Algoritmi e Strutture Dati

Capitolo 9
Union-find

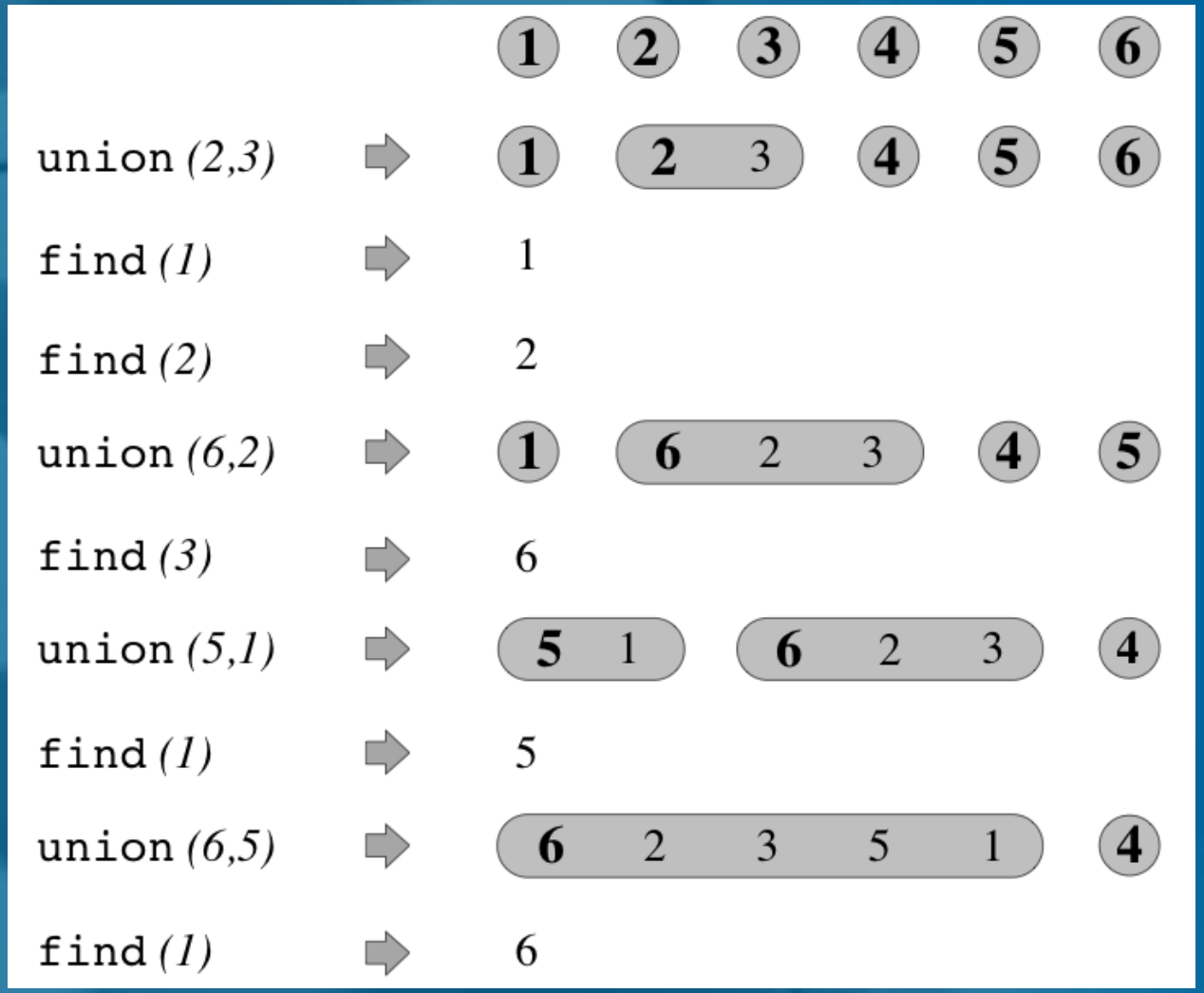
Camil Demetrescu, Irene Finocchi,
Giuseppe F. Italiano

Il problema Union-find

Mantenere una collezione di insiemi disgiunti di elementi distinti (interi in 1 … n) durante una sequenza delle seguenti operazioni:

- union(A,B) = unisce gli insiemi A e B in un unico insieme, di nome A, e distrugge i vecchi insiemi A e B
- find(x) = restituisce il nome dell'insieme contenente l'elemento x
- makeSet(x) = crea il nuovo insieme {x}

