Binarno iskalno drevo (BST)

1. Splošna predstavitev problema

Drevesa so podatkovne strukture, ki nastopajo v aplikacijah, kjer je potrebno pogosto poiskati nek podatek, kar pomeni, da mora biti iskanje po tej podatkovni strukturi izjemno hitro. Ponavadi so del kompleksnih geometrijskih algoritmov z logaritemsko časovno zahtevnostjo. Samo ime drevo izhaja iz teorije grafov, kjer je drevo acikličen povezan graf. Najpogosteje uporabljena drevesa v aplikacijah so dvojiška in štiriška drevesa. V prvem primeru ima lahko vsako vozlišče drevesa dva potomca, v drugem primeru pa štiri. Slednja se najpogosteje uporabljajo v algoritmih računalniške grafike, saj z njimi implementiramo delitev scene v manjše dele, kar je bistveno za določanje vidljivosti in pohitritev izrisa, ter hitrejšo orientacijo po delu scene v katerem se nahajamo, so torej pomemben del 3D računalniških iger. Dvojiška drevesa se uporabljajo v manj zahtevnih aplikacijah, predvsem zaradi njihove preprostosti in hitrega iskanja po njihovi strukturi. Binarno iskalno drevo je poseben tip dvojiškega drevesa, za katerega veljajo naslednje lastnosti:

- vsako vozlišče ima vrednost,
- vozlišča so urejena glede na to vrednost,
- vsa vozlišča levega poddrevesa vsebujejo vrednosti manjše, od vrednosti trenutnega vozlišča,
- vsa vozlišča desnega poddrevesa vsebujejo vrednosti večje, od vrednosti trenutnega vozlišča.

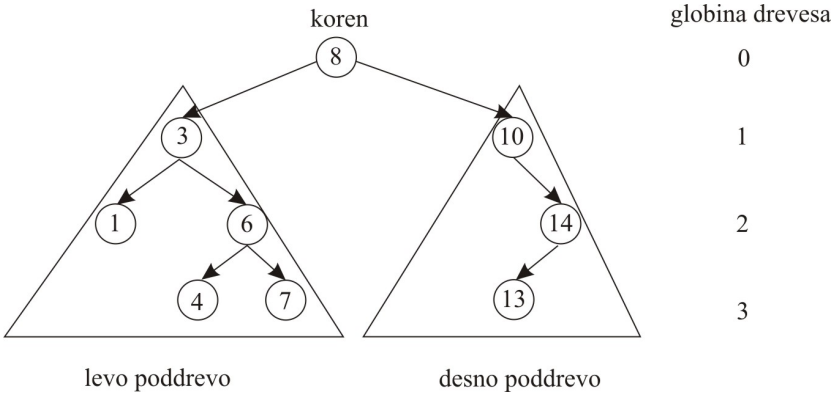
Glede na te lastnosti, ima vsako vozlišče naslednje podatke: *ključ*, ta določa vrednost vozlišča in tri kazalce. Kazalca na levega in desnega potomca (*leviSin*, *desniSin*), kažeta na vozlišči z manjšo in večjo vrednostjo. V kolikor tako vozlišče ne obstaja sta kazalca enaka *NIL*. Tretji kazalec je kazalec na očeta, ki kaže na predhodno vozlišče.

Zapis elementa v C++ bi bil v primeru binarnega iskalnega drevesa naslednji (tudi v tem primeru predpostavimo, da je ključ celo število):

```
struct vozlisce {
      int key;
      vozlisce *oce;
      vozlisce *leviSin, *desniSin;
};
```

Prvo vozlišče v drevesu imenujemo *koren*. Zanj velja, da je kazalec na očeta enak *NIL* in to je edino tovrstno vozlišče v drevesu. Vozlišče pri katerem sta kazalca *leviSin* in *desniSin* enaka *NIL* imenujemo list drevesa. Listov je lahko v drevesu seveda veliko. Pomemben pojem, s katerim se srečujemo pri drevesih je tudi globina drevesa. Globina drevesa je določena s številom vozlišč med korenom in listi, pri čemer pri določanju globine upoštevamo tudi liste. Koren drevesa se nahaja vedno na globini 0. Globina drevesa je pomembna za določanje časovne zahtevnosti operacij nad drevesom.

