



VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS
FUNDAMENTINIŲ MOKSLŲ FAKULTETAS
INFORMACINIŲ TECHNOLOGIJŲ KATEDRA

Artūras Novičenko

**INTERVALINIŲ MEDŽIO TIPO DUOMENŲ STRUKTŪRŲ ĮTAKA,
SPRENDŽIANT DIDELIŲ DUOMENŲ APDOROJIMO UŽDAVINIUS
IMPACT OF INTERVAL TREE TYPE DATA STRUCTURES ON BIG DATA
PROCESSING TASKS**

Baigiamasis magistro darbas

Informacinių technologijų studijų programa, valstybinis kodas 6211BX016

Duomenų gavybos technologijos specializacija

Informatikos inžinerijos studijų kryptis

Vilnius, 2020

Vilniaus Gedimino technikos universitetas
Fundamentinių mokslų fakultetas
Informacinių technologijų katedra

ISBN
Egz. sk.
Data-.....-.....

Antrosios pakopos studijų **Informacinių technologijų** programos magistro baigiamasis darbas 4

Pavadinimas **Intervalinių medžio tipo duomenų struktūrų įtaka, sprendžiant didelių duomenų apdorojimo uždavinius**

Autorius **Artūras Novičenko**

Vadovas **Dmitrij Šešok**

Kalba: lietuvių

Anotacija

Baigiamojo darbo tikslas – iširti intervalinių medžio tipo duomenų struktūrų įtaką didelių duomenų apdorojimo uždaviniams spręsti.

Baigiamajame darbe yra detaliai apžvelgiamos segmentinio medžio ir dvejetainio indeksuoto medžio intervalinės duomenų struktūros. Pagal gautą duomenų struktūrų analizę yra sumodeliuojamos sumos užklausų ir sumos užklausų matricose užduotys. Šios užduotys yra realizuojamos ir išbandomos su dideliais duomenų kiekiais. Pagal atliktą analizę yra išskiriami greičio, atminties panaudojumo, universalumo ir kodo sudėtingumo požymiai. Vadovaujantis šiais požymiais yra ištiriamos, įvertinamos ir palyginamos pasirinktos duomenų struktūros.

Darbo apimtis 55p. teksto be priedų, 38 pav., 6 lent., 20 bibliografinių šaltinių.

Prasminiai žodžiai: intervalinė duomenų struktūra, dideli duomenys, segmentis medis, dvejetainis indeksuotas medis

Vilnius Gediminas Technical University
Faculty of Fundamental Sciences
Department of Information Technologies

ISBN ISSN
Copies No.
Date-.....-.....

Master Degree Studies **Information Technologies** study programme Master Graduation Thesis

Title **Impact of Interval Tree Type Data Structures on Big Data Processing Tasks**

Author **Artūras Novičenko**

Academic supervisor **Dmitrij Šešok**

Thesis language: Lithuanian

Annotation

The main thesis objective is to explore the impact of interval tree type data structures on big data processing tasks.

This thesis provides a full overview of segment tree and binary indexed tree data structures. According to the data structure overview interval sum query and interval sum query inside matrices problem statements are formulated. These problems are then implemented and evaluated using big data. Using the analysis of speed, memory, universality and code complexity attributes are marked out. Using these attributes the data structures are explored, evaluated and compared.

The thesis consists of: 55p. text without appendixes, 38 pictures, 6 tables, 20 bibliographical entries.

Keywords: interval data structure, big data, segment tree, binary indexed tree

Vilniaus Gedimino technikos universiteto
egzaminų sesijų ir baigiamųjų darbų
rengimo bei gynimo organizavimo tvarkos
aprašo
2 priedas

(Baigiamojo darbo sąžiningumo deklaracijos forma)

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

Artūras Novičenko, 20145745

(Studento vardas ir pavardė, studento pažymėjimo Nr.)

Fundamentinių mokslų fakultetas

(Fakultetas)

Informacinės technologijos, DGTfm-18

(Studijų programa, akademinė grupė)

BAIGIAMOJO DARBO (PROJEKTO)

SĄŽININGUMO DEKLARACIJA

2020 m. gegužės 14 d.

Patvirtinu, kad mano baigiamasis darbas tema „Intervalinių medžio tipo duomenų struktūrų įtaka, sprendžiant didelių duomenų apdorojimo uždavinius“ patvirtintas 2020 m. balandžio 09 d. dekanų potvarkiu Nr. 113fm, yra savarankiškai parašytas. Šiame darbe pateikta medžiaga nėra plagijuota. Tiesiogiai ar netiesiogiai panaudotos kitų šaltinių citatos pažymėtos literatūros nuorodose.

Prenkant ir įvertinant medžiagą bei rengiant baigiamąjį darbą, mane konsultavo mokslininkai ir specialistai: docentas daktaras Dmitrij Šešok. Mano darbo vadovas docentas daktaras Dmitrij Šešok.

Kitų asmenų indėlio į parengtą baigiamąjį darbą nėra. Jokių įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs (-usi).

(Parašas)

Artūras Novičenko

(Vardas ir pavardė)

Turinys

ĮVADAS	9
1. Intervalinės medžio tipo duomenų struktūros	10
1.1. Segmentinis medis.....	10
1.1.1. Struktūra	10
1.1.2. Konstravimas.....	11
1.1.3. Intervalinės užklausos	12
1.1.4. Atnaujinimo užklausos	14
1.1.5. Intervalinės atnaujinimo užklausos	15
1.1.6. Dviejų dimensijų segmentinis medis.....	18
1.1.7. Išsaugojimas segmentinio medžio istorijos	23
1.2. Dvejetainis indeksuotas medis	26
1.2.2. Struktūra	26
1.2.3. Konstravimas.....	29
1.2.4. Intervalinės užklausos	30
1.2.5. Atnaujinimo užklausa.....	32
1.2.6. Dviejų dimensijų dvejetainis indeksuotas medis.....	33
2. Problemos leidžiančios įvertinti intervalinių medžio tipo duomenų struktūrų efektyvumą.....	36
3. Įvertinimas skirtingų intervalinių medžio tipo duomenų struktūrų	38
3.1. Duomenų generavimas	39
3.2. Greičio įvertinimas	41
3.2.1. Konstravimo greitis	41
3.2.2. Intervalinės užklausos greitis	46
3.2.3. Atnaujinimo užklausos greitis su intervalinėmis užklausomis.....	49
3.3. Atminties naudojimo įvertinimas	50
3.4. Testavimas	51
IŠVADOS.....	53
Literatūros sąrašas	54

Paveikslukų sąrašas

1.1 pav. Segmentinio medžio struktūros pavizdys (Segment Tree, 2018).....	11
1.2 pav. Segmentinio medžio konstravimo realizacija c++ programavimo kalba	12
1.3 pav. Segmentinio medžio sumos užklausos realizacija c++ programavimo kalba	13
1.4 pav. Segmentinio medžio atnaujinimo užklausos realizacija c++ programavimo kalba	15
1.5 pav. Segmentinio medžio nuleidimo metodo realizacija c++ programavimo kalba.....	16
1.6 pav. Segmentinio medžio intervalinės atnaujinimo užklausos realizacija c++ programavimo kalba	17
1.7 pav. Segmentinio medžio gavimo metodo realizacija c++ programavimo kalba	17
1.8 pav. Konstravimas dviejų dimensijų segmentinio medžio x koordinatės atžvilgiu.....	18
1.9 pav. Konstravimas dviejų dimensijų segmentinio medžio y koordinatės atžvilgiu.....	19
1.10 pav. Pavyzdinė duomenų matrica (Laaksonen, 2017)	20
1.11 pav. Dviejų dimensijų segmentinio medžio vizualizacija (Laaksonen, 2017).....	21
1.12 pav. Dviejų dimensijų segmentinio medžio sumos užklausos realizacija c++ programavimo kalba	22
1.13 pav. Konstravimas segmentinio medžio naudojant rodykles	24
1.14 pav. Segmentinio medžio sumos užklausa naudojant rodykles	24
1.15 pav. Segmentinio medžio viršūnės struktūra	25
1.16 pav. Segmentinio medžio atnaujinimo užklausa naudojant rodykles	25
1.17 pav. Dvejetainis indeksuotas medis parodantis indeksų atsakingumą ir viršūnių sumas (Binary Indexed Trees, 2014)	28
1.18 pav. Pirmoji dvejetainio indeksuoto medžio konstravimo algoritmo realizacija c++ programavimo kalba	29
1.19 pav. Antroji dvejetainio indeksuoto medžio konstravimo algoritmo realizacija c++ programavimo kalba	30
1.20 pav. Dvejetainio indeksuoto medžio konstravimo realizacija c++ programavimo kalba	30
1.21 pav. Dvejetainio indeksuoto medžio sumos užklausos realizacija c++ programavimo kalba ..	32
1.22 pav. Dvejetainio indeksuoto medžio atnaujinimo užklausos realizacija c++ programavimo kalba	33

1.23 pav. Dviejų dimensijų dvejetainio indeksuoto medžio konstravimo realizacija c++ programavimo kalba	33
1.24 pav. Stačiakampio sumos suradimas išsireiškiant per kelias sumas (Belwariar, 2015).....	34
1.25 pav. Dviejų dimensijų dvejetainio indeksuoto medžio sumos užklausos realizacija c++ programavimo kalba	35
1.26 pav. Dviejų dimensijų indeksuoto medžio atnaujinimo užklausos realizacija c++ programavimo kalba	35
3.1 pav. Startinių duomenų generavimas į tekstinę rinkmeną	40
3.2 pav. Sumos užklausų duomenų generavimas.....	41
3.3 pav. Standartinio masyvo konstravimo realizacija c++ programavimo kalba	42
3.4 pav. Matricos konstravimo realizacija c++ programavimo kalba.....	42
3.5 pav. Prefiksinės sumos konstravimo pseudo kodo formulė	43
3.6 pav. Duomenų struktūrų konstravimo greičio diagrama.....	44
3.7 pav. Dviejų dimensijų duomenų struktūrų konstravimogreičio diagrama	45
3.8 pav. Sumos užklausų greičio diagrama esant mažiems nuotoliams tarp intervalo rėžių	47
3.9 pav. Sumos užklausų diagrama esant vidutiniems nuotoliams tarp intervalo rėžių.....	47
3.10 pav. Sumos užklausų diagrama esant alternuojantiems/dideliems nuotoliams tarp intervalo rėžių	48
3.11 pav. Duomenų struktūrų atnaujinimo užklausų greitis	49
3.12 pav. Duomenų struktūrų įvertinimo testavimo realizacija c++ programavimo kalba.....	51

Lentelių sąrašas

1.1 lentelė. Prefiksinių sumų pavyzdys.....	26
1.2 lentelė. Skaičiaus išreiškimas per dvejetų laipsnių sumas	26
1.3 lentelė. Skaičiai užrašyti dešimtainę ir dvejetainę skaičiavimo sistema	27
1.4 lentelė. Dvejetainio indeksuoto medžio duomenų pavyzdys (Binary Indeded Trees, 2018).....	27
1.5 lentelė. Konjunkcijos teisingumo lentelė	31
3.1 lentelė. Galimų užklausų pavyzdžiai.....	40

IVADAS

Pasaulyje duomenų kiekis sparčiai auga, šiuos duomenis darosi sunku suprasti ir apdoroti, kad mes surastumėme mums reikiamą informaciją. Duomenų organizavimas ir išgavimas tapo neatsiejamomis kasdienėmis operacijomis. Viena iš šių operacijų yra informacijos išgavimas pateikus tam tikrą intervalą, kur reikėtų ieškoti duomenų. Norint efektyviai atlikti šią operaciją yra naudojamos įvairios intervalinės medžio tipo duomenų struktūros. Jos duomenis perteikia įvairių sąlygų sąryšiais ir suteikia galimybę juos efektyviau valdyti ir saugoti. Kadangi šiuo metu intervalinių medžio tipo duomenų struktūrų pasirinkimo spektras nėra mažas, yra sunku išsirinkti, kuri duomenų struktūra yra efektyvesnė greičio, atminties, kodo sudėtingumo arba universalumo aspektu.

Norint išsirinkti reikiamą intervalinę medžio tipo duomenų struktūrą reikia atlikti tyrimą, kuris parodytų rezultatus, pagal kuriuos galėtumėme atsirinkti, kuri duomenų struktūra tinkama tam tikram atvejui. Dėl to šiame darbe buvo pasirinktos ir ištirtos kelios intervalinės medžio tipo duomenų struktūros skirtos darbui su dideliais duomenimis.

Darbo tikslas – ištirti intervalinių medžio tipo duomenų struktūrų įtaką didelių duomenų apdorojimo uždaviniams spręsti.

Uždaviniai:

1) Atlikti intervalinio medžio tipo duomenų struktūrų mokslinės literatūros apžvalgą, taikomą dideliems duomenims apdoroti ir sumodeliuoti keletą uždavinių, leidžiančių įvertinti pasirinktų intervalinių medžio tipo duomenų struktūrų efektyvumą.

2) Sukurti kompiuterines programas, sprendžiančias iškeltus uždavinius, naudojant aptartas intervalines medžio tipo duomenų struktūras.

3) Įvertinti skirtingų intervalinių medžio tipo duomenų struktūrų pasirinkimo įtaką uždaviniams spręsti ir pateikti gautus rezultatus.

1. Intervalinės medžio tipo duomenų struktūros

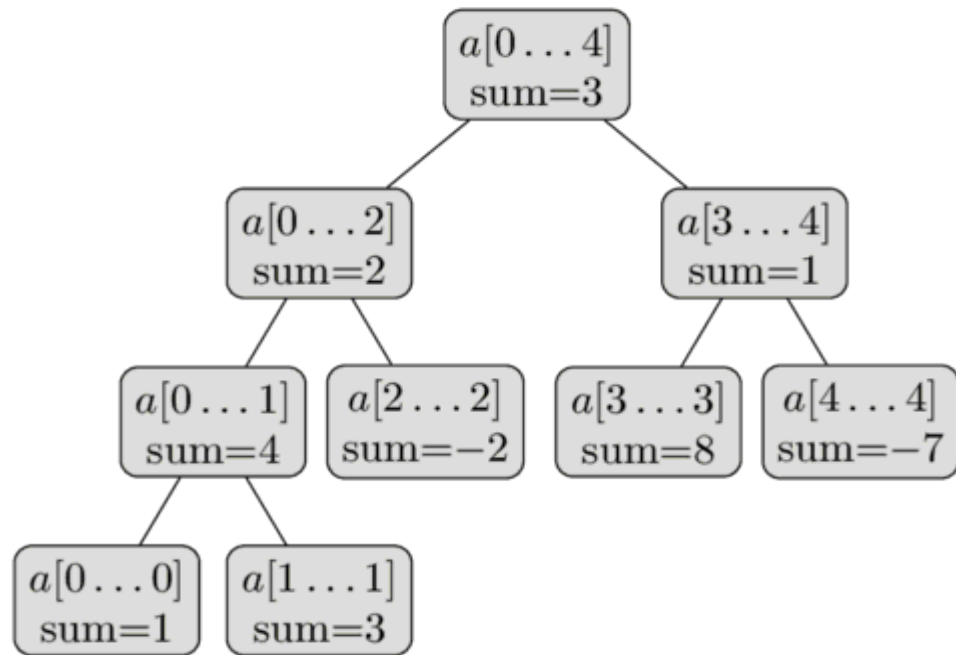
1.1. Segmentinis medis

Viena iš duomenų struktūrų kuri efektyviai padeda susidoroti su intervalinėmis problemomis yra segmentinis medis. Informatikoje ši duomenų struktūra yra taip pat žinoma kaip statistinis medis. Ši hierarchinio tipo duomenų struktūra suteikia galimybę operatyviai atsakyti į intervalines užklausas, pareikšti, kuris intervalas turi ieškomą tašką, bei taip pat palaiko efektyvią atnaujinimo užklausą, kuri atnaujinama duomenų struktūros reikšmę nekeisdama jo struktūros ir kitas operacija aprašomas šiame skyriuje (Berg, Kreveld, Overmars & Schwarzkopf, 2000).

1.1.1. Struktūra

Segmentinio medžio struktūra yra ganėtinai sudėtinga, ją geriausiai išsina suprasti naudojantis pavyzdžiais. Kaip pavyzdį galima būtų paminėti toki atvejį, kai egzistuoja sveikųjų skaičių masyvas¹ $\alpha[0..n-1]$, kur n reiškia masyvo ilgį. Pirmiausia siekiant apskaičiuoti visų masyvo elementų sumą, tai būtų $[0..n-1]$ segmento suma. Norint apskaičiuoti kitus masyvo segmentus šį masyvo segmentą reikia dalinti į dvi dalis: $[0..\frac{n}{2}]$ ir $[\frac{n}{2}+1 \dots n-1]$ ir skaičiuoti gautųjų segmentų sumas. Šias dvi dalis toliau reikia atskirai dalinti į dar dvi dalis ir skaičiuoti gautas naujųjų dar mažesnių segmentų sumas. Šį procesą reikia kartoti tol, kol visų dalinamųjų segmentų dydžiai netampa vieno elemento dydžio. Taigi, mūsų pradinį segmentą $[0..n-1]$ reikia dalinti pusiau(jeigu tai nėra segmentas kuriam priklauso vienas elementas) ir kviesti tą pačią dalinimo procedūrą abiem naujai gautiems segmentams, saugant apskaičiuojamas sumas. Šie segmentai suformuoja dvejetainį medį: dvejetainio medžio šaknis yra segmentas $\alpha[0..n-1]$, ir kiekviena viršūnė (išskyrus viršūnės lapus) turi dvi viršūnes kurie yra jos vaikai. Būtent dėl to ši duomenų struktūra ir yra vadinama segmentiniu medžiu, nors daugumoje realizacijų medis nėra aiškiai sukonstruojamas. Pavyzdžiui, norint sudaryti segmentinį medį $\alpha = [1, 3, -2, 8, 7]$ masyvui reikia 9 viršūnių (žr. 1.1 pav.)

¹ Indeksais sužymėtų vienodo tipo duomenų(objektų) eilė.



1.1 pav. Segmentinio medžio struktūros pavidys (Segment Tree, 2018)

Segmentinis medis visuomet turi tiesiškai priklausomą viršūnių skaičių nuo pradinio masyvo dydžio, taip yra todėl, kadangi pirmasis lygis turi visada vieną viršūnę (medžio šaknį), antrasis lygis turi dvi viršūnes, trečiasis keturias ir taip toliau kol viršūnių skaičius nepasiekia n . Viską apžvelgus galima įvertinti viršūnių skaičių pagal formulę: $1 + 2 + 4 + \dots + 2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 n \rceil + 1} < 4n$. Taip pat svarbu paminėti tai, jog kai masyvo dydis n nėra 2 laipsnis, ne visi segmentinio medžio lygiai yra užpildomi. Segmentinio medžio aukštis yra $\log_2 n$, kadangi leidžiantis į apačią nuo šaknies segmentų dydžiai visuomet mažėja per pusę (Segment Tree [interaktyvus] 2018 [žiūrėta 2019 m. lapkričio 22d.]. Prieiga per internetą: https://cp-algorithms.com/data_structures/segment_tree.html).

