SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Izgradnja binarnog stabla valića kao RRR strukture

Mislav Magerl, Matija Milišić, Mateo Šimonović
Voditelj: *doc. dr. sc. Mirjana Domazet-Lošo*

Zagreb, siječanj 2017.

SADRŽAJ

1. Uvod

U ovom radu opisana je implementacija stabla valića[1] koje se koristi za brze $rank$ i $select$ upite nad ulaznim nizom znakova. Konkretna implementacija bazirana je na RRR strukturi [4].

Brza implementacija metoda $rank$ i $select$ znače da se stablo valića može iskoristiti za pronalaženje podudarajućih nizova, odnosno podnizova. Neke od struktura koji rješavaju isti problem su sufiksna stabla, sufiksna polja. Svaka od ovih struktura, bilo stablo valića, sufiksno stablo ili sufiksno polje ima neke svoje predsnosti i nedostatke u vidu memorije i vremena. U ovom radu obradit će se upravo stablo valića.

U poglavlju 2 opisana je RRR struktura i dan je primjer upita koji se nad tom strukturom mogu zvati. U poglavlju 3 opisano je stablo valića te je također dan primjer upita koji se nad tom strukturom mogu zvati. U poglavlju 4 napisana su mjerenja koja su pokrenuta nad sintetičkim ulazima te nad stvarnim ulazima koji predstavljaju genome nekih virusa, bakterija i organizama. Također napravljena je usporedba s implementacijom studenata s kolegija Bioinformatika akademske godine 2015/2016. U poglavlju 5 opisani su neki detalji implementacije projekta u jeziku C++.

U zadnjem 6. poglavlju dan je kratki zaključak o implementaciji i rezultatima ispitivanja rješenja te je dano nekoliko primjera gdje bi se, u budućnosti, mogla napraviti poboljšanja u odnosu na ovu implementaciju.

2. RRR struktura

RRR struktura je podatkovna struktura koja se koristi za brze $rank$, $select$ te $access$ upite nad binarnim nizom znakova. RRR strukturu su predložili Raman, Raman and Rao [4] te je po njima i dobila ime. Metoda $rank(i)$ odgovara na pitanje koliko se jedinica nalazi u ulaznom nizu do, uključivo, indeksa i. Metoda $select(i)$ odgovara na pitanje na kojem se indeksu u ulaznom nizu nalazi i-ta jedinica. Metoda $access(i)$ odgovara koji znak se nalazi na indeksu i u ulaznom nizu - nula ili jedinica.

2.1. Izgradnja strukture

Ulazni binarni niz podijeljen je u blokove, a blokovi su podijeljeni u superblokove. Broj znakova u bloku (b) te broj blokova u superbloku (f) su parametri strukture. Na slici 2.1 prikazan je primjer RRR strukture s parametrima $b = 5$ te $f = 2$. Ako nisu eksplicitno zadani, parametri se računaju u ovisnosti o veličini ulaznog niza (n): $b = \frac{\log_2 n}{2}$, $f = 2 \log_2 n$. Svaki blok struktura pamti par vrijednosti - klasu (engl. *class*) - c i pomak (engl. *offset*) - o. Klasa je broj jedinica u bloku, a pomak je indeks permutacije jedinica u lookup tablici koja je zadjednička za cijelu strukturu.

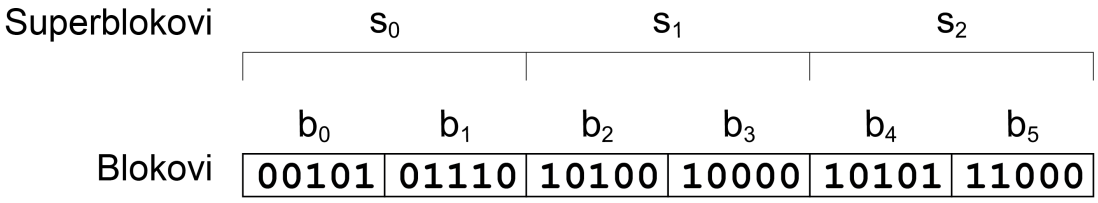
Lookup tablica

Lookup tablica sadrži 2^b redova. Svaki red prikazuje jednu od mogućih različitih blokova veličine b - sve kombinacije nula i jedinica u nizu veličine b. Dodatno, tablica je na taj način izgrađena da su prvo sve permutacije s nula jedinica, zatim sve permutacije s jednom, dvije te naposlijetku b jedinica. Za svaki zapis zapisan je niz kumulativne sume do i-tog elementa u tom bloku. Dio lookup tablice prikazan je na slici 2.2. Vremenska složenost izgradnje lookup tablice je $O(b \cdot 2^b)$, odnosno izraženo preko veličine ulaznog niza $O(n \cdot \log_2 n)$. Memorijska složenost je također $O(b \cdot 2^b)$.

