Tampereen yliopisto

Juha-Matti Ryttyläinen

C++-STANDARDIKIRJASTON SÄILIÖT JA NIIDEN TEHOKKUUS

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Huhtikuu 2019

TIIVISTELMÄ

Juha-Matti Ryttyläinen: C++-standardikirjaston säiliöt ja niiden tehokkuus
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikka, TkK
Huhtikuu 2019

Ohjelmoinnissa väistämättä tulee eteen tilanteita, joissa saman tyyppistä dataa on organisoitava ja säilöttävä. Tätä tarvetta varten on kehitetty tietorakenteita, joita C++-standardikirjastossa kutsutaan säiliöiksi. Säiliöt ovat siis C++-standardin määrittelemiä yleisiä tietorakenteita. Standardikirjasto tarjoaa useita eri säiliöitä eri käyttötarkoituksiin, joista osa on yleiskäyttöisiä ja osa hyvin tilannekohtaisia.

Tämän kandidaatintyön tavoite on tutkia standardikirjaston säiliöitä ja niiden suorituskykyä. Työssä esitellään ensin perustavanlaatuisia tietorakenteita joihin säiliöt perustuvat. Säiliöt käydään läpi ryhmittäin ja liitetään esiteltyihin tietorakenteisiin. Säiliöiden suorituskykyä tutkitaan työssä säiliöoperaatioiden asymptoottisen aikavaativuuden ja suorituskykytestejen näkökulmasta. Lopuksi työssä annetaan käyttösuosituksia säiliöiden käytöstä.

Työssä tehty analyysi ei paljastanut mitään yllättävää. Työssä tehty tutkimus ja johtopäätökset tukevat kirjallisuudesta löytyvää konsensusta siitä, mitä säiliöitä tulisi käyttää oletusarvoisesti ja mitkä ovat enemmän tilannekohtaisia.

Avainsanat: Tietorakenteet, Säiliöt, C++, C++-standardikirjasto

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

ASCII	American Standard Code of Information Interchange
ISO	Kansainvälinen standardointiorganisaatio
Iteraattori	Olio joka mahdollistaa kulkemisen säiliön alkiosta toiseen.

1 JOHDANTO

C++ on standardisoitu yleiskäyttöinen ohjelmointikieli. Ohjelmointikielenä C++ on oliopohjainen ja sen syntaksi on C-kielen tyylinen. C++-ohjelmointikielen tyypillisimmät käyttökohteet ovat systeemiohjelmointi, videopelimoottorit ja sulautetut järjestelmät. Kirjoitushetkellä voimassaoleva C++-standardi on ISO/IEC 14882:2017, joka määrittelee ydinkielen lisäksi myös C++-standardikirjaston. Standardikirjastolla tarkoitetaankin ydinkielellä kirjoitettuja luokkia ja funktioita, jotka ovat myös itsessään osa C++ ISO -standardia. Standardikirjasto tarjoaa ohjelmoijalle merkkijonotyypin, IO-virtoja, säiliöitä, funktioita säiliöiden manipuloimiseen ja muita yleisiä funktioita. C++-ohjelmointikielestä ja standardikirjastosta on monia eri toteutuksia, jotka noudattavat standardia vaihtelevasti. Kuitenkin kielen yleisimmin käytetyissä toteutuksissa kielen keskeisimmät ominaisuudet on toteutettu hyvin samalla tavalla.

Tässä työssä käsitellään C++-standardikirjaston tarjoamia säiliöitä, säiliöiden tehokkuutta ja käyttökohteita. Työssä verrataan säiliöiden tehokkuutta toisiinsa ja selvitetään millaisissa käyttötilanteissa tulisi käyttää mitäkin säiliötä. Työssä ei perehdytä syvällisesti siihen, miten eri toteutukset standardikirjastosta toteuttavat työssä käsiteltyjä säiliöitä.

Luvussa kaksi esitellään standardikirjaston tarjoamat säiliöt ja niiden pohjalla olevat tietorakenteet. Kolmannessa luvussa tutkitaan tavallisten säiliöoperaatioiden asymptoottista suorituskykyä ja testataan eri säiliöiden tehokkuutta ohjelmoituja suorituskykytestejä hyödyntäen. Neljäs luku pohtii eri säiliöiden käyttötilanteita ja tehdään kappaleen 3 suorituskykytestien perusteella johtopäätöksiä siitä, milloin mitäkin säiliötä tulisi käyttää. Yhteenvedossa (luku 5) koostetaan johtopäätöksiä siitä, mitä työssä on tehty ja tehdään lopulliset pohdinnat aiheesta.

2 STANDARDIKIRJASTON SÄILIÖIDEN ESITTELY

C++-standardi määrittelee kaikki standardikirjaston säiliöt ja asettaa vaatimukset niiden toteutukselle. Koska standardi määrittää säiliöille vaatimukset, mutta ei pakota toteuttamaan säiliöitä samalla tavalla, eri toteutusten välillä saattaa olla pieniä eroja. Tästä eteenpäin sanalla säiliö viitataan vain C++-standardikirjaston säiliöihin, ellei toisin mainita. C++-standardin mukaan säiliöt ovat olioita, jotka säilövät muita olioita. Säiliöt kontrolloivat näiden olioiden muistinvarausta ja muistinvapautusta rakentajien, purkajien sekä alkioiden lisäys- ja poisto-operaatioiden avulla.[2]

Säiliöt ovat siis geneerisiä luokkia, joiden avulla ohjelmoija voi hyödyntää yleisiä tietorakenteita. Säiliöt jaetaan kolmeen eri luokkaan: sarjat, assosiatiiviset säiliöt ja järjestämättömät assosiatiiviset säiliöt. Säiliö hallitsee muistia, jota sen alkioille on varattu, ja mahdollistaa jäsenfunktioiden avulla alkioiden käsittelyn suoraan tai iteraattoreiden avulla. Monet säiliöt jakavat samanlaisia toiminnallisuuksia ja jäsenfunktioita keskenään.[1]

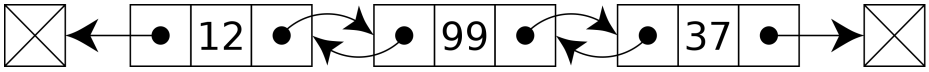
2.1 Tietorakenteet

Monet C++-säiliöt perustuvat samoihin perustavanlaatuisiin tietorakenteisiin, kuten taulukko (array), linkitetty lista (linked list), punamusta puu (red-black tree) ja hajautustaulu (hash table). Ennen standardisäiliöiden käsittelyä täytyy ymmärtää miten kyseiset tietorakenteet toimivat.

Taulukko on tietorakenne, joka koostuu kokoelmasta alkioita. Taulukon alkiot erotellaan toisistaan indeksillä. Indeksi on järjestysnumero, joka kuvaa alkion paikkaa taulukossa. Yksinkertaisimmillaan taulukkoa voidaan siis ajatella kokoelmana peräkkäisiä muistipaikkoja. Taulukko perustavanlaatuisena tietorakenteena on laajasti käytetty ja moni muu tietorakenne perustuu siihen. Taulukot voivat myös sisältää alkioinaan muita taulukoita, jolloin taulukko on moniulotteinen.

