Heikki Heikkinen

Tarvitaanko adaptoituvia tietorakenteita

Tietotekniikan
(ohjelmistotekniikka)
pro gradu -tutkielma
30. marraskuuta 2011

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Heikki Heikkinen
Yhteystiedot: heikki.heikkinen@jyu.fi
Työn nimi: Tarvitaanko adaptoituvia tietorakenteita
Title in English: Do we need adaptive data structures
Työ: Tietotekniikan (ohjelmistotekniikka) pro gradu -tutkielma
Sivumäärä: 124
Tiivistelmä: Tutkielma käsittelee eri tietorakenteiden suorituskykyä. Tietorakenteita vertaillaan algoritmianalyysin ja suorituskyvyn mittaamisen avulla. Tutkimustulosten avulla tutkielmassa esitellään ehdotus adaptoituvasta tietorakenteesta, joka pyrkii yhdistämään kahden hyvin suoriutuneen luokan ominaisuuksia.
Abstract: The thesis examines the performance of different data structures. Data structures are compared using algorithm analysis and performance measurement. Based on the research results a proposal for an adaptive data stucture is presented. This data strucutre aims to combine characteristics of two data structures that performed well in the performance measurement.
Avainsanat: Tietorakenne, luokka, luokkakirjasto, suorituskyky, kompleksisuus, algoritmianalyysi, adaptoituva, C#-ohjelmointikieli, suorituskyvyn mittaaminen
Keywords: Data structure, class, class library, performance, complexity, algorithm analysis, adaptive, C# programming language, performance measurement

Sisältö

1 Johdanto

Ohjelmoijalla on nykyään käytettävissä suuri määrä eri tietorakennetoteutuksia. Monet tietorakenteet toteuttavat saman abstraktin tietotyypin ominaisuudet, mutta eivät käyttäydy muuten samalla tavalla. Esimerkiksi yksi tietorakenne saattaa suoriutua yhdestä operaatioista tehokkaasti, mutta ei toisesta, kun taas toinen tietorakenne saattaa käyttäytyä juuri päinvastaisella tavalla. Syynä tähän erilaisuuteen voi olla tietorakenteen käyttämä sisäinen rakenne tai valittu algoritmi.

Tutkielmassa adaptoituvalla tietorakenteella tarkoitetaan tietorakennetta, joka osaisi vaihtaa sisäistä tallennusrakennettaan tai käyttäytymistään tilanteen mukaan tehokkaammaksi. Tällainen tietorakenne voisi käyttäytyä eri operaatioiden aikana eri tavoin ja käyttää eri algoritmeja sen mukaan, mikä olisi kussakin tapauksessa tehokkainta.

Tutkielmassa tarkastellaan C#-kielen kirjastoista löytyviä tietorakennetoteutuksia ja tutkitaan näiden suorituskykyä eri operaatioiden suhteen. Tietorakenteita analysoimalla ja vertailemalla pyritään löytämään vastaus siihen, tarvitaanko adaptoituvia tietorakenteita, vai onko jo olemassa tietorakenne, joka suoriutuu kaikista perusoperaatioista parhaiten.

Luvussa 2 esitellään eri abstrakteja tietotyyppejä, joiden toteutuksia tutkielmassa tutkitaan. Luvussa 3 esitellään eri suorituskyvyn analysointimenetelmiä, joiden avulla tietorakenteita vertaillaan. Luvussa 4 esitellään tutkielmassa tutkittavat perustietorakenteet, analysoidaan niiden suorituskykyä algoritmianalyysillä ja tämän perusteella listataan näiden vahvuuksia ja heikkouksia. Luvussa 5 esitetään tutkielmassa suoritettujen mittausten tuloksia. Tietorakenteiden suorituskykyä tarkastellaan perusoperaatioiden eli lisäys-, etsimis- ja poisto-operaatioiden suhteen. Luvussa 5.4 tulosten perusteella tutkitaan, onko tutkittavien tietorakenteiden joukossa tietorakenne, joka suoriutui kaikista operaatioista parhaiten. Tulosten perusteella voidaan päätellä, että tällaista tietorakennetta ei tutkittavien tietorakenteiden joukossa ole. Luvussa 6 esitellään ehdotus adaptoituvasta tietorakenteesta, jossa yhdistetään listan ja hajautusrakenteen ominaisuuksia. Luvussa myös verrataan toteutusta olemassaoleviin lista- ja sanakirja-luokkien suorituskykyyn. Ehdotettu luokka suoriutuu parhaiten lisäysoperaatiossa. Etsimis- ja poisto-operaatioissa, luokka sijoittuu

lista- ja sanakirja-luokkien väliin. Luvussa 7 ehdotetaan adaptoituvan tietorakenteen toteutukseen liittyviä aiheita jatkotutkimusta varten. Lopuksi luvussa 8 esitellään tutkielman yhteenveto.

2 Abstraktit tietotyypit

Abstraktilla tietotyypillä (engl. ***A**bstract **D**ata **T**ype*) tarkoitetaan kokonaisuutta, joka koostuu arvoista tai alkioista, joita tietorakenteeseen tallennetaan, ja operaatioista, joita tietorakenteeseen voidaan kohdistaa. Abstrakti tietotyyppi ei sanele tietorakenteen toteutusta. Kuten Wood [15, s.3] kirjassaan mainitsee, yhdelle abstraktille tietotyypille voi olla olemassa monia tietorakenteita, jotka tukevat sen operaatioita.

Abstrakteja tietotyyppejä ovat esimerkiksi: joukko (engl. *Set*) ja kokoelma (engl. *Collection*). Joukko on abstrakti tietotyyppi, jonka tietorakenteeseen voidaan tallentaa n kappaletta alkioita, mutta yksikään alkio ei saa esiintyä tietorakenteessa useammin kuin kerran, kuten Kingston [7, s.36] mainitsee. Kokoelma on joukkoa vastaava rakenne, mutta sallii alkioiden esiintyvän rakenteessa useammin kuin kerran. Näistä abstrakteista tietotyypeistä voidaan johtaa vielä erikoistuneempia tyyppejä kuten: järjestetty joukko ja järjestetty kokoelma. Nämä tyypit sisältävät edellä mainittujen tyyppien ominaisuudet, mutta lisäksi alkioiden täytyy olla kokoelmassa järjestyksessä.

Connorin [2, s.1] mukaan abstrakti tietotyyppi on ohjelmoijan määrittelemä tyyppi, joka koostuu määritelmästä ja ainakin yhdestä toteutuksesta. Connor [2, s.1] mainitsee myös, että abstraktin tietotyypin toteutus koostuu konkreettisesta toteutuksesta, operaatioista, joita toteutukseen voidaan kohdistaa sekä rajapinnasta noihin operaatioihin. Rajapinnan tulee myös vastata abstraktin tietotyypin määritelmää. Korkean tason olio-ohjelmointikielissä, kuten Java ja C#(.NET), abstraktit tietotyypit voidaankin ajatella koostuvan rajapinnoista (engl. *Interface*) ja luokista (engl. *Class*), jotka toteuttavat näitä rajapintoja. Näissä kielissä onkin olemassa valmiita kirjastoja, Javassa collections framework ja .NET:ssä System.Collections-nimiavaruus, joissa on määritelty joukko rajapintoja ja näille toteutuksia.

