Ayudantía 05

Heaps y Ordenamiento Lineal

Elías Ayaach, Alexander Infanta, Amelia González, Paula Grune, Joaquín Peralta

Heaps

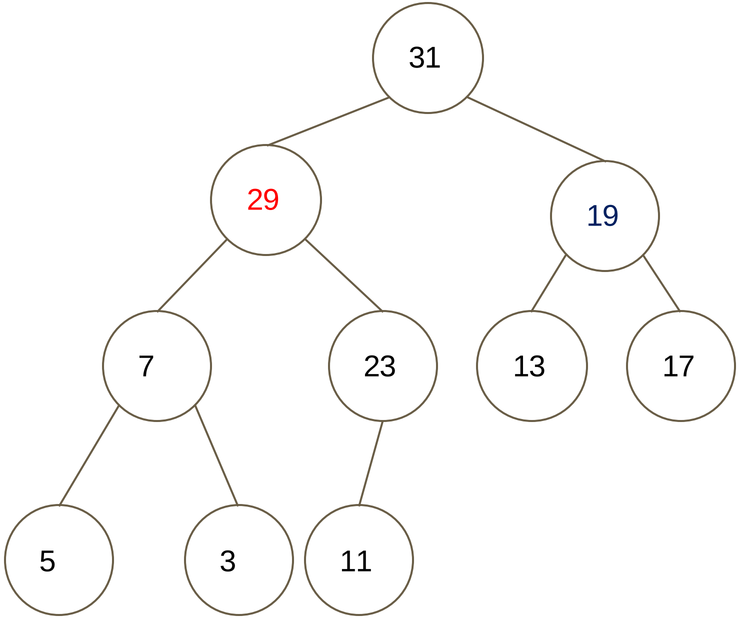
¿Qué es un Heap?

- Un heap es un árbol binario que nos permite mantener un **orden de prioridad** los elemento de un conjunto. La prioridad la podemos determinar según un MIN-Heap o un MAX-heap

- A medida que bajamos de nivel, los nodos hijos tendrán menor prioridad que el padre. **Entre hermanos no hay ninguna restricción**

- Para nuestro objetivo de extraer e insertar de forma eficiente, no necesitamos un orden estricto. **Basta tener un orden entre subsectores**

Ejemplo de Heap



valor	31	29	19	7	23	13	17	5	3	11						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

Al ir llenando el árbol por nivel ganamos:

- Minimizar la altura del árbol y compactar en el array
- Una relación entre padres y hijos:
  - El elemento **H[k]** es padre de **H[2k + 1] (left)** y **H[2k + 2] (right)**
  - El padre del elemento H[k] es **H[⌊(k − 1)/2⌋]**
- agrupar los niveles

valor	31	29	19	7	23	13	17	5	3	11						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
	nivel 0	nivel 1		nivel 2				nivel 3								

Recordamos que para extracción e inserción eficiente

1. Efectuamos la operación manteniendo un árbol binario casi-lleno
2. Restablecemos las propiedades de heap (acorde a la prioridad)

1) extracción eficiente

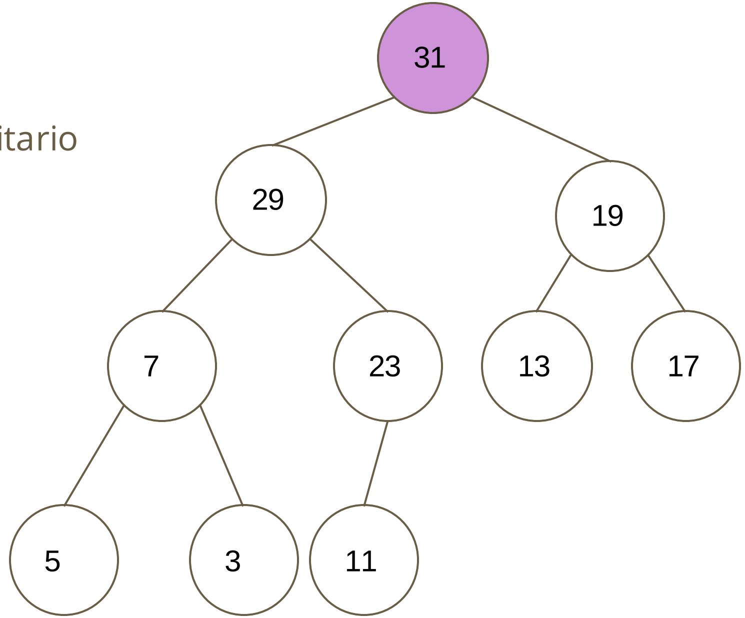
```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H, 0)
    return best
```

```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```

O(log(n))

1) extracción eficiente

Queremos sacar el elemento más prioritario

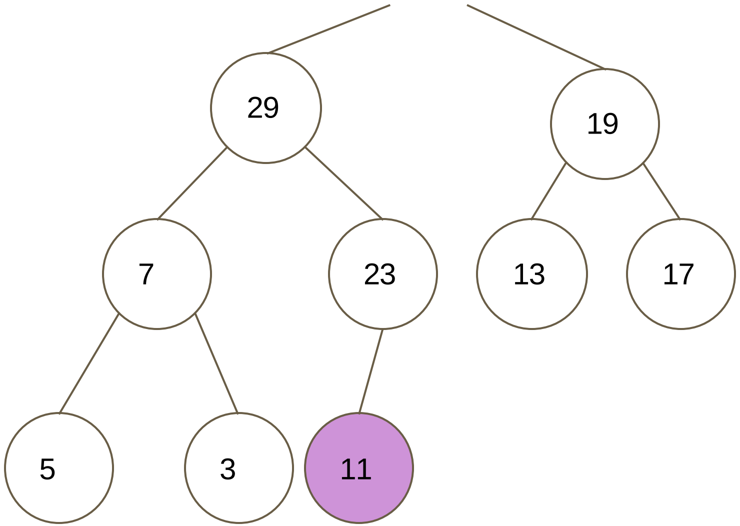


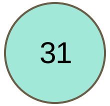
`Extract`(H):

- $i \leftarrow$ última celda no vacía de H
- $best \leftarrow H[0]$
- $H[0] \leftarrow H[i]$
- $H[i] \leftarrow \varnothing$
- `SiftDown`$(H, 0)$
- **return** $best$

valor	31	29	19	7	23	13	17	5	3	11						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

1) extracción eficiente





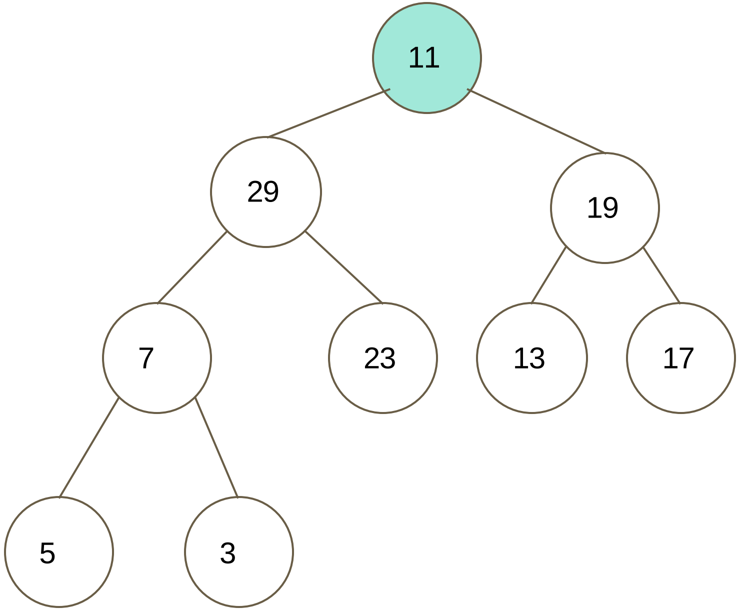
Extraemos

Extract(H):
- $i \leftarrow$ última celda no vacía de H
- $best \leftarrow H[0]$
- $H[0] \leftarrow H[i]$
- $H[i] \leftarrow \varnothing$
- SiftDown(H, 0)
- **return** $best$

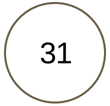
valor		29	19	7	23	13	17	5	3	11						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

1) extracción eficiente

```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```



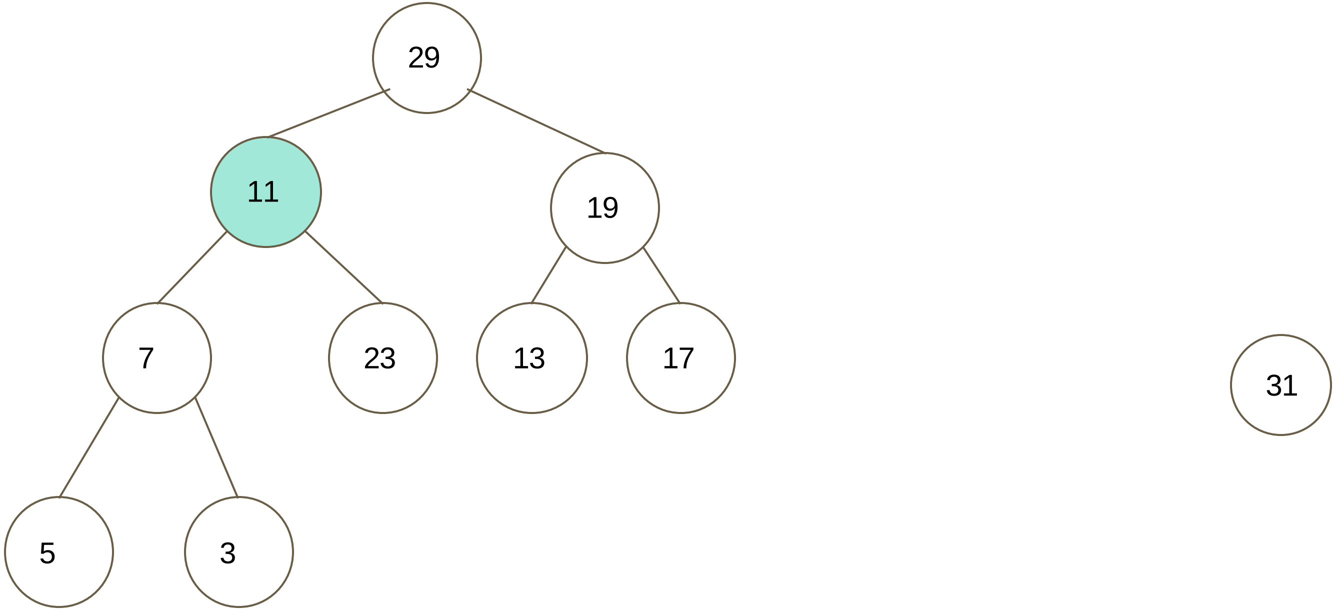
Empezamos con el ShiftDown para reacomodar



valor	11	29	19	7	23	13	17	5	3							…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

