# T04 - ESTRUCTURAS DE DATOS.


**1. TABLAS, LISTAS Y ÁRBOLES ..... 3**

1.1 TIPOS ABSTRACTOS DE DATOS ..... 3
*1.1.1 EL CONCEPTO DE ALGORITMO ..... 3*
*1.1.2 COSTE DE UN ALGORITMO ..... 4*
*1.1.3 IMPLANTACIÓN TRADICIONAL FRENTE A LOS TAD ..... 5*
*1.1.4 OPERACIONES DE LOS TAD ..... 6*
*1.1.5 IMPLANTACIÓN Y TIPOS DE TAD ..... 6*
1.2 TABLAS ..... 6
*1.2.1 TIPOS ..... 7*
*1.2.2 OPERACIONES ..... 7*
*1.2.3 REPRESENTACIÓN ..... 8*
*1.2.4 IMPLANTACIÓN ..... 8*
1.3 LISTAS ..... 8
*1.3.1 TIPOS ..... 9*
*1.3.2 OPERACIONES ..... 9*
*1.3.3 REPRESENTACIÓN ..... 9*
*1.3.4 IMPLANTACIÓN ..... 10*
1.4 ÁRBOLES ..... 11
*1.4.1 TIPOS ..... 12*
*1.4.2 OPERACIONES ..... 13*
*1.4.3 REPRESENTACIÓN ..... 15*
*1.4.4 IMPLANTACIÓN ..... 15*

**2. ALGORITMOS: ORDENACIÓN, BÚSQUEDA, RECURSIÓN, GRAFOS ..... 15**

2.1 ORDENACIÓN ..... 15
*2.1.1 SELECCIÓN ..... 16*
*2.1.2 BURBUJA ..... 16*
*2.1.3 INSERCIÓN DIRECTA ..... 17*
*2.1.4 INSERCIÓN BINARIA ..... 18*
*2.1.5 SHELL ..... 18*
*2.1.6 INTERCALACIÓN ..... 18*
2.2 BÚSQUEDA ..... 19
*2.2.1 BÚSQUEDA SECUENCIAL ..... 19*
*2.2.2 BÚSQUEDA BINARIA O DICOTÓMICA ..... 20*
*2.2.3 BÚSQUEDA UTILIZANDO TABLAS HASH ..... 20*
2.3 RECURSIVIDAD ..... 21
*2.3.1 DEFINICIÓN DE RECURSIVIDAD ..... 21*
*2.3.2 USO DE LA RECURSIÓN ..... 22*
*2.3.3 ORDENACIONES Y BÚSQUEDAS RECURSIVAS ..... 22*
2.4 GRAFOS ..... 23
*2.4.1 COMPONENTES DE UN GRAFO ..... 24*
*2.4.2 DEFINICIÓN FORMAL DE GRAFO ..... 24*
*2.4.3 CONCEPTOS FUNDAMENTALES SOBRE GRAFOS ..... 25*
*2.4.4 EXPLORACIÓN DE GRAFOS ..... 26*
*2.4.5 IMPLANTACIÓN DE LA EXPLORACIÓN DE GRAFOS ..... 27*

**3. ORGANIZACIONES DE FICHEROS ..... 27**

3.1 ESTRUCTURA DE UN FICHERO ..... 29
3.2 CONCEPTOS BÁSICOS SOBRE FICHEROS ..... 29
3.3 OPERACIONES SOBRE FICHEROS ..... 31
3.4 ORGANIZACIÓN DE UN SISTEMA DE FICHEROS ..... 31
3.5 ORGANIZACIÓN Y ACCESO A FICHEROS ..... 33

*3.5.1 ORGANIZACIÓN SECUENCIAL* ... *33*
*3.5.2 ORGANIZACIÓN DIRECTA O ALEATORIA* ... *34*
*3.5.3 ORGANIZACIÓN INDEXADA* ... *35*
3.6 IMPLANTACIÓN DE FICHEROS ... 36
*3.6.1 ASIGNACIÓN CONTIGUA* ... *36*
*3.6.2 ASIGNACIÓN ENLAZADA* ... *37*
*3.6.3 ASIGNACIÓN INDEXADA* ... *38*
3.7 DIRECTORIOS ... 39
3.8 IMPLANTACIÓN DE DIRECTORIOS ... 40
*3.8.1 DIRECTORIOS EN CP/M* ... *40*
*3.8.2 DIRECTORIOS EN MS-DOS* ... *41*
*3.8.3 DIRECTORIOS EN UNIX* ... *41*

**4. FORMATOS DE INFORMACIÓN Y FICHEROS ... 41**

4.1. FORMATOS DE INFORMACIÓN ... 42
*4.1.1 FICHEROS CON INFORMACIÓN DE TEXTO* ... *42*
*4.1.2 FICHEROS CON INFORMACIÓN DE IMAGEN* ... *42*
*4.1.3 FICHEROS CON INFORMACIÓN COMPUESTA* ... *42*
*4.1.4 FICHEROS CON INFORMACIÓN COMPRIMIDA* ... *43*
4.2. TIPOS DE FICHERO SEGÚN SU USO ... 43
4.3. DENOMINACIÓN DE FICHEROS ... 43

**5. CONCLUSIÓN ... 44**

**6. BIBLIOGRAFÍA ... 45**

**7. ESQUEMA – RESUMEN ... 46**

# 1. TABLAS, LISTAS Y ÁRBOLES

## 1.1 TIPOS ABSTRACTOS DE DATOS

Al escribir un programa para resolver un problema, el enfoque tradicional consistía en pasar de la realidad a una implantación en el lenguaje de programación utilizado, lo que conducía, inevitablemente, a que la forma de "ver" los datos estuviera muy influida por la máquina donde se ejecutaba el programa. Con el tiempo y la experiencia acumulada surgió otro enfoque mejor para afrontar esta situación, que consistía en establecer un nivel intermedio, donde se modela lo esencial de la realidad, mediante técnicas de abstracción, sin comprometerse con detalles de implantación. Estas técnicas de abstracción han evolucionado en el sentido de alejar los elementos que aparecen en los sistemas de software de las nociones propias de las máquinas sobre las que se implantan dichos sistemas, aproximándolos a las nociones propias de los dominios en que se presentan las situaciones modeladas.

La principal de estas técnicas está basada en los ***Tipos Abstractos de Datos (TAD).***

Esta técnica está centrada en las abstracciones de datos, consiste en:

- □ Identificar los distintos tipos de datos que intervienen en el sistema y la función principal de dicho sistema.
- □ Caracterizar cada tipo de datos en función de las operaciones que se puedan realizar con los objetos de los distintos tipos (haciendo abstracción de sus representaciones concreta).
- □ Componer el sistema utilizando objetos de los tipos definidos junto con sus operaciones características.
- □ Implantar cada uno de los tipos utilizados.

Estas características son la base conceptual de los TAD. De forma sintetizada, puede decirse que:

*Un TAD es una estructura algebraica, a saber, un conjunto de objetos estructurados de alguna forma y con ciertas operaciones definidas sobre ellos*.

Piénsese, por ejemplo, en una calculadora; los elementos que maneja son cantidades numéricas y las operaciones definidas son operaciones aritméticas. Otro ejemplo posible es el TAD matriz matemática; los elementos que maneja son las matrices y las operaciones son las que nos permiten crearlas, sumarlas, invertirlas, etc. Otro tipo de TAD es cualquier tipo de cola de espera: en el autobús, cine, la compra, etc. Los elementos de las colas son las personas y las operaciones más usuales son entrar o salir de la cola, pero no son las únicas. Desde un punto de vista abstracto podemos pensar en operaciones como crear un elemento de la cola, comprobar si la cola está vacía, si está llena, etc.

Es conveniente observar que las operaciones de un TAD son de diferentes clases. Algunas nos deben permitir crear objetos nuevos, otras determinar su estado, construir otros objetos a partir de algunos ya existentes, etc.

Puesto que los TAD deben ser los más independientes posibles de los detalles de implantación, es obvio que no deben tener ninguna dependencia con respecto al lenguaje de programación elegido para implantarlos. Aún así, ciertos lenguajes no incorporan ciertas estructuras de datos necesarias para implantar algunos tipos de TAD.

### 1.1.1 EL CONCEPTO DE ALGORITMO

Un **algoritmo** es *un conjunto de pasos o instrucciones que se deben seguir para realizar una determinada tarea*.

Estas instrucciones deben cumplir las siguientes características:

- □ **FINITUD**: Debe ser *un conjunto finito de instrucciones* y que se realicen en un *tiempo finito*.
- □ **PRECISIÓN**: Debe indicar el *orden de realización de cada instrucción o paso de forma inequívoca*.

- **DEFINICIÓN**: Debe tener un *número finito (0 ... N) de datos de entrada* y un *número finito (0 ... M) de datos de salida (resultados).* Frente a un mismo conjunto de datos de partida se debe llegar siempre a un mismo conjunto de resultados.

Otra cualidad deseable en un buen algoritmo, aunque no imprescindible para ser considerado como tal, es que sea *óptimo*, es decir, que sea la forma más fácil y rápida de hacer una determinada tarea, si bien es cierto que en muchas ocasiones facilidad y rapidez son dos cualidades contrapuestas.

De forma gráfica, la siguiente figura engloba las principales características de un algoritmo.

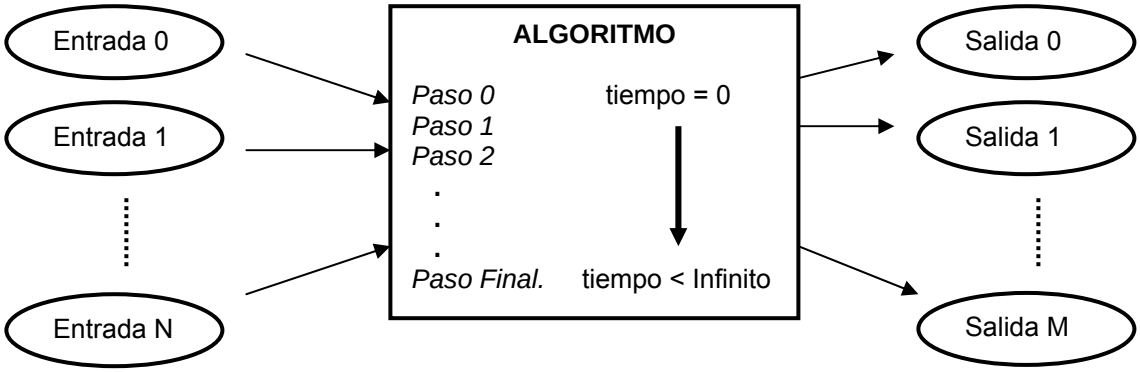



*Figura 1. Concepto de algoritmo.*

### 1.1.2 COSTE DE UN ALGORITMO

Normalmente, si se escribe un programa para resolver un problema, se hace para que éste sea utilizado muchas veces, por lo que resulta conveniente caracterizarlo según su tiempo de ejecución y la calidad de la respuesta. Cuando estudiamos algoritmos es muy importante caracterizar la solución obtenida de cada algoritmo antes de estar seguros de que dos algoritmos son equivalentes (para un mismo valor de entrada dan exactamente el mismo resultado) o son similares (pueden dar resultados diferentes, pero desde el punto de vista del problema que estamos resolviendo somos indiferentes a cualquiera de los resultados).

Por esta razón, uno de los criterios más importantes para seleccionar un algoritmo es evaluar el tiempo que tarda en ejecutarse, que llamaremos ***coste del algoritmo*** ó ***tiempo de ejecución***. Para analizar el coste de un algoritmo adoptamos un modelo que nos dice que recursos usados por la implantación del algoritmo son importantes para su desempeño. La complejidad de un algoritmo bajo un modelo computacional es la cantidad de recursos, en tiempo o espacio, que el algoritmo usa para resolver el problema.

Desafortunadamente, por lo general es imposible predecir el comportamiento exacto de un algoritmo, ya que existe la influencia de muchos factores, de aquí que se trate de extraer las características principales de un algoritmo. Así, se definen ciertos parámetros y ciertas medidas que son las más importantes en el análisis y se ignoran detalles relativos a la implantación.

Así, el análisis es una aproximación, no es perfecto, pero lo importante es que se pueden comparar diferentes algoritmos para determinar cuál es mejor para un propósito determinado.

La metodología que generalmente se utiliza para predecir el tiempo de ejecución de algoritmos se basa en el ***comportamiento asintótico***, en este método se ignoran los factores constantes y se concentra en el comportamiento del algoritmo cuando el tamaño de la entrada tiende a infinito. Es totalmente análogo al estudio de los límites de una función cuando su variable independiente tiende a infinito, por ejemplo:

$F(x) = 6x^2 + x + 3$. Haciendo $x \to \infty$ (infinito) esta función es equivalente a: $F(x) = 6x^2$

De hecho, es muy usual evaluar el coste de un algoritmo como una función matemática de el número de entradas del algoritmo (N), siendo un algoritmo de mayor coste que otro cuando para un mismo N la función que expresa el coste da un resultado superior. Así pues, los costes o tiempos de ejecución más usuales de los algoritmos, de menor a mayor, son:

| | |
|---|---|
| **1** | Constante |
| **Log N** | Logarítmico |
| **N** | Lineal |
| **N * (Log N)** | Semi-Logarítmico |
| **$N^2$** | Cuadrático |
| **$N^3$** | Cúbico |
| **N!** | Factorial |
| **$2^n$** | Exponencial |

*Figura 2. Costes más usuales de los algoritmos. N es el número de entradas.*

Por otra parte, conviene considerar dos casos límite, el ***mejor caso*** y el ***peor caso***. Puede ocurrir que al realizar una operación se parta de unos datos iniciales del TAD que hagan que al aplicar el algoritmo éste se ejecute de la manera más rápida posible, y puede ocurrir también lo contrario, encontrarnos con el caso en que se ejecute de la forma más lenta. Por ejemplo, si queremos ordenar unos datos, parece lógico pensar que el mejor caso será cuando los datos ya estén ordenados, y el peor caso cuando ningún dato inicial esté en el orden que tendrá cuando se haya ejecutado la ordenación.

De forma estadística se supone que se parte de un caso medio. Por otra parte, hay algoritmos cuya diferencia de coste entre los casos mejor y peor es muy grande, y otros en los que, debido a la naturaleza del algoritmo, la diferencia es muy pequeña o incluso inexistente.

Si un algoritmo crece de manera proporcional a N, se dice que es de orden N. En general, el tiempo de ejecución es proporcional, esto es, multiplica por una constante a alguno de los costos anteriormente propuestos, además de la suma de algunos términos más pequeños. Así, un algoritmo cuyo tiempo de ejecución sea $T = 3N^2 + 6N$ se puede considerar proporcional a $N^2$. En este caso se diría que el algoritmo es del orden de $N^2$, y se escribe $O(N^2)$, llamada notación *O-grande*. La notación O-grande ignora los factores constantes, es decir, ignora si se hace una mejor o peor implantación del algoritmo, además de ser independiente de los datos de entrada del algoritmo. La utilidad de aplicar esta notación a un algoritmo es encontrar un límite superior del tiempo de ejecución, es decir, el peor caso.

### 1.1.3 IMPLANTACIÓN TRADICIONAL FRENTE A LOS TAD

Según Nicklaus Wirth, un programa responde a la ecuación:

(Ec. 1) ***Programa = Datos + Algoritmos***

El enfoque tradicional se ciñe bastante bien a esta concepción. En la ecuación de Wirth la parte *Algoritmo* la podemos expresar como:

(Ec. 2) ***Algoritmo = Algoritmo de datos + Algoritmo de control***

Se entiende como *Algoritmo de datos* a la parte del algoritmo encargada de manipular las estructuras de datos del problema, y *Algoritmo de control* a la parte restante ( la que representa en sí el método de solución del problema, también llamada *lógica de negocio*, independiente hasta cierto punto de las estructuras de datos seleccionadas ).

Con los TAD se identifican ciertas operaciones o partes del algoritmo que manipulan los datos. Además de proporcionar una estructura a los datos, por lo que podemos sustituir el sumando "*Datos"* de la ecuación anterior por el sumando "*Estructura de Datos".* De esta forma, podemos entonces escribir:

(Ec. 3) ***Programa = Estructura de Datos + Algoritmo de Datos + Algoritmos de Control***

Definiendo:

(Ec. 4) ***Implantación del TAD = Estructura de Datos + Algoritmos de Datos***

Se establece la *ecuación fundamental* que describe el enfoque de desarrollo con Tipos Abstractos de Datos:

(Ec. 5) ***Programa = Implantación del TAD + Algoritmo de Control***

A la hora de crear un tipos abstracto de datos, la ecuación a seguir es (Ec. 4), es decir, determinar como es la ***estructura de datos*** y cuales son los algoritmos de control para manejar los datos, en otras palabras, las ***operaciones sobre los datos*** que se pueden hacer, o interesa hacer, sobre la mencionada estructura de datos.

### 1.1.4 OPERACIONES DE LOS TAD

Las operaciones en los TAD pueden servir para crear nuevos objetos abstractos, para obtener información acerca de ellos, y algunas para construir nuevos elementos a partir de otros ya existentes. De esta forma las operaciones las podemos clasificar de esta manera:

- Operaciones de **CREACIÓN**: Se utilizan para definir y/o crear el TAD y sus elementos.
- Operaciones de **TRANSFORMACIÓN**: Se utilizan para realizar algún tipo de transformación en los componentes del TAD. Por ejemplo: asignación de valor, ordenaciones, permutaciones, balanceados, etc.
- Operaciones de **ANÁLISIS**: Se utilizan para obtener información concerniente a cualquiera de los componentes del TAD o de su estructura. Por ejemplo: búsquedas, recorridos, obtención de algún dato del TAD (valor de un componente, cantidad de elementos, profundidad, número de niveles, etc.)

### 1.1.5 IMPLANTACIÓN Y TIPOS DE TAD

Cuando ya se tiene bien diseñado un Tipo Abstracto de Dato, el siguiente paso es decidir una implantación. Esto supone elegir algunas de las estructuras de datos que nos proporcione el lenguaje de programación utilizado para representar cada uno de los objetos abstractos y, por otro lado, escribir una rutina (Procedimiento o función) en tal lenguaje que simule el funcionamiento de cada una de las operaciones que especificadas para el TAD.

La selección de las estructuras de datos determina la complejidad del algoritmo que implanta una operación y su elección es, por esta razón, de gran importancia. Existen estructuras de datos muy dependientes de un lenguaje de programación y debido a esto deben trata de evitarse cuando el TAD se quiere que sea portable.

Existen muchos tipos de TAD, pero los más utilizados son:

- Tablas.
- Listas.
- Árboles.

## 1.2 TABLAS

Una ***tabla***, o *matriz*, es un TAD que representa una estructura homogénea de datos donde se cumple:

- Todos sus *componentes son del mismo tipo*.
- Tiene un *número predefinido* de componentes que no puede variarse en tiempo de ejecución.
- Los elementos de la tabla contienen *una clave* que los identifica de forma unívoca. El conjunto de claves forman un conjunto de *índices* para localizar los elementos.
- Se permite el *acceso directo* a cualquiera de los elementos de la tabla a través de los índices.

La operación fundamental en el uso de una tabla es ***localizar la posición*** de sus elementos con una clave conocida "a priori". Dicho de otra forma, lo más usual es que se quiera conocer el contenido del i-esimo (índice i) elemento de una tabla. Es menos frecuente la operación inversa, dado un valor saber los índices de los elementos cuyo contenido coincide con el valor dado; aunque también resulta útil en ocasiones esta operación.

