Tampereen yliopisto

Jesse Kokkonen

C++-STANDARDIKIRJASTON JA BOOST.CONTAINER-KIRJASTON SÄILIÖT

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Huhtikuu 2021

TIIVISTELMÄ

Jesse Kokkonen: C++-standardikirjaston ja Boost.container-kirjaston säiliöt
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikka, TkK
Huhtikuu 2021

Nykypäivänä ohjelmointikielille on tarjolla useita erilaisia kirjastoja, jotka sisältävät ohjelmointia nopeuttavia ja helpottavia ominaisuuksia. Näistä merkittäviä ovat esimerkiksi valmiiksi ohjelmoidut, heti käyttövalmiit säiliöt. C++-standardikirjasto ja Boost.container-kirjasto tarjoavat useita eri käyttötilanteisiin sopivia säiliöitä. C++-standardikirjasto on tarkkaan määritelty ja se noudattaa standardia, kun taas Boost.container on avoimeen lähdekoodiin perustuva kirjasto. Boost.container-kirjaston lähdekoodi on vapaassa jaossa Boost Software License -lisenssin alla. Tällä lisenssillä kuka tahansa saa käyttää, muokata ja jakaa Boost.container-kirjastoa vapaasti.

Tämän työn tarkoituksena on tutkia C++-standardikirjaston ja Boost.container-kirjaston tarjoamia säiliöitä, ja vertailla niiden suorituskykyä erilaisissa tilanteissa. C++-standardikirjasto on laajemmin tunnettu ja käytetty kirjasto, joten työssä keskitytään erityisesti löytämään tilanteita, joissa Boost.container-kirjaston säiliöt toimivat standardikirjaston säiliöitä tehokkaammin.

Työssä käydään aluksi läpi muutamia alkeellisia tietorakenteita, joihin molempien kirjastojen säiliöt perustuvat. Sen jälkeen suoritetaan katsaus C++-standardikirjaston säiliöihin ja iteraattoreihin. Tämän jälkeen tutkitaan Boost.container-kirjaston säiliöitä, jotka muistuttavat C++-standardikirjaston säiliöitä. Lopuksi säiliöitä vertaillaan niille saatavilla olevien operaatioiden asymptoottisten aikakompleksisuuksien avulla, ja niille etsitään optimaalisia käyttötilanteita. Katsauksesta selviää, että kirjastojen säiliöitä yhdistävät lähinnä nimeämiskäytännöt, sillä niiden sisäiset toteutukset todetaan toisistaan hyvin poikkeaviksi.

Työn aikana löydettiin useita tilanteita, joissa vähemmän tunnetusta Boost.container-kirjastosta löytyvä säiliö toimii tehokkaammin kuin jokin standardikirjaston säiliöistä. Löydöksiä tukevat työn aikana suoritetut säiliöiden operaatioiden aikakompleksisuuksien vertailut ja aiheesta löydetty kirjallisuus.

Avainsanat: C++, STL, Boost, säiliö, kirjasto

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

IEC International Electrotechnical Commission

ISO Kansainvälinen standardointiorganisaatio

STL Standard Template Library

1. JOHDANTO

Ohjelmoinnissa tulee usein eteen tilanne, jossa ohjelman tarvitsee tallentaa tietoa, ja siihen on päästävä käsiksi myöhemmin ohjelman suorituksen aikana. Siinä missä ennen ohjelmoijan on täytynyt joka kerta ohjelmoida tarkoitukseen sopiva tietorakenne itse, on nykyään tarjolla runsaasti erilaisia valmiiksi luotuja tietorakenteita. Ne ovat valmiita malleja tallentaa tietokoneen käyttämää tietoa monissa eri muodoissa, ja niitä tarjotaan useimmille kielille erilaisissa kirjastoissa.

Tässä työssä tarkastellaan C++:n standardikirjastoon kuuluvaa Standard Template Library -kirjastoa ja avoimeen lähdekoodiin perustuvaa Boost.container-kirjastoa, joka on osa suurempaa Boost-kirjastoa. Työssä perehdytään erityisesti Boost.container kirjaston tarjoamiin säiliöihin, jotka pohjautuvat STL:ään, mutta eivät eroavien ominaisuuksiensa myötä täytä kirjoitushetkellä voimassaolevaa ja STL:n määrittelevää ISO/IEC 14882:2020 -standardia [1]. Työssä keskitytään löytämään tilanteita, joissa Boost.container kirjaston tarjoamilla tietorakenteilla saavutetaan parannuksia ohjelmakoodin tehokkuuteen.

Toisessa luvussa käsitellään tämän työn kannalta oleellinen teoria tietorakenteista, niiden toteutuksista, läpikäynnistä ja näihin perustuvista standardikirjaston säiliöistä. Kolmannessa luvussa esitellään Boost.container-kirjaston säiliöt ja niiden toteutukset kirjoitushetkellä uusimmassa Boost.container-kirjaston versiossa 1.75. Neljännessä luvussa etsitään lähteiden ja läpikäydyn teorian pohjalta kullekin tietorakenteelle sopivia käyttötapauksia ja uusien ominaisuuksien aiheuttamia muutoksia kunkin tietorakenteen tehokkuudessa. Viidennessä luvussa tehdään yhteenveto, jossa käydään läpi työssä ilmenneet tulokset ja pohditaan mahdollisia jatkotutkimuksia aiheesta.

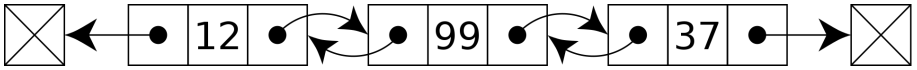
2. STANDARDIKIRJASTON SÄILIÖT

C++-standardin mukaan määriteltävät tietorakenteet perustuvat alkeelliseempiin tietorakenteisiin. Tämän työn kannalta merkittävät alkeelliset tietorakenteet ovat C-taulukko, linkitetty lista ja puu. Seuraavaksi esitellään niiden taustalla oleva teoria ja niistä johdetut standardikirjaston säiliöt `array`, `vector`, `forward_list`, `list`, `deque`, `map` ja `set`.

