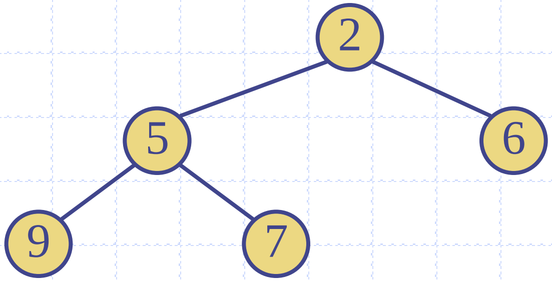
Heap



TDA Coda di Priorità (§ 8.1.3)

- Una coda di priorità memorizza una collezione di elementi
- Ogni **entry** è una coppia (key, value)
- Metodi principali del TDA Priority Queue
  - insert(k, x)
    inserisce un entry con chiave k e valore x
  - removeMin()
    elimina e restituisce l'entry con chiave minore
- Metodi addizionali
  - min()
    restituisce, ma non rimuove, un entry con chiave minore
  - size(), isEmpty()
- Applicazioni:
  - Voli in attesa di partire
  - Aste
  - Mercato azionario

Richiamo di Sorting basato su Coda di Priorita'

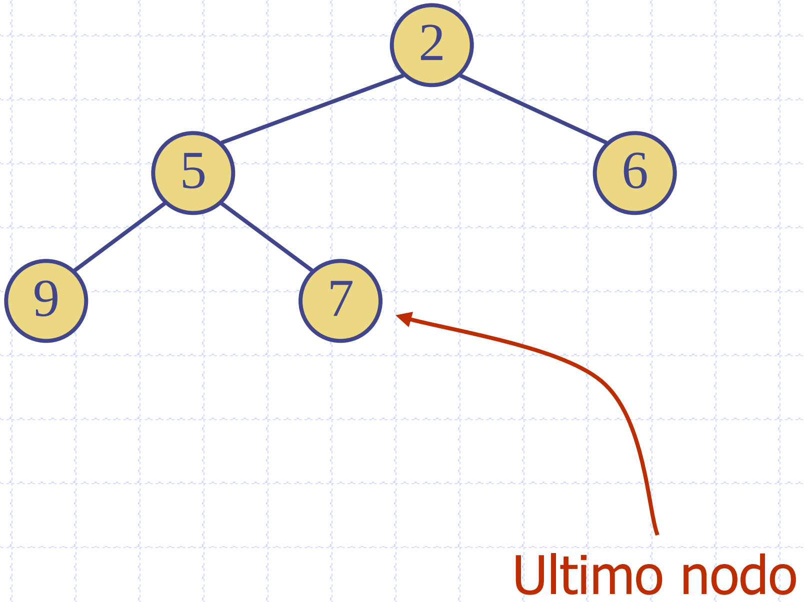
- Possiamo usare una Coda di Priorita' per ordinare un insieme di elementi confrontabili
  - Inserisci gli elementi con una serie di operazioni di insert
  - Rimuovi gli elementi in sequenza ordinata con una serie di operazioni di removeMin
- Il tempo di esecuzione dipende dalla implementazione della coda di priorita' :
  - Sequenza non ordinata per selection-sort: $O(n^2)$ time
  - Sequenza ordinata per insertion-sort: $O(n^2)$ time
- Possiamo fare di meglio?

Algorithm ***PQ-Sort***(***S***, ***C***)
  Input sequence ***S***, comparator ***C*** for the elements of ***S***
  Output sequence ***S*** sorted in increasing order according to ***C***
  P ← priority queue with
    comparator ***C***
  while ¬***S.isEmpty*** ()
    e ← ***S.remove*** (***S. first*** ())
    P.insertItem(***e***, ***e***)
  while ¬***P.isEmpty***()
    e ← ***P.removeMin***()
    S.insertLast(***e***)

Heap (§8.3)

- Un heap e' un albero binario che memorizza chiavi sui nodi e soddisfa le seguenti proprieta':
  - Heap-Order: per ogni nodo interno diverso dalla radice, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: sia h l'altezza dell'heap:
    - for $i = 0, \ldots, h-1$, ci sono 2^i nodi di profondita' i
    - A profondita' $h-1$, i nodi interni sono alla sinistra dei nodi esterni

- L'ultimo nodo di un heap e' il nodo piu' a destra di profondita' h

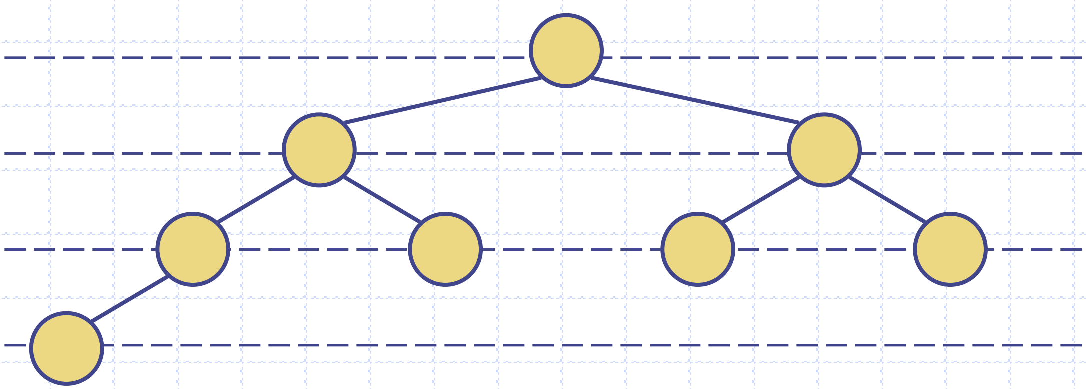


Altezza di un Heap (§ 8.3.1)

