Code a priorità

Progettazione di Algoritmi 2018-19

Matricole congrue a 1

Docente: Annalisa De Bonis

Coda a priorità

- Una coda a priorità è un collezione di elementi a ciascuno dei quali è assegnata una chiave
  - L'elemento v ha chiave key(v)
  - Le chiavi determinano la priorità degli elementi, ovvero l'ordine in cui vengono rimossi dalla coda
- Applicazioni
  - Viaggiatori in standby
  - Processi in attesa di usare una risorsa condivisa
  - Struttura ausiliaria di algoritmi

Priority scheduling

- Ad ogni processo è assegnata una priorità
- I processi in attesa di essere eseguiti sono inseriti in una coda priorità
- Viene estratto dalla coda ed eseguito il processo con priorità più grande
- Problema
  - Starvation: i processi con priorità più bassa non vengono mai eseguti
- Soluzione
  - Aging: le priorità dei processi in coda vengono gradualmente aumentate

Esempi dell'uso della coda a priorità come struttura dati ausiliaria in un algoritmo

- Prim e Dijkstra
  - Algoritmi di ottimizzazione *greedy*
    - Scelta greedy basata sul valore delle chiavi assegnate ai vertici
    - I vertici del grafo vengono inseriti in una coda a priorità
      - Ad ogni passo viene estratto dalla coda il vertice con priorità più alta (chiave più piccola)

Relazione di ordine totale ($\leq$)

- Proprietà
  - Riflessiva:

    $x \leq x$

  - Antisimmetrica:

    $x \leq y$ e $y \leq x \Rightarrow x = y$

  - Transitiva:

    $x \leq y$ e $y \leq z \Rightarrow x \leq z$

- Su qualsiasi insieme finito sono sempre definiti sia il max che il min

Operazioni principali

- Insert (P, o)
  inserisce l'entrata o nella coda a priorità P restituendola in output;
- ExtractMin(P)
  rimuove e restituisce l'entrata con chiave (key) più piccola;
- ChangeKey(P,x,k)

  sostituisce la chiave di x con k

Operazioni aggiuntive

- IsEmpty(P)

  restituisce true se e solo se P è vuota
- FindMin(P)

  restituisce l'entrata con chiave più piccola (senza cancellarla)
- Delete(P,x)

  rimuove e restituisce l'elemento x;

Implementazione con una lista non ordinata

- Memorizza gli elementi della coda in una lista in un ordine qualsiasi
- Complessità:
  - Insert richiede tempo $O(1)$ in quanto possiamo semplicemente inserire l'elemento alla fine della lista
  - ExtractMin, FindMin richiedono tempo $O(n)$ in quanto bisogna scorrere tutta la lista per determinare l'elemento con chiave minima

Implementazione con una lista ordinata

- Memorizza gli elementi della coda in una lista per valore di chiave in ordine non decrescente
- Complessità:
  - Insert richiede tempo $O(n)$ in quanto occorre trovare il posto dove inserire l'oggetto in modo da mantenere l'ordine non decrescente delle chiavi
  - ExtractMin, FindMin richiedono tempo $O(1)$ in quanto la chiave minima è all'inizio della lista

Ordinamento con PriorityQueue

- Si può usare una coda a priorità per ordinare un insieme di elementi
  1. inserisci un elemento alla volta con Insert
  2. rimuovi gli elementi uno alla volta con ExtractMin
- L'analisi della complessità di tempo di questo algoritmo dipende da come è implementata la coda a priorità

```
Algorithm PQ-Sort(S)
    Input sequenza S
    Output sequenza S ordinata per
    valori crescenti
    P ← coda a priorità vuota
    while (S non è vuota)
        e ← primo elemento di S
        key(e)← e
        P.Insert(e)
        Cancella primo elemento di S
    while !(isEmpty(P))
        e ← ExtractMin(P)
        aggiungi e alla fine di S
```

Selection-Sort

- Selection-sort è una variante di PQ-sort dove la coda a priorità è implementata con una lista non ordinata
- Tempo di esecuzione di Selection-sort:
  1. Inserire gli elementi nella coda richiede n chiamate a Insert, e quindi tempo $O(n)$
  2. Rimuovere gli elementi dalla coda in ordine richiede n chiamate a ExtractMin, e quindi tempo

$$O(1 + 2 + \ldots + n\text{-}1) = O(n\,(n-1)\,/\,2) = O(n^2)$$

Selection-sort richiede tempo $O(n^2)$ e spazio aggiuntivo $O(n)$

Insertion-Sort

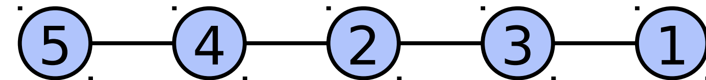
- Insertion-sort è una variante di PQ-sort dove la coda a priorità è implementata con una lista ordinata
- Tempo di esecuzione di Insertion-sort:
  1. Inserire gli elementi nella coda in ordine richiede n chiamate a Insert, e quindi tempo $O(1 + 2 + \ldots + n-1) = O(n(n-1)/2) = O(n^2)$
  2. Rimuovere gli elementi dalla coda in ordine richiede n chiamate a ExtractMin, e quindi tempo $O(n)$

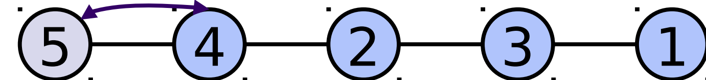
Insertion-sort richiede tempo $O(n^2)$ e spazio aggiuntivo $O(n)$

