Algoritmi in podatkovne strukture – 2

Disjunktne množice

Definicija

Imamo množico množic elementov:

$$\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$$

ki so paroma disjunktne.

Razred, ki ga bomo definirali, *ne* predstavlja ena množice, ampak zbirko množic!

Vsaka od množic S_i je *določena s svojim predstavnikom*, ki je lahko načeloma poljuben element iz S_i. (ALI JE LAHKO ISTI ELEMENT PREDSTAVNIK DVEH MNOŽIC?)

Elementi v množicah so iz poljubnega razreda `Object`. Edina lastost, ki jo zahtevamo od njih je, da se nanje lahko sklicujemo (referenca).

Definicija – operacije

Definirajmo naslednje operacije nad množicami:

- `MakeSet(elt)` – naredi novo množico, ki vsebuje samo element `elt`
- `Union(Si, Sj)` – naredi novo množico, stari dve pa prenehata obstajati.
- `Find(elt)` – vrne množico, katere član je `elt`

(KAKO SE ŽE SKLICUJEMO NA MNOŽICE?)

Uporaba

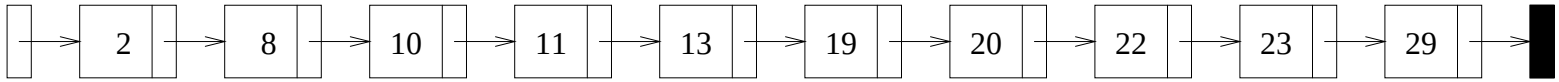
Eden od najpogostejših primerov uporabe paroma disjunktnih množic je pri algoritmih, ki nek problem razdeljujejo dinamično na podprobleme, katere nato rešijo ter na koncu združijo rešitve. Rešitev torej gradijo od spodaj navzgor.

PRIMER: Imamo graf in iščemo v njem povezane komponente. Uporabimo naslednji postopek:

1. vsako vozlišče je samo zase povezana komponenta
2. v množici povezav izberemo poljubno povezavo (u, v). Načeloma ta povezava lahko združuje v povezano komponentno dve povezani komponenti, ki ju predstavimo kot množici S_u in S_v. Zatorej: poiščemo predstavnika vozlišča u in vozlišča v (`Find`) ter, če sta različna, ju združimo (`Union`).
3. to ponavljamo, dokler ne zmanjka povezav.
4. na koncu nam preostale množice predstavljajo povezane komponente grafa

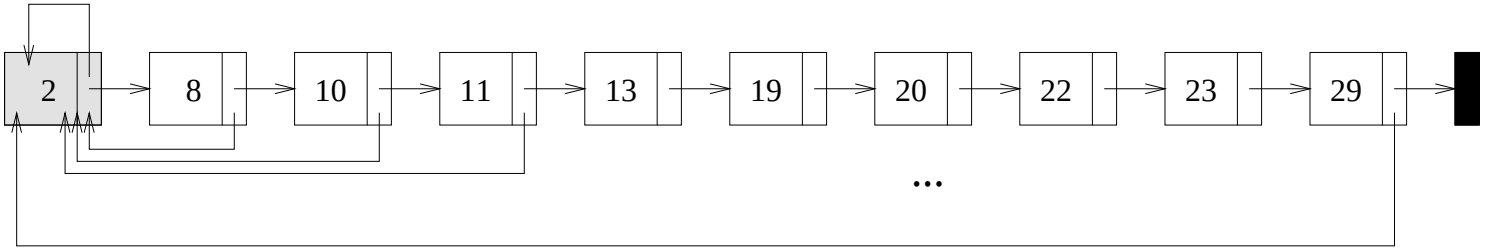
Izvedba s povezanimi seznami

Doslej smo o seznamu govorili kot o strukturi, ki sestoji iz glave in repa, kjer je rep ponovno seznam.



Pri izvedbi smo že govorili tudi o tem, da se v resnici struktura sestoji iz glave in *reference* na rep.