- Teorema: Un heap che memorizza n chiavi ha altezza $O(\log n)$

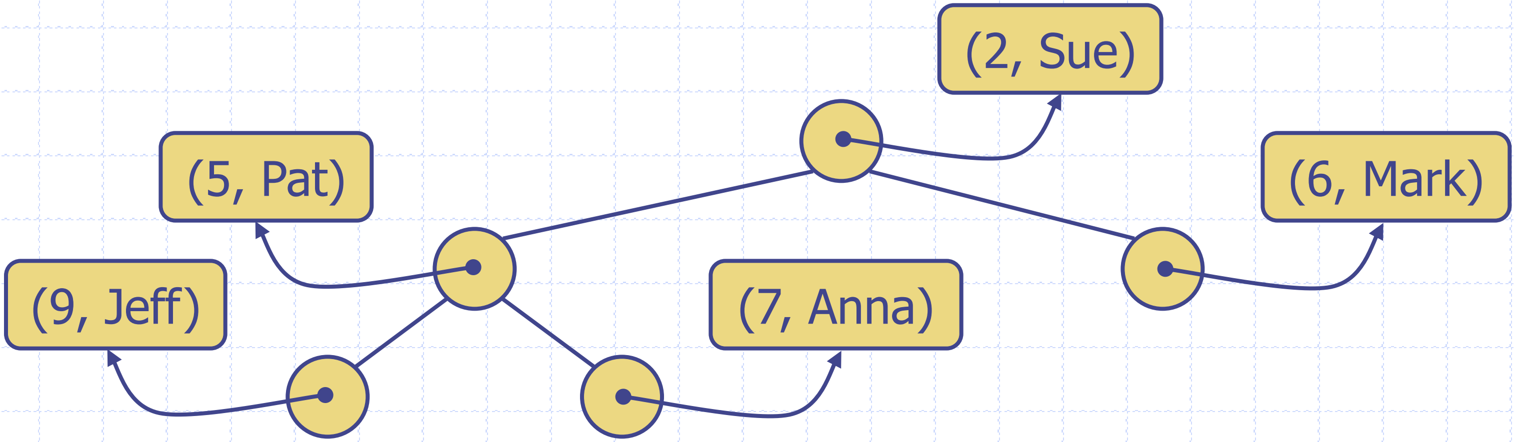
  Prova: (applichiamo la proprieta' di albero binario completo)
  - Sia h l'altezza di un heap che memorizza n chiavi
  - Poiche ci sono 2^i chiavi a profondita' $i = 0, \dots, h-1$ e almeno una chiave a profondita' h, abbiamo $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
  - Quindi, $n \geq 2^h$, e quindi $h \leq \log n$

prof.	chiavi
0	1
1	2
$h-1$	2^{h-1}
h	1



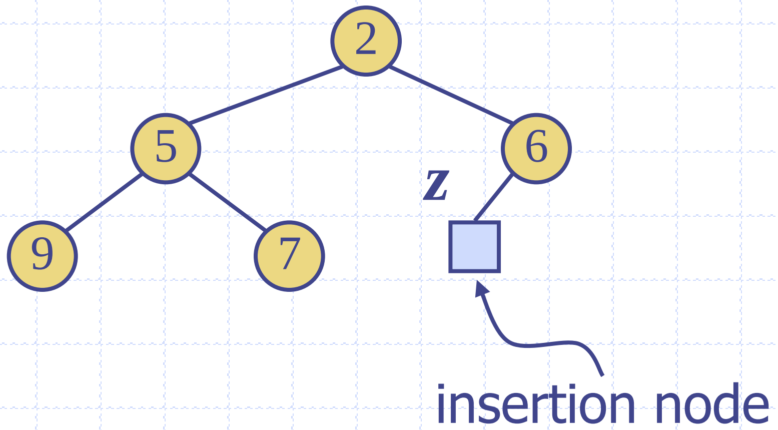
Heap e Code di Priorita’

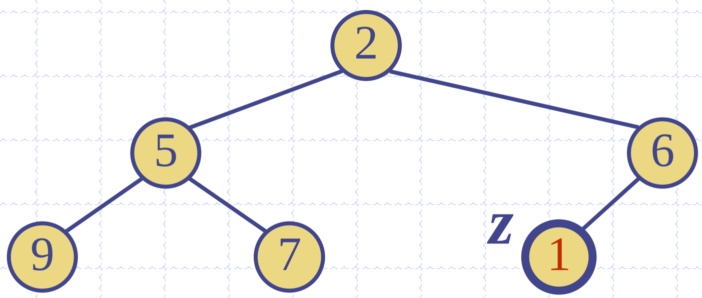
- Possiamo usare un heap per implementare una coda di priorita’
- Memorizziamo una coppia (key, element) ad ogni nodo interno
- Teniamo traccia della posizione dell’ultimo nodo
- Per semplicita’ mostriamo solo le chiavi nella figura



Inserimento in un Heap (§ 8.3.3)