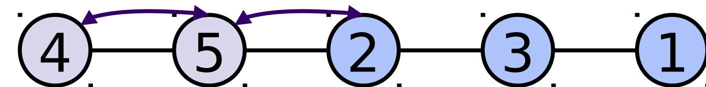
Insertion-sort sul posto

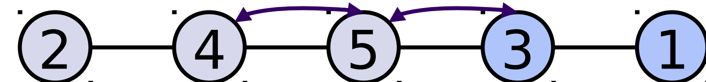
- Invece di usare una struttura di appoggio, sia Insertion-sort che Selection-sort si possono implementare sul posto

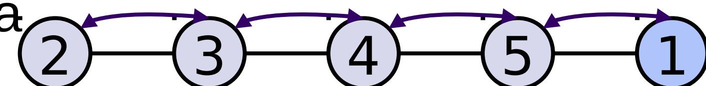
- Usiamo swapElements per spostare gli elementi invece di modificare la struttura della lista
- Una porzione della sequenza in input viene usata nello stesso modo in cui è usata la coda a priorità in PQSort.
- **Insertion-sort sul posto**
  - Manteniamo ordinata la prima parte della sequenza
  - Ad ogni passo prendiamo un elemento della seconda parte della sequenza e lo **inseriamo** nella parte già ordinata della sequenza

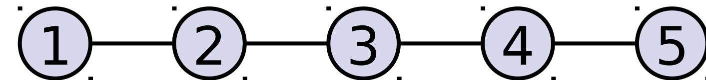






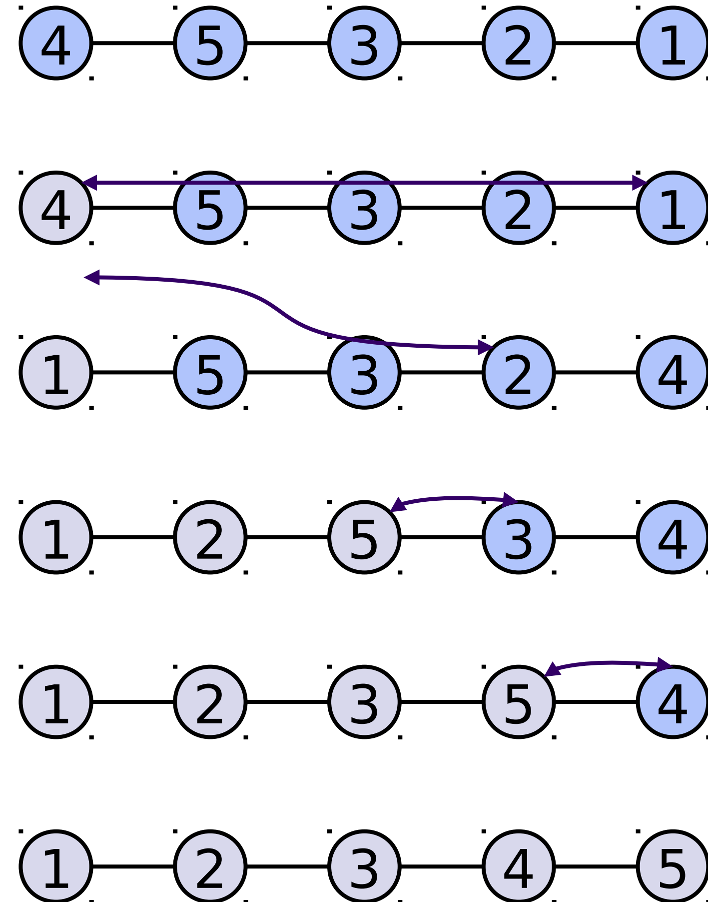






Selection-sort sul posto

- **Selection-sort sul posto**
  - Manteniamo ordinata la prima parte della sequenza
  - Al passo i-esimo **selezioniamo** l'elemento minimo tra quelli non ancora ordinati e lo scambiamo con l'i-esimo elemento della sequenza



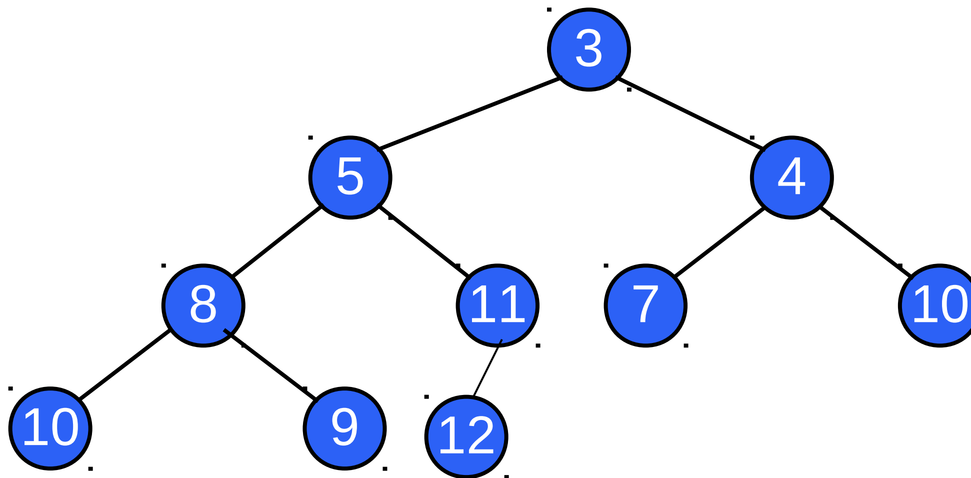
Iimplementazione della coda a priorità mediante un heap

