# 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(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).

RRR struktura koristi se za sažimanje bit-vektora te svodi složenost rank operacije 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.

# 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).

# Implementacije

## Implementacija u programskom jeziku C

**Vremena izvođenja**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ulazna datoteka, veličina u oktetima, veličina abecede** | **Testovi** | **Vrijeme izgradnje stabla valića + RRR** | **Vremena izvođenja rank operacije (prebrojavanje u nizu)** | **Vremena izvođenja rank operacije (nad stablom valića + RRR)** |
| Input.txt, n = 138, Σ = 16 | Rank ('t', 100) = 9 | ~ 1 sekunda | 1 us | 4 us |
| Rank('e', 138) = 10 | 1 us | 4 us |
| Rank('w', 20) = 2 | < 1 us | 5 us |
| Input.fa, n = 502,  Σ = 4 | Rank('A', 234)= 58 | ~130 us | 2 us | 1 us |
| Rank('C', 432)= 92 | 3 us | 1 us |
| Rank('T', 432)= 114 | 3 us | 1 us |
| Input1.fa,  n = 4 570 937,  Σ = 4 | Rank('G', 12345) = 3383 | ~1.4 sekunde | 103 us | 5 us |
| Rank('A', 123456) = 30103 | 1 012 us | 4 us |
| Rank('C', 1234567) = 305484 | 9 742 us | 5 us |
| Input2.fas,  n = 5 524 970,  Σ = 4 | Rank ('t', 4000000) = 990024 | ~2 sekunde | 45 928 us | 9 us |
| Rank ('g', 5500000) = 1384977 | 57 467 us | 9 us |
| Rank ('a', 543210) = 132287 | 4 642 us | 5 us |

Tablica 1. – vremena izvođenja implementacije u C-u.

Kao što je vidljivo iz tablice, vrijeme izgradnje stabla valića i RRR strukture ovisi o duljini ulaznog niza, ali i o veličini abecede ulaznog niza. To je najočitije u prvom ulaznom nizu koji je relativno kratak (138 znakova) no za njegovu izgradnju potrebna je otprilike 1 sekunda. Također, za izvođenje rank operacije nad tim stablom trošimo jednako vremena kao i u trećem primjeru gdje imamo mnogo veći ulazni niz. Razlika je što će se za ulaze FASTA datoteka izgraditi stablo s korijenskim čvorom i samo 2 čvora djeteta (2 razine), dok će se za prvu datoteku izgraditi stablo s 4 razine, ukupno 15 čvorova.

Usporedbom vremena izvođenja u prvom retku, vidimo da će se brže izvesti „primitivna“ operacija brojanja znakova složenosti O(n).

Za FASTA datoteke vidimo da ovisno o parametru granice operacije rank, vrijeme izvođenja „primitivne operacije“ proporcionalno raste. Izvođenje operacije rank nad stablom valića ovisiti o veličini blokova i superblokova, ali i o tome gdje će granica „upasti“ (npr. ako je granica točno na početku nekog superbloka, rezultat je ujedno i spremljena vrijednost sume tog superbloka).

**Utrošak memorije**

Utrošak memorije računan je pomoću funkcije te su u obzir uzete veličina RRR bitmape, veličina pohranjenih suma i odmaka superblokova (2 polja cjelih brojeva) te veličina RRR „lookup“ tablice(1 bajt za oznaku klase i 2 bajta za odmak - blok).

Kao ulazi korištene su gore navedene datoteke.

|  |  |  |
| --- | --- | --- |
| **Ulazna datoteka** | Veličina ulazne datoteke (u oktetima) | Utrošak memorije (u oktetima) |
| input.txt | 138 | 877 |
| input.fa | 502 | 630 |
| input1.fa | 4 570 937 | 1 652 140 |
| input2.fas | 5 524 970 | 2 684 831 |

Kao što vidimo iz tablice u prva dva slučaja nemamo nikakvu uštedu memorije. Uzrok tomu su ponajviše RRR strukture (bitmapa,polje za superblokove, tablica) koje su prisutne u svakom čvoru.

U zadnja 2 slučaja postoji ušteda. Kao i kod vremena izvođenja bitno je primjetiti da FASTA ulazi generiraju samo 3 čvora, što značajno doprinosi uštedi memorije.