Primer binarnega iskalnega drevesa lahko vidimo na sliki 1.



Slika 1: Primer binarnega iskalnega drevesa

Da smo dobili drevo na sliki 1, smo v drevo vstavili vrednosti 8, 10, 3, 14, 6, 1, 14, 4, 13 in 7.

Na binarnem iskalnem drevesu izvajamo naslednje operacije: *VSTAVLJANJE, ISKANJE*, *MINIMUM*, *MAKSIMUM*, *PREDHODNIK*, *NASLEDNIK* in *BRISANJE*. Pri tem je potrebno omeniti, da osnovno binarno iskalno drevo omogoča zgolj hranjenje različnih vrednosti, kar ni vedno tudi dovolj.

Na binarnih iskalnih drevesih lahko izvajamo precejšnje število operacij. Najpogostejše med njimi so:

- vstavljanje vrednosti v drevo
- iskanje vozlišča z določeno vrednostjo,
- določanje minimuma in maksimuma drevesa,
- iskanje predhodnika in naslednika danega vozlišča,
- brisanje vozlišča iz drevesa.

Vse omenjene operacije se izvedejo v času O(h), pri čemer je h globina drevesa. Brisanje vozlišča iz drevesa mora biti izvedeno tako, da ohranja urejenost drevesa.

2. Pomoč pri implementaciji

Na začetku bomo osredotočili zgolj na dve funkciji, to je vpis nove vrednosti v drevo in branje teh vrednosti iz drevesa tako, da jih dobimo izpisane v urejenem vrsten redu. Ogledali si bomo postopek za implementacijo najpogostejše operacije, to je iskanja v drevesu, nato pa še iskanje minimuma in maksimuma drevesa, predhodnika in naslednika danega vozlišča. Zaključili bomo z brisanjem vozlišča iz drevesa.

Vpis nove vrednosti

Predpostavimo, da imamo znan kazalec na koren drevesa, ki ga označimo s črko *T*. V drevo želimo vstaviti nov podatek, ki ga hranimo v spremenljivki *k*. Psevdokod funkcije *VSTAVI* je prikazan v izpisu 1.

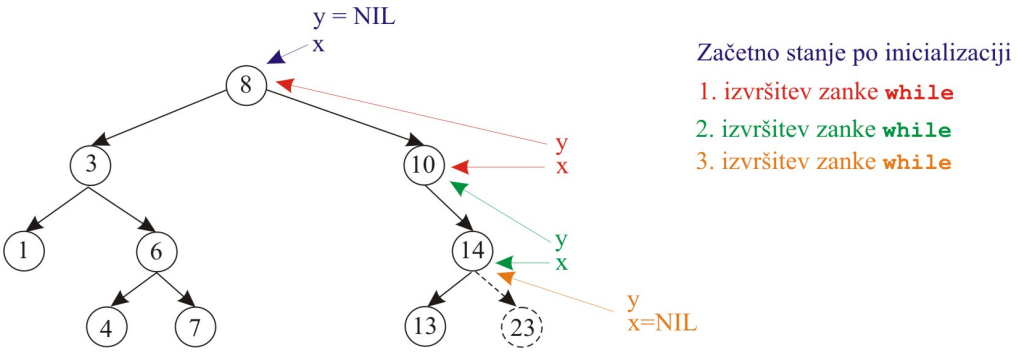
```
function VSTAVI(T, k)
 begin
      y := NIL;
      x := T;

      while x<>NIL do
            y := x;
            if k < x.key then
              x := x.leviSin;
            else
              if k > x.key then
               x := x.desniSin;
              else return Napaka;
      end
      z = USTVARI_VOZLISCE();
      z.key = k;
      z.oce := y;

      if y = NIL then
        T := z;
      else
        if z.key < y.key then
          y.leviSin := z;
        else
          y.desniSin := z;
 end
```

