Strutture dati per insiemi disgiunti

Servono a mantenere una collezione

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

di *insiemi disgiunti*.

Ogni insieme S_i è individuato da un *rappresentante* che è un particolare elemento dell'insieme.

Operazioni sugli insiemi disgiunti:

MakeSet*(*x*)** : aggiunge alla struttura dati un nuovo insieme singoletto { ***x }.

Si richiede che ***x*** non compaia in nessun altro insieme della struttura.

FindSet*(*x*)** : ritorna il rappresentante dell'insieme che contiene ***x.

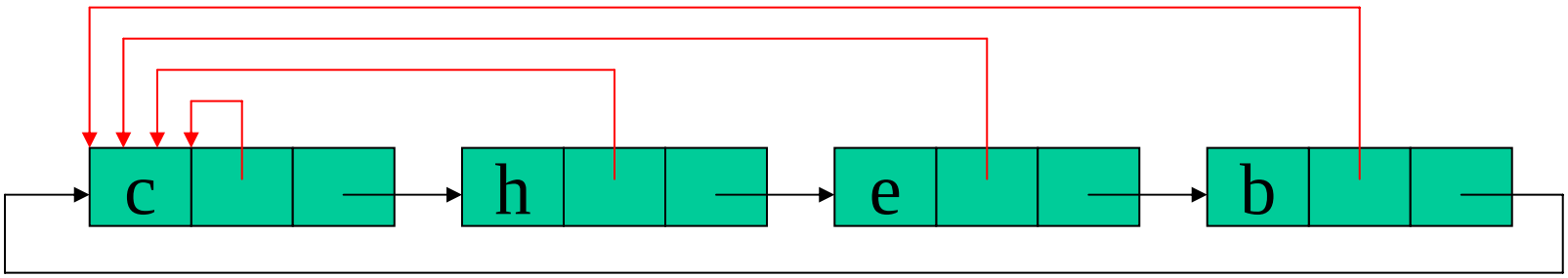
Union*(*x*,*y*)** : unisce i due insiemi contenenti ***x ed ***y*** in un'unico insieme.

Esercizio 29.

Descrivere un algoritmo che utilizza una struttura dati per insiemi disgiunti per determinare se un grafo non orientato G = (V,E) è connesso.

Rappresentazione con liste

Si usa una lista circolare per ciascun insieme.



Ogni nodo ha
- un puntatore al nodo successivo
- un puntatore al nodo rappresentante.

I nodi hanno i seguenti campi:

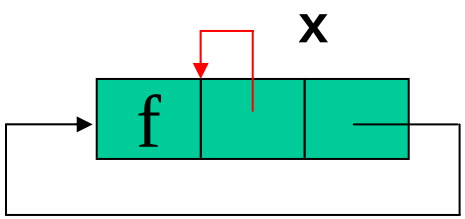
info : l'informazione contenuta nel nodo

