▼ **9.0 - Union-find**

▼ **9.1 - Introduzione all’union-find**

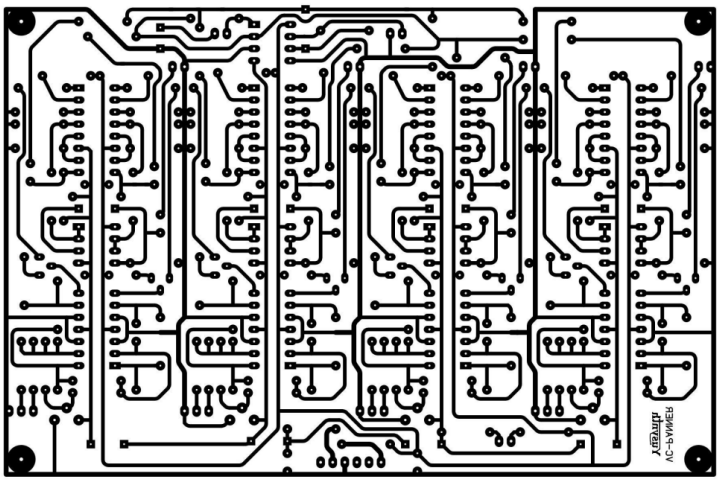
Definizione

Per arrivare al concetto di struttura dati union-find occorre precisare che tutto, anche un singolo elemento, viene considerato un insieme.

> Una struttura dati **union-find** è una collezione $S = \{S_1, S_2, \ldots, S_k\}$ di insiemi dinamici disgiunti in cui ogni insieme è identificato da un rappresentante univoco.

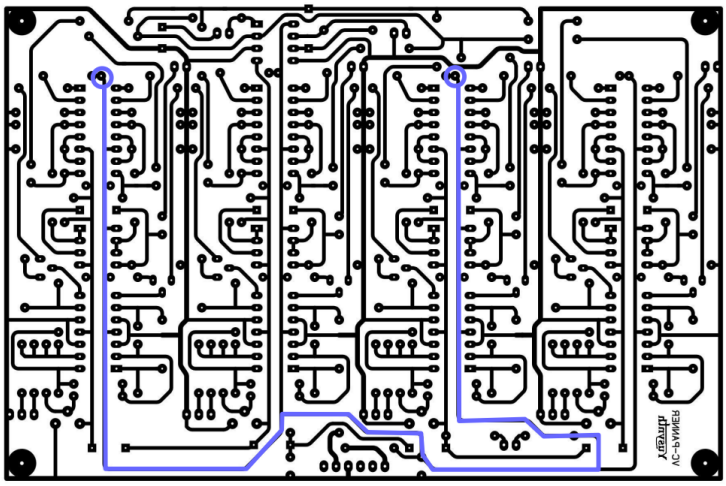
Il **rappresentante** di un insieme può essere un qualsiasi membro dell’insieme, ed essendo univoco qualsiasi operazione di ricerca del rappresentante in un certo insieme deve restituire sempre lo stesso risultato. Tale rappresentante inoltre può cambiare solo nel caso di unione con un altro insieme.

Possibili applicazioni



Circuito elettronico.

Una possibile applicazione della struttura dati union-find può essere trovata nell’ambito dei **circuiti elettronici**, nei quali è possibile creare degli insiemi che contengono i pin collegati da segmenti conduttivi, e in questo modo poter controllare tramite le operazioni elementari per union-find se due di questi pin sono contenuti nello stesso insieme, ovvero se esiste una strada percorribile che li connette.



Operazioni elementari

- **makeSet(x)**: crea un nuovo insieme il cui unico elemento e rappresentate è x (x non deve appartenere a nessun altro insieme).
- **find(x)**: restituisce il rappresentante dell'insieme che contiene x.
- **union(x, y)**: unisce i due insiemi rappresentati da x e da y, eliminando i due insiemi x e y e scegliendo un rappresentante univoco per il nuovo insieme.

