UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Jovan Dmitrović

IMPLEMENTACIJA PRILAGODLJIVOG RADIKS-STABLA U PROGRAMSKOM JEZIKU *RUST*

master rad

Beograd, 2022.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Vesna Marinković, docent
Univerzitet u Beogradu, Matematički fakultet

dr Nina Radojičić Matić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Majci, ocu i bratu

Naslov master rada: Implementacija prilagodljivog radiks-stabla u programskom jeziku *Rust*

Rezime: Radiks-stabla (engl. *radix tree*) su drvolika struktura koja pohranjuju parove ključ-vrednost. Prilagodljiva radiks-stabla (engl. *adaptive radix tree*) predstavljaju nadogradnju radiks-stabala, optimizujući potrošnju memorije. Programski jezik *Rust* je moderan i popularan jezik, ali za koji ne postoji potpuna podrška za rad sa prilagodljivim radiks-stablima. Cilj rada je implementacija prilagodljivih radiks-stabala u programskom jeziku *Rust*, koja podržava osnovne operacije: umetanje, čitanje, ažuriranje i brisanje. U radu je takođe prikazano testiranje implementirane strukture podataka, kao i poređenje performansi sa drugim strukturama podataka implementiranih u programskom jeziku *Rust*.

Ključne reči: programski jezici, *Rust*, stabla pretrage, prilagodljiva radiks-stabla, algoritmi i strukture podataka

Sadržaj

Glava 1

Uvod

Programski jezik *Rust* je moderan i izuzetno popularan programski jezik, čija prva verzija je objavljena 2015. godine. Odlike ovog jezika su robusnost i stabilnost. Najveći doprinos stabilnosti programskog jezika *Rust* daju njegovi jedinstveni koncepti vlasništva i pozajmljivanja, koji garantuju memorijsku bezbednost. Pojam vlasništva se odnosi na činjenicu da svaka vrednost ima jedinstvenu promenljivu koja poseduje tu vrednost, dok se pozajmljivanje vrednosti vrši putem referenci. Koncept vlasništva takođe omogućava odsustvo skupljača otpadaka, što doprinosi efikasnosti. Usled fokusa na stabilnosti i bezbednosti, programski jezik *Rust* se koristi u sistemima renomiranih kompanija kao što su *Mozilla*, *Google*, *Amazon* i *Microsoft*.

Stabla pretrage su bitan koncept u domenu struktura podataka. Njihova osobina da su uvek sortirana ih čini pogodnim za korišćenje u određenim sistemima. Jedna primena stabala pretrage su baze podataka, gde se stabla pretrage primenjuju kao mape, odnosno kao struktura podataka koja pohranjuje parove ključ-vrednost. Međutim, njihova efikasnost zavisi od količine unetih elemenata, što je neefikasno u poređenju sa nekim drugim strukturama podataka kao što su heš mape. Usled toga se često koriste radiks-stabla. Efikasnost operacija nad radiks-stablima ne zavisi od količine unetih elemenata, već samo od dužine ključa (prevedenog u binarni sistem) nad kojim se operacija izvodi. Problem kod radiks-stabala je sprega između potrošnje memorije i efikasnosti, koja zavisi od veličine unutrašnjih čvorova. Novo rešenje predstavljaju prilagodljiva radiks-stabla, koja prilagođavaju veličinu unutrašnjih čvorova po potrebi, time optimizujući potrošnju memorije.

U okviru programskog jezika *Rust* ne postoji potpuna podrška za prilagodljiva radiks-stabla. Osnovni cilj ovog rada je implementacija biblioteke *rustART*, koja

daje mogućnoćnost izvršavanja osnovih operacija nad prilagodljivim radiks-stablima: umetanja, ažuriranja, pretraživanja i brisanja.

U poglavlju 2 se nalaze istorijat, opis osnovnih karakteristika programskog jezika *Rust* kao i instrukcije za instalaciju kompilatora i drugih alata vezanih za *Rust*. Detaljniji opis radiks-stabala i prilagodljivih radiks-stabala je dat u poglavlju 3. Opis implementacije biblioteke *rustART*, kao i opis metodologije testiranja i merenja performansi osnovnih operacija, se nalazi u poglavlju 4. U zaključku rada, koji se nalazi u poglavlju 5, dat je kratki osvrt na programski jezik *Rust*, kao i na implementaciju biblioteke *rustART*.

Glava 2

Programski jezik *Rust*

Rust je statički tipiziran jezik fokusiran na bezbednost i performanse. Od svog nastanka, ovaj jezik je dobio veliku pažnju u svetu programiranja, čemu svedoči i činjenica da je *Rust* proglašen za „omiljeni programski jezik“ već petu godinu za redom u anketi koju je sprovela popularna veb-stranica *Stack Overflow* [24].

Danas se *Rust* koristi na velikom broju komercijalnih projekata. Na primer:

- U AWS servisima firme Amazon, poput *Lambda*, *EC2* i *Cloudfront* [3],
- U okviru operativnog sistema kompanije *Google ChromeOS* [10],
- U određenim komponentama *Microsoft* platforme *Azure*, uključujući i IoT sigurnosni servis *edgelet* [13],
- U registru *JavaScript* paketa *npm*, kod procedura koje prouzrokuju veliko opterećenje procesora [22],
- U veb-pretraživaču *Firefox* [11].

2.1 Razvoj jezika *Rust*

Programski jezik *Rust* je dizajnirao Grejdon Hor (engl. *Graydon Hoare*) koji je, u to vreme, bio zaposlen u kompaniji *Mozilla*. Hor je rad na ovom jeziku započeo 2006. godine kao svoj lični projekat, na kojem je samostalno radio naredne tri godine. Sredinom 2010. godine, u projekat se uključila i sama *Mozilla*, koja i danas sponzoriše njegov razvoj. Pored zaposlenih u kompaniji *Mozilla*, pošto je

u pitanju programski jezik otvorenog koda, svoj doprinos je dalo i preko 5000 dobrovoljaca [23].

Pre nego što se Mozila priključila projektu, *Rust* je izgledao dosta drugačije nego danas. U svojoj početnoj fazi, *Rust* je bio čist funkcionalni jezik, tj. nije imao bočne efekte; takođe, postojala je i analiza promene stanja (engl. *typestate analysis*), koja je omogućavala proveru operacija koje se mogu izvoditi nad specifičnim tipom podataka pri kompiliranju. Za razliku od ove dve osobine, neka dizajnerska rešenja su ostala do danas, kao što je imutabilnost i kontrola pristupa memoriji [15].

Nedugo nakon priključivanja kompaniji *Mozilla*, Grejdon Hor napušta projekat 2012. godine. U ovom periodu *Rust* dobija svoj menadžer paketa *Cargo* zajedno sa repozitorijumom paketa `crates.io`. Proces *RFC*, inspirisan procesom *PEP* programskog jezika *Python* [26], se osniva 2014. godine u svrhu strogog kontrolisanja novina u samom jeziku.

Prva verzija programskog jezika *Rust*, odnosno *Rust* 1.0, objavljena je 2015. godine [1]. U *Rust* zajednici je formiran model izbacivanja novih verzija svakih šest nedelja. To je dinamičniji pristup u odnosu na većinu programskih jezika gde je taj period minimalno godinu dana. Ovom odlukom se stavlja akcenat na stabilnost jezika time što će svaka nova verzija biti slična svom prethodniku, dok se kod jezika sa dugim periodom između verzija očekuju velike promene, što može da šteti kompatibilnosti.

U februaru 2021. godine ozvaničen je nastanak fondacije *Rust* (engl. *Rust Foundation*), neprofitne organizacije osnovane u svrhe daljeg razvoja programskog jezika Rust [27]. Pored Mozile, sponzori fondacije *Rust* su i kompanije *Google*, *AWS*, *Huawei*, *Facebook* i *Microsoft*.

2.2 Instalacija i korišćenje sistema *Cargo*

Ukoliko se zvanična veb-stranica programskog jezika *Rust* poseti koristeći mašinu koja ima *Windows* operativni sistem, biće ponuđene instalacione datoteke za 32-bitne i 64-bitne *Windows* sisteme.

Na *GNU/Linux* operativnim sistemima potrebno je uneti sledeću komandu u terminal:

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Potvrda da li se instalacija uspešno izvršila može se dobiti proverom verzije *Rust* kompilatora komandom:

```
rustc --version
```

Pored samog *Rust* kompilatora, u instalaciju su uključeni i sistem *Cargo* i alat *rustup*. Alat *rustup* daje mogućnost dobavljanja nove verzije *Rust* kompilatora i alata sa veba, kao i mogućnost deinstalacije komandama:

```
rustup update
rustup self uninstall
```

Pored toga što je menadžer paketa, *Cargo* vrši i automatizaciju prevođenja. Kreiranje novog projekta uz pomoć ovog sistema izvršava se komandom:

```
cargo new novi_projekat
```

Navedenom komandom se pravi novi direktorijum `novi_projekat` koji sadrži konfiguracionu datoteku `Cargo.toml` i direktorijum `src` koji treba da sadrži izvorne datoteke obeležene ekstenzijom *.rs* i u kojem se inicijalno nalazi datoteka *main.rs*. Takođe, sa novim direktorijumom se inicijalizuje i novi *Git* repozitorijum.

Generisana `Cargo.toml` datoteka je prikazana na listingu 2.1.

```
1 [package]
2 name = "novi_projekat"
3 version = "0.1.0"
4 authors = ["jovan <jdmitrovic@gmail.com>"]
5 edition = "2018"
6
7 [dependencies]
```

Listing 2.1: Inicijalna *Cargo.toml* datoteka

U prvoj liniji koda, `[package]` označava sekciju koja opisuje paket koji je napravljen. Informacije koje se ovde nalaze su dobijene iz promenljivih okruženja. Posle oznake `[dependencies]` se popisuju svi paketi koji su neophodni za rad sa novim paketom, tako da ih *Cargo* može dopremiti.

Na listingu 2.2 je prikazana `main.rs` datoteka. U njoj se, po osnovnim podešavanjima, nalazi *Hello World* program. Kao što se na listingu može videti, sintaksa programskog jezika *Rust* podseća na sintaksu programskog jezika *C*.

```
1 fn main() {
2     println!("Hello, world!");
```

```
3 }
```

Listing 2.2: Inicijalna *main.rs* datoteka

Cargo prevodi projekat komandom `cargo build`, a prevodi ga i pokreće `cargo run`. Korišćenjem komande `cargo check` može se proveriti da li se kôd kompilira, bez generisanja izvršne datoteke, što je korisno jer je ova opcija efikasnija od korišćenja pomenute `build` komande. Kompiliranjem projekta se pravi nova putanja `target/debug`, gde će se generisati izvršne datoteke.

Kompilator ima dva profila: `dev` profil, koji se koristi za prevođenje koda tokom razvoja, i `release` profil za prevođenje završne verzije programa, spremne za isporuku. Razlika između profila ogleda se u nivou optimizacije izvršnog koda: koristeći `dev` profil, kompilator će koristiti minimalan nivo optimizacije zarad bržeg prevođenja, dok se za `release` profil projekat prevodi sa maksimalnim nivoom optimizacije zarad dobijanja najboljih performansi programa. Podrazumevano ponašanje je da se koristi `dev` profil pokretanjem komande `cargo build`, dok se prevođenje u `release` profilu izvršava komandom:

```
cargo build --release
```

2.3 Osnovne karakteristike jezika

U nastavku će biti opisane bitne karakteristike programskog jezika *Rust*, koje nisu nužno u okviru jednog stila programiranja. Zastupljeno je nekoliko programskih paradigmi:

- Imperativna,
- Objektno-orijentisana, gde se umesto klasa koriste svojstva (engl. *traits*),
- Generička, u vidu generičkih tipova,
- Funkcionalna, u vidu iteratora i zatvorenja [17],
- Konkurentna [16].

Promenljive i konstante

Promenljive se mogu definisati korišćenjem ključne reči `let`, čime se podrazumeva da je takva promenljiva zapravo imutabilna, tj. ona se ne može menjati. Imu-

tabilnost promenljivih omogućava kompilatoru da prepozna razne vrste grešaka već u fazi kompilacije. Ipak, Rust dozvoljava i definisanje mutabilnih promenljivih korišćenjem ključnih reči `let mut`.

Pored promenljivih, *Rust* dozvoljava i definisanje konstanti upotrebom ključne reči `const`. Konstante se razlikuju od imutabilnih promenljivih po tome što se mogu definisati u bilo kom opsegu, uključujući i globalni, i po tome što konstante samo mogu imati vrednost konstantnog izraza, ali ne i vrednost izvršavanja funkcije.

