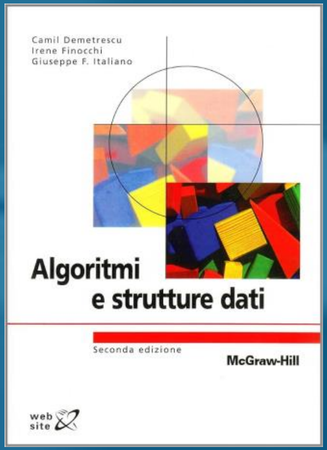
Il problema della gestione di insiemi disgiunti (Union-find)



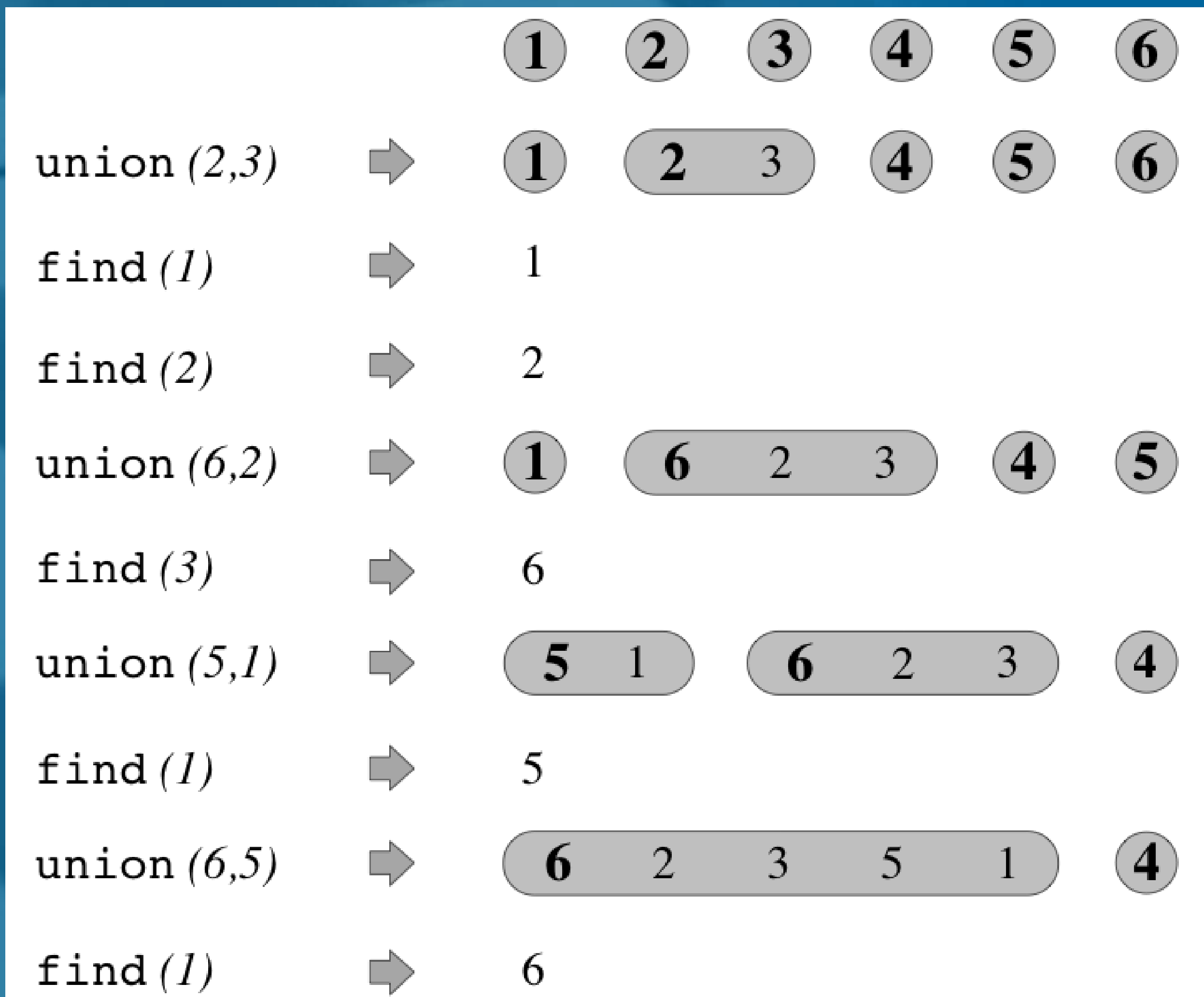
Capitolo 9

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}di nome 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: $\boxed{\text{QuickFind}}$ 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)

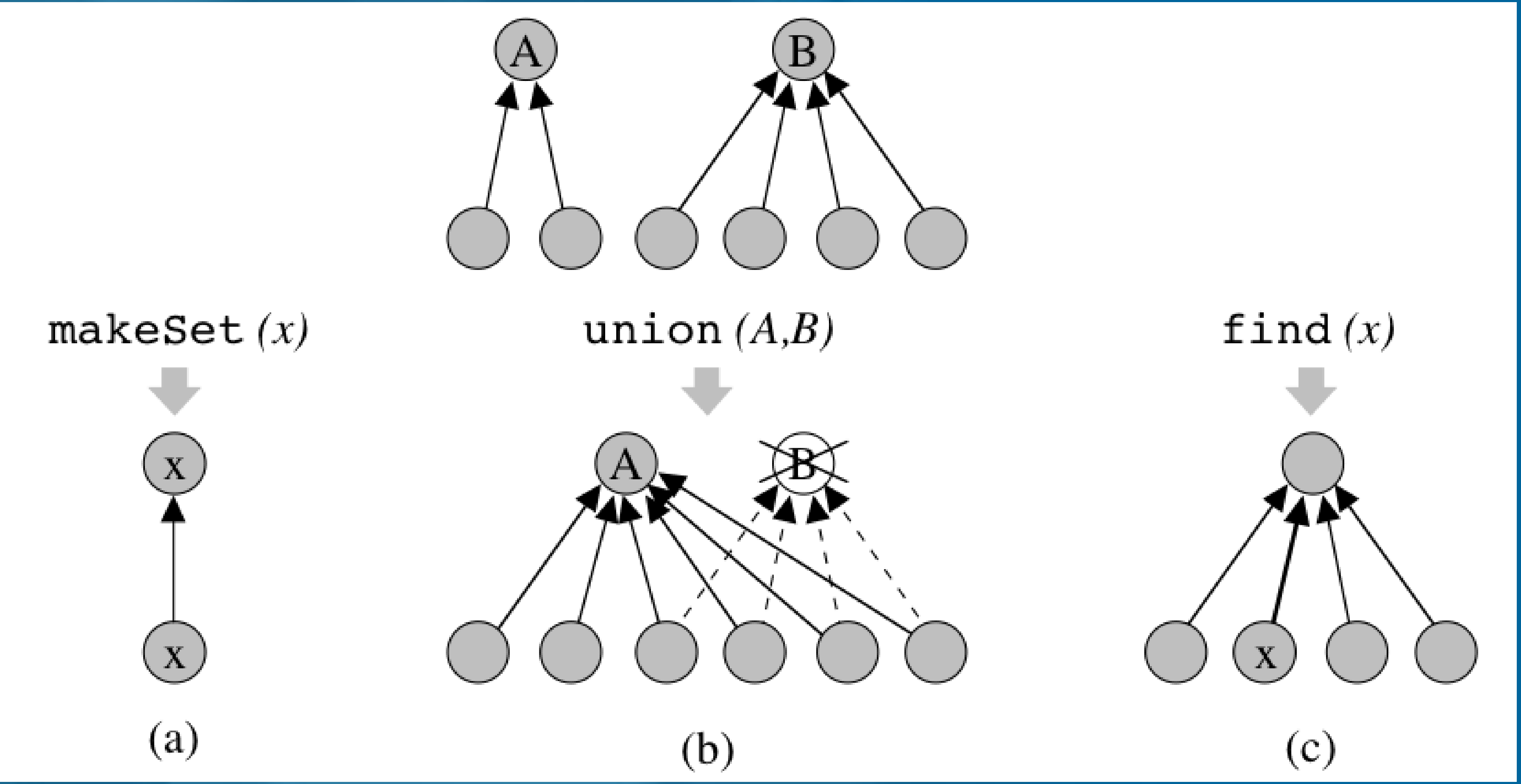
$\texttt{union}(name\ a, name\ b)$ $\qquad T(n) = O(n)$

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.

$\texttt{find}(elem\ e) \rightarrow name$ $\qquad T(n) = O(1)$

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.

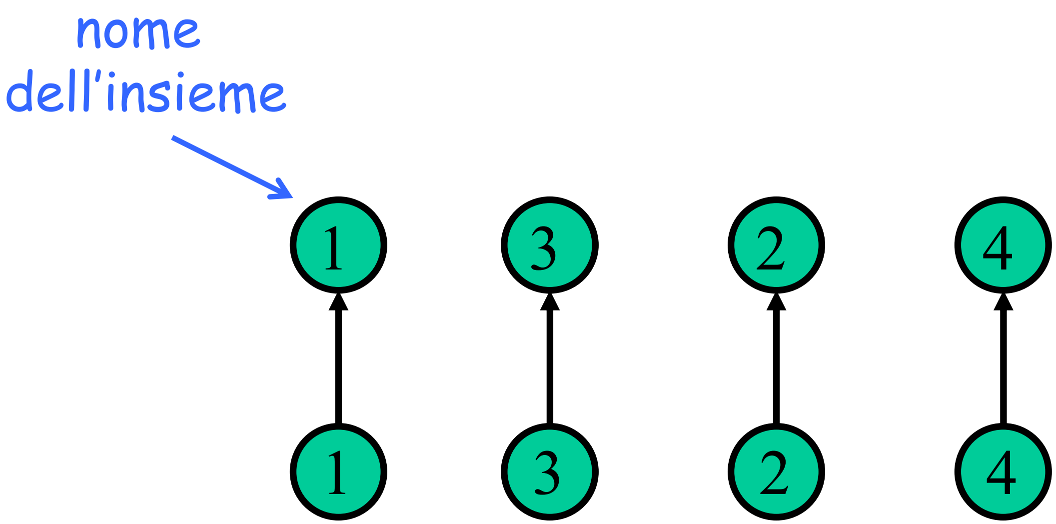
Esempio



un esempio:

Sequenza di operazioni:

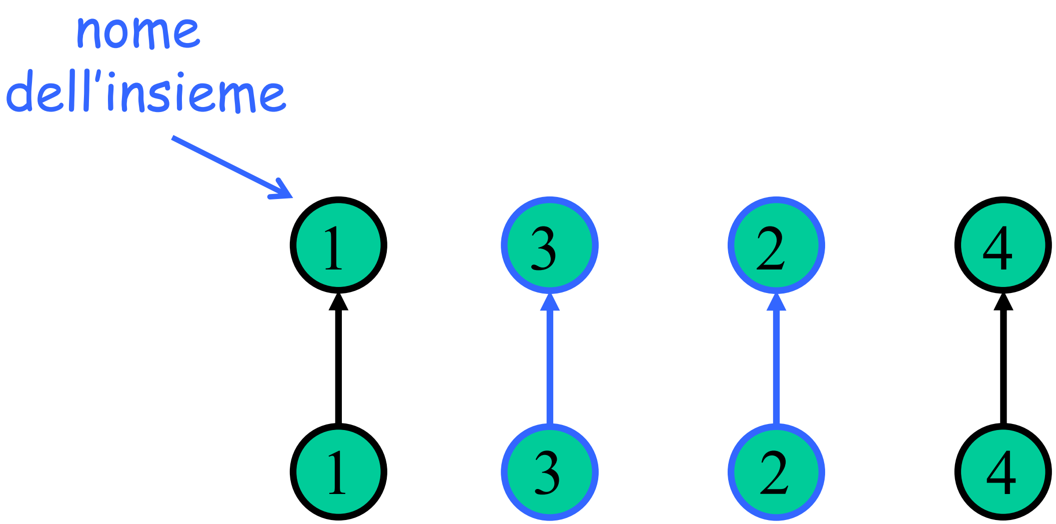
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

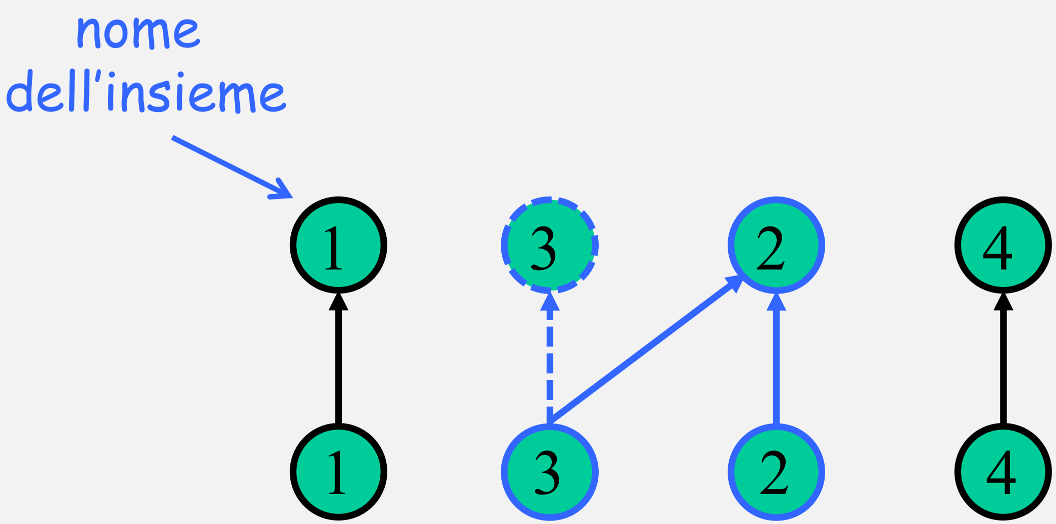
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

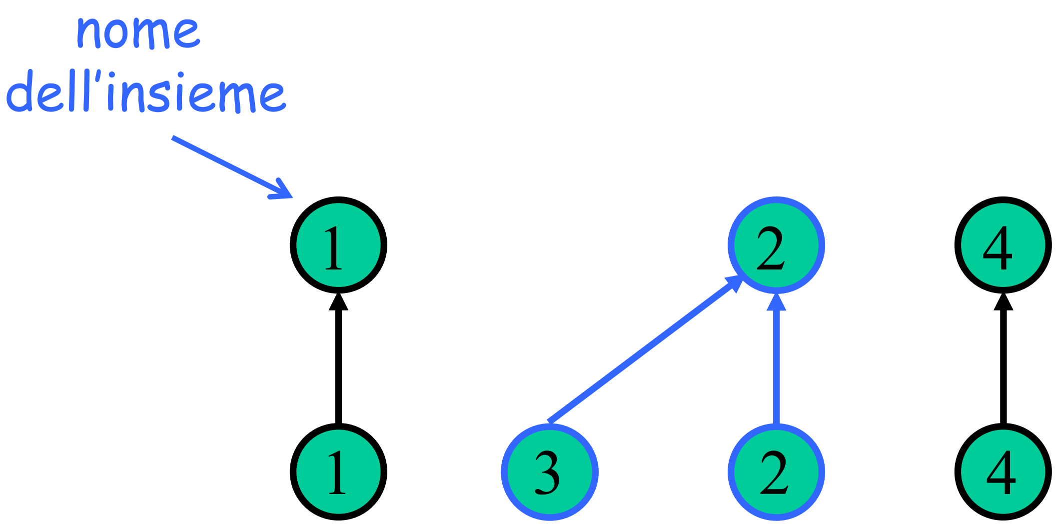
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

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

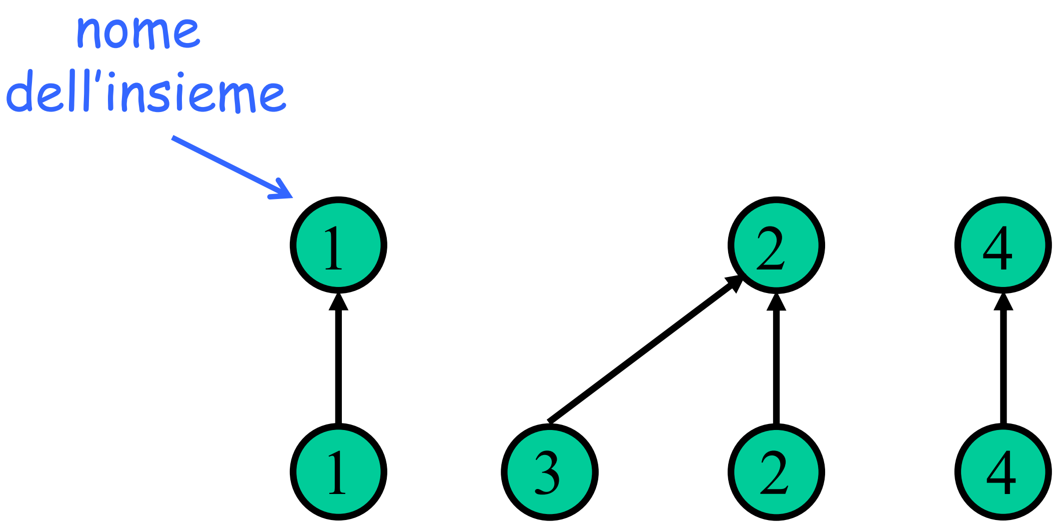


un esempio:

Sequenza di operazioni:

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

union(4,2)

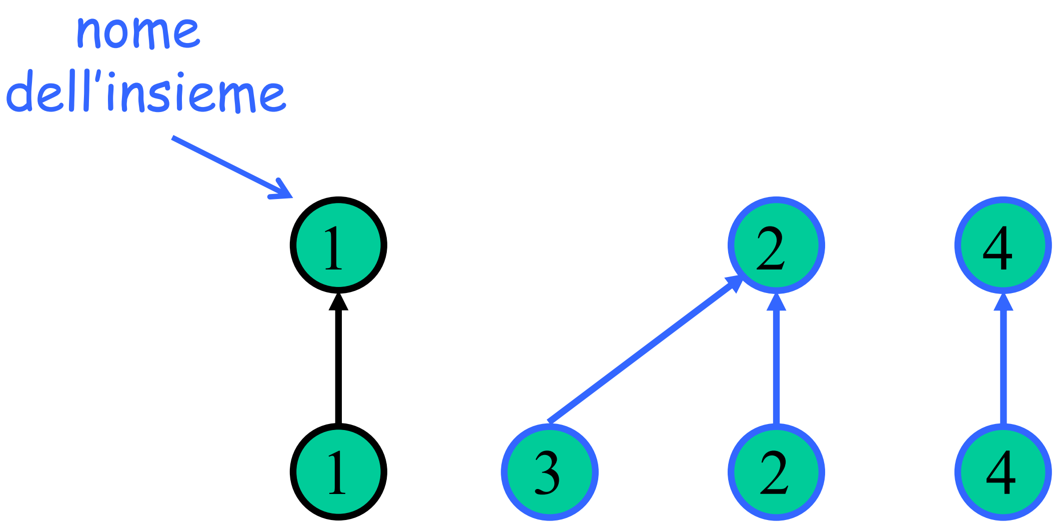


un esempio:

Sequenza di operazioni:

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

`union`(4,2)

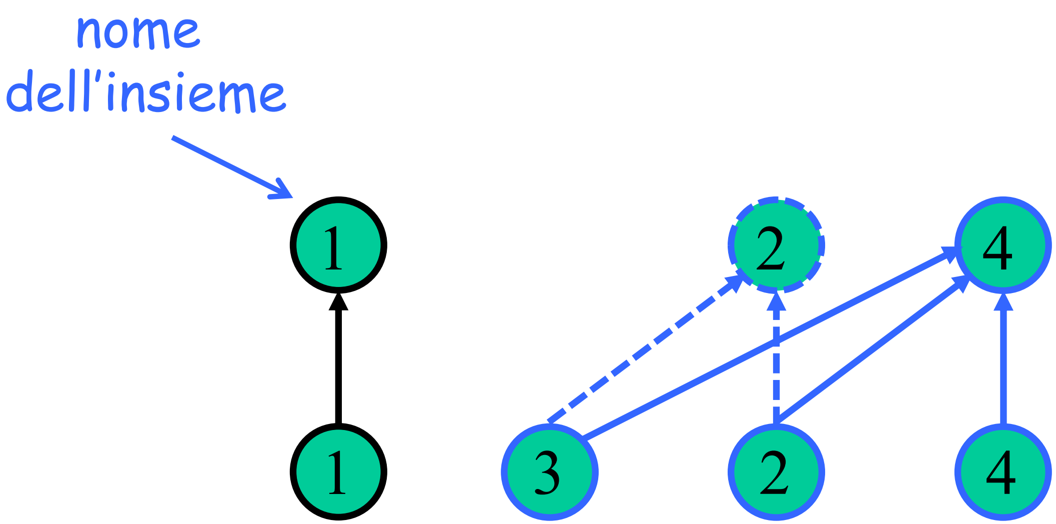


un esempio:

Sequenza di operazioni:

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

union(4,2)

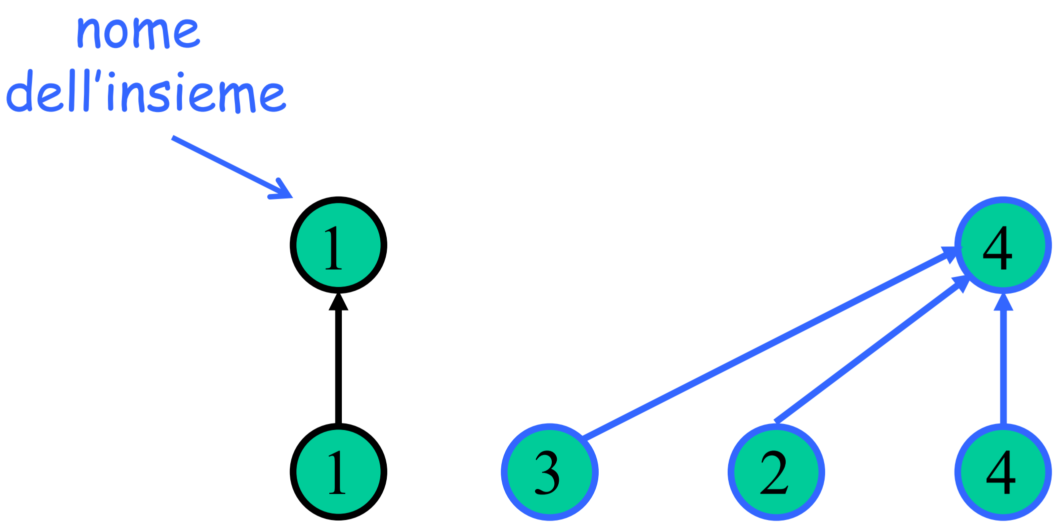


un esempio:

Sequenza di operazioni:

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

union(4,2)



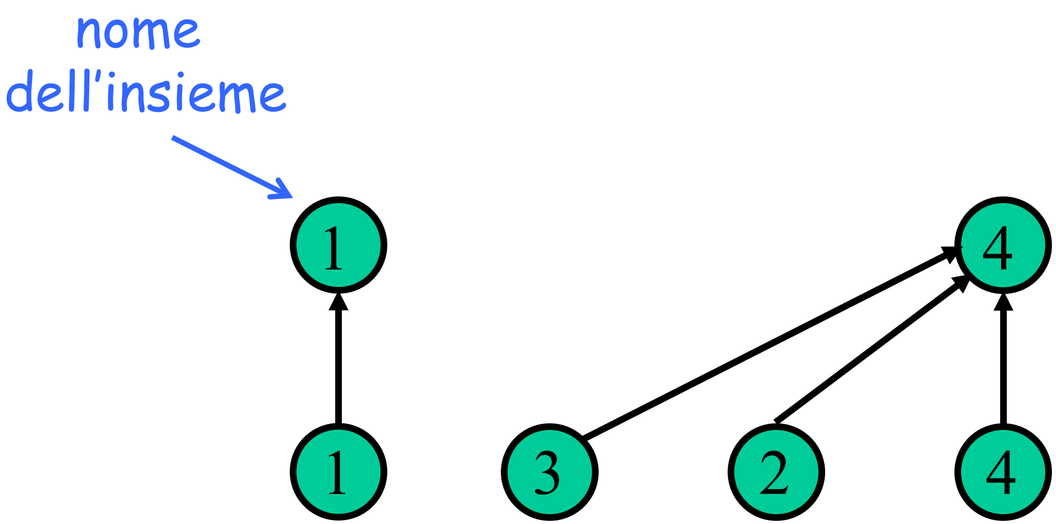
un esempio:

Sequenza di operazioni:

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

union(4,2)

find(2)



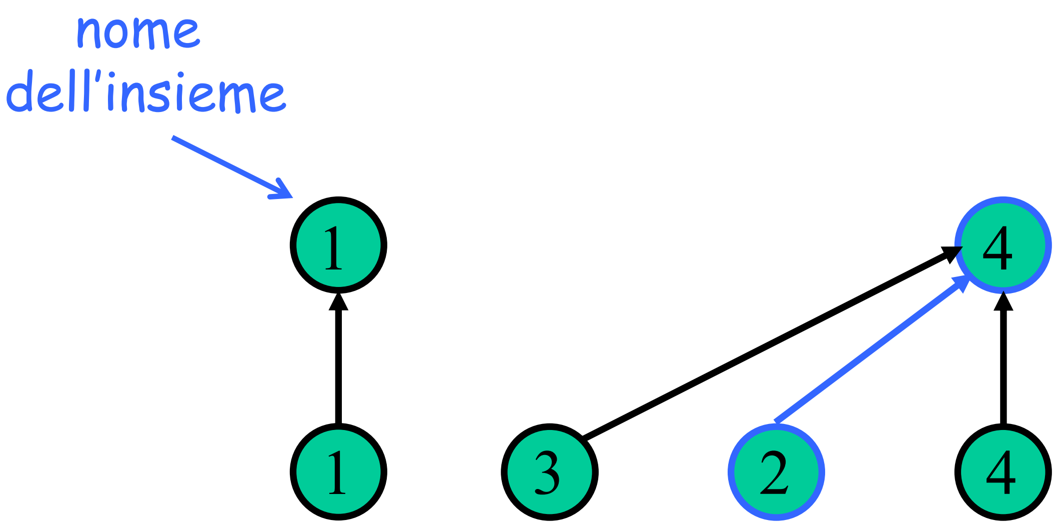
un esempio:

Sequenza di operazioni:

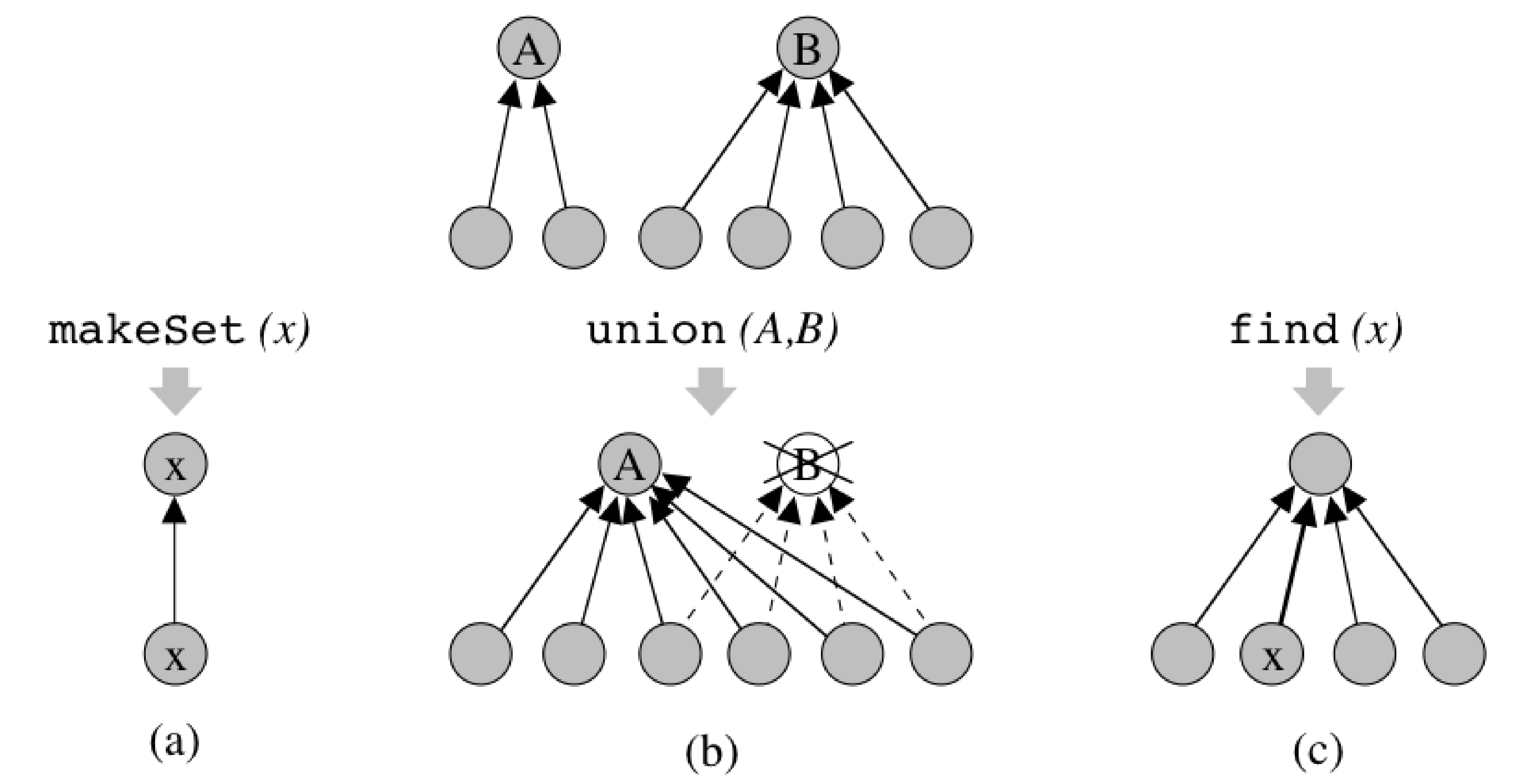
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)

union(4,2)

find(2)



Complessità temporale per singola operazione



O(1) O(n) O(1)

e se eseguo una sequenza arbitraria di operazioni?

