Ayudantía 03

Heaps y Ordenamiento Lineal

Isabella Cherubini, Bernardita Nazar, Joaquín Viñuela, Paula Grune

Heaps

¿Qué es una cola de prioridad?

- Una estructura de datos que nos permite almacenar datos según su prioridad (dada por un valor). Por ejemplo, pruebas para las que hay que estudiar o urgencia de atención de una persona en una sala de espera

- Un criterio de prioridad: Orden de llegada (FIFO, LIFO). Importa la posición en la cola

- Podría interesarnos otro criterio, pero siempre nos interesará mantener estas funciones:
  - Insertar un dato con prioridad dada
  - Extraer el dato con mayor prioridad
  - Idealmente, cambiar la prioridad de un dato

Array vs Lista ligada

¿Donde almaceno mi cola de prioridad?

Array vs Lista ligada

¿Donde almaceno mi cola de prioridad?

Lo haremos en un array

Array vs Lista ligada

¿Donde almaceno mi cola de prioridad?

Lo haremos en un array

¿Por qué? El tamaño es limitado. Mejor usar una lista

Array vs Lista ligada

¿Donde almaceno mi cola de prioridad?

Lo haremos en un array

¿Por qué? El tamaño es limitado. Mejor usar una lista ligada

Hay casos donde usar una lista ligada nos dará problemas

- Buscar el máximo valor → O(n)
- Insertar en la posición correcta/mantener orden → O(n)

caro

Usar arrays nos limitará la cantidad de elementos, pero nos dará facilidad para acceder a los elementos (en O(1)) y mantener cierto “orden”

Array vs Lista ligada

¿Donde almaceno mi cola de prioridad?

Lo haremos en un array

¿Por qué? El tamaño es limitado. Mejor usar una lista ligada

Hay casos donde usar una lista ligada nos dará problemas

- Buscar el máximo valor $\rightarrow O(n)$
- Insertar en la posición correcta/mantener orden $\rightarrow O(n)$

caro

Usar arrays nos limitará la cantidad de elementos, pero nos dará facilidad para acceder a los elementos (en $O(1)$) y mantener cierto "orden"

Para las colas de prioridad usaremos Heaps

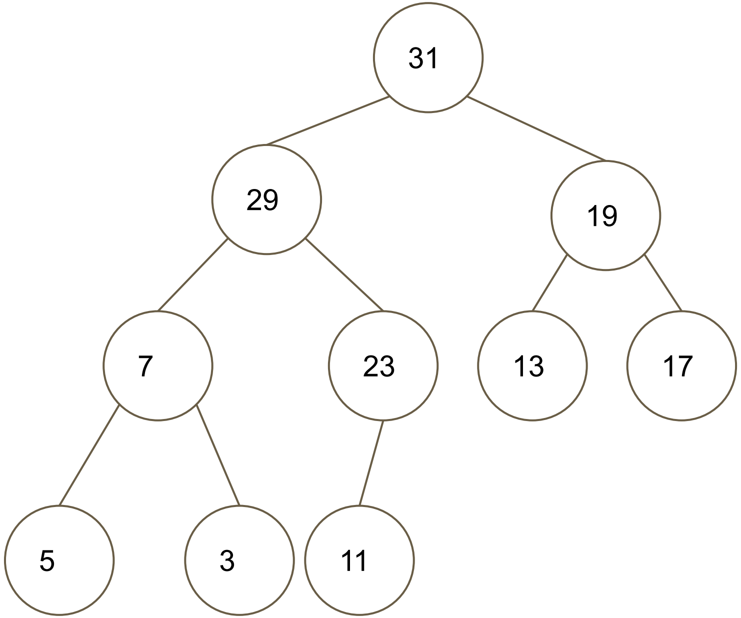
¿Qué es un Heap?

- 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 $H[2k + 1]$ y $H[2k + 2]$
  - El padre del elemento $H[k]$ es $H[\lfloor (k - 1)/2 \rfloor]$
- 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								

A lo que vinimos... extracción e inserción eficiente

1. Efectuamos la operación manteniendo un árbol binario casi-lleno
2. Restablecemos las propiedades de heap

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(?)

1) extracción eficiente

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 $best$

SiftDown(H, i):

if i *tiene hijos* **:**

$j \leftarrow$ hijo de i con mayor prioridad

if $H[j] > H[i]$ **:**

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

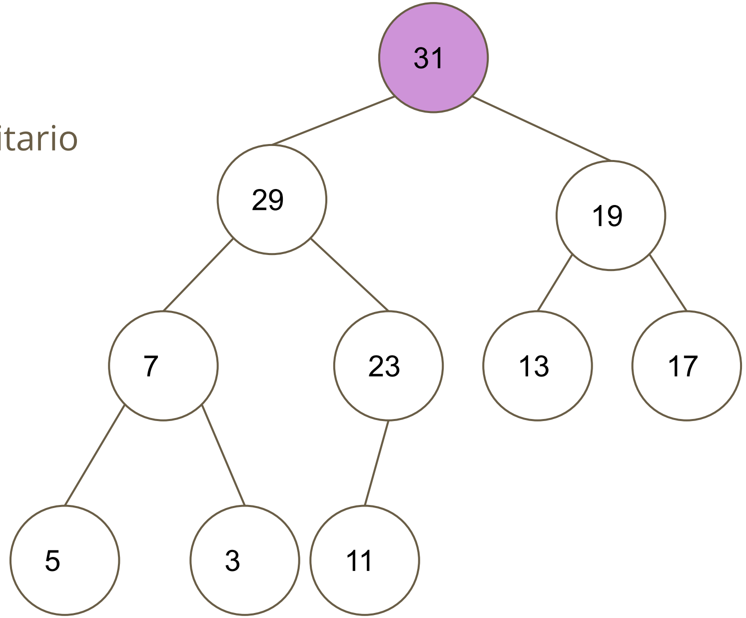
SiftDown(H, j)

