Búsqueda en arrays

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, en cuyo caso la condición de salida cambiaría a:

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

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

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

En este último caso, cuando sólo interesa la primera posición, se puede utilizar un centinela, esto es, dar a la posición siguiente al último elemento de array el valor del elemento, para estar seguro de que se encuentra el elemento, y no tener que comprobar a cada paso si seguimos buscando dentro de los límites del array:

```
array[N]=elemento;
for(i=0;;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.

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. Por ejemplo, para buscar el elemento 3 en el array {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 subarray: {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

subarray: {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 el subarray a dividir está vacio {}, el elemento no se encuentra en el array. La implementación sería:

```
int desde,hasta,medio,elemento,posicion; // desde y
      // hasta indican los límites del array que se está mirando.
int array[N];

// Dar valor a elemento.

for(desde=0,hasta=N-1;desde<=hasta;)
{
   if(desde==hasta) // si el array sólo tiene un elemento:
   {
      if(array[desde]==elemento) // si es la solución:
         posicion=desde; // darle el valor.
      else // si no es el valor:
         posicion=-1; // no está en el array.
      break; // Salir del bucle.
    }
   medio=(desde+hasta)/2; // Divide el array en dos.
   if(array[medio]==elemento) // Si coincide con el central:
   {
      posicion=medio; // ese es la solución
      break; // y sale del bucle.
    }
   else if(array[medio]>elemento) // si es menor:
      hasta=medio-1; // elige el array izquierda.
   else // y si es mayor:
      desde=medio+1; // elige el array de la derecha.
 }
```

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.

Transformación de claves o hashing:

Es un método de búsqueda que aumenta la velocidad 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. 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, por ejemplo coger las 3 últimas cifras para guardar a los empleados

en un array de 1000 elementos. Para buscar a uno de ellos, bastaría con realizar la transformación a su DNI y ver si está o no en el array.

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 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. Por ejemplo, si el número de expediente de un alumno universitario está formado por el año de entrada en la universidad, seguido de un número identificativo de tres cifras, y suponiendo que entran un máximo de 400 alumnos al año, se le asignarían las claves:

  ```
  1998-000 --> 0 = 1998000-1998000
  1998-001 --> 1 = 1998001-1998000
  1998-002 --> 2 = 1998002-1998000
     ...
  1998-399 --> 399 = 1998399-1998000
  1999-000 --> 400 = 1999000-1998000+400
     ...
  yyyy-nnn --> N = yyyynnn-1998000+(400*(yyyy-1998))
  ```

- 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, y es tratado más adelante. Si el número N es el 13, los números siguientes quedan transformados en:

  ```
  13000000 --> 0
  12345678 --> 7
  13602499 --> 1
  71140205 --> 6
  73062138 --> 6
  ```

- Mitad del cuadrado: consiste en elevar al cuadrado la clave y coger las cifras centrales. Este método también presenta problemas de colisión:

  ```
  123*123=15129 --> 51
  136*136=18496 --> 84
  730*730=532900 --> 29
  301*301=90601 --> 06
  625*625=390625 --> 06
  ```

- Truncamiento: consiste en ignorar parte del número y utilizar los elementos restantes como índice. También se produce colisión. Por ejemplo, si un número de 8 cifras se debe ordenar en un array de 1000 elementos, se pueden coger la primer, la tercer y la última cifras para formar un nuevo número:

13000000 --> 100
12345678 --> 138
13602499 --> 169
71140205 --> 715
73162135 --> 715

- Plegamiento: consiste en dividir el número en diferentes partes, y operar con ellas (normalmente con suma o multiplicación). También se produce colisión. Por ejemplo, si dividimos los número de 8 cifras en 3, 3 y 2 cifras y se suman, dará otro número de tres cifras (y si no, se cogen las tres últimas cifras):

  13000000 --> 130=130+000+00
  12345678 --> 657=123+456+78
  71140205 --> 118 --> 1118=711+402+05
  13602499 --> 259=136+024+99
  25000009 --> 259=250+000+09

Pero ahora se nos presenta el problema de qué hacer con las 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 más efectivo es, en vez de crear un array de número, 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 (ver listas).

Ordenación en arrays

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. Por ejemplo, si tenemos el array {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.
```

Si el array tiene N elementos, el número de comprobaciones que hay que hacer es de N*(N-1)/2, luego el tiempo de ejecución está en $O(n^2)$

```
int array[N];
int i,j,menor,aux;

// Dar valores a los elementos del array

for(i=0;i<N-1;i++)
{
  for(j=i+1,menor=i;j<N;j++)
  if(array[j]<array[menor]) // Si el elemento j es menor que el menor:
    menor=j; // el menor pasa a ser el elemento j.
  aux=array[i]; // Se intercambian los elementos
  array[i]=array[menor]; // de las posiciones i y menor
  array[menor]=aux; // usando una variable auxiliar.
}
```

Burbuja:

Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Con el array anterior, {40,21,4,9,10,35}:

```
Primera pasada:
{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:
{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.
```

Ya están ordenados, pero para comprobarlo habría que acabar esta segunda comprobación y hacer una tercera.

Si el array tiene 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 tiempo de ejecución, al igual que en la selección, en $O(n^2)$.

```
int array[N];
int i,j,aux;

// Dar valores a los elementos del array

for(i=0;i<N-1;i++) // Hacer N-1 pasadas.
{
  for(j=0;j<N-i-1;j++) // Mirar los N-i-1 pares.
  {
    if(array[j+1]<array[j]) // Si el elemento j+1 es menor que el
elemento j:
    {
```

```
      aux=array[j+1]; // Se intercambian los elementos
      array[j+1]=array[j]; // de las posiciones j y j+1
      array[j]=aux; // usando una variable auxiliar.
    }
  }
}
```

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. 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} <-- 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} <-- El array 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. 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 $O(n^2)$.

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.

```
int array[N];
int i,j,aux;

// Dar valores a los elementos del array
```

```
for(i=1;i<N;i++) // i contiene el número de elementos de la sublista.
{
  // Se intenta añadir el elemento i.
  aux=array[i];
  for(j=i-1;j>=0;j--) // Se recorre la sublista de atrás a adelante
para buscar
  {                   // la nueva posición del elemento i.
    if(aux>array[j])  // Si se encuentra la posición:
    {
      array[j+1]=aux; // Ponerlo
      break;          // y colocar el siguiente número.
    }
    else              // si no, sigue buscándola.
      array[j+1]=array[j];
  }
  if(j==-1)     // si se ha mirado todas las posiciones y no se ha
encontrado la correcta
    array[0]=aux; // es que la posición es al principio del todo.
}
```

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 (ver algoritmos de búsqueda), 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.

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.

