SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

BIOINFORMATIKA - PROJEKT

Izgradnja binarnog stabla valića kao RRR strukture

Duhović Nina

Peroš Matej

Zagreb, siječanj 2018.

Sadržaj

[1. Uvod 3](#_Toc504465481)

[2. Stablo valića 4](#_Toc504465482)

[2.1. Upit rank nad stablom valića 5](#_Toc504465483)

[2.2. Upit select nad stablom valića 7](#_Toc504465484)

[2.3. Upit access nad stablom valića 9](#_Toc504465485)

[3. RRR struktura podataka 10](#_Toc504465486)

[3.1. Upit rank nad RRR strukturom podataka 13](#_Toc504465487)

[3.2. Upit select nad RRR strukturom podataka 14](#_Toc504465488)

[3.3. Upit access nad RRR strukturom podataka 16](#_Toc504465489)

[4. Implementacija 17](#_Toc504465490)

[5. Eksperimentalni rezultati 18](#_Toc504465491)

[6. Zaključak 20](#_Toc504465492)

[7. Literatura 21](#_Toc504465493)

# Uvod

Stablo valića struktura je nad kojom se efikasno mogu izvršavati različiti upiti u svrhu rješavanja brojnih problema. U svrhu danjeg poboljšanja efikasnosti stabla valića u ovome radu implementirano je stablo valića koristeći RRR strukturu umjesto običnog bitvektora za spremanje niza i podnizova u čvorovima. RRR podatkovna struktura jest sažimajuća struktura podataka za pohranu binarnih nizova nad kojom se *rank* upiti izvode u konstantnom vremenu. Nad ulaznim nizom se vrši kompresija, a nad istime nije potrebno izvršiti dekompresiju kako bi se izvršavali upiti.

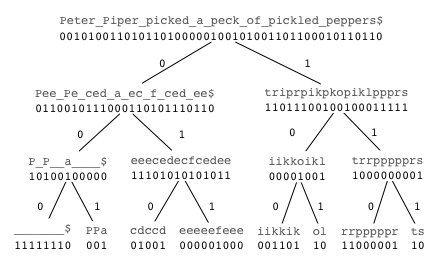
Nad stablom valića i RRR podatkovnom strukturom mogu se izvršavati operacije *rank, select i access* te su iste implementirane u svrhu ostvarenja ovog projekta.

# Stablo valića

Stablo valića(*eng. Wavelet tree*) *[1]* je podatkovna struktura koja rekurzivno rastavlja ulazni niz u strukturu binarnog stabla u ovisnosti o vrijednostima koje ulazni niz sadrži. Vrijednosti koje ulazni niz sadrži definiraju abecedu stabla. Kod stabla valića, svaki čvor sadrži podniz ulaznog niza takav da je abeceda lijevog čvora jednaka lijevoj polovini abecede čvora roditelja a analogno tome, abeceda desnog čvora jednaka desnoj polovini abecede čvora roditelja. Znakovi lijeve polovine abecede kodiraju se nulama, a desne polovine jedinicama. *[2]* Stablo se na taj način rekurzivno dijeli dok veličina abecede čvora lista nije jednaka ili manja od dva elemeta(ovisno o implementaciji), na način prikazan *Slikom 1.*

Stablo valića podržava 3 vrste upita: *rank*, *access* i *select,* koji će biti podrobnije opisani u nastavku.

Σ = [$,\_,P,a,c,d,e,f,i,k,l,o,p,r,s,t]



Σ = [s,t]

Σ = [p,r]

Σ = [i,k] 

Σ = [l,o]

Σ = [e,f]

Σ = [c,d]

Σ = [P,a]

Σ = [$,\_]

Σ = [p,r,s,t]

Σ = [i,k,l,o]

Σ = [$,\_,P,a]

Σ = [c,d,e,f]

Σ = [$,\_,P,a,c,d,e,f]

Σ = [i,k,l,o,p,r,s,t]

Slika 1. Prikaz stabla valića konstruiranog od ulaznog niza 'Peter\_Piper\_picked\_a\_peck\_of\_pickled\_peppers$' sa abecedom Σ = [$,\_,P,a,c,d,e,f,i,k,l,o,p,r,s,t]. U stablu su sadržani samo bitvektori dok je znakovna reprezentacija na slici prikazana radi lakoće razumijevanja.