Linkitetty lista on tietorakenne, jossa talletettavat oliot ovat lineaarisessa järjestyksessä. Linkitetyn listan sisältämiä olioita kutsutaan solmuiksi. Toisin kuin taulukossa, jossa rakenteen järjestys määritellään indekseillä, linkitetyssä listassa järjestys määritetään jokaisen solmun sisältämällä osoittimella. Linkitetty lista voi olla yhteen tai kahteen suuntaan linkitetty. Yhteen suuntaan linkitetyssä listassa jokainen solmu sisältää vain yhden seuraavaan solmuun osoittavan osoittimen. Yhteen suuntaan linkitettyä listaa voi tästä syystä kulkea vain yhteen suuntaan. Kahteen suuntaan linkitetyssä listassa taas jokainen solmu

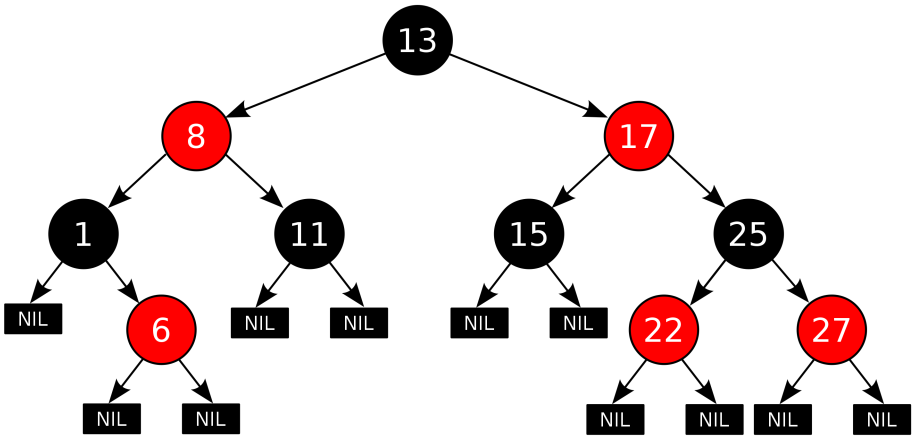
sisältää sekä edeltävään, että seuraavaan solmuun osoittavat osoittimet. Kuvassa 2.1 on esitelty kahteen suuntaan linkitetty lista joka sisältää kokonaislukuja. Linkitetyn listan tyypillisiä operaatioita ovat uuden alkion asettaminen listaan, alkion poistaminen listasta ja alkion etsiminen listasta. Linkitettyä listaa pitkin voidaan helposti kulkea seuraamalla aina osoitinta seuraavaan solmuun.[3]



Kuva 2.1. *Kahteen suuntaan linkitetty lista.[4]*

Punamusta puu on binäärihakupuu, joka on tasapainotettu taatakseen tietyille operaatioille tietyn nopeuden. Jokaiselle punamustan puun solmulle on määritelty väri: joko punainen tai musta. Solmujen väriä hyödyntämällä voidaan rajoittaa polut puun juuresta lehteen siten, että yksikään tälläinen polku ei ole yli kaksi kertaa pidempi kuin toinen. Näin ollen punamusta puu on suunnilleen tasapainotettu. Jokainen puun solmu sisältää värin, arvon, osoittimen vasemalle ja osoittimen oikealle. Kuvassa 2.2 on esimerkki punamustasta puusta.[3] Punamusta puu on siis binääripuu, jonka solmut täyttää seuraavat ehdot:

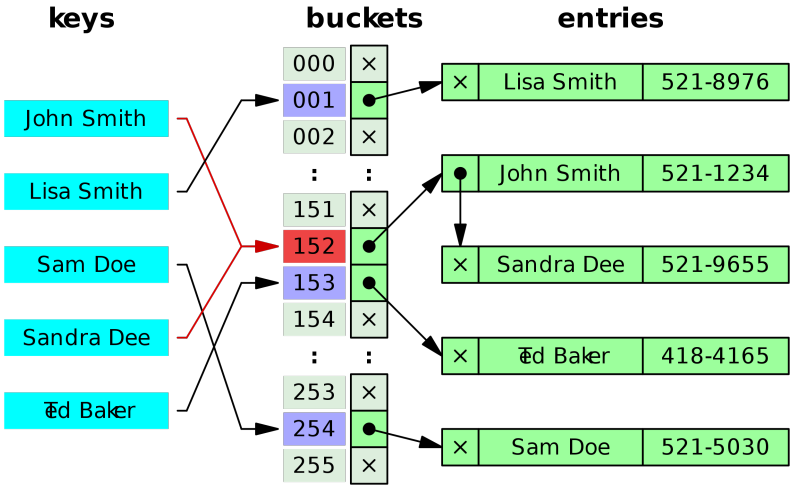
1. Jokainen solmu on joko punainen tai musta.
2. Juurisolmu on musta.
3. Jokainen lehti on musta.
4. Jos solmu on punainen sen lapsisolmut ovat mustia.
5. Jokaisesta solmusta sen lehtiin lähtevät polut sisältävät saman verran mustia solmuja.



Kuva 2.2. *Punamusta puu.[8]*

Hajautustaulu (hash table) on tietorakenne, joka perustuu tavalliseen taulukkoon. Suoran indeksoinnin sijaan hajautustaulussa indeksointi tapahtuu hajautusfunktion avulla. Hajautustaulussa indeksointi tapahtuu epäsuorasti avainten välillä. Avain voi olla käytännössä mitä vain tietotyyppiä riippuen hajautustaulun toteutuksesta. Syöttämällä avain hajautusfunktioon saadaan avaimen tiiviste (hash) palautusarvona. Avaimen tiiviste on tavallinen

kokonaisluku, joka viittaa hajautustaulun tapauksessa johonkin taulukon indeksiin. Hajautusfunktiot ovat luonteeltaan sellaisia, että erilaiset avaimet saattavat joskus tuottaa saman tiivisteen. Tälläistä tilannetta kutsutaan törmäykseksi (collision). Törmäyksiä tapahtuu silloin tällöin, joten ne on käsiteltävä. Törmäyksiä voidaan käsitellä siten, että jokainen alkio hajautustaulussa onkin osoitin linkitetyn listan alkuun. Näin ollen jos törmäys sattuu lisätään uusi alkion kyseisestä indeksistä löytyvän linkitetyn listan perään. Näitä ylivuotolistoja kutsutaan usein ämpäreiksi.[3] Kuvassa 2.3 on esitelty ylivuotolistoja hyödyntävä hajautustaulu.



Kuva 2.3. *Hajautustaulu.[7]*

2.2 Iteraattorit

Tietorakenteita voidaan kulkea alusta loppuun ja usein myös lopusta alkuun. Tietorakenteita pitkin kulkiessa alkioiden arvoja voidaan lukea ja kirjoittaa. Kuitenkin eri säiliöitä pitkin kuljetaan eri tavoin. Esimerkiksi taulukon tapauksessa alkioon viitataan sen indeksillä, kun taas linkitetyn listan tapauksessa alkioon viitataan sitä edeltävästä alkiosta löytyvällä osoittimella. Jotta samoja algoritmejä voidaan hyödyntään eri tietorakenteisiin, tämä erilaisuus on ratkaistava jollain tavalla. Ratkaisemaan tätä ongelmaa on keksitty tietorakenteen kulkemista abstrahoiva olio nimeltä iteraattori.

