Tema VI: Ordenación y Búsqueda

Introducción a la ordenación. Clasificación de los algoritmos de ordenación. Métodos directos. Métodos logarítmicos. Intercalación. Ordenación por montículos. Problema de la búsqueda estática. Búsqueda secuencial. Búsqueda binaria. Búsqueda interpolada. Cola de prioridades.

*Búsqueda y ordenamiento constituyen dos tareas fundamentales en la administración de datos, especialmente si se trata de grandes volúmenes. Las dos operaciones se desarrollan con base en un dato de referencia al que comúnmente se llama clave. La búsqueda permite encontrar un elemento particular en el conjunto, mientras que el ordenamiento consiste en ubicar los datos atendiendo a un criterio de manera que sea más fácil encontrar el elemento que se requiere o identificar las relaciones entre los datos.*

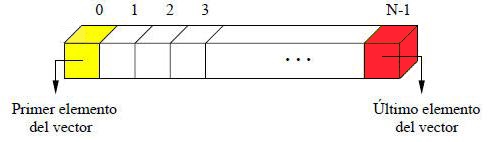
# 6.1 Búsqueda

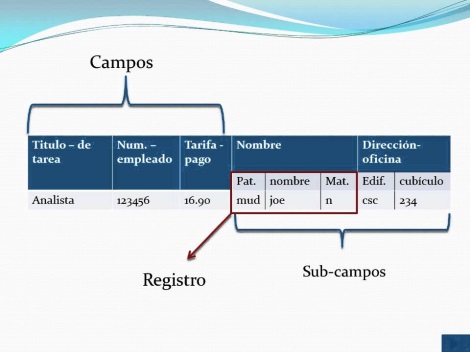
La operación de búsqueda en una estructura de datos nos permite encontrar datos que están previamente almacenados. La operación puede ser un éxito, si se localiza el elemento buscado o un fracaso en otros casos.

La búsqueda se puede realizar sobre un conjunto de datos ordenados, lo cual hace la tarea más fácil y consume menos tiempo; o se puede realizar sobre elementos desordenados, tarea más laboriosa y de mayor insumo de tiempo.

*La operación de búsqueda de un elemento X en un conjunto consiste en determinar si el elemento X pertenece al conjunto y en este caso dar su posición, o bien, determinar que el elemento X no pertenece al conjunto.*

## Tipos de búsqueda

*** Búsqueda Interna*** será aquella acción que se realice sobre datos que se encuentran en la memoria principal, por ejemplo en un arreglo.

******

***Búsqueda Externa*** es cuando todos sus elementos se encuentran en memoria secundaria (archivos almacenados en dispositivos de cinta, disco, etc.-)

## Métodos

Los métodos más usuales para la búsqueda son:

* *Búsqueda secuencial o lineal*
* *Búsqueda binaria*
* *Búsqueda por transformación de claves*

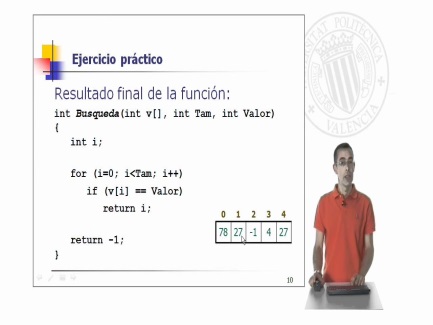
# 6.2 Búsqueda secuencial

El algoritmo más básico de todos es la **búsqueda secuencial** que nos permitirá recorrer una lista o un arreglo desde el inicio hasta el fin y se detiene cuando encontró el elemento buscado o que se llegue el final de la estructura, lo cual indicará que no se ha encontrado nada

La búsqueda secuencial es la técnica más simple para buscar en una lista de datos. Este método consiste en recorrer una lista (o arreglo) en forma secuencial y comparar cada elemento del arreglo (o de la lista) con el valor deseado, hasta que éste se encuentre o finalice el arreglo (o la lista).

Normalmente cuando la función de búsqueda termina con éxito, es decir encontró el dato buscado, interesa conocer en qué posición fue encontrado el dato buscado. Esta idea se puede generalizar en todos los métodos de búsqueda.

