Ayudantía N° 6

Heapsort

Repaso

Definición

Qué son las colas de prioridad?

Una **cola de prioridad** es una estructura de datos que permite ordenar sus elementos dada una caracteristica, asegurando que el elemento más prioritario este primero.

Por ejemplo, si van a una caja de supermercado, esta tiene una cola de personas, las cuales se ordenan según hora de llegada, quien haya llegado antes (el más prioritario) va primero.

Definición

Cómo las implemetamos?

Queremos ser capaces de insertar y eliminar elementos de manera eficiente, además de modificar la prioridad de un elemento y que no se rompa el orden de nuestra estructura.

Para eso vamos a querer usar arreglos y no listas ligada, ya que vamos a querer no solo buscar elementos más prioritarios, sino que también vamos a querer acceder a índices no necesariamente continuos lo que sale más caro en listas ligadas.

Definición

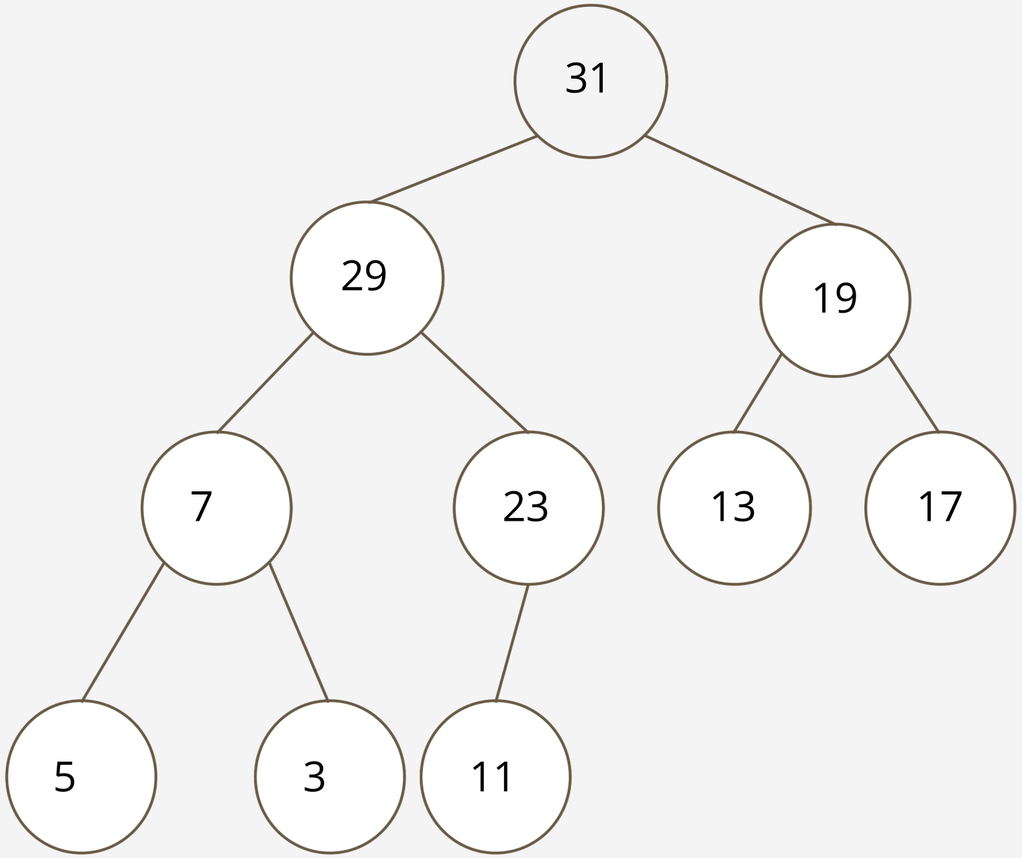
Heaps

Los heaps van a ser arreglos donde su índice va a representar su posición en un árbol binario. De esta manera cuando queramos saber el padre de un nodo o sus hijos lo podremos hacer mediante sus índices.

Estos arreglos representan colas de prioridad, por lo que todas las operaciones que hagamos sobre ellos van a asegurar que el orden prioritario no se rompa.

Heaps

Ejemplo



Valor	31	29	19	7	23	13	17	5	3	11
Índice	0	1	2	3	4	5	6	7	8	9

Pseudocódigo

Cómo extraer el elemento más prioritario?

