In questa lezione

- **Code di priorità**

[CLRS01] cap. 6 da pag. 114 a pag. 117

Coda di priorità: cos'è?

- **Una coda di priorità è una struttura dati dinamica che permette di gestire dei dati con chiave numerica, la loro priorità.**

- **In una coda la modalità di prelievo è basata sull'ordine di inserimento: il primo inserito è il primo a essere estratto**

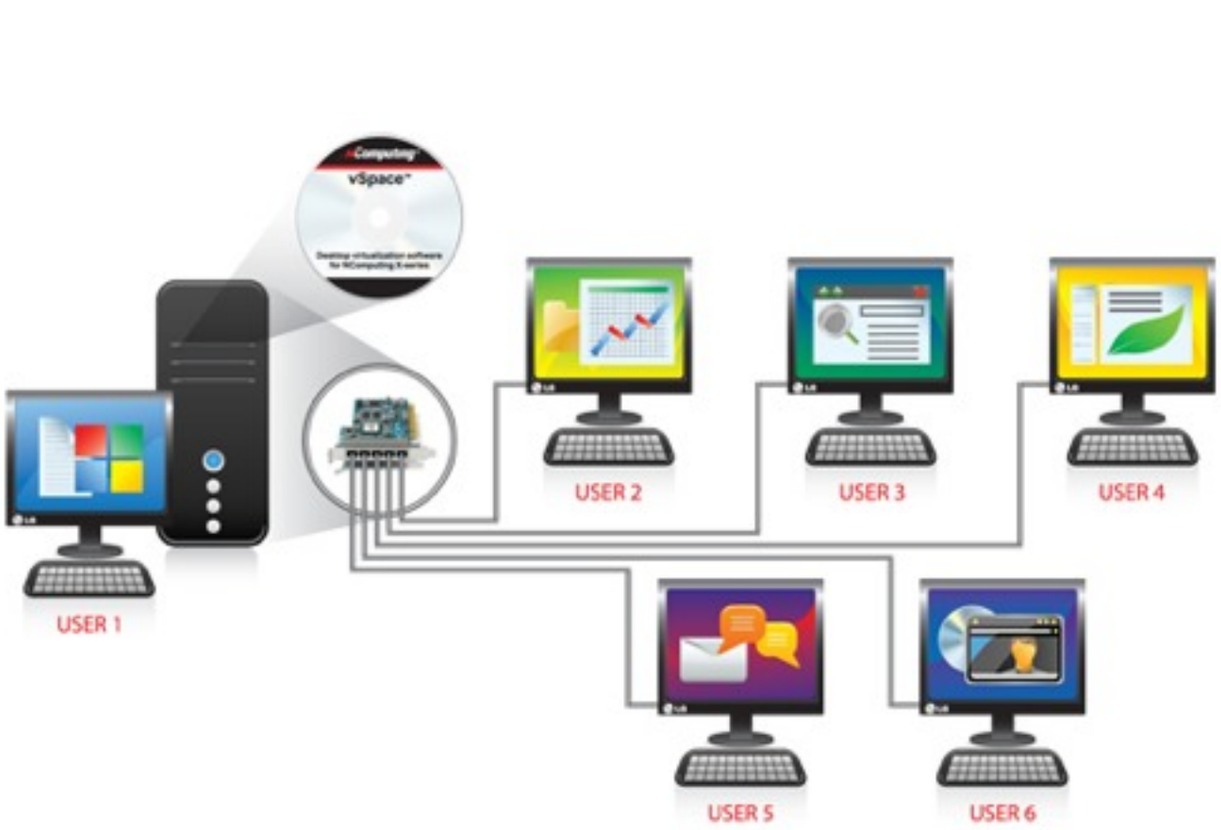
- **In una coda di priorità la modalità di prelievo si basa sulla priorità associata agli elementi**

Coda di max-priorità: le operazioni

- **Una coda di max-priorità offre le operazioni di**
  - **Insert(S,x): inserisce x in S**
  - **Maximum(S): restituisce l'elemento con priorità massima**
  - **Extract-Max(S): restituisce l'elemento con priorità massima e lo rimuove dall'insieme**
  - **Increase-key(S,x,k): cambia la priorità di x al valore k, se maggiore o uguale di quella presente**

Applicazioni

In caso di condivisione di un computer fra più utenti: viene svolto per primo il lavoro con più alta priorità



Coda di min-priorità: le operazioni

- **Una coda di min-priorità offre le operazioni di**

  - **Insert(S,x): inserisce x in S**

  - **Minimum(S): restituisce l'elemento con priorità minima**

  - **Extract-Min(S): restituisce l'elemento con priorità minima e lo rimuove dall'insieme**

  - **Decrease-key(S,x,k): porta a k la priorità di x, se k è minore di quella presente**

Applicazioni

Simulazione guidata da eventi: Gli elementi della coda sono quelli da simulare e ad ogni elemento è associato il tempo nel quale si può verificare.