O(log(n))

1) extracción eficiente

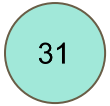
Queremos sacar el
elemento más prioritario

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



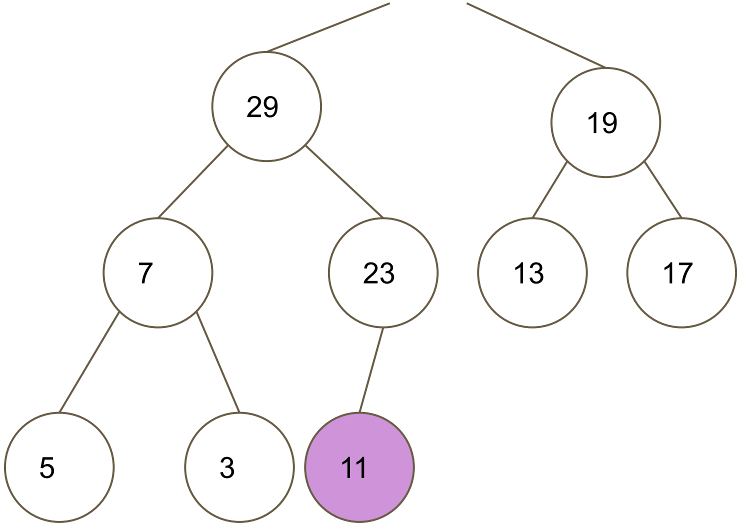
valor	31	29	19	7	23	13	17	5	3	11						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

1) extracción eficiente



Extraemos

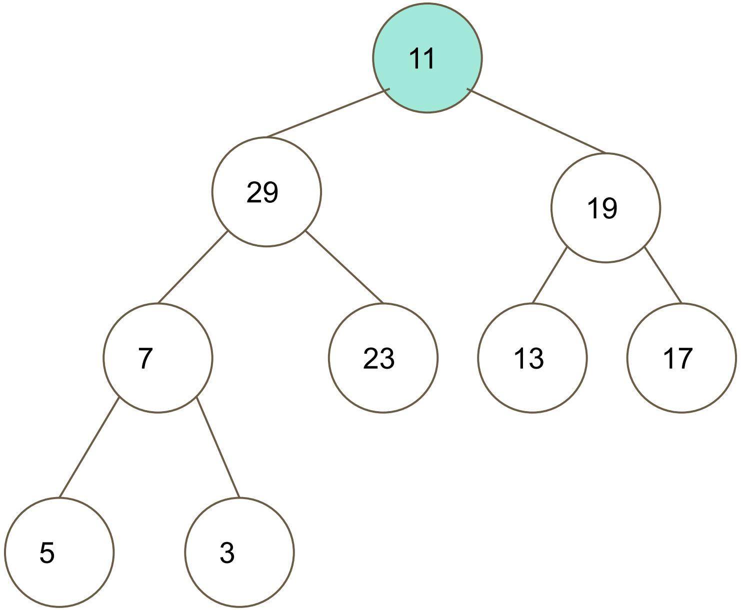
Extract(H):
  $i \leftarrow$ última celda no vacía de H
  $best \leftarrow H[0]$
  $H[0] \leftarrow H[i]$
  $H[i] \leftarrow \varnothing$
  SiftDown($H, 0$)
  return $best$



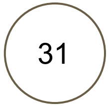
valor		29	19	7	23	13	17	5	3	11						…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

1) extracción eficiente

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



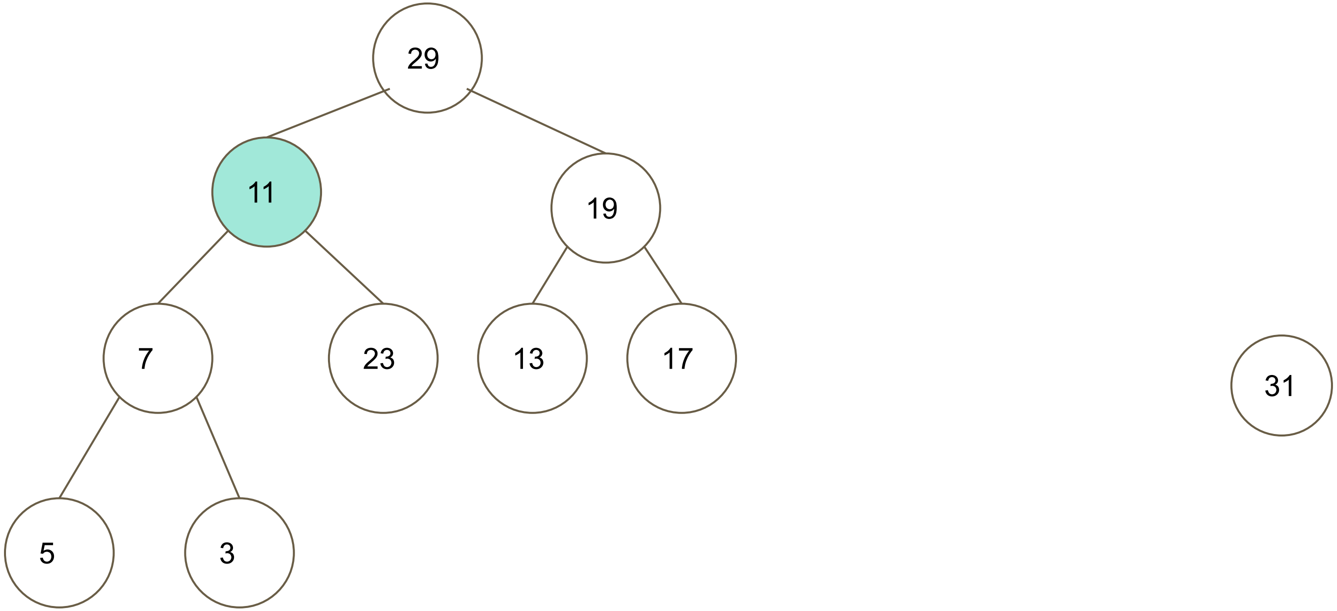
Empezamos con el ShiftDown para reacomodar



