SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

BIOINFORMATIKA - PROJEKT

Izgradnja binarnog stabla valića (*eng. wavelet tree*) kao RRR strukture

Denis Čaušević

Hajrudin Ćoralić

Zagreb, siječanj, 2016

Sadržaj

1. Uvod

Količina digitalno dostupnih tekstualnih podataka drastično je porasla tijekom proteklih par desetljeća. Kako su istovremeno tekstualni dokumenti (sekvence) postajali sve veći, rasla je potreba za učinkovitim algoritmima njihovog pretraživanja.

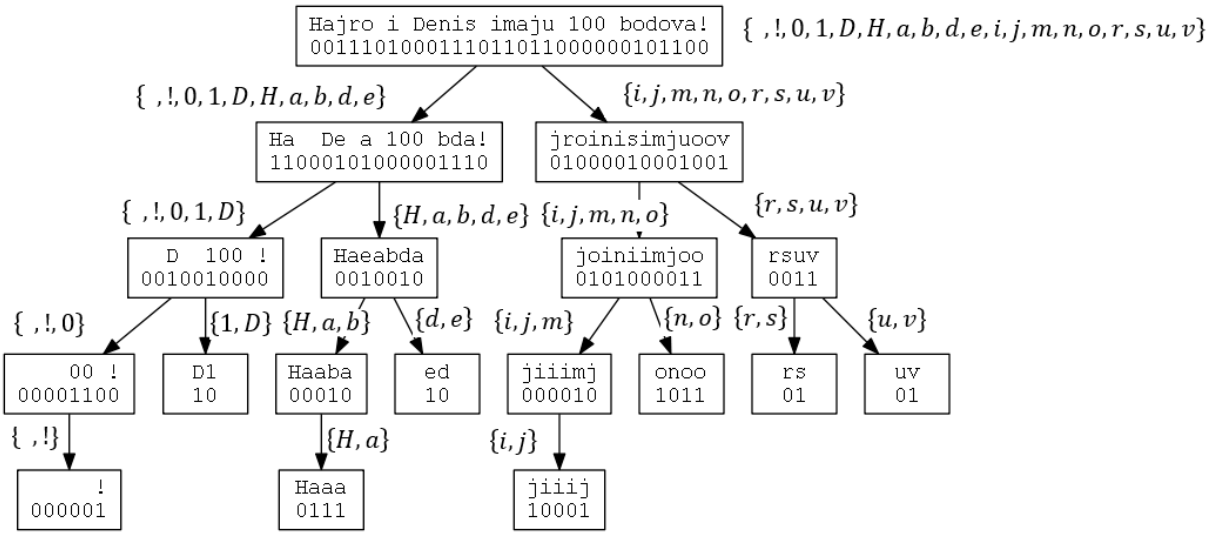
Najprije su se pojavila sufiksna stabla[11] koja su omogućila učinkovito rješavanje problema podudaranja uzoraka. Glavni nedostatak sufiksnih stabala bili su veoma veliki memorijski zahtjevi. Budući da tekstualne sekvence danas postaju sve veće i veće, velika važnost pridijeljena je istraživanju sažimajućih (*eng. succinct*) podatkovnih struktura. Prvi korak bila je pojava sufiksnih polja[12], koja su predstavljala svojevrstan kompromis između vremenskih i memorijskih zahtjeva, te su zahtijevali manje memorijskih resursa u odnosu na sufiksna stabla.

S ciljem ostvarivanja učinkovitog pretraživanja teksta, ubrzo su se javili samostojni indeksi, koji su između ostalog omogućavali brzo prebrojavanje uzoraka, te podudaranje uzoraka u tekstualnim sekvencama. Za razliku od potpunih indeksa, samostojni indeksi vrše kompresiju teksta, te originalni tekst mogu rekonstruirati iz komprimiranog oblika. Prvi samostojni indeks bio je FM-indeks[14]. On se temelji na Burrows-Wheelerovoj[13] transformaciji teksta, te omogućava brzu implementaciju podudaranja uzoraka, za koju koristi $rank$ upite, koji određuju broj pojavljivanja traženog znaka do zadane pozicije u tekstu. Kako bi se podudaranje uzoraka moglo brzo izvoditi, potrebna je vremenski učinkovita implementacija $rank$ upita. Naivna implementacija zahtijevala bi sekvencijalni prolazak kroz tekst (linearna vremenska složenost). Međutim, željeni upiti mogu se realizirati u logaritamskom vremenu primjenom stabla valića[2][3][7], koje ulaznu sekvencu kodira kao stablo binarnih vektora. Ukoliko se stablo implementira kao RRR struktura[4][8], konačna vremenska složenost izvođenja $rank$ upita iznosila bi $O(\log|\Sigma|)$, gdje je Σ abeceda tekstualne sekvence budući da se navedeni upiti mogu nad RRR strukturom izvesti u vremenu $O(1)$. U narednom poglavlju detaljnije je opisano stablo valića, te operacije koje se nad njime mogu izvoditi. Izloženi algoritmi ilustrirani su na jednostavnom primjeru u istom poglavlju. Analogan opis za RRR strukturu dan je u trećem poglavlju. Četvrto poglavlje sadrži eksperimentalne rezultate (memorijsko zauzeće te vremensku analizu) za implementaciju opisanu u petom poglavlju.

2. Stablo valića

Stablo valića[2][3][7] (eng. wavelet tree) je podatkovna struktura koja kodira ulazni niz znakova kao stablo binarnih vektora. Navedeno stablo gradi se rekurzivno pri čemu se abeceda niza u trenutnom čvoru stabla dijeli na dva dijela. Svi znakovi iz prvog dijela abecede kodirani su nulama, dok su znakovi drugog dijela abecede kodirani jedinicama. Znakovi koji su zakodirani nulama prosljeđuju se lijevom djetetu, dok se preostali znakovi prosljeđuju desnom djetetu. Listovi stabla obično predstavljaju točno jedan znak abecede. Međutim, stablo se može definirati i tako da se izgradnja zaustavi na prethodnoj razini kada se jednoznačno zna koji će znak biti predstavljen lijevim odnosno desnim djetetom.

Slika 1 prikazuje primjer binarnog stabla valića za ulazni niz „Hajro i Denis imaju 100 bodova!". U ovom stablu nisu građeni dodatni listovi kada se jednoznačno može odrediti znak koji bi njima bio predstavljen. Stablo sa slike koristit će se u narednim poglavljima za opis operacija koje se mogu izvoditi nad stablom valića.



Slika 1 Stablo valića za niz znakova „Hajro i Denis imaju 100 bodova!"

2.1 Izgradnja stabla

Kao što je rečeno u prethodnom poglavlju, stablo se gradi rekurzivno dijeljenjem ulazne abecede Σ na podskupove pridružene djeci trenutnog čvora. Pseudokod koji

opisuje izgradnju stabla prikazan je u nastavku (Tablica 1). Izgradnja stabla kreće od korijenskog čvora te se rekurzivno spušta do listova.

