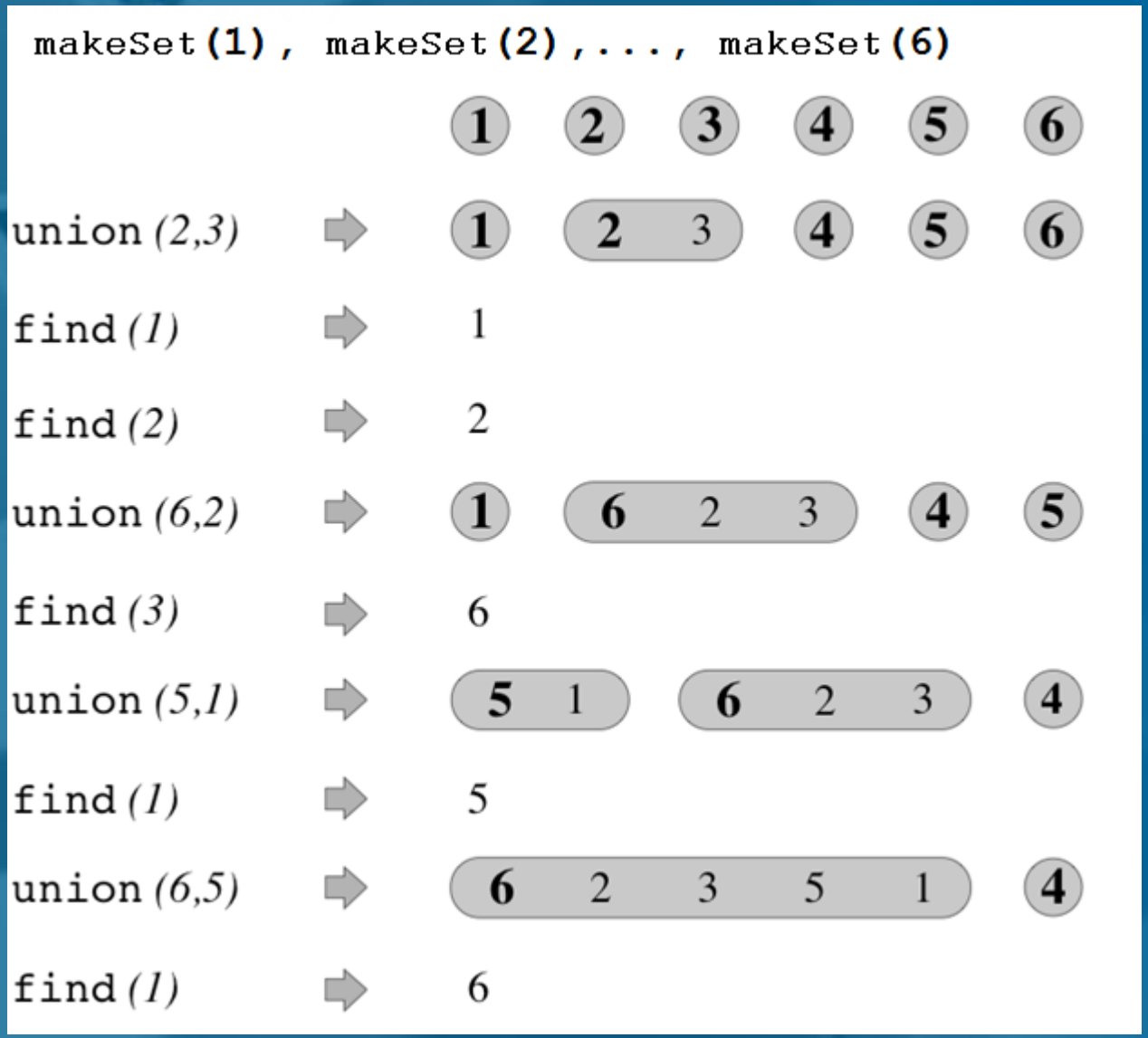
Algoritmi e Strutture Dati

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(elemento x) = crea il nuovo insieme di **nome** x={x}; in questo momento, x è l'elemento rappresentativo dell'insieme, poiché ne determina il nome;
  - union(nome x, nome y) = unisce gli insiemi di nome x e y in un unico insieme di nome x, e distrugge i vecchi insiemi di nome x e y (si suppone di accedere **direttamente** all'elemento rappresentativo degli insiemi x e y)
  - find(elemento 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, 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

