6.10. BÚSQUEDA EN LISTAS: BÚSQUEDAS SECUENCIAL Y BINARIA

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arrays y registros, y por ello será necesario determinar si un array contiene un valor que coincida con un cierto *valor clave*. El proceso de encontrar un elemento específico de un array se denomina *búsqueda*. En esta sección se examinarán dos técnicas de búsqueda: *búsqueda lineal* o *secuencial*, la técnica más sencilla, y *búsqueda binaria* o *dicotómica*, la técnica más eficiente.

6.10.1. Búsqueda secuencial

La **búsqueda secuencial** busca un elemento de una lista utilizando un valor destino llamado ***clave.*** En una búsqueda secuencial (a veces llamada ***búsqueda lineal***), los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro. La búsqueda secuencial es necesaria, por ejemplo, si se desea encontrar la persona cuyo número de teléfono es 958-220000 en un directorio o listado telefónico de su ciudad. Los directorios de teléfonos están organizados alfabéticamente por el nombre del abonado en lugar de por números de teléfono, de modo que deben explorarse todos los números, uno después de otro, esperando encontrar el número 958-220000.

El algoritmo de búsqueda secuencial compara cada elemento del array con la *clave* de búsqueda. Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro. De promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. El método de búsqueda lineal funcionará bien con arrays pequeños o no ordenados. La eficiencia de la búsqueda secuencial es pobre, tiene complejidad lineal, $O(n)$.

6.10.2. Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Una búsqueda binaria típica es la búsqueda de una palabra en un diccionario. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra del primer apellido o de la palabra que busca. Se puede tener suerte y acertar con la página correcta; pero, normalmente, no será así y se mueve el lector a la página anterior o posterior del libro. Por ejemplo, si la palabra comienza con «J» y se está en la «L» se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la lista.

Una idea similar se aplica en la búsqueda en una lista ordenada. Se sitúa la lectura en el centro de la lista y se comprueba si la clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sigue la búsqueda uno en la mitad inferior o superior del elemento central de la lista. En general, si los datos de la lista están ordenados se puede utilizar esa información para acortar el tiempo de búsqueda.

Ejemplo 6.4

Se desea buscar el elemento 225 y ver si se encuentra en el conjunto de datos siguiente:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
13	44	75	100	120	275	325	510

El punto central de la lista es el elemento `a[3] (100)`. El valor que se busca es `225`, mayor que `100`; por consiguiente, la búsqueda continúa en la mitad superior del conjunto de datos de la lista, es decir, en la sublista,

```
a[4]     a[5]     a[6]     a[7]
120       275      325      510
```

Ahora el elemento mitad de esta sublista `a[5] (275)`. El valor buscado, `225`, es menor que `275` y, por consiguiente, la búsqueda continúa en la mitad inferior del conjunto de datos de la lista actual; es decir, en la sublista de un único elemento:

```
a[4]
120
```

El elemento mitad de esta sublista es el propio elemento `a[4] (120)`. Al ser `225` mayor que `120`, la búsqueda debe continuar en una sublista vacía. Se concluye indicando que no se ha encontrado la clave en la lista.

6.10.3. Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista está almacenada como un array, los índices de la lista son: `bajo = 0` y `alto = n-1` y `n` es el número de elementos del array, los pasos a seguir:

1. Calcular el índice del punto central del array

   `central = (bajo + alto)/2` (división entera)

2. Comparar el valor de este elemento central con la clave:

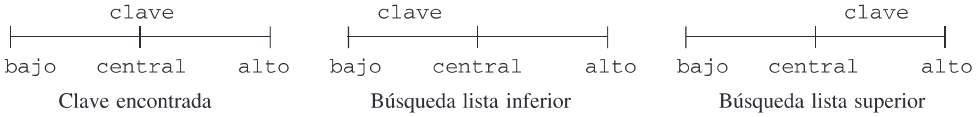


Figura 6.5. Búsqueda binaria de un elemento.

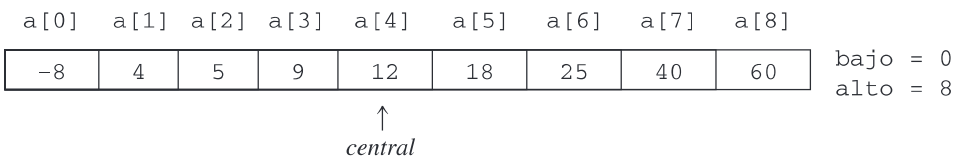
- Si `a[central] < clave`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo = central+1..alto`.
- Si `clave < a[central]`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo..central-1`.

