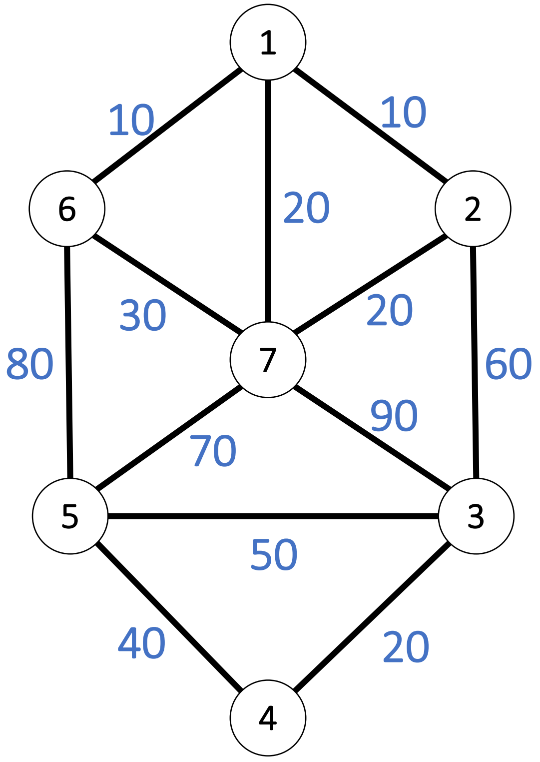
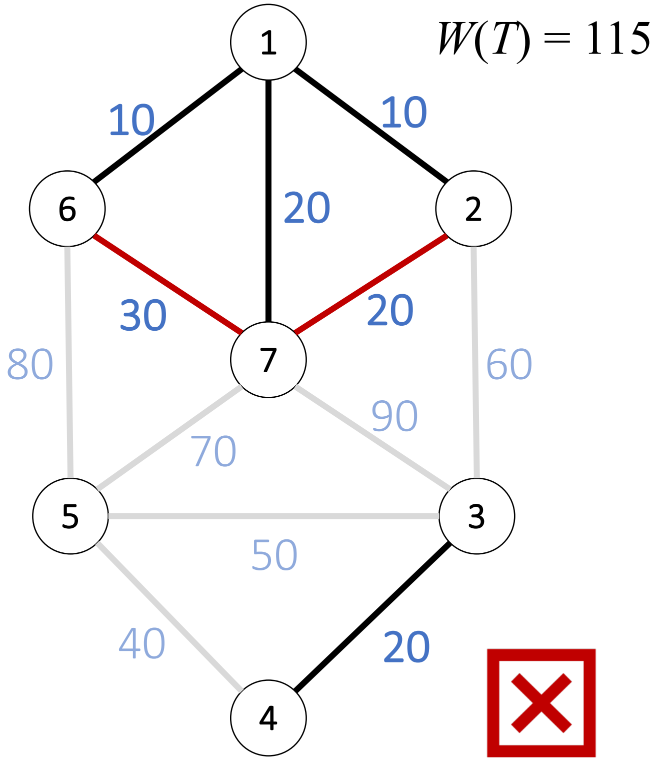
Università Roma Tre – Dipartimento di Matematica e Fisica
IN440 – Ottimizzazione Combinatoria
Prof. Marco Liverani

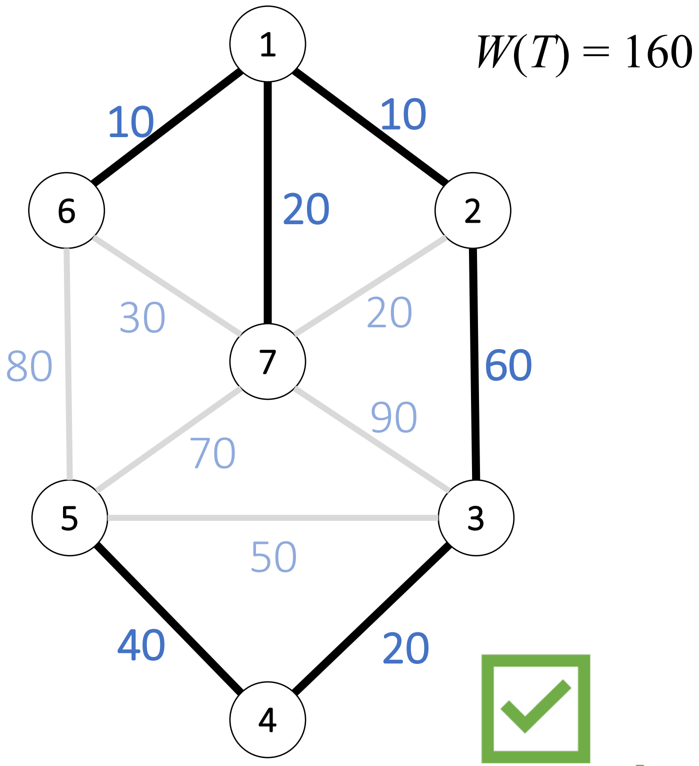
Alberi ricoprenti di costo minimo

Minimum Spanning Tree (MST)

