SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

**BIOINFORMATIKA - PROJEKT**

# Izgradnja binarnog stabla valića kao RRR strukture

**Jure Čular 0036479001 Bartol Freškura 0036480392**
**Filip Gulan 0036479428**

Zagreb, siječanj 2018.

# SADRŽAJ

# 1. Uvod

U području bioinformatike mogu se susresti sekvence znakova velikih duljina koje je potrebno obraditi ili izvršavati određene upite nad istima, poput pronalaska broja pojavljivanja određenog znaka do danog indeksa i sličnih. Slijedno obrađivanje navedenih sekvenci bi bilo vrlo sporo i neučinkovito te se poseže za nešto bržim, no i kompliciranijim rješenjima. Jedan od predstavnika je i binarno stablo valića kao RRR struktura koja je obrađena u ovom projektu.

Zadatak ovog projekta je bila izgradnja gore navedene strukture, provedba testiranja nad sintetičkim i stvarnim podacima u vidu brzine izgradnje stabla, potrošnje memorije te prosječnog vremena izvršavanja upita *rank*, *select* i *access*. U nastavku dokumenta je prikazan opis RRR strukture, stabla valića te implementacije istih. Na kraju su dani rezultati i usporedba s onima od prošlogodišnjeg projekta [6].

# 2. RRR struktura podataka

RRR je struktura podatka za pohranjivanje bit-vektora pomoću koje je moguće u kratkom vremenu izvršiti upite. Navedena struktura je sažeta (engl. *Succint data structure*), no unatoč tomu nije ju potrebno u cijelosti raspakirati kako bi se obavio upit nad istom.

## 2.1. Izgradnja RRR strukture

Prilikom izgradnje RRR strukture prvi korak je podjela ulaznog niza bitova na blokove, odnosno superblokove. Veličine superbloka i bloka moguće je proizvoljno definirati. Nakon odabira veličine bloka potrebno je izgraditi RRR tablicu.

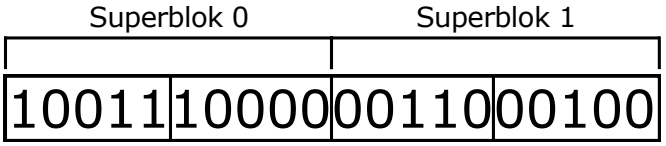
| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 00000 00000 | 3 | 0 | 00111 00123 |
| 1 | 0 | 00001 00001 | | 1 | 01011 01123 |
| | 1 | 00010 00011 | | 2 | 01101 01223 |
| | 2 | 00100 00111 | | 3 | 01110 01233 |
| | 3 | 01000 01111 | | 4 | 10011 11123 |
| | 4 | 10000 11111 | | 5 | 10101 11223 |
| 2 | 0 | 00011 00012 | | 6 | 10110 11233 |
| | 1 | 00101 00112 | | 7 | 11001 12223 |
| | 2 | 00110 00122 | | 8 | 11010 12233 |
| | 3 | 01001 01112 | | 9 | 11100 12333 |
| | 4 | 01010 01122 | 4 | 0 | 01111 01234 |
| | 5 | 01100 01222 | | 1 | 10111 11234 |
| | 6 | 10001 11112 | | 2 | 11011 12234 |
| | 7 | 10010 11122 | | 3 | 11101 12334 |
| | 8 | 10100 11222 | | 4 | 11110 12344 |
| | 9 | 11000 12222 | 5 | 0 | 11111 12345 |

**Slika 2.1:** Sadržaj RRR tablice za $b = 5$, preuzeto iz [6]

Tablica je pomoćna struktura koja će se koristiti u izvršavanju upita. Sama tablica će sadržavati sve moguće permutacije blokova za sve moguće rangove za danu veličinu bloka te niz sume rangova do svake pozicije unutar bloka. Implementacijski gledano tablica će sadržavati pokazivače na tablice za svaki rang bloka unutar kojih će biti

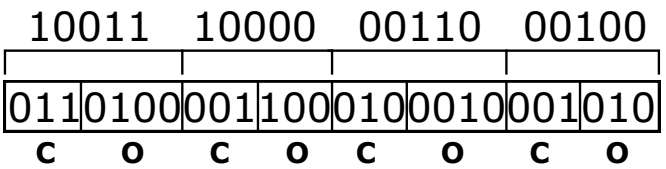
definirane permutacije tog ranka i odgovarajuća sumu jedinica za svaku poziciju u odgovarajućem bloku. Primjer jedne takve tablice prikazan je na slici 2.1.

Kroz sljedeći primjer bit će prikazana izgradnja RRR strukture iz ulaznog niza bitova, za veličinu bloka $b = 5$ i superbloka $f = 2$. Na slici 2.2 je prikazana podjelu niza na četiri bloka i dva superbloka. Nakon podjele niza bitova u blokove i superblokove potrebno je za svaki blok odrediti rang te odmak unutar tablice blokova, odnosno pozicija bloka unutar tablice koji odgovara danom bloku. Tako dobiveni parovi se spremaju u niz koji će se koristiti u daljnjim operacijama nad strukturom. Uz navedeni niz, također se sprema i niz parova koji sadrže ukupnu sumu rangova unutar prethodnih superblokova te odmak prvog bloka za pojedini superblok.




**Slika 2.2:** Podjela bit-vektora na blokove i superblokove

Korištenjem tablice 2.1 nad primjerom danim na slici 2.2 dobio bi se kodiran niz prikazan u obliku parova $[(3, 4), (1, 4), (2, 2), (1, 2)]$, gdje prvi element para predstavlja rang, a drugi odmak u tablici. Isti taj niz, zapisan kao niz bitova prikazan je na slici 2.3. Niz parova suma ranga i odmaka prvog sljedećeg bloka bi ovisio o tome kako se zapisuju navedeni parovi u memoriju. Naime, moguće je navedene parove zapisivati kao dva broja ili ih kodirati tako da se postigne maksimalna memorijska učinkovitost. No takav način kodiranja zahtjeva i dekodiranje prilikom obavljanja upita što može dodatno usporiti brzinu njihova izvršavanja. Razlika u ta dva načina zapisa je u odmaku prvog bloka, za nekodirane nizove odmak je samo indeks prvog slijedećeg bloka, dok za kodirane iznosi broj bitova do odmaka (odmak unutar RRR tablice) prvog slijedećeg bloka. Za primjer dan na slici 2.2 i bez kodiranja taj niz bi iznosio $[(0, 0), (4, 2)]$, dok bi uz kodiranje iznosio $[(0, 3), (4, 16)]$.




