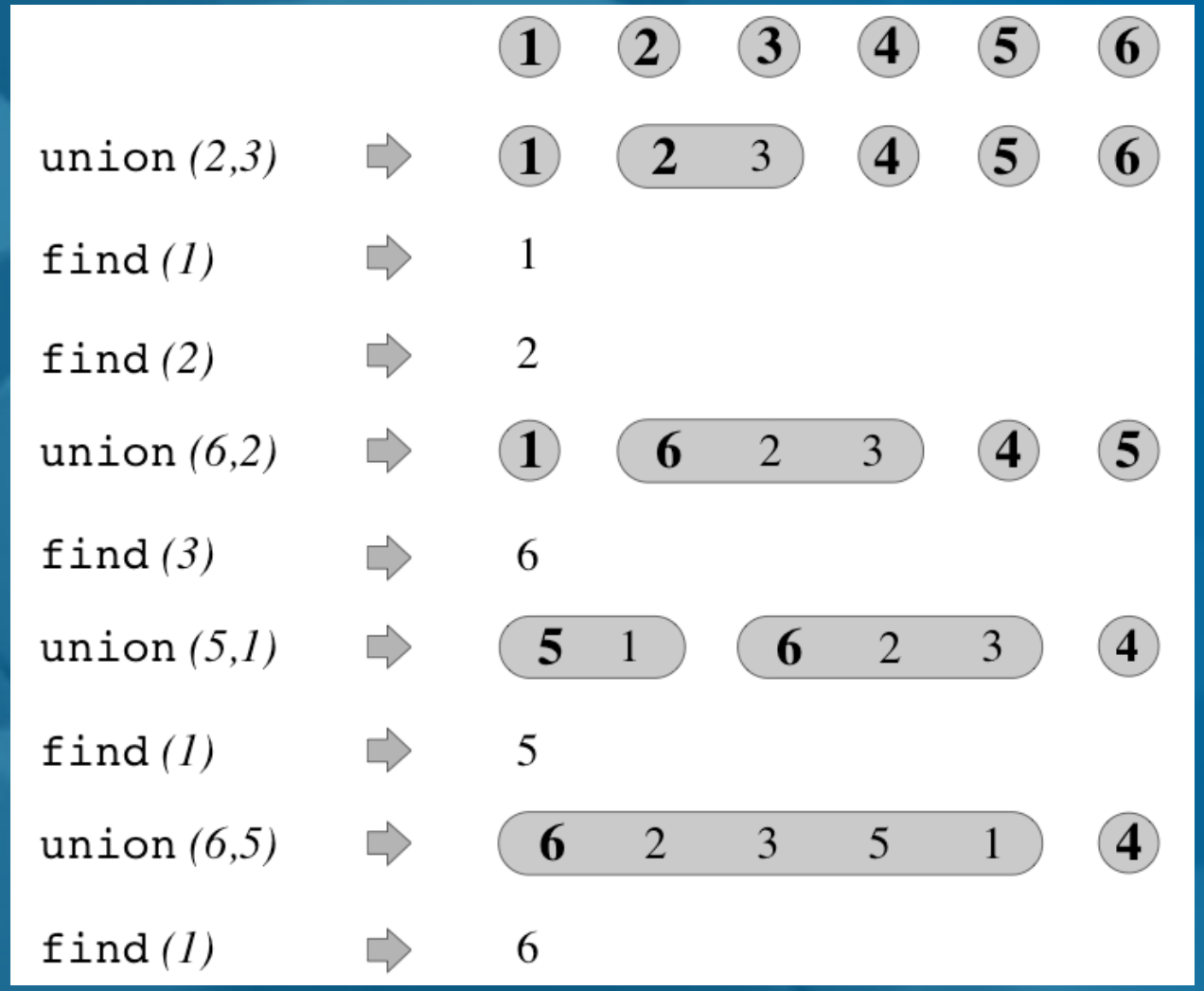
# ASD (II Mod)

Il problema della gestione
di insiemi disgiunti (**Union-Find**)

# Il problema Union-find

- Mantenere una collezione di insiemi disgiunti contenenti elementi distinti di un insieme (universo) **U** (ad esempio, **U** = {1,…,n}) permettendo sequenze di operazioni del seguente tipo:
  - **makeSet(x)** = crea il nuovo insieme x={x}
  - **union(A,B)** = unisce gli insiemi A e B in un unico insieme, di nome A, e distrugge i vecchi insiemi A e B (si suppone di accedere direttamente agli insiemi A,B)
  - **find(x)** = restituisce il nome dell'insieme contenente l'elemento x (si suppone di accedere **direttamente** all'elemento x)

# Esempio




n = 6

L'elemento in **grassetto** dà il nome all'insieme

D: Se ho n elementi, quante union posso fare al più?

R: n-1

Obiettivo: progettare una struttura dati che sia efficiente su una *sequenza arbitraria* di operazioni

# Idea generale: rappresentare gli insiemi disgiunti con una foresta

Ogni insieme è un albero radicato

La **radice** contiene il **nome** dell'insieme
(elemento rappresentativo)

# Alberi QuickFind

- Usiamo un foresta di alberi di altezza **1** per rappresentare gli insiemi disgiunti. In ogni albero:
  - Radice = nome dell'insieme
  - Foglie = elementi (incluso l'elemento rappresentativo, il cui valore è nella radice e dà il nome all'insieme)

# Realizzazione (1/2)

**classe** `QuickFind` **implementa** `UnionFind`:

**dati**: $S(n) = O(n)$

una collezione di insiemi disgiunti di elementi $elem$; ogni insieme ha un nome $name$.

**operazioni**:

`makeSet`$(elem\ e)$ $T(n) = O(1)$

crea un nuovo albero, composto da due nodi: una radice ed un unico figlio (foglia). Memorizza $e$ sia nella foglia dell'albero che come nome nella radice.

