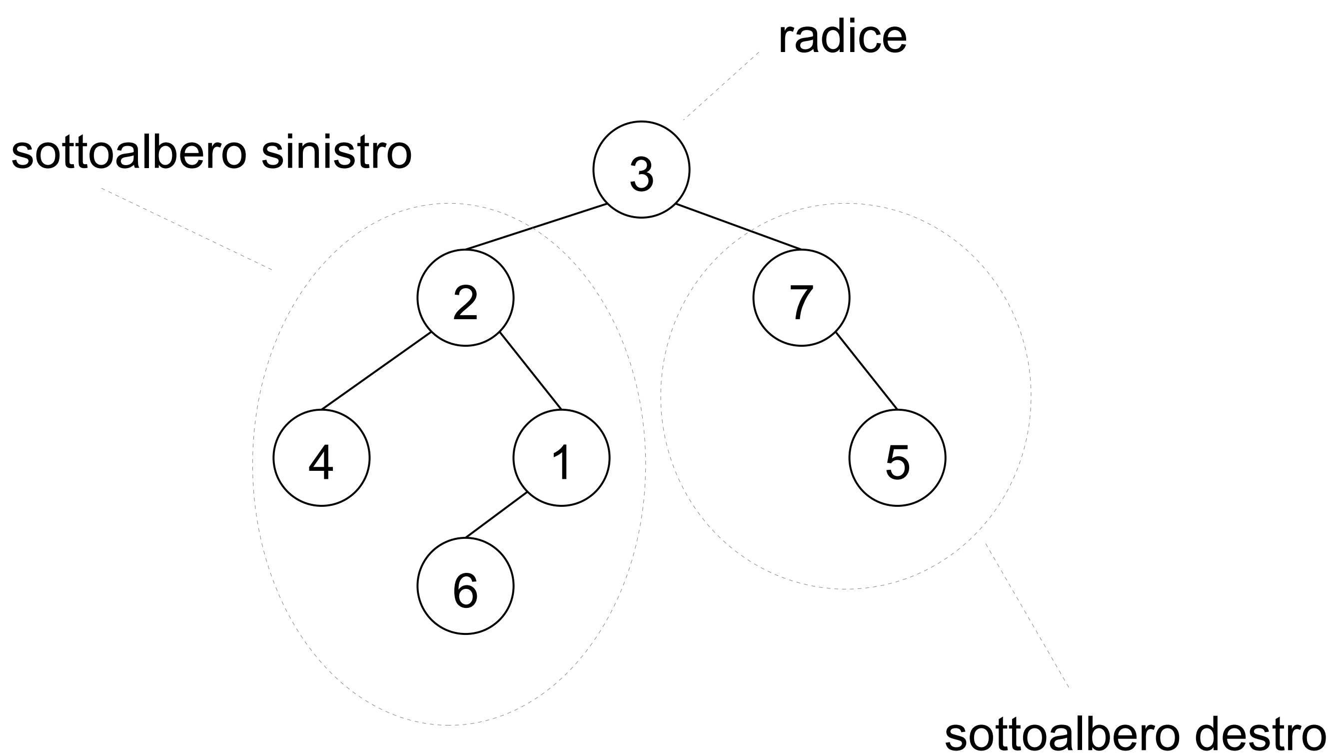
Strutture dati
(parte II: alberi e grafi)

Alberi binari

- Un **albero binario** è fatto di 3 elementi: un nodo *radice*; un albero binario che è il *sottoalbero sinistro*, ed 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*

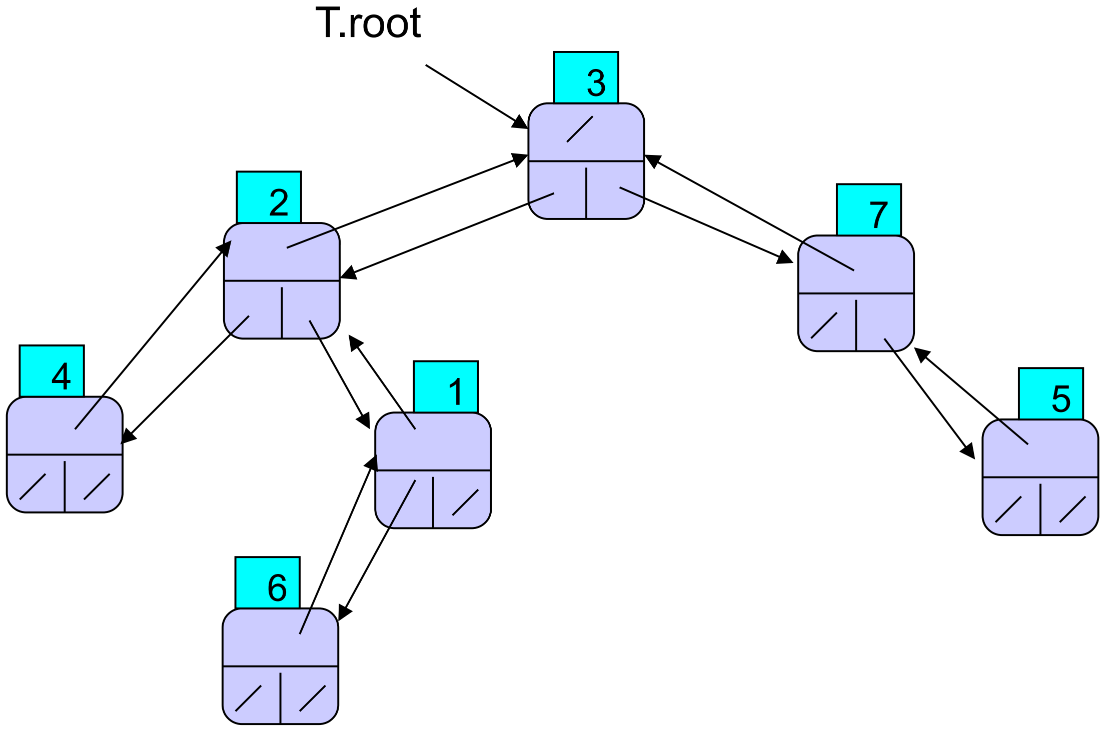
Esempio di albero binario



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

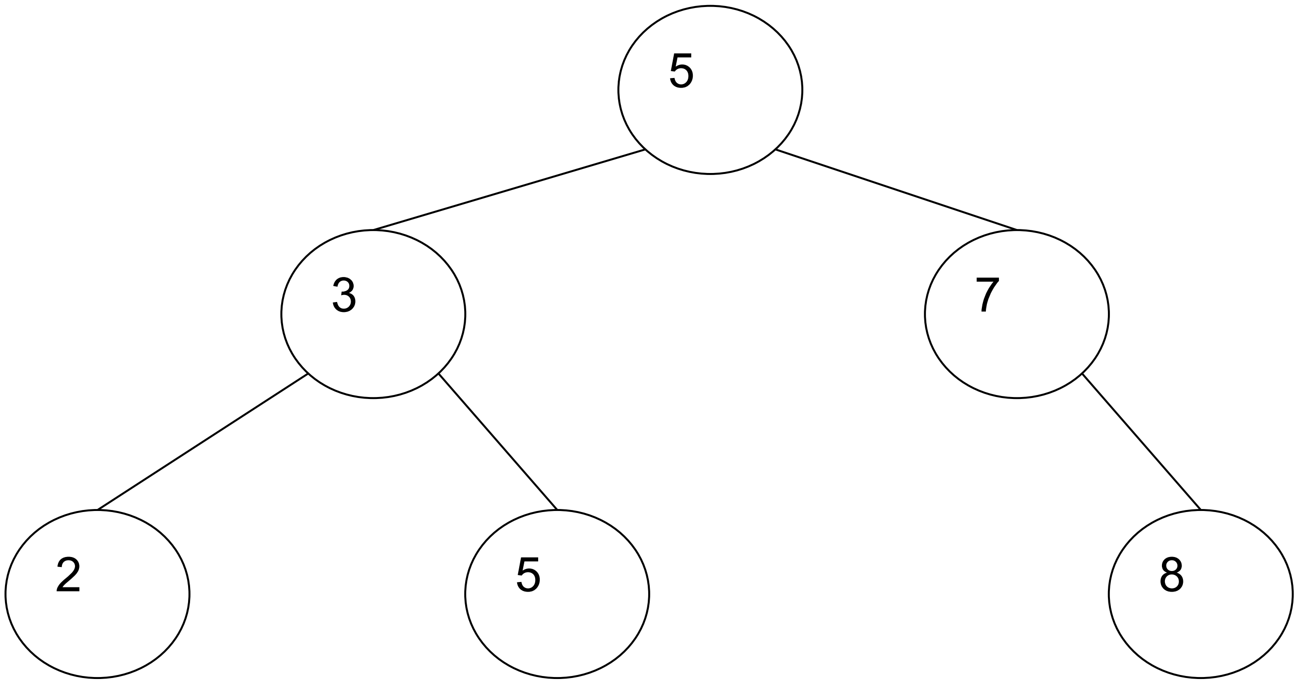
- Si noti che:
  - se il sottoalbero sinistro (destro) di un nodo *x* è vuoto, allora *x.left* = *NIL* (*x.right* = *NIL*)
  - *x.p* = *NIL* se e solo se *x* è la radice (cioè *x* = *T.root*)
- Per esempio:



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



- Una tipica operazione che viene fatta su un albero è di attraversarlo (*walk* through)
- Lo scopo dell'attraversamento di un albero è di produrre (le chiavi associate a) gli elementi dell'albero
- Ci sono diversi modi di attraversare un albero; un modo è l'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
- Si noti che:
  - come spesso accade con gli algoritmi sugli alberi (che è un astruttura 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)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

- Se *T* è un BST, INORDER-TREE-WALK(*T.root*) stampa tutti gli elementi di *T* in ordine crescente
- 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 possibili strategie di attraversamento: *anticipato* (**preorder** tree walk), e *posticipato* (**postorder** tree walk)
  - in preorder, la radice è restituita *prima* dei sottoalberi
  - 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 grande, cerca nel sottoalbero destro

```
TREE-SEARCH(x, k)
if x = NIL or k = x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

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

- 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

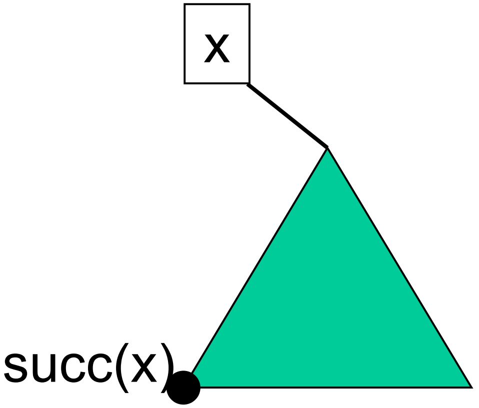
```
TREE-MINIMUM(x)
while x.left ≠ NIL
  x := x.left
return x
```

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

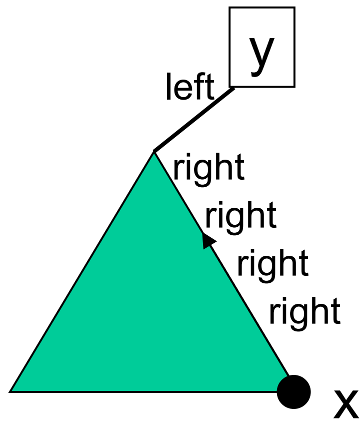
- Entrambi gli algoritmi hanno tempo di esecuzione che è O(*h*), con *h* l'altezza dell'albero

Operazioni sui BST (2)

- 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*



- 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*



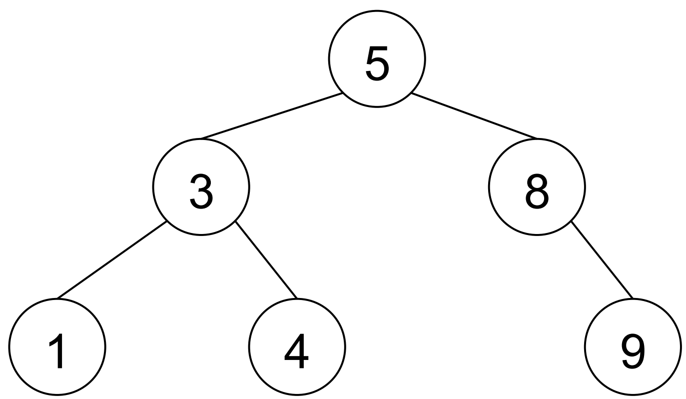
Molto informalmente:
per trovare y mi basta salire dai "right" fino a quando risalgo un "left"

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
  return TREE-MINIMUM(x.right)
y := x.p
while y ≠ NIL and x = y.right
  x := y
  y := y.p
return y
```

