-

UT9 - Ordenación

Clasificación, Introducción

- Existe un orden lineal definido para los elementos del conjunto a clasificar, por ejemplo, “menor que”.
- La clasificación puede dividirse en interna y externa.
- Estabilidad del método (capacidad de mantener el orden relativo de elementos con iguales claves).
- Los algoritmos más simples requieren tiempos de $O(n^2)$, otros O(n * log n) y algunos, para clases especiales de datos, O(n).
- Los objetos a clasificar son estructuras complejas y contienen al menos un elemento del tipo para el cual se define la relación de ordenación (clave).

Algoritmos y Estructuras de Datos 6

6

Clasificación por Inserción

- en el i - ésimo recorrido se inserta el i - ésimo elemento en el lugar correcto entre los (i-1) elementos anteriores, los cuales fueron ordenados previamente.
- Después de hacer la inserción, se encuentran clasificados los elementos V[1], ..., V[i].
- En un vector:

```
for i = 2 to n do
  mover V[i] hacia la posición j <= i tal que
      V[i].clave < V[k].clave para j<=k<i , y
      V[i].clave >= V[j-1].clave o j=1.
```

Algoritmos y Estructuras de Datos 11

11

Ordenar por inserción directa

1	2	3	4	5	6	7	8	
44	55	12	42	94	18	6	67	i
44	55	12	42	94	18	6	67	2
12	44	55	42	94	18	6	67	3
12	42	44	55	94	18	6	67	4
12	42	44	55	94	18	6	67	5
12	18	42	44	55	94	6	67	6
6	12	18	42	44	55	94	67	7
6	12	18	42	44	55	67	94	8

Mostrando cómo queda el vector después de terminada la iteración para cada valor de "i".

20

Inserción directa en vector: análisis del método

```
Comienzo
(1) Desde i = 2 hasta N
    hacer
(2)     Aux ← V[i]
(3)     j = i - 1
(4)     mientras j > 0 y
    Aux.clave < V[j].clave
    hacer
(5)         V[j+1] ← V[j]
(6)         j ← j-1
(7)     fin mientras
(8)     V[j+1] ← Aux
(9) fin desde
Fin
```

```
Comienzo
(1) Desde: exactamente N-1
    veces.
(2)     O(1)
(3)     O(1)
(4) mientras: peor caso N-i veces,
    mejor caso 1 vez

(5)         O(1)
(6)         O(1)
(7)     fin mientras
(8)     O(1)
(9) fin desde
Fin
```

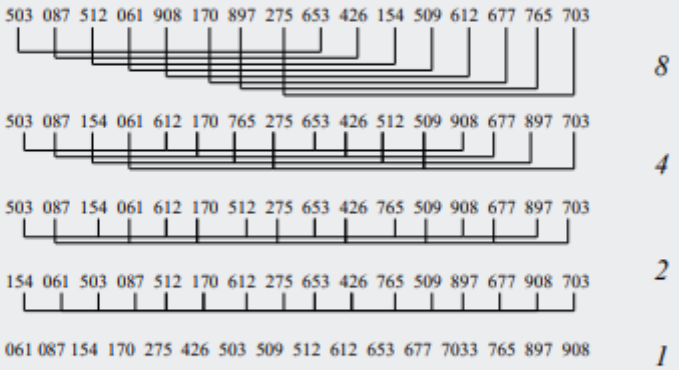

21

Método de Shell (Shellsort)

O clasificación por disminución de incrementos.

- Si el algoritmo mueve los elementos sólo una posición por vez su tiempo de ejecución será proporcional a N^2.
- Buscamos un mecanismo para que los elementos puedan dar grandes saltos en vez de pequeños pasos.
- Ejemplo: primero dividimos los 16 registros en 8 grupos de dos, $(R_1, R_9),(R_2, R_{10}),\ldots,(R_8, R_{16})$ y clasificamos cada grupo por separado.
- A continuación dividimos los elementos en 4 grupos de 4, y clasificamos cada grupo por separado.
- Continuamos así hasta tener un sólo grupo con los 16 elementos.

Clasificación por disminución de incrementos



Análisis del algoritmo de Shell.

- Para elegir una buena secuencia de incrementos es necesario analizar el tiempo de ejecución en función de estos incrementos.
- No se conoce la mejor secuencia para grandes valores de N.
- Los incrementos no deben ser múltiplos de sí mismos, de forma que las cadenas se mezclen entre sí lo más a menudo posible.
- Secuencias razonables (en orden inverso) :
  - 1, 4, 13, 40, 121 ...
  - 1, 3, 7, 15, 31 ...
- El orden es de $n^{1.26}$.

