Crveno-crna stabla

Vujičić, Barbara

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO–MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Barbara Vujičić

CRVENO-CRNA STABLA

Diplomski rad

Zagreb, 2017.

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO–MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Barbara Vujičić

CRVENO-CRNA STABLA

Diplomski rad

Voditelj rada:
doc. dr. sc. Goranka Nogo

Zagreb, 2017.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____ , predsjednik
2. _____ , član
3. _____ , član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

U ovom radu prezentirat ćemo crveno-crno stablo kao vrstu binarnog stabla pretraživanja s dobrom balansiranošću. Zbog dobre vremenske složenosti operacija nad njom, ova struktura korištena je u implementaciji mnogih *collectiona* i *containera* za pohranu podataka, kao što su `TreeMap` i `TreeSet` u jeziku Java te `set`, `multiset`, `map` i `multimap` iz Standard Template Libraryja u jeziku C++, ali i u implementaciji nekih algoritama, kao što je, na primjer, *scheduling* algoritam po kojem radi "organizator procesa" operacijskog sustava Linux Kernel zvan Completely Fair Scheduler.

U prvom poglavlju dat ćemo osnovnu definiciju crveno-crnog stabla i navesti crveno-crna svojstva koja takvo stablo mora zadovoljavati. Pokazat ćemo da ta svojstva osiguravaju da nijedan put od korijena do lista nije više od dva puta dulji od bilo kojeg drugog puta od korijena do lista te dokazati da zato takvo stablo s n čvorova ima visinu $O(\log n)$.

U sljedećem poglavlju bavit ćemo se operacijama nad crveno-crnim stablom. Operaciju pretraživanja nećemo opisivati jer je jednaka onoj za općenita stabla. Ukratko ćemo opisati i analizirati trajanje rotacija koje ćemo, uz mijenjanje boja, najčešće koristiti u "popravljanju" crveno-crnog stabla. Veći značaj dat ćemo operacijama dodavanja i brisanja. Obje operacije baziraju se na istim operacijama definiranim za općenito binarno stablo, ali ćemo vidjeti da se mogu kršiti svojstva crveno-crnog stabla. Skicama ćemo prikazati takve slučajeve i kako ih riješiti. Također ćemo dati pseudokôdove tih operacija i analizirati njihovu vremensku složenost.

U zadnjem poglavlju opisat ćemo projekt Red Black Trees koji je napravljen u sklopu ovog rada i koji implementira rješenje jednog geometrijskog problema uz pomoć crveno-crnog stabla. Objasnit ćemo kako se pokreće i koristi projekt te dati i opisati dijelove implementacije.

Poglavlje 1

Definicija i osnovna svojstva

Prije samog upoznavanja s crveno-crnim stablom, prisjetimo se, na kratko, što je binarno stablo pretraživanja.

Definicija 1. *Binarno stablo T je **binarno stablo pretraživanja** ako su ispunjeni sljedeći uvjeti:*

1. *Čvorovi od T su označeni podacima nekog tipa nad kojim je definiran totalni uređaj.*
2. *Neka je i bilo koji čvor od T. Tada su oznake svih čvorova u lijevom podstablu od i manje od oznake od i. Također, oznake svih čvorova u desnom podstablu od i veće su ili jednake od oznake od i.*

Drugi uvjet osigurava da inorder obilazak stabla daje te podatke sortirane po definiranom totalnom uređaju.

U Uvodu smo spomenuli da ćemo prikazati crveno-crno stablo kao stablo s dobrom balansiranošću, stoga ćemo se prisjetiti i što je to balansirano binarno stablo.

Definicija 2. ***Balansirano binarno stablo*** *je binarno stablo kojem su čvorovi približno jednako distribuirani na lijevom i desnom podstablu.*

Visina mu je približno $\log n$ i operacije nad ovakvim stablima u principu su složene.

Ponovimo i što znači da funkcija $f(n)$ ima vremensku složenost $O(g(n))$.

Definicija 3. *Neka su $f, g : \mathbb{N} \to \mathbb{R}^+$ dvije funkcije. Kažemo da je funkcija g **asimptotska gornja međa** za funkciju f ako postoji $c > 0$ i $n_0 \in \mathbb{N}$ tako da za svaki $n \geq n_0$ vrijedi $f(n) \leq cg(n)$. Pišemo: $f(n) = O(g(n))$.*

1.1 Definicija

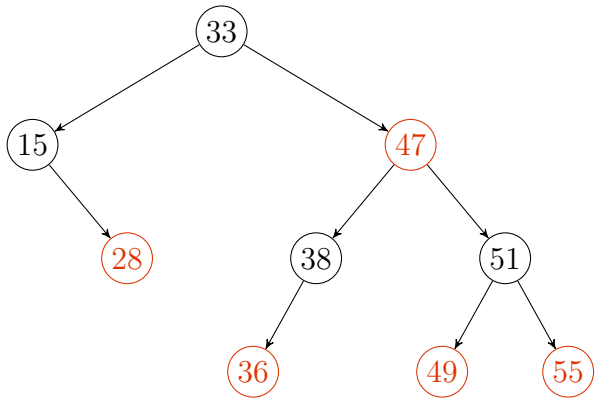
Definicija 4. ***Crveno-crno stablo*** *je binarno stablo pretraživanja u kojem svaki čvor, uz oznaku, roditelja, lijevo i desno dijete, ima i boju, koja može biti crvena ili crna.*

Čvoru koji nema roditelja ili neko od djece za ta svojstva pridružuje se *stražarski čvor* (*sentinel*). To je čvor s istim svojstvima koje imaju obični čvorovi crveno-crnog stabla. Boja mu je crna, a sve ostale vrijednosti proizvoljne. Za crveno-crno stablo T označava se s $T.nil$, a pri crtanju stabla obično se izostavlja.

1.2 Osnovna svojstva

Sad možemo navesti crveno-crna svojstva koja svako crveno-crno stablo mora imati:

1. Svaki čvor je ili crven ili crn.
2. Korijen je crn.
3. Stražarski čvor je crn.
4. Ako je čvor crven, onda su oba njegova djeteta crna.
5. Svi putevi od nekog čvora do listova koji su mu potomci sadrže jednak broj crnih čvorova.



Slika 1.1: Primjer crveno-crnog stabla

Zbog zadnjeg svojstva dobro je definirana crna visina čvora x kao broj crnih čvorova na svakom putu od čvora x (ne računajući ga) do lista koji mu je potomak. Crnu visinu čvora x označavamo s $bh(x)$. Crnu visinu crveno-crnog stabla definiramo kao crnu visinu korijena tog stabla.

Neka je $bh(x)$ crna visina nekog čvora x. Najmanji put od njega do lista koji mu je potomak može biti dugačak upravo toliko - kad su svi čvorovi tog puta crni. Radi četvrtog svojstva, najduži put od tog istog čvora do nekog drugog lista koji mu je potomak je put u kojem je svaki drugi čvor crn. Taj put je dugačak $2 * bh(x)$.

Stoga je najduži put od nekog čvora do lista koji mu je potomak maksimalno dva puta duži od najkraćeg puta od tog istog čvora do nekog drugog lista koji mu je potomak. Zato je ovakvo stablo približno dobro balansirano.

Sljedeći teorem dokazuje da je visina crveno-crnog stabla s n čvorova jednaka $O(\log n)$.

Teorem 1. *Crveno-crno stablo s n čvorova ima visinu najviše* $2\log(n+1)$.