Por ejemplo, los 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.

```
int array[N];
int salto,cambios,aux,i;

for(salto=N/2;salto!=0;salto/=2) // El salto va desde N/2 hasta 1.
{
  for(cambios=1;cambios!=0; )    // Mientras se intercambie algún
elemento:
  {
    cambios=0;
    for(i=salto;i<N;i++)          // se da una pasada
      if(array[i-salto]>array[i]) // y si están desordenados
      {
        aux=array[i];             // se reordenan
        array[i]=array[i-salto];
        array[i-salto]=aux;
        cambios++;                // y se cuenta como cambio.
      }
  }
```

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 éstos. Para hacer esta división, se toma un valor del array 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 encuantra 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.
```

La implementación es claramente recursiva, y suponiendo el pivote el primer elemento del array, el programa sería:

```
#include <stdio.h>

void ordenar(int *,int,int);

void main()
{
  // Dar valores al array

  ordenar(array,0,N-1); // Para llamar a la función
```

```
}

void ordenar(int *array,int desde,int hasta)
{
  int i,d,aux; // i realiza la búsqueda de izquierda a derecha
               // y j realiza la búsqueda de derecha a izquierda.
  if(desde>=hasta)
    return;

  for(i=desde+1,d=hasta; ; ) // Valores iniciales de la búsqueda.
  {
    for( ;i<=hasta && array[i]<=array[desde];i++); // Primera búsqueda
    for( ;d>=0 && array[d]>=array[desde];d--);     // segunda búsqueda
    if(i<d)                  // si no se han cruzado, intercambiar
    {
      aux=array[i];
      array[i]=array[d];
      array[d]=aux;
    }
    else                     // si se han cruzado, salir del bucle
      break;
  }
  if(d==desde-1) // Si la segunda búsqueda se sale del array es que el
    d=desde;     // pivote es el elemento más pequeño: se cambia con
él mismo
  aux=array[d];  // Colocar el pivote en su posición
  array[d]=array[desde];
  array[desde]=aux;

  ordenar(array,desde,d-1); // Ordenar el primer subarray.
  ordenar(array,d+1,hasta); // Ordenar el segundo subarray.
}
```

En C hay una función que realiza esta ordenación sin tener que implementarla, llamada qsort (incluida en stdlib.h):

```
qsort(nombre_array,número,tamaño,función);
```

donde nombre_array es el nombre del array a ordenar, número es el número de elementos del array, tamaño indica el tamaño en bytes de cada elemento y función es un puntero a una función que hay que implementar, que recibe dos elementos y devuelve 0 si son iguales, algo menor que 0 si el primero es menor que el segundo, y algo mayor que 0 si el segundo es menor que el primero. Por ejemplo, el programa de antes sería:

```
#include <stdio.h>
#include <stdlib.h>

int funcion(const void *,const void *);

void main()
{
  // Dar valores al array

  qsort(array,N,sizeof(array[0]),funcion);
}

int funcion(const void *a,const void *b)
{
  if(*(int *)a<*(int *)b)
```

```
    return(-1);
  else if(*(int *)a>*(int *)b)
    return(1);
  else
    return(0);
}
```

Claramente, es mucho más cómodo usar qsort que implementar toda la función, pero hay que tener mucho cuidado con el manejo de los punteros en la función, sobre todo si se está trabajando con estructuras.

Intercalación:

No es propiamente un método de ordenación, consiste en la unión de dos arrays ordenados de modo que la unión esté también ordenada. Para ello, basta con recorrer los arrays 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 los arrays {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}
```

```
int i1,i2,is;
int array1[N1],array2[N2],suma[N1+N2];

for(i1=i2=is=0;i1<N1 && i2<N2;is++) // Mientras no se me acabe ni
array1 ni array2:
{
  if(array1[i1]<array2[i2]) // Si el elemento de array1 es menor:
  {
    suma[is]=array1[i1];    // se utiliza el de array1.
    i1++;
  }
  else                 // Pero si el elemento de array2 es menor:
  {
    suma[is]=array2[i2];    // se utiliza el de array2.
    i2++;
  }
}
for( ;i1<N1;i1++,is++)  // Añadir los elementos de array1 (si quedan).
  suma[is]=array1[i1];
for( ;i2<N2;i2++,is++)  // Añadir los elementos de array2 (si quedan).
  suma[is]=array2[i2];
```

Operaciones sobre listas

Operaciones básicas:

Inserción al comienzo de una lista:

Es necesario utilizar una variable auxiliar, que se utiliza para crear el nuevo nodo mediante la reserva de memoria y asignación de la clave. Posteriormente es necesario reorganizar los enlaces, es decir, el nuevo nodo debe apuntar al que era el primer elemento de la lista y a su vez debe pasar a ser el primer elemento.
En el siguiente ejemplo se muestra un programa que crea una lista con cuatro números. Notar que al introducir al comienzo de la lista, los elementos quedan ordenados en sentido inverso al de su llegada. Notar también que se ha utilizado un puntero auxiliar **p** para mantener correctamente los enlaces dentro de la lista.

```
#include <stdlib.h>

struct lista
{
  int clave;
  struct lista *sig;
};

int main(void)
{
  struct lista *L;
  struct lista *p;
  int i;
  L = NULL; /* Crea una lista vacia */

  for (i = 4; i >= 1; i--)
  {
    /* Reserva memoria para un nodo */
    p = (struct lista *) malloc(sizeof(struct lista));
    p->clave = i; /* Introduce la informacion */

    p->sig = L; /* reorganiza */
    L = p;      /* los enlaces */
  }
  return 0;
}
```

Recorrido de una lista.

La idea es ir avanzando desde el primer elemento hasta encontrar la lista vacía. Antes de acceder a la estructura lista es fundamental saber si esa estructura existe, es decir, que no está vacía. En el caso de estarlo o de no estar inicializada es posible que el programa falle y sea difícil detectar donde, y en algunos casos puede abortarse inmediatamente la ejecución del programa, lo cual suele ser de gran ayuda para la depuración.
Como se ha dicho antes, la lista enlazada es una estructura recursiva, y una posibilidad para su recorrido es hacerlo de forma recursiva. A continuación se expone el código de un programa que muestra el valor de la clave y almacena la suma de todos los valores en una variable pasada por referencia (un puntero a entero). Por el hecho de ser un proceso recursivo se utiliza un procedimiento para

hacer el recorrido. Nótese como antes de hacer una operación sobre el elemento se comprueba si existe.

```
int main(void)
{
  struct lista *L;
  struct lista *p;
  int suma;
  L = NULL;
  /* crear la lista */
  ...

  suma = 0;
  recorrer(L, &suma);
  return 0;
}

void recorrer(struct lista *L, int *suma)
{
  if (L != NULL) {
    printf("%d, ", L->clave);
    *suma = *suma + L->clave;
    recorrer(L->sig, suma);
  }
}
```

Sin embargo, a la hora de hacer un programa, es más eficaz si el recorrido se hace de forma iterativa. En este caso se necesita una variable auxiliar que se desplace sobre la lista para no perder la referencia al primer elemento. Se expone un programa que hace la misma operación que el anterior, pero sin recursión.

```
int main(void)
{
  struct lista *L;
  struct lista *p;
  int suma;
  L = NULL;
  /* crear la lista */
  ...
  p = L;
  suma = 0;
  while (p != NULL) {
    printf("%d, ", p->clave);
    suma = suma + p->clave;
    p = p->sig;
  }
  return 0;
}
```

A menudo resulta un poco difícil de entender la instrucción p = p->sig; Simplemente cambia la dirección actual del puntero p por la dirección del siguiente enlace. También es común encontrar instrucciones del estilo: p = p->sig->sig; Esto puede traducirse en dos instrucciones, de la siguiente manera:
p = p->sig;
p = p->sig;
Obviamente sólo debe usarse cuando se sepa que p->sig es una estructura no vacía, puesto que si fuera vacía, al hacer otra vez p = p->sig se produciría una referencia a memoria no válida.

Operaciones sobre listas ordenadas:

Las listas ordenadas son aquellas en las que la posición de cada elemento depende de su contenido. Por ejemplo, podemos tener una lista enlazada que contenga el nombre y apellidos de un alumno y queremos que los elementos -los alumnos- estén en la lista en orden alfabético.

