

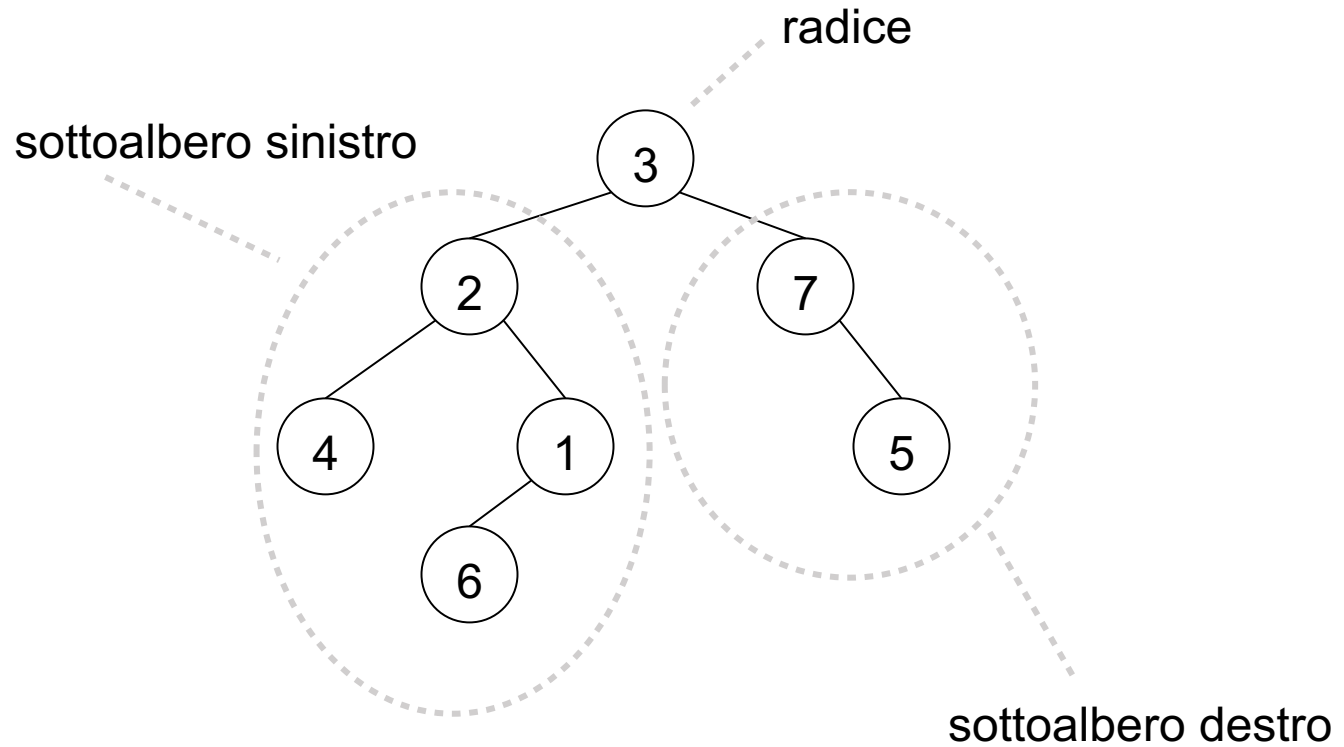
Strutture Dati

Parte II: alberi e grafi

Alberi binari

- Un **albero binario** è fatto di 3 elementi:
 - un nodo *radice*;
 - un albero binario che è il *sottoalbero sinistro*;
 - un albero binario che è il *sottoalbero destro*
- È una definizione ricorsiva
 - un sottoalbero può essere *vuoto (NIL)*
 - Ad ogni nodo dell'albero associamo un oggetto con una *chiave*

Esempio di albero binario

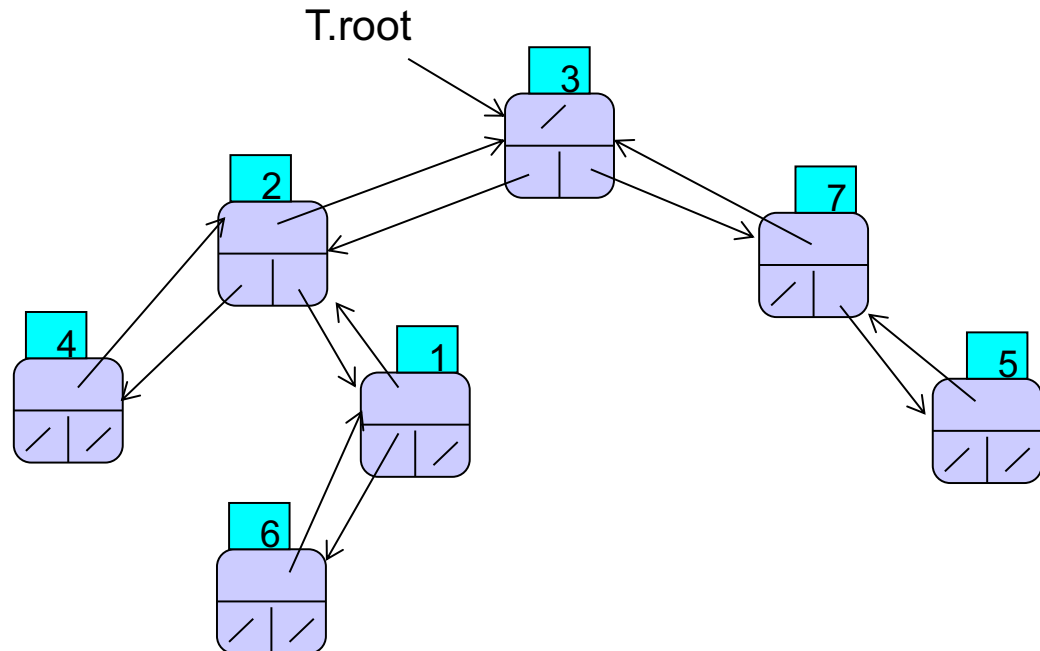


Rappresentazione di alberi binari

- Tipicamente si rappresentano alberi binari mediante strutture dati concatenate
 - abbiamo però anche visto la rappresentazione mediante array
- Ogni nodo dell'albero è rappresentato da un oggetto che ha i seguenti attributi
 - *key*, la chiave del nodo (che per noi ne rappresenta il contenuto)
 - tipicamente ci sono anche i dati satelliti
 - *p*, che è il (puntatore al) nodo padre
 - *left*, che è il (puntatore al) sottoalbero sinistro
 - *left* è la radice del sottoalbero sinistro
 - *right*, che è il (puntatore al) sottoalbero destro
 - è la radice del sottoalbero destro
- Ogni albero *T* ha un attributo, *T.root*, che è il puntatore alla radice dell'albero

Casi particolari

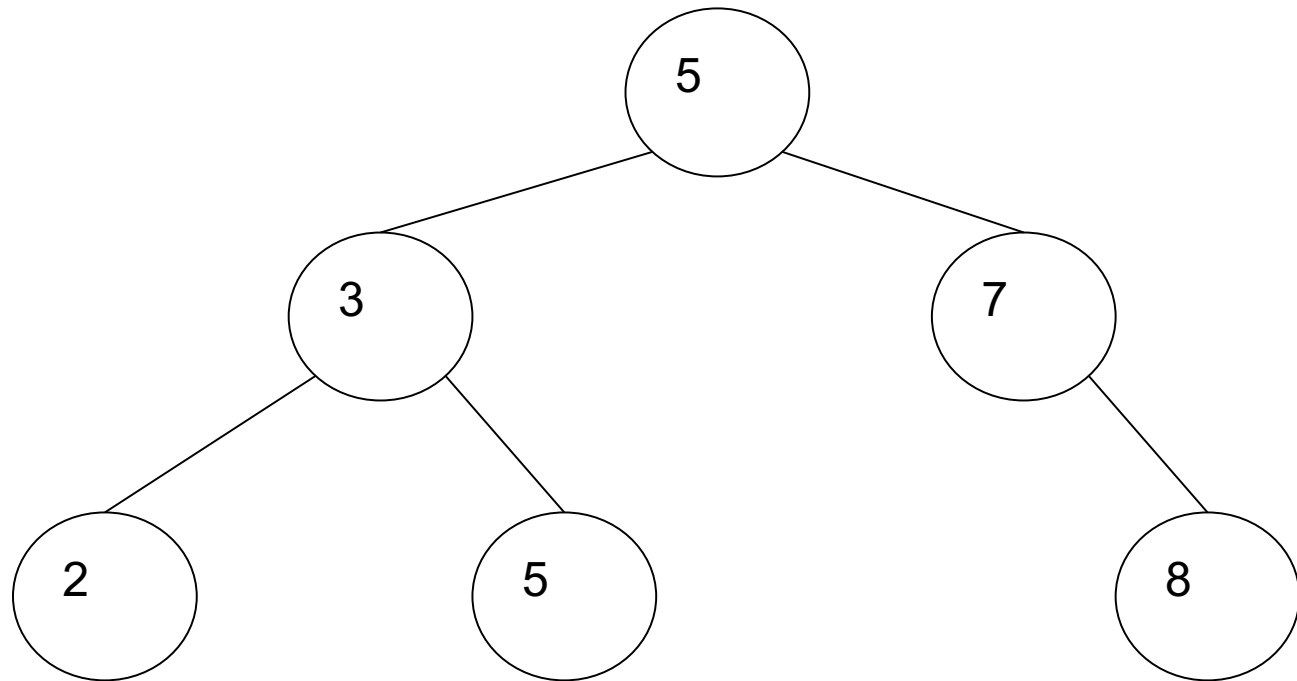
- Se il sottoalbero sinistro (destro) di un nodo x è vuoto, allora
 $x.left = NIL$ ($x.right = NIL$)
- $x.p = NIL$ se e solo se x è la radice (cioè $x = T.root$)
- Per esempio:



Alberi binari di ricerca (Binary Search Trees)

- Un albero binario di ricerca (Binary Search Tree, BST) è un albero binario che soddisfa la seguente proprietà:
 - per tutti i nodi x del BST, se l è un nodo nel sottoalbero sinistro, allora $l.key \leq x.key$; se r è un nodo del sottoalbero destro, allora $x.key \leq r.key$
 - tutti i nodi l del sottoalbero sinistro di un nodo x sono tali che, per tutti i nodi r nel sottoalbero destro di x vale $l.key \leq r.key$

Esempio di BST



Attraversamento simmetrico di un BST

- Operazione tipica: attraversamento (walk through)
- Lo scopo è di produrre gli elementi dell'albero
 - cioè le chiavi ad essi associate
- Attraversamento simmetrico (**inorder tree walk**)
 - prima si visita il sottoalbero sinistro e si restituiscono i suoi nodi
 - quindi si restituisce la radice
 - quindi si visita il sottoalbero destro e si restituiscono i suoi nodi
- Come spesso accade con gli alberi (struttura dati inerentemente ricorsiva), l'attraversamento simmetrico è un algoritmo ricorsivo
 - con l'attraversamento simmetrico gli elementi di un albero sono restituiti *ordinati*
 - per esempio, l'attraversamento simmetrico sull'albero precedente produce le chiavi seguenti: 2, 3, 5, 5, 7, 8

Algoritmi di attraversamento

```
INORDER-TREE-WALK(x)  
1  if x ≠ NIL  
2    INORDER-TREE-WALK(x.left)  
3    print x.key  
4    INORDER-TREE-WALK(x.right)
```

- Se T è un BST, INORDER-TREE-WALK($T.root$) stampa tutti gli elementi di T in ordine non decrescente
- Se n è il numero di nodi nel (sotto)albero, il tempo di esecuzione per INORDER-TREE-WALK è $\Theta(n)$
 - se l'albero è vuoto, è eseguito in tempo costante c
 - se l'albero ha 2 sottoalberi di dimensioni k e $n-k-1$, $T(n)$ è dato dalla ricorrenza $T(n) = T(k) + T(n-k-1) + d$, che ha soluzione $(c+d)n+c$
 - lo si può vedere sostituendo la soluzione nell'equazione

Altre strategie di attraversamento

- *Anticipato* (**preorder tree walk**)
 - in preorder, la radice è restituita prima dei sottoalberi
- *Posticipato* (**postorder tree walk**)
 - in postorder, la radice è restituita dopo i sottoalberi
- **Esercizio: scrivere lo pseudocodice per
PREORDER-TREE-WALK e POSTORDER-TREE-WALK**

Operazioni sui BST

- Sfruttiamo la proprietà di essere un BST per realizzare la *ricerca*:
 - confronta la chiave della radice con quella cercata
 - se sono uguali, l'elemento è quello cercato
 - se la chiave della radice è più grande, cerca nel sottoalbero sinistro
 - se la chiave della radice è più piccola, cerca nel sottoalbero destro

TREE-SEARCH

```
TREE-SEARCH(x, k)
```

```
1 if x = NIL or k = x.key
```

```
2   return x
```

```
3 if k < x.key
```

```
4   return TREE-SEARCH(x.left, k)
```

```
5 else return TREE-SEARCH(x.right, k)
```

- Il tempo di esecuzione è $O(h)$, con h l'altezza dell'albero

Minimo e massimo elemento

- L'elemento *minimo* (risp. *massimo*) in un BST è quello che è più a sinistra (risp. destra)
- Sfruttiamo questa proprietà per definire il seguente algoritmo, che semplicemente “scende” nell'albero
 - MINIMUM scende a sinistra, mentre MAXIMUM scende a destra
 - gli algoritmi restituiscono l'oggetto nell'albero con la chiave minima, non la chiave stessa

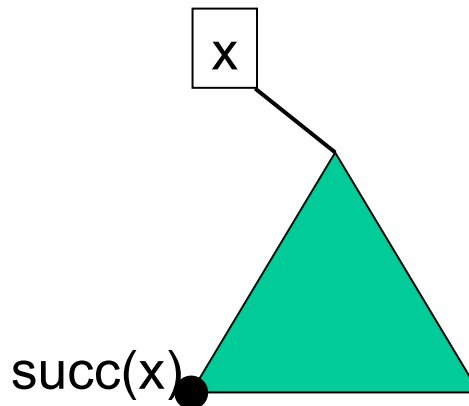
```
TREE-MINIMUM(x)
1 while x.left ≠ NIL
2   x := x.left
3 return x
```

```
TREE-MAXIMUM(x)
1 while x.right ≠ NIL
2   x := x.right
3 return x
```

Entrambi gli algoritmi hanno tempo di esecuzione che è $O(h)$, con h l'altezza dell'albero

Operazioni sui BST: successore e predecessore

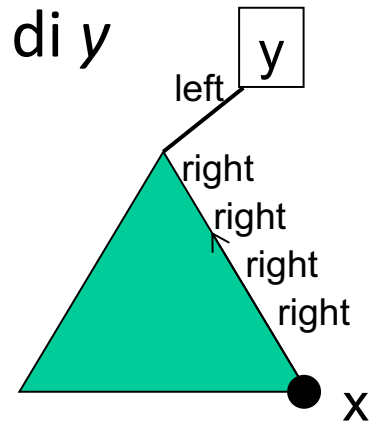
- Il *successore* (risp. *predecessore*) di un oggetto x in un BST è l'elemento y del BST tale che $y.key$ è la più piccola (risp. più grande) tra le chiavi che sono più grandi (risp. piccole) di $x.key$
 - di fatto, se il sottoalbero destro di un oggetto x dell'albero non è vuoto, il successore di x è l'elemento più piccolo (cioè il minimo) del sottoalbero destro di x



Operazioni sui BST: TREE-SUCCESSOR

- se il sottoalbero destro di x è vuoto, il successore di x è il primo elemento y che si incontra risalendo nell'albero da x tale che x è nel sottoalbero sinistro di y

```
TREE-SUCCESSOR( $x$ )
1 if  $x.right \neq \text{NIL}$ 
2   return TREE-MINIMUM( $x.right$ )
3  $y := x.p$ 
4 while  $y \neq \text{NIL}$  and  $x = y.right$ 
5    $x := y$ 
6    $y := y.p$ 
7 return  $y$ 
```