1.1.2. Konstravimas

Segmentinį medį efektyviai galima sukonstruoti pradedant nuo apatinio lygio ir kylant į viršų, pradedama konstruoti nuo viršūnių lapų, viršūnė yra vadinama lapu jeigu jos segmento dydis yra lygus vienam. Nuo šių viršūnių galima sukonstruoti vienu lygiu aukštesnį lygį ir kartoti šią konstravimo procedūrą tol, kol yra pasiekiamas pirmasis lygis (šaknies viršūnė). Visas šias operacijas yra patogiu atlikti naudojant rekursinę² funkciją. Norint sukonstruoti visą segmentinį

² Programų sudarymo metodas, kai programa kreipiasi į save, esant mažesnėms argumentų reikšmėms.

```

void build(int medzio_indeksas, int masyvo_kairysis_indeksas, int masyvo_desinysis_indeksas)
{
    if( masyvo_kairysis_indeksas == masyvo_desinysis_indeksas ) {
        //Jeigu kairės ir dešinės indeksai sutampa reiškia segmentui priklauso vienas elementas.
        segmentinis_medis[medzio_indeksas] = a[masyvo_kairysis_indeksas];
    }
    else
    {
        // Paskaičiuojamas dabartinio segmento vidurys.
        int masyvo_vid_indeksas = (masyvo_kairysis_indeksas + masyvo_desinysis_indeksas) / 2;
        // Nustatomas kairiojo segmento indeksas.
        int kairiojo_segmento_indeksas = 2 * medzio_indeksas + 1;
        // Nustatomas dešiniojo segmento indeksas.
        int desiniojo_segmento_indeksas = 2 * medzio_indeksas + 2;
        // Skaičiuojamas kairysis poemdis.
        build(kairiojo_segmento_indeksas, masyvo_kairysis_indeksas, masyvo_vid_indeksas);
        // Skaičiuojamas dešinysis poemdis.
        build(desiniojo_segmento_indeksas, masyvo_vid_indeksas + 1, masyvo_desinysis_indeksas);
        // Pagal paskaičiuotus rezultatus paskaičiuojamas dabartinis segmentas.
        segmentinis_medis[medzio_indeksas] = segmentinis_medis[kairiojo_segmento_indeksas] +
            segmentinis_medis[desiniojo_segmento_indeksas];
    }
}

```

medį reikia $O(n)$ laiko, kadangi viršūnių skaičius nėra didesnis už $4n$. Segmentinį medį galima sukonstruoti lengvai(žr. 1.2 pav.)

1.2 pav. Segmentinio medžio konstravimo realizacija c++ programavimo kalba

Kaip matyti iš programinio kodo, segmentinio medžio segmentai irgi yra saugomi masyve, taip pat galima pastebėti, jog kiekvienas „tėvo“ segmentas medžio saugomoje struktūroje gali pasiekti savo vaikus pagal indeksų skaičiavimo formules: $2 * dabartinis_indeksas + 1$ ir $2 * dabartinis_indeksas + 2$, jeigu būtų naudojama nulinė indeksavimo bazė, tai „vaikus“ būtų galima pasiekti per $2 * dabartinis_indeksas$ su $2 * dabartinis_indeksas + 1$ indeksus (Segment Tree, 2016).

1.1.3. Intervalinės užklausos

Kaip jau buvo minėta struktūros poskyryje, segmentinis medis suteikia galimybę atlikti intervalines užklausas. Norint atlikti šias intervalines užklausas, yra pateikiamas norimas intervalas, dažniausiai tai būna masyvo indeksai, režio pradžia ir režio pabaiga, kuriuose norėtumėme gauti segmento reikšmę. Naudojantis struktūros poskyryje minėtų pavyzdžiu, jog turime masyvą $a[0..n-1]$ ir norėtumėme gauti tam tikrą šio masyvo sumą indeksuose nuo $a[l..r]$ t.y reikėtų gauti sumą nuo masyvo indekso l iki indekso r , kur $l \leq r$, $l \geq 0$ ir $r < n$. Šią sumą galima išreikšti pagal formulę: $a[l] + a[l+1] + a[l+2] + \dots + a[r]$. Įprastai tokiai sumai gauti reikėtų $O(r-l+1)$ laiko, tačiau segmentinis medis suteikia galimybę į šią sumos užklausą atsakyti ilgiausiai per

$O(\log_2 n)$ laiką, o tai labai pagerina vienos užklausos atsako laiką, jeigu užklausa nėra maža. Segmentinis medis yra gerai vertinamas todėl, kad jį sukonstruoti reikia vieną kartą ir jis gali atsakyti į užklausas apie bet kuriuos intervalo režius. Jis yra įprastai naudojamas, kai yra siunčiama daug užklausų į skirtingus intervalus. Užklausos segmentiniuose medžiuose vykdomos tokiu būdu: tarkime, jog šiuo metu mes esame viršūnėje, kuri padengia segmentą $a[tl..tr]$ tai mums suteikia 3 galimybes:

1) Lengviausias atvejis, kai segmentas $a[l..r]$ lygus segmentui $a[tl..tr]$, tada galima baigti ir grąžinti apskaičiuotą sumą, kuri yra saugoma viršūnėje, kuri yra šiuo metu apdorojama.

2) Alternatyviai užklausos segmentas gali visiškai priklausyti kairiajam arba dešiniajam dabartinės viršūnės po medžiui tai yra arba intervalas priklauso $a[tl..tm]$ arba $a[tm + 1..tr]$ kur $tm = \frac{(tl+tr)}{2}$ šiuo atveju reikia eiti į tą vaiko viršūnę kuriam priklauso intervalas ir tikrinti tris suteikiamas galimybes.

3) Trečia galimybė yra tai, jog užklausos segmentas susikerta su abiem dabartinės viršūnės „vaikais“, tada nėra kito išeities tik eiti į abu „vaikus“ ir sujungti jų atsakymo užklausas. Pirmiausia reikia eiti į kairįjį „vaiką“, $a[tl..tm]$ ir ten pasirinkti vieną iš trijų galimybių, o po to tą patį padaryti su dešiniuoju „vaiku“ $a[tm + 1..tr]$.

Taigi, norint įvykdyti sumos užklausą būtini rekursyvūs kvietimai „į save“ vieną kartą į kairįjį arba dešinįjį po medį arba į juos abu. Rekursija pasibaigia tada, kai dabartinės viršūnės

```
int sum(int segmentinio_medzio_indeksas, int tl, int tr, int l, int r)
{
    if( l > r ) { // Patikriname ar validus segmentas.
        return 0;
    }

    if(l == tl && r == tr ) {
        // Jeigu viršūnės režiai lygus mūsų ieškomiems grąžiname suma.
        return segmentinis_medis[segmentinio_medzio_indeksas];
    }

    int tm = (tl + tr) / 2; // Daliname segmentą į dvi dalis.
    // Apskaičiuojame kairiojo pomedžio sumą.
    int sum_kairysis_pomedis = sum(segmentinio_medzio_indeksas * 2, tl, tm, l, min(r, tm));
    // Apskaičiuojame dešiniojo pomedžio sumą.
    int sum_desinysis_pomedis = sum(segmentinio_medzio_indeksas * 2 + 1, tm + 1, tr, max(l, tm + 1), r);
    return sum_kairysis_pomedis + sum_desinysis_pomedis;
}
```

segmento režiai susikerta su ieškomojo segmento režiais. Intervalų sumos užklausą nėra sunku realizuoti(žr. 1.3 pav.)

1.3 pav. Segmentinio medžio sumos užklausos realizacija c++ programavimo kalba

Kadangi segmentinis medis yra labai universali duomenų struktūra, ji neapsiriboja vien tik sumos užklausų uždaviniais, bet padeda išspręsti ir kitas problemas kylančias su intervalais pvz.: minimumo arba maksimumo suradimas norimuose intervalo režiuose, principas yra tas pats kaip su sumos užklausomis, tačiau skirtumas tas, kad viršūnėje saugoma ne segmento suma, o segmento minimali arba maksimali reikšmė. Be abejo, sumą galime pakeisti kitomis komutatyviomis operacijomis, tokiomis kaip daugyba, disjunkcija, konjunkcija, griežtoji disjunkcija ir daugybe kitų.

Taip pat segmentinį medį galima pateikti kaip dviejų dimensių, tuomet galima vykdyti užklausas matricose. Tai mums suteikia galimybes spręsti tokias problemas kaip nustatymo minimalaus ir maksimalios reikšmės dėžėje, nustatymas kiek yra taškų koordinatų plokštumoje stačiakampyje, kurio kairiojo apatinio kampo koordinatės yra x_1 ir y_1 , o dešiniojo viršutinio x_2 ir y_2 ir daugybę kitų problemų (Ibte haz, Kaykobad and Rahman, 2018).

1.1.4. Atnaujinimo užklausa

Be intervalinių užklausų segmentinis medis palaiko atnaujinimo užklausas. Atnaujinimo užklausa dažniausiai susideda iš masyvo indekso, kurį norime atnaujinti ir reikšmės į kurią norime atitinkamą masyvo elementą pakeisti, norint visai tai padaryti reikia perkonstruoti mūsų segmentinį medį. Ši užklausa yra daug kartų paprastesnė už intervalinę užklausą. Kadangi kiekvienas segmentinio medžio lygis suformuoja masyvo dalį, tai vienas bet kuris masyvo elementas yra viename segmente kiekviename lygyje, tai sako, jog norint atnaujinti vieną masyvo elementą tereikia atnaujinti tik $\log_2 n$ segmentinio medžio viršūnių. Lengva pastebėti, kad atnaujinimo operaciją galima realizuoti su rekursyvią funkciją. Funkcijai yra paduodama dabartinė viršūnė, nauja reikšmė ir masyvo indeksas, kurį reikia atnaujinti, funkcija rekursiškai kviečia viena iš savo „vaikų“ į kuri įeiną dabartinis nagrinėjamas elementas ir jį atnaujiną, šis procesas vyksta tol kol nepasiekiami viršūnė lapas (žr. 1.4 pav.) (Sharma, 2016).

```

void update(int medzio_indeksas, int tl, int tr, int masyvo_indeksas, int nauja_reiksme)
{
    if( tl == tr ) {
        segmentinis_medis[medzio_indeksas] = nauja_reiksme; // Viršūnė lapas.
    }
    else {
        int tm = (tl + tr) / 2; // Paskaičiuojamas segmento vidurys.
        if ( masyvo_indeksas <= tm ) {
            // Elementas priklauso kairiajam pomedžiui.
            update(medzio_indeksas * 2, tl, tm, masyvo_indeksas, nauja_reiksme);
        }
        else {
            // Elementas priklauso dešiniajam pomedžiui.
            update(medzio_indeksas * 2 + 1, tm + 1, tr, masyvo_indeksas, nauja_reiksme);
        }

        // Atnaujiname sumą, kadangi pasikeitė elementai.
        segmentinis_medis[medzio_indeksas] = segmentinis_medis[medzio_indeksas * 2] +
            segmentinis_medis[medzio_indeksas * 2 + 1];
    }
}

```

1.4 pav. Segmentinio medžio atnaujinimo užklausos realizacija c++ programavimo kalba

1.1.5. Intervalinės atnaujinimo užklausos

Dažnai būna situacijų, kai reikia atnaujinti ne vieną reikšmę medyje, o visą ištisą segmentą, šiai operacijai atlikti segmentinis medis turi labai naudingą techniką vadinama „tingiu dauginimusi“. Naudojant šią techniką galima atidėti dalį atnaujinimų(praleisti rekursyvius kvietimus atnaujinimo funkcijoje) ir atlikti juos tik tuomet, kai jie yra reikalingi (Mittal, 2016).

Šiuo atveju kaip pavyzdys galėtų būti modifikacijos užklausa, kuri sako, jog mes turime priskirti kiekvienam segmento elementui specifinę reikšmę p , o sekančioji po jos einanti užklausa būtų nuskaitymas $a[i]$ masyvo reikšmės. Norint atlikti šią modifikaciją visam segmentui reikia saugoti atributą kiekvienoje segmentinio medžio viršūnėje, kuris nusako ar nagrinėjamas segmentas yra pilnai padengtas viena reikšme ar keliomis reikšmėmis. Tai leidžia atlikti „tingų atnaujinimą“: vietoj keitimo visų segmentų esančių medyje, kurie yra perdengiami užklausos segmento, užtenka pakeisti tik kelis segmentus ir pažymėti viršūnę. Pažymėta viršūnė nurodys tai, kad kiekvienas elementas esantis nagrinėjamame segmente turi priskirtą reikšmę ir pilnas po medis taip pat turėtų turėti tik šią reikšmę. Apibendrinus galima sakyti, jog yra „tingiai“ su uždelsimais įrašoma nauja reikšmė į kiekvieną iš viršūnių. Šiuo principu galima atidėti atnaujinimo užklausa vėlesniam laikui, kada medžiui atliekama intervalinė užklausa.

Taigi, po modifikacijos užklausos kai kurios medžio dalys tampa nesvarbiomis – kai kurios modifikacijos niekada nėra pilnai įvykdomos. Pavyzdžiui, kai modifikacijos užklausa „priskirti skaičių visam masyvui $a[0 \dots n - 1]$ “ yra įvykdoma, segmentiniame medyje pasikeičia tik vienas

dalykas – reikšmė esanti šaknyje, bei šaknies viršūnė yra pažymima. Likusieji segmentai išlieka nepakeisti, nors teoriškai reikšmė turėtų būti įrašyta visam medžiui.

Dar vienas iš pavyzdžių galėtų būti, jog antroji modifikacijos užklausa sako, kad pirmoji masyvo dalis $a \left[0 \dots \frac{n}{2}\right]$ turėjo būti priskirta kokiai nors reikšmei. Norint įvykdyti šią užklausą reikia priskirti kiekvienam elementui esančiam kairiajame šaknies po medyje nurodytą reikšmę, tačiau prieš tai atliekant reikia sutvarkyti šaknies viršūnę. Komplikuotumas čia atsiranda toks, jog dešinioji masyvo pusė vis dar turėtų būti priskirta reikšmei, kuria suteikė pirmoji modifikacijos užklausa ir kol kas šiuo metu jokia informacija nėra saugoma dešiniajame šaknies po medyje. Tai išsprendžiama perleidžiant informaciją iš šaknies savo „vaikams“, tai yra, jeigu medžio šaknis turėjo priskirtą kokią nors reikšmę, tuomet tą reikšmę reikia priskirti kairiajam ir dešiniajam šaknies „vaikui“ ir ištrinti žymą šaknyje. Po šio pakeitimo galima priskirti kairiajam šaknies „vaikui“ naują reikšmę neprarandant jokios informacijos.

Viską apibendrinus galima teigti, jog esant bet kokiai užklausiai (atnaujinimo arba nuskaitymo) nusileidimo medyje metu visuomet reikia nuleisti informaciją iš esamosios viršūnės į abidvi viršūnes „vaikus“. Tai reiškia, jog leidžiantis medžiu yra atliekamos uždelstos modifikacijos užklauskos, tačiau tik tiek kiek yra būtina (kad nepablogintumėme laiko kompleksiško $O(\log_2 n)$).

Norint daryti intervalines atnaujinimo užklauskas prie jau turimų metodų (žr. 1.3 pav., 1.4 pav.), reikia papildomai pasirašyti viršūnės nuleidimo metodą (žr. 1.5 pav.)

```
void push(int virsune)
{
    if ( pazymeta[viirsune] ) // Patikriname ar viršūnė yra pažymeta.
    {
        //Nuleidžiame informaciją kairiajam vaikui.
        segmentinis_medis[viirsune * 2] = segmentinis_medis[viirsune];
        // Nuleidžiame informaciją dešiniajam vaikui.
        segmentinis_medis[viirsune * 2 + 1] = segmentinis_medis[viirsune];
        // Pažymime kairįjį vaiką.
        pazymeta[viirsune * 2] = true;
        // Pažymime dešinyjį vaiką.
        pazymeta[viirsune * 2 + 1] = true;
        // Atžymime dabartinę viršūnę, kadangi informacija jau perdavėme vaikams.
        pazymeta[viirsune] = false;
    }
}
```

1.5 pav. Segmentinio medžio nuleidimo metodo realizacija c++ programavimo kalba

Šis naujasis metodas gaus dabartinę nagrinėjamą viršūnę ir nuleis informaciją į apačią savo „vaikams“, atžymės dabartinę viršūnę, kadangi informacija jau bus nuleista į apačią ir pažymės nagrinėjamos viršūnės „vaikus“. Šį metodą reikia kviesti užklausų metodų pradžioje, tačiau nereikia kviesti jų „vaikams“, nes šiuo metu nereikia leisti informacijos dar žemiau. Taip pat reikia šiek tiek pakeisti atnaujinimo metodą (žr. 1.6 pav.) ir pasirašyti gavimo metodą (žr. 1.7 pav.), kad viskas vyktų sklandžiai.

```
void update(int virsune, int tl, int tr, int kaire, int desine, int nauja_reiksme)
{
    if( kaire > desine ) return; // Patikriname ar validus segmentas.
    if( kaire == tl && tr == desine )
    {
        // Jeigu viršūnės rėžiai lygus mūsų ieškomiems priskiriame reikšmę.
        segmentinis_medis[virsune] = nauja_reiksme;
        pazymeta[virsune] = true; // Pažymime šią viršūnę.
    }
    else
    {
        push(virsune); // Nuleidžiame informaciją vaikams, jeigu to reikia.
        int vidurys = (tl + tr) / 2; // Surandame segmento vidurį.
        // Atnaujiname kairinį vaiką.
        update(virsune * 2, tl, vidurys, kaire, min(desine, vidurys), nauja_reiksme);
        // Atnaujiname dešinįjį vaiką.
        update(virsune * 2 + 1, vidurys + 1, tr, max(kaire, vidurys + 1), desine, nauja_reiksme);
    }
}
```

1.6 pav. Segmentinio medžio intervalinės atnaujinimo užklausos realizacija c++ programavimo kalba

```
int get(int virsune, int tl, int tr, int pozicija)
{
    // Kai sutampa segmentai randame ieškomąjį elementą.
    if( tl == tr ) return segmentinis_medis[virsune];
    push(virsune);
    int vidurys = (tl + tr) / 2; // Randame vidurį.
    if( pozicija <= vidurys ) { // Jeigu ne daugiau už vidurį ieškome kairiajame pomedyje.
        return get(virsune * 2, tl, vidurys, pozicija);
    }
    else { // Kitu atveju ieškome dešiniajam pomedyje.
        return get(virsune * 2 + 1, vidurys + 1, tr, pozicija);
    }
}
```

1.7 pav. Segmentinio medžio gavimo metodo realizacija c++ programavimo kalba

1.1.6. Dviejų dimensijų segmentinis medis

Segmentinis medis gali būti sukuriamas bet kokiam dimensijų skaičiui. Norint sukonstruoti dviejų dimensijų segmentinį medį reikia, jog kiekviena viršūnė esanti dviejų dimensijų segmentiniame medyje turėtų vienos dimensijos segmentinį medį, taigi tai iš esmės galima pavadinti segmentiniu medžiu iš segmentinių medžių (Aggrawal, 2017).