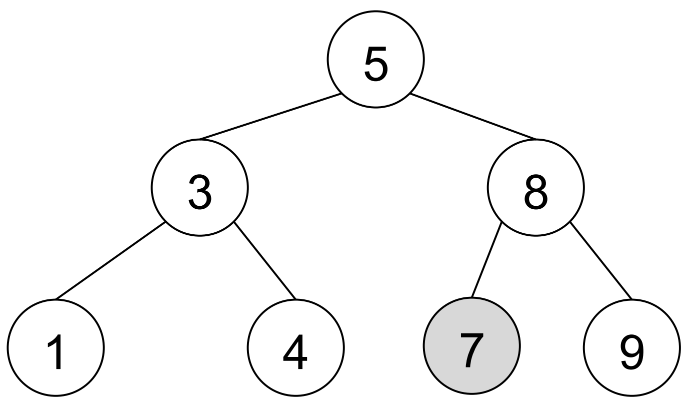
- Il tempo di esecuzione per `TREE-SUCCESSOR` è O(*h*)
- Esercizio: scrivere l'algoritmo `TREE-PREDECESSOR` e darne la complessità

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:



- eseguiamo i seguenti passi:
  - confrontiamo 5 con 7 e decidiamo che il nuovo elemento deve essere aggiunto al sottoalbero destro di 5
  - confrontiamo 8 con 7 e decidiamo che 7 deve essere aggiunto al sottoalbero sinistro di 8
  - notiamo che il sottoalbero sinistro di 8 è vuoto, e aggiungiamo 7 come sottoalbero sinistro di 8
- quindi, otteniamo il nuovo albero:



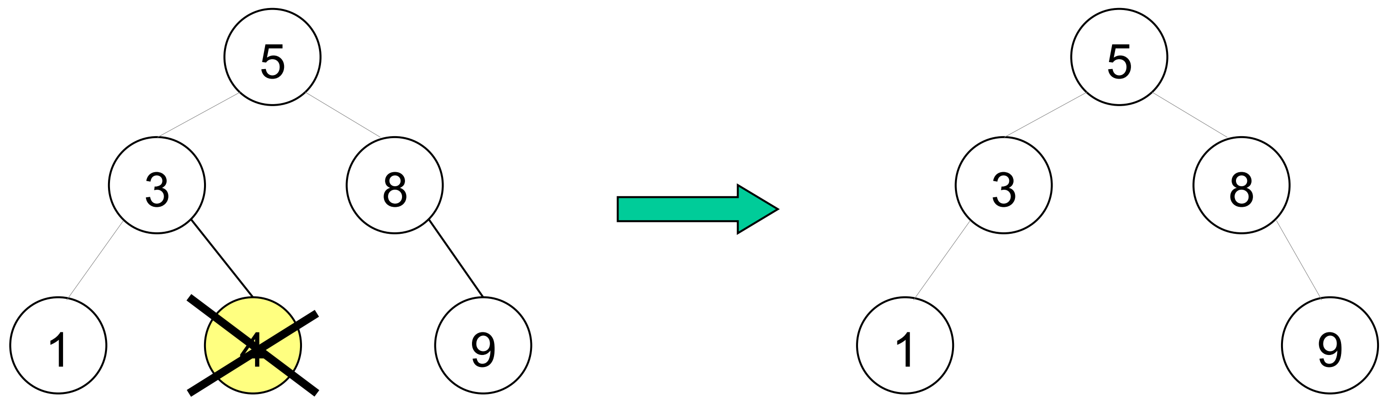
Insert: pseudocodice

```
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  elsif z.key < y.key
12    y.left := z
13  else y.right := z
```

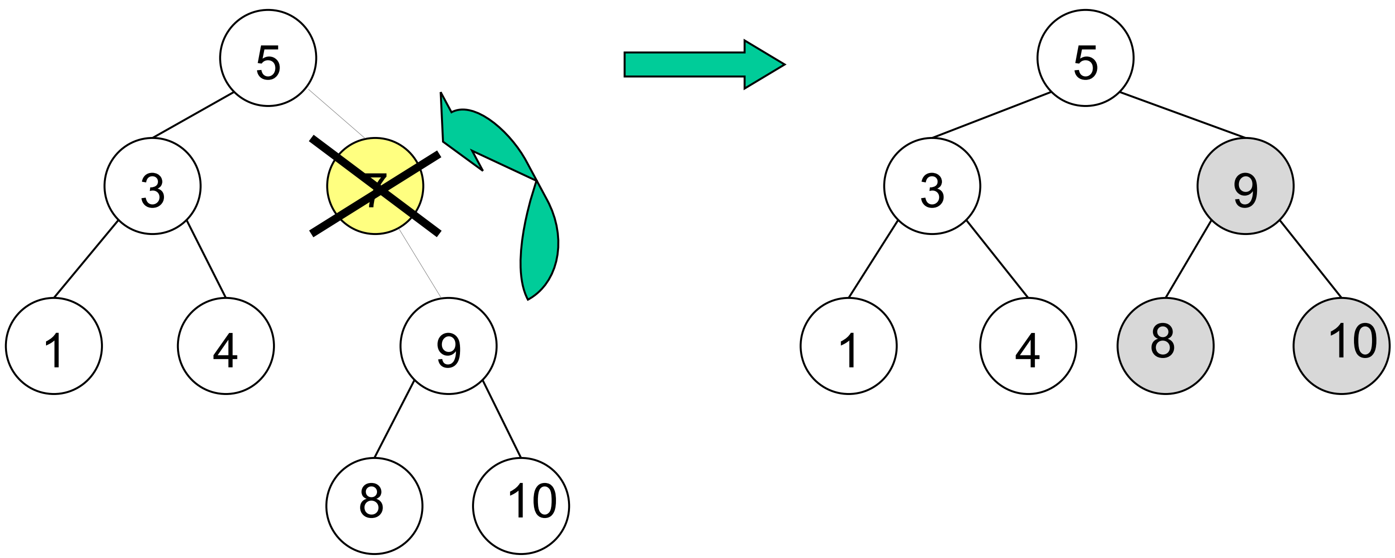
- Si noti che inseriamo un oggetto, *z*, che assumiamo sia stato inizializzato con la chiave desiderata
- Il tempo di esecuzione di TREE-INSERT è O(*h*)
  - infatti, scendiamo nell'albero nel ciclo **while** (che al massimo richiede tante ripetizioni quanta è l'altezza dell'albero), e il resto (linee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

Cancellazione

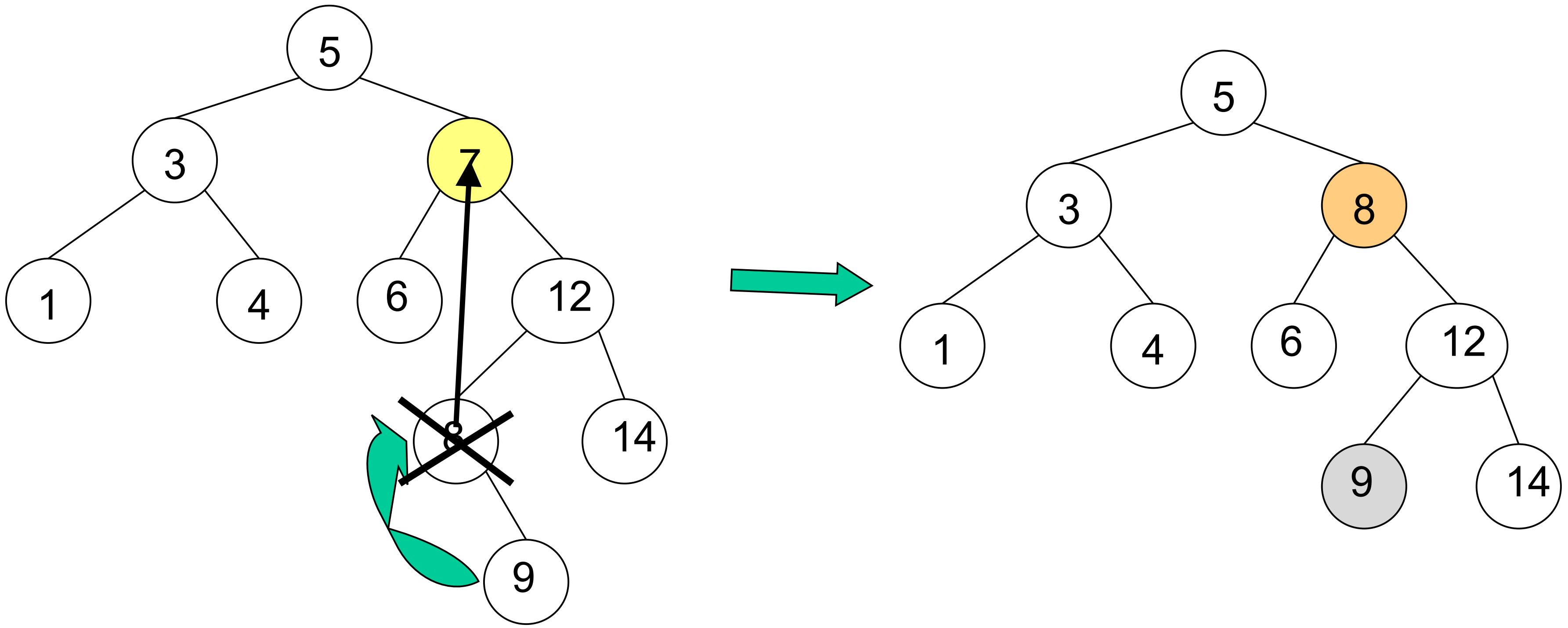
- Quando cancelliamo un oggetto *z* da un albero, abbiamo 3 possibili casi (a seconda che *z* sia una foglia o un nodo interno):
  - il nodo *z* da cancellare *non ha sottoalberi*
  - il nodo *z* da cancellare *ha 1 sottoalbero*
  - il nodo *z* da cancellare *ha 2 sottoalberi*
- Il caso 1 è quello più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



- Nel caso 2, dobbiamo spostare l'intero sottoalbero di *z* su di un livello:



- 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
    - si noti che 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*)



Delete: pseudocodice

```
TREE-DELETE(T, z)
if z.left = NIL or z.right = NIL
  y := z
else y := TREE-SUCCESSOR(z)
if y.left ≠ NIL
  x := y.left
else x := y.right
if x ≠ NIL
  x.p := y.p
if y.p = NIL
  T.root := x
elsif y = y.p.left
  y.p.left := x
else y.p.right := x
if y ≠ z
  z.key := y.key
return y
```

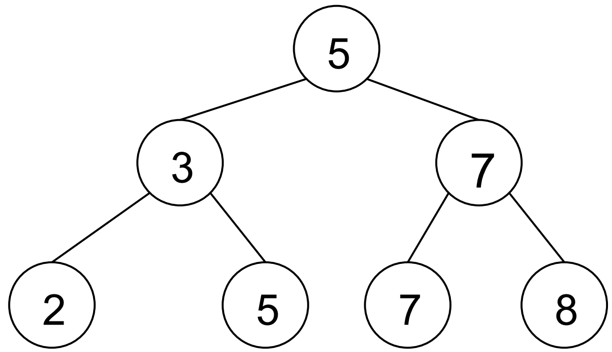
Nota:
y è il nodo da eliminare
x è quello con cui lo sostituiamo

- 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 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*

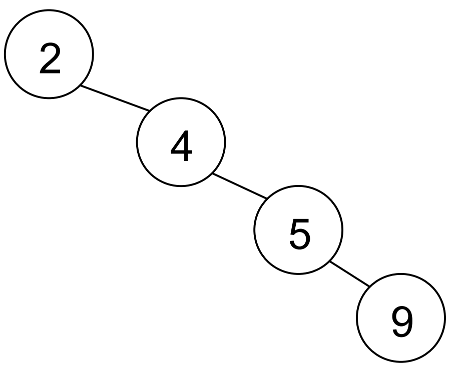
Analisi di complessità

- 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)?

