Tema 3

Tipos abstractos y estructuras de datos. Organizaciones de ficheros. Algoritmos. Formatos de información y ficheros.

Guion-resumen

1. **Tipos abstractos y estructuras de datos. Tipos abstractos**
   - 1.1. Estructuras de datos
   - 1.2. Tipos abstractos
2. **Organización de ficheros**
   - 2.1. Conceptos
   - 2.2. Clasificación de ficheros según acceso
   - 2.3. Organización de archivos
   - 2.4. Asignación del espacio de almacenamiento
   - 2.5. Métodos de acceso en los sistemas de archivos
   - 2.6. Operaciones soportadas por el subsistema de archivos
   - 2.7. Algunas facilidades extras de los sistemas de archivos
3. **Algoritmos**
   - 3.1. Introducción a los algoritmos
   - 3.2. Complejidad algorítmica
   - 3.3. Algoritmos de ordenación
   - 3.4. Algoritmos de búsqueda
4. **Formatos de información y ficheros**
   - 4.1. Codificaciones de caracteres alfanuméricos
   - 4.2. Codificaciones de números
   - 4.3 Sistemas de archivos

1. Tipos abstractos y estructuras de datos. Tipos abstractos

1.1. Estructuras de datos

Los sistemas o métodos de organización de datos que permiten un almacenamiento eficiente de la información en la memoria principal son conocidos como estructuras de datos. Un lenguaje de programación contiene una serie de tipos de datos simples: enteros, reales, carácter, booleanos,... y otros no simples, como los punteros o los objetos. Podríamos decir pues, que **una estructura de datos es un conjunto de variables de un determinado tipo agrupados y organizados para representar un comportamiento**. Pretenden facilitar el esquema lógico para manipular los datos en función del problema y del algoritmo. Según su comportamiento durante la ejecución del programa, existen los siguientes tipos de datos:

— **Estáticos:** tamaño de memoria fijo definido en tiempo de compilación.

— **Dinámicos:** tamaño de memoria definido en tiempo de ejecución.

1.1.1. Array

Se trata de una secuencia contigua de un número fijo de elementos (estáticos) homogéneos (del mismo tipo). Dichos elementos se almacenarán en memoria en posiciones contiguas. El acceso se realiza por índices. Los arrays de una dimensión se les llama **vectores**, de dos se denominan **matrices** y, en general, de más dimensiones se les llama **poliedros**, aunque la mayoría de los programadores les siguen llamando matrices. Pueden tener tantas dimensiones como se desee. Por tanto, desde el punto de vista del programa, **un array (matriz o vector) es una zona de almacenamiento contigua**.

Pueden estar definidos por un solo índice o por varios, al número de índices se le llama dimensión: ej. int x [5]; o int x [2] [6] [5];

Como vemos la sintaxis usada sería: tipo nombre-array[m1] [m2]... [mx]; donde x sería la dimensión y m1*m2* mx sería el tamaño.

Nota 1: en Java es posible hacer crecer el tamaño de un array de manera dinámica a través de la clase Vector y en Visual Basic a través de Variant.

Nota 2: se permite definir en todos los lenguajes orientados a objetos arrays de objetos, con los cuales afirmar que los arrays están solo compuestos por tipos simples sería muy arriesgado.

1.1.2. Punteros

Un **puntero es una variable que almacena la dirección de memoria de otra variable**, es decir, almacena el valor del lugar físico en la memoria en donde se encuentra almacenada dicha variable. Para operar con ellos recurrimos dependiendo del lenguaje a unos operadores u otros, pero por ejemplo

en C y C++, usamos & para acceder a la dirección de memoria ocupada por el puntero y * para acceder al valor de la variable apuntada por el puntero. Ejemplo:

PROGRAMA	RESULTADOS
int *punt;	
int x=8;	
punt=&x;	
imprime (punt);	8E4
imprime (&punt);	8E33
imprime (*punt);	8
imprime (x);	8
imprime (&x);	8E4

MEMORIA PRINCIPAL	
Dirección de Memoria	**Contenido**
8E4	8
8E33	8E4

Una **referencia** es una variable que almacena la dirección de memoria en donde se ubica un objeto. Nótese que si bien la definición es prácticamente idéntica a la de puntero, la diferencia radica en que una referencia solo puede apuntar a objetos residentes en memoria. A partir de esta definición se puede concluir que toda variable en Java, que no sea de tipo primitivo, es una referencia. Por ejemplo, todas las clases en Java heredan de la clase Object. Una instancia de esta clase se declara como: Object aux=new Object();

1.1.3. Listas

Se trata de una estructura de datos secuencial. Podemos clasificarlas en:

A) Lista densa

La estructura de la lista determina cuál es el siguiente elemento de la lista (Ej. Array).

B) Lista enlazada

Es dinámica. La posición del siguiente elemento la determina el elemento actual. Se trata de una serie de nodos conectados entre sí a través de una referencia. Es necesario almacenar al menos la posición de memoria del primer elemento. Se pueden reorganizar sus elementos y al ser dinámicas cambiar su tamaño en tiempo de ejecución. **La lista enlazada es un Tipo Abstracto de Dato (TDA)** que nos permite almacenar datos de una forma organizada, al igual que los vectores, pero, a diferencia de estos, esta estructura es dinámica, por lo que no tenemos que saber a priori los elementos que puede contener. En una lista enlazada, cada elemento

apunta al siguiente excepto el último que no tiene sucesor y el valor del enlace es null. Por ello los elementos son registros que contienen el dato a almacenar y un enlace al siguiente elemento. Los elementos de una lista suelen recibir también el nombre de nodos de la lista.

- **Las listas enlazadas simples**

Cada elemento solo contiene una referencia al nodo siguiente y solo puede recorrerse en un solo sentido.

Ejemplo: En C:

```
struct  lolo{
       int a;
       lolo *punt;
}
```

- **Las listas enlazadas dobles**

Se trabaja con dos referencias por nodo, uno al siguiente y otro al anterior, lo que permite recorrer la lista en ambos sentidos. Son más rápidas que las anteriores, pero ocupan más.

Ejemplo: En C:

```
struct  lolo{
       int a;
       lolo *punt1;
       lolo *punt2;
}
```

C) Listas ordenadas

La posición de cada elemento depende de su contenido. Cuando hay que insertar un nuevo elemento hay que hacerlo en el lugar que le corresponda dependiendo del orden y de la clave escogidos; para ello se deben de realizar los siguientes pasos:

— Localizar el lugar correspondiente del elemento a insertar.

— Reservar memoria.

— Enlazarlo.

D) Listas reorganizables

Son aquellas en las que cada vez que se accede a un elemento, este se coloca al comienzo de la lista.

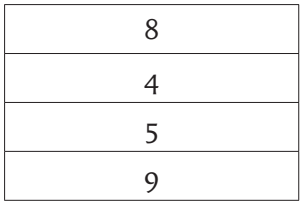
E) Listas circulares

Son aquellas en las que el último elemento tiene un enlace al primero. Su uso está relacionado con las colas.

