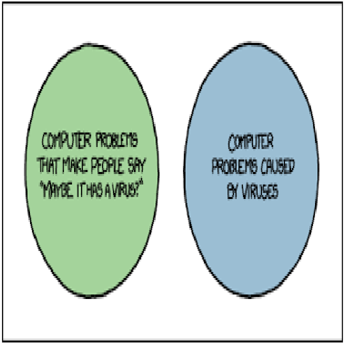
Algoritmi e strutture dati

Strutture per insiemi disgiunti

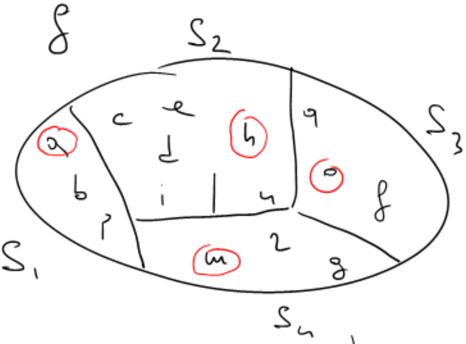


Menú di questa lezione

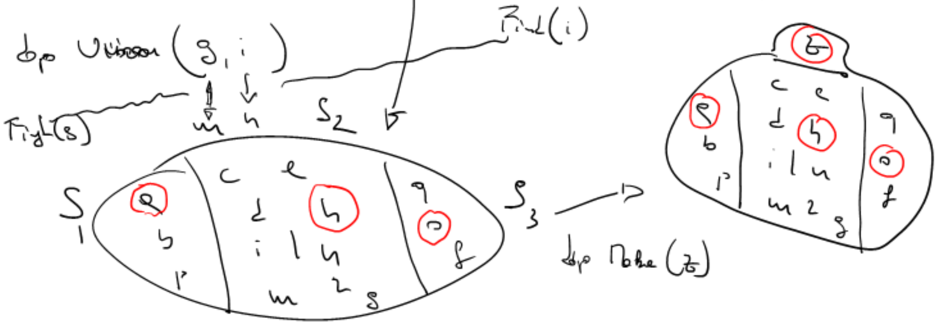
In questa lezione vedremo diverse implementazioni di strutture dati per insiemi disgiunti, e ne studieremo la complessità delle operazioni ad esse associate.

Insiemi disgiunti

Gli insiemi disgiunti sono una struttura dati astratta, parzialmente dinamica, parzialmente sparsa, e non basata sull'ordinamento, fondamentalmente diversa da quelle viste fino ad ora. La caratteristica principale di un insieme di insiemi disgiunti é che le operazioni ad esso associate sono, tipicamente: *MakeSet*, che costruisce un nuovo insieme disgiunto; *Union*, unisce due insiemi disgiunti in uno solo; e *FindSet*: trova il rappresentante dell'insieme al quale appartiene l'elemento. Ogni insieme è quindi dotato di un elemento rappresentativo (che può essere uno qualsiasi). Gli insiemi crescono solo in due modi: quando vengono creati (e contengono esattamente un elemento), e quando vengono uniti due insiemi in uno solo che contiene gli elementi di entrambi.



Fornite quanto situazione arbitraria eventualmente cosa accade dopo un' operazione



Insiemi disgiunti

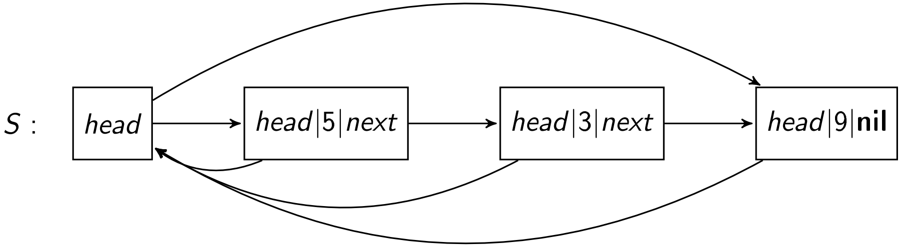
Immaginiamo, ad esempio, di avere i seguenti insiemi: $S_1 = \{5, 12, 20\}$, $S_2 = \{7\}$, $S_3 = \{13, 2\}$. Ognuno di essi puó essere rappresentato da uno qualsiasi dei suoi elementi, e ció che dobbiamo mantenere è l'informazione sull'insieme stesso, il quale non ha una vera struttura interna. Quindi, indipendentemente dall'implementazione scelta (ne vediamo tre) possiamo immaginare che $\mathcal{S}$ abbia almeno un array, che contiene tutte queste chiavi, e, per ognuna di esse, un'informazione aggiuntiva che ci permetta di ricostruire tutta la sua struttura. Questa informazione aggiuntiva cambia a seconda dell'implementazione scelta.

