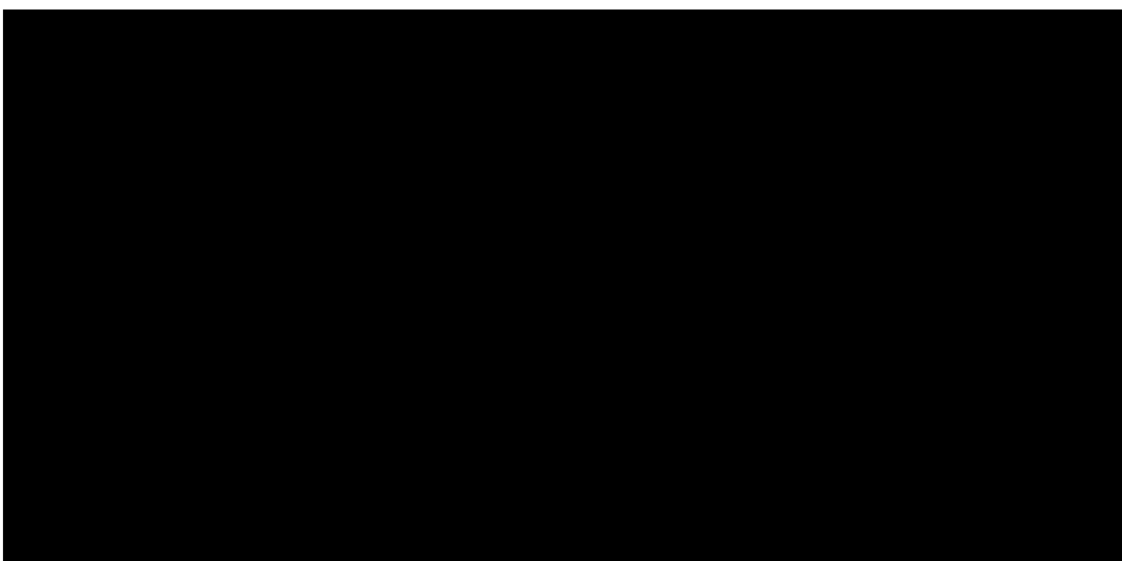


Figura 2.10. TDA Conjunto con sus Elementos

2.2. Arreglos

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 especifico 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 se debe especificar cómo se puede acceder a ella. Un arreglo tiene dos tipos de datos asociados, los numéricos y los caracteres. Las dos operaciones básicas a realizar en un arreglo son la extracción y la alimentación. La primera acepta un acceso a un elemento con la ayuda de un dato de tipo índice ya sea ordinario, inicializándolo desde el 0. El elemento más pequeño de un arreglo del tipo índice es su límite inferior, y el más alto su límite superior.

2.2.1 Arreglos Unidimensionales

Una estructura homogénea secuencial comúnmente se le denomina vector por estar definida por una sola dimensión y sus nodos leídos en una sola dirección. Se definirán dos Vectores de diferentes formas, la primera como un rango, y la segunda con la ayuda de una constante.

```
Var Lectura [1…100] of Array.

Const  I = 100;
Var Lectura[1..I] of Array;
```

2.2.2 Arreglos multidimensionales

Puede haber arreglos de más de dos dimensiones. Por ejemplo, uno tridimensional, que está especificado por medio de tres subíndices: e[4] [1][3] o c[4,I,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

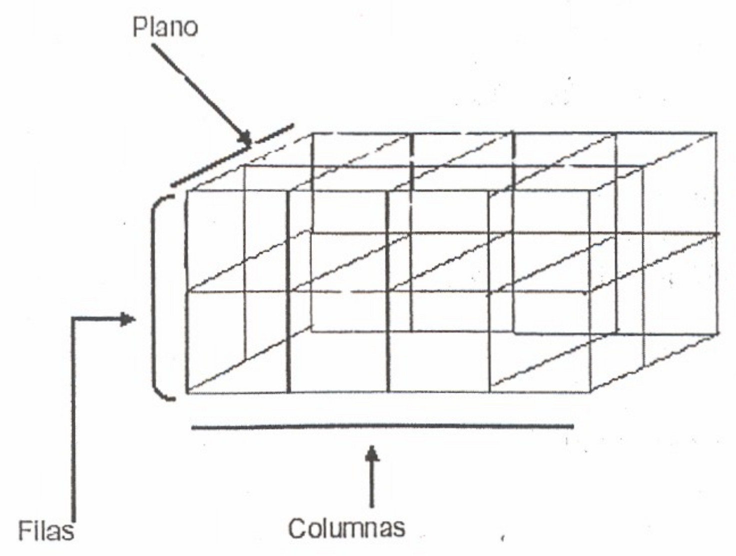


Figura 2.11. Descripción gráfica de un arreglo de tres dimensiones

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

2.2.3 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.

Arreglos bidimensionales

El tipo de un arreglo puede ser otro arreglo. Y un elemento de este arreglo puede ser ingresado 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 [2,41—, observamos que a [2J 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	X(1,1)				
FILA 2			X(2,3)		
FILA 3					X(3,5)

Figura 2.12 Arreglo de dos Dimensiones

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.

2.3 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.

2.3.1 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, ubicados por 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.

2.3.2 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).

2.3.3 Implantación de una pila

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

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.

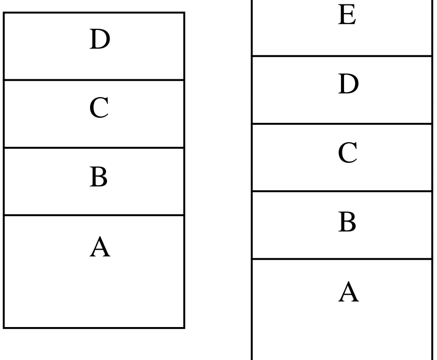


Figura 2.13. Movimiento en la Pila

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
```

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.

Operación para remover el último elemento insertado en la pila

La operación *stackpop* (avance de elementos) determina el elemento superior de la pila: basta con retirarlo y reasignar el valor del elemento de la parte superior de la pila.

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 de forma independiente.

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 la 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). En este arreglo unidimensional las operaciones son por un extremo. El extremo en si, se puede cambiar a juicio del programador, pero ya definido, solo por ahí se podrá acceder a la Estructura.[2]

En la vida ordinaria se ven y emplean varios tipos de pilas, y su manipulación se hace desde su cima ya sea para eliminar o suprimir elementos. La posición inicial con el índice con valor a cero (I = 0), el cual se va incrementando conforme se le agregan elementos. Al momento de querer acceder o suprimir

[2] Ana Ma. Toledo Salinas, "¿Qué es pilas?", material en línea, disponible en: http://boards4.melodysoft.com/app?ID=2005AEDI0303&msg=19, recuperado el 18/10/08.

un elemento, será necesario recorrer todos los elementos hasta encontrarlo y su índice será asignado al elemento inmediatamente siguiente abajo en caso de que elimine el elemento buscado.

En esta estructura, hay una serie de operaciones necesarias para su manipulación: Una de ellas es la de:
Comprobar si hay disponibilidad en la Memoria Interna para alojar una Estructura Nueva.

Crear pila

Crear el Espacio en la Memoria con New()

- 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.

Nota: Se reasignan los valores de los Índices de los Elementos restantes.
El anterior proceso no es extensivo, sino ilustrativo de los preparativos para crear una pila.

En el Leguaje Pascal con la Instrucción New(); se crea un espacio en la memoria para después asignarle un tamaño, un formato y un nombre (identificador). Cuando se quiera eliminar la Estructura, eso se hace con la instrucción Dispose():

Representación de las pilas

Los lenguajes de programación de alto nivel no suelen disponer de un tipo de datos pila. Por 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 de todos y cada uno de los elementos individuales (ítems) de la Estructura mediante el empleo de índices.

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 con un tamaño determinado.
El Indice define la extensión a priori de la Estructura.

Se puede definir una pila como una variable:

Pila: array (1..n) de T

donde el 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]

Implantación dinámica de una pila

La pila incorpora la inserción y supresión de elementos, por lo que esta es un objeto dinámico constantemente variable. La definición especifica que 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 DINÁMICAS

PUSH: insertar un elemento

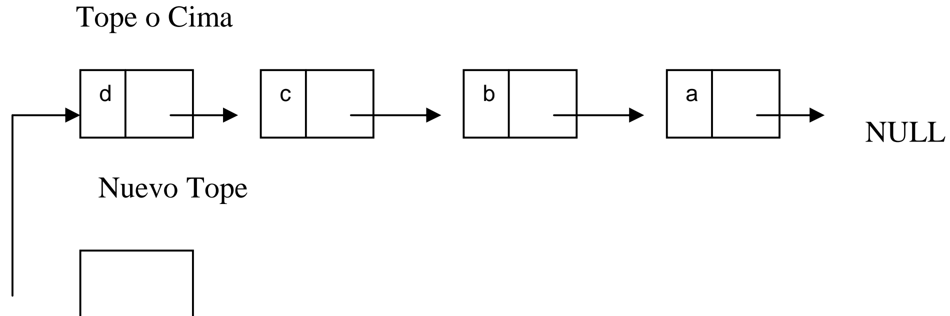


Figura 2.14a Inserción en una Pila

[3] Rigel Galeón, “Estructuras dinámicas de datos”, p. 5, material en línea, disponible en: rigel.galeon.com/dinamicas.doc, recuperado el 18/10/08.

PILAS DINÁMICAS

Función para insertar elementos –PUSH-

```
void push (PIRSTACK *Tope int valor}
        {
          PIRSTACK Nuevo:

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

PILAS DINÁMICAS

PUSH: insertar un elemento

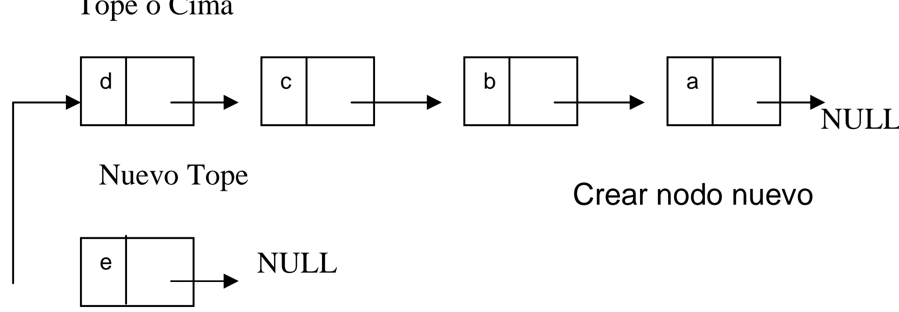


Figura 2.14b. Inserción en una Pila

PILAS DINÁMICAS

Estructura para pilas dinámicas

```
Struct stack {
     Int dato;
     Struct stack *siguiente;
```

```
} ;

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

2.4 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.

2.4.1 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.

2.4.2 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 final.
3. Eliminar su primer elemento.
4. Vaciarla.
5. Verificar su estado.

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
items: 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 una sobrecarga (*overflow)* si ésta posee más elementos de los que fueron reservados en el arreglo. Ignorando la posibilidad de la subcarga y sobrecarga por el momento, la operación *insert* (q, x) se podría implementar mediante las declaraciones siguientes:

```
q.rear: = q.rear +1;
q.items[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 2.15a ilustra un arreglo de cinco elementos utilizados para representar una cola (maxqueue = 5). Inicialmente, la cola está vacía (figura 2.15aa). En la figura 2.15b, se han agregado o insertado los elementos A, B y C; en la 2.15ab, retirando dos elementos; y en la 2.15bd, se han insertado:

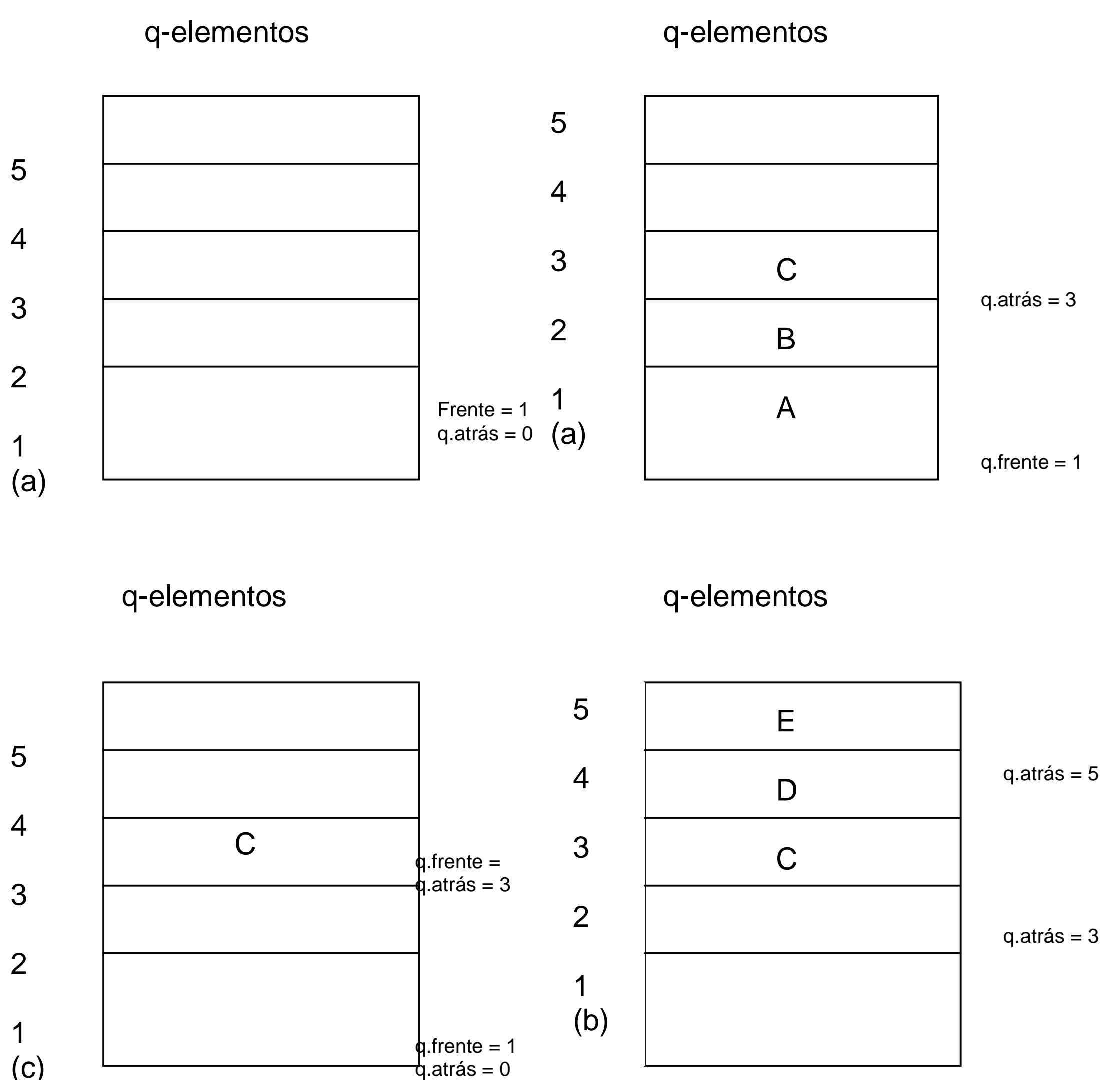
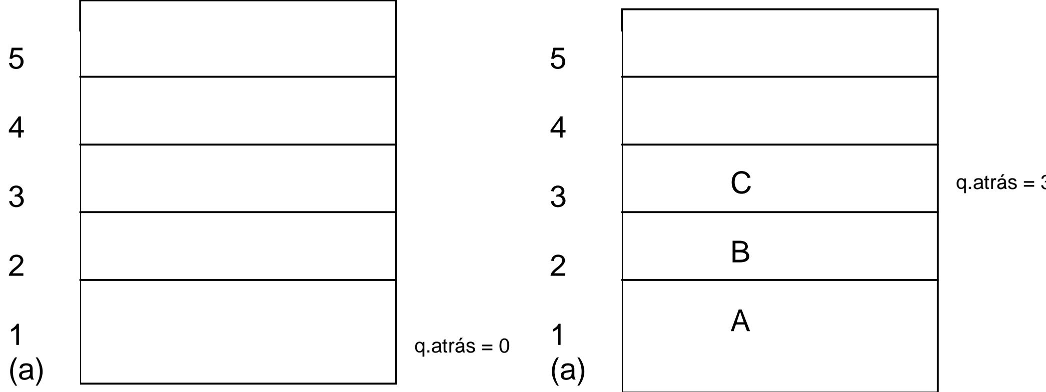


Figura 2.15a Arreglo



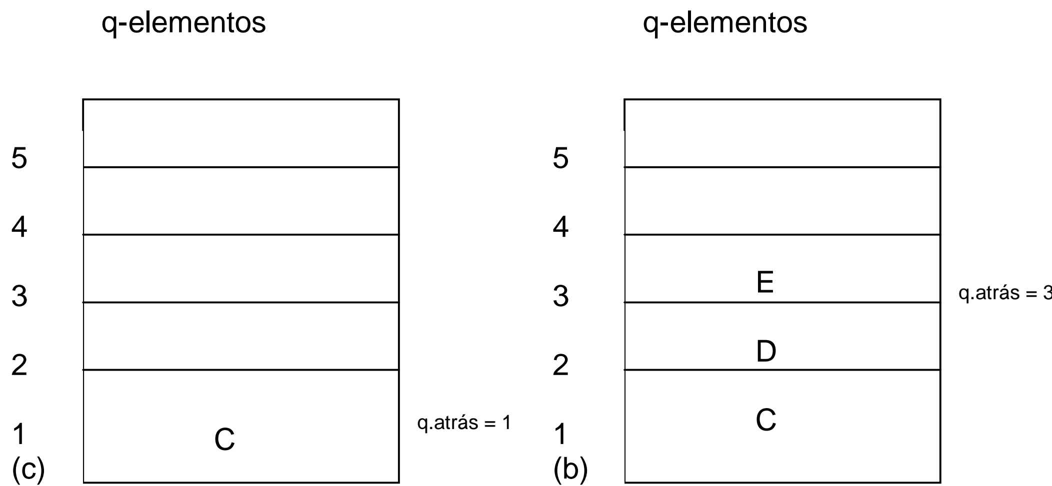


Figura 2.15b Movimientos en Arreglo

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 una sobrecarga. 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.

