UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milan Z. Kocić

STRUKTURE PODATAKA ZA EFIKASNO REŠAVANJE PROBLEMA PRETHODNIKA ELEMENTA

master rad

Beograd, 2023.

Mentor:

dr Vesna Marinković, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Mirko Spasić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Najmilijima

Naslov master rada: Strukture podataka za efikasno rešavanje problema prethodnika elementa

Rezime: U svetu savremenog računarstva, strukture podataka igraju ključnu ulogu u efikasnom upravljanju i organizovanju informacija. U radu će biti istraživane strukture podataka kojima se može efikasno rešiti problem prethodnika. Problem prethodnika podrazumeva da je za dati skup S, i broj x, potrebno naći najveći broj skupa S koji je manji ili jedak broju x. Predstavljena su tri pristupa rešavanju problema prethodnika korišćenjem vEB stabla, x-brzog prefiksnog stabla i y-brzog prefiksnog stabla. vEB stablo rekurzivno deli skup na manje celine, i na taj način omogućava da problem smanjimo na problem mnogo manje dimenzije. X-brzo prefiksno stablo kombinuje prefiksno stablo sa binarnom pretragom, čime postiže da se preskoče neki čvorovi u stablu, i brzo nađe prethodnik broja. Pristup zasnovan na y-brzom prefiksnom stablu unapređuje postupak rešavanja problema prethodnika broja zasnovanog na x-brzom prefiksnom stablu koristeći tehniku indirekcije. U radu su opisane karakteristike, implementacija, prednosti i nedostaci ovih struktura podataka. Izvršeno je eksperimentalno upoređivanje struktura na problemu prethodnika i predstavljeni su rezultati eksperimenata. U zaključku je istaknuto zašto je svaka od ovih struktura značajna i gde se koristi.

Ključne reči: vEB stablo, prefiksno stablo, struktura podataka, problem prethodnika, x-brzo prefiksno stablo, y-brzo prefiksno stablo

Sadržaj

Glava 1

Uvod

1.1 Problem prethodnika

Problem pronalaženja prethodnika elementa (eng. predecessor problem) u skupu možemo smatrati uopštenjem problema ispitivanja pripadnosti nekog elementa skupu. Problem ispitivanja pripadnosti nekog elementa skupu podrazumeva da se za dati skup S i element x utvrdi da li važi $x \in S$. Problem prethodnika neformalno podrazumeva nalaženje elementa y tako da važi da je y najveći element skupa S koji je manji ili jednak broju x. Pojasnimo zašto problem prethodnika možemo posmatrati kao uopštenje problema ispitivanja pripadnosti. Naime, ako x pripada skupu S on je rezultat izračunavanja prethodnika broja x, dok u slučaju da x ne pripada skupu S ne dobijamo samo odgovor ne, već kao rezultat dobijemo najveći element skupa S manji od x. Nekada se problem prethodnika definiše i tako da ne obuhvata sam element x, odnosno da se traži element strogo manji od x. Formalnije, problem prethodnika dobija skup S i broj x kao ulaz i potrebno je odrediti maksimalni element skupa $S' = \{y | y \in S \ \wedge \ y \leq x\}$. U slučaju kada je broj x manji od svih elemenata skupa S tada prethodnik broja x ne postoji u skupu S.

Iako na prvi pogled problem prethodnika ne deluje previše značajno pokazuje se da on ima mnogobrojne primene. Neke od bitnijih primena su sortiranje celih brojeva [2], sortiranje niski [3], pretraga niski [4] i rutiranje paketa na mreži [6].

Može se razmatrati dinamička i statička verzija problema: dinamička podrazumeva da se skup S može menjati, odnosno dozvoljava se upis novih elemenata kao i brisanje već postojećih, dok statička verzija ne dozvoljava nikakve izmene početnog skupa S. Za statički problem prethodnika je dovoljno na pametan način pripremiti elemente skupa S i onda odgovarati na upite. Obe varijante problema su značajne.

U ovom radu će akcenat biti stavljen na statičku verziju problema, ali ćemo se u par navrata dotaći i dinamičke verzije.

Problem pronalaženja prethodnika elementa je bio tema velikog broja istraživanja i razvijen je veliki broj različitih rešenja. U zavisnosti od modela izračunavanja koji se koristi imamo rešenja različite složenosti. Primetimo da ako znamo da rešimo problem prethodnika, možemo jednostavno sortirati elemente datog skupa: naime, možemo u linearnoj vremenskoj složenosti pronaći maksimalni element skupa, i onda iznova tražiti prethodnika maksimuma, zatim prethodnika tog elementa, itd. U ovom postupku podrazumevamo da operacija prethodnika ne vraća isti element. Tako bismo dobili sortirani poredak elemenata skupa. Samim tim očekivano je da u modelu izračunljivosti koji se zasniva na upoređivanju elemenata ne možemo dobiti rešenje koje ima složenost pripreme za problem prethodnika manju od $O(n \log n)$ i složenost upita pronalaženja prethodnika datog elementa manju od $O(\log n)$, jer bi to značilo da možemo dobiti algoritam sortiranja n elemenata efikasniji od $O(n \log n)$, što po teorijskoj donjoj granici problema sortiranja znamo da nije moguće. Videćemo, međutim, da možemo dobiti neka rešenja koja su bolje složenosti, ali koriste neke drugačije modele izračunavanja.

1.2 Neka jednostavnija rešenja

U ovom poglavlju biće predstavljena neka od najjednostavnijih rešenja problema prethodnika koja su donekle prihvatljiva po pitanju složenosti.

Jedno od najjednostavnijih rešenja bi bilo da skup S čuvamo kao sortiran niz, a da onda prethodnik elementa tražimo binarnom pretragom. Ovo rešenje je prihvatljivo ako razmatramo statičku verziju problema, i ima složenost upita prethodnika $O(\log n)$ gde je n broj elemenata skupa S, dok je za dinamičku verziju problema složenost izmena na skupu složenosti $O(n)$, što nije dovoljno efikasno.

Malo naprednije rešenje ovog problema bi predstavljala samobalansirajuća uređena binarna stabla. Složenost operacije računanja prethodnika u ovom pristupu bi za statički problem prethodnika bila identična, međutim, u slučaju dinamičke varijante problema, operacije upisivanja i brisanja elemenata iz skupa S bi bile dosta efikasnije. Naime, operacije kojima se menja skup S su u ovom slučaju složenosti $O(\log n)$, a to je znatno efikasnije nego što je to slučaj sa nizom.

Ukoliko su elementi skupa S celi brojevi ili je moguće nekako napraviti preslikavanje f elemenata skupa S u skup celih brojeva $\{0, 1, \ldots, u-1\}$, gde $u-1$ pred-

stavlja najveći broj koji bi mogao da se nađe u skupu S, efikasnost upita možemo dodatno ubrzati na relativno jednostavan način. Naime, možemo unapred izračunati prethodnika svakog elementa i odgovarajuću informaciju smestiti u niz. Time bi se prethodnik elementa x pamtio u tom nizu na indeksu $f(x)$. Primetimo da ukoliko su elementi skupa S celi brojevi za f se može uzeti $f(x) = x$. Računanje prethodnika nekog elementa bi se na ovaj način svelo na čitanje vrednosti na nekom indeksu u nizu, što je konstantne vremenske složenosti, ali je rešenje memorijski zahtevno, jer zahteva da za svaki mogući element skupa rezervišemo memorijski prostor. Takođe, ovo rešenje nije prihvatljivo za dinamičku verziju problema, jer je složenosti operacija kojima se menja skup u najgorem slučaju $O(u)$.

Glava 2

vEB stablo

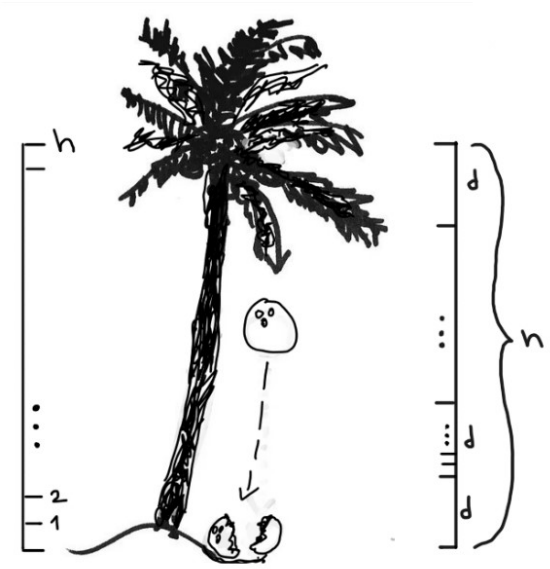
2.1 Opis strukture podataka

1975. godine Peter van Embde Boas je predložio strukturu podataka koja efikasno rešava problem prethodnika [10]. Struktura je po njemu nazvana *van Embde Boas stablo*, ili skraćeno vEB stablo. Uobičajne strukture podataka kao što su balansirana uređena binarna stabla rešavaju ovaj problem u logaritamskom vremenu, i nisu dovoljno efikasne za velike skupove podataka. Ova struktura je predstavljala značajno poboljšanje u odnosu na prethodno poznato najbolje rešenje.

vEB stablo je jedna od prvih struktura koja koristi ideju ograničenog domena da bi se ubrzalo rešavanje problema. Naime, ako koristimo model koji je zasnovan na poređenju elemenata, donja granica za problem prethodnika bi bila $O(\log n)$. Međutim, ako se algoritam ne oslanja na poređenje elemenata skupa, moguće je dobiti efikasnije strukture i algoritme. Naravno, ako vršimo poređenja elemenata onda moraju postojati neka druga ograničenja. Kod vEB stabla pretpostavljamo da svi elementi moraju biti iz ograničenog domena, što je donekle i realno očekivati ako radimo nad podacima u računaru (na primer, celobrojni tip podataka je najčešće ograničen sa 2^{32} ili 2^{64}).

