SVEUČILIŠTE U ZAGREBU

**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

PROJEKT IZ PREDMETA BIOINFORMATIKA

**IZGRADNJA BINARNOG STABLA VALIĆA (eng. *Wavelet tree)* KAO RRR STRUKTURE**

Boris Kanski

Branimir Pervan

Ivan Karlo Pintar

Luka Postružin

Nikola Kurdija

Vlado Huško

Zagreb, siječanj 2015.

Sadržaj

[1. Uvod 3](#_Toc409033652)

[2. Općeniti primjer algoritma 8](#_Toc409033653)

[2.1. Izgradnja stabla 8](#_Toc409033654)

[2.2. Rank operacija 9](#_Toc409033655)

[2.3. Select operacija 10](#_Toc409033656)

[3. Implementacije 11](#_Toc409033657)

[3.1. Implementacija u programskom jeziku C 11](#_Toc409033658)

[3.2. Implementacija u programskom jeziku Java 12](#_Toc409033659)

[3.3. Implementacija u programskom jeziku C++ 14](#_Toc409033660)

[3.4. Implementacija u programskom jeziku C# (v2) 15](#_Toc409033661)

[4. Analiza rezultata 16](#_Toc409033662)

[5. Zaključak 23](#_Toc409033663)

[6. Literatura 24](#_Toc409033664)

# Uvod

Stablo valića (eng. Wavelet tree) struktura je koja se koristi za sažimanje niza znakova. Stablo se gradi rekurzivnim dijeljenjem abecede ulaznog niza u dva skupa. Listovi stabla odgovaraju točno određenom znaku abecede, a svaki čvor u *bit-vektor* pohranjuje u koji skup abecede znak spada (Slika 1).



Slika 1. – primjer stabla valića

Opisano i prikazano stablo valića je tzv. binarno stablo valića (eng. binary wavelet tree).

Ovakva struktura pogodna je za provođenje operacije *rank(i)*. Rank (i) operacija nad nekim *bit-vektorom* će vratiti broj bitova koji su postavljeni u 1, od nulte do *i*-te pozicije ( [0,i] ). Npr za bit-vektor „0010110“ operacija Rank (3) vratiti će 1 jer se od nulte do treće pozicije nalazi samo jedna jedinica. Operacija rank zapravo je generalizacija operacije popcount tj. Hammingove težine.

Operacija rank može se proširiti tako da se uvede dodatni argument koji predstavlja znak u nekom nizu. Npr. operacija rank (a, 4) nad nizom „banana“ vratit će 2 jer se znak 'a' dvaput pojavljuje u do pozicije 4 ulaznog niza.

Složenost ovakve operacije očigledno je O(N) gdje je N duljina ulaznog niza. Konstrukcijom stabla valića za ulazni niz duljine N, složenost operacije postaje O(log2(A)) gdje je A veličina abecede. Ako spremamo ulazni niz kao bit-vektor (bez RRR) ukupna složenost je O(N\*log2(A)) - log2(A) operacija rank (c, i) koje pozivaju binarne rank operacije rank(i) koje imaju složenost O(n).

Operaciju rank (c, i) nad stablom valića provodimo na sljedeći način (krećemo od korijenskog čvora):

1. Nad bit-vektorom pohranjenim u čvoru provedemo operaciju i = rank (value(c), i) gdje je value (c) vrijednost znaka c u tom čvoru (0 ili 1).
2. Ako je value(c) 0 krećemo u lijevo dijete, ako je 1 u desno dijete.
3. Ponavljamo korake 1. i 2. sve dok ne dođemo do lista, ali tako da je parametar ***i*** u operaciji rank uvijek rezultat prethodno izračunate operacije (u roditeljskom čvoru).

Sljedeća operacija koja je pogodna za izvođenje nad stablom valića je select(i). Operacija select, svrojevrsni je inverz operacije rank – rezultat je pozivija i-te jedinice u bit-vektoru. Stoga primjećujemo da vrijedi:

Rank (Select(i)) = i (napomena: obrat ne vrijedi nužno)

Operacija select (c, i) provodi se nad stablom valića slično kao i rank, samo ovdje ne počinjemo u korijenu, nego u listu u kojem je spremljena „konačna vrijednost“ znaka c:

1. Nad bit-vektorom pohranjenim u čvoru provedemo operaciju i =select (value(c), i) gdje je value (c) vrijednost znaka c u tom čvoru (0 ili 1).
2. Prelazimo na roditeljski čvor.
3. Ponavljamo korake 1. i 2. sve dok ne dođemo do korijena, ali tako da je parametar ***i*** u operaciji rank uvijek rezultat prethodno izračunate operacije (u čvoru djetetu).

RRR struktura koristi se za sažimanje bit-vektora te svodi složenost binarne rank operacije nad pojedinim čvorom na O(1). Operaciju je moguće obaviti bez dekompresije (eng. Succinct data structure). Za izgradnju RRR strukture, bit-vektor rastavljamo na blokove i superblokove. Npr. za b=5 i f=3 gdje je b veličina bloka, a f veličina superbloka nad bit-vektorom dobivamo slučaj sa slike 2.



Slika 2. – podijela bit-vektora na blokove i superblokove

U RRR strukturi blokovi se zamjenjuju uređenim parovima (c, o) gdje je **c** klasa (eng. class), a **o** odmak (eng. offset). Klasa označava popcount bloka, a odmak označava indeks permutacije bloka te klase. Te permutacije spremljene su u tablici – ključ je klasa a vrijednosti su svi blokovi (permutacije) te klase. Primjer je na slikama 3. i 4.



Slika 3. – pretvorba bit-vektora u RRR strukturu



Slika 4. – Izgled tablice u kojoj se pohranjuju odmaci (offsets)

Broj klasa će uvijek odgovarati veličini bloka + 1 (jer je najveći popcount bloka njegova duljina – slučaj gdje su sve jedinice). 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 je kao što je već rečeno varijabilan, odnosno varira od klase do klase i iznosi log2() gdje je b veličina bloka, a C je vrijednost klase (zapravo se radi o broju permutacija).

