Strutture dati per insiemi disgiunti

Scopo

Gestire in modo efficiente una *collezione*

$$\mathbf{S} = \{S_1, S_2, ..., S_k\}$$

di *insiemi disgiunti* qualora le sole operazioni consentite siano:

1) creare un nuovo insieme contenente un solo elemento
 (tale elemento non deve comparire in nessun altro insieme della collezione)

2) individuare a quale insieme appartiene un elemento dato

3) sostituire due insiemi S_i ed S_j con l'insieme $S_i \cup S_j$

Idea

Ogni insieme della collezione è individuato da un *rappresentante* che è uno qualsiasi degli elementi dell'insieme

Individuare a quale insieme appartiene un elemento corrisponde a trovare qual è il rappresentante di quell'insieme

$\mathbf{S} = \{S_1, S_2, S_3\}$

$S_1 = \{b,c,e,h\}$ c è il rappresentante di S_1

$S_2 = \{d,f,g\}$ f è il rappresentante di S_2

$S_3 = \{a\}$ a è il rappresentante di S_3

Procedure

`MakeSet`(x) : aggiunge alla struttura dati un nuovo insieme contenente solo l'elemento $\boldsymbol{x}$.

`FindSet`(x) : ritorna il rappresentante dell'insieme che contiene $\boldsymbol{x}$.

`Union`(x,y) : riunisce i due insiemi U e V contenenti rispettivamente $\boldsymbol{x}$ ed $\boldsymbol{y}$ in un'unico insieme $U \cup V$.

- L'elemento rappresentante dell'unione o è il rappresentante di U o è quello di V
- L'operazione di unione distrugge i vecchi insiemi U e V

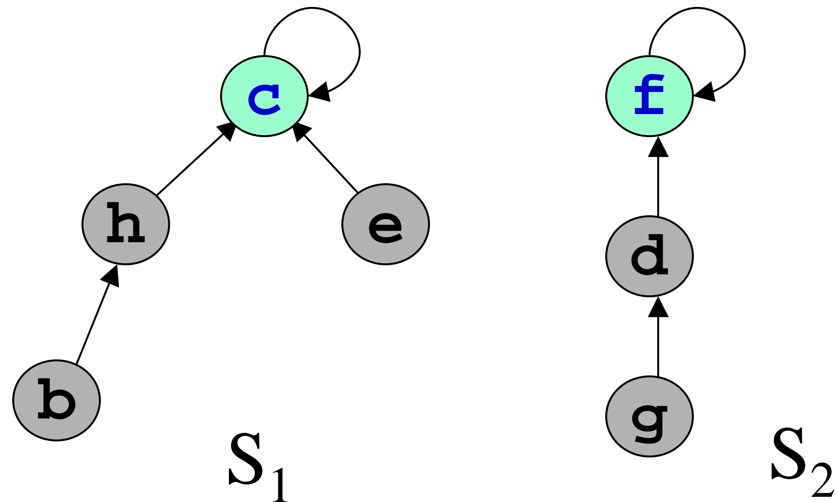
Rappresentazione semplice mediante alberi

- Ogni insieme è rappresentato con un albero;
- ogni nodo dell'albero corrisponde ad un elemento dell'insieme;
- ogni nodo contiene:
  - il campo ***info*** con le informazioni associate all'elemento;
  - il campo ***parent*** che punta al padre;
- il rappresentante dell'insieme è la radice dell'albero

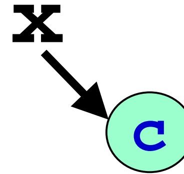
$\mathbf{S} = \{S_1, S_2\}$

$S_1 = \{b,c,e,h\}$

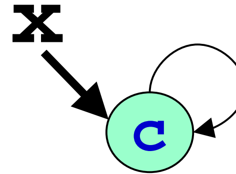
$S_2 = \{d,f,g\}$



```
MakeSet(x)
    parent[x] ← x
```

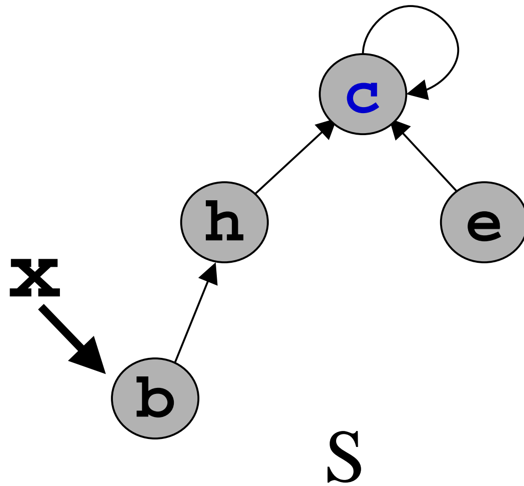


```
MakeSet(x)
    parent[x] ← x
```

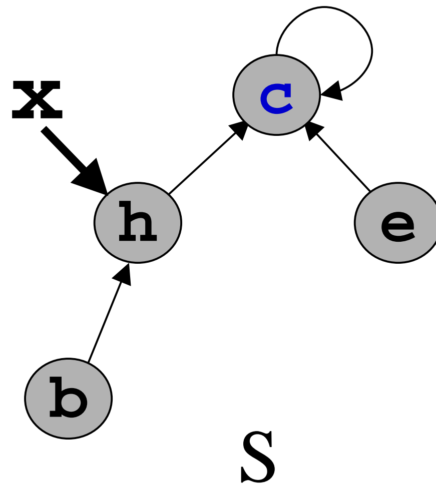


$S = \{$ c $\}$

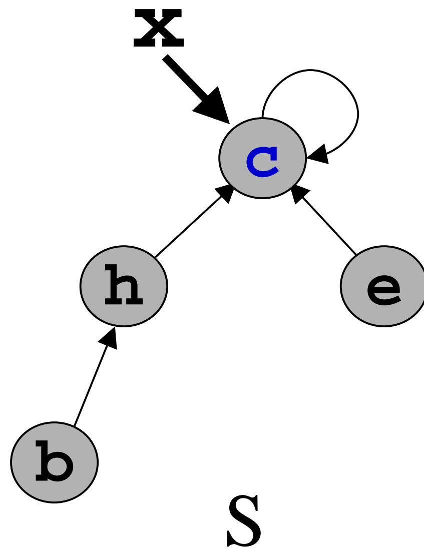
```
FindSet(x)
    while parent[x] ≠ x do
        x ← parent[x]
    return x
```



