Strutture Merge-Find

Gianluigi Zavattaro
Dipartimento di Informatica—Scienza e Ingegneria
Università di Bologna

gianluigi.zavattaro@unibo.it

Struttura dati per insiemi disgiunti

- Operazioni fondamentali:
  - Creare *n* insiemi composti da un singolo elemento; assumiamo che gli insiemi siano {1}, {2}, … {*n*}
  - Unire due insiemi
  - Identificare l'insieme a cui appartiene un elemento
- Ogni insieme è identificato da un rappresentante univoco
  - Il rappresentante è un qualsiasi membro dell'insieme
  - Operazioni di ricerca del rappresentante su uno stesso insieme devono restituire sempre lo stesso elemento
  - Solo in caso di unione con altro insieme il rappresentante può cambiare

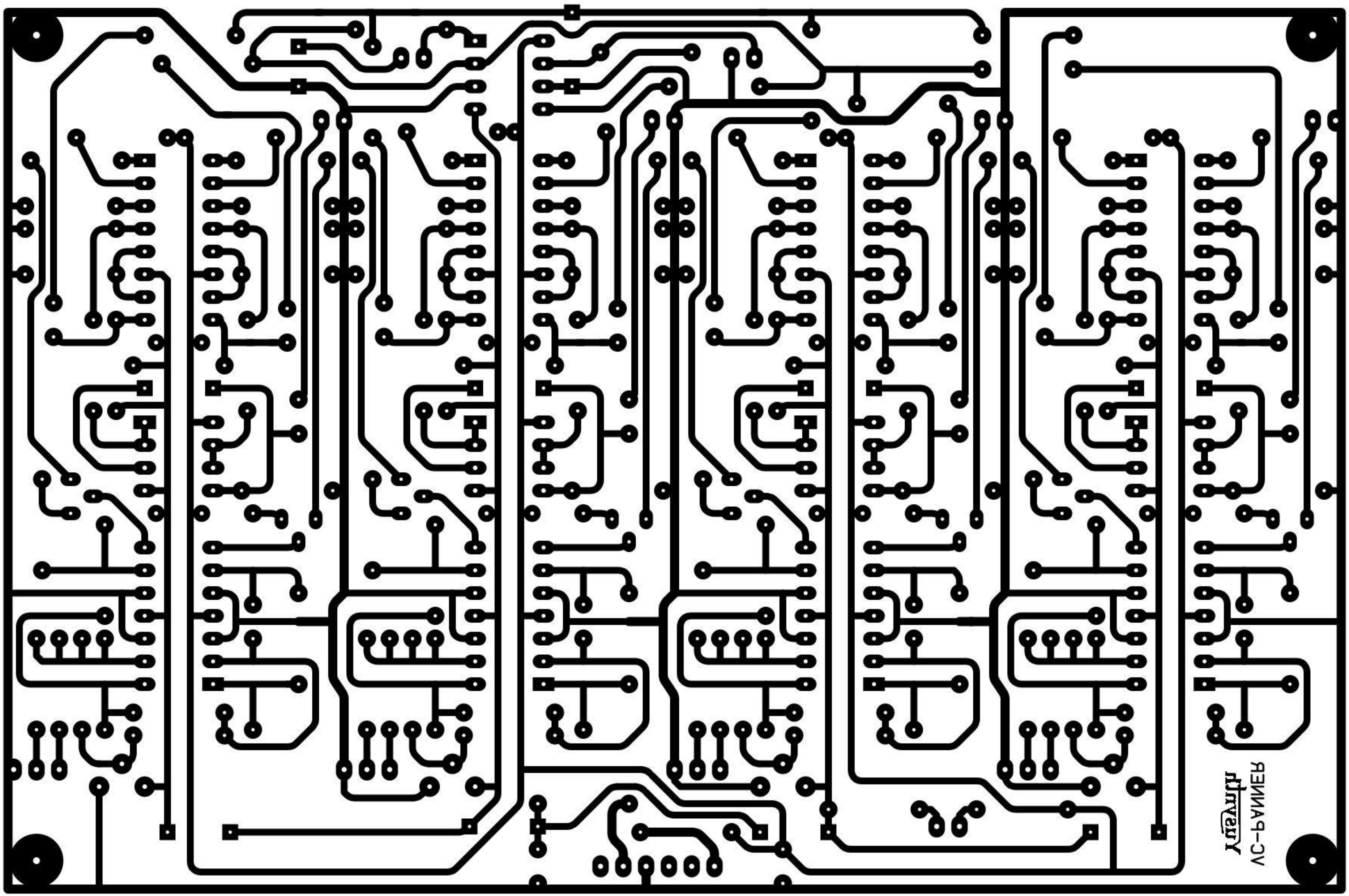
Operazioni su strutture Merge-Find

- **`Mfset(integer n)`**
  - Crea *n* insiemi disgiunti {1}, {2}, … {*n*}
- **`integer find(integer x)`**
  - Restituisce il rappresentante dell'unico insieme contenente *x*
- **`merge(integer x, integer y)`**
  - Unisce i due insiemi che contengono *x* e *y* (se *x* e *y* appartengono già allo stesso insieme, non fa nulla)
  - Il rappresentante puo' essere scelto in modo arbitrario; ad esempio, come uno dei vecchi rappresentanti degli insiemi contenenti *x* e *y*.

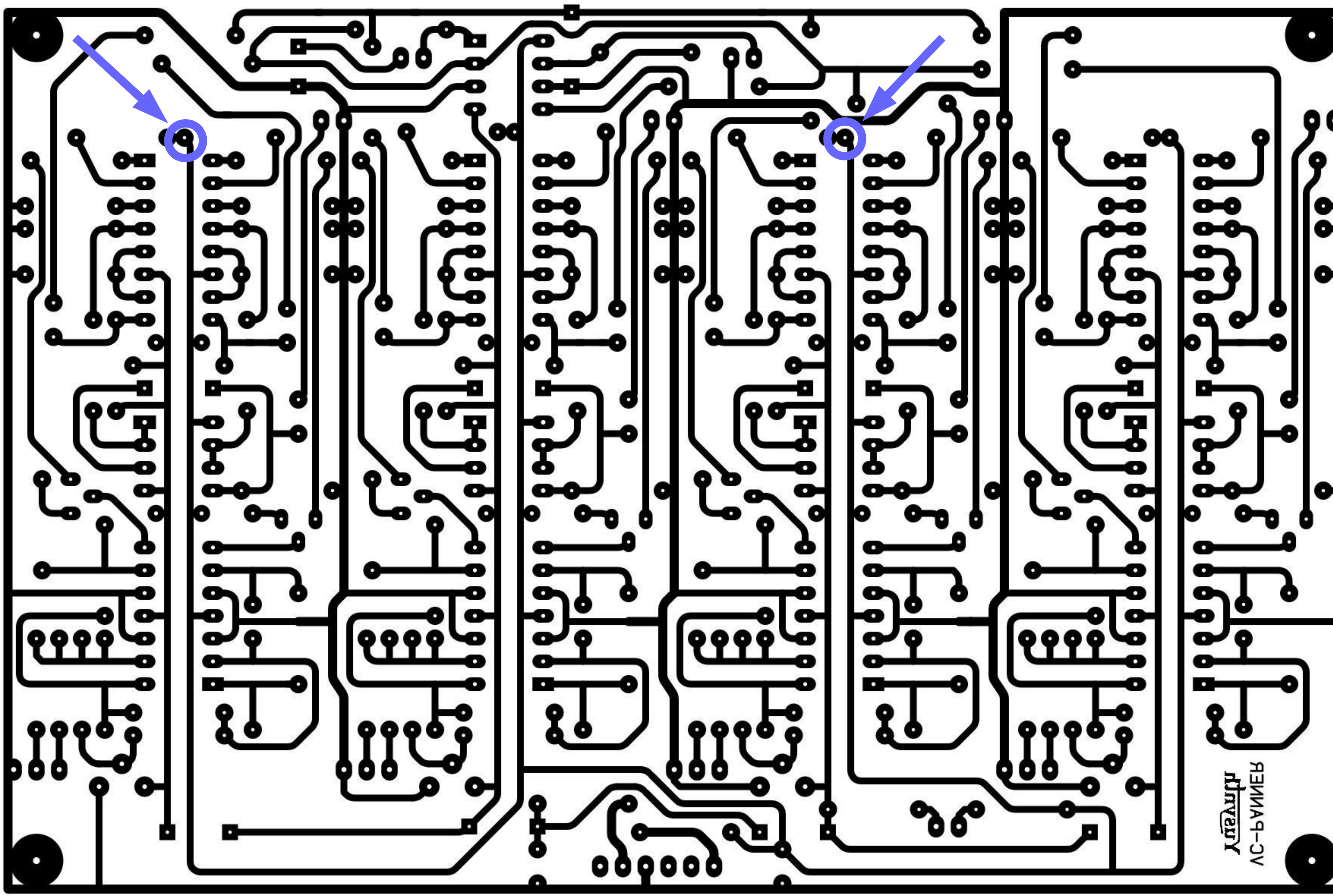
Esempio
(i valori sottolineati indicano il rappresentante)

`Mfset(6)`	1 · 2 · 3 · 4 · 5 · 6
`merge(1,2)`	1, 2 · 3 · 4 · 5 · 6
`merge(3,4)`	1, 2 · 3, 4 · 5 · 6
`merge(5,6)`	1, 2 · 3, 4 · 5, 6
`merge(2,3)`	1, 2, 3, 4 · 5, 6
`merge(4,6)`	1, 2, 3, 4, 5, 6