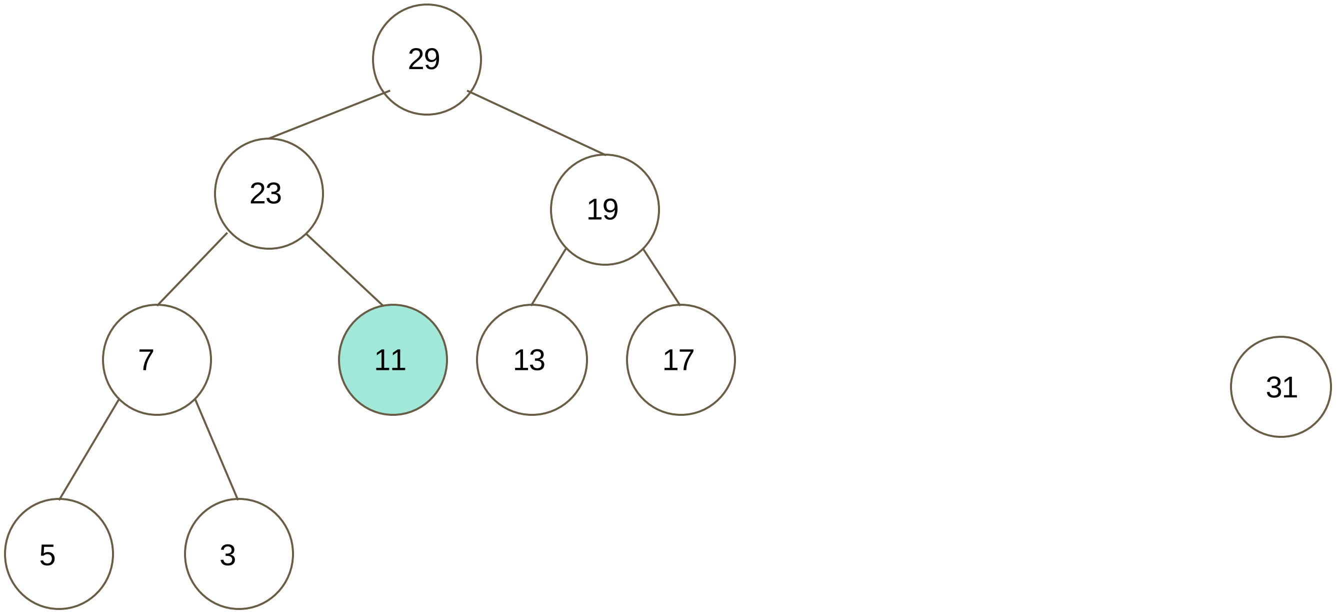
1) extracción eficiente

```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇆ H[i]
            SiftDown(H, j)
```



valor	29	11	19	7	23	13	17	5	3							…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

1) extracción eficiente



SiftDown(H, i):
  if i *tiene hijos* **:**
    $j \leftarrow$ hijo de i con mayor prioridad
    if $H[j] > H[i]$ **:**
      $H[j] \leftrightharpoons H[i]$
      SiftDown(H, j)

valor	29	23	19	7	11	13	17	5	3							…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

2) inserción eficiente

$\texttt{Insert}(H)$:

$i \leftarrow$ primera celda vacía de H

$H[i] \leftarrow e$

$\texttt{SiftUp}(H, i)$

$\texttt{SiftUp}(H, i)$:

if i *tiene padre* :

$j \leftarrow \lfloor i/2 \rfloor$

if $H[j] < H[i]$:

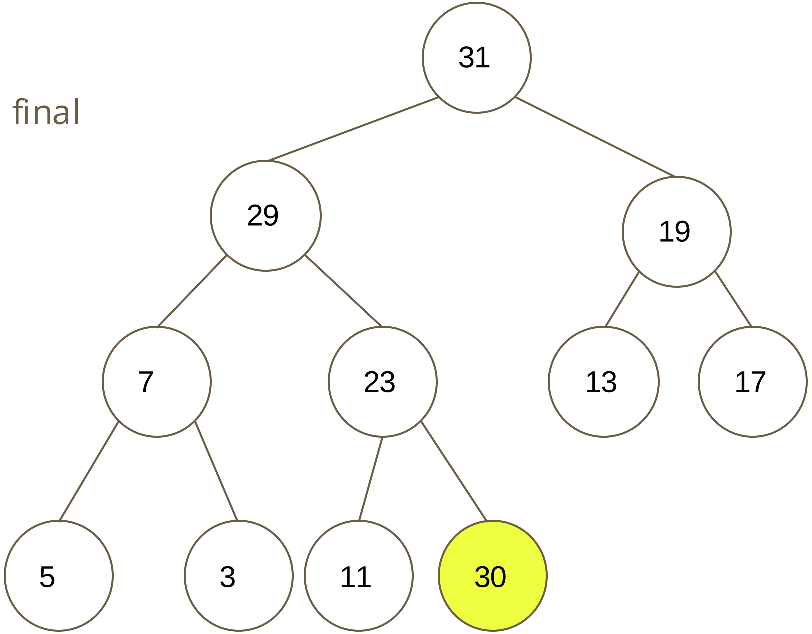
$H[j] \leftrightharpoons H[i]$

$\texttt{SiftUp}(H, j)$

O(log(n))

2) inserción eficiente

Insertamos, inicialmente, al final del array al nuevo nodo




```
Insert(H):
    i ← primera celda vacía de H
    H[i] ← e
    SiftUp(H, i)
```

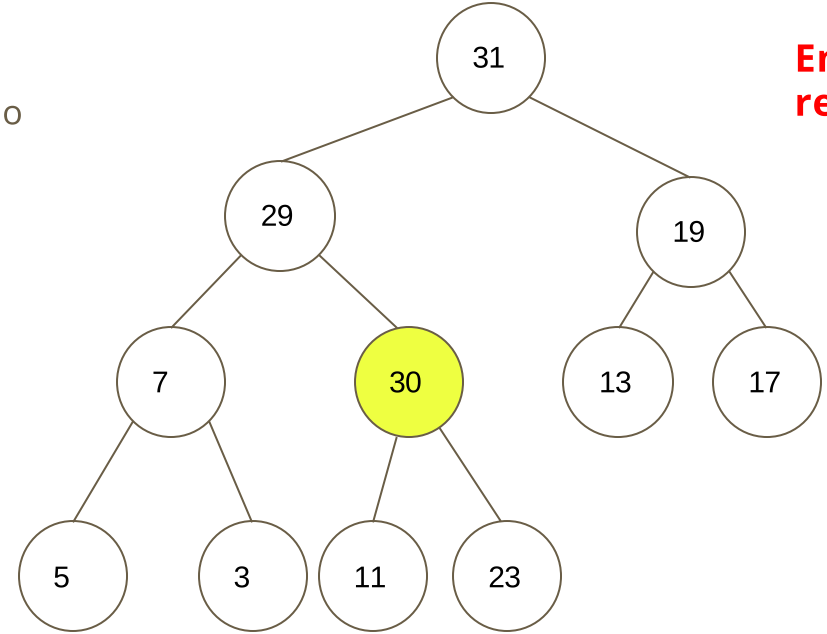
valor	31	29	19	7	23	13	17	5	3	11	30					…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

2) inserción eficiente

Empezamos a subir el nodo

Empezamos con el ShiftUp para reacomodar

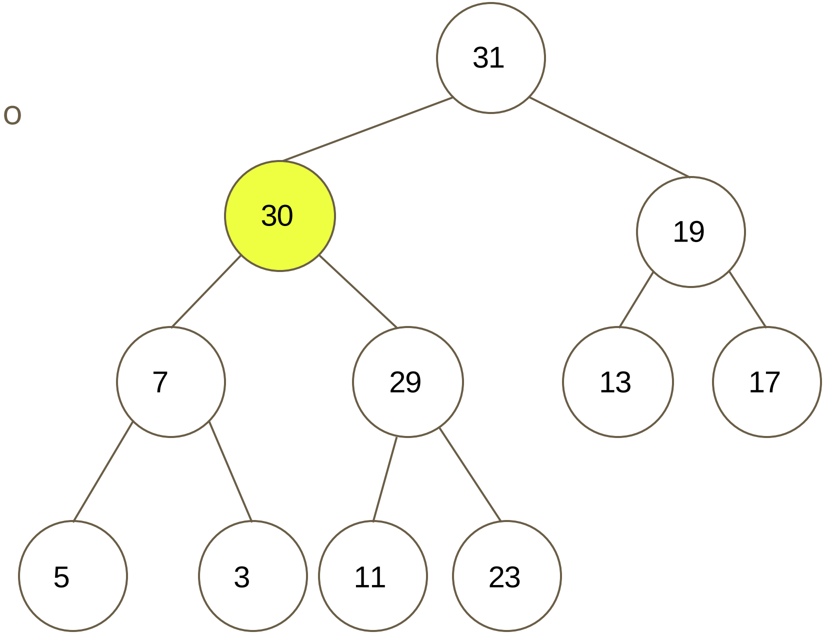
SiftUp(H, i):
    if i *tiene padre* :
        $j \leftarrow \lfloor i/2 \rfloor$
        if $H[j] < H[i]$:
            $H[j] \leftrightharpoons H[i]$
            SiftUp(H, j)



valor	31	29	19	7	30	13	17	5	3	11	23					…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