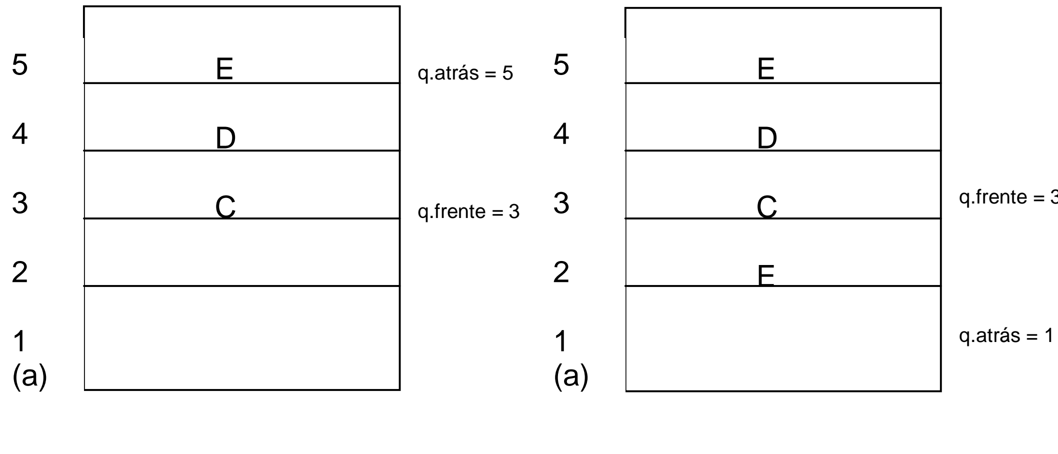
Una solución a este problema consiste en modificar la operación *remove* de tal manera que cuando se retire un elemento, se desplace toda la cola al principio del arreglo. La operación x; = *remove* (q) podría entonces ser modificada (nuevamente ignorando la posibilidad de bajoflujo) a:

```
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 2.16a presenta los elementos sin inserción y con el elemento a insertar se presenta la cola de la figura 2.16d y el resultado final de la operación en la figura 2.16e.

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.



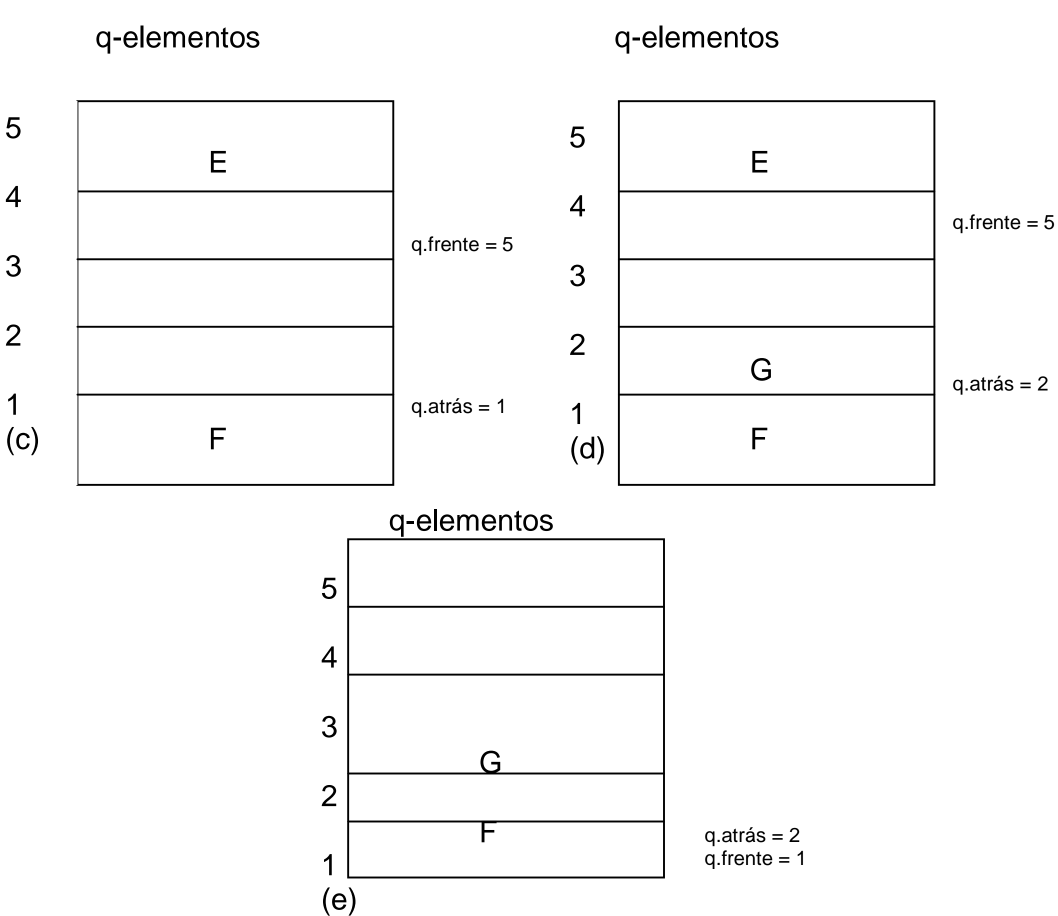


Figura 2.16 Arreglos de q Elementos

Consideremos un ejemplo. Asumamos que una cola contiene tres elementos en las posiciones 3, 4 y 5 de un arreglo de cinco elementos. Esta situación de muestra en la 2.16a reproducida como la figura 2.16b. 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 2.16a. 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 2.16b y c se muestran el estado de la cola cuando los dos primeros elementos, C y D, son retirados, G, insertado y, finalmente, E, retirado.

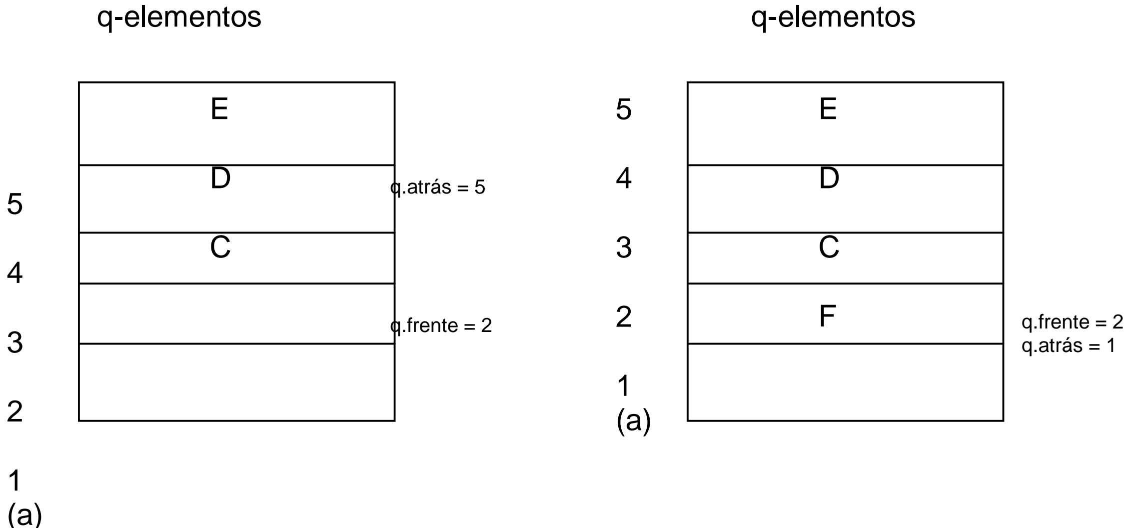
Operación para insertar un elemento a una cola

La operación *insert* se encarga de la sobrecarga (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 2.17a; 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.items (5), q.rear = 5. Y ya que el primer elemento de la cola está en q.items (3), entonces q.front es igual a 2. En las figuras 2.17a y b, 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 sobrecarga Pero esto está marcado por el hecho de que q.front = q.rear (indicación de una subcarga). 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 sobrecarga. La rutina *insert* puede escribirse como sigue:

```
Procedure insert (var q: queue; x: interger);

begin
```



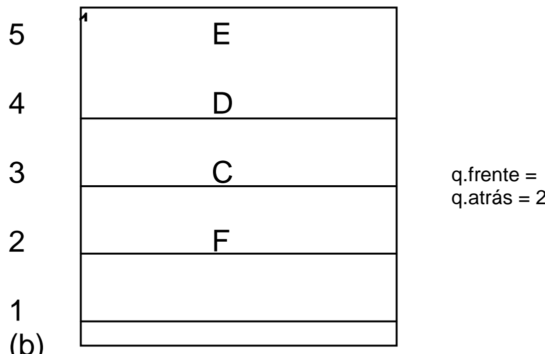


Figura 2.17 Inserción en un Arreglo

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

La evaluación para sobrecarga en la rutina *insert* se presenta después de que se le ha asignado el valor a q.rear, mientras que la prueba para la subcarga en la rutina de *remove* inmediatamente al entrar la rutina, antes de actualizar q-front1.

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 sobrecarga.

Operación para revisar si una cola está vacía
Sin embargo, 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 está 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 primer 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 ,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
          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:

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

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

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)
```

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

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

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.

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 elemento 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;
```

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 mínimo de *apq* y regresa a su valor). La segunda es similar a la anterior, pero sólo se permite la eliminación del elemento *mayor*. Las operaciones aplicables a una cola de este tipo, *dqp*, 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.

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* (adecuar con los parámetros necesarios) 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             empy (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.

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 límite 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 prioridad1. 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 cola2.

Una cola de prioridad es una estructura de datos que permiten 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:

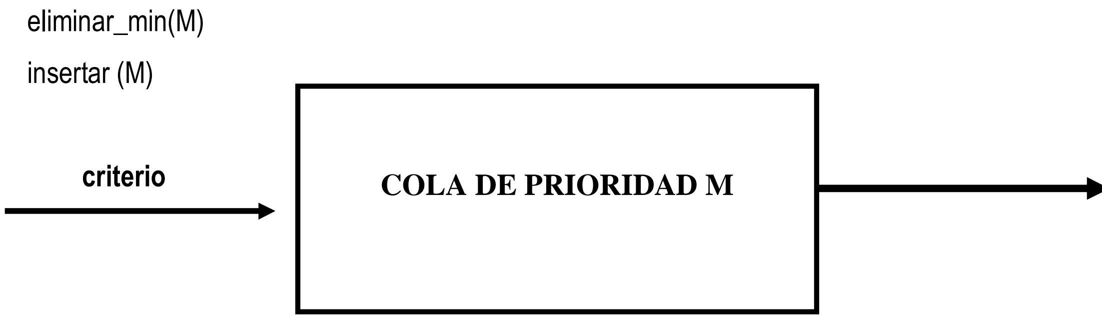


Figura 2.18 Cola con Prioridad

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:[4]

[4] Véanse, Mark Allen Weiss, *Estructura de datos y algoritmos*, México, Addison-Wesley Iberoamericana, 1995, p.180.; y Luis Joyanes Aguilar, *Fundamentos de programación: algoritmos y estructura de datos,* 2ª ed., México, McGraw-Hill, 1996, p.407.

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

                    else
                    begin
(3)                            M.tamaño := M.tamaño +1
(4)                         i :=M.tamaño;
(5)
                         while M.elementos (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 insertar realizando intercambios repetidos hasta establecer el orden correcto (recordemos que un intercambio requiere tres enunciados de asignación). Si un elemento es filtrado hacia arriba de niveles, el número de asignaciones efectuadas por los intercambios podría ser 3d. Así, nuestro método aplicado usa d + 1 arriba en el árbol en algún punto, y será 1, y se deseará romper el ciclo *boole(loop).* 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 *boole* 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 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.

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*/
```

2.4.3 Bicolas

Existe una variante de las colas simples, la doble cola o bicola. Esta 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*.

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.

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.
InsertIzq (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.

Operación para construir una bicola vacía

```
with unchecked_deallocation;
package body bicolas is
     procedure   disponer   is   new   uncheckned_deallocation
(unDato,ptUnDato);
     procedure creaVacia (b:out bicola) is
     begin
          b:= (null,null,0):
     end creaVacia;
```

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 es
begin
     return b.der.dato;
end observaDer;
function esVacia (b:in bicola) return  oolean is
begin
     return b.n=0;
end esVacia;
function long (b:in bicola) return natural is
begin
     return b.n;
end long;
```

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 es
begin
     return b.der.dato;
```

```
end observaDer;
function esVacia (b:in bicola) return  oolean is
begin
     return b.n=0;
end esVacia;
function long (b:in bicola) return natural is
begin
     return b.n;
end long;
```

Operación para obtener el elemento que está al final 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;
```

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;
```

Operación para insertar un elemento al inicio de la bicola

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

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 es
begin
return b.der.dato;
end observaDer;
function esVacia (b:in bicola) return  oolean is
begin
return b.n=0;
end esVacia;
function long (b:in bicola) return natural is
begin
return b.n;
end long;
```

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;
creVacia (b);
end libera;
```

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 bicola; e:in out 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  oolean;
function long (b:in bicola) return natural;
procedure asigna (bout: out bicola; bin:in bicola);
procedure libera (b:in out bicola);
 oolean "=" (b1, b2:in bicola) oolea oolean;
```

```
private
type unDato;
type ptDato 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;
```

2.4.4. Implantación de una Cola

En el ámbito de la Informática, el término Implantar se le da la connotación de adecuar las características de un compilador, el cual no tiene definida o precargada una Estructura determinada para que por medio de algoritmos y estructuras ya definidas por el compilador se pueda realizar alguna aplicación requerida. En este caso, en el Lenguaje C, se implantará una Cola con la ayuda de los Punteros.

```
#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 a la zona de memoria */
free (ptr);
```

Los Registros son Estructuras cuyos elementos son de diferentes tipos o formatos a diferencia de los Arreglos, los cuales están constituidos por ítems del mismo tipo. Los Vectores se manipulan en la Memoria Interna de la Computadora, los Arreglos y Registros se almacenan y se invocan desde la

Memoria Externa. Los Arreglos y los Registros están definidos por Estructuras en varios Lenguajes en la Actualidad. Se crean Arreglos y Registros de forma separada o combinada de acuerdo a las necesidades, dando lugar a diferentes manejos de los datos, tanto en la memoria interna de la computadora como en los compiladores actuales, sin mencionar que las imágenes son también objetos, los cuales no son objeto en la presente asignatura.

Registros

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 *ci* que cada uno de ellos se identifica por medio de su propio campo.

Acceso a los campos de un registro

Tenemos una variable llamada Articulo: 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:

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

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

LIBRO.DBF

El registro
completo se llama:

Artículos

Los campos de registro se
llaman:

Artículos. Autor
Artículos. Título
Artículos. Pubperiódica

Artículos. Volumen
Artículo. Página
Artículo. Año

Figura 2.19ª. Componentes del Registro Libro

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 dar valor constante a una variable de registro; hay que dar valores individualmente a los diversos campos.

Artículo. Autor := ‘Paz, Octavio.

Artículo. Titulo := ‘Aura ‘;

Artículo. PubPeriódica := ‘Letras Vivas’

Artículo. Volumen := 11;

Artículo. Página :=147;

Artículo. Año := 1968;

Libro.DBF

Figura 2.19b. **Componentes del Registro Libro**

La única operación de registro completo que ofrece Pascal es el enunciado de asignación: si el Articulo 1 y Articulo 2 son dos registros exactamente del mismo tipo (en este ejemplo, del tipo Articulo Publicado), podemos asignar el valor de uno a otro:

Articulo1 := Articulo2;

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

Articulol. Autor := Articulo2. Autor;

Articulol. Titulo := Articulo2. Titulo;

Articulo1. PubPeriódica := Articulo2. PubPeriódica;

Articulol. Página := Articulo2. Página;

Articulol. Ensayos := Articulo2. Ensayos;

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 oraciones anteriores, a continuación escribiremos nuestro propio procedimiento para leer un registro:

Lee del teclado datos bibliográficos de un Articulo publicado; Se leen seis campos:

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

Otros procedimientos requeridos:

```
          ReadLnCadena (lee un valor cadena 40);

PROCEDURE Leer ArticuloPublicado (VAR ArticuloPublicado);

          BEGiN
          WriteLn (‘ Autor o Autores: (40 caracteres como
          máximo)’);
          ReadLnCadena (Articulo. Autor);
          WriteLn (‘Titulo del Articulo: 40 caracteres como
          máximo)’);
          ReadLnCadena (Artl. Titulo);
          WriteLn ( Nombre de la publicación: (40
          caracteres como máximo)’);
          ReadLnCadena (Artl. PubPeriódica);
          WriteLn (‘Nümero de volumen: (entero)’);
          ReadLn (Articulo.Volumen);
          WriteLn ( ‘N1’mero de página inicial: (entero)’ )
          ReadLn (Articulo. Página);
          WriteLn (‘Año: (entero de cuatro digitos)’);
          ReadLn (Articulo.Aflo);
          Wri teLn
          END; {fin de leer ArticuloPubljcado }
```

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 tiene un valor por asignación, o mediante Leer el Registro ArticuloPublicado podremos exhibir su contenido con la ayuda de un procedimiento como el que mostramos a continuación (darnos por hecho que ya disponemos de un procedimiento EscribirCadena para exhibir valores de tipo cadena 40):

1. Exhibe datos bibliográficos de un artículo publicado.
2. Otros procedimientos requeridos:

```
EscribirCadena (escribe un valor cadena 40)
PROCEDURE EscribirArticuloPublicado ( Articulo:
ArticuloPublicado );
     BEGÍN

               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) : Rulfo, Juan.
Título: Pedro Páramo
Publicación: Fondo de Cultura Económica
Volumen: 1
Página: 10
Año: 1974

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;
Titulo :cadena40;
```

```
PubPeriodica : cadena40;
Volumen: Integer;
Pagina: Integer;
Año : Integer;
END;

Bibliografia = ARRAY [1..TamañoBiblio ] OF
ArticuloPublicado;
VAR
Biblia: Bibliografia;
```

Estas declaraciones describen un arreglo Biblia 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 Biblia.
Bibilio [1]. Autor es el nombre del primer campo del primer registro. Es una cadena de 40 caracteres.
Biblio [1]. Titulo es el nombre del segundo campo del primer registro. Es una caden2 de 40 caracteres.
Biblio [1]. PubPeriodica 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.

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, ingresamos un valor a cada campo del registro por separado.

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.

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: {xl, x2,…,xN}: sup`
- `Invariante: inf < X¡< sup /* todos los. Elementos están en el rango [inf…sup ]*/`

```
              xi ¡= xk, i!= k /*no hay elementos
repetidos */
         1 < inf < 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 clem);
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 < infv elem> sup, insertarConj=
RANGO)
 v
(Ei / xl = elem, insertarCo_j= DUPLI)}
eliminarConj {error: elem! = Xj Aj, 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 supenorConj (Conjunto conj);
```

- Manejo de error: ninguna analizadora puede fallar.

Operaciones interesantes: se colocan operaciones para copiar, comparar, visualizar y destruir. .

