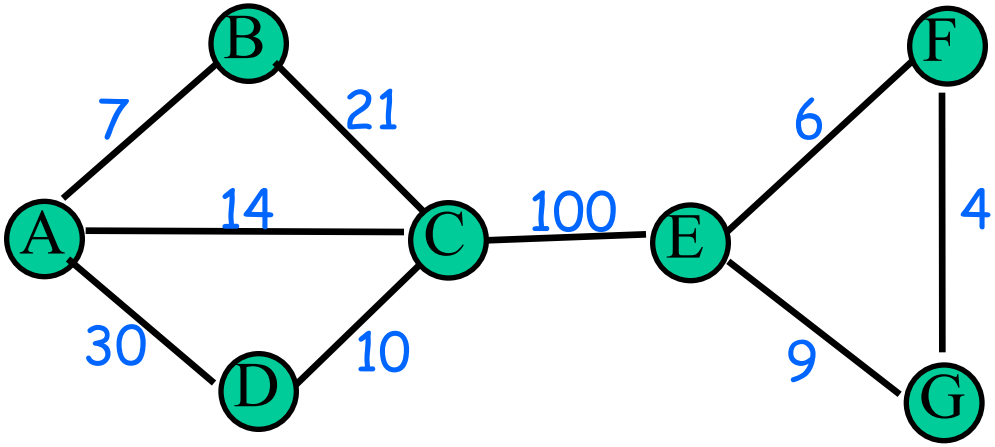
Algoritmi e Strutture Dati

Capitolo 12
Minimo albero ricoprente:
Algoritmo di Prim

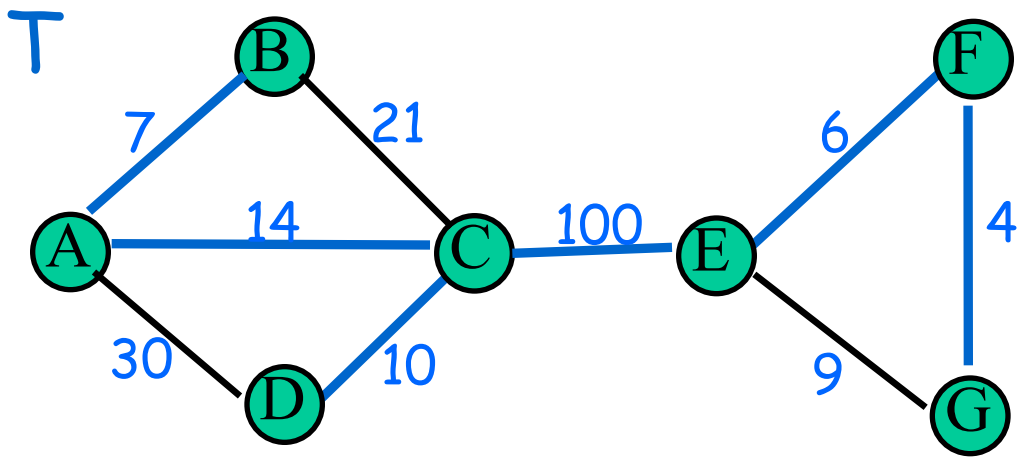
Il problema del calcolo di un Minimum Spanning Tree (MST)

- Input:
  - un grafo non orientato e pesato $G=(V,E,w)$
- Soluzione ammissibile:
  - un albero di copertura (uno spanning tree) di G, ovvero un albero $T=(V,F)$ con $F \subseteq E$
- Misura della soluzione (da minimizzare):
  - costo di T: $\sum_{e \in F} w(e)$

Un esempio



Un esempio



Riepilogo: regole del taglio e del ciclo

Scegli un taglio del grafo che non è attraversato da **archi blu**. Tra tutti gli archi non ancora colorati che attraversano il taglio, scegline uno di costo minimo e coloralo di blu (cioè, **aggiungilo alla soluzione**).

Scegli un ciclo nel grafo che non contiene **archi rossi**. Tra tutti gli archi non ancora colorati del ciclo, scegline uno di costo massimo e coloralo di rosso (cioè, **scartalo per sempre**).

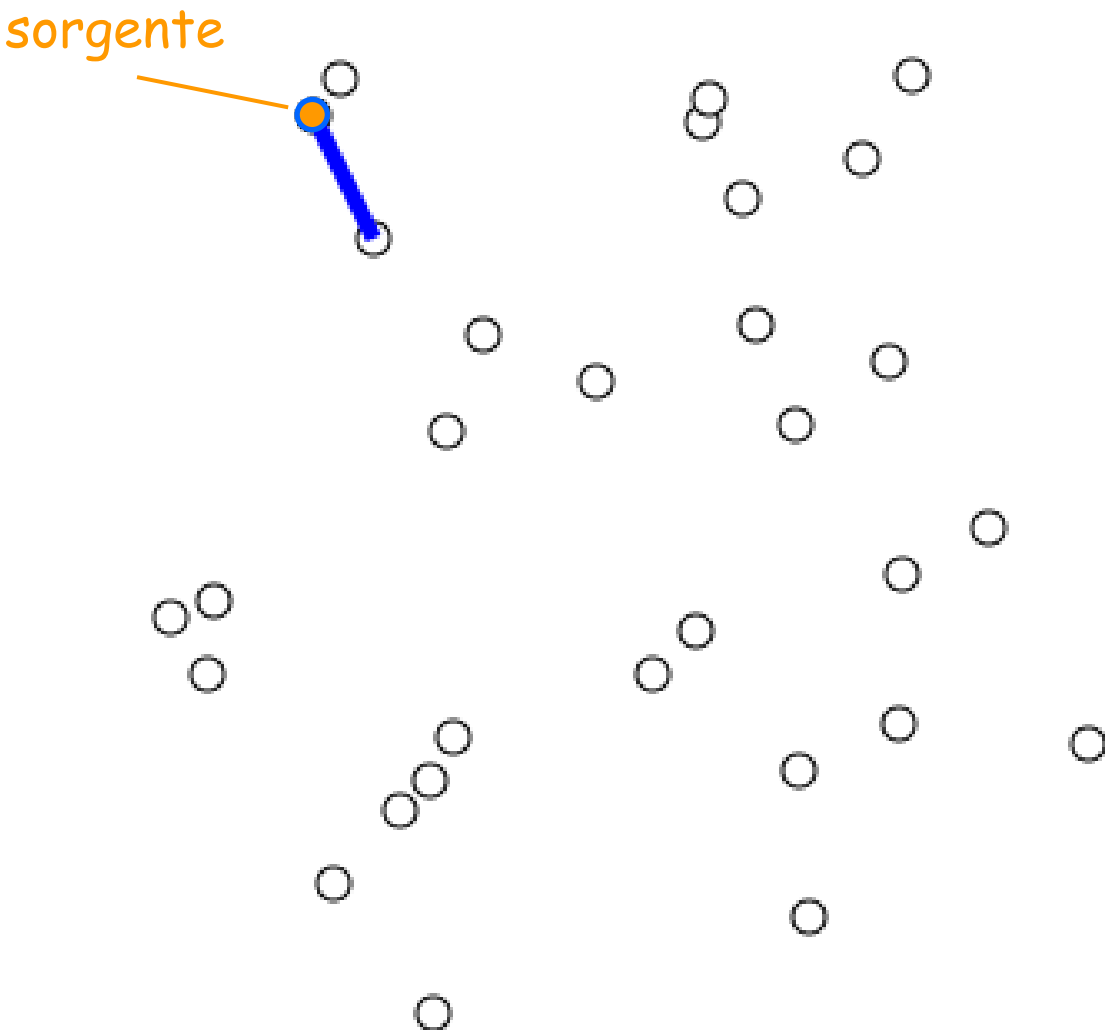
Algoritmo di Prim (1957)
(in realtà scoperto nel 1930 da Jarník)

Idea:
mantiene un albero blu T (inizialmente è un vertice arbitrario)

Applica n-1 volte il seguente passo:

scegli arco di peso minimo che attraversa il taglio (V(T), V\V(T) e coloralo di blu

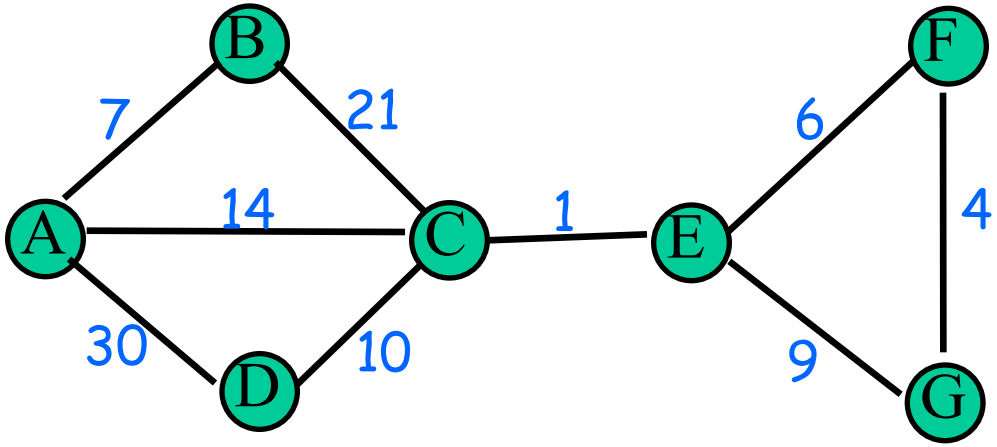
} regola del taglio, da cui segue la correttezza



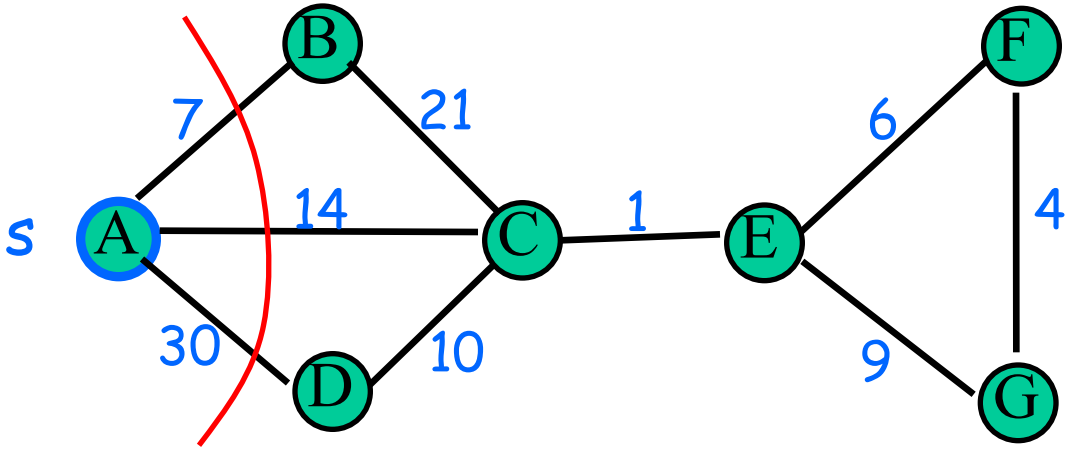
grafo completo euclideo:

nodi disposti su un piano;
per ogni coppia di nodi u e v il peso dell'arco (u,v) è la distanza euclidea fra u e v.