2) inserción eficiente

Empezamos a subir el nodo



$\texttt{SiftUp}(H, i)$:
  if i *tiene padre* :
    $j \leftarrow \lfloor i/2 \rfloor$
    if $H[j] < H[i]$:
      $H[j] \leftrightharpoons H[i]$
      $\texttt{SiftUp}(H, j)$

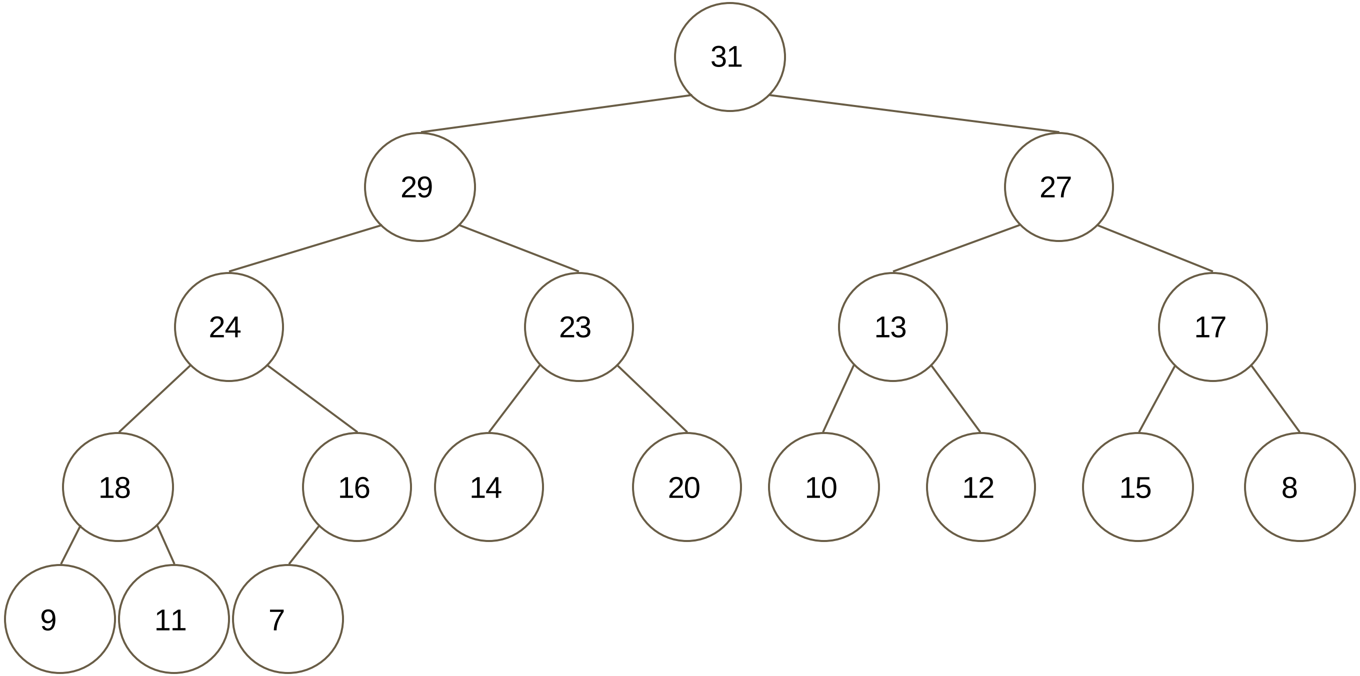
valor	31	30	19	7	29	13	17	5	3	11	23					…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

Lista ligada vs Array: ¿Qué ganamos?

	Con una lista ligada	Con un array
Extracción del elemento más prioritario	O(n)	O(1)
Insertar manteniendo orden	O(n)	O(log(n))

¿Y la actualización de valores?

Escribe un algoritmo que ordene el Heap en caso de que se actualice el valor del nodo i



valor	31	29	27	24	23	13	17	18	16	14	20	10	12	15	8	9	11	7		
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

¿Qué pasa si actualizamos el valor?

Tres casos:

- El nodo se actualiza con un valor que no supera al padre ni queda por debajo de alguno de los hijos

  → No pasa nada

- El nodo se actualiza con un valor que supera al del padre

  → Hay que subir el nodo

- El nodo se actualiza con un valor que lo hace quedar por debajo de alguno de los hijos

  → Hay que bajar el nodo y cambiarlo por ese hijo

input: Heap, índice del nodo a actualizar, nuevo valor del nodo

```
Update(H, i, v):
        H[i] = v
        if H tiene padre and H[i] > H[⌊(i−1)/2⌋]:
                ShiftUp(H, i)
    else if H tiene hijos and H tiene menor prioridad que alguno de ellos:
        ShiftDown(H, i)
```

Max Heap

prioridad mayor

prioridad menor

Max Heap

MAX-HEAPIFY(A, i)

```
l = 2i
r = 2i + 1
if l ≤ A.heap_size && A[l] > A[i] then
    largest = l
else
    largest = i
end if
if r ≤ A.heap_size && A[r] > A[largest] then
    largest = r
end if
if largest ≠ i then
    A[i] ↔ A[largest]
    MAX-HEAPIFY(A,largest)
end if
```

MAX-HEAP-EXTRACT(A)

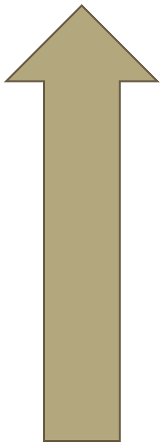
```
max = A[1]
A[1] = A[A.heap_size]
A.heap_size = A.heap_size − 1
MAX-HEAPIFY(A, A.heap_size)
return max
```

MAX-HEAP-INSERT(A, key)

```
A.heap_size = A.heap_size + 1
A[A.heap_size + 1] = key
while i > 1 && A[i//2] < A[i] do
    A[i//2] ↔ A[i]
    i = i//2
end while
```

Min Heap

prioridad menor



prioridad mayor

Min Heap

```
MIN-HEAPIFY(A, i)
l = 2i
r = 2i + 1
if l ≤ A.heap_size && A[l] < A[i] then
    smallest = l
else
    smallest = i
end if
if r ≤ A.heap_size && A[r] < A[smallest] then
    smallest = r
end if
if smallest ≠ i then
    A[i] ↔ A[smallest]
    MIN-HEAPIFY(A, smallest)
end if
```

```
MIN-HEAP-EXTRACT(A, i)
min = A[1]
A[1] = A[A.heap_size]
A.heap_size = A.heap_size − 1
MAX-HEAPIFY(A, A.heap_size)
return min
```

```
MIN-HEAP-INSERT(A, key)
A.heap_size = A.heap_size + 1
A[A.heap_size + 1] = key
while i > 1 && A[i//2] > A[i] do
    A[i//2] ↔ A[i]
    i = i//2
end while
```

¿Cómo construyo un heap?

¡Usaremos BuildHeap(A) para construir nuestro heap!

Una forma ingeniosa de hacer un heap a partir de un arreglo es usar *SiftDown(H, i)* en algunos elementos del arreglo

input : arreglo $A[0 \ldots n-1]$

$\texttt{BuildHeap}(A)$:

  for $i = \lfloor n/2 \rfloor - 1 \ldots 0$: $\triangleright$ loop decreciente

    $\texttt{SiftDown}(A, i)$

Ojo: los elementos de A en los cuales **no** se llama directamente *SiftDown* son hojas del último nivel del árbol

¿Puedo aprovechar un Heap para ordenar?

Ordenando con Heaps

Sabemos que en un max-heap, la raíz es estrictamente mayor a todos los demás valores, por ende debe ser el último elemento del arreglo ordenado.

