



Universidad Nacional Autónoma de México
Facultad de Ingeniería
Departamento de Computación
Estructuras de datos y algoritmos II

TEMA 2

ALGORITMOS DE BÚSQUEDA

2 Algoritmos de búsqueda

Objetivo: Conocer algoritmos de búsqueda y aplicar el apropiado a conjuntos de datos residentes en la memoria principal para generar algoritmos que resuelvan búsquedas.

2 Algoritmos de búsqueda

2.1 Generalidades.

2.2 Definición de la operación de búsqueda.

2.3 Búsqueda por comparación de llaves.

2.3.1 Lineal.

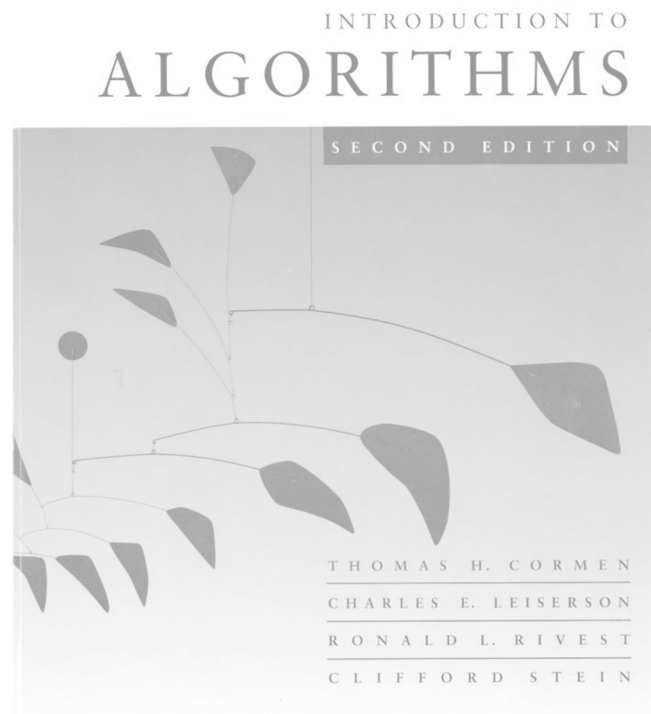
2.3.2 Binaria.

2.4 Búsqueda por transformación de llaves.

2.4.1 Funciones de hash.

2.4.2 Colisiones.

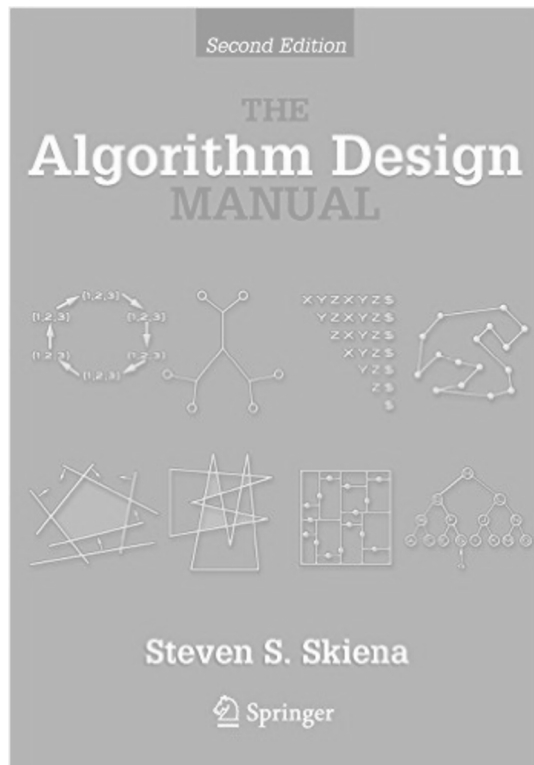
Bibliografía



Introduction to Algorithms.
Thomas H. Cormen, Charles E.
Leiserson, Ronald L. Rivest, Clifford
Stein, McGraw-Hill.

Bibliografía

The Algorithm Design Manual.
Steven S. Skiena, Springer.



“There is no programming language – no matter how structured – that will prevent programmers from making bad programs.”

Larry Flon
(He worked at USC's Information Sciences Institute)



2.1 Generalidades

2. ALGORITMOS DE BÚSQUEDA

2.1 Generalidades

En el procesamiento de información se poseen diversas estructuras de datos sobre las cuales se pueden efectuar varias operaciones. Las operaciones más frecuentes en la estructuras de datos son agregar, eliminar y consultar.

La operación de consulta, a su vez, puede ser de actualización (modificar un elemento), de lectura (obtener los datos de un elemento) o de verificación (comprobar la existencia de un elemento).

Las operaciones de actualización, lectura o verificación requieren de una operación común denominada búsqueda.