La búsqueda secuencial no requiere ningún requisito para el lote de datos, y por lo tanto no necesita estar ordenado.

https://www.youtube.com/watch?time\_continue=12&v=W3ClRnYb0KM&feature=emb\_logo

**algoritmo** busqueda\_secuencial  
inicio  
leer(t) // t es el número que buscamos  
encontrado <- falso  
desde i<-1 hasta N hacer  
    si A[i] = t entonces  
        encontrado <- verdadero  
    fin si  
fin desde  
si encontrado entonces  
    escribir('Elemento encontrado')  
sino  
    escribir('Elemento no encontrado')  
fin si

*secuenEjemplo*

**Inicio**

**Ingresar**  X

**Leer** V(100)

**Desde** I = 1 hasta 100 **hacer**

**Si**  V(i) = X  **entonces**

**Imprimir** “Encontrado” V(i), “Posición” i

**Fin**

**Fin Si**

**Fin desde**

**Imprimir** “Dato no encontrado”

**Fin**

## 6.2.1 Consideraciones

* El método es solo adecuado para listas cortas de datos.
* A la hora de analizar la complejidad del método secuencial, tenemos que tener en cuenta el caso más favorable y el más desfavorable.
* Cuando el elemento no se encuentra tiene que realizar las *n* comparaciones. Y en los casos en que el elemento buscado se localiza, este podrá estar en el primer lugar, en el último o en un lugar intermedio.
* Entonces, al buscar un elemento en un arreglo de N componentes se harán:
  + *N comparaciones* si el elemento no se localiza
  + *N comparaciones* si el elemento está en la última posición
  + *1 comparación* si está en el primer lugar
  + *i comparaciones* si está en un lugar intermedio (posición 1 < i < N)
* Podemos suponer que el número medio de comparaciones a realizar es de **(*n + 1) / 2***, que es aproximadamente igual a la mitad de los elementos de la lista.

## 6.2.2 Casos

Una aplicación de la búsqueda secuencial puede ser en:

* Vector: Se recorre el vector hasta encontrar una componente cuyo valor coincida con el buscado ó hasta que se acabe el vector. En este último caso el algoritmo debe indicar la no-existencia de dicho valor. Este método es válido para vectores ordenados o no, aunque para los vectores ordenados veremos otros métodos más eficientes.
* Vector Ordenado: Cuando el vector de búsqueda esta ordenado se consigue un algoritmo más eficiente, sin más que modificar la condición de fin del caso anterior. La ventaja que se obtiene es que una vez sobrepasado el valor buscado, no es necesario recorrer el resto del vector para saber que el valor no existe.
* Matriz: Se realiza mediante el anidamiento de dos bucles de búsqueda HASTA cuya finalización vendrá dada por la aparición del valor buscado o la terminación de la matriz. El recorrido se podrá hacer indistintamente por fila o por columna.

### Búsqueda en vector

Los procesos de búsqueda involucran recorrer un arreglo completo con el fin de encontrar algo. Lo más común es buscar el menor o mayor elemento (cuando se puede establecer un orden), o buscar el índice de un elemento determinado.

Para buscar el menor o mayor elemento de un arreglo, podemos usar cualquier estrategia vista en Estructura de datos I, por ejemplo la estrategia de suponer que el primero o el último es el menor (mayor), para luego ir comparando con cada uno de los elementos, e ir actualizando el menor (mayor). A esto se le llama Búsqueda lineal o secuencial.

|  |  |  |
| --- | --- | --- |
| Búsqueda del menor | Búsqueda del mayor | Búsqueda del elemento |
| for (i=1;i<n;i++)  if ( a[i]<menor )  menor=a[i]; | for (i=0;i<n-1;i++)  if ( a[i]>mayor )  mayor=a[i]; | for (i=0;i<n;i++)  if ( a[i]==elemento\_buscado )  encontrado=i; |

Ejemplo 6.1: Desarrollar un programa que posea una función que reciba como parámetro un arreglo de 10 enteros, y un entero, y retorne la posición del entero si es que se encuentra, de lo contrario devolver –1.

