Renteria Magaña Rayni Damian
Algoritmos de busqueda

Se concluye que la operación de búsqueda se puede llevar a cabo sobre elementos ordenados o desordenados. Cabe destacar que la búsqueda es más fácil y ocupa menos tiempo cuando los datos se encuentran ordenados.

Los métodos de búsqueda se pueden clasificar en **internos** y **externos**, según la ubicación de los datos sobre los cuales se realizará la búsqueda. Se denomina **búsqueda interna** cuando todos los elementos se encuentran en la memoria principal de la computadora; por ejemplo, almacenados en arreglos, listas ligadas o árboles. Es **búsqueda externa** si los elementos están en memoria secundaria; es decir, si hubiera archivos en dispositivos como cintas y discos magnéticos.

La **búsqueda interna** trabaja con elementos que se encuentran almacenados en la memoria principal de la máquina. Éstos pueden estar en estructuras estáticas —arreglos— o dinámicas —listas ligadas y árboles—. Los métodos de búsqueda interna más importantes son:

- Secuencial o lineal
- Binaria
- Por transformación de claves
- Árboles de búsqueda

La **búsqueda secuencial** consiste en revisar elemento tras elemento hasta encontrar el dato buscado, o llegar al final del conjunto de datos disponible.

Primero se tratará sobre la búsqueda secuencial en arreglos, y luego en listas enlazadas. En el primer caso, se debe distinguir entre arreglos ordenados y desordenados.

Esta última consiste, básicamente, en recorrer el arreglo de izquierda a derecha hasta que se encuentre el elemento buscado o se termine el arreglo, lo que ocurra primero. Normalmente cuando una función de búsqueda concluye con éxito, interesa conocer en qué posición fue hallado el elemento que se estaba buscando. Esta idea se puede generalizar para todos los métodos de búsqueda.

La **búsqueda binaria** consiste en dividir el intervalo de búsqueda en dos partes, comparando el elemento buscado con el que ocupa la posición central en el arreglo. Para el caso de que no fueran iguales se redefinen los extremos del intervalo, según el elemento central sea mayor o menor que el elemento buscado, disminuyendo de esta forma el espacio de búsqueda. El proceso concluye cuando el elemento es encontrado, o cuando el intervalo de búsqueda se anula, es vacío.

El método de búsqueda binaria funciona exclusivamente con arreglos ordenados. No se puede utilizar con listas simplemente ligadas —no podríamos retroceder para establecer intervalos de búsqueda— ni con arreglos desordenados. Con cada iteración del método el espacio de búsqueda se reduce a la mitad; por lo tanto, el número de com-

paraciones a realizar disminuye notablemente. Esta disminución resulta significativa cuanto más grande sea el tamaño del arreglo. A continuación se presenta el algoritmo de búsqueda binaria.

Analicemos ahora un ejemplo para ilustrar el funcionamiento de este algoritmo.

Sea *V* un arreglo unidimensional de números enteros, ordenado de manera creciente, como se muestra en la figura 9.3.

En la tabla 9.2 se presenta el seguimiento del algoritmo 9.7 cuando *X* es igual a 325 (*X* = 325).

En la figura 9.4 se observa gráficamente, para este caso en particular, cómo se va reduciendo el intervalo de búsqueda.

V

101	215	325	410	502	507	600	610	612	670
1	2	3	4	5	6	7	8	9	10

Paso	BAN	IZQ	DER	CEN	*X* = *V*[CEN]	*X* > *V*[CEN]
1	Falso	1	10	5	325 = 502 ? No	325 > 502 ? No
2	Falso	1	4	2	325 = 215 ? No	325 > 215 ? Sí
3	Falso	3	4	3	325 = 325 ? Sí	
4	Verdadero					

La tabla 9.3, por otra parte, muestra nuevamente el seguimiento del algoritmo 9.7 para *X* = 615, valor que no se encuentra en el arreglo.

Búsqueda por transformación de claves

Los dos métodos analizados anteriormente permiten encontrar un elemento en un arreglo. En ambos casos el tiempo de búsqueda es proporcional a su número de componentes. Es decir, a mayor número de elementos se debe realizar mayor número de comparaciones. Se mencionó además que si bien el método de búsqueda binaria es más eficiente que el secuencial, existe la restricción de que el arreglo se debe encontrar ordenado.

Esta sección se dedica a un nuevo método de búsqueda. Este método, conocido como **transformación de claves** o ***hash***, permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados. Cuenta con la ventaja de que el tiempo de búsqueda es independiente del número de componentes del arreglo.