- Un heap è un albero binario ai cui elementi sono assegnate delle chiavi e che soddisfa le seguenti proprietà:

  - Heap-Order: per ogni nodo $v \neq$ radice
    - Chiave dell'elemento in $v \geq$ chiave dell'elemento nel padre(v))

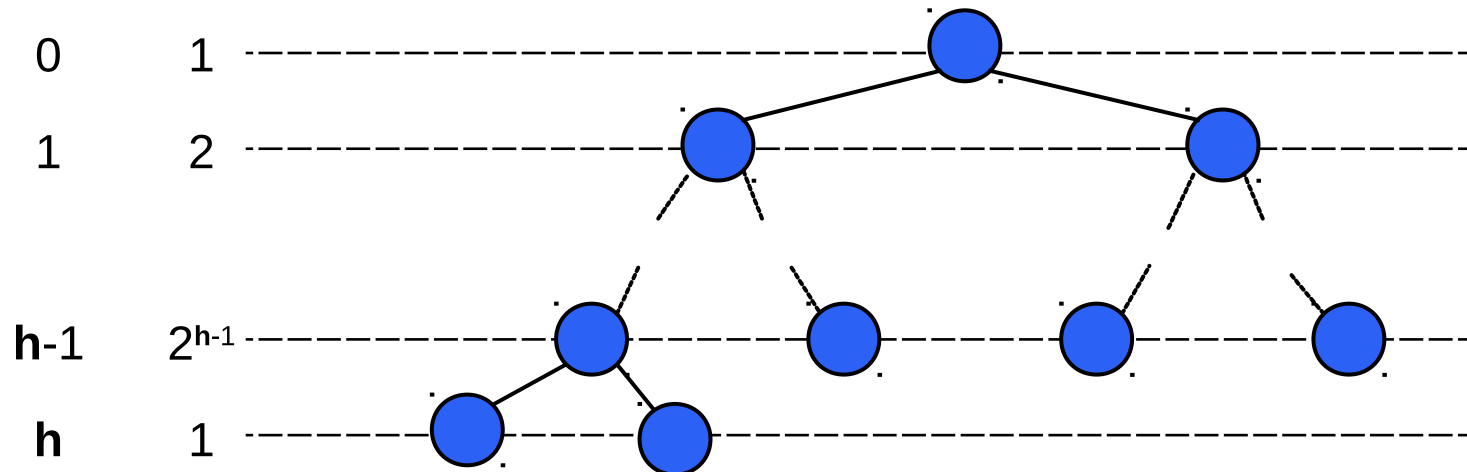
  - Albero binario completo: dato un heap di altezza **h**
    - per **i** = 0, … , **h**-1, ci sono 2^i nodi di profondità i (tutti i livelli, salvo al più l'ultimo, sono pieni)
    - L'ultimo livello è riempito da sinistra verso destra

Esempio



Altezza di un heap

- Un heap che memorizza n chiavi ha altezza $\lfloor \log n \rfloor$
- Dimostrazione: Sia h l'altezza dell'albero
  - Ci sono 2^i chiavi a profondità i = 0, … , h - 1 ed almeno una chiave a profondità h → $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h$
  - quindi $h \leq \log n$



Altezza di un heap

- D'altra parte sappiamo che il numero max di nodi di un albero binario di altezza h è
  - $n \leq 1 + 2 + 4 + \ldots + 2^h = 2^{h+1}-1$

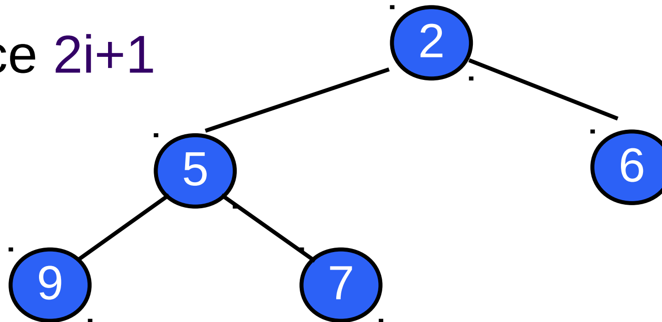
- ➔ $2^h \leq n \leq 2^{h+1}-1$ ➔ $\log(n+1)-1 \leq h \leq \log n$

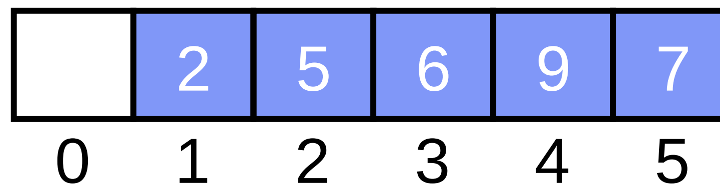
➔ $\log(n)-1 < h \leq \log n$

$h = \lfloor \log n \rfloor$

Implementazione con array

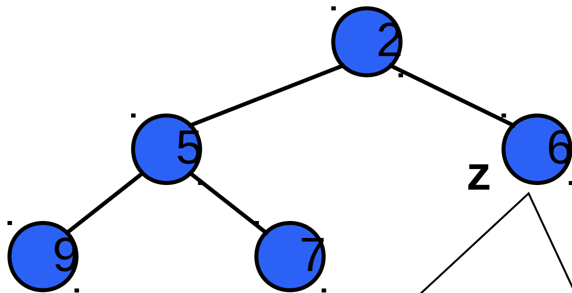
- Per un albero con n nodi si usa un array H di lunghezza n+1
  - entrata di indice 0 vuota
- Per un nodo di indice i
  - Il figlio sinistro ha indice 2i
  - Il figlio destro ha indice 2i+1