Iteraattori on siis olio, joka mahdollistaa eri tietorakenteiden kulkemisen samaa rajapintaa hyödyntäen. Tietorakenteen alkua ja loppua kuvataan parilla iteraattoreita, joita kutsutaan alku- ja loppuiteraattoreiksi.[9] Alku- ja loppuiteraattoreihin päästään käsiksi säiliöiden jäsenfunktioilla `begin()` ja `end()`. Alkuiteraattori osoittaa aina säiliön ensimmäiseen alkioon, kun taas loppuiteraattori osoittaa viimeisen alkion ohi. Iteraattoreita voi kasvattaa `++`-operaattorilla, jolloin iteraattori siirtyy osoittamaan seuraavaan alkioon.

Category	Container	After **insertion**, are...		After **erasure**, are...		Conditionally
		iterators valid?	**references** valid?	**iterators** valid?	**references** valid?	
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
			No	No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		
Associative containers	`set` `multiset` `map` `multimap`	Yes		Yes, except erased element(s)		
Unordered associative containers	`unordered_set` `unordered_multiset` `unordered_map` `unordered_multimap`	No	Yes	N/A		Insertion caused rehash
		Yes		Yes, except erased element(s)		No rehash

Kuva 2.4. *Iteraattoreiden mitätöityminen.[1]*

Iteraattoreita käytettäessä on otettava huomioon niiden mitätöityminen (engl. invalidation). Mitätöitymisellä tarkoitetaan tilannetta missä iteraattorin osoittama alkio hukkuu syystä tai toisesta, jolloin iteraattori muuttuu käyttökelvottomaksi. Puhtaat lukuoperaatiot eivät koskaan mitätöi iteraattoreita, mutta säiliön sisältöä muuttavat operaatiot saattavat näin tehdä.[1] Kuvassa 2.4 on esitelty iteraattoreiden mitätöitymiseen johtavat operaatiot eri säiliöille.

2.3 Sarjasäiliöt

Sarjasäiliöt organisoivat äärellisen joukon samaa tyyppiä olevia olioita lineaariseen järjestykseen. Standardikirjasto tarjoaa neljä perustavanlaatuista sarjasäiliötä: vector, forward list, list ja deque. Näiden lisäksi tarjolla on array, joka tarjoaa rajallisen määrän sarjaoperaatioita, koska sen alkioiden määrä on luonnin jälkeen vakio.[2]

Array on yksinkertainen staattisen kokoinen säiliö, joka perustuu luvussa 2.1 esiteltyyn, C-kielestä tuttuun taulukkoon. Array yhdistää C-tyylisen taulukon tehokkuuden ja käytettävyyden standardisäiliön ominaisuuksien kanssa. Normaalien taulukon ominaisuuksien lisäksi array tietää kokonsa, mahdollistaa alkioon sijoittamisen ja tarjoaa rajapinnan iteraattoreiden käyttöön.[1] Taulukon tyypillisimmän operaation eli indeksoinnin lisäksi array-rajapinta tarjoaa monia muita jäsenfunktioita.

Vector on sarjasäiliö, joka muuttaa kokoaan kun siihen lisätään alkioita. Kuten array, vector on myös taulukkotietorakenteeseen perustuva säiliö, mutta vector perustuu dynaamiseen, eli kokoaan muuttavaan taulukkoon. Vector siis kasvaa tai pienenee automaattisesti tarpeen vaatiessa. Yleensä vector vie enemmän tilaa kuin staattiset taulukot, sillä se varaa muistia tulevaisuutta varten. Tästä seuraa, että vectorin ei tarvitse varata lisää muistia joka kerta kun siihen lisätään alkio.[1]

Deque on sarjasäiliö, jonka tavoitteena on tarjota nopea alkion lisääminen ja poistami-

nen kumpaankin päähän tietorakennetta. Toisin kuin vectorissa, dequen alkioita ei tallenneta peräkkäin. Tyypilliset toteutukset dequesta hyödyntävät sarjaa erikseen allokoituja staattisen kokoisia taulukkoja. Deque perustuu kaksipäiseen jonoon (engl. double-ended queue).[1]

List on säiliö, joka tukee vakioaikaista alkion lisäämistä ja poistamista missä vain kohtaa rakennetta. List on toteutettu luvussa 2.1 esiteltynä kahteen suuntaan linkitettynä listana, joten se ei tue nopeaa satunnaishakua. Koska list on toteutettu kahteen suuntaan linkitettynä listana, sitä voi iteroida kumpaankin suuntaan. Listasta poistaessa ja listaan lisätessä sen iteraattorit eivät mitätöidy. [1]

Forward list on säiliö, joka tukee nopeaa alkion lisäämistä ja poistamista missä vain kohtaa rakennetta. Forward list on toteutettu kappaleessa 2.1 esiteltynä yhteen suuntaan linkitettynä listana, joka tarkoittaa, että forward list on tilan suhteen tehokkaampi kuin list. Koska forward list on yhteen suuntaan linkitetty sitä voi iteroida vain yhteen suuntaan. Säiliöstä alkion poistaminen ja säiliöön alkion lisääminen eivät mitätöi sen iteraattoreita.[1]

2.4 Assosiatiiviset säiliöt

Assosiatiiviset säiliöt ovat järjestettyjä säiliöitä, joista voidaan nopeasti etsiä alkioita avainten avulla. Avaimella tarkoitetaan jotain oliota, esimerkiksi merkkijonoa, jota käytetään tunnistamaan tietty alkio tietorakenteessa. Standardikirjasto tarjoaa neljä perustavanlaatuista assosiatiivista säiliötä: set, multiset, map ja multimap. Jotkut assosiatiiviset säiliöt vaativat, että avaimet ovat uniikkeja, kun taas toiset taas sallivat saman avaimen käytön useampaan kertaan.[2]

Set on assosiatiivinen säiliö, joka sisältää järjestetyn joukon uniikkeja avaimena toimivia olioita. Järjestystä ylläpidetään vertailemalla lisättävää avainta muihin sitä lisätessä. Operaatiot: haku, lisääminen ja poisto ovat logaritmisiä aikavaativuudeltaan. Set on yleensä toteutettu luvussa 2.1 esiteltynä punamustana puuna.[1]

Map on järjestetty assosiatiivinen säiliö, joka sisältää avain-arvo -pareja. Mapin järjestystä ylläpidetään vastaavasti kuin setissä, eli vertailemalla lisättävää avainta muihin. Niin ikään haku, lisääminen ja poistaminen ovat logaritmisiä aikavaativuudeltaan. Myös map on toteutettu luvussa 2.1 esiteltynä punamustana puuna.[1]

Multiset ja multimap vastaavat muuten toteutukseltaan set- ja map-säiliöitä, mutta multiset ja multimap eivät vaadi avainten olevan uniikkeja. Multiset ja multimap tarjoavat samankaltaiset operaatiot ja aikavaativuudet kuin set ja map.

