UNION FIND

[Deme, seconda edizione] cap. 9

Il problema

Mantenere una **collezione di insiemi disgiunti** (sottoinsiemi di un insieme-universo E) sulla quale siano possibili le seguenti operazioni:

union(A,B): fonde gli insiemi A e B in un unico insieme $A \cup B$ (i vecchi insiemi A e B vengono quindi **persi**)

find(x): restituisce **il nome dell'insieme contenente** l'elemento **x**

makeSet(x): crea il nuovo **insieme {x}**, avente x come unico elemento.

Nota: Gli **insiemi** rimangono sempre **disgiunti**, quindi ad ogni istante ogni elemento appartiene a non più di un insieme.
Nota2: Per distinguere tra diversi insiemi senza dover elencare tutti i loro elementi, possiamo individuare (per ogni insieme) un elemento **rappresentante**, cosicché **union(a,b)** fonda i due insiemi rappresentati dagli elementi a e b

Esempio

Elementi: società che hanno svolto un ruolo nella storia economica

Insiemi o rappresentanti: società esistenti ad un certo istante

{**Sip**} {**Stipel**} {**Teti**} {**Fiat**} {**Lancia**} {**AlfaRomeo**} {**Ferrari**} {**Innocenti**}

union(Sip, Teti); union(Sip, Stipel); union(Lancia, Innocenti);

{**Sip**, Stipel, Teti}, {**Fiat**}, {**Lancia**, Innocenti}, {**Ferrari**}

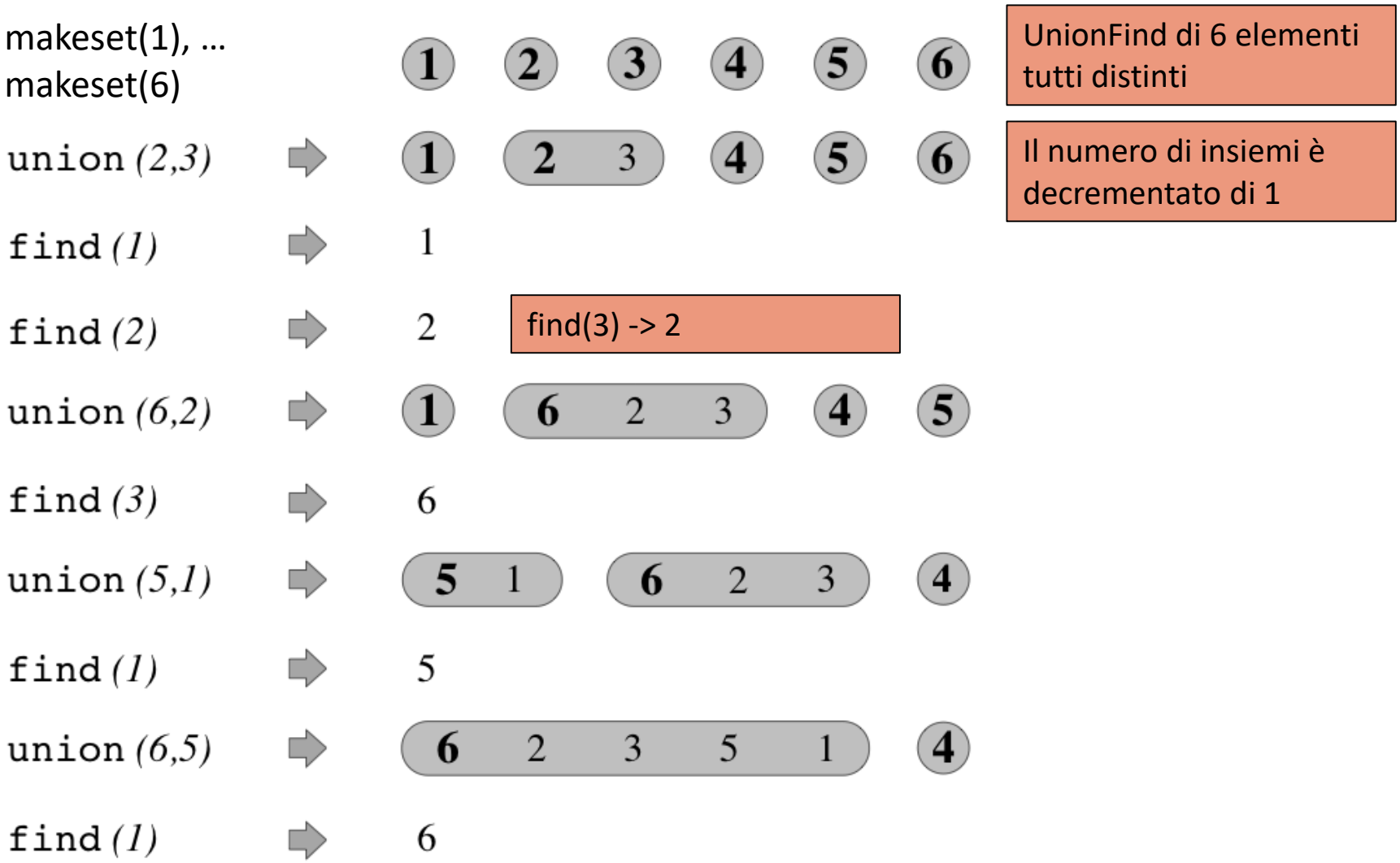
makeSet(Telecom); union(Telecom, Sip); union(Fiat, Lancia); union(Fiat, Ferrari);

{**Telecom**, Sip, Stipel, Teti}, {**Fiat**, Lancia, Innocenti, Ferrari}

eccetera.