2.1 Taustalla olevat tietorakenteet

C-taulukko on tietorakenne, joka koostuu etukäteen määriteltävästä määrästä alkioita. Nimitys C-taulukko tulee siitä, että säiliö on lähtöisin C-kielestä ja kuuluu itse kieleen eikä ole peräisin esimerkiksi standardikirjastosta. C-taulukolla itsellään on tieto sen maksimikoosta ja sen alkioista. Jokainen C-taulukon alkio on indeksoitu ja tämä määrittelee alkion paikan C-taulukossa. C-taulukko siis sisältää korkeintaan sen maksimimäärän verran samankokoisia ja peräkkäisiä muistipaikkoja. C-taulukko voi sisältää myös taulukoita alkioinaan ja tällöin se on moniulotteinen taulukko. [1][2]

Linkitetty lista on tietorakenne, jossa alkiot eivät välttämättä ole muistissa vierekkäin. Linkitetyn listan alkiot sisältävät itse tiedon lisäksi yhden tai kaksi osoitinta riippuen listan määrittelystä, eli onko se yksisuuntainen, kaksisuuntainen vai renkaaksi linkitetty lista. Kuvassa 2.1 on esitetty kokonaisluvuista koostuva kahteen suuntaan linkitetty lista. Työn kannalta riittää tarkastella kaksisuuntaisia, ei renkaaksi linkitettyjä listoja. Niiden alkiot sisältävät osoittimet edellisen ja seuraavan alkion muistiosoitteeseen. Linkitetyn listan toteutustavasta johtuen sinne voidaan lisätä ja sieltä voidaan poistaa alkioita muuttamatta jo olemassa olevien alkioiden sijaintia muistissa.



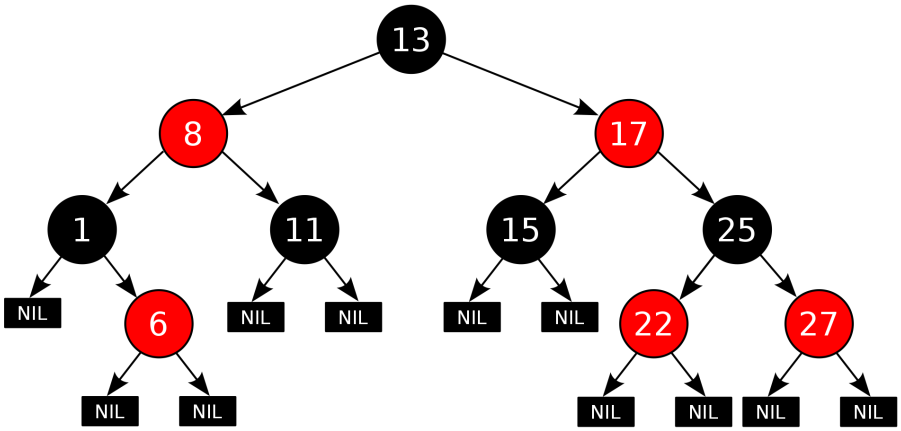
Kuva 2.1. *Kahteen suuntaan linkitetty lista. [3]*

Tavallisen puurakenteen alkiot kuvataan solmuina, jotka on linkitetty toisiinsa hierarkkisesti. Puun solmuilla sanotaan olevan vanhempia ja lapsia, mikä muodostaa tämän hierarkian. Puun ensimmäistä solmua sanotaan juureksi, ja tästä lähteviä lapsia sen alipuiksi. Tätä voidaan jatkaa rekursiivisesti, kunnes saavutetaan niin sanottu lehti eli solmu, jolla ei ole lapsia. Tyypillisesti käytetään binääristä hakupuuta eli puuta, jonka solmulla voi olla

enintään kaksi lasta, ja solmusta vasemmalle lähtevässä alipuussa on vain sitä pienempiä alkioita. Tätä vastaavasti oikealle lähtevässä alipuussa on vain alkiota suurempia alkioita. Tavallisen puun ongelmakohta on se, että alkioiden lisäysjärjestys voi vaikuttaa puun tehokkuuteen merkittävästi. Mikäli alkiot lisätään puuhun suuruusjärjestyksessä, jokaista alkiota seuraava alkio lisätään oikeanpuoleiseksi alkioksi. Tällöin jokaisen alkion osoitin vasempaan lapseen jää tyhjäksi, ja puu ketjuuntuu käytännössä linkitetyksi listaksi. Hakeminen tällaisesta puusta on asymptoottisesti lineaarista suhteessa alkioiden määrään, kun optimaalisessa tilanteessa hakeminen on asymptoottiselta tehokkuudeltaan logaritmista. Tasapainottamalla puuta saavutetaan siis mahdollisesti suuria etuja hakualgoritmien tehokkuudessa.

Tasapainotettu puu on erikoistapaus tavallisesta puurakenteesta. Punamusta puu on yleinen tapa toteuttaa tasapainotettu puu. Kuvassa 2.2 on esitetty tasapainossa oleva punamusta puu, johon on eksplisiittisesti piirretty puun varsinaisista lehdistä lähtevät osoittimet NIL-solmuihin. Varsinaisessa tietorakenteen toteutuksessa nämä eivät ole tarpeen, vaan esimerkiksi NULL-osoitin lapsiosoittimena osoittaa solmun olevan lehti. Punamustassa puussa solmu sisältää arvon ja osoittimien lisäksi tiedon solmun väristä, joka on punainen tai musta. Näiden värien käytölle on luotu seuraavanlaiset ehdot, jotta puu olisi punamusta binääripuu:

1. Jokainen solmu on musta tai punainen.
2. Juuri on musta.
3. Jokainen lehti on musta.
4. Jos solmu on punainen, niin molemmat sen lapsista ovat mustia.
5. Jokainen solmusta lehtiin lähtevä polku sisältää yhtä monta mustaa solmua. [4]



Kuva 2.2. *Punamusta puu. [5]*

Kun puu täyttää nämä ehdot, voidaan sen sanoa olevan tasapainossa oleva punamusta puu.