Tablica 1 Izgradnja binarnog stabla valića

```
Neka je Σ abeceda trenutnog čvora, te S niz znakova koji se
želi zakodirati u trenutnom čvoru.


def izgradi(Σ,S):
    Podijeli abecedu Σ na dva jednaka dijela Σ1 i Σ2.
    Kodiraj sve znakove c ∈ Σ1 u nizu S sa nulama. Sve preostale
    znakove c ∈ Σ2 kodiraj sa jedinicama.
    Ako je |Σ| > 2:
        Znakove kodirane nulama grupiraj u niz S1
        izgradi(Σ1,S1)
        Ako je |Σ2| > 1:
            Znakove kodirane jedinicama grupiraj u niz S2
            izgradi(Σ2,S2)
```

Promotrimo izgradnju dijelova stabla za primjer sa slika 1. Najprije razmotrimo što se događa u korijenskom čvoru.

Niz znakova koji želimo zakodirati je $S = Hajro\ i\ Denis\ imaju\ 100\ bodova!$. Promatrani niz dugačak je 31 znak, te ima sljedeću abecedu:

$$\Sigma = \{\ ,!,0,1,D,H,a,b,d,e,i,j,m,n,o,r,s,u,v\}.$$

Pritom je $|\Sigma| = 19$. Znakovi su sortirani prema ASCII kodu. Prvo je potrebno podijeliti abecedu na dva približno jednaka dijela:

$$\Sigma_1 = \{\ ,!,0,1,D,H,a,b,d,e\} \text{ i } \Sigma_2 = \{i,j,m,n,o,r,s,u,v\}$$

Podjela je napravljena tako da se promatraju indeksi prvog i posljednjeg elementa u Σ, naime 0 i 18, te se zatim odredi prag kao $\frac{0+18}{2} = 9$. Svi elementi s indeksima iz intervala $[0,9]$ smještaju se u Σ_1, a svi ostali u Σ_2.

Sljedeći korak sastoji se u kodiranju niza $Hajro\ i\ Denis\ imaju\ 100\ bodova!$ sa nulama i jedinicama, tako da se svi znakovi iz Σ_1 označe nulama, a svi znakovi iz Σ_2 jedinicama. Na taj način generira se sljedeći binarni niz $00111010001110110110000000101100$.

Znakovi koji su kodirani nulama prosljeđuju se lijevom djetetu, dok se preostali znakovi prosljeđuju desnom djetetu. Postupak se ponavlja najprije za lijevo dijete sa sljedećim postavkama:

$$S = Ha\ \ De\ a\ 100\ bda!$$

$$\Sigma = \Sigma_1 = \{\ ,!, 0, 1, D, H, a, b, d, e\},$$

a potom analogno i za desno dijete.

Razmotrimo još slučaj kada se neće stvoriti novo dijete. Promotrimo čvor sa sljedećim postavkama:

$$S = onoo$$

$$\Sigma = \{n, o\}$$

Za promatrani čvor ponovno se vrši podjela abecede i binarno kodiranje:

$$\Sigma_1 = \{n\}$$

$$\Sigma_2 = \{o\}$$

$$S_{kodirano} = 1011$$

Budući da je $|\Sigma| = 2$, aktivira se uvjet iz pseudokoda usljed kojeg se ne generiraju djeca ovog čvora.

2.2 Operacije nad stablom

Stablo valića omogućava izvođenje tri osnovne operacije:

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

2.2.1 rank

Pseudokod $rank$ operacije prikazan je u nastavku (Tablica 2). Indeksi i započinju s nulom. U navedenom pseudokodu javljaju se $rank$ operacije nad binarnim vektorom koje će biti objašnjene detaljnije u narednom poglavlju. $rank_{0v}(i)$ računa broj pojavljivanja nula do uključivo i-te pozicije u binarnom vektoru v. $rank_{1v}(i)$ radi na jednak način uz računanje broja pojavljivanja jedinica umjesto nula.

Tablica 2 Rank operacija nad stablom valića

```
Neka v_lijevo i v_desno predstavljaju lijevo odnosno desno dijete
čvora v. Ukoliko v nema desno odnosno lijevo dijete, v_desno
odnosno v_lijevo je jednako NULL. rank_0v odnosno rank_1v
predstavljaju rank operacije nad binarnom (kodiranom)
reprezentacijom čvora v. (Objašnjeno u narednom poglavlju)

def rank(c, i):
    v ← korijenski čvor stabla
    r ← i
    Dok je v != NULL
        Ako v nije korijenski čvor: r ← r − 1
        Ako je c ∈ v_lijevo:
            r ← rank_0v(r)
            v ← v_lijevo
        Inače:
            r ← rank_1v(r)
            v ← v_desno
        Ako je r = 0: vrati 0
    vrati r
```

Promotrimo operaciju $rank(i, 15)$ na primjeru sa slike (Slika 1). Na početku se nalazimo u korijenskom čvoru v, te je $r = 15$. Kako je u korijenskom čvoru znak i zakodiran sa jedinicom, izvodimo $rank_{1v}(15)$ što je jednako 9 budući da se u podnizu 0011101000111011 (bitovi na pozicijama 0-15 u kodiranom zapisu korijenskog čvora) nalazi 9 jedinica. Pomičemo se u desno dijete čvora v.

Sad se nalazimo u čvoru $jroinisimjuoov$, za koji je $\Sigma_1 = \{i, j, m, n, o\}$ te $\Sigma_2 = \{r, s, u, v\}$. Smanjujemo r na 8. Kako je u tom čvoru znak i zakodiran nulom, računamo $rank_{0v}(8) = 7$ jer podniz 010000100 (bitovi na pozicijama 0-8 u kodiranom zapisu trenutnog čvora) sadrži 7 nula, te se pomičemo u lijevo dijete.

Sad se nalazimo u čvoru $joiniimjoo$, za koji je $\Sigma_1 = \{i, j, m\}$ te $\Sigma_2 = \{n, o\}$. Smanjujemo r na 6. Kako je u tom čvoru znak i zakodiran nulom, računamo $rank_{0v}(6) = 5$ jer podniz 0101000 (bitovi na pozicijama 0-6 u kodiranom zapisu trenutnog čvora) sadrži 5 nula, te se pomičemo u lijevo dijete.

Sad se nalazimo u čvoru $jiiimj$, za koji je $\Sigma_1 = \{i, j\}$ te $\Sigma_2 = \{m\}$. Smanjujemo r na 4. Kako je u tom čvoru znak i zakodiran nulom, računamo $rank_{0v}(4) = 4$ jer podniz 00001 (bitovi na pozicijama 0-4 u kodiranom zapisu trenutnog čvora) sadrži 4 nule, te se pomičemo u lijevo dijete.

Sad se nalazimo u čvoru $jiiij$, za koji je $\Sigma_1 = \{i\}$ te $\Sigma_2 = \{j\}$. Smanjujemo r na 3. Kako je u tom čvoru znak i zakodiran nulom, računamo $rank_{0v}(3) = 3$ jer podniz 1000 (bitovi na pozicijama 0-3 u kodiranom zapisu trenutnog čvora) sadrži 3 nule, te se pomičemo u lijevo dijete. Kako trenutni čvor nema lijevo dijete izlazimo iz petlje algoritma te je rezultat $rank(i, 15)$ operacije jednak 3.