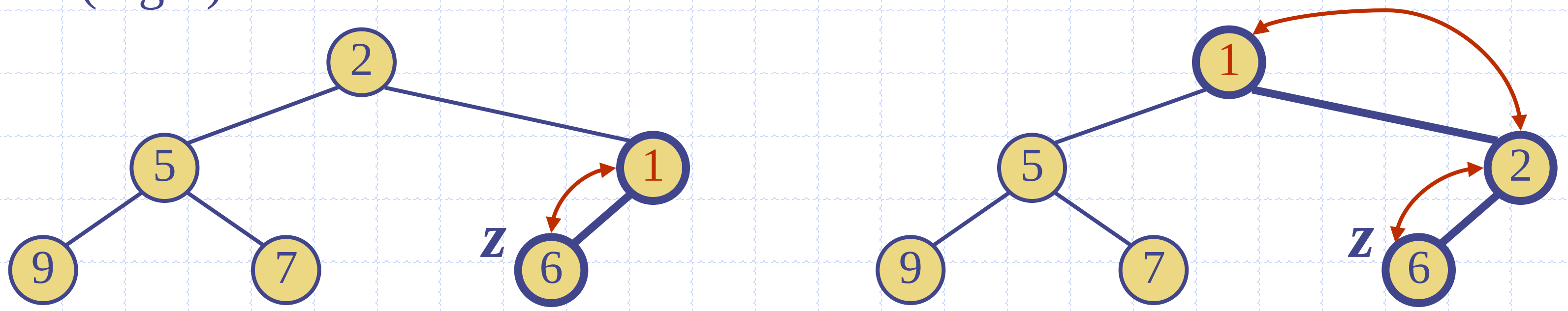
- Metodo insertItem del TDA coda di priorita' corrisponde all'inserimento di una chiave k nell'heap
- L'algoritmo di inserimento consiste di 3 fasi:
  - Trova il nodo di inserimento z (il nuovo ultimo nodo)
  - Memorizza k in z
  - Riporta l'ordine di heap (segue)





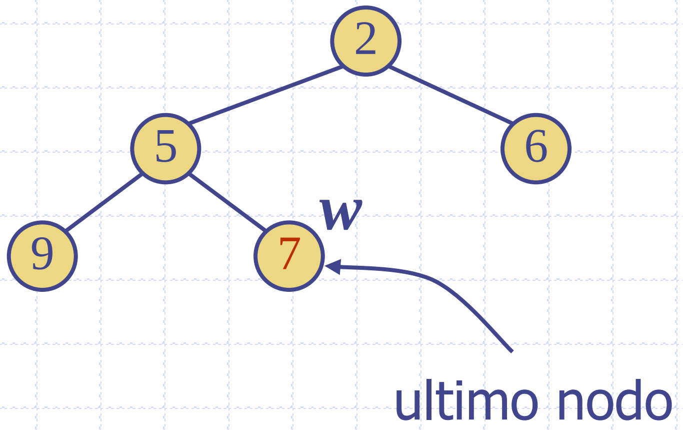
Upheap

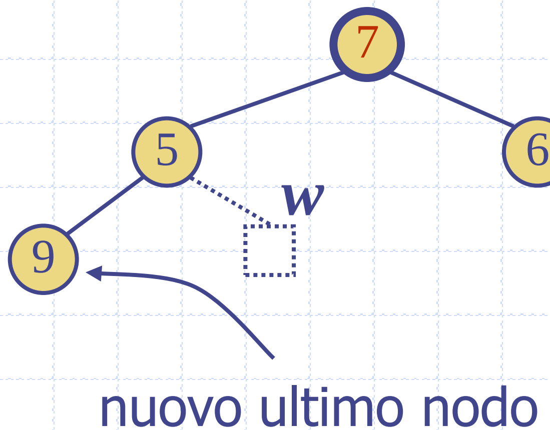
- Dopo l'inserimento di una nuova chiave k, l'ordine di heap potrebbe essere violato
- L'algoritmo upheap riporta l'ordine di heap scambiando la chiave k lungo il cammino ascendente dal nodo inserito
- Upheap termina quando la chiave k raggiunge la radice o un nodo il cui padre ha una chiave minore o uguale di k
- Poiche' un heap ha altezza $O(\log n)$, upheap e' eseguito in tempo $O(\log n)$



Rimozione da un Heap (§ 8.3.3)

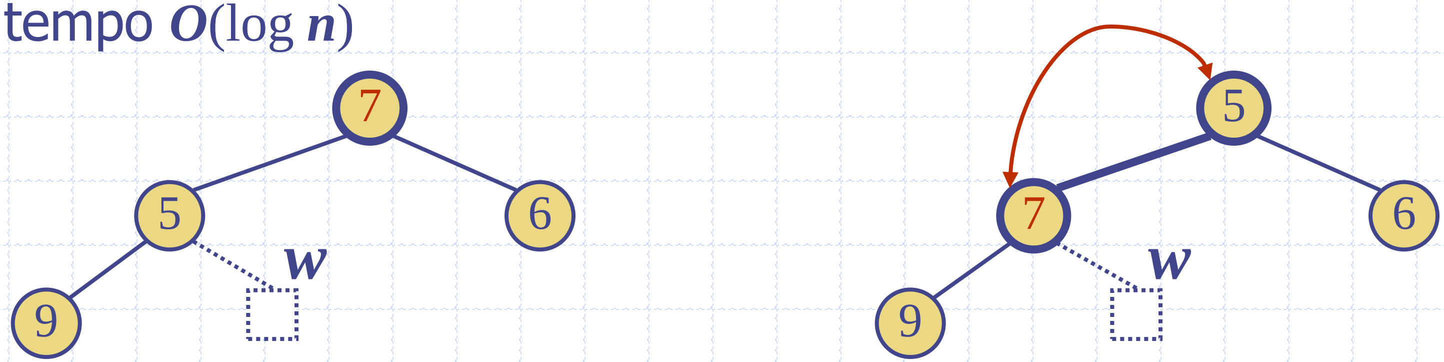
- Metodo removeMin del TDA coda di priorita' corrisponde alla rimozione della chiave radice dall'heap
- L'algoritmo di rimozione consiste di 3 passi
  - Sostituisci la chiave radice con la chiave dell'ultimo nodo w
  - Rimuovi w
  - Riporta la proprieta' di ordine di heap (segue)