#include <stdio.h>

int encuentra(int A[], int b) {

int k=1, result=-1;

do{

if (A[k]== b)

result =k;

else

k++;

}while ((result==-1)&&(k<10));

return result;

}

int main() {

int i, x[10];

for(i=0;i<10;i++)

scanf("%d",&x[i]);

i = encuentra(x, 10);

printf("resultado %d\n",i);

return 0;

}

## 6.2.3 Búsqueda externa

Se denomina búsqueda externa cuando todos los elementos se encuentran en memoria secundaria (archivos almacenados en dispositivos tales como cintas y discos magnéticos).

Se debe tener en cuenta para el desarrollo del método con búsqueda externa lo siguiente:

**a)** El ingreso de la información: los datos ingresan de a uno en la memoria.

**b)** Se supone al primer dato como mayor (no cualquiera de los campos como en búsqueda interna)

**c)** Se realiza la comparación de una variable (nota promedio) con respecto a la variable auxiliar MAYOR, y se repite dicha comparación hasta el fin del lote de datos, formando de esta manera un CICLO DE BUSQUEDA.

Las mismas consideraciones son válidas para la búsqueda de MINIMOS.

Dado el siguiente problema: *encontrar en un archivo el mayor promedio de un alumno.*

En este caso se resuelve la búsqueda de máximo promedio con la técnica del supuesto. Se supone que el primer registro es el máximo.

mayor

Con esta técnica necesariamente debemos realizar doble lectura, una versión mejorada consiste en inicializar a la variable promedio con un valor lo suficientemente chico para que en la primera lectura, el primer valor leído resulte el mayor promedio.

mayor0

## 6.3 Búsqueda binaria

Es válido exclusivamente para datos ordenados y consiste en comparar en primer lugar con la componente central de la lista, y si no es igual al valor buscado se reduce el intervalo de búsqueda a la mitad derecha o izquierda según donde pueda encontrarse el valor a buscar.

El algoritmo termina si se encuentra el valor buscado o si el tamaño del intervalo de búsqueda queda anulado.

Este mecanismo es muy eficaz para buscar un elemento cualquiera que esté en una lista ordenada, y recibe el nombre de **Búsqueda Binaria o Dicotómica** cuya resolución se base en el algoritmo de divisiones sucesivas en mitades.

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos.

## 6.3.1 Desarrollo del método

Es posible aprovechar mejor la lista ordenada si somos inteligentes en nuestras comparaciones. En la búsqueda secuencial, cuando comparamos contra el primer ítem, hay a lo sumo n−1 ítems restantes para verificar si el primer ítem no es el valor que estamos buscando. En lugar de buscar secuencialmente en la lista, una búsqueda binaria comenzará examinando el ítem central. Si ese ítem es el que estamos buscando, hemos terminado. Si no es el ítem correcto, podemos utilizar la naturaleza ordenada de la lista para eliminar la mitad de los ítems restantes. Si el ítem que buscamos es mayor que el ítem central, sabemos que toda la mitad inferior de la lista, así como el ítem central, se pueden ignorar de la consideración posterior. El ítem, si es que está en la lista, debe estar en la mitad superior.

Podemos entonces repetir el proceso con la mitad superior. Comenzar en el ítem central y compararlo con el valor que estamos buscando. Una vez más, o lo encontramos o dividimos la lista por la mitad, eliminando por tanto otra gran parte de nuestro espacio de búsqueda posible. La figura muestra cómo este algoritmo puede encontrar rápidamente el valor **54.**

Arreglo Inicial

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 17 | 23 | 31 | 48 | 51 | **54** | 63 | 78 | 88 | 91 |
| Izq= 0  Primera pasada |  |  |  | Valor medio |  |  |  |  | Der=9 |
| 17 | 23 | 31 | 48 | 51 | **54** | 63 | 78 | 88 | 91 |
| Segunda pasada |  |  |  |  | Izq= 5 |  | Valor medio |  | Der=9 |
| 17 | 23 | 31 | 48 | 51 | **54** | 63 | 78 | 88 | 91 |
| Tercera pasada |  |  |  |  | Izq= 5 | Valor medio=der |  |  |  |
| 17 | 23 | 31 | 48 | 51 | **54** | 63 | 78 | 88 | 91 |

Valor buscado < valor medio

El algoritmo de resolución seria

*En Pseudocódigo:*

**Comenzar**

**Leer** X

**Ingresar** V(I) I = 1,100

ULTIMO = 100

PRIMERO = 1

**Hasta** PRIMERO <= ULTIMO

CENTRAL = [(PRIMERO + ULTIMO) / 2]

