

# algoritmos de ORDENACION

## Ordenación Interna (en RAM)

### Bubble Sort (Burbuja)

**Idea:** Compara pares adyacentes e intercambia si están mal.

**Usar cuando:** lista pequeña (<20) o para enseñanza.

**No usar cuando:** listas grandes (muy lento).

**Complejidad:** Mejor  $O(n)$ , Peor  $O(n^2)$ .

```
void bubbleSort(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

### Ejemplo

[5,3,8,4] → [3,5,8,4] → [3,5,4,8] → [3,4,5,8]

### Selection Sort (Selección)

**Idea:** Buscar el mínimo, lo pone al inicio, y repite.

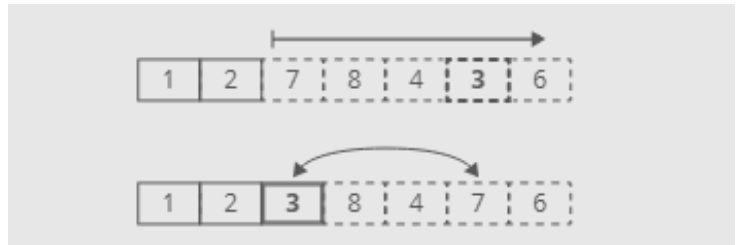
**Usar cuando:** intercambios son costosos y lista pequeña.

**No usar cuando:** velocidad es prioridad.

**Complejidad:** Siempre  $O(n^2)$ .

```
void selectionSort(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n-1; i++) {
        int min = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[min]) min = j;
        }
        int temp = arr[min];
        arr[min] = arr[i];
        arr[i] = temp;
    }
}
```

### Ejemplo



### Insertion Sort (Inserción)

**Idea:** Inserta cada elemento en su posición en la parte ordenada.

**Usar cuando:** lista pequeña o casi ordenada.

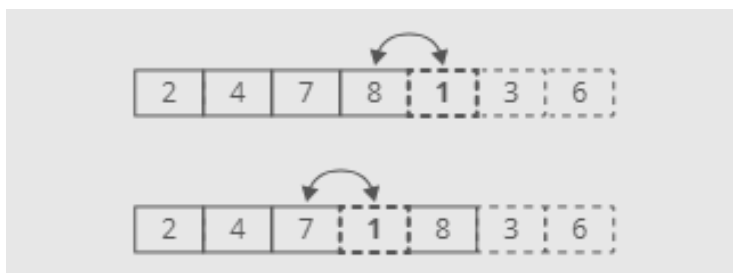
**No usar cuando:** listas grandes y muy desordenadas.

**Complejidad:** Mejor  $O(n)$ , Peor  $O(n^2)$ .