Tutkielmassa tullaan käsittelemään seuraavia abstrakteja tietotyyppejä: kokoelma, joukko ja sanakirja.

3 Suorituskyvyn analysointi

Tutkielmassa tullaan käsittelemään tietorakenteiden suorituskykyä analysoimalla niihin kohdistettavien operaatioiden aikavaativuutta eli kompleksisuutta sekä mittaamalla ajonaikaisesti aikaa, joka operaatioihin kuluu. Operaatioiden kompleksisuutta käsitellään luvussa 3.1, jossa myös esitellään tutkielmassa käytettävä iso-O-notaatio. Suorituskyvyn mittaamista ajonaikaisesti käsitellään luvussa 3.2. Itse tietorakenteiden operaatioita analysoidaan luvuissa 4.1, 4.2, 4.3 ja 4.4. Mittaustuloksia tietorakenteiden operaatioista tarkastellaan luvussa 5, jossa tulokset esitetään taulukkomuotoisina ja kuvaajina.

3.1 Algoritmianalyysi

Algoritmeja ja niiden suorituskykyä voidaan analysoida monella tavalla. Voidaan laskea jokainen atominen operaatio, joka algoritmin aikana tehdään. Suoritettavien operaatioiden määrä voidaan myös suhteuttaa algoritmin syötteisiin. Algoritmin syötteisiin suhteuttaminen on erityisen hyödyllistä tietorakenteisiin kohdistettavien algoritmien analysoinnissa, koska algoritmin syötteisiin kuuluu tietorakenne itse. Algoritmeja analysoitaessa voidaan myös keskittyä parhaimman tapauksen arvioimiseen, pahimman tapauksen arvioimiseen ja keskimääräisen suorituskyvyn arvioimiseen. Suorituskyvyn lisäksi algoritmeja voidaan arvioida niiden vaatiman muistinkäytön suhteen.

Tarkastellaan esimerkiksi yksinkertaisen taulukkorakenteen lisäysoperaatiota. Oletetaan, että rakenteessa säilytetään tieto viimeisestä vapaasta indeksistä tai rakenteessa olevien alkioiden määrästä ja taulukon kapasiteetista. Lisäysalgoritmi voisi olla seuraavanlainen:

1. Tarkista onko taulukko täynnä, eli onko alkioiden määrä sama kuin taulukon kapasiteetti.
2. Jos taulukko on täynnä:
   (a) Luo uusi suurempi taulukko.
   (b) Kopioi alkuperäisen taulukon alkiot uuteen taulukkoon.

3. Lisää uusi alkio vapaaseen indeksiin.

4. Kasvata alkioiden määrää yhdellä.

Tarkastellaan ensin parasta tapausta, eli tapausta, jossa taulukkoon mahtuu vielä alkioita. Tällöin algoritmissa tarvitsee tehdä kolme operaatiota: kapasiteetin tarkistaminen, uuden alkion lisääminen ja alkioiden määrän kasvattaminen. Parhaassa tapauksessa lisäysoperaatio ei siis näyttäisi suoraan olevan riippuvainen taulukon alkioiden määrästä, koska aina tarvitaan vain kolme operaatiota, oli taulukon alkioiden määrä sitten viisi tai tuhat. Voitaisiin siis sanoa, että algoritmin aikavaativuus on kolme operaatioita ($T(n) = 3$). Muistinkäytön suhteen tämä tapaus on myös paras. Koska taulukkoon mahtuu vielä alkioita, ei ylimääräistä muistia tarvitse varata esimerkiksi uudelle taulukolle.

Tarkastellaan sitten pahinta tapausta, eli tapausta, jossa taulukko on täynnä. Tällöin algoritmissa tarvitsee tehdä ensin tarkistus kapasiteetista, luoda uusi taulukko, kopioida taulukon alkiot uuteen taulukkoon, lisätä uusi alkio ja kasvattaa alkioiden määrää yhdellä. Nyt algoritmin aikavaativuus onkin riippuvainen taulukon alkioiden määrästä, koska vanhan taulukon alkiot pitää kopioida uuteen taulukkoon. Oletetaan myös yksinkertaisuuden vuoksi, että taulukon alkiot kopioidaan uuteen taulukkoon yksi kerrallaan. Olkoon taulukon alkioiden määrä ennen lisäystä n. Tällöin algoritmissa tarvitsee tehdä $n+$ neljä operaatiota ($T(n) = n + 4$). Siis kapasiteetin tarkistus (1), uuden taulukon luominen (1), alkion lisääminen (1), alkioiden määrän kasvattaminen (1) ja alkioiden kopioiminen (n). Tämä on pahin tapaus myös muistinkäytön suhteen, koska algoritmin aikana täytyy varata muistia uudelle, suuremmalle taulukolle. Lisäksi, kun uusi taulukko luodaan, niin järkevässä algoritmissa tilaa varataan uudelle taulukolle enemmän kuin vain uuden alkion vaatima määrä. Yleinen tapa on kaksinkertaistaa taulukon koko, joka tarkoittaa sitä, että muistia algoritmin aikana käytetään mahdollisesti kolminkertainen määrä alkuperäisen taulukon kokoon nähden. Tietysti alkuperäinen taulukko tuhotaan algoritmin lopuksi ja sen varaama muisti vapautetaan. Silti algoritmin päätyttyä taulukolle on varattu kaksinkertainen määrä muistia alkuperäiseen taulukkoon nähden.

Tällaisesta aikavaativuusanalyysistä saadaan tuloksiksi tietorakenteen alkioiden määrän funktioita. Entä jos algoritmi on monimutkaisempi ja sisältää useita ehtolauseita ja tietorakenteen alkioiden läpikäymistä useammassa paikassa? Voisi kuvitella, että aikavaativuudeksi saataisiin hyvin monimutkainen funktio. Entä jos analyysi on tehty algoritmille, joka on kirjoitettu korkean tason ohjelmointikielellä, joka

tullaan muuttamaan esimerkiksi välikieleksi, kuten Javassa ja C#:ssa, ja vielä ajonaikaisesti kääntämään konekielelle? Kuinka tällaisia algoritmeja voitaisiin analysoida tarkasti laskemalla kaikkien tehtävien operaatioiden määrä? Onko näin tarkka analysointi edes tarpeellista, kuten Goodrich ja Tamassia [5, s.114] kysyvät?

Yksinkertaistamaan analyysia ja helpottamaan algoritmien vertailua voidaan käyttää iso-O-notaatiota, joka luokittelee algoritmit niiden aikavaativuuden korkeimman asteen mukaan. Iso-O:n sääntönä on, että aikavaativuus $T(n)$ kuuluu luokkaan $O(F(n))$, jos on olemassa positiiviset vakiot c ja n_0 siten, että $T(n) \leq cF(n)$, kun $n \geq n_0$, kuten Weiss [13, s.161] mainitsee.

Nyt voidaan siis sanoa, että edellä mainitun algoritmin parhaimman tapauksen aikavaativuus on siis $O(1)$ eli n:stä riippumaton tai vakioaikainen, koska $T(n) = 3 \leq 3 * 1$, jossa $n \geq 1$. Voidaan myös sanoa, että pahimman tapauksen aikavaativuus on $O(n)$, koska $T(n) = n + 4 \leq 2 * n$, kun $n \geq 4$.

