a) visitar la raíz,
b) recorrer el subárbol derecho,
c) recorrer el subárbol izquierdo,

utilizando la misma regla recursivamente en todos los subárboles no-vacíos. ¿Este nuevo orden guarda alguna relación simple con los otros tres ya descritos? ¿Que clase de enhilamientos deberían definirse para hacer que el recorrido en este orden sea más fácil?

26. Describa qué sucede cuando una transformación de árbol general a binario se aplica a un árbol binario.

27. Escriba un procedimiento para reconstruir un árbol general a partir del árbol binario que resulte de aplicar el algoritmo de transformación, presentado en este capítulo.

28. Escriba un programa que tome árboles generales como entrada y cuya salida sean los correspondientes árboles binarios.

29. Escriba procedimientos iterativos para recorrer en en-orden un árbol binario que está enhilado en en-orden; otro para ejecutar un recorrido en pre-orden en un árbol binario enhilado en pre-orden; otro más para recorrer un árbol binario en post-orden y que está enhilado en post-orden. Escriba programas para implantar estos procedimientos.

30. Escriba procedimientos recursivos para implantar los recorridos en en-orden, pre-orden y post-orden para árboles binarios no enhilados. Escriba el correspondiente programa.

31. Escriba los correspondientes procedimientos no recursivos (e.i. iterativos) para recorrer árboles binarios no enhilados en post-orden, en-orden y pre-orden. Escriba el respectivo programa.

32. ¿Que ventajas y desventajas deberán tomarse en cuenta para decidir si el recorrido del árbol debiera implantarse mediante rutinas recursivas o iterativas?

33. Escriba procedimientos para insertar nuevos nodos en un árbol binario que está enhilado en en-orden, en pre-orden, en post-orden.

34. Escriba los procedimientos para borrar nodos de un árbol binario y de un árbol binario enhilado.

35. ¿Cuál es la diferencia entre un árbol balanceado y un árbol balanceado por su altura?

36. Escriba procedimientos no recursivos (i.e. iterativos) para buscar e insertar nodos en un árbol de búsqueda binario.

37. Modifique los procedimientos para buscar e insertar nodos en árboles de búsqueda binarios de tal forma que reconozcan nodos que han sido marcados como nodos borrados, pero aún no han sido removidos del árbol.

38. Escriba un procedimiento para borrar un nodo dado de un árbol de búsqueda binario. Asegúrese de considerar las situaciones en que un nodo tiene 0, 1 o 2 subárboles no-nulos.

39. Lea acerca de las técnicas para conservar un árbol en balance. Escriba un programa para implantar una de las técnicas.

capítulo nueve

búsqueda y ordenamiento

En este capítulo tomaremos un breve descanso con respecto al aprendizaje de nuevas estructuras de datos y en su lugar estudiaremos las técnicas para encontrar valores de datos particulares en las estructuras y para ordenar dichos valores. La búsqueda y el ordenamiento son procesos estrechamente relacionados, como ya veremos.

Considere una colección de registros, cada uno de los cuales tiene una *llave* que puede usarse para identificarlo. Una llave puede estar compuesta por uno o más campos. El valor de la llave puede ser el único identificador del registro, aunque también se pueden permitir valores duplicados. Si se permiten valores de llave duplicados para los registros, es aconsejable diferenciarlos de acuerdo al orden en que vayan apareciendo. El concepto de llave se utilizará a través de los capítulos subsecuentes de este libro. La *búsqueda* es el proceso de localizar un registro con valor de llave particular. El *ordenamiento* es el proceso de arreglar registros de tal forma que queden ordenados por su valor llave.

BUSQUEDA SECUENCIAL

Un algoritmo de búsqueda es una técnica para encontrar un registro que tenga algún valor de llave en especial. Llamaremos al valor de la llave, digamos k, el *argumento* de la búsqueda. La búsqueda termina exitosamente cuando se localiza el registro que contenga la llave k, o termina sin éxito, cuando se determina que no aparece ningún registro con la llave k.

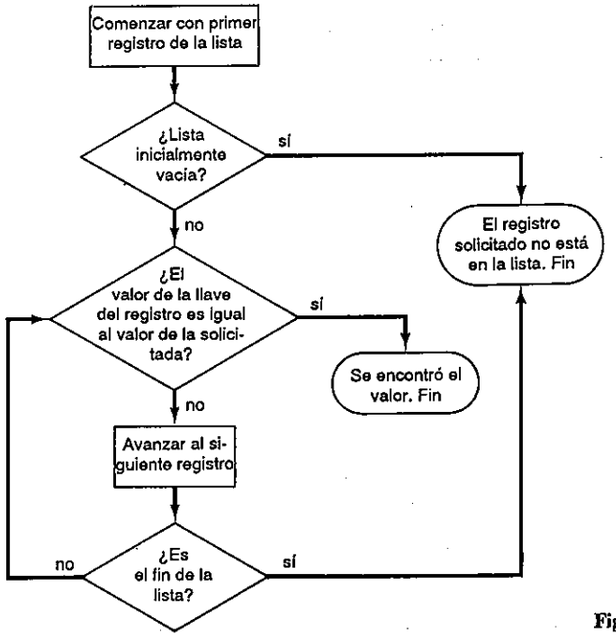


Figura 9-1 Algoritmo de búsqueda secuencial.

Existen muchos algoritmos de búsqueda; en esta sección consideraremos la *búsqueda secuencial*, también conocida como *búsqueda lineal*. Más adelante en este capítulo, consideraremos búsquedas no lineales, las cuales tienden a un desempeño mejor que las técnicas secuenciales. Sin embargo, las búsquedas secuenciales son más simples y proporcionan una buena forma de empezar nuestra investigación de búsqueda y ordenamiento.

Supongamos que la colección de registros, en la cual buscamos, se ha organizado como una lista lineal. Hemos visto que una lista lineal puede representarse por un arreglo o por una lista ligada. Aquí usaremos una representación de arreglo; posteriormente usted podrá determinar como se alteran los algoritmos al usar listas ligadas.

Sea la llave(i) el valor de la llave del *i*-ésimo registro de la lista. El algoritmo básico de búsqueda secuencial, consiste en empezar en el inicio de la lista, e ir a través de cada registro hasta encontrar el registro con la llave indicada (k), o hasta llegar al final de la lista. Un diagrama de flujo del algoritmo es presentado en la figura 9-1. El algoritmo puede implantarse en lenguaje Pascal como sigue. Suponga que existen n registros en la lista y que la llave, k, y n son variables globales.

```
procedure encuentra-k;
var encontró: boolean;i:1 . . n;
begin
    encontró := false;
    i := l;
```



```
    while (not encontró) and i ≤ n
do if k = llave[i]
    then encontró := true
    else i := i + 1;
if(encontró)
then writeln ('en registro con la llave', k, 'se encontró en la posición', i)
else writeln ('no se encontró el registro con la llave', k)
end;
```

Si el algoritmo termina con encontró = true, entonces, el registro con la llave k es el *i*-ésimo registro de la lista. Si el algoritmo termina con encontró = false, entonces, el registro con llave k no aparece en la lista.

¿Qué tan eficiente es este algoritmo? Sea prob(*i*) la probabilidad de que el *i*-ésimo registro sea el registro buscado, entonces

$$\sum_{i=1}^{n} \text{prob}(i) + Q = 1, \qquad \text{donde } Q = \text{probabilidad de que la llave no esté presente.}$$

La situación óptima es que el registro a buscar sea el primero en ser examinado. El peor caso es cuando las llaves de todos los *n* registros son comparadas con k, y ya sea que llave [n] = k o que el registro a buscar no esté en la lista.

El número de comparaciones promedio esperadas, es:

$$\sum_{i=1}^{n} i*\text{prob}(i) + Q*n$$

Con la probabilidad prob(1), se requiere de una comparación; con la probabilidad prob(2), se necesitan 2 comparaciones; y así con la probabilidad prob(*n*) + *Q*, se necesitan *n* comparaciones. Si todos los registros tienen la misma posibilidad de ser recuperados y cada llave examinada está en la lista, entonces:

$$\text{prob}(i) = \frac{1}{n} \text{ para toda } i$$

y entonces;

$$\sum_{i=1}^{n} i*\text{prob}\,(i) = \sum_{i=1}^{n} \frac{i}{n}, \text{ que es } \frac{n+1}{2.}$$

Esto significa que la mitad de las llaves en promedio serán comparadas con el argumento k.

En algunas situaciones, los valores de la llave no son únicos. Para encontrar todos los registros con una llave particular, se requiere buscar en toda la lista. Se deben comparar todos los valores de llave con el argumento de búsqueda.

Un algoritmo de búsqueda secuencial, se dice que requiere "sobre el orden de N" comparaciones. Es decir, el número esperado de comparaciones es una función lineal del número de registros en el conjunto. "Sobre el orden de N" se denota frecuentemente como $O(N)$. Duplicar el número de llaves significa que el proceso tomará el doble de tiempo.

Así, ¡éste no es un método de búsqueda particularmente bueno! Por ejemplo, usted seguramente nunca ha utilizado esta técnica para buscar un nombre en un directorio telefónico, el cual es realmente una lista lineal. ¿Cómo podríamos mejorar la situación?

MEJORANDO LA EFICIENCIA DE LA BUSQUEDA SECUENCIAL

Considere otra vez la expresión para el número esperado de comparaciones con una búsqueda secuencial.

$$\sum_{i=1}^{n} i*\text{prob}(i) + Q*n$$

Si tenemos la libertad de arreglar los registros de la lista lineal en cualquier orden que escojamos, entonces la expresión de arriba se minimiza cuando:

$$\text{prob}(1) \geq \text{prob}(2) \geq \ldots \geq \text{prob}(n)$$

Esto es

$$\text{prob}(I) \geq \text{prob}(J) \text{ para } I < J.$$

Al ordenar los registros por frecuencias descendentes de acceso, podemos mejorar la probabilidad de que se reduzca el número de comparaciones. Aunque puede ser el caso de que sean necesarias las comparaciones (cuando la llave que buscamos no está en la lista), el número *esperado* de comparaciones será minimizado.

Muestreo de accesos

Consideremos ahora una situación más normal, en la cual la frecuencia relativa de acceso a los registros es desconocida. Existen algunos esquemas para resolver este problema. Uno consiste en observar los requerimientos a la lista en un periodo, registrando el número total de accesos a cada registro. Cuando se haya obtenido, una muestra representativa de la actividad, los registros pueden reordenarse de acuerdo con las probabilidades de acceso detectadas.