Kako bi postigli složenost O(1) koristimo superblokove. Superblokovi grupiraju blokove u cjeline te spremaju sumu cijelog bloka te pokazivač na prvi blok sljedećeg superbloka (slika 5.). Time postižemo da će se za izračun operacije rank iterirati po najviše f blokova (f je „duljina“ superbloka) jer će svi prethodni superblokovi biti unaprijed izračunati.



Slika 5. – Superblokovi

Računanje operacija rank (i) sada izgleda ovako:

1. Izračunamo indeks bloka: (ib je globalni indeks bloka).
2. Izračunamo indeks superbloka u kojem se nalazi dobiveni blok: (is je indeks superbloka).
3. Kao početnu sumu postavljamo sumu svih prethodnih superblokova (koja je izračunata prilikom izgradnje RRR strukutre).
4. Iteriramo po blokovima dobivenog superbloka pa sve do dobivenog bloka, te iz (c, o) parova dodajemo c u sumu (klasa je ujedno i popcount bloka).
5. 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.

Na sličan način se iz RRR strukture računa i operacija select (i):

1. Pronađi prethodnik superbloka za koji vrijedi da je pohranjena suma > i
2. Postavi trenutni rezultat na j \* veličina superbloka gdje je j indeks dobivenog superbloka
3. Iteriraj po blokovima dobivenog superbloka, te dodaj njihove klase ukupnoj sumi i povećavaj rezultat dodajući veličinu bloka , dok ne dođeš do bloka za kojeg vrijedi: suma bloka + ukupna suma > i
4. Dobiveni blok dohvati iz RRR tablice te na ukupnu sumu pridodaj jedinice (ili nule, ovisi o vrijednosti ulaznog znaka u tom čvoru) iz tog bloka, sve do ukupna suma ne bude jednaka i.

Dakle, ako spremimo ulazni niz kao RRR ukupna složenost rank (i select) operacija bit će O(log2(A)) - log2(A) poziva rank (c, i) od kojih svaki poziva binarnu rank operaciju nad RRR strukturom čija je složenost O(1). Binarna select operacija nad RRR strukturom ima logaritamsku složenost.

Izraz za memorijsku složenost stabla valića dan je s: nH0(S) + o(|S| log2(A) ) gdje je H0(S) nulta entropija ulaznog niza, a A veličina abecede. Izraz za memorijsko zauzeće RRR strukture kada njome kodiramo bit-vektor B glasi: nH0(B) + o(n) bitova.

# Općeniti primjer algoritma

## Izgradnja stabla

Uzmimo za primjer neki kratki genom: **ATGATCGTAAGGCTCA** -> Abeceda: {A, T, G, C}

Prvo gradimo binarno stablo valića:

ATGATCGTAAGGCTCA

0010011000111010

ATATTA ATA GCGGGCC

010110010 0100011

U korijenskom čvoru abeceda je {A, T, G, C}. Nju dijelimo na 2 dijela te pridružujemo svakom znaku abecede vrijednost 0 ili 1. Time dobivamo {A=0, T=0, G=1, C=1} te zapisujemo ulazni niz u bit-vektor preslikavajući znak u odgovarajući bit. U lijevo dijete ulazi niz koji sadrži samo znakove koji su u abecedi korijena imali vrijednost 0 (isto tako u desno dijete idu oni s vrijednosti 1). Abeceda se ponovno dijeli {A=0, T=1} u lijevom dijetetu i {G=0, C=1} u desnom dijetetu. Ponovno preslikavamo niz u bit-vektor. U ovom trenutku izgradnja stabla završava jer u krajnjim čvorovima imamo točno definirano koji znak ima koju vrijednost. Napomena: ulazni nizovi se ne spremaju u čvorovima, njega se može dobiti rekonstrukcijom iz bit-vektora.

Nakon toga potrebno je iz bit-vektora dobiti RRR strukturu. Prvo određujemo veličinu blokova i superblokova po formulama:

Veličina bloka: , gdje je N duljina ulaznog niza.

Veličina superbloka: , gdje je b veličina bloka, a N duljina ulaznog niza.

U našem slučaju N = 16, uvrštavanjem u gore navedene formule, dobivamo b = 2 i f = 8. To znači da ćemo bit-vektor predstaviti sa 8 blokova i 2 superbloka (4 bloka su 1 superblok). No prvo treba napraviti „lookup“ tablicu odmaka. Pošto je duljina bloka 2 postoje 2 +1 = 3 različite klase, svaka sa različitih odmaka (eng. offset).

Tablica:

[0] -> {00};

[1] -> {01, 10};

[2] -> {11};

Pomoću tablice sada gradimo RRR strukturu za bit-vektor u korijenu čvora:

00 | 10 | 01 | 10 | 00 | 11 | 10 | 10

(0,0)(1,1)(1,0)(1,1)(0,0)(2,0)(1,1)(1,1)

Za zapis klase nam je potrebno 2 bita, a za prikaz odmaka 1 bit (uobičajno će se odmaci u pojedinim klasama prikazivati različitim brojem bitova, no ovdje se zbog jednostavnosti primjera svi prikazuju s jednim bitom).

Kada kodiramo uređene parove dobivamo:

000 | 011 | 010 | 011 | 000 | 100 | 011 | 011

Prilikom konstrukcije pamtimo i sume unutar superblokova te pokazivače na početak sljedećeg superbloka tj. ukupne duljine prethodnih superblokova (kako bi znali gdje idući počinje).

Sume: 3 | 7

Duljine: 12 | 24

Ekvivalentne pretvorbe obavljaju se za lijevi i desni čvor. Sada imamo potpuno inicijalizirano binarno stablo valića s RRR strukturom.

## Rank operacija

Rank operaciju pozivamo s dva parametra: prvi je znak čija pojavljivanja u nizu želimo izbrojati, a drugi je granica do koje brojimo. Npr. Rank (A, 6) će nam reći koliko se puta A pojavljuje u prvih 6 znakova.

Pozovimo npr. Rank ('G', 13).