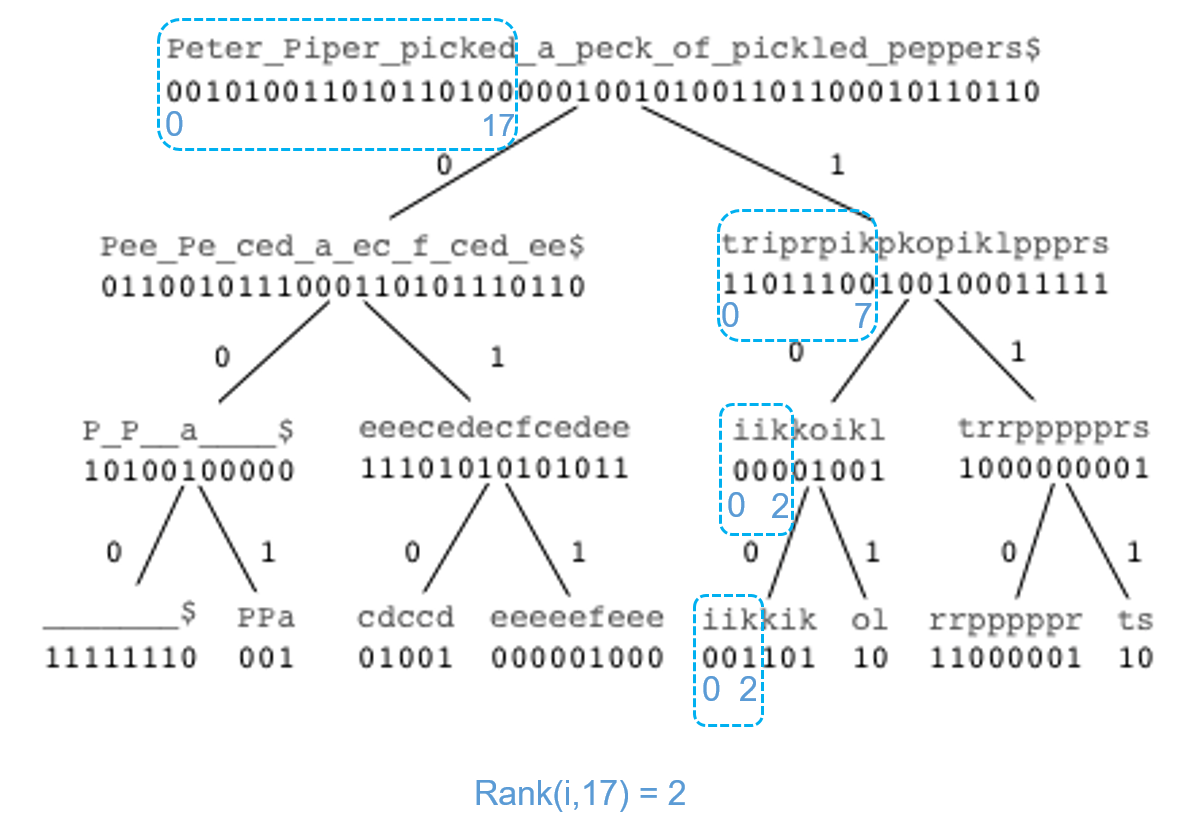
## Upit *rank* nad stablom valića

*Rank* znaka *c* u ulaznom nizu *S* do pozicije *p* zapisuje se kao *rankp(c)* i definiran je kao broj ponavljanja znaka *c* u podnizu *S[0,..,p].* Izvršavanje upita *rank* nad stablom valića je rekurzivno i započinje od korijenskog čvora te se pomiče niz stablo dok se ne dosegne čvor list čija abeceda sadrži znak *c*. U svakom čvoru računa se binarni rank znaka *c* u bitvektoru tog čvora te je isti korišten kao pozicijski parametar u čvoru djetetu. Upit *rank* razmatra samo definirani znak *c* do pozicije *p*, pa se indeks pozicije računa za svaki čvor dijete. Konačna vrijednost ranka je broj znakova *c* do konačnog indeksa čvora lista.