Insiemi disgiunti

La particolare scelta delle operazioni che si vogliono rendere disponibili influenza (come sempre, d'altronde) la struttura dati, che tendenzialmente è ottimizzata per quelle operazioni e, se repentinamente si volesse offrire un'altra operazione diversa da quelle originali, questo risulterebbe impossibile o troppo costoso. Gli insiemi disgiunti hanno una applicazione fondamentale in uno degli algoritmi su grafi che vedremo piú avanti (per il calcolo dell'albero di copertura minimo), ma quando una struttura è complessa, è piú difficile convincersi che emerga anche in contesti del mondo reale. Consideriamo allora questo scenario: abbiamo una rete molto grande di criminali, e tutti usano molti **alias** diversi tra loro. I nostri informatori riescono, di tanto in tanto, a scoprire che due **alias** sono la stessa persona. Qual è la struttura dati ottima per mantenere questa informazione?

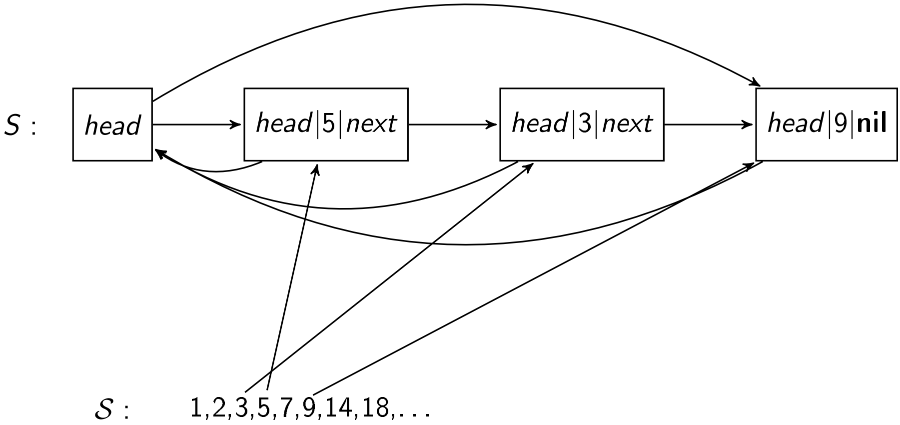
Insiemi disgiunti: liste

L'implementazione piú intuitiva per gestire $\mathcal{S}$ passa attraverso l'uso delle liste collegate. L'elemento $S \in \mathcal{S}$ è quindi una lista dotata degli attributi $S.head$ (che punta al primo elemento) e $S.tail$ (che punta all'ultimo elemento). Ogni elemento x è dotato di $x.next$ (come sempre) e $x.head$ che punta all'insieme S che lo contiene.



Insiemi disgiunti: liste

In questa versione, l'informazione aggiuntiva che contiene ogni $\mathcal{S}[i]$ è un puntatore all'elemento i in memoria, cioè alla casella x che contiene la chiave i. Lo chiamiamo per esempio $\mathcal{S}[i].element$.



Insiemi disgiunti: liste

In questa maniera, implementare *MakeSet*(x) è banale: crea un nuovo oggetto S, tale che $S.head = S.tail = x$. Se poi decidiamo che il rappresentate di ogni S è precisamente l'elemento puntato da $S.head$, allora implementare *FindSet*(x) è altrettanto banale: dato x cerchiamo prima $x.head$ e poi $x.head.head$ per arrivare al suo rappresentante. Entrambe queste operazioni costano $O(1)$. Pertanto tutta la complessitá risiede nell'operazione di unione. Tra le sue caratteristiche, osserviamo: é commutativa (non vi é ordine alcuno nell'unione), viene eseguita a partire da elementi dei due insiemi uniti (esempio: unisci l'insieme che contiene 3 con quello che contiene 7, assumendo che siano diversi), e distrugge il secondo insieme (tutti i suoi elementi appartengono al primo dopo l'operazione).