▼ **9.2 - QuickFind e QuickUnion**

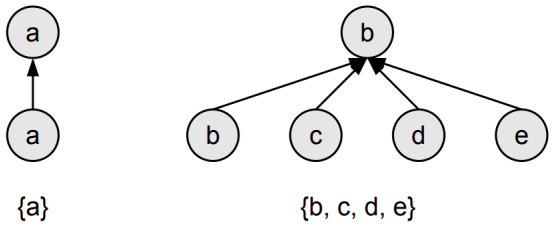
È possibile rappresentare una struttura dati union-find tramite diverse tipologie di implementazione. Noi analizzeremo le due implementazioni **QuickFind** e **QuickUnion**, che, come suggerito dal nome, rendono asintoticamente veloci rispettivamente le operazioni find e union:

- **QuickFind**: alberi di altezza 1
  - makeSet: $O(1)$.
  - find: $O(1)$.
  - union: $O(n)$.
- **QuickUnion**: alberi generali
  - makeSet: $O(1)$.
  - find: $O(n)$.
  - union: $O(1)$.

Il consiglio è dunque quello di utilizzare l'implementazione QuickFind quando le find sono frequenti e le union sono rare, altrimenti utilizzare QuickUnion.

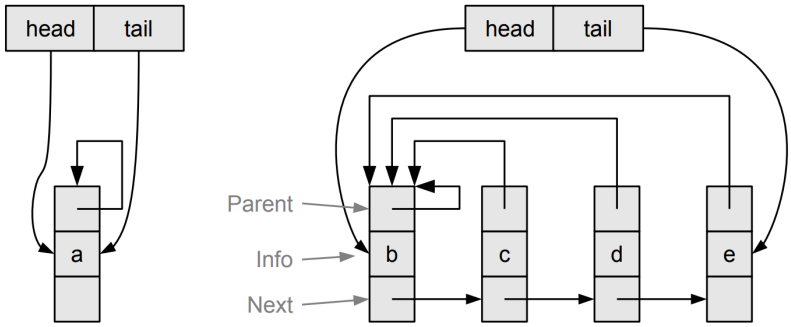
QuickFind

L'implementazione **QuickFind** utilizza **alberi di altezza 1** nelle quali foglie sono contenuti gli elementi dell'insieme, mentre nella radice è contenuto il rappresentante dell'insieme.



Nota implementativa

È possibile rappresentare gli alberi di altezza 1 tramite liste in cui ogni elemento ha un puntatore che punta al rappresentante dell'insieme.



Costi computazionali

Le operazioni **makeSet(x)** e **find(x)** hanno un costo computazionale $O(1)$ in quanto makeSet crea un albero in cui l'unica foglia è x e la radice è il rappresentante di x, ovvero x stesso, e find restituisce il puntatore al padre di x.

L'operazione **union(x, y)** richiede invece che tutte le foglie dell'albero y vengano spostate nell'albero x, quindi, avendo y nel caso peggiore $n - 1$ foglie, con n il numero degli elementi nel nuovo insieme, il costo computazionale è $O(n)$. È comunque possibile abbassare il costo computazionale della funzione union utilizzando un'euristica sul peso.

Euristica sul peso

Una **strategia** per diminuire il costo computazionale dell'operazione union è quella di memorizzare negli alberi QuickFind la dimensione di tali alberi (tale operazione di mantenimento della dimensione può avvenire con un costo $O(1)$) e ogni volta che avviene un union appendere l'insieme con meno elementi all'insieme con più elementi.

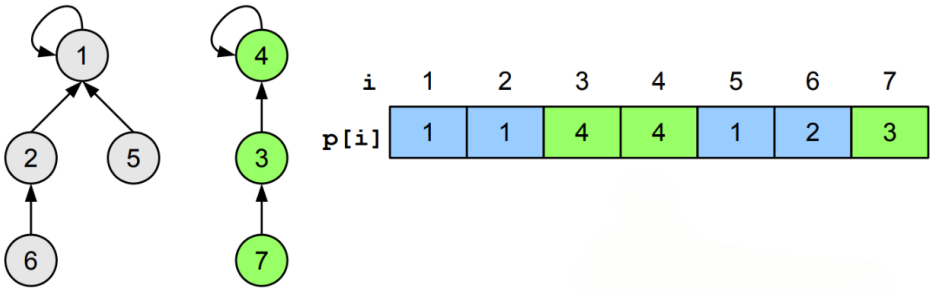
In questo modo **osserviamo** che ogni foglia che acquista un nuovo padre farà parte di un insieme almeno il doppio più grande dell'insieme di partenza, dunque una foglia facente parte di un insieme di n elementi ha cambiato padre al più $\log n$ volte. In base a questa osservazione, calcoliamo il **costo ammortizzato** dell'operazione union, ricordando che è dato dal costo complessivo di k esecuzioni diviso k. Considerando un insieme di dimensione n creato con $n - 1$ unioni, ovvero il numero massimo di unioni possibili, e basandoci sull'osservazione di prima notiamo che per costruirlo tramite operazioni di union tutte le n foglie hanno avuto al massimo $\log n$ cambi di padre. Il costo ammortizzato risulta quindi essere $\frac{O(n \cdot \log n)}{n-1} = O(\log n)$.