Dokaz. Prvo ćemo, matematičkom indukcijom, dokazati da bilo koje podstablo s korijenom u čvoru x sadrži barem $2^{bh(x)} - 1$ unutarnjih čvorova.
Ako je $bh(x) = 0$, onda je x list, pa to stablo zaista sadrži $2^0 - 1 = 0$ unutarnjih čvorova.
Pretpostavimo sad da je x unutarnji čvor i da ima dva djeteta. Oni imaju crnu visinu jednaku $bh(x)$ ako je čvor crven ili $bh(x) - 1$ ako je čvor crn. Sigurno je onda da podstabla s korijenima u tim čvorovima imaju barem $2^{bh(x)-1} - 1$ unutarnjih čvorova. Podstablo s korijenom u x tada ima barem $2*(2^{bh(x)-1}-1)+1 = 2^{bh(x)}-2+1 = 2^{bh(x)}-1$ unutarnjih čvorova.
Završimo sada dokaz teorema. Zbog četvrtog svojstva vrijedi da je barem pola čvorova bilo kojeg puta od korijena do lista crno, odnosno vrijedi: $bh(x) \geq \frac{h(x)}{2}$. Iz prethodnog dokazanog vrijedi:

$$\begin{aligned} n &\geq 2^{\frac{h(x)}{2}} - 1 \\ n+1 &\geq 2^{\frac{h(x)}{2}} \\ \frac{h(x)}{2} &\leq \log(n+1) \\ h(x) &\leq 2\log(n+1). \end{aligned}$$

□

Rekli smo da je pretraživanje u ovakvom stablu jednako pretraživanju u binarnom stablu traženja, a kako je, za svako binarno stablo pretraživanja, ta operacija u najgorem slučaju jednaka visini stabla, ovaj teorem nam daje zaključiti da je vremenska složenost pretraživanja crveno-crnog stabla jednaka $O(\log n)$.

Operacije dodavanja i brisanja čvora na crveno-crnom stablu također se mogu obaviti u $O(\log n)$ vremenu, što ćemo dokazati kasnije. Dodavanje i brisanje nekog čvora u crveno-crnom stablu može rezultirati time da to stablo više ne zadovoljava sva crveno-crna svojstva. Radi toga se moraju obaviti promjene s bojama čvorova ili rotacije.

Poglavlje 2

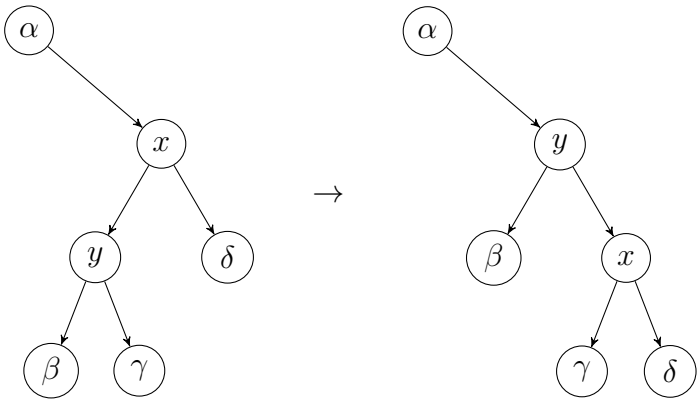
Operacije nad crveno-crnim stablima

2.1 Rotacija u crveno-crnom stablu

Rotacije su jednake onima na običnom binarnom stablu pretraživanja. Čuvaju osnovno svojstvo binarnih stabla pretraživanja. Rotacije se kod običnog binarnog stabla koriste kako bi se smanjila visina stabla, pomičući veće podstablo prema korijenu, a manje u visinu. Ovdje će se rotacije izvoditi kako bi se kod dodavanja ili brisanja iz crveno-crnog stabla ispravili slučajevi koji krše četvrto ili peto crveno-crno svojstvo, što će nam dati bolju balansiranost.

2.1.1 Opis operacije

Dajemo opis desne rotacije. Desnu rotaciju oko čvora x izvodit ćemo samo ako njegovo lijevo dijete y nije stražarski čvor. Kao lijevo dijete čvora x postavit ćemo desno dijete čvora y. Čvor koji je bio roditelj x-u sada je roditelj y-u, a y dijete tom



Slika 2.1: Desna rotacija

čvoru - lijevo ili desno ovisno o tome koje je x bio. To znači da, ako je x bio korijen stabla, sada to postaje y. Čvor y postaje roditelj od x, a x njegovo desno dijete.

Slijedi i pseudokôd desne rotacije. Lijeva rotacija je analogna s promjenama lijevo - desno.

```
1  Right-Rotate(T,x) {
2      y = x.left
3      x.left = y.right
4      if y.right != T.nil
5          y.right.p = x
6      y.p = x.p
7      if x.p == T.nil
8          T.root = y
9      else if x == x.p.right
10         x.p.right = y
11     else
12         x.p.left = y
13     y.right = x
14     x.p = y
15 }
```

Metoda `Right-Rotate` prima čvor oko kojeg se obavlja rotacija. U liniji 3 tom čvoru za lijevo dijete postavljamo desno dijete njegovog lijevog djeteta. Ako to dijete nije stražarski čvor, onda ćemo mu postaviti roditelja (linija 5). Linija 6 lijevom djetetu čvora oko kojeg rotiramo postavlja roditelja koji je bio tom čvoru. Ako je taj roditelj stražarski čvor, tada će linija 8 lijevo dijete postaviti kao korijen stabla. Inače ga linije 9 - 12 postavljaju kao lijevo ili desno dijete roditelju. I na kraju, linije 13 i 14 zamjenjuju odnos roditelj - dijete čvoru oko kojeg smo rotirali i njegovom lijevom djetetu.

2.1.2 Analiza

Iz pseudokôda se lako vidi da rotacija samo mijenja nekoliko pokazivača i ne ovisi o broju čvorova stabla, radi čega je vremenska složenost ove operacije jednaka $O(1)$.

2.2 Dodavanje čvora u crveno-crno stablo

2.2.1 Opis operacije

Samo dodavanje čvora u crveno-crno stablo modifikacija je dodavanja definiranog za binarna stabla pretraživanja - čvor se stavlja na ono mjesto koje će sačuvati osnovno svojstvo binarnog stabla. Dodatno, taj čvor bojimo crveno jer ćemo tako sigurno sačuvati peto svojstvo. Međutim, ako je roditelj upravo dodanog čvora također

crven, narušili smo četvrto svojstvo. To ćemo popraviti rotacijama i promjenama boje čvorova.

Slijedi pseudokôd dodavanja čvora u crveno-crno stablo.

```
1  Insert(T,z) {
2      y = T.nil
3      x = T.root
4
5      while x != T.nil
6          y = x
7          if z.key < x.key
8              x = x.left
9          else
10             x = x.right
11
12     z.p = y
13     if y == T.nil
14         T.root = z
15     else if z.key < y.key
16         y.left = z
17     else
18         y.right = z
19
20     z.left = T.nil
21     z.right = T.nil
22     z.color = Red
23
24     Insert-Fixup(T,z)
25 }
```

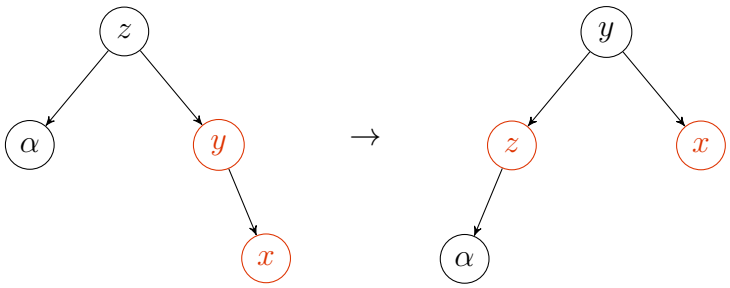
Čvor koji ulazi u metodu `Insert` već sadrži podatak. Treba mu sad naći roditelja. U linijama 5 - 10 traži se mjesto u stablu gdje ćemo dodati čvor tako da uspoređujemo podatak trenutnog čvora i čvora kojeg dodajemo. Ako je podatak čvora kojeg dodajemo manji, tada trenutni čvor postaje lijevo dijete trenutnog čvora. Inače, to postaje desno dijete. To radimo dok god ne dođemo do kraja puta, odnosno dok trenutni čvor nije stražarski. Kada se to dogodi, znači da smo našli roditelja. Linija 12 postavit će roditelja novom čvoru, a linije 13 - 18 njega kao djeteta ili, pak, kao korijen stabla. Novom čvoru postavljamo crvenu boju i stražarski čvor kao lijevo i desno dijete. Na kraju pozivamo metodu `Insert-Fixup` koja će popraviti kršenje crveno-crnih svojstava.