La creación de una lista ordenada es igual que antes:

```
struct lista *L;
L = NULL;
```

Cuando haya que insertar un nuevo elemento en la lista ordenada hay que hacerlo en el lugar que le corresponda, y esto depende del orden y de la clave escogidos. Este proceso se realiza en tres pasos:
1.- Localizar el lugar correspondiente al elemento a insertar. Se utilizan dos punteros: *anterior* y *actual*, que garanticen la correcta posición de cada enlace.
2.- Reservar memoria para él (puede hacerse como primer paso). Se usa un puntero auxiliar (*nuevo*) para reservar memoria.
3.- Enlazarlo. Esta es la parte más complicada, porque hay que considerar la diferencia de insertar al principio, no importa si la lista está vacía, o insertar en otra posición. Se utilizan los tres punteros antes definidos para actualizar los enlaces.

A continuación se expone un programa que realiza la inserción de un elemento en una lista ordenada. Suponemos claves de tipo entero ordenadas ascendentemente.

```
#include <stdio.h>
#include <stdlib.h>

struct lista
{
  int clave;
  struct lista *sig;
};

  /* prototipo */
void insertar(struct lista **L, int elem);

int main(void)
{
  struct lista *L;
  L = NULL;  /* Lista vacia */

    /* para probar la insercion se han tomado 3 elementos */
  insertar(&L, 0);
  insertar(&L, 1);
  insertar(&L, -1);
  return 0;
}
void insertar(struct lista **L, int elem)
{
  struct lista *actual, *anterior, *nuevo;

  /* 1.- se busca su posicion */
  anterior = actual = *L;
  while (actual != NULL && actual->clave < elem) {
    anterior = actual;
    actual = actual->sig;
  }
```

```
  /* 2.- se crea el nodo */
  nuevo = (struct lista *) malloc(sizeof(struct lista));
  nuevo->clave = elem;

  /* 3.- Se enlaza */
  if (anterior == NULL || anterior == actual) { /* inserta al
principio */
    nuevo->sig = anterior;
    *L = nuevo;  /* importante: al insertar al principio actuliza la
cabecera */
  }
  else {                        /* inserta entre medias o al final */
    nuevo->sig = actual;
    anterior->sig = nuevo;
  }
}
```

Se puede apreciar que se pasa la lista L con el parámetro **L . La razón para hacer esto es que cuando se inserta al comienzo de la lista (porque está vacía o es donde corresponde) se cambia la cabecera.

A continuación se explica el borrado de un elemento. El procedimiento consiste en localizarlo y borrarlo si existe. Aquí también se distingue el caso de borrar al principio o borrar en cualquier otra posición. Se puede observar que el algoritmo no tiene ningún problema si el elemento no existe o la lista está vacía.

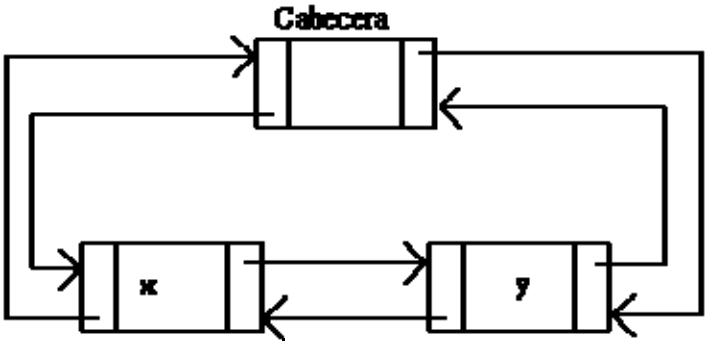
```
void borrar(struct lista **L, int elem)
{
  struct lista *actual, *anterior;
  /* 1.- busca su posicion. Es casi igual que en la insercion, ojo al
(<) */
  anterior = actual = *L;
  while (actual != NULL && actual->clave   < elem) {
      anterior = actual;
      actual = actual->sig;
  }

  /* 2.- Lo borra si existe */
  if (actual != NULL && actual->clave == elem) {
    if (anterior == actual)        /* borrar el primero */
      *L = actual->sig;   /* o tambien (*L)->sig; */
    else                           /* borrar en otro sitio */
      anterior->sig = actual->sig;
    free(actual);
  }
}
```

Operaciones sobre listas doblemente enlazadas:

Son listas que tienen un enlace con el elemento siguiente y con el anterior. Una ventaja que tienen es que pueden recorrerse en ambos sentidos, ya sea para efectuar una operación con cada elemento o para insertar/actualizar y borrar. La otra ventaja es que las búsquedas son algo más rápidas puesto que no hace falta hacer referencia al elemento anterior. Su inconveniente es que ocupan más memoria por nodo que una lista simple.

Se realizará una implementación de lista **ordenada** con doble enlace que aproveche el uso de la cabecera y el centinela. A continuación se muestra un

gráfico que muestra una lista doblemente enlazada con cabecera y centinela, para lo que se utiliza un único nodo que haga las veces de cabecera y centinela.



Declaración:

```
struct listaDE
{
  int clave;
  struct listaDE *ant,
                 *sig;
};
```

Procedimiento de creación:

```
void crearDE(struct listaDE **LDE)
{
  *LDE = (struct listaDE *) malloc(sizeof(struct listaDE));
  (*LDE)->sig = (*LDE)->ant = *LDE;
}
```

Procedimiento de inserción:

```
void insertarDE(struct listaDE *LDE, int elem)
{
  struct listaDE *actual, *nuevo;

  /* busca */
  actual = LDE->sig;
  LDE->clave = elem;

  while (actual->clave < elem)
    actual = actual->sig;

  /* crea */
  nuevo = (struct listaDE *) malloc(sizeof(struct listaDE));
  nuevo->clave = elem;

  /* enlaza */
  actual->ant->sig = nuevo;
  nuevo->ant = actual->ant;
  nuevo->sig = actual;
  actual->ant = nuevo;
}
```

Procedimiento de borrado:

```
void borrarDE(struct listaDE *LDE, int elem)
{
  struct listaDE *actual;

  /* busca */
  actual = LDE->sig;
  LDE->clave = elem;

  while (actual->clave < elem)
    actual = actual->sig;

  /* borra */
  if (actual != LDE && actual->clave == elem) {
    actual->sig->ant = actual->ant;
    actual->ant->sig = actual->sig;
    free(actual);
  }
}
```

Algoritmo de ordenación:

Consta de dos partes, una parte de intercalación de listas y otra de divide y vencerás.

- Primera parte: ¿Cómo intercalar dos listas ordenadas en una sola lista ordenada de forma eficiente?

Suponemos que se tienen estas dos listas de enteros ordenadas ascendentemente:

lista *1*: 1 -> 3 -> 5 -> 6 -> 8 -> 9
lista *2*: 0 -> 2 -> 6 -> 7 -> 10

Tras mezclarlas queda:

lista: 0 -> 1 -> 2 -> 3 -> 5 -> 6 -> 6 -> 7 -> 8 -> 9 -> 10

Esto se puede realizar mediante un único recorrido de cada lista, mediante dos punteros que recorren cada una. En el ejemplo anterior se insertan en este orden -salvo los dos 6 que puede variar según la implementación-: 0 (lista 2), el 1 (lista 1), el 2 (lista 2), el 3, 5 y 6 (lista 1), el 6 y 7 (lista 2), el 8 y 9 (lista 1), y por llegar al final de la lista 1, se introduce directamente todo lo que quede de la lista 2, que es el 10.