Da bismo bolje razumeli ideju koju koristi vEB stablo evo jednog zanimljivog zadatka. Imamo dva kokosova oraha i jedno drvo, i potrebno je naći najveću visinu (izraženu u centimetrima) sa koje možemo baciti kokos sa drveta, a da on ne pukne. Naravno ovde na problem gledamo donekle idealizovano, odnosno kokosu dajemo osobinu da svaki put kada se baci sa neke visine, a ne pukne, nije ni na koji način oslabljen i ako bi se opet bacio sa te ili manje visine ponovo ne bi pukao. Ukoliko bismo imali jedan kokos, ne bismo imali pametniji način od toga da prvo bacimo

kokos sa 1cm, nakon toga sa 2cm i tako dalje, sve do onog trenutka kada bismo ga bacili sa x cm, pri čemu bi on pukao. U tom trenutku bismo mogli sa sigurnošću da zaključimo da je odgovor jednak $x - 1$. S obzirom da mi na raspolaganju imamo dva kokosa, nadamo se da rešenje možemo dobiti na neki efikasniji način. Ako pretpostavimo da je drvo visine h, mi tu visinu možemo podeliti na n jednakih delova, tako da svaki od tih delova bude dužine $d = h/n$ cm (slika 2.1). U ovom slučaju bismo prvi kokos bacali sa visine d cm, pa onda sa $2 \cdot d$ cm i tako dalje dok on ne bi pukao. Ukoliko je kokos pukao pri bacanju sa visine $x \cdot d$ znamo da je visina koju tražimo u intervalu $((x-1) \cdot d, x \cdot d]$, i drugi kokos bismo mogli da koristimo kao u prvoj taktici da odredimo tačnu visinu. Lako se pokazuje da je za najbrže računanje tražene visine potrebno da broj intervala i broj elemenata budu isti, tj da važi $n = d = \sqrt{h}$.

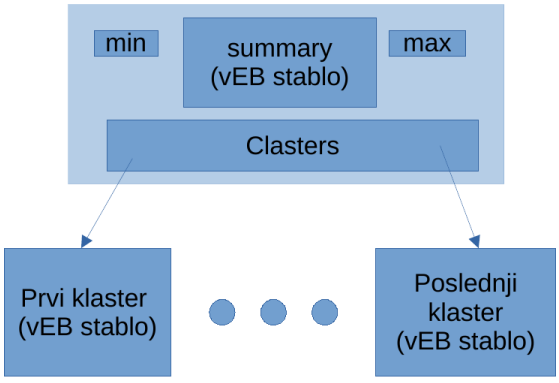


Slika 2.1: Ilustracija zadataka sa kokosom

U sličnom duhu je formulisano i vEB stablo. vEB stablo pretpostavlja da su svi elementi iz nekog velikog, konačnog skupa U koji nazivamo *univerzum*, koji je obično oblika $\{0, 1, ..., u-1\}$ gde je u broj elemenata skupa U. Univerzum se deli na $\sqrt{u}$ delova sa isto toliko elemenata i svaki od tih delova zovemo *klaster*. Dakle, imamo $\sqrt{u}$ klastera, sa po $\sqrt{u}$ elemenata. Međutim, ako bismo samo podelili univerzum i čuvali klastere u nizu, to nam ne bi bilo od velike pomoći, jer bismo onda polazni problem smanjili na probleme koje ne možemo rešiti rekurzivno na isti način, već bismo ih morali ručno rešavati. Zbog toga svaki od klastera posmatramo kao novo

vEB stablo koje je manje dimenzije, odnosno na svaki podsegment primenjujemo identičnu taktiku kao u slučaju zagonetke sa kokosom.

Svaki element ima svoje jedinstveno mesto u strukturi vEB stabla. Na primer, broj x koji se nalazi u strukturi podataka biće smešten u $(x/\sqrt{u})$-tom klasteru, gde je sa / označeno celobrojno deljenje, i u tom klasteru će biti $(x \mod \sqrt{u})$-ti po redu element. Ako je vEB stablo odgovarajuće dimenzije, operacije celobrojnog deljenja i izračunavanja ostatka pri deljenju se mogu zameniti bitovskim operatorima. Naime, ako je $\sqrt{u}$ neki stepen dvojke, onda se deljenje broja x brojem $\sqrt{u}$ može svesti na izdvajanje viših bitova broja x, a operacija računanja ostatka pri deljenju na izdvajanje nižih bitova broja x. Dakle, svaki od klastera čuva brojeve $[0, \sqrt{u} - 1]$, a onda se u zavisnosti od klastera u kom se nalazi broj određuje koji broj to predstavlja u vEB stablu. Ako bismo u strukturi podataka čuvali samo klastere, ne bismo imali podršku za izvršavanje nekih složenijih operacija, poput operacije računanja prethodnika broja. Zato u samoj strukturi podataka pored klastera imamo i informacije o minimalnom i maksimalnom elementu u stablu i, dodatno, imamo još jedno vEB stablo koje se obično naziva *sažetak* (eng. summary), i koje čuva informacije o tome koji klasteri imaju neke elemente. Preciznije, u sažetku se nalaze redni brojevi klastera u koje smo ubacivali neke elemente.

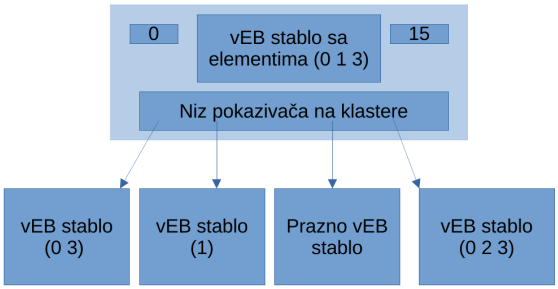


Slika 2.2: Struktura vEB stabla

Na slici 2.2 prikazana je struktura vEB stabla: minimum, maksimum, sažetak, kao i svaki od $\sqrt{u}$ klastera koji predstavljaju vEB stablo dimenzije $\sqrt{u}$.

Na slici 2.3 ilustrovano je vEB stablo čiji je univerzum dimenzije 16 i koji sadrži elemente $0, 3, 5, 12, 14, 15$. Vidimo da prvi klaster sadrži elemente 0 i 3 i to je opet vEB stablo, što znači da ima svoj sažetak, minimalni i maksimalni element. Sažetak stabla sadrži elemente $0, 1$ i 3 što označava da ti klasteri nisu prazni. Naime, drugi

klaster je prazan, pa se on ne nalazi u sažetku vEB stabla. Primetimo da se, na primer broj 5 ne čuva kao broj 5, već se čuva kao 1 u klasteru na indeksu 1. Broj 5 dobijamo tako što pomnožimo indeks klastera sa korenom veličine stabla, u ovom slučaju $\sqrt{16} = 4$ i dodamo vrednost koja se nalazi u klasteru, odnosno $1 \cdot 4 + 1 = 5$.



Slika 2.3: Primer vEB stabla dimenzije 16

Razmotrimo na koji način bismo izvršili proveru da li neki element pripada vEB stablu. Kako za svaki element znamo u kom klasteru se nalazi, smanjujemo problem sa dimenzije u na dimenziju $\sqrt{u}$ u konstantnom vremenu. Dakle, ispitivanje da li se neki element nalazi u vEB stablu svodi se na proveru odgovarajućeg klastera, a kako je klaster novo vEB stablo, složenost ove operacije se može izraziti rekurentnom jednačinom $T(u) = T(\sqrt{u}) + O(1)$ čije je rešenje $O(\log \log u)$. Slično važi i za operacije upisivanja elementa i brisanje elementa iz strukture. Za upisivanje elementa u vEB stablo potrebno je upisati element u odgovarajući klaster, i ukoliko je to prvi element u klasteru upisati u sažetak vEB stabla da određeni klaster više nije prazan. U slučaju kada po prvi put upisujemo u neki klaster neki element, imamo dva rekurzivna poziva, jedan za upisivanje elementa u klaster, drugi za upisivanje rednog broja klastera u sažetku vEB stabla. Oba rekurzivna poziva su dimenzije $\sqrt{u}$ pa je rešenje te rekurentne jednačine $O(\log u)$, a želimo postići $O(\log \log u)$. Zato je potrebno donekle modifikovati upis elemenata u klaster. To radimo tako što se minimalni i maksimalni element stabla ne smeštaju u klaster, već samo kao podatak da je to minimalni element stabla. Znači da u slučaju da upisujemo broj x u neki od klastera stabla koji je prazan, upisaćemo da je $x \mod \sqrt{u}$ minimum i maksimum tog klastera, i informaciju da $x/\sqrt{u}$-ti klaster vise nije prazan, ali broj $x \mod \sqrt{u}$ se neće naći u samom klasteru. Na taj način dobijamo da je jedan rekurzivni poziv konstantne vremenske složenosti, dok je drugi dimenzije $\sqrt{u}$, čime dobijamo istu rekurentnu jednačinu kao za proveru da li je element u stablu, čije je rešenje $O(\log \log u)$.

Operacije prethodnika i sledbenika elementa biće objašnjene malo kasnije.

2.2 Implementacija

Sada, nakon opisa interne strukture vEB stabla, možemo razmotriti i njegovu implementaciju. Videli smo da nam je u strukturi potrebna informacija o minimalnom i maksimalnom elementu u stablu, i da su nam potrebni klasteri, odnosno niz vEB stabala, kao i sažetak, predstavljen takođje vEB stablom sa informacijama o klasterima. Kako unutar strukture vEB stabla imamo istu tu strukturu, vEB stablo definišemo kao rekurzivnu strukturu.

vEB stablo podržava naredne operacije:

- upis elementa u vEB stablo,
- brisanje elementa iz vEB stabla,
- pronalaženje minimuma i maksimuma stabla,
- proveravanje da li je neki element u stablu i
- traženje prethodnika i sledbenika nekog elementa.

