Union-Find

Gianluigi Zavattaro
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna
gianluigi.zavattaro@unibo.it

Slide realizzate a partire da materiale fornito dal Prof. Moreno Marzolla

Struttura dati per insiemi disgiunti

- Motivazioni
  - In alcune applicazioni siamo interessati a gestire insiemi disgiunti di oggetti
- Operazioni fondamentali:
  - Creare un insieme a partire da un singolo elemento
  - Unire due insiemi
  - Identificare l'insieme a cui appartiene un elemento
- Struttura dati
  - Una collezione $S = \{ S_1, S_2, ..., S_k \}$ di insiemi dinamici disgiunti
  - Gli insiemi contengono complessivamente $n \geq k$ elementi
  - Ogni insieme è identificato da un rappresentante univoco

Scelta del rappresentante

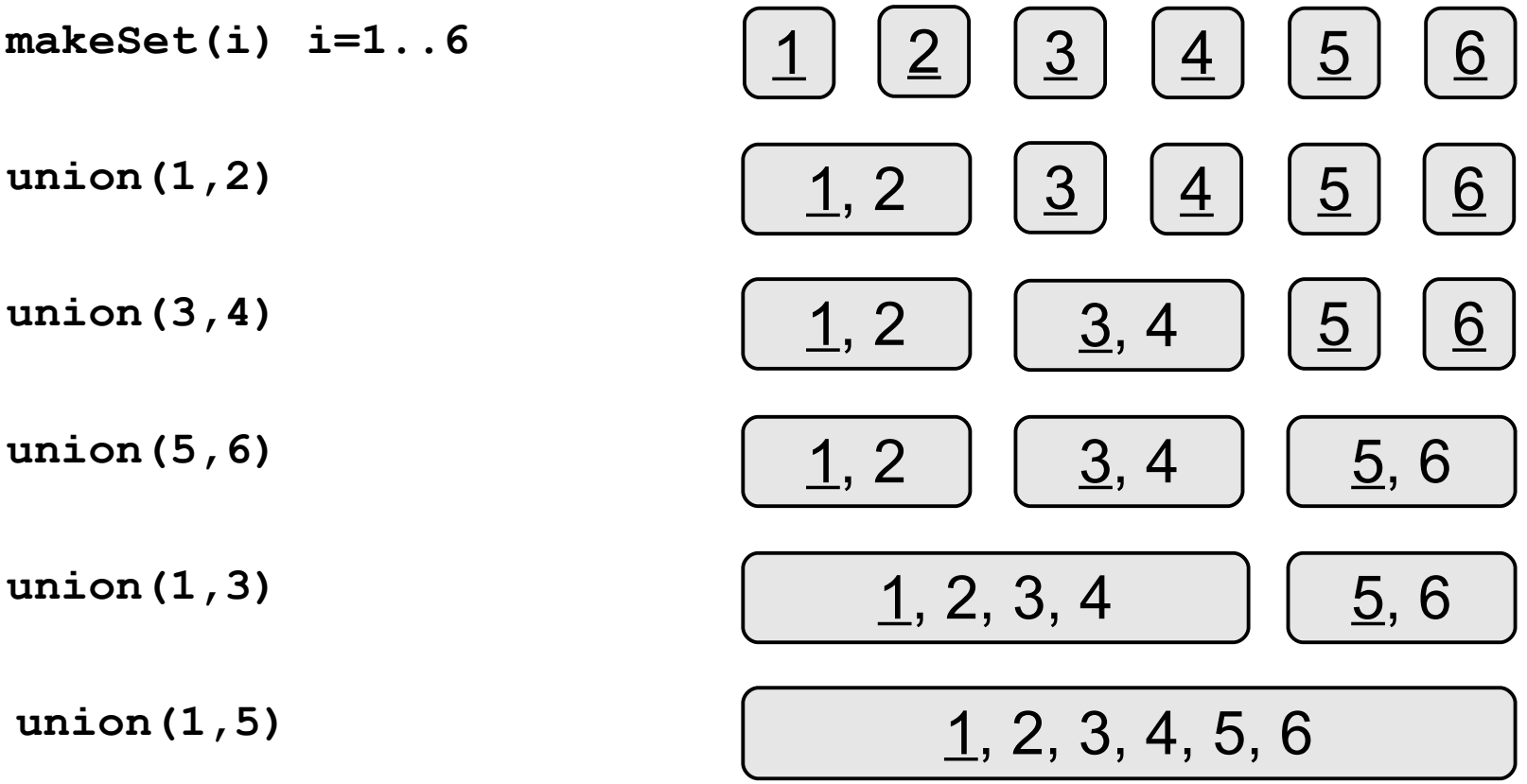
- Il rappresentante di S_i può essere un qualsiasi membro dell'insieme S_i
  - Operazioni di ricerca del rappresentante su uno stesso insieme devono restituire sempre lo stesso oggetto
  - Solo in caso di unione con altro insieme il rappresentante può cambiare

Operazioni su strutture Union-Find

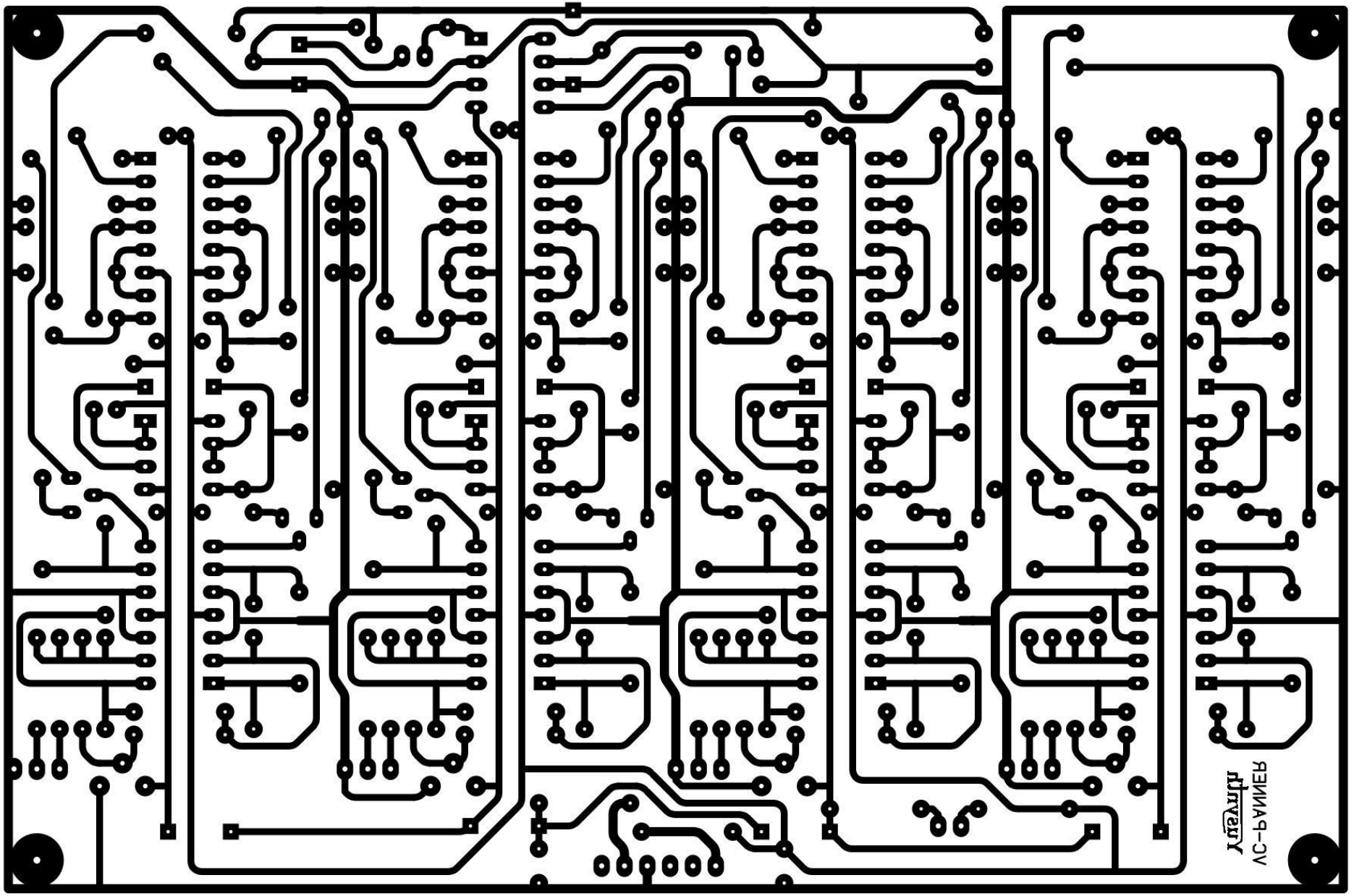
- `makeSet(elem x)`
  - Crea un insieme il cui unico elemento (e rappresentante) è *x*
  - *x* non deve appartenere ad un altro insieme esistente
- `find(elem x)` → `name`
  - Restituisce il rappresentante dell'unico insieme contenente *x*
- `union(name x, name y)`
  - Unisce i due insiemi rappresentati da *x* e da *y*
  - Assumiamo che il nome del nuovo insieme sia *x* (assunzione non strettamente necessaria: basta che venga scelto un rappresentante univoco per il nuovo insieme)
  - I vecchi insiemi devono essere distrutti