Izpis 1: Psevdokod funkcije *VSTAVI*

Na sliki 2 imamo prikazan primer vpisa števila 23 v drevo s slike 1. Najprej je treba poiskati mesto vpisa. Z iskanjem začnemo v korenu drevesa. Najprej ustvarimo dva pomožna kazalca *x* in *y*, pri tem pa postavimo kazalec *x* tako, da kaže na koren drevesa, to je vozlišče z vrednostjo 8, kazalec *y* pa postavimo na *NIL*. Nato se s pomočjo zanke **`while`** pomikamo po drevesu proti listom in poiščemo mesto za vnos. Ker je 23 več kot 8, se pomaknemo desno, kazalec *x* pa sedaj kaže na vozlišče 10. Ob ponovni iteraciji zanke tudi kazalec *y* kaže na vozlišče z vrednostjo 10, *x* pa prestavimo na vozlišče z vrednostjo 14, ker je število 23 večje od vrednosti 10.



Slika 2: Vstavljanje novega števila v binarno iskalno drevo

V naslednji iteraciji zanke **`while`** postavimo kazalec *y* na vozlišče 14, kazalec *x*, pa zopet premaknemo na desnega sina, saj je 23 večji tudi od 14. Ker vozlišče z vrednostjo 14 desnega sina nima, dobi *x* vrednost *NIL* zanka **`while`** pa se zaključi. Po zaključku zanke **`while`**, je tako kazalec *x* enak *NIL*, kazalec *y* pa kaže na vozlišče z vrednostjo 14, to pa pomeni, da drevo ni prazno. Zato preverimo ali je vrednost

novega vozlišča večja ali manjša od števila 14, da ustrezno postavimo kazalec levi ali desni sin. V našem primeru pomeni, da kazalec *desniSin* kaže na vozlišče *z*.

Izpis drevesa

Binarno iskalno drevo lahko uporabimo tudi za urejanje vrednosti in sicer to dosežemo s preprostim obiskom vozlišč drevesa v pravilnem zaporedju. Tako urejanje sicer ni najbolj učinkovito, a bo dovolj za testiranje pravilnosti vstavljanja v iskalno drevo. Urejen izpis iz drevesa dobimo z algoritmom, prikazanim v izpisu 2.

```
function UREJEN_IZPIS(x)
 begin
      if x <> NIL then
       begin
        UREJEN_IZPIS(x.leviSin);
        Izpiši x.key;
        UREJEN_IZPIS(x.desniSin);
       end
 end
```

Izpis 2: Psevdokod funkcije *UREJEN_IZPIS*

Izpis zaženemo s klicem funkcije *UREJEN_IZPIS*(*T*), pri čemer je *T* kazalec na koren drevesa.

Drugi način za testiranje pravilnosti delovanja aplikacije je izpis vseh povezav, ki je prikazan v izpisu 3.

```
function IZPIS_POVEZAV(x)
 begin
   if x.leviSin<>NIL then
    begin
     Print (x.key -> x.leviSin.key)
     IZPIS_POVEZAV(x.leviSin);
    end

   if x.desniSin<>NIL then
    begin
     Print (x.key -> x.desniSin.key)
     IZPIS_POVEZAV(x.desniSin);
    end
end
```

Izpis 3: Psevdokod funkcije *IZPIS_POVEZAV*

Iskanje vozlišča z določeno vrednostjo

Dan imamo kazalec na koren drevesa *T* in ključ *k*. Poiskati želimo vozlišče, ki vsebuje vrednost enako vrednosti *k*. Iskanja se lahko lotimo na dva načina: rekurzivnim in iterativnim. Zaradi lažjega razumevanja rekurzije si bomo ogledali oba postopka. Začeli bomo z iterativnim postopkom. V iterativni varianti bomo preiskovanje drevesa