Iso-O-notaation lisäksi on olemassa muitakin sääntöjä, kuten iso-Omega (Ω) , iso-Theta (Θ) ja pieni-O (o), kuten Weiss [13, s.161] mainitsee. Iso-Omegan sääntö on seuraava: $T(n) = \Omega(F(n))$, jos on olemassa positiiviset vakiot c ja n_0 siten, että $T(n) \geq cF(n)$, kun $n \geq n_0$. Iso-Thetan sääntö on seuraava: $T(n) = \Theta(F(n))$, jos ja vain jos $T(n) = O(F(n))$ ja $T(n) = \Omega(F(n))$. Pieni-o:n sääntö on seuraava: $T(n) = o(F(n))$, jos on olemassa positiiviset vakiot c ja n_0 siten, että $T(n) < cF(n)$, kun $n \geq n_o$.

Nyt voidaan huomata, että koska parhaimman tapauksen kompleksisuudeksi saatiin $O(1)$, kaikkien tietorakenteen alkioiden määrästä riippumattomien algoritmien kompleksisuus on luokassa $O(1)$. Eli aikavaativuus funktio, jossa n:n kerroin on 0, on aina luokkaa $O(1)$. Pahimman tapauksen aikavaativuus olisi myös voitu kirjoittaa muodossa $n + 4 = n + O(1)$, jolloin aikavaativuudeksi olisi saatu $O(n) + O(1)$. Iso-O:n ja sen sukuisten notaatioiden avulla voidaan eliminoida epäoleellisia yksityiskohtia kompleksisuuden esittämisestä, kuten Cormen, Leiserson ja Rivest [3, s.28] mainitsevat. Tällöin $O(n) + O(1) = O(n)$.

3.2 Suorituskyvyn mittaaminen

Tietorakenteiden operaatioiden kompleksisuuksien eli aikavaativuuksien lisäksi tutkielmassa testataan operaatioiden suorituskykyä käytännössä. Käytännön testaamiseen on kaksi päätapaa: empiirinen testaus ja simulaatio, kuten Wood [15, s.41-43] mainitsee. Käytännön mittaamisella saadaan lisäinformaatiota tietorakenteiden

suorituskyvystä. Esimerkiksi, vaikka tietorakenteet kuuluisivat tietyn operaation suhteen samaan luokkaan, jos niitä analysoitaisiin esimerkiksi iso-O-notaation mukaisesti, voivat niiden suorituskyvyt silti erota toisistaan huomattavastikin.

Empiirisessä testauksessa suorituskykyä mitataan todellisessa ympäristössä, oikealla sovelluksella ja todellisella datalla. Oikean sovelluksen suorituskykyä testataan ajamalla sovellusta ja mittaamalla sen suorituskykyä esimerkiksi profilointityökaluilla, jotka mittaavat suoritinajan- ja muistinkäyttöä.

Simulaatiossa rakennetaan malli ja suorituskykyä testataan simuloidulla datalla. Malli voi olla todellinen sovellus tai yksinomaan testausta varten ohjelmoitu sovellus.

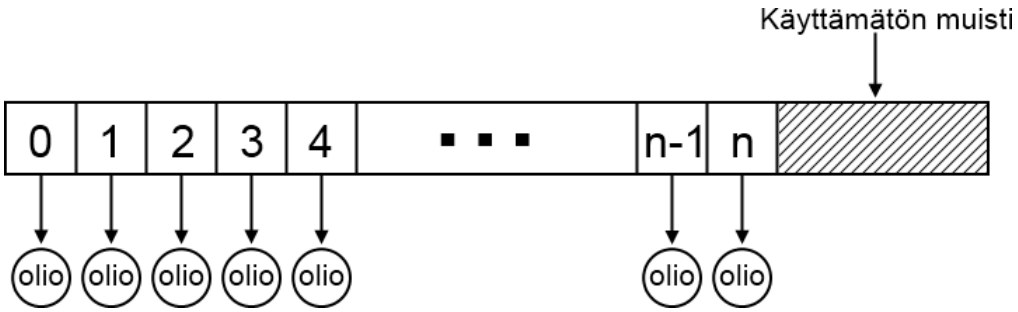
Tutkielmassa käytännön testauksen menetelmäksi valittiin simulaatio, koska tutkielmassa halutaan keskittyä pelkästään tietorakenteisiin ja niiden suorituskykyyn. Testausta varten ohjelmoitiin testisovellus, josta tietorakenteiden testauksia voitiin käynnistää. Testisovellus kirjoitettiin C#-kielellä ja testattaviksi tietorakenteiksi valittiin System.Collections- ja System.Collection.Generic-nimiavaruuksissa olevia tietorakenteita. Lisäksi testattiin itse tehtyä binääripuuluokkaa.

4 Perustietorakenteet

Tässä luvussa käsitellään perustietotyyppejä, joita voidaan käyttää abstraktien tietotyyppien toteutuksessa. Luvussa käsiteltäviksi tietorakenteiksi on valittu perustietoranteita, jotka löytyvät lähes jokaisesta ohjelmointikielestä. Tällaisia rakenteita ovat taulukkorakenteet ja taulukoilla toteutetut listat, linkitetyt rakenteet kuten linkitetyt listat, hajautusrakenteet kuten hajautustaulukot ja sanakirjat sekä puut kuten binääripuut ja nelipuut. Luvussa käsitellään kunkin tietorakenteen kohdalla käyttötarkoituksia, joihin kyseiset tietorakenteet sopivat hyvin ja tapauksia, joista kyseiset tietorakenteet eivät suoriudu tehokkaasti.

4.1 Taulukkorakenteet

Taulukko on hyvin yleiskäyttöinen rakenne ja sitä voikin käyttää monen abstraktin tietotyypin perustietorakenteena. Taulukolla voidaan toteuttaa muun muassa lista-, puu- ja hajautusrakenteita. Taulukko on peräkkäistietorakenne, jonka alkioihin voidaan viitata järjestysnumerolla. Taulukon alkiot yleisesti tallennetaan muistiin peräkkäin kuten Earley [4, s.1] huomauttaa. Taulukko onkin niin sanottu hajasaantirakenne (engl. *random access*) kuten Wirth [14, s.30] mainitsee. Hajasaanti tarkoittaa sitä, että mihin tahansa taulukon alkioon voidaan viitata satunnaisesti, ja mikä tahansa alkio on yhtä lailla saatavilla. Esimerkiksi taulukon kolmanteen alkioon voidaan viitata sen järjestysnumerolla suoraan viittaamatta esimerkiksi sitä edeltäviin alkioihin. Taulukko on myös staattinen tietorakenne. Tämä tarkoittaa sitä, että taulukon koko täytyy määrittää taulukon luomisen yhteydessä. Esimerkiksi, jos luodaan taulukko, johon mahtuu kolme alkiota, ei tähän taulukkoon voi lisätä neljättä alkiota. Jos neljäs alkio halutaan lisätä, täytyy luoda uusi vähintään neljän alkion kokoinen taulukko, johon edellisen taulukon alkiot täytyy kopioida ja tämän jälkeen voidaan lisätä neljäs alkio. Tämän vuoksi taulukko sopiikin parhaiten tilanteisiin, joissa taulukkoon lisättävien alkioiden määrä tiedetään taulukon luomisen yhteydessä.



