Rdeče-črna drevesa

Uvod

Rdeče-črna drevesa so uravnotežena dvojiška iskalna drevesa, kjer ima vsako vozlišče dodaten bit informacije o njegovi barvi. Z dodatnimi pravili je zagotovljeno, da se drevo ne more izroditi in da je približno uravnoteženo.

Rdeče-črna drevesa so bila prvič predstavljena leta 1978, uporabna pa so predvsem takrat, ko potrebujemo hitro iskanje, pri čemer se podatki med samim iskanjem večkrat spreminjajo. V primerjav z AVL drevesi so operacije vstavljanja in brisanja hitrejše, je pa iskanje nekoliko počasnejše. So slabše uravnotežena kot AVL drevesa.

Koristna so predvsem v realno časovnih aplikacijah in v funkcijskem programiranju. Najdemo jih na primer v implementaciji čakalnih vrst procesov v jedru Liunx sistemov in pa v implementaciji HashMap podatkovne strukture v Javi.

Rdeče črno drevo ima vozlišča, ki so bodisi rdeča, bodisi črna. Da lahko drevo poimenujemo rdeče-črno, mora v njem veljati pet pravil:

1. Vozlišče je bodisi rdeče, bodisi črno,
2. Koren je črn,
3. Vsak list je črn - kot liste, štejemo v r-č drevesu tudi kazalce z vrednostjo null (NIL),
4. Če je vozlišče rdeče, sta njegova otroka črna,
5. Vsaka preprosta pot iz poljubnega vozlišča do lista vsebuje enako število črnih vozlišč.

Implementacija

Struktura vozlišča

Drevo implementiramo kot medsebojno povezana vozlišča, pri čemer ima vsako vozlišče ključ, kazalec na očeta, kazalca na levega in desnega sina in informacijo o barvi vozlišča.

```
Node {
	k;// vrednost ključa
	p; //kazalec/referenca na očeta
	l; //kazalec/referenca na levega sina
	r; //kazalec/referenca na desnega sina
	c; //barva vozlišča;
}
```

Ko ustvarimo novo vozlišče, so kazalci na očeta in potomce (p, l, r) enaki null. V k shranimo vrednost, pri čemer lahko imamo v rdeče-črnih drevesih vrednosti poljubnega tipa, dokler lahko med njimi definiramo operatorje večje in manjše (števila, nizi, datumi,…). Prvo vozlišče si označimo kot vrh (ga shranimo v posebno spremenljivko) in nam predstavlja koren drevesa (root). Iz njega lahko nato obiščemo vsa vozlišča drevesa, ter preko njega tudi začnemo z iskanjem, vstavljanjem in brisanjem.

Iskanje

Ker je rdeče-črno drevo podvrsta dvojiškega drevesa, so vozlišča v njem urejena. Slednje pomeni, da so vrednosti elementov, ki se nahajajo v levem poddrevesu vedno manjše od vrednosti ključa trenutnega vozlišča, vrednosti, ki se nahajajo v desnem poddrevesu, pa vedno večja od vrednosti ključa trenutnega vozlišča.

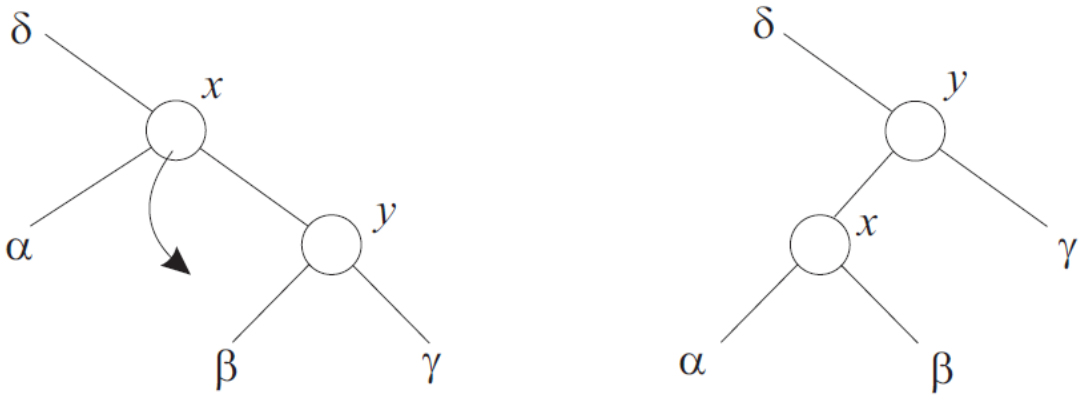
Če želimo v takšnem vozlišču najti določen ključ, pričnemo z iskanjem v korenu. Iskano vrednost (data) primerjamo z vrednostjo ključa v trenutnem vozlišču. Če je vrednost data manjša od vrednosti ključa, iskanje nadaljujemo v levem sinu, če je večja pa v desnem sinu. V kolikor je vrednost enaka, vrnemo trenutno vozlišče kot rezultat iskanja. Če pridemo do lista (vrednost null), je bilo iskanje neuspešno.

```
function search(root,data){
x=root;
while(x!=null&&x.k!=data){
        if(data<x.k)
                x=x.l;
        else if(data>x.k)
                x=x.r;
}
return x;
}
```