- 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



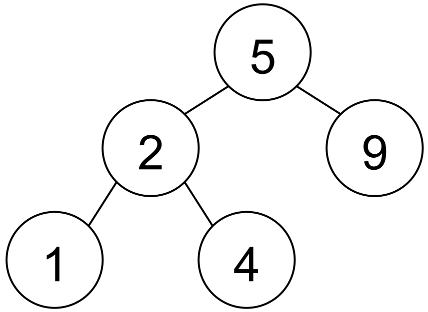
- Nel caso pessimo, però, che si ha se tutti i nodi sono “in linea”, abbiamo $h = \Theta(n)$



Analisi di complessità (2)

- 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:



- Ci sono diverse tecniche per mantenere un albero bilanciato:
  - alberi *rosso-neri* (red-black)
  - alberi AVL
  - etc.
- Inoltre, si può dimostrare che l'altezza attesa di un albero è O(log(n)) se le chiavi sono inserite in modo casuale, con distribuzione uniforme

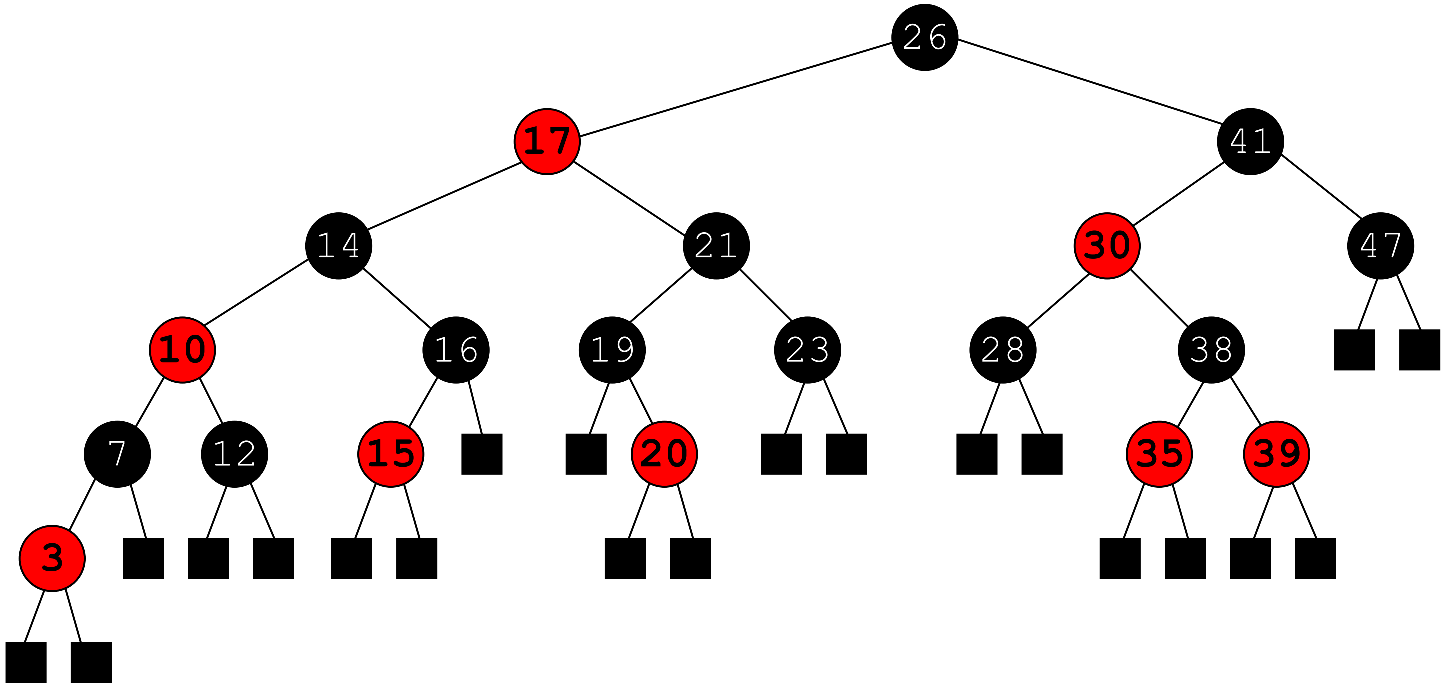
Alberi rosso-neri (red-black)

- 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

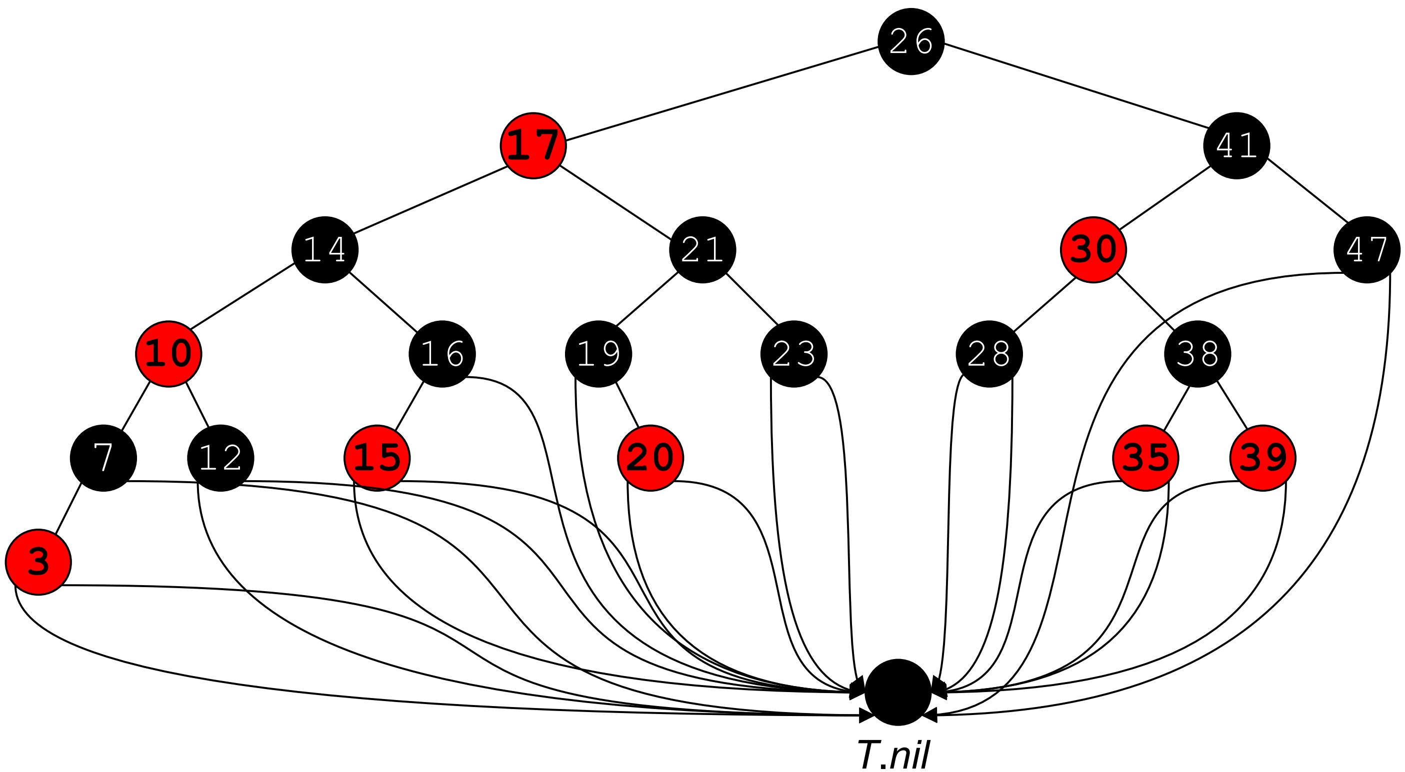
- Ogni nodo di un albero RB ha 5 attributi: *key*, *left*, *right*, *p*, e *color*
  - convenzione: le foglie sono i nodi NIL, tutti i nodi non NIL (che hanno quindi una chiave associata) sono nodi interni

- 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



- 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*

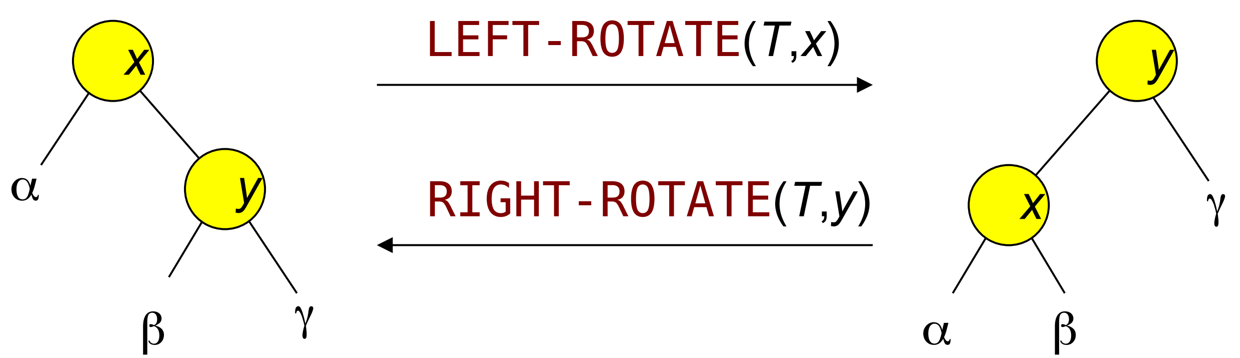


Proprietà degli alberi RB

- Un albero rosso-nero 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 (sostituendo al punto precedente) $n \geq 2^{h/2}-1$, 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` 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)
y := x.right
x.right := y.left  //il sottoalbero sinistro di y
                   //diventa quello destro di x
if y.left ≠ T.nil
  y.left.p := x
y.p := x.p         //attacca il padre di x a y
if x.p = T.nil
  T.root := y
elsif x = x.p.left
  x.p.left := y
else x.p.right := y
y.left := x        //mette x a sinistra di y
x.p := y
```

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

`RB-INSERT`

- L'inserimento è fatto in modo analogo a quello dei BST, ma alla fine occorre ristabilire le proprietà dagli alberi RB se queste sono state violate
  - per ristabilire le proprietà di 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(T, z)
  y := T.nil          // y padre del nodo considerato
  x := T.root         // nodo considerato
  while x ≠ T.nil
    y := x
    if z.key < x.key
      x := x.left
    else x := x.right
  z.p := y
  if y = T.nil
     T.root := z   //l'albero T e' vuoto
  elsif z.key < y.key
     y.left := z
  else y.right := z
  z.left := T.nil
  z.right := T.nil
  z.color := RED
  RB-INSERT-FIXUP(T, z)
```

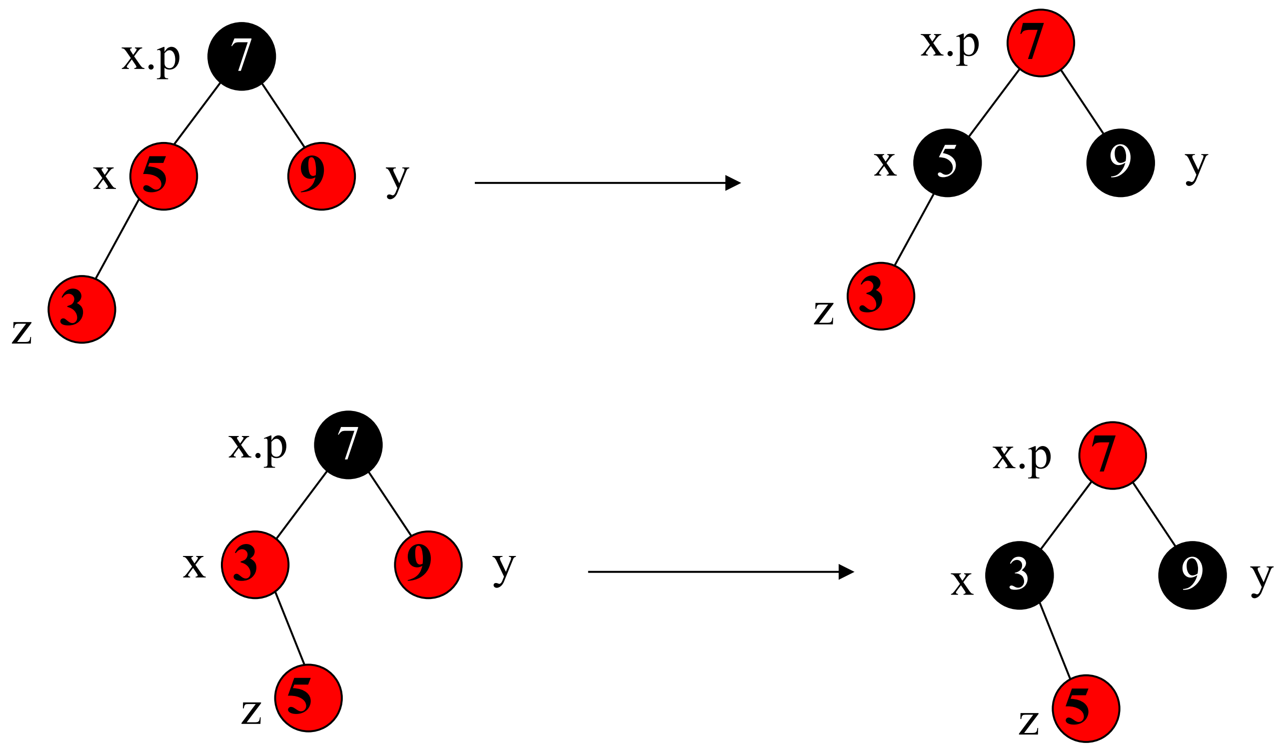
```
RB-INSERT-FIXUP(T, z)
1  if z = T.root
2    T.root.color = BLACK
3  else x := z.p                 // x e' il padre di z
4    if x.color = RED
5      if x = x.p.left           // se x e' figlio sin.
6        y := x.p.right          // y e' fratello di x
7        if y.color = RED
8          x.color := 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”↔“left”)
```

- 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:

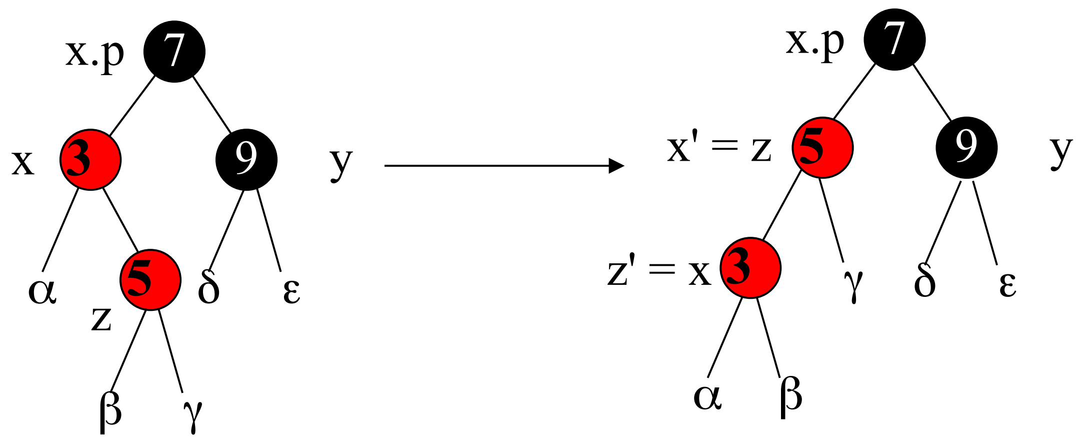
Funzionamento:

- Caso 1: *y* rosso



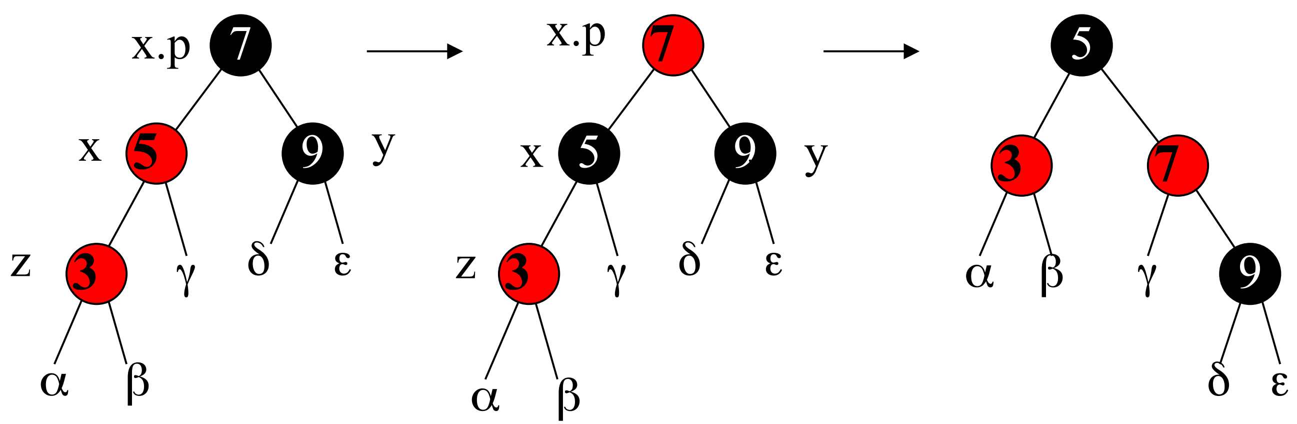
- quindi ripeto la procedura su *x.p*, in quanto il padre di *x.p* potrebbe essere di colore rosso, nel qual caso la proprietà 4 degli alberi RB non sarebbe (ancora) verificata

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



- a questo punto ci siamo messi nel caso 3

- 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(T, z)
if z.left = T.nil or z.right = T.nil
  y := z
else y := TREE-SUCCESSOR(z)
if y.left ≠ T.nil
  x := y.left
else x := y.right
x.p := y.p          // x potrebbe essere T.nil
if y.p = T.nil
  T.root := x
elsif y = y.p.left
  y.p.left := x
else y.p.right := x
if y ≠ z
  z.key := y.key
if y.color = BLACK
  RB-DELETE-FIXUP(T,x)
return y
```

- 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

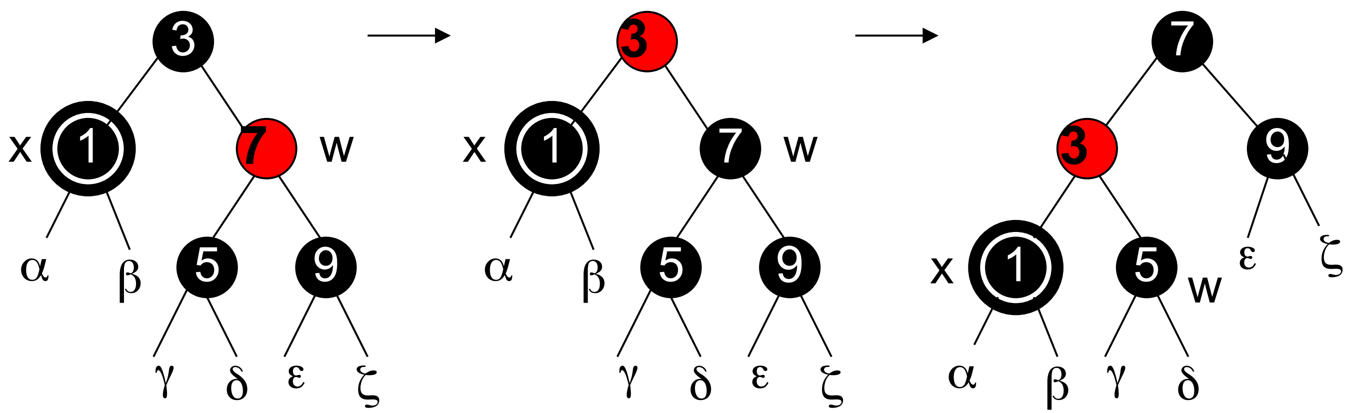
```
RB-DELETE-FIXUP(T, x)
1  if x.color = RED or x.p = T.nil
2    x.color := BLACK                 // Caso 0
3  elsif x = x.p.left                 // x e' figlio sinistro
4    w := x.p.right                   // w e' fratello di x
5    if w.color = RED
6      w.color := BLACK               // Caso 1
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
19 else (come 4-21, scambiando “right”↔“left”)
```

Idea: il nodo x passato come argomento si porta dietro un “nero in più”, che, per fare quadrare i conti, può essere eliminato solo a certe condizioni

Funzionamento di RB-DELETE-FIXUP

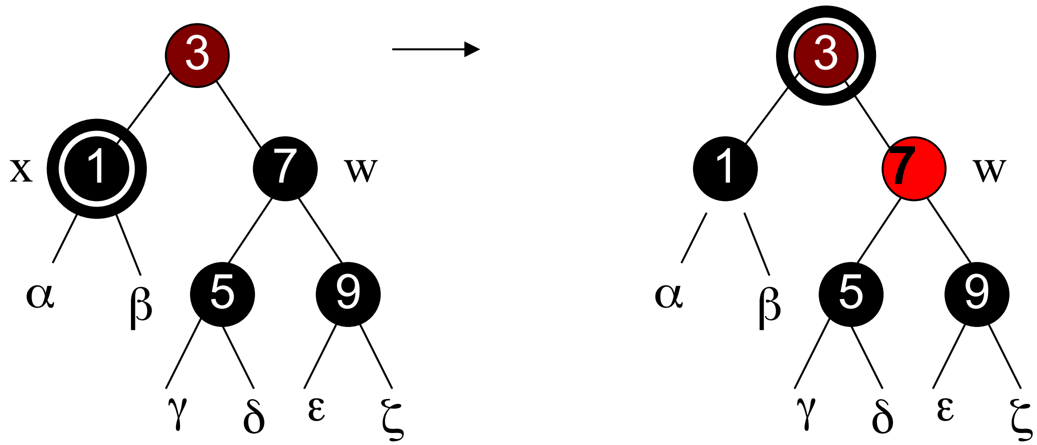
- Caso 0: *x* è un nodo rosso, oppure è la radice

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



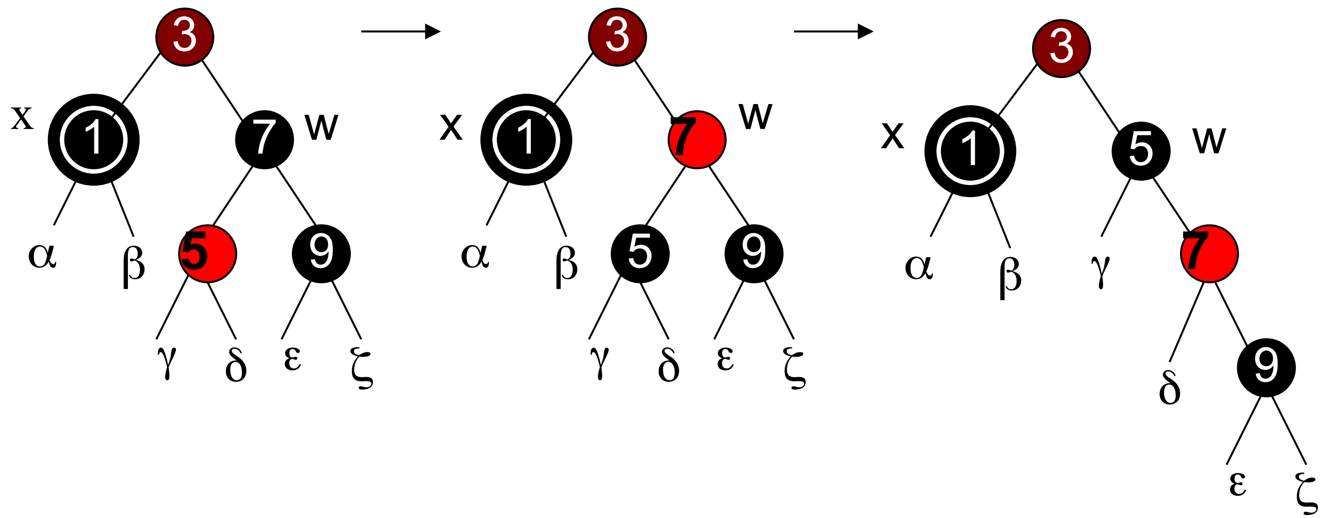
  - diventa o il caso 2, o il caso 3, o il caso 4

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



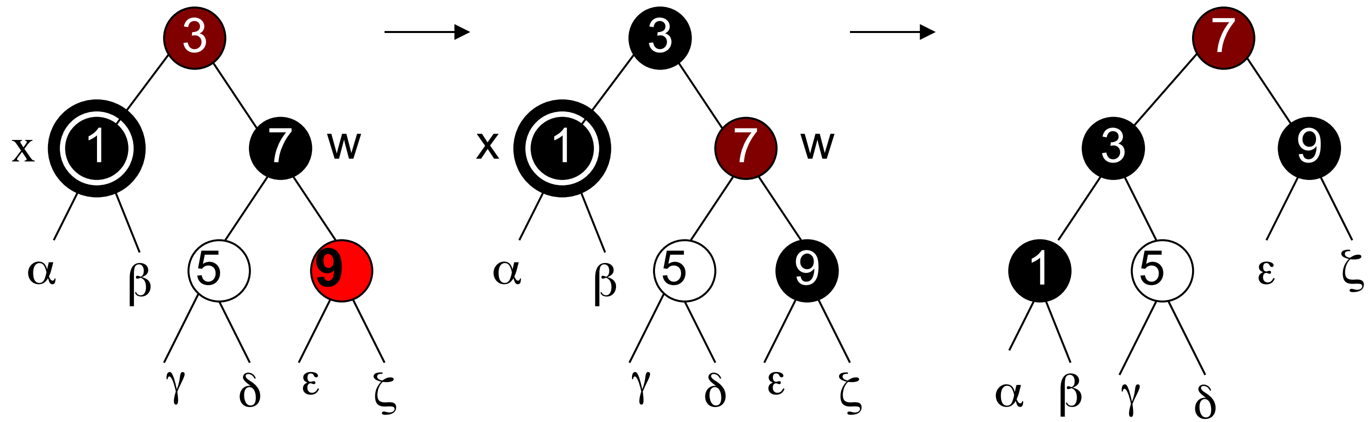
- se arriviamo al caso 2 dal caso 1, allora *x.p* è rosso, e quando `RB-DELETE-FIXUP` viene invocato su di esso termina subito (arriva subito al caso 0)

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



diventa il caso 4

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

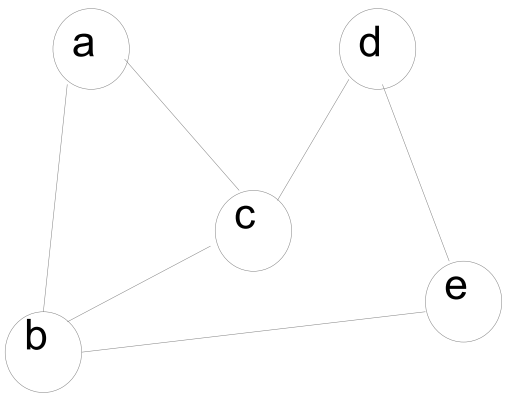


- Ogni volta che `RB-DELETE-FIXUP` viene invocato esso può o terminare (casi 0, 1, 3 e 4), o venire applicato ricorsivamente risalendo un livello nell'albero (caso 2 non proveniente da 1)
  - quindi può essere invocato al massimo O(*h*) volte, cioè O(log(*n*))
  - si noti che una catena di invocazioni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

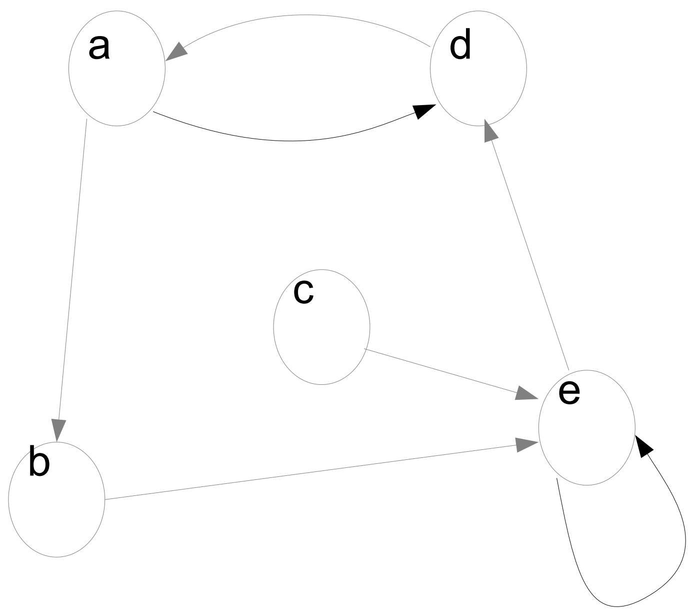
- Ci sono 2 tipi di grafi: *orientati* e *non orientati*
  - in un grafo non orientato, un arco (*u*, *v*) è lo stesso di (*v*, *u*) (non c'è nozione di direzione da un nodo all'altro)
  - In un grafo orientato(*u*, *v*) "va dal" nodo *u* al nodo *v*, ed è diverso da (*v*, *u*)



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

  - L'ordine dei vertici negli archi è irrilevante

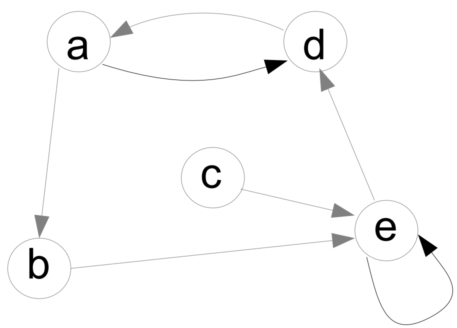
esempio di grafo orientato

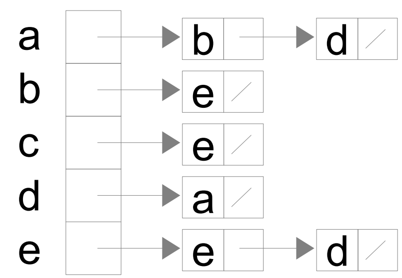


V = {a, b, c, d, e}
E = {(a, b) (a, d) (d, a) (b, e)
(c, e) (e, d), (e, e)}

Rappresentazione di grafi in memoria

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





Liste ad.

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

$$
\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array}
\begin{array}{c}
\begin{array}{ccccc} a & b & c & d & e \end{array} \\
\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}
\end{array}
$$

Matrice ad.

- 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 $\Theta(|V|^2)$

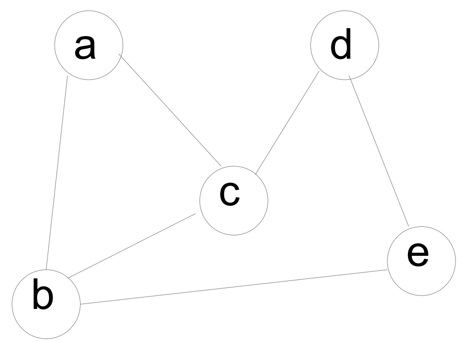
- Le liste di adiacenza sono in generale migliori quando $|E| \neq \Theta(|V|^2)$, cioè quando il grafo è *sparso* (quando il numero di nodi connessi “non è tanto grande”)
  - si ricordi che $|E| \leq |V|^2$, cioè $|E| = O(|V|^2)$
- Se il grafo è *completo* (o quasi), tanto vale usare una matrice di 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

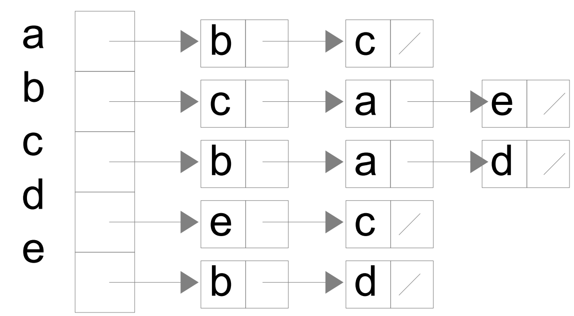
- Quale è la complessità temporale per determinare se un arco (*u*, *v*) appartiene al grafo:
  - quando il grafo è rappresentato mediante liste di adiacenza?
  - quando il grafo è rappresentato con una matrice di adiacenza?
- Quale è la complessità temporale per determinare il numero di archi che escono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

Rappresentazione di grafi (cont.)

- 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*

- es:





$$\begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 0 & 0 \\ b & 1 & 0 & 1 & 0 & 1 \\ c & 1 & 1 & 0 & 1 & 0 \\ d & 0 & 0 & 1 & 0 & 1 \\ e & 0 & 1 & 0 & 1 & 0 \end{array}$$

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

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

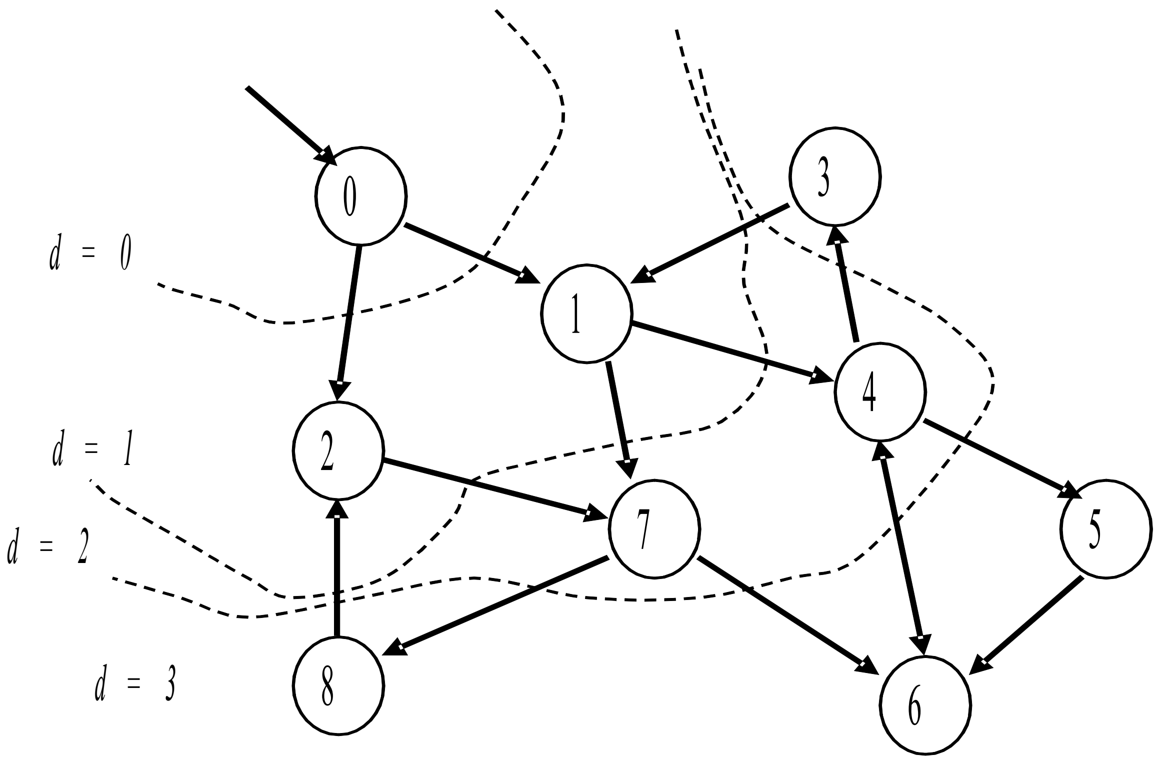
Visita in ampiezza (Breadth-First Search)

- 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 $u.dist$

- 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

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: pseudocodice