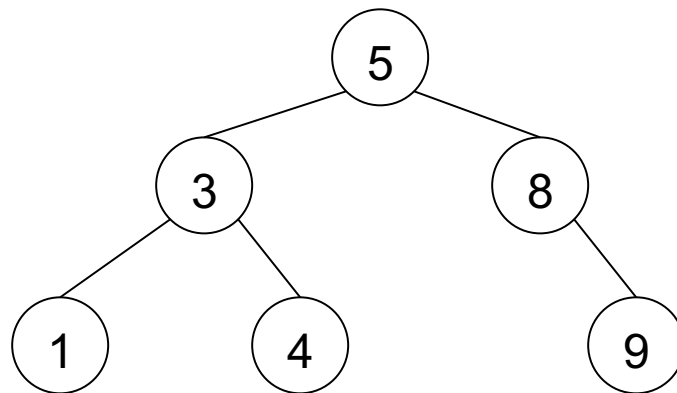


informalmente: per trovare y mi basta salire dai "right" fino a quando risalgo un "left"

- Il tempo di esecuzione per TREE-SUCCESSOR è $O(h)$
- Esercizio: scrivere l'algoritmo TREE-PREDECESSOR e darne la complessità

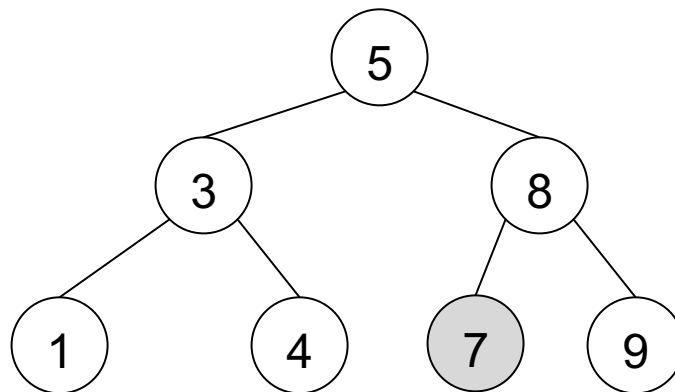
Operazioni sui BST: inserimento

- Idea di base per l'inserimento: scendere nell'albero fino a che non si raggiunge il posto in cui il nuovo elemento deve essere inserito, ed aggiungere questo come foglia
- Supponiamo, per esempio, di volere inserire un nodo con chiave 7 nell'albero seguente:



Operazioni sui BST: inserimento

- Eseguiamo i seguenti passi:
 - confrontiamo 5 con 7:
 - 7 deve essere aggiunto al sottoalbero destro di 5
 - confrontiamo 8 con 7:
 - 7 deve essere aggiunto al sottoalbero sinistro di 8
 - il sottoalbero sinistro di 8 è vuoto, quindi aggiungiamo 7 come sottoalbero sinistro di 8



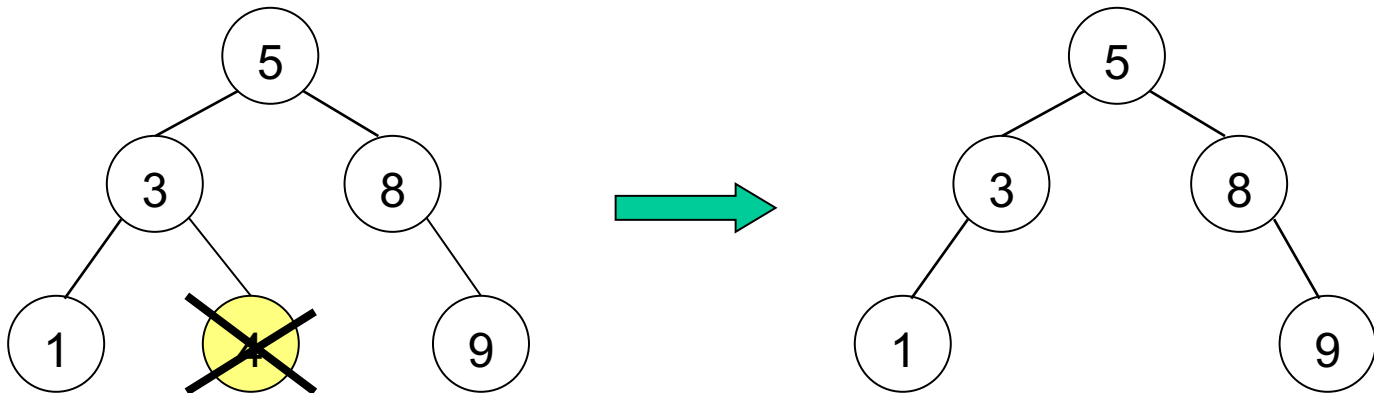
Operazioni sui BST: TREE-INSERT

```
TREE-INSERT(T, z)
1  y := NIL
2  x := T.root
3  while x ≠ NIL
4      y := x
5      if z.key < x.key
6          x := x.left
7      else x := x.right
8  z.p := y
9  if y = NIL
10     T.root := z
        //l'albero T e' vuoto
11 elseif z.key < y.key
12     y.left := z
13 else y.right := z
```

- Si noti che inseriamo un oggetto, *z*, che assumiamo sia stato inizializzato con la chiave desiderata
- Il tempo di esecuzione di TREE-INSERT è $O(h)$
 - infatti, scendiamo nell'albero nel ciclo **while** (che al massimo richiede tante ripetizioni quanta è l'altezza dell'albero), e il resto (linee 8-13) si fa in tempo costante

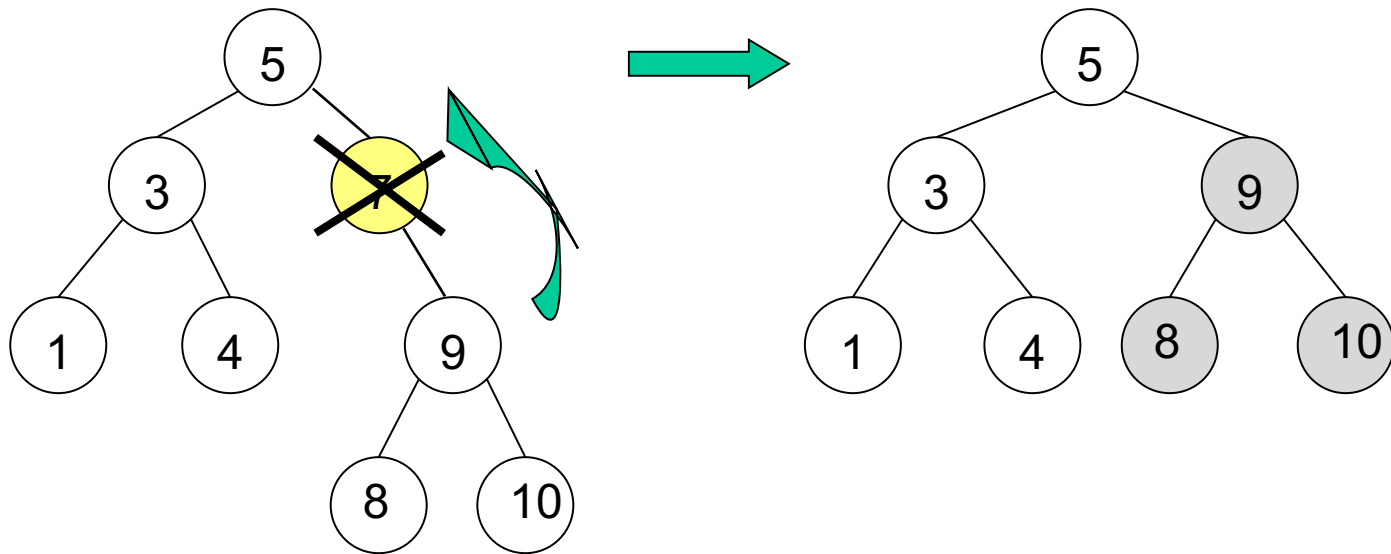
Operazioni sui BST: cancellazione

- Quando cancelliamo un oggetto z da un albero, abbiamo 3 possibili casi. Il nodo z da cancellare:
 - non ha sottoalberi
 - ha 1 sottoalbero
 - ha 2 sottoalberi
- Il primo caso è quello più facile: basta mettere a *NIL* il puntatore del padre di z che puntava a z :



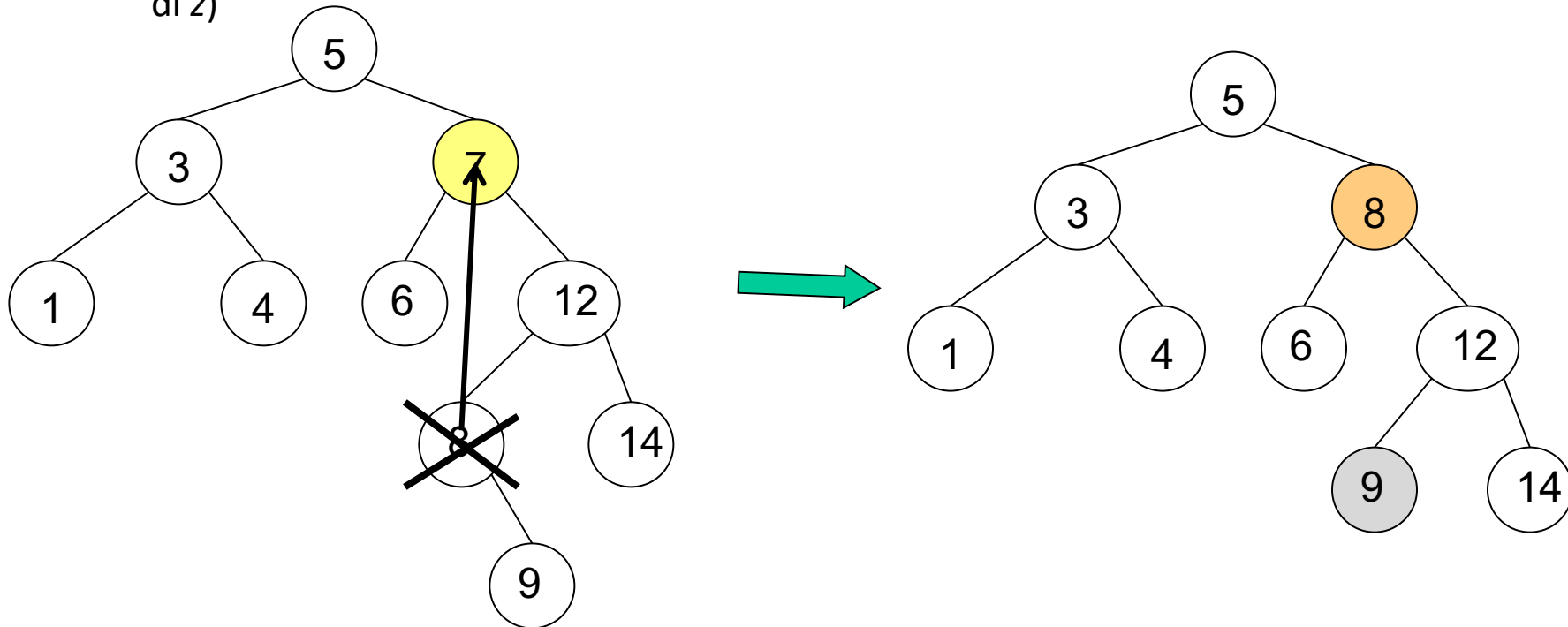
Cancellazione con un sottoalbero

- Nel secondo caso dobbiamo spostare l'intero sottoalbero di z all'insù di un livello



Cancellazione con due sottoalberi

- Nel caso 3 dobbiamo trovare il successore del nodo da cancellare z, copiare la chiave del successore in z, quindi cancellare il successore
 - cancellare il successore potrebbe richiedere di spostare un (il) sottoalbero del successore un livello su
 - in questo caso l'oggetto originario z non è cancellato, ma il suo attributo *key* viene modificato (l'oggetto effettivamente cancellato è quello con il successore di z)



Operazioni sui BST: TREE-DELETE

```
TREE-DELETE(T, z)
1  if z.left = NIL or z.right = NIL
2    y := z
3  else y := TREE-SUCCESSOR(z)
4  if y.left ≠ NIL
5    x := y.left
6  else x := y.right
7  if x ≠ NIL
8    x.p := y.p
9  if y.p = NIL
10   T.root := x
11 elsif y = y.p.left
12   y.p.left := x
13 else y.p.right := x
14 if y ≠ z
15   z.key := y.key
16 return y
```

Nota:

y è il nodo da eliminare

x è quello con cui lo sostituiamo

Funzionamento di TREE-DELETE

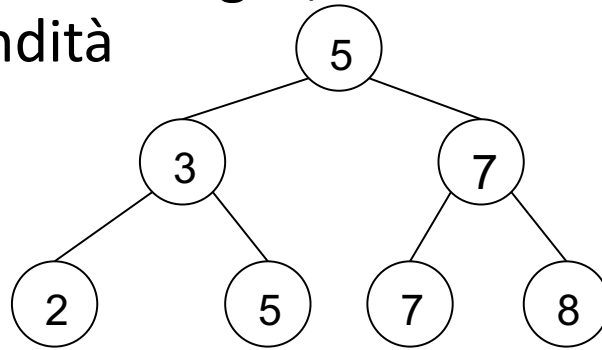
- In TREE-DELETE, y è il nodo effettivamente da cancellare
- Se z ha non più di un sottoalbero, allora il nodo y da cancellare è z stesso; altrimenti (se z ha entrambi i sottoalberi) è il suo successore (linee 1-3)
 - Si noti che y non può avere più di un sottoalbero
- Nelle linee 4-6, ad x viene assegnata la radice del sottoalbero di y se y ne ha uno, NIL se y non ha sottoalberi
- Le linee 7-13 sostituiscono y con il suo sottoalbero (che ha x come radice)
- Nelle linee 14-15, se z ha 2 sottoalberi (che corrisponde al caso in cui il nodo y da cancellare è il successore di z , non z stesso), la chiave di z è sostituita con quella del suo successore y

Analisi di complessità di TREE-DELETE

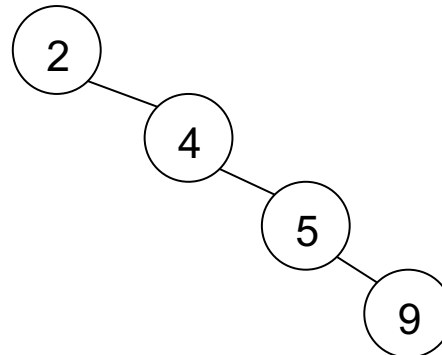
- Il tempo di esecuzione per TREE-DELETE è $O(h)$
 - TREE-SUCCESSOR è $O(h)$, il resto è fatto in tempo costante
- Tutte le operazioni sui BST (SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, INSERT, DELETE) hanno tempo di esecuzione che è $O(h)$
 - cioè, alla peggio richiedo di scendere nell'albero
- Quindi... quanto vale l'altezza di un BST (rispetto al numero dei suoi nodi)?