Implementazione

- **Per implementare una coda con priorità si potrebbero utilizzare gli array**
- **In ogni implementazione si utilizza un handle (aggancio), che può essere un puntatore o un intero, per legare l'oggetto alla sua priorità nell'array e viceversa (qui basta un intero).**

implementazione	insert	Extract-Max	Max
array non ordinato	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
array ordinato	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Gli heap consentono un'implementazione efficiente per tutte queste operazioni!

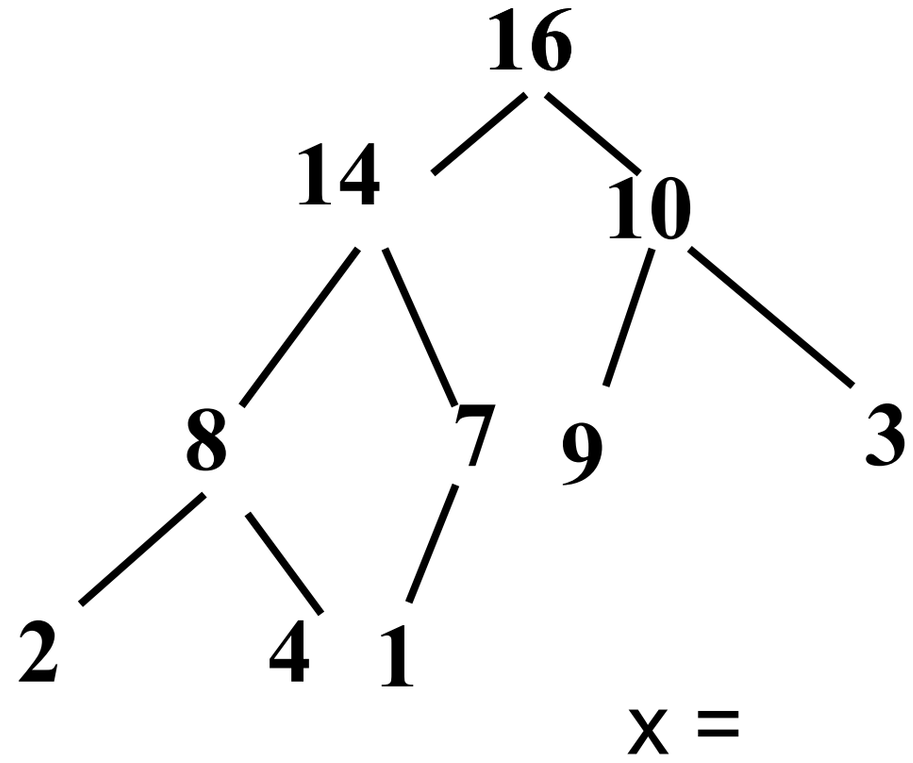
Implementazione Maximum(A)

A=

16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

Heap-Maximum (A)
return A[1]

Θ(1)