Union di costo lineare

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

union $(n-1, n)$	1 cambio puntatore
union $(n-2, n-1)$	2 cambi puntatori
union $(n-3, n-2)$	3 cambi puntatori
$\vdots$	:
union $(2, 3)$	n-2 cambi puntatori
union $(1, 2)$	n-1 cambi puntatori

$\Theta(n^2)$

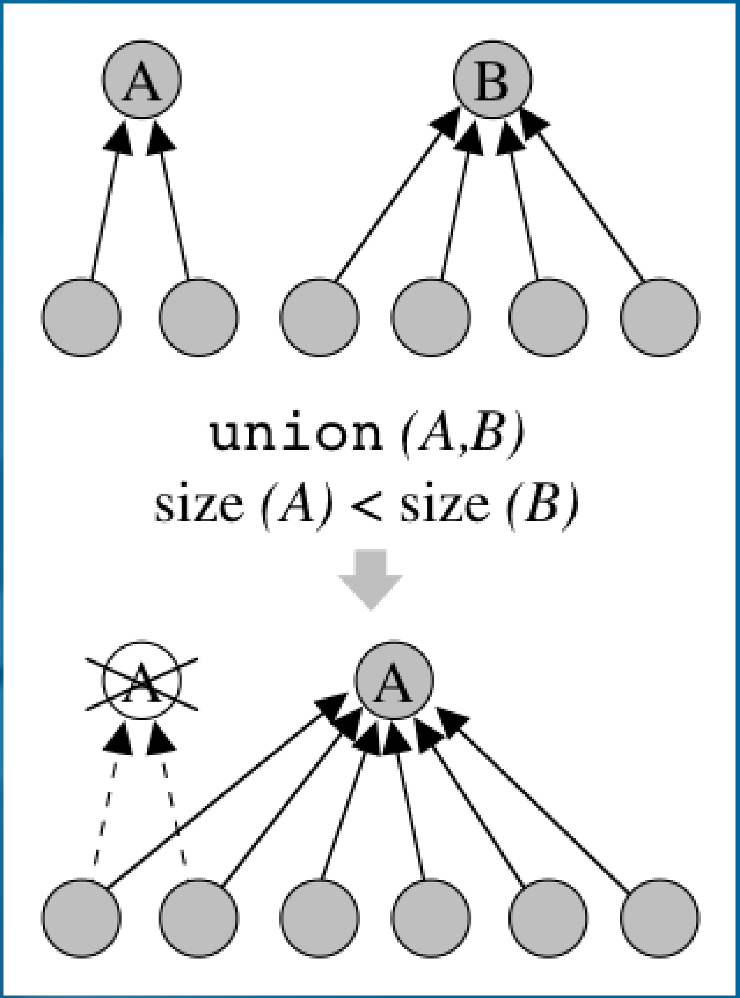
Migliorare la struttura QuickFind: euristica *union by size*

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

Union by size

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 (per aggiornare il nome)

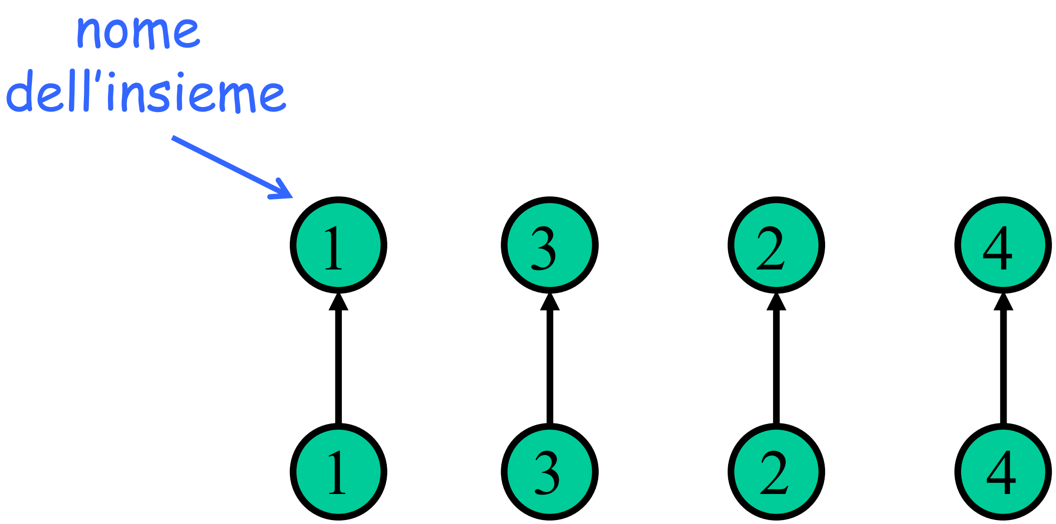
Ogni insieme mantiene esplicitamente anche la propria size (numero di elementi)



un esempio:

Sequenza di operazioni:

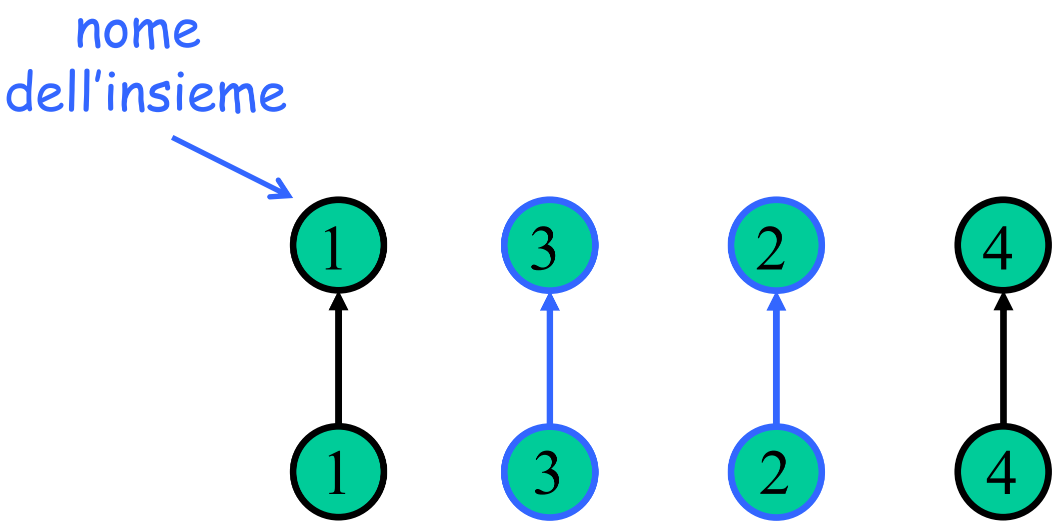
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

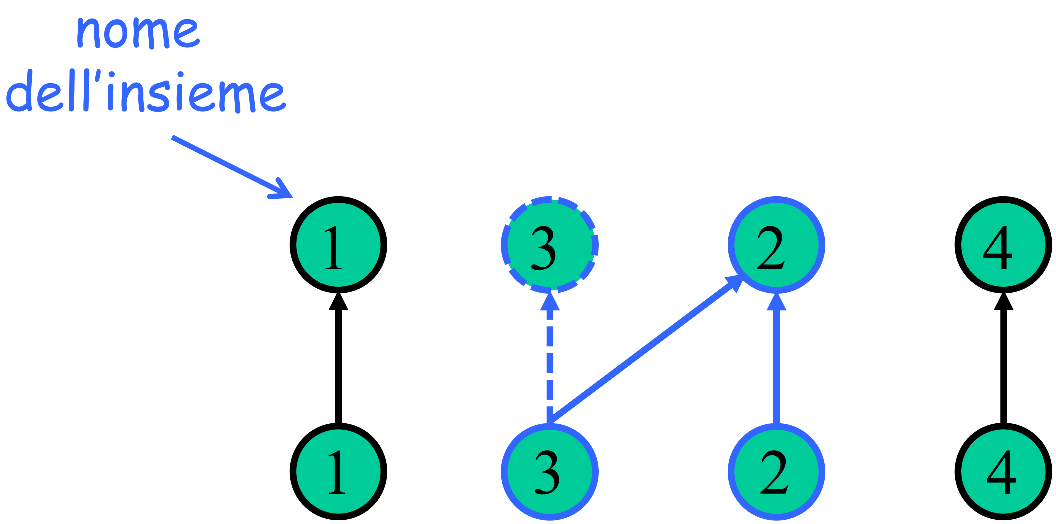
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

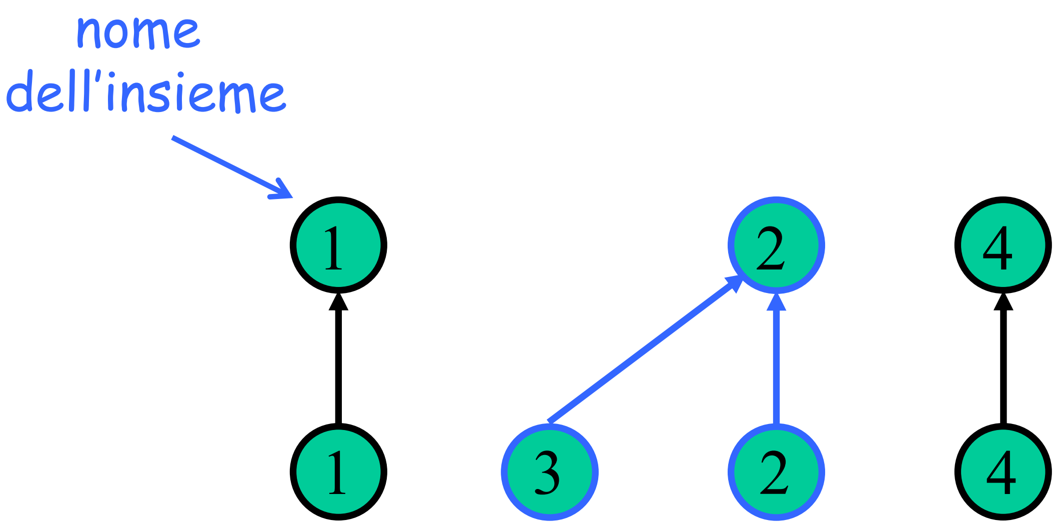
`makeSet(1)` `makeSet(3)` `makeSet(2)` `makeSet(4)` `union(2,3)`



un esempio:

Sequenza di operazioni:

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



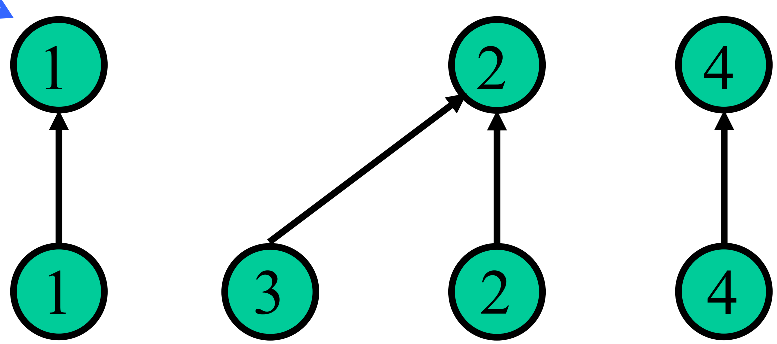
un esempio:

Sequenza di operazioni:

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

union(4,2)

nome dell'insieme

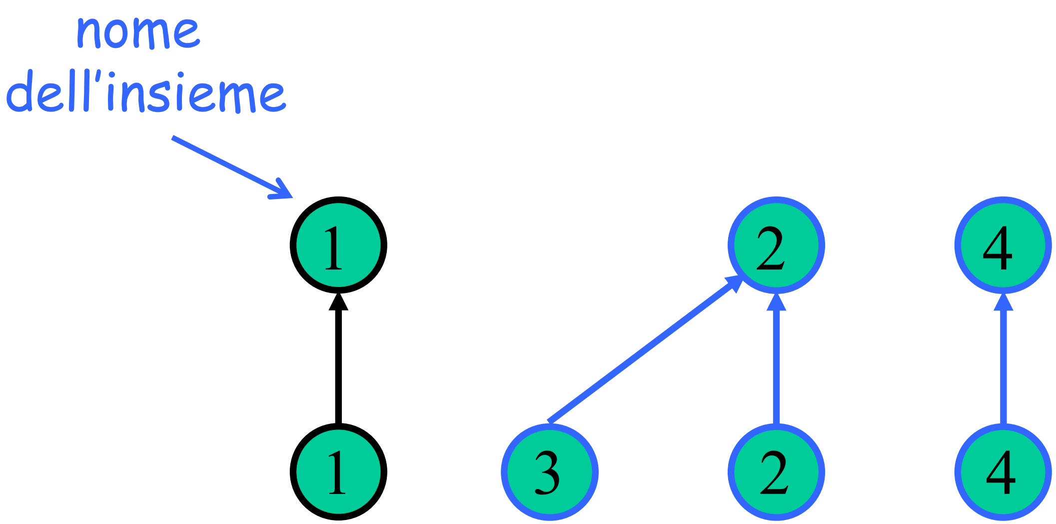


un esempio:

Sequenza di operazioni:

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

`union`(4,2)



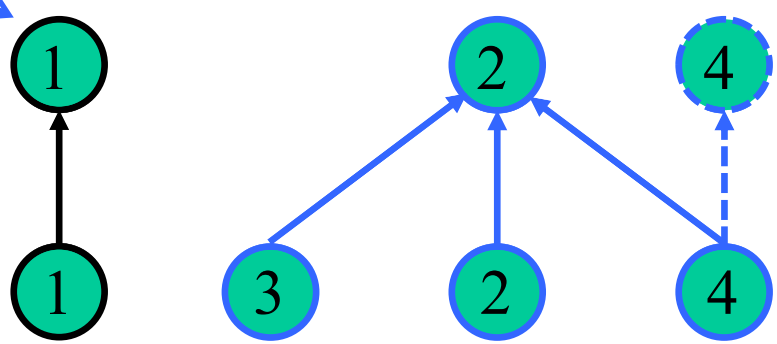
un esempio:

Sequenza di operazioni:

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

`union`(4,2)

nome dell'insieme

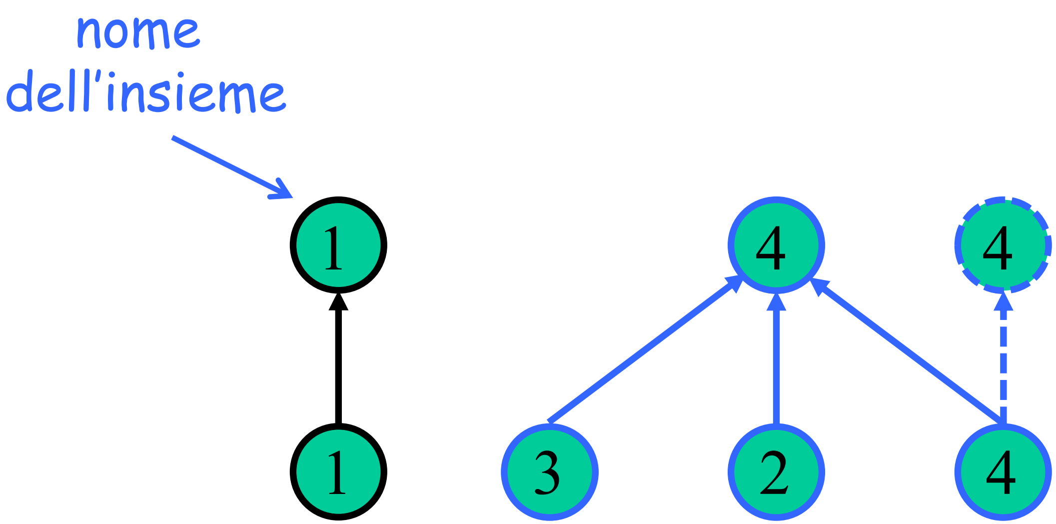


un esempio:

Sequenza di operazioni:

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

`union`(4,2)



un esempio:

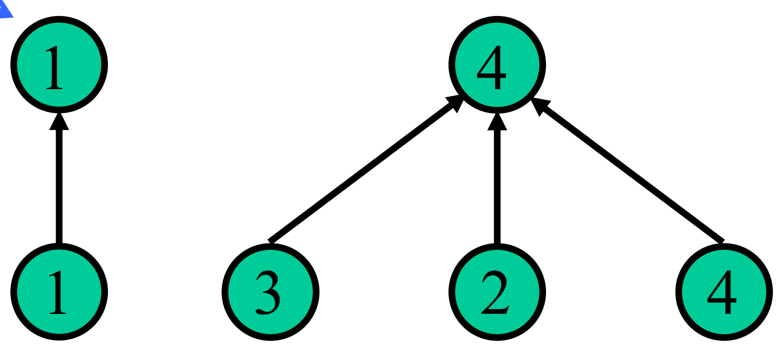
Sequenza di operazioni:

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

`union`(4,2)

`find`(2)

nome dell'insieme



un esempio:

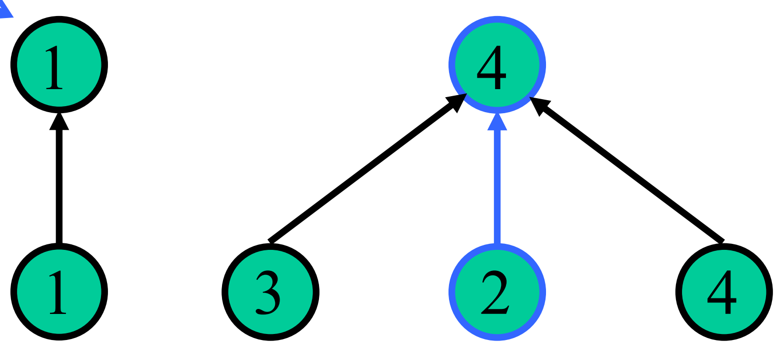
Sequenza di operazioni:

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

union(4,2)

find(2)

nome dell'insieme



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 size$(x) = 1$ alla radice x.

Realizzazione (2/3)

`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.

Realizzazione (3/3)

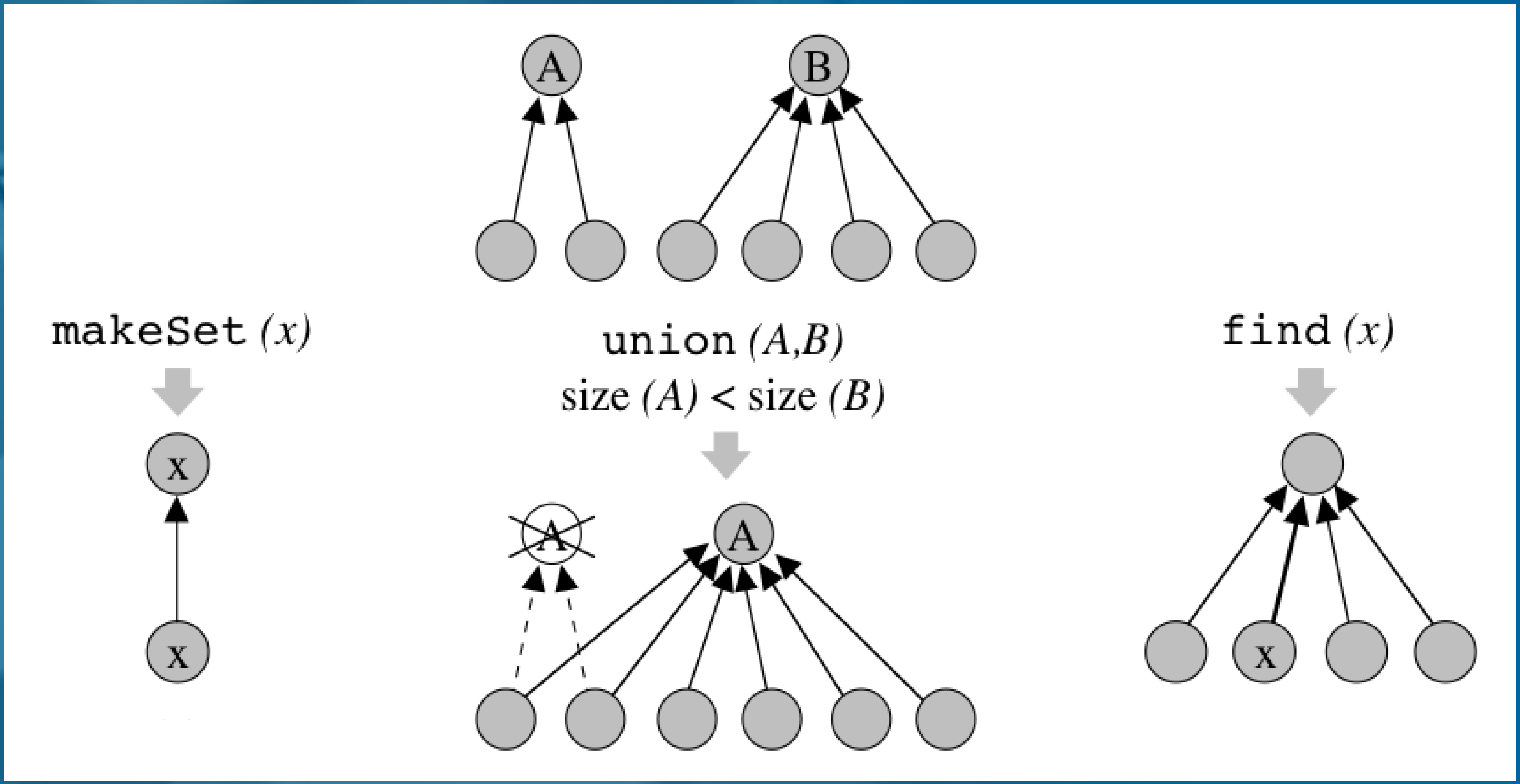
`union`$(name\ a, name\ b)$ $T_{am} = O(\log n)$

