




FEBRUAR 2021.

DOKUMENTACIJA ZA PROJEKAT2

STRUKTURE PODATAKA I ALGORITMI

HAMZA KOLAŠINAC
PRIRODNO-MATEMATIČKI FAKULTET
Sarajevo



Sadržaj

Ideja projekta	2
Podržane operacije	2
Balansiranje.....	2
Implementacije.....	3
Find.....	3
FindCvor.....	3
Insert.....	4
PopraviStablo.....	5
Brisanje.....	6
PopraviStabloObratno	9
Razdvajanje	9
Spajanje	10
SklopiIh	11
Ispisi	11
Kraj	11

Ideja projekta

Ideja je da u stablo osim vrijednosti pamtimo i prioritet. Taj prioritet će nasumično biti izabran i brojevi sa najvećim prioritetima trebaju biti višoj u stablu. To ćemo postići određenim rotacijama u stablu kada nam nije zadovoljen heap za prioritete. Ovo sve radimo da bi balansiranost stabla održali što bliže \log_n -u. Po teoriji vjerovatnoće stablo će biti obično visine oko \log_n .

Podržane operacije

Pretraga – Vraća true ili false u zavisnosti da li je broj u stablu

Umetanje – Ubacuje broj u stablo i balansira ako je potrebno

Brisanje – Briše broj iz stabla i balansira ako je potrebno

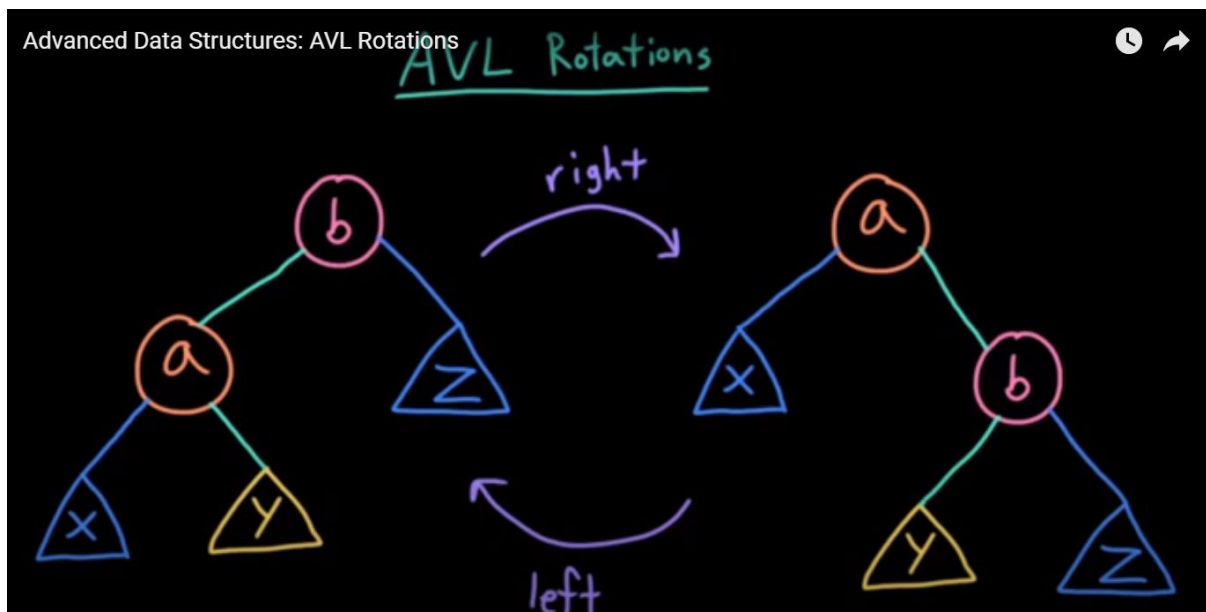
Razdvajanje – Prima broj i vraća dva stabla, jedan gdje su svi elementi manji od broja, a drugi gdje su svi elementi veći od broja

Spajanje – Prima dva stabla gdje su svi elementi jednog veći od svih elemenata drugog i vraća jedno stablo spojeno od ta dva stabla

Unija – Vraća uniju dva stabla

Balansiranje

Način na koji balansiramo je najvažnija stvar u ovom projektu jer se na njemu i na vremenu kada balansiramo zasniva projekat. Za balansiranje koristimo AVL rotacije ukoliko dijete nekog roditelja ima veći prioritet od njega. Ispod je slika AVL rotacija korištenih.