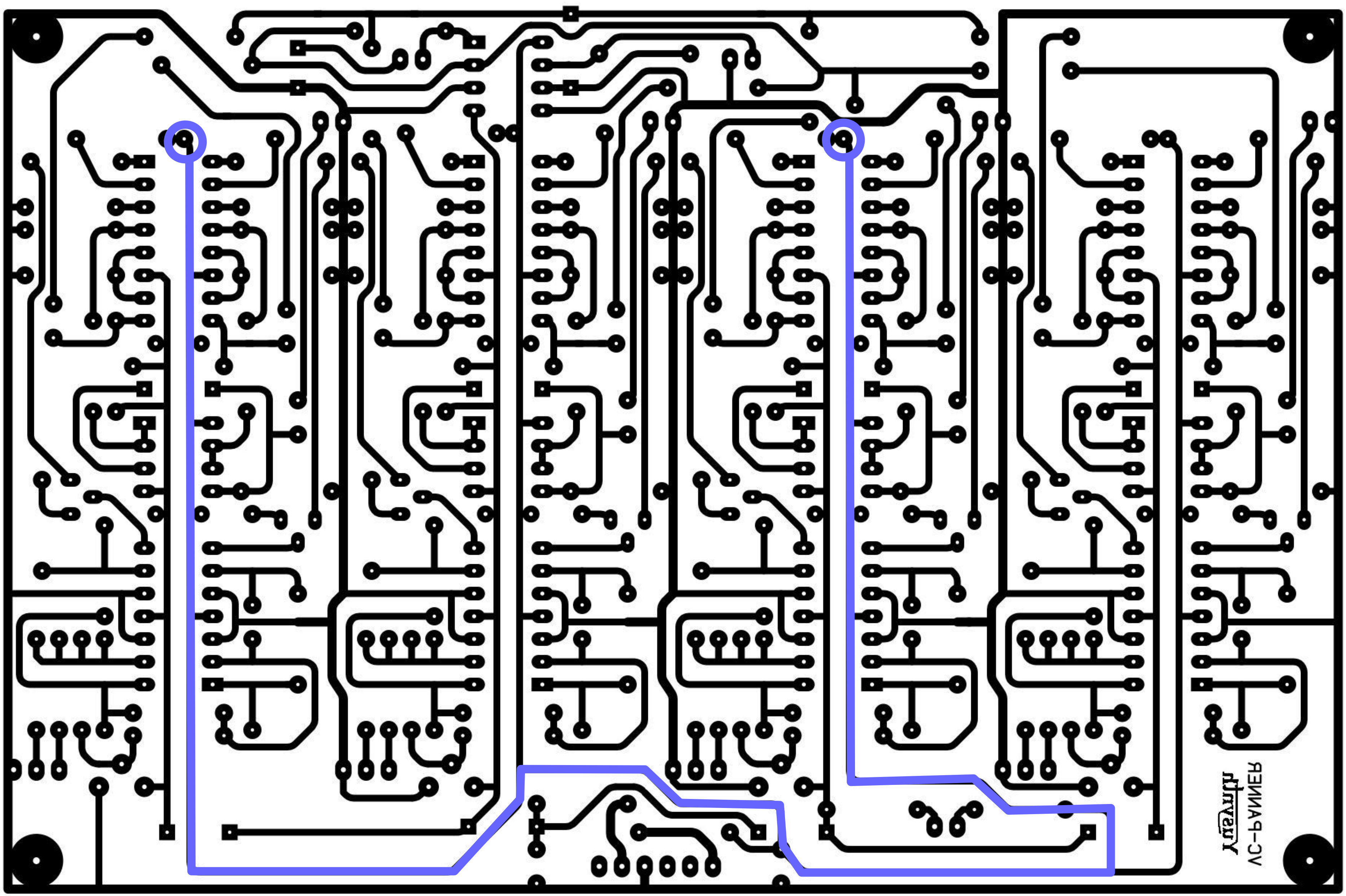
Un esempio di applicazione



Un esempio di applicazione



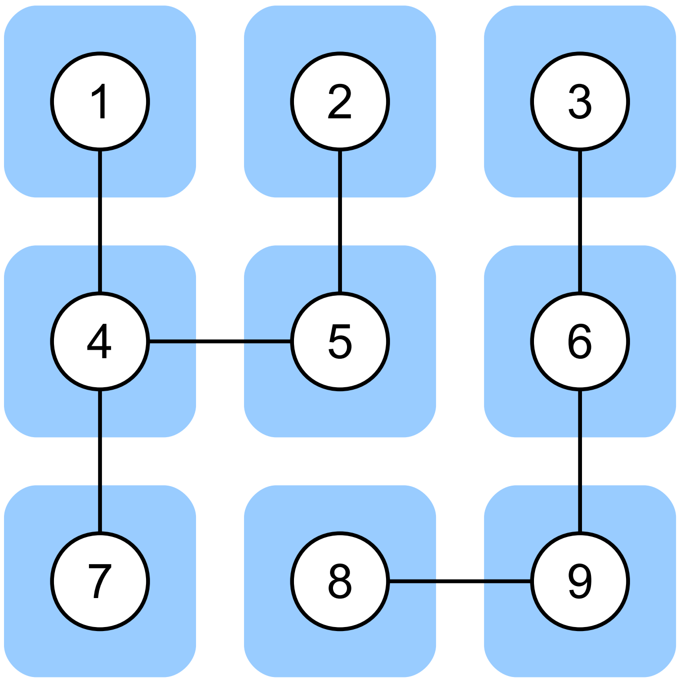
Un esempio di applicazione



Un esempio di applicazione

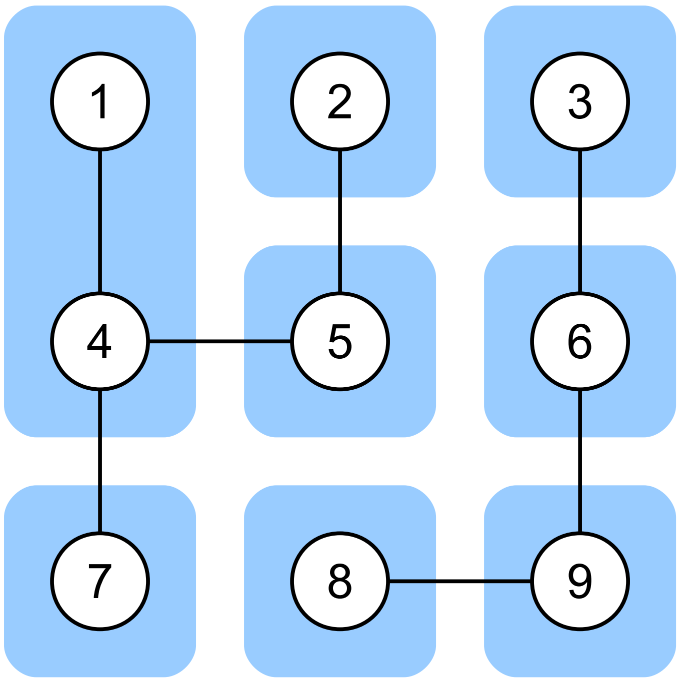
- Rappresentiamo il circuito con un insieme $V = \{1, ..., n\}$ di n nodi (pin) collegati da segmenti conduttivi
- Indichiamo con E la lista di coppie (v_1, v_2) di pin che sono tra di loro adiacenti (collegati)
- Vogliamo pre-processare il circuito in modo da rispondere in maniera efficiente a interrogazioni del tipo: "*i pin x e y sono tra loro collegati?*"

Esempio



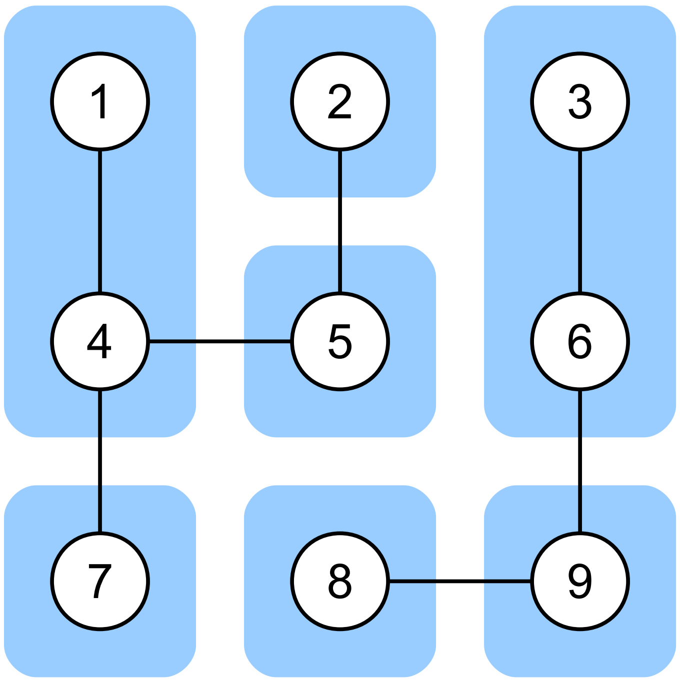
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



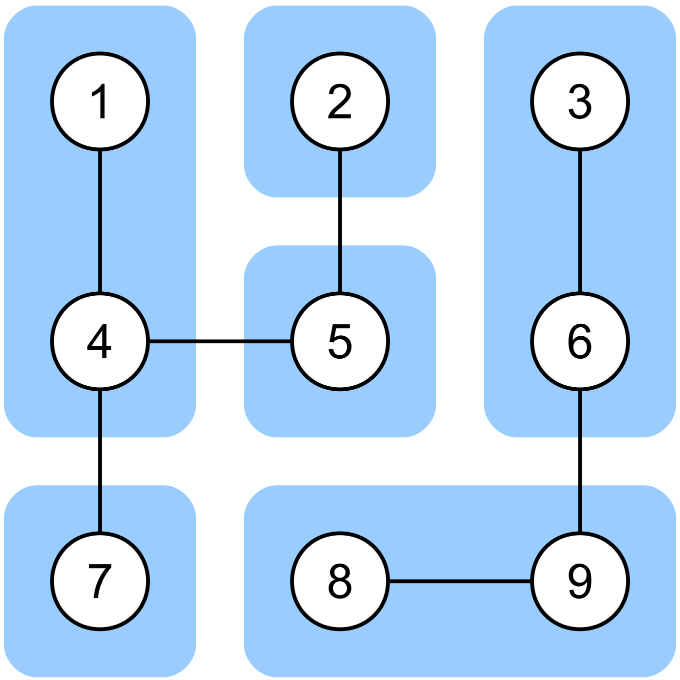
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



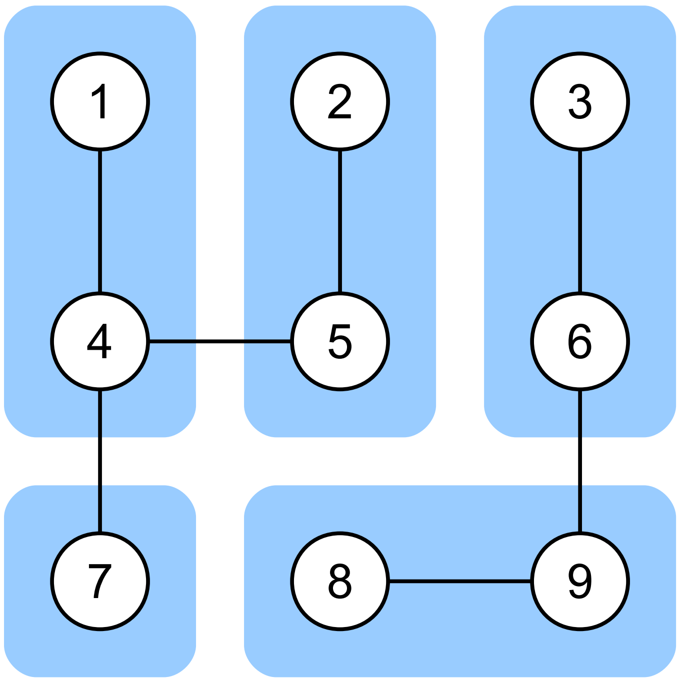
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



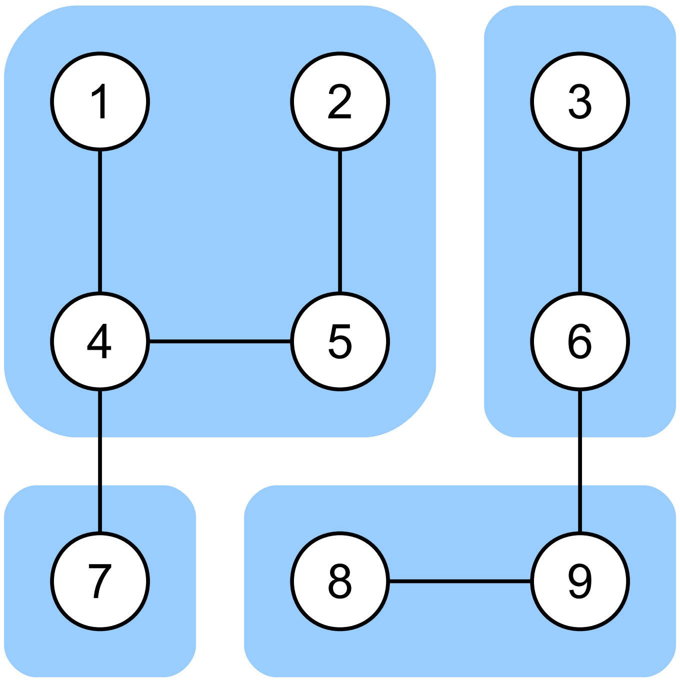
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