Insiemi disgiunti: liste

In ogni operazione, gli elementi sono ipotizzati giá **creati e nella memoria principale**. Creare un nuovo insieme (*MakeSet*) signfica: creare un oggetto S (che noi ipotizziamo giá esistente), creare un oggetto x con la chiave che vogliamo (che noi ipotizziamo giá creato), e collegarli. La differenza con una lista 'normale' è che l'oggetto 'lista' (denotato da S, qui) è in memoria principale e non nello stack, ma è un dettaglio implementativo di minima importanza. L'operazione di *FindSet*, dato un oggetto x, che contiene una chiave della quale vogliamo conoscere il rappresentante, consiste nel seguire il puntatore *head* di x per arrivare a S, e poi nuovamente il puntatore *head*. Finalmente, l'operazione di *Union* di x e y consiste nel trovare S_1 ed S_2 (rispettivamente, gli insiemi di x e y), e, se sono diversi, aggiornare $S_1.tail.next$ a $S_2.head$ e, per ogni z tale che $z.head = S_2$, impostare $z.head = S_1$.

Insiemi disgiunti: liste

Vediamo il codice delle operazioni. Nonostante sia stato detto che un elemento nuovo è ipotizzato giá creato in memoria, dobbiamo ugualmente passare la sua chiave all'operazione di *MakeSet*, per poter correttamente indicizzare il suo insieme in $\mathcal{S}$. In questo possiamo garantire di non perdere la traccia dell'elemento durante la creazione dell'insieme.

proc ***MakeSet*** $(\mathcal{S}, S, x, i)$

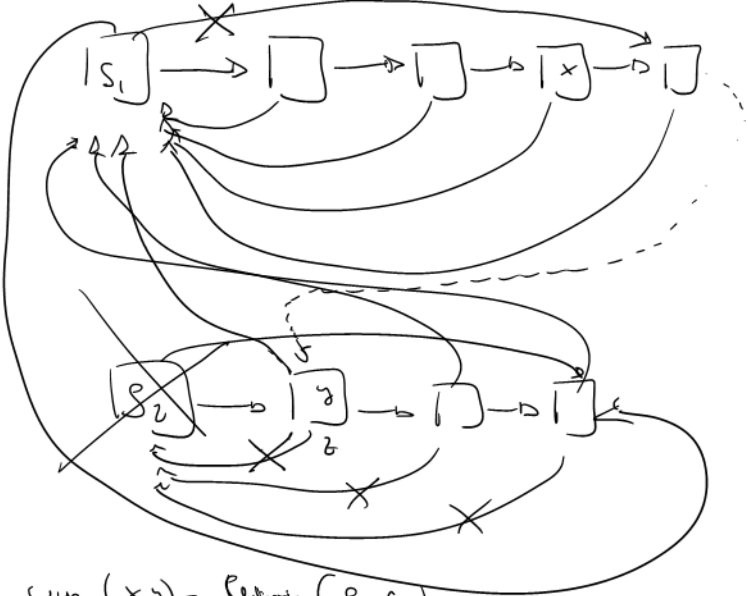
$$\begin{cases} \mathcal{S}[i].set = x \\ S.head = x \\ S.tail = x \end{cases}$$

proc ***Union*** (x, y)

$$\begin{cases} S_1 = x.head \\ S_2 = y.head \\ \textbf{if } (S_1 \neq S_2) \\ \quad \textbf{then} \\ \begin{cases} S_1.tail.next = S_2.head \\ z = S_2.head \\ \textbf{while } (z \neq \textbf{nil}\ ) \\ \begin{cases} z.head = S_1 \\ z = z.next \end{cases} \\ S_1.tail = S_2.tail \end{cases} \end{cases}$$

proc ***FindSet*** (x)

