PRESEK
List za mlade matematike, fizike, astronome in računalnikarje

ISSN 0351-6652
Letnik **27** (1999/2000)
Številka 2
Strani 78–82

Martin Juvan:

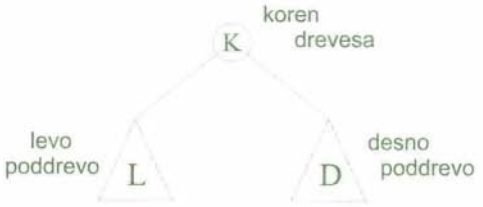
REKONSTRUKCIJA DREVES, 1. del

Ključne besede: računalništvo, programiranje, dvojiška drevesa, pascal.

Elektronska verzija: http://www.presek.si/27/1393-Juvan.pdf

REKONSTRUKCIJA DREVES, 1. DEL

Dvojiška drevesa (angl. *binary trees*) so ena od podatkovnih struktur, s katero se pri programiranju zelo pogosto srečamo. Najlažje jih opišemo rekurzivno: dvojiško drevo je bodisi prazno ali pa je sestavljeno iz odlikovanega vozlišča (pravimo mu *koren*) in dveh poddreves (*levo* in *desno* poddrevo), ki sta zopet dvojiški drevesi. S sliko to običajno prikažemo takole:



Smisel uporabe dvojiških dreves je, da v vozliščih hranimo podatke. Ti so lahko števila, znaki, nizi znakov, kazalci itn. Poleg podatkov potrebujemo še informacijo o tem, kako so vozlišča med seboj povezana v dvojiško drevo (običajno sta to kazalca na korena obeh poddreves).

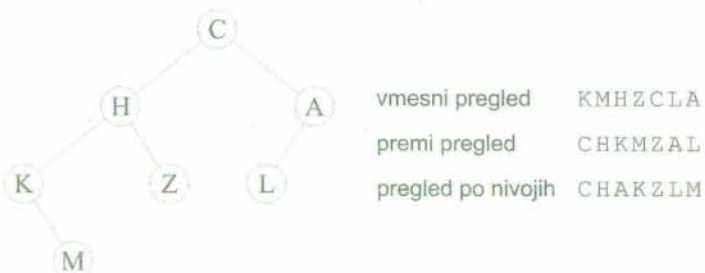
Poglejmo, kakšna je običajna predstavitev dvojiških dreves s kazalci v programskem jeziku pascal. Podatek, ki ga bomo hranili v posameznem vozlišču, bo en znak (pri resnih uporabah so podatki v vozliščih seveda bolj obsežni in zapleteni). V pascalu vozlišče predstavimo s sestavljenim tipom – zapisom (besedica `record`), drevo pa s kazalcem na njegov koren:

```
type
  Drevo = ^Vozlisce;
  Vozlisce = record
    x: char;
    levo,desno: Drevo;
  end;
```

Eno od pogostih opravil z dvojiškimi drevesi (pa tudi z drugimi podatkovnimi strukturami) je sprehod skozi vsa vozlišča drevesa. Smisel sprehoda je, da pri tem pogledamo (*obiščemo*) podatke v vseh vozliščih in v vsakem vozlišču primerno ukrepamo (izpišemo podatek, ga spremenimo, morda izračunamo vsoto ali kako drugo statistično informacijo o podatkih v vozliščih, včasih pa pri pregledu tudi spremenimo samo "obliko" drevesa). Vozlišča drevesa lahko pregledamo v različnih vrstnih redih. Zelo pogosto srečamo naslednje tri preglede (vrstne rede obiska vozlišč):

- *Vmesni pregled.* Pri njem najprej obiščemo levo poddrevo, nato koren drevesa, nazadnje pa še desno poddrevo. Seveda obe poddrevesi spet pregledamo v vmesnem vrstnem redu. Za to vrsto pregleda večkrat uporabljamo kratico LKD.
- *Premi pregled.* Pri njem najprej obiščemo koren drevesa, nato pa levo in nazadnje še desno poddrevo. Obe poddrevesi seveda pregledamo v premem vrstnem redu. Za to vrsto pregleda uporabljamo kratico KLD.
- *Pregled po nivojih.* Vozlišča v drevesu lahko razdelimo glede na oddaljenost od korena. Za vozlišča drevesa, ki so enako oddaljena od korena, pravimo, da sestavljajo *nivo* drevesa. Tako je nivo 0 sestavljen le iz korena drevesa, nivo 1 iz sinov korena itn. Pri pregledu po nivojih najprej pregledamo nivo 0 (torej koren), nato nivo 1 (sinove korena), potem nivo 2 (sinove sinov korena) itn. Vozlišča, ki so na istem nivoju, pregledamo "od leve proti desni", torej v enakem vrstnem redu, kot jih srečamo tudi pri vmesnem in premem pregledu.

