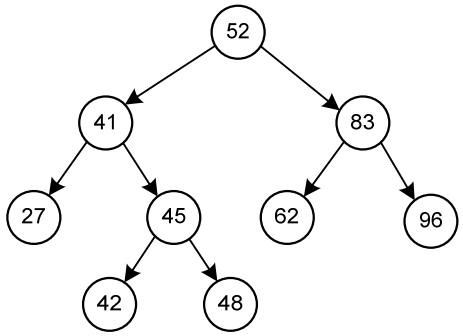
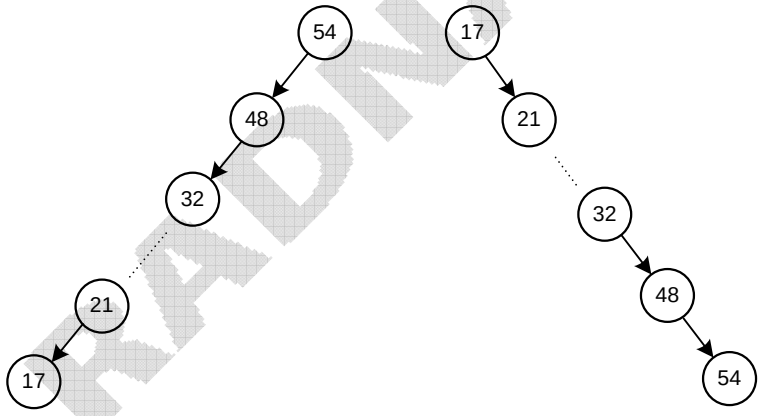
Balansirana stabla

U prethodnom poglavlju smo mogli da vidimo koje su osnovne karakteristike binarnih stabala za pretraživanje i da se upoznamo sa osnovnim operacijama nad ovom vrstom stabala. Ključna stvar kod formiranja binarnih stabala za pretraživanje je svakako dodavanje novih elemenata. Postupak dodavanja novih elemenata mora biti izveden tako da se pri umetanju novog elementa ne naruši osnovni kriterijum koje binarno stablo za pretraživanje mora da zadovolji, a to je da za svaki čvor stabla mora da važi da su svi elementi manji od njega u njegovom levom podstablu, a svi ostali u njegovom desnom podstablu (Slika ###). Naravno, prilikom brisanja čvora iz stabla, takođe je potrebno voditi računa o tome da se ovaj poredak očuva.



Slika ### Binarno stablo za pretraživanje

Ovakva struktura binarnog stabla obezbeđuje pretraživanje čvorova sa kompleksnošću $O(\log n)$. Međutim, u zavisnosti od rasporeda čvorova u binarnom stablu, ovakav način predstavljanja podataka može biti vrlo nepovoljan. Ukoliko bi se, na primer, čvorovi dodavali u stablo od najvećeg ka najmanjem, dogodilo bi se da svi čvorovi imaju samo levu granu, kao što je prikazano na Slici ###. Slično bi se dogodilo i u slučaju dodavanja čvorova od najmanjeg ka najvećem, samo što bi u tom slučaju svi čvorovi imali samo desnu granu.



Slika ### Nepovoljni oblici binarnih stabala

Jasno je da bi se u ovakvim slučajevima pretraživanje stabla praktično svelo na pretraživanje liste, što znači da bi kompleksnost algoritma postala $O(n)$. Pored ovih ekstremnih slučajeva, slični problemi se javljaju i kod svih ostalih stabala velike dubine sa malo grananja.

Iz tog razloga neophodno je stablo izbalansirati tako da levo i desno podstablo svakog čvora budu približno iste veličine. Takva struktura obezbediće maksimalan broj grananja u stablu, čime se postiže minimalan broj provera prilikom pretraživanja.

Stablo balansirano po dubini

Stablo balansirano po dubini je binarno stablo kod koga za svaki čvor važi da je apsolutna vrednost razlike dubina njegovih podstabala najviše 1. Kod ovakvih stabala dubina stabla veoma sporo raste sa povećanjem broja čvorova, pa je samim tim i pretraživanje brže, s obzirom da je maksimalan broj provera ograničen dubinom stabla.

