



UNIVERSITAT DE BARCELONA



Estructura de datos

Heaps

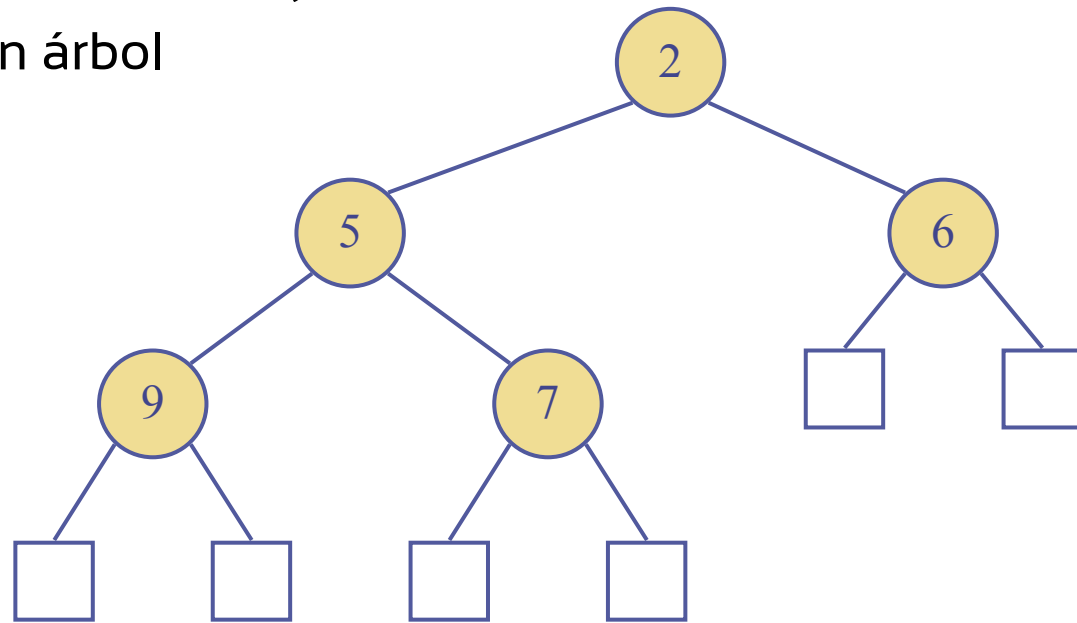
Santi Seguí | 2014-15

Índice

- ¿ Qué son los Heaps?
 - Insert y upheap
 - removeMin y downheap
 - Como implementar los heaps?
- HeapSort

Heaps

- Estructura de datos utilizada para implementar una cola de prioridad
 - `insert(key, element)`
 - `removeMin()`
- Se puede implementar utilizando árboles (link-based) o mediante un array
 - Analicemos primero un árbol



Recordatorio : Cola prioridad

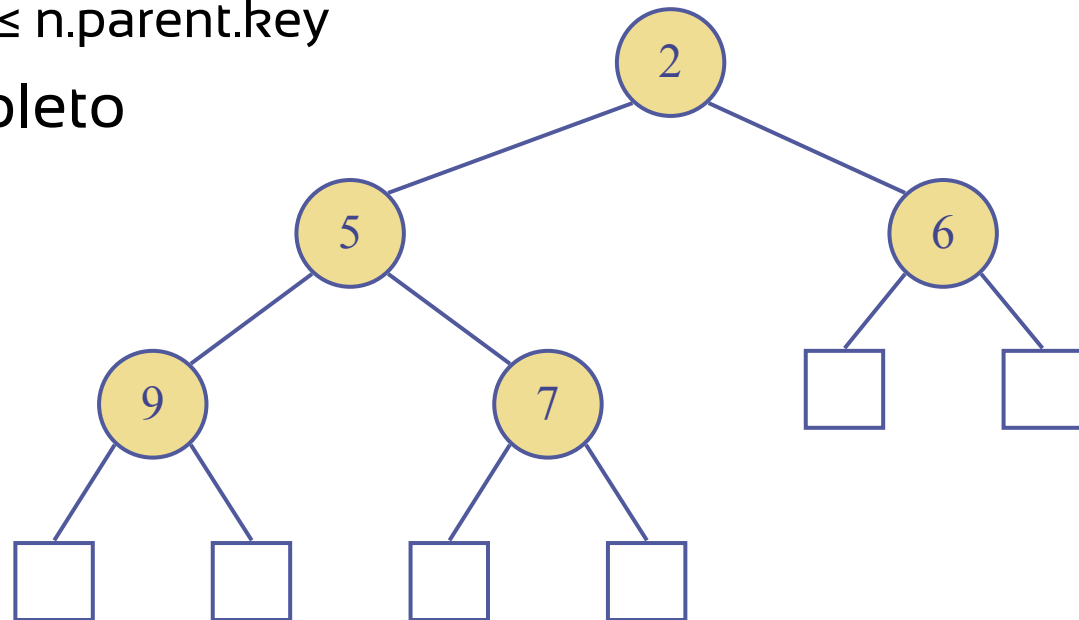
- Motivación:
 - Una diversidad de problemas son resueltos utilizando una gran colección de datos donde cada ítem tiene una prioridad.
 - Ejemplos:
 - Salidas de vuelos
 - Los distintos vuelos necesitan la pista de despegue
 - Hay vuelos que tiene más prioridad que otros
 - Gestión del ancho de banda
 - Los datos de alta prioridad (real-time data como skype) han de transmitirse primero

TAD : Cola de Prioridad

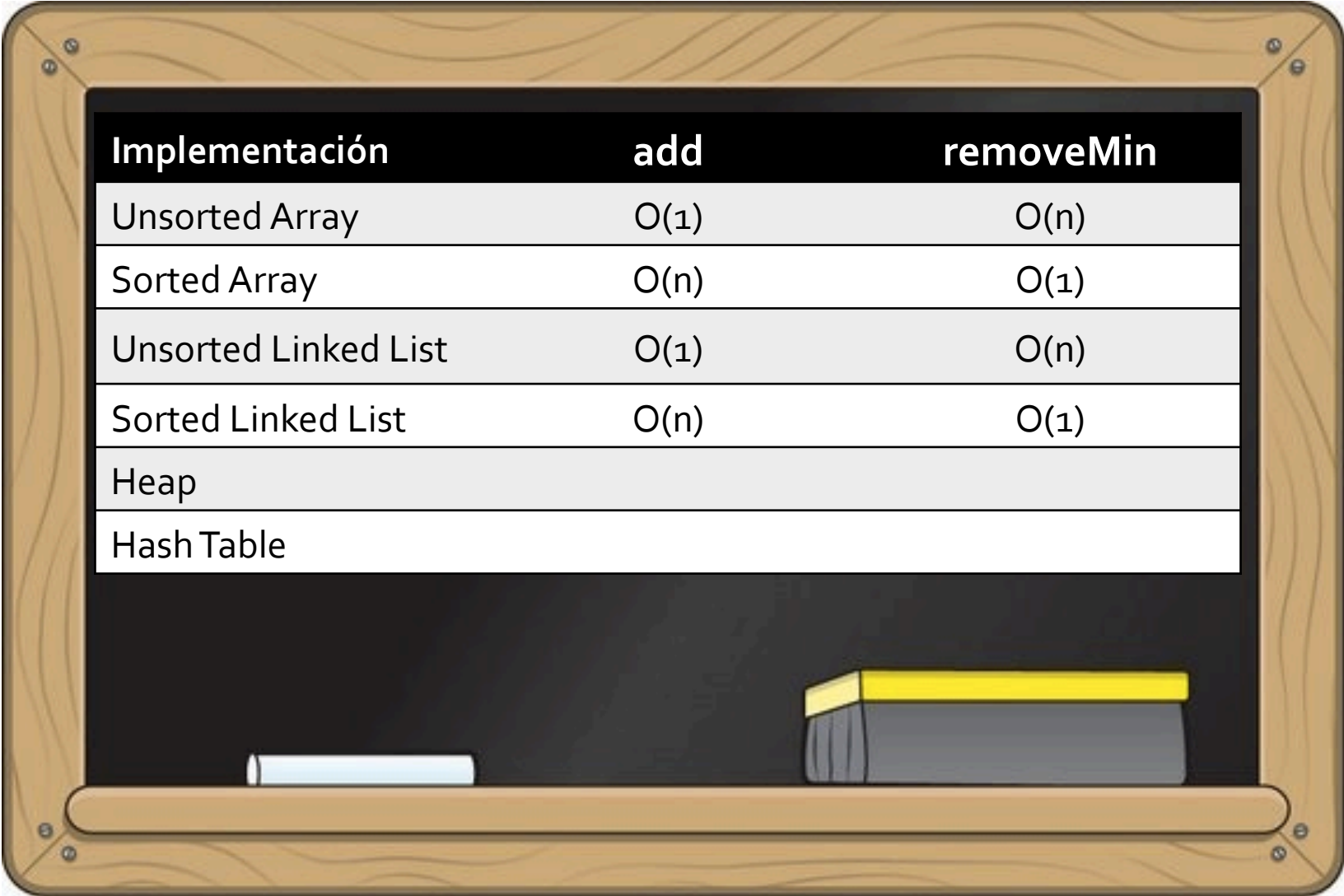
- Una cola de prioridad guarda una colección de ítems
- Un ítem es una pareja de: (clave, elemento)
 - La clave define la posición del elemento dentro de la cola.
- Métodos principales:
 - `insert(key, element)`
inserta un ítem con su clave y elemento dentro de la cola.
 - `removeMin()`
Elimina el ítem con la clave menor y retorna el elemento asociado.

Propiedades de los Heaps

- Árbol Binario
 - Cada nodo tiene como mucho dos hijos.
- Clave (o “prioridad”) en cada nodo
- Orden del Heap
 - Para min-heap: $n.key \geq n.parent.key$
 - Para max-heap: $n.key \leq n.parent.key$
- Es un árbol casi completo
- Altura $O(\log n)$



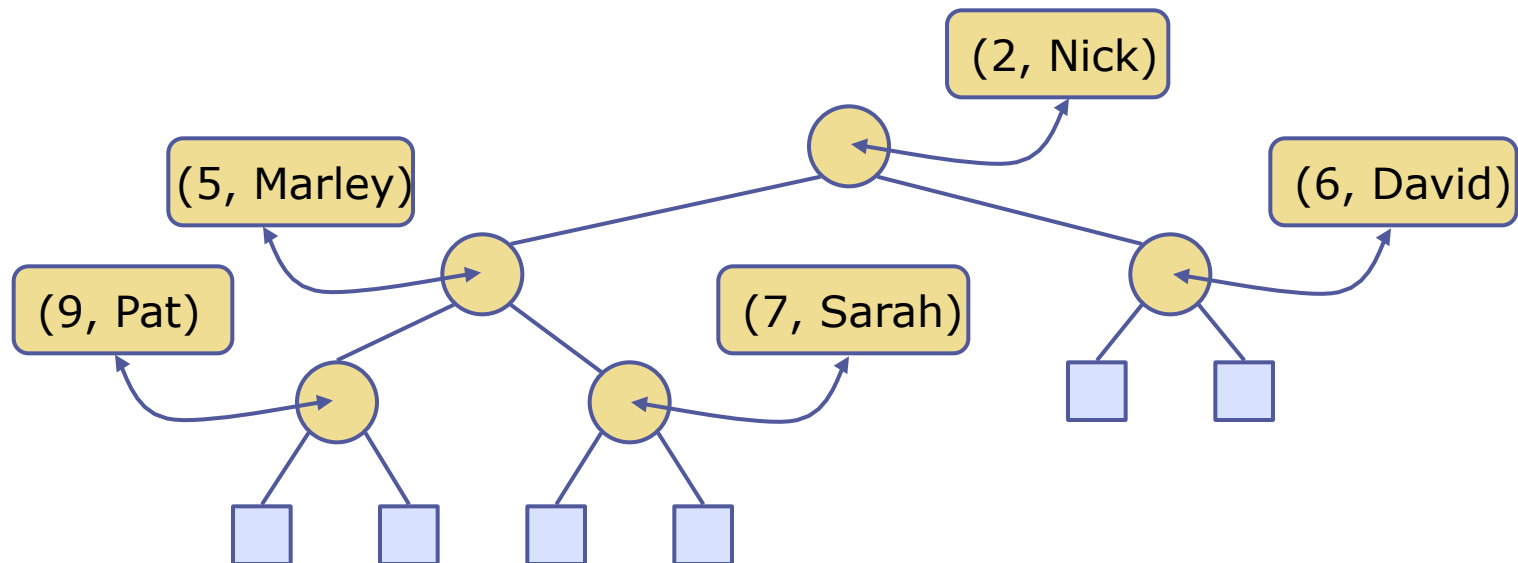
Complejidad Operaciones



Implementación	add	removeMin
Unsorted Array	$O(1)$	$O(n)$
Sorted Array	$O(n)$	$O(1)$
Unsorted Linked List	$O(1)$	$O(n)$
Sorted Linked List	$O(n)$	$O(1)$
Heap		
Hash Table		