```
int igualConj (conjunto cl, conjunto c2); /* infonna si
cl=c2*/
int subConj (conjunto el, conjunto c2); /*infonna 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);
/*nÚInero de elementos de un conj*/
Void unirConjunto (conjunto el, conjunto c2); /*cl=
clUc2*/
```

- 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

{pre: 1 < infer < super}
{post: crearconj = infer: Jl-=-_er}

Int insertarconj (conjunto conj, int elem )

/* inserta al conjunto un elemento válido */


{pre: conj = inf: {xl, x2,…, xN}: sup, elem¡=X¡ A¡.inf <
elem < sup}
{post: conj = inf: {xl,…, xN, elem}: sup}
```

Int eliminarconj (conjunto conj, int elem)

/* elimina un elemento del conjunto * /

{pre: conj = inf: {xl, x2,…, xN}: sup, X¡ = elem}

{post: conj = inf: {xl,…, xi-l, xi+l,…, xN}: sup}

Int estaconj (conjunto conj, int elem)

/* informa si un elemento se encuentra en el conjunto */

{post: estaconiunto == E¡ /xi = elem)}

Int inferiorconj (conjunto conj)

/* retorna el límite inferior del rango de valores válidos del conjunto */

{post: inferiorConj = ini)

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

{post: superiorConj = sup}
```

2.5 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 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:

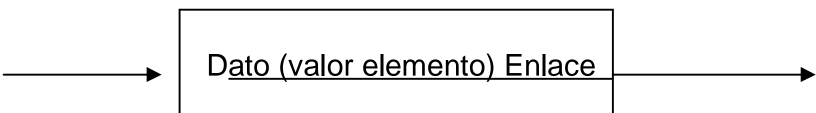


Figura 2.20 Enlace y Encadenamiento.

2.5.1 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 encogido 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.

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) 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.

Figura 2.21 Lista representada como Arreglo

2.5.2 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 un 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.

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-ésimo está dentro del rango permitido de los elementos 1< = j<1; en este caso, asignar el valor del elemento P(j) a una variable B; si el elemento j-ésimo 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
ListainicLista(void)
/ * Crea y retorna una lista vacía * /
post:inicLista =<>
```

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:

Casos de Inserción en la Lista

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.

Primer Caso. Inserción al Inicio de la Lista

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

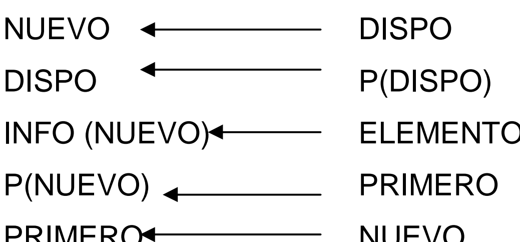
   Algoritmo inserción

   Comprobación de sobrecarga

   Si DISPO=NULO

   Entonces, escribir “sobrecarga”

   Si no:

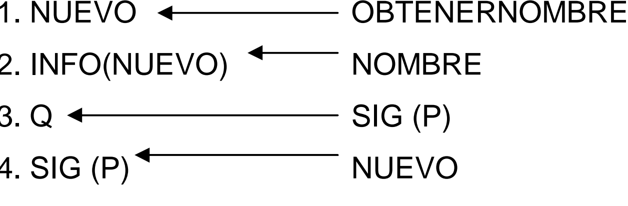


2. Fin_si

Segundo Caso. Inserción a continuación de un nodo específico.

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:



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 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).*

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 sobrecarga }

Si DISPO = NULO

Entonces, escribir "sobrecarga"

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

Operación para eliminar la cabeza de una lista

El algoritmo que elimina de la lista enlazada el elemento siguiente apuntando 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)

2.5.3 Implantación de una lista

Es una implantación con base en arreglos que permite visualizar la 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.

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 *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: arrary (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.

2.6 Tablas de Dispersión y Función Hash

En la operación de búsqueda de elementos dentro de estructuras, el empleo de llaves o índices se hace muy necesario para compaginar el valor de índice con el valor buscado, de tal forma que esta operación puede llevar tiempo hasta que coincidan. Un método diferente para realizar la búsqueda calcula la posición de la llave en la tabla con base en el valor de la llave. Para acelerar esta operación, se necesita encontrar una función *h* que pueda transformar una llave particular *k*, como cadena, número o registro en un índice de la tabla usado para almacenar elementos del mismo tipo que *k*. La función *h* se llama *función hash o función de dispersión.* Si *h* transforma diferentes llaves en números distintos se llama

> *función hash perfecta y* debe coincidir el mismo número de posiciones posiciones que el número de elementos que se está transformando. transformando. Pero el número de elementos no siempre se conoce *a* conoce *a priori.* Con 1000 celdas se pueden almacenar los nombres de nombres de las variables de un programa.[5]

Las llaves no necesitan almacenarse en la misma tabla, cada posición de la posición de la tabla se asocia con una lista ligada o cadena de estructuras estructuras cuyos campos de información almacenan llaves o referencias a las referencias a las llaves. Este método se llama *encadenamiento separado o separado o dispersión cerrada* y una tabla de apuntadores se llama *tabla de* llama *tabla de dispersión.* Con este método “no hay sobrecarga debido a que debido a que las listas ligadas se amplían solo una vez que llegan las llaves llegan las llaves nuevas. En la medida que aumentan los elementos, el tiempo elementos, el tiempo de recuperación aumenta”.[6]

Si la longitud de los vectores en fija, los tiempos de recuperación se mantienen recuperación se mantienen constantes. La función de dispersión debe ser fácil debe ser fácil de calcular y, asegurar que dos claves distintas se correspondan correspondan con celdas diferentes y debe también distribuir homogéneamente homogéneamente las claves entre celdas y detectar espacios dentro de la tabla dentro de la tabla para saber si la tabla debe crecer o no. Resta escoger una escoger una función y decidir el tamaño de la tabla y qué hacer cuando dos cuando dos claves caen en la misma celda. Las tablas de dispersión se usan dispersión se usan para representar diccionarios en los que se busca una clave busca una clave y se devuelve su definición, es decir, la letra inicial de la inicial de la palabra debe coincidir con el contenido de índice del diccionario diccionario para que la búsqueda sea eficiente.

valor que un elemento ya insertado, tenemos una **colisión** y hay que que resolverla. La Colisión ocurre cuando dos apuntadores apuntan a una

---

[5] DROZDEK, A. (2007). *Estructura de Datos y Algoritmos en Java* (Segunda ed.). México: Thomson Learning, p. 518.

[6] Ibid., p. 528.

apuntan a una misma dirección provocando conflicto para el almacenamiento almacenamiento de la información en la página de memoria interna de la interna de la computadora.

Por ejemplo, si las claves son números enteros, *clave mod N* es una función: y una función: y el valor será un valor entero. Si *N* fuese 100 y todas las claves todas las claves terminasen en cero, esta función de dispersión sería una mala sería una mala opción. Si el resultado de Mod genera un número primo, habría número primo, habría menos colisiones. Si se generan números enteros de enteros de forma aleatoria, las claves en la tabla se distribuirán de forma de forma uniforme. Por lo general, las claves son cadenas de caracteres cuya caracteres cuya longitud depende de las propiedades y restricciones de la restricciones de la función *mod.*

A continuación, se anotará una forma de generar claves.

```
   Una opción es sumar los valores ASCII de los caracteres.
caracteres.
función Dispersión1 (Clave, TamañoClave): ´Indice
valor := ascii(Clave[1]);
para i := 2 hasta TamañoClave hacer
valor := valor + acii(Clave[i])
fin para
devolver valor mod N
fin función
```

Es una función fácil de implementar, y se ejecuta con rapidez. Pero si el Pero si el tamaño de la tabla es grande, esta función no distribuye bien las distribuye bien las claves.

Por ejemplo, si $N = 10007$ y las claves tienen 8 caracteres, la función sólo toma función sólo toma valores entre 0 y 1016 = 127 ·8.

[7] Facultad de Informática. Universidad de A Coruña. *Algoritmos*. http://quegrande.org/apuntes/EI/2/Alg/teoria/07-08/tema_2.4_-_tablas_de_dispersion.pdf octubre 18, 2008.

```
función Dispersión2 (Clave, TamañoClave): ´Indice
devolver (ascii(Clave[1]) + 27*ascii(Clave[2])
+ 729*ascii(Clave[3])) mod N
fin función
```

En esta función de dispersión se supone que la clave tiene al menos tres menos tres caracteres. Si los primeros caracteres son aleatorios y el tamaño de y el tamaño de la tabla es 10007, esperaríamos una distribución bastante bastante homogénea. Lamentablemente, los lenguajes naturales no son naturales no son aleatorios. Aunque hay 273 = 17576 combinaciones posibles, combinaciones posibles, en un diccionario el número de combinaciones combinaciones diferentes que nos encontramos es menor que 3000. Sólo un 3000. Sólo un porcentaje bajo de la tabla puede ser aprovechada por la aprovechada por la dispersión

Función Hash[8]

Tabla hash

> Una tabla **hash** o **mapa hash** es una estructura de datos que asocia asocia *llaves* o *claves* con *valores*. La operación principal que soporta de soporta de manera eficiente es la *búsqueda*: permite el acceso a los a los elementos (teléfono y dirección, por ejemplo) almacenados a partir a partir de una clave generada (usando el nombre o número de cuenta, cuenta, por ejemplo). Funciona transformando la clave con una **función función hash** en un *hash*, un número que la tabla hash utiliza para para localizar el valor deseado.

[8] Esta sección, con todo y las figuras, fue recuperada de Wikipedia en la entrada “Tabla hash”, actualizada el 27/07/08, disponible en: http://es.wikipedia.org/wiki/Tabla_hash, fecha de recuperación: octubre 18 del 2008.

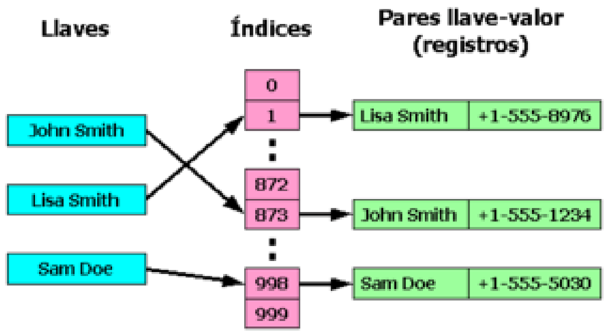


Figura No. 2.22 .Ejemplo de tabla hash

Las **tablas hash** se suelen implementar sobre *arrays* de una dimensión, dimensión, aunque se pueden hacer implementaciones multi-dimensionales basadas en varias claves. Como en el caso de los arreglos, los arreglos, las tablas hash proveen tiempo constante de búsqueda búsqueda promedio O, sin importar el número de elementos en la tabla. la tabla. Sin embargo, en casos particularmente malos el tiempo de de búsqueda puede llegar a O(*n*), es decir, en función del número de número de elementos.

Comparada con otras estructuras de arreglos asociadas, las tablas hash tablas hash son más útiles cuando se almacenan grandes cantidades de cantidades de información.
Las tablas hash almacenan la información en posiciones pseudo-pseudo-aleatorias, así que el acceso ordenado a su contenido es bastante es bastante lento. Otras estructuras como árboles binarios auto-balanceables son más lentos en promedio (tiempo de búsqueda O(log *n*)) O(log *n*)) pero la información está ordenada en todo momento.

Funcionamiento

Las operaciones básicas implementadas en las tablas hash son:
inserción(llave, valor)
búsqueda(llave) que devuelve valor

La mayoría de las implementaciones también incluyen borrar(llave).

borrar(llave). También se pueden ofrecer funciones como iteración en la en la tabla, crecimiento y vaciado. Algunas tablas hash permiten almacenar múltiples valores bajo la misma clave.

Para usar una *tabla hash* se necesita:

- Una estructura de acceso directo (normalmente un *array*).
- Una estructura de datos con una clave
- Una función resumen (*hash*) cuyo dominio sea el espacio de claves y su claves y su imagen (o rango) los números naturales.

Inserción

Para almacenar un elemento en la tabla hash se ha de convertir su clave clave a un número. Esto se consigue aplicando la función resumen a la a la clave del elemento.

0. El resultado de la función resumen ha de *mapearse* al espacio de de direcciones del *array* que se emplea como soporte, lo cual se consigue consigue con la función módulo. Tras este paso se obtiene un índice índice válido para la tabla.
0. El elemento se almacena en la posición de la tabla, obtenido en el paso el paso anterior.
0. Si en la posición de la tabla ya había otro elemento, se ha producido una producido una colisión. Este problema se puede solucionar asociando asociando una lista a cada posición de la tabla, aplicando otra función o función o buscando el siguiente elemento libre. Estas posibilidades han posibilidades han de considerarse a la hora de recuperar los datos. datos.

Búsqueda

Para recuperar los datos, es necesario únicamente conocer la clave del del elemento, a la cual se le aplica la función resumen.

0. El valor obtenido se mapea al espacio de direcciones de la tabla. tabla.
0. Si el elemento existente en la posición indicada en el paso anterior tiene anterior tiene la misma clave que la empleada en la búsqueda, entonces entonces es el deseado. Si la clave es distinta, se ha de buscar el el elemento según la técnica empleada para resolver el problema de las

problema de las colisiones al almacenar el elemento.

Prácticas recomendadas para las funciones hash

Una buena función hash es esencial para el buen rendimiento de una tabla hash. Las colisiones son generalmente resueltas por algún tipo de búsqueda lineal, así que si la función tiende a generar valores similares, las búsquedas resultantes se vuelven lentas.

En una función hash ideal, el cambio de un simple bit en la llave (incluyendo el hacer la llave más larga o más corta) debería cambiar la mitad de los bits del hash, y este cambio debería ser independiente de los cambios provocados por otros bits de la llave. Como una función hash puede ser difícil de diseñar, o computacionalmente cara de ejecución, se han invertido muchos esfuerzos en el desarrollo de estrategias para la resolución de colisiones que mitiguen el mal rendimiento del *hasheo*. Sin embargo, ninguna de estas estrategias es tan efectiva como el desarrollo de una buena función hash de principio.

Es deseable utilizar la misma función hash para arreglos de cualquier tamaño concebible. Para esto, el índice de su ubicación en el arreglo de la tabla hash se calcula generalmente en dos pasos:

1. Un valor hash genérico es calculado, llenando un entero natural de máquina
2. Este valor es reducido a un índice válido en el arreglo encontrando su módulo con respecto al tamaño del arreglo.

El tamaño del arreglo de las tablas hash es con frecuencia un

número primo. Esto se hace con el objetivo de evitar la tendencia de que tendencia de que los hash de enteros grandes tengan divisores comunes comunes con el tamaño de la tabla hash, lo que provocaría colisiones tras colisiones tras el cálculo del módulo. Sin embargo, el uso de una tabla de una tabla de tamaño primo no es un sustituto a una buena función hash. función hash.

Un problema bastante común que ocurre con las funciones hash es el es el aglomeramiento. El **aglomeramiento** ocurre cuando la estructura de estructura de la función hash provoca que llaves usadas comúnmente comúnmente tiendan a caer muy cerca unas de otras o incluso consecutivamente en la tabla hash. Esto puede degradar el rendimiento rendimiento de manera significativa, cuando la tabla se llena usando usando ciertas estrategias de resolución de colisiones, como el sondeo sondeo lineal.

Cuando se depura el manejo de las colisiones en una tabla hash, suele hash, suele ser útil usar una función hash que devuelva siempre un valor un valor constante, como 1, que cause colisión en cada inserción. inserción.

Resolución de colisiones

Si dos llaves generan un hash apuntando al mismo índice, los registros registros correspondientes no pueden ser almacenados en la misma misma posición. En estos casos, cuando una casilla ya está ocupada, ocupada, debemos encontrar otra ubicación donde almacenar el nuevo nuevo registro, y hacerlo de tal manera que podamos encontrarlo cuando encontrarlo cuando se requiera.

Para dar una idea de la importancia de una buena estrategia de resolución resolución de colisiones, considérese el siguiente resultado, derivado de la derivado de la *paradoja de las fechas de nacimiento*. Aún cuando cuando supongamos que el resultado de nuestra función hash genera genera índices aleatorios distribuidos uniformemente en todo el arreglo, e

arreglo, e incluso para arreglos de 1 millón de entradas, hay un 95% de 95% de posibilidades de que al menos una colisión ocurra antes de de alcanzar los 2500 registros.

Hay varias técnicas de resolución de colisiones, pero las más populares populares son **encadenamiento y direccionamiento abierto**.

Encadenamiento

En la técnica más simple de encadenamiento, cada casilla en el arreglo arreglo referencia una lista de los registros insertados que colisionan en la colisionan en la misma casilla. La inserción consiste en encontrar la casilla la casilla correcta y agregar al final de la lista correspondiente. El borrado borrado consiste en buscar y quitar de la lista.

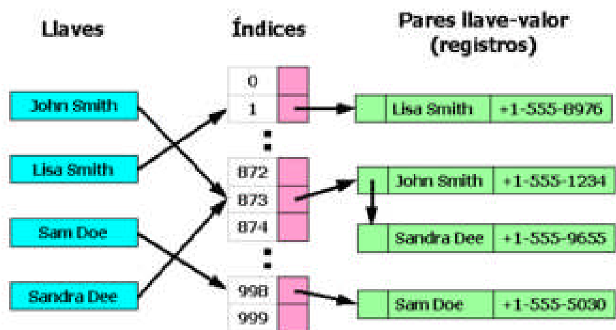


Figura 2.23. Ejemplo de encadenamiento

La técnica de encadenamiento tiene ventajas sobre direccionamiento direccionamiento abierto. Primero el borrado es simple y segundo el segundo el crecimiento de la tabla puede ser pospuesto durante mucho mucho más tiempo dado que el rendimiento disminuye mucho más más lentamente incluso cuando todas las casillas ya están ocupadas. De ocupadas. De hecho, muchas tablas hash encadenadas pueden no no requerir crecimiento nunca, dado que la degradación de rendimiento es rendimiento es lineal en la medida que se va llenando la tabla. Por Por ejemplo, una tabla hash encadenada con dos veces el número de elementos recomendados, será dos veces más lenta en promedio que la

promedio que la misma tabla a su capacidad recomendada.

