



Tema 5 Heaps

Maria Salamó Llorente
Estructura de Dades

Grau d'Enginyeria Informàtica

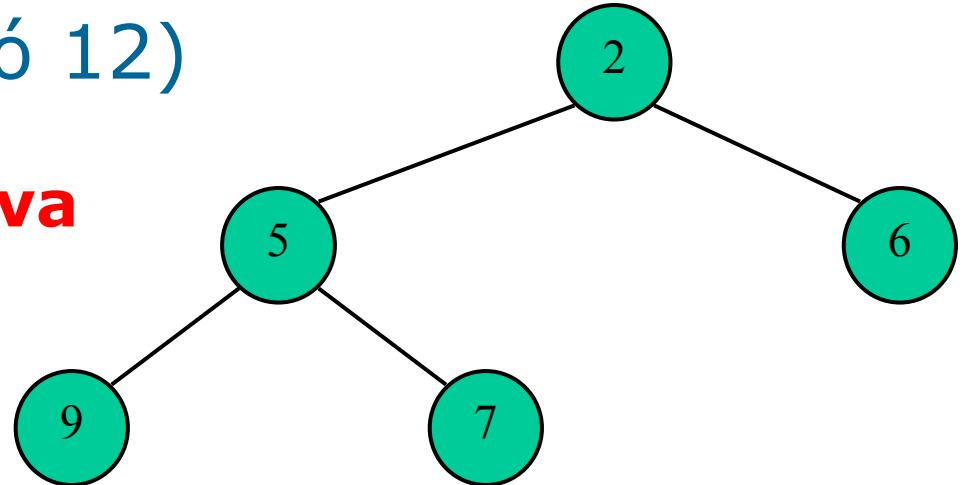
Facultat de Matemàtiques i Informàtica,

Universitat de Barcelona

Contingut

- 5.0 TAD Cua de prioritària (sessió 11)
- 5.1 Introducció als heaps (sessió 11)
- 5.2 Insert i unheap (sessió 11)
- 5.3 removeMin i downheap (sessió 11)
- 5.4 Com implementar els heaps (sessió 11)
- 5.5 Heapsort (sessió 12)

**La cua prioritària estava
al Tema 3 i la veurem
conjuntament amb el
Tema 5 de HEAPS**



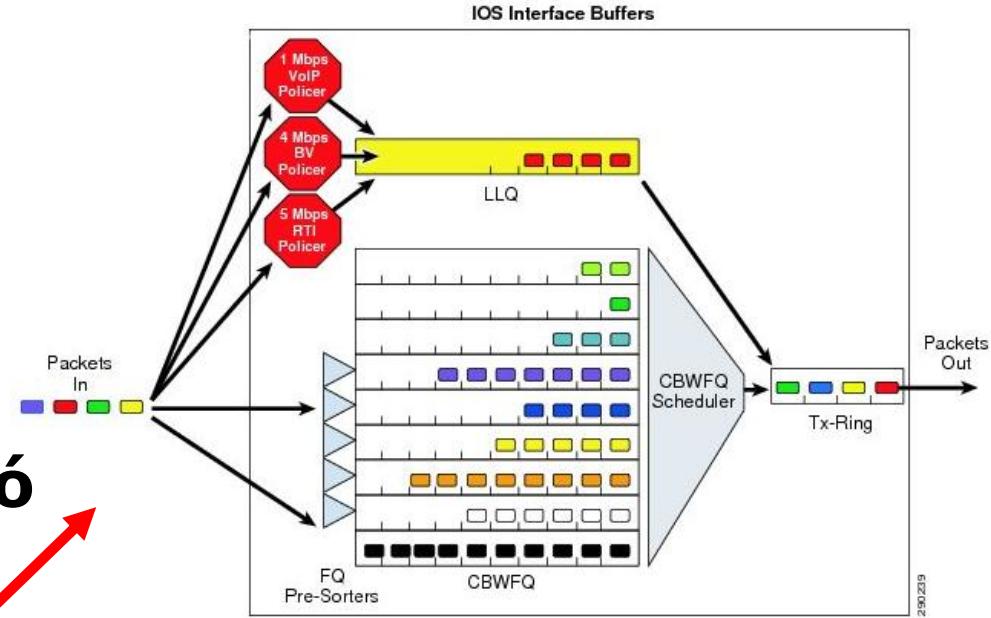
5.0 TAD cua prioritària

**La cua prioritària estava
al Tema 3 i la veurem
conjuntament amb el
Tema 5 de HEAPS**



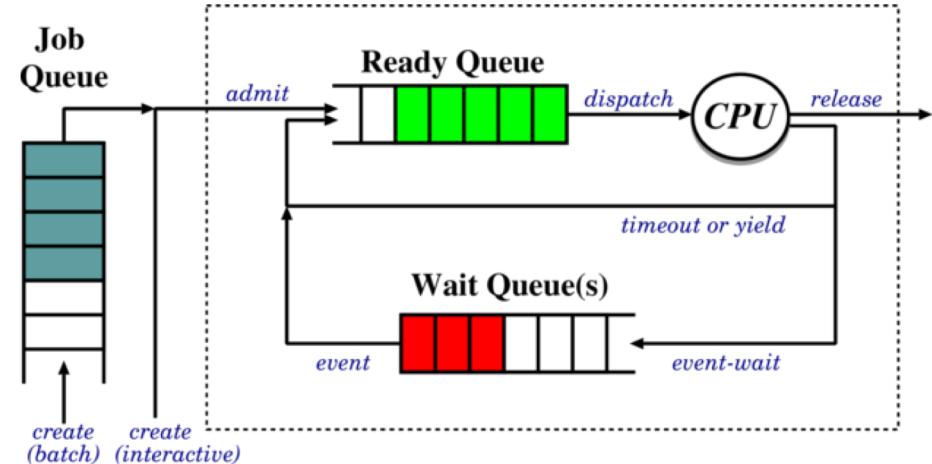
TAD cua prioritària

Les **cues de prioritat** s'utilitzen per planificar tasques en aplicacions informàtiques on la prioritat es correspon amb la **clau d'ordenació**



Exemples:

- Gestió de paquets
- Scheduler





TAD cua prioritària

- Una **cua prioritària** guarda una col·lecció d'entrades
 - Cada entrada (entry) és un parell (**clau, valor**), on la clau indica la prioritat
 - **Dos entrades diferents en una cua prioritària poden tenir la mateixa clau**
- **Mètodes**
 - insert(e) insereix una entry e
 - removeMin() elimina la entry amb la clau mínima
 - En una cua prioritària tradicionalment es considera la clau mínima com a més prioritària
- **Mètodes addicionals:**
 - min() retorna però no elimina la entry amb clau mínima
 - size(), empty()



Ordenació amb cues prioritàries

- Es pot usar una cua prioritària per ordenar un conjunt d'elements
 1. Insereix els elements un a un amb una sèrie d'operacions de `insert`
 2. Extreu els elements en ordre amb una sèrie d'operacions de `removeMin`
- El **temps de l'ordenació** depèn de la implementació de la cua prioritària

Algorithm $PQ\text{-}Sort}(S, C)$

Input seqüència S , comparador C pels elements de S

Output seqüència S ordenada en ordre creixent segons C

$P \leftarrow$ cua prioritària amb el comparador C

while $\neg S.\text{empty}()$

$e \leftarrow S.\text{front}(); S.\text{eraseFront}()$

$P.insert(e, \emptyset)$

while $\neg P.\text{empty}()$

$e \leftarrow P.\text{removeMin}()$

$S.insertBack(e)$

Cues prioritàries basades en encadenaments (ordenació)

- Llista desordenada



- Eficiència:

- insert tarda $O(1)$ en temps ja que quan s'insereix un ítem, s'insereix al començament de la seqüència
 - removeMin i min tarden $O(n)$ en temps ja que s'ha de recórrer tota la seqüència per trobar la clau mínima

- Llista ordenada



- Eficiència:

- insert tarda $O(n)$ en temps ja que s'ha de trobar el lloc on correspon inserir l'ítem
 - removeMin i min tarden $O(1)$ en temps, ja que la clau mínima està sempre al començament



TAD cua prioritària

Implementacions del TAD cua prioritària:

- **Llistes enllaçades**
 - El cost computacional insert o removeMin serà $O(n)$
- **Vectors**
 - El cost computacional insert o removeMin serà $O(n)$
- **Heap** és la forma més eficient (en castellà montículo).
 - Aquesta implementació té un cost logarítmic $O(\log n)$



5.1 Introducció als heaps



Introducció als heaps

- **Motivació:**

- Una diversitat de problemes són resolts usant una gran col·lecció de dades on cada ítem té una prioritat
- Exemples:
 - *Sortides de vols*
 - Els diferents vols necessiten una pista d'enlairament
 - Hi ha vols que tenen més prioritat que d'altres
 - *Gestió de l'ample de banda*
 - Les dades d'alta prioritat (real-time data com skype) s'han de transmetre primer

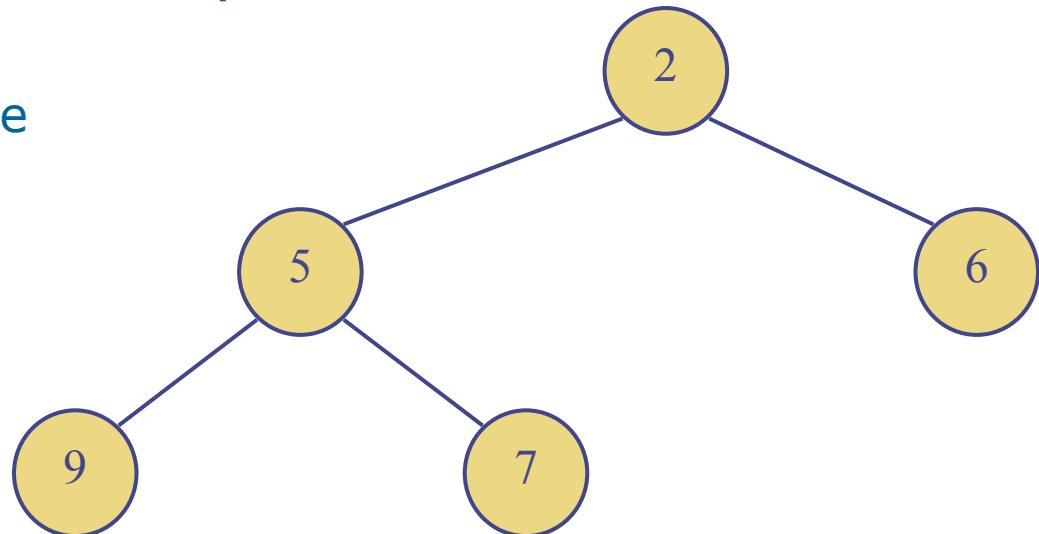


Introducció als heaps

- Una cua prioritària és una col·lecció de zero o més elements
 - Cada element té una **prioritat** o valor
- L'ordre per buscar/eliminar elements ve determinat per la **prioritat**
- Els elements són eliminats en ordre creixent o decreixent de prioritat, enllloc de l'ordre en que han arribat a la cua
- Existeixen dos tipus de cues de prioritat:
 - Min Priority queue elimina l'element amb mínima prioritat
 - Max Priority queue elimina l'element amb màxima prioritat