1.1.4. Pilas

Es una lista de elementos en la cual solo se puede extraer el último elemento insertado. Se trata de una estructura de datos de acceso restrictivo a sus elementos. La posición del último elemento se denomina tope de la pila. Es una estructura de tipo LIFO *(Last In- First Out)*. Las cuatro operaciones básicas que se pueden realizar son: inicializar, apilar, desapilar y vacío. En el ejemplo imaginemos que puntero1 es un puntero al primer elemento que se va a apilar a la lista y puntero2 al que se va a desapilar.

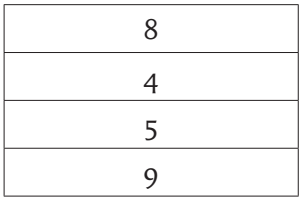
puntero1 **puntero2**



1.1.5. Colas

Se trata de otro tipo de estructura de acceso restrictivo a sus elementos. Es una estructura FIFO *(First In-First Out)* en donde las operaciones básicas son: inicializar, encolar, desencolar y vaciar. En este tipo de listas se insertan pues los nuevos elementos al final de la lista y se extraen desde el inicio de la lista. En el ejemplo imaginemos que puntero1 es un puntero al primer elemento que se va a apilar a la lista y puntero2 al que se va a desapilar.

puntero1



puntero2

1.1.6. Árboles

Un árbol es una estructura de datos que puede definirse de forma recursiva como:

— Una estructura vacía.

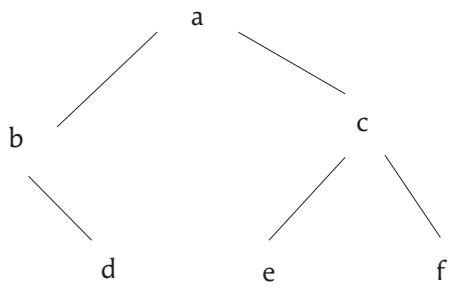
— Un elemento o clave de información (nodo) más un número finito de estructuras tipo árbol, disjuntos, llamados subárboles. Si dicho número de estructuras es inferior o igual a 2, se tiene un árbol binario.

Es, por tanto, una estructura no secuencial.

Otra definición nos da el árbol como un tipo de grafo: un árbol es un grafo acíclico, conexo y no dirigido. Es decir, es un grafo no dirigido en el que existe exactamente un camino entre todo par de nodos. Esta definición permite implementar un árbol y sus operaciones empleando las representaciones que se utilizan para los grafos.

- **Formas de representación**

— Mediante un grafo:



— Mediante un diagrama encolumnado:

a		
	b	
		d
	c	
		e
		f

En la computación se utiliza mucho una estructura de datos, que son los árboles binarios. Estos árboles tienen 0, 1 o 2 descendientes como máximo.

A) Nomenclatura sobre árboles

— **Raíz:** es aquel elemento que no tiene antecesor; ejemplo: a.

— **Subárbol:** subconjunto de elementos de un árbol con estructura de árbol.

— **Rama:** arista entre dos nodos.

— **Antecesor o Padre:** un nodo X es antecesor de un nodo Y si por alguna de las ramas de X se puede llegar a Y.

— **Sucesor o Hijo:** un nodo X es sucesor de un nodo Y si por alguna de las ramas de Y se puede llegar a X.

— **Grado de un nodo:** el número de descendientes directos que tiene. Ejemplo: c tiene grado 2, d tiene grado 0, a tiene grado 2.

— **Hoja:** nodo que no tiene descendientes: grado 0. Ejemplo: d.

— **Nivel:** número de ramas que hay que recorrer para llegar de la raíz a un nodo. Ejemplo: el nivel del nodo a es 1 (es un convenio), el nivel del nodo e es 3.

— **Altura:** el nivel más alto del árbol. En el ejemplo 1 la altura es 3.

— **Anchura:** es el mayor valor del número de nodos que hay en un nivel. En el ejemplo la anchura es 3.

— **Grado de un árbol:** es el máximo de los grados de todos sus nodos.

B) Declaración de árbol binario

Se definirá el árbol con una clave de tipo entero (puede ser cualquier otro tipo de datos) y dos hijos: izquierdo (izq) y derecho (der). Para representar los enlaces con los hijos se utilizan punteros. El árbol vacío se representará con un puntero nulo.

C) Recorridos sobre árboles binarios

Se consideran dos tipos de recorrido: recorrido en profundidad y recorrido en anchura o a nivel. Puesto que los árboles no son secuenciales como las listas, hay que buscar estrategias alternativas para visitar todos los nodos.

- **Recorridos en profundidad**

— Recorrido en **preorden**: consiste en acceder al nodo actual, después acceder al subárbol izquierdo y una vez accedido, acceder al subárbol derecho y así sucesivamente, de manera recursiva hasta que se llegue a los nodos hoja del árbol. En el ejemplo sería en el orden siguiente: a,b,d,c,e,f.

```
procedure preorden(Arbol: tArbol);
begin
     if Arbol<> nil then
           begin
```

```
                write(Arbol^.info, '-');
                preorden(Arbol^.iz);
                preorden(Arbol^.de)
        end
end;
```

— Recorrido en **inorden** u orden central: se accede al subárbol izquierdo, al nodo actual, y después se visita el subárbol derecho y así sucesivamente, de manera recursiva hasta que se llegue a los nodos hoja del árbol. En el ejemplo sería en este orden: b, d, a, e, c, f.

```
procedure inorden(Arbol: tArbol);
begin
     if Arbol<> nil then
           begin
                inorden(Arbol^.iz);
                write(Arbol^.info, '-');
                inorden(Arbol^.de)
        end
end;
```

— Recorrido en **postorden**: se accede primero al subárbol izquierdo, después al subárbol derecho, y por último al nodo actual y así sucesivamente, de manera recursiva hasta que se llegue a los nodos hoja del árbol. En el ejemplo sería: d, b, e, f, c, a.

```
procedure postorden(Arbol: tArbol);
begin
     if Arbol<> nil then
           begin
                postorden(Arbol^.iz);
                postorden(Arbol^.de)
                write(Arbol^.info, '-');
        end
end;
```

- **Recorrido en amplitud**

Consiste en ir visitando el árbol por niveles. Primero se visitan los nodos de nivel 1 (la raíz), después los nodos de nivel 2, así hasta que ya no queden más.

Si se hace el recorrido en amplitud del árbol de la figura se visitarían los nodos en este orden: a, b, c, d, e, f.