2.2 Sarjasäiliöt

C++-standardin mukaan sarjasäiliöt sisältävät äärellisen määrän saman tyyppisiä alkioita lineaarisessa järjestyksessä. Edellisessä luvussa esitellyistä tietorakenteista C-taulukko ja linkitetty lista täyttävät tämän ja muut standardissa esitetyt ehdot, joten niitä voidaan pitää sarjasäiliöinä. Sarjasäiliöille on määritelty vaatimuksia, jotka jokaisen sarjasäiliön on toteutettava. Nämä ovat alustus-, lisäys-, poisto- ja tyhjennysoperaatiot. Sarjasäiliöille on myös määritelty vapaaehtoisia vaatimuksia, jotka ovat indeksointi, viittaukset ensimmäiseen ja viimeiseen alkioon, sekä lisäys ja poisto alkuun tai loppuun. [1]

Standardikirjaston taulukko (`std::array`) on vahvasti C-taulukkoon pohjautuva tietorakenne. C-taulukon tavoin sillä on maksimikoko ja kaikki sen alkiot on alustettava taulukon luontihetkellä. Taulukon käyttöä suositellaan välttämään modernissa C++ ohjelmoinnissa, ja sen sijaan suositellaan käytettäväksi seuraavaksi esiteltävää vector-rakennetta. [6]

Vektori (`std::vector`) on sarjasäiliö, joka pohjautuu C-taulukkoon. Toisin kuin C-taulukko, vektorin koko ei ole etukäteen määritelty, vaan vektori varaa käyttöönsä lisää muistia sen täyttyessä. Vektori hallitsee muistinkäyttöään itse, mutta sitä voidaan ohjata esimerkiksi varaamaan tarpeeksi muistia etukäteen tehokkuuden parantamiseksi. Vektorin alkiot ovat muistissa lineaarisesti ja se tukee amortisoidusti vakioaikaista lisäystä ja poistoa sen loppuun. [1] Tämä tarkoittaa sitä, että itse vektorin loppuun lisääminen on vakioaikaista, mutta sen muistin täyttyessä vanhojen alkioiden siirtämiseen kuluu lineaarinen aika suhteessa niiden lukumäärään. Vektorin kasvaessa tämä aika jaettuna jokaiselle lisäysoperaatiolle lähestyy vakioaikaista, ja tästä tulee nimitys amortisoidusti vakioaikainen. Vektori täyttää sarjasäiliöiden vaatimusten lisäksi myös lähes kaikki sarjasäiliöiden vapaaehtoiset ominaisuudet. Vektorilta puuttuu näistä lisäys- ja poisto-operaatiot alkuun. [1][6]

Standardikirjastosta löytyy `std::forward_list`, joka on yhteen suuntaan linkitetty lista eli yksisuuntainen lista. Se tukee iterointia yhteen suuntaan ja vakioaikaista alkioiden lisäystä ja poistoa mihin tahansa kohtaan. Yksisuuntaisen listan toteutus standardikirjastossa huolehtii itse omasta muistinhallinastaan. Yksisuuntainen lista täyttää standardin määritelmän säiliöstä muuten, mutta se ei sisällä `size` -jäsenfunktiota. Minkä tahansa yksisuuntaisen listan alkion muokkaamiseksi, tai sellaisen lisäämiseksi, tarvitaan pääsy sitä edeltävään alkioon. Yksisuuntainen lista on optimoitu lyhyille sarjoille, joita iteroidaan aina niiden alusta asti.[1][6]

Standardikirjastosta löytyy myös kaksisuuntainen lista (`std::list`). Se tukee iterointia kahteen suuntaan ja vakioaikaista alkioiden lisäystä ja poistoa mihin tahansa kohtaan. Kaksisuuntaisen listan toteutus standradikirjastossa huolehtii itse omasta muistinhallinnastaan. Kaksisuuntainen lista täyttää standardin määrittelemien vaatimusten lisäksi lähes kaikki vapaaehtoiset sarjasäiliön ominaisuudet. Näistä puuttuvat indeksointioperaatiot `[]` ja `at`. [1][6]

Pakka (`std::deque`) on sarjasäiliö, joka toimii kaksipäisenä jonona. Pakka tukee vakioaikaista lisäystä ja poistoa alkuun ja loppuun. Alkion lisääminen keskelle vie lineaarisen ajan. Pakka on tyypillisesti sisäiseltä toteutukseltaan kokoelma eri muistialueilla sijaitsevia taulukoita.[1] Pakka on siis optimoitu alkioiden lisäykseen tai poistoon sen alkuun ja loppuun. Pakka täyttää kaikki sarjasäiliöiden ominaisuudet ja sillä on kaikki sarjasäiliön vapaaehtoiset ominaisuudet [1].