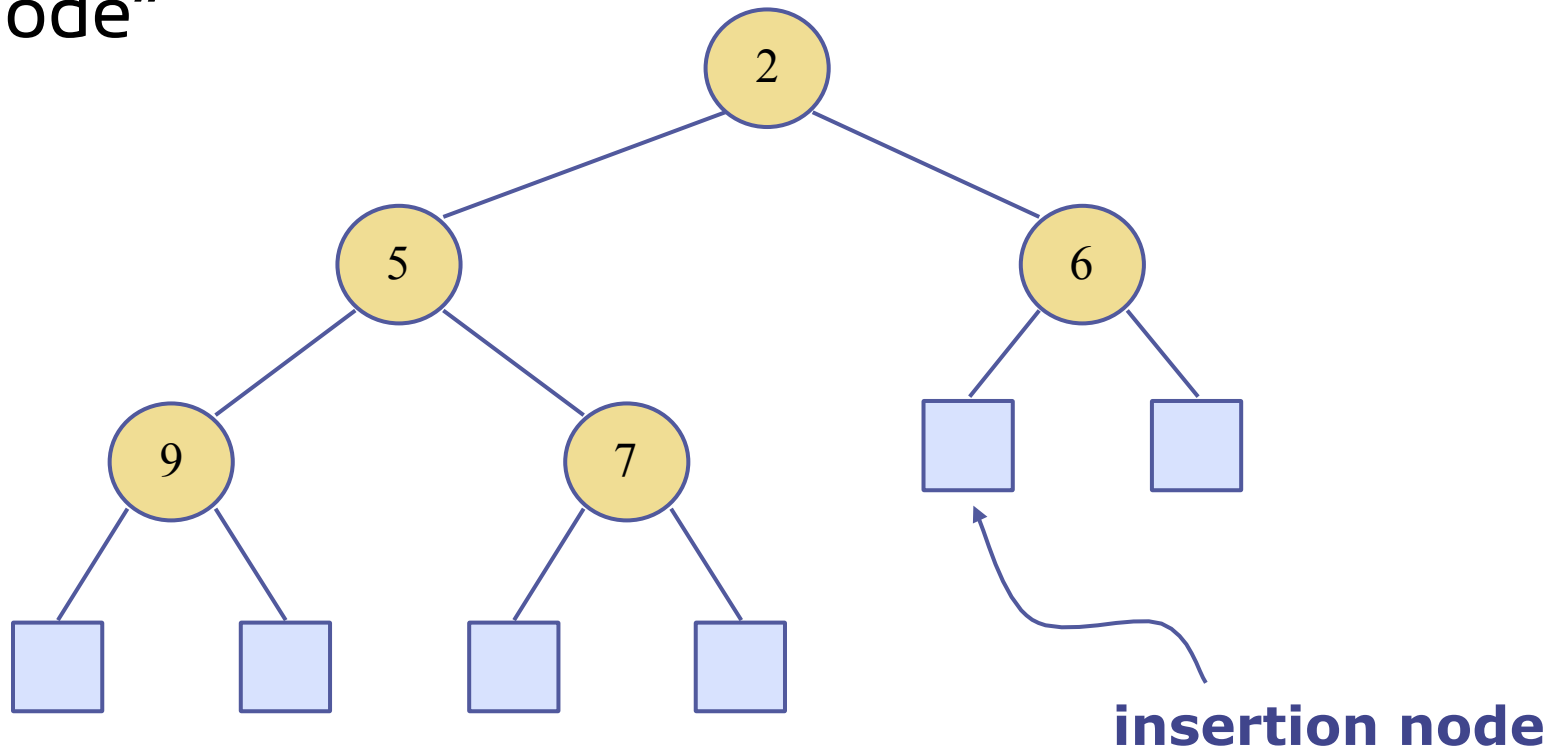
Heaps y Colas de Prioridad

- Podemos utilizar un heap para implementar una cola de prioridad.
- Guardamos un ítem (clave, elemento) en cada nodo



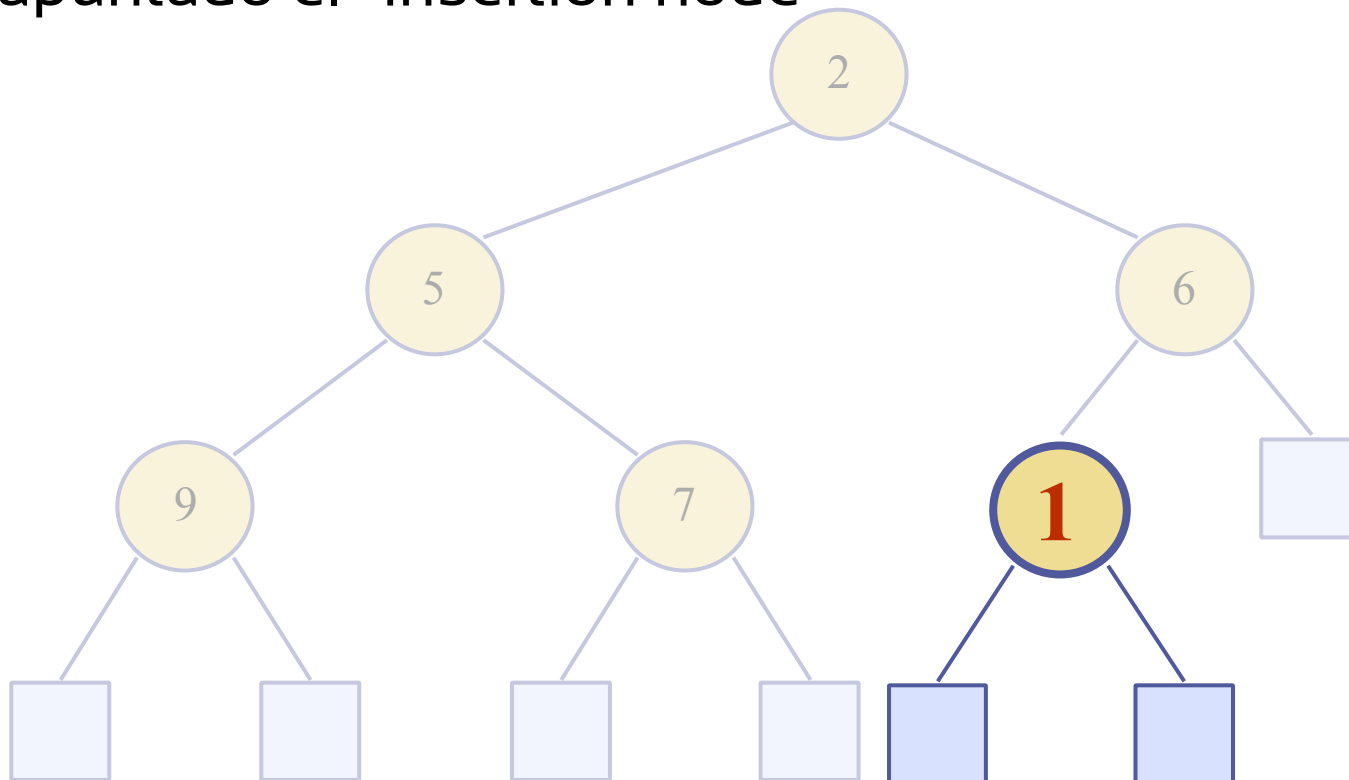
insert()

- Hay que guardar la posición (nodo) donde debemos insertar el siguiente elemento para mantener el árbol semi-completo: "insertion node"



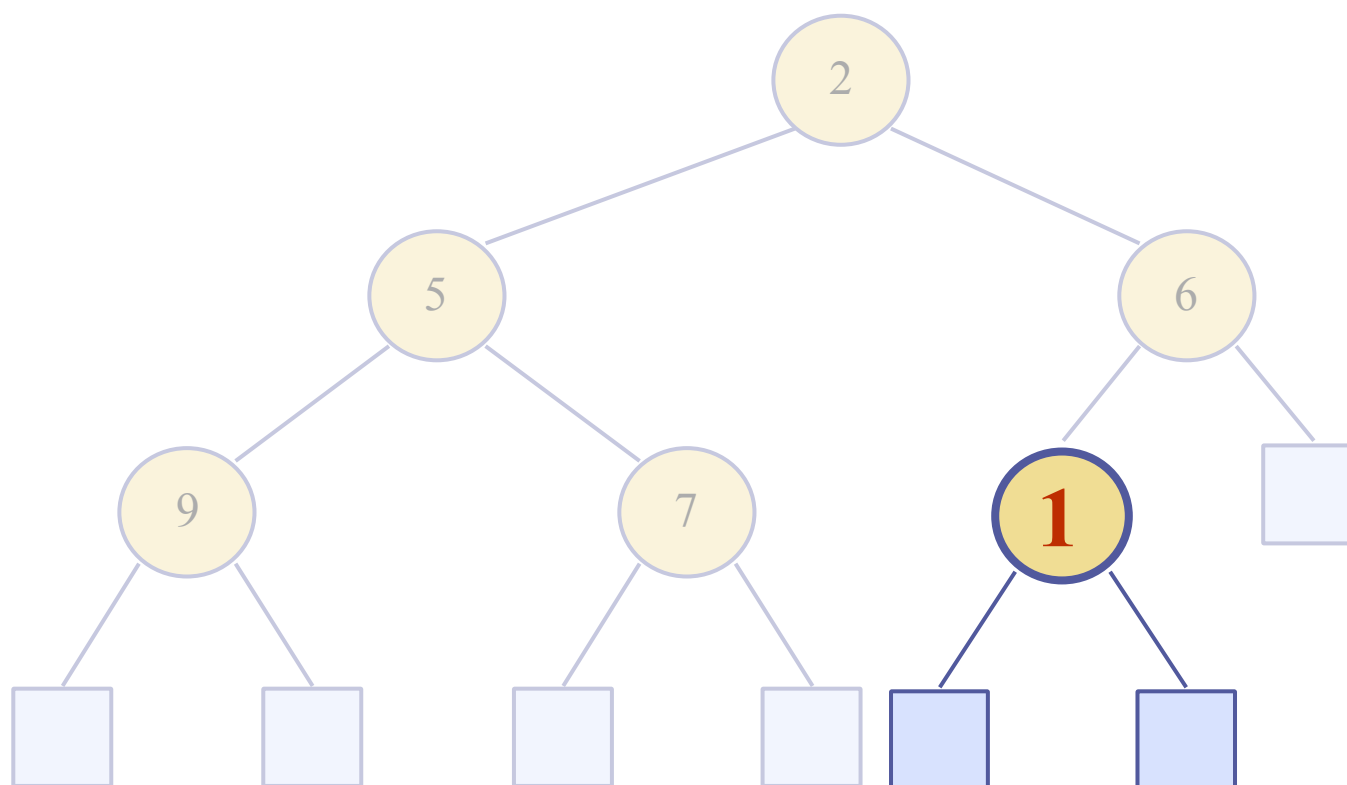
insert() (2)

- Ejemplo: insert(1)
- Insertamos el nuevo nodo allí donde tenemos apuntado el "insertion node"



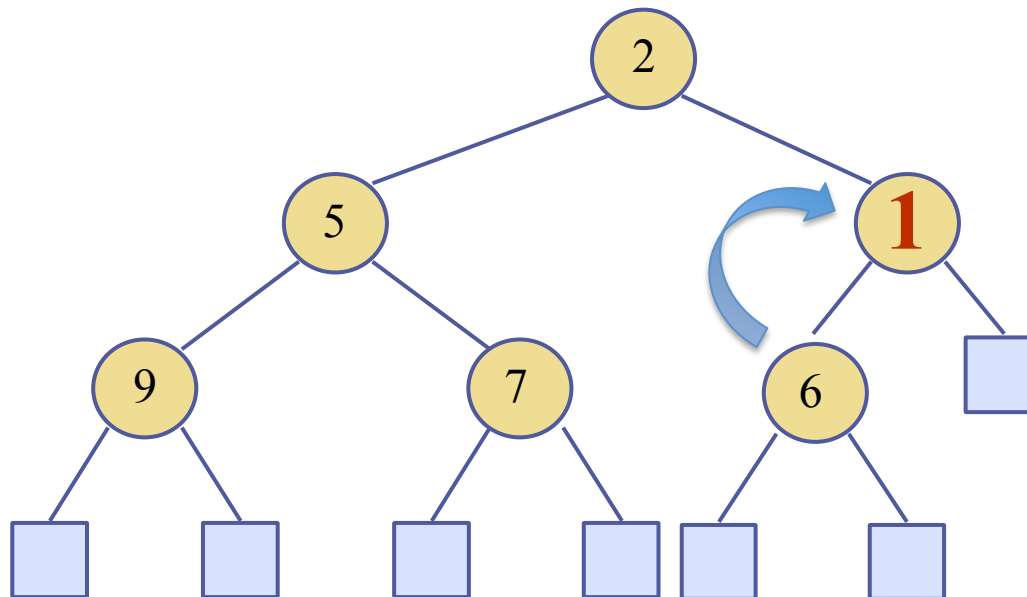
insert() (3)

- Que pasa ahora? El orden del heap puede ser violado!!