Si sabemos que el último elemento del arreglo luego del intercambio está ordenado, entonces:

- No queremos moverlo más
- **Reduciremos el tamaño del heap** para no tocarlo
- A esto le llamaremos **A.heap_size**

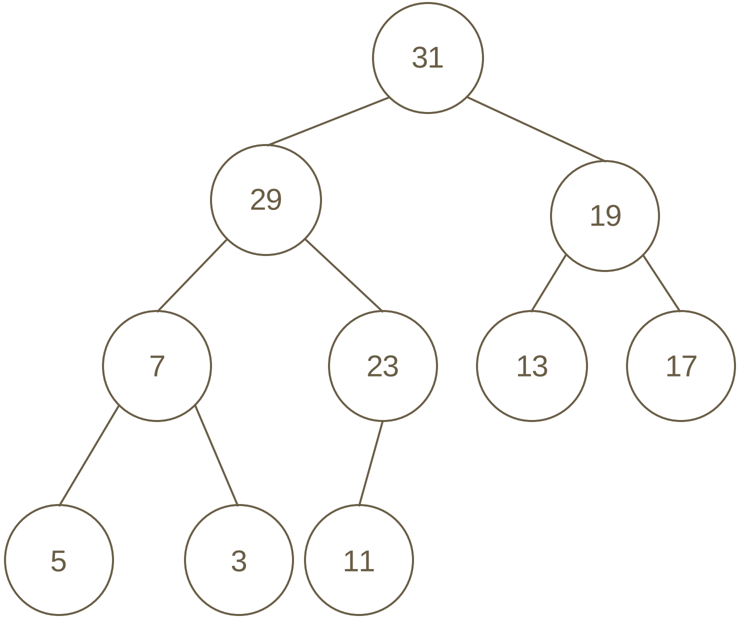
HeapSort

Entonces, considerando el A.heap_size acorde los últimos elementos vayan siendo ordenados, usaremos **HeapSort** que tiene complejidad **O(nlog(n))**

input : arreglo $A[0 \dots n-1]$

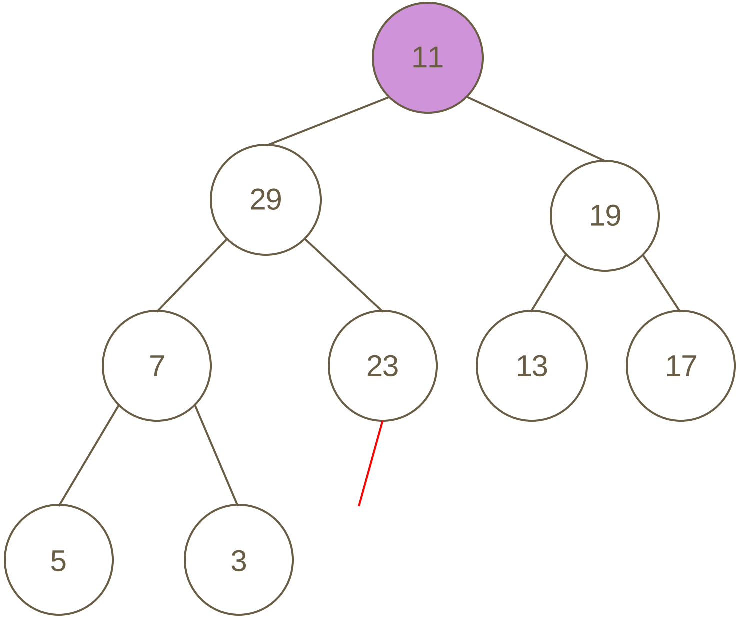
```
HeapSort(A):
    BuildHeap(A)
    for i = n − 1 ... 1 :          ▷ loop decreciente
        A[0] ⇋ A[i]
        A.heap_size = A.heap_size − 1
        ShiftDown(A, 0)
```

HeapSort en Acción



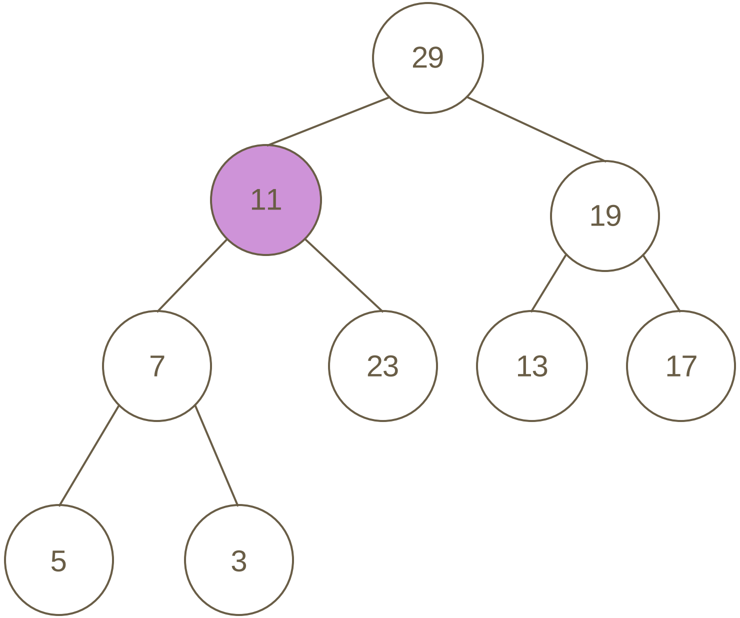
valor	31	29	19	7	23	13	17	5	3	11						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



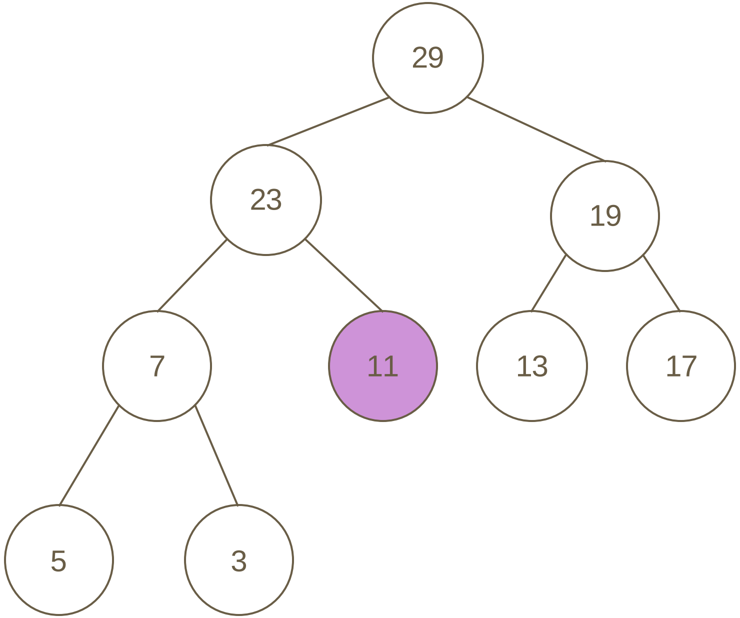
valor	11	29	19	7	23	13	17	5	3	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



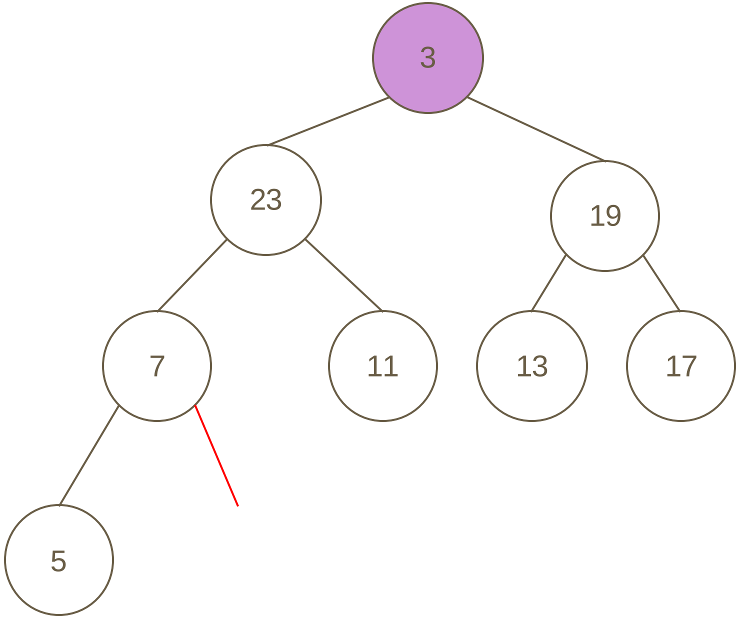
valor	29	11	19	7	23	13	17	5	3	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



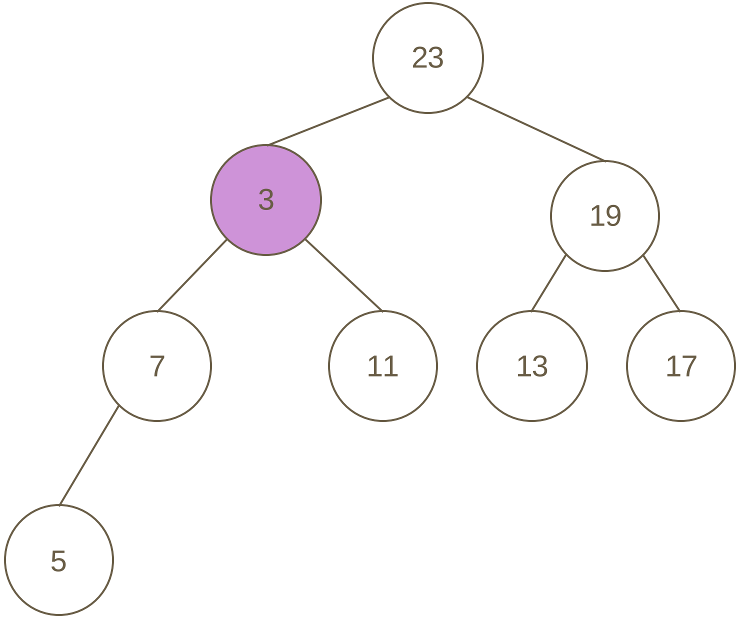
valor	29	23	19	7	11	13	17	5	3	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