2.2.2 Popravljanje crveno-crnih svojstava

Popravljanje ćemo proučavati pretpostavivši da se čvor, koji krši četvrto svojstvo, nalazi u desnom podstablu svog djeda i tu analizirati tri slučaja. Nazovimo taj čvor

x, njegovog roditelja y, a djeda, za kojeg sigurno znamo da je crn, z. Prva su dva slučaja kada je ujak čvora x crn.

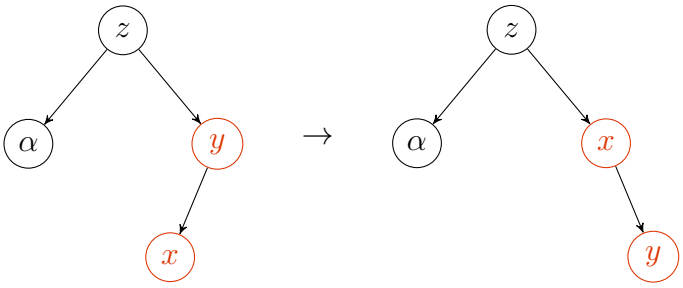
1. slučaj: Ako je x desno dijete čvora y, napravimo lijevu rotaciju oko z i promijenimo boju od y u crno, a boju od z u crveno.



Slika 2.2: 1. slučaj

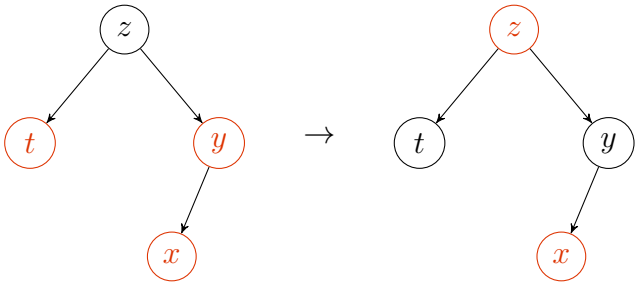
Ovako je vraćeno četvrto crveno-crno svojstvo te su sigurno sačuvana i prva tri. Lijevo je podstablo dobilo jedan crni (y), ali je i izgubilo jedan crni čvor koji je postao crveni (z). Time je crna visina lijevog podstabla ostala ista. Primijetimo također da ovo bojanje čvora z u crveno nije napravilo novo kršenje četvrtog svojstva u lijevom podstablu jer nam je pretpostavka da ujak od x nije crven. Desno podstablo je izgubilo jedan crni čvor (z), a jedan crveni se obojao u crno (y), čime se i tu sačuvala crna visina. Znači, sačuvali smo i peto svojstvo.

2. slučaj: Ako je x lijevo dijete od y, radimo desnu rotaciju oko y. Primijetimo na Slici 2.3 da tako 2. slučaj pretvaramo u 1., pa ga dalje tako i rješavamo.



Slika 2.3: 2. slučaj

3. slučaj: Ako je ujak od x također crven, najjednostavnije rješenje za vraćanje četvrtog svojstva je bojanje roditelja i ujaka u crno, a djeda u crveno. Nebitno je je li x lijevo ili desno dijete.



Slika 2.4: 3. slučaj

Ovim izmjenama boja čvorova nismo narušili peto svojstvo. Također su sačuvani prvo i treće svojstvo. Četvrto svojstvo je možda narušeno ovog puta više u stablu - između čvora z i njegovog roditelja. To se može riješiti ponovo analizirajući ove slučajave. Ako je narušeno drugo svojstvo, tada korijen samo obojimo u crno. Time smo sačuvali sva svojstva, a crnu visinu stabla povećali za jedan.

Metoda `Insert-Fixup` prima čvor koji smo upravo dodali u stablo i koji možda krši drugo ili četvrto svojstvo. Stavljamo samo dio pseudokôda - za slučaj kada je čvor u desnom podstablu svog djeda. Obrnuti slučaj je analogan s promjenama lijevo-desno.

```
1  Insert-Fixup(T,z) {
2      while z.p.color == Red
3          if z.p == z.p.p.right
4              y = z.p.p.left
5
6              if y.color == Black
7                  if z == z.p.left
8                      z = z.p
9                      Right-Rotate(T, z)
10
11                 z.p.color = Black
12                 z.p.p.color = Red
13                 Left-Rotate(T,z.p.p)
14
15             else
16                 z.p.color = Black
17                 y.color = Black
18                 z.p.p.color = Red
19                 z = z.p.p
20         else
21             // right and left exchanged
22     T.root.color = Black
23 }
```

Linije 6 - 13 ove metode izvode se ako ujak čvora koji krši svojstvo nije crven. Prvo provjeravamo istinitost 2. slučaja, kako bi ga, ako on zaista je istinit, mogli odmah

transformirati u 1. slučaj i riješiti taj. Primijetimo da upravo linija 11 zaustavlja petlju `while`. Linije 16 - 19 implementiraju 3. slučaj. Linija 19 postavlja novi čvor koji možda krši neko svojstvo. Ako krši četvrto, onda se ponovo izvršava petlja `while`. Inače se krši drugo kojeg popravljamo u liniji 22.

2.2.3 Analiza

Izračunajmo kolika je vremenska složenost cijele operacije dodavanja čvora u crveno-crno stablo. Petlja `while` u metodi `Insert` traje dok ne dođemo do kraja puta, što je, u najgorem slučaju, jednako visini stabla. Dakle ta petlja traje $O(\log n)$. Ostale operacije u toj metodi ne ovise o broju čvorova. U metodi `Insert-Fixup` petlja `while` izvodi se dok god je istinit 3. slučaj. Međutim, kako se to narušavanje četvrtog crveno-crnog svojstva pomiče samo više u stablo (i to čak dvije razine - s čvora na njegovog djeda), ova petlja se, u najgorem slučaju, izvodi $O(\log n)$. Jednom, kada je istinit 1. slučaj, petlja staje. Također, maksimalan broj rotacija koje se mogu izvesti je 2. Zaključujemo da je vremenska složenost dodavanja u crveno-crno stablo jednaka $O(\log n)$.

2.3 Brisanje čvora iz crveno-crnog stabla

2.3.1 Opis operacije

Brisanje čvora iz crveno-crnog stabla započinje modifikacijom iste operacije definirane za binarno stablo pretraživanja. Ta operacija čuva osnovno svojstvo binarnog stabla na način da se čvor, kojeg treba obrisati, zamijeni jednim njegovim djetetom ako nema drugo ili njegovim suksesorom - čvorom koji se nalazi "skroz lijevo" u njegovom desnom podstablu, odnosno čvorom čiji je podatak najmanji u desnom podstablu.

Pokažimo pseudokôd te operacije.

```
1  Delete(T,z) {
2      if z.left == T.nil
3          y = z
4          y_original_color = y.color
5          x = z.right
6
7          Transplant(T,z,z.right)
8
9      else if z.right == T.nil
10         y = z
11         y_original_color = y.color
12         x = z.left
13
14         Transplant(T,z,z.left)
```

```
15
16      else
17          y = Minimum(T, z.right)
18          y_original_color = y.color
19
20          x = y.right
21
22          if y.p != z
23              Transplant(T,y,y.right)
24              y.right = z.right
25              y.right.p = y
26
27          Transplant(T,z,y)
28          y.left = z.left
29          y.left.p = y
30          y.color = z.color
31
32      if y_original_color == Black
33          Delete-Fixup(T,x)
34  }
```

Metoda prima čvor kojeg bi trebalo obrisati. Kod prva dva izvođenja naredbe `if`, čvor kojeg trebamo obrisati nema jedno dijete, pa ga zamjenjujemo onim drugim. Zamjenu radi metoda `Transplant`, a predstavlja operaciju nad binarnim stablom u kojoj se drugi čvor kao dijete postavlja onom čvoru koji je bio roditelj prvom. Odnosno, ako je prvi čvor bio korijen stabla, sada je to drugi. Ovako smo izgubili onu boju koju je imao prvi čvor te nju spremamo u varijablu `y_original_color`. Vrijednost od `x` je dijete koje ga je zamijenilo.

