UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Mirjana Jočović

PERZISTENTNE STRUKTURE PODATAKA

master rad

Beograd, 2026.

Mentor:

prof. dr Vesna MARINKOVIĆ, vanredni profesor,
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Filip MARIĆ, redovni profesor,
Univerzitet u Beogradu, Matematički fakultet

Strahinja STANOJEVIĆ, asistent,
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Sadržaj

Glava 1

Uvod

Savremeni softverski sistemi sve češće zahtevaju da se nad podacima ne posmatra samo trenutno stanje, već i istorija promena koje su do tog stanja dovele. Klasične, odnosno efemerne strukture podataka, zasnivaju se na destruktivnom modelu izmene: operacija ažuriranja neposredno menja postojeću strukturu, prepisuje vrednosti ili preusmerava pokazivače, čime prethodno stanje prestaje da bude dostupno kao zasebna verzija. Takav pristup je jednostavan i memorijski efikasan kada je potrebno čuvati samo poslednje stanje podataka, ali postaje ograničavajući u sistemima u kojima je potrebno vraćanje na prethodno stanje, analiza istorije, paralelno poređenje više verzija ili implementacija mehanizama kao što su *Undo* i *Redo*.

Perzistentne strukture podataka nastaju kao odgovor na ovo ograničenje. Njihova osnovna ideja jeste da operacija izmene ne uništava prethodnu verziju strukture, već proizvodi novu verziju, dok ranije verzije ostaju dostupne za čitanje i dalju analizu. Ovakav model posebno je značajan u funkcionalnom programiranju, gde se podaci posmatraju kao nepromenljive vrednosti, pa se novo stanje formira konstruisanjem novih delova strukture umesto destruktivnom izmenom postojećih. Sa teorijske strane, perzistentnost je detaljno razmatrana kroz opšte metode transformacije klasičnih pokazivačkih struktura u strukture koje čuvaju više verzija, uključujući tehnike koje omogućavaju efikasno deljenje memorije između stare i nove verzije [5, 3].

Predmet ovog rada jeste praktična implementacija perzistentnog binarnog stabla pretrage u programskom jeziku Go. Binarno stablo pretrage izabrano je zato što predstavlja jednu od osnovnih struktura za organizaciju uređenih podataka, dok istovremeno jasno pokazuje izazove koji nastaju pri očuvanju istorije promena. U klasičnom stablu operacije umetanja i brisanja menjaju pokazivače na putanji od korena do mesta izmene, pa se prethodno stanje gubi. U perzistentnoj verziji cilj je

da se takva promena izvrši bez narušavanja ranijih verzija stabla.

Osnovna tehnika korišćena u radu jeste *kopiranje putanje* (engl. *path copying*). Umesto kopiranja celog stabla nakon svake izmene, kopiraju se samo čvorovi koji se nalaze na putanji od korena do izmenjenog mesta, dok se sva ostala podstabla dele između verzija. Na taj način se postiže kompromis između očuvanja istorije i kontrole utroška memorije. Svaka nova verzija stabla dobija sopstveni koren, ali značajan deo strukture ostaje zajednički sa prethodnim verzijama. Takav pristup omogućava da se istorija stanja čuva efikasnije nego kod naivnog kopiranja cele strukture.

Pored same implementacije perzistentnog stabla, u radu je razvijen i poseban menadžer istorije, čija je uloga da upravlja verzijama strukture i obezbedi trenutni pristup prethodnim i narednim stanjima. Na taj način implementiran je *Undo/Redo* mehanizam kod koga se prelazak između već kreiranih verzija izvršava u vremenu $O(1)$, jer se ne rekonstruiše sadržaj stabla, već se samo menja referenca na odgovarajući koren verzije. Time se jasno pokazuje praktična vrednost perzistentnosti: dodatno zauzeće resursa nastaje tokom ažuriranja i čuvanja verzija, ali se zauzvrat dobija veoma efikasan pristup istoriji stanja.

Cilj rada nije samo implementacija perzistentnog binarnog stabla pretrage, već i njena empirijska evaluacija. Zbog toga su performanse perzistentnog binarnog stabla pretrage upoređene sa klasičnim efemernim stablom. Posebno su analizirani vreme izvršavanja operacija umetanja i pristupa istoriji, kao i memorijsko zauzeće koje nastaje zbog čuvanja dodatnih verzija. Merenja su sprovedena za različite vrednosti parametra N, uključujući i slučajeve do milion elemenata. Dobijeni rezultati potvrđuju očekivani kompromis pri dizajnu: perzistentno stablo troši osetno više memorije i nešto je sporije pri upisu, ali omogućava direktan $O(1)$ pristup istorijskim stanjima preko sačuvanih korenova verzija.

Rad je organizovan u šest poglavlja. Poglavlje 2 daje teorijske osnove struktura podataka, objašnjavajući ograničenja klasičnih efemernih struktura i uvodeći osnovni koncept perzistentnosti u računarstvu, uz analizu njenih primena i memorijske efikasnosti. Poglavlje 3 bavi se tehnikama implementacije perzistentnosti, gde se detaljno analiziraju metoda „debelih" čvorova, metoda kopiranja i metoda razdvajanja čvorova, uz uporednu analizu savremenih pristupa. Poglavlje 4 predstavlja konkretne perzistentne strukture podataka, sa posebnim osvrtom na perzistentni stek, red, binarno stablo pretrage i segmentno stablo. Poglavlje 5 predstavlja centralni deo rada u kome je prikazana praktična implementacija perzistentnog binarnog stabla pretrage, arhitektura *Undo/Redo* sistema i empirijska analiza performansi. Na kra-

ju, Poglavlje 6 iznosi zaključna razmatranja, sumira dobijene rezultate i predlaže pravce za dalji razvoj i istraživanje.

Glava 2

Teorijske osnove struktura podataka

2.1 Klasične (efemerne) strukture podataka i njihova ograničenja

Klasične, odnosno efemerne strukture podataka, predstavljaju strukture kod kojih u svakom trenutku postoji samo njihova najnovija verzija. Prilikom izmene takvih struktura, prethodno stanje se efektivno uništava i zamenjuje novim stanjem, tako da za dalju upotrebu ostaje dostupna isključivo nova verzija. Zbog toga se efemerne strukture mogu posmatrati kao model podataka u kome istorija promena nije deo same strukture, već se svaka operacija izvršava isključivo nad trenutno važećim stanjem [3, 5].

Ovakav način rada posebno je vidljiv kod povezanih struktura podataka zasnovanih na čvorovima i pokazivačima. Povezana struktura podataka se u opštem smislu definiše kao konačna kolekcija čvorova, pri čemu svaki čvor sadrži fiksan broj imenovanih polja. Ta polja mogu biti polja sa vrednošću, koja čuvaju pojedinačni podatak određenog tipa, ili pokazivačka polja, koja sadrže pokazivač na drugi čvor ili specijalnu vrednost `null`. Pristup ovakvim strukturama obezbeđuje se preko fiksnog broja imenovanih pristupnih pokazivača, dok se same strukture posmatraju kao usmereni grafovi čiji čvorovi imaju konstantan izlazni stepen. U takvom modelu, organizacija podataka ne zavisi samo od vrednosti u čvorovima, već primarno od trenutno važeće konfiguracije pokazivača koji određuju međusobne veze.

Primer binarnog stabla pretrage jasno pokazuje posledice efemernog ažuriranja. Binarno stablo pretrage može se predstaviti kao povezana struktura u kojoj svaki čvor sadrži jedno polje za element i dva pokazivačka polja, levo i desno, koja upućuju

na levog i desnog sina, dok je koren stabla jedini ulazni čvor strukture. Operacija pretrage u takvom stablu gradi putanju od korena, prateći pokazivače sve dok se traženi element ne pronađe ili dok se ne dođe do `null` pokazivača. Kada se izvršava umetanje, najpre se pronalazi mesto na kome pretraga završava nedostajućim čvorom, a zatim se on zamenjuje novim čvorom koji sadrži novi element. Tada se pokazivač, koji je prethodno imao vrednost `null`, menja tako da upućuje na novokreirani čvor. U efemernom modelu ta promena ne ostavlja prethodnu konfiguraciju dostupnom, te staro stanje više ne postoji kao posebna verzija strukture [1].

Sličan princip važi i kod operacije brisanja. Pri brisanju se najpre pronalazi čvor koji sadrži traženi element, a ukoliko on ima dva deteta, zamenjuje se odgovarajućim prethodnikom u simetričnom poretku, čime se problem svodi na uklanjanje čvora sa najviše jednim detetom. Završni korak sastoji se u uklanjanju tog čvora i preusmeravanju pokazivača njegovog roditelja na jedino dete obrisanog čvora ili na `null`. Ako neki čvor više nije označen nijednim pokazivačem u strukturi, on nestaje iz nje, pri čemu se ne zahteva eksplicitna dealokacija memorije. Time prethodna organizacija stabla postaje nepovratno izgubljena, jer više ne postoji pristupni put koji bi omogućio rekonstrukciju ranije verzije.

Glavno ograničenje efemernih struktura proizlazi iz činjenice da one podržavaju samo nizove operacija u kojima se svaka naredna operacija primenjuje isključivo na najskoriju verziju podataka. Zbog toga je efemerni model nedovoljan u računarskim problemima u kojima je neophodno čuvati i koristiti više istorijskih stanja, kao što je to slučaj u računarskoj geometriji, sistemima za uređivanje teksta i implementaciji programskih jezika vrlo visokog nivoa. Kao odgovor na ova ograničenja razvijene su tehnike kojima je moguće postići perzistentnost, sa ciljem da se kod povezanih struktura sačuvaju prethodne verzije i omogući njihovo kasnije pretraživanje. Za razliku od efemernog pristupa, perzistentne strukture omogućavaju pristup višestrukim verzijama, pri čemu delimična perzistentnost dozvoljava pristup svim verzijama uz ažuriranje samo najnovije, dok potpuna perzistentnost dozvoljava i pristup i ažuriranje svake postojeće verzije [3, 5].

2.2 Koncept perzistentnosti u računarstvu

Koncept perzistentnosti menja tradicionalni pristup tako što se struktura više ne posmatra isključivo kao trenutno važeće stanje elemenata i njihovih veza, već kao dinamički sistem verzija koje nastaju sukcesivnim operacijama i ostaju dostupne za

dalju analizu.

U literaturi se najčešće razlikuju dva osnovna oblika perzistentnosti: *parcijalna* (engl. *partial persistence*) i *potpuna perzistentnost* (engl. *full persistence*). Parcijalna perzistentnost označava model u kome su sve prethodne verzije dostupne za operacije pristupa (čitanja i pretraživanja), ali se operacije ažuriranja mogu primenjivati samo na najnoviju verziju strukture. Drugim rečima, istorija strukture može se pregledati, ali se ne može menjati iz neke ranije tačke. U parcijalno perzistentnom modelu svako novo ažuriranje primenjuje se isključivo na neposredno prethodnu, odnosno najnoviju verziju.

Potpuna perzistentnost predstavlja opštiji model, jer dozvoljava da se svaka postojeća verzija ne samo čita, već i ažurira. Kada se ažuriranje primeni na neku raniju verziju, ta verzija se ne menja, već se kao rezultat operacije stvara nova, zasebna verzija strukture. Time verzije više ne moraju činiti samo linearan niz, već mogu formirati razgranatu istoriju: iz jedne ranije verzije moguće je dobiti novu granu razvoja strukture. Zbog toga je navigacija kroz verzije kod potpuno perzistentnih struktura složenija nego kod efemernih struktura, jer trenutna verzija nije nužno i najnovija verzija, već se verzija nad kojom se operacija izvršava mora eksplicitno odrediti [3].

Pored ova dva osnovna oblika, u literaturi se pominju i složeniji oblici perzistentnosti, kao što je *konfluentna perzistentnost* (engl. *confluent persistence*). Ona se nadovezuje na potpunu perzistentnost time što omogućava da se dve postojeće verzije strukture spoje jednom operacijom ažuriranja. Takav model predstavlja dodatno proširenje pojma verzionisanja, jer istorija strukture više nije samo razgranata, već dopušta i ukrštanje različitih grana.

Označavanje verzija predstavlja važan deo perzistentnog modela. U osnovnom formalnom opisu, početna struktura označava se kao verzija 0, dok se verzija i dobija nakon i-te operacije ažuriranja. Kod tehnika zasnovanih na čvorovima, vrednosti polja mogu imati i vremenske oznake (oznake verzija), koje pokazuju u kojoj verziji je određena vrednost nastala ili promenjena. Na taj način, pristup određenoj verziji ne zavisi samo od trenutnih pokazivača, kao kod efemernih struktura, već od toga koje su vrednosti polja važeće za traženu verziju. Perzistentna struktura zato mora omogućiti razlikovanje verzija i pronalaženje odgovarajućeg stanja podataka za svaku od njih, čime se osnovni model pristupa bitno razlikuje od klasičnog rada nad jednom trenutno važećom strukturom [3, 5].

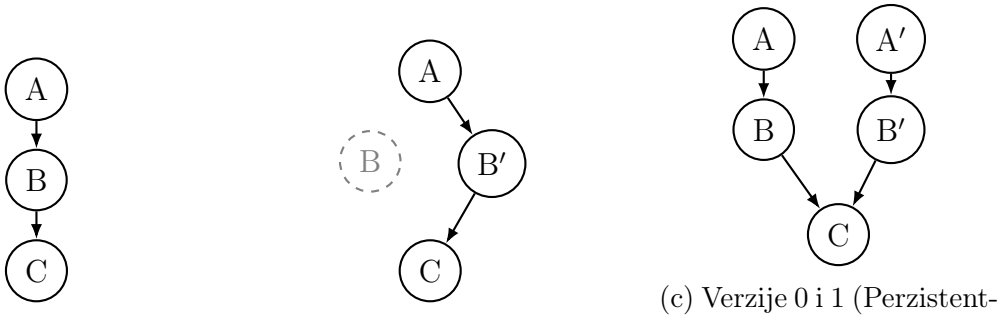
2.3 Ilustrativni primeri i vizuelizacija koncepta

Razumevanje razlike između efemernog i perzistentnog modela ažuriranja znatno je jednostavnije ukoliko se promene ne posmatraju samo apstraktno, već i kroz način na koji se menjaju veze između čvorova. Budući da su povezane strukture podataka određene ne samo vrednostima smeštenim u čvorovima, već i pokazivačima koji te čvorove povezuju, vizuelizacija toka promena omogućava jasnije uočavanje trenutka u kome se prethodno stanje gubi ili, u slučaju perzistentnosti, ostaje sačuvano kao posebna verzija strukture.

Kao ilustrativni primer može se posmatrati jednostavna povezana struktura sastavljena od tri čvora: A, B i C. U početnom stanju, koje se označava kao Verzija 0, čvor A sadrži pokazivač na čvor B, dok čvor B sadrži pokazivač na čvor C. Na taj način formira se linearna putanja $A \to B \to C$. Ova verzija predstavlja početno stanje strukture, pre izvršavanja bilo kakvog ažuriranja, i može se prikazati kao osnovni referentni oblik strukture, kao što je prikazano na Slici 2.1a. U ovom trenutku postoji samo jedan ulazni pokazivač, koji vodi ka početku strukture, odnosno ka čvoru A.

U efemernom scenariju, neka operacija ažuriranja menja čvor B tako da se dobija nova vrednost, označena kao B'. Ako se ova promena realizuje destruktivno, struktura se menja neposredno: pokazivač iz čvora A sada vodi ka novom ili izmenjenom čvoru B', a dalje se nastavlja veza ka čvoru C. Posle ažuriranja, dostupno stanje strukture jeste $A \to B' \to C$, koje se označava kao Verzija 1, kao što je prikazano na Slici 2.1b. Međutim, prethodno stanje $A \to B \to C$ više nije dostupno kroz samu strukturu, jer ne postoji sačuvan podatak da je pokazivač iz čvora A ranije vodio ka čvoru B. Ukoliko je B prepisan, njegova ranija vrednost je izgubljena; ukoliko je fizički ostao u memoriji, ali nijedan pokazivač više ne vodi do njega, on je izolovan sa stanovišta strukture. U oba slučaja, Verziji 0 se više ne može pristupiti kao validnom stanju strukture.