Implementacije

Find

Find je jedina funkcija koja ne zahtijeva balansiranje nikakvo. Jedino što radi je vraća true ili false u zavisnosti da li je neki broj u stablu.

```
template <typename Tip>
bool Stablo<Tip>::FindRek(Tip element, Cvor *cvor) {
    if (cvor->element == element) return true;
    if (element < cvor->element){
        if (cvor->ld != nullptr) return FindRek(element, cvor->ld);
        else return false;
    }
    else{
        if (cvor->dd != nullptr) return FindRek(element, cvor->dd);
        else return false;
    }
}
```

Implementacija je jednostavna, u Find pozivamo FindRek funkciju i proslijeđujemo joj element i početnu vrijednost odakle će početi tražiti. Svaki put kada uđe u funkciju FindRek provjeri da li smo na tom elementu i ako jesmo onda vrati true. Ako nismo na tom elementu provjeri da li je element manji ili veći od čvora kojeg smo poslali. Ako je manji pogleda samo ima li lijevo dijete od čvora i ako ima return-a šta god se vrati iz ponovnog zvanja funkcije FindRek gdje mu proslijedimo ponovo element i sada lijevo dijete čvora. Ako ne postoji lijevo dijete samo vratimo false. Ista je situacija za slučaj kada je element veći od čvora samo što sada umjesto za lijevo dijete da provjeravamo, provjeravamo za desno dijete. Ova funkcija radi u vremenu \log_n jer svaki put kad se pozove ponovo duplo manje nam ostane potencijalnih brojeva.

FindCvor

Slična funkcija kao Find samo umjesto da je bool povratni tip, ova funkcija vraća čvor.

```
template <typename Tip>
typename Stablo<Tip>::Cvor* Stablo<Tip>::FindCvor(Tip element) {
    Cvor *trenutni = korijen;
    while (trenutni != nullptr) {
        if (trenutni->element == element)
            return trenutni;
        if (trenutni->element > element)
            trenutni = trenutni->ld;
        else trenutni = trenutni->dd;
    }
    return nullptr;
}
```

Implementacija je sa vježbi i jedina razlika od Find funkcije je ako nađe vrati čvor umjesto true a ako ne nađe vrati nullptr.

Insert

Insert funkcija je prva funkcija gdje se koristi balansiranje po vrijednostima.

```
template <typename Tip>
void Stablo<Tip>::Insert(Tip element) {
    if(korijen == nullptr) {
        korijen = new Cvor(element);
        korijen->prioritet = rand();
        velicina = 1;
        return;
    }

    Cvor *trenutni = korijen;
    while(trenutni != nullptr) {
        if(trenutni->element == element)
            return;
        if(trenutni->element > element) {
            if(trenutni->ld != nullptr)
                trenutni = trenutni->ld;
            else {
                trenutni->ld = new Cvor(element, trenutni);
                trenutni = trenutni->ld;
                break;
            }
        }
        else {
            if(trenutni->dd != nullptr)
                trenutni = trenutni->dd;
            else {
                trenutni->dd = new Cvor(element, trenutni);
                velicina++;
                trenutni = trenutni->dd;
                break;
            }
        }
    }

    if(trenutni->rod->ld == trenutni) PopraviStablo(trenutni, true);
    else PopraviStablo(trenutni, false);
}
```

U iznad implementaciju vidimo da se prvo provjeri da li je prazno stablo jer ako jeste onda samo dodamo kao novi cvor, stavimo mu nasumičan prioritet i postavimo veličinu stabla na 1. Ako nije koristimo sličan kod kao u FindRek gdje tražimo da li postoji taj broj jer ako postoji onda možemo samo vratiti jer ne možemo insertati ga u stablo, ako ne postoji doći ćemo do mjesta gdje je nullptr lijevo ili desno dijete od čvora u zavisnosti da li je element manji ili veći od čvora i kada nađemo, ako je manji, stavimo lijevo dijete od čvora da novi čvor sa elementom