2.5 Järjestämättömät assosiatiiviset säiliöt

Järjestämättömät assosiatiiviset säiliöt toteuttavat järjestämättömiä tiivisteisiin perustuvia tietorakenteita. Nämä säiliöt tarjoavat datan nopean haun avaimen perusteella. Standardikirjasto tarjoaa neljä eri järjestämätöntä assosiatiivista säiliötä: unordered set, unordered map, unordered multiset ja unordered multimap.[2]

Unordered set on säiliö, joka sisältää joukon uniikkeja avaimina toimivia olioita. Unordered set perustuu kappaleessa 2.1 esiteltyyn hajautustauluun. Hajautustaulussa alkiot eivät ole järjestyksessä, vaan jaettuna useaan ämpäriin. Se, mihin ämpäriin alkio lisätään, riippuu alkion tiivisteen arvosta. Alkioita ei voi muuttaa säiliössä, koska tällöin myös tiivisteen pitäisi muuttua.[1]

Unordered map on järjestämätön avain-arvo -pareja sisältävä säiliö. Unordered map vaatii, että avaimet ovat uniikkeja.[1] Sisäisesti alkiot on järjestetty vastaavasti kuin unordered setissä. Myös unordered map on toteutettu luvussa 2.1 esiteltynä hajautustauluna.

Unordered multiset ja unordered multimap ovat versioita unordered set ja unordered map -säiliöistä, jotka eivät vaadi avainten olevan uniikkeja. Unordered multiset ja unordered multimap on toteutettu vastaavasti, kuin unordered set ja unordered map, joten ne tarjoavat vastaavat operaatiot.[1]

2.6 Säiliösovittimet

Säiliösovittimet (container adaptors) tarjoavat erilaisia yleisiä rajapintoja sarjasäiliöille. Säiliösovittimet siis käyttävät sisäisesti aina jotain sarjasäiliötä, mutta tarjoavat tiettyyn sovellukseen rajatun rajapinnan.

Stack, eli pino on sovitin, joka tarjoaa LIFO-tyylisen rajapinnan. LIFO:lla (last-in, first-out) tarkoitetaan, että viimeksi lisätty alkio on ensimmäisenä poistettava alkio. Stack voi käyttää sisäisesti sarjasäiliöitä kuten vector, deque tai list. Vakiona stack käyttää deque-säiliötä.[1]

Queue, eli jono on sovitin, joka tarjoaa FIFO-tyylisen rajapinnan. FIFO:lla (first-in, first-out) tarkoitetaan, että ensimmäisenä lisätty alkio on myös ensimmäisenä rakenteesta poistettu alkio. Queue voi käyttää sisäisesti vectoria tai listaa.[1]

Priority queue, eli prioriteettijono on sovitin, joka tarjoaa vakioaikaisen haun isoimmalle alkiolle rakenteessa. Priority queue voi sisäisesti käyttää vector- tai deque-säiliöitä.[1]

3 SÄILIÖIDEN TEHOKKUUS

Ohjelmistojen tehokkuus on tärkeä seikka ohjelmistokehityksessä. Historiallisesti ohjelmistojen suorituskykyyn panostaminen oli tärkeä osa sovelluskehittäjän työtä, koska tietokoneet olivat yleisesti hitaita. Nykyaikaisten prosessorien tehokkuuden takia suorituskyky on siirtynyt yhä enemmän pois sovelluskehittäjien mielestä. Suorituskyky on kuitenkin nykyäänkin tärkeää, mutta osittain muista syistä kuin ennen. Tänä päivänä akun varassa toimivat mobiilialustat, kuten älypuhelimet ovat hyvin laajassa käytössä. Älypuhelimissa ja muissa mobiililaitteissa akun kesto on tärkeä seikka, johon voidaan vaikuttaa positiivisesti hyvin optimoiduilla ohjelmistoilla. Toinen yleinen alue, missä suorituskyky on tärkeää, ovat palvelinohjelmistot. Monet yritykset ja organisaatiot ylläpitävät valtavia määriä palvelimia, joko suoraan osana liiketoimintaansa, tai osana liiketoimintaa tukevaa infrastruktuuriaan. Nämä suuret määrät palvelimia kuluttavat paljon sähköä, joten jos sähkönkulutusta saadaan pienemmäksi, pienentää se palvelinten ylläpitokustannuksia. Näin ollen jos palvelimilla suoritettavat ohjelmistot ovat hyvin optimoituja, palvelimiin liittyvät kulut laskevat.

Tässä kappaleessa tarkastellaan säiliöoperaatioiden tehokkuutta sekä aikavaativuuden kannalta, että suorituskykytestien muodossa. Aikavaativuustarkastelu antaa yleensä hyvän kuvan operaatioiden suorituskyvystä kussakin käyttökohteessa. Nykyaikaiset alustat ovat kuitenkin niin monimutkaisia, että suorituskykytesteissä voi ilmetä yllättäviäkin tuloksia. Tästä syystä suorituskyvyn mittaaminen käyttötilannekohtaisesti on tärkeää.

3.1 Säiliöoperaatioiden asymptoottinen aikavaativuus

Asymptoottista aikavaativuutta tutkiessa tarkastellaan algoritmin suoritusajan kasvua suhteessa sisäänmenevien alkioiden määrään. Tämä tarkastelu on hyödyllinen, sillä siihen eivät vaikuta alustakohtaiset erot. Aikavaativuuden tarkastelu ja ymmärtäminen on oleellista algoritmien ja tietorakenteiden valinnassa. Taulukoissa 3.1, 3.2 ja 3.3 on esiteltynä yleisten operaatioiden aikavaativuuksia eri säiliöille. Taulukoissa tähdellä (*) merkityt solut sisältävät keskimääräisiä aikavaativuuksia, eli huonoimman tapauksen aikavaativuudet ovat näissä tilanteissa korkeampia. Taulukoissa sana ”amortized” tarkoittaa, että usean samanlaisen operaation aikavaativuus keskiarvoittuu kyseiseen vaativuuteen.

Taulukko 3.1. *Sarjasäiliöiden operaatioiden aikavaativuudet. [5][1]*

operation	vector	deque	list
at()/operator[]	O(1)	O(1)	
begin()/end()	O(1)	O(1)	O(1)
front()/back()	O(1)	O(1)	O(1)
push_front()		O(1) (amortized)	O(1)
push_back()	O(1) (amortized)	O(1) (amortized)	O(1)
insert()	O(n)	O(n)	O(1)
pop_front()		O(1)	O(1)
pop_back()	O(1)	O(1)	O(1)
erase()	O(n)	O(n)	O(1)
std::sort()/sort()	O(n log n)*	O(n log n)*	O(n log n)
std::find()	O(n)	O(n)	O(n)

Taulukko 3.2. *Järjestettyjen assosiatiivisten säiliöiden operaatioiden aikavaativuudet. [5][1]*

operation	set	multiset	map	multimap
at()/operator[]			O(log n)	
begin()/end()	O(1)	O(1)	O(1)	O(1)
insert()	O(log n)	O(log n)	O(log n)	O(log n)
erase()	O(1) (amortized)	O(1) (amortized)	O(1) (amortized)	O(1) (amortized)
find()	O(log n)	O(log n)	O(log n)	O(log n)

Taulukko 3.3. *Assosiatiivisten säiliöiden operaatioiden aikavaativuudet. [1]*

operation	unordered_set	unordered_multiset	unordered_map	unordered_multimap
at()/operator[]			O(1)*	
begin()/end()	O(1)	O(1)	O(1)	O(1)
insert()	O(1)*	O(1)*	O(1)*	O(1)*
erase()	O(1)*	O(1)*	O(1)*	O(1)*
find()	O(1)*	O(1)*	O(1)*	O(1)*

Asymptoottisia aikavaativuuksia tarkastellessa voidaan tehdä useita huomioita. Taulukosta 3.1 voidaan tulkita, että list on operaatioidensa aikavaativuudelta joko parempi tai yhtä hyvä, kuin vertailtavana olevat vector ja deque. Taulukosta 3.2 voidaan tulkita, että set, multiset, map ja multimap tarjoavat esitellyille operaatioille keskenään samat aikavaativuudet. Taulukosta 3.3 voidaan tulkita, että myös unordered set, unordered multiset, unordered map ja unordered multimap tarjoavat esitellyille operaatioille keskenään samat

aikavaativuudet. Jos verrataan taulukoita 3.2 ja 3.3 huomataan, että järjestämättömät säiliöt tarjoavat paremman keskiarvoisen aikavaativuuden jokaisen vertailtavan operaation kohdalla järjestettyihin verrattuna.