U okviru jezika dozvoljeno je i tzv. sakrivanje (engl. *shadowing*). Sakrivanje je ponovno definisanje promenljivih. Za razliku od korišćenja ključne reči `mut`, koja omogućava promenu vrednosti promenljive, prilikom sakrivanja je moguće promeniti tip promenljive koja se ponovo definiše.

Tipovi

Programski jezik *Rust* je statički tipiziran jezik, ali ne zahteva pisanje tipa uz svaku promenljivu, osim ako je to neophodno. Primer je korišćenje funkcije `parse`, date na listingu 2.3, koja prevodi nisku u odgovarajuću brojčanu vrednost. Budući da funkcija `parse` može vratiti i nepostojeću vrednost kada niska nije napisana u odgovarajućem formatu, ulančava se i funkcija `expect` koja prinudno zaustavlja program i ispisuje prosleđenu poruku.

```
1 fn main() {
2     let broj: i32 = "10".parse().expect("Nije unet broj!");
3     println!("Unet broj je: {}", broj);
4 }
```

Listing 2.3: Korišćenje funkcije `parse`

Ukoliko se pokuša prevođenje koda sa listinga 2.3, *Cargo* će pokazati grešku:

```
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let num = "10".parse().expect("Nije unet broj!");
  |         ^^^ consider giving `num` a type
```

Ova greška se javlja zbog toga što funkcija `parse` prima generičke parametre, te je kompilatoru neophodna informacija kojeg je tipa promenljiva `num`.

Rust sadrži četiri vrste prostih tipova: cele brojeve, brojeve zapisane u pokretnom zarezu, karaktere i Bulove konstante *true* i *false*. Celi brojevi mogu biti označeni ili neoznačeni. Označeni brojevi su predstavljeni tipovima `i8`, `i16`, `i32`, `i64` i `i128`, gde broj posle karaktera `i` predstavlja veličinu tipa u bitovima. Neoznačeni brojevi su predstavljeni analogno označenim, s tim da oni počinju karakterom `u`. Postoje i tipovi `isize` i `usize` čija veličina zavisi od arhitekture. Analogon tipovima `float` i `double` iz programskog jezika *C* su tipovi `f32` i `f64`. Tip `char` je veličine četiri bajta, gde su karakteri predstavljeni *Unicode* vrednostima.

Osnovni složeni tipovi u programskom jeziku *Rust* su torke i nizovi. Torke se predstavljaju navođenjem liste tipova, razdvojene zarazima u okviru zagrada. Primer sa listinga 2.4 ilustruje upotrebu torki:

```
fn main() {
    let koordinate: (i32, f32, i32) = (1, 2.0, 3);
    let (x, y, z) = koordinate;

    println!("x: {}, y: {}", x, koordinate.1);
}
```

Listing 2.4: Inicijalizacija i upotreba torki

Torke mogu sadržati vrednosti različitog tipa, a može im se pristupiti ili korišćenjem novih promenljivih, ili korišćenjem `.i` sintakse, čime se pristupa elementu na i-toj poziciji, gde se elementi torke broje počevši od 0.

Nizovi predstavljaju kolekciju vrednosti istog tipa i, poput torki, fiksne su veličine. Obeležavaju se uglastim zagradama, unutra kojih su elementi razdvojeni zarezima.

Niske se u programskom jeziku *Rust* javljaju u dva oblika: u obliku literala i u obliku niske promenljive dužine (u programskom jeziku *Rust* se one zovu *string*, odnosno *String*). Literalima se veličina zna pre kompilacije i njihov sadržaj se ne može promeniti, pa se one mogu čuvati na steku, dok se niske promenljive dužine moraju čuvati na hipu.

Funkcije

Funkcije se definišu ključnom reči `fn`, navođenjem imena funkcije za kojim sledi lista parametara i njihovih tipova u zagradama, razdvojenih zarezima. Nakon liste parametara, moguće je napisati tip povratne vrednosti posle oznake `->`. Ukoliko

povratni tip izostane, kompilator će ga automatski izvesti. Na kraju se nalazi telo funkcije između vitičastih zagrada. Primer funkcije koja izračunava zbir kvadrata dva broja je dat u okviru listinga 2.5.

```
fn zbir_kvadrata(x: i32, y: i32) -> i32 {
  x*x + y*y
}
```

Listing 2.5: Funkcija koja izračunava zbir kvadrata

Pored standardnog korišćenja ključne reči `return` kao oznake za povratnu vrednost funkcije, funkcija će vratiti vrednost poslednje naredbe koja se ne završava karakterom `;`.

Kontrola toka

Rust podržava klasične imperativne načine kontrole toka sa `if`, `while` i `for` komandama. Postoji par dodatnih mogućosti koje *Rust* takođe nudi:

- Ukoliko je potrebno napisati beskonačnu petlju, nije neophodno koristiti `while` naredbu sa uvek tačnim uslovom, već se može koristiti ključna reč `loop`, kao u listingu 2.6, gde se *loop* koristi u svrhe nalaženja najvećeg zajedničkog delioca dva broja.
- I `while` i `for` se mogu naći u okviru `let` naredbe, s tim da se u telu petlje mora naći `break` naredba uz koju se dopisuje povratna vrednosti kao u primeru 2.6, analogno korišćenju `return` naredbe u funkcijama.
- Iteriranje kroz kolekciju pomoću `for` petlje se može vršiti ili preko indeksa trenutnog elementa, ili pomoću `iter` funkcije. U listingu 2.7, u `for` petlji se nad vektorom `a` pravi iterator funkcijom `iter`, na koji se primenjuje funkcija `rev` koja menja smer čitanja elemenata iteratora. Brojevi koji će biti ispisani na standardni izlaz su, redom, `3`, `2` i `1`.

```
fn main() {
    let mut a: u32 = 65342;
    let mut b: u32 = 23456;

    let gcd = loop {
        if b == 0 || a == b {
```

```
            break a;
        }

        if a == 0 {
            break b;
        }

        if a > b {
            a -= b;
        } else {
            b -= a;
        }
    };

    println!("Najveci zajednicki delilac a i b je {}", gcd);
}
```

Listing 2.6: Korišćenje naredbe *loop* i petlje u okviru *let* naredbe

```
fn main() {
    let a = [1, 2, 3];

    for elem in a.iter().rev() {
        println!("{}", elem);
    }
}
```

Listing 2.7: Korišćenje `iter` i `rev` funkcija

Vlasništvo

Koncept **vlasništva** je novina u odnosu na druge programske jezike. Ova osobina dozvoljava programskom jeziku *Rust* da funkcioniše bez sakupljača otpadaka. Za razliku od sakupljača otpadaka, sistem vlasništva ne utiče ni na koji način na izvršavanje programa, jer se postupanje prema skupu pravila vlasništva proverava prilikom prevođenja.

Sistem vlasništva se može svesti na tri pravila:

1. Svaka vrednost u programskom jeziku *Rust* ima promenljivu koja je poseduje.
2. U jednom trenutku, za svaku vrednost, postoji tačno jedan vlasnik.
3. Kada promenljiva završi svoj životni vek, tada se vrednost koju ta promenljiva poseduje automatski briše iz memorije.

Primer funkcionisanja ovog sistema se može dati uz pomoć pomenutih niski promenljive dužine, odnosno tipa `String`: ako se, kao u listingu 2.8, pokuša ulančavanje pokazivača na istu vrednost, *Rust* kompilator će prikazati sledeću grešku:

```
error[E0382]: borrow of moved value: `niska1`
 --> src/main.rs:6:32
  |
2 |     let niska1 = String::from("Hello World!");
  |         ------ move occurs because `niska1` has type `String`, which
does not implement the `Copy` trait
3 |     let niska2 = niska1;
  |                  ------ value moved here
...
6 |     println!("Prva niska: {}", niska1);
  |                                ^^^^^^ value borrowed here after move
```

Kompilator ukazuje na promenu poseda vrednosti na koju pokazuje `niska1`, čime promenljiva `niska1` gubi mogućnost manipulisanja vrednošću nad kojom je imala vlasništvo, time poštujući pravilo da vrednost pripada samo jednoj promenljivoj.

```
1 fn main() {
2     let niska1 = String::from("Hello World!");
3     let niska2 = niska1;
4
5     println!("Druga niska: {}", niska2);
6     println!("Prva niska: {}", niska1);
7 }
```

Listing 2.8: Prenos vlasništva između promenljivih

Može se uočiti da je kopiranje promenljive `niska1` plitko. Duboko kopiranje promenljivih se odvija pomoću metoda `clone` koja je implementirana za tip `String`.

Kada se vrednost promenljive šalje kao parametar funkcije, vlasništvo te vrednosti prelazi u posed te funkcije, odnosno, promenljiva iz pozivaoca se ne može koristiti nakon pozivanja funkcije. Međutim, pozvana funkcija može *vratiti* vlasništvo nad promenljivom tako što iskoristi tu promenljivu kao povratnu vrednost: time se vlasništvo te promenljive vraća pozivaocu te funkcije. Promenljiva koja je vraćena može biti bilo koja promenljiva nad kojom ta funkcija ima vlasništvo, uključujući i promenljive koje ne potiču odatle (dakle, i sami parametri funkcije).

Pozajmljivanje

Proces davanja i preuzimanja vlasništva može da bude nepogodan u nekim slučajevima; zbog toga, postoji i opcija **pozajmljivanja** (engl. *borrowing*). Pozajmljivanje omogućava funkcijama da pozajme vrednosti promenljivih, koje se moraju vratiti vlasniku po završetku izvršavanja te funkcije, što znači da se pozajmljene promenljive mogu kasnije koristiti. Promenljive se pozajmljuju koristeći simbol `&`, što je ilustrovano na listingu 2.9, gde će izvršavanje koda ispisati: `Niska „Hello world" ima 2 reci`.

```
fn main() {
    let s = String::from("Hello world");

    println!("Niska \"{}\" ima {} reci", s, broj_reci(&s));
}

fn broj_reci(s: &String) -> usize {
    s.matches(" ").count() + 1
}
```

Listing 2.9: Pozajmljivanje vrednosti promenljivih

Na sličan način se definišu i reference na promenljive. Vrednosti pozajmljenih promenljivih se ne mogu menjati preko referenci definisanih na ovaj način; u te svrhe se koriste mutabilne reference koje se obeležavaju sa `&mut`. Za razliku od referenci u drugim programskim jezicima, može postojati samo jedna mutabilna referenca za jednu promenljivu u istom dosegu u kojem ne sme postojati ni imutabilna referenca te promenljive.

U programskom jeziku *Rust*je nemoguće napraviti „zalutalu“ referencu (engl. *dangling reference*), tj. referencu koja referiše na prostor u memoriji koji je već

oslobođen. Prilikom prevođenja koda, kompilator će dati preporuku za korišćenje statičke reference uz pomoć sintakse `&'static`. Ukoliko bi se pokušalo prevođenje koda iz listinga 2.10, kompilator ne bi dozvolio prevođenje bez `static` oznaka. Konvencija je da imena promenljivih sa statičkim životnim vekom počinju velikim početnim slovom.

```
fn main() {
   println!("{}", num());
}

fn num() -> &'static u32 {
    static N: u32 = 1234;
    return &N;
}
```

Listing 2.10: Korišćenje promenljivih sa statičkim životnim vekom

Strukture

Definisanje struktura se vrši na uobičajen način, ali *Rust* nudi mogućnosti koje skraćuju sintaksu, kao u listingu 2.11, gde je dat primer stukture podataka koja modeluje studente.

```
struct Student {
    ime: String,
    indeks: String,
    email: String
}

fn kreiraj_studenta(ime: String, indeks: String) -> Student {
    Student {
        ime,
        email: format!("{}@matf.bg.ac.rs", indeks),
        indeks
    }
}

fn main() {
    let student1 = kreiraj_studenta(
```

```
        String::from("Ana"),
        String::from("mi15123")
    );
    let student2 = Student {
        indeks: String::from("mr14321"),
        ..student1
    };
}
```

Listing 2.11: Kreiranje novih struktura

Sa listinga 2.11 se može primetiti kako se uz pomoć oznake `..` mogu definisati nove instance strukture gde se imena polja poklapaju sa imenima parametara funkcije. U istom primeru će instanca `student2` imati isto ime kao i instanca `student1`, ali će polje `indeks` imati drugačiju vrednost.

Strukture se mogu i definisati putem torki, s tim da njihova polja neće imati imena, već će im se pristupati `.` notacijom. Strukture definisane na ovaj način ne moraju nužno imati polja, već im se struktura i ponašanje može postaviti putem interfejsa.

Da bi se implementirali metodi u okviru struktura, koristi se ključna reč `impl` kao u listingu 2.12, gde se menja e-mail adresa strukture `Student` iz listinga 2.11.

```
impl Student {
    fn promena_mejla(&mut self, novi_email: String) {
        self.email = novi_email;
    }
}
```

Listing 2.12: Definisanje metoda

Mogu se implementirati i metodi koji ne uzimaju instancu kao argument; njima se pristupa pomoću `::` notacije.

