UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
Matematika – praktična matematika (VSŠ)

Simon Butalič

LOMLJENA DREVESA

Diplomska naloga

Ljubljana, 2006

ZAHVALA

Z diplomsko nalogo se zaključuje moje najlepše obdobje študijskega življenja. Zato bi se ob tej priložnosti rad zahvalil svojim staršem za finančno in ostalo pomoč skozi ves čas študija. Hvala moji Bredi in vsem prijateljem za moralno podporo. Posebej pa se moram zahvaliti vsem sošolcem, ki so mi kakorkoli pomagali skozi lepe naporne študijske dni. Zahvaljujem se mentorju mag. Matiju Lokarju, ki si je ob svojem prepolnem urniku vedno vzel dovolj časa za pomoč pri nastanku moje diplomske naloge.

Diplomsko nalogo posvečam svojim staršem.

KAZALO

PROGRAM DIPLOMSKEGA DELA

V diplomski nalogi predstavite lomljena drevesa.

Osnovna literatura:

"Self-adjusting Binary Search Trees", Sleator and Tarjan, JACM Volume 32, No. 3, July 1985, pp. 652-686.

H. Biermann, Splay Trees, http://www.cs.nyu.edu/algvis/java/SplayTree.html.

I. Kononenko, Načrtovanje podatkovnih struktur in algoritmov, Založba FE in FRI, 1996, Ljubljana.

mentor

mag. Matija Lokar

POVZETEK

Lomljena drevesa so dvojiška iskalna drevesa, ki se lomijo (preoblikujejo) ob vsakem posegu v drevo: vstavljanju, brisanju in celo pri iskanju elementa.
Običajni problem z iskalnimi drevesi je njihovo izrojevanje – proces, kjer pridemo do drevesa, kjer se pri enem ali več poddrevesih višina levega poddrevesa precej razlikuje od višine desnega poddrevesa.
Z uporabo lomljenih dreves ob razmeroma majhnem povečanju časovne zahtevnosti osnovnih operacij, kot so vstavljanje, iskanje in brisanje, dosežemo, da se drevesa ne izrodijo. Za posamezne operacije in za drevesa z manjšim številom elementov to sicer ne velja. Zagotovljeno pa je, da bo višina drevesa potem, ko bomo izvedli večje število operacij, ostala bližje log(n) kot n, seveda, če bo zaporedje operacij ustrezalo nekemu povprečnemu vzorcu.

Math. Subj. Class (2000): 68P05, 68P10.

Ključne besede: uravnoteženo iskalno drevo, uravnoteževanje, časovna zahtevnost, lomljeno drevo.
Key words: balanced search tree, balancing, time complexity, splay tree.

1 ABSTRAKTNA PODATKOVNA STRUKTURA SLOVAR

Slovar je primer podatkovne strukture, ki omogoča tri osnovne operacije: vstavljanje, brisanje in iskanje elementa. Vse te operacije želimo izvajati kar se da hitro. Poleg izraza slovar se uporablja tudi izraz tabela simbolov.
Obstaja veliko problemov, ki za podatkovno strukturo potrebujejo podatkovni tip slovar. Oglejmo si nekaj takih problemov:

- Organizacija podatkovne baze zahteva hitro iskanje, brisanje in dodajanje elementov. Pri tem so elementi urejeni po enem ali več ključev. Slovar je vsekakor primerna podatkovna struktura za hranjenje ključev.
- Pri obdelavi besedil je potrebno preverjati, če je beseda pravilno črkovana. Besedo je torej potrebno hitro najti, uporabnik pa mora imeti tudi možnost dodati novo besedo v slovar.
- Če želimo zgostiti besedilo, je potrebno izračunati frekvenco črk in/ali besed v besedilu. Pri izračunu frekvenc besed v besedilu je potrebno besede iskati in dodajati nove.
- Pri pregledovanju grafa si moramo zapomniti vozlišča, ki smo jih je že preiskali, da ne bi po nepotrebnem ponavljali iskanja. Za hranjenje potrebujemo slovar z dodajanjem in iskanjem elementov. Nekateri algoritmi dovoljujejo, da se pod določenim pogoji dostop do nekega vozlišča ponovno dovoli. V tem primeru potrebujemo tudi brisanje vozlišč iz slovarja.

Pri implementaciji slovarja si moramo izbrati tako predstavitev, ki bo omogočala tako učinkovito spreminjanje slovarja, kot tudi iskanje elementov v slovarju. Ker moramo v slovarju elemente iskati, je potrebno med elementi slovarja vpeljati urejenost. Urejenost med elementi omogoča hitro iskanje elementa v slovarju.

Zaradi enostavnosti bomo predpostavili, da so elementi, ki jih hranimo v slovarju, kar cela števila. Zato bomo elemente lahko primerjali kar z relacijskimi operatorji in ne s posebnimi metodami.

Abstraktni podatkovni tip SLOVAR torej omogoča naslednje operacije:

- PRIPRAVI(D) naredi prazen slovar D.
- VSTAVI(x, D) vstavi element *x* v slovar D.
- BRIŠI(x, D) zbriše element *x* iz slovarja D.
- IŠČI(x, D) preveri, če je element *x* v slovarju D.

Za implementacijo slovarja imamo na voljo več možnosti – od tabel do različnih oblik dreves. Ker želimo učinkovito iskanje, običajno uporabimo eno od oblik iskalnih dvojiških dreves. Iskalno dvojiško drevo je drevo, za katerega velja, da so vsi elementi v levem poddrevesu manjši od korena in v desnem poddrevesu večji od korena. Tudi levo in desno poddrevo sta iskalni dvojiški drevesi.

Iskalna drevesa, implementirana s kazalci (referencami), omogočajo poleg učinkovitega iskanja, dodajanja in brisanja elementov tudi učinkovito iskanje minimalnega in maksimalnega elementa ter iskanje predhodnika in naslednika danega elementa.
Težava pri uporabi iskalnih dvojiških dreves pa je izrojevanje drevesa. To je proces, kjer nas zaporedje izvajanj operacij pripelje do drevesa, kjer se pri enem ali več poddrevesih višina levega poddrevesa precej razlikuje od višine desnega poddrevesa. Do izrojevanja iskalnega dvojiškega drevesa lahko na primer pride pri zaporednem vstavljanju urejenega zaporedja elementov. V skrajnem primeru lahko dobimo povsem izrojeno drevo, kjer ima vsako vozlišče le levega ali desnega sina. Dobimo verižni seznam. Operacije nad izrojenim drevesom so časovno neučinkovite, saj so sorazmerne številu elementov v drevesu, torej $O(n)$, v nasprotju z $O(\log(n))$ zahtevnostjo pri iskalnih drevesih, kjer se višina levih poddreves ne razlikuje veliko od višine desnih poddreves (t.i. uravnoteženih ali skoraj uravnoteženih drevesih). To pomanjkljivost iskalnih dvojiških dreves lahko rešimo s tehniko preoblikovanja (uravnoteženja) dreves.

Poznamo različne oblike iskalnih dreves, ki omogočajo operacije na slovarju s časovno zahtevnostjo reda $O(\log(n))$, kjer je n število elementov slovarja. Navedimo jih kronološko glede na njihov nastanek:

- AVL-drevesa, katerih ime sestavljajo prve črke priimkov dveh avtorjev (Adel'son-Vel'skii in Landis), nastala pa so leta 1962.
- 2-3 drevesa je definiral J.E. Hopcroft leta 1970.
- B-drevesa so posplošitev 2-3 dreves, definirala pa sta jih Bayer in McCreight leta 1972.
- Rdeča-črna drevesa (red-black trees) je definiral R. Bayer leta 1972, ime ″rdeče-črna drevesa″ pa uvedla Leo Guibas in Robert Sedgewick leta 1978.
- k-d drevesa je opisal Bentley leta 1975.
- Lomljena drevesa (splay trees) sta jih uvedla Daniel Sleator in Robert Tarjan leta 1983.

2 LOMLJENA DREVESA

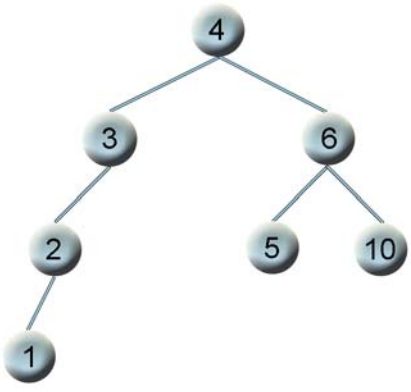
2.1 Definicija lomljenega drevesa

Lomljena drevesa so dvojiška iskalna drevesa, ki se lomijo (preoblikujejo) ob vsakem posegu v drevo:

- Pri vstavljanju element vstavimo v list drevesa. Zatem z zaporedjem operacij nad obliko drevesa, ki jih imenujemo rotacije, dosežemo, da vozlišče s tem elementom postane koren drevesa.
- Pri brisanju elementa iz drevesa vozlišče s tem elementom z zaporedjem rotacij najprej postane koren drevesa. Šele zatem ga po določenem postopku zbrišemo iz drevesa.
- Ob iskanju elementa v drevesu vozlišče z najdenim elementom, oziroma, če elementa v drevesu ni, zadnje vozlišče na poti iskanja, z zaporedjem rotacij postane koren drevesa.

Drevo lomimo iz dveh razlogov. Prvi je ta, da s tem preprečimo izrojevanje drevesa. V lomljenem drevesu pri majhnem številu elementov sicer lahko pride do položaja, ko je drevo izrojeno. Velja pa, da pri večjem številu elementov in večjem številu operacij nad drevesom to v splošnem ne bo izrojeno, ampak bo bolj ali manj uravnoteženo. Drugi razlog za lomljenje pa je, da z lomljenjem podatek, ki ga obravnavamo (iščemo, vstavljamo, ...), pride v korensko vozlišče. Ker pri številnih uporabah pogosto pride do položaja, ko več zaporednih operacij zahteva iste ali sorodno velike podatke, s tem pohitrimo izvajanje.

Lomljeno drevo je načeloma poljubno iskalno drevo:



Slika 1: Iskalno drevo.

Kot tehniko preoblikovanja drevesa uporabljamo rotacije. Za izvedbo rotacij na lomljenih drevesih vozlišče poleg podatka o obeh sinovih potrebujemo tudi podatek o očetu. Podatek o očetu potrebujemo, ker se metoda za lomljenje drevesa pomika od trenutnega elementa navzgor proti korenu. Zato v vozlišču lomljenega drevesa poleg elementa hranimo še kazalca (referenci) na levega in desnega sina, ter kazalec na očeta.

```
public class VozelLomDrevesa{

  // KOMPONENTE
  private int element;              // ključ
  private VozelLomDrevesa levi;     // kazalec na levega sina
  private VozelLomDrevesa desni;    // kazalec na desnega sina
  private VozelLomDrevesa oce;      // kazalec na očeta
  ...
}
```

2.2 Lomljeno drevo (implementacija)

V tem razdelku si bomo ogledali implementacijo lomljenega drevesa, napisano v programskem jeziku Java. Tu bomo glavne metode le navedli. Njihovo izvedbo bomo predstavili v svojih poglavjih.

Najprej definirajmo razred *`VozelLomDrevesa`*. Povedali smo že, da bomo v vozliščih hranili element (ključ) ter kazalce na levega in desnega sina ter na očeta. Že prej smo omenili, da bomo zaradi enostavnosti predpostavili, da je ključ (hranjeni element) kar celo število.

Razred vsebuje konstruktorje in metode s katerimi nastavimo in vračamo posamezne komponente tipa *`VozelLomDrevesa`*. Do spremenljivk bo mogoče dostopati in jih nastavljati samo z vnaprej določenimi metodami, torej bodo imele vse privatni dostop. Razred vsebuje še metodo za izpis elementov v vozlišču in elementov, ki so v vozliščih, potomcih tega vozlišča.

```
/**
 * Razred <code> VozelLomDrevesa </code>, predstavlja vozlišče
drevesa.
 * V vozlišču hranimo cela števila.
 *
 *  @version  1.0
 *  @author   Simon Butalič
 */
public class VozelLomDrevesa{

  // KOMPONENTE
  private int element;              // element (ključ), ki ga hranimo
  private VozelLomDrevesa levi,
                          desni;    // kazalec na levega in desnega sina
  private VozelLomDrevesa oce;      // kazalec na očeta

  // KONSTRUKTORJI
  /**
   * Ustvari novi objekt tipa VozelLomDrevesa. V vozlišču je element
   * 0, sinov in očeta pa ni.
   */
  public VozelLomDrevesa(){
    element = 0;
    levi = null;
    desni = null;
    oce = null;
  }

  /**
   * Ustvari novi objekt tipa VozelLomDrevesa, nastavi element.
```

```
 *
 * @param  x  je element, ki ga shranimo v vozlišču.
 */
public VozelLomDrevesa(int x){
  this();
  element = x;
}

/**
 * Ustvari novi objekt tipa VozelLomDrevesa, nastavi element
 * in očeta.
 *
 * @param  x  je element, ki ga shranimo v vozlišču.
 * @param  p  je oče tega vozlišča.
 */
public VozelLomDrevesa(int x, VozelLomDrevesa p){
  this(x);
  oce = p;
}

// METODE
/**
 * Pove, kakšen element je v vozlišču.
 *
 * @return  <code> int </code>.
 */
public int vrniElement(){
  return element;
}

/**
 * Spremenimo element.
 *
 * @param  x  je nova vrednost elementa v vozlišču.
 * @return  <code> void </code>.
 */
public void nastaviElement(int x){
  element = x;
}

/**
 * Vrnemo levega sina vozlišča.
 *
 * @return  <code> VozelLomDrevesa </code>.
 */
public VozelLomDrevesa vrniLeviSin(){
  return levi;
}

/**
 * Spremenimo levega sina.
 *
 * @param  vozR  je vozel, ki bo postal levi sin.
 * @return  <code> void </code>.
 */
public void nastaviLeviSin(VozelLomDrevesa vozR){
  levi = vozR;
}

/**
 * Vrnemo desnega sina vozlišča.
```

```
     *
     * @return  <code> VozelLomDrevesa </code>.
     */
    public VozelLomDrevesa vrniDesniSin(){
      return desni;
    }

    /**
     * Spremenimo desnega sina.
     *
     * @param  vozR  je vozel, ki bo postal desni sin.
     * @return  <code> void </code>.
     */

    public void nastaviDesniSin(VozelLomDrevesa vozR){
      desni = vozR;
    }

    /**
     * Vrnemo očeta.
     *
     * @return  <code> VozelLomDrevesa </code>.
     */
    public VozelLomDrevesa vrniOce(){
      return oce;
    }

    /**
     * Spremenimo očeta.
     *
     * @param  p  je vozel, ki bo novi oče vozlišča.
     * @return  <code> void </code>.
     */
    public void nastaviOce(VozelLomDrevesa p){
      oce = p;
    }

     /**
     * Spremenimo očeta korenu drevesa.
     *
     * @return  <code> void </code>.
     */
    public void nastaviOce(){
      oce = null;
    }

    /**
     * Metoda za izpis elementov v vozlišču in v potomcih.
     *
     * @return  <code> void </code>.
     */
    public void print(){
      System.out.println(" Koren " + "[ " + vrniElement() + " ] =
                         [ Levi sin: " + (vrniLeviSin() == null ? "xx" :
                         "" + vrniLeviSin().vrniElement()) + " ],
                         [ Desni sin: " + (vrniDesniSin() == null ? "xx"
                         : "" + vrniDesniSin().vrniElement()) + " ] ");
      if(vrniLeviSin() != null) vrniLeviSin().print();
      if(vrniDesniSin() != null) vrniDesniSin().print();
    }
  }
```

Tako. Sestavne dele za naše drevo imamo. Sedaj sestavimo razred, ki bo predstavitev lomljenega drevesa. V razredu *LomljenoDrevo* bomo poleg osnovnih metod, ki omogočajo vstavljanje, brisanje in iskanje elementa v drevesu, potrebovali še pomožne metode za rotacijo in lomljenje drevesa. V tem razdelku bomo metode le navedli. Njihovo izvedbo bomo predstavili v svojih poglavjih.

```
/**
 * Razred <code> LomljenoDrevo </code> vsebuje metode za
 * vstavljanje, iskanje in brisanje elementa.
 *
 *  @version  1.0
 *  @author   Simon Butalič
 */
public class LomljenoDrevo{

  // KOMPONENTE
  private VozelLomDrevesa koren;   // za drevo

  // KONSTRUKTOR
  /**
   * Ustvari novi objekt tipa LomljenoDrevo.
   */
  public LomljenoDrevo(){
    koren = null;
  }

  // METODE
  /**
   * Vstavi element v drevo, z lomljenjem poskrbimo, da vozlišče z
   * vstavljenim elementom postane koren.
   *
   * @param  x  je element, ki ga vstavljamo.
   * @return  <code> void </code>.
   */
  public void vstavi(int x){
  }

  /**
   * Poišče mesto vstavljanja v drevesu, kamor vstavimo element in
   * vstavi vozlišče z elementom, vrne kazalec na vstavljeno vozlišče.
   * Če je element že v drevesu, vrne kazalec na vozlišče z tem
   * elementom.
   *
   * @param  x  je element, ki ga vstavljamo.
   * @param  vozR  je koren drevesa.
   * @return  <code> VozelLomDrevesa </code>, kazalec na
   * vstavljeno vozlišče oz. vozlišče z elementom v drevesu.
   */
  private VozelLomDrevesa vstavi(int x, VozelLomDrevesa vozR){
  }

  /**
   * Poišče vozlišče z elementom, izbriše ga iz drevesa in preuredi
   * drevo. Če vozlišča z elementom ni v drevesu, zadnje vozlišče na
   * poti iskanja prestavi v koren drevesa.
   *
   * @param  x  je element, ki ga brišemo.
   * @return  <code> void </code>.
   */
```

```
public void brisi(int x){
}

/**
 * Poišče vozlišče z elementom, prestavi ga v koren drevesa. Če
 * vozlišča z elementom ni v drevesu, zadnje vozlišče na poti
 * iskanja prestavi v koren drevesa.
 *
 * @param  x  je element, ki ga iščemo.
 * @return  <code> true </code>, če je element v drevesu.
 */
public boolean isci(int x){
}

/**
 * Poišče vozlišče z elementom, vrne kazalec na najdeno vozlišče. Če
 * vozlišča z elementom ni v drevesu, vrne kazalec na zadnje
 * vozlišče na poti iskanja.
 *
 * @param  x  je element, ki ga iščemo.
 * @param  vozR  je koren (pod)drevesa.
 * @return  <code> VozelLomDrevesa </code>, kazalec na
 * iskano oz. zadnje vozlišče.
 */
private VozelLomDrevesa isci(int x, VozelLomDrevesa vozR){
}

/**
 * Pri vozlišču vozP lomi drevo (ga preoblikuje z rotacijami),
 * vozlišče prestavi v koren drevesa.
 *
 * @param  vozP  je vozel, kandidat za novi koren.
 * @return  <code> VozelLomDrevesa </code>, novi koren drevesa.
 */
private VozelLomDrevesa lomljenjeDrevesa(VozelLomDrevesa vozP){
}

/**
 * Zarotira vozlišče vozX, desna rotacija.
 *
 * @param  vozP  je oče vozlišča, ki ga rotiramo.
 * @return  <code> VozelLomDrevesa </code>, kazalec na vozX.
 */
private VozelLomDrevesa desnaRotacija(VozelLomDrevesa vozP){
}

/**
 * Zarotira vozlišče vozX, leva rotacija.
 *
 * @param  vozP  je oče vozlišča (vozX), ki ga rotiramo.
 * @return  <code> VozelLomDrevesa </code>, kazalec na vozX.
 */
private VozelLomDrevesa levaRotacija(VozelLomDrevesa vozP){
}

/**
 * Metoda za izpis drevesa.
 *
 * @return  <code>void</code>.
 * @return  <code>"DREVO JE PRAZNO!"</code>, drevo je prazno.
 */
```

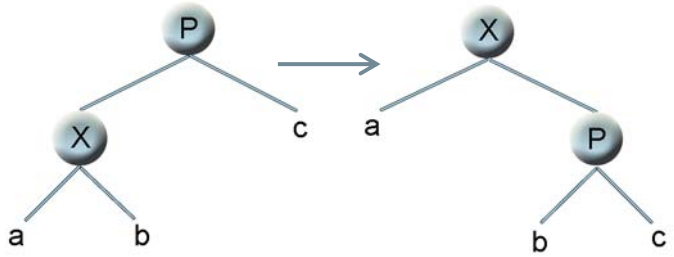
```
    public void print()throws Exception{
      if(koren == null)
         throw new NullPointerException("DREVO JE PRAZNO!");
      else koren.print();
    }
}
```

2.3 Lomljenje drevesa in rotacije

Lomljenje drevesa je operacija, s katero premaknemo trenutni element proti korenu oziroma v koren drevesa. S tem želimo doseči, da je drevo bolj uravnoteženo ter da je obravnavano vozlišče bližje vrhu drevesa. Predpostavljamo namreč, da bomo tisto vozlišče, ki ga obravnavamo, v kratkem spet iskali. Zato je smiselno, da je čim bližje vrhu drevesa, saj ga bomo tako hitreje našli.