opravili s pomočjo zanke **while**, kjer se bomo premikali levo ali desno po drevesu, dokler ne bomo našli vrednosti, ki jo iščemo, ali pa bomo prišli do listov drevesa in pot naprej ne bo več mogoča. Psevdokod iterativne variante je prikazan v izpisu 4.
Pri rekurzivni varianti je potrebno zanko **while** nadomestiti z rekurzivnim klicem procedure. Da je lahko rekurzivni način uspešen, je potrebno razmisliti o zaključnem pogoju. Spet se osredotočimo na stavek **while** pri iterativni metodi. Vidimo, da se po drevesu premikamo, dokler je kazalec na trenutno vozlišče različen od *NIL*, ključa pa še nismo našli. Od tod lahko zaključimo, da se bo naša rekurzija zaključila, ko bo kazalec, s katerim se pomikamo po drevesu enak *NIL*.

```
function POISCI(x, k)
 begin
      while x<>NIL and k <> x.key do
        if k < x.key then
              x := x.leviSin;
        else
              x := x.desniSin;
      return x;
end
```

Izpis 4: Psevdokod iterativne variante funkcije *POISCI*

Rekurzivna varianta funkcije *POISCI* je podana v izpisu 5.

```
function POISCI(x, k)
 begin
      if x=NIL or x.key = k then
            return x;
      else
        if k < x.key then
              POISCI(x.leviSin, k);
        else
              POISCI(x.desniSin, k);
end
```

Izpis 5: Psevdokod rekurzivne variante funkcije *POISCI*

V obeh primerih začnemo iskanje vrednosti v korenu drevesa. V obeh primerih imamo tudi enake vhodne parametre, to je kazalec na vozlišče drevesa in vrednost, ki jo želimo najti v drevesu.

Iskanje minimuma in maksimuma v drevesu

Za razliko od prejšnje točke se bomo tukaj osredotočili le na iterativne variante algoritmov. Najprej poglejmo algoritem za iskanje minimuma v drevesu. Ta je podan v izpisu 6.

```
function MINIMUM(x)
 begin
      while x.leviSin<>NIL do
            x := x.leviSin;
      return x;
end
```

Izpis 6: Psevdokod funkcije *MINIMUM*

Tudi v primeru iskanja minimuma z iskanjem začnemo v korenu drevesa, tako da bo prvi klic funkcije enak *MINIMUM*(*T*), kjer je *T* kazalec na koren drevesa.
Funkcija za iskanje maksimuma v drevesu je zelo podobna. Njen psevdokod je podan v izpisu 7.

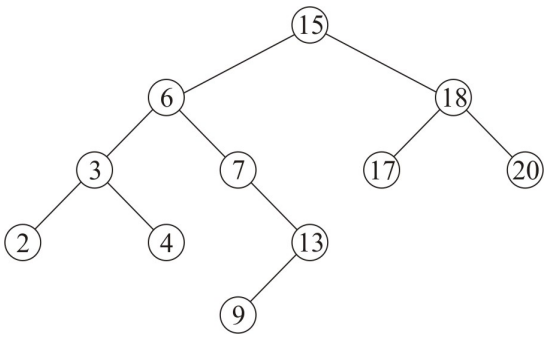
```
function MAKSIMUM(x)
 begin
      while x.desniSin<>NIL do
            x := x.desniSin;
      return x;
end
```

Izpis 7: Psevdokod funkcije *MAKSIMUM*

Tudi v tem primeru iskanje maksimuma zaženemo s klicem *MAKSIMUM*(*T*).

Iskanje predhodnika in naslednika danega vozlišča

Denimo, da imamo binarno iskalno drevo zgrajeno na neurejenem zaporedju števil. V tem primeru se zgodi, da je treba v drevesu poiskati naslednika ali predhodnika danega vozlišča. Naslednik danega vozlišča je vozlišče z najmanjšo vrednostjo, ki je večja od vrednosti v danem vozlišču. Če pogledamo drevo na sliki 3, potem je naslednik vozlišča 15, vozlišče 17, saj je to vozlišče z najmanjšo vrednostjo, ki je večja od 15. V tem primeru je tudi iskanje naslednika enostavno. Poiščemo namreč zgolj minimum desnega poddrevesa vozlišča 15 in smo nalogo opravili.