- Dato un grafo $G = (V, E)$ con dei costi non negativi $w_{ij} \geq 0$ assegnati agli spigoli del grafo, si chiede di individuare un albero ricoprente $T = (V, E')$ di costo minimo
  - il costo $W(T)$ dell'albero ricoprente T è dato dalla somma dei costi assegnati ai suoi spigoli
  - l'albero ricoprente di costo minimo esiste sempre, ma può non essere unico
  - l'albero ricoprente ha certamente n vertici ed $n - 1$ spigoli, ma scegliere gli $n - 1$ spigoli di G che pesano meno potrebbe non portare ad una soluzione (l'albero deve essere privo di cicli e connesso)







Algoritmo generico

- Gli algoritmi per la costruzione del MST procedono nella costruzione dell'albero ricoprente T sulla base di diversi criteri con cui è definito l'*ordine di valutazione* degli spigoli da aggiungere all'albero

- Gli algoritmi usano la tecnica ***greedy*** (golosa): ad ogni passo viene fatta la scelta *localmente ottima*, nella speranza che questo procedimento conduca ad una soluzione *globalmente ottima*

- La scelta localmente ottima (golosa) è quella di considerare lo spigolo più leggero tra quelli ammissibili (aggiungere uno spigolo non deve creare un ciclo in T)

Algoritmo 14 MST-GENERICO($G, w : E(G) \to \mathbb{R}$)

Input: Il grafo G ed una funzione peso $w(u,v)$ degli spigoli di G
Output: Un albero ricoprente T di peso minimo per G

1: $T = \emptyset$
2: **fintanto che** T non è ricoprente per G ($|E(T)| < n-1$) **ripeti**
3:     trova uno spigolo $(u,v) \in E(G)$ tale che $(u,v) \notin E(T)$ e $E(T) \cup \{(u,v)\}$ è ancora un sottoalbero di un albero di copertura minimo
4:     $E(T) = E(T) \cup \{(u,v)\}$
5: **fine-ciclo**

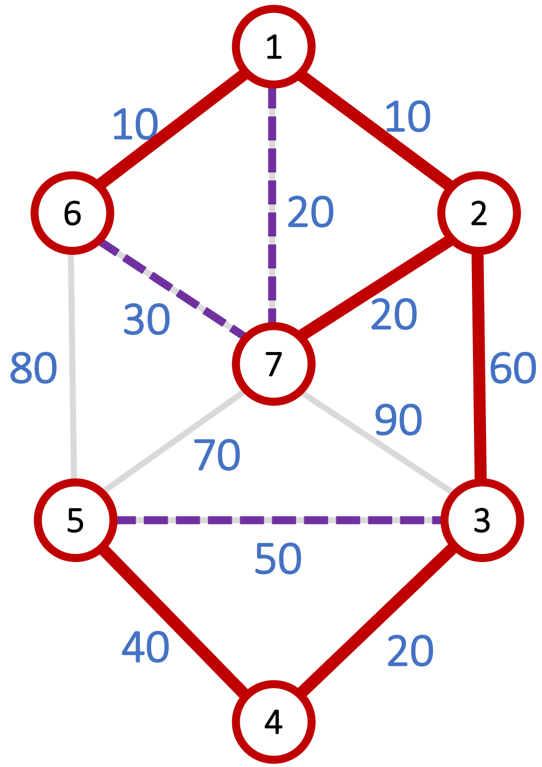
- Per poter attuare la tecnica greedy il problema deve godere della proprietà di **sottostruttura ottima**: la soluzione ottima contiene le soluzioni ottime per i sotto-problemi; nel problema MST l'albero ricoprente di costo minimo contiene i sottoalberi ricoprenti di costo minimo per i sottoproblemi

Algoritmo di Kruskal


Joseph Kruskal
(1928 – 2010)

- L'**algoritmo di Kruskal** (1956) costruisce la soluzione aggregando gli alberi di una *spanning forest* composta da *n* vertici indipendenti ottenendo di volta in volta una spanning forest con un albero in meno ed un peso complessivo maggiore, ma minimo rispetto a tutte le possibili spanning forest con tale numero di alberi

- Si passa da una spanning forest alla successiva aggiungendo alla foresta uno spigolo che collega fra loro due alberi della foresta:
  - tale spigolo viene scelto tra tutti gli spigoli che uniscono vertici di alberi distinti, selezionando quello di peso minimo
  - lo spigolo aggiunto non può introdurre dei cicli perché unisce due componenti non connesse della foresta