En este caso el recorrido no se realizará de forma recursiva sino iterativa, utilizando una cola como estructura de datos auxiliar. El procedimiento consiste en encolar (si no están vacíos) los subárboles izquierdo y derecho del nodo extraído de la cola, y seguir desencolando y encolando hasta que la cola esté vacía.

```
procedure amplitud (Arbol: tArbol);
varA: tArbol;
cola: tCola;
begin
      Inicia (cola);
      A:= Arbol;
      ifA<>nil then Encolar (cola, A);
      while not EsVacia(cola) do
      begin
            Desencolar (cola, A);
            write(A^.info, '—');
            if A^.iz<> nil then Encolar (cola, A^.iz);
            if A^.de <> nil then Encolar (cola, A^.de)
      end
end;
```

- **Árboles Binarios de Búsqueda**

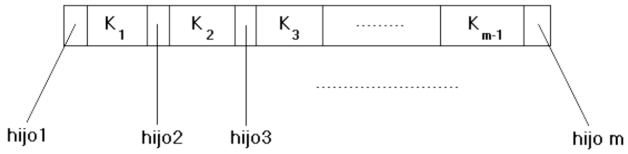
Son árboles con la particularidad de que cada nodo solo puede tener como máximo dos nodos hijo, de ahí su nombre, binario y además las inserciones se realizan de manera ordenada en aras de mejorar futuras búsquedas sobre el mismo. Las operaciones que se pueden realizar son la ya mencionada inserción, borrado y búsqueda.

— **Árboles Equilibrados en Altura:** AVL (ADELSON-VELSKII y LANDIS, 1962) es un árbol binario en el que la diferencia de alturas de los subárboles izquierdo y derecho correspondientes a cualquier nodo del árbol no es superior a 1. El factor de equilibrio o balance de un nodo se define como la altura del subárbol derecho menos la altura del subárbol izquierdo correspondiente. El factor de equilibrio de cada nodo en un árbol equilibrado

será 1, -1 o 0. Las operaciones que se pueden realizar son igualmente inserción, borrado y búsqueda. Tanto realizar inserciones como borrados no garantizan que el estado final del árbol cumpla con el factor de equilibrio de ahí que haya que recalcularlo mediante lo que se llaman rotaciones de nodos. Las hay simples y dobles.

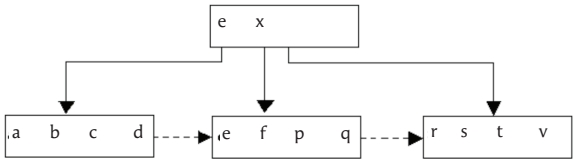
— Árboles B

Los árboles B son árboles cuyos nodos pueden tener un número múltiple de hijos y se dicen árboles B de orden m, siendo m el número máximo de hijos que puede tener un nodo de este árbol, según se muestra en la figura.



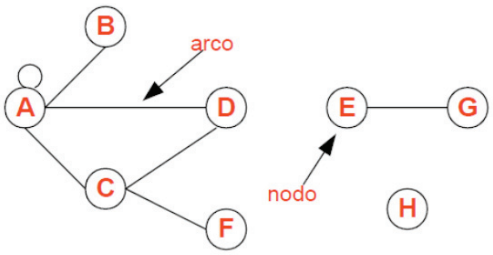
— Árboles B+

Los árboles B+ constituyen una mejora sobre los árboles B, ya que conservan la rapidez de acceso aleatorio y permiten además un recorrido secuencial por sus nodos hoja. En un árbol B+ todas las claves se encuentran en los nodos hojas, duplicándose en la raíz y nodos intermedios aquellas que resulten necesarias para definir los caminos de búsqueda. Para facilitar el recorrido secuencial las hojas se enlazan mediante punteros, obteniéndose, de esta forma, una trayectoria secuencial para recorrer las claves del árbol. Los árboles B+ ocupan algo más de espacio que los árboles B, por las duplicidades expuestas. Los nodos raíz y nodos intermedios son utilizados como índices. Ejemplo de inserción según el siguiente orden: p v d e b c s a r f t q.



— Grafos

Un grafo está formado por un conjunto de nodos (o vértices) y un conjunto de arcos. Cada arco en un grafo se especifica por un par de nodos. El conjunto de nodos es {A, B, C, D, F, G, H} y el conjunto de arcos {(A, B), (A, D), (A, C), (C, D), (C, F), (E, G), (A, A)} para el siguiente grafo:



Si los pares de nodos en los arcos son dirigidos, el grafo se denomina grafo directo, dirigido o dígrafo.

Algunos conceptos básicos referidos a grafos:

— Dos nodos son adyacentes si hay un arco que los une.

— Al número de nodos del grafo se le llama orden del grafo.

— Un grafo nulo es un grafo de orden 0 (cero).

— En un grafo dirigido, si A es adyacente de B, no necesariamente B es adyacente de A.

— Camino es una secuencia de uno o más arcos que conectan dos nodos.

— Un grafo se denomina conectado cuando existe siempre un camino que une dos nodos cualesquiera y desconectado en caso contrario.

— Un grafo es completo cuando cada nodo está conectado con todos y cada uno de los nodos restantes.

— El camino de un nodo se llama ciclo.

— Un grafo que no tiene ningún ciclo es un árbol.

1.2. Tipos abstractos

Un tipo abstracto de datos es un conjunto de valores y de operaciones definidos mediante una especificación independiente de cualquier representación lo que favorece la reusabilidad de un mismo código. Construir aplicaciones muy parecidas para resolver los mismos problemas propició la aparición de la programación orientada a objetos fomentando que el código se pueda reutilizar.

Como antecedente de los tipos de datos abstractos en los lenguajes estructurados podemos mencionar los tipos de datos predefinidos donde un tipo de datos no solo es el conjunto de valores, sino también sus operaciones con sus propiedades.

Los tipos abstractos de datos son pues un conjunto de datos u objetos al cual se le asocian operaciones abstrayéndose de cómo estén materializadas

dichas operaciones, pudiendo ser implementados utilizando distintas estructuras de datos resultando la misma funcionalidad.

Los tipos abstractos básicos son las listas, las pilas y las colas.

— Una **lista s**e define como una serie de N elementos E1, E2, ..., EN, ordenados de manera consecutiva, es decir, el elemento Ek (que se denomina elemento k-ésimo) es previo al elemento Ek+1. Las operaciones que se pueden realizar en la lista son: insertar un elemento en la posición k, borrar el k-ésimo elemento, buscar un elemento dentro de la lista y preguntar si la lista está vacía.

— Una **pila** es una lista de elementos de la cual solo se puede extraer el último elemento insertado. La posición donde se encuentra dicho elemento se denomina tope de la pila. También se conoce a las pilas como listas LIFO.

— Una **cola** es una lista de elementos donde siempre se insertan nuevos elementos al final de la lista y se extraen elementos desde el inicio de la misma. También se conoce a las colas como listas FIFO.

2. Organización de ficheros

2.1. Conceptos

Entendemos como fichero un conjunto ordenado de información, aunque por norma general podríamos decir que es cualquier tipo de información almacenada en un soporte informático.

— **Registro:** conjunto de información relativa a cada uno de los elementos componentes del fichero.

