Ayudantía 04

QuickSort y Heaps

QuickSort

Pseudocódigo

Input: secuencia A y dos enteros i, f

Output: Nada

```
   QuickSort (A, i, f):
1      if i ≤ f :
2          p ← Partition(A, i, f)
3          Quicksort(A, i, p − 1)
4          Quicksort(A, p + 1, f)
```

El verdadero Héroe

- Elige el pivote
- Coloca al pivote en su posición correcta en el arreglo ordenado
- Pone los elementos menores que él a su izquierda y los mayores a su derecha

```
Partition (A, i, f):
1     x ← índice aleatorio en {i, ..., f}
2     p ← A[x]
3     A[x] ⇄ A[f]
4     j ← i
5     for k = i ... f − 1 :
6         if A[k] < p :
7             A[j] ⇄ A[k]
8             j ← j + 1
9     A[j] ⇄ A[f]
10    return j
```

Resumen de quicksort

- Elegir pivote
- Mover elementos a cada lado del pivote, a un lado los mayores y al otro los menores
- La lista va a quedar separada en dos sublistas, una con los elementos a la izq. del pivote y otra con los de la derecha
- Repetir recursivamente para cada sublista mientras estas contengan más de un elemento.
- Lista ordenada!

Complejidad

Mejor Caso	**O(n * log(n))**
Caso Promedio	**O(n * log(n))**
Peor Caso	**O(n^2)**

¿Cuándo ocurre cada uno de ellos?

Veamos la visualización !

https://visualgo.net/en/sorting



Ejercicio

Supongamos que al ejecutar Quicksort sobre un arreglo particular, la subrutina partition hace siempre el mayor número posible de intercambios; ¿cuánto tiempo toma Quicksort en este caso? ¿Qué fracción del mayor número posible de intercambios se harían en el mejor caso? Justifica

- Intercambios de líneas 3 y 9 se hacen siempre
- Dentro del loop, sólo se hacen cuando A[k] < p
- Si queremos hacer la mayor cantidad de intercambios posibles, queremos que siempre A[k]< p.
- Entonces si tenemos un subarreglo de tamaño m, se hacen 2 + m-1 = m+1 intercambios
- Quedan subarreglos de largos 0 y m-1

```
Partition (A, i, f):
1     x ← índice aleatorio en {i, ..., f}
2     p ← A[x]
3     A[x] ⇄ A[f]
4     j ← i
5     for k = i ... f − 1 :
6         if A[k] < p :
7             A[j] ⇄ A[k]
8             j ← j + 1
9     A[j] ⇄ A[f]
10    return j
```

Con un arreglo de tamaño n tenemos:

$$T(n) = n + (n-1) + (n-2) + \ldots + 1$$

$$= \frac{n \times (n+1)}{2} = \frac{1}{2}\left(n^2 + n\right)$$

$$\in O\left(n^2\right)$$

b) El algoritmo quicker-sort llama a la subrutina pq-partition, que utiliza dos pivotes $\boldsymbol{p}$ y $\boldsymbol{q}$ $(\boldsymbol{p} < \boldsymbol{q})$ para particionar el arreglo en 5 partes: los elementos menores que $\boldsymbol{p}$, el pivote $\boldsymbol{p}$, los elementos entre $\boldsymbol{p}$ y $\boldsymbol{q}$, el pivote $\boldsymbol{q}$, y los elementos mayores que $\boldsymbol{q}$. Escribe el pseudocódigo de pq-partition. ¿Es quicker-sort más eficiente que quick-sort? Justifica.

Ya que la definición dice que $p < q$, entonces es posible asumir que no hay datos repetidos. De todos modos, esto no afecta el análisis.

Lo más simple es implementar partition con listas ligadas como vimos en clases:

```
1:  procedure PQ-PARTITION(lista ligada L)
2:      p, q ← dos nodos de L tal que p < q. Quitar estos nodos de L.
3:      A, B, C ← listas vacias    ▷ Los elementos menores a p, entre p y q y mayores a q respectivamente
4:      for nodo x ∈ L do
5:          if x < p then
6:              agregar x al final de A
7:          else if x < q then
8:              agregar x al final de B
9:          else
10:             agregar x al final de C
11:         end if
12:     end for
13:     return A, p, B, q, C
14: end procedure
```

El paso en la linea 2 es fácil de hacer en $\mathcal{O}(1)$, basta con extraer los primeros dos elementos de L, e intercambiarlos si $p > q$.

Recordemos que el peor caso de Quicksort se produce cuando partition separa una secuencia de m datos en dos secuencias disparejas, de 0 y $m-1$ datos cada una. En ese caso la complejidad para una secuencia inicial de n datos es de:

$$T(n) = n + (n-1) + (n-2) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

En este caso esto también puede suceder, solo que se separa en 3 secuencias disparejas, de 0, 0 y $m-2$ elementos cada una. En este caso la complejidad para una secuencia inicial de n datos es de:

$$T(n) = n + (n-2) + (n-4) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

Ambos algoritmos son iguales en el peor caso, así que no podemos afirmar que quickersort sea mejor que quicksort.