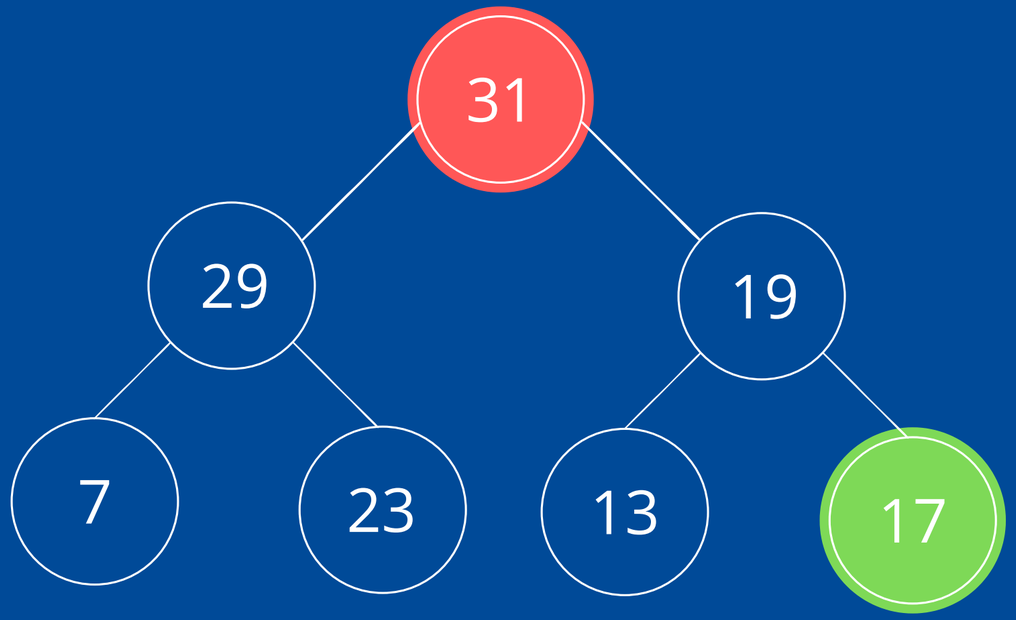
```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H, 0)
    return best
```

```
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)
```

Extracción

1°

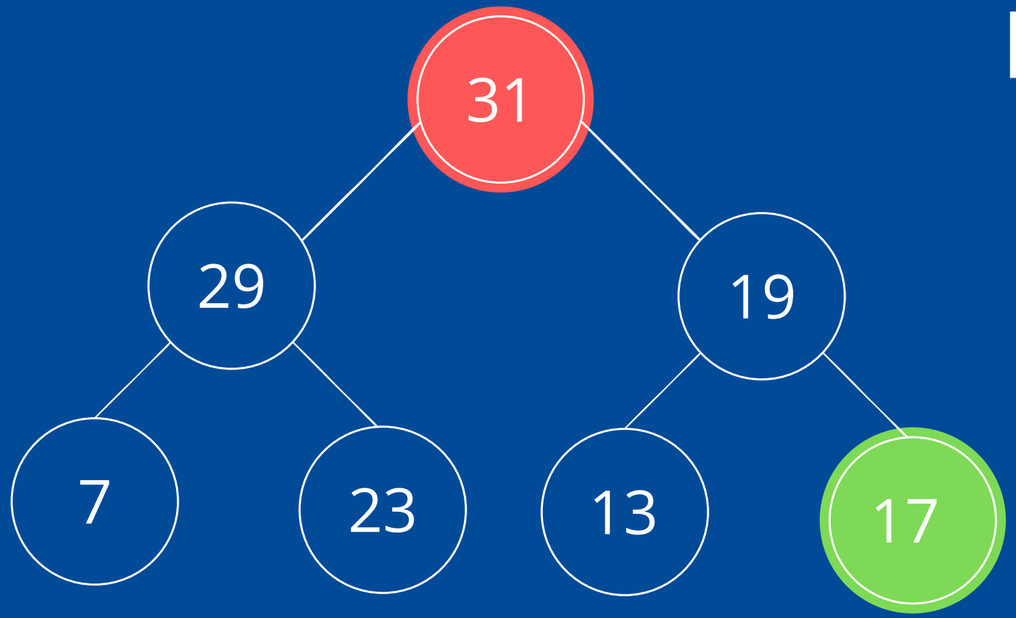
Marcamos el elemento a eliminar y el último elemento no vacío del arreglo



Valor	31	29	19	7	23	13	17
Índice	0	1	2	3	4	5	6

Extracción

2°
Guardamos en otro lugar el elemento prioritario.