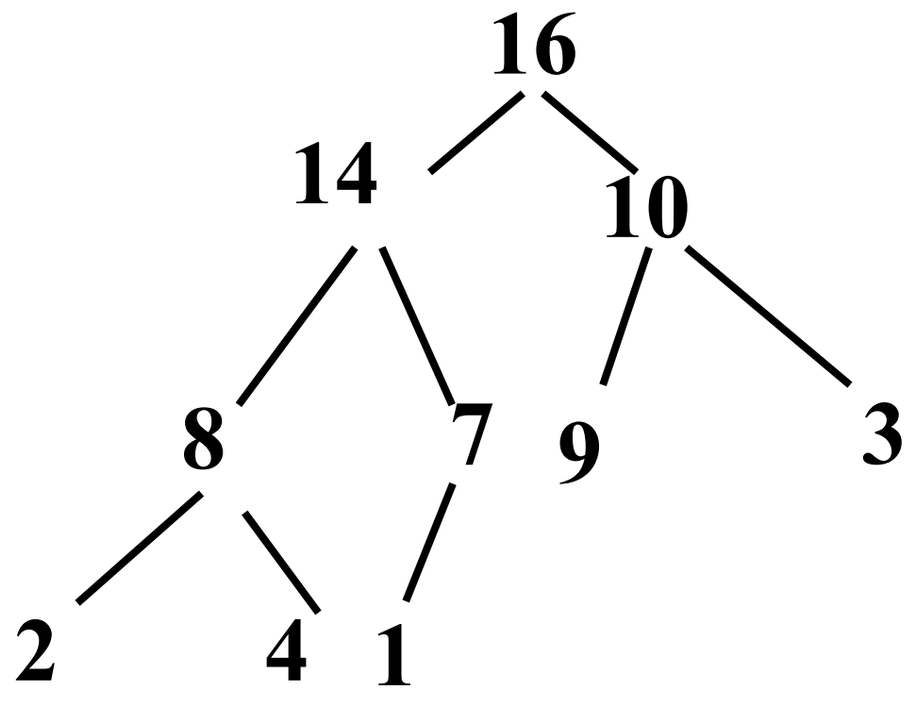


Implementazione Extract-Max(A)

A=

16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

x =



??

Implementazione Extract-Max(A)

A=

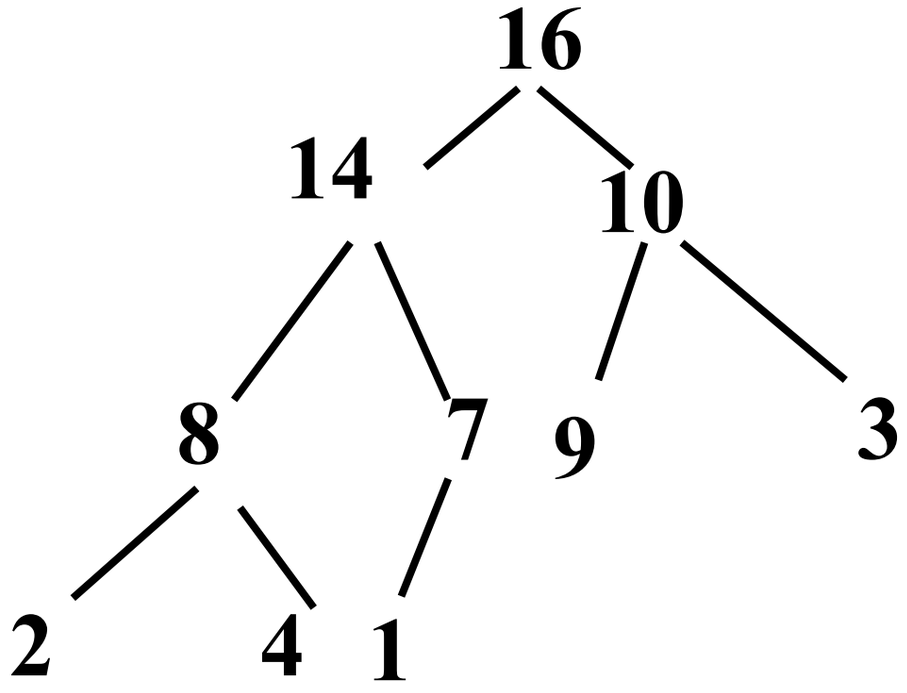
16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

Heap-extract-Max(A)

