Tampereen yliopisto

Lasse Vuorinen

C++:N JA GOLANGIN TIETORAKENTEIDEN TEHOKKUUKSIEN VERTAILU

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Toukokuu 2022

TIIVISTELMÄ

Lasse Vuorinen: C++:n ja Golangin tietorakenteiden tehokkuuksien vertailu
Kandidaatintutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikka
Toukokuu 2022

Tehokkaiden ohjelmistojen valmistamisessa keskeinen osa on käyttää tehokkaita ja kyseiseen käyttötarkoitukseen mahdollisimman hyvin soveltuvia tietorakenteita. C++ ja Golang ovat molemmat todella nopeita käännettäviä ohjelmointikieliä. Molemmat ohjelmointikielet sisältävät hyvin samankaltaisia perustietorakenteita. Tässä työssä vertailtiin C++:n vektoritietorakenteen tehokkuutta Golangin viipaletietorakenteen tehokkuuteen ja C++:n sanakirjatietorakenteen tehokkuutta Golangin sanakirjatietorakenteen tehokkuuteen. Tavoitteena oli selvittää, kumman ohjelmointikielen kyseiset tietorakenteet ovat tehokkaampia ja miten tietorakenteiden sisältämien alkioiden määrä vaikuttaa niiden tehokkuuteen. Tutkimus toteutettiin käyttämällä kirjallisia lähteitä ja suorittamalla itse vertailukokeita.

Työssä on kaksi osaa. Ensimmäisessä osassa on kirjallisuuden pohjalta esitetty tietorakenteiden tekniset toteutukset ja niiden ominaisuudet. Lisäksi siinä on kerrottu asymptoottisesta tehokkuudesta ja sen erisuuruisista tehokkuuskäyristä. Toisessa osassa toteutettiin tietorakenteiden tehokkuuksia vertailevia käytännön kokeita. Siinä mitattiin alkioiden lisäämistä, poistamista ja hakemista tietorakenteista.

Tutkimuksen tuloksena Golangin viipale osoittautui yksittäisiä alkioita koskevissa operaatioissa C++:n vektoria nopeammaksi, mutta jos operaatio vaati tietorakenteen läpikäymistä tai uudelleenallokointia, niin C++:n vektori oli nopeampi. Sanakirjoja koskevissa kokeissa C++:n järjestämätön sanakirja oli kaikista nopein lähes kaikissa kokeissa. Golangin sanakirja ja C++:n järjestetty sanakirja olivat sitä nopeampia vain usean alkion lisäämisessä kahdella suurimmalla syötemäärällä ja alkion poistamisessa arvon avulla kahdella suurimmalla syötemäärällä. Golangin sanakirja oli yksittäisiä alkioita koskevissa kokeissa C++:n järjestettyä sanakirjaa nopeampi. C++:n järjestetty sanakirja puolestaan oli Golangin sanakirjaa nopeampi usean alkion lisäämisessä ja alkion poistamisessa arvon perusteella pienimmällä alkiomäärällä.

Avainsanat: C++, Golang, tietorakenne, tehokkuus, taulukko, vektori, sanakirja, viipale

SISÄLLYSLUETTELO

1. JOHDANTO

Nykyään useissa teknisissä laitteissa vasteajat ovat pieniä tai käsiteltävät datamäärät suuria. Näin ollen kyseisten laitteiden tarvitsee olla nopeita ja tehokkaita. Se, että laitteiden komponentit ovat nopeita, ei pelkästään riitä vaan myös suoritettavien ohjelmistojen on oltava tehokkaita. Ohjelmistojen tehokkuuteen vaikuttaa moni asia, kuten esimerkiksi ohjelmointikieli ja ohjelmoijan tekemät ratkaisut.

Tässä työssä käsitellään C++- ja Go -ohjelmointikieliä. C++ on vanha käännettävä ja staattisesti tyypitetty ohjelmointikieli, joka kehitettiin 1980-luvulla. Se on kehitetty C-kielestä lisäämällä siihen olio-ohjelmointi ja muita ominaisuuksia. C++ on ollut suosittu ohjelmointikieli koko sen eliniän ja on sitä vielä nykyäänkin. Se oli edelleen vuonna 2021 kymmenen käytetyimmän ohjelmointikielen joukossa, eikä sen käytölle ole näkyvissä loppua. [1]

Golang eli lyhyemmin Go on sen sijaan kohtuullisen uusi, vuonna 2009 Googlen kehittämä käännettävä ja staattisesti tyypitetty ohjelmointikieli. Googlen tarkoituksena oli kehittää Gosta nopea ja samalla syntaksiltaan yksinkertainen ohjelmointikieli. Go ei ole vielä kovin käytetty ohjelmointikieli, mutta se yleistyy käytössä koko ajan yhä enemmän [2].

Molemmat ohjelmointikielet ovat ominaisuuksiltaan tehokkaita. Tehokkuuden kannalta ohjelmointikielissä tärkeää on muun muassa tehokkaat tietorakenteet. Tämä työ tutkii ja vertailee kyseisten ohjelmistokielten perustietorakenteita ja niiden tehokkuuksia. Tietorakenteiden tehokkuutta voidaan mitata monella eri tavalla, esimerkiksi muistin kannalta tai toimintoihin kuluneen ajan kannalta.

Tässä työssä tehokkuutta mitataan suoritusaikojen perusteella. Tutkimus toteutetaan käyttäen aiheeseen liittyviä kirjallisia lähteitä ja lisäksi vertaillaan tietorakenteiden tehokkuuksia mittaustyön avulla. Kokeissa tietorakenteisiin lisätään, poistetaan ja haetaan tietoalkioita. Kyseisten toimenpiteiden suoritusaikoja vertaillaan tämän jälkeen ohjelmointikielten kesken.

Luvussa kaksi perehdytään tietorakenteiden teknisiin toteutuksiin ja niiden eroavaisuuksiin ohjelmointikielten kesken. Luvussa kolme esitetään kokeiden rakenne, suoritustapa sekä parametrinä toimivan tietoalkioita sisältävän testidatan sisältö. Luvussa neljä toteutetaan varsinaiset kokeet. Luvussa viisi analysoidaan saatuja tuloksia ja vertaillaan niitä molempien ohjelmointikielien kesken.

2. TAUSTA

C++ ja Go sisältävät useita moniin eri käyttötarkoituksiin kehitettyjä tietorakenteita, jotka voivat erota toisistaan hyvinkin paljon. Siitä huolimatta molemmat ohjelmointikielet sisältävät keskenään hyvin samankaltaiset perustietorakenteet. Tässä työssä perehdytään C++:n taulukko-, vektori- sanakirja- ja järjestämätön sanakirjatietorakenteisiin (array, vector, map, unordered map). Lisäksi perehdytään Gon taulukko-, viipale- ja sanakirjatietorakenteisiin (array, slice, map). Ohjelmointikielten taulukot vastaavat toisiaan ja C++:n vektori vastaa Gon viipaletta. Kaikkia sanakirjoja vertaillaan keskenään.

2.1 Taulukot

Taulukot ovat listamaisia tietorakenteita, joita esiintyy useissa ohjelmointikielissä. Tässä tutkielmassa toteutettavissa kokeissa ei käsitellä taulukoita, mutta ne ovat tärkeitä tietorakenteita sekä C++:ssa että Gossa.

2.1.1 C++ -taulukko

C++:ssa on kahdenlaisia kiinteäkokoisia taulukoita. Toinen niistä on suoraan C-ohjelmointikielen taulukko ja toinen on standardikirjaston taulukko (std::array). Standardikirjaston taulukko on C-tyylisestä taulukosta muodostettu luokka, joka sisältää lisämetodeja.