BEST: 31

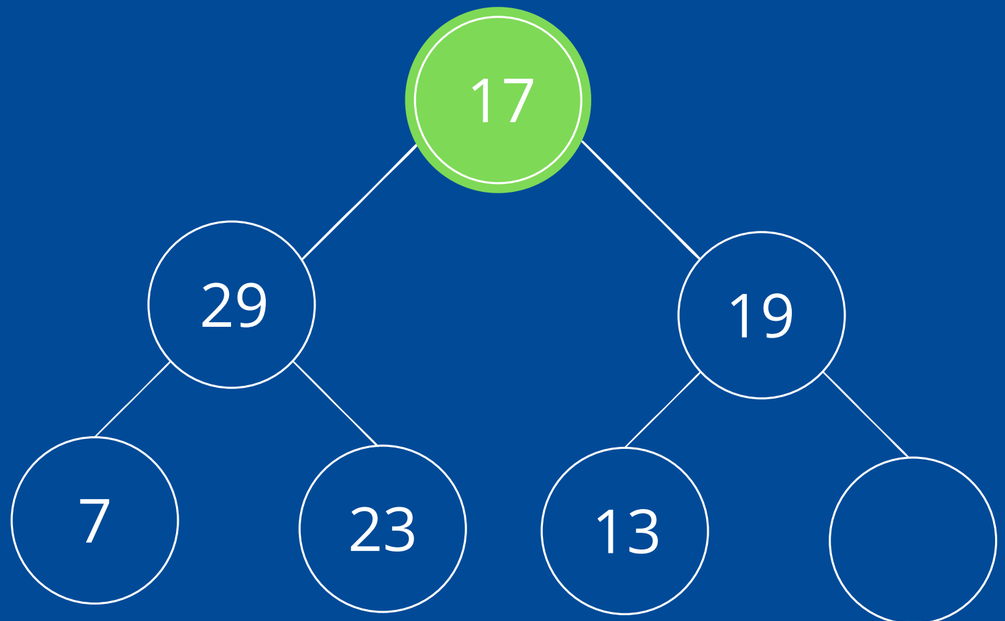
Valor	31	29	19	7	23	13	17
Índice	0	1	2	3	4	5	6

Extracción

BEST: 31

3°

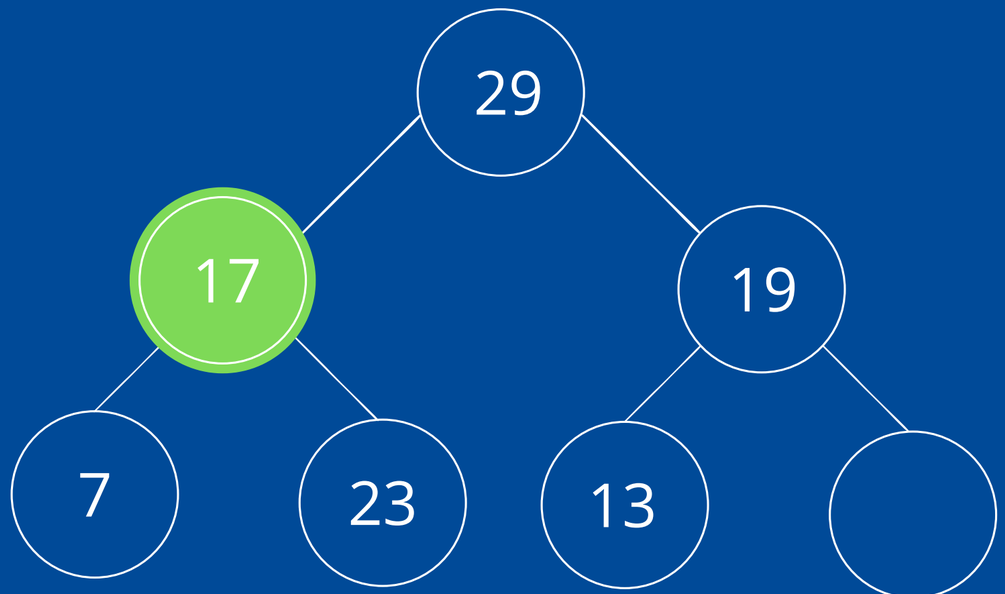
Hacemos que el último elemento sea el primer elemento del heap y borramos su posición anterior.