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 →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”

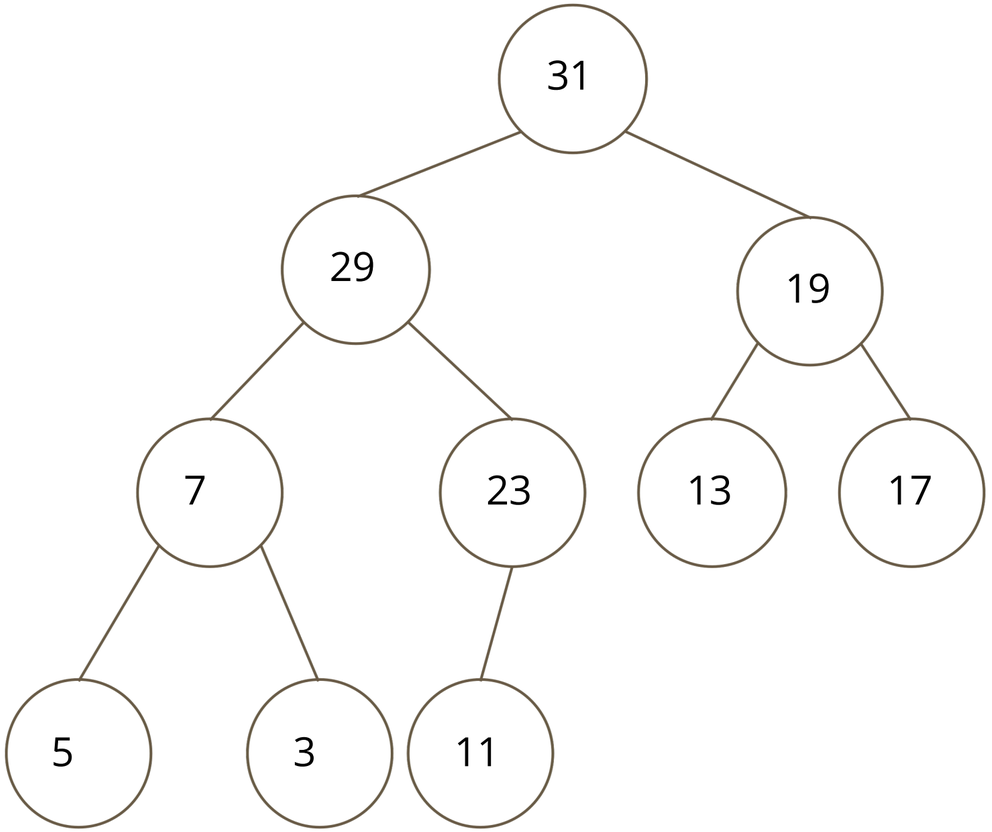
Para las colas de prioridad usaremos Heaps

¿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

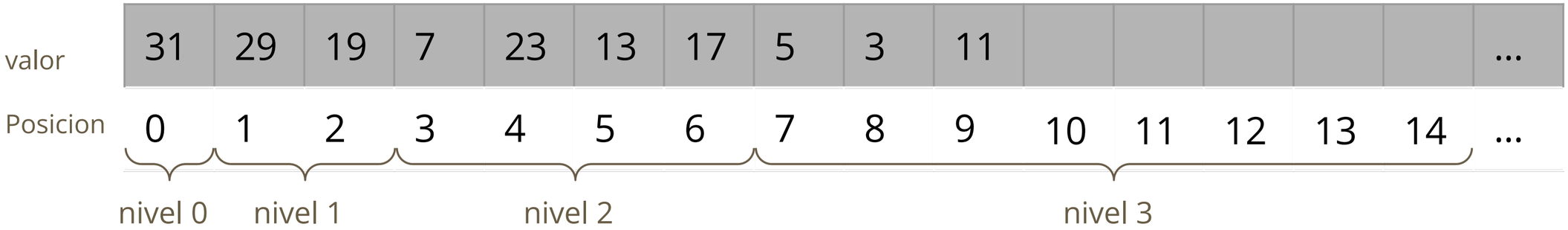
Ejemplo de Heap



valor	31	29	19	7	23	13	17	5	3	11						...
posicion	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[⌊(k − 1)/2⌋]
- agrupar los niveles