# Realizzazione (2/2)

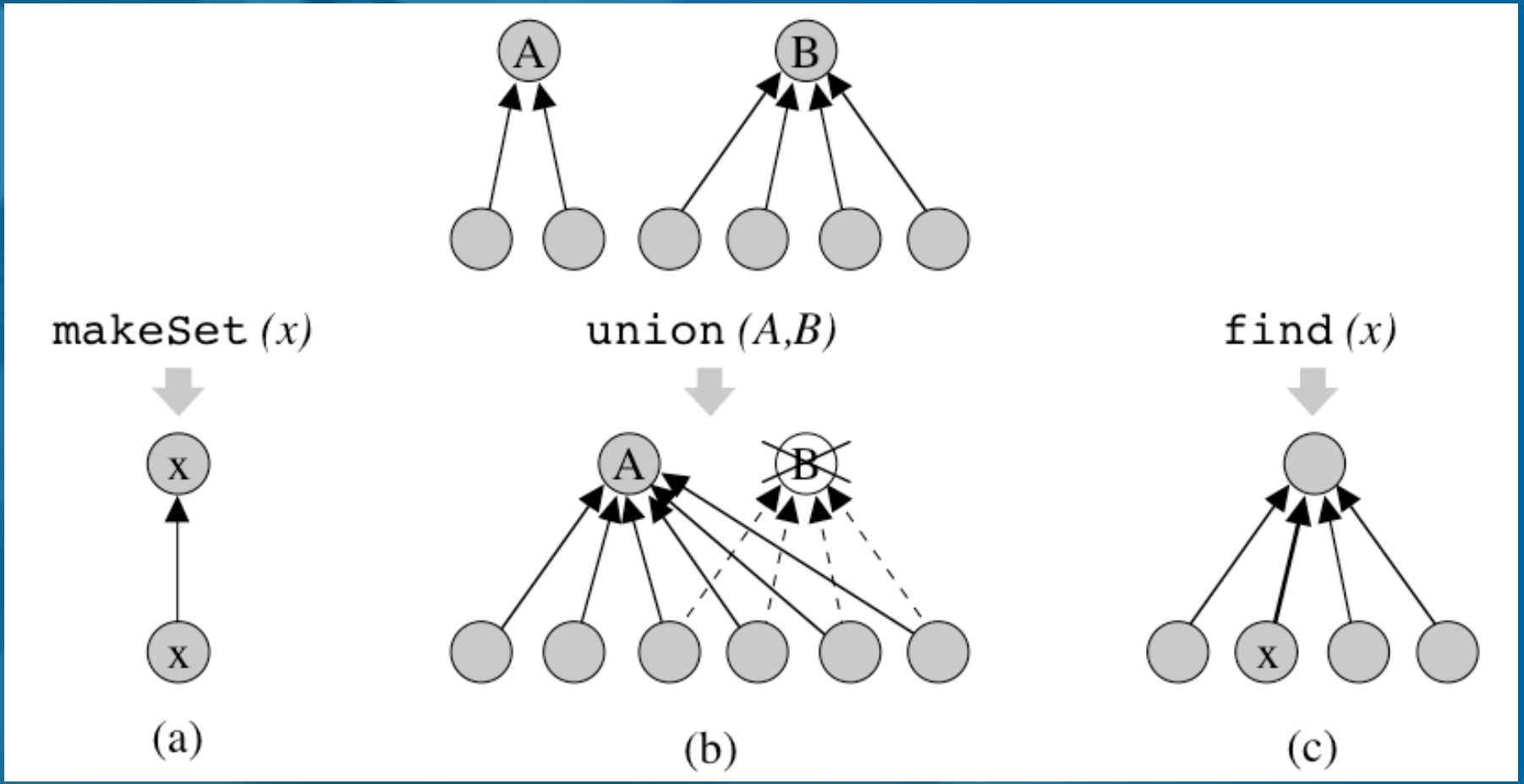
`union`$(name\ a, name\ b)$ $\qquad T(n) = O(n)$

considera l'albero $A$ corrispondente all'insieme di nome $a$, e l'albero $B$ corrispondente all'insieme di nome $b$. Sostituisce tutti i puntatori dalle foglie di $B$ alla radice di $B$ con puntatori alla radice di $A$. Cancella la vecchia radice di $B$.

`find`$(elem\ e) \rightarrow name$ $\qquad T(n) = O(1)$

accede alla foglia $x$ corrispondente all'elemento $e$. Da tale nodo segue il puntatore al padre, che è la radice dell'albero, e restituisce il nome memorizzato in tale radice.

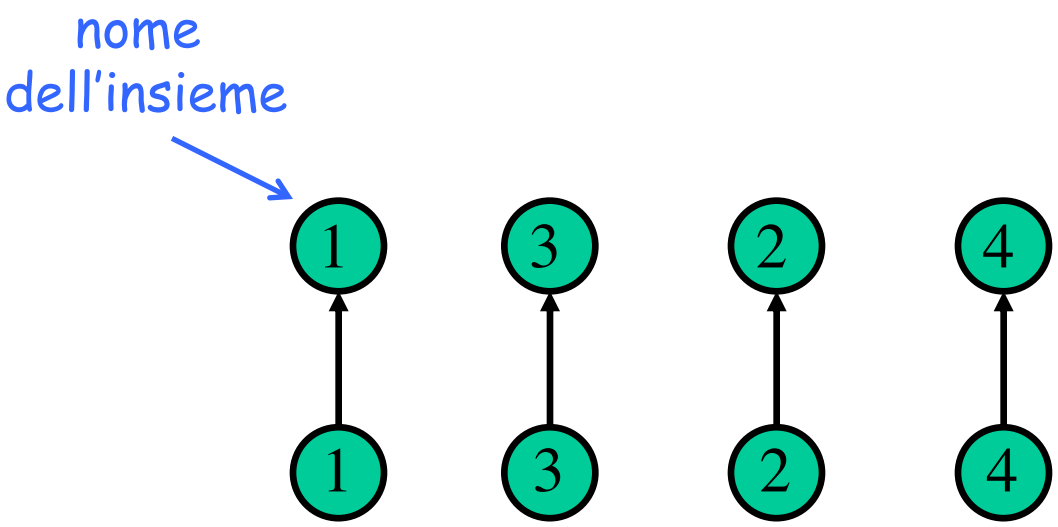
# Esempio



un esempio:

Sequenza di operazioni:

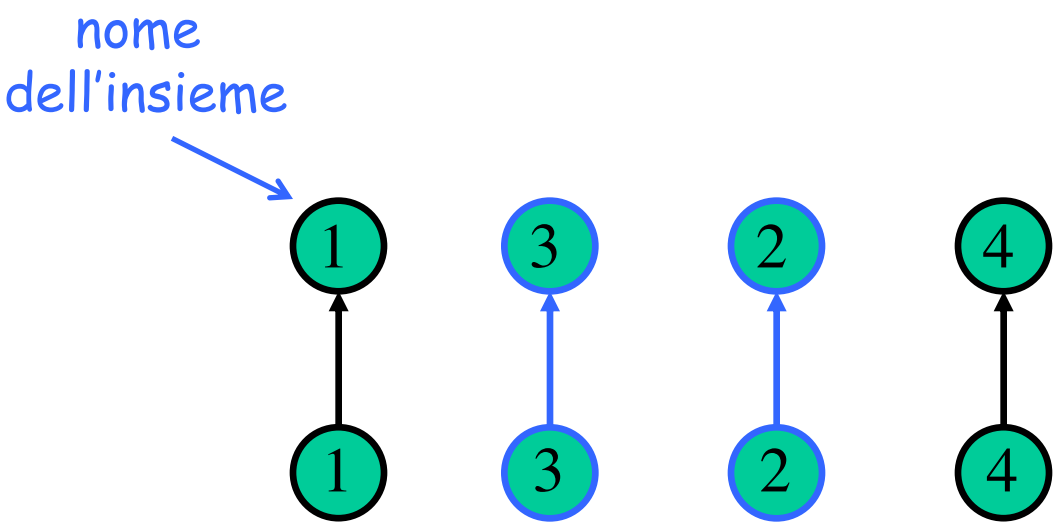
`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`



un esempio:

Sequenza di operazioni:

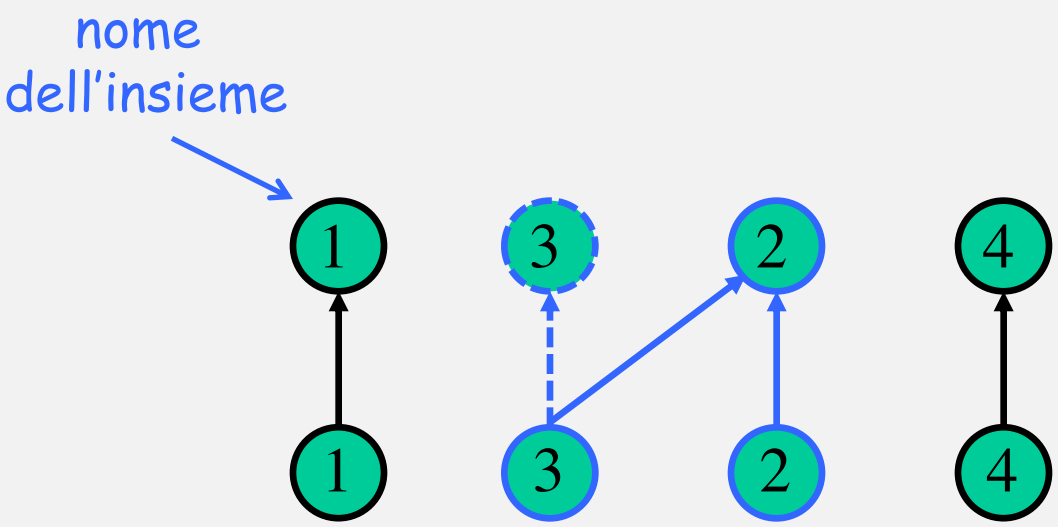
`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`



un esempio:

Sequenza di operazioni:

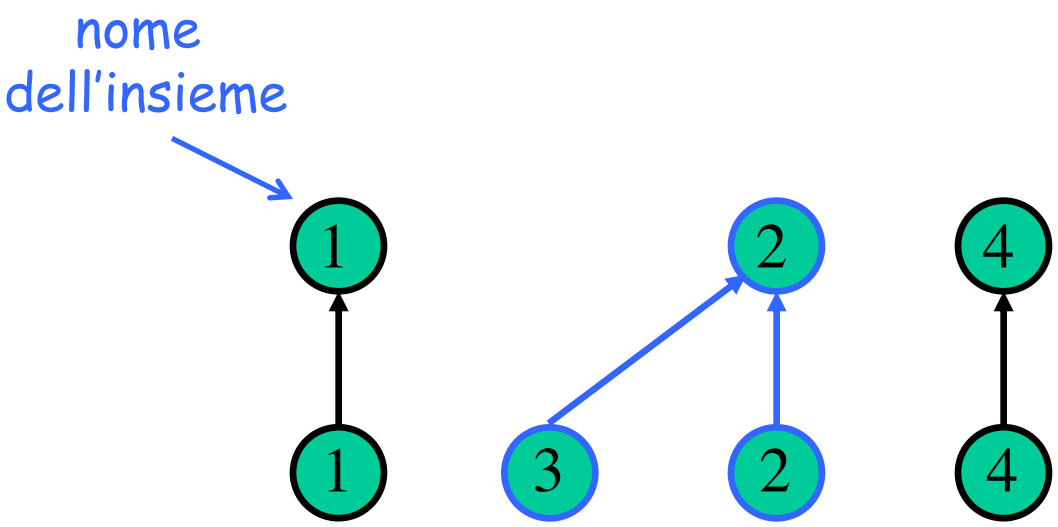
`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`