Šiuo atveju kaip pavyzdys galėtų būti, jog turime matricą $a[0 \dots n - 1, 0 \dots m - 1]$, kur čia n yra eilučių skaičius, o m yra stulpelių skaičius, turint šią matricą reikia surasti sumą (arba minimumą/maksimumą) kokioje nors matricos dalyje $a[x_1 \dots x_2, y_1 \dots y_2]$, bei atlikti modifikacijas individualiuose matricos elementuose, pvz.: užklausas formoje priskirti matricos elementui reikšmę $a[x][y] = u$.

Norint palengvinti konstravimo procesą galima pamiršti, jog matrica yra dviejų dimensijų ir palikti pirmąją koordinatę (koordinatės šiuo metu žymimos x ir y). Pirmiausia reikia konstruoti vienos dimensijos segmentinį medį naudojant pirmąją koordinatę, tačiau vietoj skaičiaus saugojimo segmentinio medžio viršūnėje, kaip jau minėta vienos dimensijos atveju konstravimo poskyryje reikia saugoti visa segmentinį medį tai yra šiuo momentu reikia prisiminti, jog egzistuoja antroji koordinatė. Kadangi šiuo metu pirmoji koordinatė yra fiksuota intervale $[l \dots r]$, reikia atlikti pakeitimus su $a[l \dots r, 0 \dots m - 1]$ matricos dalimi ir jai sukurti segmentinį medį.

Taigi, konstravimą dviejų dimensijų segmentinio medžio galima išskirti į dvi dalis:

- 1) Konstravimą x koordinatės atžvilgiu (žr. 1.8 pav.)
- 2) Konstravimą y koordinatės atžvilgiu (žr. 1.9 pav.)

```
void build_x(int virsune_x, int kaire_x, int desine_x)
{
    // Kuriame vienos dimensijos segmentinį medį x koordinate tol, kol turime segmentą.
    if (kaire_x != desine_x)
    {
        int vidurys_x = (kaire_x + desine_x) / 2; // Surandame vidurį.
        // Kuriame kairinį pomedį x koordinatės atžvilgiu.
        build_x(virsune_x * 2, kaire_x, vidurys_x);
        // Kuriame dešinįjį pomedį x koordinatės atžvilgiu.
        build_x(virsune_x * 2 + 1, vidurys_x + 1, desine_x);
    }

    // Kuriame segmentinį medį ties y koordinate, kai x koordinatė yra fiksuota.
    build_y(virsune_x, kaire_x, desine_x, 1, 0, m - 1);
}
```

1.8 pav. Konstravimas dviejų dimensijų segmentinio medžio x koordinatės atžvilgiu

```

void build_y(int virsune_x, int kaire_x, int desine_x, int virsune_y, int kaire_y, int desine_y)
{
    if (kaire_y == desine_y) // Patikriname ar tai jau galime sukonstruoti lapo viršūnę.
    {
        // Priskiriame reikšmę tiesiogiai iš masyvo.
        if (kaire_x == desine_x) seg_tree[virsune_x][virsune_y] = a[kaire_x][kaire_y];
        else {
            // Sukombinuojame reikšmę iš kairiojo vaiko.
            seg_tree[virsune_x][virsune_y] = seg_tree[virsune_x * 2][virsune_y];
            // Pridedame ir dešiniąjį vaiką.
            seg_tree[virsune_x][virsune_y] += seg_tree[virsune_x * 2 + 1][virsune_y];
        }
    }
    else
    {
        int vid_y = (kaire_y + desine_y) / 2; // Surandame vidurį.
        // Kuriame kairinį pomedį y koordinatės atžvilgiu.
        build_y(virsune_x, kaire_x, desine_x, virsune_y * 2, kaire_y, vid_y);
        // Kuriame dešiniąjį pomedį y koordinatės atžvilgiu.
        build_y(virsune_x, kaire_x, desine_x, virsune_y * 2 + 1, vid_y + 1, desine_y);
        // Sukombinuojame reikšmę iš kairiojo vaiko.
        seg_tree[virsune_x][virsune_y] = seg_tree[virsune_x][virsune_y * 2];
        // Pridedame ir dešiniąjį vaiką.
        seg_tree[virsune_x][virsune_y] += seg_tree[virsune_x][virsune_y * 2 + 1];
    }
}

```

1.9 pav. Konstravimas dviejų dimensijų segmentinio medžio y koordinatės atžvilgiu

Lapų konstravimui y koordinatės atžvilgiu yra išskiriami 2 skirtingi atvejai:

2.1) Kai dabartinis nagrinėjamas segmentas pirmosios koordinatės [$kairė_x \dots dešinė_x$] turi ilgį lygų vienam, tuomet tiesiogiai yra inicializuojamas segmentinis medis iš matricos.

2.2) Kai dabartinis nagrinėjamas segmentas pirmosios koordinatės [$kairė_x \dots dešinė_x$] turi ilgį didesnį negu vienas, tuomet yra kombinuojamos reikšmės iš kairiojo ir dešiniojo viršūnės „vaikų“ pagal x koordinatę.

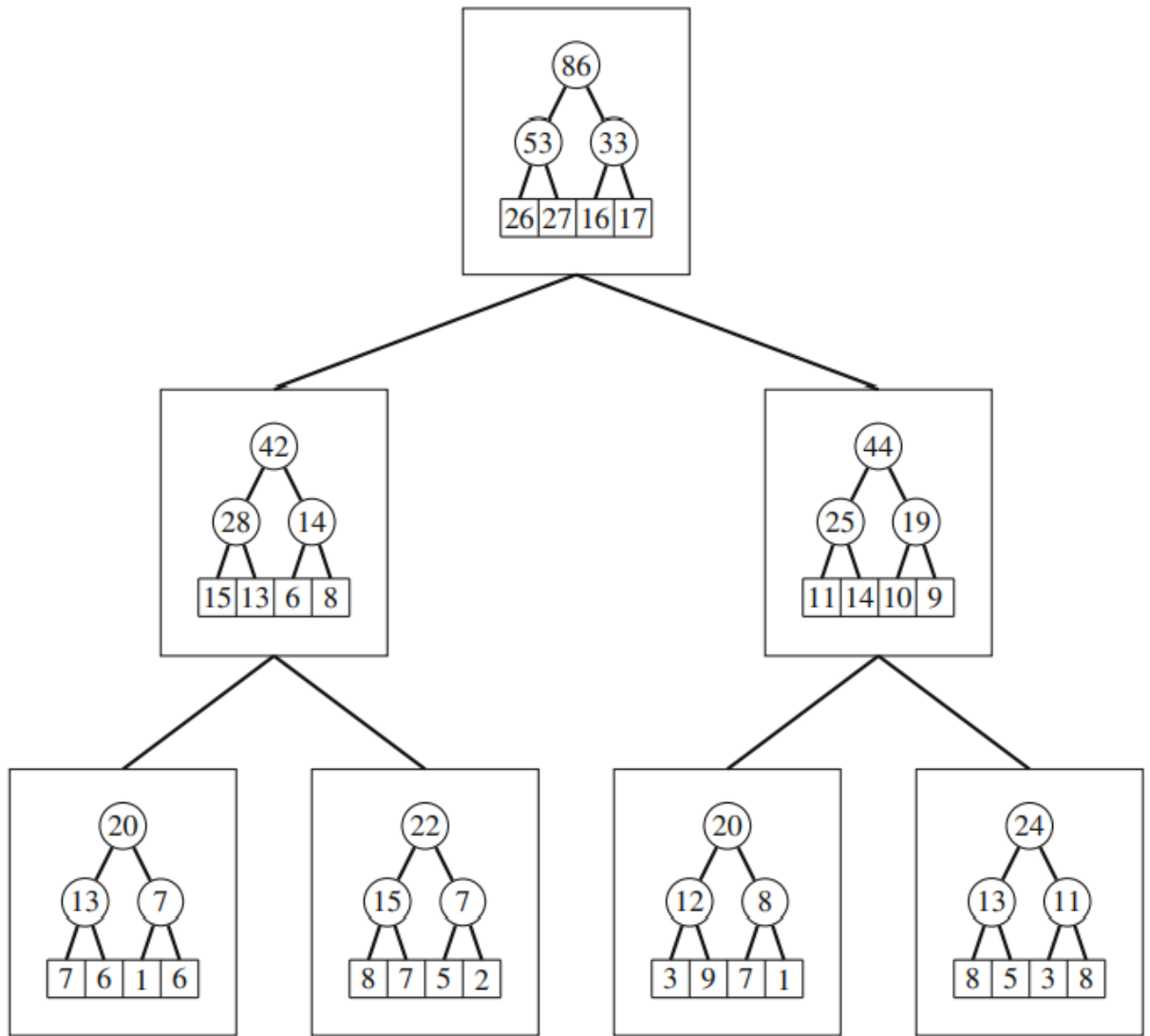
Toks segmentinis medis dar vis kaip ir vienos dimensijos atvejis naudoja tiesinį kiekį atminties, tačiau su didele konstanta: $O(16 * n * m)$.

Vizualiai sukonstruoto dviejų dimensijų segmentinio medžio struktūra gali būti sunku įsivaizduoti, tačiau ją galima nupiešti turint 4×4 matricą su užpildytais duomenimis. (žr. 1.10 pav.)

7	8	3	8
6	7	9	5
1	5	7	3
6	2	1	8

1.10 pav. Pavyzdinė duomenų matrica (Laaksonen, 2017)

Iš šios duomenų matricos sukonstruotas dviejų dimensijų segmentinis medis palaiko dvi užklausas: sumų apskaičiavimą pasirinktuose matricų stačiakampiuose, bei atnaujinimą bet kurios matricos reikšmės (žr. 1.11 pav.). Pavyzdžiui suma viso stačiakampio yra lygi 86, suma stačiakampio pirmų dviejų eilučių yra lygi 53, šias sumų užklausas galima apskaičiuoti efektyviai atsakyti naudojantis dviejų dimensijų segmentiniu medžiu.



1.11 pav. Dviejų dimensijų segmentinio medžio vizualizacija (Laaksonen, 2017)

Dviejų dimensijų užklausa yra vykdoma tuo pačiu principu kaip ir konstravimas, pirmiausią užklausa reikia vykdyti pagal pirmąją koordinatę, su kuria reikia pasiekti atskaitos viršūnę, o esant joje iškviesti užklausa antrajai koordinatei (žr. 1.12 pav.).

Ši užklausa veikia $O(\log_2 n * \log_2 m)$ greičiu, kadangi pirmiausia pereina per pirmosios koordinatės segmentinį medį ir po to kiekvienai surastai viršūnei daro užklausa pagal antrąją koordinatę.

```

int sumy(int virsune_x, int virsune_y, int kaire_y, int desine_y, int kaire_uzklauso_y,
        int desine_uzklauso_y)
{
    if (kaire_uzklauso_y > desine_uzklauso_y) return 0; // Patikriname ar validus segmentas
    if (kaire_uzklauso_y == kaire_y && desine_y == desine_uzklauso_y)
        return seg_tree[virsune_x][virsune_y]; // Grąžiname sumą.
    int vidurys = (kaire_y + desine_y) / 2; // Paskaičiuojame vidurį.
    // Paskaičiuojame sumą pagal kairinį pomedį.
    int suma_kaire = sumy(virsune_x, virsune_y * 2, kaire_y, vidurys,
                        kaire_uzklauso_y, min(desine_uzklauso_y, vidurys));
    // Paskaičiuojame sumą pagal dešinįjį pomedį.
    int suma_desine = sumy(virsune_x, virsune_y * 2 + 1, vidurys + 1, desine_y,
                        max(kaire_uzklauso_y, vidurys + 1), desine_uzklauso_y);
    return suma_kaire + suma_desine;
}

int sumx(int virsune_x, int kaire_x, int desine_x, int kaire_uzklauso_x,
        int desine_uzklauso_x, int kaire_uzklauso_y, int desine_uzklauso_y)
{
    if (kaire_uzklauso_x > desine_uzklauso_x) return 0; // Patikriname ar validus segmentas.
    // Kiekvienai viršūnei skaičiuojame pagal antrąją koordinatę.
    if (kaire_uzklauso_x == kaire_x && desine_x == desine_uzklauso_x)
        return sumy(virsune_x, 1, 0, m - 1, kaire_uzklauso_y, desine_uzklauso_y);
    int vidurys = (kaire_x + desine_x) / 2; // Paskaičiuojame vidurį.
    // Paskaičiuojame sumą pagal kairinį pomedį.
    int suma_kaire = sumx(virsune_x * 2, kaire_x, vidurys, kaire_uzklauso_x,
                        min(desine_uzklauso_x, vidurys),
                        kaire_uzklauso_y, desine_uzklauso_y);
    // Paskaičiuojame sumą pagal dešinįjį pomedį.
    int suma_desine = sumx(virsune_x * 2 + 1, vidurys + 1, desine_x,
                        max(kaire_uzklauso_x, vidurys + 1),
                        desine_uzklauso_x, kaire_uzklauso_y, desine_uzklauso_y);
    return suma_kaire + suma_desine;
}

```

1.12 pav. Dviejų dimensijų segmentinio medžio sumos užklauso realizacija c++ programavimo kalba

Taip pat egzistuoja tokių problemų kur yra reikalinga dviejų dimensijų segmentinio medžio kompresija. Pavyzdžiui koordinačių plokštumoje egzistuoja n taškų, kurie yra apibrėžiami savo koordinatėmis (x_i, y_i) ir yra užduodamos užklauso, kurios sako, kad reikia suskaičiuoti kiek yra taškų stačiakampyje $((x_1, y_1), (x_2, y_2))$. Gavus tokią problemą tampa akivaizdu, kad neapsimoka konstruoti dviejų dimensijų segmentinio medžio su $O(n^2)$ elementų, kadangi didžioji dalis atminties nebus netgi naudojama. Pilna atmintis nebus naudojama dėl to, kad kiekvienas taškas gali pakliūti tik į $\log_2 n$ segmentus pagal pirmąją koordinatę iš to seka, kad naudingas dydis visų segmentų pagal antrąją koordinatę turėtų būti $O(n * \log_2 n)$.

Taigi, norint išspręsti šią problemą reikia imti kiekvieną segmentinio medžio viršūnę pagal pirmąją koordinatę ir saugoti joje segmentinį medį pagal antrosios koordinatės viršūnes, kurios pasikartoja pirmosios koordinatės nagrinėjamame segmente. Jeigu reikia konstruoti segmentinį medį viršūnėje $virsune_x$, kurios rėžiai yra $kaire_x$ ir $desine_x$, nagrinėti reikia tik tuos taškus,

kurie pakliūva į intervalą $x \in [kaire_x, desine_x]$. Taip darant bus pasiekta, jog kiekviena segmentinio medžio antroji koordinatė užims tik tiek atminties kiek reikia. Rezultatas šios optimizacijos bus toks, kad atminties panaudojimas sumažės iš $O(n^2)$ iki $O(n * \log_2 n)$. Norint toliau atsakyti užklausas per $O(\log_2 n * \log_2 n)$ laiką, tereikės naudoti dvejetainę paiešką³ einant per antrąją koordinatę. Ši operacija laiko kompleksiskumo nepablogins, kadangi dvejetainė paieška veikia per $O(\log_2 n)$ laiką (Laber, Molinaro, 2009) (Clampett, 1964). Vienas šios struktūros panaudos minusas yra tas, kad nebus galima efektyviai atlikti modifikacijos užklausų, kadangi atsiradus naujam taškui, kurį reikia įdėti į segmentinio medžio vidurį pagal antrąją koordinatę reikia perdaryti dalį segmentinio medžio struktūros.

1.1.7. Išsaugojimas segmentinio medžio istorijos

Segmentinis medis, kuris gali išsaugoti prieš tai buvusią būseną po bet kokių modifikacijų yra vadinamas pastoviu segmentiniu medžiu (angl. *Persistent Segment Tree*). Šio tipo segmentinis medis gali suteikti prieigą prie bet kurios segmentinio medžio versijos, kurios reikia ir įvykdyti užklausą būtent toje versijoje. Be abejo, kadangi yra saugoma visa medžio istorija nėra išvengiama didesnio atminties sunaudojimo, kas daro įtaką ir laiko kompleksiskumui (Kumar, 2016).

Norint iš segmentinio medžio gauti pastovųjį segmentinį medį užtenka pakeisti tik $O(\log_2 n)$ viršūnių pradedant šaknies viršūne. Norint tai padaryti reikia segmentinį medį kurti naudojant rodykles (angl. *Pointers*) tai yra kiekviena viršūnė turi saugoti rodyklę į savo kairįjį ir dešinįjį „vaiką“, taip saugant viršūnės sulaukus modifikacijos užklausos nereikia keisti esamųjų viršūnių, užtenka sukurti naujas viršūnes (žr. 1.13 pav., 1.14 pav., 1.15 pav., 1.16 pav.). Viršūnės kurios nėra paveiktos modifikacijos užklausos toliau gali rodyti į senas viršūnes. Taigi, įvykus modifikacijos užklausiai bus sukurtos $\log_2 n$ naujos viršūnės, bei nauja segmentinio medžio šaknies viršūnė, o sena segmentinio medžio versija išliks nepakitusi. Be abejo galima kurti kiekvienai segmentinio medžio versijai naują segmentinį medį tačiau toks sprendimas nėra optimalus, kadangi kiekvienas naujos versijos sukūrimas užims $O(\log_2 n)$ laiko konstravimui, bei $O(n * \log_2 n)$ papildomai atminties.