Algoritam izvodimo od korijenskog čvora. Znak G u korijenskom je čvoru predstavljen s 1 što znači da je potrebno izbrojati jedinice do 13. znaka. Pošto smo pohranili bit-vektor kao RRR računamo rank na sljedeći način:

1. Računamo indeks bloka: (cjelobrojno dijeljenje).
2. Računamo indeks superbloka: (faktor superbloka je 4 jer superblok obuhvaća 4 bloka).
3. Trenutna suma postaje suma prethodnih superblokova (suma[0] = 3), a superblok s traženim blokom počinje na odmak[0] = 12.
4. Iteriramo po blokovima do traženog bloka (u ovom slučaju to su 2 bloka: 000 | 100) te zbrajanjem njihovih klasa i dodavanjem sumi dobivamo:

suma = suma[0] + 0 + 2 = 3 + 0 + 2 = 5

1. Sada iz idućeg bloka (011) treba dobiti originalni blok te izračunati njegov popcount. Iz tablice dobivamo da je [1][1] -> 10, te nam je konačna suma = 6.

Ovdje, postupak nad korijenskim čvorom završava, a operacija rank se poziva nad desnim dijetetom korijenskog čvora (jer je vrijednost od G u riječniku 1). Kao što je i rečeno, rezultat roditeljske rank operacije postaje granica rank operacije u dijetetu, tako da se nad tim čvorem poziva rank (G, 6) te će rezultat te operacije biti konačno rješenje (nema više djece). Konačni rezultat: rank (G, 13) = 3.

## Select operacija

Kao ulaznu strukturu uzet ćemo stablo valića iz prethodnog primjera. Pokušajmo sada izračunati select (G, 3). Dakle tražimo poziciju trećeg slova G u ulaznom nizu. Kao što algoritam kaže počinjemo iz lista u kojem se nalazi G. U našem slučaju to je desno dijete korijena u kojem je G = 0.

Dakle brojimo nule dok ne izbrojimo 3 i vratimo zadnju poziciju. Lako možemo vidjeti da je rezultat select (G, 3) = 4

U ovom trenutku se pomičemo u roditeljski čvor (korijen). U korijenu je G = 1, dakle brojimo jedinice.

Na sličan način kao za rank operaciju nad RRR-om (detalji su objašnjeni u Uvodu), dobivamo da je rezultat operacije select (G, 4) = 11.

# Implementacije

## Implementacija u programskom jeziku C

Implementacija u programskom jeziku C strukturirana je u nekoliko datoteka.

U datotekama WaveletTree.c i WaveletTree.h napisane su funkcije za izgradnju stabla valića i dane su implementacije operacija rank i select (visoke razine) te rank i select koje rade s bit-vektorima.

Funkcija *buildWaveletTree* gradi stablo iz ulaznog niza koji joj se šalje kao parametar. Ona će iz niza „izvući“ abecedu (funkcija *extractAlphabet*) te će predati abecedu i niz funkciji *buildWaveletNode.* Funkcija *buildWaveletNode* rekurzivno gradi stablo stvaranjem struktura WaveletNode. Ona će prvo izgraditi bit-mapu pa iz nje i RRR strukturu, a zatim će podijeliti abecedu i ulazni niz na lijevi i desni te će ih rekurzivno proslijediti za lijevo i desno dijete.

Funkcije *rankOperation* i *selectOperation* su implementacije algoritama rank(c, i) i select (c, i) koje su opisane u uvodu.

Funkcije popcount i selectOnBitmap su implementacije binarnih rank i select operacija. U oba slučaja iterira se po bit-vektoru te se broje jedinice ili nule (uvjet zaustavljanja je drukčiji).

Deklarirane strukture su WaveletTree i WaveletNode. WaveletTree sadrži samo pokazivač na korijenski WaveletNode. WaveletNode sadrži pokazivače na bit-vektor, RRR-strukturu, abecedu te čvorove-djecu i roditelja.

U datotekama RRR.c i RRR.h nalaze se implementacije funkcija za izgradnju RRR strukture iz bit-vektora, računanje rank i select operacija nad RRR strukturom te funkcija koja računa veličine bloka i superbloka. Također sadrži funkciju za izgradnju RRR tablica.

Funkcija bitmapToRRR prima ulazni bit-vektor te na temelju izračunatih veličina blokova i superblokova gradi RRR strukturu na način koji je opisan u uvodu.

Funkcije popcountRRR i selectRRR implementacije su binarnih rank(i) i select(i) operacija koje su također opisane u uvodu.

Funkcija buildRRRTable na temelju izračunatih veličina blokova i superblokova gradi RRR „lookup“ tablicu.

Deklarirane strukture su RRRStruct, RRRTable i RRRTableEntry. RRRStruct predstavlja RRR strukturu te sadrži pokazivač na bit-vektor koji je predstavlja. Također sadrži spremljene vrijednosti suma i odmaka superblokova u tom bit-vektoru. RRRTable predstavlja „lookup“ tablicu te sadrži niz RRRTableEntry struktura. RRRTableEntry predstavlja zapis jedne klase u tablici. Sastoji se od klase (suma bloka) te pokazivača na blokove koje pripadaju toj klasi.

U datotekama Utils.c i Utils.h nalaze se funkcije za rad sa strukturom Dictionary. Struktura Dictionary predstavljat će abecedu pohranjenu u čvoru– sastoji se od varijable char i njezine vrijednosti boolean.

Funkcija *extractAlphabet* izvlači abecedu iz danog niza te im pridjeljuje vrijednosti. Funkcija *splitAlphabet* dijeli zadanu abecedu na lijevu i desnu te ih vraća preko pokazivača. Funkcije *charInDictionary* i *getDictionarValue* provjeravaju postojanje odnosno vraćaju vrijednost danog znaka.

Uz Dictionary, deklarirana je i struktura BitMap koja predstavlja bit-vektor. BitMap sadrži pokazivač na polje char znakova, ali tako da se unutar jednog char znaka nalazi 8 bitova.