Esempio



n = 6

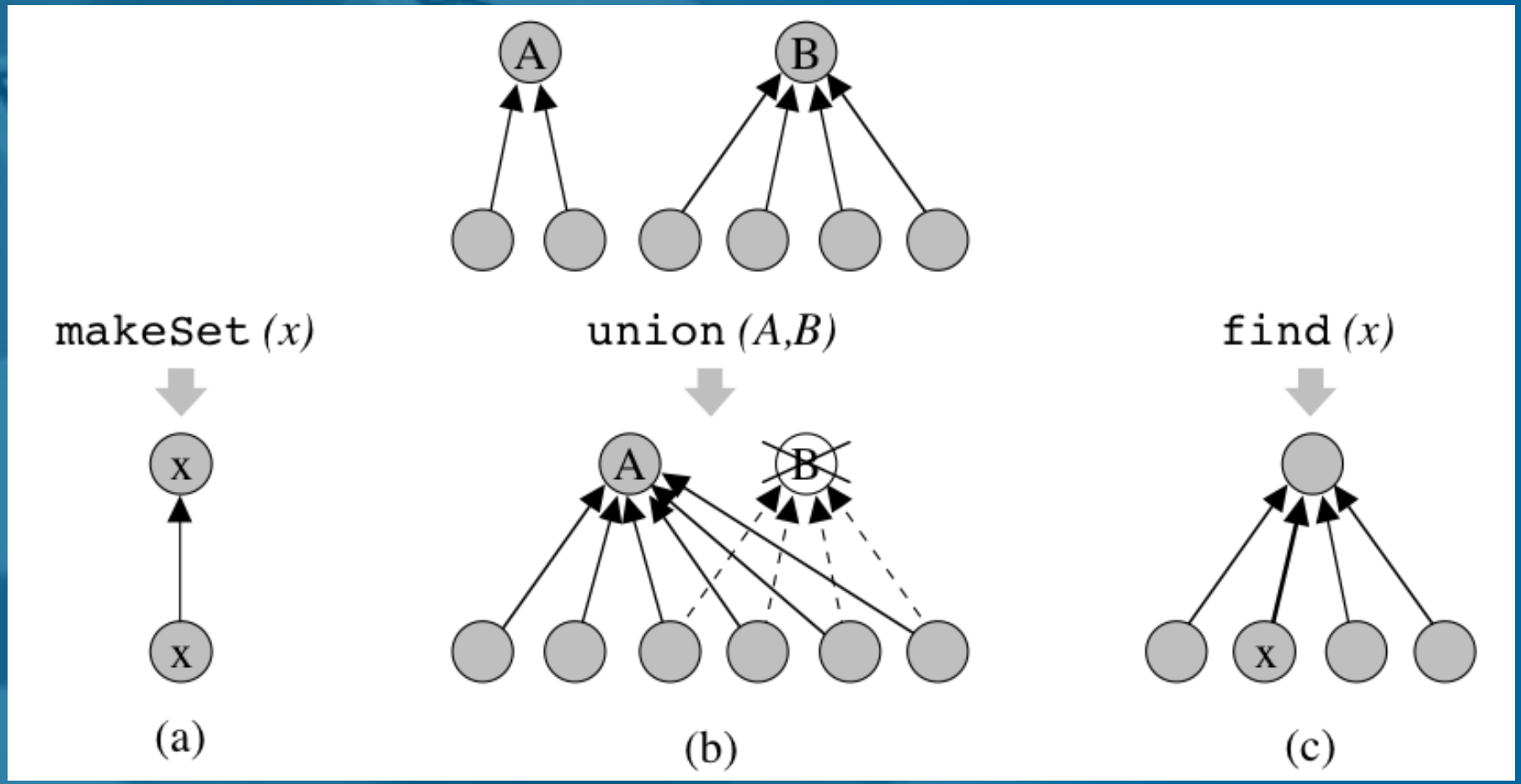
L'elemento in grassetto dà il nome all'insieme

Approcci elementari

Algoritmi di tipo QuickFind

- Usano alberi di altezza uno per rappresentare gli insiemi disgiunti:
  - Radice = nome dell'insieme
  - Foglie = elementi
- find e makeSet richiedono solo tempo O(1), ma union è molto inefficiente: O(n) nel caso peggiore (i collegamenti foglia-radice del secondo insieme devono essere aggiornati).