### 1.2.1 TIPOS

Recordando la (ec. 4) de Wirth, desde el punto de vista de la *Estructura de Datos* de una tabla, la principal característica de su estructura es la *dimensión.*

Se habla entonces de:

- Tabla **Monodimensional**:

  Se refiere a tablas de una dimensión con un determinado número (N) de elementos La declaración de estas tablas responde a la siguiente sintaxis genérica:

  *Nombre_Tabla* = ***matriz*** *[ 1..N ]* ***de*** *Tipo_Elemento;*

- Tablas **Multidimensional**:

  Se refiere a tablas de mas de una dimensión (d), con un determinado número de elementos para cada dimensión ($N_1$, $N_2$, ..., $N_d$). Su declaración es:

  *Nombre_Tabla* = ***matriz*** *[1..$N_1$, 1..$N_2$, ... , 1..$N_d$]* ***de*** *Tipo_Elemento;*

  Aunque cada dimensión puede tener diferente numero de elementos, todas las dimensiones tienen el mismo tipo de elemento, no pudiéndose declarar diferentes tipos de elementos según las diferentes dimensiones de la matriz, pues supondría violar la primera característica de las tablas.

### 1.2.2 OPERACIONES

Desde el punto de las *operaciones* (Algoritmos de Datos), las operaciones básicas en una tabla son:

- ***Definir/crear*** la tabla: Se refiere a la forma que cada lenguaje debe tener para definir la estructura de una tabla y crear una variable del tipo de la tabla definida.
- ***Insertar/Eliminar*** elementos de la tabla: Aunque cuando se define una tabla se indica "a priori" el número de elementos que tiene, eso no quiere decir que desde el principio esos elementos tengan un valor significativo para el uso que se les piensa dar. Es muy común al crear una tabla asignar a todos sus elementos un valor especial (nulo) que indique que en caso de que el elemento tenga ese valor, a todos los efectos, desde un punto de vista abstracto, es como si ese elemento no existiese.
- ***Buscar*** elementos de la tabla: Esta operación es la fundamental en el uso de tablas. Existen distintas formas de hacer búsquedas en una tabla, y según los valores de los componentes y el tipo de búsqueda ésta será más o menos eficiente (rápida).
- ***Ordenar*** los elementos de la tabla: Esta operación resulta muy útil a la hora de realizar búsquedas. Veremos más adelante que resulta mucho más efectivo realizar búsquedas sobre tablas donde se ha establecido algún tipo de orden sobre otras donde no lo hay.

- ***Contar*** los elementos de la tabla: Calcular el número de elementos que hay en la tabla en un momento dado. La acción de preguntar si una tabla está llena o vacía (de elementos significativos) son casos particulares de la operación de contar. También pueden considerarse recuentos más especializados, como contar el número de elementos con un determinado valor, o con valor superior a uno dado, etc.

Hay varios algoritmos que implantan las operaciones de búsqueda y ordenación en una tabla (las más complejas y utilizadas). Las operaciones de definición, creación y recuento resultan muy sencillas de implantar.

### 1.2.3 REPRESENTACIÓN

La forma más simple de representación abstracta de una tabla es:

Nombre de Tabla = { $e_0$, $e_1$, ..., $e_i$,..., $e_{N-1}$ }

Para índices que van desde 0 a (N-1) y siendo $e_i$ el contenido de elemento i de la tabla.

Es muy común la siguiente representación, más intuitiva, de una tabla:

| **TABLA MONODIMENSIONAL DE N ELEMENTOS** | | | | | |
|---|---|---|---|---|---|
| **ÍNDICE** del elemento | *0* | *1* | *2* | **....** | *N – 1* |
| **VALOR** del elemento | Juan | Pedro | Inés | **....** | Ana |

*Figura 3. Representación de una tabla monodimensional.*

En la figura anterior se representa una tabla de una dimensión con N elemento, cuyos índices van desde 0 hasta (N-1), y cuyo contenido se refiere a nombres de persona. Para representar tablas multidimensionales, basta con utilizar una representación como la indicada para cada dimensión de la tabla.

### 1.2.4 IMPLANTACIÓN

Puesto que una de las características de las tablas es la invariabilidad del número de componentes, prácticamente todos los lenguajes de programación implantan las tablas usando memoria estática (en oposición a la memoria dinámica). Una vez que el programa ha sido preparado para ser ejecutado (compilado y enlazado), la posición y el espacio de memoria que utilizará la tabla en la ejecución del programa ya está fijado y no puede de ninguna forma cambiarse o utilizarse para otros fines que no sea almacenar los elementos de la tabla.

Hay otros tipos de TAD cuya implantación se realiza frecuentemente con memoria dinámica, cuyo significado fundamental es que, en oposición a la memoria estática, el tamaño y la posición del espacio de memoria utilizado si puede cambiar en tiempo de ejecución, pudiéndose utilizar para otros fines.

## 1.3 LISTAS

Una ***lista*** es una estructura de datos que cumple:

- Todos sus *componentes son del mismo tipo*.
- Cada *elemento o nodo va seguido de otro* del mismo tipo o de ninguno.
- Sus componentes se almacenan según *cierto orden*.

Nótese las diferencias con respecto a las tablas:

- Los elementos de las tablas no se almacenan según un orden. En las listas siempre hay un orden.

- No existe unos conjuntos de índices asociados a cada posición, tal como ocurre con las tablas. Por tanto, en las listas será preciso ir recorriendo los elementos hasta encontrar el buscado.

### 1.3.1 TIPOS

Una manera de clasificar las listas es por la forma de acceder al siguiente elemento:

- ***Lista densa***: La propia estructura determina cuál es el siguiente elemento de la lista. Por ejemplo, aplicando ciertos "trucos" se puede usar una tabla para implementar una lista, de forma que el siguiente elemento de la lista fuera dado por el orden del índice (no confundir con el orden de los elementos del array). Nótese sin embargo que, puesto que un array es estático, la memoria reservada para implementar esta "lista" también será estática. Veremos más adelante un ejemplo de esta circunstancia.
- ***Lista enlazada***: Cada elemento contiene la información necesaria para llegar al siguiente. La posición del siguiente elemento de la estructura la determina el elemento actual. Es necesario almacenar al menos la posición de memoria del primer elemento. Además es dinámica, es decir, su tamaño cambia durante la ejecución del programa.

  Dentro de las listas enlazadas podemos distinguir, según la cantidad de enlaces por nodo, entre:

  - **Lista simplemente enlazada**: Cada elemento conoce que elemento es el que le sucede en el orden, pero no cual le precede.
  - **Lista doblemente enlazada**: Cada elemento conoce que elemento le precede y cual le sucede en el orden.
  - **Lista con enlaces múltiples**: Son aquellas listas en donde cada elemento, aparte de conocer los elementos que le preceden o suceden, también tiene uno o varios enlaces al primer elemento de otra lista, siendo éstas sublistas de la anterior.

También podemos clasificar las listas en función de que los elementos se coloquen en la lista por orden de llegada y se acceda a ellos por su posición. Los ejemplos más usuales de este tipo de listas son:

- **PILAS** (Estructuras LIFO. Last In First Out): El último elemento en entrar es el primero en salir.
- **COLAS** (Estructuras FIFO. First In First Out): El primer elemento en entrar es el primero en salir.

### 1.3.2 OPERACIONES

Las operaciones básicas a realizar en una lista son las mismas que en las tablas, sin bien la forma en que se realizan en las listas difiere notablemente entre ambos TAD, sobre todo en lo concerniente a las operaciones de Inserción, Eliminación, Búsqueda y Ordenación.

A la hora de insertar o eliminar elementos de una lista, puede distinguirse entre insertar o eliminar el primer elemento de la lista, un elemento intermedio o el último elemento. En todos los casos resulta de capital importancia reordenar los diferentes enlaces entre los elementos de forma correcta, pues realizar mal un enlace o perderlo supone la imposibilidad definitiva de acceder a parte de la lista, o incluso a toda ella.

Respecto a las búsquedas y ordenaciones, veremos en próximos apartados como se implementan estas operaciones en las listas.

### 1.3.3 REPRESENTACIÓN

La representación más habitual de una lista se hace colocando sus elementos entre paréntesis, separados por comas, según muestra el siguiente ejemplo:

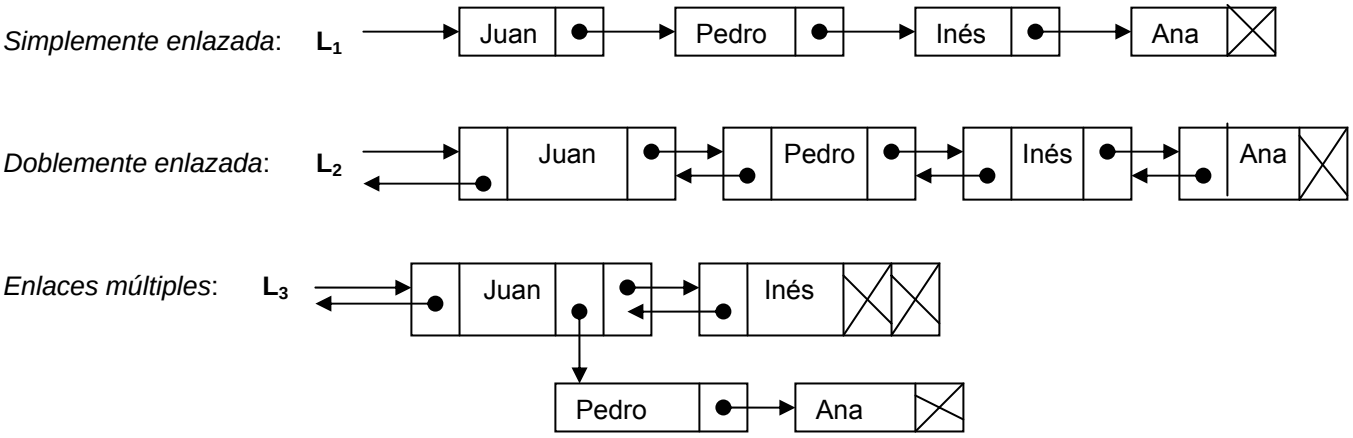
Nombre de Lista = ( $e_1$, $e_2$, ..., $e_n$ )

Siendo $e_1$, $e_2$, ..., $e_n$ los elementos de la lista.

Puesto que los componentes de una lista se almacenan según cierto orden, es muy importante notar que:

$\mathbf{L_1}$ = ( $e_1$, $e_2$, ..., $e_n$ ) es una lista **distinta** de $\mathbf{L_2}$ = ( $e_n$, $e_1$, ..., $e_2$ )

De forma gráfica:




En estas representaciones las flechas indican los enlaces entre los componentes de las listas. El punto negro de la flecha indica el elemento origen y la punta de flecha el elemento destino, de tal forma que el origen conoce el destino, pero el destino no conoce su elemento origen. Por esta razón es por lo que las listas doblemente enlazadas precisan "dos" enlaces, uno en cada sentido. En la lista $\mathbf{L_1}$ el elemento de contenido "Juan" sabe que le sigue el elemento "Pedro", y este sabe que le sigue "Inés", pero "Pedro" no sabe que le precede "Juan":

El símbolo **X** o aspa que aparece en algunos elementos indican que no tienen enlace a ningún sitio, se entiende que es un enlace nulo. Obsérvese que todas las listas tienen que tener un "punto de entrada", es decir, una referencia al primer elemento de la lista. Estas referencias, en nuestro ejemplo, son $\mathbf{L_1}$, $\mathbf{L_2}$ y $\mathbf{L_3}$ .

También conviene fijarse de forma especial en las listas con enlaces múltiples. Mientras que en las listas simples y dobles únicamente hay una forma de "recorrer" la lista, desde el primer elemento hasta el último, en las listas con múltiples enlaces surgen varias formas de recorrerlas, según nos decidamos a avanzar por uno de los múltiples enlaces a otras *sublistas* que puede tener cada elemento.

Realmente este tipo de listas son un anticipo, o una forma distinta de ver otro tipo de TAD que estudiaremos en un próximo apartado, los TAD de tipo árbol. E incluso veremos con posterioridad que tanto las listas como los árboles realmente son casos particulares de un tipo más general de TAD, los *grafos*.

### 1.3.4 IMPLANTACIÓN

Las listas son un tipo de TAD cuya implantación puede realizarse de múltiples maneras. Según el tamaño, manejo o rendimiento que queramos que tengan ciertas operaciones, como búsquedas, ordenaciones, inserciones, etc, las listas pueden implementarse con arrays, ficheros secuenciales o punteros. Estructuras de datos todas ellas muy comunes en la mayoría de los lenguajes de programación.

También conviene observar que para implantar listas puede usarse tanto memoria estática (arrays) como memoria dinámica (punteros). Si bien es poco frecuente usar los arrays, pues es esencia son tabla. Para implementar una lista simple usando tablas hace falta recurrir a algún tipo de "truco", por ejemplo, almacenar en el contenido de los elementos del array dos datos, el contenido de los elementos de la lista y el "puntero" al siguiente elemento.

Veamos un ejemplo:

| **ARRAY (TABLA)** | | | | |
|---|---|---|---|---|
| **ÍNDICE** del array | ***0*** | ***1*** | ***2*** | ***3*** |
| **VALOR** del elemento | Juan#2 | Inés#3 | Pedro#1 | Ana#(-1) |

*Figura 4. Lista $L_1$ implementada con un array.*

Nótese como junto al nombre de las personas, usamos un carácter especial (#) para separar los nombre de los valores numéricos que hacen de puntero, indicando que índice de la tabla contiene el siguiente elemento de la lista. Para indicar el valor nulo usamos un valor de índice imposible, en este caso el negativo (-1). Hay que sobrentender que el puntero de comienzo de la lista es siempre el índice cero del array.

Si se eliminasen algunos elementos de la lista, por ejemplo a Inés y a Pedro, supondría reorganizar los apuntadores, de forma que tendríamos: Juan#3; Inés# (-1); Pedro# (-1); Ana#(-1). Desde un punto de vista abstracto la lista sólo tiene ahora dos elemento, Juan y Ana, sin embargo esto no quiere decir que el array haya disminuido su tamaño al dejar de apuntar a los elementos Inés y Pedro; simplemente estamos diciendo que hemos dejado de apuntarlos, pero el array sigue teniendo el mismo espacio reservado en memoria, y este espacio no está disponible para otro uso más que para almacenar los elementos del array. Esto es debido a que un array se implementa con memoria estática.

Usando memoria dinámica (punteros) no se tendría esta desventaja. Bajo esta circunstancia, si se elimina de la lista un elemento, el espacio de memoria que ocupaba queda disponible para ser usado para otras finalidades.

### 1.4 ÁRBOLES

Como dice David Harel en su libro “The Spirit of Computing”:

“*Una de las estructuras de datos más importantes y prominentes que existen es el **árbol**. No es un árbol en el sentido botánico de la palabra, sino uno de naturaleza más abstracta. Todos hemos visto usar tales árboles para describir conexiones familiares. Los dos tipos más comunes de árboles familiares son el -árbol de antecesores-, que empieza en un individuo y va hacia atrás a través de padres, abuelos, etc., y el -árbol de descendientes-, que va hacia delante a través de hijos, nietos, etc*.”

Un *árbol* es un TAD cuya estructura corresponde con el siguiente gráfico:

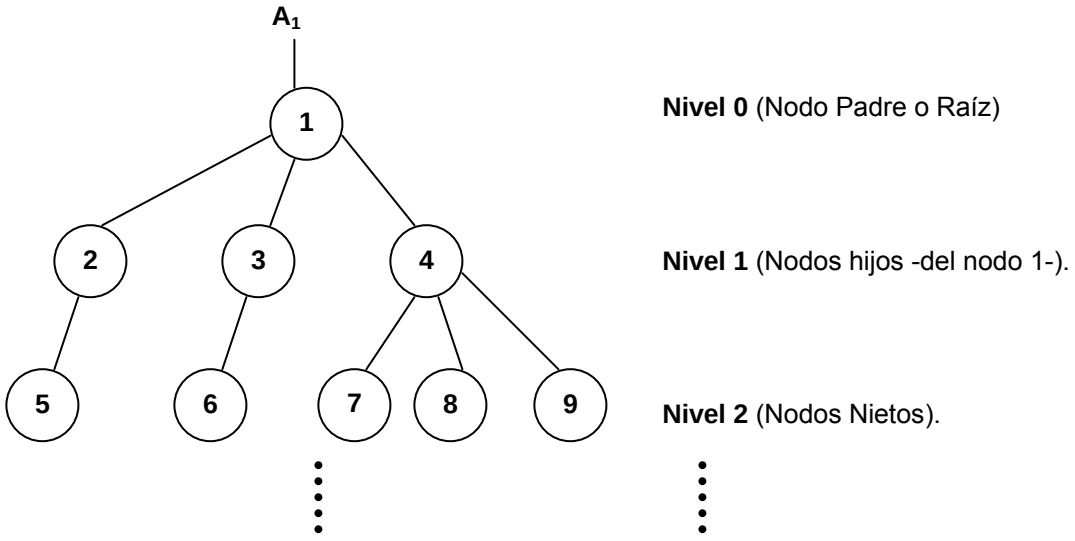



*Figura 5. Gráfico representativo de un árbol genérico.*

En donde podemos ya notar las principales características de los árboles. Cada uno de las diferentes "circunferencias" representa un elemento o nodo del TAD tipo árbol. El nodo 1 suele ser llamado *nodo de nivel 0* (también nodo raíz por su similitud con un árbol real). Es importante percatarse de que sólo puede haber un nodo de nivel 0 en cada árbol.

Los nodos 2, 3 y 4 son llamados nodos de nivel 1, por estar todos ellos al mismo "nivel" al ser todos "hijos" del nodo 1. Este razonamiento se puede ir aplicándose a los sucesivos niveles, hablándose de nodos nietos (o hijos si nos referimos al nivel inmediatamente superior).

Por la similitud con un árbol en el sentido botánico, también se dice que los nodos que no tiene hijos son nodos hoja y los nodos que sí tienen hijos, salvo el raíz, son llamados nodos rama o nodos internos. Con el ejemplo del árbol de la figura 5 ($\mathbf{A_1}$), mostramos los principales conceptos usados al hablar de árboles:

- ***Antecesor Directo*** *o* ***Padre***: El nodo 4 es el Antecesor Directo o Padre de los nodos 7, 8 y 9.
- ***Antecesor*** *o* ***Ancestro***: El nodo 1 es el Antecesor de todos los otros nodos.
- ***Sucesor Directo*** *o* ***Hijo***: El nodo 5 es el Sucesor Directo o Hijo del nodo 2.
- ***Sucesor*** *o* ***Descendiente***: Todos los nodos son Sucesores o Descendientes del nodo 1.
- ***Nodo de Nivel Cero*** *o* ***Raíz***: En nuestro ejemplo es el nodo 1.
- ***Nodo Interno*** *o* ***Rama***: En nuestro ejemplo son los nodos 2, 3 y 4.
- ***Nodo Hoja***: En nuestro ejemplo son los nodos 5, 6, 7, 8 y 9.
- ***Nivel de un Nodo***: El nodo 1 tiene nivel 0. Los nodos 2, 3 y 4 nivel 1. Los nodos 5, 6, 7, 8 y 9 nivel 2.
- ***Grado de un Nodo***: Es el número de descendientes directos que tiene un nodo. El grado del nodo 2 es 1.
- ***Grado del Árbol***: Es el mayor de los grados de los nodos que los componen. En nuestro caso es 3.
- ***Altura del Árbol***: Es el mayor de los niveles del árbol. En nuestro caso es 2.
- ***Longitud de Camino de un Nodo***: Número de enlaces o arcos que hay que atravesar para ir de la raíz a un nodo. La longitud de camino del nodo raíz es la unidad. Por ejemplo, el nodo 5 tiene longitud de camino 3.