Kuva 4.1: Taulukkorakenne

Kuvassa 4.1 on kuvattu taulukkorakenne. Kuvassa 0–n merkityt ruudut ovat taulukon indeksejä, joista lähtee viitteitä taulukon alkioihin. Taulukon alkiot voivat olla arvotyyppisiä, jolloin taulukon alkioiden muisti on varattu pinosta peräkkäisenä muistialueena, tai alkiot voivat olla referenssityyppisiä, jolloin taulukoon varattu muisti koostuu osoittimista, jotka varataan pinosta peräkkäisenä muistialueena ja itse alkioiden muisti varataan keosta. Taulukon lopussa on käyttämätöntä muistialuetta, joka on varattu taulukon alkioille tai niiden osoittimille, ja jota ei voida käyttää muuhun tarkoitukseen ennen kuin taulukolle varattu muisti vapautetaan.

Taulukkoa käyttävissä listarakenteissa voi kuitenkin olla metodeja, joilla ylimääräinen käyttämätön tila voidaan vapauttaa luomalla pienempi taulukko nykyisen tilalle ja kopioimalla taulukon alkiot siihen. Esimerkiksi C#:n System.Collections.Generic-nimiavaruudesta löytyvällä List-luokalla on *TrimExcess*-metodi, joka tekee taulukosta täsmälleen siihen tallennettujen alkioiden määrän kokoisen. MSDN:n [10] mukaan tämä metodi ei kuitenkaan pienennä taulukkoa, jos siitä on yli 90 prosenttia käytetty. Tämä ratkaisu on tehty, koska tästä operaatiosta saatava hyöty olisi hyvin pieni verrattuna työn määrään suurien taulukkojen kopioimisessa.

4.1.1 Perusoperaatiot taulukkorakenteella

Kuten aikaisemmin mainittiin, taulukko on staattinen rakenne, johon voi lisätä vain niin monta alkiota kuin sille on luomisen yhteydessä varattu tilaa. Tämä seikka on huomioitava, kun taulukkorakenteeseen lisätään alkioita. Parhaimmassa tapauksessa alkion lisääminen taulukkorakenteeseen on vakioaikainen eli kompleksisuudeltaan $O(1)$. Alkio voidaan lisätä taulukkorakenteeseen vakioaikaisesti, jos taulukossa on tilaa lisättävälle alkiolle ja taulukon seuraava tyhjä indeksi on tiedossa alkion lisäämisen aikana. Monet taulukkoa käyttävät tietorakenteet pitävätkin yllä tietoa seuraavasta tyhjästä indeksistä. Jos tietoa seuraavasta tyhjästä indeksistä ei

ole, täytyy taulukko käydä indeksi indeksiltä läpi, kunnes tyhjä indeksi löytyy. Tässä tapauksessa alkion lisäys onkin lineaariaikainen eli kompleksisuudeltaan $O(n)$. On siis hyödyllistä säilyttää tieto seuraavasta tyhjästä indeksistä. Vaikka seuraava tyhjä indeksi olisikin tiedossa, voi alkion lisääminen taulukkoon silti olla vakioaikaista kalliimpi operaatio. Tämä johtuu taulukon staattisuudesta. Jos taulukko on jo täynnä, ei siihen voida lisätä uutta alkiota. Lisätilan varaaminen vaatiikin uuden suuremman taulukon luomista ja alkioiden kopioimista tähän uuteen taulukkoon. Myös tällöin lisäysoperaatiota voidaan pitää lineaariaikaisena. Toisaalta kielissä kuten Java ja C# on tätä varten toiminnallisuus vanhan taulukon alkioiden kopioimisesta uuteen taulukkoon toteutettu virtuaalikoneen tasolla natiivina toteutuksena, jolloin tämä operaatio ei ole yhtä kallis kuin jokaisen alkion kopioiminen yksitellen uuteen taulukkoon. MSDN:n [9] mukaan C#:ssa taulukon kopiointi toiseen taulukkoon, eli Array-luokan staattinen Copy-metodi, vastaa C++:n memmove-funktiota, jossa lähde- ja kohdemuistialue voivat olla limittäin. Tämän vuoksi lähde- ja kohdetaulukot voivat olla sama taulukko.

Lisätilan varaamiseen liittyy kuitenkin uusia haasteita. On päätettävä, kuinka paljon lisätilaa uuteen taulukkoon halutaan. Jos taulukkoon varataan lisätilaa ainoastaan uutta alkiota varten, joudutaan lisätilaa varaamaan taulukon täyttymisen jälkeen jokaisen uuden lisättävän alkion yhteydessä. Tämän lähestymistapa pitäisi muistinkäytön minimitasolla, mutta olisi lisäysoperaation kompleksisuuden kannalta huono vaihtoehto. Usein lisätilaa varataankin vanhan taulukon kokoon nähden kaksinkertaisesti. Tämä lähestymistapa varmistaa, että lisätilaa ei tarvitse varata jokaisen lisäysoperaation yhteydessä. Toisaalta lisätilaa saatetaan varata myös turhan paljon. Esimerkiksi, jos taulukkorakenteeseen tarvitsisi lisätä enää vain yksi alkio sen elinaikana, ja vanhan taulukon koko olisi jo todella suuri, lähes puolet taulukosta jäisi käyttämättä. Koska tämän tilanvaraus olisi tehty taulukolle, ei mikään muu voisi hyväksi käyttää varattua muistialuetta. Muistihukka on suurempi, jos taulukko luodaan suurille arvotyyppisille alkioille, kuin jos taulukko luodaan viitetyyppisille alkioille. Arvotyyppiset alkiot tallennetaan suoraan taulukon indeksien osoittamiin muistialueisiin, kun taas viitetyyppisten alkioiden tapauksessa taulukon indeksien osoittamiin muistialueisiin tallennetaan viitteitä itse alkioihin.

Koska taulukko on hajasaantirakenne, voidaan taulukkoon lisätä alkio mihin sen indeksiin tahansa. Tällöin lisättävässä indeksissä oleva ja siitä seuraavat alkiot siirretään yhden askeleen eteenpäin, jolloin lisättävä alkio voidaan lisätä indeksiin. Tässä tapauksessa, jos indeksi, johon alkio lisätään, on taulukon ensimmäinen indeksi,

on lisäysoperaatio lineaariaikainen ($O(n)$). Vastaavasti jos indeksi, johon alkio lisätään, on taulukon viimeinen indeksi, on operaatio vakioaikainen ($O(1)$). Edelleen, jos taulukossa ei ole tilaa lisättävälle alkiolle on tehtävä uusi suurempi taulukko, johon alkiot kopioidaan.

Alkion etsiminen taulukkorakenteesta on lineaariaikainen ($O(n)$) operaatio, jos etsittävän alkion indeksiä ei tiedetä. Tällöin taulukko on käytävä läpi indeksi indeksiltä kunnes etsittävä alkio löytyy. Jos etsittävän alkion indeksi tiedetään, on alkion etsiminen luonnollisesti vakioaikainen ($O(1)$). Tämä on hajasaantirakenteen ominaisuus. Jos kuitenkin tiedetään, että taulukon alkiot ovat järjestyksessä, voidaan alkio löytää taulukosta logaritmisessa ajassa ($O(log_2(n))$), jos käytetään puolitushakua. Puolitushaussa otetaan taulukon keskimmäinen alkio. Jos alkio on etsitty alkio, lopetetaan. Jos alkio ei ole etsitty alkio ja etsitty alkio on järjestyksessä ennen alkiota, valitaan alkio taulukon ensimmäisen ja nykyisen alkion väliltä. Jos etsitty alkio on taas järjestyksessä nykyisen alkion jälkeen, otetaan keskimmäinen alkio nykyisen alkion ja taulukon viimeisen alkion väliltä. Tätä toistetaan, kunnes etsitty alkio löytyy tai puolitusta ei voida enää tehdä.