Slika 3: Primer binarnega iskalnega drevesa

Situacija pa je povsem drugačna, če želimo poiskati naslednika vozlišča z vrednostjo 4, na sliki 3. Iz slike vidimo, da je njegov naslednik vozlišče z vrednostjo 6, ki je njegov dedek. V tem primeru se iskanje nekoliko zaplete. Način iskanja predhodnika v takem primeru lahko vidimo v prevdokodu funkcije *NASLEDNIK* v izpisu 8.

```
function NASLEDNIK(x)
 begin
      if x.desniSin<>NIL then
            return MINIMUM(x.desniSin);

      y = x.oce;
      while y <> NIL and x = y.desniSin do
        begin
            x := y;
            y := y.oce;
        end

      return y;
end
```

Izpis 8: Psevdokod funkcije *NASLEDNIK*

V tem primeru namreč uvedemo nov pomožni, kazalec *y*, ki ga postavimo na očeta vozlišča *x*, katerega naslednika iščemo. Če je vozlišče *x* desni sin tako postavljenega pomožnega vozlišča *y*, se po drevesu pomaknemo navzgor, kazalca *x* in *y* pa vsakokrat ustrezno popravimo. To ponavljamo, dokler je ta pogoj izpolnjen. Ko pa *x* ni več desni sin vozlišča *y*, funkcija *NASLEDNIK* kazalec *y* vrne kot rešitev našega iskanja.

Predhodnik danega vozlišča, pa je vozlišče, katerega vrednost je maksimum vseh manjših vrednosti. Tako je predhodnik vozlišča 15 vozlišče 13. Dobimo ga kot maksimum levega poddrevesa vozlišča z vrednostjo 15.

Podobno pot pri iskanju naslednika pa se zaplete pri iskanju predhodnika vozlišča, ki levega poddrevesa nima, na primer vozlišče z vrednostjo 4, katerega predhodnik je vozlišče z vrednostjo 3. V tem primeru se je zopet potrebno premakniti na očeta danega vozlišča, za kar uvedemo pomožni kazalec *y*, ki ga tako pot prej postavimo, da kaže na očeta od *x*. Nato gremo v zanko **`while`**, kjer preverjamo ali je vozlišče *y* različno od *NIL* in ali je vozlišče *x* levi sin pomožnega vozlišča *y*. Dokler to drži se pomikamo po drevesu navzgor. Ko to več ne gre tudi funkcija *PREDHODNIK* vrne pomožni kazalec *y* kot rešitev problema. Ker sta torej funkciji zelo podobni, psevdokoda za funkcijo *PREDHODNIK* ne bomo napisali, ampak to prepuščamo bralcu samemu, smo pa v izpisu 9 nakazali potek te funkcije.

```
function PREDHODNIK(x)
 begin
      if x.leviSin<>NIL then
            return MAKSIMUM(x.leviSin);

      y = x.oce;

                           ...

      return y;
end
```