La operación de búsqueda se puede llevar a cabo sobre cualquier conjunto de datos lineal, sin importar si estos se encuentran ordenados o no.

2.2 Definición de la operación de búsqueda

2. ALGORITMOS DE BÚSQUEDA



2.2 Definición de la operación de búsqueda

Los métodos de búsqueda se pueden clasificar en internos (sobre datos que se encuentran en memoria principal) y externos (sobre datos que se encuentran en memoria secundaria).

Los principales métodos de búsqueda son:

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

2.3 Búsqueda por comparación de llaves

2. ALGORITMOS DE BÚSQUEDA



2.3 Búsqueda por comparación de llaves

Los métodos de búsqueda que se analizarán en este tema se basan en la búsqueda directa de una llave a través de los nodos de la estructura de datos.

Las búsquedas que se basan en la comparación de llaves son la búsqueda lineal y la búsqueda binaria.

2.3.1 Búsqueda lineal

La búsqueda lineal, también llamada secuencial, es el método de búsqueda más sencillo, consiste en revisar elemento tras elemento hasta encontrar el dato buscado o llegar al final del conjunto de datos.

Este método se puede aplicar a estructuras de datos tanto ordenadas como desordenadas. Se suele utilizar cuando el conjunto de datos es pequeño.

Si los elementos del conjunto residen en memoria secundaria, hay que considerar el tiempo de acceso al dispositivo, además del tiempo de recorrido.

Sea A un conjunto de elementos contenidos en una estructura de datos lineal:

A 6 7 50 5 0 4 23 22 11 10 9 17

A	6	7	50	5	0	4	23	22	11	10	9	17
---	---	---	----	---	---	---	----	----	----	----	---	----

Se requieren 7 comparaciones para conocer si el número 23 se encuentra en el conjunto de datos. Este caso representa el caso promedio de búsqueda.

A 6 7 50 5 0 4 23 22 11 10 9 17

El número promedio de comparaciones para buscar un elemento en la estructura está dado por el número de elementos entre 2, es decir, $n/2$. Para este conjunto el número promedio de comparaciones es 6.

A 6 7 50 5 0 4 23 22 11 10 9 17

Se requieren un total de doce comparaciones para darse cuenta de que un elemento no está en la estructura. Este es el peor caso de complejidad para la búsqueda lineal.

Algoritmo de búsqueda lineal

Generar un algoritmo iterativo de búsqueda lineal que, a partir de un conjunto de datos lineal y un valor dado, permita encontrar la primera coincidencia de la llave dentro del conjunto.

Algoritmo de búsqueda lineal

Generar un algoritmo recursivo de búsqueda lineal que, a partir de un conjunto de datos lineal y un valor dado, permita encontrar la primera coincidencia de la llave dentro del conjunto.

Un algoritmo que permite realizar una búsqueda lineal iterativa para encontrar la primera coincidencia de un dato dado es:

```
FUNC linear_search_iterative (list[: Entero, value: Entero) dev ENTERO
    pos ← 0: ENTERO
    MIENTRAS pos < length(list) Y list[pos] <> value HACER
        pos ← pos + 1
    FIN_MIENTRAS

    SI pos == length(list) ENTONCES
        DEV -1
    FIN_SI
    EN_CASO_CONTRARIO
        DEV i
    FIN_ECC
FIN_FUNC
```

El algoritmo del método de búsqueda lineal permite encontrar la primera ocurrencia de un número, pero no las subsecuentes, es decir, si la clave 'value' se encuentra repetida en el conjunto, solo se devuelve la posición de la primera ocurrencia. El algoritmo se puede modificar para encontrar elementos repetidos pero el tiempo para el mejor y el caso promedio se vuelve lineal $O(n)$.

El algoritmo anterior se puede transformar en recursivo, para ello hay que considerar tres casos: cuando el contador i es mayor al tamaño del conjunto n , cuando el conjunto en la i ésima posición $lista[i]$ es igual a 'x' y el caso recursivo.

Un algoritmo que permite realizar una búsqueda lineal recursiva para encontrar la primera coincidencia de un dato dado es:

```
FUNC linear_search_recursive (list[: Entero, value: Entero, cont: Entero)
    dev ENTERO

    SI cont == length(list) ENTONCES
        dev -1
    FIN SI
    SI list[cont] == value ENTONCES
        dev cont

    cont ← cont + 1
    dev linear_search_recursive (list, value, cont)
FIN_FUNC
```

El análisis de complejidad del método de búsqueda lineal está dado por el número total de comparaciones.

El peor caso de complejidad se presenta cuando el elemento buscado (x) es el último o no se encuentra en el conjunto, en tal caso se realizarán n comparaciones ($C_{\max} = n$). El mejor caso se presenta cuando el elemento buscado (x) se encuentra en la primera posición y se realiza 1 comparación ($C_{\min} = 1$). En el caso promedio se realizan $n/2$ comparaciones ($C_{\text{med}} = n/2$).