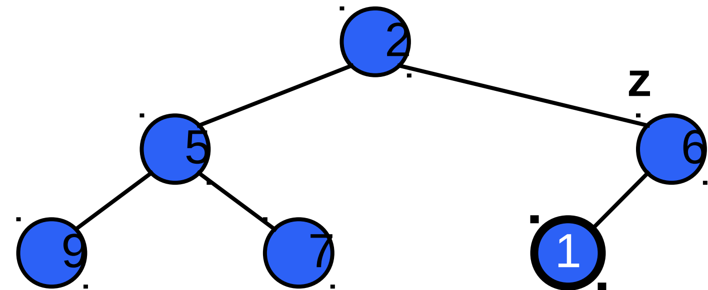




Insert

- Insert effettua l'inserimento di un elemento nell'heap in questo modo:
  - Aggiunge all'albero una nuova foglia e inserisce il nuovo elemento in questa foglia (in altre parole inserisce il nuovo elemento nella prima locazione libera dell'array)
  - Ristabilisce l'heap-order

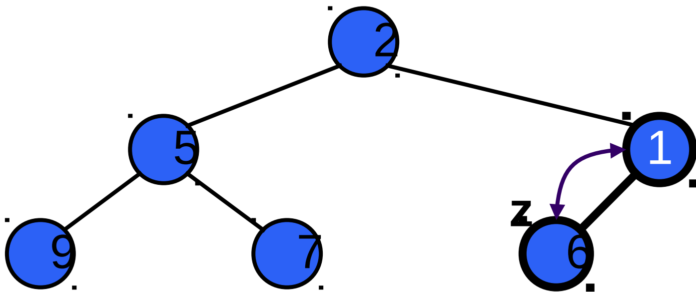


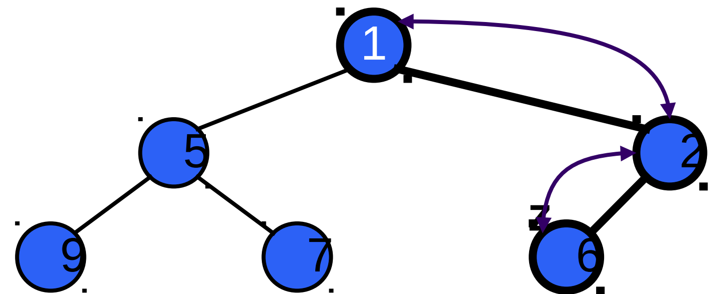


Ripristino dell’heap-order

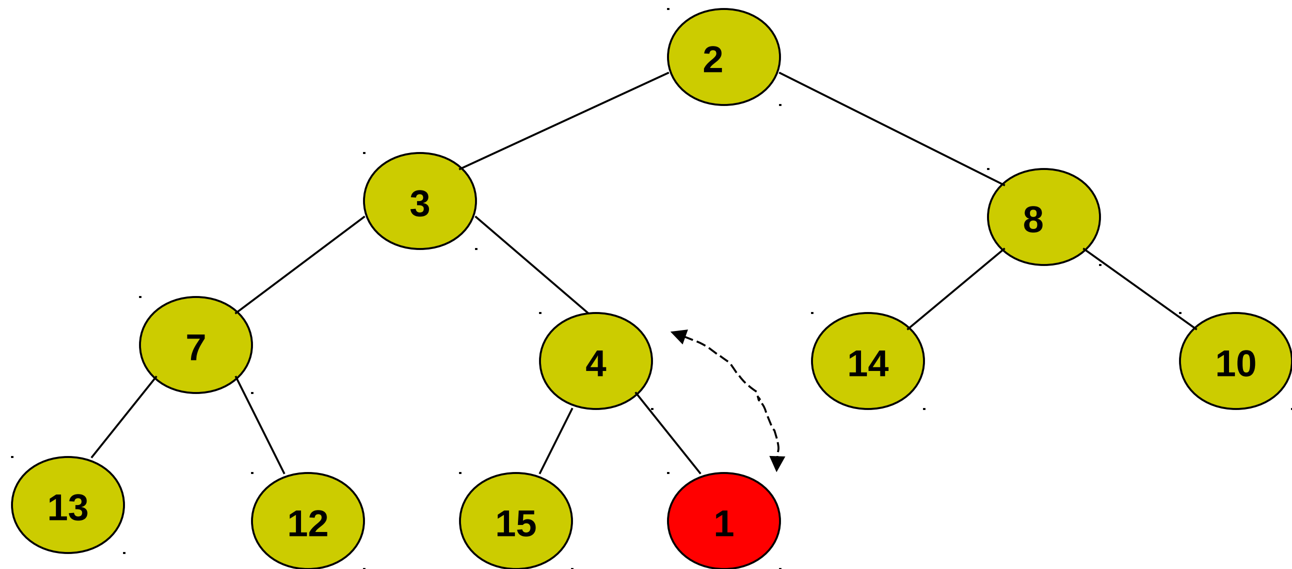
- L’algoritmo Heapify-up ripristina l’heap-order scambiando il nuovo elemento z con i suoi antenati fino a che z raggiunge la radice o si incontra un antenato con chiave minore di **key(z)**

- Siccome un heap ha altezza **O**(log **n**), l’algoritmo Heapify-up ha tempo di esecuzione in **O**(log **n**) time