Ovakva stabla se nazivaju još i **AVL stabla**, po njihovim pronalazačima (G.M. **A**delson-**V**elsky and E.M. **L**andis).

Realizacija stabla balansiranog po dubini

Da bi bilo moguće efikasno realizovati koncept balansiranog stabla, neophodno je da u svakom čvoru postoji i informacija o razlici dubina njegovih podstabala. Sada bi struktura koja predstavlja čvor stabla mogla da izgleda ovako:

```
struct cvor
{
   int podatak;
   int balans;
   struct cvor *levi,*desni;
}
```

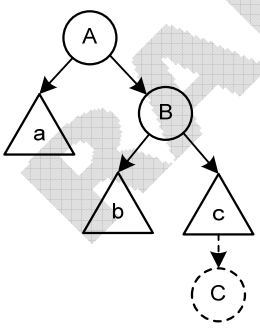
pri čemu promenljiva *balans* predstavlja razliku dubina desnog i levog stabla, odnosno

$$balans = Dubina(desni) - Dubina(levi)$$

To praktično znači da promenljiva balans može imati 3 vrednosti:

- -1 ukoliko je dubina levog podstabla za jedan veća od dubine desnog podstabla
- 0 ukoliko oba podstabla imaju istu dubinu
- 1 ukoliko je dubina desnog podstabla za jedan veća od dubine levog podstabala

Da bi se u svakom trenutku očuvala izbalansiranost stabla, neophodno je napraviti funkciju za dodavanje novih čvorova, koja će voditi računa o tome da balans u svakom čvoru ostane u granicama -1 do 1. Međutim, prilikom dodavanja novog čvora u balansirano stablo, može se desiti da dubina jednog podstabla postane za dva veća od dubine drugog podstabla. To bi se dogodilo u slučaju da se čvor dodaje baš u ono podstablo koje je već imalo dubinu za jedan veću od drugog podstabla (Slika ###). U tom slučaju neophodno je izvršiti određena pregrupisavanja u stablu, kako bi se poremećeni balans vratio u granice od -1 do 1.

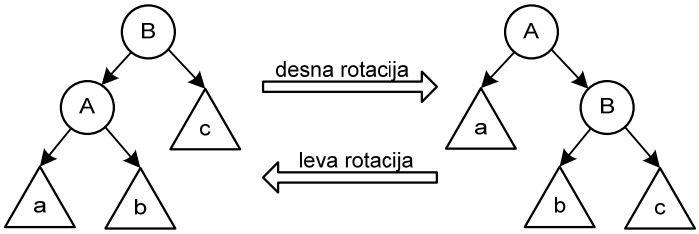


Slika ### Dodavanje novog čvora *C* remeti izbalansiranost stabla

Rotacija u binarnim stablima

Rotacija u binarnim stablima je operacija koja menja strukturu stabla, bez narušavanja leksikografskog poretka u njemu. Ovakve operacije mogu biti veoma korisne u održavanja stabla izbalansiranim. U zavisnosti od toga da li se povećava dubina levog ili desnog podstabla, rotacija može biti **leva rotacija** i **desna rotacija**.

Leva rotacija podrazumeva pregrupisavanje čvorova tako da se povećava dubina levog podstabla, kao što je prikazano na Slici ###. Suprotno, desna rotacija je pregrupisavanje čvorova koje dovodi do povećanja dubine desnog podstabla.



Slika ### Leva i desna rotacija

Funkcija koja bi izvršila levu rotaciju u čvoru *t* bi mogla da izgleda ovako:

```
void lrotacija(struct cvor **t)
{
   struct cvor *poml,*pomd;

   poml = *t;
   pomd = poml->desni;
   poml->desni = pomd->levi;
   pomd->levi = poml;

   *t=pomd;
}
```

Funkcija za desnu rotaciju u čvoru *t* bi mogla izgledati ovako:

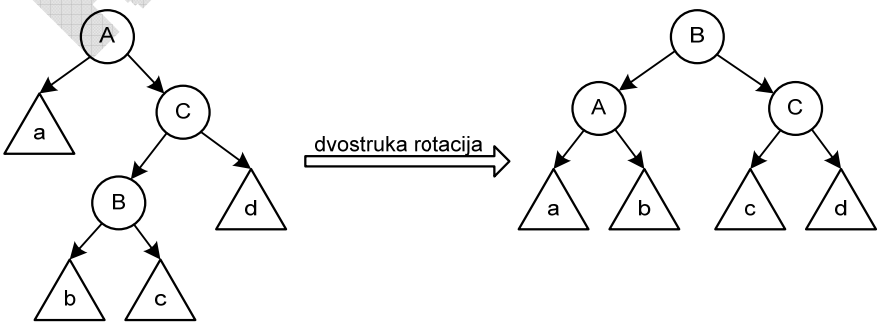
```
void drotacija(struct cvor **t)
{
   struct cvor *poml,*pomd;

   pomd = *t;
   poml = pomd->levi;
   pomd->levi = poml->desni;
   poml->desni = pomd;

   *t=poml;
}
```

Dvostruke rotacije

Nešto složenije transformacije jesu dvostruke rotacije, koje takođe mogu biti leva i desna. Na Slici ### prikazana je **dvostruka leva rotacija**.



Slika ### Dvostruka leva rotacija

Transformacija simetrična dvostrukoj levoj rotaciji je **dvostruka desna rotacija**.

Obe ove rotacije mogu se predstaviti pomoću dve jednostruke rotacije. Tako se, na primer, dvostruka leva rotacija u čvoru *t* može predstaviti kao

```
DesnaRotacija(t->desni); LevaRotacija(t);
```

a dvostruka desna rotacija kao

```
LevaRotacija(t->levi); DesnaRotacija(t);
```

Rekonstrukcija dubine

Prilikom vršenja rotacije u nekom čvoru binarnog stabla dolazi do promene dubina njegovih podstabala, pa je neophodno izvršiti ažuriranje informacije o balansu unutar tog čvora, kako bi ona odslikavala stvarno stanje u njegovim podstablima. Iz tog razloga potrebno je u funkcije za levu i desnu rotaciju dodati deo koji će izračunati novonastali balans:

```
void lrotacija(struct cvor **t)
{
   struct cvor *poml,*pomd;

   poml = *t;
   pomd = poml->desni;
   poml->desni = pomd->levi;
   pomd->levi = poml;

   *t=pomd;

   //Azuriranje balansa
   stari_balans = poml->balans;
   poml->balans = stari_balans - 1 - max(pomd->balans,0);
   pomd->balans = min(stari_balans-2, stari_balans+pomd->balans-2, pomd->balans-1);
}
```

gde su *min* i *max* funkcije koje racunaju minimum, odnosno maksimum parametara navedenih unutar zagrada.

Na sličan način se može napisati i funkcija za desnu rotaciju koja će obavljati i ažuriranje balansa:

```
void drotacija(struct cvor **t)
{
   struct cvor *poml,*pomd;

   pomd = *t;
   poml = pomd->levi;
   pomd->levi = poml->desni;
   poml->desni = pomd;

   *t=poml;

   //Azuriranje balansa
   stari_balans = pomd->balans;
   pomd->balans = stari_balans + 1 + min(poml->balans,0);
   poml->balans = max(stari_balans+2, stari_balans+poml->balans+2, poml->balans+1);
}
```

U funkcijama za dvostruku levu i desnu rotaciju nije potrebno pisati koda za ažuriranje balansa iz razloga što se ove rotacije mogu razložiti predstaviti pomoću dve jednostruke rotacije, koje će se pobrinuti da informacija o balansu bude u skladu sa stanjem u stablu.