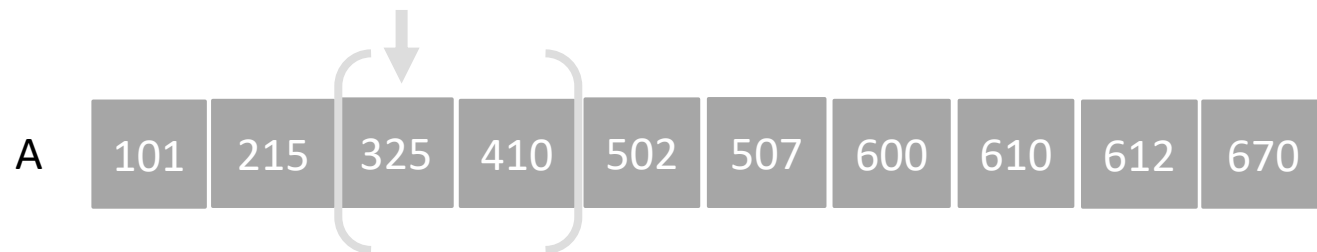
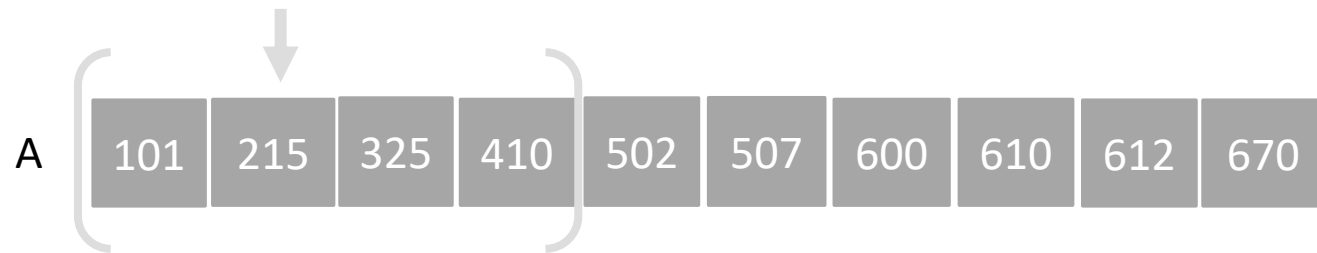
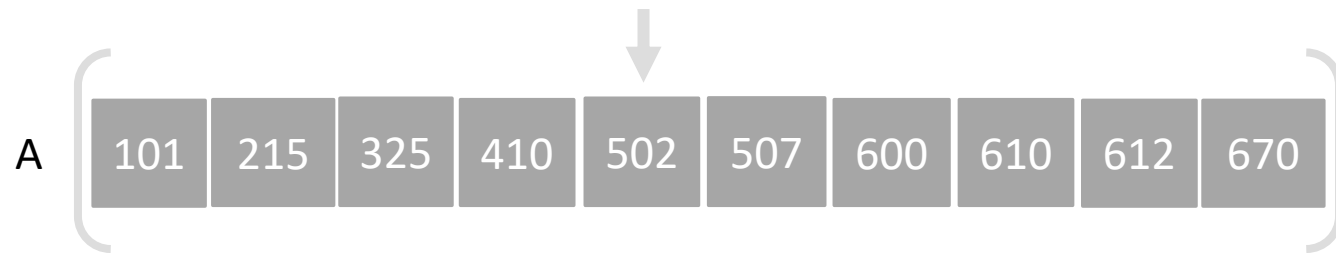
2.3.2 Búsqueda binaria

El método de búsqueda binaria funciona, únicamente, sobre conjunto de datos ordenados.

El método consiste en dividir el intervalo de búsqueda en dos partes y compara el elemento que ocupa la posición central del conjunto. Si el elemento del conjunto no es igual al elemento buscado se redefinen los extremos del intervalo, dependiendo de si el elemento central es mayor o menor que el elemento buscado, reduciendo así el espacio de búsqueda.

Dado un conjunto de elementos contenidos en una estructura de datos lineal ordenado de manera ascendente, encontrar el elemento 325.

A	101	215	325	410	502	507	600	610	612	670
---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

$x = 325$ 

Algoritmo de búsqueda binaria

Idear un algoritmo iterativo de búsqueda binaria que a partir de un conjunto de datos lineal y ordenado y un valor dado permita encontrar la primera coincidencia de la llave dentro del conjunto.

Algoritmo de búsqueda binaria

Idear un algoritmo recursivo de búsqueda binaria que a partir de un conjunto de datos lineal y ordenado y un valor dado permita encontrar la primera coincidencia de la llave dentro del conjunto.