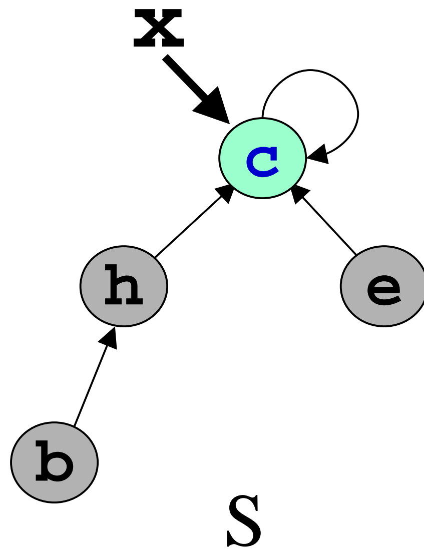
```
FindSet(x)
    while parent[x] ≠ x do
        x ← parent[x]
    return x
```



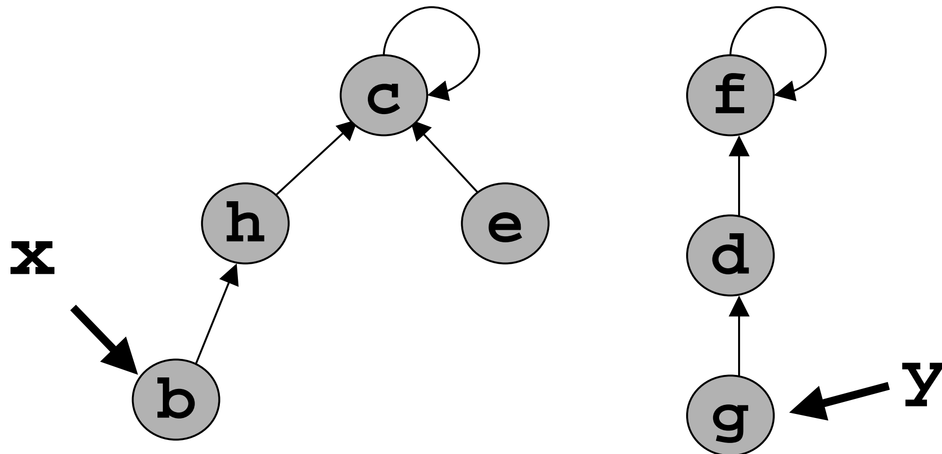
```
FindSet(x)
    while parent[x] ≠ x do
        x ← parent[x]
    return x
```



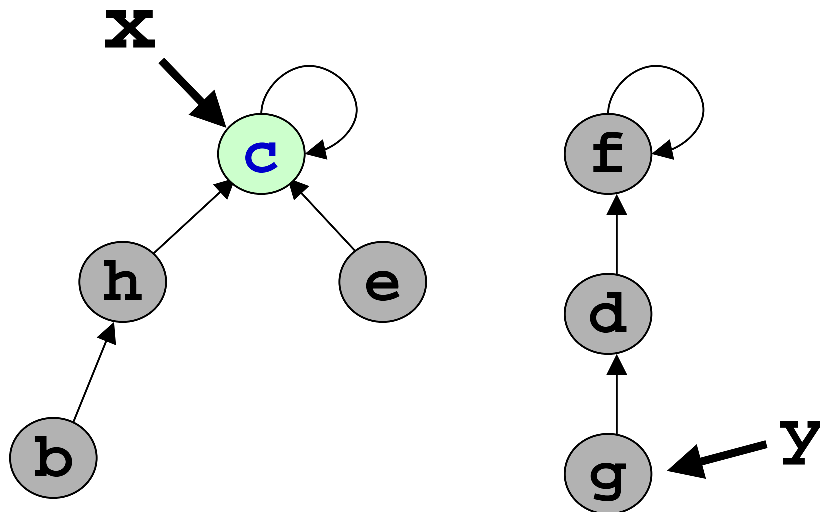
```
FindSet(x)
   while parent[x] ≠ x do
      x ← parent[x]
   return x
```



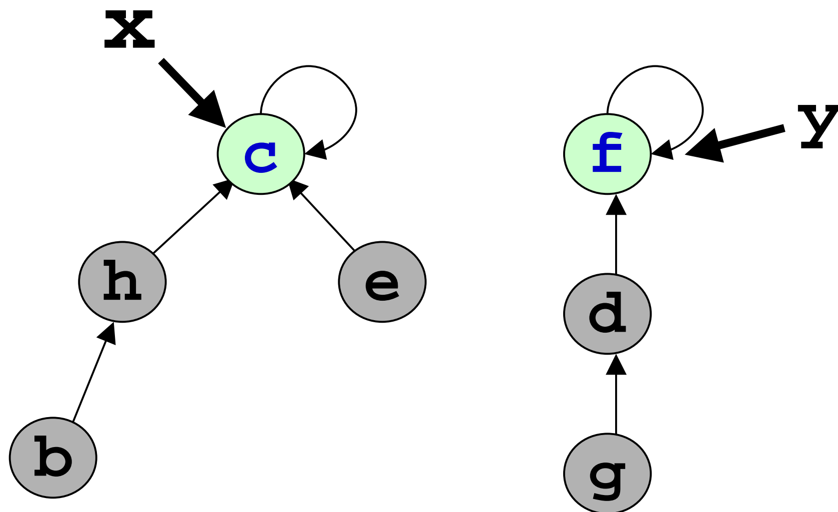
```
Union(x,y)
    x ← FindSet(x)
    y ← FindSet(y)
    parent[x] ← y
```



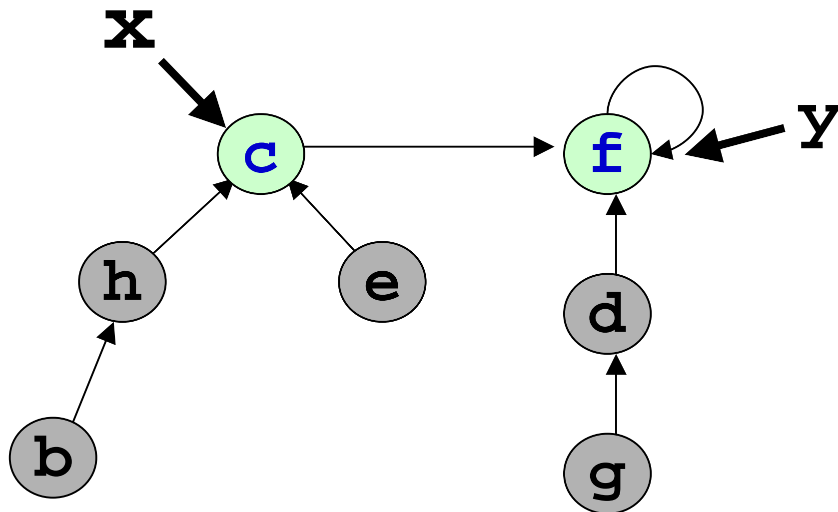
```
Union(x,y)
    x ← FindSet(x)
    y ← FindSet(y)
    parent[x] ← y
```



```
Union(x,y)
    x ← FindSet(x)
    y ← FindSet(y)
    parent[x] ← y
```



```
Union(x,y)
    x ← FindSet(x)
    y ← FindSet(y)
    parent[x] ← y
```



Complessità dell'implementazione semplice

La complessità di **`FindSet`**(x) è proporzionale alla lunghezza del cammino che congiunge il nodo x alla radice dell'albero.