Razmotrimo još vremensku složenost $rank$ operacije nad stablom valića. Ako se za ulazni niz duljine n izgradi stablo valića uz primjenu RRR struktura za pohranu binarnih vektora, složenost $rank$ operacije postaje $O(\log(|\Sigma|))$ budući da se $rank$ upiti nad RRR-om mogu izvesti u konstantnom vremenu. Σ označava abecedu ulaznog niza znakova. Ako se ne bi koristila RRR struktura za pohranu binarnih vektora, složenost bi postala $O(nlog|\Sigma|)$ budući da se u najgorem slučaju treba slijedno proći čitav binarni vektor kako bi se odredio broj pojavljivanja nule ili jedinice. Bez primjene stabla valića složenost $rank$ operacije zbog potrebe slijednog čitanja niza znakova postaje $O(n)$.

2.2.2 select

Pseudokod $select$ operacije prikazan je u nastavku (Tablica 3). Rezultat $select$ operacije je indeks koji može biti nula. U navedenom pseudokodu javljaju se $select$ operacije nad binarnim vektorom koje će biti detaljnije objašnjene u narednom

poglavlju. $select_{0v}(i)$ računa indeks i-te nule u binarnom vektoru v. $select_{1v}(i)$ radi na jednak način uz računanje indeksa i-te jedinice umjesto nule.

Tablica 3 Select operacija nad stablom valića

Neka $\mathbf{select_{0v}}$ *odnosno* $\mathbf{select_{1v}}$ *predstavljaju select operacije nad binarnom (kodiranom) reprezentacijom čvora* v. *(Objašnjeno u narednom poglavlju)*

```
def select(c, i):
    v ← čvor stabla koji predstavlja znak c
    r ← i
    Ako je c zakodiran nulom u čvoru v:
        r ← select_0v(r)
    Inače:
        r ← select_1v(r)
    Dok je v != korijenski čvor
        r ← r + 1
        p ← roditelj(v)
        Ako je v lijevo dijete od p:
            r ← select_0p(r)
        Inače:
            r ← select_1p(r)
        v ← p
    vrati r
```

Promotrimo operaciju $select(D, 1)$ na primjeru sa slike (Slika 1). Prvo je potrebno pronaći u stablu list koji predstavlja znak D. Jednostavnim spuštanjem niz stablo uz provjeru particije abecede čvorova, dolazi se do čvora v čiji je sadržaj $S = D1$. Budući da je u tom čvoru znak D zakodiran sa jedinicom, izvodi se upit $select_{1v}(1) = 0$ jer pozicija prve jedinice u binarnom prikazu čvora v (10) iznosi 0.

Vrijednost r se ažurira na 1. Roditelj čvora v je čvor p sa sadržajem $S = \ D\ 100$!. Kako je v desno dijete od p, izvodi se upit $select_{1p}(1) = 2$ jer je indeks prve jedinice u

binarnom prikazu 0010010000 čvora p jednak 2. Penjemo se uz stablo do roditelja. Čvor p postaje novi čvor v.

Vrijednost r se ažurira na 3. Roditelj čvora v je čvor p sa sadržajem $S = Ha\ De\ a\ 100\ bda!$. Kako je v lijevo dijete od p, izvodi se upit $select_{0p}(3) = 4$ jer je indeks treće nule u binarnom prikazu 11000101000001110 čvora p jednak 4. Penjemo se uz stablo do roditelja. Čvor p postaje novi čvor v.

Vrijednost r se ažurira na 5. Roditelj čvora v je čvor p sa sadržajem $S = Hajro\ i\ Denis\ imaju\ 100\ bodova!$. Kako je v lijevo dijete od p, izvodi se upit $select_{0p}(5) = 8$ jer je indeks pete nule u binarnom prikazu čvora p jednak 8. Penjemo se uz stablo do roditelja. Čvor p postaje novi čvor v. Kako je čvor v postao korijenski čvor, upit završava te se znak D u ulaznom nizu prvi put pojavljuje na poziciji 8.

Vremenska složenost $select$ upita nad stablom valića iznosi $O(\log|\Sigma|)$ ukoliko se $select$ upiti nad binarnim vektorima izvršavaju u konstantnom vremenu. Ako se ne bi koristilo stablo valića za ulazni niz dužine n znakova, složenost $select$ upita iznosila bi $O(n)$ zbog potrebe za slijednim prolaskom kroz niz znakova. Σ označava abecedu ulaznog niza.

Usporedbom $select$ i $rank$ operacija, može se uvidjeti da vrijedi sljedeće:

$$rank\big(c, select(c, i)\big) = i$$

Obrat ne vrijedi nužno.

2.2.3 access

Pseudokod $access$ operacije prikazan je u nastavku (Tablica 4). U navedenom pseudokodu javlja se $access$ operacija nad binarnim vektorom, koja čita bit na i-toj poziciji.

Tablica 4 Access operacija nad stablom valića

Neka $\mathbf{access}_v$ *predstavlja access operaciju nad binarnom (kodiranom) reprezentacijom čvora* v. $rank_{0v}$ *i* $rank_{1v}$ *su definirani kao u tablici 4. Neka* Σ_v *predstavlja polje znakova koji se kodiraju u čvoru* v. *(njegov dio abecede)*

```
def access(i):
    v ← korijenski čvor
    r ← i
    Dok je v != NULL
        Ako v nije korijenski čvor: r ← r − 1
        Ako je access_v(r) = 0:
            r ← rank_0v(r)
            Ako je v_lijevo = NULL: vrati prvi znak iz Σ_v
            v ← v_lijevo
        Inače:
            r ← rank_1v(r)
            Ako je v_desno = NULL: vrati zadnji znak iz Σ_v
            v ← v_desno
    vrati 0
```

Promotrimo operaciju $access(9)$ na primjeru sa slike (Slika 1). Na početku algoritma nalazimo se u korijenskom čvoru. Kako je rezultat upita $access_v(9) = 0$, ažuriramo r na vrijednost $rank_{0v}(9) = 6$ te se spuštamo u lijevo dijete.

Ažurira se r na vrijednost 5. Kako je binarna vrijednost trenutnog čvora 11000101000001110, $access_v(5) = 1$, te ažuriramo r na vrijednost $rank_{1v}(5) = 3$ i spuštamo se u desno dijete.

Ponovno se ažurira r na vrijednost 2. Kako je binarna vrijednost trenutnog čvora 0010010, $access_v(2) = 1$, te ažuriramo r na vrijednost $rank_{1v}(2) = 1$ i spuštamo se u desno dijete.