Introducció als heaps

- Estructura de dades utilitzada per implementar una **cua de prioritat**
 - `insert(key, element)`: inserta un ítem amb la seva clau i l'element dins de la cua
 - `removeMin()`: elimina l'ítem amb la clau menor i retorna l'element associat
- Es pot implementar usant arbres (amb estructura encadenada o amb un array)
 - Analitzem primer un arbre



Propietats dels Heaps

Un **heap** es defineix com un arbre binari quasi complet de n nodes , tal que el contingut de cada node es major o igual (o menor o igual) que el contingut dels seus fills

- **Arbre binari**

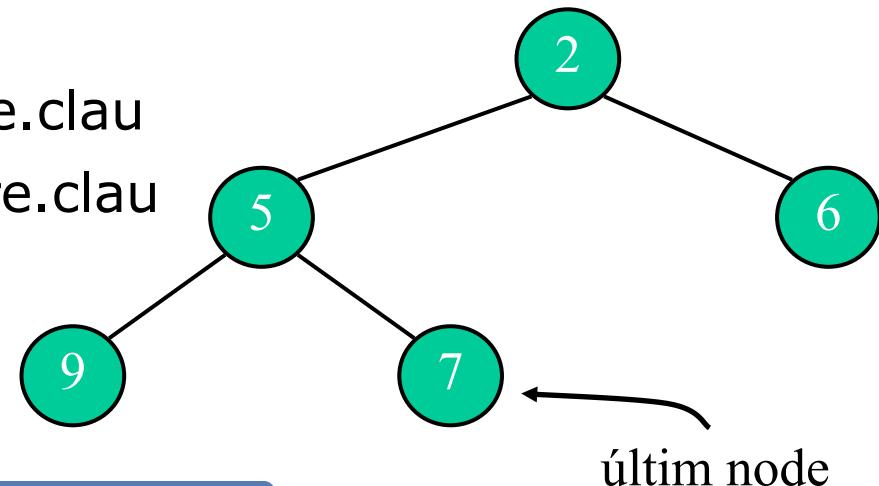
- Cada node té com a màxim dos fills
- És un arbre quasi complet

- **Clau** (o “prioritat”) en cada node

- **Ordre del Heap**

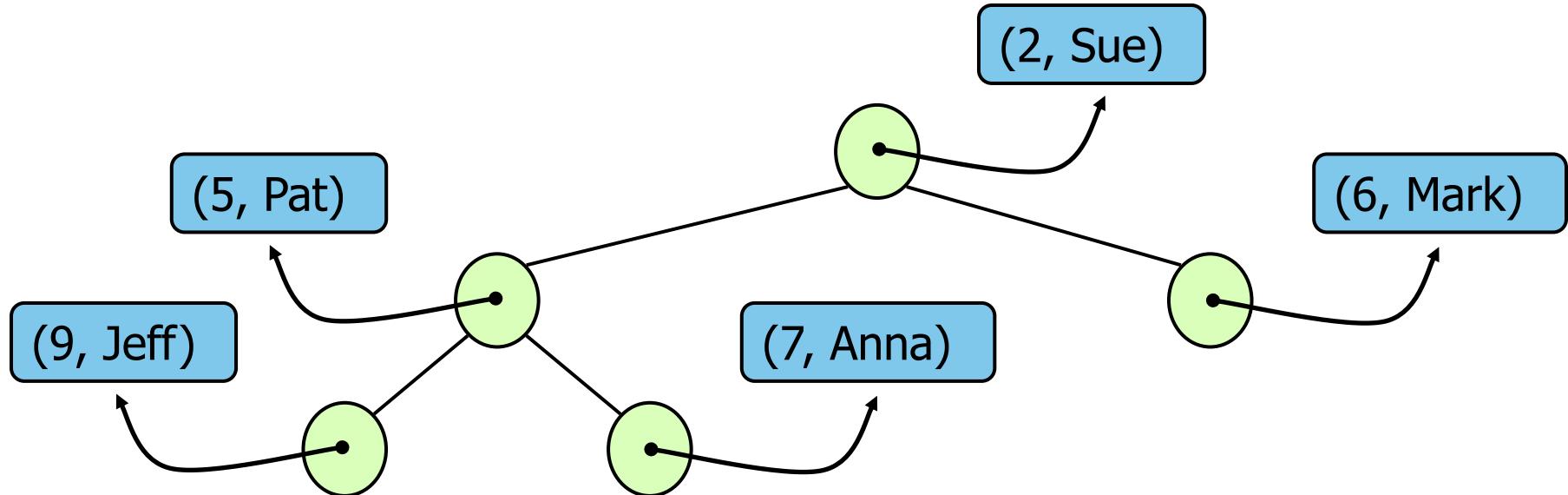
- Per **min-heap**: $n.\text{clau} \geq n.\text{pare.clau}$
- Per **max-heap**: $n.\text{clau} \leq n.\text{pare.clau}$

- **Alçada** $O(\log n)$



Heaps i Cues de Prioritat

- Podem usar un heap per implementar una cua de prioritat
- Guardem un ítem (**clau, element**) en cada node





Operacions al heap

Les operacions més importants del TAD heap són:

- Constructor
- `insert(entry)`: inserta una nova entry amb la seva clau i l'element dins del heap
- `min()`: retorna la entry o l'element amb clau mínima del heap
- `removeMin()`: elimina la entry amb la clau menor i retorna l'element associat
- `isEmpty()`: retorna si el heap està buit o no

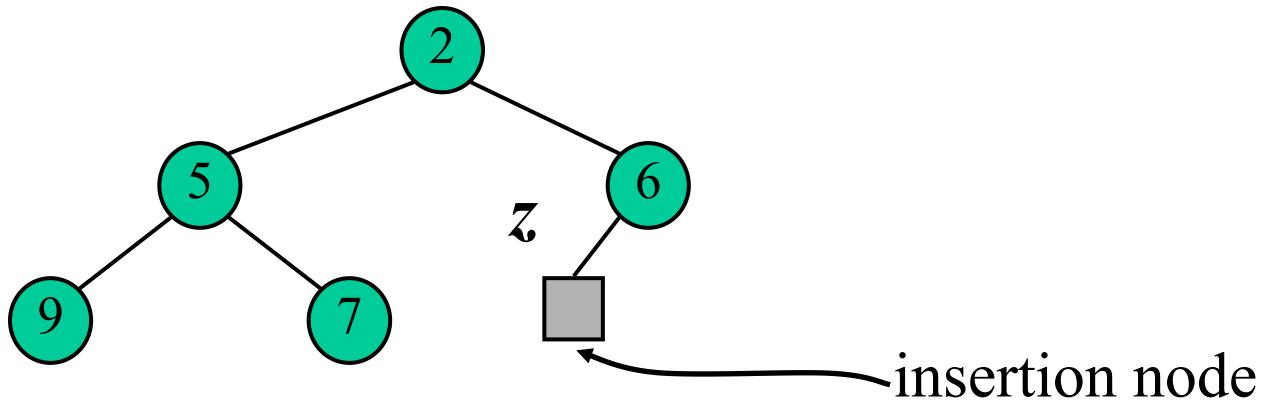
A continuació veurem com es fa per mantenir la condició d'ordre a `insert` i `removeMin`.



5.2 Insert i upheap

Insert en un Heap

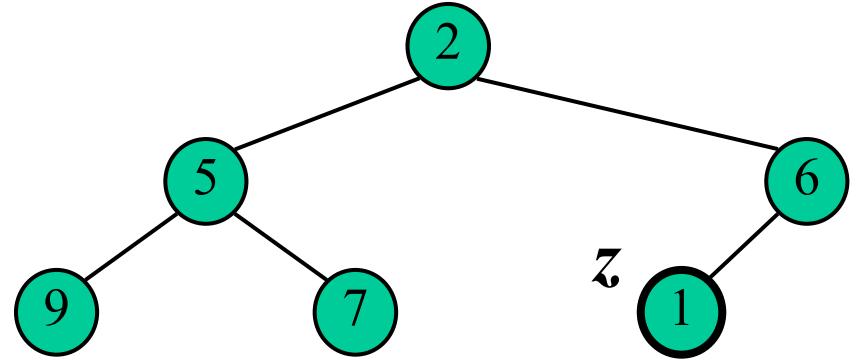
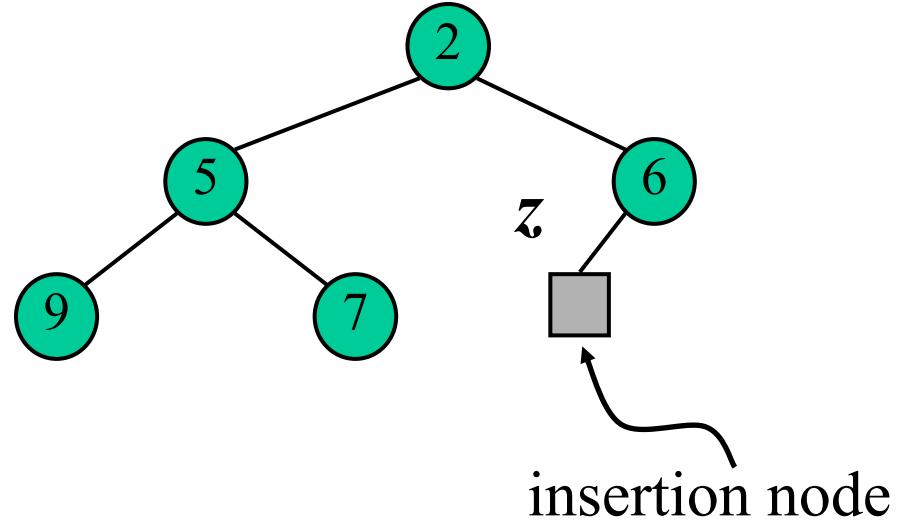
- En un heap, s'ha de guardar la posició (node) on s'ha d'inserir el següent element per mantenir l'arbre semi-complet: “**insertion node**”
- Consisteix en inserir una clau k al heap
- **3 passos:**
 - Buscar el **insertion node** z
 - Guardar k a z
 - Reestructurar la propietat d'ordre del heap (a continuació)