Enumeratori

Enumeratori u programskom jeziku *Rust* imaju dodatne mogućnosti u odnosu na enumeratore u drugim programskim jezicima: svaki „tip“ u enumeratoru može se posmatrati kao torka vrednosti. Ovime se, na primer, postiže ponašanje enumeratora `Option` koji premošćava izostanak vrednosti `null`. `Option` je definisan na listingu 2.13.

```
enum Option<T> {
    Some(T),
    None
}
```

Listing 2.13: Definisanje enumeratora

Ono što `Option` nudi jeste mogućnost izostanka neke vrednosti, ali da taj izostanak ne može uticati na integritet programa. Pre nego što se koristi, vrednost tipa `Option` se mora otpakovati (engl. *unwrap*), odnosno, mora se utvrditi da li je vrednost generičkog tipa `T` izostala. Otpakivanje se vrši `match` naredbom kao u listingu 2.14.

```
fn main() {
    let a: Option<i8> = Some(10);

    match a {
        Some(x) => println!("Vrednost promenljive a je: {}", x),
        None => println!("Promenljiva a nema vrednost!"),
    }
}
```

Listing 2.14: Korišćenje `match` naredbe

Ukoliko se promenljiva ne otpakuje, kompilator će ispisati odgovarajuću grešku. Slično, greška će se ispisati i kada naredba `match` ne pokrije sve moguće vrste enumeratora, gde je moguće koristiti `_` notaciju za obeležavanje podrazumevane grane.

Na listingu 2.15 je dat primer korišćenja podrazumevane grane. Enumerator `Boja` podržava tri različita modela boja: RGB, CYMK i HSV. Naredba `match` sa `_` granom omogućava ispisivanje vrednosti komponenti boja koje su pohranjene isključivo u RGB modelu.

```
enum Boja {
    RGB(u32, u32, u32),
    CYMK(u32, u32, u32, u32),
    HSV(u32, u32, u32),
}

fn main() {
    let b = Boja::RGB(15, 233, 121);
```

```
9
10      match b {
11          Boja::RGB(r, g, b) => {
12              println!("Crvena: {}, Zelena: {}, Plava: {}", &r, &g, &b);
13          }
14          _ => println!("Nije uneta boja u RGB modelu!"),
15      }
16  }
```

Listing 2.15: Korišćenje `match` naredbe sa podrazumevanom granom

Strukture podataka promenljivog sadržaja

Pored torki i nizova, *Rust* ima podršku za strukture podataka kojima se sadržaj može menjati: to su vektori i heš mape.

Vektori

Vektori su predstavljeni tipom `Vec<T>`, gde je `T` generički tip podataka sadržanih u vektoru. Novi vektor se inicijalizuje funkcijom `Vec::new` ili korišćenjem naredbe `vec!` kao u listingu 2.16. U istom listingu su prikazane funkcije dodavanja odnosno brisanja elemenata u vektoru: `push`, odnosno `remove`. Poziv funkcije `remove(i)` briše element koji se nalazi na poziciji i u vektoru. Da bi elementi vektora mogli menjati njihovu unutrašnju strukturu, mora se dodati ključna reč `mut` prilikom inicijalizacije. Vektorima koji su definisani bez te ključne reči se i dalje mogu dodavati ili brisati elementi, ali naknadne promene tih elemenata nisu dozvoljene.

Pravila vlasništva su i dalje na snazi, pa se prilikom iteriranja kroz petlju koriste reference na elemente vektora. Da bi se te reference mogle menjati, prvo ih je potrebno dereferencirati operatorom `*`.

```
fn main() {
    let mut v: Vec<i32> = vec![4, 77, 10, 923];
    v.push(50);
    v.push(33);

    v.remove(2);

```

```
8       for elem in &mut v {
9           *elem *= 10;
10          print!("{} ", elem);
11      }
12  }
```

Listing 2.16: Inicijalizacija, dodavanje i brisanje elemenata iz vektora

Kao i u nizu, u vektoru se moraju naći elementi istog tipa. Međutim, ovo ograničenje se može prevazići korišćenjem enumeratora, pošto je kompilatoru neophodno samo da ima informaciju koji se tipovi mogu naći u vektoru da bi se mogla rezervisati adekvatna količina memorije.

Heš mape

Za razliku od vektora, deo standardne biblioteke koja implementira heš mape se mora ručno uključiti u program naredbom `use`. Funkcija `insert` će uneti odgovarajući par u mapu, ali će postaviti novu vrednost, bez obzira da li se ključ već nalazi u mapi.

Korišćenjem funkcije `entry`, može se dobiti enumerator `Entry` koji ukazuje da li vrednost za dati ključ postoji u mapi. Ako se na `Entry` nadoveže funkcija `or_insert`, ona će uneti odgovarajući par u mapu samo ako taj ključ ne postoji u mapi. Nakon toga, funkcija `or_insert` će vratiti referencu na par koji je napravljen, ili koji je nađen u mapi, kao u listingu 2.17. U istom listingu se koristi naredba `println!` sa argumentom `{:?}`, koja ispisuje odgovarajuće informacije za debagovanje; u ovom slučaju, biće ispisano `{„Pas": 7, „Macka": 3}`.

```
fn main() {
    use std::collections::HashMap;

    let mut m = HashMap::new();

    m.insert(String::from("Pas"), 2);
    m.insert(String::from("Macka"), 3);

    let p = m.entry(String::from("Pas")).or_insert(4);
    *p += 5;

    println!("{:?}", m);
```

```
}
```

Listing 2.17: Inicijalizacija heš mape i korišćenje `insert` i `or_insert` funkcija

Generički tipovi i interfejsi

U sekciji 2.3, pomenut je enumerator `Option` koji predstavlja omotač za vrednosti bilo kog tipa, da bi se izbeglo korišćenje nedostajućih vrednosti. Da bi se `Option` mogao koristiti na taj način, moraju se koristiti generički tipovi, koji se navode između znakova `<` i `>`, kao u definiciji enumeratora `Option` u listingu 2.13.

Prilikom definisanja metoda u okviru strukture podataka, kada struktura koristi jedan ili više generičkih tipova, moguće je implementirati metod samo za jedan tip, kao u listingu 2.18, gde se rastojanje između tačaka može izračunati kada su koordinate zapisane u pokretnom zarezu.

```
struct Tacka<T> {
    x: T,
    y: T
}

impl Tacka<f32> {
    fn dist(&self, t: &Tacka<f32>) -> f32 {
        ((self.x - t.x).powi(2) + (self.y - t.y).powi(2)).sqrt()
    }
}

fn main() {
    let t1 = Tacka {
        x: 32.0,
        y: 40.0
    };
    let t2 = Tacka {
        x: 13.0,
        y: 51.0
    };

    println!("Rastojanje izmedju tacaka je: {}", t1.dist(&t2));
```

```
23 }
```

Listing 2.18: Korišćenje generičkog tipa prilikom definisanja strukture

Korišćenje generičkih tipova ne prouzrokuje pad performansi, zato što će se prilikom prevođenja koda, kao i u programskom jeziku *C++* izvršiti *monomorfizacija*. Monomorfizacija podrazumeva pretvaranje generičkih tipova u konkretne tipove koji se koriste u programu, time garantujući da će prevedeni kôd biti podjednako efikasan kao i kôd koji ne koristi generičke tipove.

Interfejsi (engl. *traits*) predstavljaju način definisanja zajedničkog ponašanja struktura i enumeratora. U listingu 2.19 je definisan novi interfejs `Objekat2D` koji obavezuje objekte da sadrže implementaciju funkcije `povrsina`.

```
1  use std::f32::consts::PI;
2
3  trait Objekat2D {
4      fn povrsina(&self) -> f32;
5  }
6
7  struct Krug {
8      r: f32,
9  }
10
11 impl Objekat2D for Krug {
12     fn povrsina(&self) -> f32 {
13         self.r * self.r * PI
14     }
15 }
```

Listing 2.19: Definisanje novog interfejsa

U okviru definicije interfejsa se može naći i podrazumevana implementacija metoda. U tom slučaju se metode ne moraju implementirati za pojedinačne strukture podataka.

Generički tipovi i interfejsi se mogu koristiti zajedno tako da se obezbedi ograničenje konkretnih tipova koji se koriste u definiciji funkcija ili struktura. U listingu 2.20 je definisana funkcija koja kao argument uzima samo tipove koji implementiraju interfejs `Objekat2D`.

```
1  fn povrsina_objekta<T: Objekat2D>(objekat: &T) {
2      println!("Povrsina objekta je: {}", objekat.povrsina());
```

```
}
```

Listing 2.20: Definisanje funkcije sa ograničenjem generičkih parametara

Za definisanje ograničenja generičkih tipova se može koristiti sintaksička pomoć u vidu ključne reči `where`. U listingu 2.21 je prepravljena funkcija iz listinga 2.20 tako da generički tip mora implementirati interfejs `Display`, čime se obezbeđuje mogućnost korišćenja parametra funkcije u okviru naredbe `println!`. Bitno je napomenuti da se više različitih interfejsa ograničenja deklariše oznakom `+`.

```
fn povrsina_objekta<T>(objekat: &T)
    where T: Objekat2D + Display
{
    println!("Povrsina objekta {} je: {}", objekat, objekat.povrsina());
}
```

Listing 2.21: Definisanje funkcije sa ograničenjem višestrukih generičkih parametara

Životni vek referenci

Svaka referenca u programskom jeziku *Rust* ima svoj životni vek, čak iako on nije eksplicitno naveden. U sekciji 2.3 su pomenute zalutale reference, odnosno reference koje referišu na već oslobođeni prostor. U istoj sekciji je predstavljen i jedan od načina na koji se problem zalutalih referenci može zaobići: korišćenjem statičkog životnog veka. Drugi način rešavanja ovog problema jeste korišćenjem eksplicitno navedenih životnih vekova. Životni vekovi referenci su implementirani u vidu generičkih parametara koji se obeležavaju sa apostrofom i identifikatorom životnog veka posle znaka `&` i pre tipa reference. Prilikom definisanja funkcije se navode između karaktera `<` i `>` zajedno sa generičkim tipovima. U listingu 2.22 je eksplicitno naveden životni vek referenci `s1` i `s2`. Kako je životni vek ove dve reference obeležen istim identifikatorom `a`, funkcija `duza_rec` očekuje parametre sa istim životnim vekom. Slično kao i kod generičkih tipova, identifikator `a` se ne odnosi na konkretan životni vek.

```
fn duza_rec<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    let w1 = s1.split_whitespace()
               .next()
               .unwrap_or("");

```

```
    let w2 = s2.split_whitespace()
               .next()
               .unwrap_or("");

    if w1.len() > w2.len() {
        w1
    } else {
        w2
    }
}

fn main() {
    let s1 = String::from("Prva recenica");
    let s2 = String::from("Druga recenica");

    println!("Duza rec je: {}", duza_rec(&s1, &s2));
}
```

Listing 2.22: Definisanje funkcije čiji parametri imaju eskplicitno navedene životne vekove

Nije uvek neophodno eksplicitno deklarisati životni vek svake reference prilikom definisanja funkcije, zato što postoje **pravila spajanja životnih vekova** (engl. *lifetime elision rules*). U pravila spajanja se ubrajaju tri pravila po kojima kompilator automatski određuje životni vek reference:

1. Svaki parametar funkcije koji je referenca dobija jedinstveni identifikator životnog veka.

2. Ako postoji samo jedan parametar funkcije sa životnim vekom i povratna vrednost je takođe referenca, onda će i povratna vrednost imati isti životni vek kao i taj parametar.

3. Ako postoji više parametara funkcije koji su reference, ali jedan od njih je `&self` ili `&mut self` sa životnim vekom `'a`, onda će povratna vrednost, ukoliko je ona referenca, imati životni vek `'a`.

U listingu 2.22 funkcija `duza_rec` ima navedene životne vekove jer kompilator nije u mogućnosti da izvede sve životne vekove samo iz pravila spajanja. Da životni

vekovi nisu eksplicitno napisani, oba parametra funkcije bi imala jedinstveni životni vek, a životni vek povratne vrednosti bi ostao nepoznat jer se nijedno pravilo spajanja ne odnosi na njega.

Glava 3

Prilagodljiva radiks-stabla

Razvoj računarske memorije je doprineo da se baze podataka mogu u potpunosti smestiti u radnu sistemsku memoriju. Ovakve baze podataka se nazivaju **memorisane baze podataka** (engl. *in-memory databases*), a njihovi predstavnici su *Apache Ignite* [2], *VoltDB* [25] i *Redis* [21]. Korišćenjem radne umesto sekundarne memorije dobijaju se značajna poboljšanja u vidu smanjenja odziva, jer sistemska memorija ima za šest redova veličine brže vreme odziva od sekundarne memorije bazirane na magnetnim diskovima.