3.2 Suorituskyvyn testaus

Yleisellä tasolla ohjelmistojen nopeuteen eniten vaikuttava tekijä on muistin nopeus. Nykyaikaisen prosessorin näkökulmasta muisti on todella hidasta, jopa niin hidasta, että se rajoittaa lähes jokaisen ohjelman suorituskykyä. Hidas muisti heikentää suorituskykyä, sillä prosessori joutuu odottamaan käskyjensä operandien noutamista muistista ennen kuin käsky voidaan suorittaa. Myös muistiin kirjoittaminen voi heikentää suorituskykyä, koska keskusmuistiin kirjoittamiseen käytetyt muistipuskurit voivat täyttyä odottaessaan mahdollisuutta kirjoittaa hitaaseen keskusmuistiin.[6]

Nykyaikaisissa prosessoreissa käytetäänkin pieniä ja nopeita muisteja, joita kutsutaan välimuisteiksi (cache). Välimuisteilla pyritään parantamaan muistiviittauksien viivettä. Prosessoreissa on nykyään tyypillisesti monta tasoa välimuistia, joita kutsutaan L1-, L2- ja L3-välimuistiksi. Mitä suurempi välimuisti on, sen hitaampi se on. L1 on siis nopein ja pienin välimuisti ja vastaavasti L3 hitain ja isoin välimuisti. Prosessorin välimuisteissa säilytetään datan lisäksi myös konekäskyjä. Hitaimmatkin välimuistit ovat 5 - 10 kertaa nopeampia kuin keskusmuisti. Nopeimmat välimuistit ovat satoja kertoja keskusmuistia nopeampia.[6]

Keskusmuistin hitauden takia pelkän aikavaativuuden tarkastelu voi johtaa harhaan tietyissä tilanteissa. Nykyaikaisissa prosessoriarkkitehtuureissa olevat välimuistit vaikuttavat paljon käytännössä näkyvään nopeuteen algoritmeissa ja tietorakenteissa. Prosessorien muistipaikallisuus on tärkeä konsepti, joka vaikuttaa merkittävästi suorituskykyyn. Muistipaikallisuudella tarkoitetaan, että ohjelman käyttämä data sijaitsee mahdollisimman pienessä osassa muistiavaruutta. Kun korkea muistilokaliteetti on saavutettu välimuistista ohi menevät muistiviittaukset vähenevät.

Taulukko 3.4. *Suorituskykytestien alkiomäärät suhteessa alkiokokoihin.*

Alkiomäärät alkion koolla 5	Alkiomäärät alkion koolla 500	Alkiomäärät alkion koolla 5000
$2*10^6$	$2*10^4$	$2*10^3$
$4*10^6$	$4*10^4$	$4*10^3$
$6*10^6$	$6*10^4$	$6*10^3$
$8*10^6$	$8*10^4$	$8*10^3$

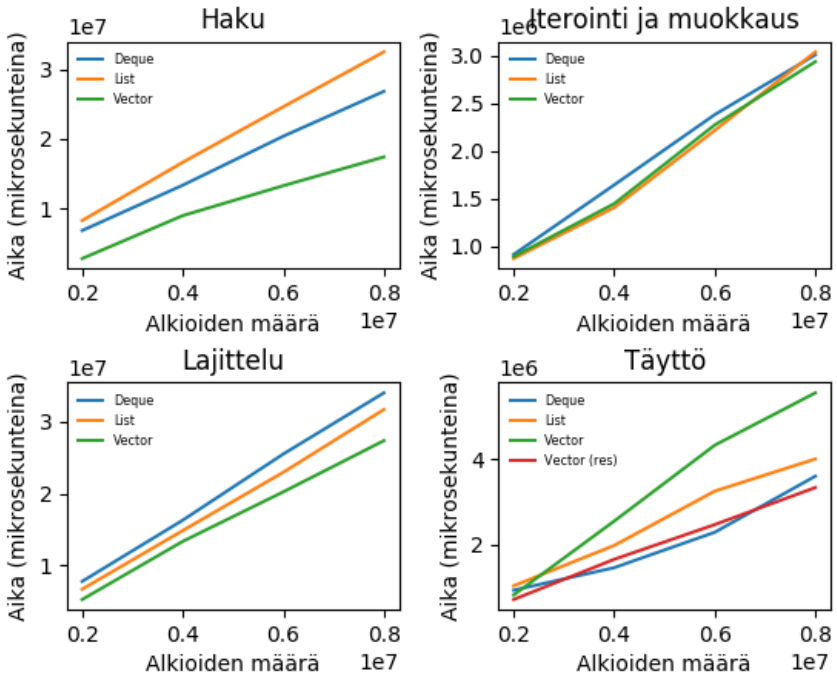
Paras suorituskykytesti säiliöille on testata suorituskykyä juuri siinä käyttökohteessa, jota varten säiliötä ollaan valitsemassa. Säiliöiden tehokkuutta voidaan kuitenkin mallintaa synteettisillä suorituskykytesteillä, joiden perusteella pystytään muodostamaan nyrkkisääntöjä oikean säiliön valitsemiseen. Tässä luvussa esitellään muutama synteettinen suorituskykytesti eri säiliöiden operaatioille, jotta voidaan kartoittaa, millaisista operaa-

tioista mikäkin säiliö suoriutuu ja miten hyvin. Luvussa esitellyt suorituskykytestit on suoritettu Fedora-linux virtuaalikoneella. Kyseiselle virtuaalikoneelle allokoitiin yksi ydin Intel Xeon CPU E5-2680-prosessorista sekä 8 Gt keskusmuistia. Kaikki esitellyt suorituskykytestit käännettiin ilman kääntäjäoptimointeja GCC:n versiolla 8.3.1. Testien suoritusajat mitattiin standardikirjastosta löytyvällä Chrono-kirjastolla. Jokainen säiliöille tehty suorituskykytesti suoritettiin samalla alkiokoolla neljä kertaa säliötä kohden. Testien eri suorituskerroilla käytettiin eri määriä alkioita, jotta saataisiin esiteltyä suoritusajan kehittymistä alkiomäärien kasvaessa.