2.3 Assosiatiiviset säiliöt

Assosiatiiviset säiliöt liittävät varsinaisen tiedon yhteyteen avaimen, jonka perusteella tieto järjestetään ja käsitellään. Assosiatiiviset säiliöt tukevat nopeita hakuja näillä avaimilla. Jotta ohjelmoijan itse määrittelemiä tyyppejä voitaisiin käyttää assosiatiivisessä säiliössä avaimena, tarvitsee niille kirjoittaa järjestyksen määrittelevä `Compare(k1, k2)` jäsenfunktio tai antaa se säiliön rakentajalle parametrina. [1]

Standardikirjaston `map` on assosiatiivinen tietorakenne, joka on lähes aina toteutettu luvussa 2.1 esiteltynä punamustana binääripuuna. Tällöin siihen tehtävät haut avaimien perusteella ovat asymptoottiselta tehokkuudeltaan $O(log\ n)$. Standardikirjaston `map` käyttää yksikäsitteisiä avaimia eli jokainen sen avaimista on olemassa vain kerran. [1]

Standardikirjaston `set` on toteutukseltaan lähes samanlainen tietorakenne kuin edellä esitelty `map`. Ero `set`- ja `map`-rakenteen välillä on niiden sisältämässä tiedossa. Avaimeen ei ole liitetty varsinaista tietoa `set`-rakenteessa, vaan avain itse voidaan ajatella olevan tieto. [6] Jos ohjelmoija lisää omia tietotyyppejään `set`-rakenteeseen, näille on välttämätöntä kirjoittaa niiden järjestyksen määrittävä funktio, jotta iteraattori osaa käydä alkiot järjestyksessä läpi. Tästä syystä `set`-rakenteessa olevaa tietoa ei voida myöskään muuttaa; tieto itsessään on avain ja avaimia ei tule muokata.

Jos `map`- tai `set`-rakenteeseen tarvitsee tallentaa eri tietoa samoilla avaimilla, on tätä varten standardissa määritelty tietorakenteet `multimap` ja `multiset`. Näiden ero aiemmin esiteltyihin vastineisiinsa on se, että ne tukevat useita samanarvoisia avaimia. Molempien tapauksessa `find` palauttaa osoittimen *johonkin* hakua vastaavista avaimista, tyypillisesti ensimmäiseen, mikäli avain löytyy. Kaikkien hakuun sopivien avaimien hakemiseen on oma metodinsa `equal_range`. [1][2]

2.4 Iteraattorit

Edellisessä luvussa esiteltyjä tietorakenteita halutaan usein kulkea alusta johonkin alkioon tai loppuun asti. Tätä kutsutaan tietorakenteiden iteroinniksi, ja standardikirjasto määrittelee kuusi eri tyyppistä iteraattoria. Nämä tyypit ovat `input`, `output`, `forward`, `bidirectional`, `random_access` ja `contiguous`. Yleisesti termillä iteraattori viitataan vain input- ja output-iteraattoreihin. [1] Iteraattoriluokat määrittelevät yhteisen rajapinnan,

jota voidaan hyödyntää kaikkiin standardikirjaston säiliöihin. Tällä tavalla toteutukseltaan samanlaiset algoritmit saadaan iteraattoreita hyödyntämällä toimimaan eri tietorakekenteille ilman merkittäviä muutoksia itse algoritmiin.

Iteraattoreita käsiteltäessä on tärkeää huomioida niiden mahdollinen mitätöityminen. Esimerkiksi vektorin muistin täyttyessä vektori siirtää alkionsa uuteen, niille varattuun suurempaan muistialueeseen. Tällöin iteraattorit jäävät osoittamaan edelliselle muistialueelle ja ne mitätöityvät, koska ne eivät enää osoita oikeaan alkioon. Taulukossa 2.1 on lueteltu, missä tilanteissa iteraattorit aiemmin esiteltyihin tietorakenteisiin ovat valideja.

Taulukko 2.1. *Iteraattoreiden mitätöityminen. [2]*

Säiliö	Lisäys	Poisto	Ehto
Vektori	Ei	N/A	Muistia varataan lisää
	Validi	Validi	Sijaitsee ennen muutettuja alkioita
	Ei	Ei	Sijaitsee muutettujen alkioiden jälkeen
Kaksisuuntainen lista	Validi	Validi, pl. poistetut	
Yksisuuntainen lista	Validi	Validi, pl. poistetut	
map ja multimap	Validi	Validi, pl. poistetut	
set ja multiset	Validi	Validi, pl. poistetut	

Kaksi iteraattoria määrittelevät tietorakenteesta sarjan (engl. sequence) lopusta avoimella välillä $[begin, end)$. Tässä `begin` tarkoittaa sarjan ensimmäiseen alkioon osoittavaa iteraattoria ja `end` sarjan viimeistä seuraavaan alkioon osoittavaa iteraattoria. [6] Kaikki säiliöt tarjoavat nämä metodit kyseisten iteraattoreiden käyttöä varten [1]. Jokainen säiliö muodostaa siis alkioistaan sarjan, joka voidaan kulkea läpi hyödyntämällä alku- ja loppuosoittimia. Iteraattorit ovat pohjimmiltaan osoittimia, joita käsitellään iteraattoriluokkien tarjoamien rajapintojen kautta. Koska osoittimille on määritelty toiminnot ++ ja $*$, löytyvät ne myös iteraattoreilta. Näistä ensimmäisellä iteraattori voidaan siirtää osoittamaan sarjassa sitä seuraavaan alkioon. Jälkimmäinen toiminto antaa mahdollisuuden lukea iteraattorin päässä olevan alkion tietoja.