Ponovno se ažurira r na vrijednost 0. Kako je binarna vrijednost trenutnog čvora 10, $access_v(0) = 1$, te ažuriramo r na vrijednost $rank_{1v}(0) = 1$. Kako je trenutni čvor v list, ne možemo se dalje spustiti niz stablo te kao rezultat vraćamo zadnji znak iz abecede trenutno čvora v. Kako je abeceda jednaka $\Sigma_v = \{d, e\}$ rezultat operacije $access(9)$ iznosi e.

Vremenska složenost $access$ upita nad stablom valića iznosi $O(\log|\Sigma|)$ ukoliko se za pohranu binarnih vektora koristi RRR, budući da se $access$ upiti nad RRR-om izvršavaju u konstantnom vremenu. Σ označava abecedu ulaznog niza.

3. RRR struktura podataka

RRR[4][8] je sažimajuća struktura podataka za pohranu binarnih nizova. Koristi pomoćne memorijske strukture kako bi omogućila izvedbu $rank$ upita u konstantnom vremenu. Ulazni se binarni niz implicitno komprimira, no ne postoji potreba za njegovom eksplicitnom dekompresijom prilikom obavljanja upita. Postoje definicije RRR-a koje koriste dodatne memorijske strukture kako bi se i $select$ upiti izvodili u konstantnom vremenu, međutim takve inačice nisu razmatrane u okviru ovog projekta.

3.1 Izgradnja RRR-a

Izgradnja RRR-a započinje sa podjelom izvornog binarnog vektora na superblokove zadane veličine. Nakon toga svaki superblok dijeli se na blokove veličine b bita. Broj blokova po superbloku označimo sa f. Za ulazni binarni niz duljine n veličina bloka određuje se prema [2][10]:

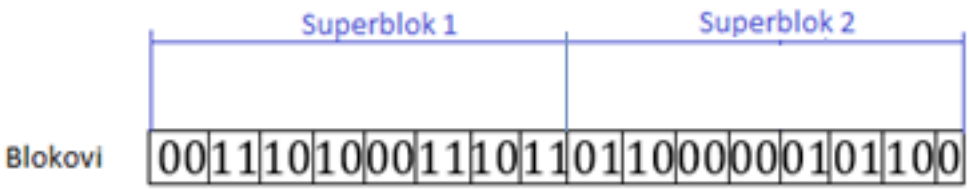
$$b = \left\lfloor \frac{\log_2 n}{2} \right\rfloor$$

Veličina superbloka može se odrediti prema sljedećem izrazu [10]:

$$s = \lfloor \log_2 n \rfloor^2$$

$$f = \frac{s}{b}$$

Razmotrimo podjelu izvornog binarnog niza na superblokove i blokove za slučaj korijenskog čvora sa slike (Slika 1). Podjela je prikazana na sljedećoj slici (Slika 2) pri čemu je $n = 31$, $b = 2$, $s = 16$, $f = 8$.

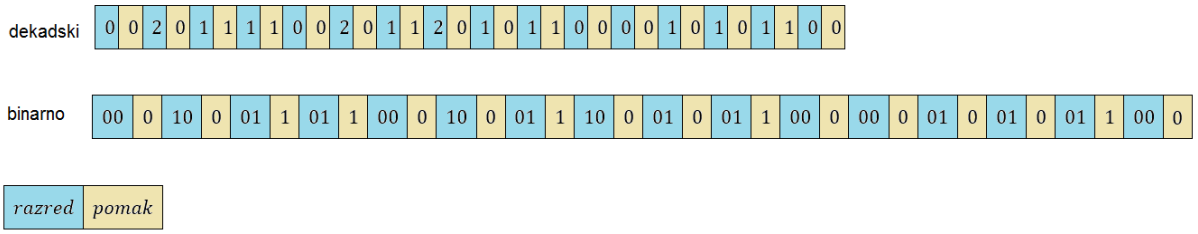


Slika 2 RRR - podjela na superblokove i blokove

Svaki blok sa prethodne slike (Slika 2) bit će kodiran sa dvije vrijednosti:

- Razred c – broj jedinica u bloku
- Pomak o – pomak u pretincu lookup tablice

Razred c koristi se za odabir pretinca lookup tablice, dok pomak određuje koja od permutacija nula i jedinica pohranjenih u tom pretincu odgovara izvornom bloku (detaljnije objašnjeno kasnije). Kodirani blokovi pohranjuju se u binarnom obliku u RRR, pri čemu je za pohranu razreda c potrebno $\lceil \log_2(b+1) \rceil$ bita budući da broj postavljenih bita u bloku duljine b može biti cijeli broj iz skupa $\{0,1,\dots,b\}$. Broj bita potreban za pohranu pomaka o ovisi o razredu c budući da broj permutacija duljine b bita sa c jedinica ovisi o broju c. Iz prethodnog slijedi da je za pohranu pomaka o potrebno $\left\lceil \log_2\binom{b}{c} \right\rceil$ bita. Za primjer sa prethodne slike (Slika 2) kodirani blokovi prikazani su na sljedećoj slici (Slika 3).



Slika 3 RRR – blokovi

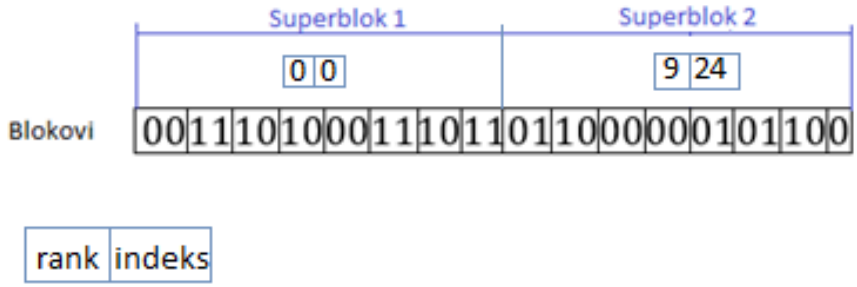
Izvorni sadržaj prvog bloka sa slike (Slika 2) je 00, te će on biti kodiran kao 000 pri čemu je njegov razred kodiran sa 00 jer izvorni blok ne sadrži niti jednu jedinicu, dok će pomak iznositi 0 jer je permutacija 00 zapisana u pretincu 0 (dekadska vrijednost od 00) na poziciji 0 (Slika 5).

Izvorni sadržaj drugog bloka je 11, te će on biti kodiran kao 100 pri čemu je njegov razred kodiran sa 10 jer izvorni blok sadrži dvije jedinice, dok će pomak iznositi 0 jer je permutacija 11 zapisana u pretincu 2 (dekadska vrijednost od 10) na poziciji 0 (Slika 5).

Blokovi čine drugu razinu prilikom izgradnje RRR-a. Prvu razinu čine superblokovi. Za svaki superblok pamti se:

- Kumulativni $rank$ (broj postavljenih nula) svih superblokova do trenutnog superbloka
- Indeks prvog bita razreda prvog bloka u kodiranom zapisu sa slike (Slika 3)

Slika 4 prikazuje prethodno navedene komponente superblokova za primjer sa slike (Slika 2).