Za memorisane baze podataka je neophodna struktura podataka koja kodira parove ključ-vrednost tako da se za odgovarajući ključ može naći vrednost koja mu je pridružena, gde i ključevi i vrednosti mogu biti proizvoljnog tipa. Takođe, ova struktura bi trebalo da podrži upite opsega (engl. *range queries*), tj. upite kod kojih se dohvata više podataka čiji se ključevi nalaze u nekom datom opsegu. Na primer, taj opseg mogu biti niske koje počinju sa karakterima između `A` i `D`. Glavni cilj ove strukture podataka je da koristi procesor što je efikasnije moguće, jer vreme odziva više nije problem.

Strukture koje se tradicionalno koriste u bazama podataka jesu drvolike strukture podataka. Međutim, problem je u njihovoj neefikasnosti, jer funkcije dodavanja i brisanja elemenata u najboljem slučaju imaju asimptotsko vreme izvršavanja $\mathcal{O}(\log n)$, gde je n ukupan broj unetih elemenata. U ove svrhe su stvorene nadograđene varijante stabala, kao što je struktura podataka FAST (engl. ***F**ast **A**rchitecture **S**ensitive **T**ree*) [14] koja se oslanja na hardverski implemententirane SIMD instrukcije [12], uz pomoć kojih je moguće izvršiti paralelne operacije nad podacima koji su sekvencionalno smešteni u memoriji. FAST je efikasniji prilikom pretrage unetih podataka, ali gubi na efikasnosti umetanja i brisanja elemenata.

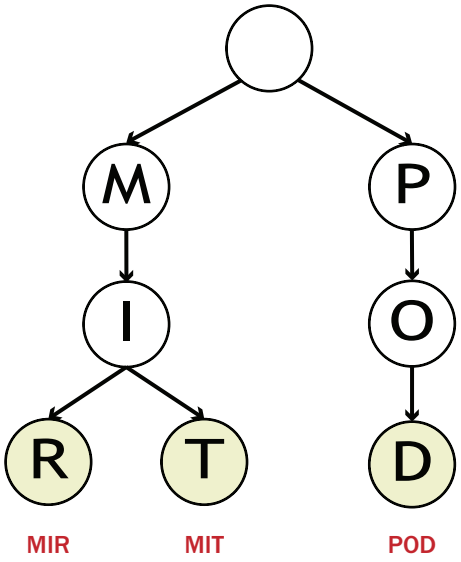
Pored drvolikih struktura, u ove svrhe se koriste i heš mape. Nasuprot stablima, heš mape su efikasne za sve osnovne operacije (dodavanje, brisanje, pretraživanje), sa prosečnom vremenskom složenošću $\mathcal{O}(1)$. Uprkos efikasnosti heš mapa, one nisu dobro rešenje za baze podataka zbog upita opsega. U heš mapama gde se ključevi heširaju i gde operacija inverzna operaciji heširanja ne postoji, nije moguće dohvatiti elemente čiji ključ nije u potpunosti poznat bez prolaska kroz sve unete parove. Takođe, povećavanje kapaciteta heš mape je skupo i podrazumeva reorganizaciju čitave mape, sa linearnom vremenskom složenošću.

Da bi se ispunili zahtevi memorisanih baza podataka koriste se tzv. **radiks--stabla** (engl. *radix tree*), odnosno **PATRICIA stabla** [19], kako su prvobitno bila nazvana. Radiks-stablo je drvolika struktura koja koristi binarnu reprezentaciju ključa. Svaki čvor stabla predstavlja jedan deo ključa, a neki čvorovi sadrže i vrednosti. Posmatrajući put od korena do čvora koji sadrži vrednost dobija se ključ koji je uparen sa tom vrednošću.

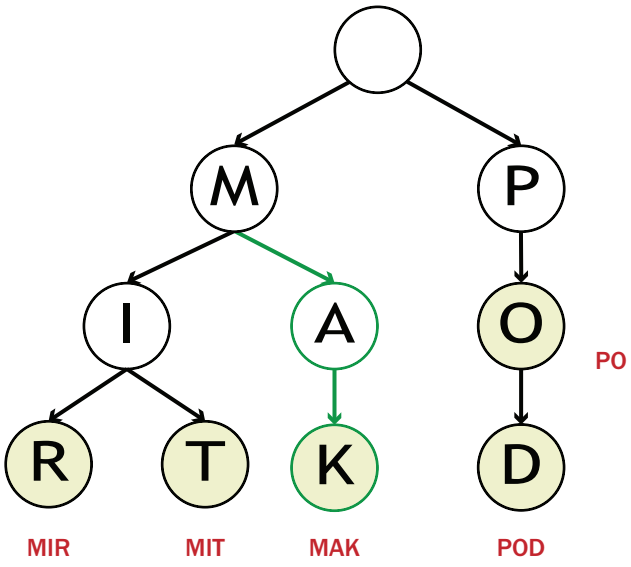
Na slici 3.1 predstavljeno je radiks-stablo čiji čvorovi predstavljaju po jedan bajt ključa i gde su uneti parovi sa ključevima `MIR`, `MIT` i `POD` u UTF-8 formatu [7]. Kako se svaki karakter engleskog alfabeta u UTF-8 formatu predstavlja jednim bajtom, svakom čvoru u stablu odgovara jedan karakter ključa. Svaki unutrašnji čvor u stablu izuzev korena može sadržati i vrednost. S druge strane, svaki list mora sadržati vrednost, u suprotnom je suvišan. Na primeru su čvorovi koji sadrže vrednosti obeleženi žutom bojom, a kako različite vrednosti u čvorovima ne utiču na strukturu radiks-stabla, vrednosti nisu prikazane na slici.

Ukoliko bi se u primeru na slici 3.1 dodao par sa ključem `PO`, stablo ne bi menjalo svoju strukturu, već bi se samo čvoru `O` pridružila nedostajuća vrednost. Pri dodavanju para sa ključem `MAK`, pretraga bi prilikom prolaska kroz stablo došla do čvora `M` i tu bi stala, pošto ne postoji dete `A`. Tada se dodaje novo podstablo čvoru `M`, gde se dobija novo stablo kao na slici 3.2.

Grupisanje sličnih ključeva u isto podstablo rezultuje vremenskom složenošću čitanja, umetanja i brisanja elemenata koja ne zavisi od broja elemenata unetih u strukturu niti od veličine same strukture, već od dužine ključa u njegovoj binarnoj reprezenaciji. Iako je vremenska složenost linearna u odnosu na dužinu ključa, osobina da efikasnost ne zavisi od broja unetih elemenata je pogodna za primenu u memorisanim bazama podataka. Prednost radiks-stabala je i u činjenici da ona, za razliku od stabala pretrage, ne zahtevaju rebalansiranje zarad održavanja visine stabla.



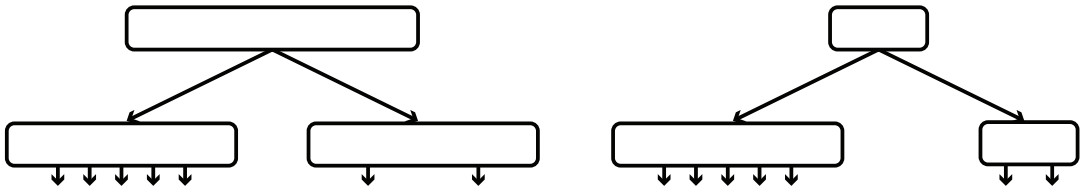
Slika 3.1: Primer radiks-stabla, gde su čvorovi koji sadrže vrednosti obeleženi žutom bojom, a crvenom bojom su napisani ključevi koji im odgovaraju



Slika 3.2: Radiks-stablo dobijeno dodavanjem ključeva `PO` i `MAK`

Problem koji se javlja kod radiks-stabala jeste prevelik utrošak memorije. Ukoliko svaki čvor predstavlja jedan bajt ključa, to znači da svaki čvor može imati, u najgorem slučaju, 2^8 dece, odnosno 256 pokazivača na decu. Kako svaki čvor neće imati maksimalan broj dece, rezervisanje memorijskog prostora za najgori slučaj rezultuje prekomernim korišćenjem memorije u prosečnom slučaju. Lajs, Kemper i Nojman su, na IEEE internacionalnoj konferenciji inženjeringa podataka 2013.

godine, predstavili prilagodljiva radiks-stabla, skraćeno nazvana **ART** [18] (engl. ***A**daptive **R**adix **T**ree*) kao potencijalno novo rešenje, zato što ona dinamički prilagođavaju veličinu unutrašnjih čvorova u odnosu na broj dece. Na slici 3.3 je ilustrovana razlika između radiks-stabla i prilagodljivog radiks-stabla.



Slika 3.3: Radiks-stablo (levo) i prilagodljivo radiks-stablo (desno)

3.1 Struktura prilagodljivih radiks-stabala

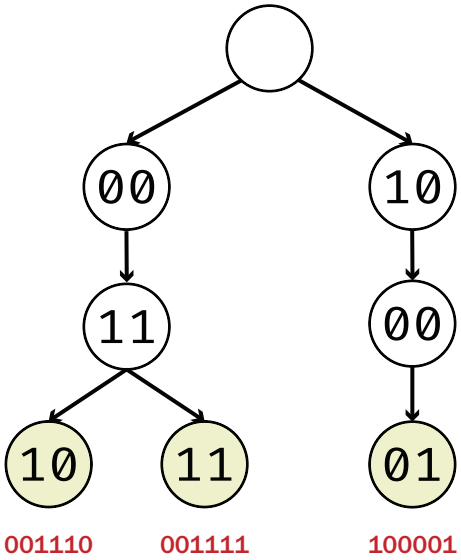
Čvorovi radiks-stabla ne moraju nužno memorisati jedan bajt ključa. Broj bitova koji unutrašnji čvorovi memorišu se naziva **raspon** (engl. *span*). To znači da, u svom osnovnom obliku, radiks-stabla imaju visinu koja zavisi od dužine najdužeg ključa k i od raspona s, i ona iznosi $\lceil \frac{k}{s} \rceil$.

Odabir raspona ima uticaja i na vremensku i na memorijsku složenost radiks--stabla. Ukoliko je raspon mali, onda je potencijalni broj dece po unutrašnjem čvoru manji, pa je memorija bolje iskorišćena. S druge strane, visina stabla je velika, te su operacije na stablu sporije. I suprotno važi: ukoliko je raspon veliki, onda je visina stabla mala i operacije se vrše brzo, ali je memorijska složenost prevelika. Implementacije radiks-stabala GPT [5] i LRT [8] koriste četiri, odnosno šest bitova raspona.

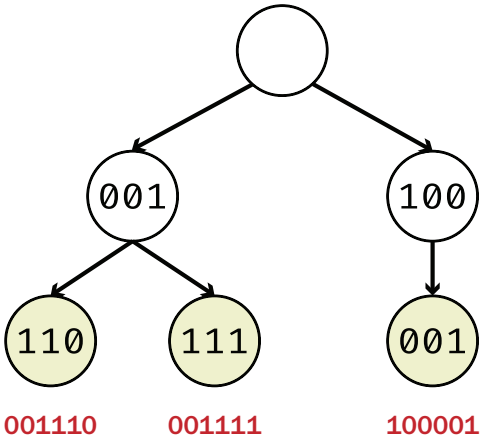
Na slikama 3.4 i 3.5 su prikazani primeri radiks-stabala sa vrednostima raspona 2 i 3. U oba stabla su uneti parovi sa istim šestobitnim ključevima: `001110`, `001111` i `100001`.