considera l'albero A corrispondente all'insieme di nome a, e l'albero B corrispondente all'insieme di nome b. Se size$(A) \geq$ size(B), muovi tutti i puntatori dalle foglie di B alla radice di A, e cancella la vecchia radice di B. Altrimenti (size$(B) >$ 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 (size(A) + 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 temporale per singola operazione



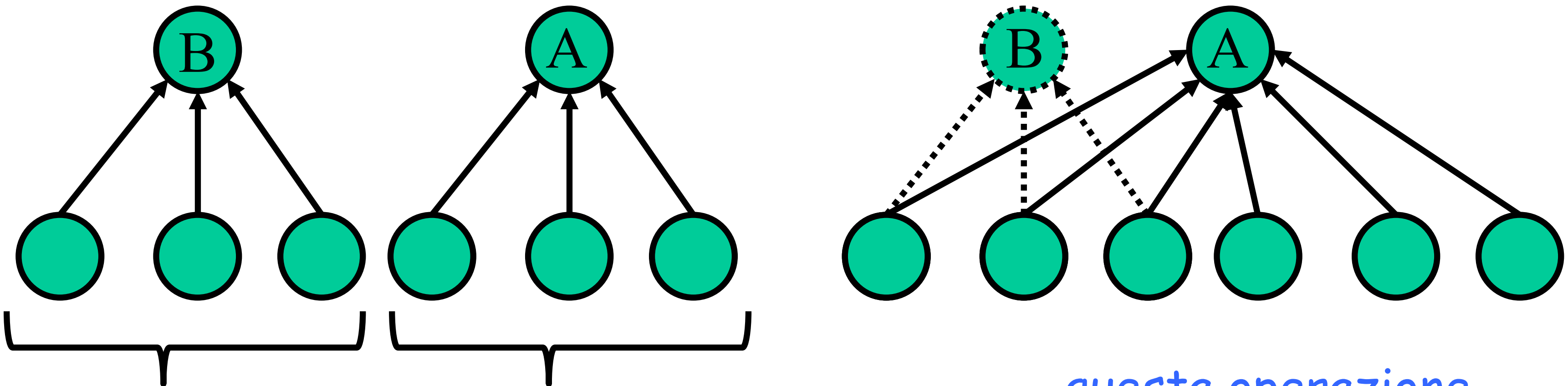
O(1)

O(n) nel caso peggiore
O(log n) ammortizzata

O(1)

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/elemento 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

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

$\blacksquare$

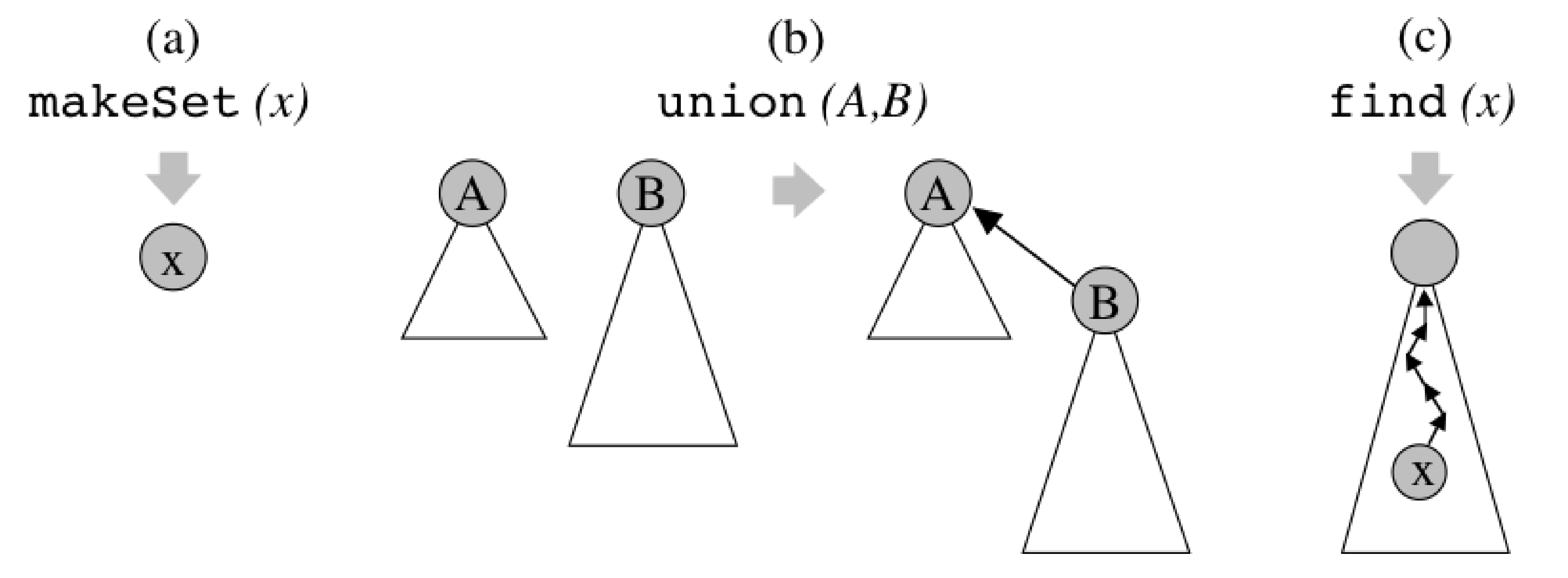
Approcci elementari (basati su alberi)

Due strategie: QuickFind e $\boxed{\text{QuickUnion}}$

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)

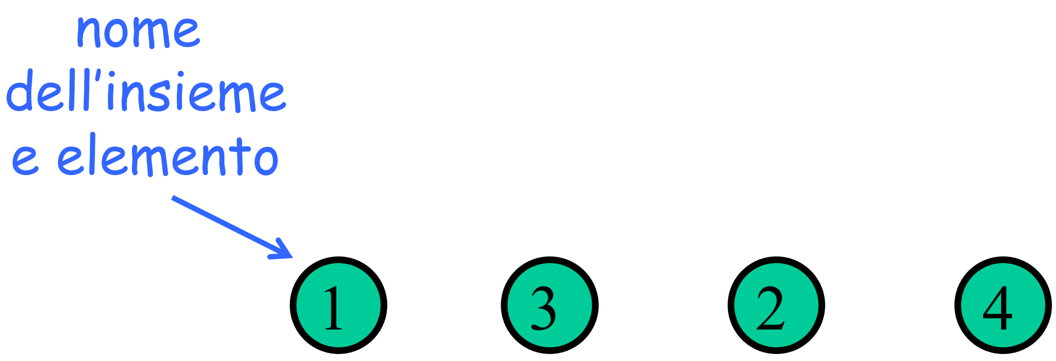
Implementazioni delle operazioni



un esempio:

Sequenza di operazioni:

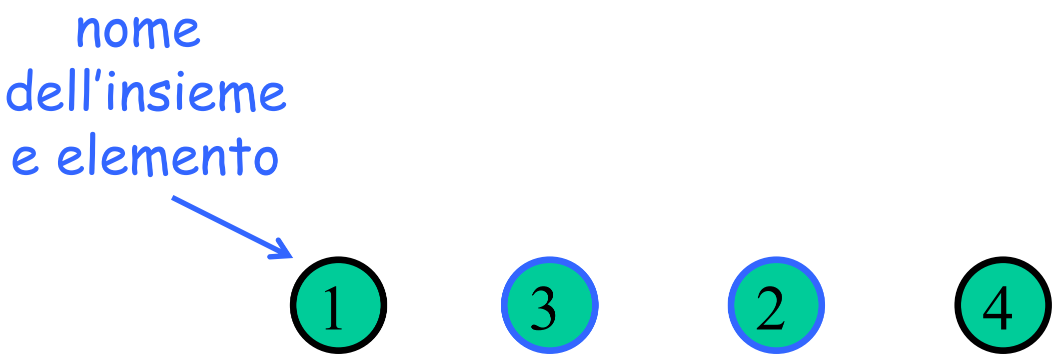
makeSet(1) makeSet(3) makeSet(2) makeSet(4) union(2,3)



un esempio:

Sequenza di operazioni:

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

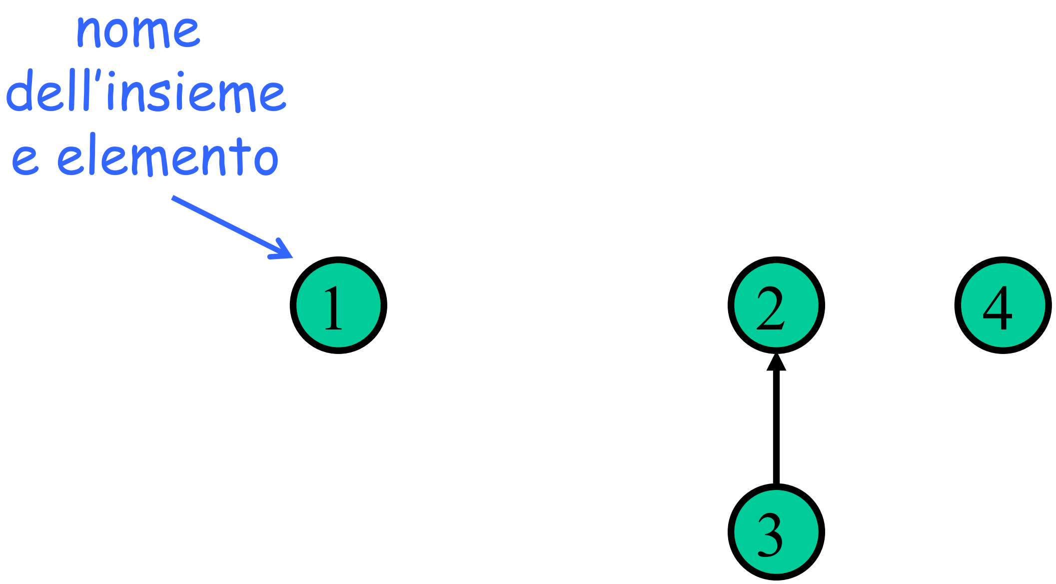


un esempio:

Sequenza di operazioni:

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

`union`(4,2)



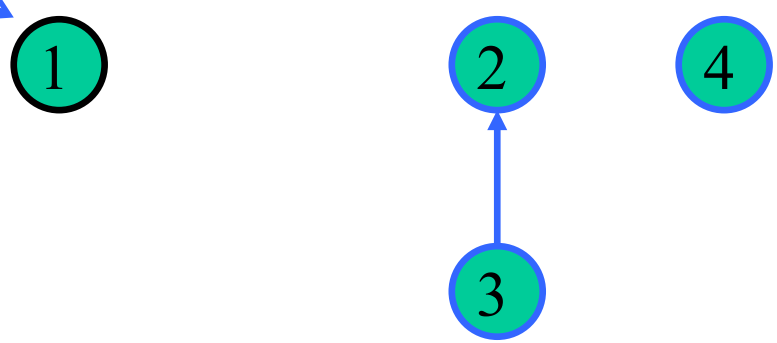
un esempio:

Sequenza di operazioni:

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

union(4,2)

nome dell'insieme e elemento



un esempio:

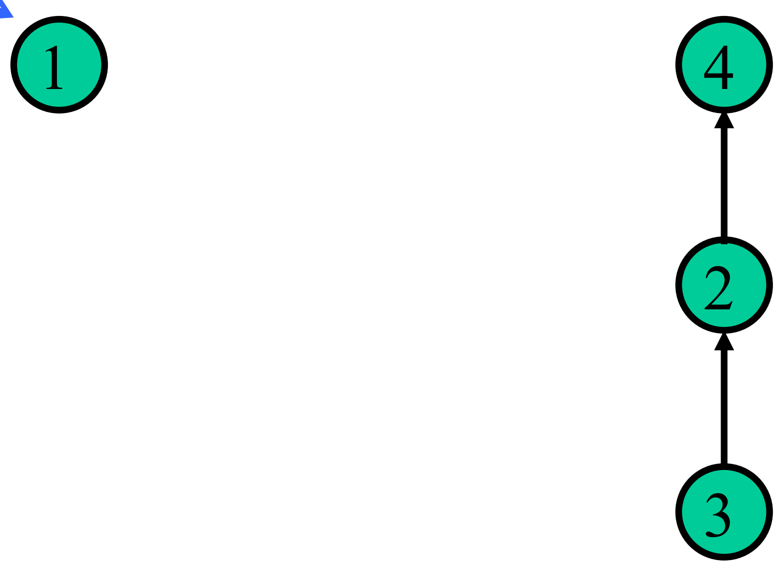
Sequenza di operazioni:

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

`union`(4,2)

`union`(4,1)

nome dell'insieme e elemento



un esempio:

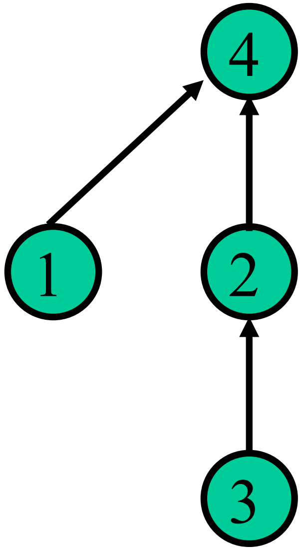
Sequenza di operazioni:

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

union(4,2)

union(4,1)

find(3)



un esempio:

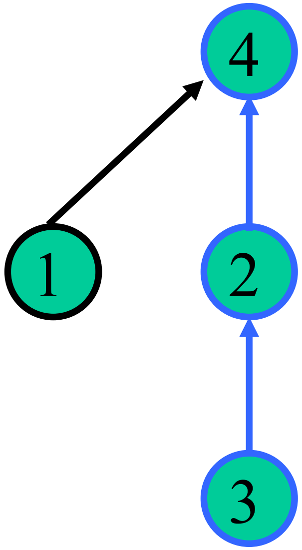
Sequenza di operazioni:

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

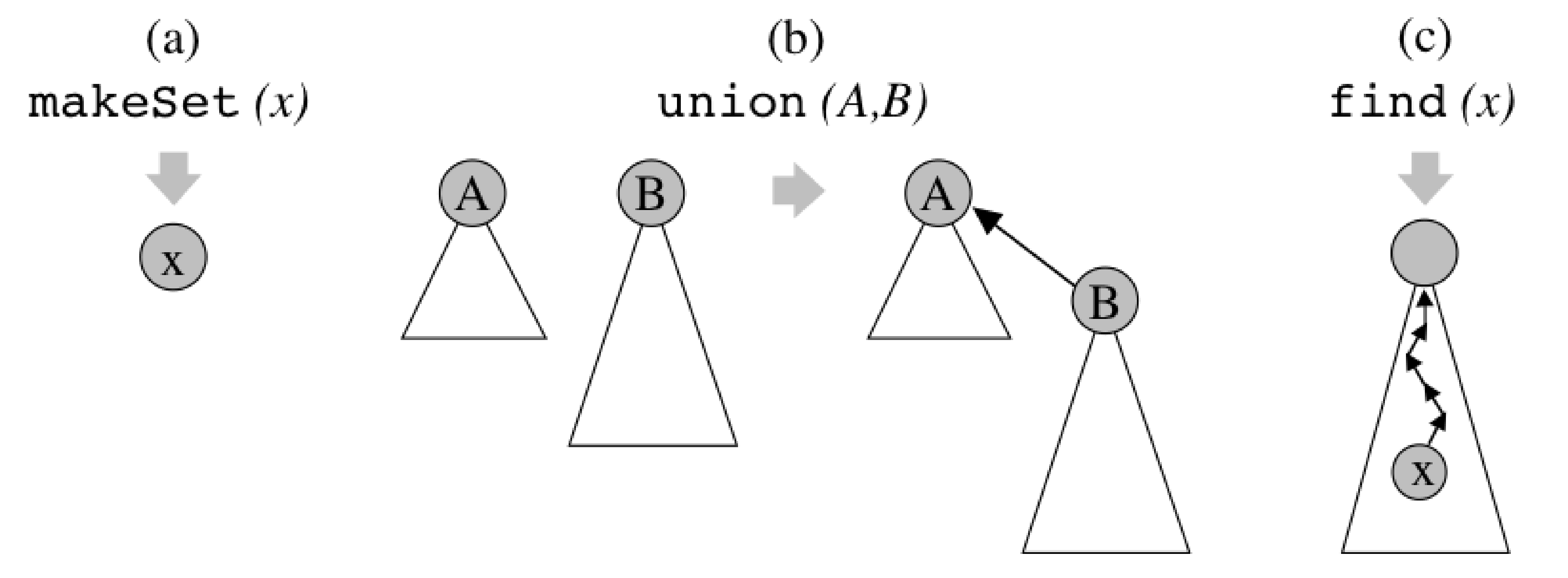
union(4,2)

union(4,1)

find(3)



Complessità delle operazioni



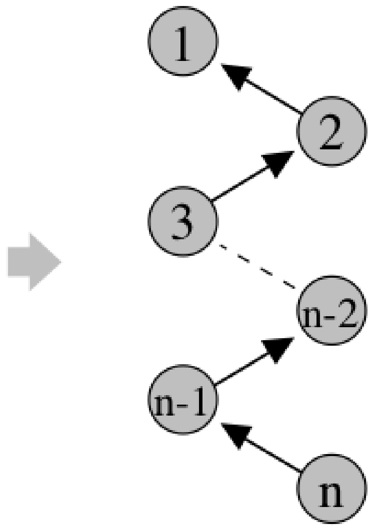
O(1) O(1) O(n)

e se eseguo una sequenza arbitraria di operazioni?