La complessità di **`Union`** è essenzialmente quella delle due chiamate **`FindSet`**(x) e **`FindSet`**(y).

Nel caso peggiore una sequenza di n operazioni può richiedere tempo $O(n^2)$.

Miglioramento rappresentazione mediante alberi

Euristica dell'unione per rango:

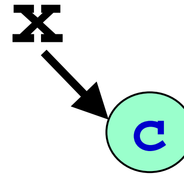
si aggiunge ad ogni nodo ***x*** un campo ***rank*** che è un limite superiore all'altezza del sottoalbero di radice ***x***.

L'operazione **`Union`** mette la radice con ***rank*** minore come figlia di quella con ***rank*** maggiore.

Euristica della compressione dei cammini:

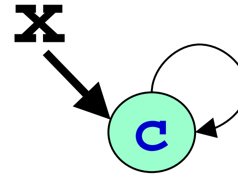
durante l'esecuzione di **`FindSet(`*x*`)`** si fanno puntare alla radice dell'albero i puntatori al padre di tutti i nodi incontrati lungo il cammino da ***x*** alla radice. Le eventuali successive operazioni **`FindSet`** sui nodi di tale cammino risulteranno molto meno onerose.

```
MakeSet(x)
    parent[x] ← x
    rank[x] ← 0
```



Quando viene creato un nuovo insieme con **MakeSet** il rango iniziale dell'unico nodo nell'albero è 0

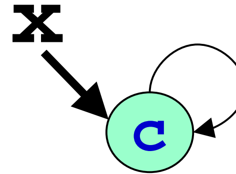
```
MakeSet(x)
    parent[x] ← x
    rank[x] ← 0
```



rank

Quando viene creato un nuovo insieme con **MakeSet** il rango iniziale dell'unico nodo nell'albero è 0

```
MakeSet(x)
    parent[x] ← x
    rank[x] ← 0
```



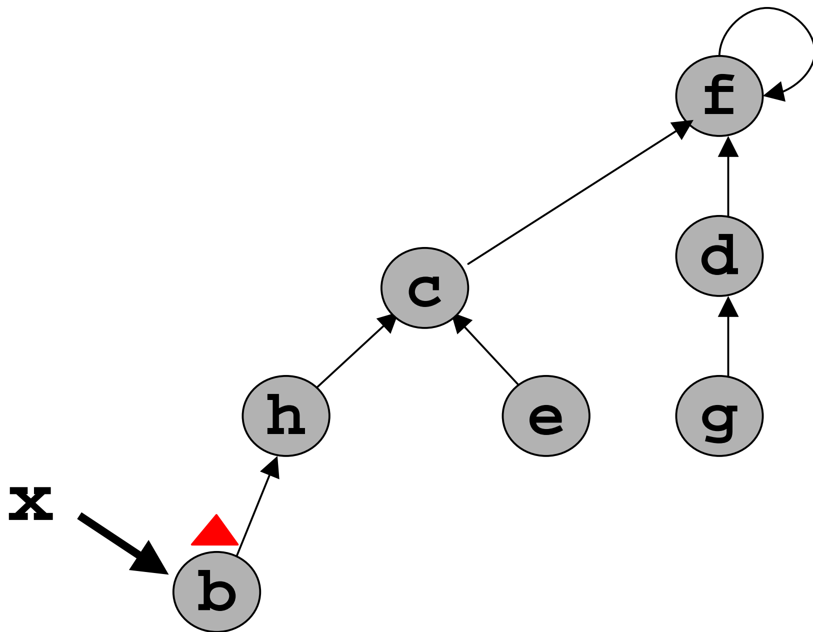
S = {c}

c

rank 0

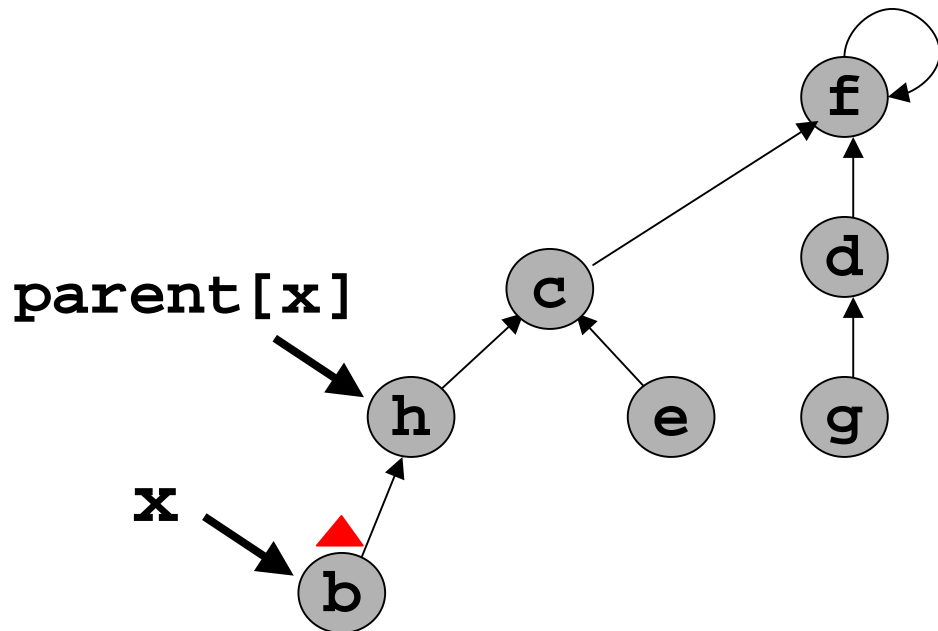
Quando viene creato un nuovo insieme con **MakeSet** il rango iniziale dell'unico nodo nell'albero è 0