Esempio



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)

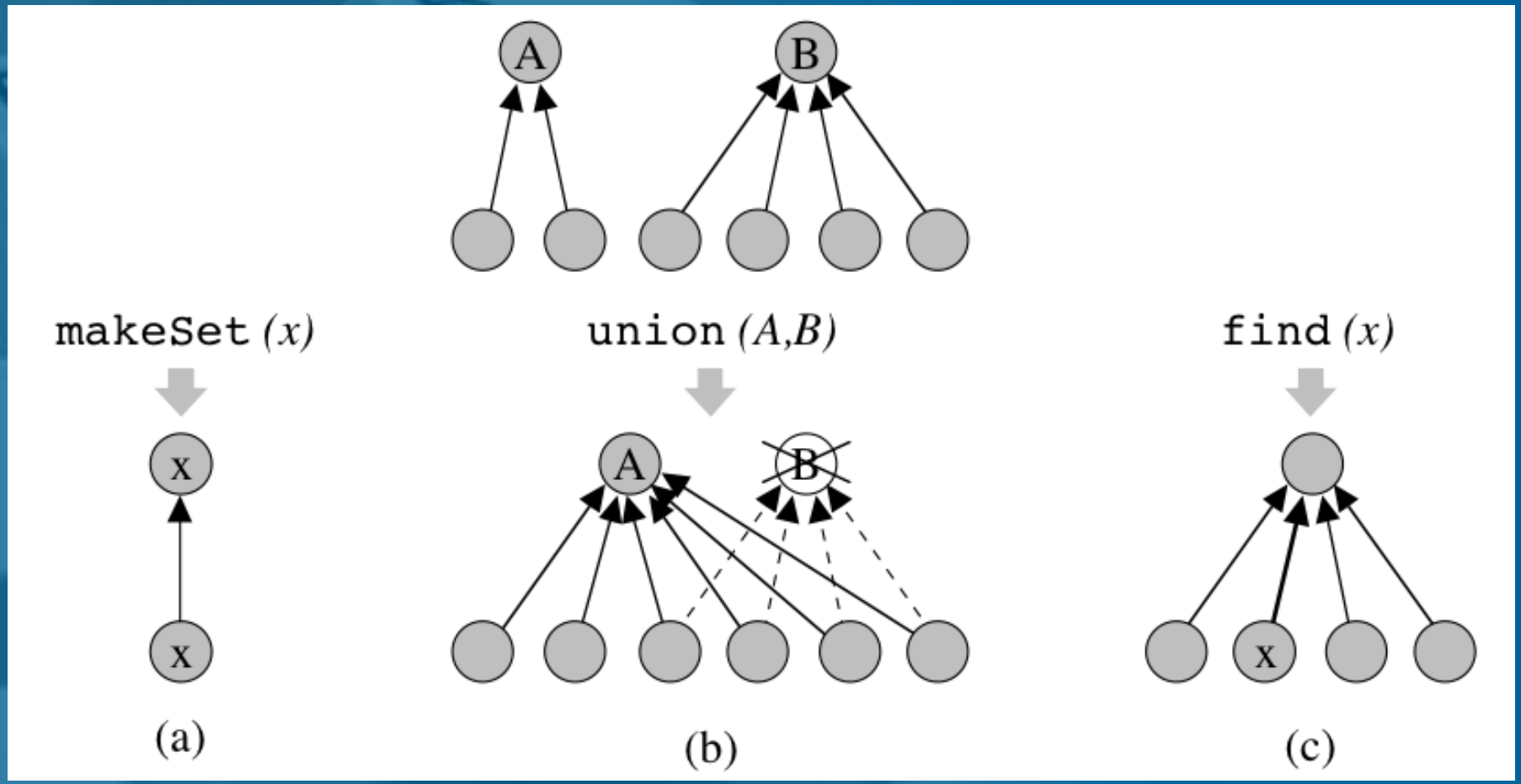
`union`$(name\ a, name\ b)$ $\qquad T(n) = O(n)$