Ako čvor pak ima oba djeteta, zamijenit ćemo ga njegovim suksesorom. Ali prije toga, suksesora će zamijeniti njegovo desno dijete. Suksesor će dobiti boju čvora kojeg mijenja. Tako smo izgubili boju suksesora koju spremamo u `y_original_color`, a `x` će biti čvor koji ga je zamijenio.

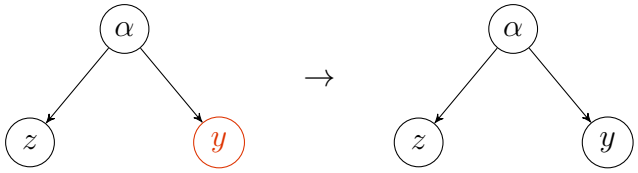
Jedino crveno-crno svojstvo koje može biti narušeno je peto, i to samo ako smo izgubili crnu boju jer smo tako na bar jednom putu smanjili crnu visinu. To narušavanje nalazi se na mjestu čvora `x`. Njega ćemo proslijediti metodi `Delete-Fixup` koja će vratiti crveno-crna svojstva.

2.3.2 Popravljanje crveno-crnih svojstava

Popravljanje svojstava analizirat ćemo u nekoliko slučajeva - ovisno o boji čvora gdje se krši to svojstvo i boji njegovih susjeda.

1. slučaj: Ako je čvor gdje se krši peto svojstvo crven, tada je najjednostavnije rješenje obojati taj čvor u crno. Tako smo na svim putevima koji su sadržavali

obrisani crni čvor i kojima se, nakon njegovog brisanja, broj crnih čvorova smanjio za jedan, ponovo vratili taj broj. Druga svojstva time nismo mogli poništiti.

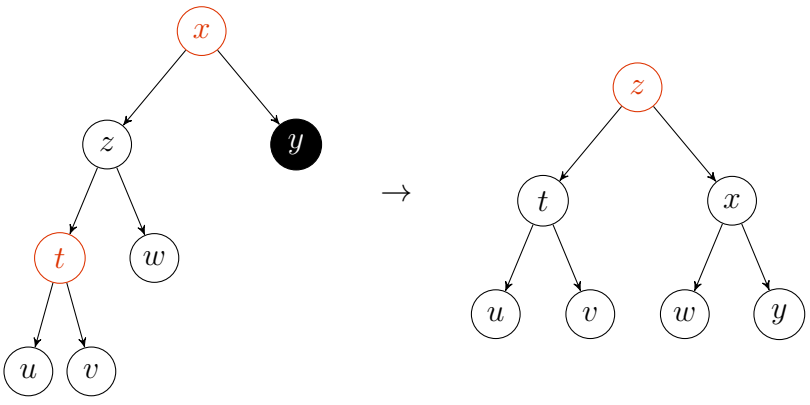


Slika 2.5: 1. slučaj

Sljedeći slučajevi su kada taj čvor nije crven. Rješavanje tih slučajeva započet ćemo nečim što nam se na prvu neće činiti smislenim. Taj crni čvor obojat ćemo još jednom u crno. Sad je taj čvor obojan duplo crno i doprinosi dva put u računanju crne visine, čime je, na neki način, popravljeno peto crveno-crno svojstvo. Medutim, duplo crno nije boja u crveno-crnom stablu i to trebamo riješiti. Razlog zašto bojamo u duplo crno je samo da nam bude lakše pratiti mjesto u stablu gdje se krši to svojstvo. U kôdu to nećemo raditi. Sad kad smo obojali taj čvor u duplo crno, analizirajmo ostala četiri slučaja.

Pretpostavit ćemo da je promatrani čvor desno dijete svog roditelja. Također nazovimo čvorove koji će nam biti od važnosti kako bi nam bilo lakše. Neka je naš duplo crni čvor y, njegov roditelj x i brat z.

2. slučaj: Ako je z crn čvor koji ima crveno lijevo dijete t, napravit ćemo desnu rotaciju oko x. Na mjesto od x sada dolazi z i njegova boja postaje ona boja koju je imao x. Čvoru y brišemo jednu crnu boju, a čvor t bojamo u crno. Čvor x postaje crn ako je bio crven. Inače, ostaje crn.

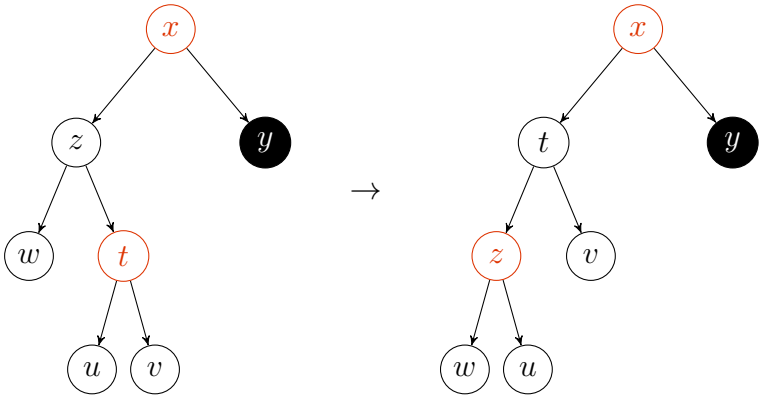


Slika 2.6: 2. slučaj

Na skici se najbolje vidi kako smo riješili problem duplo crne boje i sačuvali sva svojstva. Lijevo podstablo je izgubilo jedan crni čvor (z), ali smo bojanjem crvenog

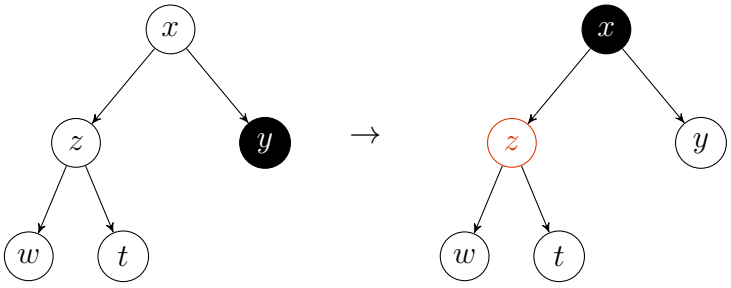
čvora t u crno sačuvali crnu visinu. U desnom podstablu smo duplo crnom čvoru makli jednu crnu, ali smo u stablo dodali jedan crni čvor (x). Na skici je prikazan slučaj kada je x crven.

3. slučaj: Ako je z crn čvor koji ima crveno desno dijete t, tada radimo lijevu rotaciju oko z. Ti čvorovi medusobno zamjenjuju boje. Primijetimo da smo time dobili 2. slučaj, pa ga tako dalje i riješavamo.



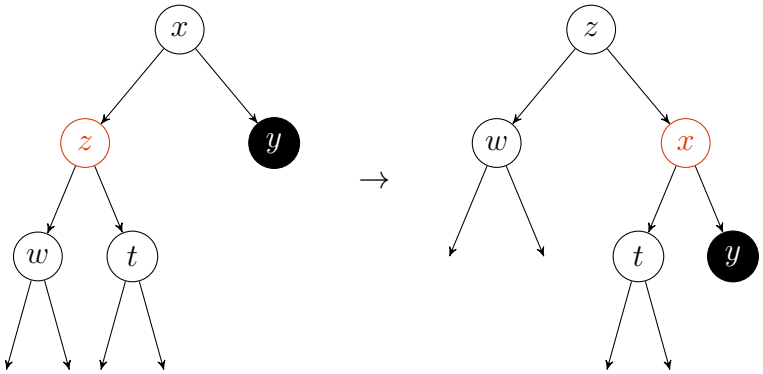
Slika 2.7: 3. slučaj

4. slučaj: Ako je z crn i nema crveno dijete, bojamo njega u crveno, y-u brišemo jednu crnu boju, a njihovog roditeja x bojamo u crno. Oba podstabla od x su izgubila jednu crnu boju, ali ju je x dobio. Tako smo sačuvali crnu visinu. Također, ako je x bio crven, a sad smo ga obojali u crno, nismo mogli uništiti bilo koje drugo svojstvo te smo riješili problem. Međutim, ako je x bio crn, sad smo njega obojali u duplo crno. Time smo pomakli problem na malo više u stablu, ali ga rješavamo ponovo analizirajući slučajeve.