**Slika 2.3:** Ulazni niz `1001110000011000100` kodiran u RRR strukturu

### 2.1.1. Kodiranje parova

Kodiranje parova moguće je izvršiti tako da se uzme broj bitova potrebnih za najveći rang bloka unutar RRR tablice odnosno $r\_bit = \lceil(\log_2(b+1))\rceil$ gdje je $b$ veličina bitova u bloku, dok se broj bita odmaka uzme veličina koja ovisi o rangu $o\_bit = \lceil(\log_2(\binom{b}{r}))\rceil$ gdje je $r$ rang u bloku. Uz pomoć te dvije veličine moguće je jedan par spremiti unutar $o\_bit + r\_bit$ bitova.

### 2.1.2. Dekodiranje parova

Dekodiranje parova se vrši na način da se prvo dekodira rang jer je broj bitova za kodiranje ranga stalan te je poznat početak bloka. Nakon dekodiranja ranga, iz RRR tablice možemo saznati potreban broj bitova za odmak te dekodirati i sam odmak.

## 2.2. Upiti nad RRR strukturom

Nad RRR strukturom moguće je obavljati tri vrste upita:

- $rank_b(i)$- broj pojavljivanja bita $b$ do pozicije $i$ u ulaznom nizu bitova
- $select_b(i)$ - pozicija $i$-tog pojavljivanja bita $b$ u ulaznom nizu bitova
- $access(i)$ - bit na $i$-toj poziciji u ulaznom nizu bitova

### 2.2.1. *Rank*

Nad RRR strukturom možemo obavljati *rank1* i *rank0* upite, gdje *rank1* vraća broj bitova postavljenih na $1$ do danog indeksa u nizu, te *rank0* analogno tomu broj bitova postavljenih na $0$. Algoritam *rank1* i *rank0* upita prikazan je pseudokodom 1 i 2.

**Pseudokod 1** *rank1* upit nad RRR strukturom, preuzeto iz [2] i [6]

```
1: function RANK1(i)
2:     i_b ← ⌊i/b⌋
3:     i_s ← ⌊i_b/(b*s)⌋
4:     suma ← rank i_s-tog super bloka
5:     trenutni_blok ← prvi blok i_s tog super bloka
6:     while trenutni_blok ≠ i_b-ti blok do
7:         suma ← suma+ razred od trenutni_blok
8:         trenutni_blok ← slijedeći blok
9:         j ← i % b
10:        suma ← suma+ pozicija na kojoj se nalazi j-ta jedinica u bloku
11:        return suma
12:    end while
13: end function
```

**Pseudokod 2** *rank0* upit nad RRR strukturom, preuzeto iz [2] i [6]

```
1: function RANK0(i)
2:     return i + 1− RANK1(i)
3: end function
```

### 2.2.2. *Select*

Kao i za *rank* upite, nad RRR strukturom možemo izvoditi *select1* i *select0* upite. Oni odgovaraju pozicijom $i$-tog bita postavljenog na $1$, odnosno $0$. Algoritam *select1* i *select0* upita prikazan je pseudokodom 3 i 4.

**Pseudokod 3** *select1* upit nad RRR strukturom, preuzeto iz [2] i [6]

```
1:  function SELECT1(i)
2:      i_s ← indeks superbloka kojem je rank bitova 1 < i
3:      suma ← rank i_s-tog superbloka
4:      trenutni_blok prvi blok i_s-tog superbloka
5:      indeks ← i_s * s
6:      while ima blokova do
7:          s ← suma+ razred od trenutni_blok
8:          if s ≥ i then
9:              izađi iz petlje
10:         else
11:             suma = s
12:         end if
13:         trenutni_blok ← sljedeći blok
14:         indeks ← indeks + b
15:     end while
16:     indeks ← indeks+ indeks na kojem se nalazi (broj − suma)-ti bit 1 u bloku
17:     return indeks
18: end function
```

**Pseudokod 4** *select0* upit nad RRR strukturom, preuzeto iz [2] i [6]

```
1: function SELECT0(i)
2:     i_s ← indeks super bloka kojem je rank bitova 0 < i
3:     trenutni_blok prvi blok i_s-tog super bloka
4:     indeks ← i_s * s
5:     suma ← indeks− rank i_s-tog super bloka
6:     while ima blokova do
7:         s ← suma + (b− razred od trenutni_blok)
8:         if s ≥ i then
9:             izađi iz petlje
10:        else
11:            suma = s
12:        end if
13:        trenutni_blok ← sljedeći blok
14:        indeks ← indeks + b
15:    end while
16:    indeks ← indeks+ indeks na kojem se nalazi (broj − suma)-ti bit 0 u bloku
17:    return indeks
18: end function
```

### 2.2.3. *Access*

*Access* upit vraća vrijednost bita na $i$-toj poziciji. Implementacija se može izvesti pomoću dva poziva *rank1* upita. Algoritam *access* upita prikazan je pseudokodom 5.

**Pseudokod 5** Access operacija nad RRR strukturom [2][6]

```
1: function ACCESS(i)
2:     if i = 0 then
3:         return RANK1(i)
4:     else
5:         return RANK1(i) - RANK1(i − 1)
6:     end if
7: end function
```