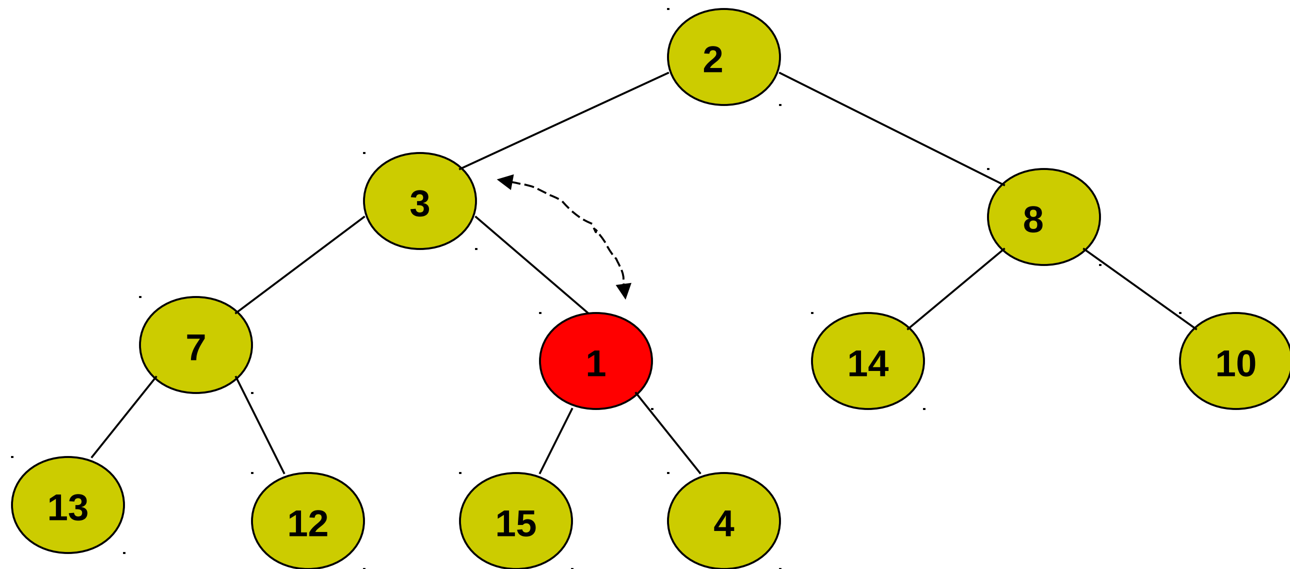




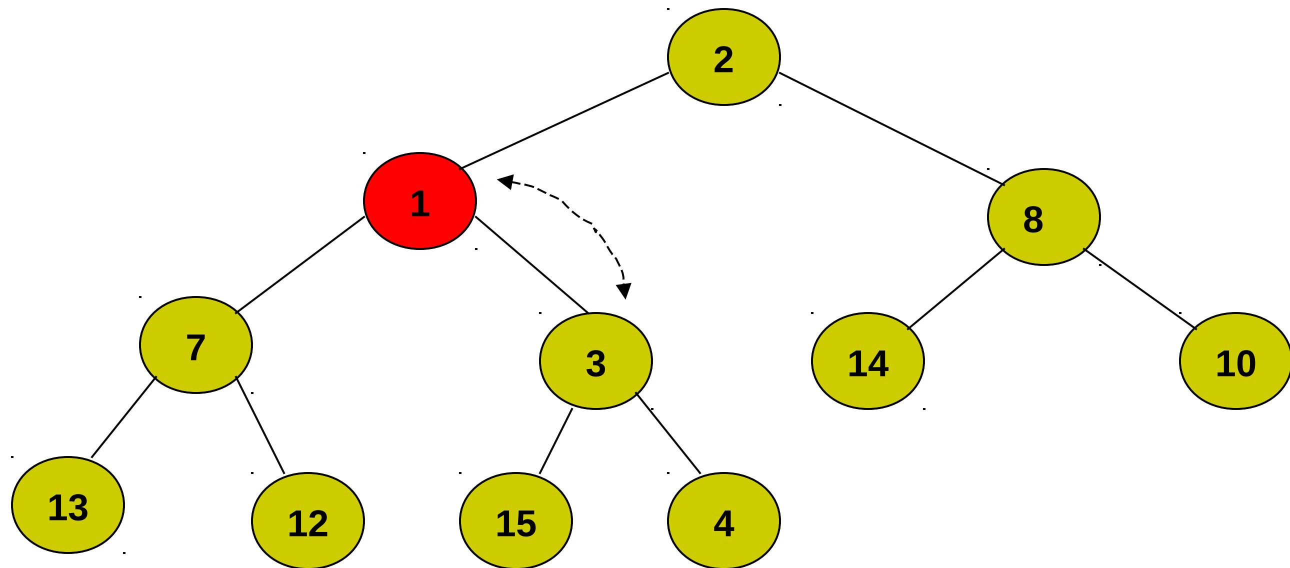
Inserimento della chiave 1



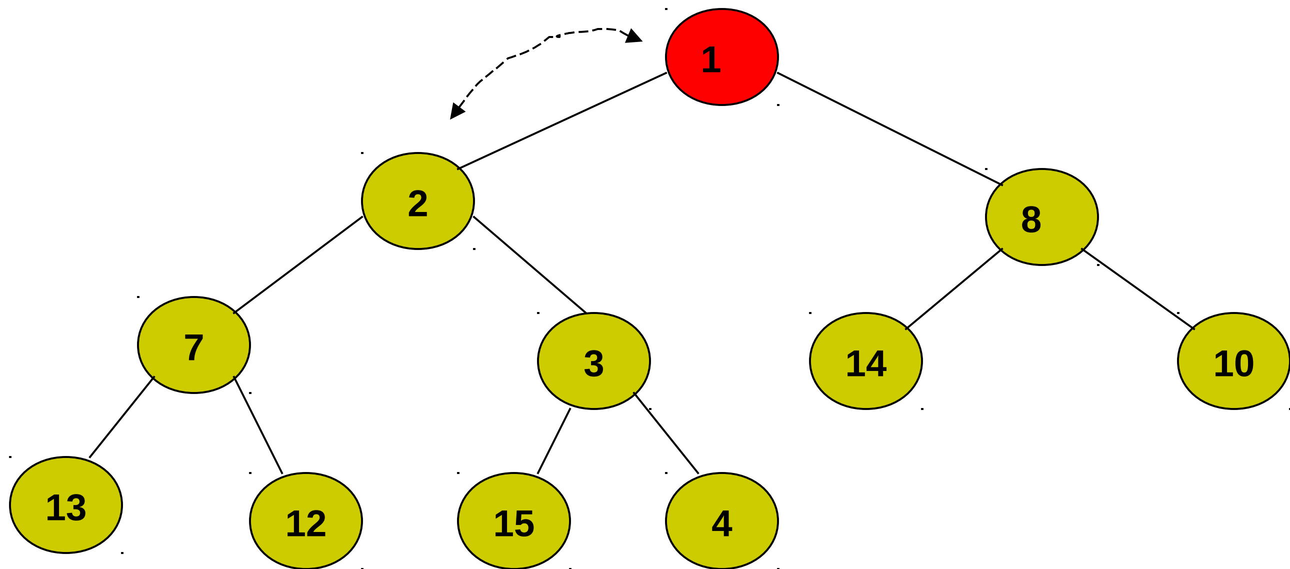
Inserimento della chiave 1



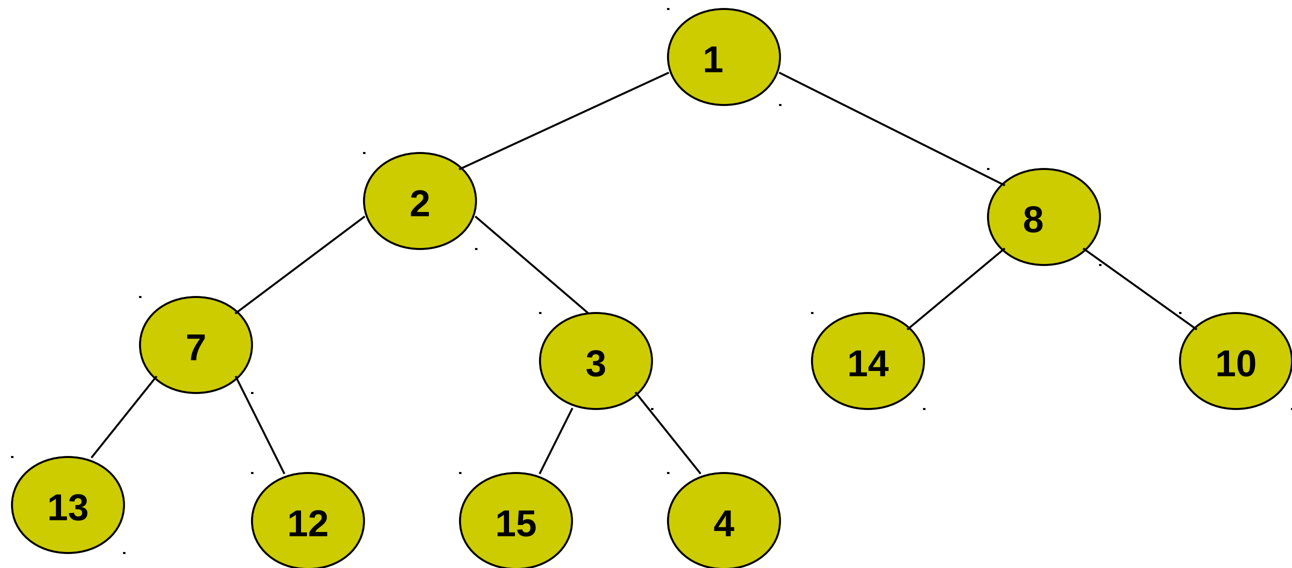
Inserimento della chiave 1



Inserimento della chiave 1



Inserimento della chiave 1

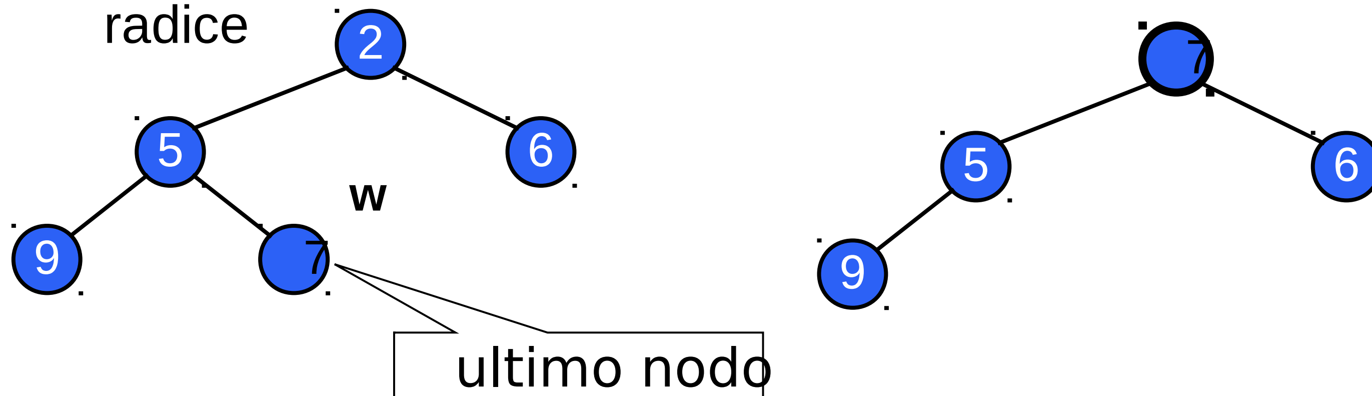


Heapify-up

```
Heapify-up(H,i):
 If i > 1 then
   let j =  ⌊i/2⌋
   If key[H[i]] < key[H[j]] then