Slika 2.8: 4. slučaj

5. slučaj: Ako je z crven, obavit ćemo desnu rotaciju oko x te im zamijeniti boje. I dalje je y duplo crno obojen, ali sad ima brata koji je prije bio z-ovo desno dijete, a taj je sigurno crn. Time smo ovaj slučaj transformirali u 2., 3. ili 4. slučaj, pa ga tako dalje i riješavamo.



Slika 2.9: 5. slučaj

Slijedi pseudokôd metode `Delete-Fixup`.

```
1  Delete-Fixup(T,x) {
2      while x != T.root && x.color == Black
3          if x == x.p.right
4              w = x.parent.left
5
6              if w.color == Red
7                  w.color = Black
8                  x.p.color = Red
9                  Right-Rotate(T,x.p)
10
11                 w = x.p.left
12
13             if w.left.color == Black && w.right.color == Black
14                 w.color = Red
15                 x = x.p
16
17             else
18                 if w.left.color == Black
19                     w.right.color = Black
20                     w.color = Red
21                     Left-Rotate(T,w)
22
23                     w = x.p.left
24
25                 w.color = x.p.color
26                 x.p.color = Black
27                 w.left.color = Black
28                 Right-Rotate(T,x.p)
29
30                 x = T.root
31         else
```

```
32              // right and left exchanged
33      x.color = Black
34  }
```

Kao što smo rekli, čvor `x` proslijeđen ovoj metodi bit će dijete izbrisanog crnog čvora. Prvi slučaj, kada je taj čvor crven, rješava se u zadnjoj liniji ove metode. U petlji `while` provjeravat će se i izvršavati svi ostali slučajevi i to obrnutim redoslijedom kojim smo ih naveli. Prvo provjeravamo istinitost 5. slučaja u liniji 6 tako da ga, ako vrijedi, transformiramo u jedan od preostala 3 slučaja, koja odmah zatim provjeravamo. Ako vrijedi 4. slučaj, u liniji 15 promijenit ćemo vrijednost čvora `x` i, ako mu je boja crna, petlja `while` će se ponovo izvesti. Ako ne vrijedi 4. slučaj, onda ćemo prvo, u liniji 18, provjeriti vrijedi li 3. slučaj. Ako vrijedi, transformiramo ga u 2. koji je implementiran u linijama 25 - 28. Rekli smo da ovaj slučaj popravlja peto svojstvo i ne stvara ga nigdje drugdje, pa moramo izaći iz petlje `while`. To radimo linijom 30.

2.3.3 Analiza

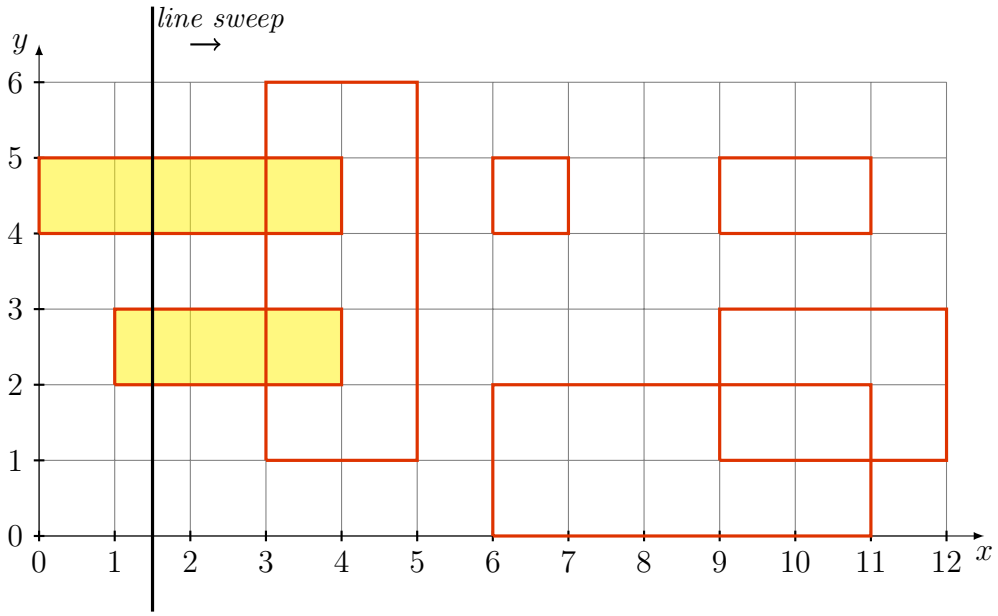
Izračunajmo koliko traje ova operacija. Metoda `Delete` će najduže trajati ako bude morala pozvati metodu za traženje suksesora. To traženje je, u najgorem slučaju, jednako visini stabla, odnosno $O(\log n)$. Petlja `while` u metodi `Delete-Fixup` izvodit će se dok god je istinit 4. slučaj. Duplo crna boja pomiče se na roditelja trenutnog čvora, tako da se i ova petlja, u najgorem slučaju, izvodi $O(\log n)$. U svim ostalim slučajevima petlja staje. Također, slučajevi 2, 3 i 5 će izvesti maksimalno 3 rotacije. Vremenska složenost brisanja iz crveno-crnog stabla jednaka je $O(\log n)$.

Poglavlje 3

Opis projekta Red Black Trees

3.1 Opis problema i njegovo rješenje

Projekt Red Black Trees rješava pitanje postojanja presjeka među n pravokutnika čije su stranice paralelne koordinatnim osima. Naivno rješenje ovog problema bila bi provjera po svim parovima pravokutnika što bi trajalo $O(n^2)$. Mi ćemo dati bolje rješenje i objasniti primjenu crveno-crnog stabla u njemu.



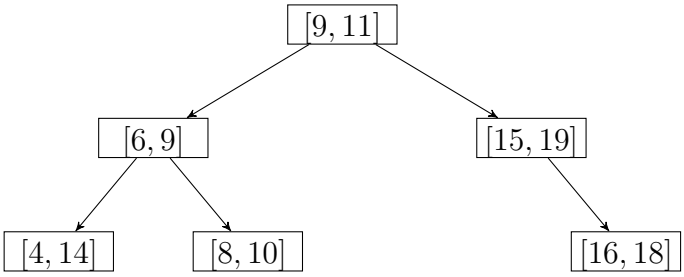
Slika 3.1: *line sweep* proces

Rješenje problema započinjemo *line sweep* metodom - zamišljena linija kreće se s lijeva na desno te kreira i modificira set aktivnih pravokutnika koji se mijenja na svakom novom događaju. Događaj se dogodi svaki put kada linija dotakne neku od vertikalnih stranica pravokutnika. Ako je vertikalna stranica lijeva stranica nekog pravokutnika, tada se upravo taj pravokutnik dodaje u aktivni set. Inače, ako je linija

dotakla desnu stranicu nekog pravokutnika, tada taj pravokutnik mičemo iz aktivnog seta. Kada linija dotakne lijevu stranicu pravokutnika, prije samog njegovog ubacivanja u aktivni set, htjet ćemo provjeriti da li u aktivnom setu postoji pravokutnik koji se presijeca s njim (jer ako postoji, tada smo gotovi). Dakle, nad strukturom podataka u kojoj ćemo čuvati pravokutnike radit ćemo operacije dodavanja, brisanja i traženja. Kako se u aktivnom setu moraju nalaziti pravokutnici koje linija presijeca, svi ti pravokutnici dijele iste x koordinate do sljedećeg događaja, što bi značilo da u aktivnom setu ne moramo čuvati "cijele" pravokutnike, već samo intervale njihovih y vrijednosti.

Promotrimo mogućnost čuvanja tih intervala u crveno-crnom stablu. Neka se za oznaku čvora uzima donja granica intervala pohranjenog u njemu. To znači da ćemo u operaciji dodavanja novog čvora s intervalom $[a, b]$ uspoređivati a s oznakama ostalih čvorova u stablu.

