Algoritmi e Strutture Dati

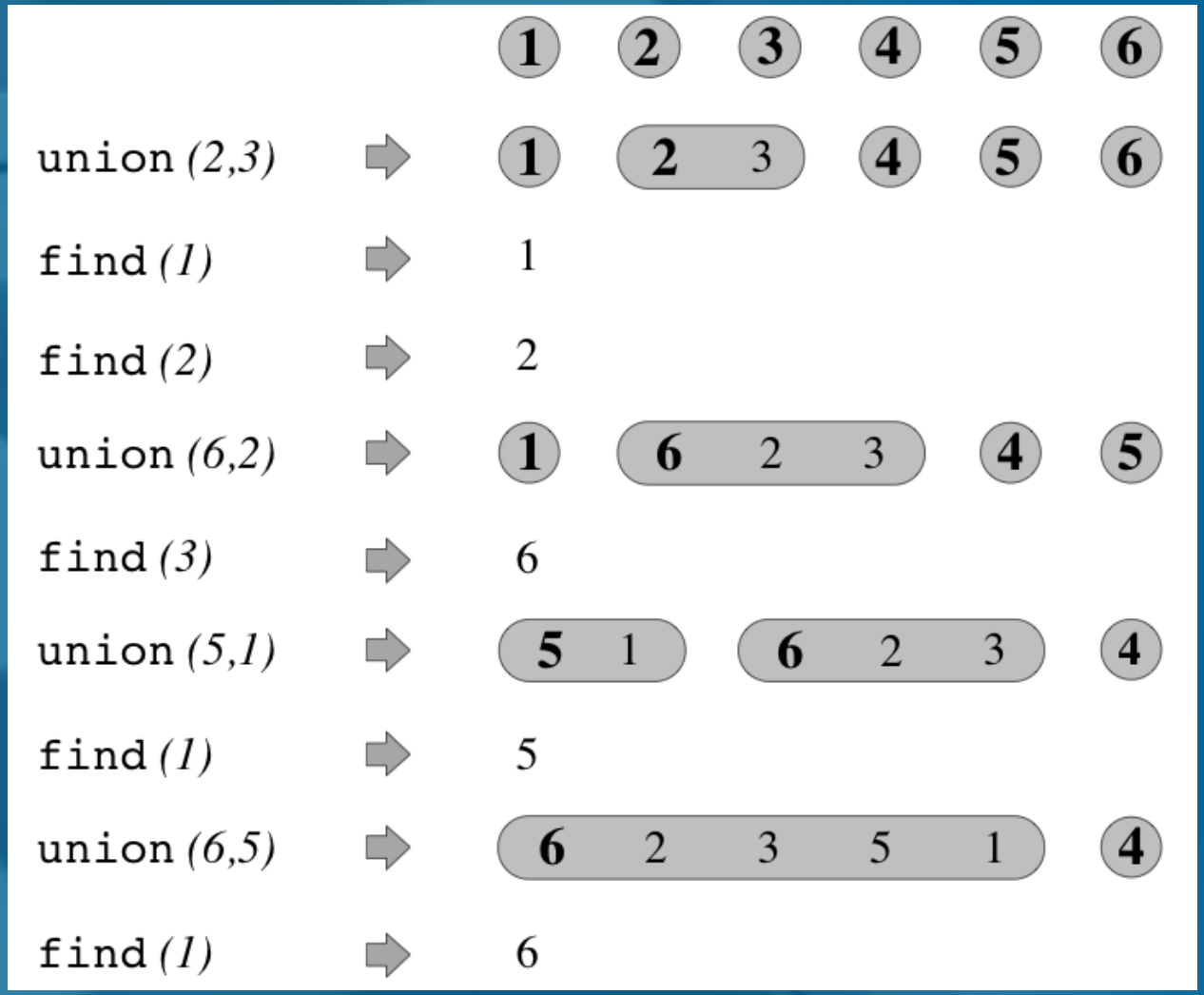
Capitolo 9

Il problema della gestione
di insiemi disgiunti (Union-find)

Il problema Union-find

- Mantenere una collezione di insiemi disgiunti contenenti elementi distinti (ad esempio, interi in 1…n) durante l'esecuzione di una sequenza 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)
- Applicazioni: algoritmo di Kruskal per la determinazione del minimo albero ricoprente di un grafo, calcolo degli minimi antenati comuni, ecc.

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)

Approcci elementari (basati su alberi)

Due strategie: QuickFind e QuickUnion

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)

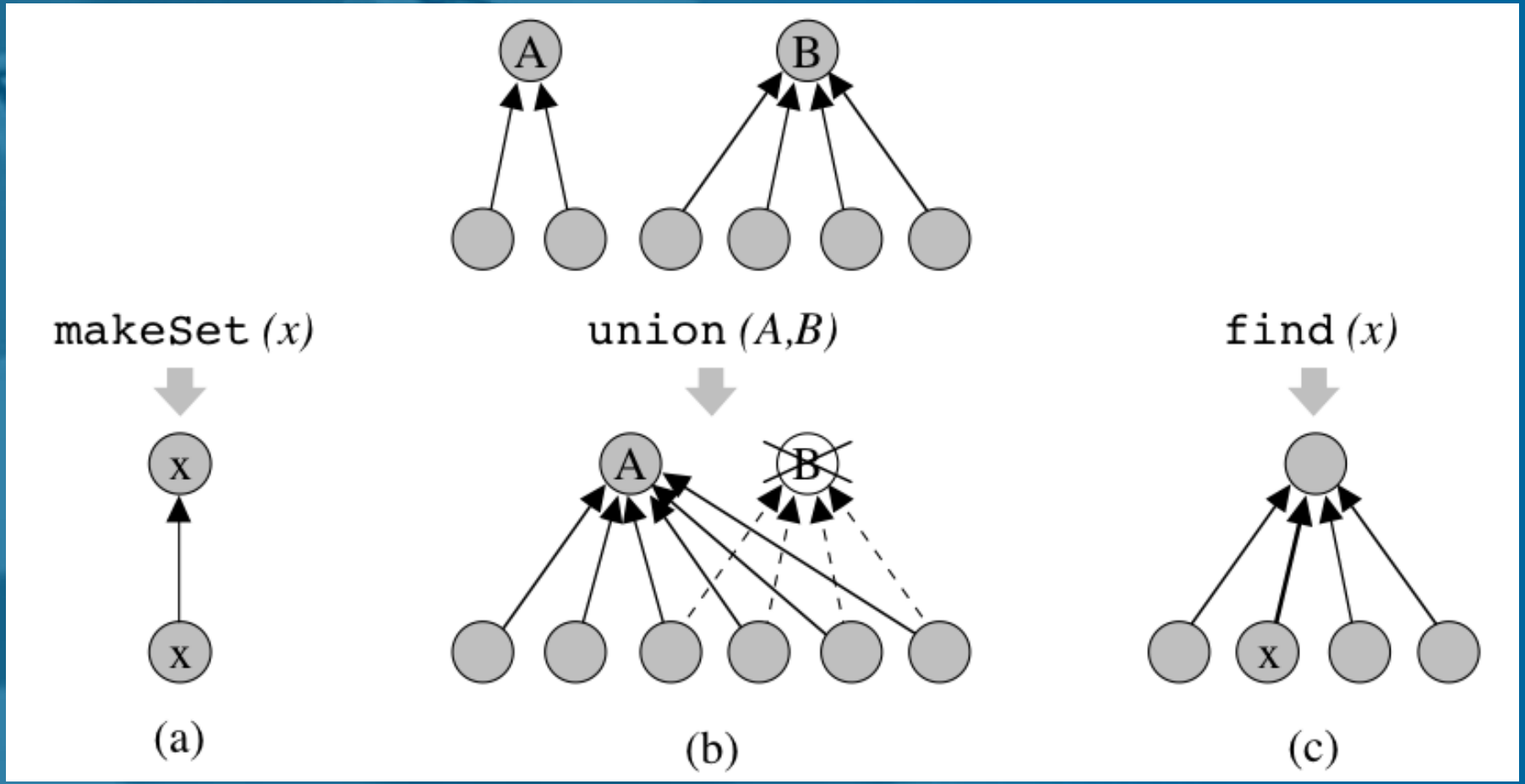
`union`$(name\ a, name\ b)$ $\qquad T(n) = O(n)$

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

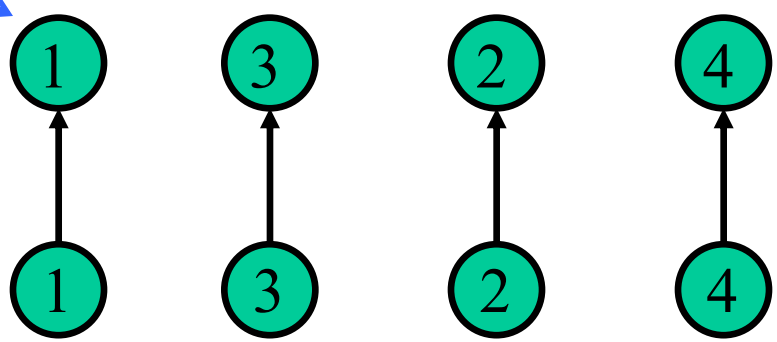


un esempio:

Sequenza di operazioni:

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

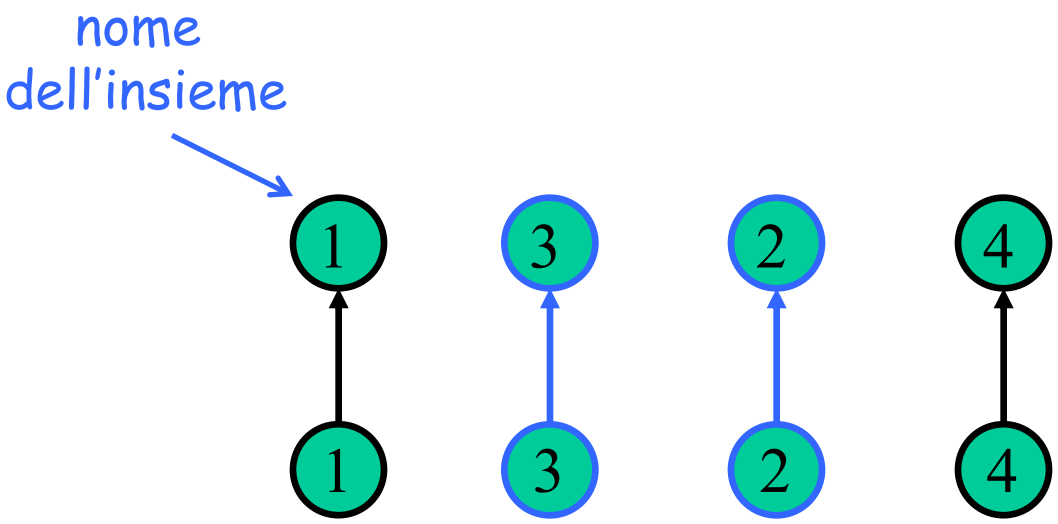
nome dell'insieme



un esempio:

Sequenza di operazioni:

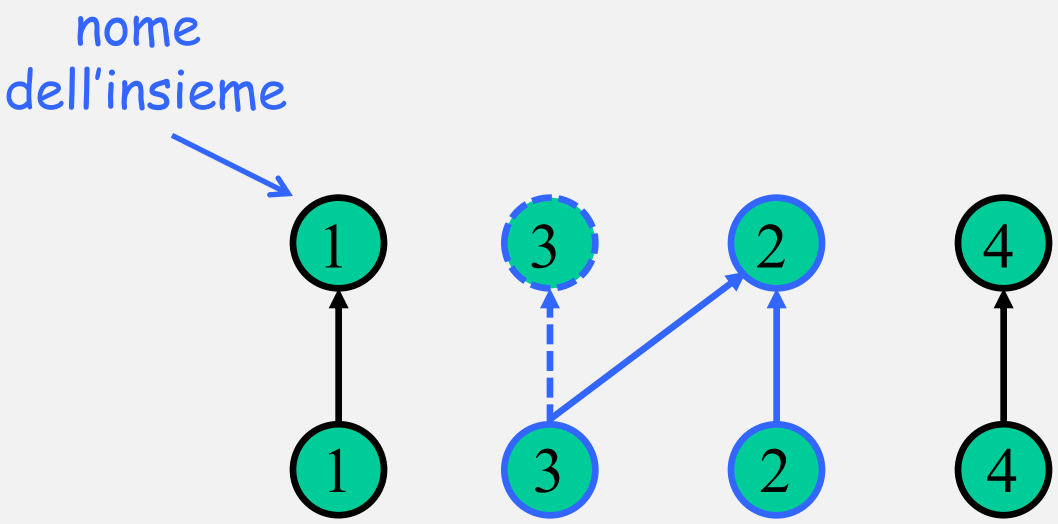
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

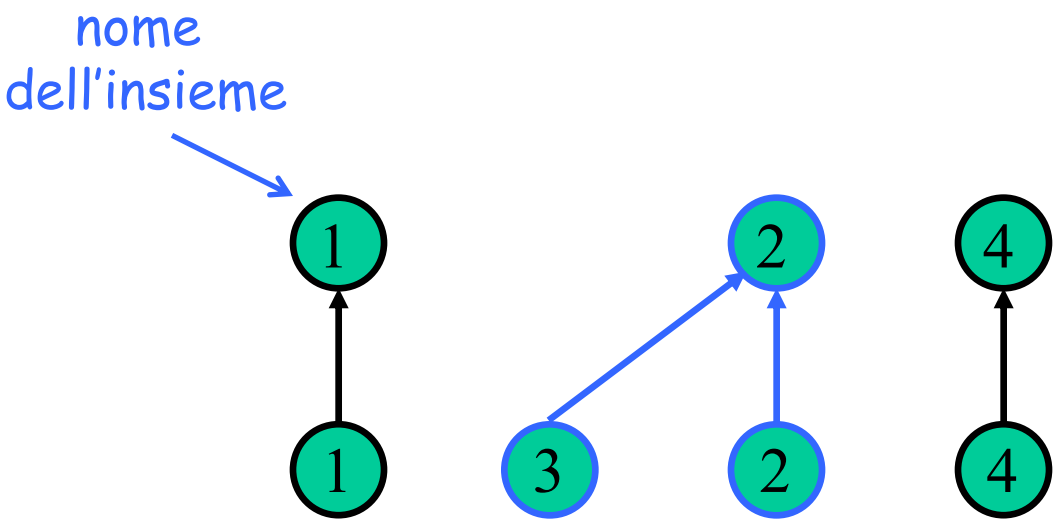
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

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



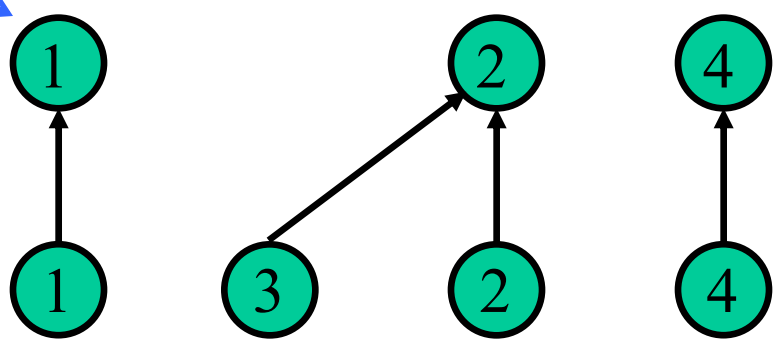
un esempio:

Sequenza di operazioni:

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

union(4,2)

nome dell'insieme

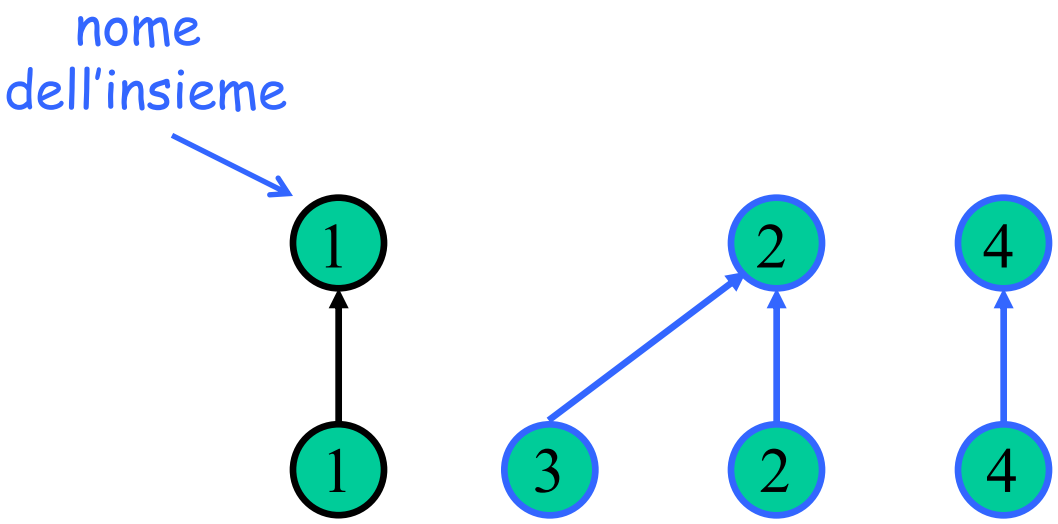


un esempio:

Sequenza di operazioni:

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

union(4,2)



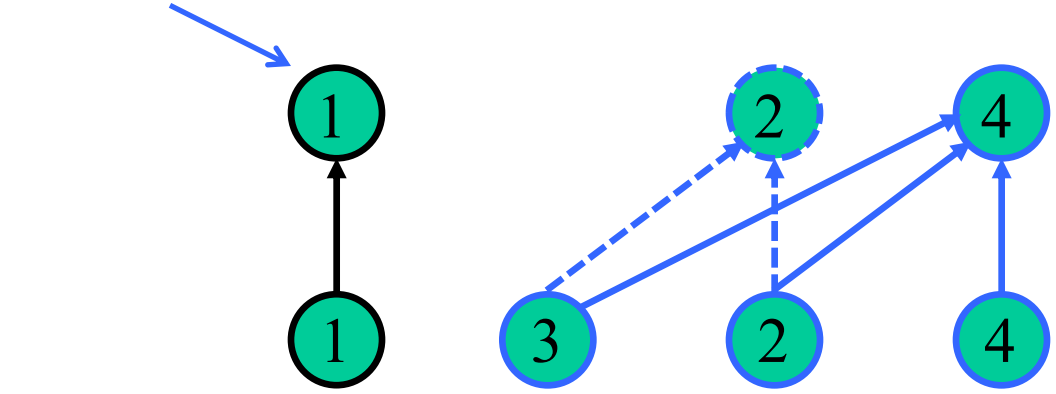
un esempio:

Sequenza di operazioni:

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

union(4,2)

nome dell'insieme

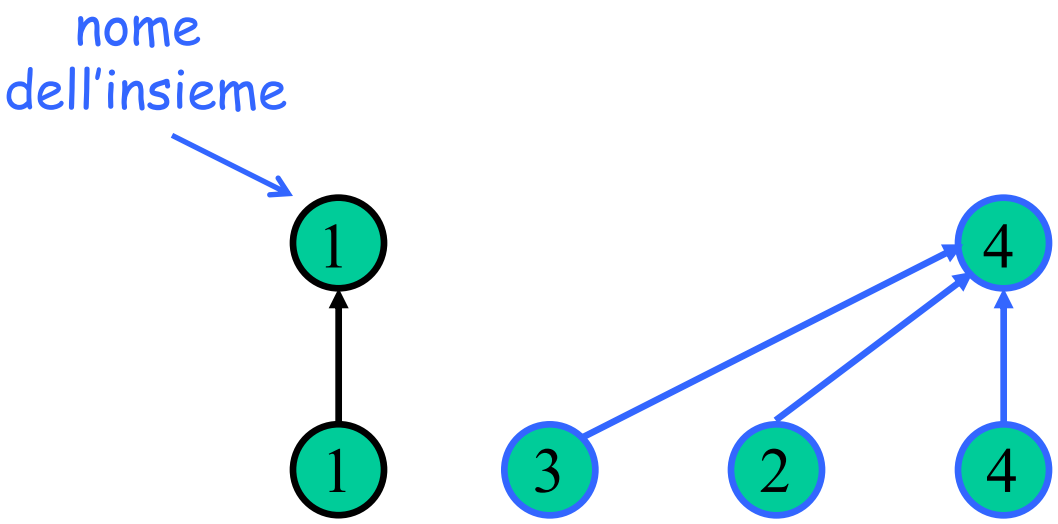


un esempio:

Sequenza di operazioni:

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

union(4,2)



un esempio:

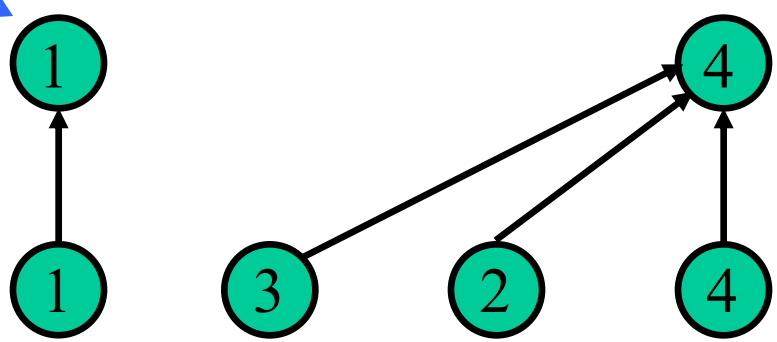
Sequenza di operazioni:

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

union(4,2)

find(2)

nome dell'insieme



un esempio:

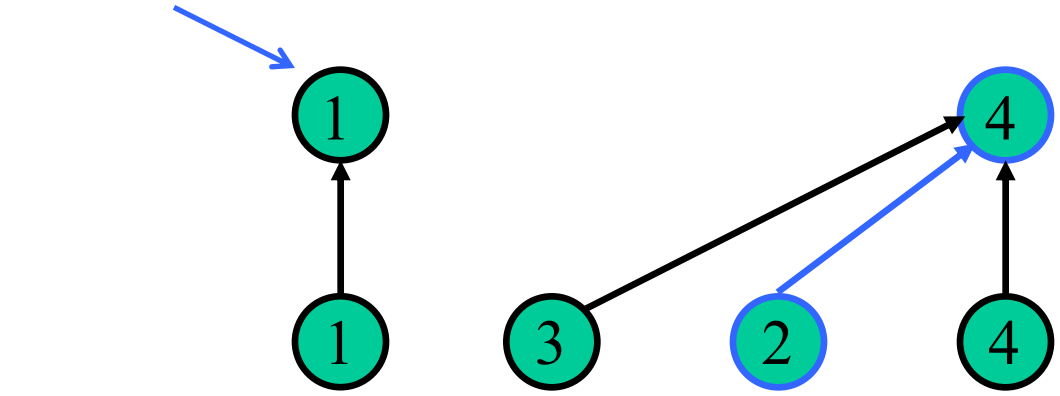
Sequenza di operazioni:

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

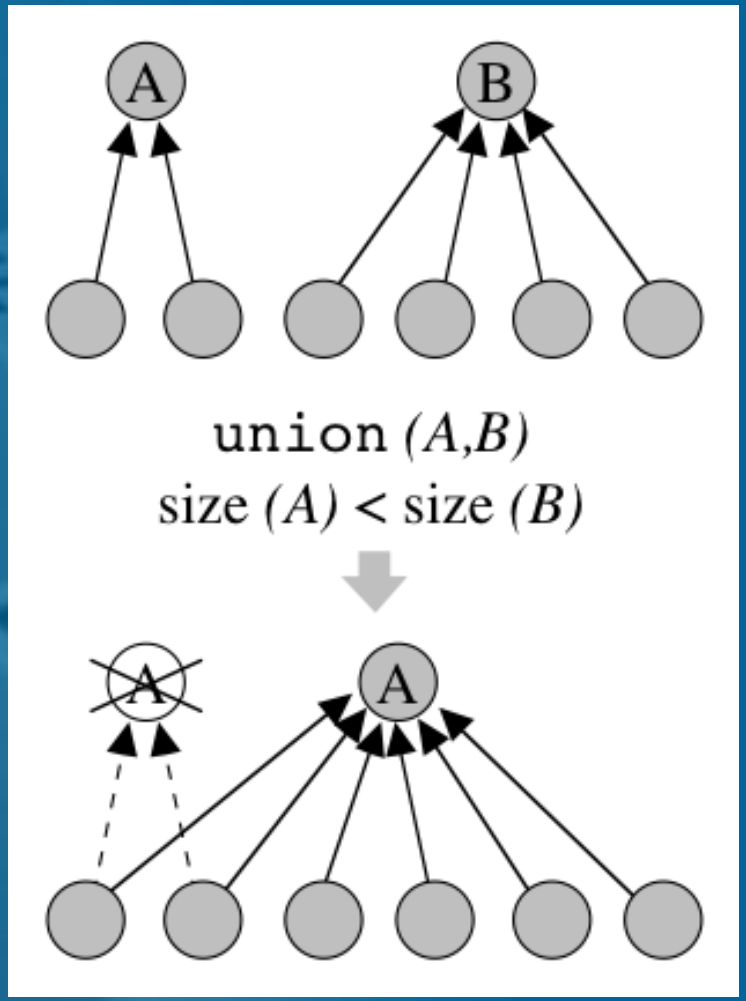
⇒ Se eseguiamo n makeSet, n-1 union come sopra, ed m find (in qualsiasi ordine), il tempo richiesto dall'intera sequenza di operazioni è $O(n+1+2+\ldots+(n-1)+m) = O(m+n^2)$

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

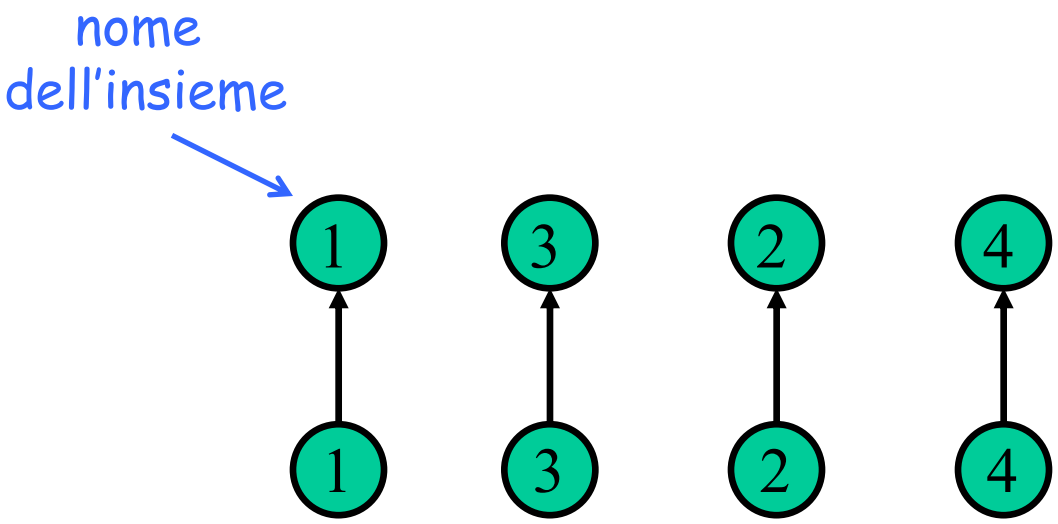
Nell'unione degli insiemi A e B, attacchiamo gli elementi dell'insieme di cardinalità minore a quello di cardinalità maggiore, e se necessario modifichiamo la radice dell'albero ottenuto (cosiddetta union by size)



un esempio:

Sequenza di operazioni:

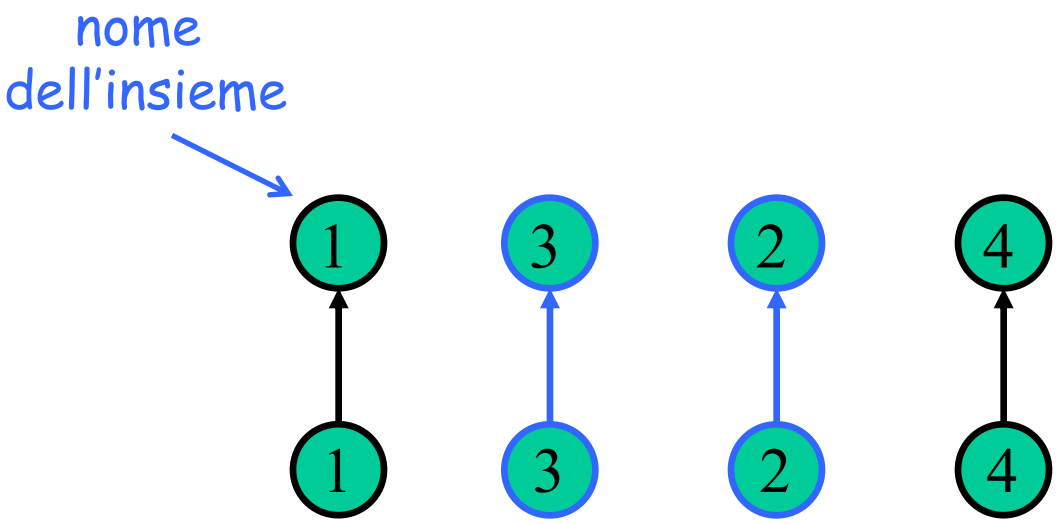
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

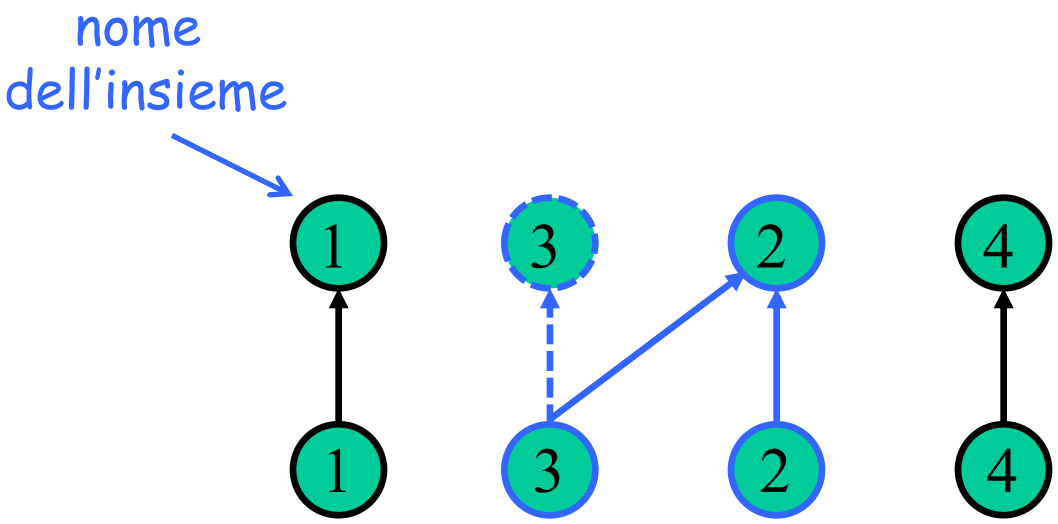
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

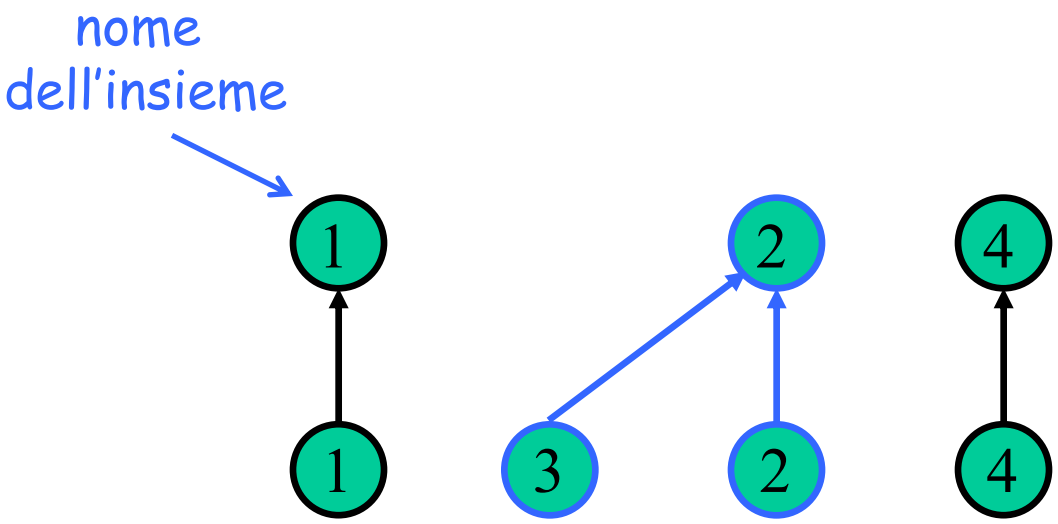
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

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



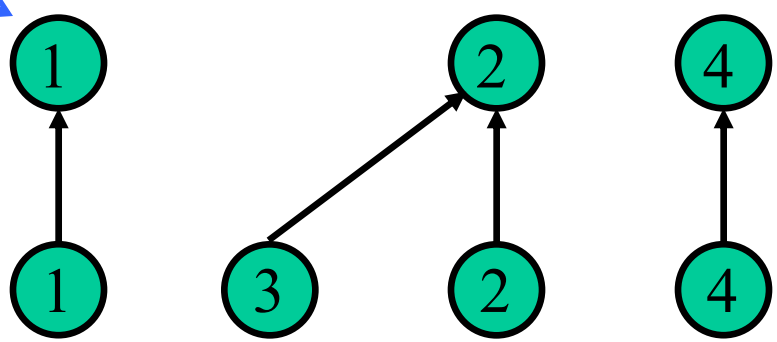
un esempio:

Sequenza di operazioni:

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

union(4,2)

nome dell'insieme

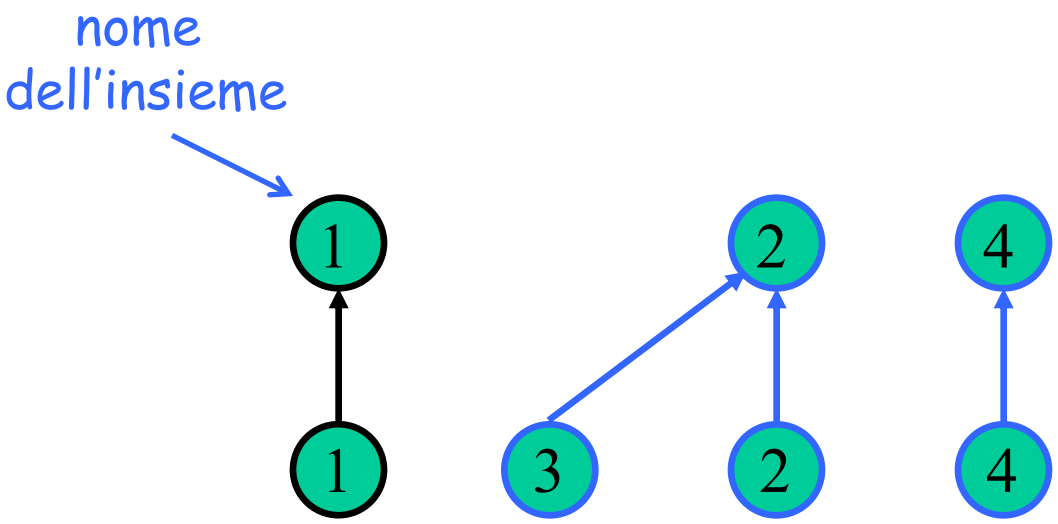


un esempio:

Sequenza di operazioni:

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

union(4,2)

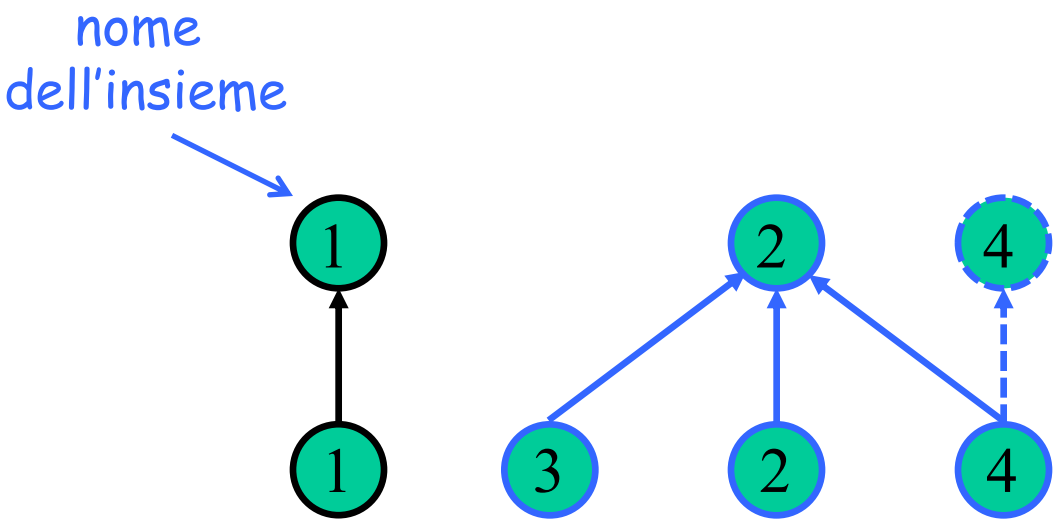


un esempio:

Sequenza di operazioni:

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

union(4,2)



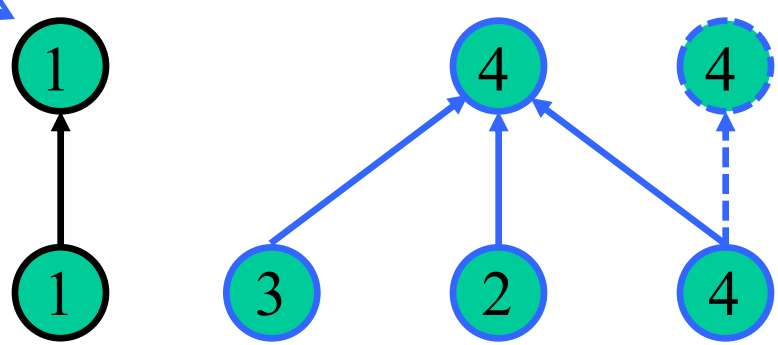
un esempio:

Sequenza di operazioni:

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

union(4,2)

nome dell'insieme



un esempio:

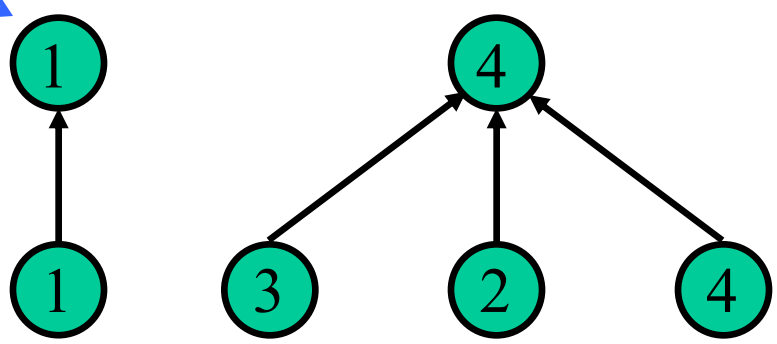
Sequenza di operazioni:

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

union(4,2)

find(2)

nome dell'insieme



un esempio:

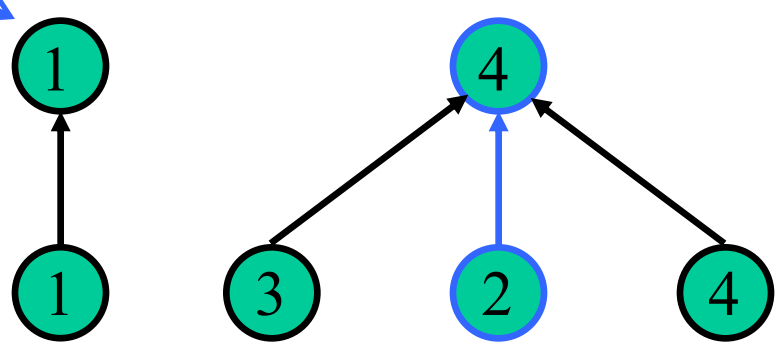
Sequenza di operazioni:

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)

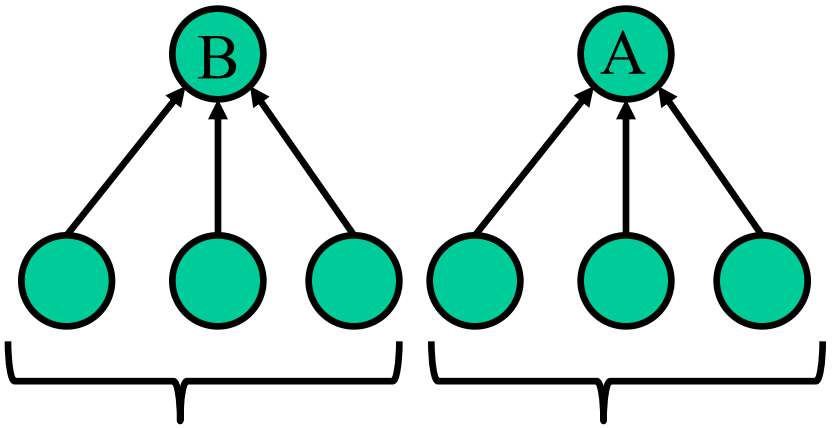
`union`$(name\ a, name\ b)$ $T_{am} = O(\log n)$

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 $O(n \log n)$)

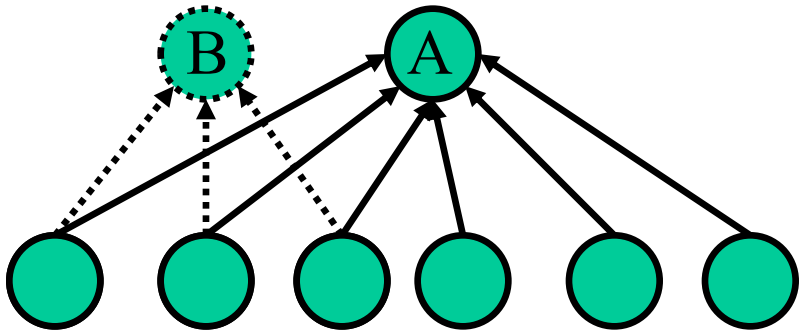
complessità di un'operazione di Union

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

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

Alberi QuickUnion

- Usiamo una foresta di alberi di altezza anche maggiore di 1 per rappresentare gli insiemi disgiunti. In ogni albero:
  - Radice = elemento rappresentativo dell'insieme
  - Rimanenti nodi = altri elementi (escluso l'elemento nella radice)

Realizzazione (1/2)

classe `QuickUnion` **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 un unico nodo x. Memorizza e in tale nodo, sia come valore che come nome del nodo.

Realizzazione (2/2)

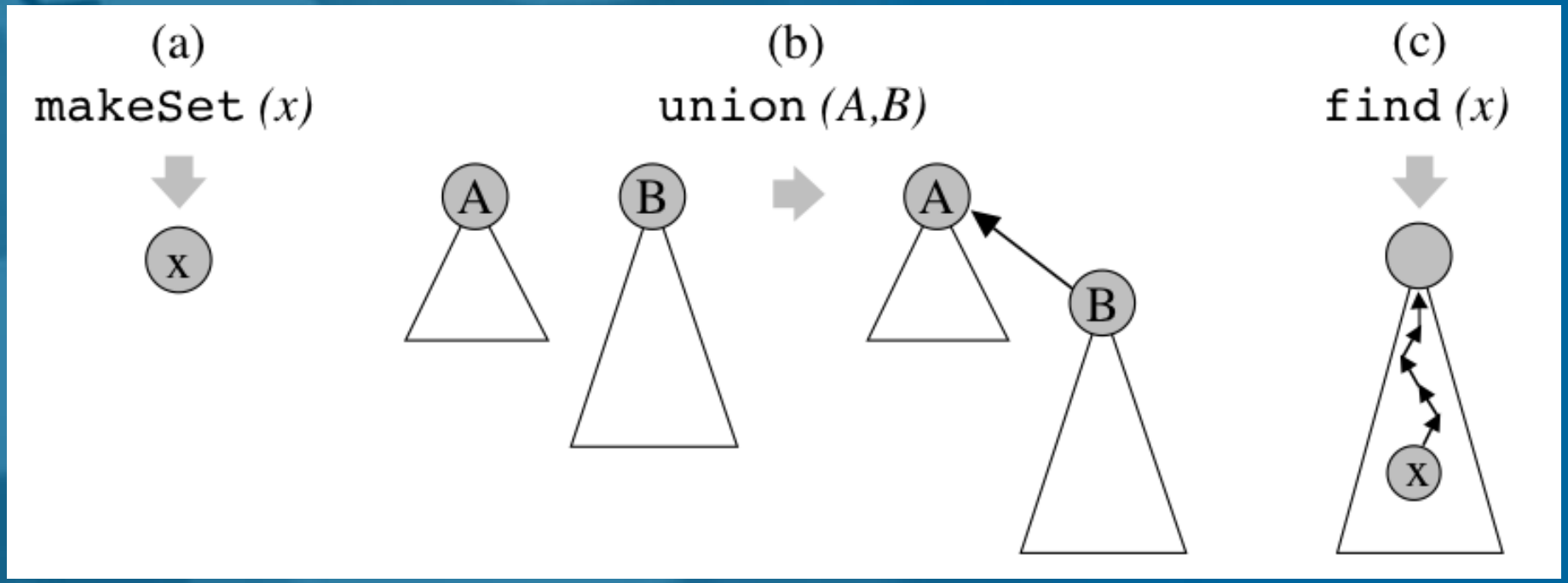
`union`$(name\ a, name\ b)$ $\qquad T(n) = O(1)$

considera l'albero A corrispondente all'insieme di nome a, e l'albero B corrispondente all'insieme di nome b. Rende la radice di B figlia della radice di A, introducendo un puntatore dalla radice di B alla radice di A.