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



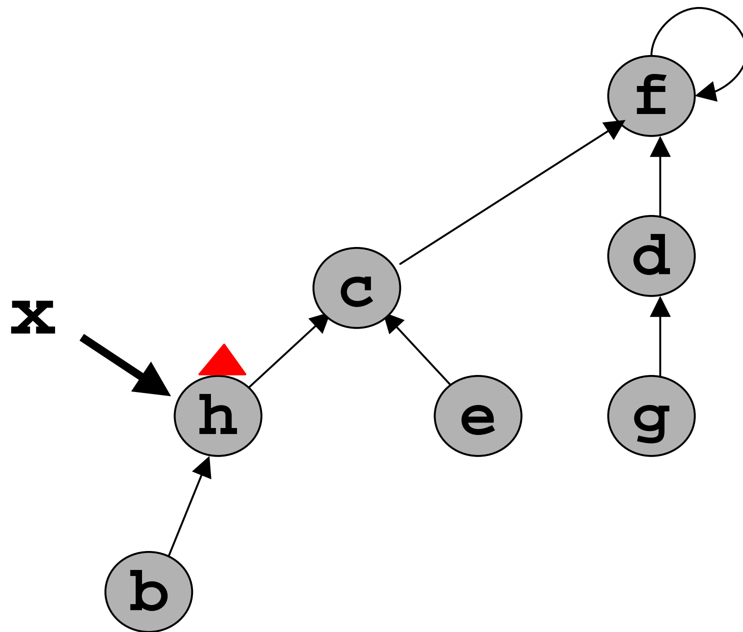
l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



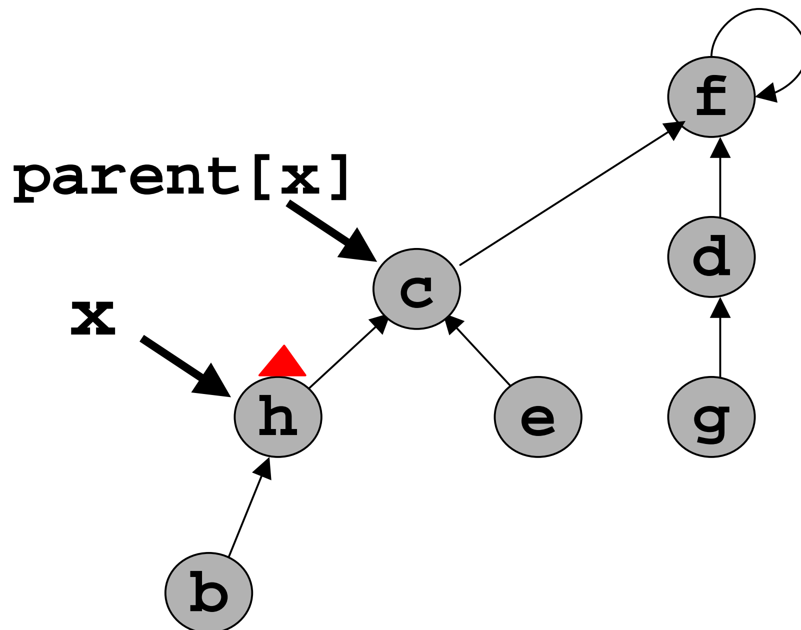
l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



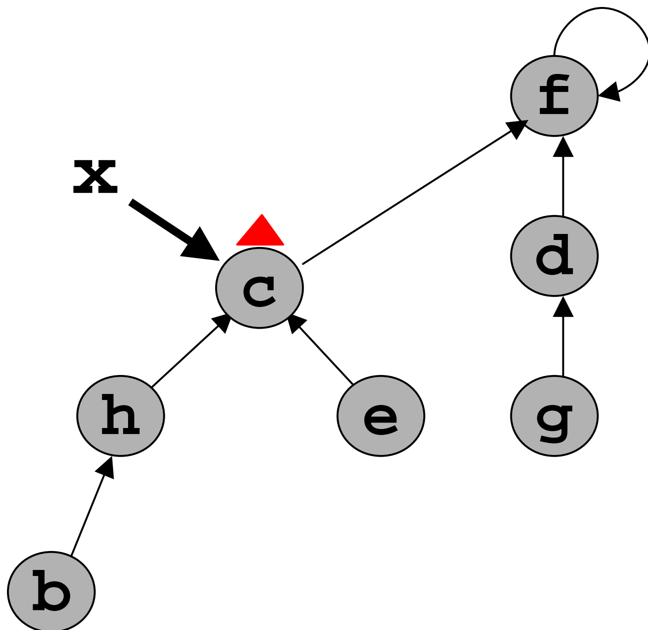
l'esecuzione dell'operazione **`FindSet`** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



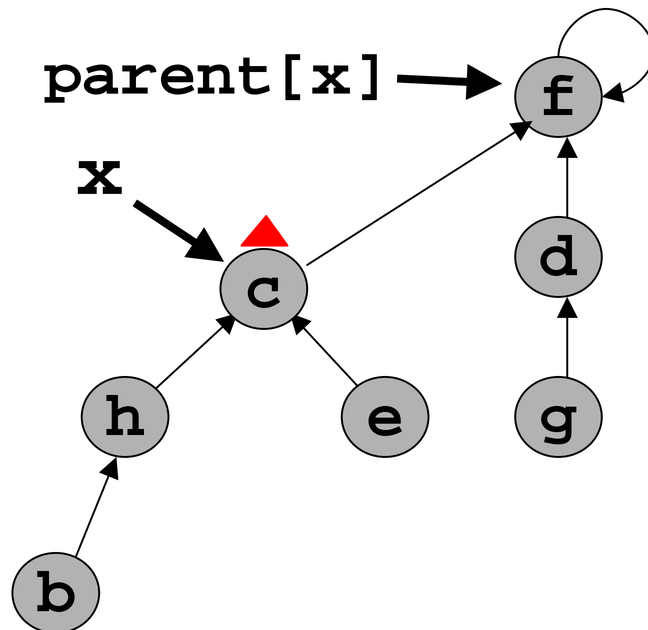
l'esecuzione dell'operazione **`FindSet`** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