U datoteci main.c nalazi se funkcija main() koja prima ime ulazne datoteke, rank ili select, znak i granicu. Funkcija parsira FASTA datoteku te učitava ulazni niz. Zatim poziva buildWaveletTree za izgradnju stabla te na kraju poziva (ovisno o ulaznom parametru) rankOperation ili selectOperation.

## Implementacija u programskom jeziku Java

Implementacija u programskom jeziku Java može biti podijeljena na nekoliko modularnih ključnih cjelina koji u konačnici grade skupnu implementaciju:

1. Sučelja TreeOperations, Rankable, Selectable

Sučelja Rankable i Selectable naslijeđuju sučelje TreeOperations, a ukupno sadrže dvije metode koje su srž postojanja stabla valića i RRR strukture ( i njihove kompozicije):

* Public int rank(char c, int endPos, TreeNode rootNode) – funkcija vraća rank parametra 'c' na poziciji endPos za stablo kojem je korijen zadan s rootNode
* Public int select(char c, int boundary, TreeNode, rootNode) – funkcija vraća select parametra 'c' do pozicije boundary za stablo kojem je korijen zadan s rootNode

Ova sučelja implementiraju svi razredi nad kojima se mogu izvesti ove dvije operacije, a u ovom konkretnom slučaju to su razredi TreeNode, RRRBlock i BooleanArrray.

1. Razredi za opis stabla valića: TreeNode, WaveletTree, WaveletTreeBuilder

Ovi razredi modeliraju stablo valića. Razred TreeNode modelira jedan čvor stabla, a parametriziran je parametrom T koji mora implementirati sučelje TreeOperations. Na taj način se postiže da čvor stabla valića može biti bilo koja struktura nad kojom se mogu izvesti operacije rank i select.

Razred WaveletTreeBuilder služi izgradnji stabla valića. Pored pomoćnih (privatnih) metoda, najvažnije su:

* private TreeNode<BooleanArray> fromString(String string, Map<Character, Boolean> charMap, int min, int max)
* private TreeNode<RRRBlock> fromString(String string, Map<Character, Boolean> charMap, int min, int max)

Obje metode su privatne, pozivaju ih javne metode istog imena koje kao parametar primaju samo ulazni znakovni niz. Razlog ovakve arhitekture jest rekurzivna izgradnja stabla. Gore navedene metode kao parametre primaju ulazni niz, Hash mapu preslikavanja pojedinog karaktera u boolean te gornju i donju granicu izgradnje.

1. Razredi za opis RRR strukture: RRRBlock, RRRLookUpTable

Ovi razredi modeliraju RRR strukture. Razred RRRBlock predstavlja jednu standardnu RRR strukturu sa svim podacima i parametrima. RRR struktura se stvara iz BooleanArray – a kojeg kao parametar prima u konstruktoru razreda. Razred RRRLookUpTable modelira Look – up tablicu za klase i offsete RRR strukture. U ovoj verziji implementacije, svaki čvor stabla valića (tj. svaka RRR struktura koja jest čvor tog stabla) ima vlastitu RRR Look – up tablicu čime se postiže veća granularnost nauštrb memorije. Ovo ponašanje može se relativno lako promijeniti pretvaranjem razreda RRRLookUpTable u singleton. Važnije metode su:

* RRRBlock:
  + Private void init() – inicijalizira RRR strukturu (stvara klase, offsete i superblokove) iz BooleanArray – a
  + Private int getOffset(BooleanArray inputArray, int popCount) – vraća pripadajuću permutaciju za BooleanArray i razred
* RRRLookUpTable:
  + Public void generateTable(int length) – generira Look – up tablicu za zadanu duljinu niza
  + Public static int nextPermutation(int v) – vraća sljedeću binarnu permutaciju za integer zadan parametrom v

1. Razred za opis polja bitova: BooleanArray

Ovaj razred enkapsulira obično polje boolean – a a uveden je radi parametrizacije razreda TreeNode jer u suprotnom ne bi bilo moguće postaviti gornju ogradu na parametar T (T extends TreeOperations). Razred prima polje boolean – a u konstruktoru te stvara lokalnu kopiju tog polja.

1. Razred za učitavanje Fasta datoteka: Fasta

Razred Fasta enkapsulira datoteku po FASTA standardu. Osim field – ova za komentare i očitanja za koje postoje pripadajući getteri i setteri, razred ima metodu:

* Public static Fasta fromFile(String fileName) throws IOException

Koji kao parametar prima putanju do FASTA datoteke, a vraća pripadajući model te iste datoteke.

1. Main razred: Main

Razred Main sadrži ulaznu točku programa (metodu main) koja se primarno koristila za testiranje programa. Osim metode main, razred sadrži nekoliko statičkih metoda za analitiku programa:

* inOrder(TreeNode<RRRBlock> rootNode) – Inorder obilazak stabla valića gdje je čvor RRR struktura
* inOrderBA(TreeNode<BooleanArray> rootNode) – Inorder obilazak stabla valića gdje je čvor BooleanArray

Ove dvije metode Inorder obilaskom stabla daju ukupno memorijsko zauzeće stabla. Glavni program korištenjem statičke metode System.nanoTime() računa trajanje izvršavanja izgradnje stabla, operacije rank i operacije select.

Za pokretanje implementacije potrebni su slijedeći koraci:

* Otvoriti konzolu
* Pozicionirati se u direktorij s implementacijom
* Pozicionirati se u direktorij out/production/WaveletTreeRRR
* Naredba „java hr.fer.bio.project.main.Main FileName (R|S) Character EndBound“ gdje je FileName putanja do FASTA datoteke, (R|S) karakter kojim se određuje zahtjev na operaciju rank ili select, Character karakter za koji se izvodi tražena operacija i EndBound granica izvršavanja tražene operacije.

## Implementacija u programskom jeziku C++

Implementacija u programskom jeziku C++ ostvarena je kroz 4 klase: RRR, RRRBlock, RRRTable, WaveletTree i sučelje WTNode kojeg implementiraju klase WTInternalNode i WTLeafNode.