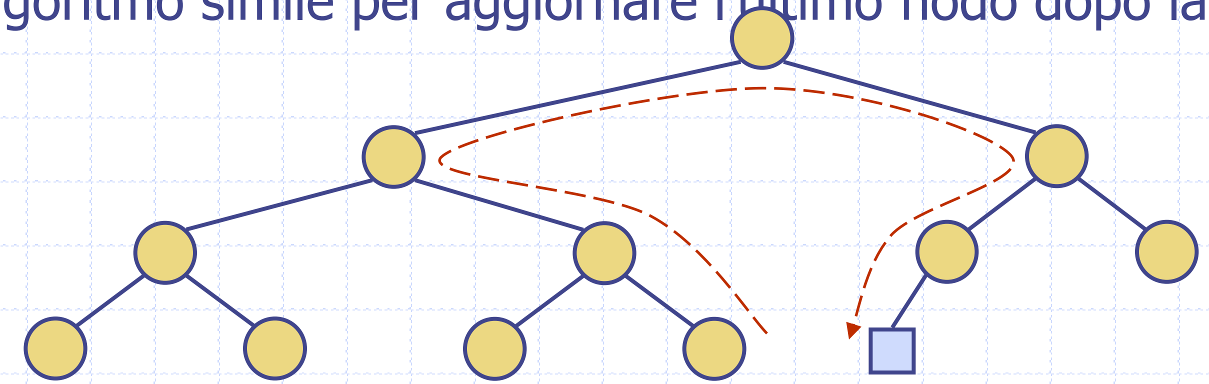
Downheap

- Dopo la sostituzione della chiave radice con la chiave k dell'ultimo nodo, la proprieta' di ordine di heap puo' essere violata
- L'algoritmo downheap riporta la proprieta' di ordine di heap scambiando la chiave k lungo un cammino discendente dalla radice
- Upheap termina quando la chiave k raggiunge una foglia o un nodo i cui figli hanno chiavi maggiori o uguali a k
- Poiche' un heap ha altezza $O(\log n)$, downheap e' eseguito in tempo $O(\log n)$

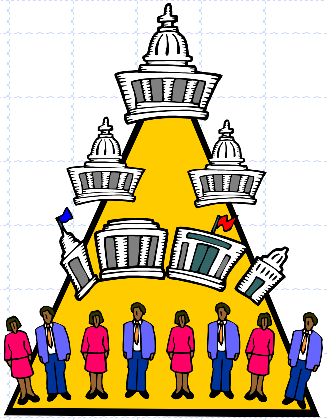


Aggiornamento dell'ultimo nodo

- Il nodo da inserire puo' essere identificato attraversando un cammino di $O(\log n)$ nodi
  - Segui il cammino ascendente finche' un figlio sinistro o la radice vengono raggiunti
  - Se un figlio sinistro e' raggiunto, vai al figlio destro
  - Procedi sul cammino discendente a sinistra finche' una foglia viene raggiunta
- Algoritmo simile per aggiornare l'ultimo nodo dopo la rimozione



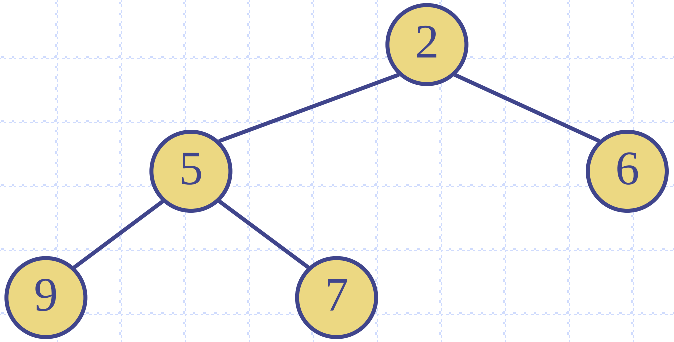
Heap-Sort (§8.3.5)

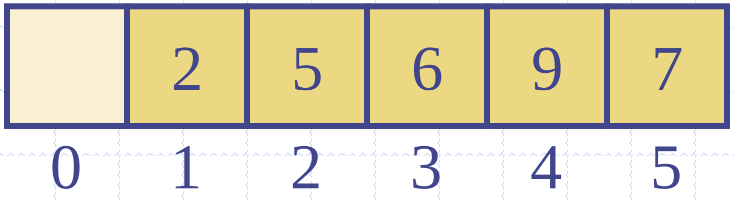


- Si conideri una coda di priorita' con n elementi implementati attraverso un heap
  - Lo spazio usato e' $O(n)$
  - Metodi insert e removeMin in tempo $O(\log n)$
  - Metodi size, isEmpty, e min in tempo $O(1)$
- Usando una coda di priorita' basata sugli heap, possiamo ordinare una sequenza di n elementi in tempo $O(n \log n)$
- L'algoritmo risultante e' chiamato heap-sort
- Heap-sort e' molto piu' veloce degli algoritmi di ordinamento con complessita' quadratica, come insertion-sort e selection-sort