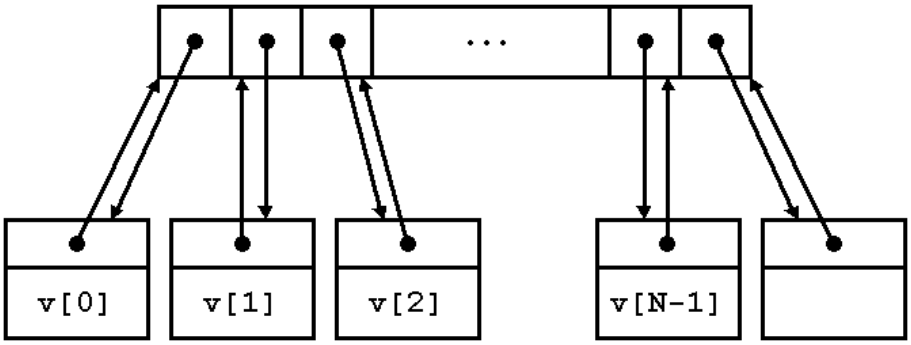
3. BOOST.CONTAINER SÄILIÖT

Boost on C++ standardia *myötäilevä* kirjasto. Sen lähdekoodi on avoimessa jakelussa Boost Software License -lisenssin alla. Se mahdollistaa kenen tahansa käyttää, muokata ja jakaa kirjastoja vapaasti. Boostin tarjoamat kirjastot ovat riippumattomia alustasta ja tukevat useimpia kääntäjiä. Niinsanottu Boost-yhteisö on vastuussa kirjaston kehityksestä ja jakelusta. Yhteistön tavoite on kehittää ja koota korkealaatuisia kirjastoja jotka täydentävät standardikirjaston jättämiä aukkoja.[7] Monet standardikirjaston ominaisuudet ovat peräisin Boost-kirjastosta, josta ne on myöhemmin otettu mukaan standardiin ja sitä myötä standardikirjastoon. Muunmuassa standardikirjaston kirjastot älykkäille osoittimille (engl. smart pointer), säikeille (engl. thread) ja monikoille (engl. tuple) ovat saaneet alkunsa Boostin kirjastoista. [8]

Boost.container on yksi Boost-kirjaston tarjoamista kirjastoista. Boost.container sisältää tietorakenteita, joita löytyy myös standardikirjastosta. Näiden ero on kuitenkin siinä, että Boost.container-kirjaston säiliöt tarjoavat enemmän joustavuutta tai uusia ominaisuuksia. [7] Tämä tullaan huomaamaan myöhemmin etenkin stabiilin vektorin käytössä. Kaikki Boost.container-kirjaston säiliöt tukevat keskeneräisiä tyyppejä (engl. incomplete type) ja niitä voidaan käyttää rekursiivisten säiliöiden määrittelyyn [7]. Rekursiivinen säiliö sallii esimerkiksi luokan tapauksessa sen itsensä käyttämisen jäsenmuuttujanansa. Kuten aiemmin todettu, kaikki Boost.container-kirjaston säiliöt eivät uusien ominaisuuksiensa myötä täytä standardikirjaston määrittävää standardia, joten ne eivät ole siinä mukana.

Ensimmäisenä esiteltävä Boost.container-kirjaston tietorakenne on `stable_vector` eli stabiili vektori. Se on yhdistelmä standardikirjaston vektoria ja kaksisuuntaista listaa. Nimensä mukaisesti stabiili vektori on stabiili. Sen standardikirjaston vastine, vektori, on epästabiili. Epästabiilius tarkoittaa, että osoittimet ja iteraattorit säiliön alkioihin voivat mitätöityä. Yksi standardikirjaston stabiileista tietorakenteista on kaksisuuntainen lista, ja stabiilissa vektorissa onkin hyödynnetty sen toteutustapaa.[9] Tämä toteutus on esitetty kuvassa 3.1

Jokainen stabiilin vektorin alkio on tallennettu omaan solmuunsa. Nämä solmut vastaavat listan solmuja. Jokaista solmua käsitellään osoittimilla, jotka on sijoitettu vierekkäin taulukkoon (engl. contiguous array). Jokaisesta solmusta lähtee myös niin sanottu ylösosoitin, joka osoittaa takaisin solmun sijaintikohtaan taulukossa. Stabiilin vektorin iteraat-



Kuva 3.1. *Stabiilin vektorin toteutus. [10]*

torit osoittavat itse solmuihin. Koska näiden solmujen paikka muistissa ei koskaan muutu, iteraattorit säilyvät valideina. Stabiiliuden saavuttamiseksi stabiilista vektorista on pudotettu alkioiden peräkkäisyys (engl. element contiguity) muistissa. Stabiili vektori täyttää kaikki C++-standardin vaatimukset sarjasäiliöille, ja sillä on kaikki standardikirjaston vectorin vapaaehtoiset ominaisuudet. Stabiilin vektorin operaatioiden asymptoottiset $O()$-suoritusajat ovat täysin vastaavat standardikirjaston vectorin operaatioiden kanssa. [9]

Boost.container-kirjaston staattinen vektori (`static_vector`) on yhdistelmä standardikirjaston vektoria ja taulukkoa. Se on sarjasäiliö, jolle on taulukon tavoin määritelty maksimikoko. Ero standardikirjaston taulukkoon tulee alkioiden alustuksessa. Standardikirjaston taulukon täytyy alustaa jokainen sen alkio silloin, kun taulukko luodaan. Staattinen vektori sen sijaan voi alustaa alkioitaan dynaamisesti silloin, kun niitä lisätään sinne. Staattinen vektori tarjoaa myös satunnaisen pääsyn (engl. random access) eli indeksoinnin, vakioaikaisen lisäyksen ja poiston loppuun sekä lineaariaikaisen lisäyksen ja poiston alkuun tai keskelle. [9]

Pieni vektori (`small_vector`) on Boost.container-kirjaston tietorakenne, joka on optimoitu pienelle määrälle alkioita. Se sisältää joitain etukäteen alustettuja alkioita valmiiksi, mikä mahdollistaa dynaamisen muistin varaamisen välttämisen silloin, kun varsinaisten alkioiden määrä on alle etukäteen määritellyn maksimin. Staattisesta vektorista poiketen pienen vektorin kapasiteetti voi kasvaa yli etukäteen määritetyn maksimin.