### 1.4.1 TIPOS

En función de la estructura que define un árbol, se pueden establecer distintos tipos de árboles atendiendo a ciertos aspectos en la *forma* del árbol. Según esto es muy corriente hablar de los siguientes tipos de árboles:

- Árbol **BINARIO** o árbol **B**: Es el árbol cuyo máximo nº de nodos hijos que tiene cualquier nodo es dos.

  Un ejemplo sería:

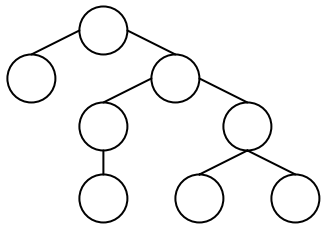


*Figura 6. Árbol Binario.*

- Árbol **MULTIRAMA**: Son aquellos en los que no hay límite para el número de nodos hijo. El número de niveles no crece tanto como en los binarios. Es el caso genérico de árbol. Por ejemplo $\mathbf{A_1}$.
- Árbol **BALANCEADO**: Son aquellos árboles en los cuales se cumple que *entre todos sus nodos hoja no hay una diferencia de nivel superior a la unidad*. Veamos algunos ejemplos de árboles binarios balanceados (también denominados *árboles AVL*) y no-balanceados.

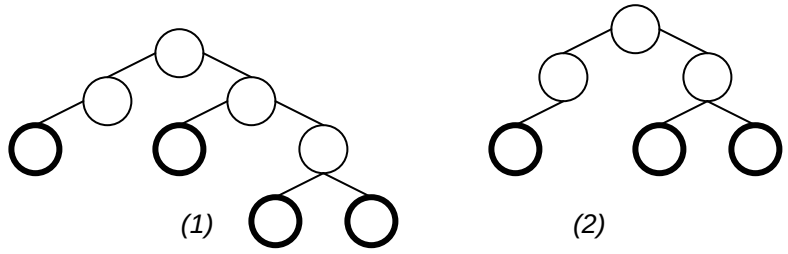



*Figura 7. Árboles Balanceados.*

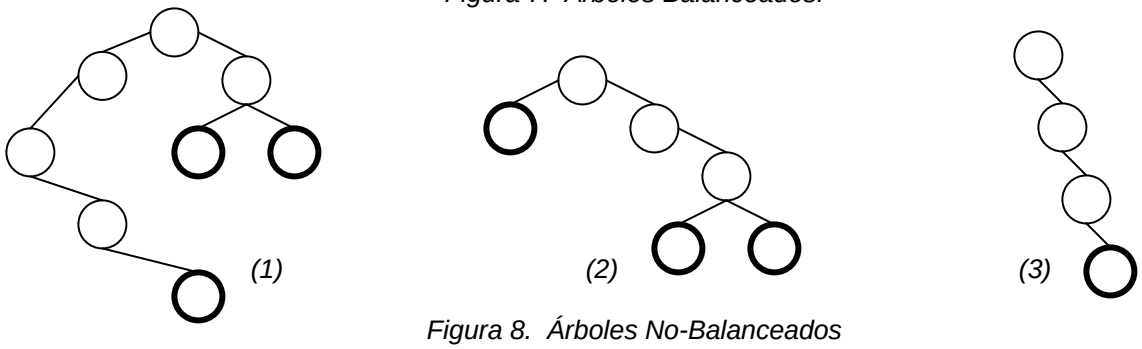



*Figura 8. Árboles No-Balanceados*

Es de observar que una lista realmente es un árbol en el que cada nodo únicamente tiene un hijo, y se considera al primer nodo de la lista como el nodo raíz del árbol (tal como se muestra en el árbol (3) de la figura 8.

Una lista de más de dos elementos se corresponde a la estructura de una árbol con un grado de no-balanceo máximo (sólo hay un nodo hoja). Se suele decir en estos caso que la lista es un *árbol totalmente degenerado*.

### 1.4.2 OPERACIONES

Las operaciones básicas que se pueden realizar sobre los árboles son:

- ***Definir/crear*** el árbol: Se refiere a la forma que cada lenguaje debe tener para definir la estructura de tipo árbol y crear una variable de este tipo.
- ***Recorrer*** los diferentes caminos del árbol: Esta es una operación que resulta muy importante en los árboles, pues es sobre la que se apoyan las demás operaciones. Vamos a ver que existen dos formas de recorrer un árbol, en ambas, por convenio, se asume que siempre nos desplazaremos de *izquierda a derecha* en horizontal, al igual que la lectura de un libro. Respecto a la dimensión vertical, nos moveremos de arriba hacia abajo o viceversa.

  Existen dos formas de recorrer un árbol, a saber:

  - Recorrido en ***AMPLITUD***: Se trata de recorrer consecutivamente los nodos que se encuentran en el mismo nivel (siempre de izquierda a derecha). La dimensión que prima en el recorrido es la horizontal. Tomando como ejemplo el árbol $\mathbf{A_1}$ de la figura 5, el recorrido en amplitud nos daría la siguiente secuencia de nodos: (1,2,3,4,5,6,7,8,9).
  - Recorrido en ***PROFUNDIDAD***: La dimensión que prima al recorrer el árbol el la *vertical*. Siempre se tenderá a alejarse del nodo raíz mientras se pueda. Sin embargo, a diferencia del recorrido en amplitud, existen diferentes formas de recorrido en profundidad; estas formas son:

    - Profundidad en *PRE-ORDEN*: Se empieza por el nodo raíz y se tiende a alejarse lo máximo posible de él. Moviéndose en vertical de arriba hacia abajo. En nuestro ejemplo obtendríamos la secuencia:

      (1,2,5,3,6,4,7,8,9).

    - Profundidad en *POST-ORDEN*: Se comienza por el nodo hoja más a la izquierda y se mantiene la máxima distancia que se pueda con el nodo raíz. Para el árbol $\mathbf{A_1}$ se obtiene la secuencia:

(5,2,6,3,7,8,9,4,1).

- Profundidad en *ORDEN-CENTRAL*: Como el recorrido post-orden pero cuidando que no aparezcan, siempre que se posible, dos nodos del mismo padre (nodos hermanos) de forma consecutiva en la secuencia de recorrido. En nuestro caso se tendría:

  (5,2,1,6,3,7,4,8,9). (Obsérvese que únicamente el 8 y el 9 aparecen consecutivos).

  Para los árboles binarios, en este tipo de recorrido jamás pueden aparecer consecutivos dos nodos hermanos.

□ ***Insertar/eliminar*** elementos en el árbol: Al igual que en las listas, hay que distinguir entre insertar el nodo raíz, un nodo interno o un nodo hoja. También resulta muy importante nunca cambiar de forma incorrecta o perder un enlace, pues supondría la imposibilidad de acceder a parte o todo el árbol.

□ ***Ordenar*** los elementos del árbol: Ordenar un árbol no implica cambiar su estructura, sino modificar el contenido de los nodos para que sigan algún tipo de orden deseado. Las posibles ordenaciones de un árbol son aquellas que hacen que al hacer el recorrido de un árbol se obtenga una secuencia ordenada según algún criterio. Por ejemplo, el árbol de la figura 5 está ordenado en amplitud, pues al hacer un recorrido de este tipo se obtiene la secuencia ordenada (1,2,3,4,5,6,7,8,9).

Como ejemplo de ordenación, el árbol $\mathbf{A_1}$ ordenado en profundidad pre-orden sería:

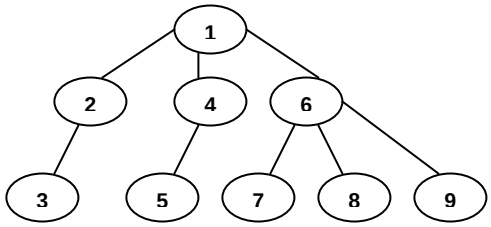



*Figura 9. Árbol ordenado en pre-orden. Secuencia (1,2,3,4,5,6,7,8,9).*

□ ***Buscar*** elementos en el árbol: Consiste en localizar el/los nodo/s que contiene un dato igual al dato dado como referencia. Es muy importante darse cuenta de que según esté el árbol ordenado o no, si está balanceado o no, y dependiendo del recorrido que se haga, las búsquedas pueden ser más o menos efectivas (rápidas).

□ ***Contar*** los elementos de la tabla: Básicamente consiste en recorrer el árbol en cualquiera de sus formas e ir contando los elemento. Existen sin embargo variantes de esta contabilidad. Por ejemplo, podría solicitarse el número de nodos que tiene el nivel N de un árbol, lo cual exigiría un recorrido en amplitud teniendo en cuenta siempre en nivel donde se está en cada momento. Otra posible cuenta sería conocer el número de nodos que hay desde la raíz hasta el nodo hoja de menor nivel. Etc.

□ ***Balancear*** el árbol: Consiste en hacer que un árbol no-balanceado pase a ser balanceado. Esta operación sí que supone un cambio en la estructura del árbol, suponiendo siempre el cambio de nodo raíz y la variación de niveles de ciertos nodos. Veamos un ejemplo de balanceo de un árbol binario.

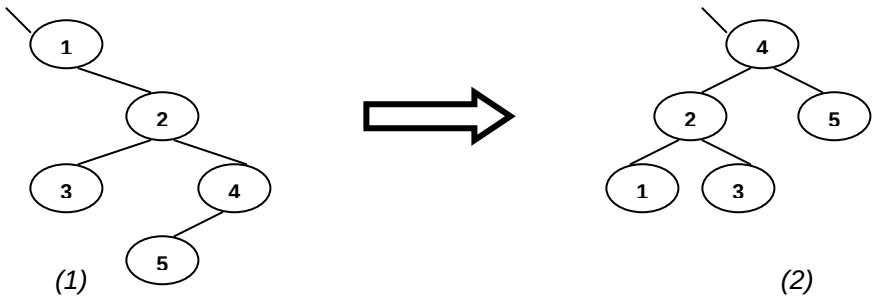



*Figura 10. Balanceo de un árbol binario.*

### 1.4.3 REPRESENTACIÓN

Ya hemos visto una primera forma de representar un árbol muy intuitiva y clarificadora, usando circunferencias para representar los nodos o elementos del árbol y líneas para representar los enlaces.

No obstante, sin tocar en absoluto la estructura del árbol, conviene considerar la forma de enlazar los nodos del árbol, pues al igual que con las listas, los nodos del árbol pueden estar simplemente enlazados o doblemente enlazados. Esta diferencia afecta en la facilidad o dificultad de implementar ciertas operaciones sobre los árboles, sobre todo las diferentes operaciones de recorrido del árbol. Cuando se quiere especificar en un gráfico el tipo de enlaces del árbol, se utiliza una forma parecida a las listas, veamos un ejemplo de un mismo árbol simplemente enlazado y doblemente enlazado.

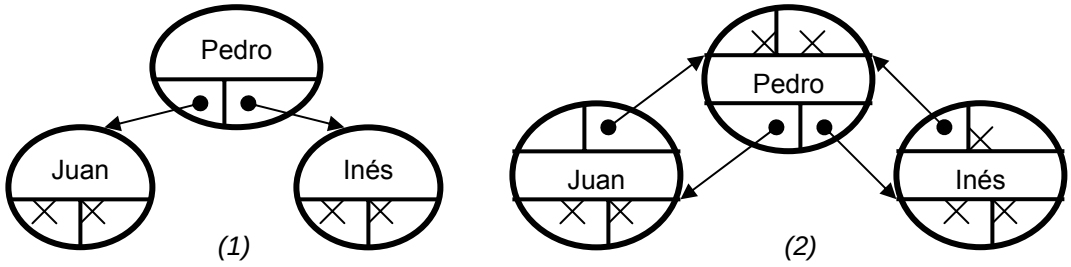



*Figura 11. Mismo árbol simplemente y doblemente enlazado.*

### 1.4.4 IMPLANTACIÓN

Al igual que las lista, con los árboles, según diversas consideraciones sobre el tamaño, manejo o rendimiento que queramos que tengan ciertas operaciones, tales como recorridos, búsquedas, ordenaciones, inserciones, etc. los árboles puede implantarse con arrays, ficheros secuenciales o punteros. Estructuras de datos todas ellas muy comunes en la mayoría de los lenguajes de programación.

Lo más habitual es que se utilice memoria dinámica, es decir, punteros, para la implantación de todo tipo de árboles, aunque en ciertas ocasiones pueden considerarse la implantación con memoria estática mediante arrays.

## 2. ALGORITMOS: ORDENACIÓN, BÚSQUEDA, RECURSIÓN, GRAFOS

La búsqueda de un elemento es, con diferencia, la operación más utilizada en los TAD, y la más importante. Por otra parte, la ordenación de los elementos del TAD puede resultar de gran ayuda a la hora de realizar búsquedas, así pues, la ordenación de los elementos del TAD es la segunda operación más útil. Se pueden distinguir dos tipos dentro de estos algoritmos:

- Algoritmos **Recursivos**: En un apartado posterior examinaremos la recursión. Básicamente los algoritmos recursivos son aquellos que se basan en el uso a rutinas de programación que se llaman a sí mismas.
- Algoritmos **Iterativos**: Se consideran algoritmos iterativos a todos aquellos que no son recursivos.

### 2.1 ORDENACIÓN

La finalidad de los algoritmos de ordenación es organizar ciertos datos (estos datos pueden estar contenidos en diferentes estructuras, ya sean TAD o en ficheros) en un orden creciente o decreciente mediante una regla prefijada (numérica, alfabética,...).

Atendiendo al tipo de elemento que se quiera ordenar puede ser:

- ***Ordenación interna***: Los datos se encuentran en memoria (ya sean tablas, listas, árboles, etc.).

- ***Ordenación externa***: Los datos están en un dispositivo de almacenamiento externo (ficheros), y su ordenación es más lenta que la interna.

En este apartado vamos a estudiar los métodos de ordenación interna para los TAD vistos en apartados anteriores, esto es, para ordenar tablas, listas y árboles. Inicialmente nos centraremos en los TAD tabla y lista (simple o doble) y más adelante comentaremos el caso de listas múltiples y árboles.

Además, aquí consideraremos métodos de ordenación iterativa, es decir, no-recursivos. Cuando estudiemos la recursividad veremos ejemplos de ordenaciones de tipo recursivo.

Los principales algoritmos de ordenación interna de tipo iterativo son:

- Selección.
- Burbuja.
- Inserción Directa.
- Inserción Binaria.
- Shell.
- Intercalación.

### 2.1.1 SELECCIÓN

Este método consiste en buscar el elemento más pequeño del TAD y ponerlo en la primera posición; luego, entre los restantes, se busca el elemento más pequeño y se coloca en segundo lugar, y así sucesivamente hasta colocar el último elemento.

La idea de esta ordenación es independiente del tipo de TAD a la que se aplique, lo único puede variar son ciertos aspectos o detalles en la forma explícita de hacer la ordenación según se implemente el TAD con memoria estática (arrays) o memoria dinámica (punteros).

Por ejemplo, si tenemos el array {40,21,4,9,10,35}, o la lista (40,21,4,9,10,35), los pasos a seguir son:

{4,21,40,9,10,35} ← Se coloca el 4, el más pequeño, en primera posición: se cambia el 4 por el 40.
{4,9,40,21,10,35} ← Se coloca el 9, en segunda posición: se cambia el 9 por el 21.
{4,9,10,21,40,35} ← Se coloca el 10, en tercera posición: se cambia el 10 por el 40.
{4,9,10,21,40,35} ← Se coloca el 21, en tercera posición: ya está colocado.
{4,9,10,21,35,40} ← Se coloca el 35, en tercera posición: se cambia el 35 por el 40.

Como se tienen N elementos, el número de comprobaciones que hay que hacer es de N*(N-1)/2, luego el coste del algoritmo es $O(N^2)$

### 2.1.2 BURBUJA

Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Utilizando como ejemplo la tabla anterior {40,21,4,9,10,35}:

Primera pasada: Se comienza por el primer elemento.
{21,40,4,9,10,35} ← Se cambia el 21 por el 40.
{21,4,40,9,10,35} ← Se cambia el 40 por el 4.
{21,4,9,40,10,35} ← Se cambia el 9 por el 40.
{21,4,9,10,40,35} ← Se cambia el 40 por el 10.
{21,4,9,10,35,40} ← Se cambia el 35 por el 40.

Segunda pasada: Se comienza por el segundo elemento.
{4,21,9,10,35,40} ← Se cambia el 21 por el 4.
{4,9,21,10,35,40} ← Se cambia el 9 por el 21.
{4,9,10,21,35,40} ← Se cambia el 21 por el 10.

Llegado a este punto ya están ordenados, pero el algoritmo realmente acaba cuando se comienza el recorrido a partir del penúltimo elemento. Por tanto, si hay N elementos, para estar seguro de que el array está ordenado, hay que hacer N-1 pasadas, por lo que habría que hacer (N-1)*(N-i-1) comparaciones, para cada i desde 1 hasta N-1. El número de comparaciones es, por tanto, N(N-1)/2, lo que nos deja un coste del algoritmo, al igual que en la selección, de $O(N^2)$.

### 2.1.3 INSERCIÓN DIRECTA

En este método lo que se hace es tener una sublista ordenada de elementos de la lista, o array, e ir insertando el resto en el lugar adecuado para que la sublista no pierda el orden. La sublista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada.

Para el ejemplo {40,21,4,9,10,35}, se tiene:

{40,21,4,9,10,35} ← La primera sublista ordenada es {40}.

Insertamos el 21:
{40,40,4,9,10,35} ← Usamos una variable auxiliar aux=21;
{21,40,4,9,10,35} ← Ahora la sublista ordenada es {21,40}.

Insertamos el 4:
{21,40,40,9,10,35} ← aux=4;
{21,21,40,9,10,35} ← aux=4;
{4,21,40,9,10,35} ← Ahora la sublista ordenada es {4,21,40}.

Insertamos el 9:
{4,21,40,40,10,35} ← aux=9;
{4,21,21,40,10,35} ← aux=9;
{4,9,21,40,10,35} ← Ahora la sublista ordenada es {4,9,21,40}.

Insertamos el 10:
{4,9,21,40,40,35} ← aux=10;
{4,9,21,21,40,35} ← aux=10;
{4,9,10,21,40,35} ← Ahora la sublista ordenada es {4,9,10,21,40}.

Y por último insertamos el 35:
{4,9,10,21,40,40} ← aux=35;
{4,9,10,21,35,40} ← Ya está ordenado.

En el peor de los casos, el número de comparaciones que hay que realizar es de N*(N+1)/2-1, lo que nos deja un tiempo de ejecución en $O(N^2)$. En el mejor caso (cuando la lista ya estaba ordenada), el número de comparaciones es (N-2), y todas ellas son falsas, con lo que no se produce ningún intercambio. El tiempo de ejecución está en O(N).

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Vemos que cuanto más ordenada esté inicialmente más se acerca a O(N) y cuanto más desordenada, más se acerca a $O(N^2)$. Este es un caso de algoritmo donde hay una gran diferencia entre el mejor caso y el peor caso. El peor caso es igual que en los métodos de burbuja y selección, pero el mejor caso es lineal, algo que no ocurría en éstos, con lo que para ciertas entradas podemos tener ahorros en tiempo de ejecución.

### 2.1.4 INSERCIÓN BINARIA

Es el mismo método que la inserción directa, excepto que la búsqueda del orden de un elemento en la sublista ordenada se realiza mediante una búsqueda binaria (veremos este tipo de búsqueda mas adelante), lo que en principio supone un ahorro de tiempo. No obstante, dado que para la inserción sigue siendo necesario un desplazamiento de los elementos, el ahorro, en la mayoría de los casos, no se produce, si bien hay compiladores que realizan optimizaciones que lo hacen ligeramente más rápido.