Find di costo lineare

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

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



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

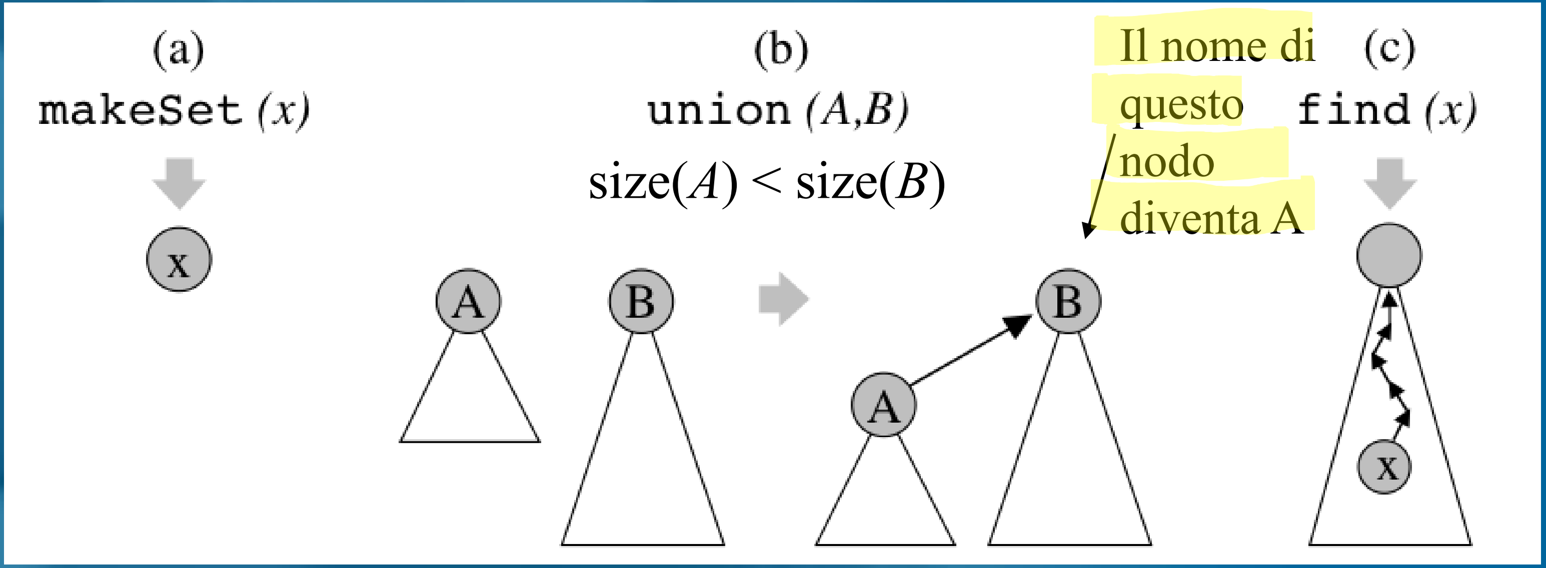
Migliorare la struttura QuickUnion: euristica *union by size*

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

Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano

Bilanciamento in alberi QuickUnion

Union by size: nell'unione degli insiemi A e B, rendiamo la radice dell'albero con meno nodi figlia della radice dell'albero con più nodi



un esempio:

Sequenza di operazioni:

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

un esempio:

Sequenza di operazioni:

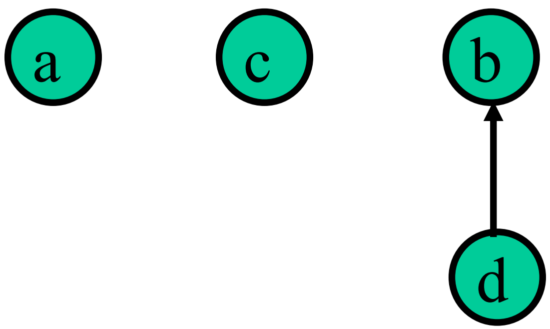
`makeSet`(**a**) `makeSet`(**c**) `makeSet`(**b**) `makeSet`(**d**) `union`(**b,d**)

un esempio:

Sequenza di operazioni:

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

union(a,c)

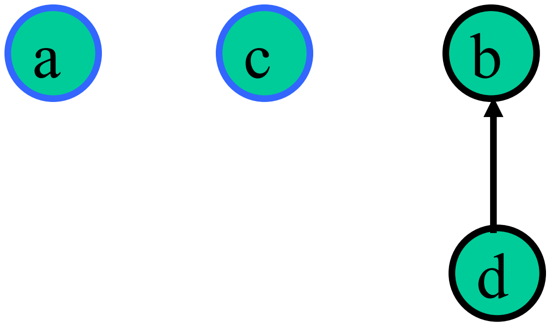


un esempio:

Sequenza di operazioni:

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

`union`(**a,c**)

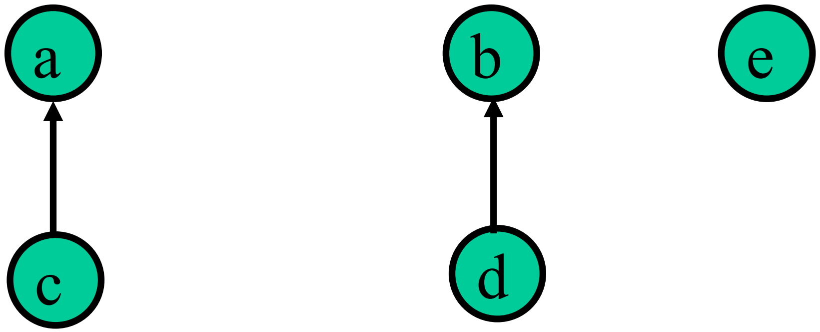


un esempio:

Sequenza di operazioni:

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

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

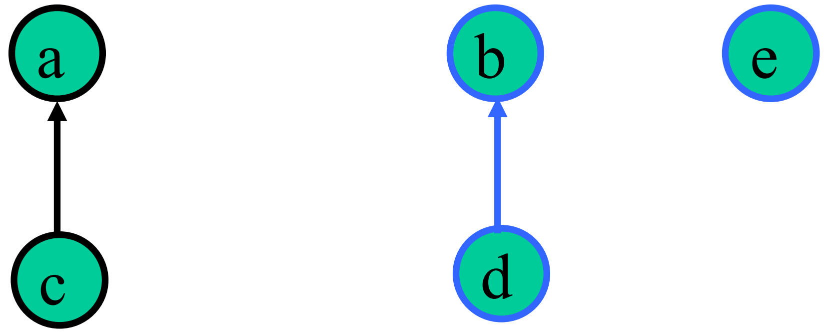


un esempio:

Sequenza di operazioni:

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

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

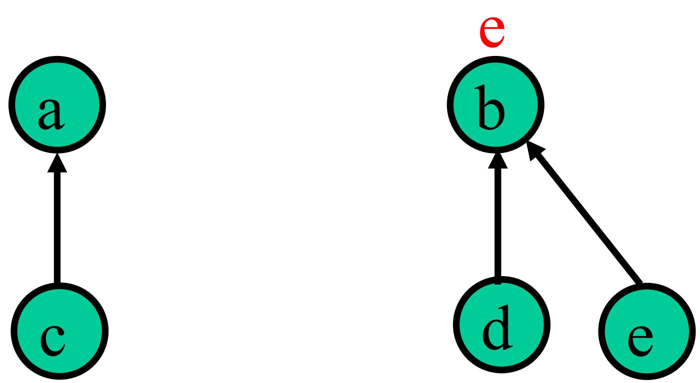


un esempio:

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)

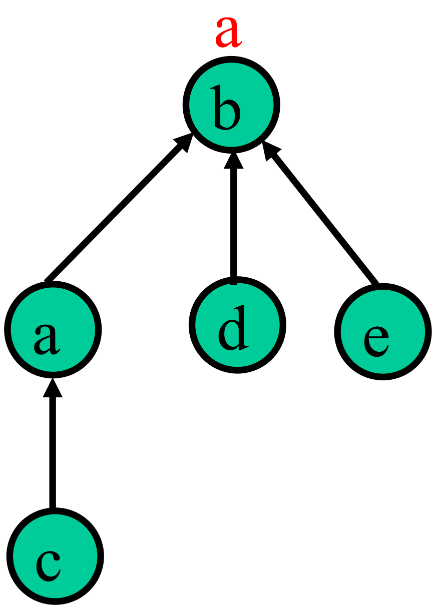


un esempio:

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**)



Lemma: Con la union by size, dato un albero QuickUnion con size (numero di nodi) s e altezza h vale che $s \geq 2^h$.

dim: provate a dimostrarlo voi.

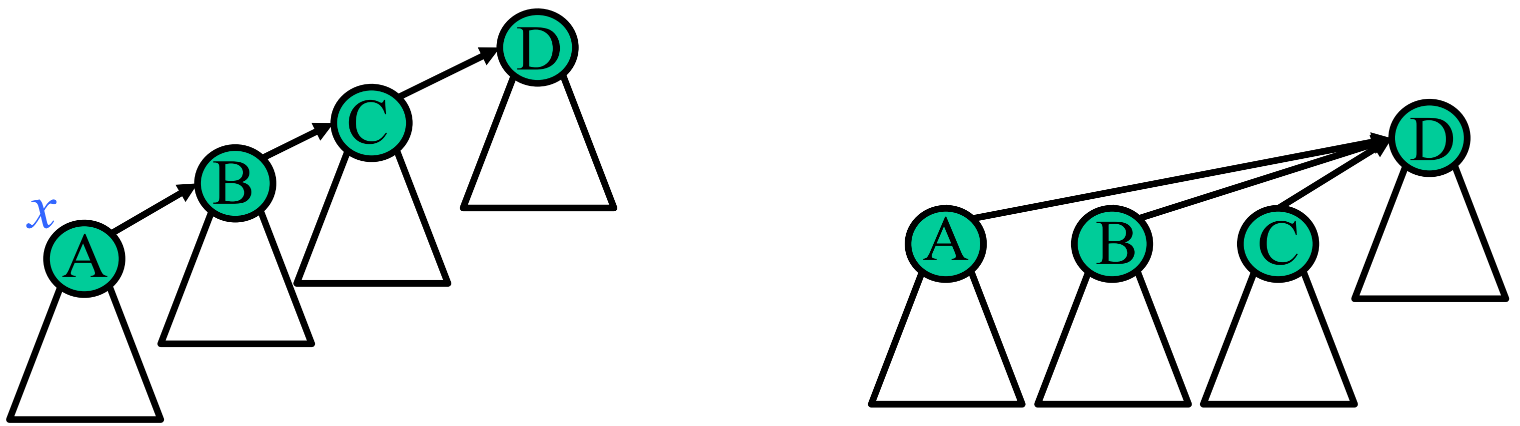
$\Downarrow$

L'operazione find richiede tempo O(log n)

$\Downarrow$

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

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}	$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}16$	$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}16$	$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}16$

La funzione $\alpha(m,n)$

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

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

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)

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

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

$= \left. 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \right\} 16 \quad >> 10^{80}$ $\cong$ numero stimato di atomi nell'universo!

$\Rightarrow$ quindi, $\alpha(m,n) \leq 4$ per tutti gli $n < 2^{10^{80}}$

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 $\lfloor m/n \rfloor > \log^* \log_2 n$

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?