Podatkovna struktura za disjunktne množice

↓↓↓
Damjan Strnad

→ **Pri reševanju praktičnih problemov z računalniškimi algoritmi pogosto naletimo na naslednjo situacijo: dano je večje število objektov, ki jih združujemo v vedno večje množice, tako da v vsakem trenutku vsak objekt pripada natanko eni množici.**

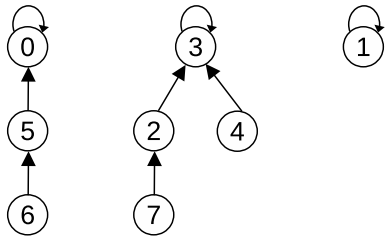
Pri tem na trenutni skupini množic pogosto izvajamo naslednji dve operaciji:

- preverjanje, ali dva objekta x in y pripadata isti množici, in
- združevanje množic, ki jima pripadata objekta x in y.

Običajno začetno stanje je takšno, da vsak objekt predstavlja samostojno množico, te pa nato zaporedoma združujemo.

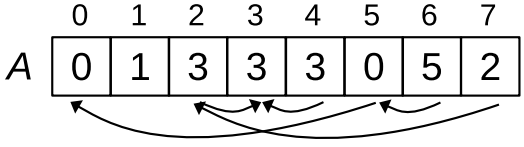
Za množici, ki nimata skupnih elementov, pravimo, da sta *disjunktni* (disjoint). Objekti so torej v vsakem trenutku razporejeni v skupino disjunktnih množic, katerih število se z njihovim združevanjem samo manjša. Z naivno implementacijo disjunktnih množic lahko dosežemo hitro izvajanje ene od operacij, ne pa obeh hkrati. Če npr. disjunktne množice predstavimo s seznami elementov, bo združevanje množic hitro, za preverjanje pripadnosti elementov isti množici pa bomo morali izvesti pregled seznama in pri tem v splošnem opraviti reda N primerjav, kjer je N število elementov. Po drugi strani bi bila implementacija s poljem, v katerem hranimo oznake množic za posamezne elemente, neučinkovita pri združevanju množic, kjer bi morali posodobiti reda N oznak v polju. Potrebujemo torej boljšo rešitev. V tem prispevku bomo opisali implementacijo *podatkovne strukture za disjunktne množice* (disjoint-set data structure), ki omogoča učinkovito izvajanje zaporedja prej opisanih operacij. Ker to dosežemo z implementacijo dveh metod, imenovanih `IŠČI` (`FIND`) in `UNIJA` (`UNION`), to podatkovno strukturo v literaturi pogosto imenujejo tudi *podatkovna struktura `UNIJA–IŠČI`* (union-find data structure).

Pri implementaciji podatkovne strukture za disjunktne množice je vsaka množica predstavljena kot drevo, v katerem so elementi množice hierarhično povezani preko kazalcev na starše, pri čemer koren drevesa kaže sam nase. Kot primer vzemimo množico objektov, ki so označeni s celimi števili od 0 do 7. Če so ti objekti razdeljeni v tri disjunktne množice $\{0, 5, 6\}$, $\{2, 3, 4, 7\}$ in $\{1\}$, lahko to grafično ponazorimo s sliko 1. Oblika posameznih dreves je lahko tudi drugačna in je odvisna od tega, v kakem vrstnem redu smo množice združevali.



SLIKA 1.
Predstavitev disjunktnih množic z drevesi, v katerih so elementi povezani s kazalci na starše. Reprezentativni element množice je koren, ki kaže sam nase.

V podatkovni strukturi za disjunktne množice je vsaka množica enolično določena z njenim *reprezentativnim elementom*, ki je v tem primeru koren drevesa. V nadaljevanju bomo zato za reprezentativni element množice uporabljali kar krajši izraz *koren* (root). Pripadnost dveh objektov isti množici lahko ugotavljamo s preverjanjem enakosti njunih korenov, pri združevanju dveh množic pa koren unije postane eden od dosedanjih dveh korenov. Podatkovna struktura za disjunktne množice je torej neke vrste nadstruktura ali gozd dreves, ki predstavljajo posamezne disjunktne množice. Označitev objektov z zaporednimi celimi števili od 0 naprej omogoča še posebej elegantno programsko predstavitev dreves v strnjenem polju A dolžine N, v katerem i-ti element polja hrani oznako starša objekta i. Disjunktne množice s slike 1 bi lahko tako opisali s poljem na sliki 2.