Movimiento hacia el frente

Un segundo esquema es que la lista de registros se reorganice dinámicamente por sí misma. Con el método de *movimiento hacia el frente*, cada vez que la búsqueda de una

llave sea exitosa, el registro correspondiente se mueve a la primera posición de la lista. El registro que era el número uno, ahora se convierte en el número dos, y así sucesivamente. El algoritmo de movimiento hacia el frente se da en la figura 9-2. Este algoritmo puede ser implantado en Pascal como sigue, suponiendo que existen n registros en la lista y que registro (el cual contiene la llave), k, y n son variables conocidas globalmente:

```
procedure mueve-al-frente;
var encontró:boolean;
    i:1 . . n;
    j:0 . . n;
    reg-temporal:tipo-registro;
begin
    encontró := false;
```

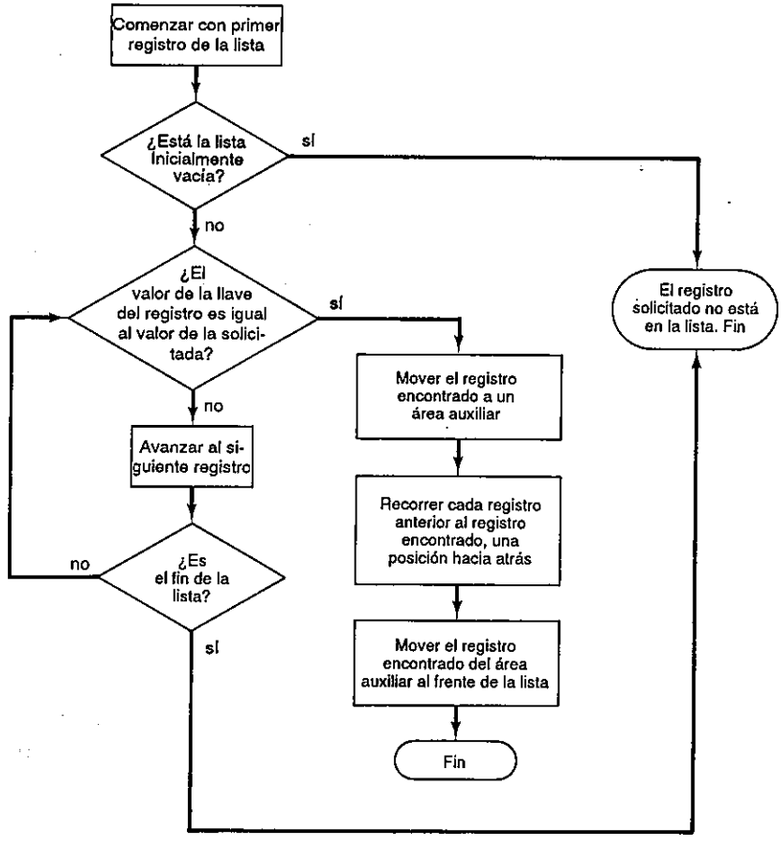


Figura 9-2 Algoritmo de búsqueda secuencial con "movimiento hacia el frente".

```
        i := 1;
        while (not encontró) and i ≤ n
        do if k = llave[i]
              then encontró := true
              else i := i + 1;
        if(encontró)
        then begin writeln ('el registro con llave ', k, 'se encontró');
                   reg-temporal := registro[i];
                   j := i - 1;
                   while j > 0
             do begin registro[j + 1] := registro [j];
                   j := j - 1
                end;
             registro[1] := reg-temporal
     end;
  else writeln ('no se encontró el registro con llave', k)
end;
```

Este método se comporta mejor en una lista lineal representada por una lista ligada que por un arreglo. ¿Por qué?

Transposición

Otro esquema de reorganización dinámica es el método de *transposición*. Siempre y cuando exista una búsqueda exitosa para una llave, el registro correspondiente es intercambiado con el registro inmediatamente anterior. En Pascal, haremos la misma suposición de que la declaración de variables, como en el procedimiento de mueve-al-frente, es global.

```
procedure transpon;
var encontró:boolean;
    i:1 . . n;
    reg-temporal:tipo-registro;
begin
    encontró := false;
    i:= 1;
    while (not encontró) and i ≤ n
    do if(k = llave[i])
       then encontró := true
       else i := i + 1;
    if (encontró and i > 1)
    then begin writeln ('el registro con llave', k, 'se encontró');
                reg-temporal := registro[i];{intercambia registros adyacentes}
                registro[i] : = registro [i - 1];
                registro[i - 1] := reg-temporal
         end;
    else writeln ('el registro con llave', k, 'no se encontró')
end;
```


Con este procedimiento, entre más accesos tenga el registro, más rápidamente avanzará hacia la primera posición. Comparado con el método de movimiento al frente, el método requiere más tiempo de actividad para reorganizar el conjunto de registros. Una ventaja del método de transposición es que no permite que el requerimiento aislado de un registro, cambie de posición todo el conjunto de registros. De hecho, un registro debe ganar poco a poco su derecho a alcanzar el inicio de la lista, a través de una serie de demandas.

Ordenamiento

Una forma de reducir el número de comparaciones esperadas cuando hay una significativa frecuencia de búsquedas sin éxito, es la de ordenar los registros en base al valor de la llave. Esto es:

$$\text{llave}(I) \leq \text{llave}(J) \quad \text{para} \quad I < J$$

[o bien llave(I) ≥ llave(J) para $I < J$, en orden descendente]. Esta técnica es útil cuando la lista es una lista de excepciones, tales como números erróneos de tarjetas de crédito para establecer las comparaciones, en cuyo caso la mayoría de las búsquedas no tendrán éxito. Con este método, una búsqueda sin éxito termina cuando se encuentra el primer valor de la llave mayor que el del argumento, en lugar de al final de la lista, como se

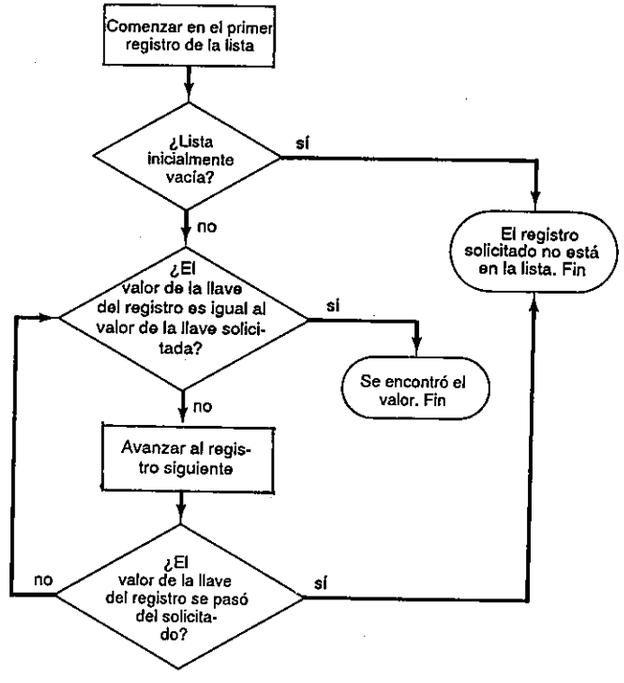


Figura 9-3 Algoritmo de búsqueda secuencial para una lista ordenada.

muestra en el diagrama de flujo de la figura 9-3. El algoritmo puede codificarse en Pascal como sigue:

```
encontró := false;
i := 1;
while (not encontró) and llave [i] ≤ k
do if(k = llave[i])
    then encontró := true
    else i := i + 1;
```

Más tarde, en el capítulo, consideraremos técnicas para ordenar los elementos de la lista.

BUSQUEDA BINARIA

Todas las técnicas de búsqueda secuencial requieren O(n) comparaciones. La eficiencia de la búsqueda secuencial básica puede ser mejorada mediante un posicionamiento más inteligente de los registros (conociendo las probabilidades de acceso, moviendo los registros inmediatamente, o más despacio al frente de la lista, o guardando los registros en orden, sin embargo el comportamiento de O(n) aún persistirá. En realidad para mejorar la eficiencia de búsqueda, es necesario usar diferentes enfoques, con técnicas no lineales.

En los capítulos 7 y 8 empezamos nuestro estudio de estructuras de datos no lineales. Descubrimos que al estructurar datos en un árbol binario, el tiempo de búsqueda para un registro en particular puede reducirse significativamente, con respecto al de una estructura de datos lineal. Para un conjunto de n registros, el tiempo esperado de búsqueda promedio en un árbol binario balanceado es O($\log_2 n$), mientras que el tiempo promedio de búsqueda en listas lineales es O($n/2$). Para una n grande, el árbol binario es mucho más eficiente. Tal como la ramificación de estructuras no lineales ayuda a reducir el tiempo de búsqueda, también las técnicas de búsqueda y ordenamiento no lineales pueden mejorar la eficiencia con respecto a los métodos secuenciales.

Ya hemos cubierto una estrategia no lineal de búsqueda basada en árboles, en nuestra discusión sobre árboles binarios. Cuando estuvimos buscando el nodo con un valor de llave en particular, las comparaciones en cada nodo intermedio nos dirigieron a lo largo de la ramificación adecuada del árbol, hasta llegar a nuestro destino. En un árbol completamente balanceado, cada comparación divide a la mitad la ruta de búsqueda, de esta forma el promedio de tiempo de búsqueda esperado, en un árbol binario balanceado, es O($\log_2 n$). Esta es una mejor eficiencia comparada con la de los métodos de búsqueda secuencial.

La técnica de búsqueda binaria puede ser aplicada tanto a los datos en listas lineales como en árboles binarios de búsqueda. Los prerrequisitos principales para la búsqueda binaria son: la lista debe estar ordenada en un orden específico de acuerdo al valor de la llave y debe conocerse el número de registros. La técnica de búsqueda binaria sólo podrá aplicarse si son satisfechos estos dos prerrequisitos.

La búsqueda procede mediante una serie de pruebas sucesivas en la lista. La primer prueba compara el valor de la llave, que está a la mitad de la lista, contra el valor buscado.

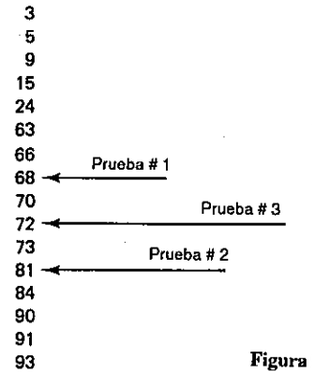


Figura 9-4 Pruebas en una búsqueda binaria.

Si el valor buscado es menor, entonces la segunda mitad de la lista puede ignorarse. Por otra parte, si el valor buscado es mayor, entonces, se considerará a la segunda mitad de la lista y la primer mitad se ignorará. La segunda prueba compara la llave de la mitad de la lista que queda de la lista original, si el valor buscado es menor, entonces se procede a partir otra vez a la mitad la nueva lista y se considera a la primer mitad. Si el valor a buscar es mayor, se procede a tomar la segunda mitad de la lista. De esta forma, se continúa partiendo las listas donde esté acotado el valor buscado hasta encontrarlo o hasta probar que no existe en la lista.

Por ejemplo, la figura 9-4 muestra la secuencia de pruebas, involucradas en una búsqueda binaria, para la llave con valor 72 en una lista lineal de muestra. Inicialmente hay 16 llaves a considerar. La primer prueba es a la octava llave (16/2): 68. Dado que 72 es mayor que 68, entonces consideraremos la segunda mitad de la lista. La segunda prueba es a la doceava llave ((8 + 16)/2): 81. Dado que 72 es menor que 81, entonces podemos descartar el último cuarto de la lista. Una tercera prueba es para la décima llave ((8 + 12)/2): 72, que es justamente igual al valor buscado. Como se podrá observar, sólo fue necesario aplicar tres comparaciones; una búsqueda lineal hubiera requerido 10 comparaciones. Para listas más largas la eficiencia se incrementa dramáticamente.

La lista se corta a la mitad una y otra vez hasta que algún valor de la mitad de las listas resultantes sea el valor de la llave buscada, o hasta que el tamaño de la lista restante sea de cero, lo cual implica que el valor de la llave buscada no está en la lista.