Algoritam za rank znaka *c* do pozicije *p* prikazan je *Tablicom 1*. *[1]*, a primjer rada algoritma *Slikom 2*. *[2]*

Složenost upita *rank* korištenjem RRR bitvektora jest ( gdje označava abecedu ulaznog niza) kako se upiti nad RRR bitvektorom mogu izvesti u konstantnom vremenu.Korištenjem trivijalnog bitvektora složenost postaje kako se u najgorem slučaju treba slijedno proći čitav binarni vektor.

|  |
| --- |
| **function**Rank(c,p)  **if** Self.IsLeaf **then**  **return** p  **endif**  CharBit ← bit representing c in bitmap of current node  o ← BinaryRank(CharBit,p)  **if** CharBit = 1 **then**  Rank ← RightChildNode.Rank(c,o)  **else**  Rank ← LeftChildNode.Rank(c,o)  **endif**  **return** Rank  **end function** |

Tablica 1. Algoritam izvršavanja upita Rank(c,p) 

Slika 2. Primjer izvršavanja upita Rank('i', 17) nad ulaznim nizom 'Peter\_Piper\_picked\_a\_peck\_of\_pickled\_peppers$'. Konačan rezultat je 2 jer se u listu znak 'i' ponavlja dva puta od indeksa 0 do indeksa 2.

## Upit select nad stablom valića

Upit *selectc(i*) rekurzivna je operacija kojom se dohvaća pozicija *p*(odnosno indeks ulaznog niza) *i*-tog znaka *c*.

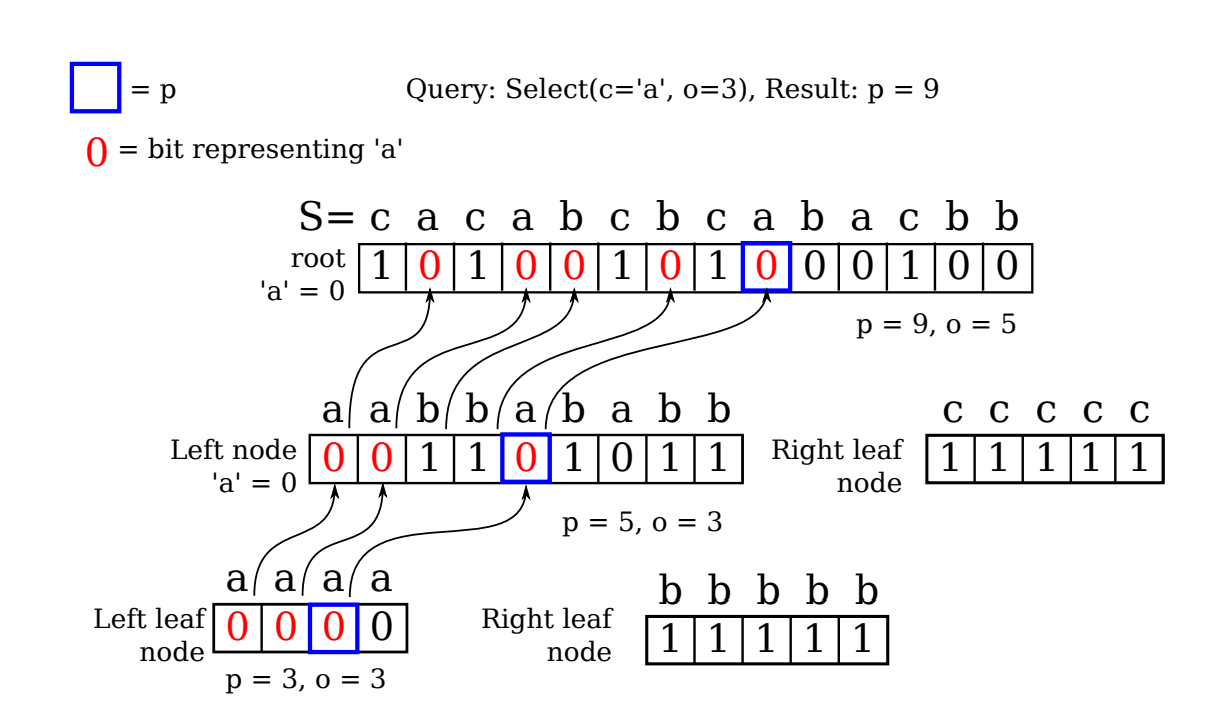
Spuštanjem niz stablo pronalazi se čvor list čija abeceda sadrži znak c te se u tom čvoru pronalazi i-ti znak c, odnosno njegov indeks. Zatim započinje penjanje stablom: ukoliko je pronađeni čvor lijevo dijete poziva se *select0(i*) te analogno tome, ukoliko je pronađeni čvor lijevo dijete poziva se *select1(i*) i procedura se ponavlja do korijenskog čvora, u svakom koraku ažurirajući indeks traženog znaka.