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

accede al nodo x corrispondente all'elemento e. Partendo da tale nodo, segue ripetutamente i puntatori al padre fino a raggiungere la radice dell'albero. Restituisce il nome memorizzato in tale radice.

Esempio



un esempio:

Sequenza di operazioni:

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

nome dell'insieme e elemento

un esempio:

Sequenza di operazioni:

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

nome dell'insieme e elemento

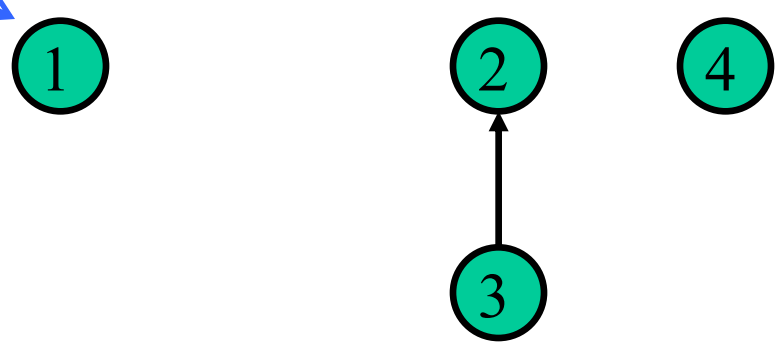
un esempio:

Sequenza di operazioni:

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

union(4,2)

nome dell'insieme e elemento



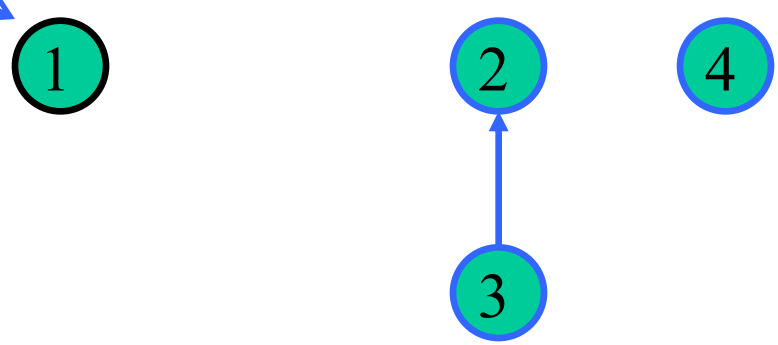
un esempio:

Sequenza di operazioni:

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

union(4,2)

nome dell'insieme e elemento



un esempio:

Sequenza di operazioni:

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

union(4,2)

union(4,1)

nome dell'insieme e elemento

un esempio:

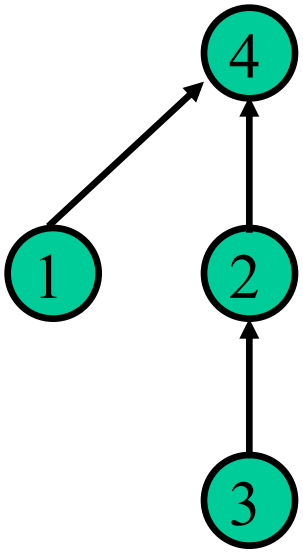
Sequenza di operazioni:

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

`union(4,2)`

`union(4,1)`

`find(3)`



un esempio:

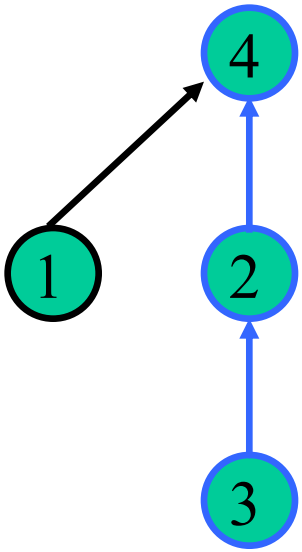
Sequenza di operazioni:

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

union(4,2)

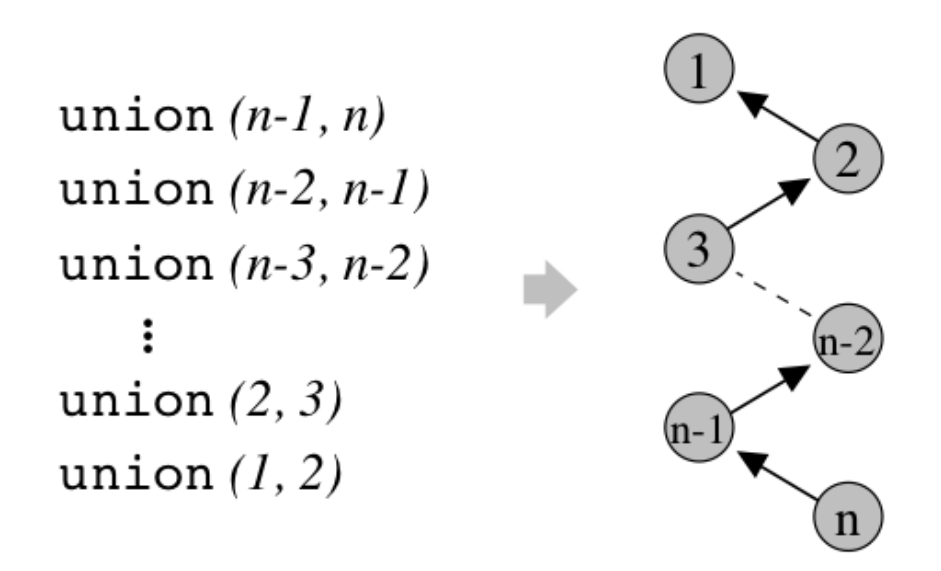
union(4,1)

find(3)



Find di costo lineare

union e makeSet richiedono solo tempo O(1), ma particolari sequenze di union possono generare un albero di altezza lineare, e quindi la find è molto inefficiente (costa n-1 nel caso peggiore)



⇒ Se eseguiamo n makeSet, n-1 union come sopra, seguite da m find, il tempo richiesto dall'intera sequenza di operazioni è O(n+n-1+mn)=O(mn)

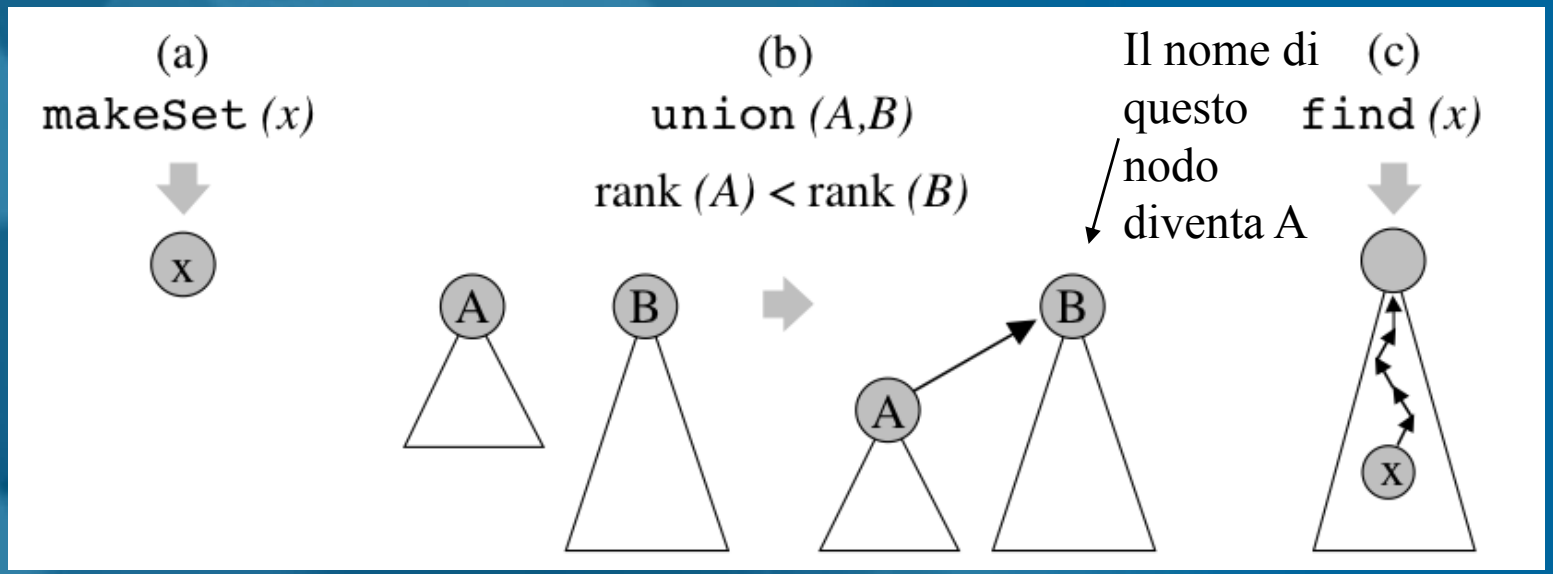
Migliorare la struttura QuickUnion: euristica *union by rank*

Idea: fare in modo che per ogni insieme l'albero corrispondente abbia altezza piccola.

Bilanciamento in alberi QuickUnion

Union by rank (o by height): nell'unione degli insiemi A e B, rendiamo la radice dell'albero più basso figlia della radice dell'albero più alto

rank(x) = altezza dell'albero di cui x è radice



Realizzazione (1/3)

classe `QuickUnionBilanciato` **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 un unico nodo x. Memorizza e sia come valore che come nome in tale nodo. Inizializza $\text{rank}(x) = 0$ (l'altezza del nuovo albero è 0), memorizzando nel nodo x anche tale valore di rank.