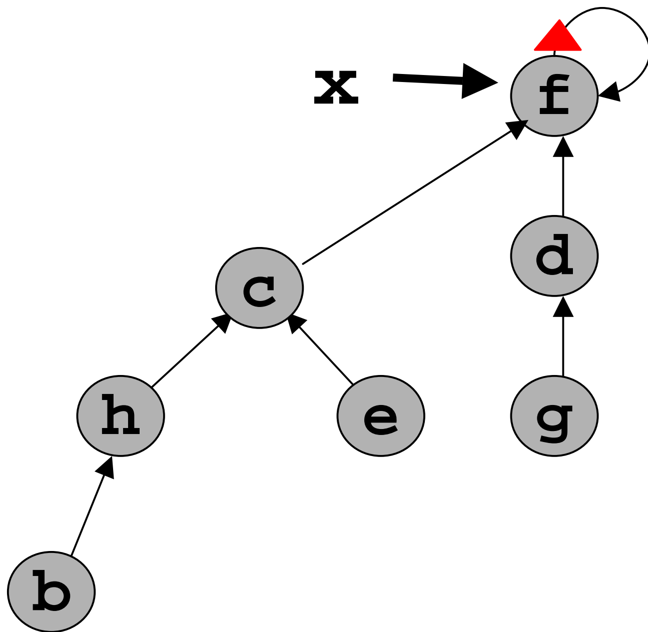
l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



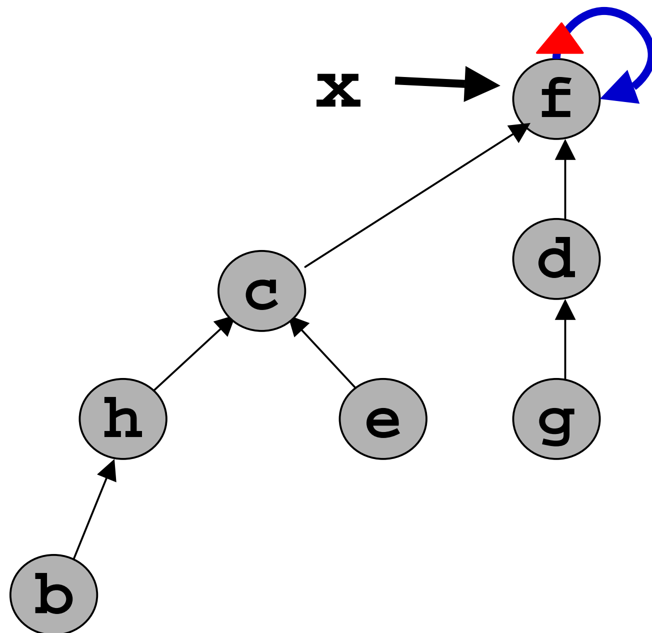
l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



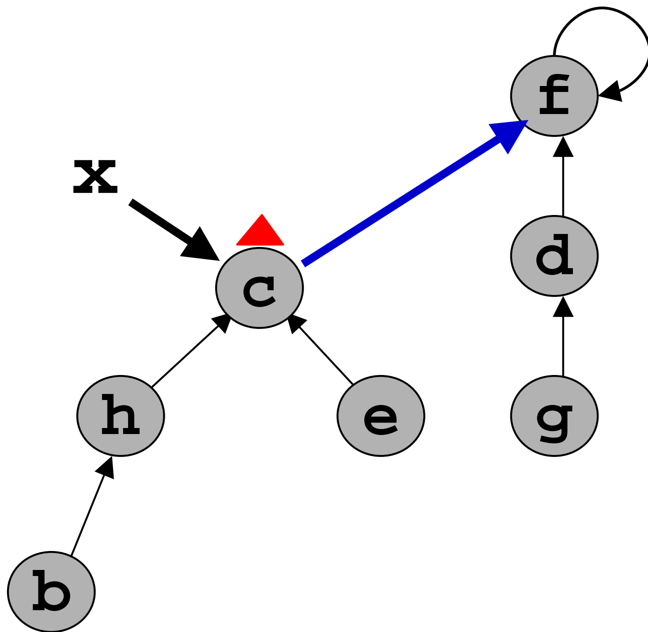
l'esecuzione dell'operazione **`FindSet`** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



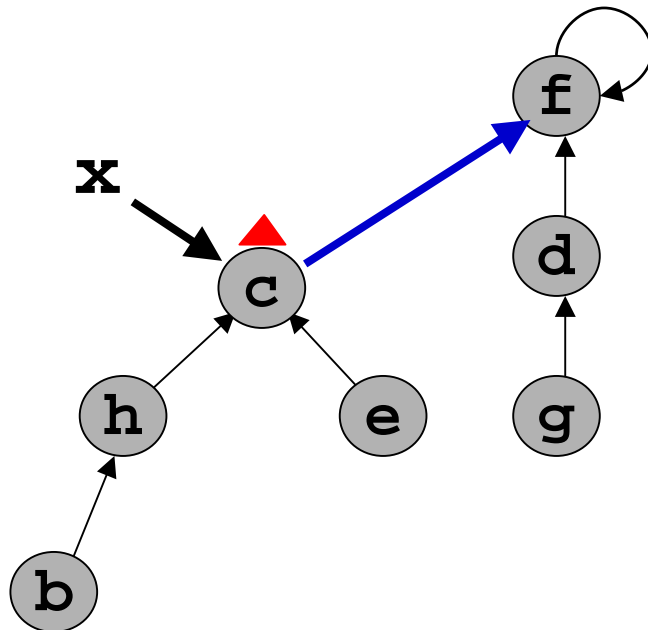
l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
    if parent[x] ≠ x then
        parent[x] ← FindSet(parent[x])
    return parent[x]
```



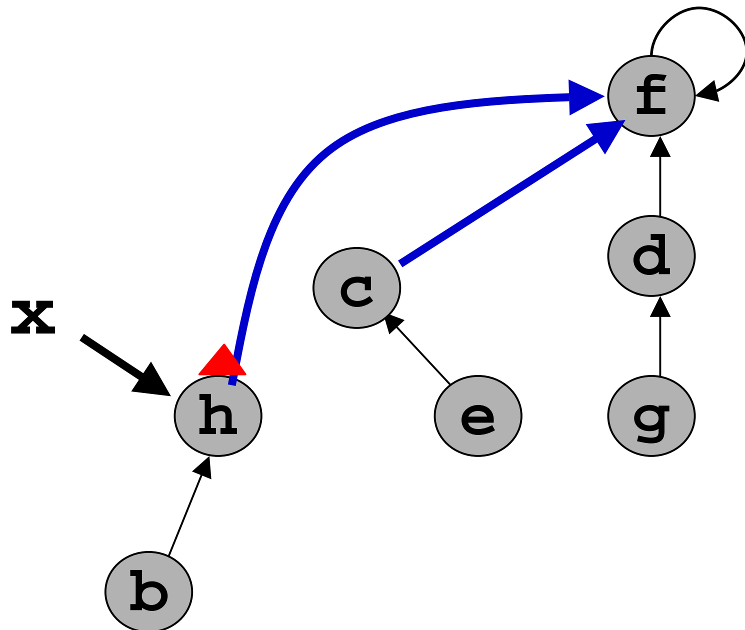
l'esecuzione dell'operazione **`FindSet`** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
    if parent[x] ≠ x then
        parent[x] ← FindSet(parent[x])
    return parent[x]
```