Insert en un Heap

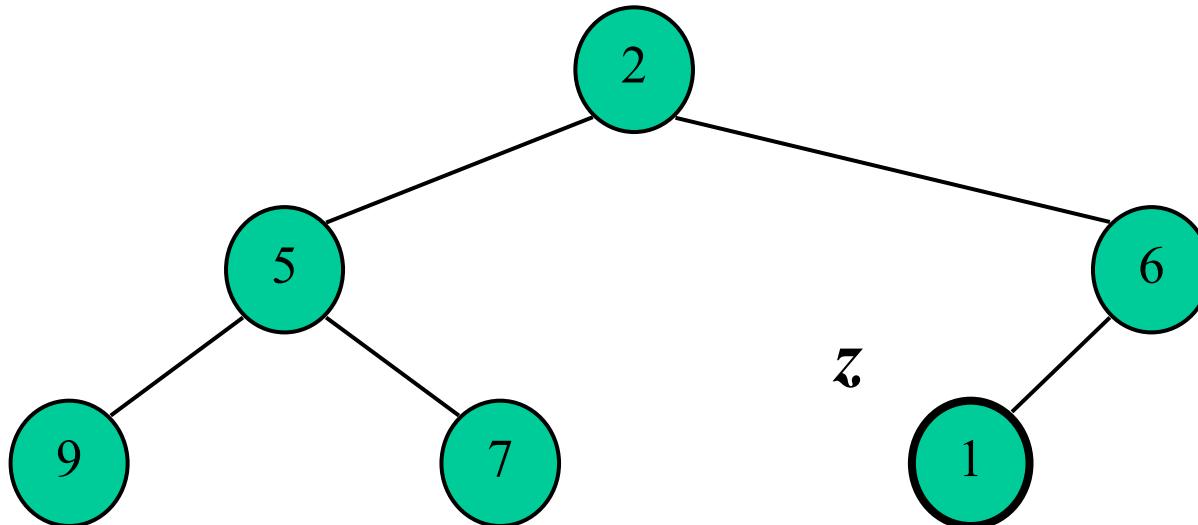
Exemple: insert(1)

- Inserim el nou node allà on tenim apuntant el “**insertion node**”



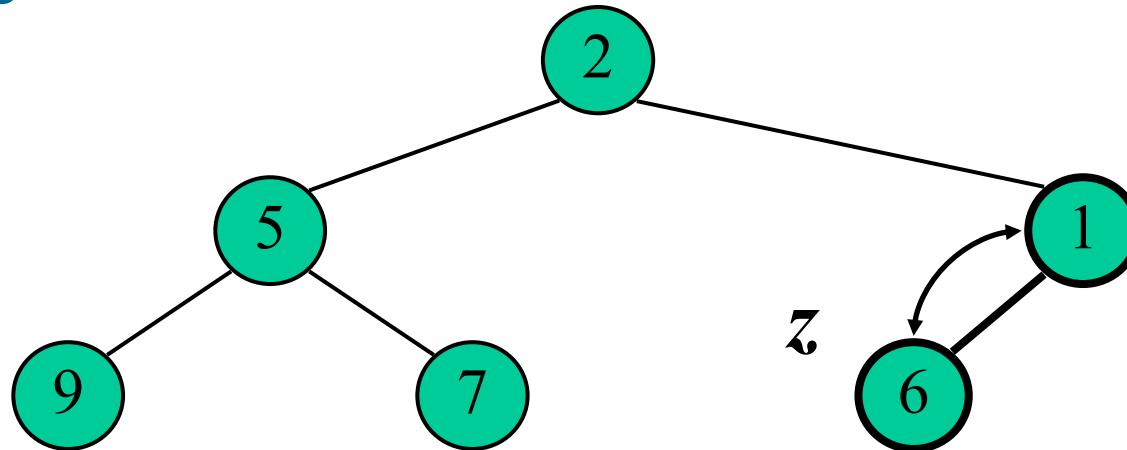
Insert en un Heap

- **Què ocorre ara?**
 - L'ordre del heap no es compleix!!



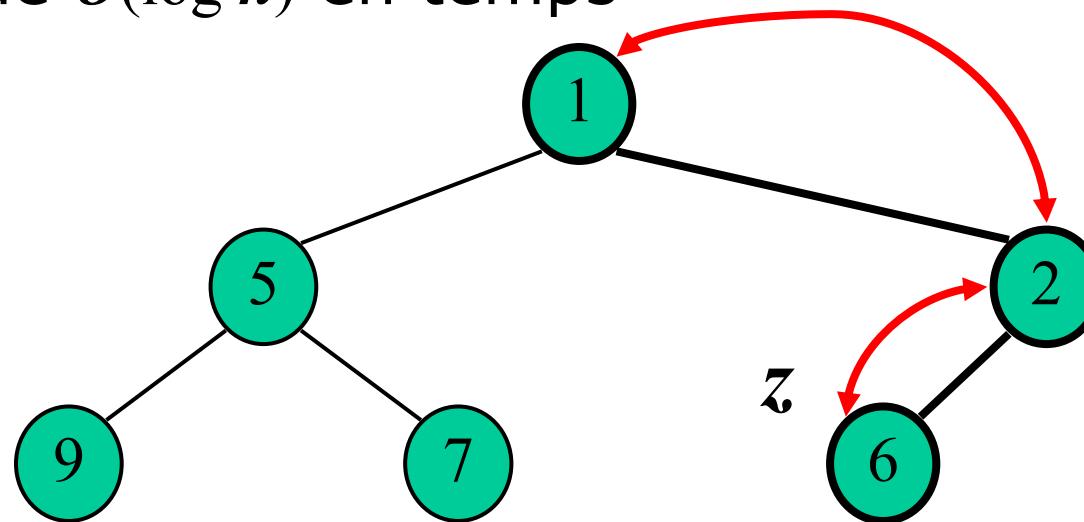
Upheap

- Hem de restablir l'ordre del heap
 - Intercanviant els ítems cap a dalt fins que la propietat d'ordre del heap quedi restablert.
 - Acaba quan troba el root o quan el pare és menor o igual a k
- El primer intercanvi restableix tot per sota de la nova posició



Upheap

- Hem de fer un altre swap, ja que 1 es menor que el 2 en el node superior
- Finalment, les propietats del Heap es satisfan
- Donat que l'arbre té alçada $O(\log n)$, upheap té un cost de $O(\log n)$ en temps

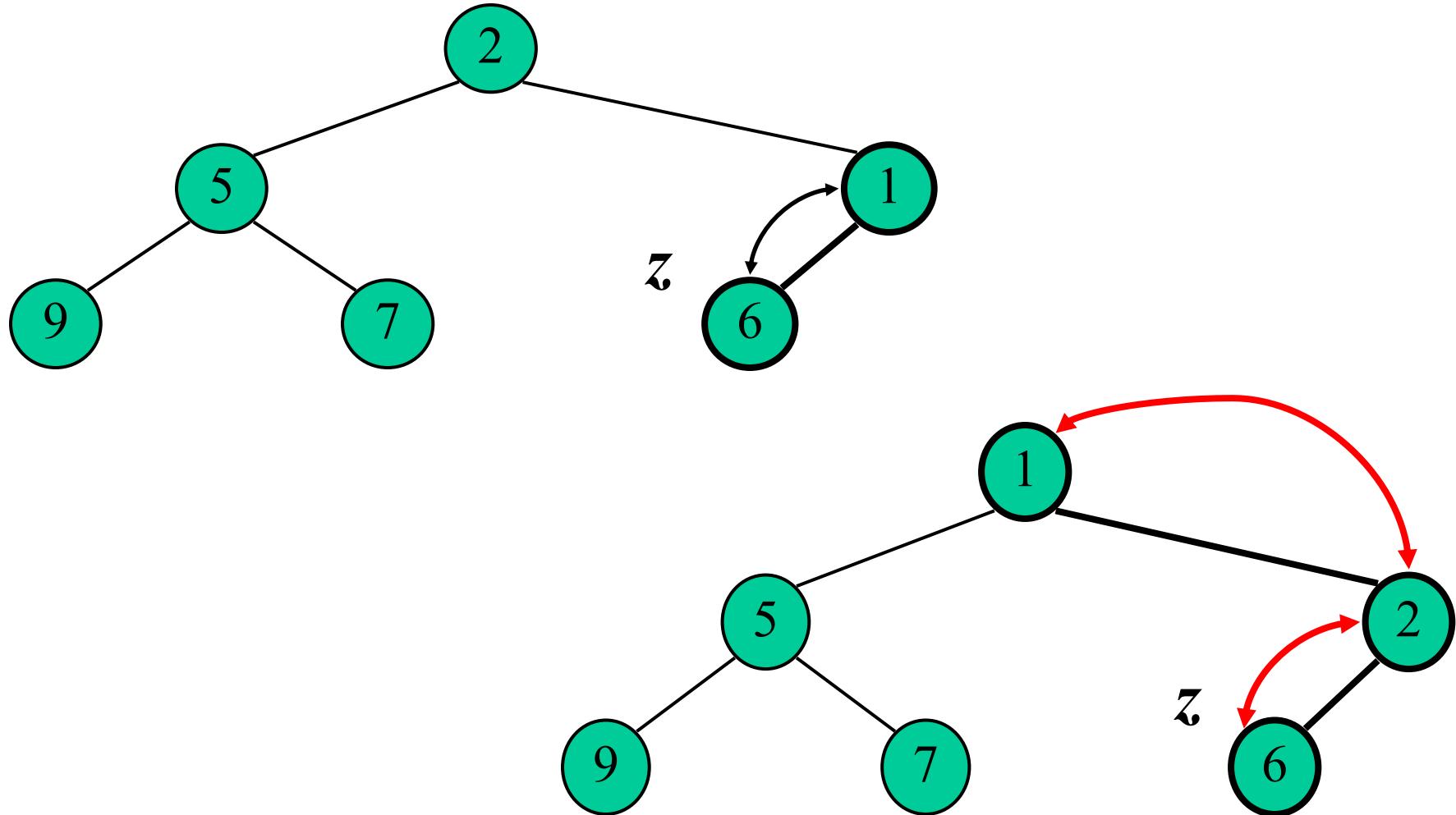




Upheap

- Després d'inserir un nou node amb clau k, la propietat d'ordre del heap pot no complir-se
- L'algorisme upheap restableix l'ordre del Heap intercanviant els nodes en el camí des del node inserit fins a l'arrel de l'arbre
- Upheap acaba quan la clau arriba a l'arrel o en un node on el seu pare té una clau més petita o igual a k
- Donat que el heap té una alçada $O(\log n)$, upheap té una complexitat de $O(\log n)$ en temps