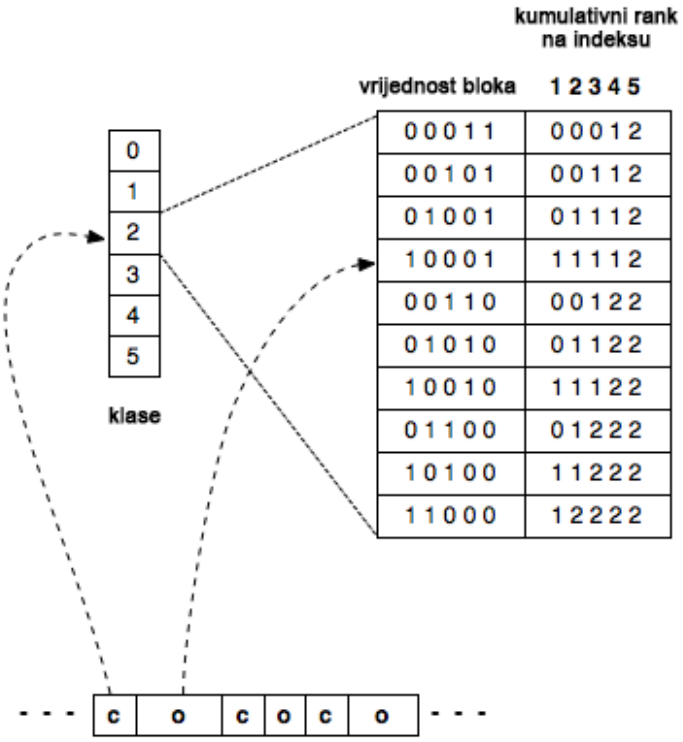
Izgradnja strukture

Nakon izgradnje lookup tablice za svaki blok se računa klasa i pomak te se spremaju u strukturi umjesto originalnog niza. Jedan zapis o klasi veličine je 1B iz razloga što je ukupan broj klasa $log_2 b$ što za vrijednosti b do 16, odnosno vrijednosti n do 10^{10}, iznosi 4 bita. Jedan zapis o pomaku je veličine 2B iz razloga što za svaku klasu c ima $\binom{b}{c}$ zapisa, tj. maksimalno $\binom{b}{\frac{b}{2}}$. Ako se

uzme da je n maksimalno 10^{10}, b iznosi maksimalno 16. Najveći pomak u nekoj klasi tada iznosi $\binom{16}{8} = 12870$ za što nije potrebno više od 2B. Ukupno se, dakle, za jedan blok troši 3B memorije. Ukupna memorijska složenost je $O(\frac{n}{b})$, odnosno izraženo samo preko veličine niza $O(\frac{n}{log_2 n})$ Struktura također pamti i kumulativnu sumu do svakog superbloka. Memorijska složenost te sume je $O(fracnbf)$, odnosno izraženo samo preko veličine niza $O(\frac{n}{(\log_2 n)^2})$.



Slika 2.1: Prikaz blokova i superblokova RRR strukture



Slika 2.2: Prikaz lookup tablice unutar RRR strukture

2.2. Operacije nad strukturom

U sljedećim odlomcima bit će opisane implementacije operacija $rank$, $select$ i $access$ nad RRR strukturom.

2.2.1. Rank

U RRR strukturi postoje dvije rank operacije - $rank_0(i)$ te $rank_1(i)$. Operacija $rank_0(i)$ vraća broj nula do, uključivo, indeksa i, a $rank_1(i)$ broj jedinica do indeksa. Jedna metoda može se vrlo jednostavno izraziti pomoću druge. Tako implementacija metode $rank_0(i)$ izgleda ovako:

```
Funkcija rank0(i)
    vrati i + 1 - rank1(i);
```

Algoritam 1: Pseudokod metode $rank_0$

Operacija $rank_1(i)$ se računa na sljedeći način:

1. Izračuna se indeks superbloka te bloka u kojem se pozicija i nalazi
2. Izračuna se suma svih blokova do bloka u kojem se pozicija i nalazi
3. Pomoću lookup tablice se na sumu dodaje suma brojeva do pozicije i

Pseudokod metode $rank_1$:

```
Funkcija rank1(i)
    indexBloka = i / b;
    indexSuperbloka = i / (b*f);
    rez = kumulativnaSumaSuperbloka(indexSuperbloka - 1);
    za i = indexSuperbloka * f ..  indexBloka čini
        rez += klasa[i];
    kraj
    c = klasa[indexBloka];
    o = pomak[indexBloka];
    rez += lookup[c][o][index - indexBloka * b];
    vrati rez;
```

Algoritam 2: Pseudokod metode $rank_1$

Složenost rank operacija je $O(f)$, odnosno $O(log_2 n)$.

Primjer izvođenja operacije $rank_1$:

Tražimo na primjeru slike 2.1 $rank_1(25)$. Ulazni niz, blokovi i superblokovi su 0-indeksirani. Parametri RRR-a su $b = 5$ i $f = 2$. U prvom koraku računamo $indexBloka = \frac{25}{5} = 5$. U sljedećem računamo $indexSuperbloka = \frac{25}{5 \cdot 2} = 2$ (ovdje koristimo cjelobrojnu aritmetiku). U variablu

rez dodajemo kumulativnu sumu do superbloka 1. Ona iznosi 8. Zatim u petlji s varijablom i idemo od $indexSuperbloka \cdot f = 4$ do $indexBloka = 5$, isključivo. U toj petlji idemo samo s $i = 4$ te u varijablu rez dodajemo klasu četvrtog bloka koja iznosi 3. Sada je vrijednost varijable rez jednaka 11. Naposlijetku čitamo klasu i indeks traženog bloka koje iznose: $c = 2$, $o = 9$. Dodajemo $rez+ = lookup[2][9][25 - 5 * 5]$, što možemo sa slike 2.2 iščitati da iznosi 1. Konačna vrijednost u operacija $rank_1$ vraća je 12.