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* **:**
        $j \leftarrow$ hijo de i con mayor prioridad
        if $H[j] > H[i]$ **:**
            $H[j] \leftrightharpoons H[i]$
            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

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

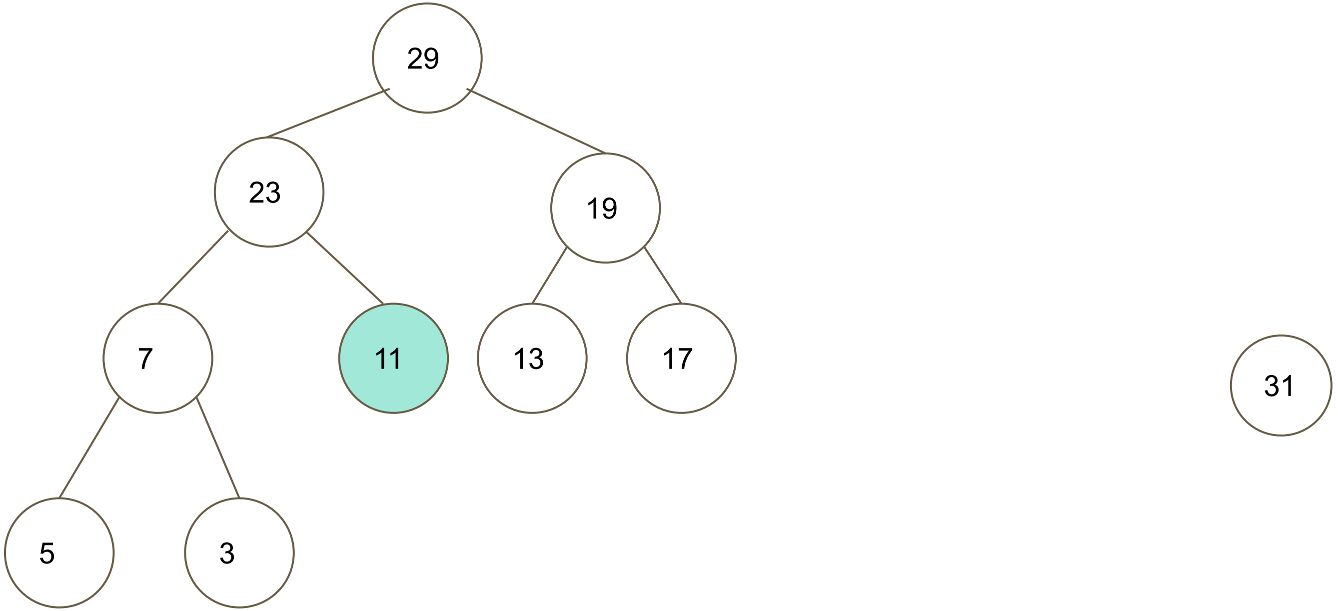
    if *i tiene hijos* **:**

        $j \leftarrow$ hijo de i con mayor prioridad

        if $H[j] > H[i]$ **:**

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

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



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

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

```
SiftUp(H, i):
    if i tiene padre :
        j ← ⌊i/2⌋
        if H[j] < H[i] :
            H[j] ⇋ H[i]
            SiftUp(H, j)
```

O(?)

