Ordenación interna

(Parte II)

Prof. Cristian Cappo

Agosto/2020

En esta clase

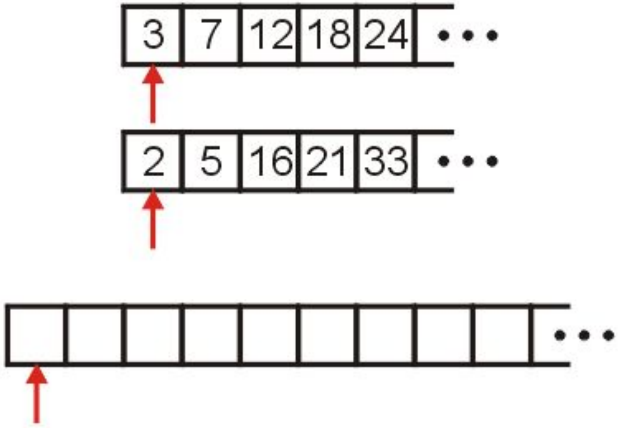
- Revisión de algoritmos
  - Mezcla (*MergeSort*)
  - Montículo (*HeapSort*)
  - CountingSort
  - Residuos (*RadixSort*)
  - Cubetas (BucketSort o BinSort)
- Ejercicios...

Ordenación
MergeSort

- Definido recursivamente
- Primera implementación por John von Neumann en 1945 en la ENIAC
- Suponga que :
  - Dividimos una lista desordenada en dos sub-listas
  - Ordenamos ambas sub-listas
- ¿Como puedo recombinar estas dos sub-listas en una sola lista ordenada?

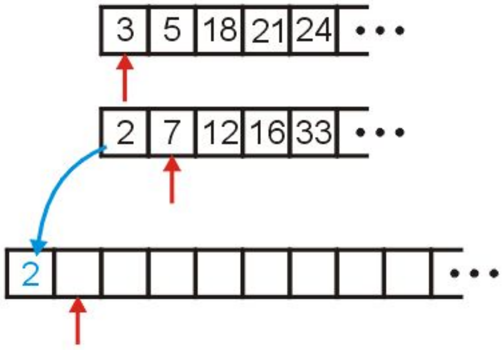
Ordenación
MergeSort

- Ejemplo:
  - Considere dos listas ordenadas y un arreglo vacío
  - Definimos tres índices al principio de cada arreglo



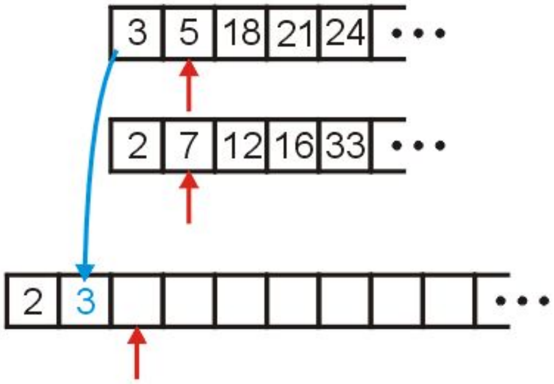
Ordenación
MergeSort

- Ejemplo:
  - Ahora comparamos 2 y 3 : 2 < 3
  - Copiamos 2 en el arreglo vacío
  - Incrementamos los índices



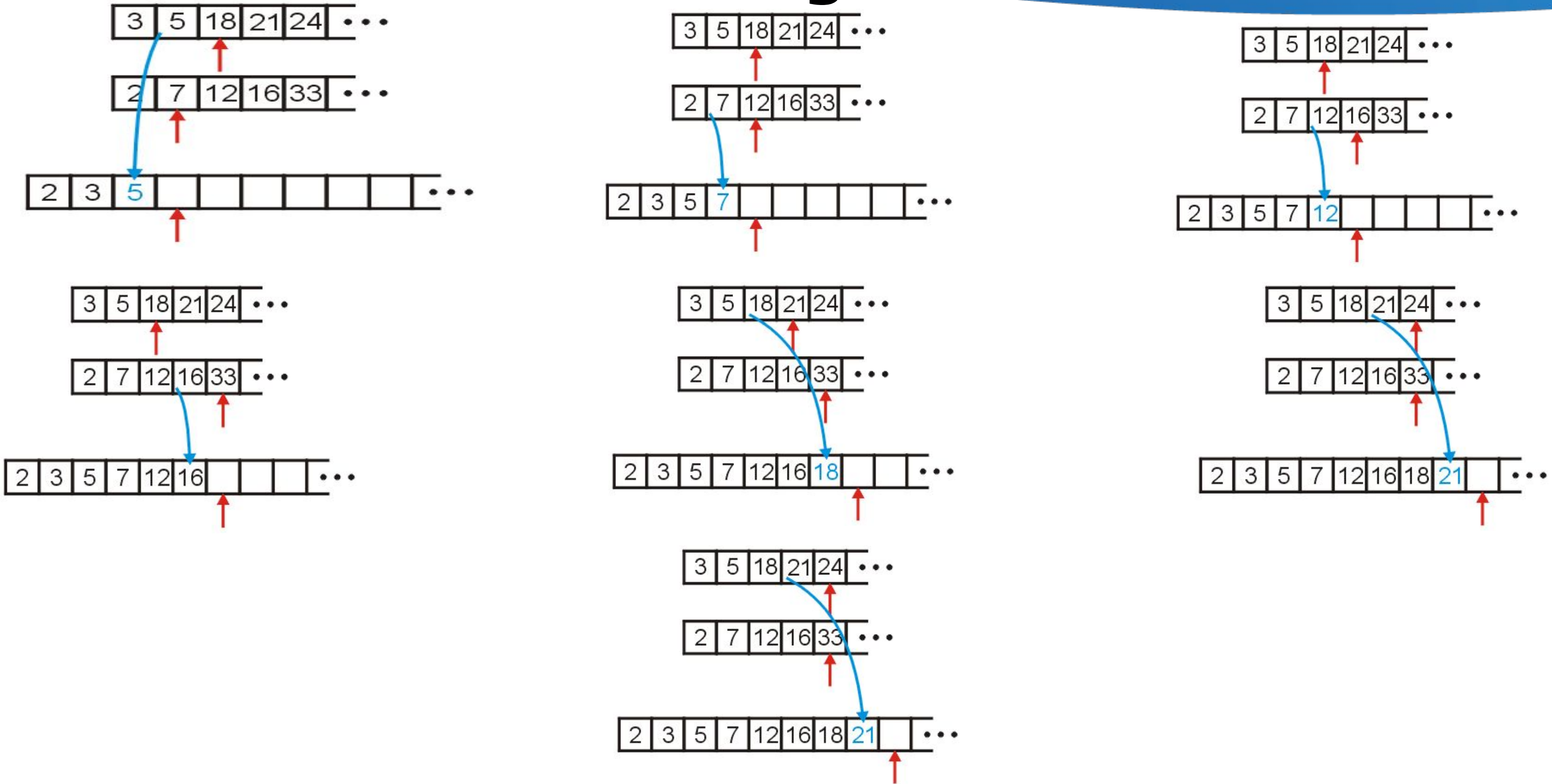
Ordenación
MergeSort

- Ahora comparamos 3 y 7
- Copiamos 3 en el arreglo de abajo
- Incrementamos los índices



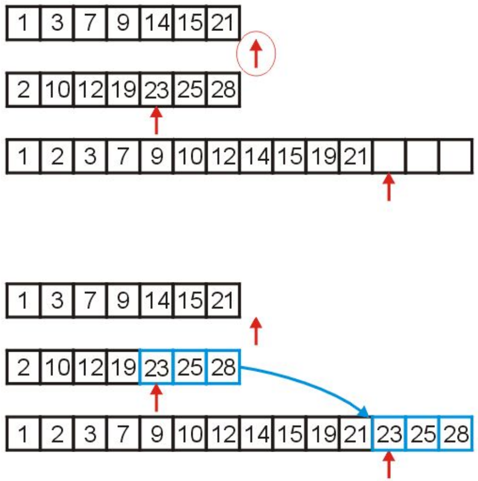
Ordenación

MergeSort



Ordenación
MergeSort

- Continuamos hasta que hayamos pasado fuera de los límites de uno de los arreglos.
- Luego, nosotros simplemente copiamos el resto de las entradas del arreglo que todavía no finalizamos de analizar



Ordenación
MergeSort

Ejemplo del código MERGE (un poco diferente al visto en clases anteriores):

```
int in1 = 0, in2 = 0, out = 0; /* Indices */
while ( in1 < n1 && in2 < n2 ) {
   if ( array1[in1] < array2[in2] ) {
      arrayout[out] = array1[in1];
      ++in1;
   } else {
      arrayout[out] = array2[in2];
      ++in2;
   }
   ++out;
}
/* Copiamos el resto de las listas */
for (; in1 < n1; ++in1, ++out ) arrayout[out] = array1[in1];
for (; in2 < n2; ++in2, ++out ) arrayout[out] = array2[in2];
```