Las tablas hash encadenadas heredan las desventajas de las listas listas ligadas. Cuando se almacenan cantidades de información pequeñas, el gasto extra de las listas ligadas puede ser significativo. significativo. También los viajes a través de las listas tienen un rendimiento de caché muy pobre.

Otras estructuras de datos pueden ser utilizadas para el encadenamiento encadenamiento en lugar de las listas ligadas. Al usar árboles auto-auto-balanceables, por ejemplo, el tiempo teórico del peor de los casos los casos disminuye de O(*n*) a O(log *n*). Sin embargo, dado que se se supone que cada lista debe ser pequeña, esta estrategia es normalmente ineficiente a menos que la tabla hash sea diseñada para diseñada para correr a máxima capacidad o existan índices de colisión colisión particularmente grandes. También se pueden utilizar arreglos arreglos dinámicos para disminuir el espacio extra requerido y mejorar el mejorar el rendimiento del caché cuando los registros son pequeños. pequeños.

Direccionamiento abierto

Las tablas hash de direccionamiento abierto pueden almacenar los los registros directamente en el arreglo. Las colisiones se resuelven resuelven mediante un *sondeo* del arreglo, en el que se buscan diferentes diferentes localidades del arreglo (*secuencia de sondeo*) hasta que el que el registro es encontrado o se llega a una casilla vacía, indicando que indicando que no existe esa llave en la tabla.

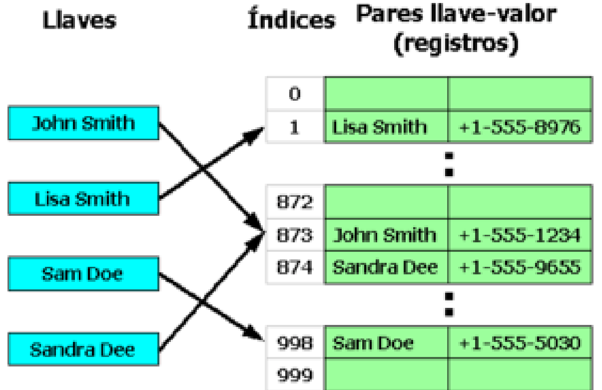


Figura 2.24 Ejemplo de direccionamiento abierto

Las secuencias de sondeo más socorridas incluyen:

- sondeo lineal
  en el que el intervalo entre cada intento es constante-- frecuentemente 1.
- sondeo cuadrático
  en el que el intervalo entre los intentos aumenta linealmente (por lo linealmente (por lo que los índices son descritos por una función cuadrática), y
- doble *hasheo*
  en el que el intervalo entre intentos es constante para cada registro registro pero es calculado por otra función hash.

El sondeo lineal ofrece el mejor rendimiento del caché, pero es más más sensible al aglomeramiento, en tanto que el doble hasheo tiene pobre tiene pobre rendimiento en el caché pero elimina el problema de aglomeramiento. El sondeo cuadrático se sitúa en medio. El doble hasheo doble hasheo también puede requerir más cálculos que las otras formas otras formas de sondeo.

Una influencia crítica en el rendimiento de una tabla hash de direccionamiento abierto es el porcentaje de casillas usadas en el arreglo. el arreglo. Conforme el arreglo se acerca al 100% de su capacidad, el

capacidad, el número de saltos requeridos por el sondeo puede aumentar aumentar considerablemente. Una vez que se llena la tabla, los algoritmos algoritmos de sondeo pueden incluso caer en un círculo sin fin. Incluso Incluso utilizando buenas funciones hash, el límite aceptable de capacidad capacidad es normalmente 80%. Con funciones hash pobremente pobremente diseñadas el rendimiento puede degradarse incluso con poca con poca información, al provocar aglomeramiento significativo. No se No se sabe a ciencia cierta qué provoca que las funciones hash generen generen aglomeramiento, y es muy fácil escribir una función hash que, sin hash que, sin querer, provoque un nivel muy elevado de aglomeramiento. aglomeramiento.

***Ventajas e inconvenientes de las tablas* hash**

Una tabla *hash* tiene como principal ventaja que el acceso a los datos datos suele ser muy rápido si se cumplen las siguientes condiciones: condiciones:

- Una razón de ocupación no muy elevada (a partir del 75% de ocupación ocupación se producen demasiadas colisiones y la tabla se vuelve vuelve ineficiente).
- Una función resumen que distribuya uniformemente las claves. Si la Si la función está mal diseñada, se producirán muchas colisiones. colisiones.

Los inconvenientes de las tablas *hash* son:

- Necesidad de ampliar el espacio de la tabla si el volumen de datos datos almacenados crece. Se trata de una operación costosa.
- Dificultad para recorrer todos los elementos. Se suelen emplear listas o listas o colecciones(Collection usadas en .net) para procesar la totalidad de los elementos .
- Desaprovechamiento de la memoria. Si se reserva espacio para todos todos los posibles elementos, se consume más memoria de la necesaria; se suele resolver reservando espacio únicamente para para punteros a los elementos.

Implementación en pseudocódigo

El pseudocódigo que sigue es una implementación de una tabla hash de hash de direccionamiento abierto con *sondeo* lineal para resolución de resolución de colisiones y progresión sencilla, una solución común que que funciona correctamente si la función hash es apropiada.

```
 registro par { llave, valor }
 var arreglo de pares casilla[0..numcasillas-1]

 function buscacasilla(llave) {
     i := hash(llave) módulo de numcasillas
     loop {
         if casilla[i] esta libre or casilla[i].llave =
casilla[i].llave = llave
             return i
         i := (i + 1) módulo de numcasillas
     }
 }

 function busqueda(llave)
     i := buscacasilla(llave)
     if casilla[i] está ocupada   // llave en la tabla
la tabla
         return casilla[i].valor
     else                        // llave es está en la
en la tabla
         return no encontrada

 function asignar(llave, valor) {
     i := buscacasilla(llave)
     if casilla[i] está ocupada
         casilla[i].valor := valor
     else {
         if tabla casi llena {
```

```
                    hacer tabla más grande (nota 1)
                    i := buscacasilla(llave)
                }
                casilla[i].llave := llave
                casilla[i].valor := valor
            }
        }
```

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

Las Estructuras de Datos son una representación abstracta de cómo la cómo la computadora manipula la información en la memoria interna para su interna para su uso posterior.

Las Estructuras de Datos deben de cumplir con ciertos requisitos para que no requisitos para que no se pierda su representatividad ni su lógica de lógica de manipulación.

Los tipos de datos más frecuentes utilizados en los lenguajes de programación programación son:

Entero (integer)

(integer)

Real (real)

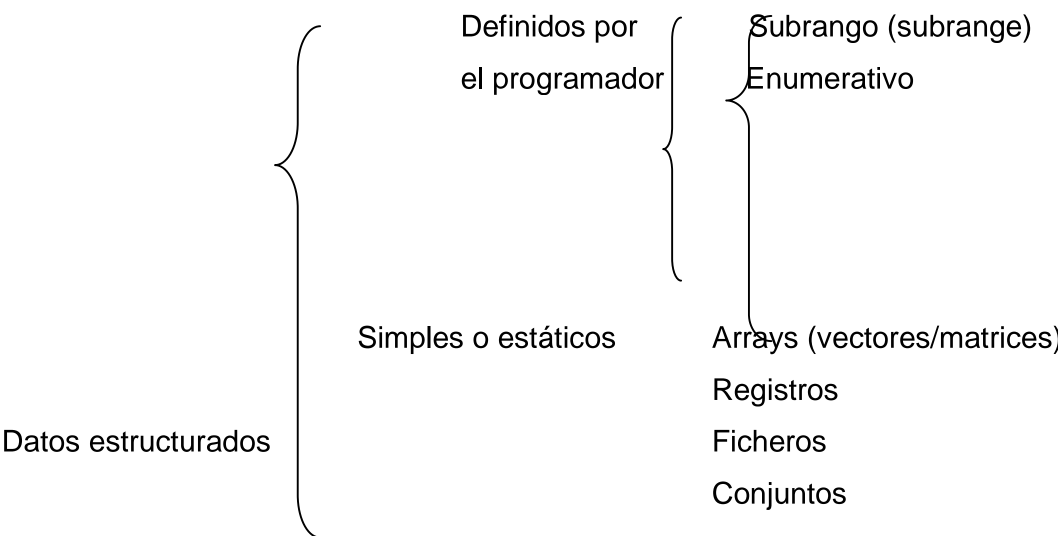
Carácter (caracter)

(caracter)

Datos simples estándar Estándar Lógico (Boolean)

(Boolean)

Float (punto flotante)



Grafos

Compuestos o dinámicos

Cadenas (string)

Listas (pilas/colas)

Lista enlaza

Árboles

Cuadro 2. 2 Clasificación de las Estructuras de Datos

Bibliografía del tema 2

Ana Ma. Toledo Salinas, "¿Qué es pilas?", material en línea, disponible en: disponible en: http://boards4.melodysoft.com/app?ID=2005AEDI0303&msg=19

Costales, Felipe, "Programación no numérica: grafos", material en línea, en línea, disponible en:

http://www.monografias.com/trabajos/grafos/grafos.shtml

Drozdek, A. (2007). *Estructura de Datos y Algoritmos en Java*, 2ª ed., México: 2ª ed., México: Thomson Learning, 2005 [isbn 0-534-49252-5].

Facultad de Informática. Universidad de A Coruña. *Estructura de datos: Tablas datos: Tablas de dispersión. Algoritmos*. Disponible en: http://quegrande.org/apuntes/EI/2/Alg/teoria/07-08/tema_2.4_-_tablas_de_dispersion.pdf

Galeón, Rigel: “Estructuras dinámicas de datos”, p. 5, material en línea, línea, disponible en: http://rigel.galeon.com/dinamicas.doc.

Hernández Castillo, Vicente. *GUÍA DIDÁCTICA DE INFORMÁTICA II.* SUA-*INFORMÁTICA II.* SUA-UNAM 1995.

Joyanes Aguilar, Luis. F*undamentos de programación.* México, McGraw Hill, McGraw Hill, México 1990.

Programación fácil, “C++ Estructuras o Registros”, material en línea, dispobible línea, dispobible en: http://www.programacionfacil.com/cpp:estructuras_registros

Weiss, Mark Allen, *Estructura de datos y algoritmos*, México, Addison-Wesley Addison-Wesley Iberoamericana, 1995.

Wikipedia, “Tabla hash”, 27/07/08, en línea, disponible n: http://es.wikipedia.org/wiki/Tabla_hash.

Actividades de Aprendizaje

A.2.1. Investigar la Aplicación más común de los Arreglos paralelos y entregar paralelos y entregar por escrito el resultado de tu investigación.

A.2.2. Diseñar el TDA Auto para los usuarios Cobranza, Servicio y Cliente con y Cliente con sus correspondientes Objetos y Atributos.

A.2.3. Investigar en el Lenguaje Pascal y en C++ cuál es la Estructura para Estructura para definir un Arreglo y un Registro.

A.2.4. Investigar cuáles son las Características de la Programación Orientada a Programación Orientada a Objetos (POO) y entregar por escrito el el resultado de tu investigación.

A.2.5. Investigar si el Lenguaje Visual Basic contiene todas las características características de POO y entregar por escrito el resultado de tu investigación.

A.2.6.Investigar qué es el Puntero y sus aplicaciones. Presentar por escrito el por escrito el resultado de tus investigaciones.

A.2.7. Consultar la siguiente página http://quegrande.org/apuntes/EI/2/Alg/teoria/07-08/tema_2.4_-_tablas_de_dispersion.pdf para ver más a detalle los conceptos del del archivo PDF y los programas ahí mostrados.

A.2.8. Investigar la diferencia entre Invariante (característica de los TDA) y los TDA) y Herencia (característica de la POO) y entregar el resultado de resultado de tu comparativo por escrito.

Cuestionario de Autoevaluación

0. Anota el concepto de Registro.
0. Anota el concepto de Arreglo.
0. Anota el concepto de vector.
0. Anota el concepto de TDA y qué significa su sigla.
0. Anota la Estructura de un TDA.
0. Anota el concepto de puntero y sus aplicaciones.
0. Anota las instrucciones para definir un Registro en los Lenguajes Pascal y Lenguajes Pascal y C++.
0. En el Lenguaje C++, ¿para qué sirve la instrucción Struct?
0. ¿Qué es el Atributo de un Objeto?
0. Explica las características de la POO.

Examen de Autoevaluación

I. Completa las oraciones

0. El ________ es un tipo de dato de iguales tipos de sus elementos. elementos.
0. El ________ sirve para apuntar a la dirección de una localidad de de memoria.

II. Indica si las siguientes oraciones son falsas o verdaderas

1.

puede cambiar el Tope o Cima de una Pila

Arreglos solo emplean un índice

tienen cualidades

son un refinamiento de los Registros

Arreglos están en memoria Externa

Registros están en Memoria Externa

Registros Anidados se definen subcampos.

TEMA 3. ESTRUCTURAS DE DATOS AVANZADAS

Objetivo Particular.

Al finalizar el alumno conocerá y comprenderá los Tipos de Datos Dinámicos, Datos Dinámicos, su importancia, las diferencias con respecto a las anteriores las anteriores estructuras y sus más importantes aplicaciones.

Temario Detallado

3.1. Listas simplemente enlazadas

- 3.1.1.-Definición del tipo de dato abstracto lista
- 3.1.2.-Definición de las operaciones sobre listas
- 3.1.3.-Implantación de una lista

3.2. Listas doblemente enlazadas

- 3.2.1.-Definición del tipo de dato abstracto lista doble
- 3.2.2.-Definición de las operaciones sobre lista doble
- 3.2.3.-Implantación de una lista doble

3.3. Árboles

- 3.3.1.-Definición del tipo de dato abstracto árbol binario
- 3.3.2.-Definición de las operaciones sobre árboles binarios
- 3.3.3.-Implantación de un árbol binario

3.4. Grafos

- 3.4.1.-Definición del tipo de dato abstracto grafo
- 3.4.2.-Operaciones sobre un grafo
- 3.4.3.-Implantación de un grafo

Introducción

Las Operaciones con Listas son comunes en la comprobación (mapeo) de la (mapeo) de la memoria Interna de la Computadora y la información se maneja información se maneja empleando estructuras más simples que las listas para las listas para poder procesar su formato y estructura en vectores lineales. vectores lineales.

3.1. Listas simplemente enlazadas

> Una lista lineal es un conjunto de elementos de un tipo dado que se que se encuentren ordenados (pueden variar en número). Los elementos elementos de una lista se almacenan normalmente de manera contigua contigua (un elemento detrás de otro) en posiciones de la memoria.[9] memoria.[9]

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

Dato→ (valor elemento)→ Enlace

2.0.0. Definición del tipo de dato abstracto lista

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

<e1, e2 ………, en>

[9] Luis Joyanes Aguilar. op. cit., p. 441.

5 | | 4 | | 1 | | 18 | | 1 | | 9 |

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

Representaciones de listas

Las **listas enlazadas** tienen una terminología propia que suelen utilizar utilizar normalmente. Los valores se almacenan en un nodo cuyos cuyos componentes se llaman campos. Un nodo tiene al menos un un campo de dato o valor y un enlace (indicador o puntero) con el el siguiente campo. El campo enlace apunta (proporciona la dirección dirección de) al siguiente nodo de la lista. El último nodo de la lista lista enlazada, por un convenio, suele representarse por un enlace con enlace con la palabra reservada *nil* (nulo), una barra inclinada (/)y, en (/)y, en ocasiones, el símbolo eléctrico de la tierra o masa.[10]

5

lista

Figura 3.1 Representación Gráfica de Lista Ligada

3.1.2 Definición de las operaciones sobre listas (especificación (especificación algebraica)

Las operaciones que pueden realizarse con listas lineales contiguas son: contiguas son:

[10] Héctor Fiestas Bancayán, “Estructuras dinámicas de datos”, en línea, disponible en: es.geocities.com/hwfiestasb/Catedras/EstructurasDatos/SESION_9_A_ESTRUCTURAS_DINAMICAS.pdf, recuperado el 18/10/08.

0. Insertar, eliminar o localizar un elemento.
0. Determinar el tamaño de la lista.
0. Recorrer la lista para localizar un determinado elemento.
0. Clasificar los elementos de la lista en orden ascendente o descendente.
0. Unir dos o más listas en una sola.
0. Dividir una lista en varias sublistas.
0. Copiar la lista.
0. Borrar la lista.

Operaciones que normalmente se ejecutan con las listas enlazadas:
enlazadas:

0. Recuperar información de un nodo especificado.
0. Encontrar un nodo nuevo que contenga información específica.
0. Insertar un nodo nuevo en un lugar específico de la lista.
0. Insertar un nuevo nodo en relación con una información particular.
particular.
0. Borrar un nodo existente que contenga información específica.

Operación para construir una lista vacía

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

El pseudocódigo correspondiente es:

←

```
Inicio
                LeerL (longitud de la lista)
                Si L = 0 Entonces, visualizar la lista
                   Si no, si 1 < = j < = L Entonces, B P (j)
                       Si no, visualizar elemento no
                       Fin-si
                 Fin-si
                Fin
                ListainicLista(void)
                / * Crea y retorna una lista vacía * /
                post:inicLista =<>
```

Operación para insertar un elemento a una lista

En una **Lista enlazada** se desea insertar un nodo N que debe ocupar un lugar ocupar un lugar después del elemento X, o bien entre dos nodos A y B. La nodos A y B. La inserción presenta dos casos particulares:

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

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

Algoritmo inserción

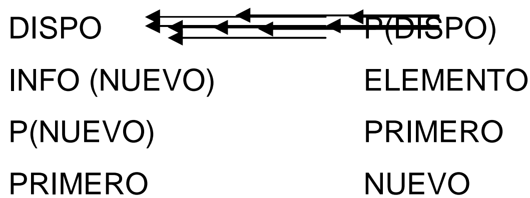
'Comprobación de sobrecarga'

Si DISPO=NULO

Entonces, escribir "sobrecarga" ←----- Impacto en la Memoria

Si no:

NUEVO DISPO



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

Fin_si

Inserción a continuación de un nodo específico.