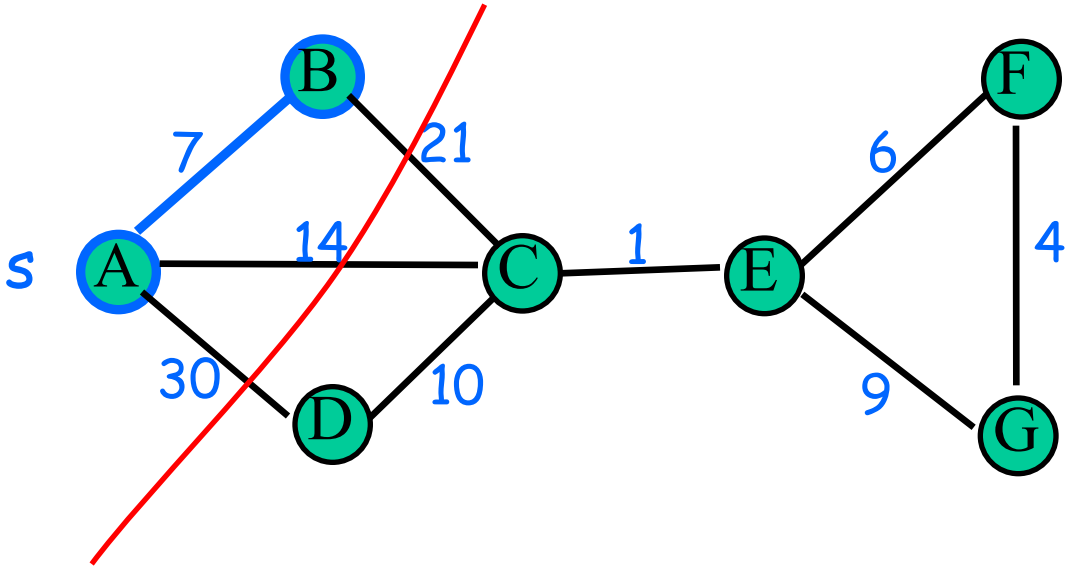
Esempio



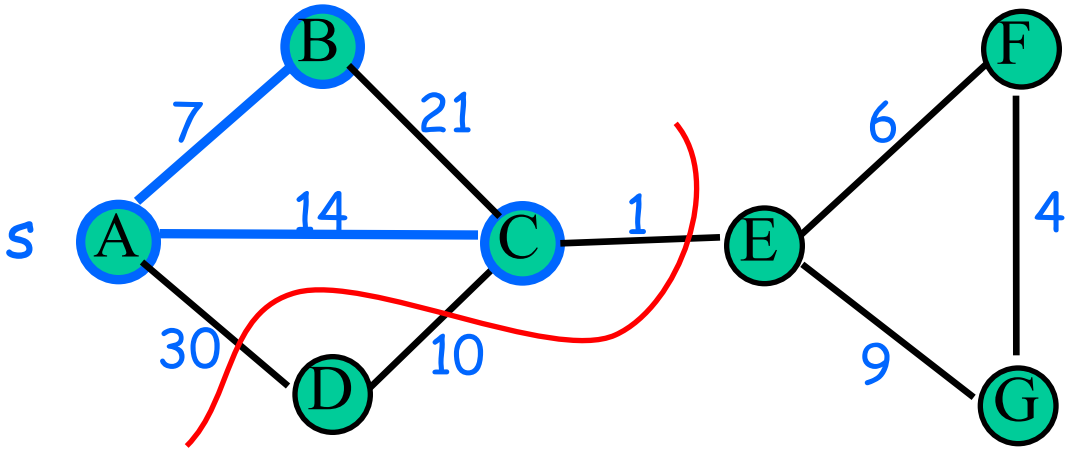
Esempio



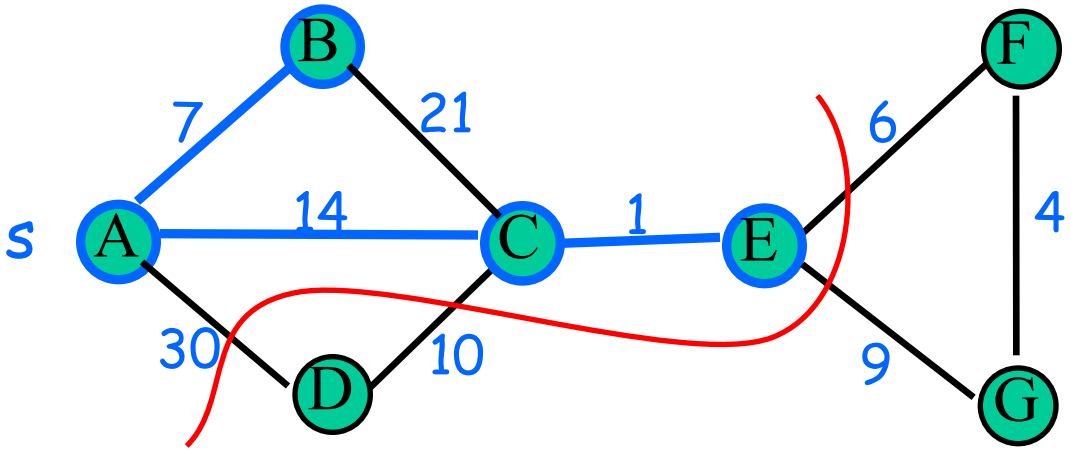
Esempio



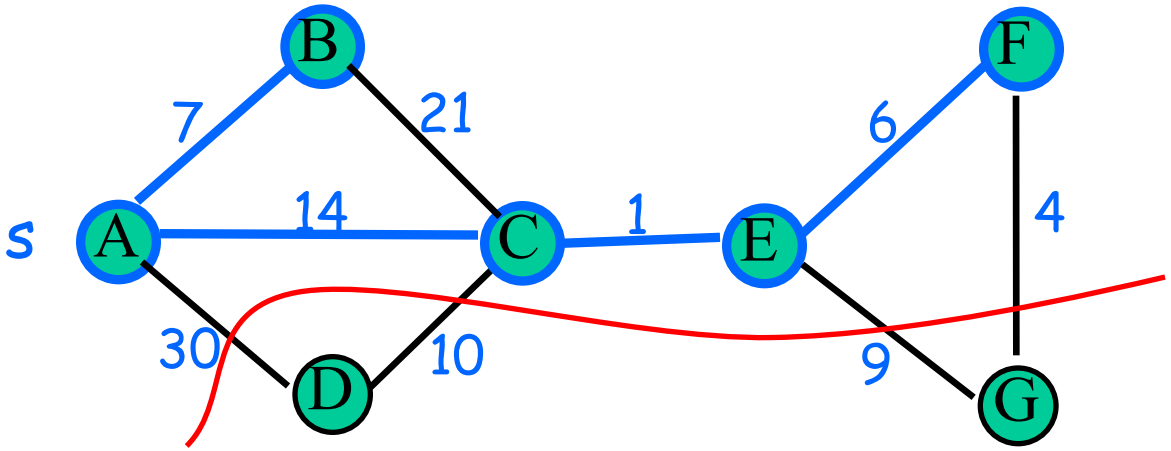
Esempio



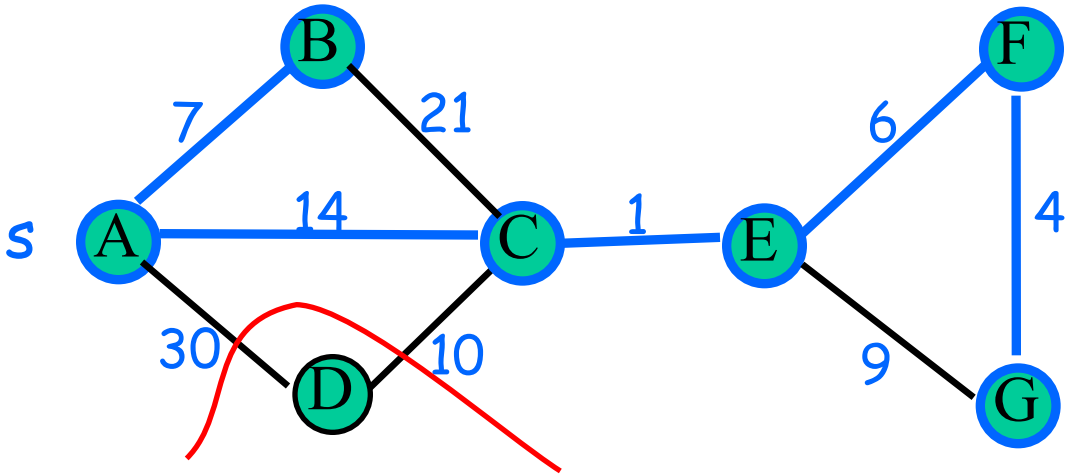
Esempio



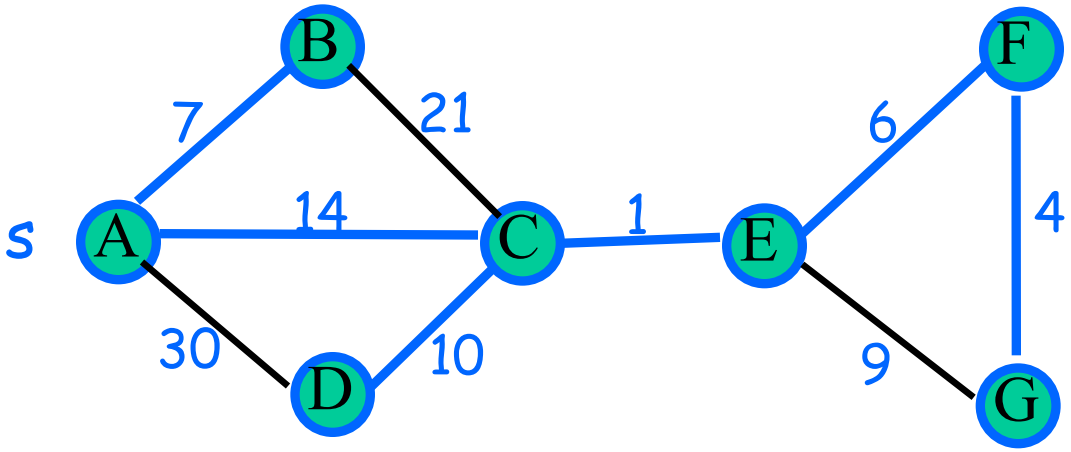
Esempio



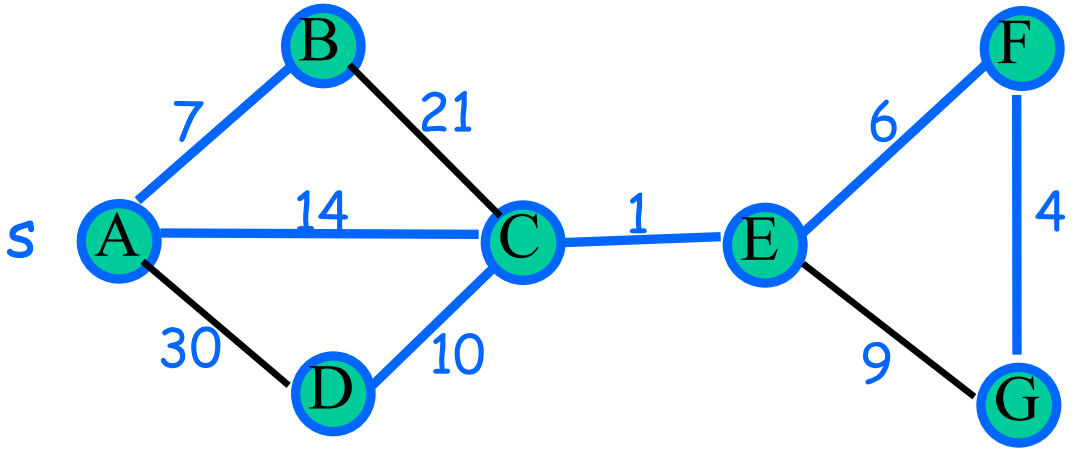
Esempio



Esempio



Complessità?



un approccio banale:
In ognuno degli n-1 passi,
guardo tutti gli O(m) archi che attraversano il taglio (V(T),V\V(T)) corrente, e scelgo quello di peso minimo

➡ costo O(m·n)

possiamo fare meglio?

Un approccio più efficiente

- Per $v \notin T$, definiamo arco azzurro associato a v un arco (u,v) tale che $u \in T$, ed (u,v) ha peso minimo tra tutti gli archi che connettono v ad un vertice in T
- L'algoritmo mantiene in una coda di priorità i nodi non ancora aggiunti alla soluzione, aventi ciascuno per chiave il peso del rispettivo arco azzurro associato ($+\infty$ nel caso in cui esso non esista); l'insieme delle chiavi viene memorizzato anche in un vettore ausiliario $d[1..n]$;
- Ad ogni passo, viene estratto il minimo dalla coda, aggiungendo il nodo associato alla soluzione, e si procede quindi all'eventuale aggiornamento delle chiavi nella coda di priorità

Pseudocodice

```
    algoritmo Prim (grafo G) → albero
1.      for each ( vertice v in G ) do d(v) ← +∞
2.      T ← albero formato da un solo nodo s
3.      CodaPriorita S
4.      d(s) ← 0
5.      S.insert(s, 0)
6.      while ( not S.isEmpty() ) do
7.          u ← S.deleteMin()
8.          for each ( arco (u, v) in G ) do
9.              if (d(v) = +∞) then
10.                 S.insert(v, w(u, v))
11.                 d(v) ← w(u, v)
12.                 rendi u padre di v in T
13.             else if (w(u, v) < d(v) and v∈S ) then
14.                 S.decreaseKey(v, d(v)−w(u, v))
15.                 d(v) ← w(u, v)
16.                 rendi u nuovo padre di v in T
17.     return T
```

Tempo di esecuzione utilizzando heap

Supponendo che il grafo G sia connesso e rappresentato tramite liste di adiacenza, avremo n `insert`, n `deleteMin` e al più m `decreaseKey`

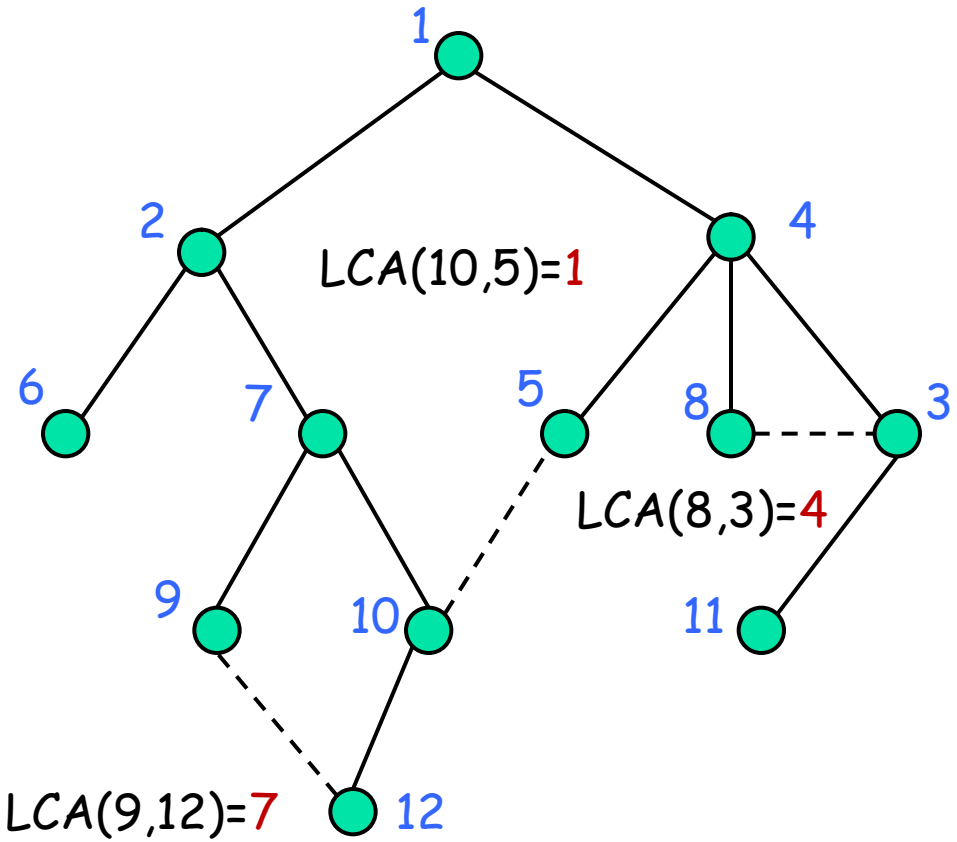
- $n \cdot O(\log n) + n \cdot O(\log n) + O(m) \cdot O(\log n) = O(m \log n)$ utilizzando heap binari o binomiali (come Kruskal)
- $n \cdot O(1) + n \cdot O(\log n) + O(m) \cdot O(1)^*$ $= O(m+n \log n)$ utilizzando heap di Fibonacci (meglio di Kruskal, che costava $O(m \log n)$, se $m=\omega(n)$, mentre i due approcci si equivalgono se $m=\Theta(n)$).