Boost.container-kirjastosta löytyvä kaksisuuntainen vektori (`devector`) on standardikirjaston vektorin ja pakan yhdistelmä. Kaksisuuntainen vektori tarjoaa standardikirjaston pakan ominaisuuksista lisäyksen tai poiston alkuun ja loppuun.[9] Standardikirjaston pakka tarjoaa luvun 2.2 mukaisesti nämä ominaisuudet täysin vakioaikaisena. Kaksisuuntaisessa vektorissa nämä ovat amortisoidusti vakioaikaisia, koska kaksisuuntainen vektori tarjoaa myös standardikirjaston vektorin ominaisuuksia. Esimerkki tällaisesta vektorin ominaisuudesta on alkioiden vierekkäisyys muistissa. Toisin kuin standardikirjaston vektori, kaksisuuntainen vektori sisältää vapaita muistipaikkoja myös sen alkupäässä ennen varsinaisia alkioita. Tämä mahdollistaa tehokkaan toteutuksen metodeille, jotka muokkaavat kaksisuuntaista vektoria sen alusta. Yleisesti ottaen kaksisuuntaisen vektorin käytös-

sä olevat metodit ovat standardikirjaston vektorin metodien ylijoukko, joilla on samanlaiset toiminnallisuudet. Tähän poikkeuksena on iteraattoreiden mitätöitymisiin liittyvät varmistukset. [9]

Boost.container-kirjaston `slist` on yhteensuuntaan linkitetty lista. Poiketen standardikirjaston yksisuuntaisesta listasta Boost.container-kirjaston versio tarjoaa `size` metodin, ja `splice` metodi vaatii lineaarisen suoritusajan. Muut listojen ominaisuudet ovat samanlaiset, joten niiden voidaan sanoa täydentävän toisiaan. [9]

Standardikirjastosta eniten poikkeavat Boost.container-kirjaston tarjoamat litteät assosiatiiviset säiliöt. Litteällä tarkoitetaan tässä tapauksessa sitä, että standardikirjastosta esitellyt assosiatiiviset säiliöt on toteutettu puurakenteina, mutta Boost.container-kirjaston assosiatiiviset säiliöt on ikään kuin litistetty puurakenteesta järjestetyn vectorin tyyliseksi yhdeksi yhtenäiseksi muistiosuudeksi. Boost.container-kirjasto tarjoaa litteinä assosiatiivisina säiliöinään litteitä `(multi)map`- ja litteitä `(multi)set`-rakenteita. [9]. Boost.container-kirjaston tapauksessa esimerkiksi litteän `multimap`- ja litteän `map`-rakenteen ero on on sama kuin standardikirjastossa, eli litteä `multimap` tukee useita samanarvoisia avaimia.

Litteät assosiatiiviset säiliöt hyödyntävät yhtä tietotekniikan perusalgoritmeista, binäärihakua. Säiliöt siis säilyttävät avaimiaan järjestetyssä vektorissa. Tämä mahdollistaa $O(log\ n)$ kompleksiset haut samoin kuin binäärihakupuut, mutta vektori käyttää lähes kolme kertaa vähemmän muistia binäärihakupuuhun verrattuna. Tällä kuitenkin on oma ehtonsa. Lisääminen järjestettyyn vektoriin vie $O(n)$ kompleksisen ajan, kun lisäys binäärihakupuuhun vie $O(log\ n)$ kompleksisen ajan.[11]

4. SÄILIÖIDEN VERTAILU

Seuraavaksi vertaillaan luvuissa 2 ja 3 esiteltyjä tietorakenteita ja pohditaan kullekkin sopivimpia käyttötarkoituksia. Vertailu perustuu esiteltyyn teoriaan ja lähteistöön. Oikeanlaisen tietorakenteen valitseminen on ohjelman tehokkuuden kannalta tärkeä valinta. Tämä pätee etenkin silloin, kun käsitellään suuria määriä dataa tai ohjelmalta vaaditaan lähes reaaliaikaista laskentaa pienemmilläkin datamäärillä. On kuitenkin huomattava, että oikeasti reaaliaikainen laskenta on mahdotonta, vaikka puhuttaisiin reaaliaikaisuudesta. Taulukkoon 4.1 on koottu aiemmin esiteltyjen tietorakenteiden eri operaatioiden aikakompleksisuudet.

Taulukko 4.1. *Säiliöiden operaatioiden aikakompleksisuudet. [6][9]*

Standardikirjasto	Indeksointi	Alku	Loppu	Väli
taulukko	O(1)			
vektori	O(1)		O(1)	O(n)
yksisuuntainen lista		O(1)		O(1)
kaksisuuntainen lista		O(1)	O(1)	O(1)
pakka	O(1)	O(1)	O(1)	O(n)
(multi)map	O(log n)			O(log n)
(multi)set	O(log n)			O(log n)
Boost.container-kirjasto	Haku	Alku	Loppu	Väli
stabiili vektori	O(1)		O(1)	O(n)
staattinen vektori	O(1)		O(1)	O(n)
pieni vektori	O(1)		O(1)	O(n)
kaksisuuntainen vektori	O(1)	O(1)	O(1)	O(n)
yksisuuntainen lista		O(1)		O(1)
litteä (multi)map	O(log n)			O(n)
litteä (multi)set	O(log n)			O(n)

Taulukossa tyhjä merkintä tarkoittaa, että operaatiota ei ole kyseiselle säiliölle. "Väli" tarkoittaa alkion lisäystä tai poistoa säiliön keskelle. Assosiatiivisten säiliöiden tapauksessa alku- ja loppuoperaatioita ei ole huomioitu erikseen, koska alkioiden lisäykseen on tarjolla vain yksi metodi. Alku-, loppu- ja välioperaatiot sisältävät lisäyksen ja poiston. Alku- ja

loppuoperaatioilla tarkoitetaan erityisesti sitä, että säiliö tarjoaa suoraan metodin lisätä tai poistaa alkio näille paikoille. Esimerkiksi standardikirjaston yksisuuntaisen listan tapauksessa loppuun on mahdollista lisätä alkio, mutta sille ei ole suoraa metodia.