³ Paieškos algoritmas, kai elemento ieškoma surikiuotame sąrašė jį dalinant pusiau.

```

CVirsune * build(int kaire, int desine)
{
    if( kaire == desine) {
        // Sukuriame naują viršūnę, kadangi segmentai sutampa.
        return new CVirsune(a[kaire]);
    }

    int vidurys = (kaire + desine) / 2; // Surandame vidurį.
    // Sukuriame kairinį vaiką.
    CVirsune *pKaire = build(kaire, vidurys);
    // Sukuriame dešinį vaiką.
    CVirsune *pDesine = build(vidurys + 1, desine);
    // Sukonstruojame viršūnę iš dviejų vaikų.
    return new CVirsune(pKaire, pDesine);
}

```

1.13 pav. Konstravimas segmentinio medžio naudojant rodykles

```

int sum(CVirsune *pVirsune, int tl, int tr, int kaire, int desine)
{
    if( kaire > desine ) {
        return 0; // Patikriname ar validus segmentas.
    }

    if( kaire == tl && desine == tr ) {
        // Jeigu ieškomasis segmentas sutampa su nagrinėjamu grąžiname sumą.
        return pVirsune->GautiSuma();
    }

    int vidurys = (tl + tr) / 2; // Surandame vidurį.
    // Paskaičiuojame kairiojo pomedžio sumą.
    int suma_kaire = sum(pVirsune->GautiKairiajaVirsune(), tl, vidurys,
                        kaire, std::min(desine, vidurys));
    // Paskaičiuojame dešiniojo pomedžio sumą.
    int suma_desine = sum(pVirsune->GautiDesiniajaVirsune(), vidurys + 1, tr,
                        std::max(kaire, vidurys + 1), desine);
    return suma_kaire + suma_desine;
}

```

1.14 pav. Segmentinio medžio sumos užklausa naudojant rodykles

```

class CVirsune
{
public:
    CVirsune(int suma) : m_suma(suma), m_pKaire(nullptr), m_pDesine(nullptr) { }
    CVirsune(CVirsune *pKaire, CVirsune *pDesine) : m_pKaire(pKaire), m_pDesine(pDesine), m_suma(0)
    {
        if( pKaire ) { // Jeigu egzistuoja kairysis vaikas pridedame jo sukauptą sumą
            m_suma += pKaire->GautiSuma();
        }

        if( pDesine ) { // Jeigu egzistuoja dešinysis vaikas pridedame jo sukauptą sumą
            m_suma += pDesine->GautiSuma();
        }
    }

public:
    int GautiSuma() const { return m_suma; }
    CVirsune *GautiKairiajaViršune() const { return m_pKaire; }
    CVirsune *GautiDesiniajaViršune() const { return m_pDesine; }

private:
    int m_suma;           // Suma esamoje viršūnėje
    CVirsune *m_pKaire;   // Kairysis viršūnės vaikas
    CVirsune *m_pDesine;  // Dešinysis viršūnės vaikas
};

```

1.15 pav. Segmentinio medžio viršūnės struktūra

```

CVirsune *update(CVirsune *pViršune, int tl, int tr, int pozicija, int nauja_reiksme)
{
    // Jeigu segmentai sutampa radome viršūnę ir ją atnaujiname sukurdami.
    if( tl == tr ) return new CVirsune(nauja_reiksme);
    int vidurys = (tl + tr) / 2; // Surandame vidurį.
    CVirsune *pKaire = pViršune->GautiKairiajaViršune();
    CVirsune *pDesine = pViršune->GautiDesiniajaViršune();
    if( pozicija <= vidurys ) {
        // Atnaujiname kairinį pomedį.
        pKaire = update(pKaire, tl, vidurys, pozicija, nauja_reiksme);
    }
    else {
        // Atnaujiname dešinįjį pomedį.
        pDesine = update(pDesine, vidurys + 1, tr, pozicija, nauja_reiksme);
    }

    // Sukuriame naują atnaujintą viršūnę pagal kairinį dešinįjį pomedžius.
    return new CVirsune(pKaire, pDesine);
}

```

1.16 pav. Segmentinio medžio atnaujinimo užklausa naudojant rodykles

1.2. Dvejetainis indeksuotas medis

Dvejetainis indeksuotas medis buvo sukurtas 1994 metais norint pagerinti aritmetinio kodavimo kompresijos algoritmus. Ši duomenų struktūra literatūroje dar žinoma kaip „Fenwicko medis“, jos paskirtis yra efektyviai atnaujinti elementus ir paskaičiuoti prefiksines sumas skaičių lentelėse (Fenwick, 1994). Prefiksinė suma informatikoje yra ne kas daugiau kaip sumavimas iš eilės einančių elementų $x_0, x_1, x_2, \dots$ kur bet kuriame indekse gali būti pateikiama prefiksinė suma t.y. suma visų prieš taiėjusių elementų (žr. 1.1 lentelė)

1.1 lentelė. Prefiksinių sumų pavyzdys

Įvesties skaičiai	1	2	3	4	5	6	7	8	9
Prefiksinės sumos	1	3	6	10	15	21	28	36	45

Nors teoriškai dvejetainiai indeksuoti medžiai yra laikomi medžiais, tačiau praktikoje jie yra realizuojami kaip nuspėjamos duomenų struktūros⁴(angl. *Implicit data structures*) naudojant masyvus. Dvejetainis indeksuotas medis yra daugiausia naudojamas intervalinių užklausų problemoms spręsti logaritminiu greičiu (Gopani, 2019).

1.2.2. Struktūra

Kadangi bet kokį natūralų skaičių galima išsireikšti per dvejetų laipsnių sumą (Nazarathy, 2017) (žr. 1.2 lentelė). Šiuo faktu galima pasinaudoti ir prefiksinių sumų skaičiavimui jas suskirsčius pagal atitinkamus indeksus, kuriuos sudėjus galima gauti bet kokį skaičių.

1.2 lentelė. Skaičiaus išreiškimas per dvejetų laipsnių sumas

Suma	Daugianaris
17	$2^4 + 2^0$
12	$2^3 + 2^2$
14	$2^3 + 2^2 + 2^1$

Prieš tai padarant reikia žinoti, jog bet kokį skaičių esanti dešimtainėje skaičiavimo sistemoje galima konvertuoti į dvejetainę skaičiavimo sistemą. „Norint dešimtainį skaičių užrašyti dvejetainėje skaičiavimo sistemoje reikia šį skaičių ir jo dalmenis dalinti iš dviejų tol, kol dalmuo bus mažesnis už 2, tada gautas dalybos liekanas ir paskutinį dalmenį reikia surašyti atvirkščia

⁴ Duomenų struktūra, kuri saugo tik pačią reikalingiausią informaciją ir kurios elementai yra susieti ryšiais pagal užimamas pozicijas

gavimui tvarka tai yra pirma gauta liekana yra paskutinis skaitmuo, antra - priešpaskutinis ir t.t., paskutinis dalmuo - pirmas skaitmuo“ (Ragaišis, 2001) (žr. 1.3 lentelė).

1.3 lentelė. Skaičiai užrašyti dešimtainę ir dvejetainę skaičiavimo sistema

Skaičius dešimtainėje skaičiavimo sistemoje	Skaičius dvejetainėje skaičiavimo sistemoje
17	10001
12	1100
14	1110

Taigi, pagal šiame skyriuje minėta teorija tegul *index* būna dvejetainio indeksuoto medžio indeksas, o *k* būna pozicija esanti skaičiaus *index* dvejetainėje formoje, kuri nurodo paskutinį nelygu nuliui skaitmenį, t.y *k* nurodo paskutinį vienetuką esanti skaičiaus *index* dvejetainėje formoje. Įvedant dar kelis kintamuosius:

1) $a = (a_1, a_2 \dots, a_n)$ sveikųjų skaičių masyvas sudarytas iš *n* elementų.

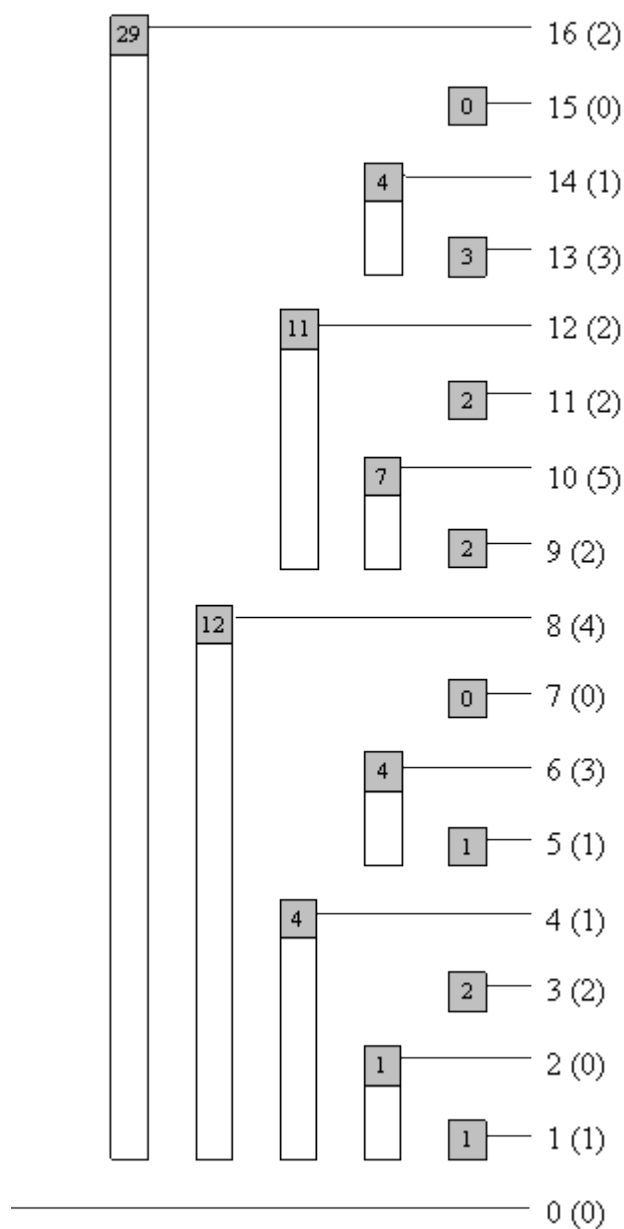
2) $prefixa = (prefixa_1, prefixa_2 \dots, prefixa_n)$ masyvo *a* prefiksinių sumų masyvas sudarytas iš *n* elementų.

3) $medis = (medis_1, medis_2, \dots, medis_n)$ dvejetainis indeksuotas medis iš *n* elementų.

Tuomet $medis[index]$ parodo sumą elementų nuo indeksų $[index - 2^k + 1 \dots index]$ (žr. 1.4 lentelė). Taip pat svarbu pažymėti, jog *index* yra atsakingas už medžio indeksus esančius nuo $index - 2^k + 1$ iki *index* (žr. 1.17 pav.).

1.4 lentelė. Dvejetainio indeksuoto medžio duomenų pavyzdys (Binary Indexed Trees, 2018)

Indeksas	Masyvas <i>a</i>	Masyvas <i>prefixa</i>	Masyvas <i>medis</i>
1	1	1	1
2	0	1	1
3	2	3	2
4	1	4	4
5	1	5	1
6	3	8	4
7	0	8	0
8	4	12	12
9	2	14	2
10	5	19	7
11	2	21	2
12	2	23	11
13	3	26	3
14	1	27	4
15	0	27	0
16	2	29	29



1.17 pav. Dvejetainis indeksuotas medis parodantis indeksų atsakingumą ir viršūnių sumas (Binary Indexed Trees, 2014)

Norint gauti sumą pirmų trylikos masyvo elementų reikia pirmų 13 medžio indeksų. Dvejetainėje skaičiavimo sistemoje skaičius 13 yra lygus 11012. Iš viršuje pateikto medžio galima nustatyti, kad norint paskaičiuoti pirmų trylikos masyvo elementų sumą reikia sudėti medžio 1000₂, 1100₂, 1101₂ elementus, kas dešimtainėje skaičiavimo sistemoje atitiktų 8, 12, 13 medžio elementus, kadangi kaip matyti iš vizualizacijos aštuntasis medžio elementas apima pirmuosius 8 elementus, dvyliktasis medžio elementas apima sekančius 4 elementus, o tryliktasis medžio

elementas apima paskutinį vienintelį reikiamą elementą iš ko ir susideda suma pirmųjų trylikos masyvo elementų.

1.2.3. Konstravimas

Dažniausiai pasitaiko keli populiarūs dvejetainio indeksuoto medžio konstravimo būdai. Apžvelgus šio medžio struktūrą galima įsitikinti, jog pakankamai nesunku pasiekti medžio elementus ir jais manipuluoti.

Vienas iš paprasčiausių, bet mažiausiai efektyvių būdų inicializuoti dvejetainio indeksuoto medžio duomenų struktūrą yra naudojant šio medžio atnaujinimo operaciją, apie kurią bus rašomas atskiras poskyris. Pirmiausia yra inicializuojamos visos medžio viršūnės nuliui, o po to yra einama pa žingsniui per duotą masyvą ir yra iškviečiama atnaujinimo operacija kiekvienai viršūnei (žr. 1.18 pav.). Laiko kompleksiskumas šiam konstravimui yra lygus $O(n * \log_2 n)$, kur n yra masyvo elementų skaičius, laiko kompleksiskumas yra toks atitinkamai dėl to, kad kaip bus nustatyta tolimesniame poskyryje atnaujinimo užklausa užima $O(\log_2 n)$ laiko (Binary Indexed Tree or Fenwick Tree, 2014). Be abejo atminties naudojimas yra $O(n)$, kadangi medyje yra n elementų, tiek pat kiek ir masyve.

```
void build(int n)
{
    for (int index = 1; index <= n; index++) {
        binarinis_indeksuotas_medis[index] = 0; // Inicializuojame visą dvejetainį indeksuotą medį nuliui
    }

    for (int index = 1; index <= n; index++) {
        update(index, a[index]); // Atnaujiname reikiamas medžio viršūnes
    }
}
```

1.18 pav. Pirmoji dvejetainio indeksuoto medžio konstravimo algoritmo realizacija c++ programavimo kalba

Dar vienas iš būdų sukonstruoti dvejetainį indeksuotą medį yra pasinaudoti faktu, kad kiekviena medžio indeksė $index$ esanti reikšmė gali būti išsiskaičiuojama iš prieš tai paskaičiuotų prefiksinių sumų pagal formulę: $medis[index] = prefa[index] - prefa[index - 2^k + 1]$ (Binary Indexed Trees, 2019). Realizacijos konstravimo laiko kompleksiskumą palyginus su prieš tai pateikta realizacija yra geresnė, kadangi laikas yra tiesinis $O(n)$, taip yra dėl to, kad kiekvienas medžio elementas bet kuriame indekse gali būti paskaičiuojamas per $O(1)$ laiko kompleksiskumą. Be abejo viso medžio konstravimo laikas turi mažą konstantą lygi 2, kadangi prieš tai reikia

susidaryti prefiksinių sumų masyvą (žr. 1.19 pav.). Atminties naudojimas šios realizacijos yra toks pat kaip ir pirmosios $O(n)$.

```
void build()
{
    prefa[0] = 0; // Inicializuojame bazinę reikšmę
    for(int i = 1; i <= n; ++i) {
        prefa[i] = prefa[i - 1] + a[i]; // Paskaičiuojame prefiksinių sumų masyvą.
    }

    for(int index = 1; index <= n; index++)
    {
        int kaire = index - (index & (-index)); // Paskaičiuojame kairės pusės indeksą
        binarinis_indeksuotas_medis[index] = prefa[index] - prefa[kaire]; // Inicializuojame index medžio viršūnę
    }
}
```

1.19 pav. Antroji dvejetainio indeksuoto medžio konstravimo algoritmo realizacija c++ programavimo kalba

Pats efektyviausias būdas inicializuoti dvejetainį indeksuotą medį yra pastebėti faktą, kad kiekvienam indeksui $index$ reikia pridėti $a[index]$ reikšmę pozicijose $index, index + (index \& -index)$ ir taip toliau iteruojant per dvejetainio indeksuoto medžio struktūrą. Be to, $a[index + (index \& -index)]$ reikšmė turi būti pridėta visose šiose pozicijose išskyrus poziciją $index$ (žr. 1.20 pav.). Šio algoritmo laiko kompleksiskumas yra taip pat tiesinis $O(n)$, bet neegzistuoja jokia papildoma konstanta. Atminties naudojimas kaip ir su ankstesniu ju yra taip pat $O(n)$.

```
void build_dvej()
{
    for(int index = 1; index <= n; ++index) {
        // Pridedame masyvo reikšmę į indeksuotą medį.
        binarinis_indeksuotas_medis[index] += a[index];
        int naujas_index = index + (index & -index);
        // Patikriname ar neužeiname už ribų.
        if( naujas_index <= n ) {
            // Pridedame reikiamą medžio reikšmę.
            binarinis_indeksuotas_medis[naujas_index] += binarinis_indeksuotas_medis[index];
        }
    }
}
```

1.20 pav. Dvejetainio indeksuoto medžio konstravimo realizacija c++ programavimo kalba

1.2.4. Intervalinės užklausos

Viena iš užklausų į kurią gali atsakyti dvejetainis indeksuotas medis yra intervalinė užklausa. Jis sugeba atsakyti į $[0..r]$ tipo sumos užklausas, o tai reiškia, kad nurodžius kokį nors intervalo rėžį r jis turėtų pasakyti kokia yra suma nuo pirmojo masyvo elemento iki r -tojo. Tačiau galima atsakinėti ir į kitokio tipo užklausas, kurios yra tokios formos: $[l..r]$ tai yra atliekama

suskaičiuojant dvi sumas $[0..l-1]$ ir $[0..r]$, gaunant jų skirtumą, kur $[0..r]$ yra turinys, o $[0..l-1]$ yra atėminys. Norint paskaičiuoti sumą iki indekso *index* reikia atlikti keletą labai nesunkiai suprogramuojamų veiksmų (žr. 1.21 pav.):

- 1) Inicializuoti dabartinę esamą sumą nuliui.
- 2) Pridėti *medis[index]* reikšmę prie esamos sumos.
- 3) Išmesti paskutinį vienetą iš dvejetainės *index* skaičiaus reikšmės.
- 4) Kartoti nuo 2 punkto tol, kol *index* yra daugiau už nulį.

Norint atlikti trečiąjį žingsnį reikia turėti gerą algoritmą išmesti paskutinį vienetą, kadangi ši operacija yra labai dažnai atliekama visos užklauskos metu. Šią operaciją nėra sunku sugalvoti kaip atlikti efektyviai. Žiūrint į skaičiaus dvejetainę reprezentaciją ją galima užrašyti formoje $a1b$, kur a reiškia visus skaičius einančius prieš paskutinį vienetuką, o b reiškia visus nuliukus einančius po paskutinio vienetuko. Žinant, kad šio dvejetainio skaičiaus reprezentaciją galime užrašyti tokia forma: $invert(a1b) + 1 = invert(a)(0)invert(b) + 1$, kadangi neigiamas skaičius dvejetainėje formoje yra konvertuojamas invertuojant⁵ visus skaitmenis ir pridendant prie rezultato vienetą. Dėl to, kad seniau b susidėjo iš visų vienetukų dabar $invert(b)$ susideda vien iš vienetukų. Galiausiai yra gaunamas tokių transformacijų perėjimas (Fenwick, 1993):

- 1) $invert(a1b) + 1$
- 2) $invert(a)(0)invert(b) + 1$
- 3) $invert(a)(0)(1 \dots 1) + 1$
- 4) $invert(a)1(0 \dots 0)$
- 5) $invert(a)1b$

Paskutinįjį vienetą galima išmesti naudojant konjunkciją⁶. Pavyzdžiui galimos reikšmės sujungiant teiginius x ir y (žr. 1.5 lentelė)

1.5 lentelė. Konjunkcijos teisingumo lentelė

x	y	x ir y
Teisinga	Teisinga	Teisinga
Teisinga	Klaidinga	Klaidinga
Klaidinga	Teisinga	Klaidinga
Klaidinga	Klaidinga	Klaidinga

⁵ Loginis veiksmas, kurį atlikus esama loginė reikšmė pakeičiama priešinga.

⁶ Loginis veiksmas, sujungiantis du ar daugiau teiginių.