Dodavanje novog čvora u balansirano stablo

Sada kada smo definisali funkcije za levu i desnu rotaciju, možemo napisati i funkciju za dodavanje novog čvora u balansirano stablo:

```
int dodaj(struct cvor **p, int vrednost)
{
   int inkrement, rezultat;
   struct cvor *t = *p;
```

```
    rezultat = 0;

    if(t==NULL)
    {
       t=(struct cvor *)malloc(sizeof(struct cvor));
       if(t==NULL)
       {
          printf("Greska pri alociranju memorije\n");
          exit(0);
       }
       t->podatak = vrednost;
       t->levi = t->desni = NULL;
       t->balans = 0;
       rezultat = 1;
    }
    else
    {
       if( vrednost > t->podatak ) inkrement=dodaj(&(t->desni),vrednost);
       else inkrement=-dodaj(&(t->levi),vrednost);

       t->balans += inkrement;

       if( inkrement!=0 && t->balans!=0 )
       {
          if( t->balans<-1 )
          {
             if( t->levi->balans<0 ) drotacija(&t);
             else
             {
                lrotacija(&(t->levi)); drotacija(&t);
             }
          }
          else if( t->balans>1 )
          {
             if( t->desni->balans>0 ) lrotacija(&t);
             else
             {
                drotacija(&(t->desni)); lrotacija(&t);
             }
          }
          else rezultat = 1;
       }
    }

    *p = t;

    return(rezultat);
}
```


B-stabla

Stabla kao strukture podataka omogućavaju izvršavanje osnovnih operacija, kao što su pretraživanje, dodavanje, brisanje, određivanje prethodnika sledbenika, minimuma, maksimuma itd, u vremenu koje je proporcionalno dubini stabla. U idealnom slučaju, ukoliko je stablo balansirano, dubina stabla je $\log n$, pri čemu je n broj čvorova u stablu. Da bi se obezbedilo da dubina stabla bude što je moguće manja, koriste se balansirana stabla, kao što je AVL, crveno-crno stablo ili b-stablo.

Kada se radi sa velikim brojem podataka, često nije moguće ili nije poželjno držati čitavu strukturu u primarnoj memoriji (RAM). Umesto toga, relativno mala količina podataka se čuva u primarnoj memoriji, a dodatni podaci se po potrebi učitavaju sa nekog sekundarnog skladišta podataka. Na žalost, magnetni disk, kao najčešće korišćeni uređaj za skladištenje podataka, znatno je sporiji od RAM memorije. Iz tog razloga sistem često troši mnogo više vremena na čitanje podataka nego na njihovu obradu.

B-stabla su balansirana stabla koja su optimizovana za situacije kada deo stabla ili celo stablo moraju da budu smešteni na sekundarno skladište, kao što je hard disk. S obzirom na činjenicu da je pristup disku vremenski skupa operacija, B-stablo pokušava da minimizuje broj pristupanja disku. Na primer, B-stablo dubine 2 i faktorom grananja 1001 može da uskladišti preko milijardu ključeva, ali zahteva najviše dva pristupa disku da bi pronašao bilo koji čvor.

Struktura B-stabla

Za razliku od binarnog stabla, svaki čvor B-stabla može imati promenljiv broj ključeva k_i i potomaka p_i. Ključevi u B-stablu su raspoređeni u neopadajućem redosledu, tako da važi relacija

$$k_1 \le k_2 \le k_3 ... \le k_d$$

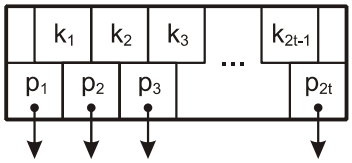
pri čemu je d broj ključeva u čvoru.

Svaki ključ k_i ima pridruženog potomka p_i, koji predstavlja koren podstabla čiji svi čvorovi imaju ključ manji ili jednak posmatranom ključu, i veći od ključa koji prethodi posmatranom ključu. Čvor takođe sadrži i "najdesnijeg" potomka, koji predstavlja koren podstabla čiji svi čvorovi imaju ključ veći od bilo kog ključa u posmatranom čvoru. Relacije između ključeva i odgovarajućih potomaka možemo zapisati kao

$$K(p_1) < k_1 < K(p_2) < k_2 < K(p_3) < k_3 ... < K(p_d) < k_d < K(p_{d+1})$$

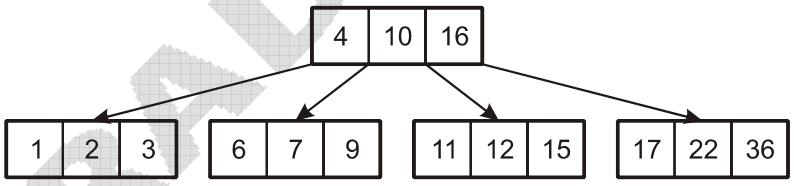
pri čemu je $K(p_i)$ bilo koji ključ u podstablu p_i.

