Problema Union-Find

Introduciamo in questa sezione il problema della Union-Find la cui risoluzione rappresenta un passo propedeutico alla analisi di un altro problema: l'albero minimo ricoprente.

Introdurremo il problema riportando prima la definizione formale ed infine presentando i diversi approcci di risoluzione applicabili

1. Il problema Union-Find

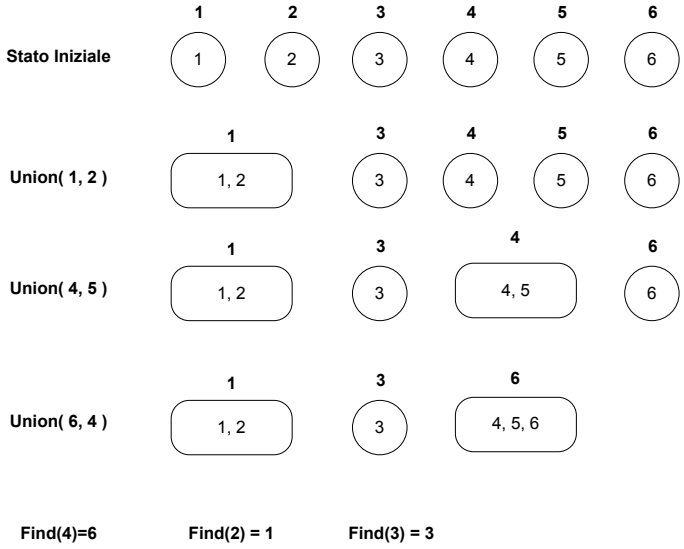
Nel problema Union-Find l'oggetto di nostro interesse è rappresentato da un gruppo di insiemi disgiunti e contenenti elementi qualsiasi che indicheremo con *S*.
Per facilitare l'esposizione, senza comunque ledere il principio di generalità, rappresentiamo gli insiemi di *S* mediante numeri interi, pertanto *S={1, 2, ..., n}*. Infine, sempre in linea con il principio di generalità etichettiamo gli elementi generici contenuti all'interno dei singoli insiemi di *S* mediante numeri interi: *1, 2, 3, ...*
Il problema della Union-Find consiste nella progettazione di algoritmi efficienti che realizzano le seguenti operazioni:

- ***Union(A,B)***, siano *A* e *B* due etichette rappresentanti due insiemi di *S*, l'operazione di *Union* unisce gli elementi dei due insiemi in un unico insieme che sarà denominato A, infine, cancella i vecchi insiemi *A*,*B*;

- ***Find(a)***, sia *a* un elemento appartenente al dominio del problema, l'operazione di *Find* restituisce l'etichetta associata all'insieme contenente l'elemento *a*.

Esempio

Segue una raffigurazione delle modifiche apportate dalla operazione di union e dei valori ritornati dall'operazione di find.



Nell'analizzare il problema della Union-Find assumeremo sempre che inizialmente la composizione sia di *S* che degli insiemi di *S* rispecchi lo stato iniziale raffigurato nell'esempio sopra. Ossia, *S={1, 2, ..., n}* dove *n* è il numero di insiemi appartenenti ad *S* ed ogni insieme di *S* ha un unico elemento etichettato con la stessa etichetta dell'insieme di appartenenza.
Segue la descrizione del problema espressa mediante i formalismi dell'ADT:

Tipo di dato: Union Find
Collezione di insiemi disgiunti identificati mediante una etichetta

Operazioni: Makeset(Elemento *a*)
Crea un insieme formato da un solo elemento, a, ed associa all'insieme l'etichetta ' a'

Union(Insieme *A*, Insieme *B*)
Unisce gli elementi degli insiemi A e B in un unico insieme che etichetta con 'A'. I vecchi insiemi A ed B saranno cancellati

Find(Elemento *a*)
Restituisce l'etichetta dell'insieme contenente l'elemento a

2. Rappresentazioni Quick-Find e Quick-Union

La Quick-find e la Quick-Union sono due strutture dati utilizzate per rappresentare l'ADT Union-Find. In termini di complessità, la rappresentazione Quick-Find favorisce l'implementazione dell'operazione di Find, mentre la rappresentazione Quick Union favorisce l'operazione di Union. Entrambi penalizzano rispettivamente l'operazione di Union e di Find.

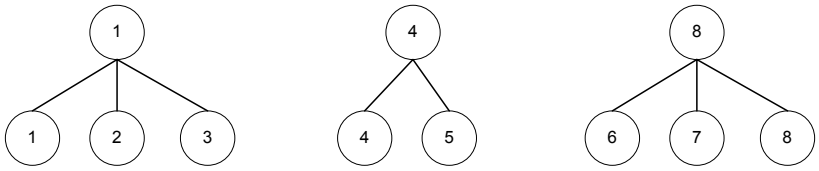
2.1 Algoritmo Quick-Find

Nel rispetto del nome assegnatale la rappresentazione Quick Find favorisce, in termini di complessità, l'implementazione dell'operazione di find, penalizzando però l'operazione di union.

