Algoritmi greedy

Progettazione di Algoritmi 2017-18

Matricole congrue a 1

Docente: Annalisa De Bonis

Scelta greedy

Un algoritmo greedy è un algoritmo che effettua ad ogni passo la scelta che in quel momento sembra la migliore (localmente ottima) nella speranza di ottenere una soluzione globalmente ottima.

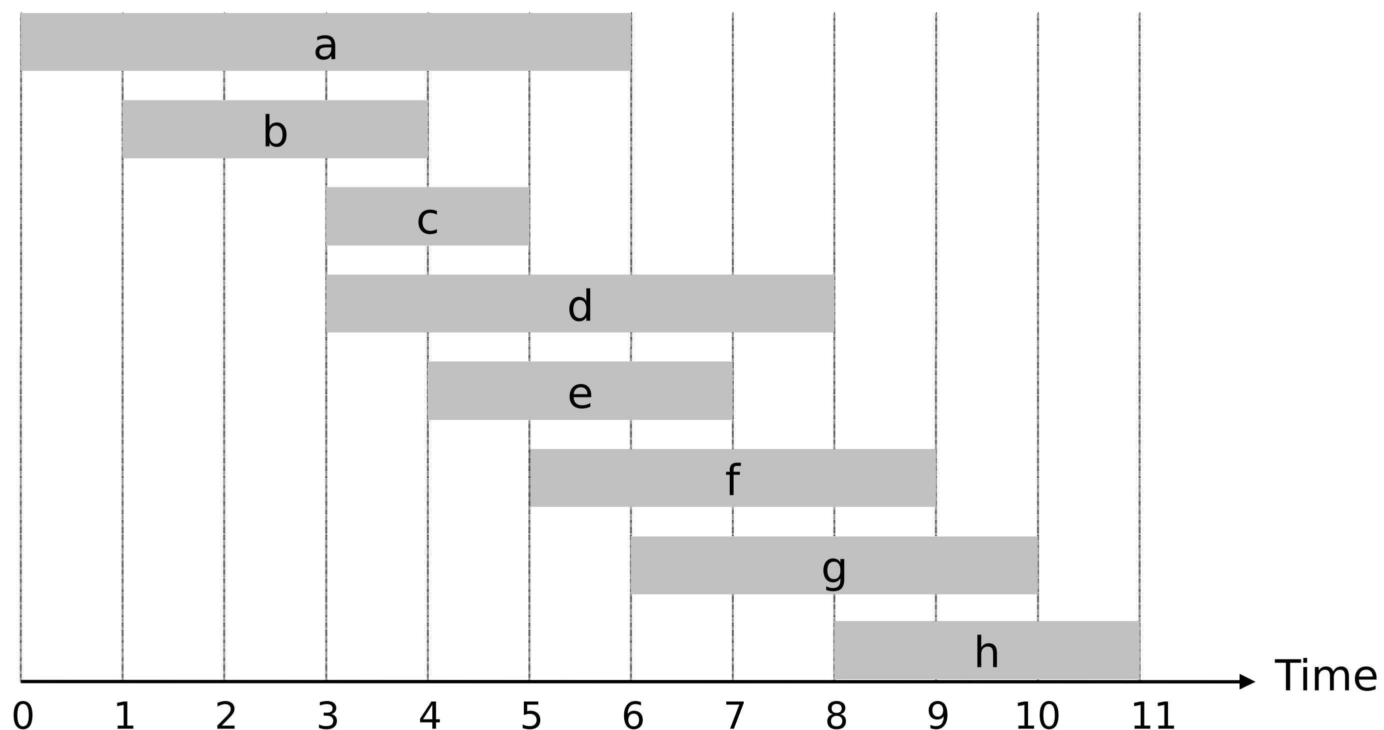
Domanda. Questo approccio porta sempre ad una soluzione ottima?

Risposta. Non sempre ma per molti problemi sì.

Interval Scheduling

Interval scheduling.

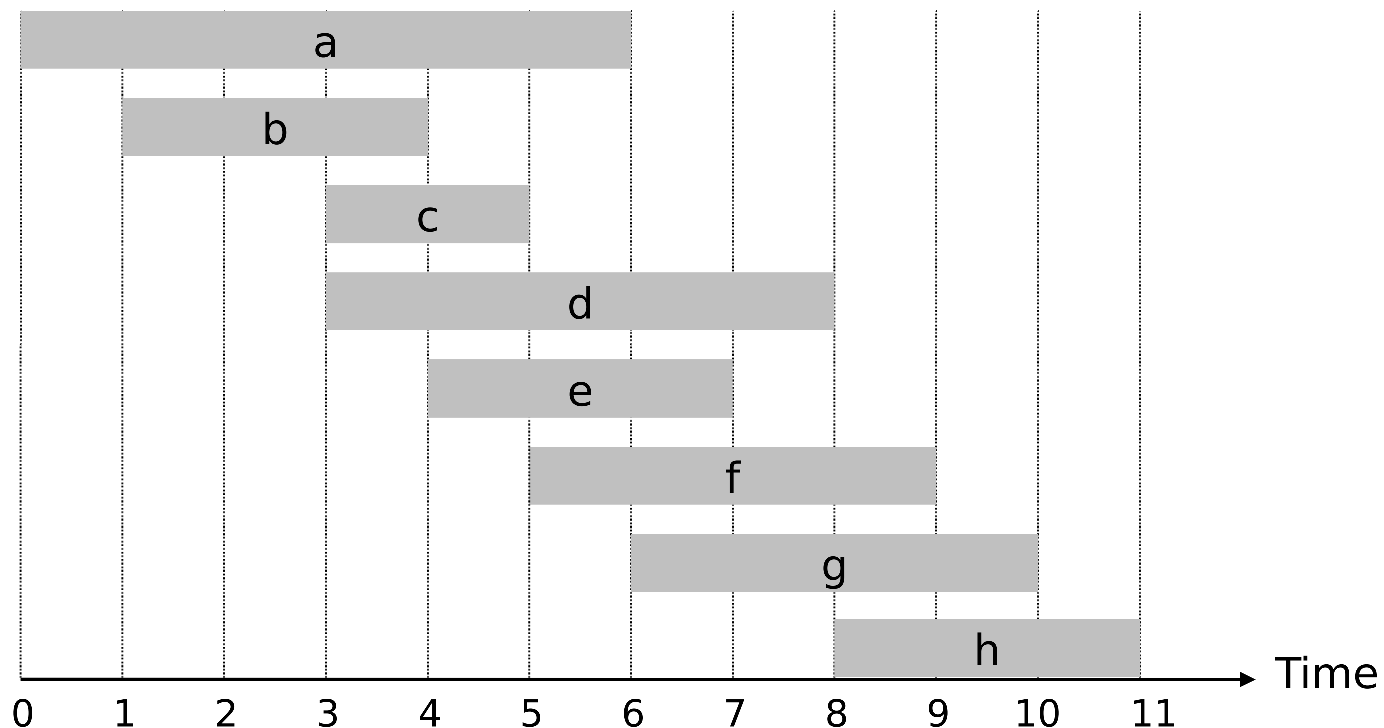
- Input: un insieme di attività ognuna delle quali inizia ad un certo instante s_j e finisce ad un certo istante f_j. Può essere eseguita al più un'attivita alla volta.
- Obiettivo: fare in modo che vengano svolte quante più attività è possibile.



Interval Scheduling

Interval scheduling.

- Il job j comincia nell'instante s_j e finisce all'istante f_j.
- Due job sono compatibili se non si sovrappongono
- Obiettivo: trovare un sottoinsieme di cardinalità massima di job a due a due compatibili.



Interval Scheduling

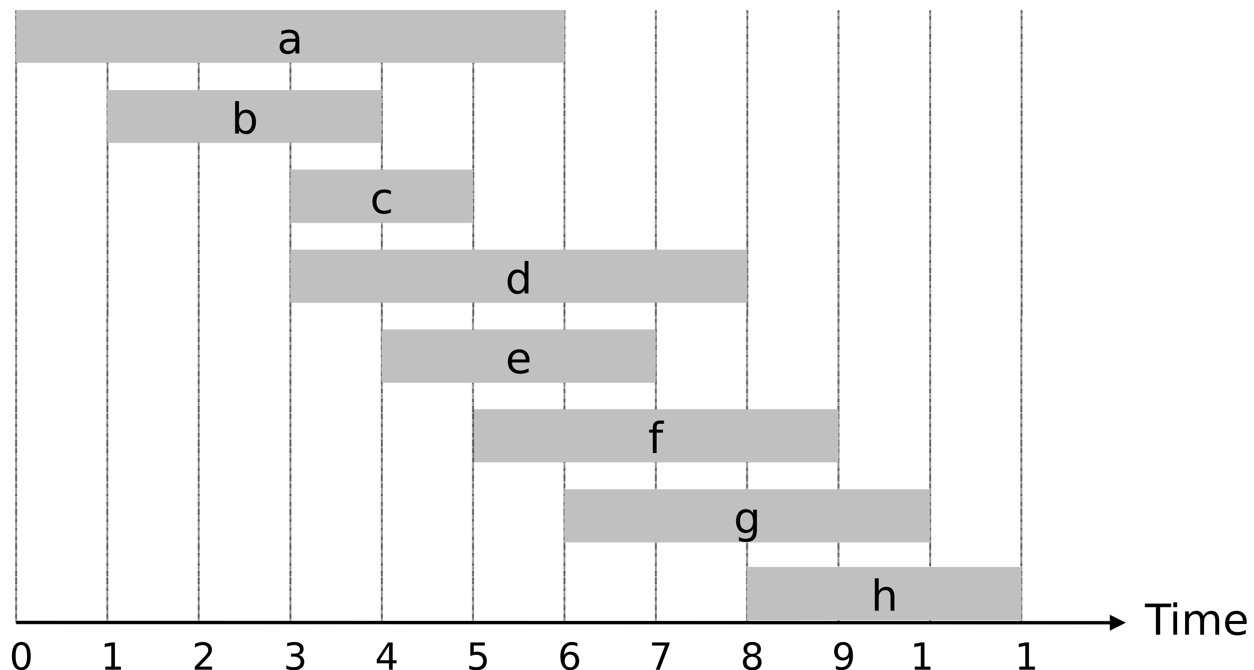
Se all'inizio scegliamo a poi possiamo scegliere o g o h.