Upheap





Exemple insert

Com queda el **min-heap** al inserir els següents elements:

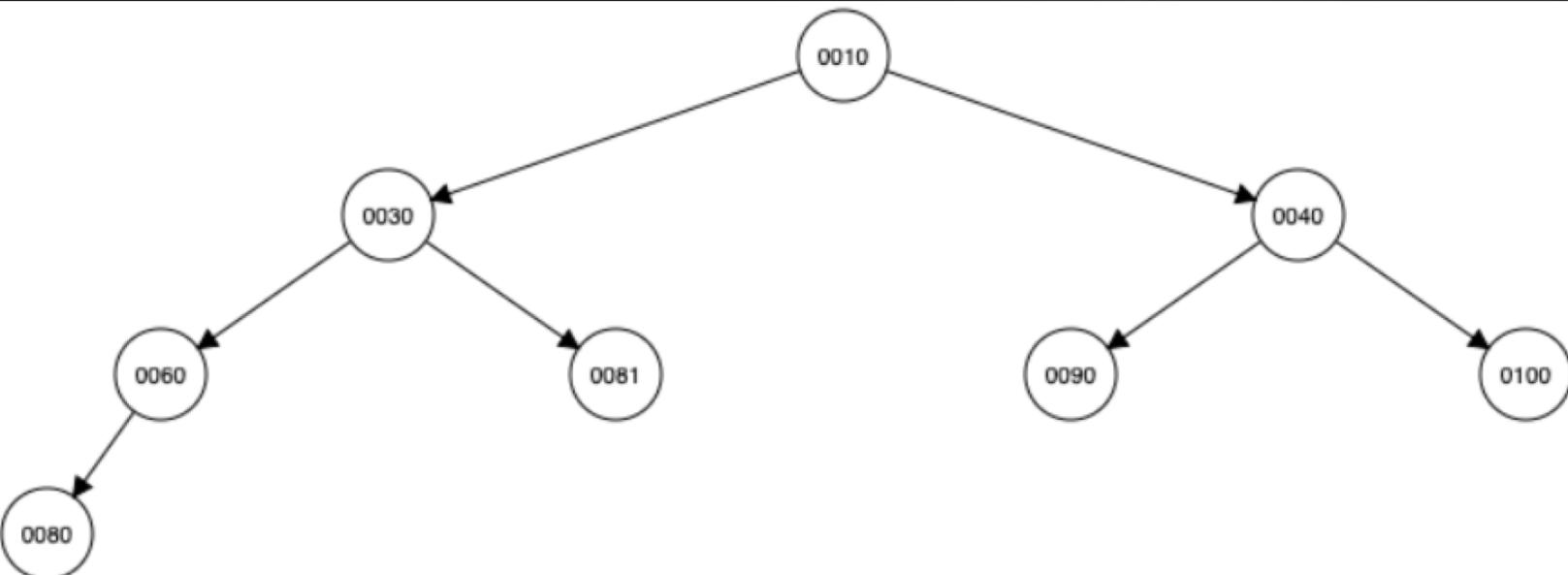
80, 40, 30, 60, 81, 90, 100, 10

Comproveu la vostra solució a:

<https://www.cs.usfca.edu/~galles/visualization/Heap.html>

Exemple insert (Solució)

Com queda el min-heap al inserir els següents elements: 80, 40, 30, 60, 81, 90, 100, 10

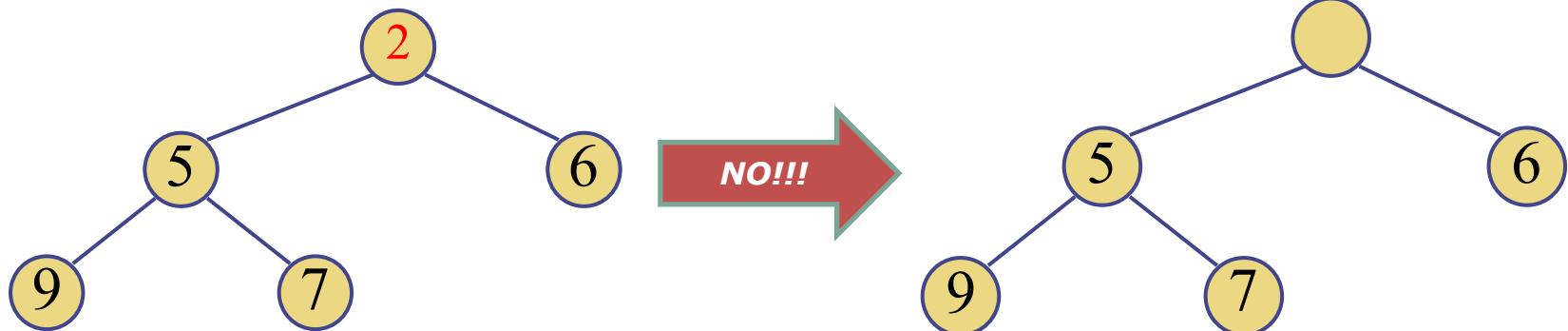




5.3 removeMin i downheap

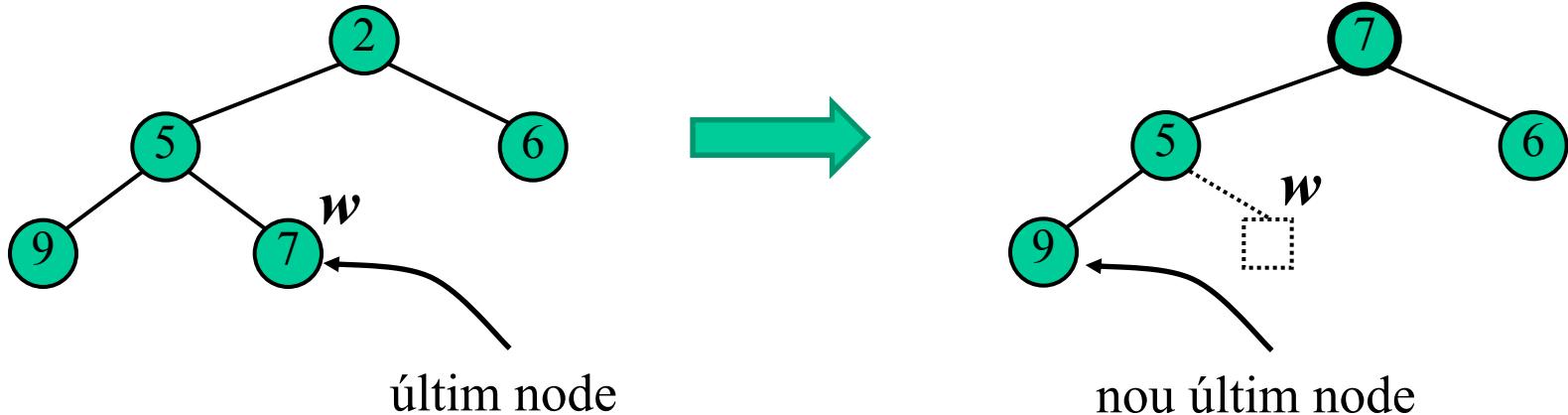
removeMin

- L'element mínim d'un heap sempre és l'arrel (degut a l'ordre del heap)
- Com eliminarem un element del heap sense destruir el seu ordre?



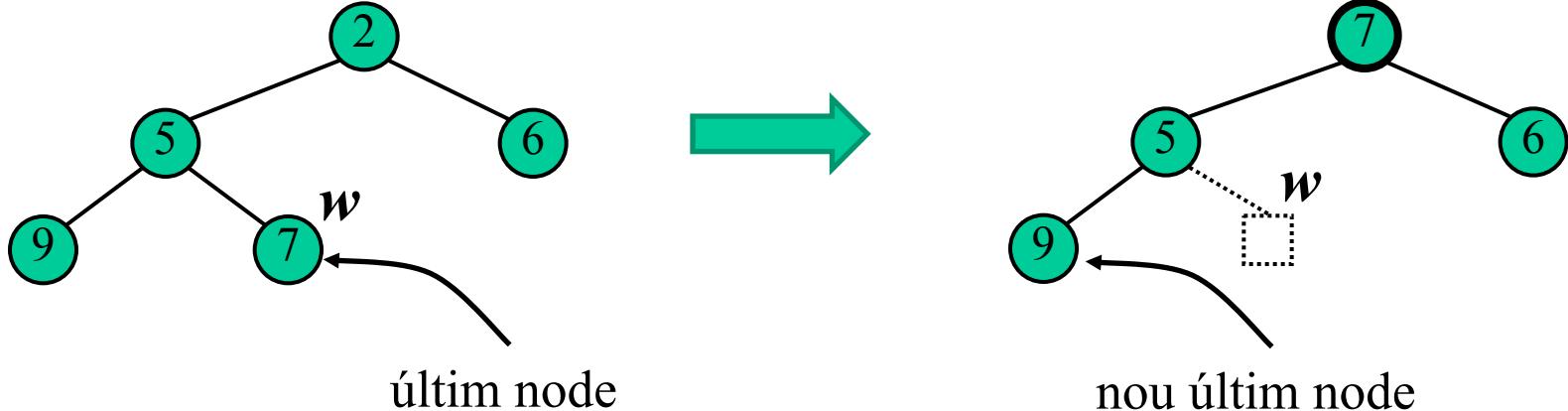
removeMin

- El mètode ***removeMin*** té tres pasos:
 1. Intercanviar l'arrel (l'element que volem eliminar) amb l'últim element del heap
 2. Eliminar l'element w
 3. Restablir l'ordre del heap