En la siguiente implementación no se crea una nueva lista realmente, sólo se *modifican* los enlaces destruyendo las dos listas y fusionándolas en una sola. Se emplea un centinela que apunta a sí mismo y que contiene como clave el valor más grande posible. El último elemento de cada lista apuntará al centinela, incluso si la lista está vacía.

```
struct lista
{
  int clave;
  struct lista *sig;
};


struct lista *centinela;
centinela = (struct lista *) malloc(sizeof(struct lista));
```

```
centinela->sig = centinela;
centinela->clave = INT_MAX;
...

struct lista *fusion(struct lista *l1, struct lista *l2)
{
  struct lista *inic, *c;

  if (l1->clave < l2->clave) { inic = l1; l1 = l1->sig; }
  else { inic = l2; l2 = l2->sig; }
  c = inic;
  while (l1 != centinela && l2 != centinela) {
    if (l1->clave < l2->clave) {
      c->sig = l1; l1 = l1->sig;
    }
    else {
      c->sig = l2; l2 = l2->sig;
    }
    c = c->sig;
  }
  if (l1 != centinela) c->sig = l1;
  else if (l2 != centinela) c->sig = l2;
  return inic;
}
```

- Segunda parte: divide y vencerás. Se separa la lista original en dos trozos del mismo tamaño (salvo listas de longitud impar) que se ordenan recursivamente, y una vez ordenados se fusionan obteniendo una lista ordenada. Como todo algoritmo basado en divide y vencerás tiene un caso base y un caso recursivo.

* *Caso base*: cuando la lista tiene 1 ó 0 elementos (0 se da si se trata de ordenar una lista vacía). Se devuelve la lista tal cual está.

* *Caso recursivo*: cuando la longitud de la lista es de al menos 2 elementos. Se **divide** la lista en dos trozos del mismo tamaño que se ordenan recursivamente. Una vez ordenado cada trozo, se fusionan y se devuelve la lista resultante.

El esquema es el siguiente:

```
Ordenar(lista L)
inicio
  si tamaño de L es 1 o 0 entonces
    devolver L
  si tamaño de L es >= 2 entonces
    separar L en dos trozos: L1 y L2.
    L1 = Ordenar(L1)
    L2 = Ordenar(L2)
    L = Fusionar(L1, L2)
    devolver L
fin
```

El algoritmo funciona y termina porque llega un momento en el que se obtienen listas de 2 ó 3 elementos que se dividen en dos listas de un elemento (1+1=2) y en dos listas de uno y dos elementos (1+2=3, la lista de 2 elementos se volverá a dividir), respectivamente. Por tanto se vuelve siempre de la recursión con listas ordenadas (pues tienen a lo sumo un elemento) que hacen que el algoritmo de fusión reciba siempre listas ordenadas.

Se incluye un ejemplo explicativo donde cada sublista lleva una etiqueta identificativa.

Dada: 3 -> 2 -> 1 -> 6 -> 9 -> 0 -> 7 -> 4 -> 3 -> 8 (lista original)

se **divide** en:
3 -> 2 -> 1 -> 6 -> 9 (lista 1)
0 -> 7 -> 4 -> 3 -> 8 (lista 2)
· se ordena recursivamente cada lista:
· 3 -> 2 -> 1 -> 6 -> 9 (lista 1)
· · se **divide** en:
· · 3 -> 2 -> 1 (lista 11)
· · 6 -> 9 (lista 12)
· · se ordena recursivamente cada lista:
· · 3 -> 2 -> 1 (lista 11)
· · · se **divide** en:
· · · 3 -> 2 (lista 111)
· · · 1 (lista 112)
· · · se ordena recursivamente cada lista:
· · · 3 -> 2 (lista 111)
· · · se **divide** en:
· · · 3 (lista 1111, que no se divide, *caso base*). Se devuelve 3
· · · 2 (lista 1112, que no se divide, *caso base*). Se devuelve 2
· · · se fusionan 1111-1112 y queda:
· · · 2 -> 3. Se devuelve 2 -> 3
· · · 1 (lista 112)
· · · 1 (lista 1121, que no se divide, *caso base*). Se devuelve 1
· · · se fusionan 111-112 y queda:
· · · 1 -> 2 -> 3 (lista 11). Se devuelve 1 -> 2 -> 3
· · 6 -> 9 (lista 12)
· · · se **divide** en:
· · · 6 (lista 121, que no se divide, *caso base*). Se devuelve 6
· · · 9 (lista 122, que no se divide, *caso base*). Se devuelve 9
· · · se fusionan 121-122 y queda:
· · · 6 -> 9 (lista 12). Se devuelve 6 -> 9
· · se fusionan 11-12 y queda:
· · 1 -> 2 -> 3 -> 6 -> 9. Se devuelve 1 -> 2 -> 3 -> 6 -> 9
· 0 -> 7 -> 4 -> 3 -> 8 (lista 2)
· ... tras repetir el mismo procedimiento se devuelve 0 -> 3 -> 4 -> 7 -> 8
· se fusionan 1-2 y queda:
· 0 -> 1 -> 2 -> 3 -> 3 -> 4 -> 6 -> 7 -> 8 -> 9, que se devuelve y se termina.

La implementación propuesta emplea un centinela sobre la lista inicial que apunte hacia sí mismo y que además contiene el máximo valor de un entero. La lista dispone de cabecera y centinela, pero obsérvese como se elimina durante la ordenación.

```
#include <stdlib.h>
#include <limits.h>

struct lista
{
  int clave;
  struct lista *sig;
};
```

```
/* lista con cabecera y centinela */
struct listacc
{
  struct lista *cabecera,
               *centinela;
};

/* centinela declarado como variable global */
struct lista *centinela;

  /* fusiona dos listas */
struct lista *fusion(struct lista *l1, struct lista *l2)
{
  struct lista *inic, *c;

  if (l1->clave < l2->clave) { inic = l1; l1 = l1->sig; }
  else { inic = l2; l2 = l2->sig; }
  c = inic;
  while (l1 != centinela && l2 != centinela) {
    if (l1->clave < l2->clave) {
      c->sig = l1; l1 = l1->sig;
    }
    else {
      c->sig = l2; l2 = l2->sig;
    }
    c = c->sig;
  }
  if (l1 != centinela) c->sig = l1;
  else if (l2 != centinela) c->sig = l2;
  return inic;
}

  /* algoritmo de ordenación por fusión mediante divide y vencerás */
struct lista *ordenfusion(struct lista *l)
{
  struct lista *l1, *l2, *parte1, *parte2;

    /* caso base: 1 ó 0 elementos */
  if (l->sig == centinela) return l;


    /* caso recursivo */
  /* avanza hasta la mitad de la lista */
  l1 = l; l2 = l1->sig->sig;
  while (l2 != centinela) {
    l1 = l1->sig;
    l2 = l2->sig->sig;
  }
  /* la parte en dos */
  l2 = l1->sig;
  l1->sig = centinela;

  /* ordena recursivamente cada parte */
  parte1 = ordenfusion(l);
  parte2 = ordenfusion(l2);

  /* mezcla y devuelve la lista mezclada */
  l = fusion(parte1, parte2);
  return l;
}
```

```
  /* operaciones de lista */
void crearLCC(struct listacc *LCC)
{
  LCC->cabecera = (struct lista *) malloc(sizeof(struct lista));
  LCC->centinela = (struct lista *) malloc(sizeof(struct lista));
  LCC->cabecera->sig = LCC->centinela;
  LCC->centinela->sig = LCC->centinela;
}

  /* inserta un elemento al comienzo de la lista */
void insertarPrimero(struct listacc LCC, int elem)
{
  struct lista *nuevo;

  nuevo = (struct lista *) malloc(sizeof(struct lista));
  nuevo->clave = elem;
  nuevo->sig = LCC.cabecera->sig;
  LCC.cabecera->sig = nuevo;
}

int main(void)
{
  struct listacc LCC;
  crearLCC(&LCC);
  centinela = LCC.centinela;
  centinela->clave = INT_MAX;

  insertarPrimero(LCC, 8);
  insertarPrimero(LCC, 3);
  insertarPrimero(LCC, 4);
  insertarPrimero(LCC, 7);
  insertarPrimero(LCC, 0);
  insertarPrimero(LCC, 9);
  insertarPrimero(LCC, 6);
  insertarPrimero(LCC, 1);
  insertarPrimero(LCC, 2);
  insertarPrimero(LCC, 3);
  LCC.cabecera = ordenfusion(LCC.cabecera->sig);

  return 0;
}
```