Approcci elementari basati su alberi

Alberi QuickFind

- Usiamo un foresta di alberi di altezza **1** per rappresentare gli insiemi disgiunti. In ogni albero:
  - Radice = elemento rappresentativo, ovvero elemento che dà il nome all'insieme
  - Foglie = tutti gli elementi dell'insieme (incluso l'elemento rappresentativo contenuto nella radice; questa ridondanza deriva dal fatto che per mantenere l'invariante che tutti gli alberi della foresta hanno altezza 1, quando creo un nuovo insieme devo creare un albero con 2 nodi di altezza 1, in cui la radice e la foglia contengono lo stesso elemento)

Realizzazione (1/2)

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

dati: $S(n) = \Theta(n)$

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.

La radice e la foglia possono essere pensati come due record contenenti un campo *name* e un campo *elem*, rispettivamente

Realizzazione (2/2)

Notazione: di seguito, per non fare confusione tra nomi degli insiemi ed elementi, indicherò il nome degli insiemi (e degli alberi che li rappresentano) con lettere maiuscole

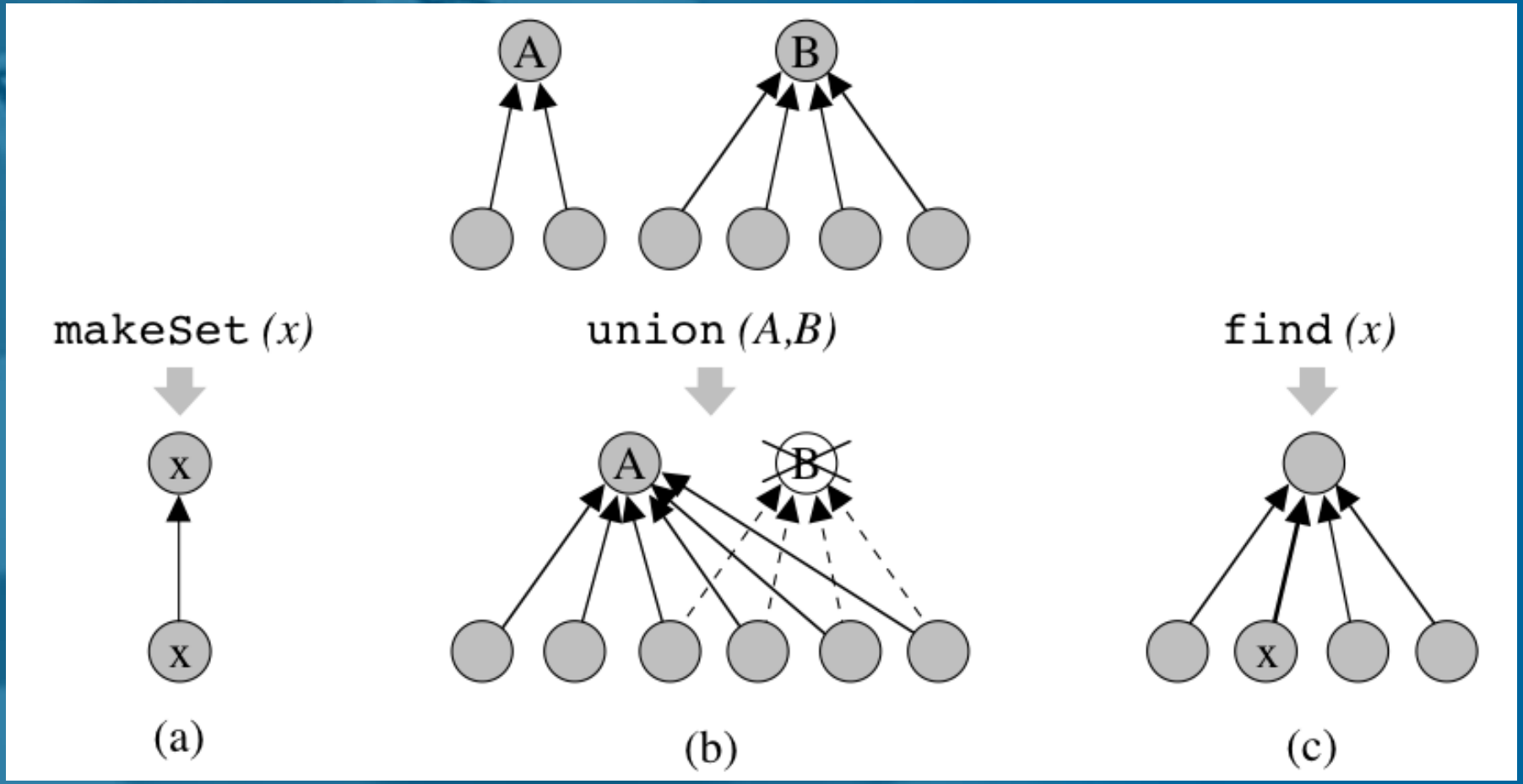
`union`$(name\ a, name\ b)$ $T(n) = O(n)$

