Reprezentarea mulțimilor disjuncte

Definiție

Fie mulțimea $U=\{1, 2, ... , n\}$, $n\geq 1$. Sistemul $S=\{S_1, S_2, ... , S_k\}$, $k\geq 1$, constituie o *partiție* a mulțimii U dacă sunt îndeplinite următoarele condiții :

1. $S_i \subseteq U, \forall i\in\{1, 2, ... , k\}$
2. $S_i \neq \emptyset, \forall i\in\{1, 2, ... , k\}$
3. $S_i \cap S_j = \emptyset, \forall i\neq j, i, j \in\{1, 2, ..., k\}$
4. $S_1\cup S_2 \cup ... \cup S_k = U$.

Dată fiind mulțimea U şi $S=\{S_1, S_2, ... , S_k\}$ o partiție a mulțimii U, problema constă în a proiecta o structură de date care să permită executarea eficientă a următoarelor două operații fundamentale:

– `find(x)`: determină mulțimea $S_i\subseteq S$ căreia îi aparține elementul x, $x\in U$;
– `union(Si,Sj)`: reuneşte mulțimea S_i cu mulțimea S_j, obținându-se un nou element al mulțimii S, ce va înlocui S_i şi S_j.

Observație

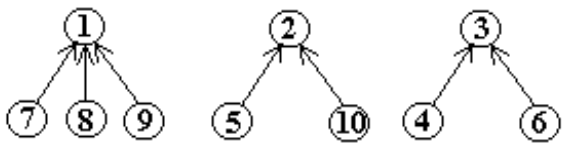
Această problemă este cunoscută şi sub denumirea de *problema claselor de echivalență* (x, y se vor numi echivalente dacă aparțin aceleiaşi mulțimi $S_i\subseteq S$, sau, altfel spus, `find(x)=find(y)`), conceptele de relație de echivalență şi partiție reprezentând două abordări diferite ale aceleiaşi structuri matematice.

De exemplu, dat fiind un graf neorientat, componentele conexe ale grafului constituie o partiție a mulțimii vârfurilor grafului. Un algoritm de determinare a componentelor conexe ale unui graf neorientat poate fi formulat cu ajutorul operațiilor *union–find* astfel:

```
Pentru ∀i∈{1, 2, ... , n}, Si := {i};
 Pentru ∀[i, j] muchie în graf
        A := find(i);
        B := find(j);
        dacă A≠B atunci union(A, B);
```

O soluție eficientă este reprezentarea mulțimilor disjuncte cu ajutorul arborilor cu rădăcină. Fiecare arbore reprezintă o mulțime, iar fiecare nod din arbore un element al mulțimii. Rădăcina arborelui va fi elementul reprezentativ al mulțimii.

De exemplu, să considerăm $n=10$ şi o partiție a mulțimii $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ formată din $S_1=\{1, 7, 8, 9\}$, $S_2 = \{2, 5, 10\}$, $S_3=\{3, 4, 6\}$. Partiția $S=\{S_1, S_2, S_3\}$poate fi reprezentată ca o pădure formată din trei arbori:

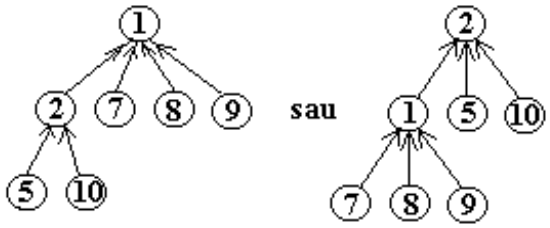


Reprezentăm arborii prin referințe ascendente, deci pentru fiecare nod din arbore, cu excepția rădăcinii, reținem legătura spre părintele său: `tata(x)`= nodul părinte al lui x, sau 0, dacă x este rădăcina arborelui.