Atlikus konjunkcijos operaciją su dabartiniais mūsų turimais duomenimis yra gaunama: $a1b$ ir $invert(a)1b = (0..0)1(0..0)$, kur čia a reprezentacijoje yra gaunami nuliukai dėl to, kad pažiūrėjus pagal konjunkcijos teisingumo lentelę visą laiką skirtingus teiginius sujungiant rezultate yra gaunamas nulis.

```
int prefixsuma(int index)
{
    int suma = 0;

    for(;index > 0;)
    {
        suma += binarinis_indeksuotas_medis[index]; // Didiname sumą
        index -= (index & -index); // Išmetame paskutinį vienetuką dvejetainėje reprezentacijoje
    }

    return suma;
}

int suma(int l, int r) {
    return prefixsuma(r) - prefixsuma(l - 1);
}
```

1.21 pav. Dvejetainio indeksuoto medžio sumos užklausos realizacija c++ programavimo kalba

Dvejetainio indeksuoto medžio sumos užklausos laiko kompleksiskumas yra $O(\log_2 \text{maxindex})$, kur čia maxindex yra maksimalus galimas indeksas.

1.2.5. Atnaujinimo užklausa

Dar viena iš užklausų kurią sugeba įvykdyti dvejetainis indeksuotas medis yra atnaujinimo užklausa, kuri teigia, kad atsinaujinus kokiam nors masyvo elementui su indeksu index turi atsinaujinti ir visas medis. Norint tai atlikti reikia vadovautis tuo pačius principu kaip ir su sumos užklausomis. Norint padidinti indekse index esančią reikšmę skaičiumi x , reikia:

- 1) Padidinti startiniame nagrinėjamame indekse reikšmę x vienetais, tai yra indeksas, kuris pirmiausia ir pasikeitė.
- 2) Pridėti paskutinį vienetą esantį dvejetainėje dabartinio nagrinėjamo indekso reprezentacijoje prie savęs.
- 3) Antrą žingsnį kartoti tol, kol indeksas neviršija maksimalaus galimojo indekso.

Iš esmės atnaujinimo užklausos funkcija naudoja invertuota sumos užklausos funkcijos veiksmą, minusą pakeičia pliusu (žr. 1.22 pav.).

Šios užklauso laiko kompleksiškumas yra logaritminis, kadangi niekada nereikia atnaujinti daugiau negu $\log_2 \text{maxindex}$ viršūnių.

```
void update(int index, int x)
{
    for( ;index <= n; )          // Tikriname ar neviršijome maksimalaus galimo indekso
    {
        binarinis_indeksuotas_medis[index] += x; // Atnaujiname indekse index esančią reikšmę.
        index += (index & -index); // Pridedame paskutinį vienetuką esanti dvejetainėje
                                   // dabartinio nagrinėjamo indekso reprezentacijoje prie savęs.
    }
}
```

1.22 pav. Dvejetainio indeksuoto medžio atnaujinimo užklauso realizacija c++ programavimo kalba

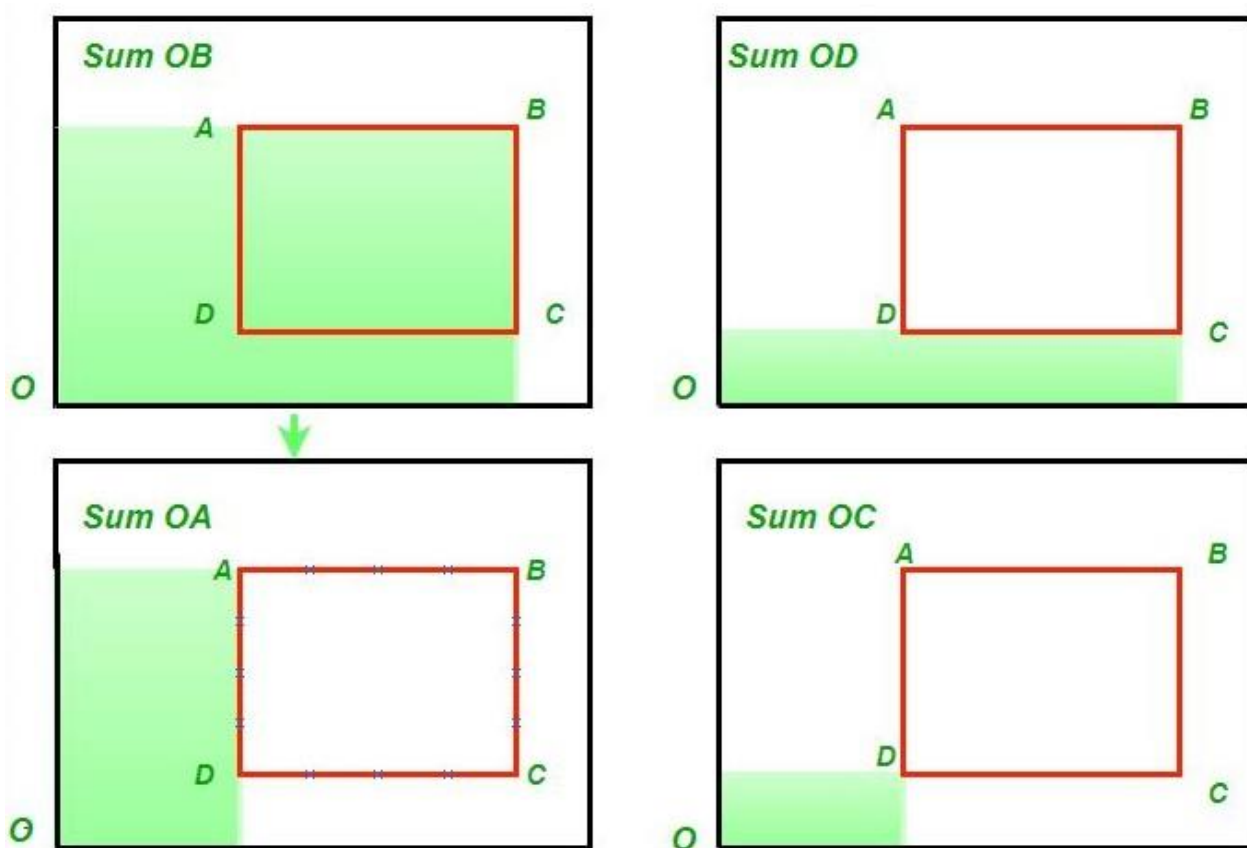
1.2.6. Dviejų dimensijų dvejetainis indeksuotas medis

Dvejetainis indeksuotas medis gali būti konstruojamas bet kokiam dimensijų skaičiui. Norint sukonstruoti dviejų dimensijų dvejetainį indeksuotą medį reikia kiekvienoje dvejetainio indeksuoto medžio viršūnėje saugoti kitą dvejetainį indeksuotą medį. Konstravimas dviejų dimensijų dvejetainio indeksuoto medžio yra atliekamas analogiškai kaip ir vienos dimensijos atveju pasirinkus vieną iš jau seniau minėtų konstravimo poskyrio algoritmų, pavyzdžiui pasirinkus paskutinį, pati efektyviausia konstravimo algoritmą, medis yra sukonstruojamas einant per viršūnes viena po kitos (žr. 1.23 pav.).

```
void build2D()
{
    for(int index_x = 1; index_x <= n; ++index_x) {
        for(int index_y = 1; index_y <= m; ++index_y) {
            // Pridedame masyvo reikšmę į indeksuotą medį.
            binarinis_indeksuotas_medis[index_x][index_y] += a[index_x][index_y];
            // Pridedame paskutinį vienetuką esanti dvejetainėje reprezentacijoje prie savęs.
            int naujas_index_x = index_x + (index_x & -index_x);
            // Pridedame paskutinį vienetuką esanti dvejetainėje reprezentacijoje prie savęs.
            int naujas_index_y = index_y + (index_y & -index_y);
            // Patikriname ar neužeiname už ribų.
            if( naujas_index_x <= n && naujas_index_y <= m ) {
                // Pridedame reikiamą medžio reikšmę.
                binarinis_indeksuotas_medis[naujas_index_x][naujas_index_y] +=
                    binarinis_indeksuotas_medis[index_x][index_y];
            }
        }
    }
}
```

1.23 pav. Dviejų dimensijų dvejetainio indeksuoto medžio konstravimo realizacija c++ programavimo kalba

Kaip ir su segmentiniu medžiu dvejetainis indeksuotas medis sudaro galimybes atlikti sumos užklausas pateikiant matricų koordinates tarp kurių norimą surasti stačiakampio sumą, tai yra nurodžius x_1 su y_1 bei x_2 su y_2 . Tačiau, žinant iš senesnių skyrių įprastai dvejetainis indeksuotas gali atsakyti tik į prefiksinės sumos užklausas tai yra nuo pradžios iki tam tikro indekso, standartinio medžio atveju buvo pasinaudota faktu, kad galima paskaičiuoti intervalo sumą apskaičiuojant dvi sumas $[0..l-1]$ ir $[0..r]$, gaunant jų skirtumą, kur $[0..r]$ yra turinys, o $[0..l-1]$ yra atėminys. Dviejų dimensijų atveju taip pat galima tokį reiškinį išsireikšti tačiau jau reikia skaičiuoti nebe dvi sumas, o keturias. Norint paskaičiuoti matricoje ieškomo stačiakampio sumą, kuris turi keturias viršūnes $ABCD$ žymimas iš viršaus į apačią iš kairės į dešinę ir koordinatų ašies pradžios tašką žymimą O nėra sudėtinga parodyti, kad mūsų ieškomą sumą galima išsireikšti pagal formulę $SUM(OB) - SUM(OD) - SUM(OA) + SUM(OC)$ (žr. 1.24 pav.).



1.24 pav. Stačiakampio sumos suradimas išsireikškiant per kelias sumas (Belwariar, 2015)

Sumos užklausa bendrai skiriasi nuo standartinio atveju visai nedaug, tiesiog pasiekus bet kurią viršūnę pagal vieną koordinatę reikia eiti iškart pagal antrąją (žr. 1.25 pav.)

```
int prefixsuma(int x, int y)
{
    int prefixsuma = 0;
    for(; x > 0;)
    {
        for(; y > 0;)
        {
            prefixsuma += medis2D[x][y]; // Didiname sumą
            y -= y & -y; // Išmetame paskutinį vienetuką dvejetainėje reprezentacijoje
        }
        x -= x & -x; // Išmetame paskutinį vienetuką dvejetainėje reprezentacijoje
    }
    return prefixsuma;
}

int suma(int x1, int y1, int x2, int y2) {
    // Paskaičiuojame stačiakampio suma
    return prefixsuma(x2, y2) - prefixsuma(x2, y1) - prefixsuma(x1, y2) + prefixsuma(x1, y1);
}
```

1.25 pav. Dviejų dimensijų dvejetainio indeksuoto medžio sumos užklauskos realizacija c++ programavimo kalba

Atnaujinimo užklauskos dviejų dimensijų dvejetainiame indeksuotame medyje vykdomos analogiškai kaip ir standartiniu atveju, bet kaip ir sumos atveju pasiekus vieną viršūnę reikia eiti iškart pagal antrąją (žr. 1.26 pav.). Pateikta funkcija atnaujiną visus elementus regione nuo (x, y) iki (n, m) , kur čia n, m yra atitinkamai maksimali x koordinatė visos matricos atžvilgiu ir maksimali y koordinatė visos matricos atžvilgiu.

```
void update2D(int x, int y, int reiksme)
{
    for (; x <= n; x += (x & -x))
    {
        for (; y <= m; y += (y & -y)) {
            medis2D[x][y] += reiksme; // Atnaujiname reikšmę medžio x, y viršūnėje
        }
    }
}
```

1.26 pav. Dviejų dimensijų indeksuoto medžio atnaujinimo užklauskos realizacija c++ programavimo kalba

2. Problemos leidžiančios įvertinti intervalinių medžio tipo duomenų struktūrų efektyvumą

Norint įvertinti duomenų struktūras reikia sumodeliuoti sprendžiamą problemą. Viena iš problemų yra sumų užklausų su reikšmių pakeitimais problema. Tegul $\text{suma}(l, r)$ reiškia suma masyvo elementų nuo l -tojo elemento iki r -tojo imtinai, t.y. suma lygi $a[i] + a[i + 1] + a[i + 2] + \dots + a[r - 1] + a[r]$, kur čia i yra $i \geq l$ ir $i \leq r$, o masyvas a yra skaitinių duomenų masyvas. Surasti šios užklausos elementų suma yra pakankamai nesunku pirminiu intuityviu sprendimu: eiti nuo masyvo l -tojo elemento iki r -tojo taip aplankant $r - l + 1$ elementų. Šis sprendimas yra pakankamai geras kai duomenų kiekis yra nedidelis arba užklausų yra labai mažai, tačiau duomenų kiekiui arba užklausų kiekiui išaugus ši užklausa bus vykdoma per ilgai ir galimai netgi išvis nebus įvykdyta. Norint efektyviai atsakyti į šią užklausa galima naudoti prefiksinių sumų masyvą. Kiekviena reikšmė prefiksinių sumų masyve turėtų būti lygi sumai reikšmių originalaus masyvo iki atitinkamos pozicijos, tai yra iki pozicijos k , kuri būtų lygi $\text{suma}(0, k)$. Tuomet, kadangi prefiksinių sumų masyvas turi sumos reikšmę $\text{suma}(0, k)$ galima nesunkiai paskaičiuoti $\text{suma}(l, r)$ per konstantinį laiką $O(1)$ naudojant formulę: $\text{suma}(l, r) = \text{suma}(0, r) - \text{suma}(0, l - 1)$. Šis sprendimas yra pats optimaliausias jeigu reikėtų tik paskaičiuoti intervalines užklausas nekintančio masyvo arba masyvas kistu labai retai, o intervalinių užklausų būtų labai daug. Taip yra todėl, kad kiekvieną kartą atsinaujinus masyvo elementui blogiausiu atveju reikėtų pakeisti $O(n)$ prefiksinių sumos masyvo elementų. Taigi, jeigu užklausos yra dviejų tipų: atnaujinimo ir sumos užklausoje galima naudoti teorinėje dalyje aprašytas intervalines medžio tipo duomenų struktūras šiam uždaviniui išspręsti, kadangi jos sugeba į kiekvieną iš užklausų atsakyti efektyviai, blogiausiu atveju per logaritminį laiką $O(\log_2 n)$.

Intervalinių sumos užklausų problemą galima išspręsti ir matricose ją šiek tiek modifikuodami, tegul $\text{suma}(x1, y1, x2, y2)$ reiškia sumą elementų matricoje nuo $(x1, y1)$ kampo iki $(x2, y2)$ kampo elementų imtinai t.y. suma lygi $a[i][i1] + a[i + 1][i1] + \dots + a[x2 - 1][i1] + a[x2][i1] + a[i][i1 + 1] + a[i + 1][i1 + 1] + \dots + a[x2][i1 + 1] + \dots + a[x2 - 1][y2] + a[x2][y2]$ kur čia $i \geq x1$, $i \leq x2$, $i1 \geq y1$, $i1 \leq y2$, o a yra matrica skaitinių duomenų. Matricoje suma galima surasti tokiais pat būdais kaip ir standartiniu atveju, sumuojant visus elementus iš eilės einant nuo $x1$ -ojo iki $x2$ -ojo tuo pačiu nuo $y1$ -ojo iki $y2$ -ojo. Taip pat sumos užklausas galima valdyti naudojant prefiksines sumas tik jau šiuo atveju naudojant kitokią formulę: $SUM(OB) - SUM(OD) - SUM(OA) + SUM(OC)$ (žr. 1.24 pav.). Kaip ir vienos dimensijos atveju yra susiduriama su tomis pačiomis problemomis, tai yra pirmu atveju efektyvumo problema kai egzistuoja daug užklausų arba yra didelis duomenų kiekis, o antruoju atveju kai yra didelis atnaujinamų užklausų kiekis, dėl to ir bus naudojamos teorinėje darbo dalyje aprašytos duomenų

struktūros, kurios su matricose esančiais stačiakampiais sugeba atlikti užklausas per $\log_2 n * \log_2 m$ laiką, kur čia n yra matricos eilučių skaičius, o m stulpelių skaičius, tą mes jau parodėme ankstesniuose teorinės dalies skyriuose.

3. Įvertinimas skirtingų intervalinių medžio tipo duomenų struktūrų

Kad įvertinti segmentinio medžio ir dvejetainio indeksuoto medžio duomenų struktūras reikia įvertinti kaip jie susidoroja su dideliais duomenimis palyginant jas tarpusavyje ir su baziniais problemų sprendimo atvejais. Tam, kad tai atlikti turi būti sugeneruoti duomenys, bei sukurtos programos leidžiančios įvertinti visų skirtingų sprendimo būdų plusus ir minusus, bei turi būti ištestuota, jog mūsų intervalinės medžio tipo duomenų struktūros skaičiuoja taisyklingus rezultatus. Pagal ankstesniame skyriuje minėtas problemas galima analizuoti duomenų struktūras pagal kelis aspektus:

1) Greičio, nustatyti kaip greitai skirtingi sprendimo būdai gali susidoroti su turimais duomenimis esant vienai ar kitai duomenų imčiai.

1.1) Palyginti konstravimo greitį skirtingų duomenų struktūrų sukūrimui.

1.2) Palyginti intervalinės užklausos atžvilgiu, naudojant skirtingo dydžio intervalus.

1.3) Palyginti taško atnaujinimo atžvilgiu

2) Atminties naudojimo, nustatymo kiek operatyviosios kompiuterio atminties yra užimama naudojant skirtingus sprendimo būdus.

3) Kodo sudėtingumo atžvilgiu

Negana to, iškeltas duomenų struktūras galima palyginti ir universalumo atžvilgiu, tam nereikia duomenų, užtenka apžvelgti kokias skirtingas problemas gali išspręsti viena ar kita duomenų struktūra. Taigi, pagal teorinėje darbo dalyje atliktą intervalinių medžio tipo duomenų struktūrų aprašą yra matoma, kad dvejetainis indeksuotas medis sugeba susidoroti su intervalinių užklausų taškų radimo, paskaičiavimo problemomis, turi galimybę atnaujinti bet kokiame taške esančią reikšmę, bei turi galimybę būti naudojamas keliose dimensijose. Tomis pačiomis savybėmis pasižymi ir segmentinio medžio duomenų struktūra, tačiau turi ir papildomus panaudos atvejus tokius kaip išsaugojimas medžio pakitimo istorijos, taip pat jeigu operaciją nėra galima invertuoti tai dažniausiai galima naudoti segmentinį medį šios operacijos atlikimui, kas nėra įmanoma su dvejetainiu indeksuotu medžiu, kadangi jis naudoja kaupiamuosius duomenų kiekius iš kurių norint išsitraukti informaciją reikia turėti, kad mūsų sprendžiama problema turėtų invertuojamą operaciją kaip pavyzdžiui sumai yra atimtis. Viską ką galima atlikti su dvejetainiu indeksuotu medžiu yra galimybė atlikti su segmentiniu medžiu dėl to galima teigti, jog lyginant šias dvi duomenų struktūras segmentinis medis yra žymiai universalesnis. Kadangi teorinėje dalyje buvo aprašytas programinis kodas galima nesunkiai įvertinti kodo sudėtingumą.