U perzistentnom scenariju isto ažuriranje mora biti izvedeno tako da se nova verzija dobije bez uništavanja stare. Umesto da se postojeća veza iz čvora A destruktivno promeni, kreira se nova verzija relevantnog dela strukture. Novi čvor B' predstavlja izmenjeno stanje čvora B, dok se stari čvor B zadržava radi očuvanja Verzije 0. Da bi nova verzija imala sopstveni tok pokazivača, kreira se i novi čvor A', čiji pokazivač vodi ka B'. Time se dobijaju dve dostupne putanje: stara putanja $A \to B \to C$, koja odgovara Verziji 0, i nova putanja $A' \to B' \to C$, koja odgovara



(a) Verzija 0 (Početno) (b) Verzija 1 (Efemerno) (c) Verzije 0 i 1 (Perzistentno)

Slika 2.1: Uporedni prikaz ažuriranja povezanih struktura. Čvor C je deljen u perzistentnom modelu.

Verziji 1, kao što je prikazano na Slici 2.1c. Razlika između verzija ne mora podrazumevati kopiranje cele strukture, jer čvor C nije izmenjen i zato može biti zajednički za obe verzije.

Ovaj primer pokazuje da se suština perzistentnosti oslanja na kontrolisano upravljanje pokazivačima i efikasno deljenje memorije. Efemerni model zadržava samo poslednje stanje, dok perzistentni model mora obezbediti da svaka verzija ima očuvan ulaz u odgovarajuću konfiguraciju čvorova. Istovremeno, prostorna efikasnost se postiže time što se ne kopiraju svi delovi strukture, već samo oni koji su neophodni da bi se nova verzija razlikovala od prethodne. Delovi koji nisu promenjeni, kao što je u ovom primeru čvor C, mogu se deliti između više verzija, čime se izbegava nepotrebno dupliranje podataka i priprema osnova za tehnike implementacije perzistentnosti koje se zasnivaju na pažljivom čuvanju istorije promena.

2.4 Primena i značaj u savremenim sistemima

Perzistentne strukture podataka imaju značajnu ulogu u savremenim računarskim sistemima zato što omogućavaju da se promene nad podacima ne posmatraju kao nepovratno prepisivanje prethodnog stanja, već kao formiranje nove verzije uz očuvanje ranijih stanja. Takav model povećava sigurnost rada sa podacima u situacijama u kojima je potrebno vratiti se na ranije stanje, proveriti istoriju izmena ili izvršiti analizu nad više vremenskih preseka iste strukture, čime se uvodi eksplicitna podrška za praćenje promena kroz vreme.

Poseban značaj perzistentnosti može se uočiti u funkcionalnom programiranju. U funkcionalnom modelu programiranja podaci se, po pravilu, ne menjaju direktno u mestu, već se prilikom transformacije formiraju nove vrednosti. Takav princip

nepromenljivosti (engl. *immutability*) prirodno vodi ka perzistentnim strukturama podataka, jer se prethodno stanje ne sme uništiti samom operacijom ažuriranja. Ako se struktura ne menja destruktivno, već se iz postojeće strukture izvodi nova, ranija verzija ostaje dostupna i nakon nastanka nove verzije. Zbog toga se može zaključiti da su nepromenljive strukture stroži oblik u odnosu na perzistentne, jer ne dozvoljavaju izmenu postojećeg stanja, dok perzistentnost zahteva da prethodno stanje ostane dostupno nakon promene.

Druga važna oblast primene odnosi se na mehanizme za poništavanje i ponovno izvršavanje operacija, odnosno *Undo* i *Redo* funkcionalnosti. Aplikacije za uređivanje teksta i datoteka predstavljaju prirodan primer sistema u kojima je potrebno čuvati istoriju stanja, jer korisnik često mora da se vrati na raniju verziju dokumenta ili da ponovo primeni prethodno poništenu izmenu. Umesto da se svaka izmena posmatra kao konačna zamena prethodnog sadržaja, perzistentni model omogućava da se niz stanja dokumenta organizuje kao skup dostupnih verzija. Na taj način operacija poništavanja može biti shvaćena kao prelazak na prethodnu verziju, dok ponovno izvršavanje odgovara povratku na kasnije nastalu verziju. Uređivanje teksta i datoteka navodi se kao jedna od oblasti u kojima je održavanje višestrukih verzija strukture podataka od posebnog značaja [5].

Sistemi za kontrolu verzija i distribuirano računarstvo takođe se mogu posmatrati kroz širi značaj perzistentnosti. U sistemima koji čuvaju istoriju promena neophodno je da ranija stanja podataka ostanu dostupna za pregled, poređenje i rekonstrukciju. Perzistentni model pruža konceptualnu osnovu za takav način rada, jer svaka izmena proizvodi novu verziju, dok prethodne verzije ostaju očuvane. U distribuiranim i konkurentnim okruženjima dodatna prednost proizlazi iz činjenice da se stara stanja ne moraju prepisivati. Kada različiti procesi ili niti mogu da čitaju ranije verzije bez destruktivne izmene istih podataka, smanjuje se potreba za agresivnom sinhronizacijom nad deljenim stanjem. Ova primena predstavlja proširenje osnovne ideje perzistentnosti: očuvanje prethodnih verzija omogućava sigurniji i pregledniji rad u sistemima u kojima se podaci razvijaju kroz više nezavisnih tokova izvršavanja.

Računarska geometrija predstavlja jedno od najistaknutijih područja u kome se javlja potreba za perzistentnim strukturama podataka. U geometrijskim problemima često je potrebno pretraživati strukture koje opisuju različita stanja prostora ili različite faze algoritma. Umesto da se nakon svake promene izgubi ranije stanje, perzistentna struktura omogućava da se više verzija iste strukture održava istovremeno i da se nad njima kasnije izvršavaju operacije pristupa. Posebno je značajan

primer perzistentnih binarnih stabala pretrage, koja su korišćena u rešavanju problema lociranja tačke u ravni, a zatim poslužila kao osnova za razvoj opštijih tehnika perzistentnosti kod povezanih struktura podataka.

Značaj perzistentnih struktura zato ne leži samo u teorijskoj mogućnosti čuvanja ranijih verzija, već i u praktičnoj sposobnosti da se nad podacima obezbedi istorijski kontinuitet. U sistemima koji zahtevaju pouzdano praćenje promena, vraćanje na prethodna stanja, paralelno čitanje ili analizu više verzija, perzistentnost pruža model koji prevazilazi ograničenja klasičnih efemernih struktura. Time se omogućava da struktura podataka ne bude samo trenutni zapis podataka, već i organizovana reprezentacija njihovog razvoja kroz vreme [3, 5].

2.5 Usklađivanje vremenske i memorijske efikasnosti

Uvođenje perzistentnosti u strukture podataka ne predstavlja samo pitanje očuvanja prethodnih verzija, već i pitanje efikasnog upravljanja vremenom izvršavanja i potrošnjom memorije. Kada struktura mora da podrži više verzija istovremeno, svaka operacija ažuriranja potencijalno proizvodi novo stanje koje treba sačuvati, dok operacije pristupa moraju moći da pronađu odgovarajuću verziju bez značajnog usporavanja. Zbog toga se u osnovi perzistentnih struktura nalazi kompromis između vremenske složenosti i memorijske potrošnje: što se više verzija čuva eksplicitno, pristup starim verzijama može biti jednostavniji, ali je zauzeće memorije veće; što se manje podataka čuva, rekonstrukcija ranijih verzija postaje vremenski zahtevnija.

Najjednostavniji, ali i najneefikasniji način da se obezbedi perzistentnost jeste eksplicitno čuvanje svake verzije strukture. U tom pristupu, posle svake operacije ažuriranja kopira se celokupna efemerna struktura, tako da svaka verzija postoji kao zasebna kopija. Prednost ovakvog modela jeste konceptualna jednostavnost: pristup određenoj verziji svodi se na izbor odgovarajuće kopije strukture. Međutim, takav nivo resursne zahtevnosti čini ga nepraktičnim za veće strukture. Ako trenutna verzija sadrži n čvorova, kopiranje cele strukture nakon svake izmene zahteva $\Omega(n)$ vremena i $\Omega(n)$ dodatnog prostora po ažuriranju. Takav pristup je posebno nepovoljan zato što i mala lokalna izmena, na primer promena jednog pokazivača ili jednog polja u čvoru, dovodi do kopiranja cele strukture. Time se gubi osnovna prednost povezanih struktura podataka, kod kojih se efemerna ažuriranja često mogu izraziti malim brojem lokalnih promena.

Alternativni naivni pristup pokušava da smanji memorijsku potrošnju tako što se ne čuvaju pojedinačne verzije strukture, već samo celokupan dnevnik operacija ažuriranja. U tom slučaju zauzeće prostora može biti znatno manje, jer se čuva niz promena umesto kopija svih stanja. Ukoliko se svaka operacija ažuriranja može zapamtiti u konstantnom prostoru, ukupna potrošnja memorije za m ažuriranja iznosi $O(m)$. Međutim, problem ovakve uštede javlja se pri pristupu starim verzijama. Da bi se dobila verzija i, struktura se mora rekonstruisati od početnog stanja ponovnim izvršavanjem odgovarajućeg prefiksa operacija, pa pristup verziji i može zahtevati $\Omega(i)$ vremena, čak i kada je pojedinačno ažuriranje u efemernom modelu $O(1)$. Ovaj pristup zato prebacuje teret iz prostora u vreme: memorija se štedi, ali se efikasnost pristupa ranijim verzijama značajno narušava.

Između ova dva ekstrema moguće je konstruisati hibridne metode, u kojima se čuvaju dnevnik promena i povremene potpune kopije verzija. Takav pristup smanjuje potrebu da se svaka verzija rekonstruiše od početka, ali i dalje uvodi kompromis koji zavisi od učestalosti sačuvanih kopija i od složenosti efemernih operacija. Ako se kopije čuvaju retko, smanjuje se memorijska potrošnja, ali se produžava rekonstrukcija verzija između dve sačuvane kopije; ako se kopije čuvaju često, pristup je brži, ali raste zauzeće prostora. Zbog toga hibridni modeli ne rešavaju suštinski problem, već samo pomeraju granicu usklađivanja između vremena i memorije.

Glavni cilj naprednih tehnika perzistentnosti jeste da se izbegnu upravo ovi ekstremi. Umesto kopiranja cele strukture ili rekonstrukcije verzije iz dnevnika promena, efikasne metode nastoje da sačuvaju samo one informacije koje su neophodne da bi se prethodne verzije razlikovale od nove. U tom smislu, tehnike kao što su metoda debelih čvorova, metoda kopiranja i metoda razdvajanja čvorova pokušavaju da očuvaju lokalnost promena karakterističnu za efemerne strukture, uz istovremeno omogućavanje pristupa ranijim verzijama. Posebno značajan cilj jeste postizanje konstantnog dodatnog utroška resursa od $O(1)$ po koraku ažuriranja, bilo u najgorem slučaju ili amortizovano, kako u pogledu vremena tako i u pogledu memorije. Time se perzistentnost ne posmatra samo kao teorijsko proširenje modela struktura podataka, već kao praktično primenljiv mehanizam za rad sa višestrukim verzijama u realnim sistemima [3, 5].

Glava 3

Tehnike implementacije perzistentnosti

3.1 Metoda „debelih" čvorova

Metoda „debelih" čvorova (engl. *Fat Node Method*) predstavlja jedan od osnovnih pristupa za transformaciju efemernih povezanih struktura podataka u perzistentne strukture. Njena osnovna ideja zasniva se na tome da se promene nad poljima čvorova ne izvršavaju destruktivnim brisanjem prethodnih vrednosti, već se sve izmene beleže unutar samih čvorova. Umesto da čvor sadrži samo trenutno važeće vrednosti svojih polja, dozvoljava se da čvor postane proizvoljno „debeo", odnosno da čuva proizvoljan broj vrednosti za svako svoje polje. Takav čvor zadržava ista polja za vrednost i pokazivače kao odgovarajući efemerni čvor, ali se proširuje dodatnim prostorom u kome se skladište vrednosti polja koje su nastajale kroz različite verzije cele strukture [3].

Važno je naglasiti da se oznake verzija u ovoj metodi ne odnose samo na izolovane verzije pojedinačnih čvorova, već pre svega na verzije strukture podataka kao celine. Ako se kao primer posmatra binarno stablo pretrage, svaka operacija ažuriranja, kao što je umetanje ili brisanje čvora, proizvodi novu verziju stabla. Pojedinačni čvorovi zatim u sebi čuvaju lokalne zapise o tome kako su se njihova polja menjala kroz te verzije stabla. Prema tome, verzionisane vrednosti u čvorovima predstavljaju mehanizam pomoću koga se može rekonstruisati stanje cele strukture u izabranoj verziji, a ne samo istorija jednog čvora posmatranog izolovano.

Mehanizam verzionisanja zasniva se na povezivanju svake dodatne vrednosti sa metapodacima koji određuju njeno važenje. Svaka dodatna vrednost polja sadrži ime

polja na koje se odnosi, samu vrednost i oznaku verzije, odnosno vremenski pečat (engl. *version stamp*) koji pokazuje u kojoj verziji strukture je to polje promenjeno. Pored toga, i sam čvor može imati sopstvenu oznaku verzije, kojom se označava verzija strukture u kojoj je taj čvor nastao. Na taj način se istorija promena ne čuva kao niz potpunih kopija strukture, već kao skup lokalnih zapisa unutar čvorova nad čijim poljima su izmene izvršene [3].

Proces ažuriranja u metodi „debelih" čvorova može se posmatrati kao simulacija efemernog ažuriranja bez gubitka prethodnih verzija strukture. Kada efemerni korak ažuriranja kreira novi čvor, na primer pri umetanju novog elementa u stablo, u perzistentnoj strukturi se kreira odgovarajući „debeli" čvor sa oznakom verzije tog ažuriranja i sa početnim vrednostima originalnih i pokazivačkih polja. Ta oznaka znači da čvor postoji počev od verzije strukture u kojoj je kreiran, dok u ranijim verzijama nije deo strukture. Kada se menja vrednost nekog polja u postojećem čvoru, stara vrednost se ne uklanja, već se u isti čvor dodaje nova vrednost, zajedno sa imenom promenjenog polja i oznakom verzije strukture u kojoj je promena nastala. Za jedno polje u jednom čvoru čuva se najviše jedna vrednost po verziji; ukoliko već postoji vrednost istog polja sa istom oznakom verzije, ona se zamenjuje novom vrednošću [3].

Navigacija kroz perzistentnu strukturu zasniva se na pristupu izabranoj verziji cele strukture i na izboru vrednosti polja koje su u toj verziji važeće. Kada se, na primer, želi pretražiti verzija i nekog stabla, najpre se mora odrediti ulazni pokazivač koji odgovara toj verziji, kao što je koren stabla u verziji i. Zbog toga se za pristupne pokazivače koriste pomoćni nizovi, u kojima se nakon operacije ažuriranja i čuvaju trenutne vrednosti pristupnih pokazivača na poziciji i. Nakon što je određen odgovarajući ulaz u strukturu, dalja navigacija se vrši kroz čvorove tako što se za svako polje bira vrednost koja je bila važeća u traženoj verziji. Kada se simulira pristup polju f u verziji i, u odgovarajućem „debelom" čvoru ne bira se nužno najnovija fizički upisana vrednost tog polja, već ona vrednost čije je ime polja jednako f i čiji je vremenski pečat najveći među svim pečatima koji nisu veći od i. Drugim rečima, pristup verziji i zahteva da se za svako posećeno polje pronađe poslednja promena koja je nastala najkasnije u toj verziji strukture [3].