El algoritmo para insertar un elemento (por ejemplo, NOMBRE) en una lista en una lista enlazada a continuación del nodo específico P pasa por obtener un 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

Operación para revisar si una lista es vacía o no

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

A

Adam

línea vacía <error>----------→

Figura 3.2 Lista con campos vacíos

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 seguirá recorriendo hasta encontrar algún elemento en la búsqueda. búsqueda.

El pseudocódigo correspondiente es:

Algoritmo búsqueda

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

     Si no: Q                  1
     Fin Si
```

...

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

Operación para obtener la cabeza de una lista

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

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

Algoritmo inserción

{Comprobación de sobrecarga }

Si DISPO = NULO **Entonces**, escribir “sobrecarga”

Si no:

NUEVO DISPO

DISPO P(DISPO)

INFO(NUEVO) ELEMENTO

P(NUEVO) PRIMERO

PRIMERO NUEVO

Fin_si

A

Figura 3.3 Tope y Sima de una Lista

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

NUEVO NODO

INFO (NUEVO) X

SIG(NUEVO) SIG(CABECERA)

SIG(CABECERA) NUEVO

Operación para eliminar la cabeza de una lista

El algoritmo que elimina de la lista enlazada el elemento siguiente apuntando siguiente apuntando por P utiliza un puntero auxiliar Q, y se establece primero establece primero para apuntar al elemento que se elimina:

1. Q SIG(P)
2. SIG(P) SIG(Q)

3. LIBERAR NODO (Q) -----→ libera espacio de memoria.
memoria.

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

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

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

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

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

DATOS (i) Elemento i de la lista.

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

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

Cálculo de la longitud de una lista

La longitud de una lista se define como el número de elementos que la

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

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

1 2i n length <cadena>

<cadena>

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

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

lista).

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*/
STRING*/
                   /* dependiendo del tipo del */
                   /* elemento correspondiente */
     unión
```

```
            int ival;

            float fval;

            char pval; /*apuntador a una cadena*/

               element;

            struct node *next;

};
```

Operación para invertir los elementos de una lista

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

lenguaje COBOL):

Habrá que tomar en consideración las diferentes versiones del Lenguaje Lenguaje COBOL, pues hay diferencias muy marcadas entre COBOL-ANSI, COBOL-ANSI, COBOL MICROSOFT,

COBOL 2000, etc.

```
            PROCEDURE DIVISION.

……………………………………………………..

--------------------------------------------------------------

        01 APUNTADORES AUXILIARES.

            02 ESTE-NODO PIC 9(3)

            02 NODO-ANTERIOR PIC9 (3)

            02 NODO-SIGUIENTE PIC9(3)

        INVERSIÓN-DE-LA LISTA

            IF PRIMER-NODO NOT=0

            THEN COMPUTE ESTE-NODO =PRIMER NODO

            COMPUTE NODO-SIGUIENTE = SIG-NODO (ESTE-NODO)
```



```
(ESTE-NODO)

            COMPUTE SIG-NODO (ESTE-NODO) = 0
```

BUSCA NODO

COMPUTE NODO-ANTERIOR = ESTE NODO

COMPUTE ESTE-NODO = NODO SIGUIENTE

COMPUTE NODO SIGUIENTE <0 SIG-NODO (ESTE NODO)

NODO)

COMPUTE SIG-NODO (ESTE-NODO) = NODO-ANTERIOR

ANTERIOR

……………………………………………………………………

……………………………………………………………………

STOP RUN.

……………………………………………………………………

END.

Funciones de orden superior

Otras operaciones interesantes pueden enriquecerse con operaciones de operaciones de manejo de persistencia y destrucción, de acuerdo con los acuerdo con los siguientes:

TAD Lista tipL

Void destruir Lista (Lista lst)

/* destruye el objeto abstracto, retornando toda la memoria ocupada por éste * /

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 * /

{ { ( ) } } }

post: se ha construido la lista que corresponde a la imagen de la información 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 de la lista está indefinida.

Además, se puede traer de la memoria secundaria una lista; modificar su modificar su contenido eliminando todas las ocurrencias de un valor dado y valor dado y hacer persistir de nuevo la lista resultante. La complejidad es 0 (n), complejidad es 0 (n), donde n es la longitud de la lista. Se debe tener en cuenta tener en cuenta que la constante asociada con esta función es muy alta, dado muy alta, dado el elevado costo en tiempo que tiene el acceso a la información la información en memoria secundaria.

```
………………
**definición necesaria de los parámetros de funcion**
funcion**
//* prototipo de funcion
Void actualizar Lista (char nombre     , tipo L val)
val)
FILE *fp = fopen (nombre, "r")
Lista 1st = cargarLista (fp)
Fclose (fp) ;
ElimtodosLista (2st, val);
Fp = forpen (nombre, "w");
SalvarLista (1st, fp);
```

```
Fclose (fp);
DestruirLista (1st);
```

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

3.1.3 Implantación de una Lista

Implantación de las operaciones sobre listas

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

Implantación dinámica (mediante el uso de apuntadores)

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

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

La forma lógica de satisfacer la condición 1 es tener un arreglo global de arreglo global de registros. Para cualquier celda de arreglo, puede usarse su 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: arrary (0...TAMAÑO_ESPACIO) of nodo;
of nodo;
```

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

Listas generalizadas

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

Definición del tipo de dato abstracto lista generalizada

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

Definición de las operaciones sobre listas generalizadas

Definiremos ahora una serie de operaciones abstractas cuyas propiedades propiedades lógicas son las operaciones *head* y *tail*, que no están definidas si están definidas si su argumento no es una lista. Una sublista de una lista R es una lista R es una lista que resulta de aplicación de cero a más operaciones *tail* operaciones *tail* a 1.

Operación para construir una lista generalizada

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

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

Operación para obtener la cabeza de una lista generalizada

Para hacer este proceso, el algoritmo se vale de la operación *push* para

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
deberá insertarse if (q = = null) /* insertar x a la cabeza
a la cabeza de la lista.
push (list, x);
else
insafter (q, x)
```

Top de la Lista

Figura 3.4 Representación de una Lista Circular

Operación para verificar si la cabeza de una lista generalizada es un generalizada es un elemento de la misma.

Un **nodo atómico** no contiene apuntadores, sólo un elemento de datos simple. de datos simple. Existirían varios tipos diferentes de nodos atómicos, si cada atómicos, si cada uno de los elementos únicos de datos correspondiera a uno correspondiera a uno de los tipos de datos legales.

Un nodo de lista comprende un apuntador a un nodo atómico y un indicador de un indicador de tipo que señala la clase de nodo atómico a la que apunta (así que apunta (así como un apuntador al siguiente nodo de la lista, por supuesto). por supuesto). Cuando es necesario colocar un nodo nuevo en una lista, debe una lista, debe asignarse un nodo atómico del tipo apropiado y de su valor, y de su valor, y deben establecerse el campo de la información del nodo de lista del nodo de lista para que apunte al nuevo nodo atómico y el tipo de campo tipo de campo correcto.

```
          switch (typecode) {
          case t1;            //hacer algo con node1
node1
          case t2;       //hacer algo con node2
                         ...
          caset10;       //hacer algo con node10

          }              //fin de swiich
```

Definición de la semántica de las operaciones sobre listas generalizadas generalizadas

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

```
     struct nodetype {
          int utype;              /*utype   es   igual   a
igual a INTGR, CH, O LST*/
          unión  {
          int intgrinfo;          /*utype = INTGR*/
INTGR*/
          char charinfo;     /* utype = CH */
```

```
        struct nodetype *lstinfo;      /* utype = LIST */
= LIST */ }  info;
        struct nodetype *next
    } ;
    typedef struct nodetype *NODEPTR;
```

Implantación dinámica de las operaciones sobre listas con prioridades prioridades

Lenguaje C

```
        Int getnode (void)

        Int p;
            If (avail ==-1)
                printf ("overflow\n");
                exit (1);

            p=avail;
            avail =node (avail).next:
            return (p);
    [           /*fin de getnode*/
```

3.2 Listas doblemente enlazadas

3.2.1 Definición del Tipo de Dato Abstracto Lista Doble

En algunas aplicaciones podemos desear recorrer la lista hacia adelante y adelante y hacia atrás, o dado un elemento, podemos desear conocer conocer rápidamente los elementos anterior y posterior. En tales situaciones situaciones podríamos desear darle a cada celda sobre una lista un puntero a un puntero a las celdas siguiente y anterior en la lista tal y como se muestra en se muestra en la figura.

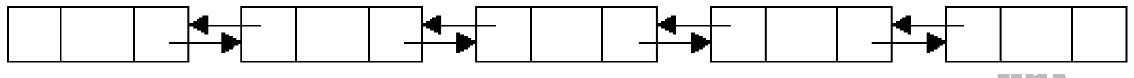


Figura 3.5 Representación Grafica de Lista Doble

Otra ventaja de las listas doblemente enlazadas es que podemos usar un podemos usar un puntero a la celda que contiene el i-ésimo elemento de una elemento de una lista para representar la posición i, mejor que usar el puntero usar el puntero a la celda anterior aunque lógicamente, también es posible la es posible la implementación similar a la expuesta en las listas simples simples haciendo uso de la cabecera. El único precio que pagamos por estas pagamos por estas características es la presencia de un puntero adicional en adicional en cada celda y consecuentemente procedimientos algo más largos algo más largos para algunas de las operaciones básicas de listas. Si usamos listas. Si usamos punteros (mejor que cursores) podemos declarar celdas que declarar celdas que consisten en un elemento y dos punteros a través de:

través de:

```
typedef struct celda{
      tipoelemento elemento;
      struct celda *siguiente,*anterior;
}tipocelda;

typedef tipocelda *posicion;
```

Un procedimiento para borrar un elemento en la posición p en una lista una lista doblemente enlazada es:

```
void borrar (posicion p)
{
   if (p->anterior != NULL)
      p->anterior->siguiente = p->siguiente;
   if (p->siguiente != NULL)
      p->siguiente->anterior = p->anterior;
   free(p);
}
```


El procedimiento anterior se expresa de forma gráfica en como muestra la muestra la figura:

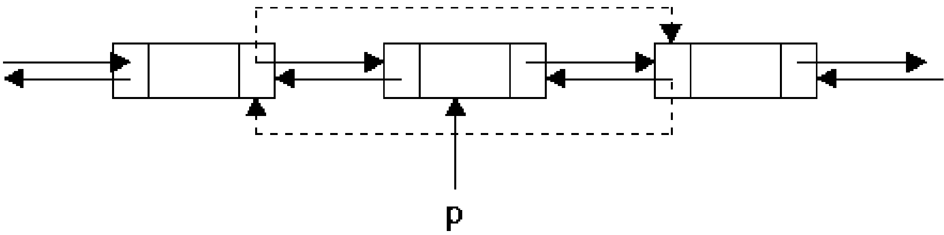


Figura 3.6 Borrado de un Elemento de la Lista

Donde los trazos continuos denotan la situación inicial y los punteados la final. punteados la final. El ejemplo visto se ajusta a la supresión de un elemento o un elemento o celda de una lista situada en medio de la misma. Para obviar los Para obviar los problemas derivados de los elementos extremos (primero y (primero y último) es práctica común hacer que la cabecera de la lista la lista doblemente enlazada sea una celda que efectivamente complete el complete el círculo, es decir, el anterior a la celda de cabecera sea la última sea la última celda de la lista y la siguiente la primera. De esta manera no manera no necesitamos verificar para NULL en el anterior procedimiento procedimiento borrar.

Por consiguiente, podemos realizar una implementación de listas doblemente listas doblemente enlazadas con cabecera tal que tenga una estructura circular estructura circular en el sentido de que dado un nodo y por medio de los medio de los punteros *siguiente* podemos volver hasta él como se puede se puede observar en la figura (de forma análoga para *anterior*).

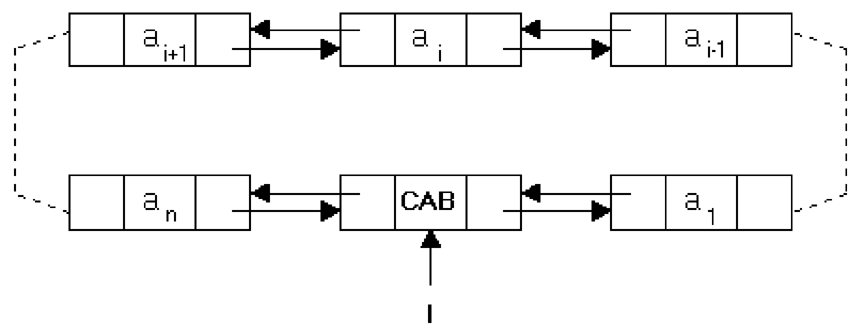


Figura 3.7 Lista Circular

Es importante notar que aunque la estructura física de la lista puede hacer puede hacer pensar que mediante la operación *siguiente* podemos alcanzar de podemos alcanzar de nuevo un nodo de la lista, la estructura lógica es la de lógica es la de una lista y por lo tanto habrá una posición *primero* y una *primero* y una posición *fin* de forma que al aplicar una operación *anterior* o *anterior* o *siguiente* respectivamente sobre estas posiciones el resultado será resultado será un error.

Respecto a la forma en que trabajarán las funciones de la implementación que implementación que proponemos hay que hacer constar los siguientes puntos: siguientes puntos:

- La función de creación debe alojar memoria para la cabecera y hacer hacer que los punteros *siguiente* y *anterior* apunten a ella, devolviendo devolviendo un puntero a dicha cabecera.

- La función *primero(l)* devolverá un puntero al nodo siguiente a la cabecera.

- La función *fin(l)* devolverá un puntero al nodo cabecera.

- Trabajar con varias posiciones simultáneamente tendrá un comportamiento idéntico al de las listas enlazadas excepto respecto al respecto al problema referente al borrado cuando se utilizan posiciones posiciones consecutivas. Es posible implementar la función de borrado borrado de tal forma que borrar un elemento de una posición p invalida invalida el valor de dicha posición p y no afecta a ninguna otra posición. posición. Nosotros en nuestra implementación final optaremos por pasar por pasar un puntero a la posición para el borrado de forma que la la posición usada quede apuntando al elemento siguiente que se va a se va a borrar al igual que ocurría en el caso de las listas simples. Otra simples. Otra posible solución puede ser que la función devuelva la devuelva la posición del elemento siguiente a ser borrado.

- La inserción se debe hacer a la izquierda del nodo apuntado por la la posición ofrecida a la función insertar. Esto implica que al contrario contrario que en las listas simples, al insertar un nodo, el puntero puntero utilizado sigue apuntando al mismo elemento al que apuntaba y

apuntaba y no al nuevo elemento insertado. Si se desea, es posible posible modificar la función de forma que se pase un puntero a la la posición de inserción para poder modificarla y hacer que apunte al apunte al nuevo elemento insertado. En cualquier caso, el comportamiento final de la función deberá quedar reflejado en el conjunto de especificaciones del TDA.

3.2.2. Definición de operaciones sobre Lista Doble

Dentro del tipo abstracto de *listas doblemente enlazadas* podemos proponer las siguientes primitivas:

```
tLista crear ()
void destruir (tLista l)
tPosicion primero (tLista l)
tPosicion fin (tLista l)
void insertar (tElemento x, tPosicion p, tLista
l)
void borrar (tPosicion *p, tLista l)
tElemento elemento(tPosicion p, tLista l)
tPosicion siguiente (tPosicion p, tLista l)
tPosicion anterior (tPosicion p, tLista l)
tPosicion posicion (tElemento x, tLista l)
```

Especificación semántica y sintáctica

tLista crear ()

Argumentos: Ninguno.

Efecto: (Constructor primitivo). Crea un objeto del tipo tLista.

void destruir (tLista l)

Argumentos: Una lista.

Efecto: Destruye el objeto l liberando los recursos que empleaba. Para volver a usarlo habrá que crearlo de nuevo.

tPosicion primero (tLista l)

Argumentos: Una lista.

Efecto: Devuelve la posición del primer elemento de la lista.

tPosicion fin (tLista l)

Argumentos: Una lista.

Efecto: Devuelve la posición posterior al último elemento de la lista.

lista.

- **void insertar (tElemento x, tPosicion p, tLista l)**

  Argumentos:

  l: Es modificada.

  p: Es una posición válida para la lista l.

  x: Dirección válida de un elemento del tipo T con que se instancia la lista, distinta de NULL.

  Efecto: Inserta elemento x en la posición p de la lista l desplazando desplazando todos los demás elementos en una posición.

- **void borrar (tPosicion *p, tLista l)**

  Argumentos:

  l: Es modificada.

  p: Es una posición válida para la lista l.

  Efecto: Elimina el elemento de la posición p de la lista l desplazando desplazando todos los demás elementos un una posición.

- **tElemento elemento(tPosicion p, tLista l)**

  Argumentos:

  l: Una lista.

  p: Es una posción válida de la lista l.

  Efecto: Devuelve el elemento que se encuentra en la posición p de la de la lista l.

- **tPosicion siguiente (tPosicion p, tLista l)**

  Argumentos:

  l: Una lista.

  p: Es una posición válida para la lista l, distinta de fin(l).

  Efecto: Devuelve la posición siguiente a p en l.

- **tPosicion anterior (tPosicion p, tLista l)**

  Argumentos:

  l: Una lista.

  p: Es una posición válida para la lista l, distinta de

  Efecto: Devuelve la posición que precede a p en l.

- **tPosicion posicion (tElemento x, tLista l)**

  Argumentos:

  l: Una lista.

  x: Dirección válida de un elemento del tipo T con que se instancia la lista, distinta de NULL.

  Efecto: Si x se encuentra entre los elementos de la lista l, devuelve la devuelve la posición de su primera ocurrencia. En otro caso, devuelve la devuelve la posición fin(l).