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

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

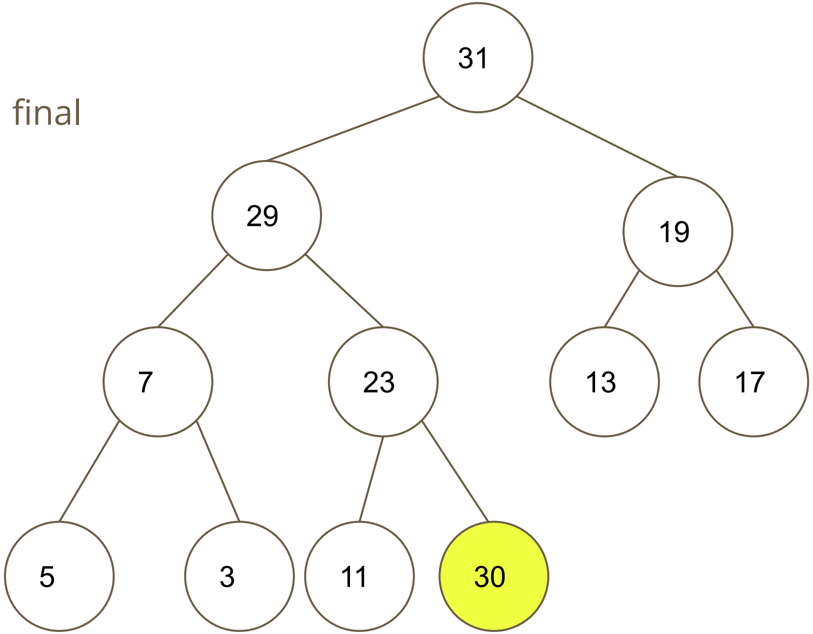
$\texttt{SiftUp}(H, j)$

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)
```



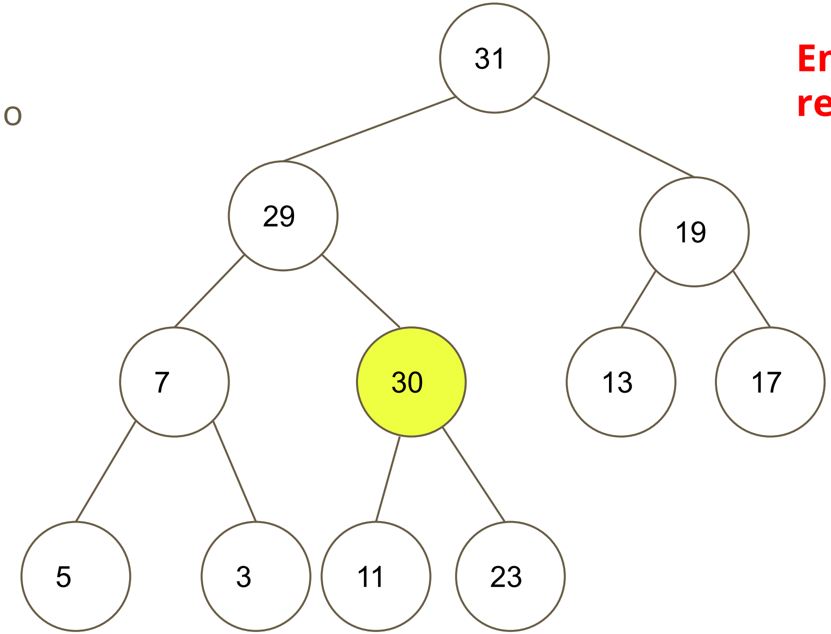
valor	31	29	19	7	23	13	17	5	3	11	30					…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

2) inserción eficiente

Empezamos a subir el nodo

Empezamos con el ShiftUp para reacomodar

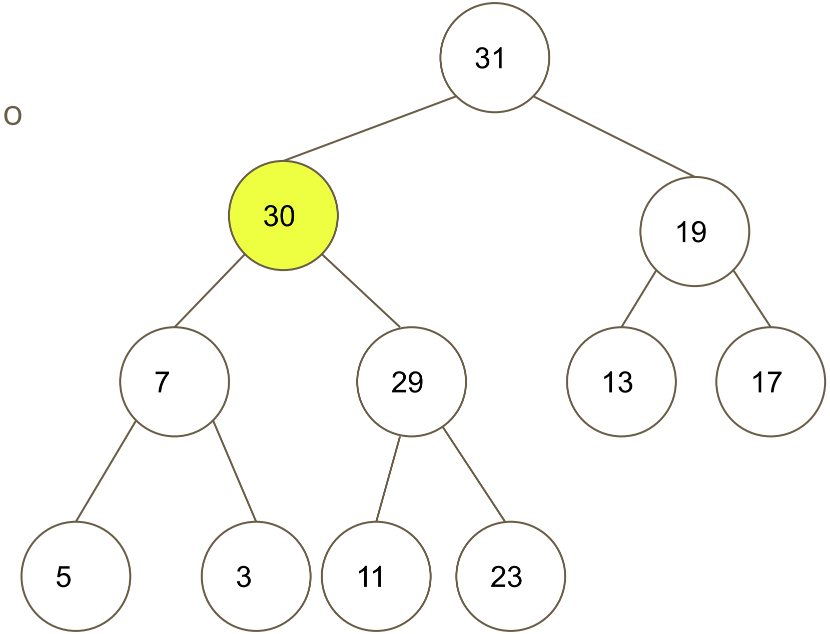
$\texttt{SiftUp}(H, i)$:
- **if** i *tiene padre* :
  - $j \leftarrow \lfloor i/2 \rfloor$
  - **if** $H[j] < H[i]$:
    - $H[j] \leftrightharpoons H[i]$
    - $\texttt{SiftUp}(H, j)$



valor	31	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



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

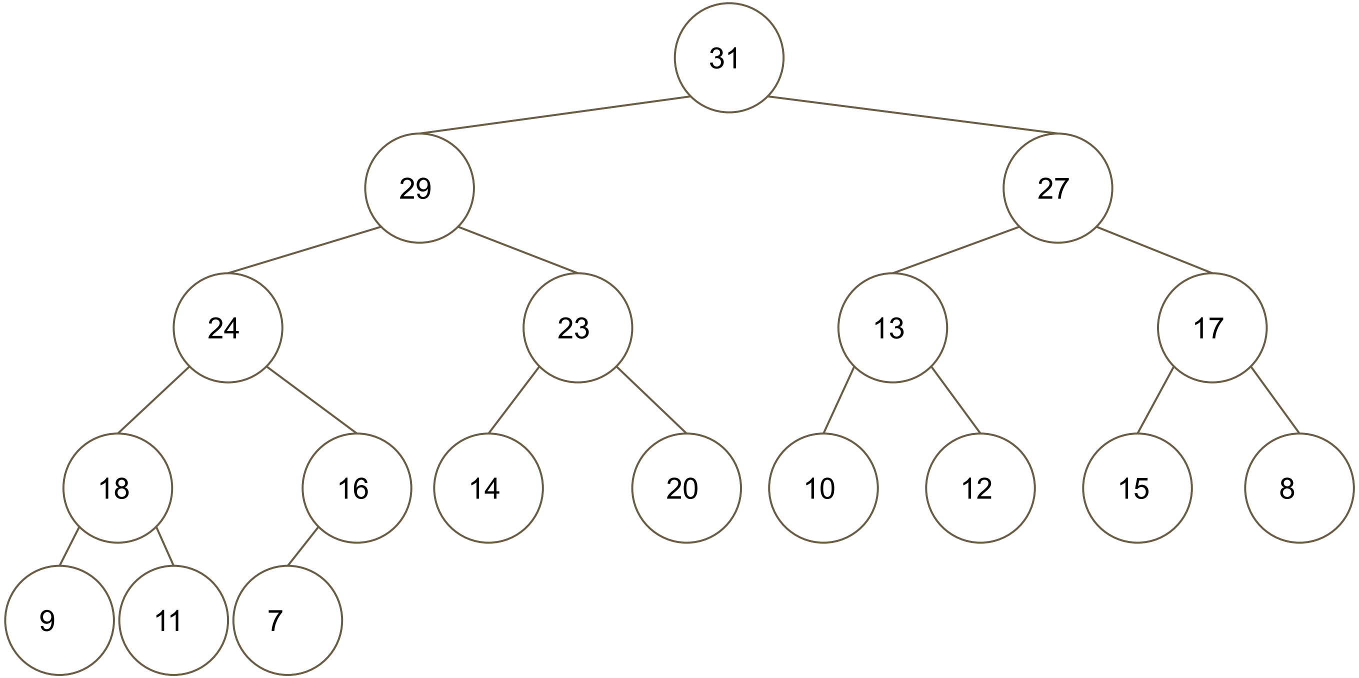
valor	31	30	19	7	29	13	17	5	3	11	23					…
posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

¿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:

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

   → No pasa nada

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

   → Hay que subir el nodo

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

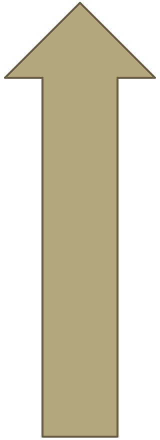
   → Hay que bajar el nodo y cambiarlo por ese hijo

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)

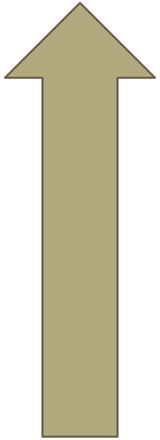
```
max = A[1]
A[1] = A[A.heap_size]
A.heap_size = A.heap_size − 1
MAX-HEAPIFY(A, A.heap_size)
return max
```

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
```