Prednost metode „debelih" čvorova ogleda se u tome što se perzistentnost postiže lokalnim beleženjem promena, bez potrebe za kopiranjem cele strukture pri svakom ažuriranju. Zbog toga zauzeće prostora po koraku ažuriranja iznosi $O(1)$ u najgorem slučaju. Međutim, osnovno ograničenje ove metode proizlazi iz činjenice

da čvorovi mogu akumulirati proizvoljno mnogo vrednosti istog polja iz različitih verzija strukture. Pri svakom pristupu neophodno je locirati odgovarajuću vrednost na osnovu vremenskog pečata, što narušava konstantno vreme izvršavanja operacija. Ukoliko su verzionisane vrednosti unutar čvora organizovane prema oznakama verzija upotrebom binarnog stabla pretrage, vrednost za traženu verziju pronalazi se logaritamskom pretragom umesto u konstantnom vremenu. Shodno tome, simulacija jednog efemernog koraka pristupa zahteva $O(\log m)$ vremena, gde m predstavlja ukupan broj operacija ažuriranja. Detaljnijom analizom, u kojoj se posmatra isključivo broj izmena nad samim čvorom, vremenska složenost se može izraziti kao $O(\log h)$, pri čemu je h maksimalan broj promena izvršenih nad pojedinačnim efemernim čvorom. Prema tome, metoda „debelih" čvorova ostvaruje visoku prostornu efikasnost, ali uz logaritamsko usporavanje operacija u poređenju sa klasičnim efemernim strukturama [3].

Iako su primeri u nastavku prikazani kroz operacije umetanja u binarno stablo pretrage, isti princip važi i za operacije brisanja. Kod čvora sa dva deteta brisanje se može svesti na zamenu sa prethodnikom u simetričnom poretku, nakon čega se uklanja čvor koji ima najviše jedno dete. Brisanje elementa u binarnom stablu pretrage zato se takođe svodi na konačan niz lokalnih izmena originalnih i pokazivačkih polja: pokazivač koji je ranije vodio ka obrisanom čvoru može biti promenjen tako da vodi ka njegovom detetu ili ka vrednosti `null`, u zavisnosti od položaja čvora koji se briše. U efemernoj strukturi takva promena zamenjuje prethodno stanje, dok se u perzistentnoj strukturi stara vrednost pokazivača mora očuvati kao deo odgovarajuće ranije verzije. Zbog toga metode opisane u nastavku ne zavise od konkretne semantike operacije umetanja ili brisanja, već od načina na koji se pojedinačni koraci ažuriranja odražavaju na polja čvorova [3].

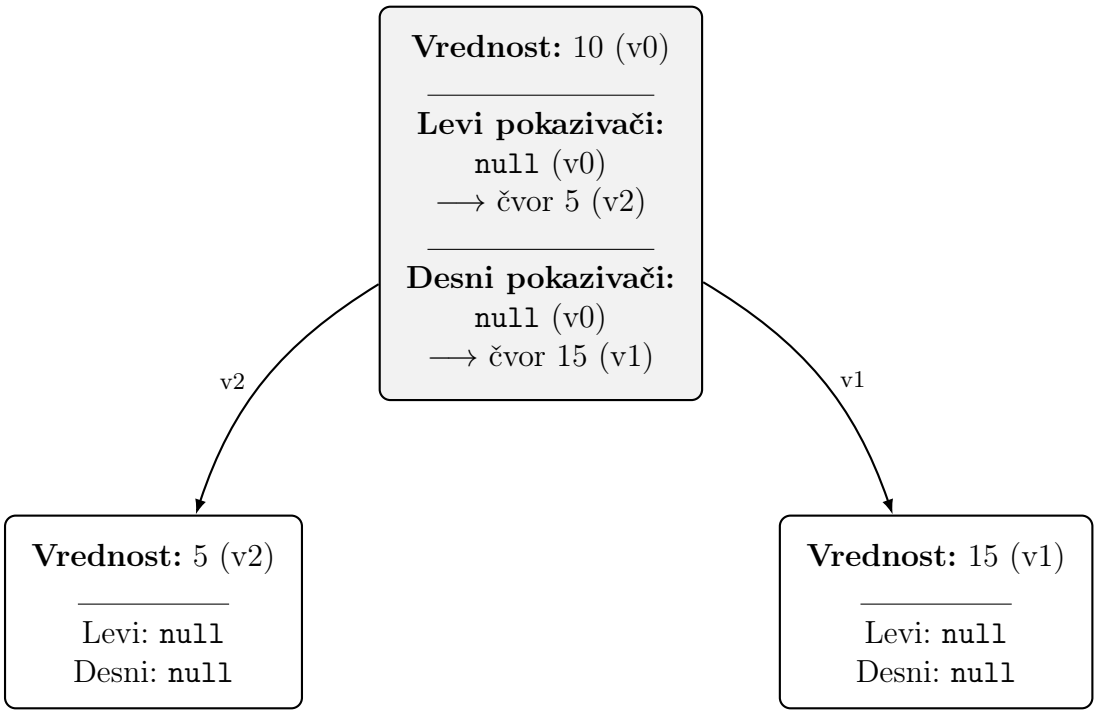
Primer izvršavanja operacija – Metoda debelih čvorova

Radi jasnijeg razumevanja metode „debelih" čvorova, može se posmatrati jednostavan primer perzistentnog binarnog stabla pretrage u kome se promene čuvaju unutar samih čvorova, umesto da se pri svakom ažuriranju kopira cela struktura. Početno stanje, označeno kao Verzija 0, sastoji se samo od jednog korenskog čvora sa vrednošću 10. Budući da u toj verziji ne postoje ni levi ni desni potomak, pokazivačka polja korena imaju vrednost `null`. Ove početne vrednosti takođe predstavljaju deo istorije čvora, jer određuju stanje stabla pre bilo kakvog umetanja.

Prvi korak ažuriranja predstavlja umetanje vrednosti 15. Pošto je 15 veće od

vrednosti korena 10, novi čvor se umeće kao desni potomak korenskog čvora. U efemernom stablu ovo bi značilo neposrednu promenu desnog pokazivača korena sa vrednosti `null` na pokazivač ka novom čvoru. U metodi „debelih" čvorova, međutim, prethodna vrednost se ne briše. Umesto toga, korenski čvor se proširuje dodatnim zapisom za polje desnog pokazivača, pri čemu se kao nova vrednost upisuje pokazivač na čvor 15, a kao vremenski pečat navodi se Verzija 1. Na taj način čvor 10 istovremeno čuva informaciju da je u Verziji 0 njegov desni pokazivač bio `null` i da od Verzije 1 desni pokazivač vodi ka čvoru 15.

Drugi korak ažuriranja predstavlja umetanje vrednosti 5. Pošto je 5 manje od vrednosti korena 10, novi čvor se umeće kao levi potomak korenskog čvora. Kao i u prethodnom koraku, korenski čvor se ne kopira, već se dodatno proširuje novim zapisom. U njegovo polje levog pokazivača dodaje se vrednost koja upućuje na novi čvor 5, zajedno sa vremenskim pečatom Verzije 2. Time korenski čvor postaje „deblji", jer u sebi čuva više verzionisanih vrednosti svojih pokazivačkih polja. Ipak, on ostaje isti fizički čvor u memoriji, dok se razlike između verzija određuju na osnovu vremenskih pečata pridruženih pojedinačnim vrednostima polja.



Slika 3.1: Struktura korenskog „debelog" čvora nakon Verzije 2. Sve izmene se pamte u obliku vremenskih pečata unutar samog čvora.

Posebno je važno razmotriti pristup staroj verziji nakon što je već formirana

Verzija 2. Ako se u tom trenutku zatraži pristup Verziji 1 i pokuša čitanje levog pokazivača korena, algoritam ne sme da koristi fizički najnoviju vrednost tog polja, jer ona pripada Verziji 2. Umesto toga, pretražuju se zapisi za levo pokazivačko polje i bira se ona vrednost čiji je vremenski pečat najveći, ali nije veći od tražene verzije. Pošto zapis za levi pokazivač ka čvoru 5 ima vremenski pečat Verzije 2, on nije važeći u Verziji 1. Zbog toga se kao važeća vrednost za levi pokazivač u Verziji 1 dobija početna vrednost `null`. Unutrašnje stanje korenskog čvora nakon Verzije 2 prikazano je na Slici 3.1 i u Tabeli 3.1.

Tabela 3.1: Unutrašnje stanje „debelog" čvora (korena) nakon Verzije 2

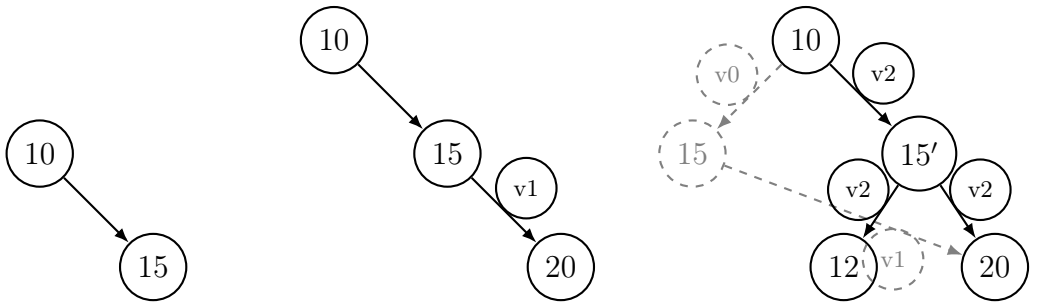
Ime polja	**Vrednost (pokazivač / informacija)**	**Vremenski pečat**
osnovna vrednost	10	Verzija 0
levi pokazivač	`null`	Verzija 0
desni pokazivač	`null`	Verzija 0
desni pokazivač	pokazivač na čvor 15	Verzija 1
levi pokazivač	pokazivač na čvor 5	Verzija 2

Ovaj primer pokazuje osnovni kompromis metode „debelih" čvorova. Prostorna ušteda postiže se time što se korenski čvor ne kopira pri svakom ažuriranju, već se u njega dodaju samo novi verzionisani zapisi. Istovremeno, čitanje postaje složenije nego u efemernom modelu, jer se pri svakom pristupu mora proveriti koji je zapis odgovarajućeg polja važeći za traženu verziju. Prema tome, metoda jasno razdvaja fizičko stanje čvora u memoriji od logičkog stanja stabla u pojedinačnoj verziji.

3.2 Metoda kopiranja čvorova

Metoda kopiranja čvorova (engl. *Node-Copying Method*) predstavlja tehniku za postizanje parcijalne perzistentnosti kod povezanih struktura podataka, razvijenu sa ciljem da se otkloni osnovno ograničenje metode „debelih" čvorova. Kod metode „debelih" čvorova čvorovi mogu sadržati proizvoljno mnogo verzionisanih vrednosti, zbog čega se pri pristupu odgovarajućoj verziji mora pretraživati skup zapisa unutar čvora. Metoda kopiranja čvorova uvodi drugačiji pristup: broj dodatnih polja u svakom perzistentnom čvoru ograničava se konstantom, čime se sprečava neograničen rast čvora i omogućava da vreme pristupa ostane $O(1)$ u najgorem slučaju [3].

U ovoj metodi svaki efemerni čvor može biti predstavljen jednim ili većim brojem perzistentnih čvorova. Skup perzistentnih čvorova koji odgovaraju istom efemernom



(a) Verzija 0 (b) Verzija 1 (c) Verzija 2 (Kopiranje čvora 15)

Slika 3.2: Prikaz metode kopiranja čvorova kroz verzije. U Verziji 2 (c), stari čvor 15 i nova kopija 15′ dele zajednički čvor 20.

čvoru naziva se porodicom, a članovi porodice povezani su pokazivačima na kopiju, u rastućem redosledu oznaka verzija. Najnoviji član porodice predstavlja živi čvor, dok se stariji članovi smatraju mrtvim, jer se u parcijalnoj perzistentnosti ažuriranja izvršavaju samo nad najnovijom verzijom strukture. Struktura perzistentnog čvora sastoji se od originalnih pokazivačkih polja koja odgovaraju poljima efemernog čvora, dodatnih pokazivačkih polja namenjenih čuvanju promena, pokazivača na kopiju čvora i pokazivača na prethodnike, odnosno inverznih pokazivača. Ako se sa d označi broj pokazivačkih polja u efemernom čvoru, sa p maksimalan broj prethodnika jednog čvora u bilo kojoj verziji, a sa e broj dodatnih pokazivačkih polja, perzistentni čvor sadrži $d + p + e + 1$ pokazivačkih polja [3].

Proces ažuriranja zasniva se na tome da se manje izmene prvo smeštaju u dostupna dodatna polja postojećeg perzistentnog čvora. Kada se, na primer, menja pokazivačko polje u verziji i, nova vrednost može se upisati kao dodatni pokazivač sa imenom polja i oznakom verzije i, pod uslovom da u čvoru postoji slobodno dodatno polje. Međutim, kada su sva dodatna polja iskorišćena, čvor više ne može da primi novu verzionisanu vrednost. Tada se kreira nova kopija čvora sa oznakom verzije trenutne operacije ažuriranja, pri čemu se u kopiju unose najnovije, odnosno trenutno važeće vrednosti originalnih i pokazivačkih polja. Na taj način se sprečava da jedan čvor postane proizvoljno veliki, a nova verzija strukture nastavlja da koristi kopiju kao aktuelnog predstavnika odgovarajućeg efemernog čvora [3].

Ključni problem pri kopiranju čvora jeste održavanje ispravnih veza iz njegovih prethodnika. Ako neki čvor pokazuje na čvor koji je upravo kopiran, tada odgovarajući pokazivač mora biti ažuriran tako da u novoj verziji pokazuje na novu kopiju. Da bi se takvi prethodnici mogli pronaći bez pretraživanja cele strukture, svaki živi pokazivač ima odgovarajući inverzni pokazivač, smešten u čvoru na koji se poka-

zuje. Inverzni pokazivači zato omogućavaju da se pri kopiranju čvora neposredno pronađu čvorovi koji moraju da promene svoje pokazivače. Međutim, ova korekcija može izazvati lančano kopiranje: ako prethodnik nema slobodno dodatno polje u koje bi se upisao novi pokazivač ka kopiji, i on mora biti kopiran, što zatim može zahtevati ažuriranje njegovih prethodnika. U opštim povezanim strukturama takvo širenje može propagirati kroz strukturu, dok se kod binarnih stabala proces odvija kontrolisanije, duž predaka čvora nad kojim je promena nastala [3].

Iako pojedinačno ažuriranje može izazvati kaskadno kopiranje većeg broja čvorova, analiza metodom potencijala pokazuje da je prosečan dodatni utrošak resursa po koraku ažuriranja ograničen konstantom. Uz odgovarajući izbor konstantnog broja dodatnih polja, amortizovano memorijsko zauzeće po koraku ažuriranja iznosi $O(1)$, a ista analiza pokazuje i da amortizovano vreme izvršavanja ažuriranja iznosi $O(1)$. Istovremeno, vreme pristupa ostaje $O(1)$ u najgorem slučaju po operaciji ažuriranja osnovne strukture, jer se broj verzionisanih vrednosti koje se pretražuju unutar jednog čvora ne dopušta da raste neograničeno. Prema tome, metoda kopiranja čvorova zadržava memorijsku efikasnost perzistentne reprezentacije, ali uklanja logaritamsko usporavanje karakteristično za metodu „debelih" čvorova [3].

Primer izvršavanja operacija – Metoda kopiranja čvorova

Radi jasnijeg prikaza metode kopiranja čvorova, može se posmatrati jednostavan primer parcijalno perzistentnog binarnog stabla pretrage (Slika 3.2). Neka se u početnom stanju, označenom kao Verzija 0, posmatra stablo čiji je koren čvor sa vrednošću 10, a njegov desni potomak čvor sa vrednošću 15. Čvor 15 u ovoj verziji nema potomke, pa su njegovi levi i desni pokazivač jednaki `null`. Pošto se radi o binarnom stablu pretrage, svaki čvor ima najviše dva pokazivačka polja, levi i desni pokazivač, pa je u ovom primeru $d = 2$. Takođe, svaki čvor u jednoj verziji stabla ima najviše jednog roditelja, odnosno jednog neposrednog prethodnika, zbog čega je $p = 1$. Radi jednostavnosti i bržeg prikaza prepunjavanja, pretpostavlja se da svaki perzistentni čvor ima tačno jedno dodatno pokazivačko polje, odnosno da je $e = 1$.