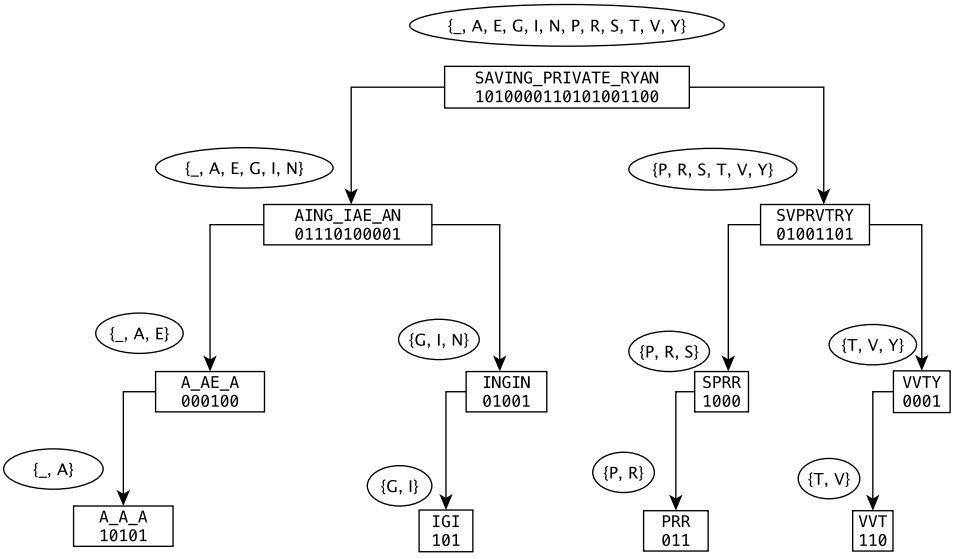
# 3. Stablo valića

Stablo valića je struktura koja iz ulaznog niza bitova gradi binarno stablo, gdje svaki čvor sadrži podniz početnog niza bitova [5] [2]. Navedenu strukturu su predložili Grossi, Gupta i Vitter [2] u vidu predstavljanja jako dugih sekvenci i obavljanja brzih rang upita nad istima.

## 3.1. Izgradnja stabla valića

Za izgradnju stabla valića potrebno je za svaki čvor definirati abecedu ulaznog niza znakova, potom je potrebno abecedu podijeliti na dvije polovice te sve znakove ulaznog niza koji se nalaze u prvom dijelu abecede zamijeniti s 0, a ostale s 1. Obje polovice abecede se dalje dijele rekurzivno u djeci čvorovima sve dok nije moguće više podijeliti abecedu, to jest ostanu samo dva znaka u abecedi trenutnog čvora.

Algoritam dan pseudokom 6 prikazuje postupak izgradnje stabla valića, te je na slici 3.1 prikazan primjer jednog tako izgrađenog stabla valića.




**Slika 3.1:** Binarno stablo valića za niz *Saving private Ryan*

**Pseudokod 6** Izgradnja čvora binarnog stabla valića, preuzeto iz [6]

```
1: function IZGRADI(Σ, S)
2:     Podijeli abecedu Σ na dva jednaka dijela Σ_1 i Σ_2
3:     Kodiraj sve znakove c ∈ Σ_1 u nizu S s 0, a ostale s 1
4:     if |Σ_1| ≥ S then
5:         S_1 ← znakovi kodirani 0 iz S
6:         IZGRADI(Σ_1, S1)
7:     end if
8:     if |Σ_2| ≥ S then
9:         S_2 ← znakovi kodirani 1 iz S
10:        IZGRADI(Σ_2, S2)
11:    end if
12: end function
```

## 3.2. Upiti nad stablom valića

Nad stablom valića definirani su sljedeći upiti:

- $rank_c(i)$ - broj pojavljivanja znaka $c$ do pozicije $i$ u ulaznom nizu znakova
- $select_c(i)$ - pozicija $i$-tog znaka $c$ u ulaznom nizu znakova
- $access_c(i)$ - znak na $i$-toj poziciji u ulaznom nizu znakova

### 3.2.1. *Rank*

*Rank* upit za dani znak $c$ i poziciju $i$ ispituje koliko znakova $c$ se pojavljuje do pozicije $i$, uključeno s $i$-tom pozicijom. Algoritam je prikazan pseudokodom 7.

**Pseudokod 7** *Rank* upit nad stablom valića, preuzeto iz [6]

```
1:  function RANK(c, i)
2:      v ← korijen stabla
3:      r ← i
4:      while v ≠ null do
5:          if v ≠ korijen then
6:              r ← r − 1
7:          end if
8:          if c ∈ Σ_1 then
9:              r ← RANK0(r)
10:             v ← v_lijevo
11:         else
12:             r ← RANK1(r)
13:             v ← v_desno
14:         end if
15:         if r = 0 then return 0
16:         end if
17:     end while
18:     return r
19: end function
```

### 3.2.2. *Select*

*Select* upit za dani znak $c$ i broja pojavljivanja znaka tog znaka $i$ ispituje na kojoj poziciji se nalazi $i$-ti znak $c$. Algoritam je prikazan pseudokodom 8.

**Pseudokod 8** *Select* upit nad stablom valića, preuzeto iz [6]

```
1:  function SELECT(c, i)
2:      v ← čvor koji predstavlja znak c
3:      r ← i
4:      if c ∈ Σ_1 then
5:          r ← SELECT0_v(r)
6:      else
7:          r ← SELECT1_v(r)
8:      end if
9:      while v ≠ korijen do
10:         r ← r + 1
11:         p ← RODITELJ(v)
12:         if v lijevo dijete od p then
13:             r ← SELECT0_p(r)
14:         else
15:             r ← SELECT1_p(r)
16:         end if
17:         v ← p
18:     end while
19:     return r
20: end function
```

### 3.2.3. *Access*

*Access* upit za danu poziciju $i$ vraća $i$-ti znak sekvence. Algoritam je prikazan pseudokodom 9.

**Pseudokod 9** *Access* upit nad stablom valića, preuzeto iz [6]

```
1: function ACCESS(i)
2:     v ← korijen stabla
3:     r ← i
4:     while v ≠ null do
5:         if v ≠ korijen then
6:             r ← r − 1
7:         end if
8:         if ACCESS(r) = 0 then
9:             r ← RANK0(r)
10:            if v_lijevo = null then return prvi znak iz Σ_v
11:            end if
12:            v ← v_lijevo
13:        else
14:            r ← RANK1(r)
15:            if v_desno = null then return zadnji znak iz Σ_v
16:            end if
17:            v ← v_desno
18:        end if
19:    end while
20:    return r
21: end function
```

# 4. Testiranje i rezultati

U ovom poglavlju predstavljeni su rezultati memorijskih i vremenskih mjerenja te su uspoređeni s prošlogodišnjom implementacijom [6]. Mjerenja su napravljena na umjetnim i stvarnim podacima. Prilikom provedbe mjerenja diverizfikacija je postignuta različitim veličinama abecede i ulaznih sekvenci.