Primere pregledov si lahko ogledate na spodnji sliki:



Več o delu z drevesi (in kodo v pascalu) si lahko preberete v poglavju o dinamičnih podatkovnih strukturah v knjigi *N. Wirth: Računalniško programiranje, 1. del.*

V nadaljevanju prispevka nas bo zanimalo, kako iz gornjih pregledov rekonstruirati (torej ponovno zgraditi) drevo. Kot pregled drevesa razumemo zaporedje podatkov iz vozlišč drevesa v takem vrstnem redu, kot do njih pride izbrani pregled drevesa. V našem primeru bomo torej poznali zaporedja znakov, iz njih pa bomo poskušali zgraditi drevo, katerega pregledi bodo ravno začetna zaporedja znakov.

Seveda, če poznamo le en pregled, drevo z njim (razen če je prazno oziroma ima le eno vozlišče) ni enolično določeno. Pri premem pregledu in pregledu po nivojih sicer lahko določimo koren (to je kar prvo obiskano

vozlišče), ali je drugo obiskano vozlišče koren levega ali desnega poddrevesa, pa se ne moremo več odločiti. Iz vmesnega pregleda pa ne moremo razbrati niti korena; katerokoli vozlišče iz pregleda bi lahko bilo koren.

Zato bomo privzeli, da poznamo dva od gornjih treh pregledov drevesa. Pri rekonstrukciji naletimo še na eno težavo. Zgodi se lahko, da se podatki v vozliščih ponavljajo (v skrajnem primeru je lahko v vseh vozliščih isti znak). V takem primeru za pojavitve znakov v prvem pregledu ne moremo enolično določiti pripadajočih pojavitev v drugem pregledu. Na primer, če vsa vozlišča drevesa vsebujejo neki znak x, potem vsi pregledi takega drevesa vrnejo zaporedje x-ov, in sicer ne glede na to, kakšno je v resnici drevo (važno je le, koliko vozlišč ima). Ker bomo poskušali rekonstruirati le drevesa, ki so s pregledi enolično določena, bomo predpostavili, da vsak znak nastopi kvečjemu v enem vozlišču (vozlišča vsebujejo različne znake).

Vmesni in premi pregled: Prvi znak v premem pregledu pripada korenu drevesa. Poiščemo ga v vmesnem pregledu. Znaki, ki v vmesnem pregledu nastopajo pred njim, pripadajo vozliščem levega poddrevesa, znaki, ki so za njim, pa vozliščem desnega poddrevesa. Hkrati še izvemo, koliko vozlišč ima levo in koliko desno poddrevo. Ker tudi pri premem pregledu pregledamo levo poddrevo v celoti pred desnim poddrevesom, lahko premi pregled (brez prvega znaka – korena) razdelimo na del, ki pripada levemu poddrevesu, in del, ki pripada desnemu poddrevesu:

vmesni pregled			premi pregled		
$\underbrace{K\ M\ H\ Z}$	$\underbrace{C}$	$\underbrace{L\ A}$	$\underbrace{C}$	$\underbrace{H\ K\ M\ Z}$	$\underbrace{A\ L}$
levo poddrevo	koren	desno poddrevo	koren	levo poddrevo	desno poddrevo

Obe poddrevesi zgradimo rekurzivno na enak način.

```
function Zgradi(vmesni,premi: string; var napaka: boolean): Drevo;
{ Iz vmesnega in premega pregleda rekonstruira dvojiško drevo. }

function Zgradi_R(zacV,zacP,vel: integer): Drevo;
{ Vmesni pregled se začne pri zacV, premi pa pri zacP. Drevo ima }
{ vel vozlišč. Funkcija uporablja spremenljivke vmesni, premi in }
{ napaka iz funkcije Zgradi.                                      }
  var
    kor: integer; { indeks korena v vmesnem pregledu }
    kaz: Drevo; { pomožni kazalec za novo vozlišče }
```

```
begin
  Zgradi_R := nil;
  if vel>0 then begin { Drevo ni prazno. }
    kor := zacV; { Poiščemo koren v vmesnem pregledu. }
    while (vmesni[kor]<>premi[zacP]) and (kor<zacV+vel) do
      kor := kor+1;
    if kor>=zacV+vel then
      napaka := true { Napačni vhodni podatki. }
    else begin
      new(kaz);
      kaz^.x := premi[zacP];
      kaz^.levo := Zgradi_R(zacV,zacP+1,kor-zacV);
      kaz^.desno := Zgradi_R(kor+1,zacP+kor-zacV+1,zacV+vel-kor-1);
      Zgradi_R := kaz;
    end;
  end;
end; {Zgradi_R}

begin {Zgradi}
  napaka := length(vmesni)<>length(premi);
  if napaka then
    Zgradi := nil
  else
    Zgradi := Zgradi_R(1,1,length(vmesni));
end; {Zgradi}
```