Rotacije

Da se lahko lotimo vstavljanja elementov v rdeče-črno drevo, je potrebno najprej spoznati dva načina menjave vozlišč v drevesih, ki ohranjajo njegovo urejenost. Poznamo dve rotaciji, levo in desno.

V levi rotaciji zamenjamo vrstni red vozlišč tako, da trenutno vozlišče nadomestimo z njegovim desnim sinom, samo pa postane njegov levi sin. Pri tem se urejenost vozlišč ohrani.

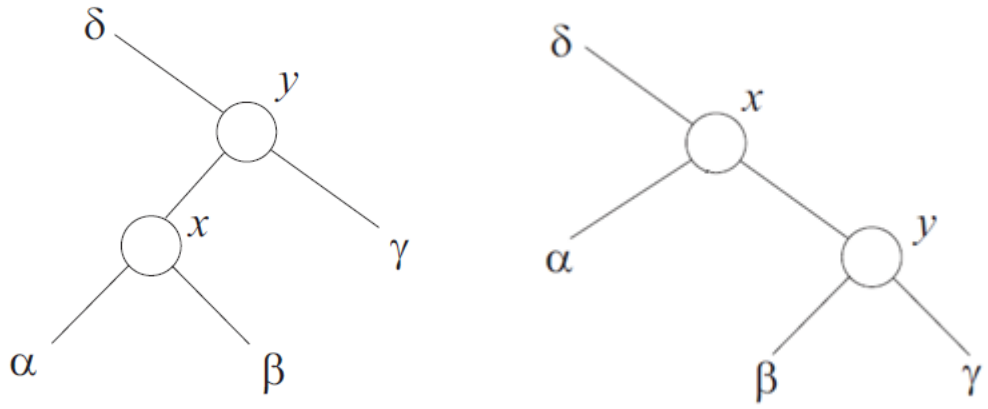


Pseudokod leve rotacije:

```
function leftRotate(root,x){
var y=x.r; //določimo y, x je na vrhu
x.r=y.l; //desni od x (b) bo postal b, ki je sedaj levi od y
if(y.l){
        y.l.p=x; //če ta obstaja, potem mu določimo starša, ki je x
}
y.p=x.p; //y prestavimo na vrh, tako da mu določimo starša od x
if(!x.p){ //če ta ne obstaja, ga naredimo za koren
        root=y;
}
else if(x==x.p.l) { //če obstaja pa pogledamo, ali je levi ali desni sin
        x.p.l=y; //če je levi sin , potem levega sina starša x-a določimo kot y
}
```

```
else {
        x.p.r=y; //če je desni sin, potem desnega sina določimo kot x
}
y.l=x; //levi od y postane x
x.p=y; //x določimo novega starša, to je y, s tem končamo menjavo
}
```

V desno rotaciji zamenjamo vrstni red vozlišč tako, da trenutno vozlišče nadomestimo z njegovim levim sinom, samo pa postane njegov desni sin. Pri tem se urejenost vozlišč prav tako ohrani.