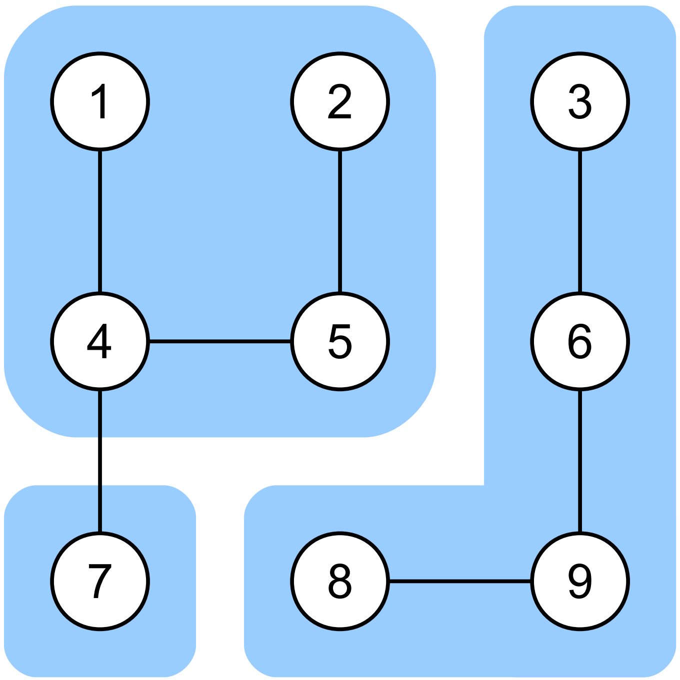
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



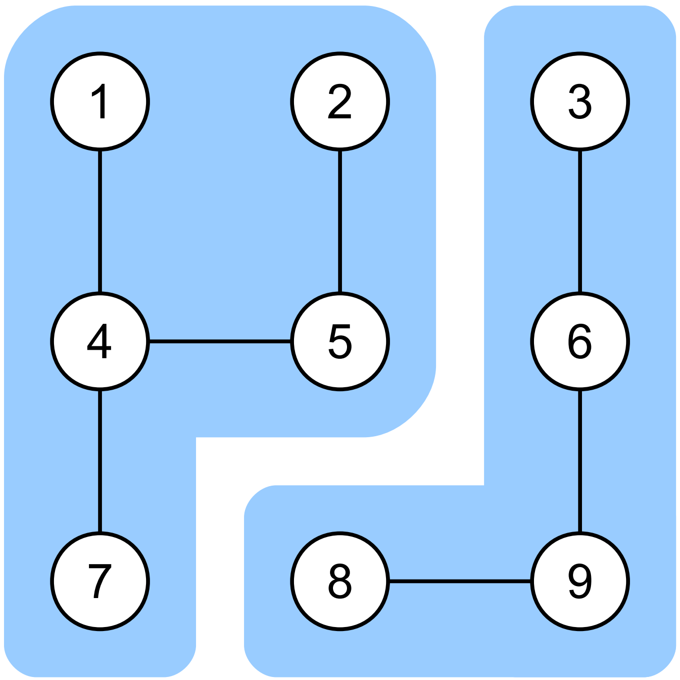
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



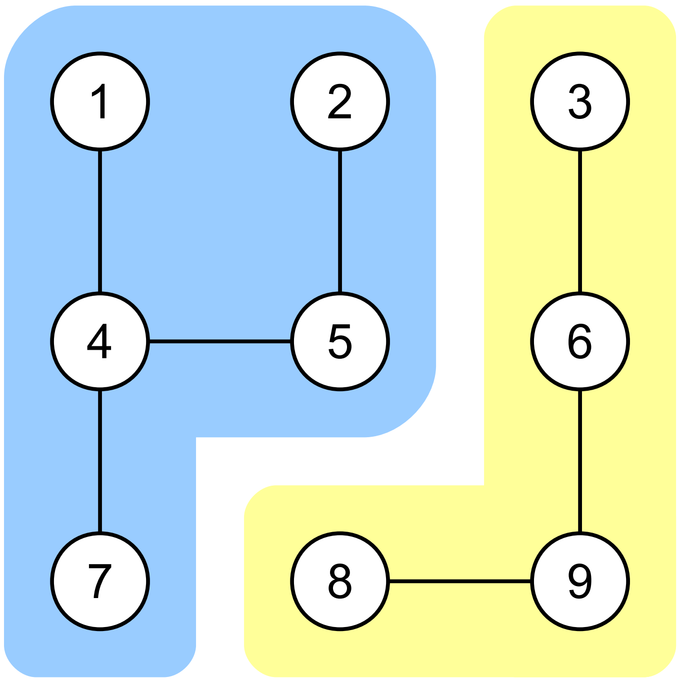
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



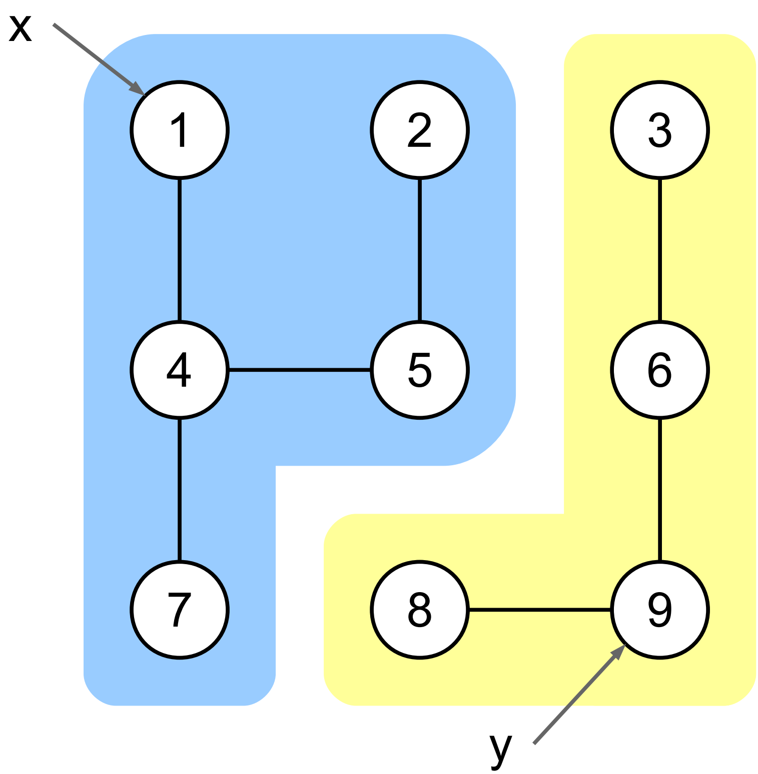
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

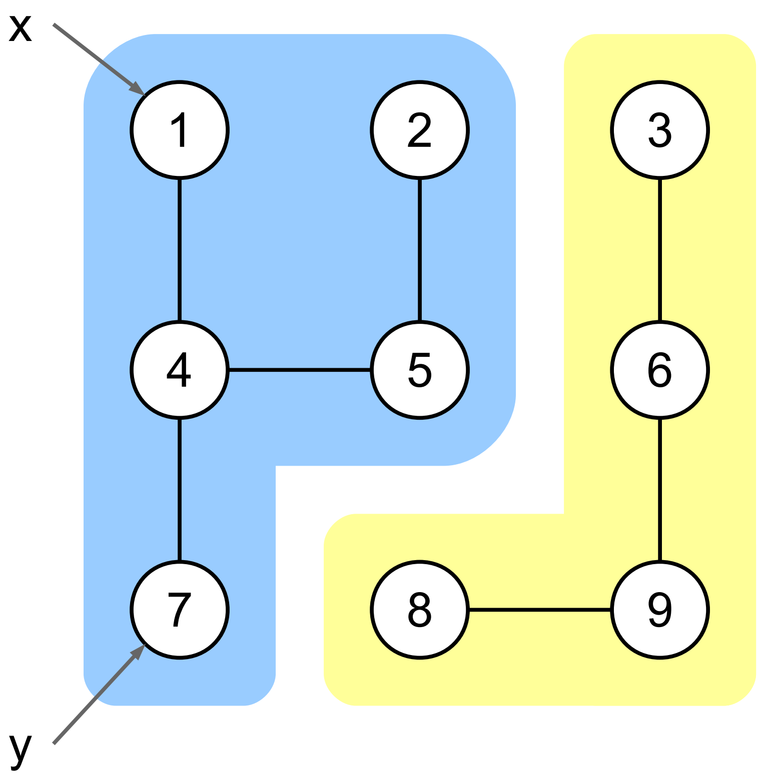
Esempio



I pin *x* e *y* non sono tra loro collegati perché

find(x) ≠ find(y)

Esempio



I pin *x* e *y* sono tra loro collegati perché

$$find(x) = find(y)$$

Esempio

Serve per trovare componenti connesse in un grafo

```
Mfset MF ← Mfset(n);
for each (v, w) ∈ E do
    MF.merge(v, w);
endfor
integer x ← ... ;
integer y ← ... ;
if ( MF.find(x) = MF.find(y) ) then
    print "x e y sono collegati";
else
    print "x e y non sono collegati";
endif
```

Crea una Struttura Union-Find per n elementi

Per ogni arco (v,w) nel grafo E, unisce i set contenenti v e w con MF. merge(v,w)

Verifica la connessione di due elementi

Implementazione di Merge-Find

- Strutture dati di base:
  - Strutture QuickFind: liste (alberi di altezza 1)
  - Strutture QuickUnion: alberi generali
- Algoritmi basati su euristiche di bilanciamento
  - QuickFind—Euristica sul peso → Tiene traccia della dimensione (numero di elementi) di ciascun albero
  - QuickUnion—Euristica sul rango → Tiene traccia di una stima dell'altezza (chiamata rank) di ciascun albero.

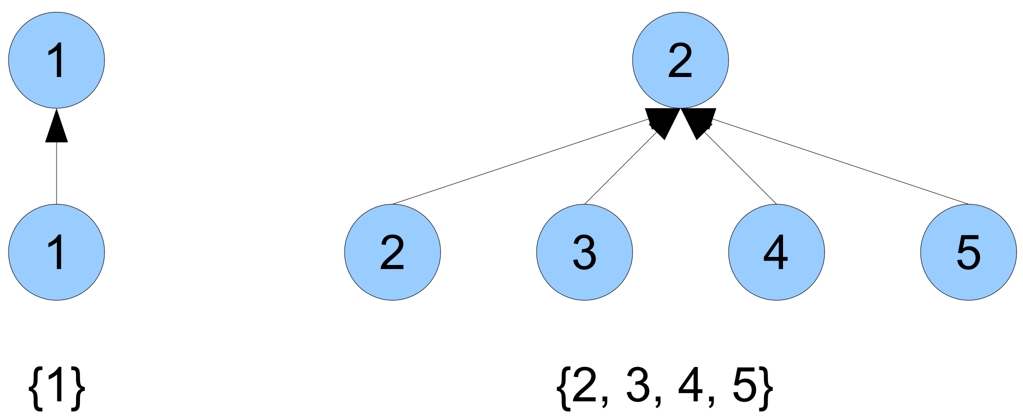
Le euristiche in questo contesto sono tecniche intelligenti che si applicano durante le operazioni union (o merge) o find per mantenere la struttura dati efficiente

↳ L'obbiettivo e' evitare che gli alberi diventino troppo sbilanciati, il che renderebbe l'operazione find lenta.