Taulukko itsessään on kiinteäkokoinen sekvenssisäiliö, joka varastoi kaikki elementit yhteen osaan muistia [3]. Jokaisen taulukon alkion on oltava samaa tietotyyppiä ja kaikki taulukon alkiot käyttävät keskenään saman verran muistia. Siihen ei voi lisätä, eikä siitä voi poistaa alkioita. Yhden alkion varaaman muistialueen pituutta merkitään merkinnällä sizeof(type), jossa ”type” on taulukon alkioiden tyyppi. Ensimmäisen alkion muistiosoite on Base Address (BA). Näistä kahdesta tiedosta muodostetaan lauseke $BA + i * sizeof(type)$, jonka avulla päästään käsiksi taulukon jokaisen alkion muistiosoitteeseen alkion indeksillä i. Taulukko sijaitsee aina muistin pinossa. [4]

C-tyylinen taulukko on toimiva, mutta se sisältää muutamia rajoituksia, joiden takia se ei ole välttämättä paras mahdollinen ratkaisu. Tärkeimpiä rajoituksia ovat seuraavat:

- Muistin varaaminen ja vapauttaminen tulee suorittaa manuaalisesti. Muistin vapauttamisessa tapahtuva virhe voi aiheuttaa muistivuodon, jos muistiosoite katoaa.

- Operaattori [] ei tarkista, onko indeksi suurempi tai yhtä suuri kuin taulukon koko. Väärin käytettynä tämä voi johtaa segmentointivirheisiin tai muistin vioittumiseen.
- Sisäkkäisten taulukoiden syntaksi monimutkaistuu ja vaikeuttaa koodin luettavuutta.
- Taulukon syväkopiointia ei ole toteutettu valmiiksi, vaan se tarvitsee toteuttaa itse.

Näiden ongelmien välttämiseksi C++:aan on toteutettu standardikirjaston taulukko. [4]

Standardikirjaston taulukko varaa ja vapauttaa muistin automaattisesti. Se on luokka, joka ottaa kaksi parametria: alkioiden tietotyypin ja alkioiden määrän. Standardikirjaston taulukko tukee myös [] -operaattoria, mutta siihen on lisätty at(index) funktio, jota suositellaan käytettävän. At -funktio antaa virheen, jos sille annettu argumentti ei ole käypä. Standardikirjaston taulukon välittäminen toiselle funktiolle tapahtuu samalla tavalla kuin minkä tahansa sisäänrakennetun tietotyypin välittäminen. Sen voi välittää arvon tai viitteen avulla ja on myös mahdollista haluttaessa käyttää const -arvoa. Syntaksi ei sisällä mitään osoittimeen liittyviä operaatioita tai viittaustoimintoja. Näin ollen syntaksi on helposti luettavaa jopa moniulotteisissa rakenteissa. Lisäksi standardikirjaston taulukkoa välitettäessä, kaikki alkiot kopioituvat uuteen taulukkoon automaattisesti. Näin ollen standardikirjaston taulukko suorittaa syväkopioimisen vakiona. [4]

Standardikirjaston taulukko on yleensä C-tyylistä taulukkoa suositeltavampi vaihtoehto ja se onkin paljon enemmän käytetty. Tämän työn muissa luvuissa käytetään standardikirjaston taulukkoa ja siitä puhutaan jatkossa pelkkänä taulukkona.

2.1.2 Golangin taulukko

Gon taulukko on kiinteä datakokoelma, jonka pituus ja alkioiden tietotyyppi määritetään taulukon luontihetkellä. Taulukon kaikilla alkioilla täytyy olla keskenään sama tietotyyppi. [5] Taulukoilla pystytään suunnittelemaan yksityiskohtaista muistin käyttöä ja joissain tilanteissa välttämään tiedon hajaantumista muistin eri osiin. Pääasiassa Gon taulukot on kuitenkin tarkoitettu Gon viipaleen sisäisten osien toteutukseen. [6]

Gon taulukon ja C-tyylisen taulukon toteutuksien välillä on muutama suuri ero. Gon taulukot ovat arvoja, eli yhden taulukon määrittäminen toiselle taulukolle kopioi kaikki alkiot muistiin. Näin ollen, jos taulukon antaa parametrina funktiolle, se vastaanottaa kopion taulukosta, eikä osoitinta siihen. Gon taulukon tämän kaltainen arvo-ominaisuus on monissa tilanteissa hyödyllinen, mutta samalla kallis. Gossa voi siksi myös välittää osoittimen taulukkoon. Lisäksi yksi suuri ero C-tyyliseen taulukkoon on se, että Gon taulukon

koko on osa sen tietotyyppiä. Esimerkiksi taulukot *[5]int* ja *[10]int* ovat eri tietotyyppejä. [6]

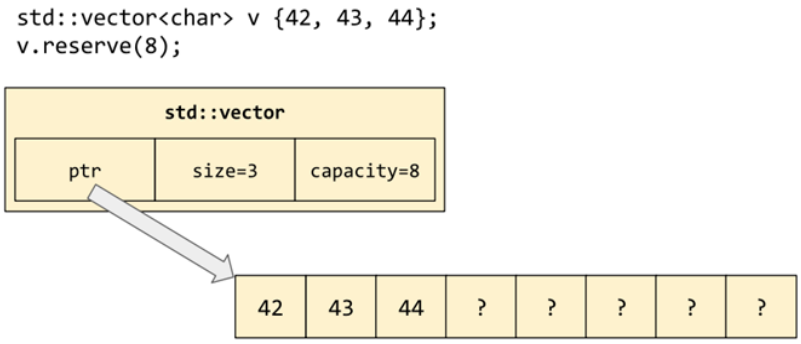
2.2 Vektori ja viipale

Taulukoita muistuttavat muuttuvakokoiset tietorakenteet ovat yleisiä tietorakenteita useissa ohjelmointikielissä. Tässä työssä perehdytään C++:n vektoriin ja Gon viipaleeseen.

2.2.1 C++:n vektori

Vektori on yksi C++:n yleisemmin käytetyistä standardikirjaston säiliöistä [4]. Vektori on dynaaminen sekvenssisäiliö, joka tallentaa sen alkiot peräkkäin muistiin. Se on C-tyylisen taulukon kanssa hyvin samankaltainen, mutta se on helpompi ja turvallisempi käyttää, koska vektori vapauttaa automaattisesti muistinsa ja kasvaa kooltaan uusien alkioiden myötä. [7]

Vektorilla on kaksi ominaisuutta: koko ja kapasiteetti. Vektorin koko on siihen tallennettujen alkioiden määrä. Vektorin koko muuttuu, kun siihen lisätään tai siitä poistetaan alkioita. Vektorin kapasiteetti taas on niiden alkioiden määrä, joka voidaan tallentaa vektorin varaamalle muistille. [4] Kuvassa 1 on esitetty vektorin rakenne.



Kuva 1. *Vektorin rakenne. [8]*

Kun vektori alustetaan, sille on annettava alkioiden tietotyyppi ja se varaa muistista tietyn alueen käyttöönsä. Lisäksi vektorille voidaan antaa alustuksen yhteydessä alkioita tai määritellä vektorin koko ohjelman 1 tapaan. [4]

```
// Tyhjä vektori
std::vector<int> vec;

// Vektorin alustus alkioiden kanssa voidaan tehdä kahdella tapaa.
std::vector<int> vec = {1,2,3,4,5};
// tai
std::vector<int> vec {1,2,3,4,5};

// Vektori alustaminen tietyn kokoiseksi.
std::vector<int> vec(10)

```

Ohjelma 1. *Vektorin alustaminen [4]*

Jos alustuksessa ei määritellä vektorin kokoa eikä vektoriin alkioita, niin kapasiteetille varattavan muistin määrää riippuu käytetyn kääntäjän toteutuksesta. Vektorin koko on silloin nolla, mutta kapasiteetti voi olla jokin pieni luku tai nolla. [4]