2.2.2. Select

Kao i za rank operaciju i select operacija ima svoje dvije varijante - $select_0(i)$ te $select_1(i)$ koje vraćaju indeks i-tog pojavljivanja nule odnosno jedinice. Te dvije metode mogu se napisati općenito, jednom funkcijom koja kao parametar prima boolean parametar koji određure traži zove li se $select_0$ ili $select_1$. Operacija $select_1(i)$ računa se na sljedeći način:

1. Pronađi binarnim pretraživanjem superblok u kojem se nalazi tražena jedinica
2. U brojač pohrani sumu jedinica do prethodnog superbloka
3. Dodaj u brojač blokove sve dok ne dođeš do bloka koji prethodi bloku u kojem je tražena jedinica
4. Binarnim pretraživanjem pronađi poziciju tražene jedinice u bloku.

Pseudokod:

```
Funkcija select(i, tip)
    indexSuperbloka = pronađiSuperblok(i, tip);
    brojač = kumulativnaSumaSuperbloka(indexSuperbloka - 1,
    tip);
    indexBloka = indexSuperbloka * f;
    dok brojač < i čini
        brojač = brojač + dohvatiKlasuBloka(indexBloka, tip);
        indexBloka = indexBloka + 1;
    kraj
    indexBloka = indexBloka - 1;
    brojač = brojač - dohvatiKlasuBloka(indexBloka, tip);
    indexUBloku = pronađiIndeksUBloku(i - brojač, tip);
    vrati indexBloka * b + indexUBloku;
```

Algoritam 3: Pseudokod metode $select$

Metode `pronađiSuperblok`, `kumulativnaSumaSuperbloka`, `dohvatiKlasuBloka` i `pronađiIndeksUBloku` primaju kao parametar `tip`. Varijabla `tip` je istinita ako se traži jedinica, odnosno lažna ako se traži nula. Ovisno o tom parametru se metode i ponašaju. Metode `pronađiSuperblok` i `pronađiIndeksUBloku` rade binarno pretraživanje kako bi pronašle indekse. Složenost možemo rastaviti na nekoliko dijelova. Vremenska složenost traženja superbloka je $O(\frac{n}{bf})$, tj. $O(\frac{n}{(\log_2 n)^2})$. Zatim se traži indeks bloka unutar superbloka - $O(f)$, tj. $O(\log_2 n)$. Složenost dohvaćanja klase je konstantna, $O(1)$. Složenost traženja indeksa u bloku je $O(\log_2 b)$, odnosno $O(\log_2(\log_2 n))$.

Primjer izvođenja operacije $select_1$:

Tražimo na primjeru slike 2.1 $select_1(12)$. Primjetimo da bi rezultat trebao biti 25 jer je $rank_1(25) = 12$. Ulazni niz, blokovi i superblokovi su 0-indeksirani. Parametri RRR-a su $b = 5$ i $f = 2$. U prvom koraku binarno pretražujemo superblok u kojem se nalazi dvanaesta jedinica. Dobivamo $indexSuperbloka = 2$. U varijablu $brojač$ spremamo kumulativnu sumu do superbloka s indeksom 1 - $brojač = 8$. Varijabla $indexBloka = 2 \cdot 2 = 4$. U *dok* petlji se brojač poveća na vrijednost 11, zatim na vrijednost 13. Uz brojač, varijabla $indexBloka$ se poveća najprije na 5 pa na 6. Na izlasku iz petlje, $indexBloka$ i $brojač$ se pomiću ulijevo. Sada je $indexBloka = 5$, a $brojač = 11$. Metoda $pronaiIndeksUBloku(12 - 11)$ radi binarno pretraživanje nad lookup tablicom i vraća vijednost $indexUBloku = 0$. Konačno, funkcija vraća vijednost $5 \cdot 5 + 0 = 25$.

2.2.3. Access

Metoda $access(i)$ vraća koji se znak nalazi na danoj poziciji - nula ili jedan. Metoda je vrlo jednostavna. Jednostavno se računa razlika između $rank1$ od trenutne pozicije i prethodne pozicije.

```
Funkcija access(i)
    ako i==0 onda
        vrati rank1(i);
    inače
        vrati rank1(i) - rank1(i - 1);
    kraj
```

Algoritam 4: Pseudokod metode $access$

Primjer izvođenja operacije $access$:

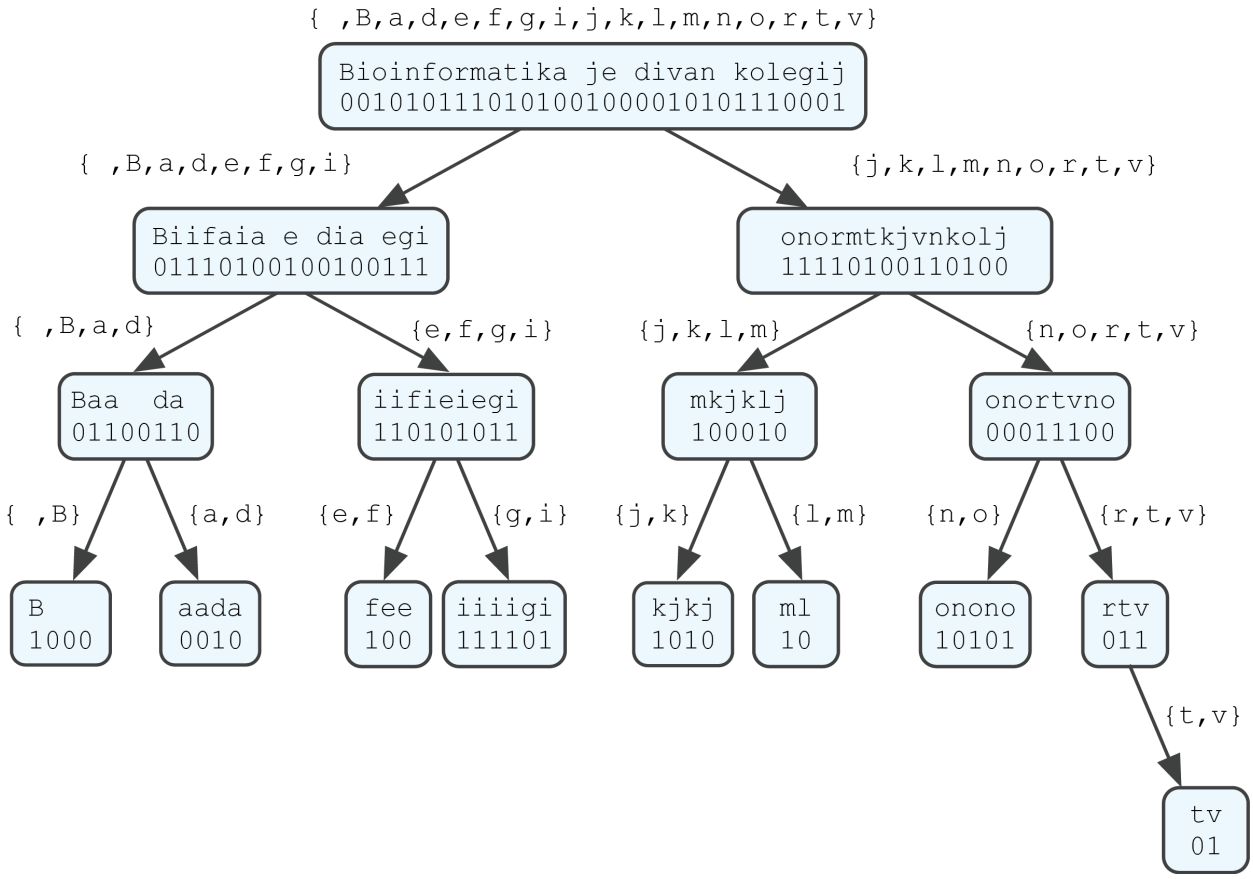
Tražimo na primjeru slike 2.1 $access_1(25)$. Ulazni niz, blokovi i superblokovi su 0-indeksirani. Parametri RRR-a su $b = 5$ i $f = 2$. Rezultat je $rank_1(25) - rank_1(24)$ što je jednako 1.

3. Stablo valića

Stablo valića (engl. *Wavelet Tree*) je struktura podataka koja omogućava efikasno spremanje niza znakova nad proizvoljnom abecedom u memoriji računala i efikasnog odgovaranja na određena pitanja nad istim. Iako originalno predstavljeno od tvoraca strukture Grossija, Gupte i Vittera [2] za rad isključivo nad nizovima, stablo valića se pokazalo kao vrlo efikasna struktura podataka za predstavljanje permutacija i rešetke točaka i pronašlo je primjenu u raznim područjima, od obrade i indeksiranja teksta, do teorije grafova i geometrije. [3]

Stablo valića je podatkovna struktura koja ulazni niz znakova kodira kao stablo binarnih nizova. Na taj je način ostvarena sažeta struktura podataka (engl. *succinct data structure*), odnosno to je struktura podataka koja koristi memorije blisko donjoj granici definiranoj teorijom informacije (konkretno, $N + o(N)$ bitova memorije gdje je N teorijski optimum za pohranu podataka), a i dalje nudi efikasnu implementaciju odgovora na razne upite o sadržanim podacima.

