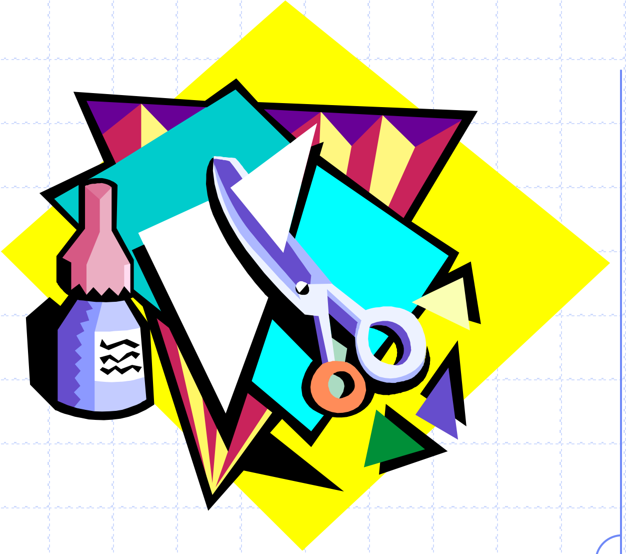
Partizioni con operazioni di union e find

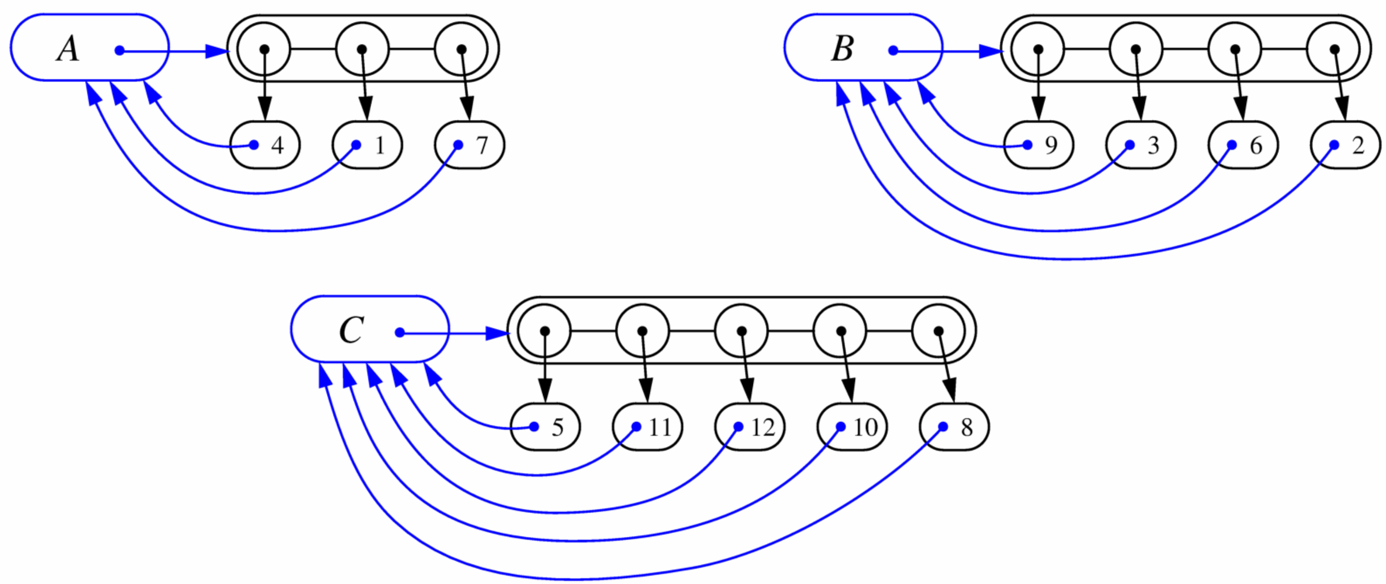


Partizioni con operazioni di union e find (§ 11.6.2)