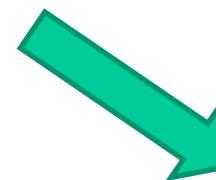
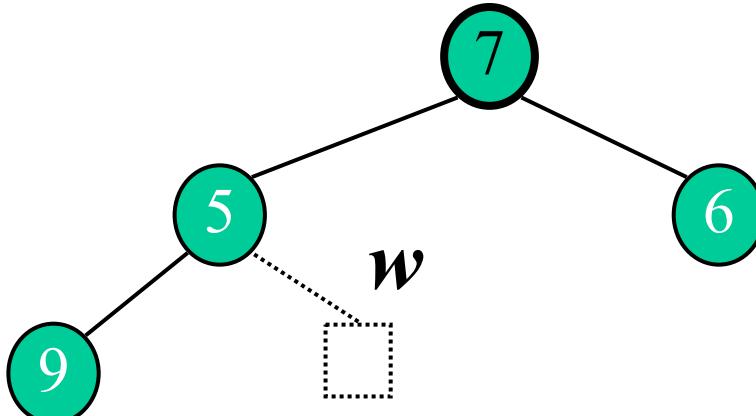
removeMin

- **Pas 1.** Intercanviar el node arrel amb l'últim element
 - D'aquesta manera l'ordre del heap no es conserva
- **Pas 2.** Eliminar el node (w) i re-assignar últim node
- **Pas 3.** Fer un downheap per restablir l'ordre

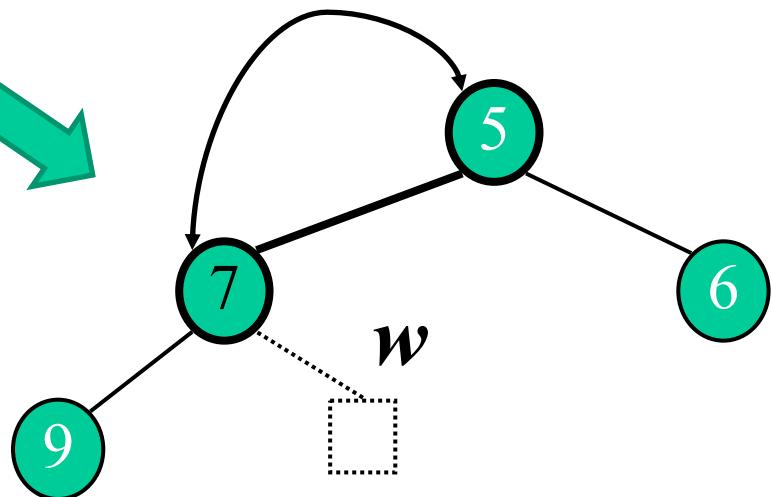


Downheap

- Anem baixant l'element de l'arrel cap a baix tant com sigui necessari



- Ara l'ordre es conserva



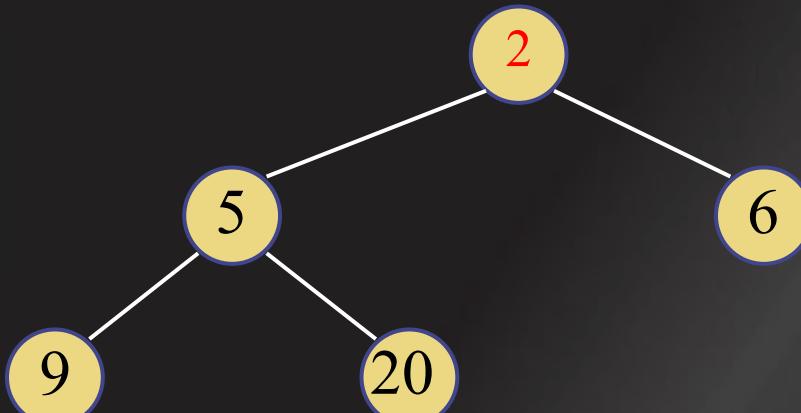


Downheap

- Downheap restableix l'ordre del heap intercanviant els ítems en una trajectòria descendent des de l'arrel, intercanviant el node pel **menor dels seus fills**.
- Downheap acaba quan la clau k arriba a una fulla o en un node on el seu fill tingui una clau més gran o igual a k
- Donat que el heap té una alçada de $O(\log n)$, el downheap té una complexitat de $O(\log n)$ en temps

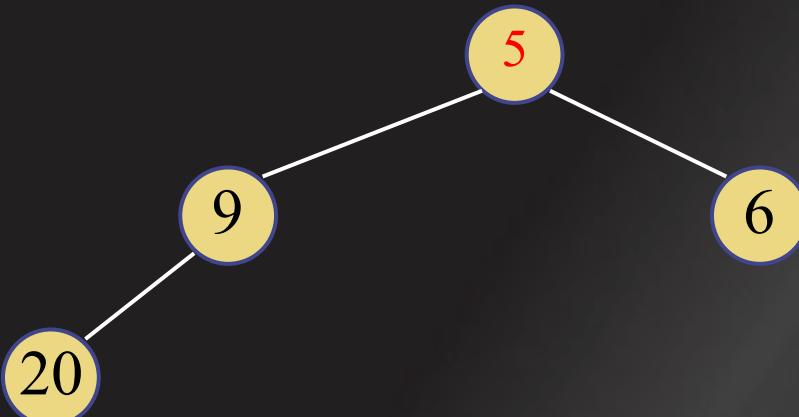
Exemple removeMin()

Com queda l'estructura si fem removeMin()?



Exemple removeMin (Solució)

Com queda l'estructura si fem removeMin()?



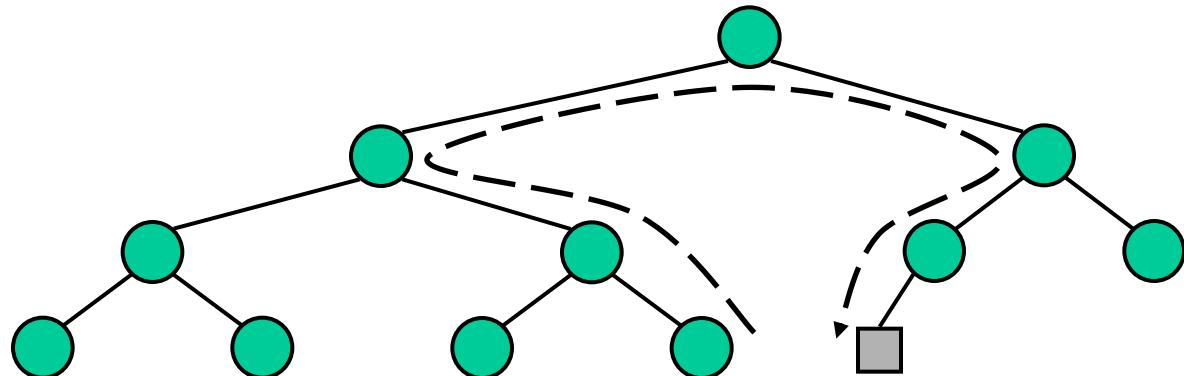


Implementació del heap: Resum

- **insert():**
 - Inserim un ítem en l' “**insertion node**”, el següent espai lliure
 - **Upheap** des de sota fins a on sigui necessari
- **removeMin():**
 - Intercanviar l'arrel amb l'últim node inserit en el heap
 - Eliminem el node arrel intercanviat
 - **Downheap** des de l'arrel fins a on sigui necessari

Com trobar el “Insertion Node”

- El “insertion node” pot ser trobat recorrent un camí de $O(\log n)$ nodes:
 - Comencem amb l’últim node afegit.
 - Anem cap a dalt fins a trobar un fill dret o bé arriben a l’arrel
 - Si ens trobem en un fill esquerra ens situarem al seu germà (el corresponent fill dret)
 - Anem cap a baix fins a trobar una fulla



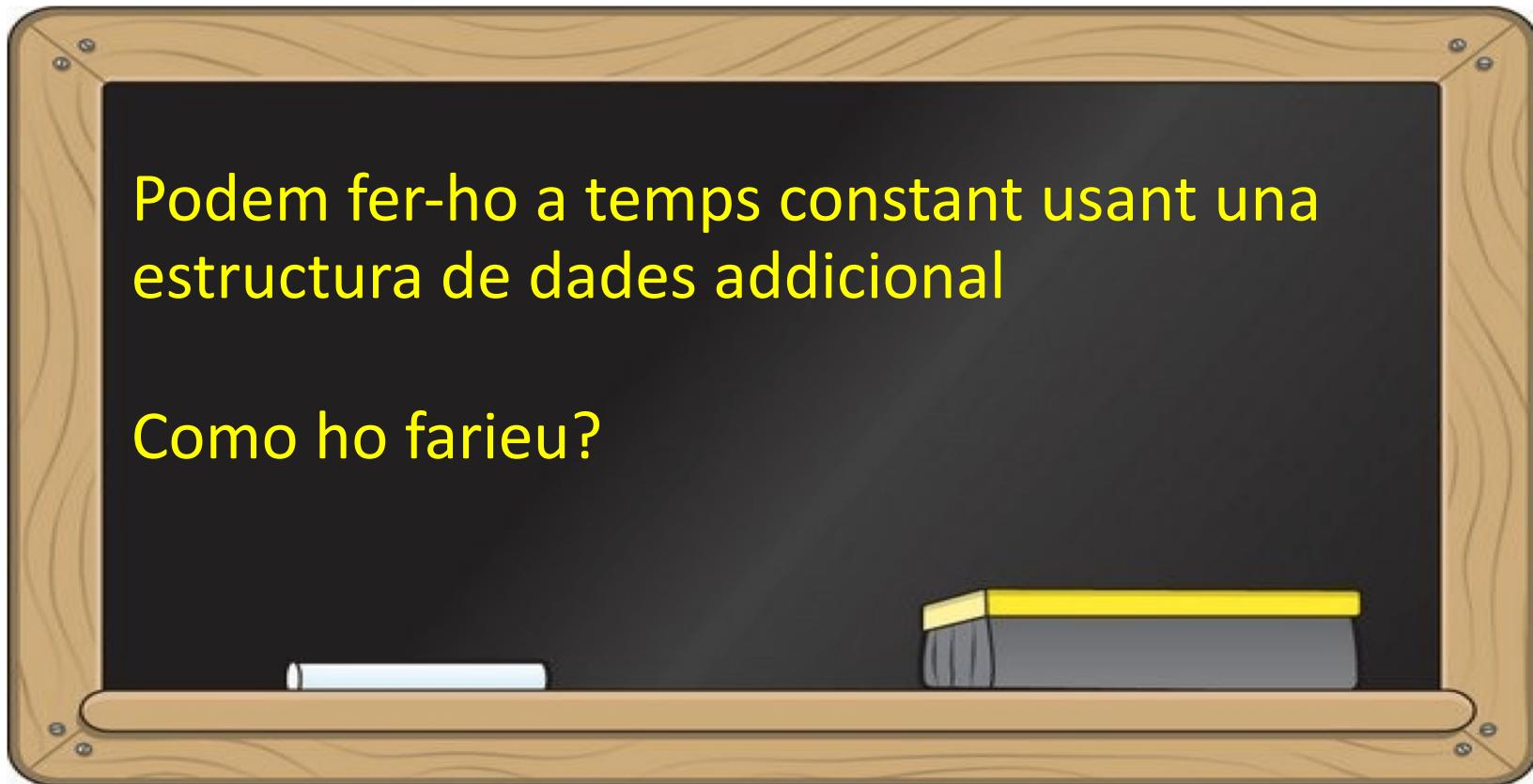


Com trobar el “Insertion Node”

- Per trobar el “Insertion Node” hem vist que tenim un algorisme $O(\log n)$.