B-stablo ima minimalni dozvoljeni broj potomaka svakog čvora, koji se naziva **faktor minimizacije** ili **minimum grananja**. Ukoliko je *t* minimizacioni faktor, svaki čvor mora imati najmanje *t-1* ključ. Pod određenim uslovima, koren stabla može da naruši ovo ograničenje, tako što će imati manje od *t-1* ključeva. Svaki čvor može imati najviše *2t-1*, odnosno *2t* potomaka (Slika ###).



Slika ### Čvor B-stabla može sadržati najviše *2t-1* ključeva i *2t* potomaka

S obzirom da svaki čvor teži da ima veliki faktor grananja (veliki broj potomaka), da bi se našao određeni ključ, najčešće je potrebno proći kroz mali broj čvorova. Ukoliko pristup svakom čvoru zahteva pristup disku, onda B-stablo minimizuje broj potrebnih pristupanja disku. Faktor minimizacije se obično bira tako da ukupna veličina svakog čvora odgovara celobrojnom umnošku veličine blokova na uređaju za skladištenje podataka. Ovakav izbor pojednostavljuje i optimizuje pristup disku. Samim tim, B-stablo je idealna struktura podataka u situacijama kada se svi podaci ne mogu smestiti u primarnu memoriju, a pristup sekundarnim skladištima podataka je vremenski zahtevan.



Slika ### Primer B-stabla sa minimumom grananja *t=2*

Visina B-stabala

Za *n* veće ili jednako 1, visina B-stabla sa *n* čvorova i minimumom grananja *t*, pri čemu je *t* veće ili jednako 2 je

$$h \le \log_t \frac{n+1}{2}$$

U najgorem slučaju visina stabla je $O(\log n)$. S obzirom da razgranatost B-stabla može biti velika u odnosu na druga balansirana stabla, samim tim se povećava i osnova logaritma, pa je broj čvorova koje je potrebno

obići prilikom pretraživanja manji nego u drugim stablima. Iako ovo ne utiče na visinu u najgorem slučaju, B-stabla teže tome da imaju manju visinu od drugih stabala.

Realizacija B-stabla

Slično kao i u slučaju binarnog stabla, čvor B-stabla se u programskom jeziku C može predstaviti strukturom. Međutim, za razliku od binarnog stabla koje sadrži ključ i pokazivače na levog i desnog potomka, B-stablo može sadržati *2t-1* ključeva i *2t* potomaka, pri čemu je *t* faktor minimizacije, odnosno minimalan broj grananja u čvoru. Struktura koja omogućava čuvanje svih ovih podataka bi mogla da izgleda ovako

```
struct bscvor
{
   int d;                          // Stvaran broj kljuceva u cvoru
   int k[2*MIN_GRAN-1];            // Kljucevi
   struct bscvor *p[2*MIN_GRAN]; // Pokazivaci na podstabla potomke
}
```

Pretraživanje B-stabla

Pretraživanje B-stabla se obavlja tako što se u trenutnom čvoru ispituje da li postoji traženi ključ. Ako traženi ključ postoji, onda je pretraživanje završeno. Ukoliko traženi ključ ne postoji, zahvaljujući poznatim relacijama između ključeva i potomaka, pretraživanje nastavljamo u jednom od potomaka. Podstablo u kome treba nastaviti pretraživanje određujemo tako što u trenutnom čvoru nalazimo prvi ključ k_i čija je vrednost veća od tražene. Ako takav ključ postoji, pretraživanje nastavljamo u podstablu p_i. U suprotnom nastavljamo sa pretraživanjem u podstablu p_{d+1}. Pretraživanje se prekida u slučaju da je pronađen traženi ključ ili ako je dostignuto dno stabla, odnosno ukoliko je posmatrani čvor list.

Funkcija za pretraživanje u B-stablu bi mogla da izgleda ovako:

```
struct bscvor* pronadji(struct bscvor *q, int kljuc, int *indeks)
{
   int i;

   while( q!=NULL )
   {
      for(i=0; i<q->d && kljuc>q->k[i]; i++);

      if( i<q->d && kljuc==q->k[i] )
      {
         *indeks=i;
         return(q);
      }

      q=q->p[i];
   }

   return(NULL);
}
```

Dodavanje ključa u B-stablo

Da bismo pokazali kako se može izvršiti dodavanje novog ključa u B-stablo, definišimo prvo neke pomoćne funkcije, koje ćemo kasnije koristiti.