### 2.1.5 SHELL

Es una mejora del método de inserción directa, utilizado cuando la tabla tiene un gran número de elementos. En este método no se compara a cada elemento con el de su izquierda, como en el de inserción, sino con el que está a un cierto número de lugares (llamado salto) a su izquierda. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Por ejemplo, lo pasos para ordenar el array {40,21,4,9,10,35} mediante el método de Shell serían:
Salto=3:

Primera pasada:
{9,21,4,40,10,35} ← Se intercambian el 40 y el 9.
{9,10,4,40,21,35} ← Se intercambian el 21 y el 10.

Salto=1:

Primera pasada:
{9,4,10,40,21,35} ← Se intercambian el 10 y el 4.
{9,4,10,21,40,35} ← Se intercambian el 40 y el 21.
{9,4,10,21,35,40} ← Se intercambian el 35 y el 40.

Segunda pasada:
{4,9,10,21,35,40} ← Se intercambian el 4 y el 9.

Con sólo 6 intercambios se ha ordenado el array, cuando por inserción se necesitaban muchos más.

### 2.1.6 INTERCALACIÓN

No es propiamente un método de ordenación, consiste en la unión de dos tablas o listas ordenadas de modo que la unión esté también ordenada. Para ello, basta con recorrer los TAD de izquierda a derecha e ir cogiendo el menor de los dos elementos, de forma que sólo aumenta el contador del array del que sale el elemento siguiente para el array-suma. Si quisiéramos sumar las tablas {1,2,4} y {3,5,6}, los pasos serían:

Inicialmente: i1=0, i2=0, is=0.

Primer elemento: mínimo entre 1 y 3 = 1. Suma={1}. i1=1, i2=0, is=1.
Segundo elemento: mínimo entre 2 y 3 = 2. Suma={1,2}. i1=2, i2=0, is=2.
Tercer elemento: mínimo entre 4 y 3 = 3. Suma={1,2,3}. i1=2, i2=1, is=3.
Cuarto elemento: mínimo entre 4 y 5 = 4. Suma={1,2,3,4}. i1=3, i2=1, is=4.

Como no quedan elementos del primer array, basta con poner los elementos que quedan del segundo array en la suma:

Suma = {1,2,3,4} + {5,6} = {1,2,3,4,5,6}

## 2.2 BÚSQUEDA

Como se ha mencionado anteriormente la búsqueda es la operación más importante en el procesamiento de la información, y permite la recuperación de datos previamente almacenados. El tipo de búsqueda se puede clasificar como interna o externa, según el lugar en el que esté almacenada la información (en memoria o en dispositivos externos). Todos los algoritmos de búsqueda tienen dos finalidades:

- Determinar si el elemento buscado se encuentra en el conjunto en el que se busca.
- Si el elemento está en el conjunto, hallar la posición en la que se encuentra.

En este apartado nos centramos en la búsqueda interna iterativa. Como principales algoritmos en tablas y listas tenemos las búsquedas:

- Secuencial.
- Binaria o Dicotómica.
- Utilizando tablas Hash.

### 2.2.1 BÚSQUEDA SECUENCIAL

Esta búsqueda consiste en recorrer y examinar cada uno de los elementos hasta encontrar el o los elementos buscados, o hasta que se han mirado todos los elementos.

Por ejemplo, para la lista (10,12,4,10,9); donde queremos encontrar el elemento cuyo contenido es 10, la idea del algoritmo sería:

Primer paso:

( **10**, 12, 4, 10, 9 )
↑ **Encontrado.**

Segundo paso:

( 10, 12, 4, 10, 9 )
↑ No encontrado.

Tercer paso:

( 10, 12, 4, 10, 9 )
↑ No encontrado.

Cuarto paso:

( 10, 12, 4, **10**, 9 )
↑ **Encontrado.**

Quinto y último paso:

( 10, 12, 4, 10, 9 )
↑ No encontrado.

Puesto que queremos encontrar todas las ocurrencias del valor 10, es necesario recorrer siempre toda la lista o tabla, así pues habrá recorrer los N elemento, lo cual hace que el coste del algoritmo sea N, y su orden O(N).

Si sólo queremos encontrar la primera ocurrencia que se produzca, o tenemos la certeza de que no puede haber elementos repetidos, se puede parar la búsqueda en cuanto se produzca la primera ocurrencia, con lo cual el algoritmo es más eficiente. En este caso, el número medio de comparaciones que hay que hacer antes de encontrar el elemento buscado es de (N+1)/2. Aún así, el orden sigue siendo O(N).

### 2.2.2 BÚSQUEDA BINARIA O DICOTÓMICA

Para utilizar este algoritmo, se precisa que la tabla o lista considerada esté ordenada. La búsqueda binaria consiste en dividir la tabla por su elemento medio en dos subtablas más pequeñas, y comparar el elemento con el del centro. Si coinciden, la búsqueda se termina. Si el elemento es menor, debe estar (si está) en el primera subtabla, y si es mayor está en la segunda.

Por ejemplo, para buscar el elemento 3 en {1,2,3,4,5,6,7,8,9} se realizarían los siguientes pasos:

Se toma el elemento central y se divide el array en dos:
{1,2,3,4}-5-{6,7,8,9}
Como el elemento buscado (3) es menor que el central (5), debe estar en el primer subtabla: {1,2,3,4}

Se vuelve a dividir el array en dos:
{1}-2-{3,4}
Como el elemento buscado es mayor que el central, debe estar en el segundo subtabla: {3,4}

Se vuelve a dividir en dos:
{}-3-{4}
Como el elemento buscado coincide con el central, lo hemos encontrado.

Si al final de la búsqueda todavía no lo hemos encontrado, y la subtabla a dividir está vacía {}, quiere decir que el elemento no se encuentra en la tabla.

En general, este método realiza $\{\log_2 (N+1)\}$ comparaciones antes de encontrar el elemento, o antes de descubrir que no está. Este número es muy inferior que el necesario para la búsqueda lineal para casos grandes.

Este método también se puede implementar de forma recursiva, siendo la función recursiva la que divide la tabla o lista en más pequeñas.

### 2.2.3 BÚSQUEDA UTILIZANDO TABLAS HASH

Este método no es realmente un método de búsqueda, sino una forma de mejorar la velocidad de búsqueda al utilizar algún otro método.

Consiste en asignar a cada elemento un índice mediante una transformación del elemento. Esta correspondencia se realiza mediante una función de conversión, llamada función hash. La correspondencia más sencilla es la identidad, esto es, al número 0 se le asigna el índice 0, al elemento 1 el índice 1, y así sucesivamente.

Pero si los números a almacenar son demasiado grandes esta función es inservible. Por ejemplo, se quiere guardar en un array la información de los 1000 usuarios de una empresa, y se elige el número de DNI como elemento identificativo. Es inviable hacer un array de 100.000.000 elementos, sobre todo porque se desaprovecha demasiado espacio. Por eso, se realiza una transformación al número de DNI para que nos de un número menor, de 3 cifras pues hay 1000 usuarios, y utilizar este resultado de la función hash como índice de un array de 1000 elementos para guardar a los empleados.

Así, dado un DNI a buscar, bastaría con realizar la transformación según la función hash a un número de 3 cifras; usar este número como índice de búsqueda del array de 1000 elementos, con cualquiera de los métodos anteriores, y obtener el contenido el array dado por la posición cuyo índice es el resultado de la función hash.

La función de hash ideal debería ser biyectiva, esto es, que a cada elemento le corresponda un índice, y que a cada índice le corresponda un elemento, pero no siempre es fácil encontrar esa función, e incluso a veces es inútil, ya que puede no saberse “a priori” el número de elementos a almacenar.

La función de hash depende de cada problema y de cada finalidad, y se pueden utilizar con números o cadenas, pero las más utilizadas son:

- *Restas sucesivas:* Esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas.
- *Aritmética modular:* El índice de un número es resto de la división de ese número entre un número N prefijado, preferentemente primo. Los números se guardarán en las direcciones de memoria de 0 a N-1. Este método tiene el problema de que cuando hay (N+1) elementos, al menos un índice es señalado por dos elementos (teorema del palomar). A este fenómeno se le llama colisión.
- *Mitad del cuadrado:* Consiste en elevar al cuadrado la clave y coger las cifras centrales. Este método también presenta problemas de colisión.
- *Truncamiento:* Consiste en ignorar parte del número y utilizar los elementos restantes como índice. También se produce colisión.
- *Plegamiento:* consiste en dividir el número en diferentes partes, y operar con ellas (normalmente con suma o multiplicación). También se produce colisión.

Ahora se nos presenta el problema de qué hacer con las colisiones, es decir, el ***Tratamiento de Colisiones***. ¿Qué pasa cuando a dos elementos diferentes les corresponde el mismo índice?. Pues bien, hay tres posibles soluciones:

- Cuando el índice correspondiente a un elemento ya está ocupado, se le asigna el primer índice libre a partir de esa posición. Este método es poco eficaz, porque al nuevo elemento se le asigna un índice que podrá estar ocupado por un elemento posterior a él, y la búsqueda se ralentiza, ya que no se sabe la posición exacta del elemento.
- También se pueden reservar unos cuantos lugares al final del array para alojar a las colisiones. Este método también tiene un problema: ¿Cuánto espacio se debe reservar? Además, sigue la lentitud de búsqueda si el elemento a buscar es una colisión.
- Lo mejor es en vez de crear un array de números, crear un array de punteros, donde cada puntero señala el principio de una lista enlazada. Así, cada elemento que llega a un determinado índice se pone en el último lugar de la lista de ese índice. El tiempo de búsqueda se reduce considerablemente, y no hace falta poner restricciones al tamaño del array, ya que se pueden añadir nodos dinámicamente a la lista.

## 2.3 RECURSIVIDAD

### 2.3.1 DEFINICIÓN DE RECURSIVIDAD

La ***recursividad*** es una característica que permite que una determinada acción se pueda realizar en función de invocar la misma acción pero en un caso mas sencillo, hasta llegar a un punto (caso base) donde la realización de la acción sea muy sencilla. Esta forma de ver las cosas es útil para resolver problemas definibles en sus propios términos. En cierta medida, es análogo al principio de inducción.

Para desarrollar algoritmos recursivos hay que partir del supuesto de que ya hay un algoritmo que resuelve una versión más sencilla del problema. A partir de esta suposición debe hacerse lo siguiente:

- Identificar subproblemas atómicos de resolución inmediata. Los denominados *casos base*.
- Descomponer el problema en subproblemas resolubles mediante el algoritmo pre-existente; la solución de estos subproblemas debe aproximarnos a los casos base.

No existen problemas intrínsecamente recursivos o iterativos; cualquier proceso iterativo puede expresarse de forma recursiva y viceversa. Si bien ciertos problemas se prestan mucho mejor que otros al uso de la recursividad.

En el mundo de las matemáticas, un clásico ejemplo de recursividad lo tenemos en el cálculo del factorial de un número, a saber:

Factorial del número natural n (incluido el 0) = n!

(1) si n = 0 entonces: 0! = 1
(2) si n > 0 entonces: n! = n · (n-1)!

Aunque en este estudio de los TAD hemos evitado el uso de lenguajes de programación específicos, por ser una de las características de los tipos abstractos de datos la independencia de su implantación; en esta ocasión recurriremos al lenguaje C para poner una ejemplo muy claro e ilustrativo de cómo implementar un algoritmo recursivo. El siguiente programa es un ejemplo del cálculo del factorial de un número n.

```
int factorial(int n)
{
  if (n == 0) return 1;              // Caso base.
  return (n * factorial(n-1));       // Llamada a si mismo.
}
```

La función factorial es llamada pasándole un determinado entero y devuelve otro numero entero. Como se observa, en cada llamada recursiva se reduce el valor de n, llegando el caso en el que n es 0 y no efectúa más llamadas recursivas.

Aunque la función factorial se preste muy bien al uso de la recursividad, no quiere decir esto, tal como hemos mencionado anteriormente, que no pueda ser implementado de forma iterativa. De hecho el factorial puede obtenerse con facilidad sin necesidad de emplear funciones recursivas, es más, el uso del programa anterior es muy ineficiente (con un número n grande, al ejecutarse en una computadora, consumiría mucha más memoria y tiempo que si se usase un algoritmo iterativo), pero es un ejemplo muy claro.

### 2.3.2 USO DE LA RECURSIÓN

La pregunta que surge de manera natural es: *¿Cuándo utilizar la recursividad?*. No se debe utilizar si la solución iterativa es sencilla y clara. En otros casos, obtener una solución iterativa es mucho más complicado que una solución recursiva, y entonces se planteará si merece la pena transformar la solución recursiva en una iterativa.

Por otra parte, hay TAD que, debido a sus características, sus operaciones se adaptan muy bien al uso de la recursión. Casi todos los algoritmos basados en los esquemas de vuelta atrás y divide y vencerás son recursivos. Otras estructuras que se adaptan muy bien a la recursividad son los grafos, y cuando se habla de grafos se está incluyendo a las listas y a los árboles.

Si nos centramos en el mundo de la programación, algunos lenguajes de programación no admiten el uso de recursividad. Es obvio que en este caso se requerirá una solución no recursiva (iterativa). Aunque parezca mentira, es en general mucho más sencillo escribir un programa recursivo que su equivalente iterativo. Y desde luego siempre resulta más económico en cantidad de líneas de código. Como prueba, el opositor puede intentar escribir el anterior algoritmo del cálculo del factorial de forma iterativa.

### 2.3.3 ORDENACIONES Y BÚSQUEDAS RECURSIVAS

Ya hemos visto en apartados anteriores como se realizan ordenaciones y búsquedas iterativas en TAD tipo tabla y lista. Veamos ahora algoritmos de naturaleza recursiva para realizar estas operaciones.

La ***Ordenación Rápida (Quicksort)*** es un algoritmo de ordenación ilustrativo del uso de la recursividad en ordenaciones. Este método se basa en la táctica "divide y vencerás", que consiste en ir subdividiendo el TAD considerado, en nuestro caso una tabla o lista en partes más pequeñas, y ordenar éstas.

Para hacer esta división, se toma un valor como pivote, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha. A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.

Normalmente se toma como pivote el primer elemento de array, y se realizan dos búsquedas: una de izquierda a derecha, buscando un elemento mayor que el pivote, y otra de derecha a izquierda, buscando un elemento menor

que el pivote. Cuando se han encontrado los dos, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran. Por ejemplo, para dividir el array {21,40,4,9,10,35}, los pasos serían:

{21,40,4,9,10,35} ← Se toma como pivote el 21. La búsqueda de izquierda a derecha encuentra el valor 40, mayor que pivote, y la búsqueda de derecha a izquierda encuentra el valor 10, menor que el pivote.

Se intercambian:
{21,10,4,9,40,35} ← Si seguimos la búsqueda, la primera encuentra el valor 40, y la segunda el valor 9, pero ya se han cruzado, así que paramos. Para terminar la división, se coloca el pivote en su lugar (en el número encontrado por la segunda búsqueda, el 9, quedando:
{9,10,4,21,40,35} ← Ahora tenemos dividido el array en dos arrays más pequeños: el {9,10,4} y el {40,35}, y se repetiría el mismo proceso.

Respecto a los ***árboles***, la idea fundamental que hace posible el uso de recursividad es que realmente un árbol puede considerarse como un nodo raíz del que cuelgan otros árboles, llamados subárboles. Véase la siguiente figura como ejemplo de esta idea en un árbol binario.

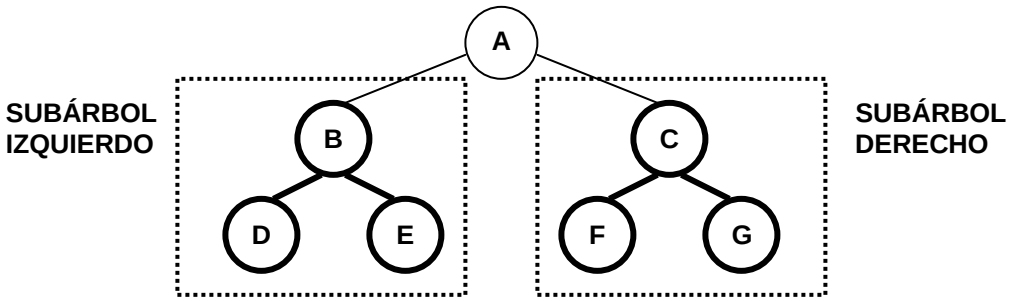



*Figura 12. Subárbol izquierdo y derecho de un árbol.*

Para realizar cualquier exploración, recorrido o búsqueda en un árbol binario (o cualquier árbol genérico) bastaría con aplicar la misma operación a cada uno de sus subárboles, hasta llegar al caso base, que sería la búsqueda o recorrido en un subárbol formado por el nodo raíz y nodos hojas, pues en este caso la búsqueda o recorrido es muy fácil, basta seguir los enlaces del nodo raíz para visitar los nodos hojas. Una vez hecho esto el algoritmo recursivo consiste en hacer lo mismo, esto es, seguir los enlaces del nodo raíz, pero tomando como nodo raíz el nodo padre de los nodos caso base. Se repete el proceso hasta llegar al nodo raíz del árbol completo.

En nuestro ejemplo, los casos bases se darían al llegar a los nodos B y C. Ambos nodos son casos base pues únicamente tienen nodos hojas. Una vez visto sus enlaces y sus nodos hojas se realizaría la misma operación pero tomando el nodo padre de B y C, el nodo A, y tratando a los nodos B y C como si fuesen hojas.

Veremos en el siguientes apartado sobre grafos que un TAD tipo árbol no es más que un tipo particular de grafo. Y con respecto a estos, comprobaremos que sus operaciones, entre las que está la búsqueda y la ordenación, son de índole profundamente recursiva.

## 2.4 GRAFOS

Un grafo es un objeto matemático que se utiliza para representar circuitos, redes, caminos, etc. Los grafos son muy utilizados en computación, ya que permiten resolver problemas muy complejos.

Supongamos el siguiente ejemplo. Disponemos de una serie de ciudades y de carreteras que las unen. De cada ciudad saldrán varias carreteras, por lo que para ir de una ciudad a otra se podrán tomar diversos caminos. Cada carretera tendrá un coste asociado (por ejemplo, la longitud de la misma). Gracias a la representación por grafos podremos elegir el camino más corto que conecta dos ciudades, determinar si es posible llegar de una ciudad a otra, si desde cualquier ciudad existe un camino que llegue a cualquier otra, etc. Para tener una idea visual de lo expuesto, supongamos que representamos las ciudades como circunferencias y los caminos por líneas que unen las distintas circunferencias (ciudades):

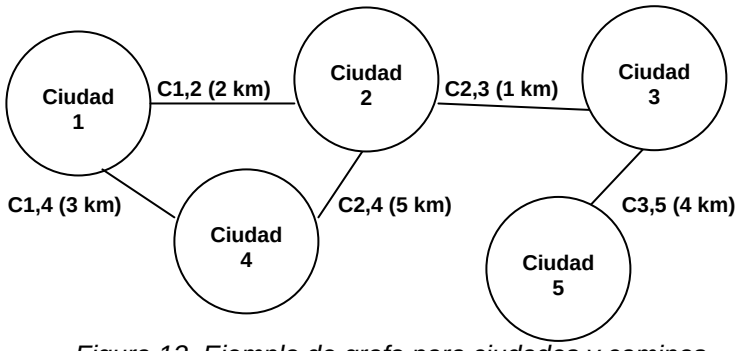



*Figura 13. Ejemplo de grafo para ciudades y caminos.*

Una primera cosa que salta a la vista es que tanto una lista como un árbol no son más que un caso particular de grafo, en donde hemos puesto ciertas restricciones.