- Dopo aver scelto a e g oppure a e h, non possiamo scegliere nessun altro job. Totale = 2

Se all'inizio scegliamo b poi possiamo scegliere uno tra e, f, g e h.

- Se dopo b scegliamo e poi possiamo scegliere anche h. Totale = 3
- Se dopo b scegliamo f poi possiamo non possiamo scegliere nessun altro job. Totale = 2

Se all'inizio scegliamo uno tra f, g e h, non possiamo selezionare nessun altro job. Totale = 1



Interval Scheduling: Algoritmi Greedy

Schema greedy. Considera i job in un certo ordine. Ad ogni passo viene esaminato il prossimo job nell'ordinamento e se il job è compatibile con quelli scelti nei passi precedenti allora il job viene inserito nella soluzione.

L'ordinamento dipende dal criterio che si intende ottimizzare localmente.

Diverse strategie basate su diversi tipi di ordinamento :

- [Earliest start time] Considera i job in ordine crescente rispetto ai tempi di inizio s_j. Scelta greedy consiste nel provare a prendere ad ogni passo il job che inizia prima tra quelli non ancora esaminati.
- [Earliest finish time] Considera i job in ordine crescente rispetto ai tempi di fine f_j. Scelta greedy consiste nel provare a prendere ad ogni passo il job che finisce prima tra quelli non ancora esaminati.
- [Shortest interval] Considera i job in ordine crescente rispetto alle lloro durate $f_j - s_j$. Scelta greedy consiste nel provare a prendere ad ogni passo il job che dura meno tra quelli non ancora esaminati.
- [Fewest conflicts] Per ogni job, conta il numero c_j di job che sono in conflitto con lui e ordina in modo crescente rispetto al numero di conflitti. Scelta greedy consiste nel provare a prendere ad ogni passo il job che ha meno conflitti tra quelli non ancora esaminati.

Interval Scheduling: Algoritmi Greedy

La strategia "Earliest Start Time" sembra la scelta più ovvia ma…

Problemi con la strategia "Earliest Start Time". Può accadere che il job che comincia per primo finisca dopo tutti gli altri o dopo molti altri.

Interval Scheduling: Algoritmi Greedy

Ma se la lunghezza dei job selezionati incide sul numero di job che possono essere selezionati successivamente perché non provare con la strategia “Shortest Interval”?

Questa strategia va bene per l’input della slide precedente ma…

Problemi con la strategia “Shortest Interval”. Può accadere che un job che dura meno di altri si sovrapponga a due job che non si sovrappongono tra di loro. Se questo accade invece di selezionare due job ne selezioniamo uno solo.

Interval Scheduling: Algoritmi Greedy

Visto che il problema sono i conflitti, perché non scegliamo i job che confliggono con il minor numero di job?

Questa strategia va bene per l'input nella slide precedente ma…

Problemi con la strategia "Fewest Conflicts". Può accadere che un job che genera meno conflitti di altri si sovrapponga a due job che non si sovrappongono tra di loro. Se applichiamo questa strategia all'esempio in questa slide, invece di selezionare 4 job ne selezioniamo solo 3.

Interval Scheduling: Algoritmo Greedy Ottimo

L'algoritmo greedy che ottiene la soluzione ottima è quello che usa la stategia "Earliest Finish Time".