kao vrijednost. Ako je veći onda isto radimo sa desnim djetetom. U oba povećamo veličinu i break-amo. Sada kada smo insertali broj gledamo da li je lijevo dijete ili desno dijete i šaljemo funkciji PopraviStablo čvor na kojem je taj novi element i true ili false u zavisnosti da li je lijevo dijete ili desno. Prije nego što pređemo na PopraviStablo treba primjetiti da je Insert prije nego što balansiramo $O(\log_n)$ jer je sličan postupak kao u Find funkciji gdje kada idemo kroz stablo uzimamo samo podstabla i dijelimo na 2 stablo svaki put. Ukoliko je PopraviStablo $O(\log_n)$ ili brže onda će Insert ostati iste brzine i sa balansiranjem jer će biti $O(\log_n) + (\leq O(\log_n))$ što je jednako samo $O(\log_n)$. Pogledajmo sada balansiranje.

PopraviStablo

PopraviStablo funkcija je pomoćna funkcija koja namješta stablo po prioritetima dok su prioriteti u lošem poretku. Samo gleda poslati čvor i iznad njega sve i pretpostavlja da je sve u dobrom poretku ako su poslati čvor i njegov roditelj u dobrom poretku.

```
template <typename Tip>
void Stablo<Tip>::PopraviStablo(Cvor* pointer, bool lijevi){
    if(pointer != korijen){
        if(pointer->prioritet > pointer->rod->prioritet){
            if(lijevi){
                pointer->rod->ld = pointer->dd;
                if(pointer->dd != nullptr) pointer->dd->rod = pointer->rod;
                pointer->dd = pointer->rod;
                pointer->rod = pointer->dd->rod;
                pointer->dd->rod = pointer;
                if(pointer->rod != nullptr){
                    if(pointer->rod->element > pointer->element) pointer->rod->ld = pointer;
                    else pointer->rod->dd = pointer;
                }
                else korijen = pointer;
            }
            else{
                pointer->rod->dd = pointer->ld;
                if(pointer->ld != nullptr) pointer->ld->rod = pointer->rod;
                pointer->ld = pointer->rod;
                pointer->rod = pointer->ld->rod;
                pointer->ld->rod = pointer;
                if(pointer->rod != nullptr){
                    if(pointer->rod->element > pointer->element) pointer->rod->ld = pointer;
                    else pointer->rod->dd = pointer;
                }
                else korijen = pointer;
            }
        }
        else return;
    }
    else return;

    if(pointer != korijen){
        if(pointer->rod->ld == pointer) PopraviStablo(pointer, true);
        else PopraviStablo(pointer, false);
    }
}
```

PopraviStablo funkcija radi na principu da prvo provjeri da slučajno nije korijen poslat jer ako jeste onda samo završimo jer je dobar balans. Ako nije korijen onda provjeravamo da li je loš prioritet između poslatog čvora i njegovog roditelja. Ako nije onda samo završimo jer je onda

dobar balans, a ako jeste onda gledamo da li je čvor lijevo ili desno dijete i radimo odgovarajuću AVL rotaciju. Ukoliko je lijevo dijete onda postavljamo lijevo dijete od čvorovog roditelja na desno dijete od čvora i ako nije nullptr stavljamo tom čvoru roditelja kao roditelja od čvora. Dalje stavljamo od čvora desno dijete da je sada njegov roditelj i stavljamo da mu je sad novi roditelj stari roditelj od njegovog starog roditelja. Sada njegov stari roditelj ima novog roditelja što je sada čvor i ako novi roditelj od čvora postoji onda je njemu dijete naš čvor a ako ne postoji onda je naš čvor postao korijen. To je to za scenario kada je čvor lijevo dijete, kada je desno dijete je slična priča samo što se mijenjaju lijevo dijete umjesto desnog djeteta od čvorovog roditelja i sada će čvorov roditelj biti lijevo dijete čvoru a ne desno. Na kraju, ako pointer nije postao korijen, zove se ponovo PopraviStablo na isti način kao ranije. PopraviStablo radi u vremenu $O(\log_n)$ kao i Insert jer maksimalno može provjeriti h brojeva gdje je h visina stabla. Najgori slučaj je ako je u dnu a ima prioritet najveći u stablu gdje opet samo h brojeva prođe. Pošto i PopraviStablo radi u $O(\log_n)$ vremenu to znači da je i Insert funkcija u istom vremenu.