L'elemento in grassetto è il rappresentante di un insieme

Esempio (generico)



Nota: raffinamento della definizione di Union

Per comodità, invece che definire union(a,b) come l'unione degli insiemi **rappresentati** da a e b, definiamo union come l'unione degli insiemi **contenenti** a e b.

In questo modo scriviamo

union(a,b)

intendendo

union(find(a),find(b))

Implementazione

Esistono 2 tipi di approcci elementari:

- Quelli che **privilegiano** un'esecuzione efficiente dell'**operazione di find** (**QuickFind**)
- e Quelli che **privilegiano** un'esecuzione efficiente dell'**operazione di union** (**QuickUnion**)

ATTENZIONE: sono quelli che vedremo noi. Ce ne possono essere tanti differenti.

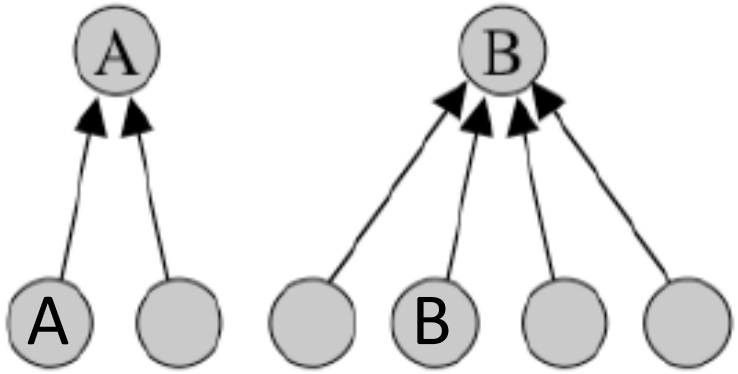
QuickFind

Gli approcci di tipo QuickFind utilizzano **alberi con 2 livelli** per rappresentare le UnionFind.

La **radice** di ogni albero contiene il **rappresentante di un insieme**.

Le **foglie** rappresentano **gli elementi dell'insieme**.

Nota: il rappresentante è contenuto **sia** nella **radice** che in una **foglia**.

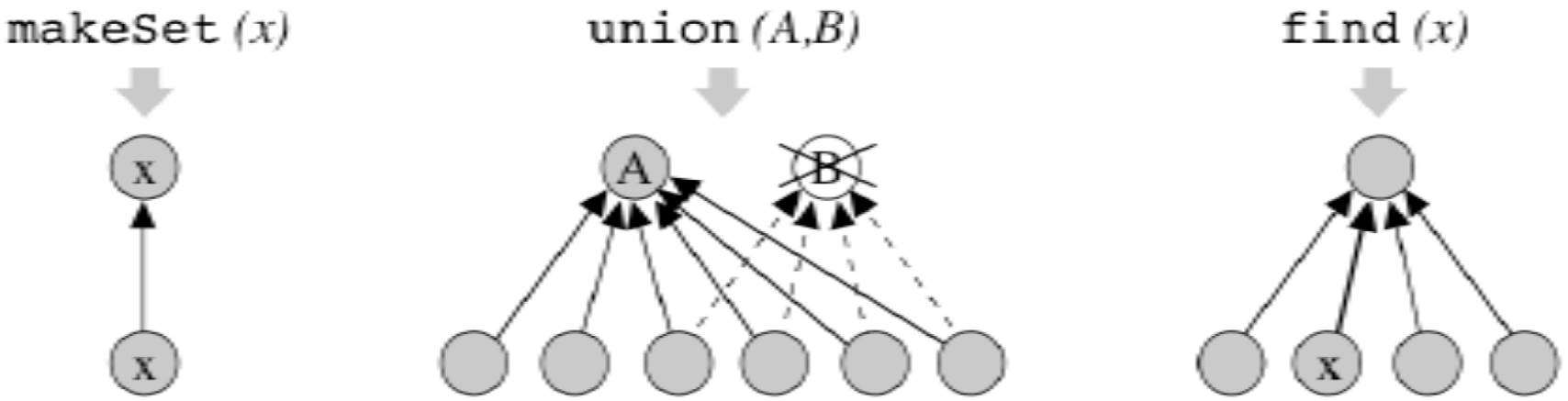


Operazioni su QuickFind

makeSet(x) crea un nuovo albero composto da una **radice ed una foglia**. In entrambi posiziona x. Costo **O(1)**.