La técnica de búsqueda binaria puede ser programada tanto por un algoritmo iterativo como por un algoritmo recursivo. Un procedimiento de búsqueda binaria iterativa, escrito en COBOL, sigue. La lista lineal de N llaves es alojada en un arreglo llamado LISTA-DE-LLAVES, el valor de la llave buscado es LLAVE-BUSCADA. ENCONTRO se iguala al subíndice del elemento del arreglo con un valor LLAVE-BUSCADA si existe, y en cero si no es encontrado en la lista.

```
01  LISTA-DE-LLAVES.
    02  VALOR-LLAVE  OCCURS N  TIMES  PICTURE S9(5).
01  VARIABLES-AUXILIARES.
    02  LIMITE-INFERIOR        PICTURE 9(4).
```

```
           02  LIMITE-SUPERIOR        PICTURE 9(4).
           02  MITAD                  PICTURE 9(4).
           02  LLAVE-BUSCADA          PICTURE S9(5).
           02  ENCONTRO               PICTURE 9(4).
       BUSQUEDA-BINARIA.
           COMPUTE ENCONTRO = 0.
           COMPUTE LIMITE-INFERIOR = 1.
           COMPUTE LIMITE-SUPERIOR = N.
           PERFORM PRUEBA UNTIL LIMITE-INFERIOR > LIMITE
                                    SUPERIOR OR ENCONTRO > 0.
```

donde

```
       PRUEBA.
           COMPUTE MITAD = (LIMITE-INFERIOR + LIMITE-SUPERIOR)/2.
           IF LLAVE-BUSCADA = VALOR-LLAVE (MITAD)
                   COMPUTE ENCONTRO = MITAD
           ELSE IF LLAVE-BUSCADA < VALOR-LLAVE (MITAD)
                   COMPUTE LIMITE-SUPERIOR = MITAD - 1
                 ELSE COMPUTE LIMITE-INFERIOR = MITAD + 1
```

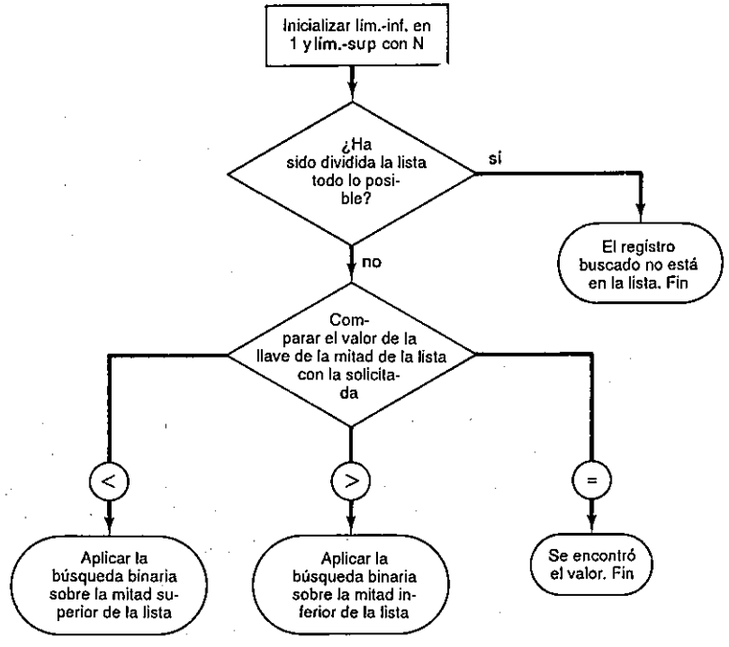


Figura 9-5 Algoritmo de búsqueda binaria recursivo.

En esta rutina, los subíndices LIMITE-INFERIOR, LIMITE-SUPERIOR y MITAD no tomarán valores fraccionales porque fueron declarados como enteros.

El diagrama de flujo del algoritmo de búsqueda binaria recursivo está dado en la figura 9-5. El algoritmo se puede escribir en Pascal como sigue, suponiendo que los nombres de variables declaradas son iguales que los del procedimiento en COBOL, que el arreglo lista-de-llaves y la variable llave-buscada son globales, y que el rango de subíndices para la llave es de 1 a n.

```
procedure búsqueda-binaria (var límite-inferior,límite-superior, encontró : integer)
var mitad : integer;
begin if (límite-inferior > límite-superior) then encontró := 0
        else begin {prueba}
                mitad := (límite-inferior + límite-superior) div 2;
                if (llave-buscada = lista-de-llaves[mitad])
                then encontró := mitad
                else if (llave-buscada < lista-de-llaves[mitad])
                    then búsqueda-binaria(límite-inferior, mitad - 1,encontró)
                    else búsqueda-binaria(mitad + 1,límite-superior,encontró)
        end {prueba}
end {búsqueda-binaria};
```

Este procedimiento se puede invocar por medio de una instrucción, como búsqueda-binaria(1,n,encontró); la variable encontró puede alcanzar el valor del subíndice de la entrada con valor de llave-buscada, o puede volverse cero si la llave-buscada no existe en el arreglo.

El número máximo de comparaciones requeridas para la búsqueda binaria, en una lista lineal ordenada de n elementos, es $\log_2 n$; el número mínimo de comparaciones requeridas es 1. El número esperado de comparaciones es $1/2 \log_2 n$. Por lo tanto, se dice que la búsqueda binaria es de $O(\log_2 n)$.

Una búsqueda binaria deberá ser usada siempre que la lista de elementos sea muy grande y además los elementos estén ordenados en base a la llave de búsqueda. De hecho, es ventajoso construir listas ordenadas para poder utilizar la técnica de búsqueda binaria. Realmente, la indicación de siempre utilizar la técnica de búsqueda binaria podrá ser modificada ligeramente para indicar su aplicabilidad a la búsqueda de colecciones de datos totalmente residentes en memoria principal. La búsqueda binaria, debemos aclarar, es un método de búsqueda interna. El concepto de búsqueda no lineal también puede ser aplicado a conjuntos de datos almacenados en dispositivos de almacenamiento secundario; consideraremos tales organizaciones de datos más tarde en este libro.

INTRODUCCION AL ORDENAMIENTO

Veamos ahora, nuevas técnicas para almacenar un conjunto de registros en orden progresivo lo que, como ya hemos visto, resulta importante para mejorar la eficiencia de búsqueda.

Existen dos categorías básicas de técnicas de ordenamiento: ordenamientos internos y ordenamientos externos. Los métodos de *ordenamiento interno* se aplican cuando el

conjunto de datos a clasificar es lo suficientemente pequeño, de tal forma que pueda caber dentro de la memoria principal. El tiempo requerido para leer o escribir registros no se considera significativo para la evaluación del rendimiento de las técnicas de ordenamiento interno. Algunos métodos de ordenamiento interno se introducen en este capítulo. Los métodos de *ordenamiento externo* se aplican a grandes volúmenes de datos, que residen parcial o totalmente en dispositivos de almacenamiento secundario, tales como discos o cintas magnéticas. Aquí, el tiempo de acceso de lectura y escritura influye en la determinación de la eficiencia del ordenamiento. Más tarde estudiaremos métodos de ordenamiento externo, en la parte de procesamiento de archivos.

Aunque estemos interesados en obtener una lista ordenada de registros, rutinariamente ignoraremos los registros y nos interesaremos solamente en obtener listas ordenadas de llaves. No se confunda con esta simplificación. Si usted prefiere, cuando indiquemos que "una llave" debe ser insertada en una lista, o que "una llave" debe seleccionarse de una lista, o que "dos llaves" en la lista deben intercambiar su posición, interprete que "una llave" es "una llave y su respectivo registro" o "una llave y el apuntador a su correspondiente registro".

Siempre supondremos que el orden de ordenamiento de las llaves es ascendente y usted podrá hacer los cambios necesarios para ordenar en forma descendente el conjunto de llaves.

No requeriremos que los valores de las llaves sean los únicos identificadores de los registros. Un algoritmo de ordenamiento se dice que es *estable* si el ordenamiento original de registros, con llaves iguales se preserva en el ordenamiento de esos registros. Es importante usar un método de ordenamiento estable si el *ordenamiento* original de registros es significativo. Por ejemplo, este ordenamiento se pudo haber establecido para conservar la cronología de entrada de los registros en la lista original. No todos los métodos de ordenamiento son estables. Como en nuestra discusión sobre técnicas de búsqueda lineal, asumimos aquí que la lista lineal por clasificar está representada por un arreglo. Ustedes deberán determinar cómo la representación de listas ligadas modifica los algoritmos.

ORDENAMIENTO POR SELECCION

Una familia de algoritmos de ordenamiento interno, es el grupo de *ordenamiento por selección*. La idea básica de un ordenamiento por selección es la selección repetida de la llave menor restante en una lista de datos no clasificados, como la siguiente llave, en una lista de datos ordenada que crece (Figura 9-6).

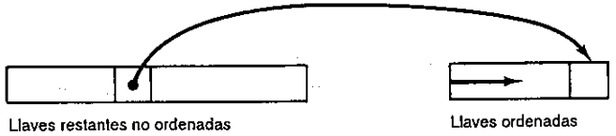


Figura 9-6 Ordenamiento por selección.

La totalidad de la lista de llaves no ordenadas, debe estar disponible, para que

nosotros podamos seleccionar la llave con valor mínimo en esa lista. Sin embargo, la lista *ordenada*, podrá ser puesta en la salida, a medida que avancemos.

Por ejemplo, considere la siguiente lista de llaves no ordenadas:

14 3 22 9 10 14 2 7 25 6

El primer paso de selección identifica el 2 como valor mínimo, lo saca de dicha lista y lo agrega como primer elemento en una nueva lista ordenada.

14 3 22 9 10 14 7 25 6 — 2

llaves restantes no ordenadas — lista ordenada

El segundo paso identifica el 3 como el siguiente elemento mínimo y lo retira de la lista para incluirlo en la nueva lista de elementos ordenados.

14 22 9 10 14 7 25 6 — 2 3

llaves restantes no ordenadas — lista ordenada

Después del sexto paso, tenemos la siguiente lista.

14 22 14 25 — 2 3 6 7 9 10

llaves restantes no ordenadas — lista de elementos ordenados

En COBOL, la *ordenación por selección* directa es:

```
01  LISTA-NO-ORDENADA.
    02  ENTRADA-NO-ORDENADA      OCCURS N TIMES      PIC S99.
01  LISTA-ORDENADA.
    02  ENTRADA-ORDENADA         OCCURS N TIMES      PIC S99.
01  ELEMENTOS-AUXILIARES.
    02  LLAVE-MINIMA             PIC S99 VALUE HIGH-VALUES
    02  SIGUIENTE                PIC 9(3)      VALUE 1.
    02  I                        PIC 9(3).
    02  POSICION                 PIC 9(3).

    PERFORM SELECCIONA-AL-SIGUIENTE N TIMES.
```

donde

```
SELECCIONA-AL-SIGUIENTE.
    PERFORM IDENTIFICA-AL-MINIMO VARYING I FROM 1 BY 1
                                 UNTIL I > N.
    MOVE ENTRADA-NO-ORDENADA (POSICION) TO
         ENTRADA-ORDEN (SIGUIENTE).
```

```
MOVE HIGH-VALUES TO ENTRADA-NO-ORDEN (POSICION).
MOVE HIGH-VALUES TO LLAVE-MINIMA.
```

e

```
IDENTIFICA-AL-MINIMO.
   IF ENTRADA-NO-ORDENADA (I) < LLAVE-MINIMA.
         MOVE I TO POSICION
         MOVE ENTRADA-NO-ORDENADA (I) TO LLAVE-MINIMA
```

Para una lista de n registros, este algoritmo requiere n pases sobre la lista no ordenada. En el i-ésimo pase, se habrán hecho $n - i$ comparaciones de llaves. Por lo tanto el número total de comparaciones será:

$$\sum_{i=1}^{n} (n - i)$$

La cual da $n(n - 1)/2$. Esta ordenación se dice que requiere de $O(n^2)$ comparaciones, porque el término n^2 domina a la expresión. El número de comparaciones es proporcional al cuadrado del número de llaves en el conjunto. Así que, duplicar el número de llaves, significará que el proceso tomará cuatro veces más tiempo.

ORDENAMIENTO POR SELECCION CON INTERCAMBIO

El programa anterior requiere dos veces más espacio del necesario. Una modificación al método directo de ordenación por selección es el *ordenamiento por selección con intercambio*, en el cual el valor de la llave seleccionada es movido a su posición final por intercambio con la llave que inicialmente ocupaba esa posición. Consideremos otra vez la lista no clasificada:

14 3 22 9 10 14 2 7 25 6

Después del primer paso, 2 es seleccionado,

2	3 22 9 10 14 14 7 25 6
ordenada	no ordenada

Después del segundo paso, 3 es seleccionado,

2 3	22 9 10 14 14 7 25 6
ordenada	no ordenada

Después del sexto paso,

2 3 6 7 9 10	14 14 25 22
ordenada	no ordenada

Note que éste no es un ordenamiento estable. Por ejemplo, aquí el primer paso colocó al primer 14 después del segundo 14; en los pasos séptimo y octavo, los 14's se agregan a la lista ordenada en ese orden invertido.