un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

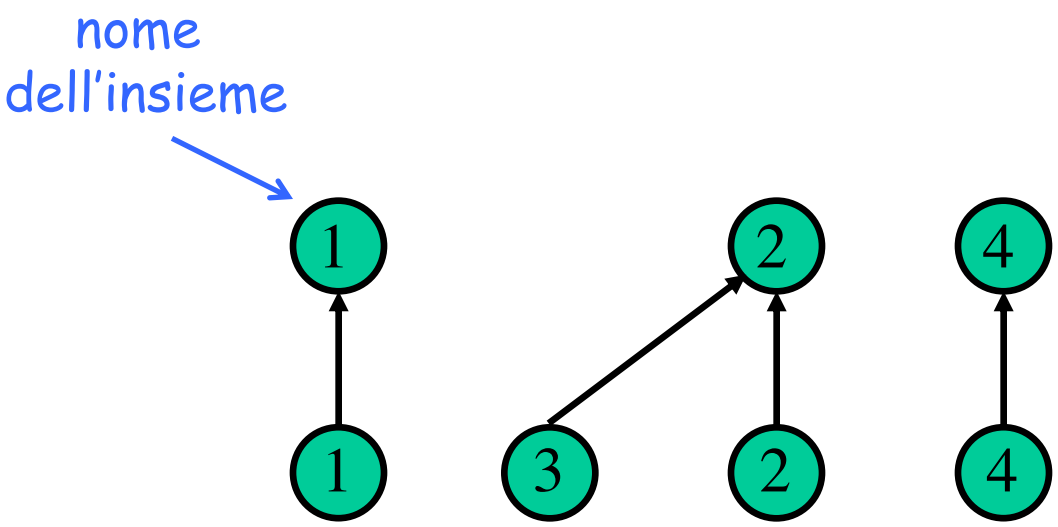


un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

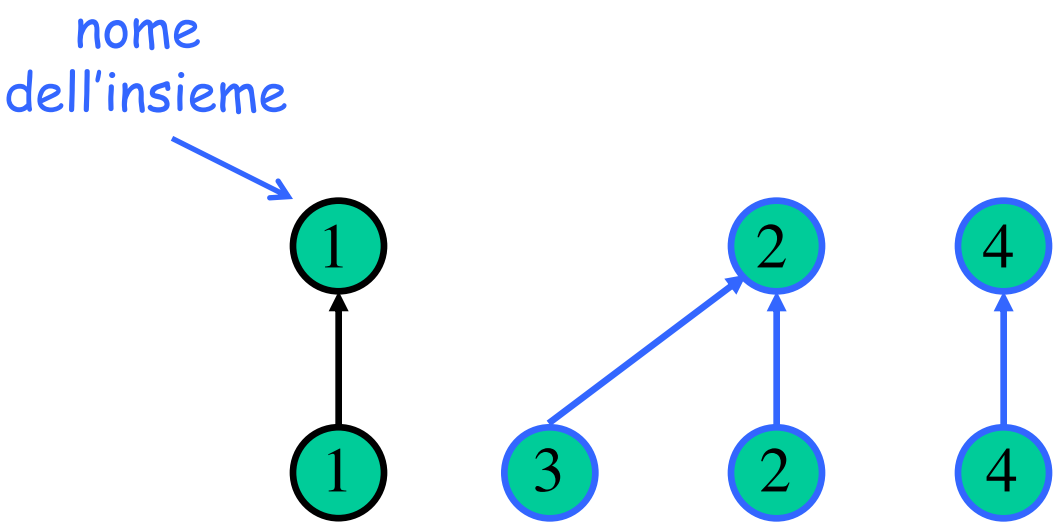


un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

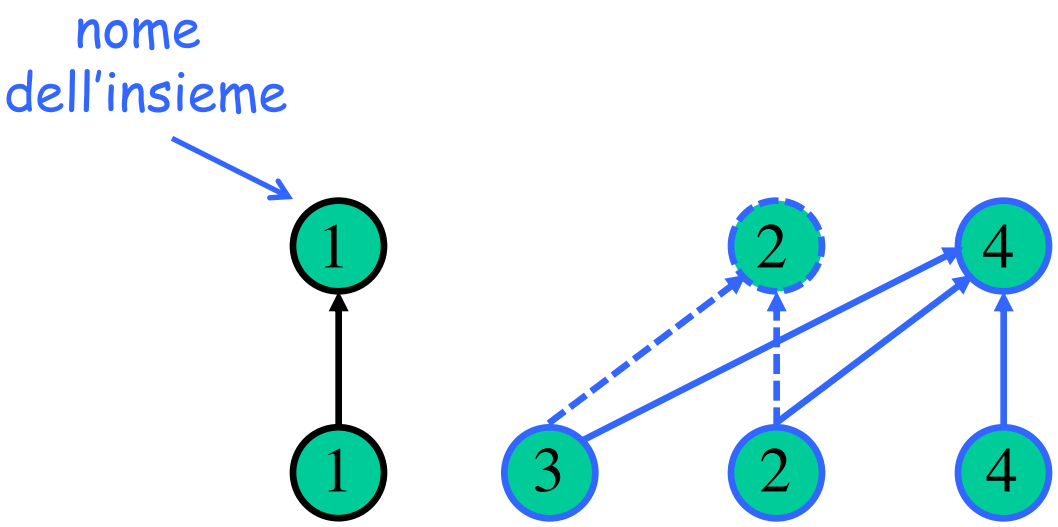


un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

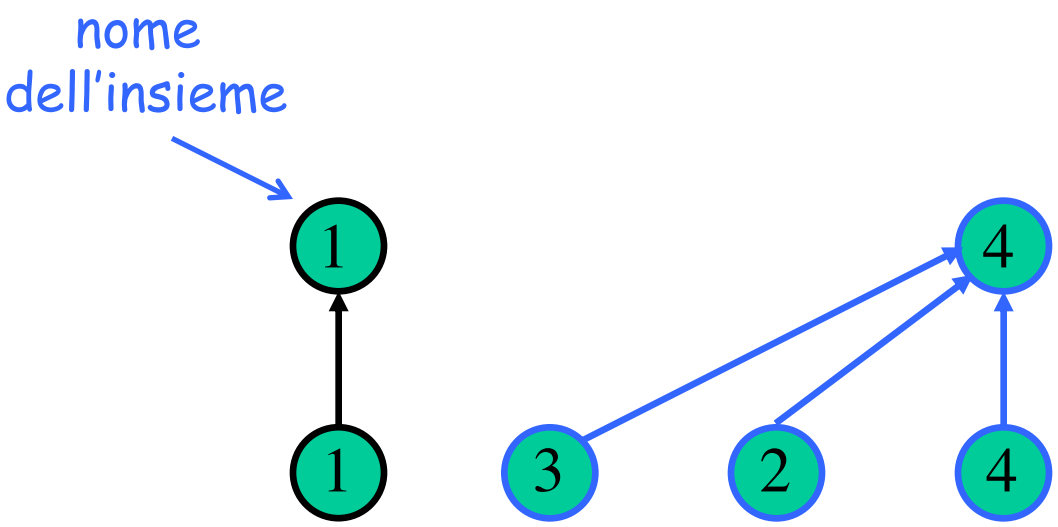


un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`



un esempio:

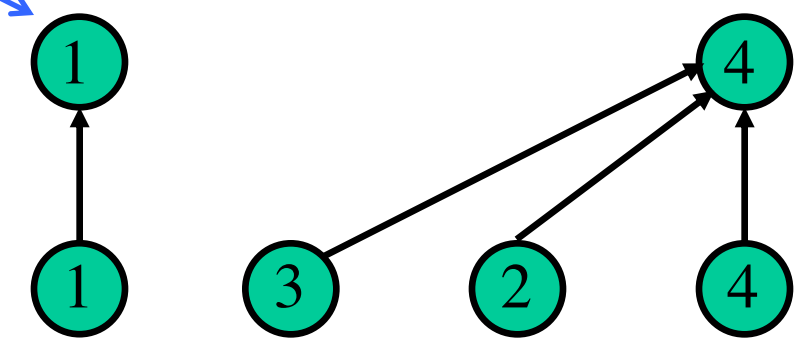
Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

`find(2)`

nome dell'insieme



un esempio:

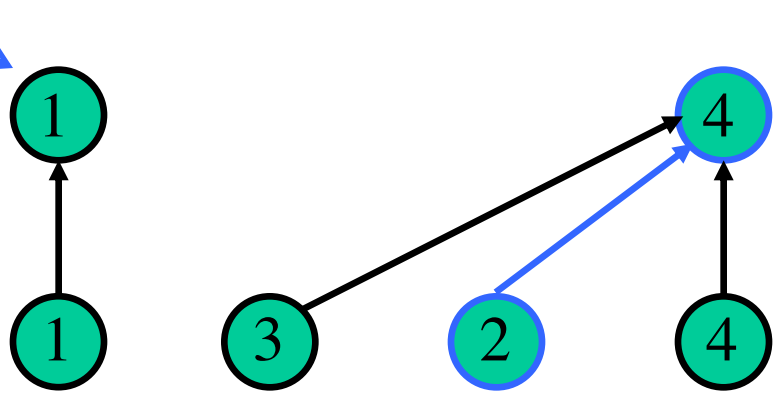
Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

`find(2)`

nome dell'insieme



# Union di costo lineare

find e makeSet richiedono solo tempo O(1), ma particolari sequenze di union possono essere molto inefficienti:

```
union (n-1, n)
union (n-2, n-1)
union (n-3, n-2)
   ⋮
union (2, 3)
union (1, 2)
```

1 operazione
2 operazioni
3 operazioni
:
n-2 operazioni
n-1 operazioni

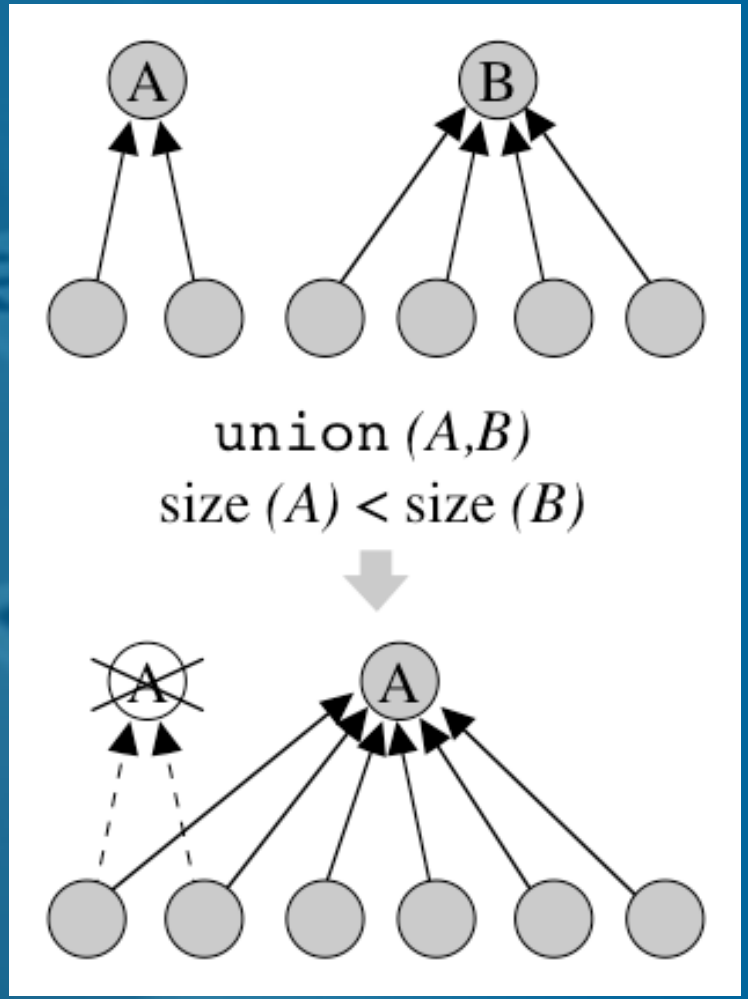
⇒ Se eseguiamo n makeSet, n-1 union come sopra, ed m find (in qualsiasi ordine), il tempo richiesto dall'intera sequenza di operazioni è $O(n+1+2+\ldots+(n-1)+m) = O(m+n^2)$

# Migliorare la struttura QuickFind: euristiche di bilanciamento nell'operazione union

Idea: fare in modo che un nodo/elemento non cambi troppo spesso padre

# Bilanciamento in alberi QuickFind

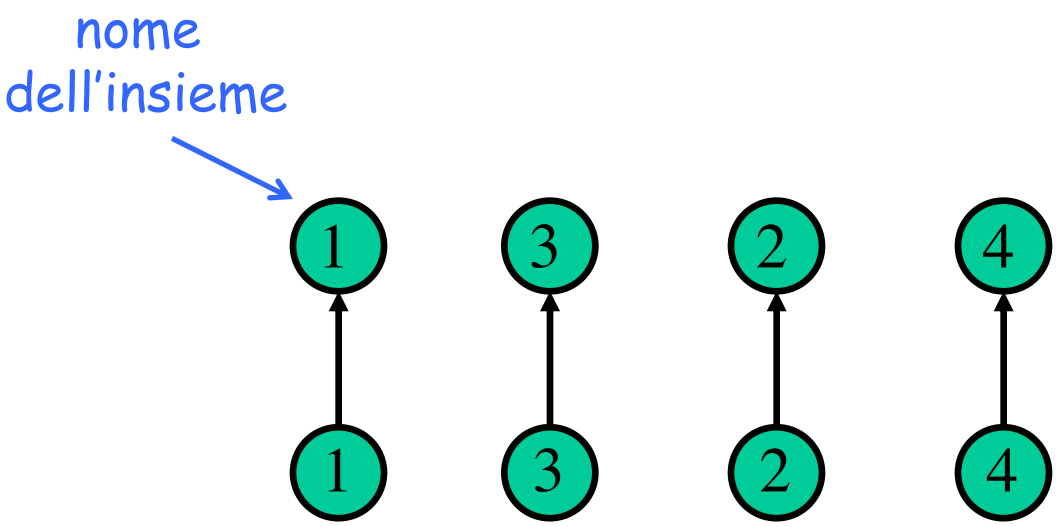
Nell'unione degli insiemi A e B, attacchiamo gli elementi dell'insieme di cardinalità minore a quello di cardinalità maggiore, e se necessario modifichiamo la radice dell'albero ottenuto (cosiddetta union by size)



un esempio:

Sequenza di operazioni:

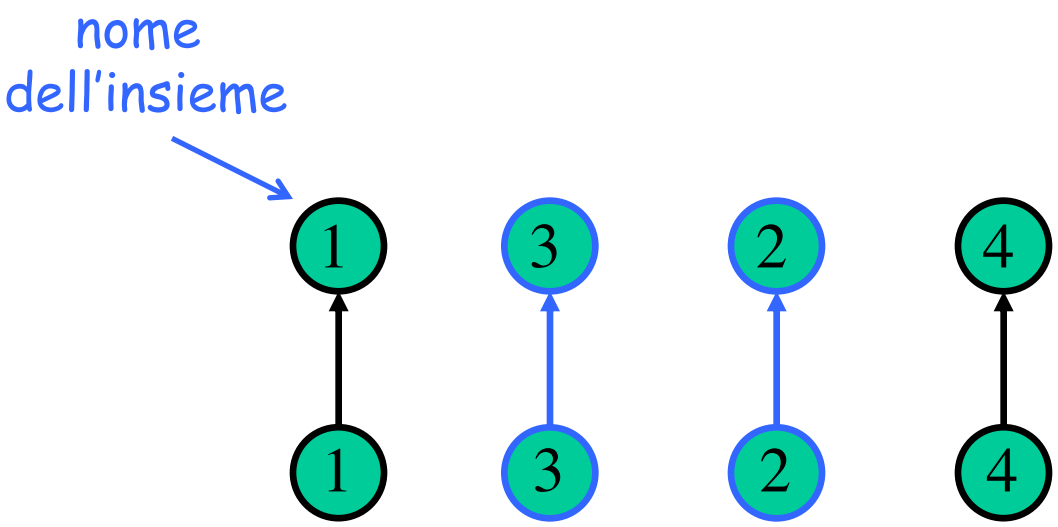
`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`



un esempio:

Sequenza di operazioni:

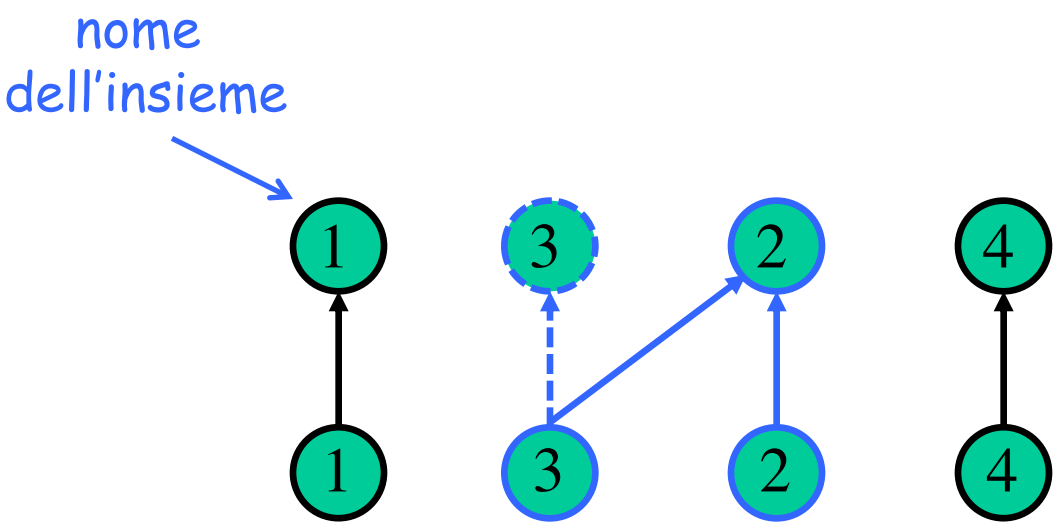
`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`



un esempio:

Sequenza di operazioni:

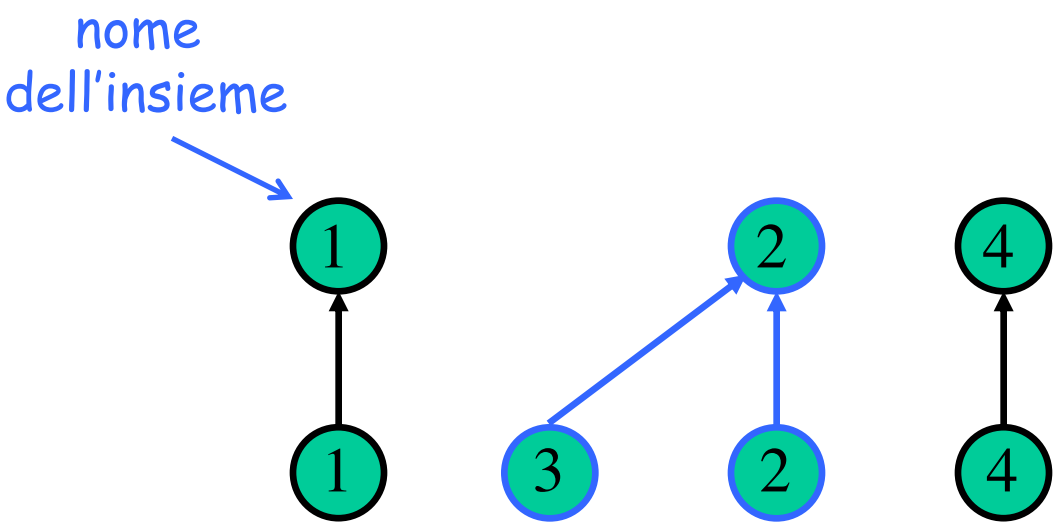
`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`