Pseudokod desne rotacije (simetrična levi):

```
function rightRotate(root,y){
var x=y.l; //določimo x, y je na vrhu
y.l=x.r; //levi od y postane desni od x
if(x.r){
        x.r.p=y; //če obstaja desni od x (B), mu določimo starša
}
x.p=y.p; //x gre na vrh, torej oče od x je oče od y
if(!x.p){ //če je to koren, potem nastavi koren drevesa na x
        root=x;
}
else if(y==y.p.l) { //če obstaja pa pogledamo, ali je levi ali desni sin
        y.p.l=x;
}
else {
        y.p.r=x;
}
x.r=y;
y.p=x;
}
```

Vstavljanje

Vstavljanje vrednosti v rdeče črno drevo je v osnovi enako vstavljanju v dvojiško drevo. V dvojiškem drevesu se lahko nahajajo tudi podvojene vrednosti, vendar bomo v našem primeru vstavljane zasnovali tako, da se enaka vrednost ne vstavi. Prvi korak vstavljanja je tako zelo podoben iskanju, kjer najdemo vozlišče, v katerega bi morali zapisati vrednost. Z novo vrednostjo ustvarimo novo vozlišče in mu določimo rdečo barvo. V kolikor ne gre za koren, vozlišče ustrezno povežemo.

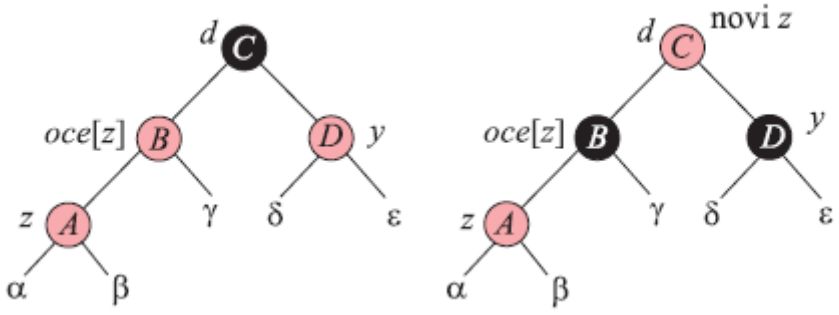
```
Function add(data, root) {
```

```
y=null;
x=root;

while(x!=null){
y=x; //zapomnimo si starša
if(data<x.k)
	x=x.l;
else if(data>x.k)
	x=x.r;
else
	return;
}
//v y imamo shranjeno vozlišče, k kateremu dodamo potomca

var z=new Node(data); //ustvarimo novo vozlišče, kjer ima ključ vrednost data
//vozlišče je privzeto rdeče barve
z.p=y; //nastavimo da je prednik novega vozlišča y
if(y==null) //če gre za koren, ustrezno priredimo vrednost
	root=z;
else if(z.k<y.k) //izberemo ali gre za levega ali desnega potomca
	y.l=z;
else
	y.r=z;
//popravimo drevo, da ohrani lastnosti rdeče-črnega drevesa
rbInsertFixup(root,z);
```

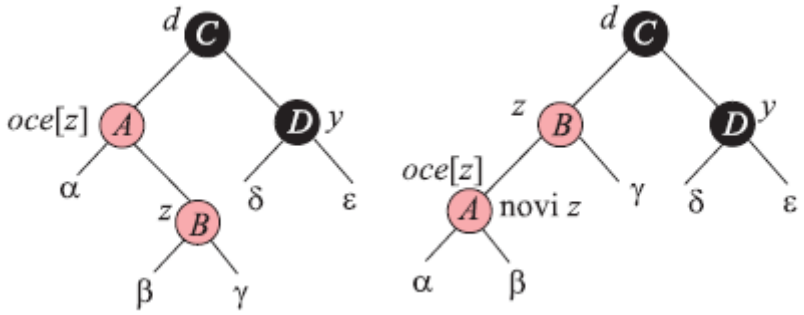
Na koncu vstavljanje je potrebno popraviti drevo na način, da zadostuje pogojem rdeče-črnega drevesa. To storimo s funkcijo rbInsertFixup, ki dobi kot vhod drevo in trenutno vstavljeno vozlišče.

Vstavljanje rdečega vozlišča (z), povzroči kršenje pravil rdeče-črnega drevesa samo v primeru, da je bilo dodano staršu, ki je rdeče barve, saj mora rdeče vozlišče imeti dva črna potomca. Zato zanko izvajanja popravkov izvajamo samo tako dolgo, dokler je oče od vstavljenega vozlišča z rdeče barve, oziroma, dokler ne pridemo do korena.