Sve ove operacije biće detaljnije opisane kroz implementaciju u programskom jeziku C++. Jedan od načina na koji možemo napisati definiciju strukture vEB stabla je sledeći:

```
1  class vEB{
2  public:
3    vEB(int x);
4    void upisi(int x);
5    void obrisi(int x);
6    int unutra(int x);
7    int sledbenik(int x);
8    int prethodnik(int x);
9  private:
10   int min;
11   int max;
12   vector<struct vEB*> klasteri;// u i-tom klasteru se nalaze
       brojevi od i*sqrt(u) do (i+1)*sqrt(u)-1 siftovani na 0-sqrt(u)-1
13   struct vEB* info; // ako i-ti klaster nije prazan u info se
       nalazi broj i
```

```
14   unsigned long velicina;
15   unsigned velicina_klastera; //sqrt(velicina) a i broj vEB stabla
      u vektoru klasteri
16 };
```

S obzirom na to da je struktura vEB stabla rekurzivna, i većina algoritama za rad sa ovom strukturom će biti rekrzivna. U samoj strukturi dovoljno je čuvati informaciju o veličini stabla ili veličini klastera jer su one međusobno jednoznačno određene ($velicina\_klastera = \sqrt{velicina}$), ali može olakšati zapis ako imamo obe ove vrednosti. Ovako definisana struktura bi imala prostornu složenost $O(u)$, gde je u veličina univerzuma. To sledi iz činjenice da strukturu vEB stabla čini $\sqrt{u}+1$ struktura dimenzije $\sqrt{u}$, pa dobijamo rekurentnu jednačinu $T(u) = (\sqrt{u}+1)T(\sqrt{u})$ čije je rešenje $O(u)$. Srećom postoje tehnike koje omogućavaju da se prostorna složenost smanji dosta. Međutim, originalna implementacija vEB stabla zahteva $O(u)$ prostora, zato ćemo u ovom radu objašnjavati tu ideju. Svakako, treba istaknuti da neka od poboljšanja poput korišćenja heširanja umesto niza i čuvanje samo klastera koji sadrže neke elemente mogu smanjiti prostornu složenost na $O(n)$, gde je n broj elemenata koje čuvamo u vEB stablu.

U prethodnom poglavlju je objašnjeno da zbog efikasnosti prvi element klastera obrađujemo drugačije od ostalih, odnosno prvi element ne upisujemo rekurzivno, već ga pamtimo samo kao minimalni element klastera. Iz tog razloga postoji veći broj različitih slučajeva koje je potrebno obraditi. Sam kôd upisivanja elementa u stablo bi mogao izgledati ovako:

```
1  void vEB::upisi(int x){
2    if(x == min || x == max)
3      return;
4    if(x > velicina-1) {
5      cout<<"element je van granica\n";
6      return;
7    }
8    if(min == max){//jedan element ili prazno stablo
9      if(min == -1){
10       min = max = x;
11     }
12     else if(x < min) min = x;
13      else max = x;
14     return;
15   }
16   //ako je broj manji od minimuma, u strukturi upisujemo
```

```
17    //trenutni minimum, a novi element postaje minimum
18    //isto vazi i za maksimum
19    if(x < min){
20      swap(min,x);
21    }else if(x > max){
22      swap(max,x);
23    }
24    if(velicina == 2)return;
25    int uklasteru = x % velicina_klastera;
26    int klaster = x / velicina_klastera;
27    if(klasteri[klaster] == nullptr){
28      sazetak->upisi(klaster);
29      klasteri[klaster]= new vEB(velicina_klastera);
30      klasteri[klaster]->min = klasteri[klaster]->max = uklasteru;
31      return;
32    }
33    klasteri[klaster]->upisi(uklasteru);
34  }
```

Nakon razmatranja da li je novi element minimum ili maksimum stabla, nalazi se ključni deo operacije upisivanja elementa u vEB stablo. Proverava se da li je klaster u koji je potrebno upisati element prazan i ukoliko jeste moramo ažurirati sažetak upisivanjem informacije da taj klaster više nije prazan. Možemo primetiti da ne pozivamo da se rekurzivno upiše element u klaster, već ga proglašavamo minimumom i maksimumom klastera. S druge strane, kada je bilo nekog elementa u klasteru, dovoljno je samo upisati taj element u klaster, jer je već upisana informacija da klaster nije prazan u sažetku. Međutim, pošto sada vršimo upis u manje vEB stablo, više ne upisujemo isti broj, već upisujemo vrednost $x \mod velicina\_klastera$, ako se vrši upisivanje u klaster, a $x \,/\, velicina\_klastera$ ako se vrši upisivanje u sažetak strukture.

Moguće je omogućiti i brisanje elemenata iz vEB stabla u složenosti $O(\log \log u)$ i ono se implementira na sličan način kao upisivanje elementa u vEB stablo.

Pronalaženje minimuma i maksimuma stabla je konstantne složenosti jer tu informaciju čuvamo u samom stablu i možemo samo pročitati iz strukture.

Ispitivanje da li se neki element nalazi u stablu je još jedna bitna operacija, i ona je složenosti $O(\log \log u)$. Za razliku od upisivanja elementa, kod provere nije potrebno izvršavati operacije nad sažetkom stabla. Dovoljno je pretražiti odgovarajući klaster i ustanoviti da li se traženi element nalazi u tom klasteru rekurzivno. Primer kôda za pretragu elementa u strukturi možemo videti ovde:

```
int vEB::pretraga(int x){
  if(x == min || x == max) return true;
  if(velicina <= 2) return false;
  int klaster = x / velicina_klastera;
  int uklasteru = x % velicina_klastera;
  if(klasteri[klaster] == nullptr) return false;
  return klasteri[klaster]->pretraga(uklasteru);
}
```

2.3 Problem prethodnika uz pomoć vEB stabla

Razmotrimo na kraju implementaciju nama posebno zanimljivog dela, a to je nalaženje prethodnika i sledbenika nekog elementa korišćenjem vEB stabla. Pretpostavimo da tražimo prethodnika broja x u vEB stablu. Napomenimo da ukoliko bismo tražili sam broj x, tada bi imalo smisla pretraživati klaster u kom bi broj x završio, a to je klaster sa rednim brojem $k = x/velicina\_klastera$ u nizu klastera vEB stabla. U zavisnosti od minimuma klastera k možemo imati dva slučaja. Jednostavniji slučaj je kada je minimum klastera k manji od x: to bi značilo da u tom klasteru sigurno imamo bar jedan element koji je manji od x. Samim tim prethodnik broja x se nalazi u klasteru k i tada je problem dimenzije $O(u)$ smanjen na identičan problem dimenzije $O(\sqrt{u})$ koji možemo rešiti rekurzivno. U slučaju kada je broj x manji od minimuma klastera k problem postaje nešto složeniji. Naime, u tom slučaju element koji tražimo nije u klasteru k već u nekom od prethodnih klastera. Ako bismo proveravali prvi prethodni klaster, taj klaster bi mogao biti prazan. Samim tim se ni u njemu ne bi našao prethodnik broja x. Stoga je potrebno na neki malo smisleniji način utvrditi u kom klasteru će se sigurno naći bar jedan element. Kako sažetak vEB stabla sadrži redne brojeve klastera koji imaju bar jedan element, da bismo našli klaster koji sadrži element manji od broja x, tražićemo prethodnika broja k u sažetku stabla. Sažetak je vEB stablo dimenzije $\sqrt{u}$, te smo problem uspešno smanjili kao i u prvom slučaju. Međutim, sada ne dobijamo konačan odgovor, već dobijamo informaciju u kom klasteru se nalazi prethodnik broja x. Kako su u tom klasteru svi elementi sigurno manji od broja x, potrebno je naći maksimum tog klastera. Pošto u vEB stablu imamo informaciju o minimumu i maksimumu, a svaki klaster je vEB stablo, možemo pročitati maksimum klastera u konstantnom vremenu. Tako dobijamo odgovor i u slučaju kada je broj x manji od svih elemenata

klastera u kom bi se on trebao naći. Složenost operacije prethodnika elementa u vEB stablu se može izraziti već viđenom rekurentnom jednačinom $T(u) = T(\sqrt{u}) + O(1)$, čije je rešenje $O(\log \log u)$.

Primetimo da nam je, kada radimo sa vEB stablom, korisnije da `prethodnik(x)` bude element koji je strogo manji od x. Ovo je zbog toga što u slučaju da pretražujemo sažetak stabla, a klaster k sadrži neke elemente, ali su oni svi veći od broja x, tražimo prethodnika klastera k. Međutim, kao odgovor nam ne odgovara da dobijemo klaster k jer se u njemu ne nalazi prethodnik elementa x. Samim tim ćemo podrazumevati da `prethodnik(x)`, ako u stablu postoji prethodnik broja x, vraća broj koji je strogo manji od broja x. Prisetimo se da smo na početku problem definisali tako da traži element koji je manji ili jednak broju x. To možemo jednostavno postići tako što ćemo poziv `prethodnik(x)` zameniti pozivom `prethodnik(x+1)`. Takođe, kako vEB stablo može čuvati brojeve od 0 do $u - 1$, ukoliko ne postoji element stabla koji je manji od x možemo vratati -1 kao rezultat poziva funkcije, jer smo sigurni da taj element nije u stablu.

```
1  int vEB::prethodnik(int x){
2    if(x > max) return max;
3    if(velicina <= 2) return min;
4    if(x <= min) return -1;
5    int klaster = x / velicina_klastera;
6    int uklasteru = x % velicina_klastera;
7    if(klasteri[klaster] != nullptr
8          && uklasteru > klasteri[klaster]->min){
9      return klaster*velicina_klastera
10             + klasteri[klaster]->prethodnik(uklasteru);
11     }else if(klasteri[klaster] == nullptr
12               || uklasteru <= klasteri[klaster]->min){
13       int tmp = sazetak->prethodnik(klaster);
14       if(tmp == -1) {return min;}
15       return tmp * velicina_klastera + klasteri[tmp]->max;
16   }else{
17     cerr<<"greska";
18     return -1;
19   }
20 }
```

Sledbenika nekog elementa možemo dobiti na veoma sličan način.

Glava 3

X-brzo prefiksno stablo

3.1 Opis strukture podataka

Otprilike deceniju posle vEB stabla Dan Willard je predložio *x-brzo prefiksno stablo* [11] (eng. X-fast trie) kao prostorno efikasniju varijantu vEB stabla. Iako rešavaju slične probleme u sličnoj složenosti koncepti ove dve strukture podataka su dosta drugačiji. Ideja koja stoji iza x-brzog prefiksnog stabla je korisćena za čitav niz naprednijih struktura: jedna od njih koja će biti obrađena u ovom radu je *y-brzo prefiksno stablo* (eng. Y-fast trie).