rappr : il puntatore al rappresentante

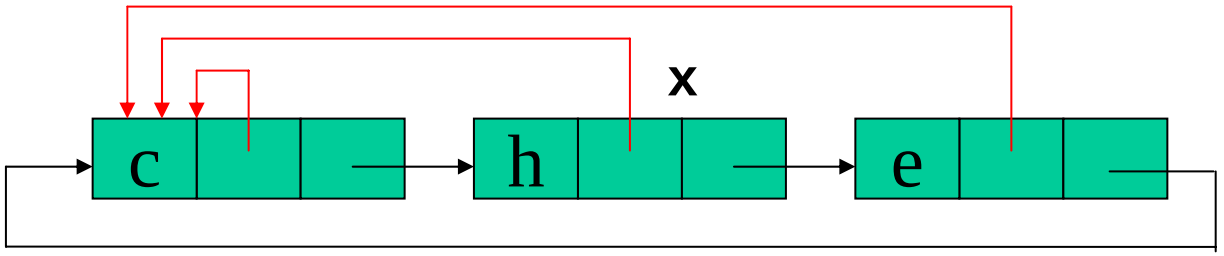
succ : il puntatore al nodo successivo

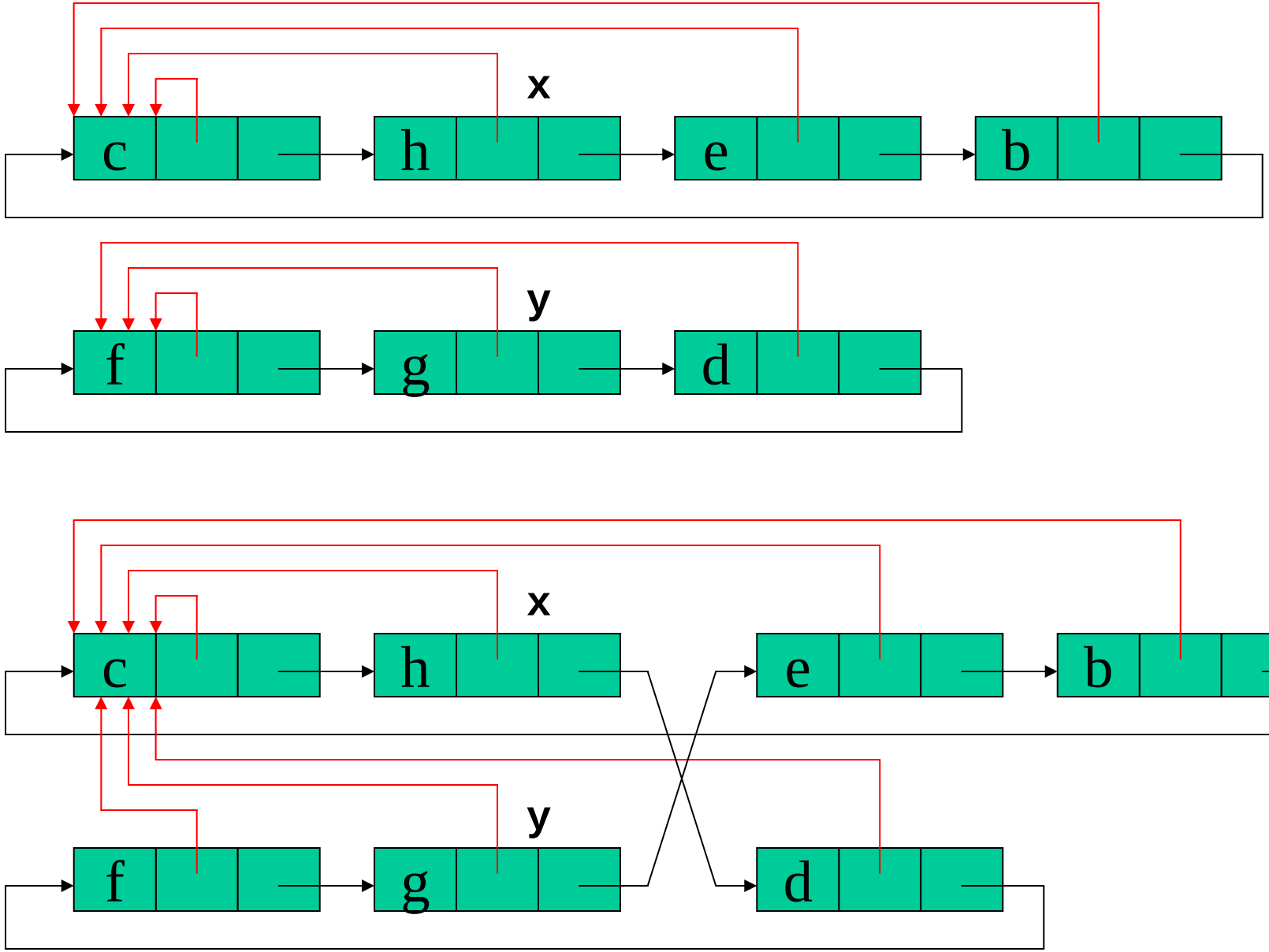
Le operazioni sono:

***MakeSet*(*x*)**
    rappr*[*x*] ← *x
    succ*[*x*] ← *x



***FindSet*(*x*)**
    return *rappr*[*x*]




x
c
h
e
b
y
f
g
d
x
c
h
e
b
y
f
g
d

```
Union(x, y)
    ▷ cambia i puntatori al rappresentante nella lista di y
    rappr[y] ← rappr[x], z ← succ[y]
    while z ≠ y do
        rappr[z] ← rappr[x], z ← succ[z]
    ▷ concatena le due liste
    z ← succ[x], succ[x] ← succ[y], succ[y] ← z
```

La complessità di ***Union*** dipende dal numero di iterazioni richieste dal ciclo che cambia i puntatori al rappresentante dei nodi della lista contenente ***y***.