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



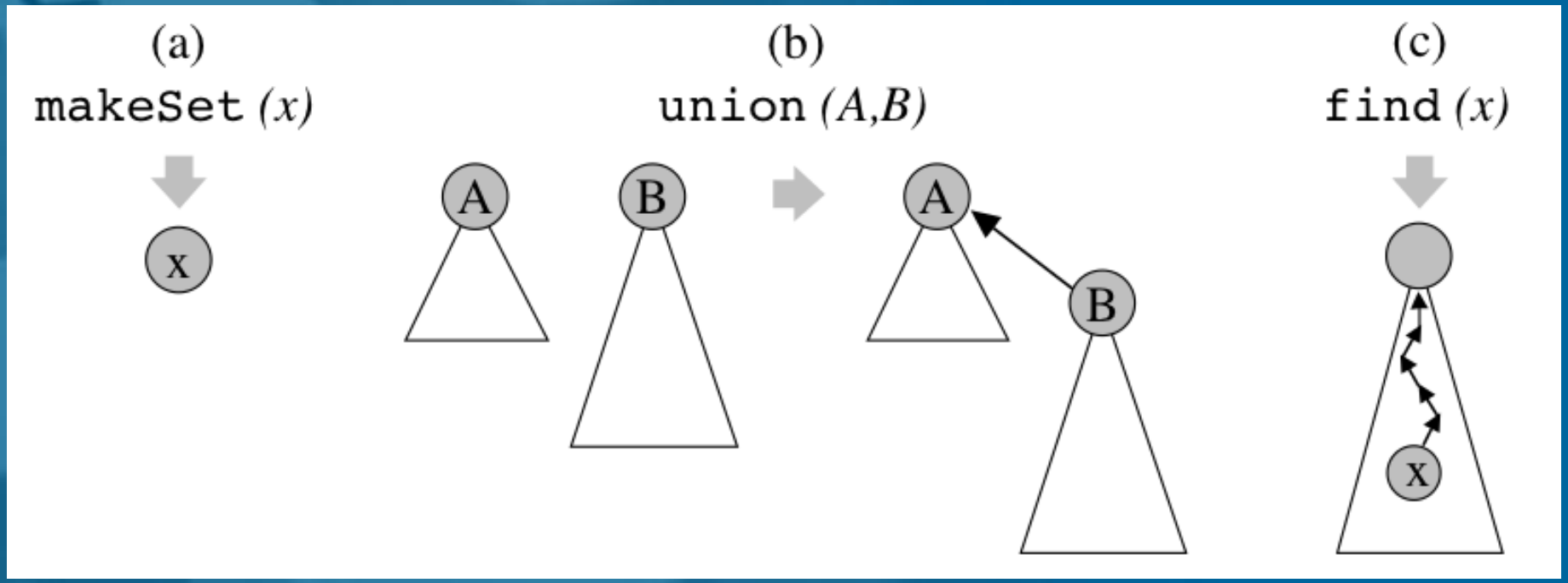
Complessità QuickFind

Teorema: *Alberi QuickFind sono in grado di supportare operazioni makeSet e find in tempo costante. Una union può richiedere nel caso peggiore tempo O(n), dove n è il numero di operazioni makeSet. L'occupazione di memoria è O(n)*

Algoritmi di tipo QuickUnion

- Usano alberi di altezza anche maggiore di uno per rappresentare gli insiemi disgiunti:
  - Radice = elemento rappresentativo dell'insieme
  - Altri nodi = altri elementi dell'insieme
- union e makeSet richiedono solo tempo O(1), ma find è molto inefficiente: O(n) nel caso peggiore
- Oss: makeset crea un solo nodo contenente l'elemento, che rappresenta sia l'elemento che il nome dell'insieme

Esempio



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)