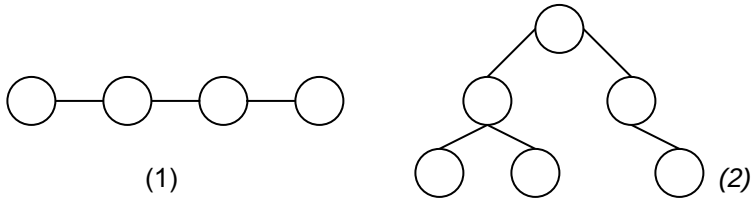



*Figura 14. Grafos correspondientes a una lista y un árbol binario.*

### 2.4.1 COMPONENTES DE UN GRAFO

Así pues, un grafo consta de:

- ***Vértices*** (o nodos): Los vértices son objetos que contienen información. Para representarlos se suelen utilizar puntos o circunferencias con el contenido del nodo escrito dentro. En nuestro ejemplo serían las circunferencias que representan las ciudades.
- ***Aristas***: Son las conexiones entre vértices. Para representarlos se utilizan líneas. Es frecuente añadir junto a la línea el nombre de los nodos origen y destino y el peso de la arista en algún tipo de unidad.

### 2.4.2 DEFINICIÓN FORMAL DE GRAFO

Conviene decir que la definición de un grafo no depende de su representación. Desde un punto de vista matemático puramente formal, la definición de grafo es la siguientes:

"*Un grafo G es un par (V(G), A(G)), donde V(G) es un conjunto no vacío de elementos llamados vértices, y A(G) es una familia finita de pares no ordenados de elementos de V(G) llamados aristas*".

Al ser una familia de aristas se permite la posibilidad de aristas múltiples en el grafo, es decir, la existencia de más de una arista con el mismo par de vértices como origen y destino. También se permite la existencia de aristas bucles, con inicio y destino el mismo vértice.

Por ejemplo, según lo dicho y utilizando la forma de representación indicada en el apartado anterior, se puede observar que el conjunto de vértices V(G) {a, b, c, d, e, f} y el de aristas A(G) formado por los pares {a, b}, {a, b}, {b, c}, {a, d}, {d, e}, {d, e}, {b, f}, {b, f}, {c, e}, {a, a}, {b, b} y {c, c} determinan el grafo de la figura siguiente.

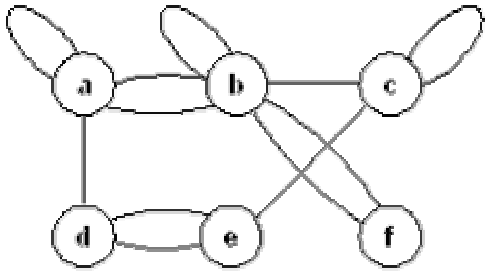



*Figura 15. Ejemplo de un grafo genérico.*

### 2.4.3 CONCEPTOS FUNDAMENTALES SOBRE GRAFOS

Algunos conceptos fundamentales al hablar de grafos son:

Un ***CAMINO*** o ***CIRCUITO*** entre dos vértices es una lista de vértices en la que dos elementos sucesivos están conectados por una arista del grafo.

Desde un punto de vista formal, cualquier secuencia finita de aristas de G de la forma (v0v1), (v1v2), (v2v3), ....,(vm-1vm) será denominada *camino* o *circuito* de G. Es frecuente que cuando entre dos vértices, por ejemplo los vértices “v” y “w”, existe uno o más caminos, se hable de cualquiera de estos caminos como un camino “vw”.

Se habla de ***GRAFO CONEXO*** si existe un camino desde cualquier nodo del grafo hasta cualquier otro. Si no es conexo constará de varias *componentes conexas*. Formalmente, un grafo G es *conexo* si para cualquier par de vértices "v", "w" de G existe una camino de "v" a "w" (“vw”).

Un ***PUENTE*** o ***ARISTA PUENTE*** será cualquier arista de un grafo conexo que mediante su eliminación deje al grafo dividido en dos componentes conexas.

En el grafo de la figura siguiente la arista {c, f} será una arista puente que dejará el grafo G dividido en las componentes conexas formada por los vértices {f} y { a, b, c, d, e }.

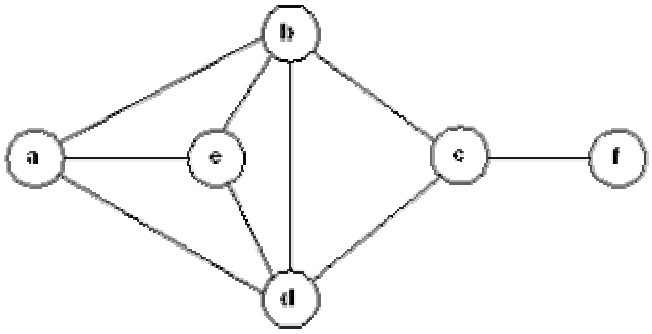
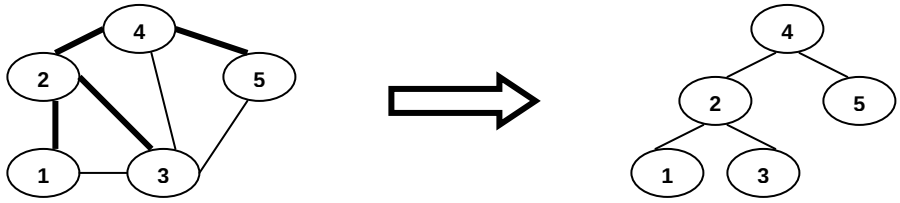



*Figura 16. Arista puente entre los vértices c y f.*

Un ***CAMINO SIMPLE*** es un camino desde un nodo a otro en que ningún nodo se repite (no se pasa dos veces). Si el camino simple tiene como primer y último elemento al mismo nodo se denomina *ciclo*.

Cuando el grafo no tiene ciclos tenemos un *árbol.*. Varios árboles independientes forman un *bosque*.

Un ***ÁRBOL DE EXPANSIÓN*** de un grafo es una reducción del grafo en el que solo entran a formar parte el número mínimo de aristas que forman un árbol y conectan a todos los nodos. Por ejemplo:




*(1)* *(2)*

*Figura 17. Árbol de expansión de un grafo.*

Según el número de aristas que contiene, se hable de ***GRAFO COMPLETO*** si cuenta con todas las aristas posibles (es decir, todos los nodos están conectados con todos), ***GRAFO DISPERSO*** si tiene relativamente pocas aristas y ***GRAFO DENSO*** si le faltan pocas para ser completo.

Las aristas son la mayor parte de las veces bidireccionales, es decir, si una arista conecta dos nodos A y B se puede recorrer tanto en sentido hacia B como en sentido hacia A. Estos son llamados ***GRAFOS NO DIRIGIDOS***.

Sin embargo, en ocasiones tenemos que las uniones son unidireccionales. Estas uniones se suelen dibujar con una flecha y definen un ***GRAFO DIRIGIDO***.

Se habla de ***GRAFO PONDERADO*** Cuando las aristas llevan un coste asociado (un entero denominado ***peso***).

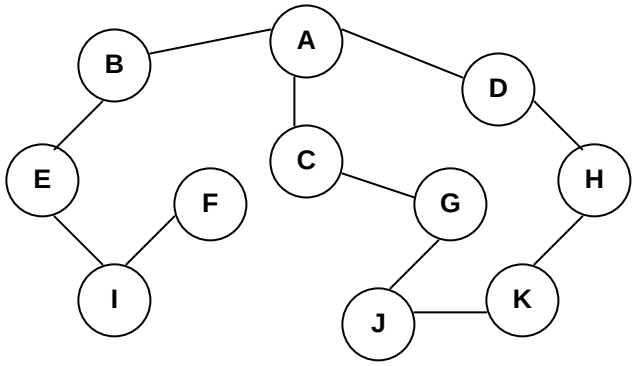
Una ***RED*** es un grafo dirigido y ponderado.

### 2.4.4 EXPLORACIÓN DE GRAFOS

Cuando se habla de la exploración de un grafo nos referimos a la exploración de todos los vértices (con algún tipo de fin, por ejemplo, un recuento) o hasta que se encuentra uno determinado, es decir, una búsqueda.

El orden en que los vértices éstos son "visitados" decide radicalmente el tiempo de ejecución del algoritmo.

Supongamos el siguiente grafo de ejemplo:




A la hora de explorar el grafo de la figura anterior, nos encontramos con dos métodos distintos.

- Exploración o búsqueda en ***Anchura*** o ***Amplitud***: Lo que prima en la exploración es la dimensión horizontal, de manera que por cada vértice del nodo visitamos primeramente todos los nodos enlazados

directamente con él (vecinos), y una vez hecho esto realizamos la misma operación con estos nodos vecinos visitados.

- Exploración o búsqueda en ***Profundidad***: Lo que prima en la exploración es la dimensión vertical. Una vez visitado uno de los nodos vecinos de un nodo, antes de visitar a cualquiera de los demás vecinos del nodo, se vuelve a realiza la misma operación con el nodo vecino recién visitado.

Suponiendo que el orden en que están almacenados los nodos en la estructura de datos correspondiente es A-B-C-D-E-F-G-H-I-J-K (el orden alfabético), tenemos que:

En un recorrido en anchura el orden sería, por contra: **A-B-C-D-E-G-H-I-J-K-F**

El orden que seguiría el recorrido en profundidad sería: **A-B-E-I-F-C-G-J-K-H-D**

### 2.4.5 IMPLANTACIÓN DE LA EXPLORACIÓN DE GRAFOS

En el ejemplo anterior, es destacable que el nodo D es el último en explorarse en la búsqueda en profundidad pese a ser adyacente al nodo de origen (el A). Esto es debido a que primero se explora la rama del nodo C, que también conduce al nodo D.

Es decir, hay que tener en cuenta que es fundamental el orden en que los nodos están almacenados en las estructuras de datos. Si, por ejemplo, el nodo D estuviera antes que el C, en la búsqueda en profundidad se tomaría primero la rama del D (con lo que el último en visitarse sería el C), y en la búsqueda en anchura se exploraría antes el H que el G.

Otro punto a observar es que, cuando hemos hablado en el apartado anterior de exploraciones en anchura y en profundidad, aparecen las siguientes frases “una vez hecho esto realizamos *la misma operación*” y “se vuelve a realizar *la misma operación*”. Esto está sugiriendo de forma muy clara que las exploraciones en grafos, tanto en anchura o en profundidad, tiene una fuerte naturaleza recursiva. De hecho, es frecuente que se implanten ambos algoritmos usando recursividad.

Esto no quiere decir que no puedan implementarse exploraciones de grafos de forma iterativa. Usando algoritmos iterativos, la diferencia principal entre implementar una exploración en anchura frente a una en profundidad esta en la estructura de datos usada, de forma que:

- Las *exploraciones o búsquedas en anchura* iterativas usa una estructura de datos tipo *cola*, pues las características de este tipo de estructura FIFO (primero en entrar, primero en salir) se adaptan muy bien a este tipo de recorrido.
- Las *exploraciones o búsquedas en profundidad* iterativas usa una estructura de datos tipo *pila*, pues las características de este tipo de estructura LIFO (último en entrar, primero en salir) se adaptan muy bien a este tipo de recorridos.

## 3. ORGANIZACIONES DE FICHEROS

Todas las aplicaciones necesitan almacenar y recuperar información. En una computadora, cuando se ejecuta una aplicación (un proceso) la información se almacena en la memoria principal electrónica del computador; este es un tipo de memoria volátil, de forma que cuando la aplicación termina la información se pierde. Esto es inaceptable para muchas aplicaciones, que pueden requerir que la información permanezca disponible durante largos periodos de tiempo.

Con respecto a la memoria principal de las computadoras, se trata de un tipo de memoria electrónicas cuyas principales características son:

- La memoria principal tiene poca capacidad de almacenamiento. No se pueden manipular grandes cantidades de datos, ya que puede haber casos en los que no quepan en la memoria principal.

- La memoria principal es volátil.
- Acceso rápido a la información.

Otro problema es que varios procesos pueden necesitar acceder a una misma información de forma concurrente. Como los espacios de memoria de los procesos son privados, un proceso no puede acceder a los datos en el espacio de memoria de otro. La solución es hacer que la información sea independiente de los procesos.

Por tanto, hay tres requisitos esenciales para almacenar información durante un tiempo indefinido:

- Debe ser posible almacenar una gran cantidad de información.
- La información debe mantenerse (persistir) tras la terminación de los procesos que la usan.
- Varios procesos deben de ser capaces de acceder a la información de forma concurrente.

La solución a estos problemas consiste en almacenar la información en discos magnéticos u otros dispositivos en unas unidades llamadas ***ficheros*** o ***archivos.***

*Un fichero es una abstracción de un mecanismo que permite almacenar información en un dispositivo y leerla posteriormente. Podemos definir un fichero como una colección de información que tiene un nombre.*

Los fichero pueden ser leídos y escritos por cualquier proceso: .son una forma de almacenamiento denominada ***memoria secundaria***. Sus principales cualidades son:

- Capacidad de almacenamiento sólo limitada por el soporte físico de que se disponga.
- La información está almacenada permanentemente.
- Acceso lento a la información, ya que tiene que ser transportada desde el dispositivo externo hasta la memoria principal para su tratamiento. Existe un área de memoria principal destinada a recibir esta información procedente del dispositivo secundario. Esta área se denomina *Buffe*r.

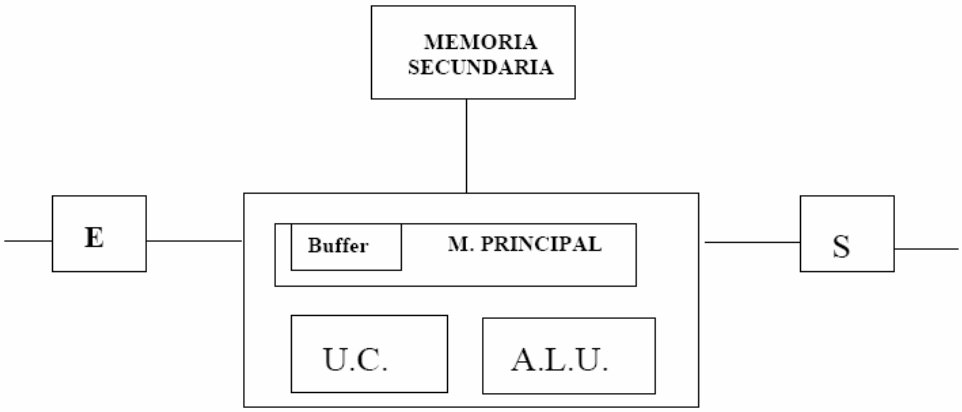



*Figura 18. Memoria principal y memoria secundaria.*

En la figura anterior representamos de manera muy esquemática la forma de operar de un procesador, siendo:

- **U.C. U**nit **C**ontrol (Unidad de Control). Circuito principal de control del procesador.
- **A.L.U. A**ritmetic **L**ogic **U**nit. (Unidad aritmético Lógica). Circuito especializado en operaciones aritméticas.
- **E**: Datos de **E**ntrada.
- **S**: Datos de **S**alida.

La información almacenada en ficheros debe ser ***persistente***, es decir, no debe verse afectada por la creación y finalización de los procesos. La gestión de ficheros es tarea del sistema operativo, y la parte del mismo que realiza dicha gestión se conoce como ***sistema de ficheros.***

Desde el punto de vista de los usuarios, el aspecto más importante de un sistema de ficheros es cómo éste se presenta a ellos. Es decir, qué es un fichero, cómo se nombra, qué operaciones se permiten, etc. En definitiva, al usuario le interesa saber qué es lo que puede hacer. Desde el punto de vista de los diseñadores de sistemas, lo interesante es saber cómo está implantado el sistema de ficheros.

## 3.1 ESTRUCTURA DE UN FICHERO

De la definición vista de fichero, se deduce que existen diferentes tipos de ficheros en función de:

- La información contenida
- El método de organización de la información.

Una primera clasificación de los ficheros se puede hacer según el método usado para codificarla información:

- ***Ficheros de texto***: Se guarda la información en caracteres, tal y como se mostraría en pantalla.
- ***Ficheros binarios***: Se guarda la información en binario, tal y como está en memoria

Otra clasificación de los ficheros es según la forma que tiene su estructura. Las formas mas usuales son:

- Organizar un fichero como ***una secuencia de bytes***. De esta forma, el sistema operativo no conoce el significado del contenido de los ficheros, lo que simplifica la gestión de los mismos. Serán los programas de aplicación los que deberán de conocer la estructura de los ficheros que utilizan. Este enfoque es el empleado por MS-DOS y UNIX.
- Un esquema más estructurado es considerar un fichero como ***una secuencia de registros de longitud fija***, cada uno de los cuales presenta una estructura determinada. La idea es que las operaciones de lectura devuelvan un registro y las escritura modifiquen o añadan un registro. El sistema operativo CP/M usa registros de 128 bytes.
- Una tercera forma es organizar fichero en forma de ***árbol de registros***, que no tienen porque tener la misma longitud. Cada registro tiene un campo clave por el que está ordenado el árbol de forma que las operaciones de búsqueda por clave se realizan rápidamente. Este esquema se emplea en grandes computadores (*mainframes*) orientados al proceso de grandes cantidades de información.

## 3.2 CONCEPTOS BÁSICOS SOBRE FICHEROS

Veamos ahora una serie de conceptos básicos:

- ***Registro lógico***: Un registro es una colección de información relativa a una entidad particular. Por tanto, el registro va a contener a todos aquellos campos lógicamente relacionados, referentes a una determinada entidad, y que pueden ser tratados globalmente por un programa. Por ejemplo la información de un determinado alumno, que contiene los campos DNI, nombre, apellidos, fecha de nacimiento, etc.
- ***Clave de un registro lógico***: Una clave es un campo o conjunto de campos de datos que identifica al registro lógico y lo diferencia del resto de registros lógicos del fichero. Por tanto, esta clave debe ser distinta para cada registro.
- ***Registro activo***: El registro lógico que va a procesarse en la siguiente operación del fichero.
- ***Apuntador***: Marca interna que siempre apunta al registro lógico activo. Se incrementa automáticamente cada vez que se procesa un registro (se lee o se escribe).

- ***Marca de fin de fichero***: Una marca situada al final de cada fichero, para no acceder mas allá del último registro lógico existente, ya que el tamaño del fichero no está limitado y no se conoce a priori. Existe una función lógica, ***eof (end of file)***, que toma el valor verdadero cuando llegamos al final del fichero y falso en caso contrario.
- ***Registro físico o bloque***: Un registro físico o bloque es la cantidad más pequeña de datos que pueden transferirse en una operación de entrada/salida entre la memoria principal del ordenador y los dispositivos periféricos o viceversa. El tamaño del bloque o registro físico dependerá de las características del ordenador. En la mayoría de los casos el tamaño del bloque suele ser mayor que el del registro lógico. La adaptación consiste en empaquetar en cada bloque tantos registros lógicos como se pueda. El empaquetamiento puede ser de tipo *fuerte* o *débil*, según que se permita o no aprovechar el sobrante de un bloque, situando registros a caballo entre dos bloques contiguos. La siguiente figura ilustra ambas formas de empaquetamiento.
- ***Factor de bloqueo***: Factor de bloqueo es el nº de registros lógicos que puede contener un registro físico.