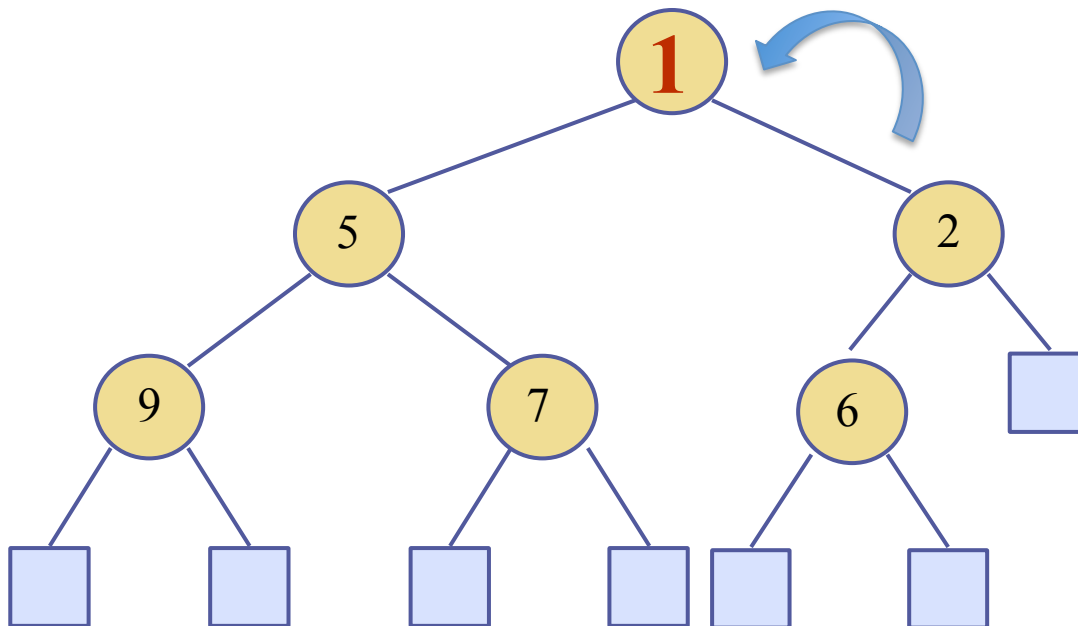
Upheap

- Debemos restablecer el orden del heap intercambiando los ítems hacia arriba hasta que las propiedad del orden del heap sean restablecidas.
- El primer intercambio restablece todo por debajo de la nueva posición



Upheap (2)

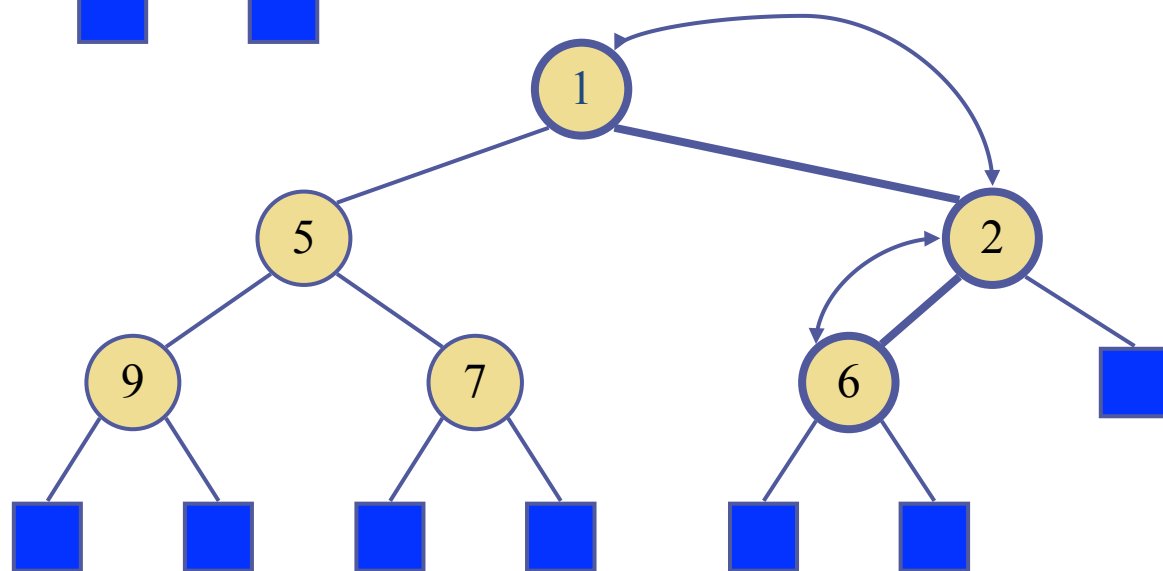
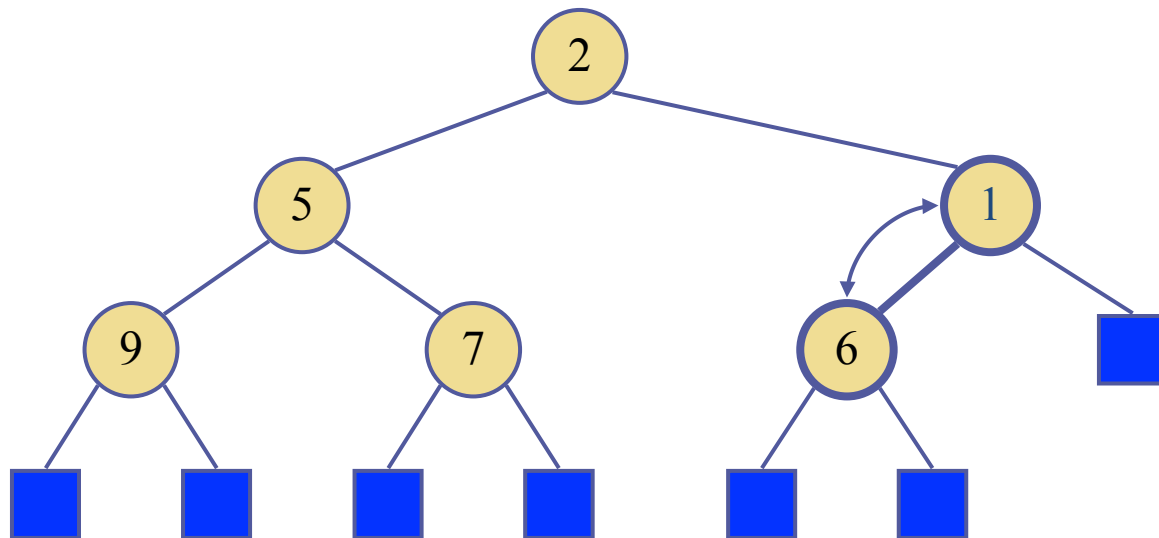
- Debemos hacer otro swap, ya que 1 es menor que el 2 en el nodo superior.
- Finalmente, las propiedades del Heap se satisfacen



Upheap (3)

- Después de insertar un nuevo nodo con clave k , las propiedades del orden del heap pueden ser violadas.
- El algoritmo upheap restablece el orden del Heap intercambiando los nodos en el camino desde el nodo insertado en la raíz del árbol
- Upheap termina cuando la clave llega a la raíz o en un nodo donde su padre tiene una clave más pequeño o igual a k
- Dado que el heap tiene altura $O(\log n)$, upheap tiene una complejidad de $O(\log n)$ en tiempo.

Upheap (4)



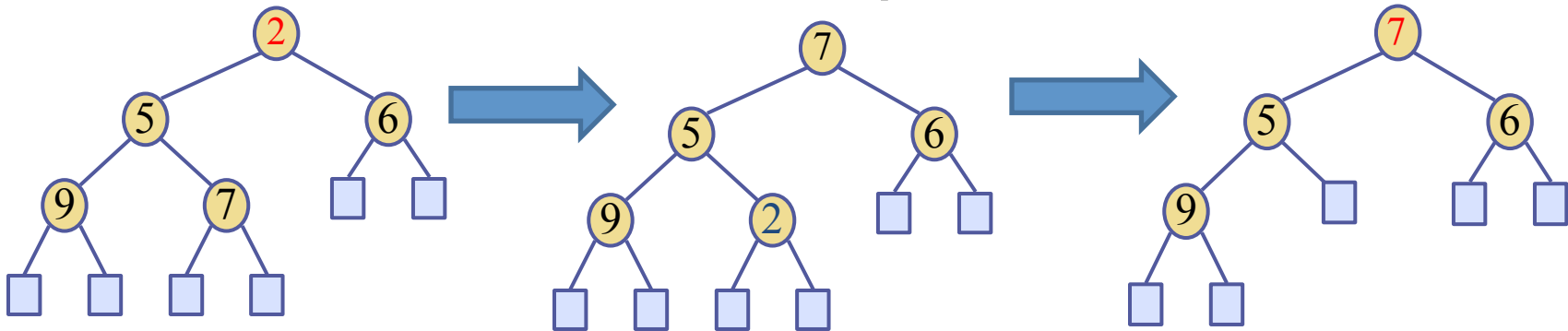
removeMin()

- El elemento mínimo de un heap siempre es la raíz (debido al orden heap)
- ¿Como eliminamos un elemento del heap sin destruir su orden?



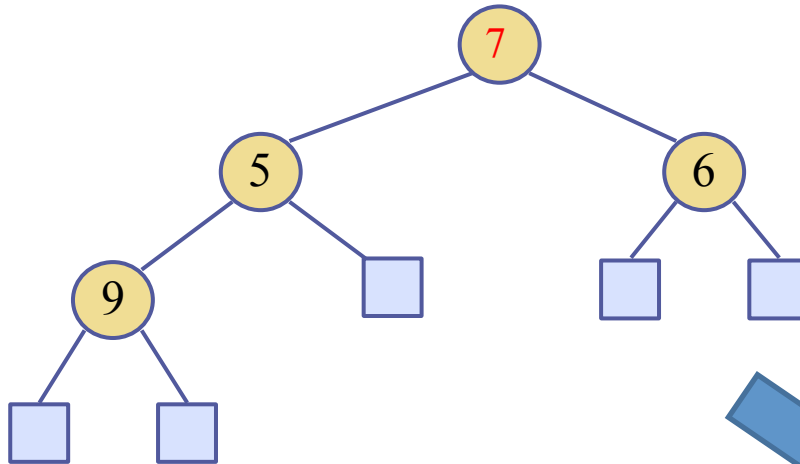
removeMin() (2)

- Intercambiar la raíz (el elemento que queremos eliminar) con el último elemento de heap
 - Eliminar el elemento desde la última posición es fácil
- Pero el orden del heap no se conserva.

