Union-find

Luca Becchetti

Presentazione tratta dalle slide che accompagnano il testo Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Astrazione Union-find

- Oggetti
- Insiemi *disgiunti* di oggetti
- Si vogliono implementare efficientemente le seguenti operazioni
  - makeCluster(x): crea l’insieme {x}
  - union(a, b): sostituire agli insiemi contenenti gli oggetti a e b la loro unione
  - find(x): restituisce il *leader* dell’insieme contenente x
    - Si veda più avanti per il significato di leader

Applicazioni

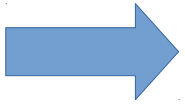
- Mantenere le componenti connesse di un grafo → v. più avanti nel corso
- Algoritmi per Minimum Spanning Tree → v. più avanti nel corso

Implementazione con sequenze

Rappresentazione con array

- Per semplicità assumiamo che gli oggetti siano interi
  - Altrimenti, usiamo una tabella
- Ad ogni oggetto i associamo id[i], che ne identifica l'insieme di appartenenza
  - Oggetti appartenenti allo stesso sottoinsieme hanno stesso valore di id[i]
  - Tipicamente, id[i] = <uno degli elementi appartenenti al sottoinsieme di i}>
  - makeCluster(x) → id[x] = x

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

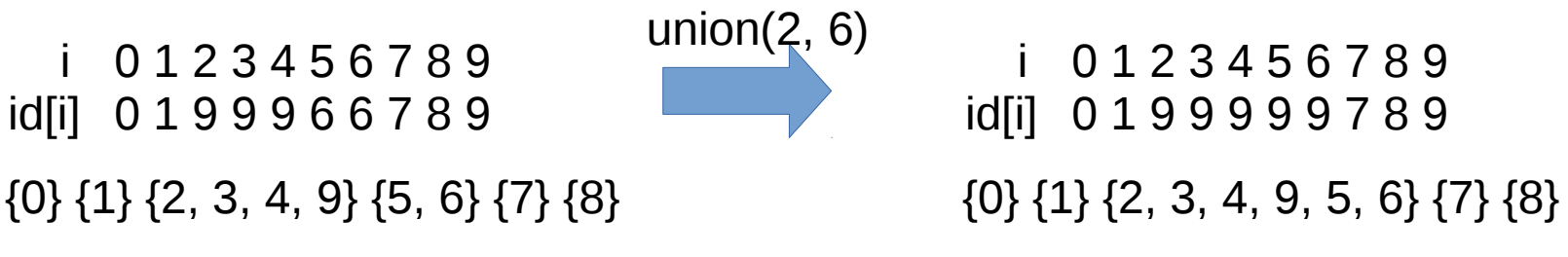


{0} {1} {2, 3, 4, 9} {5, 6} {7} {8}

Implementazione

- find(x): return id[x]
- find(a,b): return id[a] == id[b]
- union(a, b): modifichiamo gli id di uno dei sottoinsiemi

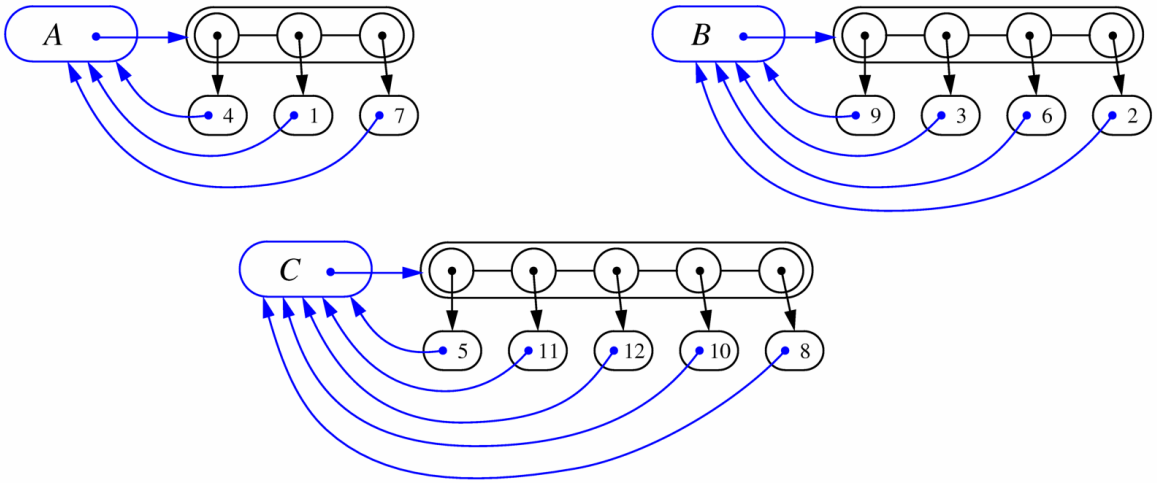
```
Algorithm union(p, q) {
    int pid = id[p];
    for (int i = 0; i < id.length; i++)
        if (id[i] == pid) id[i] = id[q];
}
```