En COBOL, el ordenamiento por selección con intercambio es:

```
01 LISTA-DE-LLAVES.
   02 LLAVE                  OCCURS N TIMES     PIC S99.
01 ELEMENTOS-AUXILIARES.
   02 LLAVE-MINIMA           PIC S99            VALUE HIGH-VALUES.
   02 SIGUIENTE              PIC 9(3)           VALUE 1.
   02 I                      PIC 9(3).
   02 POSICION               PIC 9(3).
   02 LLAVE-TEMPORAL         PIC S99.

MANEJADOR-DE-INTERCAMBIO.
   PERFORM SELECCIONA-SIG VARYING SIGUIENTE FROM
                              1 BY 1 UNTIL SIGUIENTE = N.
```

donde

```
SELECCIONA-SIG.
   PERFORM IDENTIFICA-LLAVE VARYING I FROM
                        SIGUIENTE BY 1. UNTIL
                        I > N.
   MOVE LLAVE (SIGUIENTE)
         TO LLAVE-TEMPORAL.
   MOVE LLAVE (POSICION)
         TO LLAVE (SIGUIENTE).
   MOVE LLAVE-TEMPORAL
         TO LLAVE (POSICION).
   MOVE HIGH-VALUES
         TO LLAVE-MINIMA.
```

e

```
IDENTIFICA-LLAVE.
   IF LLAVE (I) < LLAVE-MINIMA
         MOVE I TO POSICION
         MOVE LLAVE (I) TO LLAVE MINIMA.
```

El ordenamiento por selección con intercambio tiene esencialmente los mismos requerimientos de comparación que el ordenamiento por selección directa; es de $O(N^2)$.

Más tarde consideraremos otros miembros de la familia de ordenamientos por selección que son más eficaces que éste.

ORDENAMIENTO POR INSERCION

Otra familia de algoritmos de ordenamiento interno es el grupo de *ordenamiento por inserción*. La idea básica de una clasificación por inserción es tomar la siguiente llave de una lista no clasificada e insertarla en su posición relativa correspondiente en una lista creciente de datos clasificados (Figura 9-7).

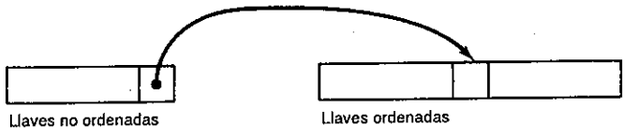


Figura 9-7 Ordenamiento por inserción.

La lista completa de llaves ordenadas deberá estar disponible, a lo largo del proceso, para poder insertar una llave en su posición relativa apropiada. La lista puede alimentarse durante el proceso.

Compare esta técnica de ordenamiento con el método de ordenamiento por selección de la sección anterior. Cuando usted ordena una mano de cartas de juego, si usted toma cada carta cuando se la sirven y la coloca en la "ranura" apropiada, con respecto a las demás cartas que ya tiene, entonces está usando un ordenamiento por inserción. Sin embargo, si usted espera hasta que la mano entera se reparta, después procede a identificar qué carta debe ir más a la izquierda y la coloca, y así para cada una, se dice entonces que está usando ordenamiento por selección.

Ambas técnicas son relativamente pobres, pues cada una es de $O(N^2)$, pero cada una es relativamente fácil de comprender y de programar. Ambas son muy utilizadas.

Considere otra vez nuestro ejemplo de lista de llaves no clasificadas.

14 3 22 9 10 14 2 7 25 6

En el primer paso se considera a la primera llave, la cual es 14, y el resultado es:

3 22 9 10 14 2 7 25 6 — lista no ordenada

14 — lista ordenada

Después del segundo paso:

22 9 10 14 2 7 25 6 — lista no ordenada

3 14 — lista ordenada

Después del sexto paso:

2 7 25 6 — lista no ordenada

3 9 10 14 14 22 — lista ordenada

Como en el ordenamiento por selección con intercambio, el ordenamiento por inserción puede ser programado para ordenar en el mismo espacio de trabajo. En Pascal

```
procedure ordenamiento-por-inserción;
var llave : array [1 . . n] of integer;
    llave-auxiliar : integer;
    i, j, jj : 1 . . n;
    encontró : boolean;
begin
    for i := 2 to n
    do begin j := 1;
          encontró := false;
          while (not encontró) and j < i
          do if(llave [i] < llave [j])
             then encontró := true
             else j := j + 1;
          if {encontró}
          then begin llave-auxiliar := llave [i];
                     jj := i - 1;
                     while (jj > j - 1)
                     do begin llave [jj + 1] := llave [jj];
                              jj := jj - 1
                        end;
                     llave[j] := llave-auxiliar;
               end;
    end;
end.
```

Este ordenamiento aquí programado es estable. ¿Por qué? ¿Qué cambios al código lo harían inestable?

Una variación en el ordenamiento por inserción directa se basa en el conocimiento previo de la distribución de los valores de la llave representados en la lista. En un intento de reducir la cantidad de movimientos de datos, las llaves son insertadas en sus posiciones relativas, dejando algunas ranuras vacías entre las llaves, para acomodar las llaves que deben ser insertadas en los pasos subsecuentes.

Más adelante consideraremos otros miembros de la familia de ordenamientos por inserción, que muestran un mejor comportamiento que $O(N^2)$.

ORDENAMIENTO POR INTERCAMBIO: EL METODO DE LA BURBUJA

Una tercera familia de algoritmos de ordenamiento interno es el grupo de *ordenamientos por intercambio*. La idea básica del ordenamiento por intercambio es la de comparar

pares de valores de llaves e intercambiarlos si no están en sus posiciones relativas correctas (Figura 9-8).

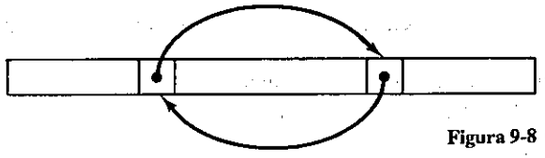


Figura 9-8 Ordenamiento por intercambio.

Un buen comportamiento del ordenamiento puede resultar de la identificación inteligente de los pares de llaves comparadas. En esta sección, sin embargo, primero consideraremos un ordenamiento por intercambio no tan inteligente: el *método de la burbuja*. Como los ordenamientos por selección e inserción presentados en este capítulo, el método de la burbuja requiere $O(N^2)$ comparaciones. No obstante, el método de la burbuja es frecuentemente usado. Más adelante investigaremos mejores métodos de ordenamiento por intercambio.

La idea básica del método de la burbuja es la de permitir que cada llave flote a su posición adecuada a través de una serie de pares de comparaciones e intercambios con los valores adyacentes. Cada paso hace que una llave suba a su posición final, como una burbuja, en la lista ordenada.

Considere otra vez nuestro ejemplo de lista de llaves no ordenadas.

6
25
7
2
14
10
9
22
3
14

Como las burbujas siempre suben, entonces, coloquemos la lista de llaves verticalmente con la primera llave en el fondo. Cada llave se compara con la llave que está encima de ella y se intercambia, si la llave de arriba es más pequeña. Cuando una llave mayor que la llave sujeto se encuentra, la llave sujeto queda encima, y el proceso continúa. Después de la pasada, todas las llaves arriba de la última por intercambiar deberán estar en su posición final. No necesitarán examinarse en pasos posteriores.

La actividad del primer paso sube a 14, a 22, y a 25.

6 25
25 6
7 22
2 22 7
14 22 2
10 22 14
9 22 10
22 9
3 14
14 3

el resultado es

25
6
22
7
2
14
10
9
14
3

Ahora sabemos que 25 está en su posición final.

La actividad del segundo paso sube a 14, a 14, y a 22

6 22
22 6
7 14
2 14 7
14 2
10 14
9 14 10
14 9
3

el resultado es

25
22
6
14
7
2
14
10
9
3

Y sabemos que ahora 22 está en su posición final.

La actividad del tercer paso sube a 14 y a 14.

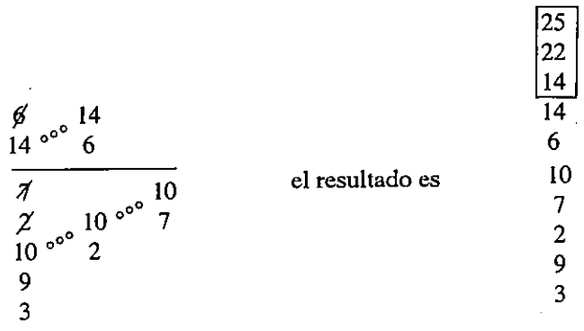
6 14
14 6
7 14
2 14 7
14 2
10
9
3

el resultado es

25
22
14
6
14
7
2
10
9
3

Así, sabemos que 14 está en su posición final.

La actividad del cuarto paso, sube a 10 y a 14.



el resultado es

y así sucesivamente.

El método de la burbuja en realidad es muy poco recomendado, sin embargo, es muy conocido (tal vez debido a su nombre) y desafortunadamente muy utilizado (puede ser debido a su relativa facilidad de implantación). Su comportamiento es un poco parecido al método de intercambio selectivo, en el cual las llaves más pequeñas bajan al fondo de la lista.

ORDENAMIENTO POR PARTICION E INTERCAMBIO (ORDENAMIENTO RAPIDO O QUICKSORT)

Consideremos ahora, un método de ordenamiento por intercambio mucho más efectivo: el *ordenamiento por partición e intercambio*, también conocido como método *quicksort de Hoare* o simplemente como *ordenamiento rápido*. Este método es atribuido a C. A. R. Hoare (1962).

La idea básica del método quicksort es la de usar los resultados de cada paso de comparación para guiar el siguiente paso de comparación. Durante un paso de comparación, las llaves se intercambian de tal forma que, cuando el paso es completado, la lista se ha particionado de tal manera que, los valores de las llaves en una partición (aunque desordenados) son todos menores que el valor de una llave en particular y, los valores de las llaves en la otra partición (aunque desordenados) son todos mayores que el valor de la llave en particular. El siguiente paso de comparación puede proseguir después con las dos particiones resultantes, tratadas de manera independiente. Los pasos de comparación se localizan sucesivamente con particiones más y más pequeñas.

Quizás esta técnica de ordenamiento se explique mejor con un ejemplo. Consideremos otra vez las lista de llaves que hemos venido manejando.

6
25
7
2
14

10
9
22
3
14

La primera fase de comparación tiene como objetivos:

1. Identificar la posición final de la primera llave sujeto (aquí es 6) en la lista ordenada.
2. Particionar la lista en dos partes, una de las cuales contiene sólo valores de llave menores que la primera llave sujeto, y la otra contiene solamente los valores mayores que la primera llave sujeto.

Para alcanzar estos objetivos, la llave sujeto se compara primero con el último elemento (aquí es el 14) de la lista, después con el penúltimo elemento (aquí es 3), hasta que se encuentre una llave que pueda preceder, en lugar de seguir, a la llave sujeto; tal llave intercambia después posiciones con la llave sujeto (Figura 9-9a)). La comparación después prosigue, con la *misma llave sujeto*, desde la otra dirección; 6 se compara con 25. Debido a que los dos no están en su posición relativa apropiada, se efectúa un

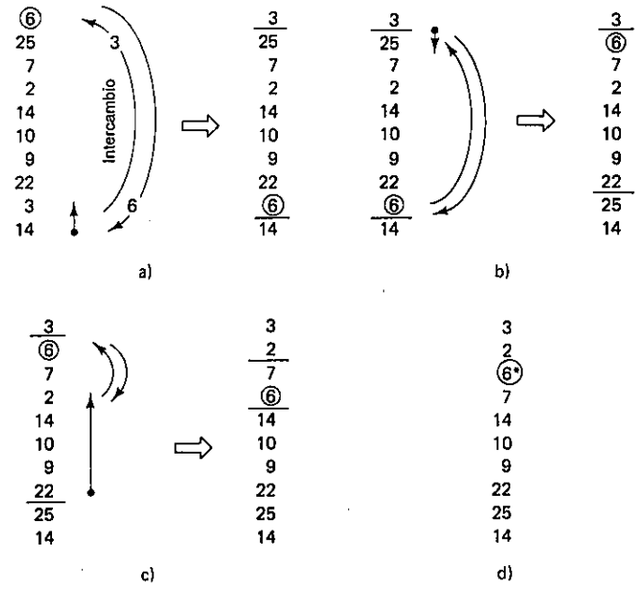


Figura 9-9 Pasos del método Quicksort

intercambio (Figura 9-9b)). Ahora, la comparación cambia de dirección otra vez, de abajo hacia arriba. El 6 es comparado con 22; con 9, con 10, con 14, después con 2, el cual está en el lado "equivocado" con respecto a 6. Otro intercambio ocurre (Figura 9-9c)). La comparación cambia de nuevo de dirección, de arriba hacia abajo. El 6 se compara con 7, el cual está en el lado equivocado con respecto a 6. Un intercambio ocurre para completar el paso en la figura 9-9d). En este punto, 6 se encuentra en su posición final; todo lo que está antes de 6, es menor de 6; y todo lo que está después es mayor de 6. Las dos particiones resultantes ahora pueden ordenarse independientemente.
Considere la segunda partición.