Brisanje

Brisanje, kao Insert, ima razloga za balansiranje. Ukoliko se pogrešan čvor obriše mora se zadovoljiti heap za prioritete ponovo. Brisanja je izvršeno u više funkcija, Brisanje, BrisanjeKorijena i BrisanjeRek za samo brisanje kao i PopraviStabloObratno za balansiranje.

```
template <typename Tip>
void Stablo<Tip>::Brisanje(Tip element){
    if(!Find(element)) return;
    Cvor* za_brisanje = FindCvor(element);
    if(za_brisanje->element != korijen->element){
        BrisanjeRek(za_brisanje);
    }
    else{
        BrisanjeKorijena(za_brisanje);
    }
}
```

Što se tiče Brisanje funkcije

ona je jednostavna. Ona provjeri da li postoji element, namjesti čvor za_brisanje preko FindCvor funkcije na čvor sa vrijednošću element i provjerava da li je korijen taj čvor. Ako jeste šalje za_brisanje u funkciju BrisanjeKorijena, a ako nije onda ga šalje u BrisanjeRek. Za brisanje korijena je slično kao i za brisanje ostalih čvorova samo što se neke veze drugačije namještaju. Veze kao roditeljske su drugačije kod korijena jer on nema roditelja, ali mora namještati korijen da je sada taj novi element koji dođe.

```
template <typename Tip>
void Stablo<Tip>::BrisanjeKorijena(Cvor* za_brisanje){
    if(za_brisanje->ld == nullptr && za_brisanje->dd == nullptr){
        korijen = nullptr;
        delete za_brisanje;
        velicina--;
        za_brisanje = nullptr;
    }
}
```

Prva provjera je ako je korijen bez djece. Ukoliko se to desi korijen stavimo samo na nullptr, obrišemo čvor za_brisanje, veličinu smanjimo i stavimo za_brisanje na nullptr.

```

else if(za_brisanje->ld != nullptr && za_brisanje->dd == nullptr){
    korijen = za_brisanje->ld;
    za_brisanje->ld->rod = nullptr;
    delete za_brisanje;
    za_brisanje = nullptr;
    velicina--;
}
else if(za_brisanje->ld == nullptr && za_brisanje->dd != nullptr){
    korijen = za_brisanje->dd;
    za_brisanje->dd->rod = nullptr;
    delete za_brisanje;
    za_brisanje = nullptr;
    velicina--;
}

```

Druga i treća provjera su ukoliko čvor ima jedno dijete. Druga je za lijevo dijete a treća za desno. U ovom slučaju je također jednostavno, samo korijen stavimo na odgovarajuće dijete, maknemo tom djetetu vezu sa roditeljom jer korijen nema roditelja, obrišemo za_brisanje, stavimo ga na nullptr i smanjimo veličinu. Ove tri promjene nisu problematične što se tiče prioriteta jer su ili obrisani svi elementi (prvi slučaj) ili je čvor postao element bez roditelja a ispod njega se niko mijenjao nije(drugi i treći slučaj). Sada u četvrtom i zadnjem slučaju imamo problema sa balansiranjem.

```

else{
    Cvor* zamjena = NadjiNajmanji(za_brisanje->dd);
    za_brisanje->element = zamjena->element;
    za_brisanje->prioritet = zamjena->prioritet;
    BrisanjeRek(zamjena);
    zamjena = nullptr;
    PopraviStabloObratno(za_brisanje);
    za_brisanje = nullptr;
}
}

```

Sada kada znamo da ima dvoje djece namještamo na sljedeći način. Tražimo dijete koje je najviše lijevo od desnog djeteta čvora. To dijete ako se zamijeni sa našim korijenom neće ugroziti integritet stabla jer znamo da je veće od svih elemenata lijevo od korijena jer smo desno išli da ga nađemo, a znamo da je manji od svih desno elemenata jer je najmanji element u desnom podstablu. Da ga nađemo koristimo funkciju NadjiNajmanji kojoj šaljemo desno dijete od čvora našeg za_brisanje.

```

template <typename Tip>
typename Stablo<Tip>::Cvor* Stablo<Tip>::NadjiNajmanji(Cvor* pointer){
    while(true){
        if(pointer->ld == nullptr) return pointer;
        pointer = pointer->ld;
    }
}

```

Implementacija je jednostavna gdje samo dok pointer->ld nije nullptr stavljamo pointer na pointer->ld, kada bude nullptr samo vratimo pointer.