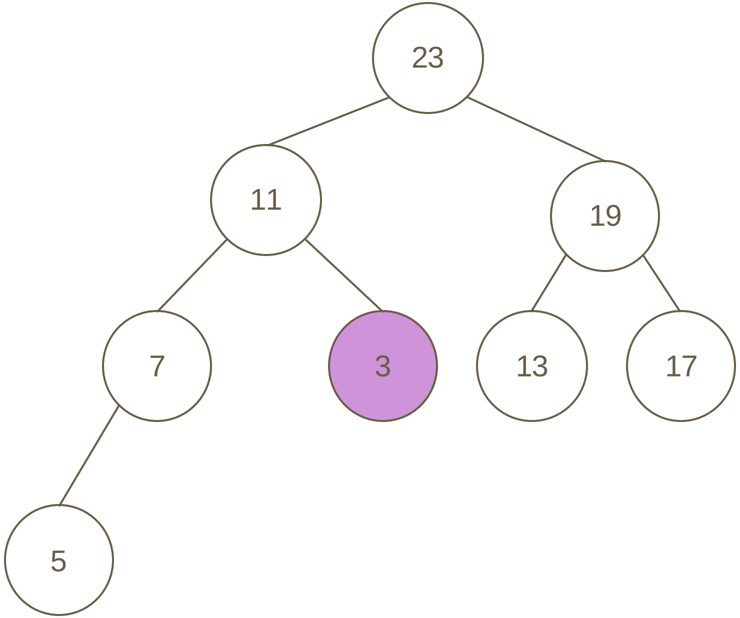
valor	3	23	19	7	11	13	17	5	29	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



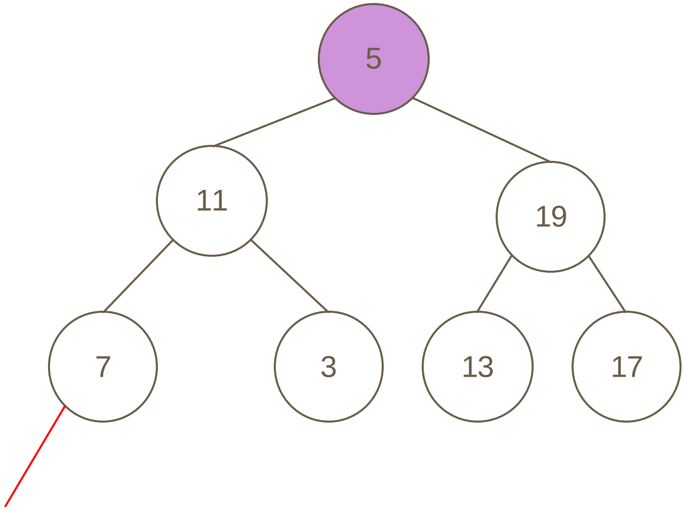
valor	23	3	19	7	11	13	17	5	29	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



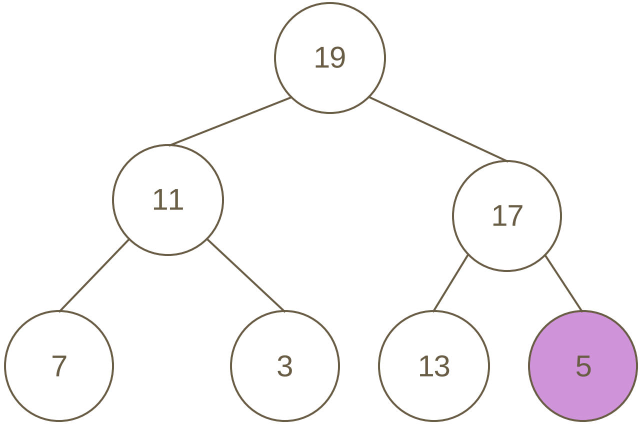
valor	23	11	19	7	3	13	17	5	29	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



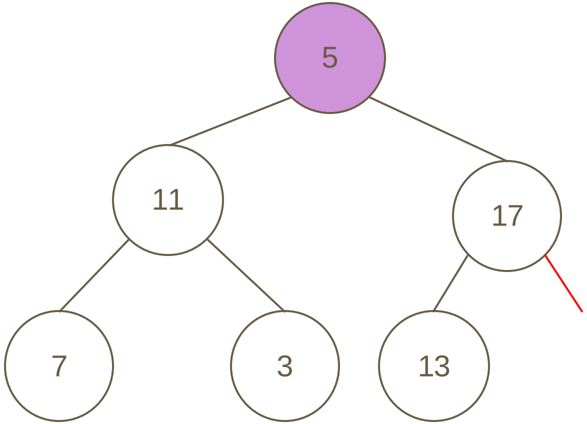
valor	5	11	19	7	3	13	17	23	29	31					…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



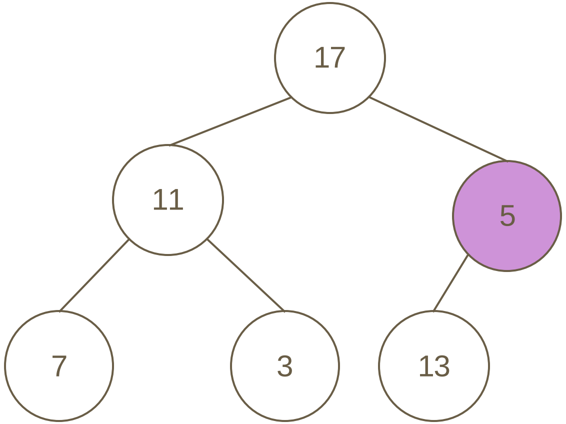
valor	19	11	17	7	3	13	5	23	29	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



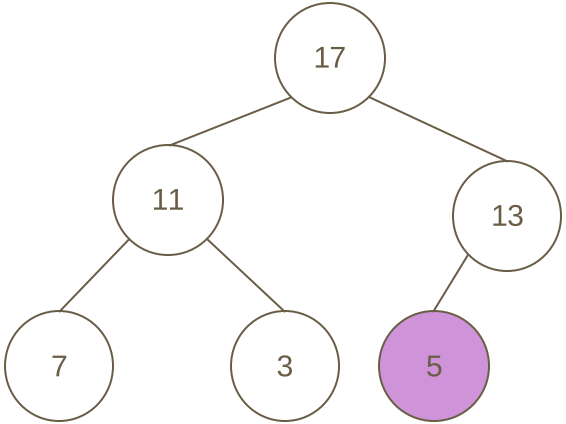
valor	5	11	17	7	3	13	19	23	29	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



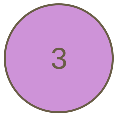
valor	17	11	5	7	3	13	19	23	29	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



valor	17	11	13	7	3	5	19	23	29	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

HeapSort en Acción



El proceso termina cuando queda solo un nodo en el heap y es el mínimo

valor	3	5	7	11	13	17	19	23	29	31						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

Ejercicio HeapSort

a) Escriba un algoritmo In-place, tal que ordene los elementos de un Max-Heap de menor a mayor en un mismo arreglo.

b) Calcule la complejidad de su algoritmo

Ejercicio HeapSort

a) **Escriba un algoritmo In-place, tal que ordene los elementos de un Max-Heap de menor a mayor en un mismo arreglo.**

```
SortingHeap(H):
    H: MaxHeap Inicializado
    i = len(H) - 1
    while(i >= 0):
        x = Extraer(H)
        H[i] = x
        i -= 1
```

Ejercicio HeapSort

b) Calcule la complejidad del algoritmo:

El algoritmo Extract() tiene una complejidad de O(logn) y en SortingHeap() lo llamamos n veces, por lo que la complejidad total es **O(n logn)**

Ordenación
Lineal

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

`CountingSort` (A, k):

- $B[0 \ldots n-1] \leftarrow$ arreglo vacío de n celdas
- $C[0 \ldots k] \leftarrow$ arreglo vacío de $k+1$ celdas
- **for** $i = 0 \ldots k$:
  - $C[i] \leftarrow 0$
- **for** $j = 0 \ldots n-1$:
  - $C[A[j]] \leftarrow C[A[j]] + 1$
- **for** $p = 1 \ldots k$:
  - $C[p] \leftarrow C[p] + C[p-1]$
- **for** $r = n-1 \ldots 0$:
  - $B[C[A[r]] - 1] \leftarrow A[r]$
  - $C[A[r]] \leftarrow C[A[r]] - 1$
- **return** B

- Este algoritmo guarda en un arreglo auxiliar, el número de elementos menores o iguales a cada elemento del arreglo inicial. Luego utiliza esta información para colocar cada elemento en la posición correcta en el arreglo ordenado.
- Complejidad O(n+k), si k < n, entonces O(n)
- Memoria adicional O(n+k)