El algoritmo se termina bien porque se ha encontrado la clave o porque el valor de `bajo` excede a `alto` y el algoritmo devuelve el indicador de fallo de −1 (*búsqueda no encontrada*).

Ejemplo 6.5

Sea el array de enteros `A (-8, 4, 5, 9, 12, 18, 25, 40, 60)`, buscar la clave, clave = 40.

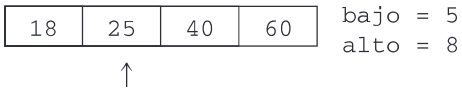
1.



$$central = \frac{bajo + alto}{2} = \frac{0+8}{2} = 4$$

`clave (40) > a[4] (12)`

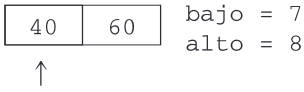
2. Buscar en sublista derecha



$$central = \frac{bajo + alto}{2} = \frac{5+8}{2} = 6 \quad (división\ entera)$$

`clave (40) > a[6] (25)`

3. Buscar en sublista derecha



$$central = \frac{bajo + alto}{2} = \frac{7+8}{2} = 7$$

`clave (40) = a[7] (40)` (*búsqueda con éxito*)

4. El algoritmo ha requerido 3 comparaciones frente a 8 comparaciones ($n - 1$, $9 - 1 = 8$) que se hubieran realizado con la búsqueda secuencial.

La codificación del algoritmo de búsqueda binaria:

```
/*
  búsqueda binaria.
  devuelve el índice del elemento buscado, o bien -1 caso de fallo
*/

int busquedaBin(int lista[], int n, int clave)
{
  int central, bajo, alto;
  int valorCentral;

  bajo = 0;
  alto = n-1;
  while (bajo <= alto)
```

```
  {
    central = (bajo + alto)/2;       /* índice de elemento central */
    valorCentral = lista[central];   /* valor del índice central */
    if (clave == valorCentral)
      return central;                /* encontrado, devuelve posición */
    else if (clave < valorCentral)
      alto = central -1;             /* ir a sublista inferior */
    else
      bajo = central + 1;            /* ir a sublista superior */
  }
   return -1;                        /* elemento no encontrado */
}
```

6.10.4. Análisis de los algoritmos de búsqueda

Al igual que sucede con las operaciones de ordenación cuando se realizan operaciones de búsqueda es preciso considerar la eficiencia (complejidad) de los algoritmos empleados en la búsqueda. El grado de eficiencia en una búsqueda será vital cuando se trata de localizar una información en una lista o tabla en memoria, o bien en un archivo de datos.

Complejidad de la búsqueda secuencial

La complejidad de la búsqueda secuencial diferencia entre el comportamiento en el caso peor y mejor. El mejor caso se encuentra cuando aparece una coincidencia en el primer elemento de la lista y en ese caso el tiempo de ejecución es $O(1)$. El caso peor se produce cuando el elemento no está en la lista o se encuentra al final de la lista. Esto requiere buscar en todos los n términos, lo que implica una complejidad de $O(n)$.

El caso medio requiere un poco de razonamiento probabilista. Para el caso de una lista aleatoria es probable que una coincidencia ocurra en cualquier posición. Después de la ejecución de un número grande de búsquedas, la posición media para una coincidencia es el elemento central $n/2$. El elemento central ocurre después de $n/2$ comparaciones, que define el coste esperado de la búsqueda. Por esta razón, se dice que la prestación media de la búsqueda secuencial es $O(n)$.

Análisis de la búsqueda binaria

El caso mejor se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso la complejidad es $O(1)$ dado que sólo se realiza una prueba de comparación de igualdad. La complejidad del caso peor es $O(\log_2 n)$ que se produce cuando el elemento no está en la lista o el elemento se encuentra en la última comparación. Se puede deducir intuitivamente esta complejidad. El caso peor se produce cuando se debe continuar la búsqueda y llegar a una sublista de longitud de 1. Cada iteración que falla debe continuar disminuyendo la longitud de la sublista por un factor de 2. El tamaño de las sublistas es:

$$n \quad n/2 \quad n/4 \quad n/8 \quad \ldots \quad 1$$

La división de sublistas requiere m iteraciones, en cada iteración el tamaño de la sublista se reduce a la mitad. La sucesión de tamaños de las sublistas hasta una sublista de longitud 1:

$$n \quad n/2 \quad n/2 \quad n/2^3 \quad n/2^4 \quad \ldots \quad n/2^m$$

siendo $n/2^m = 1$.

