ALGORITMO DI PRIM

- Un altro algoritmo che serve a calcolare un albero di copertura minimo in un grafo semplice, connesso, pesato.

  anche detto non orientato

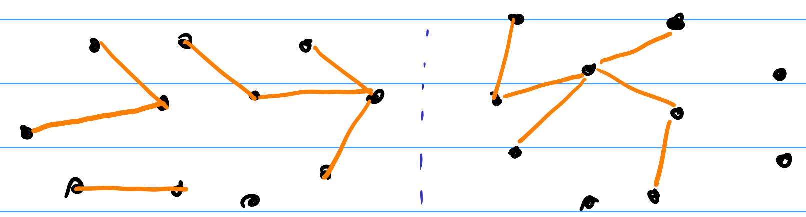
Dal TEOREMA DEL TAGLIO sappiamo che esiste un algoritmo greedy "generale" che trova un albero di copertura minimo (in inglese Minimum Spanning Tree, abbreviato MST).

Input : $G=(V,E,w)$ $\quad w: E \to \mathbb{R}$
Output $T \subseteq E$ $\quad T$ è un MST di $G=(V,E,w)$
$T \leftarrow \emptyset$
while $|T| < n-1$:
  Partiziona V in A e B, in modo che T non abbia archi $\{u,v\}$
  sia e = arco di costo minimo della forma $\{u,v\}$
  $T \leftarrow T \cup \{\{u,v\}\}$
return T

ESERCIZIO Dimostrate che l'algoritmo precedente calcola un albero di copertura minimo, dato $G=(V,E,w)$ non orientato, connesso e pesato.

Mentre Kruskal costruisce molti pezzi dell'albero unendoli due a due fino ad ottenere un MST, Prim invece costruisce un solo albero, agganciando mano mano vertici che ancora non sono connessi.

Kruskal | Prim

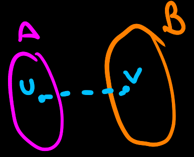


Dopo l'i-esimo passo dell'algoritmo di Prim esiste

- 1 componente connessa di $i+1$ vertici
- $n-i-1$ vertici isolati

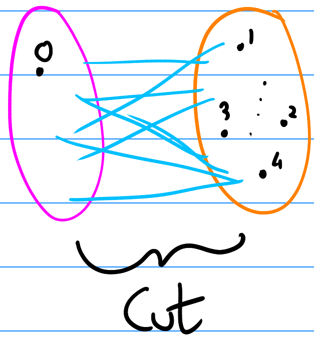
VEDERE ESEMPIO PRECALCOLATO


```python
def Prim(G,W):
    n     = len(G)
    A     = [0]     # A contiene i vertici già connessi
    B     = [1..n-1]# B contiene quelli da connettere
    T     = []      # Archi dell'albero

    Cut= ...        # Archi nel taglio

    while len(B)>0:
        u,v = min(Cut)
        T.append(u,v)
        A.append(v)
        B.remove(v)
        update(Cut)
    return T
```



$A = \{0\}$

$B = V(G)/\{0\}$



- La correttezza dell'algoritmo segue dal teorema del taglio e dall'esercizio precedente

- Il problema che rimane aperto é: COME GESTIRE GLI ARCHI NEL TAGLIO
  - dobbiamo sempre trovare il minimo
  - il taglio cambia ad ogni passo, e quindi dobbiamo aggiornare di volta in volta la struttura dati

Domanda: Qual é la complessitá dell'algoritmo se implementato banalmente? $O(|V|\cdot|E|)$?

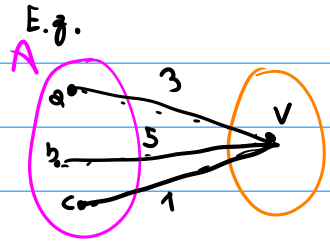
OSS • Dato A e un v in B, se ci sono $u, u' \in A$ tali che $w(u',v) > w(u,v)$ allora (u',v) non sará mai scelto come arco da aggiungere

- per ogni vertice v in B ci interessa tenere traccia di una sola connessione verso A: la migliore

Def $\text{dist}(A,v) = \min_{u \in A \cap \Gamma(v)} w(\{u,v\})$

$\text{closest}(A,v) = \arg\min_{u \in A \cap \Gamma(v)} w(\{u,v\})$

E.g.