Un algoritmo iterativo que realiza una búsqueda binaria para encontrar la primera coincidencia de un dato dado es:

```
FUNC binary_iterative_search(intList[ ], intValue): ENTERO
    lowIndex  $\leftarrow$  0: ENTERO
    highIndex  $\leftarrow$  length(intList)-1: ENTERO
    MIENTRAS lowIndex  $\neq$  highIndex ENTONCES
        medium  $\leftarrow$  (highIndex + lowIndex)//2
        SI intList[medium] = intValue ENTONCES
            DEV medium
        EN CASO CONTRARIO:
            SI intValue < intList[medium] ENTONCES
                highIndex  $\leftarrow$  medium
            EN CASO CONTRARIO
                lowIndex  $\leftarrow$  medium + 1
    FIN MIENTRAS
    SI intList[lowIndex] = intValue ENTONCES
        DEV lowIndex
    EN CASO CONTRARIO
        DEV -1
FIN FUNC
```

Un algoritmo recursivo que realiza una búsqueda binaria para encontrar la primera coincidencia de un dato dado es:

```
FUNC binary_recursive_search(intList [], intValue, lowIndex, highIndex): ENTERO
    SI intList[lowIndex] = intValue ENTONCES
        DEV lowIndex

    SI lowIndex <> highIndex:
        medium ← (highIndex + lowIndex)//2
        SI intList[medium] = intValue ENTONCES
            DEV medium
        EN CASO CONTRARIO
            SI intValue < intList[medium] ENTONCES
                highIndex ← medium
            EN CASO CONTRARIO
                lowIndex ← medium + 1
            DEV binary_recursive_search(intList, intValue, lowIndex, highIndex)

FIN FUNC
```

El análisis de complejidad del método de búsqueda binaria está dado por el número total de comparaciones.

El mejor caso de complejidad se presenta cuando el elemento buscado (x) se encuentra a la mitad del conjunto $C_{\min} = 1$. El peor caso de complejidad se presenta cuando el elemento no se encuentra o se encuentra en la última comparación $C_{\max} = \log_2 (n)$ comparaciones. El caso promedio está dado por el número máximo de comparaciones entre 2, es decir $C_{\text{med}} = \log_2 (n) / 2$.

Se podría pensar que la búsqueda binaria es más eficiente que la lineal, empero, hay que recordar que la búsqueda binaria funciona dentro de un conjunto ordenado y, por ende, se debe ordenar antes el conjunto para que dicha búsqueda se pueda realizar.

Por lo tanto, el tiempo que se debe contabilizar para encontrar un elemento en un conjunto de datos desordenado a través de una búsqueda binaria está también en función del tiempo que se lleve ordenar el conjunto o en insertar de manera ordenada.

“50 % of the computer programming is trial and error, the other 50 % is copy and paste.”

@mobiledev_pawan

Algoritmos de búsqueda. Parte I.

PRÁCTICA 4

4. Algoritmos de búsqueda.

Parte I.

- Implementar búsqueda secuencial en Python tanto de forma iterativa como de forma recursiva para encontrar un nodo (con el parámetro de búsqueda que desees).
- Implementar búsqueda binaria en Python tanto de forma iterativa como de forma recursiva para encontrar un nodo (con el parámetro de búsqueda que desees). Antes de realizar la búsqueda se debe utilizar un método de ordenamiento directo (n^2) y uno logarítmico ($n \cdot \log(n)$).
- Obtener la complejidad algorítmica para cada implementación (búsqueda secuencial, búsqueda binaria con ordenamiento directo y búsqueda binaria con ordenamiento logarítmico).
- Graficar el comportamiento de los algoritmos para el mejor, el peor y el caso promedio para cada implementación (búsqueda secuencial, búsqueda binaria con ordenamiento directo y búsqueda binaria con ordenamiento logarítmico).

2.3 Búsqueda por transformación de llaves

2. ALGORITMOS DE BÚSQUEDA



2.3 Búsqueda por transformación de llaves

Las técnicas de búsqueda por transformación de llaves se basan en la idea de calcular una dirección en forma directa a partir de la llave de un nodo.

Normalmente se expresan utilizando una función de mapeo $H: K \rightarrow Y$, donde K es el dominio de la función formado por el conjunto de llaves que pueden ser transformadas y Y es el rango de la función formado por el conjunto de números enteros que representan a las direcciones de almacenamiento.

Esta técnica tiene un tiempo de búsqueda independiente al número de nodos en la estructura y su eficiencia es proporcional al tiempo utilizado para llevar a cabo la transformación.