Alkioita voidaan lisätä vektoriin *push_back*- ja *insert*- funktioilla. *push_back* lisää alkion vektorin loppuun, kun taas *insert*- funktio lisää alkion parametrina saamaansa iteraattorin osoittamaan paikkaan. [4] Jos vektoriin lisätään alkioita niin, että vektorin koko ylittää sen kapasiteetin, vektorin on suoritettava uudelleenallokointi, koska kaikki alkiot on tallennettava samaan muistialueeseen [7]. Uudelleenallokoinnissa vektori varaa uuden muistialueen, joka on kooltaan kaksi kertaa suurempi kuin vanha muistialue. Vektori siirtää tai kopioi vanhat alkiot varaamalleen uudelle muistialueelle. Lopuksi vektori lisää uudet alkiot uuden muistialueen loppuun. Vanha muistialue pyyhitään pois automaattisesti. [4]

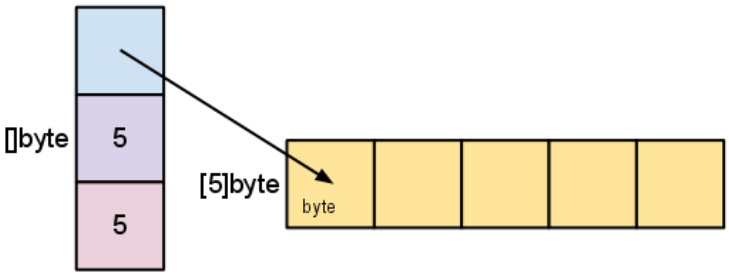
Alkioiden poistaminen vektorista pienentää vektorin kokoa, mutta ei muuta sen kapasiteettia. Kapasiteetin muuttamista varten vektorilla on omat funktiot. Alkion poistamiseen on kaksi eri funktiota: *pop_back* ja *erase*. *pop_back* poistaa vektorin viimeisen alkion, kun taas *erase* -funktio poistaa alkion parametrina saadun iteraattorin osoittamasta kohdasta. [7]

Vektorin alkioiden lisääminen ja poistaminen ovat yleensä tehokkaita operaatioita. Alkion lisääminen vektorin loppuun on tehokkuudeltaan yleensä *O(1)*, mutta uudelleenallokoitaessa se on *O(n)*. Alkion lisääminen muualle vektoriin on tehokkuudeltaan *O(n)*. Alkion poistaminen lopusta on tehokkuudeltaan *O(1)* ja muualta *O(n)*. Koska vektorin alkiot on tallennettu peräkkäin samaan muistialueeseen, on vektorin alkioihin pääsy aina yhtä tehokasta *O(1)*. [7]

2.2.2 Golangin viipale

Gon viipale on taulukon avulla toteutettu tehokas muuttuvan kokoinen tietorakenne, joka tallentaa alkiot peräkkäin muistiin. Tallennettavilla alkioilla täytyy olla keskenään sama tietotyyppi. [6]

C++:n vektorin tavoin Gon viipaleella on koko ja kapasiteetti. Viipale muodostuu kolmesta osasta: taulukkoon osoittavasta osoittimesta, viipaleen sisältävien alkioiden määrän kertovasta koosta ja taulukkoon mahtuvien alkioiden määrän kertovasta kapasiteetista. Viipaleen alkiot ovat siis todellisuudessa tallennettuna taulukkoon, johon viipale osoittaa. Kuvassa 2 on esitetty viipaleen rakenne. Taulukon täyttyessä alkioista on viipaleen koko ja kapasiteetti yhtä suuret, jolloin luodaan uusi kaksi kertaa edellistä taulukkoa suurempi taulukko ja vanhat alkiot kopioidaan siihen. Kopioinnin jälkeen viipaleen osoitin alkaa osoittamaan luotuun taulukkoon ja vanha taulukko tuhotaan. Näin ollen viipaleen kapasiteetti kaksinkertaistuu alkuperäisestä. [6]



Kuva 2. *Viipaleen rakenne. [6]*

Viipaletta välitettäessä se välitetään viitteenä. Näin ollen esimerkiksi funktiolle välitetty viipale on todellisuudessa viipalemuuttujan muistiosoite, ja siten funktion tekemät muutokset tapahtuvat alkuperäiseen viipaleeseen. Tämän lisäksi suuren viipaleen välittäminen funktiolle on huomattavasti nopeampaa kuin samankokoisen taulukon, koska sitä ei tarvitse kopioida toisin kuin taulukkoa välitettäessä. [9]

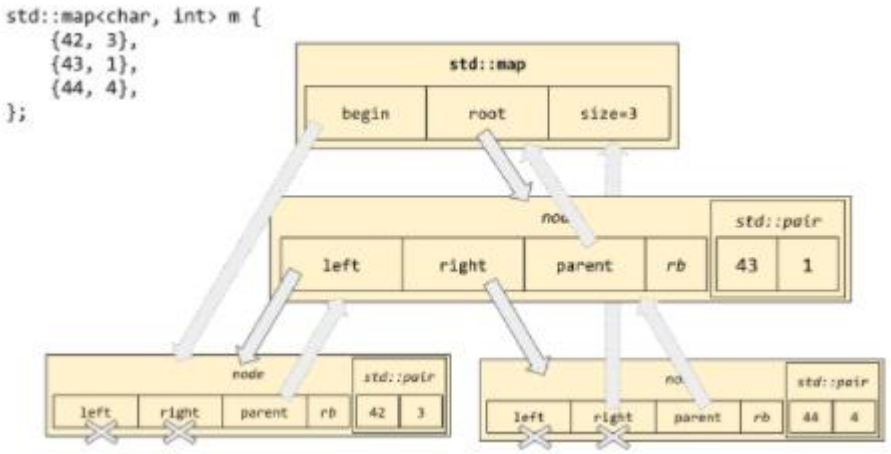
Viipaleen alkioiden lisääminen loppuun on yleensä tehokkuudeltaan *O(1)*, mutta sen kapasiteetin tullessa täytteen on lisääminen tehokkuudeltaan *O(n)*. Alkion poistaminen lopusta ja alkioon käsiksi pääseminen ovat tehokkuudeltaan aina *O(1)*. Alkion poistaminen muualta viipaleesta on tehokkuudeltaan *O(n)*. [9]

2.3 Sanakirja

Sanakirja on yleinen tietorakenne monissa ohjelmointikielissä. Siihen varastoidaan avain-arvo-pareja. Sanakirjan rakenne riippuu toteutuksesta. Tässä työssä perehdytään C++:n sanakirjaan, C++:n järjestämättömään sanakirjaan ja Gon sanakirjaan.

2.3.1 C++:n sanakirja

C++:n standardikirjaston sanakirja on järjestetty assosiatiivinen säiliö, jonne tallennetaan avain-arvo-pareja. Sanakirja on toteutukseltaan binäärihakupuu. Yleisemmissä toteutuksissa se on tarkalleen ottaen punamustapuu (kuva 3), joka on eräänlainen itsensä tasapainottava binäärihakupuu. Sanakirjan puu pitää automaattisesti tasapainoa yllä, vaikka siihen lisäisi tai siitä poistaisi alkioita kuinka paljon tahansa. Tasapainosta johtuen alkioiden lisääminen, poistaminen tai niiden hakeminen avaimen perusteella sanakirjasta vie keskimäärin *O(log n)*. [8]