SLIKA 2.
Zapis drevesnih struktur s slike 1 s poljem. Vrednost na položaju i v polju je indeks starša objekta i.

Ob inicializaciji podatkovne strukture za disjunktne množice z N objekti je potrebno ustvariti N množic, katerih koreni (in hkrati edini elementi) so posamezni objekti. Pri zgoraj opisani implementaciji s poljem A je postopek zelo enostaven, potrebno je le vsem objektom postaviti »kazalec« nase (algoritem 1).

Algoritem 1 Inicializacija disjunktnih množic

```
function INICIALIZACIJA(N)
    for i ← 0...N−1 do
        A[i] ← i
    end for
end function
```

Ključni metodi, ki ju implementira podatkovna struktura za disjunktne množice, sta že omenjeni `IŠČI` in `UNIJA`. Metoda `IŠČI` kot argument prejme oznako objekta in vrne oznako korena disjunktne množice, ki ji objekt pripada. Osnovna implementacija metode je zelo preprosta, saj je potrebno le slediti verigi staršev od danega objekta navzgor proti korenu. Slednjega prepoznamo po tem, da kaže sam nase. Postopek je v obliki rekurzivne funkcije zapisan v algoritmu 2, možna pa je tudi iterativna implementacija z enako časovno zahtevnostjo, ki zaporedje prednikov hrani na skladu.

Algoritem 2 Osnovna metoda `IŠČI`

```
function IŠČI(x)
    if A[x]=x then
        return x
    else
        return IŠČI(A[x])
    end if
end function
```

Problem zgornjega postopka je v tem, da bomo ob naslednjem klicu `IŠČI` z istim argumentom spet morali prehoditi isto zaporedje kazalcev, kar postane ob velikem številu ponavljajočih se klicev neučinkovito. Podatkovna struktura za disjunktne množice zato uporabi t. i. *stiskanje poti* (path compression), pri katerem ob vračanju iz rekurzije vsem objektom na poti postavimo kazalec na starša na najdeni koren množice, kot prikazuje algoritem 3.

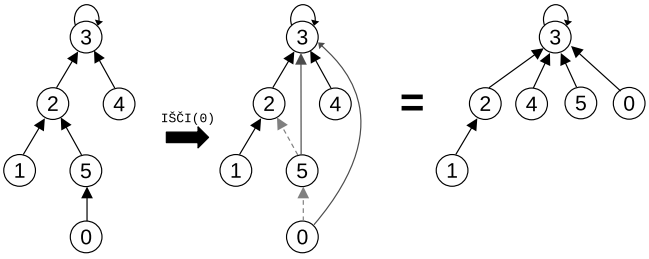
Algoritem 3 Metoda `IŠČI` s stiskanjem poti

```
1: function IŠČI(x)
2:     if A[x]=x then
3:         return x
4:     else
5:         A[x] ← IŠČI(A[x])
6:         return A[x]
7:     end if
8: end function
```

Princip delovanja stiskanja poti prikažimo na zgledu disjunktne množice na levi strani slike 3. Po izvedbi klica `IŠČI(0)` bo novo stanje drevesa in pripadajočega polja takšno, kot je prikazano na desni strani slike. Vsak naslednji klic `IŠČI(0)` ali `IŠČI(5)` se bo sedaj zaključil v enem koraku.

S tako definirano metodo `IŠČI` lahko preverjanje, ali objekta x in y pripadata isti disjunktni množici, izvedemo s primerjavo `IŠČI(x)=IŠČI(y)`.

Druga ključna operacija na podatkovni strukturi za disjunktne množice je združevanje ali unija dveh množic. Metoda `UNIJA` kot argument prejme dva



SLIKA 3.
Stiskanje poti pri klicu IŠČI(0) poveže vse elemente prehojene verige neposredno s korenom množice, zaradi česar bodo naslednji klici IŠČI bolj učinkoviti.

objekta in izvede združevanje disjunktnih množic, ki jima ta objekta pripadata. Če objekta že pripadata isti disjunktni množici, se ne zgodi nič. V nasprotnem primeru je potrebno povezati drevesi obeh množic tako, da koren ene množice priključimo kot naslednika korenu druge množice, ki s tem postane koren celotne unije. Združevanje dveh dreves si želimo izvesti tako, da bo imelo drevo unije čim manjšo višino, saj bo povprečna dolžina poti v takšnem drevesu manjša in bo iskanje korena zato učinkovitejše. Kadar torej združujemo dve drevesi različnih višin, je potrebno nižje drevo priključiti višjemu, katerega višina se zaradi tega ne spremeni (slika 4 levo). Če pa združujemo dve drevesi enake višine, je smer priključevanja nepomembna, višina združenega drevesa pa bo za ena večja (slika 4 desno).