- Una partizione è una collezione di insiemi *disgiunti*
- Operazioni
  - makeSet(x): crea un insieme composto dal solo elemento x e restituisci la posizione che memorizza x in questo insieme
  - union(A,B): Restituisci l'insieme A U B, distruggendo i vecchi A e B
  - find(p): Restituisci l'insieme contenente l'elemento in posizione p

Implementazione basata su lista

- Ciascun insieme è memorizzato in una sequenza rappresentata attraverso una lista collegata
- Ciascun nodo mantiene un oggetto contenente l'elemento dell'insieme e un riferimento al nome dell'insieme

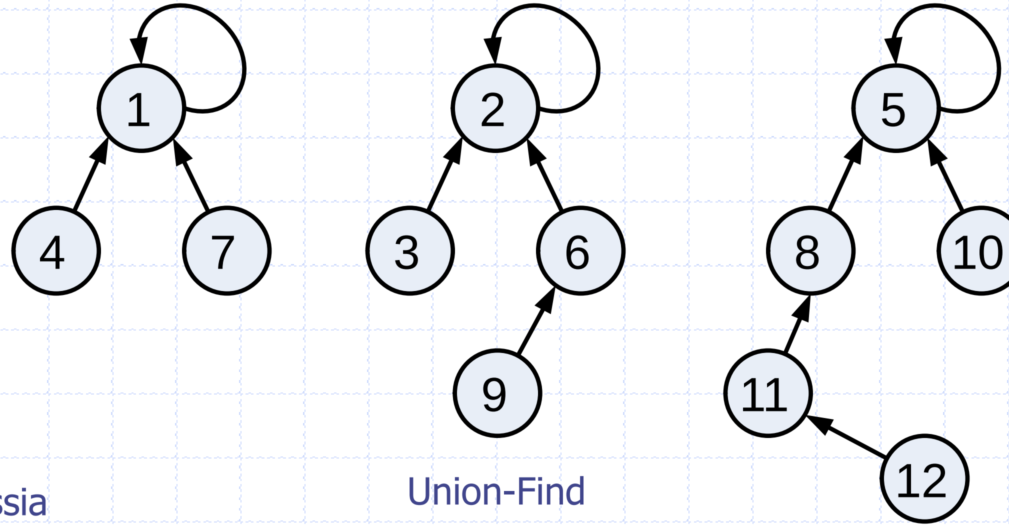


Analisi della rappresentazione basata su lista

- Nel fare unioni, muovere sempre elementi dall'insieme più piccolo a quello più grande
  - Ogni volta che un elemento viene mosso questo raggiunge un insieme di dimensione almeno doppia
  - Di conseguenza ogni elemento può essere mosso al più O(log n) volte, essendo n il numero totale di elementi
- Il tempo totale per eseguire n operazioni partendo da una partizione vuota è O(n log n)
  - O(log n) *ammortizzato* per operazione

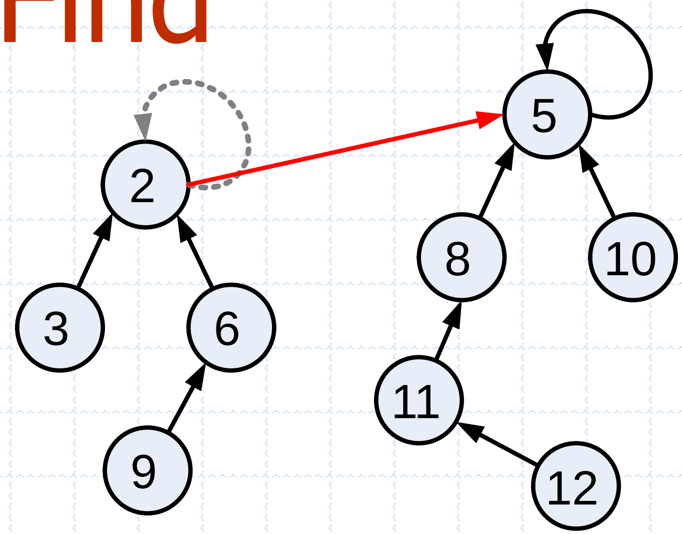
Implementazione basata su albero (§ 11.6.3)