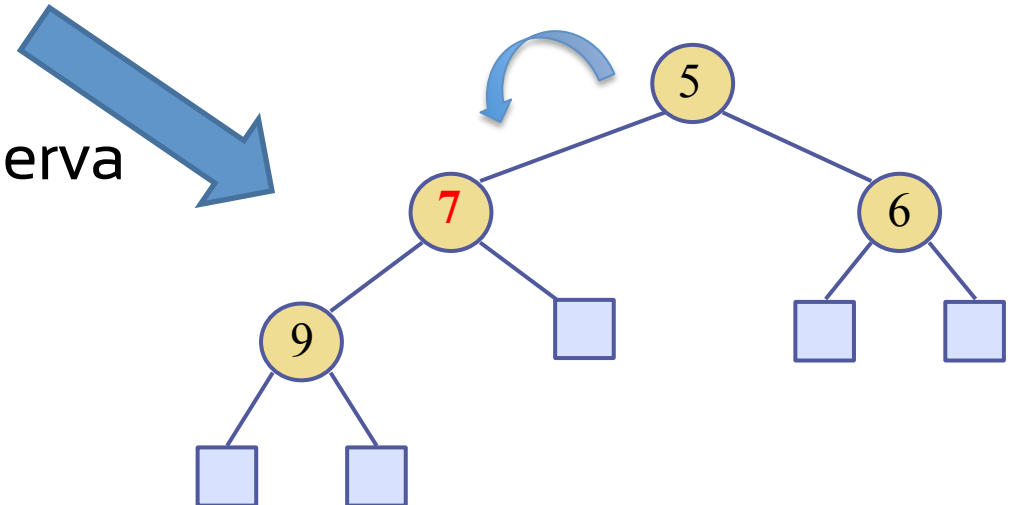


Downheap

- Vamos Bajando el elemento de la raíz hacia abajo tanto como sea necesario.



- Ahora el orden se conserva

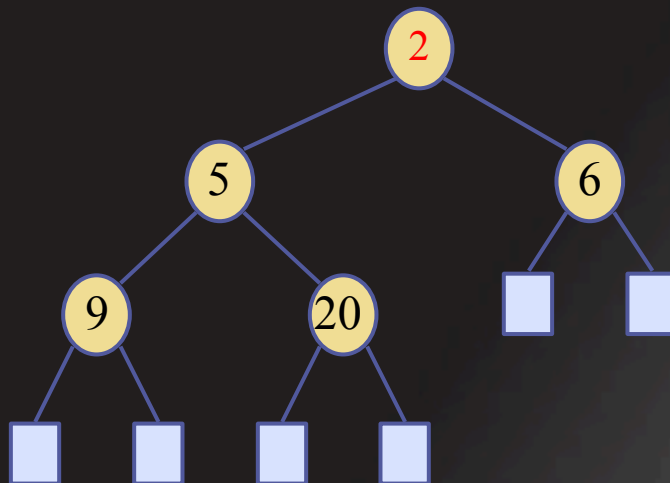


Downheap (2)

- Downheap restablece el orden del heap intercambiando los ítems en una trayectoria descendiente desde la raíz, intercambiándolo por el **menor de sus hijos**.
- Downheap termina cuando la clave k llega a una hoja o en un nodo donde su hijo tenga unas claves mas grandes o iguales que k
- Dado que el heap tiene altura $O(\log n)$, el downheap tiene una complejidad $O(\log n)$ en el tiempo

removeMin()

¿Como queda la estructura si hacemos removeMin()?

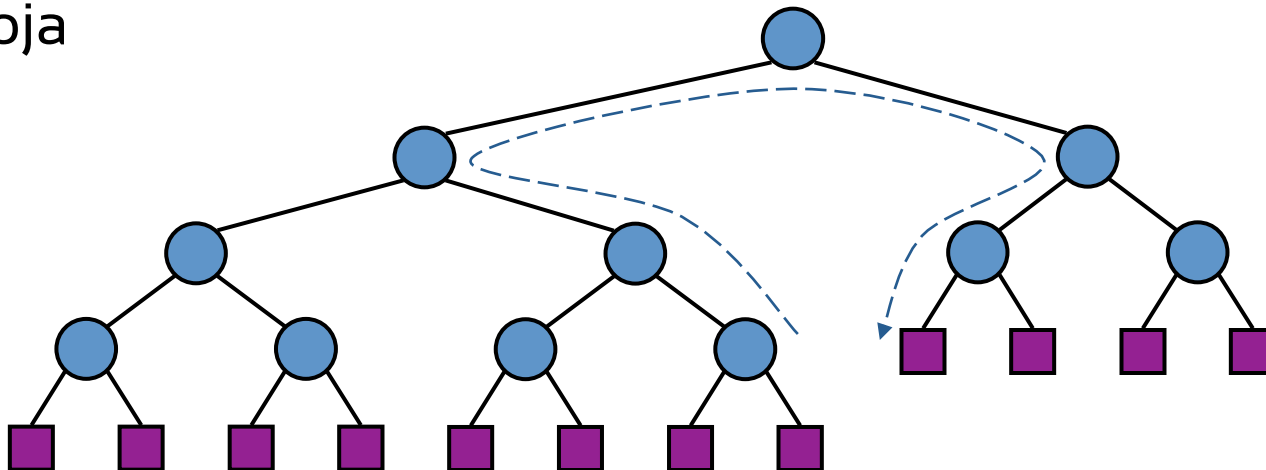


Implementación del Heap: Resumen

- `insert()`:
 - Insertamos un ítem en el “insertion node”, el siguiente espacio libre (tenemos que tener un control de ello)
 - **Upheap** desde abajo hasta que sea necesario
- `removeMin()`:
 - Intercambiar raíz con el último nodo insertado en el heap
 - Eliminamos el nodo raíz intercambiado
 - **Downheap** desde la raíz hasta donde sea necesario

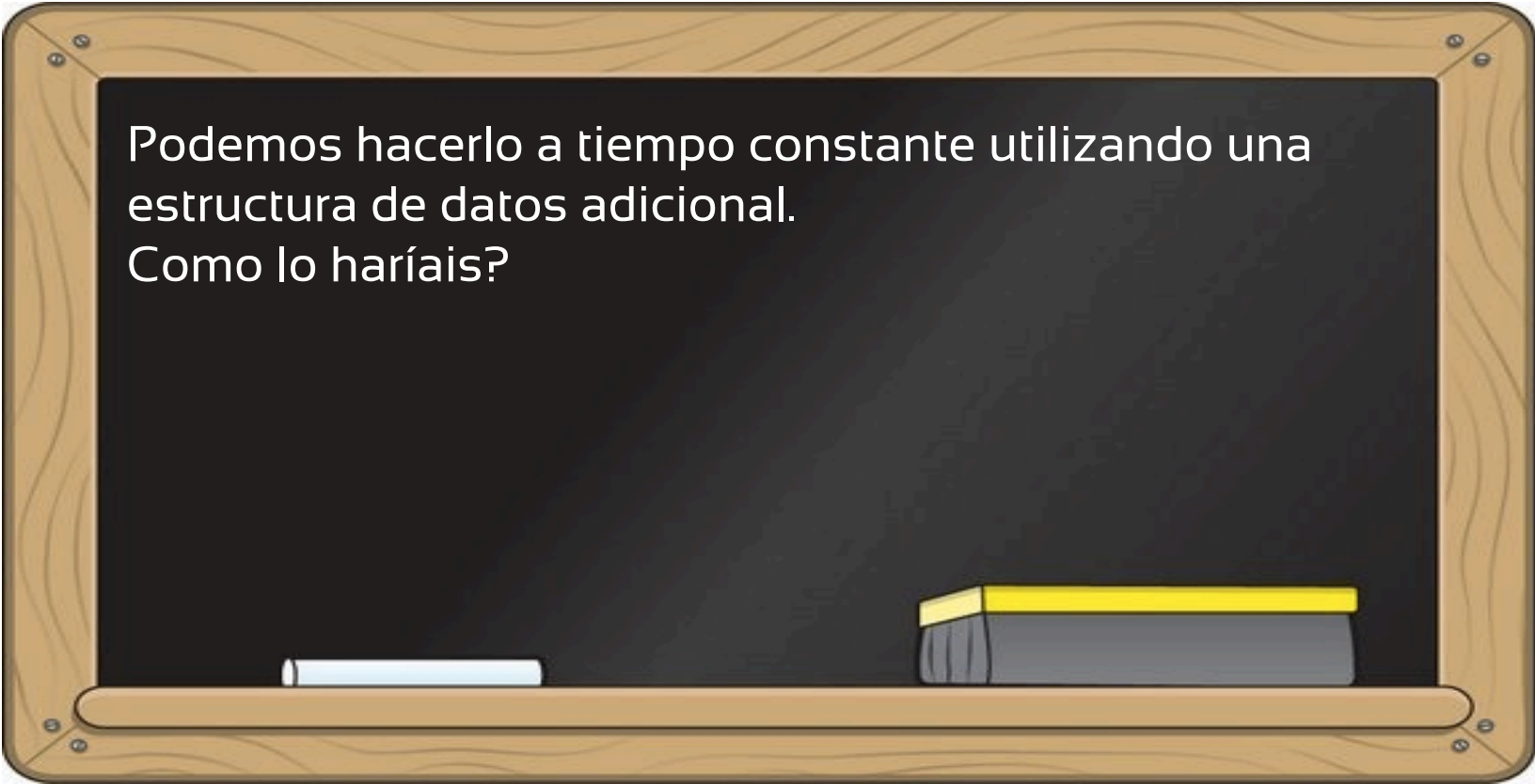
Como encontrar el “Insertion Node”

- El “insertion node” puede ser encontrado recorriendo un camino de $O(\log n)$ nodos:
 - Empezamos con el último nodo añadido.
 - Vamos hacia arriba hasta encontrar un hijo derecho o bien llegamos a la raíz
 - Si nos encontramos en un hijo izquierdo nos situaremos a su hermano (el correspondiente hijo derecho)
 - Vamos hacia abajo hasta encontrar hasta encontrar una hoja



Como encontrar el “Insertion Node” (2)

- Para encontrar el “Insertion Node” hemos visto que tenemos un algoritmo $O(\log n)$.



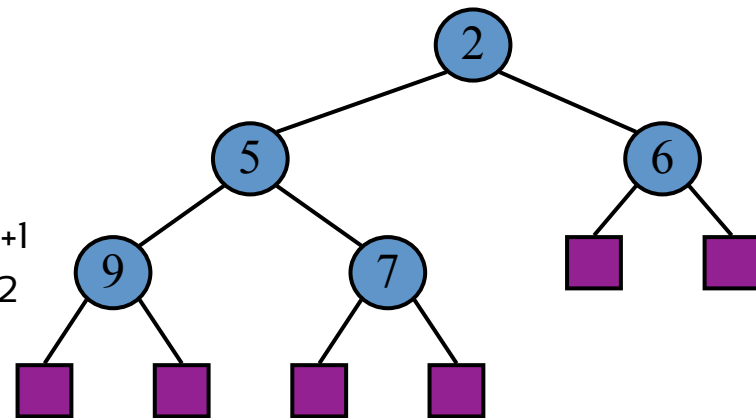
Podemos hacerlo a tiempo constante utilizando una estructura de datos adicional.
Como lo haríais?

Implementación mediante Arrays

- Podemos representar un heap de n claves mediante un array de longitud $n+1$

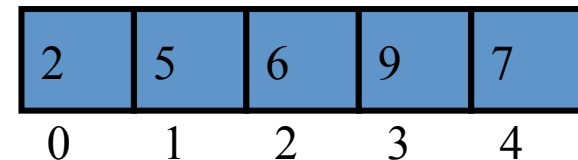
- Implementación

- Para el nodo en índice i
 - El hijo izquierdo se encuentra en la posición $2i+1$
 - El hijo derecho se encuentra en la posición $2i+2$
- Las hojas y los enlaces no tienen que estar representados.



- Operaciones

- `insert` corresponde insertar en el índice $n+1$
- `removeMin` corresponde intercambiar con el índice n , intercambiar y eliminar.