```
void insertionSort(int arr[]) {
    int n = arr.length;
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key)
            arr[j+1] = arr[j];
        j--;
        arr[j+1] = key;
    }
}
```



### Ejemplo



## Ordenación Externa (Disco, datos grandes)

### Mezcla Directa (Direct Merge Sort)

Algoritmo de ordenamiento externo que divide el archivo en particiones de tamaño fijo y las va mezclando en sucesivas pasadas hasta ordenar todo.

# algoritmos de ORDENACION

## Proceso resumido

Primera pasada:

- Leer clave por clave del archivo original (Fo).
- Distribuir alternadamente en F1 y F2 » bloques de tamaño 1.
- Intercalar bloques de F1 y F2 en Fo » secuencias de longitud 2.

Iteraciones sucesivas:

- Se duplican los tamaños de bloques en cada paso: 2, 4, 8, 16...
- Cada fusión genera bloques más grandes y ordenados.

1. Repetir hasta que el archivo original (Fo) quede completamente ordenado.

### VENTAJA

Fácil de implementar. No necesita algoritmos adicionales (trabaja directo con el archivo).

### DESVENTAJA

Ineficiente en la primera pasada. Genera bloques de tamaño fijo sin aprovechar secuencias ya ordenadas.

## Mezcla Equilibrada (Natural Merge Sort)

Optimización de la mezcla directa: en lugar de bloques fijos de 1, aprovecha secuencias ya ordenadas naturalmente

## Proceso resumido

Primera iteración (mejora clave):

- Detectar secuencias ya ordenadas en el archivo (runs).
- Distribuir las en F1 y F2.
- Así los bloques iniciales no tienen tamaño fijo de 1, sino que pueden ser más largos.

Iteraciones siguientes:

- Igual que mezcla directa: fusionar bloques entre F1 y F2, duplicando tamaños.

Repetir hasta que quede un archivo final completamente ordenado

### VENTAJA

Eficiente, usa pocos archivos, rápido en datos grandes.

### DESVENTAJA

Complejo, sin ventaja en datos pequeños, depende del algoritmo interno.

## Algoritmo Polifásico

Variante optimizada de mezcla múltiple. Usa tres archivos auxiliares y fases sucesivas de intercalación para reducir accesos a disco.

## Proceso resumido

Fase 1 – División y Ordenación:

- Leer bloques del archivo original (Fo).
- Ordenar cada bloque en RAM (algoritmo interno).
- Distribuir bloques ordenados en F1 y F2.

Fase 2 – Intercalación:

- Intercalar primer bloque de F1 con F2 » guardar en Fo.
- Siguiendo bloque de F1 y F2 » guardar en F3.
- Repetir alternando salida entre Fo y F3.
- Repetir fases hasta que todo quede ordenado en Fo.

### VENTAJA

Eficiente, menos archivos, rápido en grandes datos.

### DESVENTAJA

Complejo, inútil en datos pequeños, depende del algoritmo interno.

# algoritmos de BUSQUEDA

## Búsqueda Interna

### Tablas Hash

- Tabla Hash = estructura de datos que guarda pares clave-valor.
- Operaciones principales: Búsqueda, Inserción, Eliminación en tiempo casi constante  $O(1)$ .

### Funciones de Dispersión (Hash Functions)

Transforman una clave en un índice dentro de la tabla.

Deben distribuir uniformemente los datos para evitar colisiones.

Ejemplos:

- Suma de Dígitos  $\Rightarrow$  suma de valores de los caracteres.
- Multiplicación  $\Rightarrow$  clave  $\times$  constante  $\Rightarrow$  parte fraccional  $\times$  tamaño de tabla.
- División (mod)  $\Rightarrow$  clave  $\%$  tamaño de la tabla.

### Colision

Suceden cuando dos claves generan el mismo índice

### Métodos de Resolución

#### Encadenamiento (Chaining)

- Cada posición guarda una lista enlazada con las claves que colisionan.
- Simple y efectivo cuando no hay muchas colisiones.

#### Dirección Abierta (Open Addressing)

- Se busca otra posición libre en la misma tabla.

#### Estrategias:

- Sondeo Lineal  $\Rightarrow$  revisar posiciones secuenciales.
- Sondeo Cuadrático  $\Rightarrow$  revisar posiciones con incrementos cuadrados ( $1^2, 2^2, 3^2 \dots$ ).
- Doble Hashing  $\Rightarrow$  usar segunda

## Ventajas de Tablas Hash

- Acceso rápido a datos.
- Muy usadas en diccionarios, conjuntos, caches.
- Dependen de:
  - Buena función hash (distribución uniforme).
  - Manejo eficiente de colisiones.

## Búsqueda Externa

### Búsqueda Secuencial mediante Bloques

Cómo funciona: Lee bloques de datos de tamaño fijo uno por uno hasta encontrar el dato o llegar al final.

Ventajas:

- Fácil de implementar