Implementazione di un Heap basata su vettori

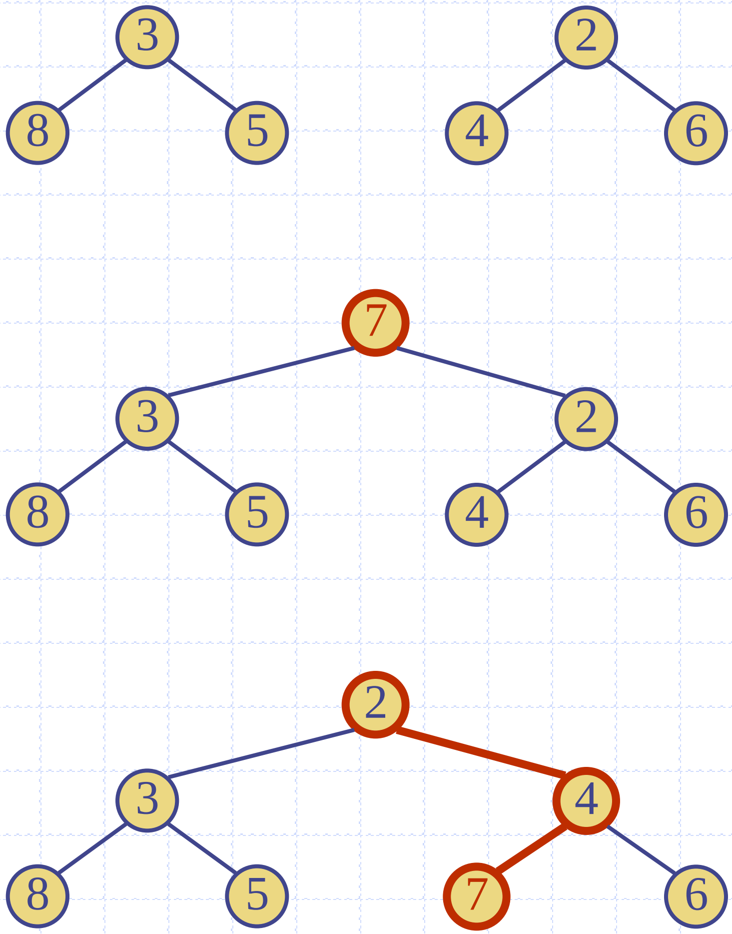
- Possiamo rappresentare un heap con n chiavi attraverso un vettore di lunghezza $n + 1$
- Per il nodo di rank i
  - il figlio sinistro ha rank $2i$
  - Il figlio destro ha rank $2i + 1$
- Gli archi tra i nodi non sono memorizzati esplicitamente
- La locazione di rank 0 non e' usata
- L'operazione di insert corrisponde all'inserimento alla posizione di rank $n + 1$
- L'operazione removeMin corrispoande alla rimozione alla poszione di rank n
- Otteniamo heap-sort sul posto





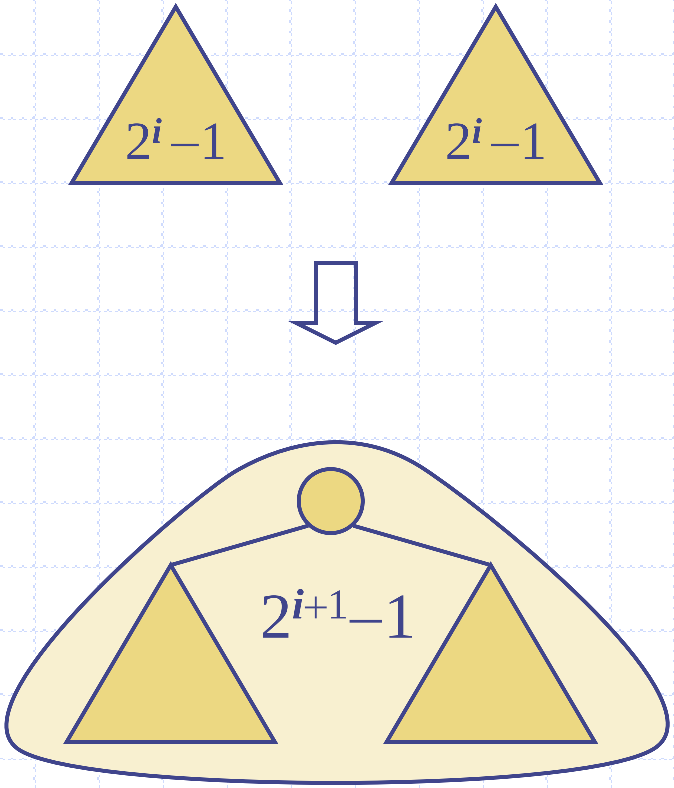
Fusione di 2 Heaps

- Dati due heaps e una chiave ***k***
- Creiamo un nuovo heap con il nodo radice che memorizza ***k*** e con due heap come sottoalberi
- Operiamo downheap per riportare la proprieta' di ordine di heap