ART ima raspon od osam bita, odnosno jedan bajt. Da bi se sprečila prekomerna upotreba memorije, koriste se četiri različita tipa unutrašnjih čvorova u odnosu na maksimalan broj pokazivača na decu, koji su ilustrovani na slici 3.6:

Node4 Najjednostavniji oblik unutrašnjeg čvora i sastoji se od dva vektora dužine 4: vektor ključeva koji sadrži bajtove ključa i vektor dece koji sadrži



Slika 3.4: Primer radiks-stabla sa rasponom 2



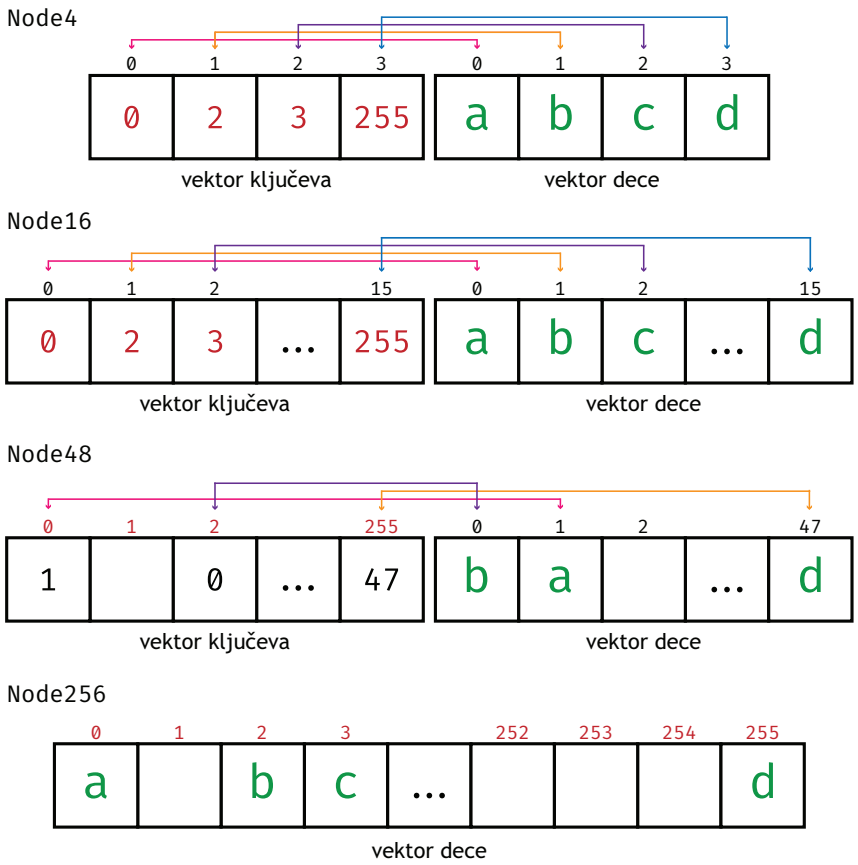
Slika 3.5: Primer radiks-stabla sa rasponom 3

pokazivače na decu. Svaki element vektora ključeva odgovara elementu na istoj poziciji u vektoru dece. Budući da je vektor ključeva male veličine, on se pretražuje linearnom pretragom.

Node16 Koristi dva vektora dužine 16. Kao i kod `Node4`, elementi dva vektora su upareni. Kako su vektori većeg kapaciteta, tako linearna pretraga predstavlja nezanemarljiv utrošak procesorskog vremena. Zbog toga se koriste pomenute SIMD instrukcije koje su hardverski podržane od strane procesora i koje omogućavaju paralelnu pretragu svih elemenata vektora ključeva.

Node48 Koristi vektor ključeva koji sadrži maksimalnih 256 elemenata kao i vektor dece koji sadrži 48 elemenata. Bajtovi pretrage su implicitno pohranjeni kao indeksi vektora ključeva. U vektoru ključeva se nalaze indeksi elemenata vektora dece, koji odgovaraju datom bajtu pretrage. SIMD instrukcije se u ovom slučaju ne mogu koristiti zato što su SIMD vektori hardverski ograničeni po broju bitova koje mogu sadržati.

Node256 Sastoji se od samo jednog vektora koji sadrži pokazivače na decu, dok se bajtovi pretrage implicitno čuvaju kao indeksi vektora dece. Ovaj tip koristi više memorije od `Node48`, zato što pokazivači zauzimaju više memorije od čuvanja indeksa koji se predstavljaju jednim bajtom.



Slika 3.6: Četiri tipa unutrašnjeg čvora u strukturi ART. Strelicama su povezani odgovarajući parovi elemenata u vektorima ključeva i dece. Sa tri tačke su obeleženi elementi koji nisu prikazani na slici. Crvenom bojom su obeleženi bajtovi pretrage, dok su pokazivači na decu predstavljeni zelenim slovima.

Ukoliko čvor popuni svoj kapacitet, onda se on transformiše u naredni čvor po veličini.

3.2 Memorijska optimizacija unutrašnjih čvorova

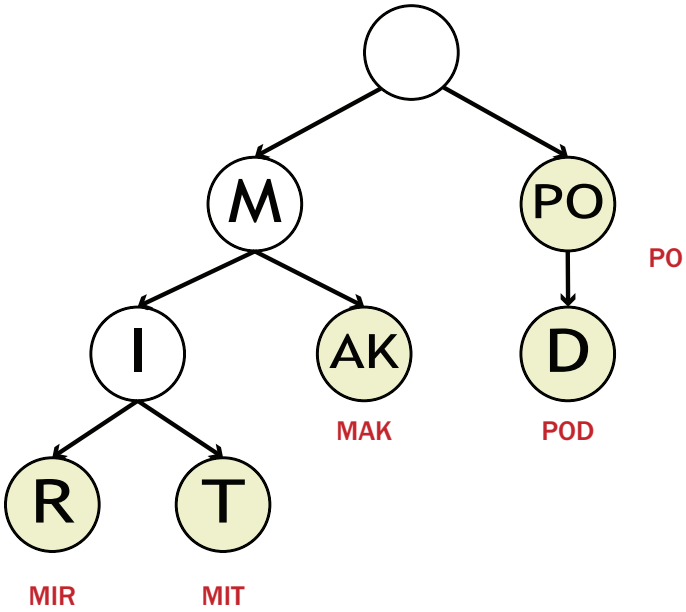
Parcijalni ključ čvora radiks-stabla se odnosi na deo ključa koji taj čvor reprezentuje. Nadovezujući parcijalne ključeve čvorova na putu od korena stabla do bilo kog drugog čvora predstavlja konkretan ključ tog čvora. U svom osnovnom obliku, u radiks-stablu su parcijalni ključevi fiksirane dužine.

Da bi se memorijska iskorišćenost optimizovala, koristi se metoda **sažimanja puteva** (engl. *path compression*) koja udružuje unutrašnje čvorove koji imaju samo jedno dete sa tim detetom, ukoliko roditelj ne sadrži vrednost. Prilikom udruživanja dva čvora, parcijalni ključevi tih čvorova se nadovezuju. Kako broj bitova parcijalnih ključeva reprezentuju nije više fiksirana veličina, svaki bit izvan raspona stabla se mora čuvati eksplicitno. Postoje dva metoda čuvanja parcijalnih ključeva unutrašnjih čvorova:

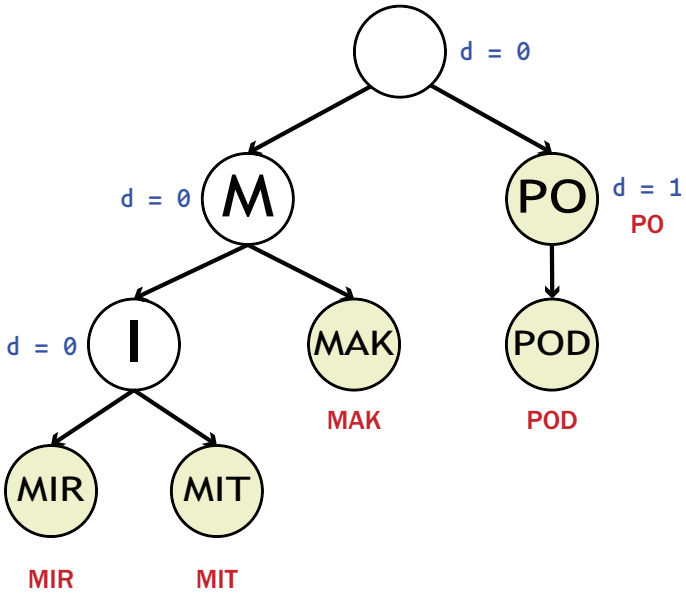
Pesimističan Svaki unutrašnji čvor ima odgovarajući vektor u kome se čuvaju bitovi parcijalnog ključa. Primer pesimističnog sažimanja puteva dat je na slici 3.7, gde je sažimanje puteva primenjeno na primeru sa slike 3.2.

Optimističan Svaki unutrašnji čvor čuva informaciju o dužini parcijalnog ključa d. Prilikom pretraživanja stabla preskače se d bitova ključa pretrage bez provere jednakosti, i pretraga se nastavlja dalje. Kad pretraga stigne do lista, tada se ceo ključ proverava. Na slici 3.8 je prikazano optimistično sažimanje puteva, gde su vrednosti d ispisane plavom bojom za svaki unutrašnji čvor.

Pesimističan metod daje brzu proveru korektnosti putanje, ali sa sobom nosi povećanu potrošnju memorije. Uz to, može doći i do fragmentacije memorije kada se vektori parcijalnih ključeva menjaju. Nasuprot tome, optimističan metod nema problema vezanih za memoriju, ali pretraga može da ode pogrešnom granom zato što se nije dokazala jednakost parcijalnih ključeva. Tada se pretraga mora vratiti unazad, što otežava implementaciju i ima negativan uticaj na performanse. Postoji i hibridan način čuvanja parcijalnih ključeva, gde se u unutrašnjim čvorovima čuva



Slika 3.7: Pesimistično sažimanje puteva radiks-stabla, gde su samo čvorovi `A` i `K`, kao i `P` i `O` sažeti, dok `PO` i `D` nisu, jer čvor `PO` sadrži vrednost.



Slika 3.8: Optimistično sažimanje puteva radiks-stabla, gde su samo čvorovi `A` i `K`, kao i `P` i `O` sažeti, dok `PO` i `POD` nisu, jer čvor `PO` sadrži vrednost. Pošto `PO` sadrži vrednost, bajt `O` se mora čuvati.

fiksni broj bajtova parcijalnog ključa. Sve preko unapred određenog broja bajtova tog vektora se ne memoriše, već se koristi optimističan metod čuvanja.

3.3 Algoritmi pretrage i umetanja

U svom radu iz 2013. godine, autori prilagodljivih radiks-stabala su objavili i algoritme pretrage i umetanja. Ovi algoritmi su veoma slični algoritmima za radiks--stabla u osnovnom obliku, i zato će biti navedeni i objašnjeni samo algoritmi za rad sa strukturom podataka ART. Algoritam pretrage je dat na listingu 3.1.

```
1  search(node, key, depth):
2      if node == NULL
3          return NULL
4      if isLeaf(node)
5          if leafMatches(node, key, depth)
6              return node
7          return NULL
8      if checkPrefix(node, key, depth) != node.prefixLen
9          return NULL
10     depth += node.prefixLen
11     next = findChild(node, key[depth])
12     return search(next, key, depth + 1)
```

Listing 3.1: Algoritam pretrage strukture podataka ART

Funkcija `search` ima tri parametra: trenutni čvor pretrage, ključ pretrage i dubinu trenutnog čvora. Kako je ovaj algoritam rekurzivan, prvo se proverava bazni slučaj kada je trenutni čvor ili prazan ili list, gde se rezultat vraća samo ukoliko je čvor list i ključ u tom listu se poklapa sa ključem pretrage. U drugom slučaju, ako je pretraga trenutno u unutrašnjem čvoru, prvo se proverava poklapanje parcijalnog prefiksa u čvoru u liniji 5 i pretraga se završava ako dođe do nejednakosti parcijalnih ključeva. Zatim se koriguje dubina pretrage promenljivom `depth`, zato što se on odnosi na indeks bajtova koji se porede u tom trenutku. Na kraju, nalazi se idući čvor funkcijom `findChild` prosleđivanjem trenutnog čvora i odgovarajućeg bajta i pretraga se nastavlja rekurzivno. Isti algoritam pretrage se može primeniti i za radiks-stabla u osnovnom obliku. Algoritam funkcije `findChild` je dat na listingu 3.2.

```
1  findChild(node, byte):
2      if node.type == Node4
3          for (i = 0; i < node.count; i = i + 1)
4              if node.key[i] == byte
5                  return node.child[i]
```

```
6           return NULL
7       if node.type == Node16
8           for (i = 0; i < node.count; i = i + 1)
9               key = _mm_set1_epi8(byte)
10              cmp = _mm_cmpeq_epi8(key, node.key)
11              mask = (1 << node.count) - 1
12              bitfield = _mm_movemask_epi8(cmp) & mask
13              if bitfield
14                  return node.child[ctz(bitfield)]
15              else
16                  return NULL
17      if node.type == Node48
18          if node.childIndex[byte] != EMPTY
19              return node.child[node.childIndex[byte]]
20          else
21              return NULL
22      if node.type == Node256
23          return node.child[byte]
```

Listing 3.2: Algoritam pronalaska idućeg čvora pretrage

Naredni čvor se nalazi na četiri različita načina, u odnosu na tip trenutnog čvora koji je poslat kao parametar funkciji `findChild`:

Node4 Naredni čvor se nalazi linearnom pretragom svih elemenata vektora ključeva.