Izpis 9: Psevdokod funkcije *PREDHODNIK*

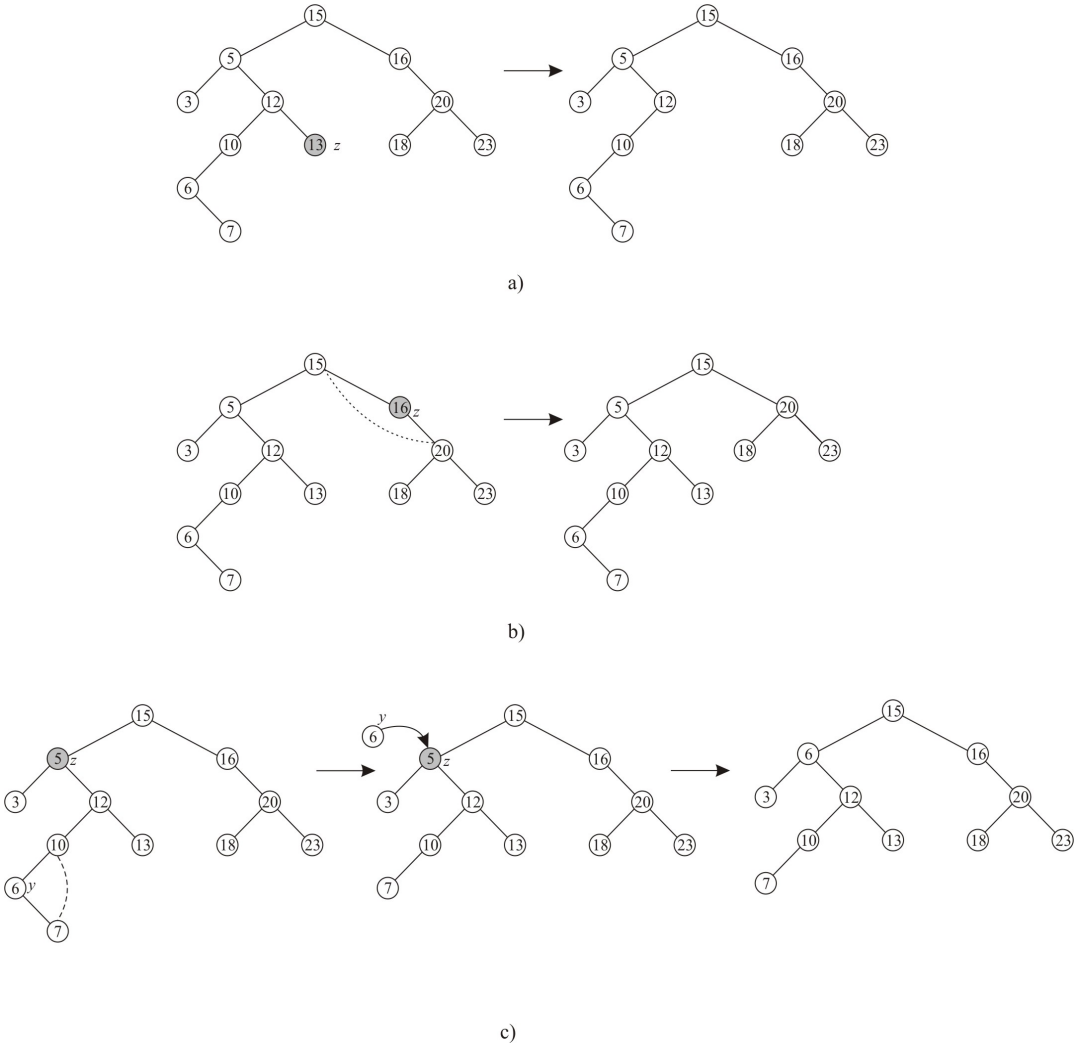
Tako funkciji *NASLEDNIK*, kot tudi *PREDHODNIK* ne delujeta, v kolikor nimamo v drevesu samih različnih vrednosti, oziroma smo naše drevo razširili s seznamom enakih vrednosti.

Brisanje vozlišča iz drevesa

Brisanje vozlišča iz drevesa je najtežavnejša funkcija, saj moramo pri tem ohranjati strukturo drevesa. Akcije ob brisanju so odvisne od števila potomcev, ki jih ima dano vozlišče. Denimo, da je *z* kazalec na vozlišče, ki ga želimo brisati. Če je *z* brez otrok, je njegovo brisanje enostavno, saj v staršu kazalec, ki kaže na vozlišče *z* postavimo enostavno na *NIL* (glej slika 4a).

Enostavna situacija je tudi, ko ima vozlišče *z* enega samega potomca. Tedaj kazalec v očetu vozlišča *z*, ki kaže na vozlišče *z,* preusmerimo na edinega sina vozlišča *z* in brisanje smo opravili.