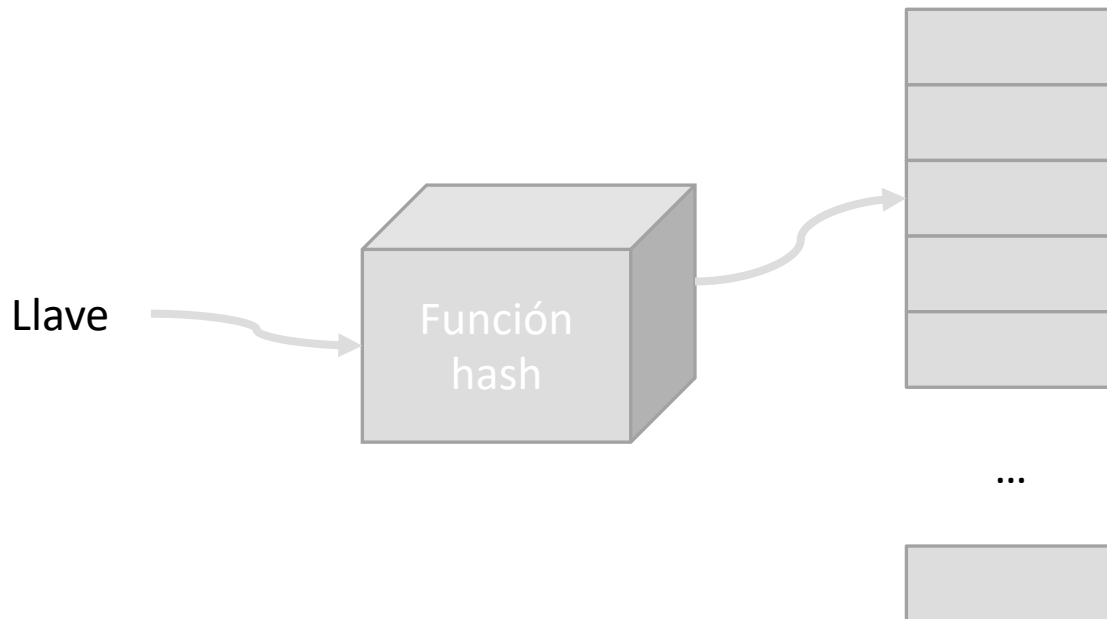
Hay que resaltar que si se quiere ocupar este tipo de técnicas se deben usar tanto en la recuperación de la información como en el almacenamiento de la misma.

2.4.1 Funciones hash

El método hash permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados y, a diferencia de los métodos anteriores donde su complejidad es proporcional al número de elementos del conjunto (entre más elementos más comparaciones), el método hash es independiente del número de nodos de la estructura.

Empero, para que este mecanismo de búsqueda funcione es necesario almacenar (insertar) los datos utilizando la misma función hash.

Una función hash permite calcular la posición donde se almacenará el dato aplicando una fórmula predefinida a un valor (llave) dado.



El método de transformación de claves permite tener acceso directo a los elementos del conjunto utilizando una función que convierte una clave dada en una dirección (o índice) dentro del conjunto.

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

La función hash (H) aplicada a la clave genera un índice o dirección en el conjunto que permite acceder directamente al elemento.

Funciones hash por módulo (división)

La función hash por módulo consiste en tomar el residuo de la división de la clave entre el número de componentes del conjunto, es decir:

$$H(k) = (k \bmod n) + 1$$

donde

n: es el número de elementos

k: es la clave del dato a buscar/almacenar

El resultado del lado derecho de la ecuación genera un número en el rango $[1, n]$.

Se tiene un conjunto de 100 elementos y se deben asignar direcciones en el rango $[1, 100]$ para almacenar o recuperar un dato. Obtener la posición que se ocupa para almacenar el elemento $k = 7,259$ por medio de la función hash por módulo.

$$H(k) = (7,259 \bmod 100) + 1$$
$$H(k) = 60$$

Funciones hash cuadrado

La función hash cuadrado consiste en elevar al cuadrado la clave dada y tomar los dígitos centrales como dirección. El número de dígitos que se deben considerar está determinado por el rango del índice. La función hash cuadrada está definida por:

$$H(k) = \text{dígitos_centrales}(k^2) + 1$$

Donde

k: es la clave del dato a buscar/almacenar

El resultado del lado derecho de la ecuación genera un número en el rango $[1, n]$.

Se tiene un conjunto de 100 elementos y se deben asignar direcciones en el rango [1, 100] para almacenar o recuperar un dato. Obtener la posición que se ocupa para almacenar los elementos $k = 7,259$ por medio de la función hash cuadrado.

$$k^2 = 52,693,081$$

$$H(k) = \text{dígitos_centrales}(52,6\mathbf{93},081) + 1 = 94$$

Debido a que el rango de índices varía de 1 a 100, se toman solamente dos dígitos centrales del cuadrado de las claves.