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
f ← 0
A ← φ
for j = 1 to n {
   if (sj ≥ f)
      A ← A ∪ {j}
      f ← fj
}
return A
```

Analisi tempo di esecuzione. O(n log n).

- Costo ordinamento O(nlog n)
- Costo for O(n): mantenendo traccia del tempo di fine f dell'ultimo job selezionato, possiamo capire se il job j è compatibile con A verificando che $s_j \geq f$

Interval Scheduling: Ottimalità soluzione greedy

Teorema. L'algoritmo greedy basato sulla strategia "Earliest Finish Time" è ottimo.

Dim.

- Denotiamo con i_1, i_2, … i_k l'insieme di job selezionati dall'algoritmo nell'ordine in cui sono selezionati, cioè in ordine non decrescente rispetto ai tempi di fine.
- Denotiamo con j_1, j_2, … j_m l'insieme di job nella soluzione ottima, disposti in ordine non decrescente rispetto ai tempi di fine.

1. Dimostriamo prima che per ogni indice r, $1 \leqq r \leqq k$, l'esecuzione dei job i_1, i_2, … i_r termina non più tardi di quella dei job j_1, j_2, … j_r
2. Usiamo il punto 1 per dimostrare che non è possibile che k sia minore di m e che quindi la soluzione greedy è ottima.

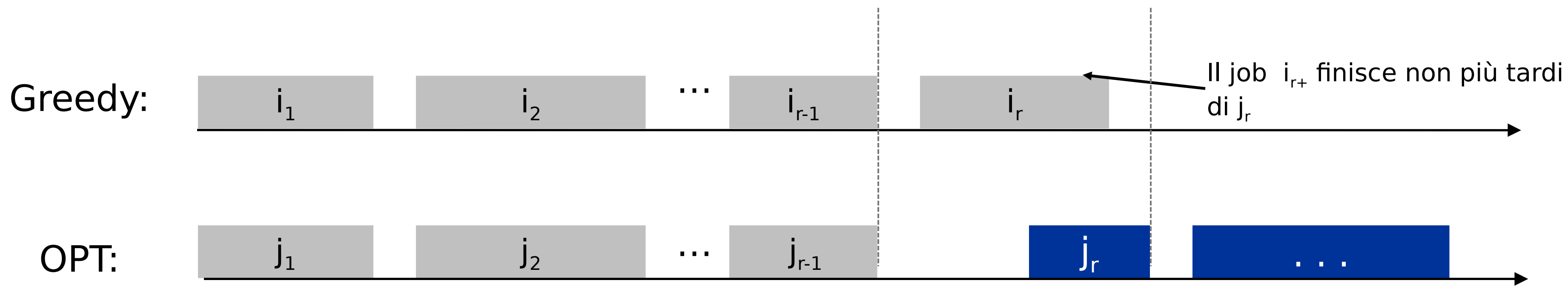
Continua nella prossima slide

Interval Scheduling: Ottimalità soluzione greedy

- Dimostriamo il punto 1.: Dimostriamo per induzione che per ogni indice r, $1 \leqq r \leqq k$, si ha che il tempo di fine di i_r è **non** più grande di quello di j_r.

- **Base.** r=1 : Banalmente vero perchè la prima scelta greedy seleziona la richiesta con il tempo di fine più piccolo.

- **Passo induttivo.** Supponiamo per ipotesi induttiva che per $r-1 \geqq 1$, il tempo di fine di i_{r-1} sia **non** più grande di quello di j_{r-1}. Di conseguenza il job j_r è compatibile con i_{r-1}. Poichè i tempi di fine di $i_1, i_2, \ldots i_{r-2}$ sono non più grandi di quello di i_{r-1} allora j_r è compatibile con $i_1, i_2, \ldots i_{r-2}\ i_{r-1}$. Siccome all'r-esimo passo l'algoritmo greedy sceglie i_r, vuol dire che i_r finisce non più tardi di tutti i job compatibili con $i_1, i_2, \ldots i_{r-1}$ e quindi anche non più tardi di j_r.



Continua nella prossima slide

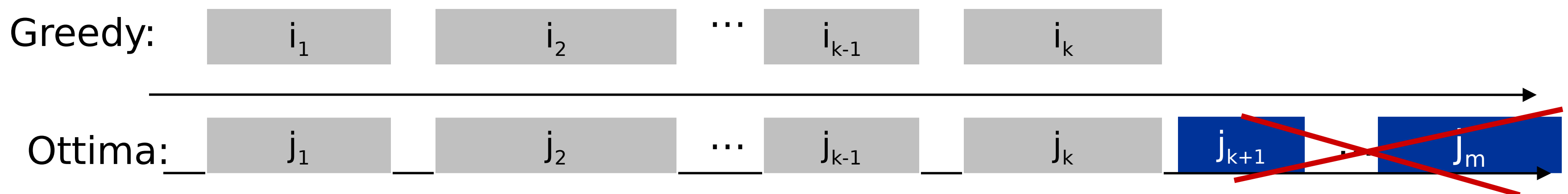
Interval Scheduling: Ottimalità soluzione greedy

- Abbiamo dimostrato che, per ogni indice $r \leqq k$, il tempo di fine di i_r è **non** più grande di quello di j_r .
- Di conseguenza il tempo di fine di i_k è non più grande di quello di j_k. Dal momento che i_1, i_2, ... i_k sono ordinati in base ai tempi di fine allora si ha che i tempi di fine di i_1, i_2, ... i_k sono **non** maggiori di quello di j_k .
- Poiché inoltre j_1, ... j_m sono ordinati in base ai tempi di fine allora, per il punto precedente, l'esecuzione della sequenza di job i_1, i_2, ... i_k termina non più tardi dell'esecuzione della sequenza di job j_1, j_2,... j_k

Continua nella prossima slide

Interval Scheduling: Ottimalità soluzione greedy

- Dimostriamo il punto 2.
- Supponiamo per assurdo che la soluzione greedy non sia ottima e quindi che $k < m$.
- Per il punto 1, l'esecuzione di $i_1, i_2, \ldots i_k$ termina **non** più tardi dell'esecuzione di $j_1, j_2, \ldots j_k$. e si ha quindi che $i_1, i_2, \ldots i_k$ sono compatibili con j_{k+1}
- L'algoritmo greedy, dopo aver inserito $i_1, i_2, \ldots i_k$ nella soluzione, passa ad esaminare i job con tempo di fine maggiore o uguale a quelli di $i_1, i_2, \ldots i_k$. Tra questi job vi è j_{k+1} che è compatibile con $i_1, i_2, \ldots i_k$ per cui è impossibile che l'algoritmo greedy inserisca nella soluzione solo k job. Siamo giunti ad una contraddizione.



Provare l'ottimalità della soluzione greedy

- Come abbiamo provato l'ottimalità della soluzione greedy?
- Abbiamo prima di tutto dimostrato che la soluzione greedy "sta sempre avanti" a quella ottima.
  - Cosa vuol dire "sta sempre avanti"?
  - L'idea alla base della strategia greedy Earliest Finish Time è la seguente: quando si usa la risorsa è bene liberarla il prima possibile perchè ciò massimizza il tempo a disposizione per eseguire le restanti richieste
  - In questa ottica, una soluzione per il problema dell'interval scheduling "sta sempre avanti" ad un'altra se ad ogni passo seleziona una richiesta che termina non più tardi della corrispondente richiesta della soluzione ottima.
- Abbiamo usato il fatto che la soluzione greedy "sta sempre avanti" a quella ottima per provare che la soluzione greedy non può contenere un numero di richieste inferiore a quello della soluzione ottima

Partizionamento di intervalli

In questo caso disponiamo di più risorse identiche tra di loro e vogliamo che vengano svolte tutte le attività in modo tale da usare il minor numero di risorse e tenendo conto del fatto che due attività non possono usufruire della stessa risorsa allo stesso tempo.

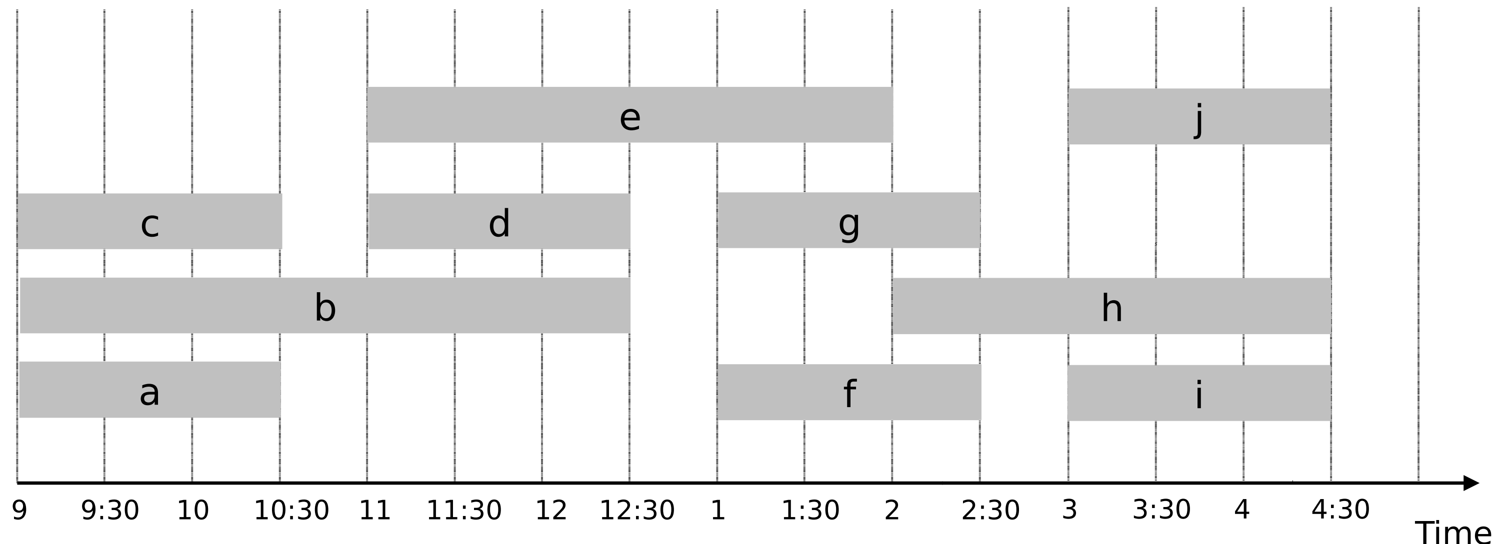
Nell'interval scheduling avevamo un'unica risorsa e volevamo che venissero svolte il massimo numero di attività

Partizionamento di intervalli

Esempio. Attività = lezioni da svolgere; Risorse= aule

- La lezione j comincia ad s_j e finisce ad f_j.
- Obiettivo: trovare il minor numero di aule che permetta di far svolgere tutte le lezioni in modo che non ci siano due lezioni che si svolgono contemporaneamente nella stessa aula.

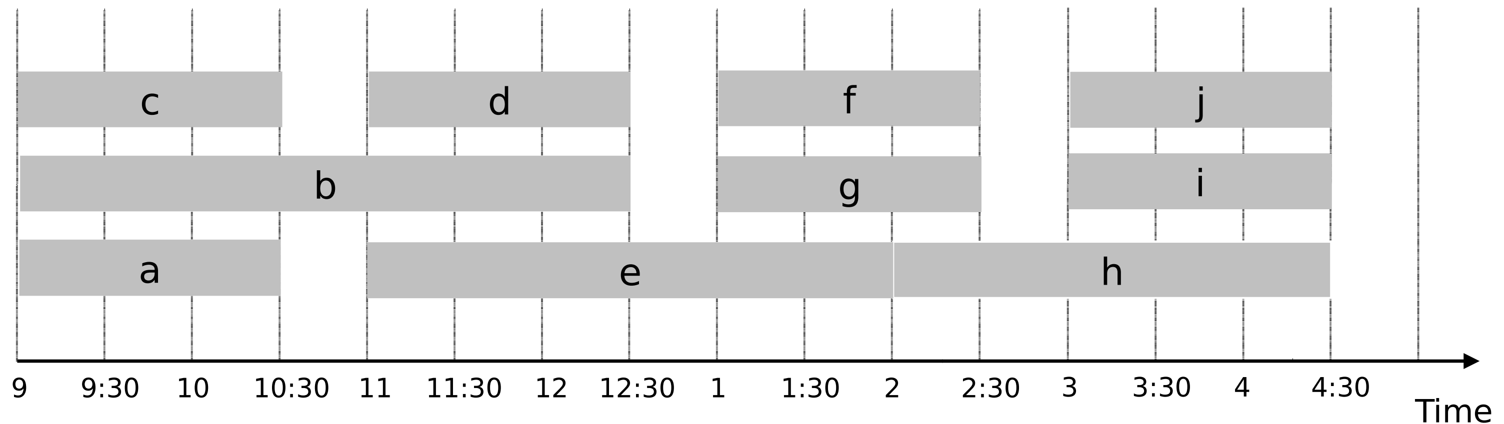
Esempio: Questo schedule usa 4 aule (una per livello) per 10 lezioni: e, j nella prima aula; c,d,g nella seconda aula; b, h nella terza aula; a, f, i nella quarta aula



Partizionamento di intervalli

Esempio. Questo schedule usa solo 3 aule per le stesse attività: {c,d,f,j},{b,g,i},{a,e,h} .

Si noti che la disposizione delle lezioni lungo l'asse delle ascisse è fissato dall'input mentre la disposizione lungo l'asse delle y è fissato dall'algoritmo.

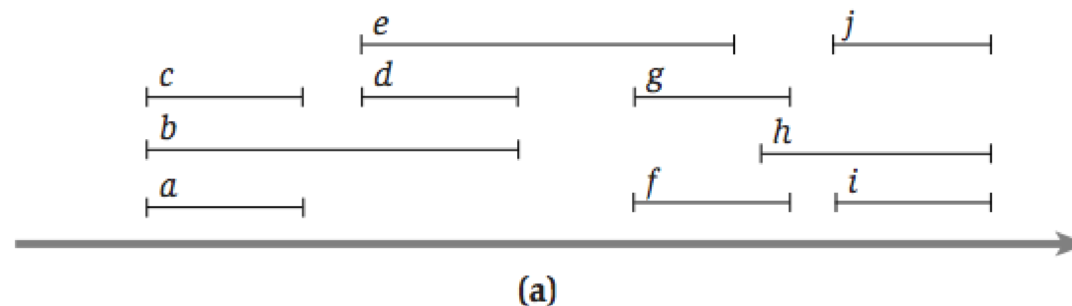


Partizionamento di intervalli: limite inferiore alla soluzione ottima

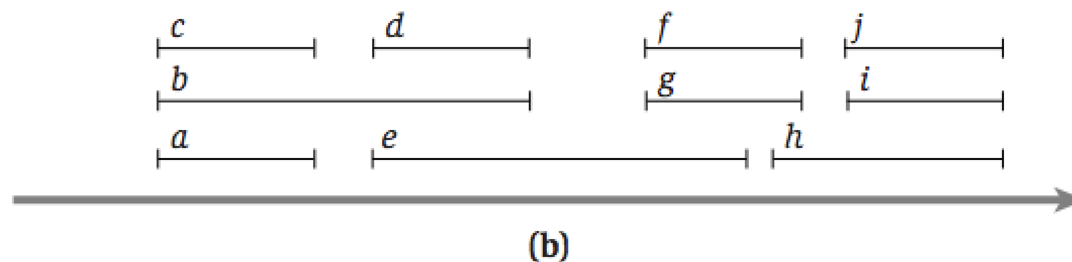
Def. Immaginiamo di disporre gli intervalli lungo l'asse delle ordinate in modo da non avere due intervalli che si sovrappongono alla stessa altezza (un qualsiasi schedule fa al nostro caso). La profondità di un insieme di intervalli è il numero massimo di intervalli intersecabili con una singola linea verticale

Limite inferiore: **Numero di classi necessarie ≥ profondità.**

Esempio. L'insieme di intervalli in figura (a) ha profondità 3. Lo schedule in figura (b) usa 3 risorse ed è quindi ottimo



(a)



(b)

Partizionamento di intervalli: soluzione ottima

Domanda. E' sempre possibile trovare uno schedule pari alla profondità dell'insieme di intervalli?
Osservazione. Se così fosse allora il problema del partizionamento si ridurrebbe a constatare quanti intervalli si sovrappongono in un certo punto. Questa è una caratteristica locale per cui un algoritmo greedy potrebbe essere la scelta migliore.

Nel seguito vedremo un algoritmo greedy che trova una soluzione che usa un numero di risorse pari alla profondità e che quindi è ottimo

Idea dell'algoritmo applicata all'esempio delle lezioni:
- Considera le lezioni in ordine non decrescente dei tempi di inizio
- Ogni volta che esamina una lezione controlla se le può essere allocata una delle aule già utilizzate per qualcuna delle lezioni esaminate in precedenza. In caso contrario alloca una nuova aula.

Partizionamento di intervalli: Algoritmo greedy

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0

for j = 1 to n {
   if (interval j can be assigned an already allocated resources v)
      assign resource v to interval j
   else
      allocate a new resource d + 1
      assign the new resource d+1 to interval j
      d ← d + 1
}
```