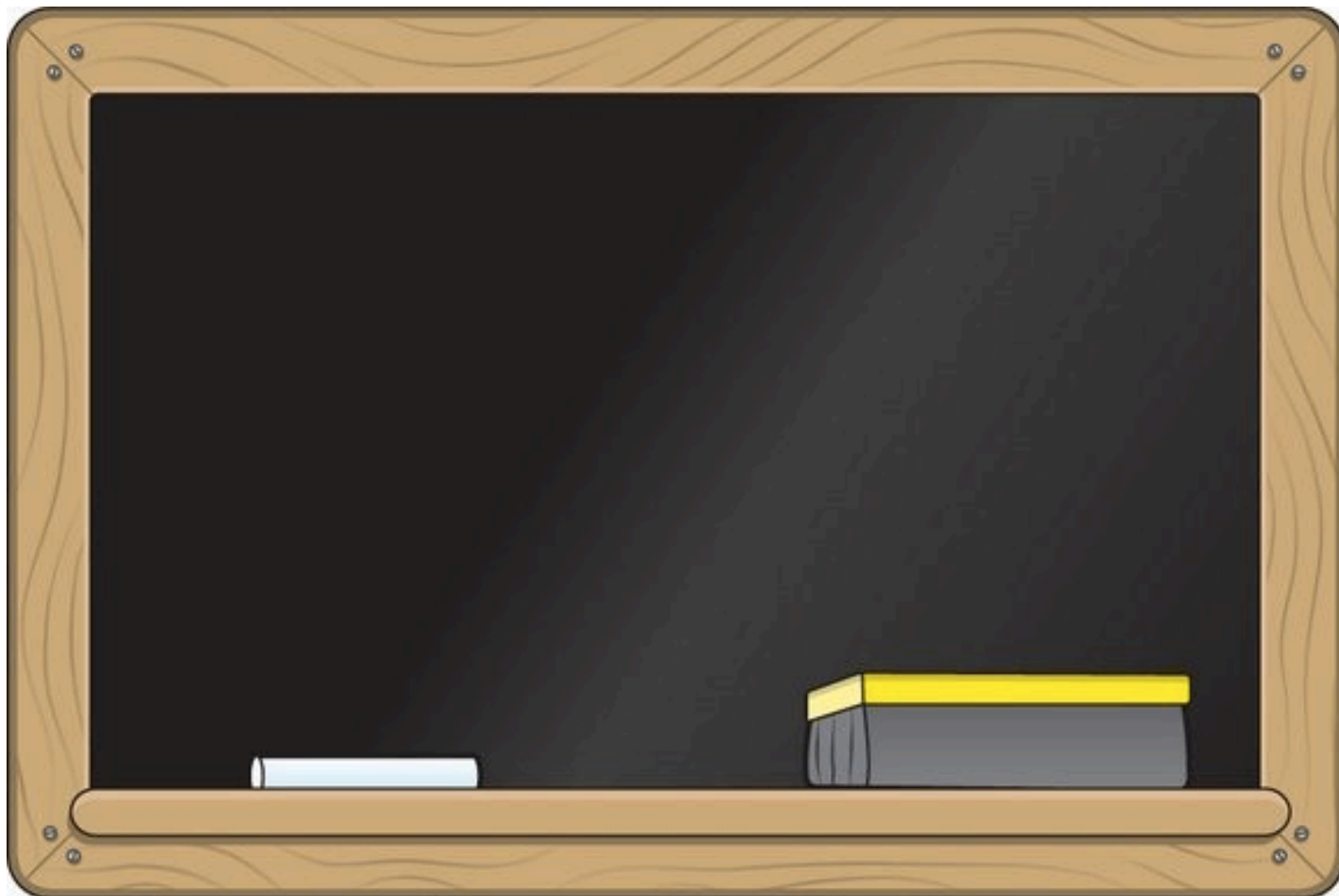


Analisis de la complejidad

Implementación	add	removeMin
Unsorted Array	$O(1)$	$O(n)$
Sorted Array	$O(n)$	$O(1)$
Unsorted Linked List	$O(1)$	$O(n)$
Sorted Linked List	$O(n)$	$O(1)$
Heap	$O(\log n)$	$O(\log n)$
Hash Table		

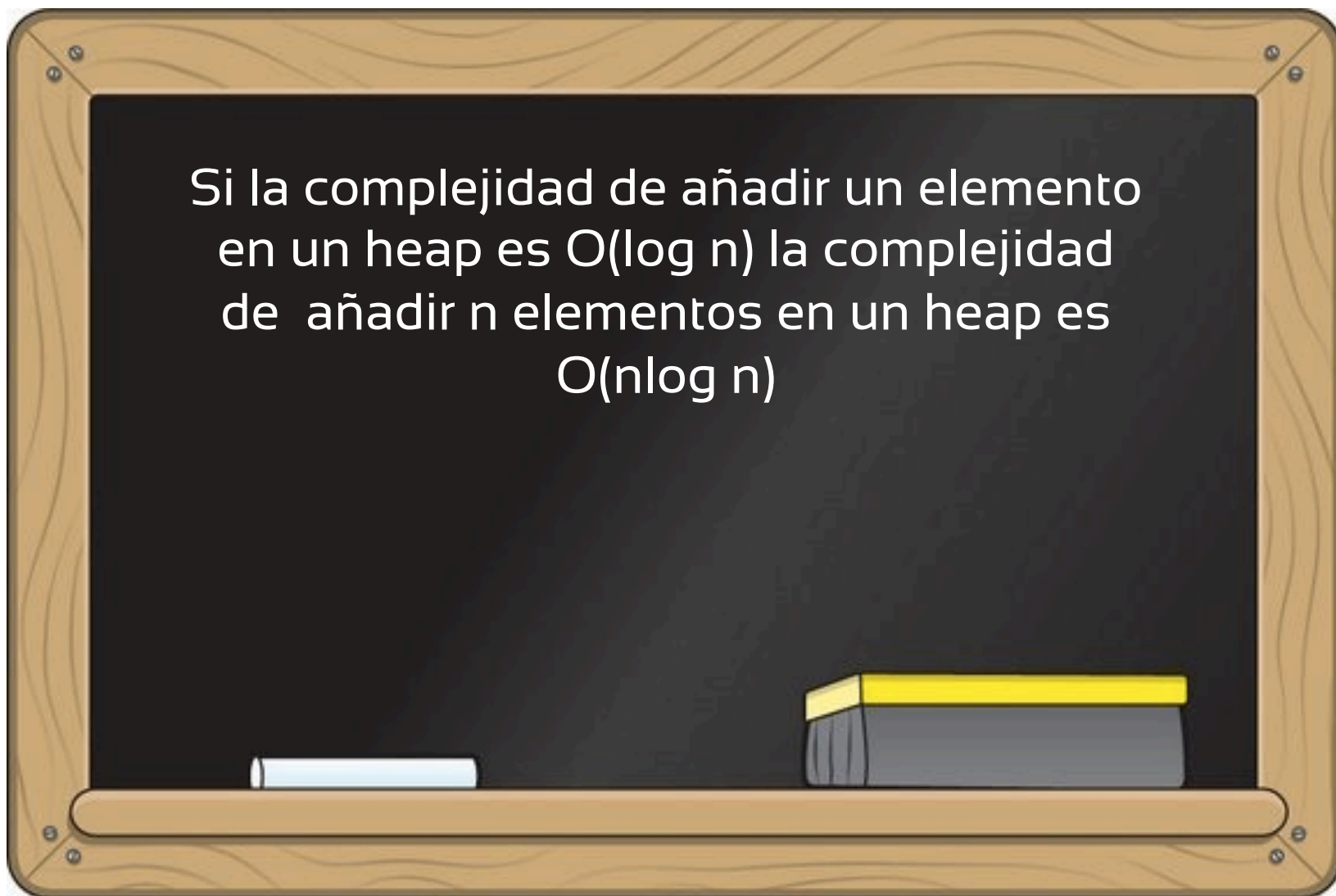
SUBLINEAR == AWESOME

Complejidad de crear un HEAP

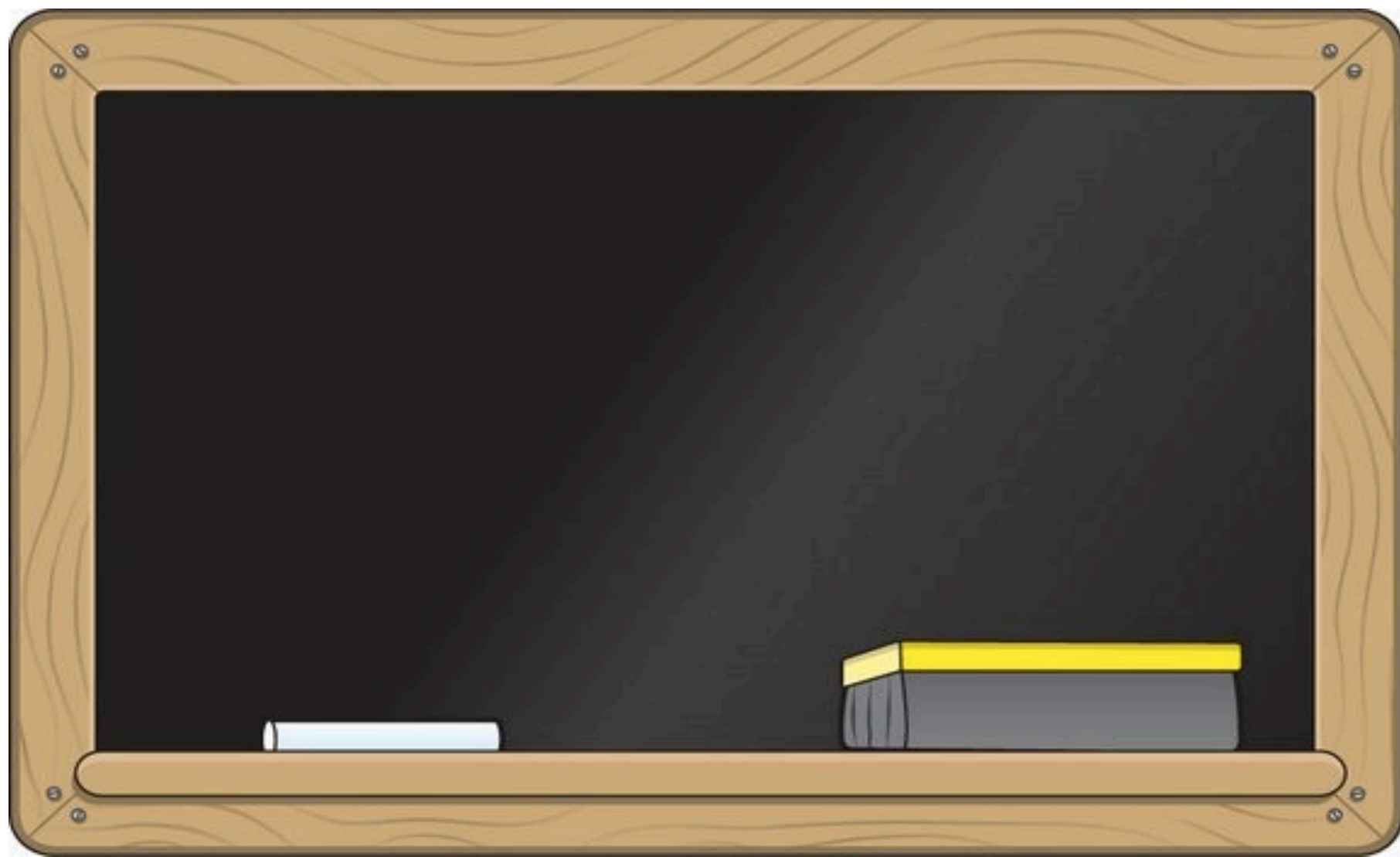


Complejidad de crear un HEAP

Si la complejidad de añadir un elemento en un heap es $O(\log n)$ la complejidad de añadir n elementos en un heap es $O(n \log n)$



Método insert?