Esempio

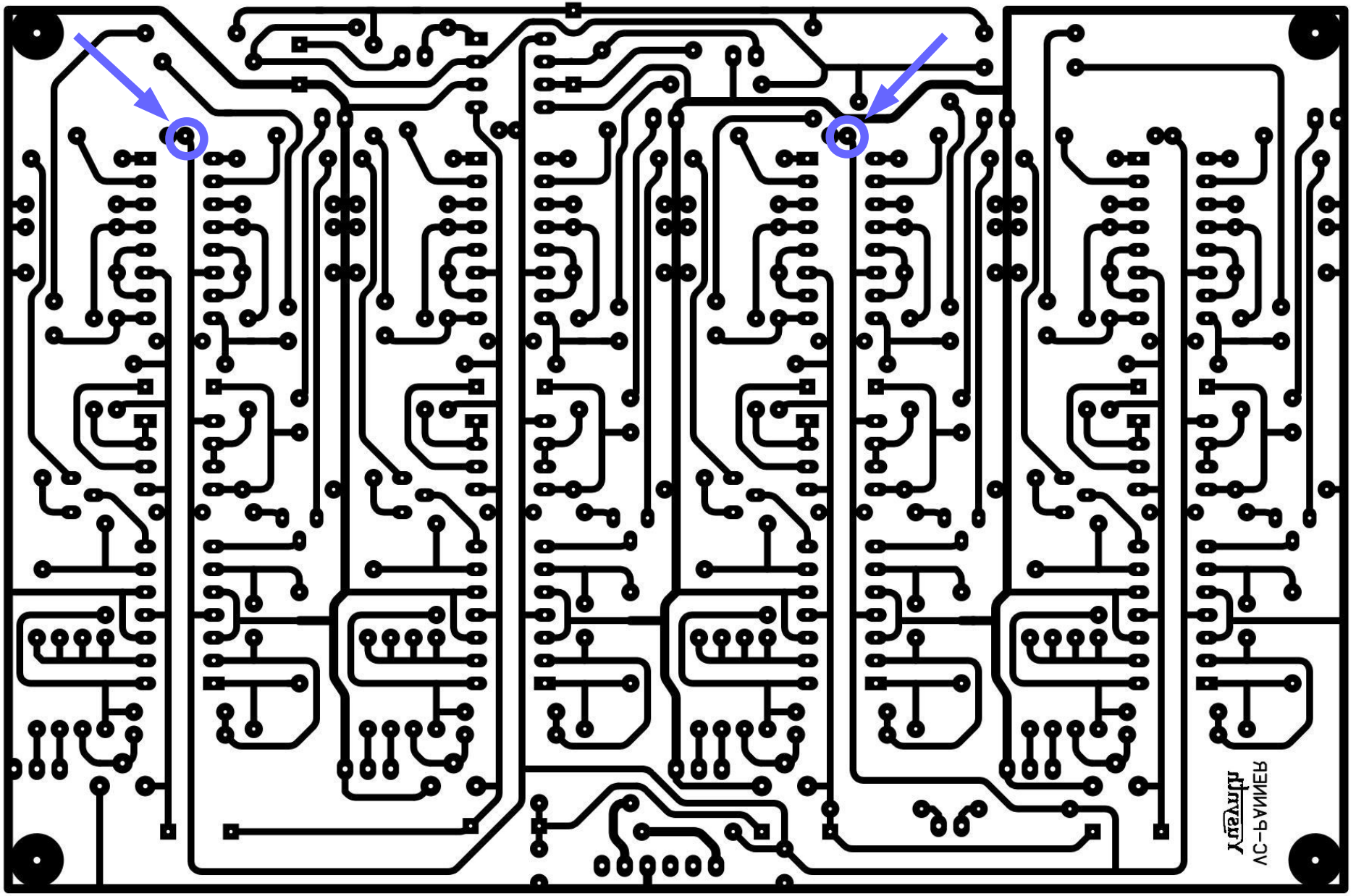
(i valori sottolineati indicano il rappresentante)



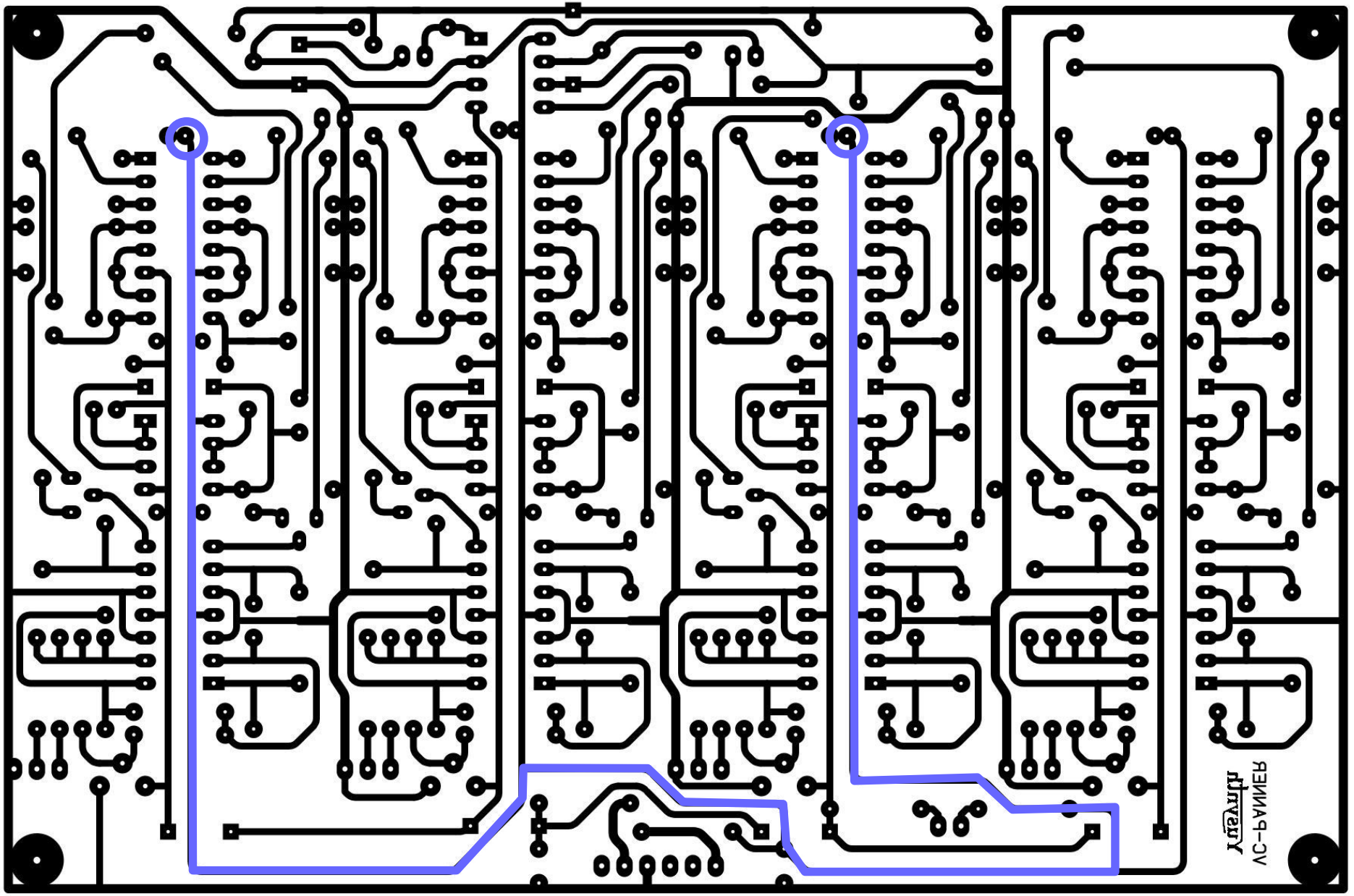
Un esempio di applicazione



Un esempio di applicazione



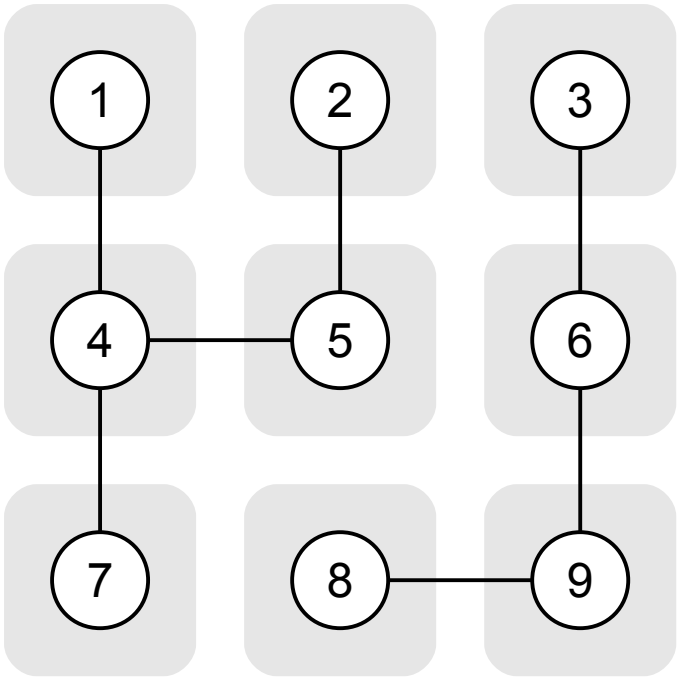
Un esempio di applicazione



Un esempio di applicazione

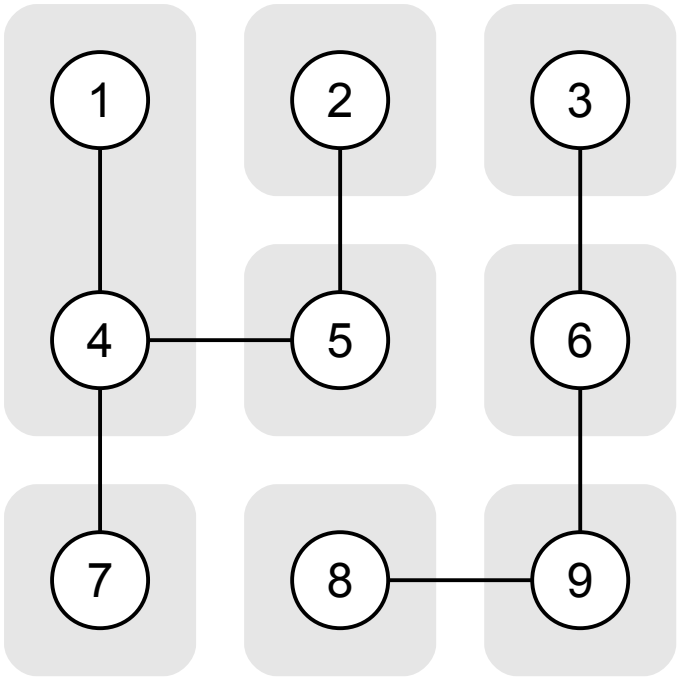
- Rappresentiamo il circuito con un insieme $V = \{1, ..., n\}$ di n nodi (pin) collegati da segmenti conduttivi
- Indichiamo con E la lista di coppie (v_1, v_2) di pin che sono tra di loro adiacenti (collegati)
- Vogliamo pre-processare il circuito in modo da rispondere in maniera efficiente a interrogazioni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