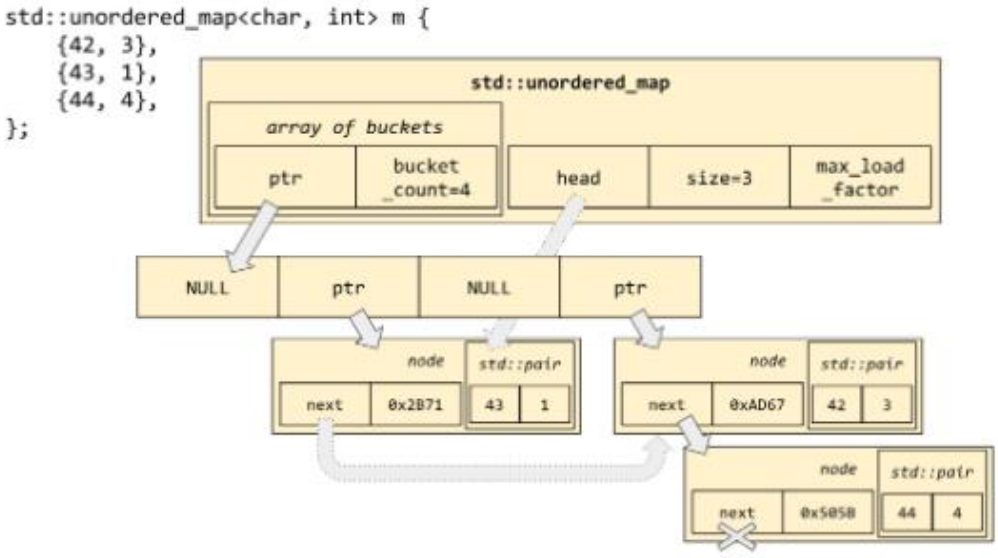


Kuva 3. *Sanakirjan binääripuun rakenne. [8]*

Sanakirjan sisältämien alkioiden arvoihin päästään käsiksi niiden avaimien avulla. Siksi sanakirjassa ei voi olla identtisiä avaimia. Alkiot järjestetään sanakirjassa avainten perusteella pienimmästä suurimpaan. Näin ollen alkiot eivät ole lisäämisjärjestyksessä, ja siksi sanakirja ei tue *push_back- ja push_front* -funktioita vaan alkioiden lisääminen tapahtuu joko *insert* -funktiolla tai [] -operaattorilla. Alkioiden poistaminen tapahtuu *erase* -funktiolla. [7]

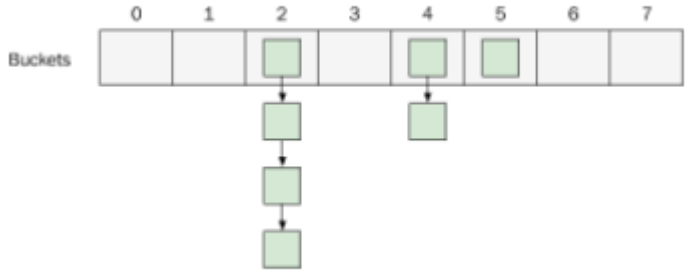
2.3.2 C++:n järjestämätön sanakirja

C++:n standardikirjaston järjestämätön sanakirja on järjestämätön assosiatiivinen säiliö. Sen käyttö on hyvin samankaltaista kuin sanakirjan, mutta siinä olevat alkiot eivät ole järjestyksessä. Järjestämättömän sanakirjan toteutus eroaa selkeästi sanakirjan toteutuksesta. Sanakirja on toteutettu binäärihakupuuna, kun taas järjestämätön sanakirja on toteutettu hajautustaulun (kuva 4) avulla. [8]



Kuva 4. *Järjestämättömän sanakirjan hajautustaulun rakenne. [8]*

Hajautustaulu on muuttuvan kokoinen taulukko ämpäreitä (buckets). Alkiota lisättäessä hajautustauluun sille lasketaan kokonaislukuindeksi tiivistefunktion avulla. Alkio lisätään hajautustauluun indeksiä vastaavaan ämpäriin. On mahdollista, että useat alkiot voivat saada saman indeksin arvon. Tämän takia ämpärit eivät sisällä vain yksittäisiä alkioita, vaan ämpärit ovat eräänlaisia säiliöitä. Ämpäreiden tarkkaa tietorakennetta ei ole määritelty, mutta usein käytetty toteutus on linkitetty lista. Samalla indeksillä olevat alkiot ovat siis saman ämpärin sisällä linkitetyssä listassa (kuva 5). Ämpäristä alkioiden hakeminen hidastuu, kun alkioiden määrä kasvaa ämpärissä, koska alkiot on käytävä läpi lineaarisesti. [10]



Kuva 5. *Hajautustaulun jokainen ämpäri sisältää nolla tai enemmän alkioita. [10]*

Järjestämättömän sanakirjan keskimääräinen tehokkuus saadaan hajautustaulun ansiosta paremmaksi kuin sanakirjan. Hajautustauluun lisääminen, siitä poistaminen ja hakeminen avaimen perusteella ovat keskiarvoiltaan vakioaikaisia *O(1)* operaatioita. Huonoimmassa tapauksessa kyseiset operaatiot ovat lineaarisia *O(n)*. [7]

2.3.3 Golangin sanakirja

Gossa sanakirja on toteutettu hajautustauluna. Sanakirjan avainarvo voi olla tietotyypiltään mikä tahansa vertailun mahdollistava tietotyyppi. Sanakirjan arvo voi olla tietotyypiltään mikä tahansa. [6]

Gon sanakirjan hajautustaulu on toteutukseltaan samanlainen kuin C++:n järjestämättömän sanakirjan hajautustaulu. Gon sanakirjan toteutuksessa käytetty hajautustaulu tallentaa alkiot ämpäreiden sisällä oleviin linkitettyihin listoihin. [9]

Sanakirjaan alkioiden lisääminen tapahtuu pelkästään [] -operaattorilla ja poistaminen tapahtuu *delete* -funktiolla. Hajautustauluun lisääminen, siitä poistaminen ja hakeminen avaimen perusteella ovat keskiarvoiltaan vakioaikaisia *O(1)* operaatioita. Huonoimmassa tapauksessa kyseiset operaatiot ovat lineaarisia *O(n)*. [9] Gon sanakirja on siis asymptoottiselta tehokkuudeltaan vastaava C++:n järjestämättömän sanakirjan kanssa.

3. METODOLOGIA

Perinteisesti algoritmien tai funktioiden laskennallinen aika-arviointi tehdään vertailuanalyysin ja asymptoottisten notaatioiden avulla. Vertailuanalyysissa pääidea on ajaa algoritmit tietokoneella ja mitata niiden nopeus joissakin aikayksiköissä. Vertailuanalyysiin perustuvaan arviointiin ei voida luottaa täysin, koska se mittaa tietyn ohjelman tehokkuutta, joka on kirjoitettu tietyllä kielellä ja joka toimii tietyllä alustalla tietyllä kääntäjällä sekä tietyllä syötetiedolla. Tämän yhden vertailuarvon perusteella on vaikea ennustaa, kuinka paljon aikaa algoritmi vie, jos se otetaan käyttöön järjestelmässä, jolla on erilainen joukko spesifikaatioita. Tämä ongelma voidaan välttää käyttämällä asymptoottisiin notaatioihin perustuvaa analyyttista lähestymistapaa.

3.1 Asymptoottinen suoritusaika

Algoritmien suoritusaikaa pystyy ennustamaan käyttämällä hyväkseen asymptoottisia notaatioita. Niiden avulla algoritmille pystytään muodostamaan tietyt rajat, jotka kuvaavat algoritmin suoritusaikaa syötemäärien kasvaessa tiettyä määrää suuremmaksi. Kolme yleisemmin käytettyä asymptoottista notaatiota ovat O-notaatio (iso-O-notaatio), Ω-notaatio (iso-omega-notaatio) ja Θ-notaatio (theetanotaatio) [11].