Realizzazione (2/3)

`union`$(name\ a, name\ b)$ $\qquad T(n) = O(1)$

considera l'albero A corrispondente all'insieme di nome a, e l'albero B corrispondente all'insieme di nome b. Confronta $\mathrm{rank}(A)$ e $\mathrm{rank}(B)$, distinguendo tre casi.

1. Se $\mathrm{rank}(B) < \mathrm{rank}(A)$, rende la radice dell'albero B figlia della radice dell'albero A.
2. Se $\mathrm{rank}(A) < \mathrm{rank}(B)$, rende la radice dell'albero A figlia della radice dell'albero B, e memorizza A come nome nella radice del nuovo albero.
3. Se $\mathrm{rank}(A) = \mathrm{rank}(B)$, rende la radice dell'albero B figlia della radice dell'albero A, ed aggiorna $\mathrm{rank}(A) = \mathrm{rank}(A) + 1$.

Realizzazione (3/3)

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

accede al nodo x corrispondente all'elemento e. Partendo da tale nodo, segue ripetutamente i puntatori al padre fino a raggiungere la radice dell'albero. Restituisce il nome memorizzato in tale radice.

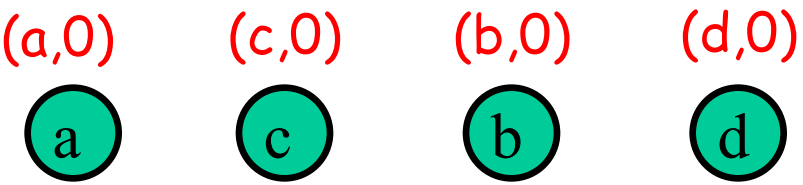
un esempio:

Sequenza di operazioni:

makeSet(a) makeSet(c) makeSet(b) makeSet(d) union(b,d)

(nome,rank)

informazioni mantenute nella radice dell'albero



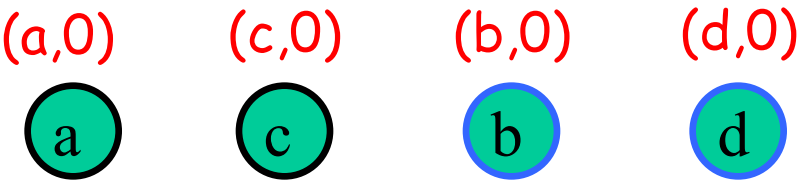
un esempio:

Sequenza di operazioni:

makeSet(a) makeSet(c) makeSet(b) makeSet(d) union(b,d)

(nome,rank)

informazioni mantenute nella radice dell'albero



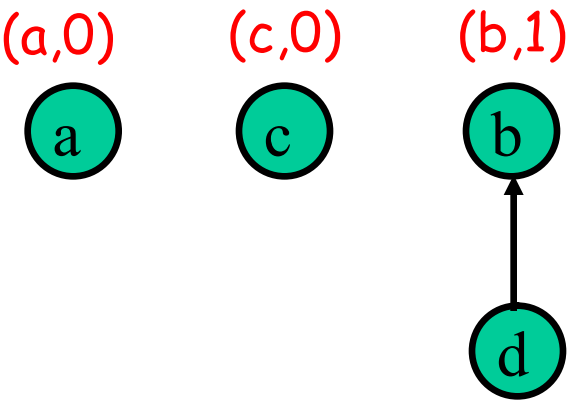
un esempio:

Sequenza di operazioni:

makeSet(a) makeSet(c) makeSet(b) makeSet(d) union(b,d)
union(a,c)

(nome,rank)

informazioni mantenute nella radice dell'albero



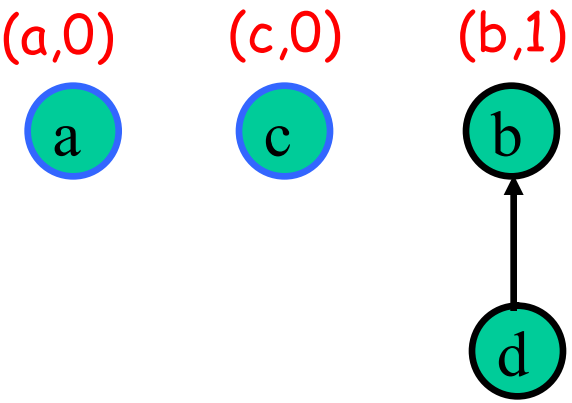
un esempio:

Sequenza di operazioni:

makeSet(a) makeSet(c) makeSet(b) makeSet(d) union(b,d)
union(a,c)

(nome,rank)

informazioni mantenute nella radice dell'albero



un esempio:

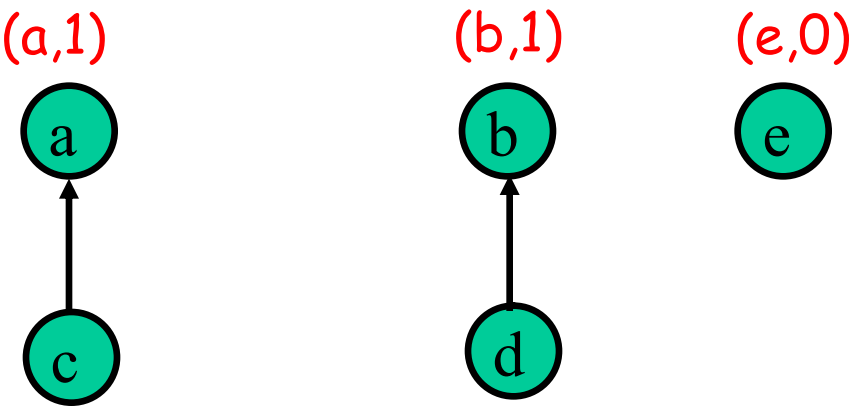
Sequenza di operazioni:

makeSet(a) makeSet(c) makeSet(b) makeSet(d) union(b,d)

union(a,c) makeSet(e) union(e,b)

(nome,rank)

informazioni mantenute nella radice dell'albero



un esempio:

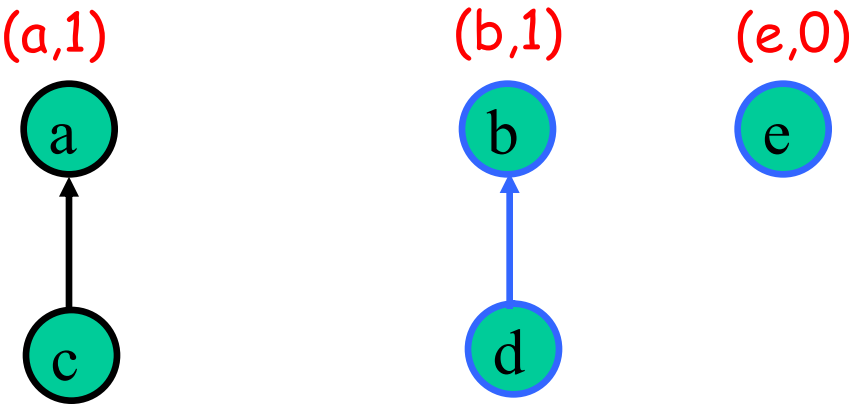
Sequenza di operazioni:

makeSet(a) makeSet(c) makeSet(b) makeSet(d) union(b,d)

union(a,c) makeSet(e) union(e,b)

(nome,rank)

informazioni mantenute nella radice dell'albero



un esempio:

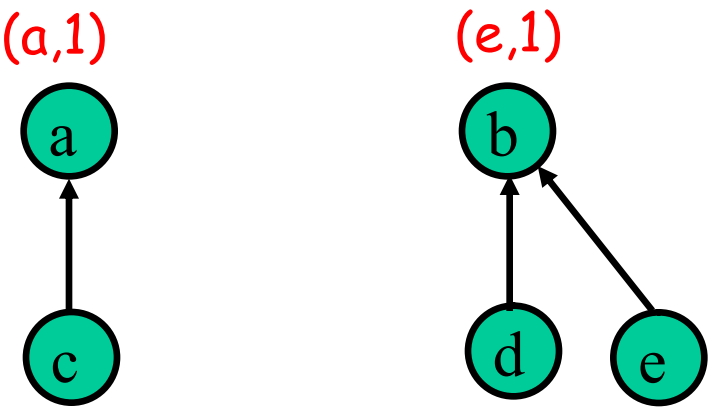
Sequenza di operazioni:

makeSet(a) makeSet(c) makeSet(b) makeSet(d) union(b,d)

union(a,c) makeSet(e) union(e,b) union(a,e)

(nome,rank)

informazioni mantenute nella radice dell'albero



un esempio:

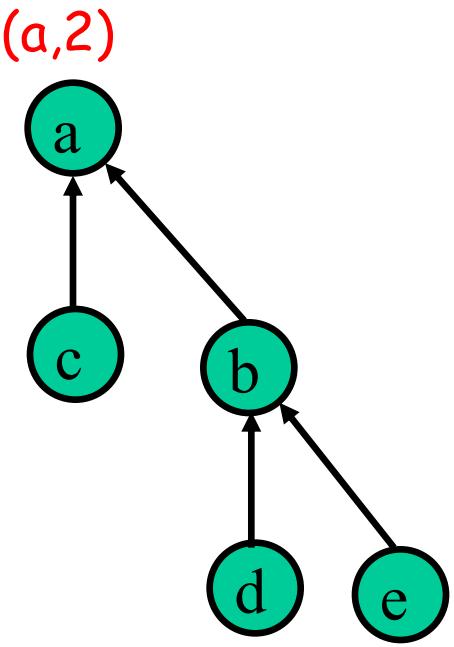
Sequenza di operazioni:

makeSet(a) makeSet(c) makeSet(b) makeSet(d) union(b,d)

union(a,c) makeSet(e) union(e,b) union(a,e)

(nome,rank)

informazioni mantenute nella radice dell'albero



Complessità computazionale

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(n+m log n)

Idea della dimostrazione:

- È facile vedere che union e makeSet richiedono tempo O(n)
- Per analizzare il costo delle operazioni di find, dimostreremo che l'altezza degli alberi si mantiene logaritmica nel numero di elementi contenuti in un albero

Conseguenza del bilanciamento

Lemma: Con la union by rank, un albero QuickUnion con radice x ha almeno $2^{rank(x)}$ nodi.