Dalje za brisanje, kada nađemo najmanji element u desnom podstablu stavimo ga u varijablu zamjena. Zamijenimo od našeg za_brisanje i element i prioritet sa elementom i prioritetom od zamjene i pošaljemo zamjenu da se obriše. Na ovaj način se briše zamjena a nju smo samo zamijenili umjesto za_brisanje. Zamjenu sada stavimo na nullptr, pozovemo PopraviStabloObratno i stavimo onda tek za_brisanje na nullptr. To je to za brisanje, nakon što se izbalansira stablo završena je funkcija. Kao i za Insert, pošto su odvojene funkcije, ako je PopraviStabloObratno iste brzine ili brže od Brisanja, onda je Brisanje funkcija brzine $O(\log_n)$. Ovo je zato što da nađemo čvor potrošimo \log_n vremena, a ostatak brisanja ide brže. Prije nego što pogledamo PopraviStabloObratno kako je implementiran, pogledajmo brzo razlike kod BrisanjeRek.

```
template <typename Tip>
void Stablo<Tip>::BrisanjeRek(Cvor* za_brisanje){
    if(za_brisanje->dd == nullptr && za_brisanje->ld == nullptr){
        if(za_brisanje->rod->ld == za_brisanje) za_brisanje->rod->ld = nullptr;
        else za_brisanje->rod->dd = nullptr;
        delete za_brisanje;
        za_brisanje = nullptr;
        velicina--;
    }
}
```

Isti je pristup, samo što različite stvari namještamo, ovdje je razlika ako nema djece da se mora promijeniti za roditelja njegovog dijete.

```
    else if(za_brisanje->dd != nullptr && za_brisanje->ld == nullptr){
        if(za_brisanje->rod->ld == za_brisanje) za_brisanje->rod->ld = za_brisanje->dd;
        else za_brisanje->rod->dd = za_brisanje->dd;
        za_brisanje->dd->rod = za_brisanje->rod;
        delete za_brisanje;
        za_brisanje = nullptr;
        velicina--;
    }
    else if(za_brisanje->dd == nullptr && za_brisanje->ld != nullptr){
        if(za_brisanje->rod->ld == za_brisanje) za_brisanje->rod->ld = za_brisanje->ld;
        else za_brisanje->rod->dd = za_brisanje->ld;
        za_brisanje->ld->rod = za_brisanje->rod;
        delete za_brisanje;
        za_brisanje = nullptr;
        velicina--;
    }
}
```

Za drugi i treći slučaj je ista priča, jedina razlika je da roditelju dijete promijenimo na dijete od čvora našeg.

```
    else{
        Cvor* zamjena = NadjiNajmanji(za_brisanje->dd);
        za_brisanje->element = zamjena->element;
        za_brisanje->prioritet = zamjena->prioritet;
        BrisanjeRek(zamjena);
        zamjena = nullptr;
        PopraviStabloObratno(za_brisanje);
        za_brisanje = nullptr;
    }
}
```

Za čvor sa dvoje djece ostaje isto sve jer ne mijenjamo tu veze nikako. Sada nastavimo.

PopraviStabloObratno

```
template <typename Tip>
void Stablo<Tip>::PopraviStabloObratno(Cvor* za_brisanje){

    while(true){
        if(za_brisanje->ld != nullptr){
            if(za_brisanje->dd != nullptr){
                if((za_brisanje->ld->prioritet >= za_brisanje->dd->prioritet)
                    && (za_brisanje->ld->prioritet > za_brisanje->prioritet)) PopraviStablo(za_brisanje->ld, true);
                else if (za_brisanje->dd->prioritet > za_brisanje->prioritet) PopraviStablo(za_brisanje->dd, false);
                else break;
            }
            else if (za_brisanje->ld->prioritet > za_brisanje->prioritet) PopraviStablo(za_brisanje->ld, true);
            else break;
        }
        else if ((za_brisanje->dd != nullptr) && (za_brisanje->dd->prioritet > za_brisanje->prioritet))
            PopraviStablo(za_brisanje->dd, false);
        else break;
    }
}
```