Definirajmo še glavo kot *predstavnika* množice. Da bo vsak element vedel, kdo je predstavnik njegove množice, mu dodajmo referenco na glavo



Povezani seznam

```
public class DisjointSetLL {
  Object        rest;
  DisjointSetLL tail;
  DisjointSetLL representative
  ...
  DisjointSetLL MakeSet(Object elt) { ... }
  DisjointSetLL Find(Object elt) { ... }
  DisjointSetLL Union(DisjointSetLL S1, DisjointSetLL S2) { ... }
}
```

Kakšna je časovna in kakšna prostorska zahtevnost?

Kje so težave

Operacija `MakeSet` je hitra – ni težav.

Operacija `Find` je tudi hitra – ni težav.

Operacija `Union` je počasna, saj moramo popraviti referenco pri vseh elementih enega od seznamov. (SE DÂ TO KAKO IZBOLJŠATI? KAKO? KAKŠNA *hevristika*?)

Analiza težav

Doslej smo seznam uporabljali:

- kot strukturo, v katero smo shranjevali elemente množice

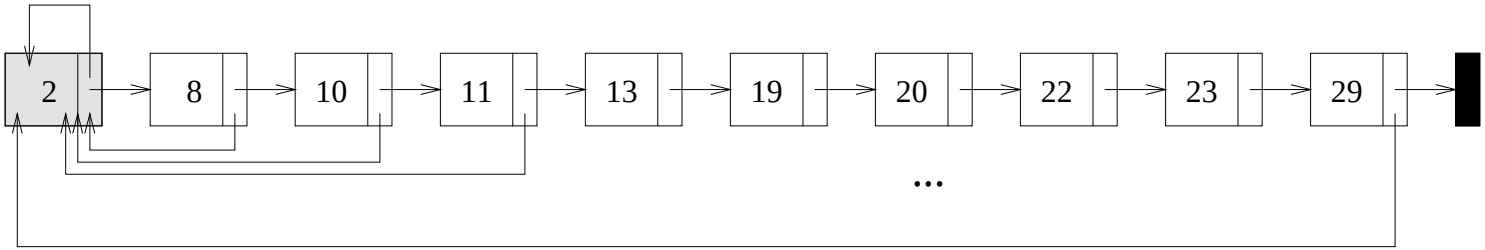
in strukturo smo dodano opremili z referenco na glavo, da smo zadovoljili zahtevi o predstavniku množice.

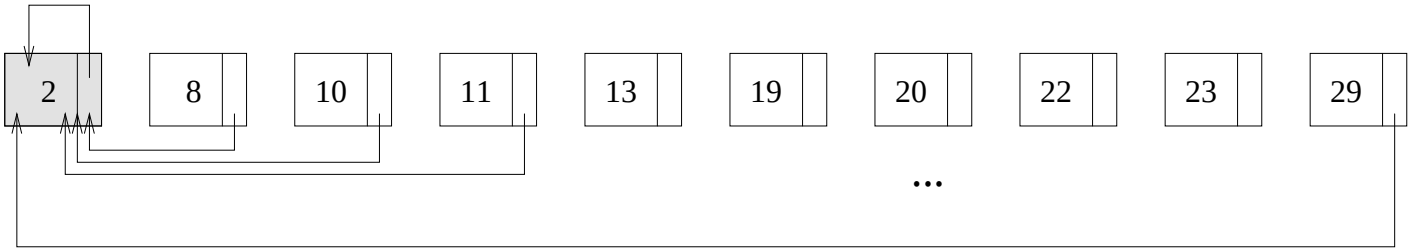
Kaj če uporabili splošnejši pristop in bi uporabili kar *slovar*?

Pa sploh potrebujemo strukturo, v katero shranjujemo elemente množice?

Rešitev težav

V resnici ne potrebujemo strukture izrecno, ampak samo implicitno – če vsak element množice vê, kateri množici pripada, je tudi množica implicitno definirana (operacija `Find`).