RRR strukture ostvarena je u klasi RRR. Klasa u konstruktoru prima veličinu ulaznog niza na temelju kojeg računa i pohranjuje veličinu blokova i broj blokova i jednom superbloku. Nakon toga blokovi u RRR strukturu se dodaju metodom AddBlock koja kao parametre prima blok i veličinu bloka. Dodavanjem novog bloka u klasi se pamti veličina zadnjeg bloka jer zadnji blok može biti i manji od izračunate veličine. Blokovi se predaju kao uint16\_t tip podataka. Prilikom dodavanja određuje se njegov razred (broj jedinica u bloku) i offset u tablici permutacija, te se za svaki blok pohranjuju te vrijednosti. Klasa RRR ima metode Access koja kao parametar prima indeks i vraća vrijednost i-tog bita u bitmapu kodiranog RRR strukturom, te metode Rank0, Rank1, Select0 i Select1 za operacije Rank i Select.

RRRTable je klasa koja implementira tablicu svih 16-bitnih permutacija za sve moguće kombinacije nula i jedinica. Klasa je ostvarena kao dvodimenzionalno polje vektora pri čemu je prvi parametar razred bloka, a drugi blok offset. RRRTable je ostvarena pomoću Singleton obrasca tako da u memoriji postoji samo jedna instanca tablice koju koriste sve RRR strukture u programu. Klasa sadrži i brojač koliko objekata koristi tablicu, te se stvara kada ju koristi barem jedan objekt, a briše se iz memorije kada brojač ima vrijednost nula. RRRTable ima metode GetBlock za dohvaćanje bloka iz tablice i GetOffset za određivanje offseta bloka u tablici.

RRRBlock je klasa koja implementira jedan RRR blok. Klasa pohranjuje samo dvije vrijednosti: razred i offset. Ima i dvije metode: GetClass i GetOffset za dohvaćanje koje vraćaju razred i offset bloka.

Wavelet stablo ostvareno je pomoću razreda WaveletTree. Klasa ima dvije static metode za stvaranje stabla iz ulaznog stringa i iz FASTA datoteke. Ima metodu Rank koja kao ulazne argumente prima znak i index, te vraća broj pojavljivanja danog znaka od početka stringa do zadanog indeksa uključivo. Metoda select također prima znak i broj, te vraća indeks n-tog pojavljivanja zadanog znaka gdje je n zadan kao drugi parametar. Ukoliko se stablo stvara iz FASTA datoteke, tada se u klasi pohranjuje i opis niza.

Klase WTInternalNode i WTLeafNode ostvaruju čvorove wavelet stabla. WTInternalNode se koristi za unutarnje čvorove. On sadrži RRR niz i dva pokazivača na lijevo i desno dijete. Ima metode GetChaRank, GetCharSelect i Access pomoću kojih su ostvarene operacije Rank, Select i Access.

Klasa WTLeafNode sadrži samo jedan znak abecede. Pozivom metode Access vraća se pohranjeni znak.

## Implementacija u programskom jeziku C# (v1)

Implementacija u programskom jeziku C# sastoji se od 3 klase, RRRStruct, Tree i Program.

U klasi Program učitava se ulazni niz iz datoteke te se on parsira kako bi se maknuli nepotrebni znakovi i retci (komentari i oznake za novi red). Nakon toga kreira se instanca klase Tree i parsiraju argumenti iz komandne linije kako bi se mogla pozvati željena operacija.

Klasa program prima niz znakova (string) kao ulaz i iz njega stvara binarni niz (predstavljen listom bool - ova) iz koje se kasnije izrađuje rrr-struktura (RRRStruct). Osim toga, vrste ulaznih znakova se dijele u dvije skupine koristeći njihov abecedni redoslijed. Iz obje skupine znakova koristeći ulazni niz, tvore se novi nizovi koji će poslužiti za stvaranje djece (također RRRStruct objekti) trenutnog čvora. Dakle unutar konstruktora klase RRRStruct rekurzivno se poziva isti konstruktor, ali na podnizu ulaznog niza. Time se dobiva konačna struktura WaveletTree.

Osim pomoćnih metoda, ova klasi sadrži i metode za izračunavanje select i rank operacija nad nizovima znakova. Obje se baziraju na rekurziji, pri čemu pozivaju metode klase RRRStruct kao što je opisano u uvodu. Za ulazne nizove kraće od 4 znaka implementirane su i metode CalculateRankDummy i CalculateSelectDummy jer nema smisla graditi rrr-strukturu za tako kratke nizove.

Klasa RRRStruct služi za stvaranja rrr-strukture i računanje rank i select operacija nad binarnim nizovima. Konstruktor klase prima kao ulaz niz bitova (predstavljen listom bool-ova) te stvara samu strukturu. U stvaranje spada računanje duljina blokova, superblokova, računanje permutacijskih tablica i stvaranje komprimiranog zapisa rrr-strukture. Stvaranje komprimirane rrr-strukture vrši se unutar funkcije PopulateBlocksInfoAndSuperBLocksInfo. Unutar nje se prolazi po svim blokovima ulaznog niza te se za svaki od njih računa komprimirani zapis koristeći permutacijske tablice.

Same permutacijske tablice ostvarene su kao tip Dictionary. Njegov ključ su sve moguće rank vrijednosti za izračunatu duljinu bloka, a vrijednosti su liste koje sadrže permutacije (uz permutacije spemaju se i kumulativne sume samih permutacija tako da tablice odgovaraju strukturi opisanoj u uvodu). Same permutacije stvaraju se rekurzivnom metodom GeneratePermutations.

Važnije metode unutar ove klase su one za izračun rank i slect operacija.