Ordenación
MergeSort

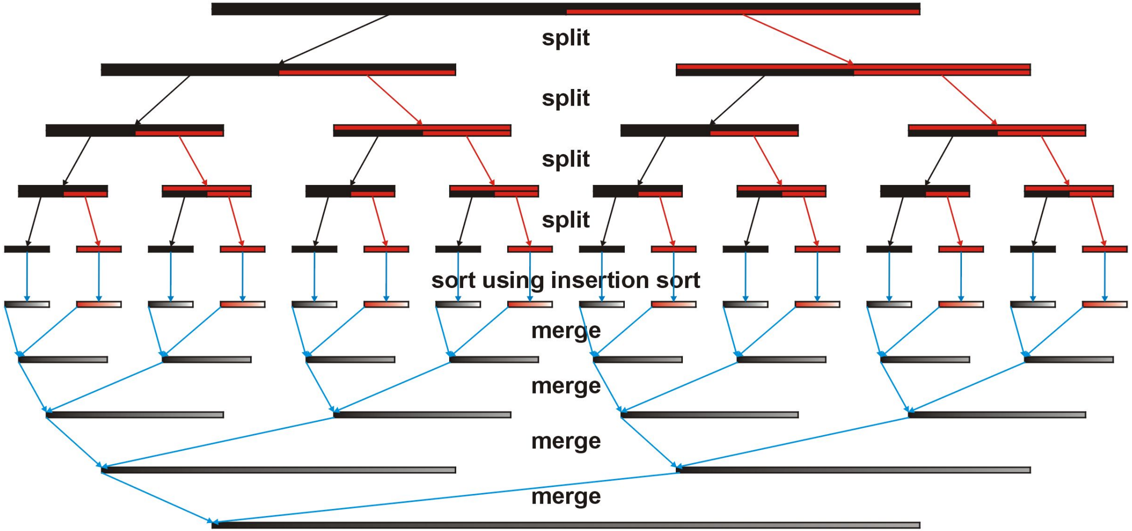
- El tiempo de ejecución del algoritmo de Merge o Mezcla :
  - Asumimos que la suma de la longitud de cada sub-lista es *n*
  - La sentencia *++out* solo puede ejecutarse *n* veces
  - Por tanto, el cuerpo de los ciclos se ejecutan exactamente *n* veces
  - Así, la operación de mezcla es realizado en $\Theta(n)$

Ordenación
MergeSort

- Nosotros partimos la lista en dos listas y las ordenamos. ¿Cómo hago esto?
  - Si el tamaño de la sub-lista es > 1, volvemos a utilizar la ordenación merge.
  - Si la sub-lista es de tamaño 1, no hago nada: esta ordenada.
  - *Aunque teóricamente lo de arriba es correcto, en la práctica se define un umbral a partir del cual podemos utilizar un algoritmo más simple como el de Inserción.*

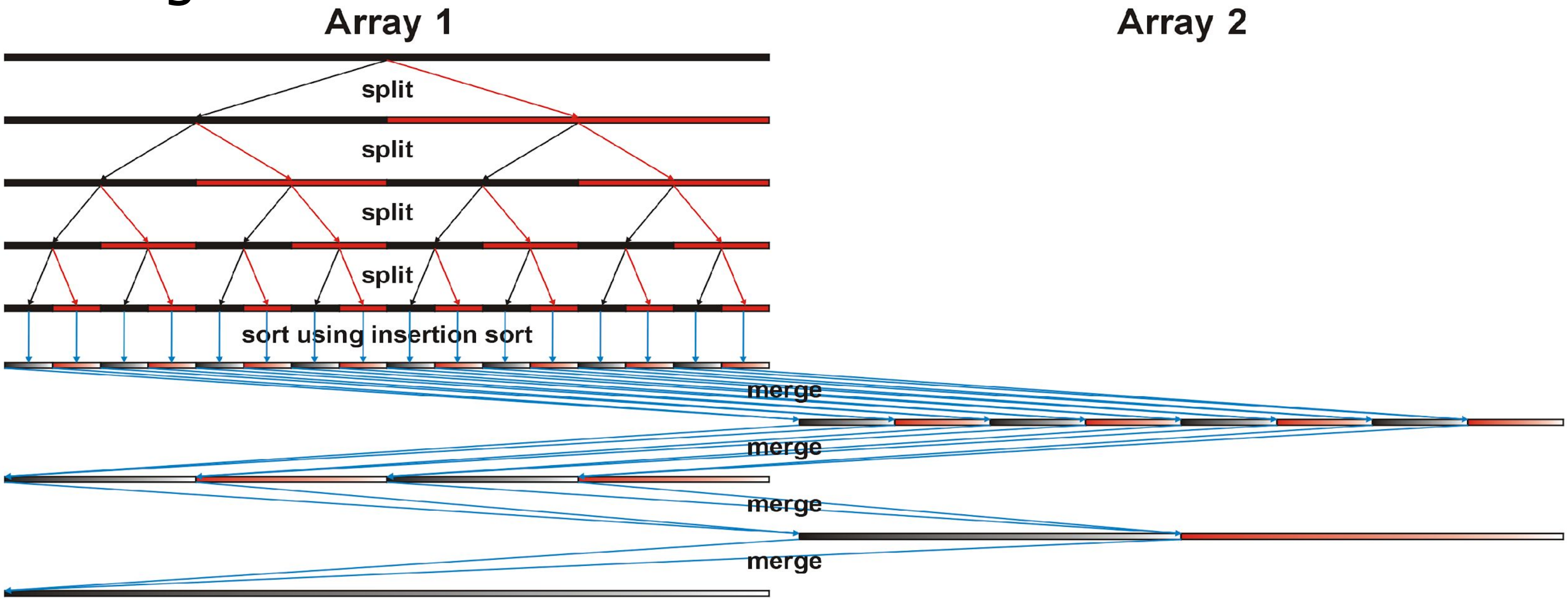
Ordenación
MergeSort

- Ejemplo gráfico:



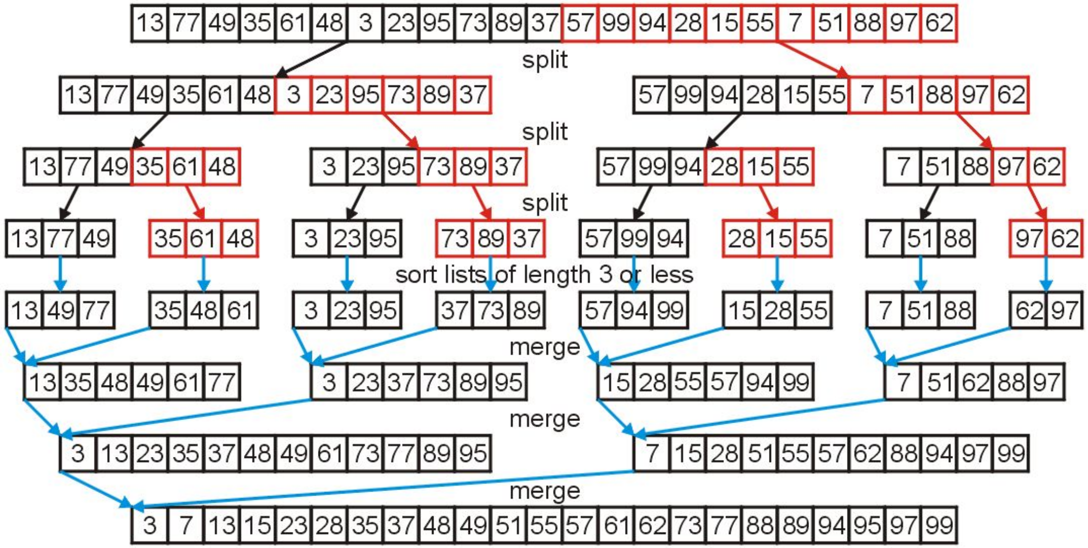
Ordenación
MergeSort

- Para realizar la operación de Merge se necesita un arreglo adicional.