Algoritam za select i-tog znaka *c* prikazan je *Tablicom 2*. *[1]* a primjer rada algoritma *Slikom 3. [1]*

Vremenska složenost upita select nad stablom valića iznosi ukoliko se upiti nad binarnim vektorima izvršavaju u konstantnom vremenu.

|  |
| --- |
| **function**Select(c,i)  Leaf ← GetLeaf(c)  **if** Leaf.isRightChild **then**  CharBit ← 1  else  CharBit ← 0  **endif**  **return** Leaf.Parent.SelectRec(CharBit,i)  **end function**  **function**SelectRec(CharBit,i)  Position ← BinarySelect(CharBit, Occurrence)  **if** Self.isRootNode **then**  **return** Position  **end if**  **if** Self.isRightChild **then**  CharBit ← 1  **else**  CharBit ← 0  **end if**  **return** Parent.SelectRec(CharBit, Position)  **end function**    **function** GetLeaf(c)  **if** Self.isLeaf **then**  **return** Self  **end if**  **if** c ∈ Σright **then**  **return** RightChild.GetLeaf(c)  **else**  **return** LeftChild.GetLeaf(c)  **end if**  **end function** |

Tablica 2. Algoritam izvršavanja upita Select(c,i)



Slika 3. Primjer rada algoritma za upit Select('a',3) nad ulaznim nizom 'cacabcbcabacbb' čiji je rezultat 9.

## Upit access nad stablom valića

Upitom access(p) pronalazi se znak c na poziciji p početnog ulaznog niza. Algoritam započinje u korijenskom čvoru te ukoliko je rezultat upita 0, prelazimo na lijevo dijete, u suprotnome ako je rezultat upita 1, prelazimo na desno dijete. Prije prelaska u čvor dijete potrebno je znati kojem indeksu korelira pozicija p u podnizu. Algoritam postupka prikazan je *Tablicom 3*. *[1]*

Vremenska složenost upita nad stablom valića iznosi ukoliko se upiti nad binarnim vektorima izvršavaju u konstantnom vremenu.

Tablica 3. Algoritam izvršavanja upita Access(p)

|  |
| --- |
| **function**Access(p)  v ← RootNode  r ← p  **while** v.isNotNull  **if** v.isNotRootNode  r ← r-1  **if** accesv(r)=0  r ← rank0v(r)  **if** vleft.isNull  **return** first element of  **endif**  v ← vleft  **else**  r ← rank1v(r)  **if** vright.isNull  **return** last element of  **endif**  **endif**  **return** 0  **end function** |

# RRR struktura podataka

RRR struktura podataka*[5],* tako nazvana po svojim tvorcima Ramanu, Ramanu i Rau, jest sažimajuća struktura podataka za pohranu binarnih nizova nad kojom se *rank* upiti izvode u konstantnom vremenu.Ulazni binarni niz se komprimira, no isti se ne mora dekomprimirati kako bi se nad njime izvršavali upiti.

Kako bi se RRR struktura izgradila, prvo je potrebno podijeliti izvorni binarni vektor na superblokove veličine s, gdje je

koji se zatim dodatno dijele na blokove veličine b bita, gdje je

Ukupni broj superblokova f, odnosno faktor superbloka možemo izračunati dijeljenjem veličine superbloka, s veličinom bloka b. [4] Primjer je prikazan *Slikom 4*. [3]



Slika 4 Podjela RRR strukture na blokove i superblokove, gdje je b=5 a f=3.

Prvo se blokovi zamijenjuju parom vrijednosti, vrijednošću razreda C(popcount bloka) i vrijednošću pomaka O(indeks permutacije bloka te klase). Permutacije se spremaju u tablicu(Slika 6.) [3] čiji je ključ klasa a vrijednosti su svi blokovi (permutacije) te klase. Tako koditani blokovi pohranjuju se u obliku bitvektora(*Slika 5*.). [3]



Slika 5. Izgradnja RRR strukture iz bitvektora

Slika 6. Izgled tablice u kojoj se pohranjuju pomaci O

Sažimanje se dobiva uporabom varijabilnih duljina odmaka u RRR strukturi. Broj bitova za klase je uvijek konstantan i iznosi log2(b) + 1 gdje je b duljina bloka. Broj bitova za odmake odnosno varira od klase do klase i iznosi log2() gdje je b veličina bloka, a C je vrijednost klase odnosno broj permutacija.