Funciones hash por plegamiento

La función hash por plegamiento consiste en dividir la clave en partes, tomando igual número de dígitos según el rango del arreglo y realizar operaciones con ellos.

Sea k la clave del dato a buscar, k está formada por los dígitos d_1, d_2, \dots, d_n . La función hash por plegamiento se define como:

$$H(k) = \text{díg_menos_significativo} ((d_1 \dots d_i) + (d_{i+1} \dots d_j) + \dots + (d_1 \dots d_n)) + 1$$

La operación entre dígitos puede ser suma (como en la función anterior) o multiplicación. Al final se genera un número en el rango $[1, n]$.

Se tiene un conjunto de 100 elementos y se deben asignar direcciones en el rango [1, 100] para almacenar o recuperar un dato. Obtener la posición que se ocupa para almacenar los elementos $k = 7,259$ por medio de la función hash por plegamiento.

$$H(k) = \text{díg_menos_significativo} (72 + 59)$$

$$H(k) = 131$$

$$H(k) = \text{díg_menos_significativo} (1 + 31) + 1$$

$$H(k) = 33$$

Debido a que el rango de índices varía de 1 a 100, se toman solamente dos dígitos del número (de derecha a izquierda).

Funciones hash por truncamiento

La función hash por truncamiento consiste en tomar algunos dígitos de la clave y formar con ellos una dirección. Este método es de los más sencillos, pero ofrece muy poca uniformidad en la distribución de claves.

Sea k la clave de dato a buscar, k está formada por los dígitos d_1, d_2, \dots, d_n . La función hash por truncamiento está dada por:

$$H(k) = \text{elegir_dígitos}(d_1, d_2, \dots, d_n) + 1$$

La elección de los dígitos es arbitraria, es decir, se pueden tomar los dígitos de las posiciones impares o de las posiciones pares. Además, se pueden unir de izquierda a derecha o viceversa.

Se tiene un conjunto de 100 elementos y se deben asignar direcciones en el rango [1, 100] para almacenar o recuperar un dato. Obtener la posición que se ocupa para almacenar los elementos $k = 7,259$ por medio de la función hash por truncamiento.

$$H(k) = \text{elegir_dígitos}(7,259) + 1$$

$$H(k) = 75 + 1$$

$$H(k) = 76$$

En este caso, se eligen los dígitos en las posiciones 1 y 3 y se unen de izquierda a derecha.

2.4.1 Colisiones

Una función hash debe ser simple de calcular y asignar direcciones de la manera más uniforme posible (suficientemente dispersas).

Sin embargo, cuando se realiza una asignación de la misma dirección a dos o más claves distintas, se dice que existe una colisión:

$$H(k_1) = d$$

$$H(k_2) = d$$

$$k_1 \neq k_2$$

Se tiene un conjunto de 100 elementos y se deben asignar direcciones en el rango $[1, 100]$ para almacenar o recuperar un dato. Obtener la posición que se ocupan para almacenar los elementos $k_1 = 7,259$ y $k_2 = 9,359$ utilizando la función hash por módulo.

$$H(k_1) = (7,259 \bmod 100) + 1 = 60$$

$$H(k_2) = (9,359 \bmod 100) + 1 = 60$$

Cuando $H(k_1) = H(k_2)$ para $k_1 \neq k_2$, se presenta una colisión que se debe resolver para poder almacenar la información.

Cuando se ocupan funciones hash es probable la presencia de colisiones, por ello, cuando ocurren, se debe tener alguna estrategia para solventarlas. Para el ejemplo anterior, una posible solución es aplicar la fórmula con un número primo cercano al rango de direcciones (100), el resultado sería:

$$H(k_1) = (7,259 \bmod 97) + 1 = 82$$

$$H(k_2) = (9,359 \bmod 97) + 1 = 48$$

Aplicando este cambio se puede observar que para $H(k_1)$ y $H(k_2)$ ya no hay colisión. Sin embargo, este cambio no garantiza que se presenten colisiones nuevamente para otro par de números.

La elección de un método adecuado para resolver colisiones es tan importante como la elección de una buena función hash.

Generalmente, no importa el método elegido, resulta costoso tratar las colisiones. Por ello se debe realizar un esfuerzo importante para encontrar una función que ofrezca la mayor uniformidad posible en la distribución de las claves.

La manera natural de resolver el problema de las colisiones es reservar una casilla por clave, pero esta solución puede tener un alto costo en memoria. Por ejemplo, si se intenta almacenar un número de cuenta de la UNAM, se tendrían que reservar 1 billón de registros y se ocuparían muy pocas localidades.