Podem fer-ho a temps constant usant una estructura de dades addicional

Como ho farieu?

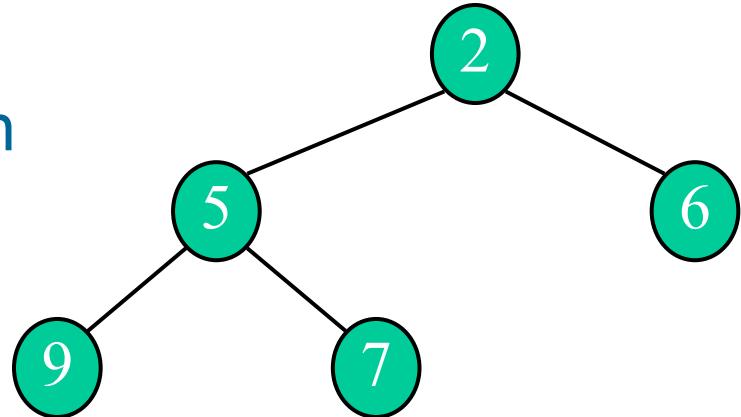




5.4 Com implementar els heaps

Implementació amb Arrays

- Podem representar un heap de n claus mitjançant un array de longitud $n+1$
- Implementació
 - Per el node en l'índex i
 - El fill esquerra es troba a la posició $2i+1$
 - El fill dret es troba a la posició $2i+2$
- Operacions
 - `insert` correspon a inserir en l'índex $n+1$
 - `removeMin` correspon a intercanviar amb l'índex n , intercanvia i eliminar



Complexitat Operacions

Implementació	insert	removeMin
Unsorted Array		
Sorted Array		
Unsorted Linked List		
Sorted Linked List		
Heap		
Hash Table		

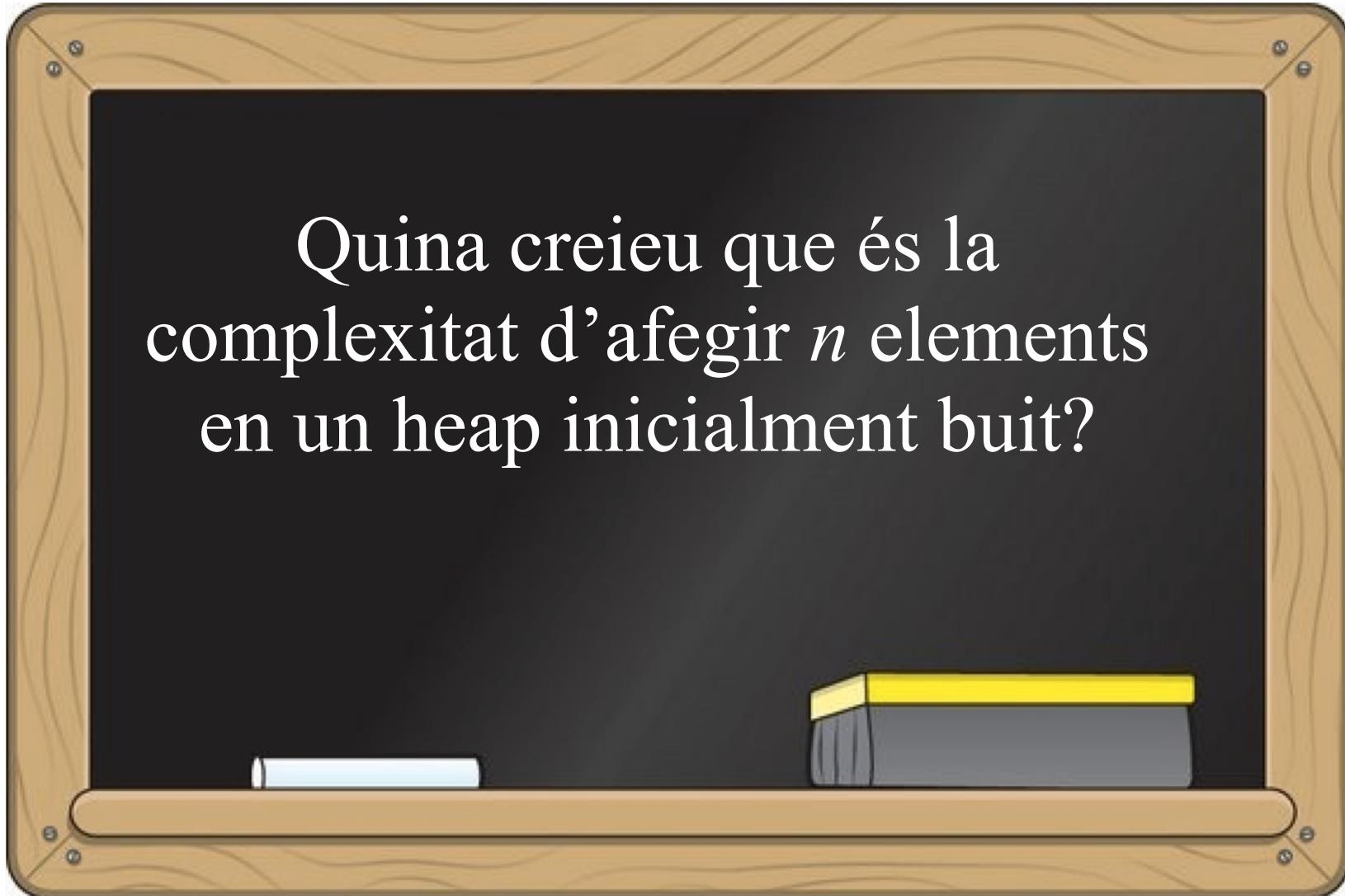
Complexitat Operacions

Implementació	insert	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		



Complexitat per crear un HEAP

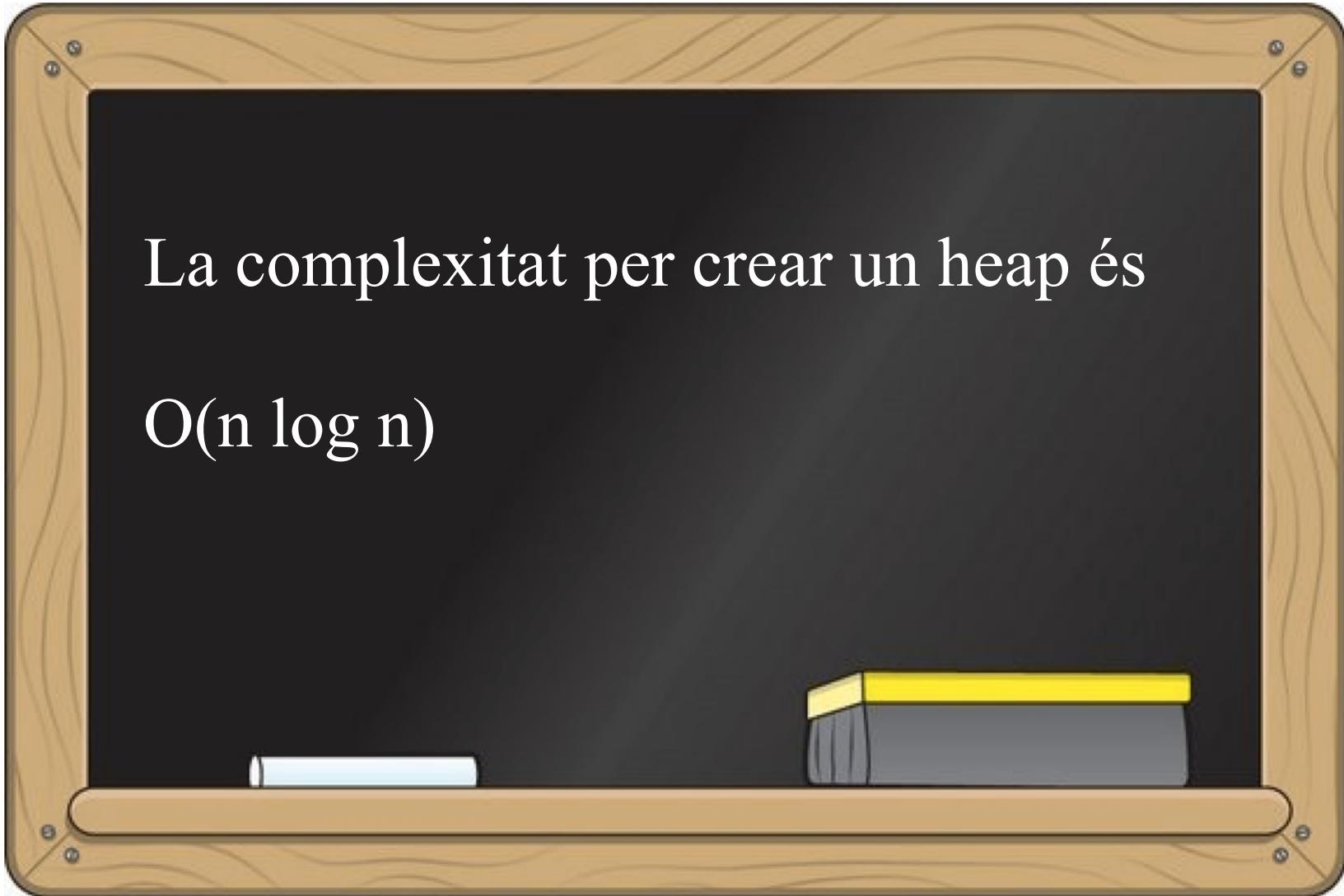
Quina creieu que és la complexitat d'afegir n elements en un heap inicialment buit?