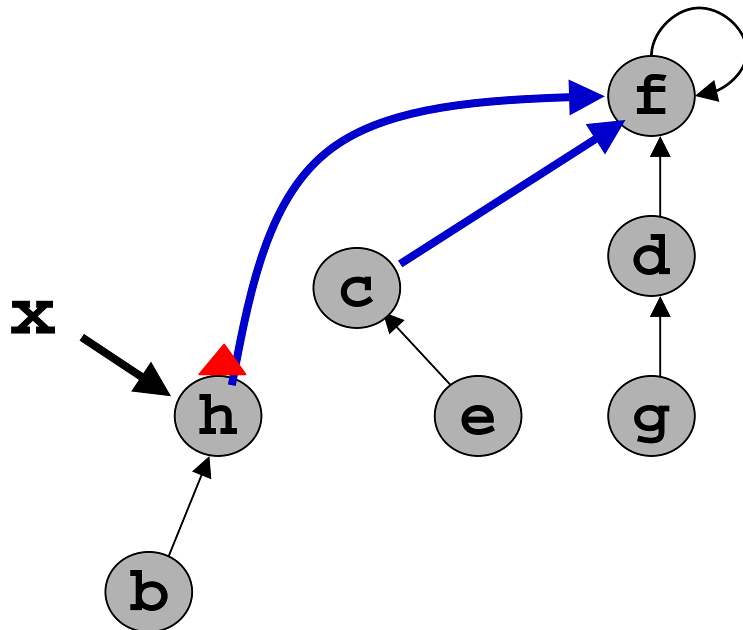
l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
    if parent[x] ≠ x then
        parent[x] ← FindSet(parent[x])
    return parent[x]
```



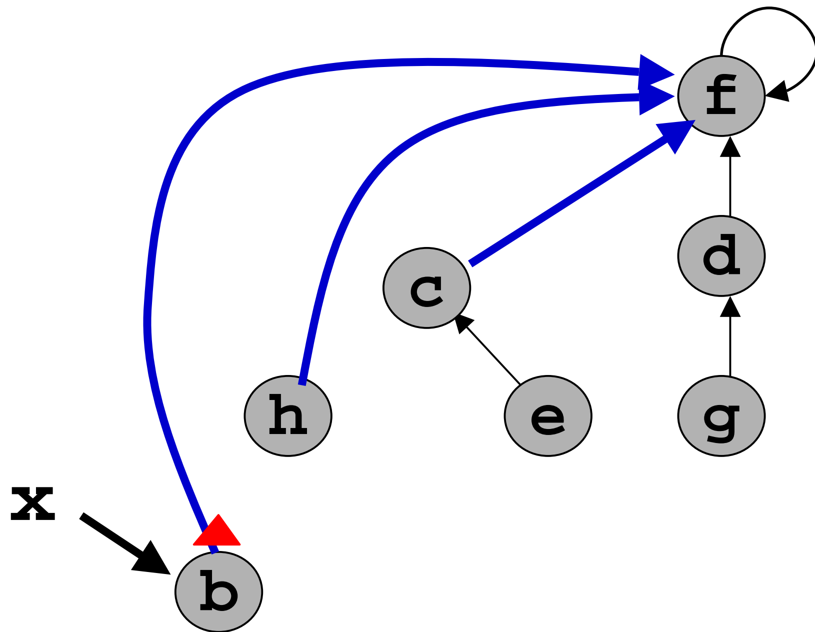
l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



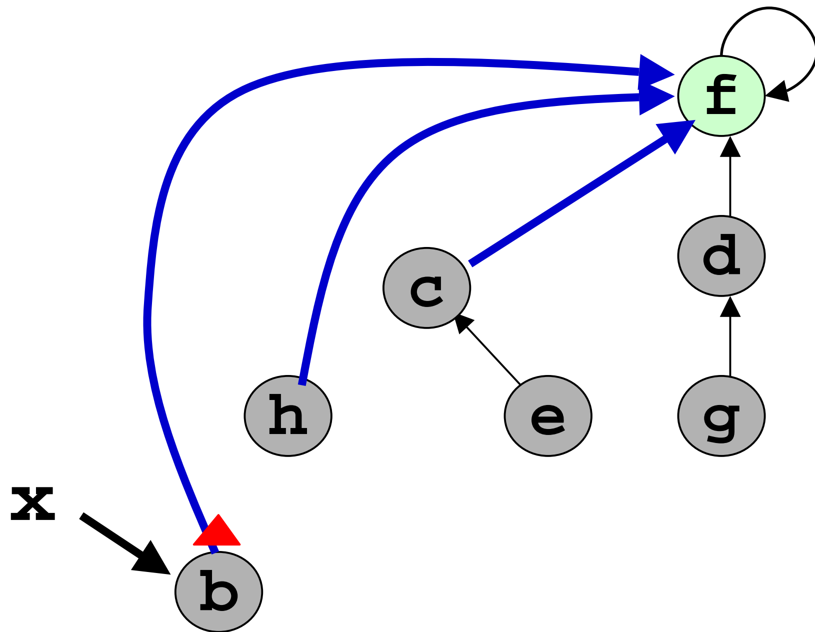
l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
FindSet(x)
   if parent[x] ≠ x then
      parent[x] ← FindSet(parent[x])
   return parent[x]
```



l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

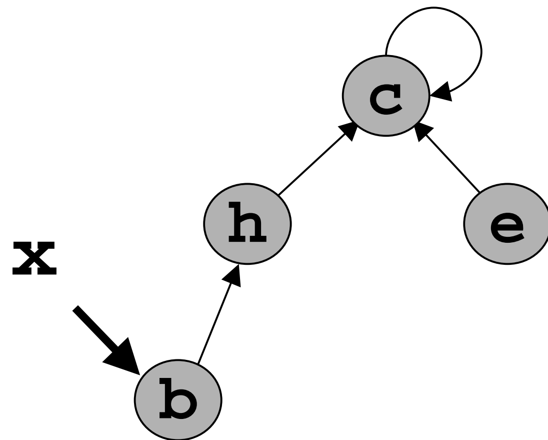
```
FindSet(x)
    if parent[x] ≠ x then
        parent[x] ← FindSet(parent[x])
    return parent[x]
```