Esempio



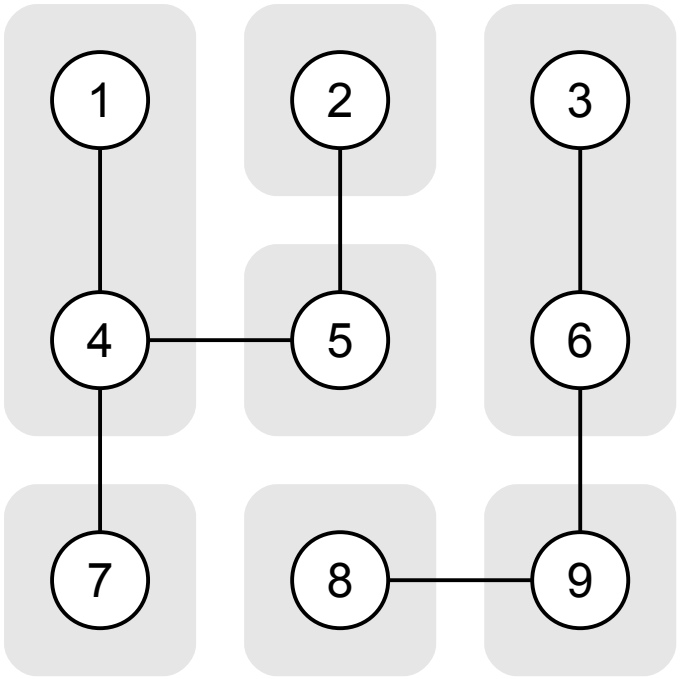
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



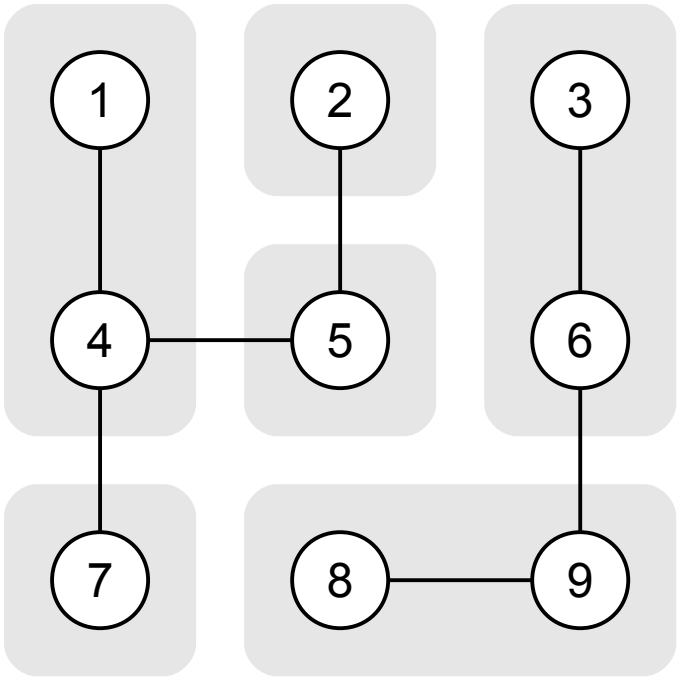
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



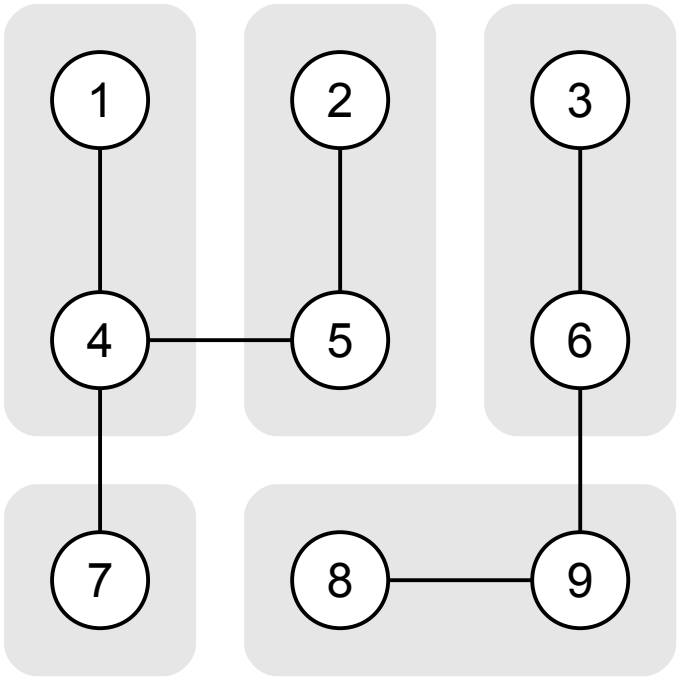
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



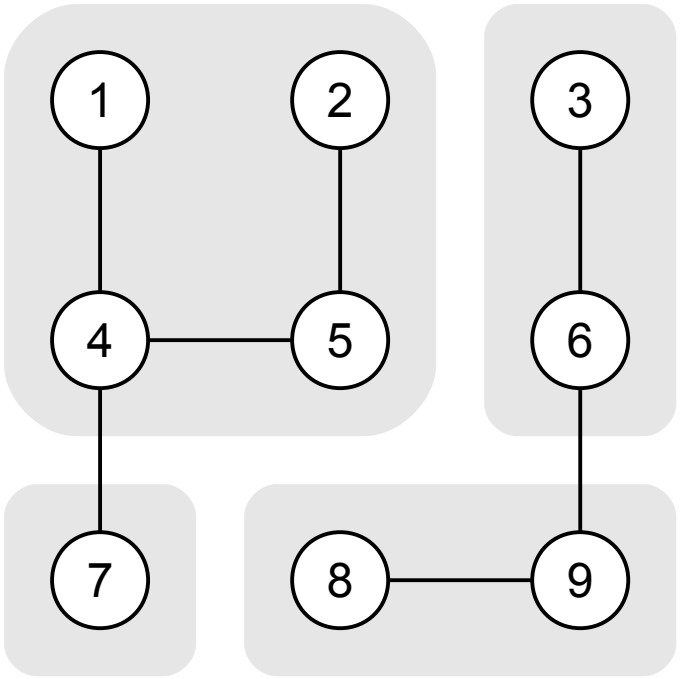
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



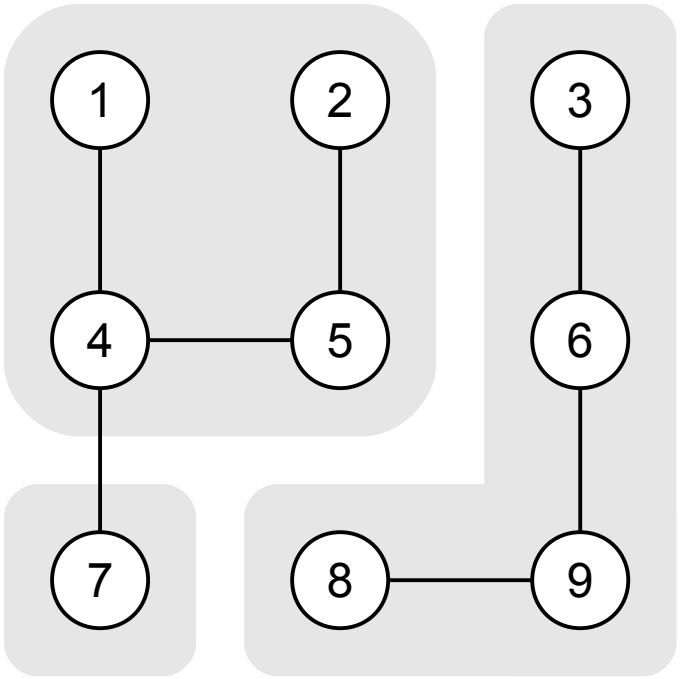
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



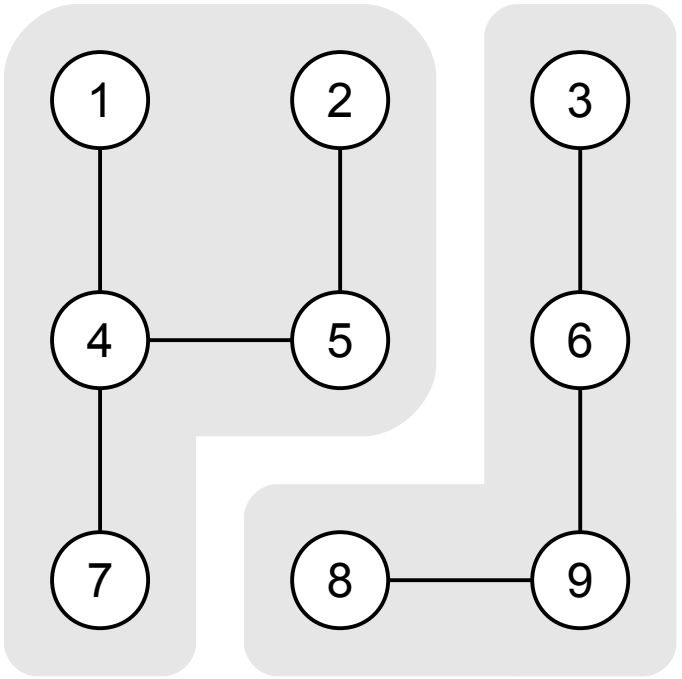
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