— **Campo:** características particulares e individuales que forman los registros, por lo que en definitiva son los elementos que nos dan la información detallada de cada registro.

2.2. Clasificación de ficheros según acceso

Un sistema de archivos *(file system)* es una estructura de directorios con algún tipo de organización el cual nos permite almacenar, crear y borrar archivos en diferentes formatos. En esta sección se revisarán conceptos importantes relacionados a los sistemas de archivos.

2.2.1. Almacenamiento físico de datos

En un sistema de cómputo es evidente que existe la necesidad, por parte de los usuarios y aplicaciones, de almacenar datos en algún medio, a veces por periodos largos y a veces por instantes. Cada aplicación y cada usuario debe tener ciertos derechos con sus datos, como son el poder crearlos y borrarlos, o cambiarlos de lugar, así como tener privacidad contra otros usuarios o aplicaciones. El

subsistema de archivos del sistema operativo se debe encargar de estos detalles, además de establecer el formato físico en el cual almacenará los datos en discos duros, discos de estado sólido o cintas.

En caso de utilizar discos duros, hay que manejar distintos tiempos, de este modo, al tiempo que una cabeza lectora/escritora tarda en ir de una pista a otra se le llama **tiempo de búsqueda** y dependerá de la distancia entre la posición actual y la distancia a la pista buscada. El tiempo que tarda una cabeza en ir del sector actual al sector deseado se le llama **tiempo de latencia** y depende de la distancia entre sectores y la velocidad de rotación del disco. El impacto que tiene las lecturas y escrituras sobre el sistema está determinado por la tecnología usada en los platos y cabezas y por la forma de resolver las peticiones de lectura y escritura, es decir, los algoritmos de planificación.

2.3. Algoritmos de planificación

Los algoritmos de planificación de peticiones de lectura y escritura a discos se encargan de registrar dichas peticiones y de responderlas en un tiempo razonable. Los algoritmos más comunes para esta tarea son:

— **Primero en llegar, primero en ser servido (FIFO):** las peticiones son encoladas de acuerdo al orden en que llegaron y de esa misma forma se van leyendo o escribiendo las mismas. La ventaja de este algoritmo es su simplicidad y no causa sobrecarga, su desventaja principal es que no aprovecha para nada ninguna característica de las peticiones, de manera que es muy factible que el brazo del disco se mueva muy ineficientemente, ya que las peticiones pueden tener direcciones en el disco unas muy alejadas de otras. Por ejemplo, si se están haciendo peticiones a los sectores 6, 10, 8, 21 y 4, las mismas serán resueltas en el mismo orden.

— **Primero el más cercano a la posición actual**: en este algoritmo las peticiones se ordenan de acuerdo a la posición actual de la cabeza lectora, sirviendo primero a aquellas peticiones más cercanas y reduciendo, así, el movimiento del brazo, lo cual constituye la ventaja principal de este algoritmo. Su desventaja consiste en que puede haber solicitudes que se queden esperando para siempre, en el infortunado caso de que existan peticiones muy alejadas y en todo momento estén entrando peticiones que estén más cercanas. Para las peticiones 6, 10, 8, 21 y 4, las mismas serán resueltas en el orden 4, 6, 8, 10 y 21.

— **Por exploración (algoritmo del elevador):** en este algoritmo el brazo se estará moviendo en todo momento desde el perímetro del disco hacia su centro y viceversa, resolviendo las peticiones que existan en la dirección que tenga en turno. En este caso las peticiones 6, 10, 8, 21 y 4 serán resueltas en el orden 6, 10, 21, 8 y 4; es decir, la posición actual es 6 y como va hacia los sectores de mayor numeración (hacia el centro, por ejemplo), en el camino sigue el sector 10, luego el 21 y ese fue el más central, así que ahora el brazo resolverá las peticiones en su camino hacia afuera y la primera que se encuentra es la del sector 8 y luego la 4. La ventaja de este algoritmo es que el brazo se moverá mucho menos que en FIFO y evita la espera indefi-

nida; su desventaja es que no es justo, ya que no sirve las peticiones en el orden en que llegaron, además de que las peticiones en los extremos interior y exterior tendrán un tiempo de respuesta un poco mayor.

— **Por exploración circular:** es una variación del algoritmo anterior, con la única diferencia de que al llegar a la parte central, el brazo regresa al exterior sin resolver ninguna petición, lo cual proveerá un tiempo de respuesta más cercana al promedio para todas las peticiones, sin importar si están cerca del centro o del exterior.

2.4. Organización de archivos

El subsistema de archivos se debe encargar de localizar espacio libre en los medios de almacenamiento para guardar archivos y para después borrarlos, renombrarlos o agrandarlos. Para ello se vale de localidades especiales que contienen la lista de archivos creados y por cada archivo una serie de direcciones que contienen los datos de los mismos. Esas localidades especiales se llaman directorios y de algún modo han de estar organizados los archivos en ellos contenidos. A continuación se enumeran qué tipos de organizaciones existen.

2.4.1. Organización Secuencial

A) Organización Secuencial Serial

— Cada directorio contiene los nombres de archivos, la dirección del bloque inicial de cada archivo, así como el tamaño de cada uno. Por ejemplo, si un archivo comienza en el sector 17 y mide 10 bloques, cuando el archivo sea accedido, el brazo se moverá inicialmente al bloque 17 y de ahí hasta el 27.

— Si el archivo es borrado y luego creado otro más pequeño, quedarán huecos inútiles entre archivos útiles, produciendo lo que se conoce como fragmentación externa y que para recuperar esos espacios habrá que recurrir a la compactación.

— Las inserciones una vez creadas son costosas, hay que replantear el espacio asignado en función del nuevo tamaño.

— El número de búsquedas y tiempos de búsqueda son mínimos.

— En cuanto a los accesos, permite el acceso secuencial y el acceso directo.

B) Organización Secuencial Encadenada

— Los archivos son listas enlazadas de bloques de datos, es decir, los directorios contienen los nombres de archivos y por cada uno de ellos la dirección del bloque inicial que compone al archivo. Cuando un archivo es leído, el brazo va a esa dirección inicial y encuentra los datos iniciales junto con la dirección del siguiente bloque y así sucesivamente.

— Con este criterio no es necesario que los bloques estén contiguos y no existe la fragmentación externa, pero en cada "eslabón" de la cadena se desperdicia un pequeño espacio con la dirección del siguiente elemento.

2.4.2. Organización Indexada

— En este esquema se guarda en el directorio un bloque de índices para cada archivo, con enlaces o punteros hacia todos sus bloques de datos, de manera que el acceso directo se agiliza notablemente, a cambio de sacrificar varios bloques para almacenar dichos punteros.

— Cuando se quiere leer un archivo o cualquiera de sus partes, se realizan dos accesos: uno al bloque de índices y otro a la dirección deseada. Este es un esquema excelente para archivos grandes pero no para pequeños, porque la relación entre bloques destinados para índices respecto a los asignados para datos es incosteable.