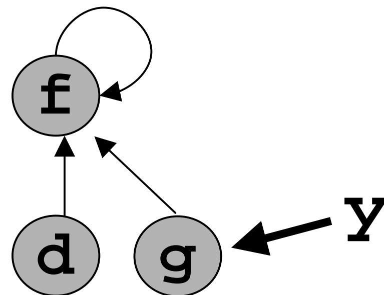


l'esecuzione dell'operazione **FindSet** non modifica i ***rank*** infatti non cambia il limite superiore dell'altezza (riduce l'altezza degli alberi, non la incrementa)

```
Union(x,y)
   x ← FindSet(x)
   y ← FindSet(y)
   Link(x, y)
```

```
Link(x, y)
   if rank[x] > rank[y] then parent[y]← x
   else
    parent[x] ← y
    if rank[x] = rank[y] then rank[y]← rank[y] + 1
```



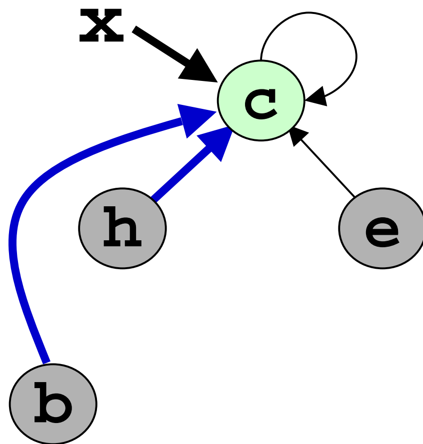


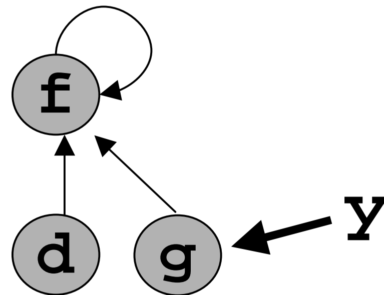
Esempio 1

	b	c	d	e	f	g	h
rank	0	2	0	0	1	0	1

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```



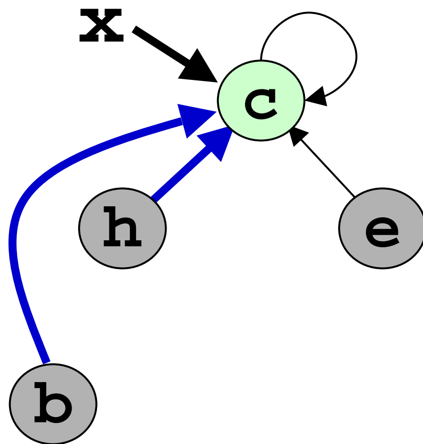


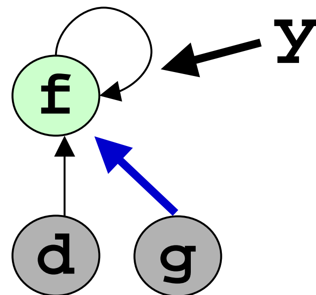
Esempio 1

	b	c	d	e	f	g	h
rank	0	2	0	0	1	0	1

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```



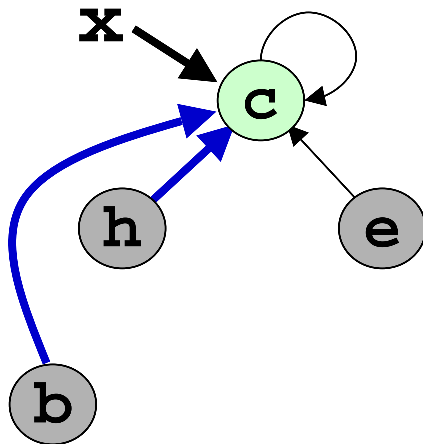


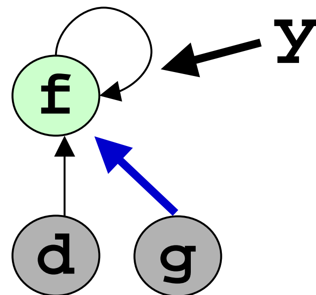
Esempio 1

	b	c	d	e	f	g	h
rank	0	2	0	0	1	0	1

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```





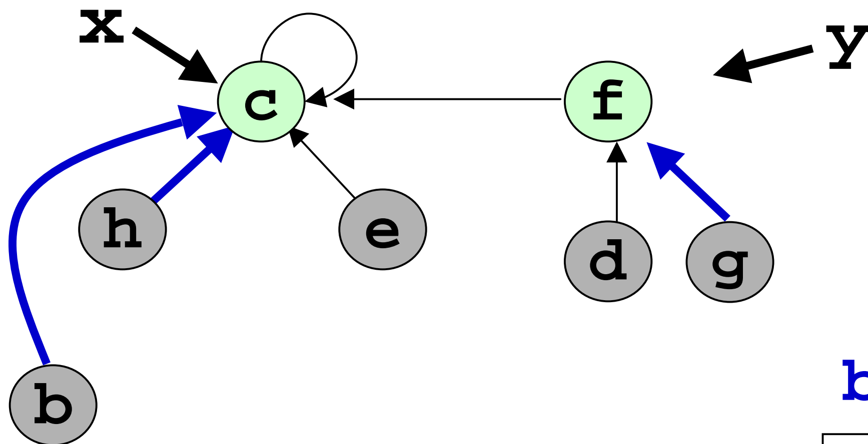
Esempio 1

	b	c	d	e	f	g	h
rank	0	2	0	0	1	0	1

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

si rende la radice con *rank* maggiore padre di quella con *rank* minore; il *rank* della radice non cambia;