Sva prikazana mjerenja su dobivena na *macOS High Sierra* operativnom sustavu te procesoru *Intel® Core™ i7-7660U*. Sve izvršne datoteke, one naše i od kolega prošlogodišnje implementacije, su prevedene u *release* konfiguraciji, što znači da je izvršni kod optimiran.

Umjetno stvoreni podaci su preuzeti od prošlogodišnjeg projekta [6] te su sastavljeni od različitih kombinacija veličine abecede i ulaznog niza. Tako su prisutne veličine abecede $4$, $15$ i $26$ te duljine ulaznih nizova: $100, 1.000, 10.000, 100.000$ i $1.000.000$. Stvarni podaci su *FASTA* datoteke preuzete s javno dostupnih baza genoma te je njihova statistika prikazana u tablici 4.1.

**Tablica 4.1:** Statistike za FASTA datoteke

| Datoteka | Duljina niza | Veličina abecede |
|---|---|---|
| HIV | 999 | 5 |
| E Coli | 5.534.367 | 4 |
| Flu | 3.990 | 4 |
| Camelpox | 205.719 | 4 |
| Bact1 | 1.587.120 | 4 |

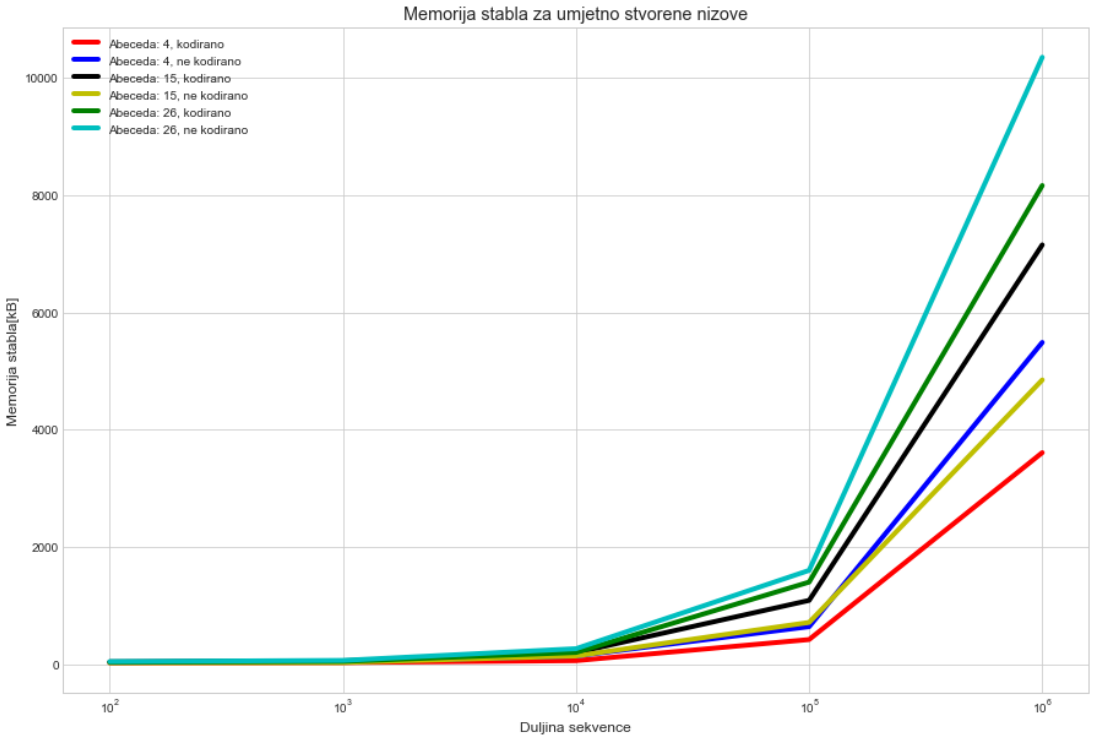
## 4.1. Memorijska mjerenja

Tablice 4.2 i 4.3 prikazuju zauzeće memorije stabla u dvije inačice, kodirana i nekodirana. Kodirana inačica sadrži RRR strukture koje su kodirane kako je prethodno opisano, dok nekodirana inačica ne sadrži kodirane RRR strukture, već zapisuje rang

i odmak kao standardne brojeve. Tablica 4.2 sadrži mjerenja za umjetno stvorene nizove, dok 4.3 sadrži mjerenja za stvarne nizove.

**Tablica 4.2:** Potrošnja memorije stabla valića za umjetno stvorene nizove

| Duljina sekvence | Veličina abecede | | | | | |
|---|---|---|---|---|---|---|
| | 4 | 15 | 26 | 4 | 15 | 26 |
| | Kodiran ($kB$) | | | Nekodiran ($kB$) | | |
| 100 | 24 | 32 | 40 | 24 | 32 | 44 |
| 1000 | 28 | 44 | 56 | 28 | 48 | 64 |
| 10000 | 60 | 140 | 196 | 140 | 200 | 264 |
| 100000 | 420 | 640 | 1.088 | 712 | 1.400 | 1.600 |
| 1000000 | 3.608 | 5.488 | 7.152 | 4.848 | 8.164 | 10.352 |




**Slika 4.1:** Memorija stabla valića za umjetno stvorene nizove

**Tablica 4.3:** Potrošnja memorije stabla valića za *FASTA* datoteke

| Datoteka | Kodiran ($kB$) | Nekodiran ($kB$) |
|---|---|---|
| HIV | 32 | 32 |
| E Coli | 16.240 | 24.728 |
| Flu | 36 | 52 |
| Camelpox | 900 | 1.008 |
| Bact1 | 4.776 | 7.312 |