U Verziji 1 umeće se vrednost 20. Pošto je 20 veće od 10, a zatim i od 15, novi čvor se umeće kao desni potomak čvora 15. Čvor 15 u tom trenutku ima slobodno dodatno polje, pa se promena ne realizuje kopiranjem, već se u postojeći perzistentni čvor upisuje dodatni zapis koji predstavlja desni pokazivač na čvor 20 sa oznakom Verzije 1. Time čvor 15 ostaje isti fizički čvor, ali njegovo jedino dodatno polje postaje zauzeto. Istovremeno, čvor 20 dobija inverzni pokazivač ka čvoru 15, jer je

čvor 15 njegov neposredni prethodnik u najnovijoj verziji strukture.

U Verziji 2 umeće se vrednost 12. Pretraga u binarnom stablu najpre dolazi do korena 10, zatim prelazi u desno podstablo ka čvoru 15, a pošto je 12 manje od 15, nova vrednost treba da postane levi potomak čvora 15. Međutim, čvor 15 više nema slobodno dodatno polje, jer je ono već iskorišćeno u Verziji 1 za desni pokazivač ka čvoru 20. Zbog toga se aktivira mehanizam kopiranja čvora. Kreira se nova kopija čvora 15, označena kao $15'$, koja dobija oznaku Verzije 2. U novu kopiju se prepisuju najnovije važeće vrednosti čvora 15: osnovna vrednost ostaje 15, desni pokazivač vodi ka čvoru 20, dok se nova promena, odnosno levi pokazivač ka čvoru 12, upisuje u kopiju kao trenutno važeća vrednost.

Pošto je kopiran unutrašnji čvor, nije dovoljno samo napraviti novu kopiju. Potrebno je obezbediti da nova verzija stabla od korena vodi ka toj kopiji. U Verziji 0 i Verziji 1 koren 10 svojim desnim pokazivačem vodi ka starom čvoru 15. Međutim, u Verziji 2 desni pokazivač korena mora voditi ka kopiji $15'$. Upravo ovde se vidi uloga inverznih pokazivača. Stari čvor 15 sadrži informaciju o svom prethodniku, odnosno o čvoru 10, pa se pri njegovom kopiranju ne mora pretraživati cela struktura da bi se pronašao čvor koji pokazuje na njega. Na osnovu inverznog pokazivača pronalazi se prethodnik 10 i u njemu se ažurira odgovarajući desni pokazivač tako da u Verziji 2 pokazuje na novu kopiju $15'$. Pošto koren 10 u ovom primeru ima slobodno dodatno polje, ta promena se može upisati kao dodatni desni pokazivač sa oznakom Verzije 2. Da koren nije imao slobodno dodatno polje, i on bi morao biti kopiran, čime bi se kopiranje propagiralo prema korenu strukture.

Na ovaj način stari čvor 15 ostaje dostupan za ranije verzije, dok kopija $15'$ predstavlja čvor koji se koristi u najnovijoj verziji stabla. Unutrašnje stanje relevantnih čvorova nakon Verzije 2 prikazano je u Tabeli 3.2.

Ovaj primer pokazuje da metoda kopiranja čvorova ne dozvoljava neograničen rast pojedinačnog čvora, već uvodi kopiranje kada se unapred određen broj dodatnih polja popuni. Inverzni pokazivači imaju ključnu ulogu u održavanju ispravnih veza, jer omogućavaju da se pri kopiranju čvora pronađu njegovi prethodnici i da se njihovi pokazivači ažuriraju ka novoj kopiji. Na taj način se veličina čvorova drži pod kontrolom, dok se pristup odgovarajućim vrednostima u čvoru može izvršiti u vremenu $O(1)$.

Tabela 3.2: Stari čvor sa vrednošću 15, kreiran pre kopiranja

Polje	**Vrednost**	**Oznaka verzije**
osnovna vrednost	15	Verzija 0
originalni levi pokazivač	`null`	Verzija 0
originalni desni pokazivač	`null`	Verzija 0
dodatno polje	desni pokazivač na čvor 20	Verzija 1
inverzni pokazivač	prethodnik je čvor 10	Verzija 0–1
pokazivač na kopiju	pokazivač na čvor 15′	Verzija 2

Tabela 3.3: Nova kopija čvora 15′, kreirana u Verziji 2

Polje	**Vrednost**	**Oznaka verzije**
osnovna vrednost	15	Verzija 2
originalni levi pokazivač	pokazivač na čvor 12	Verzija 2
originalni desni pokazivač	pokazivač na čvor 20	Verzija 2
dodatno polje	slobodno	–
inverzni pokazivač	prethodnik je čvor 10	Verzija 2
pokazivač na kopiju	`null`	–

3.3 Metoda razdvajanja čvorova

Metoda razdvajanja čvorova (engl. *Node-Splitting Method*) uvodi se kao varijanta metode kopiranja čvorova namenjena upravo postizanju potpune perzistentnosti kod povezanih struktura podataka ograničenog ulaznog stepena. Njena osnovna ideja jeste da se spreči neograničen rast čvorova, ali i da se omogući dalji razvoj različitih grana stabla verzija. Kao i kod metode kopiranja čvorova, perzistentni čvor ima ograničen broj polja, pa se ne dozvoljava da se u jednom čvoru akumulira proizvoljno mnogo verzionisanih pokazivača. Ako se sa d označi broj pokazivačkih polja u neproširenom efemernom čvoru, a sa p konstantno ograničenje broja prethodnika, uključujući i pristupne pokazivače, tada prošireni efemerni čvor ima $k = d + p$ pokazivačkih polja. Perzistentni čvor u ovoj metodi sadrži $k + 2e + 1$ pokazivačkih polja, pri čemu je e konstanta koja određuje broj dodatnih polja, a sabirak $+1$ predstavlja pokazivač ka sledećem čvoru u istoj porodici kopija [3].

Ključna razlika u odnosu na metodu kopiranja čvorova javlja se u trenutku kada se čvor prepuni. Kod parcijalne perzistentnosti nova kopija čvora obično sadrži najnovije važeće vrednosti, jer se dalja ažuriranja izvršavaju samo nad najnovijom verzijom. U potpunoj perzistentnosti takav pristup nije dovoljan, pošto i starija i novija grana verzija mogu kasnije biti ažurirane. Zbog toga se pri razdvajanju čvo-

ra ne prebacuje samo trenutno stanje u novu kopiju, već se verzionisani pokazivači raspoređuju između starog i novog čvora. Kada čvor postane prepunjen dodatnim pokazivačima, kreira se novi čvor, a približno polovina dodatnih pokazivača premešta se iz starog čvora u novi. Time se ostavlja slobodan prostor u oba čvora, što je presudno za potpunu perzistentnost, jer se i stariji i noviji interval verzija mogu kasnije nezavisno širiti novim ažuriranjima [3].

Održavanje ispravnih pokazivača u ovoj metodi znatno je složenije nego kod parcijalnog kopiranja čvorova. Pošto se čvor razdvaja na dva dela koji pokrivaju različite intervale važenja, potrebno je obezbediti da svi pokazivači koji vode ka tom čvoru u odgovarajućim verzijama sada vode ka pravom članu njegove porodice. Zbog toga se koriste inverzni pokazivači, koji omogućavaju pronalaženje prethodnika čvora koji se razdvaja, kao i inverzni zapisi za pristupne pokazivače. Kada se jedan čvor razdvoji, može biti potrebno da se ažuriraju pokazivači u njegovim prethodnicima. Ako ti prethodnici nemaju dovoljno slobodnog prostora za nove verzionisane pokazivače, i oni se moraju razdvojiti. Na taj način razdvajanje može izazvati kaskadni proces koji se širi kroz strukturu, pri čemu se naizmenično dodaju novi pokazivači i razdvajaju čvorovi sve dok se ponovo ne uspostavi ispravnost svih veza [3].

Iako pojedinačna operacija može izazvati lančano razdvajanje većeg broja čvorova, matematička analiza pokazuje da je ukupno opterećenje sistema kontrolisano. Primenom metode potencijala (engl. *potential method*) dokazuje se da, uz odgovarajući izbor konstante e, amortizovano memorijsko zauzeće po koraku ažuriranja iznosi $O(1)$, a isti amortizovani okvir važi i za vreme izvršavanja ažuriranja. Istovremeno, navigacija kroz strukturu zadržava strogo $O(1)$ vreme u najgorem slučaju po operaciji nad osnovnom strukturom, jer svaki perzistentni čvor sadrži samo konstantno ograničen broj pokazivačkih polja. Prema tome, metoda razdvajanja čvorova omogućava da se povezane strukture ograničenog ulaznog stepena učine potpuno perzistentnim uz konstantnu amortizovanu složenost ažuriranja i konstantno vreme pristupa po koraku [3].

Primer izvršavanja operacija – Metoda razdvajanja čvorova

Radi jasnijeg prikaza metode razdvajanja čvorova, može se posmatrati primer potpuno perzistentnog binarnog stabla pretrage u kome istorija verzija ne predstavlja linearan niz, već stablo verzija. U početnom stanju, označenom kao Verzija 0, stablo sadrži korenski čvor sa vrednošću 10 i njegovog desnog potomka sa vrednošću 15. Čvor 15 u toj verziji nema ni levog ni desnog potomka. Pretpostavlja se da

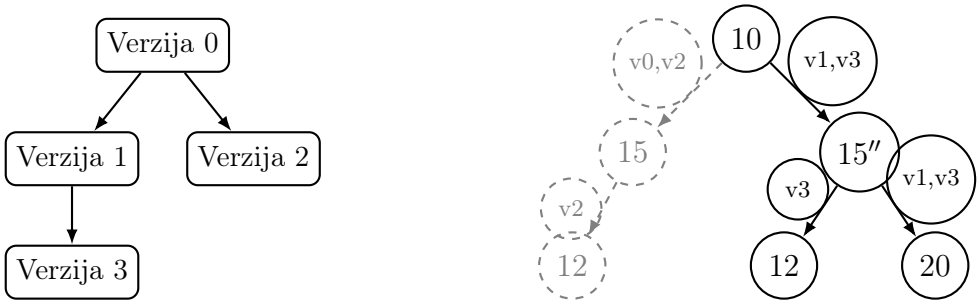
svaki perzistentni čvor ima prostor za $e = 2$ dodatna pokazivačka polja, što znači da čvor može privremeno da primi dve verzionisane promene pre nego što postane neophodno njegovo razdvajanje.

Najpre se iz Verzije 0 formira Verzija 1, tako što se umeće vrednost 20. Pošto je 20 veće od 10, a zatim i od 15, novi čvor se postavlja kao desni potomak čvora 15. U perzistentnoj reprezentaciji ova promena se beleži u jednom dodatnom polju čvora 15, kao desni pokazivač ka čvoru 20 sa oznakom Verzije 1. Time čvor 15 ne biva kopiran niti razdvojen, već samo dobija prvi verzionisani pokazivač. Zatim se novo ažuriranje ne vrši iz Verzije 1, već ponovo iz Verzije 0, čime nastaje Verzija 2. U toj drugoj grani istorije umeće se vrednost 12, koja se, prema pravilima binarnog stabla pretrage, postavlja kao levi potomak čvora 15. Ova promena se upisuje u drugo dodatno polje istog čvora, kao levi pokazivač ka čvoru 12 sa oznakom Verzije 2. Nakon toga čvor 15 sadrži dva dodatna pokazivača koji pripadaju različitim granama istorije, pa su oba njegova dodatna polja iskorišćena.

Do razdvajanja dolazi kada se pokuša novo ažuriranje nad jednom od već postojećih grana. Neka se iz Verzije 1 formira Verzija 3, pri čemu se u toj grani umeće vrednost 12. Sa stanovišta Verzije 1, levi pokazivač čvora 15 još uvek ima vrednost `null`, jer promena iz Verzije 2 pripada drugoj grani istorije i ne važi u Verziji 1. Zbog toga bi umetanje vrednosti 12 iz Verzije 1 trebalo da doda novu verzionisanu vrednost levog pokazivača čvora 15, sada sa oznakom Verzije 3. Međutim, čvor 15 više nema slobodnih dodatnih polja, jer su ona već iskorišćena za pokazivač ka čvoru 20 u Verziji 1 i za pokazivač ka čvoru 12 u Verziji 2. U tom trenutku aktivira se mehanizam razdvajanja čvora.

Razdvajanje se ne svodi na jednostavno kopiranje samo najnovijeg stanja, jer u potpunoj perzistentnosti ne postoji jedinstvena najnovija budućnost strukture. Umesto toga, kreira se novi čvor, označen kao $15''$, a postojeći verzionisani pokazivači raspoređuju se između starog i novog člana iste porodice čvorova. Deo promena ostaje u starom čvoru 15, dok se deo prebacuje u novi čvor $15''$. Na taj način se intervali verzija razdvajaju, ali se istovremeno obezbeđuje da oba čvora imaju slobodan prostor za buduća ažuriranja. U ovom pojednostavljenom prikazu stari čvor 15 zadržava promenu koja pripada grani Verzije 2, dok novi čvor $15''$ preuzima promene koje pripadaju grani koja polazi od Verzije 1, uključujući desni pokazivač ka čvoru 20 i novi levi pokazivač ka čvoru 12 u Verziji 3.

Pošto se u ovom primeru razdvaja unutrašnji čvor stabla, potrebno je održati i veze iz njegovih prethodnika. Čvor 15 ima prethodnika 10, čiji desni pokazivač u od-



(a) Stablo verzija (b) Razdvajanje čvora 15

Slika 3.3: Prikaz metode razdvajanja čvorova. Stablo verzija prikazuje nastanak dve nezavisne grane, dok graf strukture prikazuje razdvajanje verzionisanih pokazivača između starog čvora 15 i novog razdvojenog čvora 15″.

Tabela 3.4: Unutrašnje stanje čvorova nakon razdvajanja čvora 15

Stari čvor 15		
Polje	**Vrednost**	**Oznaka verzije**
osnovna vrednost	15	Verzija 0
originalni levi pokazivač	`null`	Verzija 0
originalni desni pokazivač	`null`	Verzija 0
dodatno polje 1	levi pokazivač na čvor 12	Verzija 2
dodatno polje 2	slobodno	–
inverzni pokazivač	prethodnik je čvor 10	za verzije u starom čvoru
pokazivač na kopiju	pokazivač na čvor 15″	Verzija 3

Novi razdvojeni čvor 15″		
Polje	**Vrednost**	**Oznaka verzije**
osnovna vrednost	15	Verzija 3
originalni levi pokazivač	pokazivač na čvor 12	Verzija 3
originalni desni pokazivač	pokazivač na čvor 20	Verzija 1
dodatno polje 1	slobodno	–
dodatno polje 2	slobodno	–
inverzni pokazivač	prethodnik je čvor 10	za verzije u novom čvoru
pokazivač na kopiju	`null`	–

govarajućim verzijama vodi ka tom čvoru. Inverzni pokazivač u čvoru 15 omogućava da se prethodnik pronađe neposredno, bez pretrage cele strukture. Nakon razdvajanja, pokazivači iz čvora 10 moraju biti usklađeni sa intervalima važenja starog i novog člana porodice čvora 15, tako da verzije koje pripadaju jednoj grani istorije vode ka starom čvoru, dok verzije koje pripadaju drugoj grani vode ka novom razdvojenom čvoru 15″. U opštem slučaju, ako prethodnik nema dovoljno slobodnog prostora za upis takve promene, i on može zahtevati razdvajanje, čime nastaje kaskadno širenje postupka prema korenu strukture. Odnos između verzija u istoriji, kao i fizička struktura čvorova u Verziji 3, prikazani su na Slici 3.3. Unutrašnje stanje čvorova nakon razdvajanja prikazano je u Tabeli 3.4.