- Osservazione. L'algoritmo greedy non assegna mai la stessa risorsa a due intervalli incompatibili

- Dimostreremo che alla j-esima iterazione del for il valore di d è pari alla profondita dell'insieme ordinato di intervalli {1,2,…,j}
- Il valore finale di d è quindi pari alla profondità dell'insieme di intervalli {1,…,n}

Partizionamento di intervalli: Ottimalità soluzione greedy

Lemma. Alla j-esima iterazione del for, il valore di d è pari alla profondità dell'insieme di intervalli {1,2,…,j}

Dim.

Caso in cui alla j-esima iterazione del for viene allocata una nuova risorsa

- Supponiamo che alla j-esima iterazione del ciclo di for d venga incrementato.
- La risorsa d è stata allocata perchè ciascuna delle altre risorse già allocate è assegnata al tempo s_j ad un intervallo che non è ancora terminato al tempo s_j.
- Siccome l'algoritmo considera gli intervalli in ordine non decrescente dei tempi di inizio, tutti gli intervalli a cui sono assegnate le risorse iniziano non più tardi di s_j.
- Di conseguenza, ci sono d intervalli (uno per risorsa allocata) che si sovrappongono al tempo s_j (sono cioè intersecabili da una retta verticale che passa per s_j).
    - Per definizione di profondità d$\leqq$ profondità di {1,2,…,j}
- Abbiamo osservato che il numero di risorse allocate per l'intervallo {1,2,…,j} è maggiore uguale della profondità di {1,2,…,j}
    - Di conseguenza d $\geqq$ profondità di {1,2,…,j}

Dagli ultimi due punti segue che d = profondità di {1,2,…,j}

Partizionamento di intervalli: Ottimalità soluzione greedy

Caso in cui alla j-esima iterazione del for non viene allocata una nuova risorsa:

- Sia j' l'ultima iterazione prima della j-esima in cui viene allocata una nuova risorsa. Per quanto dimostrato nella slide precedente, d= profondità {1,2,…,j'}.

- Siccome {1,2,…,j'} è contenuto in {1,2,…,j} allora si ha che profondità {1,2,…,j'} ≦ profondità di {1,2,…,j} e quindi per il punto precedente d ≦ profondità di {1,2,…,j}.

- Abbiamo osservato che il numero di risorse allocate per un insieme di intervalli è maggiore o uguale della profondità dell'insieme. Si ha quindi d ≧ profondità di {1,2,…,j}.

- Gli ultimi due punti implicano che d=profondità di {1,2,…,j}

Partizionamento di intervalli: Ottimalità soluzione greedy

Teorema. L'algoritmo greedy usa esattamente un numero di risorse pari alla profondità dell'insieme di intervalli {1,2,…,n}

Dim. Per il lemma precedente quando j=n il numero di risorse allocate è uguale alla profondità dell'intervallo {1,2,…,n}

Partizionamento di intervalli: Analisi Algoritmo greedy

Implementazione che richiede O(n log n).

- Per ogni risorsa p, manteniamo il tempo di fine più grande tra quelli degli intervalli assegnati fino a quel momento a p. Indichiamo questo tempo con k_p
- Usiamo una coda a priorità (si vedano le slide successive) contenente tutte le risorse p allocate fino a quel momento. La chiave k(p) della risorsa p è l'istante k_p fino al quale p è occupata.
  - In questo modo l'elemento con chiave minima indica la risorsa v che si rende disponibile per prima
- Sia v la risorsa con chiave più piccola (la ottengo invocando FindMin).
  - Se s_j>k(v) allora all'intervallo j può essere allocata la risorsa v. In questo caso sostituiamo (con ChangeKey) la chiave di v con f_j.
  - In caso contrario allochiamo una nuova risorsa e la inseriamo (con Insert) nella coda associandole la chiave f_j .
- Se usiamo una coda a priorità implementata con heap allora FindMin richiede tempo O(1) e Insert e ChangeKey richiedono tempo O(log m), dove m è la profondità dell'insieme di intervalli. Poiché vengono fatte O(n) operazioni di questo tipo, il costo complessivo del for è O(n log m).
- A questo va aggiunto il costo dell'ordinamento che è O(nlog n). Siccome m≤n il costo dell'algoritmo è O(nlogn)

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

$$O(1 + 2 + \ldots + n-1) = O(n(n-1)/2) = O(n^2)$$

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:

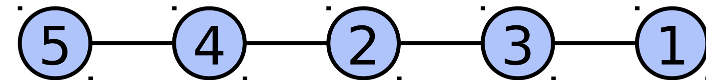
1. Inserire gli elementi nella coda in ordine richiede n chiamate a Insert, e quindi tempo $O(1 + 2 + \ldots + n\text{-}1) = O(n(n-1)/2) = O(n^2)$
2. Rimuovere gli elementi dalla coda in ordine richiede n chiamate a ExtractMin, e quindi tempo $O(n)$

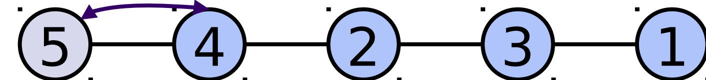
Insertion-sort richiede tempo $O(n^2)$ e spazio aggiuntivo $O(n)$