Kaip matoma iš realizuoto kodo dvejetainio indeksuoto medžio atveju jo kiekis yra žymiai mažesnis, tačiau kas vyksta šį kodą vykdant yra pakankamai sunku suprasti nežinant kokia duomenų struktūra yra naudojama. Šio darbo autoriaus nuomonę kodo sudėtingumo atžvilgiu lengviau yra suprasti segmentinio medžio programinį kodą, kadangi jis yra labiau intuityvesnis. Jeigu norima realizuoti kodą jo nesuprantant geresnis sprendimas būtų naudoti dvejetainio indeksuoto medžio kodą, kadangi jis užima mažiau vietos, taip pat jeigu yra supratimas abejais atvejais ką programinis kodas atlieka geriau naudoti mažiau užimanti variantą, kad analizuojant kodą būtų sutaupoma laiko.

Atliekant bet kokius kitus vertinimus norima pabrėžti, kad visi jie buvo atlikti naudojant šia techninę įrangą:

- 1) Intel(R) Core(TM) i5-4200U 2.30 GHz procesorius.
- 2) Windows 7 Ultimate operacinė sistema.
- 3) Operatyvioji atmintis 4GB.
- 4) AMD RADEON HD 8670M vaizdo plokštė.
- 5) Samsung 860 EVO 1TB SATA3 SSD išorinis diskas.

3.1. Duomenų generavimas

Pirmiausia, kad galėtumėme išspręsti mūsų problemas mums reikia susigeneruoti duomenis, kadangi visos sumodeliuotos užduotys tai yra intervalinių sumų užklausų su reikšmių pakeitimais, intervalinių sumų užklausų su reikšmių pakeitimais matricose naudoja skaitinius duomenis galima visus duomenis naudoti tuos pačius, bent jau užpildant problemos startinį masyvą. Tam, kad duomenų struktūrų įvertinimas vyktų sklandžiai visos jos turi būti tiriamos naudojant tuos pačius duomenis, kuriuos nėra sudėtinga sugeneruoti, ko pasekoje yra susigeneruojamas didelis duomenų kiekis, kuris yra pasiskirstęs intervale $[0..100000]$ (žr. 3.1 pav.).

```

int main()
{
    freopen("duomenys.txt", "w", stdout); // Pasirenkame, kad duomenis išvesime į rinkmeną
    const size_t nDuomenuKiekis = 1000000; // Nustatome duomenų kiekį
    std::random_device rd; // Sugeneruojame unikalų startinį tašką.
    std::mt19937_64 eng(rd()); // Naudojame 64-bitų Mersenne Twister 19937 generatoriu.
    // Nustatome pasiskirstymą, kad būtų tarp integer tipo reikšmių.
    std::uniform_int_distribution<int> distr;

    for(size_t i = 0; i < nDuomenuKiekis; ++i)
    {
        // Sugeneruojame skaitinę reikšmę.
        const int skaitmenine_reiksme = distr(eng) % (100001);
        std::cout << skaitmenine_reiksme << '\n'; // Išvedame reikšmę į rinkmeną
    }

    return 0;
}

```

3.1 pav. Startinių duomenų generavimas į tekstinę rinkmeną

Generuoti pradinis duomenis skirtingus nėra jokios reikšmės, kadangi tai neturės jokios įtakos programų veikimui. Visiškai skirtingai yra su sumos užklausų duomenis, kur bus nurodomas užklausų skaičius ir intervalo režiai (žr. 3.1 lentelė).

3.1 lentelė. Galimų užklausų pavyzdžiai

Užklauso pavadinimas	Užklauso struktūra	Paaiškinimas
Sumų užklausa	n l_1, r_1 l_2, r_2 l_n, r_n	Sumų užklausoje n reiškia kiek sumų užklausų bus iš viso, o l_i, r_i reiškia atitinkamai i -tosios sumos užklauso kairįjį ir dešinįjį intervalus tarp kurių norima surasti sumą pradinėje duomenų aibėje.
Sumų užklausa matricose	n $x1_1, y1_1, x2_1, y2_1$ $x1_2, y1_2, x2_2, y2_2$ $x1_n, y1_n, x2_n, y2_n$	Šiuo atveju n reiškia kiek sumų užklausų matricose iš viso yra, o $x1_i, y1_i, x2_i, y2_i$ atitinkamai i -tosios sumos užklauso kairiojo apatinio stačiakampio kampo koordinatas ir dešiniojo viršutinio kampo koordinatas kurio sumą norima surasti pradinėje duomenų aibėje.
Atnaujinimų užklausa	n k_1, v_1 k_2, v_2 k_n, v_n	Atnaujinimų užklausoje n reiškia kiek atnaujinimo užklausų bus iš viso, k_i reiškia i -tojo atnaujinimo pradinio masyvo indeksą, o v_i reikšmę į kurią norima atnaujinti pradinio masyvo k_i elementą.
Atnaujinimų užklausa matricose	n x_1, y_1, v_1 x_2, y_2, v_2 x_n, y_n, v_n	Atnaujinimų užklausoje n reiškia kiek atnaujinimo užklausų matricose bus iš viso, x_i, y_i vadinasi i -tojo atnaujinimo pradinės matricos tašką, o v_i reikšmę į kurią norima atnaujinti matricos x_i, y_i elementą.

Juos galima sugeneruoti skirtingus, kadangi tai turės įtakos galutiniams rezultatams. Vienus duomenis galima sugeneruoti kurių atstumas tarp intervalų yra mažas, kitą dalį kuriame jis yra vidutinis arba kintantis, bei dar sugeneruoti papildomai kelias duomenų aibes kur užklausų skaičius yra didelis, mažas arba vidutinis ir ištirti kaip tai paveikia rezultatus sumų užklausų atvejais. Atnaujinimo užklausų atvejais galima naudoti papildomai ir kintantį duomenų kiekį, kadangi tai paveiktų kai kurių duomenų struktūrų skaičiavimo laiką (žr. 3.2 pav.).

```
void SugeneruotiSumosUzklausuDuom(int nMaksimalusAtstumasTarpIntervalu, pair<int, int> pUzklausos[])
{
    std::random_device rd; // Sugeneruojame unikalų startinį tašką.
    std::mt19937_64 eng(rd()); // Naudojame 64-bitų Mersenne Twister 19937 generatoriu.
    // Nustatome pasiskirtymą, kad būtų tarp integer tipo reikšmių.
    std::uniform_int_distribution<int> distr;
    freopen("duomenys-uzklausos.txt", "w", stdout);
    cout << nMaksimalusUzklausuKiekis << '\n';
    for(int i = 0; i < nMaksimalusUzklausuKiekis; ++i)
    {
        int desinysis_intervalas = -1;
        int kairysis_intervalas = -1;
        // Uztikriname kad naudosime validžius intervalus.
        do {
            int nAtstumas = distr(eng) % (nMaksimalusAtstumasTarpIntervalu + 1);
            desinysis_intervalas = distr(eng) % (nMaksimalusDuomenuKiekis);
            kairysis_intervalas = desinysis_intervalas - nAtstumas;
        }
        while( kairysis_intervalas < 0 );
        pUzklausos[i] = std::make_pair(kairysis_intervalas, desinysis_intervalas);
        cout << kairysis_intervalas << " " << desinysis_intervalas << '\n';
    }
    fclose(stdout);
}
```

3.2 pav. Sumos užklausų duomenų generavimas

3.2. Greičio įvertinimas

3.2.1. Konstravimo greitis

Konstravimo greitis yra vertinamas laiko tarpu per kurį duomenų struktūra yra pilnai sukonstruojama iš pradinių duomenų, kad galėtų operuoti ir atsakinėti į užklausas. Problemas sprendžiant tai dažniausiai yra vienkartinė operacija, jeigu duomenų struktūra efektyviai palaiko atnaujinimo operacijas.

Visų uždavinių atvejais standartinio masyvo konstravimo greitis yra tiesinis, kadangi duomenims didėjant laikas taip pat tiesiškai didėja. Šiuo atveju laiko kompleksiskumas yra tiesinis be jokios papildomos konstantos, kuris gali būti žymimas $O(n)$ vienmačiu atveju, kur čia n yra lygus duomenų kiekiui (žr. 3.3 pav.) ir $O(n * m)$, kai yra kuriama matrica, kur čia n yra eilučių skaičius, o m stulpelių skaičius (žr. 3.4 pav.).

```

// Nurodome maksimalų duomenų kiekį.
const int nMaksimalusDuomenuKiekis = 10000000;
// Pasirenkame, kad duomenis nuskaitysime iš rinkmenos.
freopen("duomenys.txt", "r", stdin);
int dabartine_skaitytine_verte;
int *pDuomenys = new int[nMaksimalusDuomenuKiekis];
// Skaitome reikšmes iki rinkmenos pabaigos
for(int i = 0; cin >> dabartine_skaitytine_verte; ++i) {
    pDuomenys[i] = dabartine_skaitytine_verte;
}

```

3.3 pav. Standartinio masyvo konstravimo realizacija c++ programavimo kalba

```

// Nurodome maksimalų duomenų kiekį.
const int nMaksimalusDuomenuKiekis = 10000000;
// Pasirenkame, kad duomenis nuskaitysime iš rinkmenos.
freopen("duomenys.txt", "r", stdin);
// Patogumo atvejui nusistatome, kad tarkime matricoje bus 10000 stulpelių
int m = 10000;
// Nusistatome, kiek bus stulpelių matricoje pagal eilutes
int n = nMaksimalusDuomenuKiekis / m;
int **pDuomenys = new int*[m];
// Skaitome reikšmes iki rinkmenos pabaigos
for(int i = 0; i < m; ++i)
{
    pDuomenys[i] = new int[n];
    for(int j = 0; j < n; ++j) {
        cin >> pDuomenys[i][j];
    }
}

```

3.4 pav. Matricos konstravimo realizacija c++ programavimo kalba

Konstruojant prefiksinių sumų masyvą arba matricą konstravimo greitis vis dar išlieka tiesinis be jokios papildomos konstantos pridėjimo. Skirtumas realizacijos nuo paprasto masyvo metodo yra tik tas, kad po inicializavimo vieno duomenų elemento yra papildomai inicializuojamas ir prefiksinių sumų elementas pagal taisykles:

1) Masyvo atveju

1.1) Jeigu tai yra pats pirmas duomenų elementas prefiksinės sumos masyvo elementas bus lygus duomenų elementui.

1.2) Visais kitais atvejais prefiksinės sumos masyvo elementas bus lygus duomenų elemento ir prieš tai paskaičiuoto prefiksinių sumos masyvo elemento sumai.

2) Matricos atveju

2.1) Jeigu tai yra pirmasis duomenų elementas prefiksinės sumos matricos elementas bus lygus duomenų elementui.

2.2) Jeigu tai yra pirmosios duomenų matricos eilutės elementas, tuomet prefiksinės sumos masyvo elementas bus lygus duomenų elemento ir prieš tai paskaičiuoto prefiksinio sumos matricos stulpelio elemento sumai.

2.3) Jeigu tai yra pirmosios duomenų matricos stulpelio elementas, tuomet prefiksinės sumos masyvo elementas bus lygus duomenų elemento ir prieš tai paskaičiuoto prefiksinio sumos matricos eilutės elemento sumai.

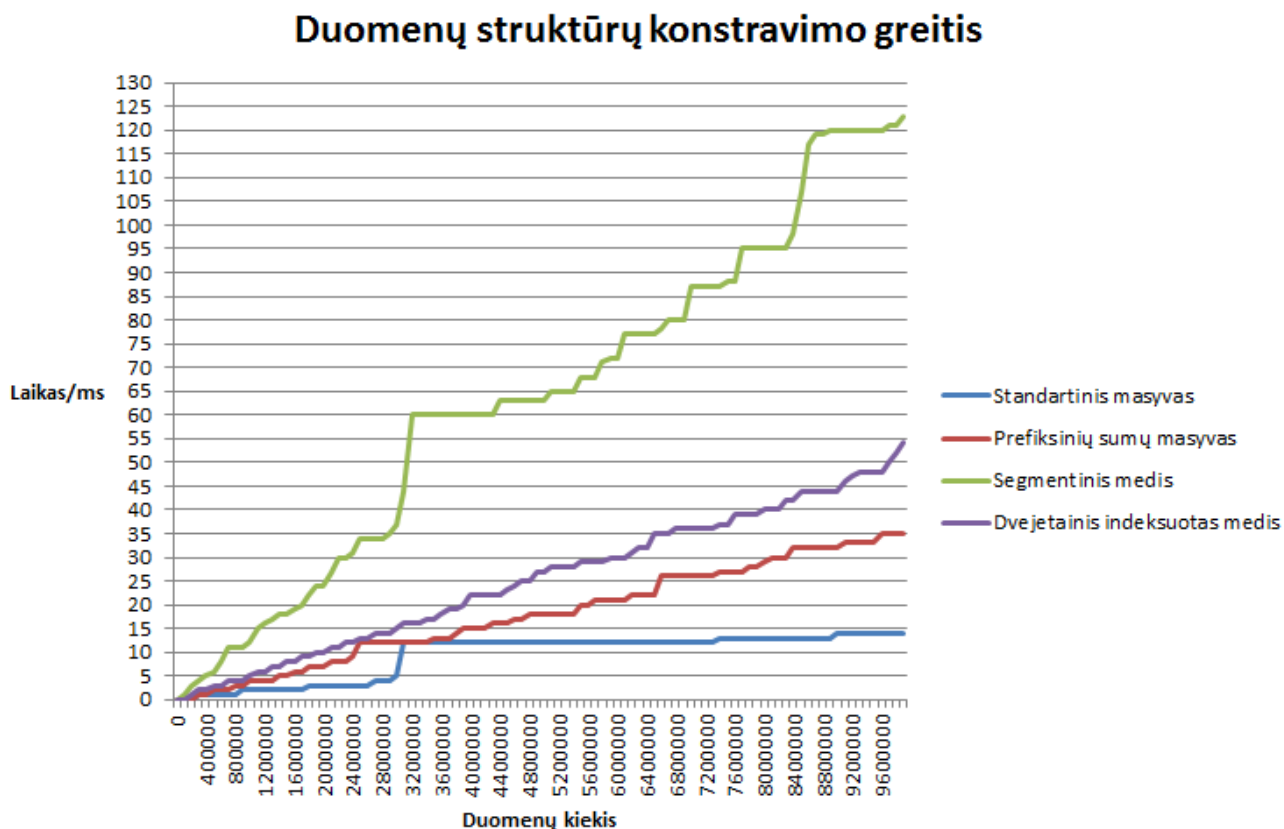
2.4) Visais kitais atvejais prefiksinės sumos matricos elementas yra apskaičiuojamas sumuojant duomenų elementą su prieš tai paskaičiuoto prefiksinio sumos matricos stulpelio elementu, bei pridėdant prieš tai paskaičiuoto prefiksinio sumos matricos eilutės elementą ir iš viso rezultato atimant prieš tai ėjusių paskaičiuoto prefiksinės sumos matricos stulpelio ir eilutės elementą (žr. 3.5 pav.)

```
pref_sumos[i][j] = pref_sumos[i - 1][j] + pref_sumos[i][j - 1]
                  - pref_sumos[i - 1][j - 1] + pDuomenys[i][j];
```

3.5 pav. Prefiksinės sumos konstravimo pseudo kodo formulė

Kaip jau buvo minėta teorinėje darbo dalyje segmentinis medis taip pat turi tiesinį konstravimo laiką, tačiau jis turi konstantą mažesnę negu keturis standartiniu atveju ir konstantą mažesnę negu 16 matricos atveju (žr. 1.2 pav., 1.8 pav., 1.9 pav.). Dvejetainis indeksuotas medis kaip jau gana išsamiai aprašytas teoriniame darbo skyriuje taip turi galimybę būti sukurtas per tiesinį konstravimo laiką be jokios papildomos konstantos tiek standartiniu, tiek matricos atvejais (žr. 1.20 pav., 1.23 pav.).

Kad geriau suprasti kaip skiriasi visų išvardintų skirtingų duomenų struktūrų konstravimo greičiai buvo sukurta programa, kuri nustato kiek laiko yra konstruojama kiekviena struktūra esant skirtingiems duomenų kiekiams (žr. 3.6 pav).