Para tratar de resolver colisiones existen diversos tipos de métodos, los cuales se pueden clasificar en:

- Reasignación
- Arreglos anidados
- Encadenamiento

Reasignación

Esta clasificación se basa en el principio de comparación y reasignación de elementos. Algunos métodos que trabajan bajo este esquema son:

- Prueba lineal
- Prueba cuadrática
- Doble dirección hash

Reasignación: Prueba lineal

El método de prueba lineal consiste en recorrer el conjunto de forma secuencial a partir del punto de colisión buscando un espacio para reservar el dato (búsqueda) o hasta encontrar el dato (inserción).

El proceso de búsqueda concluye cuando el elemento es hallado o cuando se encuentre una posición vacía. El método trata al conjunto como una estructura circular, es decir, el siguiente elemento del último es el primero.

Dentro de una estructura de datos lineal se ingresan las claves 25, 43, 56, 35, 54, 13, 80 y 104 utilizando la función hash por módulo:

$$H(k) = (k \bmod 10) + 1$$

Para tratar las colisiones se aplica una reasignación mediante prueba lineal.

$H(k) = (k \bmod 10) + 1$
Colisiones: prueba lineal

k	h(k)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

1	80
2	
3	
4	43
5	54
6	25
7	56
8	35
9	13
10	104

Si se aplica la función hash por módulo para buscar la clave 35, se obtiene la posición 6, sin embargo, en dicha dirección no se encuentra el elemento buscado, por lo que se inicia el recorrido secuencial del conjunto a partir de la siguiente posición.

k	$h(k)$
35	6

1	80
2	
3	
4	43
5	54
6	25
7	56
8	35
9	13
10	104



Reasignación: Prueba cuadrática

El método de prueba cuadrática consiste en recorrer el conjunto a partir del punto de colisión buscando el elemento con direcciones alternativas de la forma $D + i^2$, donde i es un valor entero aleatorio. Esta variación permite una mejor distribución de las llaves que colisionan.

Dentro de una estructura de datos lineal se ingresan las claves 25, 43, 56, 35, 54, 13, 80 y 104 utilizando la función hash por módulo:

$$H(k) = (k \bmod 10) + 1$$

Para tratar las colisiones se aplica una reasignación mediante prueba cuadrática, con $i = 2$.

$H(k) = (k \bmod 10) + 1$
Colisiones: prueba cuadrática, $i=2$


k	h(k)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

1	80
2	
3	
4	43
5	54
6	25
7	56
8	13
9	104
10	35

Si se aplica la función hash para buscar la clave 35 se obtiene una dirección igual a 6, sin embargo, en dicha dirección no se encuentra el elemento buscado, por lo que se calcula la siguiente posición a través de una prueba cuadrática.

k	h(k)
35	6

1	80
2	
3	
4	43
5	54
6	25
7	56
8	13
9	104
10	35



Reasignación: Doble dirección hash

El método de doble dirección hash consiste en generar otra dirección aplicando otra función hash a la dirección obtenida en la colisión. La función hash puede ser la misma aplicada al obtener la colisión u otra diferente.

Dentro de una estructura de datos lineal se ingresan las claves 25, 43, 56, 35, 54, 13, 80 y 104, según la función hash por módulo:

$$H(k) = (k \bmod 10) + 1$$

Además, se define la función H' para calcular direcciones alternativas en caso de colisión, aplicando otra vez función hash por módulo

$$H'(d) = ((d+1) \bmod 10) + 1$$

$$H(k) = (k \bmod 10) + 1$$

Colisiones: doble hash $H(d) = (d \bmod 10) + 1$

k	h(k)	h'(d)	h'(d')	h'(d'')
25	6			
43	4			
56	7			
35	6	8		
54	5			
13	4	6	8	10
80	1			
104	5	7	9	

1	80
2	
3	
4	43
5	54
6	25
7	56
8	35
9	104
10	13

Si se aplica la función hash para buscar la clave 13 se obtiene la dirección 4, sin embargo, en dicha dirección no se encuentra el elemento buscado, por lo que se aplica H' reiteradamente hasta localizar el elemento deseado.

k	$h(k)$	$h'(d)$	$h'(d')$	$h'(d'')$
13	4	6	8	10

1	80
2	
3	
4	43
5	54
6	25
7	56
8	35
9	104
10	13

Arreglos anidados

Este método consiste en crear un arreglo para cada elemento (llave) del conjunto donde se presenten colisiones.

La desventaja de esta solución es que resulta ineficiente o hasta insuficiente el espacio en memoria dependiendo del tamaño del conjunto.

Dada una estructura de datos lineal se ingresan las claves 25, 43, 56, 35, 54, 13, 80 y 104 utilizando la función hash por módulo

$$H(k) = (k \bmod 10) + 1$$

Para resolver las colisiones se utilizan arreglos anidados.