Un algoritmo per calcolare i minimi antenati comuni in un albero

un'altra applicazione interessante
della struttura dati Union-Find

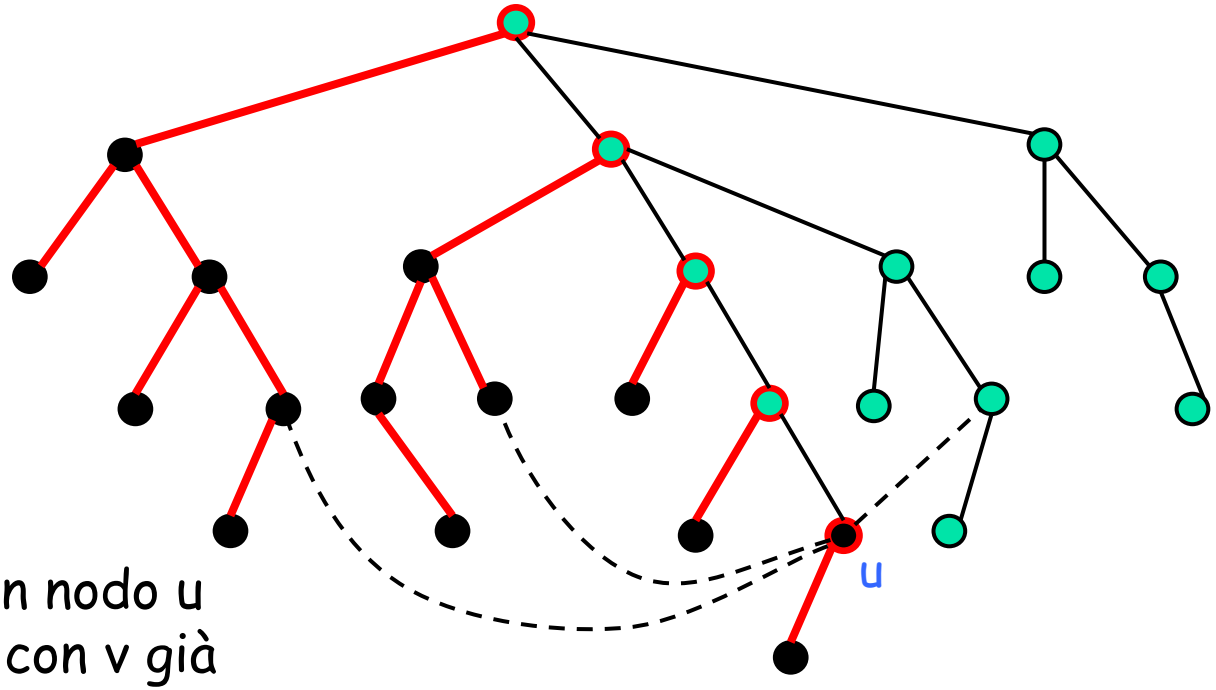
Il problema del calcolo dei minimi antenati comuni in un albero

- Input:
  - un albero radicato $T=(V,E)$
  - un insieme S di archi non dell'albero
- Output:
  - per ogni $(u,v) \in S$, vogliamo il minimo antenato comune $LCA(u,v)$ (da *Least Common Ancestor*) di u e v, ovvero l'antenato di u e v che è più lontano dalla radice



l'idea

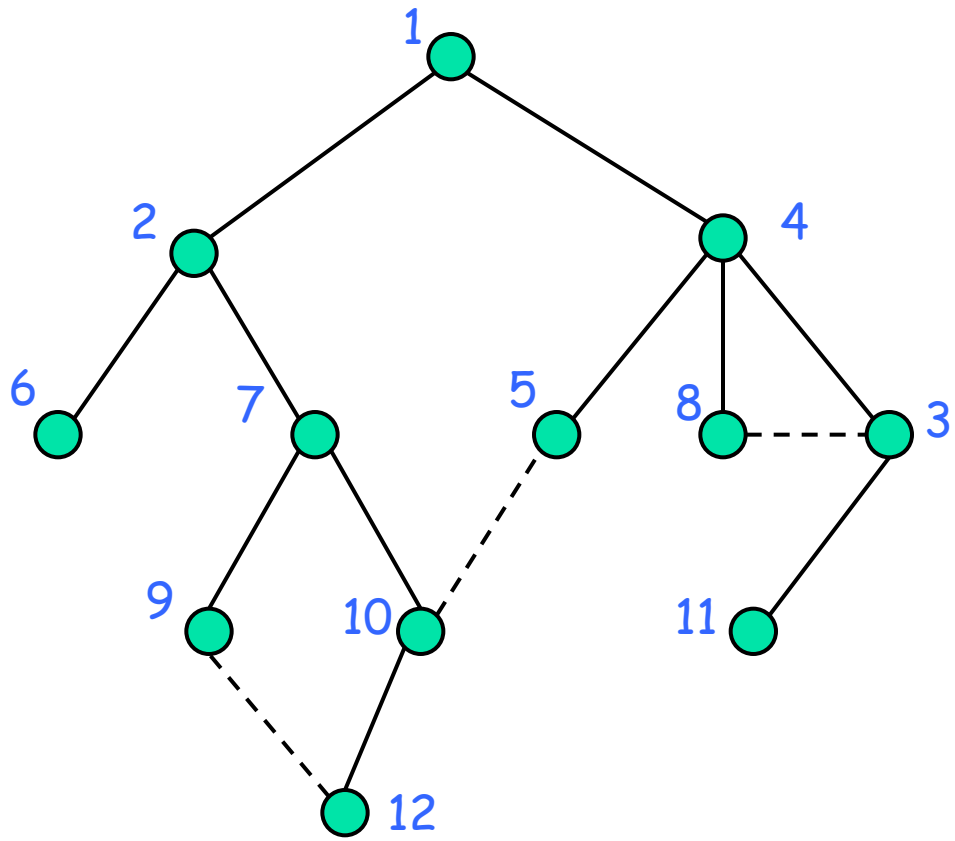
- eseguo una visita DFS
- mantengo con insiemi disgiunti le "parti" di albero già visitate (alberi rossi)
- il nome di una "parte" di albero è il nodo più vicino alla radice
- prima di abbandonare un nodo u posso trovare LCA(u,v) con v già visitato (abbandonato) facendo una find(v)



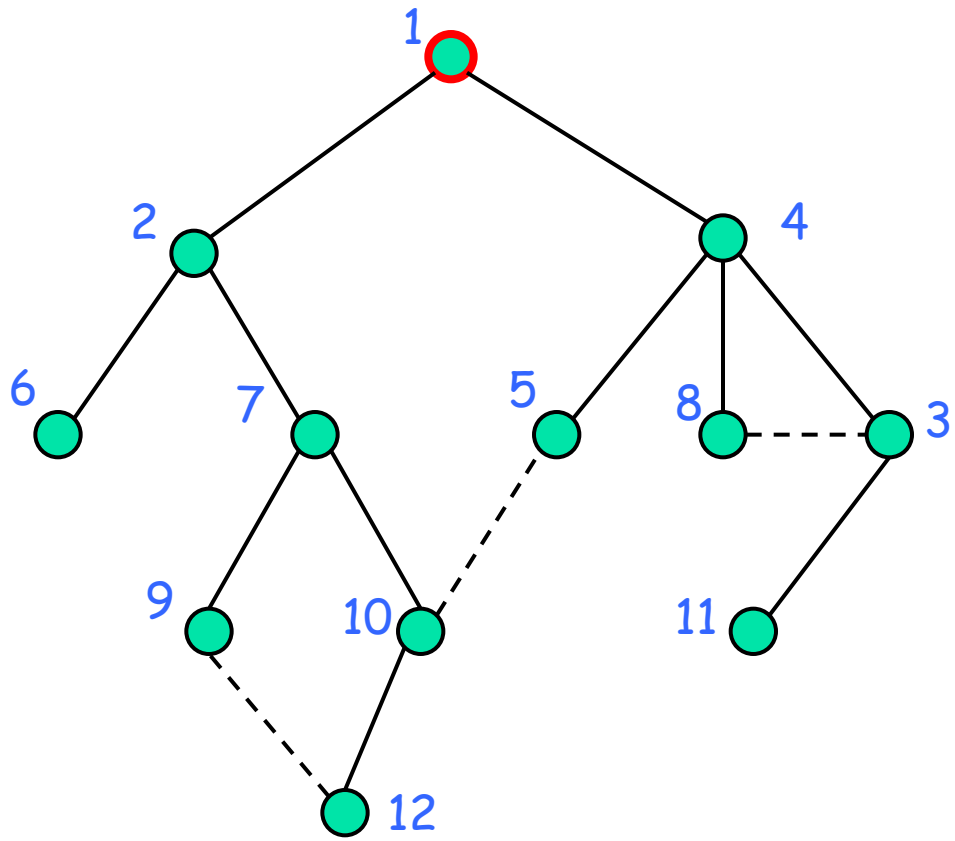
l'algoritmo (Tarjan '79)

```
procedure LCA(T, r)
    for ogni nodo v di T do
        MakeSet(v)
        Color[v] ← White

    Visita(T, r)

procedure Visita(T, u)
    for ogni figlio v di u in T do
        Visita(T, v)  Union(u, v)

    Color[u] ← Black

    for ogni coppia (u, v) ∈ S do
        if Color[v] = Black then
            Print Il minimo antenato comune tra u e v è Find(v)
```