4.1 Sarjasäiliöiden vertailu

Aloitetaan käsittely standardikirjaston vektorista. Vektori on standardikirjaston ensisijainen tietorakenne, eli sitä suositellaan käyttämään ensimmäisenä vaihtoehtona. Mikäli ensimmäiseksi vaihtoehdoksi ajatellaan standardikirjaston listaa tai taulukkoa, tulisi C++-kielen luojan Bjarne Stroustrup mukaan asiaa tarkastella uudestaan. [1][6] Vektori on tehokkuudeltaan suorituskykyinen, taulukkoa muistuttava, helppokäyttöinen tietorakenne, joten se on ansainnut paikkansa standardikirjaston ensisijaiseksi suositeltuna tietorakenteena.

Standardikirjaston taulukko on tietorakenteena vanha, ja sen käyttöä suosivat tilanteet ovat hyvin spesifejä. Jo ohjelman käännösaikana tulee olla tiedossa, että alkioiden määrä tulee pysymään samana, ja että alkioiden sisältöä tullaan tarvitsemaan koko ohjelman suorituksen ajan.

Yksisuuntaisista listoista todettiin, että standardikirjaston versiossa ei ole `size` operaatiota. Tällä saatiin aikaiseksi vakioaikainen `splice` operaatio, jolla voidaan siirtää listan alkioita toiseen. Boost.container-kirjaston versio sisältää nämä operaatiot päinvastoin. Standardikirjaston yksisuuntainen lista siis toimii paremmin tilanteessa, jossa käsitellään useita eri listoja ja niiden alkioita tarvitsee yhdistellä usein. On myös huomattava, että jos missään vaiheessa ohjelman suoritusta tulisi listaan lisätä alkioita *ennen* tiettyä alkiota, on standardikirjaston tarjoama kaksisuuntainen lista parempi vaihoehto.

Kuten luvussa 2 todettiin, stabiili vektori sisältää samat ominaisuudet kuin standardikirjaston vektori. Se kuitenkin tarjoaa stabiilit iteraattorit, joista on hyötyä esimerkiksi tilanteessa, jossa rakenteeseen tarvitsee lisätä ja poistaa alkioita jatkuvasti, mutta sinne osoittavia iteraattoreita hyödynnetään näiden operaatioiden välissä. Myös tilanne, jossa rakenteen sisältämät alkiot ovat suuria, ja täten siirtyvät muistissa hitaammin, tukee stabiilin vektorin käyttöä, koska stabiilin vektorin varsinaiset alkiot eivät missään vaiheessa liiku muistissa [9].

Boost.container-kirjaston staattisen vektorin kerrottiin sisältävän dynaamisen määrän samantyyppisiä alkioita, mutta alkioille oli maksimimäärä. Tämä tekee staattisesta vektorista erinomaisen ehdokkaan esimerkiksi puskuriksi, tai käytettäväksi jonkin muun luokan sisäisen ominaisuuden toteutuksessa. Se on myös tehokas vaihtoehto kun tiedetään, että alkioita lisätään ja poistetaan tietorakenteeseen, mutta tietorakenteen samaan aikaan sisältämille alkioille on tietty, etukäteen määritelty maksimi. [9] On kuitenkin huomioitava, että staattinen vektori nostaa ajonaikaisen virheen, mikäli sinne yritetään lisätä enemmän

alkioita kuin sen maksimi on [12]. Se siis jättää kapasiteettinsa valvonnan ohjelmoijan vastuulle.

Boost.container-kirjaston pienen vektorin todettiin olevan optimoitu pienelle määrälle alkioita. Pieni määrä alkioita tarkoittaa samaa, myös staattiselle vektorille annettavaa maksimiarvoa. Selvä käyttötapaus sille on siis silloin, kun alkioiden määrän tiedetään pysyvän alle maksimimäärän. Toisin kuin staattinen vektori, pieni vektori pystyy varaamaan muistia enemmän, kuin mitä se käännösaikana varasi. [9]

Standardikirjaston pakan ja Boost.container-kirjaston kaksisuuntaisen vektorin erona on se, että pakka koostuu sisäisesti useista taulukoista, kun kaksisuuntaisen vektorin alkiot sijaitsevat vierekkäin muistissa. Molemmat ovat siis hyviä valintoja kaksipäistä jonorakennetta tarvittaessa, mutta mikäli rakennetta tarvitsee käsitellä osoittimia liikuttelemalla (engl. offsetting a pointer), on kaksisuuntainen vektori parempi valinta. Tämä on sen takia, että kaksisuuntainen vektori takaa alkioiden vierekkäisyyden muistissa.

Lopuksi tiivistettynä Boost.container-kirjaston sarjasäiliöillä saavutetaan selkeitä etuja seuraavissa tilanteissa:

- vektorin iteraattoreiden validius halutaan varmistaa
- säiliön sisältämät alkiot ovat suurikokoisia, eikä niitä haluta liikutella muistissa
- halutaan käyttää puskurina sellaiseksi optimoitua säiliötä
- tarvitaan kaksipäistä jono-rakennetta, jota täytyy käsitellä osoittima liikuttelemalla.

4.2 Assosiatiivisten säiliöiden vertailu

Viimeisenä vertaillaan kirjastojen tarjoamia assosiatiivisia säiliöitä. Kirjastojen sisäisesti niiden toteutukset ovat riittävän lähellä toisiaan, joten niitä voidaan käsitellä yhtenä. Standardikirjaston assosiatiiviset rakenteet käyttävät luvun 2 mukaisesti punamustaa binäärihakupuuta avaimiensa säilyttämiseen. Boost.container-kirjaston assosiatiiviset rakenteet käyttävät avaimiensa säilyttämiseen luvun 3 mukaisesti järjestettyä vektoria.