2.3.1 Rotacije

Za lomljenje drevesa uporabljamo enojne in dvojne rotacije. Rotacije so postopki, ki zamenjajo vlogo sinov in očeta. Seveda pri tem ne smemo pokvariti »iskalnosti« drevesa (drevo mora biti še vedno iskalno). Enojne rotacije so enostavnejše in jih uporabljamo predvsem takrat, ko je oče vozlišča, ki ga rotiramo, koren. Dvojna rotacija je kombinacija dveh enojnih rotacij. V večini primerov enojne rotacije ne zadoščajo, da bi se izognili izrojevanju drevesa. Npr., če lomimo drevo z enojnimi rotacijami z elementom na dnu izrojenega drevesa, bi se zopet izoblikovalo izrojeno drevo.
Poznamo leve in desne rotacije. Leva rotacija je zrcalna desni rotaciji. Enojna rotacija je zamenjava očeta - vozlišča z elementom *P* z njegovim levim (desna rotacija) ali desnim (leva rotacija) sinom - vozliščem z elementom *X*.



Slika 2: Enojna desna rotacija.

Uporabljali bomo naslednje tipe rotacij: levo, desno, levo-desno, desno-levo, levo-levo in desno-desno rotacijo.
Rotacije uporabljamo pri vstavljanju, brisanju in iskanju vozlišča z elementom *X* oz. pri vsakem posegu v drevo. Ločimo dva robna in dva splošna primera uporabe rotacij:

- Prvi robni primer uporabe rotacij je ta, ko rotacij sploh ne potrebujemo, saj je ustrezno vozlišče že koren.
- Drugi robni primer nastopi, ko je oče obravnavanega vozlišča koren drevesa.
- Prvi splošni primer rotacije uporabimo, če sta tako obravnavano vozlišče, kot tudi njegov oče, leva (oziroma oba desna) sinova svojih očetov.

- Drugi splošni primer rotacije uporabimo, če je vozlišče levi sin očeta, ki pa je desni sin svojega očeta, oziroma ravno obratno.

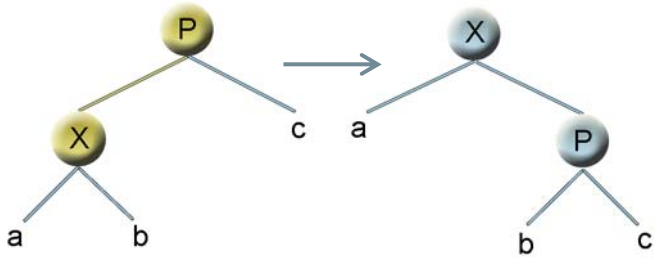
Na slikah bomo z rumeno označili vozlišča, okoli katerih izvajamo rotacijo (vozlišče samo in njegovega očeta).

1. Robna primera:
   a. Trenutno vozlišče z elementom *X* je v korenu, rotacija ni potrebna.

Slika 3: Trenutno vozlišče z elementom *X* je v korenu.

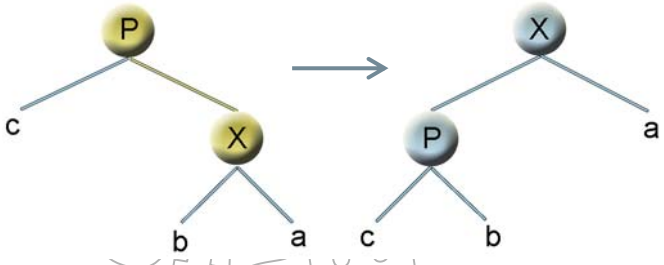
   b. Trenutno vozlišče z elementom *X* ima za očeta koren – vozlišče z elementom *P*. Za lomljenje uporabimo enojno rotacijo in postopek je končan. Ločimo primer, ko je vozlišče z elementom *X* levi sin korena in ko je vozlišče z elementom *X* desni sin korenskega vozlišča.

      Poglejmo si primer, ko pride do desne rotacije:
      Vozlišče z elementom *X* postane novi koren, vozlišče z elementom *P* pa postane njegov desni sin (in seveda je vozlišče z elementom *X* njegov oče). Če poddrevo (na sliki označeno z *b*) obstaja (ni prazno), ga pripnemo kot levo poddrevo vozlišča z elementom *P*. Pri tem moramo paziti, da ne pozabimo vozliščem, ki postanejo drugačni sinovi, nastaviti očete. Tako moramo vozlišču z elementom *X* nastaviti očeta na `null`, saj je vozlišče z elementom *X* novi koren, torej je brez očeta.



Slika 4: Iskalno drevo pred desno rotacijo in po rotaciji.

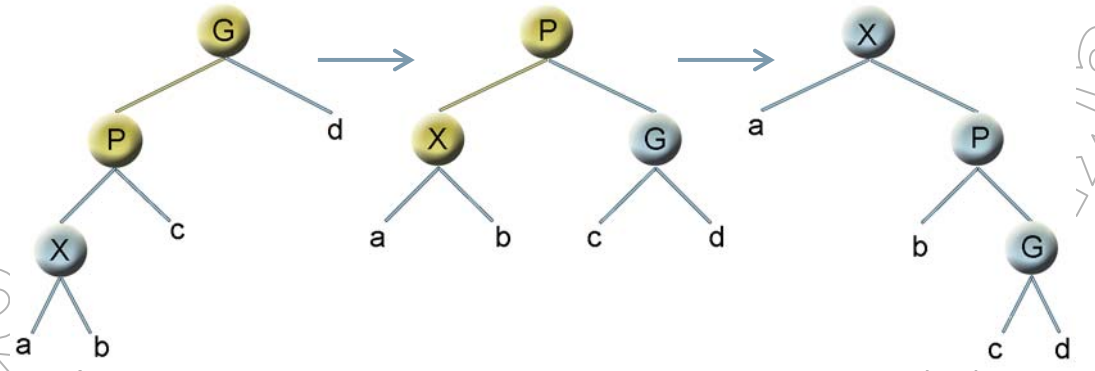
Leva rotacija je zrcalna slika desne rotacije in je ne bomo posebej razlagali.



Slika 5: Iskalno drevo pred levo rotacijo in po rotaciji.

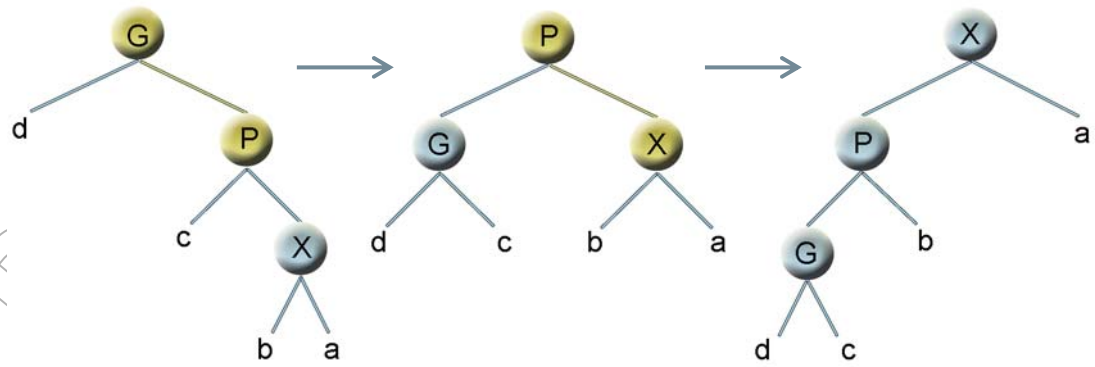
2. Splošna primera:
   a. Če sta tako vozlišče z elementom *X*, kot tudi vozlišče z elementom *P*, oba leva (oziroma oba desna) sinova svojih očetov, izvedemo dvojno rotacijo. Dvakrat izvedemo isto enojno rotacijo – torej izvedemo desno-desno oziroma levo-levo rotacijo. Naj bo vozlišče z elementom *G* stari oče vozlišča z elementom *X* (torej oče vozlišča z elementom *P*). Če sta tako vozlišče z elementom *X* in vozlišče z elementom *P* leva sinova svojih očetov, izvedemo dvojno desno-desno rotacijo. Dvojno desno-desno rotacijo izvedemo z dvema enojnima desnima zaporednima rotacijama. Prvo desno rotacijo izvedemo z vozlišči z elementoma *P* in *G*, drugo desno rotacijo pa izvedemo z vozlišči z elementoma *X* in *P*.

      Oglejmo si, kako desno-desna rotacija poteka. Od tu naprej bomo večkrat uporabili krajše izražanje in sicer bomo rekli kar vozlišče *X*. S tem bomo mislili na vozlišče, ki vsebuje element *X*.
      Prvo desno rotacijo izvedemo z vozlišči *P* in *G*. Vozlišče *P* postane koren poddrevesa, vozlišče *G* pa postane njegov desni sin in ima za očeta vozlišče *P*. Vozlišču *P* zamenjamo očeta z očetom vozlišča *G*. Če vozlišče *G* ni koren drevesa, potem očetu vozlišča *G* nastavimo vozlišče *P* za levega oz. desnega sina. Če ima vozlišče *P* neprazno desno poddrevo (na sliki označeno s *c*) potem poddrevo *c* postane levo poddrevo vozlišča *G*. Pri tem seveda ne smemo pozabiti, da moramo vozliče *G* nastaviti kot očeta korena tega poddrevesa (*c*). Drugo desno rotacijo izvedemo z vozlišči *X* in *P*. Tako z dvema desno-desno zaporednima rotacijama postane vozlišče *X* koren poddrevesa, njegov prejšnji oče (vozlišče *P)* njegov desni sin in prejšnji stari oče (vozlišče *G)* pa desni sin prejšnjega očeta, vozlišča *P*. Desno poddrevo vozlišča *X* (označeno z *b*) postane levo poddrevo vozlišča *P*, desno poddrevo vozlišča *P* (*c*), pa postane levo poddrevo vozlišča *G*.



Slika 6: Desno-desna rotacija. Iskalno drevo pred prvo desno rotacijo, po prvi desni rotaciji in drugi desni rotaciji.

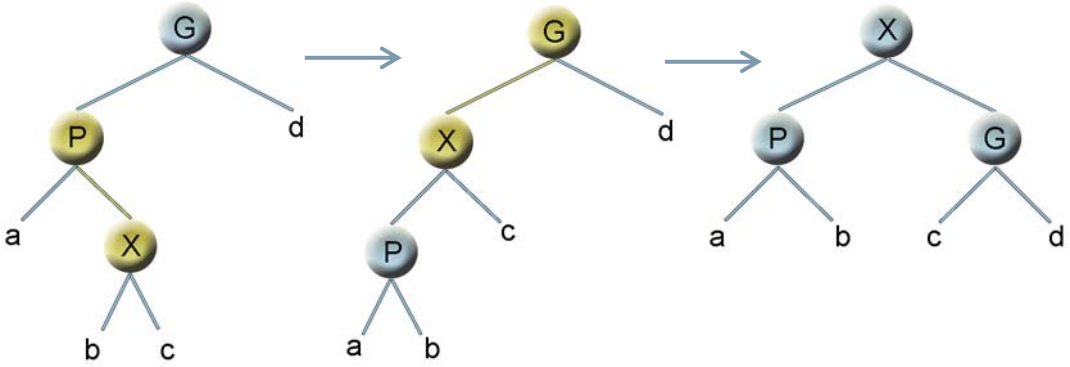
Levo-leva rotacija je zrcalna slika desno-desne rotacije in je ne bomo posebej razlagali. Oglejmo si le sliko



Slika 7: Levo-leva rotacija. Iskalno drevo pred prvo levo rotacijo, po prvi levi rotaciji in drugi levi rotaciji.

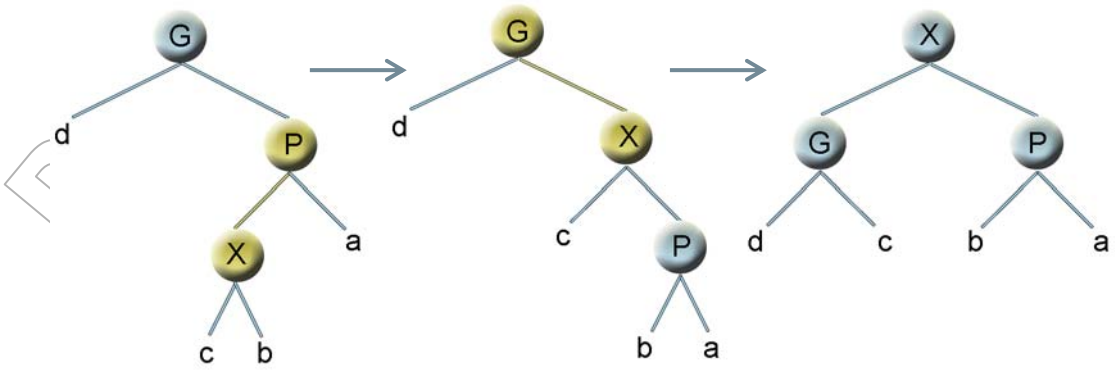
b. Če je vozlišče *X* na nasprotni strani očeta – vozlišča *P*, kot je oče - vozlišče *P* glede na svojega očeta – vozlišče *G,* izvedemo dvojno levo-desno oziroma desno-levo rotacijo. Prva pride v poštev, če je vozlišče *X* desni sin vozlišča *P*, vozlišče *P* pa levi sin vozlišča *G*, druga pa v obratnem primeru.
Dvojno levo-desno rotacijo izvedemo z dvema zaporednima enojnima rotacijama. Najprej izvedemo levo in nato desno rotacijo. Prvo levo rotacijo izvedemo z vozlišči *X* in *P*, drugo desno rotacijo pa izvedemo z vozlišči *X* in *G.*

Oglejmo si, kako leva-desna rotacija poteka.
Prvo levo rotacijo izvedemo z vozlišči *X* in *P*. Vozlišče *X* postane levi sin korena - vozlišča *G* in ima za očeta vozlišče *G*. Vozlišče *P* postane levi sin vozlišča *X* in ima za očeta vozlišče *X*. Če ima vozlišče *X* neprazno levo poddrevo (na sliki označeno s *b*), potem poddrevo *b* postane desno poddrevo vozlišča *P*. Pri tem seveda ne smemo pozabiti, da moramo vozlišče *P* nastaviti kot očeta korena tega poddrevesa (*b*). Drugo desno rotacijo izvedemo z vozlišči *X* in *G.* Tako z dvema zaporednima rotacijama postane vozlišče *X* koren poddrevesa, njegov prejšnji oče – vozlišče *P* njegov levi sin in prejšnji stari oče - vozlišče *G* desni sin. Levo poddrevo vozlišča *X* (označeno z *b*) postane desno poddrevo vozlišča *P*, desno poddrevo (*c*) pa postane levo poddrevo vozlišča *G*.



Slika 8: Levo-desna rotacija. Iskalno drevo pred prvo levo rotacijo, po prvi levi rotaciji in drugi desni rotaciji.

Pri desno-levi rotacija je postopek zrcalen tistemu, ki smo ga opisali prej, zato ga ne bomo opisovali podrobneje.

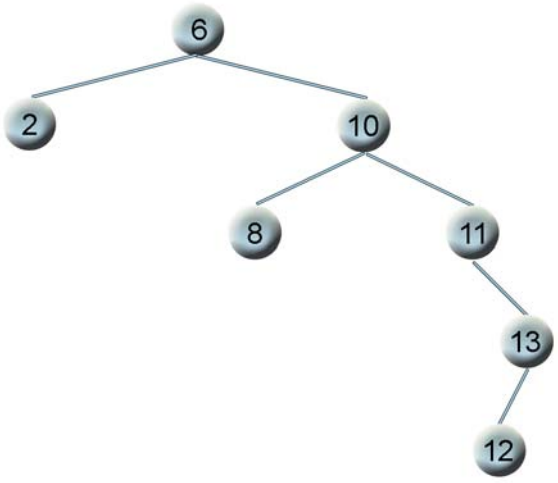


Slika 9: Desno-leva rotacija. Iskalno drevo pred prvo desno rotacijo, po prvi desni rotaciji in drugi levi rotaciji.

Če torej izvajamo desno-desno rotacijo (ali levo-levo), začnemo »zgoraj« - pri starem očetu in očetu. V primeru levo-desne oziroma desno-leve rotacije pa začnemo pri vozlišču samem in njegovem očetu.

Da bomo dobili boljši občutek za to, kaj rotacije počnejo, si oglejmo primer, ko izvajamo rotacije pri sestavljanju lomljenega drevesa. Kdaj in kako pride do lomljenja, si bomo ogledali v nadaljnjih razdelkih. Tu želimo ilustrirati le izvajanje rotacij. Rumena barva kot prej označuje vozlišči uporabljeni za rotacijo (vozlišče samo in njegovega očeta).