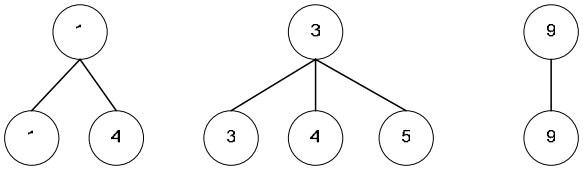
La Quick-Find rappresenta ogni insieme dell'ADT Union Find mediante alberi di altezza uguale ad uno; la radice dell'albero contiene l'etichetta dell'insieme, mentre le foglie dell'albero contengono gli elementi dell'insieme.

Esempio

Sia S formato dai seguenti insiemi: 1={1,2,3}; 4={4,5}; 8= {6,7,8}. La rappresentazione Quick Find di S è:



Mentre se S fosse formato dai seguenti insiemi: 1={1,4}; 3={3,4,5}; 9={9}, la rappresentazione Quick Find di S sarebbe



Valutiamo la complessità delle operazioni di makeset, union e find in relazione al totale del numero di elementi contenuti nei diversi insiemi di *S*.

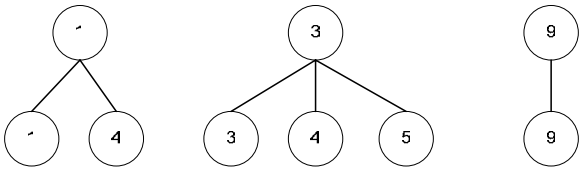
Makeset, l'operazione di makeset richiede la generazione di un albero con due nodi: radice e foglia. La complessità è pertanto costante.

Find**,** l'operazione di find, similmente alla makeset, ha anch'essa complessità costante poiché dato un elemento dell'insieme, *a*, la find dovrà tornare semplicemente l'etichetta associata al padre del nodo contenente *a*.

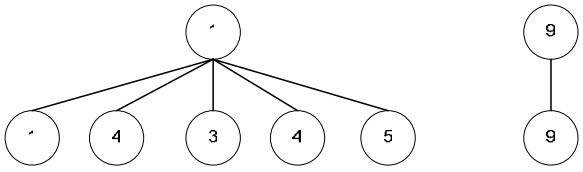
Union**,** come già anticipato l'operazione di union viene penalizzata da tale rappresentazione. L'unione di due insiemi *A, B* richiede la modifica del campo padre di ogni foglia contenuta nell'albero rappresentante *B* e l'eliminazione della radice dell'albero *B*. La complessità pertanto è linearmente proporzionale al numero di elementi dell'insieme *B*, che nel caso pessimo può essere uguale al totale meno uno del numero di elementi contenuti nei diversi insiemi di *S*, ovvero al numero di operazioni di tipo makeset effettuate. Pertanto, siano *n* le operazioni di makeset richiamate la complessità dell'operazione di union è *O(n)*.

Esempio

Sia consideratala seguente rappresentazione Quick-Find.



l'operazione Union(1, 3) modifica la foresta di alberi come raffigurato di seguito:



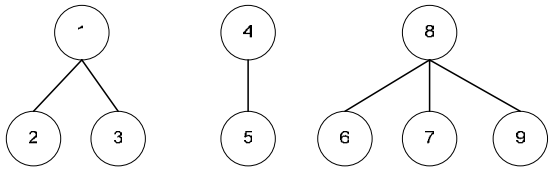
2.2 Rappresentazione Quick-Union

Contrariamente alla Quick-Find, la rappresentazione Quick-Union favorisce, in termini di complessità, l'implementazione dell'operazione di union, penalizzando però l'operazione di find.

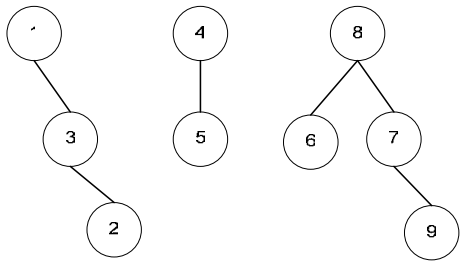
La Quick-Union rappresenta gli insiemi dell'ADT mediante alberi non limitati in altezza e dove la radice dell'albero contiene l'elemento avente etichetta uguale all'etichetta dell'insieme.

Esempio

Sia S formato dai seguenti insiemi: 1={1,2,3}; 4={4,5}; 8= {6,7,8,9}. La seguente è una raffigurazione di una valida rappresentazione Quick Union di S:



così come anche la seguente è una raffigurazione di una valida rappresentazione Quick Union di S:



Valutiamo la complessità delle operazioni di makeset, union e find. Sia ancora *n* il totale del numero di elementi contenuti nei diversi insiemi di *S*.