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

accede direttamente alla foglia contenente l'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



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 $\Theta(n+1+2+\ldots+(n-1)+m) = \Theta(m+n^2)$

Alberi QuickUnion

- Usiamo una foresta di alberi di altezza ≥ 0 per rappresentare gli insiemi disgiunti. In ogni albero:
  - Radice = elemento rappresentativo dell'insieme
  - Rimanenti nodi = altri elementi (escluso l'elemento rappresentativo contenuto nella radice: non devo mantenere l'invariante che tutti gli alberi della foresta hanno altezza 1)

Realizzazione (1/2)

classe `QuickUnion` **implementa** `UnionFind`:

dati: $S(n) = \Theta(n)$

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. Memorizza e in tale nodo, sia come valore che come nome del nodo.

Il nodo può essere pensato come un record contenente due campi *name* ed *elem*, che in questa prima implementazione coincideranno sempre

Realizzazione (2/2)

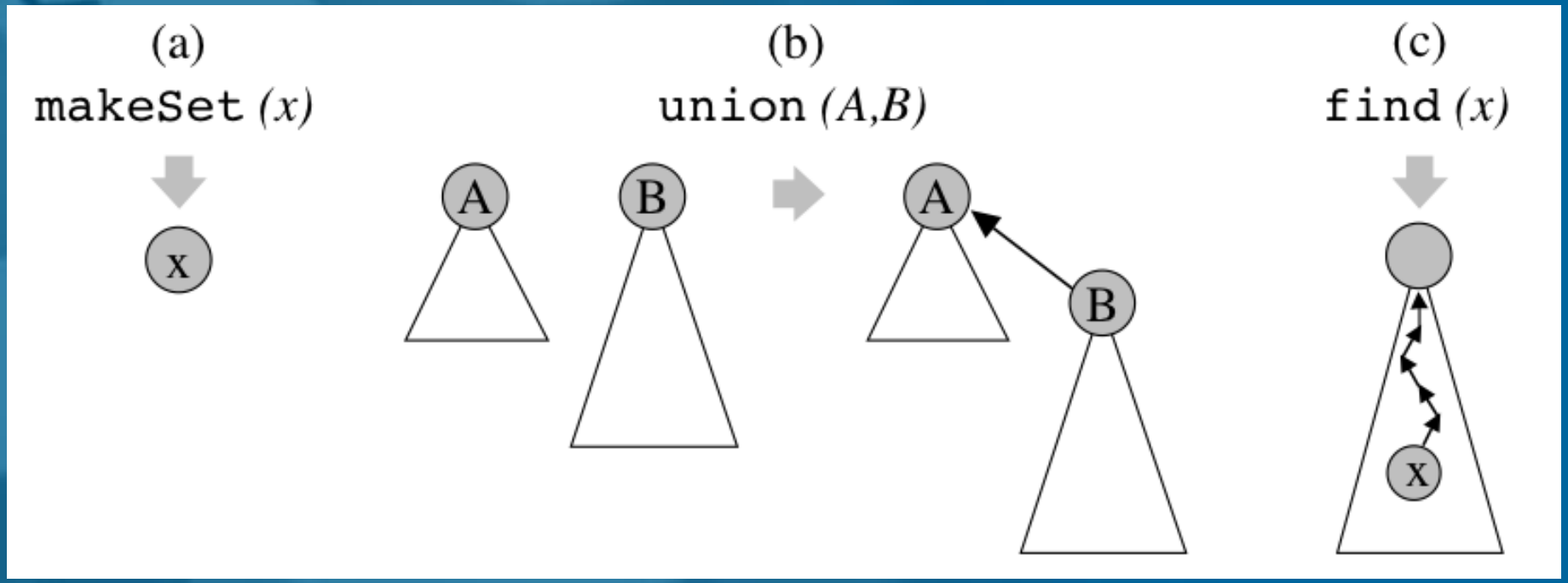
`union`$(name\ a, name\ b)$ $\quad T(n) = O(1)$

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

accede direttamente al nodo corrispondente ad 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.

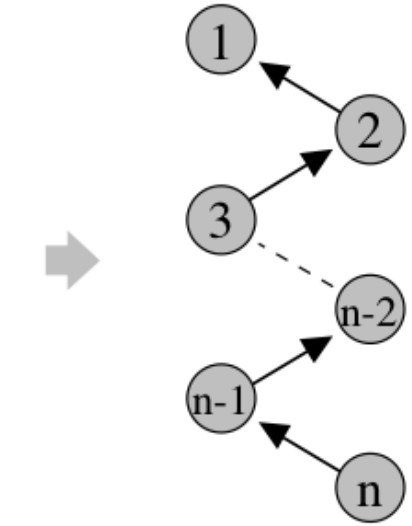
Esempio



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 diventa 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)

(Piccolo) Approfondimento

Domanda: quando è preferibile un approccio di tipo QuickFind rispetto ad un approccio di tipo QuickUnion?

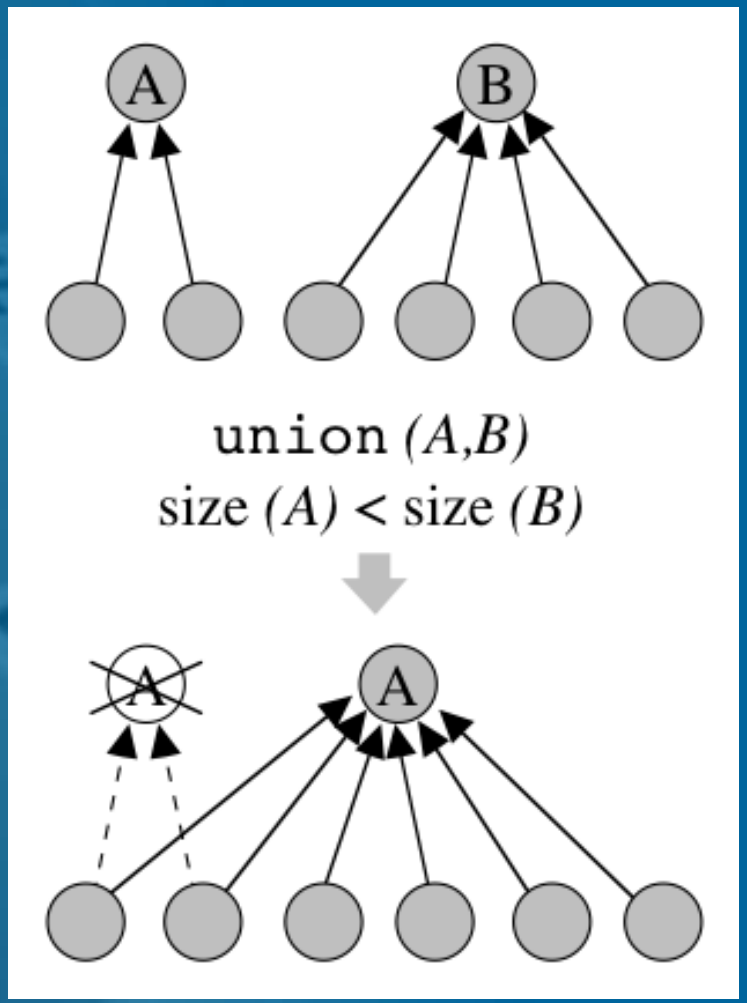
Risposta: Non esiste una risposta univoca: dipende dalla sequenza di operazioni, in particolare dal numero di operazioni di find e dalla loro collocazione rispetto alla sequenza di union. Se confrontiamo i casi peggiori dei due approcci, essi sono equivalenti quando $\Theta(m+n^2)=\Theta(mn)$, ovvero per $m=\Theta(n)$; quindi:

1. QuickUnion è più efficiente (nel caso peggiore) per $m=o(n)$, in quanto in tal caso $mn=o(m+n^2)$;
2. Viceversa, QuickFind è più efficiente (nel caso peggiore) per $m=\omega(n)$, in quanto in tal caso $m+n^2=o(mn)$.

Euristiche di bilanciamento nell'operazione union

Bilanciamento in alberi QuickFind

Union by size: associamo alla radice di ogni albero A un valore size(A) che è pari al numero di elementi contenuti nell'insieme (albero A); quindi, nell'unione di due 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



Realizzazione (1/3)

classe `QuickFindBilanciato` **implementa** `UnionFind`:

dati: $S(n) = \Theta(n)$

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 direttamente alla foglia contenente l'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 $O(n \log n)$)

Analisi ammortizzata (1/2)

Vogliamo dimostrare che se eseguiamo m find, n makeSet, e 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 tutte le operazioni di find e makeSet richiedono complessivamente tempo $\Theta(m+n)$
- Per analizzare le operazioni di union, quando creiamo un nuovo insieme, assegniamo al corrispondente elemento $\log n$ crediti $\Rightarrow$ in totale, assegniamo $n \log n$ crediti, e mostreremo che tali crediti copriranno **complessivamente** tutti i costi legati alle union

Analisi ammortizzata (2/2)

- Quando eseguiamo una union, ogni nodo che cambia padre pagherà il tempo speso (pari ad O(1)) con uno dei crediti assegnati al nodo
- Osserviamo ora che ogni nodo può cambiare al più 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!

$\Rightarrow$I log n crediti assegnati al nodo sono quindi sufficienti per pagare tutti i cambiamenti di padre, e l'intera sequenza di n union costa quindi O(n log n).