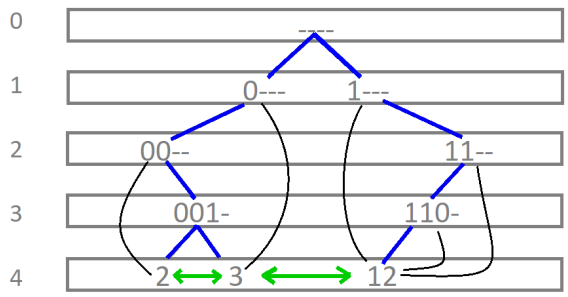
X-brzo prefiksno stablo predstavlja uopštenje klasičnog prefiksnog stabla [5]. Ono za smeštanje podataka koristi binarno stablo, tako što brojeve razdvaja na pojedinačne bitove, i u svakom čvoru čuva prefiks broja koji se nalazi u stablu. Visina x-brzog prefiksnog stabla je $O(\log_2 u)$ gde je u najveći broj koji se može smestiti u stablo. Dakle, visina stabla ne zavisi od broja elemenata koji se nalaze u strukturi podataka, već je visina stabla oređena brojem mogućih elemenata koji se mogu naći u strukturi. Stoga, kao i vEB stablo, x-brzo prefiksno radi sa univerzumom oblika $\{0, 1, ..., u-1\}$ gde je u najveći broj koji se može smestiti u strukturu podataka. Kako x-brzo prefiksno stablo radi sa bitovima, obično se u bira tako da je oblika 2^a, jer je u tom slučaju prostor najbolje iskorišćen. Svaki list x-brzog stabla čuva potpun zapis broja koji je u stablu, dok unutrašnji čvorovi čuvaju prefikse tih elemenata u binarnom zapisu. Odnosno ako unutrašnji čvor čuva prefiks 101 njegov levi potomak bi čuvao prefiks 1010, dok bi desni potomak čuvao 1011. Kako se u svakom nivou stabla dopisuje jedan bit, na nivou lista ćemo imati potpun zapis broja. U samom stablu se čuvaju samo oni čvorovi koji odgovaraju prefiksima brojeva u stablu. Na primer ukoliko imamo broj čiji je prefiks 1011 imaćemo čvorove koji čuvaju prefikse

1,10,101 i 1011, dok u slučaju da u stablu ne postoji element sa prefiksom 1010, nećemo imati prefiks 1010 kao ni bilo koji drugi prefiks koji počinje sa 1010. Po ovom principu funkcioniše klasično prefiksno stablo i visina ovakvog stabla je $\theta(\log u)$, kao i složenost skoro svih operacija koje su nam od interesa.

Da bismo našli prethodnika broja x u ovakvoj strukturi potrebno je naći najduži zajednički prefiks elementa u stablu sa brojem x. To možemo postići prateći od korena redom bitove broja x sa leva na desno, tako što kad naiđemo na nulu idemo u levog potomka tekućeg čvora, dok u slučaju jedinice prelazimo u desnog potomka. U slučaju kada za tekući bit prefiksa ne postoji odgovarajući potomak čvora, tada čvor u kom se nalazimo čuva najduži zajednički prefiks broja x sa proizvoljnim elementom stabla. Nazovimo taj čvor y. U zavisnosti od položaja čvora y u stablu može biti jednostavnije ili složenije naći prethodnika elementa. Najjednostavniji slučaj je kada čvor y ima levog potomka i tada je prethodnik broja x maksimalni element levog podstabla. Problem postaje složeniji u slučaju kada čvor ima samo desnog potomka, jer se u tom slučaju moramo vraćati kroz stablo dok ne nađemo neki čvor koji ima levog potomka. Ovo možemo prevazići tako što ćemo listove stabla, odnosno same elemente, smestiti u dvostruko povezanu listu, i to u sortiranom poretku. Drugim rečima, kada bismo nacrtali stablo, najleviji list bi bio prvi element dvostruko povezane liste, onda prvi sledeći sa leve strane bi bio drugi i tako dalje. Na ovaj način u slučaju kada u stablu ne postoji levi potomak čvora y, možemo jednostavno pronaći element koji je njegov sledbenik, odnosno najmanji element desnog podstabla. Prethodnika elementa x dobijamo tako što se vratimo jedan element kroz ovu dvostruko povezanu listu. Složenost ove operacije je $O(\log u)$.

Pokušajmo da popravimo efikasnost ove operacije. Do sada smo linearno pratili čvorove kroz stablo da bismo našli čvor koji sadrži najduži zajednički prefiks sa brojem x. Da li je moguće ovaj čvor pronaći brže? Možemo probati da binarnom pretragom proverimo da li postoji unutrašnji čvor stabla koji sadrži levu polovinu bitova broja x. Ukoliko bi postojao, onda bismo mogli tražiti čvor sa dužim prefiksom od polovine na identičan način, a inače bismo tražili sa kraćim prefiksom. Ovako bismo element sa najdužim zajedničkim prefiksom mogli pronaći dosta efikasnije, u složenosti $O(\log \log u)$. To možemo postići tako što ćemo sve čvorove čuvati u heš mapi, gde su ključevi sami prefiksi, a vrednosti sami čvorovi stabla. Sada možemo prefikse broja x pretražiti binarnom pretragom i efikasno naći najduži zajednički prefiks, odnosno čvor y. Međutim, to nam i dalje nije dovoljno da nađemo prethodnika elementa u složenosti $O(\log \log u)$. Naime, kako bismo od čvora y morali da se

spustimo do lista prateći pokazivače iz čvorova, dolazimo opet do složenosti $O(\log u)$ u najgorem slučaju. Pokazuje se da ne možemo primeniti istu taktiku i binarnom pretragom tražiti put kojim bismo išli, jer su svi putevi mogući, pa nam samim tim ovde heš mapa ne može pomoći. Međutim možemo se poslužiti drugačijim trikom. Za čvor y sa sigurnošću možemo tvrditi da nema oba potomka. Ovo je zato što bi onda sigurno postojao čvor koji ima duži zajednički prefiks sa brojem x. Stoga u čvoru y postoji jedan pokazivač koji se ne koristi. Samim tim možemo ga iskoristiti da ubrzamo put do lista koji nam treba. Naime, ako čvor y nema levog potomka, mogli bismo povezati taj pokazivač na čvor koji sadrži najmanji element koji počinje prefiksom koji se čuva u čvoru y, odnosno sledbenika broja x. Dok u slučaju da nema desnog potomka možemo povezati taj pokazivač sa najvećim brojem u stablu koji počinje tim prefiksom, odnosno sa prethodnikom broja x. Primetimo da sada kada nađemo čvor koji sadrži najveći zajednički prefiks sa brojem x imamo direktnu vezu ili do prethodnika ili do sledbenika, samim tim možemo u konstantnom vremenu odrediti jedno od ta dva. U slučaju da tražimo prethodnika broja x, a imamo vezu do sledbenika, to možemo jednostavno uraditi tako što nađemo sledbenika, i onda se u dvostruko povezanoj listi vratimo unazad jedan element. Ovo možemo uraditi jer između prethodnika i sledbenika nekog broja u skupu ne može postojati ni jedan element.



Slika 3.1: Struktura x-brzog prefiksnog stabla

Na slici 3.1 prikazana je struktura x-brzog prefiksnog stabla u koje se mogu smestiti brojevi $0, 1, \ldots, 15$. Za zapis broja 15 su nam potrebna 4 bita, samim tim imamo pet nivoa u stablu, odnosno 5 heš mapa. Svaka čuva čvorove sa prefiksima odgovarajuće dužine. Kao što možemo primetiti prefiksi koji nemaju oba potomka imaju pokazivač na list, koji sadrži čvor koji nam olakšava nalaženje prethodnika ili sledbenika broja koji bi imao taj prefiks. Primetimo takođe da su svi listovi redom povezani u dvostruko povezanu listu.

3.2 Implementacija

Sama implementacija x-brzog prefiksnog stabla nije toliko složena. Najzahtevniji deo implementacije je upisivanje brojeva u strukturu, zbog postojanja velikog broja slučajeva, i potrebe za netrivijalnim povezivanjem odgovarajućih elemenata u strukturi. Pored toga, potrebno je ubaciti sve prefikse datog broja u heš mape. Kako uglavnom ne možemo čuvati proizvoljan broj bitova, čuvaćemo cele brojeve kao prefikse. Na primer, prefiks 5 za čiji binarni zapis nam je dovoljno samo 3 bita, čuvaćemo u promenljivoj tipa int za čiji zapis se koristi bar 32 bita. Samim tim broj 5, biti prefiks broja 11, koji se binarno zapisuje 1011. Iz tog razloga ne možemo čuvati jednu heš mapu, jer nećemo znati da li je broj 5 prefiks dužine 3, neki veći prefiks ili je broj koji se čuva u strukturi. Ako gledamo kao prefiks dužine 3 to znači da u strukturi imamo element čija su 3 najviša bita 101, ako je dužine 5 to znači da je 5 najviših bitova nekog broja u stablu jednako 00101, dok u slučaju da je to element stabla imaćemo da su najniža tri bita elementa 101 i svi ostali bitovi su jednaki nuli. Stoga ćemo čuvati po jednu heš mapu za svaku dužinu prefiksa elemenata koji se nalaze u stablu. Odnosno, ukoliko je $u = 2^{32} - 1$ najveći broj koji se može zapisati u stablu, imaćemo prefikse svih dužina od 1 do 32, jer je potrebno 32 bita za zapis najvećeg broja. Samim tim imaćemo niz od 32 heš mape, koje će čuvati prefikse različitih dužina. Sada se bilo koji broj može naći u vise heš mapa, ali svako od tih pojavljivanja imaće različito značenje. Sve te heš mape će se čuvati u jednom namenskom nizu. Ukoliko čuvamo elemente sa brojem bitova w, u nizu na indeksu w će se naći heš mapa koja sadrži elemente stabla. U svim ostalim mapama će biti prefiksi elemenata, odnosno heš mapa na indeksu i će sadržati prefikse dužine i. To znači da ukoliko se broj 5 (binarno 101) ne nalazi u heš mapi sa indeksom 3, u stablu nemamo nijedan element koji će početi sa tim binarnim prefiksom. Sam taj niz heš mapa može predstavljati strukturu x-brzog prefiksnog stabla. Na indeksu nula u tom nizu bila bi heš mapa sa jednim elementom, i to je koren stabla, jer koren stabla odgovara praznom prefiksu (dužine 0). Direktni potomci korena bi bili čvorovi u heš mapi koja ima indeks 1, i to levi potomak na indeksu 0, a desni potomak na indeksu 1 u istoj heš mapi. Struktura x-brzog prefiksnog stabla bi se mogla definisati na sledeći način:

```
class x_brzo_prefiksno{
  int w;
  vector<unordered_map<int, x_cvor*>> mapa;
}
```

gde w predstavlja broj bitova u binarnom zapisu najvećeg broja koji možemo zapisati u x-brzo prefiksno stablo, dok je `x_cvor` je struktura koja predstavlja čvor stabla, i njena definicija može izgledati ovako:

```
struct x_cvor{
  int nivo;
  int vrednost;
  x_cvor *levo, *desno;
};
```