Este es un buen algoritmo de ordenación, pues no requiere espacio para una nueva lista y sólo las operaciones recursivas consumen algo de memoria. Es por tanto un **algoritmo ideal** para ordenar listas.

La complejidad es la misma en todos los casos, ya que no influye cómo esté ordenada la lista inicial -esto es, no existe ni mejor ni peor caso-, puesto que la intercalación de dos listas ordenadas siempre se realiza de una única pasada. La complejidad es proporcional a N·logN, característica de los algoritmos "Divide y Vencerás". Para hacer más eficiente el algoritmo es mejor realizar un primer recorrido sobre toda la lista para contar el número de elementos y añadir como parámetro a la función dicho número.

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.

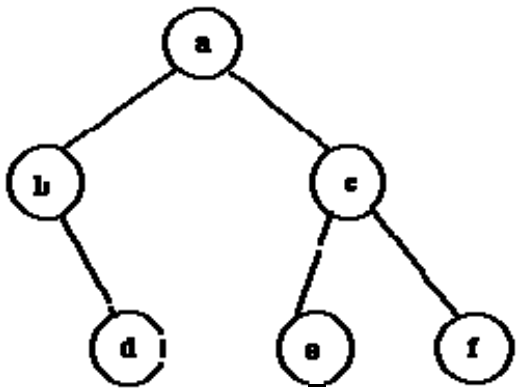


Figura 1

- **Recorridos en profundidad**:

* Recorrido en **preorden**: consiste en visitar el nodo actual (visitar puede ser simplemente mostrar la clave del nodo por pantalla), y después visitar el subárbol izquierdo y una vez visitado, visitar el subárbol derecho. Es un proceso recursivo por naturaleza.
Si se hace el recorrido en preorden del árbol de la figura 1 las visitas serían en el orden siguiente: a,b,d,c,e,f.

```
void preorden(tarbol *a)
{
  if (a != NULL) {
    visitar(a);
    preorden(a->izq);
    preorden(a->der);
  }
}
```

* Recorrido en **inorden** u orden central: se visita el subárbol izquierdo, el nodo actual, y después se visita el subárbol derecho. En el ejemplo de la figura 1 las visitas serían en este orden: b,d,a,e,c,f.

```
void inorden(tarbol *a)
{
  if (a != NULL) {
    inorden(a->izq);
    visitar(a);
    inorden(a->der);
  }
}
```

* Recorrido en **postorden**: se visitan primero el subárbol izquierdo, después el subárbol derecho, y por último el nodo actual. En el ejemplo de la figura 1 el recorrido quedaría así: d,b,e,f,c,a.

```
void postorden(arbol *a)
{
  if (a != NULL) {
    postorden(a->izq);
    postorden(a->der);
    visitar(a);
  }
}
```

La ventaja del **recorrido en postorden es que permite borrar el árbol de forma consistente**. Es decir, si visitar se traduce por borrar el nodo actual, al ejecutar este recorrido se borrará el árbol o subárbol que se pasa como parámetro. La razón para hacer esto es que no se debe borrar un nodo y después sus subárboles, porque al borrarlo se pueden perder los enlaces, y aunque no se perdieran se rompe con la regla de manipular una estructura de datos inexistente. Una alternativa es utilizar una variable auxiliar, pero es innecesario aplicando este recorrido.

- **Recorrido en amplitud**:

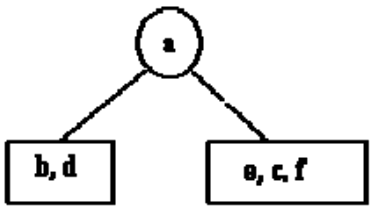
Consiste en ir visitando el árbol por niveles. Primero se visitan los nodos de nivel 1 (como mucho hay uno, la raíz), después los nodos de nivel 2, así hasta que ya no queden más.
Si se hace el recorrido en amplitud del árbol de la figura una visitaría los nodos en este orden: a,b,c,d,e,f
En este caso 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 extraido de la cola, y seguir desencolando y encolando hasta que la cola esté vacía.
En la codificación que viene a continuación no se implementan las operaciones sobre colas.

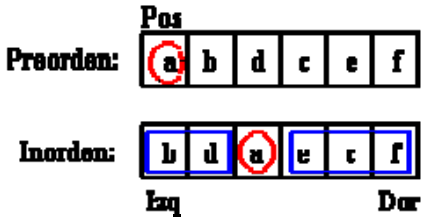
```
void amplitud(tarbol *a)
{
  tCola cola;   /* las claves de la cola serán de tipo árbol binario
*/
  arbol *aux;

  if (a != NULL) {
    CrearCola(cola);
    encolar(cola, a);
    while (!colavacia(cola)) {
      desencolar(cola, aux);
      visitar(aux);
      if (aux->izq != NULL) encolar(cola, aux->izq);
      if (aux->der != NULL) encolar(cola, aux->der);
    }
  }
}
```

Construcción de un árbol binario

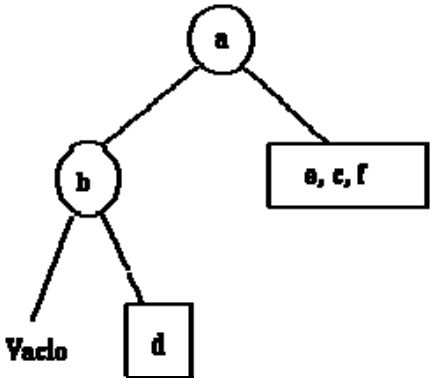
A continuación se estudia un método para **crear un árbol binario que no tenga claves repetidas partiendo de su recorrido en preorden e inorden,** almacenados en sendos arrays.