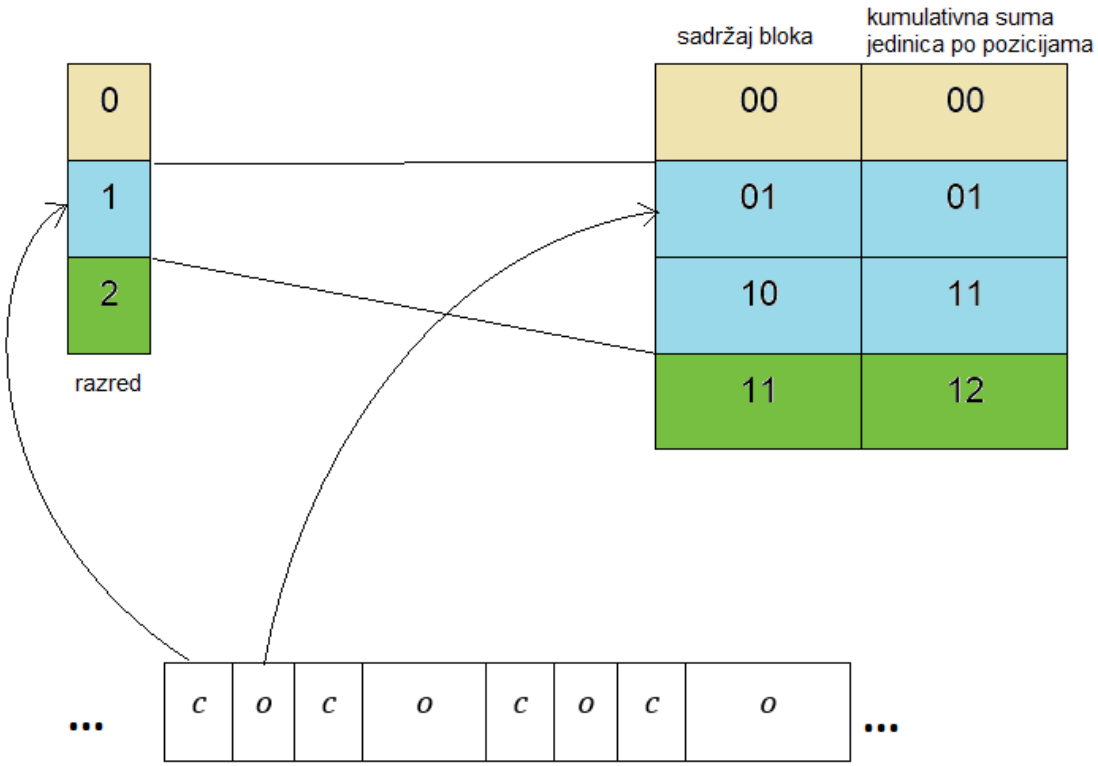


Slika 4 RRR – superblokovi

Kumulativni $rank$ drugog superbloka iznosi 9 budući da je suma jedinica zapisanih u svim prethodnim blokovima (dakle suma razreda blokova) jednaka 9. Druga komponenta drugog superbloka iznosi 24 budući da je to indeks prvog bita prvog bloka iz tog superbloka u prikazu sa slike (Slika 3).

Osnovni razlog zbog kojeg se blokovi grupiraju u superblokove jest skraćivanje vremena potrebno za odgovaranje na $rank$ upit. Umjesto prolaska po svim blokovima i akumuliranja njihovih razreda c, može se akumulirati kumulativni $rank$ zapisan u superblokovima.

Važna komponenta RRR strukture, koja omogućava brzo izvođenje $rank$ upita je lookup tablica. Kao što je već prethodno rečeno, tablica je podijeljena u pretince čiji indeks odgovara razredu bloka. Svaki pretinac sadrži zapise koji predstavljaju stvarni sadržaj bloka duljine b sa c jedinica. Indeks zapisa u pretincu određen je pomakom o bloka. Za svaki zapis pohranjuje se kumulativna suma jedinica za svaku poziciju u stvarnom sadržaju bloka. Svaki pretinac posjeduje onoliko zapisa koliko postoji permutacija niza nula i jedinica duljine b sa c jedinica. Tablica za primjer sa slike (Slika 2) prikazana je na sljedećoj slici (Slika 5).



Slika 5 RRR – lookup tablica

Lookup tablica može se definirati tako da istu tablicu dijeli više RRR-ova. U tom slučaju tablica se izgradi za RRR sa najvećim blokovima, te se prilikom kodiranja pomaka blokova ostalih (manjih) RRR-ova pomak određuje razmatrajući prvih b bitova zapisanih u elementima pretinaca tablice. Pritom b označava duljinu bloka kod manjeg RRR-a. Pogledajmo to na primjeru. Neka je tablica izgrađena za blokove duljine 2 (Slika 5). Istu tablicu želimo iskoristiti za blokove duljine 1. Neka je sadržaj bloka 1. Dakle njegov razred je 1 (jedna jedinica u bloku). Da bismo odredili pomak o, odlazimo u drugi pretinac tablice (indeks 1), te promatramo samo prvi bit kod svakog zapisa. Budući da se prvi bit drugog zapisa 10 podudara sa sadržajem bloka (1), kao pomak odabiremo 1.

Na temelju izloženog vidimo da se izgradnja RRR strukture izvodi sljedećim redoslijedom:

- Izračunaju se veličine blokova i superblokova
- Izgradi se lookup tablica

- Ulazni binarni niz čita se blok po blok te kodira (svaki blok predstavlja se parom razred-pomak)
- Kada se popuni superblok ažuriraju se njegovi podaci ($rank$, indeks) te se dodaje novi blok.

3.2 Operacije nad RRR-om

RRR struktura omogućava izvođenje tri osnovne operacije (analogno stablu valića):

- $rank_b(i)$ – broj pojavljivanja bita b do uključivo i-te pozicije u ulaznom binarnom nizu
- $select_b(i)$ – indeks i-tog bita b u ulaznom binarnom nizu
- $access(i)$ – bit na i-toj poziciji u ulaznom binarnom nizu

3.2.1 $rank$

Pseudokod $rank$ operacije prikazan je u nastavku (Tablica 5). Indeksi i započinju s nulom. U navedenom pseudokodu c predstavlja oznaku bita, dakle 0 ili 1. Binarni niz nad kojim se izvodi operacija označavamo sa v pa su zapisi $rank_c$ i $rank_{cv}$ ekvivalentni. U pseudokodu se javljaju sljedeće operacije:

- $rank(x)$ – vraća kumulativni rank superbloka x
- $prviBlok(x)$ – vraća prvi blok superbloka x
- $razred(x)$ i $pomak(x)$ – vraćaju razred c odnosno pomak o bloka x

Tablica 5 Rank operacija nad RRR strukturom

```
def rank_1(i):
    i_b ← ⌊i/b⌋
    i_s ← ⌊i_b/f⌋
    suma ← rank(superblok_{i_s})
    trenutniBlok ← prviBlok(superblok_{i_s})
    Dok trenutniBlok != blok_{i_b}:
        suma ← suma + razred(trenutniBlok)
        trenutniBlok ← sljedeći blok
```

```
    j ← i mod b
    suma ← suma + lookup(razred(blok_{i_b}), pomak(blok_{i_b}), j)
    vrati suma

def rank_0(i):
    vrati i + 1 − rank_1(i)
```