Tomando logaritmos en base 2 en la expresión anterior quedará:

$n = 2^m$
$m = \log_2 n$

Por esa razón la complejidad del caso peor es $O(\log_2 n)$. Cada iteración requiere una operación de comparación:

$$Total\ comparaciones \approx 1 + \log_2 n$$

Comparación de la búsqueda binaria y secuencial

La comparación en tiempo entre los algoritmos de búsqueda secuencial y binaria se va haciendo espectacular a medida que crece el tamaño de la lista de elementos. Tengamos presente que en el caso de la búsqueda secuencial, en el peor de los casos, coincidirá el número de elementos examinados con el número de elementos de la lista tal como representa su complejidad $O(n)$.

Sin embargo, en el caso de la búsqueda binaria, tengamos presente, por ejemplo, que $2^{10} = 1.024$, lo cual implica el examen de 11 posibles elementos; si se aumenta el número de elementos de una lista a 2.048 y teniendo presente que $2^{11} = 2.048$ implicará que el número máximo de elementos examinados en la búsqueda binaria es 12. Si se sigue este planteamiento, se puede encontrar el número m más pequeño para una lista de 1.000.000, tal que

$2^n \geq 1.000.000$

Es decir, $2^{19} = 524.288$, $2^{20} = 1.048.576$ y por tanto el número de elementos examinados (en el peor de los casos) es 21.

La Tabla 6.2 muestra la comparación de los métodos de búsqueda secuencial y búsqueda binaria. En la misma tabla se puede apreciar una comparación del número de elementos que se deben examinar utilizando búsquedas secuencial y binaria. Esta tabla muestra la eficiencia de la búsqueda binaria comparada con la búsqueda secuencial y cuyos resultados de tiempo vienen dados por las funciones de complejidad $O(\log_2 n)$ y $O(n)$ de las búsquedas binaria y secuencial respectivamente.

Tabla 6.2. Comparación de las búsquedas binaria y secuencial

	Números de elementos examinados	
Tamaño de la lista	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000

A tener en cuenta

La búsqueda secuencial se aplica para localizar una clave en un vector no ordenado. Para aplicar el algoritmo de búsqueda binaria la lista, o vector, donde se busca debe de estar ordenado.

La *complejidad* de la *búsqueda binaria* es *logarítmica*, **O(log n)**, más eficiente que la *búsqueda secuencial* que tiene *complejidad lineal*, **O(n)**.

RESUMEN

- Una de las aplicaciones más frecuentes en programación es la ordenación.
- Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos.*
- Los datos se pueden ordenar en orden ascendente o en orden descendente.
- Cada recorrido de los datos durante el proceso de ordenación se conoce como *pasada* o *iteración.*
- Los algoritmos de ordenación básicos son:

  — Selección.
  — Inserción.
  — Burbuja.

- Los algoritmos de ordenación más avanzados son:

  — Shell.
  — Mergesort.
  — Radixsort.
  — Binsort.
  — Quicksort.

- La eficiencia de los algoritmos de burbuja, inserción y selección es $O(n^2)$.
- La eficiencia de los algoritmos, *radixsort*, *mergesort* y *quicksort* es $O(n \log n)$.
- La búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista
- Existen dos métodos básicos de búsqueda en arrays: **secuencial** y **binaria.**
- La **búsqueda secuencial** se utiliza normalmente cuando el array no está ordenado. Comienza en el principio del array y busca hasta que se encuentra el dato buscado y se llega al final de la lista.
- Si un array está ordenado, se puede utilizar un algoritmo más eficiente denominado **búsqueda binaria.**
- La eficiencia de una búsqueda secuencial es $\boldsymbol{O(n)}$.
- La eficiencia de una búsqueda binaria es $\boldsymbol{O(\log n)}$.

EJERCICIOS

6.1. ¿Cuál es la diferencia entre ordenación por intercambio y ordenación por el método de la burbuja?

6.2. Se desea eliminar todos los números duplicados de una lista o vector (array). Por ejemplo, si el array toma los valores

4 7 11 4 9 5 11 7 3 5

ha de cambiarse a

4 7 11 9 5 3

Escribir una función que elimine los elementos duplicados de un array.

6.3. Escribir una función que elimine los elementos duplicados de un vector ordenado. ¿Cuál es la eficiencia de esta función? Comparar la eficiencia con la correspondiente a la función del Ejercicio 6.2.

6.4. Un vector contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción. ¿Cuál será el valor de los elementos del vector después de tres pasadas más del algoritmo?

3	13	8	25	45	23	98	58