Altezza di un BST

- Per un albero *completo*, $h = \Theta(\log(n))$
 - un albero è completo se e solo se, per ogni nodo x , o x ha 2 figli, o x è una foglia, e tutte le foglie hanno la stessa profondità



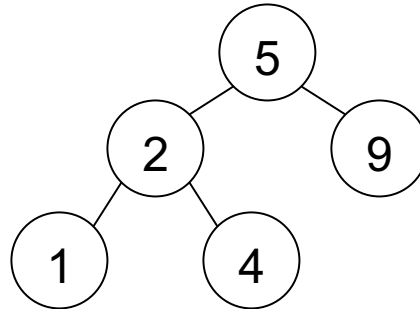
- Nel caso pessimo (tutti i nodi “in linea”), $h = \Theta(n)$



Alberi bilanciati

- Tuttavia, un BST non deve per forza essere completo per avere altezza h tale che $h = \Theta(\log(n))$
 - abbiamo per esempio visto che questa proprietà vale anche per alberi *quasi completi*
- Abbiamo che $h = \Theta(\log(n))$ anche per un albero *bilanciato*
 - informalmente, diciamo che un albero è bilanciato se e solo se non ci sono 2 foglie nell'albero tali che una è “molto più lontana” dalla radice dell'altra (se si trovano a profondità molto diverse)
 - ci potrebbero essere diverse nozioni di "molto più lontano"
 - una possibile definizione di albero bilanciato (Adelson-Velskii e Landis) è la seguente: un albero è *bilanciato* se e solo se, per ogni nodo x dell'albero, le altezze dei 2 sottoalberi di x differiscono al massimo di 1

Esempio di albero bilanciato



- Ci sono diverse tecniche per mantenere un albero bilanciato:
 - alberi *rosso-neri* (red-black)
 - alberi AVL (di Adelson-Velskii e Landis)
 - etc.
- Inoltre, si può dimostrare che l'altezza attesa di un albero costruito inserendo le chiavi in ordine casuale con distribuzione uniforme è $O(\log(n))$

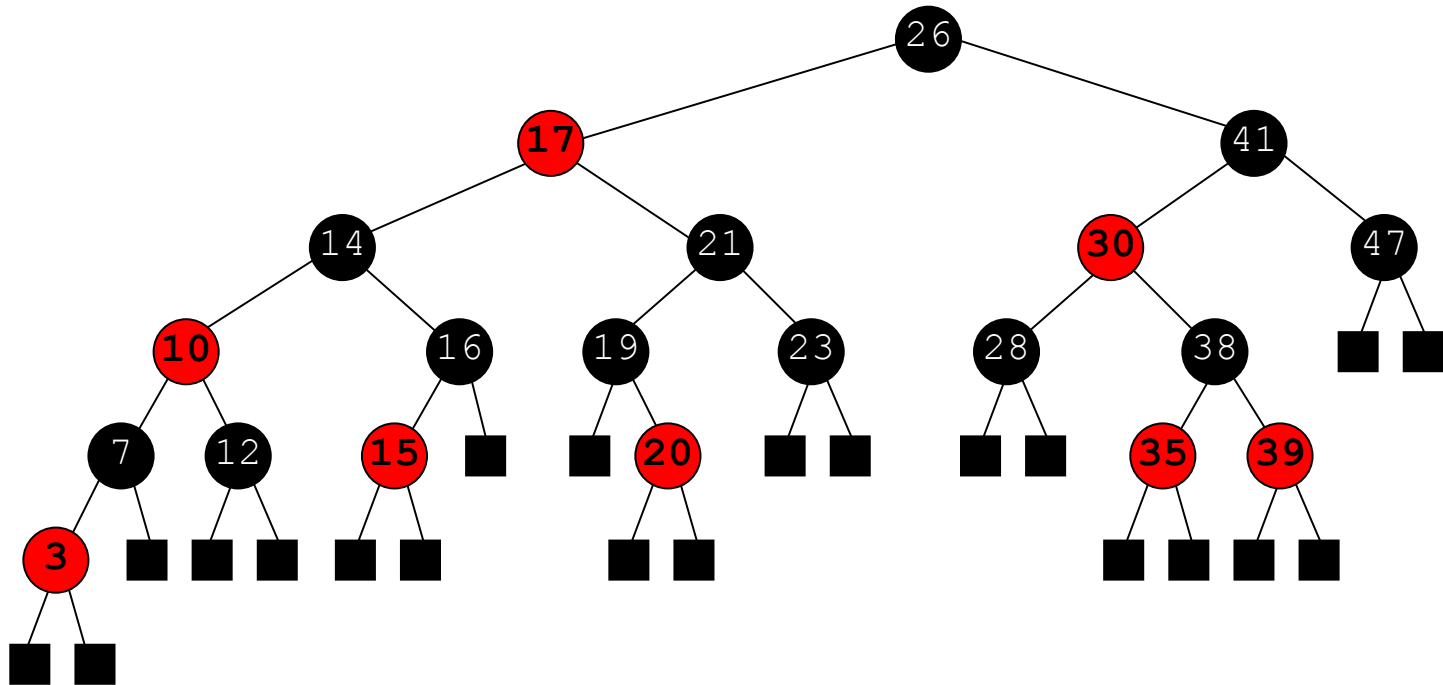
Alberi rosso-neri (red-black)

- Gli alberi rosso-neri (RB) sono BST “abbastanza” bilanciati, cioè l'altezza dell'albero $h = O(\log(n))$ ed è possibile realizzare tutte le operazioni più importanti in tempo $O(\log(n))$
- Negli alberi RB non si ha mai che un ramo dell'albero sia lungo più del doppio di un altro ramo
 - è una nozione di “bilanciamento” diversa da quella degli alberi AVL, ma dà comunque $h = O(\log(n))$
- Idea alla base degli alberi RB:
 - ogni nodo ha un colore, che può essere solo rosso o nero
 - i colori sono distribuiti nell'albero in modo da garantire che nessun ramo dell'albero sia 2 volte più lungo di un altro

Definizione di albero RB

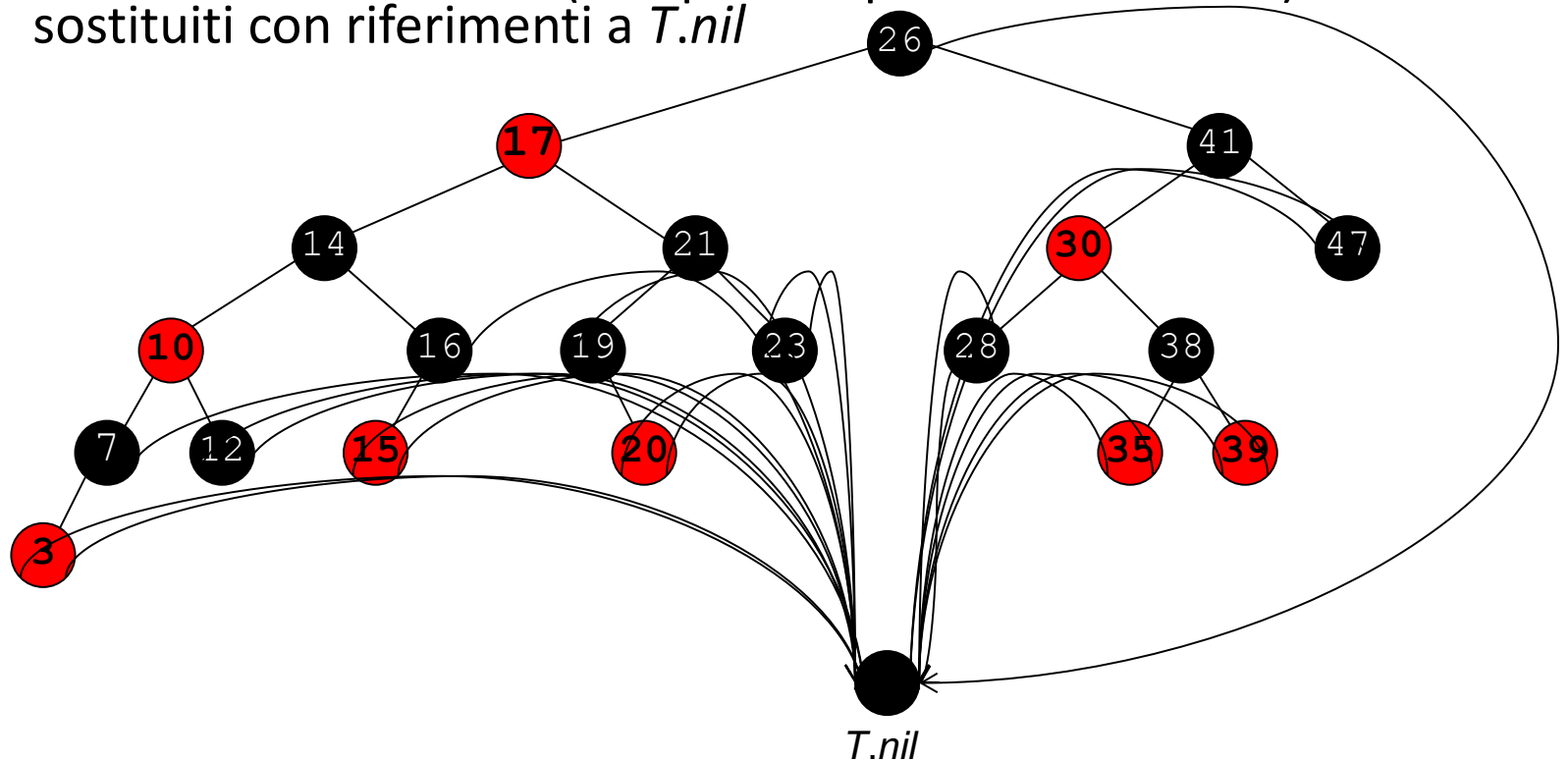
- Ogni nodo di un albero RB ha 5 attributi: *key*, *left*, *right*, *p*, e *color*
 - convenzione: le foglie sono i nodi *NIL*, tutti i nodi non *NIL* (che hanno quindi una chiave associata) sono nodi interni
- Un BST è un albero RB se soddisfa le seguenti 5 proprietà:
 1. ogni nodo è o rosso o nero
 2. la radice è nera
 3. le foglie (*NIL*) sono tutte nere
 4. i figli di un nodo rosso sono entrambi neri
 5. per ogni nodo x tutti i cammini da x alle foglie sue discendenti contengono lo stesso numero $bh(x)$ di nodi neri
 - $bh(x)$ è la *altezza nera* (black height) del nodo x
 - Il nodo x non è contato in $bh(x)$ anche se nero.

Esempi di alberi RB



Rappresentazione delle foglie

- Per comodità, per rappresentare tutte le foglie *NIL*, si usa un unico nodo sentinella *T.nil*
 - è un nodo particolare accessibile come attributo dell'albero *T*
 - tutti i riferimenti a *NIL* (compreso il padre della radice) sono sostituiti con riferimenti a *T.nil*

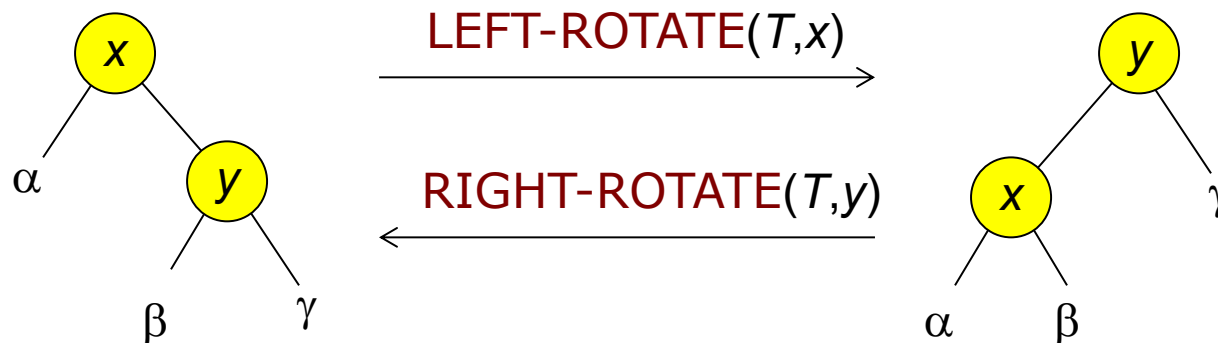


Proprietà degli alberi RB

- Un albero RB con n nodi interni (n nodi con chiavi, per la convenzione usata) ha altezza $h \leq 2 \log_2(n+1)$
 - il numero di nodi interni di un (sotto)albero con radice x è $\geq 2^{bh(x)} - 1$, (si dimostra per induzione sull'altezza di x)
 - per la proprietà 4, almeno metà dei nodi dalla radice x (esclusa) ad una foglia sono neri, quindi $bh(x) \geq h/2$, e $n \geq 2^{h/2} - 1$, da cui discende che $h \leq 2 \log_2(n+1)$
- Come conseguenza di questa proprietà, SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR richiedono tempo $O(\log(n))$ se applicate ad un albero RB con n nodi
 - queste operazioni non modificano l'albero, che viene semplicemente trattato come un BST
 - il loro pseudocodice è come quello visto in precedenza

INSERT e DELETE negli alberi RB

- INSERT e DELETE si possono anch'esse fare con complessità $O(\log(n))$, ma devono essere modificate rispetto a quelle viste prima
 - devono essere tali da mantenere le 5 proprietà degli alberi RB
- Il meccanismo fondamentale per realizzare INSERT e DELETE è quello delle rotazioni: verso sinistra (LEFT-ROTATE) o verso destra (RIGHT-ROTATE)

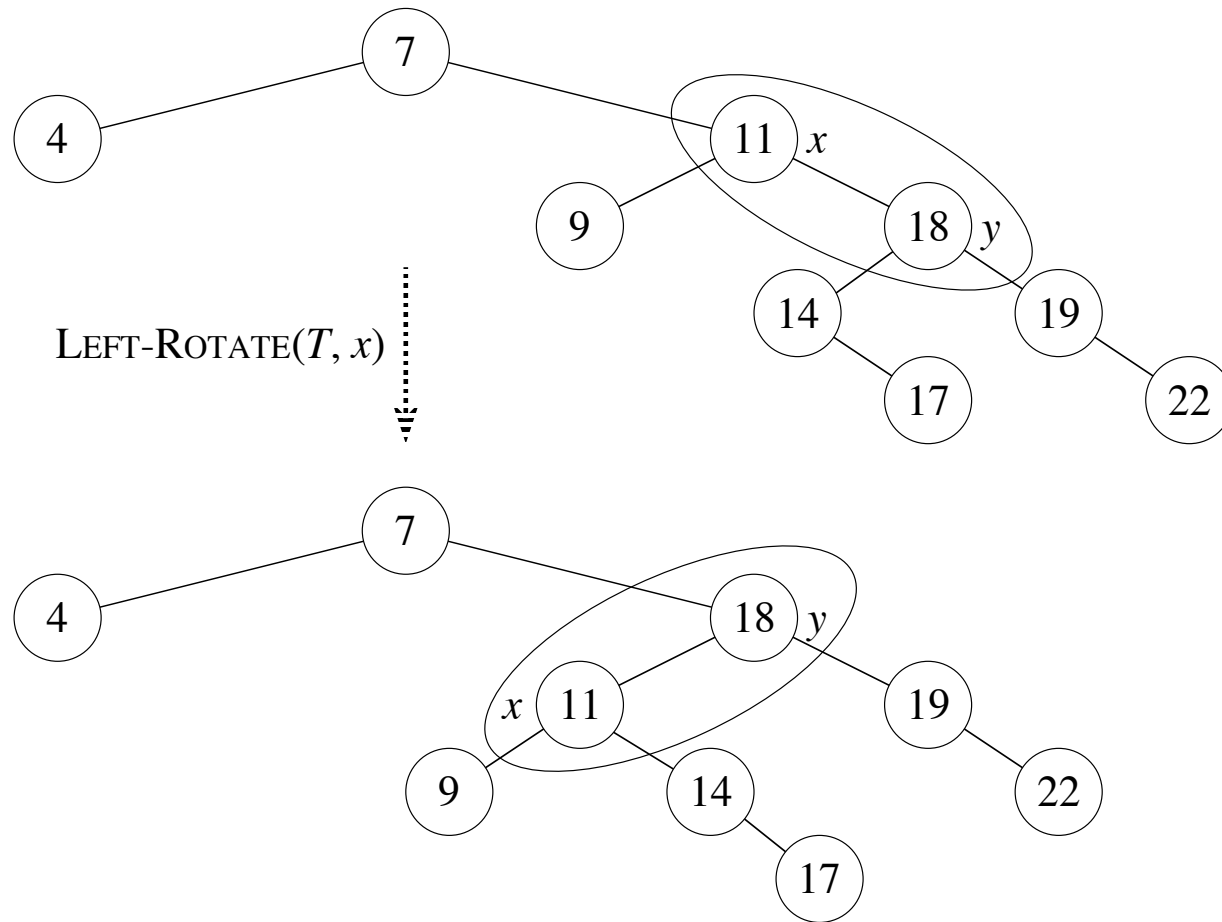


Pseudocodice per rotazioni

```
LEFT-ROTATE (T, x)
1  y := x.right
2  x.right := y.left    //il sottoalbero sinistro di y
                        //diventa quello destro di x
3  if y.left ≠ T.nil
4    y.left.p := x
5  y.p := x.p          //attacca il padre di x a y
6  if x.p = T.nil
7    T.root := y
8  elsif x = x.p.left
9    x.p.left := y
10 else x.p.right := y
11 y.left := x         //mette x a sinistra di y
12 x.p := y
```