EFICIENCIA

Comparación de la eficiencia para las distintas implementaciones de las listas:

implementaciones de las listas:

Operaciones	Matricial	Enlace Simple	Enlace Doble
Crear	O(1)	O(1)	O(1)
Destruir	O(1)	O(n)	O(n)
Primero	O(1)	O(1)	O(1)
Fin	O(1)	O(n)/O(1)	O(1)
Insertar	O(n)	O(1)	O(1)
Borrar	O(1)	O(n)	O(n)
Elemento	O(1)	O(1)	O(1)
Siguiente	O(1)	O(1)	O(1)
Anterior	O(1)	O(n)	O(1)
Posicion	O(n)	O(n)	O(n)
• Memoria	• Tamaño fijo	• Sobrecarga de un puntero por nodo. • Un nodo sin información.	• Sobrecarga de dos punteros por nodo. • Un nodo sin información.

Cuadro 3.1 Comparativo entre las Operaciones para Listas con tres tipos con tres tipos de Enlaces

3.2.3 Implantación de una Lista Doble

Una vez aclaradas las posibles ambigüedades y dudas que se puedan puedan plantear, la implementación de las listas doblemente enlazadas enlazadas quedaría como sigue:

```
typedef struct celda {
   tElemento elemento;
   struct celda *siguiente,*anterior;
```

```
} tipocelda;

typedef tipocelda *tPosicion;
typedef tipocelda *tLista;
static void error(char *cad)
{
  fprintf(stderr, "ERROR: %s\n", cad);
   exit(1);
}

tLista Crear()
{
   tLista l;


   l = (tLista)malloc(sizeof(tipocelda));
   if (l == NULL)
      Error("Memoria insuficiente.");
   l->siguiente = l->anterior = l;
   return l;
}

void Destruir (tLista l)
{
   tPosicion p;


   for (p=l, l->anterior->siguiente=NULL; l!=NULL; p=l) {
l!=NULL; p=l) {
      l = l->siguiente;
      free(p);
   }
}
```

tPosicion Primero (tLista l)

```
{

   return l->siguiente;
}
```

tPosicion Fin (tLista l)

```
{

   return l;
}
```

void Insertar (tElemento x, tPosicion p, tLista l)

```
{
   tPosicion nuevo;


   nuevo = (tPosicion)malloc(sizeof(tipocelda));
(tPosicion)malloc(sizeof(tipocelda));
   if (nuevo == NULL)
      Error("Memoria insuficiente.");
   nuevo->elemento = x;
   nuevo->siguiente = p;
   nuevo->anterior = p->anterior;
   p->anterior->siguiente = nuevo;
   p->anterior = nuevo;
}
```

void Borrar (tPosicion *p, tLista l)

```
{
   tPosicion q;

   if (*p == l){
      Error("Posicion fin(l)");
```

```
    }
    q = (*p)->siguiente;
    (*p)->anterior->siguiente = q;
    q->anterior = (*p)->anterior;
    free(*p);
    (*p) = q;
}
```

tElemento elemento(tPosicion p, tLista l)

```
{

    if (p == l){
        Error("Posicion fin(l)");
    }
    return p->elemento;
}
```

tPosicion siguiente (tPosicion p, tLista l)

```
{

    if (p == l){
        Error("Posicion fin(l)");
    }
    return p->siguiente;
}
```

tPosicion anterior(tPosicion p, tLista l)

```
{

    if (p == l->siguiente){
        Error("Posicion primero(l)");
    }
    return p->anterior;
}
```

tPosicion posicion (tElemento x, tLista l)

```
{
   tPosicion p;
   int encontrado;
   p = primero(l);
   encontrado = 0;
   while ((p != fin(l)) && (!encontrado))
      if (p->elemento == x)
            encontrado = 1;
      else
            p = p->siguiente;
   return p;
}
```

3.3. Árboles

Definiciones:

Es un conjunto no vació de vértices (nodos) y aristas (enlaces) que que cumple una serie de requisitos.[11]

Son Estructuras de Datos no lineales. Es una colección de nodos donde nodos donde cada uno, además de almacenar información, guarda la guarda la dirección de sus sucesores.[12]

Los Árboles, a diferencia de las Listas, **sirven** para representar Estructuras Jerárquicas, hecho que no se puede con las Listas Lineales y Lineales y mucho menos con los Arreglos. Las Pilas y Colas aunque aunque representan cierta jerarquía, están limitadas a emplear una sola una sola dimensión.[13]

Un Árbol se **puede representar** como Conjuntos Venn, Anidación de

---

[11] Robert Sedgewick, *Algoritmos en c++*. México, Pearson Education 2000, p 40.
[12] Silvia Guardati, *Estructura de datos orientada a objetos*, México, Pearson Educación 2007, p.. 313.
[13]Adam Drozdek, *Estructura de datos y algoritmos en Java,* México Thomson, 2ª dic.2007, p. 214.

Anidación de Paréntesis, Grafos y como una Estructura Jerárquica.
Jerárquica.

Elementos del Árbol

- Nodo Raíz, Nodos Hijos, Nodos Hermanos, Altura, Recorridos, Dirección.
- Todo Árbol tiene un solo Nodo Raíz.
- Los Árboles pueden tener o no Nodos Hijos, si sí los tiene, puede puede haber Nodos Hermanos.
- Si únicamente tiene un Nodo Raíz su Altura = 0 y su Nivel = 1.
- Los Recorridos pueden ser en preorden, postorden y en orden.
- Un Árbol puede recorrerse en dirección Top-Down De arriba abajo, abajo, De abajo arriba, (Down-Top), A partir de su rama Izquierda o Izquierda o a Partir de su rama Derecha.

Para mayor claridad se representa la siguiente figura.

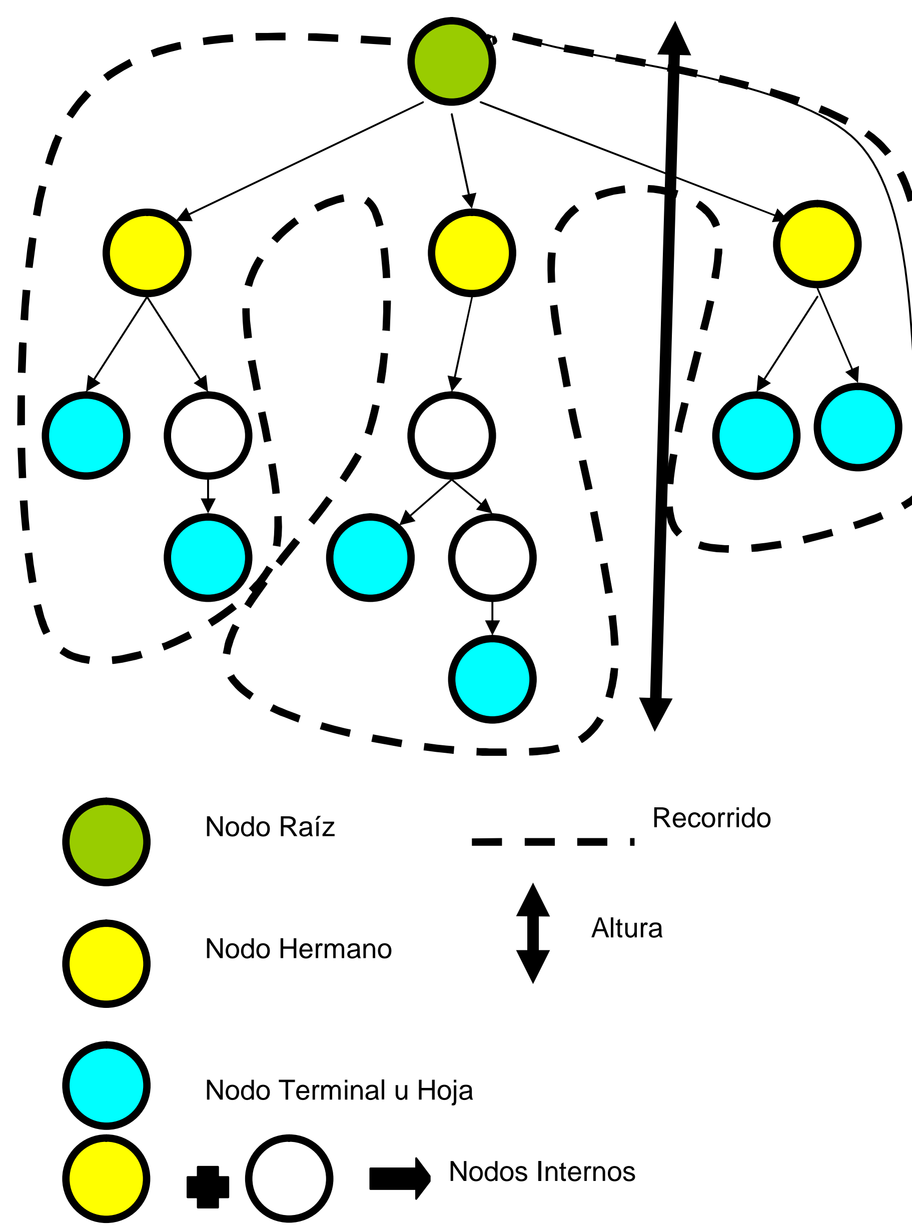


Figura 3.8 Elementos de un Árbol

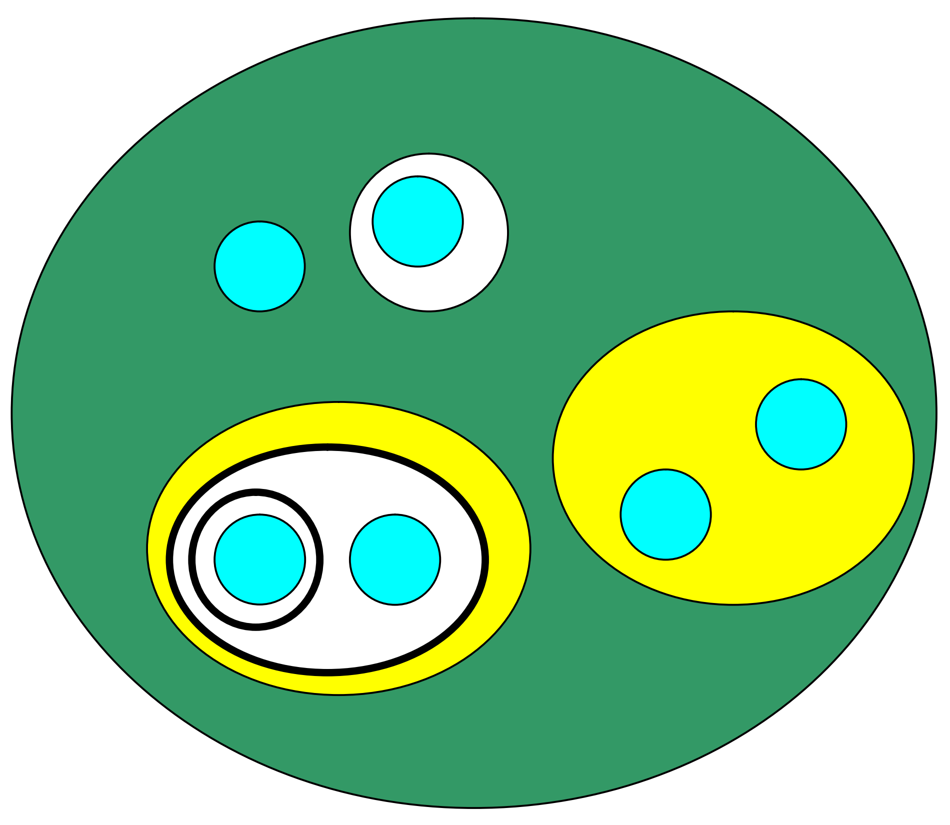


Figura 3.9 Diagrama de Venn del Árbol de la Figura 3.9

Glosario Fundamental de Árboles

Árbol.- Es un conjunto no vacío de vértices y aristas que cumple una serie de una serie de requisitos. (Sedgewick, p. 40).

Altura del Árbol.- Es el máximo de los niveles considerando todos los nodos. todos los nodos. (Drozdek, p. 214 y (Sedgewick, p. 41).

Altura de un Árbol.- Es el nivel máximo del Árbol (o la máxima distancia entre distancia entre la raíz y cualquier nodo). (Guardati, p. 315).

Árbol Multicamino.- Cuando cada nodo debe tener un número específico de específico de hijos colocados en un orden determinado. (Sedgewick, p. 41). (Sedgewick, p. 41).

Árbol Binario.- Es un conjunto finito de elementos que está vacío o dividido vacío o dividido en tres subconjuntos separados. (Yedidyah, ET AL, 249). AL, 249).

Árbol Binario Lleno.- Es aquel en que los nodos internos llenan todos los todos los niveles, con la posible excepción del último. (Sedgewick, p. 42). (Sedgewick, p. 42).

Árbol Binario Completo.- Es un Árbol Binario Lleno en el que los nodos nodos internos del último nivel aparecen todos a la izquierda de los nodos los nodos externos de ese mismo nivel. (Drozdek, 216 y Sedgewick, p. 42). Sedgewick, p. 42).

Árbol Ordenado.- Es aquel en el que se ha especificado el orden de los hijos orden de los hijos de todos los nodos. (Sedgewick, p. 41).

Arista.- Conexión entre dos nodos.

Arco.- Relación entre nodos

Bosque.- Conjunto de Árboles. (Sedgewick, p. 41).

Camino de un Árbol.- Es una lista de nodos distintos en que dos consecutivos se enlazan mediante arista, sin embargo, existe un camino camino entre la raíz y cada uno de los otros nodos del árbol de acuerdo a la acuerdo a la Dirección y Recorrido previamente establecidos. Si no existe tal no existe tal camino entonces se estará ante una Estructura de tipo Grafo. tipo Grafo. (Sedgewick, p. 40).

Grado del Árbol.- Es el máximo de los grados, considerando todos sus todos sus nodos. (Guardati, p. 315).

Grado de un Nodo.- Es el número de hijos que dicho nodo tiene. (Guardati, (Guardati, p. 315).

Longitud del Camino.- Es la suma de los niveles de todos los nodos del árbol nodos del árbol (la suma de las longitudes de los caminos desde cada nodo desde cada nodo a la raíz. (Sedgewick, p. 41).

Longitud del Camino Interno.- Es la suma de los niveles de los nodos nodos Internos del Árbol. (Drozdek, p. 222 y Sedgewick, p. 41).

Longitud del Camino Externo.- Es la suma de los niveles de los nodos nodos Externos del Árbol. (Sedgewick, p. 41).

Nivel de un Nodo.- Es el número de nodos del camino que lleva desde este desde este hasta la raíz (sin incluirse el mismo). (Guardati, p. 314)
314)

Nivel de un Nodo.- Es el número de arcos que deben ser recorridos, recorridos, partiendo de la raíz, para regresar a dicha raíz. (Sedgewick, p. (Sedgewick, p. 41).

Nodo.- Cada elemento del Árbol (Yedidyah, ET AL, p. 249).

Nodo Interno.- Aquellos que no son Nodos Terminales ni la Raíz. (Guardati, Raíz. (Guardati, p. 315).

Nodo Terminal.- Aquel que no tiene nodos Hijos. (Guardati, p. 315).
315).

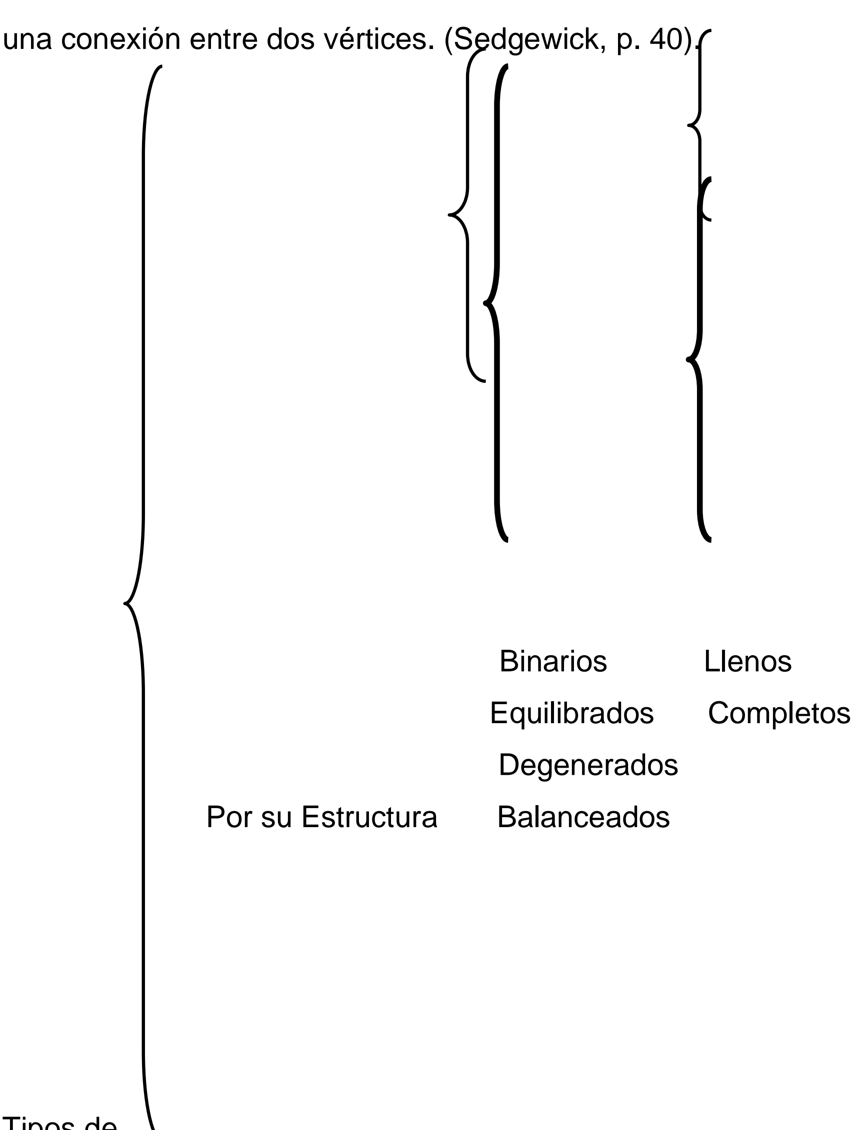
Recorrido del Árbol.- Es el proceso de visitar cada nodo en el árbol árbol exactamente una vez. (Drozdek, p. 223).

Recorrido en Amplitud.- Es visitar cada nodo empezando por el nivel inferior nivel inferior (0 superior) y moverse hacia abajo (o hacia arriba) nivel por nivel por nivel, visitando nodos en cada nivel de izquierda a derecha, o derecha, o viceversa.(Drozdek, p. 224)

Recorrido en Profundidad.- Continua lo mas lejos posible a la izquierda (o a izquierda (o a la derecha), luego regresa hacia arriba hasta el primer cruce, primer cruce, avanza un paso a la derecha (o a la izquierda) y de nuevo lo de nuevo lo más lejos posible a la izquierda (o a la derecha). (Drozder, p. (Drozder, p. 225).