Complexitat per crear un HEAP

La complexitat per crear un heap és

$O(n \log n)$





Exercici

Exercici:

Escriviu el codi del mètode insert del TAD heap

```
template<class Element>
class MinHeap{
    private:
        int heap_size;
        std::vector<Position<Element>> data;
        // Aquí sota aniran els mètodes
};
```



5.5 HeapSort



HeapSort

- Considera una cua de prioritat amb n ítems implementada amb un heap
 - l'espai usat és $O(n)$
 - els mètodes insert i removeMin tenen un cost de $O(\log n)$ en temps
 - Els mètodes size, empty, i min tenen un cost $O(1)$ en temps
- Usant una cua prioritària implementada amb un heap, es pot ordenar una seqüència de n elements amb un cost $O(n \log n)$ en temps
- L'algorisme resultant s'anomena heapsort
- **HeapSort és més ràpid que els mètodes d'ordenació d'inserció i de selecció**

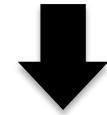


HeapSort

- El **HeapSort** és un algorisme d'ordenació basat en comparacions d'elements on s'utilitza un **Heap**

Input

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



Output

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

HeapSort

Pas 1: 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

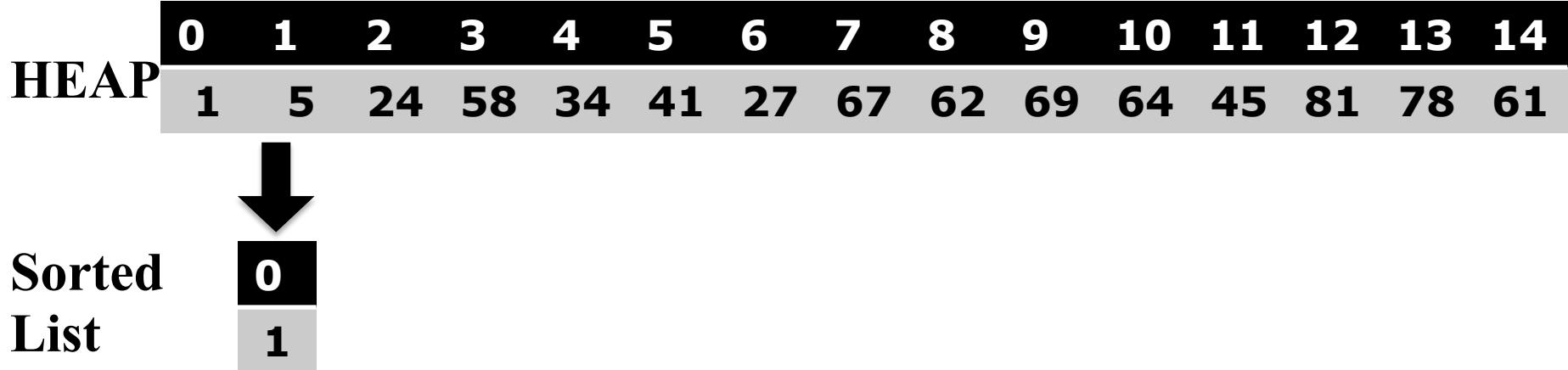


HeapSort

- Pas 2. Utilitzar Heap per 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

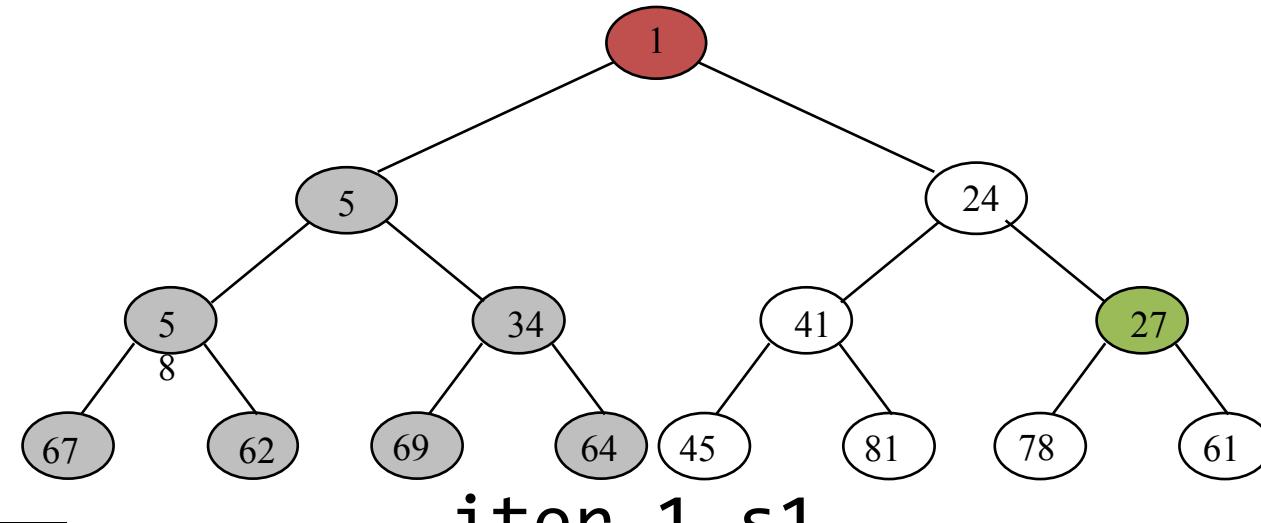
- Anem traient el menor element i es posa en una nova llista: >`removeMin()`