I3 2023-1 P1

La atención al afiliado en la sucursal de una Isapre requiere que, para ser atendidos, éstos deben registrar su RUT y el tipo de atención solicitada (venta de bonos, reembolsos, pago de cotizaciones, cambio de planes, etc.) en un tótem de auto-atención. Este entrega un número secuencial ascendente, los que son invocados en orden por los ejecutivos de la sucursal al estar disponibles para atender al afiliado.

I3 2023-1 P1

La atención al afiliado en la sucursal de una Isapre requiere que, para ser atendidos, estos deben registrar su RUT y el tipo de atención solicitada (venta de bonos, reembolsos, pago de cotizaciones, cambio de planes, etc.) en un tótem de auto-atención. Este entrega un <u>número secuencial ascendente</u>, los que son <u>invocados en orden</u> por los ejecutivos de la sucursal al estar disponibles para atender al afiliado.

(a) El tótem utiliza la función **`new(A, b, tipo, sec)`** para registrar en **`A`** los datos del afiliado **`b`** que solicita atención con secuencia **`sec`** entregada por el tótem. El sistema utiliza la llamada **`b = next(A)`** para sacar de **`A`** al siguiente afiliado **`b`** a atender según la secuencia. Escriba el pseudocódigo de ambas funciones e indique explícitamente qué estructura de datos utiliza para representar **`A`**.

I3 2023-1 P1

(a) El tótem utiliza la función `new(A, b, tipo, sec)` para registrar en **A** los datos del afiliado **b** que solicita atención con secuencia **sec** entregada por el tótem . El sistema utiliza la llamada `b = next(A)` para sacar de **A** al siguiente afiliado **b** a atender según la secuencia. Escriba el pseudocódigo de ambas funciones e indique explícitamente qué estructura de datos utiliza para representar **A**.

SOLUCIÓN:

Utilizando como estructura una cola FIFO implementada como lista ligada, los métodos son:

```
new(A, b, tipo, sec):
    b.tipo ← tipo
    b.sec ← sec
    Insert(A, b)

next(A):
    return Remove(A, 0)
```

donde `Insert(A, b)` agrega **b** al final de la lista ligada **A**, y `Remove(A, 0)` retorna el primer elemento (cabeza) de la lista ligada **A** luego de extraerlo.

I3 2023-1 P1

La atención al afiliado en la sucursal de una Isapre requiere que, para ser atendidos, estos deben registrar su RUT y el tipo de atención solicitada (venta de bonos, reembolsos, pago de cotizaciones, cambio de planes, etc.) en un tótem de auto-atención. Este entrega un <u>número secuencial ascendente</u>, los que son <u>invocados en orden</u> por los ejecutivos de la sucursal al estar disponibles para atender al afiliado.

(b) La gerente de la oficina desea mejorar la atención de los adultos mayores (> 65 años), considerando que al registrar al afiliado es posible incluir en **`b`** su edad (en meses según **`b.edad`**). Pide entonces promover que los adultos mayores sean atendidos, primero, de mayor a menor edad, y luego los demás afiliados según la secuencia entregada por el tótem. Modifique el pseudocódigo de las funciones de (a) e indique explícitamente qué estructura de datos usa para **`A`**. Puede suponer que no hay dos afiliados con la misma edad en meses.

Pista: La secuencia del tótem es creciente no negativa

I3 2023-1 P1

(b) La gerente de la oficina desea mejorar la atención de los adultos mayores (> 65 años), considerando que al registrar al afiliado es posible incluir en `b` su edad (en meses según `b.edad`). Pide entonces promover que los adultos mayores sean atendidos primero, de mayor a menor edad, y luego los demás afiliados según la secuencia entregada por el tótem. Modifique el pseudocódigo de las funciones de (a) e indique explícitamente qué estructura de datos usa para `A`. Puede suponer que no hay dos afiliados con la misma edad en meses.

Pista: La secuencia del tótem es creciente no negativa

SOLUCIÓN:

En este caso, se utiliza un MIN-heap donde la prioridad se almacena en el atributo `sec`. Para adultos mayores, este valor se reemplazará por su edad en negativo, de forma que el atributo sec por sí solo sirve como prioridad del MIN-heap. Con esto, los métodos quedan…

I3 2023-1 P1

SOLUCIÓN:

En este caso, se utiliza un MIN-heap donde la prioridad se almacena en el atributo `sec`. Para adultos mayores, este valor se reemplazará por su edad en negativo, de forma que el atributo sec por sí solo sirve como prioridad del MIN-heap. Con esto, los métodos quedan...

```
new(A, b, tipo, sec):
    b.tipo ← tipo
    if b.edad < 65 * 12:
        b.sec ← sec
    else:
        b.sec ← -1 * b.edad
    Insert(A, b)

next(A):
    return Extract(A, 0)
```

donde `Extract(A)` es el método de extracción de raíz en MIN-heap.