find(x) accede alla foglia corrispondente all'elemento x. Risale di **un livello** nell'albero incontrando la **radice** (l'albero ha solo 2 livelli) e restituisce l'elemento contenuto nella radice. Costo **O(1)**

union(a,b) considera l'albero A contenente a e B contenente b. Per ogni foglia di B, **sostituisce il puntatore al padre** con un **puntatore alla radice di A**. Cancella la radice di B. Costo **O(n)**



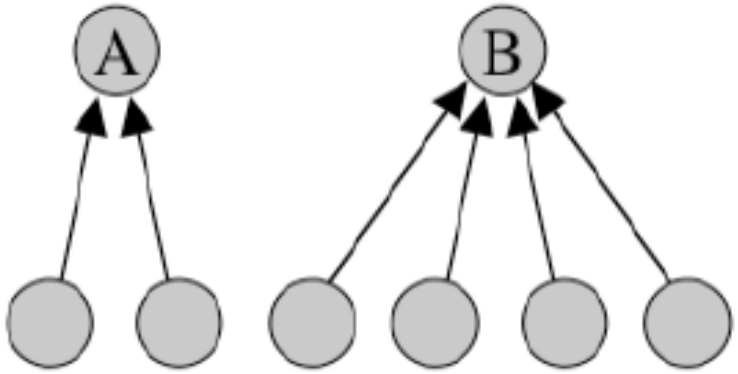
QuickUnion

Gli approcci di tipo QuickUnion utilizzano alberi con **più di 2 livelli** per rappresentare le UnionFind.

La **radice** di ogni albero contiene il **rappresentante di un insieme**.

I **nodi** (tutti) rappresentano **gli elementi dell'insieme**.

Nota: in questo caso il rappresentante sta **solo nella radice**.

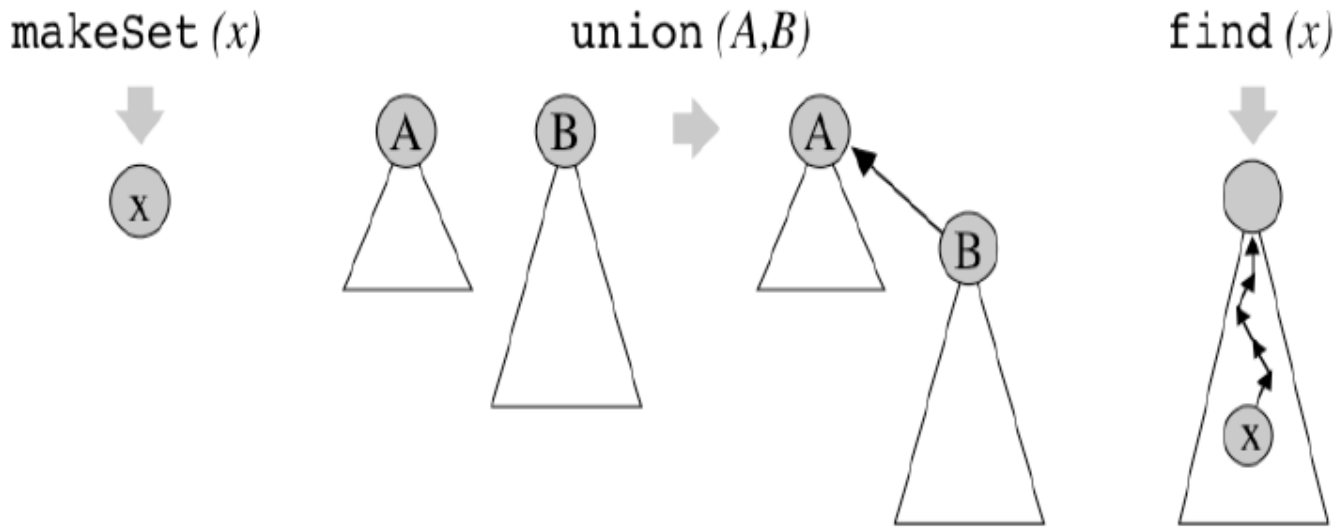


Operazioni su QuickUnion

makeSet(x) crea un nuovo albero composto da **un unico nodo** contenente x. Costo **O(1)**.

find(x) accede alla foglia corrispondente all'elemento x. Risale **fino alla radice** dell'albero e restituisce l'elemento contenuto nella radice. Costo **O(n)** **(Perché?)**

union(a,b) considera l'albero A contenente a e B contenente b. Rende **la radice di B figlio della radice di A**. Costo **O(1)**



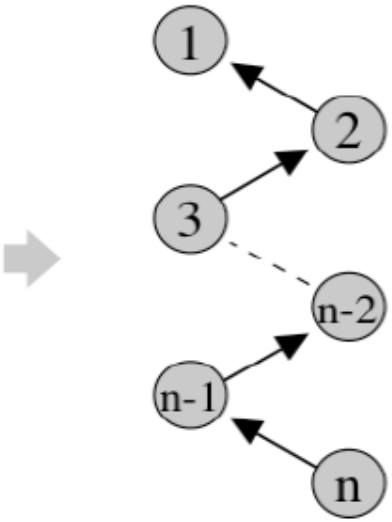
Perché O(n)?