Na slici 3.1 prikazan je primjer izgrađenog stabla valića za ulazni niz znakova "Bioinformatika je divan kolegij". Stablo je binarno i balansirano, a gradi se rekurzivnim postupkom počevši od korijena stabla koji predstavlja cijeli ulazni niz pa do listova koji predstavljaju jedinstveno označene znakove. U poglavlju 3.1 detaljnije je opisan postupak izgradnje stabla na danom primjeru. U praksi postoje dvije moguće implementacije izgradnje stabla valića: prva koja gradi stablo dok svako slovo u ulaznoj abecedi ne postane list s jedinstvenim binarnim kodom, a druga koja gradnju stabla zaustavi kada čvor sadrži samo dva različita znaka jer ih je tada binarnim nizom već moguće razlikovati. U ovom je radu korištena druga implementacija radi uštede memorije zbog nepotrebnog spremanja dodatne dubine stabla. Važno je primjetiti da o duljini ulaznog niza ovisi duljina vektora bitova u svakom čvoru stabla, dok dubina stabla ovisi samo o veličini abecede ulaznog niza. Također, bitno je napomenuti da su znakovi u čvorovima prikazani radi razumijevanja postupka izgradnje stabla, ali se u strukturi ne pamte. Stablo valića omogućuje efikasnu implementaciju operacija koje se u literaturi pojavljuju kao rank, select i access, a njihova važnost i implementacijski detalji objašnjeni su u poglavlju 3.2.



Slika 3.1: Stablo valića za ulazni niz "Bioinformatika je divan kolegij"

3.1. Izgradnja stabla

Stablo valića gradi se rekurzivnim postupkom koji kreće od niza znakova $S = s_1 s_2 \dots s_n$, uz $s_i \in \Sigma$, gdje je Σ abeceda (engl. *alphabet*) ulaznog niza znakova. [3] Pseudokod algoritma za izgradnju stabla valića prikazan je u nastavku. Algoritam kreće od korijena stabla i rekurzivno se spušta do listova.

Funkcija *izgradiStabloValića(*S*,* Σ*)*
  ako $|\Sigma| \leqslant 2$ **onda**
    vrati;
  kraj
  $\Sigma_1, \Sigma_2 \leftarrow$ podijeliNaDvaJednakaDijela(Σ);
  pridijeli svim znakovima $s_i \in S$ kod 0 **ako** $s_i \in \Sigma_1$, **inače** kod 1;
  $S_1, S_2 \leftarrow$ grupiraj znakove iz S prema kodu;
  izgradiStabloValica(S_1, Σ_1);
  izgradiStabloValica(S_2, Σ_2);

Algoritam 5: Izgradnja stabla valića

U nastavku slijedi opis izgradnje stabla za primjer sa slike 3.1. Na samom početku, ulazni niz znakova je $S = Bioinformatika\ je\ divan\ kolegij$. Abeceda niza S je

$$\Sigma = \{\ , B, a, d, e, f, g, i, j, k, l, m, n, o, r, t, v\}$$

. Pritom je duljina ulaznog niza znakova $|S| = 31$ znak, a veličina abecede $|\Sigma| = 17$ znakova. Znakovi abecede su poredani uzlazno prema ASCII kodu.

Zatim se abeceda Σ dijeli na dva približno jednaka dijela $\Sigma_1 = \{, B, a, d, e, f, g, i\}$ i $\Sigma_2 = \{j, k, l, m, n, o, r, t, v\}$. Sljedeći je korak kodirati ulazni niz znakova S na način da se svakom znaku ulaznog niza s_i pridijeli kod 0 ako je $s_i \in \Sigma_1$, te kod 1 ako je $s_i \in \Sigma_2$. Rezultat ove operacije je sljedeći binarni niz $0010101110101001000010101110001$.

Na osnovi dobivenog binarnog niza stvaraju se nizovi znakova S_1 i S_2 tako da se znakovi kodirani kodom 0 izdvoje u niz S_1, a znakovi kodirani kodom 1 u niz S_2. Na ovom su primjeru na taj način dobiveni sljedeći nizovi $S_1 = Biifaia\ e\ dia\ egi$ te $S_2 = onormtkjvnkolj$. S tim se nizovima, zajedno s odgovarajućim abecedama Σ_1 i Σ_2, postupak ponavlja za lijevo i desno dijete u stablu.

Poseban je slučaj kada postupak dolazi do lista i ne stvara novo dijete u stablu, a to se događa kada je $|\Sigma| <= 2$. Razlog zbog kojeg nije potrebno stvarati nove čvorove u stablu u tom slučaju je zbog toga što se svaki znak ulaznog niza može jedinstveno kodirati samo s oznakama 0 i 1. Jedan od čvorova u stablu iz primjera sa slike 3.1 kod kojeg će se dogoditi taj poseban slučaj je čvor čiji je ulazni niz $S = aada$ te abeceda $\Sigma = \{a, d\}$. Za promatrani čvor podijeli se abeceda na dva jednaka dijela i dobije $\Sigma_1 = \{a\}$ i $\Sigma_2 = \{d\}$ te se jednostavno čvoru pridružuje kodirani zapis $S_{kod} = 0010$ i ne generiraju djeca trenutnog čvora.

3.2. Operacije nad stablom

Stablo valića omogućava efikasno izvođenje tri osnovne operacije:

1. $rank(i, c)$ - broj pojavljivanja znaka c do uključivo i-te pozicije u ulaznom nizu znakova
2. $select(i, c)$ - indeks i-tog znaka c u ulaznom nizu znakova
3. $access(i)$ - znak na i-toj poziciji u ulaznom nizu znakova

U idućim poglavljima dani su opisi svake od navedenih operacija, njihov pseudokod te je pokazan primjer njihovog izvođenja nad stablom izgrađenim u poglavlju 3.1. Važno je napomenuti nekoliko implementacijskih detalja iz kojih slijede pseudokodovi u narednim poglavljima. Svaki je čvor u grafu definiran strukturom koja u sebi sadrži reference na djecu kojima se u nastavku pristupa sintaksom `čvor.lijevo` i `čvor.desno` te referencu na čvor roditelj kojem se pristupa s `čvor.roditelj`. Svi indeksi počinju od nule.