I3 2023-1 P1

La atención al afiliado en la sucursal de una Isapre requiere que, para ser atendidos, estos deben registrar su RUT y el tipo de atención solicitada (venta de bonos, reembolsos, pago de cotizaciones, cambio de planes, etc.) en un tótem de auto-atención. Este entrega un <u>número secuencial ascendente</u>, los que son <u>invocados en orden</u> por los ejecutivos de la sucursal al estar disponibles para atender al afiliado.

(c) Al medir el comportamiento de la mejora anterior se observa que el tiempo promedio de espera para ser atendido depende del tipo de atención de quienes llegaron antes, ya que algunas atenciones son rápidas (venta de bonos) y otras lentas (cambio de planes). Un consultor propone aplicar la estrategia SJF (Short Job First, Tarea Más Corta Primero) para privilegiar a los que requieren atenciones breves. Así, se puede usar un factor $0 \leq f \leq 1$ donde 1 representa la tarea más breve. Indique (no se requiere el pseudocódigo) qué debería modificar en su respuesta de (b) para incorporar este cambio.

I3 2023-1 P1

(c) Al medir el comportamiento de la mejora anterior se observa que el tiempo promedio de espera para ser atendido depende del tipo de atención de quienes llegaron antes, ya que algunas atenciones son rápidas (venta de bonos) y otras lentas (cambio de planes). Un consultor propone aplicar la estrategia SJF (Short Job First, Tarea Más Corta Primero) para privilegiar a los que requieren atenciones breves. Así, se puede usar un factor $0 \leq f \leq 1$ donde 1 representa la tarea más breve. Indique (no se requiere el pseudocódigo) qué debería modificar en su respuesta de (b) para incorporar este cambio.

SOLUCIÓN:

Se debe actualizar la forma en que se calcula la prioridad en el método new. Una estrategia es dividir la prioridad por el factor f de manera que las tareas más largas tienen mayores valores de prioridad luego del cambio.

Ordenación Lineal

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
```

- 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 =

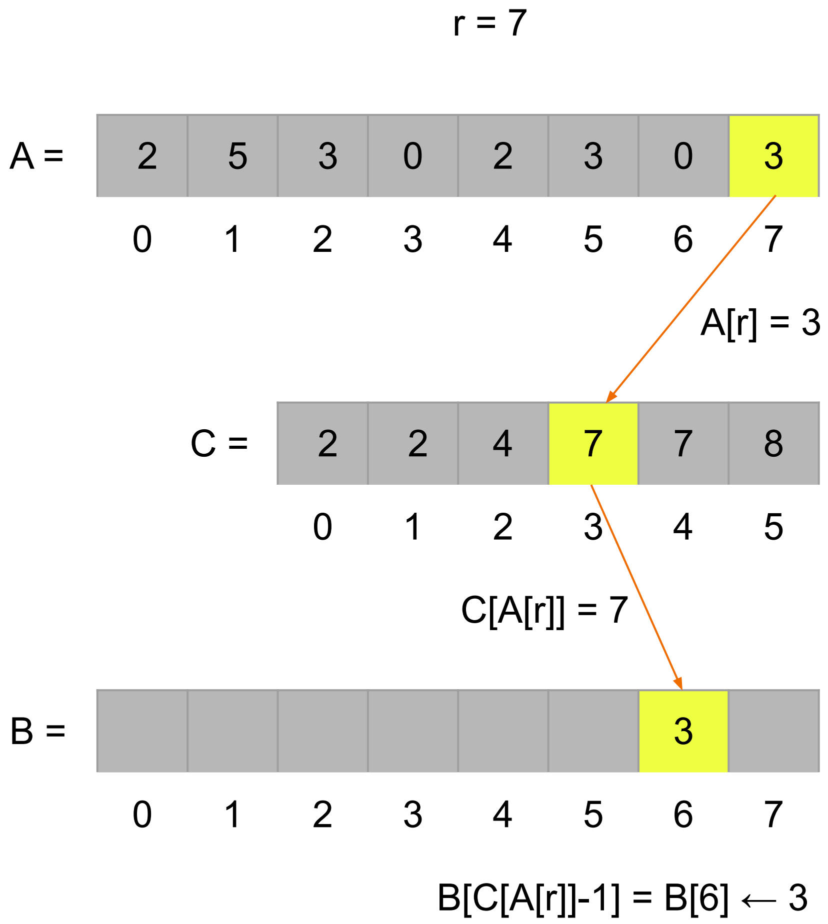
2	2	4	7	7	8
0	1	2	3	4	5

CountingSort()

input : Arreglo $A[0 \ldots n-1]$, 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 $r = n-1 \dots 0$ **:**
  $B[C[A[r]] - 1] \leftarrow A[r]$
  $\boxed{C[A[r]] \leftarrow C[A[r]] - 1}$
return B

r = 7

A =

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

A[r] = 3

C =

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

C[A[r]] ⟵ C[A[r]] - 1 = 6

B =

						3	
0	1	2	3	4	5	6	7

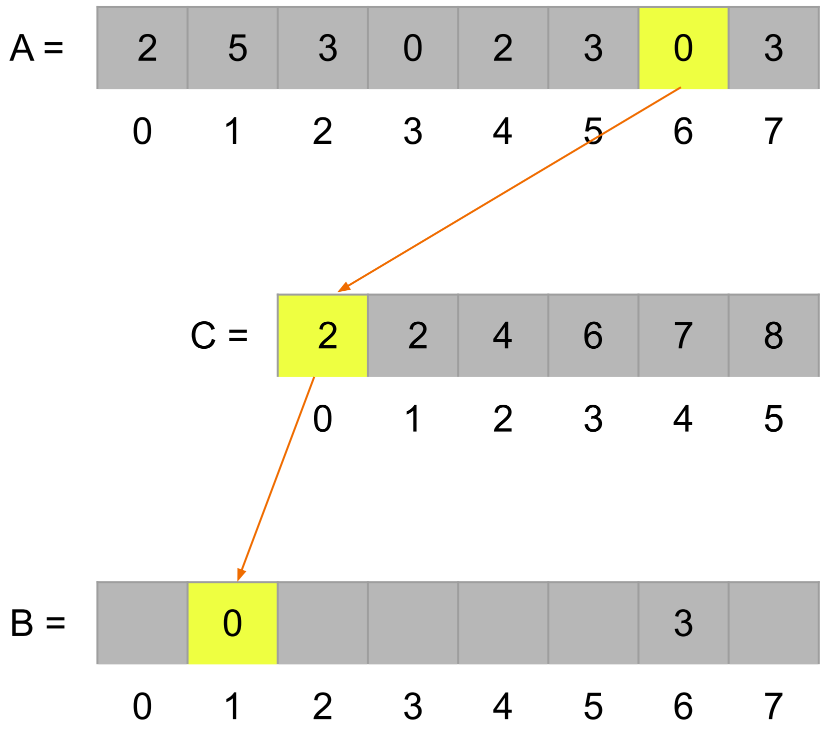
CountingSort()

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

r = 6



A[r] = 0 C[A[r]] = 2 B[C[A[r]]-1] = B[1] $\leftarrow$ 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
```