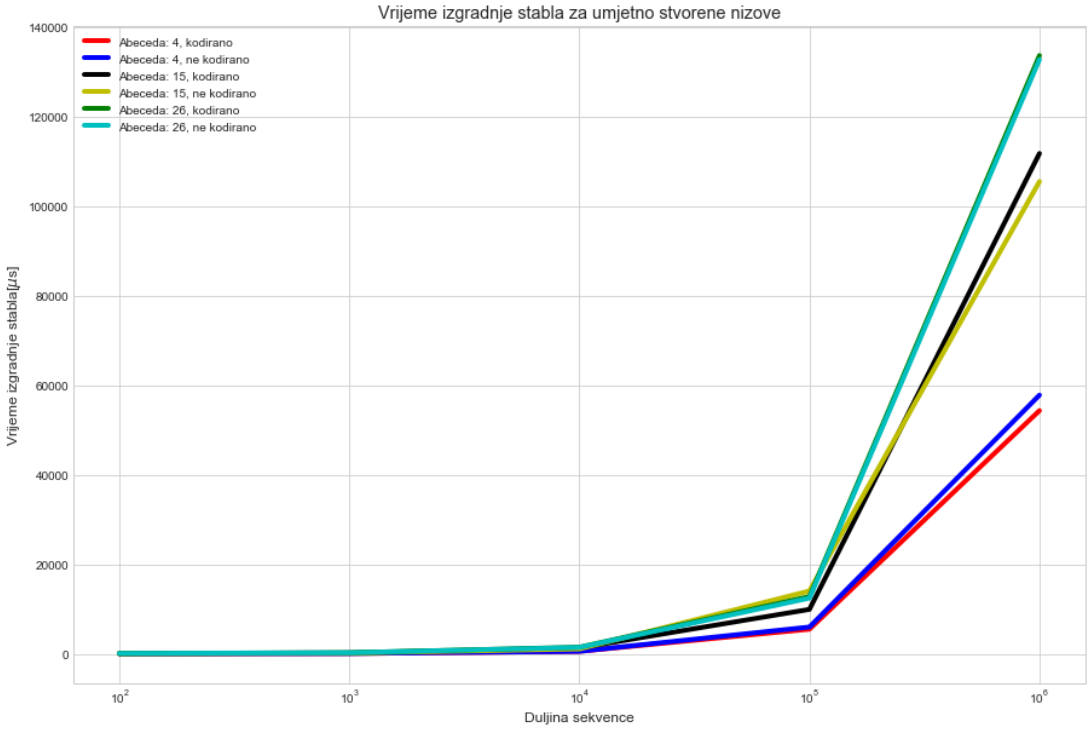
Iz tablica 4.2 i 4.3 vidljivo je da za stabla koja koriste kodiranje zauzimaju manje memorije kod većih sekvenci, od $30\%$ do $60\%$ manje memorije. Iako je zauzeće memorije manje, kodiranje uzima svoj danak prilikom izvođenja upita što je prikazano u sljedećem pod-poglavlju.

## 4.2. Vremenska mjerenja

Tablice 4.4 i 4.5 prikazuju vrijeme potrebno za izgradnju stabla dvije inačica, kodirana i nekodirana. Tablica 4.4 sadrži mjerenja za umjetno stvorene nizove, dok 4.5 sadrži mjerenja za stvarne nizove. Iz tablice 4.4 vidljivo je kako je više vremena potrebno za izgradnju stabla sa kodiranjem za veće sekvence s većom abecedom, dok je iz tablice 4.5 vidljivo kako jedino za HIV potrebno više vremena za izgradnju stabla bez kodiranja, no razlika je zanemariva i vjerojatno je posljedica nepreciznosti mjerene metode.

**Tablica 4.4:** Vrijeme izgradnje stabla valića za umjetno stvorene datoteke

| Duljina sekvence | Veličina abecede | | | | | |
|---|---|---|---|---|---|---|
| | 4 | 15 | 26 | 4 | 15 | 26 |
| | Kodiran ($\mu s$) | | | Nekodiran ($\mu s$) | | |
| 100 | 61 | 136 | 120 | 61 | 96 | 136 |
| 1000 | 118 | 244 | 280 | 116 | 215 | 293 |
| 10000 | 598 | 1.349 | 1.516 | 596 | 1.133 | 1.478 |
| 100000 | 5.536 | 9.966 | 12.819 | 6.023 | 14.074 | 12.490 |
| 1000000 | 54.351 | 111.773 | 133.642 | 57.810 | 105.501 | 132.749 |




**Slika 4.2:** Vrijeme izgradnje stabla valića za umjetno stvorene nizove

**Tablica 4.5:** Vrijeme izgradnje stabla valića za FASTA datoteke

| Datoteka | Kodiran ($\mu s$) | Nekodiran ($\mu s$) |
|---|---|---|
| HIV | 118 | 182 |
| E Coli | 313.131 | 284.176 |
| Flu | 346 | 248 |
| Camelpox | 9.086 | 8.647 |
| Bact1 | 83.498 | 77.282 |

Tablice 4.6, 4.7 i 4.8 prikazuju rezultate izvršavanja upita nad umjetno stvorenim i stvarnim podacima, također u dvije inačice, kodiranih i nekodiranih stabla. Isti podaci iz tablica su prikazani u obliku grafa na slikama 4.3, 4.4, 4.5.

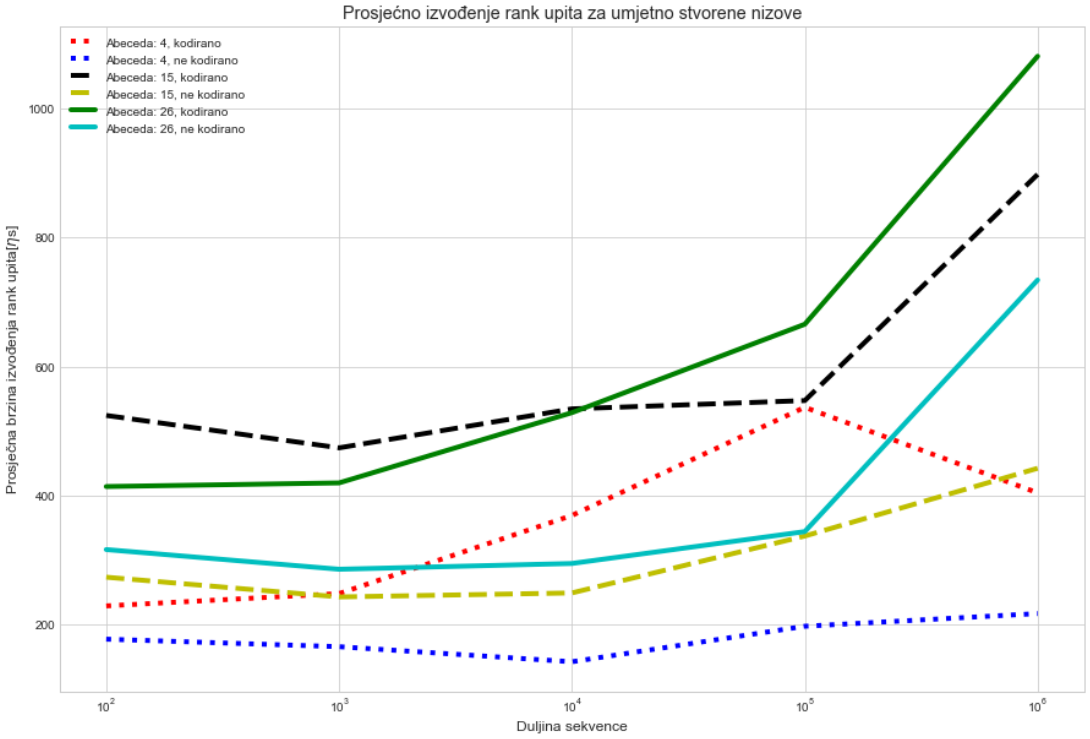
**Tablica 4.6:** Prosječno vrijeme izvršavanja upita za umjetno stvorene datoteke (kodiran)