Algoritmo di Kruskal

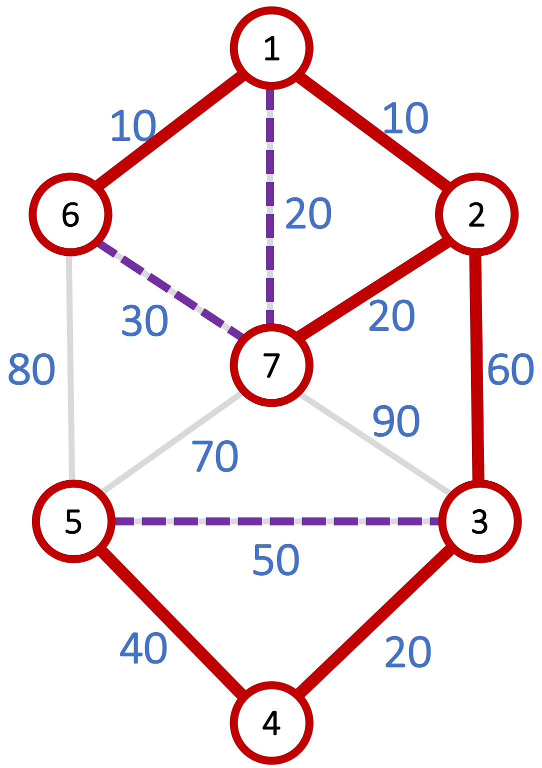
- Gli spigoli vengono scelti in ordine crescente di peso: per questo vengono ordinati in base al peso w_{uv} (riga 5)
- L'operazione critica è verificare se uno spigolo collega due vertici appartenenti allo stesso sotto-albero o a due alberi differenti
- Kruskal definisce delle strutture dati per la rappresentazione di una **collezione di insiemi disgiunti**:
  - vertici di uno stesso sotto-albero ricoprente appartengono allo stesso insieme
  - inizialmente ogni vertice appartiene ad un insieme a sé stante (ciclo 2-4)
- **Make-Set(v)**: crea un insieme con il solo vertice v (riga 3)
- **Find-Set(v)**: trova il rappresentante dell'insieme a cui appartiene il vertice v (riga 7)
- **Union(u, v)**: unisce gli insiemi disgiunti a cui appartengono i vertici u e v (riga 9)

Algoritmo 15 MST-KRUSKAL($G, w : E(G) \to \mathbb{R}$)

Input: Il grafo G ed una funzione peso $w(u,v)$ degli spigoli di G
Output: Un albero ricoprente T di peso minimo per G

1: $T = (V, \emptyset)$
2: **per ogni** $v \in V(G)$ **ripeti**
3:     Make-Set(v)
4: **fine-ciclo**
5: ordina E in ordine non decrescente di peso $w(u,v)$, dallo spigolo di peso minimo a quello di peso massimo
6: **per ogni** $(u,v) \in E(G)$ in ordine non decrescente di peso **ripeti**
7:     **se** Find-Set(u) $\neq$ Find-Set(v) **allora**
8:         $E(T) = E(T) \cup \{(u,v)\}$
9:         Union(u, v)
10:     **fine-condizione**
11: **fine-ciclo**

Algoritmo di Kruskal



Algoritmo 15 MST-KRUSKAL($G, w : E(G) \to \mathbb{R}$)

Input: Il grafo G ed una funzione peso $w(u,v)$ degli spigoli di G

Output: Un albero ricoprente T di peso minimo per G

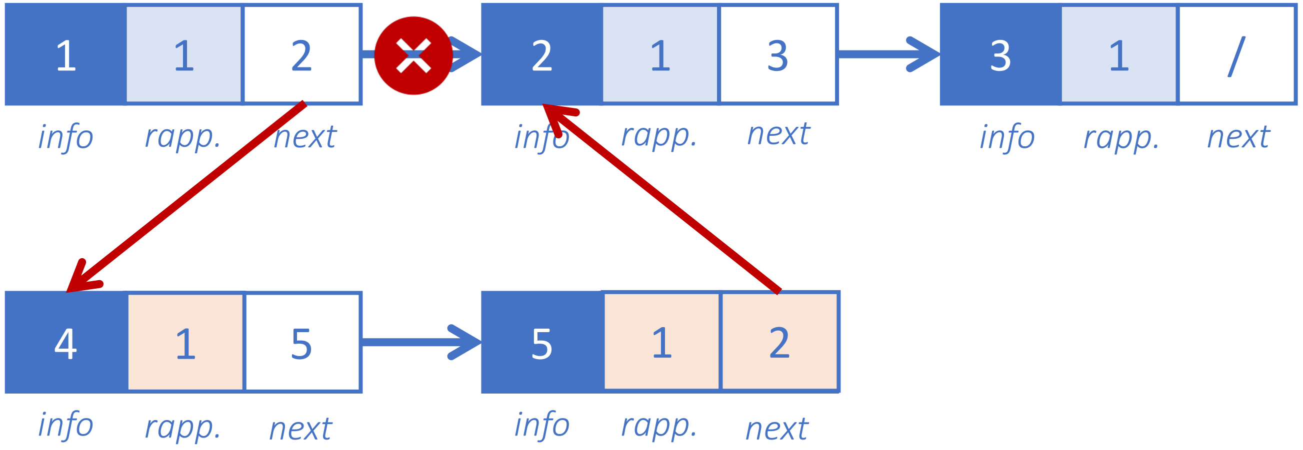
```
T = (V, ∅)
per ogni v ∈ V(G) ripeti
    Make-Set(v)
fine-ciclo
ordina E in ordine non decrescente di peso w(u,v), dallo spigolo
di peso minimo a quello di peso massimo
per ogni (u,v) ∈ E(G) in ordine non decrescente di peso ripeti
    se Find-Set(u) ≠ Find-Set(v) allora
        E(T) = E(T) ∪ {(u,v)}
        Union(u,v)
    fine-condizione
fine-ciclo
```