QuickFind

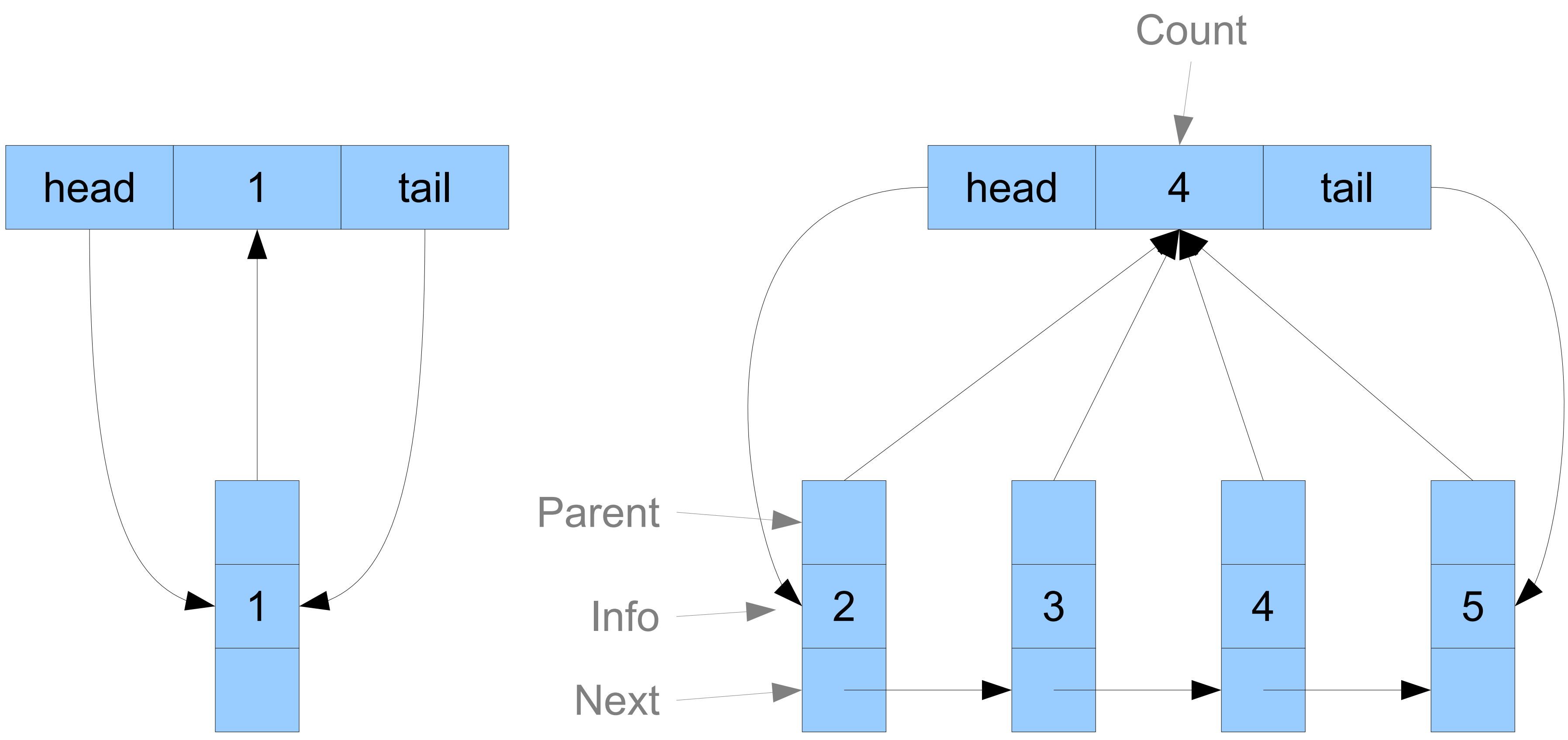
QuickFind

- Ogni insieme viene rappresentato (concettualmente) con un albero di altezza uno
  - Le foglie dell'albero contengono gli elementi dell'insieme
  - Il rappresentante è la radice

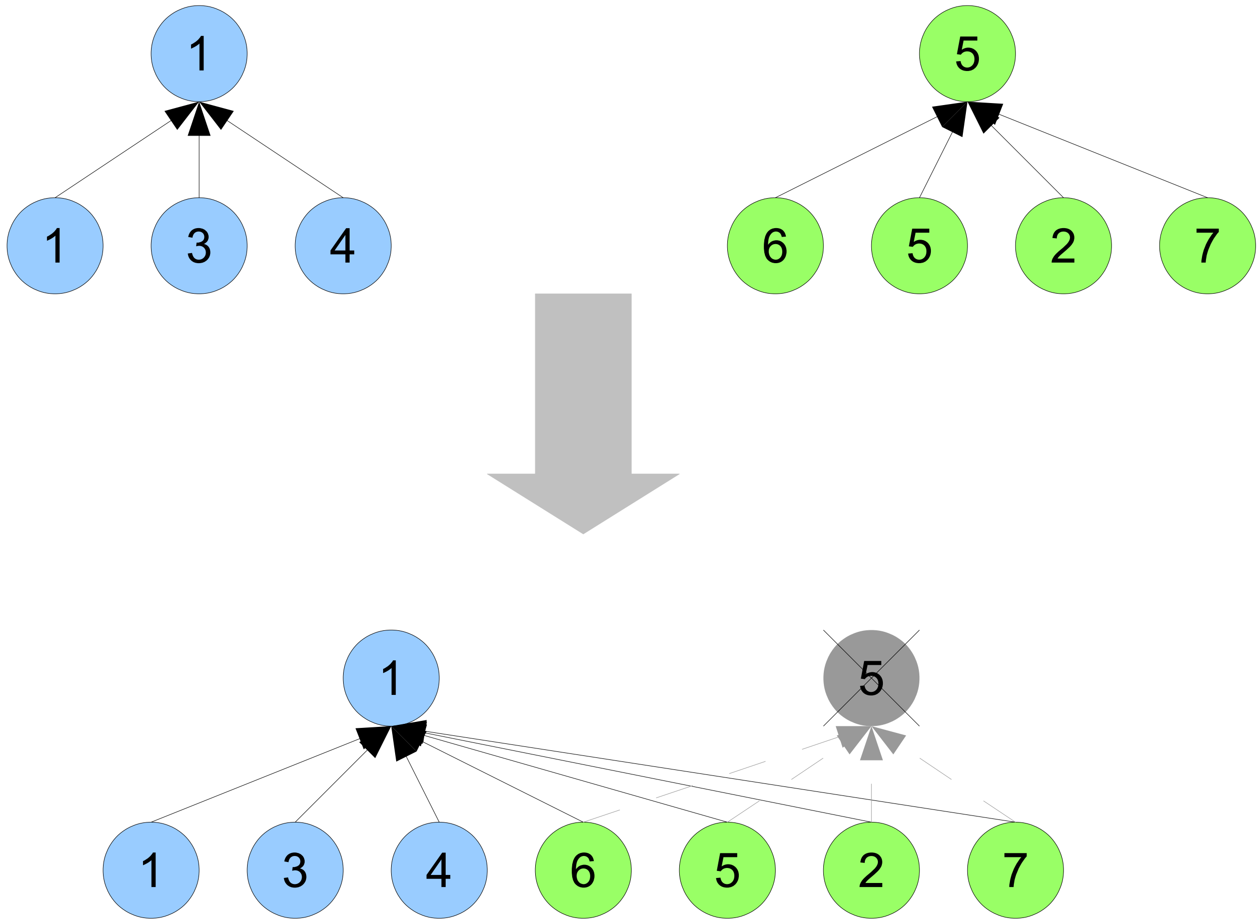


Implementazione

- Generalmente si rappresentano gli insiemi QuickFind con strutture concatenate simili a liste



QuickFind: Esempio
`merge(3,2)`



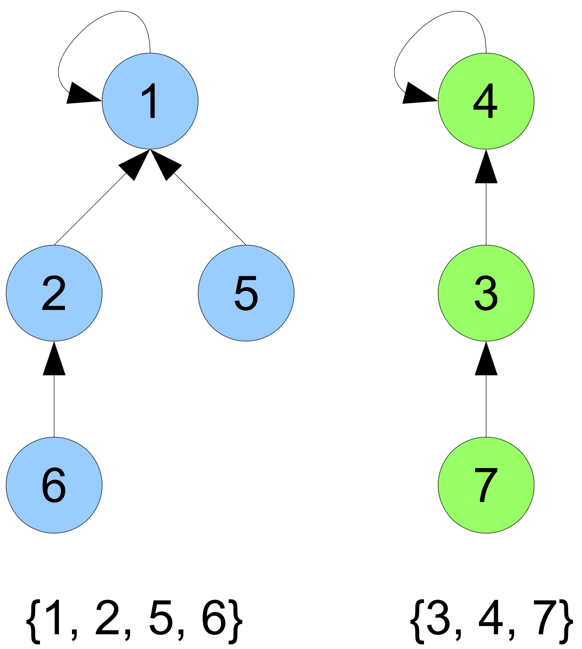
QuickFind

- **`Mfset(n)`**
  - Crea *n* liste, ciascuna contenente un singolo intero
  - Costo O(*n*)
- **`find(x)`**
  - Restituisce il riferimento al rappresentante di *x*
  - Costo O(1)
- **`merge(x,y)`**
  - Tutti gli elementi della lista contenente *y* vengono spostati nella lista contenente *x*
  - Costo nel caso pessimo O(*n*), essendo *n* il numero complessivo di elementi in entrambi gli insiemi disgiunti
    - Nel caso peggiore l'insieme contenente *y* ha *n* - 1 elementi