7
14
10
9
22
25
14

El paso de comparación sobre esta partición muestra que 7 está ya en su posición adecuada. La siguiente partición a considerar es:

14
10
9
22
25
14

El paso prosigue con 14 como llave sujeto (Figura 9-10).

A medida que los tamaños de las particiones se acortan, se podrá aplicar un algoritmo de ordenamiento más sencillo, como el de inserción o selección directa, para reducir el sobrepaso.

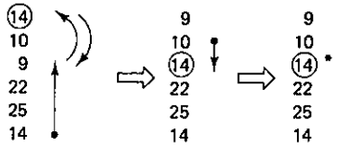


Figura 9-10 Paso final del método Quicksort.

Rendimiento

El análisis del algoritmo quicksort es bastante directo. Consideremos primero el caso promedio; supongamos que la posición final adecuada para el primer elemento K_i, se encuentra a la mitad de la lista no ordenada (Figura 9-11).

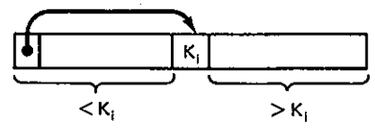


Figura 9-11 Paso del método Quicksort.

Del primer paso resulta la partición de la lista en dos sublistas, cada una de aproximadamente $n/2$ llaves. Definamos un "superpaso" como un paso que procesa la lista entera. El primer superpaso requiere $n - 1$ comparaciones.

El segundo superpaso involucra al procesamiento de ambas sublistas (Figura 9-12).

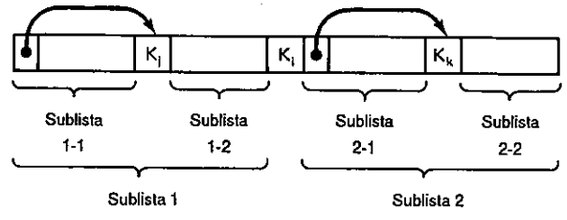


Figura 9-12 Segundo superpaso del método Quicksort.

Procesar la sublista 1 requiere aproximadamente $^1/_2(n - 3)$ comparaciones, lo mismo que para procesar la lista 2. De hecho, el proceso de cada superpaso requiere $O(n)$ comparaciones.

En promedio se requieren $\log_2 n$ superpasos para clasificar la lista completa. Luego el método de quicksort requiere de un promedio de $O(n\log_2 n)$ comparaciones. Este es el mejor desempeño encontrado hasta aquí en una técnica de ordenamiento; recordemos que los métodos lineales requieren todos de $O(n^2)$ comparaciones.

En algunos casos, el método quicksort requiere más de $\log_2 n$ superpasos. De hecho en el peor de los casos, se necesitarán n superpasos. El peor de los casos sucede cuando la lista inicial ya está ordenada. En lugar de que tienda a reducirse a la mitad el número de comparaciones que necesita en cada superpaso, sólo se reduce en una comparación en cada superpaso. El comportamiento, en el peor de los casos, de $O(n^2)$ es tan malo como el de los algoritmos de ordenamiento lineal. Sin embargo, es posible lograr que el método quicksort no llegue nunca a caer en el peor de los casos. En sí, es un muy buen método.

ORDENAMIENTO POR APILAMIENTO (HEAPSORT)

El método quicksort en promedio mejora la eficiencia de ordenamiento con respecto a la de los algoritmos de ordenamiento lineal. En esta sección consideraremos otro método

importante de ordenamiento no lineal, el *ordenamiento por apilamiento*, conocido también como *Heapsort*, cuyo desempeño es en promedio tan bueno como el del quicksort y que se comporta mejor que este último en los peores casos. Aunque el heapsort tiene un mejor desempeño general que cualquier otro método presentado de clasificación interna, es bastante complejo de programar. (¡Nada es gratis!) Una adecuada programación del quicksort, es algunas veces más rápida que una buena programación del heapsort y es bastante menos compleja de programar. El heapsort fue desarrollado en 1964 por J. W. J. Williams.

El heapsort está basado en el uso de un tipo especial de árbol binario (llamado apilamiento) para estructurar el proceso de ordenamiento. La estructura de ramificación del árbol, conserva el número de comparaciones necesarias en $O(n\log_2 n)$.

Hay dos fases en el heapsort.

1. Creación del apilamiento.
2. Procesamiento del apilamiento.

En la primera fase, las llaves no ordenadas son colocadas en un árbol binario de tal manera que constituyan un árbol heap.

Estructura del apilamiento

Un apilamiento de tamaño n, es un árbol binario de n nodos que se adhiere a las siguientes restricciones.

1. El árbol binario está *casi completo*, es decir, existe un entero k tal que:
   a) cada hoja del árbol está en el nivel k o nivel $k + 1$ y
   b) si un nodo tiene un subárbol derecho en el nivel $k + 1$ entonces ese nodo también tiene un subárbol izquierdo en el nivel $k + 1$.
2. Las llaves están acomodadas en los nodos de tal manera que, para cada nodo i

   $K_i \leq K_j$

   donde el nodo j es el padre del nodo i.

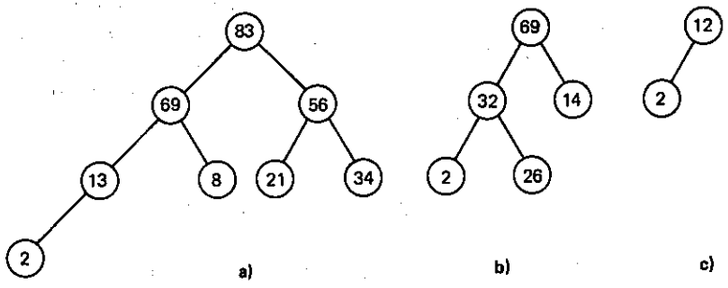


Figura 9-13 Ejemplos de apilamientos.

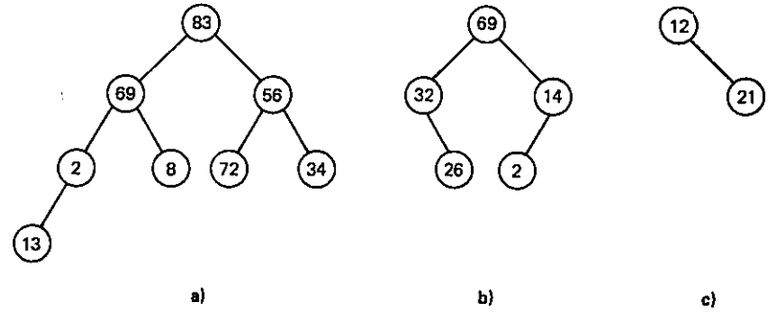


Figura 9-14 Arboles binarios que no son apilamientos.

La primer restricción significa que los niveles del apilamiento se llenen de izquierda a derecha, y que ningún nodo se coloque en un nuevo nivel, hasta que el nivel anterior esté lleno. Los árboles binarios de la figura 9-13 son de tipo apilamiento. Note que estos árboles no son árboles binarios de búsqueda, porque el recorrido en en-orden no toma la secuencia de los nodos en orden ascendente de llaves.

Los árboles binarios de la figura 9-14 *no* son apilamientos. La restricción de colocación de las llaves es violada en los nodos 2-13 y 56-72, en el árbol a); la restricción de estar casi completo es violada en el nivel 2 del árbol b); la restricción de colocación de las llaves y la restricción de estar casi completo son violadas en el árbol c).

Creación del apilamiento

Consideremos primero la fase de creación del apilamiento. En esta fase pasamos secuencialmente a través de las llaves no ordenadas, colocándolas en un apilamiento. El tamaño del apilamiento crece con cada nueva llave que se agrega. Para crear un apilamiento de tamaño i, la i-ésima llave (K_i) se coloca dentro de un apilamiento existente, de tamaño $i - 1$. El nodo es posicionado primero de tal manera que la condición de "casi completo" sea satisfecha. Después, el valor de K_i se compara con el valor de la llave del padre del nodo. Si K_i es mayor, entonces el contenido de este nuevo nodo y el nodo padre se intercambian. Este proceso de comparación e intercambio se repite hasta que, el valor de la llave del padre no sea menor que K_i, o que K_i se encuentre en la raíz del árbol. De está manera el árbol será calificado como un apilamiento de tamaño i. De manera más formal, en Pascal, para crear un apilamiento de tamaño i, agregando una llave (llamada aquí, nueva-llave) un apilamiento (conocido globalmente, llamado llave y alojado en un arreglo) de tamaño $i - 1$ (donde $i \geq 1$):

```
procedure crea apilamiento-i (i, nueva-llave:integer);
var padre, auxiliar, siguiente : integer;
begin siguiente := i;
        padre := siguiente div 2;
        llave[siguiente] := nueva-llave;
```

```
while (siguiente <> 1 and llave[padre] < = llave[siguiente])
do begin {se intercambian padre e hijo}
        auxiliar := llave[padre];
        llave[padre] := llave[siguiente];
        llave[siguiente] := auxiliar;
        {avanza sobre el árbol}
        siguiente := padre;
        padre := siguiente div 2
    end;
end;
```

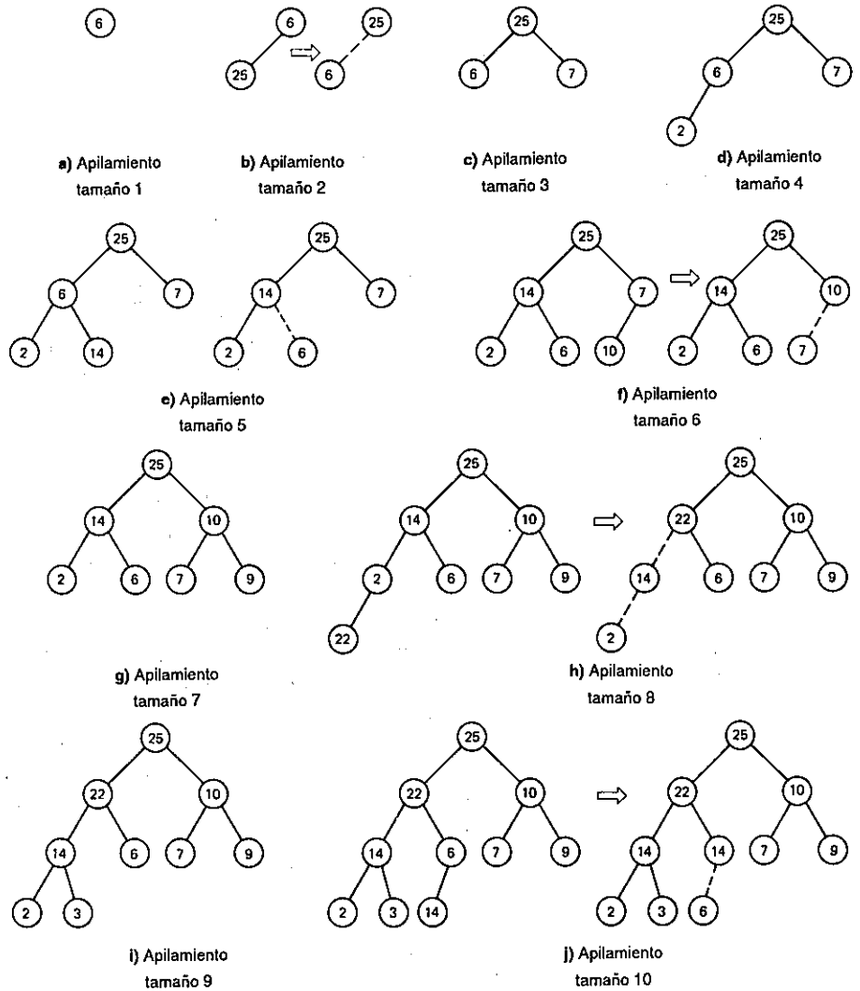


Figura 9-15 Pasos en la construcción de un apilamiento de ejemplo.

Este procedimiento se llama cada vez que se agrega una nueva llave al apilamiento.