— Soporta el acceso secuencial y directo.

— Áreas que componen este tipo de organización:

- **Área de datos o de registros.** Contiene los registros en un soporte de almacenamiento directo, en orden ascendente, de acuerdo con los valores de la clave y en páginas o bloques de longitud fija.

- **Área de índices.** La crea el sistema operativo al almacenar datos. Contiene una tabla que asocia claves con direcciones del área de datos. Cada entrada está formada por el valor más alto de la clave de cada grupo de registros y un puntero con la dirección del primer registro del grupo.

- **Área de desbordamiento *(overflow)*.** Zona de grabación de registros sin cabida en el área de datos. Los nuevos registros se insertan y enlazan con punteros conservando el orden lógico que marca la clave. El tratamiento de índices y punteros lo realiza el sistema operativo, de forma transparente al usuario.

2.4.3. Organización Directa o Direccionada, Relativa o Aleatoria o Dispersa

— Un archivo está organizado en modo directo cuando el orden físico no se corresponde con el orden lógico. Los datos se sitúan en el archivo y se accede a ellos directa aleatoriamente mediante su posición, es decir, el lugar relativo que ocupan.

— Esta organización tiene la ventaja de que se pueden leer y escribir registros en cualquier orden y posición. Son muy rápidos de acceso a la información que contienen.

— La organización directa tiene el inconveniente de que se necesita programar la relación existente entre el contenido de un registro y

la posición que ocupa. El acceso a los registros en modo directo implica la posible existencia de huecos libres dentro del soporte, y por consecuencia pueden existir huecos libres entre registros.

— La correspondencia entre clave y dirección debe poder ser programada y la determinación de la relación entre el registro y su posición física se obtiene mediante unas funciones de transformación.

— Componentes:

  • Clave de direccionamiento, que permite localizar un registro sobre el dominio de la clave.

  • Dirección relativa: que es una posición ordinal sobre el espacio de direccionamiento posible.

  • Dirección Base que es una dirección física del disco.

— Funciones de Transformación, que serán las encargadas de, según qué método que a continuación se enumeran, obtener la clave de direccionamiento:

  • División-resto o residuo.

  • Truncamiento.

  • Plegado.

  • Cambio de Base.

  • Método de Lin.

— Tipos:

  • Organización Direccionada Directa: cada registro tiene su dirección reservada, esto es, una dirección para cada clave de direccionamiento y la clave de direccionamiento es también clave de identificación. Ejemplo: Buzones de un edificio.

    – Ventajas: la localización es inmediata, es decir, 1 acceso.

    – Desventajas:

      a) Difícil de encontrar una clave y función de transformación adecuadas.

      b) Suelen proporcionar baja densidad de uso.

    – Tipos:

      a) Organización Direccionada Directa Absoluta: la clave de direccionamiento es la dirección base.

b) Organización Direccionada Directa Relativa: la clave de direccionamiento necesita alguna tranformación.

- Organización Direccionada Dispersa o Calculada *(hashing)*: en este caso una dirección corresponde a varias claves de direccionamiento, por ejemplo, las estanterías de una biblioteca en la que varios libros comparten balda de estantería y pertenencia a una signatura, es decir, clave de direccionamiento apareciendo el problema de las colisiones.
  - Métodos de Resolución de Colisiones:
    a) *Hashing* abierto: ante colisiones se establecen listas enlazadas.
    b) *Hashing* cerrado: ante colisiones se define un área donde se almacenan las colisiones.

2.5. Métodos de acceso en los sistemas de archivos

Los métodos de acceso se refieren a las capacidades que el subsistema de archivos provee para acceder a datos dentro de los directorios y medios de almacenamiento en general. Se ubican tres formas generales:

— **Acceso secuencial:** es el método más lento y consiste en recorrer los componentes de un archivo de uno en uno hasta llegar al registro deseado. Se necesita que el orden lógico de los registros sea igual al orden físico en el medio de almacenamiento. Este tipo de acceso se usa comúnmente en discos, cintas y cartuchos usados para la realización de backups. Las primitivas utilizadas son *write* (datos), *read* (datos) y *rewind* en el caso de las cintas.

— **Acceso directo:** permite acceder a cualquier sector o registro inmediatamente ya que no existe ningún orden específico. Este tipo de acceso es rápido y se usa comúnmente en discos duros y discos o archivos manejados en memoria de acceso aleatorio. Las primitivas utilizadas son *write* (posición, datos), *read* (posición, datos) y *seek* (posición).

— **Acceso directo indexado:** este tipo de acceso es útil para grandes volúmenes de información o datos. Consiste en que cada archivo tiene una tabla de enlaces, donde cada apuntador va a la dirección de un bloque de índices y este a su vez enlaces a bloques de datos, lo cual permite que el archivo se expanda a través de un espacio enorme. Consume una cantidad importante de recursos en las tablas de índices pero es muy rápido.

2.6. Operaciones soportadas por el subsistema de archivos

Independientemente de los algoritmos de asignación de espacio, de los métodos de acceso y de la forma de resolver las peticiones de lectura y escritura, el subsistema de archivos debe proveer un conjunto de llamadas al

sistema para operar con los datos y de proveer mecanismos de protección y seguridad. Las operaciones básicas que la mayoría de los sistemas de archivos soportan son:

- **Crear** ***(create)***: permite crear un archivo sin datos, con el propósito de indicar que ese nombre ya está usado y se deben crear las estructuras básicas para soportarlo.
- **Borrar** ***(delete)***: eliminar el archivo y liberar los bloques para su uso posterior.
- **Abrir** ***(open)*****:** antes de usar un archivo se debe abrir para que el sistema conozca sus atributos, tales como el dueño, la fecha de modificación, etc.
- **Cerrar** ***(close)*****:** después de realizar todas las operaciones deseadas, el archivo debe cerrarse para asegurar su integridad y para liberar recursos de su control en la memoria.
- **Leer o escribir** ***(read, write)***: añadir información al archivo o leer el caracter o una cadena de caracteres a partir de la posición actual.
- **Concatenar** ***(append)***: es una forma restringida de la llamada "write", en la cual solo se permite añadir información al final del archivo.
- **Localizar** ***(seek)***: para los archivos de acceso directo se permite posicionar el apuntador de lectura o escritura en un registro aleatorio, a veces a partir del inicio o final del archivo.
- **Leer atributos:** permite obtener una estructura con todos los atributos del archivo especificado, tales como permisos de escritura, de borrado, ejecución, etc.
- **Poner atributos:** permite cambiar los atributos de un archivo, por ejemplo en Unix, donde todos los dispositivos se manejan como si fueran archivos, es posible cambiar el comportamiento de una terminal con una de estas llamadas.
- **Renombrar** ***(rename)*****:** permite cambiarle el nombre e incluso a veces la posición en la organización de directorios del archivo especificado. Los subsistemas de archivos también proveen un conjunto de llamadas para operar sobre directorios, las más comunes son crear, borrar, abrir, cerrar, renombrar y leer. Sus funcionalidades son obvias, pero existen también otras dos operaciones no tan comunes que son la de "crear una liga" y la de "destruir la liga". La operación de crear una liga sirve para que desde diferentes puntos de la organización de directorios se pueda acceder a un mismo directorio sin necesidad de copiarlo o duplicarlo. La llamada a "destruir una liga" lo que hace es eliminar esas referencias, siendo su efecto el de eliminar las ligas y no el directorio real. El directorio real no es eliminado hasta que la llamada a "destruir una liga" se realiza sobre él.