In molti alberi (**bilanciati**) visti fino ad ora abbiamo sempre considerato le operazioni di «risalita» da una foglia alla radice di costo O(log n). Perché qui è O(n)?

Perché in questo caso **non abbiamo nessun bilanciamento sull'albero**, quindi nel caso peggiore dobbiamo considerare alberi «lineari» in altezza.

Questa sequenza di union produce un albero lineare in altezza

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



Nota sulla complessità

In generale, le UnionFind si usano per gestire insiemi che vengono via via uniti, fino a risultare in un unico insieme. Durante queste unioni, vengono fatte una serie di find. In generale, si considerano

n makeSet,

n-1 union (n-1 è il limite massimo di union eseguibili dopo n makeSet) e

m find (il numero di find invece non dipende dalle makeSet)

Per questo motivo, non è «saggio» concentrarsi sul costo nel caso peggiore di una singola operazione.

Ciò che ci interessa è il **costo totale** di queste operazioni, o in altre parole il **costo ammortizzato** di ogni singola operazione su una (qualsiasi) **sequenza** di questo tipo.

Euristiche di Bilanciamento e Compressione

Le ottimizzazioni proposte cercano di ridurre il costo **ammortizzato** delle operazioni su una sequenza di n makeSet, n-1 union ed m find.

Ci concentreremo su

- Il **bilanciamento** dell'operazione di union
  - Per algoritmi di tipo **QuickFind** (con una union inefficiente), cerchiamo di **migliorare le prestazioni della union**
  - Per algoritmi di tipo **QuickUnion** (con una find inefficiente), cerchiamo di **mantenere alberi con meno livelli** (così da migliorare la find)
- La **compressione** in fase di find (**comprimiamo** gli alberi «approfittando» delle risalite fatte per la find)

Bilanciamento su QuickFind

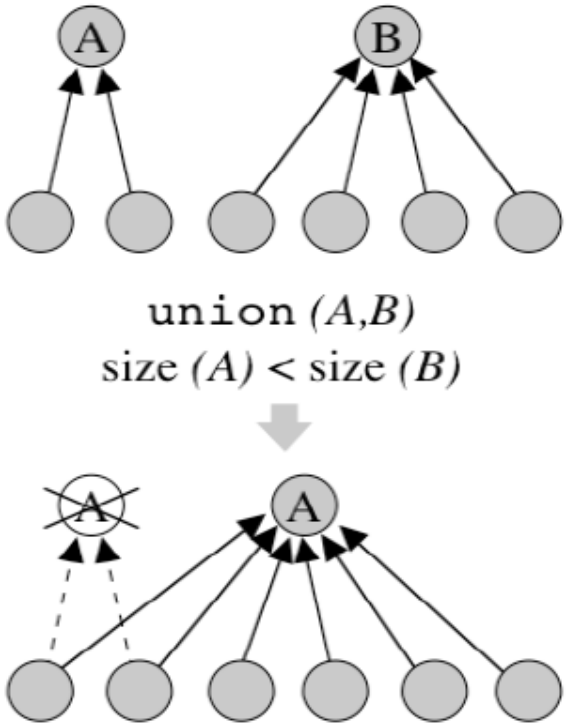
L'operazione più costosa dell'approccio QuickFind è la union.

Per migliorarla, invece che operare alla cieca su A e B, andiamo a considerare come **insieme primario quello con cardinalità maggiore**, e **modifichiamo il padre delle foglie dell'altro insieme**.

Introduciamo un valore **size(x)** (inizializzato da makeSet a 1), che memorizza **cardinalità dell'insieme x**.

Quando facciamo union(A,B), usiamo **size** per identificare **l'insieme** primario, poi **scriviamo la radice di A nella radice dell'albero risultante**.

In più **size(A) = size(A)+size(B)**.



Analisi Ammortizzata

Qual è la complessità della gestione di una QuickFind bilanciata?

(analizziamo il costo di m find, n makeSet e n-1 union)

m find hanno costo **O(m)**

n makeSet e n-1 union analizzato con il **metodo dei crediti**

Metodo dei Crediti (o Accantonamenti)

(vedi [Deme] Sez. 2.7.1 o [Cormen] Sez. 17.1)

Utilizzato per determinare il costo **ammortizzato** di una sequenza di operazioni, senza andare nel dettaglio delle dipendenze tra di loro.

1 credito vale **O(1) passi di esecuzione**

Le funzioni **meno costose** «**depositano**» dei crediti sugli oggetti (caricandosi di un costo maggiore di quello che fanno effettivamente).

I crediti così depositati possono essere «**prelevati**» dalle funzioni **più costose.**

Il costo ammortizzato di ogni operazione nella sequenza è poi dato dalla **somma** di tutti i costi **diviso il numero di operazioni**.

NOTA: L'importante è definire i crediti in maniera che qualsiasi sequenza lecita di operazioni non porti i crediti accumulati ad essere negativi.

Metodo dei Crediti - esempio

DOBBIAMO CALCOLARE LA COMPLESSITA' DI:

Push (di un elemento) e Multipop (di k elementi) su uno stack:

Push ha complessità **O(1)** e, nel caso peggiore, **Multipop O(n)**.