$$\{\textbf{return } x.head.head$$



$\text{Union}(x, y) = \text{Link}(S_1, S_2)$

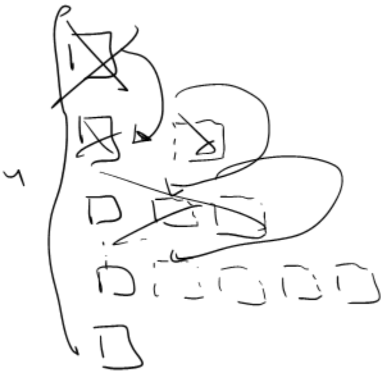
Insiemi disgiunti: liste

Queste operazioni sono chiaramente **corrette** e **terminanti**. Per calcolare la **complessitá**, però, dobbiamo fare ricorso ad un tipo leggermente diverso di analisi, chiamata **analisi ammortizzata**. Si tratta di calcolare il costo **medio** di una operazione qualsiasi nel contesto di un gruppo di operazioni, piuttosto che il costo per operazione. Per poterlo calcolare, diciamo che m denota il numero di operazioni qualsiasi fatte su $\mathcal{S}$, ed $n \leq m$ denota il numero di *MakeSet* tra le m operazioni. Consideriamo il caso peggiore. Non è difficile definire (per avere il caso peggiore) una situazione in cui abbiamo gli oggetti $x_1, \ldots, x_n$ ed ognuno costituisce il suo proprio insieme. Quindi abbiamo n operazioni *MakeSet* seguite da $n-1$ *Union* in maniera che $m = 2 \cdot n - 1$. Spendiamo $\Theta(n)$ per generare gli insiemi. Poi, nel caso peggiore, spendiamo 1 per la prima unione, 2 per la seconda, e cosí via, fino all'ultima unione che costa n. Il totale é $\Theta(n^2)$. Le operazioni di *FindSet* non contribuiscono a cambiare la struttura di $\mathcal{S}$ e hanno un costo costante; pertanto le abbiamo escluse dall'analisi.

CASO PEGGIORE

① Tocco tutti gli n liste

② Faccio tutte le unioni peggiori; mettendo sempre le liste più grandi in quelle più piccole



Conto di tutte le unioni:

$$1 + 2 + 3 \ldots + n-1$$

$$\Theta(n^2)$$

ho al massimo solo $n-1$ unioni

Quindi, il costo complessivo di m operazioni di cui n sono Multi è $\Theta(n^2)$

Pertanto, il costo medio di ogni operazione è

$$\frac{\Theta(n^2)}{2n-1} = \boxed{\Theta(n)}$$

Insiemi disgiunti: liste

Qual è la differenza tra l'analisi ammortizzata e quella del caso medio? Nell'analisi ammortizzata non ci sono considerazioni probabilistiche: si calcola il costo medio di ogni operazione nei casi ottimo, medio e pessimo (nel nostro caso lo abbiamo fatto solo nel caso pessimo), in maniera, però, da tenere conto dell'influenza mutua tra operazioni. E perchè non abbiamo avuto occasione di utilizzarla prima? Perchè altre strutture dati, come le liste, per le quali abbiamo visto diverse operazioni, non sono tali che eseguire una operazione influenza in maniera evidente il costo di eseguire altre operazioni. Questo è il primo (e unico, per noi) caso in cui succede ció, e lo trattiamo in maniera particolare.

Una prima strategia che possiamo usare per migliorare la situazione è chiamata **unione pesata**. Il principio sul quale si basa è semplice: se manteniamo in ogni insieme S anche il numero degli elementi dell'insieme, allora possiamo implementare *Union* in maniera che gli aggiornamenti dei puntatori si facciano sempre sull'insieme piú piccolo.

proc ***MakeSet*** $(\mathcal{S}, S, x, i)$

$$\begin{cases} \mathcal{S}[i].set = x \\ S.head = x \\ S.tail = x \\ S.rank = 0 \end{cases}$$

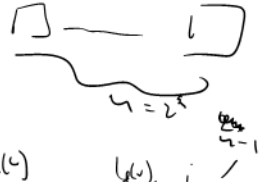
proc ***Union*** (x, y)

$$\begin{cases} S_1 = x.head \\ S_2 = y.head \\ \textbf{if } (S_1 \neq S_2) \\ \quad \textbf{then} \\ \quad \begin{cases} \textbf{if } (S_2.rank > S_1.rank) \\ \quad \textbf{then } \; SwapVariable(S_1, S_2) \\ S_1.tail.next = S_2.head \\ z = S_2.head \\ \textbf{while } (z \neq \textbf{nil}) \\ \begin{cases} z.head = S_1 \\ z = z.next \end{cases} \\ S_1.tail = S_2.tail \\ S_1.rank = S_1.rank + S_2.rank \end{cases} \end{cases}$$

Caso peggiore (con unione pesata)

$\boxed{\text{Costo totale delle unioni}} = \underbrace{\frac{n}{2}\cdot 1 + \frac{n}{4}\cdot 2 + \frac{n}{8}\cdot 4 \ \dots}_{\log(n)}$