	✓	✓	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	
$E(T) = \langle$	(1,2),	(1,6),	(3,4),	(2,7),	(1,7),	(6,7),	(4,5),	(3,5),	(2,3),	(5,7),	(5,6),	(3,7)	$\rangle$
	10	10	20	20	20	30	40	50	60	70	80	90	

$W(T) = 160$

Algoritmo di Kruskal

- Strutture dati per insiemi disgiunti



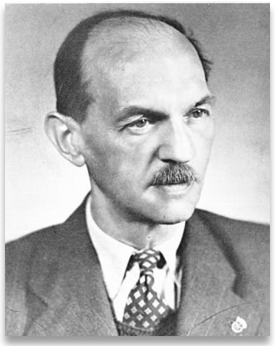
- **Make-Set**(v): $O(1)$, deve assegnare il valore v a info e rappresentante (viene ripetuto per n volte)
- **Sort**($\mathrm{E(G)}$): $O(m \log_2 m)$, viene ordinato un insieme con m elementi (gli spigoli del grafo G)
- **Find-Set**(v): $O(1)$, deve restituire il rappresentante di v (viene ripetuto al massimo per $2m$ volte)
- **Union**(u, v): $O(n)$, deve inserire la lista di v nella lista di u (tra il primo e il secondo elemento) e aggiornare tutti i rappresentanti degli elementi della lista di v (viene ripetuto $n - 1$ volte)

- Complessità algoritmo di Kruskal: $O(n + m \log_2 m + m + n)$; siccome $m < n^2$ si può scrivere $O(m \log_2 n)$

Algoritmo di Prim

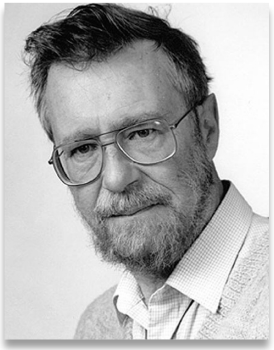
- L’algoritmo noto con il nome di **Robert Prim** venne proposto per la prima volta, negli anni ‘30 dal matematico cecoslovacco **Vojtěch Jarnik** e solo successivamente, negli anni ‘50, fu definito autonomamente anche da Prim e da Dijkstra, che lo diffusero su larga scala

- Partendo da un vertice arbitrario si espande l’albero ricoprente (è sempre unico durante l'esecuzione dell'algoritmo) scegliendo di volta in volta lo spigolo più leggero tra quelli ammissibili
- Per la scelta dello spigolo si usa una coda di priorità (un *heap*, ad esempio) in cui si collocano i vertici del grafo
- La chiave della coda di priorità è il peso minimo di uno spigolo incidente il vertice nella coda e un vertice non appartenente alla coda: $k_v = \min_{u \notin Q} w_{uv}$ per ogni $v \in Q$



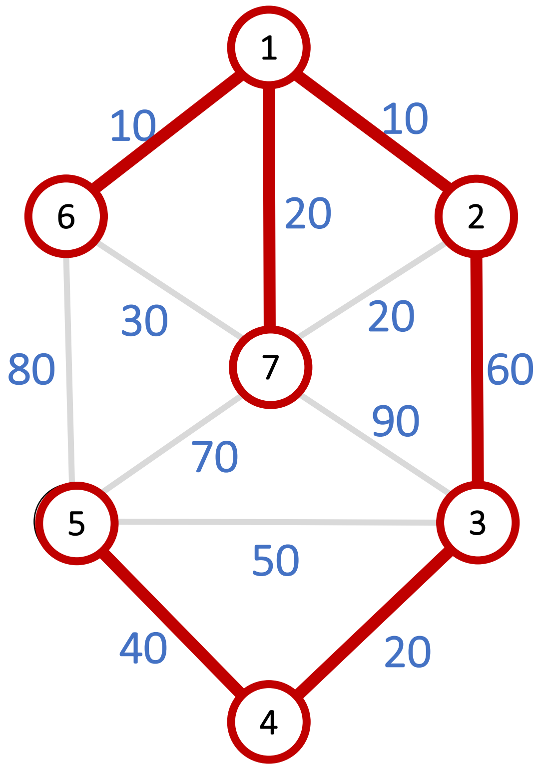
Vojtěch Jarník
(1897 – 1970)

Robert C. Prim
(1921)



Edsger W. Dijkstra
(1930 – 2002)

Algoritmo di Prim



Q

k_v	0	10	60	20	40	10	20
	1	2	3	4	5	6	7

Algoritmo 16 MST-PRIM($G, w : E(G) \to \mathbb{R}, s$)

Input: Il grafo G ed una funzione peso $w(u,v)$ degli spigoli di G e un vertice $s \in V(G)$