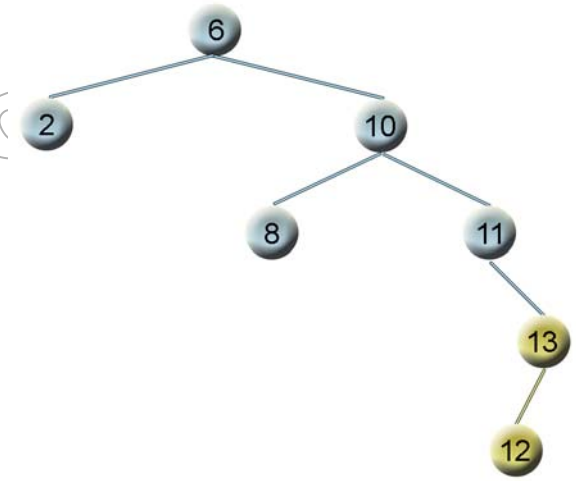
Dano je iskalno drevo, v katerega smo ravno vstavili vozlišče z elementom *12*.



Slika 10: Iskalno drevo po vstavljanju elementa *12*.

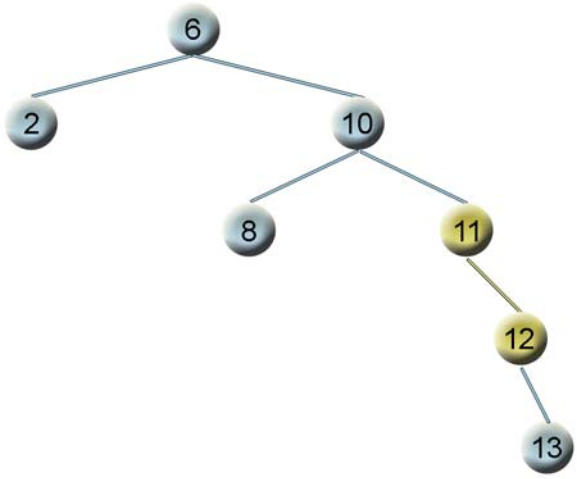
Vozlišče *12* želimo z več rotacijami prestaviti v koren drevesa. Analizirajmo položaj.

Vozlišče *12* ima za očeta vozlišče *13* in za starega očeta vozlišče *11*. Oče (vozlišče *13*) je desni sin starega očeta (vozlišča *11*). Ker je vozlišče *12* levi sin svojega očeta, izvedemo desno-levo rotacijo. Prvo desno rotacijo izvedemo z vozlišči *12* in *13*.



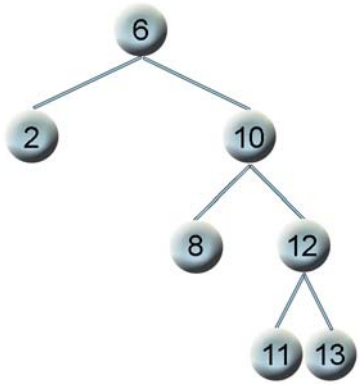
Slika 11: Na vrsti je desna rotacija z vozlišči *12* in *13*.

Vozlišče *12* postane koren poddrevesa, vozlišče *13* pa njegov desni sin. Ker je vozlišče *13* desni sin svojega očeta (vozlišča *11*), postane novi koren poddrevesa, vozlišče *12*, desni sin vozlišča *11*. Nadaljujemo z levo rotacijo vozlišč *11* in *12*.



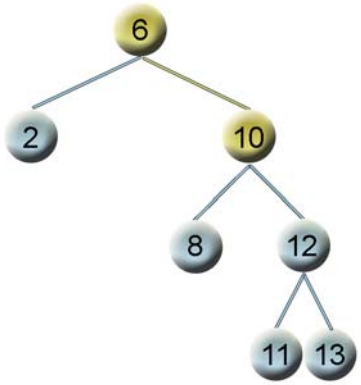
Slika 12: Na vrsti je leva rotacija z vozlišči *11* in *12*.

Vozlišče *12* postane koren poddrevesa. Sedaj ima za očeta vozlišče *10* in je njegov desni sin. Vozlišče *11* je postalo levi sin vozlišča *12*. S tem smo zaključili desno-levo rotacijo.



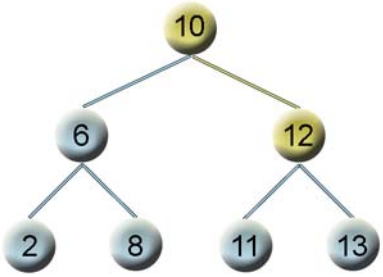
Slika 13: Drevo po zaključeni prvi desno-levi rotaciji.

Ker vozlišče *12* še ni v korenu drevesa, nadaljujemo z rotacijami.
Vozlišče *12* ima za očeta vozlišče *10* in za starega očeta vozlišče *6*. Oba sinova v tej verigi sta desna sinova, zato izvedemo levo-levo rotacijo. Prvo levo rotacijo bomo izvedli s starim očetom in očetom, torej z vozlišči *6* in *10*.



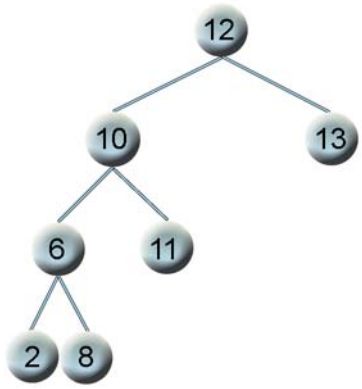
Slika 14: Na vrsti je leva rotacija z vozlišči *10* in *6*.

Oče (vozlišče *10*) postane koren drevesa in zato nima več očeta (ustrezen kazalec postavimo na *null*). Vozlišče *6* postane levi sin vozlišča *10*. Ne pozabimo, da smo za slednje morali spremeniti tako levega sina vozlišča *10*, kot očeta vozlišča *6*. Levi sin vozlišča *10*, vozlišče *8*, postane desni sin vozlišča *6*. Nadaljujemo z drugo levo rotacijo z vozlišči *12* in *10*.



Slika 15: Na vrsti je leva rotacija z vozlišči *12* in *10*.

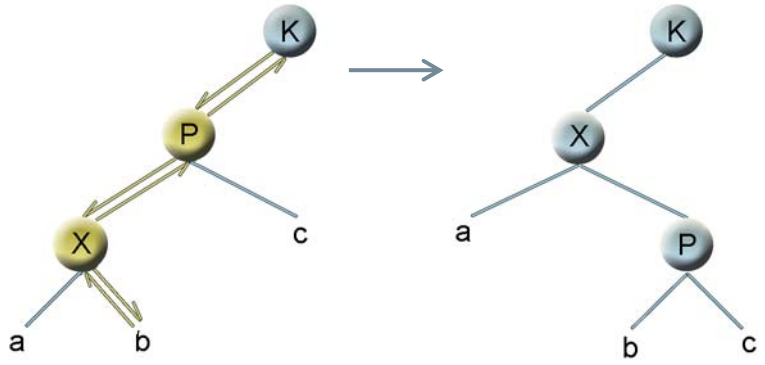
Vozlišče *12* postane koren drevesa (seveda je sedaj brez očeta), vozlišče *10* pa postane njegov levi sin. Levi sin vozlišča *12*, vozlišče *11*, postane desni sin vozlišča *10*. Ker je s tem vozlišče *12* prišlo v koren drevesa, končamo z lomljenjem drevesa.



Slika 16: Preoblikovano drevo (vozlišče *12* je v korenu).

2.3.2 Metodi za rotaciji

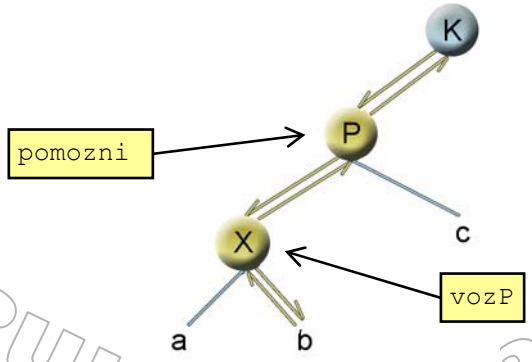
Da bomo lahko izvajali rotacije, sestavimo metodi *desnaRotacija(VozelLomDrevesa vozP)* in *levaRotacija(VozelLomDrevesa vozP)*. Metodi za rotacijo kot rezultat vračata kazalec na novi položaj vozlišča *X*. Da bomo lažje sledili metodi, si ponovno oglejmo sliko desne rotacije z vozlišči *X* in *P*. Vozlišče *P* ima očeta *K*.



Slika 17: Iskalno drevo pred desno rotacijo in po rotaciji.

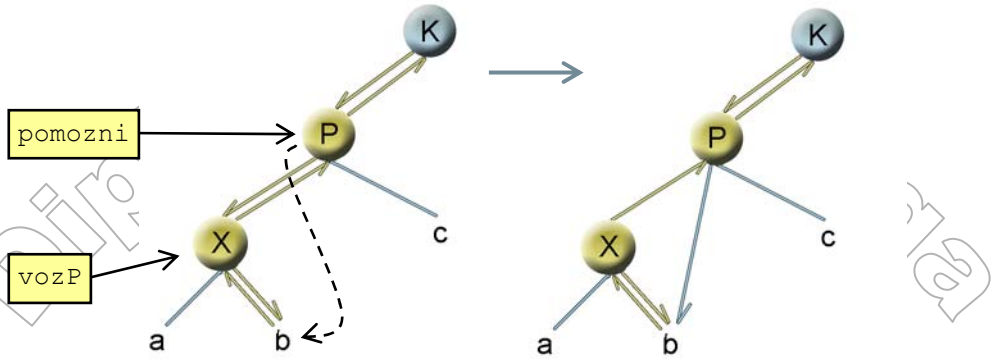
V metodi bomo z *vozX* označili kazalec na vozlišče *X* in z *vozP* kazalec na vozlišče *P*. Dvojna (usmerjena relacija) povezava prikazuje kazalec na očeta oz. kazalec na sina. Na slikah bomo s črtkano črto označili novi položaj vozlišča, s pikčasto črto pa novega očeta vozlišča. Metodo za desno rotacijo si oglejmo podrobneje in jo bomo vmes prekinjali s slikami stanja.

```
/**
 * Zarotira vozlišče vozX, desna rotacija.
 *
 * @param  vozP  je oče vozlišča, ki ga rotiramo.
 * @return  <code> VozelLomDrevesa </code>, kazalec na vozX.
 */
private VozelLomDrevesa desnaRotacija(VozelLomDrevesa vozP){

  VozelLomDrevesa pomozni = vozP;
  // vozP kazalec na novi koren (vozlišče X)
  vozP = (VozelLomDrevesa)vozP.vrniLeviSin();
```



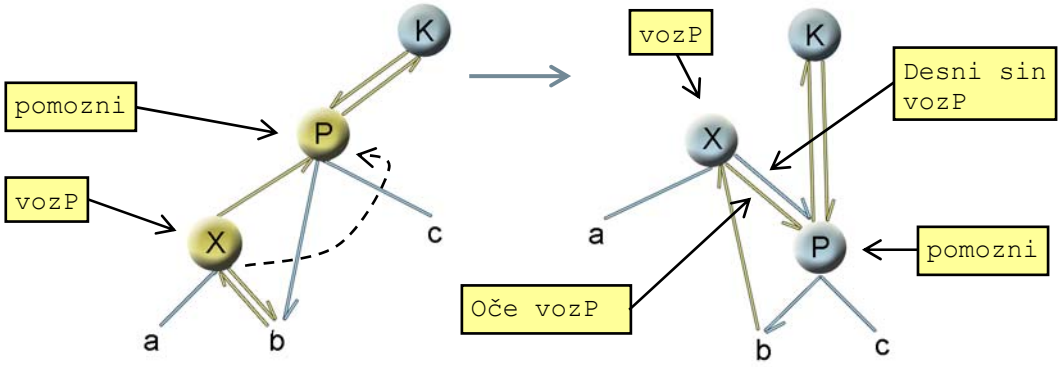
Slika 18: Prestavimo kazalca.

```
  // pomožnemu nastavimo levega sina
  pomozni.nastaviLeviSin(vozP.vrniDesniSin());
```



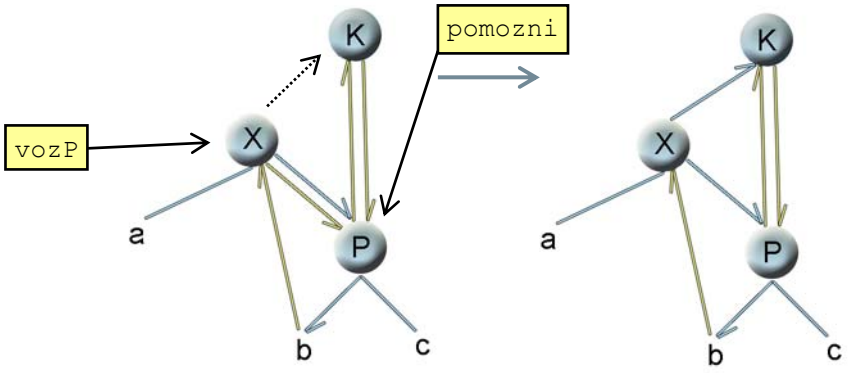
Slika 19: Drevo pred nastavitvijo levega sina vozlišču *P* (*pomozni*) in po nastavitvi levega sina.

```
// vozP nastavimo desnega sina
vozP.nastaviDesniSin(pomozni);
```



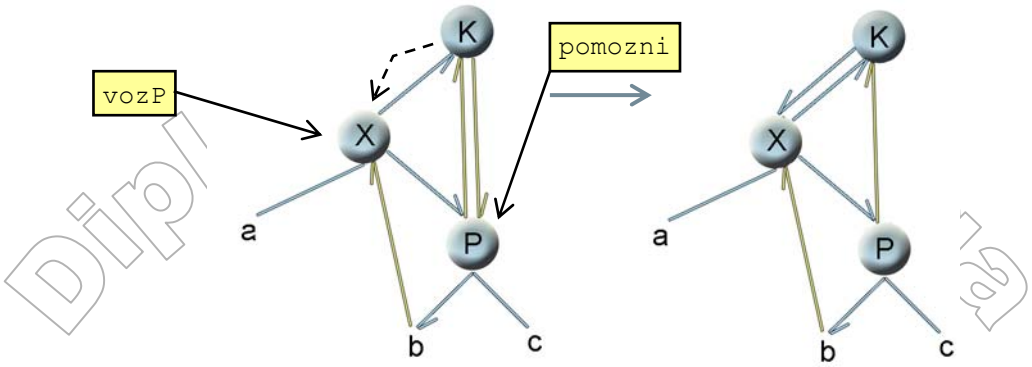
Slika 20: Drevo pred nastavitvijo desnega sina vozlišču *X* (*vozP*) in po nastavitvi desnega sina.

```
// novemu korenu vozlišču X (vozP), nastavimo za očeta
// očeta starega korena
vozP.nastaviOce(pomozni.vrniOce());
```



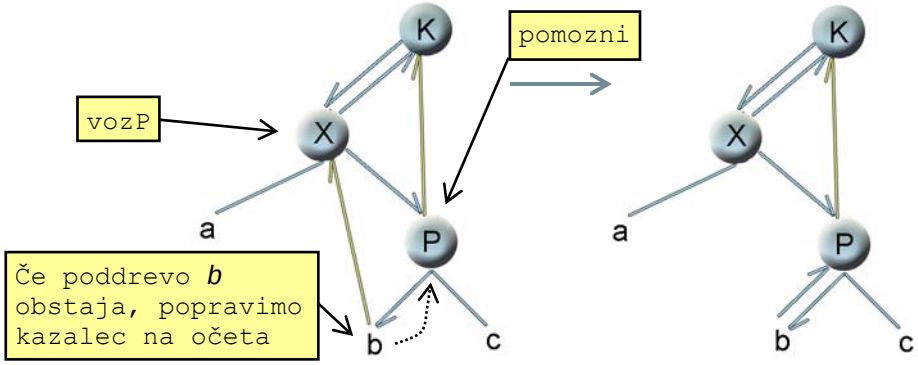
Slika 21: Drevo pred nastavitvijo očeta vozlišču *X* (*vozP*), in po nastavitvi očeta.

```
// če ima stari koren očeta
if(pomozni.vrniOce() != null){
  // če je stari koren levi sin glede na svojega očeta
  if(pomozni.vrniOce().vrniLeviSin() == pomozni)
    // novi koren je sedaj levi sin istega očeta
    pomozni.vrniOce().nastaviLeviSin(vozP);
```



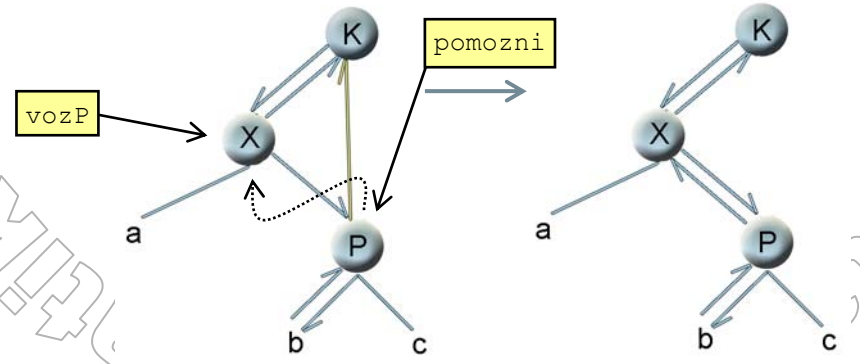
Slika 22: Drevo pred nastavitvijo levega sina očetu starega korena (vozlišču *K*), novi koren (vozlišče *X*) in po njem.

```
    // če je stari koren desni sin glede na svojega očeta
    else if(pomozni.vrniOce().vrniDesniSin() == pomozni)
      // novi koren je sedaj desni sin istega očeta
      pomozni.vrniOce().nastaviDesniSin(vozP);
  } // if(pomozni.vrniOce() != null)
  // če obstaja levi sin od desnega sina korena
  if(pomozni.vrniLeviSin() != null)
    // za očeta nastavimo novega desnega sina korena
    ((VozelLomDrevesa)pomozni.vrniLeviSin()).nastaviOce(pomozni);
```



Slika 23: Poddrevesu *b* popravimo kazalec na očeta.

```
  // novemu desnemu sinu korena, za očeta nastavimo novi koren
  pomozni.nastaviOce(vozP);
```



Slika 24: Drevo pred nastavitvijo očeta poddrevesu *b* in po nastavitvi očeta.

```
  // vrnemo kazalec na novo korensko vozlišče
  return vozP;
} // konec desnaRotacija
```

Leva rotacija je podobna, zato bodo tu zadoščali le komentarji v metodi.