```
BFS(G, s)
 for each u ∈ G.V – {s}
   u.color := WHITE
   u.dist := ∞
s.color := GREY
s.dist := 0
Q := ∅
ENQUEUE(Q, s)
while Q ≠ ∅
   u := DEQUEUE(Q)
   for each v ∈ u.Adj
       if v.color = WHITE
          v.color := GRAY
          v.dist := u.dist +1
          ENQUEUE(Q, v)
   u.color := BLACK
```

- 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 lui 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.

- 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)

DFS: considerazioni

- 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
- 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 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) di un nodo, ed il “tempo” di quando la scoperta termina
    - l'inizio della scoperta di un nodo *u* è memorizzata nell'attributo *u.d*, mentre la sua fine nell'attributo *u.f*

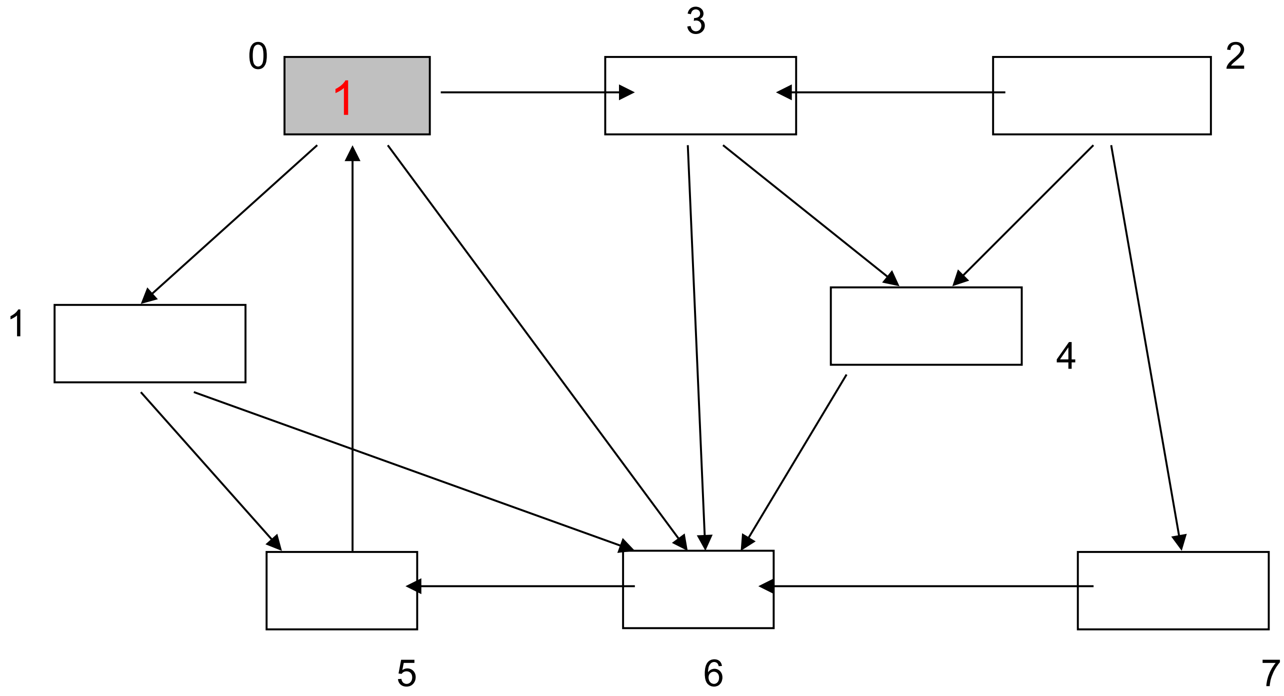
```
DFS(G)
1 for each u ∈ G.V
2   u.color := WHITE
3 time := 0
4 for each u ∈ G.V
5   if u.color = WHITE
6     DFS-VISIT(u)
```

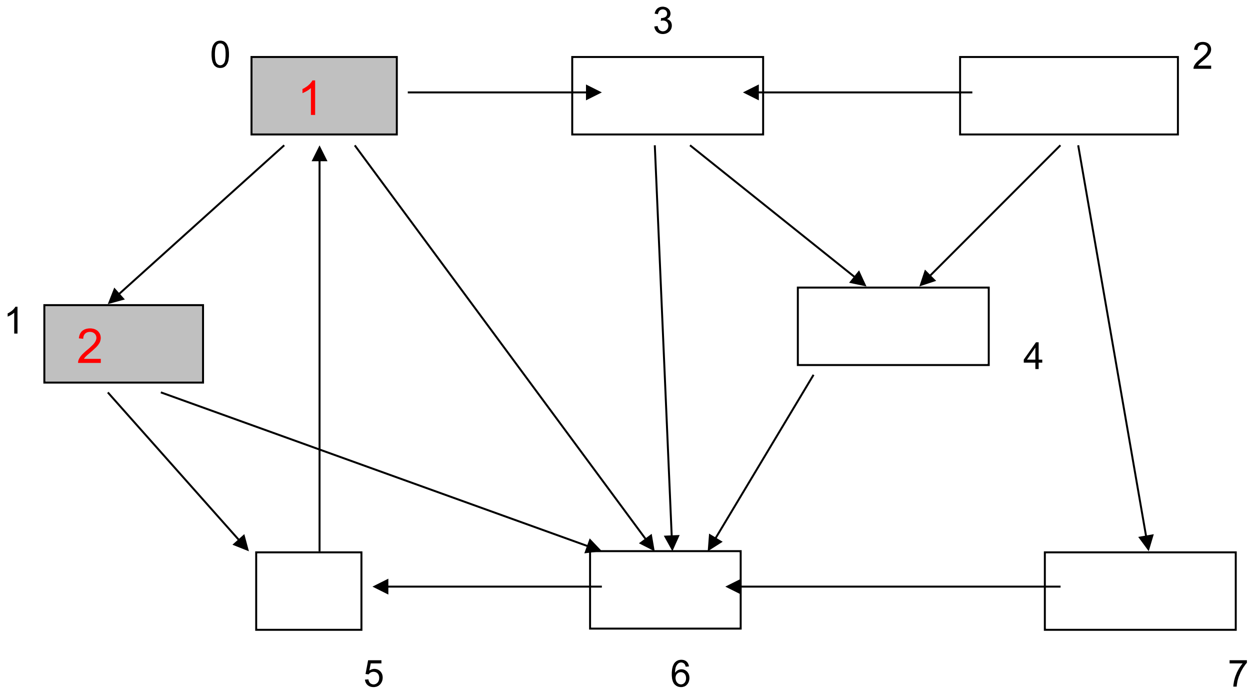
```
DFS-VISIT(u)
1 u.color := GRAY
2 time := time + 1
3 u.d := time
4 for each v ∈ u.Adj