Ovaj primer pokazuje zašto je razdvajanje čvorova posebno važno u potpunoj perzistentnosti. Kada se različite grane istorije razvijaju nezavisno, nije dovoljno čuvati samo trenutno najnovije vrednosti, već se moraju očuvati intervali važenja pokazivača koji pripadaju različitim granama stabla verzija. Razdvajanjem se verzionisani pokazivači raspoređuju između više čvorova iste porodice, tako da svaki od njih zadržava ograničenu veličinu i prostor za buduće izmene. Inverzni pokazivači obezbeđuju da se pri tom postupku mogu pronaći i uskladiti prethodnici čvora koji se razdvaja, što je neophodno za očuvanje ispravnog kretanja kroz različite verzije strukture. Time se omogućava da obe grane istorije nastave nezavisno da rastu, dok navigacija kroz pojedinačni čvor ostaje ograničena na konstantan broj polja, odnosno na vreme $O(1)$ po jednoj operaciji nad originalnom strukturom.

3.4 Savremene optimizacije i „lenji" pristup

Pored klasičnih tehnika zasnovanih na čuvanju promena u čvorovima ili na kontrolisanom kopiranju delova strukture, u savremenom proučavanju perzistentnih struktura podataka značajno mesto zauzima i „lenji" pristup izvršavanju. Osnovna ideja lenje evaluacije jeste da se rezultat neke operacije ne izračunava odmah u trenutku kada se operacija formalno izvršava, već se njegovo izračunavanje odlaže do prvog trenutka kada je ta vrednost zaista potrebna. U kontekstu perzistentnih struktura podataka, to znači da se određeni delovi strukture, transformacije ili pomoćne operacije mogu predstaviti kao odložena izračunavanja, umesto da se sva memorijska alokacija i sve izmene izvrše neposredno pri ažuriranju [5].

Ovakav pristup razlikuje se od ranije opisanih metoda, kao što su metoda „debelih" čvorova, metoda kopiranja čvorova i metoda razdvajanja čvorova. Te metode

promene tretiraju neposredno: pri svakom ažuriranju dodaju se nova verzionisana polja, kopiraju se čvorovi ili se čvorovi razdvajaju kako bi se obezbedilo očuvanje prethodnih verzija. Takav pristup može se opisati kao strogo, odnosno neposredno izvršavanje, jer se struktura fizički prilagođava u trenutku ažuriranja. Nasuprot tome, lenji pristup dozvoljava da se deo rada privremeno odloži, pri čemu se u strukturi čuva opis operacije ili odložena vrednost, a stvarno izračunavanje se pokreće tek kada se toj vrednosti pristupi [5].

Ključni mehanizam koji omogućava efikasnost lenje evaluacije jeste pamćenje već izračunatih rezultata, odnosno tehnika memoizacije (engl. *memoization*). Kada se odložena vrednost prvi put zatraži, ona se izračunava, a zatim se dobijeni rezultat čuva u memoriji. Svaki naredni pristup istoj vrednosti ne ponavlja celokupan postupak izračunavanja, već koristi prethodno zapamćen rezultat. Na taj način se izbegava višestruko ponavljanje iste složene operacije, što je naročito važno u perzistentnim strukturama, gde više verzija može deliti iste delove strukture i naknadno pristupati istim odloženim vrednostima [5].

Lenja evaluacija posebno je značajna u strogo funkcionalnom programiranju, gde se strukture podataka grade bez destruktivnih izmena postojećih objekata. Budući da se prethodne verzije ne menjaju, već ostaju dostupne, funkcionalne strukture prirodno poseduju perzistentna svojstva. Međutim, neposredno kopiranje delova strukture pri svakoj operaciji može dovesti do previsokog usporavanja i rasta memorije. Uvođenjem lenje evaluacije i tehnike memoizacije moguće je deo tog opterećenja rasporediti kroz kasnije pristupe, čime se za pojedine strukture postiže efikasna amortizovana složenost, uključujući operacije sa amortizovanim vremenom $O(1)$ [5].

Prema tome, savremeni „lenji” pristup ne zamenjuje osnovnu ideju perzistentnosti, već predstavlja drugačiji način upravljanja trenutkom izvršavanja i alokacije memorije. Dok klasične metode nastoje da odmah fizički organizuju sve promene koje nastaju ažuriranjem, lenja evaluacija uvodi mogućnost da se deo rada odloži, a zatim izvrši samo ako postane potreban. Time se perzistentne strukture približavaju modelu u kome se memorijska efikasnost, deljenje strukture i amortizovana vremenska složenost postižu kombinovanjem nepromenljivosti, odloženog izračunavanja i pamćenja već dobijenih rezultata.

3.5 Uporedna analiza metoda implementacije

Tri osnovne metode implementacije perzistentnosti razlikuju se prema načinu na koji čuvaju istoriju promena i stepenu perzistentnosti koji podržavaju, nudeći različite kompromise između vremenske efikasnosti i zauzeća memorije [3].

Izbor konkretne metode zavisi od nivoa perzistentnosti koji struktura treba da podrži, ali i od odnosa između prostorne efikasnosti i brzine pristupa. Metoda „debelih" čvorova nudi najjednostavniji model, jer ne zahteva složeno preusmeravanje pokazivača niti održavanje kopija čvorova pri svakoj promeni. Promene se beleže lokalno, pa je memorijsko zauzeće po koraku ažuriranja $O(1)$. Međutim, pošto jedan čvor može akumulirati više verzionisanih vrednosti istog polja, pri pristupu određenoj verziji mora se pronaći odgovarajuća vrednost na osnovu oznake verzije. Zbog toga vreme pristupa više nije konstantno, već iznosi $O(\log h)$ ukoliko se posmatra broj promena nad pojedinačnim čvorom.

Metode kopiranja i razdvajanja čvorova rešavaju ovaj problem ograničavanjem broja dodatnih polja u jednom čvoru. Pošto je veličina čvora unapred ograničena konstantom, pretraga unutar čvora ne zavisi od ukupnog broja verzija niti od broja prethodnih izmena, pa vreme pristupa ostaje $O(1)$ u najgorem slučaju po operaciji nad osnovnom strukturom. Metoda kopiranja čvorova pogodna je za parcijalnu perzistentnost, gde verzije formiraju linearnu istoriju i ažuriranja se izvršavaju samo nad najnovijom verzijom. Nasuprot tome, metoda razdvajanja čvorova uvodi dodatnu složenost potrebnu za potpunu perzistentnost, u kojoj se istorija verzija organizuje kao stablo, a starije verzije mogu postati osnova za nove nezavisne grane. Uporedne karakteristike ovih metoda prikazane su u Tabeli 3.5 [3].

Na osnovu prikazanog poređenja može se zaključiti da metoda „debelih" čvorova predstavlja jednostavan i memorijski efikasan pristup, ali po cenu sporijeg pristupa verzionisanim vrednostima. Metoda kopiranja čvorova uvodi strože ograničenje veličine čvora i time obezbeđuje konstantno vreme pristupa, ali je prirodno vezana za parcijalnu perzistentnost i linearan tok verzija. Metoda razdvajanja čvorova zadržava prednosti ograničene veličine čvora, ali dodatnim mehanizmima podele verzionisanih pokazivača i održavanja inverznih pokazivača omogućava rad u potpuno perzistentnom okruženju. Prema tome, složenost implementacije raste od metode „debelih" čvorova ka metodi razdvajanja čvorova, ali se istovremeno povećava i sposobnost strukture da efikasno podrži složenije obrasce verzionisanja [3].

Tabela 3.5: Uporedna analiza metoda implementacije perzistentnosti

Metoda	**Podržani tip**	**Vreme pristupa (najgori slučaj)**	**Ažuriranje (amortizovano)**	**Zauzeće prostora (po ažuriranju)**
Metoda „debelih" čvorova	Parcijalna i potpuna	$O(\log h)$	$O(\log h)$	$O(1)$
Metoda kopiranja čvorova	Parcijalna perzistentnost	$O(1)$	$O(1)$	$O(1)$ amortizovano
Metoda razdvajanja čvorova	Potpuna perzistentnost	$O(1)$	$O(1)$	$O(1)$ amortizovano

Glava 4

Konkretne perzistentne strukture podataka

Nakon razmatranja opštih tehnika implementacije perzistentnosti, pažnja se može usmeriti na konkretne strukture podataka u kojima se perzistentnost javlja kao prirodna osobina ili kao sredstvo za efikasno rešavanje specifičnih algoritamskih problema. U funkcionalnom programiranju, nepromenljivost podataka omogućava da prethodne verzije strukture ostanu dostupne nakon svake operacije ažuriranja, zbog čega se strukture poput steka i reda mogu analizirati kao osnovni primeri perzistentnog modela. Sa druge strane, u računarskoj geometriji se perzistentne strukture, naročito perzistentna stabla pretrage, koriste za rešavanje složenijih problema, kao što je problem lociranja tačke unutar poligonalne podele ravni, gde je potrebno efikasno pristupati različitim stanjima strukture nastalim tokom izvršavanja algoritma. Na taj način se teorijski koncepti iz prethodnog poglavlja povezuju sa konkretnim strukturama podataka i primenama u kojima perzistentnost ima neposrednu algoritamsku vrednost [5, 1].

4.1 Perzistentni stek i red kroz funkcionalnu prizmu

U funkcionalnom programiranju, zbog osnovnog principa nepromenljivosti podataka, perzistentnost nije dodatni mehanizam već prirodna posledica dizajna. Novo stanje strukture dobija se konstruisanjem nove vrednosti koja deli neizmenjene delove prethodne strukture, dok originalna verzija ostaje netaknuta i dostupna [5].

Najjednostavniji primer takvog ponašanja javlja se kod steka. Stek se u funkcionalnom okruženju prirodno implementira pomoću jednostruko povezane liste, pri čemu vrh steka odgovara početku liste. Neka se posmatra stek S_1 koji sadrži elemente $A \to B \to C$, gde je A vrh steka, a čvorovi B i C predstavljaju njegov nastavak. Kada se izvrši operacija $push(X)$, ne menja se postojeći čvor A, niti se bilo koji pokazivač u postojećoj listi prepisuje. Umesto toga, kreira se novi čvor X, čiji pokazivač vodi ka dotadašnjem vrhu, odnosno ka čvoru A. Tako nastaje nova verzija steka S_2, predstavljena listom $X \to A \to B \to C$, dok prethodna verzija S_1 i dalje postoji kao lista $A \to B \to C$.

Ovaj primer ilustruje princip deljenja strukture (engl. *structural sharing*). Verzije S_1 i S_2 nisu dve nezavisne kopije celog steka, već dve strukture koje dele isti rep liste $A \to B \to C$. Jedini novi memorijski objekat koji nastaje pri operaciji $push(X)$ jeste čvor X, pa operacija zahteva $O(1)$ vremena i $O(1)$ dodatnog prostora. Slično tome, operacija *pop*() ne briše fizički vrh steka, već samo formira novu verziju čiji ulazni pokazivač pokazuje na sledeći čvor u listi. Ako se *pop*() primeni na stek S_2, nova verzija ponovo počinje od čvora A, dok verzija S_2 ostaje dostupna ukoliko postoji referenca na njen vrh X. Zbog toga stek predstavlja primer strukture kod koje funkcionalna nepromenljivost gotovo neposredno proizvodi potpunu i efikasnu perzistentnost [5].

Kod reda je situacija složenija, jer red ne koristi samo jedan kraj strukture. Red funkcioniše po principu FIFO, što znači da se elementi dodaju na jedan kraj, a uklanjaju sa drugog. Takvo ponašanje je jednostavno u efemernom modelu, gde se mogu menjati pokazivači na početak i kraj strukture, ali je problematično u funkcionalnom modelu, jer bi dodavanje na kraj jednostruko povezane liste zahtevalo izmenu poslednjeg čvora. Pošto takva destruktivna izmena nije dozvoljena, funkcionalni red se konstruiše pomoću dva steka, odnosno dve liste. Jedan stek predstavlja prednji deo reda, iz koga se elementi uklanjaju, dok drugi stek predstavlja zadnji deo reda, u koji se elementi dodaju.

U takvoj reprezentaciji operacija dodavanja elementa ne pokušava da menja kraj postojeće liste, već dodaje novi element na vrh zadnjeg steka. Time se zadržava ista efikasnost kao kod običnog funkcionalnog steka, jer se kreira samo jedan novi čvor. Operacija uklanjanja elementa izvršava se iz prednjeg steka, takođe u vremenu $O(1)$, sve dok taj stek nije prazan. Problem nastaje kada se prednji stek isprazni. Tada se elementi iz zadnjeg steka moraju obrnuti i prebaciti u prednji stek, kako bi se obnovio ispravan FIFO redosled. Na primer, ako su elementi u zadnjem steku memorisani

obrnutim redosledom dodavanja, tek njihovo obrtanje daje redosled u kome treba da budu uklonjeni iz reda.

Cena obrtanja zadnjeg steka može biti $O(n)$, jer zahteva prolazak kroz sve elemente tog steka. U efemernom okruženju ova složenost može se rasporediti kroz niz operacija, pa se dobija amortizovano vreme $O(1)$. Međutim, u perzistentnom okruženju javlja se dodatni problem. Ako se neposredno pre skupog obrtanja formira više različitih verzija reda, svaka od tih verzija može kasnije nezavisno zahtevati isto obrtanje. Bez dodatnog mehanizma, isti linearni utrošak resursa mogao bi biti plaćen više puta, što bi narušilo amortizovanu efikasnost. Drugim rečima, standardna amortizovana analiza iz efemernog modela nije dovoljna ako se odloženi postupak može ponavljati kroz više perzistentnih verzija [5].

Rešenje se zasniva na lenjoj evaluaciji i tehnici memoizacije. Lenja evaluacija omogućava da se skupo obrtanje zadnjeg steka ne izvrši odmah u celini, već da se predstavi kao odloženo izračunavanje koje se realizuje tek kada rezultat zaista postane potreban. Tehnika memoizacije zatim obezbeđuje da se jednom izračunati deo tog rezultata zapamti, tako da naredni pristupi, bilo iz iste verzije ili iz verzija koje dele isti odloženi proračun, ne ponavljaju isti rad. Na taj način se sprečava višestruko ponavljanje iste operacije obrtanja, a red zadržava amortizovano vreme $O(1)$ za osnovne operacije i u perzistentnom okruženju.

Zato se perzistentni stek i perzistentni red mogu posmatrati kao dva različita nivoa složenosti funkcionalnog pristupa strukturama podataka. Stek ilustruje najjednostavniji slučaj, u kome nepromenljivost i deljenje repa liste neposredno daju efikasnu perzistentnost. Red pak ilustruje složeniji slučaj, u kome ista pravila nepromenljivosti zahtevaju pažljiviju organizaciju pomoću dva steka, kao i upotrebu lenje evaluacije i tehnike memoizacije radi očuvanja amortizovane efikasnosti. Time se jasno pokazuje da funkcionalne strukture nisu namerno komplikovane, već da njihova organizacija proizlazi iz osnovnog ograničenja da se postojeća memorija ne sme destruktivno menjati [5].