Sekä punamusta binäärihakupuu että järjestetty vektori tukevat $O(log\ n)$-kompleksista hakua avaimella, mutta alkion lisäys järjestettyyn vektoriin vie $O(n)$-kompleksisen ajan verrattuna binäärihakupuun $O(log\ n)$-kompleksisuuteen alkioita lisätessä. Binäärihakupuun solmu- ja osoitinrakenteesta johtuen haut standardikirjaston `set`-rakenteeseen vievät 2 kertaa enemmän aikaa kuin Boost.container-kirjaston järjestettyyn vektoriin perustuvaan litteään `set`-rakenteeseen tehtävät haut. Binäärihakupuun rakenne vie myös lähes kolminkertaisen määrän muistia verrattuna järjestettyyn vektoriin. [11]

Mikäli säiliöltä siis halutaan avaimen perusteella mahdollisimman nopeaa ja mahdollisimman muistitehokasta hakua lisäysoperaation kompleksisuuden hinnalla, ovat Boost.container-kirjaston järjestettyyn vektoriin perustuvat säiliöt todennäköisesti parempi valinta. Esi-

merkki tällaisesta voisi olla tilanne, jossa tiedetään, että rakenteeseen ei tulla lisäämään alkioita kovin usein. Myös tilanne, jossa alkiot voidaan esimerkiksi lisätä rakenteeseen kerralla sellaisena ajankohtana, että hitaammasta lisäyksestä ei koidu haittaa, on Boost.container-kirjaston säiliöt parempi valinta.

Lopuksi koottuna yhteen tilanteet, joissa Boost.container-kirjaston assosiatiivisten säiliöiden käytöllä saavutetaan selkeitä etuja ovat seuraavat:

- hakujen tehokkuus halutaan maksimoida
- muistin käyttö halutaan minimoida.

Molempiin tapauksiin liittyy ehto siitä, että lisäysoperaation tehokkuuden laskeminen on pienempi haitta, kuin säiliön vaihdolla saavutettava etu.

5. YHTEENVETO

Työssä käytiin läpi osa C++-standardikirjaston ja Boost.container-kirjaston säiliöistä. Työssä selvitettiin kullekin säiliölle optimaalinen käyttötilanne. Ensimmäiseksi käytiin läpi alkeellisia säiliöitä ja niistä johdettuja standardikirjaston säiliöitä. Tämän jälkeen esiteltiin Boost.container-kirjaston säiliöitä. Lopuksi selvitettiin säiliöiden operaatioiden asymptoottisia aikakompleksisuuksia vertailemalla niille sopivimpia käyttötilanteita.

Säiliöistä voitiin todeta, että Boost.container-kirjasto tarjoaa säiliöillään standardikirjastoon verrattuna enemmän joustavuutta, tai uusia ominaisuuksia. Näiden ansiosta löydettiin useita tilanteita, joissa Boost.container-kirjaston säiliöt tuovat selkeitä etuja ohjelman tehokkuuteen. Todennettiin myös se, että standardikirjaston vektori on edelleen hyvä lähtökohta sarjasäiliön valinnalle. Assosiatiivisista säiliöistä voitiin todeta, että mikäli aikakompleksisuuden kasvu lisäysoperaatioille ei aiheuta merkittäviä haittoja, on Boost.container-kirjaston tarjoamat järjestettyyn vektoriin perustuvat assosiatiiviset säiliöt tehokkaampi valinta.

Todetaan lopuksi, että oikeanlaisen säiliön valinta on nykypäivän tietokoneilla merkittävää vasta datamäärän kasvaessa suureksi, tai mikäli ohjelmalta vaaditaan lähes reaaliaikaista laskentaa. Aihetta voisi tutkia lisää suorittamalla varsinaisia suorituskykytestejä työssä esitellyille säiliöille. Tällöin voitaisiin myös vahvistaa vanhempien lähteiden paikkansapitävyys.

LÄHTEET

[1] *Programming Languages — C++*. 31. maaliskuuta 2020. URL: https://isocpp.org/files/papers/N4860.pdf (viitattu 07. 03. 2021).

[2] *C++ reference*. 11. elokuuta 2020. URL: https://en.cppreference.com/w/ (viitattu 05. 03. 2021).

[3] *Doubly linked list*. 14. kesäkuuta 2007. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/5/5e/Doubly-linked-list.svg/1920px-Doubly-linked-list.svg.png (viitattu 08. 03. 2021).

[4] Cormen, T., Leiserson, C., Rivest, R. ja Stein, C. *Introduction to algorithms*. 2009.

[5] *Red-black tree*. 13. marraskuuta 2019. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/6/66/Red-black_tree_example.svg/1920px-Red-black_tree_example.svg.png (viitattu 08. 03. 2021).

[6] Stroustrup, B. *The C++ Programming Language*. Pearson Addison Wesley, 2013.

[7] Schäling, B. *The Boost C++ Libraries*. 2014. URL: https://theboostcpplibraries.com (viitattu 10. 04. 2021).

[8] Polykin, A. *Boost C++ Application Development Cookbook*. 1st ed. PACKT Publishing, 2013.

[9] *Boost C++ Libraries*. 3. joulukuuta 2020. URL: https://www.boost.org/doc/libs/1_75_0/doc/html/container.html (viitattu 07. 03. 2021).

[10] *Stable_vector*. 11. joulukuuta 2020. URL: https://www.boost.org/doc/libs/1_75_0/libs/container/doc/images/stable_vector.png (viitattu 10. 04. 2021).

[11] Austern, M. *Why You Shouldn't Use set, and What You Should Use Instead*. Huhtikuu 2000. URL: http://lafstern.org/matt/col1.pdf (viitattu 14. 04. 2021).

[12] Mukherjee, A. *Learning Boost C++ libraries: solve practical programming problems using powerful, portable, and expressive libraries from Boost*. 1st ed. PACKT Publishing, 2015.