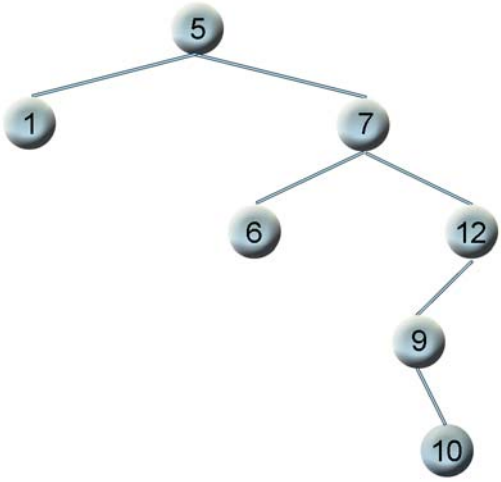
```
/**
 * Zarotira vozlišče vozX, leva rotacija.
 *
 * @param  vozP  je oče vozlišča, ki ga rotiramo.
 * @return  <code> VozelLomDrevesa </code>, kazalec na vozX.
 */
private VozelLomDrevesa levaRotacija(VozelLomDrevesa vozP){

  VozelLomDrevesa pomozni = vozP;
  // Naj vozP kaže na novi koren
  vozP = (VozelLomDrevesa)vozP.vrniDesniSin();
  // novemu levemu sinu korena nastavimo desnega sina
  pomozni.nastaviDesniSin(vozP.vrniLeviSin());
  // korenu nastavimo levega sina
  vozP.nastaviLeviSin(pomozni);
  // novemu korenu nastavimo kot očeta očeta starega korena
  // če stari koren nima očeta, smo pač očeta nastavili na null
  vozP.nastaviOce(pomozni.vrniOce());
  // če ima stari koren očeta, je potrebno popraviti levega
  // oz. desnega sina tega vozlišča
  if(pomozni.vrniOce() != null){
    // če je stari koren levi sin svojega očeta
    if(pomozni.vrniOce().vrniLeviSin() == pomozni)
      // novi koren je sedaj levi sin istega očeta
      pomozni.vrniOce().nastaviLeviSin(vozP);
    // če je stari koren desni sin svojega očeta
    else if(pomozni.vrniOce().vrniDesniSin() == pomozni)
      // novi koren je sedaj desni sin istega očeta
      pomozni.vrniOce().nastaviDesniSin(vozP);
  } // if(pomozni.vrniOce() != null)
  // če obstaja desni sin levega sina korena
  if(pomozni.vrniDesniSin() != null)
    // kot očeta nastavimo korenov novi desni sin
    ((VozelLomDrevesa)pomozni.vrniDesniSin()).nastaviOce(pomozni);
  // novemu levemu sinu korena, za očeta nastavimo novi koren
  pomozni.nastaviOce(vozP);
  // vnemo novi koren
  return vozP;
} // konec levaRotacija
```

Kot smo že omenili, rotacije uporabljamo pri lomljenju dreves. Tako bosta metodi za levo in desno rotacijo sprejeli od metode za lomljenje drevesa `lomljenjeDrevesa(VozelLomDrevesa vozP)` kazalec na očeta vozlišča *X*, kjer drevo lomimo. V primeru desne-desne in leve-leve rotacije pa bomo metodi za prvo desno oziroma levo rotacijo poklicali s parametroma, ki bosta oče in stari oče vozlišča *X*.

2.3.3 Lomljenje drevesa

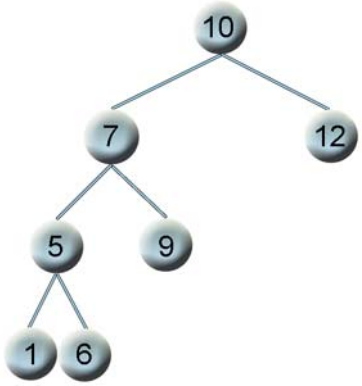
Z lomljenjem drevesa premaknemo trenutno vozlišče v koren drevesa. S tem želimo doseči, da je drevo bolj uravnoteženo ter da je obravnavano vozlišče bližje vrhu drevesa. Predpostavljamo namreč, da bomo tisto vozlišče, ki ga obravnavamo, v kratkem spet iskali. Zato je smiselno, da je čim bližje vrhu drevesa, saj ga bomo tako hitreje našli.

Oglejmo si zgled. Po zaporedju operacij je naše iskalno drevo



Slika 25: Iskalno drevo pred lomljenem drevesa.

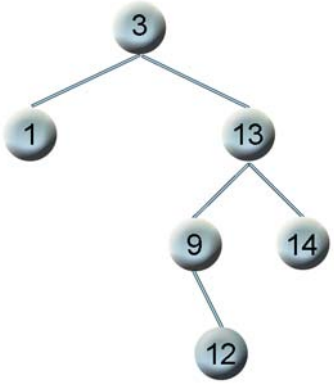
Zadnje vozlišče, ki je v operacijah nastopalo, je vozlišče *10*. Z zaporedjem rotacij ga spravimo v koren (zlomimo drevo) in dobimo



Slika 26: Iskalno drevo po končanem lomljenju drevesa (primer-a 25).

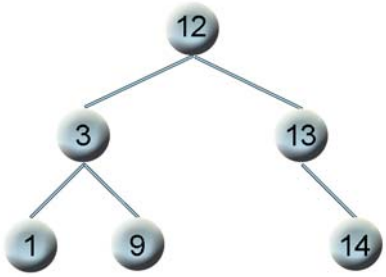
Znižala se je višina drevesa, kar pomeni, da je drevo bolj uravnoteženo.

Oglejmo si še en zgled. Po zaporedju operacij je naše iskalno drevo



Slika 27: Iskalno drevo pred lomljenem drevesa.

Zadnje vozlišče, ki je v operacijah nastopalo, je vozlišče *12*. Z zaporedjem rotacij ga spravimo v koren in dobimo

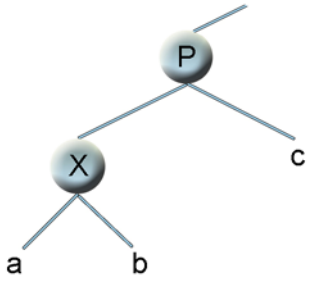


Slika 28: Iskalno drevo po končanem lomljenju drevesa (primer-a 27).

Tudi v tem primeru se je višina drevesa znižala, saj je smisel lomljenja ravno uravnoteženje drevesa.

2.3.4 Metoda za lomljenje

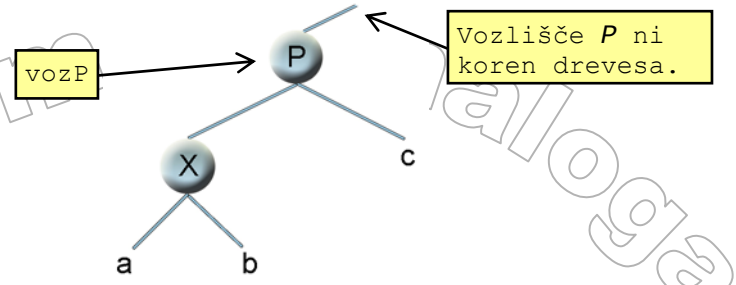
Za lomljenje drevesa sestavimo metodo *`lomljenjeDrevesa(VozelLomDrevesa vozP)`*. Vozlišče *`vozP`* je kazalec na očeta vozlišča *X* v primeru desne-desne ali leve-leve rotacije pa za prvo rotacijo kazalec na starega očeta vozlišča *X*. Metoda lomi drevo z ustreznimi rotacijami in določi kazalec na novi položaj vozlišča *X*. Za tem se celotni postopek ponovi od novega položaja vozlišča *X* navzgor, dokler ne pridemo do korena drevesa. Metodo za lomljenje drevesa si oglejmo podrobneje. Vmes jo bomo prekinjali s slikami stanja. Da bomo lažje sledili metodi, si ponovno oglejmo sliko



Slika 29: Iskalno drevo.

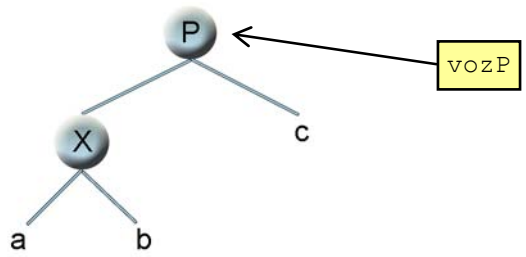
Levo-leva rotacija je zrcalna slika desno-desne rotacije in levo-desna je zrcalna slika desno-leve rotacije, zato bodo tu zadoščali le komentarji v metodi.

```
/**
 * Pri vozlišču vozP lomi drevo (ga preoblikuje z rotacijami),
 * vozlišče prestavi v koren drevesa.
 *
 * @param  vozP  je vozlišče X.
 * @return  <code> VozelLomDrevesa </code>, novi koren drevesa.
 */
private VozelLomDrevesa lomljenjeDrevesa(VozelLomDrevesa vozP){

  // če je drevo prazno ali je vozP koren, lomljenje ni potrebno
  if((vozP == null) || (vozP.vrniOce()== null)) return vozP;
  // če ima koren drevesa vsaj enega sina
  else{
    int x = vozP.vrniElement();
    vozP = vozP.vrniOce();
```



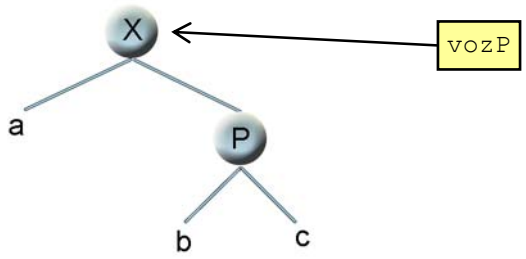
Slika 30: Prestavimo kazalca.

```
// premikanje po drevesu navzgor
while(true){
  // točka 1.b, enojni rotaciji (vozP je koren)
  if(vozP.vrniOce() == null){

    // Vozlišče vozP je torej koren. Če je vozlišče X levi
    // sin (glej sliko 30), izvedemo desno rotacijo. Če pa je
    // vozlišče X desni sin izvedemo, levo rotacijo
    // (v veji else, ki sledi).
```



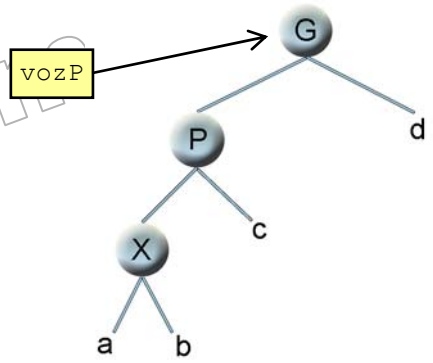
Slika 31: Iskalno drevo pred desno rotacijo.

```
    //če je vozlišče X levi sin vozlišča P
    if(vozP.vrniLeviSin() != null &&
       vozP.vrniLeviSin().vrniElement() == x)
      vozP = desnaRotacija(vozP);
```



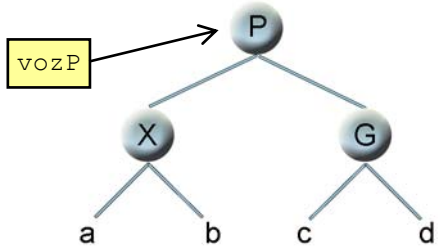
Slika 32: Drevo po zaključeni desni rotaciji.

```
    // če je vozlišče X desni sin vozlišča P
  else vozP = levaRotacija(vozP);
  } // if vozP je koren

  // točka 2.a, desna-desna rotacija
  // Če vozlišče vozP ni koren in sta tako vozlišče X, kot tudi
  // vozlišče P, oba leva sinova svojih očetov, izvedemo dvojno
  // desno-desno rotacijo (glej sliko 33).

  else if(vozP.vrniLeviSin() != null &&
          vozP.vrniLeviSin().vrniElement() == x &&
          vozP.vrniOce().vrniLeviSin() == vozP){
    vozP = vozP.vrniOce();
```



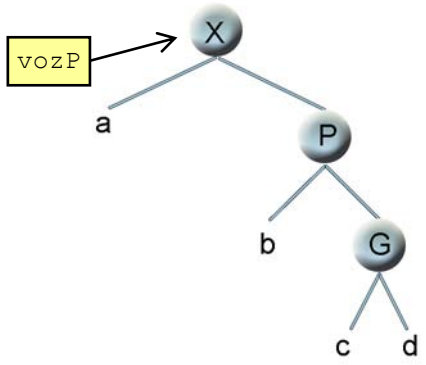
Slika 33: Iskalno drevo pred desno-desno rotacijo.

```
// prva desna rotacija s starim očetom vozlišča X
vozP = desnaRotacija(vozP);
```



Slika 34: Drevo po zaključeni prvi desni rotaciji.

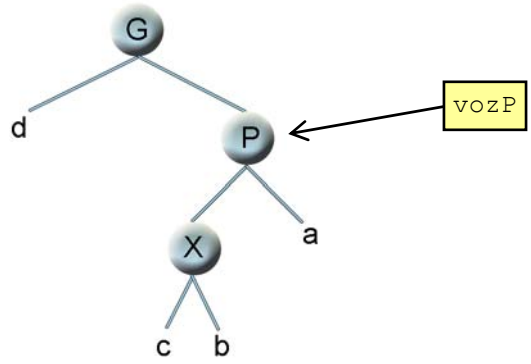
```
// druga desna rotacija z očetom vozlišča X
vozP = desnaRotacija(vozP);
```



Slika 35: Drevo po zaključeni desno-desni rotaciji.

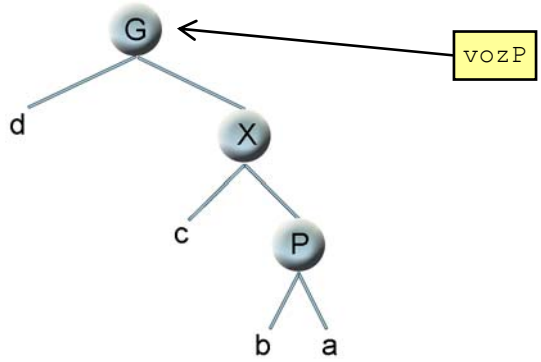
```
} // else if(vozP.vrniLeviSin() != null...

// točka 2.b, leva-leva rotacija
// Če vozlišče vozP ni koren in sta tako vozlišče X,
// kot tudi vozlišče P, oba desna sinova svojih očetov,
// izvedemo dvojno levo-levo rotacijo

else if(vozP.vrniDesniSin() != null &&
        vozP.vrniDesniSin().vrniElement() == x &&
        vozP.vrniOce().vrniDesniSin() == vozP){
  vozP = vozP.vrniOce();
  // prva leva rotacija s starim očetom vozlišča X
  vozP = levaRotacija(vozP);
  // druga leva rotacija z očetom vozlišča X
  vozP = levaRotacija(vozP);
} // else if(vozP.vrniDesniSin() != null...
```

```
// točka 2.b, desna-leva rotacija
// Če vozlišče vozP ni koren in je vozlišče X levi sin
// vozlišča P, vozlišče G je stari oče vozlišča X in
// vozlišče P desni sin vozlišča G, izvedemo
// desno-levo rotacijo (glej sliko 36)

else if(vozP.vrniLeviSin() != null
        && vozP.vrniLeviSin().vrniElement() == x
        && vozP.vrniOce().vrniDesniSin() == vozP){
```



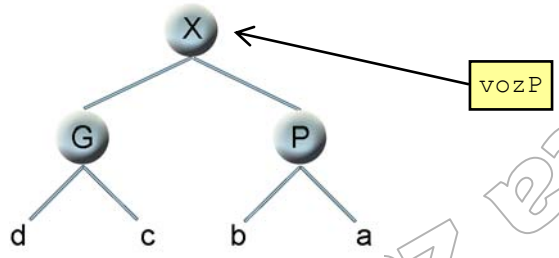
Slika 36: Iskalno drevo pred desno-levo rotacijo.

```
// prva desna rotacija z očetom vozlišča X
vozP = desnaRotacija(vozP);
vozP = vozP.vrniOce();
```



Slika 37: Drevo po zaključeni prvi desni rotaciji.

```
// druga leva rotacija z starim očetom vozlišča X
vozP = levaRotacija(vozP);
```



Slika 38: Drevo po zaključeni desni-levi rotaciji.

```
} // else if(vozP.vrniLeviSin() != null

// točka 2.b, leva-desna Rotacija
// Če vozlišče vozP ni koren in je vozlišče X desni sin
```

```
        // vozlišča P, vozlišče G je stari oče vozlišča X in
        // vozlišče P levi sin vozlišča G, izvedemo levo-desno
        // rotacijo

        else{
          // prva leva rotacija z očetom vozlišča X
          vozP = levaRotacija(vozP);
          vozP = vozP.vrniOce();
          // druga desna rotacij s starim očetom vozlišča X
          vozP = desnaRotacija(vozP);
        } // else
        // vozlišče je v korenu
        if(vozP.vrniOce() == null) return vozP;
        else vozP = vozP.vrniOce();
      } // while
    } // else
  } // konec lomljenjeDrevesa
```

2.4 Vstavljanje elementa v drevo

Rotacije in lomljenje drevesa so samo pomožne metode, ki jih potrebujemo za uspešno izpeljavo »pravih« operacij, torej tistih, ki jih bodo uporabniki lomljenih dreves uporabljali. Najbolj osnovna je seveda vstavljanje elementa v drevo. Element vedno dodamo kot nov list drevesa, tako, da najprej poiščemo ustrezno mesto vstavljanja. V ta namen se rekurzivno sprehodimo po drevesu. Glavni premislek tega dela postopka je sledeč. Če je element, ki ga vstavljamo, slučajno enak elementu v korenu, smo s postopkom končali. V iskalnem dvojiškem drevesu namreč ni podvojenih elementov, zato elementa ne vstavimo. Če je element manjši od elementa v korenu, postopek rekurzivno nadaljujemo v levem poddrevesu, drugače pa v desnem poddrevesu. Slej ko prej bomo s tem postopkom prišli bodisi do ustreznega mesta za novi list, ali pa do vozlišča drevesa, kjer je že enak element kot element, ki ga želimo vstaviti. Ko smo z vstavljanjem na tak ali drugačen način končali, drevo še zlomimo.

Pri vstavljanju elementa v drevo torej ločimo tri primere:

1. Če je drevo prazno, element vstavimo v koren drevesa in postopek je končan.
2. Če drevo ni prazno in v drevesu ni enakega elementa, kot ga vstavljamo, pridemo do lista drevesa. Takrat naredimo novi objekt (vozlišče) in ga povežemo v drevo. Pokličemo metodo za lomljenje drevesa z kazalcem na vozlišče z dodanim elementom in ga premaknemo v koren drevesa.
3. Element, ki ga vstavljamo, je enak elementu v drevesu. Zato postopek vstavljanja končamo. Pokličemo metodo za lomljenje drevesa s kazalcem na vozlišče z enakim elementom v drevesu in ga premaknemo v koren drevesa.

Oglejmo si zgled. Ta nam bo ilustriral postopek vstavljanja elementa v drevo. Ker pa potem, ko element vstavimo v drevo, vstavljeno vozlišče tudi zlomimo, bo hkrati to še nova ilustracija prej opisanega postopka lomljenje drevesa. V začetku prazno lomljeno drevo bomo zaporedoma vstavljali elemente {*17*, *28*, *5*, *35*, *32*, *30*, *31*, *30*}.