Quindi ***Union*** ha complessità $O(n_y)$ dove n_y è la lunghezza della seconda lista.

Consideriamo la seguente sequenza di ***2n-1*** operazioni:

MakeSet(x_1)	▷ **costo 1**
MakeSet(x_2)	▷ **costo 1**
.....	
MakeSet(x_n)	▷ **costo 1**
Union(x_2, x_1)	▷ **costo 1**
Union(x_3, x_1)	▷ **costo 2**
Union(x_4, x_1)	▷ **costo 3**
.....	
Union(x_n, x_1)	▷ **costo *n*-1**

Il costo totale è proporzionale ad $\boldsymbol{n + n(n-1)/2}$ ed è quindi $\boldsymbol{\Theta(n^2)}$. Di conseguenza ***Union*** ha costo ammortizzato $\boldsymbol{O(n)}$.

Euristica dell'unione pesata

La complessità $\Theta(n^2)$ dell'esempio è dovuta al fatto che in ogni ***Union*** la seconda lista (percorsa per aggiornare i puntatori al rappresentante) è la più lunga delle due.

L'euristica dell'*unione pesata* consiste nello scegliere sempre la lista più corta per aggiornare i puntatori al rappresentante.

Si memorizza la lunghezza di della lista in un nuovo campo ***l*** del rappresentante.

Le nuove funzioni:

```
MakeSet(x)
    l[x] ← 1
    succ[x] ← x
    rappr[x] ← x
```

```
FindSet(x)
    return rappr[x]
```

```
Union(x, y)
    x ← rappr[x]
    y ← rappr[y]
    ▷ se la lista di x è più corta scambia x con y
    if l[x] < l[y] then
        z ← x, x ← y, y ← z
    ▷ somma le lunghezze
    l[x] ← l[x] + l[y]
    ▷ cambia rappresentante alla lista di y
    rappr[y] ← x, z ← succ[y]
    while z ≠ y do
        rappr[z] ← x, z ← succ[z]
    ▷ concatena le due liste
    z ← succ[x], succ[x] ← succ[y], succ[y] ← z
```

Complessità con l'euristica dell'unione pesata.

Una sequenza di ***m*** operazioni ***MakeSet***, ***Union*** e ***FindSet*** delle quali ***n*** sono ***MakeSet***, richiede tempo $O(m + n \log n)$.

La complessità ammortizzata delle operazioni è:

$$O(\frac{m + n \log n}{m}) = O(1 + \frac{n \log n}{m}) = O(\log n)$$

Se il numero di ***MakeSet*** è molto minore del numero di ***Union*** e ***FindSet*** per cui $n \log n = O(m)$

$$O(1 + \frac{n \log n}{m}) = O(1)$$

Dimostriamo quindi che una sequenza di m operazioni ***MakeSet***, ***Union*** e ***FindSet*** delle quali n sono ***MakeSet***, richiede tempo $O(m + n \log n)$.

Dimostrazione.

Tutte le operazioni richiedono un tempo costante eccetto ***Union*** che richiede un tempo costante più un tempo proporzionale al numero di puntatori al rappresentante che vengono modificati.

Il tempo richiesto dalla sequenza di ***m*** operazioni è quindi ***O*(*m* + *N*)** dove ***N*** è il numero totale di aggiornamenti dei puntatori al rappresentante eseguiti durante tutta la sequenza di operazioni.

Il numero massimo di oggetti contenuti nella struttura è ***n***, pari al numero di ***MakeSet***.

Quando un oggetto ***x*** viene creato esso appartiene ad un insieme di cardinalità 1.

Il rappresentante di x viene aggiornato quando l'insieme contenente x viene unito ad un insieme di cardinalità maggiore o uguale.

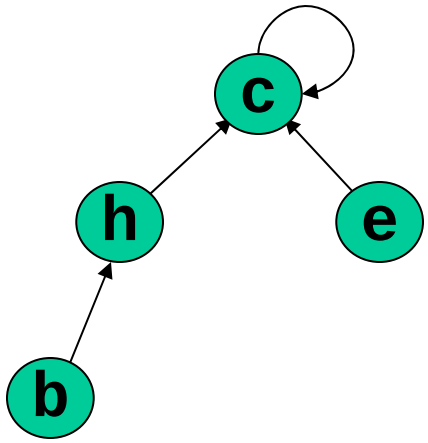
Ogni volta che viene aggiornato il puntatore al rappresentante di x la cardinalità dell'insieme a cui appartiene x viene almeno raddoppiata.