La figura 9-15 ilustra la construcción del apilamiento que contenga la siguiente lista de llaves no ordenadas: 6, 25, 7, 2, 14, 10, 9, 22, 3, 14. Una línea punteada en la figura indica que los nodos en los extremos de la arista se han intercambiado.

Este apilamiento podría representarse por un arreglo, como se muestra en la figura 9-16. Note que el nodo i es el padre de los nodos $2i$ y $2i + 1$. Luego, puesto que el árbol es un apilamiento, la llave (i) ≤ llave (i/2), donde se asume el uso de la división de enteros.

Si las llaves estuvieran inicialmente en un orden diferente, en la lista no ordenada, entonces el apilamiento resultante sería diferente. Por ejemplo, la lista no ordenada 9, 14, 10, 22, 7, 25, 3, 14, 6, 2 produce el montón mostrado en la figura 9-17.

Subíndice	1	2	3	4	5	6	7	8	9	10
Llave	25	22	10	14	14	7	9	2	3	6

Figura 9-16 Representación de un arreglo del montón de la figura 9-15j)

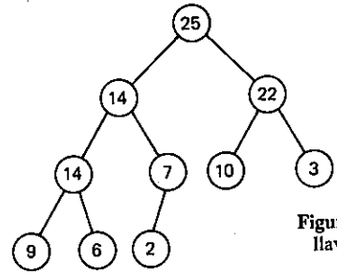


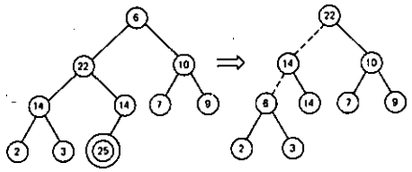
Figura 9-17 Otro montón para el mismo conjunto de llaves de la figura 9-15. Presentado en diferente orden.

Procesamiento del apilamiento

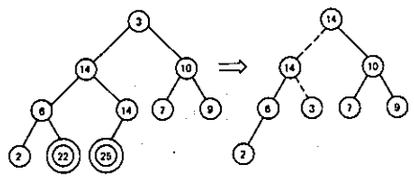
Recordemos que el objetivo del ordenamiento por apilamiento es el de obtener una lista de llaves ordenadas. Lo único que tenemos en este momento es un apilamiento de llaves. La fase de procesamiento del ordenamiento del apilamiento, recorre el apilamiento, de tal forma que, el resultado es una lista ordenada de llaves. Note que la llave más grande del apilamiento está siempre en el tope, y que la fase de proceso está basada en este hecho. Después de crear el apilamiento, el elemento del tope es otra vez removido y así, sucesivamente, hasta que el apilamiento resultante sea de tamaño 0.

De manera más formal, un procedimiento en Pascal para procesar un apilamiento de tamaño n, es el siguiente. Utilizaremos la ventaja del hecho de que el apilamiento está todavía alojado en un arreglo llamado llave (el cual resulta del procedimiento crea-apilamiento-i); de que el mayor valor de llave está en el tope del montón; llave[1]. Sucesivamente, moveremos este valor hasta el final del arreglo (llave[n], la primera vez, a través del ciclo), después ajustaremos el apilamiento a un tamaño n - 1. Al final, las llaves en el arreglo estarán ordenadas; la llave [1] con el valor más pequeño y la llave [n] con el valor más grande.

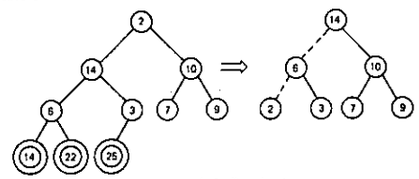
```
procedure proceso-apilamiento-n;
var padre, hijo, última, llave-anterior : integer;
for última := n downto 2
do begin {mueve la llave raíz al último lugar}
        llave-anterior := llave[última];
        llave[última] := llave[1];
        {ajusta el árbol al tamaño último - 1}
        padre := 1;
        {busca al mayor de los hijos de la raíz}
        if(última - 1 > = 3) and (llave[3] > llave[2])
        then hijo := 3
        else hijo := 2;
        {mueve las llaves hacia arriba hasta encontrar lugar}
        {para la llave-anterior almacenada}
        while (hijo < = última - 1 and llave[hijo] > llave-anterior)
do begin llave[padre] := llave[hijo];
        padre := hijo;
        hijo := padre * 2;
        {encuentra al mayor de los hijos del padre}
        if(hijo + 1 < = última - 1) and (llave[hijo + 1] > llave[hijo])
        then hijo := hijo + 1
```



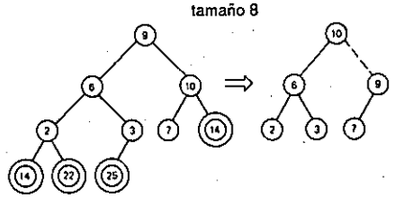
a) Apilamiento tamaño 9



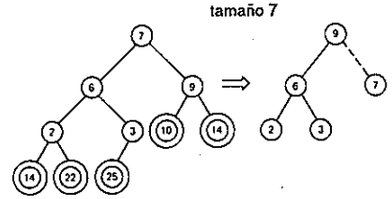
b) Apilamiento tamaño 8



c) Apilamiento tamaño 7

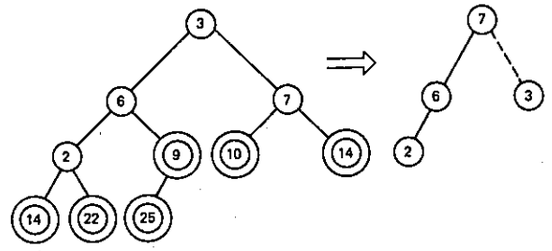


d) Apilamiento tamaño 6

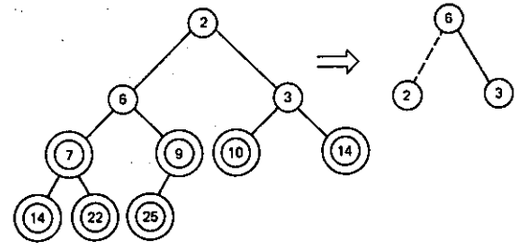


e) Apilamiento tamaño 5

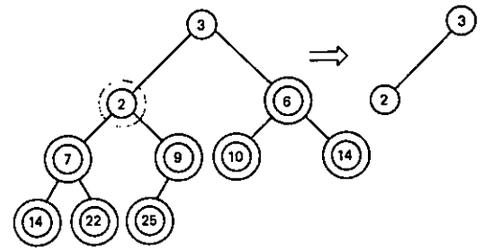
Figura 9-18 Procesamiento del montón de la figura 9-15.



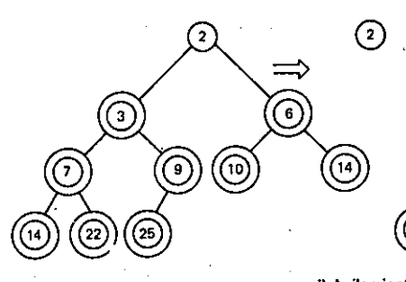
f) Apilamiento tamaño 4



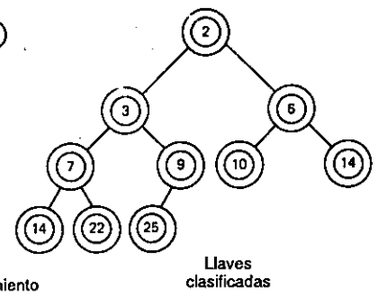
g) Apilamiento tamaño 3



h) Apilamiento tamaño 2



i) Apilamiento tamaño 1



Llaves clasificadas

Figura 9-18 (*Cont.*)

```
        end;
        llave[padre] := llave-anterior
    end;
```

La figura 9-18 ilustra el procesamiento del apilamiento construido en la figura 9-15. Los nodos con doble círculo son nodos que han sido movidos a su posición final en el arreglo, y ya no forman parte constitutiva del apilamiento. Una línea punteada indica que los nodos en los extremos de las aristas se han intercambiado, mientras se ajusta al árbol para que sea otra vez un apilamiento. Después del n-1 ésimo paso (aquí es el 9º), las llaves pueden ser leídas secuencialmente (desde 1 hasta n) dentro del arreglo que aloja al apilamiento y se encontrarán en orden.

De igual manera que para el algoritmo del método quicksort, el análisis del algoritmo de ordenamiento por apilamiento es bastante directo. Durante la fase de creación del apilamiento, la inserción de la i-ésima llave requiere de $O(\log_2 i)$ comparaciones e intercambios, aún en el peor de los casos en el que las llaves llegan en orden. La inserción de una llave en el k-ésimo nivel puede forzar la comparación e intercambio con un máximo de otras k llaves a lo largo de la ramificación hasta la raíz del apilamiento. De igual manera, durante la fase de procesamiento del apilamiento, el tratamiento de un apilamiento de tamaño i, requiere $O(\log_2 i)$ comparaciones e intercambios, aun en el peor de los casos. Los movimientos se pueden hacer solamente a lo largo de una de las ramificaciones del apilamiento.

Por lo tanto, en promedio, el número requerido de comparaciones e intercambios es:

$$\frac{1}{2}\sum_{i=2}^{n} \log_2 i \quad + \quad \frac{1}{2}\sum_{i=2}^{n} \log_2 i$$

lo cual es $(n - 1)\log_2 n$. En el peor de los casos, este número es:

$$\sum_{i=2}^{n} \log_2 i \quad + \quad \sum_{i=2}^{n} \log_2 i$$

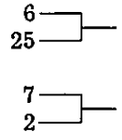
lo cual es $2(n - 1)\log_2 n$. El peor caso no es mucho peor que el caso promedio. El método de ordenamiento por apilamiento por lo tanto, *garantiza* un tiempo de proceso de $O(n\log_2 n)$. Para un tamaño n muy grande, la complejidad del algoritmo está compensada por la eficiencia del método.

ORDENAMIENTO POR TORNEO

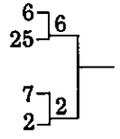
Otro método importante de ordenamiento, también basado en árboles, es el *ordenamiento por torneo*. Este método de ordenamiento algunas veces es llamado *ordenamiento por selección de árbol*, y es atribuido a E. H. Friend (1956). El proceso de ordenamiento por torneo se parece a los árboles de torneos eliminatorios, utilizados en contiendas deportivas (especialmente en tenis, ping pong, squash). Se organizan competencias entre pares de jugadores hasta establecer finalmente a un ganador.

El método de ordenamiento por torneo se usa quizás, más frecuentemente que cualquier otro método. Muchos vendedores de software incorporan este método en sus paquetes de ordenamiento de archivos. Por el momento estudiaremos aquí el funcionamiento del método de ordenamiento por torneo y, en el capítulo 12, veremos cómo utilizarlo en el ordenamiento de archivos.

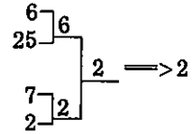
Consideremos nuevamente nuestra lista de llaves sin ordenar: 6, 25, 7, 2, 14, 10, 9, 22, 3, 14, aumentándole 8, 12, 1, 30, 13. Para propósitos de la exposición, supongamos que las restricciones de espacio disponible nos limitan a almacenar únicamente cuatro de estas llaves a la vez en memoria. El método de ordenamiento por torneo asocia estas cuatro llaves en dos pares.



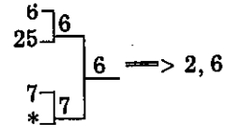
Suponiendo que ejecutamos un ordenamiento ascendente, los ganadores de estos partidos son 6 y 2, los cuales pueden ser enfrentados para las finales.



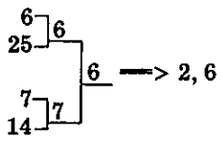
El 2 gana, y es la primera llave en salir del proceso del ordenamiento.



El siguiente paso del torneo, identificará la segunda llave de la lista ordenada. El 2 ya no puede participar en este segundo paso. Sólo podemos jugar con el 7, el cual ganará el primer juego por omisión y después jugará contra el 6.

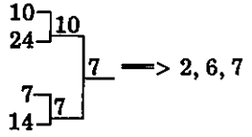


Esto resultaría en un ordenamiento por selección de árbol. Sin embargo, en lugar de eso podemos confrontar otras llaves en el torneo, del resto de la lista no ordenada: 14, 10, 9, 22, 3, 14. Entonces podemos jugar el paso segundo del torneo.