Superblokovi grupiraju blokove i spremaju sumu cijelog bloka te pokazivač na prvi blok idućeg superbloka, na način prikazan *Slikom 7*., [3] a svrha takvog grupiranja blokova u superblokove jest skraćivanje vremena potrebno za izvođenje upita. Nakon popunjavanja superbloka ažuriraju se njegovi podaci (, indeks) te se dodaje novi blok.



Slika 7. Izgled superblokova

Kao i nad stablom valića, nad RRR strukturom mogu se izvršavati upiti rank, select i access.

## Upit rank nad RRR strukturom podataka

Upit rank nad RRR strukturom podataka vraća broj pojavljivanja bita do uključivo -te pozicije u ulaznom binarnom nizu.

Prvo se izračuna globalni indeks bloka: i indeks superbloka u kojem se nalazi dobiveni blok: ,gdje je is indeks superbloka). Za početnu sumu postavlja se suma svih prethodnih superblokova izračunata prilikom izgradnje RRR strukutre te se iterira po blokovima dobivenog superbloka pa sve do dobivenog bloka, te se iz (c, o) parova sumi dodaje c. Kada dođemo do dobivenog bloka, pomoću vrijednosti c i o dohvaćamo ga iz tablice odmaka, te računamo rank tog bloka za j = i mod b bitova. Pseudokod postupka prikazan je *Tablicom 4*. *.[4]*

Vremenska složenost operacije nad RRR strukturom je konstantna odnosno O(1).

|  |
| --- |
| *def :*  🡨  🡨  🡨  🡨  Dok :  🡨  🡨 sljedeći blok  🡨  🡨  vrati  def :  vrati |

Tablica 4. Pseudokod izvršavanja upita rank

## Upit select nad RRR strukturom podataka

Upit select nad RRR strukturom vraća indeks i-tog bita b u ulaznom binarnom nizu.

Kako bi se upit izvršio, prvo se pronalazi prethodnik superbloka za koji vrijedi da je pohranjena suma veća od i te se trenutni rezultat postavlja na j(indeks dobivenog superbloka) pomnožen veličinom superbloka. Zatim se iterira po blokovima dobivenog superbloka, njihove klase dodaju se ukupnoj sumi i povećava se rezultat dodajući veličinu bloka, dok se ne dostigneblok za kojeg vrijedi da je suma bloka zbrojena s ukupna suma veća od i. Dobiveni blok se dohvaća iz RRR tablice te se na ukupnu sumu dodaju jedinice ili nule(ovisi o vrijednosti ulaznog znaka u tom čvoru) iz tog bloka, sve dok ukupna suma nije jednaka i. Pseudokod postupka prikazan je *Tablicom 5*. *.[4]*

Vremenska složenost operacije nad RRR strukturom koja ima N superblokova i b bitova po bloku je .

|  |
| --- |
| * – vraća kumulativni rank superbloka * – vraća prvi blok superbloka * i – vraćaju razred odnosno pomak bloka * – vraća indeks na kojem se pojavljuje i-ta jedinica ili nula () u zadanom bloku   *def :*  🡨  🡨  🡨  🡨  Dok :  🡨  Ako :  Izađi iz petlje  Inače:  🡨  🡨 sljedeći blok  🡨  🡨  vrati  *def :*  🡨  🡨  🡨  🡨  Dok :  🡨  Ako :  Izađi iz petlje  Inače:  🡨  🡨 sljedeći blok  🡨  🡨  vrati |

Tablica 5. Pseudokod izvršavanja upita select

## Upit access nad RRR strukturom podataka

Upit access nad RRR strukturom vraća bit na i-toj poziciji u ulaznom binarnom nizu. Koristi već opisan upit rank, te je pseudokod postupka prikazan Tablicom 6*.[4]*

Kako je vremenska složenost izvođenja upita access jest konstantna, odnosno O(1), kako koristi upit rank.