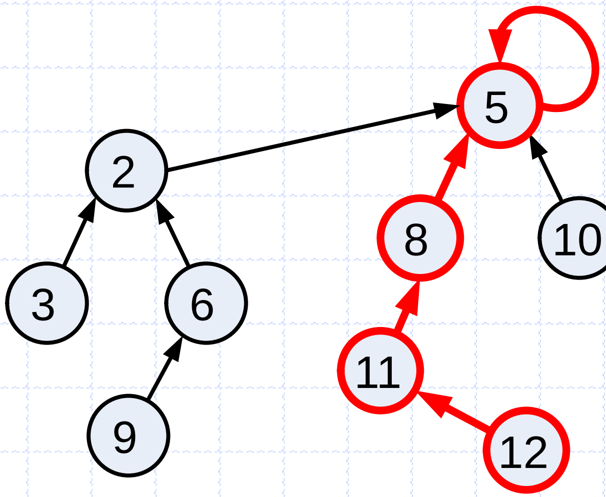
- Ciascun insieme è rappresentato da un albero i cui nodi contengono un elemento e il riferimento al genitore
- La radice dell'albero contiene un riferimento a se stessa
- L'elemento nella radice di ciascun albero è usato come nome o rappresentante dell'insieme
  - Ad esempio, gli insiemi "1", "2" e "5":



Operazioni Union-Find

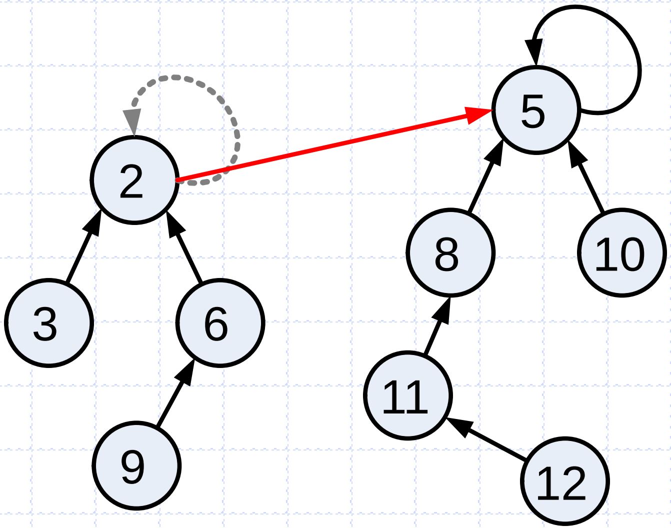
- Per eseguire una unione, basta assegnare al riferimento presente nella radice di uno dei due alberi, la posizione dell'altro sottoalbero della radice dell'altro
- Per eseguire una find, basta seguire i riferimenti, partendo dal nodo assegnato, fino ad arrivare a un nodo che punta a se stesso





Euristica 1: union-by-size

- Union-by-size:
  - a ciascuna unione, si fa puntare la radice dell'albero con meno nodi a quella dell'albero con più nodi
- Implica tempo O(n log n) per eseguire n operazioni (union o find)
  - Ogni volta che seguiamo un puntatore, raggiungiamo un sottoalbero di dimensione almeno doppia di quella del precedente sottoalbero
  - Ne segue che saranno seguiti al più O(log n) per ciascuna find