- Una rotazione su un BST mantiene la proprietà di essere BST
- Esercizio per casa: scrivere lo pseudocodice per RIGHT-ROTATE

Esempio: mantenimento proprietà BST con la rotazione



Inserimento negli alberi RB

- L'inserimento è fatto in modo analogo a quello dei BST, ma alla fine occorre ristabilire le proprietà degli alberi RB se queste sono state violate
 - per ristabilire le proprietà si usa un algoritmo RB-INSERT-FIXUP (che vedremo dopo)
- Uguale a TREE-INSERT, salvo che per l'uso di *T.nil* al posto di *NIL* e l'aggiunta delle righe 14-17:

RB-INSERT

```
RB-INSERT(T, z)
1   y := T.nil           // y padre del nodo considerato
2   x := T.root          // nodo considerato
3   while x ≠ T.nil
4       y := x
5       if z.key < x.key
6           x := x.left
7       else x := x.right
8   z.p := y
9   if y = T.nil
10      T.root := z      // l'albero T e' vuoto
11  elsif z.key < y.key
12      y.left := z
13  else y.right := z
14  z.left := T.nil
15  z.right := T.nil
16  z.color := RED
17  RB-INSERT-FIXUP(T, z)
```

RB-INSERT-FIXUP

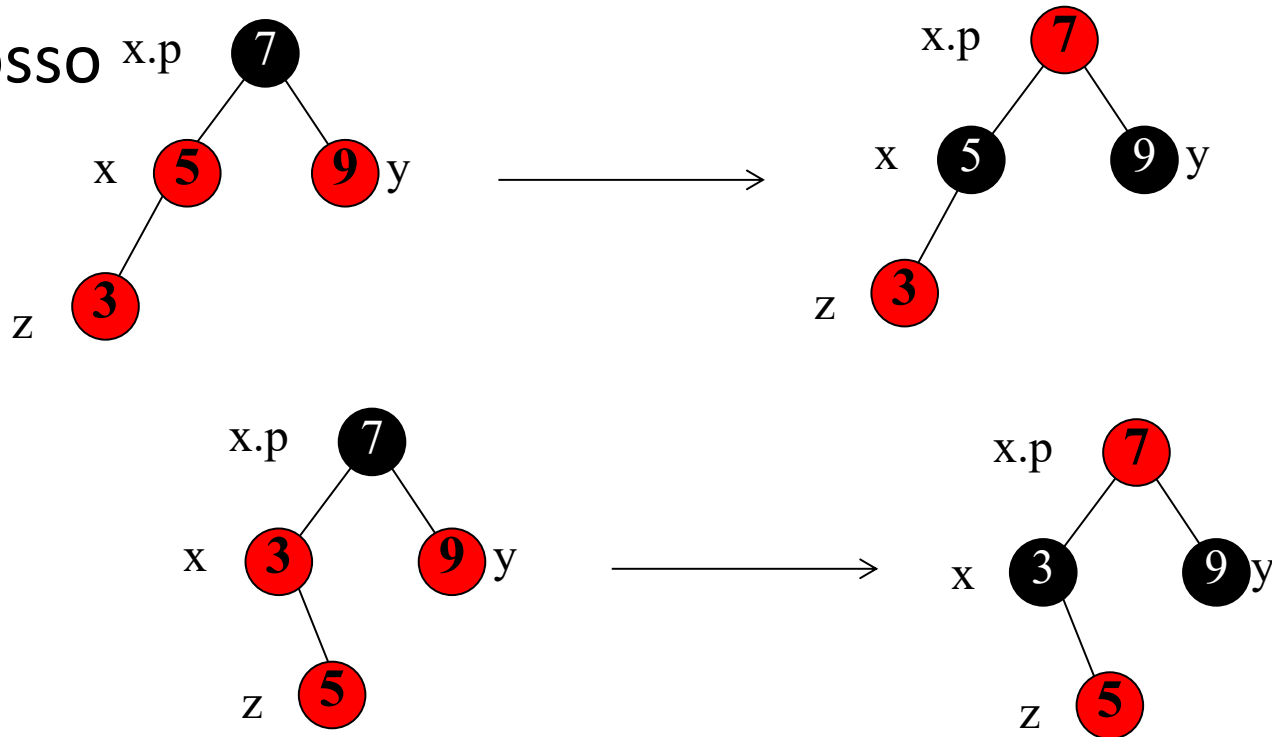
```
RB-INSERT-FIXUP(T, z)
1  if z = T.root
2      T.root.color = BLACK
3  else x := z.p           // x e' il padre di z
4      if x.color = RED
5          if x = x.p.left   // se x e' figlio sin.
6              y := x.p.right // y e' fratello di x
7              if y.color = RED
8                  x.color := BLACK        // Caso 1
9                  y.color := BLACK        // Caso 1
10                 x.p.color := RED         // Caso 1
11                 RB-INSERT-FIXUP(T, x.p)  // Caso 1
12             else if z = x.right
13                 z := x                  // Caso 2
14                 LEFT-ROTATE(T, z)       // Caso 2
15                 x := z.p                // Caso 2
16                 x.color := BLACK        // Caso 3
17                 x.p.color := RED        // Caso 3
18                 RIGHT-ROTATE(T, x.p)    // Caso 3
19             else (come 6-18, scambiando "right" e "left")
```

Note su RB-INSERT-FIXUP

- RB-INSERT-FIXUP è invocato sempre su un nodo z tale che $z.color = RED$
 - Per questo motivo, se la condizione alla linea 4 è non verificata (cioè se il padre di z è di colore nero) non ci sono ulteriori modifiche da fare
 - Ora vediamo come funziona

Funzionamento di RB-INSERT

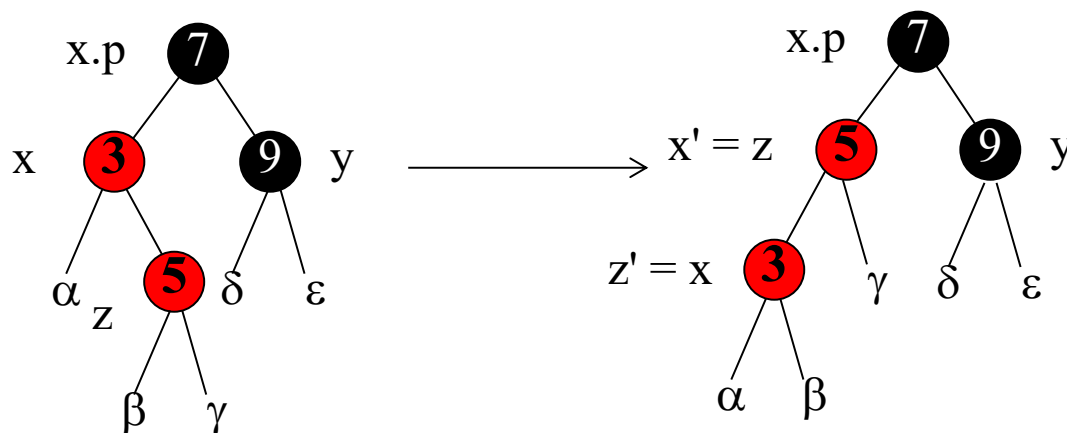
- Caso 1: y rosso



- quindi ripeto la procedura su x.p, in quanto il padre di x.p potrebbe essere di colore rosso, nel qual caso la proprietà 4 degli alberi RB non sarebbe (ancora) verificata

Funzionamento di RB-INSERT

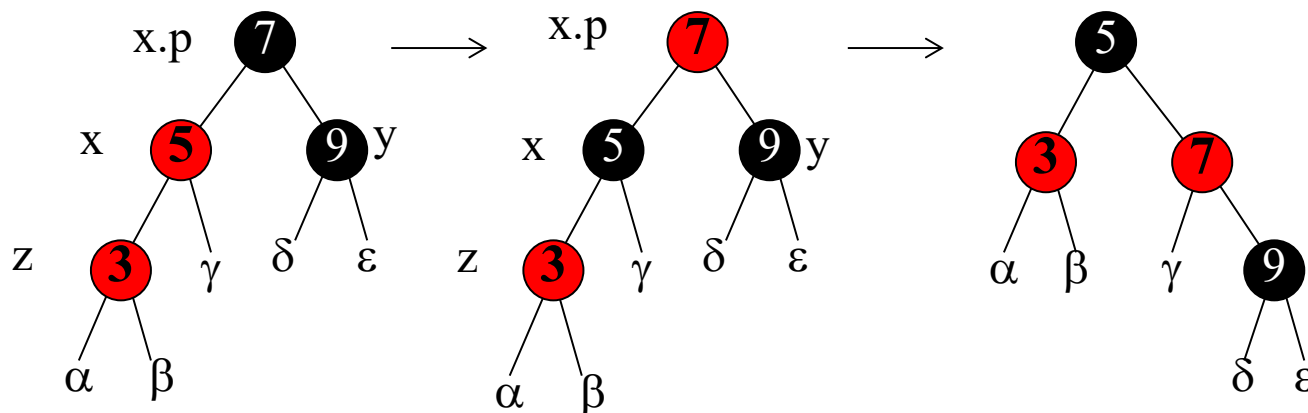
- Caso 2: y nero e z figlio destro di x



- A questo punto ci siamo messi nel caso 3

Funzionamento di RB-INSERT

- Caso 3: y nero e z figlio sinistro di x



- Ogni volta che RB-INSERT-FIXUP viene invocato esso può o terminare (casi 2 e 3), o venire applicato ricorsivamente risalendo 2 livelli nell'albero (caso 1)
 - quindi può essere invocato al massimo $O(h)$ volte, cioè $O(\log(n))$
 - si noti che una catena di invocazioni di RB-INSERT-FIXUP esegue al massimo 2 rotazioni (l'ultima chiamata)

RB-DELETE

```
RB-DELETE(T, z)
1  if z.left = T.nil or z.right = T.nil
2    y := z
3  else y := TREE-SUCCESSOR(z)
4  if y.left ≠ T.nil
5    x := y.left
6  else x := y.right
7  x.p := y.p
8  if y.p = T.nil
9    T.root := x
10 elsif y = y.p.left
11   y.p.left := x
12 else y.p.right := x
13 if y ≠ z
14   z.key := y.key
15 if y.color = BLACK
16   RB-DELETE-FIXUP(T, x)
17 return y
```

Funzionamento di RB-DELETE

- Uguale a TREE-DELETE, salvo che per l'uso di *T.nil* al posto di *NIL* (che permette l'eliminazione dell'if alla linea 7), e l'aggiunta delle righe 15-16
- Se viene cancellato un nodo rosso (cioè se *y.color* = RED) non c'è bisogno di modificare i colori dei nodi
- per come è fatto RB-DELETE, viene cancellato un nodo (*y*) che ha al massimo un figlio diverso da *T.nil*, e se *y.color* = RED il nodo *x* che prende il posto di *y* è per forza nero

RB-DELETE-FIXUP

```
RB-DELETE-FIXUP(T, x)
1  if x.color = RED or x.p = T.nil
2      x.color := BLACK           // Caso 0
3  elsif x = x.p.left             // x e' figlio sinistro
4      w := x.p.right             // w e' fratello di x
5      if w.color = RED
6          w.color := BLACK       // Caso 1
7          x.p.color := RED       // Caso 1
8          LEFT-ROTATE(T, x.p)    // Caso 1
9          w := x.p.right         // Caso 1
10     if w.left.color = BLACK and w.right.color = BLACK
11         w.color := RED         // Caso 2
12         RB-DELETE-FIXUP(T, x.p) // Caso 2
13     else if w.right.color = BLACK
14         w.left.color := BLACK  // Caso 3
15         w.color := RED         // Caso 3
16         ROTATE-RIGHT(T, w)    // Caso 3
17         w := x.p.right        // Caso 3
18         w.color := x.p.color   // Caso 4
19         x.p.color := BLACK     // Caso 4
20         w.right.color := BLACK // Caso 4
21         ROTATE-LEFT(T, x.p)   // Caso 4
22 else (come 4-21, scambiando "right" e "left")
```

Aggiustamento proprietà

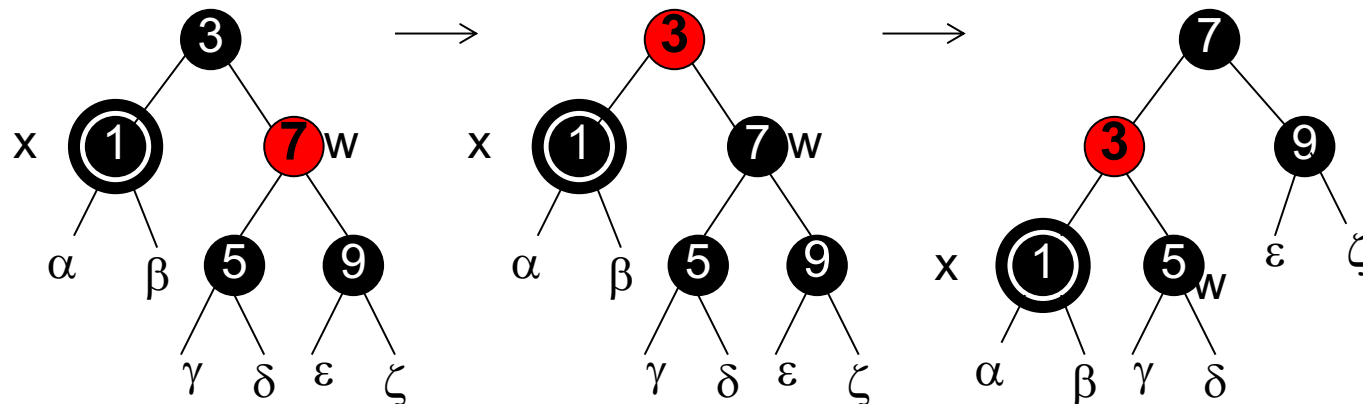
- Come al solito, y è il nodo effettivamente cancellato e x è quello che prende il suo posto
- Se y è nero possiamo aver violato le proprietà
 - 1 (radice rossa, se y è la radice e x è rosso)
 - 3 (due rossi consecutivi se $y.p$ e x sono rossi)
 - 5 (i cammini con y hanno un nero in meno)
 - Correggiamo aggiungendo «un nero in più» a x
 - Quindi x può assumere, virtualmente, colorazione «doppiamente nera» oppure «rossa e nera»
 - L'aggiustamento consiste nello spostare questo nero in più verso l'alto nell'albero

Funzionamento di RB-DELETE-FIXUP

- Caso 0: x è un nodo rosso, oppure è la radice



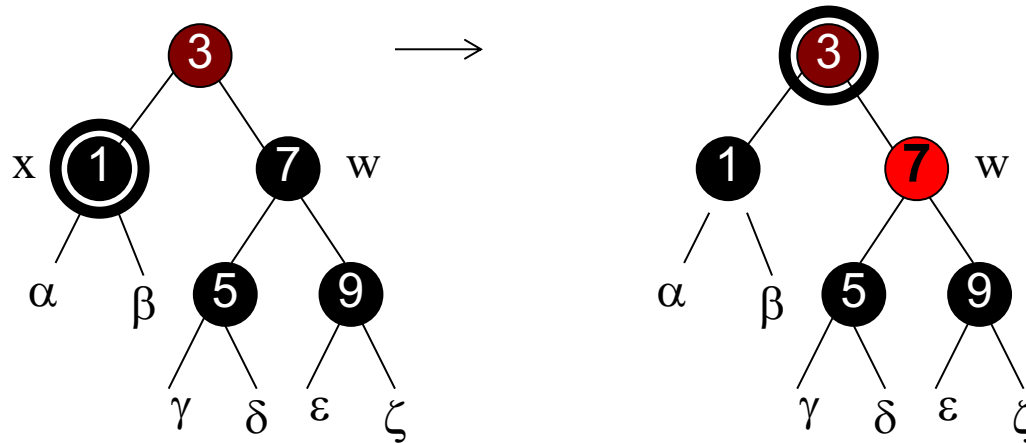
- Caso 1: x è un nodo nero, il suo fratello destro w è rosso, e di conseguenza il padre $x.p$ è nero