Promotrimo izračun $rank_1(13)$ na primjeru (Slika 4 i Slika 3). Prvo se odredi indeks bloka u kojem je pohranjen traženi bit kao $i_b = \left\lfloor \frac{13}{2} \right\rfloor = 6$. Zatim određujemo indeks superbloka u kojem se nalazi traženi bit kao $i_s = \left\lfloor \frac{6}{8} \right\rfloor = 0$. Dakle nalazimo se u prvom superbloku (sa indeksom 0). Postavlja se $suma$ na vrijednost $rank(superblok_0) = 0$ (Slika 4 – prva komponenta superbloka). Na temelju indeksa zapisanog u prvom superbloku (Slika 4 – druga komponenta superbloka), određuje se njegov prvi blok. Suma se zatim ažurira tako da joj se pridodaju vrijednosti razreda svih blokova do bloka sa indeksom 6. Dakle nova vrijednost sume iznosit će $suma = 0 + 2 + 1 + 1 + 0 + 2 = 6$. Sljedeći korak je čitanje kumulativne sume jedinica na poziciji $j = 13\ mod\ 2 = 1$ u lookup tablici za blok sa razredom $c = 1$ i pomakom $o = 1$. Iz lookup tablice (Slika 5) pročita se kumulativna suma iznosa 1 (drugi bit drugog stupca u drugom zapisu plavog pretinca tablice). Vrijednost sume se ažurira te je rezultat $rank_1(13)$ operacije jednak $suma + 1 = 6 + 1 = 7$, što odgovara broju jedinica u ulaznom nizu do uključivo 13-te pozicije (Slika 2). Rezultat operacije $rank_0(13)$ prema pseudokodu (Tablica 5) iznosi 7 što odgovara broju nula do uključivo 13-te pozicije.

Vremenska složenost $rank$ operacije nad RRR strukturom je $O(1)$ budući da nema potrebe za iteriranjem kroz sve blokove kako bi se odredila kumulativna suma jedinica. Najviše je potrebno slijedno proći kroz f blokova, pri čemu za konstantan f složenost postaje konstantna.

3.2.2 select

Pseudokod $select$ operacije prikazan je u nastavku (Tablica 6). Vrijednosti $broj$ započinju s jedinicom. U navedenom pseudokodu t predstavlja oznaku bita, dakle 0 ili 1. U pseudokodu se javljaju sljedeće operacije:

- $rank(x)$ – vraća kumulativni rank superbloka x
- $prviBlok(x)$ – vraća prvi blok superbloka x
- $razred(x)$ i $pomak(x)$ – vraćaju razred c odnosno pomak o bloka x
- $indeksZaRank_t(blok,\ i)$ – vraća indeks na kojem se pojavljuje i-ta jedinica ili nula (t) u zadanom bloku

Tablica 6 Select operacija nad RRR strukturom

```
def select_1(broj):
    i_s ← max(i) | rank_1(superblok_i) < broj
    suma ← rank(superblok_{i_s})
    tB ← prviBlok(superblok_{i_s})
    indeks_1 ← i_s * b * f
    Dok ima blokova:
        s ← suma + razred(tB)
        Ako s ≥ broj:
            Izađi iz petlje
        Inače:
            suma ← suma + razred(tB)
        tB ← sljedeći blok
        indeks_1 ← indeks_1 + b
    indeks_1 ← indeks_1 + indeksZaRank_1(tb, broj − suma)
    vrati indeks_1

def select_0(broj):
    i_s ← max(i) | rank_0(superblok_i) < broj
    tB ← prviBlok(superblok_{i_s})
    indeks_0 ← i_s * b * f
    suma ← indeks_0 − rank(superblok_{i_s})
    Dok ima blokova:
        s ← suma + (b − razred(tB))
        Ako s ≥ broj:
            Izađi iz petlje
```

```
        Inače:
            suma ← suma + (b − razred(tB))
        tB ← sljedeći blok
        indeks_0 ← indeks_0 + b
    indeks_0 ← indeks_0 + indeksZaRank_0(tb, broj − suma)
    vrati indeks_0
```

Promotrimo izračun $select_1(7)$ na primjeru (Slika 4 i Slika 3). Prvo se odredi indeks superbloka u kojem se nalazi traženi bit kao $i_s = \max(i) \mid rank_1(superblok_i) \leq 7 = 0$. Dakle nalazimo se u prvom superbloku (sa indeksom 0). Postavlja se $suma$ na vrijednost $rank(superblok_0) = 0$ (Slika 4 – prva komponenta superbloka). Na temelju indeksa zapisanog u prvom superbloku (Slika 4 – druga komponenta superbloka), određuje se njegov prvi blok i to je trenutni blok. Vrijednost $indeks_1$ postavlja se na 0, jer se nalazimo u prvom superbloku. Suma se zatim ažurira tako da joj se pridodaju vrijednosti razreda svih blokova do bloka sa indeksom 6. U svakom koraku se uvećava i $indeks_1$ za iznos broja bitova po bloku, a to je 2. Dakle nova vrijednost sume iznosit će $suma = 0 + 2 + 1 + 1 + 0 + 2 = 6$. Nova vrijednost $indeks_1$ iznosi $indeks_1 = 2 * 6 = 12$. Došli smo do traženog bloka. Sljedeći korak je čitanje na kojoj poziciji u bloku se nalazi $7 - 6 = broj - suma = 1$ jedinica. Iz lookup tablice (Slika 5) pronađe se pozicija na kojoj se pojavljuje prva jedinica. Prva jedinica se nalazi na prvoj poziciji, i na izračunati $indeks_1$ dodaje se 0. Konačna vrijednost indeksa je $indeks_1 = 12 + 0 = 12$, što odgovara indeksu 7. jedinice u ulaznom nizu (Slika 2). $select_0$ računa se na isti način, samo što se umjesto jedinica, broje nule. Tako će vrijednost računanja $select_0(7)$ operacije iznositi 13. Suma će u zadnjem koraku iznositi $suma = 2 + 0 + 1 + 1 + 2 + 0 = 6$, a $indeks_0$ 12. Iz lookup tablice (Slika 5) pronađe se pozicija na kojoj se pojavljuje prva nula ($7 - 6 = broj - suma = 1$). Prva nula se nalazi na drugoj poziciji, i na izračunati $indeks_0$ dodaje se 1. Konačna vrijednost indeksa je $indeks_0 = 12 + 1 = 13$, što odgovara indeksu 7. nule u ulaznom nizu (Slika 2).

Vremenska složenost $select$ operacije nad RRR strukturom koja ima N superblokova i b bitova po bloku je $O(\log(N) + f + \log(b))$ budući da je potrebno pronaći superblok u kojem se nalazi traženi indeks, a u pronađenom bloku poziciju preostalog broja

pojavljivanja traženog elementa, a oboje se može napraviti binarnom pretragom. Najviše je potrebno slijedno proći kroz f blokova, pri čemu za konstantan f složenost postaje $O(\log(N) + \log(b))$.