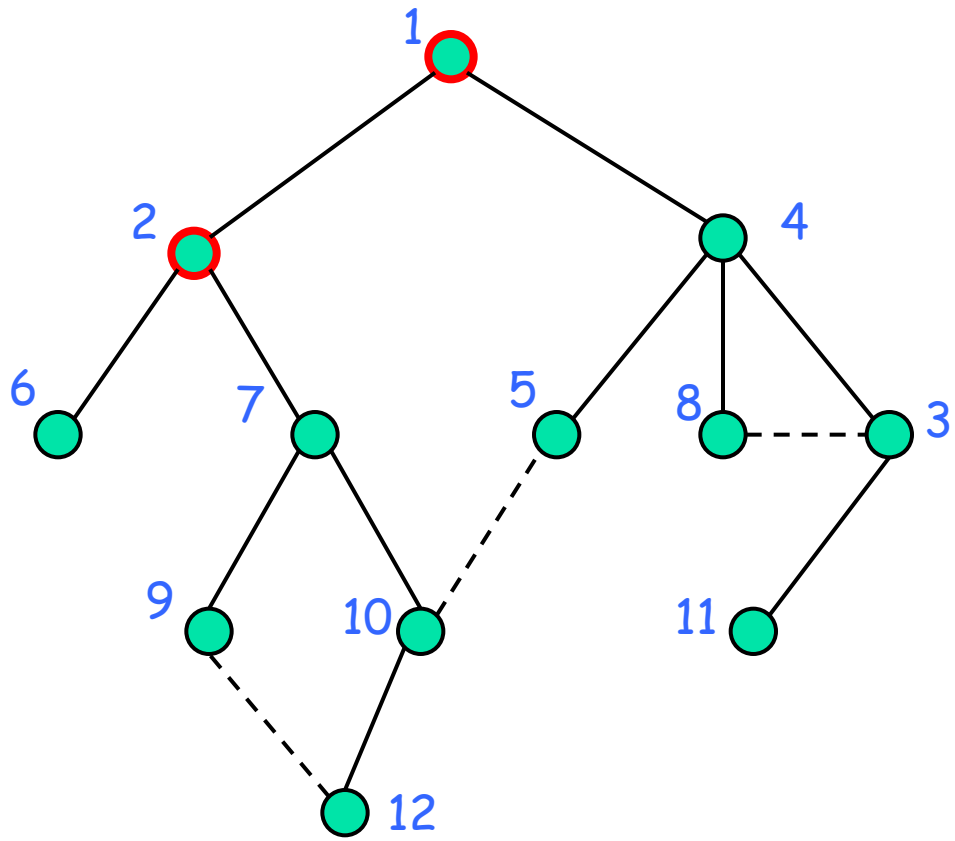
l'algoritmo



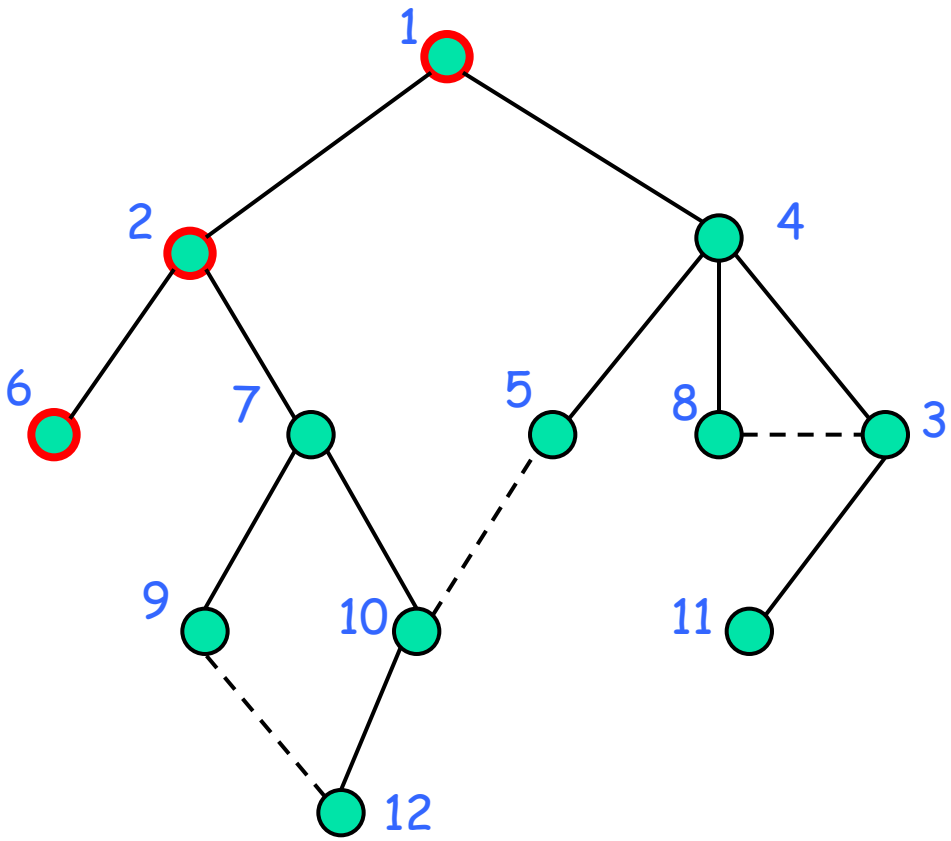
l'algoritmo



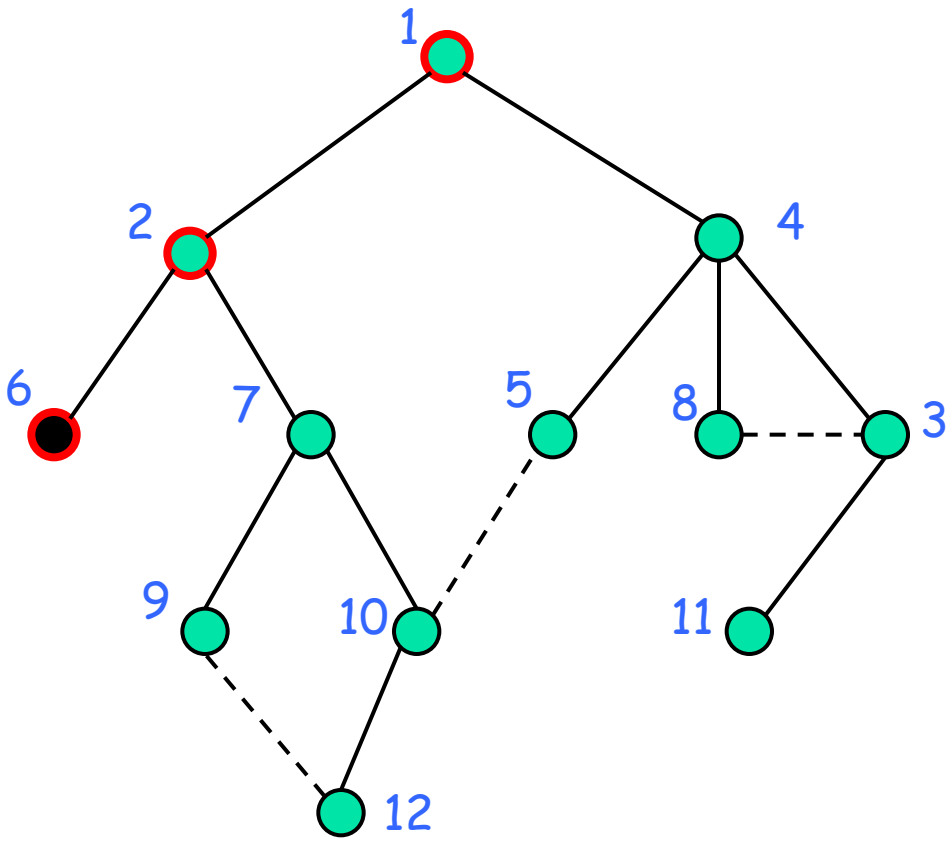
l'algoritmo



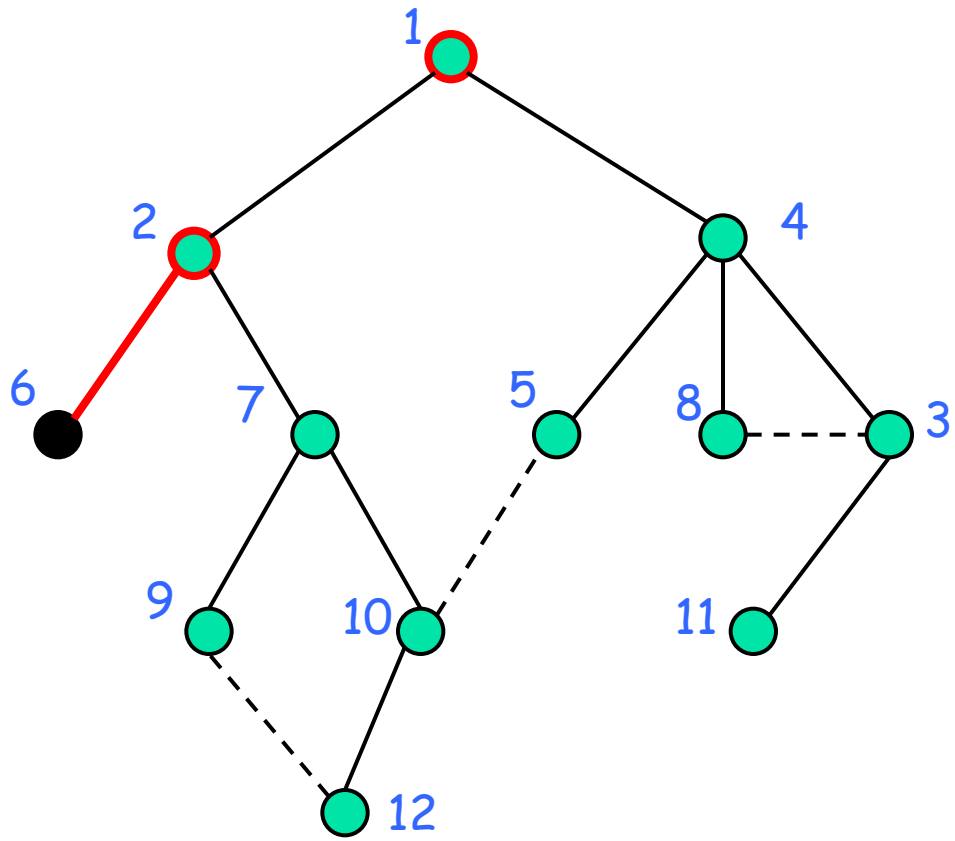
l'algoritmo



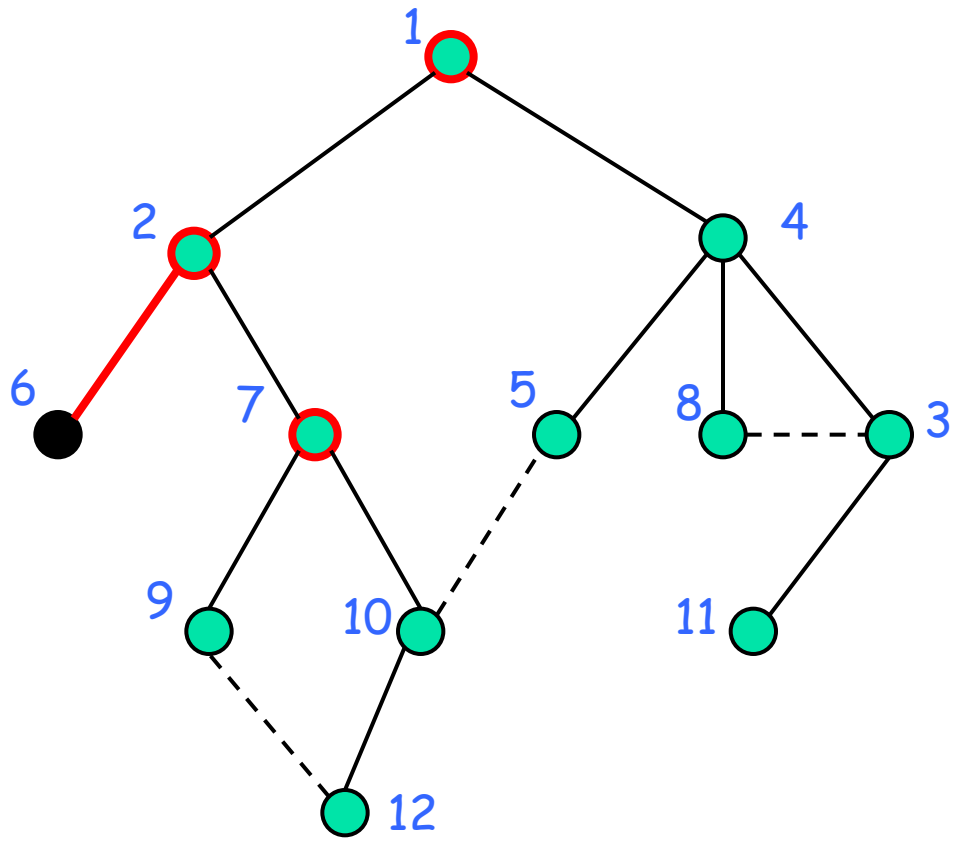
l'algoritmo



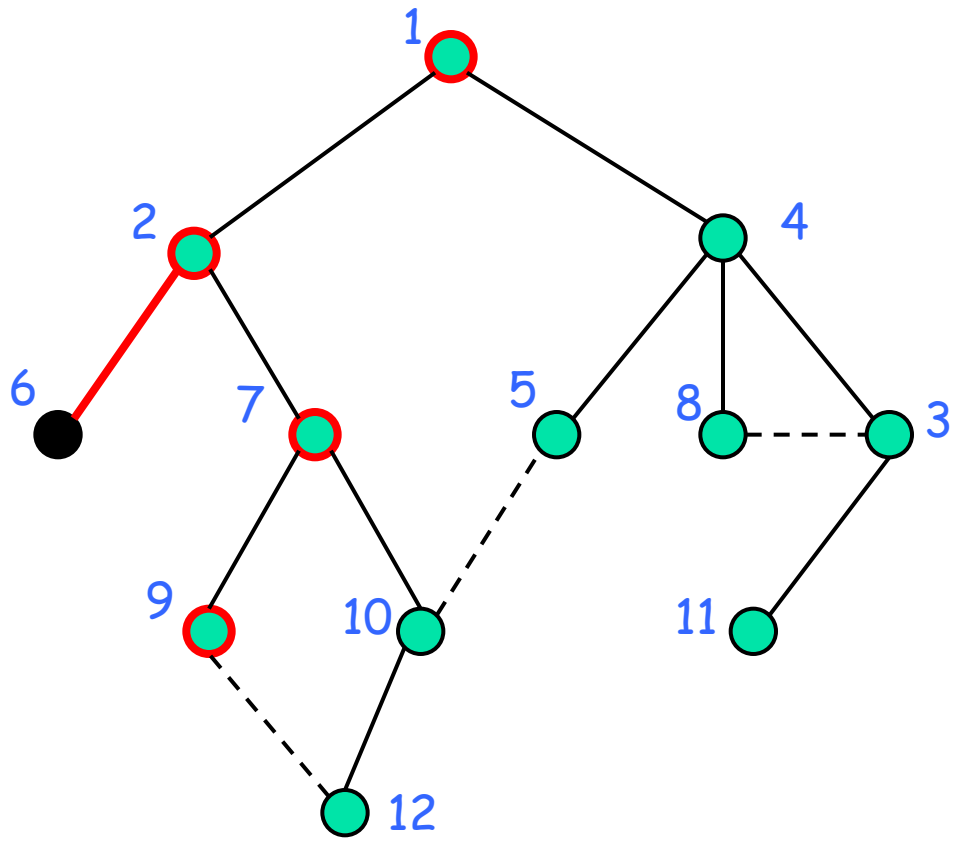
l'algoritmo



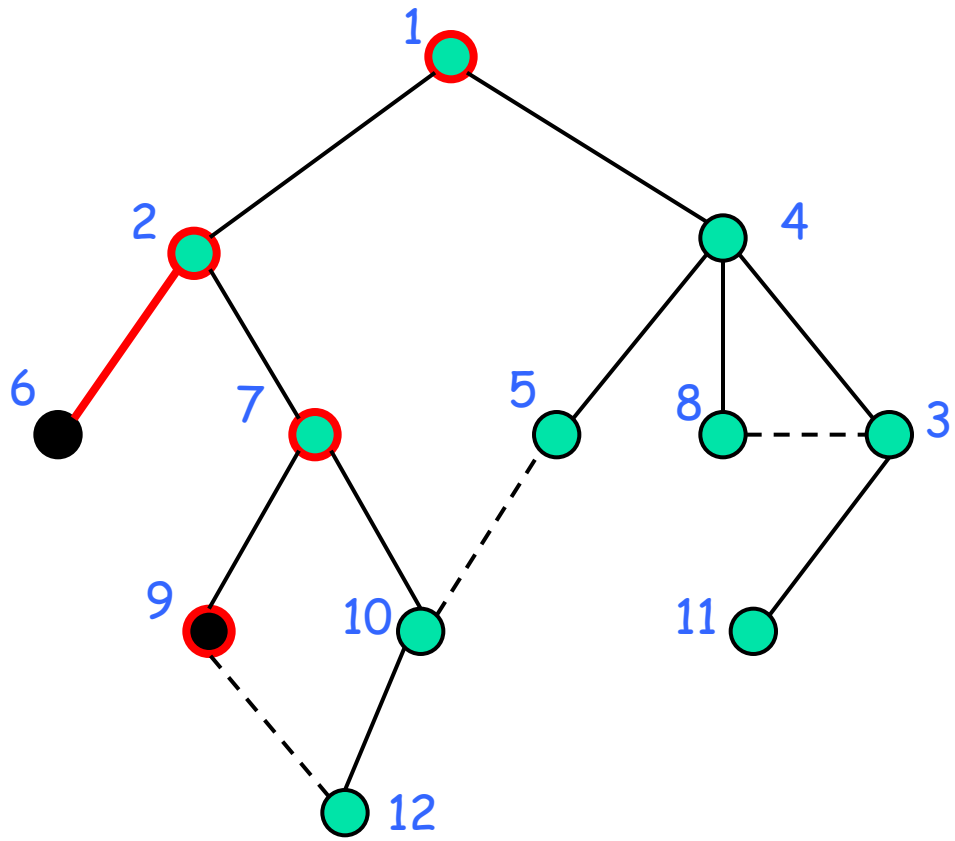
l'algoritmo



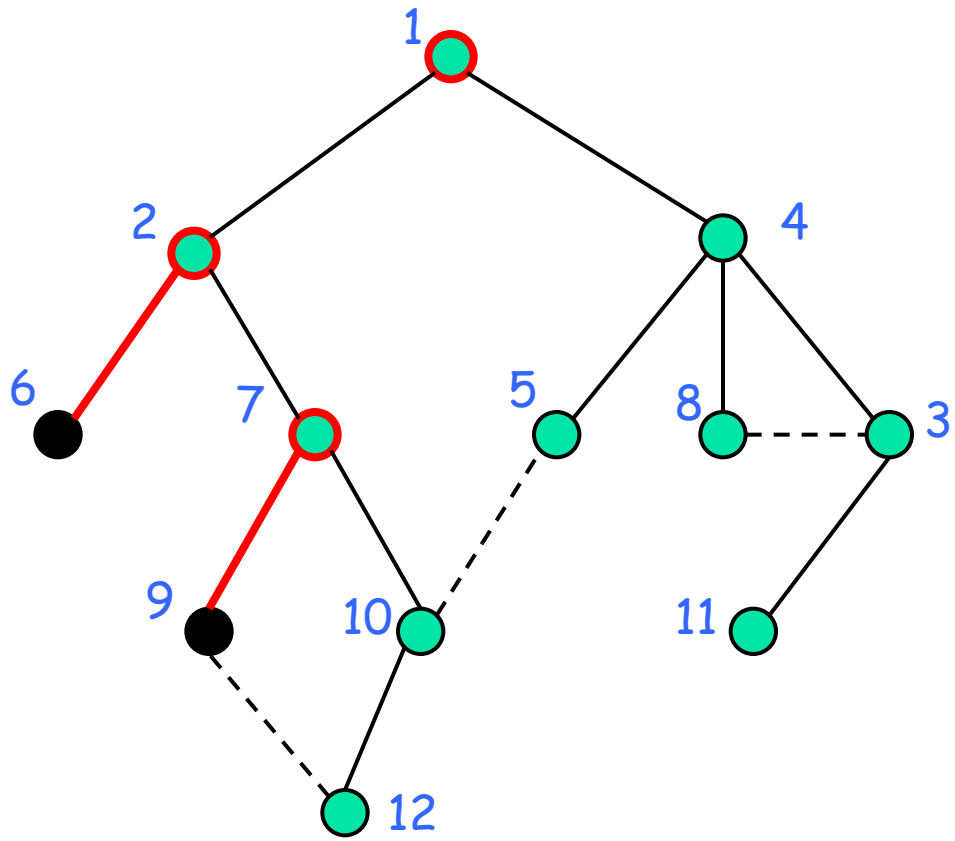
l'algoritmo



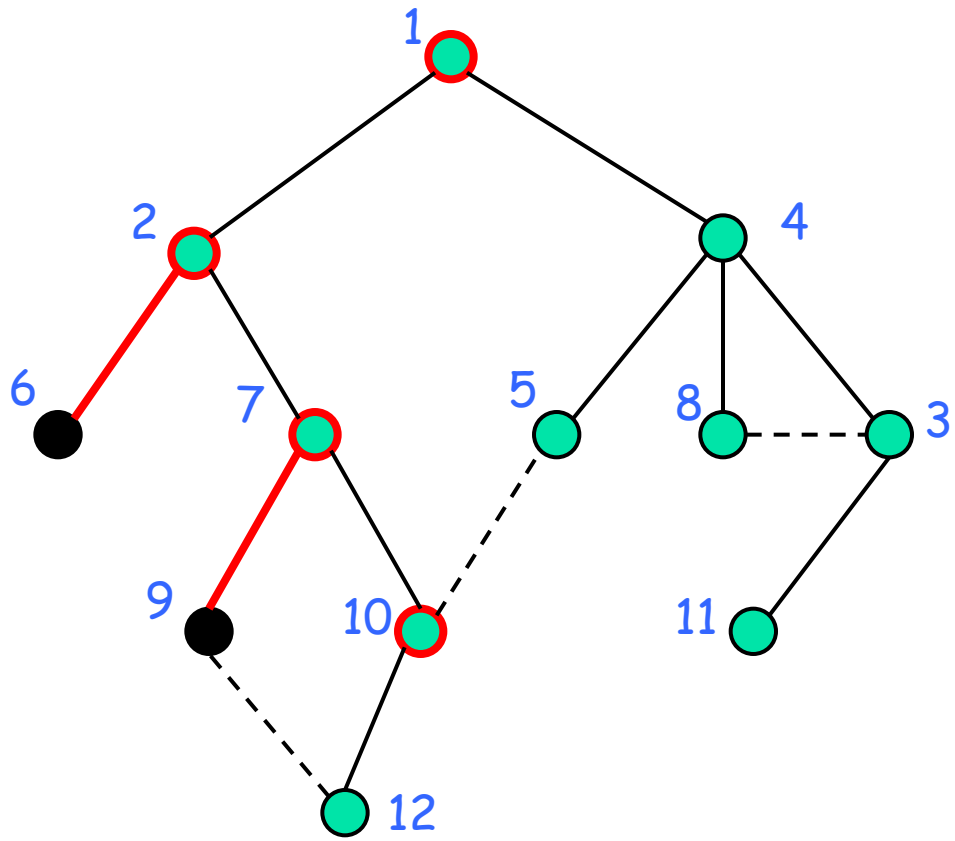
l'algoritmo



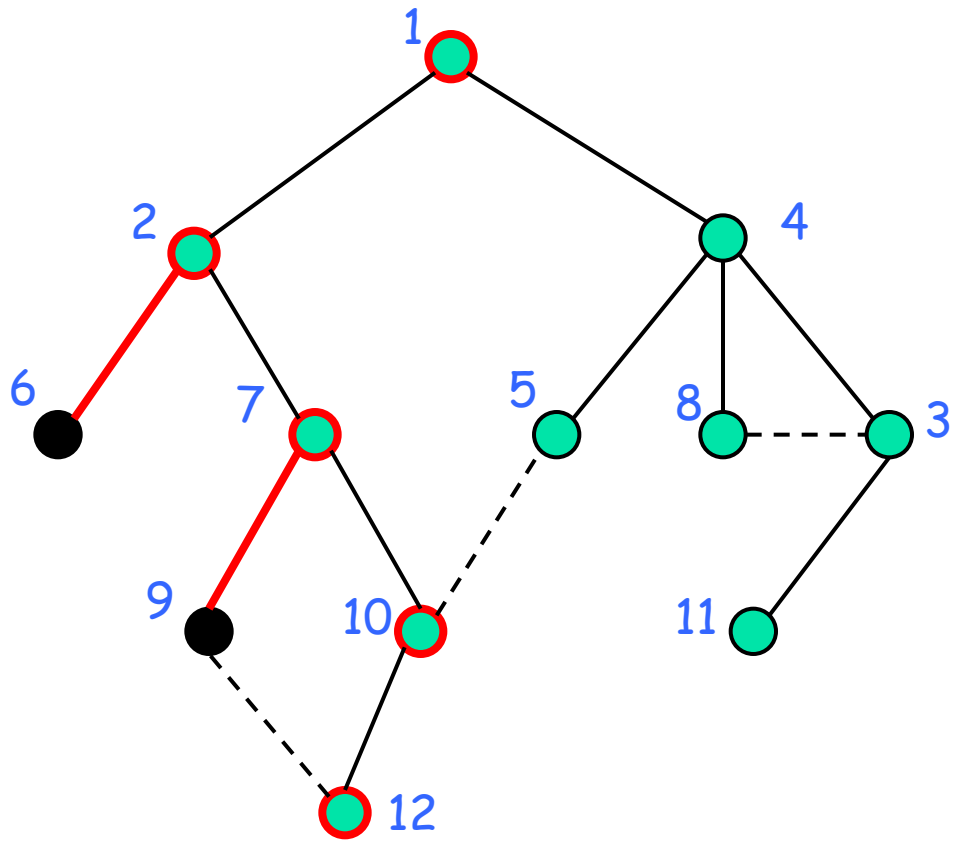
l'algoritmo



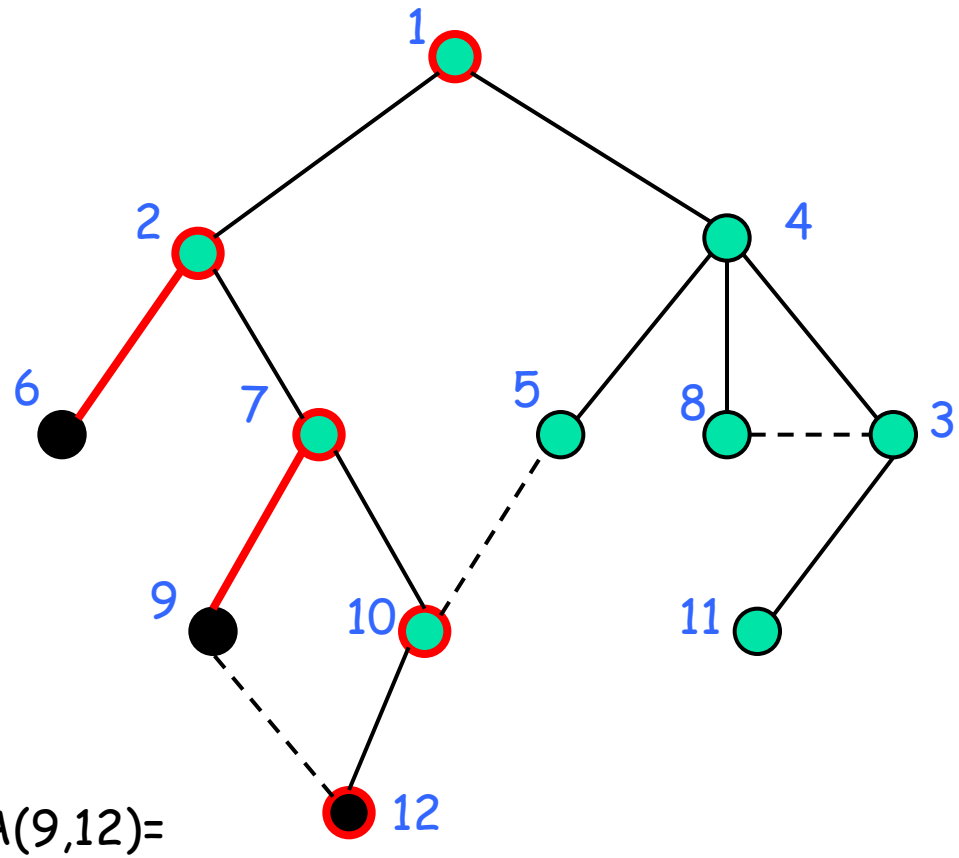
l'algoritmo



l'algoritmo

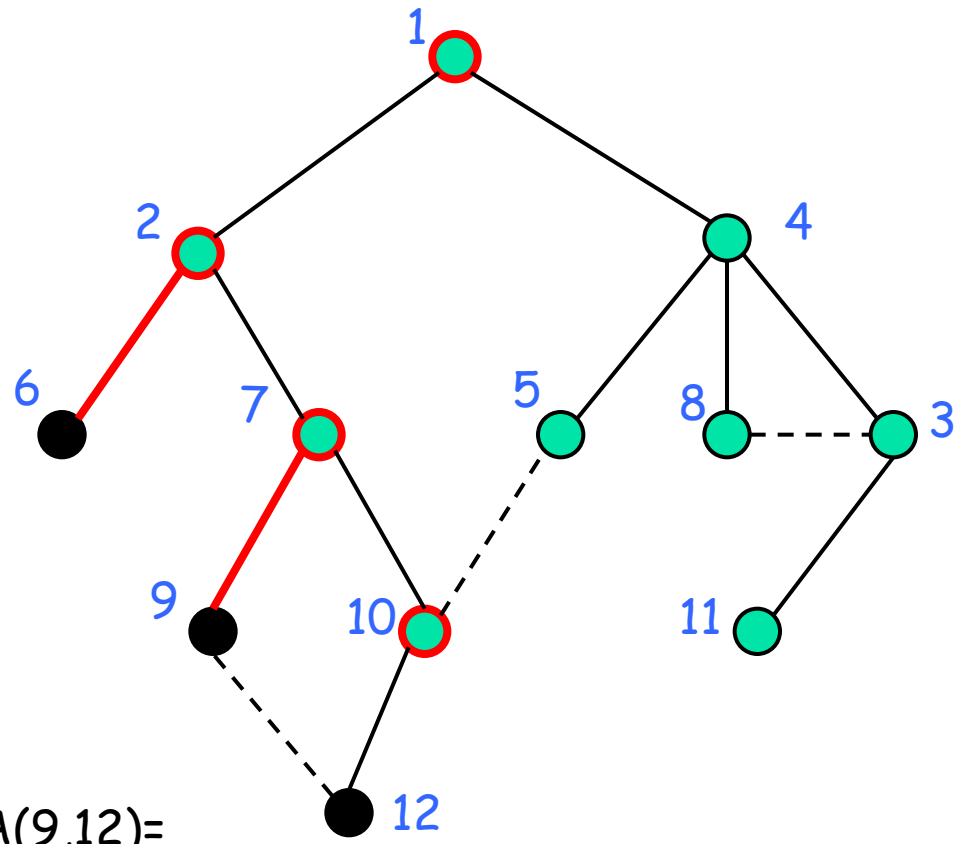


l'algoritmo



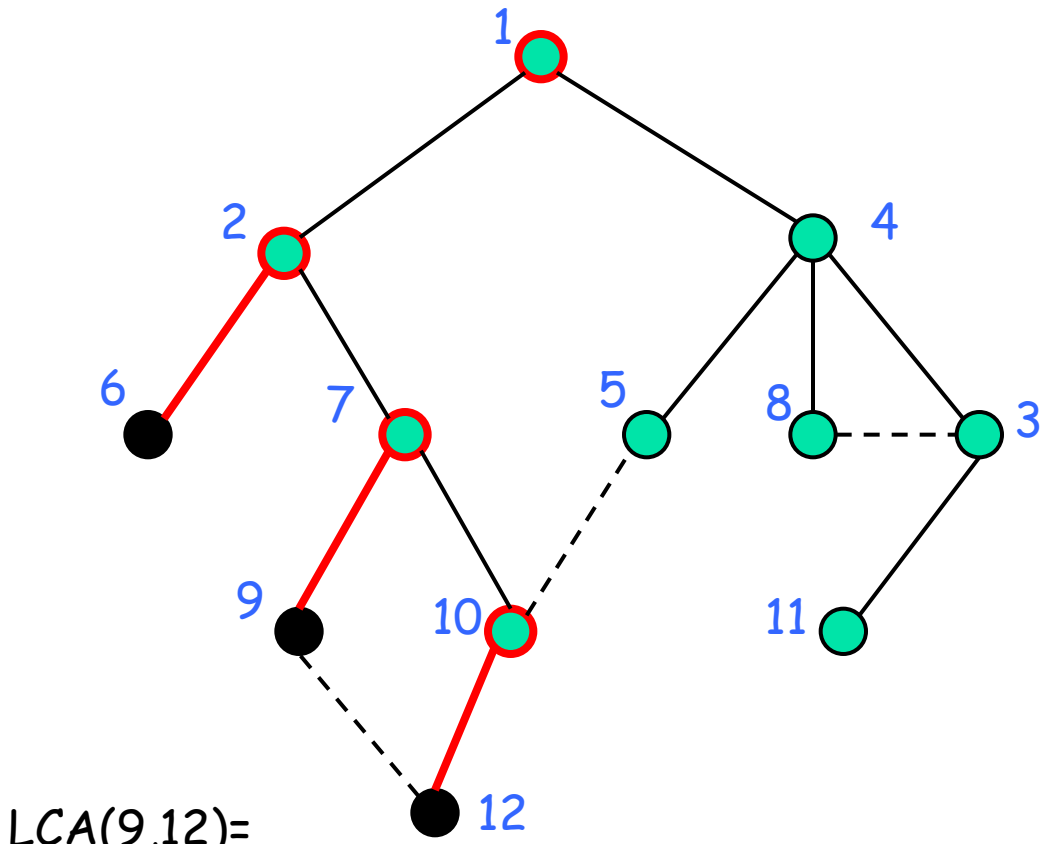
LCA(9,12)=
find(9)=7

l'algoritmo