Kreiranje novog čvora

Prva funkcija koja nam je potrebna je svakako funkcija za kreiranje novog čvora u B-stablu. Prilikom kreiranja čvora neophodno je definisati minimum potrebnih parametara da bi čvor zadovoljavao kriterijume navedene u definiciji B-stabla: jedan ključ i dva potomka.

```
struct bscvor* NoviCvor(int kljuc, struct bscvor *p0, struct bscvor *p1)
{
   struct bscvor *pom;
   pom = (struct bscvor *)malloc( sizeof(struct bscvor) );
```

```
   if( pom==NULL )
   {
      printf("Greska pri alociranju memorije.\n");
      exit(0);
   }

   pom->d = 1;
   pom->k[0] = kljuc;
   pom->p[0] = p0;
   pom->p[1] = p1;

   return( pom );
}
```

Umetanje novog ključa u čvor

Da bismo umetnuli novi ključ u čvor, neophodno je da sve postojeće ključeve veće od novog ključa pomerimo za jedno mesto u desno, kako bismo oslobodili mesto za novi ključ. S obzirom da svaki ključ ima pridruženog potomka, istovremeno sa pomeranjem ključeva, potrebno je pomeriti i odgovarajuće potomke za jedno mesto u desno. Nakon toga, na slobodno mesto upisuje se novi ključ i njegov potomak.

```
void UmetniKljuc(struct bscvor *q, int kljuc, struct bscvor *potomak)
{
   int j;

   for( j=q->d; j>0 && kljuc<q->k[j-1]; j-- )
   {
      q->k[j] = q->k[j-1];
      q->p[j+1] = q->p[j];
   }

   q->d++;
   q->k[j] = kljuc;
   q->p[j+1] = potomak;
}
```

Dodavanje novog ključa u B-stablo

U narednom kodu su date funkcije koje dodaju novi ključ u B-stablo:

```
struct bscvor *NovoStablo;

struct bscvor* Dodaj(struct bscvor *q, int kljuc)
{
   int umetnuti;

   umetnuti = DodajInterno( q, kljuc );

   // provera rasta korena
   if ( umetnuti != NEMA_KLJUCA ) return( NoviCvor( umetnuti, q, NovoStablo) );

   return( q );
};

int DodajInterno( struct bscvor* q, int kljuc )
{
   int i, j;
   int umetnuti;
   struct bscvor* pom;

   if ( q == NULL )
   {
      /* dosegnuto je dno stabla: treba izvrsiti dodavanje */
      NovoStablo = NULL;
      return( kljuc );
   }
   else
   {
      for ( i=0; i<q->d && kljuc>q->k[i]; i++ );

       if ( i<q->d && kljuc==q->k[i] )
      {
         printf("Greska. Kljuc vec postoji.\n");
```

```
            exit(0);
         }
         else
         {
            umetnuti = DodajInterno( q->p[i], kljuc );

            if ( umetnuti != NEMA_KLJUCA ) /* kljuc u "umetnuti" treba umetnuti u trenutni cvor */
               if (q->d < 2*T-1) UmetniKljuc( q, umetnuti, NovoStablo );
               else    /* trenutni cvor treba podeliti */
               {
                  /* kreiranje novog cvora */
                  if ( i<=T-1 )
                  {
                     pom = NoviCvor( q->k[2*T-2], NULL, q->p[2*T-1] );
                     q->d--;
                     UmetniKljuc( q, umetnuti, NovoStablo );
                  }
                  else pom = NoviCvor( umetnuti, NULL, NovoStablo );

                  /* pomeranje kljuceva i pokazivaca na potomke */
                  for ( j=T; j<=2*T-2; j++ )
                     UmetniKljuc( pom, q->k[j], q->p[j+1] );

                  q->d = T-1;
                  pom->p[0] = q->p[T];
                  NovoStablo = pom;
                  return( q->k[T-1] );
               }
         }

         return( NEMA_KLJUCA );
      }
};
```

U prethodnoj funkciji konstanta *T* predsatvlja faktor minimizacije, odnosno minimum grananja. Konstanta NEMA_KLJUCA sadrži neku nemoguću vrednost ključa i predstavlja indikator da nije potrebno umetati novi ključ u čvor.