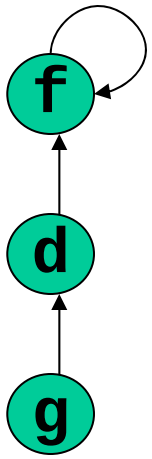
Siccome n è la massima cardinalità di un insieme il puntatore al rappresentante di x può essere aggiornato al più $\log_2 n$ volte.
Quindi $N \leq n \log_2 n$.

Rappresentazione con foreste

Una rappresentazione più efficiente si ottiene usando *foreste di insiemi disgiunti*.

Con questa rappresentazione ogni insieme è rappresentato con un albero i cui nodi, oltre al campo ***info*** che contiene l'informazione, hanno soltanto un campo ***p*** che punta al padre.


c
h
e
b

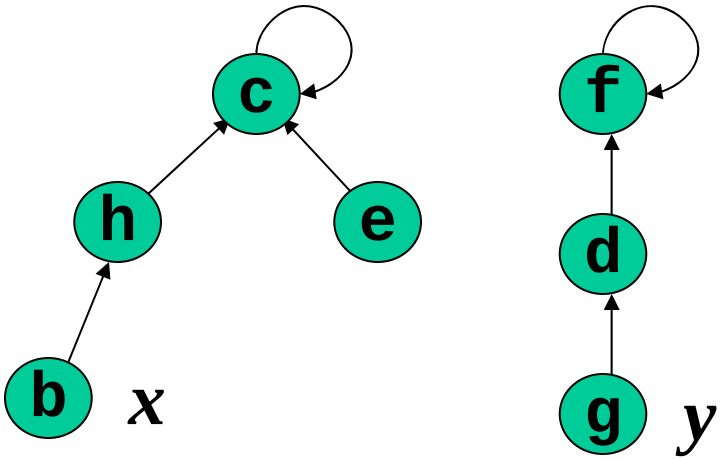

f
d
g

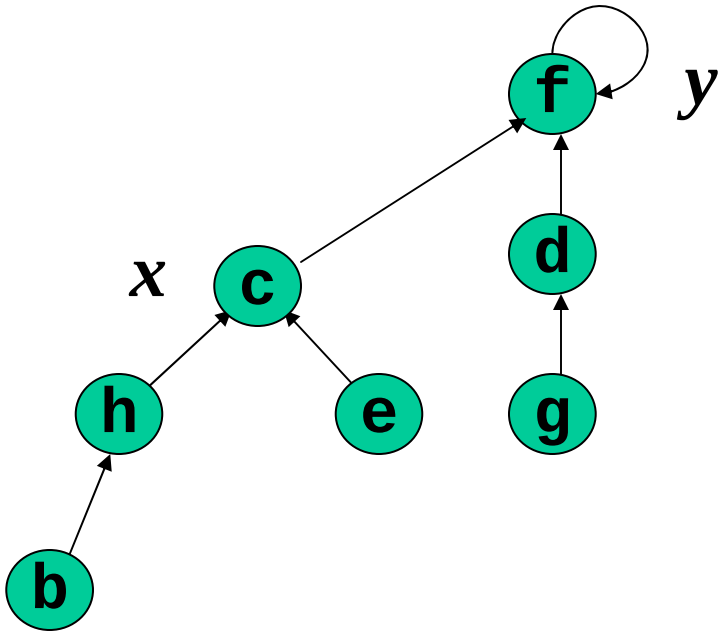
Implementazione semplice:

```
MakeSet(x)
  p[x] ← x
```

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

```
Union(x,y)
  x ← FindSet(x)
  y ← FindSet(y)
  p[x] ← y        ▷ serve controllare se x ≠ y?
```


c
h
e
b
x
f
d
g
y


f
y
x
c
d
h
e
g
b

Osservazione.

Sia nella rappresentazione con liste circolari che nella rappresentazione mediante alberi non abbiamo indicato nessun puntatore esterno alla lista o all'albero.