**Si** V(CENTRAL) = X

**Entonces**

**Imprimir** “Registro encontrado “ V(CENTRAL)

**Parar**

**Fin\_si**

**Si** V(CENTRAL) > X

**Entonces**

ULTIMO = CENTRAL – 1

**Si\_no**

PRIMERO = CENTRAL + 1

**Fin\_si**

**Fin\_Mientras**

**Imprimir** “Registro no encontrado”

**Parar**

En C:

#include<stdio.h>

int main() {

int A[20]={21,32,43,54,65,76,87,98,109,110,211,212,313,314,415,416,417,518,519,620};

Int inf,sup,mit,dato,n=20;

printf("dame un dato a buscarn: ");

scanf("%d",&dato);

inf=0;

sup=n;

while (inf<=sup) {

mit=(inf+sup)/2;

if (A[mit]==dato) {

printf("dato %d encontrado posicion %d \n",dato,mit);

break;

}

if (A[mit]>dato) {

sup=mit;

mit=(inf+sup)/2;

}

if (A[mit]<dato) {

inf=mit;

mit=(inf+sup)/2;

}

}

return 0;

}

## 6.3.2 Análisis de la búsqueda binaria

Para analizar el algoritmo de búsqueda binaria, necesitamos recordar que cada comparación elimina aproximadamente la mitad de los ítems restantes de la consideración. ¿*Cuál es el número máximo de comparaciones que este algoritmo requerirá para examinar la lista completa*? Si empezamos con *n* ítems, alrededor de n/2 ítems se dejarán después de la primera comparación. Después de la segunda comparación, habrá aproximadamente n/4. Después n/8, n/16, y así sucesivamente.

Igual que en el método secuencial la complejidad del método se va a medir por los casos extremos que puedan presentarse en el proceso de búsqueda.

El caso más favorable se dará cuando el primer elemento central es el buscado, en cuyo caso se hará una sola comparación. El caso más desfavorable se dará cuando el elemento buscado no está en las sublistas, en este caso se harán en forma aproximada log 2(n) comparaciones, ya que en cada ciclo de comparaciones el número de elementos se reduce a la mitad, factor de 2. Por lo tanto, el número medio de comparaciones que se realizarán con este método es de: **(1 + log 2(n)) / 2**

Si comparamos las fórmulas dadas en ambos métodos (ver apartado 6.2. Método secuencial), resulta que para el mismo valor de N el método binario es más eficiente que el método secuencial; además la diferencia es más significativa cuanto más crece N.

A pesar de que una búsqueda binaria es generalmente mejor que una búsqueda secuencial, será importante tener en cuenta que para valores pequeños de *n*, el costo adicional del ordenamiento probablemente no vale la pena. De hecho, siempre debemos considerar si es rentable asumir el trabajo extra del ordenamiento para obtener beneficios en la búsqueda. Si podemos ordenar una sola vez y luego buscar muchas veces, el costo del ordenamiento no es tan significativo. Sin embargo, para listas grandes, incluso ordenar una vez puede resultar tan costoso que simplemente realizar una búsqueda secuencial desde el principio podría ser la mejor opción.

# 6.4 Búsqueda por interpolación

El algoritmo de búsqueda binaria siempre selecciona el elemento central del vector para compararlo con el elemento a buscar y dividir la lista.

Es posible realizar una modificación a este algoritmo de tal forma que el elemento seleccionado no sea el central sino aquel que se “correspondería” con el elemento buscado si la distribución de valores en la lista fuera uniforme.

Este método se puede aplicar solamente a tablas o archivos ordenados. Como su nombre lo indica se trata de llegar al elemento buscado por medio de la interpolación lineal. El procedimiento es recursivo; como en el caso de la búsqueda binaria, en cada paso se van modificando los límites, disminuyendo el intervalo, hasta llegar al elemento buscado.