2.7. Algunas facilidades extras de los sistemas de archivos

Algunos sistemas de archivos proveen herramientas al administrador del sistema para facilitarle la vida. Las más notables es la facilidad de compartir archivos y los sistemas de "cotas".

La facilidad de compartir archivos se refiere a la posibilidad de que los permisos de los archivos o directorios dejen que un grupo de usuarios puedan ser accedidos para diferentes operaciones: leer, escribir, borrar, crear, etc. El dueño verdadero es quien decide qué permisos se aplicarán al grupo e, incluso, a otros usuarios que no formen parte de su grupo. La facilidad de "cotas" se refiere a que el sistema de archivos es capaz de llevar un control para que cada usuario pueda usar un máximo de espacio en disco duro. Cuando el usuario excede ese límite, el sistema le envía un mensaje y le niega el permiso de seguir escribiendo, obligándolo a borrar algunos archivos si es que quiere almacenar otros o que crezcan. La versión de Unix SunOs contiene esa facilidad.

3. Algoritmos

3.1. Introducción a los algoritmos

Un algoritmo es un conjunto finito de reglas que dan una secuencia de operaciones para resolver todos los problemas de un tipo dado. El algoritmo debe tener un número finito de pasos, debe definirse de forma precisa para cada paso. El algoritmo tendrá cero o más entradas, y tendrá una o más salidas, en relación con las entradas.

3.2. Complejidad Algorítmica

Todos los algoritmos tienen asociado el concepto de **Complejidad computacional o algorítmica**, que representa la cantidad de recursos que necesita un algoritmo para resolver un problema y se definirán distintos escenarios, el peor caso, es decir, escenario en el que se utilizarán muchos recursos y se invertirá mucho tiempo en alcanzar el resultado, el mejor caso en el que se utilizarán pocos recursos y se invertirá poco tiempo en su resolución y por último el caso promedio, que viene a ser como la media aritmética del peor y del mejor caso. Para esto se usa el concepto de "orden de una función" y se usa la notación O(n).

La mayoría de los problemas tienen un parámetro de entrada que es el número de datos que hay que tratar, esto es, N. La cantidad de recursos del algoritmo es tratada como una función de N. De esta manera puede establecerse un tiempo de ejecución del algoritmo que suele ser proporcional a una de las siguientes funciones:

— **1:** tiempo de ejecución constante. Significa que la mayoría de las instrucciones se ejecutan una vez o muy pocas.

— **logN:** tiempo de ejecución logarítmico. El programa es más lento cuanto más crezca N, pero es inapreciable, pues logN no se duplica hasta que N llegue a N^2.

— **N:** tiempo de ejecución lineal. Un caso en el que N valga 40, tardará el doble que otro en que N valga 20. Un ejemplo sería un algoritmo que lee N números enteros y devuelve la media aritmética.

— **N·logN:** el tiempo de ejecución es N·logN. Es común encontrarlo en algoritmos como Quick Sort y otros del estilo divide y vencerás. Si N se duplica, el tiempo de ejecución es ligeramente mayor del doble.

— **N^2:** tiempo de ejecución cuadrático. Suele ser habitual cuando se tratan pares de elementos de datos, como por ejemplo un bucle anidado doble. Si N se duplica, el tiempo de ejecución aumenta cuatro veces. El peor caso de entrada del algoritmo Quick Sort se ejecuta en este tiempo.

— **N^3:** tiempo de ejecución cúbico. Como ejemplo se puede dar el de un bucle anidado triple. Si N se duplica, el tiempo de ejecución se multiplica por ocho.

— **2^N:** tiempo de ejecución exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Si N se duplica, el tiempo de ejecución se eleva al cuadrado.

3.3. Algoritmos de ordenación

Su finalidad es organizar ciertos datos (arrays o ficheros) en un orden creciente o decreciente mediante una regla prefijada (numérica, alfabética...). Los algoritmos de ordenamiento se pueden clasificar según tres variables:

— La primera y más común es clasificar según **el lugar** donde se realice la ordenación:

  • Algoritmos de ordenamiento interno: los datos se encuentran en memoria (ya sean arrays, listas, etc) y son de acceso aleatorio o directo (se puede acceder a un determinado campo sin pasar por los anteriores).

  • Algoritmos de ordenamiento externo: los datos están en un dispositivo de almacenamiento externo (ficheros) y su ordenación es más lenta que la interna.

— La segunda clasificación se refiere al **tiempo** que tardan en realizar la ordenación, dadas entradas ya ordenadas o inversamente ordenadas:

  • Algoritmos de ordenación natural: tarda lo mínimo posible cuando la entrada está ordenada.

  • Algoritmos de ordenación no natural: tarda lo mínimo posible cuando la entrada está inversamente ordenada.

— La tercera clasificación se refiere a la **estabilidad**: un ordenamiento estable mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Por ejemplo, si una lista ordenada por fecha la reordenamos en orden alfabético con un algoritmo estable, todos los elementos cuya clave alfabética sea la misma quedarán en orden de fecha. Otro caso sería cuando no interesan las mayúsculas y minúsculas, pero se quiere que si una clave aBC estaba antes que AbC, en el resultado ambas claves aparezcan juntas y en el orden original: aBC, AbC. Cuando los elementos son indistinguibles (porque cada elemento se ordena por la clave completa) la estabilidad no interesa. Los algoritmos de ordenamiento que no son estables se pueden implementar de modo de que lo sean. Una manera de hacer esto es modificar

artificialmente la clave de ordenamiento de modo que la posición original en la lista participe del ordenamiento en caso de coincidencia.

A continuación se muestran algunos algunos algoritmos de ordenamiento agrupados según estabilidad tomando en cuenta la complejidad computacional.