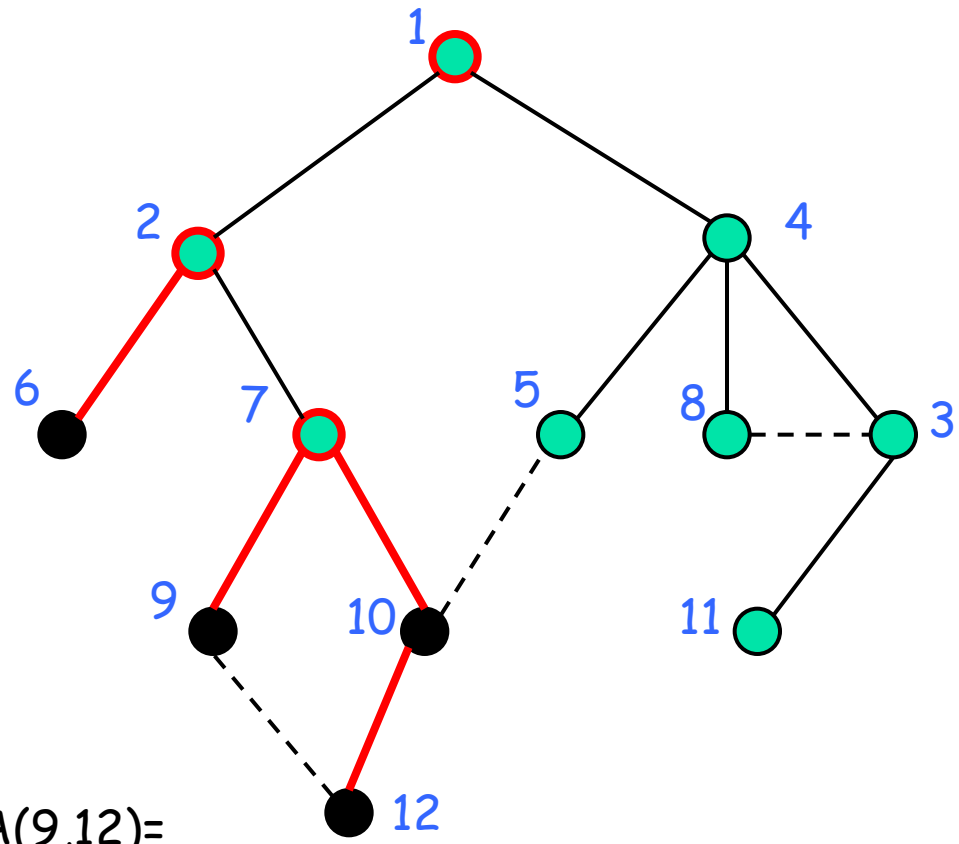
LCA(9,12)=
find(9)=7

l'algoritmo



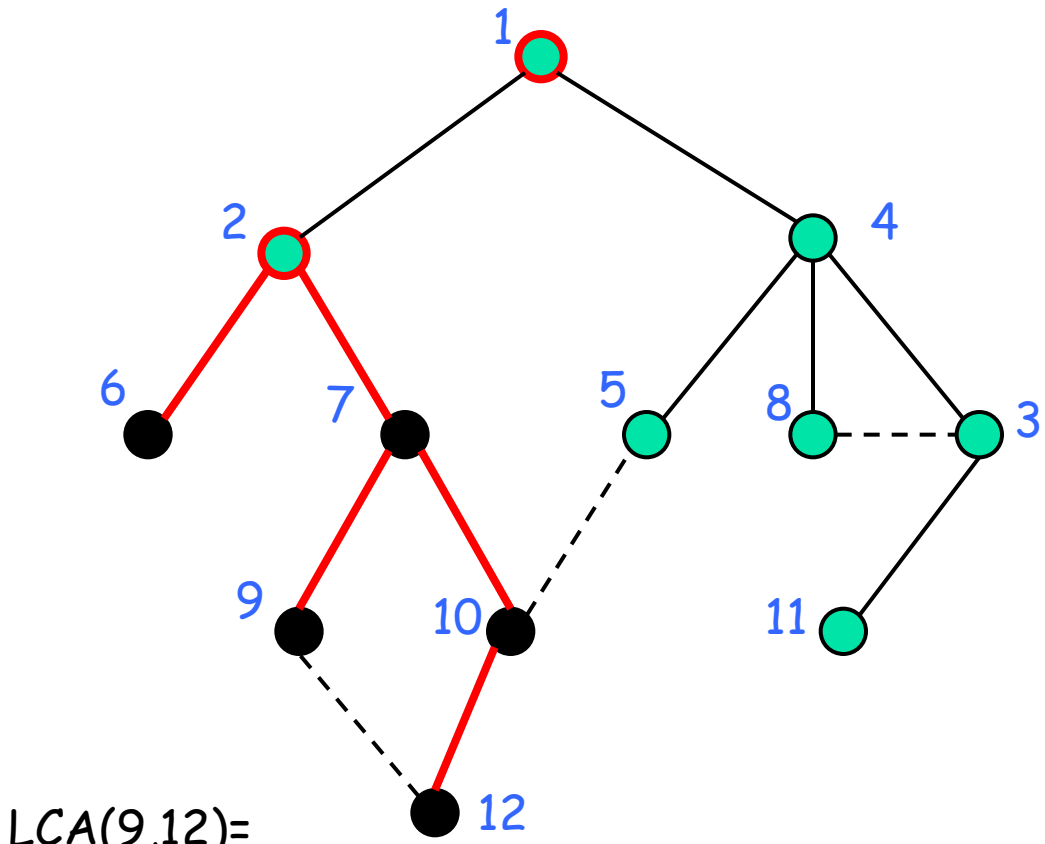
LCA(9,12)=
find(9)=7

l'algoritmo



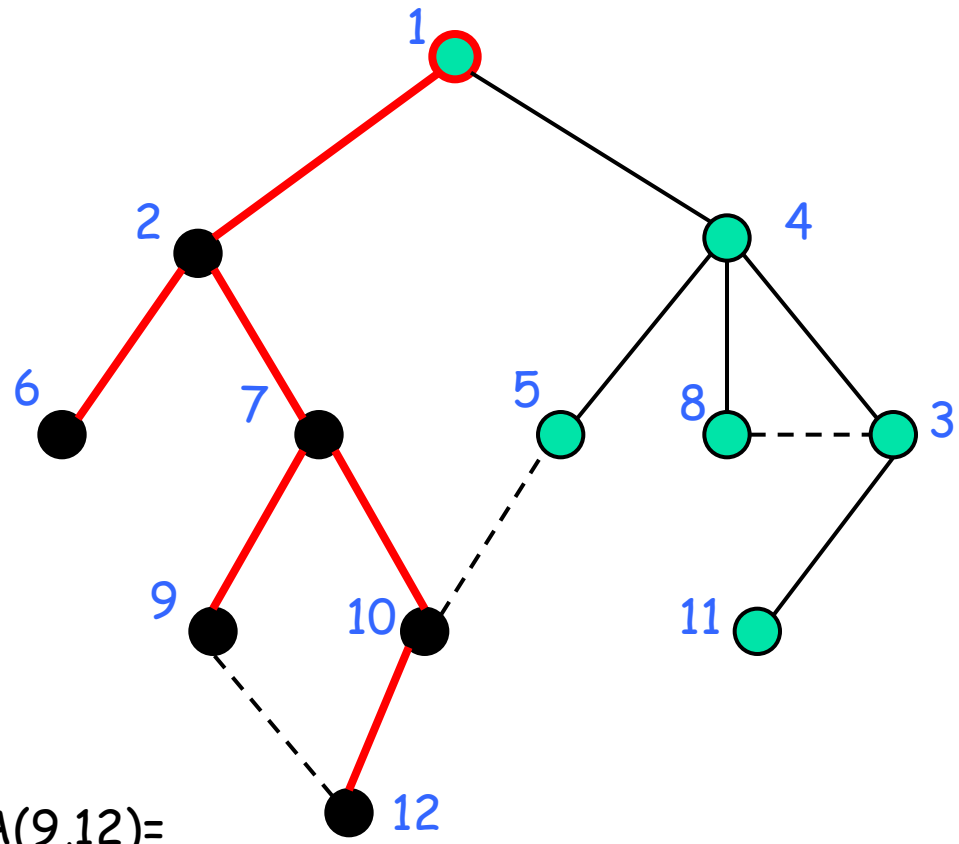
LCA(9,12)=
find(9)=7

l'algoritmo



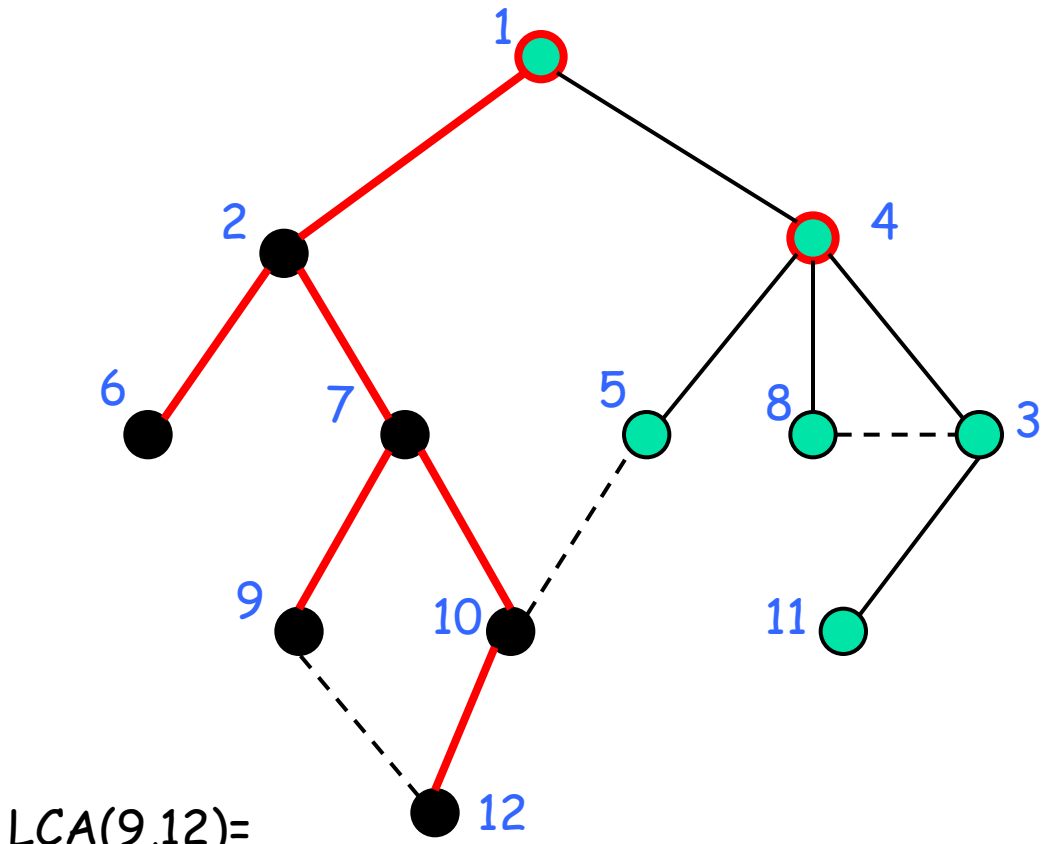
LCA(9,12)=
find(9)=7

l'algoritmo



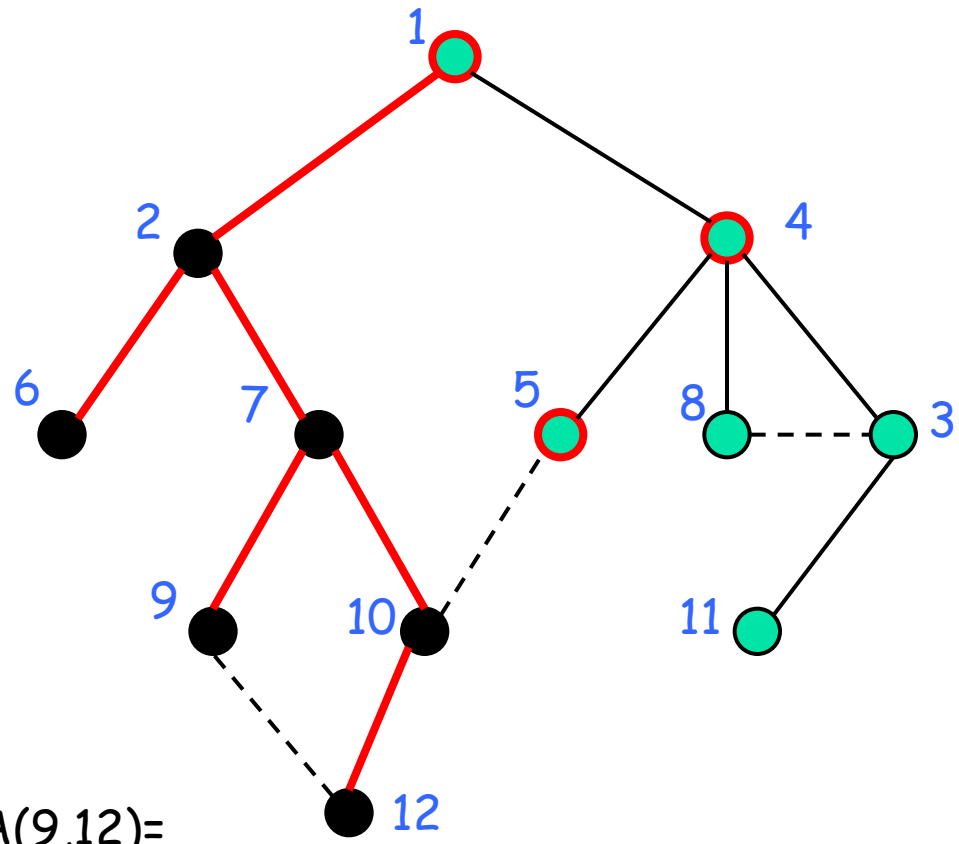
LCA(9,12)=
find(9)=7

l'algoritmo



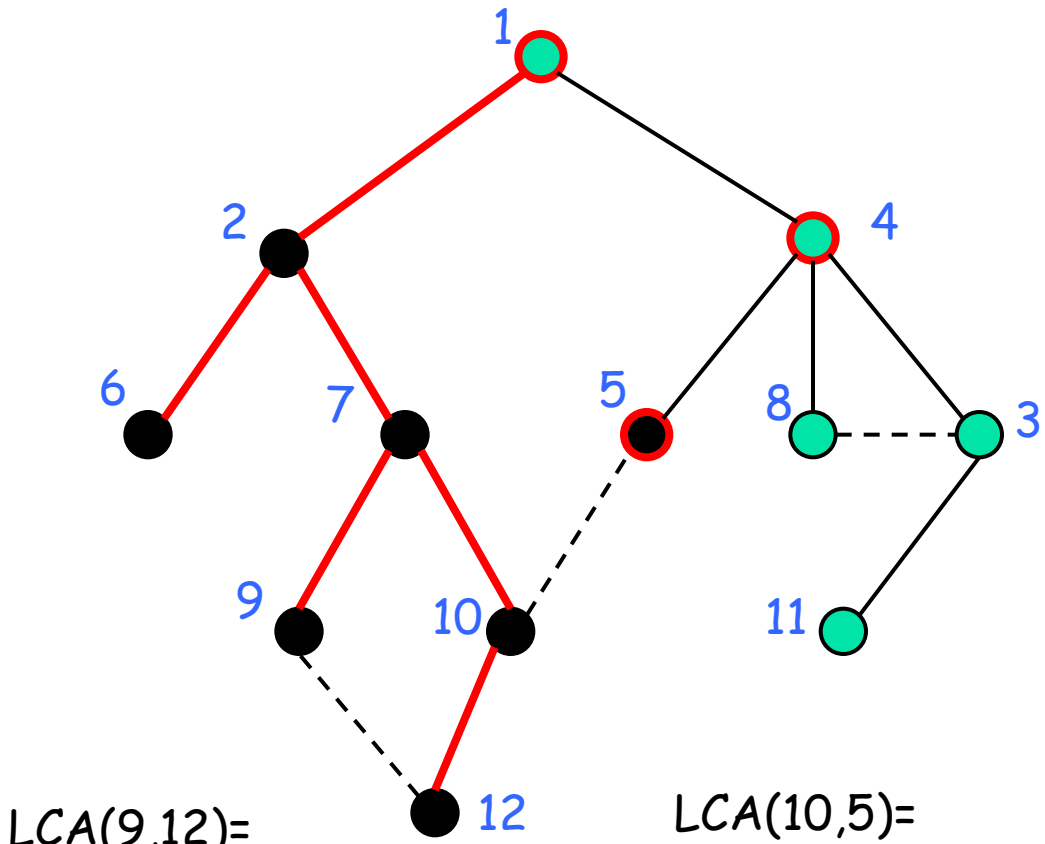
LCA(9,12)=
find(9)=7

l'algoritmo



LCA(9,12)=
find(9)=7

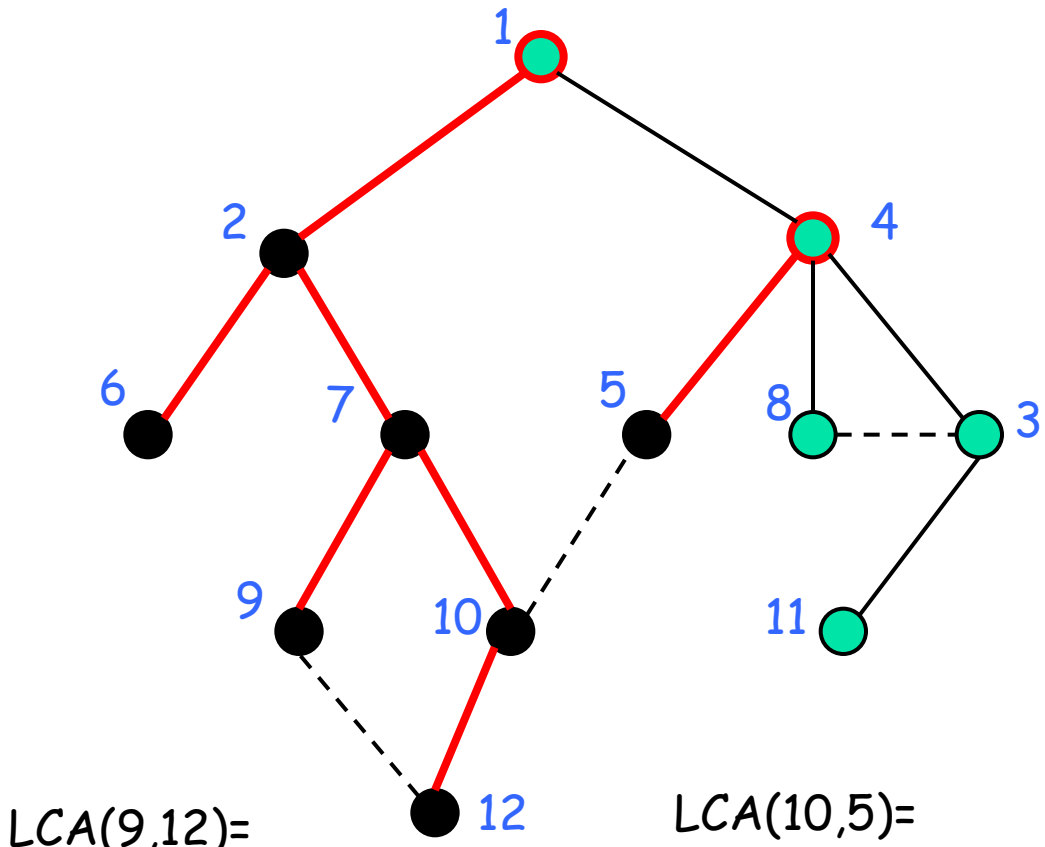
l'algoritmo



LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

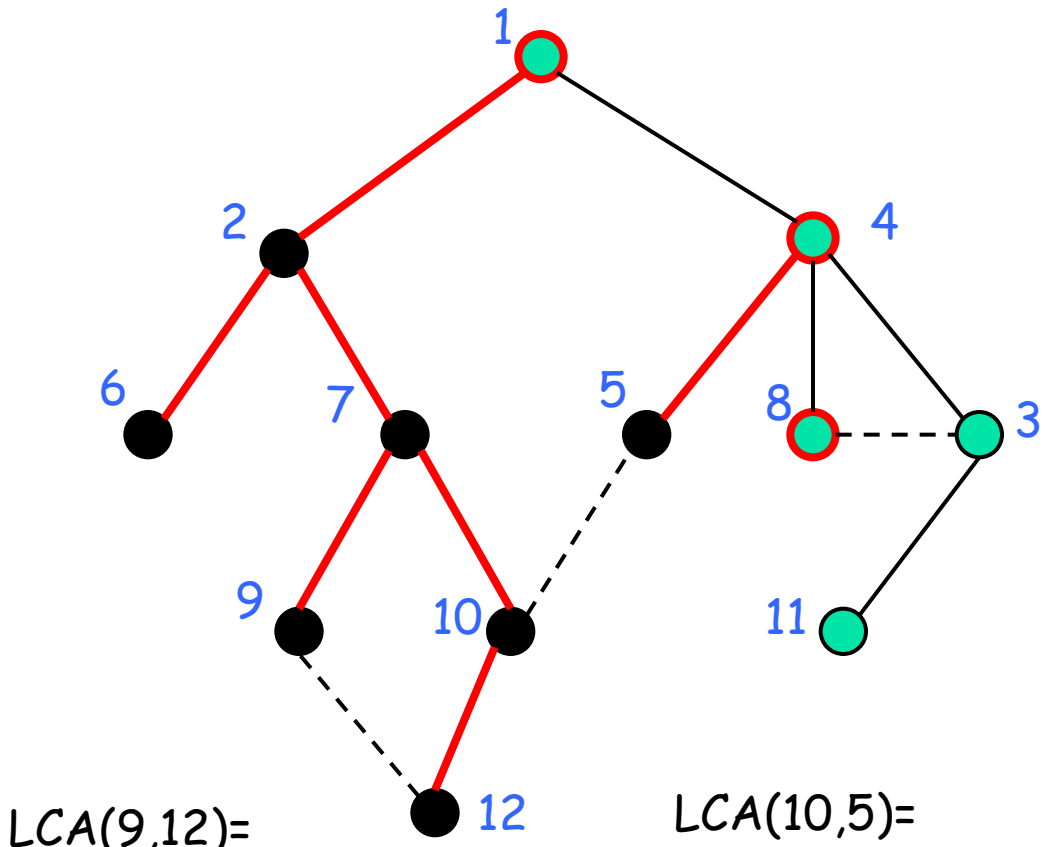
l'algoritmo



LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

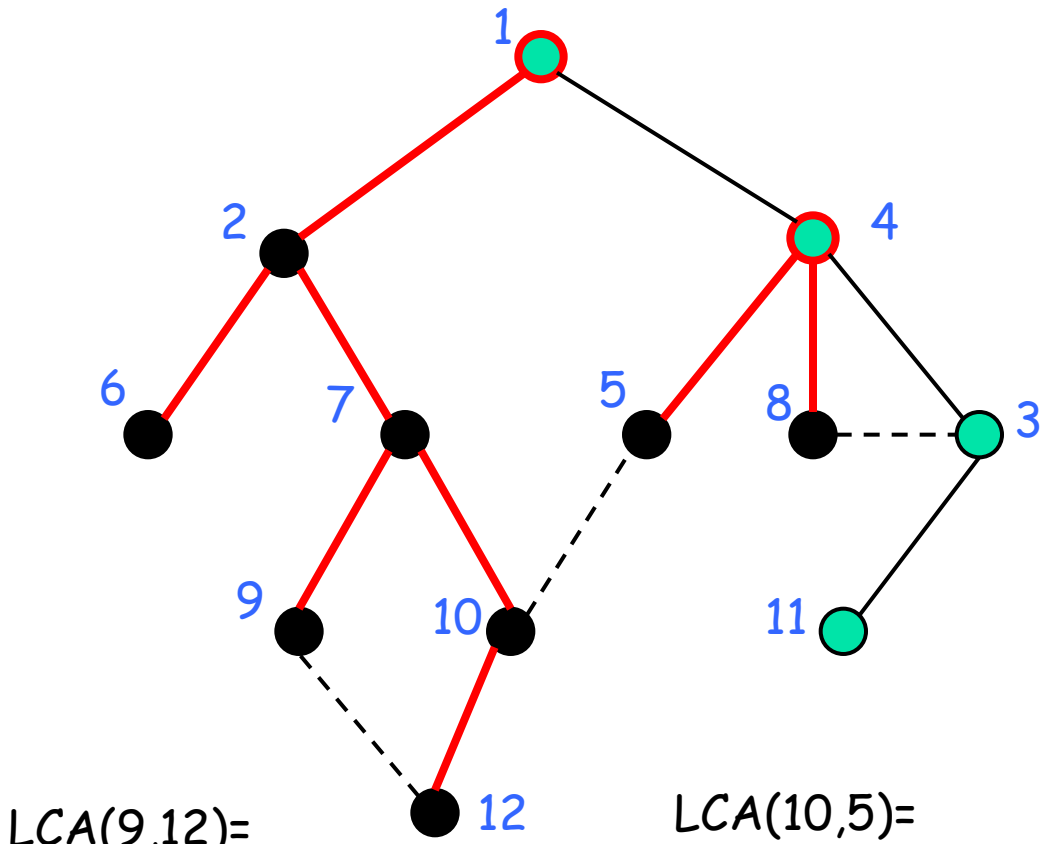
l'algoritmo



LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

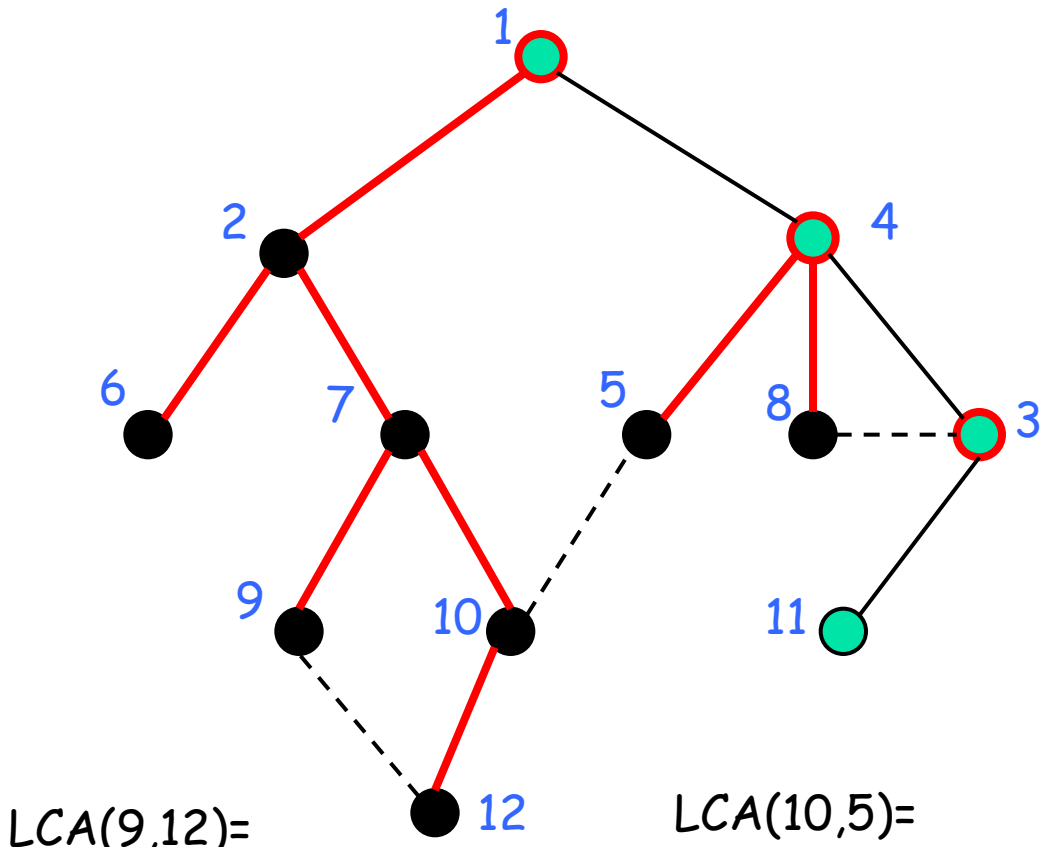
l'algoritmo



LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

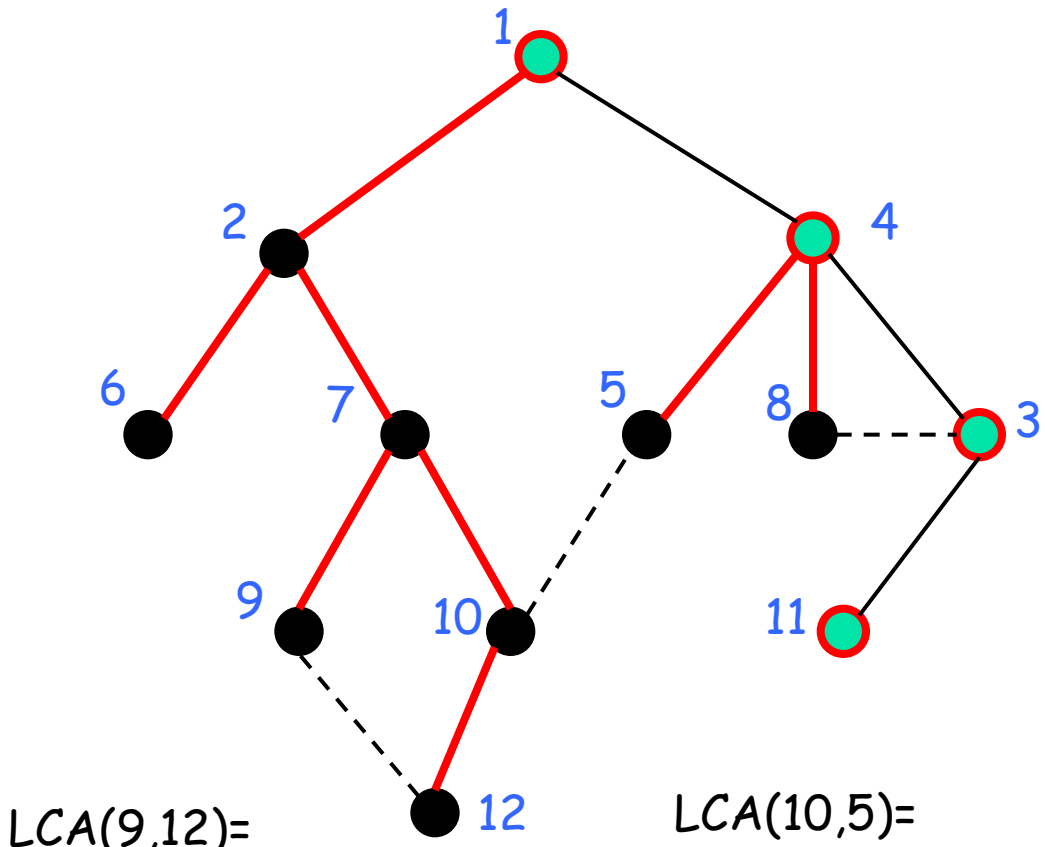
l'algoritmo



LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

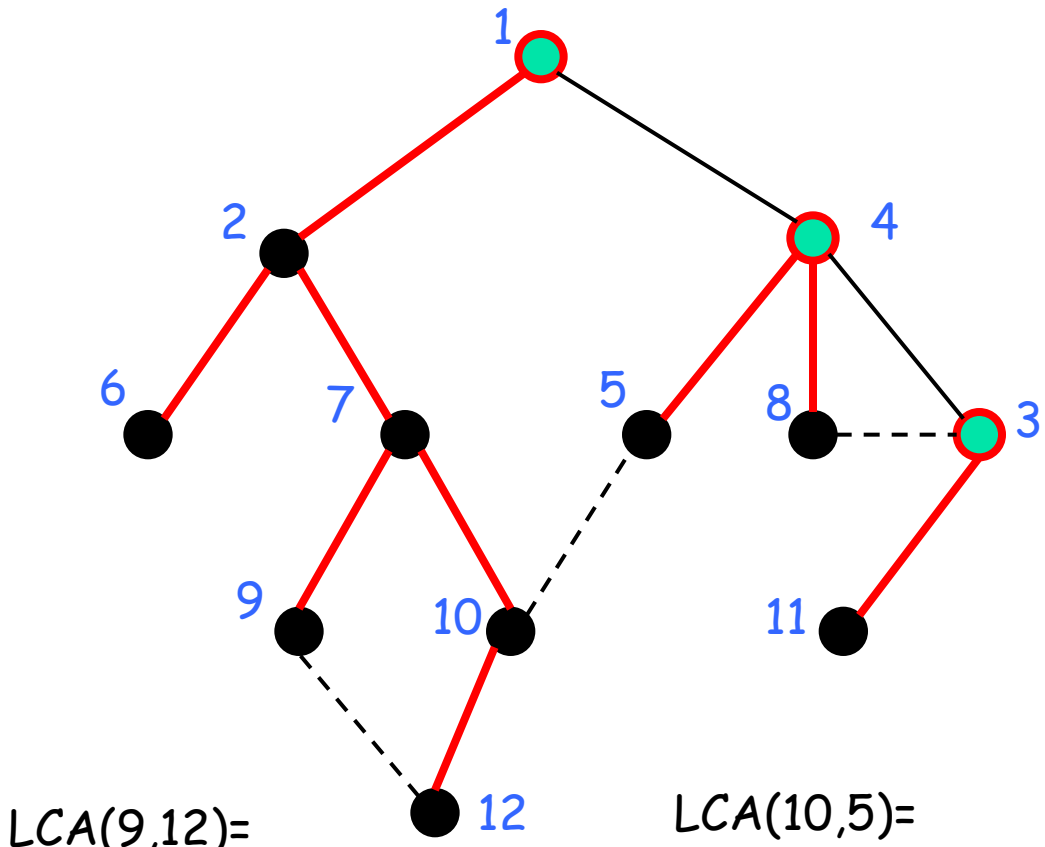
l'algoritmo



LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

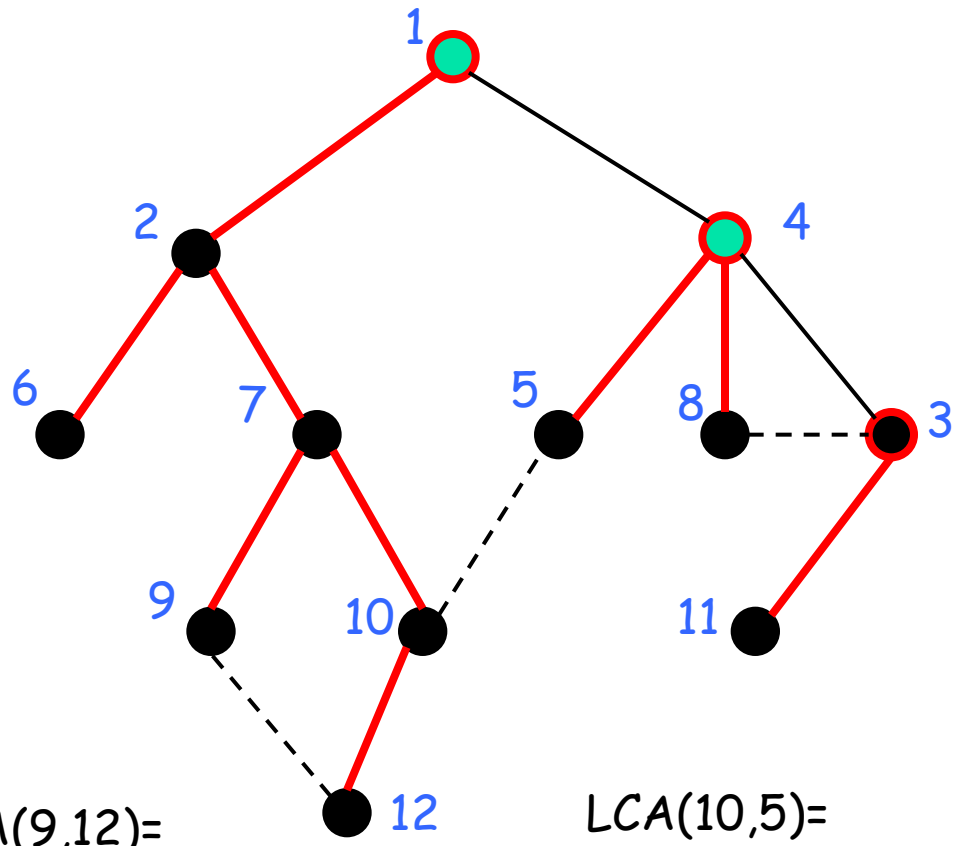
l'algoritmo



LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

l'algoritmo

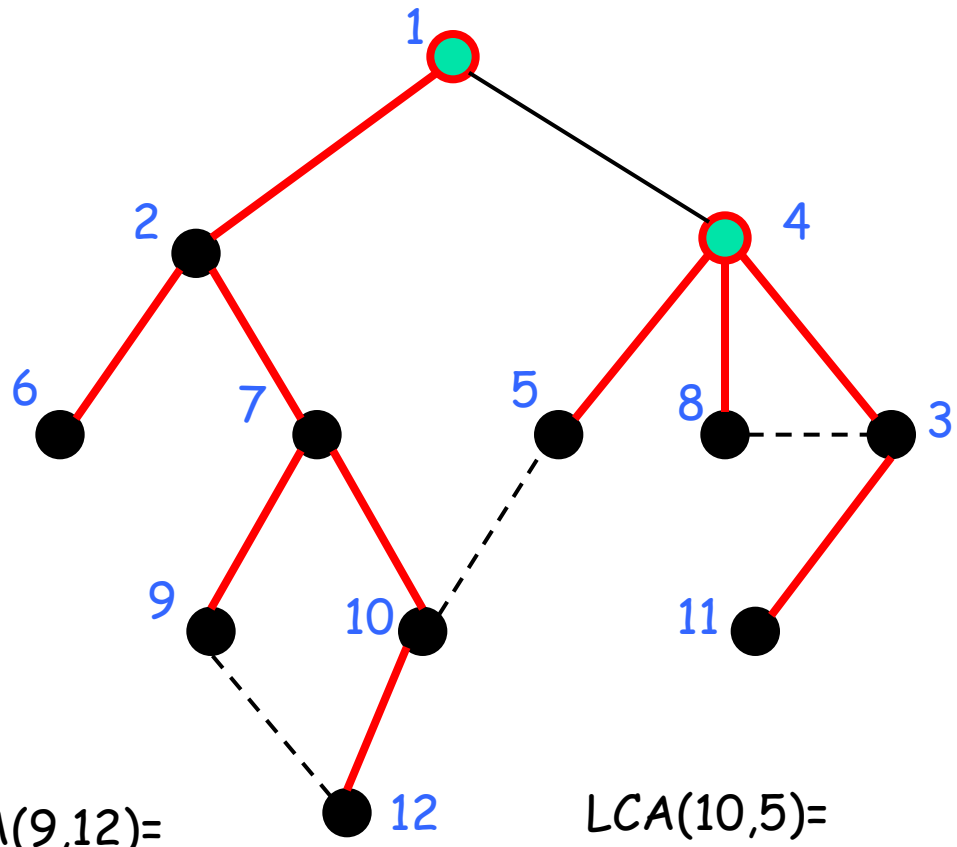


LCA(8,3)=
find(8)=4

LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

l'algoritmo

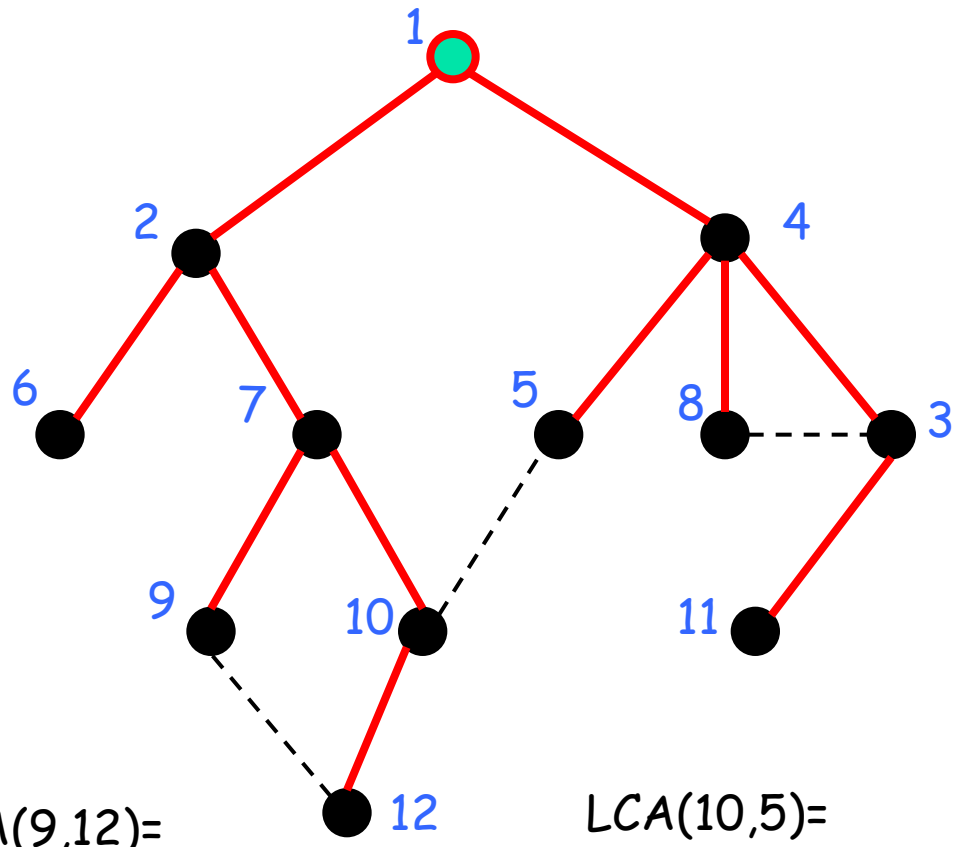


LCA(8,3)=
find(8)=4

LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