Alkio voidaan poistaa taulukkorakenteesta asettamalla kyseisen alkion indeksi tyhjäksi. Jos poistettava alkio on viitetyyppinen, voidaan indeksi asettaa null-arvoiseksi. Jos poistettava alkio on arvotyyppinen, voidaan indeksi asettaa tietotyypin oletusarvoksi. Indeksi asettaminen tyhjäksi ei kuitenkaan aina ole ainoa alkion poistamiseen liittyvä operaatio. Jos taulukosta poistetaan viitetyyppisiä alkioita, ilmestyy taulukon keskelle null-arvoisia indeksejä. Tämä ei ole suotavaa, jos esimerkiksi jokin toimenpide halutaan suorittaa kaikille taulukkorakenteen alkioille. Jos taulukossa esiintyy null-arvoja, täytyisi ennen alkioon kohdistettavaan toimenpidettä aina tarkistaa, onko kyseinen indeksi null-avoinen. Arvotyyppisten alkioiden yhteydessä ei edes voida tehdä tällaista tyhjän arvon tarkistusta, koska arvotyyppisille alkioille ei ole olemassa null-arvoa vastaavaa arvoa. Tämän vuoksi tyhjät arvot usein eliminoidaankin siirtämällä poistettavasta alkiosta seuraavia alkioita yhden askeleen verran taaksepäin, kun alkio poistetaan taulukosta. Tällöin taulukon keskelle ei jää niin sanottuja tyhjiä arvoja ja taulukon läpikäynnin yhteydessä ei tarvitse tehdä tarkistusta null-arvoisesta indeksistä. Tämän siirto-operaation vuoksi alkion poistaminen taulukkorakenteesta onkin pahimmassa tapauksessa lineaariaikainen, jos poistettava alkio on taulukon ensimmäinen alkio. Vastaavasti, jos poistettava alkio on taulukon viimeinen alkio, on alkion poistaminen vakioaikainen operaatio, koska viimeisen alkion jälkeen ei luonnollisestikaan ole alkiota, joita siirrettäisiin

taaksepäin. Toisaalta, kuten alkion lisäämisenkin yhteydessä, voidaan taulukon alkioiden siirtämiseen käyttää virtuaalikoneen tasolla toteutettua natiivia toteutusta. Tällöin taulukko, josta alkiot kopioidaan, on samalla taulukko, johon alkiot kopioidaan. Kun alkio poistetaan ja alkiota seuraavat alkiot siirretään, täytyy viimeinen alkio myös muistaa asettaa null-arvoiseksi ja asettaa seuraava tyhjä indeksi yhden askeleen taaksepäin.

4.1.2 Taulukkorakenteen vahvuudet

Taulukkorakenteen vahvuus on se, että se on hajasaantirakenne, eli sen jokainen alkio voidaan saada rakenteesta vakioaikaisesti, jos tiedetään, missä indekseissä ne ovat. Uuden alkion lisääminen taulukon loppuun on myös usein vakioaikainen operaatio, jos taulukon kokoa kasvatetaan tilan loppumisen yhteydessä esimerkiksi kaksinkertaiseksi. Tällöin tarve taulukon kasvattamiselle vähenee aina, kun taulukon kokoa kasvatetaan.

Taulukkoon kohdistettavat operaatiot ovat nopeita myös siksi, että se on peräkkäisrakenne, eli taulukon indekseille varataan yhtenäinen muistialue, jossa indeksit ovat peräkkäin. Taulukon läpikäynti onkin nopeaa, koska seuraavan indeksi muistialue on helppo laskea. Lisäksi taulukon alkioiden kopioiminen toiseen taulukkoon on myös nopeaa, koska kopioitavan taulukon muistialue voidaan kopioida sellaisenaan toiselle taulukolle. Jos taulukko on luotu arvotyyppisille alkioille, arvot kopioidaan toiseen taulukkoon, ja jos taulukko on luotu viitetyyppisille alkioille, alkioiden viitteet eli muistiosoitteet kopioidaan toiseen taulukkoon.

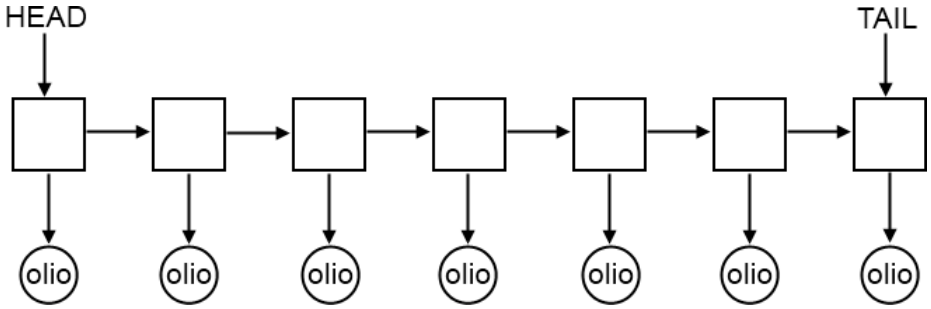
4.1.3 Taulukkorakenteen heikkoudet

Vaikka se, että taulukko on peräkkäisrakenne, on sen vahvuus, on se myös sen heikkous. Taulukolle täytyy varata muistialue staattisesti peräkkäisenä muistialueena taulukon luomisen yhteydessä. Nyt jos esimerkiksi taulukolle varataan suuri määrä muistia, mutta taulukkoon lisätään vain vähän alkioita, niin taulukolle on silti varattu sen luomisen yhteydessä varattu muisti, eikä sitä voida käyttää muuhun tarkoitukseen, niin kauan kuin taulukko on olemassa. Tämä seikka on hyvä ottaa huomioon varsinkin, kun taulukko luodaan suurille arvotyyppisille alkioille, jolloin alkioiden arvot tallennetaan taulukon muistialueeseen. Viitetyyppisille alkiolle luotuun taulukkoon tallennetaan vain alkioiden viitteet, mutta käyttämätöntä muistia ei voida silti käyttää muuhun tarkoitukseen. Taulukko ei siis välttämättä ole muis-

tinkäytön suhteen tehokas tietorakenne, jos taulukolle varataan liikaa muistia.