Exemple: 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

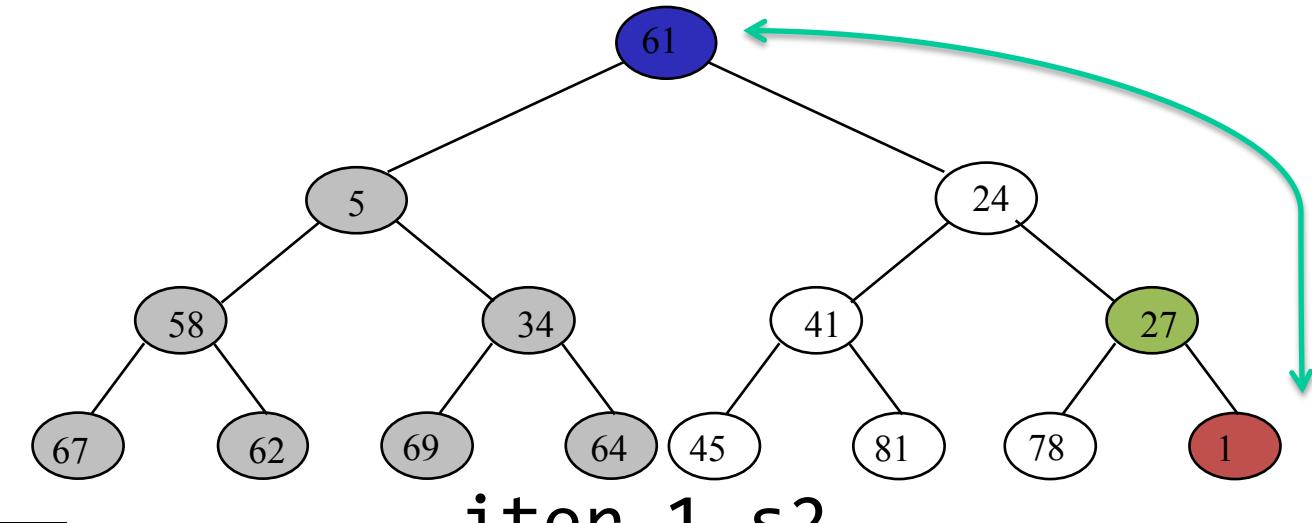
Sorted
List

0
1

Exemple: 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



iter 1-s2

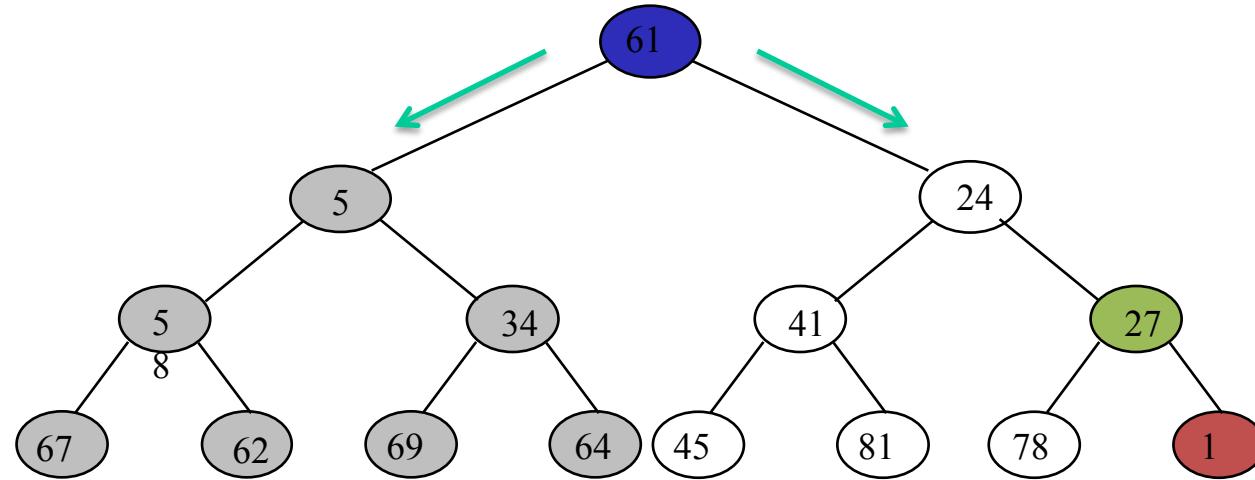
Sorted
List

0
1

Exemple: 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



iter 1-s3

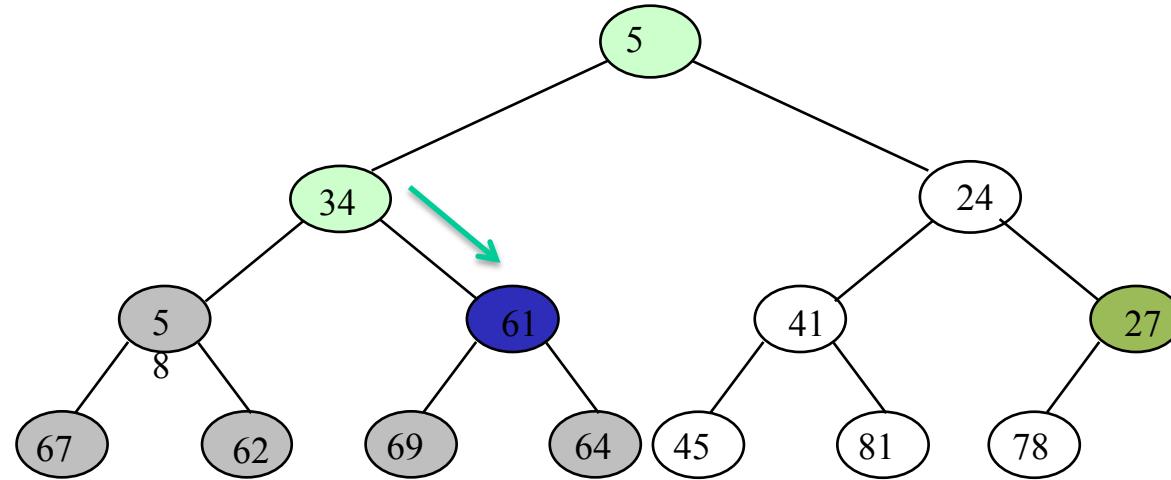
Sorted
List

0
1

Exemple: 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

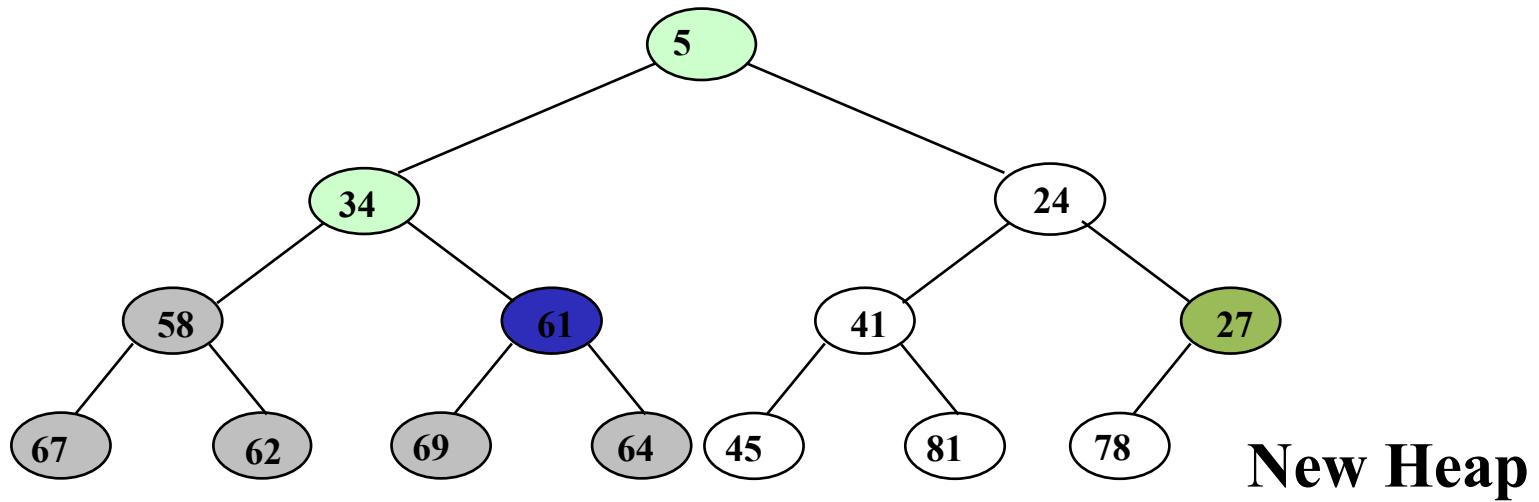
0
1

iter 1-s4

Exemple: 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

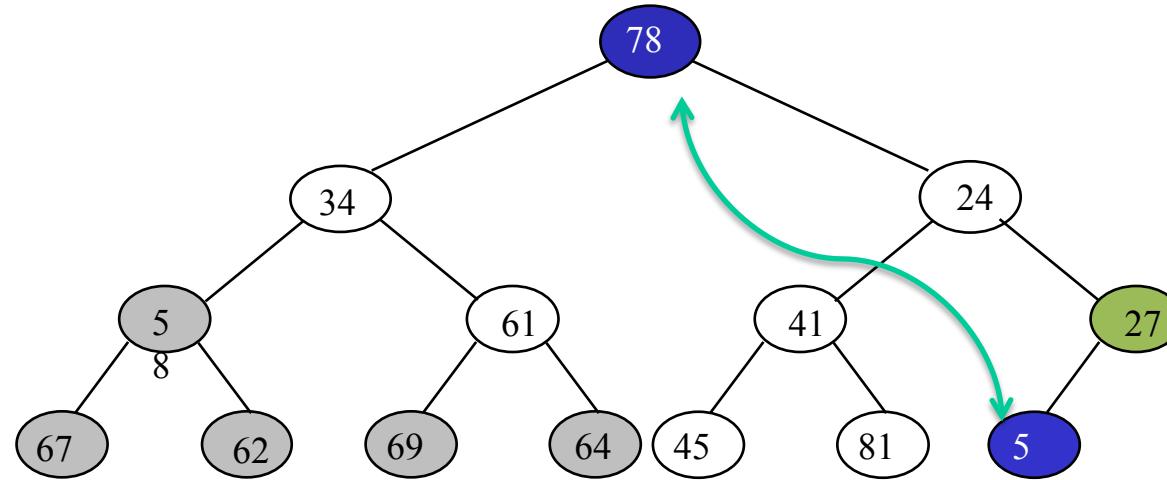


Sorted List	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	5	34	24	58	61	41	27	67	62	69	64	45	81	78

Exemple: 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



Iter 2-s1

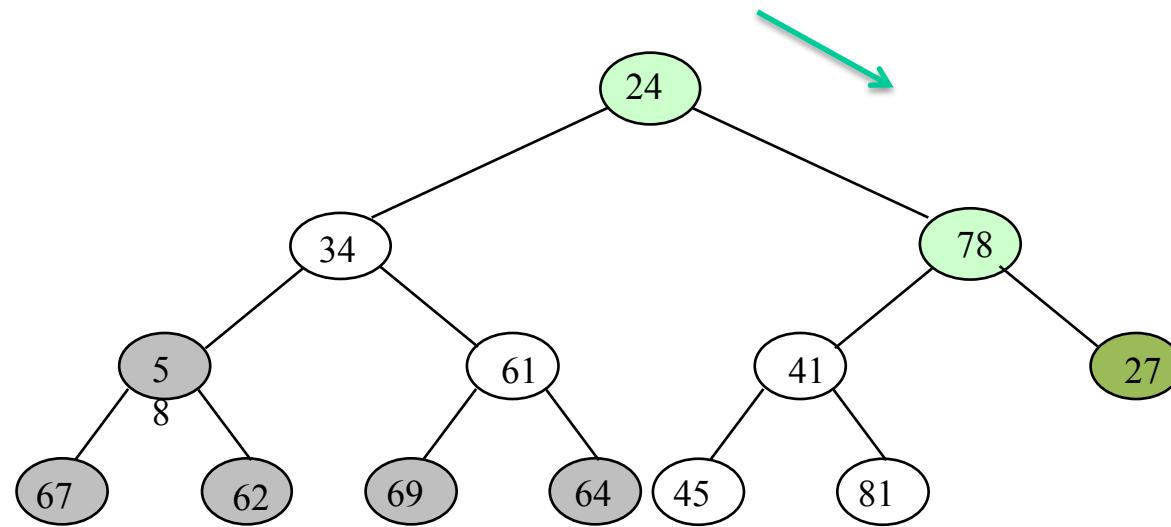
Sorted
List

0	1
1	5

Exemple: 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



Iter 2-s2

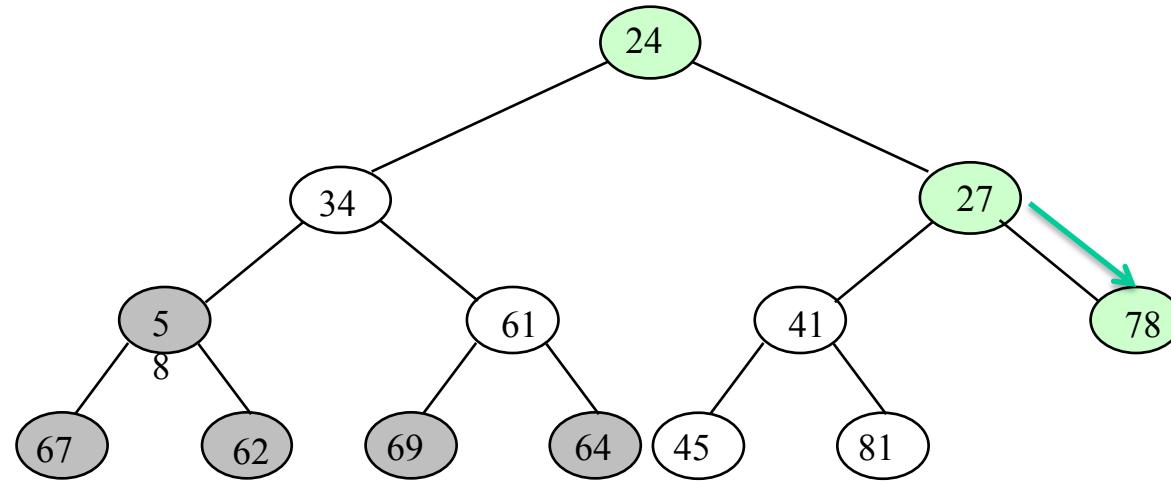
Sorted
List

0	1
1	5

Exemple: 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



Iter 2-s2

Sorted
List

0	1
1	5



Complexitat del HeapSort

Quina creieu que és la complexitat del heapsort?



Complexitat del HeapSort

La complexitat del HeapSort és $O(n \log n)$



Tema 5 Heaps

Maria Salamó Llorente
Estructura de Dades

Grau d'Enginyeria Informàtica

Facultat de Matemàtiques i Informàtica,

Universitat de Barcelona