Node16 U ovom slučaju se koriste SIMD instrukcije. Prvo se, pomoću funkcije `_mm_set1_epi8` inicijalizuje SIMD vektor od 128 bita, odnosno, 16 elemenata veličine jednog bajta. Svi elementi tog vektora biće jednaki bajtu koji je unet kao parametar. Posle inicijalizacije se porede svi elementi novog vektora i vektora ključeva trenutnog čvora funkcijom `_mm_cmpeq_epi8`. Kao rezultat se dobija vektor iste dužine, ali čiji elementi su binarne vrednosti `0` i `11111111`, u zavisnosti da li su elementi na odgovarajućoj poziciji jednaki ili ne. U liniji 11 se pravi maska zato što vektor elemenata nije nužno pun, ali postoji mogućnost poklapanja bajta pretrage sa bajtovima koji nisu u upotrebi, zato što SIMD vektor mora biti inicijalizovan u potpunosti prilikom njegovog definisanja. Maska je veličine 32 bita, pa se vektor `cmp` šalje funkciji

`_mm_movemask_epi8` koja vraća 32-bitnu vrednost gde je svaki element vektora (ukupno ih je 32) reprezentovan svojim najznačajnijim bitom. Kada se primeni maska na dobijenu vrednost, dobija se 32-bitna vrednost `bitfield`, a ključ se nalazi u mestu kojem odgovara bit sa vrednošću 1. Da bi se našao indeks prvog bita sa vrednošću 1, koristi se funkcija `ctz` (engl. ***c**ount **t**railing **z**eros*) koja broji nule na kraju vrednosti zapisane u binarnom sistemu.

Node48 Indeks narednog čvora je upisan u vektoru `childIndex` koji ima 256 elemenata, za svaku vrednost bajta pretrage. Ukoliko vrednost za taj ključ nije uneta, u vektor `childIndex` se piše fiksirana vrednost `EMPTY` koja je veća od 48 (maksimalnog broja elemenata u nizu dece).

Node256 Koristi se niz maksimalne veličine od 256 elemenata gde se bajt ključa direktno koristi za indeksiranje.

U slučaju osnovnih radiks-stabala, algoritam `findChild` je sličan `Node256` grani iz algoritma sa listinga 3.2 koja se prilagođava za različite vrednosti raspona s. Veličina vektora koji se nalaze u unutrašnjim čvorovima je uslovljena odabirom vrednosti raspona i ona iznosi 2^s.

Algoritam za umetanje novih parova ključ-vrednost u stablo je dat na listingu 3.3, dok je algoritam brisanja parova, po rečima autora, simetričan ovom algoritmu. Ponovo je u pitanju rekurzivni algoritam, te se prvo obrađuje bazni slučaj, odnosno kad se praznom podstablu dodaje list. Ako je čvor kome se dodaje element list, onda se pravi novi unutrašnji čvor funkcijom `makeNode4` koji će da sadrži čvor koji se dodaje kao i stari list. Pre toga, mora se utvrditi dužina prefiksa u petlji u liniji 8.

Ukoliko stablo nije prazno, niti je u pitanju list, onda je prosleđeni čvor unutrašnji. Ako prefiks unutrašnjeg čvora ne odgovara traženom ključu, onda dolazi do račvanja stabla na dve grane gde se ponavlja slična procedura kao i kod stabla koje sadrži samo jedan list. Na kraju, ako se prefiksi poklapaju, onda se traži sledeći čvor gde bi se pretraga mogla nastaviti. Ako naredni čvor u pretrazi postoji, onda se pretraga rekurzivno nastavlja. U suprotnom, pretraga se zaustavlja i novi list se dodaje trenutnom unutrašnjem čvoru. Trenutni unutrašnji čvor može biti popunjen, pa se njegov kapacitet mora proširiti pretvaranjem trenutnog unutrašnjeg čvora u sledeći čvor u hijerarhiji funkcijom `grow` (iz `Node4` u `Node16`, iz `Node16` u `Node48` ili iz `Node48` u `Node256`). Funkcija `grow` je jedina suštinska razlika između algoritma sa listinga 3.3 i algoritma za umetanje u radiks-stablo i

njena implemenacija zavisi od specifičnosti konkretnog programskog jezika u kome se ona implementira. Usled toga, funkcija `grow` će biti prikazana u narednom poglavlju, gde se opisuje implemenacija strukture ART u okviru programskog jezika *Rust*.

```
insert (node, key, leaf, depth):
 if node == NULL
    replace(node, leaf)
    return
 if isLeaf(node)
    newNode = makeNode4()
    key2=loadKey(node)
    for (i = depth; key[i] == key2[i]; i = i + 1)
        newNode.prefix[i - depth] = key[i]
    newNode.prefixLen = i - depth
    depth = depth + newNode.prefixLen
    addChild(newNode, key[depth], leaf)
    addChild(newNode, key2[depth], node)
    replace(node, newNode)
    return
 p=checkPrefix(node, key, depth)
 if p != node.prefixLen
    newNode = makeNode4()
    addChild(newNode, key[depth + p], leaf)
    addChild(newNode, node.prefix[p], node)
    newNode.prefixLen = p
    memcpy(newNode.prefix, node.prefix, p)
    node.prefixLen = node.prefixLen-(p + 1)
    memmove(node.prefix,node.prefix + p + 1,node.prefixLen)
    replace(node, newNode)
    return
 depth = depth + node.prefixLen
 next = findChild(node, key[depth])
 if next
    insert(next, key, leaf, depth + 1)
 else
    if isFull(node)
        grow(node)
```

```
34  addChild(node, key[depth], leaf)
```

Listing 3.3: Algoritam umetanja novih elemenata u stablo

Vremenska složenost funkcija `search` i `insert` zavisi od dubine stabla, koje iznosi $\lceil \frac{k}{s} \rceil$. Kako je raspon s konstantan i iznosi osam bita, vremenska složenost je $\mathcal{O}(k)$, gde je k dužina ključa.

Glava 4

ART u programskom jeziku *Rust*

U okviru ovog rada je implementirana biblioteka **rustART** koja implementira prilagodljiva radiks-stabla u programskom jeziku *Rust*. Izvorni kôd ove biblioteke je dostupan na veb-stranici https://github.com/jdmitrovic/rustART.

Biblioteke napisane u programskom jeziku *Rust* imaju osnovnu datoteku koja se zove `lib.rs`, koju je moguće deliti u manje datoteke koje se nazivaju moduli. U biblioteci *rustART* postoje tri modula:

`tree.rs` — Sadrži implementacije osnovnih operacija: umetanja (engl. *insert*), ažuriranja (engl. *update*), pretraživanja (engl. *find*) i brisanja (engl. *delete*) parova ključ-vrednost.

`keys.rs` — Sadrži sve strukture i funkcije za rad sa ključevima.

`node.rs` — Sadrži unutrašnju strukturu prilagodljivih radiks-stabala, odnosno strukturu unutrašnjih čvorova i listova, kao i njihovo ponašanje.

4.1 Struktura `ARTree` i svojstvo `ARTKey`

Struktura koja predstavlja prilagodljiva radiks-stabla se zove `ARTree` i data je u listingu 4.1. Definisana je u datoteci `lib.rs`, tako da je brzo dostupna krajnjem korisniku koji koristi ovu biblioteku. U definiciju `ARTree` ulaze dva generička tipa: tip ključa i tip vredosti parova koji se unose u stablo. Prvo polje strukture je `root`, koje se odnosi na koren prilagodljivog radiks-stabla. Da je koren i jedino polje ove strukture, kompilator ne bi dopustio korišćenje generičkog tipa `K` koje se ne pominje unutar same strukture. Zbog toga, mora se koristiti tzv. **fantomski**

marker, koji kao generički tip uzima `K`. Ovaj tip podataka se naziva „fantomskim“ zato što ne koristi memoriju, već je samo vrsta indikatora za kompilator.

```
pub struct ARTree<K: ARTKey, V> {
    root: ARTLink<V>,
    _marker: PhantomData<K>,
}
```

Listing 4.1: Definicija strukture `ARTree`

U listingu 4.1 generički tip `K` ima svojsvo `ARTKey`. Ovo svojstvo, dato na listingu 4.2 je definisano u modulu `keys.rs`.

```
pub trait ARTKey {
    fn convert_to_bytes(self) -> Vec<u8>;
}
```

Listing 4.2: Definicija svojstva `ARTKey`

`ARTKey` sadrži samo jednu funkciju: funkciju koja može dati ključ prevesti u vektor bajtova, odnosno `Vec<u8>`. Da bi se vrednost proizvoljnog tipa mogla koristiti kao ključ u `ARTree`, taj tip mora prvo implementirati ovo svojstvo. U datoteci `key.rs` su date implementacije ovog svojstva za primitivne tipove (celi brojevi, brojevi zapisani u pokretnom zarezu), kao i za niske. Funkcija standardne biblioteke `to_be_bytes` je definisana za sve brojeve i ona prevodi brojeve u njihovu reprezentaciju u bajtovima u *big-endian* [6] sistemu. *Big-endian* se odnosi na poredak bajtova smeštenih u memoriji, odnosno na činjenicu da su bajtovi u memoriji smešteni redom od najznačajnijeg do najmanje značajnog bajta. S druge strane, bajtovi u *little-endian* poretku su smešteni u memoriji od najmanje značajnog do najznačajnijeg bajta. Funkcija `to_be_bytes` smešta bajtove u niz, koji se mora prevesti u vektor funkcijom `into_vec`. Standardna biblioteka takođe sadrži odgovarajuću funkciju za prevođenje niski u bajtove koja se naziva `into_bytes`.

4.2 Struktura čvorova

Definicije tipova čvorova se nalaze u datoteci `node.rs`. Osnovni tip čvora se naziva `ARTNode` i on je enumerator. Na listingu 4.3 je prikazan deo koda sadrži definicije svih tipova čvorova.

U definiciji strukture `ARTree` na listingu 4.1 koren stabla ima tip `ARTLink<V>`. Ovaj tip je pseudonim tipa `Option<ARTNode<V>>` koji predstavlja pokazivače na decu.

```
pub enum ARTInnerNode<V> {
    Inner4(Box<ARTInner4<V>>),
    Inner16(Box<ARTInner16<V>>),
    Inner48(Box<ARTInner48<V>>),
    Inner256(Box<ARTInner256<V>>),
}

pub struct ARTInner4<V> {
    pkey_size: u8,
    keys: [Option<u8>; 4],
    children: [ARTLink<V>; 4],
    children_num: u8,
}

pub struct ARTInner16<V> {
    pkey_size: u8,
    keys: __m128i,
    children: [ARTLink<V>; 16],
    children_num: u8,
}

pub struct ARTInner48<V> {
    pkey_size: u8,
    keys: [Option<u8>; 256],
    children: [ARTLink<V>; 48],
    children_num: u8,
}

pub struct ARTInner256<V> {
    pkey_size: u8,
    children: [ARTLink<V>; 256],
    children_num: u16,
}

```

```
35 pub struct ARTLeaf<V>{
36     key: ByteKey,
37     value: V,
38 }
```

Listing 4.3: Definicije čvorova koji se nalaze u strukturi `ARTree`

`ARTNode` razlikuje dve vrste čvorova: unutrašnje `Inner` i listove `Leaf`. U čvorove tipa `Leaf` se smešta struktura `ARTLeaf` koja sadrži vrednost i ključ dok se u čvorove tipa `Inner` smeštaju tri podatka: enumerator `ARTInnerNode`, parcijalni ključ i vrednost. Vrednost se ovde nalazi u `Option` omotaču, jer unutrašnji čvorovi ne moraju nužno imati vrednosti.

`ARTInnerNode` sadrži četiri vrste unutrašnjeg čvora: `Inner4`, `Inner16`, `Inner48` i `Inner256` gde su, redom, smeštene odgovarajuće strukture `ARTInner4`, `ARTInner16`, `ARTInner48` i `ARTInner256`. Struktura ovih čvorova se suštinski ne razlikuje od njihovog opisa u sekciji 3.1, s tim da su dodati podaci o dužini parcijalnog čvora i broju smeštene dece. `Box` pokazivač je ovde neophodan zato što je `ARTNode` rekurzivna struktura, i kompilator bez njega ne bi znao koliko memorije treba rezervisati. Svi pomenuti tipovi čvorova imaju jedan generički tip: tip vrednosti. Generički tip ključa nije neophodan nakon njegovog transformisanja u vektor bajtova.

4.3 Implementacija osnovnih operacija

U datoteci `tree.rs` se nalaze funkcije koje implementiraju osnovne operacije. Umetanje i ažuriranje parova se spaja u jednu funkciju, nazvanu `insert`, koja umeće dati par u strukturu ukoliko ne postoji drugi par sa istim ključem u toj strukturi. U suprotnom, vrednost za taj par se ažurira i funkcija vraća staru vrednost.