$\text{closest}(A,v) = c$

$\text{dist}(A,v) = 1$

Se v non ha vicini in A, $\text{dist}(A,v) = +\infty$

$\text{closest}(A,v) = \text{NULL}$

Strategia In ogni momento dell'algoritmo ho un dato A e posso tenere traccia, con due array

$$\text{closest} = [\cdot \; \cdot \; \cdot \; \cdot \; \cdot \; \cdot \; \cdot \; \cdot] \leftarrow \text{closest}(A, v)$$

$$\text{dist} = [\cdot \; \cdot \; \cdot \; \cdot \; \cdot \; \cdot \; \cdot] \leftarrow \text{dist}(A, v)$$

v

Le posizioni per i vertici in A possono essere ignorate.

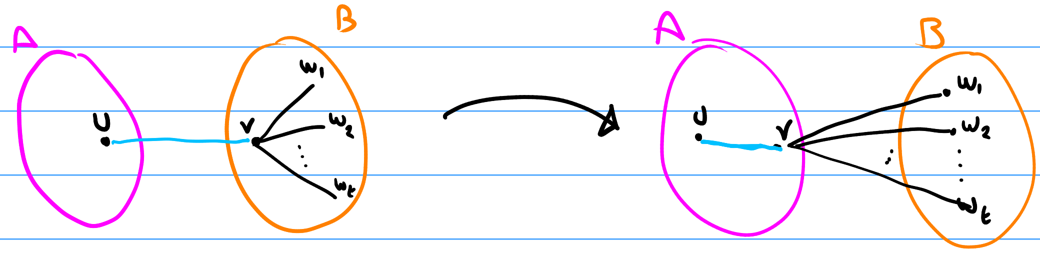
INIZIALIZZAZIONE

vertice in A

$$\text{closest}[v] = \begin{cases} 0 & \text{se } \{0, v\} \in E \\ \text{NULL} & \text{altrimenti} \end{cases} \qquad \text{dist}[v] = \begin{cases} w(\{0, v\}) & \text{se } \{0, v\} \in E \\ +\infty & \text{altrimenti} \end{cases}$$

AGGIORNAMENTO

Un vertice v viene aggiunto ad A



Spostare il vertice v in A fornisce ai vicini di v che sono in B una nuova via per connettersi ad A. In alcuni casi questa può essere migliore di quella nota, in altri può non esserlo

```
for w in Γ(v) ∩ B:
    if dist[w] > W({v,w}):
        closest[w] = v
        dist[w] = W({v,w})
```

conviene usare v per collegare w

OSS Ogni arco del grafo viene esplorato al massimo una volta durante la fase di aggiornamento di dist e closest.

Quindi $O(|V|+|E|)$ passi IN TOTALE sono sufficienti.

Con quello che abbiamo visto possiamo realizzare l'algoritmo in tempo $O(|V|^2)$

```
 1 from math import inf
 2
 3 def Prim(G,W):
 4     n     = len(G)
 5     A     = [0]      # A contiene i vertici già connessi
 6     B     = [1..n-1] # B contiene quelli da connettere
 7     T     = []       # Archi dell'albero
 8
 9     # Inizializzazione di closest/dist
10     closest = [None]*n
11     dist    = [inf]*n
12     for v in G[0]:
13         closest[v]=0
14         dist[v]=W[0,v]
15
16     while len(B)>0:
17         # Trova l'arco migliore
18         _,v = min([dist[t],t for t in B])
19
20         # Aggancia v ad A
21         u = closest[v]
22         T.append(u,v)
23         A.append(v)
24         B.remove(v)
25
26         # Aggiorna l'arco migliore
27         for w in G[v]:
28             if w in B and dist[w]>W[v,w]:
29                 closest[w]=v
30                 dist[w]=W[v,w]
31
32     return T
33
34
```

Inizializzazione di dist/closest

trova $v \in B$ per cui $dist[v]$ è minimo richiede $|B| = O(|V|)$ passi

aggiorna i valori di dist/closest dopo aver inserito v in A

In totale l'algoritmo costa $O(|V|^2)$ per cercare $|V|-1$ volte il minimo (vedi riga 18) più $O(|V|+|E|)$ passi per mantenere dist/closest

Cercare il minimo ad ogni passo è troppo costoso.