Implementazione collegata

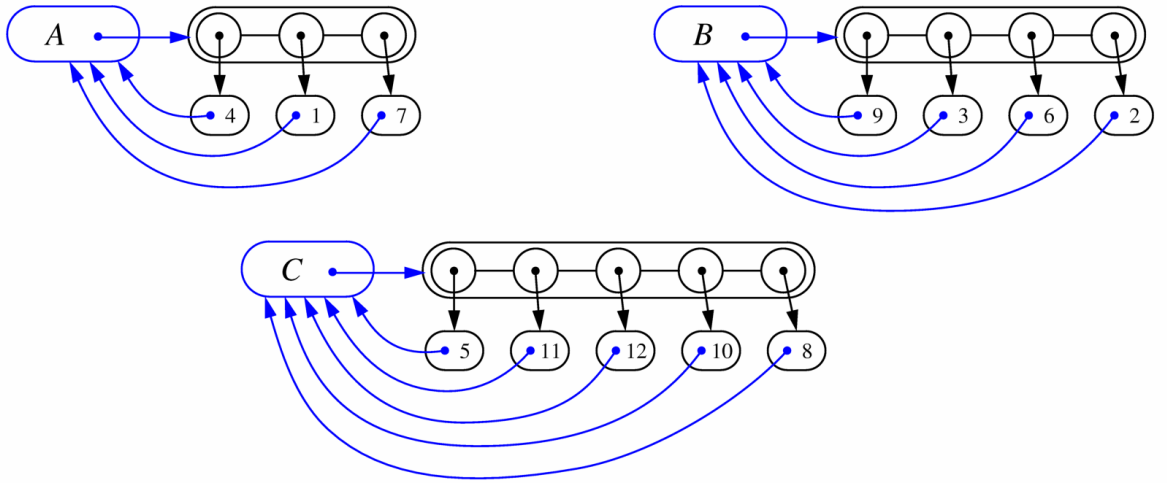
Implementazione con liste

- Ogni sottoinsieme è memorizzato come una lista
- Ogni elemento contiene (anche) un riferimento alla lista (ad esempio la testa)
  - Tale riferimento comune “rappresenta” il sottoinsieme
- Nel seguito
  - a.size: dimensione della lista cui appartiene a
  - a.leader: riferimento alla testa della lista contenente a



Implementazione delle operazioni

- MakeCluster → Costo O(1)
- find(a)
  - Restituisce riferimento alla lista contenente a
  - Costo → O(1)
- union(a, b)
  - Modifica i riferimenti associati agli elementi della lista *più piccola*
  - Costo → min{a.size, b.size}



Efficienza

- *Una serie di k operazioni makeCluster, find e union ha costo complessivo O(k + nlogn)*
- Analisi
  - Paghiamo O(1) per
    - makeCluster(x)
    - find(x)
    - Confrontare le dimensioni dei sottoinsiemi contenenti a e b in union(a, b) (es.: associamo ad ogni elemento la dimensione del sottoinsieme cui appartiene)