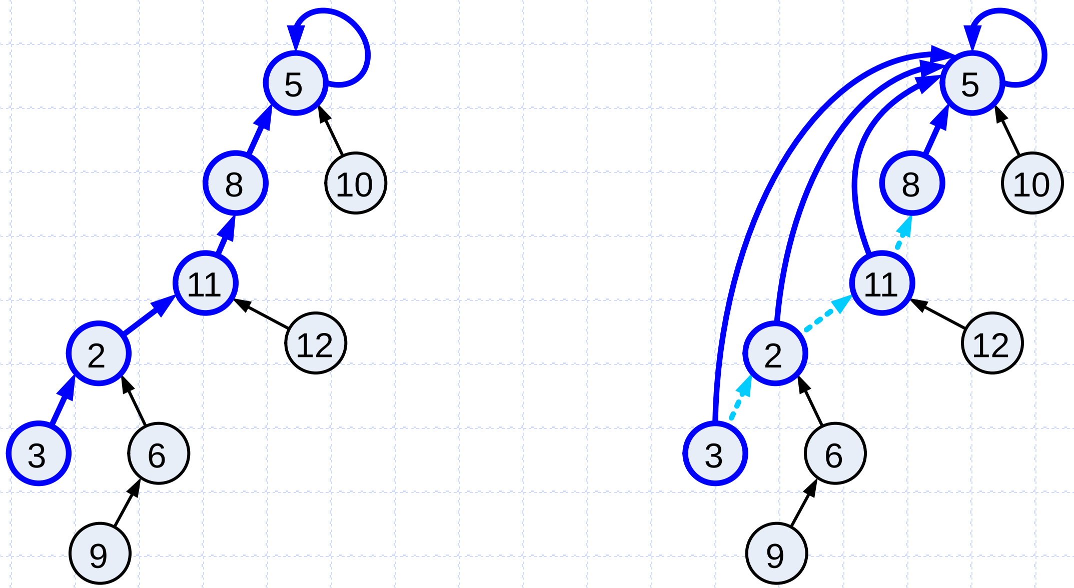


Implementazione di union-by-size

- Ogni nodo viene espanso per includere una informazione addizionale
- Per semplicità, invece di gestire il numero di nodi del suo sottoalbero, si mantiene un rank, ovvero un upper bound all'altezza del sottoalbero
- Diviene union-by-rank: la radice con rank minore viene fatta puntare a quella con rank maggiore
  - incrementa il rank della radice dell'albero unione se i rank delle due radici erano uguali prima dell'operazione

Euristica 2: path-compression

- Path-compression:
  - ad ogni find, modifica i puntatori dei nodi attraversati, facendo in modo che tali nodi puntino direttamente alla radice



- L'uso delle due euristiche garantisce tempo O(m α(n)) per eseguire m operazioni su n elementi (prova complessa), dove α(n) è l'inverso della funzione di Ackermann

Funzione di Ackermann

da Wikipedia

$f(0, y, z) = y + z$

$f(1, y, z) = y \times z$ (mediante iterazione di $z + z + z + \ldots$ per y volte)

$f(2, y, z) = z^y$ (mediante iterazione di $z \times z \times z \times \ldots$ per y volte)

$f(3, y, z) = z^{z^{z^{\cdot^{\cdot^{\cdot}}}}}$ (y volte)(mediante iterazione di $z^{z^{z^{\cdot^{\cdot^{\cdot}}}}}$ per y volte e quindi mediante iterazione di $y \times z$ e quindi mediante iterazione di y+z)

Risulta quindi una funzione con una complessità estremamente elevata anche per valori di input semplici.

per ogni n concepibile, α(n) < 5

pseudo-codice

da *Introduction to Algorithms*, di Cormen, Leiserson, Rivest, Stein

MAKE-SET(*x*)
1 $p[x] \leftarrow x$
2 $rank[x] \leftarrow 0$

UNION(*x*, *y*)
1 LINK(FIND-SET(*x*), *FIND-SET(y)*)

pseudo-codice (cont.)

```
LINK(x, y)
if rank[x] > rank[y]
      then  p[y] ← x
      else  p[x] ← y
            if rank[x] = rank[y]
                  then rank[y] ← rank[y] + 1
```

```
FIND-SET(x)
if x ≠ p[x]
      then p[x] ← FIND-SET(p[x])
return p[x]
```