Makeset, l'operazione di makeset richiede la generazione di un albero con un solo nodo: la radice. La complessità dell'operazione è pertanto costante.

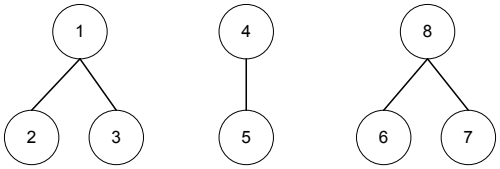
Union, l'operazione di union, similmente alla makeset, ha anch'essa complessità costante. Dati due insiemi *A, B* la union dei due insiemi avviene ponendo la radice

dell'albero rappresentante l'insieme *B* come figlio della radice dell'albero rappresentante l'insieme *A* (esempio in basso);

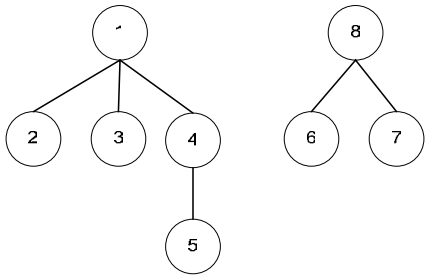
Find, l'operazione di find è linearmente proporzionale all'altezza dell'albero, poichè la ricerca dell'etichetta dell'insieme contenente un elemento *a* richiede la risalita lungo un intero verso dell'albero. Ora, essendo l'altezza massima di un albero Quick Union pari a $O(n)$[1], abbiamo che nel caso pessimo la complessità dell'operazione di find è $O(n)$.

Esempio

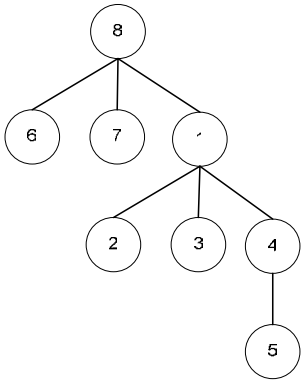
Sia S formato dai seguenti insiemi: 1={1,2,3}; 4={4,5}; 8= {6,7,8}. Sia considerata inoltre la seguente rappresentazione Quick-Union:



l'operazione Union(1, 4) modifica la foresta di alberi come segue:



infine l'operazione Union(8, 1) crea una foresta con un solo albero Quick-Union



[1] La seguente sequenza di union, union(2, 1), union(3, 2), …, union(n, n-1), genera un albero di altezza *n-1*.

3. Euristiche di bilanciamento

Discuteremo in questo capitolo particolari accorgimenti, le euristiche di bilanciamento, introducibili nelle implementazioni degli operatori di union e find al fine di migliorarne la complessità.

3.1 Algoritmo Quick Find bilanciato

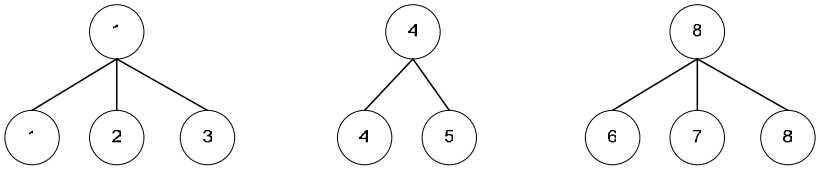
Come già visto, l'utilizzo della rappresentazione Quick Find penalizza l'operazione di Union. Sulla base di questa considerazione e nel caso specifico, gli sforzi volti a garantire un miglioramento della complessità degli algoritmi si sono rivolti esclusivamente verso l'operazione di Union.

L'accorgimento introducibile è sostanzialmente quello di:

1. memorizzare all'interno della radice di ogni albero la cardinalità dell'insieme, ovvero il numero di foglie dell'albero;
2. nella realizzazione dell'operazione di *Union(A, B)*:
   a. spostare le foglie dell'albero rappresentante l'insieme di cardinalità minore verso l'albero rappresentante l'insieme di cardinalità maggiore;
   b. memorizzare l'etichetta da associare al nuovo insieme all'interno della radice dell'albero rappresentante l'insieme unione[2].