Prefiks u strukturi `x_cvor` je ključ po kojem tražimo čvorove u heš mapama. Pokazivače levo i desno koristimo ne samo za direktne potomke, već ukoliko neki čvor nema oba direktna potomka i za efikasnije izračunavanje prethodnika i sledbenika broja sa prefiksom čuvanim u čvoru. Za svaki broj koji čuvamo u stablu imamo $\log u$ različitih prefiksa, koji se čuvaju zasebno, te na taj način dolazimo do prostorne složenosti $O(n \log u)$ [11]. Iako struktura sama po sebi ne sadrži puno informacija, upisivanje broja u x-brzo prefiksno stablo nije tako jednostavno. Naime, kada upisujemo broj, moramo upisati sve njegove prefikse koji nisu unutar heš mapa. Svaki prefiks je jedan unutrašnji čvor, samim tim na početku upisa, kada je struktura prazna, moramo upisati sve prefikse. Takođe, kada upišemo neki prefiks, on nema oba direktna potomka, pa njegov drugi pokazivač moramo povezati na odgovarajući način. Kako je kôd upisivanja broja u x-brzo prefiksno stablo malo duži, ovde ćemo dati samo pseudokod:

```
void upisi(int x)
  - Nadji prethodnika i sledbenika broja x
  - Ubaci cvor koji ima vrednost x izmedju prethodnika i
    sledbenika
  - Kreni od cvora x i popuni prefikse koji nisu u stablu
  - Za svaki ubaceni prefiks povezi drugi pokazivac na
    odgovarajuci cvor
```

Pošto je visina stabla $O(\log u)$, a mi prolazimo ceo put od korena do lista kako bismo upisali neki broj, složenost operacije upisivanja broja u x-brzo prefiksno stablo je $O(\log u)$ [11]. Na sličan način se izvršava i brisanje elementa iz strukture, i iste je

složenosti.

Kada upišemo brojeve u stablo, jednostavno možemo proveriti da li je neki broj u stablu. Kako nam se vrednosti elemenata nalaze kao ključevi heš mape koja je poslednja u nizu, provera da li je neki broj u x-brzom prefiksnom stablu je jednostavna i efikasna. Dovoljno je da u heš mapi proverimo da li postoji ključ u heš mapi koji je jednak broju koji tražimo. Kôd provere da li se broj nalazi u x-brzom prefiksnom stablu može izgledati ovako:

```
x_cvor* x_pretraga(int k){
  if(mapa[w].find(k) == mapa[w].end())
    return nullptr;
  return mapa[w][k];
}
```

Kako x-brzo prefiksno stablo u velikoj meri oslanja na tehniku heširanja, bitno je da heš mapa koja se koristi može pristupati elementima mape u amortizovanoj složenosti $O(1)$.

3.3 Problem prethodnika uz pomoć x-brzog prefiksnog stabla

Kao što smo već pomenuli, za upisivanje elementa u strukturu podataka nam je potrebno da izračunamo prethodnika i sledbenika broja. To je zato što je za svako upisivanje broja potrebno da ažuriramo dvostruko povezanu listu elemenata. Ovo znači da iako x-brzo prefiksno stablo planiramo da koristimo za statički problem prethodnika, usputno ćemo rešavati i dinamičku verziju, jer će se skup svakim ubacivanjem nekog broja menjati.

Prethodnika nekog broja ćemo tražiti iz dva dela, od kojih je prvi pronalaženje čvora sa najdužim zajedničkim prefiksom broja sa elementima iz stabla, a drugi računanje prethodnika sa prefiksom u tom čvoru. Neka je broj x broj za koji tražimo najduži zajednički prefiks sa elementima stabla. Heš mape nam omogućavaju da najduži zajednički prefiks broja x sa elementima stabla pronađemo binarnom pretragom. U konstantnoj vremenskoj složenosti možemo proveriti da li se u stablu nalazi element koji ima prefiks proizvoljne dužine isti kao i x. To možemo tako što ćemo proveriti da li u stablu, odnosno u heš mapi prefiksa tražene dužine, postoji čvor koji sadrži odgovarajući prefiks broja x. Jasno je da ako on ne postoji, onda ne možemo imati ni element koji ima isti prefiks. Primetimo da postojanje prefiksa

zadovoljava svojstvo monotonosti: ako ne postoji prefiks date dužine onda sigurno neće postojati ni prefiks veće dužine; slično, ako postoji prefiks date dužine, sigurno će postojati i prefiks proizvoljne manje dužine. Stoga, možemo primeniti binarnu pretragu po prefiksima za rešavanje ovog problema. Nazovimo traženi čvor sa y. Kada nađemo čvor y koji sadrži traženi prefiks, potrebno je iskoristiti pokazivače iz stabla da bismo došli do prethodnika. U zavisnosti od toga da li je prvi bit posle prefiksa broja x jedinica ili nula imamo dva slučaja. Ako je taj bit jednak 1, to znači da čvor y nema pravog desnog potomka, inače bi postojao duži prefiks. Samim tim taj pokazivač je iskorišćen da se poveže sa prethodnikom bilo kog broja koji bi imao taj prefiks, pa samim tim i broja x. To znači da prethodnika broja x možemo naći u čvoru na koji pokazuje pokazivač iz čvora y. U slučaju da je prvi bit posle prefiksa broja x bio 0, čvor y nema levog direktnog potomka, pa je taj pokazivač iskorišćen za sledbenika brojeva sa tim prefiksom. Sada kada imamo sledbenika, njegov pokazivač za levog potomka predstavlja prethodnika broja x. U oba slučaja od čvora y praćenjem pokazivača možemo doći do prethodnika broja x. Vremenska složenost postupka kojim se polazeći od čvora y dolazi do prethodnika elementa x je $O(1)$, dok je vremenska složenost pronalaska čvora y $O(\log \log u)$. Stoga je ukupna vremenska složenost operacije računanja prethodnika broja x u x-brzom prefiksnom stablu $O(\log \log u)$.

```
x_cvor* x_prethodnik(int k){
  int visi_nivo = 0,
      nizi_nivo = w + 1,
      sredina, prefiks;

  x_cvor *tmp = nullptr;
  while(nizi_nivo - visi_nivo > 1){
    sredina = (visi_nivo + nizi_nivo) >> 1;
    prefiks = k >> (w - sredina);
    if(mapa[sredina].find(prefiks) == mapa[sredina].end())
      nizi_nivo = sredina;
    else{
      visi_nivo = sredina;
      tmp  = mapa[sredina][prefiks];
    }
  }

  if(tmp == nullptr || tmp->nivo == 0)
    return nullptr;
```

```
20
21    if(tmp->nivo == w)
22      return tmp;
23
24    if((k >> (w - tmp->nivo - 1)) & 1)
25      tmp = tmp->desno;
26    else
27      tmp = tmp->levo;
28
29    if(tmp->vrednost > k){
30      return tmp->levo;
31    }
32    return tmp;
33  }
```

Sledbenik broja se nalazi na identičan način kao i prethodnik broja, i iste je složenosti. Samim tim neće biti objašnjavan detaljnije.

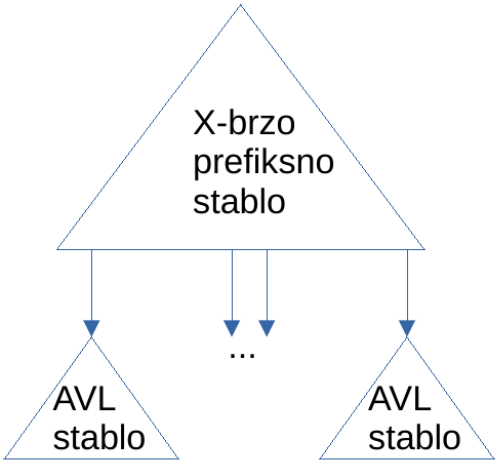
Glava 4

Y-brzo prefiksno stablo

4.1 Opis strukture podataka

Y-brzo prefiksno stablo (eng. y-fast trie) je predstavljeno 1983. godine, u istom radu kao i x-brzo prefiksno stablo i predstavlja proširenje x-brzog prefiksnog stabla [11]. Ono takođe radi nad celim brojveima iz ograničenog domena. Naime, y-brzo prefiksno stablo koristi x-brzo prefiksno stablo, uz dodatak samobalansirajućih uređenih binarnih stabala. Y-brzo prefiksno stablo omogućava vremenski efikasnije operacije ažuriranja skupa, odnosno upisivanja elementa i brisanja elementa iz skupa. Takođe, prostorna složenost y-brzog prefiksnog stabla je manja od prostorne složenosti odgovarajućeg x-brzog prefiksnog stabla. Ova struktura podataka se sastoji iz dva dela, od kojih je jedan x-brzo prefiksno stablo, a drugi je cela kolekcija samobalansirajućih uređenih binarnih stabala, na primer AVL stabala [1].