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

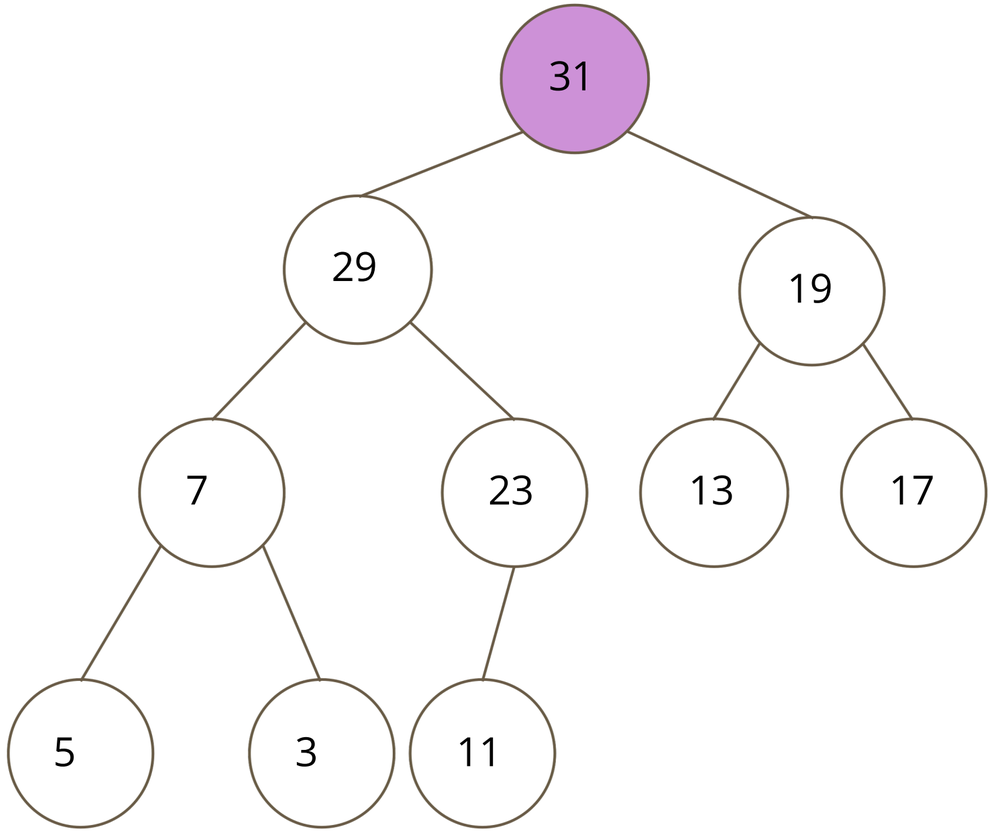
```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```

O(log(n))

1)extracción eficiente

Queremos sacar el elemento más prioritario

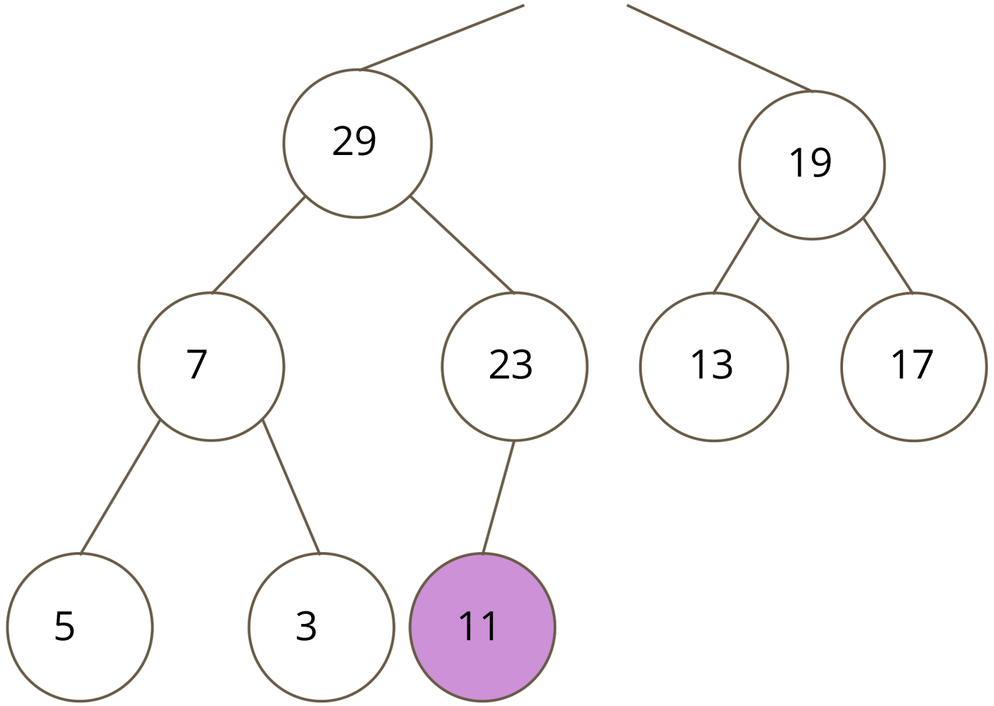
```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H, 0)
    return best
```

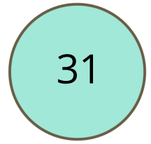


valor	31	29	19	7	23	13	17	5	3	11						...
Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

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
```