Output: Un albero ricoprente T di peso minimo per G

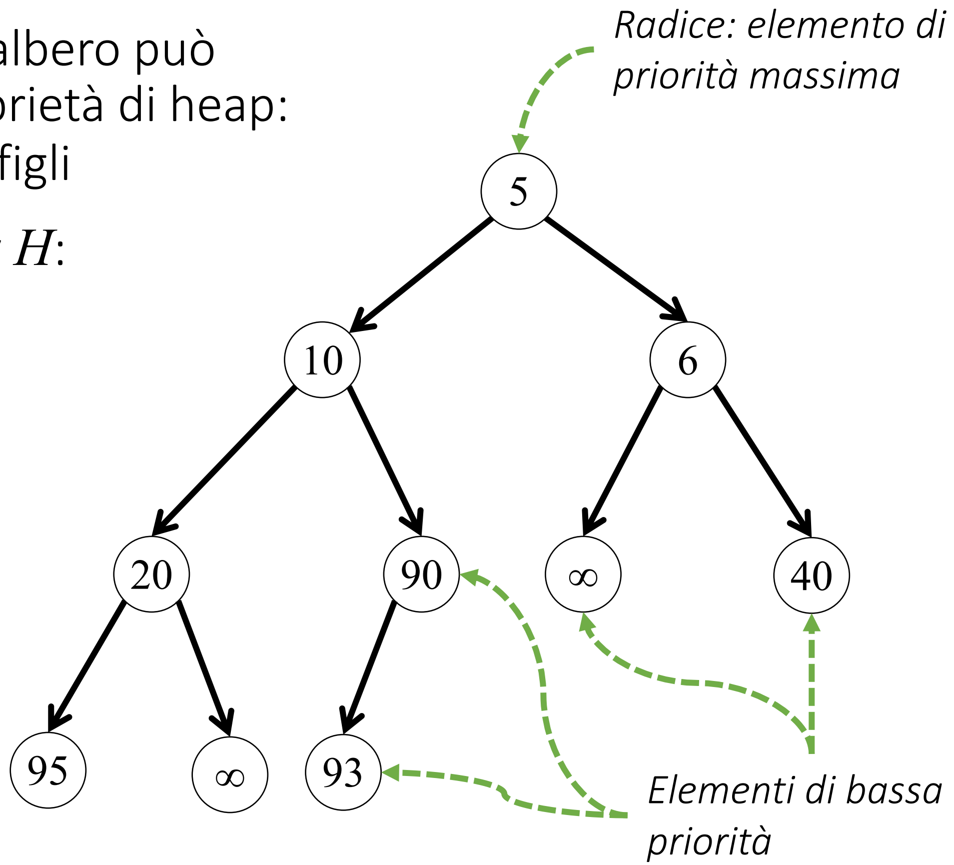
```
T = ∅
per ogni u ∈ V(G) ripeti
    k_u = ∞, π_u = nil
fine-ciclo
k_s = 0
inserisci nella coda di priorità tutti i vertici del grafo: Q = V(G)
fintanto che Q ≠ ∅ ripeti
    sia u l'elemento minimo in Q; estrai u da Q
    per ogni v ∈ N(u) ripeti
        se v ∈ Q e w(u,v) < k_v allora
            k_v = w(u,v), π_v = u
        fine-condizione
    fine-ciclo
fine-ciclo
```

Complessità algoritmo di Prim: $O(n + n + n \log_2 n + m \log_2 n)$
siccome $n - 1 \leq m < n^2$ si può scrivere $O(m \log_2 n)$

Code di priorità: la struttura dati di heap

- Una **coda di priorità** è una struttura dati in cui ad ogni elemento viene assegnata una priorità; viene estratto dalla coda di priorità di volta in volta l'elemento di priorità più alta; a parità di priorità viene estratto per primo l'elemento inserito per primo
- Una coda di priorità può essere rappresentata con una struttura di **heap**
- Un heap è un **albero binario completo** (al più l'ultimo livello dell'albero può essere incompleto, purché sia completato da sinistra) con la proprietà di heap: la priorità di un vertice è maggiore o uguale alla priorità dei suoi figli
- Un heap può essere rappresentato facilmente mediante un array H:
  - H_1 è la radice, l'elemento di priorità massima;
  - se H_i è un vertice dell'heap, i suoi figli sono in H_{2i} e H_{2i+1}
  - se H_i è un vertice dell'heap, con $i > 1$, il padre è $H_{[i/2]}$