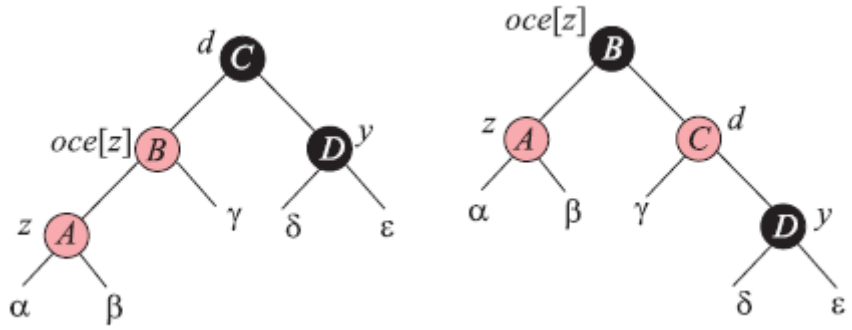
Problem dveh zaporednih rdečih vozlišč rešujemo na ta način, da moramo zagotoviti, da je stric (y) od trenutno vstavljenega vozlišča rdeč. Če sta rdeča tako oče, kot tudi stric, ju lahko prebarvamo v črno barvo, dedka, ki pa je bil zagotovo črn, pa prebarvamo v rdečo barvo. Na ta način, smo nespoštovanje pravil v drevesu prenesli dve vozlišči više (novi z), saj ne vemo, kakšne barve je njegov oče. Zato naše preverjanje in popravljanje nadaljujemo v novi iteraciji zanke (1. scenarij).



V kolikor ta pogoj ni izpolnjen, moramo zagotoviti, da je novo vstavljeno vozlišče (z) levi sin, če je črn stric desni sin (2. scenarij). To storimo z levo rotacijo med novim vozliščem in njegovim očetom.



Nato lahko z rotacijo premaknemo rdečega očeta na mesto njegovega očeta (d), ki tako postane njegov desni sin. Desnemu sinu prav tako spremenimo barvo v rdečo ter s tem zagotovimo, da so pogoji rdeče-črnega drevesa izpolnjeni (scenarij 3).



Algoritem in scenariji so simetrični za situacijo, ko je stric od novo vstavljene vrednosti y levi sin.

Pseudokod funkcije rbInsertFixup:

```
function rbInsertFixup(root,z){
while((z != root)&&(z.p.c=="red")){ //dokler je oče rdeč
        if(z.p==z.p.p.l) { //če je oče novega element levi sin
                y= z.p.p.r; //y je stric od novega vozlišča
                if(y!=null&&y.c=="red"){ //stric je rdeč -- scenarij 1
                        z.p.c="black"; //oče bo črn
                        y.c="black" //stric bo črn
                        z.p.p.c="red" //dedek bo rdeč
                        z=z.p.p; //premakni se navzgor
                        }
                else{//y je črn
                        if(z==z.p.r){ //element je desni sin -- scenarij 2
                        z=z.p;
                        leftRotate(root,z);
                        }
                        z.p.c="black";
                        z.p.p.c="red";
                        rightRotate(root,z.p.p);
                        }
        }
        else{ //vse obrnemo
```

```
                y= z.p.p.l; //y je stric od novega vozlišča
            if(y!=null&&y.c=="red"){ //stric je rdeč -- scenarij 1
                z.p.c="black"; //oče bo črn
                y.c="black" //stric bo črn
                z.p.p.c="red" //dedek bo rdeč
                z=z.p.p; //premakni se navzgor
                }
            else{//y je črn
                if(z==z.p.l){ //element je levi sin -- scenarij 2
                z=z.p;
                rightRotate(root,z);
                }
                z.p.c="black";
                z.p.p.c="red";
                leftRotate(root,z.p.p);
                }
        }
}
root.c="black"; //če smo s spremembami slučajno prišli do korena
//smo ga morda prebarvali v rdeče, zato ga prebarvamo v črno
}
```

Brisanje

Če želimo iz dvojiškega drevesa izbrisati element, se lahko soočimo s tremi scenariji. Najlažje iz drevesa brišemo list, kjer brisanje ne povzroči sprememb v strukturi dvojiškega drevesa. Drug scenarij je brisanje vozlišča, ki ima samo enega potomca. To storimo tako, da prevežemo povezavo iz njegovega starša, na edinega potomca vozlišča, ki se briše. Za brisanje vozlišča z dvema potomcema, pa moramo najti zamenjavo. Praviloma poiščemo vozlišče, z najmanjšo vrednostjo v desnem poddrevesu vozlišča, ki ga želimo izbrisati. To vozlišče ima zaradi lastnosti dvojiških dreves samo enega potomca. Obema vozliščema lahko zamenjamo vrednosti, ne da bi s tem prekršili urejenost dvojiškega drevesa. Nato lahko vozlišče z enim potomcem zbrišemo na enak način kot v drugem scenariju.