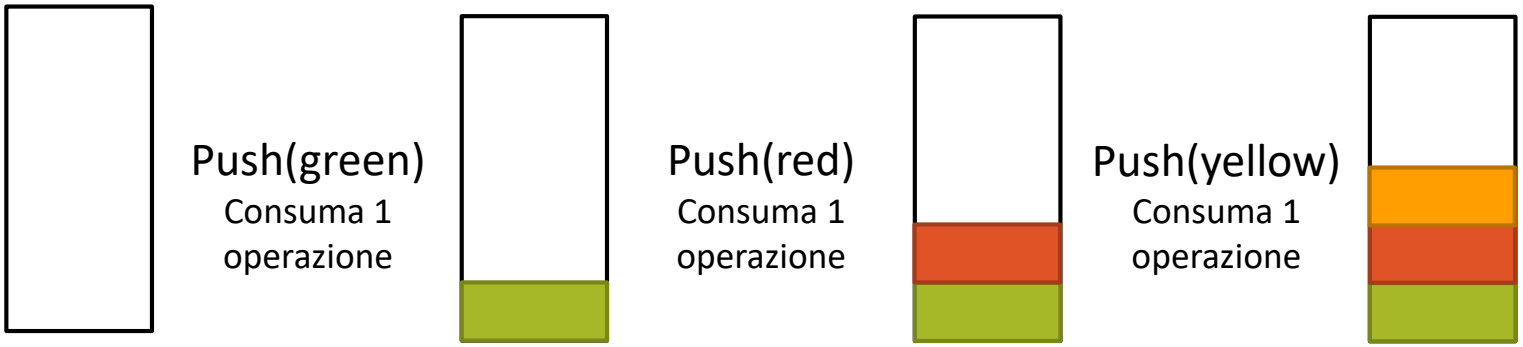
CON IL METODO DEI CREDITI:

Carichiamo Push di un **costo aggiuntivo** di una operazione.
Il suo costo ora è **2 crediti** (ancora **costante**), di cui 1 consumato dall'operazione stessa, l'altro lasciato sull'oggetto inserito.
(nota: per ogni elemento nello stack viene accumulato esattamente 1 credito)

A Multipop assegniamo il costo di **0 crediti**. Ogni volta che un'operazione Multipop rimuove un elemento, per farlo consuma il suo credito accumulato. In questo modo il costo di Multipop è **costante**.

Quindi il costo ammortizzato di ogni operazione in una **sequenza ammissibile** di Push e Multipop è **costante**, o meglio **O(1)**.