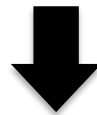
Método insert?

```
def insert(self,item):  
    // Input : item to add  
  
    self._heap.append(item)  
    curPos = len(self._heap)-1  
    found = false  
    while curPos > 0 & found==false:  
        parent = (curPos - 1) / 2  
        parentItem = self._heap[parent]  
        if parentItem <= item :  
            found = true  
        else:  
            self._heap[curPos] = self._heap[parent]  
            self._heap[parent] = item  
            curPos = parent
```

HeapSort

- El **HeapSort** es un algoritmo de ordenación basado en comparaciones de elementos donde se utiliza un **Heap**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
41	67	34	0	69	24	78	58	62	64	5	45	81	27	61

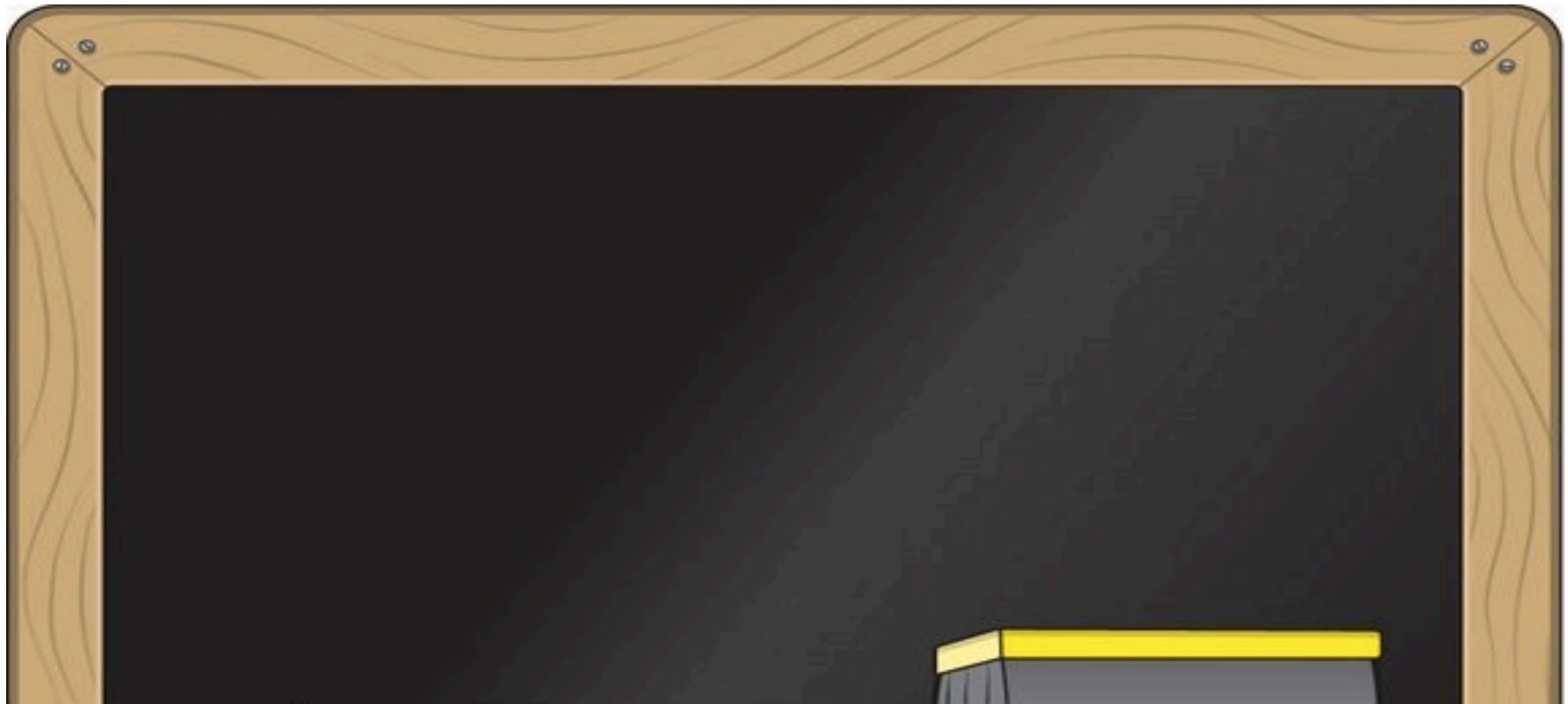


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	5	24	27	34	41	45	58	61	62	64	67	69	78	81

HeapSort

- 1. Paso: Crear un HEAP

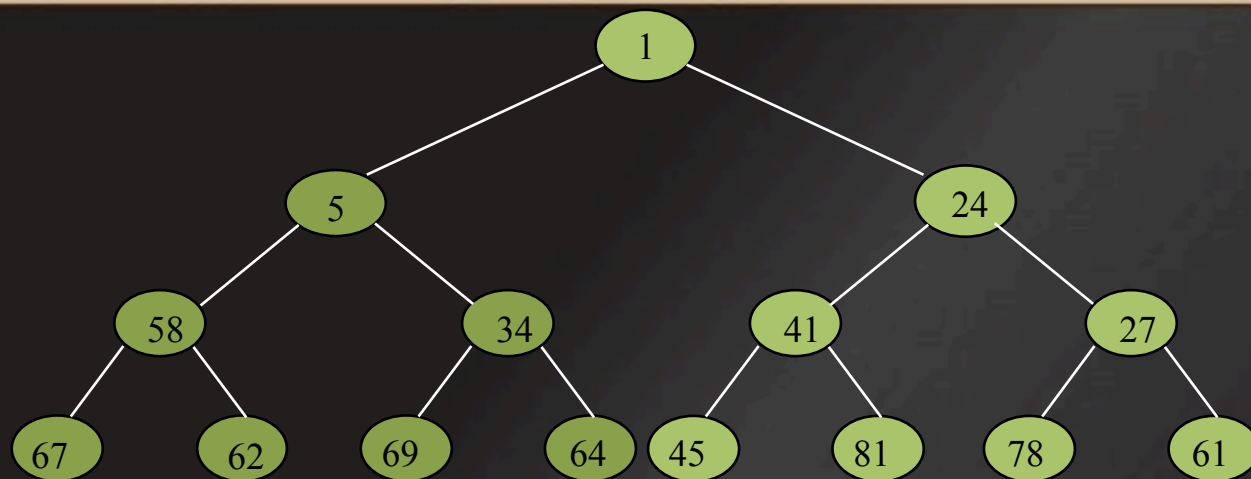
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
41	67	34	0	69	24	78	58	62	64	5	45	81	27	61



HeapSort

- 1. Paso: Crear un HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
41	67	34	1	69	24	78	58	62	64	5	45	81	27	61



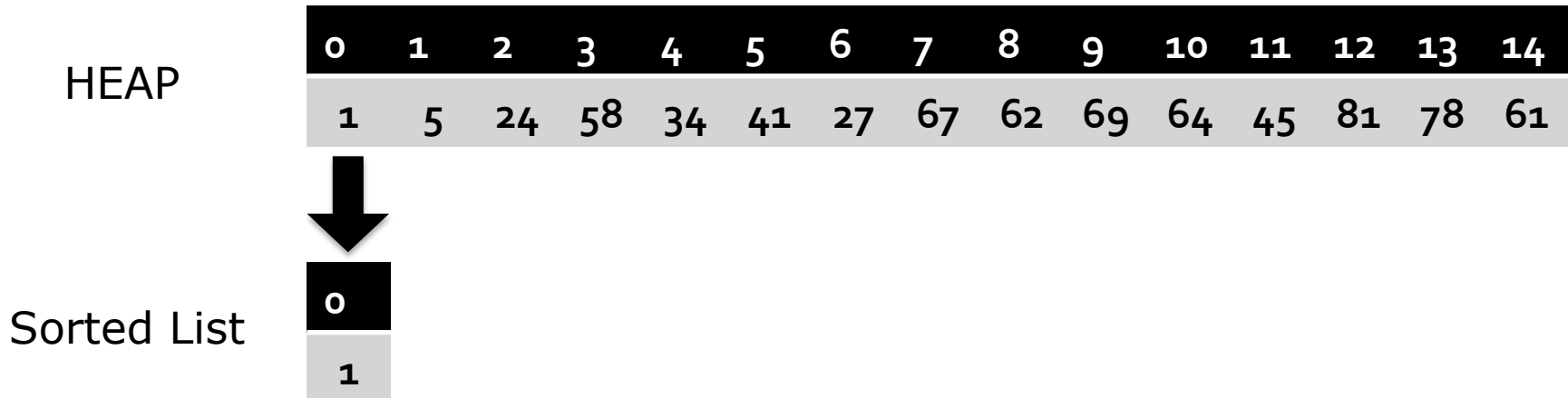
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	24	58	34	41	27	67	62	69	64	45	81	78	61

HeapSort

- 2. Paso. Utilizar Heap Para ordenar

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	24	58	34	41	27	67	62	69	64	45	81	78	61

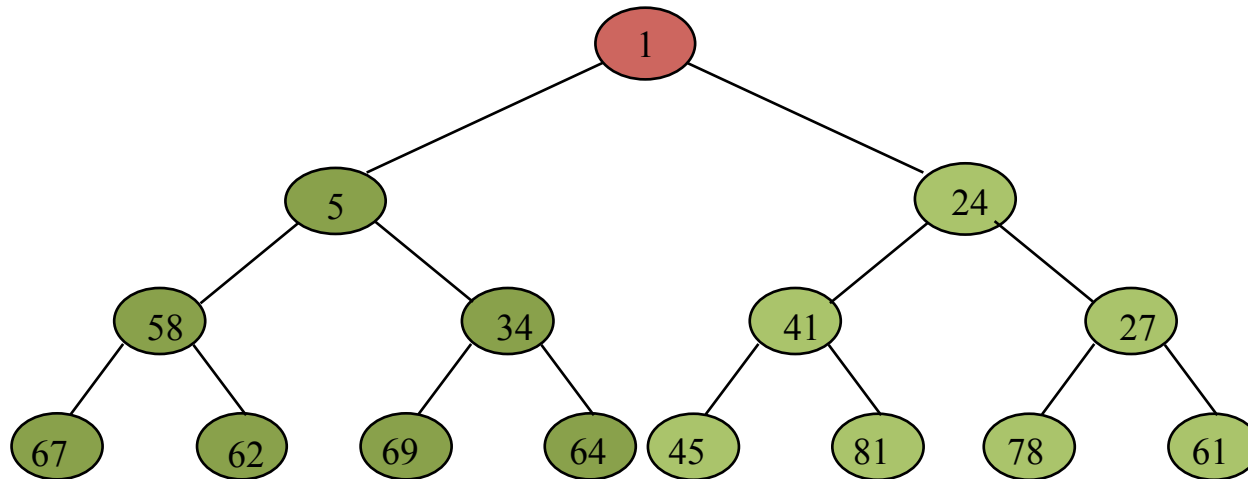
- Vamos quitando el elemento menor y poniendolo en una nueva lista: `>removeMin()`



Ejemplo: removeMin()

HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	24	58	34	41	27	67	62	69	64	45	81	78	61



Sorted List

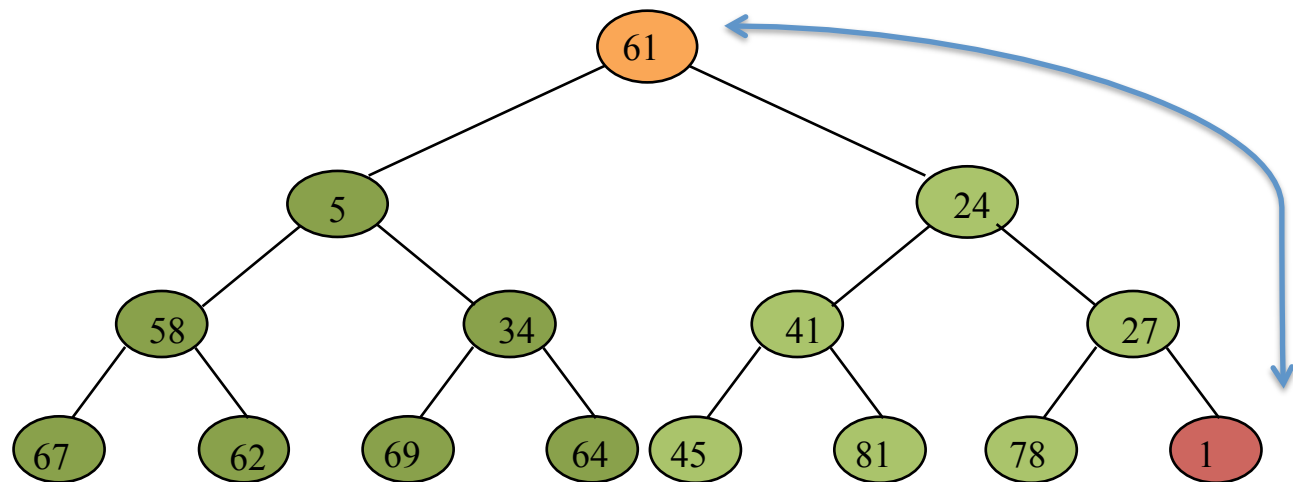
iter 1-s1

0
1

Ejemplo: removeMin()

HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	24	58	34	41	27	67	62	69	64	45	81	78	61



Sorted List

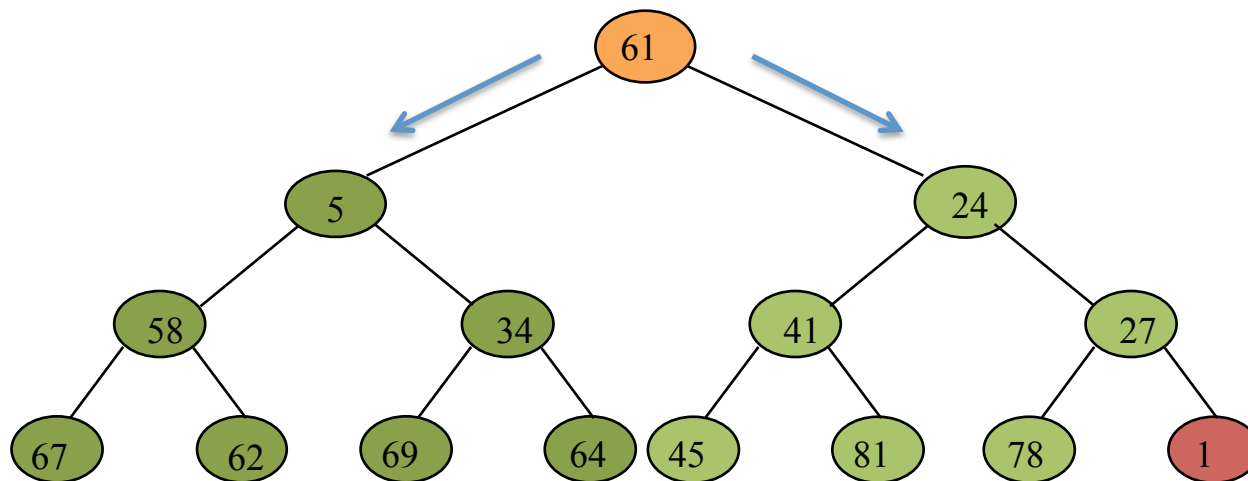
iter 1-s2

0
1

Ejemplo: removeMin()

HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	24	58	34	41	27	67	62	69	64	45	81	78	61



Sorted List

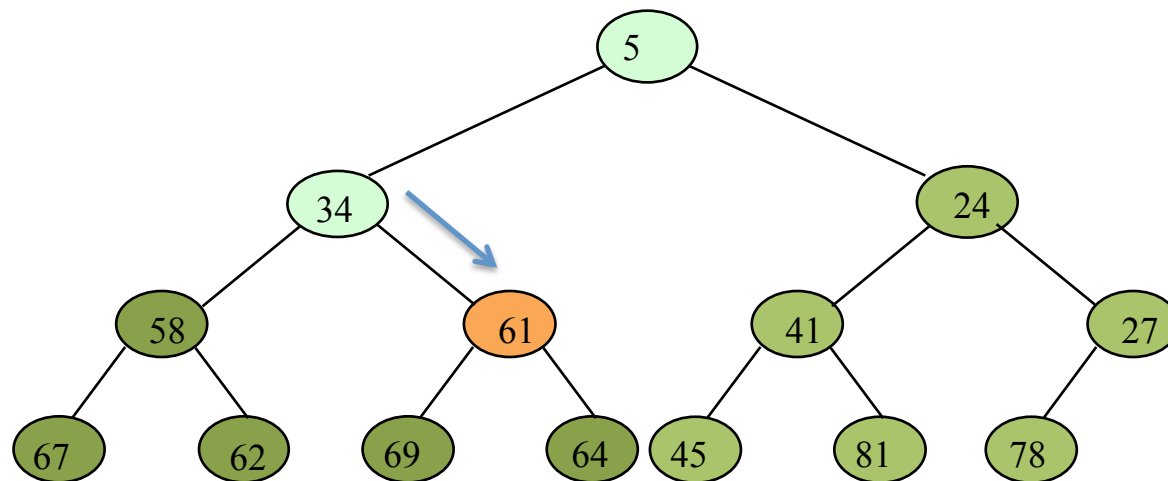
iter 1-s3

0
1

Ejemplo: removeMin()

HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	24	58	34	41	27	67	62	69	64	45	81	78	61



Sorted List

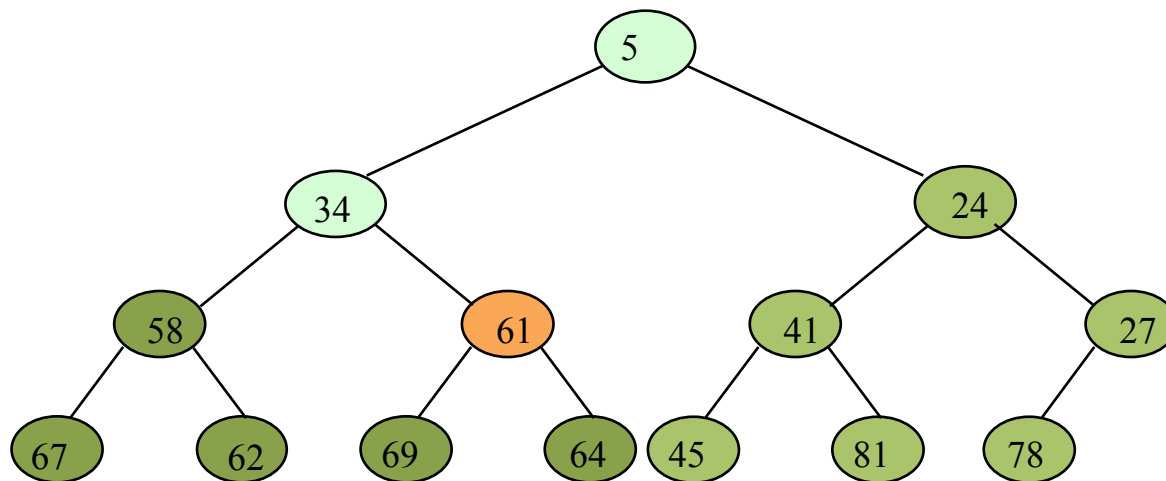
iter 1-s4

0
1

Ejemplo: removeMin()

HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	24	58	34	41	27	67	62	69	64	45	81	78	61



Sorted List

0
1

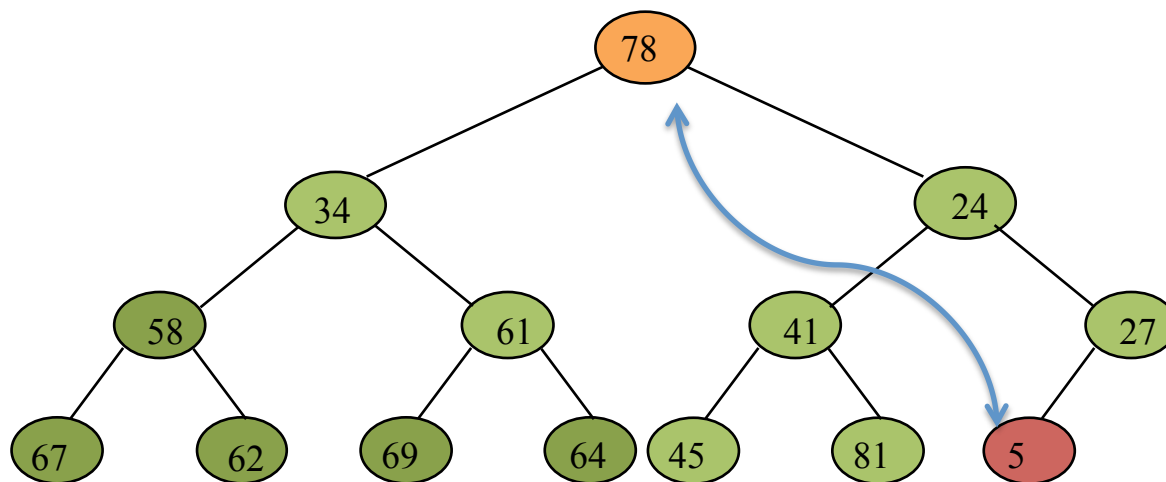
New Heap

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	34	24	58	61	41	27	67	62	69	64	45	81	78

Ejemplo: removeMin()

HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	34	24	58	61	41	27	67	62	69	64	45	81	78



Sorted List

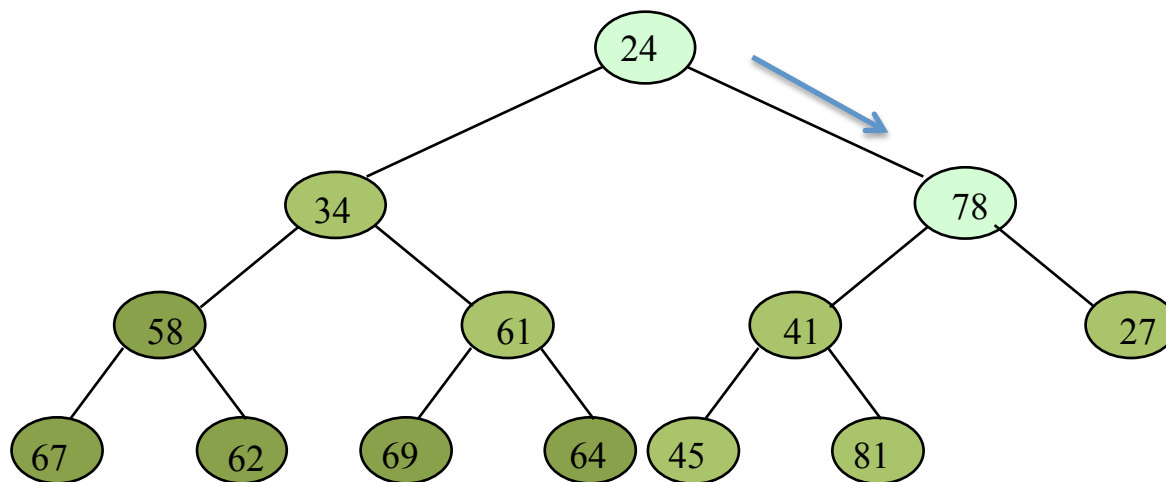
Iter 2-s1

0	1
1	5

Ejemplo: removeMin()

HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	34	24	58	61	41	27	67	62	69	64	45	81	78



Sorted List

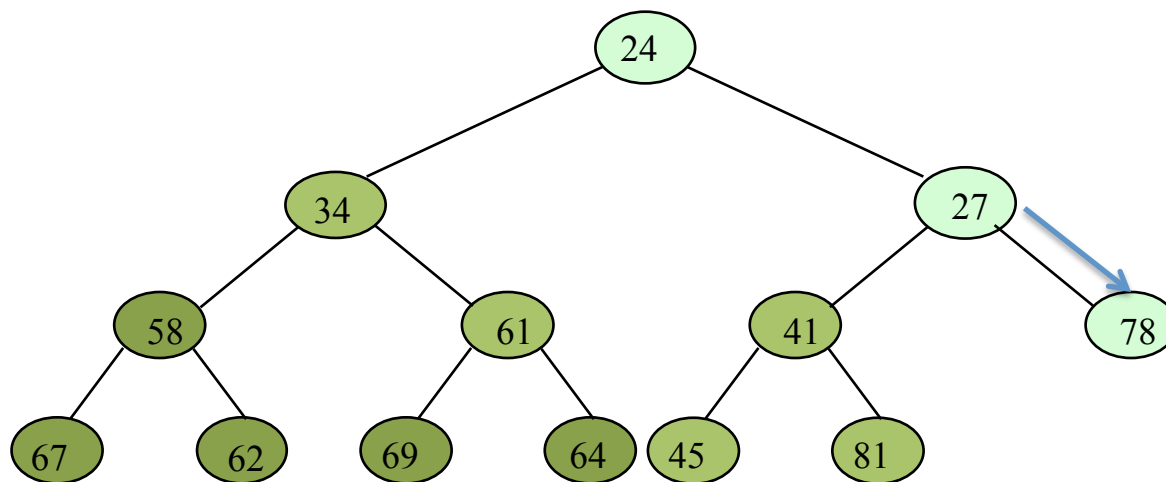
Iter 2-s2

0	1
1	5

Ejemplo: removeMin()