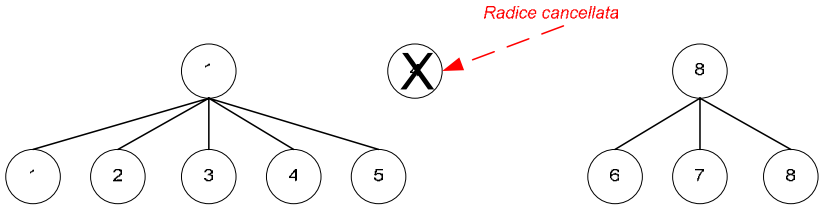
Esempio

Sia S formato dai seguenti insiemi: 1={1,2,3}; 4={4,5}; 8= {6,7,8}. Sia considerata inoltre la seguente rappresentazione Quick-Find:

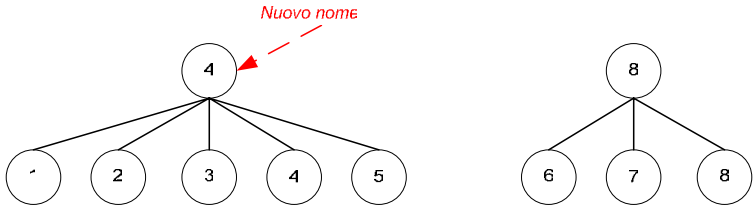


Essendo la cardinalità dell'insieme 1 maggiore della cardinalità dell'insieme 4 l'operazione Union(4, 1) sposterà dapprima gli elementi dell'insieme 4 all'interno dell'albero rappresentante l'insieme 1:

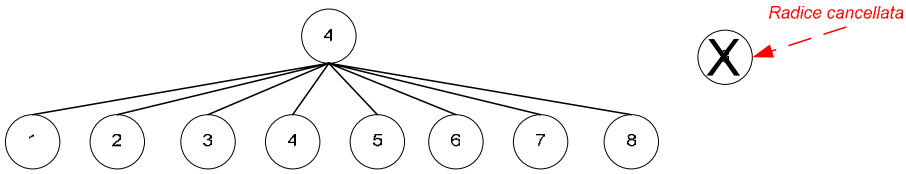
[2] Si rende necessario dar luogo al passo 2b poichè bisogna garantire che l'etichetta associata al nuovo insieme generato da un'operazione *Union(A,B)* sia *A*.



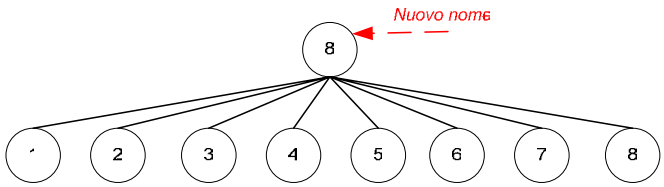
e dopodichè rinominerà l'albero che rappresentava l'insieme 1 con l'etichetta 4:



Analogamente l'operazione Union(8, 4) sposterà prima tutti gli elementi dell'albero rappreesentante l'insieme 8 come figli dell'albero rappresentante l'insieme 4



E dopodichè modificherà l'etichetta dell'albero rimasto con l'esatto valore: 8



Se l'euristica introdotta nell'algoritmo di Quick-Find è abbastanza intuitiva, lo stesso non si può dire per lo studio della complessità del nuovo algoritmo bilanciato. Non verranno pertanto dimostrati i miglioramenti introdotti dal bilanciamento, ci limiteremo solamente ad affermare che il tempo di esecuzione della *Union* applicata ad una tale rappresentazione è circa logaritmica nel numero di makeset effettuate, ovvero nel numero di elementi contenuti all'interno dell'intera foresta.

3.2 Algoritmo Quick Union bilanciato

In maniera del tutto speculare rispetto alla rappresentazione Quick-Find, nella rappresentazione Quick Union gli sforzi di ottimizzazione si sono concentrati esclusivamente sull'operazione di find, che, come abbiamo visto, è l'operazione penalizzata. Ogni accorgimento introdotto nella rappresentazione Quick Union mira fondamentalmente a controllare l'altezza dell'albero, in modo da evitare che essa cresca senza alcun freno a seguito dell'applicazione di sequenze di operazioni di Union.

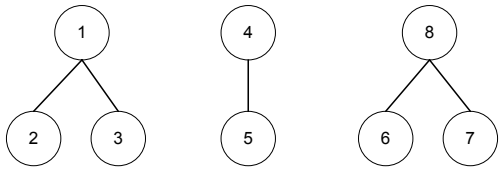
3.2.1 Union by Rank

Con Union by Rank denominiamo una variante della rappresentazione Quick-Union che, al fine di evitare che l'altezza di un albero cresca senza alcun controllo, introduce i seguenti accorgimenti:

1. memorizza all'interno di ogni radice l'altezza dell'albero;
2. nella realizzazione dell'operazione di *Union(A, B)*:
   a. la radice dell'albero avente altezza maggiore diventa padre della radice dell'albero avente altezza minore;
   b. memorizza l'etichetta da associare al nuovo insieme all'interno del nodo diventato radice dell'albero unione[3].