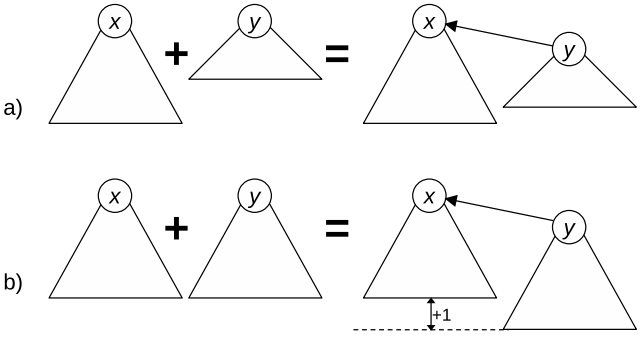
Za učinkovito implementacijo unije je torej potrebno voditi višine dreves. Ker pa se višina drevesa zaradi stiskanja poti lahko spremeni tudi ob izvajanju klicev IŠČI, je beleženje in posodabljanje točne višine dreves nepraktično. V podatkovni strukturi za disjunktne množice zato vodimo samo *range* (rank) posameznih dreves. Rang drevesa je zgornja meja višine drevesa, ki ne odraža nujno njegove dejanske višine, ampak samo njeno največjo možno vrednost. Pri združevanju množic priključimo množico z nižjim rangom tisti z višjim rangom, kar imenujemo *unija po rangu* (union by rank). Za beleženje rangov uporabimo ločeno polje R, v katerem so veljavni rangi zapisani samo pri objektih, ki so koreni svojih disjunktnih množic (algoritem 4). Začetne vrednosti vseh rangov pri inicializaciji podatkovne strukture za disjunktne množice (algoritem 1) postavimo na 0.

Algoritem 4 Unija po rangu

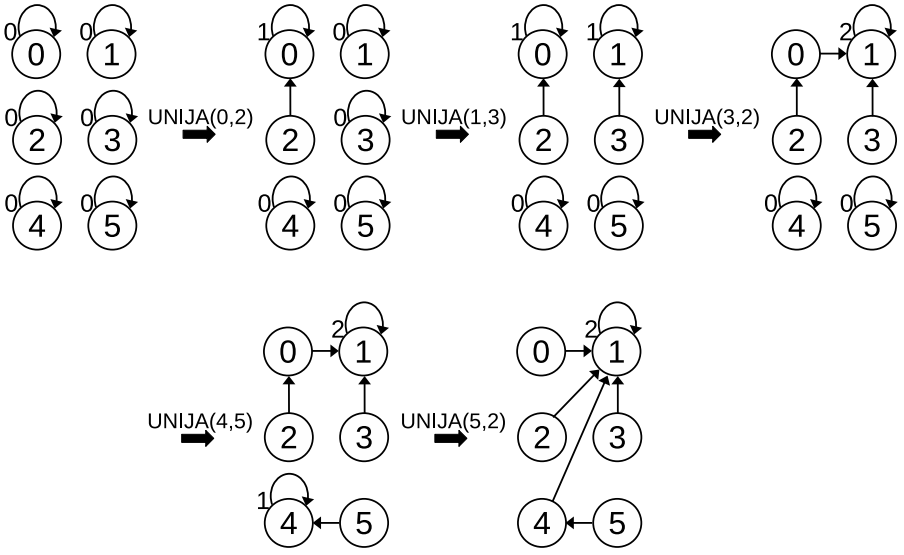
```
1:  function UNIJA(x,y)
2:      a ← IŠČI(x)
3:      b ← IŠČI(y)
4:      if a≠b then
5:          if R[a] ⩾ R[b] then
6:              A[b] ← a
7:              if R[a] = R[b] then
8:                  R[a] = R[a] + 1
9:              end if
10:         else
11:             A[a] ← b
12:         end if
13:     end if
14: end function
```

Zgled zaporednega združevanja disjunktnih množic je prikazan na sliki 5, pri čemer so rangi posameznih množic zapisani ob korenskem vozlišču.