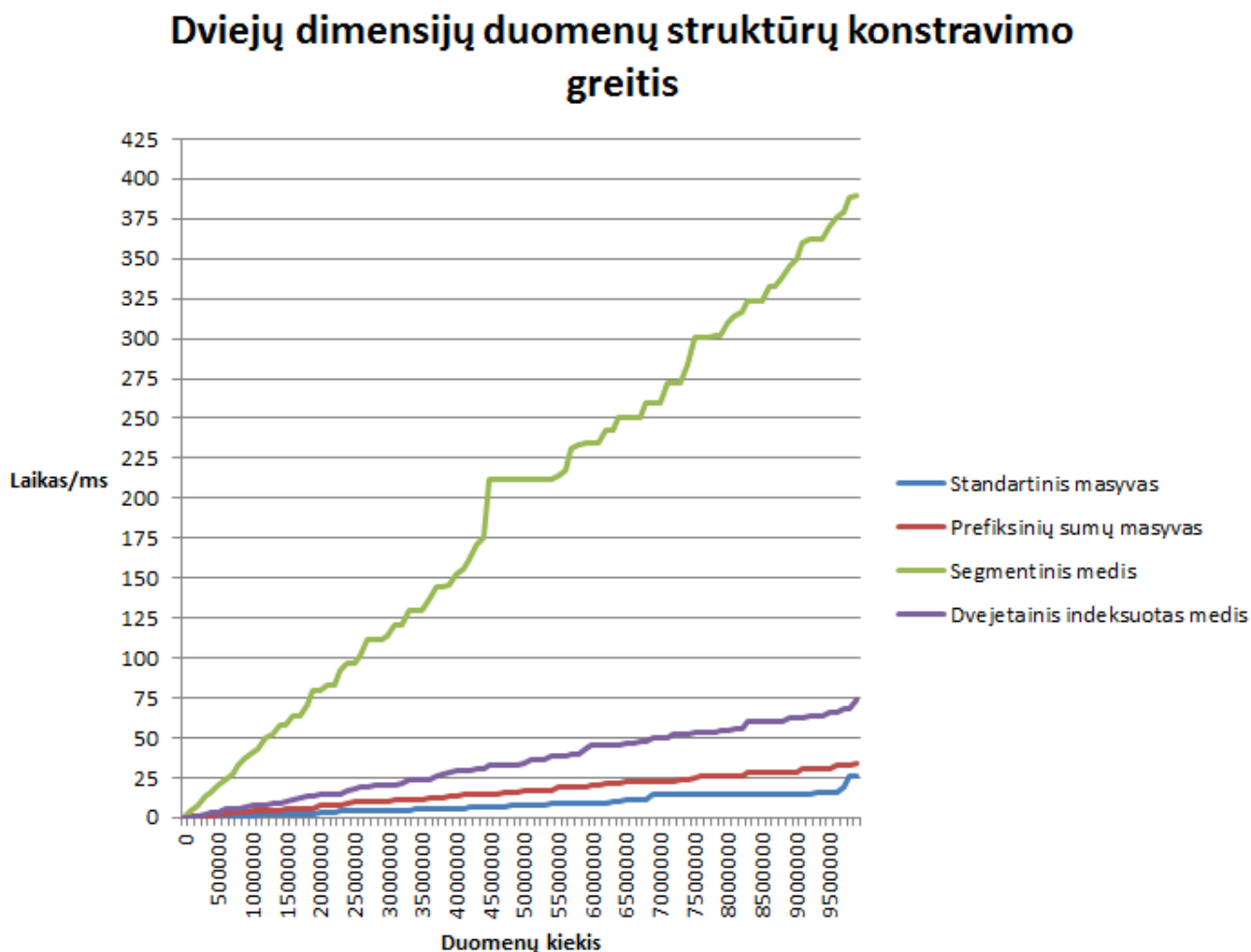


3.6 pav. Duomenų struktūrų konstravimo greičio diagrama

Iš diagramoje pavaizduotų rezultatų yra matoma, kad visoms duomenų struktūroms sukonstruoti nereikia didelio laiko tarpo, ilgiausiai konstravimas užima 123 milisekundes segmentinio medžio atveju, kai yra turima dešimt milijonų duomenų, kas yra mums laiko atžvilgiu labai nedaug ir praktiškai nepastebima. Iš šių rezultatų yra matoma, kad visais atvejais segmentinis medžio konstravimas užėmė daugiausiai laiko, kadangi nors ir turi tiesinį laiko kompleksiskumą, tačiau skirtingai nuo kitų trijų atvejų laiko kompleksiskumas yra su nedidelių konstantų, kuri yra mažesnė už keturis. Visos kitos duomenų struktūros konstravosi panašų laiko tarpą, konstravimosi laikui įtaka tik teturėjo didesnio kiekio operacijų atlikimas dvejetainio indeksuoto medžio atveju ir prefiksinių sumų atveju lyginant su standartiniu masyvu. Segmentinio medžio konstravimo laiko prieaugis lyginant su paprastu masyvu išaugo beveik 9 kartus su prefiksinių sumų masyvu apie 3.5 karto, o su dvejetainiu indeksuotu medžiu beveik 2.5 karto. Galima teigti, jog visos duomenų struktūros susikuria per pakankamai nedidelį laiko tarpą ir jei užduotis būtų sukonstruoti vieną ar

kitą duomenų struktūrą jos pasirinkimas neturėtų labai didelės įtakos, bet be abejo kaip matome iš duomenų paprastasis masyvas šiuo atveju būtų pats efektyviausias sprendimas greičio atžvilgiu, kadangi susikuria greičiausiai, bei tai pats geriausias sprendimas kodo sudėtingumo atžvilgiu, kadangi yra lengviausiai įgyvendinamas iš visų galimų sprendimo variantų.

Panašūs duomenys yra gaunami ir dvimačiu atveju (žr. 3.7 pav.). Matricos grafike išmatuoti konstravimo greičiai buvo gauti bandant kvadratinę matricę eilučių ir stulpelių skaičių išlaikyti tokį patį.



3.7 pav. Dviejų dimensijų duomenų struktūrų konstravimo greičio diagrama

Iš diagramoje pateiktų duomenų matyti, kad matricos atveju duomenų struktūros kūrimas jau šiek tiek ilgesnį laiką, ypač padidėjo segmentinio medžio kūrimo laikas, nes išaugo konstanta iš pozicijos esančios mažiau negu 4 iki mažiau negu 16, ką akivaizdžiai atspindi ir diagrama. Segmentinio medžio kūrimo laikas su tuo pačiu duomenų kiekiu padidėjo beveik 3.5 karto lyginant su prieš tai analizuota problema, dvejetainio indeksuoto medžio atveju prieaugis buvo apie 0.37 karto, prefiksinių sumų atveju prieaugio jokio nebuvo, standartinio masyvo atveju prieaugis buvo dvigubas. Iš to galima teigti, jog šiuo atveju kai yra gana didelis duomenų kiekis ir auga dimensijų

skaičius, duomenų struktūros konstravimas užtrunka ilgiau. Nors ir konstravimo laiko tarpas išaugo, tačiau ilgiausiai besikonstruojanti duomenų struktūra segmentinis medis, kuri užtruko 389 milisekundes, kad sukonstruotu dešimt milijonų duomenų neatbaido nuo jos naudojimo, kadangi tai vis dar yra pakankamai trumpas laiko tarpas.

3.2.2. Intervalinės užklausos greitis

Kaip jau buvo minėta problemų modeliavimo skyriuje sumos intervalinės užklausos atveju standartinis masyvas turi tiesinį laiko kompleksiškumą, kas sukelia didelių sunkumų esant dideliame užklausų kiekiui arba dideliame duomenų kiekiui. Į kiekvieną intervalinę užklausą, kad atsakytų ši duomenų struktūra užima $O(r - l)$ laiko tarpą, kur čia r yra dešinysis intervalo režis, o l kairysis intervalo režis. Prefiksinių sumų atveju kai egzistuoja vien intervalinės užklausos jos yra atliekamos labai greitai, per pastovų laiką (angl. *Constant time*). Kaip jau buvo minėta teorinėje dalyje tiek segmentinis medis, tiek dvejetainis indeksuotas medis intervalines užklausas sugeba įvykdyti per logaritminį laiką $O(\log_2 n)$, kur čia n yra pradinių duomenų kiekis, kas tikrai yra pakankamai įspūdinga.

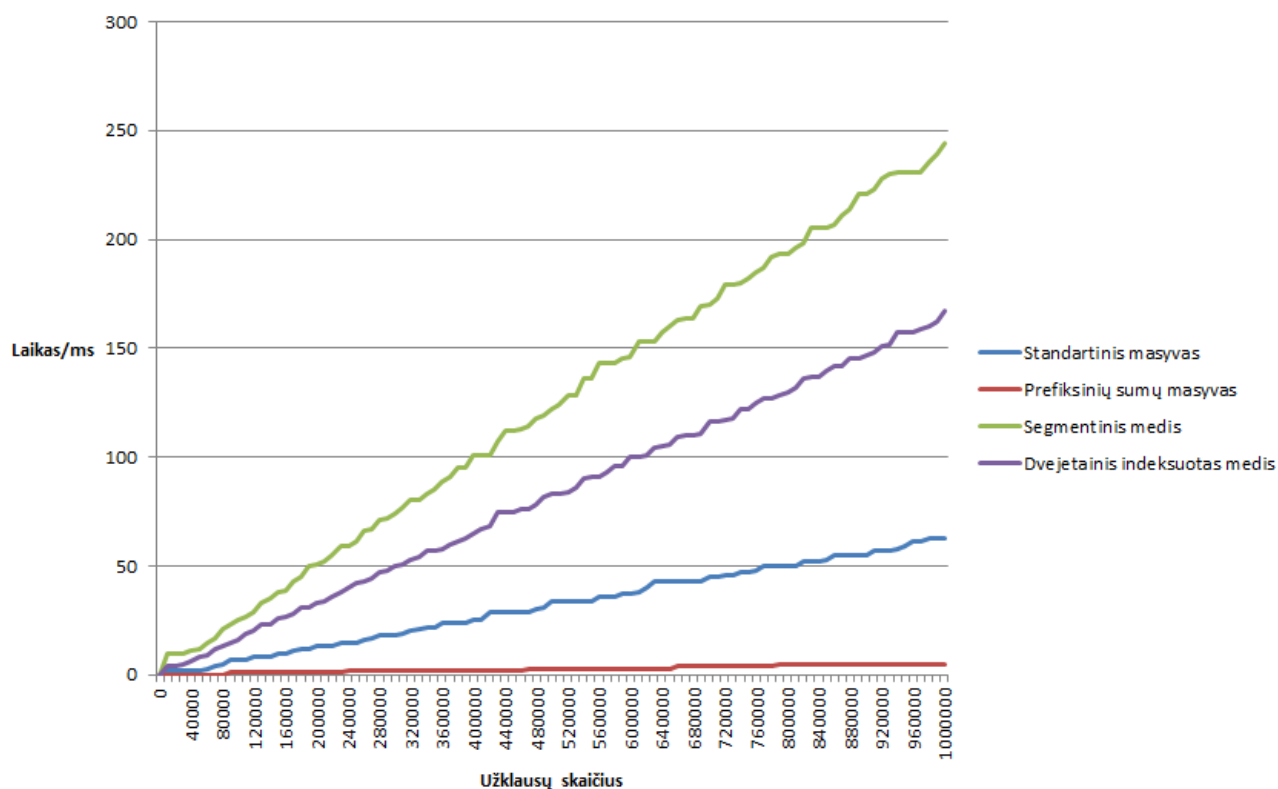
Norint išanalizuoti skirtingų duomenų struktūrų intervalinių užklausų greitį buvo naudojamas kintantis užklausų kiekis su mažais, vidutiniais ir kintančiais nuotoliais tarp intervalų, pradinių duomenų kiekis nebuvo kintantis, kadangi nėra jokio poreikio keisti atstumą tarp intervalų. Naudojamas užklausų formatas jau buvo aptartas duomenų generavimo skyriuje (žr. 3.1 lentelė). Standartinio masyvo ir prefiksinių sumų užklausų realizacijos nėra pateikiamos, kadangi jos jau buvo aprašytos problemų modeliavimo skyriuje ir yra tiesiog per paprastos, kad reikėtų pateikti, o segmentinio medžio ir dvejetainio indeksuoto medžio sumų užklausų realizacijos programinį kodą galima surasti teorinėje dalyje (žr. 1.3 pav., 1.12 pav., 1.21 pav., 1.25 pav.).

Norint suprasti kaip skiriasi visų mūsų išvardintų skirtingų duomenų struktūrų intervalinės užklausos greičiai buvo sukurta programa, kuri nustato kiek laiko užima atsakyti į kintantį sumos užklausų kiekį esant:

- 1) Mažais nuotoliais tarp intervalų, kur čia atstumas yra režiuose $[0..10]$ (žr. 3.8 pav.).
- 2) Vidutiniais nuotoliais tarp intervalų, kur čia atstumas yra režiuose $[0..1000]$ (žr. 3.9 pav.).
- 3) Kintančiais nuotoliais tarp intervalų, kur čia atstumas yra režiuose $[0..10000000]$ (žr. 3.10 pav.).

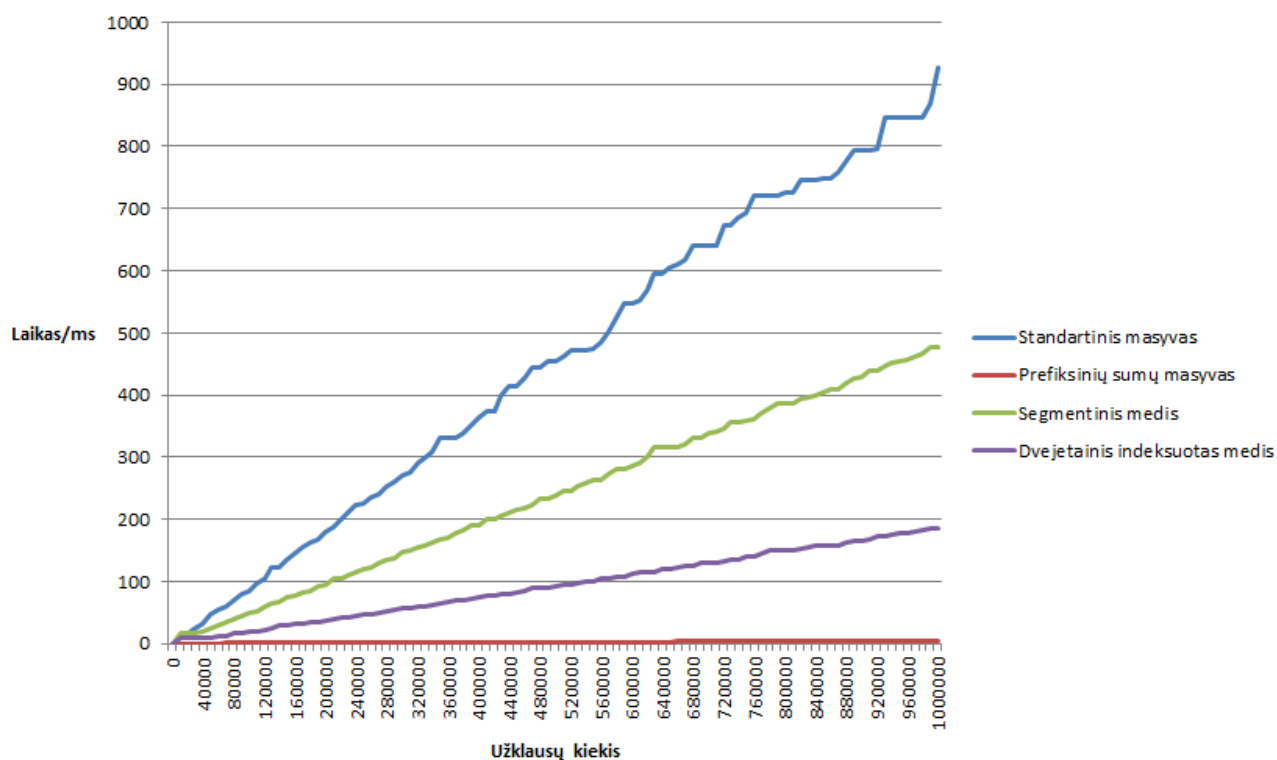
Visais atvejais buvo naudojamas fiksuotas ir tas pats duomenų kiekis sudarytas iš dešimt milijonų įrašų, keitėsi tik užklausų skaičius.

Sumos užklauso greitis esant mažiems atstumams tarp intervalo rėžių



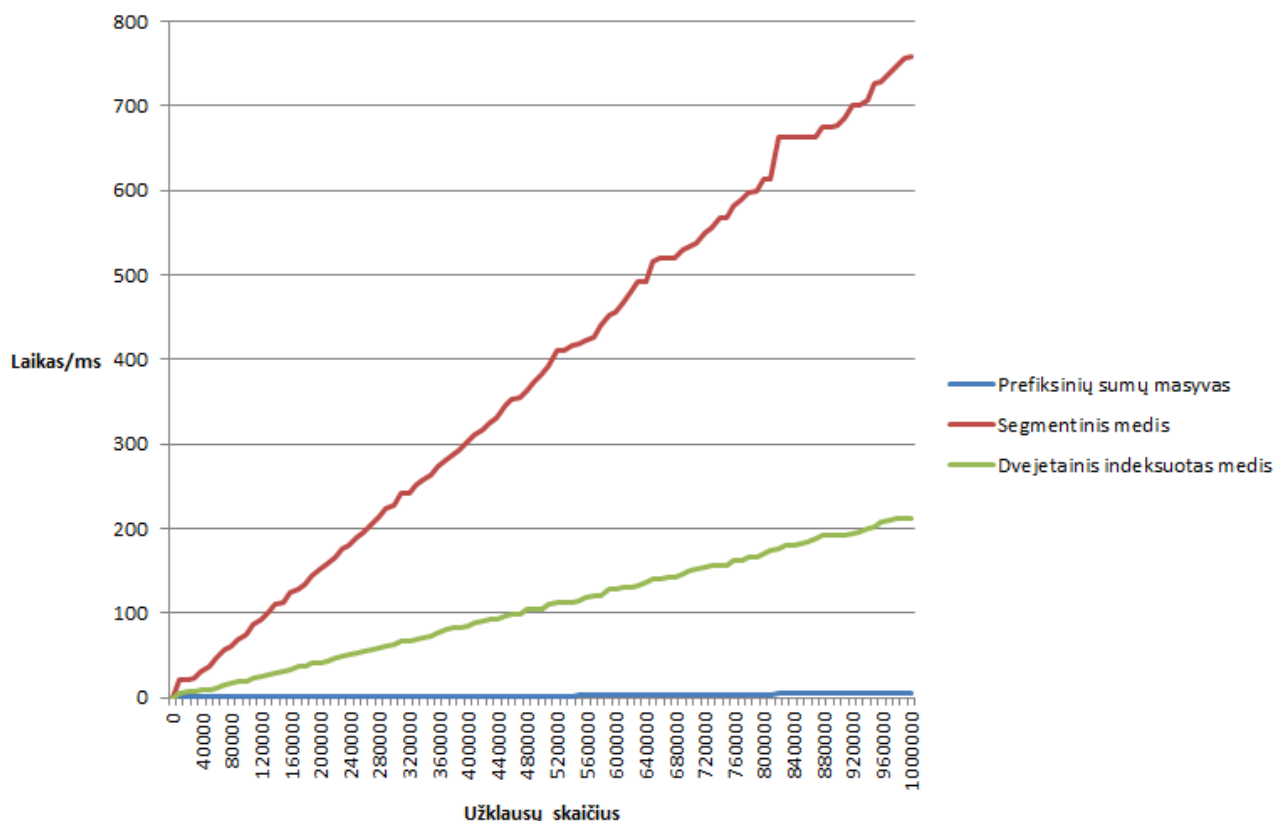
3.8 pav. Sumos užklauso greičio diagrama esant mažiems nuotoliams tarp intervalo rėžių

Sumos užklauso greitis esant vidutiniams atstumams tarp intervalo rėžių



3.9 pav. Sumos užklauso diagrama esant vidutiniams nuotoliams tarp intervalo rėžių

Sumos užklausos greitis esant alternuojantiems/dideliems atstumams tarp intervalo rėžių



3.10 pav. Sumos užklausų diagrama esant alternuojantiems/dideliems nuotoliams tarp intervalo rėžių