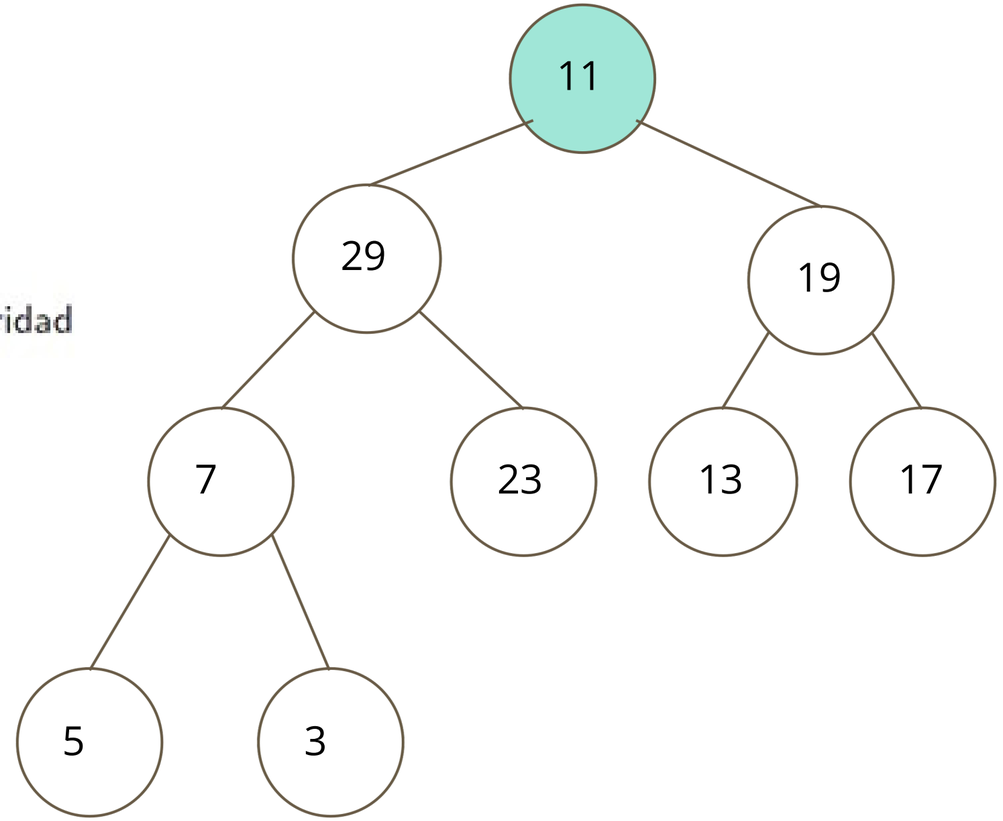


Extraemos

valor		29	19	7	23	13	17	5	3	11						...
Posicion	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)
```

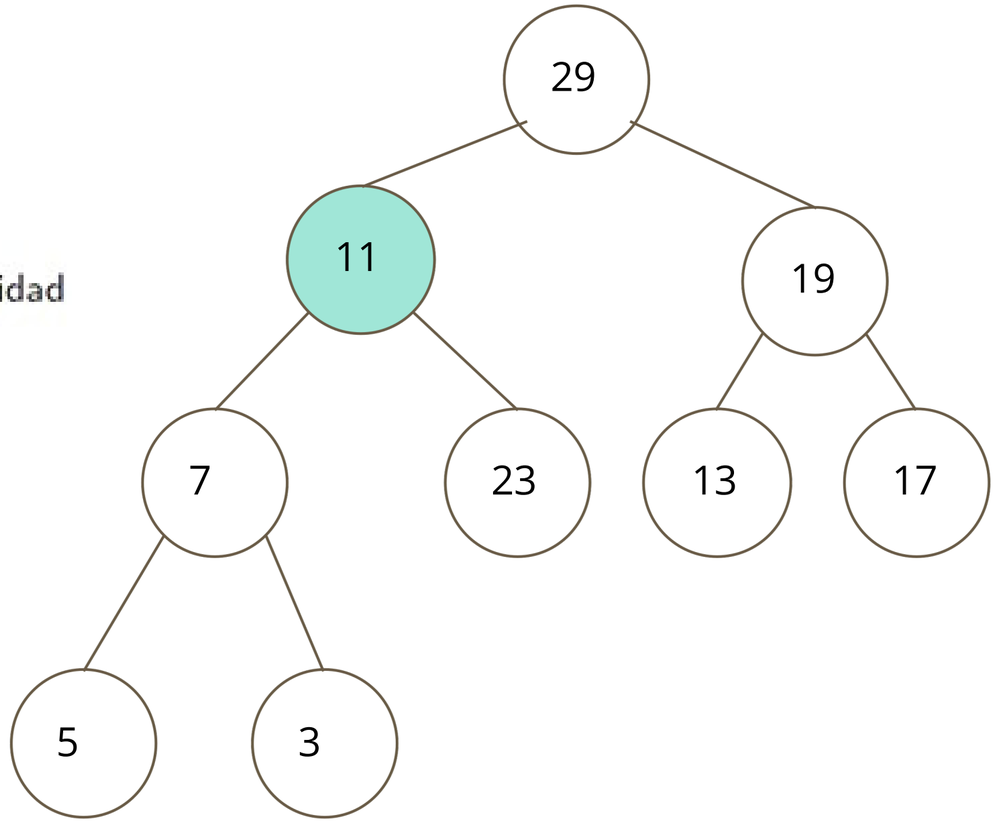


Empezamos con el ShiftDown para reacomodar

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