Pri rdeče-črnih drevesih je potrebno biti še posebej pazljiv, da po brisanju izvedemo tudi ustrezne spremembe, ki v drevesu ohranijo dodatne lastnosti rdeče-črnih dreves. To moramo narediti samo v primeru, da smo izbrisali črno vozlišče. Če smo namreč brisali rdeče vozlišče, z njegovim brisanjem nismo mogli spremeniti števila črnih vozlišč, kar pomeni da je 5. lastnost ohranjena. Prav tako pa brisanje rdečega vozlišča pomeni, da je bil njegov starš črn, saj so pred brisanjem v drevesu morale veljati vse lastnosti. Tako ne more priti do situacije, da bi imelo rdeče vozlišče, rdečega potomca. V kolikor brišemo črno vozlišče, pa se lahko pojavi kršitev lastnosti, ki jih odpravimo s funkcijo rbDeleteFixup, ki kot argument prejme vozlišče, ki nadomešča vozlišče, ki ga brišemo.

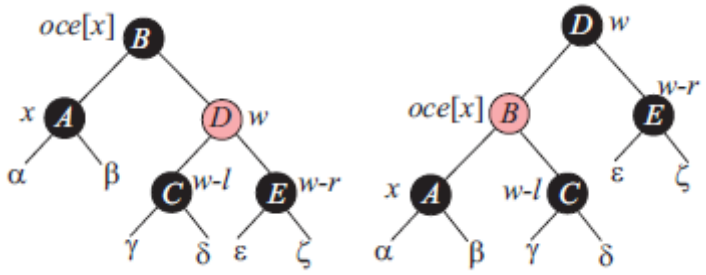
```
Function delete(data, root) {
var y=null;
var x=root;

//iskanje
while(x!=null&&x.k!=data){
        y=x;
```

```
        if(data<x.k)
                x=x.l;
        else if(data>x.k)
                x=x.r;
        }
if(x==null)
        return; //ne najdemo vozlišča za brisanje

//najdi vozlišče za brisanje
var z=x;

if(z.l==null||z.r==null) //vsaj en sin je null, lahko prevežmo
        y=z;
else{//vozlišče ima oba potomca
        y=z.r;// v desnem podrevesu
        while(y.l!=null){ //najdi najmanjši element
                y=y.l;
        }
}
//y je vozlišče, ki ga bomo brisali
if(y.l!=null)
        x=y.l;
else
        x=y.r;
//x je vozlišče, ki ga bo zamenjalo
if(x!=null)
        x.p=y.p;

if(y.p==null) //če smo v korenu
root=x;
else {
        if(y==y.p.l)
                y.p.l=x;
        else
                y.p.r=x;
}

if(y!=z) z.k=y.k; //če smo delali zamenjavo uredimo vrednosti ključa

//fixup
if(y.c=="black") rbDeleteFixup(root,x);

}
```

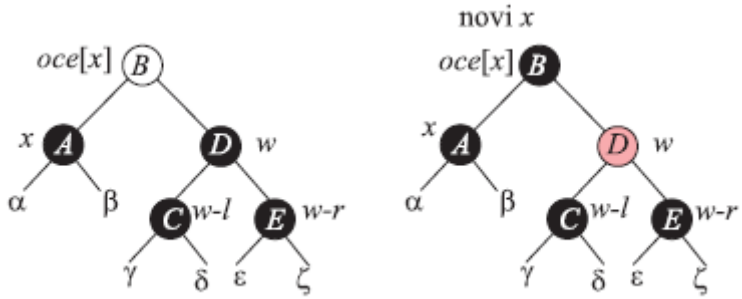
Osnovna ideja funkcije rbDeleteFixup je, da rešuje problem, ko je vozlišče, ki nadomešča brisano vozlišče (x) črne barve. Če je x rdeč, potem ga spremenimo v črnega in s tem ohranimo vse lastnosti rdeče-črnih dreves. Če je x črn, pa dobimo problem dvojne črnine, saj bi za ohranitev lastnosti rdeče-črnih dreves morali črno barvo v tem vozlišču šteti dvakrat. Ta problem poskušamo razrešiti tako, da »dvojno črnino« potiskamo po drevesu navzgor (v starša od x), tako dolgo, dokler ni slednji rdeč oziroma dokler ne pridemo do korena. V obeh primerih ga nastavimo na črno barvo ter s tem dobimo