$\Rightarrow$L'intera sequenza di operazioni costa

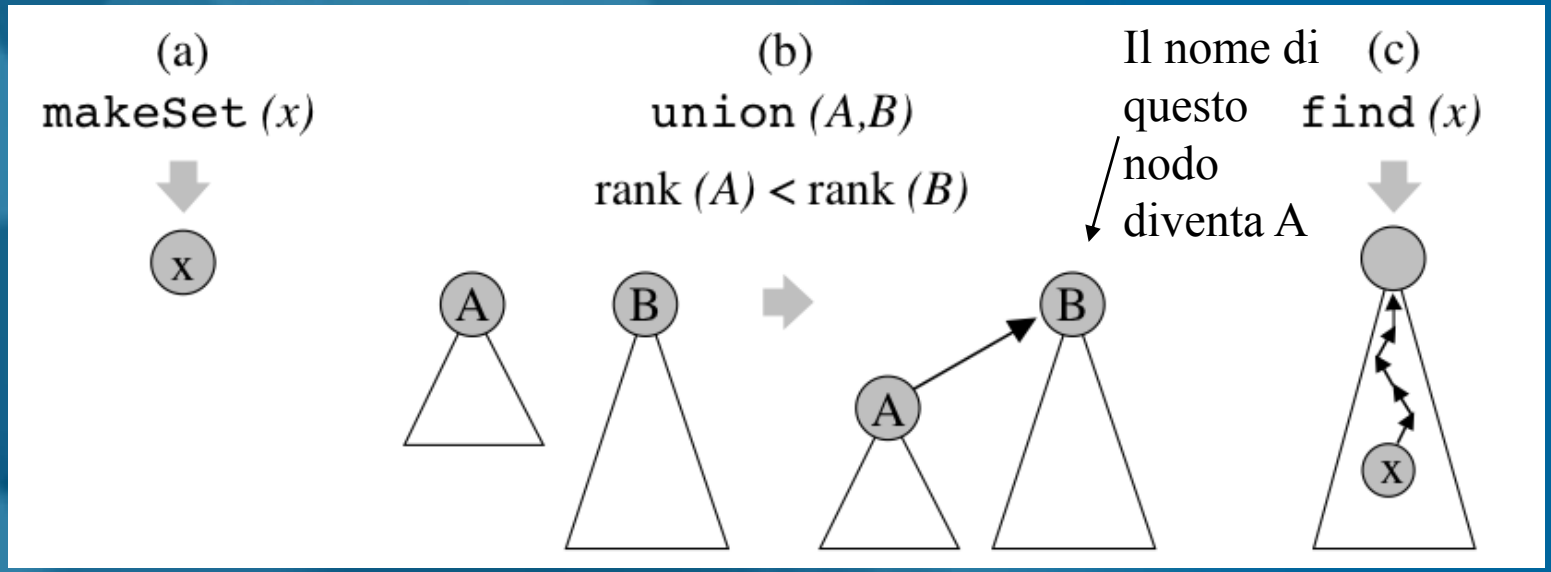
$$O(m+n+n \log n)=O(m+n \log n).$$

Costo della union nel caso peggiore

- Abbiamo appena visto che l'intera sequenza di n union costa O(n log n). Domanda: quanto può costare nel caso peggiore una singola operazione di union?
- Risposta: Θ(n), ad esempio si consideri il caso in cui si uniscono 2 insiemi contenti ciascuno n/2 elementi (o, più in generale, 2 insiemi contenenti Θ(n) elementi)

Bilanciamento in alberi QuickUnion

Union by rank (o by height): associamo alla radice di ogni albero A un valore rank(A) che è pari all'altezza dell'albero A; quindi, nell'unione di due insiemi A e B, rendiamo la radice dell'albero più basso figlia della radice dell'albero più alto, e se necessario modifichiamo la radice dell'albero ottenuto



Realizzazione (1/3)

classe `QuickUnionBilanciato` **implementa** `UnionFind`:

dati: $S(n) = \Theta(n)$

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

accede direttamente al nodo corrispondente ad 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.

Complessità computazionale

Vogliamo dimostrare che se eseguiamo m find, n makeSet, e 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 tutte le operazioni di union e makeSet richiedono complessivamente tempo $\Theta(n)$
- 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 $k>0$ 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, per ogni albero A, si ha che $|A| \geq 2^{rank(A)}$, e quindi $rank(A) \leq \log |A| \leq \log n$, 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 alla radice di ogni albero A un valore size(A) che è pari al numero di elementi contenuti nell'insieme (albero A); quindi, nell'unione di due insiemi A e B, rendiamo sempre la radice dell'albero con meno nodi figlia della radice dell'albero con più nodi, e se necessario modifichiamo la radice dell'albero ottenuto

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?