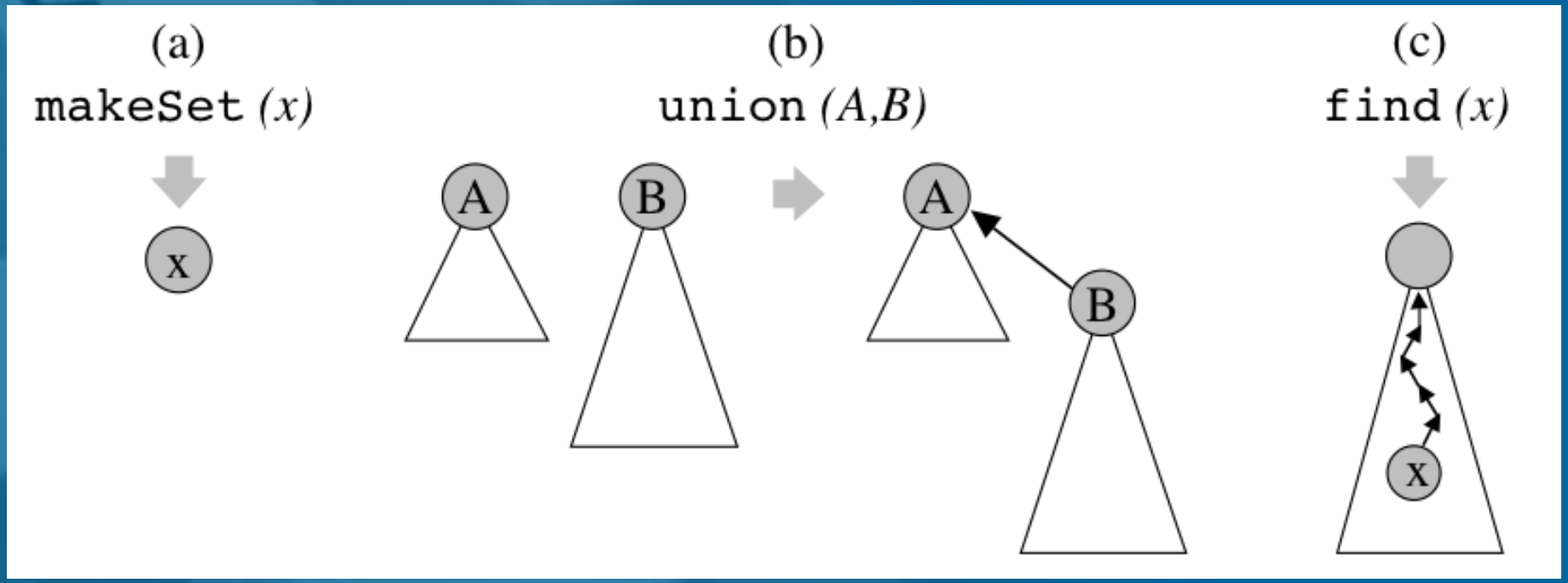
`union`$(name\ a, name\ b)$ $\qquad T(n) = O(1)$

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

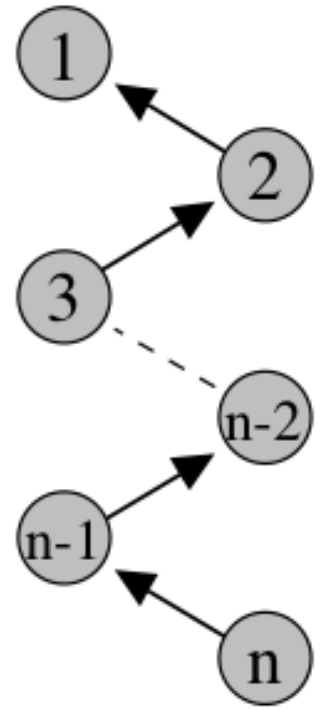
accede alla foglia x corrispondente all'elemento e. Partendo da tale nodo, segue ripetutamente i puntatori al padre fino a raggiungere la radice dell'albero. Restituisce il nome memorizzato in tale radice.