ESTABLES	
Nombre	**Complejidad**
Ordenamiento de burbuja *(Bubblesort)*	$O(n^2)$
Burbuja bidireccional *(Cocktail sort)*	$O(n^2)$
Ordenamiento por inserción *(Insertion sort)*	$O(n^2)$
Ordenamiento por casilleros *(Bucket sort)*	O(n)
Counting sort	O(n+k)
Merge sort	O(n log n)
Árbol binario *(Binary tree sort)*	O(n log n)
Pigeonhole sort	O(n+k)
Radix sort	O(nk)
Stupid sort	$O(n^3)$ versión recursiva
Gnome sort	$O(n^2)$

INESTABLES	
Nombre	**Complejidad**
Shell sort	O(n1.25)
Comb sort	O(n log n)
Selection sort	$O(n^2)$
Heapsort	O(n log n)
Smoothsort	O(n log n)
Ordenamiento rápido *(Quicksort)*	Promedio: O(n log n), peor caso: $O(n^2)$¡
Several Unique Sort	Promedio: O(n u), peor caso: $O(n^2)$; u=n; u = número único de registros

3.3.1. Ordenación interna

Se aplican principalmente a arrays unidimensionales. Los principales algoritmos de ordenación interna son:

— **Heapsort:** el ordenamiento por montículos es un algoritmo de ordenación no recursivo, no estable, con complejidad computacional O(n log n). Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego

extraerlos de la cima del montículo de uno en uno para obtener el array ordenado. Se basa su funcionamiento en la propiedad de los montículos por los cuales la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él.

— **El algoritmo de ordenamiento por mezcla:** *(Merge sort)*, es un algoritmo de ordenación estable, recursivo, de complejidad O(n log n). A grandes rasgos, el algoritmo consiste en dividir en dos partes iguales el vector a ordenar, ordenar por separado cada una de las partes, y luego mezclar ambos arrays, manteniendo la ordenación, en un solo array ordenado.

— **Selección:** este método consiste en buscar el elemento más pequeño del array y ponerlo en 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. El tiempo de ejecución está en $O(n^2)$.

— **Burbuja:** *(Bubble Sort)* consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. De este modo en la primera de las iteraciones quedará al final de la lista el elemento mayor y así sucesivamente, de modo que con un array de N elementos quedará ordenado en N-1 iteraciones. Al igual que en el método de la selección, presenta una complejidad de orden cuadrático $O(n^2)$.

Ejemplo de algoritmo de ordenación por medio del método de la burbuja en C:

```
#include <stdio.h>
void main()
int array [N];
int i,j , aux;
for (i=0; i<N-1; i++)
{
     for (j=0; j<N-i-1; j++)
     {
           if(array[j+1]< array[j])
           {
                 aux=array[j+1];
                 array[j+1]=array[j];
                 array[j]=aux;
           }
     }
}
```

— **Inserción directa:** en este método lo que se hace es tener una sublista ordenada de elementos del 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. En el peor de los casos, el tiempo de ejecución es $O(n^2)$. En el mejor caso (cuando la lista ya estaba ordenada), el tiempo de ejecución está en $O(n)$.

— **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, lo que supone un ahorro de tiempo.

— **Shell:** es una mejora del método de inserción directa, utilizado cuando el array 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. El Shell Sort es un algoritmo de ordenación de disminución incremental. El rendimiento de este algoritmo depende del orden de la tabla inicial y va de n2 en el caso peor a n4 / 3 y se puede mejorar.

— **Ordenación rápida *(QuickSort)*:** este método se basa en la táctica "divide y vencerás", que consiste en ir subdividiendo el array en arrays más pequeños, y ordenar estos. Para hacer esta división, se toma un valor del array como pivote, y se mueven todos los elementos menores de 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.

— **Ordenación basadas en urnas *(Binsort y Radixsort)***: estos métodos de ordenación utilizan urnas para depositar en ellas los registros en el proceso de ordenación. En cada recorrido de la lista se depositan en una urna aquellos registros cuya clave tienen una cierta correspondencia con i.

3.4. Algoritmos de búsqueda

La búsqueda de un elemento dentro de un array es una de las operaciones más importantes 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.

3.4.1. Búsqueda secuencial

Consiste en recorrer y examinar cada uno de los elementos del array hasta encontrar el o los elementos buscados, o hasta que se han mirado todos los elementos del array.

```
for(i=j=0;i<N;i++)
   if(array[i]==elemento)    {
      solucion[j]=i;
      j++;     }
```

Este algoritmo se puede optimizar cuando el array está ordenado:

```
for(i=j=0;array[i]<=elemento;i++)
```

o cuando solo interesa conocer la primera ocurrencia del elemento en el array:

```
for(i=0;i<N;i++)
   if(array[i]==elemento)
      break;
```

Si al acabar el bucle, i vale N es que no se encontraba el elemento. El número medio de comparaciones que hay que hacer antes de encontrar el elemento buscado es de (N+1)/2.

3.4.2. Búsqueda binaria o dicotómica

Para utilizar este algoritmo el array debe estar ordenado. La búsqueda binaria consiste en dividir el array por su elemento medio en dos subarrays más pequeños 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 primer subarray, y si es mayor está en el segundo.

Si al final de la búsqueda todavía no lo hemos encontrado, y el subarray a dividir está vacío {}, el elemento no se encuentra en el array. 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 al array en dos más pequeños.

3.4.3. Búsqueda mediante transformación de claves *(hashing)*

Es un método de asignación y de búsqueda, pero que no requiere que los elementos estén ordenados. 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.

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 puedes no saber el número de elementos a almacenar. La función de hash depende de cada problema y de cada finalidad, y se puede 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 el 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.

— **Mitad del cuadrado:** consiste en elevar al cuadrado la clave y coger las cifras centrales.

— **Truncamiento**: consiste en ignorar parte del número y utilizar los elementos restantes como índice.

— **Plegamiento:** consiste en dividir el número en diferentes partes y operar con ellas.

Existen escenarios en los que tras utilizar alguno de los métodos que se acaban de exponer, aparecen las conocidas como colisiones, es decir, dos elementos a almacenar que, tras aplicarle la función hash obtienen el mismo lugar al que ser asignados, de modo que es necesario disponer de un espacio extra de almacenamiento llamado área de desbordamiento. La manera en que se pueden resolver estas colisiones son:

— **Hashing Cerrado o Interno:** el área de desbordamiento se ubica dentro de la tabla, comportándose como un método estático.

— **Hashing Abierto o Externo:** el área de desbordamiento se ubica en listas enlazadas asociadas a cada bloque de la tabla.

4. Formatos de información y ficheros

4.1. Codificaciones de caracteres alfanuméricos

Las codificaciones más habituales son:

— **Código ASCII (*American Standard Code for Information Interchange*).** Publicado en 1963 por ASA (actual ANSI), utiliza 7 bits para codificar información alfanumérica, por tanto puede representar 27 caracteres, es decir, 128.
La extensión de ASCII conocida como ASCII de 8 bits es el estándar ISO-8859-1 que sirve para poder representar caracteres como por ejemplo la "ñ" española.

— **EBCDIC (*Extended Binary Coded Decimal Interchange Code*):** usa 8 bits para cada carácter por lo que se pueden representar 256 caracteres, de los cuales, los 32 primeros son de control. Cada byte se divide en 2 partes, la primera son los llamados bits de zona y los segundos los bits de dígito. Los 128 primeros caracteres son alfanuméricos y los 128 restantes representan caracteres gráficos.