3.2.1. Rank

Pseudokod operacije $rank(i, c)$ dan je u nastavku u algoritmu 6. To je operacija nad izgrađenim stablom valića koja odgovara na pitanje koliko je pojavljivanja znaka c do i-te pozicije u ulaznom nizu znakova. Operacije `rank1` i `rank0` koje se pojavljuju u pseudokodu su operacije nad binarnim vektorom kodova i odgovaraju operacijama definiranim u poglavlju 2.2.1.

```
Funkcija rank(i, c)
    trenutniČvor = korijenStabla;
    pozicija = i;
    dok trenutniČvor != NULL čini
        ako c ∈ trenutniČvor.desno onda
            pozicija = rank1(pozicija) - 1;
            trenutniČvor = trenutniČvor.desno;
        inače
            pozicija = rank0(pozicija) - 1;
            trenutniČvor = trenutniČvor.lijevo;
        kraj
        ako pozicija == -1 onda
            izađi;
        kraj
    kraj
    vrati pozicija + 1;
```

Algoritam 6: Pseudokod $rank$ operacije nad stablom valića

3.2.2. Select

Pseudokod operacije $select(i, c)$ prikazan je u nastavku algoritmom 7. To je operacija nad izgrađenim stablom valića koja odgovara na pitanje koji je indeks i-tog znaka c u ulaznom nizu znakova. U pseudokodu se pojavljuju operacije označene kao `select0` i `select1`, a radi se o operacijama nad binarnim vektorom kodova i odgovaraju operacijama koje su prethodno definirane u poglavlju

2.2.2.

```
Funkcija select(i, c)
    trenutniČvor = čvorZaZnak(c);
    pozicija = i;
    ako kôd(trenutniČvor) == 0 onda
        pozicija = select0(pozicija);
    inače
        pozicija = select1(pozicija);
    kraj
    dok trenutniČvor != korijenStabla čini
        ako je trenutniČvor lijevo dijete onda
            pozicija = trenutniČvor.roditelj.select0(pozicija+1);
        inače
            pozicija = trenutniČvor.roditelj.select1(pozicija+1);
        kraj
        trenutniČvor = trenutniČvor.roditelj;
    kraj
    vrati pozicija;
```

Algoritam 7: Pseudokod $select$ operacije nad stablom valića

3.2.3. Access

Pseudokod operacije $access(i)$ prikazan je algoritmom 8. To je operacija nad izgrađenim stablom valića koja odgovara na pitanje koji je znak na i-toj poziciji u ulaznom nizu znakova. U samom algoritmu koriste se funkcije `access` te `rank0` i `rank1`, a odnose se na funkcije prethodno

definirane u poglavljima 2.2.3 i 2.2.1.

```
Funkcija access(i)
    trenutniČvor = korijenStabla;
    pozicija = i;
    dok trenutniČvor != NULL čini
        ako access(pozicija) == 0 onda
            pozicija = rank0(pozicija);
            ako trenutniČvor.lijevo == NULL onda
                vrati prviElement(Σ_v);
            kraj
            trenutniČvor = trenutniČvor.desno;
        inače
            pozicija = rank1(pozicija);
            ako trenutniČvor.lijevo == NULL onda
                vrati zadnjiElement(Σ_v);
            kraj
            trenutniČvor = trenutniČvor.desno;
        kraj
    kraj
    vrati 0;
```

Algoritam 8: Pseudokod $access$ operacije nad stablom valića

4. Rezultati

U ovom poglavlju prikazane su vremenska i memorijska karakteristika stabla valića na sintetičkim i stvarnim skupovima podataka. U tu svrhu napravljena su 24 sintetička skupa podataka s veličinama od 10^2 do 10^7 znakova i veličinama abecede 4, 16 i 28. Stvarni skupovi podataka preuzeti su sa stranice `https://www.ncbi.nlm.nih.gov/` i uključuju zapise genoma nekih virusa, bakterija i organizama.

Napravljena evaluacija radila je usporedbu između ove implementacije (lijevi stupac) i implementacije studenata iz akademske godine 2015/2016 (desni stupac). Tablica 4.1 prikazuje vrijeme izgradnje stabla valića (tc) u sekundama te prosječna vremena izvođenja operacija $rank$, $select$ i $access$ u mikrosekundama za obje implementacije. Stupac označen s n predstavlja broj znakova u skupu podataka, $|\Sigma|$ veličinu abecede. Tablica 4.2 prikazuje potrošnju memorije pri izgradnji stabla i cijelog procesa za obje implementacije. Tablica 4.3 prikazuje vremena izgradnje stabla i izvršavanja upita za obje implementacije. Tablica 4.4 prikazuje potrošnju memorije pri izgradnji stabla i cijelog procesa za obje implementacije.