Razmislimo sad o operaciji koja rješava ovaj geometrijski problem, a to je traženje intervala koji ima presjek s intervalom koji je u trenutnom događaju. Naravno da takvih intervala može biti i više, ali nas samo zanima da li postoji bar jedan. Kada bi se i u ovoj operaciji odlučivali za lijevo ili desno podstablo na temelju donje granice intervala, mogli bi zaključiti da ne postoji presjek i biti u krivu. To, na primjer, možemo vidjeti u sljedećem stablu tražeći interval koji se siječe s intervalom $[12, 14]$. Jer intervali $[9, 11]$ i $[12, 14]$ nemaju presjeka te je $9 < 12$, otišli bi u desno podstablo. Istom analizom tu bi se odlučili otići u lijevo postablo i došli do *nil* čvora te bismo zaključili da u ovom stablu ne postoji interval koji se siječe s intervalom $[12, 14]$. Međutim, interval $[4, 14]$ se siječe. Crveno-crno stablo mora se proširiti.



Slika 3.2: Primjer binarnog stabla s intervalima

Neka svaki čvor ima podatak o najvećoj gornjoj granici u stablu kojem je on korijen. Pretraživanje možemo raditi na sljedeći način:

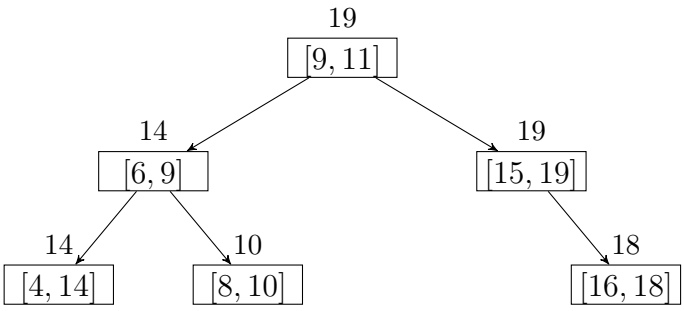
1. Ako trenutni čvor sadrži interval koji ima presjek s upitnim ili je to *nil* čvor, tada vraćamo taj čvor.

2. Inače, ako je najveća gornja granica u stablu čiji je korijen lijevo dijete trenutnog čvora manja od donje granice upitnog intervala, pretraživanje nastavljamo u desnom podstablu čvora.

3. Inače, pretražujemo lijevo podstablo.

Na ovaj način se ne može dogoditi da "propustimo" presjek. Jer, ako smo otišli u desno podstablo, to znači da je najveća gornja granica intervala u lijevom podstablu manja od donje granice našeg intervala $[x, y]$, a kako se svi intervali u lijevom podstablu sigurno nalaze "ispod" te najveće gornje granice, tada sigurno u lijevom podstablu nema presjeka s našim intervalom. Ako smo, pak, otišli u lijevo podstablo, tada postoji interval $[a, b]$, gdje je b najveća gornja granica u tom podstablu i vrijedi $b >= x$. Pretpostavimo da u ovom stablu nema presjeka. To znači da je $y < a$, a kako se a sigurno nalazi "ispod" svih intervala u desnom podstablu, naš interval nema presjeka ni u tom podstablu.

U stablu s prethodnog primjera, lijevo dijete korijena za novi podatak ima vrijednost 14, što je veće od 12. Zbog toga ćemo se odlučiti lijevo podstablo. Tu, u čvoru s intervalom $[6, 9]$, istom provjerom ponovo ćemo se odlučiti za lijevo podstablo i doći do čvora s intervalom $[4, 14]$. Ovaj interval ima presjek s intervalom $[12, 14]$, stoga tu stajemo.



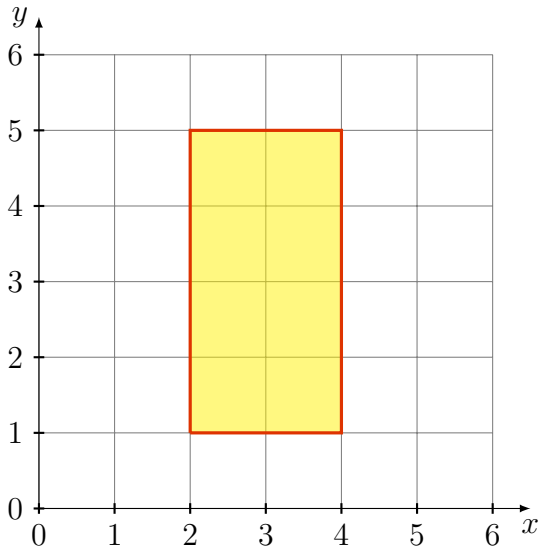
Slika 3.3: Primjer binarnog stabla s intervalima i oznakom najveće gornje granice kod svakog čvora

Proširivanjem crveno-crnog stabla s čuvanjem najveće gornje granice u svakom čvoru nismo ugrozili vremenske složenosti operacija nad njim, međutim nakon svake operacije dodavanja, rotacije i transplantacije, mora se ažurirati to novo svojstvo. U rotaciji se to svojstvo mora ažurirati samo za čvor oko kojeg se rotira te za čvor koji će prvom postati roditelj, pa to traje $O(1)$. Kod dodavanja se moraju ažurirati čvor roditelj upravo dodanom čvoru u stablo te svi čvorovi iznad njega u stablu do korijena. Ista stvar je i u transplantaciji - ažuriranje se događa od čvora koji je bio roditelj jednom, a postaje roditelj drugom od čvorova koji sudjeluju u transplantaciji. Ažuriranje se, također, mora obaviti na putu od čvora koji je prije transplantacije bio roditelj drugom čvoru do korijena. Ova ažuriranja po putu od nekog čvora do korijena mogu trajati najviše visinu stabla, što bi značilo da i ova operacija ima vremensku složenost $O(\log n)$.

Koliko ukupno traje rješenje ovog problema? Kao što smo rekli, operacije nad aktivnim setom pravokutnika radimo na svakom događaju, odnosno na svakoj vertikalnoj stranici pravokutnika. Te vertikalne stranice trebamo prvo sortirati po x-koordinati što možemo napraviti u vremenu $O(n \log n)$. Kako tih stranica ima ukupno $2n$, a za strukturu aktivnog seta smo odabrali crveno-crno stablo nad kojim će potrebne operacije trajati $O(\log n)$, možemo zaključiti da će vremenka složenost ovakvog rješenja biti $O(n \log n)$.

3.2 Programska realizacija

Projekt je rađen u programu Xcode i pisan u objektno orjentiranom jeziku Objective C. Aplikacija je *command-line* tipa te korisniku omogućava da unese n pravokutnika za koje će mu ispisati postoji li presjek među njima. Korisnik prvo mora unijeti ukupan broj pravokutnika koje zatim unosi tako da za svaki upiše četiri cijela broja: a, b, c i d. Brojevi a i b trebaju predstavljati x-koordinatu lijeve i desne stranice, a c i d y-koordinatu donje i gornje stranice pravokutnika. Na primjer, za pravokutnik s donje slike korisnik treba unijeti redom brojeve: 2, 4, 1, 5.



Za svaka takva četiri unesena broja kreira se novi objekt tipa `Rectangle` te se pridružuju redom njegovim svojstvima `xMin`, `xMax`, `yMin` i `yMax`. Svaki taj objekt dodajemo u polje `rectangles`.

Programska realizacija *line sweep* metode bit će prolaz po polju `events`. U tom polju nalaze se objekti tipa `Event` jednoznačno određenih x-koordinatom i pravokutnikom. Te objekte prethodno smo kreirali u petlji `for` po svim pravokutnicima za x-koordinate i lijeve i desne stranice te ih na kraju sortirali po x-koordinati.

```
1 void createEvents() {
2     events = [[NSMutableArray alloc] init];
```

```
3
4       for (Rectangle *rect in rectangels) {
5           [events addObject:[[Event alloc] initWithCoordinate:rect.xMin
                andRectangle:rect]];
6           [events addObject:[[Event alloc] initWithCoordinate:rect.xMax
                andRectangle:rect]];
7       }
8
9       [events sortUsingComparator:^NSComparisonResult(Event *obj1, Event
            *obj2) {
10          return obj1.coordinate > obj2.coordinate;
11      }];
12  }
```

Metoda `(void)sortUsingComparator:(NSComparator)cmptr` Objective C je metoda nad objektom tipa `NSMutableArray` i radi po principu Quick sorta, što znači da polje veličine n sortira u $O(n \log n)$.