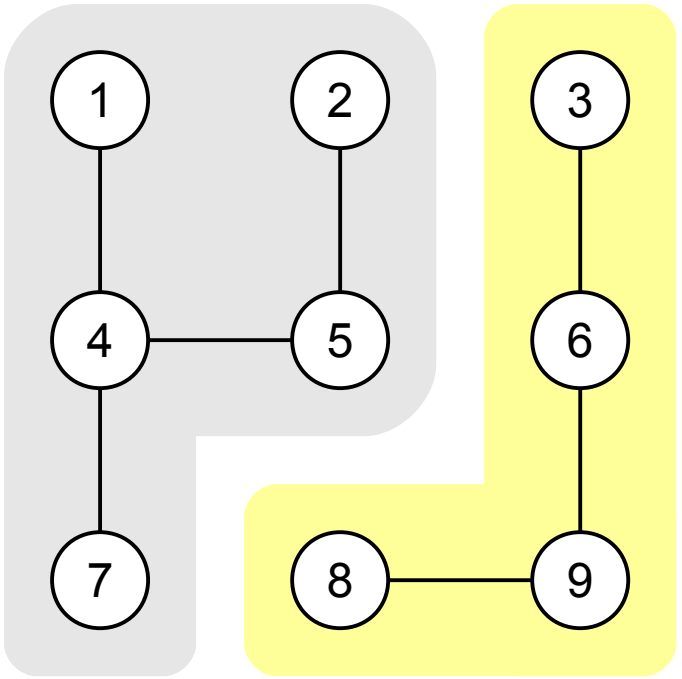
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



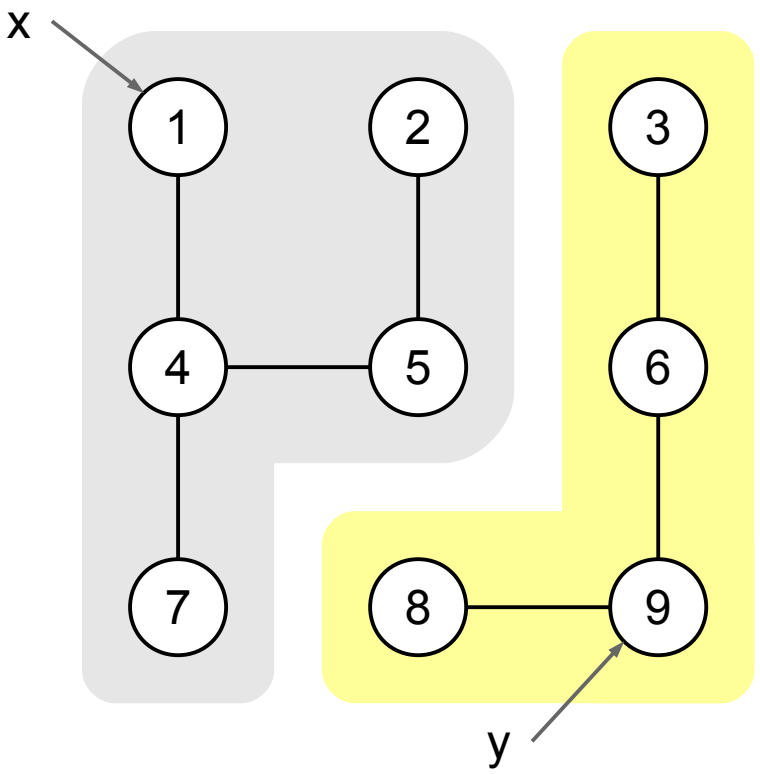
- Insieme E =
  - {1, 4}
  - {3, 6}
  - {8, 9}
  - {2, 5}
  - {4, 5}
  - {6, 9}
  - {4, 7}

Esempio



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

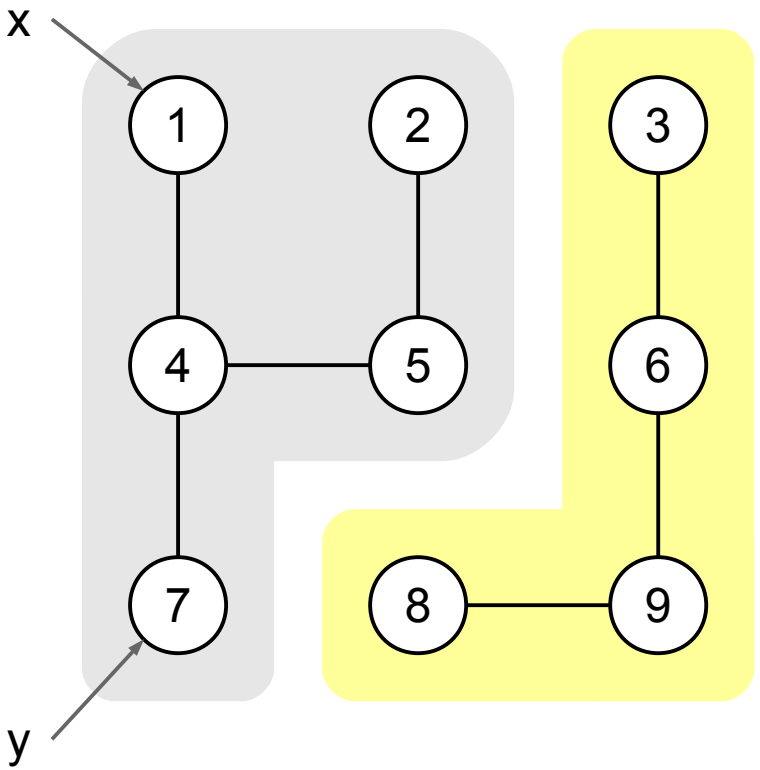
Esempio



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

$$find(x) \neq find(y)$$

Esempio



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

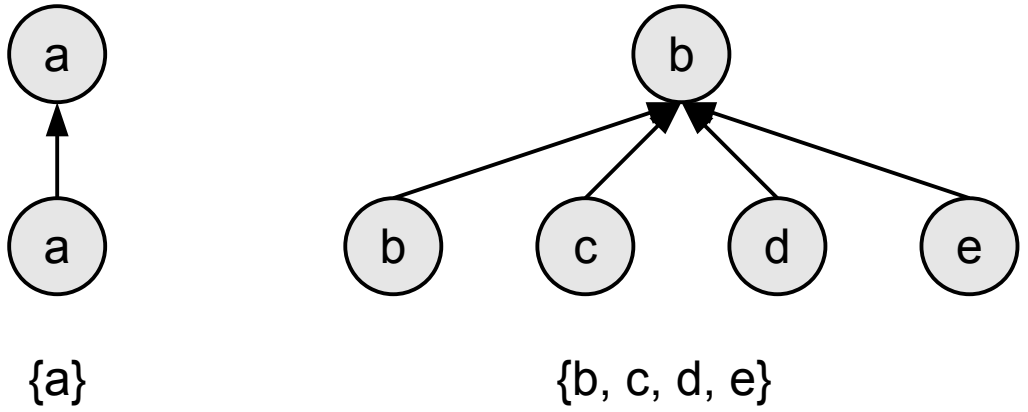
find(x) = find(y)

Implementazione di Union-Find

- Algoritmi elementari:
  - Algoritmo QuickFind: alberi di altezza uno
    - `makeSet()`,`find()`: O(1); `union()`: O(n)
  - Algoritmo QuickUnion: alberi generali
    - `makeSet()`,`union()`: O(1); `find()`: O(n)
- Algoritmi basati su euristiche di bilanciamento
  - QuickFind—Euristica sul peso
  - QuickUnion—Euristica sul rango