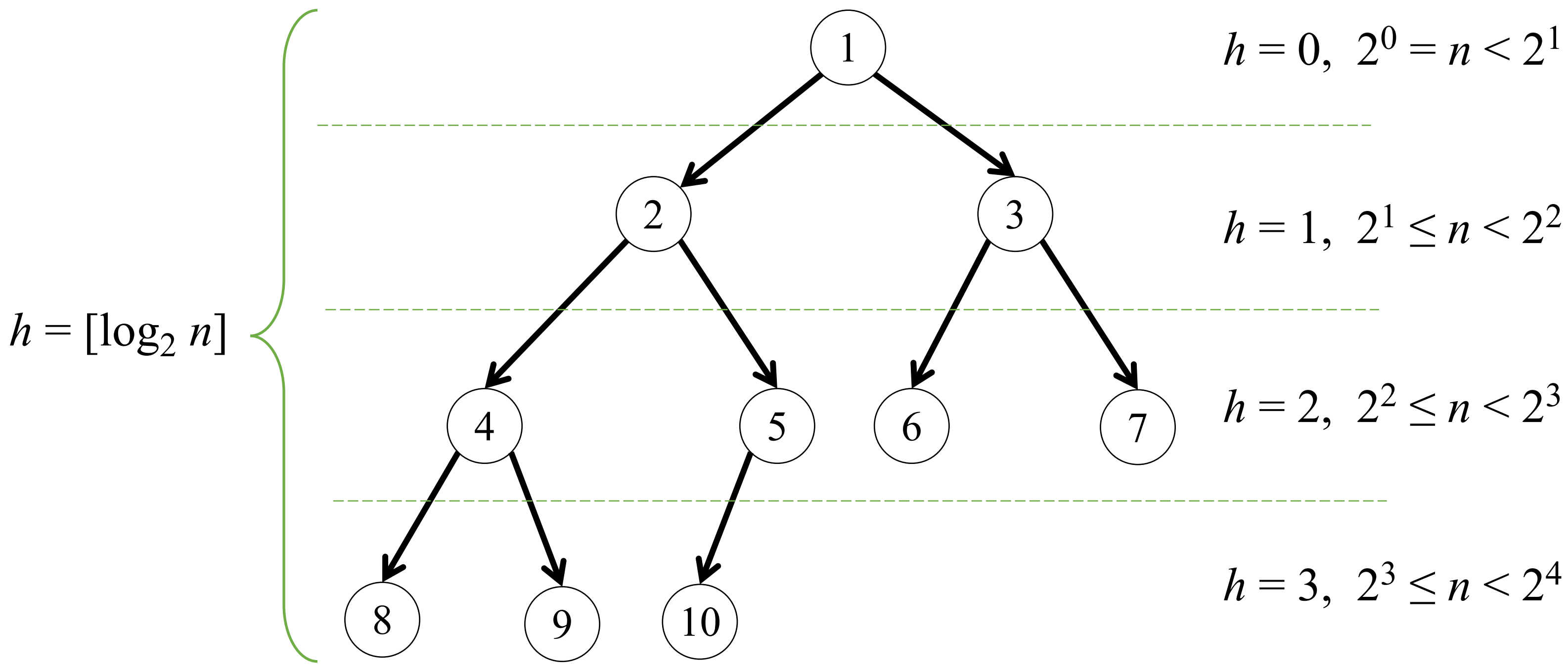
	0	1	2	3	4	5	6	7	8	9	10
H		5	10	6	20	90	∞	40	95	∞	93



Un heap binario H

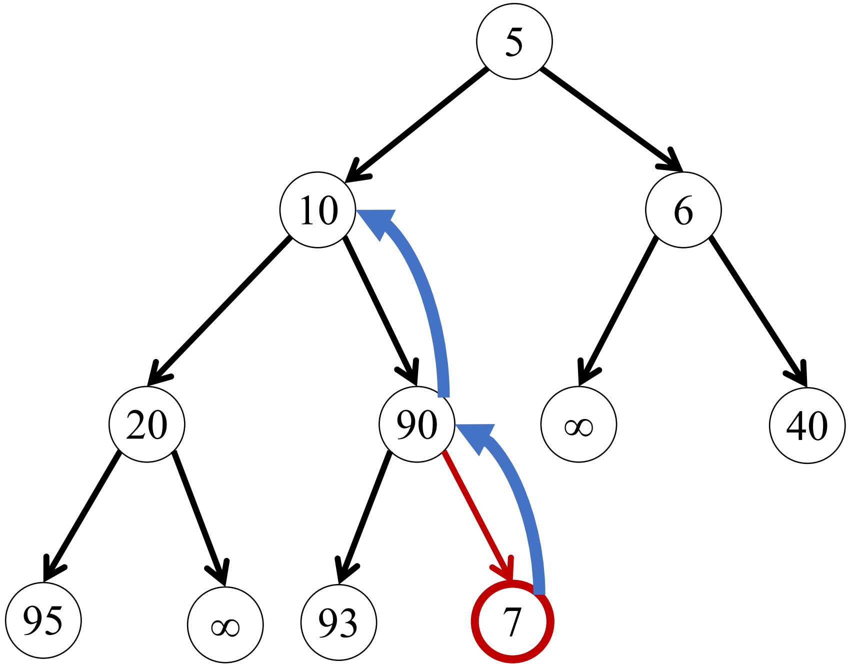
Code di priorità: la struttura dati di heap

- Su un albero binario completo di altezza h con n vertici vale la seguente relazione: $2^h \leq n < 2^{h+1}$
- Quindi l'altezza di un albero binario completo con n vertici è data da: $h = [\log_2 n]$



Code di priorità: operazioni sulla struttura dati di heap

- Su un heap si possono implementare in modo efficiente tre operazioni di base:
  - Insert(H, x): inserisce nell'heap H un elemento con priorità x
  - Extract(H): estrae dall'heap H l'elemento di priorità massima
  - Change(H, x, x'): modifica la priorità dell'elemento da x a x' (x' priorità più elevata di x)
- Le tre operazioni si possono implementare in tempo $O(\log_2 n)$, tempo lineare nell'altezza dell'heap



Algoritmo 18 INSERT(H,x)

Input: Un heap H e l'elemento con priorità x da inserire
Output: L'heap H con il nuovo elemento x

sia i l'indice della prima posizione disponibile in H
$h_i = x$
fintanto che $i > 1$ e $h_i > h_{\lfloor i/2 \rfloor}$ **ripeti**
  scambia h_i e $h_{\lfloor i/2 \rfloor}$
  $i = \lfloor i/2 \rfloor$
fine-ciclo