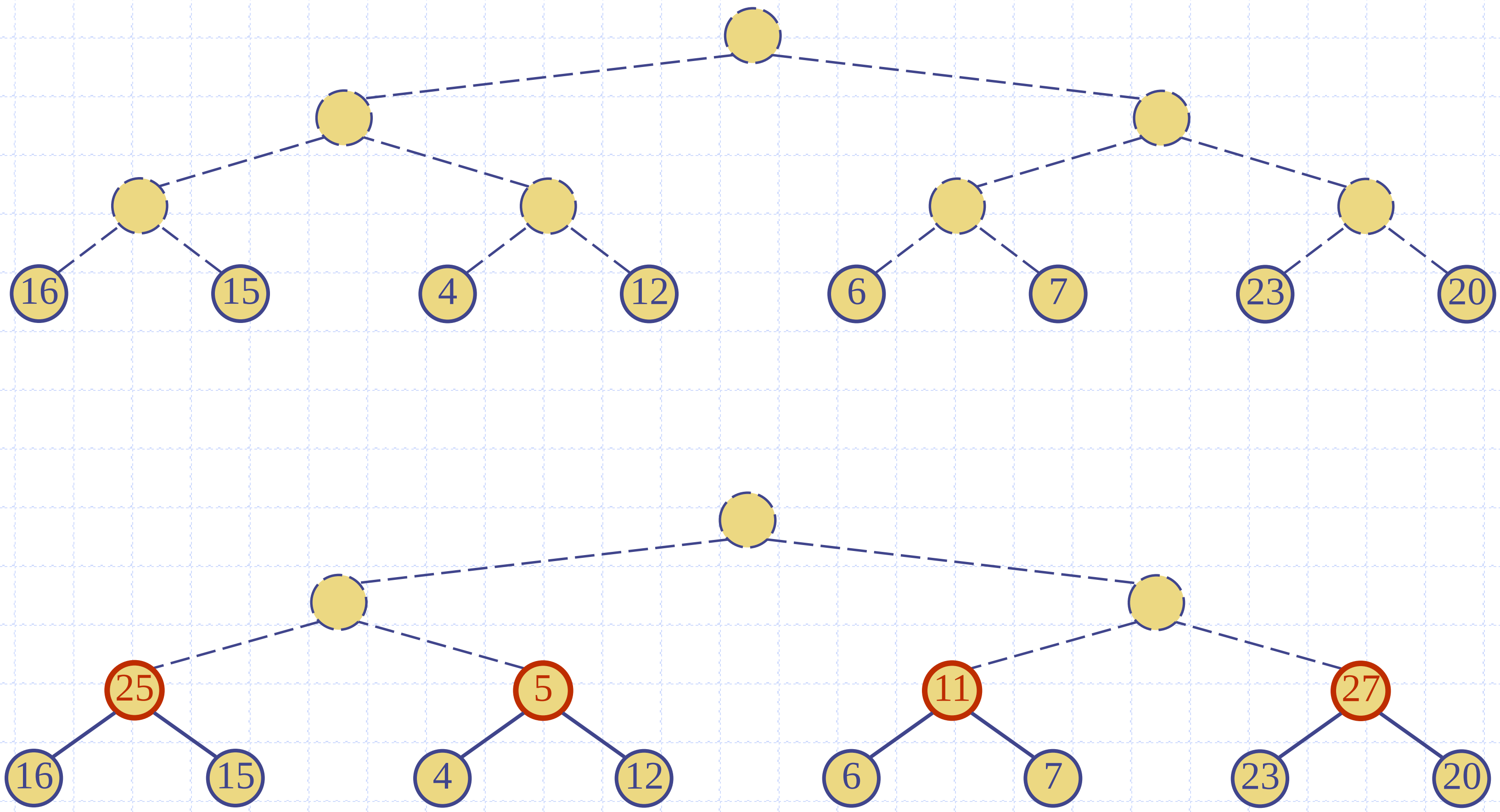


Costruzione bottom-up di un Heap

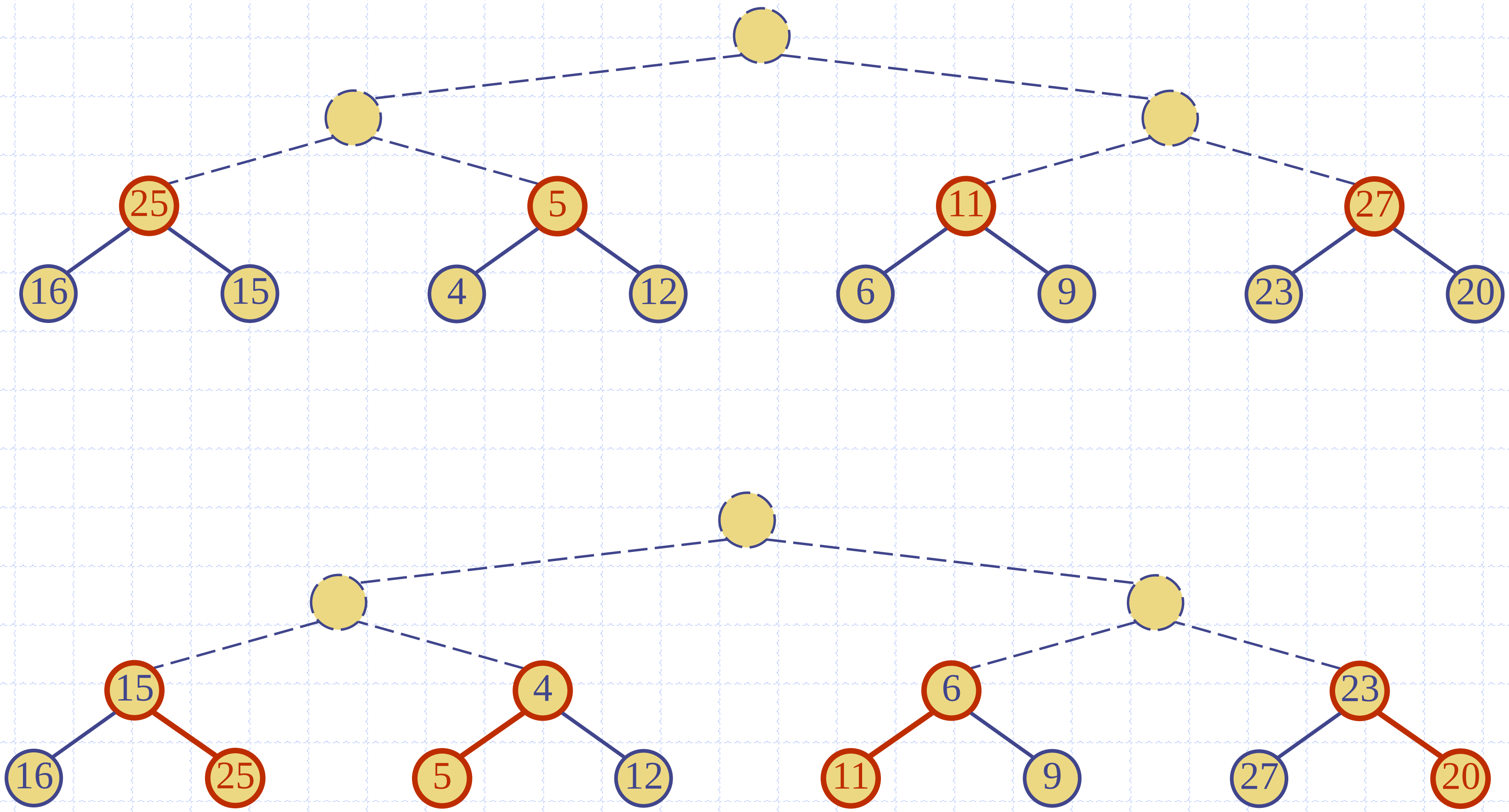
- Possiamo costruire un heap che memorizza n chiavi date usando una costruzione bottom-up con $\log n$ fasi
- Nella fase i, coppie di heap con $2^i - 1$ chiavi sono fuse in heap con $2^{i+1} - 1$ chiavi