Dim: per induzione sulla lunghezza della sequenza di union che produce un albero.

Passo base: albero prodotto da una sequenza di union di lunghezza 0, ovvero un albero iniziale: esso ha altezza 0, e la tesi è banalmente vera.

Passo induttivo: Consideriamo un albero ottenuto eseguendo una sequenza di k operazioni di union, l'ultima delle quali sia union(A,B). A e B sono ottenuti con sequenze di union di lunghezza < k, e quindi per hp induttiva $|A| \geq 2^{rank(A)}$ e $|B| \geq 2^{rank(B)}$

- Se rank(A) > rank(B), allora:

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

- Se rank(A) < rank(B): simmetrico
- Se rank(A) = rank(B):

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

Analisi (nel caso peggiore)

Per la proprietà precedente, l'altezza di un albero QuickUnion bilanciato è limitata superiormente da log n, con n = numero di makeSet

L'operazione find richiede tempo O(log n)

L'intera sequenza di operazioni costa O(n+m log n).

Un altro bilanciamento per QuickUnion

Union by size: associamo ad ogni radice x un valore size(x) che è pari al numero di elementi contenuti nell'insieme (albero); quindi, nell'unione degli insiemi A e B, rendiamo sempre la radice dell'albero con meno nodi figlia della radice dell'albero con più nodi.

Stesse prestazioni di union by rank!

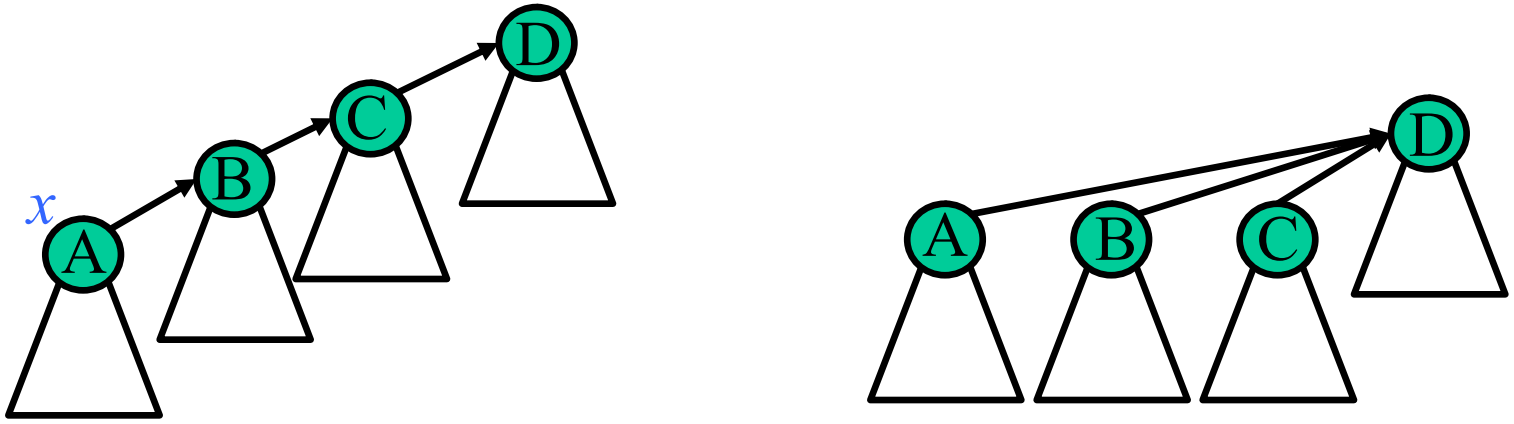
Riepilogo sul bilanciamento

	makeSet	union	find
QuickFind	$O(1)$	$O(n)$	$O(1)$
QuickFindBilanciato	$O(1)$	$O(\log n)$ amm.	$O(1)$
QuickUnion	$O(1)$	$O(1)$	$O(n)$
QuickUnionBilanciatoRank	$O(1)$	$O(1)$	$O(\log n)$
QuickUnionBilanciatoSize	$O(1)$	$O(1)$	$O(\log n)$

Approfondimenti

1. Risolvere il problema union-find usando strutture dati elementari (vettori e liste lineari), e valutarne la complessità computazionale.
2. Dimostrare che in QuickUnion, la union by size fornisce le stesse prestazioni della union by rank. (Suggerimento: induzione sul fatto che l'altezza di un albero è al più logaritmica nel numero di elementi contenuti).
3. Quando è preferibile un approccio di tipo QuickFind con union by size rispetto ad un approccio di tipo QuickUnion con union by rank?

Un'ulteriore euristica: compressione dei cammini



Idea: quando eseguo find(x) e attraverso il cammino da x alla radice, comprimo il cammino, ovvero rendo tutti i nodi del cammino figli della radice

Intuizione: find(x) ha un costo ancora lineare nella lunghezza del cammino attraversato, ma prossime find costeranno di meno

Teorema (Tarjan&van Leeuwen)

Usando in QuickUnion le euristiche di union by rank (o by size) e compressione dei cammini, una qualsiasi sequenza di n makeSet, n-1 union e m find hanno un costo di $O(n+m\ \alpha(n+m,n))$.

$\alpha(x,y)$: funzione inversa della funzione di Ackermann

La funzione di Ackermann A(i,j) e la sua inversa $\alpha(m,n)$

Notazione: con a^{b^c} intendiamo $a^{(b^c)}$, e non $(a^b)^c = a^{b \cdot c}$.
per interi $i,j \geq 1$, definiamo $A(i,j)$ come:

$$A(1,j) = 2^j \qquad j \geq 1;$$

$$A(i,1) = A(i-1,2) \qquad i \geq 2;$$

$$A(i,j) = A(i-1, A(i,j-1)) \qquad i,j \geq 2.$$

A(i,j) per piccoli valori di i e j

	j=1	j=2	j=3	j=4
i=1	2	2^2	2^3	2^4
i=2	2^2	$2^{2^2} = 2^4$	$2^{2^{2^2}} = 2^{16}$	$2^{2^{2^{2^2}}} = 2^{65536}$
i=3	2^{2^2}	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{16}$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{2^{2^{\cdot^{\cdot^{\cdot^2}}}}\ 16}$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{2^{2^{\cdot^{\cdot^{\cdot^2}}}}\ \underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{16}}$

La funzione $\alpha(m,n)$

Per interi $m \geq n \geq 0$, definiamo $\alpha(m,n)$ come:

$$\alpha(m,n) = \min\{i \geq 1 |\ A(i, \lfloor m/n \rfloor) > \log_2 n\}.$$

Proprietà di $\alpha(m,n)$

1. per n fissato, $\alpha(m,n)$ è monotonicamente decrescente al crescere di m

$$\alpha(m,n) = \min \{i>0 : \underbrace{A(i, \lfloor m/n \rfloor)}_{\text{crescente in } m} > \log_2 n\}$$

2. $\alpha(n,n) \to \infty$ for $n \to \infty$

$$\alpha(n,n) = \min \{i>0 : A(i, \lfloor n/n \rfloor) > \log_2 n\}$$
$$= \min \{i>0 : A(i, 1) > \underbrace{\log_2 n}_{\to \infty}\}$$

Osservazione

$\alpha(m,n) \leq 4$ per ogni scopo pratico
(ovvero, per valori ragionevoli di n)

$$\alpha(m,n) = \min \{i>0 : A(i, \lfloor m/n \rfloor) > \log_2 n\}$$

$$A(4,\lfloor m/n \rfloor) \geq A(4,1) = A(3,2)$$

$$= 2^{2^{\cdot^{\cdot^{2}}}} \Big\} 16 \gg 10^{80}$$

$\cong$ numero stimato di atomi nell'universo!

$\Rightarrow$ hence, $\alpha(m,n) \leq 4$ for any $n < 2^{10^{80}}$

Ultime proprietà: densità di m/n

1. $\alpha(m,n) \leq 1$ quando $\lfloor m/n \rfloor > \log_2 \log_2 n$

2. $\alpha(m,n) \leq 2$ quando $\lfloor m/n \rfloor > \log_2 \log^* n$

dove

$\log^{(1)} n = \log n$
$\log^{(i)} n = \log \log^{(i-1)} n$

$\log^* n = \min \{i>0 : \log^{(i)} n \leq 1\}$

riuscite a dimostrarle?