l'algoritmo

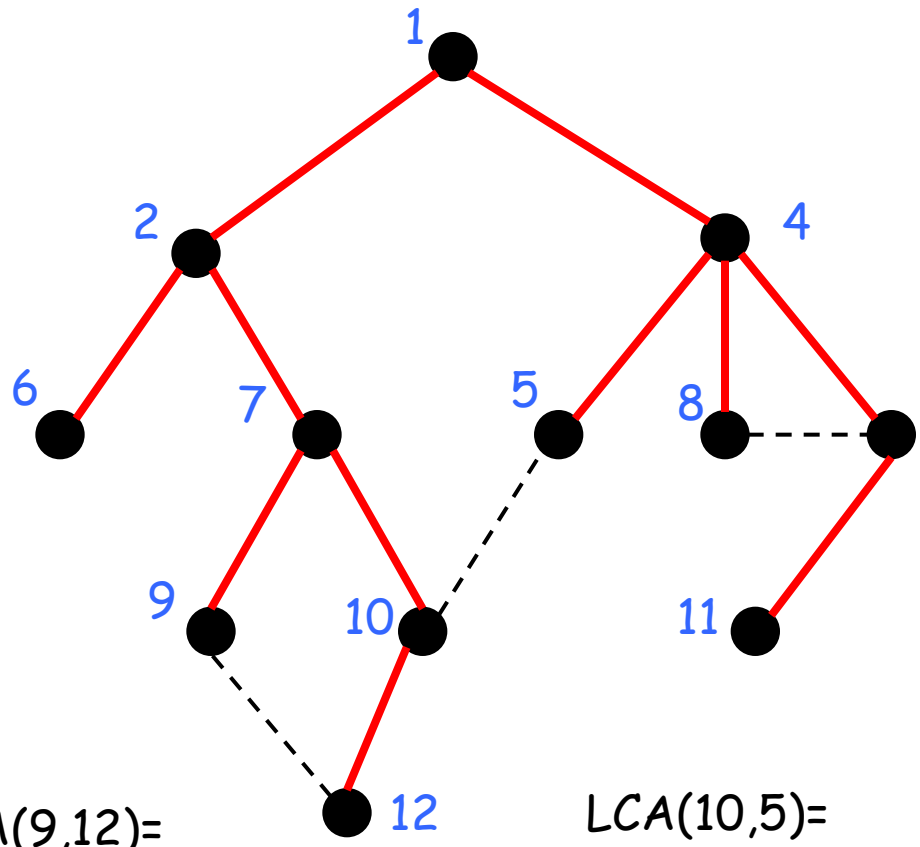


LCA(8,3)=
find(8)=4

LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

l'algoritmo



LCA(8,3)=
find(8)=4

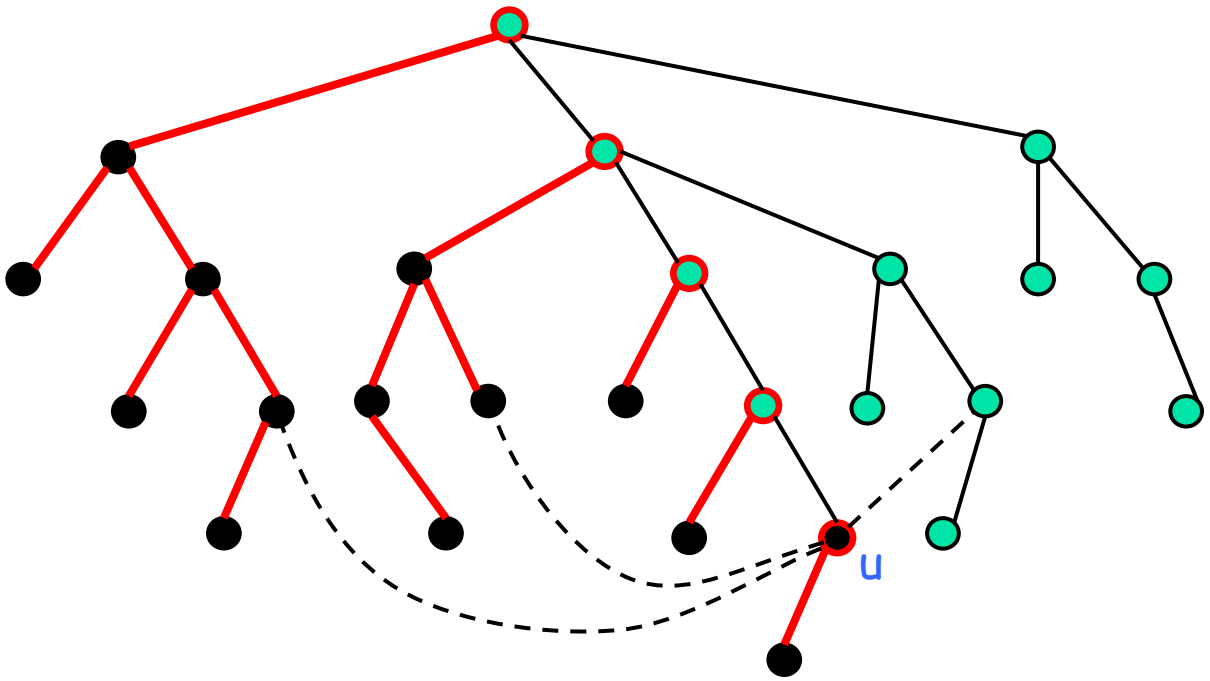
LCA(9,12)=
find(9)=7

LCA(10,5)=
find(10)=1

correttezza

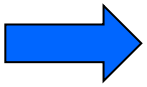
calcolo LCA(u,v) quando sto abbandonando u e v è già stato visitato (abbandonato)

cosa stanno mantenendo gli insiemi disgiunti?



analisi della complessità

- la visita DFS dell'albero – senza contare le operazioni sulla struttura dati Union-find – costa $O(n+m)$
- Costo per le operazioni sulla Union-find:
  - n operazioni di makeSet
  - $n-1$ operazioni union
  - $m=|S|$ operazioni di find



tempo $O(n+m+\text{TempoUF}(n,m))$

tempo $O(n+m\ \alpha(n+m,n))$

Esercizio

Sia $G = (V, E)$ un grafo con pesi positivi distinti sugli archi ed $e \in E$ un arco di G. Progettare un algoritmo lineare in grado di determinare se esiste un MST di G che contiene l'arco e.

Esercizio

Dato un grafo $G = (V, E)$ connesso e pesato sugli archi ed un insieme $U \subseteq V$ di nodi, progettare un algoritmo che calcoli uno spanning tree T di G di costo minimo con il vincolo che tutti i nodi in U siano foglie di T. L'algoritmo deve avere complessità $O(|E| \log |V|)$.

Esercizio

Dato un grafo diretto aciclico (DAG) $G = (V, E)$ e due nodi $s, t \in V$ progettare un algoritmo lineare (in $|V|$ e $|E|$) che calcoli il numero di cammini distinti in G da s a t. Suggerimento: si usi la programmazione dinamica.


The end