Vstavimo element *17*.

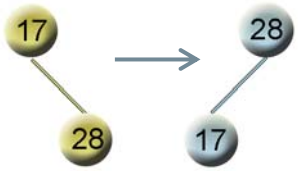
Vozlišče *17* je v korenu drevesa, zato lomljenje drevesa ni potrebno.

Slika 39: Iskalno drevo po vstavljanju elementa *17*.

Naslednji je na vrsti element *28*. Že na prvem koraku vidimo, da moramo vozlišče z elementom *28* vstaviti v desno, v tem trenutku prazno, poddrevo. Dobimo

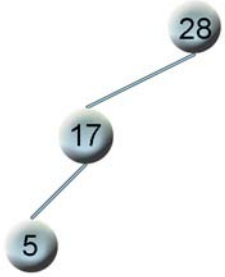
Slika 40: Iskalno drevo po vstavljanju elementa *28*.

Na vrsti je lomljenje. Vozlišče *28* ima za očeta koren (vozlišče *17*) in je njegov desni sin. Za lomljenje uporabimo enojno levo rotacijo in postopek je končan.



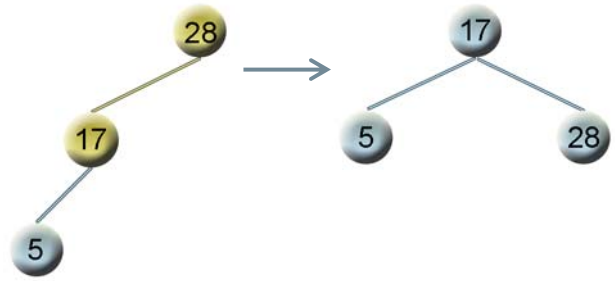
Slika 41: Iskalno drevo pred levo rotacijo z vozlišči *28* in *17* in po njej.

Vstavimo element *5*. Prvo primerjanje pove, da moramo vstavljati v levo poddrevo. Naslednje primerjanje in dejstvo, da levega poddrevesa vozlišča *17* ni, povesta, da moramo vozlišče z elementom *5* vstaviti kot levega sina vozlišča 1*7*. Dobimo



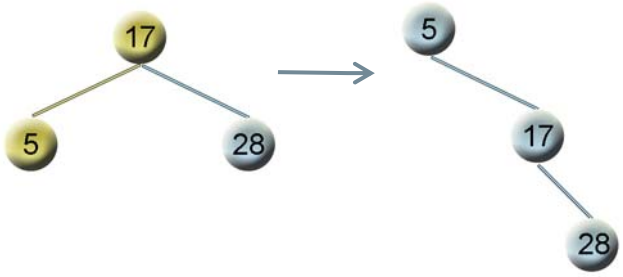
Slika 42: Iskalno drevo po vstavljanju elementa *5*.

Čaka nas lomljenje. Vozlišče z elementom *5* ima za očeta vozlišče *17* in za starega očeta vozlišče *28*. Oba sinova sta leva, zato za lomljenje uporabimo desno-desno rotacijo. Prvo desno rotacijo izvedemo z vozlišči *17* in *28*.



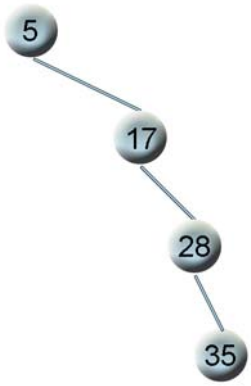
Slika 43: Iskalno drevo pred prvo desno rotacijo z vozlišči *17* in *28* in po njej.

Drugo desno rotacijo izvedemo z vozlišči *5* in *17*. Po tej rotaciji je vozlišče *5* v korenu drevesa in z lomljenjem drevesa končamo.



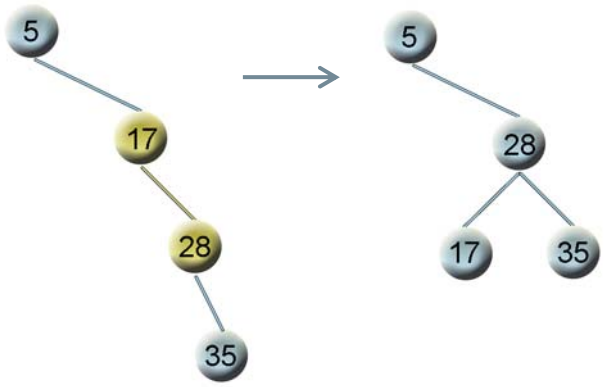
Slika 44: Iskalno drevo pred drugo desno rotacijo z vozlišči *17* in *28* in po končani desno-desni rotaciji.

Vstavimo naslednji element *35*, v lomljeno drevo. Po vstavljanju dobimo



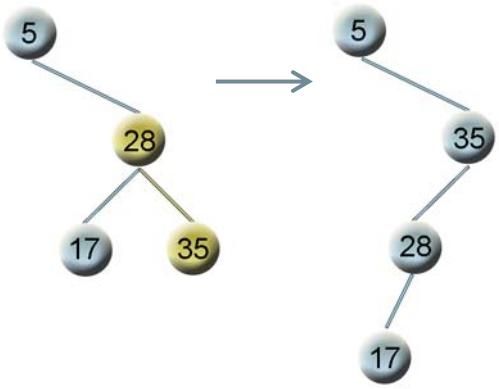
Slika 45: Iskalno drevo po vstavljanju elementa *35*.

Tudi tu oče vozlišča *35* še ni koren. V trojki vozlišče, oče in stari oče nastopajo le desni sinovi, zato izvedemo dvojno in sicer levo-levo rotacijo. Prvo levo rotacijo izvedemo z vozlišči *28* in *17*.



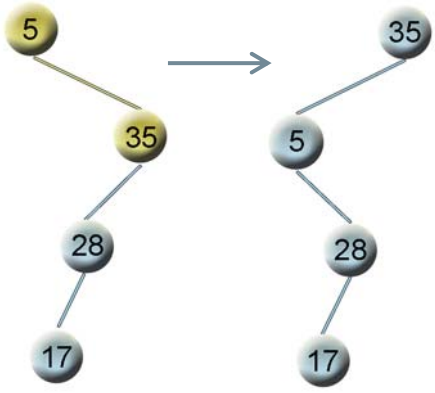
Slika 46: Iskalno drevo pred prvo levo rotacijo z vozlišči *28* in *17* in po njej.

Drugo levo rotacijo izvedemo z vozlišči *35* in *28.*



Slika 47: Iskalno drevo pred drugo levo rotacijo z vozlišči *35* in *28* in po končani levo-levi rotaciji.

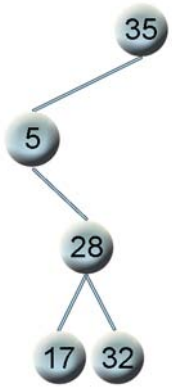
Ker vstavljeno vozlišče *35* še ni v korenu, je pa desni sin korena (vozlišča *5)*, izvedemo enojno levo rotacijo z vozlišči *35* in *5*.



Slika 48: Iskalno drevo pred levo rotacijo z vozlišči *35* in *5* in po njej.

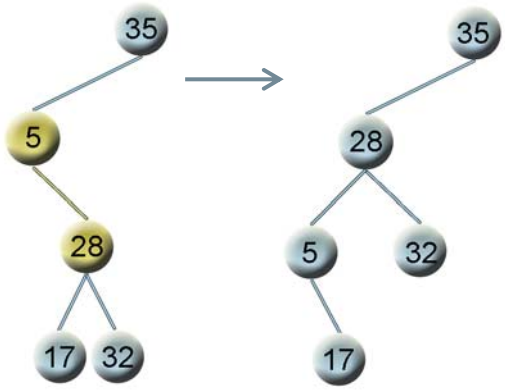
S tem smo vozlišče *35* pripeljali v koren drevesa. Zato končamo z lomljenjem drevesa. Vidimo, da se je drevo izrodilo. Kot smo omenili že prej, pri lomljenih drevesih lahko pride do položaja, ko je drevo izrojeno. Velja pa, da po večjem številu operacij in pri večjem številu elementov v drevesu to v splošnem ne bo izrojeno, ampak bo bolj ali manj uravnoteženo.

Vstavimo element *32*. Prvi klic metode za vstavljanje nas usmeri v levo poddrevo in naslednji v desno poddrevo. Tu pa že ugotovimo, da element *32* spada v trenutno prazno desno poddrevo poddrevesa s korenom *28*. Element *32* bomo torej vstavili v novi list, ki bo desni sin vozlišča *28*.



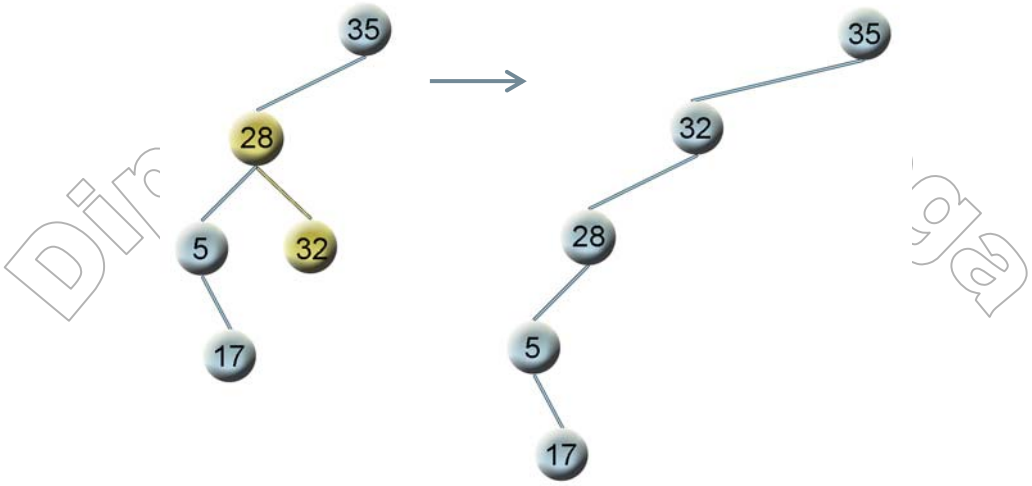
Slika 49: Iskalno drevo po vstavljanju elementa *32*.

Sedaj je na vrsti lomljenje. Vozlišče *32* ima za očeta vozlišče *28* in za starega očeta vozlišče *5*. Vozlišči *32* in 28 sta obe desna sinova svojih očetov, zato izvedemo dvojno levo-levo rotacijo. Prvo levo rotacijo izvedemo z vozlišči *28* in *5*.



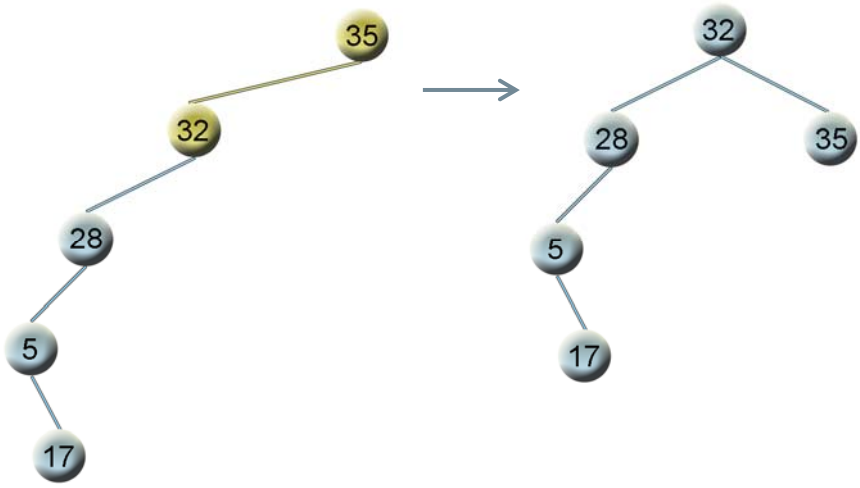
Slika 50: Iskalno drevo pred prvo levo rotacijo z vozlišči *28* in *5* in po njej.

Vozlišče *17* je bilo levi sin vozlišča *28*, sedaj pa je postalo desni sin vozlišča *5*. Drugo levo rotacijo izvedemo z vozlišči *32* in *28*.



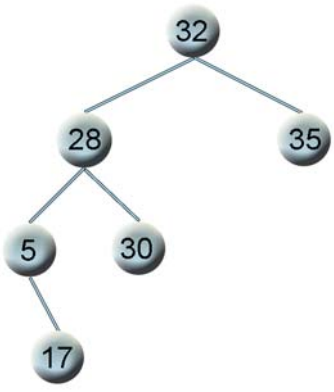
Slika 51: Iskalno drevo pred drugo levo rotacijo z vozlišči *32* in *28* in po končani levo-levi rotaciji.

Ker vstavljeno vozlišče *32* še ni v korenu, nadaljujemo z rotacijami. Vozlišče *32* je levi sin korena *35*, zato ti dve vozlišči uporabimo za enojno desno rotacijo.



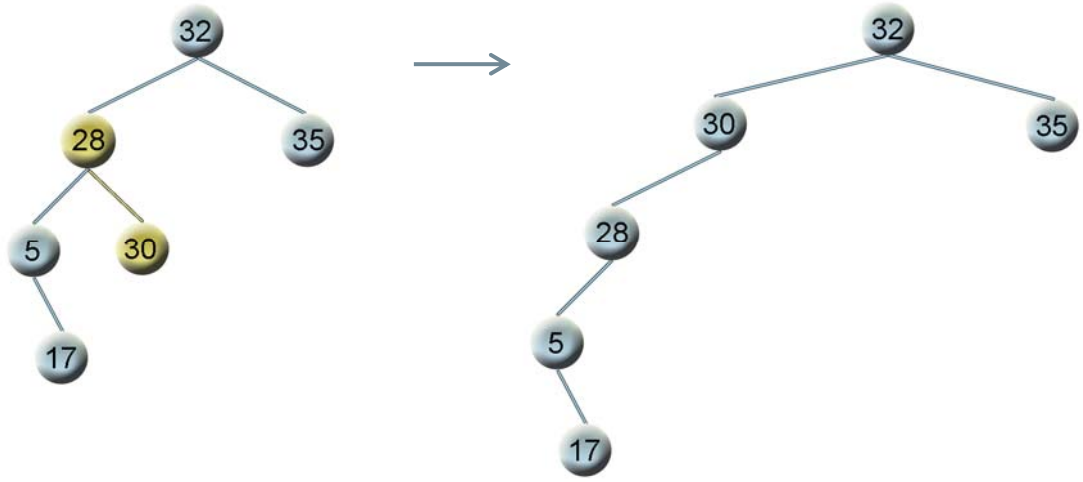
Slika 52: Iskalno drevo pred desno rotacijo z vozlišči *32* in *35* in po njej.

Vstavimo še element *30*. Prvi klic nas pošlje v levo poddrevo, kjer najdemo ustrezno mesto za vozlišče z elementom *30*.



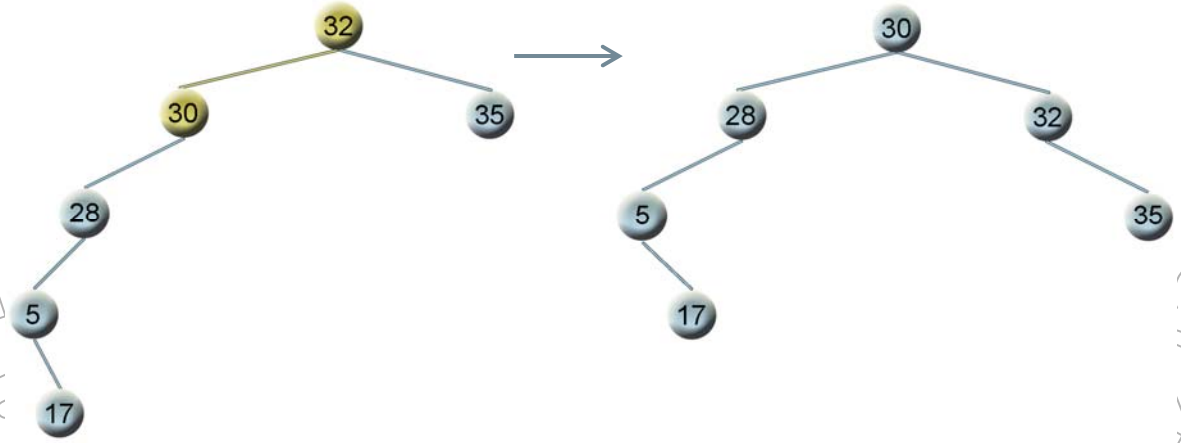
Slika 53: Iskalno drevo po vstavljanju elementa *30*.

Vozlišče *30* ima za očeta vozlišče *28* in za starega očeta vozlišče *32*. Vozlišče *30* je desni sin očeta *28*, ta pa je levi sin starega očeta (vozlišča *32)*. Zato izvedemo dvojno levo-desno rotacijo. Prvo levo rotacijo izvedemo z vozlišči *30* in *28*.



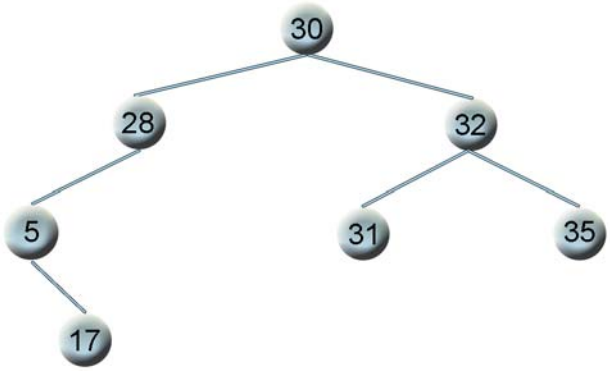
Slika 54: Iskalno drevo pred levo rotacijo z vozlišči *30* in *28* in po njej.

Vidimo, da se je drevo spet začasno izrodilo. Druga rotacija, ki jo še moramo izvesti, bo uravnoteženost popravila. Drugo desno rotacijo izvedemo z vozlišči *30* in *32*.
Vstavljeno vozlišče *30* je po dvojni levo-desni rotaciji že v korenu, zato z lomljenjem drevesa končamo.