QuickUnion

QuickUnion

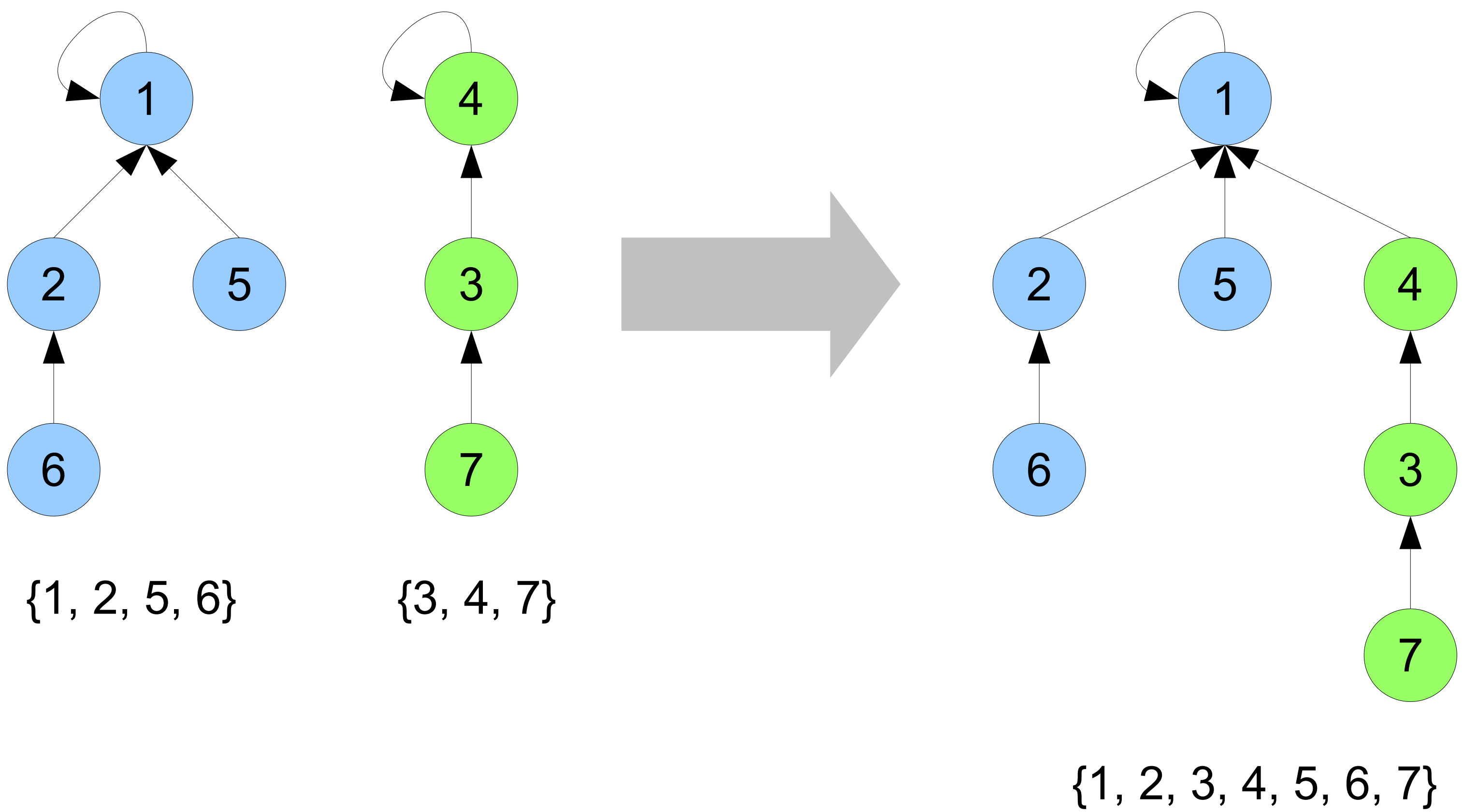
- Implementazione basata su foresta
  - Si rappresenta ogni insieme con un albero radicato generico
  - Ogni nodo contiene
    - un intero
    - un riferimento al padre (la radice è padre di se stessa)
  - Il rappresentante di un insieme è la radice dell'albero corrispondente



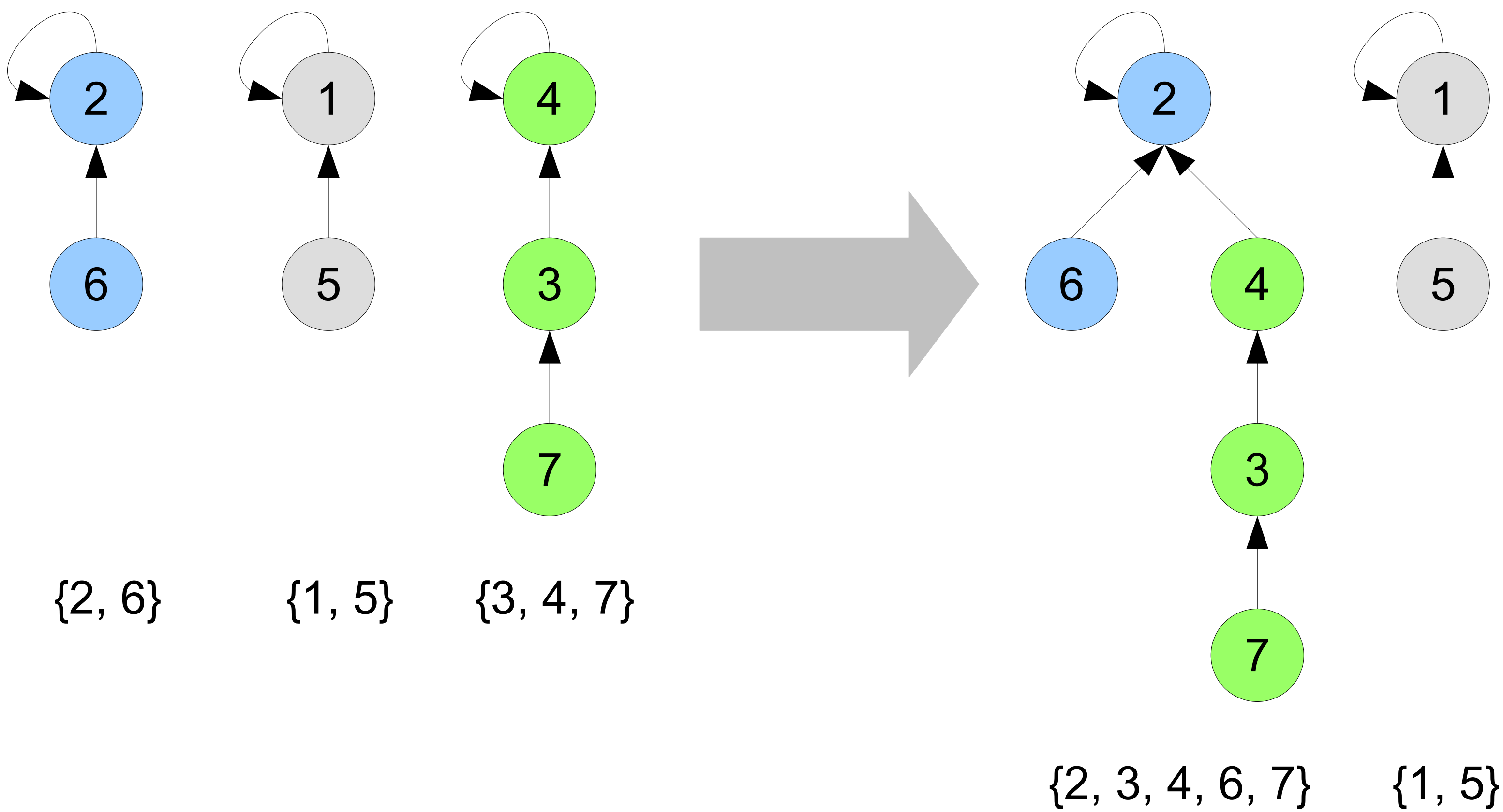
QuickUnion

- **`Mfset(n)`**
  - Crea *n* alberi, ciascuno contenente un singolo intero
  - Costo O(*n*)
- **`find(x)`**
  - Risale la lista degli antenati di *x* fino a trovare la radice e ne ritorna il contenuto come rappresentante
  - Costo O(*n*) nel caso pessimo
- **`merge(x,y)`** x diventa padre della radice di y
  - Rende la radice dell'albero che contiene *y* figlia della radice dell'albero che contiene *x*
  - Costo O(1) nel caso ottimo (*x* e *y* sono già le radici dei rispettivi alberi), O(*n*) nel caso pessimo

QuickUnion: Esempio
`merge(2,7)`



QuickUnion: Esempio
`merge(2,7)`



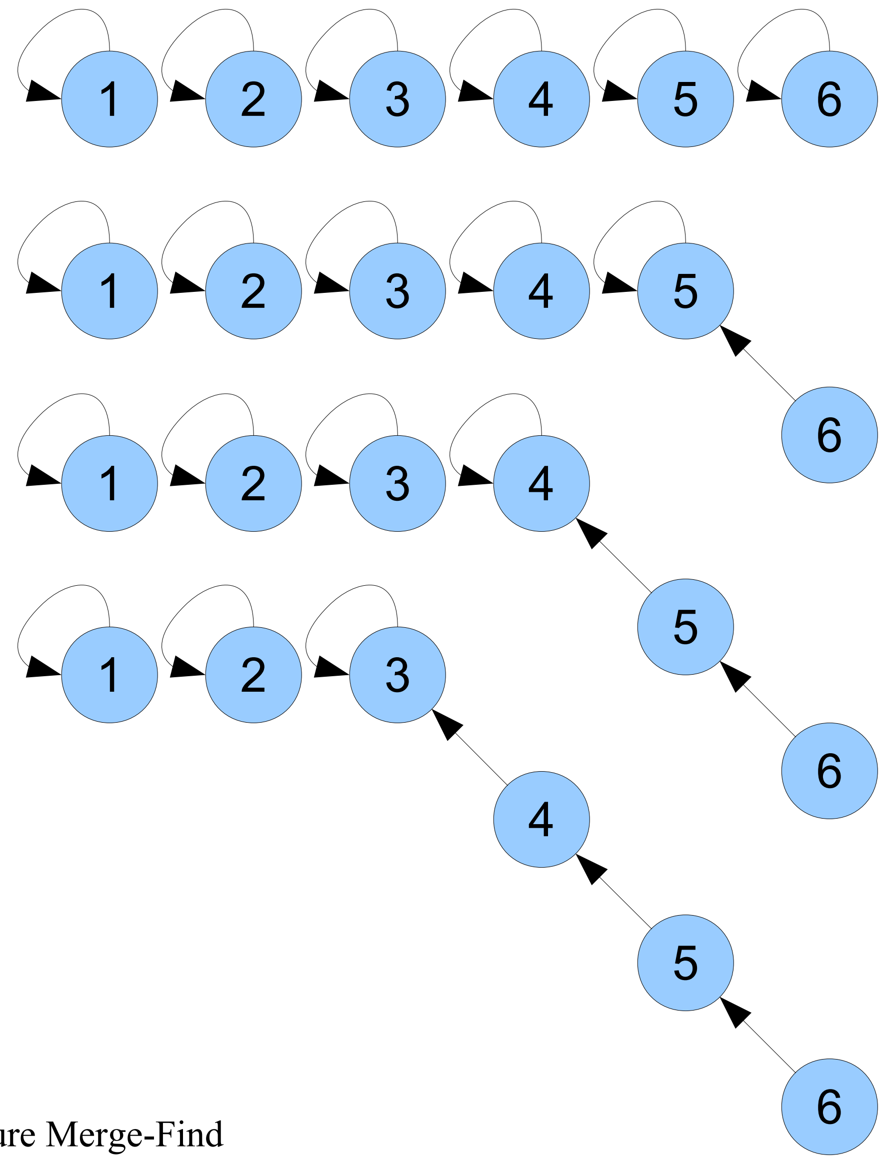
Caso pessimo per `find()`

`Mfset(6)`

`merge(5,6)`

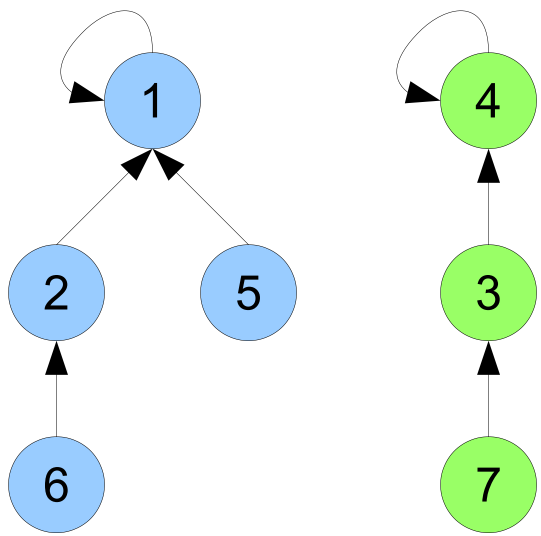
`merge(4,5)`

`merge(3,4) ...`



Nota implementativa

- Un modo molto comodo per rappresentare una foresta di alberi QuickUnion è di usare un array di interi (*vettore di padri*)



i	1	2	3	4	5	6	7
p[i]	1	1	4	4	1	2	3

Il padre del nodo `i` è il nodo `p[i]`

Implementazione Java

```java
public class QuickUnion
{
    private int[] p;
    public QuickUnion(int n) {
        p = new int[n];
        for (int i = 0; i < n; i++) {
            p[i] = i;
        }
    }
    private int find(int x) {
        while (x != p[x]) {
            x = p[x];
        }
        return x;
    }
    public void merge(int x, int y) {
        int rx = find(x);
        int ry = find(y);
        p[ry] = rx;
    }
}
```