In realtà una struttura dati per insiemi disgiunti non è pensata per memorizzare dei dati ma soltanto per raggruppare in insiemi disgiunti dei dati che sono già memorizzati in qualche altra struttura (array, pila, albero, tavola hash, ecc.).

Complessità dell'implementazione semplice.

- ***FindSet*(*x*)** : proporzionale alla lunghezza del cammino che congiunge il nodo ***x*** alla radice dell'albero.

- ***Union*** : essenzialmente quella delle due chiamate ***FindSet*(*x*)** e ***FindSet*(*y*)**.

Un esempio analogo a quello usato con le liste mostra che una sequenza di ***n*** operazioni può richiedere tempo $O(n^2)$.

Possiamo migliorare notevolmente l'efficienza usando due euristiche:

Unione per rango

Simile all'unione pesata per le liste.

Per ogni nodo ***x*** manteniamo un campo ***rank*** che è un limite superiore all'altezza del sottoalbero di radice ***x*** (e un limite inferiore del logaritmo del numero di nodi del sottoalbero).

L'operazione ***Union*** inserisce la radice con rango minore come figlia di quella di rango maggiore.

Compressione dei cammini

Quando effettuiamo una ***FindSet*(*x*)** dobbiamo attraversare tutto il cammino da ***x*** alla radice.

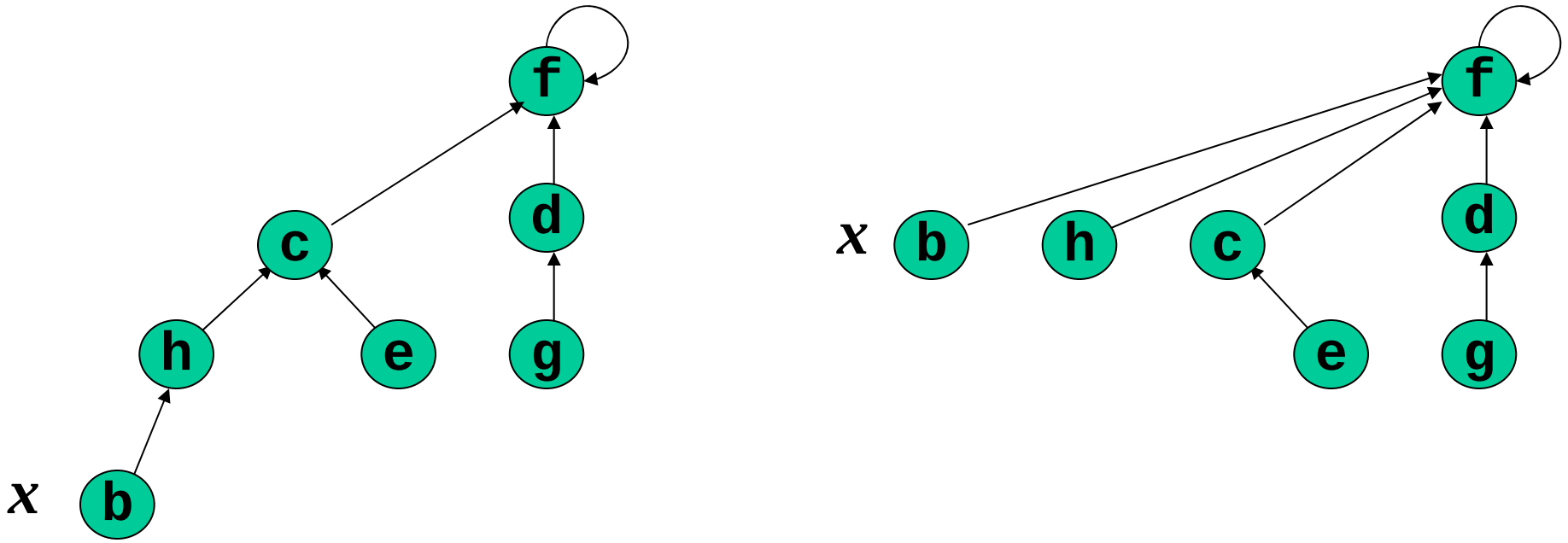
Possiamo approfittarne per far puntare alla radice dell'albero i puntatori al padre di tutti i nodi incontrati lungo il cammino.

In questo modo le eventuali operazioni ***FindSet*** successive sui nodi di tale cammino risulteranno molto meno onerose.

L’implementazione con le due euristiche è la seguente:

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

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



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

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