| Duljina sekvence | Veličina abecede | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 4 | 15 | 26 | 4 | 15 | 26 | 4 | 15 | 26 |
| | rank ($ns$) | | | select ($ns$) | | | access ($ns$) | | |
| 100 | 229 | 524 | 414 | 520 | 1.010 | 775 | 509 | 1.122 | 805 |
| 1000 | 248 | 474 | 420 | 452 | 860 | 790 | 570 | 1.110 | 1.025 |
| 10000 | 369 | 534 | 529 | 743 | 1.143 | 931 | 855 | 1.381 | 1.307 |
| 100000 | 537 | 547 | 665 | 869 | 977 | 1.183 | 1.176 | 1.324 | 1.577 |
| 1000000 | 404 | 898 | 1.080 | 679 | 1.423 | 1.801 | 930 | 1.915 | 2.346 |

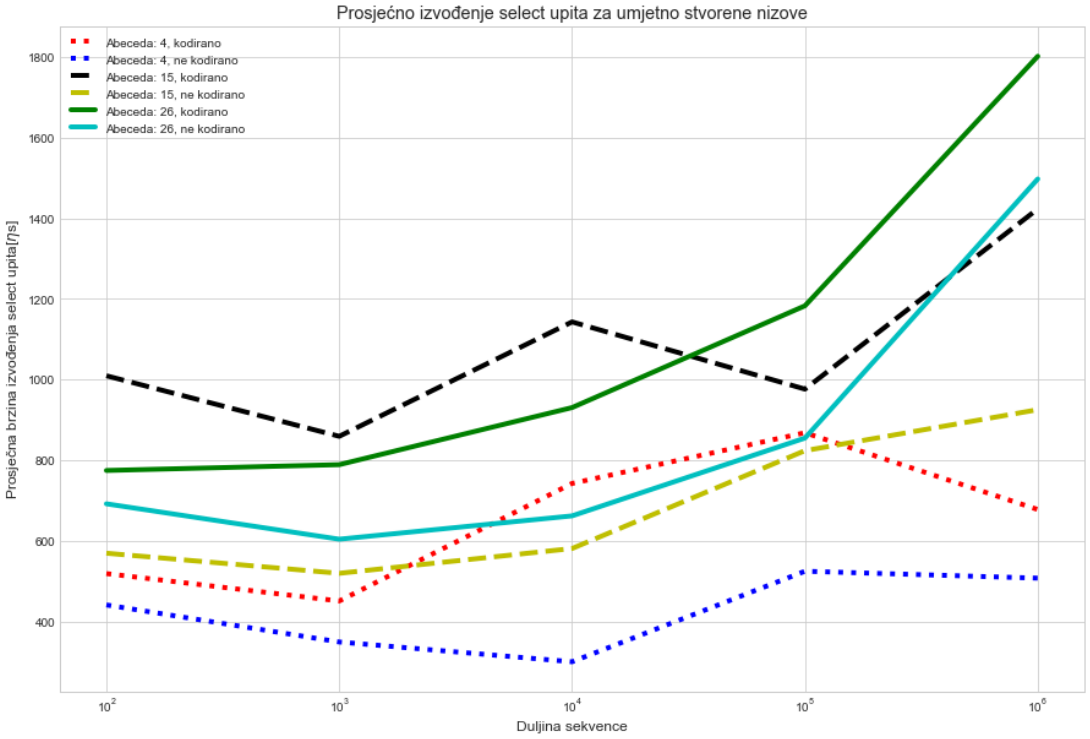
**Tablica 4.7:** Prosječno vrijeme izvršavanja upita za umjetno stvorene datoteke (nekodiran)

| Duljina sekvence | Veličina abecede | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 4 | 15 | 26 | 4 | 15 | 26 | 4 | 15 | 26 |
| | rank ($ns$) | | | select ($ns$) | | | access ($ns$) | | |
| 100 | 178 | 274 | 317 | 442 | 571 | 693 | 359 | 535 | 580 |
| 1000 | 166 | 243 | 285 | 351 | 521 | 605 | 322 | 507 | 595 |
| 10000 | 143 | 249 | 294 | 302 | 582 | 663 | 273 | 486 | 578 |
| 100000 | 198 | 337 | 344 | 526 | 824 | 856 | 342 | 596 | 619 |
| 1000000 | 217 | 442 | 734 | 509 | 926 | 1.497 | 365 | 657 | 1.037 |