drevo z vsemi lastnostmi rdeče črnih dreves. Slednje izvajamo z rotacijami, pri čemer se lahko soočimo s štirimi scenariji.

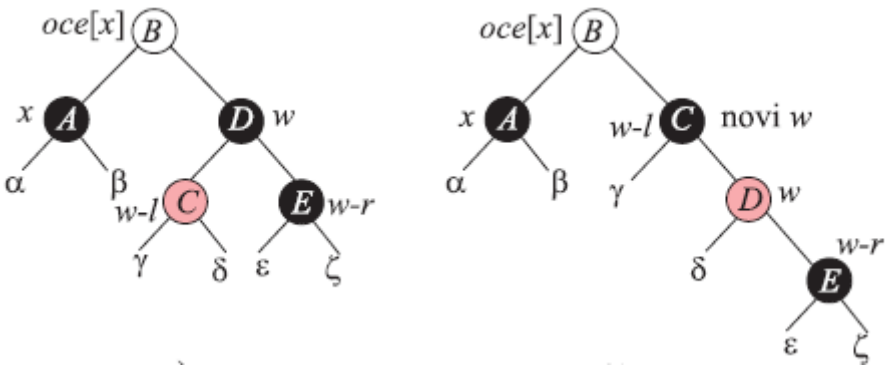
Da lahko popravimo drevo, moramo najprej zagotoviti, da ima x rdečega brata. Če ga nima, smo v 1. scenariju, kjer moramo za učinkovito zamenjavo barve izvesti levo rotacijo na očetu od x. Ta tako postane rdeče barve, s tem pa zagotovimo, da ima x rdečega brata.



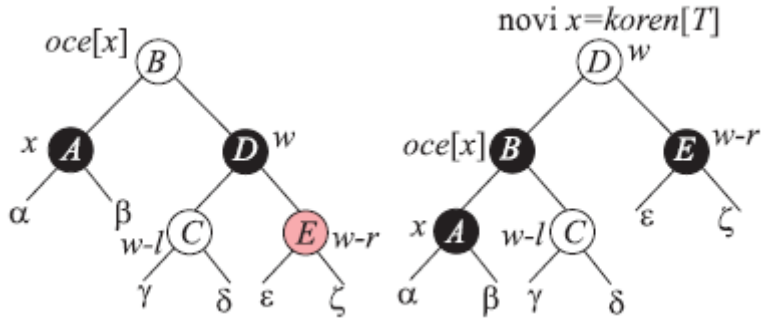
Kadar ima x črnega brata, znamo drevo ustrezno popraviti, če ima njegov brat w, dva črna sinova ali pa rdečega desnega sina. V primeru, ko ima dva črna sinova, gre za 2. scenarij. Takrat lahko brata pobarvamo v rdeče, ter njegovo črnino, ter eno izmed dveh črnin od x prenesemo na starša od x. tako je vozlišče z dvojno črnino (x) postal oče od x, postopek popravljanja pa ponovimo v novi iteraciji zanke. Oče od x je pred rotacijo označen z belo barvo, ker lahko v 2. scenarij vstopamo tudi brez izvedbe 1. scenarija in zato ne moramo biti prepričani o njegovi barvi. Ta nas v tem koraku tudi ne zanima. V kolikor je barva že pred rotacijo črna, smo v naslednji iteraciji zanke v situaciji dvojne črnine, v kolikor pa je rdeča, pa se bo zanka v naslednji iteraciji zaključila, vozlišču pa se bo priredila črna barva po koncu zanke.