Ordenación
MergeSort

- Ejemplo:
  - Ordenar la secuencia 13 77 49 35 61 48 3 23 95 73 89 37 57 99 94 28 15 55 7 51 88 97 62

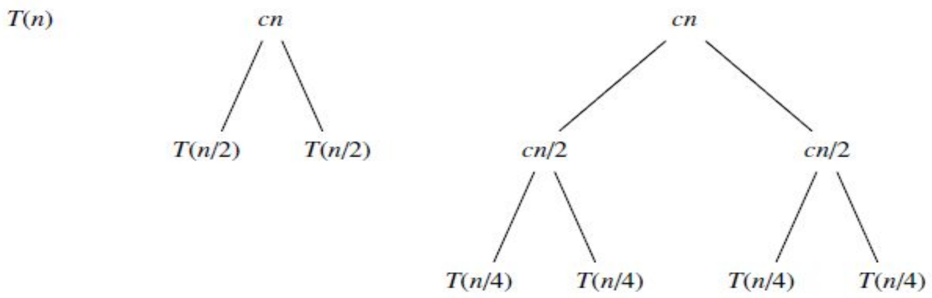


Ordenación
MergeSort

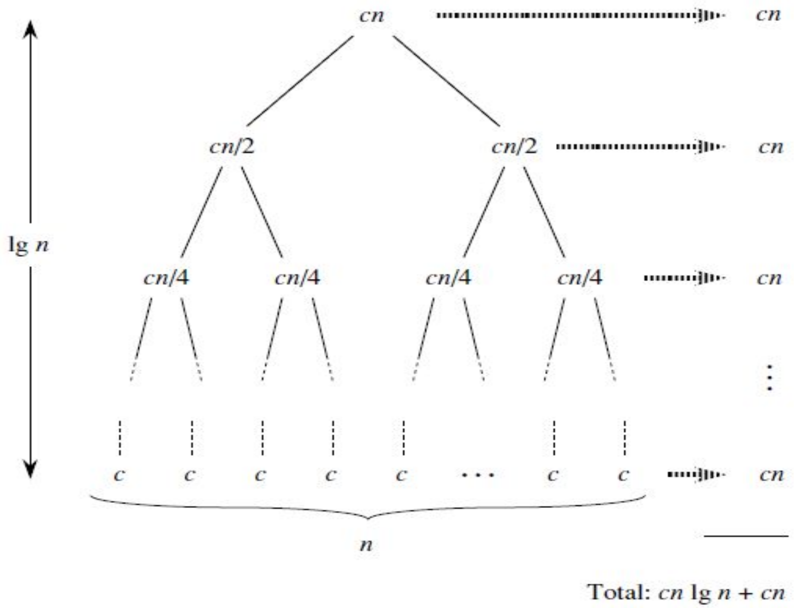
- Análisis de tiempo de ejecución
  - El tiempo requerido para ordenar la primera mitad
  - El tiempo requerido para ordenar la segunda mitad
  - El tiempo requerido para mezclar las dos listas

$$\mathrm{T}(n) = \begin{cases} \Theta(1) & n = 1 \\ 2\,\mathrm{T}\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

Ordenación
MergeSort



Θ (n log (n))



Ordenación
MergeSort

- Resumen del tiempo de ejecución de esta ordenación

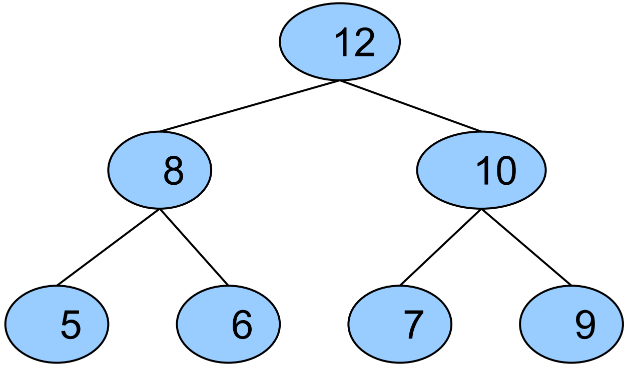
Caso	Cota	Comentario
Peor	$\Theta(n \lg(n))$	No hay
Medio	$\Theta(n \lg(n))$	
Mejor	$\Theta(n \lg(n))$	No hay

Ordenación
HeapSort

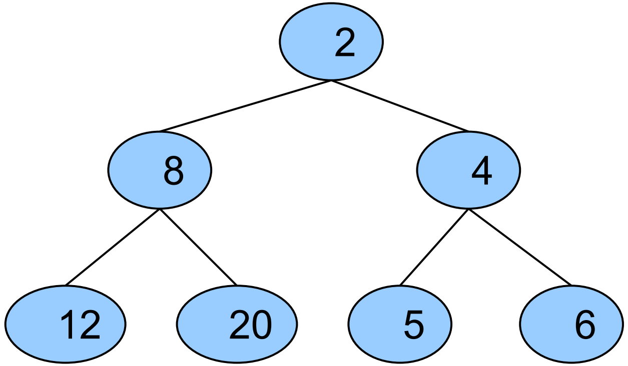
- Utiliza el concepto de Montículo Binario Maximal.
- Un montículo binario con dos condiciones:
  - Es completo (*¿Recuerdan que significa completo?*)
  - Esta parcialmente ordenado
- Normalmente se utiliza para priorizar la elección de un elemento en un árbol binario, ya sea el de menor prioridad (minimal) o el de mayor prioridad (maximal).

Ordenación
HeapSort

- Ejemplo de montículos



MAXIMAL

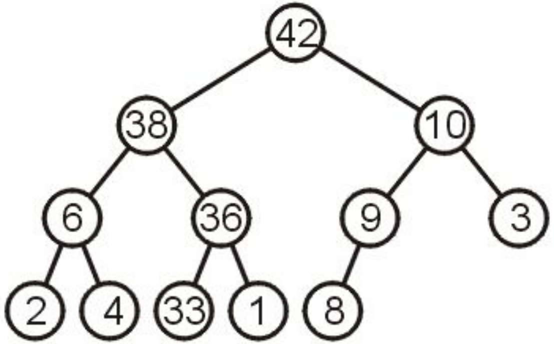


MINIMAL

Ordenación

HeapSort

- Consideramos que un arreglo representa un heap



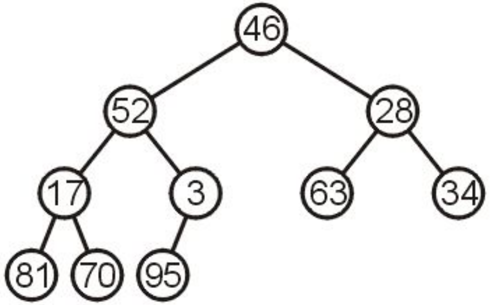
42	38	10	6	36	9	3	2	4	33	1	8

Ordenación
HeapSort

- Consideremos el siguiente arreglo desordenado

46	52	28	17	3	63	34	81	70	95

- El mismo representa el siguiente árbol binario completo

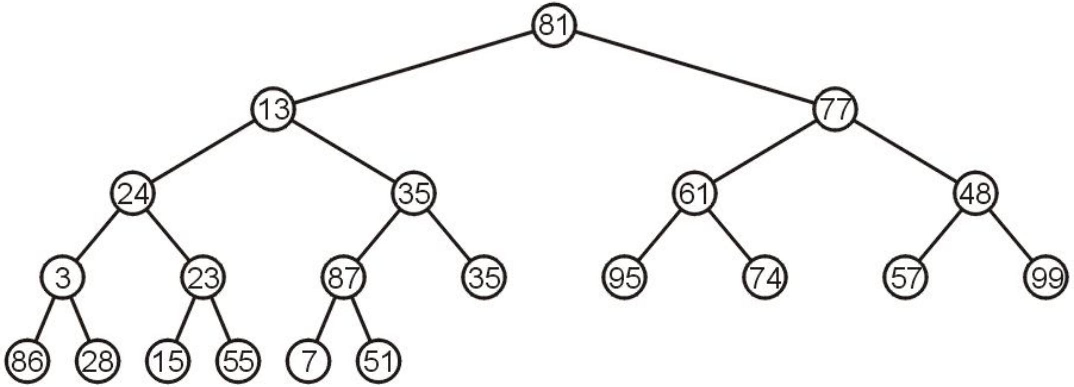


- Pero no es un max-heap, min-heap o un BST

Ordenación
HeapSort

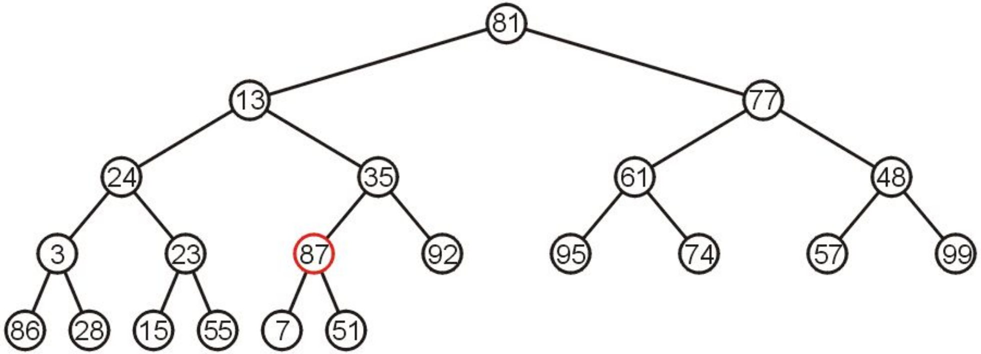
- Ahora vemos como convertir un arreglo en un max-HEAP.
- Ejemplo:

81	13	77	24	35	61	48	3	23	87	92	95	74	57	99	86	28	15	55	7	51



Ordenación
HeapSort

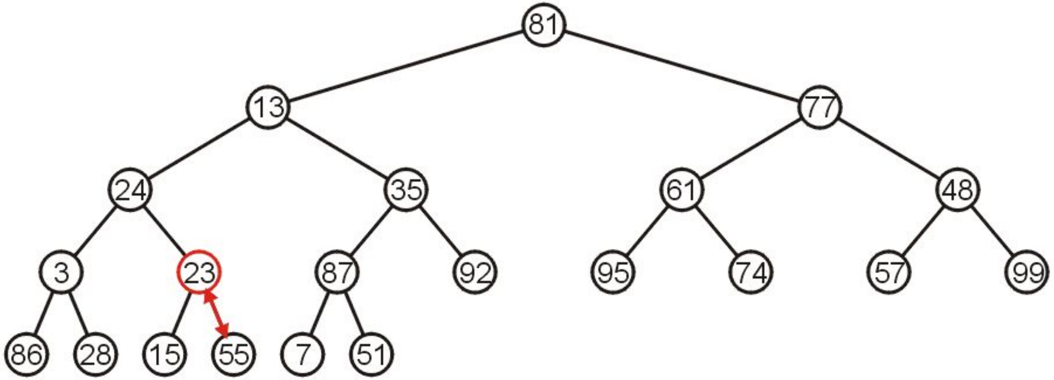
- Vemos que no es un max-HEAP aunque podemos considerar que las hojas son max-HEAP y también el sub-árbol con 87 como raíz es un max-HEAP



Ordenación

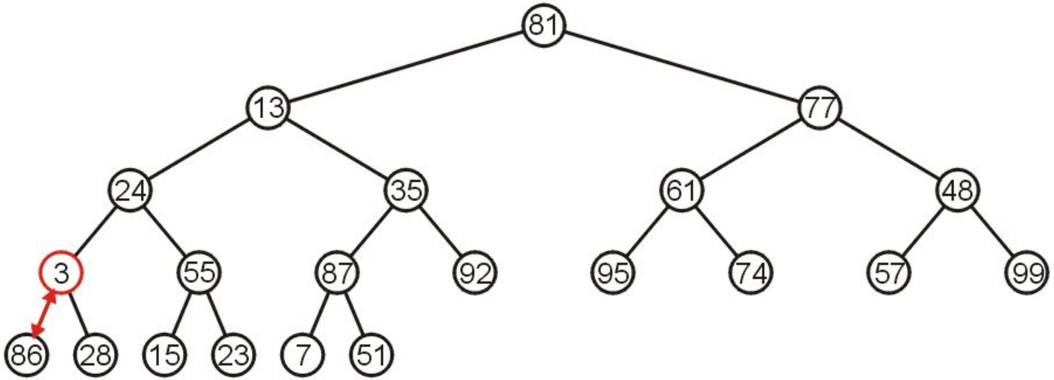
HeapSort

- El sub-arbol con raíz 23 no es un max-HEAP, pero si intercambiamos con el 55 creamos un max-HEAP



Ordenación
HeapSort

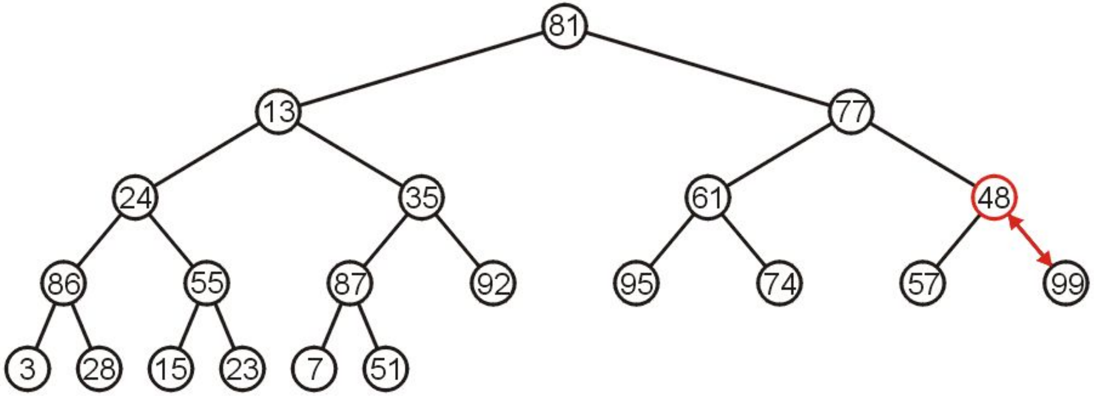
- Lo mismo sucede con el sub-arbol con 3 como raíz



Ordenación

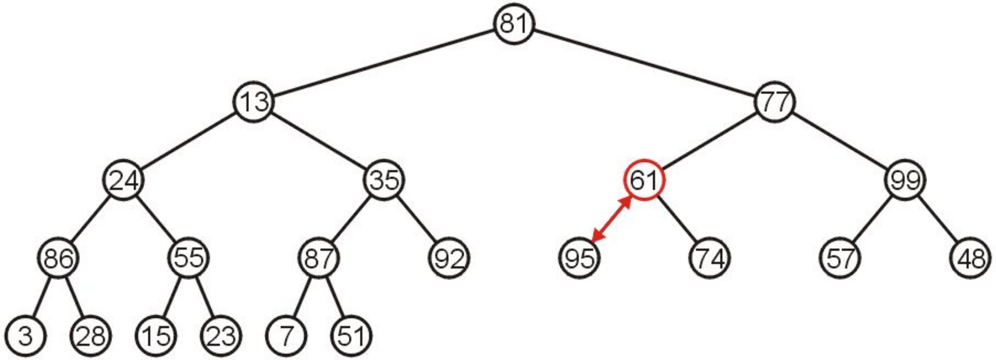
HeapSort

- Ahora subimos de nivel y analizamos el sub-árbol con raíz 48, lo convertimos en max-HEAP intercambiándolo con el 99