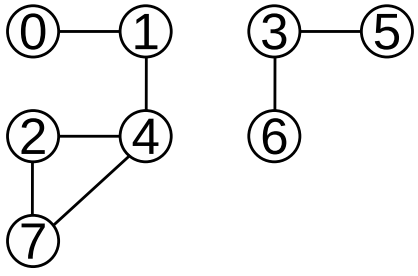
V praktičnih aplikacijah običajno želimo za posamezne disjunktne množice voditi še dodatne opisne parametre, kot je npr. število objektov v množici. Tudi sami objekti imajo lahko lastne številske atribute, ki jih želimo pri združevanju množic na določen način zlivati (npr. vsota ali povprečje vrednosti atributa elementov množice). Vsako od teh statistik lahko beležimo z ločenim dodatnim poljem, v katerem trenutno vrednost za vsako množico hranimo na indeksu njenega korena (na podoben način kot



SLIKA 4.
Pri združevanju disjunktnih množic v primeru različno visokih dreves manjše drevo priključimo večjemu, zato da višina drevesa ostane enaka (a). V primeru enako visokih dreves je smer povezovanja nepomembna, višina združenega drevesa pa se poveča za ena (b).



SLIKA 5.
Primer zaporednega izvajanja unije po rangu. V zadnjem koraku se pri iskanju korena disjunktne množice za objekt 2 izvede tudi stiskanje poti. Bodimo pozorni na to, da je vrstni red argumentov klica UNIJA pomemben, ko združujemo drevesa z enakim rangom (npr. klic UNIJA(2,3) v tretjem koraku bi tvoril drugačno drevo).



SLIKA 6.
Predstavitev stikov (povezave) med osebami (vozlišča) z grafom. Vsak povezan del grafa predstavlja neodvisen mehurček.

prej rang v polju R). Včasih pa nas tudi zanima samo število disjunktnih množic na koncu, kar je prav tako enostavno ugotoviti – po zaključku združevanja se sprehodimo skozi polje A in preštejemo primere, ko je $A[i] = i$.

Najbolj znan primer aplikacije podatkovne strukture za disjunktne množice je vodenje minimalnih vpetih dreves pri Kruskalovem algoritmu, o katerem je bilo v Preseku v preteklosti že pisano. Našo obravnavo zato zaključimo z naslednjim, za trenutne čase precej aktualnim primerom: V populaciji N oseb razsaja prenosljiva virusna bolezen, ki pa jo oboleli preboli v sedmih dneh. Ker se testiranje še ni začelo, se ne ve, kdo je okužen, imamo pa podatke o tem, kdo je bil s kom v stiku v zadnjih sedmih dneh. Da preprečimo nadaljnje širjenje bolezni, želimo oblikovati

1. po inicializaciji:

	0	1	2	3	4	5	6	7
A	0	1	2	3	4	5	6	7
R	0	0	0	0	0	0	0	0
C	1	1	1	1	1	1	1	1

2. po obravnavi (0,1):

	0	1	2	3	4	5	6	7
A	0	0	2	3	4	5	6	7
R	1	0	0	0	0	0	0	0
C	2	1	1	1	1	1	1	1

3. po obravnavi (1,4):

	0	1	2	3	4	5	6	7
A	0	0	2	3	0	5	6	7
R	1	0	0	0	0	0	0	0
C	3	1	1	1	1	1	1	1

4. po obravnavi (2,4):

	0	1	2	3	4	5	6	7
A	0	0	0	3	0	5	6	7
R	1	0	0	0	0	0	0	0
C	4	1	1	1	1	1	1	1

5. po obravnavi (2,7):

	0	1	2	3	4	5	6	7
A	0	0	0	3	0	5	6	0
R	1	0	0	0	0	0	0	0
C	5	1	1	1	1	1	1	1

6. po obravnavi (3,5):

	0	1	2	3	4	5	6	7
A	0	0	0	3	0	3	6	0
R	1	0	0	1	0	0	0	0
C	5	1	1	2	1	1	1	1

7. po obravnavi (3,6):

	0	1	2	3	4	5	6	7
A	0	0	0	3	0	3	3	0
R	1	0	0	1	0	0	0	0
C	5	1	1	3	1	1	1	1

8. po obravnavi (4,7):

	0	1	2	3	4	5	6	7
A	0	0	0	3	0	3	3	0
R	1	0	0	1	0	0	0	0
C	5	1	1	3	1	1	1	1

SLIKA 7.
Postopek reševanja problema z mehurčki

→ »mehurčke«, tj. skupine oseb, ki so bile v omenjenem času v neposrednem ali posrednem stiku. Vsak stik je podan kot par oseb (x, y), ki sta bili v stiku. Zaradi zaščite osebnih podatkov so osebe označene s števili od 0 do $N - 1$. Zanima nas število mehurčkov in velikost največjega mehurčka.