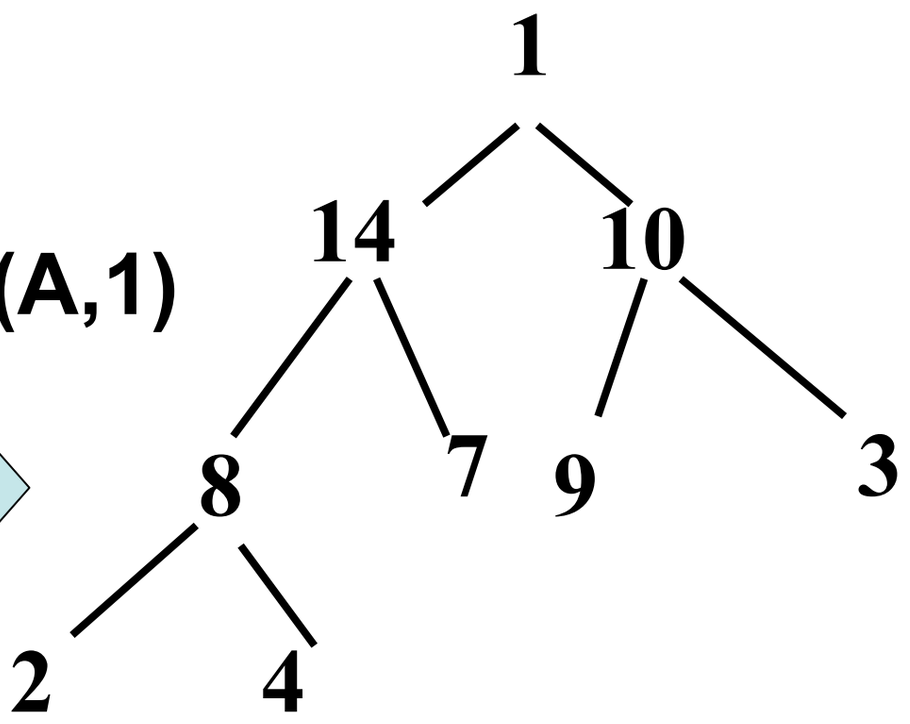
A=

14	8	10	4	7	9	3	2	1	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

max =



Max-Heapify(A,1)



Implementazione Extract-Max(A)

Heap-Extract-Max (A)

▷ **A è un max-heap, la procedura restituisce il massimo e ripristina la proprietà del max-heap sui rimanenti**

if heap-size[A] < 1 then "errore: l'heap è vuoto
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
Max-Heapify (A,1)

Θ(lg n),
n = heap-size(A)

Max-Heapify

Max-Heapify (A,i)

▷ Max- Heapify fa' sì che la proprietà del max-heap sia verificata per il nodo i, sotto l'ipotesi che lo sia per i sottoalberi radicati nei suoi figli.

Input: un l'array A tale che A[left(i)] e A[right(i)] siano radici di max-heap e in cui A[i] può essere più piccolo dei suoi figli

Output: l'array A è tale che anche A[i] è radice di un max-Heap

```
s ← left(i)
d ← right(i)
if s ≤ heap-size[A] and A[s] > A[i] then
    massimo ← s else massimo ← i
if d ≤ heap-size[A] and A[r] > A[massimo] then
    massimo ← r
 if massimo ≠ i then
    scambia A[i] e A[massimo]
     Max-Heapify (A,massimo)
```

Θ(lg n),

n = heap-size(A)

Implementazione Increase-Key

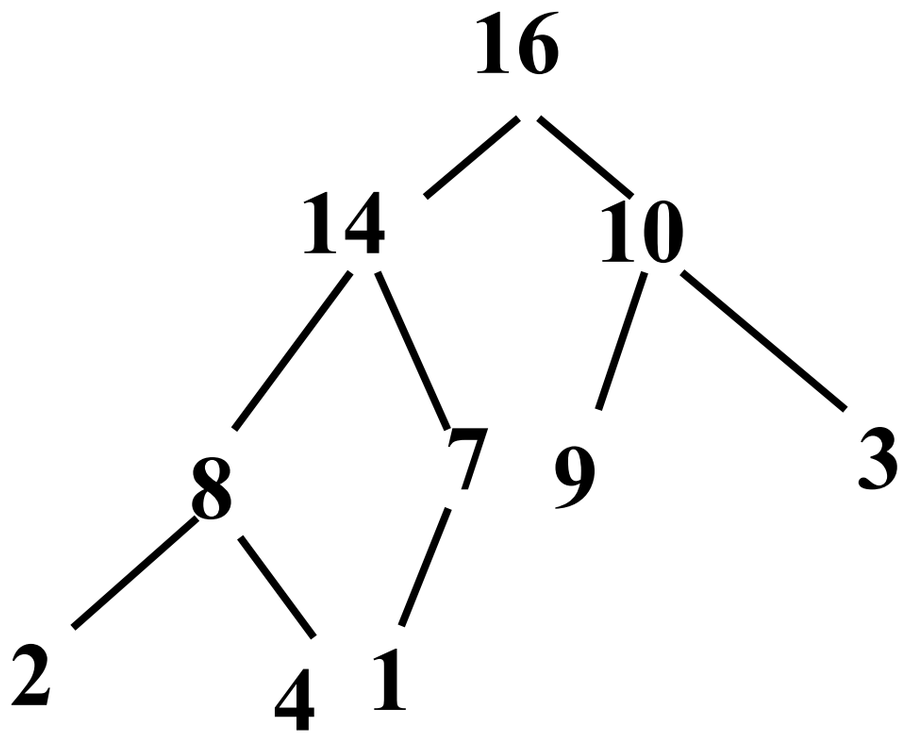
A=

16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

Increase-key(A,15)