Če v množico dodamo nove elemente, moramo samo njim pač povedati, kdo je novi predstavnik množice. Ali smo kaj rešili?

Težave ostajajo

Še vedno je draga operacija `Union`.

V realnem svetu, kako rešujemo problem, znanja? Recimo, da hočemo nekaj vedeti. Kaj naredimo?

1. najprej lahko sami vemo – stvar je rešena
2. če ne pa poznamo nekoga (npr. *Google*), ki ga vprašamo ter potem vemo

Da ta postopek deluje, moramo samo vedeti, ali vemo ali ne vemo.-)

Morda le rešitev

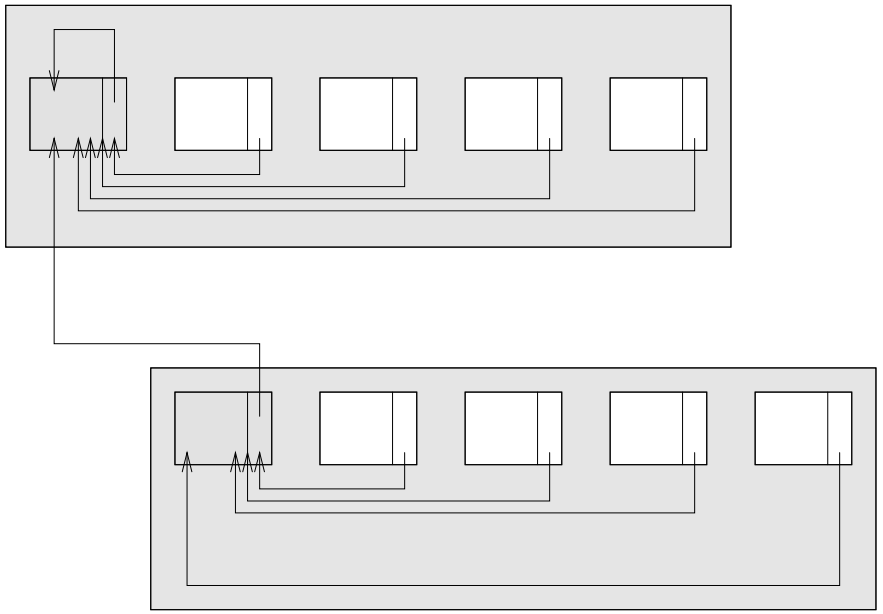
- Vsak element vê, če je predstavnik množice ali ne (vrednost prilastka `representative`).
- Če ni predstavnik množice, potem vpraša tistega, za katerega misli, da je predstavnik množice: ≫*kdo je resnični predstavnik množice*≪ in si ta podatek zapomni.
- Postopek se rekurzivno ponavlja.

Primer na tabli.-)

Analiza

Tvorjenje množice `MakeSet` – preprosto: $O(1)$.

Združevanje množic `Union` – preprosto: $O(1)$, saj vedno združujemo množice, ki jih poznamo preko njihovih predstavnikov.



Iskanje predstavnika `Find` – spet težave ...

Iskanje

Recimo, da je količina napačnih informacij (o predstavniku) mera težav. Kakšna naj bo hevristika pri združevanju množic, da bo ta čim manjša?

Pa sicer so res to težave? Kaj se zgodi, ko za poljuben element najdemo predstavnika množice?

- Vsi, kateri bodo rekurzivno vprašani po predstavniku množice, bodo odslej zanj tudi takoj vedeli.
- Vsi elementi za tem elementom bodo imeli hitrejši dostop do predstavnika.
- Dokler se element ne preseli v drugo množico (združevanje množic), se ta podatek ne bo spreminjal.

Analiza

Izkaže se, da je takšna rešitev optimalna glede na model računanja. (KAJ JE TO MODEL? KAKŠEN PA JE NAŠ MODEL?)

Časovna zahtevnost je $O(\log^* n)$ amortizirano na operacijo.