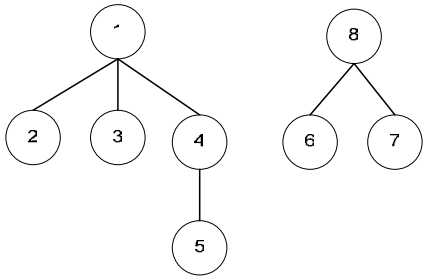
Esempio

Sia S formato dai seguenti insiemi: 1={1,2,3}; 4={4,5}; 8= {6,7,8}. Sia considerata inoltre la seguente rappresentazione Quick-Union.

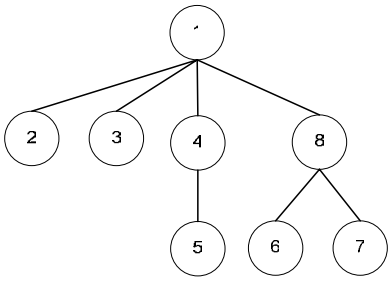


Nell'eseguire l'operazione Union(1, 4), l'algoritmo di Union by Rank unisce i due alberi ponendo indifferentemente come radice dell'albero unione la radice di uno dei due alberi da unire.

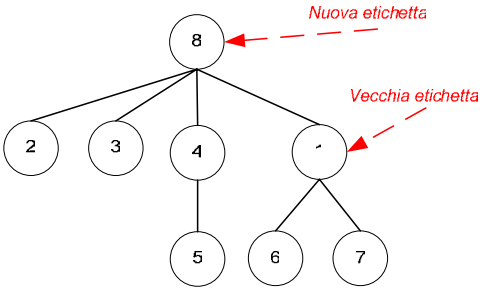
[3] Vedi nota 2



Nell'eseguire l'operazione di Union(8, 1) l'algoritmo di Union by Rank sposta dapprima la radice dell'albero rappresentante l'insieme 8 come figlio dell'albero rappresentante l'insieme 1



e dopodichè modifica la radice dell'albero poichè il nome dell'insieme unione deve essere 8



Al fine di poter valutare i benefici apportati dagli accorgimenti introdotti dall'algortimo *Union by Rank* si rende necessario dimostrare prima i seguenti risultati.

Lemma. *Sia rank(v) l'altezza di un albero Quick Union by Rank. Il numero di nodi contenuti all'interno dell'albero è maggiore o al massimo uguale a* $2^{rank(v)}$.

Dim. Dimostriamo il lemma osservando il comportamento delle sole operazioni che apportano modifiche alla foresta di alberi Quick Union: makeset e union.
Relativamente all'operazione di makeset il lemma è banalmente dimostrato in quanto l'unica modifica apportata alla foresta di alberi è quella di introdurre un nuovo albero costituito dalla sola radice. In tal caso l'altezza dell'albero è zero ed il numero di nodi è uno.
Focalizziamo pertanto la nostra attenzione sulla seconda operazione: la union. Procediamo per induzione. Poniamo come passo induttivo iniziale il momento in cui la foresta è composta da alberi Quick Union rappresentanti insiemi contenenti un solo nodo. In tal caso ogni albero è costituito dalla sola radice ed il lemma è verificato.

Procediamo pertanto ponendo vero il lemma per due alberi quick union rappresentanti due insiemi qualsiasi A e B ed indichiamo con:

- *rank(A), rank(B), rank(A∪B)* rispettivamente il rank della radice dell'albero rappresentante l'insieme A, l'insieme B e l'unione dei due insiemi;
- $|A|$, $|B|$, $|A \cup B|$ rispettivamente il numero di nodi presenti all'interno dell'albero rappresentante l'insieme A, l'insieme B e l'unione dei due insiemi[4].

Si possono verificare i seguenti tre casi:

1. ***rank(B) < rank(A)***, in tal caso la radice dell'albero rappresentante l'insieme A diventa radice dell'albero rappresentante l'unione e pertanto:
   i. $rank(A \cup B) = rank(A)$;
   ii. $|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} \geq 2^{rank(A)} = 2^{rank(A \cup B)}$
2. ***rank(B) > rank(A),*** in tal caso la radice dell'albero rappresentante l'insieme B diventa radice dell'albero rappresentante l'unione e pertanto:
   i. $rank(A \cup B) = rank(B)$;
   ii. $|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} \geq 2^{rank(B)} = 2^{rank(A \cup B)}$
3. ***rank(B) = rank(A)***, in tal caso la radice dell'albero rappresentante l'insieme A diventa radice dell'albero rappresentante l'unione e pertanto:
   i. $rank(A \cup B) = rank(A) + 1$;
   ii. $|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} = 2 * 2^{rank(A)} = 2^{rank(A)+1} = 2^{rank(A \cup B)}$

Conseguenza immediata del lemma appena dimostrato è rappresentata dal seguente:

Corollario. *Sia n il numero di makeset effettuate, si ha che l'altezza di un albero Quick Union by Rank è O(log n).*