- diventa o il caso 2, o il caso 3, o il caso 4

Funzionamento di RB-DELETE-FIXUP

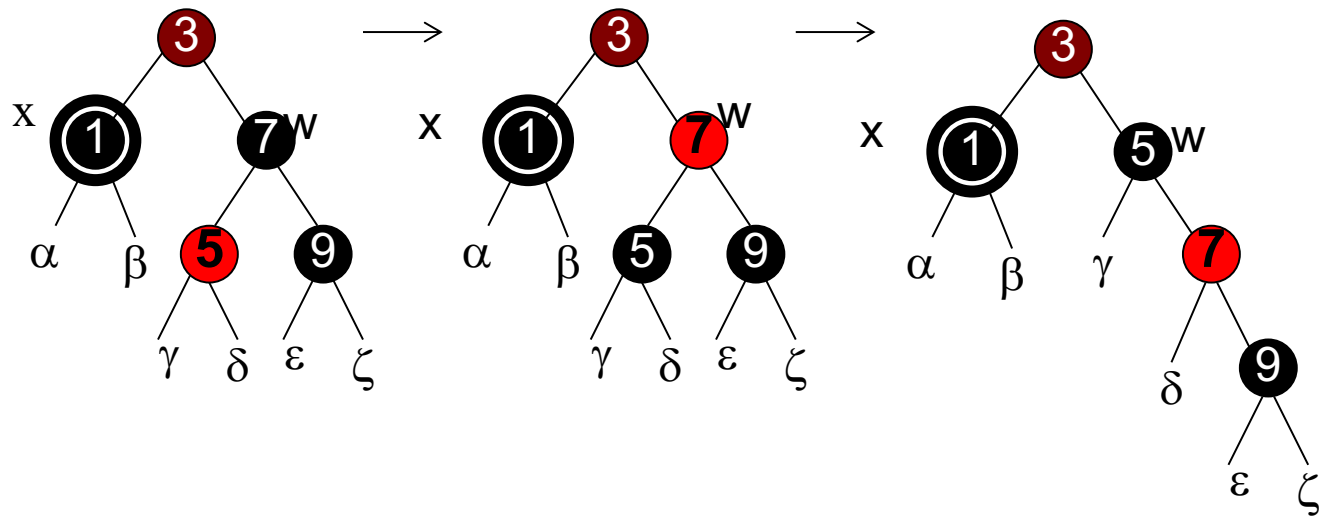
- Caso 2: x è nero, suo fratello destro w è nero con figli entrambi neri



- se arriviamo al caso 2 dal caso 1, allora $x.p$ è rosso, e quando RB-DELETE-FIXUP viene invocato su di esso termina subito (arriva subito al caso 0)

Funzionamento di RB-DELETE-FIXUP

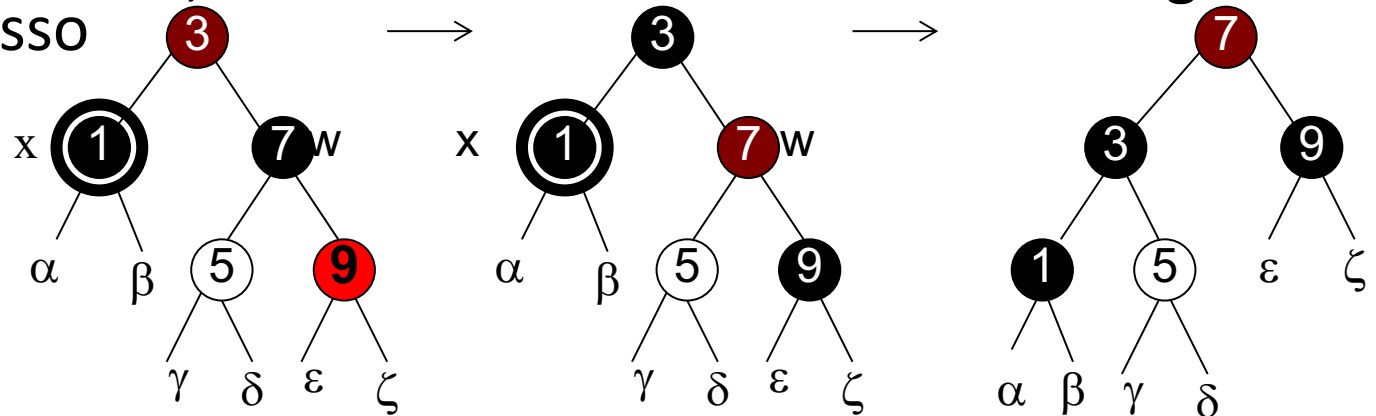
- Caso 3: x è nero, suo fratello destro w è nero con figlio sinistro rosso e figlio destro nero



- Diventa il caso 4

Funzionamento di RB-DELETE-FIXUP

- Caso 4: x è nero, suo fratello destro w è nero con figlio destro rosso



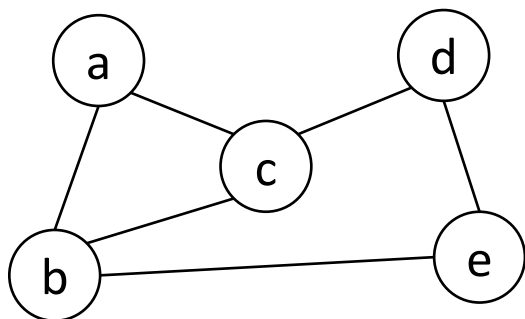
- Ogni volta che RB-DELETE-FIXUP viene invocato esso può o terminare (casi 0, 1, 3 e 4), o venire applicato ricorsivamente risalendo un livello nell'albero (caso 2 non proveniente da 1)
 - quindi può essere invocato al massimo $O(h)$ volte, cioè $O(\log(n))$
 - si noti che una catena di invocazioni di RB-DELETE-FIXUP esegue al massimo 3 rotazioni (se il caso 1 diventa 3 e poi 4)

Richiamo sui grafi

- Un grafo è una coppia $G = (V, E)$, in cui:
 - V è un insieme di *nodi* (detti anche *vertici*)
 - E è un insieme di *archi* (detti anche *lati*, o *edges*)
- Un arco è una connessione tra 2 vertici
 - 2 vertici connessi da un arco sono detti *adiacenti*
 - se un arco e connette 2 vertici u e v , può essere rappresentato dalla coppia (u, v) di vertici che connette
 - quindi, $E \subseteq V^2$
 - $|V|$ è il numero di vertici nel grafo, mentre $|E|$ è il numero di archi
 - $0 \leq |E| \leq |V|^2$

Grafi orientati e non orientati

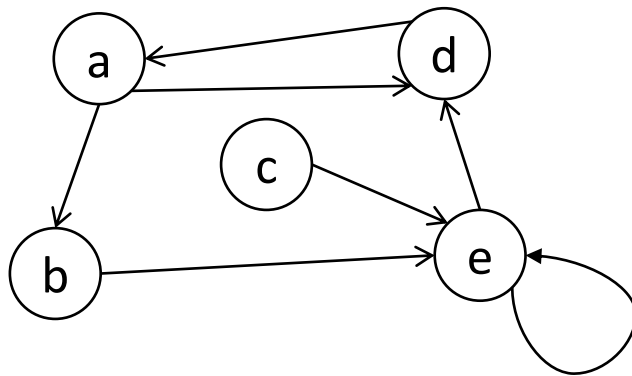
- Ci sono 2 tipi di grafi: *orientati* e *non orientati*
 - In un grafo non orientato, un arco (u, v) è lo stesso di (v, u) (non c'è nozione di direzione da un nodo all'altro)
 - In un grafo orientato (u, v) "va dal" nodo u al nodo v , ed è diverso da (v, u)
 - L'ordine dei vertici negli archi di un grafo non orientato è irrilevante



$V = \{a, b, c, d, e\}$

$E = \{(b, a) (a, c) (b, c) (d, c) (e, d) (b, e)\}$

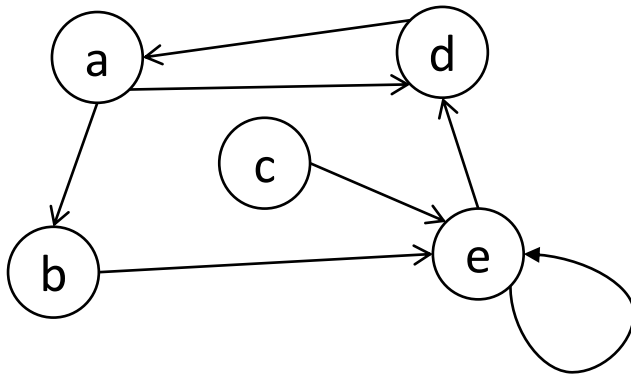
Esempio di grafo orientato



$V = \{a, b, c, d, e\}$
 $E = \{(a, b) (a, d) (d, a) (b, e)$
 $(c, e) (e, d), (e, e)\}$

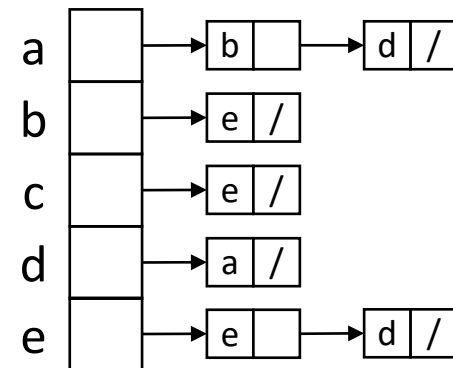
Rappresentazione di grafi in memoria

- 2 tecniche principali:
 - liste di adiacenza
 - matrice di adiacenza
- Grafo orientato



$V = \{a, b, c, d, e\}$

$E = \{(a, b) (a, d) (d, a) (b, e) (c, e) (e, d), (e, e)\}$



Liste ad.

	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	0	1
c	0	0	0	0	1
d	1	0	0	0	0
e	0	0	0	1	1

Matrice ad.

Dimensione delle rappresentazioni dei grafi

- Nel caso di liste di adiacenza abbiamo un array di liste
 - c'è una lista per ogni nodo del grafo
 - per ogni vertice v , la lista corrispondente contiene i vertici adiacenti a v
- In una matrice di adiacenza M , l'elemento m_{ij} è 1 se c'è un arco dal nodo i al nodo j , 0 altrimenti
- In entrambi i casi, dato un nodo u in un grafo G , l'attributo $u.Adj$ rappresenta l'insieme di vertici adiacenti a u
- Quanto è grande una rappresentazione con liste di adiacenza?
 - il numero totale di elementi nelle liste è $|E|$
 - il numero di elementi nell'array è $|V|$
 - la complessità spaziale è $\Theta(|V| + |E|)$
- Quanto è grande la matrice di adiacenza?
 - la dimensione della matrice è $|V|^2$, quindi la sua complessità è $\Theta(|V|^2)$

Confronto tra le rappresentazioni dei grafi

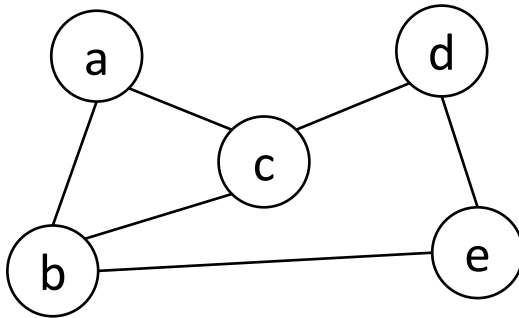
- Le liste di adiacenza sono in generale migliori quando $|E| \neq \Theta(|V|^2)$, cioè quando il grafo è sparso (quando il numero di nodi connessi “non è tanto grande”)
 - si ricordi che $|E| \leq |V|^2$, cioè $|E| = O(|V|^2)$
- Se il grafo è completo (o quasi), tanto vale usare una matrice di adiacenza
 - un grafo orientato è completo se, per ogni coppia di nodi u e v , sia l'arco (u, v) che l'arco (v, u) sono in E

Prime questioni di complessità sui grafi

- Qual è la complessità temporale per determinare se un arco (u, v) appartiene al grafo quando il grafo è rappresentato mediante:
 - liste di adiacenza?
 - una matrice di adiacenza?
- Qual è la complessità temporale per determinare il numero di archi che escono da un nodo u del grafo quando il grafo è rappresentato mediante:
 - liste di adiacenza?
 - una matrice di adiacenza?

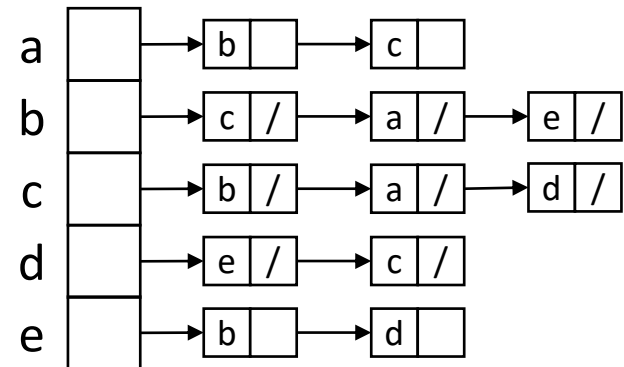
Rappresentazione di grafi non orientati in memoria

- Possiamo rappresentare un arco (u, v) di un grafo non orientato come 2 archi orientati, uno che va da u a v , ed uno che va da v ad u



$V = \{a, b, c, d, e\}$

$E = \{(b, a) (a, c) (b, c) (d, c) (e, d) (b, e)\}$



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	0	1
c	1	1	0	1	0
d	0	0	1	0	1
e	0	1	0	1	0

la matrice di adiacenza è simmetrica, quindi tutta l'informazione che serve per descrivere il grafo si trova sopra la diagonale principale

Visita in ampiezza (Breadth-First Search)

- Problema:
 - *input*: un grafo G , e un nodo s (la sorgente) di G
 - *output*: visitare tutti i nodi di G che sono raggiungibili da s (un nodo u è raggiungibile da s se c'è un cammino nel grafo che va da s a u)
- Algoritmo: Breadth-First Search (BFS)
- Idea dell'algoritmo: prima visitiamo tutti i nodi che sono a distanza 1 da s (cioè che hanno un cammino di lunghezza 1 da s), quindi visitiamo quelli a distanza 2, quindi quelli a distanza 3, e così via
- Quando visitiamo un nodo u , teniamo traccia della sua distanza da s in un attributo $u.dist$

Colorazione dei nodi durante la visita

- Mentre visitiamo i nodi, li coloriamo (cioè li marchiamo per tenere traccia della progressione dell'algoritmo)
 - un nodo è *bianco* se deve essere ancora visitato
 - un nodo è *grigio* se lo abbiamo già visitato, ma dobbiamo ancora completare la visita dei nodi ad esso adiacenti
 - un nodo è *nero* dopo che abbiamo visitato tutti i suoi nodi adiacenti
- L'algoritmo in breve:
 - all'inizio tutti i nodi sono bianchi, tranne s (la sorgente), che è grigio
 - manteniamo i nodi di cui dobbiamo ancora visitare i nodi adiacenti in una coda (che è gestita con politica FIFO!)
 - all'inizio la coda contiene solo s
 - a ogni iterazione del ciclo, eliminiamo dalla coda un elemento u , e ne visitiamo i nodi adiacenti che sono ancora bianchi (cioè che devono essere ancora visitati)
 - Si noti che, se $u.dist$ è la distanza del nodo u da s , la distanza dei nodi bianchi adiacenti ad u è $u.dist+1$