Valor	17	29	19	7	23	13	∅
Índice	0	1	2	3	4	5	6

Extracción

BEST: 31



4°

Ahora, arreglamos el orden del heap, cambiando el elemento verde hasta que estén bien ordenado.

Recordar que cambiamos con 29 porque es el hijo más prioritario

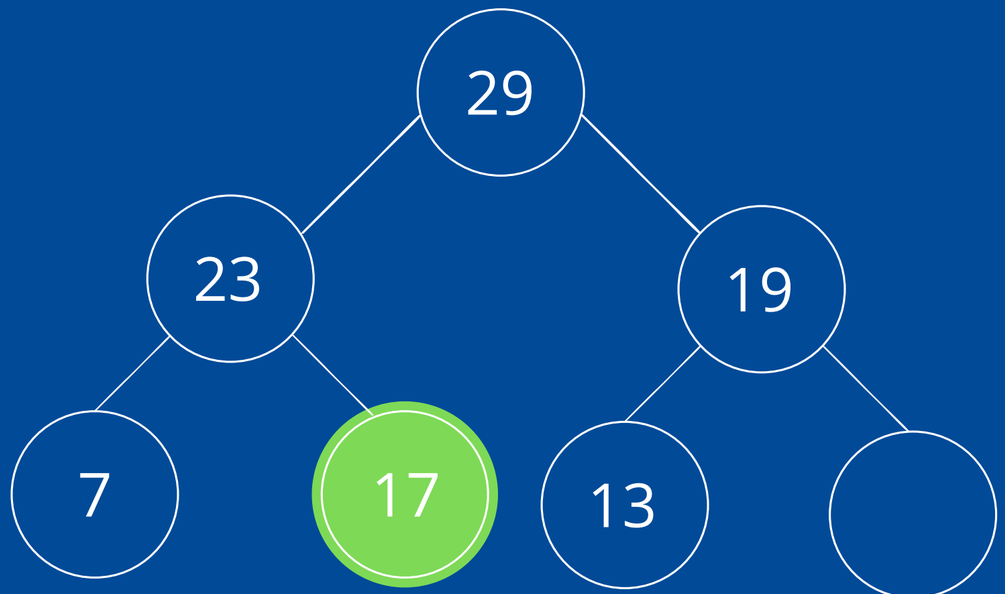
Valor	29	17	19	7	23	13	∅
Índice	0	1	2	3	4	5	6

Extracción

BEST: 31

4°

Ahora, arreglamos el orden del heap, cambiando el elemento verde hasta que estén bien ordenado.



Valor	29	23	19	7	17	13	∅
Índice	0	1	2	3	4	5	6

Pseudocódigo

Cómo insertar un nuevo elemento?

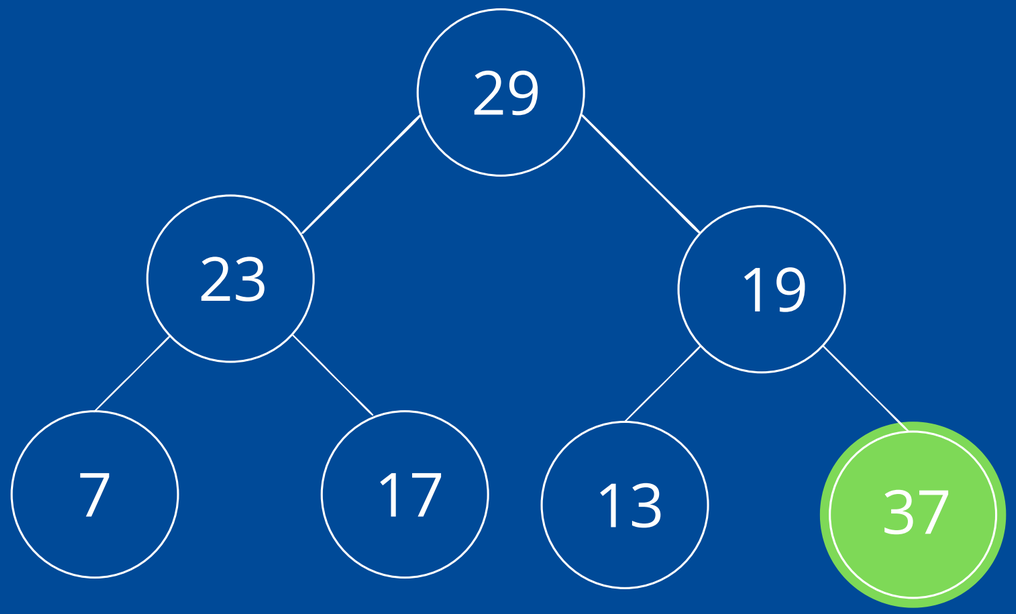
```
Insert(H):
    i ← primera celda vacía de H
    H[i] ← e
    SiftUp(H, i)
```

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

Inserción

1°

Agregamos el nuevo elemento a la última posición vacía en el heap.