Per migliorare le prestazioni useremo CODE di PRIORITÁ che ci permetteranno di:

- trovare il minimo (riga 18) in tempo $\log|V|$ invece di $|V|$
- richiede tempo $\log|V|$ per fare gli aggiornamenti in linee 29-30

COMPLESSITÁ Totale $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$

obs se $|E| \gg \frac{|V|^2}{\log|V|}$ abbiamo che $|V|^2 < |E|\log|V|$

Esercizio 23.2-2

- Implementare una versione dell'algoritmo di Prim su grafi rappresentati da matrici di adiacenza
- Riuscite ad ottenere complessità $O(|V|^2)$?

Esercizio

Considerate gli algoritmi di Prim e Kruskal che abbiamo discusso

Questi algoritmi si aspettano dei grafi non orientati, pesati e CONNESSI. Che succede se il grafo non è connesso?
Modificate gli algoritmi in modo che questi sollevino un errore in caso di grafi non connessi.

Esercizio 23.2-8

... leggetelo sul libro di testo

Coda di priorità (Capitolo 6)

OSS
Il libro descrive strutture focalizzate sul MASSIMO invece che sul MINIMO. Cambia poco

- Una coda di priorità è una struttura dati che rappresenta un insieme $Q = \{e_1, \dots, e_t\}$ di elementi in un dominio $\mathbb{D}$ ognuno dotato di un valore/chiave di ordinamento/rank/costo

$$k : \mathbb{D} \to \mathbb{R}$$

e che supporta le seguenti operazioni:

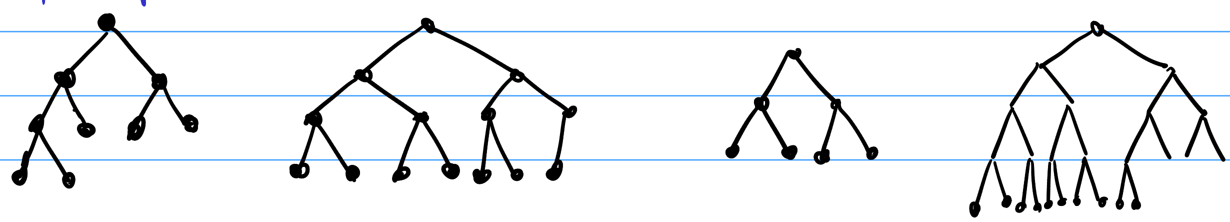
- INSERIMENTO di un elemento in Q
- MINIMO, trovare $e \in Q$ tale che $k(e) \leq k(e')$ $\forall e' \in Q$
- ESTRAZIONE MINIMO, estrae ed elimina da Q l'elemento di minore costo
- DECREMENTO del costo di un elemento in Q; cioè modifica del valore $k(e)$, per $e \in Q$, con un valore INFERIORE

Domanda Se si realizza una coda di priorità utilizzando un array ordinato, qual è la complessità delle operazioni richieste?

Un'opzione più credibile per realizzare una coda di priorità è l' Heap

Ripasso veloce sulla struttura dati Heap.

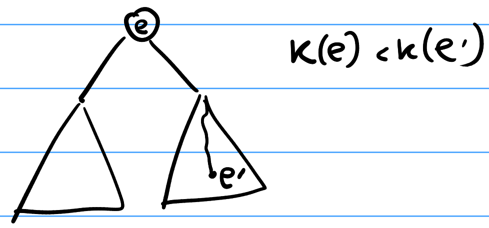
1. Gli oggetti sono inseriti in un ALBERO BINARIO COMPLETO (ovvero un albero binario nel quale tutti i livelli sono pieni tranne l'ultimo, dal quale posso mancare i nodi più a destra)