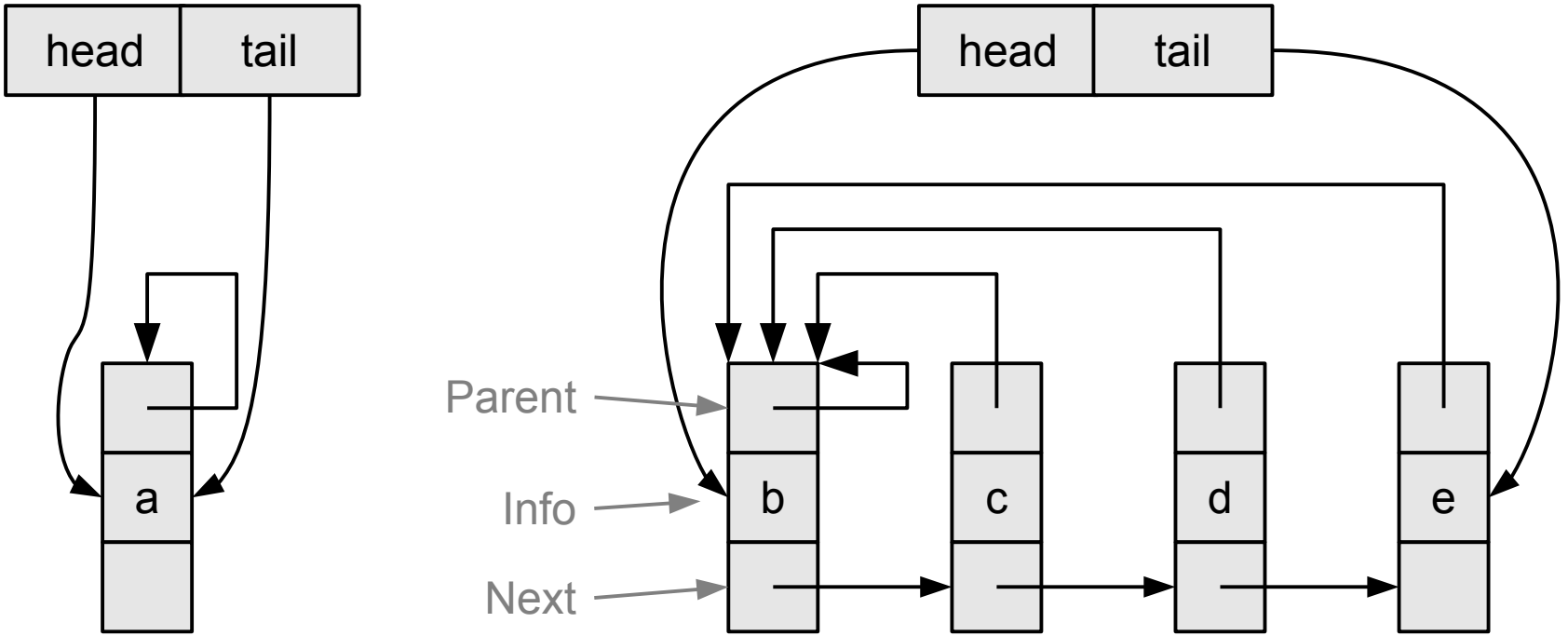
QuickFind

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

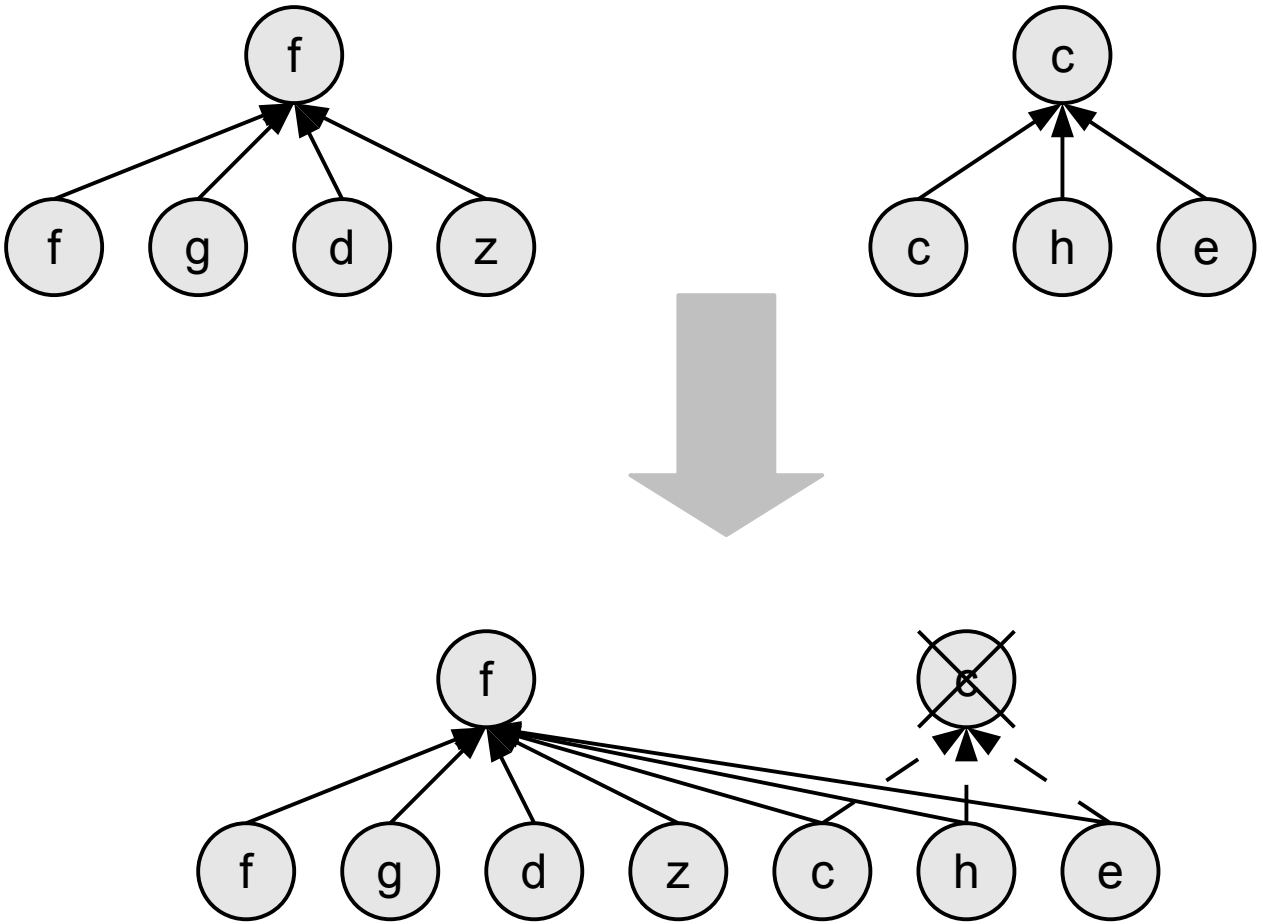


Nota implementativa

- È possibile rappresentare gli insiemi disgiunti tramite liste



QuickFind: Esempio—`union(find(g), find(h))`

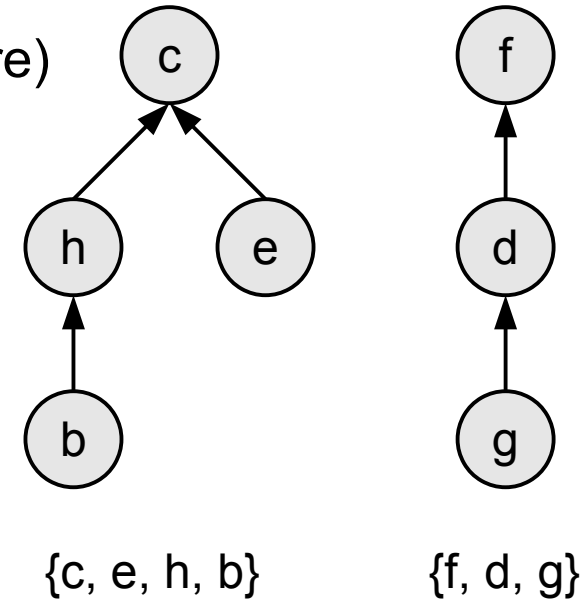


QuickFind

- Le operazioni `makeSet()` e `find()` richiedono tempo O(1):
  - `makeSet(x)`: crea un albero in cui l'unica foglia è *x* e il rappresentante di *x* è *x* stesso; costo O(1)
  - `find(x)`: restituisce il puntatore al padre di *x*; costo O(1)
- L'operazione `union(A,B)` richiede più tempo:
  - Tutte le foglie dell'albero B vengono spostate nell'albero A
  - Costo nel caso pessimo O(n), essendo n il numero complessivo di elementi in entrambi gli insiemi disgiunti
    - Infatti nel caso peggiore B ha n-1 elementi

QuickUnion

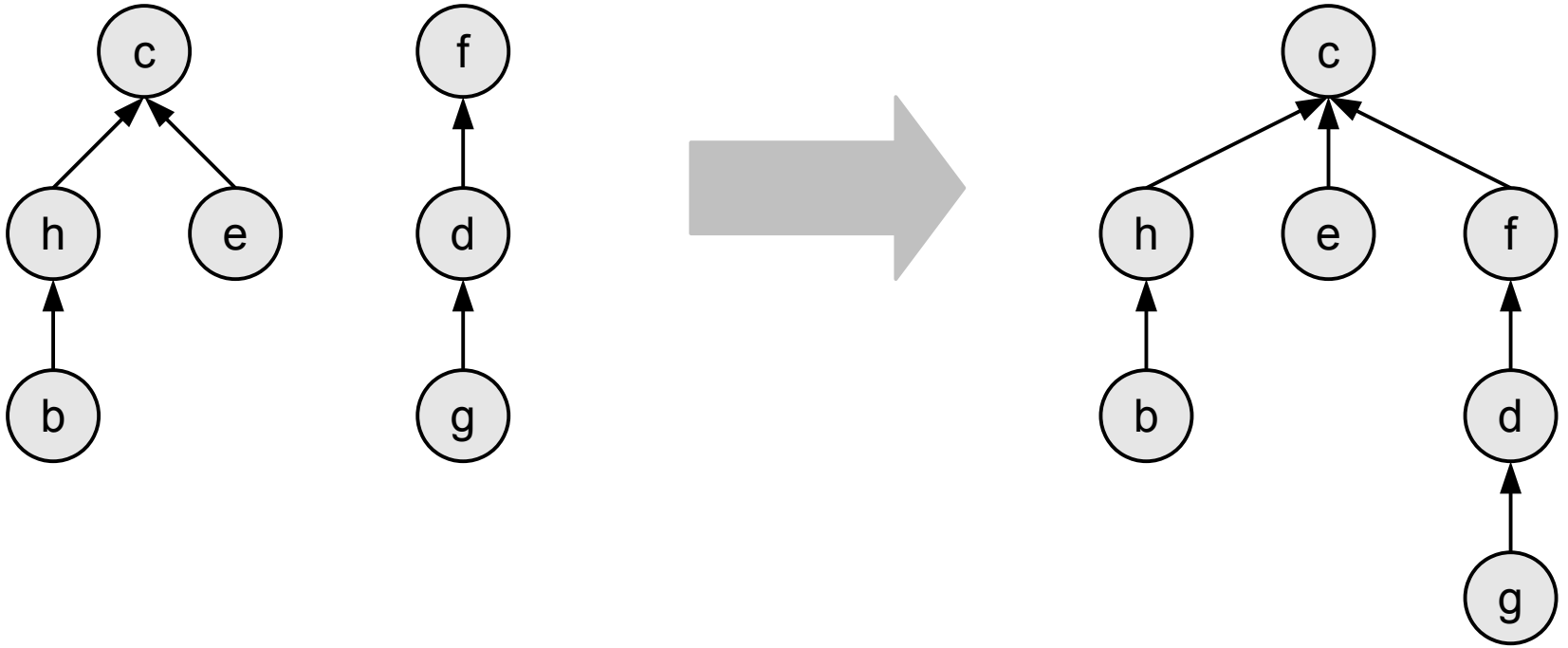
- Implementazione basata su foresta
  - Si rappresenta ogni insieme tramite un albero radicato generico
  - Ogni nodo dell'albero contiene
    - l'oggetto
    - un puntatore al padre (la radice non ha padre)
  - Il rappresentante è la radice