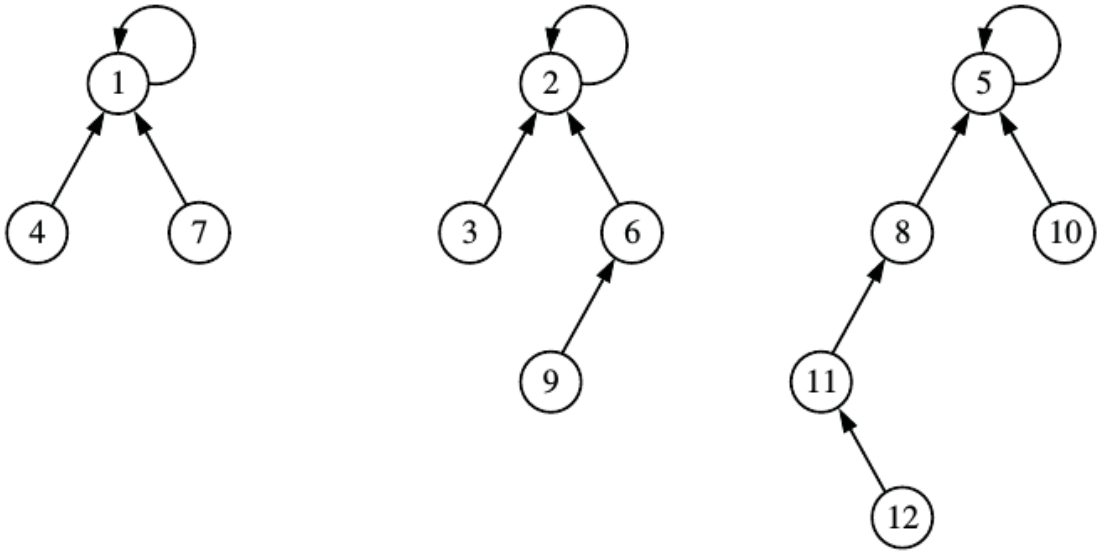
Efficienza

- Analisi/cont.
  - Per ogni elemento presente a: ci chiediamo quante volte a.leader viene aggiornato
  - Costo aggiornamenti $=\Sigma_i$(# aggiornamenti a.leader)
  - Osservazione fondamentale: ogni volta che aggiorniamo a.leader raddoppiamo (almeno) la dimensione del sottoinsieme cui a appartiene

Implementazione basata su alberi

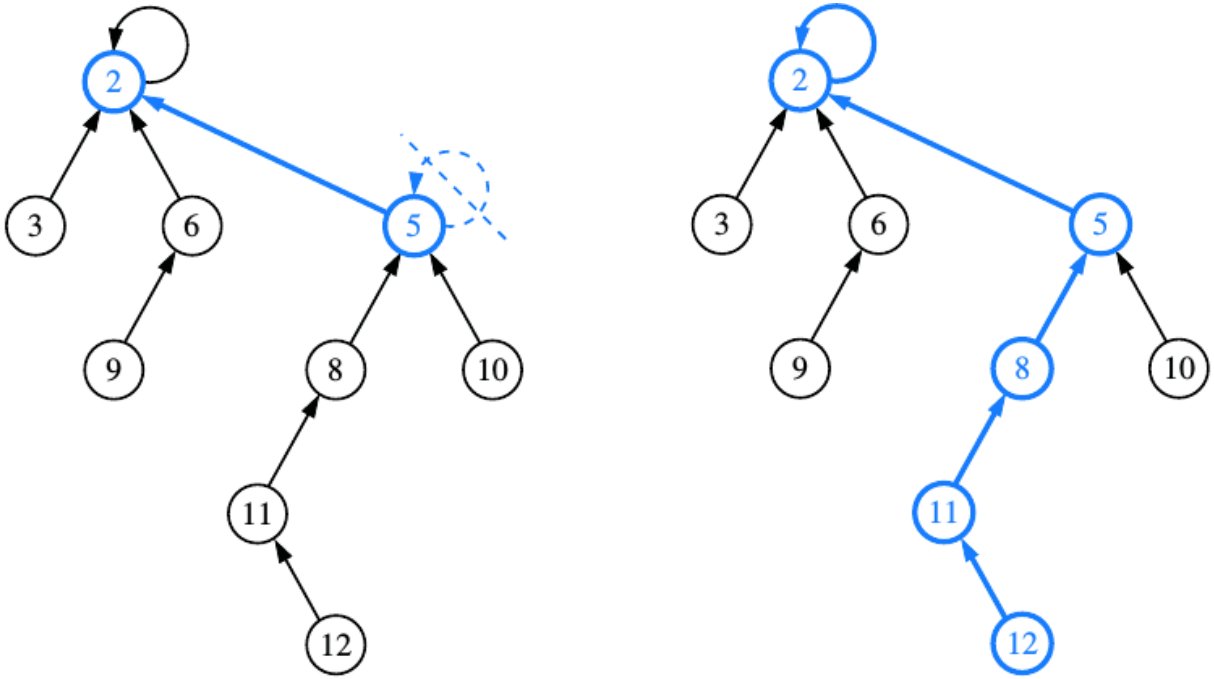
Rappresentazione con alberi

- A ogni elemento a di un insieme corrisponde un nodo di un albero (Locator nelle slide che seguono)
- Campi:
  - Elemento → a.element
  - Dimensione albero cui appartiene a → a.size
  - Riferimento al genitore di a nell'albero (o a se stesso se radice) → a.parent
- Nodi dello stesso albero corrispondono allo stesso sottoinsieme
- Il leader di un sottoinsieme è la radice del sottoalbero corrispondente