Vértice.- Es un objeto simple (nodo) que puede tener un nombre y puede nombre y puede llevar otra información asociada: una arista es una conexión

una conexión entre dos vértices. (Sedgewick, p. 40).



Binarios Llenos

Equilibrados Completos

Degenerados

Por su Estructura Balanceados

Tipos de

Árboles

Binarios

Binarios de Búsqueda

Por su Recorrido B

B+

Multicamino B*

B+ Prefijos

De Bits.

R

2-4

Cuadro 3.2 Clasificación de los Árboles[14]

3.3.1 Definición del tipo de dato abstracto árbol binario

Un árbol binario *T* se define como un conjunto finito de elementos, elementos, llamados nodos, de forma que:

) *T* es vacío (en cuyo caso se llama árbol nulo o árbol vació) o
) *T* contiene un nodo distinguido R, llamado raíz de T, y los restantes restantes nodos de T forman un par ordenado de árboles binarios binarios disjuntos T1 y T2.

Si T contiene una raíz R, los dos árboles T1 y T2 se llaman, respectivamente, subárboles izquierdo y derecho de la raíz R. Si T1 no es T1 no es vació, entonces su raíz se llama sucesor izquierdo de R; y R; y análogamente, si T2 no es vació, su raíz se llama sucesor derecho derecho de R.

Observe que:
) B es un sucesor izquierdo y C un sucesor derecho del nodo A.
) El subárbol izquierdo de la raíz A consiste en los nodos B, D, E y F, y F, y el subárbol derecho de A consiste en los nodos C, G, H, J, K y L. K y L.

[14] Confróntese, Adam Drozdek *Estructura de Datos y Algoritmos en Java*, pp. 214 - 231; Luis Joyanes Aguilar. *Fundamentos de Programación* y Robert Sedgewick *Algoritmos en C++*, pp. 39-50 y L. Yedidyah, *Estructuras de Datos de Datos con C y C++*, pp. 249- 319.

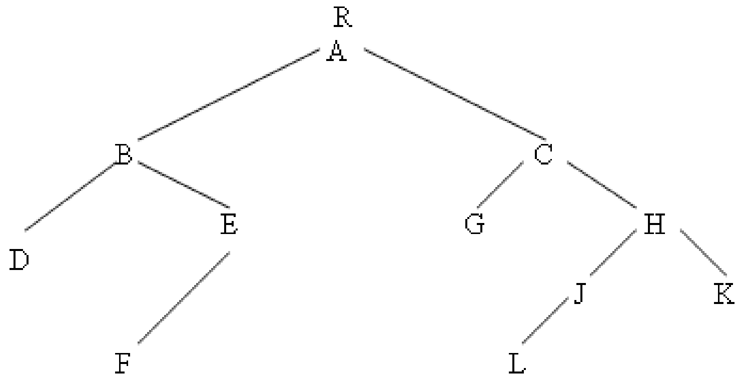


Figura 3.10 Representación Gráfica de Árbol Binario

Cualquier nodo N de un árbol binario T tiene 0, 1 ó 2 sucesores. Los Los nodos A, B, C y H tienen dos sucesores, los nodos R y J sólo tienen sólo tienen un sucesor, y los nodos D, F, G, L y K no tienen sucesores. sucesores. Los nodos sin sucesores se llaman nodos terminales. terminales.

La definición anterior del árbol binario T es recursiva, ya que T se define que T se define en términos de los subárboles binarios T1 y T2. Esto Esto significa, en particular, que cada nodo N de T contiene un subárbol subárbol izquierdo y uno derecho. Más aun, si N es un nodo terminal, terminal, ambos árboles están vacíos.

Dos árboles binarios T y T' se dicen que son similares si tienen la misma la misma estructura o, en otras palabras, si tienen la misma forma. Los forma. Los árboles se dice que son copias si son similares y tienen los tienen los mismos contenidos en sus correspondientes nodos.[15]

3.3.2 Definición de las operaciones sobre árboles binarios

árboles binarios

La Operaciones de acuerdo con su empleo generaran Estructuras de Árboles Estructuras de Árboles de Búsqueda, Binarios, B+, equilibrados y degenerados y degenerados (con una sola rama, una subrama y un nodo terminal) y de aquí terminal) y de aquí comienza su recorrido a partir desde del Nodo Raíz para ir

[15] Leandro Siso, "Área Informática, Estructuras de Datos". Monografías, http://www.monografias.com/trabajos36/arboles/arboles2.shtml, octubre 18 del 2008

Nodo Raíz para ir por las ramas, nodos internos hasta llegar a los nodos los nodos Terminales.

3.3.3 Implantación de un árbol binario

Suponga que T es un árbol general. A menos que se diga lo contrario, T contrario, T se mantendrá en memoria en términos de una representación representación enlazada que usa tres arrays paralelos, INFO, HIJO, (o HIJO, (o ABAJO) Y HERM (u HORIZ), y una variable puntero RAIZ, tal RAIZ, tal como sigue. En primer lugar, cada nodo N de T corresponderá a corresponderá a una posición K tal que:

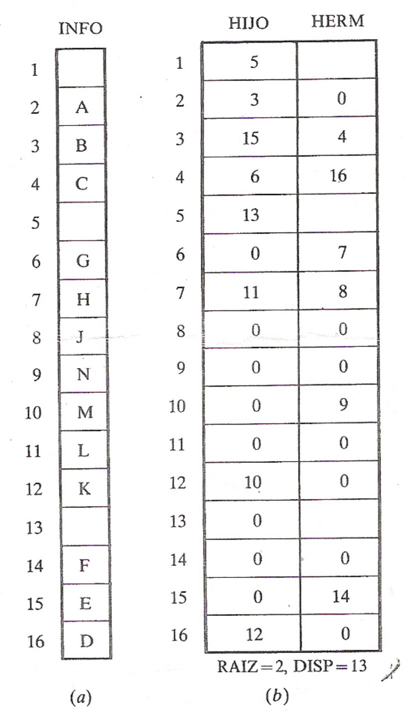
0) INFO [K] contiene los datos del nodo N.
0) HIJO [K] contiene la posición del primer hijo de N. La condición condición HIJO [K]= NULO indica que N no tiene hijos.
0) HERM [K] contiene la posición del siguiente hermano de N. La condición HERM [K] = NULO indica que N es el ultimo hijo de su su padre.

Ejemplo:

Considere el árbol general T de la figura (2) , suponga que los datos datos de los nodos de T se guardan en un Array INFO como en la la figura [3.11] las relaciones estructurales de T se obtienen asignando valores al puntero RAIZ y a los Arrays HIJO y HERM tal y tal y como sigue:

) como la raíz A de T se guarda en INFO [2], se y hace RAIZ:= 2. 2.
) Como el primer hijo de A es el nodo B, guardado en INFO [3], se se hace HIJO [2]:= 3. como A no tiene hermanos, se hace HERM [HERM [2]= NULO.
) Como el primer hijo de B es el nodo E, guardado en INFO [15], se se hace HIJO [3]:= 15. como el nodo C es el siguiente hermano de B de B y C se guarda en INFO [4], se hace HERM [3]:=4.

Y así sucesivamente. La figura [3.11] da los valores finales de HIJO y HIJO y HERM. Observe que la lista DISP de nodos vacos se mantiene en mantiene en el primer array, hijo, donde DISP = 1.



	INFO	HIJO	HERM
1		5	
2	A	3	0
3	B	15	4
4	C	6	16
5		13	
6	G	0	7
7	H	11	8
8	J	0	0
9	N	0	0
10	M	0	9
11	L	0	0
12	K	10	0
13		0	
14	F	0	0
15	E	0	14
16	D	12	0

RAIZ=2, DISP=13

(a) (b)

Figura 3.11 Árboles representados como Arreglos[16]

3.4. Grafos

Definiciones.-

"Un Grafo Simple G = (V,E) consiste en un conjunto de V de vértices y un vértices y un conjunto posiblemente vacío E de aristas (edges), siendo cada siendo cada arista un conjunto de dos vértices de V".[17]

"Son Estructuras de Datos no lineales, en las cuales cada elemento puede elemento puede tener cero o más sucesores y cero o más predecesores. Están

---

[16] Leandro Siso, Área Informática, "Estructuras de Datos", Monografías, http://www.monografias.com/trabajos36/arboles/arboles.shtml

[17] Adam Drozdek, op. cit., p. 376.

predecesores. Están formados por nodos (vértices: representan información) y información) y por arcos (aristas: relaciones entre la información)”.[18] información)”.[18]

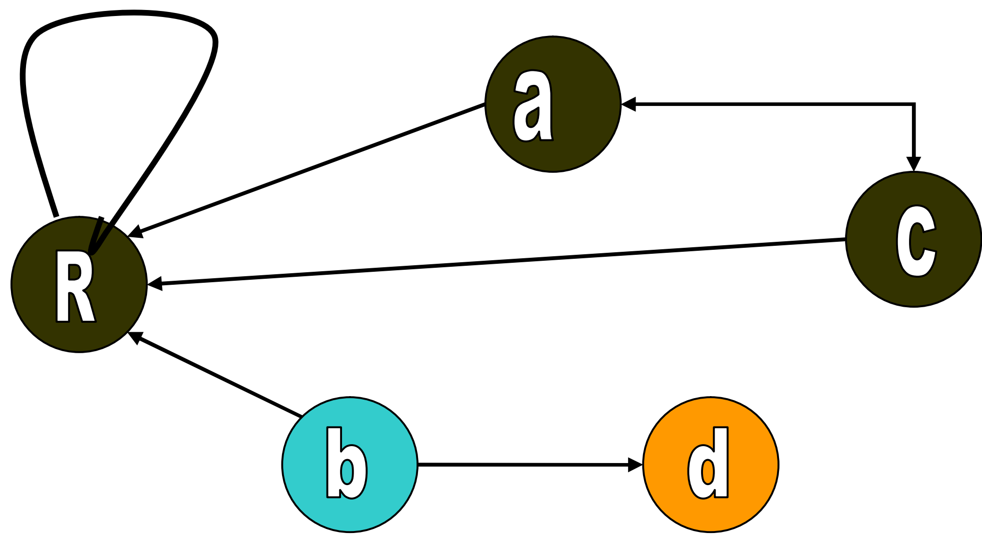


Figura 3.12 Representación de un Grafo

Glosario Fundamental[19]

Árbol de Expansión de un Grafo.- Es un Subgrafo que contiene todos los todos los vértices, pero solamente las aristas necesarias para formar un Árbol.

Bosque.- Un Grupo de Árboles sin conectar.

Camino.- Un camino entre los vértices *y* y *x* de un grafo es una lista de lista de vértices en la que dos elementos sucesivos están conectados por aristas del grafo.

Camino Cerrado.- Esto ocurre si el vértice origen es igual al vértice vértice destino.

Camino Simple.- Si todos sus vértices, con excepción del origen y destino, y destino, son distintos. El Primero y el último pueden ser

[18] Silvia Guardati, op. cit., p. 391.
[19] Véase, R. Sedgewick, op. cit., pp. 452-4; S. Guardati, op. cit., pp. 395-6.; L. Yedidyah op. cit. pp. 515- 541 y A. Drozdek, op. cit., pp. 337- 383.

iguales.

Camino Simple.- Es un camino en el que no se repite ningún vértice vértice

Camino del Vértice.- Esta formado por todas las aristas que deben deben recorrerse para llegar del origen al destino. Si se recorren *n* aristas, se dice que el camino es de longitud *n*.

Ciclo.- Es un camino simple con la característica de que el primero y el primero y el ultimo vértices son el mismo.

Ciclo.- Es un camino simple cerrado de longitud mayor o igual a tres. tres.

Grafo Completo.- Es aquel que contiene todas las aristas posibles. Si posibles. Si cada uno de sus vértices es adyacente a todos los vértices del grafo.

Grafo Conexo.- Si hay un camino desde cada nodo hacia otro nodo del nodo del grafo. Existe un camino simple entre cualesquiera de sus nodos.

Grafo No Conexo.- Esta constituido por componentes conexos.

Grafos Densos.- Son aquellos que les faltan muy pocas aristas de todas de todas las posibles.

Grafo Dirigido o grafo.- Es aquel cuyas aristas siguen cierta dirección y se dirección y se representa como $v_1 => v_2$.

Grafos Dispersos.- Son los que tienen relativamente pocas aristas (menos aristas (menos de *V log V*).

Grafo Etiquetado.- Es aquel si sus aristas tienen asociado un valor. Si este valor. Si este es un número no negativo, se le conoce con el nombre de peso, distancia o longitud.

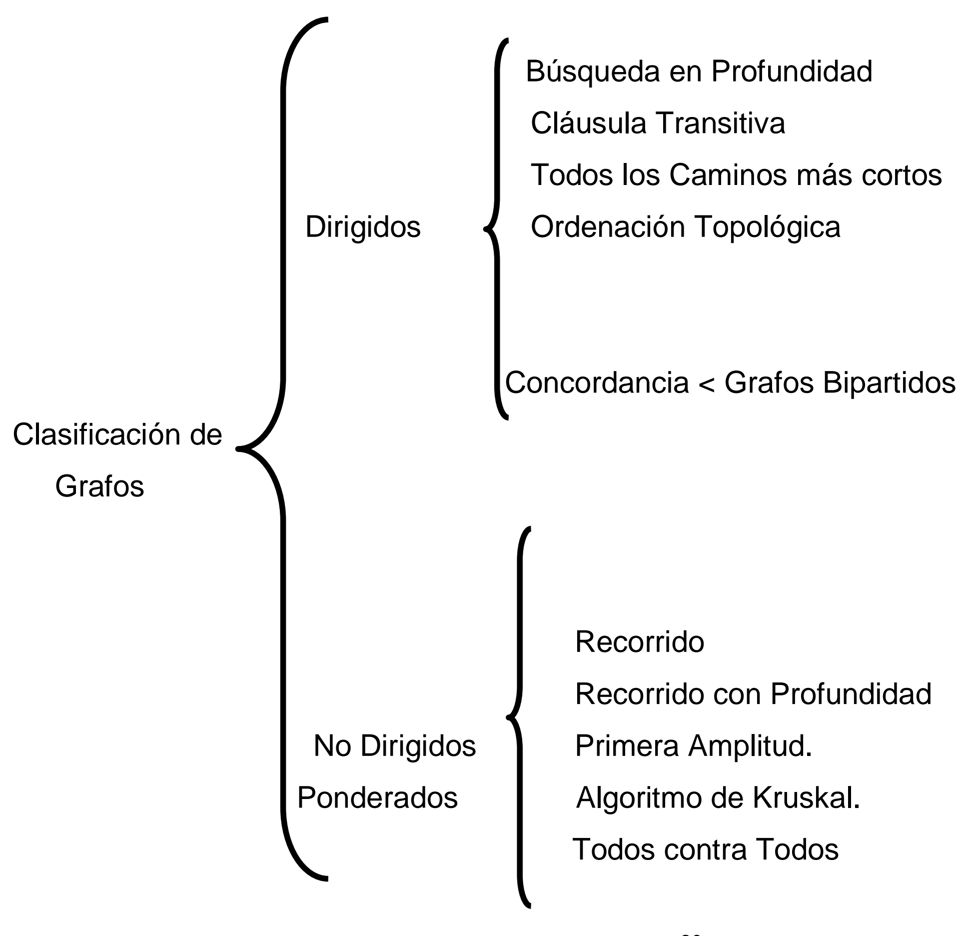
Grado de un Vértice.- Se identifica como grado (*v*), y es el total de aristas de aristas que tienen como extremo a *v*. Cuando su grado = 0, este recibe el nombre de vértice aislado.

Grafo Sin Ciclos.- Árbol, Árbol libre, Grafo conexo sin ciclos

Lazo.- Es una arista que tiene en ambos extremos el mismo vértice. Se vértice. Se expresa como a = (*v, v).*

Multigrafo.- Si al menos dos de sus vértices están unidos entre sí por dos sí por dos aristas paralelas o múltiples.

Subgrafo.- Esta formado por un subconjunto de vértices y de aristas de un grafo dado. En notación de Conjuntos: G' de G se define como G' = (V' , A'), donde V' ≤ V y A' ≤ A.



Cuadro 3.3 Tipos de Grafos[20]

3.4.1 Definición del tipo de dato abstracto grafo[21]

[20] Cf., R. Sedgewick, op. cit., pp. 515- 547 y S GUARDATI, op. cit., pp. 393- 446

Los datos contienen, en algunos casos, relaciones entre ellos que no son necesariamente jerárquicas. Por ejemplo, supongamos que unas líneas aéreas realizan vuelos entre las ciudades conectadas por líneas como se ve en la figura anterior (más adelante se presentarán grafos con estructuras de datos); la estructura de datos que refleja esta relación recibe el nombre de grafo.

Se suelen usar muchos nombres al referirnos a los elementos de una estructura de datos. Algunos de ellos son “elemento”, “ítem”, “asociación de ítems”, “registro”, “nodo” y “objeto”. El nombre que se utiliza depende del tipo de estructura, el contexto en que usamos esa estructura y quien la utiliza.

En la mayoría de los textos de estructura de datos se utiliza el término “registro” al hacer referencia a archivos y “nodo” cuando se usan listas enlazadas, árboles y grafos.

También un grafo es una terna **G = (V,A,j),** en donde **V** y **A** son conjuntos finitos, y **j** es una aplicación que hace corresponder a cada elemento de **A** un par de elementos de **V**. Los elementos de **V** y de **A** se llaman, respectivamente, **"vértices"** y **"aristas"** de **G**, y **j** asocia entonces a cada arista con sus dos vértices.

Esta definición da lugar a una representación gráfica, en donde cada vértice es un punto del plano, y cada arista es una línea que une a sus dos vértices.