      swap the array entries H[i] and H[j]
      Heapify-up(H,j)
   Endif
 Endif
```

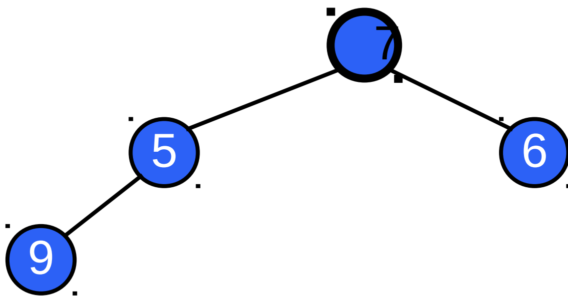
ExtractMin

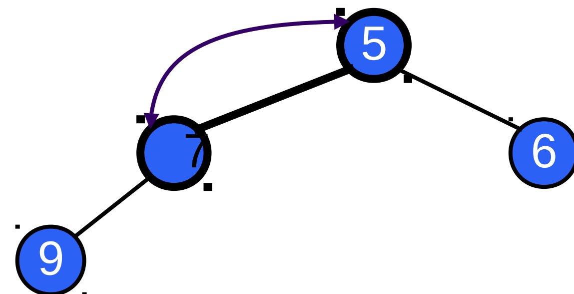
- ExtractMin è implementato rimuovendo l'entrata nella radice dell'heap
- L'algoritmo di rimozione consiste di 3 passi:
  - Sostituisci l'entrata della radice con l'entrata dell'ultimo nodo w
  - Rimuovi w
  - Ripristina con Heapify-down l'heap-order che potrebbe essere stato violato dalla sostituzione dell'entrata della radice