PopraviStabloObratno koristi PopraviStablo funkciju i samo šalje čvorove koje želi popraviti. Pošto se ova funkcija samo poziva kad smo imali dvoje djece pa znamo da smo mijenjali vrijednost nekog čvora. U while(true) petlji provjeravamo ima li lijevo i desno dijete. Ako ima i lijevo i desno dijete onda gledamo da li lijevo ima veći prioritet od desnog i da li je taj prioritet veći od čvorovog. Ako je to slučaj šaljemo u PopraviStablo lijevo dijete čvora, ako nije onda gledamo da li je prioritet desnog djeteta veći od čvorovog i ako jeste šaljemo desno dijete čvora. Ako nije ni jedno ni drugo od prošla dva uslova onda izlazi iz while(true) petlje. Sada ako je došlo do sljedećeg else if-a znamo da nema lijevo dijete i provjerimo samo ako ima desno dijete da li mu je prioritet veći od čvorovog. Ako jeste onda šaljemo desno dijete čvora, ako nije onda izlazimo iz petlje. Ovako se radi downheap po prioritetima i riješimo se problema oko balansiranja. Brzina ove funkcije je ista kao u PopraviStablo jer su provjere $O(1)$ a PopraviStablo je $O(\log_n)$ pa je sve $O(\log_n)$. Dakle i Brisanje je $O(\log_n)$ jer je sada $O(\log_n) + O(\log_n) = O(\log_n)$ asimptotski.

Razdvajanje

Razdvajanje funkcija je funkcija koja koristi dosta funkcija od koje smo već implementirali. Trebamo razdvojiti stablo u dva podstabla po vrijednosti k koja nije u stablu. Potrebno nam je da su u jednom stablu svi brojevi manji od k , a u drugom svi brojevi veći od k . Ideja je da ubacimo k u stablo, stavimo mu prioritet najveći u stablu da bi bio korijen i onda vratimo samo njegovo lijevo podstablo gdje su svi brojevi manji od njega i njegovo desno podstablo gdje su svi brojevi veći od njega. Dalje je implementacija.

```
template <typename Tip>
std::pair<Stablo<Tip>, Stablo<Tip>> Razdvajanje(Stablo<Tip> s, Tip k){
    Stablo<Tip> s1;
    Stablo<Tip> s2;
    s.VratiDvaStabla(s1, s2, k);
    std::pair<Stablo<Tip>, Stablo<Tip>> p(s1, s2);
    return p;
}
```

Napravimo dva stabla i zovnemo funkciju VратиDvaStabla i prosljedimo joj s1, s2 i k gdje su s1 i s2 po referenci. Kada se vratimo iz funkcije VратиDvaStabla napravimo novi par p i u njega ubacimo s1 i s2 i return-amo ga.

```
template <typename Tip>
void Stablo<Tip>::VратиDvaStabla(Stablo<Tip> &s1, Stablo<Tip> &s2, Tip k){
    Insert2(k, korijen->prioritet+1000000000000);
    s1.Insert(0);
    s2.Insert(0);
    Cvor* k1 = korijen->ld;
    Cvor* k2 = korijen->dd;
    s1.Promijeni(k1);
    s2.Promijeni(k2);
}
template <typename Tip>
void Stablo<Tip>::Promijeni(Cvor* pointer){
    korijen = pointer;
}
```

VратиDvaStabla radi na principu da primi dva stabla, s1 i s2 po referenci i primi k. Iskoristi Insert2 i prosljedi joj k i najveći prioritet u stablu što je korijen->prioritet + ogroman broj. Insert2 je kopija Inserta samo ima par malih razlika što su korisne za Razdvajanje. Razlika je što prima dodatnu vrijednost prio koja je prioritet na koji hoćemo taj novi element da postavimo i sljedeća linija koda:

```
FindCvor(element)->prioritet = prio;
```

Ovdje samo mijenjamo prioritet novog čvora na dati prio. Ostalo je sve isto kao i Insert funkcija. Nakon korištenja Insert2 funkcije u s1 i s2 Insert-amo neki broj, napravimo nove čvorove na vrijednosti lijevog i desnog djeteta od korijena i stavimo korijen od s1 i s2 na k1 i k2 respektivno. Za funkciju Razdvajanje je to to. Brzina izvršavanja je $O(\log_n)$ jer Insert je te brzine kao i Insert2 a ostale funkcije i ostali dio koda je $O(1)$ što je na kraju sve $O(\log_n)$.