Če pa ima w rdečega levega sina in črnega desnega sina, moramo najprej pridelati rdečega desnega sina. V tem primeru vstopamo v 3. scenarij, kjer izvedemo desno rotacijo na w, w spremenimo v rdeče vozlišče, prej rdeč levi sin, pa postane črn.



Ostane nam tako samo še 4. scenarij, ko je levi sin od w rdeč. Ker smo lahko v scenarij vstopili tudi brez izvedbe 1. in 3. scenarija, je lahko barva levega sina bodisi rdeča ali črna (vozlišče je zato označeno z belo), kar velja tudi za očeta od x. Sedaj lahko z levo rotacijo na očetu od x ter s spremembo barve desnega sina od w v črno in barve očeta od x v črno, zagotovimo lastnosti rdeče črnega drevesa. Brat w dobi barvo očeta x pred rotacijo. V tem primeru nastavimo vrednost x na koren, zato da zaključimo izvajanje zanke. Vrstica po zanki, ki x pobarva v črno, nam tako ne more narediti škode, saj mora koren vedno biti črn.



Pseudokod funkcije rbDeleteFixup:

```
function rbDeleteFixup(root,x){
while(root!=x && x.c=="black"){
if(x==x.p.l){  // brisan je levi sin
        w=x.p.r; //w je brat od x
        if(w.c=="red") //scenarij 1
        {//x-u zagotovimo črnega brata
        w.c="black";
        x.p.c="red";
        leftRotate(root,x.p);
        w=x.p.r;
        }
        if (w.l.c=="black"&&w.r.c=="black") //scenarij 2
        {//če ima brat oba črna potomca, prenesemo črno barvo k staršu
        w.c="red";
        x=x.p; //če je starš v naslednji iteraciji zanke rdeč, se zanka zaključi
        }
        else { //vsaj en otrok od brata w je rdeč
                if(w.r.c=="black"){// scenarij 3
                //rdeč je levi sin, želimo da bi bil pa desni
                w.l.c="black";
                w.c="red"
                rightRotate(root,w);
                w=x.p.r;
                }//po rotaciji je rdeč desni sin
                //scenarij 4
                w.c=x.p.c;//  brat dobi barvo od starša
                x.p.c="black"; //starš postane črn
                w.r.c="black"; //in desni sin, ki je bil rdeč, postane črn
                leftRotate(root,x.p); //izvedemo levo rotacijo
                x=T.root; //drevo je v skladu s pravili, zato zanko zapustimo.
        }
```

```
}//simetrično se izvedejo operacije če brišemo na desni strani
else { //brisan je desni sin
        w=x.p.l; //w je brat od x
        if(w.c=="red") //scenarij 1
        {
        w.c="black";
        x.p.c="red";
        rightRotate(root,x.p);
        w=x.p.l;
        }
        if (w.r.c=="black"&&w.l.c=="black"){ //scenarij 2
        w.c="red";
        x=x.p;
        }
        else {
                if(w.l.c=="black"){//scenarij 3
                w.r.c="black";
                w.c="red"
                leftRotate(root,w);
                w=x.p.l;
                }
                w.c=x.p.c; //scenarij 4
                x.p.c="black";
                w.l.c="black";
                rightRotate(root,x.p);
                x=root;
        }
}
}
x.c="black";
}
```

Vizualizacija

Pri izdelavi vizualizacije si pomagajte z obstoječimi knjižnicami za risanje grafov. V vozliščih naj bodo zapisane vrednosti elementov. Pri vizualizaciji je dovolj, da se izriše končno stanje drevesa po izvedenih operacijah (ni potrebnega izrisa v živo). Izris je lahko narejen ločeno od samega algoritma (npr. konzolna aplikacija generira sliko/html/js, ki jo lahko odpremo s kakšnim drugim orodjem). Priporočam uporabo knjižnice Graphviz ali d3.js. V kolikor ne boste izdelali vizualizacije izdelajte izpis vseh elementov z levim obhodom v globino (DFS), kjer zraven vrednosti v vozlišču izpišete tudi njegovo globino in barvo.

Kriterij

Pri nalogi je mogoče zbrati 7 točk, po sledečem kriteriju:

4 točke – vstavljanje in iskanje elementov v r-č drevesu ter časovna zahtevnost operacij.

2 točki – brisanje elementa in časovna zahtevnost brisanja.

1 točka – grafični prikaz trenutnega stanja drevesa.