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```

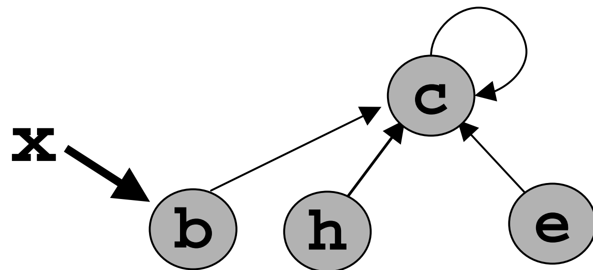


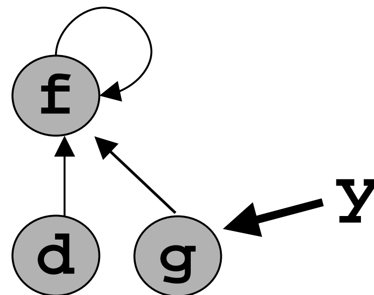
Esempio 1

	b	c	d	e	f	g	h
rank	0	2	0	0	1	0	1

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```





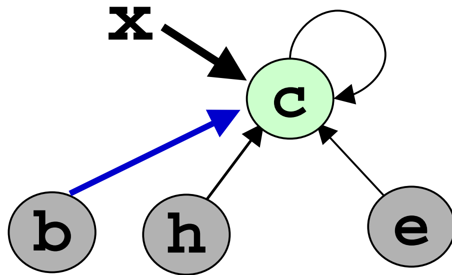
y

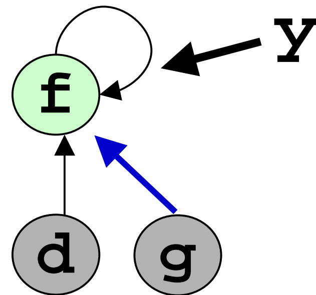
Esempio 2

	b	c	d	e	f	g	h
rank	0	1	0	0	1	0	0

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```



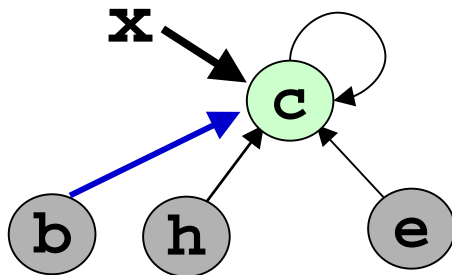


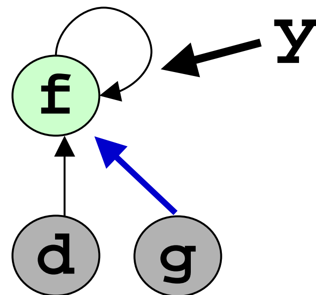
Esempio 2

	b	c	d	e	f	g	h
rank	0	1	0	0	1	0	0

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```



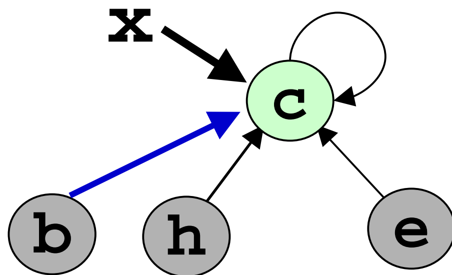


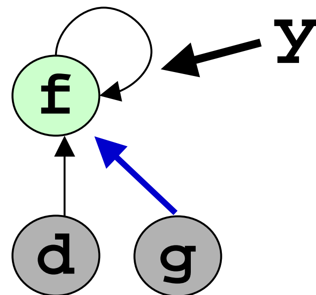
Esempio 2

	b	c	d	e	f	g	h
rank	0	1	0	0	1	0	0

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```



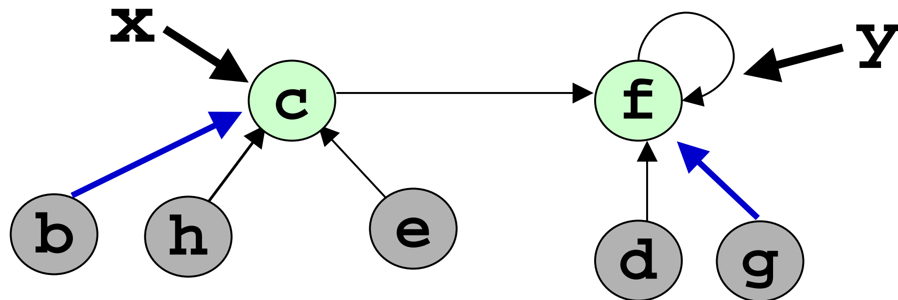


Esempio 2

	b	c	d	e	f	g	h
rank	0	1	0	0	1	0	0

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```

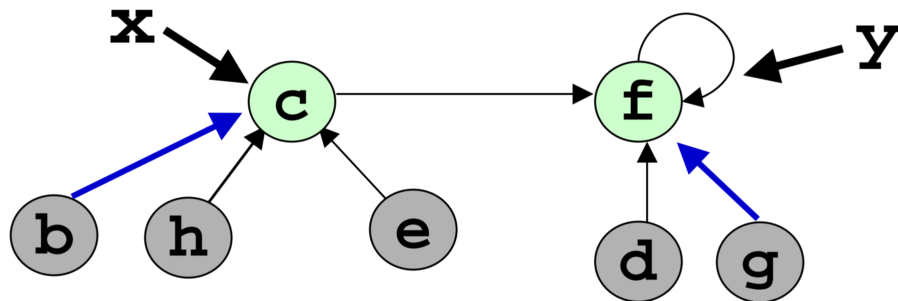


Esempio 2

	b	c	d	e	f	g	h
rank	0	1	0	0	1	0	0

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```



Esempio 2

	b	c	d	e	f	g	h
rank	0	1	0	0	1	0	0

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  Link(x, y)
```

in caso di eguaglianza fra i *rank* si sceglie arbitrariamente una delle due radici come padre; il *rank* della radice aumenta di 1

```
Link(x, y)
  if rank[x] > rank[y] then parent[y]← x
  else
   parent[x] ← y
   if rank[x] = rank[y] then rank[y]← rank[y] + 1
```

x c f y b h e d g

Esempio 2

	b	c	d	e	f	g	h
rank	0	1	0	0	2	0	0

Complessità dell'implementazione migliorata

Con l'euristica dell'unione per rango da sola una sequenza di $\boldsymbol{m}$ operazioni delle quali $\boldsymbol{n}$ sono **`MakeSet`** richiede tempo $\boldsymbol{O(m \log n)}$.

Con l'euristica della compressione dei cammini da sola una sequenza di $\boldsymbol{m}$ operazioni delle quali $\boldsymbol{n}$ sono **`MakeSet`** ed f sono **`FindSet`** richiede tempo

$$\Theta(f \log_{(1+f/n)} n) \qquad \text{se } f \geq n$$

$$\Theta(n + f \log n) \qquad \text{se } f < n$$

Usando entrambe le euristiche una sequenza di $\boldsymbol{m}$ operazioni delle quali $\boldsymbol{n}$ sono **`MakeSet`** richiede tempo $\boldsymbol{O(m\ \alpha(m,n))}$, dove $\alpha(\boldsymbol{m,n})$ è l'inversa della funzione di Ackermann: $\alpha(\boldsymbol{m,n}) \leq \mathbf{4}$ in ogni concepibile uso di una struttura per insiemi disgiunti.

Questo rende praticamente lineare in $\boldsymbol{m}$ la complessità di una sequenza di $\boldsymbol{m}$ operazioni e quindi costante la complessità ammortizzata di una singola operazione.