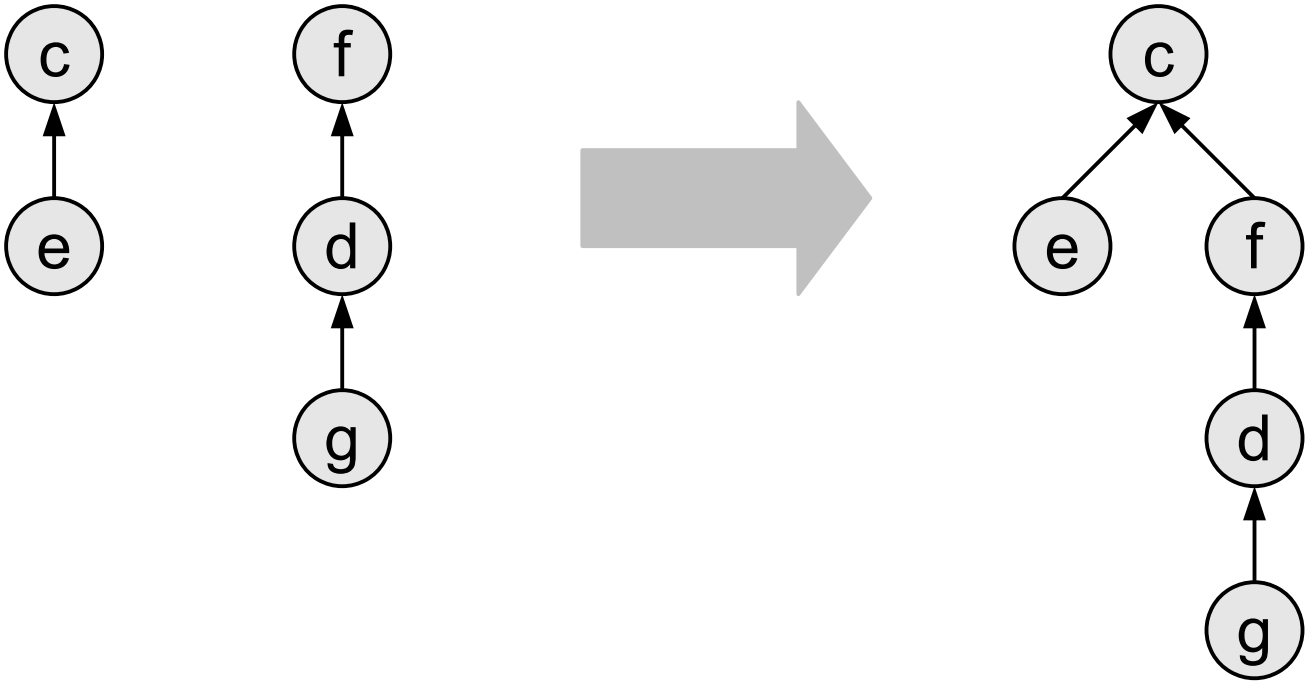
QuickUnion

- `makeSet(x)`
  - Crea un albero con un unico nodo *x*
  - Costo: O(1) nel caso pessimo
- `find(x)`
  - Risale la lista dei padri di *x* fino a trovare la radice e restituisce la radice come oggetto rappresentante
  - Costo: O(n) nel caso pessimo
- `union(A,B)`
  - Appende l'albero B ad A, rendendo la radice di B figlia della radice di A
  - Costo: O(1) nel caso pessimo

QuickUnion: Esempio–`union(c, f)`

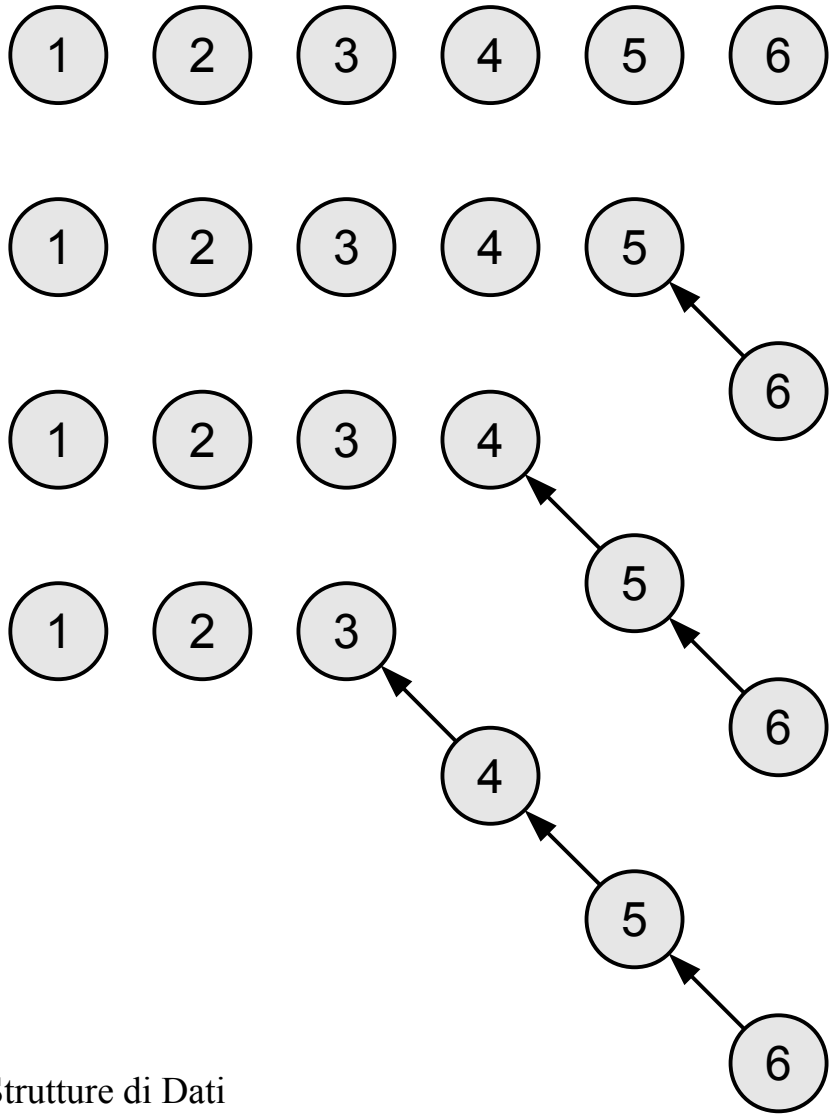


QuickUnion: Esempio–`union(c, f)`



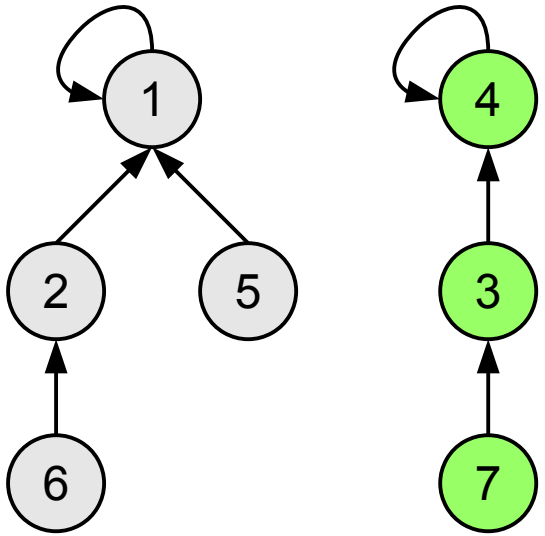
Caso pessimo per find()

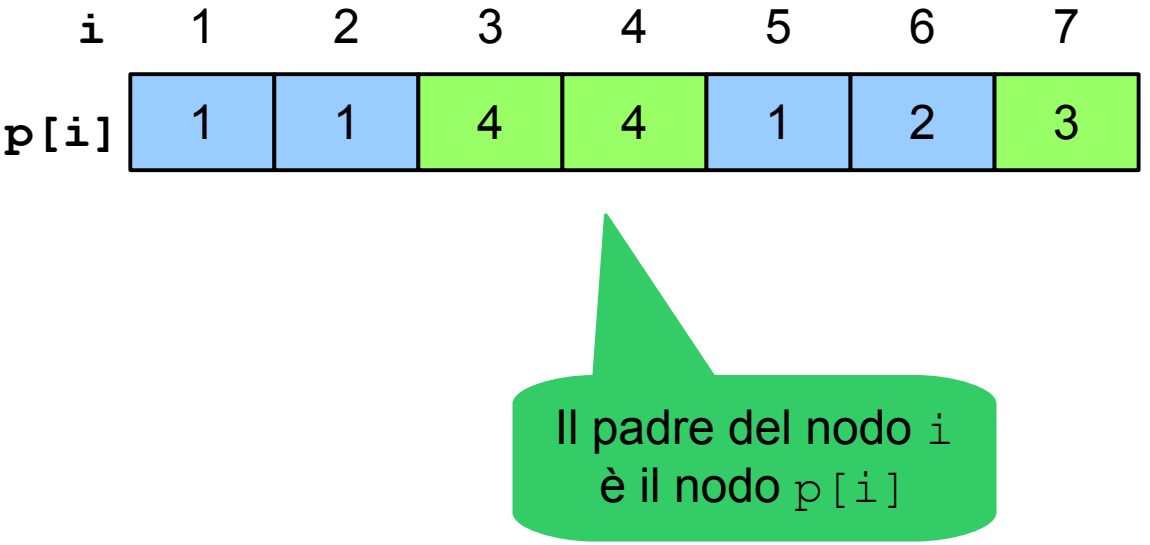
```
makeSet(i) i=1..6

union(5,6)

union(4,5)

union(3,4) ...
```