HEAP

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	34	24	58	61	41	27	67	62	69	64	45	81	78

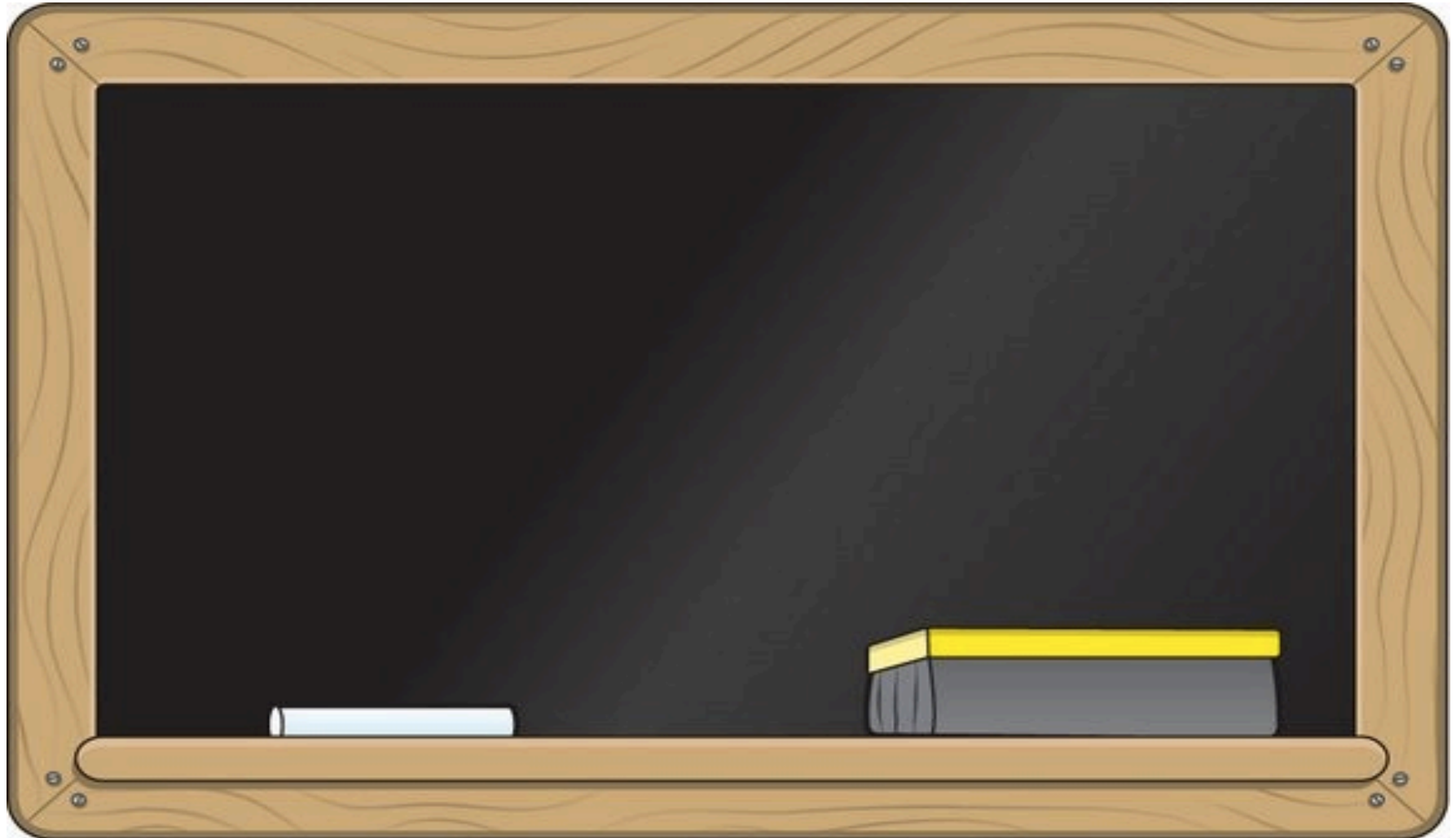


Sorted List

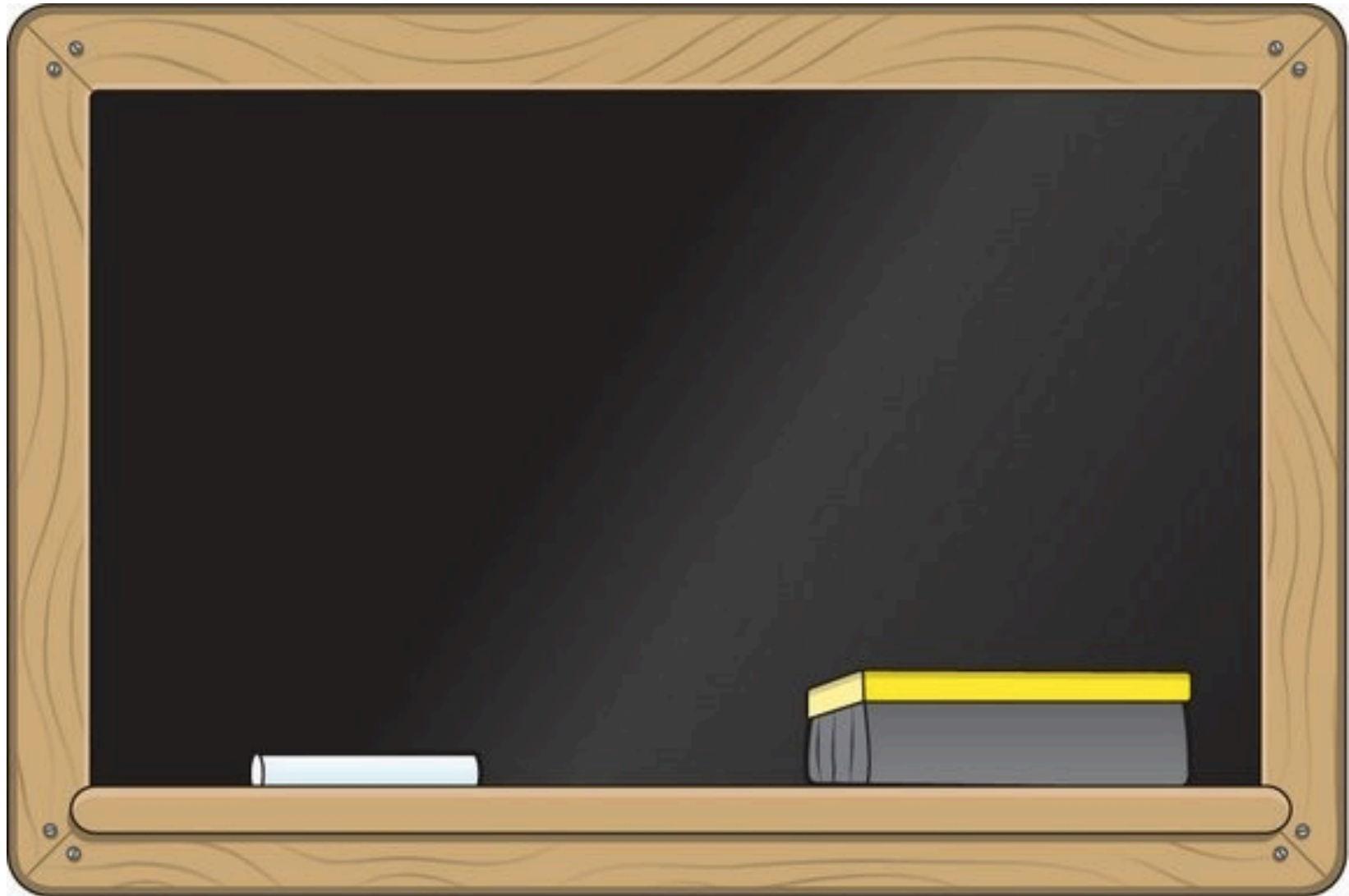
Iter 2-s2

0	1
1	5

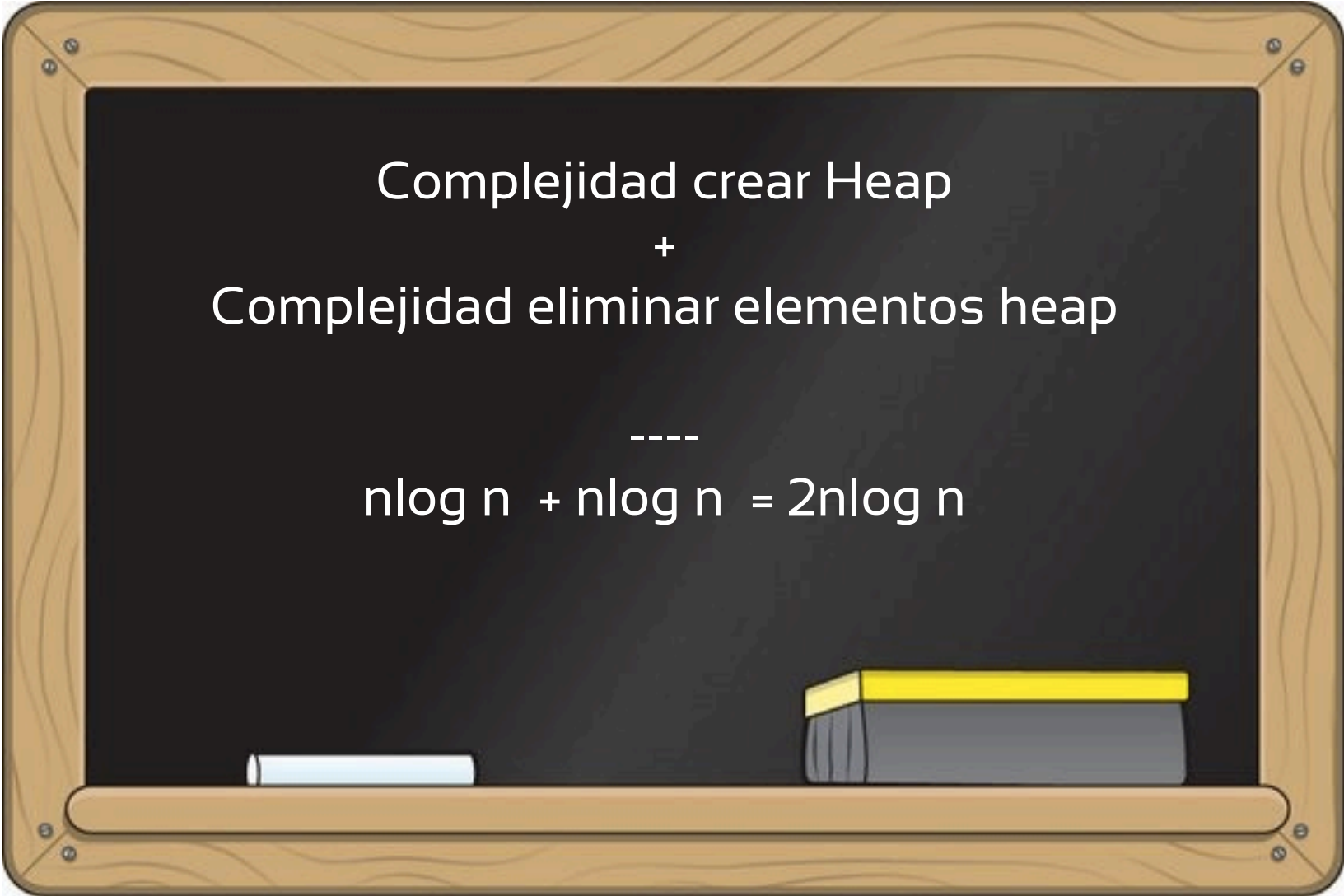
Ejemplo: `removeMin()`



Complejidad del Heap SORT



Complejidad del Heap SORT


$$\begin{array}{c} \text{Complejidad crear Heap} \\ + \\ \text{Complejidad eliminar elementos heap} \\ \text{----} \\ n \log n + n \log n = 2n \log n \end{array}$$