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()

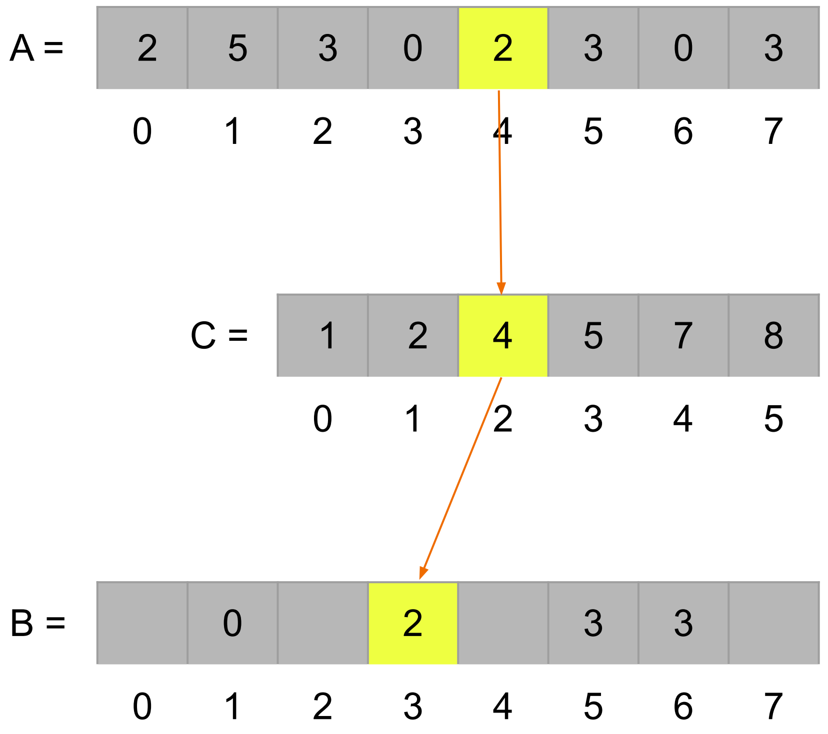
input : Arreglo $A[0 \ldots n-1]$, natural k

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

`CountingSort` (A, k):

$B[0 \ldots n-1] \leftarrow$ 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 = 4



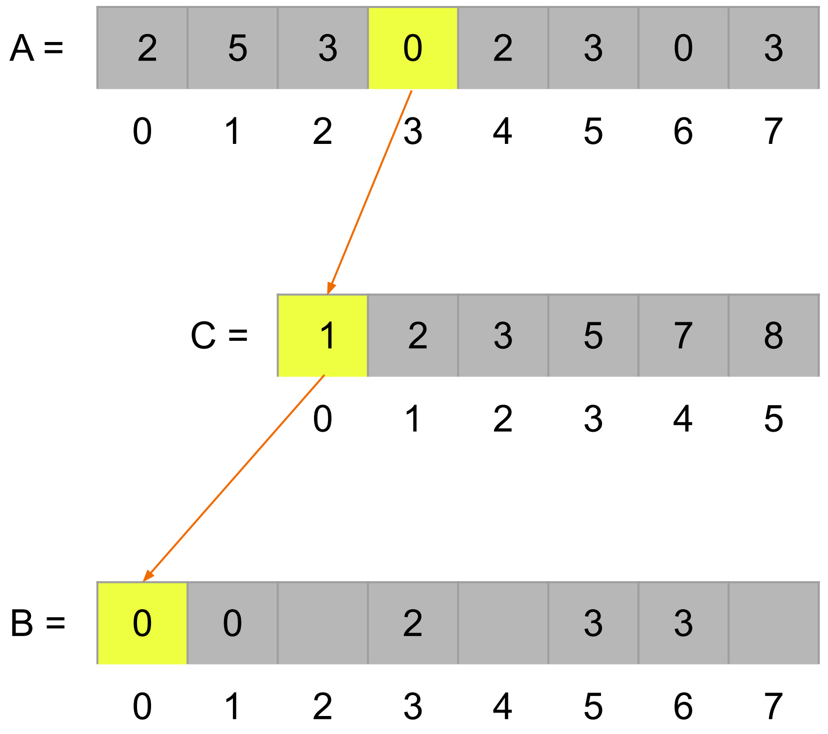
A[r] = 2   C[A[r]] = 4   B[C[A[r]]-1] = B[3] ← 2