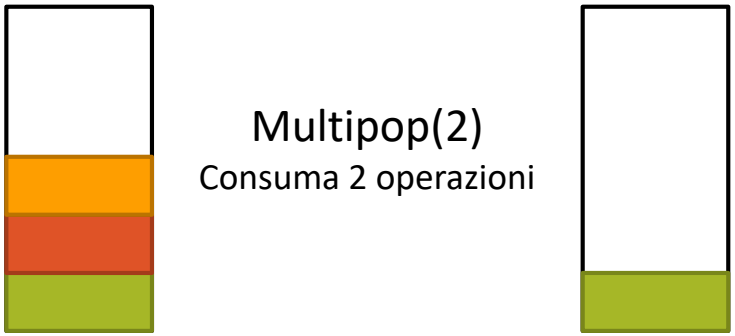
Stack senza crediti



Push(green)
Consuma 1 operazione

Push(red)
Consuma 1 operazione

Push(yellow)
Consuma 1 operazione

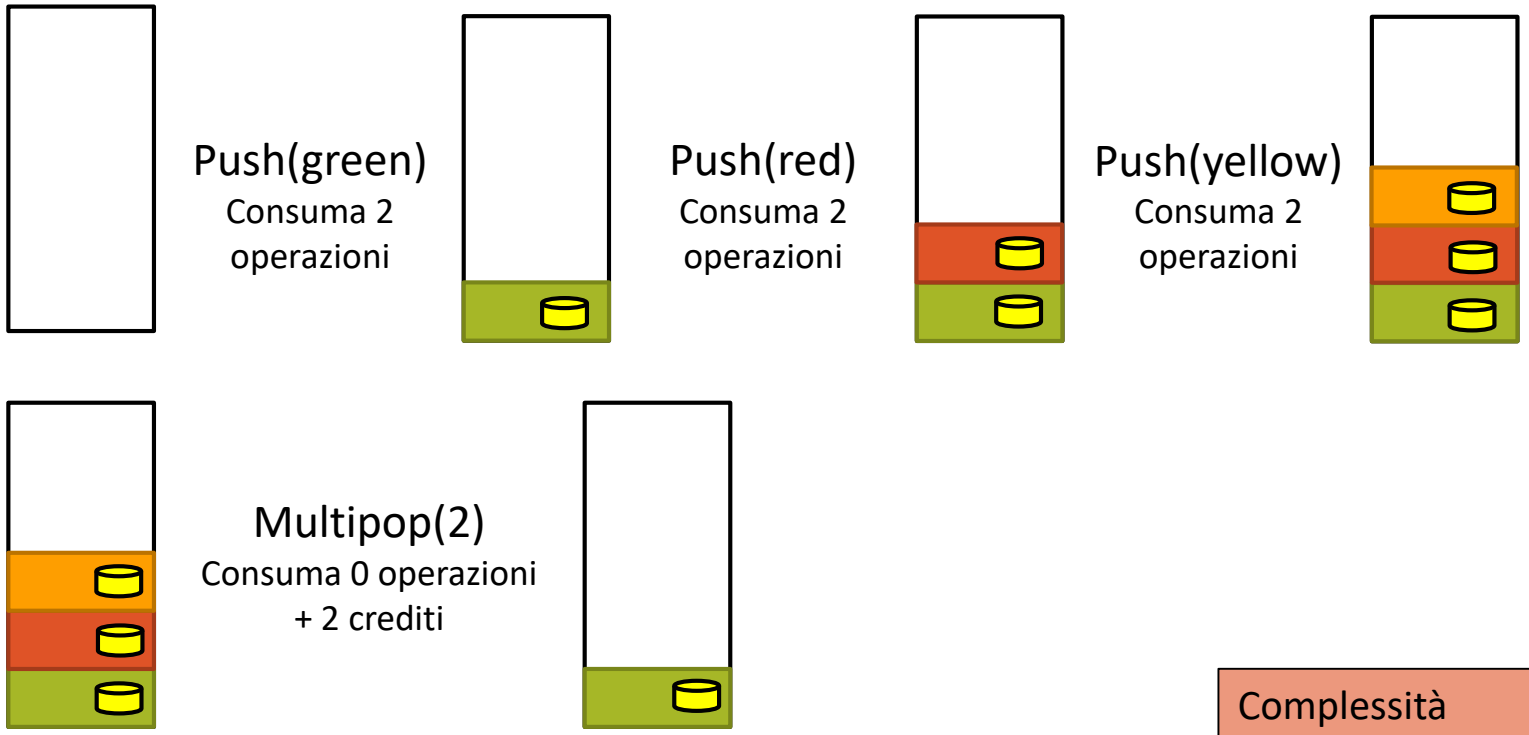


Multipop(2)
Consuma 2 operazioni

Nota: non posso fare un altro Multipop(2)!

Complessità
Push O(1)
Multipop O(n)

Stack con crediti



Complessità
Push O(2) = O(1)
Multipop O(0) = O(1)

Analisi Ammortizzata - continua

n makeSet e n-1 union analizzato con il **metodo dei crediti**:

NUMERO MASSIMO CAMBI PADRE: prima di tutto, osserviamo che ad ogni union, se una foglia cambia padre, il nuovo insieme che la contiene sarà grande **almeno il doppio di quello di partenza** (a causa del bilanciamento). Quindi una foglia dopo **k cambi di padre** apparterrà ad un insieme di 2^k elementi. Siccome il numero totale di elementi è n, $2^k \leq n$, quindi $\mathbf{k \leq \log_2 n}$.

DEPOSITO CREDITI: assegniamo ad ogni makeSet un costo aggiuntivo di $\mathbf{\log_2 n}$ crediti. In totale, con n makeSet, accumuliamo $\mathbf{n \log_2 n}$ **crediti**.

COSTO UNION: ogni union, per ogni foglia che cambia padre, consuma **1 credito** per il costo dell'operazione di **cambio del padre**. Ma poiché i cambi di padre sono limitati a $\mathbf{\log_2 n}$ **per ogni foglia**, si consumano in totale $\mathbf{n \log_2 n}$ **crediti**.

Di conseguenza, il costo ammortizzato di ogni operazione in una sequenza di n makeSet e n-1 union è $O(\log_2 n)$, quindi in totale $\mathbf{O(n \log_2 n)}$.

Costo m find, n makeSet e n-1 union: $\mathbf{O(m + n \log_2 n)}$

Bilanciamento su QuickUnion

2 ottimizzazioni possibili sulla union per approcci QuickUnion, per mantenere **alberi con meno livelli**:

- **Union by rank:** nell'unione degli insiemi A e B, rendiamo la radice dell'albero **più basso** figlia della radice dell'albero più alto
- **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

(Quick)Union by rank

Introduciamo un parametro **rank** (rank(x) inizializzato a 0 da makeSet(x)) che tiene conto dei **livelli dell'albero**.

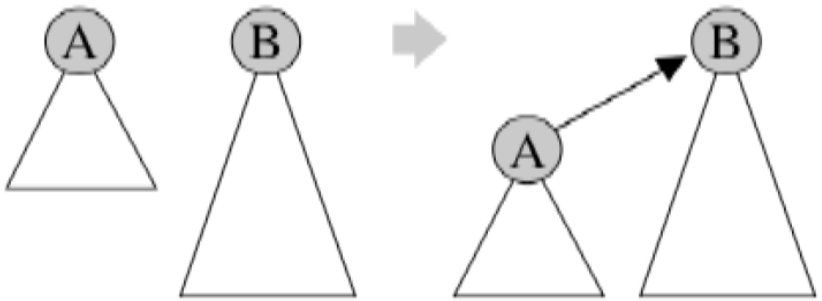
Quando facciamo una union, se **rank(B)<rank(A)** rendiamo la radice di B figlio della radice di A.