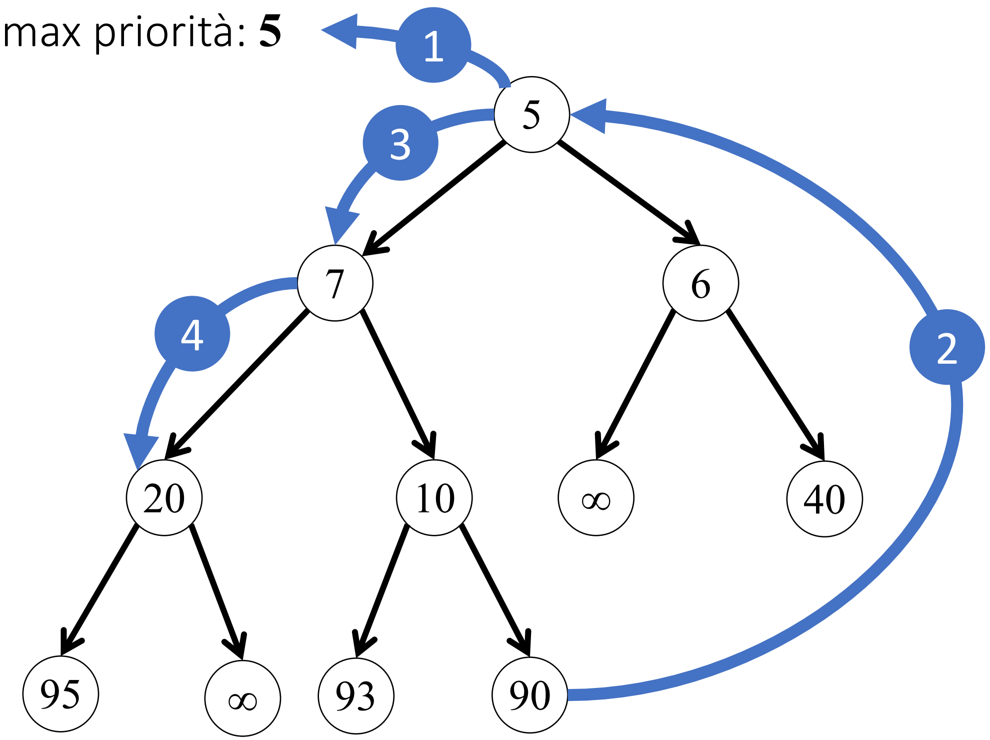
Si aggiunge il nuovo elemento x come ultima foglia dell'albero H e lo si confronta iterativamente con il padre:

- se il nuovo elemento ha priorità più alta rispetto al padre si scambia la posizione del nuovo elemento con quella del padre
- si eseguono al più $\log_2 n$ iterazioni / confronti / scambi

Code di priorità: operazioni sulla struttura dati di heap

- Estrazione dell'elemento di priorità massima:
  - Si estrae la radice dell'heap (elemento di massima priorità)
  - Si ricostruisce l'heap spostando sulla radice l'ultima foglia per poi ricollocarla nella posizione corretta confrontandola e scambiandola iterativamente con il figlio di priorità più elevata fino a trovare la posizione corretta

max priorità: **5**

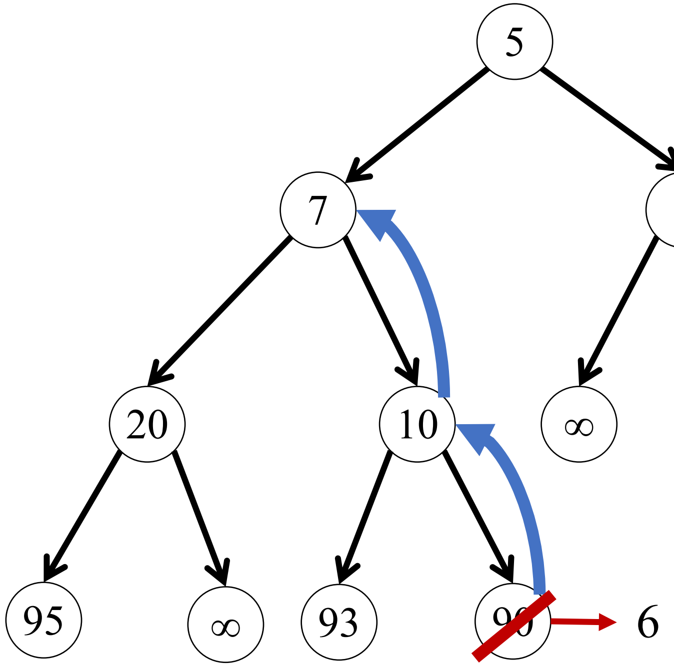


Algoritmo 19 EXTRACT(H)