- No necesita estructuras extras

Desventajas:

- Lenta en archivos grandes
- Ineficiente si el dato está al final

Aplicación: Archivos pequeños o datos cercanos al inicio.

### Búsqueda Secuencial con Índices

Cómo funciona: Usa un índice que apunta a bloques.

1. Buscar en el índice
2. Ir al bloque y buscar dentro

Ventajas:

- Más rápida que la secuencial pura
- Eficiente en archivos grandes

Desventajas:

- Requiere espacio extra
- Actualizar el índice cuesta tiempo

Aplicación: Archivos grandes con acceso frecuente.

### Búsqueda por Transformación de Claves (Hashing)

Cómo funciona: Aplica una función hash a la clave  $\Rightarrow$  devuelve dirección del dato.

Ventajas:

- Muy rápida (acceso directo)
- Eficiente para búsquedas individuales

Desventajas:

- Colisiones (dos claves en la misma dirección)
- Necesita manejo extra de colisiones (encadenamiento, direccionamiento abierto)

Aplicación: Bases de datos, sistemas de archivos.

# algoritmos de BUSQUEDA

## Búsqueda Interna

### Tablas Hash

- Tabla Hash = estructura de datos que guarda pares clave-valor.
- Operaciones principales: Búsqueda, Inserción, Eliminación en tiempo casi constante  $O(1)$ .

### Funciones de Dispersión (Hash Functions)

Transforman una clave en un índice dentro de la tabla.

Deben distribuir uniformemente los datos para evitar colisiones.

Ejemplos:

- Suma de Dígitos  $\Rightarrow$  suma de valores de los caracteres.
- Multiplicación  $\Rightarrow$  clave  $\times$  constante  $\Rightarrow$  parte fraccional  $\times$  tamaño de tabla.
- División (mod)  $\Rightarrow$  clave  $\%$  tamaño de la tabla.

### Colision

Suceden cuando dos claves generan el mismo índice

### Métodos de Resolución

#### Encadenamiento (Chaining)

- Cada posición guarda una lista enlazada con las claves que colisionan.
- Simple y efectivo cuando no hay muchas colisiones.

#### Dirección Abierta (Open Addressing)

- Se busca otra posición libre en la misma tabla.

#### Estrategias:

- Sondeo Lineal  $\Rightarrow$  revisar posiciones secuenciales.
- Sondeo Cuadrático  $\Rightarrow$  revisar posiciones con incrementos cuadrados ( $1^2, 2^2, 3^2 \dots$ ).
- Doble Hashing  $\Rightarrow$  usar segunda

## Ventajas de Tablas Hash

- Acceso rápido a datos.
- Muy usadas en diccionarios, conjuntos, caches.
- Dependen de:
  - Buena función hash (distribución uniforme).
  - Manejo eficiente de colisiones.

## Búsqueda Externa

### Búsqueda Secuencial mediante Bloques

Cómo funciona: Lee bloques de datos de tamaño fijo uno por uno hasta encontrar el dato o llegar al final.

Ventajas:

- Fácil de implementar
- No necesita estructuras extras

Desventajas:

- Lenta en archivos grandes
- Ineficiente si el dato está al final

Aplicación: Archivos pequeños o datos cercanos al inicio.

### Búsqueda Secuencial con Índices

Cómo funciona: Usa un índice que apunta a bloques.

1. Buscar en el índice
2. Ir al bloque y buscar dentro

Ventajas:

- Más rápida que la secuencial pura
- Eficiente en archivos grandes

Desventajas:

- Requiere espacio extra
- Actualizar el índice cuesta tiempo

Aplicación: Archivos grandes con acceso frecuente.

### Búsqueda por Transformación de Claves (Hashing)

Cómo funciona: Aplica una función hash a la clave  $\Rightarrow$  devuelve dirección del dato.

Ventajas:

- Muy rápida (acceso directo)
- Eficiente para búsquedas individuales

Desventajas:

- Colisiones (dos claves en la misma dirección)
- Necesita manejo extra de colisiones (encadenamiento, direccionamiento abierto)

Aplicación: Bases de datos, sistemas de archivos.

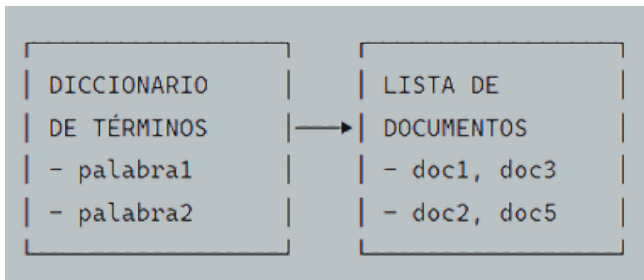
# *algoritmos de* **BUSQUEDA**

## LISTAS INVERTIDAS

### ¿QUÉ SON?

Estructuras que indexan palabras clave para encontrar rápidamente documentos

### ESTRUCTURA



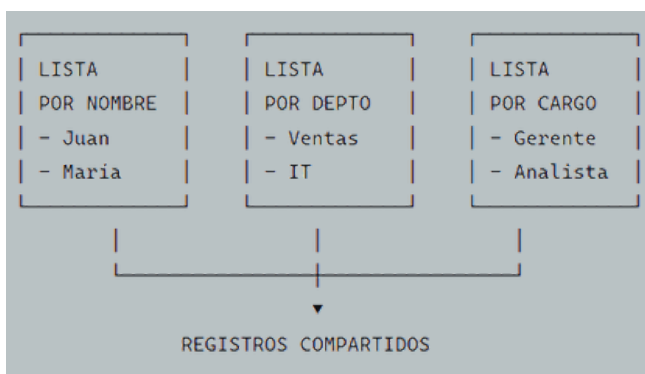
### FUNCIONAMIENTO

1. **Indexación:** Analizar documentos »> extraer términos »> actualizar diccionario
2. **Búsqueda:** Consultar término »> obtener lista de documentos directamente

## MULTILISTAS

### ¿QUÉ SON?

Generalización de listas invertidas para múltiples criterios de búsqueda



### FUNCIONAMIENTO

1. Cada registro se inserta en varias listas
2. Búsquedas combinadas usando intersección de listas