Operația `find(x)` constă în a determina rădăcina arborelui al cărui nod este x.

```
int find(int x)
{ while (tata[x]) x=tata[x];
  return x;}
```

Operația *union* se poate realiza transformând unul din arbori în subarborele celuilalt. Reuniunea $S_1 \cup S_2$ poate fi reprezentată în două moduri:



Deci rădăcina unuia din arbori devine părintele rădăcinii celuilalt arbore.

```
void union(int i, int j)
{tata[i]=j;}
```

Aceşti algoritmi sunt foarte simpli, dar nu sunt eficienți.

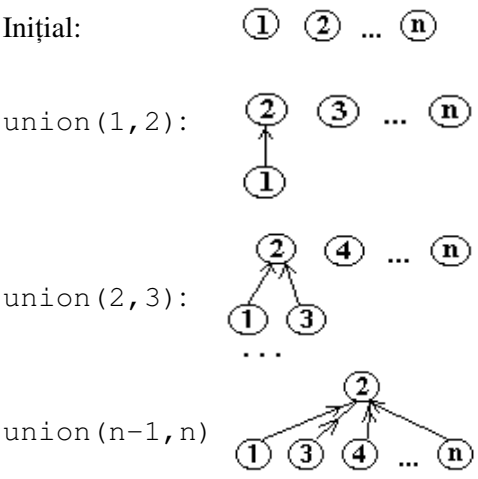
Să considerăm, de exemplu partiția `S={{1}, {2}, ..., {n}}`. Deci configurația inițială constă dintr-o pădure în care fiecare arbore este format doar din rădăcină: `tata(i)=0`, `∀i∈{1, 2, ..., n}`. Să considerăm acum următoarea secvență de operații `union` şi `find`: `union(1,2), find(1), union(2,3), find(1), union(3,4), find(1), ..., find(1),union(n-1,n).`

Această secvență conduce la următorul arbore degenerat din figura alăturată.

Cum timpul necesar operației `union` este constant, cele `n-1` operații `union` au timpul de execuție de `O(n)`. Dar fiecare operație `find(1)` parcurge tot drumul de la nodul `1` până la rădăcină. Cum timpul de execuție necesar operației `find` pentru un nod de pe nivelul `i` este `O(i)`, obținem un timp de execuție de $O(n^2)$ pentru cele `n-2` operații `find(1)`.

Putem îmbunătăți eficiența operațiilor `union` şi `find`, aplicând următoarea *regulă de ponderare* pentru `union(i,j)`: dacă numărul de niveluri din arborele cu rădăcina `i` este mai mic decât numărul de niveluri din arborele cu rădăcina `j`, atunci `j` va deveni părintele lui `i`, altfel `i` va deveni părintele lui `j`.

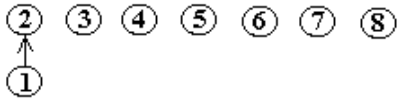
Utilizând regula de ponderare secvența de operații `union` şi `find` de mai sus va conduce la următorii arbori:



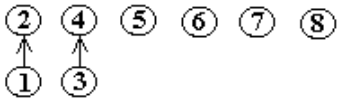
În acest caz timpul de execuție necesar operațiilor `find` este `O(n)`, deoarece arborii obținuți au înălțimea cel mult egală cu `1`. Totuşi, există cazuri mai defavorabile.

De exemplu, fie `n=8`. Inițial vom avea pădurea:

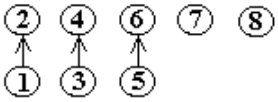
union(1,2)



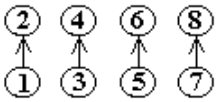
union(3,4)



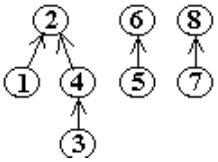
union(5,6)



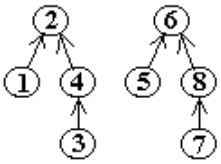
union(7,8)



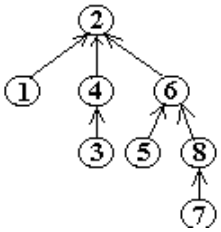
union(3,4)



union(3,4)



union(3,4)



Înălțimea arborelui obținut este $3=\log_2 8$.