Dimostrazione. Sia v la radice di un albero Quick-Union by Rank ed indichiamo con *size(v)* il numero di nodi presenti nell'albero. Si ha che:

i. $size(v) \leq n$, cioè l'albero contiene al massimo tutti i nodi creati dalle makeset
ii. $size(v) \geq 2^{rank(v)}$, dal lemma sopra dimostrato

Ne deriva quindi che: $n \geq 2^{rank(v)} \Rightarrow log(n) \geq rank(v)$

[4] Si osservi che |A∪B| = |A| + |B|

Il corollario appena dimostrato ci permette di concludere l'analisi della valutazione dei benefici apportati dalla ottimizzazione oggetto della discussione. Infatti, considerando che l'algoritmo di find applicato in tale contesto ha una complessità linearmente proporzionale all'altezza dell'albero possiamo affermare che:

Teorema. *In una rappresentazione mediante alberi Quick Union by Rank l'operazione di makeset e union hanno complessità costante, mentre l'operazione di find ha complessità pari a O(log n) dove n rappresenta il numero di makeset eseguite.*

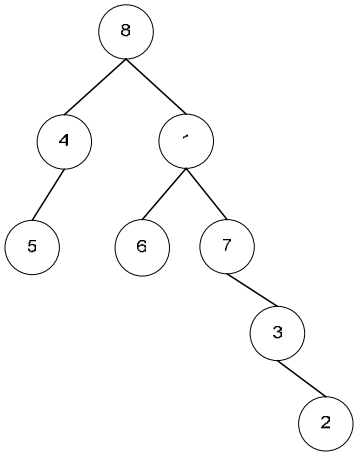
3.2.2 Path Compression

Sempre nell'ambito della rappresentazione Quick-Union, è possibile introdurre ulteriori accorgimenti volti a migliorare la complessità dell'operazione di find. Vedremo in questo paragrafo l'euristica di *Path Compression* avente come obiettivo la riduzione dell'altezza di un albero.

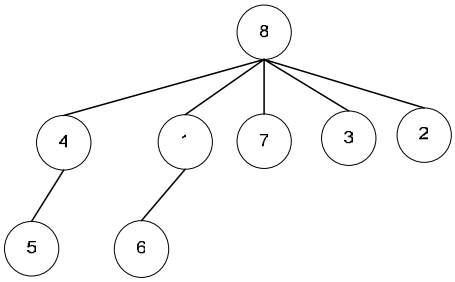
L'euristica di *Path Compression* si serve dell'algoritmo di find facendo leva sul movimento che esso esegue nella ricerca dell'etichetta posta alla radice: risalita lungo un intero verso dell'albero. L'idea alla base di tale euristica è quella di assegnare un ulteriore compito all'algoritmo di find: ristrutturare l'albero ponendo il puntatore al padre di ogni nodo incontrato lungo il cammino uguale alla radice dell'albero.
Eseguiamo in tal modo una compressione dell'altezza dell'albero lungo tutto il cammino che dal nodo contenente l'elemento argomento della find termina nella radice.

Esempio

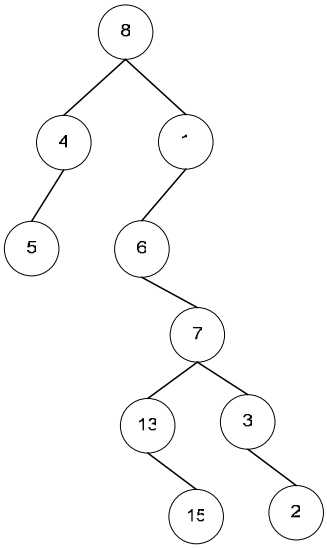
Sia considerato il seguente albero Quick-Union:



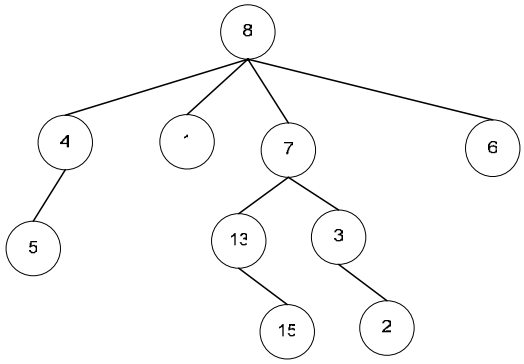
l'applicazione dell'algoritmo di Find(2) modificato secondo l'euristica di path compression ristruttura l'albero nel seguente modo:



Analogamente sia considerato il seguente albero Quick-Union



l'esecuzione dell'operazione find(7) modifica la struttura nel seguente modo



Senza alcun dubbio, l'euristica di *Path Compression* appesantisce l'algoritmo di find. Esso dovrà a questo punto eseguire, durante la risalita e livello per livello, un'operazione in più. Di contro, però, la successiva operazione di *find* richiamata su uno dei qualsiasi nodi precedentemente modificati sarà sicuramente più efficiente.