Ordenar por burbuja

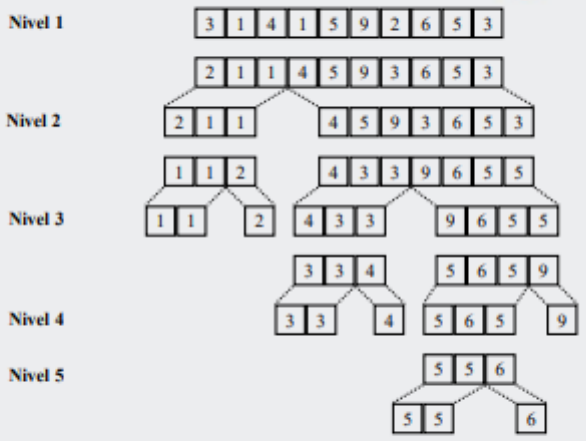
1	2	3	4	5	6	7	8	
44	55	12	42	94	18	6	67	i
6	44	55	12	42	94	18	67	1
6	12	44	55	18	42	94	67	2
6	12	18	44	55	42	67	94	3
6	12	18	42	44	55	67	94	4
6	12	18	42	44	55	67	94	5
6	12	18	42	44	55	67	94	6
6	12	18	42	44	55	67	94	7

Mostrando cómo queda el vector después de terminada la iteración para cada valor de "i".

Quicksort

- Es tal vez el algoritmo más eficiente para clasificación interna. Su orden es de **n*log n**.
- La idea es clasificar un conjunto de elementos V[1]..V[N] tomando uno de ellos, de clave V[p].clave, como *pivote*, procurando que sea la mediana del conjunto, de forma que esté precedido y sucedido por más o menos la mitad de los elementos del conjunto.
- Se permutan los elementos de forma que, para algún valor de j, todos los que tienen clave menor que V[p].clave se encuentran a la izquierda de j, y los de clave mayor o igual están a la derecha.

Quicksort



Desarrollo del algoritmo de Quicksort

- El algoritmo opera sobre un conjunto de elementos V[1]..V[N] definido de manera externa.
- El procedimiento **quicksort(i,j)** ordena desde V[i] hasta V[j] en el mismo lugar.

quicksort(i,j)

(1) Pivote ← ObtenerClavePivote(i,j)
(2) Si existe un Pivote **entonces**
(3) permutar V[i]..V[j] de forma que, para alguna **k** tal que **i+1 <= k < j**,
V[i].clave...V[k-1] .clave < Pivote y
V[k].clave...V[j] .clave >= Pivote
(5) quicksort(i,k-1)
(6) quicksort(k,j)
Fin si

Quicksort: Análisis del tiempo de ejecución

- El algoritmo insume en el mejor caso y en el caso promedio un tiempo **O(n*log n)** y en el peor caso, O(n^2).
- El mejor caso se da cuando el pivote es en cada elección la mediana del conjunto.
- El peor caso se da cuando el pivote elegido es un extremo del conjunto.
- En todos los casos las sentencias del método principal pueden ser O(**1**), excepto ***partición*** que será O(**j-i**). El orden de cada llamada será entonces O(**j-i**).
- Se ejecutarán 2N-1 llamadas al algoritmo.

Clasificación

- Ejemplos de métodos de distribución
  - Bucketsort
    - Orden "cuasi lineal"
  - Binsort (clasificación por urnas)
    - Sin claves repetidas.
    - Con claves repetidas: el rango de las claves es menor a la cantidad de ellas.
    - Orden N
  - Radix sort o clasificación por residuos.
    - Orden N.
    - Se descompone la clave sub tipos.
  - Cuentas por distribución
    - Orden N.

Clasificación por Selección

- Selección directa
  - Arroja un orden del tiempo de ejecución $O(N^2)$ en todos los casos.
  - El orden está dado por las comparaciones, ya que realiza siempre exactamente N-1 intercambios.
- Heapsort
  - La idea es mejorar las comparaciones para obtener el menor de los elementos.
  - Se obtiene un O(N log N) en todos los casos.