$max = h_1$
sia n l'indice della posizione dell'ultimo elemento presente in H
$h_1 = h_n$
$i = 1$
fintanto che $2i < n$ e $(h_i > h_{2i}$ o $h_i > h_{2i+1})$ **ripeti**
    se $h_{2i} > h_{2i+1}$ **allora**
        scambia h_i e h_{2i}
        $i = 2i$
    altrimenti
        scambia h_i e h_{2i+1}
        $i = 2i + 1$
    fine-condizione
fine-ciclo
se $i = l/2$ e $h_i < h_{2i}$ **allora**
    scambia h_i e h_{2i}
fine-condizione
restituisci max

Code di priorità: operazioni sulla struttura dati di heap

- Modifica della posizione nell'heap dell'elemento h_i che passa da priorità x a priorità superiore x':
  - si modifica la priorità dell'elemento h_i
  - si confronta iterativamente l'elemento h_i con il vertice padre $h_{[i/2]}$ e si scambiano i due elementi se la priorità di h_i è superiore a quella di $h_{[i/2]}$



Algoritmo 20 CHANGE(H, x, x')

Input: Un heap H e l'elemento che passa da priorità x a priorità x' superiore ad x

Output: L'heap H con l'elemento x' ricollocato in base alla sua priorità

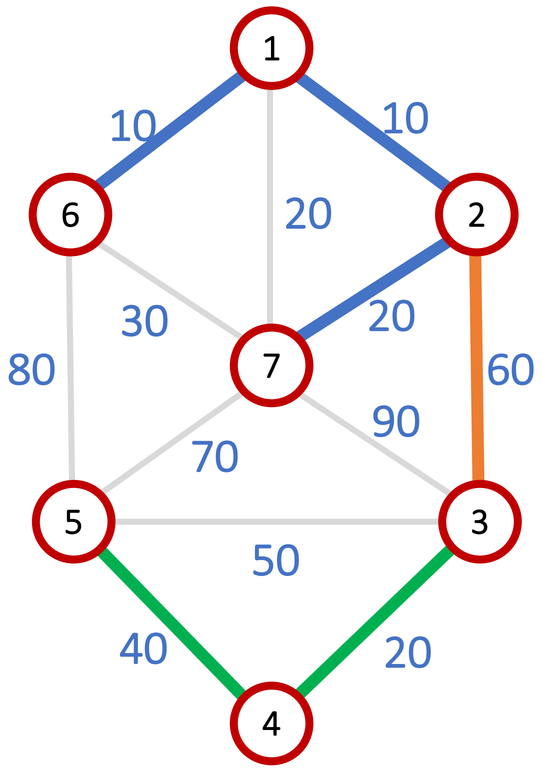
```
sia i l'indice dell'elemento di priorità x
sia x' la nuova priorità dell'elemento h_i
fintanto che i > 1 e h_i > h_⌊i/2⌋ ripeti
    scambia h_i e h_⌊i/2⌋
    i = ⌊i/2⌋
fine-ciclo
```

Algoritmo di Borůvka - Sollin

Otakar Borůvka
(1899 – 1995)

- Questo algoritmo venne progettato nel 1926 da Otakar Borůvka nell'ambito di uno studio per la realizzazione di reti di distribuzione elettrica e successivamente è stato riscoperto dal matematico Georges Sollin nel 1962
- Storicamente è il primo algoritmo per la soluzione del problema MST
- La strategia dell'algoritmo di Sollin e Borůvka per il Minimum Spanning Tree è di aggregare a due a due gli alberi di una foresta di n alberi nulli (uno per ogni vertice di G), unendo tra loro gli alberi «più vicini», in base al peso crescente degli spigoli che collegano vertici di alberi diversi
  - In questo modo, ad ogni iterazione del ciclo principale dell'algoritmo si dimezza il numero di alberi disgiunti che compongono la foresta ricoprente di G

Algoritmo di Borůvka - Sollin



Complessità algoritmo di Borůvka: $O(m \log_2 n)$
Il ciclo principale viene iterato $O(\log_2 n)$ volte e per ogni iterazione si valutano tutti gli m spigoli del grafo per individuare i più convenienti per collegare vertici di alberi distinti

Algoritmo 17 MST-BORUVKA($G, w : E(G) \to \mathbb{R}$)

Input: Il grafo G ed una funzione peso $w(u, v)$ degli spigoli di G
Output: Un albero ricoprente T di peso minimo per G

```
T = (V, ∅) foresta di alberi nulli, ciascuno con un vertice di G, ma
  senza spigoli
fintanto che |E(T)| < n − 1 ripeti
    sia T = T_1 ∪ T_2 ∪ ··· ∪ T_p la foresta ricoprente G
    per k = 1, ..., p ripeti
        sia (u_k, v_k) lo spigolo tra u_k ∈ V(T_k) e il vertice v_k ∉ V(T_k)
        più vicino
    fine-ciclo
    per k = 1, ..., p ripeti
        se u_k e v_k non appartengono allo stesso albero allora
            fondi T_{u_k} e T_{v_k}
            E(T) = E(T) ∪ {(u_k, v_k)}
        fine-condizione
    fine-ciclo
fine-ciclo
```

Riferimenti bibliografici

- Cormen, Leiserson, Rivest, Stein, «*Introduzione agli algoritmi e strutture dati*», terza edizione, McGraw-Hill (Cap. 23)
- Robert E. Tarjan, «Data Structures and Network Algorithms», SIAM, 1983