Los criterios de la búsqueda son los siguientes:

* Ningún método sistematizado busca como lo hace el ser humano, por ejemplo, al buscar en la guía telefónica, una persona cuyo apellido empieza con W.
* Nosotros tenemos idea de una cierta proporcionalidad, respecto del tamaño del archivo, y vamos directamente a la parte alta de la guía telefónica, siguiendo esa idea de proporcionalidad. En el caso de la búsqueda del apellido que comienza con W vamos a la parte final de la guía.

*Entonces ….*

El método consiste en **tratar de acertar** en qué parte del intervalo está la clave que se está buscando en lugar de ciegamente dividir el arreglo a la mitad. Para ello desarrollamos el siguiente ejemplo:

Formula de interpolación:

x = izq + (key-a[izq])\*(der-izq)/(a[der]-a[izq])

Vector a [1,2,3,4,5,6,7,8]

Valor buscado : 6

Nuestros valores iniciales serían:

* key = 6, (*valor buscado*)
* izq = 0, (*menor posición*)
* der = 7, (*mayor posición*)
* a[izq] = 1, (*valor menor posición)*
* a[der] = 8, (*valor mayor posición*)

Reemplazamos en la fórmula:

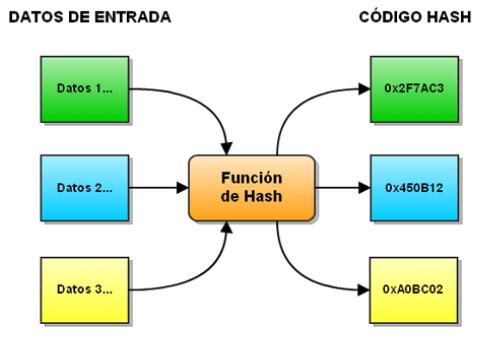
Índice = 0 + [(6-1) \* (7-0) / (8-1)] = 5

# 6.5 Búsqueda por transformación de claves (hash)

Es un método que aumenta la velocidad de búsqueda, pero que no requiere que los elementos estén ordenados. Consiste en asignar a cada elemento un índice mediante una transformación del elemento. Esta correspondencia se realiza mediante una función de conversión, llamada *función hash*. La correspondencia más sencilla es la identidad, esto es, al número 0 se le asigna el índice 0, al elemento 1 el índice 1, y así sucesivamente.

## 6.5.1 Función Hash

La función de hash ideal debería ser *biyectiva*, esto es, que a cada elemento le corresponda un índice, y que a cada índice le corresponda un elemento.

Esta función hash debe ser simple de calcular y debe asignar direcciones de la manera más uniformemente posible, es decir debe generar posiciones diferentes dadas dos claves diferentes.

Para trabajar con este método de búsqueda debe elegirse previamente:

1. Una función Hash que sea fácil de calcular y que distribuya uniformemente las claves
2. Un método para resolver colisiones. Si estas se presentan se deberá buscar un método que resuelva las colisiones.

La función de hash depende de cada problema y de cada finalidad, y se pueden utilizar con números o cadenas, pero las más utilizadas son:

### Restas sucesivas

Esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas. Por ejemplo, si el número de expediente de un alumno universitario está formado por el año de entrada en la universidad, seguido de un número identificativo de tres cifras, y suponiendo que entran un máximo de 400 alumnos al año, se le asignarían las claves:

1998-000 –> 0 = 1998000-1998000  
1998-001 –> 1 = 1998001-1998000  
1998-002 –> 2 = 1998002-1998000  
…

### Aritmética modular

El índice de un número es resto de la división de ese número entre un número N prefijado, preferentemente primo. Los números se guardarán en las direcciones de memoria de 0 a N-1. Este método tiene el problema de que cuando hay N+1 elementos, al menos un índice es señalado por dos elementos (teorema del palomar). A este fenómeno se le llama colisión, y es tratado más adelante. Si el número N es el 13, los números siguientes quedan transformados en:

**13000000 –> 0**12345678 –> 7  
13602499 –> 1  
71140205 –> 6  
73062138 –> 6

Un caso particular de la aritmética modular es la llamada aritmética del reloj. Cuando a las 10 de la mañana se le agrega 5 horas se llega a las 3 de la tarde, es decir “10 + 5 = 3”.