Nota implementativa

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





Riepilogo

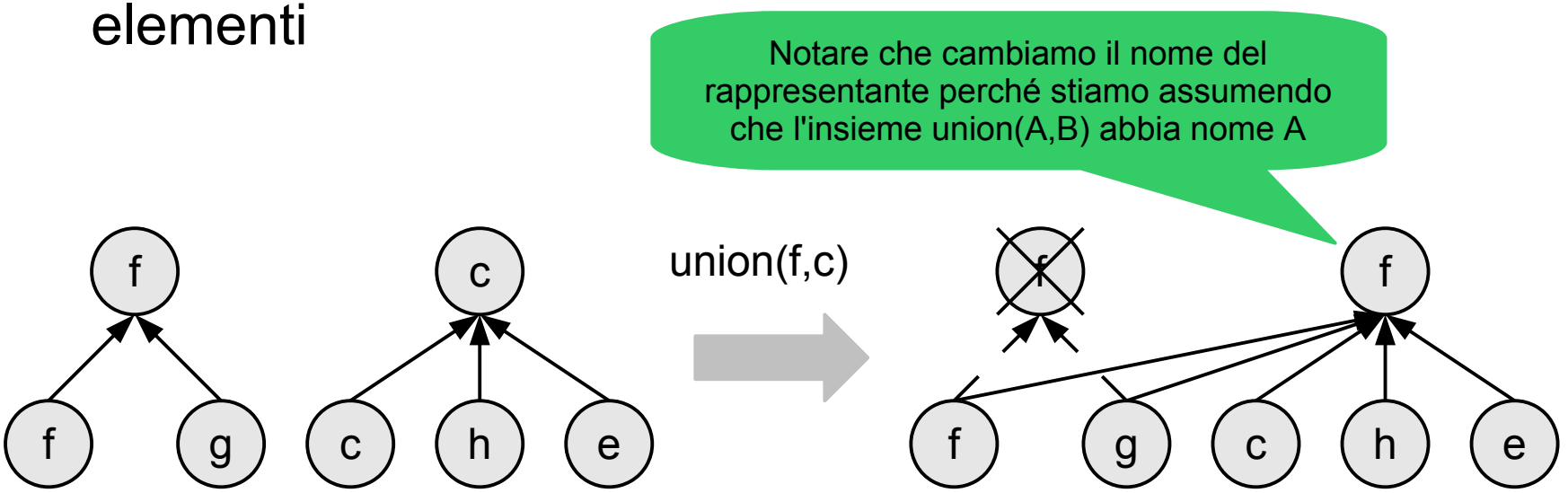
	QuickFind	QuickUnion
makeSet	O(1)	O(1)
union	O(n)	O(1)
find	O(1)	O(n)

Considerazioni

- Quando usare....
  - QuickFind?
    - Quando le `union()` sono rare e le `find()` frequenti
  - QuickUnion?
    - Quando le `find()` sono rare e le `union()` frequenti
- È importante sapere che esistono tecniche euristiche che permettono di migliorare questi risultati

QuickFind: Euristica sul peso

- Una strategia per diminuire il costo dell'operazione `union()` in QuickFind consiste nel:
  - Memorizzare nella radice il numero di elementi dell'insieme; la dimensione può essere mantenuta in tempo O(1)
  - Appendere l'insieme con meno elementi a quello con più elementi



Osservazioni / 1

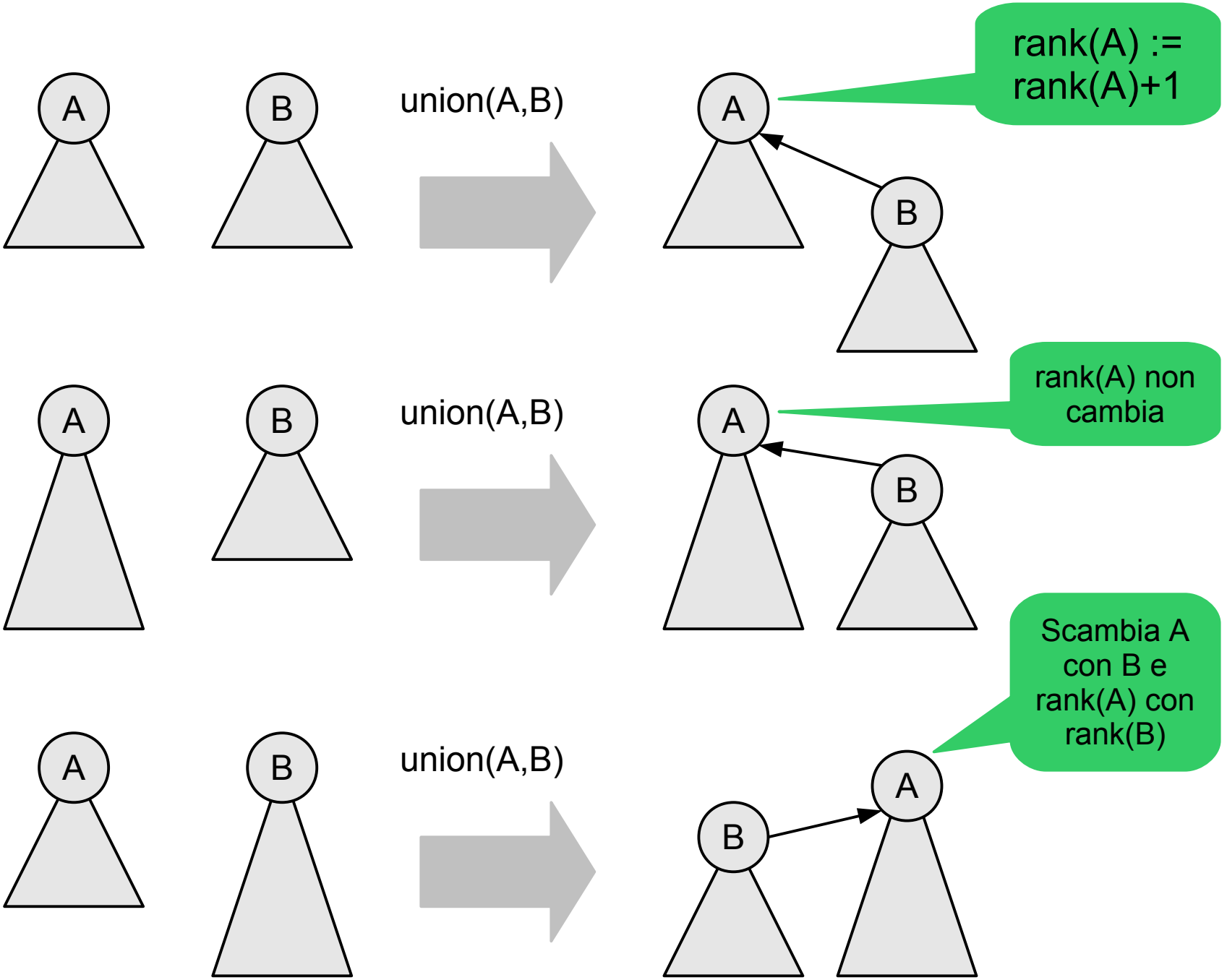
- Ogni volta che una foglia acquista un nuovo padre, fa parte di un insieme che ha *almeno il doppio di elementi* di quello cui apparteneva
- Dimostrazione
  - union(A,B) con size(A) ≥ size(B)
    - Le foglie di B cambiano padre
    - size(A)+size(B) ≥ size(B)+size(B) = 2 size(B)
  - union(A,B) con size(A) ≤ size(B)
    - Le foglie di A cambiano padre
    - size(A)+size(B) ≥ size(A)+size(A) = 2 size(A)
- Conclusione:
  - ogni foglia cambia il proprio padre al più log n volte