$= \frac{n}{2}\log(n) = \Theta(n\log(n))$



$\boxed{\text{Tot di Union}}$ $\sum_{i=1}^{\log(n)} \frac{n}{2^i} = n\sum_{i=1}^{\log(n)}\left(\frac{1}{2}\right)^i = \Theta(n)$

Costo totale : $O(n) + O(n \log(n)) = O(n \log(n))$

Make (fase) Union (fusione)

Costo ammortizzato : $\dfrac{O(n \log(n))}{2n-1} = O(\log(n))$

Insiemi disgiunti: liste con unione pesata

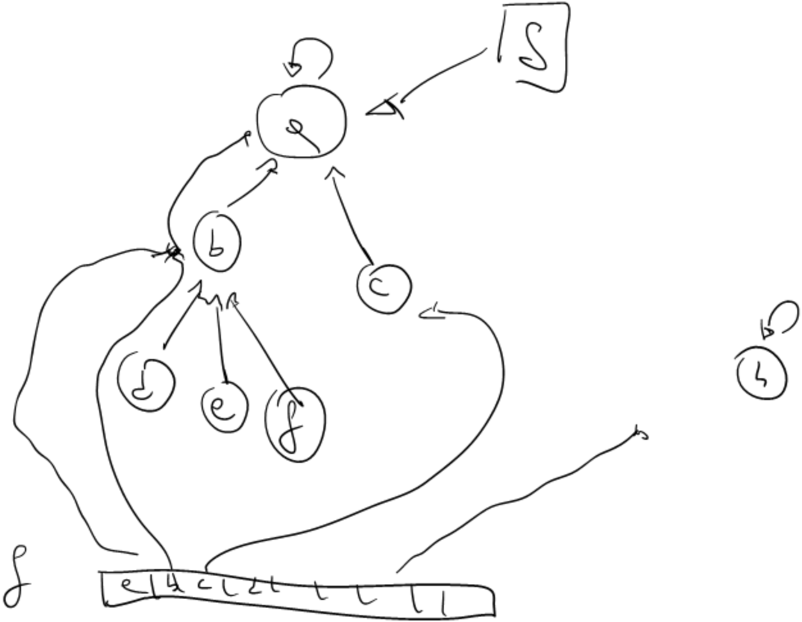
Di nuovo, concentriamoci sulla **complessità**. Adesso il caso peggiore accade quando tutti gli S sono di dimensione uguale, ma, come vedremo, l'analisi ammortizzata ci dirá che esiste un risparmio in media di tempo. Infatti, mettiamoci nelle stesse condizioni di prima: m operazioni generiche di cui n *MakeSet*. Come nel caso precedente, ci dovremo fermare quando avremo raggiunto un solo insieme con tutti gli elementi. Le operazioni di *FindSet*, come prima, vengono inizialmente escluse dal ragionamento. Dati n insiemi tutti di un solo elemento, la situazione peggiore si verifica effettuando $\frac{n}{2}$ unioni (uniamo gli insiemi a due a due): infatti, se non facessimo cosí, arriveremmo ad avere un insieme con n elementi dopo solamente $n-1$ passi, e non avremmo la situazione peggiore! Se, per comodità, ipotizziamo $n = 2^k$ per qualche k, allora possiamo proseguire ragionando nello stesso modo: $\frac{n}{2}$ unioni la prima volta (ottenendo $\frac{n}{2}$ insiemi di 2 elementi), seguite da $\frac{n}{4}$ unioni (ottenendo $\frac{n}{4}$ insiemi di 4 elementi), e cosí via, precisamente $log(n)$ volte.

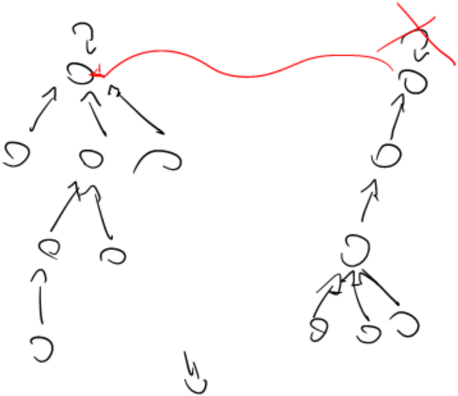
Insiemi disgiunti: liste con unione pesata