4.2 Linkitetyt listat

Linkitetyt listat ovat yksi vaihtoehto toteuttaa listarakenne. Toisin kuin taulukko, linkitetty lista ei ole hajasaantirakenne eikä sitä tarvitse tallentaa peräkkäisrakenteena muistiin. Linkitetty lista voidaan toteuttaa yksisuuntaisena (engl. *singly linked*) tai kaksisuuntaisena (engl. *doubly linked*) listana. Yksisuuntaisessa rakenteessa listan solmuun tallennetaan alkio, tai viite siihen, ja linkki seuraavaan solmuun. Kaksisuuntaisessa rakenteessa listan solmuun tallennetaan alkio, viite edelliseen solmuun ja viite seuraavaan solmuun. Koska linkitetyn listan alkioihin ei voida viitata suoraan järjestysnumerolla, esimerkiksi viite kolmanteen alkioon saadaan vain kulkemalla kaikkien sitä edeltävien solmujen kautta. Linkitetty lista vaatii enemmän muistia kuin taulukko, koska yhden alkion tallentamiseen listaan tarvitaan muistia alkioon, listan solmuun sekä seuraava-viitteeseen ja kaksisuuntaisessa listassa myös edellinen-viitteeseen. Jos esimerkiksi tallennetaan viitetyyppisiä alkioita ja linkitetty lista olisi yksisuuntainen ja tätä verrattaisiin taulukkoon, jonka täyttösuhde olisi 70%, niin tällöin yhtä alkiota varten linkitetyssä listassa tarvitaan muistia alkiolle ja kahdelle viitteelle, joista ensimmäinen viite on alkioon ja toinen viite seuraavaan solmuun, kun taas taulukossa muistia tarvitaan alkiolle ja viitteelle alkioon. Nyt jos tallennetaan 70 alkiota, niin tällöin linkitetyn listan muistinkäyttöön tarvitaan muistia 70:lle alkiolle ja 140:lle viitteelle, kun taas taulukolla muistia tarvitaan 70:lle alkiolle ja 100:lle viitteelle. Taulukkorakenne tarvitsee vähemmän muistia, vaikka taulukon täyttösuhde olisi 50%. Ja koska taulukon kokoa kasvatettaessa sen koko tuplattiin, niin taulukko on aina muistinkäytön suhteen tehokkaampi kuin linkitetty lista olettaen, että alkioita ei poisteta enempää kuin 50% taulukon kapasiteetista.



Kuva 4.2: Linkitetyn listan rakenne

Kuvassa 4.2 on kuvattu linkitetyn listan rakenne. Listalla itsellään on kaksi viitettä listan solmuihin. HEAD-viite osoittaa listan ensimmäiseen solmuun ja TAIL-viite osoittaa listan viimeiseen solmuun. Lista voi myös pitää yllä listan solmujen lukumäärää. Linkitetyn listan solmut ovat linkitettyinä toisiinsa. Jokaisella listan solmulla on seuraava-viite, joka osoittaa seuraavaan solmuun listassa. Viimeisellä solmulla seuraava-viite on aina null-arvoinen. Listan solmuissa on viitteet listan alkioihin. Linkitetyn listan muistia ei varata peräkkäisenä muistialueena, vaan jokaisen solmun muisti varataan erikseen. Näin listaan ei varata muistia, joka jäisi käyttämättömäksi, jos lista ei ole täynnä, kuten taulukon tapauksessa.

4.2.1 Perusoperatiot linkitetyllä listalla

Toisin kuin taulukkorakenne, linkitetty lista on dynaaminen tietorakenne. Tämä tarkoittaa sitä, että sille ei tarvitse varata muistia staattisesti sen luomisen yhteydessä. Alkion lisääminen linkitettyn listan loppuun on aina vakioaikainen operaatio ($O(1)$). Tällöin listaan lisätään uusi solmu, jonka alkioksi lisättävä alkio asetetaan. Lisääminen tapahtuu asettamalla vanhan viimeisen solmun seuraava-viite uuteen solmuun ja asettamalla lisätty solmu listan viimeiseksi solmuksi. Koska linkitetty lista ei ole hajasaantirakenne, ei alkiota voida suoraan lisätä mihinkään tiettyyn indeksiin. Jos alkio halutaan lisätä n:neksi alkioksi, täytyy lista käydä läpi alusta $(n-1)$:een alkioon ja asettaa lisättävän solmun seuraava viite $(n-1)$:n seuraava-viitteen osoittamaan solmuun ja $(n-1)$:n seuraava-viite asettaa lisättävään solmuun.

Alkion etsiminen linkitetystä listasta on lineaariaikainen operaatio ($O(n)$). Tällöin lista on käytävä läpi ensimmäisestä solmusta lähtien, kunnes etsittävä alkio löytyy.

Alkion poistaminen linkitetystä listasta on pahimmassa tapauksessa lineaariaikainen ($O(n)$). Jos viitettä alkion solmuun ei ole käytettävissä ennen alkion poistamista, täytyy se ensin etsiä listasta. Kun viite poistettavan alkion solmuun on löydetty, voidaan alkio poistaa asettamalla poistettavan alkion edellisen solmun seuraava-viite poistettavan alkion seuraavaan solmuun. Ja jos lista on kaksisuuntaisesti linkitetty, niin täytyy myös asettaa poistettavan alkion seuraavan solmun edellinen-viite poistettavan alkion edelliseen solmuun.

4.2.2 Linkitettyjen listojen vahvuudet

Linkitetyn listan vahvuus on se, että se on dynaaminen tietorakenne. Toisin kuin taulukolle, sille ei tarvitse etukäteen varata muistia, vaan jokaiselle solmulle varataan muisti erikseen. Toisaalta linkitetyn listan tallentamiseen tarvitaan enemmän muistia, kun samankokoisen taulukon tallentamiseen, koska muistia tarvitaan itse solmun tallentamiseen ja seuraava-, ja kaksisuuntaisen listan tapauksessa myös edellinen-viitteisiin, kun taas taulukkoon tallennetaan vain itse alkio tai viite siihen. Mutta jos taulukossa on paljon käyttämättömiä indeksejä, niin se voi silti viedä enemmän muistia kuin linkitetty lista, joka ei hukkaa muistia.

4.2.3 Linkitettyjen listojen heikkoudet

Linkitetyn lista heikkoudeksi voidaan lukea se, että se on taulukkoa hitaampi tietorakenne, kuten luvun 5 mittaustuloksista voidaan nähdä. Tämä on loogista, koska linkitetty lista ei ole peräkkäisrakenne, eli sille ei varata peräkkäistä muistialuetta, vaan sen solmuille voidaan varata muistia mistä tahansa vapaasta muistialueesta. Lisäksi jokaiselle solmulle varataan muistialue erikseen, kun uusi alkio lisätään listaan. Alkion lisääminen on siis oletettavasti nopeampaa taulukkoon, koska uudelle alkiolle on jo valmiiksi varattu muistialue, ja taulukon läpikäynti on nopeampaa, koska taulukon alkiot ovat peräkkäin säännöllisin välein muistissa, kun taas linkitetyn listan solmut on tallennettu muistiin sinne, mistä vapaata tilaa on löytynyt. Eli taulukossa minkä tahansa alkion muistiosoite voidaan helposti löytää laskennallisesti, kun taas linkitetyssä listassa täytyy navigoida solmujen seuraava-viitteitä käyttäen.

Linkitetty lista vaatii myös taulukkoa enemmän muistia, koska jokaista alkiota kohti tarvitaan solmu ja seuraava-viite, ja kaksisuuntaisen listan tapauksessa myös edellinen-viite. Taulukon tapauksessa tarvitsee vain tallentaa alkio taulukon indeksin osoittamaan muistialueeseen arvotyyppisten alkioiden tapauksessa tai viite alkioon viitetyyppisten alkioiden tapauksessa.