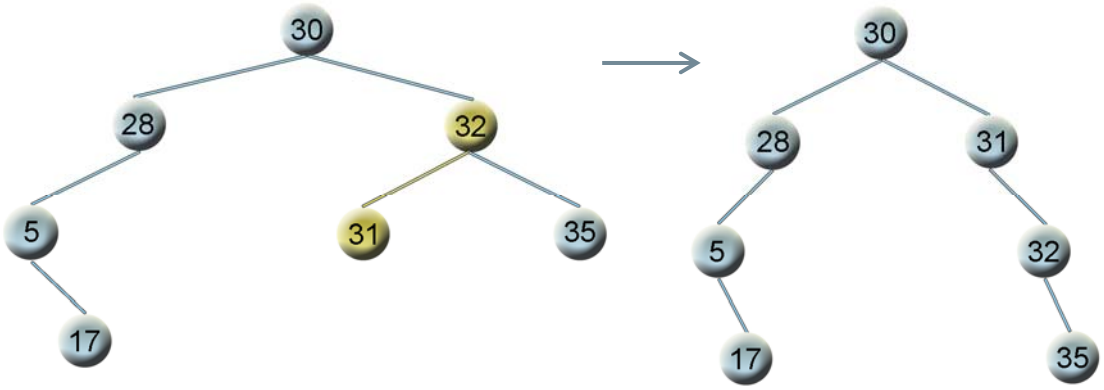
Slika 55: Iskalno drevo pred drugo desno rotacijo z vozlišči *30* in *32* in po končani levo-desni rotaciji.

Vstavimo element *31*. Samo vstavljanje je kaj hitro končano. Prvi klic nas pošlje v desno poddrevo, kjer že dobimo ustrezno mesto za vozlišče z elementom *31*.



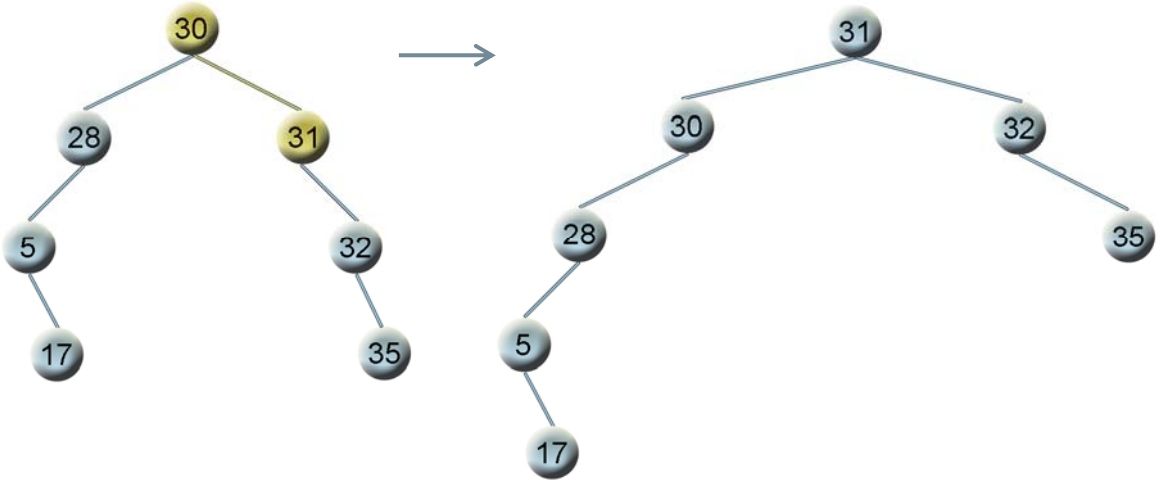
Slika 56: Iskalno drevo po vstavljanju elementa *31*.

Čaka nas še lomljenje. Vozlišče *31* ima očeta vozlišče *32,* in starega očeta vozlišče *30*. Glede na to, kakšni so sinovi, ugotovimo, da je potrebna desno-leva rotacijo. Najprej desna rotacija z vozlišči *31* in *32*.



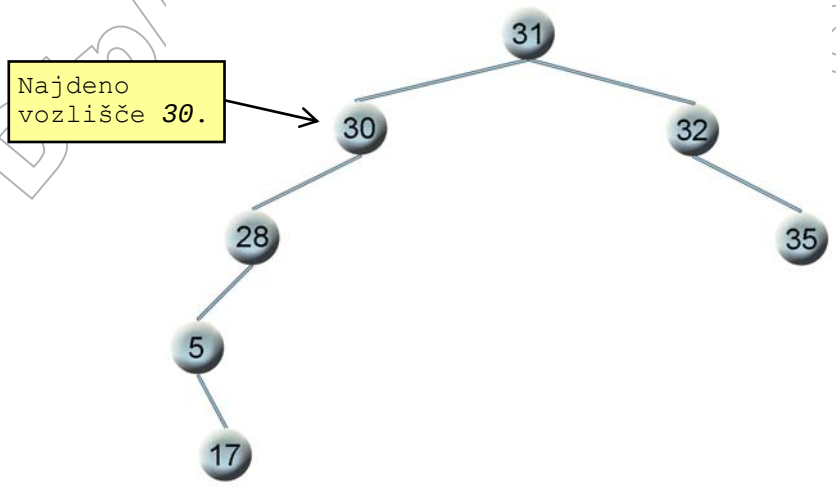
Slika 57: Iskalno drevo pred desno rotacijo z vozlišči *31* in *32* in po njej.

Drugo, tokrat levo, rotacijo pa izvedemo z vozlišči *31* in *30*.
Vstavljeno vozlišče z elementom *31* je v korenu, zato z lomljenjem drevesa končamo.



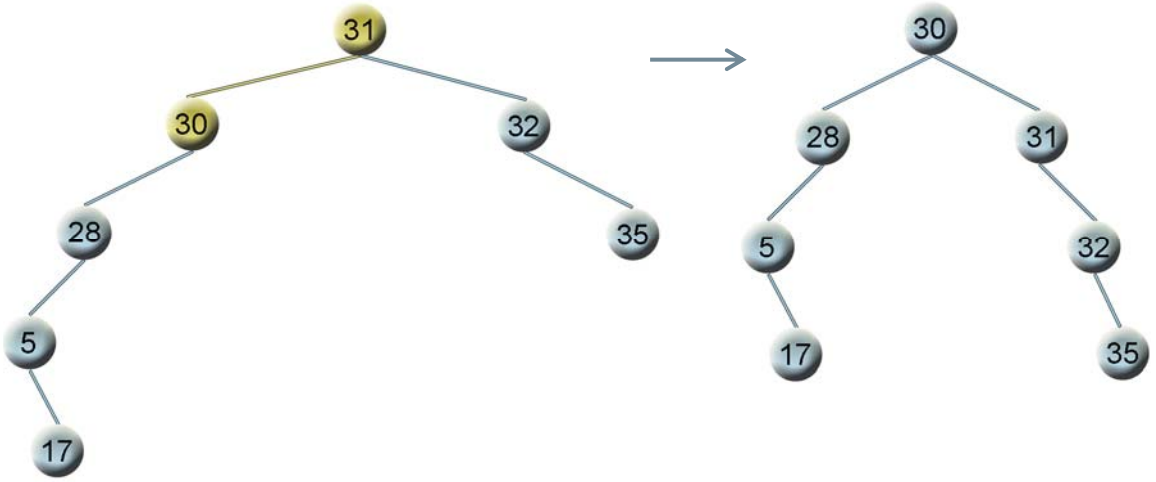
Slika 58: Iskalno drevo pred drugo levo rotacijo z vozlišči *31* in *30* in po končani desno-levi rotaciji.

Vstavimo še element *30*. Pri rekurzivnem iskanju mesta vstavljanja elementa *30* v drevo, ugotovimo, da je vozlišče z elementom *30* že v drevesu. Zato postopek vstavljanja končamo, ne da bi vstavili novi element.



Slika 59: Iskalno drevo po vstavljanju elementa *30*.

Čeprav vozlišča z elementom *30* nismo vstavili, drevo kljub temu lomimo. Lomljenje drevesa izvedemo z vozliščem *30*, ki ga premaknemo v koren drevesa. Vozlišče *30* ima za očeta koren (vozlišče *31*). Je levi sin korena, zato izvedemo le enojno desno rotacijo.



Slika 60: Iskalno drevo pred desno rotacijo z vozliščema *30* in *31*in po njej.

Vozlišče *30* postane koren drevesa. Tudi tu vidimo, da drevo ni ravno idealno uravnoteženo. A še enkrat velja pripomniti, da je lomljeno drevo lahko začasno izrojeno, ampak dolgoročno se zaradi rotacij slej ko prej uravnoteži.

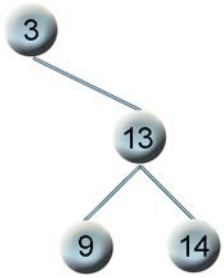
2.4.1 Metoda za vstavljanje

Za vstavljanje elementa sestavimo dve metodi: javno `vstavi(int x)` in privatno `vstavi(int x, VozelLomDrevesa r)`. Prva vstavi vozlišče z elementom v lomljeno drevo in drevo preuredi tako, da je vstavljeni element v korenskem vozlišču. Če drevo že od prej vsebuje vozlišče z enakim elementom, metoda le preuredi drevo na omenjeni način.

Za samo vstavljanje uporabljamo privatno metodo *vstavi(int x, VozelLomDrevesa r)*. Metoda dobi kazalec na koren (pod)drevesa (*VozelLomDrevesa r*) in rekurzivno poišče mesto vstavljanja vozlišča z elementom. Nato vstavi vozlišče z elementom v list drevesa (če v drevesu še ni vozlišča z enakim elementom) in vrne kazalec na vstavljeno vozlišče. Če se vozlišče z elementom že nahaja v drevesu, vrne kazalec na vozlišče s tem elementom.

Metoda *vstavi(int x)* po vstavljanju pokliče še metodo za lomljenje drevesa *lomljenjeDrevesa(VozelLomDrevesa t)*, kjer je (*VozelLomDrevesa t*) kazalec, ki ga je vrnila metoda *vstavi(int x, VozelLomDrevesa r)*.

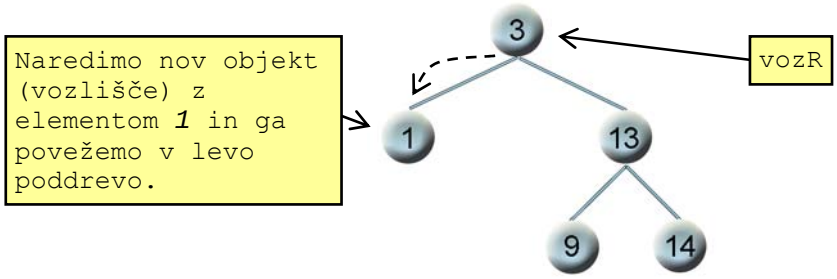
Metodo za vstavljanje v drevo si oglejmo podrobneje. Vmes jo bomo prekinjali s slikami stanja pri izvajanju metode v konkretnem primeru. V iskalno drevo na sliki 61 bomo vstavili element *1* (slika 62) in *14* (slika 63). Da bomo lažje sledili metodi, si oglejmo sliko prvotnega iskalnega drevesa.



Slika 61: Iskalno drevo.

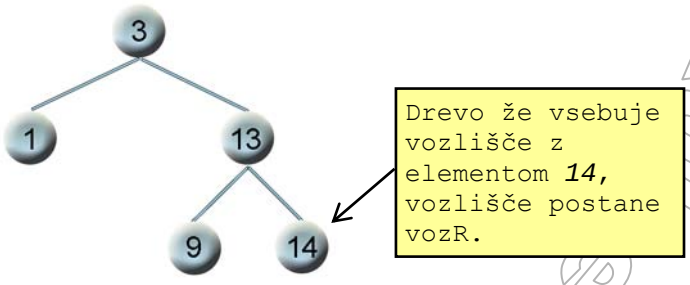
```
/**
 * Vstavi element v drevo, z lomljenjem poskrbimo, da vozlišče z
 * vstavljenim elementom postane koren.
 *
 * @param  x  je element, ki ga vstavljamo.
 * @return  <code>void</code>.
 */
public void vstavi(int x){
  // klic metode za vstavljanje
  VozelLomDrevesa vozR = vstavi(x, (VozelLomDrevesa)koren);
  // lomimo drevo z vstavljenim vozlom
  koren = lomljenjeDrevesa(vozR);
}

/**
 * Poišče mesto vstavljanja v drevesu, kamor vstavimo element in
 * vstavi vozlišče z elementom, vrne kazalec na vstavljeno vozlišče.
 * Če je element že v drevesu, vrne kazalec na vozlišče z tem
 * elementom.
 *
 * @param  x  je element, ki ga vstavljamo.
 * @param  vozR  je koren drevesa.
 * @return  <code> VozelLomDrevesa </code>, kazalec na
 * vstavljeno vozlišče oz. vozlišče z elementom v drevesu.
 */
private VozelLomDrevesa vstavi(int x, VozelLomDrevesa vozR){
```

```
// če je drevo prazno
if(vozR == null){
  // vstavimo v koren
  koren = new VozelLomDrevesa(x);
  return (VozelLomDrevesa)koren;
} // if(vozR == null)
// vstavljanje v levo poddrevo
else if(vozR.vrniElement() > x){
  // če levega sina ni
  if(vozR.vrniLeviSin() == null){
    // vstavimo v levo poddrevo
    vozR.nastaviLeviSin(new VozelLomDrevesa(x, vozR));
```



Slika 62: Vozlišče z elementom *1* vstavimo, za levega sina korenu *3*.

```
    return (VozelLomDrevesa)vozR.vrniLeviSin();
  }
  else return vstavi(x, (VozelLomDrevesa)vozR.vrniLeviSin());
} // else if(vozR.vrniElement() > x)
// vstavljanje v desno poddrevo
else if(vozR.vrniElement() < x){
  // če desnega sina ni
  if(vozR.vrniDesniSin() == null){
    // vstavimo v desno poddrevo
    vozR.nastaviDesniSin(new VozelLomDrevesa(x, vozR));
    return (VozelLomDrevesa)vozR.vrniDesniSin();
  }
  else return vstavi(x, (VozelLomDrevesa)vozR.vrniDesniSin());
} // else if(vozR.vrniElement() < x)
// vozlišče z enakim elementom
else return vozR;
```



Slika 63: V iskalnem drevesu smo našli vozlišče z elementom *14*.

```
} // vstavi
```

2.5 Iskanje elementa v drevesu

To je verjetno metoda, ki je med metodami nad lomljenimi drevesi največkrat uporabljena. Vrne nam `true`, če je iskalni element v drevesu, sicer pa `false`.

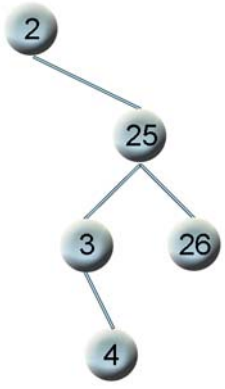
Postopek sam je enostaven. Začnemo pri korenu in se pomikamo proti listom. Če je element, ki ga iščemo, slučajno enak elementu v korenu, smo s postopkom končali. Če pa je element manjši od elementa v korenu, postopek rekurzivno nadaljujemo v levem poddrevesu, drugače pa v desnem poddrevesu. Slej ko prej bomo s tem postopkom prišli bodisi do vozlišča z enakim elementom, oz. do ugotovitve, da iskanega elementa ni v drevesu. Do sem opisani postopek je povsem enak iskanju v poljubnem dvojiškem iskalnem drevesu. Od običajnega iskanja pa se loči po postopku, ki ga izvedemo, ko že imamo odgovor. Vozlišče z iskanim elementom oz. zadnje vozlišče na poti iskanja elementa namreč premaknemo v koren drevesa.

Pri iskanju elementa v drevesu ločimo tri primere:

1. Drevo je prazno.
2. V drevesu smo našli vozlišče z iskanim elementom. Pokličemo metodo za lomljenje drevesa z kazalcem na najdeno vozlišče in ga premaknemo v koren drevesa.
3. Pridemo do lista drevesa. Iskanega elementa ni v drevesu. Pokličemo metodo za lomljenje drevesa s kazalcem na zadnje vozlišče na poti iskanja elementa. To vozlišče torej premaknemo v koren drevesa.

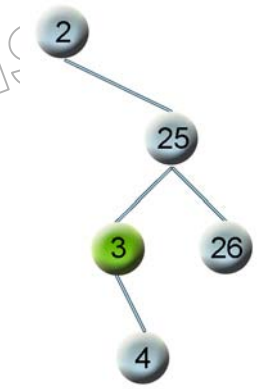
Na slikah zeleno obarvano vozlišče označuje najdeno vozlišče oz. če iskanega elementa ni v drevesu, zadnje vozlišče na poti iskanja.

Oglejmo si zgled. Dano je iskalno drevo



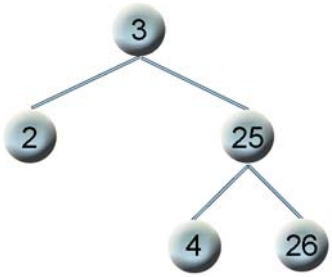
Slika 64: Iskalno drevo.

Iščemo vozlišče z elementom *3*. Ker je element (*3*) večji od elementa v korenu (*2*), gremo v desno poddrevo. Element tu (*25*) je večji od iskanega, zato nadaljujemo iskanje v levem poddrevesu. Tu v korenu najdemo vozlišče z elementom *3*.



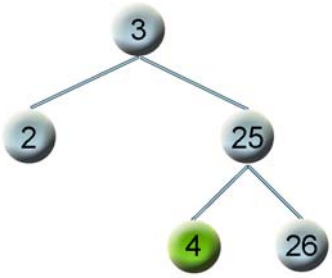
Slika 65: Iskalno drevo z najdenim vozliščem elementa *3*.

Pomožna metoda za iskanje nam vrne kazalec na vozlišče *3*. Sedaj nam ostane le še, da s pomočjo lomljenja najdeno vozlišče *3* premaknemo v koren drevesa. Dobimo naslednje lomljeno drevo



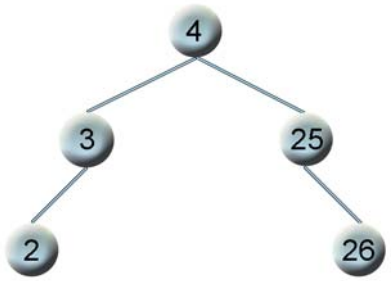
Slika 66: Iskalno drevo po končani operaciji iskanja vozlišča *3*.

Oglejmo si še en zgled. Izhajajmo kar iz pravkar dobljenega lomljenega drevesa in v njem poiščimo vozlišče z elementom *5*. Tega elementa ni v drevesu. Pomožna metoda za iskanje nam vrne kazalec na vozlišče z elementom *4*, ki je bilo zadnje vozlišče na poti iskanja.



Slika 67: Iskalno drevo z zadnjim vozliščem na poti iskanja vozlišča *5*.

Tega premaknemo v koren drevesa.

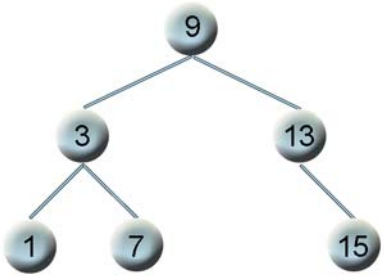


Slika 68: Iskalno drevo po končani operaciji iskanja vozlišča *5*.