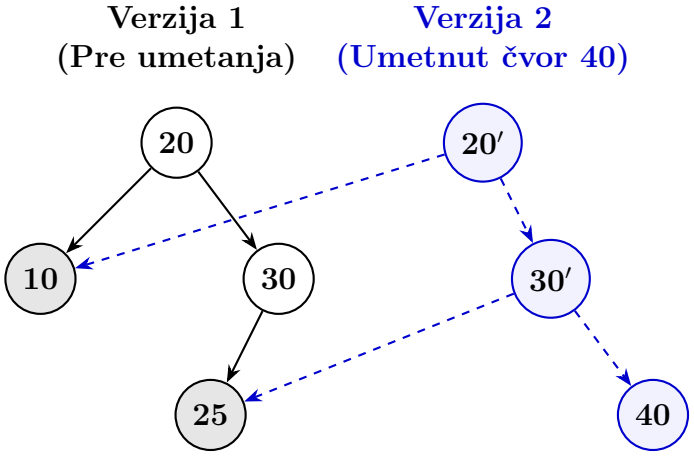
4.2 Perzistentno binarno stablo pretrage

Binarno stablo pretrage predstavlja jednu od ključnih struktura za organizaciju uređenih podataka, jer omogućava da se operacije pretrage, umetanja i brisanja oslanjaju na poredak elemenata u čvorovima. U efemernom obliku, svaka promena stabla neposredno menja postojeće pokazivače i time uništava prethodno stanje

strukture. Nakon umetanja ili brisanja, dostupna ostaje samo nova verzija stabla, dok se raniji raspored čvorova može rekonstruisati jedino ako je prethodno posebno sačuvan. U perzistentnom obliku cilj je drugačiji: svaka operacija ažuriranja treba da proizvede novu verziju stabla, ali tako da stare verzije ostanu dostupne preko svojih ranijih korena. Na taj način binarno stablo pretrage ne predstavlja samo trenutno stanje skupa podataka, već i istoriju njegovih promena kroz niz verzija.

Jedan od najprirodnijih načina za implementaciju perzistentnog binarnog stabla pretrage jeste *kopiranje putanje* (engl. *path copying*). Ova tehnika se zasniva na zapažanju da operacija umetanja ili izmene ne utiče na celo stablo, već samo na putanju od korena do mesta na kome se promena izvršava. Zbog toga nije potrebno kopirati svih n čvorova, već samo one čvorove kroz koje algoritam prolazi tokom pretrage odgovarajuće pozicije. Sva podstabla koja se ne nalaze na toj putanji ostaju neizmenjena i mogu se deliti između stare i nove verzije stabla.

Neka se, na primer, posmatra Verzija 1 stabla (Slika 4.1) u kojoj je koren čvor sa vrednošću 20. Njegovo levo dete je čvor 10, dok je desno dete čvor 30. Čvor 30 ima levo dete 25, a desno dete mu je prazno. Pretpostavlja se da u ovu strukturu treba umetnuti vrednost 40. Standardna pretraga mesta za umetanje najpre poredi 40 sa korenom 20 i, pošto je $40 > 20$, prelazi u desno podstablo. Zatim se 40 poredi sa čvorom 30 i, pošto je $40 > 30$, utvrđuje se da novi element treba da bude umetnut kao desno dete čvora 30.



Slika 4.1: Vizuelni prikaz tehnike kopiranja putanja i deljenja strukture u perzistentnom binarnom stablu pretrage. Stara i nova verzija zadržavaju sopstvene korene, dok se neizmenjena podstabla (sivi čvorovi) efikasno dele u memoriji.

U efemernom stablu bi se u tom trenutku neposredno promenio desni pokazivač postojećeg čvora 30, tako da pokazuje na novi čvor 40. Time bi prethodna verzija

stabla bila izgubljena. U perzistentnom stablu se, međutim, postojeći čvor 30 se ne menja. Umesto toga, pri povratku iz rekurzije kreira se nova kopija tog čvora, označena kao $30'$. Nova kopija $30'$ zadržava istu osnovnu vrednost kao stari čvor 30, preuzima staro levo dete 25, ali kao novo desno dete dobija čvor 40. Nakon toga se kreira i nova kopija korena, označena kao $20'$. Novi koren $20'$ preuzima staro levo dete 10, jer levo podstablo nije učestvovalo u promeni, dok njegov desni pokazivač sada vodi ka novom čvoru $30'$. Time se formira nova verzija stabla, čiji je koren $20'$, dok stara verzija i dalje počinje od starog korena 20 [3].

Ovaj primer jasno pokazuje princip deljenja strukture (engl. *structural sharing*). Stara verzija stabla ima koren 20 i desno dete 30, dok nova verzija ima koren $20'$ i desno dete $30'$. Međutim, čvorovi koji nisu bili na putanji izmene se ne kopiraju. Levo podstablo sa čvorom 10 ostaje isto i koristi ga nova verzija preko pokazivača iz čvora $20'$. Isto važi i za čvor 25, koji ostaje levo dete nove kopije $30'$, iako je fizički isti čvor koji je postojao u staroj verziji. Na taj način dve verzije stabla imaju različite korene i različite čvorove na promenjenoj putanji, ali dele sva podstabla koja promenom nisu zahvaćena.

Sa implementacionog stanovišta, ovakav model zahteva pažljivo upravljanje memorijom, jer se stari čvorovi ne smeju osloboditi dok god postoji verzija koja na njih može da pokazuje. U jezicima sa automatskim sakupljanjem otpadaka, kao što je *Go* ili *Java*, stari čvorovi ostaju živi sve dok postoji referenca na stari koren ili na neki drugi objekat koji ih čini dostupnim. Kada više ne postoji nijedna referenca na određenu verziju, sakupljač otpadaka može ukloniti čvorove koji više nisu dostupni. U jezicima u kojima se memorijom upravlja eksplicitnije, isti problem se može rešavati pametnim pokazivačima, odnosno mehanizmima kao što su deljeni pokazivači (engl. *shared pointers*), koji čuvaju podatak o tome koliko referenci vodi ka određenom objektu. Time se obezbeđuje da deljeni čvorovi ostanu u memoriji dok god ih koristi makar jedna verzija stabla [2].

Asimptotska prednost kopiranja putanja proizlazi iz činjenice da se broj novih čvorova vezuje za visinu stabla, a ne za ukupan broj čvorova u strukturi. Kada bi se pri svakoj promeni kopiralo celo stablo, i složenost operacije i memorijsko zauzeće iznosili bi $O(n)$. Kod kopiranja putanja kreiraju se samo novi čvorovi na putanji od korena do mesta izmene. Ako je stablo balansirano, njegova visina je $O(\log n)$, pa operacija umetanja, brisanja ili izmene zahteva $O(\log n)$ vremena i $O(\log n)$ dodatnog prostora. Zbog toga perzistentno binarno stablo pretrage omogućava očuvanje istorije strukture bez potpunog dupliranja memorije, uz efikasnost koja ostaje pro-

porcionalna visini balansiranog stabla.

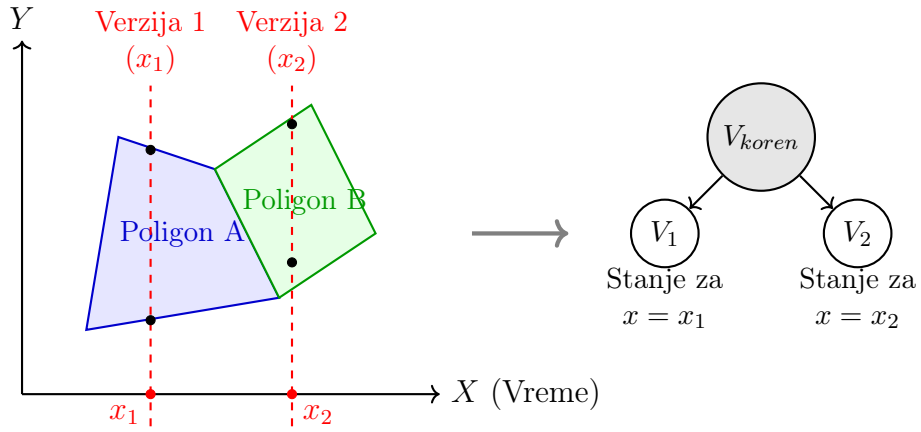
4.3 Perzistentno segmentno stablo u računarskoj geometriji

Jedna od najznačajnijih primena perzistentnih struktura podataka javlja se u računarskoj geometriji, gde je za proizvoljnu tačku $Q(x_q, y_q)$ potrebno efikasno odrediti kojoj oblasti ta tačka pripada unutar određene poligonalne podele ravni. Ako bi se za svaki upit proveravao odnos tačke prema svim poligonima ili svim segmentima koji čine granice podele, vreme izvršavanja bilo bi previsoko za sisteme u kojima se očekuje veliki broj upita. Zbog toga se ovaj izazov prevazilazi izgradnjom strukture podataka koja omogućava da se nakon početne obrade svaki upit reši logaritamskom brzinom.

Osnovna ideja efikasnijeg rešenja zasniva se na tehnici brišuće prave (engl. *sweep line*), odnosno na pretvaranju dvodimenzionalnog problema u niz jednodimenzionalnih stanja. Zamišlja se vertikalna prava (Slika 4.2) koja se kreće duž x-ose sleva nadesno kroz podeone tačke ravni. Dok se prava pomera, menja se skup segmenata koje ona trenutno preseca. Promene nastaju samo u diskretnim trenucima, odnosno u koordinatama u kojima brišuća prava nailazi na levi ili desni kraj nekog segmenta. Kada prava naiđe na levi kraj segmenta, taj segment postaje aktivan i ubacuje se u strukturu pretrage. Kada prava naiđe na desni kraj segmenta, segment prestaje da bude aktivan i briše se iz strukture. Između dva susedna diskretna trenutka skup aktivnih segmenata ostaje isti, pa se za sve tačke čija se x-koordinata nalazi u tom intervalu može koristiti ista verzija strukture [1].

U takvom modelu x-koordinata dobija ulogu vremenske dimenzije, jer određuje u kom trenutku kretanja brišuće prave se upit posmatra. Sa druge strane, y-koordinata se koristi za pretragu unutar strukture koja sadrži aktivne segmente u tom trenutku. Segmenti koji presecaju trenutnu vertikalnu pravu mogu se urediti prema visini njihovog preseka sa tom pravom, pa se pitanje određivanja oblasti kojoj pripada tačka svodi na pronalaženje odgovarajućeg položaja po y-koordinati. Time se geometrijski problem u ravni organizuje kao izbor jedne istorijske verzije strukture prema x_q, a zatim kao standardna pretraga u toj verziji prema y_q.

Međutim, ako bi se ovakav algoritam oslanjao isključivo na efemerne strukture, za odgovaranje na kasnije upite bilo bi potrebno eksplicitno sačuvati stanje strukture za svaki relevantan položaj prave. Naivan pristup bi zato podrazumevao formiranje



Slika 4.2: Konceptualni prikaz tehnike brišuće prave. Vertikalna prava prelazi preko x-ose, a svaki njen presečni položaj (x_1, x_2) odgovara posebnoj verziji perzistentnog segmentnog stabla.

posebne potpune kopije stabla nakon svakog diskretnog trenutka. Ako postoji n segmenata, broj takvih prelomnih trenutaka je linearan po n, a svaka kopija stabla bi mogla sadržati srazmerno velik broj elemenata. Takvo eksplicitno čuvanje svih stanja dovelo bi do zauzeća memorije reda $\mathcal{O}(n^2)$, što je za velike geometrijske instance praktično neprihvatljivo.

Perzistentnost rešava ovaj problem tako što se ne čuva potpuna kopija stabla za svaki trenutak promene, već samo nova verzija koja deli neizmenjene delove sa prethodnim verzijama. Svaki prelomni trenutak na x-osi proizvodi novu verziju stabla aktivnih segmenata, odnosno novi koren preko koga se može pristupiti stanju strukture koje važi u tom intervalu. Kada se segment ubacuje ili briše, menja se samo putanja u stablu pretrage koja je zahvaćena tom operacijom, dok se ostala podstabla dele između stare i nove verzije. Ovaj princip neposredno koristi tehniku kopiranja putanja. Zbog toga svaka karakteristična x-koordinata ne zahteva novu nezavisnu strukturu u memoriji, već samo novu verziju perzistentnog stabla.

Postupak pretrage zatim koristi obe koordinate tačke $Q(x_q, y_q)$ na različite načine. Najpre se prema vrednosti x_q određuje interval na x-osi, oivičen sa dva uzastopna događaja brišuće prave, u koji data tačka upada. To se može uraditi binarnom pretragom nad sortiranim nizom diskretnih događaja, čime se pronalazi koren verzije stabla koja predstavlja važeći skup aktivnih segmenata za taj interval. Nakon izbora odgovarajuće verzije, pretraga se nastavlja unutar tog perzistentnog stabla, gde se prema y_q određuje položaj tačke Q. Drugim rečima, koordinata x_q služi za izbor istorijskog stanja strukture, dok y_q služi za navigaciju unutar izabrane verzije stabla.

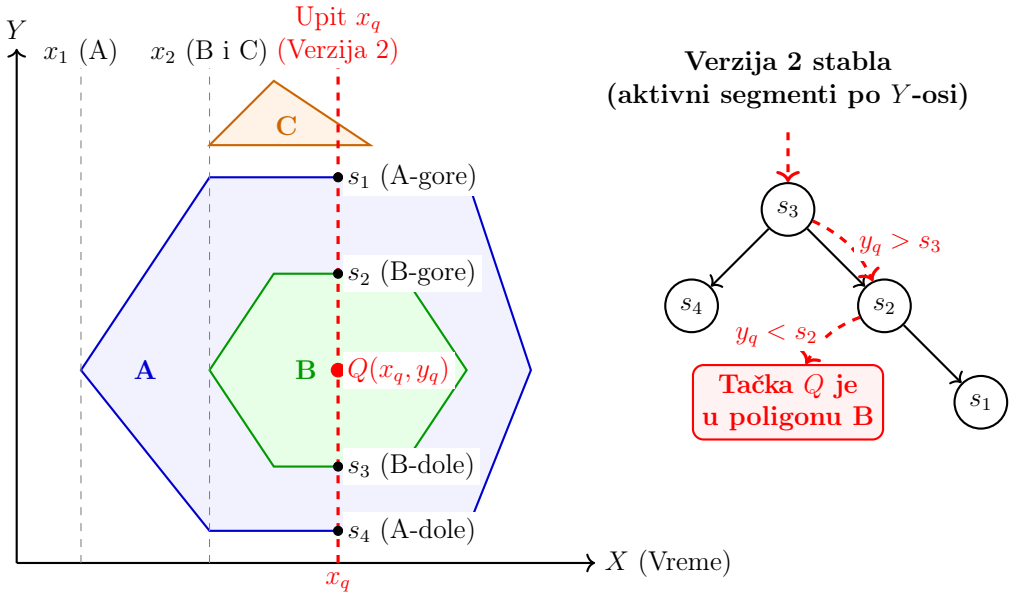
Ovakav postupak pokazuje suštinsku prednost perzistentnosti u računarskoj geometriji: moguće je čuvati veliki broj logičkih stanja strukture bez izdvajanja velike količine fizičke memorije za nezavisne kopije. Zahvaljujući tome, upit lociranja tačke može se izvršiti u vremenu $O(\log n)$, dok ukupni potrebni prostor ostaje $O(n)$ umesto nepraktičnog $O(n^2)$. Time perzistentno stablo pretrage predstavlja ključni mehanizam koji omogućava da se dinamička priroda brišuće prave pretvori u statičku strukturu pogodnu za veliki broj brzih geometrijskih upita [1].

Studija slučaja: Višestruki poligoni i kompleksna pretraga

Stvarna prednost perzistentnog segmentnog stabla najjasnije se uočava kada se umesto jednostavnog primera sa malim brojem segmenata posmatra složenija ravanska podela. Neka se, radi ilustracije, posmatra ravan u kojoj se nalaze tri poligona, označena sa A, B i C. Granice ovih poligona predstavljene su segmentima, a cilj strukture podataka jeste da za proizvoljnu tačku $Q(x_q, y_q)$ efikasno odredi kom poligonu tačka pripada. U ovakvom scenariju nije dovoljno samo proveriti da li je tačka iznad ili ispod jednog segmenta, već je potrebno razmotriti uređen skup više aktivnih segmenata koji u datom trenutku seku istu vertikalnu pravu.

Kretanje brišuće prave može se posmatrati kao hronološki niz događaja na x-osi. Na koordinati x_1 prava prvi put nailazi na poligon A, pa se segmenti koji pripadaju njegovoj granici uvode u strukturu aktivnih segmenata. Time nastaje Verzija 1 perzistentnog stabla, koja opisuje stanje ravanske podele neposredno nakon pojavljivanja poligona A. Zatim, na koordinati x_2, brišuća prava nailazi na početke poligona B i C. U tom trenutku struktura se ažurira dodavanjem novih segmenata koji postaju aktivni, ali se pri tome ne pravi potpuna kopija prethodnog stabla. Umesto toga, kreira se Verzija 2, koja deli neizmenjene delove memorije sa Verzijom 1, dok se novi čvorovi dodaju samo na putanjama koje su pogođene promenama.