Esempi di utilizzo

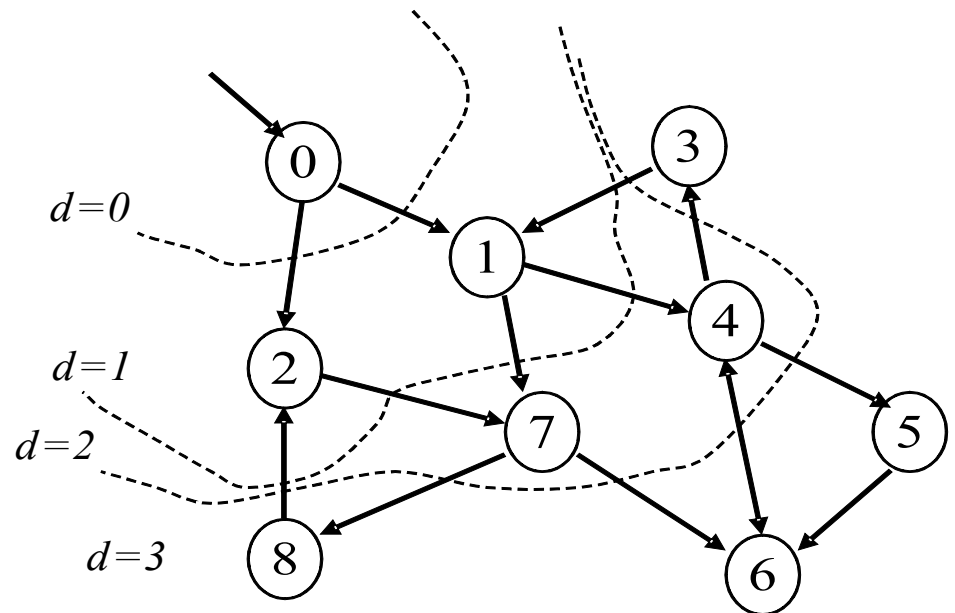
Esempio di utilizzo:

$G = [[1,2],[7,4],[7],[1],[3,5,6],[6],[4],[8,6],[2]]$

BFS ($G, 0$)

distanze dal nodo 0

$[0, 1, 1, 3, 2, 3, 3, 2, 3]$



BFS: pseudocode

```
BFS( $G, s$ )
1  for each  $u \in G.V - \{s\}$ 
2       $u.color := \text{WHITE}$ 
3       $u.dist := \infty$ 
4   $s.color := \text{GREY}$ 
5   $s.dist := 0$ 
6   $Q := \emptyset$ 
7  ENQUEUE( $Q, s$ )
8  while  $Q \neq \emptyset$ 
9       $u := \text{DEQUEUE}(Q)$ 
10     for each  $v \in u.Adj$ 
11         if  $v.color = \text{WHITE}$ 
12              $v.color := \text{GRAY}$ 
13              $v.dist := u.dist + 1$ 
14             ENQUEUE( $Q, v$ )
15      $u.color := \text{BLACK}$ 
```

Funzionamento di BFS

- Le linee 1-7 sono la fase di inizializzazione dell'algoritmo (che ha complessità $O(|V|)$)
- Le linee 8-15 sono quelle che effettivamente visitano i nodi; ogni nodo nel grafo G è accodato (e tolto dalla coda) al massimo una volta, quindi, nel ciclo **for** della linea 10, ogni lato è visitato al massimo una volta; quindi, la complessità del ciclo **while** è $O(|E|)$
- La complessità totale di BFS è $O(|V| + |E|)$

Ricerca in profondità (Depth-First Search)

- BFS si basa sull'idea di visitare i nodi con una politica FIFO (che è realizzata mediante una coda)
- Come alternativa possiamo usare una politica LIFO
- Se usiamo una politica LIFO, otteniamo un algoritmo di visita in profondità (*depth-first search*, DFS)
 - l'idea in questo caso è che, ogni volta che “mettiamo un nodo in cima allo stack”, immediatamente cominciamo a visitare i nodi a esso adiacenti
 - cioè continuiamo con la visita dei nodi che sono adiacenti a quello che da meno tempo è nello stack
 - in BFS non è così: visitiamo i nodi che sono adiacenti a quello che da più tempo è nella coda
 - è quindi sufficiente, per ottenere un algoritmo di DFS, cambiare ENQUEUE con PUSH, e DEQUEUE con POP nell'algoritmo di BFS.

Problema risolto da DFS

- In realtà, l'algoritmo DFS che vediamo risolve un problema leggermente diverso da BFS
- Problema risolto dall'algoritmo DFS
 - *input*: un grafo G
 - *output*: visitare *tutti* i nodi di G
 - nell'algoritmo BFS di prima visitiamo solo i nodi che sono raggiungibili dalla sorgente s !
- DFS è spesso usato come parte (cioè come sottoalgoritmo) di un algoritmo più complesso (da cui il problema leggermente diverso da quello risolto da BFS)
 - è spesso usato come “passo preparatorio” prima di lanciare l'algoritmo “principale”

Considerazioni su DFS

- Come in BFS, in DFS coloriamo i nodi di bianco, grigio e nero (con lo stesso significato che in BFS)
- Come detto in precedenza, questa volta usiamo una politica di visita LIFO, quindi usiamo un meccanismo analogo a quello dello stack
 - in questo caso, il meccanismo a pila viene dal fatto che l'algoritmo è ricorsivo
 - invece di fare push e pop di vertici su uno stack, facciamo push e pop di chiamate ricorsive dell'algoritmo sullo stack delle chiamate
- L'algoritmo DFS tiene traccia di “quando” i nodi sono messi sullo stack ed anche di “quando” sono tolti da esso
 - c'è una variabile (globale), *time*, che è messa a 0 nella fase di inizializzazione dell'algoritmo, e che è incrementata di 1 sia appena dopo che un nodo è messo sullo stack che appena prima di togliere un nodo dallo stack
 - usiamo la variabile *time* per tenere traccia di 2 altri valori:
 - il “tempo” di quando inizia la “scoperta” (discovery time) di un nodo, ed il “tempo” di quando la scoperta termina (finishing time)
 - l'inizio della scoperta di un nodo u è memorizzata nell'attributo $u.d$, mentre la sua fine nell'attributo $u.f$

DFS: pseudocodice

DFS(G)

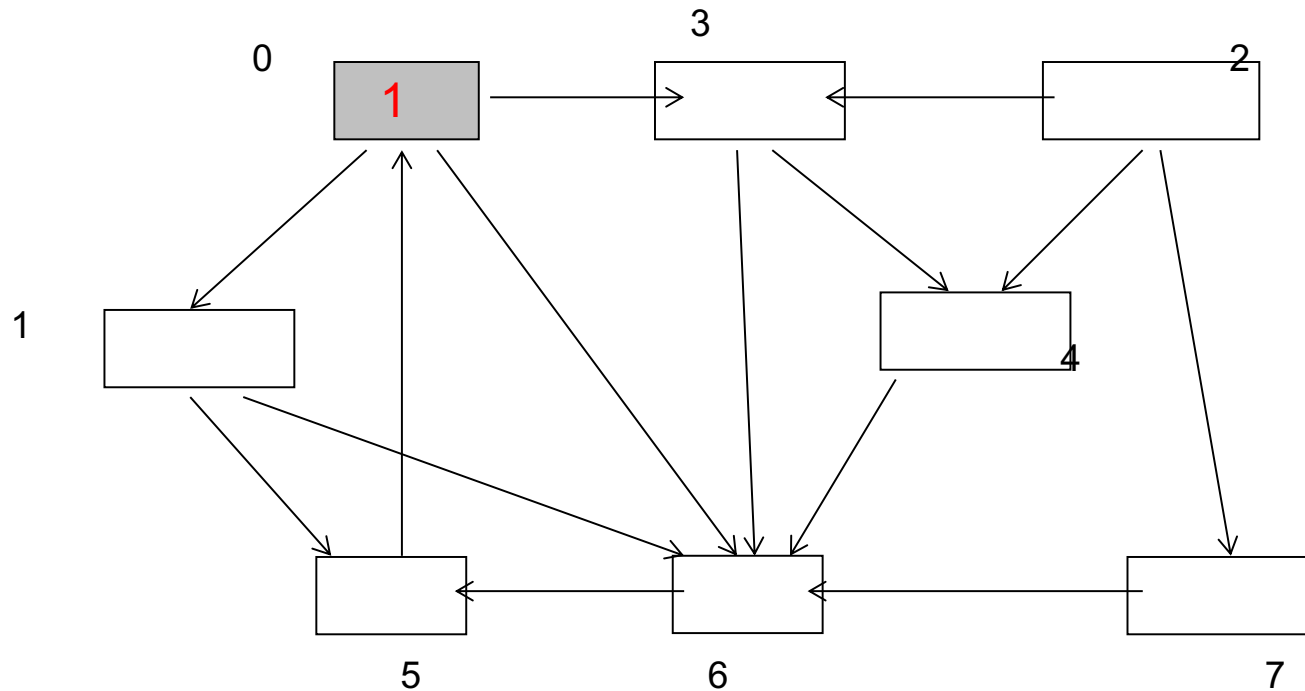
```
1 for each  $u \in G.V$ 
2    $u.color := WHITE$ 
3  $time := 0$ 
4 for each  $u \in G.V$ 
5   if  $u.color = WHITE$ 
6     DFS-VISIT( $u$ )
```

DFS-VISIT(u)

```
1  $u.color := GRAY$ 
2  $time := time + 1$ 
3  $u.d := time$ 
4 for each  $v \in u.Adj$ 
5   if  $v.color = WHITE$ 
6     DFS-VISIT( $v$ )
7  $u.color := BLACK$ 
8  $time := time + 1$ 
9  $u.f := time$ 
```

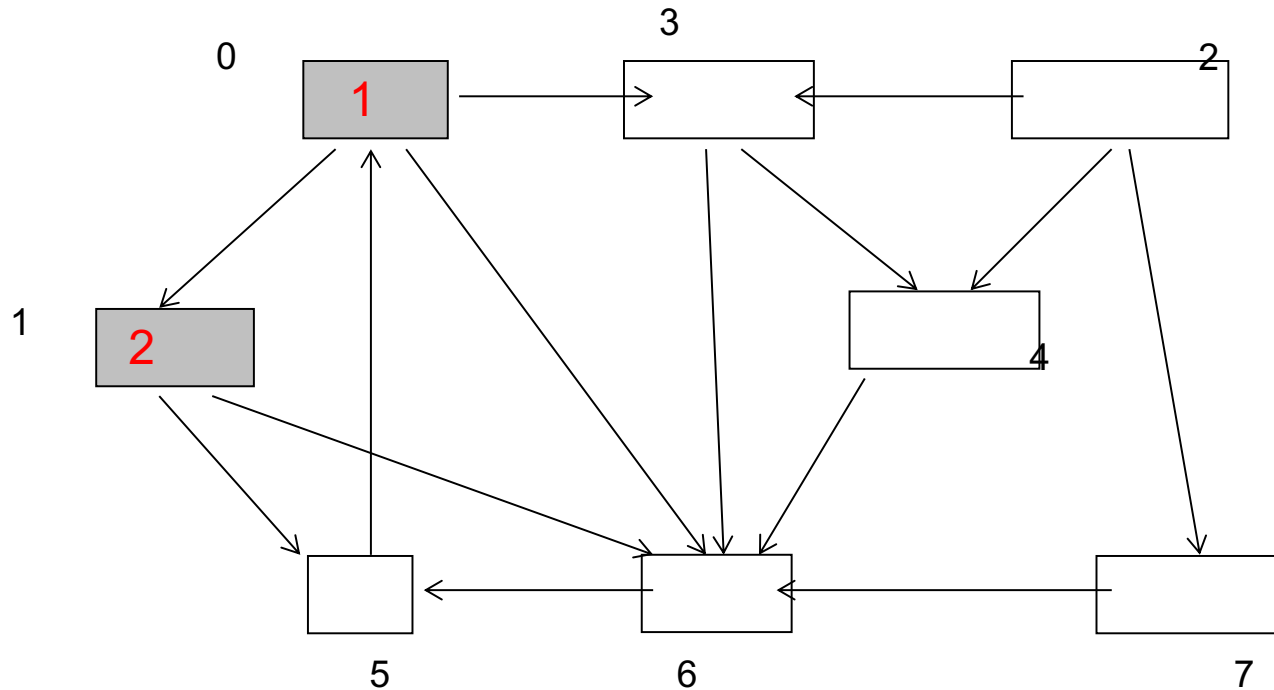
- Le linee 1-3 di DFS inizializzano i nodi colorandoli tutti di bianco, e mettono il tempo a 0: tempo di esecuzione $\Theta(|V|)$
- L'algoritmo DFS-VISIT è ripetuto (linee 4-6 di DFS) fino a che non ci sono più nodi da visitare
 - come in BFS, ogni nodo è "messo sullo stack" (che in questo caso corrisponde ad invocare DFS-VISIT sul nodo) solo una volta
 - quindi, ogni lato è visitato esattamente una volta durante l'esecuzione del ciclo for delle linee 4-6 di DFS-VISIT, quindi queste prendono tempo $\Theta(|E|)$
- In tutto, la complessità di DFS è $\Theta(|V| + |E|)$

Esempio



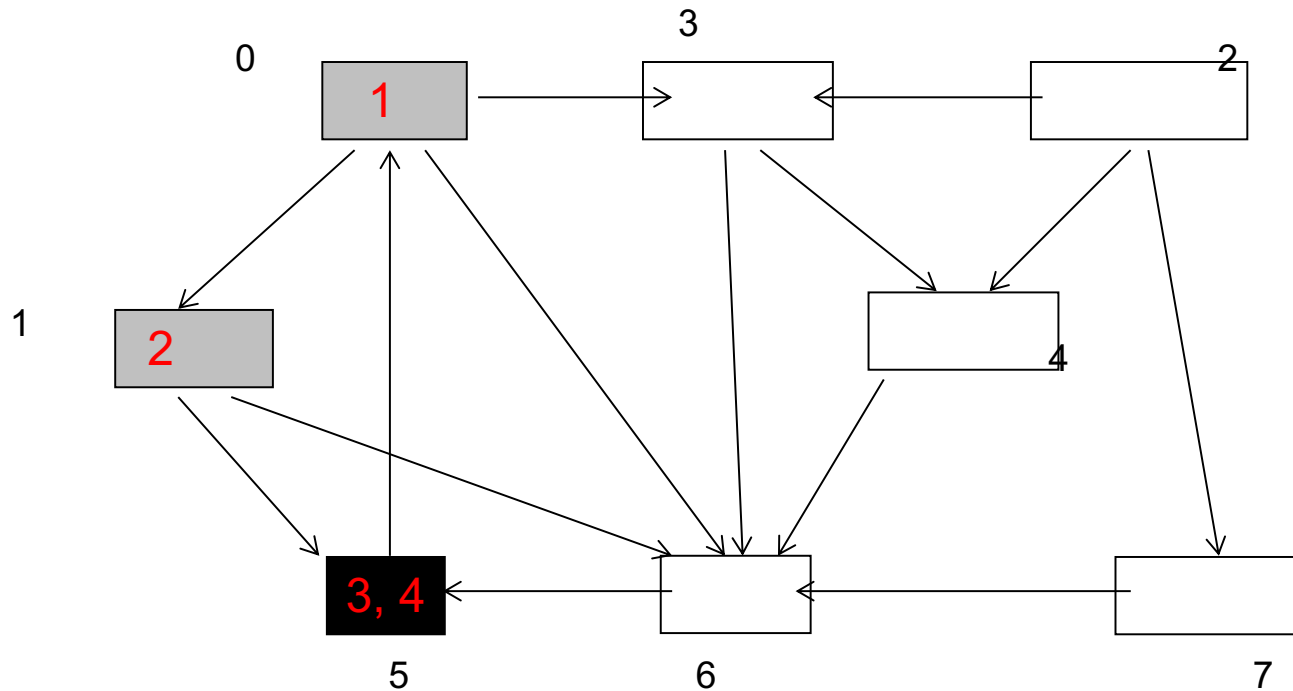
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