1)extracción eficiente

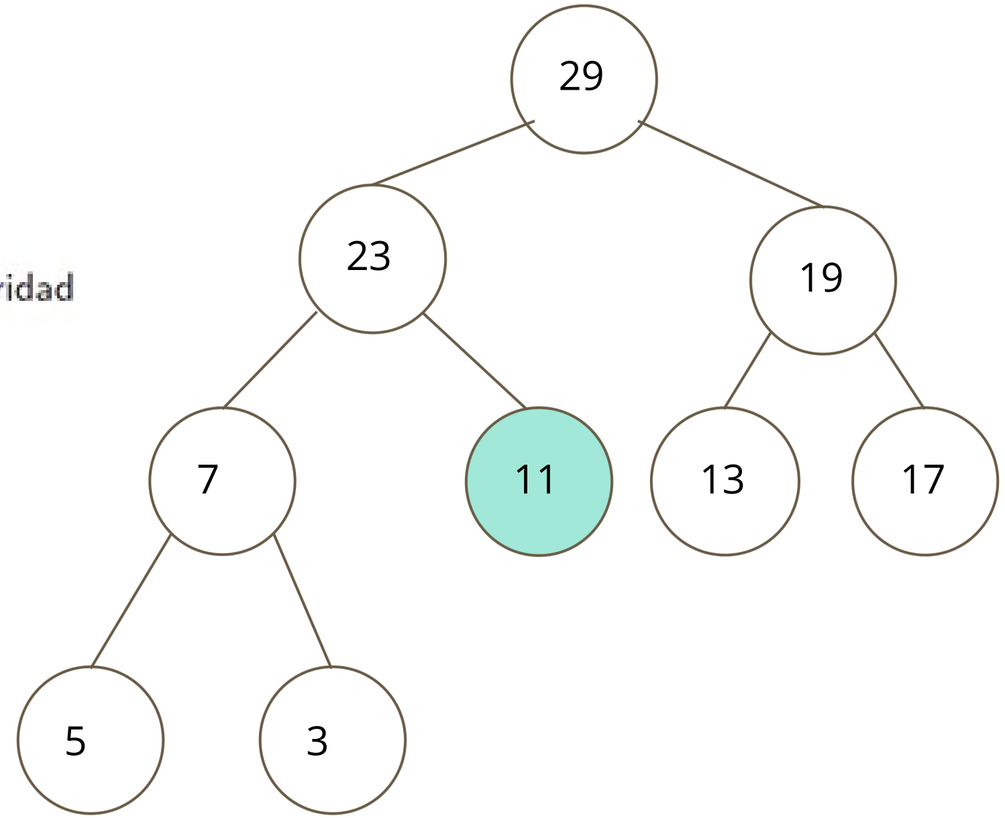
```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```



valor	29	11	19	7	23	13	17	5	3						...
Posicion	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	23	19	7	11	13	17	5	3						...
Posicion	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