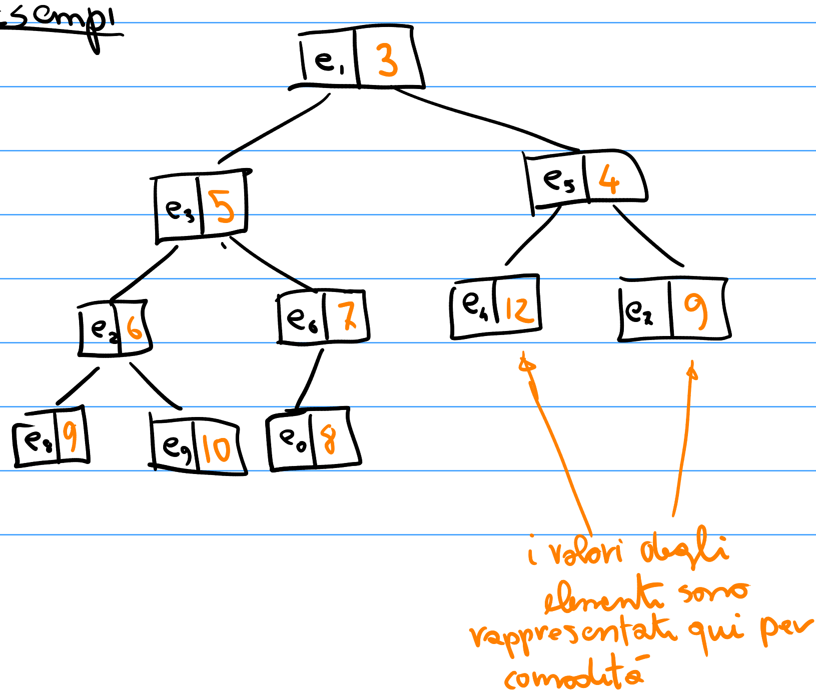
2. dato un qualunque nodo e nell'albero

$$k(e) < k(e')$$

per qualunque e' discendente da e

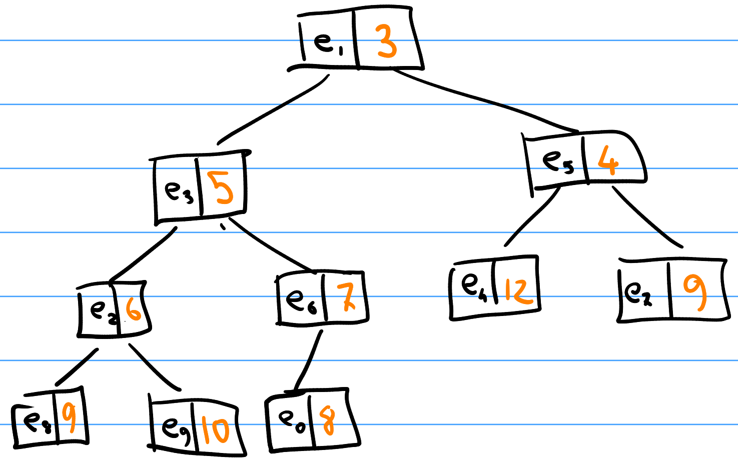


Esempi

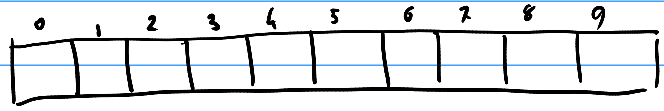


Un Heap con n elementi ha altezza $\log_2 n$.

Questo è essenziale per avere operazioni che costano $O(\log n)$

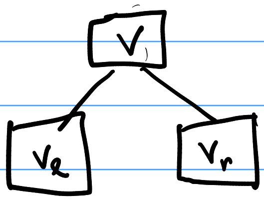


ESERCIZIO



Come sono disposti gli elementi dell'Heap, nell'array qui sopra?

Gli Heap spesso sono rappresentati in memoria in modo lineare in un array

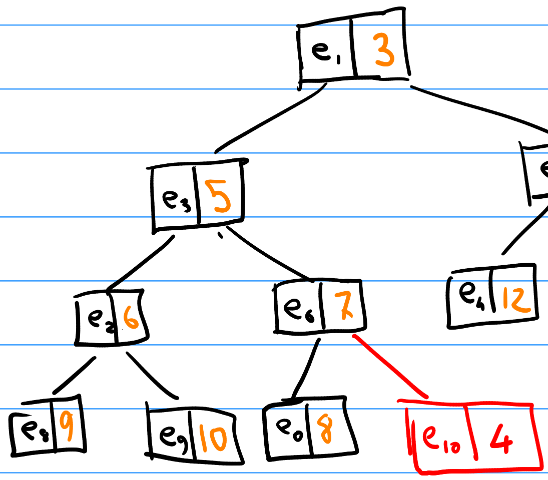


- $pos(radice) = 0$ — radice
- $pos(v_\ell) = 2 \cdot pos(v) + 1$ — figlio sinistro
- $pos(v_r) = 2 \cdot pos(v) + 2$ — figlio destro