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
    B[0 ... n-1] ← arreglo vacío de n celdas
    C[0 ... k] ← arreglo vacío de k+1 celdas
    for i = 0 ... k :
        C[i] ← 0
    for j = 0 ... n-1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1 ... k :
        C[p] ← C[p] + C[p-1]
    for r = n-1 ... 0 :
        B[C[A[r]] - 1] ← A[r]
        C[A[r]] ← C[A[r]] - 1
    return B
```

A =

2	5	3	0	2	3	0	3
0	1	2	3	4	5	6	7

k = 5

C =

0	0	0	0	0	0
0	1	2	3	4	5

C =

2	0	2	3	0	1
0	1	2	3	4	5

C =

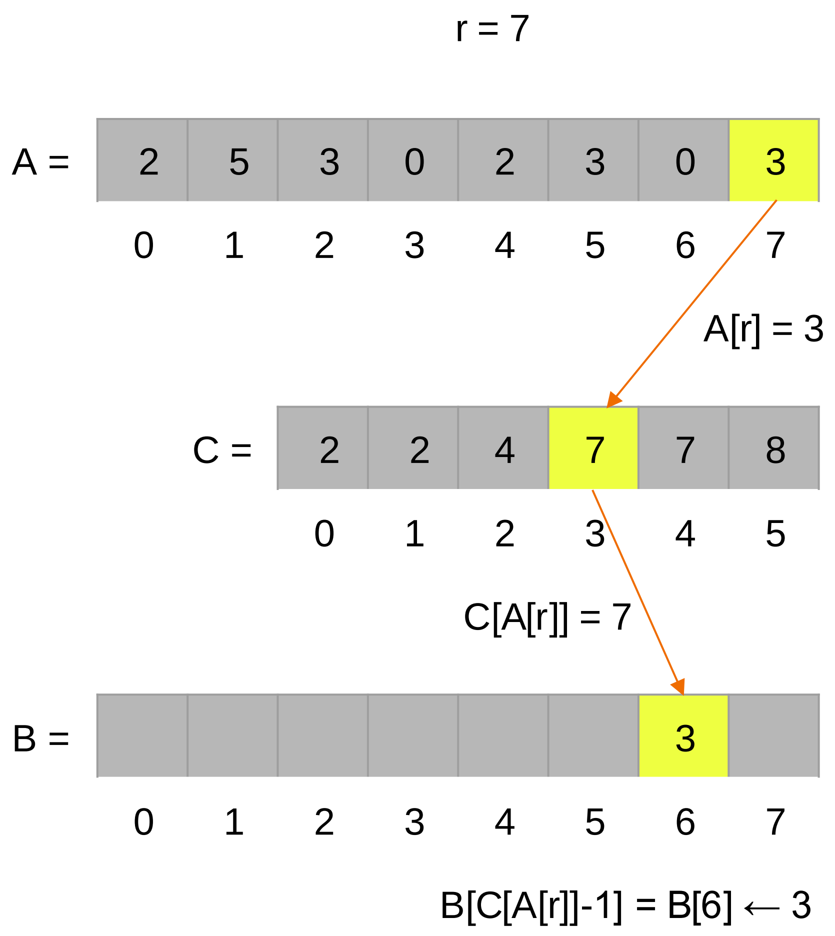
2	2	4	7	7	8
0	1	2	3	4	5

CountingSort()

input : Arreglo $A[0 \dots n-1]$, natural k
output: Arreglo $B[0 \dots n-1]$

`CountingSort` (A, k):

  $B[0 \dots n-1] \leftarrow$ arreglo vacío de n celdas
  $C[0 \dots k] \leftarrow$ arreglo vacío de $k+1$ celdas
  for $i = 0 \dots k$:
    $C[i] \leftarrow 0$
  for $j = 0 \dots n-1$:
    $C[A[j]] \leftarrow C[A[j]] + 1$
  for $p = 1 \dots k$:
    $C[p] \leftarrow C[p] + C[p-1]$
  for $r = n-1 \dots 0$:
    $\boxed{B[C[A[r]] - 1] \leftarrow A[r]}$
    $C[A[r]] \leftarrow C[A[r]] - 1$
  return B



CountingSort()

input : Arreglo $A[0 \dots n-1]$, natural k
output: Arreglo $B[0 \dots n-1]$

`CountingSort` (A, k):

- $B[0 \dots n-1] \leftarrow$ arreglo vacío de n celdas
- $C[0 \dots k] \leftarrow$ arreglo vacío de $k+1$ celdas
- **for** $i = 0 \dots k$:
  - $C[i] \leftarrow 0$
- **for** $j = 0 \dots n-1$:
  - $C[A[j]] \leftarrow C[A[j]] + 1$
- **for** $p = 1 \dots k$:
  - $C[p] \leftarrow C[p] + C[p-1]$
- **for** $r = n-1 \dots 0$:
  - $B[C[A[r]] - 1] \leftarrow A[r]$
  - $\boxed{C[A[r]] \leftarrow C[A[r]] - 1}$
- **return** B

r = 7

A =

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

A[r] = 3

C =

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

C[A[r]] ← C[A[r]] - 1 = 6

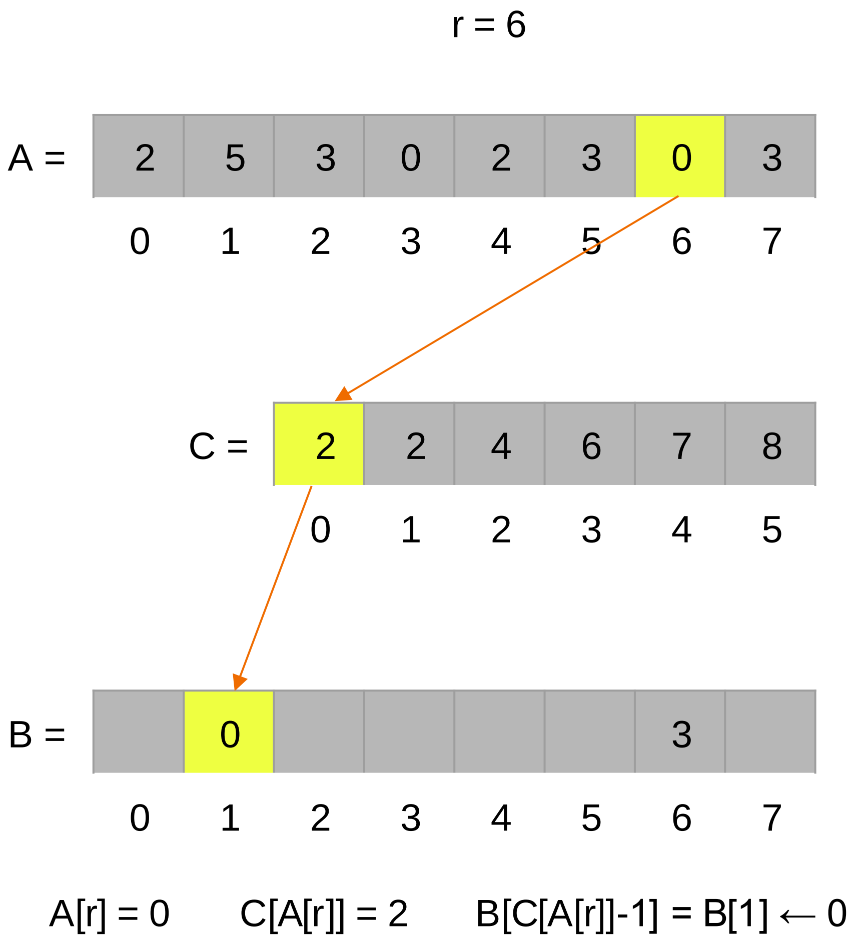
B =

0	1	2	3	4	5	6	7
						3	

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
    B[0...n−1] ← arreglo vacío de n celdas
    C[0...k] ← arreglo vacío de k+1 celdas
    for i = 0...k :
        C[i] ← 0
    for j = 0...n−1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1...k :
        C[p] ← C[p] + C[p−1]
    for r = n−1...0 :
        B[C[A[r]] − 1] ← A[r]
        C[A[r]] ← C[A[r]] − 1
    return B
```



CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
    B[0 ... n-1] ← arreglo vacío de n celdas
    C[0 ... k] ← arreglo vacío de k+1 celdas
    for i = 0 ... k :
        C[i] ← 0
    for j = 0 ... n-1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1 ... k :
        C[p] ← C[p] + C[p-1]
    for r = n-1 ... 0 :
        B[C[A[r]] - 1] ← A[r]
        C[A[r]] ← C[A[r]] - 1
    return B
```

r = 5

A =

2	5	3	0	2	3	0	3
0	1	2	3	4	5	6	7

C =

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

B =

	0				3	3	
0	1	2	3	4	5	6	7

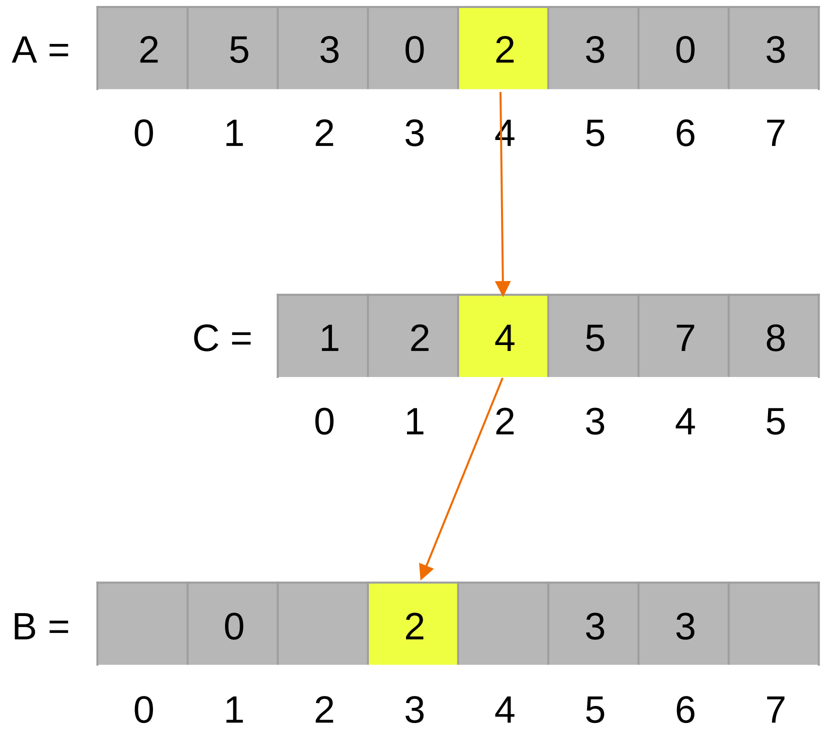
A[r] = 3 C[A[r]] = 6 B[C[A[r]]-1] = B[5] ← 3

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
    B[0 ... n − 1] ← arreglo vacío de n celdas
    C[0 ... k] ← arreglo vacío de k + 1 celdas
    for i = 0 ... k :
        C[i] ← 0
    for j = 0 ... n − 1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1 ... k :
        C[p] ← C[p] + C[p − 1]
    for r = n − 1 ... 0 :
        B[C[A[r]] − 1] ← A[r]
        C[A[r]] ← C[A[r]] − 1
    return B
```

r = 4



A[r] = 2 C[A[r]] = 4 B[C[A[r]]-1] = B[3] ← 2

CountingSort()