Neka se sada posmatra koordinata x_q tražene tačke, pri čemu važi da se x_q nalazi između x_2 i narednog događaja x_3. U tom intervalu skup aktivnih segmenata se ne menja, pa je za sve tačke čija se x-koordinata nalazi između x_2 i x_3 važeća ista verzija stabla, odnosno Verzija 2. Pretpostavlja se da brišuća prava u položaju x_q seče četiri relevantna segmenta: gornju ivicu poligona A, označenu kao s_1, gornju ivicu poligona B, označenu kao s_2, donju ivicu poligona B, označenu kao s_3, i donju ivicu poligona A, označenu kao s_4. Ovi segmenti su u tom trenutku živi, što znači da pripadaju trenutno aktivnom skupu segmenata i da su organizovani u Verziji 2 perzistentnog stabla prema svom vertikalnom poretku.



Slika 4.3: Studija slučaja složene pretrage: Brišuća prava na koordinati x_q seče segmente s_1, s_2, s_3 i s_4. Tačka $Q(x_q, y_q)$ se uspešno locira binarnom pretragom unutar istorijske Verzije 2 perzistentnog stabla.

Proces pretrage tačke $Q(x_q, y_q)$ odvija se u dve jasno odvojene faze. Najpre se koristi koordinata x_q kako bi se odredilo kojoj vremenskoj verziji stabla treba pristupiti. Pošto se x_q nalazi između događaja x_2 i x_3, binarna pretraga nad sortiranim nizom događaja na x-osi određuje da je za taj interval važeća Verzija 2. Nakon toga se iz niza korenova verzija uzima koren Verzije 2, dok se ostale verzije ne razmatraju u daljoj geometrijskoj pretrazi. Time se problem iz izbora odgovarajućeg istorijskog stanja svodi na pristup jednom konkretnom perzistentnom stablu.

U drugoj fazi koristi se vrednost y_q. Nad izabranom Verzijom 2 izvršava se binarna pretraga prema vertikalnom poretku aktivnih segmenata. Tokom pretrage sistem poredi položaj tačke $Q(x_q, y_q)$ sa segmentima smeštenim u čvorovima stabla. Ako se utvrdi da je y_q geometrijski ispod segmenta s_2, ali iznad segmenta s_3, tada se zaključuje da se tačka nalazi između gornje i donje ivice poligona B. Pošto upravo ta dva segmenta ograničavaju unutrašnjost poligona B u preseku sa vertikalnom pravom na koordinati x_q, struktura može da odredi da tačka Q pripada poligonu B.

Ovaj primer pokazuje kako perzistentno segmentno stablo omogućava lociranje tačke u vremenu $\mathcal{O}(\log n)$, bez potrebe za proverom svih poligona, svodeći geometrijski problem na kontrolisanu pretragu kroz istorijske verzije [1].

Glava 5

Praktična implementacija i analiza performansi

5.1 Arhitektura rešenja i izbor tehnologija

Cilj ovog poglavlja jeste praktična implementacija perzistentnog binarnog stabla pretrage i empirijsko poređenje takve strukture sa njenom efemernom verzijom, kao i poređenje različitih tehnika perzistentnosti. Nakon teorijskog razmatranja perzistentnosti, u ovom delu rada prikazuje se kako se koncept perzistentnog stabla može realizovati u konkretnom programskom okruženju. Kompletan izvorni kôd implementacije koji je analiziran u nastavku dostupan je na javnom repozitorijumu platforme *GitHub* na adresi: `https://github.com/mirjanajocovic/persistent-bst-undo-redo`.

Cilj rada nije samo implementacija perzistentnog binarnog stabla pretrage, već i izgradnja sistema za poništavanje i ponovno izvršavanje operacija, odnosno sistema zasnovanog na konceptima *Undo* i *Redo*. Takav sistem se neposredno oslanja na čuvanje prethodnih verzija strukture, zbog čega perzistentno binarno stablo pretrage predstavlja pogodnu osnovu za njegovo modelovanje. Posebna pažnja posvećuje se operaciji umetanja, jer ona jasno pokazuje osnovnu razliku između efemernog i perzistentnog pristupa, kao i razlike između same tehnike kopiranja putanja i metode debelih čvorova.

Za implementaciju je izabran programski jezik Go, odnosno *Golang* [2]. Go predstavlja kompajliran programski jezik visokih performansi, sa jednostavnom sintaksom i snažnim sistemom tipova. Ove osobine čine ga pogodnim za algoritamska testiranja nad velikim skupovima podataka, gde su važni brzina izvršavanja, jasna

struktura koda i kontrolisano ponašanje programa. Snažna tipizacija omogućava precizno definisanje struktura podataka, dok kompajlirana priroda jezika doprinosi pouzdanijem merenju vremena izvršavanja u poređenju sa interpretiranim okruženjima.

Posebno važan razlog za izbor jezika Go jeste postojanje ugrađenog mehanizma za automatsko upravljanje memorijom, odnosno sakupljača otpadaka (engl. *Garbage Collector*). Kod perzistentnih struktura podataka, a naročito kod tehnike kopiranja putanja, svaka izmena može dovesti do kreiranja novih čvorova na putanji od korena do mesta promene. Prethodne verzije stabla ostaju dostupne dok god postoji pokazivač na njihov koren, zbog čega se delovi strukture mogu deliti između više verzija. Međutim, kada određena istorijska verzija više nije potrebna i kada na njen koren više ne ukazuje nijedan aktivan deo programa, postavlja se pitanje oslobađanja memorije koju zauzimaju čvorovi dostupni isključivo iz te verzije.

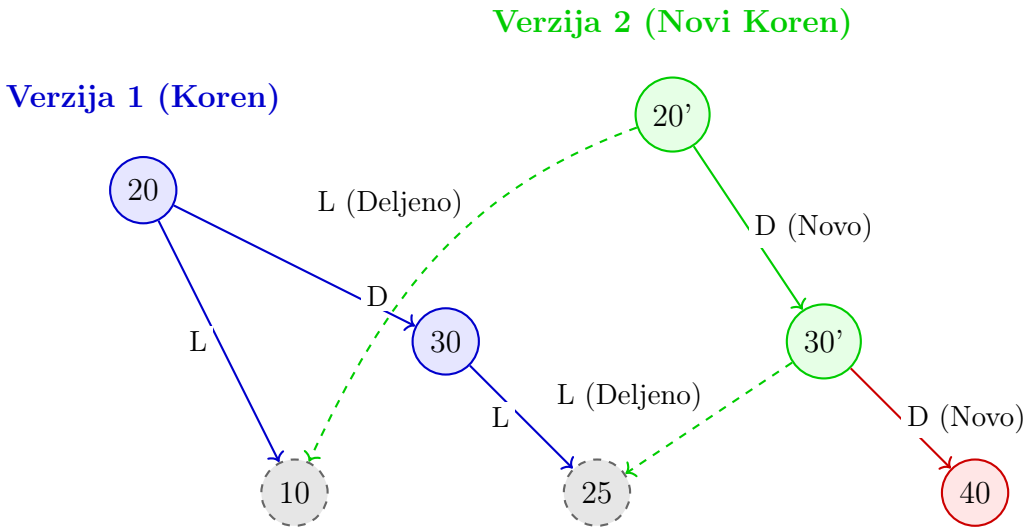
U jezicima bez automatskog upravljanja memorijom, ovakav scenario zahtevao bi pažljivo ručno praćenje referenci i eksplicitno oslobađanje čvorova, što povećava složenost implementacije i rizik od grešaka. Go taj problem rešava pomoću sakupljača otpadaka, koji automatski prepoznaje objekte do kojih se više ne može doći iz aktivnog dela programa i bezbedno oslobađa zauzetu memoriju. Na taj način programer može da se usmeri na logiku perzistentnosti i kreiranja novih verzija stabla, bez potrebe za kompleksnim ručnim upravljanjem životnim ciklusom pokazivača [2].

Arhitektura rešenja obuhvata nekoliko međusobno povezanih celina. Najpre se definiše struktura čvora binarnog stabla pretrage, kao i tri varijante implementacije: efemerna, perzistentna primenom tehnike kopiranja putanja i perzistentna primenom metode debelih čvorova radi komparativne analize. Zatim se uvodi posebna struktura za upravljanje istorijom verzija, označena kao `HistoryManager`, čija je uloga da čuva niz korenova prethodno kreiranih verzija i pokazivač na trenutno aktivnu verziju. Na taj način operacije *Undo* i *Redo* mogu da se realizuju u vremenu $O(1)$.

Pored osnovne funkcionalnosti, rešenje obuhvata i analizu graničnih slučajeva koji nastaju pri radu sa istorijom verzija. Posebno je značajan slučaj promene prošlosti, kada se nakon operacije *Undo* izvrši nova operacija umetanja. U takvoj situaciji prethodno dostupna buduća istorija mora biti odsečena. Završni deo implementacije čini eksperimentalno testno okruženje za generisanje nasumičnih vrednosti, merenje vremena izvršavanja i procenu utroška memorije različitih pristupa.

5.2 Implementacija perzistentnog binarnog stabla pretrage

Implementacija perzistentnog binarnog stabla pretrage tehnikom kopiranja putanja zasniva se na specifičnoj organizaciji pokazivača u memoriji, što je ilustrovano na Slici 5.1. Prikazana arhitektura vizuelno objašnjava osnovni princip ovog pristupa: umesto destruktivne izmene postojećih čvorova, prilikom svake operacije umetanja formira se nova verzija strukture čime se staro stanje u potpunosti čuva. Ne kopira se celo stablo, već samo čvorovi na putanji od korena do mesta izmene, dok se ostali delovi stabla dele. U stablo sa početnim korenom 20 (Verzija 1) umeće se nova vrednost 40, čime nastaju novi čvorovi 30' i 20' (Verzija 2), dok se nepromenjeni čvorovi 10 i 25 bezbedno dele između obe verzije.



Slika 5.1: Arhitektura u memoriji: Funkcija `InsertPersistent` kreira nove čvorove na putanji do mesta umetanja, dok neizmenjena podstabla ostaju deljena u memoriji.

Osnovni element implementacije predstavlja struktura `Node`. U programskom jeziku Go, ova struktura se definiše na sledeći način:

Isečak koda 5.1: Definicija čvora binarnog stabla pretrage

```
type Node struct {
        Value int
        Left  *Node
        Right *Node
}
```

Struktura sadrži celobrojnu vrednost čvora, predstavljenu poljem `Value`, kao i dva pokazivača, `Left` i `Right`. Važno je naglasiti da je ista struktura `Node` korišćena i za efemernu i za perzistentnu verziju stabla (kopiranjem putanja). Razlika se nalazi isključivo u funkcijama koje njima manipulišu. Sa druge strane, za potrebe metode debelih čvorova implementirana je posebna struktura `FatNode` koja umesto klasičnih pokazivača čuva nizove verzionisanih pokazivača.

Centralni deo implementacije perzistentnosti kopiranjem putanja predstavlja funkcija `InsertPersistent`, čiji je kod prikazan u Isečku 5.2.

Isečak koda 5.2: Implementacija perzistentnog umetanja kopiranjem putanja

```
func InsertPersistent(root *Node, val int) *Node {
        if root == nil {
                return &Node{Value: val}
        }

        if val < root.Value {
                newLeft := InsertPersistent(root.Left, val)
                if newLeft == root.Left {
                        return root
                }
                return &Node{
                        Value: root.Value,
                        Left:  newLeft,
                        Right: root.Right,
                }
        }

        if val > root.Value {
                newRight := InsertPersistent(root.Right, val)
                if newRight == root.Right {
                        return root
                }
                return &Node{
                        Value: root.Value,
                        Left:  root.Left,
                        Right: newRight,
```

```
                }
        }

        return root
}
```

Kada se nova vrednost umetne u odgovarajuće podstablo, funkcija ne menja postojeći čvor `root`. Umesto toga, pri povratku iz rekurzije kreira se novi čvor koji preuzima vrednost starog čvora, ali dobija ažurirani pokazivač ka promenjenom podstablu (linije 11-15). Desno podstablo, koje nije bilo obuhvaćeno izmenom, deli se sa prethodnom verzijom.

Poseban granični slučaj javlja se pri pokušaju umetanja vrednosti koja već postoji u stablu. U toj situaciji nije potrebno kreirati novu verziju. Implementacija to prepoznaje proverom pokazivača (`newLeft == root.Left`). Ova optimizacija sprečava nepotrebno kreiranje novih čvorova i smanjuje memorijske zahteve. Pored umetanja, ista tehnika primenjena je i prilikom brisanja elemenata kako bi perzistentnost ostala sačuvana. Pretraga elementa je operacija čitanja i ostaje identična efemernoj verziji.

Za potrebe uporedne analize u ovom radu, kao alternativna tehnika implementirana je i metoda „debelih" čvorova. Za razliku od tehnike kopiranja putanja koja koristi standardnu `Node` strukturu, metoda debelih čvorova zahteva drugačiju organizaciju u memoriji. Umesto čuvanja samo jednog važećeg pokazivača na levo i desno dete, debeli čvor mora da čuva čitav niz verzionisanih pokazivača. Ova struktura prikazana je u Isečku koda 5.3.

Isečak koda 5.3: Struktura debelog čvora sa istorijom pokazivača

```
type VersionedNode struct {
        Version int
        Node    *FatNode
}

type FatNode struct {
        Value int
        Left  []VersionedNode
        Right []VersionedNode
}
```

Kao što se može videti, svaka promena pokazivača u ovoj metodi ne zahteva kreiranje novog čvora, već se novi pokazivač, zajedno sa oznakom verzije (`Version`), samo dodaje na kraj postojećeg niza `Left` ili `Right`. Pretraga ovakvog stabla zahteva prolazak kroz niz kako bi se pronašao pokazivač koji odgovara traženoj verziji, što potvrđuje teorijske pretpostavke o uštedi memorije po cenu sporijeg čitanja.

5.3 Praktična primena: Sistem za poništavanje i ponovno izvršavanje (*Undo*/*Redo*)

Perzistentno binarno stablo pretrage omogućava izgradnju sistema u kome se prethodna stanja mogu jednostavno obnoviti posredstvom strukture `HistoryManager`. Njen zadatak jeste da čuva sve relevantne korenove perzistentnog stabla i da omogući kretanje kroz istoriju operacija u $O(1)$ vremenu. Kôd ove strukture prikazan je u Isečku koda 5.4.

Isečak koda 5.4: Struktura menadžera istorije i osnovne *Undo*/*Redo* operacije

```
type HistoryManager struct {
        history      []*Node
        currentIndex int
}

func NewHistoryManager(initialRoot *Node) *HistoryManager {
        return &HistoryManager{
                history:      []*Node{initialRoot},
                currentIndex: 0,
        }
}

func (h *HistoryManager) Undo() *Node {
        if len(h.history) == 0 {
                return nil
        }
        if h.currentIndex == 0 {
                return h.history[h.currentIndex]
        }
```

```
        h.currentIndex--
        return h.history[h.currentIndex]
}

func (h *HistoryManager) Redo() *Node {
        if len(h.history) == 0 {
                return nil
        }
        if h.currentIndex == len(h.history)-1 {
                return h.history[h.currentIndex]
        }
        h.currentIndex++
        return h.history[h.currentIndex]
}
```

Metode `Undo()` i `Redo()` isključivo menjaju celobrojni indeks unutar niza, bez ikakvog dubokog kopiranja stabla. Problem koji se mora adresirati javlja se kada korisnik nekoliko puta izvrši `Undo()`, a zatim doda novu vrednost. Zbog toga `AddVersion` odseca postojeću buduću istoriju i eksplicitno postavlja napuštene elemente na `nil`, čime sakupljač otpadaka dobija signal da oslobodi memoriju čvorova do kojih se više ne može doći.