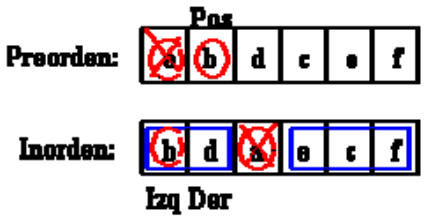
Partiendo de los recorridos preorden e inorden del árbol de la figura 1 puede determinarse que la raíz es el primer elemento del recorrido en preorden. Ese elemento se busca en el array inorden. Los elementos en el array inorden entre **izq** y la raíz forman el subárbol izquierdo. Asimismo los elementos entre **der** y la raíz forman el subárbol derecho. Por tanto se tiene este árbol:



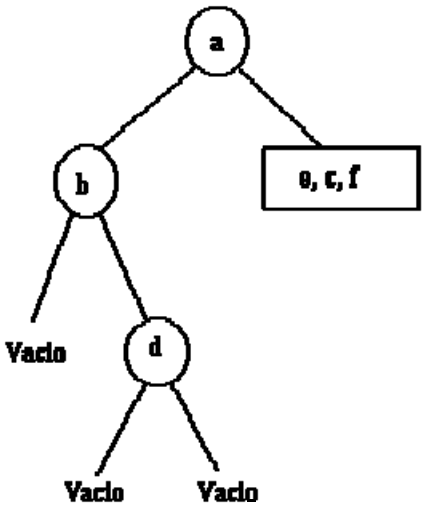


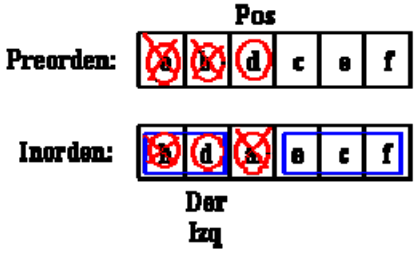
A continuación comienza un proceso recursivo. Se procede a crear el subárbol izquierdo, cuyo tamaño está limitado por los índices **izq** y **der**. La siguiente posición en el recorrido en preorden es la raíz de este subárbol. Queda esto:





El subárbol *b* tiene un subárbol derecho, que no tiene ningún descendiente, tal y como indican los índices **izq** y **der**. Se ha obtenido el subárbol izquierdo completo de la raíz *a*, puesto que *b* no tiene subárbol izquierdo:





Después seguirá construyéndose el subárbol derecho a partir de la raíz *a*.

```
#include <stdio.h>
#include <stdlib.h>


typedef struct arbol
{
  int clave;
  struct arbol *izq;
  struct arbol *der;
} arbol;

void crear(arbol **a, int *preorden , int *inorden, int *pos, int izq,
int der);


int main(void)
{
  arbol *a;
  int preorden[] = {1,2};//,4,3,5,6};
  int inorden[]  = {2,1};//,4,1,5,3,6};
  int pos = 0;

  crear(&a, preorden, inorden, &pos, 0,1);

  return 0;
}


void crear(arbol **a, int *preorden , int *inorden, int *pos, int izq,
int der)
{
  int i;

  if (izq > der)
    *a = NULL;
  else if (izq == der) {
    *a = (arbol *) malloc(sizeof(arbol));
    (*a)->clave = preorden[*pos];
    (*a)->izq = (*a)->der = NULL;
    (*pos)++;
  }
  else {
    *a = (arbol *) malloc(sizeof(arbol));
    (*a)->clave = preorden[*pos];
      /* Busca Clave 'Pos' en tabla de orden central */
    i = izq;
    while (inorden[i] != (*a)->clave) i++;
      /* actualiza la posicion */
    (*pos)++;
      /* crea los subarboles izquierdo y derecho */
    crear(&(*a)->izq, preorden, inorden, pos, izq, i-1);
    crear(&(*a)->der, preorden, inorden, pos, i+1, der);
  }
}
```

Árbol binario de búsqueda

El recorrido inorden de un árbol binario de búsqueda obtiene una lista ordenada por la clave de menor a mayor de los nodos

Operaciones básicas sobre árboles binarios de búsqueda

- **Búsqueda**

Si el árbol no es de búsqueda, es necesario emplear uno de los recorridos anteriores sobre el árbol para localizarlo. El resultado es idéntico al de una búsqueda secuencial. Aprovechando las propiedades del árbol de búsqueda se puede acelerar la localización. Simplemente hay que descender a lo largo del árbol a izquierda o derecha dependiendo del elemento que se busca.

```
boolean buscar(tarbol *a, int elem)
{
  if (a == NULL) return FALSE;
  else if (a->clave < elem) return buscar(a->der, elem);
  else if (a->clave > elem) return buscar(a->izq, elem);
  else return TRUE;
}
```

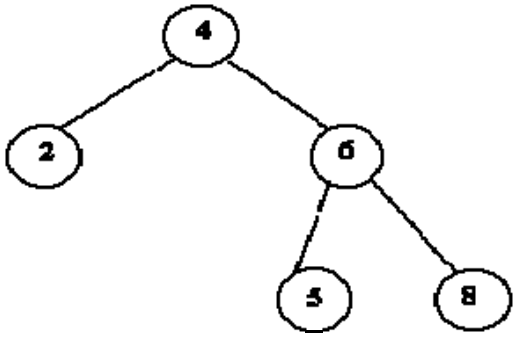
- Inserción

La inserción tampoco es complicada. Es más, resulta practicamente idéntica a la búsqueda. Cuando se llega a un árbol vacío se crea el nodo en el puntero que se pasa como parámetro por referencia, de esta manera los nuevos enlaces mantienen la coherencia. Si el elemento a insertar ya existe entonces no se hace nada.

```
void insertar(tarbol **a, int elem)
{
  if (*a == NULL) {
    *a = (arbol *) malloc(sizeof(arbol));
    (*a)->clave = elem;
    (*a)->izq = (*a)->der = NULL;
  }
  else if ((*a)->clave < elem) insertar(&(*a)->der, elem);
  else if ((*a)->clave > elem) insertar(&(*a)->izq, elem);
}
```

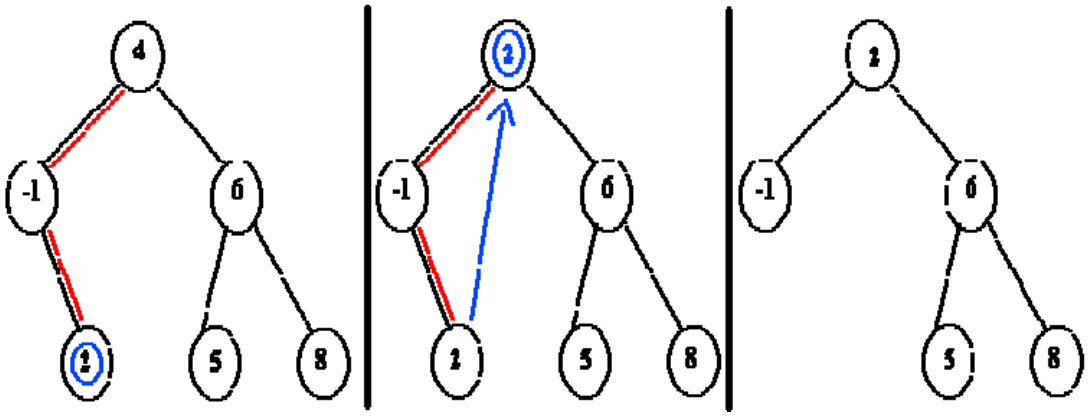
- Borrado

La operación de borrado si resulta ser algo más complicada. Se recuerda que el árbol debe seguir siendo de búsqueda tras el borrado. Pueden darse tres casos, una vez encontrado el nodo a borrar:
1) El nodo no tiene descendientes. Simplemente se borra.
2) El nodo tiene al menos un descendiente por una sola rama. Se borra dicho nodo, y su primer descendiente se asigna como hijo del padre del nodo borrado. Ejemplo: en el árbol de la figura 5 se borra el nodo cuya clave es -1. El árbol resultante es:



3) El nodo tiene al menos un descendiente por cada rama. Al borrar dicho nodo es necesario mantener la coherencia de los enlaces, además de seguir manteniendo la estructura como un árbol binario de búsqueda. La solución consiste en sustituir la información del nodo que se borra por el de una de las hojas, y borrar a continuación dicha hoja. ¿Puede ser cualquier hoja? No, debe ser la que contenga una de estas dos claves:
· la **mayor** de las claves **menores** al nodo que se borra. Suponer que se quiere borrar el nodo *4* del árbol de la figura 5. Se sustituirá la clave 4 por la clave 2.
· la **menor** de las claves **mayores** al nodo que se borra. Suponer que se quiere borrar el nodo *4* del árbol de la figura 5. Se sustituirá la clave 4 por la clave 5.