Ordenación
HeapSort

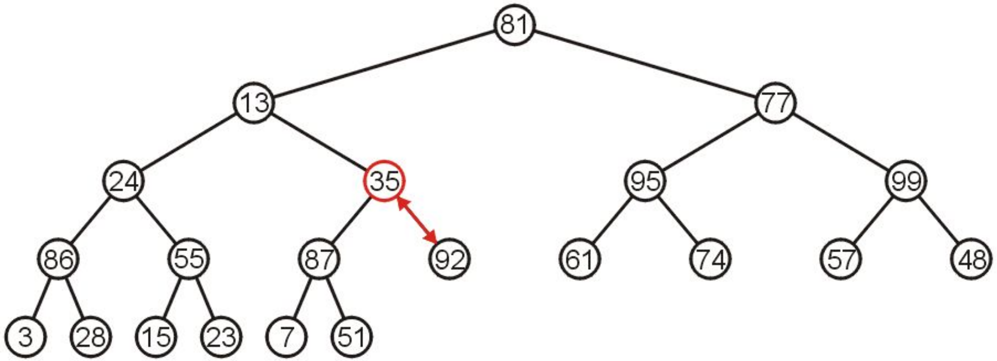
- Lo mismo ocurre con el 61



Ordenación

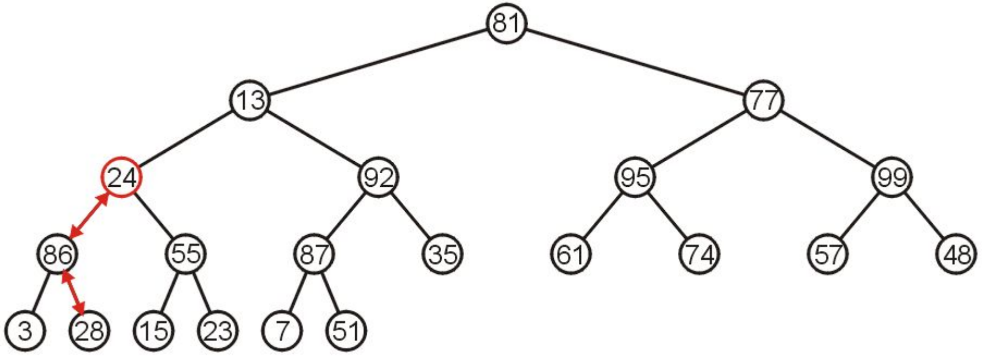
HeapSort

- Continuamos con el 35



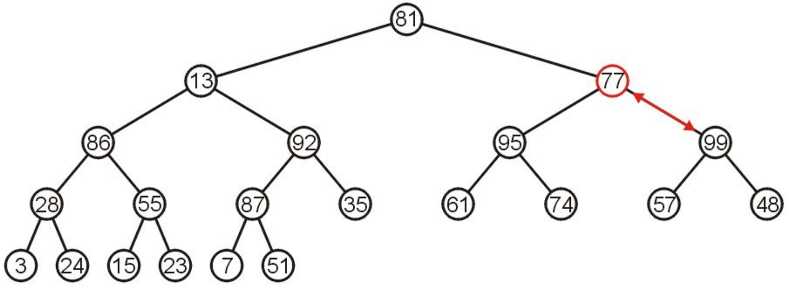
Ordenación
HeapSort

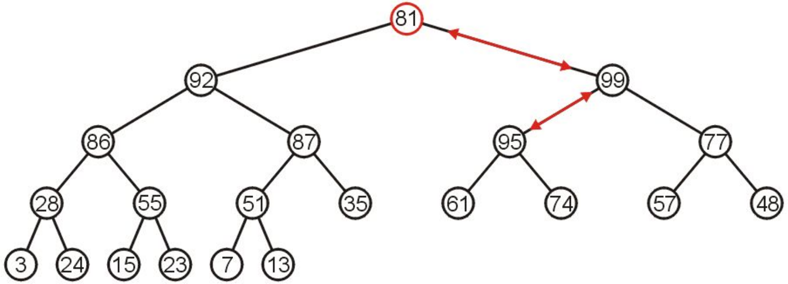
- Ahora analizamos el 24 y vemos que debemos intercambiar con el 86 y luego intercambiando con el 28. Operación de "*empujar*", "*hundir*" o *siftdown*

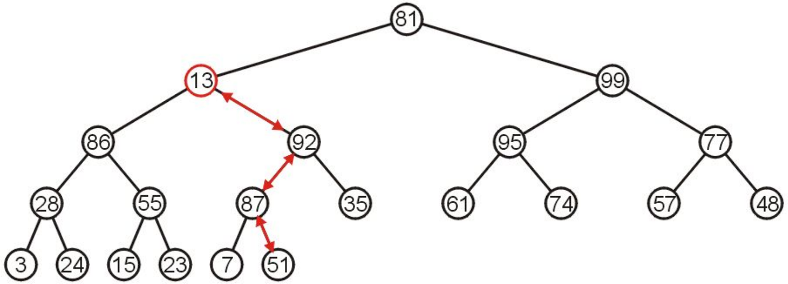


Ordenación

HeapSort

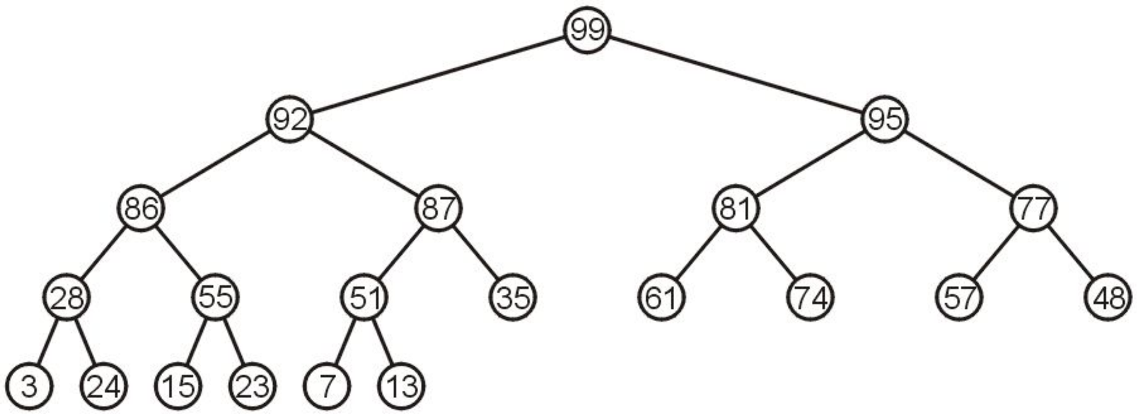






Ordenación
HeapSort

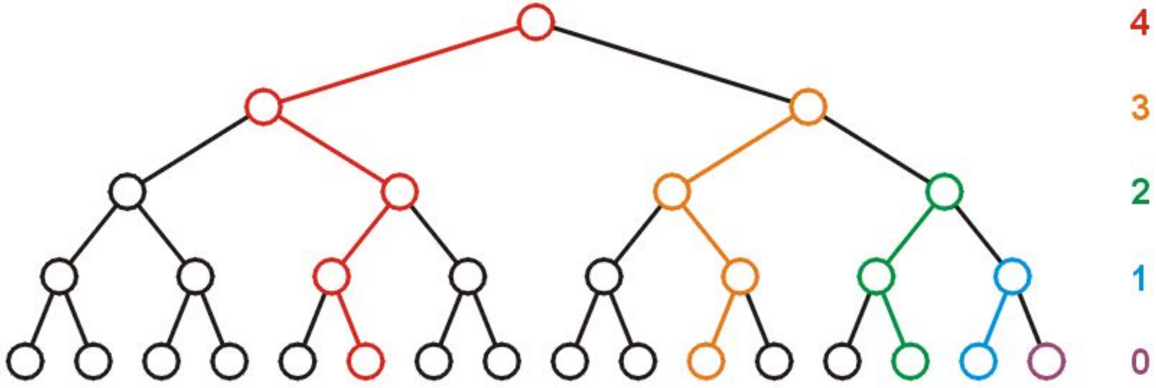
- Y finalmente tenemos construido nuestro
- max-HEAP



Ordenación

HeapSort

- La operación de construcción de un montículo maximal es O(*n*).



Ordenación
HeapSort

- En el nivel k, hay 2^k nodos, y por tanto en el peor caso todos los nodos deben ser "empujados" *h-k* niveles, requiriendo un total de $2^k(h-k)$ intercambios.
- Entonces tenemos la suma y su expresión en su forma cerrada.

$$\sum_{k=0}^{h} 2^k (h-k) = \left(2^{h+1} - 1\right) - (h+1)$$

Queda como ejercicio la demostración de la forma cerrada
Ver Libro de [Shaffer2013] Pag. 466, ejemplo 14.4

Ordenación
HeapSort

- Un árbol binario perfecto es $n = 2^{h+1} - 1$
- y $h+1 = log_2 (n+1)$, por tanto

$$\sum_{k=0}^{h} 2^k (h-k) = n - \log_2(n+1)$$

- Finalmente el tiempo esta en $O(n)$

Ordenación
Heapsort

- Algoritmo p/ convertir un arreglo en un heap

MAX-HEAPIFY(A, i, n)
  $l \leftarrow$ LEFT(i)
  $r \leftarrow$ RIGHT(i)
  if $l \leq n$ and $A[l] > A[i]$ **then**
    $largest \leftarrow l$
  else
    $largest \leftarrow i$
  if $r \leq n$ and $A[r] > A[largest]$ **then**
    $largest \leftarrow r$

  if $largest \neq i$ **then**
    exchange $A[i] \leftrightarrow A[largest]$
    MAX-HEAPIFY$(A, largest, n)$

LEFT(i)
  return 2i

RIGHT(i)
  return (2i + 1)

Ordenación
Heapsort

- Algoritmo para construir un montículo (maximal)

BUILD-MAX-HEAP(A, n)
for $i \leftarrow n/2$ **downto** 1 **do** MAX-HEAPIFY(A, i, n)

La ecuación de recurrencia de MAX-HEAPIFY es

$T(n) <= T(2n/3) + O(1)$ ***(Averigue porqué es así)***

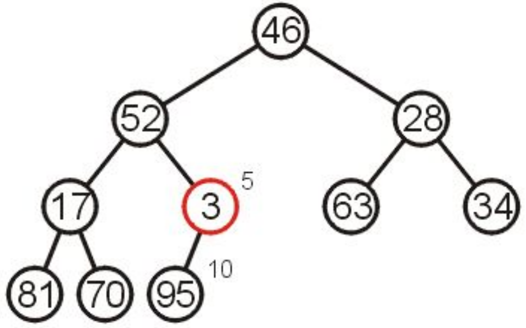
Según el teorema maestro (caso 2) => $T(n) = O(\log n)$

Ordenación
HeapSort

- Proceso de Ordenación, ejemplo.