— **UNICODE:** es un estándar internacional del consorcio Unicode. Representa cualquier carácter. Unicode surgió como un ambicioso proyecto para reemplazar los esquemas de codificación de caracteres existentes, muchos de los cuales están muy limitados en tamaño y son incompatibles con entornos plurilingües. Las codificaciones completas se pueden encontrar en:

  http://www.unicode.org/charts. Como máximo necesitan 32 bits:

  - Los primeros 7 bits permiten la compatibilidad con ASCII.
  - Con 1 byte se puede representar el código US-ASCII.
  - Con 2 bytes: caracteres latinos y alfabetos árabes, griego, cirílico, armenio, hebreo, sirio y thaana.
  - Con 3 bytes: resto de caracteres utilizados en todos los lenguajes.
  - Con 4 bytes: caracteres gráficos y poco comunes.

  Diferentes versiones de representación. Las más comunes:

  - UTF-8: códigos de 1 byte, pero son de longitud variable (se pueden utilizar 4 grupos de 1 byte para representar un símbolo).
  - UCS-2: códigos de 2 bytes de longitud fija.
  - UTF-16: códigos de 2 bytes, de longitud variable (se pueden utilizar 2 grupos de 2 bytes para representar un símbolo).
  - UTF-32: códigos de 4 bytes.

4.2. Codificaciones de números

La codificación de números depende del tipo de número a representar, del rango de representación, de la cantidad de número del rango o de la memoria disponible en la máquina.

4.2.1. Codificación de Números Enteros

- **Técnicas**

  — **Signo y Magnitud:** deja un bit para el signo y el resto para la magnitud. El principal problema es que el 0 tiene dos representaciones, la positiva y la negativa. Apenas se usa.

— **Complemento a 1:** en la práctica basta con decir que el complemento a 1 de un número en binario es el resultante de cambiar sus unos por ceros y viceversa.

— **Complemento a 2:** el complemento a 2 de un número binario resulta de sumar 1 al dígito menos significativo del complemento a 1 del número original. Este complemento solo se emplea en los números negativos. Para los números positivos, el complemento a 2 del número es el propio número.

- **Sistemas de Representación**

— **BCD Natural:** utiliza 4 bits para representar el número desaprovechando 6 símbolos. Valores válidos del 0 al 9 desaprovechando del 10 al 16.

— **Exceso a 3:** es igual que el anterior salvo que le suma 3 a cada dígito, de modo que 0 toma como valor 3, 1 toma como valor 4...

— **Aiken:** es un código BCD que realiza un cambio en el valor de los pesos dentro de cada grupo de 4 bits pasando a valer 2, 4, 2, 1 en vez del 8, 4, 2, 1.

— **Gray:** se encarga de diferenciar símbolos consecutivos mediante un bit de diferencia.

Decimal	BCD	Exceso a 3	Gray	Aiken
0	0000	0011	0000	0000
1	0001	0100	0001	0001
2	0010	0101	0011	0010
3	0011	0110	0010	0011
4	0100	0111	0110	0100
5	0101	1000	0111	1011
6	0110	1001	0101	1100
7	0111	1010	0100	1101
8	1000	1011	1100	1110
9	1001	1100	1101	1111
10	No válido		1111	
11	No válido		1110	
12	No válido		1010	
13	No válido		1011	
14	No válido		1001	
15	No válido		1000	

4.2.2. Codificación de Números Reales

La representación de números reales se realiza en coma flotante que es una forma de notación científica usada en los microprocesadores con la cual se pueden representar números racionales extremadamente grandes y pequeños de una manera muy eficiente y compacta, y con la que se pueden realizar operaciones aritméticas. El estándar para la representación en coma flotante es el IEEE 754. Constan de exponente, base y coeficiente.

4.3 Sistemas de archivos

- **Fat (tabla de asignación de archivos)**

El sistema fat era utilizado por DOS/Windows 3.X/W95/W98/me/nt/2000/xp con una limitación para particiones de 2GB.

- **FAT32 (tabla de asignación de archivos 32)**

FAT32 es un sistema de archivos que apareció para solventar las limitaciones de FAT16 manteniendo compatibilidad con él. Tiene un tamaño máximo de archivo de 4 GiB.

- **NTFS (sistema de archivos de nueva tecnología)**

El sistema de archivos de nueva tecnología (NTFS) solo puede ser leído por sistemas operativos de Microsoft tales como Windows 7/8/10. NTFS no se recomienda para su uso en discos de menos de 400 MB, ya que utiliza una gran cantidad de espacio para las estructuras del sistema.

- **Ext4 y Swap**

Los sistemas de archivos Linux Ext4 y Linux Swap se desarrollaron para el sistema operativo Linux (una versión de libre distribución o "freeware" de unix). El sistema operativo Linux ext4 admite un tamaño máximo de disco o de partición de 16 TiB, para el archivo de intercambio de linux se utiliza Linux Swap.

A continuación mostramos una tabla en la que se aprecian los tamaños de los clústeres con respecto a los tamaños de los volúmenes, distinguiendo además en distintos tipos de archivos: fat, fat32, NTFS.

Sistema Ficheros	Longitud máxima de nombre de archivo	Tamaño de Fichero máximo	Tamaño de Volumen máximo
FAT12	255 UTF-16	32 MB	1 MB hasta 32 MB
FAT16	255 UTF-16	2 GB	16 MB hasta 2 GB
FAT32	255 UTF-16	4 GB	512 MB hasta 8 TB
FATX	42 Bytes	2 GB	16 MB hasta 2 GB
MFS	255 Bytes	256 MB	256 MB
HFS	31 Bytes	2 GB	2 TB
HPFS	255 Bytes	2 GB	2 TB
NTFS	255 Caracteres	16 EB	16 EB
UFS1	255 Bytes	4 GB hasta 256 TB	256 TB
UFS2	255 Bytes	512 GB hasta 32 PB	1 YB
ext2	255 Bytes	16 GB hasta 2 TB	2 TB hasta 32 TB
ext3	255 Bytes	16 GB hasta 2 TB	2 TB hasta 32 TB
ext4	255 Bytes	16 GB hasta 16 TB	1 EB
GPFS	255 UTF-8	No limitado	2 PB
GFS	255 Bytes	2 TB hasta 8 EB	2 TB to 8 EB
NILFS	255 Bytes	8 EB	8 EB
ReiserFS	4,032 Bytes/255 Caracteres	8 TB, 4 GB	16 TB
OCFS	255 Bytes	8 TB	8 TB
OCFS2	255 Bytes	4 PB	4 PB
XFS	255 Bytes	8 EB	8 EB
JFS1	255 Bytes	8 EB	512 TB hasta 4 PB
JFS	255 Bytes	4 PB	32 PB
QFS	255 Bytes	16 EB	4 PB
BFS	255 Bytes	12,288 B hasta 260 GB	256 PB hasta 2 EB

1 YoBibyte = 2^{80} bytes.
1 ZebiByte = 2^{70} bytes.