Costo delle operazioni

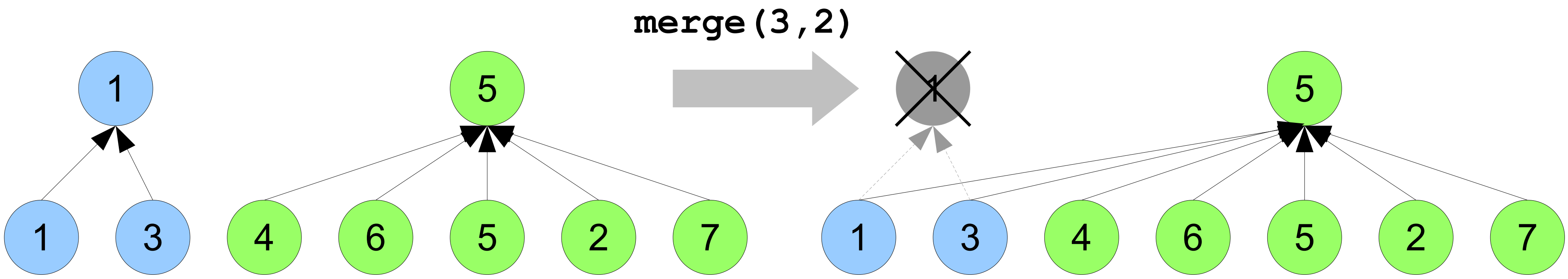
	QuickFind	QuickUnion
`Mfset(n)`	O(n)	O(n)
`merge(x,y)`	O(n)	O(1) caso ottimo O(n) caso pessimo
`find(x)`	O(1)	O(1) caso ottimo O(n) caso pessimo

- Quando usare QuickFind?
  - Quando le **`merge()`** sono rare e le **`find()`** frequenti
- Quando usare QuickUnion?
  - Quando le **`find()`** sono rare e le **`merge()`** frequenti
- Esistono euristiche che permettono di migliorare questi risultati

Ottimizzazioni

QuickFind: Euristica sul peso

- Una strategia per diminuire il costo dell'operazione **`merge()`** in QuickFind consiste nel:
  - memorizzare nel nodo rappresentante il numero di elementi dell'insieme; la dimensione corretta può essere mantenuta in tempo O(1)
  - appendere l'insieme con meno elementi a quello con più elementi

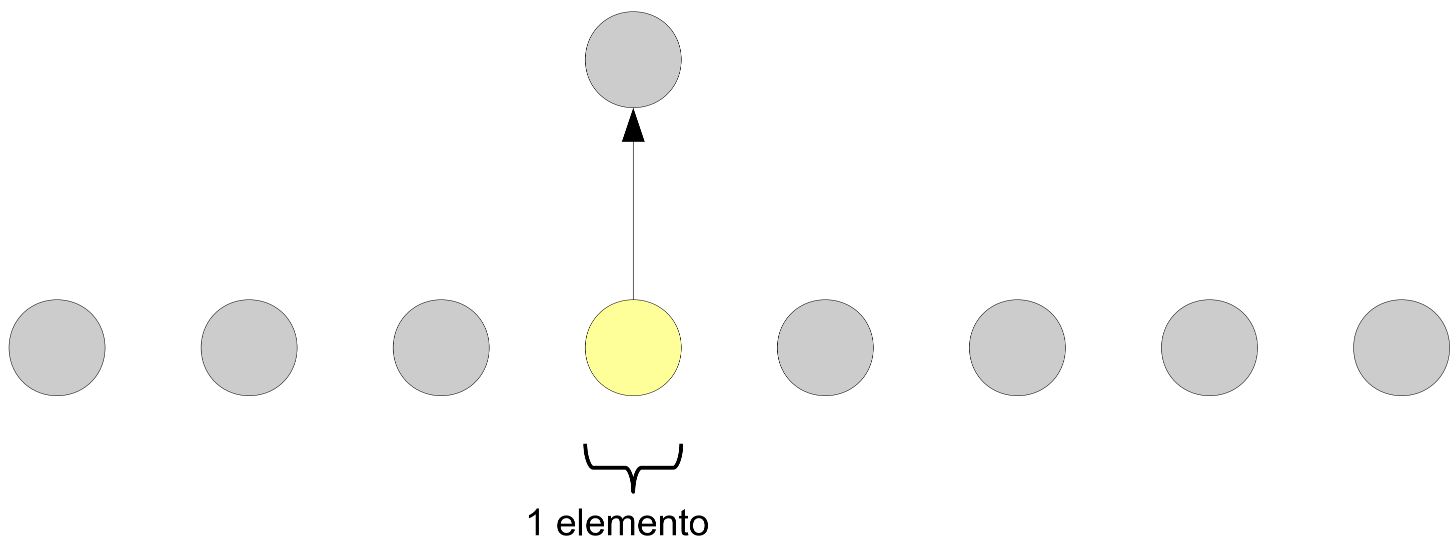


QuickFind: Euristica sul peso

- **`Mfset(n)`**
  - Costo O(n)
- **`find(x)`**
  - Costo O(1)
- **`merge(x,y)`**
  - Tutti i nodi della lista con meno nodi vengono spostati nella lista con più nodi
  - Dimostriamo che il costo complessivo di una sequenza di (n - 1) merge è O(n log n)
    - Quindi il costo ammortizzato per una singola operazione è O(log n)

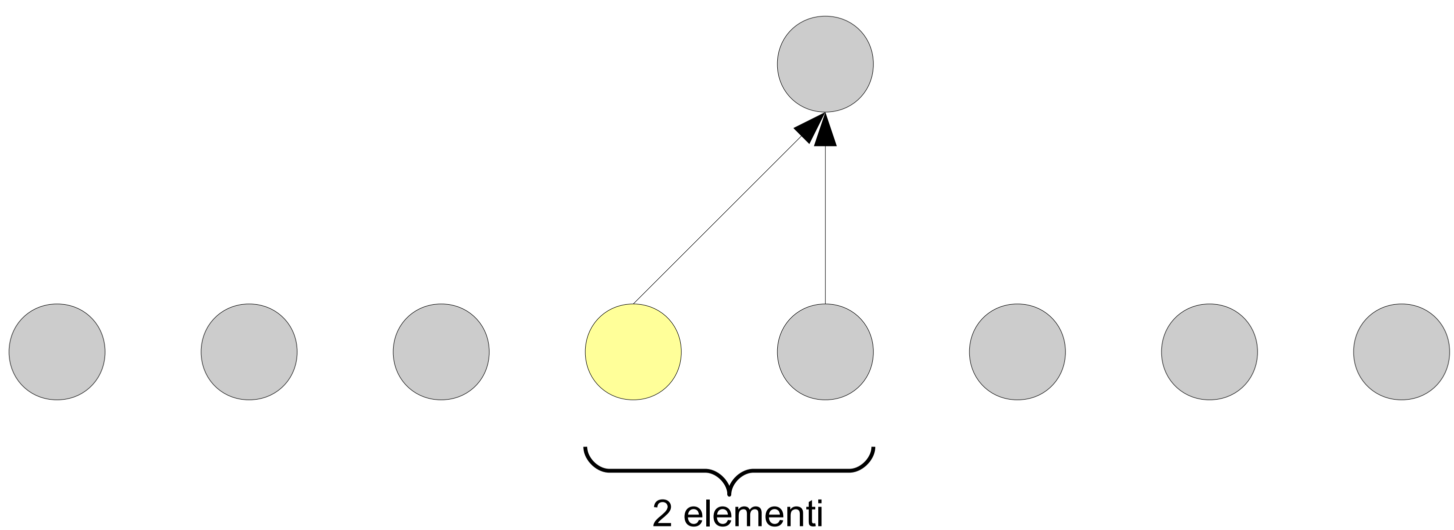
Intuizione

- Ogni volta che una foglia di un albero QuickFind cambia padre, entra a far parte di un insieme che ha almeno il doppio di elementi di quello cui apparteneva



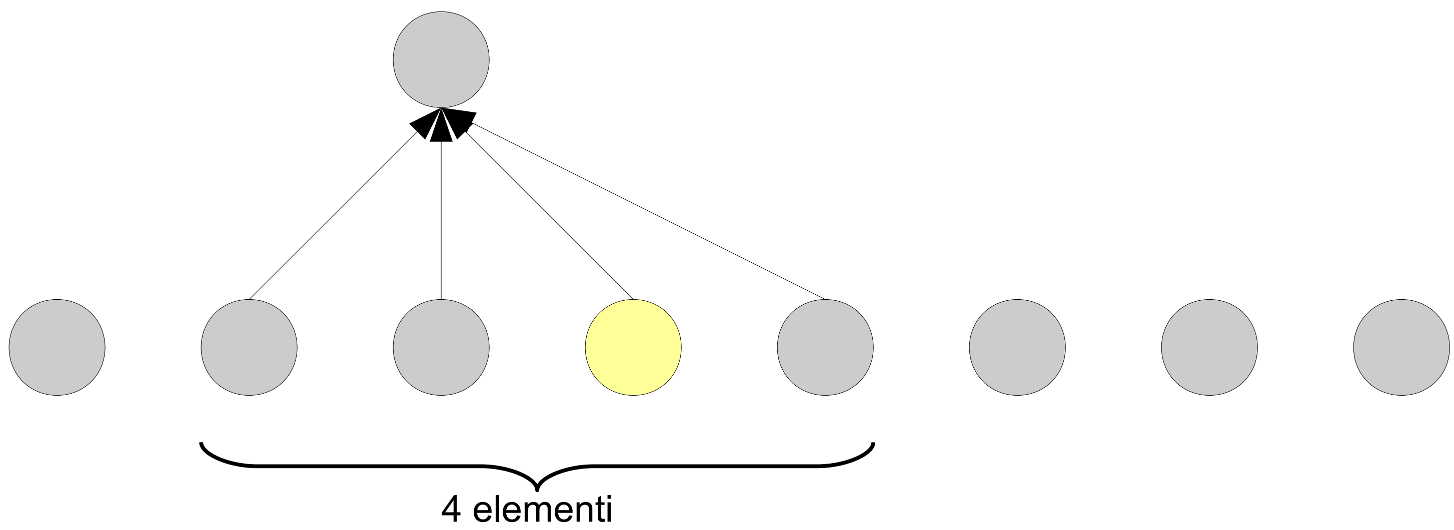
Intuizione

- Ogni volta che una foglia di un albero QuickFind cambia padre, entra a far parte di un insieme che ha almeno il doppio di elementi di quello cui apparteneva