Supongamos que se tiene una colección de datos, cada uno de ellos identificado por una clave. Es claro que resulta atractivo tener acceso a ellos de manera directa; es decir, sin recorrer algunos datos antes de localizar al buscado. El método por transformación de claves permite realizar justamente esta actividad; es decir, localizar el dato en forma directa. El método trabaja utilizando una función que convierte una clave dada en una dirección —índice— dentro del arreglo.

$$\text{dirección} \leftarrow H\ (\text{clave})$$

La función *hash* (H) aplicada a la clave genera un índice del arreglo que permite acceder directamente al elemento. El caso más trivial se presenta cuando las claves son números enteros consecutivos.

Supongamos que se desea almacenar la información relacionada con 100 alumnos cuyas matrículas son números del 1 al 100. En este caso conviene definir un arreglo de 100 elementos con índices numéricos comprendidos entre los valores 1 y 100. Los datos de cada alumno ocuparán la posición del arreglo que se corresponda con el número de la matrícula; de esta manera se podrá acceder directamente a la información de cada alumno.

Consideremos ahora que se desea almacenar la información de 100 empleados. La clave de cada empleado corresponde al número de su seguro social. Si la clave está formada por 11 dígitos, resulta por completo ineficiente definir un arreglo con 99 999 999 999 elementos para almacenar solamente los datos de los 100 empleados. Utilizar un arreglo tan grande asegura la posibilidad de acceder directamente a sus elementos; sin embargo, el costo en memoria resulta tanto ridículo como excesivo. Siempre es importante equilibrar el costo del espacio de memoria con el costo por tiempo de búsqueda.

Cuando se tienen claves que no se corresponden con índices —por ejemplo, por ser alfanuméricas—, o cuando las claves representen valores numéricos muy grandes o no se corresponden con los índices de los arreglos, se utilizará una función *hash* que permita transformar la clave para obtener una dirección apropiada. Esta función *hash* debe ser simple de calcular y asignar direcciones de la manera más uniforme posible. Es decir, debe generar posiciones diferentes dadas dos claves también diferentes. Si esto último no ocurre ($H(K_1) = d$, $H(K_2) = d$ y $K_1 \neq K_2$) hay una **colisión**, que se define como la asignación de una misma dirección a dos o más claves distintas.

En este contexto, para trabajar con este método de búsqueda se debe seleccionar previamente:

- Una función *hash* que sea fácil de calcular y distribuya uniformemente las claves
- Un método para resolver colisiones. Si éstas se presentan, se contará con algún método que genere posiciones alternativas.

Estos dos casos se tratarán en forma separada. Como ya se mencionó, seleccionar una buena función *hash* es muy importante, pero es difícil encontrarla. Básicamente porque no existen reglas que permitan determinar cuál será la función más apropiada para un conjunto de claves que asegure la máxima uniformidad en su distribución. Realizar un análisis de las principales características de las claves siempre ayuda en la elección de una función de este tipo. A continuación se explican algunas de las funciones *hash* más utilizadas.

Función *hash* por módulo: división

La función ***hash* por módulo** o **división** consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo. Supongamos, por ejemplo, que se tiene un arreglo de N elementos, y K es la clave del dato a buscar. La función *hash* queda definida por la siguiente fórmula:

$$H(K) = (K \bmod N) + 1 \qquad \textbf{Fórmula 9.3}$$

En la fórmula 9.3 se observa que al residuo de la división se le suma 1, esto último con el objetivo de obtener un valor comprendido entre 1 y N.

Para lograr mayor uniformidad en la distribución, es importante que N sea un número primo o divisible entre muy pocos números. Por lo tanto, si N no es un número primo, se debe considerar el valor primo más cercano.

En el ejemplo 9.2 se presenta un caso de función *hash* por módulo.

Supongamos que $N = 100$ es el tamaño del arreglo, y las direcciones que se deben asignar a los elementos (al guardarlos o recuperarlos) son los números del 1 al 100. Consideremos además que $K_1 = 7\,259$ y $K_2 = 9\,359$ son las dos claves a las que se deben asignar posiciones en el arreglo. Si aplicamos la fórmula 9.3 con $N = 100$, para calcular las direcciones correspondientes a K_1 y K_2, obtenemos:

$$H(K_1) = (7\,259 \bmod 100) + 1 = 60$$
$$H(K_2) = (9\,359 \bmod 100) + 1 = 60$$

Como $H(K_1)$ es igual a $H(K_2)$ y K_1 es distinto de K_2, se está ante una colisión que se debe resolver porque a los dos elementos le correspondería la misma dirección.

Observemos, sin embargo, que si aplicáramos la fórmula 9.3 con un número primo cercano a N, el resultado cambiaría:

$$H(K_1) = (7\,259 \bmod 97) + 1 = 82$$
$$H(K_2) = (9\,359 \bmod 97) + 1 = 48$$

Con $N = 97$ se ha eliminado la colisión.