Svi testovi pokazuju da je ova implementacija vremenski i memorijski kvalitetnija. Velika razlika može se primijetiti u brzini izgradnje stabla valića, te brzinama izvođenja $select$ i $access$ upita. Također, znatna je ušteda memorije koju zauzimaju sami proces i struktura tijekom i nakon izgradnje stabla valića.

Tablica 4.1: Vrijeme izgradnje stabla i izvršavanja upita - sintetičke datoteke

n	$\|\Sigma\|$	$tc\ (s)$		$rank\ (\mu s)$		$select\ (\mu s)$		$access\ (\mu s)$	
10^2	4	0.000	0.000	0.6	0.4	0.7	1.1	0.3	1.0
	16	0.000	0.000	1.0	0.6	1.4	2.1	0.7	1.8
	28	0.001	0.000	1.3	0.7	1.8	2.5	0.9	2.1
10^3	4	0.001	0.001	0.5	0.5	0.9	1.4	0.4	1.4
	16	0.002	0.002	1.1	1.0	1.6	2.6	0.9	2.6
	28	0.003	0.002	1.4	1.1	2.1	3.1	1.1	3.0
10^4	4	0.008	0.012	0.5	0.7	1.1	1.7	0.5	1.9
	16	0.024	0.032	1.2	1.3	2.0	3.3	1.0	3.5
	28	0.023	0.030	1.4	1.5	2.4	3.9	1.2	4.2
10^5	4	0.076	0.312	0.6	0.8	1.2	2.0	0.5	2.3
	16	0.173	0.702	1.2	1.5	2.3	3.9	1.2	4.4
	28	0.224	0.759	1.6	1.8	2.9	4.6	1.4	5.1
10^6	4	0.743	4.705	0.6	1.0	1.3	2.4	0.6	2.7
	16	1.718	9.843	1.4	1.8	2.6	4.6	1.4	5.2
	28	2.284	11.539	1.7	2.2	3.5	5.6	1.8	6.2
10^7	4	7.391	131.879	0.8	1.2	1.7	2.9	0.9	3.4
	16	18.045	286.005	1.9	2.6	3.8	6.0	2.2	6.7
	28	22.087	327.876	2.5	3.2	4.5	7.1	2.6	7.9

Tablica 4.2: Potrošnja memorije pri izgradnji stabla - sintetičke datoteke

n	$\|\Sigma\|$	Proces (MB)		Struktura (MB)	
10^2	4	1.57	1.59	0.18	0.18
	16	1.59	1.59	0.18	0.18
	28	1.6	1.59	0.18	0.18
10^3	4	1.58	1.6	0.19	0.19
	16	1.6	1.6	0.19	0.19
	28	1.63	1.6	0.19	0.19
10^4	4	1.62	1.67	0.2	0.2
	16	1.68	1.68	0.2	0.2
	28	1.72	1.69	0.2	0.2
10^5	4	1.81	2.16	0.04	0.3
	16	2.01	2.35	0.3	0.55
	28	2.13	2.29	0.55	0.55
10^6	4	4.26	6.81	1.79	1.8
	16	5.45	8.19	2.81	3.51
	28	6.15	8.72	3.59	3.92
10^7	4	31.8	64.69	14.3	22.29
	16	41.4	70.86	22.7	30.64
	28	45.27	75.2	25.75	34.11

Tablica 4.3: Vrijeme izgradnje stabla i izvršavanja upita - FASTA datoteke

skup podataka	n	$\|\Sigma\|$	$tc\ (s)$		$rank\ (\mu s)$		$select\ (\mu s)$		$access\ (\mu s)$	
HPV	7313	4	0.006	0.009	0.6	0.7	1.0	1.7	0.5	1.8
HIV	9181	4	0.007	0.010	0.5	0.7	1.0	1.7	0.5	1.9
Camelpox	205,719	4	0.152	0.598	0.5	0.8	1.2	2.1	0.6	2.4
Tuberculosis	4,411,532	4	3.253	56.304	0.7	1.1	1.5	3.0	0.9	3.1
Salmonella	4,791,961	9	7.394	50.783	1.2	1.6	2.1	141.6	1.9	5.7
Coli	5,231,428	12	7.775	55.846	1.4	1.9	2.6	8.2	1.7	5.7
Human21	33,543,332	5	33	532	1.5	1.9	3.6	15.8	1.6	4.6
Human3	198,295,559	9	263.215	-	1.7	-	3.0	-	2.1	-

Tablica 4.4: Potrošnja memorije pri izgradnji stabla - FASTA datoteke

skup podataka	n	$\|\Sigma\|$	Proces (MB)		Struktura (MB)	
HPV	7813	4	1.61	1.65	0.2	0.2
HIV	9181	4	1.63	1.66	0.2	0.2
Camelpox	205,719	4	2.13	2.54	0.3	0.41
Tuberculosis	4,411,532	4	14.82	25.56	6.78	7.56
Salmonella	4,791,961	9	26.34	46.38	11.14	9.11
Coli	5,231,428	12	25.84	41.52	14.19	12.59
Human21	33,543,332	5	131.57	238.55	50.38	99.05
Human3	198,295,559	9	782.22	-	190.16	-

5. Implementacija

Implementacija stabla valića i RRR strukture napisana je u programskom jeziku C++. Pomoćne skripte koje služe za generiranje sintetičkih testnih primjera i za obradu stvarnih podataka napisane su u jeziku Python.