	1	2	3	4	5	6	7	8
iter	223	784	376	285	015	440	666	007
1	007	784	376	285	015	440	666	223
2	007	015	376	285	784	440	666	223
3	007	015	223	285	784	440	666	376

12

Selección directa: análisis del orden del tiempo de ejecución

```
(1) Desde i = 1 hasta N - 1 hacer
(2)     IndiceDelMenor ← i
(3)     ClaveMenor ← V[i].clave
(4)     Desde j = i + 1 hasta N hacer
(5)         Si V[j].clave < ClaveMenor entonces
(6)             IndiceDelMenor ← j
(7)             ClaveMenor ← V[j].clave
(8)         Fin si
(9)     Fin desde
(10)    intercambia (V[i], V[IndiceDelMenor])
    Fin desde
```

- Las sentencias de las líneas 2,3,5,6,7 y 10 son todas de O(1).
- El bloque de sentencias que abarca la sentencia 4, se ejecuta exactamente N-i veces.
- Ese bloque, más la sentencia de intercambio, se ejecutan exactamente N-1 veces.
- Por lo tanto, este método es $O(N^2)$, con la particularidad que los intercambios son siempre N-1.

14

Arboles parcialmente ordenados.

– Es un ***árbol binario completo***, completado por niveles. En el nivel más bajo, si no está completo, las hojas faltantes serán del extremo derecho, es decir que el nivel se completa de izquierda a derecha

– La clave de un nodo cualquiera **v** no es mayor que la de sus hijos. Nótese que los nodos con claves pequeñas no necesitan estar a niveles más altos que los de claves mayores.

18

Arboles parcialmente ordenados

- **SuprimeMinimo**.
  – Se **devuelve el elemento con menor clave**, que se encuentra en la **raíz**. Se debe ahora **arreglar el árbol** para que se siga cumpliendo la propiedad de árbol parcialmente ordenado.
  – Para ello se **toma l**a hoja de más a la derecha del nivel más bajo y se coloca en **la raíz**.
  – Luego se lleva este elemento **lo más abajo posible**, **intercambiándolo** con el hijo que tenga la prioridad más baja, hasta que el elemento se encuentre en una hoja o en una posición en la cual **las claves de los hijos sean iguales o mayores.**

21

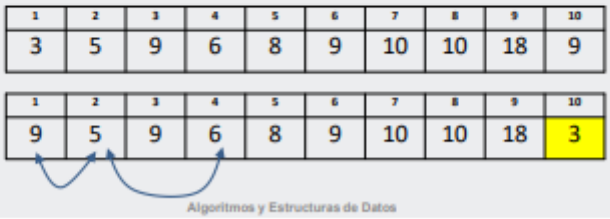
Heapsort

- Los elementos que se van eliminando de S se pueden almacenar en V[i+1],.., V[n], clasificados en orden inverso, es decir, V[i+1] >= V[i+2] >=... V[n].
- La operación **SuprimeMinimo** puede realizarse entonces:
  - intercambiando V[1] con V[i].
  - si el nuevo V[1] viola la propiedad del árbol parcialmente ordenado, debe descender en el árbol hasta su lugar, para lo cual se usa el procedimiento ***DesplazaElemento***.

1	2	3	4	5	6	7	8	9	10
3	5	9	6	8	9	10	10	18	9

Heapsort

- Los elementos que se van eliminando de S se pueden almacenar en V[i+1],.., V[n], clasificados en orden inverso, es decir, V[i+1] >= V[i+2] >=... V[n].
- La operación **SuprimeMinimo** puede realizarse entonces:
  - intercambiando V[1] con V[i].
  - si el nuevo V[1] viola la propiedad del árbol parcialmente ordenado, debe descender en el árbol hasta su lugar, para lo cual se usa el procedimiento ***DesplazaElemento***.



1	2	3	4	5	6	7	8	9	10
3	5	9	6	8	9	10	10	18	9

1	2	3	4	5	6	7	8	9	10
9	5	9	6	8	9	10	10	18	3

Heapsort: Método de desplazar un elemento

```
DesplazaElemento(Primero, Ultimo: tipo entero);
Comienzo
        Actual ← Primero;
        mientras Actual <= (Ultimo div 2) hacer
                Si ultimo = 2*Actual  entonces                          //Actual tiene un hijo
                        Si  V[Actual].clave  > V[2*Actual].clave entonces
                          Intercambia(V[Actual], V[2*Actual])
                             fin si
                         Actual ← Ultimo
                Sino                                                    // Actual tiene dos hijos
                        Menor ← MenorHijo(2*Actual, 2*Actual+1)         //indice del menor
                             Si  V[Actual].clave  > V[Menor].clave entonces
                                    Intercambia(V[Actual], V[Menor])
                                    Actual ← Menor
                             Sino  Actual ← Ultimo
                             Fin si
                Fin si
        Fin mientras
Fin
```