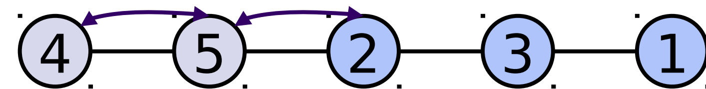
Insertion-sort sul posto

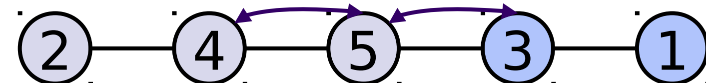
- Invece di usare una struttura di appoggio, sia Insertion-sort che Selection-sort si possono implementare sul posto

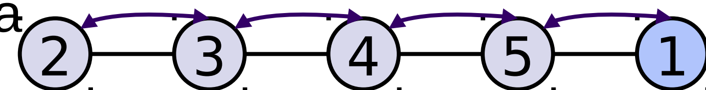
- Usiamo swapElements per spostare gli elementi invece di modificare la struttura della lista
- Una porzione della sequenza in input viene usata nello stesso modo in cui è usata la coda a priorità in PQSort.
- **Insertion-sort sul posto**
  - Manteniamo ordinata la prima parte della sequenza
  - Ad ogni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

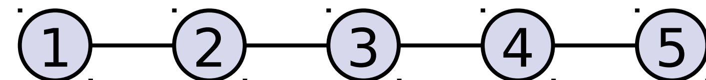






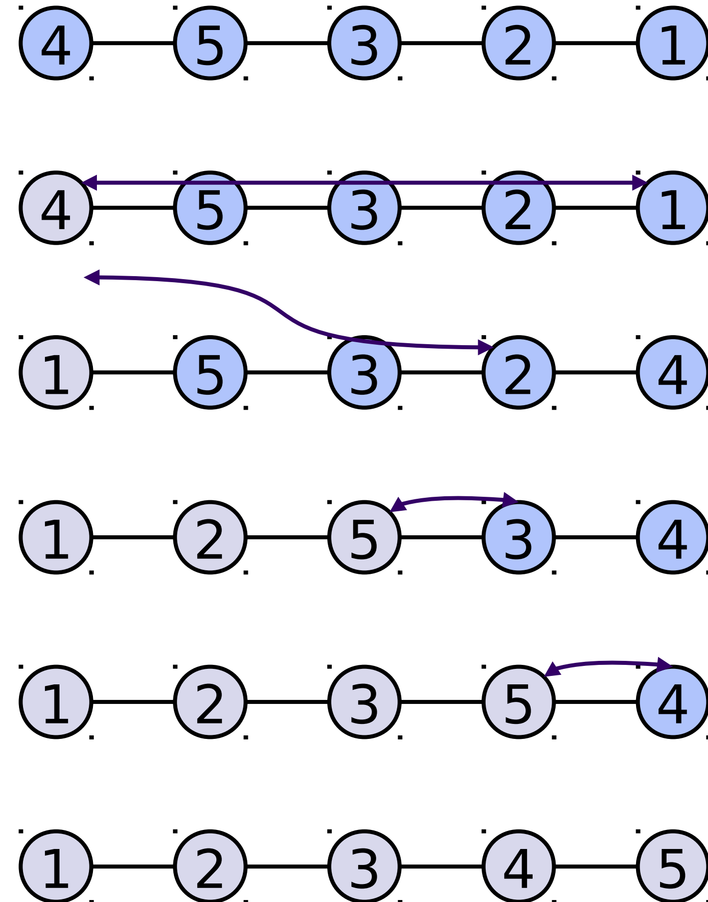






Selection-sort sul posto

- **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



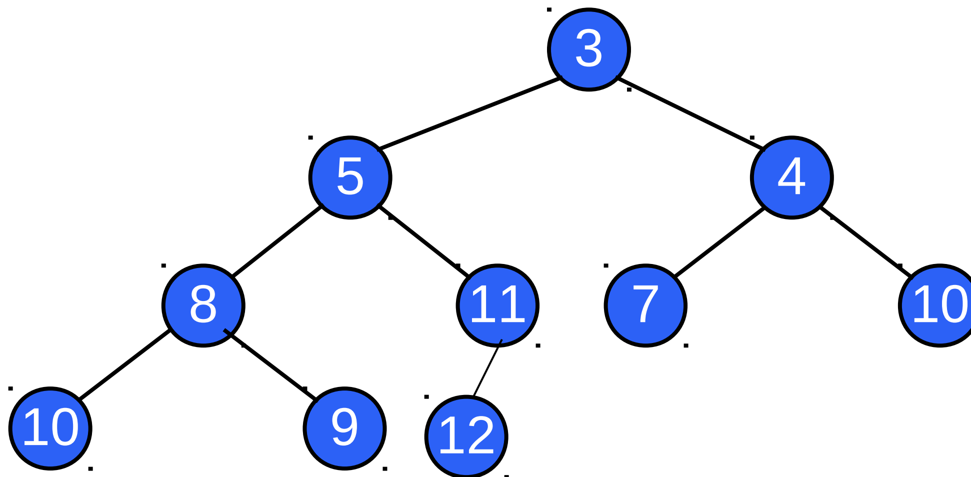
Iimplementazione della coda a priorità mediante un heap

- 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))

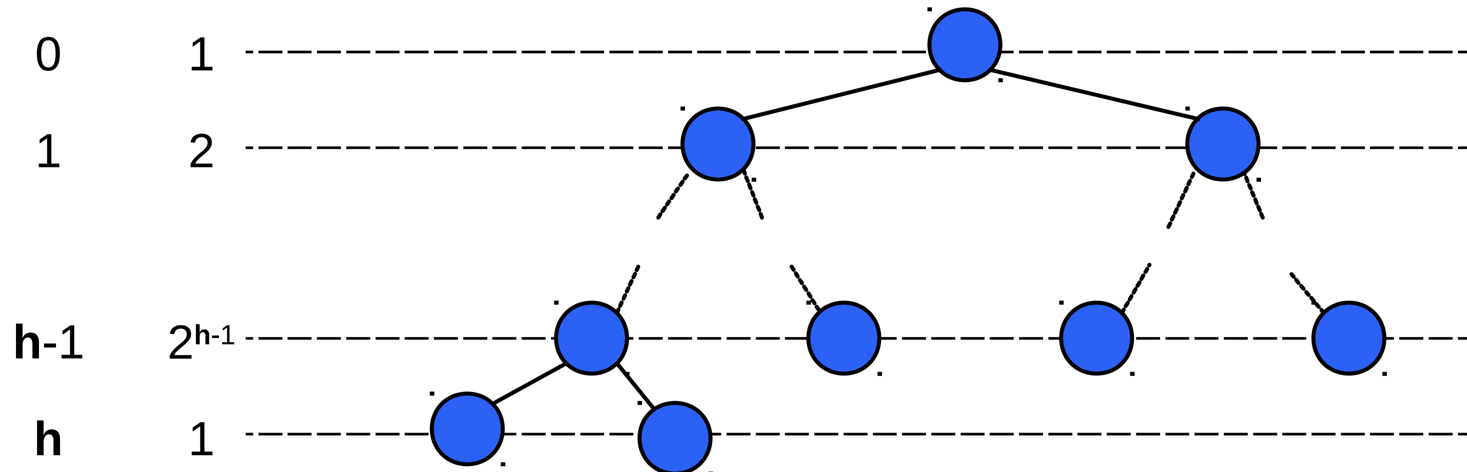
  - Albero binario completo: dato un heap di altezza **h**
    - per $i = 0, \ldots, h-1$, ci sono 2^i nodi di profondità i (tutti i livelli, salvo al più l'ultimo, sono pieni)
    - L'ultimo livello è riempito da sinistra verso destra

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
  - $n \leq 1 + 2 + 4 + \ldots + 2^h = 2^{h+1}-1$

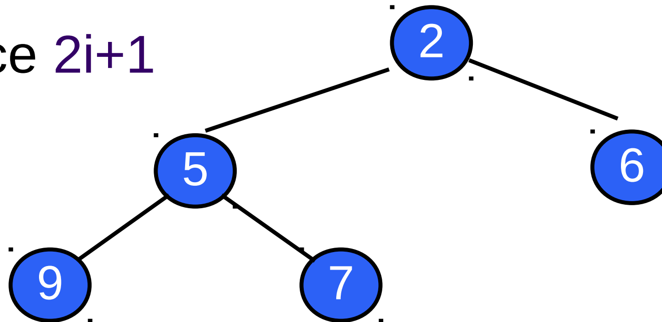
- ➔ $2^h \leq n \leq 2^{h+1}-1$ ➔ $\log(n+1)-1 \leq h \leq \log n$