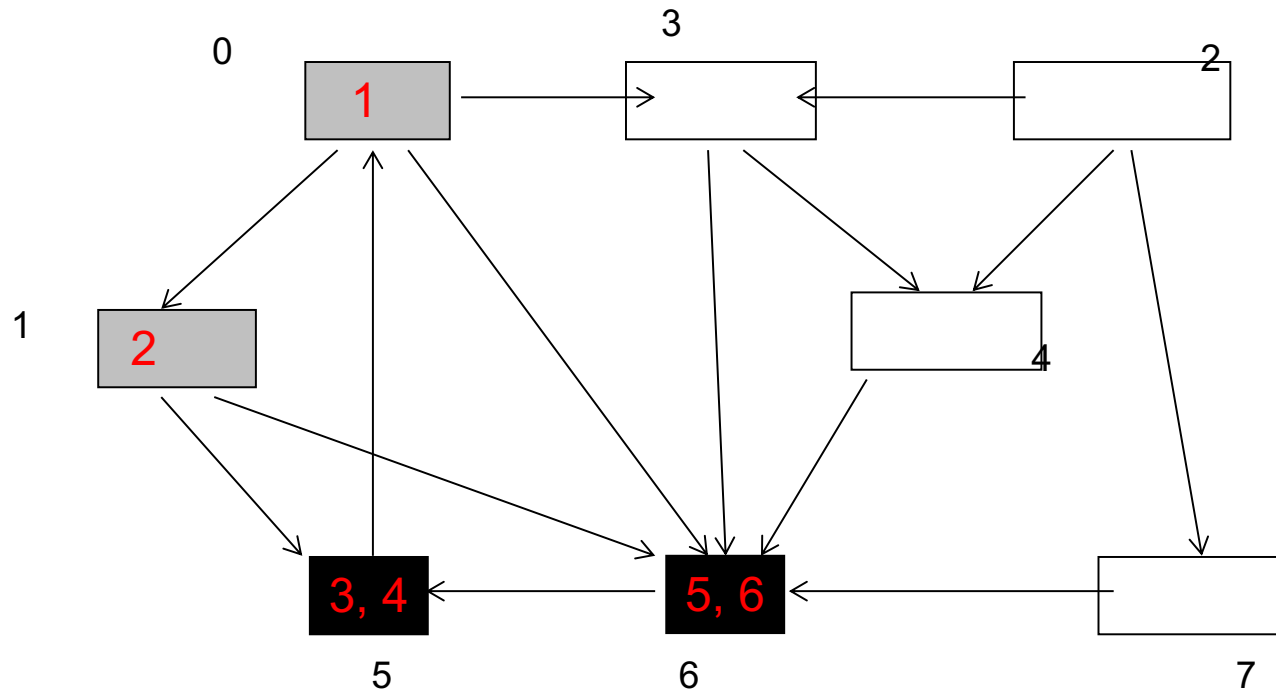
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



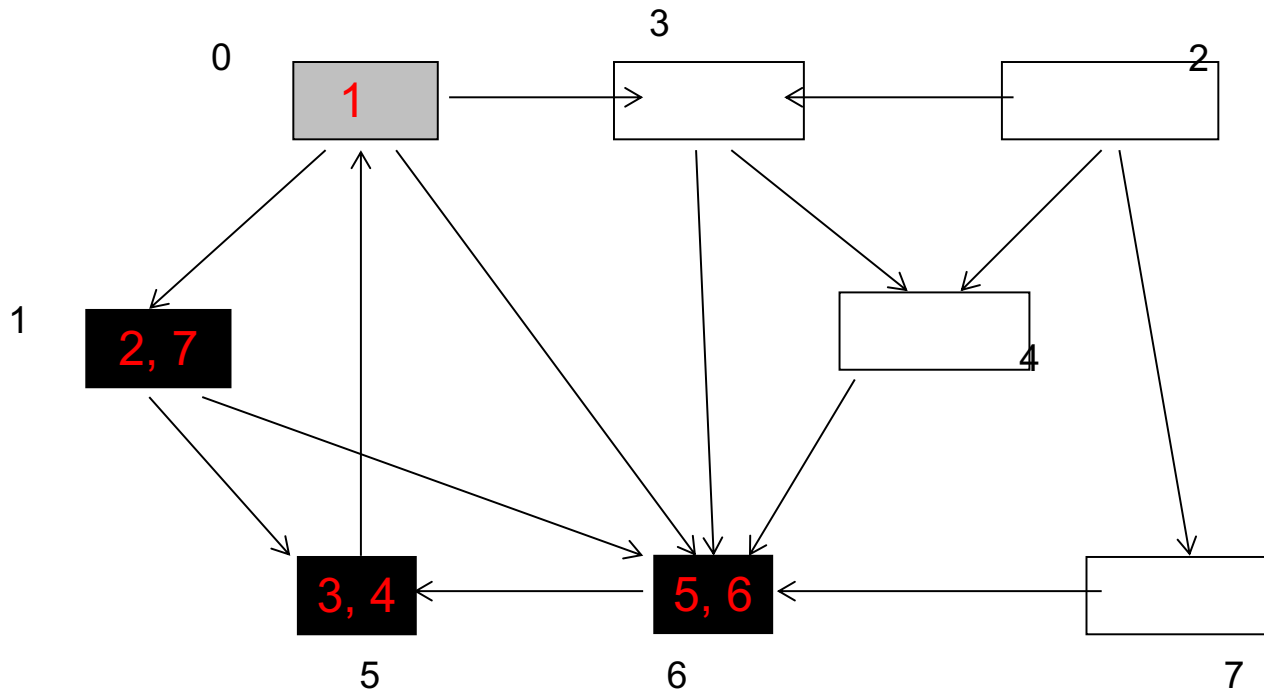
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



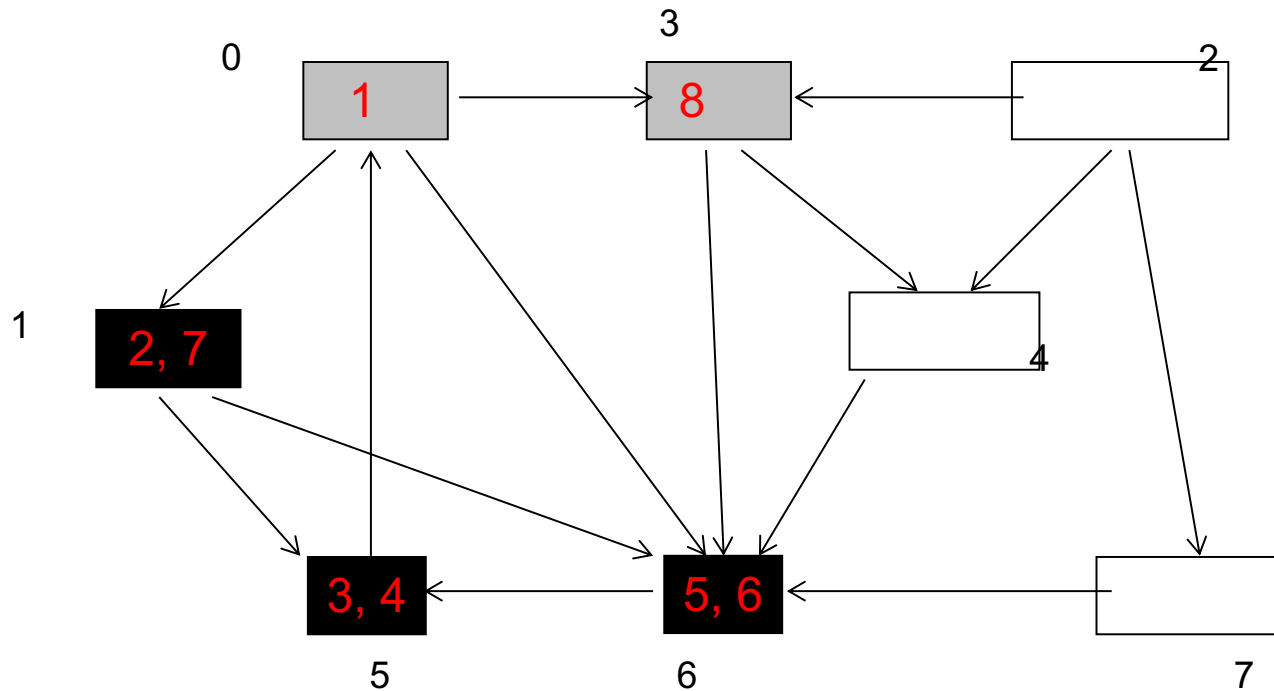
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



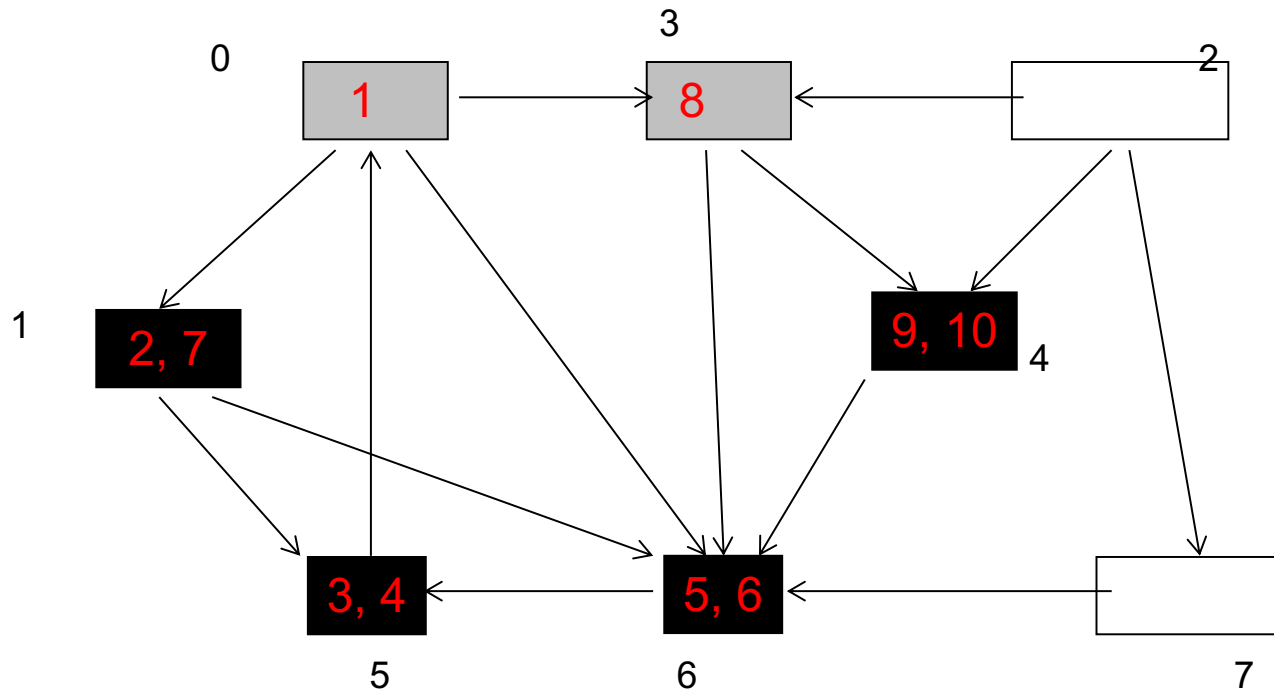
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



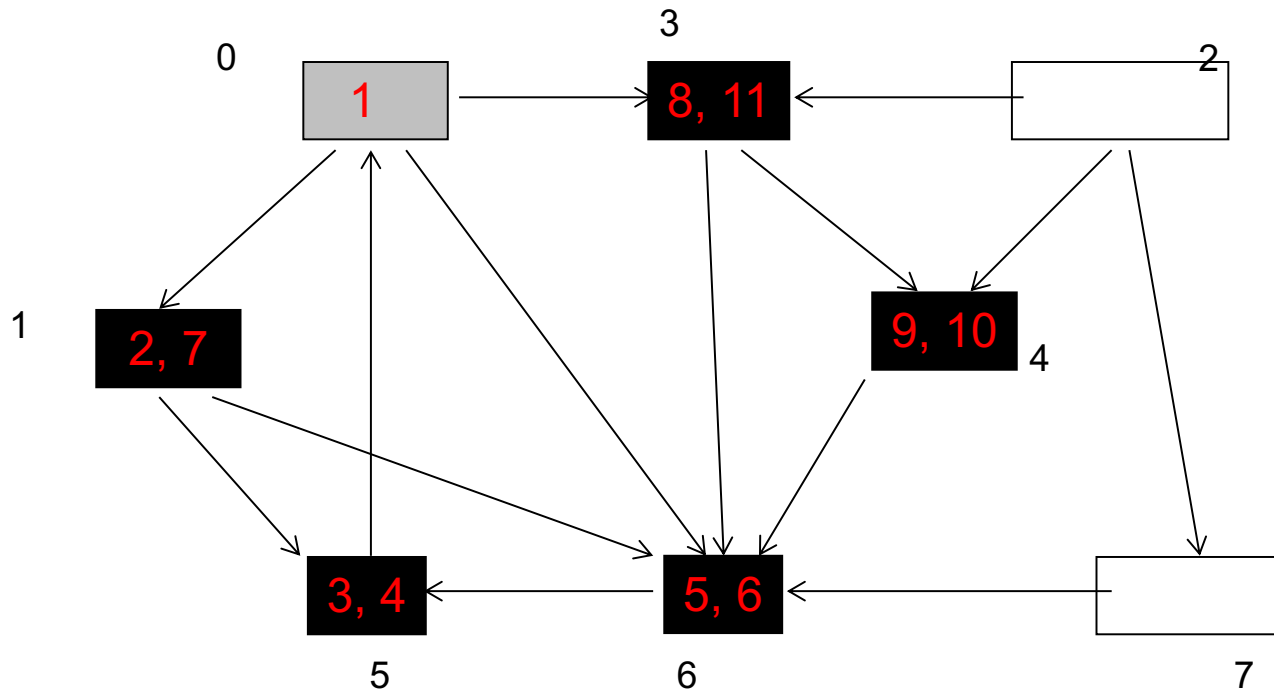
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



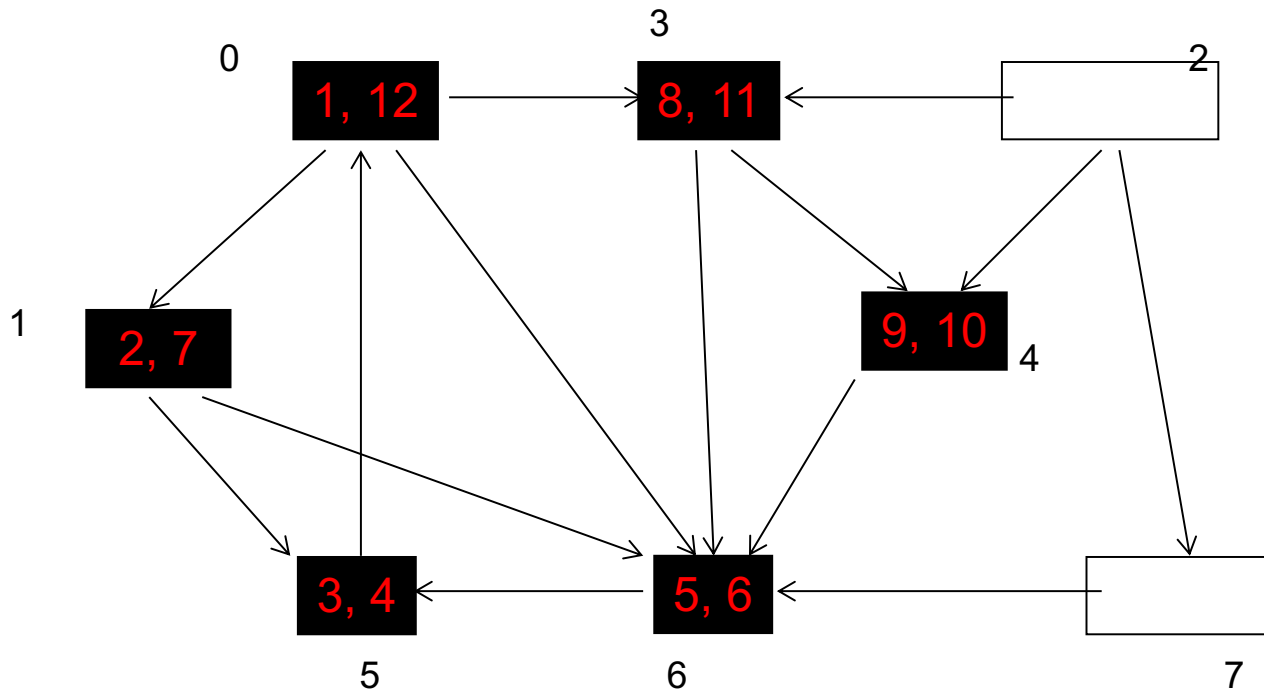
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