A=

16	15	10	14	7	9	3	2	8	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13



4 ⇨ 15

15

Correttezza Increase-key

Heap-increase-key(A,i,key)

▷ A è un max-heap. Sostituisce key ad A[i], se key ≥ A[i] e ripristina la proprietà del max-heap

if key < A[i] then "errore: la nuova chiave è più piccola"
A[i] ← key
while i > 1 and A[Parent(i)] < A[i]

Invariante: L'array A[1...heap-size] soddisfa la proprietà del max-heap tranne al più nella posizione i, se 1≤i≤heap-size, perchè A[i] potrebbe essere maggiore di A[Parent(i)]

do scambia A[Parent(i)] con A[i]
i ← Parent(i)

Correttezza Increase-key

Heap-increase-key(A,i,key)

▷ **A è un max-heap. Sostituisce key ad A[i], se key ≥ A[i] e ripristina la proprietà del max-heap**

if key < A[i] then "errore: la nuova chiave è più piccola"

A[i] ← key

while i > 1 and A[Parent(i)] < A[i]

Invariante: L'array A[1...heap-size] soddisfa la proprietà del max-heap tranne al più nella posizione i, se 1≤i≤heap-size, perchè A[i] potrebbe essere maggiore di A[Parent(i)]

do scambia A[Parent(i)] con A[i]

i ← Parent(i)

Inizializzazione: L'invariante è banalmente vero prima di ogni iterazione visto che il valore in A[i] all'entrata nel ciclo è diventato key e quindi A[i] può essere maggiore di A[Parent(i)]

Correttezza Increase-key

Heap-increase-key(A,i,key)

▷ **A è un max-heap. Sostituisce key ad A[i], se key ≥ A[i] e ripristina la proprietà del max-heap**

if key < A[i] then "errore: la nuova chiave è più piccola"

A[i] ← key

while i > 1 and A[Parent(i)] < A[i]

Invariante: L'array A[1...heap-size] soddisfa la proprietà del max-heap tranne al più nella posizione i, se 1≤i≤heap-size, perchè A[i] potrebbe essere maggiore di A[Parent(i)]

do scambia A[Parent(i)] con A[i]

i ← Parent(i)

Conservazione: supponiamo che l'array A[1...heap-size] soddisfi la proprietà del max-heap, tranne al più in A[k] che potrebbe essere maggiore di A[Parent(k)].
Se questo è il caso lo scambio assicura che la proprietà del max-heap è ripristinata per A[k],visto che il nuovo padre è maggiore anche del (vecchio) fratello, per transitività

Correttezza Increase-key

Heap-increase-key(A,i,key)

▷ A è un max-heap. Sostituisce key ad A[i], se key ≥ A[i] e ripristina la proprietà del max-heap

if key < A[i] then "errore: la nuova chiave è più piccola"

A[i] ← key

while i > 1 and A[Parent(i)] < A[i]

Invariante: L'array A[1...heap-size] soddisfa la proprietà del max-heap tranne al più nella posizione i, se 1≤i≤heap-size, perchè A[i] potrebbe essere maggiore di A[Parent(i)]

do scambia A[Parent(i)] con A[i]

i ← Parent(i)

Conclusione: sappiamo che l'array A[1...heap-size] soddisfa la proprietà del max-heap, tranne al più in A[i] che potrebbe essere maggiore di A[Parent(i)], ma i=0 o A[Parent(i)]≥ A[i], quindi la proprietà del max-heap è soddisfatta ovunque.