Esempio



Altezza lineare

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

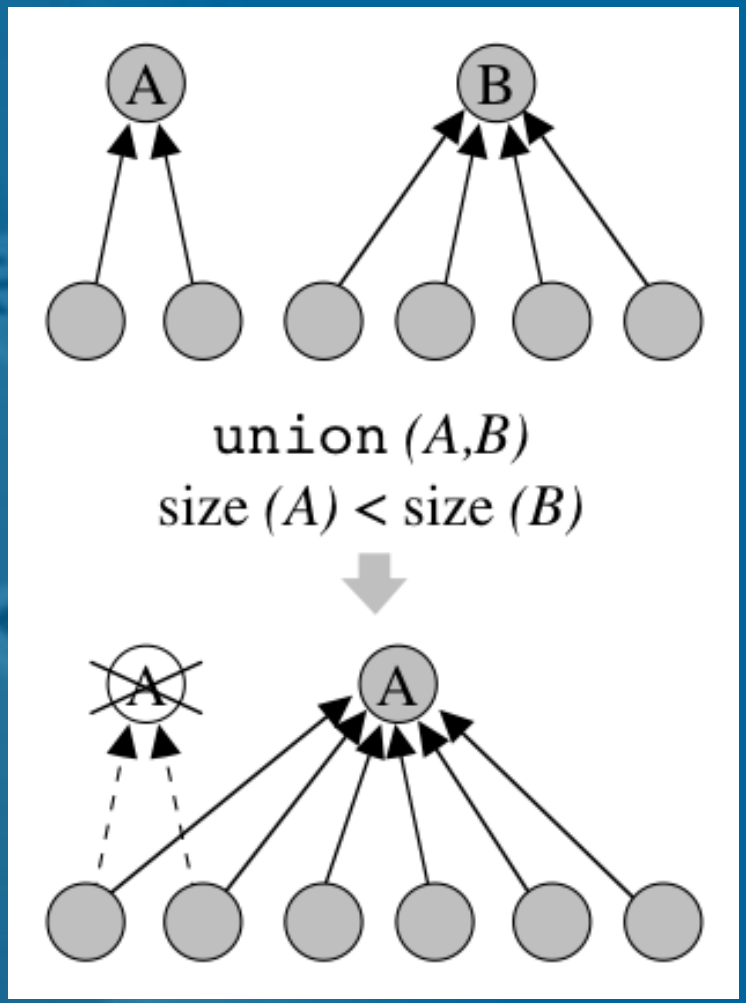


Sequenza di union che costruisce un albero di altezza lineare. Nel caso peggiore una find può comportare una complessità di O(n)

Euristiche di bilanciamento nell'operazione union

Bilanciamento in algoritmi QuickFind

Nell'unione degli insiemi A e B, modifichiamo il padre dei nodi nell'insieme di cardinalità minore



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

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

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 su una intera sequenza

Conseguenza del bilanciamento

Ogni volta che una foglia di un albero QuickFind bilanciato acquista un nuovo padre a causa di un'operazione union, dopo la union tale foglia farà parte di un insieme che è grande almeno il doppio dell'insieme di cui faceva parte precedentemente

Teorema:

Alberi QuickFind *con bilanciamento sulle union* sono in grado di eseguire una sequenza di *m* operazioni di find, *n* operazioni makeSet e al più *n-1* operazioni di union in tempo totale di O(m + n logn). L'occupazione di memoria è O(n)

Analisi ammortizzata (1/2)

Se eseguiamo m find, n makeSet (e quindi al più n-1 union), il tempo richiesto dall'intera sequenza di operazioni è O(m + n log n)

Idea della dimostrazione

- Facile vedere che find e makeSet richiedono tempo O(n+m)
- Per analizzare union, quando creiamo un nodo gli assegniamo log n crediti: in totale, O(n log n) crediti

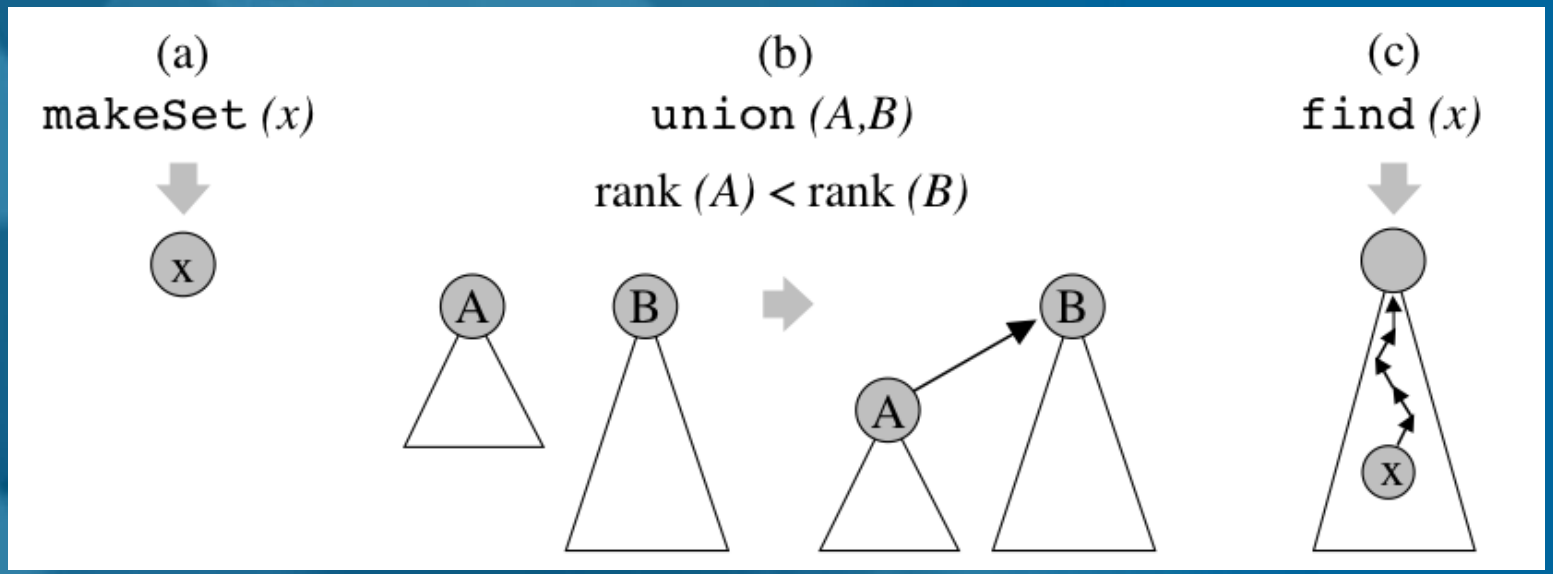
Analisi ammortizzata (2/2)