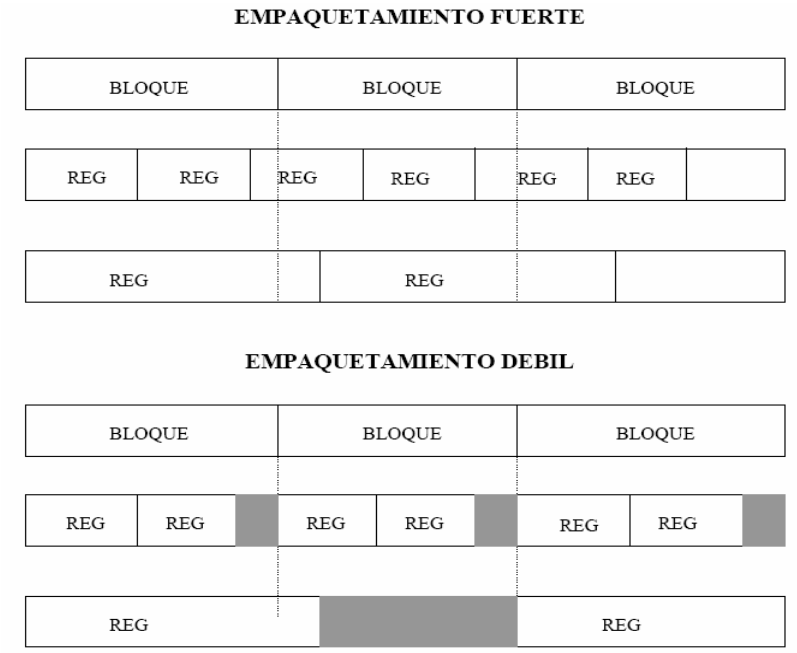



*Figura 19. Empaquetamiento fuerte y empaquetamiento débil.*

Una vez visto lo que es un registro lógico, y teniendo ya en mente la idea de fichero, podemos dar una definición más precisa de los que es un archivo o fichero.

"*Un fichero es una colección de registros lógicos relacionados entre sí con aspectos en común y organizados para un propósito específico. Los datos en los archivos deben estar organizados de tal forma que puedan ser recuperados fácilmente, actualizados o borrados y almacenados en el archivo con todos los cambios realizados".*

Desde un punto de vista puramente estructural:

*"Un fichero es una estructura de datos compuesta que agrupa una secuencia de cero o más tuplas, denominadas corrientemente registros, y que a su vez se pueden componer de otras estructuras de datos a las que se les suele llamar campos".*

## 3.3 OPERACIONES SOBRE FICHEROS

Una vez visto lo que es un fichero y los principales conceptos al hablar de ellos, pasemos ahora a estudiarlos desde un punto de vista operativo. Básicamente se trata de responder: ¿qué operaciones se pueden realizar sobre un fichero?. La respuesta es:

- **Creación:** Para poder realizar cualquier operación sobre un fichero es necesario que haya sido creado previamente, almacenando sobre el soporte seleccionado la información requerida para su posterior tratamiento, como por ejemplo el nombre del dispositivo, el nombre del fichero, etc. Con anterioridad a la creación de un archivo se requiere diseñar la estructura del mismo mediante los campos del registro, longitud y tipo de los mismos.
- **Apertura:** Para poder trabajar con la información almacenada en un fichero, éste debe estar abierto, permitiendo así el acceso a los datos, dando la posibilidad de realizar sobre ellos las operaciones de lectura y escritura necesarias.
- **Cierre:** Una vez finalizadas las operaciones efectuadas sobre el fichero, éste debe permanecer cerrado para limitar el acceso a los datos y evitar así un posible deterioro o pérdida de información. Para cerrar un fichero previamente debe estar abierto.
- **Actualización**: Esta operación permite la puesta al día de los datos del fichero mediante la escritura de nuevos registros (alta) y la eliminación (baja) o modificación de los ya existentes. La actualización puede afectar a parte o la totalidad de los registros del fichero. Cuando se escribe un nuevo registro en el fichero se debe comprobar que no existe previamente. La baja de un registro puede ser lógica o física.

  Una *baja lógica* supone el no borrado del registro en el archivo. Esta baja lógica se manifiesta en un determinado campo del registro con una bandera, indicador o "flag", o bien con la escritura o rellenado de espacios en blanco en el registro específico.

  Una *baja física* implica el borrado y desaparición del registro, de modo que se crea un nuevo archivo que no incluye al registro dado de baja.
- **Consulta:** Tiene como fin visualizar la información contenida en el fichero, bien de un modo completo, bien de modo parcial.
- **Borrado o destrucción:** Es la operación inversa a la creación de un fichero. Consiste en la supresión de un fichero del soporte o dispositivo de almacenamiento. El espacio utilizado por el archivo borrado puede ser utilizado por otros archivos. Para borrar un fichero tiene que estar cerrado.
- **Ordenación o clasificación:** Consiste en lograr una nueva disposición sobre el soporte de los registros de un archivo, con una secuencia de ubicación determinada por el valor de uno o varios campos.
- **Compactación o empaquetamiento:** Esta operación permite la reorganización de los registros de un fichero eliminando los huecos libres intermedios existentes entre ellos normalmente ocasionados por la eliminación de registros.

## 3.4 ORGANIZACIÓN DE UN SISTEMA DE FICHEROS

Los discos magnéticos son la base sobre la que se sustentan los sistemas de ficheros. Para mejorar la eficiencia, la transferencia de información entre memoria y los discos se realiza en unidades denominadas bloques. Cada bloque está formado por uno o varios sectores de disco. El tamaño de sector de un disco suele ser de 512 bytes.

El diseño de un sistema de ficheros plantea dos problemas diferentes:

- Definir cómo el sistema de ficheros aparece al usuario.
- Diseñar los algoritmos y estructuras de datos necesarias para implementar este sistema de ficheros lógico en los dispositivos físicos de almacenamiento secundario.

Un sistema de ficheros se puede estructurar en diferentes capas o niveles (Figura 20). El nivel de control de entrada/salida se compone de los manejadores de dispositivo (*device drivers*) y los manejadores de

interrupciones (*interrupt handlers*), que son necesarios para transmitir la información entre la memoria y los discos. Los manejadores de dispositivos reciben instrucciones de bajo nivel, del tipo "escribir o leer el bloque nº x", y generan el conjunto de instrucciones dependientes del hardware que son enviadas al controlador de disco.

El sistema de ficheros básico transmite las instrucciones de bajo nivel al manejador de dispositivo adecuado para leer y escribir bloques físicos en disco. Cada bloque físico se identifica por su dirección numérica en el disco, que viene dado por el dispositivo, cilindro, superficie y sector.

El módulo de organización de ficheros tiene conocimiento sobre los ficheros, los bloques lógicos que los componen, y los bloques físicos. Mediante el tipo de esquema de asignación de bloques y la localización del fichero, el módulo de organización de ficheros traslada las direcciones de disco lógicas en direcciones de disco físicas. Cada bloque de disco lógico tiene un número (de 0 a N), que no suele coincidir con la dirección de los bloques físicos, por lo que es necesario un mecanismo de traducción. Este módulo también incluye el gestor de espacio libre, que controla los bloques libres para que puedan ser usados posteriormente.

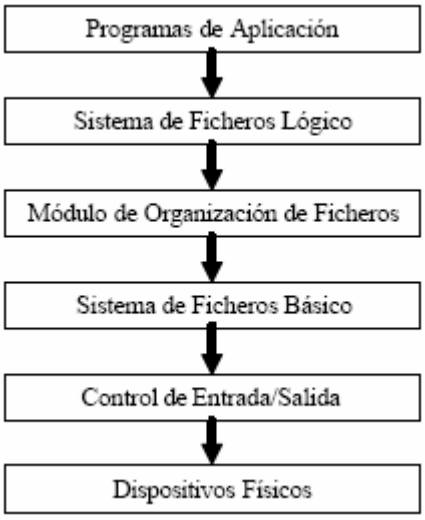



*Figura 20. Estructura en niveles de un sistema de ficheros.*

Por último, el sistema de ficheros lógico proporciona la estructura de directorio que conocida por los programas de usuario. También es responsable de proporcionar seguridad y protección al sistema de ficheros.

Por otra parte, para crear un nuevo fichero, un programa de aplicación realiza una llamada al sistema sobre el sistema de ficheros lógico. Este lee el directorio correspondiente en memoria, le añade una nueva entrada y lo escribe en disco. En este proceso, el sistema de ficheros lógico ha solicitado al módulo de organización de ficheros la dirección física del directorio, que se envía al sistema de ficheros básico y al control de entrada/salida. Cuando el directorio ha sido actualizado, el sistema de ficheros lógico lo puede usar para realizar operaciones de entrada/salida.

Para optimizar los accesos, el sistema operativo tiene en memoria una tabla de ficheros abiertos que contiene información sobre todos los ficheros abiertos en un momento dado, como nombre, permisos, atributos, direcciones de disco, etc. La primera referencia a un fichero suele ser una operación de apertura, que hace que se realice un búsqueda en la estructura de directorio y se localice la entrada para el fichero que se quiere abrir. Esta entrada se copia en la tabla de ficheros abiertos y el índice de dicha entrada (denominado descriptor de fichero) se devuelve al programa de aplicación, que lo usa para realizar las operaciones sobre el fichero. De esta forma, todas las operaciones relacionadas con la entrada de directorio del fichero se realizan en la tabla de ficheros abiertos en memoria. Cuando el fichero se cierra, la entrada modificada se copia a disco.

## 3.5 ORGANIZACIÓN Y ACCESO A FICHEROS

Se entiende por ***Organización de un fichero*** a la manera en la que los datos son estructurados y almacenados internamente en el fichero y sobre el soporte de almacenamiento. El tipo de organización de un fichero se establece durante la fase de creación del mismo. Los requisitos que determinan la organización de un fichero son del tipo: tamaño del fichero, frecuencia de utilización o uso, etc.
La organización de un fichero es muy dependiente del soporte físico en que se almacene. Hay dos tipos de soportes:

- Soportes ***Secuenciales***: Los registros están dispuestos físicamente uno a continuación de otro. Para acceder a un determinado registro se necesita pasar por todos los anteriores a él.
- Soportes ***Direccionables***: Permiten localizar a un registro directamente por su información (clave) sin tener que pasar por todos los anteriores.

Los tipos de organizaciones de ficheros fundamentales son:

- Organización **Secuencial.**
- Organización **Directa** o **Aleatoria.**
- Organización **Indexada.**

En cuando al **acceso**, se entiende por tal al procedimiento necesario que debemos seguir para situarnos sobre un registro concreto con la intención de realizar una operación sobre él. Según las características del soporte empleado y la organización se consideran dos tipos de acceso:

- El **acceso secuencial** implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro. Se puede dar en dispositivos secuénciales y direccionables.
- El **acceso directo** implica el acceso a un registro determinado, sin que ello implique la consulta de los registros precedentes. Obviamente, sólo puede darse en soportes direccionables.

### 3.5.1 ORGANIZACIÓN SECUENCIAL

Son aquellos ficheros caracterizados porque los registros se escriben o graban sobre el soporte de almacenamiento en posiciones de memoria físicamente contiguas, en la misma secuencia u orden en que han sido introducidos, sin dejar huecos o espacios libres entre ellos.

Todos los dispositivos de memoria auxiliar soportan la organización secuencial. El *acceso a los datos* almacenados en estos ficheros siempre es *secuencial* independientemente del soporte utilizado. Los registros organizados secuencialmente tienen un registro especial, el último, que tiene una marca de fin de archivo.

Sus ventajas son:

- Rapidez en el acceso a un bloque de registros que se encuentran almacenados en posiciones de memoria físicamente contiguas.
- No deja espacios vacíos entre registro y registro, optimizando al máximo la memoria ocupada.

Sus inconvenientes son:

- El acceso a registros individuales es muy lento.
- Se tiene que procesar todo el fichero para operaciones de inserción y borrado de registros.

### 3.5.2 ORGANIZACIÓN DIRECTA O ALEATORIA

También llamada organización ***aleatoria***. Estos ficheros se caracterizan porque los registros se sitúan en el fichero y se accede a ellos a través de una clave, que indica la posición del registro dentro del fichero y la posición de memoria donde está ubicado.

Estos ficheros se almacenan en *soportes direccionables*. Además, los registros han de tener un identificativo o clave, el cual indica la posición de cada registro en el fichero.

Como principales ventajas genéricas de este tipo de organización, tenemos que:

- Cada posición solamente puede ser ocupada por un registro, pues no podemos tener en el fichero más de un registro con el mismo valor de clave.
- El acceso a cualquier registro se hace de una forma directa e inmediata mediante su clave.
- La actualización de un registro es inmediata, sin que se deban utilizar archivos auxiliares para copia.
- Se puede utilizar el acceso secuencial, aunque suponga generalmente una pérdida de tiempo.

Dentro de la organización directa, según el algoritmo utilizado en la gestión de la clave, se puede distinguir entre:

- Organización directa con ***Clave Directa***: La dirección de almacenamiento del registro está indicado por la propia clave.

  Sus ventajas son:

  - Cada posición solamente puede ser ocupada por un registro, pues no podemos tener en el fichero más de un registro con el mismo valor de clave.
  - Es muy rápido el acceso a los registros individuales.

  Sus inconvenientes son:

  - Deja gran cantidad de huecos dentro del fichero, con el consecuente desaprovechamiento del soporte de almacenamiento. Esto es debido que este sistema precisa que el soporte donde se almacena la información tenga una mínima unidad de asignación (denominado *cluster*) a la cual acceder directamente siguiendo unas coordenadas de localización. Estas unidades no pueden ser usadas para almacenar información de distintos ficheros. Si los cluster en que se divide el disco son grandes, y los ficheros a almacenar pequeños, habrá muchos cluster que se quedarán a medio llenar, con el consecuente desaprovechamiento de espacio.
  - Una consulta total del fichero puede suponer un gran inconveniente, pues hay que analizar todas las posiciones de memoria, aunque algunas posiciones estén vacías. Para comprender este hecho hagamos el siguiente símil. Supongamos que tenemos un libro con todos sus capítulos correctamente ordenados pero sin índice que nos identifique la página de inicio de cada capítulo, para leer el libro entero lo único que hay que hacer es comenzar por la primera página, sin embargo si queremos leer un capítulo en concreto, tendríamos también que empezar desde el principio hasta encontrarlo. Esto sería el análogo a un fichero secuencial.

    Supongamos ahora un libro con todos los capítulos desordenados, pero con un índice al comienzo del libro que nos indica la página de comienzo de cada capítulo; es decir, el análogo al un fichero de acceso directo. Leer cada capítulo por separado es ahora muy fácil, basta con buscar el comienzo en el índice, sin embargo, si necesitamos leer el libro entero, de principio a fin, necesitamos constantemente mirar el índice para ir viendo la secuencia de los capítulos, lo cual es muy ineficiente (por no decir pesado). Esto sería el análogo a leer completamente un fichero de acceso directo

- Organización directa con ***Clave Indirecta***: La dirección de almacenamiento se obtiene a partir de la clave, después de realizar algún tipo de transformación. Este tipo de transformación se denomina algoritmo *Hashing* (recuérdese las tablas hash ya vistas en los algoritmos de búsqueda) y suele ser de tipo matemático.

En este tipo de algoritmo se pueden dar dos situaciones no deseadas (denominadas *colisiones*), que son:

- Hay direcciones que no corresponden a ninguna clave y, por tanto, zonas de disco sin utilizar.
- Hay direcciones que corresponden a más de una clave. En este caso se dice que las claves son sinónimas para esa transformación.

Hay dos formas de resolver el problema de los sinónimos o colisiones:

- Buscar secuencialmente en el archivo hasta encontrar una posición libre donde escribir el registro o aplicando a la dirección obtenida un segundo método de direccionamiento. Estos procedimientos son lentos y degradan el archivo.
- Reservar una zona de desbordamiento o de sinónimos en donde se escribirán los registros que no se pueden escribir en la dirección que le corresponde según la transformación.

### 3.5.3 ORGANIZACIÓN INDEXADA

Al fichero le acompaña un fichero de índice que tiene la función de permitir el acceso directo a los registros del fichero de datos. El índice se puede organizar de diversas formas, las más típicas son:

- **Secuencial**
- **Multinivel**
- **Árbol**

A través del índice podremos procesar un fichero de forma secuencial o de forma directa según la clave de indexación, y esto independientemente de como esté organizado el fichero por sí mismo.

El índice debe estar organizado en función de alguno de los campos de los registros de datos. Se pueden tener tantos índices como se quiera variando la clave (o campo) que se emplee. El índice está formado por registros (entradas) que contienen:

- Clave de organización.
- Puntero(s) al fichero de datos, en concreto al registro que corresponda.

Los índices se pueden clasificar en dos tipos, según cada entrada señale a la dirección de un registro del fichero de datos (*índice total o denso*), o bien apunte a un grupo de registros del fichero de datos que debe estar ordenado (*índice escaso o no denso*). En el caso de índices totales, el fichero puede estar desordenado. Véase la siguiente figura como ejemplo:

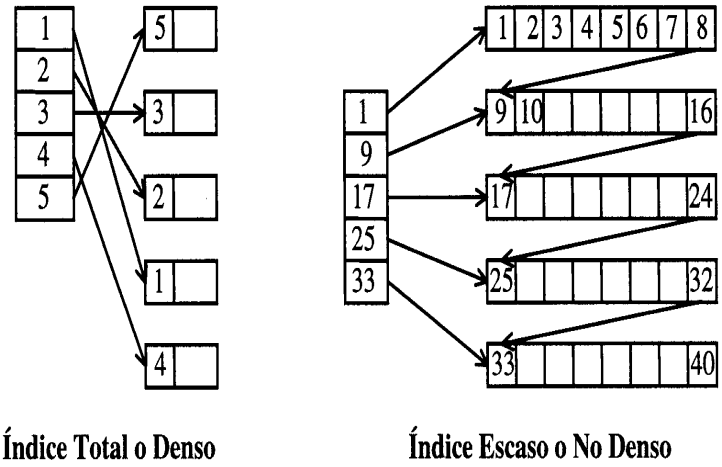



*Figura 21. Índice Total (Denso) y Escaso (No Denso).*

Con el segundo tipo se podría procesar directamente el fichero de datos de forma secuencial. Los índices totales o densos no suelen utilizarse de forma simple, sino combinados con índices escasos más cortos, de esta manera pueden almacenarse en memoria principal obteniendo así una mayor rapidez de acceso.

### 3.6 IMPLANTACIÓN DE FICHEROS

El aspecto básico de la implantación de ficheros consiste en determinar qué bloques de disco están asociados a cada fichero. El problema que se plantea es cómo asignar el espacio libre en disco a los ficheros de forma que ese espacio sea usado de forma eficiente y los ficheros puedan ser accedidos rápidamente. Hay que tener en cuenta que el tamaño de los ficheros es variable, por lo que habrá que diseñar algún mecanismo de almacenamiento dinámico tanto para los ficheros como para el espacio libre.

Existen tres métodos básicos de asignación:

- Asignación ***Contigua***.
- Asignación ***Enlazada***.
- Asignación ***Indexada***.

#### 3.6.1 ASIGNACIÓN CONTIGUA

El esquema de asignación más simple consiste en almacenar cada fichero como un secuencia adyacente de bloques en disco. La asignación contigua de un fichero se define por la dirección del primer bloque y el tamaño del fichero, es decir, simplemente responder a las preguntas ¿dónde empezar? y ¿hasta cuando seguir?.