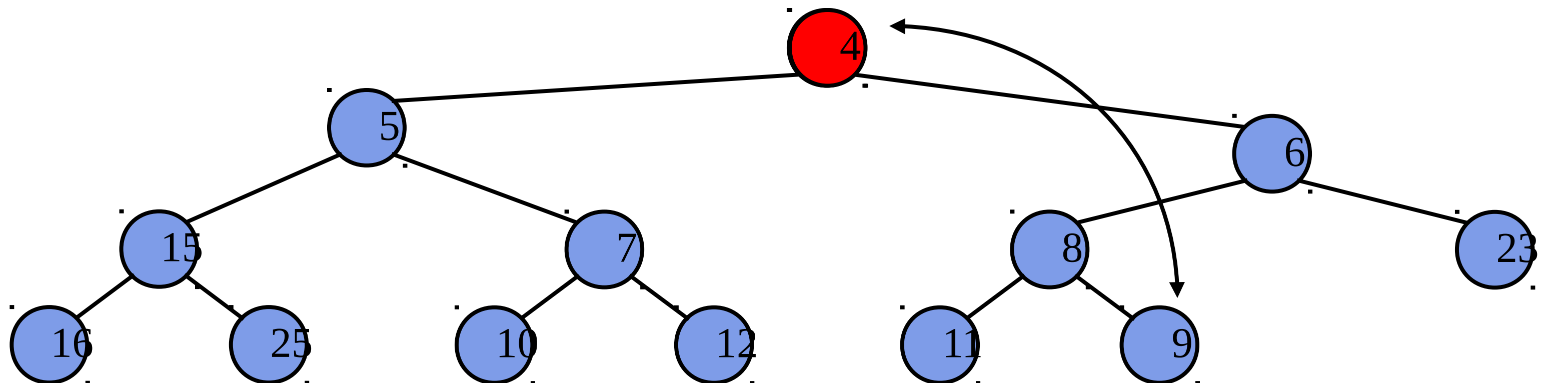
Heapify-down

- L’algoritmo Heapify-down ripristina l’heap-order scambiando ad ogni passo l’entrata v con l’entrata del figlio che ha chiave più piccola
- L’algoritmo Heapify-down termina quando v raggiunge un nodo **z** tale che **z** è una foglia o le chiavi dei figli di **z** sono maggiori o uguali di **k**
- Siccome l’altezza dell’heap è **O**(log n), Heapify-down ha tempo di esecuzione **O**(log n)