5   if v.color = WHITE
6     DFS-VISIT(v)
7 u.color := BLACK
8 u.f := time := time + 1
```

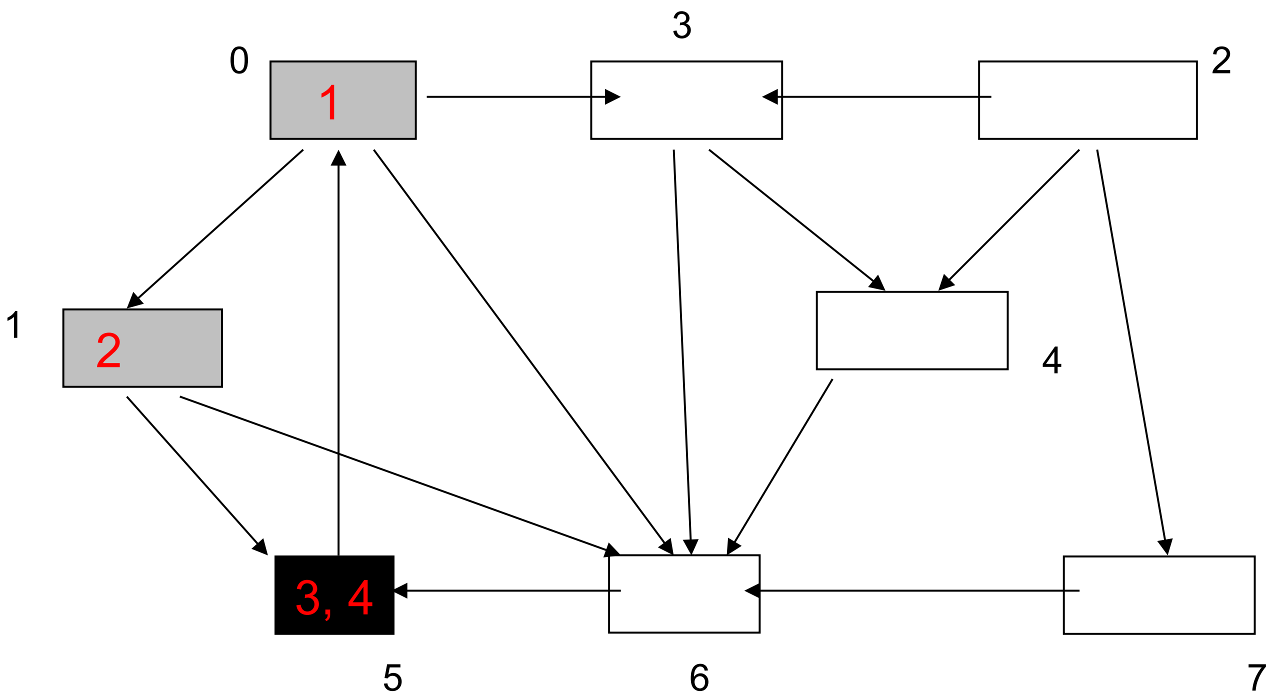
- Le linee 1-3 di `DFS` inizializzano i nodi colorandoli tutti di bianco, e mette il tempo a 0: tempo di esecuzione $\Theta(|V|)$
- L'algoritmo `DFS-VISIT` è ripetuto (linee 4-6 di `DFS`) fino a che non ci sono più nodi da visitare
  - come in BFS, ogni nodo è "messo sullo stack" (che in questo caso 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`, quindi queste prendono tempo $\Theta(|E|)$
- In tutto, la complessità di `DFS` è $\Theta(|V| + |E|)$

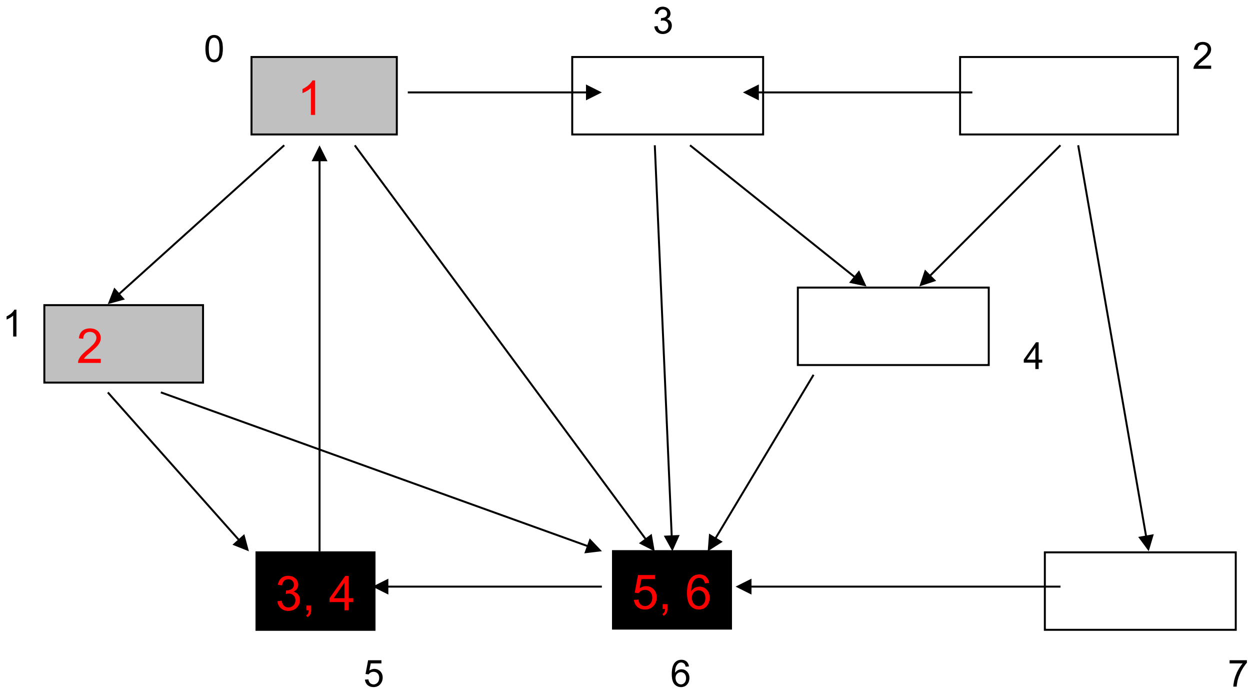
Esempio:

- per ogni nodo: d, f

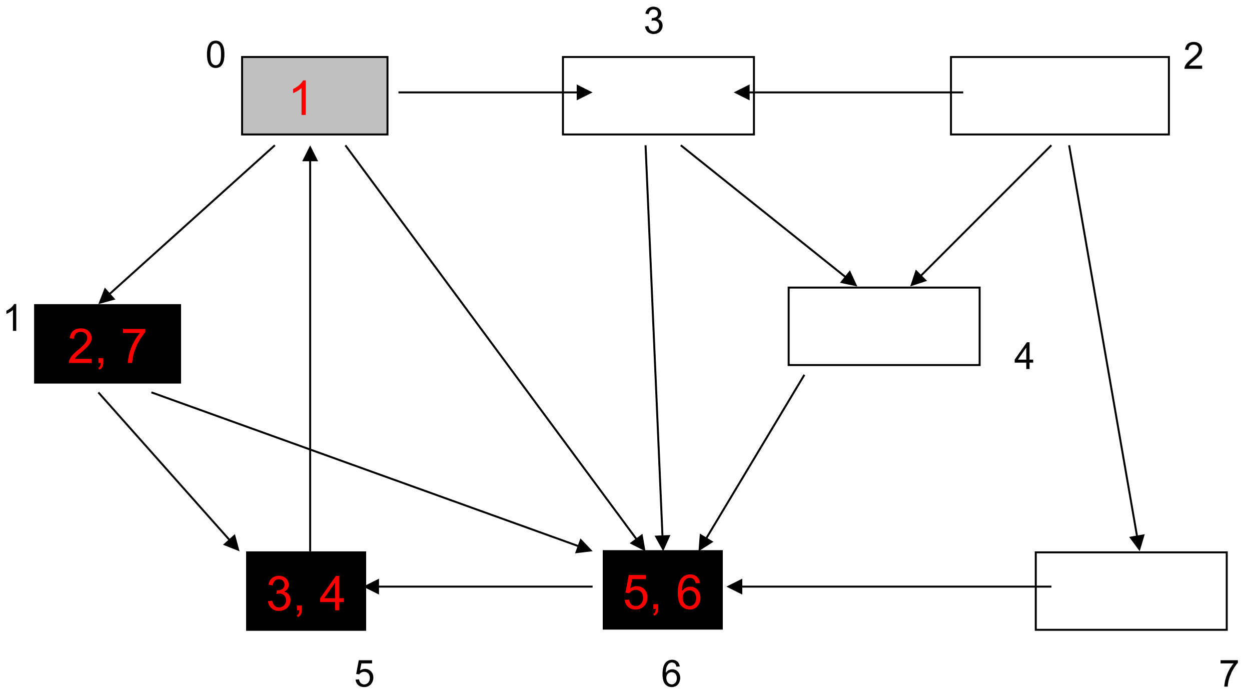


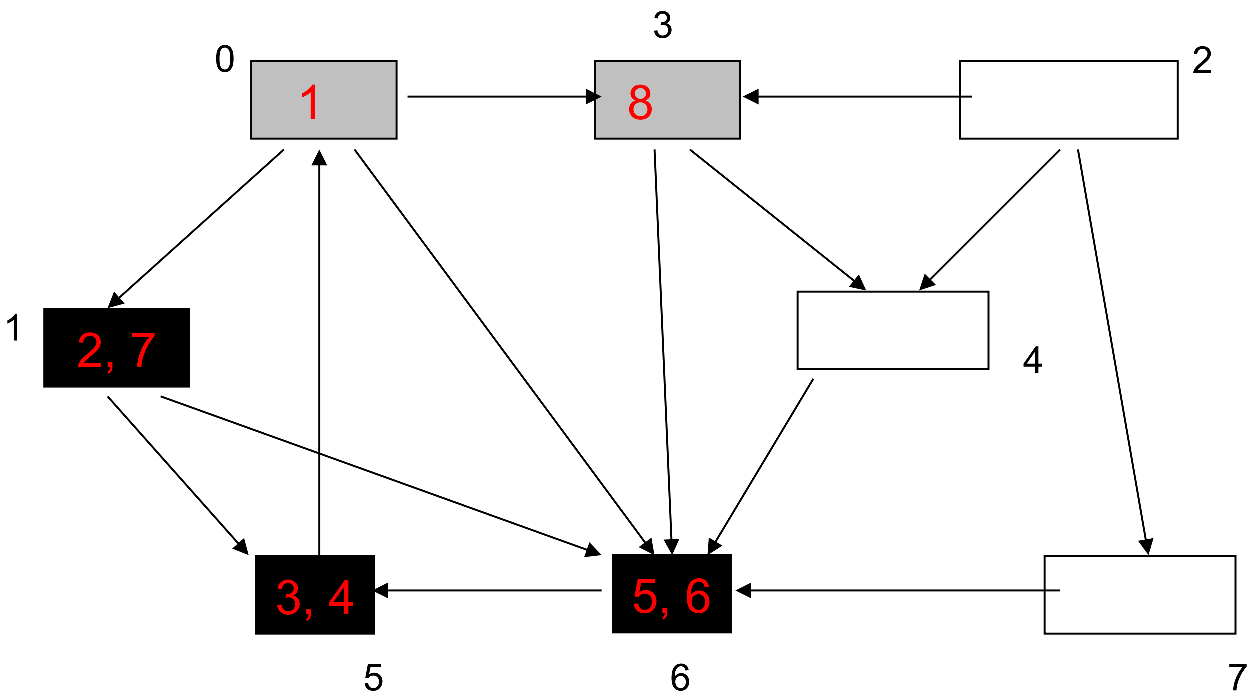

0
1
3
2
1
2
4
5
6
7


0
1
3
2
1
2
4
3, 4
5
6
7

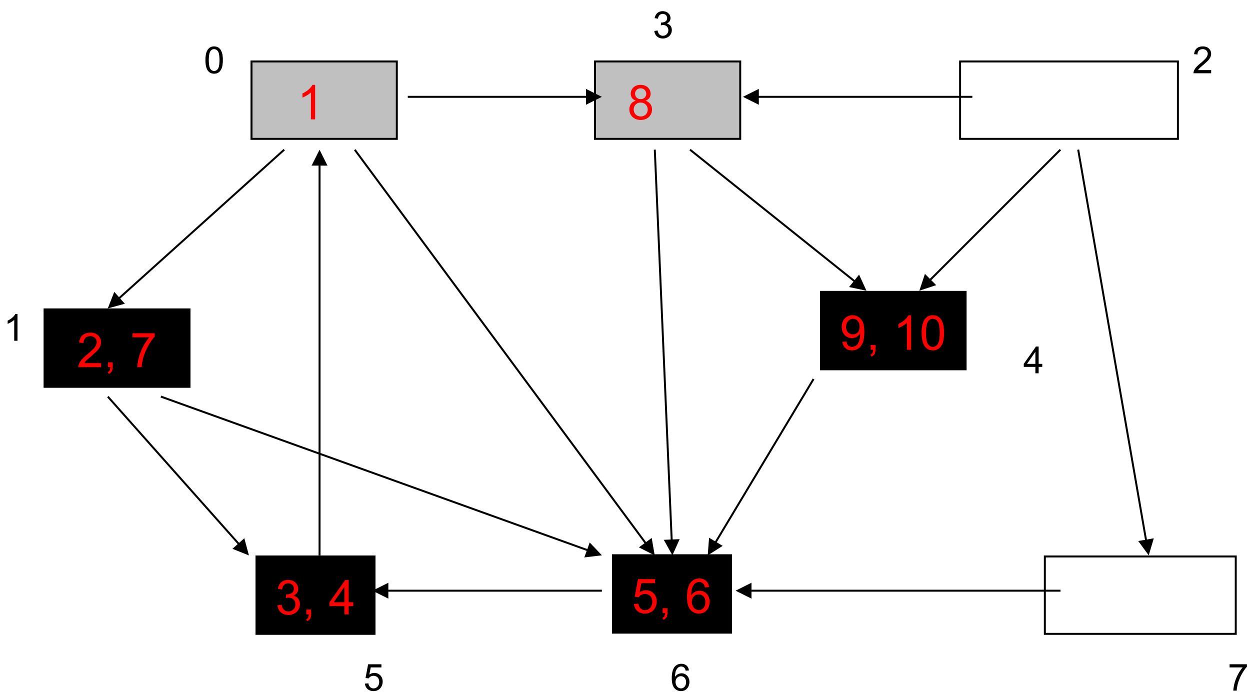


0
1
3
2
1
2
4
3, 4
5, 6
5
6
7

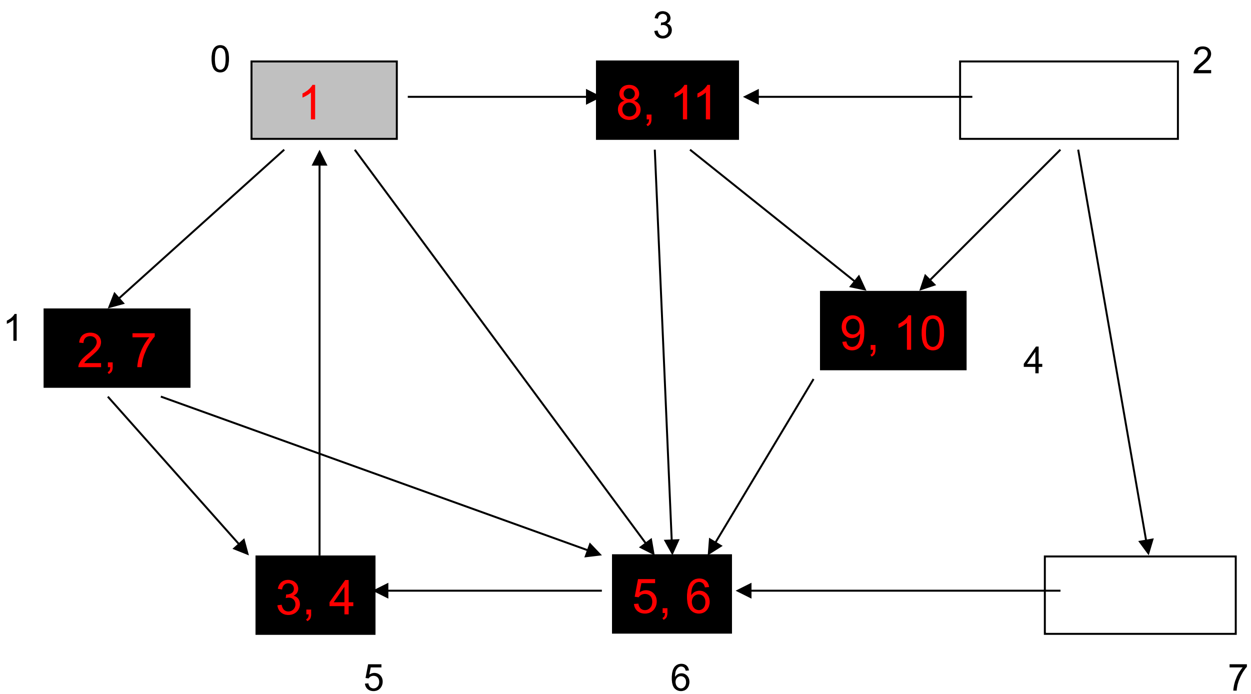

0
1
3
2
1
2, 7
4
3, 4
5, 6
5
6
7

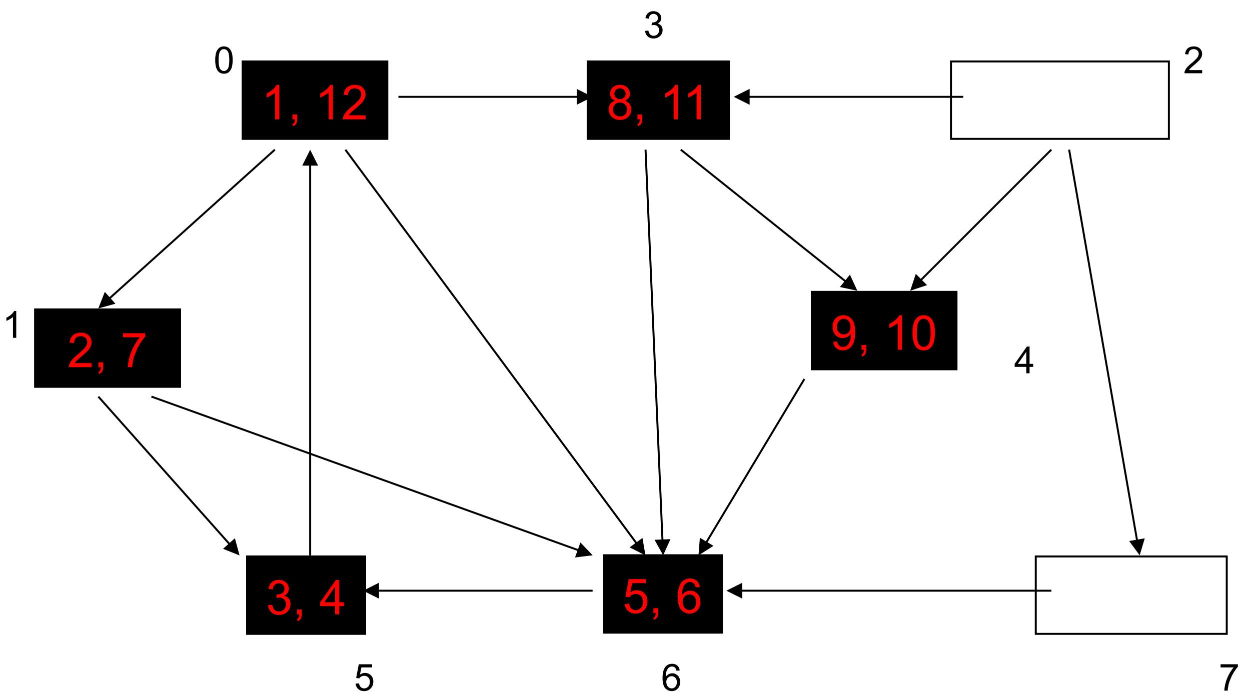


0
1
3
8
2
1
2, 7
4
3, 4
5, 6
5
6
7

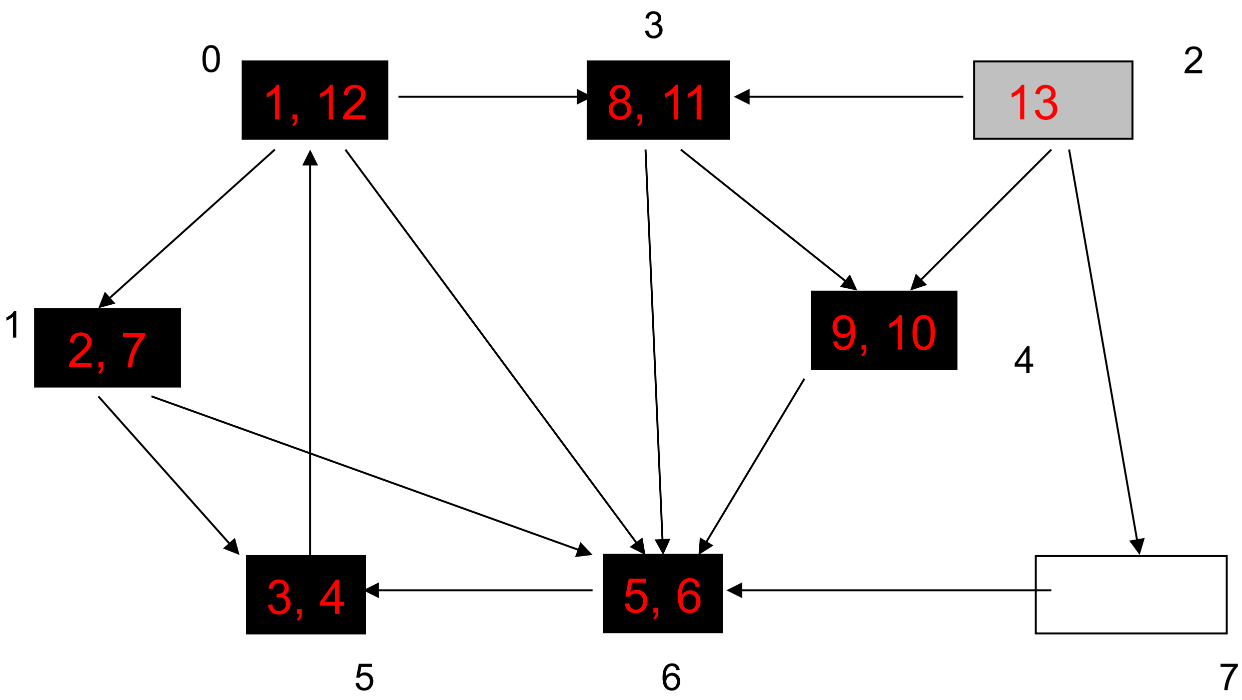


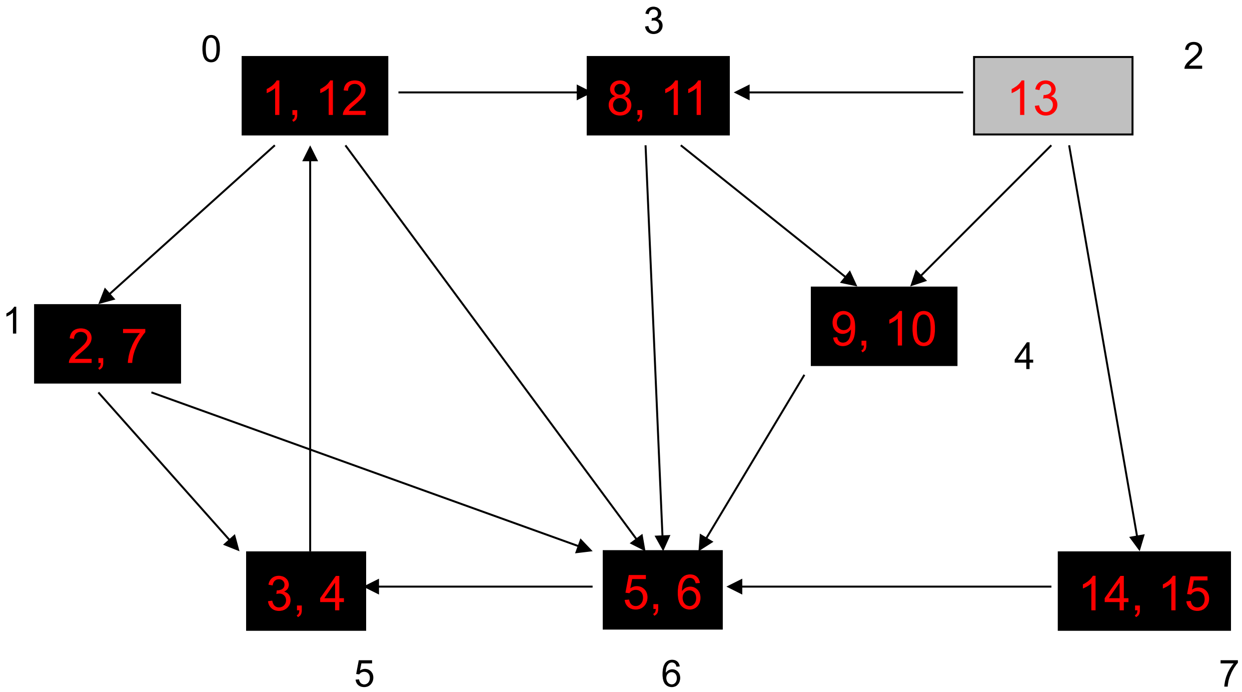
0
1
3
8
2
1
2, 7