Funkcija *DodajInterno* dodaje novi ključ u čvor koji joj je prosleđen ili u neki od njegovih potomaka rekurzivnim pozivanjem te iste funkcije. Ova funkcija najpre pokušava pronaći mesto novom ključu među postojećim ključevima u čvoru, tako što kreće od prvog ka poslednjem sve dok ne naiđe na prvi veći ključ. Kada je pronađen ključ koji je veći od ključa koji treba dodati, pokušava se dodavanje novog ključa u podstablo koje odgovara pronađenom ključu. Ako je dodavanje u podstablu bilo uspešno, a funkcija vratila vrednost NEMA_KLJUCA, postupak dodavanja čvora je završen.

Međutim, funkcija *DodajInterno* može imati i drugačije ishode. Ukoliko je funkciji prosleđen NULL pokazivač, to znači da čvor u koji se pokušava dodavanje ne postoji. U tom slučaju funkcija ne radi ništa, već samo vraća vrednost ključa funkciji koja ju je pozvala. Na taj način funkcija koja ju je pozvala će znati da dodavanje u podstablo nije bilo moguće i da se ona mora pobrinuti za taj ključ. Takođe, funkcija je mogla dodati ključ u podstablo, ali se pri tome javila potreba za podelom podstabla na dva dela. U tom slučaju, funkcija će funkciji koja ju je pozvala vratiti ključ koji predstavlja koren dva nova podstabla. Pored toga, ova funkcija će u globalnu promenljivu *NovoStablo* smestiti novostvoreno stablo, koje će kasnije biti pridruženo odgovarajućem ključu.

U slučaju da je funkcija vratila ključ čije je vrednost različita od NEMA_KLJUCA, to je znak da taj ključ treba umetnuti u trenutni čvor. Ukoliko je broj postojećih ključeva u čvoru manji od *2T-1*, umetanje se vrši jednostavnim pozivanjem funkcije *UmetniKljuč*, pri čemu se kao potomak prosleđuje *NovoStablo*. Međutim, ukoliko je broj ključeva u čvoru dostigao maksimalnu dozvoljenu vrednost, taj čvor je neophodno podeliti u dva čvora, pri čemu treba voditi računa da minimalni broj ključeva koji ostanu u podstablima ne bude manji od *T-1*. Ako je ključ potrebno dodati u levu polovinu čvora, onda se od poslednjeg ključa i njegovog desnog potomka formira pomoćni čvor, a tek onda ključ umeće na odgovarajuće mesto. U suprotnom od ovog ključa formira se pomoćni čvor, kome je desni potomak *NovoStablo*.

Nakon ovoga, svi ključevi sa desne polovine se prebacuju u pomoćni čvor, tako da broj ključeva u trenutnom čvoru postaje jednak *T-1*. Takođe, središnji potomak postaje prvi potomak pomoćnog čvora, a *NovoStablo* postaje jednako pomoćnom čvoru. Konačno, funkcija vraća funkciji koja ju je pozvala vrednost

najvećeg preostalog ključa, kako bi se ona pobrinula za njegovo umetanje i povezala ga sa stablom *NovoStablo*. Na taj način izvršena je podela čvora na dva dela i obaveštavanje roditelja o nastaloj podeli.

Međutim, ukoliko dođe do podele korena stabla, nije moguće obavestiti roditelja, jer on i ne postoji. Iz tog razloga neophodno je da postoji funkcija *Dodaj*, koja će se pobrinuti za to da i ovaj slučaj bude pokriven. Ona poziva funkciju *DodajInterno* za koren stabla i u slučaju da je došlo do podele korena, od ključa koji je dobila formira novi koren čiji je levi potomak dosadašnji koren, a desni potomak *NovoStablo*.