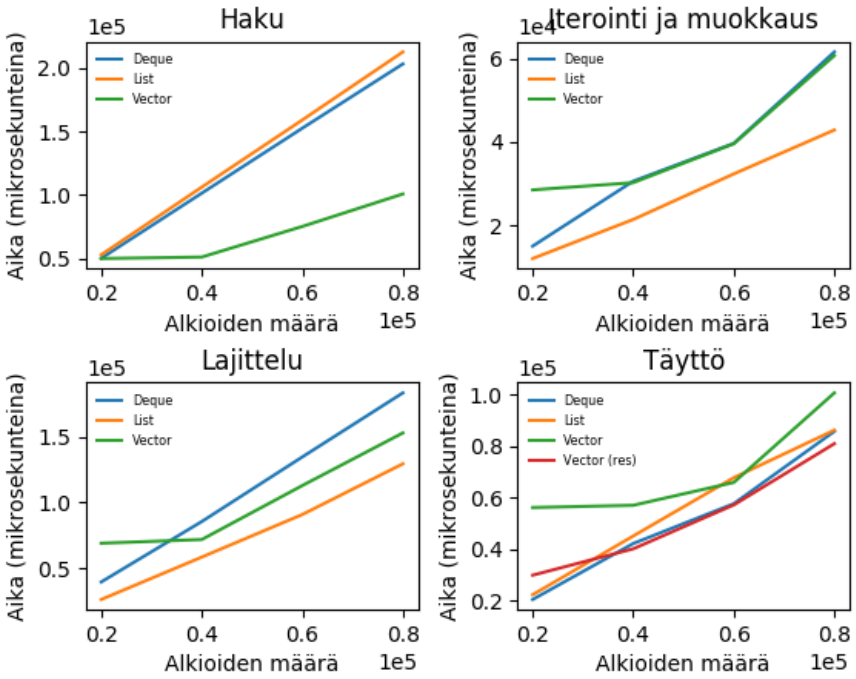
3.2.1 Sarjasäiliöiden suorituskyky

Sarjasäiliöille esitellyt suorituskykytestit ovat säiliön täyttö, säiliön lajittelu, alkion haku ja iterointi sekä alkion muuttaminen. Jokainen testi on suoritettu kolmella eri alkion koolla. Alkioina testeissä käytettiin 5-, 500- ja 5000-tavuisia standardikirjaston ASCII-merkkijonoja. Arraytä ei otettu mukaan suorituskykytesteihin, koska sen rajapinta ei tarjoa kyseisissä testeissä käytettyjä operaatioita. Arrayn suorituskyky vastaa kuitenkin vectorin suorituskykyä niiltä osin, kun samoja operaatioita löytyy. Forward listiä ei myöskään ole suorituskykytesteissä, sillä sen operaatioiden suorituskyky vastaa listaa, mutta rajapinta on suppeampi.

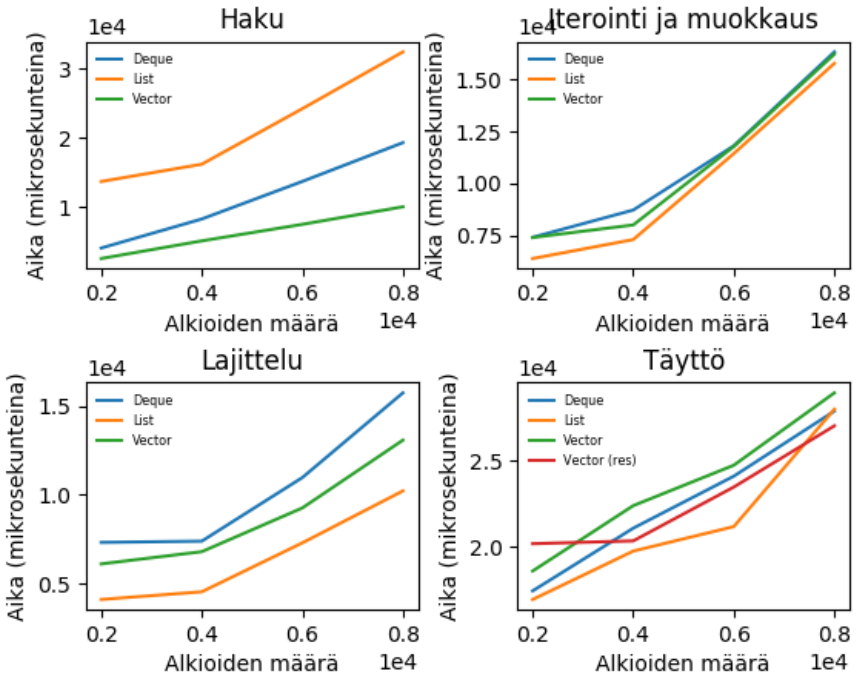
Täyttötesteissä alkioita lisättiin taulukossa 3.4 esitellyt määrät kuhunkin säiliöön ja mitattiin tähän kulunut kokonaisaika. Testissä käytettiin `push_back()`-jäsenfunktiota alkioden lisäämiseen. Lajittelutestissä taulukossa 3.4 esitellyn kokoiset säiliöt lajiteltiin ja mitattiin lajitteluun kulunut aika. Testissä list-säiliölle käytettiin funktiota `list.sort()` ja muille säiliöille standardikirjaston `std::sort()`-funktiota. Hakutestissä säiliöistä etsittiin sata satunnaisesti valittua säiliöistä löytyvää avainta ja mitattiin etsimiseen kulunut kokonaisaika. Testissä käytettiin standardikirjaston `std::find()`-funktiota. Testissä ”iterointi ja alkion muuttaminen”, säiliön läpi iteroitiin muuttaen jokaista säiliön sisältämää alkiota ja mitattiin kulunut kokonaisaika. Testissä muokattiin alkioita vaihtamalla kunkin merkkijonon kaksi ensimmäistä merkkiä standardikirjaston merkkijonon jäsenfunktiolla `replace()`.



Kuva 3.1. *Suorituskykytestit 5 ASCII-merkin merkkijonoilla.*



Kuva 3.2. *Suorituskykytestit 500 ASCII-merkin merkkijonoilla.*



Kuva 3.3. *Suorituskykytestit 5000 ASCII-merkin merkkijonoilla.*

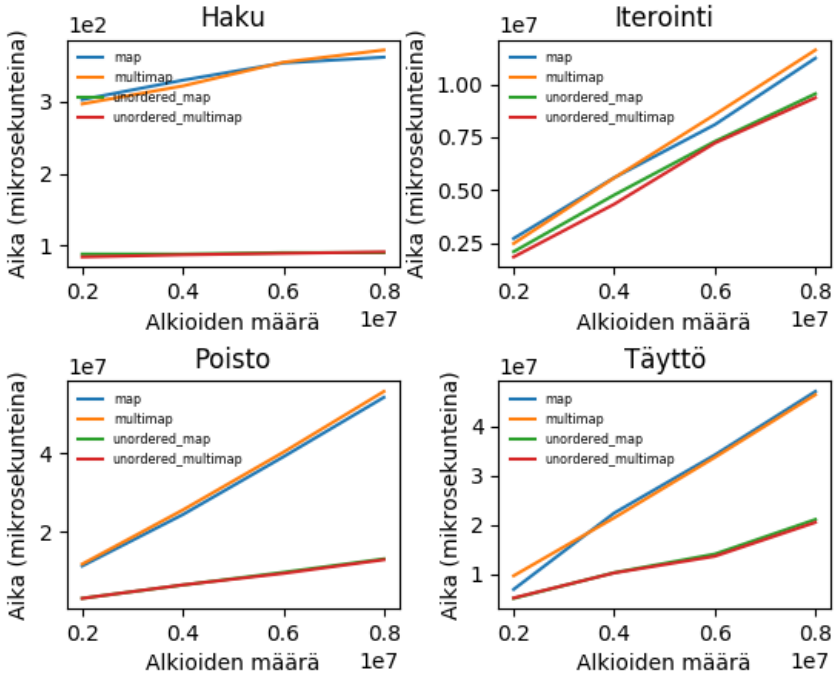
Viiden ASCII-merkin kuvasta 3.1 voidaan todeta, että kaikissa neljässä testissä vectorilla on pienimmät suoritusajat. Testeissä ”iterointi ja muokkaus” ja täyttö deque-säiliö suoriutuu yhtä hyvin kuin vector. Täyttötestissä huomataan, että vectorin kohdalla on oleellista kutsua muistia ennalta varaavaa reserve-funktiota, mikäli alkiomäärä tiedetään edes suurin piirtein etukäteen. Kuvista 3.1, 3.2 ja 3.3 voidaan todeta, että alkiokoon kasvaessa list-säiliön suoritusajat pienenevät suhteessa vectoriin. Kuvasta 3.3 huomataan, että lajittelutestin kohdalla list saavuttaa paremmat suoritusajat kuin vector.