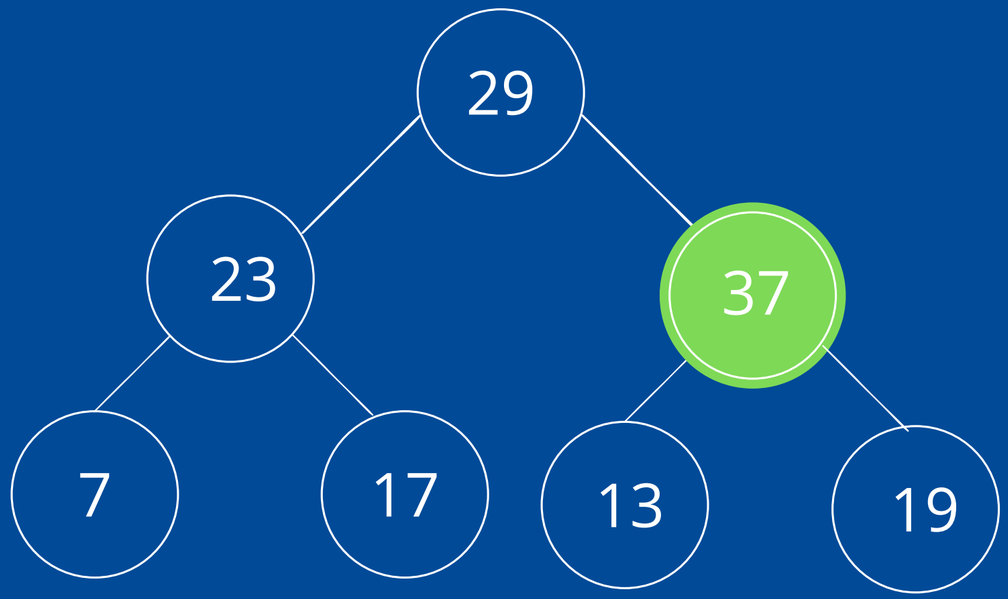


Valor	29	23	19	7	17	13	**37**
Índice	0	1	2	3	4	5	6

Inserción

2°

Ahora, arreglamos el orden del heap, cambiando el elemento verde hasta que estén bien ordenado.

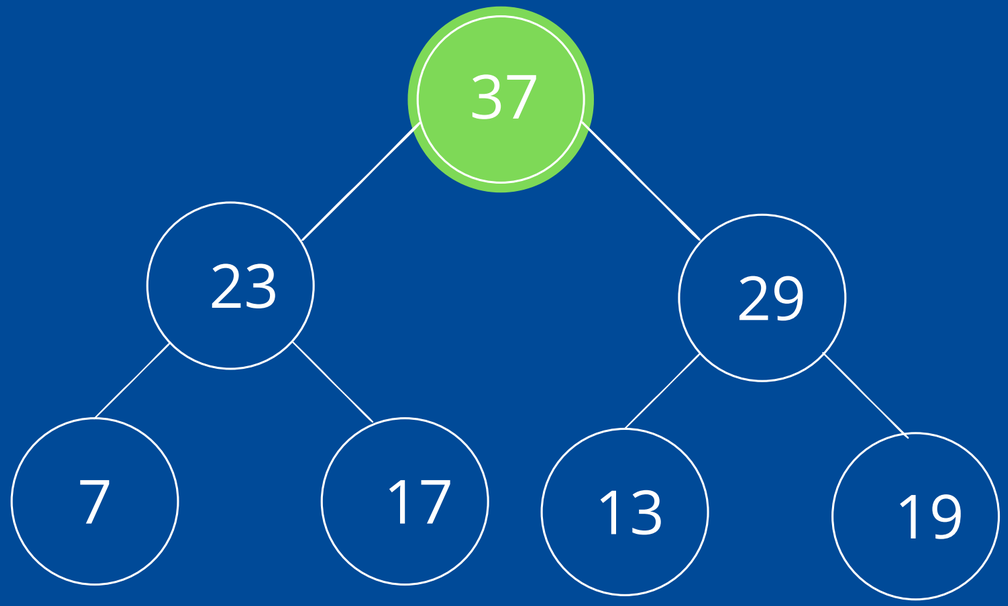


Valor	29	23	**37**	7	17	13	19
Índice	0	1	6	3	4	5	2

Inserción

2°

Ahora, arreglamos el orden del heap, cambiando el elemento verde hasta que estén bien ordenado.