Se **rank(A)<rank(B)**, rendiamo la radice di A figlio della radice di B, e scambiamo le due radici. Poi rank(A) = rank(B).

Se **rank(A)=rank(B)**, rendiamo la radice di B figlio della radice di A e rank(A) = rank(A)+1

union (A,B)

rank (A) < rank (B)



Analisi - I

Poiché in questa ottimizzazione vogliamo ottimizzare l'operazione di find, dobbiamo dimostrare qual è il limite massimo di altezza dell'albero per numero di nodi, o, in alternativa, qual è il limite minimo di nodi di un albero con una certa altezza.

Dimostriamo che **l'albero con radice x ha almeno $2^{\text{rank(x)}}$ nodi**.

DIMOSTRAZIONE (per induzione sul numero di union).

Base: rank(A) = 0 e 1 solo nodo. $|\mathbf{A}| = 1 \geq 2^0 = \mathbf{2^{rank(A)}}$.

Dopo ogni union (passo, con ip. Ind. $|\mathbf{L}| \geq \mathbf{2^{rank}}(\boldsymbol{L})$):

Se **rank(A) > rank(B)**:

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

Se **rank(A) > rank(B)** simmetrico al precedente

Se **rank(A) = rank(B)**:

$$|\mathbf{A} \cup \mathbf{B}| = |\mathrm{A}| + |\mathrm{B}| \geq 2^{\mathrm{rank(A)}} + 2^{\mathrm{rank(B)}} = 2 * 2^{\mathrm{rank(A)}} = 2^{\mathrm{rank(A)}+1} = \mathbf{2^{rank(A \cup B)}}$$

Analisi - II

Ma il numero massimo di nodi di un albero è n (il numero di makeSet).

Quindi $2^{\mathrm{rank(x)}} \leq \mathrm{n}$.

Quindi $\mathbf{rank(x) \leq \log_2 n}$.

Cioè, **l'altezza di un albero è limitata superiormente da $\log_2 \mathrm{n}$** con n = numero di makeSet.

La find richiede tempo **O(log n)**

(Quick)Union by size

Utilizziamo lo stesso parametro **size** visto per la QuickFind.

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

Quando eseguiamo una union,

se **size(B) ≤ size(A)** rendiamo la radice di B figlio della radice di A.

se **size(A) < size(B)**, rendiamo la radice di A figlio della radice di B, e scambiamo le due radici.

Infine, **size(A) = size(A) + size(B)**

Anche in questo caso, (si può dimostrare che) costruiamo alberi di altezza logaritmica.

Compressione nell'operazione find

Le euristiche di compressione vengono applicate **durante l'esecuzione di una find**, e hanno lo scopo di **diminuire** (ulteriormente) **l'altezza di un albero** per algoritmi di tipo QuickUnion (quelli di tipo QuickFind non possono essere ulteriormente diminuiti).

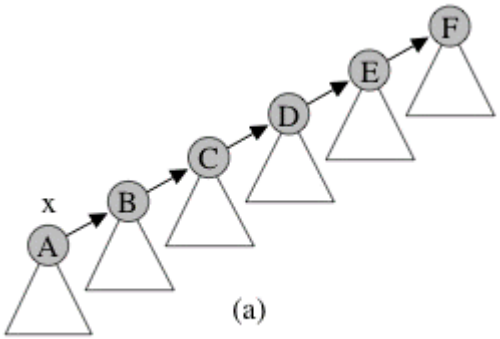
Ne vediamo 3 tipi:

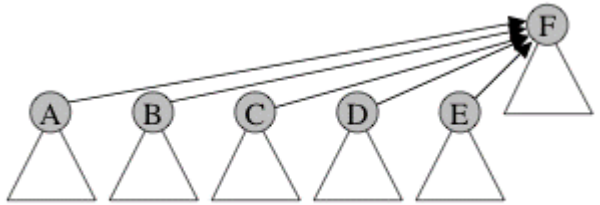
- Path **compression**
- Path **splitting**
- Path **halving**

Path Compression

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

rendi il nodo **u_i figlio della radice u_{t-1}**, per ogni $i \leq t-3$



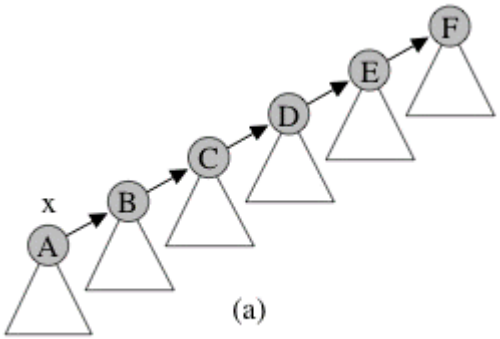


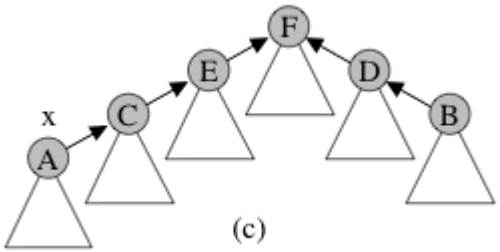
«appiattisce» molto velocemente l'albero. Tuttavia ad ogni find si deve portare dietro un gran numero di puntatori.