Metoda CalculateRank kao ulazni parametar prima traženu poziciju, a vraća broj jedinica (true) do te pozicije uključivo. Pri tome koristi se prethodno generirana rrr-struktura. Prvo se pomoću vrijednosti iz superblokova dobiva rezultat koji je približno jednak konačnom rezultatu. Zatim se iterira po blokovima i spremaju se njihove vrijednosti sve dok se ne naiđe na blok koji sadrži traženu poziciju. Tada se još dodaje vrijednost iz tog konačnog bloka. Sve tri vrijednosti zbrojene daju konačan rezultat. Ova metoda može se koristiti i za izračun rank operacije kada se traži broj nula do neke pozicije, tako da se izračuna rank za traženu poziciju, i rezultat oduzme od same pozicije.

Za računanje select operacija postoje dvije metode, CalculateSelect i CalculateSelectZero. Jedna služi za računanje select operacije nad jedinicama, a druga za računanje select operacija nad nulama. Razlog je taj što ne postoji pravilnost kao u slučaju rank operacije.

Obje funkcioniraju slično metodi CalculateRank, dakle obje prolaze po rrr-strukturi (po superblokovima i blokovima), samo je djelomično izmjenjena logika čitanja navedenih podataka.

## Implementacija u programskom jeziku C# (v2)

Implementacija stabla valića čiji čvorovi pohranjuju RRR podatkovnu strukturu u jeziku C# ostvareno je pomoću ukupno 4 razreda: WaveletTreeRRR, WaveletNode, RRRDataStructure i RRRLookupTable.

Razredi RRRDataStructure i RRRLookupTable referenciraju se iz klase WaveletNode kako bi za svaki čvor stabla valića bilo moguće pohraniti podatke potrebne za ostvarenje RRR podatkovne strukture.

Uz navedeno, razred WaveletNode koji predstavlja čvor stabla valića sadrži i reference na svoj roditeljski čvor, na svoje lijevo i desno dijete te znakovni niz u koji se pohranjuje njegova bitmapa dobivena podijelom abecede ulaznih znakova na one koje se kodiraju sa '0' i one koji se kodiraju s '1'.

Razred RRRLookupTable sadrži veličinu bloka te veličinu superbloka kako bi iteriranje kroz RRR bitmapu pohranjenu u RRR strukturi bilo čim brže i jednostavnije. RRRLookupTable sadrži i rječnik koji služi kao tablica za spremanje klasa i pripadajućih permutacija za pojedinu klasu te potreban broj bitova potreban za kodiranje klase u RRR bitmapu. Unutar navedenog razreda nalazi se i metoda za izgradnju navedenog riječnika. Za izradu rječnika dovoljna je samo veličina bloka.

Razred RRRDataStructure pohranjuje RRR bitmapu za čvor iz kojeg je instancirana te 2 niza: jedan niz u koji se spremaju sume superblokova i jedan niz u koji se spremaju pozicije koje ukazuju na mjesto u RRR bitmapi na kojem počinje sljedeći superblok. Ovi nizovi služe za brz dohvat broja nula ili jedinica do pojedinog mjesta u RRR bitmapi i za brzo pozicioniranje unutar RRR bitmape.

Razred WaveletTreeRRR sadrži ulazni niz, abecedu ulaznog niza i referencu na korijen stabla valića iz kojeg možemo doći do svih ostalih čvorova. Razred sadrži i metode za čitanje ulaznog niza, izvlačenje abecede iz ulaznog niza, gradnju stabla valića (u kojoj se poziva metoda razreda WaveletNode za izgradnju opisanog rječnika koji zatim služi za izgradnju RRR podatkovne strukture čvora).

Ostale metode koje sadrži razred WaveletTreeRRR su dvije metode rank(c,i) opisane u prijašnim poglavljima od kojih jedna koristi RRR podatkovne strukture, a druga ne te dvije metode za izračun select(c,i) operacije od kojih, također, jedna koristi RRR podatkovne strukture, a druga ne.

Korisnost korištenja RRR strukture vidljiva je u znatno bržem izvođenju select(c,i) i rank(c,i) operacija za velike ulazne nizove.

# Analiza rezultata

Rezultati performansi implementacije zadanih struktura dani su po tablicama (1-4).

U prvoj tablici dana su memorijska zauzeća i vrijeme potrebno za izgradnju stabla valića bez RRR strukture. Neki rezultati nedostaju, jer to nije bio primarni cilj projekta, ova tablica više služi kao referentna, kako bi vidjeli koliko povećanje vremena i memorije uzrokuje RRR struktura za pojedine implementacije.

U drugoj tablici dana su memorijska zauzeća i vremena potrebna za izgradnju stabla valića s RRR strukturom umjesto bit-vektora.

Treća tablica (slično kao i prva) koristi se samo za usporedbu s četvrtom. U njoj su dana vremena izvođenja rank i select operacija nad stablom valića bez RRR strukture.

U četvrtoj tablici su dana vremena izvođenja rank i select operacija s RRR strukturom.

Upute za testove dane su u datoteci „Readme\_tests.txt“

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ulazna datoteka,**  **Broj znakova,**  **Veličina abecede** | **C** | | **C++** | | **C# - v1** | | **C# - v2** | | **Java** | | **Python** | |
| Input.fa,  n = 502,  Σ = 4 | 46 us | 143 B | \* | \* | \*\* | 1kB | \*\* | 1 kB | 32.7 ms | 1 kB |  |  |
| Input1.fa,  n = 4 570 937,  Σ = 4 | 350 ms | 1.09 MB | \* | \* | \*\* | 8.72 MB | \*\* | 8.71 MB | 2.2 s | 8.71 MB |  |  |
| Input2.fas,  n = 5 524 970,  Σ = 16 | 673 ms | 2.47 MB | \* | \* | \*\* | 21.08 MB | \*\* | 21.07 MB | 4.3 s | 19.77 MB |  |  |

Tablica 1. Usporedba vremena izgradnje stabla valića s binarnim vektorima kao čvorovima

\**Napravljena je samo implementacija stabla valića sa RRR strukturom. Dakle projektni zadatak je napravljen, ovo je samo dodatna tablica koja je dana kako bi se moglo usporediti implementaciju stabla valića s i bez RRR strukture*

\*\**Napravljene su obje implementacije ali se stablo gradi i s bit-vektorom i s RRR strukturom pa je moguće procjeniti zauzeće memorije, ali ne i vrijeme izgradnje bez RRR strukture*

Iz tablice 1. za dostupne rezultate vidljivo je da vezano uz memorijsko zauzeće uvjerljivo najmanje memorije zauzima implementacija u programskom jeziku C, a znatno više implementacije u višim programskim jezicima C# i Java. Razlog tomu jest optimalno korištenje memorije na razini bita u C – u, dok u višim programskim jezicima to nije moguće. Primjerice, tip podataka boolean u C – u može se svesti na jedan jedini bit, dok u višim programskim jezicima taj tip podataka redovito zauzima jedan cijeli bajt (oktet).