Valor	**37**	23	29	7	17	13	19
Índice	6	1	0	3	4	5	2

Definición

Tipos de heaps

La prioridad puede ser lo que ustedes quieran, siempre y cuando sea comparable para que genere un orden. Pero si lo pensamos si existe una prioridad, voy a querer obtener sólo valores extremos (más prioritarios)

1. Max Heaps: Donde el valor **más prioritario** es el primero dentro de los elementos. Por ejemplo: Cola de supermercado, el más prioritario es el que lleva **más tiempo de espera.**

2. Min Heaps: Donde el valor **menos prioritario** es el primero dentro de los elementos. Por ejemplo: Ruta para un repartidor, primero va a los lugares que tiene a **menor distancia.**

Definición

HeapSort

Si tenemos un max o min heap, sobre un arreglo cualquiera que guarda número, sabemos siempre donde está el menor o mayor elemento, entonces queremos aprovecharnos de que sabemos donde está ese elemento para dejarlo en su lugar correcto.

Para eso vamos a decir que los arreglos que se comportan como heap tienen un atributo llamado heap_size, que parte siendo n.

HeapSort

Pseudocódigo

```
input : arreglo A[0 ... n − 1]

BuildHeap(A):
    for i = ⌊n/2⌋ − 1 ... 0 :          ▷ loop decreciente
        SiftDown(A, i)
```

```
input : arreglo A[0 ... 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

Pseudocódigo

Ojo, que se deje el elemento ordenado en el i-ésimo lugar del arraglo sucede unicamente porque este es un **max heap**.

Si fuese un **min heap**, nos gustaría dejar los elementos más pequeños desde el índice 1 en adelante.

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

```
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)
```

Ejercicio 1

Escribe un algoritmo llamado Update(A, i, v), el cual debe actualizar el valor del nodo en la posición i por v y mantener el orden en el heap A.

Pregunta 1

Pregunta 1 - Solución

¿Qué pasa si un nodo cambia de valor?

Hay dos escenarios principales:

- No rompe el orden del heap
- Rompe el orden del heap

De qué forma puede romper el orden?

1. Tiene una prioridad mayor a su padre.
2. Tiene una prioridad menor a su hijo.

Entonces tenemos que hacer un algoritmo que compruebe que estas dos cosas no ocurran y las arregle si ocurren.

Pero nosotros ya sabemos que hacer en ambos escenarios :o

Pregunta 1 - Solución

```
Update(A,i,v):
  nodo_actual = A[i]
  padre = A[(i-1)//2]
  if (nodo_actual tiene padre and nodo_actual > padre):
    SiftUp(A,i)
  else:
    tiene_menos_prioridad = nodo_actual < hijo_1 or nodo_actual < hijo_2
    if nodo_actual tiene hijos and tiene_menos_prioridad:
      SiftDown(A,i)
```

Ejercicio 2

En una farmacia de barrio donde llegan a lo más 99 clientes al día, existe un tótem que entrega 3 categorías de turnos, estos son A de atención normal, D de delivery y T de tercera edad. Cada cliente, cuando llega, saca ticket (turno) que correspondiente a su categoría con un número que va aumentando en cada categoría (Por ejemplo, A1, A2, T1, D1, A3, etc.). Existen 2 personas atendiendo (caja) y cada vez que terminan de atender, llaman al siguiente turno (Next) que se selecciona según su prioridad, donde T es más prioritario que D y el menor es A. Indique:

Pregunta 3 - I1-2025-1

a) (2 pts) Qué estructura de datos eficiente usaría para esta situación, indicando la forma en que se ordenan los tickets y las restricciones que aplican.

Pregunta 3a - I1-2025-1

Pregunta 3a - I1-2025-1 - Solución

Sabemos que queremos ordenar T→D→A.
Por lo que lo podemos plantear de dos maneras:
Un min heap donde A ocupe valores muy altos y T muy bajos.
Un max heap donde A ocupe valores muy bajos y T muy altos.
Vamos a ir con la opción donde A ocupa valores muy altos.

Vamos a decir que un ticker tiene una categoría T,D,A y un id, el número que le sigue, entonces podemos definir:

```
Prioridad(categoria, número):
  if categoria == T:
    return (1000-número)
  if categoria == D:
    return (100-número)
  return número.
```

Prioridad(T5)= (1000-5) = 995

b) (2 pts) Escriba en Pseudocódigo la función Ticket → turno. Fíjese en que se debe seleccionar la categoría.

Básicamente la función del sistema que otorga tickets.
Es decir *Ticket(c)*, donde c representa la categoría de la persona (T,D,A).

Pregunta 3b - I1-2025-1

Pregunta 3b - I1-2025-1 - Solución

Dadas las variables globales T_COUNT, D_COUNT, y A_COUNT que representan un contador por categoría y un único Max heap llamado H.

Además, se considera la función *prioridad(T)* que recibe un ticket y retorna su prioridad.

Pregunta 3b - I1-2025-1 - Solución

```
Ticket(c):
  if c == T:
    T_COUNT += 1
    prioridad_ticket = prioridad(c, T_COUNT)
    ticket_name = T + T_COUNT // Ej: T1
  else if c == D:
    D_COUNT += 1
    prioridad_ticket = prioridad(c, D_COUNT)
    ticket_name = D + D_COUNT
  else if c == A:
    A_COUNT += 1
    prioridad_ticket = prioridad(c, A_COUNT)
    ticket_name = A + A_COUNT
  insert(H, prioridad_ticket)
  return ticket_name
```

Ejercicio 3

Demuestra, mediante cálculos, que BuildHeap es O(n)

Pregunta 1 - I1-2020-1



Pregunta 3b - I1-2025-1 - Solución

Sabemos que la cantidad de nodos en un piso está dada por:
(Donde el piso 0 es el más poblado y el log(n) es el nodo más prioritario)

$$nodos_h = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

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

`BuildHeap`(A):
  for $i = \lfloor n/2 \rfloor - 1 \ldots 0$:
    `SiftDown`(A, i)

Además, sabemos que *SiftDown* en su peor escenario hace h pasos. Y por último, el índice desde donde parte el ciclo representa el nodo de más a la derecha del penúltimo piso. Por lo que el peor escenario está dado por la suma de todos los peores escenarios de *SiftDown* por cada piso. Es decir:

$$O(\sum_{h=1}^{log(n)} \left\lceil \frac{n}{2^{h+1}} \right\rceil * h$$

Pregunta 3b - I1-2025-1 - Solución

Nosotros sabemos que $\boxed{\lceil x \rceil \leq x + 1}$ entonces lo que acota al término de la derecha necesariamente también acota al de la izquierda, entonces podemos hacer:

$$\boxed{O(\sum_{h=1}^{log(n)} (\frac{n}{2^{h+1}} + 1) * h}$$

$$\boxed{O(\sum_{h=1}^{log(n)} (\frac{n * h}{2^{h+1}} + h)}$$

$$\boxed{O(\sum_{h=1}^{log(n)} (\frac{n * h}{2^{h+1}}) + O(\sum_{h=1}^{log(n)} (h)}$$

Pregunta 3b - I1-2025-1 - Solución

Sumatoria de los primeros h términos es un viejo conocido...

$$O(\sum_{h=1}^{log(n)}(\frac{n*h}{2^{h+1}}) + O(\frac{log(n)*(log(n)+1)}{2})$$

$$O(\sum_{h=1}^{log(n)}(\frac{n*h}{2^{h+1}}) + O(log^2(n))$$

Por fin, después de simplificar varias veces tenemos:

$$O(\frac{n}{2} * \sum_{h=1}^{log(n)}(\frac{h}{2^h}) + O(log^2(n))$$

$$O(\frac{n}{2} * 2) + O(log^2(n))$$

$$O(n)$$

Pregunta 3b - I1-2025-1 - Solución

Extra matemático:

$$\sum_{x=1}^{\infty}(x * r^x) = \frac{r}{(1-r)^2}$$

$$\sum_{x=1}^{\infty}(x * ((\frac{1}{2})^x) = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2$$

Feedback