V kolikor ima vozlišče *z* dva potomca, je potrebno poiskati njegovega naslednika, ki ga poimenujmo *y*. Lastnost vozlišča *y* je, da ima največ enega potomca. Vozlišče *y* iz drevesa izrežemo s prevezovanjem kazalcev med očetom in sinom vozlišča *y*, nato pa vrednost v vozlišču *z* zamenjamo z vrednostjo vozlišča y (glej sliko 4c).



Slika 4: Brisanje vozlišča
a) brez potomcev
b) z enim potomcem
c) dvema potomcema

Psevdokod funkcije *BRISI* je prikazan v izpisu 10. Funkcija *BRISI* sprejme dva parametra, kazalec na koren drevesa, *T* in kazalec na vozlišče, ki ga želimo izbrisati, *z*.

```
procedure BRISI(T, z)
 begin
    if z.leviSin = NIL or z.desniSin = NIL then
        y := z;
    else
        y := NASLEDNIK(z);

    if y.leviSin <> NIL then
        x := y.leviSin;
    else
        x := y.desniSin;

    if x <> NIL then
        x.oce := y.oce;
    if y.oce = NIL then
        T := x;
    else
        if y = y.oce.leviSin then
            y.oce.leviSin := x;
        else
            y.oce.desniSin := x;

    if y <> z then
        z.key := y.key;

    delete y;
 end
```

Izpis 10: Psevdokod funkcije *BRISI*

Vstavljanje enakih vrednosti

Povedali smo že, izpis 1 pa je to potrdil, da v binarno iskalno drevo brez dodatnih sprememb ne moremo vstaviti enake vrednosti tistim, ki je že shranjene v drevesu. Zaradi tega bomo sedaj pogledali, kaj bi bilo potrebno narediti, da bi nam to uspelo. Ker lahko po definiciji iskalnega drevesa vstavljamo v vozlišča le večje ali manjše vrednosti, je potrebno vozlišče drevesa razširiti tako, da bomo v vozlišče lahko shranili tudi več kot eno samo vrednost, če bo to potrebno. Na vozlišče bomo namreč vezali seznam enakih vrednosti. To pomeni, da bomo definirali element enojno povezanega seznama z vrednostjo in kazalcem na naslednje število. Glavna razlika med enojno povezanim seznamom in dvojno povezanim seznamom je ta, da element enojno povezanega seznam ne vsebuje kazalca na prejšnji element.

Vozlišče bomo definirali na naslednji način:

```
struct element {
      int key;
      element *naslednji;
};
```

```
struct vozlisce {
      int key;
      vozlisce *oce;
      vozlisce *leviSin, *desniSin;
      element *seznam;
};
```

Posledično bomo spremenili tudi način vstavljanja v drevo in sicer bomo v izpisu 1 kodo za vstavljanje spremenili tako, da bomo vrednosti vstavili v seznam. Ob začetnem vstavljanju je potrebno najprej ustvariti vozlišče in tudi seznam s prvim elementom. Ob nadaljnjih vstavljanjih vrednosti dodajamo v glavo seznama s klicem funkcije *VSTAVI_V_SEZNAM*(*x.seznam*, *z.key*), ki pa ni nič drugega kot vstavljanje v glavo enojno povezanega seznama.

Nekoliko pa bo potrebno popraviti tudi proceduro za izpis podatkov v izpisu 2 in sicer je potrebno vrstico

```
      Izpiši x.key;
```

nadomestiti z naslednjo sekvenco ukazov:

```
      IZPISI_SEZNAM(x.seznam);
```

kar pomeni, da bomo izpisali celoten seznam.

Tudi pred brisanjem je potrebno preveriti ali seznam v vozlišču še vsebuje več kot eno vrednost. V tem primeru je potrebno izbrisati eno vrednost iz seznama. V nasprotnem primeru je potrebno seznam izbrisati in klicati funkcijo BRISI iz izpisa 10.