Opišimo i implementaciju samog *line sweep* procesa.

```
1       BOOL overlapping = NO;
2       for (Event *event in events) {
3           Interval *interval = [[Interval alloc]
                initWithMin:event.rectangle.yMin andMax:event.rectangle.yMax];
4
5           if (event.coordinate == event.rectangle.xMax) {
6               [activeIntervals deleteInterval:interval];
7
8           } else {
9               if ([activeIntervals
                    containsIntervalOverlappingInterval:interval]) {
10                  overlapping = YES;
11                  break;
12              } else {
13                  if (event.coordinate == event.rectangle.xMin) {
14                          [activeIntervals insertInterval:interval];
15                  }
16              }
17          }
18      }
```

Kao što smo već rekli, prolazimo kroz polje događaja. U svakoj iteraciji kreiramo interval od najmanje i najveće vertikalne vrijednosti pravokutnika (linija 3). U liniji 5 provjeravamo predstavlja li događaj desnu stranicu pravokutnika. Ako je tako, tada sljedeća linija poziva metodu `(void)deleteInterval:(Interval *)interval` nad objektom `activeIntervals` u kojem čuvamo aktivne intervale. Ta metoda će naći i izbrisati proslijeđen joj interval. Inače, ako je događaj lijeva stranica pravokutnika,

provjeravamo sadrži li `activeIntervals` interval s kojim se preklapa ovaj kreiran u iteraciji. Ako da, tada smo gotovi i možemo izaći iz petlje. Aplikacija će korisniku javiti da postoji presjek. Inače, linija 14 nad `activeIntervals` poziva metodu `(void)insertInterval:(Interval *)interval` koja će taj interval dodati u aktivne.

Opišimo sad objekt `activeIntervals`. Kao što smo rekli, apstraktna struktura podataka u kojoj ćemo čuvati aktivni set intervala je crveno-crno stablo. To stablo programski ćemo realizirati tipom `RBTree`. To je klasa koja u svom sučelju pruža metode dodavanja i brisanja intervala te provjeru postojanja intervala koji se siječe s proslijeđenim. Kao privatno svojstvo svaki objekt ove klase ima pokazivač na `root`, koji je tipa `RBNode` i predstavlja korijen stabla. `RBNode` će biti tip objekata kojima ćemo realizirati čvorove.

```
1  @interface RBNode : NSObject
2
3  @property (nonatomic, strong) Interval *interval;
4  @property (nonatomic, assign) int treeMax;
5  @property (nonatomic, assign) NodeColor color;
6  @property (nonatomic, strong) RBNode *left;
7  @property (nonatomic, strong) RBNode *right;
8  @property (nonatomic, strong) RBNode *parent;
9
10 @end
```

Svojstva `left`, `right` i `parent` bit će pokazivači na objekte istog tipa, a predstavljaju lijevo i desno dijete te roditelja čvora. Čvor crveno-crnog stabla mora biti crvene ili crne boje, stoga će `color` biti tipa `NodeColor`, enum čije su vrijednosti `Black` i `Red`. Glavna vrijednost koju čuvamo u čvorovima je `interval`. Onaj podatak koji nam rješava traženje, a koji predstavlja najveću gornju granicu u stablu kojem je korijen trenutni čvor, bit će spremljen u `treeMax`. Vrijednost tog podatka možemo računati kao maksimum skupa koji sadrži iste te podatke lijevog i desnog djeteta te gornju granicu intervala, odnosno programski:

```
1  self.treeMax = MAX(self.interval.max, MAX(self.left.treeMax,
       self.right.treeMax));
```

Na početku ovog rada, u poglavlju Definicija, spomenuli smo i objasnili *sentinel* čvor. Međutim, nismo rekli i čemu on zapravo služi. Pretpostavimo da su `left` i `right` nekog `RBNode` objekta jednaki *nil* te da `interval` istog tog objekta za `max` ima negativnu vrijednost. Gornji izračun za `treeMax` dao bi nulu za naš objekt (jer je vrijednost `int` svojstva *nil* objekta jednaka nuli), pa bismo u traženju presjeka možda i pogriješili. To znači da smo trebali provjeriti postojanje objekata `left` i `right`. *Sentinel* čvor nam služi da upravo tu provjeru ne moramo raditi. Svaki

`RBTree` objekt imat će, uz `root`, još jedan objekt tipa `RBNode`, a predstavljat će taj *sentinel* čvor. Pri inicijalizaciji `RBTree` objekta upravo taj objekt ćemo kreirati i njegovim svojstvima pridružiti neke vrijednosti.

```
1 - (instancetype)init {
2     if (self = [super init]) {
3         self.sentinel = [[RBNode alloc] init];
4         self.sentinel.color = Black;
5         self.sentinel.treeMax = INT_MIN;
6         self.root = self.sentinel;
7     }
8     return self;
9 }
```

Kao što smo također rekli u poglavlju Definicija, boja ovog čvora je crna, što radimo u liniji 4. Linija 5 će mu za `treeMax` postaviti najmanju `int` vrijednost. Pošto ćemo upravo ovaj objekt postavljati kao vrijednost od `left` i `right` onom objektu koji predstavlja čvor bez djece, izračun za `treeMax` sad će dobro raditi.

Implementacije dodavanja i brisanja intervala, odnosno svih metoda vezanih uz to (rotacije, popravak nakon dodavanja, popravak nakon brisanja) prate pseudokôdove dane u poglavlju Operacije nad crveno-crnim stablima. Međutim, nakon ovih operacija moramo ažurirati i `treeMax`, svojstvo koje predstavlja programsku realizaciju onog svojstva kojim smo proširili čvor crveno-crnog stabla za potrebe rješavanja našeg problema. U prethodnom poglavlju smo rekli kad se točno treba ažurirati to svojstvo, pa smo i programski to napravili na istim mjestima. Također smo objasnili kako to napraviti. Dajemo sada i implementaciju te operacije kada se trebaju ažurirati svi čvorovi u stablu iznad jednog određenog.

```
1 - (void)fixTreeMaxFromNode:(RBNode *)node {
2     while (node != self.sentinel) {
3         node.treeMax = MAX(node.interval.max, MAX(node.left.treeMax,
              node.right.treeMax));
4         node = node.parent;
5     }
6 }
```

Metoda prima `RBNode *node` od kojeg će započeti ažuriranje. Samo ažuriranje se odvija u liniji 3 unutar petlje `while`, a nakon toga novi `node` postaje objekt na koji pokazuje njegovo svojstvo `parent`. Petlja se vrti dok god `node` ne postane `sentinel`.

Kako se tijekom *line sweep* procesa, ako je događaj desna stranica pravokutnika, aktivnom setu proslijeđuje interval koji se treba obrisati, klasa `RBTree` treba implementirati metodu koja traži taj interval i vraća objekt tipa `RBNode` koji ga sadrži.

Cijeli taj objekt proslijeđuje se kasnije metodi koja ga briše. Dohvat tog objekta koji sadrži `Interval *interval` radimo pozivom `[self searchInterval:interval inSubtree:self.root]` nad objektom klase `RBTree`.

```
1 - (RBNode *)searchInterval:(Interval *)interval inSubtree:(RBNode
    *)subtreeRoot {
2     if (subtreeRoot == self.sentinel || (subtreeRoot.interval.min ==
          interval.min && subtreeRoot.interval.max == interval.max)) {
3         return subtreeRoot;
4     } else if (subtreeRoot.interval.min < interval.min) {
5         return [self searchInterval:interval inSubtree:subtreeRoot.right];
6     } else {
7         return [self searchInterval:interval inSubtree:subtreeRoot.left];
8     }
9 }
```

U liniji 2 provjeravamo jesu li `min` i `max` proslijeđenog intervala jednaki onima intervala trenutnog `RBNode` objekta. Ako jesu, tada je taj trenutni objekt upravo onaj kojeg ćemo brisati. Također, ako je trenutni objekt jednak objektu `sentinel`, to znači da nema traženog intervala, stoga vraćamo taj objekt i izlazimo iz rekurzije. Ako proslijeđeni interval, pak, ima `min` veći od onog u intervala trenutnog `RBNode` objekta (linija 4), u sljedećem rekurzivnom pozivu provjeravat ćemo objekt koji taj trenutni ima za vrijednost svojstva `right` (linija 5). Inače, u liniji 7 provjeravamo onaj iz svojstva `left`.