9, 10
4
3, 4
5, 6
5
6
7

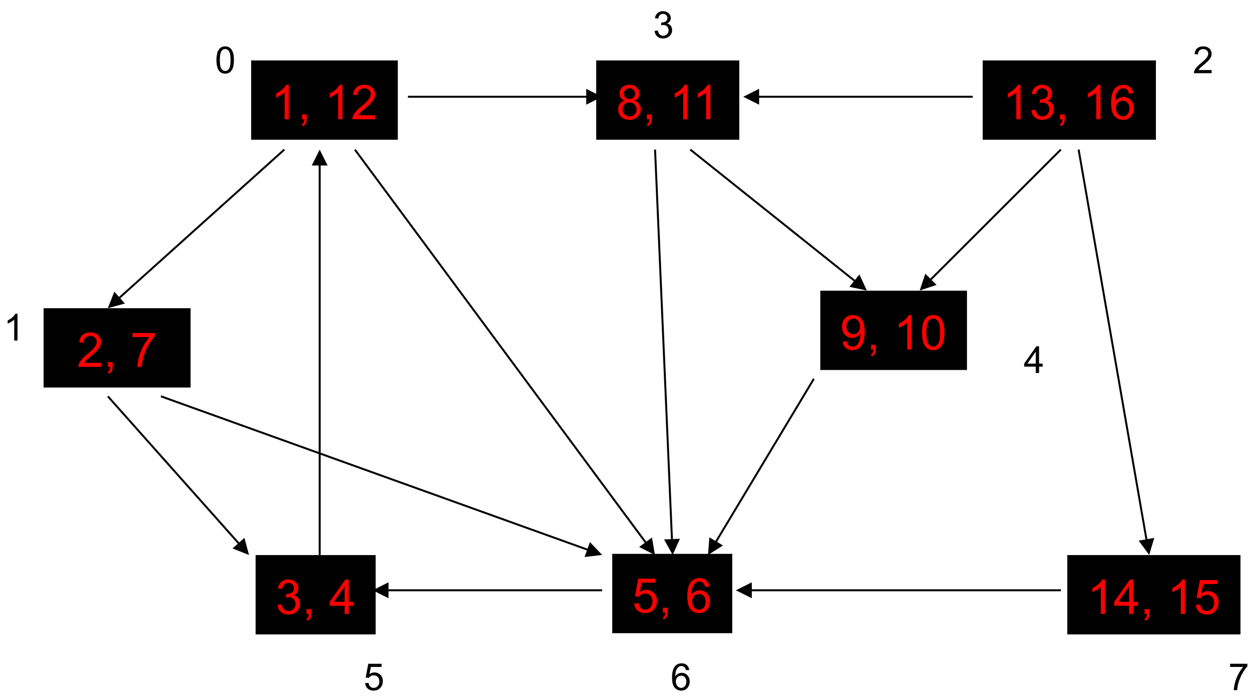

0
1
3
8, 11
2
1
2, 7
9, 10
4
3, 4
5, 6
5
6
7



0
1, 12
3
8, 11
2
1
2, 7
9, 10
4
3, 4
5, 6
5
6
7

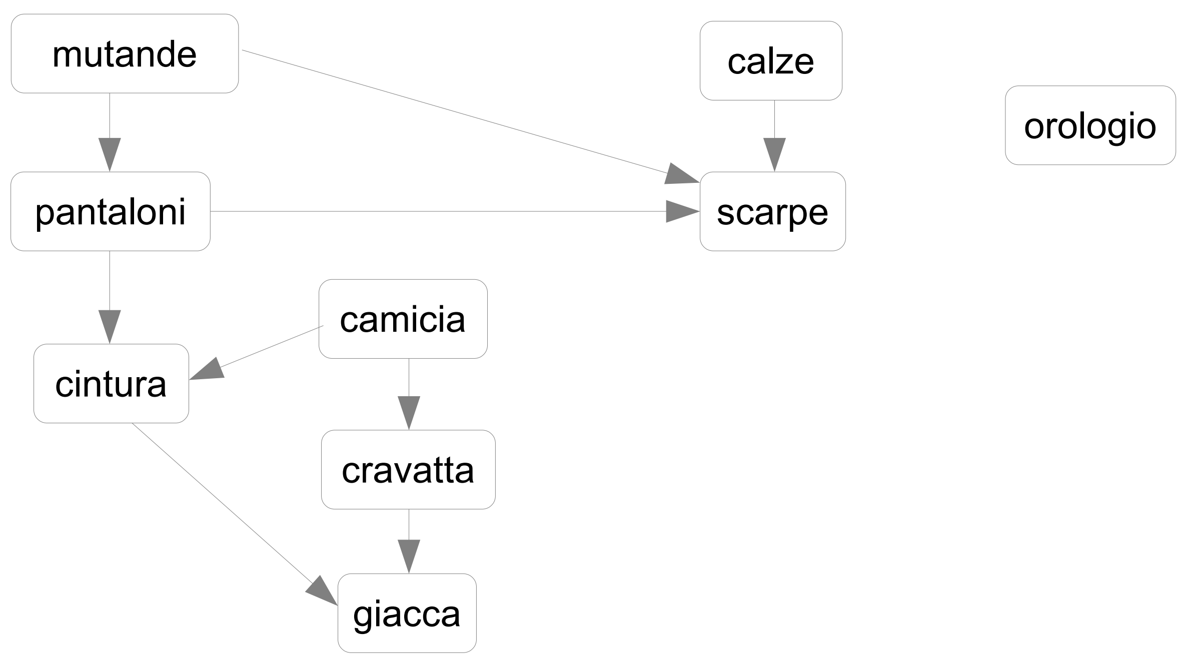

0
1, 12
3
8, 11
13
2
1
2, 7
9, 10
4
3, 4
5, 6
5
6
7


0
1, 12
3
8, 11
13
2
1
2, 7
9, 10
4
3, 4
5, 6
14, 15
5
6
7


0
1, 12
3
8, 11
13, 16
2
1
2, 7
9, 10
4
3, 4
5, 6
14, 15
5
6
7

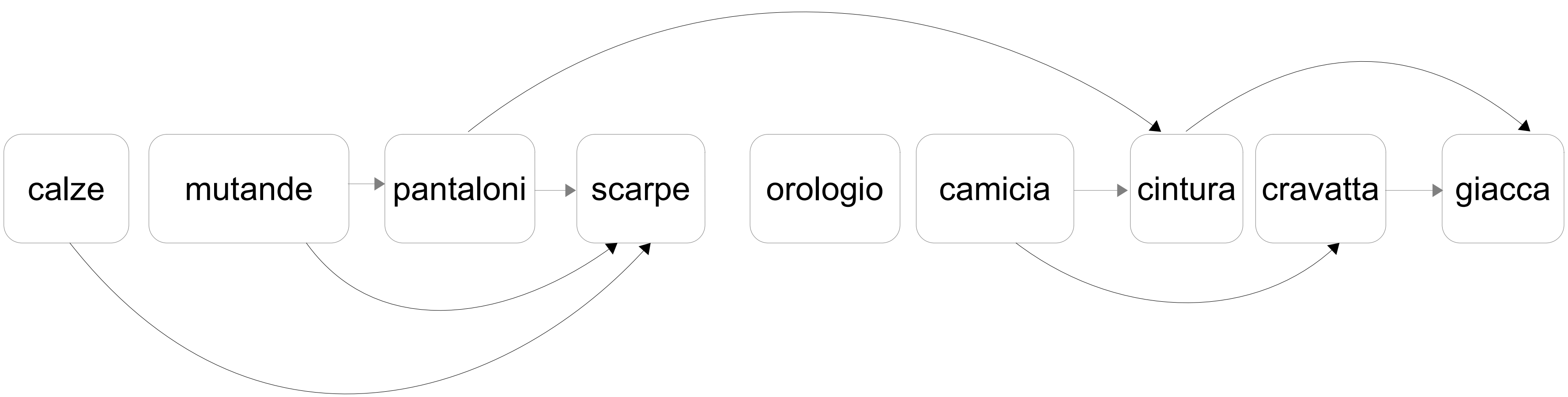
Applicazione: Ordinamento Topologico

- Supponiamo di avere un grafo orientato aciclico che rappresenta le precedenze tra eventi
- Come questo, per esempio



Ordinamento topologico (2)

- 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 primo DAG della slide precedente potrebbe dare il seguente ordinamento



- si noti che questo non è l'unico possibile ordinamento ammissibile...

- Di fatto, un ordinamento topologico restituisce un ordinamento totale 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*
    - si ricordi che una lista è un ordinamento lineare, in cui l'ordine è dato da come gli oggetti nella lista sono connessi tra loro
- 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