Path Splitting

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

rendi il nodo **u_i figlio del nonno u_{i+2}**, per ogni $i \leq t-3$



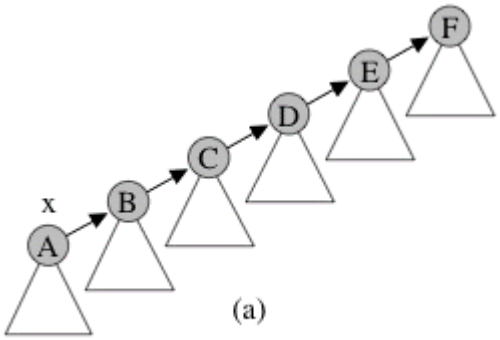


È più lento ad appiattire rispetto alla path compression, ma ha bisogno di solo 2 puntatori.

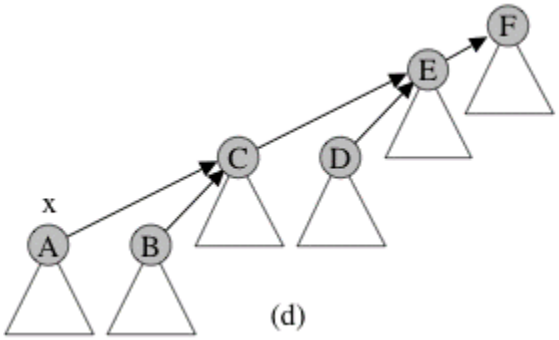
Path Halving

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

rendi il nodo **u_{2i} figlio del nonno u_{2i+2}**, per ogni $i \leq \left\lfloor \frac{(t-1)}{2} \right\rfloor - 1$



(a)



(d)

Ha bisogno di solo 1 puntatore.

Combinazione delle euristiche

È possibile **combinare** ciascuna euristica di bilanciamento con ciascuna euristica di compressione

(union by rank o union by size)

×

(path splitting, path compression, o path halving)

Ottenendo 6 algoritmi diversi.

Esempio: QuickUnion con union by rank e path compression

Sia n il numero massimo di elementi

Sia **π il vettore (foresta) dei padri** definito come segue:

- **π[i] = -1** se **i** non appartiene a **nessun insieme**
- **π[i] = i** se i è la radice del suo albero (**i** è il **rappresentante del suo insieme**)
- **π[i]** = j se **i** è **figlio di j**

Sia rank un vettore di n interi tale che **rank[i] è il rank di i se π[i] = i**

QuickUnion con union by rank e path compression - II

Definiamo le operazioni come segue:

makeSet(x)

```
π[x] <- x
rank[x] <- 0
```

find(x)

```
if π[x] ≠ x
    π[x] <- find(x)
return π[x]
```

union(x,y)

```
x <- find(x)
y <- find(y)
if rank[x] > rank[y]
    π[y] <- x
else π[x] <- y
if rank[x] = rank[y]
    rank[y] <- rank[y] + 1
```

Se n non è noto a priori, possiamo utilizzare **array dinamici** per π e rank. In questo modo il costo ammortizzato di makeSet rimane O(1).

Analisi - La funzione log*n

Definiamo log*n come

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

Informalmente, log*n ci dice **quante volte dobbiamo applicare la funzione log su n affinché il risultato sia minore di 1**.

Notiamo che cresce molto lentamente

log*x = 0	$0 < x \leq 1$	log*x = 3	$4 < x \leq 16$
log*x = 1	$1 < x \leq 2$	log*x = 4	$16 < x \leq 65536$
log*x = 2	$2 < x \leq 4$	log*x = 5	$65536 < x \leq 2^{65536}$

Analisi - II

Si può dimostrare (noi non lo facciamo) che, **combinando le euristiche di bilanciamento e compressione**, una qualunque sequenza di n makeSet, m find ed n-1 union può essere eseguita in tempo

O((n+m) log*n)

Con log*n che è «quasi» costante, possiamo dire che eseguiamo 2n – 1 + m operazioni in un tempo «circa» **O(n+m)**.

Quindi, con O(n+m) operazioni in tempo O(n+m), possiamo dire che, **in una sequenza di operazioni come sopra**, ogni operazione ha un costo ammortizzato di **O(1)**.

Cosa devo aver capito fino ad ora

- Problema UnionFind
- Approcci base
- Ottimizzazioni con euristiche e loro impatto
- Ottimizzazioni con combinazioni di euristiche e costo

…se non ho capito qualcosa

- Alzo la mano e chiedo
- Ripasso sul libro
- Chiedo aiuto sul forum
- Chiedo o mando una mail al docente