Dajmo još i implementaciju metode koja rješava pitanje postojanja presjeka.

```
1 - (RBNode *)searchIntervalOverlappingInterval:(Interval *)interval
    inSubtree:(RBNode *)subtreeRoot {
2     if (subtreeRoot == self.sentinel || [subtreeRoot.interval
          overlapsWithInterval:interval]) {
3         return subtreeRoot;
4     } else if (subtreeRoot.left.treeMax < interval.min) {
5         return [self searchIntervalOverlappingInterval:interval
              inSubtree:subtreeRoot.right];
6     } else {
7         return [self searchIntervalOverlappingInterval:interval
              inSubtree:subtreeRoot.left];
8     }
9 }
```

Metoda je slična upravo opisanoj za točno traženje intervala. Razlike su u linijama 2 i 4. U liniji 4, do koje ćemo doći ako ne vrijedi uvjet iz linije 2, radimo onu

provjeru koja odlučuje da li ćemo dalje u lijevo ili desno stablo. U liniji 2 nam uvjet sada nije da su `min` i `max` intervala jednaki, već provjeravamo da li intervali imaju presjek. To radimo pozivom metode `(BOOL)overlapsWithInterval:(Interval *)interval` definirane na klasi `Interval`.

```
1 - (BOOL)overlapsWithInterval:(Interval *)interval {
2     if (self.max < interval.min || self.min > interval.max) {
3         return NO;
4     }
5     return YES;
6 }
```

Znamo da se dva intervala $[a, b]$ i $[c, d]$ ne presijecaju ako je $b < c$ ili $a > d$. Inače se presijecaju. Ova metoda vraća NO ako se interval `self` ne siječe s proslijeđenim intervalom `interval`. Inače, vraća YES. Poziv `[self searchIntervalOverlappingInterval:interval inSubtree:self.root]` vratit će objekt tipa `RBNode`. Ako je on jednak `sentinel` objektu, tada možemo zaključiti da nema presjeka u trenutnom aktivnom setu intervala. Inače, postoji presjek.

Zaključak

Ovim radom opisano je crveno-crno stablo kao dobro balansirano binarno stablo pretraživanja. Njegova crveno-crna svojstva osiguravaju da osnovne operacije nad njim imaju vremensku složenost jednaku $O(\log n)$ radi čega se kao apstraktna struktura često koristi za organiziranje skupova usporedivih podataka.

Osim toga, projektom, koji je napravljen uz rad, dokazali smo da se takvo stablo lako može proširiti za potrebe rješavanja problema, a da se pri tom ne ugroze vremenske složenosti operacija nad njim te da on zadrži svoja osnovna svojstva. Osim problema promatranog u ovom radu, takva stabla korisna su i u rješenjima mnogih drugih geometrijskih problema.

Crveno-crno stablo nije jedina vrsta samo-balansirajućeg binarnog stabla koji za sve osnovne operacije ima vremensku složenost $O(\log n)$. Međutim, u odnosu na druga takva stabla, operacije dodavanja i brisanja kod njega su jednostavnije, što je vrlo vjerojatno veliki razlog zašto se u implementaciji mnogih struktura za pohranu podataka, koje smo spomenuli u Uvodu, odabralo baš njega.

Bibliografija

[1] Peter Brass: *Advnaced Data Structures*, Cambridge University Press, 1993.
http://212.50.14.233/Knowledge/Computing%20%26%20Games/0_Computer%20Science/2_Algorithms/Advanced%20Data%20Structures.pdf

[2] Frank Pfenning: *Lecture Notes on Red/Black Trees* [*15-122: Principles of Imperative Computation*], 2010.
https://www.cs.cmu.edu/ fp/courses/15122-f10/lectures/17-rbtrees.pdf

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms, 3rd edition*, The MIT Press, 2009.
http://faculty.mu.edu.sa/public/uploads/1360957074.1109Introduction_to_Algorithms_Third_Edition.pdf

[4] Robert Manger: *Strukture podataka i algoritmi*, Element, Zagreb, 2014.

[5] Saša Singer: *Složenost algoritama*, Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, Zagreb, 2005.
https://web.math.pmf.unizg.hr/ singer/oaa/skripta/00.pdf

Sažetak

U ovom radu prezentirali smo crveno-crno stablo kao vrstu binarnog stabla pretraživanja s dobrom balansiranošću.

U prvom poglavlju dali smo osnovnu definiciju crveno-crnog stabla i naveli crveno-crna svojstva koja takvo stablo mora zadovoljavati. Pokazali smo da ta svojstva osiguravaju da nijedan put od korijena do lista nije više od dva puta dulji od bilo kojeg drugog puta od korijena do lista te dokazali da zato takvo stablo s n čvorova ima visinu $O(\log n)$.

U sljedećem poglavlju bavili smo se operacijama nad crveno-crnim stablom. Operaciju pretraživanja nismo opisivali jer je jednaka onoj za općenita stabla. Ukratko smo opisali i analizirali trajanje rotacija koje smo, uz mijenjanje boja, najčešće koristili u "popravljanju" crveno-crnog stabla. Veći značaj dali smo operacijama dodavanja i brisanja. Obje operacije baziraju se na istim operacijama definiranim za općenito binarno stablo, ali smo vidjeli da se mogu kršiti svojstva crveno-crnog stabla. Skicama smo prikazali takve slučajeve i kako ih riješiti. Također smo dali pseudokôdove tih operacija i analizirali njihovu vremensku složenost.

U zadnjem poglavlju opisali smo projekt Red Black Trees koji je napravljen u sklopu ovog rada i koji implementira rješenje jednog geometrijskog problema uz pomoć crveno-crnog stabla. Objasnili smo kako se pokreće i koristi projekt te dali i opisali dijelove implementacije.

Summary

In this thesis we have presented the red-black tree as a type of binary search tree with good balance.

In the first chapter we have given the definition of red-black trees and listed red-black properties that such tree must satisfy. We have shown that these properties ensure that no path from the root to a leaf is not more than two times longer than any other path from the root to a leaf and proved that therefore such tree with n nodes has a height of $O(\log n)$.

In the next chapter we have dealt with the operations of the red-black tree. We have not described search operation because it is similar to that of a general tree. We have described and analyzed the duration of the rotation that has been most often used to "repair" the red-black tree, along with changing colors. We have put more emphasis on insertion and deletion. Both operations are based on the same operations defined for generally binary tree, but we have seen that they can violate the properties of red-black tree. We have presented these cases with sketches and described how to solve them. We have also given pseudocode for these operations and analyzed their time complexity.

In the last chapter we have described the project Red Black Trees that was made as part of this work and that implements the solution of one geometric problem with red-black tree. We have explained how to run and use the project, and we have given and described parts of implementation.

Životopis

Rođena sam 23.5.1992. godine u Zadru. U Osnovnoj školi Stanovi završila sam osnovno obrazovanje tijekom kojeg sam nekoliko puta sudjelovala u natjecanjima iz gramatike hrvatskog jezika. 2006. godine upisala sam opći smjer u Gimnaziji Jurja Barakovića te nekoliko puta sudjelovala u natjecanjima iz matematike. Paralelno s osnovnom i srednjom školom pohađala sam i Glazbenu školu Blagoje Bersa. Sudjelovala sam na dva međunarodna klavirska natjecanja i oba dva puta osvojila 2. nagradu.

2010. godine upisala sam preddiplomski sveučilišni studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. Isti sam završila 2013. godine te sam odmah upisala diplomski sveučilišni studij Računarstvo i matematika na istom fakultetu. Na ljeto 2015. godine pohađala sam Infinum Student Academy, tečaj programiranja softverske tvrtke Infinum. Tečaj sam završila za softversku platformu iOS. U jesen iste godine počela sam i raditi u Infinumu te tu i danas razvijam aplikacije za iPhone i iPad.