un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

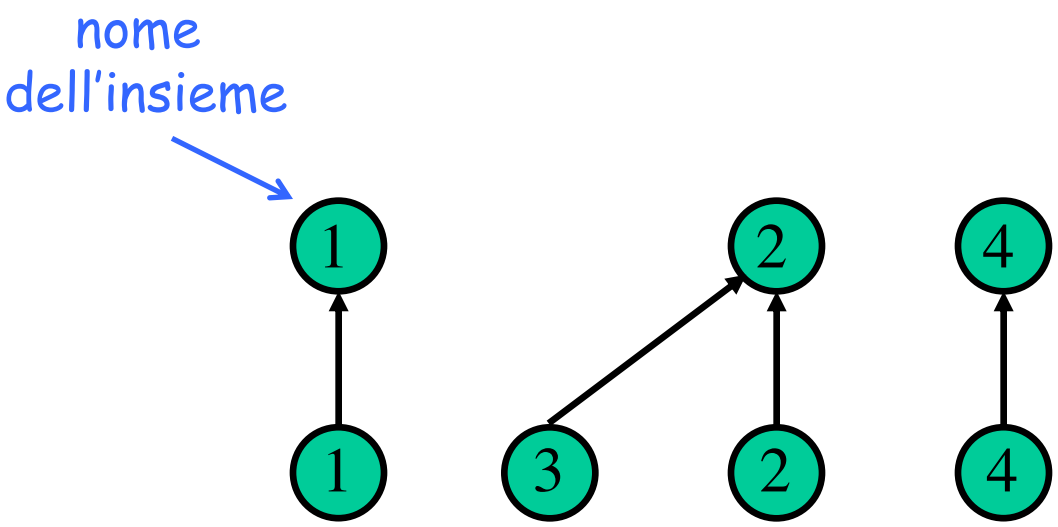


un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

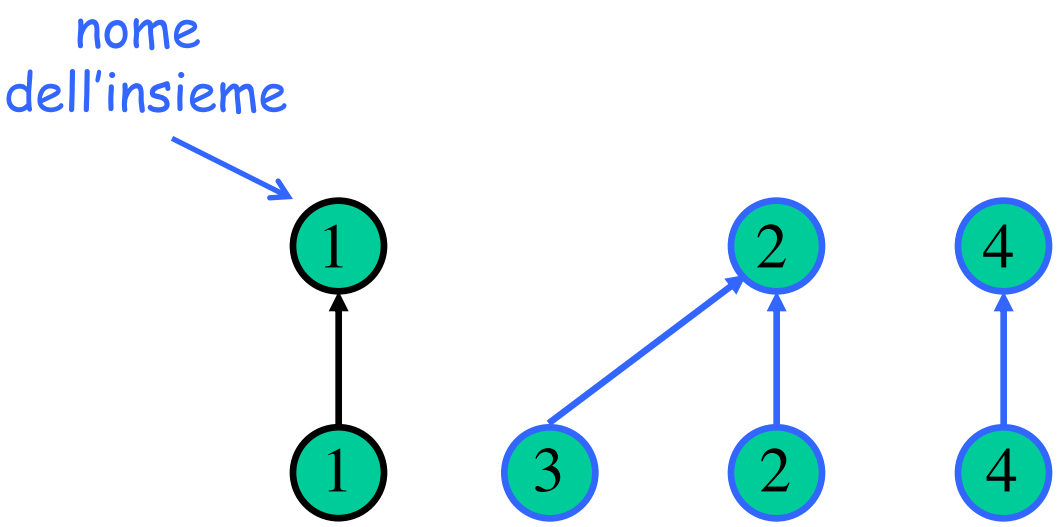


un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

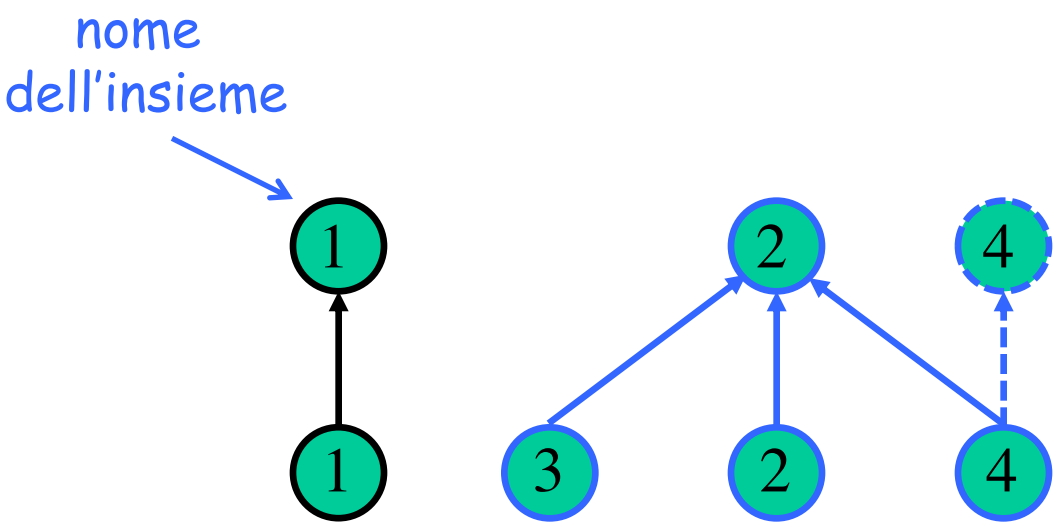


un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

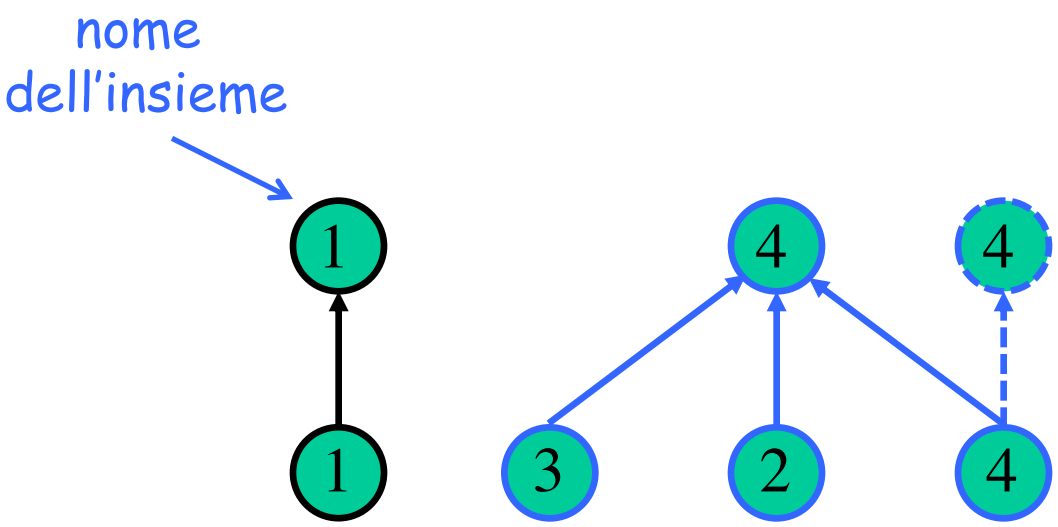


un esempio:

Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`



un esempio:

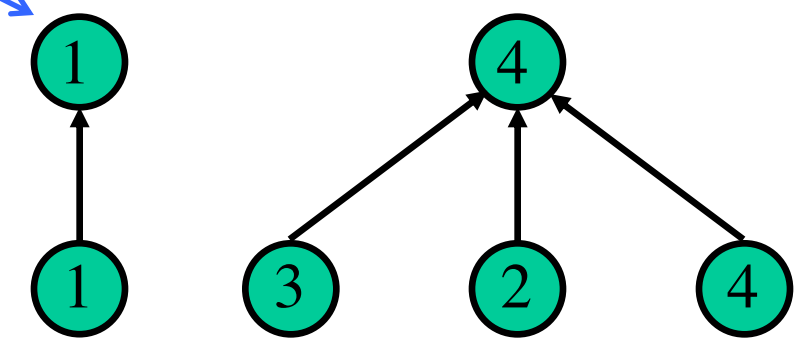
Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

`find(2)`

nome dell'insieme



un esempio:

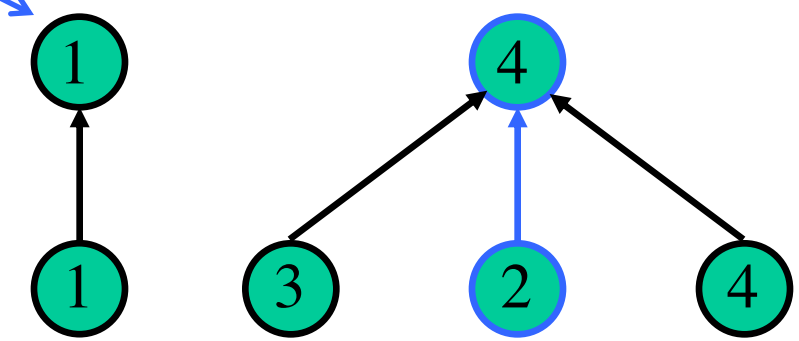
Sequenza di operazioni:

`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`

`union(4,2)`

`find(2)`

nome dell'insieme



# Realizzazione (1/3)

**classe** `QuickFindBilanciato` **implementa** `UnionFind`:
**dati**: $S(n) = O(n)$

una collezione di insiemi disgiunti di elementi $elem$; ogni insieme ha un nome $name$.

**operazioni**:

`makeSet`$(elem\ e)$ $T(n) = O(1)$

crea un nuovo albero, composto da due nodi: una radice ed un unico figlio (foglia). Memorizza $e$ sia nella radice che nella foglia dell'albero. Inizializza la cardinalità del nuovo insieme ad 1, assegnando il valore $\text{size}(x) = 1$ alla radice $x$.

# Realizzazione (2/3)

`find`$(elem\ e) \rightarrow name$ $\quad T(n) = O(1)$

accede alla foglia $x$ corrispondente all'elemento $e$. Da tale nodo segue il puntatore al padre, che è la radice dell'albero, e restituisce il nome memorizzato in tale radice.

# Realizzazione (3/3)

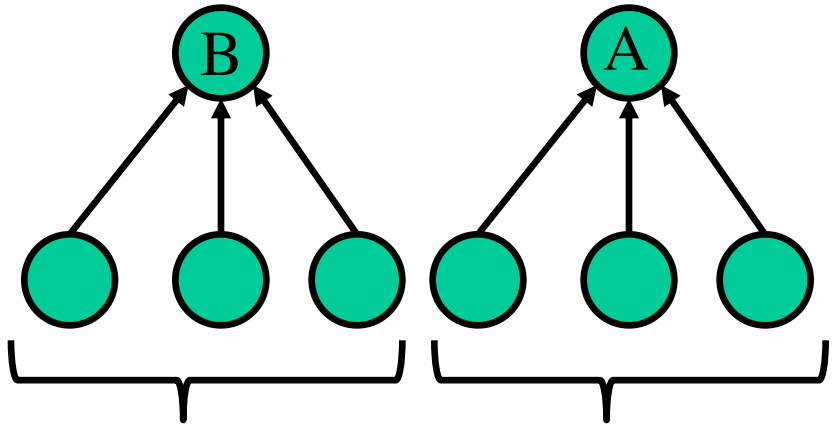
`union`$(name\ a, name\ b)$ $\qquad T_{am} = O(\log n)$

considera l'albero $A$ corrispondente all'insieme di nome $a$, e l'albero $B$ corrispondente all'insieme di nome $b$. Se $\text{size}(A) \geq \text{size}(B)$, muovi tutti i puntatori dalle foglie di $B$ alla radice di $A$, e cancella la vecchia radice di $B$. Altrimenti ($\text{size}(B) > \text{size}(A)$) memorizza nella radice di $B$ il nome $A$, muovi tutti i puntatori dalle foglie di $A$ alla radice di $B$, e cancella la vecchia radice di $A$. In entrambi i casi assegna al nuovo insieme la somma delle cardinalità dei due insiemi originali ($\text{size}(A) + \text{size}(B)$).

$T_{am}$ = tempo per operazione ammortizzato sull'intera sequenza di unioni (vedremo che una singola union può costare $\Theta(n)$, ma l'intera sequenza di n-1 union costa $O(n \log n)$)

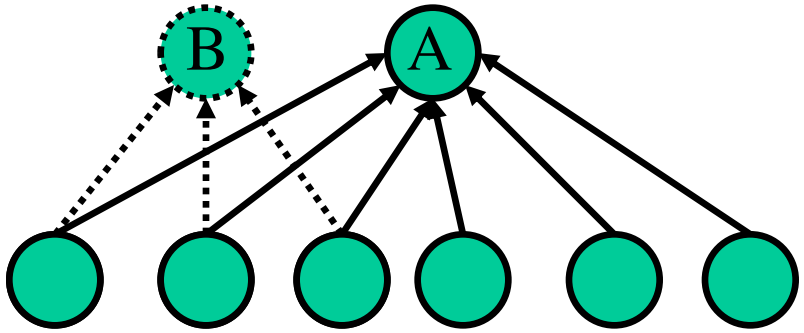
# complessità di un'operazione di Union

`Union(A,B)`




n/2 elementi

n/2 elementi




questa operazione costa $\Theta(n)$

domanda: quanto costa cambiare padre a un nodo?
…tempo costante!

domanda (cruciale): quante volte può cambiare padre un nodo?
…al più log n!

# Analisi ammortizzata (1/2)

Vogliamo dimostrare che se eseguiamo m find, n makeSet, e le al più n-1 union, il tempo richiesto dall'intera sequenza di operazioni è $O(m + n \log n)$

Idea della dimostrazione:

- È facile vedere che find e makeSet richiedono tempo $\Theta(m+n)$
- Per analizzare le operazioni di union, ci concentriamo su un singolo nodo e dimostriamo che il tempo speso per tale nodo è $O(\log n) \Rightarrow$ in totale, tempo speso è $O(n \log n)$

# Analisi ammortizzata (2/2)

- Quando eseguiamo una union, per ogni nodo che cambia padre pagheremo tempo costante
- Osserviamo ora che ogni nodo può cambiare al più O(log n) padri, poiché ogni volta che un nodo cambia padre la cardinalità dell'insieme al quale apparterrà è almeno doppia rispetto a quella dell'insieme cui apparteneva!
  - all'inizio un nodo è in un insieme di dimensione 1,
  - poi se cambia padre in un insieme di dimensione almeno 2,
  - all'i-esimo cambio è in un insieme di dimensione almeno $2^i$

$\Rightarrow$ il tempo speso per un singolo nodo sull'intera sequenza di n union è O(log n).

$\Rightarrow$ L'intera sequenza di operazioni costa

$$O(m+n+n \log n)=O(m+n \log n).$$