Sama struktura podataka je organizovana na sledeći način: elementi y-brzog prefiksnog stabla se nalaze u AVL stablima, dok se x-brzo prefiksno stablo koristi da odredimo u kom AVL stablu se nalazi element. Kako je složenost svih operacija nad AVL stablima $O(\log n)$ gde je n broj elemenata u stablu, a mi želimo da postignemo da složenost svih osnovnih operacija u y-brzom prefiksnom stablu bude $O(\log \log u)$ gde je u najveći broj koji možemo smestiti u stablo, ne sme važiti da je n mnogo veće od $\log u$, odnosno AVL stabla ne smeju imati više od $O(\log u)$ elemenata. To se može postići tako što kada se u neko od AVL stablala ubaci prevelik broj elemenata to stablo se deli na dva dela. Takođe, ne želimo ni da imamo previše AVL stabala, te u slučaju da neko stablo postane malo, spojićemo ga sa susednim stablom. Ovo ćemo raditi kada broj elemenata u nekom AVL stablu bude veći od $2 \cdot \log u$, odnosno manji od $(\log u)/2$. Svako AVL stablo će sadržati uzastopne elemente skupa, odnosno ako



Slika 4.1: Ilustracija y-brzog prefiksnog stabla

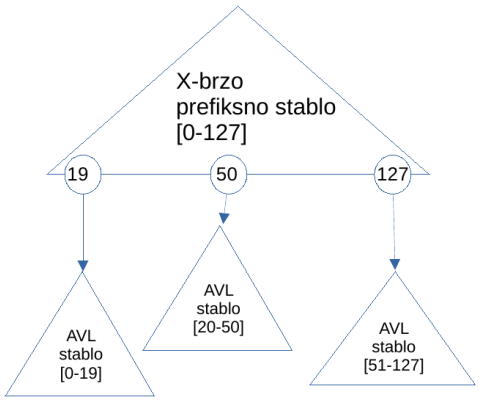
y-brzo prefiksno stablo čuva skup S, i važi $x, y, z \in S$ i $x < y < z$ ne može se desiti da x i z pripadaju jednom AVL stablu, a y nekom drugom. Kako imamo više AVL stabla, potrebno je odrediti u kom AVL stablu treba tražiti neki element. Ako to uspemo problem smo sveli na obradu stabla sa relativno malo elemenata. Međutim, nije toliko jednostavno odrediti u kom stablu treba tražiti element, jer elementi u skupu mogu biti nepravilno raspoređeni. Dodatno, moguće je da neko stablo pokriva veći interval brojeva od drugog, iako imaju približno isti broj elemenata. Na primer, može se desiti da imamo malu gustinu malih brojeva u skupu, a onda veliku gustinu brojeva bliskih nekom velikom broju x. Tada će početna stabla koja imaju $O(\log u)$ elemenata pokrivati veći interval, odnosno razlika između najmanjeg i najvećeg broja u stablu će biti veća u početnim stablima, nego u stablu koje čuva elemente bliske broju x. Samim tim se problem određivanja AVL stabla kome pripada neki element usložnjava.

Da bismo mogli da efikasno odredimo AVL stablo nad kojim treba raditi koristićemo x-brzo prefiksno stablo. Svako AVL stablo imaće svog predstavnika, kog ćemo smestiti u x-brzo prefiksno stablo. Taj predstavnik ne mora biti jedan od elemenata, ali mora zadovoljavati neke uslove. Naime, za predstavnika treba da važi da je veći ili jednak od svih elemenata iz stabla koje predstavlja, i manji od svih elemenata stabla koje predstavlja najmanji sledeći predstavnik, odnosno manji od svih elemenata prvog sledećeg stabla. Intuitivno predstavnik stabla predstavlja granicu do kog broja se elementi smeštaju u jedno AVL stablo, a od kog u drugo. To znači da kada tražimo broj a u y-brzom prefiksnom stablu, tražićemo AVL stablo

u kom bi se broj a trebao naći, tako što nađemo njegovog sledbenika u x-brzom prefiksnom stablu. Na taj način smo našli predstavnika AVL stabla i na jedinstven način odredili u kom AVL stablu treba nastaviti pretragu. Na primer, ako tražimo broj 23 u y-brzom prefiksnom stablu ilustrovanom na slici 4.2 tražimo sledbenika broja 23 u x-brzom prefiksnom stablu koje sadrži brojeve $\{19, 50, 127\}$, i to je broj 50. Na taj način smo odredili AVL stablo u kom se može naći broj 23. Sada smo problem sveli na ispitivanje da li se broj 23 nalazi u relativno malom balansiranom uređenom binarnom stablu.

Postavlja se pitanje kako efikasno održavati predstavnike stabala tako da zadovoljavaju prethodni uslov i koliko uopšte AVL stabala imamo u y-brzom stablu? Jedan veoma jednostavan, a elegantan način za rešavanje ovog problema je da kada upisujemo prvi element u y-brzo prefiksno stablo, predstavnika prvog i u tom trenutku jedinog AVL stabla postavimo na u, odnosno na najveći broj koji se može naći u stablu. Prilikom narednih upisa elemenata u y-brzo prefiksno stablo, ovom AVL stablu će se povećavati broj elemenata. U trenutku kada broj elemenata u stablu postane $2 \log u$, delimo to stablo na dva manja, tako što koren AVL stabla postaje predstavnik i element stabla koje sadrži elemente AVL stabla sa vrednostima manjim od korena, a ostali elementi ostaju u stablu sa predstavnikom u. Na ovaj način smo obezbedili da je predstavnik stabla veći ili jednak od svih elemenata u stablu koje predstavlja. Ovo je samo jedan od načina na koji možemo birati predstavnike stabala. Način na koji biramo predstavnike i redosled kojim upisujemo elemente u strukturu može uticati na broj AVL stabala u kolekciji. Kako je veličina svakog AVL stabla između $2 \cdot \log u$ i $(\log u)/2$, imaćemo između $n/(2 \cdot \log u)$ i $2n/(\log u)$ stabala, gde n predstavlja broj elemenata u y-brzom stablu. To je ujedno i broj predstavnika, pa će toliko elemenata biti u x-brzom prefiksnom stablu. Kako je prostorna složenost x-brzog stabla $O(a \log u)$ gde je a broj elemenata x-brzog prefiksnog stabla, a u veličina univerzuma, dobijamo da je prostorna složenost dela sa predstavnicima u y-brzom prefiksnom stablu $O(n)$, jer je $a \in O(n/(\log u))$. U AVL stablima imamo ukupno n elemenata, te je prostorna složenost ovog dela strukture $O(n)$. Ukupno dobijamo da y-brzo prefiksno stablo ima prostornu složenost $O(n)$ [7].

Kao i x-brzo prefiksno stablo, y-brzo prefiksno stablo omogućava efikasno rešavanje problema prethodnika i sledbenika nekog broja u skupu. Da bismo odredili prethodnika nekog broja potencijalno moramo proveriti dva AVL stabla. Ovo je zato što na osnovu x-brzog prefiksnog stabla možemo odrediti u kom intervalu su brojevi u AVL stablima, ali ne i da li postoji neki broj u određenom delu intervala.



Slika 4.2: Predstavnici AVL stabala u y-brzom prefiksnom stablu

Na primer, ako tražimo prethodnika broja 25 u stablu sa slike 4.2 nama niko ne garantuje da u AVL stablu koje pokriva interval $[20 - 50]$ imamo element koji je manji od 25. U slučaju da ne postoji traženi element u tom stablu, odgovor ćemo tražiti u AVL stablu čiji je predstavnik 19. I tu možemo biti sigurni da se nalazi traženi prethodnik, jer AVL stabla imaju najmanje $(\log u)/2$ elemenata.

4.2 Implementacija

Najveći deo implementacije y-brzog prefiksnog stabla je povezivanje x-brzog prefiksnog stabla sa AVL stablima. Pretpostavimo da su operacije nad AVL stablima, i x-brzim prefiksnim stablom već poznate. Sada je potrebno samo pametno iskombinovati sve te operacije. Struktura y-brzog prefiksnog stabla u sebi ima jedno x-brzo prefiksno stablo i više AVL stabala, koja će biti čuvana u heš mapi. Klasa koja će predstavljati y-brzo prefiksno stablo može se zadati na naredni način:

```
class YfastTrie{
  int w;
  XfastTrie predstavnici;
  unordered_map<int, Node*> stabla;
  unordered_map<int, int> velicina_stabla;
}
```

Kao što je već objašnjeno u prethodnom poglavlju, x-brzo prefiksno stablo ćemo koristiti da bismo odabrali jedno od AVL stabala u heš mapi stabla. Dodatnu heš mapu za veličinu stabla ćemo koristiti da proveravamo koliko svako AVL stablo ima elemenata. Nju ćemo koristiti da bismo znali kada podeliti AVL stablo na dva dela,

ili kada spojiti dva AVL stabla u jedno. Ukoliko ne bismo čuvali informaciju o broju elemenata u stablu, morali bismo da je računamo svaki put, za šta je potrebno $O(\log u)$ vremena, što bi ugrozilo složenost ostalih operacija za rad sa y-brzim prefiksnim stablom. Ako želimo da upišemo broj x u y-brzo prefiksno stablo, potrebno je odrediti u koje AVL stablo treba upisati broj, a to možemo naći tako sto nađemo predstavnika AVL stabla u koje treba upisati broj. Kao što smo već pomenuli, to je moguće postići pronalaženjem sledbenika broja x u x-brzom prefiksnom stablu. Na ovaj način smo odredili predstavnika AVL stabla u koje ćemo upisivati element. Ukoliko to AVL stablo ima manje od $2 \log u$ elemenata, dovoljno je jednostavno upisati broj u AVL stablo. U ovom slučaju složenost operacije upisivanja broja u y-brzo prefiksno stablo bi bila $O(\log \log u)$. Međutim, u slučaju da je AVL stablo veliko potrebno je podeliti to stablo na dva dela. Do situacije da je potrebno podeliti AVL stablo na dva dela se dolazi nakon bar $\log u$ upisivanja u stablo. Zaključujemo da se prvi slučaj kada nije potrebno menjati x-brzo prefiksno stablo mnogo češće dešava. Složenost operacije upisivanja broja u y-brzo prefiksno stablo u najgorem slučaju jeste $O(\log u)$ i to je slučaj kada je potrebno izmeniti x-brzo prefiksno stablo. Kako se taj najgori slučaj dešava jednom u $O(\log u)$ operacija, amortizovana složenost operacije upisivanja broja u y-brzo prefiksno stablo je $O(\log \log u)$ [11]. Programski kôd operacije upisivanja broja u y-brzo prefiksno stablo dat je u nastavku teksta.

```
1  void y_upisi(int a){
2    x_cvor *sad = predstavnici.x_sledbenik(a);
3    if(sad == nullptr){
4      stabla[u] = napravi_cvor(a);
5      velicina_stabla[u] = 1;
6      predstavnici.x_upisi(u);
7      return;
8    }
9    int predstavnik = sad->vrednost;
10   stabla[predstavnik] = upisi_avl(stabla[predstavnik],a);
11   velicina_stabla[predstavnik] = velicina_stabla[predstavnik]+1;
12   if(velicina_stabla[predstavnik] >= 2*w){
13     avl_cvor* levo;
14     int maks = stabla[predstavnik]->kljuc;
15     stabla[predstavnik] = avl_podeli(stabla[predstavnik],&levo);
16     predstavnici.x_upisi(maks);
17     stabla[maks] = levo;
18     velicina_stabla[maks] = broj_elemenata(levo);
19     velicina_stabla[predstavnik] -= velicina_stabla[maks];
```

```
20   }
21 }
```

Promenljiva `predstavnici` čuva predstavnike AVL stabala, i ona ima strukturu x-brzog prefiksnog stabla. Stoga su metodi `x_upisi` i `x_sledbenik` operacije nad x-brzim prefiksnim stablom. Funkcije `avl_upisi`, `avl_podeli` i `broj_elemenata` su funkcije nad AVL stablom.