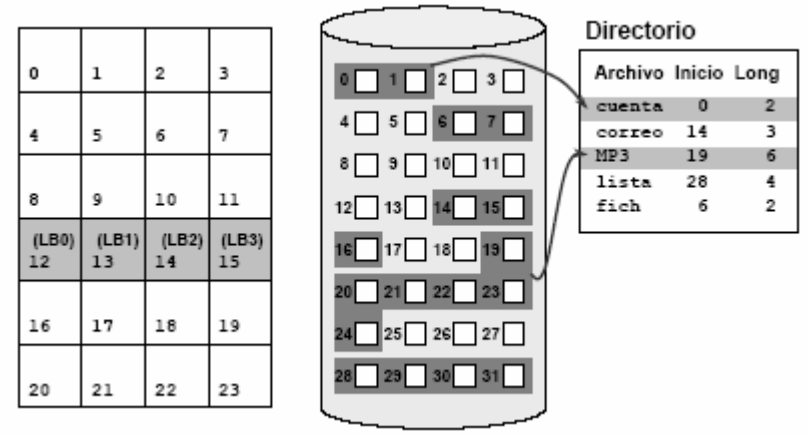



*Figura 22. Asignación Contigua.*

Las ventajas de este método son:

- Fácil implantación, ya que únicamente hay que conocer la dirección en disco del primer bloque del fichero y el número de bloques que ocupa
- Eficiencia, ya que la lectura de un fichero se puede realizar en una sola operación. El acceso a un fichero almacenado de forma contigua es sencillo y rápido, siempre y cuando pretendamos leer todo el fichero y no tengamos que hacer búsquedas de parte de su contenido. En este tipo de operaciones se pierde toda la ventaja que proporciona esta forma de asignación. Para este tipo de operaciones veremos que en mucho más ventajosa la asignación

Para acceso secuencial, el sistema de ficheros no tiene más que recordar la dirección del último bloque referenciado. El acceso directo del bloque `i` de un fichero que comienza en el bloque `b` se realiza, simplemente, accediendo al bloque `b+i`.

Un inconveniente de este esquema de asignación es que no se puede implantar a no ser que se conozca el tamaño de los ficheros en su momento de creación. Si esta información no está disponible, el sistema operativo no sabe cuánto espacio en disco debe reservar. Otra desventaja es la fragmentación externa que se produce en el disco, que se origina debido a que entre los ficheros quedan porciones de disco libre que no tienen el tamaño suficiente para ser asignadas a un nuevo fichero. La gravedad de este problema depende del espacio de almacenamiento disponible y del tamaño medio de los ficheros. La compactación de disco es una solución, pero es costosa y de poder hacerse sería cuando el sistema estuviese parado.

El problema de conocer el tamaño de los ficheros en tiempo de creación se resuelve haciendo que los ficheros sean inmutables. Esto significa que cuando un fichero se crea no se puede modificar. Una modificación implica borrar el fichero original y crear uno nuevo. Por otro lado, los ficheros se han de leer de una sola vez en memoria, por lo que la eficiencia es muy alta, pero a cambio se exigen unos requisitos mínimos de memoria muy elevados. Por último, la fragmentación de disco se soluciona teniendo discos de gran capacidad de almacenamiento.

### 3.6.2 ASIGNACIÓN ENLAZADA

Otro esquema consiste en mantener una lista enlazada con los bloques que pertenecen a un fichero, de forma que una palabra del bloque sería un índice al bloque siguiente. Con este método se elimina el problema de la fragmentación, y, al igual que en la asignación contigua, cada entrada de directorio únicamente contiene la dirección del primer bloque del fichero. Los ficheros pueden crecer de forma dinámica mientras que exista espacio libre en disco y no es necesario compactar los discos. Un inconveniente es que, mientras que el acceso secuencial es fácil de realizar, el acceso aleatorio es muy lento.

Otro inconveniente viene dado por el espacio requerido por los punteros. Si un puntero requiere 4 bytes y los bloques son de 512 bytes, significa que: (4 / 512) = 0,0078 $\rightarrow$ se pierde un 0,78% (casi un 1%) de espacio útil por bloque. Una solución a este problema consiste en no asignar bloques individuales, sino conjuntos de bloques

denominados *clusters*. De esta forma el porcentaje de espacio perdido por los punteros se reduce. Además, se mejora el rendimiento de los discos porque se lee más información en una misma operación de lectura, y se reduce la lista de bloques libres. El coste de este método es que se produce un incremento de la fragmentación interna (espacio no ocupado en un bloque o cluster).

Otro problema de este método es la fiabilidad, ya que si un bloque se daña, se puede perder el resto del fichero. Peor aún, si se altera el contenido del puntero, se puede acceder a la lista de bloques libres o a bloques de otro fichero como si fuesen bloques propios.

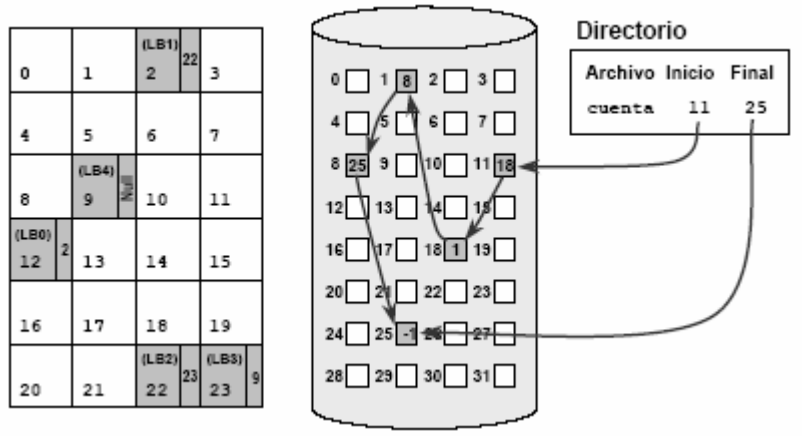



*Figura 23. Asignación enlazada.*

Un problema más sutil viene dado por el hecho de que la cantidad de información útil para el usuario que contiene cada bloque no es potencia de dos, ya que el puntero al bloque siguiente, que estará constituido por varias palabras, ocupará parte del bloque, es decir, no se puede utilizar todo el bloque para información útil, siempre habrá que dejar mínimo espacio para el puntero. Esto puede originar una pérdida de eficiencia porque los programas normalmente leen y escriben en bloques cuyo tamaño es potencia de dos. Esta desventaja del esquema de lista encadenada desaparece si el puntero de cada bloque de disco se almacena en una tabla o índice en memoria. Así, cada bloque únicamente contiene datos útiles.

Además, aunque el acceso aleatorio implica seguir una cadena, ésta está toda en memoria, por lo que la búsqueda es mucho más eficiente que en el esquema anterior. De igual modo, es suficiente que cada entrada de directorio contenga únicamente la dirección del primer bloque para localizar todo el fichero. Este esquema es empleado por el sistema operativo MS-DOS. Para que este esquema funcione, la tabla debe de permanecer entera en memoria todo el tiempo. Si el disco es grande, el gasto de memoria principal puede ser considerable.

### 3.6.3 ASIGNACIÓN INDEXADA

El último método para determinar qué bloques pertenecen a un fichero consiste en asociar a cada fichero un nodo índice (*index-node ó i-node*), que incluye información sobre los atributos del fichero y las direcciones de los bloques de disco que pertenecen al mismo.

Las direcciones de los primeros bloques se almacenan el propio nodo índice, de forma que, para ficheros pequeños, toda la información necesaria para su localización siempre está en memoria cuando son abiertos. Para ficheros de mayor tamaño, una de las direcciones del nodo índice es la dirección de un bloque simple indirecto, que contiene a su vez direcciones de otros bloques del fichero. Si no es suficiente, existen dos direcciones, una para un bloque doble indirecto, que contiene direcciones de bloques simples directos, y otra para un bloque triple indirecto, que contiene direcciones de bloques dobles indirectos.

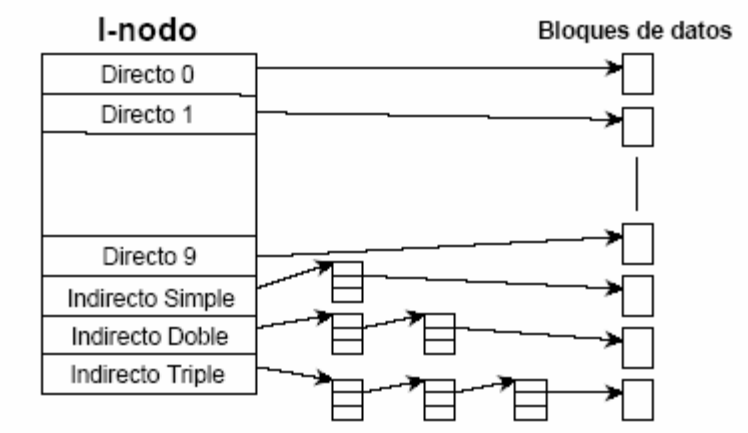



*Figura 24. Nodo índice en UNIX.*

Una ventaja es que permite el acceso directo a cualquier bloque del fichero. Además, los accesos a un fichero, una vez abierto, no implican accesos adicionales a disco a no ser que sean necesarios bloques indirectos.

Un inconveniente es que añadir o borrar bloques en medio del fichero implica el tener que reorganizar los punteros en el bloque de índices, lo cual constituye una operación costosa.

### 3.7 DIRECTORIOS

Los sistemas de ficheros pueden contener grandes volúmenes de información, por lo que los sistemas operativos proporcionan mecanismos para estructurarlos. Esta estructuración se suele realizar a dos niveles.

En primer lugar, el sistema de ficheros se divide en particiones, que se pueden considerar como discos virtuales. Un disco puede contener una o varias particiones, y una partición puede estar repartida En segundo lugar, cada partición contiene una tabla de contenidos de la información que contiene. Esta tabla se denomina directorio, y su función principal es hacer corresponder un nombre de un fichero con una entrada en la tabla. Cada entrada contiene, como mínimo, el nombre del fichero.

| | | |
|---|---|---|
| bin | atributos | direcciones |
| utiles | atributos | direcciones |
| programas | atributos | direcciones |
| correo | atributos | direcciones |

*Figura 25. Atributos y direcciones en la entrada de Directorio .*

A continuación se pueden almacenar los atributos y direcciones del fichero en disco (Figura 24), o un puntero a otra estructura de datos que contiene dicha información (Figura 25). Esta estructura de datos se suele conocer con el nombre de **nodo índice**.

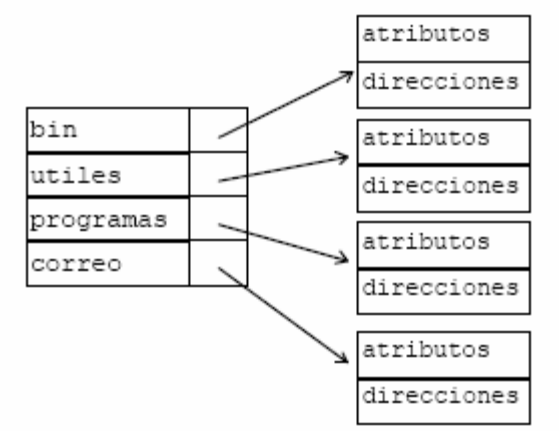



*Figura 26. Atributos y direcciones en una estructura de datos.*

Cuando se abre un fichero el sistema operativo busca en el directorio correspondiente hasta que encuentra el nombre del fichero, toma la información que necesita y la introduce en una tabla residente en la memoria principal. Las siguientes referencias al fichero utilizarán la información que ya está en memoria.

### 3.8 IMPLANTACIÓN DE DIRECTORIOS

Para acceder a un fichero, ya sea para lectura o escritura, previamente hay que abrir el fichero. Esta operación implica que el sistema operativo, a partir de la ruta de acceso, debe localizar la entrada de directorio correspondiente. Esta entrada proporciona la información necesaria para localizar los bloques de disco del fichero. Esta información, según el tipo de sistema, puede ser la dirección en disco del fichero completo (asignación contigua), el número del primer bloque (los dos métodos basados en listas encadenadas) o el número de nodo índice. En todo caso, la principal función del sistema de directorios es asociar el nombre en ASCII de un fichero con la información precisa para localizar los datos. Los atributos de los ficheros se pueden almacenar en la misma entrada de directorio. Si usan nodos índice, es posible almacenar los atributos en el nodo índice.

A continuación se exponen brevemente la implantación de directorios en varios sistemas operativos: CP/M, MS-DOS y UNIX. No vamos a mencionar nada de las características de cada sistema operativo, simplemente mencionamos estos sistemas operativos para ver cómo distintas filosofías de implementar directorios.

#### 3.8.1 DIRECTORIOS EN CP/M

En CP/M solamente existe un directorio, por lo que el sistema de ficheros únicamente tiene que buscar el nombre de un fichero en dicho directorio. Si un fichero ocupa más de 16 bloques se le asignan más entradas de directorio.

Los campos de una entrada de directorio en CP/M son los siguientes:

- El código de usario (1 byte) permite conocer el propietario del fichero.
- El nombre y extensión del fichero (8 y 3 bytes, respectivamente).
- Si un fichero ocupa más de 16 bloques, el campo magnitud (1 byte) indica el orden que ocupa esa entrada.
- El contador de bloques (1 byte) indica cuántos bloques están en uso de los 16 posibles.
- Los últimos 16 campos contienen las direcciones de bloques de disco del fichero. Como el último bloque puede no estar lleno, es imposible conocer con exactitud el tamaño en bytes de un fichero.

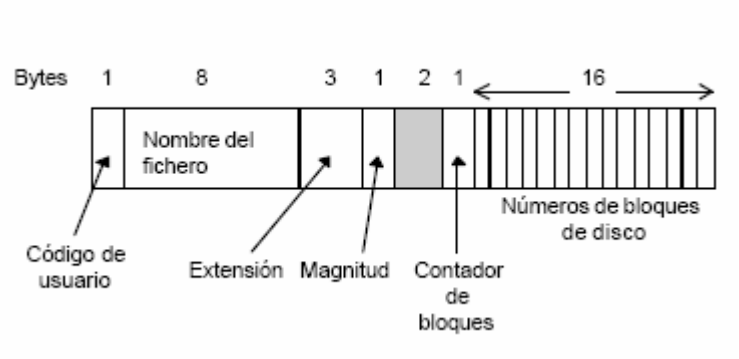



*Figura 27. Entrada de directorio en CP/M.*

### 3.8.2 DIRECTORIOS EN MS-DOS

MS-DOS tiene un sistema de ficheros jerárquico. Cada entrada de directorio tiene 32 bytes, y contiene, entre otros datos, el nombre del fichero, la extensión, los atributos y el número del primer bloque en disco del fichero. Este número es un índice a una tabla de asignación denominada FAT (*File Allocation Table*). Un directorio puede contener otros directorios, lo que origina la estructura jerárquica del sistema de ficheros.

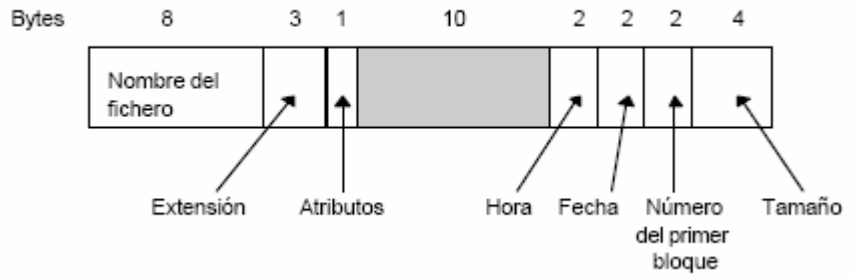



*Figura 28. Entrada de directorio en MS-DOS.*

### 3.8.3 DIRECTORIOS EN UNIX

La estructura de una entrada de directorio en un sistema UNIX tradicional es muy simple, y contiene únicamente el nombre del fichero y el número de su nodo índice (recuérdese los nodos índices vistos anteriormente). Toda la información sobre el fichero reside en el propio nodo índice.

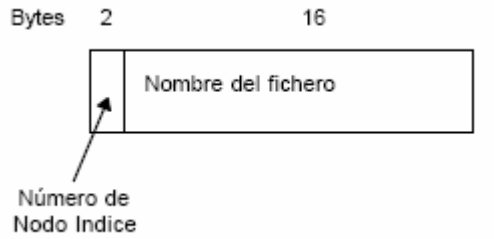



*Figura 29.- Entrada de directorio en UNIX.*

## 4. FORMATOS DE INFORMACIÓN Y FICHEROS

Independientemente de la plataforma con que se trabaje, la información obtenida se genera con un ordenador y se suele almacenar en un fichero, con la intención de recuperarla más tarde cuando sea necesaria, o compartirla con los demás a través de algún medio de transmisión de datos. En consecuencia, es conveniente conocer los formatos de archivo más indicados para almacenar los distintos tipos de información que deben contener.

En este apartado vamos a estudiar los ficheros desde un punto de vista operativo, es decir, según el uso o función que se le da al fichero. Podría decirse que desde el punto de vista del usuario, al que le interesa guardar datos en un archivo y tenerlo localizado de alguna forma en su ordenador independientemente de cómo esté organizado el fichero o cual sea su estructura.

## 4.1. FORMATOS DE INFORMACIÓN

La mayoría de aplicaciones suelen guardar la información que producen en formatos de fichero propios, de modo que podemos editarlos posteriormente con la garantía que se respetarán todas las peculiaridades de los datos y el nivel de edición que poseían en el momento de guardarlos, ahora, cuando compartimos información debemos ser muy cuidadosos con la elección del tipo de fichero ya que no debemos asumir que todo el mundo posee nuestras mismas herramientas ni nuestro mismo sistema. Por ejemplo, si enviamos un archivo pdf a alguien que no tiene instalado el Adobe Acrobar Reader de la compañía Adobe™ no podrá leerlo.

No se trata de analizar en profundidad las características de todos los tipos de archivo, sino indicar algunas referencias generales sobre los más usados que nos ayuden a decidir el formato adecuado en cada ocasión.

### 4.1.1 FICHEROS CON INFORMACIÓN DE TEXTO

Si queremos almacenar o compartir un fichero de texto tenemos dos formatos básicos independientes de la plataforma, es decir, son legibles con un editor de texto sobre cualquier sistema operativo:

- **TXT**, para ficheros de texto plano.
- **RTF**, *Rich Text Format* (Formato de texto enriquecido) cuando sea necesario incluir en el texto algunos elementos de realce como cursivas o negritas.

Por otra parte, para enviar un texto por correo electrónico no es preciso adjuntar un fichero, basta incluirlo en el cuerpo del mensaje.

### 4.1.2 FICHEROS CON INFORMACIÓN DE IMAGEN

Hay gran variedad de formatos de archivos gráficos, la mayoría compatibles con cualquier plataforma. Entre los más habituales se encuentran:

- **JPG** para imágenes de tono continuo en mapa de bits. Es un formato comprimido pues prescinde de los datos de color de la imagen que no están en el espectro visible.
- **GIF** usado especialmente con animaciones y gráficos con regiones transparentes. Tiene algunos problemas legales con los términos de su licencia y su utilización ha descendido en favor del JPG.
- **PNG** tiene similares características al GIF aunque se trata de un formato más evolucionado y de mayor calidad, con muy buenas ratios de compresión y soporte para multitransparencia. Posee una licencia libre y ha experimentado una difusión últimamente.
- **TIFF** se utiliza para almacenar imágenes sin pérdida de calidad, por lo que genera tamaños de archivo mayores que el resto pese a que incorpora un algoritmo de compresión.
- **SVG** para ilustraciones vectoriales

### 4.1.3 FICHEROS CON INFORMACIÓN COMPUESTA

Cuando se trata de compartir documentos que integran texto con imágenes o gráficos, o la composición y aspecto son fundamentales por tratarse de formularios estandarizados o similares, tenemos dos alternativas:

- **PS** es un documento PostScript o formato de impresión capaz de ser visualizado con alguna aplicación auxiliar e impreso sin problemas, directamente. Mantienen la misma calidad de resolución que el documento original.
- **PDF** es una versión del anterior, desarrollada por la compañía Adobe™ que se usa frecuentemente para compartir documentación en la Internet gracias a la difusión del visualizador gratuito Acrobat Reader™.