2.5.1 Metodi za iskanje

Za iskanje elementa sestavimo dve metodi, javno `isci(int x)` in privatno `isci(int x, VozelLomDrevesa r)`. Metoda `isci(int x, VozelLomDrevesa r)` poišče vozlišče z elementom v drevesu (`r` je koren drevesa, v katerem iščemo `x`) in vrne kazalec na iskano vozlišče z elementom. Če pa vozlišča z elementom ni v drevesu, vrne kazalec na zadnje vozlišče na poti iskanja. Metoda `isci(int x)` najprej kliče metodo `isci(int x, VozelLomDrevesa r)`. Od metode dobi kazalec na vozlišče (bodisi tisto z elementom, bodisi zadnje vozlišče na poti iskanja). Nato metoda `isci(int x)` pokliče metodo za lomljenje drevesa `lomljenjeDrevesa(VozelLomDrevesa t)`. Na koncu metoda `isci(int x)` kot rezultat vrne `true`, če element je v drevesu, sicer pa `false`.

Obe metodi za iskanje elementa v drevesu si oglejmo podrobneje. Da bo bolj razvidno, katero vozlišče vrne pomožna metoda, bomo kodo prekinili s slikami stanja pri izvajanju metode. V iskalnem drevesu (slika 69) bomo iskali element *12* (slika 70) oziroma *15* (slika 71).

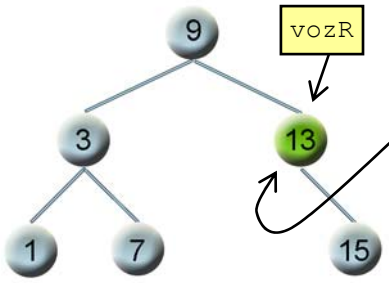


Slika 69: Iskalno drevo.

```
/**
 * Poišče vozlišče z elementom, prestavi ga v koren drevesa. Če
 * vozlišča z elementom ni v drevesu, zadnje vozlišče na poti
 * iskanja prestavi v koren drevesa.
 *
 * @param  x  je element, ki ga iščemo.
 * @return  <code> true </code>, če je element v drevesu.
 */
public boolean isci(int x){

   // drevo je prazno
   if (koren == null) return false;
   // poiščemo vozlišče z elementom x
```

```
    VozelLomDrevesa vozR = isci(x, (VozelLomDrevesa)koren);
    // lomimo drevo z vozliščem x, oz. zadnjim vozliščem na
    // iskani poti
    koren = lomljenjeDrevesa(vozR);
    // elementa ni
    if(koren.vrniElement() != x) return false;
    // elementa je v drevesu
    else return true;
  } // isci

  /**
   * Poišče vozlišče z elementom, vrne kazalec na najdeno vozlišče. Če
   * vozlišča z elementom ni v drevesu, vrne kazalec na zadnje
   * vozlišče na poti iskanja.
   *
   * @param  x  je element, ki ga iščemo.
   * @param  vozR  je koren drevesa.
   * @return  <code> VozelLomDrevesa </code>,
   * kazalec na iskano oz. zadnje vozlišče.
   */
  private VozelLomDrevesa isci(int x, VozelLomDrevesa vozR){

    // drevo je prazno
    if(vozR == null) return null;
    // element je v korenu
    else if(vozR.vrniElement() == x) return vozR;
    else{
      VozelLomDrevesa pomozni;
      // iskanje v levem poddrevesu
      if(vozR.vrniElement() > x)
        pomozni = isci(x, (VozelLomDrevesa)vozR.vrniLeviSin());
      // iskanje v desnem poddrevesu
      else pomozni = isci(x, (VozelLomDrevesa)vozR.vrniDesniSin());
      // vozlišča z elementom x ni v drevesu, vrnemo zadnje vozlišče
      // na iskalni poti
      if(pomozni == null) return vozR;
```

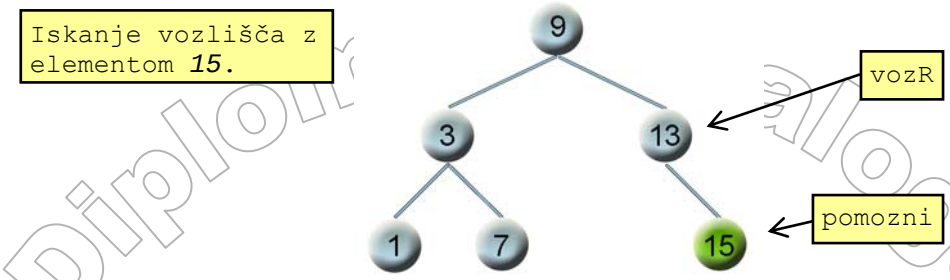
Iskanje vozlišča z elementom ***12***.



Vozlišče ***12*** bi bilo v levem poddrevesu vozlišča ***13***. Ker pa levo poddrevo vozlišča ***13*** ne obstaja, je pomozni == null.

Slika 70: Iskalno drevo z zadnjim vozliščem na poti iskanja vozlišča *12*.

```
    // iskani element
    else return pomozni;
```



Slika 71: Iskalno drevo z najdenim vozliščem *15*.

```
    } // else
}
```

2.6 Brisanje elementa iz drevesa

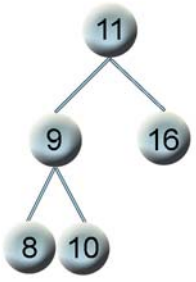
Pri brisanju elementa, ki ga želimo odstraniti iz lomljenega drevesa, najprej poiščemo vozlišče, ki ta element vsebuje. Najdeno vozlišče z elementom oz. zadnje vozlišče na poti iskanja s pomočjo lomljenja premaknemo v koren drevesa. Nato moramo v primeru, da smo vozlišče našli, korena izbrisati iz drevesa. To opravimo s postopki opisanim v nadaljevanju.

Rdeče vozlišče naj označuje vozlišče za brisanje, vijolično pa maksimalno vozlišče levega poddrevesa.

Pri brisanju elementa v drevesu ločimo tri primere:

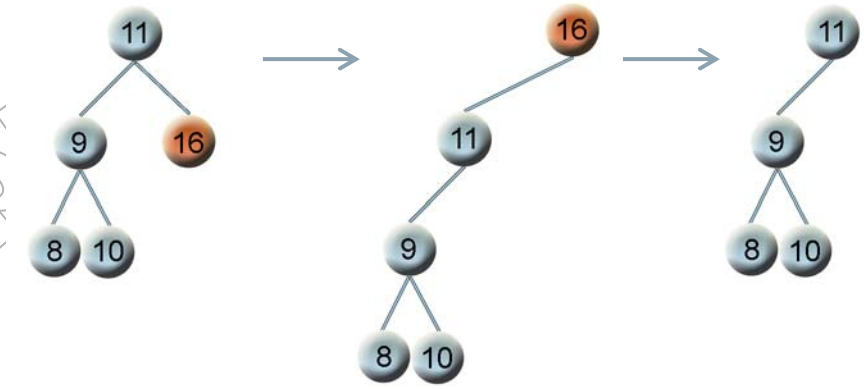
1. Drevo je prazno.
2. V drevesu smo našli vozlišče z elementom za brisanje. Pokličemo metodo za lomljenje drevesa s kazalcem na to vozlišče in ga premaknemo v koren drevesa. Ločimo dva primera brisanja korena.
   a. Če je levo poddrevo korena prazno, postane desni sin koren drevesa in element je izbrisan. Če pa je prazno desno poddrevo, vlogo korena dobi levi sin.

      Kot primer si oglejmo, če iz drevesa izbrišemo element *16*.



Slika 72: Iskalno drevo pred brisanjem vozlišča z elementom *16*.

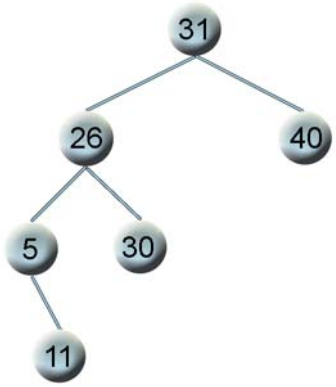
      Našli smo vozlišče z elementom *16* in ga z lomljenjem prestavili v koren drevesa. Ker je desno poddrevo prazno, levi sin (vozlišče *11*), postane novi koren drevesa. S tem je vozlišče *16* izbrisano iz drevesa.



Slika 73: Iskalno drevo pred brisanjem in po brisanju vozlišča 16.

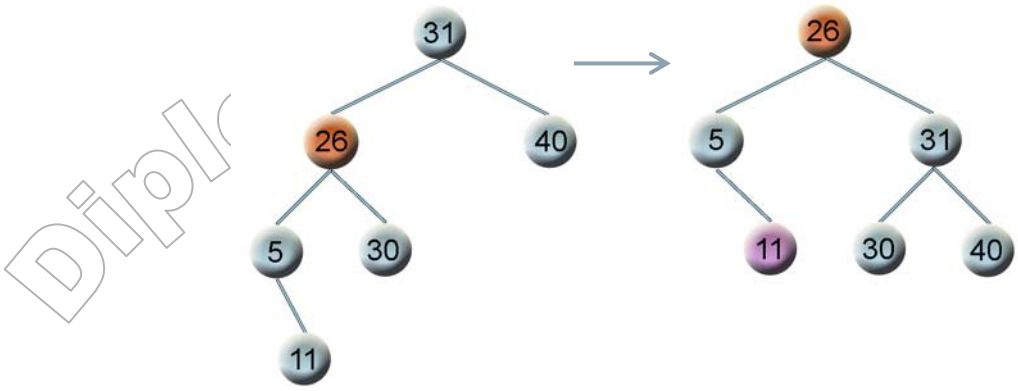
b. Če pa ni ne levo ne desno poddrevo korena prazno, v levem poddrevesu poiščemo vozlišče z maksimalnim elementom. Tega z lomljenjem premaknemo v koren levega poddrevesa korena. V korenu drevesa imamo sedaj vozlišče, ki ga želimo izbrisati, njegov levi sin pa je vozlišče z maksimalnim elementom levega poddrevesa. V tem levem poddrevesu desnega poddrevesa ni, saj je v korenu maksimalni element. Zato koren drevesa izbrišemo tako, da desno poddrevo korena postane desno poddrevo levega sina korena drevesa, levi sin korena pa postane novi koren drevesa. Vozlišče z maksimalnim elementom levega poddrevesa poiščemo tako, da v levem poddrevesu iščemo element, ki ga želimo izbrisati. Ker je v korenu, je seveda večji od vseh elementov v levem poddrevesu. Zato ga metoda za iskanje ne bo našla. Zadnje vozlišče na poti iskanja bo seveda maksimalni element tega poddrevesa in se bo z lomljenjem znašel v korenu poddrevesa.

Kot primer iz drevesa izbrišimo element *26*.



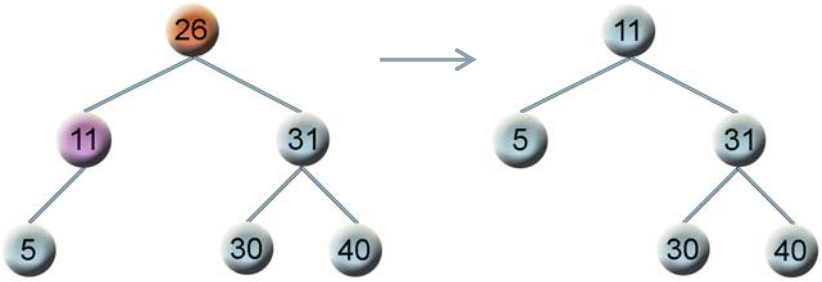
Slika 74: Iskalno drevo pred brisanjem vozlišča z elementom *26*.

Poiščemo vozlišče z elementom *26* in ga prestavimo v koren drevesa. Ker nobeno od poddreves ni prazno, v levem poddrevesu poiščemo vozlišče z maksimalnim elementom. To je vozlišče z elementom *11*.



Slika 75: Iskalno drevo pred brisanjem in med brisanjem vozlišča *26*.

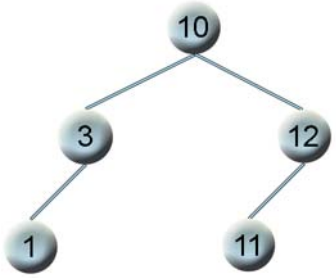
Vozlišče z maksimalnim elementom *11* prestavimo v koren levega poddrevesa. Še enkrat opozorimo na to, da po prestavljanju koren levega poddrevesa zagotovo nima desnega sina. Zato desni sin korena, vozlišče *31*, lahko postane desni sin vozlišča *11*. Vozlišče *11* sedaj postane novi koren drevesa. S tem je vozlišče *26* izbrisano iz drevesa.



Slika 76: Iskalno drevo med brisanjem in po brisanjem vozlišča *26*.

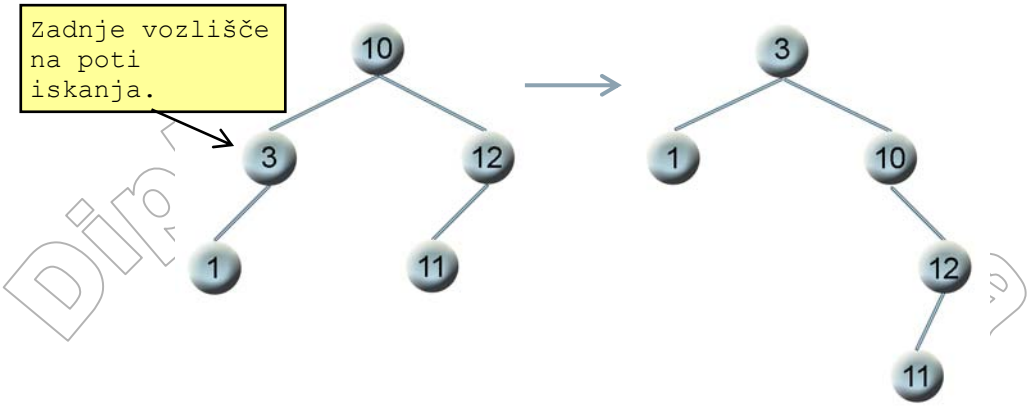
3. Iz drevesa želimo izbrisati element, ki ga ni v drevesu. Ko z iskanjem to ugotovimo, pokličemo metodo za lomljenje drevesa s kazalcem na zadnje vozlišče na poti iskanja elementa in tega premaknemo v koren drevesa.

   Kot primer si oglejmo, če želimo iz drevesa izbrisati element *4*.



Slika 77: Iskalno drevo pred brisanjem vozlišča z elementom *4*.

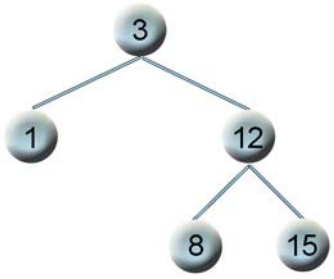
Vozlišča z elementom *4* ni v drevesu. Zato lomimo drevo z zadnjim vozliščem na poti iskanja – torej z vozliščem *3*. Ko je vozlišče *3* v korenu, končamo.



Slika 78: Iskalno drevo pred brisanjem in po brisanju vozlišča *4*.

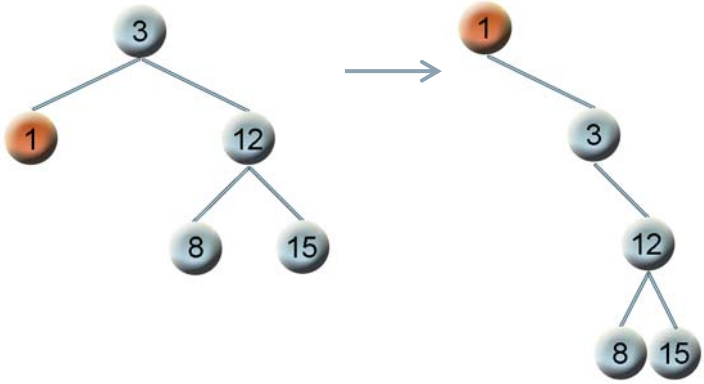
Da bomo dobili boljši občutek, kako poteka brisanje vozlišč iz lomljenega drevesa, si oglejmo še en zgled.

Dano je iskalno drevo



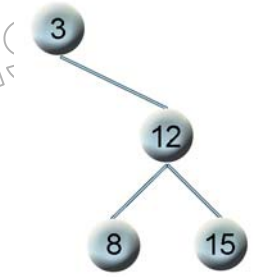
Slika 79: Iskalno drevo.

Izbrišimo iz drevesa element *1*. Poiščemo vozlišče z elementom *1* in ga premaknemo v koren drevesa.



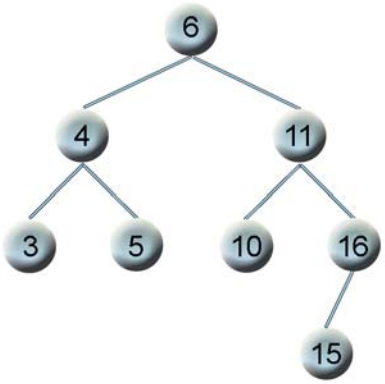
Slika 80: Iskalno drevo pred in po premiku vozlišča *1* v koren drevesa.

Vozlišče *1* je v korenu drevesa. Ker je levo poddrevo prazno, postane njegov desni sin koren drevesa in vozlišče *1* je izbrisano iz drevesa.



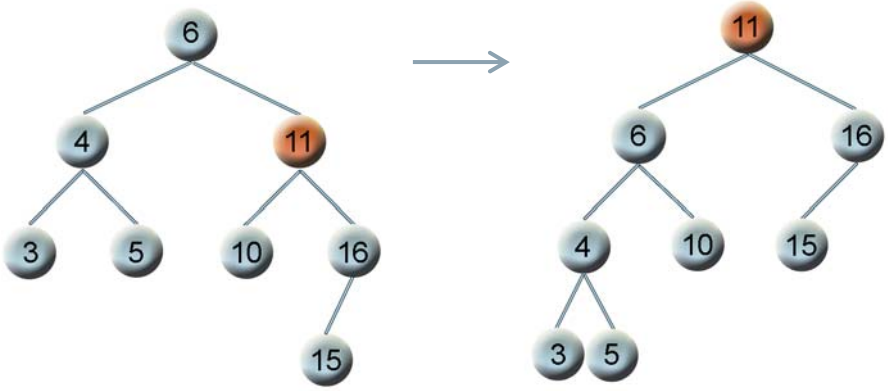
Slika 81: Iskalno drevo po brisanju vozlišča *1* iz drevesa.