4.3 Hajautusrakenteet

Hajautuksessa (engl. *Hashing*) hyödynnetään hajasaantirakenteiden periaatetta, jolla pyritään saavuttamaan $O(1)$ keskimääräinen kompleksisuus lisäys- ja etsimis-operaatioille, kuten Kingston [7, s.122] mainitsee. Hajautuksessa alkion lisäys ja haku

suoritetaan alkioon liittyvän avaimen perusteella. Hajautusrakenteen perustietorakenteena käytetään yleisesti taulukkoa, joka on hajasaantirakenne. Koska taulukon mihin tahansa indeksiin päästään käsiksi vakioajassa, soveltuu se hyvin hajautusrakenteen perustietorakenteeksi.

Hajautuksessa avaimista muodostetaan hajautusavain (engl. *Hash code*), joka vastaa yhtä taulukon indeksiä. Esimerkiksi, jos hajautusrakenteen taulukon koko olisi 64, jolloin pienin indeksi olisi 0 ja suurin indeksi olisi 63, ja rakenteeseen haluttaisiin lisätä alkio X avaimella K, jonka hajautusavain olisi 139, niin tällöin alkiota ei voitaisi lisätä indeksiin 139, koska suurin indeksi on 63. Tällöin lopullinen indeksi saadaan lukujen 139 ja 64 jakojäännöksestä: $139 \bmod 64 = 11$. Nyt alkio X voidaan lisätä taulukon indeksiin 11. Tällöin alkio X teoriassa voidaan löytää rakenteesta vakioajassa käyttämällä samaa hajautusfunktiota, jos ei ole tapahtunut yhteentörmäyksiä.

Olio-ohjelmointikielissä kuten Java ja C# kaikki oliot peritään object-luokasta, jolle on määritelty GetHashCode-metodi. Metodi palauttaa hajautusavaimen, jonka tietotyyppi on 32-bittinen kokonaisluku. Koska muuta rajoitusta kuin sen tietotyyppi ei hajautusavaimelle ole, voi se olla myös negatiivinen luku. Nyt jos lisättävän alkion hajautusavain olisi vaikka -4, ja se haluttaisiin lisätä edellä mainittuun kuudenkymmenenneljän alkion kokoiseen hajautustaulukkoon, ei hyväksyttävää indeksiä saataisikaan enää pelkästään laskemalla hajautusavaimen ja taulukon koon jakojäännöstä, koska $-4 \bmod 64 = -4$. Taulukossa ei voi olla negatiivisia indeksejä, joten hajautusfunktiota on muutettava. Hajautusfunktioon täytyy lisätä vaihe, joka varmistaa, että hajautusavain on positiivinen kokonaisluku. Tämä voidaan toteuttaa muun muassa soveltamalla ja-bittioperaattoria hajautusavaimeen ja kokonaislukujen suurimpaan arvoon ($2^{31} - 1 = 2147483647$). Tällöin lisättävän alkion indeksiksi saadaan $(-4\&2147483647) \bmod 64 = 2147483644 \bmod 64 = 60$.

Koska hajautusrakenteen taulukon koko on äärellinen ja koska kahden eri avaimen hajautusavain saattaa olla sama, tulee vastaan yhteentörmäyksiä (engl. *collision*). Yhteentörmäyksessä indeksissä, johon alkiota ollaan lisäämässä on jo toinen alkio. Wood [15, s.123-126] ja Kingston [7, s.306-318] listaavat eri tekniikoita, joita käytetään yhteentörmäyksien hallintaan. Näitä tekniikoita ovat: lineaarinen haku (engl. *Linear Probing*), ketjuttaminen (engl. *Chaining*), lokerointi (engl. *Bucketing*) ja kaksoishajautus (engl. *Double Hashing, Rehashing*).

4.3.1 Perusoperaatiot hajautusrakenteella

Hajautusrakenteet ovat staattisia tietorakenteita, koska ne käyttävät taulukoita sisäisinä rakenteinaan, joten niiden koko täytyy määrittää rakenteen luomisen tai tilan loppumisen yhteydessä. Hajautusrakenteelle pitää siis varata tilaa ennen kuin siihen voidaan lisätä alkioita. Toisin kuin luvussa 4.1 mainitulla List-luokalla, ei C#:n System.Collections.Generic-nimiavaruudesta löytyvällä Dictionary-luokalla ole *TrimExcess*-metodia vastaavaa metodia. Jos tällaiselle rakenteelle varattaisiin paljon enemmän muistia kuin se tulisi käyttämään, hukattaisiin muistia käyttämättömän tilan takia eikä sitä voitaisi vapauttaa niin kauan kuin tietorakenneilmentymä olisi olemassa. Toisaalta, vaikka Dictionary-luokalla olisikin vastaavanlainen metodi, uuden taulukon luomisen yhteydessä täytyisi alkiot hajauttaa uudestaan, mikä olisi vielä työläämpi operaatio kuin List-luokan tapauksessa. Tästä operaatiosta saatava hyöty saattaisi siis olla hyvin pieni sen aiheuttaman työmäärään suhteutettuna.

Hajautusrakenteeseen voidaan lisätä alkioita laskemalla lisättävälle alkiolle indeksi lisättävän alkion hajautusavaimen ja hajautusfunktion avulla kuten aikaisemmin todettiin. Indeksin laskeminen on vakioaikainen toimenpide, mutta lisäystä hidastavat yhteentörmäykset. Seuraavaksi käsitellään eri yhteentömäyksienkäsittelyalgoritmeja.

Lineaarisessa haussa yhteentörmäyksen tapahtuessa uusi indeksi lasketaan yksinkertaisesti lisäämällä laskettuun indeksiin yksi. Tätä jatketaan kunnes yhteentörmäystä ei enää tapahdu. Lineaarisen haun yhteentörmäyksien käsittely on yksinkertainen, mutta pahimmassa tapauksessa tehoton. Kuvitellaan tilanne, jossa kaikkien lisättävien alkioiden hajautusavain olisi sama. Tällöin kaikille alkioille laskettaisiin sama indeksi ja jokaisen alkion lisääminen kestäisi vähän kauemmin. Jos taulukko täytettäisiin näillä alkioilla viimeisen alkion lisäys olisi lineaariaikainen ($O(n)$) operaatio. Lineaarisessa haussa lisätyt alkiot kasaantuvat jo lisättyjen alkioiden ympärille. Hajautustaulukkoon muodostuu siis eräänlaisia rykelmiä. Täyttösuhteen lisääntyessä lineaarinen haku menettää tehokkuutta kuten Wood [15, s.315] mainitsee. Lisäksi kun taulukko on täynnä ja uusi alkio halutaan lisätä, täytyy taulukkoa kasvattaa ja samalla jokainen jo lisätty alkio on lisättävä uuteen taulukkoon laskemalla niille uudet indeksit hajautusfunktiota käyttäen. Jos hajautusrakenteessa on paljon alkioita, tämä on suhteellisen raskas toimenpide.