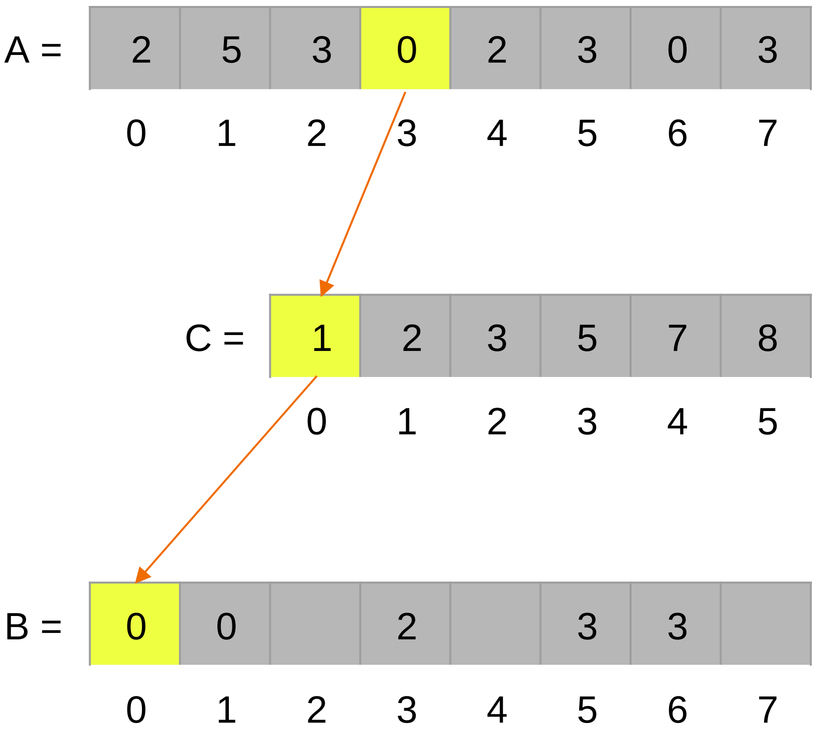
input : Arreglo $A[0 \ldots n-1]$, natural k

output: Arreglo $B[0 \ldots n-1]$

`CountingSort` (A, k):

$B[0 \ldots n-1] \leftarrow$ arreglo vacío de n celdas
$C[0 \ldots k] \leftarrow$ arreglo vacío de $k+1$ celdas
for $i = 0 \ldots k$:
    $C[i] \leftarrow 0$
for $j = 0 \ldots n-1$:
    $C[A[j]] \leftarrow C[A[j]] + 1$
for $p = 1 \ldots k$:
    $C[p] \leftarrow C[p] + C[p-1]$
for $r = n-1 \ldots 0$:
    $\boxed{B[C[A[r]]-1] \leftarrow A[r]}$
    $\boxed{C[A[r]] \leftarrow C[A[r]] - 1}$
return B

r = 3

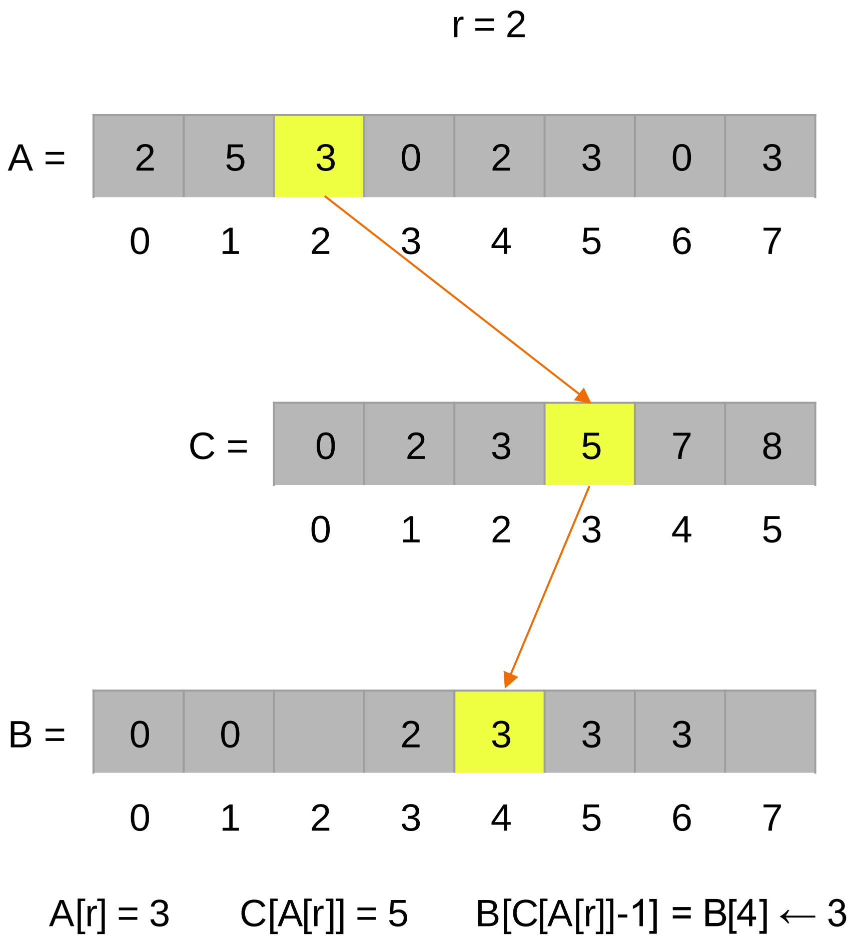


A[r] = 0 C[A[r]] = 1 B[C[A[r]]-1] = B[0] $\leftarrow$ 0

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

```
CountingSort (A, k):
    B[0...n-1] ← arreglo vacío de n celdas
    C[0...k] ← arreglo vacío de k+1 celdas
    for i = 0...k :
        C[i] ← 0
    for j = 0...n-1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1...k :
        C[p] ← C[p] + C[p-1]
    for r = n-1...0 :
        B[C[A[r]] - 1] ← A[r]
        C[A[r]] ← C[A[r]] - 1
    return B
```



CountingSort()

input : Arreglo $A[0 \dots n-1]$, natural k
output: Arreglo $B[0 \dots n-1]$

`CountingSort` (A, k):

- $B[0 \dots n-1] \leftarrow$ arreglo vacío de n celdas
- $C[0 \dots k] \leftarrow$ arreglo vacío de $k+1$ celdas
- **for** $i = 0 \dots k$:
  - $C[i] \leftarrow 0$
- **for** $j = 0 \dots n-1$:
  - $C[A[j]] \leftarrow C[A[j]] + 1$
- **for** $p = 1 \dots k$:
  - $C[p] \leftarrow C[p] + C[p-1]$
- **for** $r = n-1 \dots 0$:
  - $\boxed{B[C[A[r]]-1] \leftarrow A[r]}$
  - $\boxed{C[A[r]] \leftarrow C[A[r]] - 1}$
- **return** B

r = 1

A =

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

C =

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

B =

0	1	2	3	4	5	6	7
0	0		2	3	3	3	5

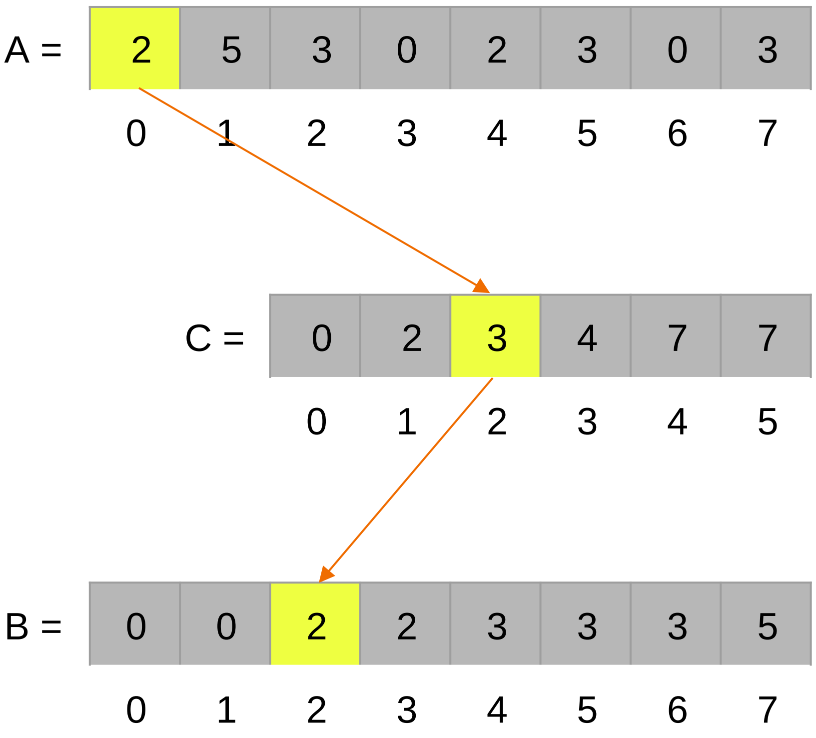
A[r] = 5 C[A[r]] = 8 B[C[A[r]]-1] = B[7] ← 5