Quanto è il costo totale delle m operazioni? Ogni unione, con questa strategia, costa: 1 la prima volta, 2 la seconda, 4 la terza, e cosí via. Quindi, il costo totale di tutte le unioni che possiamo fare prima di arrivare all'insieme con tutti gli elementi é $\Theta(n \cdot log(n))$. Allora, come nel caso precedente, succede che mettersi nel caso peggiore significa forzare che le m operazioni siano, precisamente, n *MakeSet* seguite da tutte le *Union* possibili, e questo ci dá un costo totale di $\Theta(n \cdot log(n))$. In questo caso, come nel precedente, aggiungere qualche *FindSet* nel gruppo di m operazioni puó solo migliorare la complessità, ed è per questo che le escludiamo dall'analisi del caso peggiore. Nuovamente, ci chiediamo se possiamo migliorare queste prestazioni. La strategia implementativa che ci permette una ulteriore miglioria consiste di tre passi: cambiare la rappresentazione, adattare la unione pesata alla nuova rappresentazione, e modificare l'operazione *FindSet*, per renderla **attiva**.

Insiemi disgiunti: foreste di alberi

Il modo più efficiente per trattare gli insiemi disgiunti sono le **foreste di alberi**. Accreditati per l'introduzione di questa struttura dati sono Bernard Galler e Michael Fisher, nel 1964.


S
a
b
c
d
e
f
h
f



Union

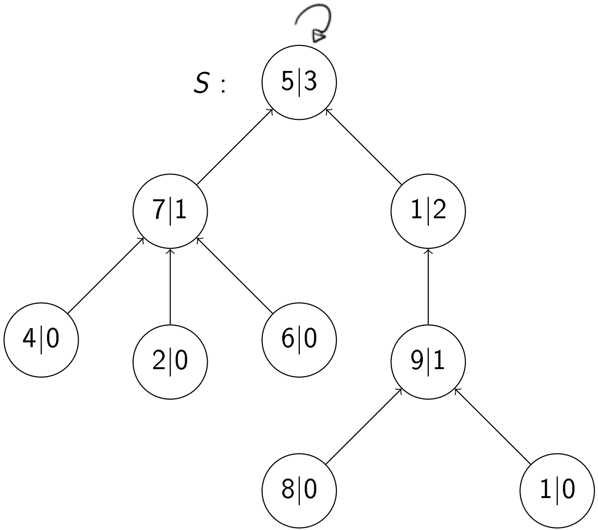
Insiemi disgiunti: foreste di alberi

Non vedremo, per questa rappresentazione, il calcolo della complessitá in dettaglio, ma daremo solo il risultato finale. La rappresentazione è basata in **alberi** piuttosto che liste. Un nodo x (un elemento) contiene le seguenti informazioni: $x.p$ (il padre), e $x.rank$ (un limite superiore all'altezza del sotto-albero radicato in x). Gli alberi (gli insiemi disgiunti) sono k-ari, e formano una **foresta** $\mathcal{S}$. L'operazione di *MakeSet* è la stessa: si crea un nuovo albero di altezza massima 0, tale che il padre dell'unico nodo x è x stesso. Attenzione: questi alberi sono liste particolari, e non vanno confusi con gli alberi e i grafi che vedremo piú avanti.

Insiemi disgiunti: foreste di alberi

Un nodo di un albero k-ario **non ha, in generale, nessun puntatore ai figli**. Infatti, non abbiamo un limite superiore a quanti figli possiamo avere, e, cosa più importante, non ci interessa agli scopi di questa struttura dati. Il rango **non** è la misura dell'altezza, ma, come detto sopra, un suo limite superiore. Questo significa che un albero in questa struttura può avere altezza h e rango qualsiasi (superiore o uguale a h). Vedremo perchè questa nozione rilassata è utile.