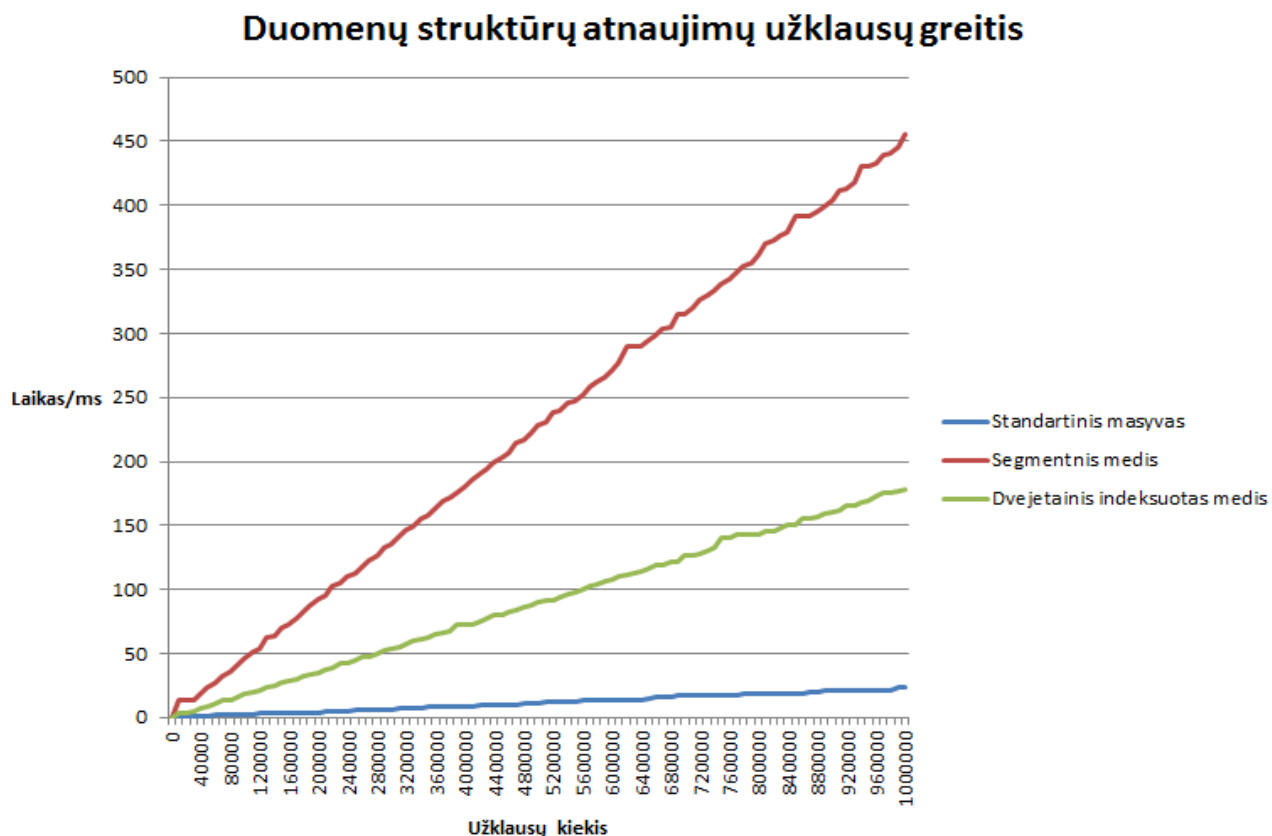
Iš diagramoje pavaizduotų rezultatų yra matoma, kad augant skirtumui tarp intervalo rėžių standartinis masyvas pradeda užimti vis daugiau laiko. Taip yra dėl to, kad jam reikia daug skaičiuojamosios galios norint paskaičiuoti ieškomą sumą esant dideliems nuotoliams. Paskutinėje diagramoje matome, jog standartinis masyvas išvis nėra pavaizduotas diagramoje, kadangi kai užklausų skaičius siekė 10000 standartinis masyvas skaičiavo jau 52 sekundes, kas jau yra labai didelis laiko tarpas, dėl to nebuvo verta jo įtraukinėti į diagramą ir iš to yra matoma, kad pilnai galima atmesti šį variantą kaip tinkamą sprendimą mūsų problemai. Kaip ir buvo tikėtasi iš rezultatų yra matoma, jog geriausiai pasirodė prefiksinių sumų masyvas, nes jis į kiekvieną užklausą gali atsakyti pastoviu laiku $O(1)$. Dar vienas iš pastebėjimų buvo tai, jog augant atstumui tarp intervalo rėžių segmentinio medžio algoritmo laiko kompleksiskumas taip pat išaugo, kas leidžia manyti, jog segmentinis medis yra priklausomas nuo nuotolio tarp intervalų žymiai labiau negu dvejetainis indeksuotas medis. Lyginant tarpusavyje segmentinį ir dvejetainį indeksuotą medį matosi, jog dvejetainis indeksuotas medis veikia greičiau, kai yra naudojamos vien sumos užklausos dėl to, kad segmentinio medžio sumos užklausa nors ir turi logaritminį laiko kompleksiskumą, tačiau turi ir nedidelę konstantą, kuri leidžia dvejetainiam indeksuotam medžiui būti efektyvesniu.

Problema su matricoje skaičiuojamomis sumos užklausomis taip pat buvo tiriama pagal tuos pačius faktorius ir rezultatai praktiškai sutapo su standartiniu atveju, be abejo skirtumas buvo toks, kad laikai proporcingai išaugo dėl konstantos padidėjimo, kaip ir konstravimo atveju.

Iš viso šio skyriaus galima padaryti išvadą, jog jeigu mums reikalingos tiksliai sumos užklausos tai kaip efektyviausią duomenų struktūrą galima naudoti prefiksinių sumų masyvą, kadangi jo atsakymo laikas yra pats geriausias.

3.2.3. Atnaujinimo užklausos greitis su intervalinėmis užklausomis

Atnaujinimo užklausos greitis yra greitis parodantis kiek užtrunka atlikti atnaujinimą duomenų struktūroje norint toliau atsakinėti į intervalines užklausas, kai yra pakeičiamas pradinis duomenų i -tasis elementas. Dažniausiai atnaujinimo užklausos neatliekamos kaip vienintelės užklausos problemoje, o kartu yra susiejamos su kitomis intervalinėmis užklausomis. Nagrinėjamoju atveju atnaujinimo užklausos yra siejamos su sumos užklausomis intervaluose, kur čia atnaujinimas reiškia pakeitimą i -tojo elemento iš reikšmės v į reikšmę v' . Pirmiausia galima apžvelgti kaip su užduotimi yra susidorojama kai egzistuoja tiksliai atnaujinimo užklausos (žr. 3.11 pav.), kai visų atnaujinimo užklausų vykdymo metu pradinis duomenų kiekis išlieka toks pat.



3.11 pav. Duomenų struktūrų atnaujinimo užklausų greitis

Iš diagramoje pavaizduotų rezultatų yra matoma, jog prefiksinio sumų duomenų struktūra netgi nebuvo įtraukta į skaičiavimus, kadangi jos atveju laikas jau esant 10000 užklausų yra ilgesnis už 5000 milisekundžių. Tai pat yra matoma, jog standartinis masyvas kai egzistuoja tik atnaujinimo užklausos greitai veikia, tačiau kaip jau aptarėme prieš tai esančiame skyriuje kai reikia atsakinėti į didelį sumos užklausų kiekį rezultatai nebėra tokie geri. Šis atnaujinimo užklausų palyginimas leidžia suprasti, jog vienintelės duomenų struktūros kurios gali efektyviai išspręsti apibrėžtas problemas, atsakyti į atnaujinimo ir sumos užklausas yra apibrėžtos intervalinės medžio tipo duomenų struktūros: segmentinis medis ir dvejetainis indeksuotas medis. Kaip yra matoma iš atnaujinimo užklausų grafiko čia vėl, kaip ir sumos užklausų atveju yra greitesnis dvejetainis indeksuotas medis, netgi daugiau negu du kartus. Tiek atnaujinimo, tiek sumos užklausas abidvi intervalinės medžio tipo duomenų struktūros atsako per logaritminį laiką, vienintelis dvejetainio indeksuotojo medžio pranašumas yra tas, jog jis neturi jokios konstantos, kas leidžia jam ir užimti geresnę poziciją efektyvumo atžvilgiu sprendžiant sumodeliuota užduotį. Panašūs rezultatai egzistuoja ir matricos atveju, skirtumas yra tik tas, kad segmentinio medžio konstanta dar labiau išauga ir atsiskyrimas nuo dvejetainio indeksuoto medžio tampa dar didesnis. Be abejo nors ir segmentinis medis šią problemą sprendžiant nėra toks efektyvus kaip dvejetainis indeksuotas medis jį vis tiek galima naudoti, kadangi jo užklausų atsakymo laikas vis tiek yra pakenčiamas, kadangi neviršija didelių ribų.

3.3. Atminties naudojimo įvertinimas

Atminties naudojimas yra vertinamas pagal tai kiek kiekviena duomenų struktūra naudoja operatyvios atmintis sukonstravus. Užklausų metu atminties tyrimo daryti nereikia, kadangi mūsų problemas sprendžiant užklausų metų papildomai atminties mums nereikia išsiskirti. Yra žinoma, kad standartinis masyvas ir prefiksinių sumų masyvas naudoja tiesinį atminties kiekį: $O(n)$. Segmentinis masyvas taip pat naudoja tiesinį atminties kiekį, tačiau su konstanta 4 vienmačiu atveju ir atitinkamai konstanta 16 dvimačiu atveju, jeigu būtų trimatis atvejis jis turėtų konstanta dar didesne, kuri būtų lygi 64. Pavyzdžiui, kadangi yra dirbama su skaitiniais duomenimis C++ programavimo kalbos aplinkoje operatyvios atminties naudojimą nėra sudėtinga apskaičiuoti, jis gali būti apskaičiuojamas baitais pagal formulę: $n * 4$, kur čia n yra pradinių duomenų kiekis, o 4 yra standartinio vieno C++ skaitinio tipo dydis baitais. Iš to galima teigti, jog standartinis masyvas, prefiksinių sumų masyvas ir dvejetainis indeksuotas medis naudoja $n * 4$ baitus sprendžiant problemą standartiniu atveju, o segmentinis medis sprendžiant šią problemą naudoja $n * 4 * 4$ baitų, dėl papildomos konstantos. Dvejetainis indeksuotas medis kaip ir paprastosios duomenų struktūros naudoja tiesinį atminties kiekį, be jokios papildomos konstantos tiek dvimačiu, tiek standartiniu

atveju. Tai leidžia mums daryti prielaidą, kad ir atminties naudojimo atveju geriausia duomenų struktūra kuria būtų galima naudoti yra dvejetainis indeksuotas medis, jeigu reikia taupyti atmintį.

3.4. Testavimas

Kad užtikrinti, jog visos duomenų struktūros buvo įvertintos teisingai, tai yra skaičiavo teisingus rezultatus buvo atliktas programinio kodo testavimas. Testavimo metu buvo ištestuotos visų duomenų struktūrų operacijos: konstravimas, sumos užklausa, atnaujinimo užklausa. Testavimas buvo atliekamas tokiu būdu:

- 1) Sukonstruojamos visos duomenų struktūros iš pradinių duomenų
- 2) Vykdomos sumos užklauskos visoms duomenų struktūroms
- 3) Sumos užklauskų rezultatai palyginami tarpusavyje ar visi sutampa
- 4) Vėl vykdomas antras punktas kol egzistuoja užklauskos

Tokiu principu yra ištestuojama ar teisingai pas mus veikia konstravimas ir sumos užklauskos, kadangi sulyginus visus rezultatus atsakymas turėtų gauti toks pat nepriklausomai nuo laiko kiek skaičiavo (žr. 3.12 pav.).

```
void TestavimasArRezultataiSutampa(int pDuomenys[], pair<int, int> pUzklausos[])
{
    std::pair<int *, long long *> bazines_strukturos = SukonstruojameVisasDumStrukturas(pDuomenys);
    int *pMasyvoDuom = bazines_strukturos.first;
    long long *pPrefixSumMasyvas = bazines_strukturos.second;
    for(int i = 0; i < nMaksimalusUzklausuKiekis; ++i)
    {
        long long suma1 = SumosUzklausaPaprastoMasyvo(pUzklausos[i].first, pUzklausos[i].second,
                                                         pMasyvoDuom);
        long long suma2 = SumosUzklausaPrefSumMasyvo(pUzklausos[i].first, pUzklausos[i].second,
                                                         pPrefixSumMasyvas);
        long long suma3 = SumosUzklausaDvejIndeksuotMedzio(pUzklausos[i].first + 1,
                                                             pUzklausos[i].second + 1);
        long long suma4 = SumosUzklausaSegmentTree(0, 0, nMaksimalusDuomenuKiekis - 1,
                                                         pUzklausos[i].first,
                                                         pUzklausos[i].second);
        if( suma1 != suma2 || suma3 != suma4 || suma1 != suma3 )
        {
            cout << "Sumos nera vienodos kazkas blogai skaiciuojasi\n";
            cout << "Paprasto masyvo suma:" << suma1 << '\n';
            cout << "Prefiksinio masyvo suma:" << suma2 << '\n';
            cout << "Dvejetainio indeksuoto medzio suma:" << suma3 << '\n';
            cout << "Segmentinio medzio suma:" << suma4 << '\n';
        }
    }
}
```

3.12 pav. Duomenų struktūrų įvertinimo testavimo realizacija c++ programavimo kalba

Atnaujinimo užklausa yra testuojama panašiu principu, pirmiausia yra įvykdoma atnaujinimo užklausa ir iškart po jos seka sumos užklausa, sumos užklauskos rezultatas yra lyginamas tarpusavyje tarp visų duomenų struktūrų. Šis atliktas testavimas mums leidžia teigti, jog mūsų duomenų struktūros buvo įvertintos korektiškai.

IŠVADOS

1) Atlikus mokslinės literatūros apžvalgą pastebėta, kad dažniausiai taikomos intervalinės duomenų struktūros darbui su dideliais duomenų kiekiais yra segmentinis medis bei dvejetainis indeksuotas medis. Pagal šių struktūrų aprašą buvo nustatyta, kokias problemas galėtų išspręsti abi intervalinės medžio tipo duomenų struktūros. Taip pat sumodeliuotos intervalinių sumos užklausų bei intervalinių sumos užklausų matricose problemos, kurios leido įvertinti abiejų duomenų struktūrų efektyvumą dirbant su dideliais duomenų kiekiais.

2) Naudojantis C++ programavimo kalbos technologijomis buvo sėkmingai realizuoti programiniai moduliai, leidžiantys įvertinti aptartas duomenų struktūras. Šių programinių modulių taisyklingą veikimą užtikrino atliekamų operacijų funkcinis testavimas. Tai leido įsitikinti, jog moduliai veikia korektiškai.

3) Atlikus tyrimą išryškinti greičio, atminties naudojimo, universalumo ir kodo sudėtingumo požymiai. Pagal nustatytus požymius buvo įvertintos duomenų struktūros. Išsiaiškinta, kad sumos užklausų su reikšmių pakeitimais atveju dvejetainis indeksuotas medis yra 2.5 karto greitesnis ir naudoja 4 kartus mažiau atminties nei segmentinis medis. Todėl dvejetainis indeksuotas medis yra tinkamesnė duomenų struktūra sprendžiant šiame darbe sumodeliuotas problemas. Tačiau pagal atliktą tyrimą teorinėje darbo dalyje buvo nustatyta, jog segmentinis medis turi universalesnių operacijų ir galimybių spektrą. Tai leidžia šiai duomenų struktūrai apimti didesnę uždavinių diapazoną, t. y. turėti didesnę galimybių kiekį.

Literatūros sąrašas

1. Berg, M.D., Kreveld, M.V., Overmars, M. and Schwarzkopf, O. (2000) Computational geometry. *Computational geometry: algorithms and applications*(2nd edition). Niujorkas: Springer [žiūrėta 2019m. lapkričio 22d.].
2. *Segment Tree* [interaktyvus] 2018 [žiūrėta 2019 m. lapkričio 22d.]. Prieiga per internetą: https://cp-algorithms.com/data_structures/segment_tree.html
3. *Segment tree: Construction* [interaktyvus] 2016 [žiūrėta 2019 m. lapkričio 25d.]. Prieiga per internetą: https://wcipeg.com/wiki/Segment_tree
4. Ibtehaz, N., Kaykobad, M. and Rahman, M.S. (2018) *Multidimensional segment trees can do range queries and updates in logarithmic time* [interaktyvus] [žiūrėta 2019m. lapkričio 26d.]. Prieiga per internetą: <https://arxiv.org/abs/1811.01226>
5. Sharma, A. (2016) *Segment Trees* [interaktyvus] [žiūrėta 2019m. lapkričio 26d.]. Prieiga per internetą: <https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/>
6. Mittal, A. (2016) *Lazy Propagation in Segment Tree* [interaktyvus] [žiūrėta 2020 sausio 3d.]. Prieiga per internetą: <https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/>
7. Aggrawal, Y. (2017) *2D Segment Tree* [interaktyvus] [žiūrėta 2020 sausio 26d.]. Prieiga per internetą: <https://iq.opengenus.org/2d-segment-tree/>
8. Laaksonen, A. (2017) *Guide to Competitive programming: Learning and Improving Algorithms Through Contests*. Helsinkis: Springer[žiūrėta 2020 m. vasario 21d.].
9. Laber, E. and Molinaro, M. (2009) *An Approximation algorithm for Binary Searching in Trees*. *Algorithmica* 59, 601–620(2011) [interaktyvus] [žiūrėta 2020 m. kovo 1 d.]. Prieiga per internetą: <https://link.springer.com/article/10.1007/s00453-009-9325-0>
10. Clampett, H.A (1964) *Randomized binary searching with tree structures*. [interaktyvus] [žiūrėta 2020 kovo 2 d.]. Prieiga per internetą: <https://dl.acm.org/doi/abs/10.1145/363958.363987>
11. Kumar, N. (2016) *Persistent segment tree | Set 1 (Introduction)* [interaktyvus] [žiūrėta 2020 kovo 3 d.]. Prieiga per internetą: <https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction/>
12. Fenwick, M.P. (1994) *Software: Practice and Experience: A new data structure for cumulative frequency tables*. Auklandas: John Wiley & Sons, Ltd. [Žiūrėta 2020 kovo 4d.].

13. Gopani, A. (2019) *Understanding Fenwick tree (Binary Indexed Tree) with Range Product* [interaktyvus] [žiūrėta 2020 kovo 4 d.] Prieiga per internetą: <https://iq.opengenus.org/fenwick-tree-range-product/>
14. Nazarathy, Y. (2017) *Adding the powers of 2* [interaktyvus] [žiūrėta 2020 kovo 5 d.] Prieiga per internetą: <https://www.1onepsilon.com/single-post/2017/02/17/Adding-the-Powers-of-2>
15. Ragaišis, S. (2001) Informatikos įvadas: *Skaičiavimo sistemos* [žiūrėta 2020 kovo 6 d.] Prieiga per internetą: <https://klevas.mif.vu.lt/~ragaisis/InfIvadas/SkSistemas.htm>
16. *Binary indexed trees* 2014 [interaktyvus] [žiūrėta 2020 kovo 6 d.]. Prieiga per internetą: <https://www.topcoder.com/community/competitive-programming/tutorials/binary-indexed-trees/>
17. *Binary Indexed Tree or Fenwick Tree* 2014 [interaktyvus] [žiūrėta 2020 kovo 8 d.]. Prieiga per internetą: <https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/>
18. *Binary Indexed Trees* 2019 [interaktyvus] [žiūrėta 2020 kovo 9 d.]. Prieiga per internetą: <https://algorithmsandme.com/binary-indexed-trees/>
19. Fenwick, M.P (1993) *A new data structure for Cumulative Probability Tables* [žiūrėta 2020 kovo 10 d.]. Prieiga per internetą: <https://www.cs.auckland.ac.nz/~peter-f/FTPfiles/TechRep88.pdf>
20. Belwariar, R. (2015) *Two Dimensional Binary Indexed Tree or Fenwick tree* [interaktyvus] [žiūrėta 2020 kovo 20 d.]. Prieiga per internetą: <https://www.geeksforgeeks.org/two-dimensional-binary-indexed-tree-or-fenwick-tree/>