$H(k) = (k \bmod 10) + 1$
Colisiones: arreglos dinámicos

k	h(k)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

1	80									
2										
3										
4	43	13								
5	54	104								
6	25	35								
7	56									
8										
9										
10										

Encadenamiento

El método de encadenamiento consiste en que cada elemento del conjunto tiene una referencia hacia una lista ligada, la cual se va generando en tiempo real y guarda los valores que colisionan.

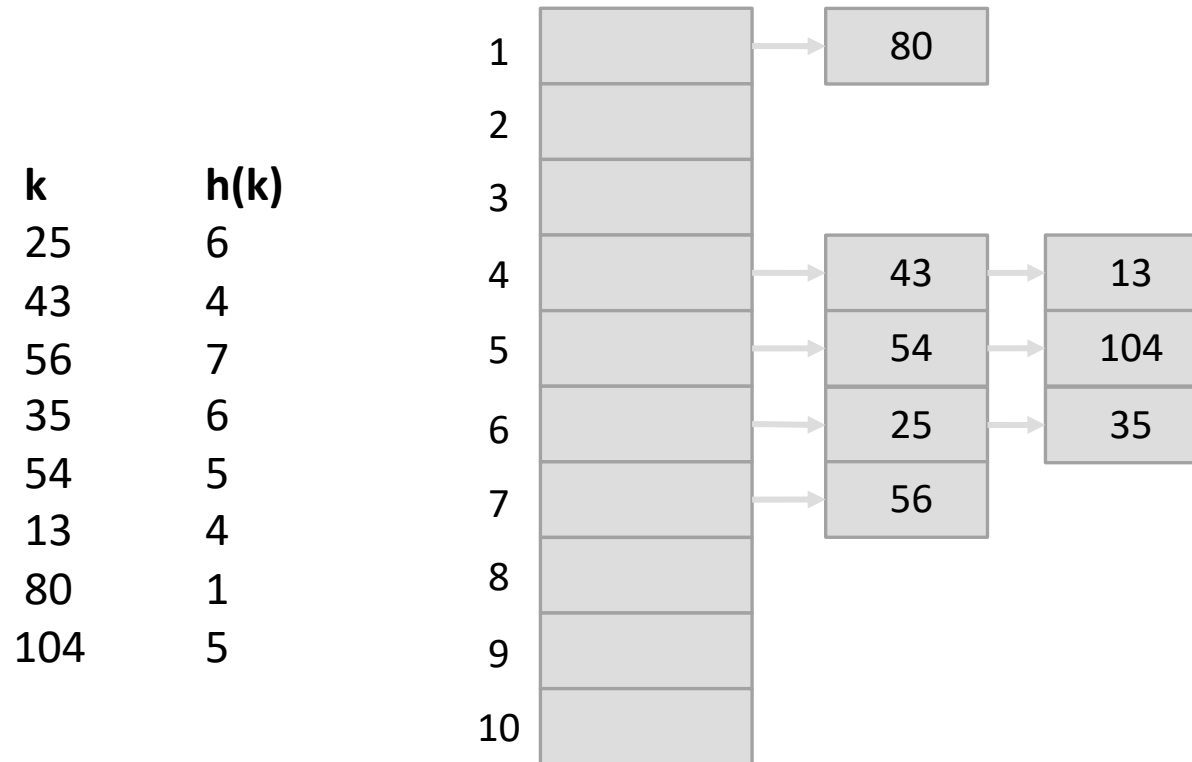
Las desventajas del método de encadenamiento es el uso propio de listas (si no se implementan correctamente) así como el tamaño que pueden tomar éstas.

Dada una estructura de datos lineal se ingresan las claves 25, 43, 56, 35, 54, 13, 80 y 104 utilizando la función hash por módulo

$$H(k) = (k \bmod 10) + 1$$

Para resolver las colisiones se utiliza encadenamiento (listas enlazadas).

$H(k) = (k \bmod 10) + 1$
Colisiones: encadenamiento



Algoritmos de búsqueda. Parte 2.

PRÁCTICA 5

5. Algoritmos de búsqueda.

Parte 2.

- Implementar un programa que permita almacenar y recuperar elementos de un nodo mediante una función hash. Se deben tratar las colisiones con algún método.
- Obtener la complejidad algorítmica (polinomio) para la implementación realizada.
- Crear las gráficas de complejidad de la función hash para el mejor, el peor y el caso promedio.

2 Algoritmos de búsqueda

Objetivo: Aplicar el método de búsqueda apropiado a conjuntos de datos residentes, tanto en la memoria principal, como en la memoria secundaria para generar algoritmos que resuelvan búsquedas.

2.1 Generalidades.

2.2 Definición de la operación de búsqueda.

2.3 Búsqueda por comparación de llaves.

2.3.1 Lineal.

2.3.2 Binaria.

2.4 Búsqueda por transformación de llaves.

2.4.1 Funciones de hash.

2.4.2 Colisiones.