Spajanje

Za spajanje imamo sličnu ideju kao za razdvajanje samo što ovdje dobijemo dva stabla i pravimo jedno veliko stablo. Ideja je da uzmemo novo stablo, ubacimo element sa malim prioritetom koji je veći od svih brojeva iz s1 a manji od svih brojeva iz s2 i onda balansirati dok ne dođe do dna i obrisati ga samo. Dalje je implementacija.

```
template <typename Tip>
Stablo<Tip> Spajanje(Stablo<Tip> s1, Stablo<Tip> s2){
    Stablo<Tip> s;
    Tip a = s.SklopiIh(s1, s2);
    s.PopraviStabloObratno(s.FindCvor(a));
    s.Brisanje(a);
    return s;
}
```

Implementacija je jednostavna, napravimo stablo, napravimo Tip a i stavimo ga da je jednak `s.SklopiIh(s1, s2)`; što vraća Tip i onda pozovemo `PopraviStabloObratno` od `FindCvor(a)` što je samo čvor od broja kojeg smo ubacili, obrišemo taj broj i vratimo s. Sada, kao i funkcije prije ove, brzina izvršavanja zavisi od funkcije `SklopiIh`, zbog `FindCvor` i `PopraviStabloObratno` i `Brisanje` brzina je bar $O(\log_n)$ ali ako je `SklopiIh` sporije onda će i ova funkcija biti sporija. Dalje je `SklopiIh` funkcija.

SklopiIh

```
template <typename Tip>
Tip Stablo<Tip>::SklopiIh(Stablo<Tip> &s1, Stablo<Tip> &s2) {
    Tip broj = s1.NadjiNajveci(s1.VratiKorijen())->element;
    Insert2(broj, -1);
    korijen->ld = s1.VratiKorijen();
    s1.VratiKorijen()->rod = korijen;
    korijen->dd = s2.VratiKorijen();
    s2.VratiKorijen()->rod = korijen;
    return broj;
}
```

`SklopiIh` uzima opet stabla po referenci i u Tip broj stavlja najveći broj u `s1`. Taj broj se ubacuje u stablo s preko funkcije `Insert2`. Dajemo mu prioritet -1 da osiguramo da je zadnji kad budemo balansovali. Stavljamu mu lijevo dijete `korijen` od `s1` a desno `korijen` od `s2` i upotpunjujemo vezu s druge strane. Na kraju samo vratimo broj. Ova funkcija radi u vremenu $O(\log_n)$ jer smo koristili `Insert2` i `NadjiNajveci` koji su tog vremena. Ovo znači da čitava funkcija `Spajanje` radi u vremenu $O(\log_n)$.

Ispisi

```
template <typename Tip>
void Stablo<Tip>::IspisiRek(Cvor* pointer) {
    if(pointer != nullptr) {
        if(pointer->ld != nullptr) IspisiRek(pointer->ld);
        if(pointer->dd != nullptr) IspisiRek(pointer->dd);
        std::cout<<"Broj je: "<<pointer->element<<"."<<std::endl;
        std::cout<<"    Prioritet mu je : "<<pointer->prioritet<<"."<<std::endl;
        if(pointer->rod != nullptr) std::cout<<"    Roditelji mu je: "<<pointer->rod->element<<"."<<std::endl;
        if(pointer->ld != nullptr) std::cout<<"    Lijevo dijete mu je: "<<pointer->ld->element<<"."<<std::endl;
        if(pointer->dd != nullptr) std::cout<<"    Desno dijete mu je: "<<pointer->dd->element<<"."<<std::endl;
    }
}
```

`Ispisi` je funkcija koja ispisuje za svaki element u stablu njega, prioritet mu, roditelja, lijevo dijete i desno dijete. Funkcija `Ispisi` poziva funkciju `IspisiRek` koja će proći kroz sveki element i ispisati koliko može.

Kraj

Za kraj je važno reći da preko ovog načina sa nasumičnim prioritetima će se u većini slučajeva dobiti stablo veličine \log_n za koje je sve ovo moguće da bude ovako brzo.