```
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

Approfondimento: cammini minimi

- Si vuole ottenere il cammino minimo tra due nodi
  (es. problema delle distanze tra stazioni ferroviarie)

Ingresso: G = (V, E) diretto, funzione di peso w : E → **R**

- Peso di un cammino:

$$p = \langle v_0, v_1, \ldots, v_k \rangle$$

$$W(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- peso cammino minimo da u a v:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\}, \text{se esiste cammino p da u a v} \\ \infty, \text{altrimenti} \end{cases}$$

- Ok pesi negativi, ma non cicli negativi! (come mai?)
- Inoltre calcoliamo i cammini minimi da un nodo fissato *s* (detto *sorgente*) (anch'esso fornito in ingresso)

Uscita: $\forall v \in V$:

- d[v] = δ(s,v)
  - all'inizio d[v] = ∞
  - viene ridotto al procedere dell'algoritmo. Però sempre d[v] >= δ(s,v)
  - d[v] viene anche detto *stima di cammino minimo* (dalla sorgente s a v)
- Pi[v] = predecessore di v nel cammino min da s
  - se non esiste: = NIL

- Gli algoritmi di questo tipo si basano sul concetto di *rilassamento*

Rilassamento di un lato (u,v):

- se d[v] > d[u]+w(u,v) allora
  d[v] := d[u]+w(u,v) e
  Pi[v] := u
- (cioè vediamo che "costa di meno" passare per il lato (u,v))

```
RELAX(u, v, adj, d, Pi)
    if d[v] > d[u] + adj[u][v]:
        d[v] := d[u] + adj[u][v]
        Pi[v] := u
```

Algoritmo di Bellman-Ford

```
BELLMAN-FORD(adj, s)
   V := vector of nodes of adj
   allocate vectors d, Pi of size adj

   for i from 0 to |adj|-1
      d[i] := ∞

   d[s] := 0
   repeat for |V|-1 times
      for u in V
         for v in adj[u]
            RELAX(u, v, adj, d, Pi)

   return [d, Pi]
```

In pratica rilasso, un passo alla volta, partendo da s.
Ad ogni passo avanzo di un passo nei miei cammini –
al |V|-1-esimo passo sicuramente avrò toccato tutti i nodi raggiungibili
(chiaramente non converge se ci sono cicli negativi…)

Esempio:

Adj = [{1:2, 2:2, 3:3}, {2:-3}, {0:1, 3:5}, {}, {0:-1,3:4}]

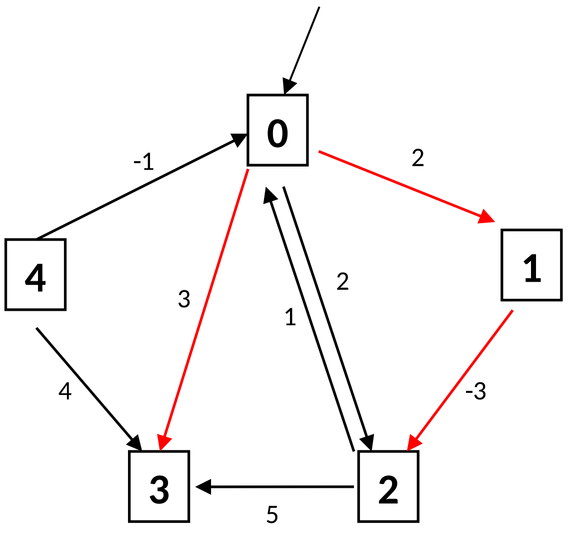
d, Pi = bellmanFord(Adj, 0);

Distanze (4 è irraggiungibile):

d: [0, 2, -1, 3, ∞]

Predecessori (codifica i cammini minimi):

Pi: [NIL, 0, 1, 0, NIL]



Costo: ciclo su tutte le adiacenze innestato in un ciclo su |V|; relax costo costante
=> $\Theta(|V||E|)$