O-notaatio antaa algoritmille tai funktiolle asymptoottisen ylärajan. Se on notaatioista käytetyin, koska se antaa takeen siitä, että algoritmi käyttää korkeintaan annetun rajan verran aikaa suoritukseen. Se kertoo siis, että algoritmi ei ole koskaan tätä rajaa hitaampi, mutta se ei ota kantaa siihen, onko algoritmi rajaa nopeampi. [11]

Ω-notaatio antaa algoritmille tai funktiolle asymptoottisen alarajan. Se antaa takeen siitä, että algoritmi käyttää vähintään annetun rajan verran aikaa suoritukseen. Sen perusteella algoritmi ei ole milloinkaan annettua rajaa nopeampi. Se ei ota kantaa algoritmin hitauteen. [11]

Θ-notaatio rajoittaa algoritmia tai funktiota sekä yläpuolelta että alapuolelta. Se lupaa siis, että algoritmi on vähintään yhtä nopea kuin yläraja ja korkeintaan niin nopea kuin alaraja. Se antaa kyseisistä asymptoottisista notaatioista kaikista tarkimman kuvauksen algoritmin nopeudesta. [11]

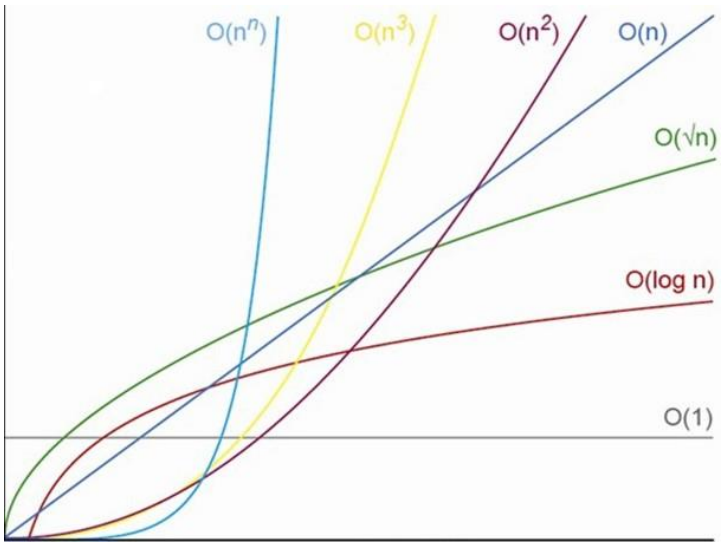
Notaatioiden kuvaamia asymptoottisia aikavaatimuksia on monenlaisia. Tehokkuuden kannalta kaikista paras mahdollinen asymptoottinen aikavaatimus on vakioaikainen aikavaatimus. Vakioaikaisessa asymptoottisessa aikavaatimuksessa algoritmin toiminta-

aika ei kasva, vaikka syötekoko kasvaisikin kuinka suureksi tahansa. Huonoin mahdollinen notaation kuvaama asymptoottinen aikavaatimus on eksponentiaalinen. Tällöin algoritmin syötteiden lisääntyminen kasvattaa suoritusaikaa eksponentiaalisesti, joten jo pienikin syötteiden määrän kasvu saa algoritmin hidastumaan huomattavasti. Näin ollen algoritmi, jonka tehokkuutta kuvaavan notaation asymptoottinen aikavaatimus on eksponentiaalinen, on jo hiemankin isommilla syötemäärillä käyttökelvoton ja vaatii optimointia toimiakseen riittävällä tehokkuudella. [11] Usein kuitenkin algoritmin tehokkuutta kuvaava notaatio saa jonkin muun asymptoottisen aikavaatimuksen näiden kahden edellä mainitun välistä. Muita mahdollisia asymptoottisia aikavaatimuksia ovat esimerkiksi lineaarinen aikavaatimus tai logaritminen aikavaatimus. Taulukossa 1 on esitetty yleisimpiä asymptoottisia aikavaatimuksia.

Taulukko 1. *Yleisimmät notaatioiden asymptoottiset aikavaatimukset. [11]*

vakio	$O(1)$
logaritminen	$O(\log n)$
lineaarinen	$O(n)$
n log n	$O(n \log n)$
neliöllinen	$O(n^2)$
kuutiollinen	$O(n^3)$
polynominen	$n^{O(1)}$
eksponentiaalinen	$2^{O(n)}$

Taulukossa asymptoottiset aikavaatimukset ovat tehokkuusjärjestyksessä siten että ylimpänä on tehokkain ja alimpana huonoin mahdollinen. Kuvassa 6 on esitetty muutamia taulukon 1 asymptoottisia aikavaatimuksia kuvaavia kuvaajia.



Kuva 6. *Notaatioiden asymptoottisten aikavaatimusten kuvaajat.*

Kuvassa 6 x-akselilla on syötteiden määrä ja y-akselilla algoritmin suoritusaika. Kuvasta nähdään, että syötemäärän kasvaessa vakioaikaista aikavaatimusta kuvaava notaatio on koko ajan vakioaikainen, kun taas eksponentiaalista aikavaatimusta kuvaavan notaation suoritusaika lähtee heti nopeaan kasvuun.

3.2 Vertailuanalyysi

Tässä työssä suoritetaan kokeita, joilla pyritään havainnoimaan C++- ja Go -ohjelmointikielten tietorakenteiden tehokkuuseroja käytännön tasolla. C++:n tietorakenteiden toimintojen suoritusaikoja mitataan Googlen kehittämällä suorituskykytestikirjastolla. Gon tietorakenteiden toimintojen suoritusaikoja mitataan sen sijaan Gon oman testauspaketin suorituskykytestifunktioilla. Molempien ohjelmointikielien suorituskykytestillä kirjoitetut funktiot pystyvät määrittämään suoritusajat alle nanosekunnin tarkkuudella. Näin ollen kokeissa saatavien tulosten tarkkuudet ovat riittävän tarkkoja.

C++:n vektorin ja Gon viipaleen ominaisuuksien tehokkuuksia mitataan kuudella eri tavalla ja neljällä eri kokoisella alkiojoukolla. Syötteinä toimivat alkiojoukot ovat kooltaan 1 000, 10 000, 1 000 000 ja 10 000 000 alkion joukkoja. Jokainen koe suoritetaan kolme kertaa ja saatavista tuloksista lasketaan keskiarvo koetuloksille.

Suoritettavat kokeet ovat:

1. Mitataan aikaa, joka kuluu koko alkiojoukon lisäämiseen tyhjään vektoriin/viipaleeseen.
2. Mitataan yhden alkion lisäämiseen kuluvaa aikaa, kun vektori/viipale sisältää jo koko syötteenä toimivan alkiojoukon. Lisättävän alkion arvo on kaikilla syötejoukoilla syötejoukon koko.
3. Mitataan viimeisen alkion poistamiseen kuluvaa aikaa, kun vektori/viipale sisältää jo koko syötteenä toimivan alkiojoukon.
4. Mitataan alkion poistamiseen vektorin/viipaleen keskeltä indeksin avulla kuluvaa aikaa. Indeksi on 70 % syötteenä toimivan alkiojoukon koosta.
5. Tarkistetaan, sisältääkö vektori/viipale alkion ja mitataan tarkistukseen kuluvaa aikaa.
6. Haetaan alkio vektorista/viipaleesta indeksin avulla ja mitataan hakemiseen kuluvaa aikaa.

Jokaisessa kokeessa sekä vektorilla että viipaleella syötteenä toimii sama alkiojoukko samassa järjestyksessä. Myös jokainen haettava alkio ja indeksi ovat molemmissa tapauksissa täysin samat. Näin pyritään saamaan molemmat koetilanteet mahdollisimman samankaltaisiksi ja näin mahdollisimman vertailukelpoisiksi keskenään.

C++:n ja Gon sanakirjatietorakenteiden ominaisuuksien tehokkuuksia mitataan viidellä erilaisella kokeella, joissa jokaisessa syötteinä toimii neljä erikokoista alkiojoukkoa. Kuten vektorin ja viipaleen kokeissa, myös näissä kokeissa alkiojoukot ovat kooltaan 1 000, 10 000, 1 000 000 ja 10 000 000 alkion joukkoja. Myös sanakirjarakenteiden kokeet suoritetaan kolmeen kertaan ja tuloksista lasketaan keskiarvo.