Si el dibujo puede efectuarse sin que haya superposición de líneas, se dice que G es un grafo plano. Por ejemplo, el siguiente es un grafo plano:

puesto que es equivalente a este otro:

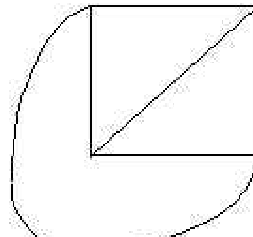


Figura 3.13. Enlaces equivalentes en un nodo del Grafo

3.4.2. Operaciones sobre un grafo[22]

Búsqueda en grafos

Para efectuar una búsqueda de los vértices de un grafo, se pueden emplear dos estrategias diferentes:

---

[21] Felipe Costales, “Área Informática, Estructuras de Datos”, Monografías, http://www.monografias.com/trabajos/grafos/grafos.shtml, octubre 18 del 2008.

[22] Ver, Seymour, Lipschutz, *Estructura de datos,* Madrid, (Schaum) McGraw Hill, Capítulo 8, página: 337.

Búsqueda en profundidad (BEP): Se comienza en cualquier vértice y en cada paso se avanza a un nuevo vértice adyacente siempre que se pueda. Cuando todos los adyacentes a **X** hayan sido visitados, se retrocede al vértice desde el que se alcanzó **X** y se prosigue. Así se consigue etiquetar (visitar) todos los vértices de la componente conexa en que se encuentre el vértice inicial.
Esta técnica se utiliza cuando necesitamos encontrar respuesta a un problema sobre un grafo sin condiciones de optimización.

La idea en general de la búsqueda en profundidad comenzando en un nodo **A** es la siguiente:

Primero examinamos el nodo inicial **A**. Luego examinamos cada nodo **N** de un camino **P** que comience en **A**; o sea, procesamos un vecino de **A**, luego un vecino de un vecino de **A** y así sucesivamente, hasta llegar a un punto muerto o final del camino **P**, y de aquí volvemos atrás por **P** hasta que podamos continuar por otro camino **P´** y así sucesivamente.
Este algoritmo es similar al del recorrido inorden de un árbol binario, y también a la forma en que se debe pasar a través de un laberinto. Observa que se hace uso una pila en lugar de una cola, y este es el detalle fundamental que hace la diferencia para realizar la búsqueda en profundidad.[23]

3.4.3. Implantación de un grafo
Algoritmo para la búsqueda en profundidad:

Este algoritmo realiza la búsqueda en profundidad el grafo **G** comenzando en un nodo **A**.

1. Inicializar todos los nodos al estado de preparado (ESTADO=1)
2. Meter el nodo inicial A en la pila y cambiar su estado a estado de espera (ESTADO=2).
3. Repetir los pasos 4 y 5 hasta que la pila esté vacía.
4. Sacar el nodo N en la cima de la pila. Procesar el nodo N y cambiar su estado al de procesado (ESTADO=3).
5. Meter en la pila todos los vecinos de N que estén en estado de preparados (ESTADO=1) y cambiar su estado a estado de espera (ESTADO=2).
   [fin de bucle del paso 3]
6. Salir.

[23] Mary Luz Sánchez S., "Grafos", material electrónico disponible en: http://www.ucentral.edu.co/pregrado/escuelainge/sisitemas/anexos-seminaro/Grafos%20%20documento%20estudiantes.doc, recuperado el 18/10/08

Búsqueda en anchura (BEA)

A diferencia con la BEP ahora se visitan todos los vecinos de un vértice antes de pasar al siguiente. Por tanto no hay necesidad de retroceder. Una vez etiquetados todos los vecinos de un vértice **X**, se continúa con el primer vértice alcanzado después de **X** en la búsqueda.

Esta técnica se utiliza para resolver problemas en los que se pide hallar una solución óptima entre varias.

En general la búsqueda en anchura comenzando de un nodo de partida **A** es la siguiente:

Primero examinamos el nodo de partida **A**.
Luego examinamos todos los vecinos de **A**. Luego examinamos todos los vecinos de los vecinos de **A** y así sucesivamente. Con el uso de una cola, garantizamos que ningún nodo sea procesado más de una vez y usando un campo **ESTADO** que nos indica el estado actual de los nodos.

Algoritmo para la búsqueda en anchura:

Este algoritmo realiza la búsqueda en anchura en un grafo **G** comenzando en un nodo de partida **A**.

```
1. Inicializar todos los nodos al estado de preparados
   (ESTADO=1).
2. Poner el nodo de partida A en la COLA y cambiar su
   estado a espera (ESTADO=2).
3. Repetir pasos 4 y 5 hasta que COLA esté vacía.
4.  Quitar el nodo del principio de la cola, N.
   Procesar N y cambiar su
        estado a procesado (ESTADO=3).
5.  Añadir a COLA todos los vecinos de N que estén en
   estado de
        preparados (ESTADO=1) y cambiar su estado al
   de espera
        (ESTADO=2).
        [ fin del bucle del paso 3 ]
6. Salir.
```
[24]

Árboles de recubrimiento mínimo (búsqueda del camino más corto): 25

Caminos mínimos en grafos

[24] Mary Luz Sánchez S., "Grafos", material electrónico disponible en: http://www.ucentral.edu.co/pregrado/escuelainge/sisitemas/anexos-seminaro/Grafos%20%20documento%20estudiantes.doc, recuperado el 18/10/08.[25] Seymour Lipschutz, *Estructura de datos*, serie Schaum McGraw Hill, Capítulo 8, página: 337.

[25] Seymour Lipschutz, *Estructura de datos*, serie Schaum McGraw Hill, Capítulo 8, página: 337.

Para lograr el propósito del recorrido mínimo dentro de un grafo G, es necesaria para nuestro caso en particular (puesto que no es la única técnica existente), la utilización del algoritmo de **WARSHALL** -para el camino mínimo-se expresa de la forma siguiente:

Sea G un grafo con m nodos, υ_1 , υ_2 , ..., υ_m suponga que queremos encontrar la matriz de caminos P para el grafo G. Warshall dio un algoritmo para este propósito que es mucho más eficiente que calcular las potencias de la matriz de adyacencia A y aplicar la proposición:

$$B_m = A + A^2 + A^3 + \ldots + A^m$$

donde sea **A** la matriz de adyacencia y **P** = **P**$_{ij}$ la matriz de caminos de un grafo **G** entonces, **P**$_{ij}$ = **1** si y solo sí hay un número positivo en la entrada **ij** de la matriz

Este algoritmo de WARSHALL se usa para calcular el camino mínimo y existe un algoritmo similar para calcular el camino mínimo de G cuando G tiene peso.

Algoritmo de WARSHALL: [26]

Un grafo dirigido **G** con **M** nodos está en memoria por su matriz adyacente **A**, este algoritmo encuentra la matriz de caminos (Booleana) **P** del grafo **G**.

```
1. [ Inicializar P ] repetir para I, J =1, 2, ... M:
      si A[ I, J ]=0, entonces: hacer P[ I, J ]:=0;
      si no: hacer P[ I, J ]:=1.
   [ fin de bucle ]
2. [ Actualizar P ] repetir paso 3 y 4 para K=1, 2,
   ..., M:
3.  repetir paso 4 para I=1, 2, ..., M:
4.       repetir para J=1, 2, ..., M:
   hacer P[ I, J ]:= P[ I, J ] V ( P[ I, J] ^ P[ K, J
   ]).
         [ fin de bucle ]
       [ fin de bucle paso 3 ]
    [ fin de bucle paso 2 ]
5. Salir.
```

Algoritmo de matriz de camino mínimo: [27]

[26] ibid., p. 322.
[27] ibid., p. 324.

Cuando se trata de un grafo con peso G de M nodos está memoria mediante su matriz de peso W; este algoritmo encuentra la matriz Q tal que [I, J] es la longitud del camino mínimo del nodo **V_I** al nodo **V_J** . INFINITO es un número muy grande y MIN es la función del valor mínimo.

1. [Inicializar Q] repetir para I, J=1, 2, . . ., M:
si W [I, J] = 0, entonces: hacer Q [I, J]:= INFINITO;
si no: hacer Q [I, J] := W [I, J].
[fin de bucle]
2. [Actualizar Q] repetir pasos 3 y 4 para K=1, 2, . . ., M:
3. repetir paso 4 para I = 1, 2, . . ., M:
4. repetir para J = 1, 2, . . ., M:
hacer Q [i, J] := MIN(Q [i, J]+ Q [i, K]+ Q [K, J]).
[fin de bucle]
[fin de bucle del paso 3]
[fin de bucle del paso 2]
5. Salir.

Enunciado para ejemplo:

Dado un grafo simple no dirigido, conexo y ponderado de **n** nodos etiquetados con los números naturales desde el 1 hasta el **n**, se numeran los ejes desde 1 hasta **m** de acuerdo con el orden. Dados a continuación dos nodos cualesquiera, se trata de encontrar el camino más corto entre ambos nodos, utilizando el algoritmo de Dijkstra.

Entrada: En la primera línea, un número natural que indica el número de casos que se van a plantear. Para cada caso, una línea con el número de nodos n del grafo, y la representación decimal del mismo (entero menor que $2\binom{n}{2}$) separados por un blanco. En la siguiente línea, separados por blancos, m números naturales que representan los pesos de los ejes del grafo. En la siguiente línea, otro número natural p nos dice cuantos pares de nodos se van a proponer, y a continuación aparecen en líneas diferentes y separados por blancos todas estas parejas.
Salida: Para cada uno de los casos propuestos, el fichero de salida contendrá una línea indicando el caso de que se trata en la forma Grafo # con el símbolo # sustituido por el número del caso. Las siguientes m líneas contendrán la lista de adyacencias del grafo en la forma:

No.delnodo	*Nodoadyacente*	*pesodeleje*	*Nodoadyacente*	*Pesodeleje...*

siempre separando con blancos y con los nodos adyacentes en orden creciente de número. A continuación, **p** líneas que resuelven las **p** parejas de nodos planteadas, componiendo cada línea en la forma:

Pesodelcamino... ...nodosintermedios... ...nodofinal...

Ejemplo de Entrada:

2
4 49
53 82 53
2
1 2
1 3
8 14728196
81 48 30 64 71 13 91 10 65
3
2 1
4 1
8 1

Ejemplo de Salida:
Grafo 1
1 2 53
2 1 53 4 82
3 4 53
4 2 82 3 53
53 1 2
188 1 2 4 3
Grafo 2
1 4 81
2 6 48 7 30 8 64
3 4 71 6 13
4 1 81 3 71 8 91
5 6 10 7 65
6 2 48 3 13 5 10
7 2 30 5 65
8 2 64 4 91
213 2 6 3 4 1
81 4 1
172 8 4 1

Algoritmo de Dijkstra:

Este algoritmo construye el árbol de caminos de longitud mínima entre un vértice fijado **V** y los restantes vértices en un grafo ponderado.
Observaciones:

1) Los pesos de las aristas deben ser no negativos.
2) El algoritmo de Dijkstra NO proporciona un árbol generador mínimo.[28]

[28] Costales, Felipe, "Programación no numérica: grafos", material en línea, disponible en: http://www.monografias.com/trabajos/grafos/grafos.shtml, recuperado el 18/10/08.

Bibliografía del tema 3

Allen Weiss, Mark, *Estructura de datos y algoritmos*, México, Addison-Wesley Iberoamericana, 1995, 180 pp.

Drozdek, Adam, *Estructura de Datos y Algoritmos en Java,* 2ª ed., México, International Thomson Editores, 2007.

Guardati Bueno, Silvia, *Estructura de Datos. Algoritmos con C++,* México Pearson Education, 2007. [ISBN 10-970-26-0792-2]

Joyanes Aguilar, Luis, *Fundamentos de programación: algoritmos y estructura de datos,* México, McGraw Hill, 1990, 407 pp.

Langsam, Yedidyah, *Estructuras de Datos en C y C++,* 2ª ed., México, Prentice Hall, 1997, 672 pp. [ISBN 0-13-036997-7].

Sedgewick, Robert, *Algoritmos en C++,* México, Pearson Education, 1995, ISBN 968-444-401-x.

Seymour, Lipschutz, *Estructura de datos,* Madrid, (Schaum) McGraw Hill, 1987

Otras fuentes

Fiestas Bancayán, Héctor, “Estructuras dinámicas de datos”, en línea, disponible en: es.geocities.com/hwfiestasb/Catedras/EstructurasDatos/SESION_9_A_ESTRUCTURAS_DINAMICAS.pdf.

Sánchez S., Mary Luz, “Grafos”, material en línea, disponible en: http://www.ucentral.edu.co/pregrado/escuelainge/sisitemas/anexos-seminaro/Grafos%20%20documento%20estudiantes.doc

Siso, Leandro, “Área Informática, Estructuras de Datos”. Monografías, disponible en: http://www.monografias.com/trabajos36/arboles/arboles2.shtml.

Algoritmia: *Estructuras de datos*, 01/07/03, material en línea, disponible en: http://www.algoritmia.net/articles.php?folder=Estructuras%20de%20Datos

Friesen, Jeff: *Estructuras de datos y algoritmos en Java*, material en línea, disponible en: http://www.programacion.com/tutorial/jap_data_alg/ o bien en inglés: http://www.javaworld.com/

Actividades de Aprendizaje

A.3.1. Investigar las definiciones de Lista, Listas con Prioridades y Bicola y entregarlas por escrito.

A.3.2. Buscar en Internet aplicaciones de Listas, Árboles y Grafos y entregarlas por escrito.

A.3.3. Representa gráficamente la inserción y eliminación en una Pila.

A.3.4. Busca la representación de Listas y Colas en el Lenguaje Java y anota el código correspondiente para después capturar y probar el Programa.

A.3.5. Investigar cuáles son las formas de Representar a los Árboles para elaborar el TDA del Objeto árbol.

A.3.6. Investigar y reportar las Propiedades de los Árboles en el libro de Robert Sedgewick , *Algoritmos en C++*, México, Pearson Educación, 2000, páginas 43 y 44.

A.3.7. Del Árbol representado en la Figura 3.8 contestar lo siguiente:

a) Niveles del Árbol.
b) Altura del Árbol.
c) Tipo de Árbol.
d) Identificar cuántos subárboles tiene.
e) Cuenta cuántos nodos internos tiene.

A.3.8. Consultar las ligas que están en Otras Fuentes para ver los códigos de los programas y comprobar su validez. Pregunta tus dudas a tu asesor.

Cuestionario de autoevaluación

1. Anota los Conceptos de Listas, Listas Enlazadas y Doblemente Enlazadas.
2. Cuáles son las diferencias entre Pilas y Colas.
3. Cuáles son las diferencias entre Árboles y Grafos.
4. Cuáles son las Partes de los Árboles y los Grafos.
5. Cuáles son las formas de representar a los Árboles.
6. ¿Qué significa el término Implementar en Programación?
7. ¿Cuáles son los tipos de Árboles?
8. ¿Cuáles son los tipos de Grafos?
9. Anota el Procedimiento para crear una Lista Nula.
10. Cuando el tamaño de Archivo crece, ¿Qué Estructuras se aplican para realizar búsquedas en él?

Examen de Autoevaluación

I. Indica si la oración es falsa o verdadera.

1.	Una Pila es una forma de Lista	F	V
2.	Una Cola puede ser cambiada de dirección en su recorrido	F	V
3.	Un Árbol B requiere de nodos	F	V
4.	En un Grafo se puede autoapuntar al mismo nodo	F	V
5.	Un Grafo es una estructura jerárquica y monolítica	F	V
6.	Existen dos recorridos en una Estructura de Árbol	F	V
7.	Las Listas se aplican al mapeo de la Memoria Interna	F	V
8.	En el Lenguaje COBOL se pueden implementar las Colas	F	V

II. Completa la oración

1. Es el ________ raíz de donde parte la Estructura de ________.
2. Varias Estructuras de Árboles forman un __________.

Bibliografía básica

Allen Weiss, Mark, *Estructura de datos y algoritmos*, México, Addison-Wesley Iberoamericana, 1995, 180 pp.

Drozdek, A. (2007). *Estructura de Datos y Algoritmos en Java*, 2ª ed., México: Thomson Learning, 2005 [isbn 0-534-49252-5].

Guardati Bueno, Silvia, *Estructura de Datos. Algoritmos con C++,* México Pearson Education, 2007. [ISBN 10-970-26-0792-2]

Hernández Castillo, Vicente. Guía Didáctica de Informática II, México, SUA-UNAM.

Joyanes Aguilar, Luis, Estructura de Datos: Algoritmos, abstracción y objetos, 3ª ed., McGraw-Hill, Madrid, 1999. [ISBN: 8448106032]

Langsam, Yedidyah, *Estructuras de Datos en C y C++,* 2ª ed., México, Prentice Hall, 1997, 672 pp. [ISBN 0-13-036997-7].

Sedgewick, Robert, *Algoritmos en C++,* México, Pearson Education, 1995, ISBN 968-444-401-x.

Seymour, Lipschutz, *Estructura de datos,* Madrid, (Schaum) McGraw Hill, 1987

Weiss, Mark Allen, *Estructura de datos y algoritmos*, México, Addison-Wesley Iberoamericana, 1995.

Sitios Web

Ana Ma. Toledo Salinas, “¿Qué es pilas?”, material en línea, disponible en: http://boards4.melodysoft.com/app?ID=2005AEDI0303&msg=19

Costales, Felipe, “Programación no numérica: grafos”, material en línea, disponible en: http://www.monografias.com/trabajos/grafos/grafos.shtml

“Estructura de datos, material en línea, disponible en: http://www.monografias.com/trabajos14/estruct-datos/estruct-datos.shtml

Facultad de Informática. Universidad de A Coruña. *Estructura de datos: Tablas de dispersión. Algoritmos*. Disponible en: http://quegrande.org/apuntes/EI/2/Alg/teoria/07-08/tema_2.4_-_tablas_de_dispersion.pdf

Galeón, Rigel: “Estructuras dinámicas de datos”, p. 5, material en línea, disponible en: http://rigel.galeon.com/dinamicas.doc.

Programación fácil, “C++ Estructuras o Registros”, material en línea, dispobible en: http://www.programacionfacil.com/cpp:estructuras_registros

http://es.wikipedia.org/wiki/Estructura_de_datos.

Wikipedia, “Tabla hash”, 27/07/08, en línea, disponible n: http://es.wikipedia.org/wiki/Tabla_hash.

Respuestas a los Exámenes de Autoevaluación

Informática II

	Tema 1	Tema 2		Tema 3	
1.	F	arreglo	V	V	nodo, árbol
2.	V	puntero	F	V	bosque
3.	V		F	V	
4.	V		V	V	
5.	F		F	F	
6.	F		F	F	
7.	V		V	V	
8.	F		V	V	
9.	F				
10.	V				