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

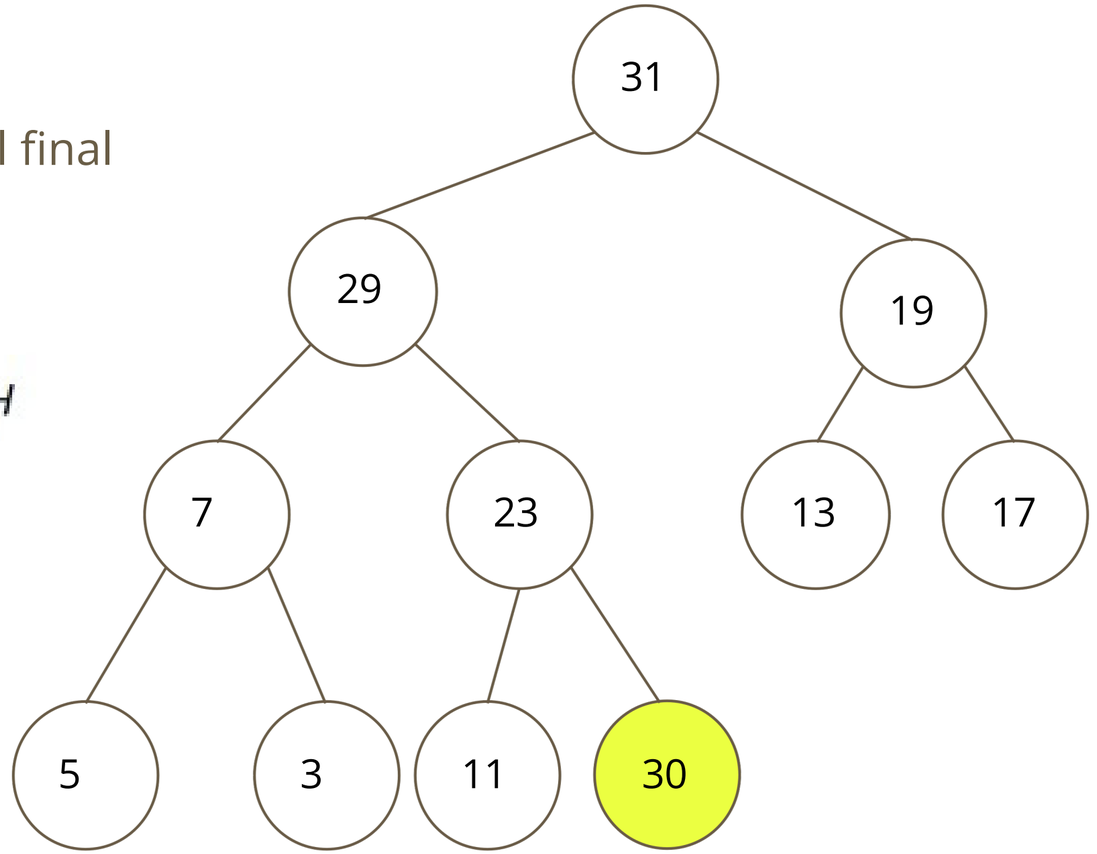
```
SiftUp(H, i):
    if i tiene padre :
        j ← ⌊i/2⌋
        if H[j] < H[i] :
            H[j] ⇋ H[i]
            SiftUp(H, j)
```