5.4 Analiza performansi i evaluacija

U cilju komparativne evaluacije implementiranih struktura podataka napisana je skripta u jeziku Go, koja testira ponašanje efemernog stabla i dve vrste perzistentnih stabala – jedne zasnovane na kopiranju putanja i druge koja se zasniva na metodi debelih čvorova. Testiranje je sprovedeno za vrednosti N od 10.000 do 1.000.000 elemenata. Za svaku veličinu ulaza generisan je niz slučajnih vrednosti. Za svaku od struktura podataka mereno je vreme izvršavanja operacija umetanja elementa i pretrage. Što se operacije brisanja elementa tiče, izvršena su merenja u slučaju efemernog stabla i perzistentnog zasnovanog na kopiranju putanja. Brisanje elementa iz perzistentnog stabla koje koristi metodu debelih čvorova nije razmatrano jer se kod mutirajućih perzistentnih čvorova ono oslanja na logičko obeležavanje (tzv. *tombstone* tehnika), što osetno komplikuje kôd i izlazi iz okvira osnovne kom-

parativne demonstracije. Isečak koda 5.5 prikazuje srž ovog testnog okruženja uz odgovarajuće komentare preporučene u praksi.

Isečak koda 5.5: Testna skripta za merenje performansi uz eksplicitnu kontrolu GC-a

```
func benchmark() {
        // Forsiramo sakupljac otpadaka (GC) pre merenja
        // radi stabilne slike memorije
        runtime.GC()
        memBefore := readMem()

        // Umetanje uz belezenje u menadzer istorije
        start := time.Now()
        var root *Node
        hm := NewHistoryManager(root)
        for _, val := range values {
                root = InsertPersistent(root, val)
                hm.AddVersion(root)
        }
        duration := time.Since(start)

        // Merenje ukupne alocirane memorije nakon strukture
        runtime.GC()
        memAfter := readMem()

        // Sprecavanje Go kompajlera da optimizuje
        // i obrise neiskoriscene varijable (Dead Code Elimination)
        sinkNode = hm.Current()
}
```

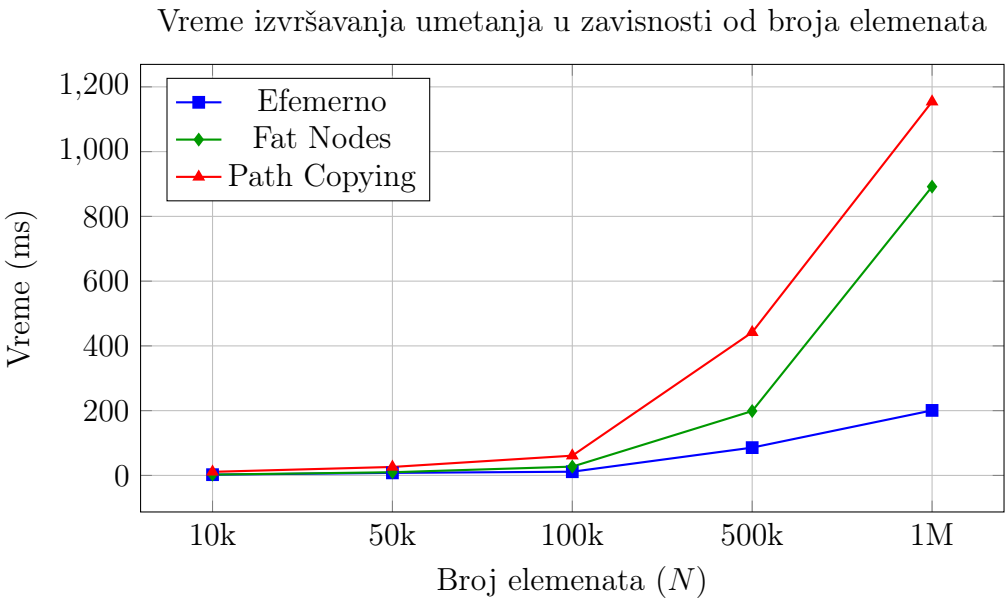
Sirovi ispis dobijen pokretanjem benchmark skripte u terminalu prikazan je u nastavku:

```
POČETAK BENCHMARK TESTIRANJA...
------------------------------------------------------------------------

>>> TEST ZA N = 10000 <<<
UMETANJE | Efemerno: 2.29 ms   | Path Copying: 11.03 ms   | Fat Nodes: 2.48 ms
PRETRAGA | Efemerno: 1.28 ms   | Path Copying: 1.46 ms    | Fat Nodes: 1.78 ms
```

```
BRISANJE | Efemerno: 1.23 ms   | Path Copying: 7.55 ms    | Fat Nodes: N/A
MEMORIJA | Efemerno: 0.22 MB   | Path Copying: 7.48 MB    | Fat Nodes: 0.72 MB

>>> TEST ZA N = 50000 <<<
UMETANJE | Efemerno: 7.83 ms   | Path Copying: 26.13 ms   | Fat Nodes: 9.77 ms
PRETRAGA | Efemerno: 5.95 ms   | Path Copying: 4.48 ms    | Fat Nodes: 7.67 ms
BRISANJE | Efemerno: 5.97 ms   | Path Copying: 22.00 ms   | Fat Nodes: N/A
MEMORIJA | Efemerno: 1.09 MB   | Path Copying: 42.63 MB   | Fat Nodes: 3.63 MB

>>> TEST ZA N = 100000 <<<
UMETANJE | Efemerno: 11.38 ms  | Path Copying: 60.96 ms   | Fat Nodes: 27.13 ms
PRETRAGA | Efemerno: 9.37 ms   | Path Copying: 14.20 ms   | Fat Nodes: 21.92 ms
BRISANJE | Efemerno: 10.60 ms  | Path Copying: 64.36 ms   | Fat Nodes: N/A
MEMORIJA | Efemerno: 2.18 MB   | Path Copying: 97.10 MB   | Fat Nodes: 7.26 MB

>>> TEST ZA N = 500000 <<<
UMETANJE | Efemerno: 85.82 ms  | Path Copying: 442.00 ms  | Fat Nodes: 198.75 ms
PRETRAGA | Efemerno: 70.58 ms  | Path Copying: 84.89 ms   | Fat Nodes: 185.30 ms
BRISANJE | Efemerno: 84.19 ms  | Path Copying: 415.61 ms  | Fat Nodes: N/A
MEMORIJA | Efemerno: 10.88 MB  | Path Copying: 542.21 MB  | Fat Nodes: 36.28 MB

>>> TEST ZA N = 1000000 <<<
UMETANJE | Efemerno: 200.76 ms | Path Copying: 1154.19 ms | Fat Nodes: 891.83 ms
PRETRAGA | Efemerno: 161.14 ms | Path Copying: 232.73 ms  | Fat Nodes: 844.03 ms
BRISANJE | Efemerno: 227.91 ms | Path Copying: 1040.74 ms | Fat Nodes: N/A
MEMORIJA | Efemerno: 21.78 MB  | Path Copying: 1107.56 MB | Fat Nodes: 72.61 MB

----------------------------------------------------------------------
Testiranje brzine Undo operacije za N = 1.000.000...
Vreme za povratak kroz 1.000.000 stanja (Undo): 548.41 µs
```

Grafik na Slici 5.2 vizuelno prikazuje poređenje vremena umetanja za tri posmatrane strukture. Efemerno stablo se pokazalo kao ubedljivo najbrže (oko 200 ms za milion elemenata) jer ažurira pokazivače u mestu. Metoda kopiranja putanja je najsporija (preko 1,15 sekundi) usled konstantnog alociranja novih čvorova na $O(\log n)$ dugačkoj putanji. Zanimljivo je primetiti da se umetanje kod debelih čvorova obavlja brže (oko 891 ms) nego kopiranje putanja. Razlog leži u činjenici da se čvorovi ne dupliraju u memoriji, već se u postojeći niz unutar samog čvora vrši dopisivanje (engl. *append*) novog zapisa pokazivača i odgovarajuće verzije.

Međutim, teorijski kompromisi obrađeni u Poglavlju 3 dolaze do punog izražaja kada se uporede vremena pretrage i zauzeće memorije. Pretraga kod efemernog

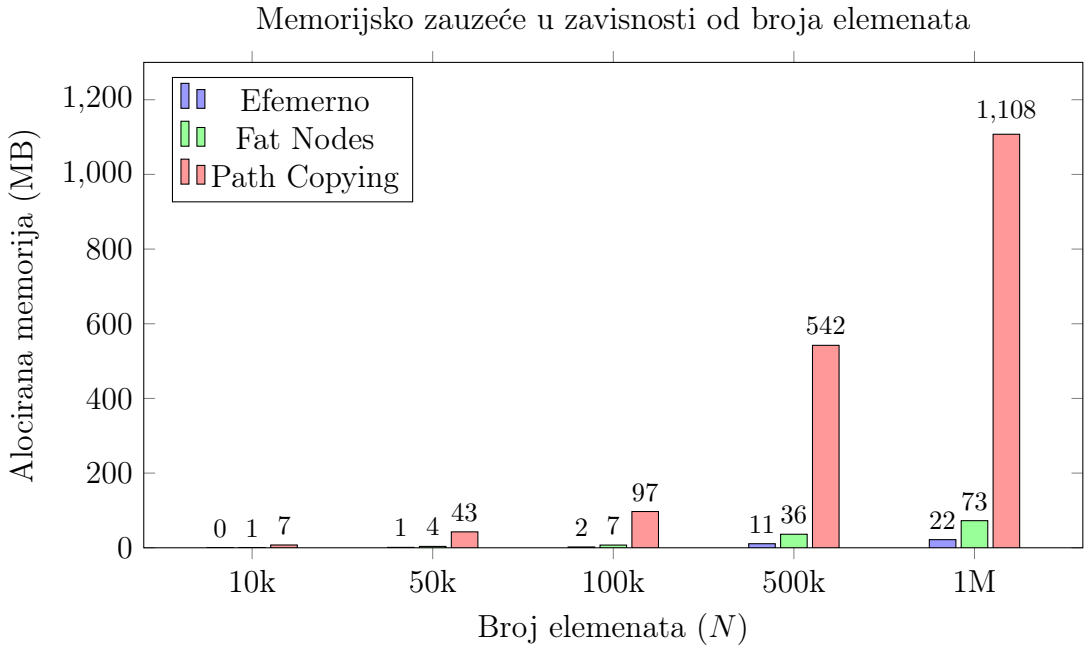


Slika 5.2: Linijski grafik rasta vremena izvršavanja operacija umetanja za tri različita tipa implementacije

stabla (161 ms) i kopiranja putanja (232 ms) postiže optimalnih $O(\log n)$ jer se kretanje vrši isključivo pratnjom direktnih pokazivača. Sa druge strane, pretraga kod debelih čvorova (844 ms) je skoro četiri puta sporija. Prilikom spuštanja kroz stablo, debeli čvor mora linearno ili binarno da pretraži svoj unutrašnji niz kako bi locirao validan pokazivač za traženu verziju, dodajući $O(\log m)$ usporavanje po čvoru.

Glavna prednost metode debelih čvorova uočava se na Slici 5.3. Prostorna složenost efemernog stabla iznosi $O(n)$ i za milion elemenata zauzima skromnih 21,78 MB. Kopiranje putanja stvara čitavu kolekciju stabala koja međusobno dele delove, generišući ukupno 1107,56 MB prostora (preko 1 GB). Međutim, metoda debelih čvorova zauzima neverovatno malo memorije za čuvanje čitave istorije — svega 72,61 MB. To neposredno dokazuje teorijsku hipotezu da se lokalnim čuvanjem izmena (bez formiranja novih $O(\log n)$ putanja) ostvaruje prostorna amortizacija perzistentnosti.

Konačno, najznačajniji rezultat u pogledu sistema za poništavanje i ponovno izvršavanje operacija ogleda se u brzini vraćanja stanja. U testu je izvršeno 1.000.000 uzastopnih vraćanja kroz istoriju koristeći menadžer istorije, a ukupno vreme izvršavanja iznosilo je svega 548,4 mikrosekundi (oko 0,5 nanosekundi po vraćanju). Razlog leži u samoj arhitekturi: vraćanje unazad ne zahteva kopiranje stabla već po-



Slika 5.3: Stubičasti grafik ukupne potrošnje memorije na kraju procesa dodavanja elemenata

meranje indeksa. Time se empirijski u potpunosti potvrđuje asimptotska složenost $O(1)$. Iako perzistentna struktura sa kopiranjem putanja zahteva veliku memoriju, ona zauzvrat nudi elegantan, nepogrešiv i trenutan povratak u bilo koju istorijsku tačku strukture.

Glava 6

Zaključak

Osnovni cilj ovog rada bio je istraživanje perzistentnih struktura podataka kao alternative klasičnim, efemernim strukturama, kod kojih se prethodno stanje gubi nakon svake izmene. Teorijski je dokazano da perzistentnost omogućava očuvanje istorije promena i brz pristup ranijim verzijama bez eksplicitnog kopiranja celokupnog sadržaja sistema. Rad je to i praktično demonstrirao uspostavljanjem perzistentnog binarnog stabla pretrage u programskom jeziku Go.

Kroz implementaciju sistema za poništavanje operacija, demonstriran je značaj strukturiranja zasnovanog na kopiranju putanja (engl. *path copying*). Paralelnom implementacijom i merenjem takozvane metode debelih čvorova (engl. *Fat Nodes*), u radu su uspešno potvrđeni i upoređeni teorijski kompromisi. Rezultati eksperimenta nedvosmisleno dokazuju da metoda kopiranja putanja drastično žrtvuje memoriju (preko 1 GB za milion elemenata) ali čuva brzu $O(\log n)$ pretragu karakterističnu za efemerna stabla, dok metoda debelih čvorova ekstremno štedi memorijski prostor (svega 72 MB) na račun usporavanja operacije navigacije i otežanog algoritma logičkog brisanja. Centralni zahtev *Undo* i *Redo* operacija izvršen je očekivanom vremenskom složenošću $O(1)$.

Ovakvi rezultati pokazuju da perzistentne strukture podataka prevazilaze puku akademsku teoriju. U kontekstu razvijenih baza podataka, naprednih editora koda, geometrijskih i geolokacijskih algoritama, kompromis trošenja veće količine memorije zarad potpunog pamćenja je itekako opravdan. Upravo tu se otvara prostor za dalja istraživanja. Budući razvoj mogao bi obuhvatiti implementaciju potpuno samobalansirajućih perzistentnih struktura (poput crveno-crnog stabla), kao i eksploataciju njihove najveće prednosti u savremenom inženjerstvu — činjenice da višenitni programi mogu potpuno bezbedno i bez brava (engl. *lock-free*) paralelno

čitati bilo koju zamrznutu verziju strukture.

Napomena o korišćenju AI alata: Tokom izrade ovog master rada, alati bazirani na veštačkoj inteligenciji korišćeni su kao pomoćno sredstvo u skladu sa savremenim akademskim praksama. Specifično, AI je povremeno konsultovan za korekciju stila i gramatike teksta, za tehničku pomoć u formatiranju LaTeX koda i vizuelizaciji grafika (upotreba *TikZ* paketa), kao i za sugestije prilikom izrade komparativne *Go* skripte za testiranje performansi. Sva rešenja, testovi i generisani tekstovi su kritički pregledani, korigovani i logički integrisani od strane autora, koji snosi potpunu odgovornost za konačnu tačnost, zaključke i originalnost prikazanog rada.

Bibliografija

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[2] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Boston, MA: Addison-Wesley Professional, 2015.

[3] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making Data Structures Persistent," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86–124, 1989.

[4] S. Karp, "A New Examination of Persistent Data Structures," 2015.

[5] C. Okasaki, *Purely Functional Data Structures*. Cambridge, U.K.: Cambridge University Press, 1998.

[6] N. Sarnak and R. E. Tarjan, "Planar Point Location Using Persistent Search Trees," *Communications of the ACM*, vol. 29, no. 7, pp. 669–679, 1986.