Oglejmo si še en zgled. Dano naj bo iskalno drevo



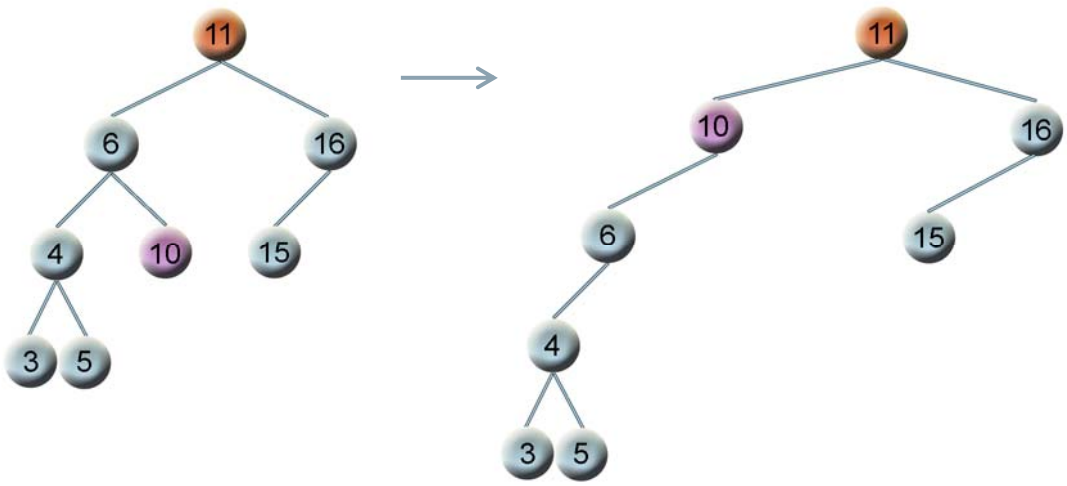
Slika 82: Iskalno drevo.

Izbrišimo element *11*.
Poiščemo vozlišče z elementom *11* in ga premaknemo v koren drevesa.



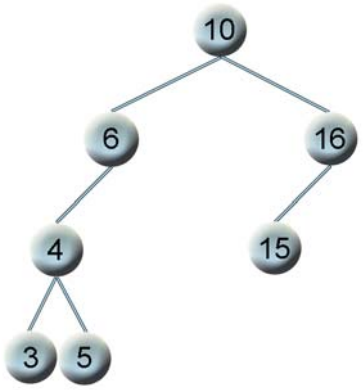
Slika 83: Iskalno drevo pred in po premiku vozlišča *11* v koren drevesa.

Vozlišče *11* je v korenu drevesa. Ker levo poddrevo ni prazno, poiščemo maksimalni element v levem poddrevesu. To je vozlišče *10*. Z lomljenjem ga prestavimo v koren levega poddrevesa.



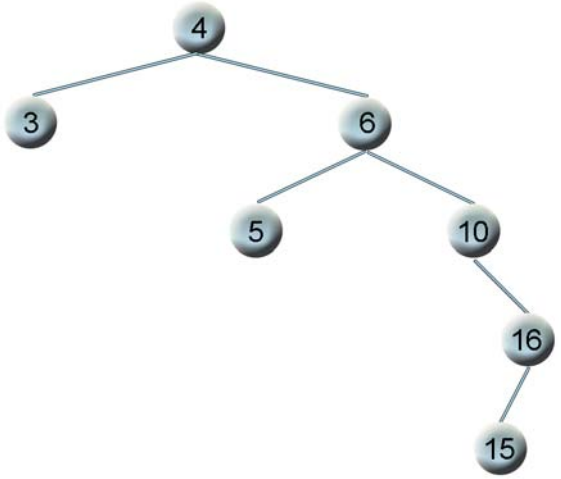
Slika 84: Vozlišče z maksimalnim elementom *10* levega poddrevesa je v korenu levega poddrevesa.

Vozlišče *11*, ki ga bomo izbrisali iz drevesa, je v korenu in maksimalno vozlišče levega poddrevesa je njegov levi sin. Desno poddrevo s korenom vozliščem *16*, postane desno poddrevo vozlišča *10*. Koren levega poddrevesa (vozlišče *10*), postane novi koren drevesa in vozlišče *11* je izbrisano iz drevesa.



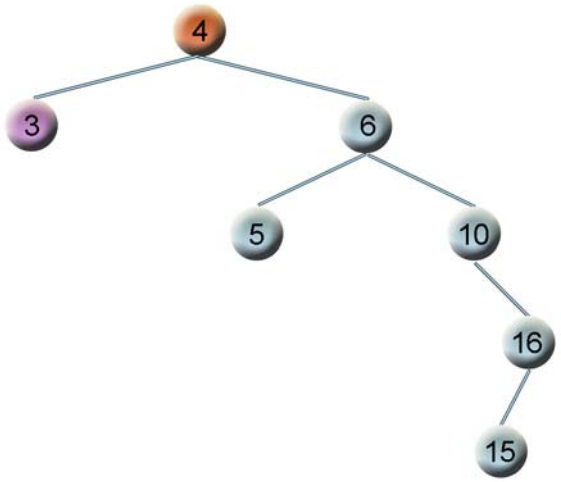
Slika 85: Iskalno drevo po končanem brisanju vozlišča *11* iz drevesa.

Še zadnji zgled brisanja. Dano je iskalno drevo



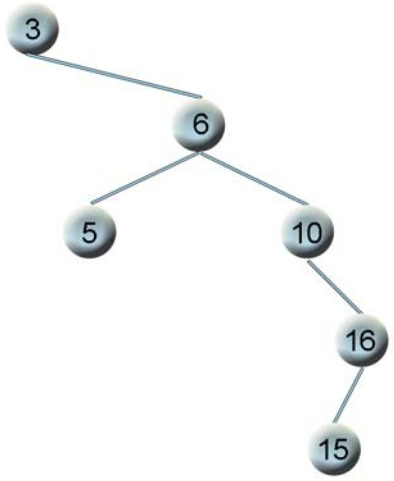
Slika 86: Iskalno drevo.

Izbrišimo iz drevesa element *4*.
Vozlišče z elementom 4, ki ga bomo izbrisali iz drevesa, je že v korenu drevesa. Ker ni ne levo ne desno poddrevo prazno, v levem poddrevesu poiščemo maksimalni element. Ker je v levem poddrevesu samo vozlišče 3, takoj končamo iskanje. Lomljenje, s katerim bi maksimalni element spravili v koren poddrevesa, tu ne bo potrebno.



Slika 87: Iskalno drevo med brisanjem vozlišča *4*.

Sedaj le še desno poddrevo pripnemo kot desnega sina vozlišču *3* in brisanje je končano.

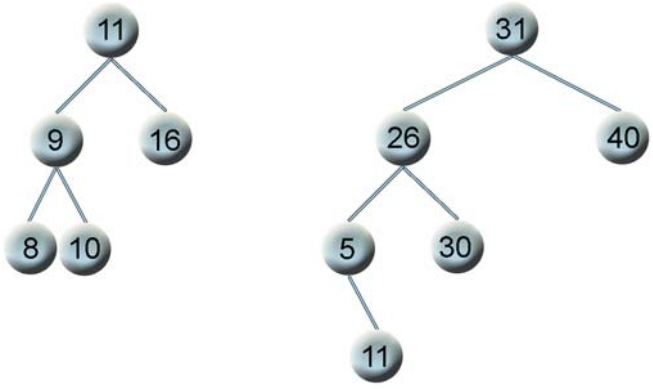


Slika 88 Iskalno drevo po končanem brisanju vozlišča *4* iz drevesa.

2.6.1 Metoda za brisanje

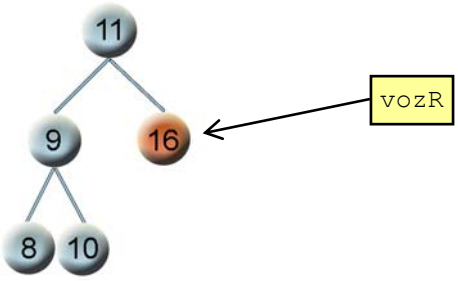
Za brisanje elementa iz drevesa sestavimo metodo `brisi(int x)`. Metoda `brisi(int x)` sprejme element, ki ga želimo brisati iz drevesa. Pri izvedbi metode bomo potrebovali že prej opisani metodi `isci(int x, VozelLomDrevesa r)` in `lomljenjeDrevesa(VozelLomDrevesa t)`. Spomnimo se, da prva poišče vozlišče z elementom (`int x`) v poddrevesu s korenom (`VozelLomDrevesa r`) in vrne kazalec na iskano vozlišče z elementom, oziroma, če elementa ni v poddrevesu, vrne kazalec na

zadnje vozlišče na poti iskanja. Druga pa zlomi ustrezno drevo tako, da dano vozlišče postane koren drevesa.
Metodo za brisanje elementa si oglejmo podrobneje. Tudi to bomo vmes prekinjali s slikami stanja. Uporabili bomo dve drevesi, pri katerih pridemo do primerov 2a in 2b.



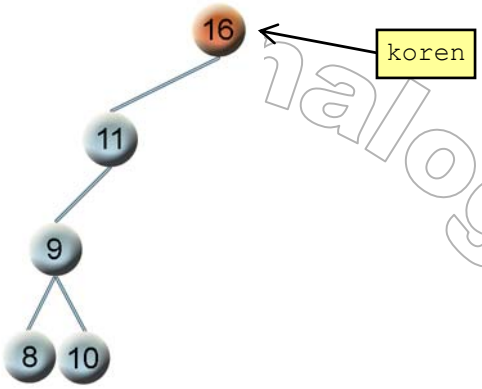
Slika 89: Iskalni drevesi. Iz prvega bomo izbrisali element *16* in iz drugega element *26*.

```
/**
 * Poišče vozlišče z elementom, izbriše ga iz drevesa in preuredi
 * drevo. Če vozlišča z elementom ni v drevesu, zadnje vozlišče na
 * poti iskanja prestavi v koren drevesa.
 *
 * @param  x  je element, ki ga brišemo.
 * @return  <code> void </code>.
 */
public void brisi(int x){

  if(koren == null) return;
  // poiščemo vozlišče za brisanje
  VozelLomDrevesa vozR = isci(x, (VozelLomDrevesa)koren);
```



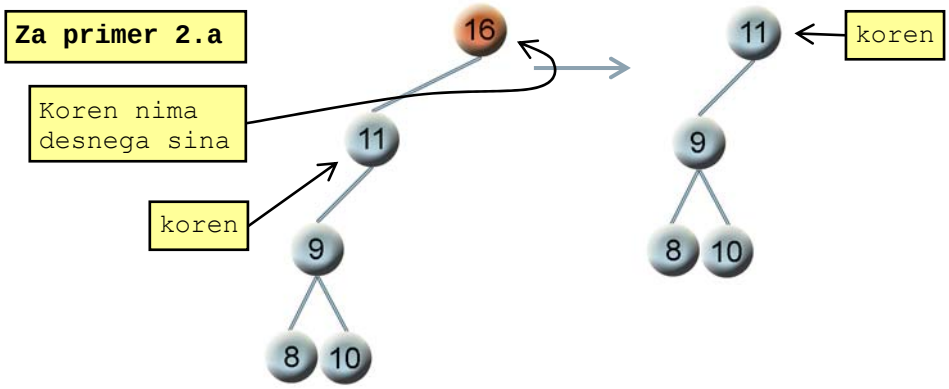
Slika 90: Iskalno drevo z najdenim vozliščem *16*.

```
  // vozlišče postavimo v koren drevesa
  koren = lomljenjeDrevesa(vozR);
```



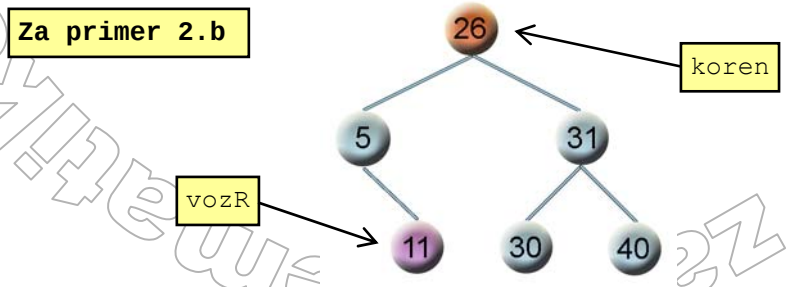
Slika 91: Vozlišče *16* za brisanje je v korenu drevesa.

```
// našli smo element za brisanje
if(koren.vrniElement() == x){
  // če je eno od poddreves prazno
  if((koren.vrniLeviSin() == null) ||
     (koren.vrniDesniSin() = null)){
    // če je levo poddrevo prazno, povežemo desno poddrevo v koren
    if(koren.vrniLeviSin() == null) koren = koren.vrniDesniSin();
    // če je desno poddrevo prazno, povežemo levo poddrevo v koren
    else koren = koren.vrniLeviSin();
```



Slika 92: Iskalno drevo po končanem brisanju vozlišča *16* iz drevesa.

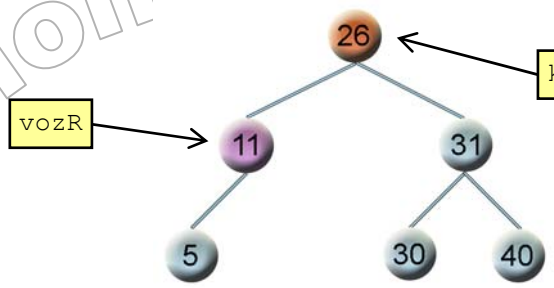
```
  } // if((koren.vrniLeviSin() == null)
  // obe poddrevesi obstajata
  else{
    // max element levega poddrevesa
    vozR = isci(x, (VozelLomDrevesa)koren.vrniLeviSin());
```



Slika 93: Iskalno drevo z najdenim vozliščem maksimalnega elementa *11* levega poddrevesa.

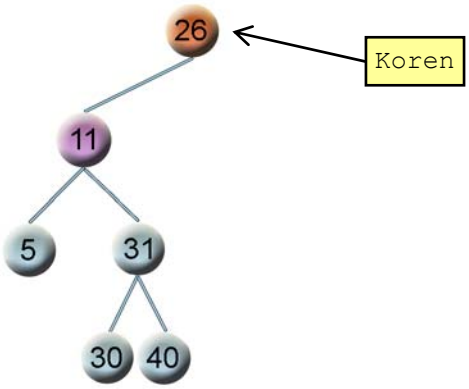
```
    // prekinemo povezavo zaradi lomljenje drevesa
    ((VozelLomDrevesa)koren.vrniLeviSin()).nastaviOce(null);
```

```
// maksimalno vozlišče levega poddrevesa premaknemo v koren
// levega poddrevesa
koren.nastaviLeviSin(lomljenjeDrevesa(vozR));
```



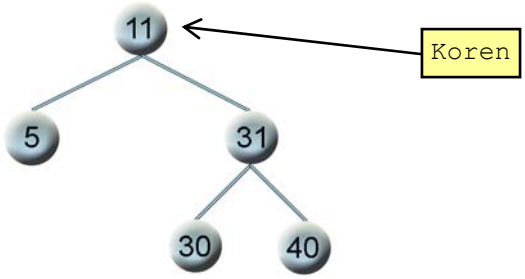
Slika 94: Vozlišče z maksimalnim elementom *11* je v korenu levega poddrevesa.

```
// korenu desnega poddrevesa nastavimo za očeta koren
// levega poddrevesa
((VozelLomDrevesa)koren.vrniDesniSin()).nastaviOce
((VozelLomDrevesa)koren.vrniLeviSin());
// korenu levega poddrevesa nastavimo desnega sina
koren.vrniLeviSin().nastaviDesniSin(koren.vrniDesniSin());
```



Slika 95: Desno poddrevo je postalo desno poddrevo korena levega poddrevesa.

```
// koren levega poddrevesa postane koren
koren = koren.vrniLeviSin();
```




```
    } // else
  } // if(koren.vrniElement() == x)
  // korenu nastavimo očeta na null
  if(koren != null) ((VozelLomDrevesa)koren).nastaviOce(null);
} // brisi
```

2.7 Časovna zahtevnost operacij

Lomljena drevesa se spreminjajo (lomijo) ob vsaki operaciji. Tako je analiza učinkovitosti operacij na lomljenih drevesih otežena. Zato bomo v tem razdelku navedli le nekaj splošnih ugotovitev.
V najslabšem primeru je časovna zahtevnost posamezne operacije vedno reda $O(n)$, kjer je n število elementov v drevesu, saj je lomljeno drevo lahko tudi izrojeno. Vendar, če imamo k zaporednih operacij, je pričakovana časovna zahtevnost izvajanja celotnega zaporedja operacij reda $O(k \log(n))$.

2.7.1 Lomljene drevesa in rotacije

Za lomljenje drevesa uporabljamo dvojno rotacijo oz. enojno rotacijo pri korenu. Oba postopka opravimo v konstantnem času $O(1)$. Ker drevo lomimo od danega elementa proti korenu drevesa, je časovna zahtevnost operacije za lomljenje drevesa sorazmerna dolžini poti od danega elementa do korena. Ker je v splošnem lomljeno drevo po višini blizu uravnoteženega, ki ima višino približno $\log_2(n)$, je pričakovana časovna zahtevnost lomljenja drevesa $O(\log(n))$. V najslabšem primeru, ko je drevo izrojeno, pa je časovna zahtevnost operacije seveda $O(n)$.

2.7.2 Vstavljanje elementa v drevo

Pri vstavljanju elementa se rekurzivno sprehodimo od korena do lista drevesa, kamor dodamo novi element. Časovna zahtevnost operacije za iskanje mesta vstavljanja elementa v list drevesa je sorazmerna višini drevesa, torej v splošnem $O(\log(n))$. Nato uporabimo operacijo za lomljenje drevesa od mesta vstavljanja elementa do korena drevesa. V prejšnjem razdelku smo že videli, da je časovna zahtevnost le-te $O(\log(n))$. Torej je skupna časovna zahtevnost operacije za vstavljanje elementa v drevo prav tako $O(\log(n))$.

2.7.3 Iskanje elementa v drevesu

Kot smo videli pri sami razlagi postopka, je iskanje elementa v lomljenem drevesu zelo podobno vstavljanju elementa v drevo. Edina razlika je v tem, da se, kadar element je v drevesu, postopek iskanja zaključi že pred listom. V vsakem primeru pa je pričakovana časovna zahtevnost samega iskanja $O(\log(n))$, lomljenja, ki sledi, pa prav tako $O(\log(n))$. Pričakovana časovna zahtevnost iskanja je torej tudi $O(\log(n))$.

2.7.4 Brisanje elementa iz drevesa

Pri brisanju elementa najprej uporabimo operacijo za iskanje elementa v drevesu. Ta operacija nam element, ki ga želimo izbrisati, prestavi v koren drevesa. Časovna zahtevnost tega postopka je, kot smo napisali že v prejšnjem razdelku, $O(\log(n))$. Če je katero od poddreves prazno, končamo. Sicer pa moramo v levem poddrevesu še premakniti maksimalni element v koren levega poddrevesa z operacijo iskanje elementa v drevesu,

katere časovna zahtevnost je prav tako $O(\log(n))$. Torej je skupna časovna zahtevnost operacije za brisanje elementa iz drevesa spet $O(\log(n))$.

LITERATURA

"Self-adjusting Binary Search Trees", Sleator and Tarjan, JACM Volume 32, No 3, July 1985, pp 652-686.

H Biermann, Splay Trees, http://www.cs.nyu.edu/algvis/java/SplayTree.html.

Kononenko, Načrtovanje podatkovnih struktur in algoritmov, Založba FE in FRI, 1996, Ljubljana.