O(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)
```

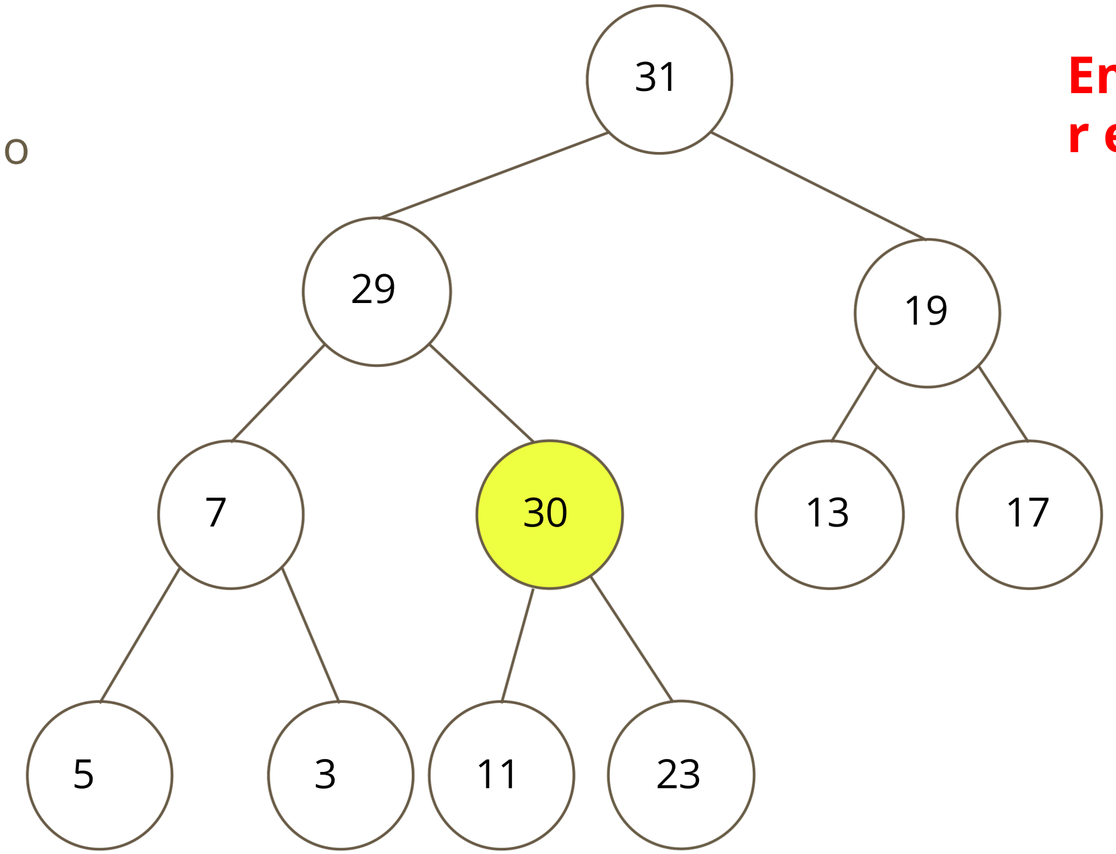


valor	31	29	19	7	23	13	17	5	3	11	3					...
Posicion	0	1	2	3	4	5	6	7	8	9	0	11	12	13	14	...

2) inserción eficiente

Empezamos a subir el nodo

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



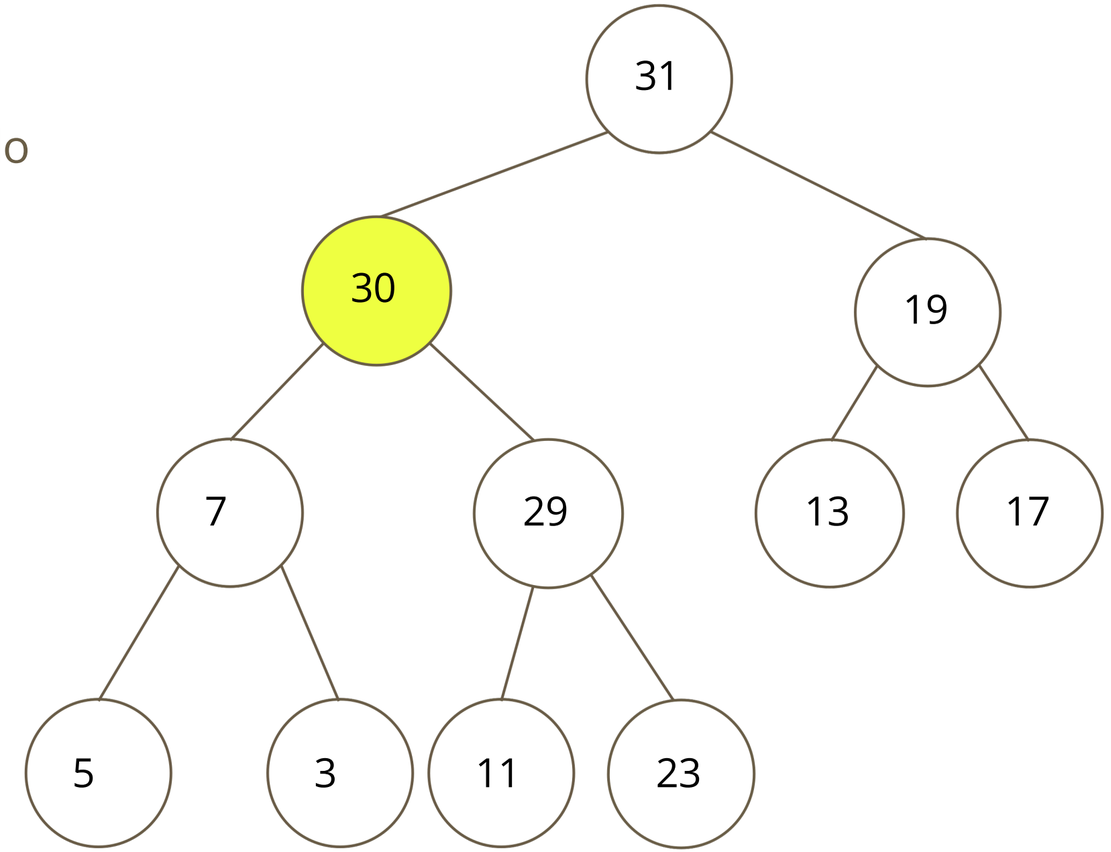
Empezamos con el ShiftUp para
r eac omo d ar

valor	31	29	19	7	30	13	17	5	3	11	2					...
Posicion	0	1	2	3	4	5	6	7	8	9	3	11	12	13	14	...

2) inserción eficiente

Empezamos a subir el nodo

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



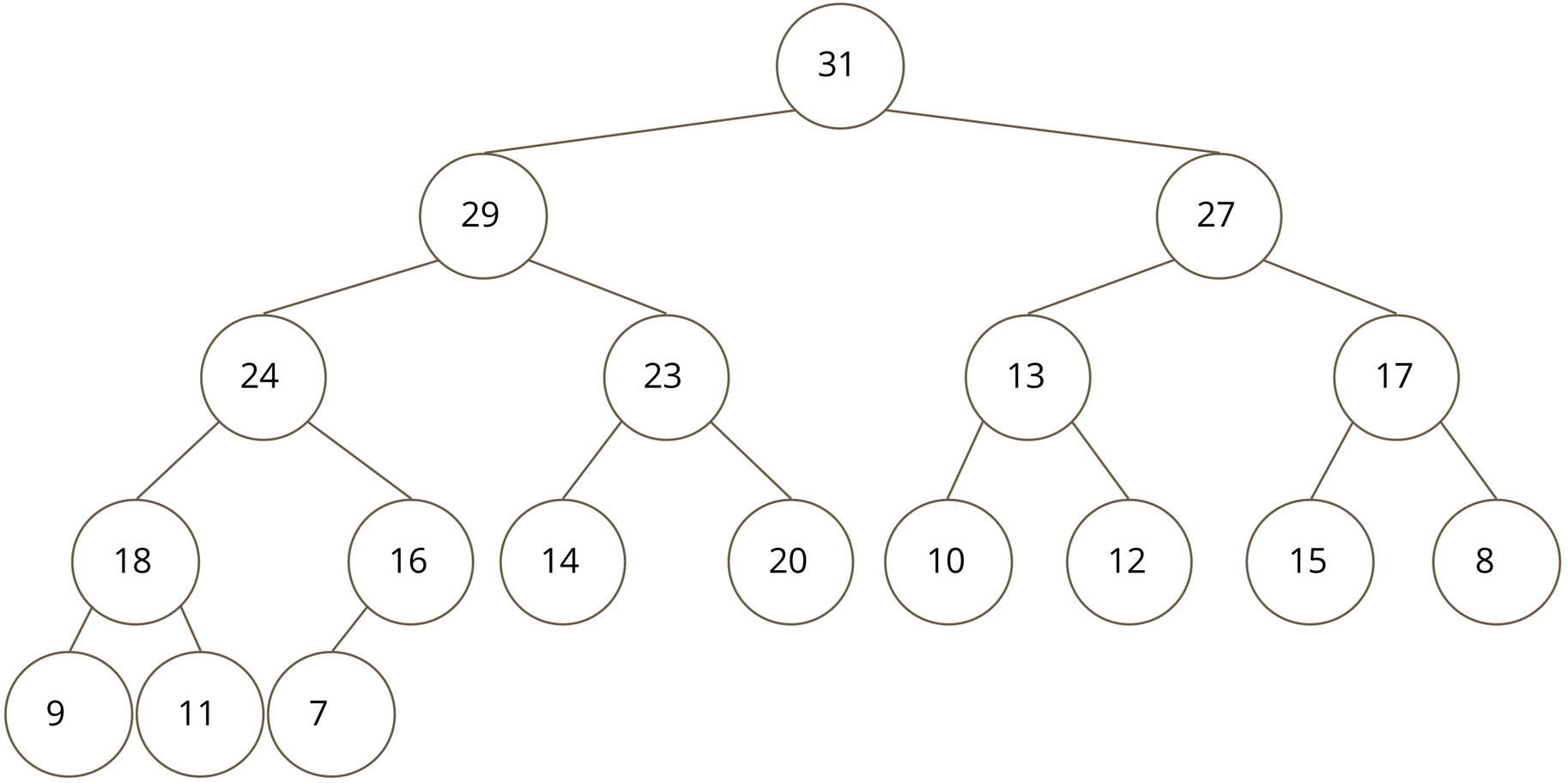
valor	31	30	19	7	29	13	17	5	3	11	2					...
Posicion	0	1	2	3	4	5	6	7	8	9	3	11	12	13	14	...

¿Qué ganamos?

	Con una lista ligada	Con un array
Extracción del elemento más prioritario	O(n)	O(1)
Insertar manteniendo or d en	O(n)	O (l og(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	2	1	1	1	8	9	1	7		
posición	0	1	2	3	4	5	6	7	8	9	0	0	2	5	14	15	1	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

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

   →Hay que subir el nodo