Increase-key: complessità

Heap-increase-key(A,i,key)

▷ A è un max-heap. Sostituisce key ad A[i], se key ≥ A[i] e ripristina la proprietà del max-heap

```
if key < A[i] then "errore: la nuova chiave è più piccola"
 A[i] ← key
while i > 1 and A[Parent(i)] < A[i]
 do scambia  A[Parent(i)] con  A[i]
      i ← Parent(i)
```

Caso peggiore: Θ(lg n),
n = heap-size(A)

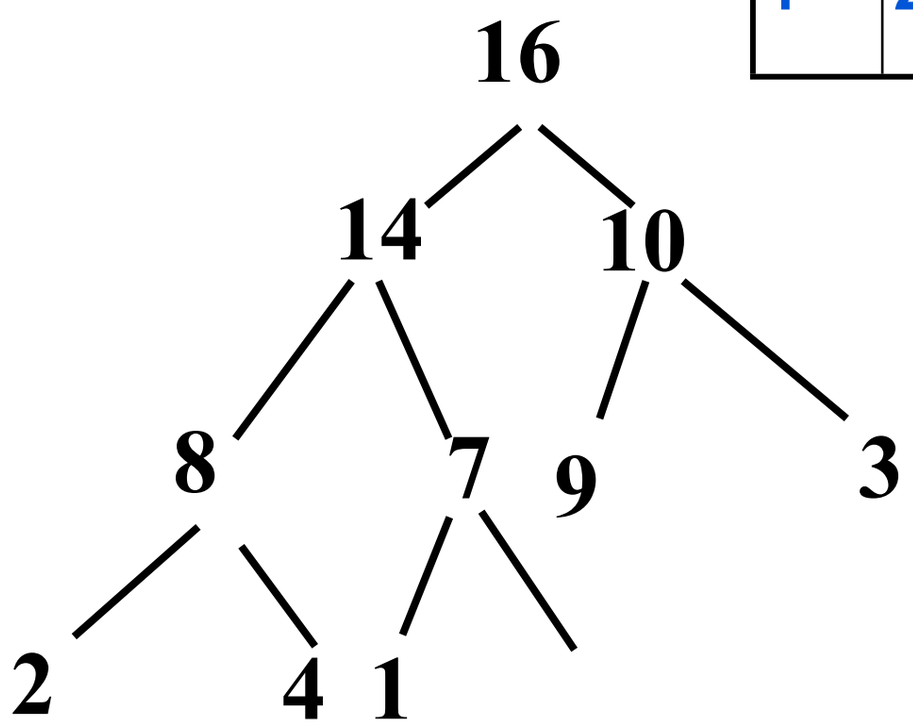
Implementazione Insert(A,x)

A=

16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

heap-size(A)=10

Insert(A,15)



A=

16	15	10	8	14	9	3	2	4	1	7	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

heap-size(A)=11

15

Implementazione Insert(A,x)

Max-Heap-insert(A,key)

▷ A è un max-heap. Inserisce key in A, ripristinando la proprietà del max-heap

if heap-size[A] = length[A] then "errore: l'heap è pieno

heap-size[A] ← heap-size[A] +1

A[heap-size[A]] ← - ∞

▷ - ∞ indica un elemento certamente più piccolo di key

Heap-increase-key(A,heap-size[A],key)

Caso peggiore: Θ(lg n),
n = heap-size(A)

Sommario dei costi

implementazione	insert	Extract-Max	Max
array non ordinato	Θ(1)	Θ(n)	Θ(n)
array ordinato	Θ(n)	Θ(1)	Θ(1)
heap	Θ(lg n)	Θ(lg n)	Θ(1)

Complessità nel caso peggiore di diverse implementazioni di una coda di priorità con n elementi

Cancellare un elemento

Può essere inoltre utile l' operazione

Delete(A,i): cancella A[i] da A

Implementazione Delete(A,x)