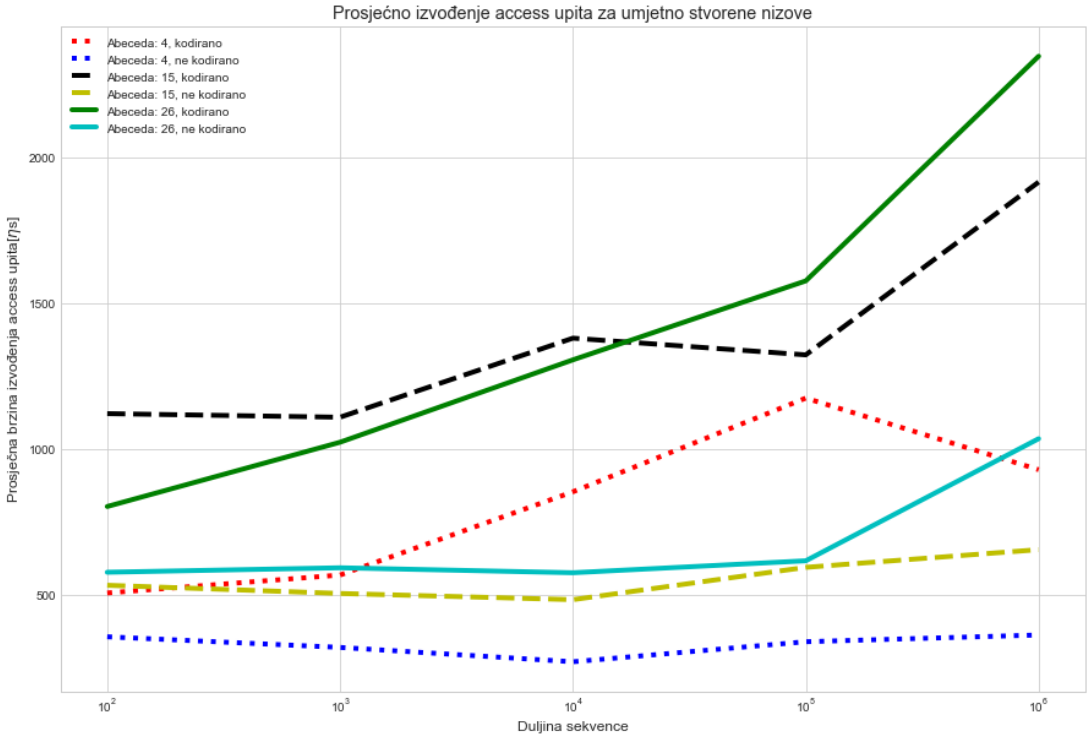
Iz priloženih grafova 4.3, 4.4, 4.5, izvršavanje upita je brže i do nekoliko puta bez kodiranja za sve veličine abeceda i sekvenci. Takav rezultat je i očekivan s obzirom da se dio vremena kod upita s kodiranjem potroši na dekodiranje.




**Slika 4.3:** Prosjećno vrijeme rank upita za umjetno stvorene nizove




**Slika 4.4:** Prosjećno vrijeme select upita za umjetno stvorene nizove




**Slika 4.5:** Prosjećno vrijeme access upita za umjetno stvorene nizove

**Tablica 4.8:** Prosječno vrijeme izvršavanja upita za FASTA datoteke

| Duljina sekvence | Veličina abecede | | | | | |
|---|---|---|---|---|---|---|
| | Kodiran ($ns$) | | | Nekodiran ($ns$) | | |
| | rank | select | access | rank | select | access |
| HIV | 248 | 451 | 587 | 251 | 522 | 507 |
| E Coli | 473 | 710 | 982 | 277 | 532 | 395 |
| Flu | 233 | 412 | 551 | 142 | 310 | 279 |
| Camelpox | 328 | 605 | 787 | 152 | 416 | 277 |
| Bact1 | 396 | 643 | 893 | 220 | 465 | 328 |

Tablica 4.5 sadrži mjerenja vremena upita za stvarne nizove za implementacije s kodiranjem i bez kodiranja. Kao i na umjetno stvorenim datotekama, i na stvarnim datotekama kodiranje višestruko uspori izvođenje upita.

## 4.3. Rezultati implementacije Brebrić, Kelemen, Volarić Horvat [6]

U nastavku su prikazani rezultati prošlogodišnjeg projekta [6]. U tablicama 4.9 i 4.10 su prikazana memorijska zauzeća stabla valića. Kolege su također koristili kodiranje zapisa te su njihovi rezultati mjerenja nešto bolji, no razlog tomu leži u korištenju manje memorijske preciznosti i *bug*-a u njihovom programu, te se to može uočiti na *E Coli* mjerenjima u tablici 4.10 gdje njihov program "podivlja" na nizovima preko nekoliko milijuna znakova.

**Tablica 4.9:** Potrošnja memorije stabla valića za umjetno stvorene datoteke

| Duljina sekvence | Veličina abecede | | |
|---|---|---|---|
| | 4 | 15 | 26 |
| | Memorija ($kB$) | | |
| 100 | 12 | 16 | 20 |
| 1000 | 16 | 20 | 24 |
| 10000 | 76 | 72 | 80 |
| 100000 | 396 | 416 | 420 |
| 1000000 | 3.584 | 4.848 | 5.356 |

**Tablica 4.10:** Potrošnja memorije stabla valića za FASTA datoteke

| Datoteka | Memorija ($kB$) |
|---|---|
| HIV | 20 |
| E Coli | 28.136 |
| Flu | 24 |
| Camelpox | 888 |
| Bact1 | 4.700 |

U tablicama 4.11 i 4.12 su prikazana vremena izgradnje stabla valića. Rezultati su približno jednaki našima, no njihova izgradnja stabla je također zahvaćena već spomenutim *bug*-om te problemi nastaju kod jako velikih nizova, što je vidljivo kod niza *E Coli* u tablici 4.12.

**Tablica 4.11:** Vrijeme izgradnje stabla valića za umjetno stvorene datoteke

| Duljina sekvence | Veličina abecede | | |
|---|---|---|---|
| | 4 | 15 | 26 |
| | Vrijeme izgradnje ($\mu s$) | | |
| 100 | 47 | 64 | 87 |
| 1000 | 95 | 163 | 213 |
| 10000 | 671 | 1.329 | 1.779 |
| 100000 | 5.635 | 9.563 | 11.834 |
| 1000000 | 52.041 | 106.479 | 113.794 |

**Tablica 4.12:** Vrijeme izgradnje stabla valića za FASTA datoteke

| Datoteka | Vrijeme ($\mu s$) |
|---|---|
| HIV | 98 |
| E Coli | 549.105 |
| Flu | 215 |
| Camelpox | 9.801 |
| Bact1 | 81.731 |

U tablicama 4.13 i 4.14 su prikazana prosječna vremena izvršavanja upita. Rezultati su približno jednaki našima s kodiranjem, čak nešto i lošija. Kolege nisu implementirali nekodiranu inačicu stabla, stoga te rezultate nije moguće usporediti, na kako je već vidljivo i u našim rezultatima, zasigurno bi bili bolji nego u slučaju kodiranih nizova.