Kot zgled podajmo primer z osmimi osebami, od katerih so bili v zadnjem tednu v stiku pari $(0, 1), (1, 4), (2, 4), (2, 7), (3, 5), (3, 6)$ in $(4, 7)$. Če osebe narišemo kot vozlišča grafa, stike pa kot povezave med njimi, dobimo graf iz dveh ločenih delov (t. i. povezani komponenti) na sliki 6, ki v tem primeru predstavljata iskana mehurčka $\{0, 1, 2, 4, 7\}$ in $\{3, 5, 6\}$.

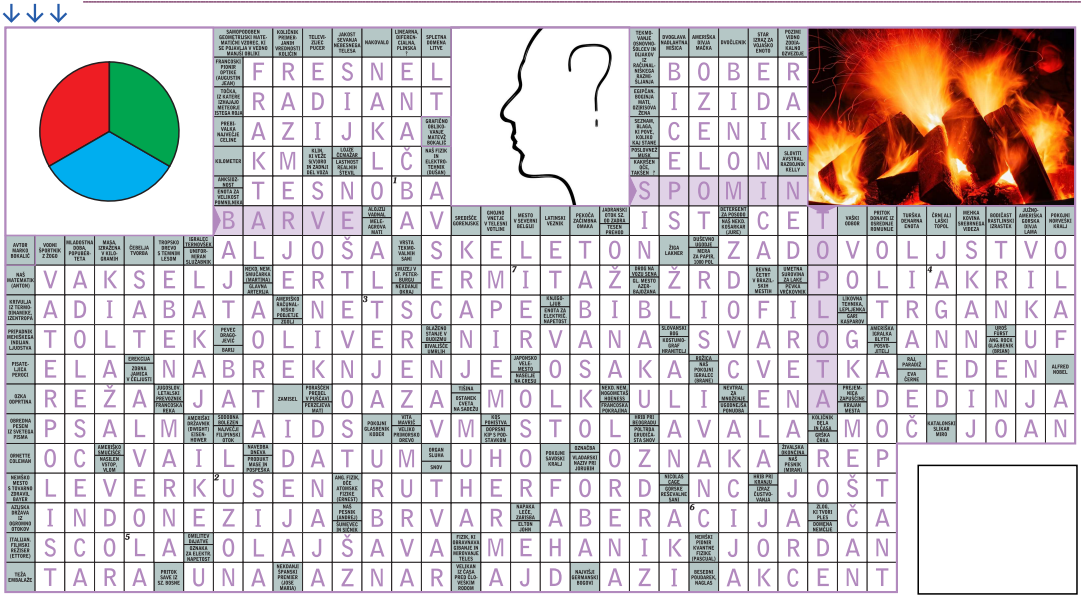
Problem lahko rešimo, če mehurčke obravnavamo kot disjunktne množice. Postopek reševanja prikazuje slika 7.

Na začetku je vsaka oseba v lastnem mehurčku, vsak ugotovljeni stik pa predstavlja možen prenos okužbe, zato je potrebno mehurčka oseb v stiku združiti (razen če sta že v istem mehurčku). V zanki zato obravnavamo zgoraj naštete stike in za vsak stik (x, y) izvedemo klic `UNION(x,y)`. Število oseb v mehurčku vodimo pri korenu pripadajoče disjunktne množice, pri združevanju pa seštejemo vrednosti pri korenih obeh množic. Na sliki 7 so prikazane vsebine polj A (indeksi staršev), R (rangi) in C (velikosti disjunktnih množic). Po zaključku postopka lahko ugotovimo, da sta osebi 0 in 3 korena dveh preostalih mehurčkov, od katerih je večji prvi ($C[0] = 5$).

Literatura

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest in C. Stein, *Introduction to Algorithms*, 3. izdaja, The MIT Press, 2009.

× × ×



REŠITEV NAGRADNE KRIŽANKE

PRESEK 48/6

→ Pravilna rešitev nagradne križanke iz šeste številke Preseka letnika 48 je **Butalci**. Izmed pravilnih rešitev so bili izžrebani Marko Kubale iz Rogaške Slatine, Anže Mihelčič iz Kresnic in Manja Ferme iz Celja, ki bodo razpisane nagrade prejeli po pošti.

× × ×