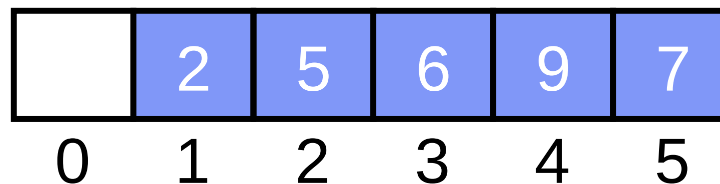
➔ $\log(n)-1 < h \leq \log n$

$h = \lfloor \log n \rfloor$

Implementazione con array

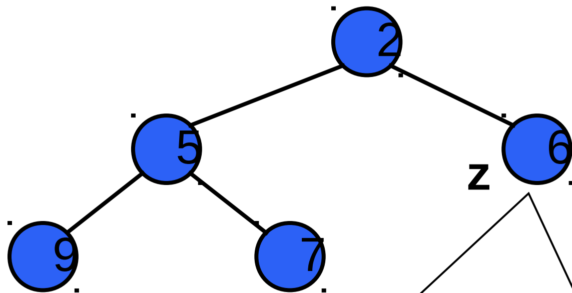
- Per un albero con n nodi si usa un array H di lunghezza n+1
  - entrata di indice 0 vuota
- Per un nodo di indice i
  - Il figlio sinistro ha indice 2i
  - Il figlio destro ha indice 2i+1

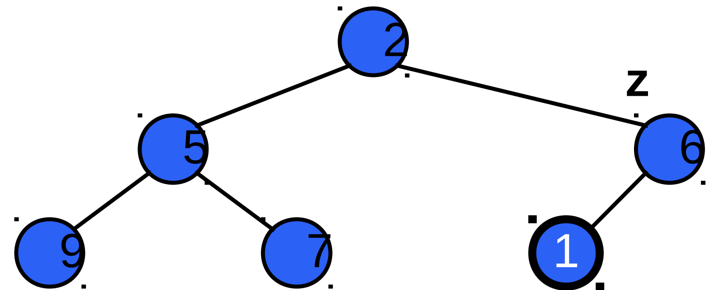




Insert

- 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

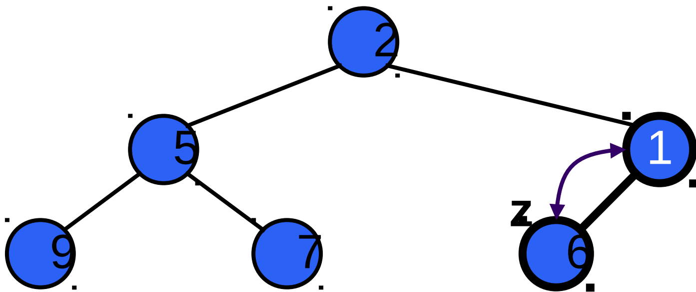


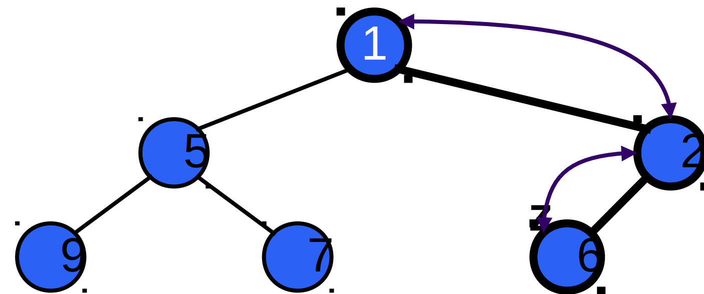


Ripristino dell'heap-order

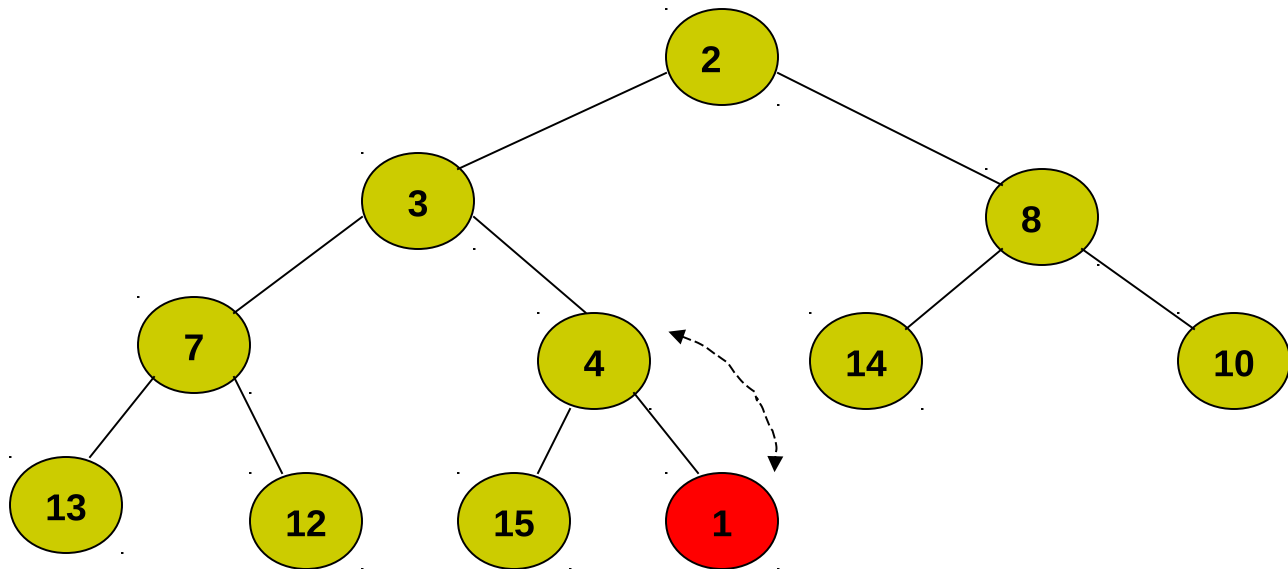
- L'algoritmo Heapify-up ripristina l'heap-order scambiando il nuovo elemento z con i suoi antenati fino a che z raggiunge la radice o si incontra un antenato con chiave minore di **key(z)**