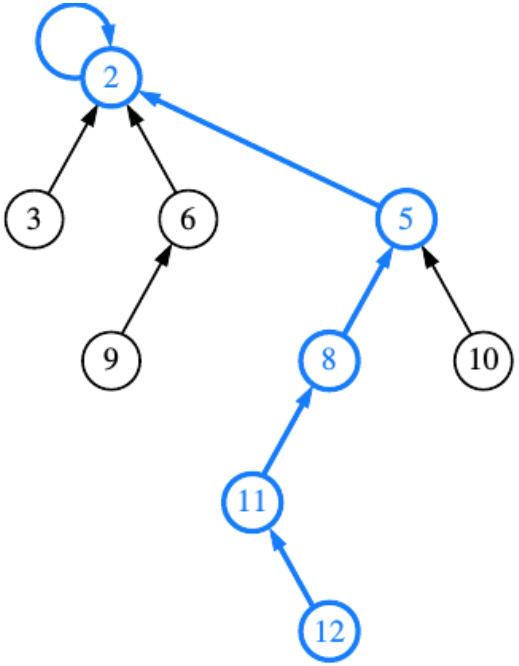
Implementazione dell’unione

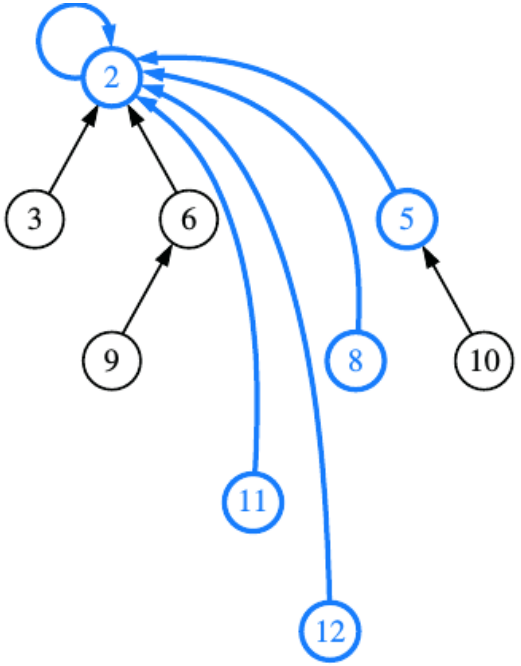
- union(2, 12)
  - Si individuano le radici dei sottoalberi contenenti 2 e 12
    - find(2)
    - Find(12)
    - Costo find(a) nel caso peggiore: proporzionale al cammino da a alla radice/leader dell’albero corrispondente
  - Si rende uno dei due alberi sottoalbero dell’altro con costo O(1)



Ottimizzazioni

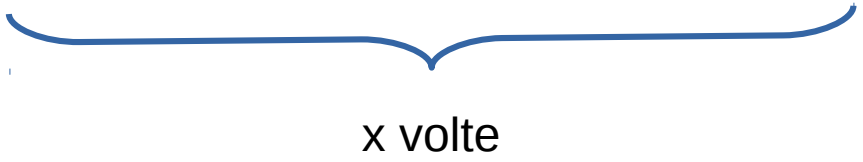
- Senza modifiche questa soluzione non è più efficiente della precedente
- Ottimizzazioni
  - Union-by-size → quando si effettua union(a, b): se a.size < b.size allora si aggiorna a.leader e viceversa
  - Compressione
    - In un'operazione find, aggiorniamo il genitore di ogni nodo q visitato alla radice





Risultato sorprendente

- *Usando compressione e union-by-size una serie di k operazioni ha costo O(klog*n)*
  - *log* è la funzione log-star: il minimo x tale che log(log(log ...(log n) ...)) diventa < 2*



Implementazione - 1

```
1  /** A Union-Find structure for maintaining disjoint sets. */
2  public class Partition<E> {
3    //--------------- nested Locator class -------------
4    private class Locator<E> implements Position<E> {
5      public E element;
6      public int size;
7      public Locator<E> parent;
8      public Locator(E elem) {
9        element = elem;
10       size = 1;
11       parent = this;              // convention for a cluster leader
12     }
13     public E getElement( ) { return element; }
14   } //--------- end of nested Locator class ---------
15   /** Makes a new cluster containing element e and returns its position. */
16   public Position<E> makeCluster(E e) {
17     return new Locator<E>(e);
18   }
```

Implementazione - 2

```
19    /**
20     * Finds the cluster containing the element identified by Position p
21     * and returns the Position of the cluster's leader.
22     */
23    public Position<E> find(Position<E> p) {
24      Locator<E> loc = validate(p);
25      if (loc.parent != loc)
26        loc.parent = (Locator<E>) find(loc.parent); // overwrite parent after recursion
27      return loc.parent;
28    }
29    /** Merges the clusters containing elements with positions p and q (if distinct). */
30    public void union(Position<E> p, Position<E> q) {
31      Locator<E> a = (Locator<E>) find(p);
32      Locator<E> b = (Locator<E>) find(q);
33      if (a != b)
34        if (a.size > b.size) {
35          b.parent = a;
36          a.size += b.size;
37        } else {
38          a.parent = b;
39          b.size += a.size;
40        }
41    }
42  }
```