3.2.2 Assosiatiivisten säiliöiden suorituskyky

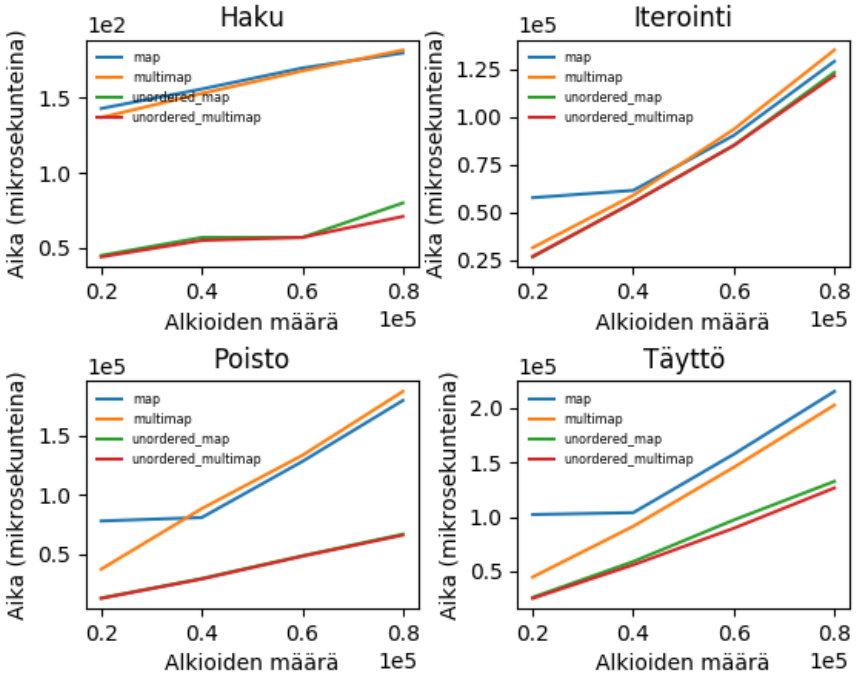
Assosiatiivisille säiliöille esitellyt suorituskykytestit ovat täyttö, haku, poisto ja iterointi. Jokainen testi on suoritettu kolmella eri alkion koolla. Vastaavasti kuin sarjasäiliöitä testatessa alkiona käytettiin 5-, 500- ja 5000-tavuisia standardikirjaston ASCII-merkkijonoja. Suorituskykytesteistä jätettiin ”set”-tyyppiset säiliöt pois, koska ne ovat toteutuksiltaan samanlaiset kuin ”map”-tyyppiset säiliöt. Ainoana erona set- ja map-säiliöiden välillä on siis se, että map liittää avaimen arvoon, kun taas set sisältää pelkän avaimen.

Täyttötestissä alkioita lisättiin 3.4 esitellyt määrät kuhunkin säiliöön ja mitattiin siihen kulunut kokonaisaika. Alkioiden lisäämiseen käytettiin säiliöiden `insert()`-jäsenfunktiota. Hakutestissä säiliöistä etsittiin sata satunnaisesti valittua säiliöistä löytyvää avainta ja mitattiin etsimiseen kulunut kokonaisaika. Avainten löytämiseen käytettiin säiliöiden `find()`-jäsenfunktiota. Poistotestissä poistettiin kustakin säiliöstä täyttötestissä lisätyt alkiot ja mi-

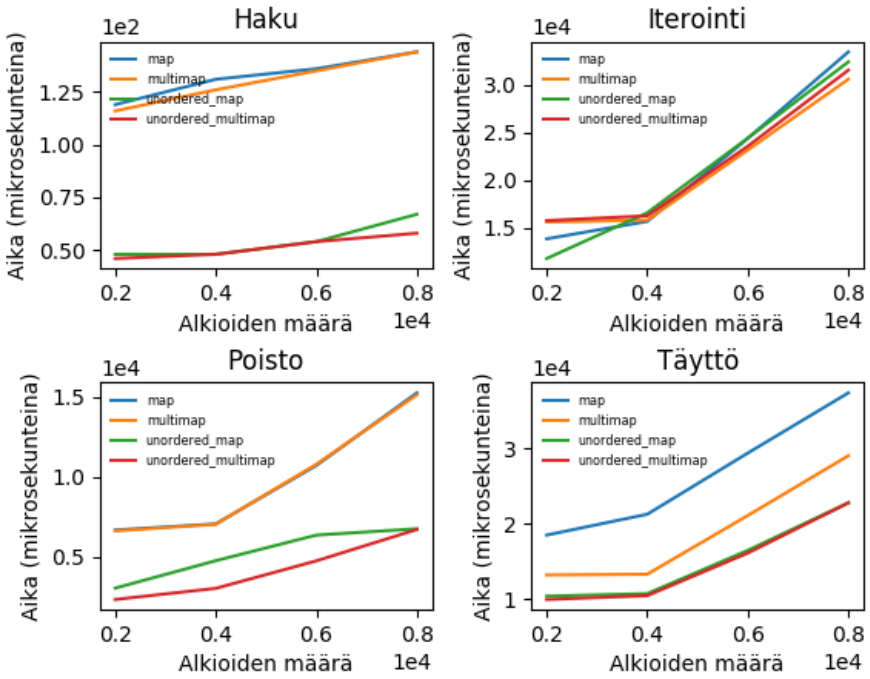
tattiin kulunut kokonaisaika. Alkioiden poistamiseen käytettiin säiliöiden `erase()`-jäsenfunktiota. Iterointitestissä iteroitiin kunkin säiliön läpi lukien jokaisella iteraatiolla alkion arvo muuttujaan ja mitattiin kulunut kokonaisaika.



Kuva 3.4. *Suorituskykytestit 5 ASCII-merkin merkkijonoilla.*



Kuva 3.5. *Suorituskykytestit 500 ASCII-merkin merkkijonoilla.*



Kuva 3.6. *Suorituskykytestit 5000 ASCII-merkin merkkijonoilla.*

Kuvista 3.4, 3.5 ja 3.6 voidaan todeta, että hajautustauluun perustuvat säiliöt unordered map ja unordered multimap suoriutuvat lähes poikkeuksetta testeistä nopeimmin. Ainoastaan iterointitestin kohdalla säiliöt suoriutuvat hyvin tasaisesti isointa alkiokokoa käyttäessä.