Za razliku od algoritama predstavljenih u sekciji 3.3, njihove implementacije u biblioteci *rustART* nisu napisane kao rekurzivne funkcije, budući da rekurzija sa sobom donosi mogućnost prekoračenja steka. Implementirana je funkcija `get` kao *Rust* ekvivalent funkciji `search` sa listinga 3.1. Rekurzija je zamenjena beskonačnom petljom iz koje se izlazi samo ako se nađe par sa traženim ključem ili se utvrdi da takav par ne postoji.

Implementirana je funkcija `find_child`, koja je pandan funkciji `findChild` sa listinga 3.2. Međutim, usled pravila pozajmljivanja, implementiran je i mutabilni dvojnik funkcije `find_child` sa nazivom `find_child_mut`. Obe funkcije su sastavni deo strukture `ARTInnerNode`.

Umetanje i ažuriranje

Implementacija funkcije `insert` je složenija zato što ova funkcija, za razliku od funkcije pretrage, zahteva menjanje same strukture, koje mora biti u skladu sa pravilima pozajmljivanja i vlasništva. Rad ove funkcije je podeljen u dva koraka:

1. Nalaženje poslednjeg čvora koji se poklapa sa zadatim ključem.
2. Dodavanje para na mesto nađeno u prvom koraku.

Cilj prvog koraka je nalaženje poslednjeg čvora na putu pretrage, ukoliko on postoji. Ovaj korak je implementiran u vidu beskonačne petlje gde se pretraga kreće od korena niz stablo sve dok se kreće po unutrašnjim čvorovima i dok se ključ pretrage i parcijalni ključevi unutrašnjih čvorova u potpunosti slažu. Ovo znači da se u petlju ne ulazi ako je stablo prazno i tada će rezultat prvog koraka biti nedostajuća vrednost. Pretraga se prekida kada se dođe do lista ili pretraga u unutrašnjem čvoru vodi ka nedostajućoj vrednosti. Takođe, u ovom koraku se proverava da li je pretraga došla do kraja, odnosno, da li treba uneti vrednost u trenutni unutrašnji čvor i vratiti staru vrednost, ako ona postoji.

Drugi korak zavisi od toga šta je rezultat pretrage u prvom koraku. Ukoliko je u pitanju nedostajuća vrednost, kako se iz prvog koraka nedostajuća vrednost može dobiti samo ako je stablo prazno, nedostajuća vrednost se zamenjuje listom koji sadrži dati par.

Ukoliko je u pitanju list, proverava se ključ pretrage i ključ u tom listu. Ako dođe do potpunog poklapanja, ažurira se vrednost u listu i vraća se stara vrednost. U suprotnom, ako je došlo do parcijalnog poklapanja ključa, pravi se novi unutrašnji čvor koji menja stari list. Parcijalni ključ novog unutrašnjeg čvora je deo ključa pretrage koji se poklapa sa ključem u listu. U novonastali čvor dodaje se stari list kao i novi list, u kome se nalazi uneti par ključ-vrednost.

Ukoliko je u pitanju unutrašnji čvor, neophodno je utvrditi zašto je došlo do prekida petlje u prvom koraku. Ako je došlo do parcijalnog poklapanja ključa pretrage i parcijalnog ključa neophodno je napraviti novi unutrašnji čvor čiji parcijalni ključ je deo ključa pretrage koji se poklapa sa parcijalnim ključem nađenog unutrašnjeg čvora. U novonapravljeni čvor se dodaje stari čvor kao i list sa novim parom. Ako nije u pitanju parcijalno poklapanje, onda pretraga vodi u nedostajuću vrednost, odnosno pretraga je došla do kraja puta koji može da prati u dosada-

šnjem stablu. Tada se dodaje list sa novim parom u unutrašnji čvor, koji se, po potrebi, može proširiti funkcijom `grow`.

Funkcija `grow` je implemenirana nad sva četiri tipa unutrašnjeg čvora, i omogućava popunjenom čvoru da uveća svoj kapacitet. Većina vektora ključeva i dece u okviru unutrašnjih čvorova se proširuje iterativnim putem, odnosno prebacivanjem svakog elementa pojedinačno iz manjeg vektora u veći. Izuzetak bi trebalo da predstavlja `Inner16`, gde se pojavljuje SIMD vektor kao vektor ključeva. Međutim, implementacija SIMD instrukcija u programskom jeziku *Rust* omogućava da se SIMD vektori ponašaju kao nizovi. U slučaju gde se `Inner256` proširuje, doći će do greške zato što je `Inner256` čvor maksimalne veličine, te se ne može proširiti.

Brisanje

Operacija brisanja briše par ključ-vrednost iz strukture i vraća staru vrednost ukoliko traženi par postoji. Slično operaciji umetanja i ažuriranja iz potsekcije 4.3, operacija brisanja se takođe sastoji iz dva koraka:

1. Nalaženje poslednjeg unutrašnjeg čvora čiji se prefiksni ključ poklapa sa odgovarajućim delom ključa pretrage.

2. Brisanje para ključ-vrednost sa mesta nađenog u prvom koraku.

Prvi korak je takođe realizovan beskonačnom petljom, gde se pretraga kreće po unutrašnjim čvorovima niz stablo. Ako pretpostavimo da je traženi par u listu, neophodno je naći njegovog roditelja zato što je potrebno izbrisati podatak o ključu lista koji se briše. Uz to, potrebno je i umanjiti brojač dece i smanjiti taj unutrašnji čvor ako je to moguće.

Ako se vrednost koja se briše ne nalazi u listu, već u unutrašnjem čvoru, u prvom koraku će se ta vrednost naći i izbrisati. Par kome pripada vrednost koja se nalazi u unutrašnjem čvoru se ne ubraja u decu tog čvora, pa se broj dece ne koriguje prilikom brisanja ove vrednosti, niti se čvor smanjuje. Nađeni unutrašnji čvor se ne briše zato što on sadrži decu, pa bi se time obrisao i deo stabla koji ne treba brisati.

Po završetku prvog koraka se ne dobija unutrašnji čvor ako je stablo prazno (u ovom slučaju se funkcija prekida) ili se sastoji od jednog lista (list se briše samo ukoliko se njegov ključ u potpunosti poklapa sa ključem pretrage). Uz pretpostavku da se u prvom koraku dobija unutrašnji čvor, briše se odgovarajući list iz dobijenog

čvora funkcijom `remove_node`. Potom se, po potrebi, unutrašnji čvor smanjuje funkcijom `shrink` i vraća se vrednost obrisanog lista.

4.4 Ispravnost i merenje performansi

Ispravnost strukture `ARTree` se proverava jediničnim testovima u `lib.rs` datoteci. Implementacija merenja performansi se nalazi u datoteci `benches.rs`, koja se nalazi u direktorijumu `benches`. Prilikom merenja performansi, istovremeno se proverava ispravnost parova unetih u `ARTree`.

Testiranje

U datoteci `lib.rs` nalaze se dva testa koji proveravaju valjanost osnovnih operacija. Testovi su napisani u vidu funkcija pre kojih je napisana anotacija `#[test]`. Testovi se pokreću komandom `cargo test` i potrebno im je 0.84 sekundi za izvršavanje na procesoru *AMD Ryzen 7 3700U*.

Prvi test je `insert_update_delete_get`, prikazan na listingu 4.4, gde se u `ARTree` unosi proizvoljno velika količina parova. Za generisanje ključeva se koristi generator pseudoslučajnih brojeva PCG [20] koji generiše niz 64-bitnih vrednosti za prosleđenu početnu vrednost `SEED`. Jednostavnosti radi, parovi se prave tako da za svaku vrednost v koja je uparena sa ključem k važi da je $v = k+1$. Nakon unosa svih parova u `ARTree` funkcijom `insert`, postojanje tih parova u samoj strukturi se proverava funkcijom `get`. Sledi provera funkcionalnosti operacije brisanja gde se koristi nasumično odabrana vrednost `SEED_DEL`, tako da se brišu samo vrednosti koje su deljive sa tom vrednošću. Nakon brisanja tih vrednosti, proverava se da li su se obrisani parovi zaista obrisali. Takođe, postoji i provera da li se neobrisani parovi i dalje nalaze u strukturi `ARTree`.

```
#[test]
fn insert_update_delete_get() {
    const SEED: u64 = 10;

    let mut rng = Pcg64::seed_from_u64(SEED);
    let mut keys: Vec<u64> = vec![0; 100_000];
    rng.fill(&mut keys[..]);
    let mut art = ARTree::<u64, u64>::new();

```

```
10      for &key in keys.iter() {
11          art.insert(key, key + 1);
12      }
13
14      for &key in keys.iter() {
15          assert_eq!(key + 1, *art.get(key).unwrap());
16      }
17
18      const SEED_DEL: u64 = 13;
19
20      for &key in keys.iter() {
21          if key % SEED_DEL == 0 {
22              art.delete(key);
23          }
24      }
25
26      for &key in keys.iter() {
27          if key % SEED_DEL == 0 {
28              assert_eq!(None, art.get(key));
29          } else {
30              assert_eq!(key + 1, *art.get(key).unwrap());
31          }
32      }
33  }
```

Listing 4.4: Test `insert_update_delete_get`

Drugi test je test `string_art`, gde se u ART unose nasumične niske sa odgovarajućim vrednostima. Ovaj test je manjeg obima, i služi samo da se proveri da li se niske adekvatno transformišu u nizove bajtova. Nakon unosa parova u `ARTree`, provere se izvršavaju na sličan način kao i u prvom testu.

Performanse

Datoteka `benches.rs` sadrži funkcije koje mere performanse umetanja i pretraživanja parova u strukturi `ARTree`. Rezultati se potom porede sa rezultatima istih operacija izvedenim nad sličnim strukturama koje su implementirane u standardnoj biblioteci programskog jezika *Rust*.

Merenje je izvršeno pomoću paketa *Criterion.rs* [9]. Sva implementirana merenja se pokreću komandom `cargo bench`. Paket `Criterion` takođe omogućava iscrtavanje grafika na osnovu dobijenih rezultata.

Deo koda koji se nalazi u `benches.rs` datoteci se nalazi na listingu 4.5. U funkciji `bench_gets` se nalazi implementacija merenja operacije pretraživanja. Na analogan način je implementirano i merenje operacije umetanja.

Prvo se uspostavavlja generator pseudoslučajnih brojeva. Zatim, pravi se nova grupa merenja sa nazivom `Gets`. U grupi testova je moguće pokrenuti više merenja uzastopno i uporediti ih. U ovom slučaju se poredi struktura `ARTree` sa heš mapama i B-stablima [4].

```
fn bench_gets(c: &mut Criterion) {
    let mut rng = Pcg64::seed_from_u64(SEED);

    let mut group = c.benchmark_group("Gets");
    for i in (200_000..4_000_001).step_by(200_000) {
        let mut keys: Vec<u64> = vec![0; i];
        rng.fill(&mut keys[..]);
        let mut art = ARTree::<u64, u64>::new();
        let mut hmap = HashMap::<u64, u64>::new();
        let mut btree = BTreeMap::<u64, u64>::new();

        for key in keys.iter() {
            art.insert(*key, *key + 1);
            hmap.insert(*key, *key + 1);
            btree.insert(*key, *key + 1);
        }

        group.bench_with_input(BenchmarkId::new("ARTree", i), &keys,
            |b, k| b.iter(|| art_get(&mut art, k)));
        group.bench_with_input(BenchmarkId::new("HashMap", i), &keys,
            |b, k| b.iter(|| hmap_get(&mut hmap, k)));
        group.bench_with_input(BenchmarkId::new("BTree", i), &keys,
            |b, k| b.iter(|| btree_get(&mut btree, k)));
    }
    group.finish();
```

```
26 }
```