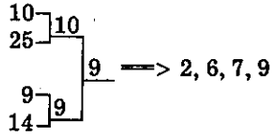


Esta política de substitución, permitirá al método generar cadenas ordenadas más largas que si no hubiéramos introducido nuevos competidores. En base a esto, a nuestro método podemos llamarle ahora con más propiedad como *ordenamiento por selección y reemplazo de árbol.*

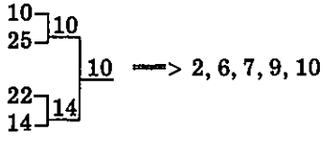
El tercer paso es entonces:



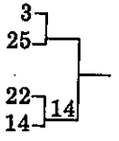
El cuarto paso es:



y el quinto paso es:



En el sexto paso se introduce al 3 en el árbol:

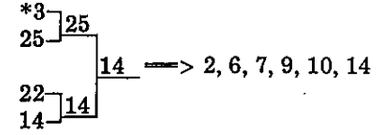


Si al 3 se le permite competir, la secuencia de ordenamiento de la lista de salida será desconfigurada. Para evitar este problema, el método de ordenamiento por selección y

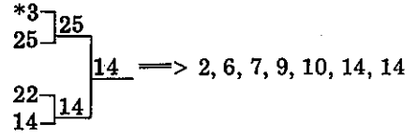
reemplazo invoca la siguiente regla:

> Si $Llave_{nueva} < Llave_{última\ en\ salir}$
> entonces $Llave_{nueva}$ se coloca en el árbol pero es temporalmente descalificada.

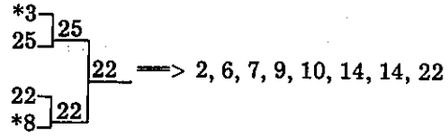
Nosotros marcaremos las llaves descalificadas con un asterisco. Así, el sexto paso tendrá el siguiente resultado:



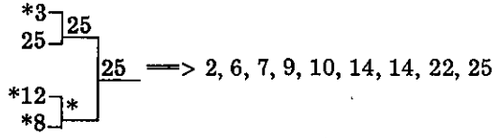
La siguiente llave de la lista no ordenada (el valor 14) es confrontado internamente y puede jugar, puesto que no es menor que 14.



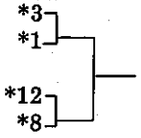
En el paso 8, entra el valor 8, el cual debería ser descalificado, porque es menor que 14, pero los otros valores que entran pueden concursar.



El paso 9 resulta en

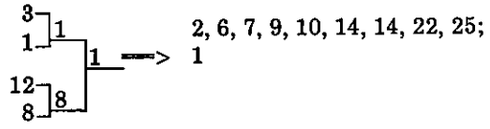


En el paso 10 entra el valor 1.

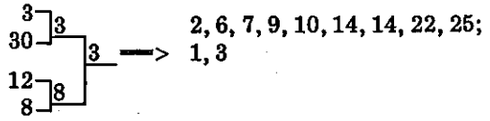


Ahora todas las entradas están descalificadas. Note que, aun cuando sólo se le permitió participar a cuatro jugadores en cada torneo, la cadena ordenada resultante contiene 9

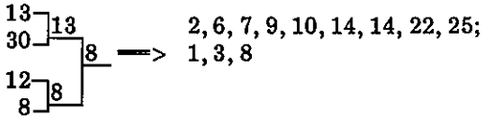
llaves. Ahora quitamos las marcas de descalificación y volvemos a empezar con el torneo. Esta vez generaremos una *segunda* cadena de ordenada. Así, el paso 10 es



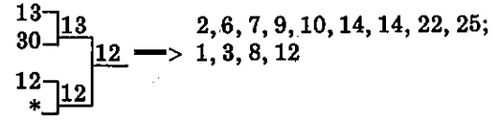
En el paso 11 concursa el valor 30.



El paso 12 es:



Ahora la lista de entrada está vacía, así que podemos meter los valores restantes y continuar con el torneo. El paso 13 resulta en:



Los pasos 14 y 15 vacían el árbol.

Con este ejemplo hemos obtenido dos listas ordenadas:

2, 6, 7, 9, 10, 14, 14, 22, 25;

1, 3, 8, 12, 13, 30

Estas dos cadenas de llaves pueden ser mezcladas para formar una sola cadena de llaves ordenadas. Los primeros elementos de cada una de las cadenas son comparados y el que tenga el valor menor es sacado a la salida y removido para no considerarlo posteriormente. Esta *mezcla en dos sentidos* sencilla se muestra en el diagrama de flujo de la figura 9-19. Puede ser programada en Pascal como sigue. Los arreglos llave-1 y llave-2 contendrán las listas ordenadas iniciales; el arreglo mezcla contendrá la lista mezclada resultante de las dos listas; n1 y n2 son los subíndices del siguiente elemento a ser considerado en los arreglos llave1 y llave2; longitud1 y longitud2 son el número de elementos en los arreglos llave1 y llave2 respectivamente; sig-llave es el subíndice que muestra el lugar para el siguiente elemento en la lista mezclada.


```
begin
        sig-llave := 1;
        n1 := 1;
        n2 := 1;
        while (n1 < = longitud1) and (n2 < = longitud2)
        do begin if (llave1[n1] < = llave2[n2])
                then begin mezcla[sig-llave]: = llave1[n1];
                        n1 := n1 + 1
                    end;
                else begin mezcla[sig-llave] := llave2[n2];
                        n2 := n2 + 1
                    end;
                sig-llave := sig-llave + 1
        end;
        while (n1 < = longitud1) {acomoda a los valores restantes de la llave1}
        do begin mezcla[sig-llave] := llave1[n1];
                n1 := n1 + 1;
                sig-llave := sig-llave + 1
        end;
        while (n2 < = longitud2) {acomoda a los valores restantes de llave2}
        do begin mezcla[sig-llave] := llave2[n2];
                n2 := n2 + 1;
```

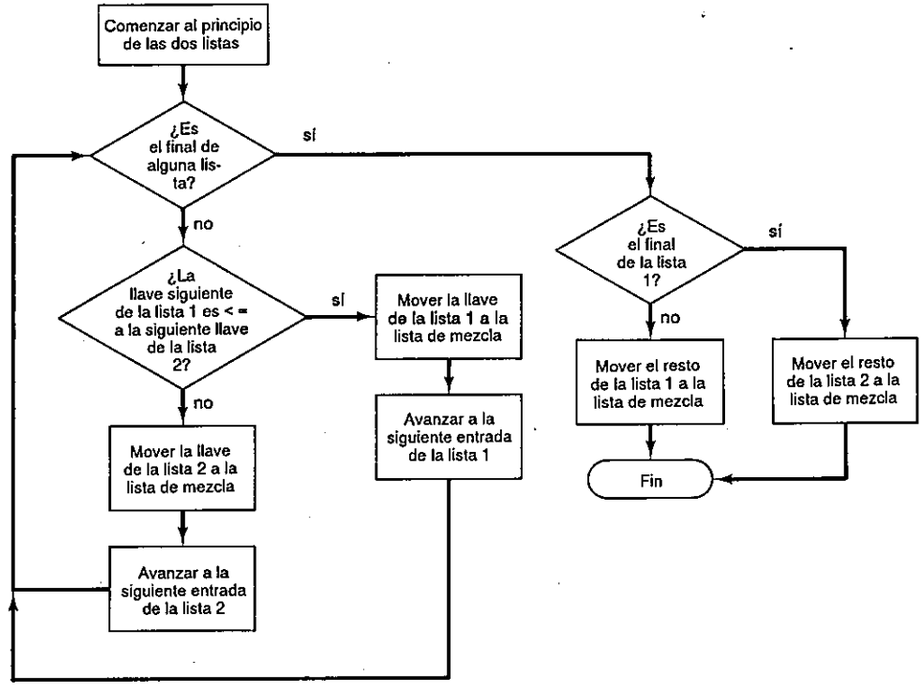


Figura 9-19 Algoritmo para una mezcla en dos sentidos.

```
                sig-llave := sig-llave + 1
            end;
        end;
```

El ejemplo fue arreglado de tal forma que resultaran 2 listas de llaves ordenadas convenientemente. En realidad, pueden resultar muchas listas ordenadas, las cuales se pueden mezclar por medio de una secuencia de mezclas en dos sentidos, o por mezclas de mayor orden, digamos tres o cuatro listas al mismo tiempo.

Desempeño

¿Qué tan bueno es el desempeño del método de ordenamiento por torneo? Considere primero, el caso en el cual el árbol de torneo es lo suficientemente espacioso como para contener todas las n llaves a ordenar. Para identificar al primer ganador, se requiere hacer

$$\frac{n}{2} + \frac{n}{4} + \ldots + \frac{n}{2^k} = n\sum_{i=1}^{k} \frac{1}{2^i}$$

comparaciones, donde k es número de niveles en el árbol de torneo, $k = \log_2 n$. Esta expresión es del orden de $O(n)$. Después de organizar de esta forma al árbol, se requiere de k comparaciones para ajustar al árbol, e identificar al siguiente ganador. Así, el número de comparaciones requeridas para procesar la totalidad del árbol es $O(n\log_2 n)$. Note que solamente un lista resulta cuando el conjunto completo de jugadores cabe en el árbol de torneo.

Considere ahora el caso en el cual el árbol de torneo puede contener solamente p llaves en un momento dado, $p << n$. Cada paso a través del árbol requiere ahora $\log_2 p$ comparaciones para identificar al siguiente ganador. Luego, $n\log_2 p$ comparaciones se necesitarán para generar las sublistas ordenadas.

Estas sublistas posteriormente deberán ser mezcladas. Puede demostrarse que la longitud esperada de cada sublista ordenada, resultado del método de ordenamiento por torneo con reemplazo de entradas, es de $2*p$. (El lector interesado, puede ver la explicación de este hecho en la bibliografía de Knuth *The Art of Computer Programming, Vol. 3, Sorting and Searching*, de Addison-Wesley, 1973). Las n llaves, de esta forma son asignadas en $n/2p$ listas. El uso del método de mezcla en dos sentidos requiere $\log_2(n/2p)$ pasos. Por ejemplo si $n = 256$ y $p = 8$, entonces el número esperado de sublistas es 16. Mezclar 16 listas, 2 a la vez, requiere $\log_2 16$, es decir, 4 pasos (Figura 9-20). Cada paso requiere n comparaciones. Así

$$n\log_2(n/2p)$$

comparaciones serán necesarias para la fase de mezclado. En conclusión, en promedio, se necesitan

$$n\log_2 p + n\log_2(n/2p)$$

comparaciones, lo cual resulta en $O(n\log_2 n)$ para $n >> p$. Volveremos a utilizar el método de ordenamiento por torneo para n's muy grandes y p's relativamente pequeñas, en el capítulo de técnicas para ordenamiento de archivos.

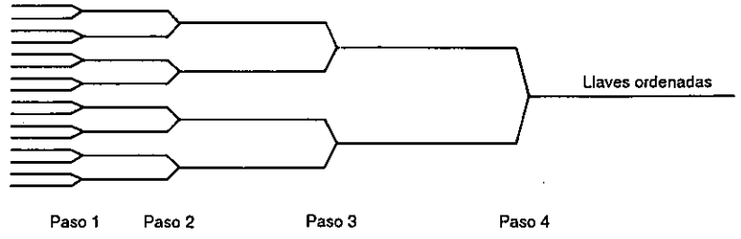


Figura 9-20 Ejemplo de los pasos para mezclar 16 sublistas.

RESUMEN

Este capítulo empezó con una discusión sobre técnicas de búsqueda lineal. Mostramos que una búsqueda secuencial puede ser mejorada ordenando las llaves en la lista, tomando en cuenta la frecuencia decreciente de acceso a esas llaves. Una lista lineal puede auto-reorganizarse dinámicamente: 1) moviendo la llave al frente de la lista cuando sea requerido o 2) intercambiando la llave con su vecino frontal cuando sea requerido, así, gradualmente se irá desplazando hacia el frente. El número de comparaciones requeridas para búsquedas sin éxito se puede reducir teniendo ordenada la lista según el valor de su llave. Todas las búsquedas secuenciales son de $O(n)$.