4 SÄILIÖIDEN KÄYTTÖSUOSITUKSET

Säiliöiden valitseminen on olennainen seikka algoritmien ja ohjelmistokomponenttien toteutuksessa. Ohjelmistoja toteuttavan henkilön on hyvä tietää mitä säiliöitä tulisi käyttää oletusarvoisesti ja millaisiin erityistilanteisiin muut säiliöt soveltuvat. Tämä on tärkeää varsinkin, jos kyseinen henkilö on kehittämässä suorituskykykriittistä osaa ohjelmistosta. Kaikki säiliöt eivät kuitenkaan sovi kaikkiin käyttötilanteisiin jo pelkästään tarjoamiensa rajapintojen takia.

Tässä kappaleessa annetaan käyttösuosituksia säiliöiden käytöstä. Suosituksia annetaan sekä suorituskykytestien ja asymptoottisen aikavaativuuden perusteella, että kirjallisuudesta löytyvien suositusten perusteella.

4.1 Sarjasäiliöiden käyttösuositukset

Luvun 3 tulosten perusteella voidaan päätellä, että sarjasäiliöiden tilanteessa, alkiokoon ollessa pieni, on syytä käyttää vectoria, kunhan se vain tarjoaa riittävän rajapinnan. Arrayn operaatioiden aikavaativuus on pitkälle sama kuin vectorilla, mutta sitä käyttäessä ei tarvitse esimerkiksi huolehtia reserve-funktion kutsumisesta. Jos siis käyttötilanne on sellainen, jossa säilytettävien alkioden määrä on vakio, on syytä käyttää array-säiliötä. Suorituskykytesteistä voidaan tulkita myös, että list tuottaa verrattaen parempia tuloksia alkiokoon kasvaessa.

C++-standardin suositus sarjasäiliöiden kohdalla vastaa tehtyjen suorituskykytestejen tulosta. Standardin mukaan vector- ja array-säiliöitä tulisi käyttää oletusarvoisesti. List- ja forward list-säiliöitä tulisi käyttää silloin, kun tehdään paljon alkion lisäys- tai poisto-operaatioita keskelle säiliötä. Deque-säiliötä taas tulisi käyttää, kun alkion lisäys- tai poisto-operaatioita tehdään paljon säiliön etu- ja takaosaan.[2] Myös C++-kielen luojan Bjarne Stroustrupin mukaan vector-säiliötä tulisi käyttää oletusarvoisesti[9].

4.2 Assosiatiivisten säiliöiden käyttösuositukset

Säiliöt, kuten unordered map, unordered set, unordered multimap ja unordered multiset tarjoavat asymptoottisen aikavaativuuden näkökulmasta suorituskykyisemmät operaatiot kuin muut assosiatiiviset säiliöt. Luvun 3 tulosten perusteella suorituskykytestit

ovat samaa mieltä aikavaativuusanalyysin kanssa. Voidaan todeta, että assosiatiivisten säiliöiden tilanteessa tulisi ensisijaisesti käyttää hajautustauluihin perustuvia säiliöitä silloin, kun suorituskyvyllä on väliä. Aikavaativuusanalyysin perusteella tilanteet jossa punamustaan puuhun perustuvat set, map, multiset ja multimap ovat suorituskyvyltään parempia ovat harvinaisia. Tälläinen tilanne voisi kuitenkin olla mahdollinen, jos vaikkapa haku-operaatioita käytetään paljon suhteessa muihin operaatioihin.

C++-kielen luojan mukaan säiliötä unordered map tulee käyttää korkean suorituskyvyn tarpeisiin map-säiliön sijasta, jos käyttötilannetta varten on mahdollista kehittää hyvä hajautusfunktio[9]. Tämä näkemys vastaa sekä aikavaativuustarkastelua, että suorituskykytestien tuloksia.

5 YHTEENVETO

Työssä esiteltiin C++-standardikirjaston säiliöt ja tutkittiin niiden suorituskykyä. Työn tavoitteena oli selvittää mitä säiliöitä tulisi käyttää suorituskyvyn näkökulmasta. Luvussa kaksi esiteltiin C++-standardikirjaston säiliöt ja säiliöiden perustana olevat tietorakenteet. Kolmannessa luvussa tutkittiin säiliöiden eri operaatioiden tehokkuutta sekä asymptoottisella aikavaativuustarkastelulla, että suorituskykytestien muodossa. Luvussa neljä annettiin suosituksia säiliöiden käytöstä luvun 3 ja kirjallisuuden perusteella.

Sarjasäiliöiden kohdalla todettiin, että vector-säiliötä tulisi käyttää oletusarvoisesti ja muut sarjasäiliöt ovat enemmän tilannekohtaisia. Huomattiin myös, että sarjasäiliöiden tilanteessa pelkkä aikavaativuustarkastelu voi johtaa harhaan. Assosiatiivisten säiliöiden kohdalla todettiin aikavaativuustarkastelun olevan hyvin linjassa suorituskykytestien kanssa. Todettiin, että hajautustauluihin perustuvia säiliöitä (unordered map, unordered set, unordered multimap ja unordered multiset) on syytä käyttää oletusarvoisesti paremman suorituskyvyn vuoksi.

Lopuksi voidaan todeta, että säiliöiden kohdalla on kohtuullisen helppo sanoa, mitä säiliöitä tulisi käyttää oletusarvoisesti, jotta tyydyttävä suorituskyky voidaan saavuttaa. Kuitenkin äärimmäistä suorituskykyä vaativissa tilanteissa säiliöt pitää valita määrätietoisemmin. On tärkeä muistaa, että paras tapa selvittää mikä säiliö on tehokkain jossain käyttötilanteessa on testata ja mitata säiliön suorituskykyä juurikin siinä käyttötilanteessa.

Aiheesta voisi tehdä paljon jatkotutkimusta. Säiliöiden suorituskykyä voisi tutkia yhä kattavammin ja mm. profiloida operaatioiden käyttäytymistä välimuistin kannalta siihen tarkoitetuilla työkaluilla. Aikakompleksisuuden rinnalla myös säiliöiden tilakompleksisuutta voisi tutkia.

LÄHDELUETTELO

[1] *C++ Reference*. 3. elokuuta 2018. URL: https://en.cppreference.com/w/cpp/container (viitattu 28. 03. 2019).

[2] *C++17 final working draft*. 21. maaliskuuta 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf (viitattu 28. 03. 2019).

[3] T. Cormen, C. Leiserson, R. Rivest ja C. Stein. *Introduction to Algorithms, Third Edition*. 2009, 229–338.

[4] *Doubly Linked list*. 15. kesäkuuta 2007. URL: https://commons.wikimedia.org/wiki/File:Doubly-linked-list.svg (viitattu 07. 04. 2019).

[5] *EECS 311: STL Containers*. URL: http://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-summary.html (viitattu 30. 03. 2019).

[6] R. Gerber, A. Bik, K. Smith ja X. Tian. *The Software Optimization Cookbook*. 2011, 107–112.

[7] *Hash Table*. 10. huhtikuuta 2009. URL: https://commons.wikimedia.org/wiki/File:Hash_table_5_0_1_1_1_1_1_LL.svg (viitattu 07. 04. 2019).

[8] *Red-Black Tree*. 30. joulukuuta 2006. URL: https://commons.wikimedia.org/wiki/File:Red-black_tree_example.svg (viitattu 07. 04. 2019).

[9] B. Stroustrup. *Programming Principles and Practice Using C++*. 2016, 720–724.