QuickUnion

L'implementazione **QuickUnion** utilizza alberi generici che nella radice contengono il rappresentante e i quali figli sono alberi che corrispondono ai sottoinsiemi contenuti nell'insieme generale.

Nota implementativa

Un modo per rappresentare alberi generici tramite un array è quello di utilizzare un cosiddetto **vettore dei padri**, il quale per ogni indice dell'array viene rappresentato un elemento dell'albero, e il contenuto di tale nodo indica l'indice del nodo padre.



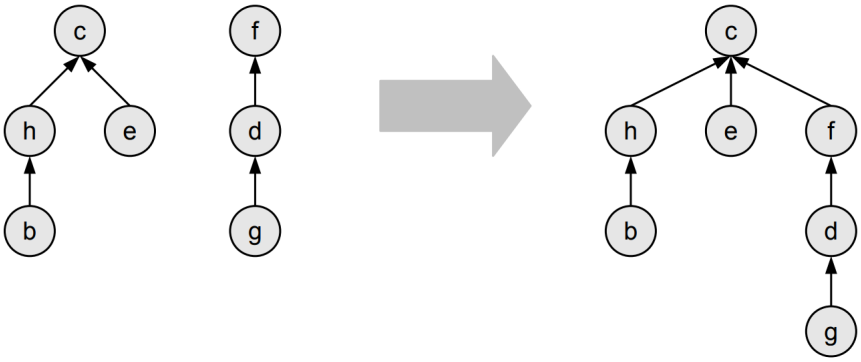
Vettore dei padri di due alberi generici.

Costi computazionali

La funzione **makeSet(x)** crea un albero che ha un unico nodo x, dunque il costo è $O(1)$.

La funzione **find(x)** invece deve ripercorrere l'albero partendo da x e arrivando alla radice, dunque il costo è $O(n)$.

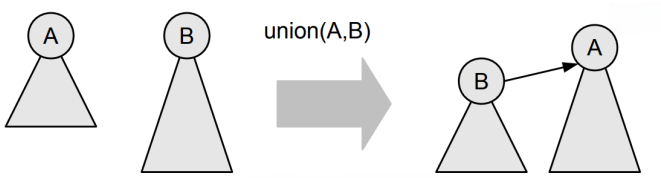
La funzione **union(x, y)** deve inserire l'albero rappresentato da y come figlio della radice x, quindi ha costo $O(1)$.



Esecuzione di union(c, f).

Euristica sul rango

Visto che il costo computazionale dell'operazione find è dato dall'altezza dell'albero che rappresenta l'insieme, allora è possibile pensare ad una **strategia** che abbassi tale costo evitando di creare alberi troppo alti. È possibile fare ciò memorizzando il **rango** del nodo radice, ovvero il numero di archi del cammino più lungo tra il nodo in input e una foglia sua discendente (tale operazione di memorizzazione del rango può avvenire con un costo $O(1)$) e quando avviene un'operazione union appendiamo l'albero con la radice con rango minore a quello con la radice con rango maggiore.



Denotando il rango di un nodo x tramite la funzione $rank(x)$ e sia n il numero di nodi contenuti in un albero costruito tramite euristica sul rango, è possibile dimostrare che $n \geq 2^{rank(x)}$, dove x è il nodo radice dell'albero.

Siccome il costo della funzione find dipende dall'altezza dell'albero, e tale altezza h è uguale a $rank(x)$, allora per l'affermazione precedente abbiamo che $h \leq \log n$, e quindi la funzione find ha un costo computazionale $\leq \log n$, quindi $O(\log n)$.

Riassunto costi computazionali

	QuickFind	QuickUnion	QuickFind eur. peso	QuickUnion eur. rank
makeSet	O(1)	O(1)	O(1)	O(1)
union	O(n)	O(1)	O(log n)	O(1)
find	O(1)	O(n)	O(1)	O(log n)

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di una struttura dati union-find.