S druge strane, za dostupne rezultate, izgradnja stabla valića s binarnim vektorima kao čvorovima redovito je za jedan red veličine brža u programskom jeziku C od one u programskom jeziku Java. Razlog toga gotovo sigurno je nativno izvođenje programa pisanog u C – u, odnosno izvođenje u virtualnom stroju programa pisanog u Javi.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ulazna datoteka,**  **Broj znakova,**  **Veličina abecede** | **C** | | **C++** | | **C# - v1** | | **C# - v2** | | **Java** | | **Python** | |
| Input.fa,  n = 502,  Σ = 4 | 125 us | 368 B | 4.2 ms | 128.8 kB | 39 ms | 1.31 kB | 17.6 ms | 1.75 kB | 63.2 ms | 3.52 kB |  |  |
| Input1.fa,  n = 4 570 937,  Σ = 4 | 1.5 s | 1.33 MB | 1.9 s | 2.62 MB | 34 s | 9.95 MB | 8.5 s | 10.3 MB | 7.5 s | 15.55 MB |  |  |
| Input2.fas,  n = 5 524 970,  Σ = 16 | 2.4 s | 2.19 MB | 2.2 s | 6.31 MB | 43 s | 17.04 MB | 14.5 s | 17.95 MB | 11 s | 35.21 MB |  |  |

Tablica 2. Usporedba vremena izgradnje stabla valića s RRR strukturama kao čvorovima

Iz tablice 2. vidljivo je da implementacije u nižim programskim jezicima (C, C++) redovito za jedan red veličine brža od implementacije u višim programskim jezicima. Također, vidljivo je da je implementacija u C – u brža od one u C++ - u što se može objasniti razinom jezika i apstrakcije. Implementacije C# - v2 i Java su sumjerljive u brzini izgradnje stabla, dok je implementacija C# - v1 znatno sporija što se vjerojatno može objasniti korištenjem određenih (sporijih) struktura i načinom implementacije.

Isto kao i u tablici 1., iz ove je tablice vidljivo da implementacije u višim programskim jezicima zauzimaju jedan red veličine više memorije, dok su memorijska zauzeća u C – u i C++ - u sumjerljiva.

Također, iz komparacije tablice 1. i tablice 2. vidljivo je kako izgradnja stabla valića s RRR strukturama kao čvorovima redovito traje i više nego duplo dulje u odnosu na implementaciju bez RRR strukture neovisno o korištenom programskom jeziku i platformi. Vezano uz memorijsko zauzeće, jedino implementacija u Javi zauzima više memorije, vjerojatno zbog virtualnog stroja. Ostale implementacije zauzimaju manje što se objašnjava implicitnom kompresijom kroz izgradnju RRR strukture.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ulazna datoteka,**  **Broj znakova,**  **Veličina abecede** | **Testovi** | **Referentni** | **C** | **C++** | **C# - v1** | **C# - v2** | **Java** | **Python** |
| Input.fa,  n = 502,  Σ = 4 | Rank('A', 234) = 58 | 2 us | 6 us | \* | \* | 7.6 ms | 85 us |  |
| Select('C', 23) = 107 | 1 us | 4 us | \* | \* | 1.3 ms | 67 us |  |
| Rank('T', 432) = 114 | 4 us | 9 us | \* | \* | 12.3 ms | 65 us |  |
| Select('G', 100) = 389 | 4 us | 11 us | \* | \* | 1 ms | 84 us |  |
| Input1.fa,  n = 4 570 937,  Σ = 4 | Rank('G', 12345) = 3383 | 106 us | 248 us | \* | \* | 1.05 s | 859 us |  |
| Select('A', 12345) = 52553 | 411 us | 1.3 ms | \* | \* | 1.8 ms | 16.5 ms |  |
| Rank('C', 1234567) = 305484 | 12.2 ms | 33.4 ms | \* | \* | \*\* | 47.9 ms |  |
| Select('T', 1000000) = 4053864 | 33.7 ms | 110.1 ms | \* | \* | 21.1 ms | 115.5 ms |  |
| Input2.fas,  n = 5 524 970,  Σ = 16 | Rank ('T', 4000000) = 990024 | 32.3 ms | 83.4 ms | \* | \* | \*\* | 75.3 ms |  |
| Select('A', 40000) = 167264 | 1.3 ms | 6.4 ms | \* | \* | 4.3 ms | 29 ms |  |
| Rank ('A', 543210) = 132287 | 4.3 ms | 16 ms | \* | \* | \*\* | 56.4 ms |  |
| Select('C', 543210) = 2226555 | 18.5 ms | 111.7 ms | \* | \* | 47.31 ms | 119.7 ms |  |

Tablica 3. Usporedba vremena operacija „Rank“ i „Select“ gdje je čvor stabla binarni vektor

\**Napravljena je samo implementacija stabla valića sa RRR strukturom. Dakle projektni zadatak je napravljen, ovo je samo dodatna tablica koja je dana kako bi se moglo usporediti implementaciju stabla valića s i bez RRR strukture*

\*\* *Izvršavanje je trajalo predugo (više od 10 s)*

Osnovni zaključak tablice 3. jest da samo izgradnjom stabla valića za navedene testne primjere nismo postigli nikakvu vremensku uštedu, što je vidljivo iz usporedbe bilo koje implementacije s referentnim programom. Asimptotska složenost referentnog programa je O(n), dok se asimptotska složenost rank i select operacije nad binarnim stablom valića svodi na O(n \* log(a)) gdje je a veličina abecede ulaznog niza.