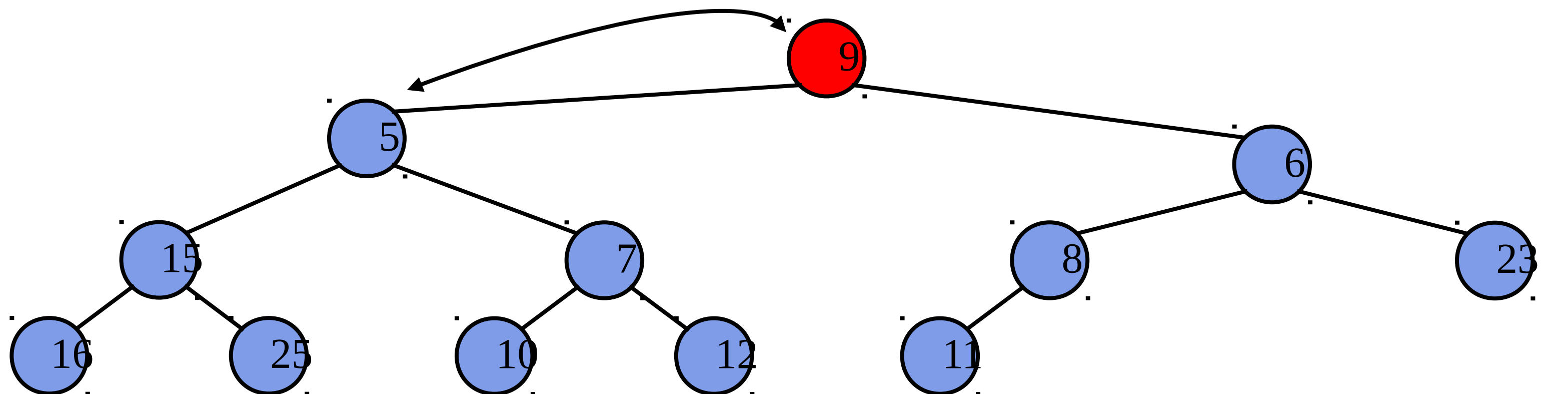




Cancellazione del minimo



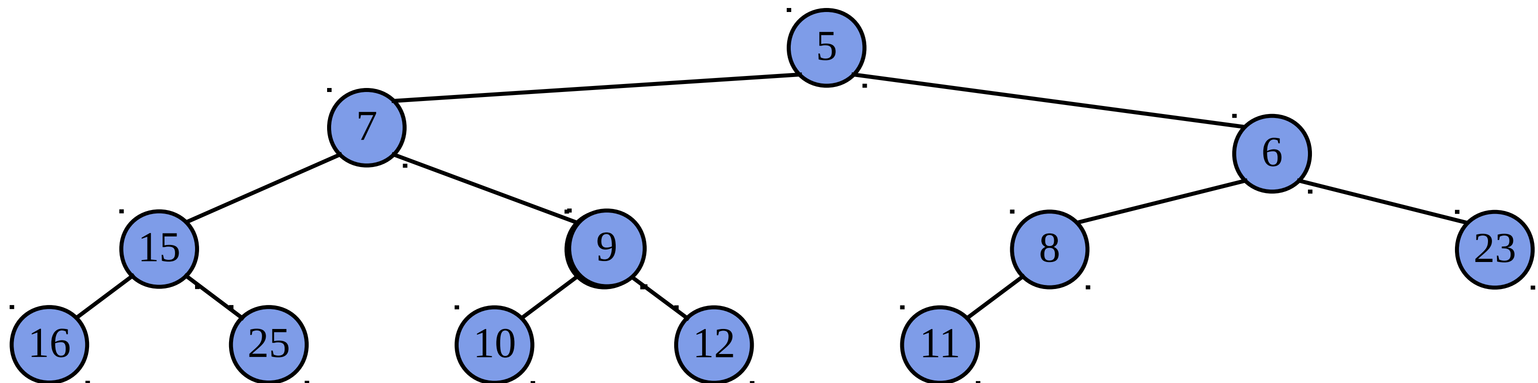
Cancellazione del minimo



Cancellazione del minimo

5
9
6
15
7
8
23
16
25
10
12
11

Cancellazione del minimo



Heapify-down

```
Heapify-down(H,i):  //ripristina heap-order a partire da i
 Let n = length (H)-1  //indici celle piene  da 1 a  length (H)-1
 If 2i > n then       //non c'è il figlio sinistro (e quindi neanche il destro)
   Terminate with H unchanged
 Else if 2i < n then      //ci sono entrambi i figli
   Let left = 2i , and right = 2i + 1
   If key[ H [left]] < key[ H [right]] then j=left else j=right
 Else if 2i = n then    //non c'è figlio destro
    Let j = 2i
 Endif
If key[ H [ j ]] < key[ H [ i ]] then
swap the array entries H [ i ] and H [ j ]
Heapify-down( H , j )
Endif
```

Motivazioni per le operazioni Delete e ChangeKey

- Alcuni algoritmi richiedono di cancellare un'entrata qualsiasi o di aggiornare l'elemento o la chiave di un'entrata qualsiasi
- Esempio:
  - Prim e Dijkstra effettuano O(E) operazioni di aggiornamento di key

Implementazione di Delete e ChangeKey

- Delete e ChangeKey operano su un elemento arbitrario della coda a priorità di cui non si conosce la posizione nell'array
- Per questo motivo manteniamo un array aggiuntivo Position che associa ad ogni elemento v dello Heap l'indice della locazione dell'array H in cui si trova v.

Implementazione di Delete e ChangeKey

- Delete(P,v):
  - Legge in Position[v] l'indice i in cui si trova v
  - Scambia l'elemento presente nell'ultima foglia di H (cioè H(n)) con v
  - Invoca Heapify-up e Heapify-down con argomento i

- ChangeKey(P,v,k):
  - Legge in Position[v] l'indice i in cui si trova v
  - Sostituisce la chiave di v con k
  - Invoca Heapify-up e Heapify-down con argomento i

Ordinamento mediante Heap

- Se in PQ-Sort si usa una coda a priorità implementata con un heap, il tempo di esecuzione dell’algoritmo è O(n log n)