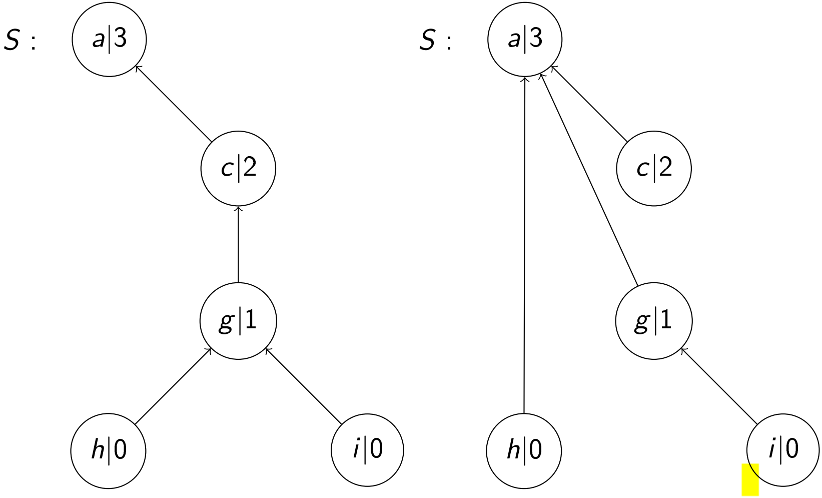
Insiemi disgiunti: foreste di alberi



L'operazione di unione, in due fasi, consiste, come prima, nel trovare i rappresentanti degli elementi utilizzati come indici; se le due radici sono x e y, rispettivamente, si sceglie quello il cui rango sia inferiore, e si aggiorna **solo il padre**, rendendolo uguale all'altro elemento. Con il criterio di **unione per rango** (il corrispondente dell'unione pesata nella versione con le liste), il rango dell'insieme risultante cambia (e si incrementa) **solo se i ranghi dei due componenti erano uguali**, e rimane inviariato (uguale al massimo tra i due) negli altri casi. L'operazione *FindSet* diventa attiva. Non solo restituisce, come sempre, il rappresentante, ma, mentre scorre i puntatori verso l'alto alla sua ricerca, li aggiorna **appiattendo** il ramo al quale appartiene. Chiamiamo **compressione del percorso** questa strategia.

Insiemi disgiunti: foreste di alberi

Un esempio di *FindSet*; a destra, l'effetto di chiamare *FindSet*(h):



Insiemi disgiunti: foreste di alberi

proc ***MakeSet*** (x)

$$\begin{cases} x.p = x \\ x.rank = 0 \end{cases}$$

proc ***Union*** (x, y)

$$\begin{cases} x = \mathit{Findset}(x) \\ y = \mathit{Findset}(y) \\ \textbf{if } (x.rank > y.rank) \\ \quad \textbf{then } y.p = x \\ \textbf{if } (x.rank \leq y.rank) \\ \quad \textbf{then} \\ \begin{cases} x.p = y \\ \textbf{if } (x.rank = y.rank) \\ \quad \textbf{then } y.rank = y.rank + 1 \end{cases} \end{cases}$$

proc ***FindSet*** (x)

$$\begin{cases} \textbf{if } (x \neq x.p) \\ \quad \textbf{then } x.p = \mathit{FindSet}(x.p) \\ \textbf{return } x.p \end{cases}$$

Insiemi disgiunti: foreste di alberi

Nuovamente, **correttezza** e **terminazione** di queste procedure non sono in discussione. Il calcolo della **complessitá** di m operazioni in questa implementazione, invece, è molto difficile. Ma la ragione per la quale l'unione per rango sommata alla compressione del percorso migliora le prestazioni globali sono chiare. In sostanza, l'unione effettua sempre al massimo un aggiornamento di puntatori. L'operazione *FindSet* aggiorna un certo numero di puntatori, ma questi, una volta aggiornati, non vengono toccati mai piú, e il prossimo *FindSet* su elementi del percorso che è giá stato compresso costerá un tempo costante. Sia α una certa funzione da $\mathbb{N}$ a $\mathbb{N}$ che cresce approssimativamente come l'inverso della funzione di Ackermann (cioé, cresce in maniera **estremamente** lenta). Nel caso concreto in questione abbiamo che $\alpha(n)$ è minore o uguale a 4 per $n \leq 10^{80}$. Una corretta analisi di m operazioni darebbe che il costo totale é $O(m \cdot \alpha(n))$, che, in ogni situazione pratica, è lo stesso che $O(m)$.

Conclusione

Gli insiemi disgiunti sono un esempio di struttura dati non intuitiva, che esiste solo perchè funzionale ad altri algoritmi. É un esercizio interessante la loro implementazione, perchè a fronte di un codice molto semplice si ottengono effetti anche difficili da modellizzare. Pertanto è un esempio di struttura dati che fornisce idee non banali a chi la studia, che possono essere riutilizzate in altri contesti.