46	52	28	17	3	63	34	81	70	95

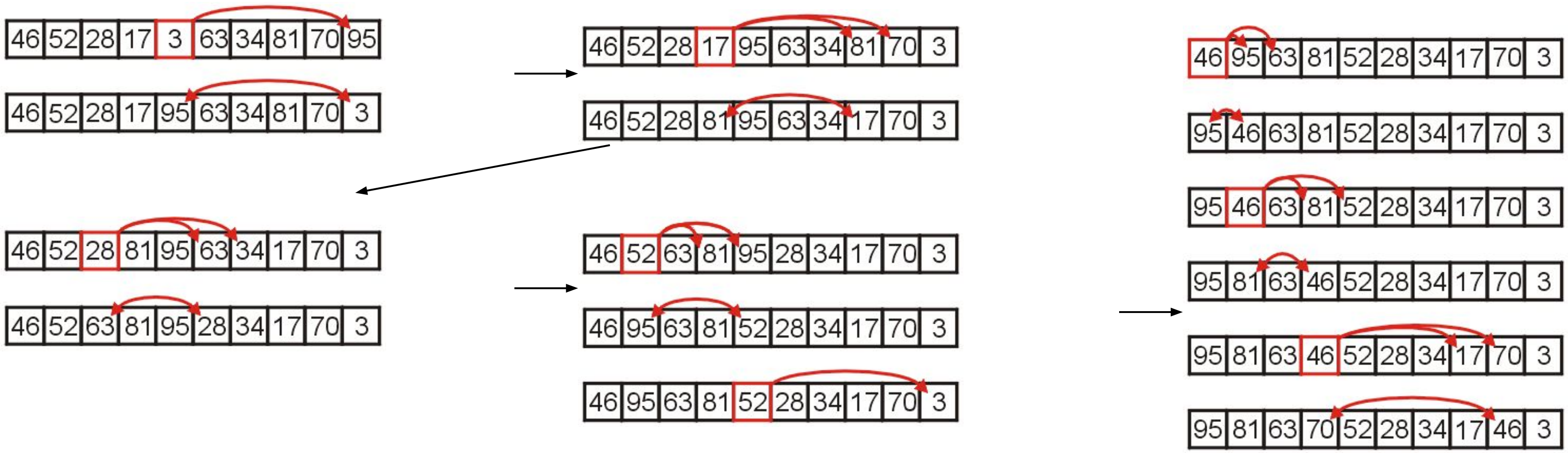
- Ninguna de las hojas necesitan ser “empujadas”, por tanto empezamos en el primer nodo “interno” en la posición n/2



- Esto es n = 10, empezamos en 5.