A=

16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

heap-size(A)=10

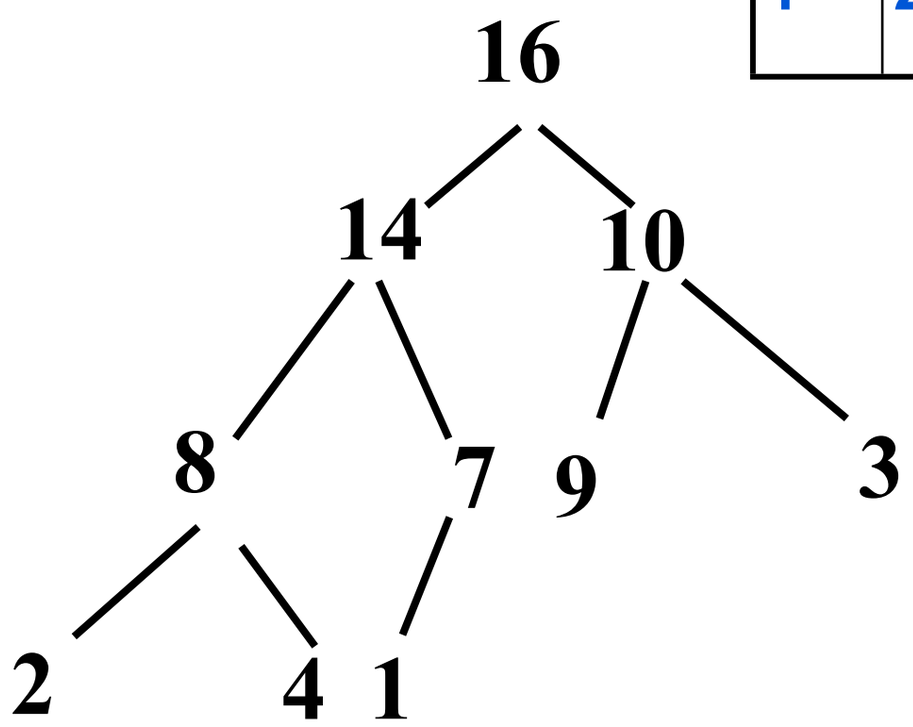
Delete(A,2)

L'ultima foglia al posto dell'elemento da cancellare

A=

16	1	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

heap-size(A)=9



Proprietà del maxHeap violata rispetto ai figli:

Max-heapfy(A,2)

Implementazione Delete(A,x)

A=

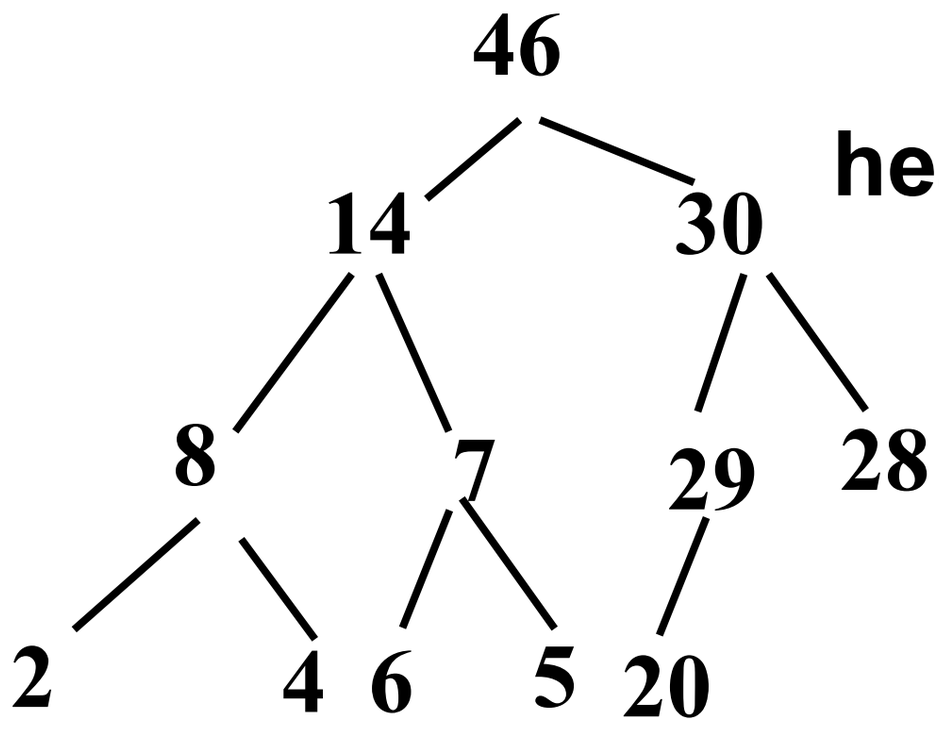
46	14	30	8	7	29	28	2	4	6	5	20	7
1	2	3	4	5	6	7	8	9	10	11	12	13

heap-size(A)=12

Delete(A,4)

A=

46	20	30	14	7	29	28	2	4	6	5	20	7
1	2	3	4	5	6	7	8	9	10	11	12	13



heap-size(A)=11

20 va al posto di 8, ma è maggiore di 14!