|  |
| --- |
| *def :*  *index = i / b*  *index\_in\_block = index mod b*  *x = prva vrijednost(lookup\_table (block[index]))*  *y = (x >> (block\_size - 1 - index\_in\_block)) & 1*  *return y* |

Tablica 5. Pseudokod izvršavanja upita select

# Implementacija

Projekt je implementiran u programskom jeziku C++. U nastavku su opisane komponente implementacije:

* wavelet\_tree.h - implementacija stabla valića, za stvaranje stabla koristi abecedu(dictionary) i čvor(node), te implementacija trivijalnog bitvektora
* rrr.cpp – implementacija rrr podatkovne strukture, te funkcije za ostvarivanje upita rank, access i select nad rrr podatkovnom strukturom.
* rrr\_table.cpp – implementacija lookup tablice korištene za izgradnju rrr podatkovne strukture
* main.cpp – glavni program ostvaren za testiranje cjelokupnog implementiranog rješenja (učitavanje FASTA datoteke, mjerenje vremenske statistike te memorijske potrošnje)

# Eksperimentalni rezultati

Eksperimentalnim rezultatima*(Tablica 6.,7.,8.)* dolazimo do zaključka da veći ulazni nizovi zahtijevaju više vremena za izgradnju, što je u skladu s očekivanjima. Također, nizovi jednake duljine ali veće abecede zahtjevaju dublje stablo pa zbog toga trebaju i više vremena za izgradnju, kako svaki dodatni čvor zahtijeva dodatno vrijeme za izgradnju RRR strukture.

Tablica 7. Eksperimentalni podatci nad stvarnim podatcima

|  |  |  |
| --- | --- | --- |
| Broj znakova u datoteci / Broj znakova abecede | 4 | 28 |
| 100 | < 1 ms | 4 ms |
| 1.000 | 4 ms | 24 ms |
| 10.000 | 44 ms | 110 ms |
| 100.000 | 555 ms | 1300 ms |
| 1.000.000 | 7787 ms | 17645 ms |
| Tablica 6. Eksperimentalni podatci nad sintetskim podatcima | | |
| Ulazna datoteka | Broj znakova | Veličina abecede | Vrijeme izgradnje |
| HIV | 999 | 5 | 6 ms |
| Coli | 3.657 | 4 | 14 ms |
| Flu | 3.990 | 4 | 15 ms |
| Camelpox | 205.719 | 4 | 1079 ms |
| Bact1 | 1.587.120 | 4 | 14988 ms |
| Pig | 1.637.716 | 5 | 18170 ms |
| Bact2 | 3.018.312 | 4 | 31037 ms |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ulazna datoteka | Broj znakova | Veličina abecede |  |  |  |
| 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 | 3 | 3 | 3 |
| Pig | 1.637.716 | 5 | 4 | 4 | 3 |
| Bact2 | 3.018.312 | 4 | 3 | 3 | 3 |

Tablica 7. Eksperimentalni podatci za upite

Vrijeme izvršavanja upita povećava se logaritamski. Veća abeceda povećava dubinu stabla i time ukupno vrijeme izvršavanja.

Vrijeme izvršavanja upita proporcionalno je vremenu izvršavanja upita kako se pomoću njega izvršava, a vrijeme povećavanja upita jest logaritamsko te dodatno povećano u ovisnosti s dubinom stabla.

# Zaključak

U svrhu ostvarenja ovog projekta opisano je i implementirano stablo valića koje koristi RRR podatkovnu strukturu. Implementirani su upiti rank, select te access koji se mogu izvršavati nad ulaznim nizom(i RRR strukturom). Eksperimentalnim rezultatima podvrđuju se pretpostavke o vremenima potrebnim za izgradnju stabla te izvođenje upita nad ulaznim nizom.

# Literatura

[1] J. Knudsen, R. L. Pedersen; Engineering Rank and Select Queries on Wavelet Trees; Diplomski rad; 2015.

[2] Bowe, A. Wavelet Trees

<http://alexbowe.com/wavelet-trees/>

[3] Bowe, A. RRR – A Succinct Rank/Select Indeks for Bit Vectors <http://alexbowe.com/rrr/>

[4] D. Čaušević,H. Ćoralić; Izgradnja binarnog stabla valića (eng. wavelet tree) kao RRR strukture; Bioinformatika-projekt; 2016.

[5] 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.