Brisanje elementa iz AVL stabla se odvija na sličan način, osim što sada u slučaju da broj elemenata u AVL stablu postane premali spajamo ga sa sledećim. Međutim, treba voditi računa da tako dobijeno spojeno stablo sada može imati prevelik broj elemenata i ako je to slučaj potrebno ga je podeliti na dva nova. To podrazumeva operacije nad x-brzim prefiksnim stablom. Analiza složenosti je identična kao kada upisujemo element u stablo.

Sada kada smo videli na koji način povezujemo x-brzo prefiksno stablo sa AVL stablima, provera da li se broj a nalazi u strukturi je jednostavna. Dovoljno je u x-brzom prefiksnom stablu naći sledbenika broja a, koji će predstavljati ključ u heš mapi čija je vrednost AVL stablo u kom trebamo tražiti broj a. Primetimo da kada pravimo prvo AVL stablo, njegov predstavnik je broj u, odnosno najveći broj koji je moguće upisati u y-brzo prefiksno stablo. To znači da će svaki broj za koji tražimo predstavnika AVL stabla u kom se treba naći imati sledbenika u x-brzom prefiksnom stablu. Složenost operacija sledbenika broja u x-brzom prefiksnom stablu je $O(\log \log u)$, a to je i složenost operacije traženja broja u AVL stablu sa $O(\log u)$ elemenata. To nas dovodi do toga da je složenost provere da li se neki broj nalazi u y-brzom prefiksnom stablu $O(\log \log u)$.

```
1 bool y_sadrzi(int a){
2   int predstavnik = predstavnici.x_sledbenik(a)->vrednost;
3   return avl_sadrzi(stabla[predstavnik],a);
4 }
```

Funkcija `avl_sadrzi` je funkcija nad AVL stablom koja proverava da li se broj a nalazi u AVL stablu sa korenom `stabla[predstavnik]`.

4.3 Problem prethodnika uz pomoć y-brzog prefiksnog stabla

Kao i ostale operacije nad y-brzim prefiksnim stablom, i operacija određivanja prethodnika biće izvršena iz dva dela. Prvi je pronalaženje AVL stabla u kom bi se mogao naći prethodnik elementa, a drugi je računanje prethodnika. Međutim, za razliku od ostalih operacija ovde ne možemo jedinstveno odrediti u kom stablu treba tražiti prethodnika. Međutim, izbor možemo svesti na dva AVL stabla, tako da se prethodnik datog broja nalazi u jednom od ta dva AVL stabla.

Pretpostavimo da tražimo prethodnika broja a. Kada nađemo sledbenika broja a u x-brzom prefiksnom stablu, našli smo i AVL stablo u kom bi se mogao naći prethodnik elementa. Međutim, u tom AVL stablu se mogu naći elementi koji su svi veći od broja a, pa je moguće da nijedan od njih nije prethodnik broja a. Tada prethodnika tražimo u prvom stablu levo od tog stabla, odnosno tražimo prethodnika broja a da bismo odredili predstavnika drugog AVL stabla. Svi elementi ovog AVL stabla su manji od broja a. Dodatno, to stablo sigurno nije prazno, jer po definiciji strukture svako AVL stablo ima bar $O(\log u)$ elemenata, te ćemo sigurno naći prethodnika broja a. Naravno, može se desiti da broj a nema prethodnika u x-brzom prefiksnom stablu. Tada ukoliko u prvom AVL stablu nemamo prethodnika, nećemo imati prethodnika u stablu uopšte.

Prisetimo se da u x-brzom prefiksnom stablu čuvamo elemente stabla u dvostruko povezanoj listi, samim tim kada nađemo sledbenika broja a prethodnik se dobija tako što se za jedno mesto vratimo kroz dvostruko povezanu listu. Takođe, primetimo da je u ovoj implementaciji maksimalni element stabla i predstavnik stabla, osim za poslednje stablo. Samim tim kada nađemo predstavnika stabla koji sadrži manje elemente od broja a imamo i najveći element stabla, pa ne moramo pretraživati stablo. Međutim, kako postoje i drugi načini za odabir predstavnika, u implementaciji je ostavljena pretraga i tog stabla. Sama implementacija operacije se nalazi u nastavku:

```
1 int prethodnik(int a){
2   Node1 *tmp = predstavnici.successor(a);
3   int predstavnik = tmp->key;
4   auto s = avl::prethodnik(stabla[predstavnik],a);
5   if(s.has_value()){
6     return s.value();
7   }
```

```
8   if(tmp->left == nullptr) return -1;
9   predstavnik = tmp->left->key;
10  s = avl::prethodnik(stabla[predstavnik],a);
11  return s.value();
12 }
```

Da bismo našli prethodnika broja a u y-brzom prefiksnom stablu, potrebno je naći sledbenika u x-brzom prefiksnom stablu dimenzije u, i pretražiti dva AVL stabla sa $O(\log u)$ elemenata. Složenost obe operacije je $O(\log \log u)$, pa je samim tim to i složenost nalaženja prethodnika u y-brzom prefiksnom stablu. Pronalaženje sledbenika se odvija analogno. Jedina razlika je to što za ovakav izbor predstavnika AVL stabla ne moramo pretraživati dva AVL stabla, već je moguće dobiti jedinstveno AVL stablo u kom će se naći sledbenik elementa. Naime, kada dobijemo AVL stablo u kom bi se našao broj a, ako u njemu nađemo sledbenika to je sledbenik broja a u y-brzom prefiksnom stablu. Ukoliko u tom AVL stablu ne postoji sledbenik broja a, možemo zaključiti da je to bilo „poslednje" AVL stablo, jer su sa ovim izborom predstavnika, svi predstavnici, sem možda jednog, u y-brzom prefiksnom stablu. Jedini predstavnik koji možda nije u strukturi je u. Tako da kada nađemo sledbenika broja a u x-brzom prefiksnom stablu, i ako to nije u, on je kandidat za sledbenika broja a, koji se nalazi u tom stablu. Na taj način je moguće izbeći proveru dva AVL stabla.

Glava 5

Poređenje struktura na problemu prethodnika

U ovom radu obrađene su neke od naprednijih struktura podataka kojim možemo rešiti problem određivanja prethodnika nekog broja. Detaljno je objašnjeno na koji način koja od struktura funkcioniše, i kako koristi podatke koje čuva da bi se rešila statička verzija problema prethodnika. Tabela 5.1 ukratko prikazuje asimptotske složenosti operacija vezanih za prethodno obrađene strukture podataka. Iako je asimptotska analiza složenosti uglavnom korisna, ne daje nam baš potpune odgovore. Na primer moguće je da se neki algoritam veće asimptotske složenosti mnogo bolje ponaša za neke dovoljno male primere od nekih složenijih algoritama koji su asimptotski efikasniji. Stoga, može se desiti da se u nekim situacijama bespotrebno gube resursi da se nešto asimptotski optimizuje. U daljem tekstu ćemo pokusati da uporedimo strukture podataka koje su obrađene u ovom radu. Da bismo pokušali da dobijemo jasniju sliku o efikasnosti ovih struktura podataka, ovom upoređivanju će biti dodata struktura AVL stabla. Poređenje će biti sprovedeno na ulazima različitih veličina. Pošto je akcenat u ovom radu stavljen na statičku verziju problema prethodnika, merićemo vreme izvršavanja samo operacije računanja prethodnika. Dakle za dalju analizu nam neće biti od značaja vreme potrebno da se generiše struktura podataka koja će u sebi sadržati tražene brojeve. Takođe, kada jednom generišemo potrebnu strukturu nećemo je dalje menjati, već ćemo samo računati prethodnike traženih brojeva.

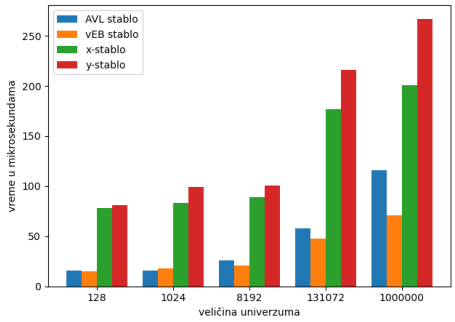
	prethodnik	pretraga	izmena skupa	min/max	prostorna složenost
Sortirani niz	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$	$O(n)$
AVL stablo	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
vEB stablo	$O(\log \log u)$	$O(\log \log u)$	$O(1)$	$O(\log \log u)$	$O(u)$
x-bps	$O(\log \log u)$	$O(1)$	$O(\log u)$	$O(\log \log u)$	$O(n \log u)$
y-bps	$O(\log \log u)$	$O(\log \log u)$	$O(\log \log u)$	$O(\log \log u)$	$O(n)$

Tabela 5.1: Asimptotske složenosti struktura

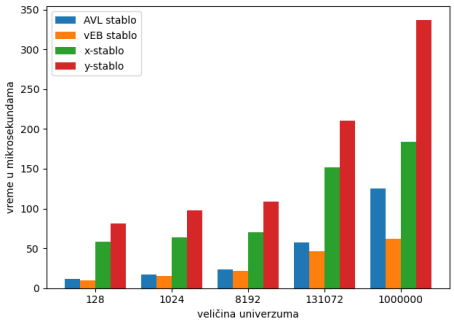
Za implementaciju struktura podataka korišćen je programski jezik C++, a za računanje vremena izvršavanja operacija korišćena je biblioteka za rad sa vremenom `chrono`. Implementacije su dostupne na linku: `https://github.com/kocic98/Masterrad`. Radi pravednosti, sve strukture podataka će biti testirane na identičnim primerima. Svi primeri su slučajno generisani i različitih veličina. Proces testiranja ćemo sprovesti na tri skupa primera: prvi skup primera se odnosi na računanje prethodnika brojeva koji se već nalaze u skupu. Drugi skup primera se odnosi na računanje prethodnika brojeva koji se ne nalaze u skupu, dok će treći skup sadržati probleme obe vrste po pola. Svaki test će biti pokrenut po 5 puta, i biće računato ukupno vreme potrebno za svih 5 pokretanja.