Anche se la tecnica introdotta dall'euristica di bilanciamento è abbastanza intuitiva, l'analisi della complessità dell'operazione di *find* così modificata è particolarmente laboriosa. Ci limiteremo, pertanto, solamente ad esporre i risultati di tale analisi. Per far questo abbiamo, però, bisogno di introdurre la funzione di *Ackerman*.

Definizione. Si definisce funzione di *Ackerman* la funzione F definita nell'insieme dei numeri naturali e descritta dalla seguente equazione di ricorrenza:

$$\begin{cases} F(0) = 1 \\ F(i) = 2^{F(i-1)} \quad \forall\, i > 0 \end{cases}$$

Esempio

La seguente tabella riporta i valori assunti dalla funzione di Ackerman per n=1,2,3,4,5.

n	***F(n)***
1	$F(1) = 2^1 = 2$
2	$F(2) = 2^{F(1)} = 2^2 = 4$
3	$F(3) = 2^{F(2)} = 2^4 = 16$
4	$F(4) = 2^{F(3)} = 2^{16} = 65536$
5	$F(5) = 2^{F(4)} = 2^{65536}$

La proprietà caratterizzante la funzione di Ackerman è quella di crescere in modo particolarmente veloce. Proprietà opposta, crescere in maniera particolarmente lenta, caratterizza invece l'inversa della funzione di Ackerman, *G(n)*, così definita:

$$G(n) = min\{ z \in N \mid F(z) \geq n \}$$

Esempio

Seguono alcuni valori assunti dalla inversa della funzione di Ackerman

N	***G(n)***
1	*G(1) = 1*
2	*G(2) = 1*
3	*G(3) = 2*
4	*G(4) = 2*
5	*G(5) = 3*
6	*G(6) = 3*
7	*G(7) = 3*
...	*....*
16	*G(16) = 3*
17	*G(17) = 4*
18	*G(18) = 4*
....	*...*
65536	*G(65536) = 4*
65537	*G(65537) = 5*
65538	*G(65538) = 5*
....	*....*
2^{65536}	$G(2^{65536}) = 5$

I valori assunti dall'inversa della funzione di *Ackerman* limitano superiormente la complessità di una sequenza qualsiasi composta da operazioni di *Union* e *Find*. Più precisamente:

Teorema. *La complessità di una sequenza qualsiasi di m operazioni Makeset, Union e Find, n delle quali sono operazioni di MakeSet, applicate in una rappresentazione mediante alberi Quick Union bilanciati con la combinazione dell'euristiche by rank e path compression è nel caso peggiore pari a O(m G(n)).*

Dim: omessa.

3.3 Off Line Min Problem: un'applicazione dell'ADT Union Find

Vediamo un esempio di utilizzo dell'ADT Union Find per la risoluzione di un problema. Risolviamo l'off line min problem.

Sia *S* un contenitore di valori interi per il quale sono definite esclusivamente due operazioni: *insert* ed *extractmin*. Formalmente:

Tipo di dato: IntegerSet
Collezione di interi

Operazioni:

Insert(intero i)
Inserisce l'intero 'i' all'interno della collezione

ExtractMin()→intero
Estrae il minimo intero presente all'interno della collezione

Indichiamo, ora, con σ una sequenza qualsiasi di operazioni di inserimenti ed estrazioni di valori *i* dove:

- $1 \leq i \leq n$ per un valore intero *n* qualsiasi;
- l'operazione di insert si ripete al massimo una volta per ogni valore *i*.

Data una sequenza σ, l'*off line min problem* consiste nell'individuare quale sarà la successione di valori estratti dalle operazioni *ExtractMin* presenti in σ.

Il problema dell'*off line min problem* può essere risolto mediante l'utilizzo dell'ADT Union Find applicando la seguente metodologia. Indichiamo con *k* il numero di operazioni di estrazione eseguite e riscriviamo l'intera sequenza σ nel seguente modo:

$$\sigma = \sigma_1 E \ldots E \sigma_k E \sigma_{k+1}$$

dove il simbolo *E* rappresenta l'avvenuta esecuzione di un'operazione di estrazione mentre, per ogni $j=1,\ldots,k+1$, il simbolo σ_j rappresenta una sottosequenza di sole operazioni di inserimento.

A partire da questa rappresentazione creiamo dapprima un set di $k+1$ insiemi, $S=\{1, 2, \ldots, k+1\}$, dove l'insieme i-esimo contiene tutti gli elementi inseriti nella sottosequenza σ_i, e dopodichè diamo vita alla seguente procedura dove con *succ(j)* indichiamo l'insieme che nella rappresentazione *Union Find* è posto alla destra dell'insieme *j*.