El algoritmo de borrado que se implementa a continuación realiza la sustitución por la mayor de las claves menores, (aunque se puede escoger la otra opción sin pérdida de generalidad). Para lograr esto es necesario descender primero a la izquierda del nodo que se va a borrar, y después avanzar siempre a la derecha hasta encontrar un nodo hoja. A continuación se muestra gráficamente el proceso de borrar el nodo de clave 4:



Codificación: el procedimiento sustituir es el que desciende por el árbol cuando se da el caso del nodo con descendientes por ambas ramas.

```
void borrar(tarbol **a, int elem)
{
  void sustituir(tarbol **a, tarbol **aux);
  tarbol *aux;

  if (*a == NULL)   /* no existe la clave */
    return;

  if ((*a)->clave < elem) borrar(&(*a)->der, elem);
  else if ((*a)->clave > elem) borrar(&(*a)->izq, elem);
```

```
  else if ((*a)->clave == elem) {
    aux = *a;
    if ((*a)->izq == NULL) *a = (*a)->der;
    else if ((*a)->der == NULL) *a = (*a)->izq;
    else sustituir(&(*a)->izq, &aux); /* se sustituye por la mayor de
las menores */

    free(aux);
  }
}

void sustituir(tarbol **a, tarbol **aux)
{
  if ((*a)->der != NULL) sustituir(&(*a)->der, aux);
  else {
    (*aux)->clave = (*a)->clave;
    *aux = *a;
    *a = (*a)->izq;
  }

}
```

Aplicación práctica de un árbol

Se tiene un fichero de texto ASCII. Para este propósito puede servir cualquier libro electrónico de la librería Gutenberg o Cervantes, que suelen tener varios cientos de miles de palabras. El objetivo es clasificar todas las palabras, es decir, determinar que palabras aparecen, y cuantas veces aparece cada una. Palabras como 'niño'-'niña', 'vengo'-'vienes' etc, se consideran diferentes por simplificar el problema.

Escribir un programa, que recibiendo como entrada un texto, realice la clasificación descrita anteriormente.
Ejemplo:
Texto: "a b'a c. hola, adios, hola"

La salida que produce es la siguiente:
a 2
adios 1
b 1
c 1
hola 2

La solución pasa por **emplear un árbol binario de búsqueda para insertar las claves**. El valor de log(20.000) es aproximadamente de 14. Eso quiere decir que localizar una palabra entre 20.000 llevaría en el peor caso unos 14 accesos. El contraste con el empleo de una lista es simplemente abismal. Por supuesto, como se ha comentado anteriormente el árbol no va a estar perfectamente equilibrado, pero nadie escribe novelas manteniendo el orden lexicográfico (como un diccionario) entre las palabras, asi que no se obtendrá nunca un árbol muy degenerado. Lo que está claro es que cualquier evolución del árbol siempre será mejor que el empleo de una lista.

Por último, una vez realizada la lectura de los datos, sólo queda hacer un recorrido inorden del árbol y se obtendrá la solución pedida en cuestión de segundos.

Una posible definición de la estructura árbol es la siguiente:

```
typedef struct tarbol
{
  char clave[MAXPALABRA];
  int contador; /* numero de apariciones. Iniciar a 0 */
  struct tarbol *izq,
                *der;
} tarbol;
```

Representación de grafos

Una característica especial en los grafos es que podemos representarlos utilizando dos estructuras de datos distintas. En los algoritmos que se aplican sobre ellos veremos que adoptarán tiempos distintos dependiendo de la forma de representación elegida. En particular, los tiempos de ejecución variarán en función del número de vértices y el de aristas, por lo que la utilización de una representación u otra dependerá en gran medida de si el grafo es denso o disperso.

Para nombrar los nodos utilizaremos letras mayúsculas, aunque en el código deberemos hacer corresponder cada nodo con un entero entre 1 y V (número de vértices) de cara a la manipulación de los mismos.

Representación por matriz de adyacencia

Es la forma más común de representación y la más directa. Consiste en una tabla de tamaño V x V, en que la que a[i][j] tendrá como valor 1 si existe una arista del nodo i al nodo j. En caso contrario, el valor será 0. Cuando se trata de grafos ponderados en lugar de 1 el valor que tomará será el peso de la arista. Si el grafo es no dirigido hay que asegurarse de que se marca con un 1 (o con el peso) tanto la entrada a[i][j] como la entrada a[j][i], puesto que se puede recorrer en ambos sentidos.

```
int V,A;
int a[maxV][maxV];

void inicializar()
{
  int i,x,y,p;
  char v1,v2;
  // Leer V y A
  memset(a,0,sizeof(a));
  for (i=1; i<=A; i++)
  {
    scanf("%c %c %d\n",&v1,&v2,&p);
    x=v1-'A'; y=v2-'A';
    a[x][y]=p; a[y][x]=p;
  }
}
```

En esta implementación se ha supuesto que los vértices se nombran con una letra mayúscula y no hay errores en la entrada. Evidentemente, cada problema tendrá una forma de entrada distinta y la inicialización será conveniente adaptarla a cada situación. En todo caso, esta operación es sencilla si el número de nodos es pequeño. Si, por el contrario, la entrada fuese muy grande se pueden almacenar los nombres de nodos en un árbol binario de búsqueda o utilizar una tabla de dispersión, asignando un entero a cada nodo, que será el utilizado en la matriz de adyacencia.

Como se puede apreciar, la matriz de adyacencia siempre ocupa un espacio de V*V, es decir, depende solamente del número de nodos y no del de aristas, por lo que será útil para representar grafos densos.

Representación por lista de adyacencia

Otra forma de representar un grafo es por medio de listas que definen las aristas que conectan los nodos. Lo que se hace es definir una lista enlazada para cada nodo, que contendrá los nodos a los cuales es posible acceder. Es decir, un nodo A tendrá una lista enlazada asociada en la que aparecerá un elemento con una referencia al nodo B si A y B tienen una arista que los une. Obviamente, si el grafo es no dirigido, en la lista enlazada de B aparecerá la correspondiente referencia al nodo A.