### 4.1.4 FICHEROS CON INFORMACIÓN COMPRIMIDA

Para aliviar las dificultades de transmitir archivos de gran tamaño a través de las redes o ahorrar espacio en disco, se desarrollaron distintos algoritmos de compresión capaces de reducir la cantidad de memoria ocupada por un fichero. Tal vez, el formato más usado para esto sea el *ZIP*.

## 4.2. TIPOS DE FICHERO SEGÚN SU USO

Atendiendo al que se le da a un fichero se pueden clasificar los ficheros en:

- *Ficheros* ***Permanentes***: Contienen los datos relevantes para una aplicación. Su vida es larga y no pueden generarse de forma inmediata. Entre estos se puede distinguir entre:
  - Ficheros ***maestros****:* contienen el estado actual de los datos susceptibles de ser modificados en la aplicación. Ejemplo: el fichero de clientes actuales de un banco.
  - Ficheros ***constantes***: contienen datos fijos para la aplicación. Ejemplo: el fichero de códigos postales.
  - Ficheros ***históricos****:* contienen datos que fueron actuales en tiempos anteriores. Ejemplo: el fichero de clientes que se han dado de baja en una entidad bancaria.
- *Ficheros* ***Temporales***: Contienen datos relevantes para un proceso o programa. Su vida es corta y se utilizan para actualizar los ficheros permanentes.
  - Ficheros de ***movimientos*****:** almacenan resultados de un programa que han de ser utilizados por otro, dentro de una misma tarea. Ejemplo: el fichero de movimientos de una cuenta bancaria.
  - Ficheros de ***maniobras****:* almacenan datos que un programa no puede conservar en memoria principal por falta de espacio. Ejemplo: editores, compiladores y programas de cálculo numérico.
  - Ficheros de ***resultados****:* se utilizan para almacenar datos elaborados que van a ser transferidos a un dispositivo de salida. Ejemplo: un fichero de impresión.

## 4.3. DENOMINACIÓN DE FICHEROS

Los ficheros tienen asignados un nombre a través del cual los usuarios se refieren a ellos. Sin embargo, las reglas de denominación de ficheros difieren de sistema a sistema. Muchos sistemas operativos dividen el nombre de los ficheros en dos partes separadas por un punto. La primera parte sería el nombre, propiamente dicho. La segunda parte se suele denominar *extensión* y suele aportar información sobre el contenido del fichero. Por ejemplos, los ficheros cuya extensión es el carácter 'c' indican que contienen programas escritos en el lenguaje C.

Las reglas básicas de denominación de ficheros en los sistemas operativos MS-DOS, UNIX y Windows NT son:

- **MS-DOS**: El nombre de un fichero se compone de un máximo de ocho caracteres, seguidos, opcionalmente, por un punto y una extensión de tres caracteres como máximo. Las mayúsculas y minúsculas son consideradas iguales. (Por ejemplo, *archivo12.doc*)
- **UNIX**: El nombre de un fichero se compone de un máximo de 256 caracteres. Se distinguen las mayúsculas de las minúsculas. Un fichero puede tener más de una extensión. (Por ejemplo, `image.tar.Z`)

- **Windows NT**: El nombre de un fichero se compone de un máximo de 256 caracteres. Las mayúsculas y minúsculas no son distinguibles y los ficheros pueden tener más de una extensión.

En muchos casos, las extensiones son meras convenciones y no relacionadas con el contenido de los ficheros. Por otro lado, muchas aplicaciones requieren que los ficheros que utilizan tengan unas extensiones concretas.

## 5. CONCLUSIÓN

Para procesar información en un ordenador es necesario hacer una abstracción de los datos que tomamos del mundo real -abstracción en el sentido de que se ignoran algunas propiedades de los objetos reales, es decir, se simplifican-. Se hace una selección de los datos más representativos de la realidad a partir de los cuales pueda trabajar el computador para obtener unos resultados.

Cualquier lenguaje suministra una serie de tipos de datos simples, como son los números enteros, caracteres, números reales. En realidad suministra un subconjunto de éstos, pues la memoria del ordenador es finita. Los punteros (si los tiene) son también un tipo de datos. El tamaño de todos los tipos de datos depende de la máquina y del compilador sobre los que se trabaja.

En principio, conocer la representación interna de estos tipos de datos no es necesaria para realizar un programa, pero sí puede afectar en algunos casos al rendimiento.

Una estructura de datos trata de un conjunto de variables de un determinado tipo agrupadas y organizadas de alguna manera para representar un comportamiento. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. A veces la dificultad para resolver un problema radica en escoger la estructura de datos adecuada. Y, en general, la elección del algoritmo y de las estructuras de datos que manipulará están muy relacionadas.

Según su comportamiento durante la ejecución del programa distinguimos estructuras de datos:

- ***Estáticas***: Su tamaño en memoria es fijo. Ejemplo: Tablas (arrays).
- ***Dinámicas***: Su tamaño en memoria es variable. Ejemplo: listas enlazadas con punteros, árboles, grafos, ficheros, etc.

Los tipos abstractos de datos (TAD) permiten describir una estructura de datos en función de las operaciones que pueden efectuar, dejando a un lado su implantación.

Los TAD mezclan estructuras de datos junto a una serie de operaciones de manipulación. Incluyen una especificación, que es lo que verá el usuario, y una implantación (algoritmos de operaciones sobre las estructuras de datos y su representación en un lenguaje de programación), que el usuario no tiene necesariamente que conocer para manipular correctamente los tipos abstractos de datos.

Se caracterizan por el encapsulamiento. Es como una caja negra que funciona simplemente conectándole unos cables. Esto permite aumentar la complejidad de los programas pero manteniendo una claridad suficiente que no desborde a los desarrolladores. Además, en caso de que algo falle será más fácil determinar si lo que falla es la caja negra o son los cables.

Por último, indicar que un TAD puede definir a otro TAD. Por ejemplo, se pueden construir pilas, colas y árboles a partir de arrays y listas enlazadas. De hecho, las listas enlazadas pueden construirse con arrays y viceversa.

Un fichero o archivo es una agrupación de datos interrelacionados que hacen referencia a un tema o temas determinados y que se ubican en uno o más lugares concretos. Un ejemplo de fichero sería un archivador conteniendo los datos relativos a cada uno de los trabajadores de una empresa, para cada trabajador existiría una ficha compuesta de apartados conteniendo información. Estos ficheros son permanentes en el tiempo (guardados en formato papel) y podían ser usados por cualquier persona, siempre y cuando tuviera autorización.

En el mundo informático desde un principio resultó imprescindible poder contar con algo parecido al tradicional concepto de fichero en el mundo real, puesto que los datos que se almacenaban en la memoria principal del ordenador no podían ser persistentes y tampoco ser usados por varios procesos en ejecución. De esta dificultad surgió la necesidad de tener una memoria secundaria, inicialmente utilizando un soporte de tipo cinta magnético, donde guardar la información con las principales características del tradicional fichero del mundo real, la permanencia en el tiempo (persistencia) y la posibilidad de poder ser usado por muchos procesos que se estén ejecutando al mismo tiempo, simplemente leyendo los datos del fichero en la memoria secundaria.

Inicialmente estos ficheros informáticos almacenaban los datos de forma secuencial y por tanto su forma de acceso también lo era. Esto producía una gran ineficiencia (lentitud) a la hora de leer los datos contenidos en el fichero. Con el tiempo fueron surgiendo nuevas formas y estructuras de datos que almacenamientos más óptimos y accesos más directos y eficientes (rápidos) a los datos.

## 6. BIBLIOGRAFÍA


- Fundamentals of data structures. E. Horowitz y S. Sahni. Computer Science Press 1994.
- Algorítmica. Concepción y análisis. G. Brassard y P. Bratley. Ed. Masson 1990.
- The Spirit of Computing. David Harel. 2a. edicion, Addison-Wesley 1992.
- Estructura de la información. G. Pascual Laporta. McGraw-Hill. 1992
- Estructura de la información. Organización de ficheros y datos. F.J. Muñoz López. Paraninfo. 1990
- Modern Operating Systems. A.S. Tanenbaum. Prentice-Hall. 1992.
- Operating Systems Concepts. Silberschatz, Peterson. Addison-Wesley, 1994.
- Sistemas Operativos, Segunda Edición. Deitel, H. M., Addison-Wesley, 1992.
- Operating Systems, Second Edition. Stallings, W. Prentice-Hall. 1995.

## 7. ESQUEMA – RESUMEN

**TABLAS, LISTAS Y ÁRBOLES.**

*Un TAD es una estructura algebraica, a saber, un conjunto de objetos estructurados de alguna forma y con ciertas operaciones definidas sobre ellos*.

Un **algoritmo** es *un conjunto de pasos o instrucciones que se deben seguir para realizar una determinada tarea*.

Estas instrucciones deben cumplir las siguientes características:

- **FINITUD**: C*onjunto finito de instrucciones* que se realicen en un *tiempo finito*.
- **PRECISIÓN**: *Orden de realización de cada instrucción o paso de forma inequívoca*.
- **DEFINICIÓN**: *Número finito de datos de entrada* y un *número finito e datos de salida.*

Uno de los criterios más importantes para seleccionar un algoritmo es evaluar el tiempo que tarda en ejecutarse, o ***coste del algoritmo*** ó ***tiempo de ejecución***. Hay dos casos límite, el ***mejor caso*** y el ***peor caso***.

La *ecuación fundamental* que describe el enfoque de desarrollo con Tipos Abstractos de Datos es:

***Programa = Implantación del TAD + Algoritmo de Control***

Las operaciones con TAD son de tipo:

- **CREACIÓN.**
- **TRANSFORMACIÓN.**
- **ANÁLISIS.**

Los TAD más utilizados son:

- Tablas
- Listas.
- Árboles.

Una ***tabla*** o ***matriz*** es un TAD que modela o representa una estructura homogénea de datos donde se cumple:

- Todos sus *componentes son del mismo tipo*.
- Tiene un *número predefinido* de componentes que no puede variarse en tiempo de ejecución.
- Los elementos de la tabla contienen *una clave* que los identifica de forma unívoca.
- Se permite el *acceso directo* a cualquiera de los elementos de la tabla a través de los índices.

Los tipos de tabla son:

- Tabla **Monodimensional.**
- Tabla **Multidimensional**.

Las operaciones sobre las tablas son:

- ***Definir/crear.***

- □ ***Insertar/Eliminar.***
- □ ***Buscar.***
- □ ***Ordenar.***
- □ **Contar.**

La forma más simple de representación abstracta de una tabla es: Nombre de Tabla = { $e_0$, $e_1$, ..., $e_i$,..., $e_{N-1}$ }

O también

| **TABLA MONODIMENSIONAL DE N ELEMENTOS** | | | | | |
|---|---|---|---|---|---|
| **ÍNDICE** del elemento | *0* | *1* | *2* | ***....*** | *N – 1* |
| **VALOR** del elemento | Juan | Pedro | Inés | **....** | Ana |

Las tablas o matrices se implementan mediante un tipo de estructura de datos denominada *array* (arreglo).

Una ***lista*** es una estructura de datos que cumple:

- □ Todos sus *componentes son del mismo tipo*.
- □ Cada *elemento o nodo va seguido de otro* del mismo tipo o de ninguno.
- □ Sus componentes se almacenan según *cierto orden*.

Una manera de clasificar las listas es:

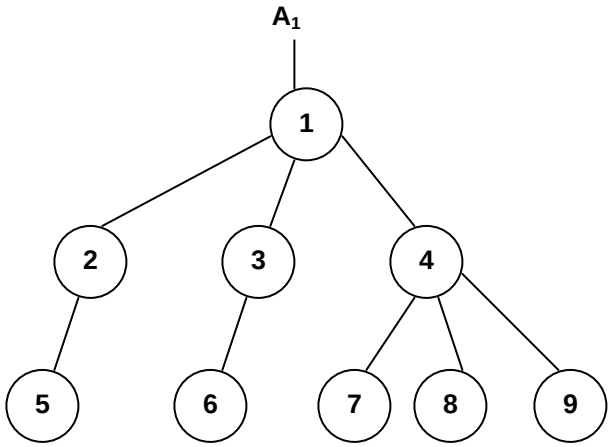
- □ ***Lista densa***: La propia estructura determina cuál es el siguiente elemento de la lista.
- □ ***Lista enlazada***: Cada elemento contiene la información necesaria para llegar al siguiente.

  Dentro de las listas enlazadas:
  - ▫ Lista *simplemente enlazada*.
  - ▫ Lista *doblemente enlazada*.
  - ▫ Lista *con enlaces múltiples*.

La representación mas habitual de una lista es: Nombre de Lista = ( $e_1$, $e_2$, ..., $e_n$ )

Las listas se implementan mediante el uso de punteros.

Un *árbol* es un TAD cuya estructura corresponde con el siguiente gráfico:




**Nivel 0** (Nodo Padre o Raiz)

**Nivel 1** (Nodos hijos -del nodo 1-).

**Nivel 2** (Nodos Nietos).

Existen los siguientes tipos de árboles:

- Árbol **BINARIO** o árbol **B**: El máximo número de nodos hijos que puede tener cualquier nodo es dos.
- Árbol **MULTIRAMA**: No hay límite para el número de nodos hijo.
- Árbol **BALANCEADO**: Entre todos sus nodos hoja no hay una diferencia de nivel superior a la unidad.

Las operaciones básicas que se pueden realizar sobre los árboles son:

- ***Definir/crear***.
- ***Recorrer*** los diferentes caminos del árbol.
  Existen dos formas de recorrer un árbol, a saber:
  - Recorrida en *AMPLITUD*.
    Recorrido en PROFUNDIDAD: en PRE-ORDEN, en POST-ORDEN y en ORDEN-CENTRAL.
- ***Insertar/eliminar***.
- ***Ordenar***.
- ***Buscar***.
- ***Contar***.
- ***Balancear***.

Para implementar los árboles se utiliza normalmente memoria dinámica.

**ALGORITMOS: ORDENACIÓN, BÚSQUEDA, RECURSIÓN, GRAFOS.**

Se pueden distinguir dos tipos de filosofía dentro de los:

- Algoritmos **Recursivos**.
- Algoritmos **Iterativos**.

Atendiendo al tipo de elemento que se quiera ordenar, los algoritmos de ordenación puede ser:

- ***Ordenación interna***.
- ***Ordenación externa***.

Los principales algoritmos de ordenación interna de tipo iterativo son:

- Selección.
- Burbuja.
- Inserción Directa.
- Inserción Binaria.
- Shell.
- Intercalación.

Todos los algoritmos de búsqueda tienen dos finalidades:

- Determinar si el elemento buscado se encuentra en el conjunto en el que se busca.

- Si el elemento está en el conjunto, hallar la posición en la que se encuentra.

Como principales algoritmos en tablas y listas tenemos las búsquedas:

- Secuencial.
- Binaria o Dicotómica.
- Utilizando tablas Hash.

La ***recursividad*** permite que una acción se pueda realizar en función de invocar la misma acción pero en un caso mas sencillo, hasta llegar a un punto (caso base) donde la realización de la acción sea muy sencilla.

Debe cumplirse lo siguiente:

- Identificar subproblemas atómicos de resolución inmediata. Los denominados *casos base*.
- Descomponer el problema en subproblemas resolubles mediante el algoritmo pre-existente; la solución de estos subproblemas debe aproximarnos a los casos base.

La ***Ordenación Rápida (Quicksort)*** es el algoritmo de ordenación ilustrativo del uso de la recursividad.

*"Un grafo G es un par (V(G), A(G)), donde V(G) es un conjunto no vacío de elementos llamados vértices, y A(G) es una familia finita de pares no ordenados de elementos de V(G) llamados aristas".*

Así pues, un grafo consta de:

- ***Vértices*** (o nodos): Los vértices son objetos que contienen información. Para representarlos se suelen utilizar puntos o circunferencias con el contenido del nodo escrito dentro.
- ***Aristas***: Son las conexiones entre vértices. Para representarlos se suelen líneas.

A la hora de explorar el grafo, nos encontramos con dos métodos distintos:

- Exploración o búsqueda en ***Anchura*** o ***Amplitud***: Prima en la exploración la dimensión horizontal.
- Exploración o búsqueda en ***Profundidad.*** Prima en la exploración es la dimensión vertical.

## ORGANIZACIONES DE FICHEROS.

*Un fichero es una abstracción de un mecanismo que permite almacenar información en un dispositivo y leerla posteriormente. Podemos definir un fichero como una colección de información que tiene un nombre.*

Los ficheros son una forma de almacenamiento llamada ***memoria secundaria***. Sus principales cualidades son:

- Capacidad de almacenamiento limitada sólo por el soporte externo utilizado.
- La información está almacenada permanentemente.
- Acceso lento a la información, ya que se debe transportar del dispositivo externo a la memoria principal.

Existen diferentes tipos de ficheros en función de:

- La información contenida
- El método de organización de la información.

Otra clasificación de los ficheros se puede hacer según el método usado para codificarla información:

- ***Ficheros de texto***.
- ***Ficheros binarios***.

Otra clasificación de los fichero es según la forma que tiene su estructura. Las formas mas usuales son:

- Organizar un fichero como ***una secuencia de bytes***.
- Considerar un fichero como ***una secuencia de registros de longitud fija***.
- Otra forma es organizar fichero en forma de ***árbol de registros***.

Las operaciones sobre ficheros son:

- **Creación**.
- **Apertura**.
- **Cierre**.
- **Actualización**.
- **Consulta**.
- **Borrado o destrucción**.
- **Ordenación o clasificación**.
- **Compactación o empaquetamiento**.

La organización de un fichero es muy dependiente del soporte físico en que se almacene. Hay dos tipos:

- Soportes ***Secuenciales***.
- Soportes ***Direccionables***.

Los tipos de organizaciones de ficheros fundamentales son:

- Organización **Secuencial.**
- Organización **Directa** o **Aleatoria.**
- Organización **Indexada.**

Existen tres métodos básicos de asignación en los ficheros:

- Asignación ***Contigua***.
- Asignación ***Enlazada***.
- Asignación ***Indexada***.

Los sistemas de ficheros pueden contener grandes volúmenes de información, por lo que los sistemas operativos proporcionan mecanismos para estructurarlos. El principal de estos mecanismos es el ***directorio***.

## FORMATOS DE INFORMACIÓN Y FICHEROS.

La mayoría de aplicaciones suelen guardar la información que producen en formatos de fichero propios, de modo que podemos editarlos posteriormente con la garantía que se respetarán todas las peculiaridades de los datos y el nivel de edición que poseían en el momento de guardarlos, ahora, cuando compartimos información debemos ser cuidadosos con la elección del tipo de fichero ya que no debemos asumir que todo el mundo posee nuestras mismas herramientas ni nuestro mismo sistema.

Hay dos formatos básicos independientes de la plataforma: **TXT** y **RTF**.

Los archivos gráficos pueden tener una gran variedad de formatos. Entre los más habituales se encuentran: **JPG**, **GIF, PNG, TIFF** y **SVG**.

Atendiendo al uso que se da a un fichero, sin considerar su estructura y organización, se pueden clasificar en:

- *Ficheros **Permanentes***. Entre estos se puede distinguir entre:
  - Ficheros ***maestros***.
  - Ficheros ***constantes***.
  - Ficheros ***históricos***.

- *Ficheros **Temporales**.* Los mas habituales son:
  - Ficheros de ***movimientos***.
  - Ficheros de ***maniobras***.
  - Ficheros de ***resultados***.