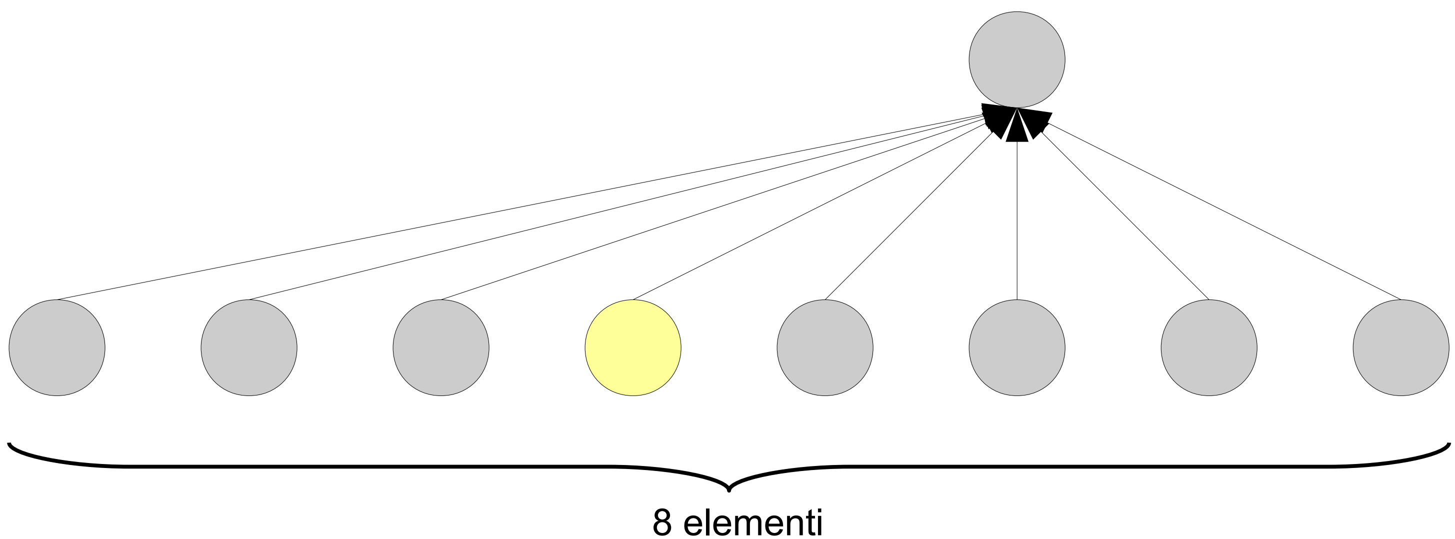
Intuizione

- Ogni volta che una foglia di un albero QuickFind cambia padre, entra a far parte di un insieme che ha almeno il doppio di elementi di quello cui apparteneva



Intuizione

- Ogni volta che una foglia di un albero QuickFind cambia padre, entra a far parte di un insieme che ha almeno il doppio di elementi di quello cui apparteneva



Dimostrazione

- Ogni volta che una foglia di un albero QuickFind cambia padre, entra a far parte di un insieme che ha almeno il doppio di elementi di quello cui apparteneva
  - **`merge(A, B)`** con size(A) ≥ size(B)
    - Le foglie di B cambiano padre
    - size(A) + size(B) ≥ size(B) + size(B) = 2 × size(B)
  - **`merge(A, B)`** con size(A) ≤ size(B)
    - Le foglie di A cambiano padre
    - size(A) + size(B) ≥ size(A) + size(A) = 2 × size(A)

Chi ha la size minore cambia padre

Costo delle operazioni

- Le strutture QuickFind supportano una operazione `Mfset(n)`, m operazioni `find()`, ed al più $(n - 1)$ `merge()` in tempo totale $O(m + n \log n)$. L'occupazione di memoria è $O(n)$
- Dimostrazione
  - `Mfset(n)` ed m `find()` costano $O(n + m)$
  - Le $(n - 1)$ `merge()` richiedono complessivamente $O(n \log n)$

Costo delle operazioni `merge`

- Il costo di una singola **`merge()`** è dato dal numero di nodi che cambiano padre
- Ogni volta che un elemento cambia padre, entra a far parte di un insieme con almeno il doppio di elementi di quello cui faceva parte
  - Dopo k **`merge()`** fa parte di un insieme con almeno 2^k elementi
  - Quindi ci possono essere $O(\log_2 n)$ cambi di paternità per ciascun elemento
- Il costo totale dell'intera sequenza di **`merge()`** è quindi $O(n \log n)$

Domanda

- Si consideri una struttura QuickFind con euristica sul peso composta inizialmente da 8 insiemi {1}, {2}, ..., {8}. Descrivere una sequenza di **`merge()`** che alla fine producano un singolo albero, e tali che il costo totale di tutte le **`merge()`** sia il *minimo* possibile.
- Ripetere l'esercizio identificando una sequenza di operazioni **`merge()`** il cui costo totale sia *massimo* possibile.

1)

Per minimizzare il costo, dobbiamo evitare di aggiornare nodi più volte. L'idea è fondere sempre alberi di dimensioni simili, in modo che gli alberi più piccoli vanno aggiornati solo una volta e poi non vengano più toccati.

Strategia: Usare un approccio "binario": fondere sempre alberi delle stesse dimensioni (o il più simili possibili)

Sequenza ottima:

{1}	{2}	cost 1
{3}	{4}	cost 1
5	6	1
7	8	1
1,2	3,4	2
5,6	7,8	2
1,2,3,4	5,6,7,8	4

Costo minimo Totale $1+1+1+1+2+2+4=12$

2)

Per massimizzare il costo, dobbiamo fare in modo che gli stessi nodi vengano aggiornati più volte. L'idea è fondere sempre l'albero più piccolo con quello più grande, in modo che i nodi dell'albero piccolo vengano aggiornati molte volte.

Strategia: aggiungere un elemento alla volta all'albero più grande

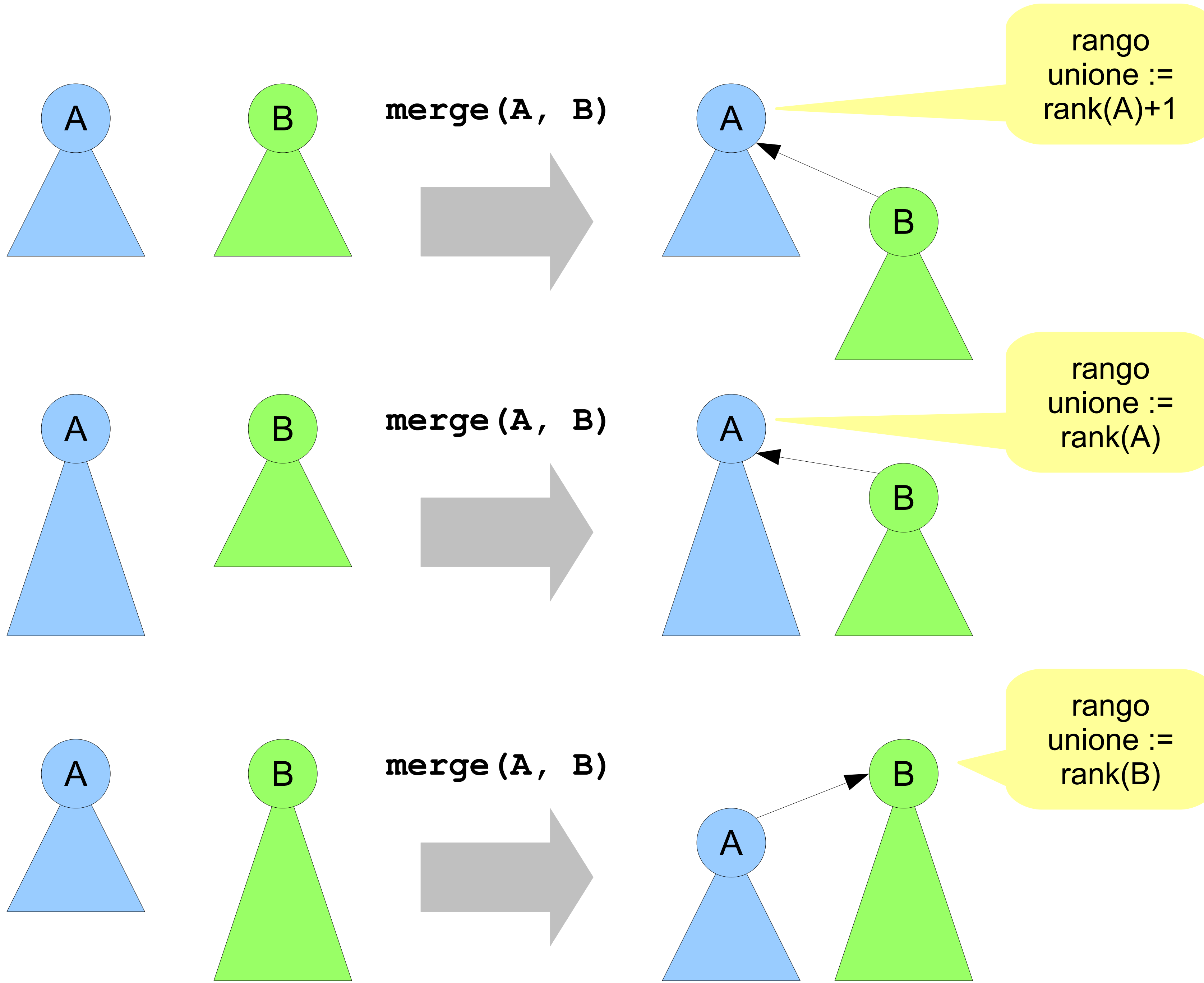
Sequenza peggiore

		costo
1	2	1
1,2	3	2
1,2,3	4	3 (aggiorno i 3 nodi di {1,2,3})
1,2,3,4	5	4
1,2,3,4,5	6	5
1,2,3,4,5,6	7	6
1,2,3,4,5,6,7	8	7

Costo Totale $1+2+3+4+5+6+7=28$

QuickUnion
Euristica “union by rank”

- Il problema degli alberi QuickUnion è che possono diventare troppo alti
  - quindi rendere inefficienti le operazioni **`find()`**
- Idea:
  - Rendiamo la radice dell'albero più basso figlia della radice dell'albero più alto
- Ogni radice mantiene informazioni sul proprio rango
  - il rango *rank*(*x*) di un nodo *x* è il numero di archi del cammino più lungo fra *x* e una foglia sua discendente
  - cioè l'altezza dell'albero radicato in *x*