3. 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⌋]:

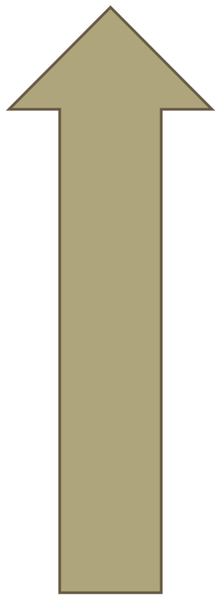
`ShiftUp`(H, i)

else if H tiene hijos **and** H tiene menor prioridad que alguno de ellos:

`Shiftdown`(H, i)

Max Heap

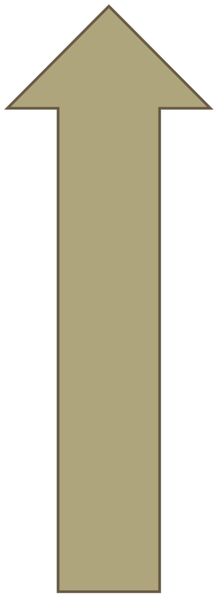
prioridad mayor



prioridad menor

Min Heap

prioridad menor



prioridad mayor

Ejercicio

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 número secuencial ascendente, los que son invocados en orden 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`** quesolicitaatenciónconsecuencia **`sec`** entregadaporeltótem.Elsistemautiliza 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**.

SOLU CI ÓN:

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**alfinaldelalistaligada**A**,y**Remove(A, 0)**retornaelprimer 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 número secuencial ascendente, los que son invocados en orden 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

SOLU CIÓN: 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 número secuencial ascendente, los que son invocados en orden 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.