Después, la técnica de *búsqueda binaria* fue discutida como un método para obtener una eficiencia de búsqueda de $O(\log_2 n)$, en una lista lineal. El prerrequisito para aplicar la técnica de búsqueda binaria es que los registros en la lista deben estar ordenados secuencialmente, en base al valor de la llave. La búsqueda procede a través de pruebas sucesivas en la lista; cada prueba elimina una mitad de la lista de llaves para no considerarlas posteriormente. La técnica de búsqueda binaria fue presentada tanto en versión recursiva como iterativa.

El resto del capítulo se enfocó a las técnicas de ordenamiento de listas de registros. El ordenamiento se utiliza normalmente para acomodar en un orden específico un conjunto de llaves y después buscar en ese conjunto.

Primero, fueron introducidos algunos métodos de ordenamiento interno. Todos requieren un número de comparaciones aproximadamente proporcional al cuadrado del número de llaves en la lista; esto es, todas requieren $O(n^2)$ comparaciones. El método de ordenamiento por inserción directa, selecciona repetidamente la menor llave de la lista no ordenada como la próxima llave de la lista. La clasificación por selección e intercambio ordena la lista en su mismo lugar. El ordenamiento por inserción directa selecciona al siguiente valor del resto de llaves en la lista no ordenada y lo inserta en la posición relativa apropiada de la lista de llaves ordenadas. El método de la burbuja (método por intercambio) compara repetidamente un par de llaves vecinas y las intercambia de lugar, si éstas no están en su posición relativa adecuada.

El *método de ordenamiento por partición e intercambio,* también conocido como el *método quicksort de Hoare,* fue presentado como un método para obtener un ordenamiento de llaves más eficiente. Durante cada paso de comparación del quicksort, las llaves son intercambiadas en tal forma que, cuando los pasos son completados, la lista ha sido particionada y las dos sublistas subsecuentemente pueden ser tratadas independientemente. El método quicksort requiere en promedio $O(n\log_2 n)$ comparaciones, pero en el peor de los casos requiere $O(n^2)$ comparaciones.

El método de *ordenamiento por apilamiento* o *heapsort* mejora la eficiencia con respecto al método quicksort; garantiza $O(n\log_2 n)$ comparaciones, aun en el peor caso. El heapsort tiene dos fases. En la primera fase, las llaves a comparar son estructuradas en un apilamiento, el cual es un caso especial de árbol binario. En la segunda fase, el apilamiento se procesa y linealiza de tal forma que resulta una lista de llaves ordenada.

El método de *ordenamiento por torneo* (conocido también como *ordenamiento por selección de árbol*), y su variante el *ordenamiento por selección y reemplazo de árbol*, fueron presentados como técnicas importantes para ordenar grandes cantidades de datos. Las llaves son introducidas dentro de un árbol de torneo y son comparadas por pares, de tal forma que generan una cadena de llaves ordenadas. Si el árbol no es lo suficientemente grande para contener el conjunto de llaves, entonces el método genera varias listas de llaves ordenadas. Estas listas se deben mezclar después para producir finalmente una lista ordenada. Con reemplazo y selección, el ordenamiento de p llaves genera una cadena ordenada con longitud esperada de $2*p$. Esta propiedad hace que el método de ordenamiento por torneo sea un método popular para la fase interna del ordenamiento de archivos externos. El método de ordenamiento por torneo, al igual que los otros métodos con árboles, requiere de $O(n\log_2 n)$ comparaciones en promedio.

La búsqueda y ordenamiento son actividades sumamente importantes en el procesamiento de datos. Las técnicas usadas pueden constituir el factor determinante para el desempeño de un sistema de información.

TERMINOLOGIA

apilamiento (heap)
argumento de búsqueda
búsqueda
búsqueda binaria
búsqueda lineal
búsqueda secuencial
llave
$O(n)$
$O(n^2)$
$O(n\log_2 n)$
ordenamiento
ordenamiento de la burbuja
ordenamiento estable
ordenamiento externo
ordenamiento interno
ordenamiento por inserción
ordenamiento por intercambio
ordenamiento por partición e intercambio
ordenamiento por selección
ordenamiento por selección de árbol
ordenamiento por selección e intercambio
ordenamiento por torneo
quicksort

REFERENCIAS SUGERIDAS

ALANKO, T. O., H. H. A. ERKIO, and I. J. HAIKALA. "Virtual memory behavior of some sorting algorithms," *IEEE Trans. on Software Engineering*, SE-10(4):422–431, July 1984.

BITTON, D., D. J. DEWITT, D. K. HSAIO, and J. MENON. "A taxonomy of parallel sorting," *ACM Computing Surveys*, 16(3):297–318, Sept. 1984.

BLASGEN, M. W., R. G. CASEY, and K. P. ESWAREN. "An encoding method for multifield sorting and indexing," *Comm. ACM*, 20(11):874–878, Nov. 1977.

CASEY, R. G. "Design of tree structures for efficient querying," *Comm. ACM*, 16(9):549–560, Sept. 1973.

COOK, C. R. and D. J. KIM. "Best sorting algorithm for nearly sorted lists," *Comm. ACM*, 23(11):620–624, Nov. 1980.

GILL, A. "Hierarchical binary search," *Comm. ACM*, 23(5): 294–300, May 1980.

HERSTER, J. H. and D. S. HIRSCHBERG. "Self-organizing linear search," *ACM Computing Surveys*, 17(3):295–311, Sept. 1985.

HIRSCHBERG, D. S. "Fast parallel sorting algorithms," *Comm. ACM*, 21(8):657–661, Aug. 1978.

HORVATH, E. C. "Stable sorting in asymptotically optimal time and extra space," *Jour. ACM*, 25(2):177–199, April 1978.

KNUTH, D. E. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Reading, MA: Addison-Wesley Publishing Co., 1973.

LOESER, R. "Some performance tests of 'quicksort' and descendants," *Comm. ACM*, 17(3):143–152, March 1974.

LORIN, H. *Sorting and Sort Systems*. Reading, MA: Addison-Wesley Publishing Co., 1975.

MANACHER, G. K. "The Ford-Johnson sorting algorithm is not optimal," *Jour. ACM*, 26(3):441–456, July 1979.

PERL Y., A. ITAI, and H. AVNI. "Interpolation search: a log log n search," *Comm. ACM*, 21(7):550–553, July 1978.

SCHNEIDERMAN, B. "Jump searching: a fast sequential search technique," *Comm. ACM*, 21(10):831–834, Oct. 1978.

SEDGEWICK, R. "Implementing quicksort programs," *Comm. ACM*, 21(10):847–857, Oct. 1978.

SHIH, Z-C., G-H. CHEN, and R. C. T. LEE. "Systolic algorithms to examine all pairs of elements," *Comm. ACM*, 30(2):161–167, Feb. 1987.

STASKO, J. T. and J. S. VITTER. "Pairing heaps: experiments and analysis," *Comm. ACM*, 30(3):234–249, March 1987.

TENENBAUM, A. "Simulations of dynamic sequential search algorithms," *Comm. ACM*, 21(9):790–791, Sept. 1978.

THOMPSON, C. D. and H. T. KUNG. "Sorting on a mesh-connected parallel computer," *Comm. ACM*, 20(4):263–271, April 1977.

YAO, A. C-C. "Should tables be sorted?" *Jour. ACM*, 28(3):615–628, July 1981.

EJERCICIOS DE REPASO

1. Muestre con un ejemplo la diferencia entre un método de ordenamiento estable y otro inestable.
2. Considere las siguientes probabilidades como argumentos de búsqueda.

llave(i)	prob(i)
8	.05
2	.26
10	.21
4	.15
12	.32

   a) ¿Cuál es la probabilidad de uña búsqueda sin éxito?
   b) Si las llaves quedan ordenadas como se muestra arriba, ¿Cuál es el número esperado de comparaciones para una búsqueda secuencial?
   c) Reordene las llaves para minimizar el número esperado de comparaciones para una búsqueda secuencial.
   d) ¿Cuál es el número esperado de comparaciones para una búsqueda secuencial, según haya contestado el inciso c)?
3. Describa dos técnicas para reorganizar dinámicamente una lista lineal que mejore el tiempo esperado de búsqueda.
4. Compare el uso del método de "movimiento al frente" sobre una lista lineal alojada en un arreglo y alojada en una lista ligada.
5. Compare el método de "transposición" sobre una lista lineal alojada en un arreglo y alojada en una lista ligada.
6. ¿Por qué es deseable realizar un ordenamiento en el mismo espacio?
7. Usando una baraja de naipes, u otro tipo de cartas con valores de llave, demuestre la diferencia entre un método por inserción, un método por selección y un método por intercambio, para ordenar las cartas.
8. Escriba un programa para implantar el método de ordenamiento de la burbuja.
9. Escriba un algoritmo para buscar secuencialmente todos los registros con un valor de llave en particular en una lista ligada.
10. El texto discute una técnica de búsqueda binaria para aplicarla a listas lineales ordenadas. Desarrolle y analice una técnica de búsqueda ternaria.
11. Los algoritmos del texto para búsqueda binaria asumieron que la lista lineal estuvo alojada en un arreglo con subíndices, en el rango de 1 a n. Modifique los algoritmos para que los subíndices estén en un rango de a hasta b, donde a y b no necesitan ser enteros positivos.

12. Los algoritmos del texto para búsqueda binaria asumieron que la lista lineal estuvo alojada en un arreglo. ¿Cuáles son las ventajas y desventajas relativas de aplicar la técnica de búsqueda binaria a una lista lineal, alojada en una lista ligada?
13. Explique las mejoras de procedimiento que distinguen a los métodos de ordenamiento que requieren $O(n\log_2 n)$ operaciones, con respecto a aquellos, que requieren $O(n^2)$ operaciones.
14. Establezca una gráfica de n^2 y $n \log_2 n$ (o de n y $\log_2 n$) contra n y convénzase usted mismo de la mejora introducida por los métodos de ordenamiento no lineales.
15. Reescriba los algoritmos dados en este capítulo para ordenar las llaves en orden descendente en lugar de orden ascendente.
16. Escriba un programa para ejecutar un ordenamiento por selección de árbol.
17. Escriba un programa para ejecutar un ordenamiento por selección y reemplazo de árbol.
18. Escriba un programa para ejecutar el método quicksort. Ejecútelo con una variedad de conjuntos de entrada de datos, y compare el desempeño esperado contra el del peor caso.
19. Escriba un programa para ejecutar el método de ordenamiento por apilamiento. Ejecútelo con el mismo conjunto de datos de entrada, utilizado en el ejercicio anterior y compare el desempeño esperado contra el del peor caso.
20. Desarrolle un algoritmo de ordenamiento que garantice un desempeño de $O(n\log_3 n)$. En base a éste desarrolle otro que garantice un desempeño de $O(n\log_x n)$.
21. ¿Existen algoritmos de ordenamiento que produzcan un desempeño de $O(\log_2 n^2)$?
22. Desarrolle las pautas para mostrar los efectos que varios valores relativos de n y p producen en el desempeño de un ordenamiento por selección y reemplazo de árboles. ¿Cómo puede ser seleccionado el valor de p? ¿Qué sucede si $p = n$? ¿Qué sucede si p está cerca de n? ¿Qué sucede si $p = 1$?
23. Considere una lista ligada bidireccional de elementos de datos, la cual se ha implantado utilizando apuntadores. ¿Cuando deberá usar una técnica de búsqueda secuencial para encontrar un elemento en particular?
24. Analice la estabilidad de cada uno de los algoritmos de ordenamiento presentados en este capítulo. Si algún algoritmo de ordenamiento es inestable, ¿Cómo puede modificarse para hacerlo estable?
25. Explique cómo puede seleccionar el tipo de método de ordenamiento que deba usarse con un conjunto particular de registros. ¿Qué factores deben ser considerados?
26. Explique qué tipo de búsqueda debe usar para encontrar un elemento en un conjunto particular de registros. ¿Qué factores deben ser considerados?
27. Lea el excelente trabajo de Knuth sobre búsqueda y ordenamiento.