A
B
merge(A, B)
A
B
rango
unione :=
rank(A)+1
A
B
merge(A, B)
A
B
rango
unione :=
rank(A)
A
B
merge(A, B)
A
B
rango
unione :=
rank(B)

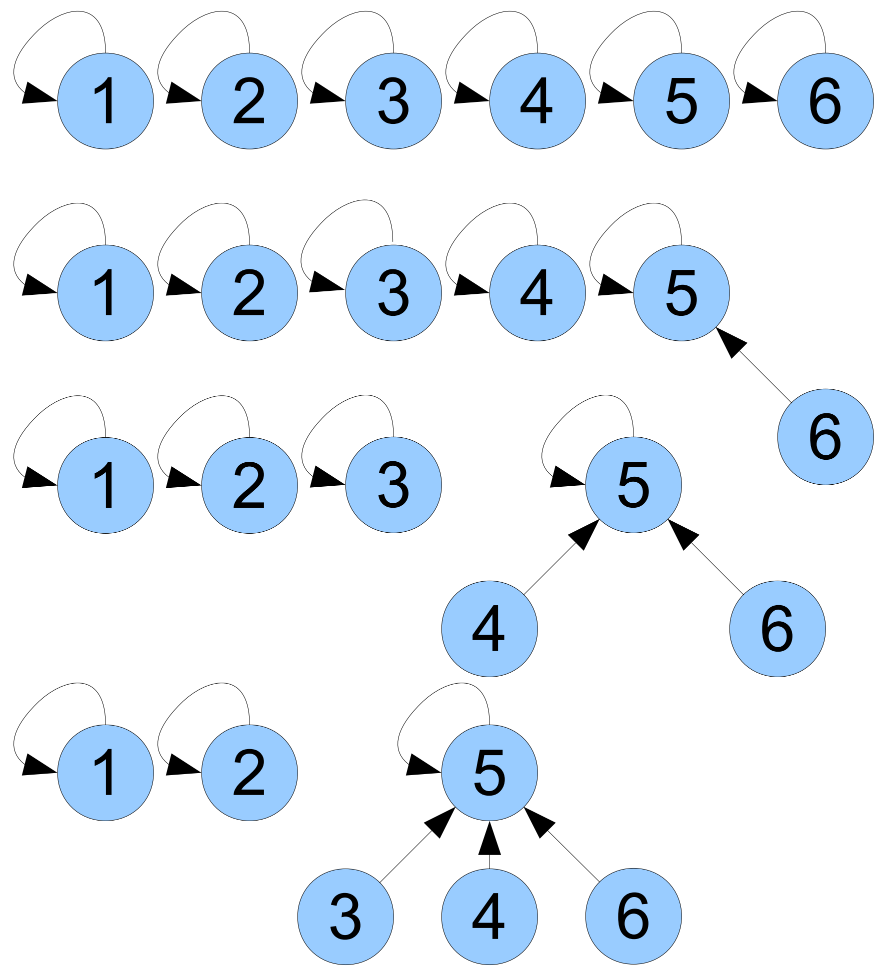
Esempio

`Mfset(6)`

`merge(5,6)`

`merge(4,5)`

`merge(3,5)`

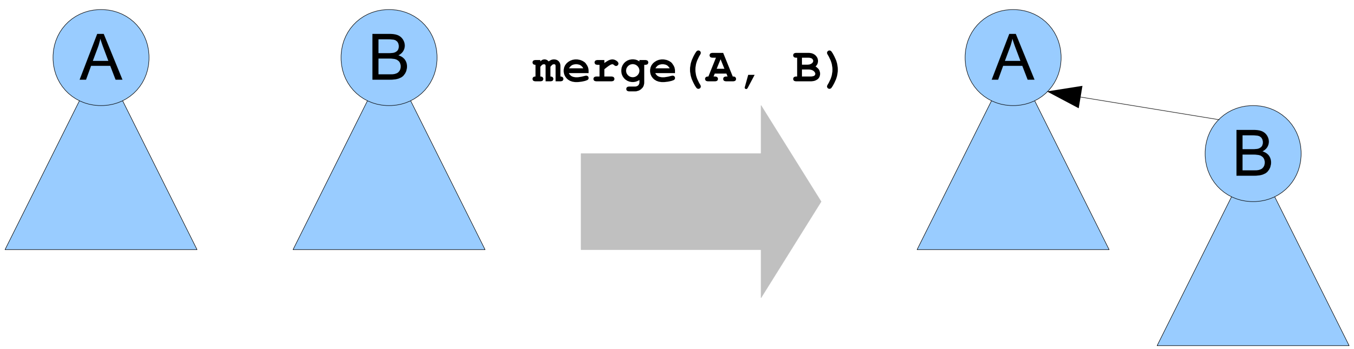


Proprietà union by rank

Dimostrazione x induzione

- Un albero QuickUnion con euristica "union by rank" avente il nodo x come radice ha $\geq 2^{\text{rank}(x)}$ nodi
- Dimostrazione: induzione sul numero di **`merge()`** effettuate
  - Base (0 operazioni merge): tutti gli alberi hanno rango zero (singolo nodo) quindi hanno esattamente $2^0 = 1$ nodi
  - Induzione: consideriamo cosa succede prima e dopo una operazione **`merge(A,B)`**
    - *A* U *B* denota l'insieme ottenuto dopo l'unione
    - rank(*A* U *B*) è l'altezza dell'albero che denota *A* U *B*
    - |*A* U *B*| è il numero di nodi dell'albero *A* U *B*, e risulta |*A* U *B*| = |*A*| + |*B*| perché stiamo unendo sempre insiemi <u>disgiunti</u>

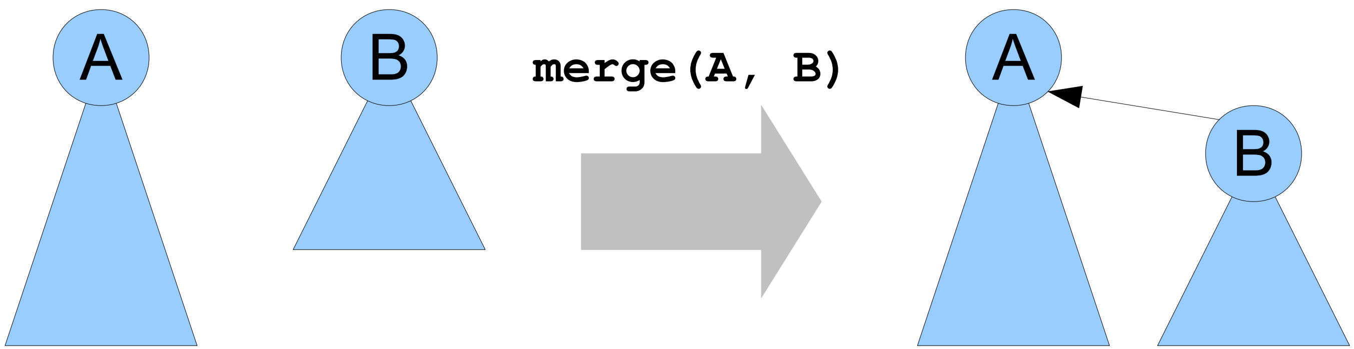
Passo induttivo
caso rank(A) = rank(B)



- $|A \cup B| = |A| + |B|$
- rank(A U B) = rank(A) +1
- Per ipotesi induttiva, $|A| \geq 2^{\text{rank}(A)}$, $|B| \geq 2^{\text{rank}(B)}$
- Quindi

$$|A \cup B| = |A| + |B|$$
$$\geq 2^{rank(A)} + 2^{rank(B)} = 2 \times 2^{rank(A)} = 2^{rank(A)+1} = 2^{rank(A \cup B)}$$

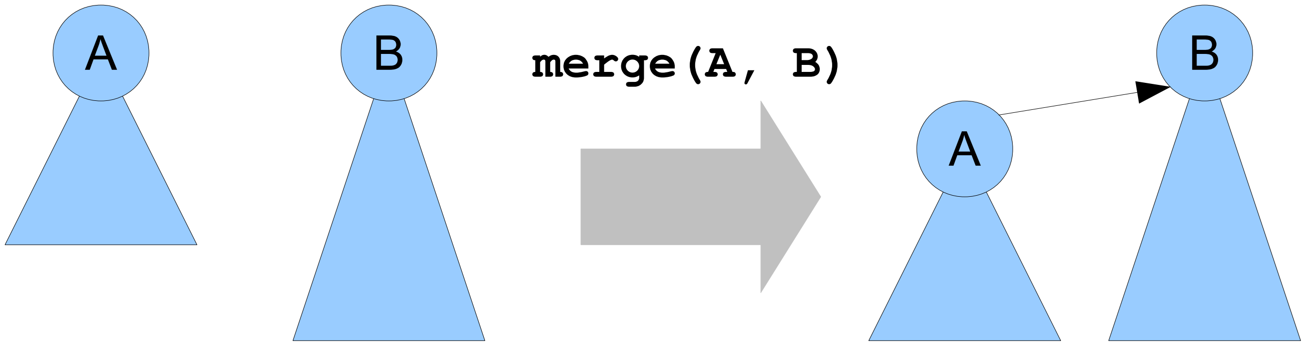
Passo induttivo
caso rank(A) > rank(B)



- $|A \cup B| = |A| + |B|$
- rank($A \cup B$) = rank(A)
  - perché l'altezza dell'albero $A \cup B$ è uguale all'altezza dell'albero A
- Per ipotesi induttiva, $|A| \geq 2^{\text{rank}(A)}$, $|B| \geq 2^{\text{rank}(B)}$
- Quindi

$$|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} > 2^{rank(A)} = 2^{rank(A \cup B)}$$

Passo induttivo
caso rank(A) < rank(B)



- $|A \cup B| = |A| + |B|$
- rank($A \cup B$) = rank(B)
  - perché l'altezza dell'albero $A \cup B$ è uguale all'altezza dell'albero B
- Per ipotesi induttiva, $|A| \geq 2^{\text{rank}(A)}$, $|B| \geq 2^{\text{rank}(B)}$
- Quindi

$$|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} > 2^{rank(B)} = 2^{rank(A \cup B)}$$

Teorema

- Durante una sequenza di operazioni **`merge()`** e **`find()`**, l'altezza di un albero QuickUnion by rank è ≤ ($\log_2 n$), essendo n il parametro della **`Mfset(n)`** iniziale
- Dimostrazione
  - L'altezza di un albero QuickUnion A è rank(A)
  - Da quanto appena visto, $2^{\text{rank(A)}} \leq n$
  - Quindi altezza = rank(A) ≤ ($\log_2 n$)
- Quindi:
  - **`Mfset(n)`** ha costo O(n)
  - **`merge()`** ha costo O(1) nel caso ottimo
    - O(log n) nel caso pessimo (vedi punto seguente)
  - **`find()`** ha costo O(log n) nel caso pessimo

Riepilogo

	QuickFind	QuickUnion	QuickFind eur. peso	QuickUnion by eur. rank
`Mfset(n)`	O(n)	O(n)	O(n)	O(n)
`merge`	O(n)	O(1) ottimo O(n) pessimo	O(log n) ammortizzato	O(1) ottimo O(log n) pessimo
`find`	O(1)	O(n)	O(1)	O(log n)