Listing 4.5: Funkcija `bench_gets`

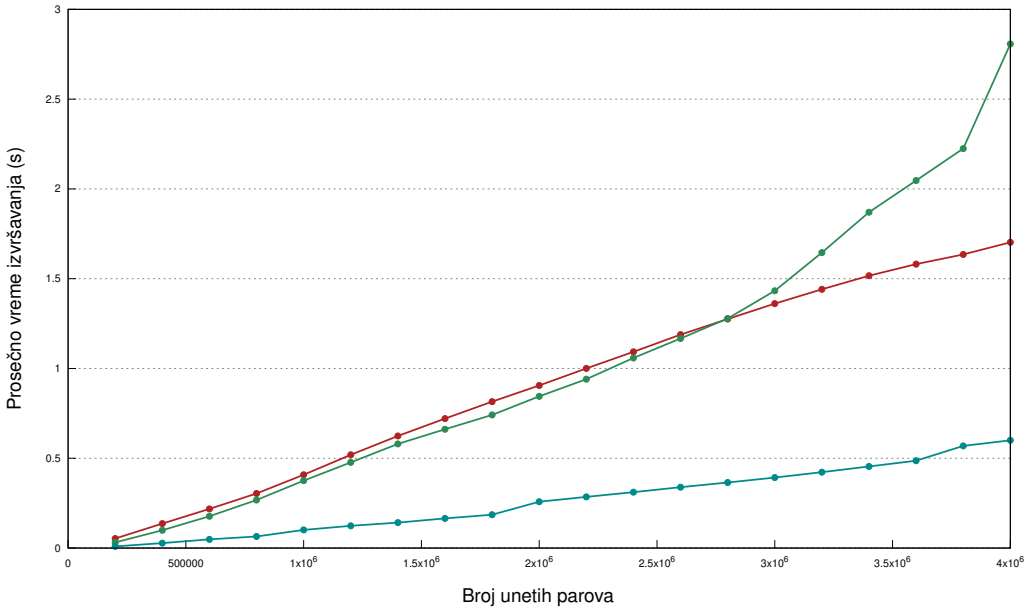
Brojačem `i` u `for` petlji se kontroliše broj pretraženih parova. Brojač se inkrementira za dve stotine hiljada u svakoj iteraciji do granične vrednosti koja je podešena na četiri miliona. Posle inicijalizovanja svake strukture sa pseudoslučajno generisanim parovima, merenja se pokreću funkcijom `bench_with_input`, za svaku strukturu ponaosob. Funkcija `bench_with_input` ima tri parametra: nisku koja predstavlja naziv merenja, ulazne podatke i funkciju koja uspostavlja merenje. U okviru poslednjeg parametra se koristi pomoćna funkcija koja traži sve unete ključeve i proverava da li je vrednost dobijena pretragom korektna. Na listingu 4.6 se nalazi jedna od pomenutih pomoćnih funkcija, koja je zadužena za pretragu unutar strukture `ARTree`. Sličnu funkcionalnost imaju i funkcije `hmap_get` i `btree_get` koje pretragu vrše unutar heš mape, odnosno B-stabla.

```
1 fn art_get(art: &mut ARTree<u64, u64>, keys: &Vec<u64>) {
2     for key in keys.iter() {
3         assert_eq!(*key + 1, *art.get(*key).unwrap());
4     }
5 }
```

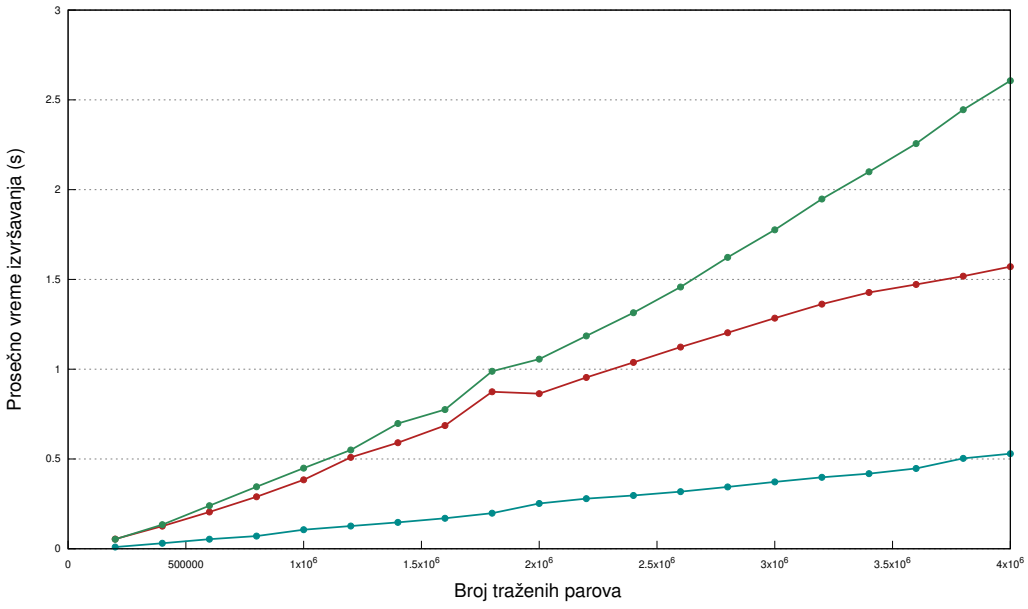
Listing 4.6: Funkcija `art_get`

Merenje je izvršeno na procesoru *AMD Ryzen 7 3700U*. Rezultati merenja operacija umetanja su prikazani na slici 4.1. Vreme izvršavanja za `ARTree` i `BTreeMap` je gotovo identično sve do tri miliona unetih parova, nakon toga `ARTree` ima prednost. Kako je struktura `BTreeMap` zasnovana na stablima pretrage, efikasnost njenih operacija zavisi od količine unetih parova. S druge strane, `ARTree` ne zavisi od broja unetih parova, što dovodi do boljih rezultata u merenjima. Heš mape imaju značajno bolje rezultate u odnosu na druge dve strukture, što je i očekivano s obzirom da je vremenska složenost njihovih operacija konstantna. U slučaju da je neophodno koristiti mapu isključivo za pretraživanja unapred poznatih ključeva, heš mape su bolje rešenje. Međutim, kada ključ nije poznat u celosti, ili je neophodno da se parovi mogu posetiti u sortiranom redosledu (po ključu), onda ima smisla koristiti drvolike strukture kao što je `ARTree`.

Na slici 4.2 se nalaze rezultati merenja operacije pretraživanja. Rezultati se ne razlikuju u velikoj meri od rezulata sa slike 4.1, s tim da se razlika između `ARTree` i `BTreeMap` ispoljava ranije, od dva miliona pretraženih parova.



Slika 4.1: Rezultati merenja za operaciju umetanja. Apscisa predstavlja broj unetih parova, dok ordinata predstavlja prosečno vreme izvršavanja u sekundama. Crvenom bojom je predstavljena struktura `ARTree`, zelenom `BTreeMap`, a plavom `HashMap`.



Slika 4.2: Rezultati merenja za operaciju pretraživanja. Apscisa predstavlja broj pretraživanih parova, dok ordinata predstavlja prosečno vreme izvršavanja u sekundama. Crvenom bojom je predstavljena struktura `ARTree`, zelenom `BTreeMap`, a plavom `HashMap`.

Glava 5

Zaključak

U radu je prikazano korišćenje programskog jezika *Rust* u svrhe predstavljanja koncepta rada prilagodljivih radiks-stabala, kao i njihovih razlika u odnosu na druge drvolike strukture. Teoretski i po rezultatima merenja, ART pokazuje da ima svoje mesto u svetu struktura podataka.

Uprkos nesvakidašnjim konceptima koje *Rust* uvodi kao što je sistem vlasništva, evidentni su razlozi zbog kojih je *Rust* jedan od najpopularnijih jezika današnjice. Usled svoje popularnosti, *Rust* ima veliku podršku zajednice, što je od velike važnosti za relativno mlad programski jezik. Memorijska bezbednost ovog jezika dolazi do izražaja prilikom implementacije struktura podataka, što je bio i slučaj sa bibliotekom *rustART*. Programski jezik *Rust* je na prvi pogled restriktivan, onemogućavajući slobodno korišćenje referenci i pokazivača. Međutim, upravo ta restriktivnost pomaže u sprečavanju grešaka koje je lako napraviti ali teško otkriti.

Eksperimentalni rezultati biblioteke *rustART* pokazuju da prilagodljiva radiks--stabla imaju bolje performanse od B-stabala, koja su implementirana u standardnoj biblioteci programskog jezika *Rust*. Značajan dobitak na performansama je očigledan nakon tri miliona unetih, odnosno dva miliona pretraživanih, parova.

Biblioteka *rustART* se zasniva na originalnom radu koji je predstavio prilagodljiva radiks-stabla 2013. godine. Međutim, od 2013. godine do danas je došlo do tehnološkog pomaka, te današnji procesori mogu koristiti veće SIMD vektore od onih koji su bili dostupni tada. Iz tog razloga se, na primer, mogu promeniti tipovi unutrašnjeg čvora tako da tipovi odgovaraju dužini konkretnog SIMD vektora, optimizujući i performanse i iskorišćenje memorije. Korišćenjem novih tipova, mogao bi se i eliminisati tip unutrašnjeg čvora sa 48 čvorova, zarad kontrolisanja granulacije unutrašnjih čvorova. Stoga se očekuje dalji razvoj novih struktura podataka

zasnovanih na radiks-stablima koje će moći iskoristiti pun potencijal modernog hardvera.

Literatura

[1] *Announcing Rust 1.0*. on-line at: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html. 2015.

[2] *Apache Ignite*. on-line at: https://ignite.apache.org/.

[3] Matt Asay. *Why AWS loves Rust, and how we'd like to help*. on-line at: https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/. 2020.

[4] Rudolf Bayer i Edward M. McCreight. „Organization and Maintenance of Large Ordered Indices". U: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET '70. Houston, Texas: Association for Computing Machinery, 1970, str. 107–141. ISBN: 9781450379410. DOI: 10.1145/1734663.1734671. URL: https://doi.org/10.1145/1734663.1734671.

[5] Matthias Boehm i dr. „Efficient in-memory indexing with generalized prefix trees". U: *Datenbanksysteme für Business, Technologie und Web (BTW)*. Ur. Theo Härder i dr. Bonn: Gesellschaft für Informatik e.V., 2011, str. 227–246.

[6] Danny Cohen. „On Holy Wars and a Plea for Peace". U: *Computer* 14.10 (1981), str. 48–54. DOI: 10.1109/C-M.1981.220208.

[7] The Unicode Consortium. *The Unicode Standard*. on-line at: https://www.unicode.org/versions/latest/.

[8] Jonathan Corbet. *Trees I: Radix trees*. on-line at: https://lwn.net/Articles/175432/. 2006.

[9] *Criterion.rs Documentation*. on-line at: https://bheisler.github.io/criterion.rs/book/criterion_rs.html.

[10] *crosvm*. on-line at: https://opensource.google/projects/crosvm.

[11] Manish Goregaokar. *Fearless Concurrency in Firefox Quantum*. on-line at: https://blog.rust-lang.org/2017/11/14/Fearless-Concurrency-In-Firefox-Quantum.html. 2017.

[12] Christopher Hughes. „Single-Instruction Multiple-Data Execution". U: *Synthesis Lectures on Computer Architecture* 10 (Maj 2015), str. 1–121. DOI: 10.2200/S00647ED1V01Y201505CAC032.

[13] *IoT Edge Security Daemon edgelet*. on-line at: https://github.com/Azure/iotedge/tree/master/edgelet.

[14] Changkyu Kim i dr. „FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs". U: *Proceedings of the 2010 ACM SIGMOD Intenational Conference on Management of data* (2010), str. 339–350.

[15] Steve Klabnik. *The History of Rust*. on-line at: https://dl.acm.org/doi/10.1145/2959689.2960081. 2016.

[16] Steve Klabnik i Carol Nichols. *Fearless Concurrency*. on-line at: https://doc.rust-lang.org/book/ch16-00-concurrency.html. 2019.

[17] Steve Klabnik i Carol Nichols. *Functional Language Features: Iterators and Closures*. on-line at: https://doc.rust-lang.org/book/ch13-00-functional-features.html. 2019.

[18] Viktor Leis, Alfons Kemper i Thomas Neumann. „The adaptive radix tree: ARTful indexing for main-memory databases". U: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, str. 38–49. DOI: 10.1109/ICDE.2013.6544812.

[19] Donald R. Morrison. „PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric". U: *J. ACM* 15.4 (Okt. 1968), str. 514–534. ISSN: 0004-5411. DOI: 10.1145/321479.321481. URL: https://doi.org/10.1145/321479.321481.

[20] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tehn. izv. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, Sept. 2014.

[21] *Redis*. on-line at: https://redis.io/.

[22] *Rust Case Study: Community makes Rust an easy choice for npm*. on-line at: https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.

[23] *Rust contributors*. on-line at: https://thanks.rust-lang.org/.

[24] *Stack Overflow Developer Survey*. on-line at: https://insights.stackoverflow.com/survey/2020. 2020.

[25] *VoltDB Documentation*. on-line at: https://docs.voltdb.com/.

[26] Barry Warsaw i dr. *PEP 1 – PEP Purpose and Guidelines*. on-line at: https://www.python.org/dev/peps/pep-0001/. 2001.

[27] Ashley Williams. *Hello World!* on-line at: https://foundation.rust-lang.org/posts/2021-02-08-hello-world/. 2021.

Biografija autora

Jovan Dmitrović rođen je 17.11.1995. u Gornjem Milanovcu. U rodnom gradu je pohađao OŠ „Momčilo Nastasijević“ i Gimnaziju „Takovski ustanak“. Započeo je studije na Matematičkom fakultetu Univerziteta u Beogradu 2014. godine, na smeru Informatika. Osnovne studije je završio u septembru 2018. godine, nakon čega je upisao master studije na istom fakultetu.