También si a las 2 de la tarde se le quita 4 horas, el resultado es las 10, lo que equivale a decir que “2 − 4 = 10”. Esta aritmética del reloj se llama más generalmente aritmética módulo 12 y se realiza dentro del conjunto Z12 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11} cuyos elementos se llaman enteros módulo 12. En realidad cualquier número entero es equivalente a un entero módulo 12 que se obtiene como el residuo (nunca negativo) de la división entre 12.

### ****Mitad del cuadrado****

Consiste en elevar al cuadrado la clave y tomar las cifras centrales. Este método también presenta problemas de colisión:

**123\*123=15129 –> 51**  
136\*136=18496 –> 84  
730\*730=532900 –> 29  
625\*625=390625 –>06

### ****Truncamiento****

Consiste en ignorar parte del número y utilizar los elementos restantes como índice. También se produce colisión. Por ejemplo, si un número de 8 cifras se debe ordenar en un arreglo de 1000 elementos, se pueden tomar la primera, la tercera y la última cifra para formar un nuevo número:

**13000000 –> 100**12345678 –> 138  
13602499 –> 169  
71140205 –> 715

### ****Plegamiento****

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

**13000000 –> 130=130+000+00**12345678 –> 657=123+456+78  
13602499 –> 259=136+024+99  
25000009 –> 259=250+000+09

## 6.5.2 Colisión

Una colisión ocurre cuando se asigna una misma dirección a dos o más claves diferentes.

#### Tratamiento de colisiones

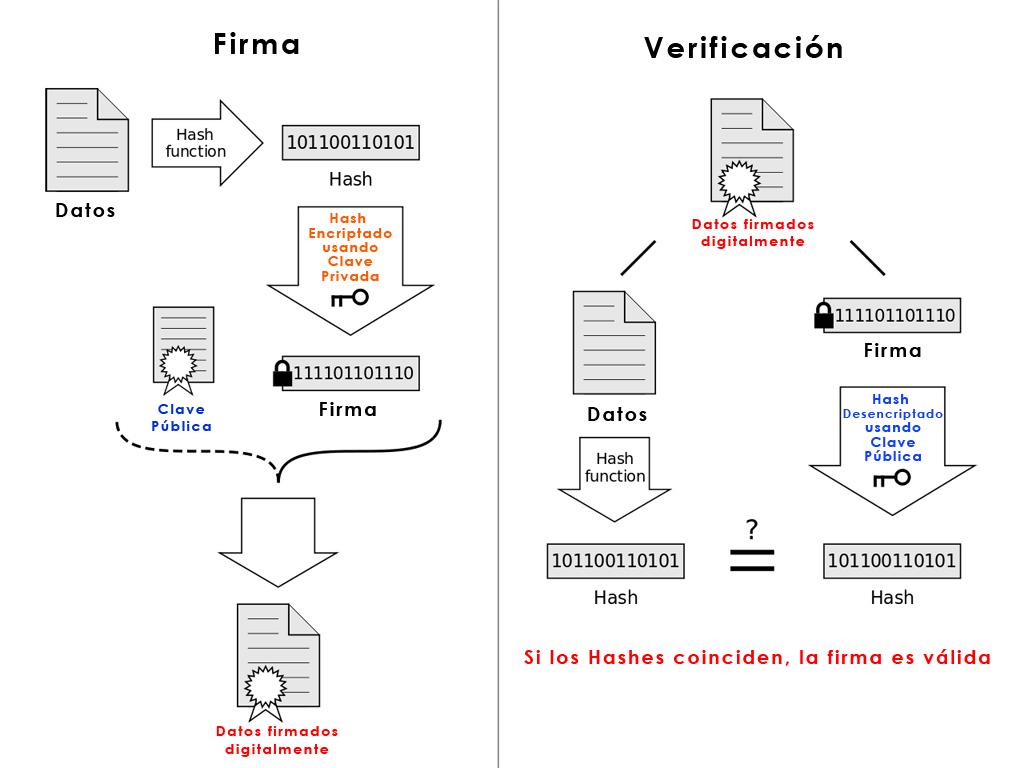
Pero ahora se nos presenta el problema de qué hacer con las colisiones, qué pasa cuando a dos elementos diferentes les corresponde el mismo índice. Pues bien, hay tres posibles soluciones:

1. Cuando el índice correspondiente a un elemento ya está ocupado, **se le asigna el primer índice libre a partir de esa posición**. Este método es poco eficaz, porque al nuevo elemento se le asigna un índice que podrá estar ocupado por un elemento posterior a él, y la búsqueda se ralentiza, ya que no se sabe la posición exacta del elemento.
2. También **se pueden reservar unos cuantos lugares al final del arreglo** para alojar a las colisiones. Este método también tiene un problema: ¿Cuánto espacio se debe reservar? Además, sigue la lentitud de búsqueda si el elemento a buscar es una colisión.
3. Lo más efectivo es, en vez de crear un arreglo de número, **crear un arreglo de punteros**, donde cada puntero señala el principio de una lista enlazada. Así, cada elemento que llega a un determinado índice se pone en el último lugar de la lista de ese índice. El tiempo de búsqueda se reduce considerablemente, y no hace falta poner restricciones al tamaño del arreglo, ya que se pueden añadir nodos dinámicamente a la lista

*Ejemplo*:

Si la posición 397 ya estaba ocupada, el registro con clave 0596397 es colocado en la posición 398, la cual se encuentra disponible. Una vez que el registro ha sido insertado en esta posición, otro registro que genere la posición 397 o la 398 es insertado en la posición siguiente disponible.

*Ejemplo de aplicación*

**

Firmar un documento no es algo novedoso, pero la firma digital si lo es un poco más y nos ayuda a verificar la identidad del emisor de un mensaje.

El método más simple de firma digital consiste en crear un **hash** de la información enviada y cifrarlo con nuestra clave privada para que cualquiera con nuestra clave pública pueda ver el hash real y verificar que el contenido del archivo es el que hemos mandado nosotros.

# 6.6 Cola de prioridad

Una cola de prioridades es una estructura de datos en la que los elementos se atienden en el orden indicado por una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen.

Este tipo especial de colas tienen las mismas operaciones que las colas FIFO, pero con la condición de que los elementos se atienden en orden de prioridad.

Ejemplos de la vida diaria serían la sala de urgencias de un hospital, ya que los enfermos se van atendiendo en función de la gravedad de su enfermedad.

Entendiendo la prioridad como un valor numérico y asignando a altas prioridades valores pequeños, las colas de prioridad nos permiten añadir elementos en cualquier orden y recuperarlos de menor a mayor.

Hay 2 formas de implementación:

* Añadir un campo a cada nodo con su prioridad. Resulta conveniente mantener la cola ordenada por orden de prioridad.
* Crear tantas colas como prioridades haya, y almacenar cada elemento en su cola.

Existen 2 tipos de colas de prioridad:

**Colas de prioridades con ordenamiento ascendente**: en ellas los elementos se insertan de forma arbitraria, pero a la hora de extraerlos, se extrae el elemento de menor prioridad.

**Colas de prioridades con ordenamiento descendente**: son iguales que las colas de prioridad con ordenamiento ascendente, pero al extraer el elemento se extrae el de mayor prioridad.

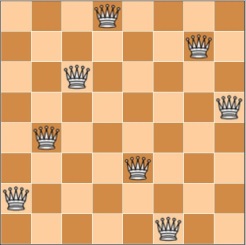
Las colas de prioridad tienen múltiples usos. Con frecuencia se emplean para implementar algoritmos voraces. Este tipo de algoritmos suele tener una iteración principal, y una de las tareas a realizar en cada una de dichas iteraciones es seleccionar un elemento de entre varios que minimiza (o maximiza) un cierto criterio de optimalidad local. El conjunto de elementos entre los que se ha de efectuar la selección es frecuentemente dinámico y admitir inserciones eficientes. Algunos de estos algoritmos incluyen:

Algoritmos de Kruskal y Prim para el cálculo del ́árbol de expansión mínimo de un grafo etiquetado.

Algoritmo de Dijkstra para el cálculo de caminos mínimos en un grafo etiquetado.

Construcción de códigos de Huffman (códigos binarios de longitud media m mínima).

# 6.7 Búsqueda por fuerza bruta

Los algoritmos de Fuerza Bruta son capaces de encontrar la solución a cualquier problema por complicado que sea. Su fundamento es muy simple, probar todas las posibles combinaciones, recorrer todos los caminos hasta dar con la situación que es igual que la solución. No le importa iniciar caminos malos o muy malos, al llegar a su final y ver que su destino no es la solución, se iniciará otro camino en busca de la solución iniciando nuevamente hasta encontrar la solución.