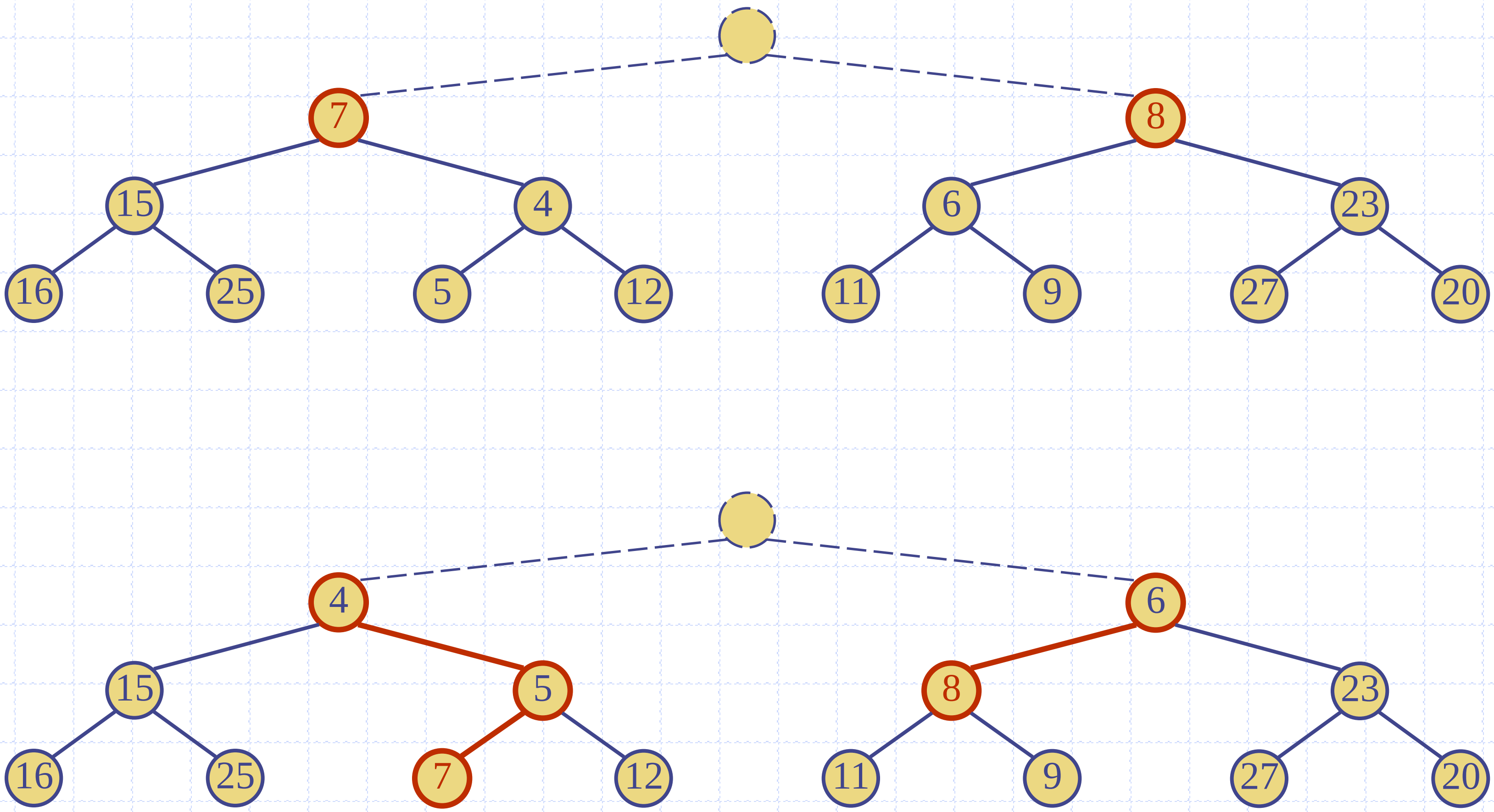
Esempio



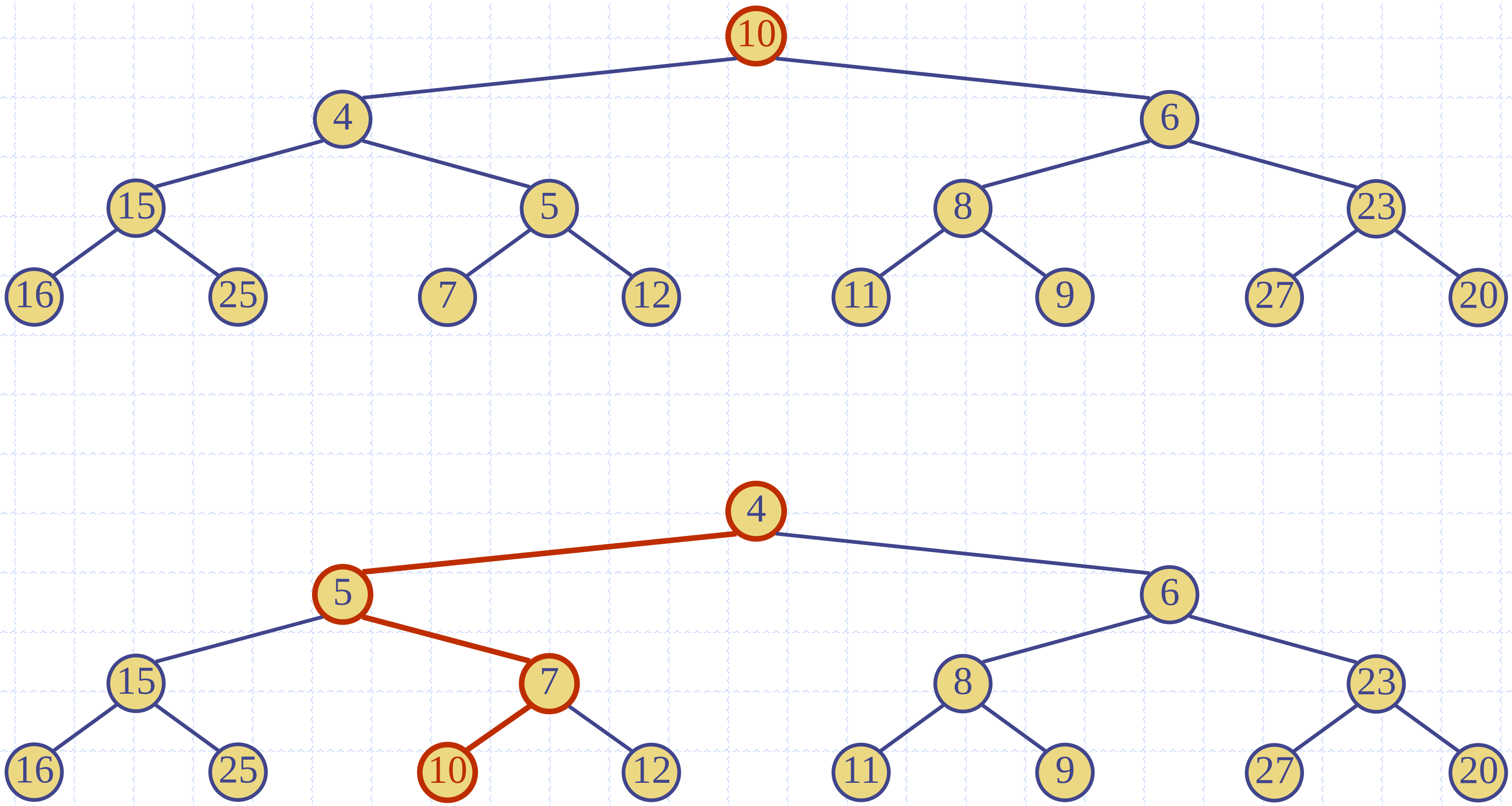
Esempio(cont.)



Esempio(cont.)



Esempio(fine)



Analisi

- Visualizziamo il tempo nel caso peggiore richiesto da un downheap con un cammino che procede prima a destra e poi ripetutamente a sinistra fino al fondo dell'heap (questo cammino puo' essere diverso dal vero cammino di downheap)
- Poiche' ogni nodo e' attraversato da al piu' 2 cammini, il numero totale di nodi nei cammini e' $O(n)$
- Quindi, la costruzione bottom-up di un heap e' eseguita con tempo $O(n)$
- La costruzione bottom-up di un heap e' piu' veloce di n inserimenti successivi e velocizza la prima parte di heap-sort