Dvije glavne datoteke u kojima je napisan kod su u direktoriju `src`:

1. `RRR.cpp`. U ovoj datoteci napisane su implementacije metoda RRR strukture. Metode razreda RRR definirane su u *header* datoteci `/include/RRR.h`. Konstruktor razreda RRR prima kao parametar vektor boolean vrijednosti. U konstruktoru se prvo gradi lookup tablica, a na temelju dobivenog vektora konstruira se par (klasa, pomak) za svaki blok u ulaznom nizu. Metode $rank$, $select$ i $access$ implementirane su kao što je opisano u poglavlju 2.

2. `WaveletTree.cpp`. U ovoj datoteci napisane su implementacije metoda stabla valića. Metode razreda `WaveletTree` definirane su u *header* datoteci `/include/WaveletTree.h`. U istim datotekama definiran je i razred `WaveletTreeNode` koji predstavlja jedan čvor u stablu valića. Taj čvor koristi razred `RRR` kako bi odgovarao na tražene upite. U konstruktoru razreda `WaveletTree` rekurzivno se gradi struktura pomoću čvorova. Čvor u svom konstruktoru generira druge čvorove sve dok je veličina abecede veća od 2. Metode $rank$, $select$ i $access$ implementirane su kao što je opisano u poglavlju 3.

U direktoriju `unittest` napisani su *Unit* testovi za razrede `RRR` i `WaveletTree`. Testovi se pokreću drugačije ovisno o operacijskom sustavu. Testovi su napisani pomoću okoline za testiranje *googletest*. Moguće je da prilikom kloniranja repozitorija neće biti moguće automatski pokrenuti testove, već će biti potrebno preuzeti i pokrenuti neke skripte dostupne na stranicama *googletest*-a:

1. `git clone https://github.com/google/googletest`
2. `cd googletest`
3. `cmake googletest && make`
4. `cp libgtest.a PROJECT/code/bin/ubuntu`

Za pokretanje testova služi skripta `bin/<OS>/run_tests.sh`.

U direktoriju `other` nalaze se izvorni kodovi implementacije studenata akademske godine 2015/2016[5] koji su preuzeti s njihovog repozitorija, a služe vremenskoj i memorijskoj usporedbi rješenja.

6. Zaključak

U ovom radu napravljena je implementacija stabla valića koristeći RRR strukturu. Stablo valića koristi se za brze upite tipa koliko ima znakova c do pozicije i u ulaznom nizu te na kojoj se poziciji u ulaznom nizu nalazi i-ti znak c. Prvi izazov bio je implementirati RRR strukturu koja ima iste upite kao i stablo valića s razlikom što je ulaz binarni niz znakova. Sljedeći izazov bio je implementirati stablo valića koristeći RRR sturkturu kao osnovu za čvor stabla valića.

U radu je napravljena vremenska i memorijska usporedba s implementacijom iste strukture studenata iz akademske godine 2015/2016. Rezultati usporedbe govore da je ova implementacija barem jednako brza kao prošlogodišnja za upite $rank$, $select$ i $access$ te višestruko brža u vremenu izgradnje stabla. Također, memorijska potrošnja je manja od prošlogodišnje.

Jedan od mogućih razloga manje memorijske potrošnje jest ograničavanje veličine niza na $2 \cdot 10^9$. Do tog ograničenja je došlo zbog napomene da rezultati moraju biti prikazani za ulaze do 10^7. Potencijalno poboljšanje ove strukture bilo bi proširenje na veće nizove. Također, neke implementacije nisu do kraja optimizirane. Primjerice u RRR strukturi $rank$ operacija trenutno radi u složenosti $O(n)$, gdje je n veličina ulaznog niza, a mogla bi raditi u $O(1)$. Takva implementacija bi svakako zahtjevala više memorije.

Još jedna stvar koja bi mogla ubrzati izgradnju te smanjiti potrošnju memorije bila bi dijeljena lookup tablica RRR strukture unutar stabla valića. Ta optimizacija zahtjevala bi veće promjene na trenutnoj implmenetaciji pa zato i nije implementirana.

7. Literatura

[1] Alex Bowe. *Multiary Wavelet Trees in Practice*. School of Computer Science and Information Technology, RMIT University, 2010. URL https://github.com/alexbowe/wavelet-paper.

[2] Roberto Grossi, Ankur Gupta, i Jeffrey Scott Vitter. High-order entropy-compressed text indexes. U *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, stranice 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0-89871-538-5. URL http://dl.acm.org/citation.cfm?id=644108.644250.

[3] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[4] Rajeev Raman, Venkatesh Raman, i Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *CoRR*, abs/0705.0552, 2007. URL http://arxiv.org/abs/0705.0552.

[5] Denis Čaušević i Hajrudin Ćoralić. Izgradnja binarnog stabla valića (eng. wavelet tree) kao rrr strukture. 2016. URL https://github.com/Vaan5/Bioinformatics-Construction-of-binary-wavelet-trees-using-RRR-structur