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

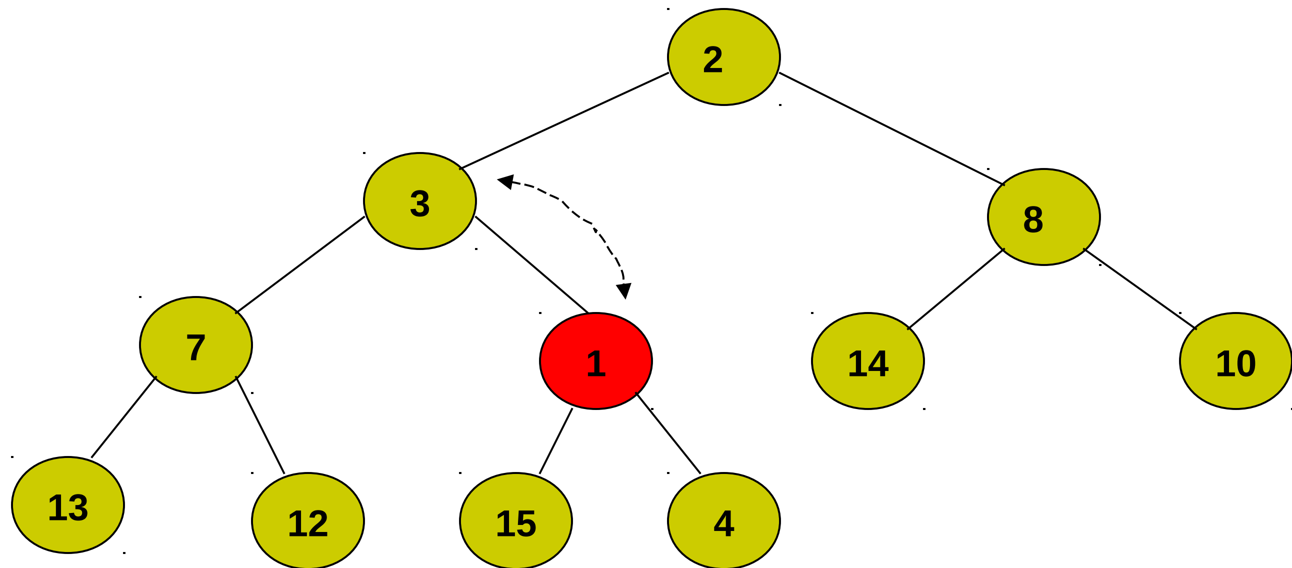




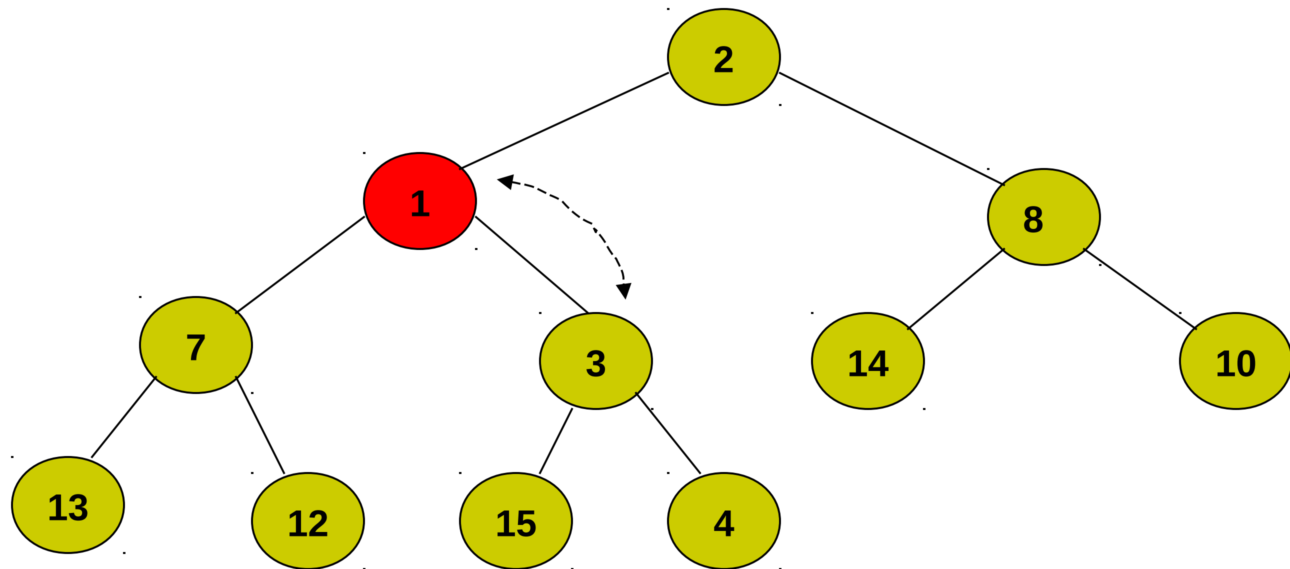
Inserimento della chiave 1



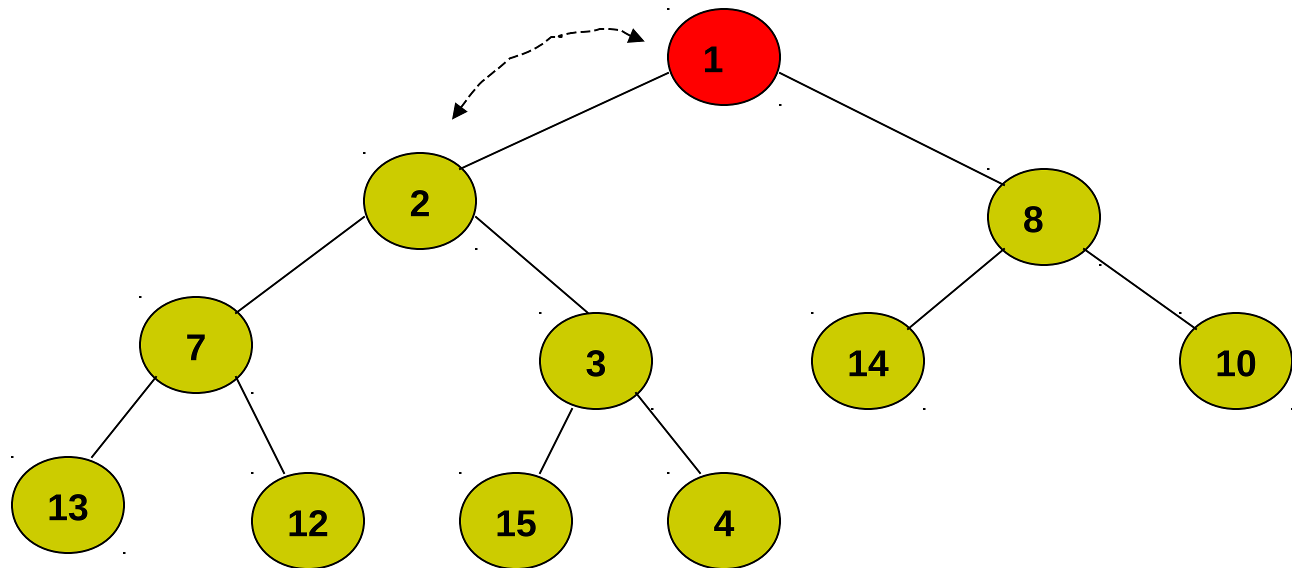
Inserimento della chiave 1



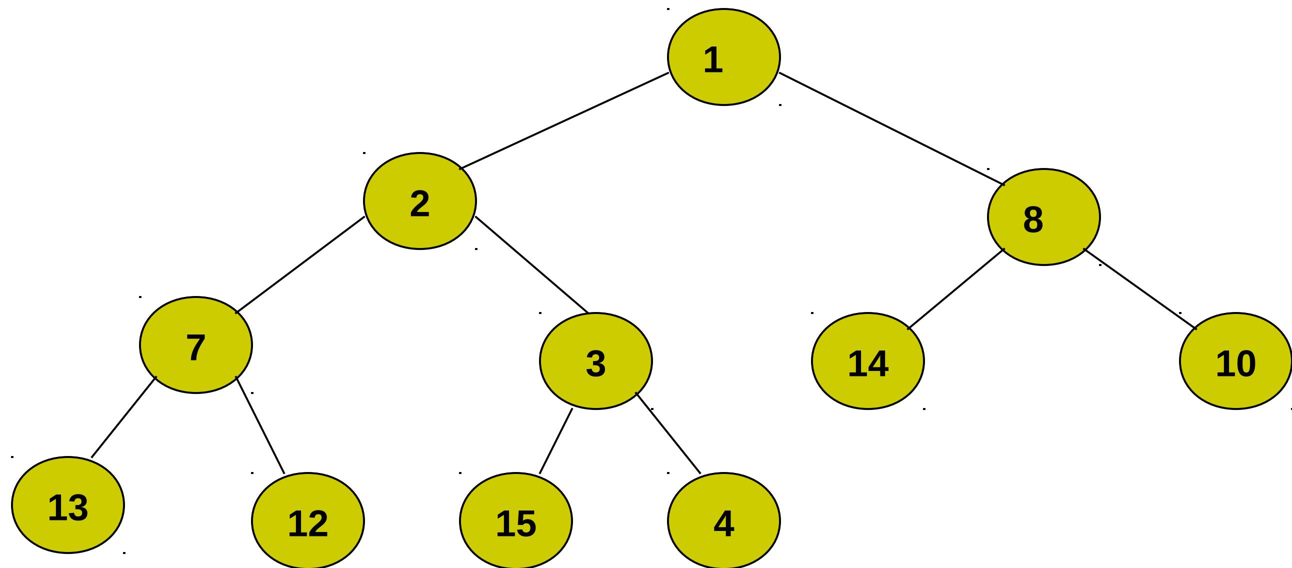
Inserimento della chiave 1



Inserimento della chiave 1



Inserimento della chiave 1

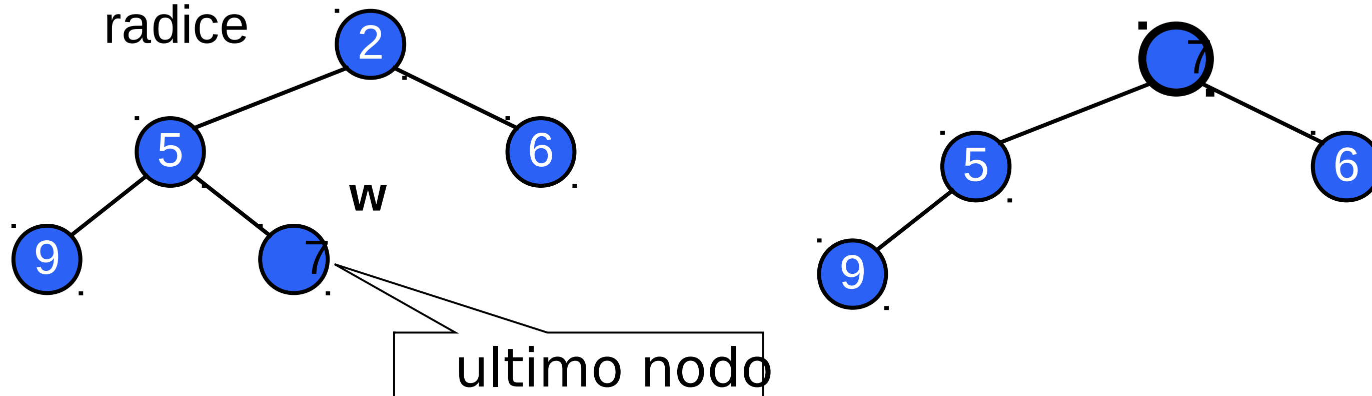


Heapify-up

```
Heapify-up(H,i):
 If i > 1 then
   let j = parent (i) = ⌊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
```