**Búsqueda por fuerza bruta**, **búsqueda combinatoria**, **búsqueda exhaustiva** o simplemente **fuerza bruta**, es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

Por ejemplo, un algoritmo de fuerza bruta para encontrar el divisor de un númeronatural *n* consistiría en enumerar todos los enteros desde 1 hasta *n*, chequeando si cada uno de ellos divide *n* sin generar resto.

La búsqueda por fuerza bruta es sencilla de implementar y, siempre que exista, encuentra una solución. Sin embargo, su coste de ejecución es proporcional al número de soluciones candidatas, el cual es exponencialmente proporcional al tamaño del problema. Por el contrario, la búsqueda por fuerza bruta se usa habitualmente cuando el número de soluciones candidatas no es elevado, o bien cuando éste puede reducirse previamente usando algún otro método heurístico.

Este método sólo se usa cuando el número de posibilidades a evitar es lo suficientemente grande y que contando con el tiempo de ejecución del código necesario para implementarlo, reduzca ampliamente el tiempo para encontrar el resultado esperado.

Características

* Es el algoritmo más simple posible.
* Consiste en probar todas las posibles posiciones del patrón en el texto.
* Requiere espacio constante.
* Realiza siempre saltos de un carácter.
* Compara de izquierda a derecha.
* Realiza la búsqueda del patrón en un tiempo O(mn).
* Realiza 2n comparaciones previstas de los caracteres del texto.



### Lógica:

* Se sitúa el patrón en la primera posición, y se compara carácter a carácter hasta encontrar un fallo o llegar al final del patrón.
* Se pasa a la siguiente posición y se repite el proceso.
* El proceso finaliza al alcanzar el final del texto
* No existe un pre procesamiento del patrón.

### Descripción

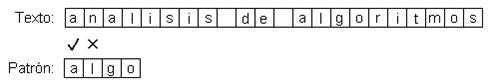
No requiere ninguna fase de pre proceso previo, ni un espacio extra constante además del espacio asignado al patrón y al texto.

### Para la búsqueda:

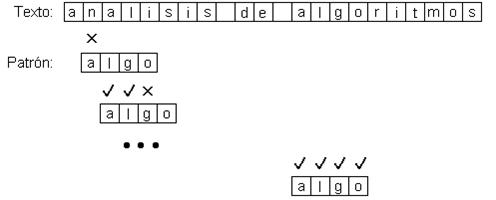
* Consiste en la comparación de todas las posiciones del texto entre 0 y el n-m, si una ocurrencia del patrón corresponde o no.
* Si encuentra una no ocurrencia, o una ocurrencia total del patrón, salta un carácter hacia la derecha.

### Ejemplo (https://www.youtube.com/watch?v=KrGUYZ\_gRAo)

Se alinea la primera posición del patrón con la primera posición del texto, y se comparan los caracteres uno a uno hasta que se acabe el patrón, esto es, se encontró una ocurrencia del patrón en el texto, o hasta que se encuentre una discrepancia.



Si se detiene la búsqueda por una discrepancia, se desliza el patrón en una posición hacia la derecha y se intenta calzar el patrón nuevamente.



**Código C**

**/\* Búsqueda \*/**

**void FB(char \*x, int m, char \*y, int n)**

**{      
  int i,j;**

**for (j = 0; j <= n - m; ++j)**

**{  
      for (i = 0; i < m && x[i] == y[i + j]; ++i);  
    if (i >= m) OUTPUT(j);  
    }  
 }**

El algoritmo encuentra todas las ocurrencias del patrón en el texto.  
El **patrón**se denota por x = x[0, ..., m-1]; su longitud es igual a m.

El **texto**se denota por y = y[0, ..., n-1]; su longitud es igual a n.

Ambas secuencias son estructuras sobre un sistema finito de caracteres llamado alfabeto denotado por S, con tamaño igual a s.

**x = GCAGAGAG**

**y = GCATCGCAGAGAGTATACAGTACG**

Otro ejemplo https://www.youtube.com/watch?v=FKIkPCE3ui4