Suoritettavat kokeet ovat:

7. Mitataan koko alkiojoukon lisäämiseen tyhjään sanakirjaan kuluvaa aikaa.
8. Mitataan yhden alkion lisäämiseen kuluvaa aikaa, kun sanakirja sisältää jo koko syötteenä toimivan alkiojoukon. Lisättävän alkion avain ja arvo ovat molemmat syötejoukon koko.
9. Mitataan alkion poistamiseen sanakirjasta alkion avaimen avulla kuluvaa aikaa. Poistettava alkio on alkio, joka sijaitsee 70 % kohdalla syötteenä toimivan alkiojoukon koosta.
10. Mitataan alkion poistamiseen sanakirjasta alkion arvon avulla kuluvaa aikaa. Poistettava alkio on alkio, joka sijaitsee 70 % kohdalla syötteenä toimivan alkiojoukon koosta.
11. Mitataan alkion hakemiseen sanakirjasta alkion avaimen avulla kuluvaa aikaa.

Molempien kielten kokeissa hakuihin ja poistoihin käytettävien alkioiden avaimet ja arvot ovat identtisiä keskenään. Myös syötteinä toimivat alkiojoukot ovat identtiset.

Kokeiden suorittamiseen käytetään Dell XPS 15-7590 kannettavaa Windows-tietokonetta. Se sisältää Intel i7-9750H -prosessorin, 32GB DDR4 RAM-muistia ja Nvidia GeForce GTX 1650 -näytönohjaimen. Käytettävä Go -versio on 1.16.3 ja C++ -versio on 17. C++ -koodi käännetään gcc-kääntäjällä. C++ -koodia käännettäessä kääntäjä on julkaisumoodissa (release), eikä testausmoodissa (debug). Lisäksi kääntäjä käyttää /o2 optimointilippua. Näin ollen käännetystä ohjelmasta tulee huomattavasti nopeampi.

4. TEHOKKUUSVERTAILU

Kappaleen 3.2 mukaiset kokeet suoritettiin kummankin ohjelmointikielen molemmille tietorakenteille. Tässä luvussa esitellään saadut koetulokset ja niistä tehdyt havainnot.

4.1 Vektorin ja viipaleen tehokkuusvertailu

Ensimmäisenä suoritettiin C++:n vektorin ja Gon viipaleen tehokkuuksia vertailevat kokeet. Kokeet 1 ja 2 sisältävät alkioiden lisäämistä vektoriin ja viipaleeseen. Taulukossa 2 on esitetty kokeen 1 tulosten keskiarvot molemmille ohjelmointikielille.

Taulukko 2. *Usean alkion lisääminen vektoriin/viipaleeseen.*

Syötteen koko	Go viipale (ns)	C++ vektori (ns)
1 000	3 398	3 908
10 000	57 050	40 837
1 000 000	5 586 296	4 163 252
10 000 000	43 065 215	39 055 265

Usean alkion lisäämisessä samanaikaisesti ohjelmointikielistä Go oli pienimmällä syötejoukolla nopeampi verrattuna C++:aan. C++ kuitenkin ohitti nopeudessa Gon 10 000 alkion syötejoukolla ja oli nopeampi myös kahdella viimeisellä syötejoukolla.

Taulukossa 3 on esitetty kokeen 2 tulokset. Kokeessa lisätään yksi alkio vektoriin/viipaleeseen, joka sisältää jo koko syötteenä toimivan alkiojoukon.

Taulukko 3. *Alkion lisääminen vektorin/viipaleen loppuun.*

Syötteen koko	Go viipale (ns)	C++ vektori (ns)
1 000	0.48	3.48
10 000	0.48	3.53
1 000 000	0.48	3.58
10 000 000	0.48	3.27

Gossa yhden alkion lisääminen oli nopeaa, eikä lisääminen hidastunut kokeissa ollenkaan, vaikka viipaleen sisältämien alkioiden määrä kasvoi kokeiden välillä. Myöskään C++:ssa alkioiden lisääminen ei hidastunut ja koetulokset olivat hyvin lähellä toisiaan jokaisella syötejoukolla. Go oli C++:aan verrattuna nopeampi. Gossa lisäys kesti alle nanosekunnin, kun taas C++:ssa lisääminen kesti noin 3.5 nanosekuntia.

Kokeet 3 ja 4 sisältävät alkioiden poistamista vektorista ja viipaleesta. Taulukossa 4 on esitetty kokeessa 3 saadut tulokset.

Taulukko 4. *Viimeisen alkion poistaminen.*

Syötteen koko	Go viipale (ns)	C++ vektori (ns)
1 000	0.24	0.49
10 000	0.24	0.48
1 000 000	0.24	0.48
10 000 000	0.25	0.49

Kolmannessa kokeessa tietorakenteista poistettiin viimeinen alkio. Go suoriutui nopeammin kuin C++. Golle mitattu aika oli joka kerta noin 0.25 nanosekuntia, kun taas C++:lle mitattu aika oli noin 0.5 nanosekuntia kaikilla syötejoukoilla. Näin ollen yksittäinen poisto ei hidastunut alkiomäärän kasvaessa ollenkaan kummallakaan ohjelmointikielellä, vaan jokainen poisto tapahtui samalla nopeudella vektorin/viipaleen koosta riippumatta.

Taulukossa 5 on esitetty kokeen 4 tulokset. Kokeessa tietorakenteista poistettiin alkio indeksin avulla.

Taulukko 5. *Alkioin poistaminen vektorin/viipaleen keskeltä.*

Syötteen koko	Go viipale (ns)	C++ vektori (ns)
1 000	25	29
10 000	188	169
1 000 000	181 950	32 051
10 000 000	1 966 160	1 165 700

Kokeessa alkio poistettiin tietorakenteesta 70 prosentin kohdalta tietorakenteen pituudesta. Pienimmällä syötemäärällä Go oli C++:aan nopeampi, mutta syötemäärien kasvaessa Go hidastui enemmän kuin C++. Näin ollen suurilla syötemäärillä C++:sta tuli Gota nopeampi.

Kokeissa 5 ja 6 tarkastellaan alkioiden hakemista tietorakenteista. Taulukossa 6 on esitetty kokeen 5 tulokset.

Taulukko 6. *Alkion hakeminen tietorakenteesta.*

Syötteen koko	Go viipale (ns)	C++ vektori (ns)
1 000	183	181
10 000	1 690	1 712
1 000 000	199 218	177 228
10 000 000	3 514 855	2 423 908

Kokeessa 5 tarkasteltiin, sisältyikö annettu alkio tietorakenteeseen ja kauanko tarkistukseen kului aikaa. Pienillä syötemäärillä ohjelmointikielet olivat tasaisia keskenään, mutta alkiomäärän kasvaessa Go hidastui enemmän kuin C++, joten suurilla syötemäärillä C++ oli jo reilusti nopeampi kuin Go.

Taulukossa 7 on esitetty kokeen 6 tulokset. Viimeisessä vektorin ja viipaleen kokeessa haettiin tietorakenteista alkioita indeksien avulla.

Taulukko 7. *Alkion hakeminen indeksillä tietorakenteesta.*

Syötteen koko	Go viipale (ns)	C++ vektori (ns)
1 000	0.48	0.72
10 000	0.48	0.73
1 000 000	0.48	0.71
10 000 000	0.48	0.72

Kokeiden perusteella Go oli C++:aa nopeampi. Gon saamat suoritusajat olivat alle puoli nanosekuntia, kun taas C++:n suoritusajat olivat hieman alle 0,75 nanosekuntia. Kumpikaan ohjelmointikieli ei hidastunut yhtään syötemäärän kasvaessa.

4.2 Sanakirjojen tehokkuusvertailu

Vektori- ja viipale-tietorakenteiden kokeiden jälkeen suoritettiin sanakirjatietorakenteiden ja niiden ominaisuuksien nopeuksia vertailevat kokeet molemmille ohjelmointikielille.