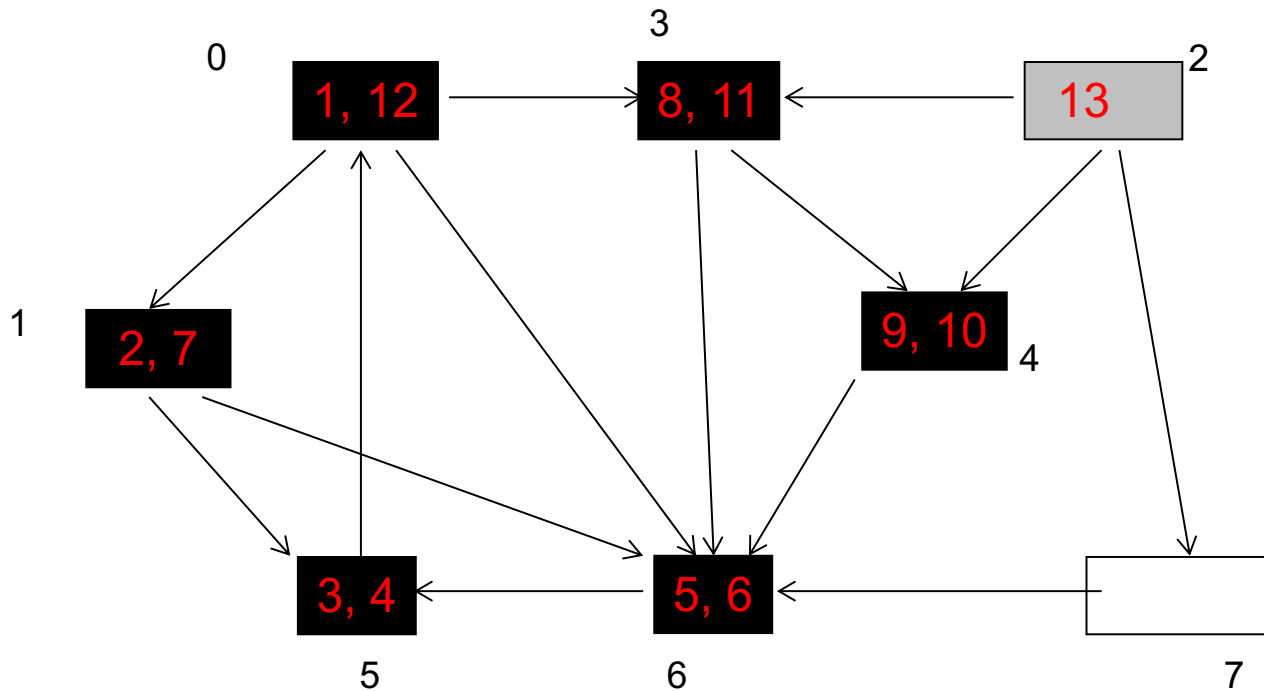
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



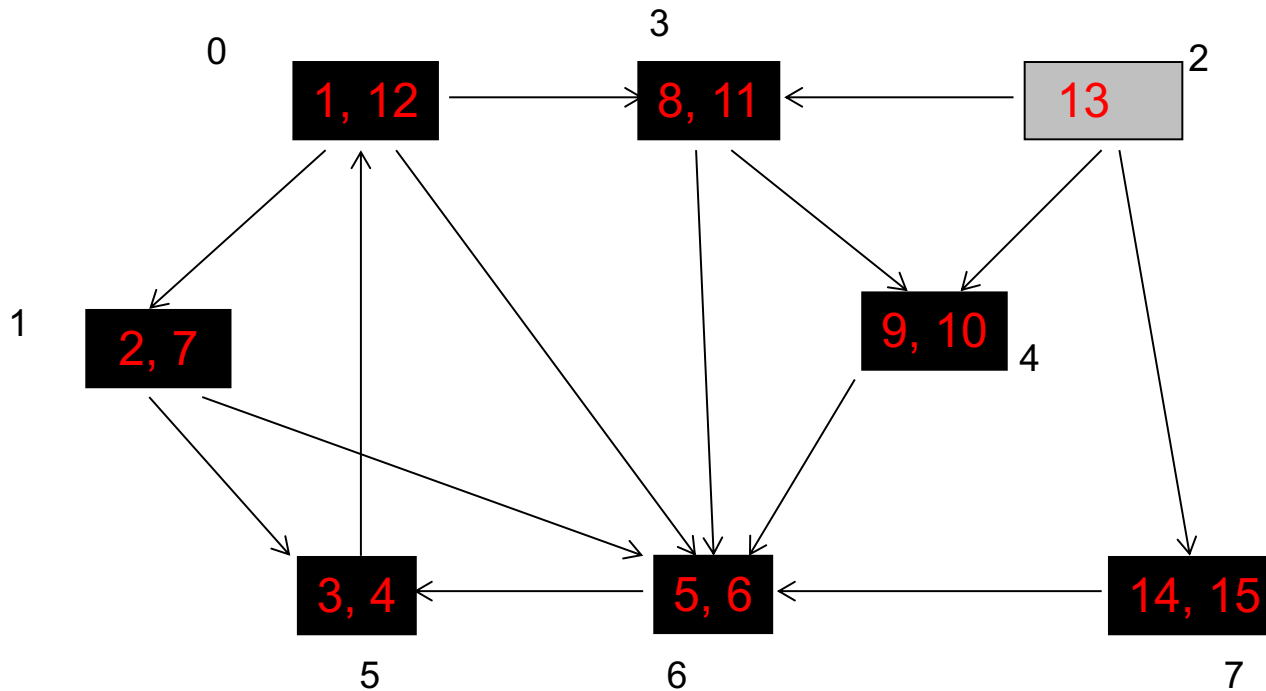
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



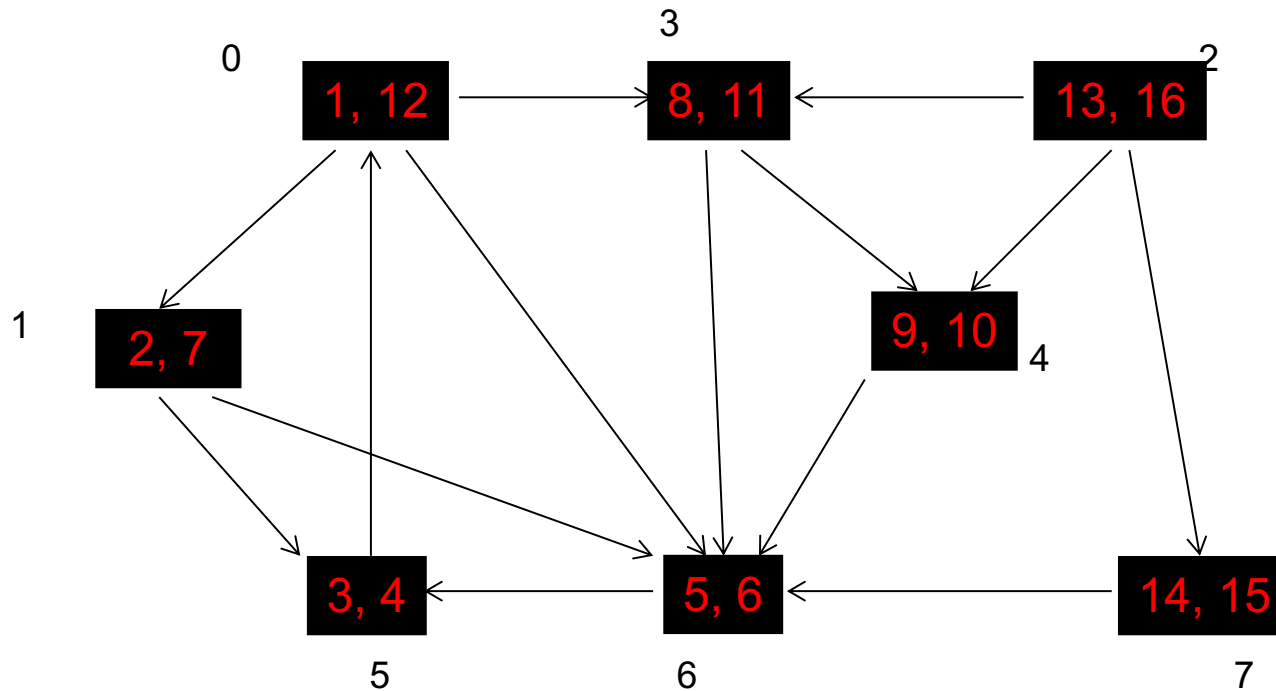
- Per ogni nodo u , mostro $u.d$ e $u.f$

Esempio



- Per ogni nodo u , mostro $u.d$ e $u.f$

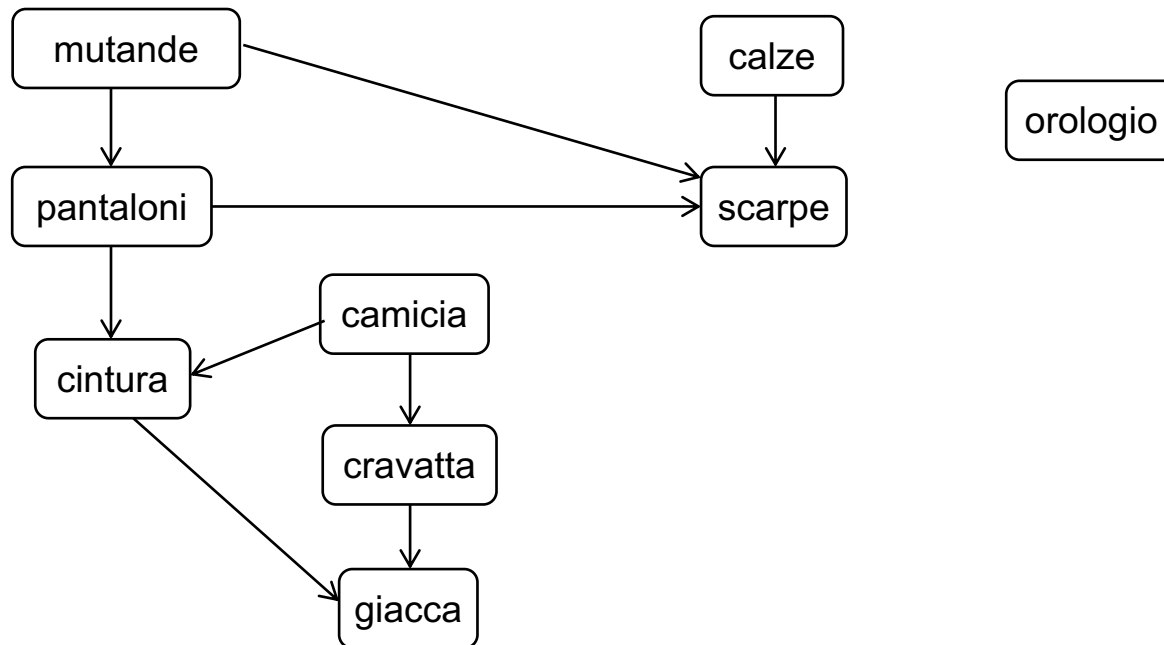
Esempio



- Per ogni nodo u , mostro $u.d$ e $u.f$

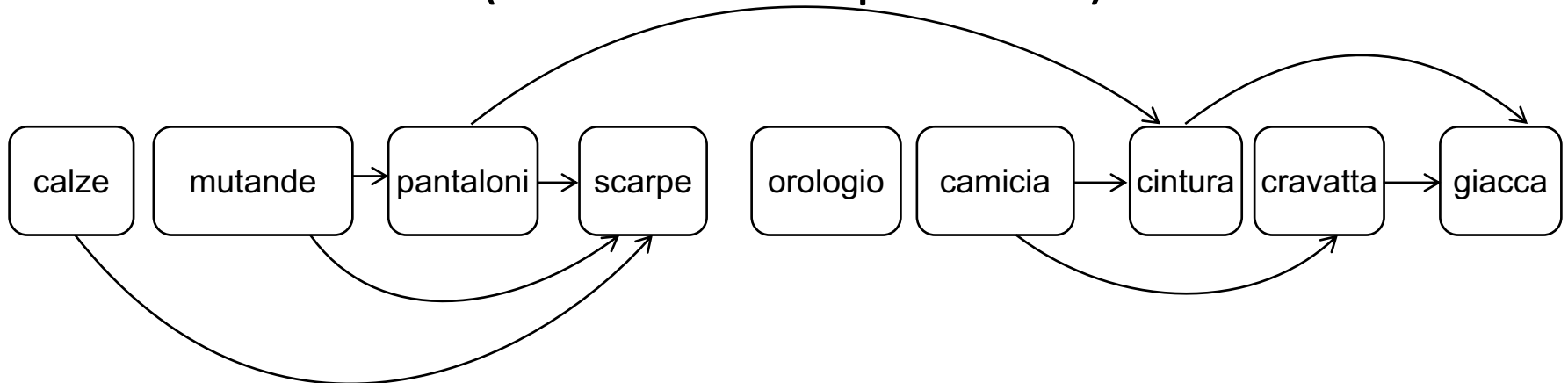
Grafo orientato aciclico

- Supponiamo di avere un grafo orientato aciclico (*directed acyclic graph*, DAG) che rappresenta le precedenze tra eventi, come questo



Ordinamento topologico

- Un *ordinamento topologico* di un DAG è un ordinamento lineare dei nodi del grafo tale che, se nel DAG c'è un arco (u, v) , allora il nodo u precede v nell'ordinamento
- Per esempio, un ordinamento topologico del DAG della slide precedente potrebbe dare il seguente ordinamento (non è l'unico possibile)



Problema dell'ordinamento topologico

- Un ordinamento topologico fornisce un ordinamento che rispetta le precedenze tra eventi
- Il problema dell'ordinamento topologico di un DAG è il seguente:
 - input: un DAG G
 - output: una lista che è un ordinamento topologico di G
 - una lista è un ordinamento lineare, e l'ordine è dalle connessioni
- Idea per l'algoritmo:
 - visitiamo il DAG con un algoritmo DFS
 - quando coloriamo un nodo u di G di nero (cioè ogni volta che finiamo di visitare un nodo di G), inseriamo u in testa alla lista
 - dopo che abbiamo visitato tutti i nodi di G , la lista che abbiamo costruito è un ordinamento topologico di G , e lo restituiamo
 - L'ordine è l'inverso di quello dato dal finishing time dei nodi

Pseudocodice di TOPOLOGICAL-SORT

```
TOPOLOGICAL-SORT(G)
```

```
1 L := ∅  
2 for each u ∈ G.V  
3   u.color := WHITE  
4 for each u ∈ G.V  
5   if u.color = WHITE  
6     TOPSORT-VISIT(L, u)  
7 return L
```

```
TOPSORT-VISIT(L, u)
```

```
1 u.color := GRAY  
2 for each v ∈ u.Adj  
3   if v.color = WHITE  
4     TOPSORT-VISIT(L, v)  
5 crea l'elemento di lista x  
6 x.key := u  
7 LIST-INSERT(L, x)  
8 u.color := BLACK
```

- Il tempo di esecuzione di TOPSORT è lo stesso di DFS, cioè $\Theta(|V| + |E|)$
 - le linee 5-7 di TOPSORT-VISIT impiegano tempo $\Theta(1)$ (come le linee 2-3 e 7-8 di DFS-VISIT), ed il resto dell'algoritmo è come DFS
 - tranne la gestione della variabile *time*, che possiamo evitare