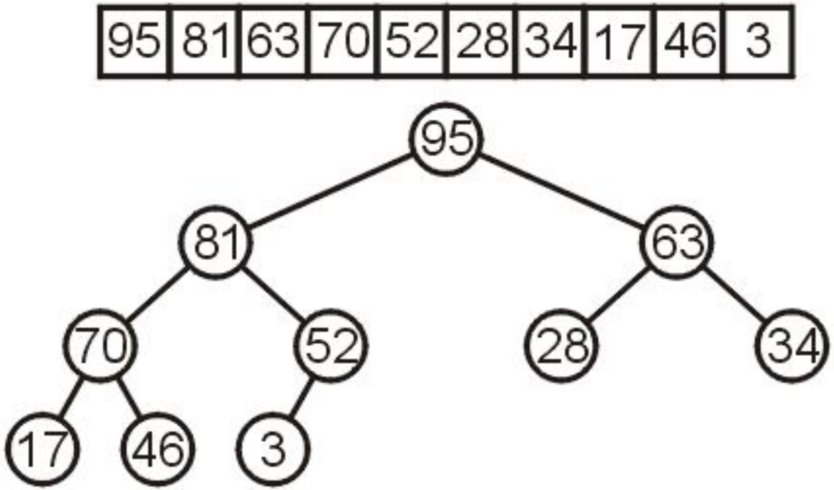
Ordenación HeapSort

- Convertimos en max-HEAP



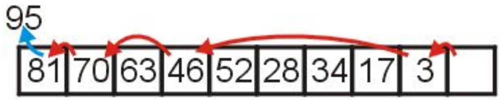
Ordenación

HeapSort

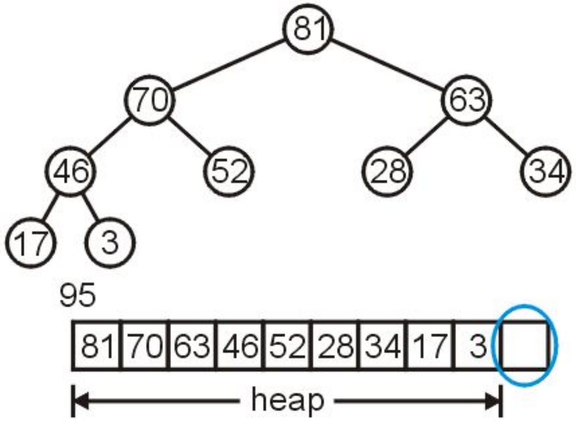


Ordenación
HeapSort

- Suponga ahora que decolamos el máximo elemento de este HEAP

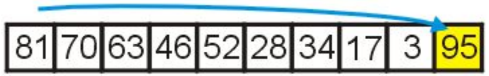


- Este deja un hueco al final del arreglo

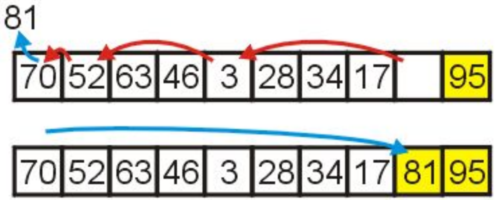


Ordenación
HeapSort

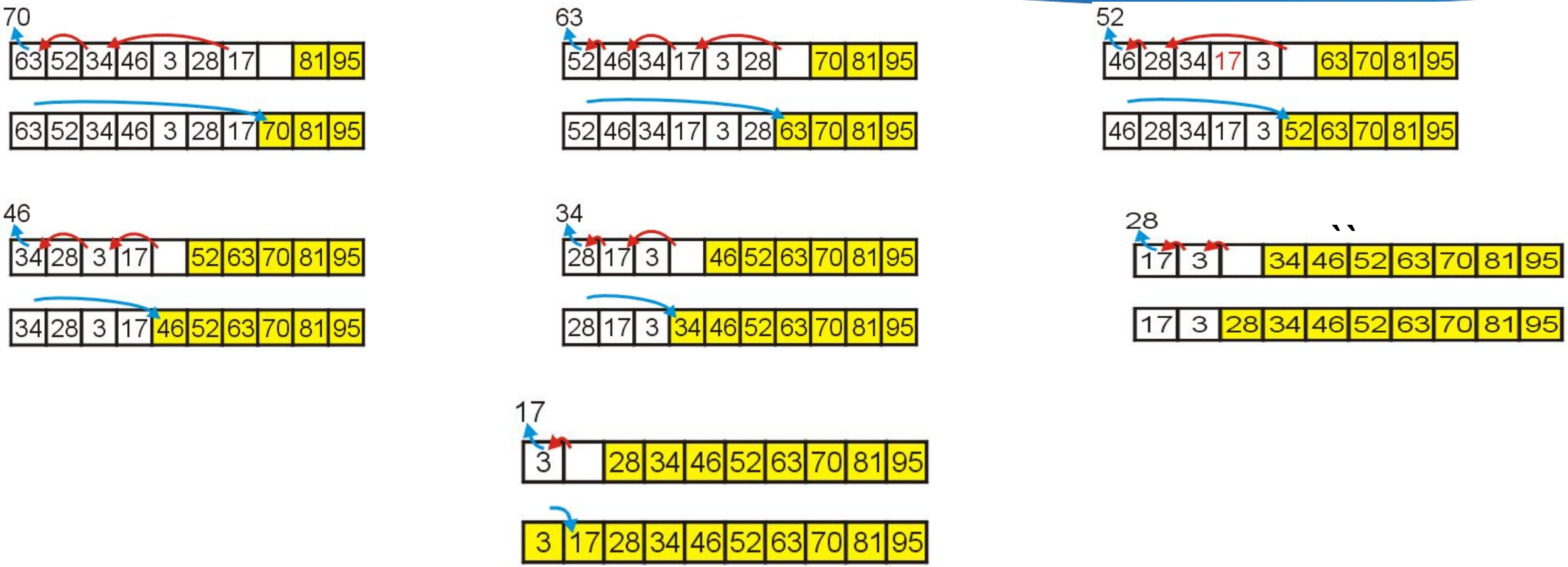
- Colocamos al final del arreglo



- Repetimos el proceso, decolamos el máximo e insertamos al final del arreglo



Ordenación
HeapSort



Finalmente queda el arreglo ordenado

Ordenación
HeapSort

- Algoritmo de ordenación

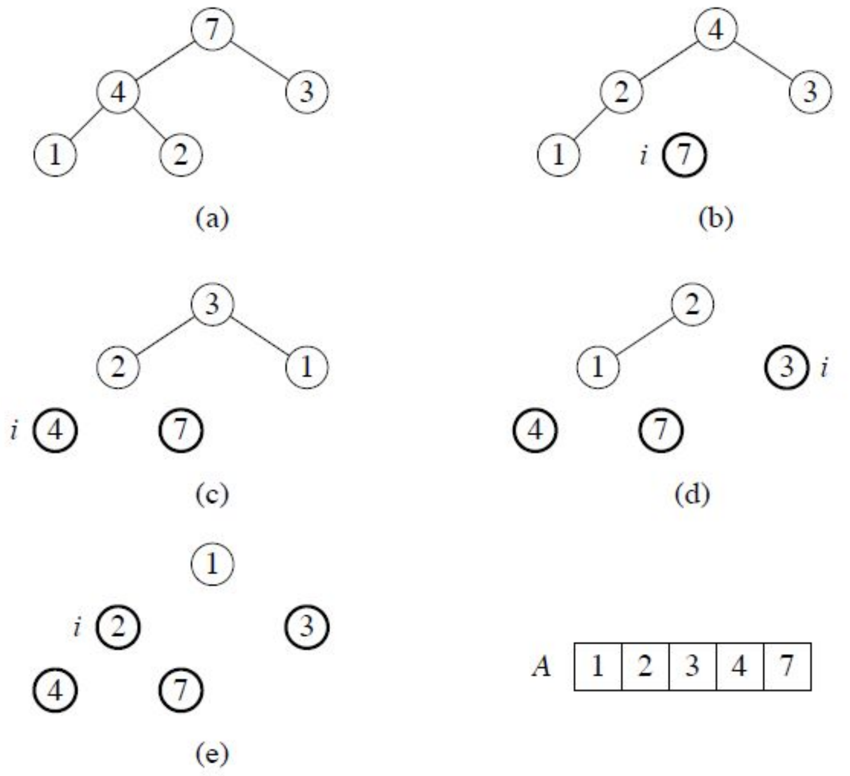
HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)

for $i \leftarrow n$ **downto** 2 **do**
exchange $A[1] \leftrightarrow A[i]$
MAX-HEAPIFY$(A, 1, i - 1)$

Ordenación

Heapsort



Ordenación
HeapSort

- El proceso de construir el Heap lleva *O(n)*
- Decolar n items de un HEAP de tamaño n, esta en *O (n log n)*
- Iterar *n-1* veces
- *I*ntercambiar esta en *O(1)*
- MAX-HEAPIFY esta en *O(log n)*
- Por tanto el algoritmo esta en *O (n log n)*

Ordenación
HeapSort

- Tiempo de ejecución

Caso	Tiempo	Comentario
Peor	$\Theta(n \ln(n))$	No hay
Medio	$\Theta(n \ln(n))$	
Mejor	$\Theta(n \ln(n))$	No hay

Ordenación

- El algoritmo de *MergeSort* y *HeapSort* poseen cotas: $O(n \log n)$ y $\Omega(n \log n)$
- Por tanto son algoritmos basados en comparación asintóticamente óptimos. ***¿Por qué?***

Ordenación

- Algoritmo que no se basan en comparación
  - CountingSort
  - Radixsort
  - BucketSort o BinSort

Ordenación

Counting*Sort*

- Posibilidad de ordenar un conjunto de datos en O(n).
- ¡No se basa en comparación!
- Si los datos se encuentran en cierto rango 0..M-1 entonces es posible hacerlo. Por tanto se puede utilizar este método cuando el conjunto de datos no es arbitrario.

Ordenación
CountingSort

- Ejemplo, se tiene una lista de números de 0 a 31 valores posibles.

0 1 4 6 7 8 10 11 12 14 15
16 18 19 20 22 23 26 27 28 29 31

Ordenación
CountingSort

- Datos repetidos
  - Contar
  - Lista Enlazada
- Ejemplo: ordenar estos dígitos

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4
9 2 3 5 1 0 9 3 5 2 3 5 4 2 1 3

0	2
1	3
2	7
3	10
4	2
5	7
6	0
7	1
8	3
9	2

Ordenación

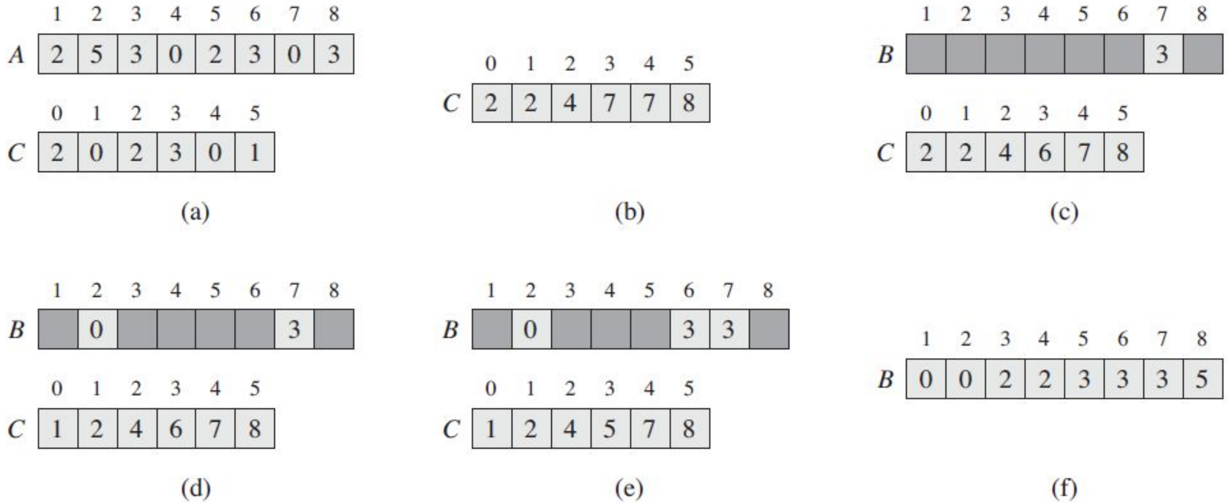
CountingSort

- **A[1..n]** arreglo de entrada.
  B[1..n] arreglo ordenado final.
  C[0..k] arreglo de trabajo temporal

- COUNTING-SORT(A,B,k)

```
for i = 0 to k do
    C[i] = 0
for j = 1 to length(A) do
    C[A[j]]= C[A[j]] + 1
      // C[i] tiene el nro. de elementos == i
for i= 1 to k do
    C[i] = C[i] + C[i-1]
      // C[i] tiene el nro. de elementos <= i
for j = length(A) to 1 do
    B[C[A[j]] = A[j]
    C[A[j]]    = C[A[j]]-1
```

Ordenación
CountingSort



a) Arreglos A y C iniciales
b) Arreglo C luego de la linea 6
c,d,e) Iteración 1,2 y 3 de las lineas 7-9 (arreglo de salida B y C)
f) Arreglo ordenado de salida B final

Ordenación

CountingSort

- Cada elemento es un entero en el rango 0-k.
- El tiempo para este algoritmo es $\Theta(n+k)$
- Es un algoritmo estable. Los elementos con el mismo valor aparecen en el mismo orden de salida y de entrada.
- **Ejercicio:** probar el algoritmo anterior. Dibujar los arreglos A, B y C.
  - `A= {2,5,3,0,2,3,0,3} y k = 5`

Ordenación
CountingSort

- Debe considerar que el número de ítems es comparable con el número posible de valores.
- Por ejemplo, si n = 20, y se tiene enteros de 1 a 1000000. ¿Es posible considerar esta ordenación? Probablemente no.

Caso	Tiempo	Comentario
Peor	$\mathbf{O}(n)$	No hay
Medio	$\mathbf{O}(n)$	
Mejor	$\mathbf{O}(n)$	No hay

Ordenación

RadixSort

- Suponga que queremos ordenar números de diez dígitos donde haya repetición.
- ¿Podemos usar CountingSort?
- Solo los contadores requerirían 10^{10} cubetas

Ordenación
RadixSort

- Considere el siguiente esquema
- Dado los números

16 31 99 59 27 90 10 26 21 60 18 57 17

- Si ordenamos primero en base al último dígito, tendríamos:

90 10 60 31 21 16 26 27 57 17 18 99 59

- Ahora ordenamos en base al primer dígito

10 16 17 18 21 26 27 31 57 59 60 90 99

¿Qué obtuvimos?