Na dijagramima u nastavku je prikazano vreme potrebno za izvršavanje 20 upita računanja prethodnika broja, i ti programi su pokretani 5 puta za svaki ulaz. Na dijagramu 5.1 prikazan je slučaj kada se brojevi čije smo prethodnike tražili nalaze u skupu. Iako na ovim primerima deluje da je rezultat za AVL stabla bolji od rezultata za x-brzo i y-brzo prefiksno stablo, možemo primetiti da je vreme izvršavanja u odnosu na veličinu razmatranog univerzuma više poraslo u slučaju AVL stabla. Naime, vreme izvršavanja za primere sa relativno malom dimenzijom univerzuma iznosi oko 16 mikrosekundi, dok je izvršavanje za najveći razmatrani primer trajalo oko 113 mikrosekunde, što je otprilike 7 puta sporije. Primetimo, da je ikod ostalih struktura podataka odnos vremena izvršavanja dosta manji i iznosi između 3 i 4. Iako ovo zapažanje u opštem slučaju ne znači ništa, moglo bi nas navesti da bi verovatno na nekom još većem ulazu određivanje prethodnika broja u AVL stablu bilo sporije od x i y brzih prefiksnih stabala. To svakako možemo zaključiti i iz asimptotske složenosti struktura podataka, da bi na dovoljno velikim ulazima x- i y-brza prefiksna stabla trebalo da budu efikasnija.

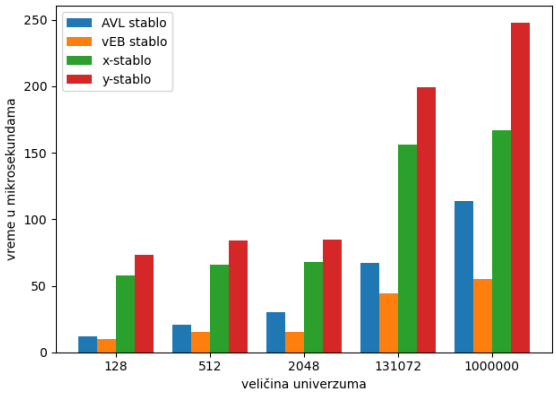
Najveći primer koji je obrađen u ovom radu je sa univerzumom veličine 10^6, i svi primeri su generisani nasumično. Odabir je vršen tako da brojevi čiji se prethodnici traže nisu previše blizu jedan do drugog. Naime, nasumično je generisan skup, a



Slika 5.1: Vremena potrebna za rešavanje problema prethodnika kada su brojevi iz skupa



Slika 5.2: Vremena potrebna za rešavanja problema prethodnika kada se brojevi ne nalaze u skupu



Slika 5.3: Vremena potrebna za rešavanje problema prethodnika sa brojevima iz i van skupa

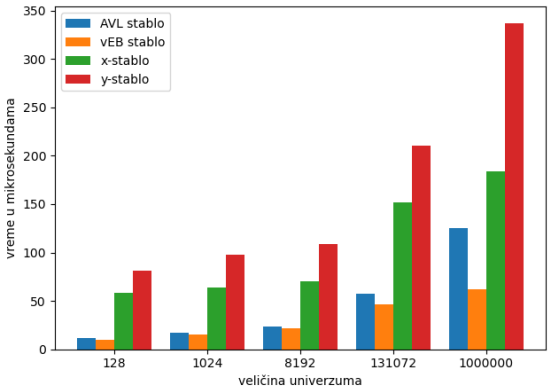
onda je od tih elemenata izabrano 20 brojeva tako da oni nisu blizu jedan drugom. U slučaju primera kada su brojevi čije prethodnike tražimo iz skupa, originalni skup bi ostao nepromenjen, dok bi u slučaju da su elementi van skupa, ti brojevi kada su izabrani bili obrisani iz skupa. Dodatno, primetimo da asimptotska složenost AVL stabla ne zavisi od veličine univerzuma kao ostale strukture, već od broja elemenata u strukturi. Međutim, primeri su pravljeni tako da svaki primer sadrži bar $u/2$ elemenata, gde je u veličina univerzuma.

Pitanje na koje nam ovi primeri ne daju odgovore je do koje veličine ulaza je efikasnije koristiti AVL stabla, a od koje je efikasnije preći na x i y brza prefiksna stabla. Sa druge strane, na dijagramima možemo videti da se vEB stablo ponaša dobro i sa relativno manjim ulazima. Skoro na svim testiranim primerima se vEB stablo pokazalo kao najefikasnije. Stoga možemo zaključiti da su operacije na vEB

stablu efikasne i na ulazima relativno male veličine. Napomenimo i to da se x i y-brza prefiksna stabla dosta oslanjaju na heširanje, i da je u implementaciji korišćena heš mapa iz standardne biblioteke za C++, a može se desiti da je neko drugo heširanje pogodnije za potrebe ovih struktura podataka.

Na dijagramima možemo primetiti da nijedna od prikazanih struktura ne pravi veliku razliku između toga da li se broj čijeg prethodnika tražimo nalazi u strukturi ili ne.

Na dijagramu 5.4 je prikazan slučaj kada se u strukturama podataka nalazi značajno manje elemenata nego u prethodnim primerima (u slučaju kada je univerzum veličine 131072 je broj elemenata sa 120000 smanjen na 10000). Pokazuje se da to ne pravi skoro nikakvu razliku u vremenima izvršavanja. Naime, iako je promenjeno vreme za većinu struktura dijagram izgleda gotovo identično kao u prethodnim slučajevima. Za očekivati je bilo da se AVL stablo dosta približi vEB stablu po vremenu izvršavanja, ali izgleda da već na 10^4 elemenata možemo osetiti razliku između ove dve strukture po vremenu izvršavanja.



Slika 5.4: Dijagram sa manje elemenata u strukturi

Glava 6

Zaključak

U ovom radu je proučavan problem određivanja prethodnika datog broja korišćenjem tri različite strukture podataka: vEB stabla, x- i y-brzog prefiksnog stabla. Svaku od ovih struktura smo detaljno opisali i istakli njihove prednosti i nedostatke. Iako možda ne spadaju u najnovija rešenja, razmatrane strukture podataka su imale veliki uticaj na dalji razvoj algoritama. Naime ideje koje se javljaju u ovim strukturama su poslužile kao inspiracija za razvoj mnogih drugih struktura podataka i algoritama. Na primer, vEB stablo je jedna od prvih struktura koja koristi ograničavanje domena kako bi se ubrzalo izvršavanje [9].

U radu je urađeno eksperimentalno poređenje ovih struktura podataka na primerima veličine do 10^6 sa strukturom AVL stabla. Videli smo da x- i y-brza prefiksna stabla nisu najbolji izbor za ulaze ovih dimenzija, dok je vEB stablo bilo efikasno u rešavanju razmatranih primera. Naravno, iako se na razmatranim primerima AVL stablo pokazalo kao efikasnija struktura od x i y-brzih prefiksnih stabala, to ne govori da su ove strukture podataka loša rešenja. Strukture podataka obrađene ovim radom su prvenstveno namenjene za ulaze ogromnih dimenzija. Na primer, jedna od primena vEB stabla je u rutiranju paketa na mreži. Kako se rutiranje paketa na mreži obavlja preko IP adresa, broj elemenata u problemu rutiranja paketa je oko $4 \cdot 10^9$ i pritom znamo da standard IPv4 nije dovoljan i da se verovatno uskoro u potpunosti prelazi na IPv6, gde je broj elemenata mnogo veći. Sa druge strane y-brza prefiksna stabla se koriste u bazama podataka koje rade sa vremenskim podacima, na primer prilikom očitavanja senzora. Jasno je da je tu reč o jako velikim brojevima jer su moderni senzori u stanju da zabeleže ogroman broj podataka za kratko vreme. Pomenimo i to da su x-brza prefiksna stabla skoro svuda zamenjena y-brzim prefiksnim stablima, međutim, da bismo razumeli i implementirali y-brzo prefik-

sno stablo, bilo nam je potrebno, a i korisno da dobro izučimo strukturu x-brzog prefiksnog stabla.

Kao dalji rad bi se moglo uzeti u razmatranje drvo fuzije [8] (eng. fusion tree) koje ima složenost $O(\log_w n)$ operacija upisivanja, pretrage, brisanja, prethodnika i sledbenika elemenata. Ova struktura podataka daje dobre rezultate kada je w veliko, odnosno kada je broj bitova potrebnih za zapis najvećeg broja koji možemo zapisati u strukturi podataka velik. Drugim rečima, što je skup veći, efikasnost ove strukture se povećava. Takođe, interesantno bi bilo testirati ove strukture na ulazima još većih dimenzija.

Bibliografija

[1] M. AdelsonVelskii. An algorithm for the organization of information. 1963.

[2] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):135–141, 1998.

[3] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeffrey Scott Vitter. On sorting strings in external memory (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 540–548, New York, NY, USA, 1997. Association for Computing Machinery.

[4] Philip Bille, Inge Li Gortz, and Frederik Rye Skjoldjensen. Deterministic Indexing for Packed Strings. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:11, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Rene De La Briandais. File searching using variable length keys. In *IRE-AIEE-ACM Computer Conference*, 1959.

[6] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. *SIGCOMM Comput. Commun. Rev.*, 27(4):3–14, oct 1997.

[7] Patrick Dinklage, Johannes Fischer, and Alexander Herlez. Engineering predecessor data structures for dynamic integer sets, 2021.

[8] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the Twenty-Second Annual ACM*

Symposium on Theory of Computing, STOC '90, page 1–7, New York, NY, USA, 1990. Association for Computing Machinery.

[9] Mihai Patrascu. Lower bound techniques for data structures. 2008.

[10] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[11] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space (n). *Information Processing Letters*, 17(2):81–84, 1983.

Biografija autora

Milan Kocić rođen je 13. oktobra 1998. godine u Beogradu. Osnovnu školu i prirodno-matematički smer gimnazije je završio u Vlasotincu. Osnovne studije na smeru Računarstvo i informatika na Matematičkom fakultetu Univerziteta u Beogradu upisao je 2017. godine, a završio 2021. godine sa prosečnom ocenom 8.85. Nakon toga je upisao master studije na istom fakultetu.

Od oktobra 2021. godine zapošljen je kao saradnik u nastavi na Matematičkom fakultetu Univerziteta u Beogradu.