3.2.3 access

Pseudokod $access$ operacije prikazan je u nastavku (Tablica 7). Indeksi i započinju s nulom.

Tablica 7 Access operacija nad RRR strukturom

```
def access(i):
    Ako je i = 0:
        vrati rank_1(i)
    vrati rank_1(i) − rank_1(i − 1)
```

Promotrimo izračun $access(13)$ na primjeru (Slika 2). Kako je $rank_1(13) = 7$ (jer je broj jedinica u ulaznom nizu do uključivo 13-te pozicije jednak 7) te $rank_1(12) = 7$ (jer je broj jedinica u ulaznom nizu do uključivo 12-te pozicije jednak 7), rezultat operacije $access(13)$ iznosit će $7 - 7 = 0$ što odgovara bitu koji se u ulaznom nizu nalazi na 13-toj poziciji.

Kako je vremenska složenost $rank$ operacije konstanta slijedi da je složenost $access$ operacije također $O(1)$.

4. Eksperimentalni rezultati

Za potrebe eksperimentalne analize performansi implementiranog stabla valića, mjereno je vrijeme potrebno za njegovu izgradnju, prosječno vrijeme izvršavanja upita, te zauzeće memorije od strane samog stabla, te cjelokupnog programa. Razmatran je utjecaj veličine abecede na prethodna mjerenja, te su u tu svrhu umjetno generirane ispitne sekvence definiranih duljina. Ponašanje implementiranog stabla ispitano je i na FASTA datotekama skinutim sa javno dostupnih baza1 2.

Tablica 8 i tablica 9 prikazuju vrijeme potrebno za izgradnju stabla za različite veličine ulaznih datoteka, te različite veličine abecede.

Tablica 8 Vrijeme izgradnje stabla valića – sintetičke datoteke

Broj znakova u datoteci / Broj znakova abecede	4	16	28
100	(154 μs) < 1 ms	(266 μs) < 1 ms	2 ms
1.000	2 ms	1 ms	2 ms
10.000	14 ms	28 ms	43 ms
100.000	378 ms	757 ms	878 ms
1.000.000	5.516 ms	11.519 ms	13.267 ms
10.000.000	155.459 ms	315.767 ms	370.959 ms
100.000.000	5.107.821 ms	10.759.557 ms	11.329.138 ms

[1] http://www.ncbi.nlm.nih.gov/
[2] http://bacteria.ensembl.org/index.html

Tablica 9 Vrijeme izgradnje stabla valića - FASTA datoteke

Ulazna datoteka	Broj znakova	Veličina abecede	Vrijeme izgradnje
HIV	999	5	1 ms
Coli	3.657	4	3 ms
Flu	3.990	4	3 ms
Camelpox	205.719	4	641 ms
Bact1	1.587.120	4	12.977 ms
Pig	1.637.716	5	9.107 ms
Bact2	3.018.312	4	25.757 ms
HumanDNA	33.543.332	5	612.881 ms
Human200	198.295.559	9	6.286.888 ms

Rezultati su u skladu s očekivanjima. Veći ulazni nizovi zahtijevaju više vremena za izgradnju. Izražena je i razlika izgradnje za datoteke koje imaju jednaku duljinu ulaznog niza, ali različite veličine abecede. Veće abecede iziskuju dublje stablo pa zbog toga trebaju i više vremena za izgradnju, jer svaki dodatni čvor zahtijeva dodatno vrijeme za izgradnju RRR strukture.

Tablica 10 i tablica 11 prikazuju prosječno vrijeme potrebno za izvršavanje upita za različite veličine ulaznih datoteka, te različite veličine abecede.

Tablica 10 Prosječno vrijeme izvršavanja upita – sintetičke datoteke

Broj znakova u datoteci / Broj znakova abecede	$rank\ (\mu s)$			$select\ (\mu s)$			$access\ (\mu s)$		
	4	16	28	4	16	28	4	16	28
100	< 1	< 1	1	1	3	5	< 1	2	5
1.000	2	< 1	1	6	3	3	4	2	3
10.000	< 1	2	1	2	7	4	2	14	5

100.000	< 1	1	1	2	4	5	2	4	5
1.000.000	< 1	1	2	2	5	7	2	5	7
10.000.000	1	3	3	3	7	8	3	7	8
100.000.000	1	3	4	5	9	11	4	8	10

Tablica 11 Prosječno vrijeme izvršavanja upita - FASTA datoteke

Ulazna datoteka	Broj znakova	Veličina abecede	$rank\ (\mu s)$	$select\ \ (\mu s)$	$access\ \ (\mu s)$
HIV	999	5	< 1	2	2
Coli	3.657	4	< 1	2	1
Flu	3.990	4	< 1	1	1
Camelpox	205.719	4	< 1	2	2
Bact1	1.587.120	4	1	3	3
Pig	1.637.716	5	1	4	3
Bact2	3.018.312	4	1	3	3
HumanDNA	33.543.332	5	2	5	5
Human200	198.295.559	9	3	12	8

Rezultati pokazuju da je vrijeme izvršavanja $rank$ upita logaritamsko. Veća abeceda povećava dubinu stabla i samim tim je ukupno vrijeme izvršavanja veće (logaritamski faktor). Kako se $access$ izvršava pomoću $rank$ upita, njegovo vrijeme je proporcionalno vremenu $rank$ upita. $Select$ upit nad RRR strukturom izvršava se u logaritamskom vremenu, te je ukupno vrijeme izvršavanja $select$ upita nad stablom valića uvećano dodatnim logaritamskim faktorom koji ovisi o dubini stabla. Time se može opravdati veće vrijeme izvršavanja $select$ upita za dublja stabla i veće ulazne nizove.

Tablica 12 i tablica 13 prikazuju izmjerenu potrošnju memorije na BioLinux platformi za cjelokupni proces i izgrađeno stablo valića.