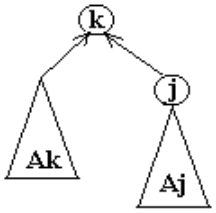
Lemă

Dacă `A` este un arbore cu `n` noduri obținut în urma aplicării operației `union` folosind regula de ponderare, înălțimea arborelui `A` este cel mult egală cu `[log n]`.

Demonstrație

Vom proceda prin inducție după `n`, numărul de noduri. Pentru `n=1`, rezultatul este evident.

Presupunem afirmația adevărată pentru arbori cu `i` noduri (`1≤i≤n-1`), obținuți în urma aplicării algoritmului `union`. Să demonstrăm că înalțimea oricărui arbore cu `n` noduri A_n, obținut în urma aplicării algoritmului `union` este cel mult egală cu `[log n]`.



Fie `union(j,k)` ultima operație `union` executată pentru obținerea arborelui A_n. Notăm cu `m` numărul de noduri din arborele cu rădăcina `j`. Arborele cu rădăcina `k` are `m-n` noduri. Putem presupune, fără a restrânge generalitatea, că `1≤m≤n/2`. Deci înălțimea `h` a arborelui A_n va fi egală cu înălțimea arborelui cu rădăcina `k` sau cu o unitate mai mare decât înălțimea arborelui cu rădăcina `j`.

În primul caz: `h≤[log (n-m)]≤[log n]`.

În cel de-al doilea caz: `h≤[log m]≤[log (n/2)]+1≤[log n]`.

Pentru a aplica regula de ponderare este necesar ca pentru fiecare arbore să cunoaştem numărul de niveluri. Fie `h`, un vector în care `h[i]` reprezintă numărul de niveluri din arborele cu rădăcina `i`. Acest vector va fi actualizat eventual la operații `union`.

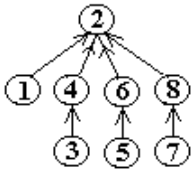
```
void union_ponderare(int x, int y)
{if (h[x]>h[y]) tata[y]=x;
     else
     {tata[x]=y;
      if (h[x]=h[y]) h[y]++;}
}
```

Lema 1 garantează un timp de execuție de `O(log n)` pentru operația `find(x)`, pentru orice nod `x` dintr–un arbore obținut prin operații `union` folosind regula de ponderare.

Pentru a îmbunătați în continuare timpul de execuție a operației `find(x)`, vom utiliza următoarea *regulă de compresie*: orice nod `y` de pe drumul de la rădăcina arborelui la nodul `x` va deveni fiu al rădăcinii arborelui.

```
int find_compresie(int x)
{int r=x;
 while (tata[r]) r=tata[r];                  // r este acum radacina arborelui
 int y=x;
 while (y!=r)                                //aplica regula de compresie
        {t=tata[y];
         tata[y]=r;
         y=t;}
return r;}
```

Funcția `find_compresie()` parcurge drumul de la nodul `x` la rădăcină de două ori, o dată pentru determinarea rădăcinii, a doua oară pentru comprimarea drumului de la `x` la rădăcină. Totuşi, această modificare reprezintă o îmbunătățire, deoarece într–o secvență de operații `union-find`, timpul total de execuție va fi mai mic.



Să considerăm arborele obținut prin secvența de operații `union` din exemplul precedent Executând prima oară operația `find(8)` obținem arborele alăturat:

Au fost necesare trei deplasări pe legături ascendente pentru a identifica rădăcina arborelui şi apoi, pentru compresia drumului de la rădăcină la nodul `8` alte două deplasări. Dar următoarele apeluri ale funcției `find(8)` vor necesita o singură deplasare până la rădăcina arborelui. Astfel costul total al unei secvențe de operații `find(8)` va fi mai mic.

În 1975 *Tarjan* a demonstrat o lemă ce caracterizează comportarea în cazul cel mai defavorabil a algoritmilor *union-find* care utilizează regula de ponderare şi regula de compresie.