Taulukossa 8 on esitetty kokeessa 7 mitattujen suoritusaikojen keskiarvot. Kokeessa 7 sanakirjatietorakenteeseen lisättiin syötemäärien verran uusia alkioita.

Taulukko 8. *Usean alkion lisääminen sanakirjaan.*

Syötteen koko	Go sanakirja (ns)	C++ sanakirja (ns)	C++ järjestämätön (ns)
1 000	76 353	35 115	4 418
10 000	670 689	452 251	95 359
1 000 000	123 993 894	79 246 000	60 801 109
10 000 000	1 641 818 300	1 574 928 000	3 760 065 700

Kolmella pienimmällä syötejoukolla C++:n järjestämätön sanakirja oli selvästi muita nopeampi. Se kuitenkin hidastui suurimmalla syötejoukolla huomattavasti. C++:n sanakirja oli kolmen pienimmän syötejoukon aikana toiseksi nopein. Se ei kuitenkaan hidastunut suurimmalla syötejoukolla läheskään yhtä paljoa kuin järjestämätön, joten se oli suurimmalla syötejoukolla tehdyn kokeen nopein. Gon sanakirja oli pienimmillä syötejoukoilla hitain, mutta se hidastui kaikista vähiten suhteessa edellisten syötejoukkojen kokeiden aikoihin. Näin ollen se ohitti suurimman syötejoukon kokeessa C++:n järjestämättömän sanakirjan, joka hidastui ylivoimaisesti eniten.

Taulukossa 9 on esitetty kokeen 8 tulokset. Kokeessa sanakirjaan, joka sisälsi syötealkiot, lisättiin yksi uusi alkio.

Taulukko 9. *Yhden alkion lisääminen sanakirjaan.*

Syötteen koko	Go sanakirja (ns)	C++ sanakirja (ns)	C++ järjestämätön (ns)
1 000	13.20	21.50	4.46
10 000	14.50	30.90	3.98
1 000 000	12.83	44.30	4.70
10 000 000	14.37	80.30	4.35

Tässä kokeessa kaikilla syötejoukoilla C++:n järjestämätön sanakirja oli muita nopeampi. Se oli noin kolme kertaa nopeampi kuin seuraavaksi nopein eli Gon sanakirja. Niillä alkioiden lisäysnopeus pysyi suunnilleen samana kaikilla syötemäärillä. C++:n sanakirjalla puolestaan alkioiden lisäysnopeus hidastui syötemäärien kasvaessa ja se oli kaikista kolmesta hitain jokaisessa kokeessa.

Taulukossa 10 on esitetty kokeen 9 suoritusaikoja. Kokeessa sanakirjasta poistetaan alkio sen avaimen avulla.

Taulukko 10. *Alkion poistaminen avaimen avulla sanakirjasta.*

Syötteen koko	Go sanakirja (ns)	C++ sanakirja (ns)	C++ järjestämätön (ns)
1 000	9.55	15.70	1.47
10 000	9.38	26.90	1.80
1 000 000	9.44	35.60	1.16
10 000 000	13.03	45.80	1.15

Koe meni tulosten järjestyksen osalta samoin kuin edellinen koe. C++:n järjestämätön sanakirja oli kaikkein nopein, Go oli toiseksi nopein ja C++:n sanakirja oli hitain. Gon sanakirjan ja C++:n järjestämättömän sanakirjan poistonopeudet pysyivät lähes muuttumattomina kaikissa kokeissa, kun taas C++:n sanakirja hidastui syötemäärien kasvaessa.

Taulukossa 11 on esitetty kokeen 10 suoritusajat. Kyseisessä kokeessa sanakirjasta poistettiin alkio sen arvon perusteella.

Taulukko 11. *Alkion poistaminen sanakirjasta arvon perusteella.*

Syötteen koko	Go sanakirja (ns)	C++ sanakirja (ns)	C++ järjestämätön (ns)
1 000	12 889	2 834	577
10 000	117 828	37 409	28 648
1 000 000	13 566 724	11 087 771	37 246 068
10 000 000	124 430 217	128 121 467	481 204 900

Kokeessa kahdella pienimmällä syötejoukolla C++:n järjestämätön sanakirja oli kaikista nopein, mutta syötemäärien noustessa miljoonaan alkioon ja sen yli se hidastui merkittävästi. Se oli kahdella suurimmalla syötemäärällä lähes neljä kertaa hitaampi kuin muut sanakirjat. Toiseksi nopein kahdella pienimmällä syötemäärällä oli C++:n sanakirja, mutta se ei hidastunut läheskään yhtä paljon kuin järjestämätön sanakirja. Gon sanakirja oli kahdella pienimmällä syötemäärällä kaikista hitain, mutta se hidastui selvästi muita vähemmän ja oli kaikkein nopein suurimmalla syötemäärällä.

Taulukossa 12 on esitetty kokeen 11 suoritusaikoja. Viimeisessä sanakirjatietorakenteen kokeessa mitattiin alkion hakemiseen kulunutta aikaa.

Taulukko 12. *Alkion hakeminen sanakirjasta.*

Syötteen koko	Go sanakirja (ns)	C++ sanakirja (ns)	C++ järjestämätön (ns)
1 000	7.96	11.30	4.21
10 000	8.72	14.80	3.91
1 000 000	7.95	22.50	4.14
10 000 000	9.12	26.90	3.99

Hakeminen tapahtui alkion avaimen avulla. Kokeessa Gon sanakirja ja C++:n järjestämätön sanakirja olivat huomattavasti C++:n sanakirjaa nopeampia, eivätkä ne hidastuneet juuri ollenkaan, vaikka alkioiden määrä sanakirjoissa moninkertaistui. C++:n sanakirja hidastui syötemäärien kasvaessa, mutta haku oli silti suorituksena varsin nopea.

5. TULOSTEN TARKASTELU

Tässä luvussa tarkastellaan kokeissa saatuja tuloksia. Lisäksi tarkastellaan tulosten yhteensopivuutta teorian kanssa.

5.1 Vektorin ja viipaleen tulokset

Alkioita lisättäessä C++:n vektori käyttäytyi odotetusti. C++:n vektoriin yksittäisen alkion lisääminen on teorian mukaan vakioaikainen operaatio ja taulukosta 2 havaittiin, että alkioiden määrän kymmenkertaistuessa myös suoritusaika suunnilleen kymmenkertaistui. Näin ollen yksittäisen alkion lisäämiseen kulunut aika oli lähes vakio, vaikka niitä lisättiinkin monta kerrallaan. Lisättäessä yksi alkio vektorin loppuun, C++:n suoritusajat pysyivät lähes samoina joka kokeessa. Tämäkin osoittaa lisäämisen käyttäytyvän vakioaikaisesti myös käytännössä.

Alkioita lisättäessä myös Gon viipale käyttäytyi vakioaikaisesti, kuten sen pitikin teorian perusteella. Taulukosta 2 havaittiin, että alkioiden määrän kymmenkertaistuessa myös suoritusaika suunnilleen kymmenkertaistui. Lisättäessä yksi alkio viipaleen loppuun, Gon suoritusajat pysyivät täsmälleen samoina.

Alkioita poistettaessa yksittäisen alkion poisto vektorin/viipaleen lopusta ei hidastunut alkiomäärän kasvaessa ollenkaan kummallakaan ohjelmointikielellä, vaan jokainen poisto tapahtui samalla nopeudella vektorin/viipaleen koosta riippumatta. Teorian perusteella molempien ohjelmointikielien poistot ovat vakioaikaisia, joten kokeen tulokset olivat odotettavissa. Poistettaessa alkio indeksin avulla, Go hidastui huomattavasti C++:aa enemmän suurilla syötemäärillä. Tuhannen ja kymmenentuhannen alkion syötemäärällä Gon ja C++:n nopeudet ovat suhteellisen lähellä toisiaan, miljoonan alkion syötemäärällä Gon nopeus on yli viisi kertaa C++:aa hitaampi, ja kymmenen miljoonan alkion syötemäärällä se on noin kaksi kertaa hitaampi kuin C++. Molemmissa ohjelmointikielissä alkion poistamisen kyseisten tietorakenteiden keskeltä pitäisi olla teorian mukaan lineaarinen operaatio, mutta taulukosta 5 havaittiin molempien tietorakenteiden hidastumisen kasvavan suurilla alkiomäärillä.