- Quando eseguiamo una union, paghiamo il tempo speso su ogni nodo il cui padre cambia con uno dei crediti assegnati al nodo
- Un nodo può cambiare al più log n padri, poiché ogni volta che cambia padre la cardinalità dell'insieme cui appartiene raddoppia
- I crediti assegnati al nodo sono quindi sufficienti per pagare tutti i cambiamenti di padre

Bilanciamento in algoritmi QuickUnion

Union by rank: 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`$(elem\ e) \rightarrow name$ $\qquad T(n) = O(\log n)$

accede alla foglia 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.

Conseguenza del bilanciamento

- Lemma: Un albero QuickUnion bilanciato in altezza con radice x ha almeno $2^{\text{rank}(x)}$ nodi
- Dimostrazione per induzione sul numero di operazioni
  - Se rank(A) > rank(B) durante una union:

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

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

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

Analisi (nel caso peggiore)

Corollario: l'altezza di un albero QuickUnion bilanciato è limitata superiormente da $\log_2 n$, con n = numero di makeSet

L'operazione find richiede nel caso peggiore tempo O(log n)

Teorema: Alberi QuickUnion bilanciati in altezza sono in grado di supportare operazioni makeSet e Union in tempo costante. Una Find richiede nel caso peggiore tempo O(log n) dove n è il numero di operazioni makeSet eseguite nella sequenza. L'occupazione di memoria è O(n)