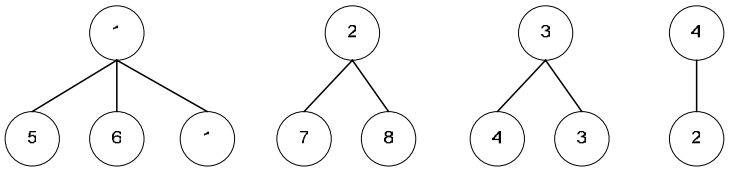
```
OffLineMinProblem
{
        UnionFindS
        Insieme    J

        i = 1
        fin quando ( i è minore o uguale al valore massimo n )
        {
                J = S.Find( i )
                se ( j è minore al numero di estrazioni eseguite k  )
                {
                        i è stato estratto alla j-esima estrazione
                        S.Union( succ(J), J )
                }
                altrimenti
                {
                        l'elemento i non è stato estratto
                }

                i = i + 1
        }
}
```

Esempio

Sia σ = { insert(5), insert(6), insert(1), extractmin, insert(7), insert(8), extractmin, insert(4), insert(3), extractmin, insert(2) } Possiamo rappresentare tale sequenza come:

$$\sigma = \{ 5, 6, 1, E, 7, 8, E, 4, 3, E, 2\}$$

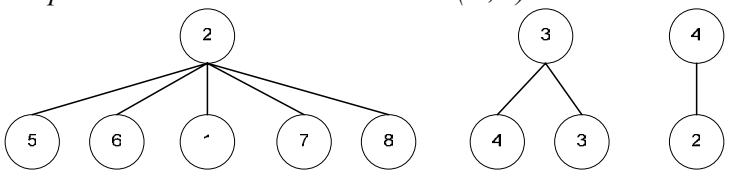
Utilizziamo per comodità di esposizione la rappresentazione Quick Find e creiamo la foresta di alberi rappresentante l'insieme S={1, 2, 3, 4}



Eseguiamo quindi l'algoritmo OffLineMinProblem.

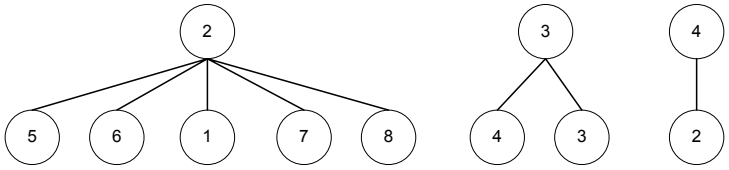
Passo 1

Variabili: i = 1; J = find(1) = 1
Azioni: l'elemento 1 è stato prelevato nella prima estrazione.
Uniamo infine il primo ed il secondo albero: Union(2, 1) .



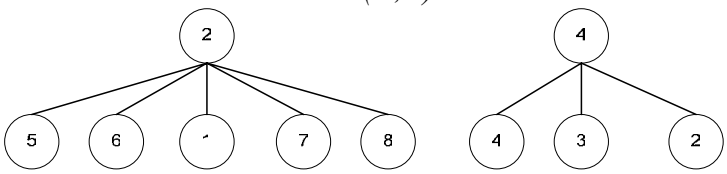
Passo 2

Variabili: i = 2; J = find(2) = 4
Azioni: l'elemento 2 non è stato mai estratto.



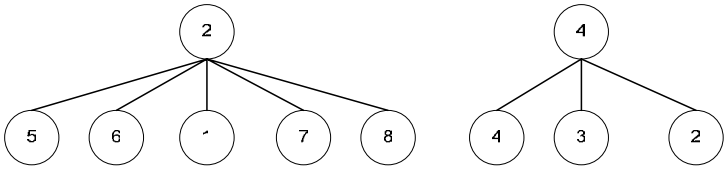
Passo 3

Variabili: i = 3; J = find(3) = 3
Azioni: l'elemento 3 è stato prelevato nella terza estrazione.
Uniamo il secondo ed il terzo albero: Union(4, 3) .



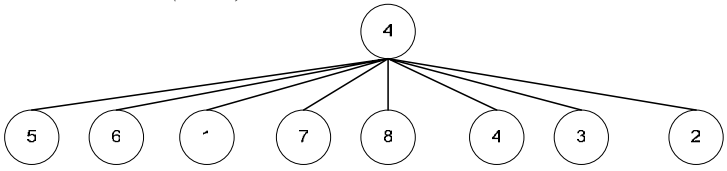
Passo 4

Variabili: i = 4; J = find(4) = 4
Azioni: l'elemento 4 non è stato mai estratto.



Passo 5

Variabili: i = 5; J = find(5) = 2
Azioni: l'elemento 5 è stato prelevato nella seconda estrazione.
Uniamo i due alberi: Union(4, 2) .



L'esecuzione delle successive Find evidenzieranno che gli elementi 6, 7, 8 non verranno mai estratti.