Osservazioni / 2

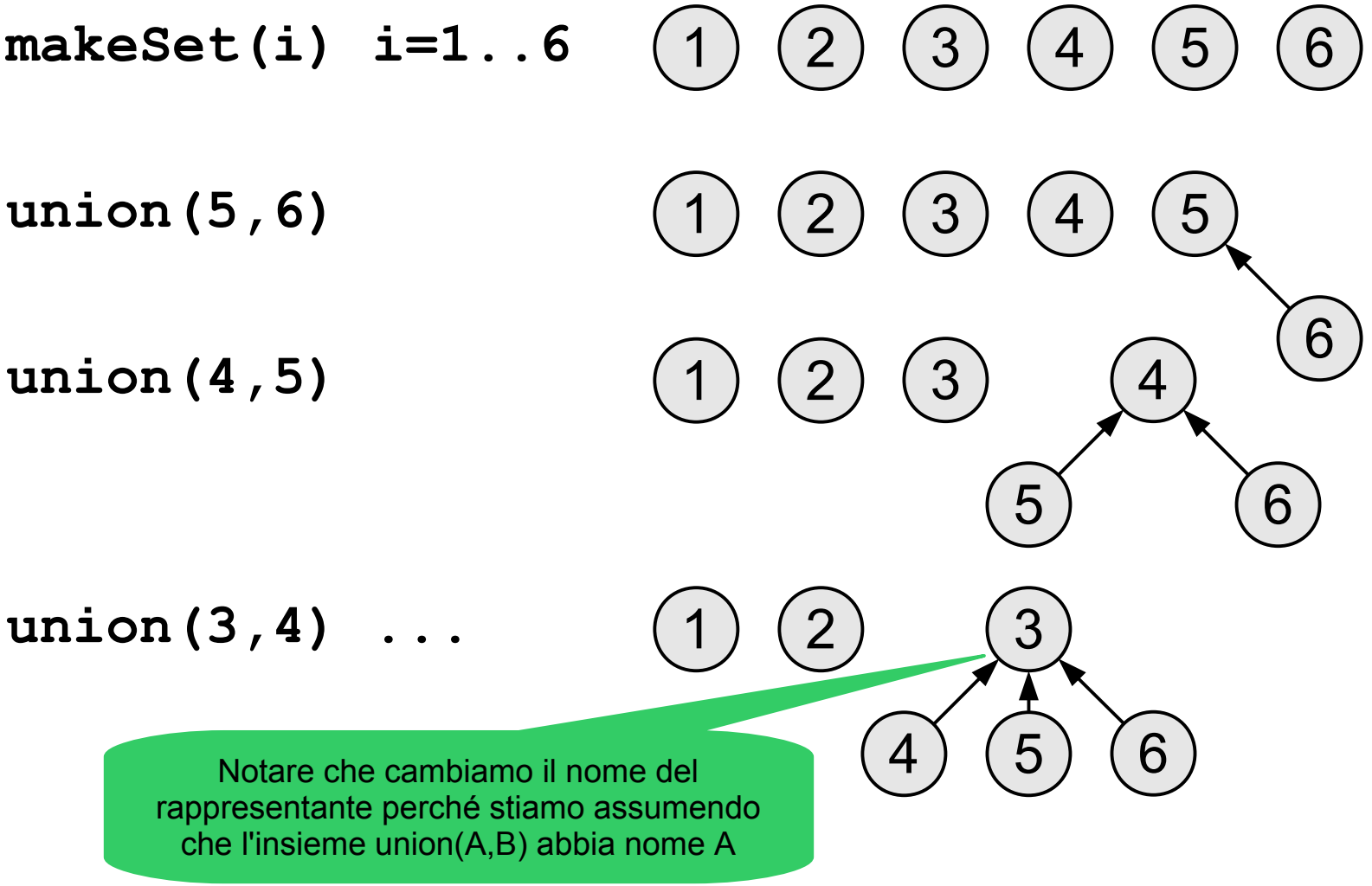
- Nel caso pessimo, durante una esecuzione di `union`, fino a n/2 elementi possono cambiare di padre:
  - Costo di `union()` nel caso pessimo: O(n)
- Quantifichiamo ora il costo ammortizzato:
  - Ricordiamo che il costo ammortizzato è dato dal costo complessivo di k esecuzioni (per un qualche k), diviso k
  - Considero n-1 esecuzioni (numero max di unioni, in quanto dopo n-1 unioni tutti gli n elementi saranno uniti)
  - Ogni esecuzione ha costo O(w), con w numero di "cambi di padre" da effettuare
  - Il numero complessivo massimo di cambi di padre è n log n
    - Ognuno degli n elementi cambia padre al più log n volte
  - Il costo ammortizzato risulta essere O(n log n) / n-1 = O(log n)

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
  - rango ≡ altezza del sottoalbero radicato sul nodo


A
B
union(A,B)
A
B
rank(A) := rank(A)+1
A
B
union(A,B)
A
B
rank(A) non cambia
A
B
union(A,B)
B
A
Scambia A con B e rank(A) con rank(B)

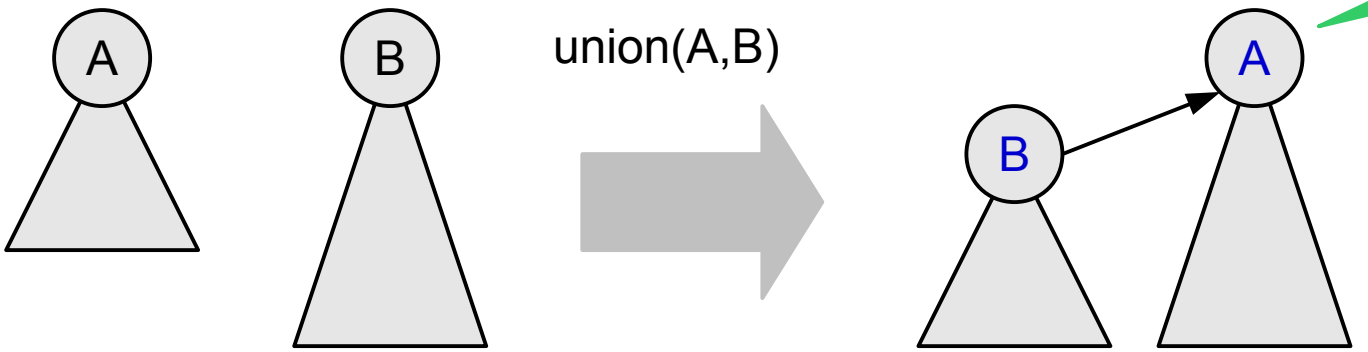
Esempio


makeSet(i) i=1..6
1 2 3 4 5 6
union(5,6)
1 2 3 4 5
6
union(4,5)
1 2 3
4
5 6
union(3,4) ...
1 2
3
4 5 6
Notare che cambiamo il nome del rappresentante perché stiamo assumendo che l'insieme union(A,B) abbia nome A

Proprietà alberi QuickUnion con euristica sul rango

- Un albero QuickUnion con euristica sul rango avente il nodo *x* come radice ha $n \geq 2^{rank(X)}$ nodi
- Dimostrazione: induzione sul numero di operazioni `union()` effettuate
  - **Base (0 operazioni union)**: tutti gli alberi hanno rango zero (singolo nodo) quindi hanno esattamente $2^0=1$ nodi
  - **Induzione**: consideriamo cosa succede prima e dopo una operazione `union(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 disgiunti

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

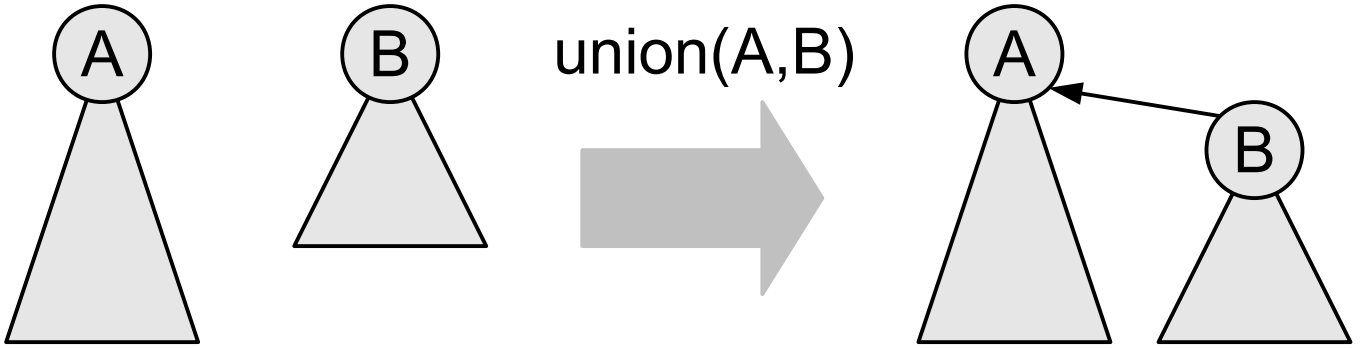


Nota: abbiamo scambiato A con B

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

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

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

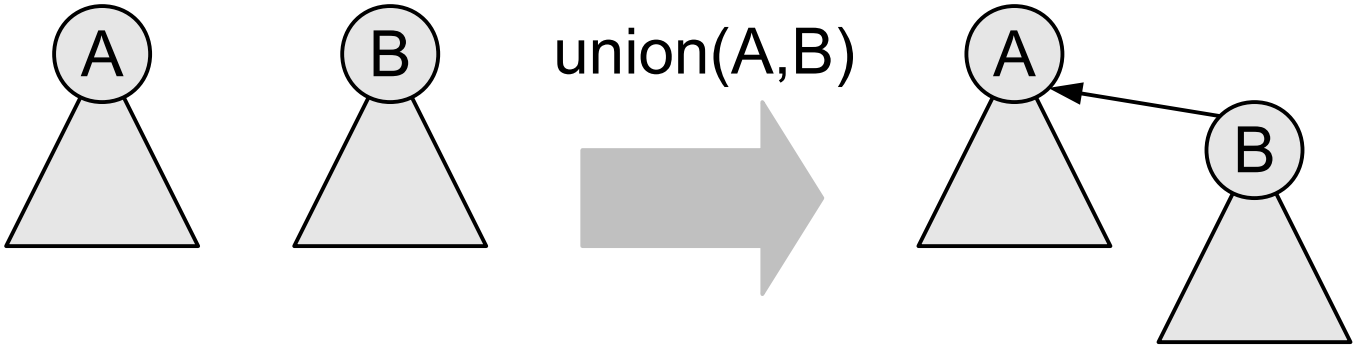


- |A U B| = |A|+|B|
- rank(A U B) = rank(A)
  - perché l'altezza dell'albero A U B è uguale all'altezza dell'albero A
- Per ipotesi induttiva, $|A| \geq 2^{rank(A)}$, $|B| \geq 2^{rank(B)}$
- Quindi

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

Passo induttivo

caso rank(A) = rank(B)



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

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

Costo operazioni di find

- Utilizzando l'euristica sul rango, `find()` ha costo O(log n) nel caso pessimo
- Dimostrazione
  - `find()` ha un costo O(h) con h altezza dell'albero
  - L'altezza di un albero QuickUnion A è rank(A)
  - Da quanto appena visto, $2^{rank(A)} \leq n$
  - Quindi altezza = rank(A) ≤ (log n)

Riepilogo

	QuickFind	QuickUnion	QuickFind eur. peso	QuickUnion eur. rank
makeSet	O(1)	O(1)	O(1)	O(1)
union	O(n)	O(1)	O(log n) amm.	O(1)
find	O(1)	O(n)	O(1)	O(log n)