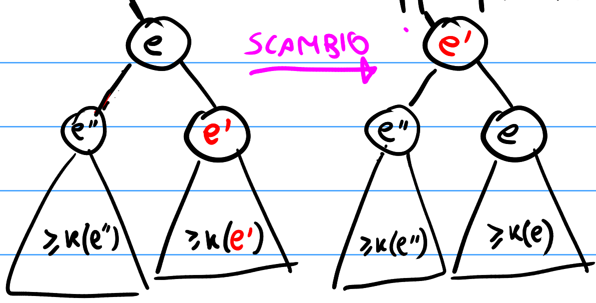
OSS Dato un elemento in pos. $i > 0$ nell'array, il genitore è in posizione $\left\lfloor \frac{i-1}{2} \right\rfloor$

OSS L'operazione MINIMO costa $O(1)$

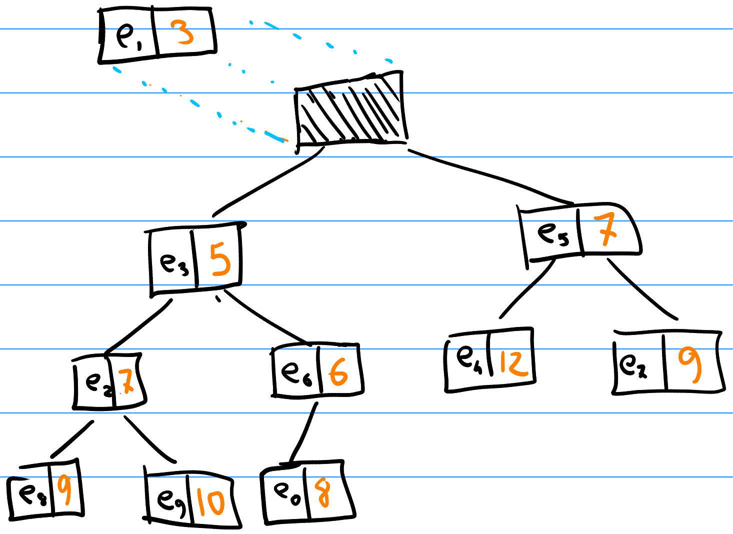
INSERIMENTO



- l'elemento viene inserito in fondo all'array (ovvero come foglio più a destra)
- poi l'elemento viene fatto "salire" come appropriato



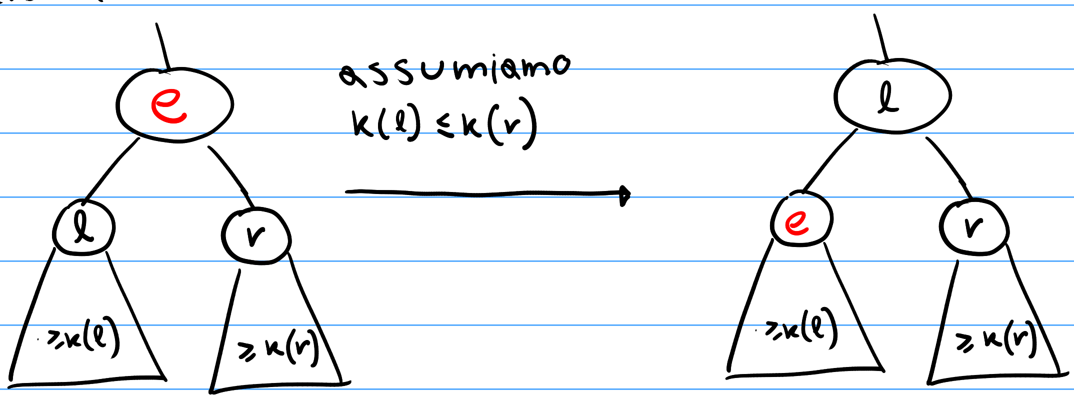
ESTRAZIONE DEL MINIMO



- L'elemento alla radice viene eliminato
  e.g. $e_1 \mid 3$
- L'ultimo elemento nell'array viene inserito in cima
  e.g. $e_0 \mid 8$
- Il nuovo elemento in cima viene "spinto verso il basso" per fargli raggiungere la posizione appropriata

- Se $k(e) \leq k(\ell)$ e $k(e) \leq k(r)$ non si muove nulla
- Se $k(e) > k(\ell)$ oppure $k(e) > k(r)$, viene messa al posto del più piccolo dei due.