33

Heapsort – armar el heap

1	2	3	4	5	6	7	8	9	10
5	6	10	18	8	9	9	10	3	9

nodos internos — hojas

- Se observa que todos los elementos de la posición 6 en adelante son hojas.
- Por lo tanto basta con desplazar el elemento de la posición 5 al lugar que le corresponde, luego 4, y así sucesivamente hasta llegar al 1.
- Esta operación es más eficiente que si se fueran insertando de a uno los elementos en un heap vacío.

35

Algoritmo de Heapsort: análisis del método

```
Comienzo
    Desde i = N div 2 hasta 1 hacer
        DesplazaElemento(i, N);          O(logN)   } O(N*logN)
    Fin desde
    Desde i = N hasta 2 hacer                                    +
        Intercambia(V[1], V[i])          O(1)
        DesplazaElemento(1, i-1)         O(logN)   } O(N*logN)
    Fin desde
Fin
                                                   ------------
                                                   O(N*logN)
```


Binsort (clasificación por urnas).

- Los algoritmos de clasificación ya vistos tienen una cota mínima de O(n*log n).
- Si se conoce el rango de los valores de las claves se puede pasar a O(n).
- Ej: Si el tipo de clave es entero con rango 1..n, y existen n claves diferentes, entonces es posible diseñar un algoritmo de clasificación de orden n.
- Ver animación en

https://www.cs.usfca.edu/~galles/visualization/CountingSort.html

Algoritmo de Binsort con listas

```
Binsort (entrada, m)
    urnas := new array de m listas vacías
    for i = 1 to n
(1)     insertar entrada[i] en urnas[DMS(entrada[i]).clave]
(2)  for i = 0 to m-1
(3)     Ordenar(urnas[i])
(4)  salida := Concatenar(urnas[0]… urnas[m-1])
     devolver salida
```

- Orden:
  - Las sentencias (1) y (2) llevan tienen un tiempo de ejecución de O(n)
  - Las (3) y (4) O(m), donde m es el número de claves diferentes.
  - El total del algoritmo es de O(m+n).
  - Animación en https://www.cs.usfca.edu/~galles/visualization/BucketSort.html

Radix o Clasificación por residuos.

- Asumimos un TipoClave constituido por k elementos, $f_1, f_2, .., f_k$, de tipos $t_1, t_2, ..., t_k$.
- Se desea clasificar los registros en orden lexicográfico.
- Ejemplos:

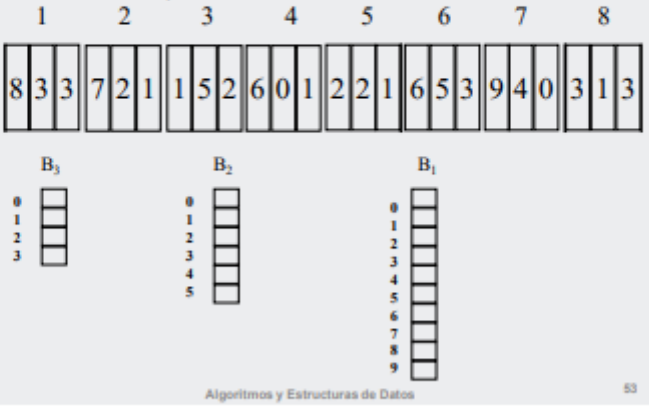
```
type
  TipoClave = record
                dia : 1..31;
                mes: 1..12;
                año: 1900..1999;
  end;
```

```
type
  TipoClave = array[1..10] of char;
```

Ejemplo de Radix

TipoClave = record
k1: 0..9;
k2: 0..5;
k3: 0..3;

A : array[1..n] of TipoClave



53

Análisis de algoritmo de Radix.

- El ciclo de la línea (2) tarda un tiempo O(s_i), donde s_i es el número de valores diferentes del tipo t_i.
- El ciclo de las líneas (3) y (4) lleva un tiempo O(n).
- El ciclo de las líneas (5) y (6) lleva un tiempo O(*si*).
- El tiempo total entonces es :

$$\sum_{i=1}^{k} O(s_i + n) = O(k * n + \sum_{i=1}^{k} s_i) = O(n + \sum_{i=1}^{k} s_i)$$

56