**Tablica 4.13:** Prosječno vrijeme izvršavanja upita za umjetno stvorene datoteke

| Duljina sekvence | Veličina abecede | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 4 | 15 | 26 | 4 | 15 | 26 | 4 | 15 | 26 |
| | rank ($ns$) | | | select ($ns$) | | | access ($ns$) | | |
| 100 | 235 | 354 | 416 | 297 | 411 | 464 | 556 | 808 | 889 |
| 1000 | 268 | 408 | 454 | 318 | 474 | 547 | 631 | 982 | 1.134 |
| 10000 | 405 | 610 | 715 | 505 | 778 | 893 | 977 | .1543 | 1.767 |
| 100000 | 450 | 634 | 739 | 578 | 814 | 964 | .1062 | 1.560 | 1.841 |
| 1000000 | 514 | 708 | 1.059 | 707 | 974 | 1.463 | .1202 | 1.753 | 2.536 |

**Tablica 4.14:** Prosječno vrijeme izvršavanja upita za FASTA datoteke

| Duljina sekvence | Vrijeme ($ns$) | | |
|---|---|---|---|
| | rank | select | access |
| HIV | 267 | 325 | 653 |
| E Coli | 1.166 | 1.583 | 2.806 |
| Flu | 254 | 312 | 614 |
| Camelpox | 426 | 567 | 1.062 |
| Bact1 | 504 | 677 | 1.138 |

# 5. Opis implementacije

Projekt je implementiran u programskom jeziku C++, programski kod dostupan je u datoteci *src*. Kod koji implementira RRR strukturu nalazi se u datoteci *src/rrr*, kod za binarno stablo valića u datoteci *src/wavelet*, dok je kod koji sadrži pomoćne funkcije unutar datoteke *src/utilty* te kod koji sadrži dijeljene definicije unutar *src/shared*. Projekt je definiran *Cmake* alatom koji služi za generiranje projekata za razna interaktivna razvijateljska sučelja za C++.

**RRRSequence**

Ovaj razred predstavlja RRR strukturu, koja se koristi za spremanje čvorova stabla valića u kodiranom ili ne kodiranom obliku. Razred sadrži metode za izvođenje upita rank0, rank1, select0, select1, access. Postoje dvije implementacije ovog razreda, jedna je pomoću kodiranja i druga bez kodiranja opisanog u prethodnom poglavlju. Implementacija bez kodiranja se nalazi na glavnoj grani git repozitroija, dok je implementacija s kodiranjem na grani feature/packing-unpacking-sequence istog repozitorija.

**RRRTable**

Ovaj razred je pomoćna struktura koja se koristi kod izgradnje i izvršavanja upita nad RRR strukturama. Sadrži metode za dohvat ranka0 i ranka1 na $i$-tom indeksu unutar nekog bloka, odmaka za dani rank i blok te metodu za dohvat broja bitova odmaka za dani rank.

**WaveletTree**

Ovaj razred predstavlja binarno stablo valića te sadrži metode za vršenje rank, select i access upita.

**WaveletNode**

Ovaj razred predstavlja čvor binarnog stabla valića sadržava referencu na RRR strukturu stvorenu prilikom izgradnje stabla. Također sadrži metode za vršenje rank, select i access upita nad.

**bioinf_utility**

Kod koji se koristi kod izvođenja glavnog programa za čitanje datoteka i dohvat memorije za pojedine operacijske sustave.

**common**

Header koji sadrži sve podatkovne tipove definirane za korištenje unutar implementacije strukture.

**main**

Glavna ulazna točka programa, sadrži metode za mjerenje memorije i vremena.

# 6. Zaključak

Pohrana i obrada ulaznih sekvenci velike duljine predstavlja pravi izazov jer u isto vrijeme želimo smanjiti potrošnju memorije i smanjiti utjecaj na performanse obrade istih. Taj problem je posebno izražen prilikom analize genetskih sekvenci, stoga je od iznimne važnosti u području bioinformatike. U ovom radu, kao struktura koja bi riješila gore navedeni problem, prikazana je implementacija statičke strukture binarnog stabla valića kao RRR strukture.

U radu je opisana teorijska podloga RRR strukture te binarnog stabla valića. Također opisana su i dva moguća pristupa prilikom izgradnje RRR strukture: kodirani i nekodirani. Kodirani pristup smanjuje memorijsko zauzeće uz mali utjecaj na performanse prilikom obavljanja upita.

Na kraju rada dani su i rezultati mjerenja i analize strukture na stvarnim i umjetnim podacima. Prava moć, to jest skalabilnost, RRR strukture vidljiva je prilikom mjerenja prosječnog vremena izvršavanja upita, gdje su se za jako velik porast duljine ulazne sekvence, i preko nekoliko milijuna znakova, vremena upita tek nešto blago linearno povećala. Memorijski rast je ipak bio nešto veći, no to je zbog same strukture stabla valića. Kao što je već spomenuto, stablo valića je statička struktura te kao takva nema mogućnost dinamičke izmjene podataka, već bi se svaki put trebalo izgraditi novo stablo, no u području bioinformatike to nam i nije neki problem jer u većini slučajeva želimo samo vršiti upite nad trenutnim podacima.

# 7. Literatura


[1] A. Bowe. Rrr – a succinct rank/select index for bit vectors, 1.6.2011. URL `http://alexbowe.com/rrr/`. pristupano 2.12.2017.

[2] A. Bowe. Multiary wavelet trees in practice. Magistarski rad, School of Computer Science and Information Technology, RMIT University, 2010.

[3] A. Bowe. Wavelet trees – an introduction, 28.6.2011. URL `http://alexbowe.com/wavelet-trees/`. pristupano 2.12.2017.

[4] R. Gonzalez, S. Grabowski, V. Makien, i G. Navarro. Practical implementation of rank and select queries, 2005.

[5] R. Grossi, A. Gupta, i J. S. Vitter. High-order entropy-compressed text indexes, 2003.

[6] L. Volarić Horvat M. Breberić i J. Keleman. Izgradnja binarnog stabla valića kao rrr strukture, 2017.

[7] G. Navarro. Wavelet trees for all, 2012.