Bilanciamento in algoritmi 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

size(x) = numero nodi nell'albero di cui x è radice

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`	$O(1)$	$O(1)$	$O(\log n)$

Euristiche di compressione nell'operazione find

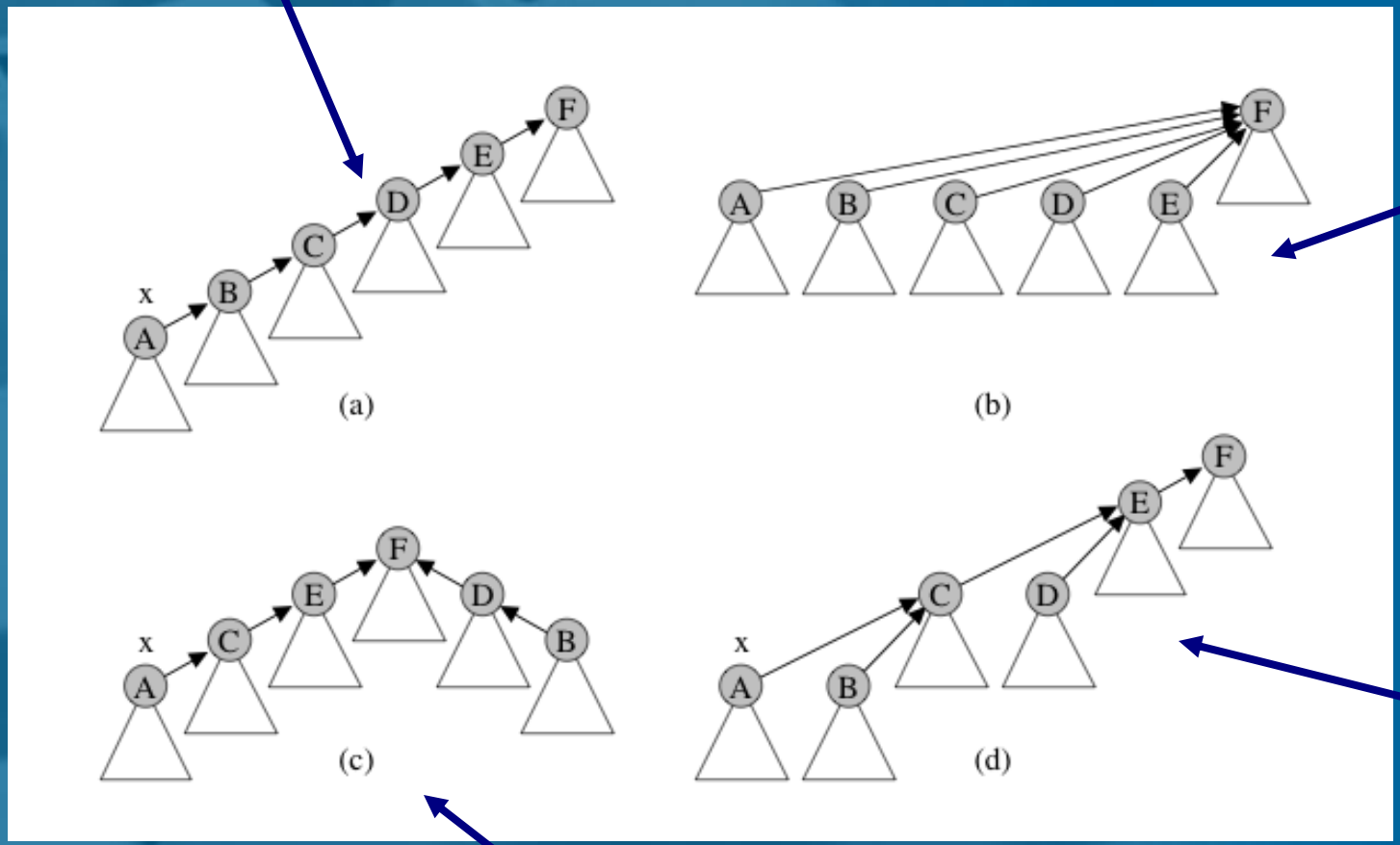
Path compression, splitting e halving

Siano $u_0, u_1, \ldots, u_{t-1}$ i nodi incontrati nel cammino esaminato da find (x), con $x = u_0$

- Path compression: rendi il nodo u_i figlio della radice u_{t-1}, per ogni $i \leq t-3$
- Path splitting: rendi il nodo u_i figlio del nonno u_{i+2}, per ogni $i \leq t-3$
- Path halving: rendi il nodo u_{2i} figlio del nonno u_{2i+2}, per ogni $i \leq \lfloor (t-1)/2 \rfloor - 1$

Esempi

albero prima di find(x)



Path compression

Path halving

Path splitting

Union-find con bilanciamento e compressione

Combinazioni delle euristiche

- Combinando le euristiche di bilanciamento e compressione si ha un totale di sei algoritmi:

  (union by rank o union by size) x (path splitting, path compression, o path halving)

- Tutti gli algoritmi hanno le stesse prestazioni, asintoticamente superiori a quanto visto finora

Terminologia

$$
\begin{aligned}
\log^{(1)} n &= \log n \\
\log^{(2)} n &= \log(\log^{(1)} n) = \log\log n \\
\log^{(3)} n &= \log(\log^{(2)} n) = \log\log\log n \\
&\dots \\
\log^{(i)} n &= \log(\log^{(i-1)} n) = \underbrace{\log\log\dots\log}_{i \text{ volte}} n, \qquad \text{per } i \geq 2
\end{aligned}
$$

$$
\log^* n = \left\{ \min i \mid \log^{(i)} n \leq 1 \right\}
$$

La funzione log * cresce molto lentamente

Analisi

Combinando le euristiche di bilanciamento e compressione, una qualunque sequenza di n operazioni makeSet, m operazioni find ed al più (n-1) operazioni union può essere implementata in tempo totale

$$O((n+m) \log^* n)$$

(Non è la migliore analisi possibile)

Conclusioni

- Strutture dati efficienti per il problema union-find
- Partendo da strutture dati semplici con prestazioni non soddisfacenti, le abbiamo migliorate tramite l'uso di opportune euristiche di bilanciamento ed euristiche di compressione di cammini
- Nonostante le euristiche siano estremamente semplici, l'analisi si è rivelata molto sofisticata