CountingSort()

input : Arreglo $A[0 \dots n-1]$, natural k
output: Arreglo $B[0 \dots n-1]$

```
CountingSort (A, k):
    B[0...n−1] ← arreglo vacío de n celdas
    C[0...k] ← arreglo vacío de k+1 celdas
    for i = 0...k :
        C[i] ← 0
    for j = 0...n−1 :
        C[A[j]] ← C[A[j]] + 1
    for p = 1...k :
        C[p] ← C[p] + C[p−1]
    for r = n−1...0 :
        B[C[A[r]] − 1] ← A[r]
        C[A[r]] ← C[A[r]] − 1
    return B
```

r = 0



A[r] = 2 C[A[r]] = 3 B[C[A[r]]-1] = B[2] ← 2

RadixSort()

```
RadixSort(A, d):
    for j = 0 . . . d − 1 :
        StableSort(A, j)
```

Ordenamos desde el dígito menos significativo (0 se refiere a unidades, 1 a decenas, 2 a centenas… y así)

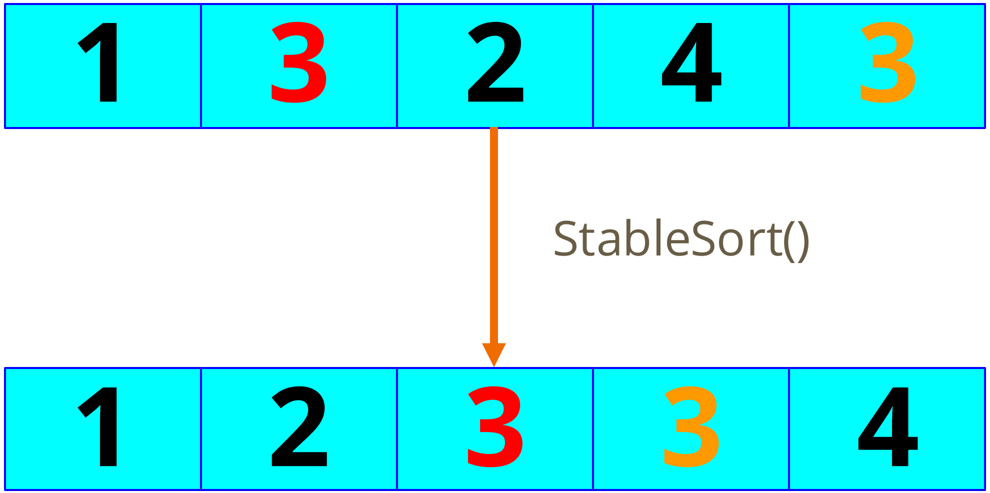
Se necesita un algoritmo de ordenamiento que sea **estable** como subrutina (CountingSort nos sirve!)

Un algoritmo es estable cuando los elementos que tenían el mismo valor antes de ordenar, mantienen su mismo orden original entre sí después de la ejecución del algoritmo.

Complejidad: O(nk)

Ejemplo estabilidad

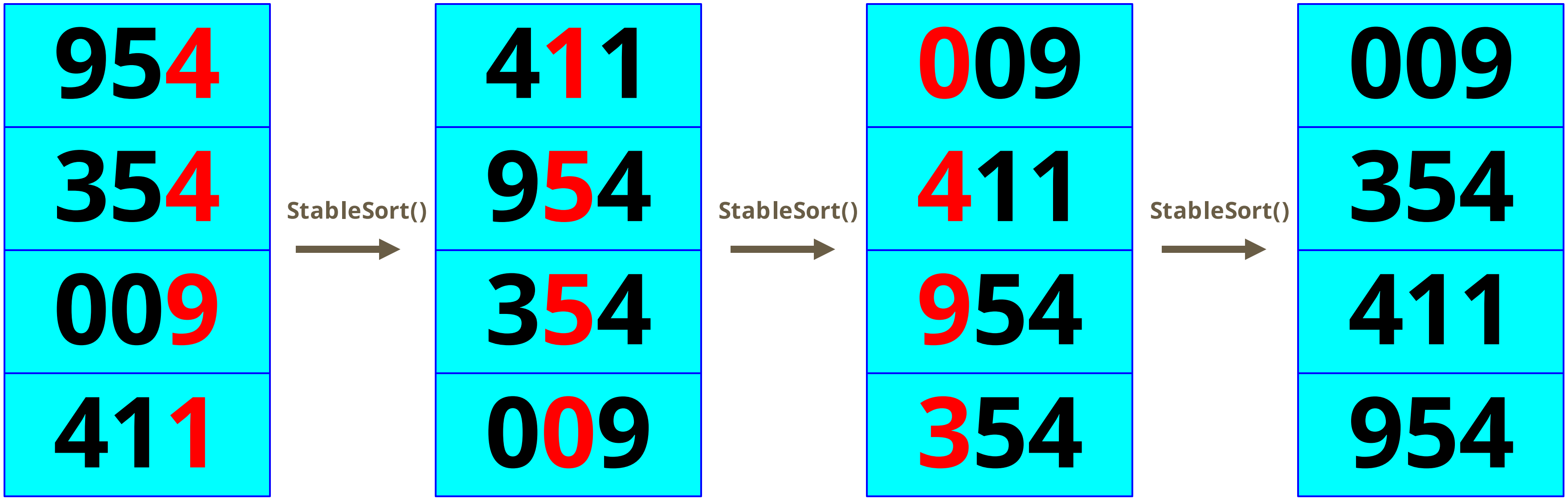
Sea StableSort() un algoritmo de ordenación estable:



Notar que el orden entre el 3 rojo y el 3 naranjo se mantiene después de la ejecución del algoritmo de ordenación estable

Para los números con menos dígitos que el número con mayor cantidad de dígitos en el arreglo, podemos considerar la ausencia de un dígito determinado como 0 (Ver 009 en el ejemplo)

RadixSort()



Sort Dígito 0 (Unidades)

Sort Dígito 1 (Decenas)

Sort Dígito 2 (Centenas)

Resultado Final

P3 I2-2022-2

Para procesar la información obtenida desde el telescopio espacial James Webb, los objetos del campo de observación son identificados utilizando el Guide Star Catalog (GSC) el cual asocia a cada objeto un identificador de la forma: `fffff0nnnnn` en que `fffff` es un valor alfanumérico (0-9,A-Z) que identifica la región del espacio en que se encuentra el objeto y `nnnnn` es un correlativo del objeto en esa región (el 0 se usa como separador). El procesamiento requiere ordenar muy eficientemente (con eficiencia lineal ya que se deben ordenar 100 mil objetos distintos cada vez) los datos obtenidos desde múltiples campos de observación distintos, utilizando como criterio de orden dicho identificador.

P3 I2-2022-2

(a) [2 ptos.] Aplique RadixSort con CountingSort para realizar lo solicitado, detallando los ajustes necesarios al pseudo código visto en clases para que sean adecuados a las características del problema planteado.

P3 I2-2022-2

Solución: El orden es el “natural”, menos significativo a la derecha y más significativo a la izquierda (de 0 a d−1 con d = 11). Modifica Radix sort para incorporar la cardinalidad del dominio del dígito a ordenar en la llamada a counting sort, y modifica counting sort para utilizar dicho parámetro:

```
RadixSort (A, d) :
    for j = 0 ... d − 1 :
        if j ≥ 0 ∧ j < 6 :      ▷ rango numérico, incluye al 0 central
            CountSort (A, j, 10)
        else:      ▷ rango alfanumérico
            CountSort (A, j, 37)
```

```
input : Arreglo A[0 ..., n − 1], j dígito a ordenar de A, k valores posibles
CountSort (A, j, k) :
    B[0 ... n − 1] ← arreglo vacío
    C[0 ... k] ← arreglo vacío
    for i = 0 ... k :
        C[i] ← 0
    for m = 0 ... n − 1 :
        C[A[m][j]] ← C[A[m][j]] + 1      ▷ A[m][j] dígito j de la palabra m
    for p = 1 ... k :
        C[p] ← C[p] + C[p − 1]
    for r = n − 1 ... 0 :
        B[C[A[r][j]]] ← A[r]
        C[A[r][j]] ← C[A[r][j]] − 1
    for q = 0 ... n − 1 :
        A[q] ← B[q]
```

P3 I2-2022-2

(b) [2 ptos.] Utilizando el resultado de (a) proponga el pseudo código para permitir seleccionar si se desea ordenar en forma ascendente o descendente, manteniendo la complejidad de tiempo en $\mathcal{O}(n)$.

P3 I2-2022-2

(b) [2 ptos.] Utilizando el resultado de (a) proponga el pseudo código para permitir seleccionar si se desea ordenar en forma ascendente o descendente, manteniendo la complejidad de tiempo en $\mathcal{O}(n)$.

Solución.

Usando la parte (a) se obtiene el orden ascendente en $\mathcal{O}(n)$, si piden el orden descendente basta con recorrer el resultado de (a) y entregarlo en orden inverso, esto también es $\mathcal{O}(n)$ y como se "suma", el desempeño global sigue en $\mathcal{O}(n)$.

P3 I2-2022-2

(c) [2 ptos.] Un compañero sugiere reemplazar CountingSort por QuickSort. Detalle el impacto que esto tendría desde el punto de la correctitud del algoritmo y su desempeño.

P3 I2-2022-2

(c) [2 ptos.] Un compañero sugiere reemplazar CountingSort por QuickSort. Detalle el impacto que esto tendría desde el punto de la correctitud del algoritmo y su desempeño.

Solución.

QuickSort es $\mathcal{O}(n\log(n))$ y no es estable, luego el algoritmo si bien termina (al ordenar todos los dígitos) no cumple su propósito, ya que no entrega la salida ordenada correctamente. Además el desempeño pasa de ser $\mathcal{O}(n+d) = \mathcal{O}(n)$ a ser $\mathcal{O}(n\log(n)+d) = \mathcal{O}(n\log(n)+d)$.