Za rezultate vrijede već uvriježeni zaključci, niži programski jezici brži su od viših programskih jezika, dok se implementacija C# - v2 pokazala za jedan red veličine brža od implementacije u Javi.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ulazna datoteka,**  **Broj znakova,**  **Veličina abecede** | **Testovi** | **Referentni** | **C** | **C++** | **C# - v1** | **C# - v2** | **Java** | **Python** |
| Input.fa,  n = 502,  Σ = 4 | Rank('A', 234) = 58 | 2 us | 2 us | 4 us | 1 ms | 3.3 ms | 102 us |  |
| Select('C', 23) = 107 | 1 us | 2 us | 6 us | 2 ms | 9.9 ms | 149 us |  |
| Rank('T', 432) = 114 | 4 us | 1 us | 4 us | 1 ms | 6.5 ms | 102 us |  |
| Select('G', 100) = 389 | 4 us | 4 us | 5 us | 3 ms | 10.3 ms | 134 us |  |
| Input1.fa,  n = 4 570 937,  Σ = 4 | Rank('G', 12345) = 3383 | 106 us | 3 us | 12 us | 42 ms | 2.3 ms | 180 us |  |
| Select('A', 12345) = 52553 | 411 us | 161 us | 12 us | 49 ms | 3 ms | 482 us |  |
| Rank('C', 1234567) = 305484 | 12.2 ms | 5 us | 9 us | 42 ms | 2.7 ms | 21.3 ms |  |
| Select('T', 1000000) = 4053864 | 33.7 ms | 17 ms | 12 us | 69 ms | 9.75 ms | 22.6 ms |  |
| Input2.fas,  n = 5 524 970,  Σ = 16 | Rank ('t', 4000000) = 990024 | 32.3 ms | 6 us | 12 us | 72 ms | 2.8 ms | 21.4 ms |  |
| Select('a', 40000) = 167264 | 1.3 ms | 931 us | 22 us | 94 ms | 4 ms | 1.5 ms |  |
| Rank ('a', 543210) = 132287 | 4.3 ms | 4 us | 11 us | 83 ms | 9.6 ms | 3.3 ms |  |
| Select('c', 543210) = 2226555 | 18.5 ms | 14.7 ms | 19 us | 113 ms | 3.5 ms | 31.6 ms |  |

Tablica 4. Usporedba vremena operacija „Rank“ i „Select“ gdje je čvor stabla RRR struktura

Napomena: rezultati rank i select operacija mogu se razlikovati +-1 ovisno o implementaciji (za rank ako se ide [0, i> ili [0, i] , a za select ako se vraća pozicija elementa gledana od 0 ili od 1

Iz tablice 4. vidljivo je radikalno smanjenje rank i select operacija u odnosu na implementacije bez RRR strukture bez obzira na programski jezik i implementaciju. Ta razlika dolazi do izražaja za veće ulazne nizove i veće abecede. Kod manjih nizova (prva testna datoteka) uštede nema, čak štoviše, u nekim slučajevima korištenje RRR struktua usporava rank i select operacije.

Kod korištenja RRR strukture implementacije u nižim programskim jezicima su na skali mikrosekunde, dok su implementacije u višim programskim jezicima također sumjerljive, ali na razini milisekunde.

Također zanimljiv je rezultat gdje je implementacija C# - v2 za select implementaciju brža čak i od implementacije u C – u.

# Zaključak

Na temelju analize rezultata možemo ustvrditi kako za velike ulazne nizove (reda veličine 106 i više) ostvarujemo značajnu uštedu u računanju rank i select operacija korištenjem stabla valića s RRR strukturom. Izgradnjom stabla valića možemo komprimirati ulazni niz, no njime ne postižemo ubrzanje operacija kako je pokazano u Tablici 3. Dodavanjem RRR strukture u općenitom slučaju povećavamo zauzeće memorije zbog dodatnih struktura, no za velike ulaze to nisu drastične promjene (moguće je i ostvariti uštedu uz dobru kompresiju bit-vektora RRR strukturom). Također se može postaviti pitanje o potrebi korištenja RRR „lookup“ tablica u svakom čvoru – za male abecede postojat će korijen i dva čvora-dijeteta koji će biti sličnog reda veličine te će se u tom slučaju generirati 3 identične tablice što dovodi do memorijskih gubitaka, a ne pruža uštedu u vremenu izvođenja. Iz Tablice 4 možemo vidjeti kako rank i select operacje daju puno bolje rezultate od istih operacija izvedenih nad običnim bit-vektorom. Također valja napomenuti kako implementacija u C++-u daje najbolje rezultate za select što je posljedica optimalne izvedbe tog algoritma – binarno pretraživanje, umjesto iteriranja po superblokovima.

Što se tiče osobnog iskustva izrade projekta zaključak je da je stablo valića i RRR strukturu jednostavno za shvatiti no prilično problematično za implementirati. Razlog tomu je neintuitivna RRR struktura, s varijablinim offset-ima i tablicom koju je teško čitati za veće veličine blokova. To sve onemogućava jednostavan „debug“ i analizu rješenja.

# Literatura

[1] http://alexbowe.com/wavelettrees/

[2] http://www.dcc.uchile.cl/~gnavarro/ps/cpm12.pdf

[3] Alexander Bowe, 2010. Multiary Wavelet Trees in Practice (Honours Thesis),

[4] <http://alexbowe.com/rrr/>

[5] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms, 3(4), 2007.

[6] [http://compbio.fmph.uniba.sk/vyuka/vvt/poznamky/p16.pdf](https://mail5.fer.hr/OWA/redir.aspx?C=40zprYcKxkKmEthgsa3vts0Wb_SIA9IIYmxKu31iABpRL0IJOk9shP_UZEo9jz7DQyR7g8m3GLk.&URL=http%3a%2f%2fcompbio.fmph.uniba.sk%2fvyuka%2fvvt%2fpoznamky%2fp16.pdf)