Funkciji `Zgradi` smo poleg obeh pregledov dodali še en parameter: logično spremenljivko `napaka`. Prek nje klicatelju funkcije sporočimo, ali je bila rekonstrukcija drevesa uspešna. Pri gradnji drevesa zaznamo dve vrsti napak. Že takoj na začetku sta pregleda lahko različne dolžine. Tedaj drevesa sploh ne poskušamo zgraditi, saj je s pregledoma očitno nekaj narobe. Če sta niza z opisoma pregledov enako dolga, pri rekurzivnih klicih funkcije `Zgradi_R` poskrbimo, da ostaneta enako dolga tudi pri nadaljnjih rekurzivnih klicih. Drugo mesto, kjer tudi lahko odkrijemo napako, pa je pri iskanju položaja korena v vmesnem pregledu. Lahko se zgodi, da korena ne najdemo. Tedaj postavimo indikator `napaka` na `true` in vrnemo prazno drevo (vrednost `nil`) kot rezultat tekočega klica funkcije `Zgradi_R`. Za celotno funkcijo `Zgradi` to pomeni, da kljub napačnim podatkom vseeno zgradi del drevesa, za katerega domneva, da je z danima pregledoma opisan pravilno. Koristilo vam bo, če funkcijo preskusite še z nekaj napačnimi podatki in si pozorno ogledate zgrajena drevesa. Če se klic funkcije `Zgradi` konča brez napake, potem sta vmesni in premi pregled zgrajenega drevesa vedno enaka vhodnima pregledoma, ni pa nujno, da

je to edino drevo s to lastnostjo. Funkcija namreč ne preverja enoličnosti rešitve (vedno vzame, da prva pojavitev znaka v premem pregledu ustreza njegovi prvi pojavitvi v vmesnem pregledu; pri predpostavki o različnih znakih v posameznih vozliščih je to povsem smiselno obnašanje).

Pokomentirajmo še časovno in prostorsko zahtevnost funkcije `Zgradi`. Naj ima iskano drevo n vozlišč. Število klicev funkcije `Zgradi_R` je enako $2n + 1$ (n klicev, v katerih naredimo vozlišča, in $n + 1$ klicev na praznih poddrevesih); če pride do napake, je klicev lahko tudi manj. Pri vsakem vozlišču poiščemo koren, torej pregledamo začetni del pripadajočega vmesnega pregleda. Število znakov, ki jih pogledamo, je omejeno z n. (Lahko se zgodi, da je koren vedno ravno zadnji pregledani znak; premislite, kakšno je potem drevo.) Celotno število operacij je torej omejeno s kvadratno funkcijo števila vozlišč, za nekatera drevesa pa je lahko tudi bistveno manjše. Ocenimo še količino pomnilnika, ki ga potrebuje funkcija. Zgrajeno drevo je sestavljeno iz n zapisov tipa `Vozlisce`, pri gradnji pa uporabljamo še pomožne spremenljivke. To so parametri in lokalne spremenljivke v funkciji `Zgradi_R`, nekaj pomnilnika pa zasedejo tudi nam "nevidne" spremenljivke, ki jih vpelje prevajalnik za "knjigovodstvo" o rekurzivnih klicih. Ker se po koncu klica funkcije pomnilnik, ki ga zasedajo parametri, lokalne spremenljivke in "spremenljivke za knjigovodstvo", sprosti, je največja hkrati zasedena količina pomnilnika sorazmerna z globino rekurzije. Največja globina rekurzije pa je odvisna od globine (števila neprazih nivojev) drevesa, ki ga gradimo. Če je drevo "izrojeno" (vsako vozlišče ima le enega sina), je globina največja (enaka je n), najmanjša pa je pri "izravnanem" drevesu (vsi nivoji, razen morda zadnjega, so v celoti zapolnjeni). Tedaj je enaka $\lfloor \log_2 n \rfloor + 1$ (zadnja trditev zahteva krajši razmislek in skico ali dve).

Pri koncu prvega dela prispevka smo. V naslednji številki Preseka si bomo ogledali, kaj lahko povemo o dvojiškem drevesu, če poleg vmesnega ali premega pregleda poznamo tudi njegov pregled po nivojih.

Martin Juvan