Alkion löytymistä tietorakenteesta tarkasteltaessa molempien kielten nopeudet hidastuivat syötemäärän noustessa suhteellisen lineaarisesti miljoonaan alkioon asti, mutta sen jälkeen syötemäärän kasvaessa kymmeneen miljoonaan, molempien kielten nopeudet hidastuivat enemmän. Gon nopeus hidastui C++:aa nopeammin. Alkiota indeksien avulla haettaessa molempien kielten suoritusnopeudet pysyivät joka kokeessa samoina. Tämä

oli odotettavissa, koska molemmista tietorakenteista indeksin avulla hakemisen pitäisi olla vakioaikainen operaatio.

Yksittäisiä alkioita koskevissa operaatioissa Gon viipale oli C++:n vektoria nopeampi, mutta operaatioissa, jotka vaativat tietorakenteen läpikäymistä tai muistin uudelleenallokointia C++:n vektori oli nopeampi. Tulosten perusteella molemmat tietorakenteet käyttäytyivät hyvin samankaltaisesti, eikä niiden välillä ollut suuria nopeuseroja.

5.2 Sanakirjojen tulokset

Teorian perusteella alkioita lisättäessä C++:n sanakirjan nopeus hidastuu logaritmisesti alkiomäärän kasvaessa, kun taas C++:n järjestämättömän sanakirjan ja Gon sanakirjan nopeuksien tulisi olla parhaimmillaan vakioaikaista kaikilla syötemäärillä ja huonoimmillaan lineaarista. Useita alkioita lisättäessä sanakirjaan havaittiin, että syötemäärien kasvaessa C++:n järjestämättömän sanakirjan ja Gon sanakirjan nopeudet eivät pysyneet vakioaikaisina. C++:n sanakirjan lisäämisnopeus hidastui logaritmisesti syötemäärien kasvaessa. Lisättäessä yksi alkio tietorakenteeseen, joka sisälsi jo valmiiksi tunnetun määrän alkioita, kaikki sanakirjatyypit käyttäytyivät odotusten mukaisesti teoriaan verrattaessa.

Alkioiden poiston avaimen avulla tulisi teorian mukaan tapahtua samoin kuin lisäämisenkin, C++:n sanakirjalla logaritmisesti ja C++:n järjestämättömällä sanakirjalla ja Gon sanakirjalla vakioaikaisesti. Koetulokset olivat avaimen avulla poistettaessa yhdenmukaisia teorian kanssa. Poistettaessa alkio sanakirjasta arvon perusteella, Gon sanakirjalla poistonopeus hidastui lineaarisesti syötemäärän kasvaessa**.** C++:n järjestämätön sanakirja hidastui kaikista eniten alkiomäärän kasvaessa. Sen hidastuminen oli lineaarista hidastumista nopeampaa. C++:n sanakirja hidastui toiseksi eniten.

Teorian perusteella alkioiden hakemisen avaimen perusteella tulisi olla vakioaikaista Gon sanakirjassa ja C++:n järjestämättömässä sanakirjassa. C++:n sanakirjalla sen tulisi olla logaritmista. Koetulokset olivat yhteneviä teorian kanssa. Alkion hakemisessa hajautustaululla toteutetut sanakirjat olivat binäärihakupuulla toteutettua C++:n sanakirjaa nopeampia.

C++:n järjestämätön sanakirja oli nopein sanakirjoista kaikissa muissa kokeissa, paitsi usean alkion lisäämisessä kahdella suurimmalla syötemäärillä ja alkion poistamisessa arvon avulla kahdella suurimmalla syötemäärillä. Gon sanakirja oli yksittäisiä alkioita koskevissa kokeissa C++:n sanakirjaa nopeampia, mutta C++:n sanakirja oli Gon sanakirjaa nopeampi usean alkion lisäämisessä ja alkion poistamisessa arvon perusteella kolmella pienimmällä alkiomäärällä.

6. YHTEENVETO

Tässä työssä tutustuttiin C++:n ja Golangin tietorakenteisiin ja vertailtiin niiden tehokkuuksia keskenään. Kielet valittiin, koska haluttiin vertailla kahta samantyyppistä ohjelmointikieltä, joista toinen on vanha ja erittäin yleisessä käytössä ja toinen uudempi eikä vielä niin käytetty ohjelmointikieli.

C++:n vektorin läpikäyminen oli nopeampaa kuin Gon viipaleen. Kun puolestaan tarkastellaan yksittäisiä operaatioita, jotka eivät vaatineet läpikäymistä tai uudelleenallokointia, Go oli nopeampi. Molemmat kielet olivat kuitenkin erittäin nopeita. Jos käytettävä ohjelmointikieli valittaisiin vektorin ja viipaleen tehokkuuden perusteella, ja tiedetään, että niihin ei tule miljoonittain alkioita, niin silloin Gon viipale on nopeampi. Jos taas vektoriin tai viipaleeseen tulee miljoonia alkioita ja suoritetaan useita hakuja, niin C++ on nopeampi.

Hajautustaululla toteutetut sanakirjat, eli C++:n järjestämätön sanakirja ja Go olivat lähes kaikissa kokeissa nopeampia kuin binäärihakupuuhun perustuva C++:n sanakirja. Näin ollen C++ -kielisissä ohjelmissa järjestämätön sanakirja on parempi vaihtoehto, jos alkioiden järjestyksellä ei ole merkitystä**.** Jos taas kieli valitaan tietorakenteiden tehokkuuden perusteella, niin C++:n järjestämätön sanakirja on nopeampi kuin Gon sanakirja, mutta ero ei ole merkittävä. Tehokkuuden perusteella ei kannata siis lähteä valitsemaan C++:n järjestämättömän sanakirjan ja Gon väliltä, ellei valmistettava ohjelma ole todella aikakriittinen.

Erot eri ohjelmointikielten välillä olivat lopulta paljon pienemmät kuin alun perin odotettiin. Gon nopeus yllätti, sillä odotuksena oli, että C++ olisi ollut kielistä huomattavasti nopeampi, koska C++ on tehokkuusvertailussa yleensä pärjännyt erinomaisesti [10].

LÄHTEET

[1] Tiobe indeksi. (2021). https://www.tiobe.com/tiobe-index/

[2] Popularity of Programming Language Index PYPL. (2021). https://pypl.github.io/PYPL.html

[3] ISO *International Standard ISO/IEC 14882:2020(E) – Programming Language C++*.

[4] Carey J, Doshi S, Rajan P. C++ Data Structures and Algorithm Design Principles: Leverage the Power of Modern C++ to Build Robust and Scalable Applications. Birmingham: Packt Publishing, Limited; 2019.

[5] Bhagvan Kommadi. Learn Data Structures and Algorithms with Golang. Packt Publishing; 2019.

[6] Golang dokumentaatio. (2021). https://go.dev

[7] Van Weert P, Gregoire M. C++ Standard Library Quick Reference. Berkeley, CA: Apress L. P; 2016.

[8] O'Dwyer A. Mastering the C++17 STL. Birmingham: Packt Publishing, Limited; 2017.

[9] Tsoukalos M. Mastering Go. 1st edition. Packt Publishing; 2018

[10] Sehr V, Andrist B. C++ high performance: boost and optimize the performance of your C++ 17 code. Birmingham: Packt Publishing; 2018.

[11] Stein C, Leiserson CE, Cormen TH, Rivest RL. Introduction to algorithms. The MIT Press; 2009.