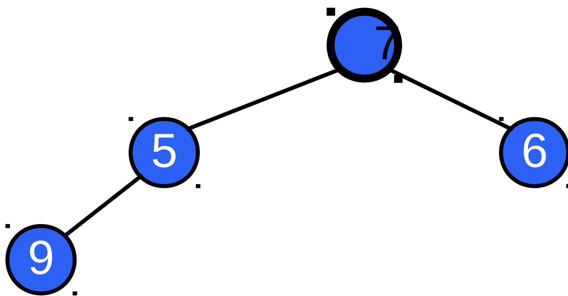
ExtractMin

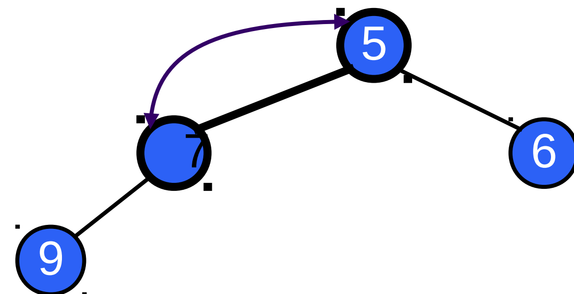
- ExtractMin è implementato rimuovendo l'entrata nella radice dell'heap
- L'algoritmo di rimozione consiste di 3 passi:
  - Sostituisci l'entrata della radice con l'entrata dell'ultimo nodo w
  - Rimuovi w
  - Ripristina con Heapify-down l'heap-order che potrebbe essere stato violato dalla sostituzione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



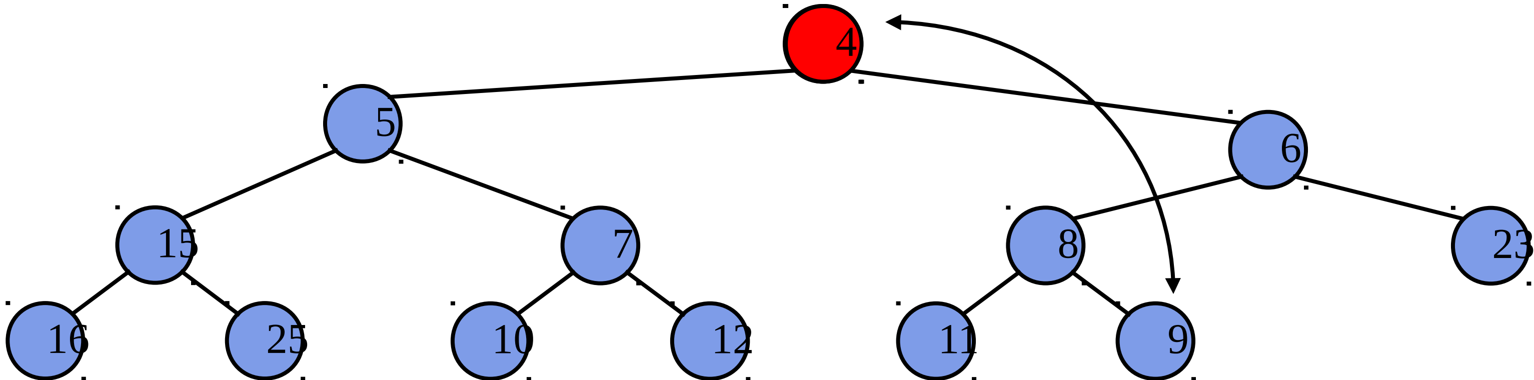
Heapify-down

- 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)

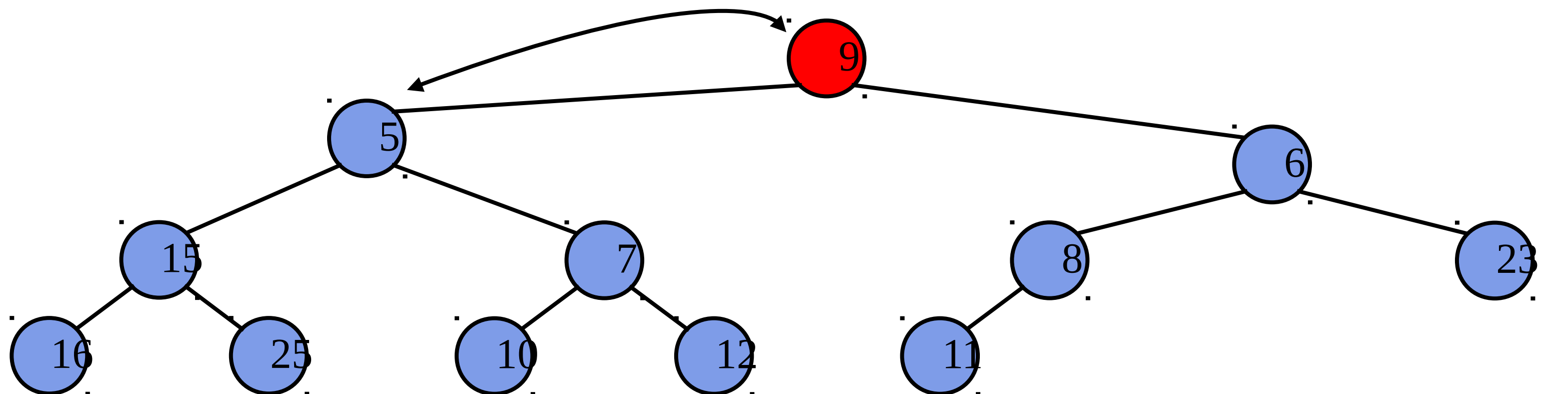




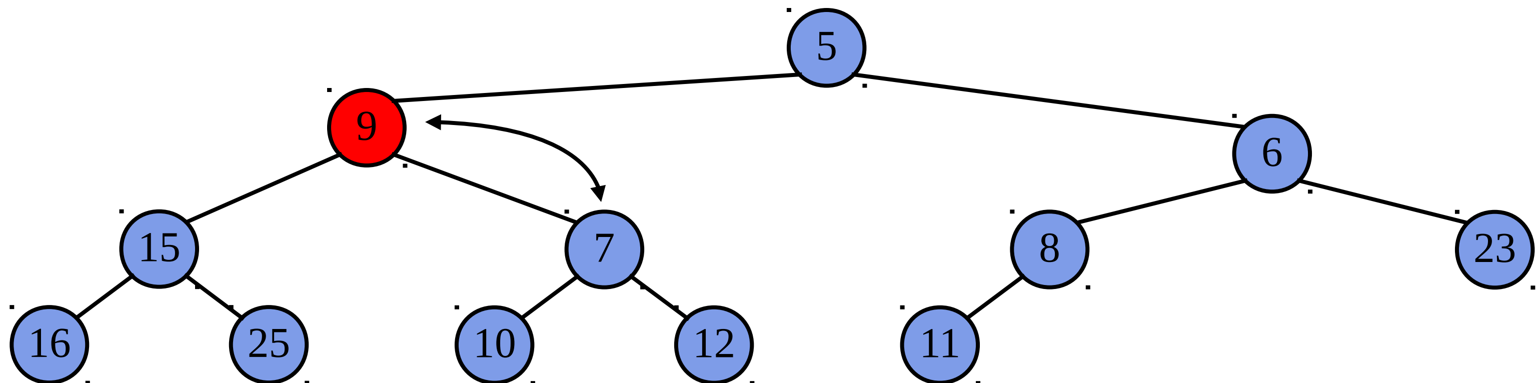
Cancellazione del minimo



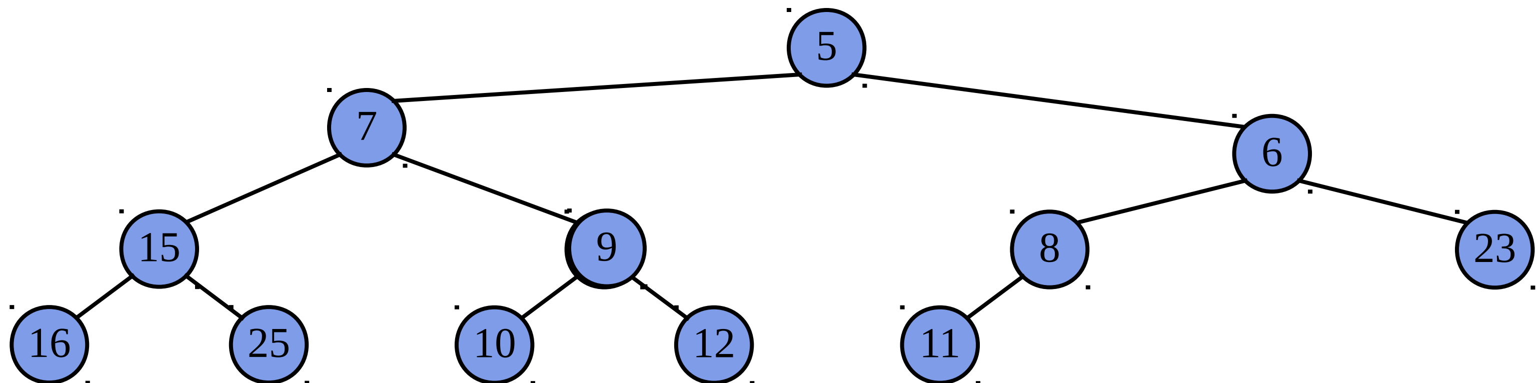
Cancellazione del minimo



Cancellazione del minimo



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
   Let j be the index that minimizes key[ H [left]] and key[ H [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 $O(n \log n)$