Crveno-crno stablo

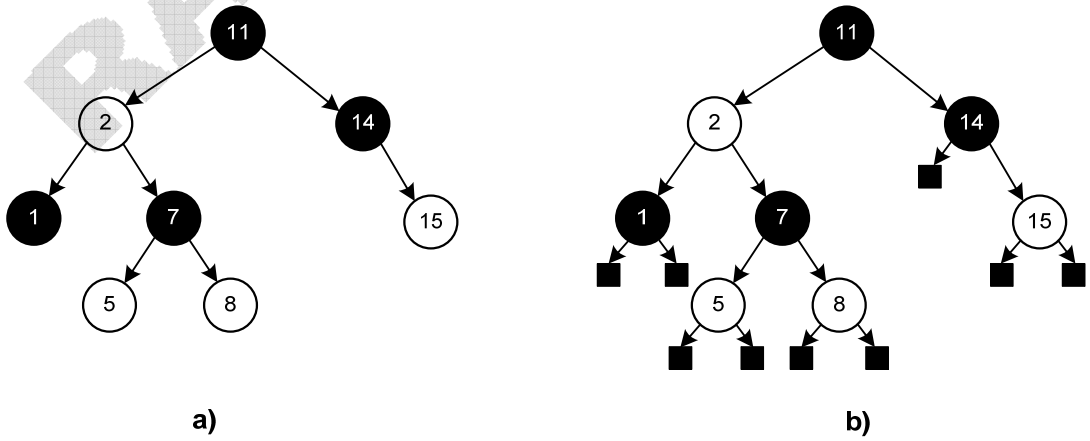
Crveno-crno stablo je binarno stablo za pretraživanje takvo da svaki njegov čvor ima pridruženu boju, koja može biti *crvena* ili *crna*. Pored osnovnih uslova koje mora da zadovoljava svako binarno stablo, crveno-crno stablo mora da zadovolji i sledeće uslove:

1. Svaki čvor je crven ili crn
2. Svi listovi su crni
3. Oba potomka svakog crvenog čvora su crna
4. Svaka putanja od čvora do njegovih listova sadrži isti broj crnih čvorova

Ovi uslovi obezbeđuju najvažniju osobinu crveno-crnih stabala: Najduža putanja od korena do lista nije više nego dva puta duža od najkraće putanje do listova u tom stablu. Rezultat toga je da je stablo grubo balansirano. S obzirom da operacije dodavanja, brisanja i pretraživanja u najgorem slučaju imaju vreme trajanja proporcionalno visini stabla, ovakva teorijska gornja granica visine omogućava crveno-crnim stablima da budu efikasna u najgorem slučaju, za razliku od običnih binarnih stabala za pretraživanje.

Da bismo razumeli zašto ovi uslovi to garantuju, dovoljno je da primetimo da, zahvaljujući 3. uslovu, nijedna putanja ne može imati dva crvena čvora jedan za drugim. Najkraća moguća putanja ima sve crne čvorove, a najduža moguća putanja ima i crvene i crne čvorove. S obzirom da 4. uslov garantuje da sve najduže putanje imaju isti broj crnih čvorova, to pokazuje da ni jedna putanja ne može biti više nego dva puta duža od bilo koje druge putanje.

U mnogim stablastim strukturama je moguće da čvor ima jednog ili ni jednog potomka, a da listovi sadrže podatak. I crveno-crno stablo je moguće predstaviti na ovaj način, ali se tako algoritam dodatno komplikuje. Iz tog razloga, često se koriste "nulti čvorovi" (eng. *null nodes*), koji ne sadrže nikakve podatke, već samo služe kao indikator kraja stabla, kao što je prikazano na Slici ###b. Ovi čvorovi se najčešće i ne prikazuju, tako da stablo prividno izgleda kao da ne zadovoljava navedene uslove, što ustvari nije tačno. Posledica ovakvog predstavljanja je da svi unutrašnji čvorovi (oni koji nisu listovi) imaju dva potomka, iako jedan ili oba potomka mogu biti nulti čvorovi. Kako bi se izbeglo bespotrebno zauzimanje memorije od strane nultih čvorova, može se kreirati samo jedan pomoćni čvor, koji će kasnije biti korišćen kao nulti čvor u čitavom stablu.



Slika ### Crveno-crno stablo (crveni čvorovi su označeni belim krugovima): a) originalno; b) originalno stablo sa dodatim nultim čvorovima