Tablica 12 Potrošnja memorije nakon izgradnje stabla – sintetičke datoteke

Broj znakova u datoteci / Broj znakova abecede	$Cijeli\ proces\ (MB)$			$Stablo\ valića\ (MB)$		
	4	16	28	4	16	28
100	1.41	1.41	1.41	0.07	0.07	0.07
1.000	1.42	1.42	1.43	0.08	0.08	0.08
10.000	1.49	1.51	1.51	0.10	0.10	0.10
100.000	2.00	2.16	2.19	0.54	0.54	0.54
1.000.000	6.80	8.42	8.68	4.41	5.97	6.47
10.000.000	59.73	71.86	75.68	48.85	60.96	64.69
100.000.000	145.29	263.92	289.97	48.67	167.30	193.35

Tablica 13 Potrošnja memorije nakon izgradnje stabla - FASTA datoteke

Ulazna datoteka	Broj znakova	Veličina abecede	$Cijeli\ proces\ (MB)$	$Stablo\ valića\ (MB)$
HIV	999	5	1.42	0.08
Coli	3.657	4	1.44	0.09
Flu	3.990	4	1.44	0.09
Camelpox	205.719	4	2.55	1.05
Bact1	1.587.120	4	9.91	6.98
Pig	1.637.716	5	11.76	8.79
Bact2	3.018.312	4	19.91	15.73
HumanDNA	33.543.332	5	67.58	34.34
Human200	198.295.559	9	290.44	100.07

Očekivano, s porastom veličine ulazne datoteke, povećava se i potrošnja memorije. Kod manjih ulaznih datoteka nije toliko izražena implicitna kompresija od strane RRR strukture budući da su definirane veličine blokova premalene, te prilikom njihovog kodiranja putem razreda i pomaka dolazi do povećanog zauzeća memorije u odnosu na veličinu ulaznog niza. Mjerenja zapisana u ovoj tablici dobivena su izravno od operacijskog sustava. Potrebno je napomenuti da je stvarna memorija, zauzeta samo za fizičke strukture čvorova stabla, dosta manja.

Za potrebe ispitivanja implementiranog rješenja generirani su unit testovi, te su dodatno slučajnim izborom generirani upiti za sve korištene ulazne datoteke. Na taj način generirano je i uspješno obavljeno 2.426.523 upita ($rank$, $select$ i $access$) na ukupno 31 ulaznoj datoteci.

5. Implementacija

Projekt je implementiran u programskom jeziku C++. Čitav kod je detaljno komentiran te će u nastavku samo ukratko biti opisani značajnije komponente rješenja. Rješenje čine četiri osnovna razreda:

- $WaveletTree$ – enkapsulira operacije koje se mogu izvoditi nad stablom valića te sadrži pokazivač na korijenski čvor stabla. Prilikom konstrukcije razreda gradi se stablasta struktura instanci $WaveletNode$-a.
- $WaveletNode$ – predstavlja čvor u stablu valića. Sadrži sve potrebne informacije za izvođenje upita nad stablom. Kodirani binarni sadržaj pohranjuje u obliku RRR-a. Prilikom konstrukcije ulazni niz se kodira i dijeli prema abecedi na dva dijela, koji se prosljeđuju lijevom i desnom djetetu trenutnog čvora.
- RRR – enkapsulira operacije koje se mogu izvoditi nad binarnim vektorom. Prilikom konstrukcije ulazni niz nula i jedinica se kodira, te zapisuje binarno u 64-bitne cijele brojeve bez predznaka. Budući da se nizovi obrađuju bit po bit, izbjegavani su pretjerani funkcijski pozivi, kako bi se smanjilo vrijeme izvođenja. Informacija o superblokovima pohranjuje se u zasebnom vektoru cijelih brojeva bez predznaka.
- $RRRTable$ – predstavlja *lookup* tablicu koju RRR koristi kako bi ostvario izvedbu $rank$ operacije u konstantnom vremenu. Tablica je realizirana kao višedimenzionalni vektor, te se prilikom izgradnje popunjava svim mogućim permutacijama nula i jedinica s odgovarajućim brojem postavljenih bita. Razred je implementiran kao *singleton*, te ga svi RRR-ovi međusobno dijele. Iz tog razreda prilikom prve konstrukcije, tablicu je potrebno konstruirati za najveći RRR.

Pored prethodno navedenih razreda, koriste se još dvije datoteke:

- $Common.h$ - sadrži često korištene definicije tipova.
- $main.h$ - sadrži kontrolnu logiku za učitavanje FASTA datoteke, mjerenje vremenske statistike, memorijske potrošnje te stvaranje grafičke reprezentacije.

6. Zaključak

Na temelju dobivenih rezultata možemo zaključiti kako stablo valića predstavlja vrlo efikasnu strukturu za izvedbu $rank$, $select$ i $access$ upita. Sam proces izgradnje stabla traje značajno vrijeme za veće ulazne nizove, međutim nakon toga svi prethodno navedeni upiti mogu se izvesti u konstantnom vremenu ($select$ uz dodatne memorijske strukture, koje ovdje nisu implementirane). Svojstvo sažimanja RRR strukture dolazi do izražaja tek za veće ulazne nizove budući da za malene nizove razred i pomak, kojim kodiramo pojedine blokove zahtijevaju više bitova od izvornog bloka. Prednost naše implementacije uključuje lookup tablicu koju dijele svi čvorovi u stablu valića, te nema potrebe za ponovnom izgradnjom tablice za svaki RRR.

Potencijalna poboljšanja uključivala bi implementaciju dodatnih memorijskih struktura koje bi omogućile izvedbu $select$ upita u konstantnom vremenu. Naravno, to bi uključivalo i veću potrošnju memorije. Dodatno bi se moglo razmotriti potencijalno ubrzanje izgradnje stabla. Brzina izgradnje stabla u konačnici nije toliko relevantna budući da se jednom izgrađeno stablo može serijalizirati i kao takvo brzo učitavati za naknadnu upotrebu.

7. Literatura

[1] Bowe, A. Multiary Wavelet Trees in Practice. Diplomski rad. School of Computer Science and Information Technology RMIT University Melbourne, 2010.

[2] González, R., Grabowski, S., Mäkinen, V., Navarro, G. Practical Implementation of Rank and Select Queries. Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms, Grčka, (2005), str. 27-38.

[3] Navarro, G. Wavelet trees for all. Journal of Discrete Algorithms. 25(2014), str. 2-20.

[4] Raman, R., Raman, V., Rao, S. S. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, (2002), str. 233-242.

[5] Clark, D. Compact Pat Trees. Doktorska disertacija. University of Waterloo, 1998.

[6] Kärkkäinen, J. Compressed bit vectors. http://www.cs.helsinki.fi/u/tpkarkka/opetus/12k/dct/lecture10.pdf, pristupljeno 10.1.2016.

[7] Bowe, A. Wavelet Trees – an Introduction. 28. 6. 2011. http://alexbowe.com/wavelet-trees/, pristupljeno 10.1.2016.

[8] Bowe, A. RRR – A Succinct Rank/Select Indeks for Bit Vectors. 1. 6. 2011. http://alexbowe.com/rrr/, pristupljeno 10.1.2016.

[9] Bowe, A. Generating Binary Permutations in Popcount Order. 9. 5. 2011. http://alexbowe.com/popcount-permutations/, pristupljeno 10.1.2016.

[10] Brejová, B. Succinct dana structures. http://compbio.fmph.uniba.sk/vyuka/vvt/poznamky/p16.pdf, pristupljeno 10.1.2016.

[11] Weiner, P. Linear pattern matching algorithm. Proc. 14th IEEE Symposium Switching Theory and Automata Theory, (1973), str. 1-11.

[12] Manber, U., Myers, G. Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing, 22(5), str. 935-948, 1993.

[13] Burrows, M., Wheeler, D.J. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[14] Ferragina, P., Manzini, G. Opportunistic dana structures with applications. Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science, (2000), str. 390-398.