Las listas de adyacencia serán estructuras que contendrán un valor entero (el número que identifica al nodo destino), así como otro entero que indica el coste en el caso de que el grafo sea ponderado. En el ejemplo se ha utilizado un nodo z ficticio en la cola.

```
struct nodo
{
  int v;
  int p;
  nodo *sig;
};

int V,A;  // vértices y aristas del grafo
struct nodo *a[maxV], *z;

void inicializar()
{
  int i,x,y,peso;
  char v1,v2;
  struct nodo *t;
  z=(struct nodo *)malloc(sizeof(struct nodo));
  z->sig=z;
  for (i=0; i<V; i++)
    a[i]=z;
  for (i=0; i<A; i++)
  {
    scanf("%c %c %d\n",&v1,&v2,&peso);
    x=v1-'A'; y=v2-'A';

    t=(struct nodo *)malloc(sizeof(struct nodo));
    t->v=y; t->p=peso; t->sig=a[x]; a[x]=t;

    t=(struct nodo *)malloc(sizeof(struct nodo));
    t->v=x; t->p=peso; t->sig=a[y]; a[y]=t;
  }
}
```

En este caso el espacio ocupado es O(V + A), muy distinto del necesario en la matriz de adyacencia, que era de $O(V^2)$. La representación por listas de adyacencia, por tanto, será más adecuada para grafos dispersos.

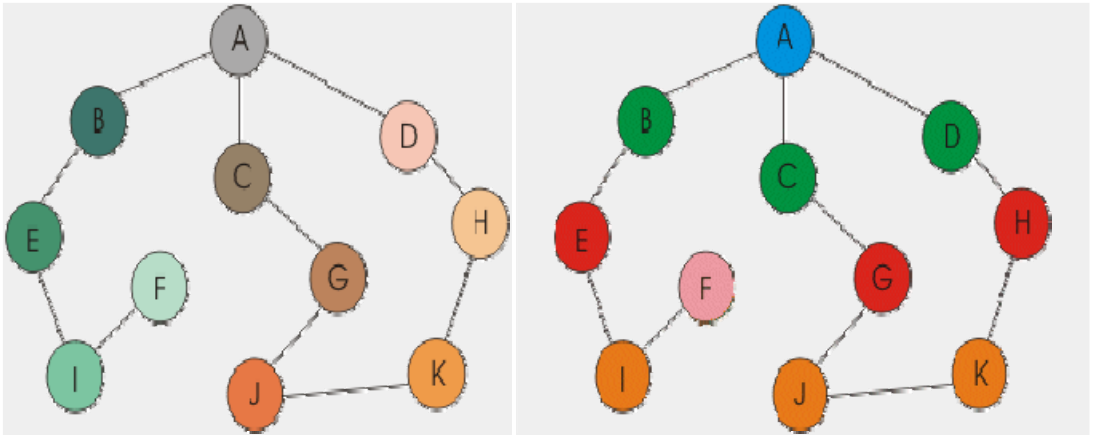
Hay que tener en cuenta un aspecto importante y es que la implementación con listas enlazadas determina fuertemente el tratamiento del grafo posterior. Como se puede ver en el código, los nodos se van añadiendo a las listas según se leen las

aristas, por lo que nos encontramos que un mismo grafo con un orden distinto de las aristas en la entrada producirá listas de adyacencia diferentes y por ello el orden en que los nodos se procesen variará. Una consecuencia de esto es que si un problema tiene varias soluciones la primera que se encuentre dependerá de la entrada dada. Podría presentarse el caso de tener varias soluciones y tener que mostrarlas siguiendo un determinado orden. Ante una situación así podría ser muy conveniente modificar la forma de meter los nodos en la lista (por ejemplo, hacerlo al final y no al principio, o incluso insertarlo en una posición adecuada), de manera que el algoritmo mismo diera las soluciones ya ordenadas.

Exploración de grafos

A la hora de explorar un grafo, nos encontramos con dos métodos distintos. Ambos conducen al mismo destino (la exploración de todos los vértices o hasta que se encuentra uno determinado), si bien el orden en que éstos son "visitados" decide radicalmente el tiempo de ejecución de un algoritmo, como se verá posteriormente.

En primer lugar, una forma sencilla de recorrer los vértices es mediante una función recursiva, lo que se denomina búsqueda en profundidad. La sustitución de la recursión (cuya base es la estructura de datos pila) por una cola nos proporciona el segundo método de búsqueda o recorrido, la búsqueda en amplitud o anchura.



Suponiendo que el orden en que están almacenados los nodos en la estructura de datos correspondiente es A-B-C-D-E-F... (el orden alfabético), tenemos que el orden que seguiría el recorrido en profundidad sería el siguiente:

A-B-E-I-F-C-G-J-K-H-D

En un recorrido en anchura el orden sería, por contra:

A-B-C-D-E-G-H-I-J-K-F

Es decir, en el primer caso se exploran primero los verdes y luego los marrones, pasando primero por los de mayor intensidad de color. En el segundo caso se exploran primero los verdes, después los rojos, los naranjas y, por último, el rosa.

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.

En estos ejemplos 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.

Búsqueda en profundidad

Se implementa de forma recursiva, aunque también puede realizarse con una pila. Se utiliza un array `val` para almacenar el orden en que fueron explorados los vértices. Para ello se incrementa una variable global `id` (inicializada a 0) cada vez que se visita un nuevo vértice y se almacena `id` en la entrada del array `val` correspondiente al vértice que se está explorando.

La siguiente función realiza un máximo de V (el número total de vértices) llamadas a la función `visitar`, que implementamos aquí en sus dos variantes: representación por matriz de adyacencia y por listas de adyacencia.

```
int id=0;
int val[V];

void buscar()
{
  int k;
  for (k=1; k<=V; k++)
    val[k]=0;
  for (k=1; k<=V; k++)
    if (val[k]==0) visitar(k);
}

void visitar(int k) // matriz de adyacencia
{
  int t;
  val[k]=++id;
  for (t=1; t<=V; t++)
    if (a[k][t] && val[t]==0) visitar(t);
}

void visitar(int k) // listas de adyacencia
{
  struct nodo *t;
  val[k]=++id;
  for (t=a[k]; t!=z; t=t->sig)
    if (val[t->v]==0) visitar(t->v);
}
```

El resultado es que el array `val` contendrá en su i-ésima entrada el orden en el que el vértice i-ésimo fue explorado. Es decir, si tenemos un grafo con cuatro nodos y fueron explorados en el orden 3-1-2-4, el array `val` quedará como sigue:

```
val[1]=2; // el primer nodo fue visto en segundo lugar
val[2]=3; // el segundo nodo fue visto en tercer lugar
val[3]=1; // etc.
val[4]=4;
```

Una modificación que puede resultar especialmente útil es la creación de un array "inverso" al array `val` que contenga los datos anteriores "al revés". Esto es, un

array en el que la entrada i-ésima contiene el vértice que se exploró en i-ésimo lugar. Basta modificar la línea

```
val[k]=++id;
```

sustituyéndola por

```
val[++id]=k;
```

Para el orden de exploración de ejemplo anterior los valores serían los siguientes:

```
val[1]=3;
val[2]=1;
val[3]=2;
val[4]=4;
```

Búsqueda en amplitud o anchura

La diferencia fundamental respecto a la búsqueda en profundidad es el cambio de estructura de datos: una cola en lugar de una pila. En esta implementación, la función del array `val` y la variable `id` es la misma que en el método anterior.

```
struct tcola *cola;

void visitar(int k) // listas de adyacencia
{
  struct nodo *t;
  encolar(&cola,k);
  while (!vacia(cola))
  {
    desencolar(&cola,&k);
    val[k]=++id;
    for (t=a[k]; t!=z; t=t->sig)
    {
      if (val[t->v]==0)
      {
        encolar(&cola,t->v);
        val[t->v]=-1;
      }
    }
  }
}
```