Spinta verso il basso:

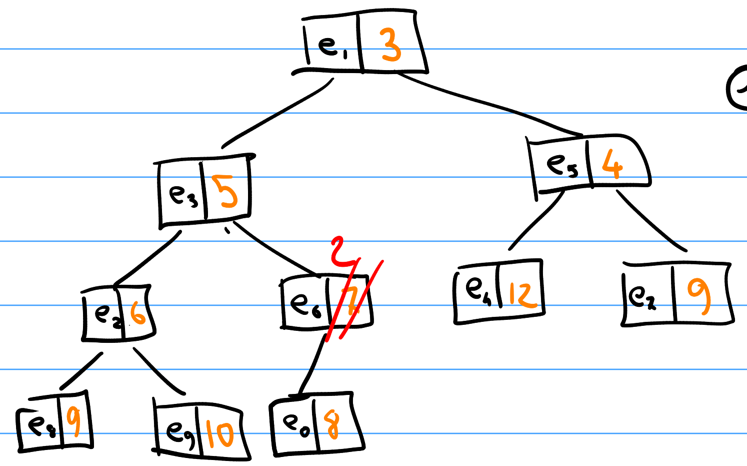


e poi si prosegue sul sottoalbero modificato

E l'operazione DECREMENTO?

- Dati un elemento x e un valore $k' < k(x)$, si setta $k(x) \leftarrow k'$ e la struttura deve essere modificata di conseguenza

Esempio $x = e_6$ $k' = 2$

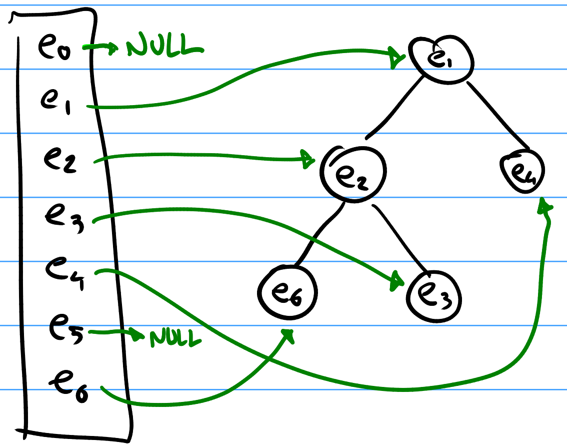


L'operazione é costituita da 3 passi

1. Trova la posizione nell'heap dell'elemento x
2. Setta $k(x) \leftarrow x'$
3. La posizione di x potrebbe dover salire

- La fase 2 dipende da come è memorizzata la funzione k
- La fase 3 è simile all'inserimento, ma partendo da una posizione interna dell'heap.

- Per la fase 1 c'è bisogno di tenere traccia delle posizioni di tutti gli elementi nell'heap



OSS Questi puntatori devono essere tenuti aggiornati man mano che gli elementi vengono spostati in giro nelle operazioni
ESTRAZIONE e INSERIMENTO

COMPLESSITÁ

MINIMO	$O(1)$
INSERIMENTO	$O(\log n)$
ESTRAZIONE	$O(\log n)$
DECREMENTO	$O(\log n)$

Versione "finale" dell'algoritmo di Prim

```python
from math import inf

def Prim(G,W):
    n  = len(G)
    T  = []      # Archi dell'albero

    Q  =...   # Coda di priorità

    # Inizializzazione di closest/dist
    closest = [None]*n
    dist    = [inf]*n
    for v in G[0]:
        closest[v]=0
        dist[v]=W[0,v]

    for i in range(1,n):
        Q.insert(v,dist[v])


    while len(Q)>0:
        v = Q.extract_min()

        u = closest[v]
        T.append(u,v)

        # Aggiorna l'arco migliore
        for w in G[v]:
            if w in Q and dist[w]>W[v,w]:
                closest[w]=v
                dist[w]=W[v,w]
                Q.decrease_key(w,dist[w])

    return T
```

Non serve tenere gli insiemi A e B
B : gli elementi in Q
$A = V(G) \setminus B$

il vertice per cui dist[v] è minore

L'array che mantiene le posizioni nella coda (vedi pag 9) può essere anche usato per verificare se un elemento è contenuto