Ordenación
RadixSort

- Esquema:
  - Tratemos de ordenar los siguientes números decimales: 86 198 466 709 973 981 374 766 473 342

Ordenación

RadixSort

- Creamos un arreglo de 10 colas

0				
1				
2				
3				
4				
5				
6				
7				
8				
9				

Ordenación
RadixSort

- Encolamos de acuerdo al tercer dígito
  - 086 198 466 709 973 981 374 766 473 342

0				
1	981			
2	342			
3	973	473		
4	374			
5				
6	086	466	766	
7				
8	198			
9	709			

- Y decolamos: 981 342 973 473 374 086 466 766 198 709

Ordenación
RadixSort

- Encolamos de acuerdo al 2do. Dígito
  - 981 342 973 473 374 086 466 766 198 709

0	709			
1				
2				
3				
4	342			
5				
6	466	766		
7	973	473	374	
8	981	086		
9	198			

- Y decolamos: 709 342 466 766 973 473 374 981 086 198

Ordenación
RadixSort

- Encolamos de acuerdo al primer dígito
  - 709 342 466 766 973 473 374 981 086 198

0	086			
1	198			
2				
3	342	374		
4	466	473		
5				
6				
7	709	766		
8				
9	973	981		

- Y decolamos: 086 198 342 374 466 473 709 766 973 981

Ordenación
RadixSort

- Y finalmente tenemos la lista ordenada:

086 198 342 374 466 473 709 766 973 981

- ¿Qué se observa?

- Suponga que ahora queramos ordenar la siguiente secuencia de números binarios:

1111 11011 11001 10000 11010 101 11100 111 1011 10101

Ordenación
RadixSort

- Necesitamos 2 colas.
- Necesitamos encolar y decolar 5 veces

Ordenación RadixSort

- Algoritmo sencillo, asume elementos de *d*-dígitos

```
• RADIX-SORT(A,d)
    - for i = 1 to d do
        • Usar un algoritmo estable de ordenacion para ordenar
          A en el dígito i.
```

- Dado *n* números de *d*-dígitos donde cada dígito puede tomar hasta *k* valores. Radix-SORT ordena en un tiempo $\Theta(d\ (n+k))$.

Ordenación

RadixSort

- Tiempo de ejecución

Caso	Tiempo	Comentario
Peor	$\mathbf{O}(n)$	No hay
Medio	$\mathbf{O}(n)$	
Mejor	$\mathbf{O}(n)$	No hay

Ordenación

BucketSort

- Igual que CountingSort asume una característica en los datos
- Asume que la entrada es generada por un proceso randómico que distribuye los elementos uniformemente en un rango [0,1)
- La idea es dividir el intervalo [0,1) en subintervalos denominados buckets o cubetas
- Para producir la salida, simplemente ordenamos los números en cada cubeta y recorremos las cubetas en orden, listando los elementos

Ordenación

BucketSort

- Algoritmo (CLRS)
  - Asume que 0 <= A[i] < 1 y *n* buckets

BUCKET-SORT(A)

$n \leftarrow length[A]$

for $i \leftarrow 1$ **to** n

  do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$

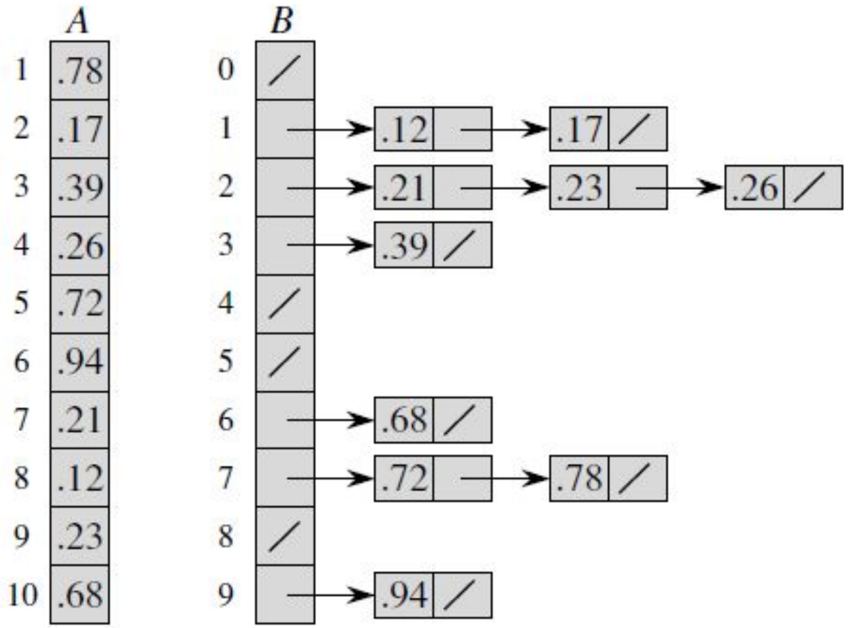
for $i \leftarrow 0$ **to** $n - 1$

  do sort list $B[i]$ with insertion sort

concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order

Ordenación
BucketSort

- Ejemplo con n=10



Ordenación
BucketSort

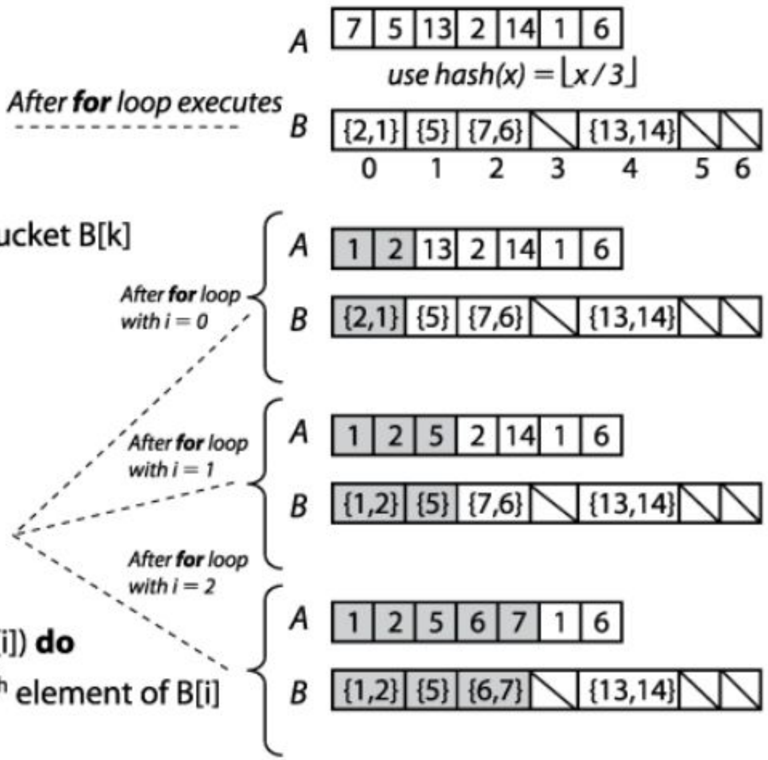
- Otro ejemplo
  - *n* = 2^m elementos. Cada elemento es un entero en el rango $[0,2^k)$ donde k >= m distribuido uniformemente.
  - Primera fase:
    - Colocamos los elementos en *n* cubetas
    - Cada *j-ésima* cubeta contiene los elementos con los primeros *m* digitos binarios corresponden a j. Si n = 2^{10}, la cubeta 3 contiene todos los elementos cuyos primeros 10 digitos binarios son 0000000011.
  - Segunda fase:
    - Cada cubeta es ordenada con algún algoritmo de ordenación
    - Concatenar cada cubeta

BucketSort

Ejemplo con hash

```
sort (A)
1.    create n buckets B
2.    for i = 0 to n − 1 do
3.       k = hash(A[i])
4.       add A[i] to the kth bucket B[k]
5.    extract (B, A)
end

extract (B, A)
1.    idx = 0
2.    for i = 0 to n − 1 do
3.       insertionSort (B)
4.       for m = 1 to size(B[i]) do
5.          A[idx++] = mth element of B[i]
end
```



Ordenación

BucketSort

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right)$$

$$E\left[n_i^2\right] = 2 - \frac{1}{n} < 2$$

La demostración en
CLRS2001 item 8.4

$$T(n) = \Theta(n) + n.O\left(2 - \frac{1}{n}\right) = \Theta(n)$$

El algoritmo de bucketSort se *espera* (tiempo promedio) que ordene en tiempo lineal.

Fuentes consultadas

- Mark Allen Weiss. *Estructura de datos en Java.* Pearson. 2013 (cap. 8 y 20)
- Clifford A. Shaffer. *Data Structures & Algorithm Analysis in Java*. Dover Publications. 2011 (cap. 7)
- T. Cormen, C. Leiserson, R. Rivest y C. Stein. *Introduction to algorithms*. Second edition. MIT Press. 2001. (cap: 6, 7 y 8).