CountingSort()

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

```
CountingSort (A, k):
    B[0...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 = 3



A[r] = 0 C[A[r]] = 1 B[C[A[r]]-1] = B[0] ← 0

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 = 2

A =

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

C =

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

B =

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

A[r] = 3 C[A[r]] = 5 B[C[A[r]]-1] = B[4] $\leftarrow$ 3

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

r = 1

A =

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

C =

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

B =

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

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

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 = 0

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

C =	0	2	3	4	7	7
	0	1	2	3	4	5

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

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)

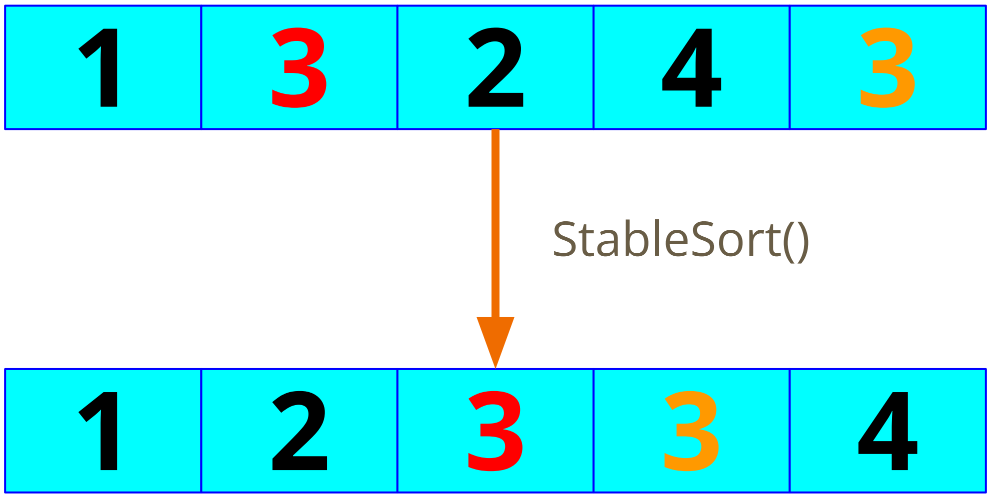
Se necesita un algoritmo de ordenamiento que sea **estable** como subrutina (CountingSort nos sirve!)

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

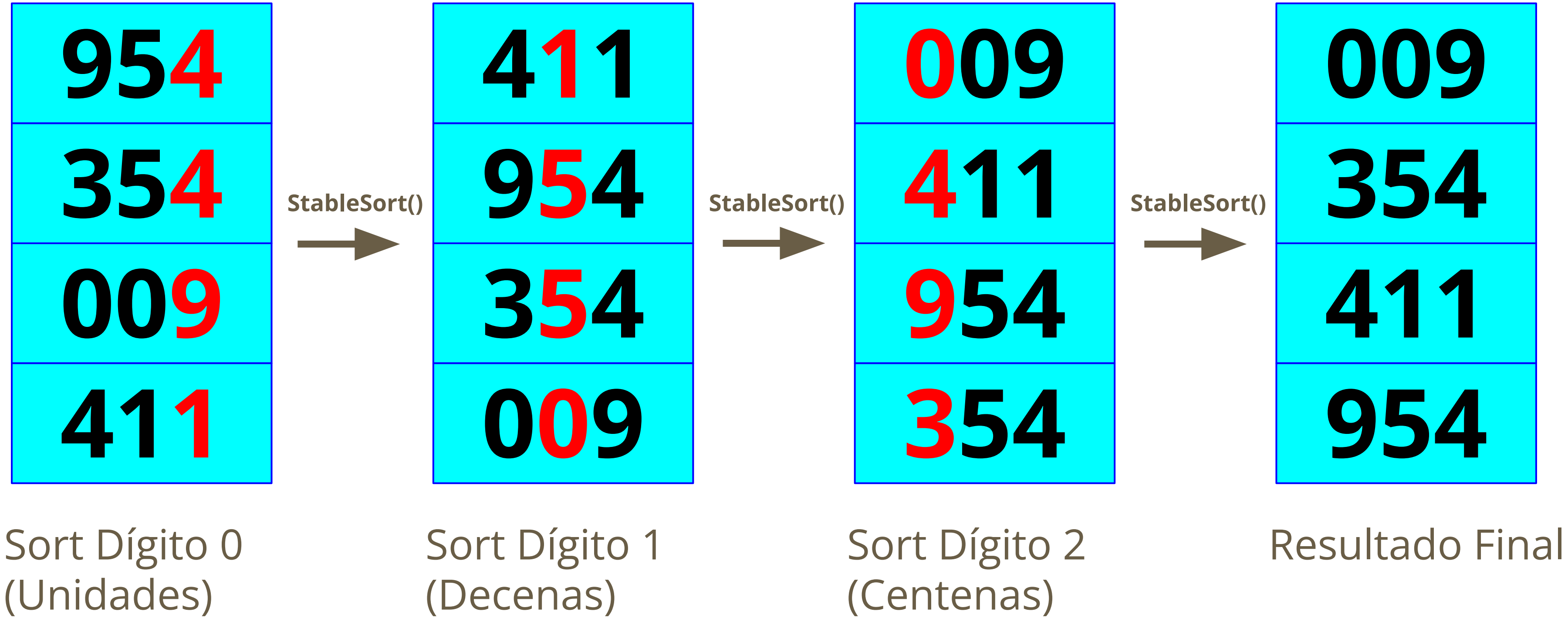
Sea StableSort() un algoritmo de ordenación estable:



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()



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 $\mathcal{O}(n\log(n)+d) = \mathcal{O}(n\log(n)+d)$.