Ketjuttamisessa jokaiseen taulukon indeksiin asetetaan linkitetty lista, jonka solmuihin lisättävät alkiot ja hajautusarvot tallennetaan. Tällöin kun alkio lisätään, sille lasketaan indeksi hajautusavaimella ja hajautusfunktiolla ja alkio lisätään indek-

sissä olevaan linkitettyyn listaan, jos samalla avaimella ei ole jo lisätty alkiota. Jos samalla avaimella on aikaisemmin lisätty alkio, asetetaan uusi alkio kyseisen solmun alkioksi. Koska linkitetyt listat ovat dynaamisia tietorakenteita, ei taulukkoa sen luomisen jälkeen tarvitse periaatteessa kasvattaa. Ketjuttamisen ongelmana on mahdollinen epätasaisuus. Pahimmassa mahdollisessa tapauksessa kaikkien lisättävien alkioiden avaimille lasketaan sama indeksi hajautusfunktiolla, jolloin kaikki alkiot lisätään samaan ketjuun. Tällöin kaikki operaatiot ovat lineaariaikaisia $O(n)$.

Lokerointi eroaa ketjuttamisesta vain siten, että linkitettyjen listojen sijaan taulukon indekseihin asetetaan taulukot indekseille. Koska taulukot ovat staattisia rakenteita yksi lokero voi sisältää vain rajallisen määrän alkiota. Kun lokero lisäämisen yhteydessä tulee täyteen, on lokeron kokoa kasvatettava. Tämä tarkoittaa uuden isomman taulukon luomista lokerolle ja aiemman lokeron alkioiden kopioimista uuteen taulukkoon. Kuten ketjuttamisessa lokeroinnissakin vaarana on hajautustaulukon epätasainen täyttyminen. Jos lokeron kokoa ei haluta kasvattaa sen täyttymisen jälkeen, voidaan lokerotaulukon kokoa kasvattaa, jolloin yhteentörmäyksiä tallennetuille alkioille tapahtuu todennäköisesti vähemmän.

Ketjuttamisen ja lokeroinnin yhteydessä on myös päätettävä, kuinka suuria ketjuista tai lokeroista voi tulla. Itse lokero- tai ketjutaulukon pituus ei rajoita tietorakenteeseen tallennettavien alkioiden määrää, jos käytetään erillisiä taulukoita tai linkitettyjä listoja alkioiden tallentamiseen, koska yhteentörmäyksien tapahtuessa lokeroon tai ketjuun lisätään uusi alkio. Voidaan esimerkiksi päättää, että tallennettujen alkioiden määrä ei saa ylittää lokerotaulukon kapasiteettia. Näin voidaan välttää liian pitkiä lokeroita tai ketjuja.

Ketjuttamisen voi toteuttaa myös ilman erillisiä listoja alkioille (engl. *Coalesced chaining*). Tässä tekniikassa hajautusrakenteella on yksi taulukko, johon tallennetaan rakenteeseen lisätyt alkiot ja toinen taulukko indeksoinnille. Nämä taulukot ovat samanpituisia, jotta hajauttaminen toimisi. Alkiotaulukkoon tallennetaan alkiot tietorakenteessa, johon tallennetaan alkiolle laskettu hajautusavain, lisäämisessä käytetty avain, indeksi seuraavaan alkioon, jolle on laskettu sama indeksi ja alkio itse. Nämä rakenteet tallennetaan alkiotaulukkoon alusta lähtien peräkkäin. Indeksitaulukkoon tallennetaan alkiolle lasketun indeksin kohtaan alkiotaulukon indeksi, johon on tallennettu tähän indeksiin laskettu viimeisin alkio. Esimerkiksi jos hajautusrakenteesta olisi tehty seitsemän alkion kokoinen, olisi rakenteessa kaksi seitsemän alkion kokoista taulukkoa. Esimerkissä alkiotaulukossa näytetään vain laskettu hajautusavain (*hash*) ja seuraavan alkion indeksi (*next*). Aluksi indeksitaulukon kaikki arvot alustetaan arvoon -1 ilmaisemaan sitä, että taulukkoon ei ole vielä lisät-

ty alkioita.

0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1

Taulukko 4.1: Tyhjä indeksitaulukko

0	1	2	3	4	5	6
hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1

Taulukko 4.2: Tyhjä alkiotaulukko

Nyt lisätään hajautusrakenteeseen alkiot, joille on laskettu seuraavat hajautusavaimet: 57, 50, 35 ja 43 tässä järjestyksessä. Alkion 57 indeksi lasketaan $57 \bmod 7 = 1$ eli alkio pitäisi lisätä indeksiin 1. Nyt tarkastetaan indeksitaulukon arvo indeksistä 1, joka on -1 eli tämä indeksi on vapaa. Otetaan nykyinen alkioiden lukumäärä, joka on 0, muuttujaan talteen ja kasvatetaan alkioiden lukumäärää yhdellä. Nyt alkio 57 voidaan tallentaa alkioiden lukumäärän osoittamaan indeksiin arvotaulukossa eli indeksiin 0. Tämän jälkeen indeksitaulukon indeksiin 1 tallennetaan siihen viimeisimmän siihen lasketun alkion indeksi alkiotaulukossa eli 0. Tilanne ensimmäisen lisäyksen jälkeen on seuraavanlainen.

0	1	2	3	4	5	6
-1	0	-1	-1	-1	-1	-1

Taulukko 4.3: Indeksitaulukko alkion 57 lisäämisen jälkeen

0	1	2	3	4	5	6
hash: 57 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1

Taulukko 4.4: Alkiotaulukko alkion 57 lisäämisen jälkeen

Seuraavaksi lisätään alkio 50. Tälle lasketaan indeksi $50 \bmod 7 = 1$. Alkio lisätään taas indeksiin 1 eli tapahtuu yhteentörmäys. Ensin tarkistetaan arvo indeksitaulukon indeksistä 1. Arvo 0 eli se on suurempi tai yhtä suuri kuin 0. Nyt tarkistetaan alkiotaulukosta indeksistä 0, onko kyseisen alkion hajautusavain ja lisäyksessä

käytetty avain samoja. 57 ja 50 eivät ole samoja, joten jatketaan. Jos avaimet olisivat olleet samoja olisi nostettu poikkeus, koska oltiin lisäämässä jo lisätyllä avaimella. Algoritmia jatketaan siten, että otetaan arvo indeksitaulukon indeksistä 1, joka on 0, ja katsotaan alkiotaulukosta tästä indeksistä next-arvo, joka on -1. Arvo on pienempi kuin nolla, joten jatketaan seuraavaan vaiheeseen. Otetaan alkioiden lukumäärä talteen muuttujaan, joka on nyt 1 ja kasvatetaan alkioiden lukumäärää yhdellä. Tallennetaan alkio 50 alkiotaulukon indeksiin 1. Next-arvo otetaan indeksitaulukon indeksistä 1, joka on nolla. Nyt next-arvo viittaa edellä lisättyyn alkioon, jolle laskettiin sama indeksi. Seuraavaksi indeksitaulukon arvo indeksissä 1 asetetaan arvoon 1, joka on nyt lisätyn alkion indeksi alkiotaulukossa. Tilanne on tämän lisäyksen jälkeen seuraavanlainen.

0	1	2	3	4	5	6
-1	1	-1	-1	-1	-1	-1

Taulukko 4.5: Indeksitaulukko alkion 57 lisäämisen jälkeen

0	1	2	3	4	5	6
hash: 57 next: -1	hash: 50 next: 0	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1

Taulukko 4.6: Alkiotaulukko alkion 57 lisäämisen jälkeen

Seuraavaksi lisätään alkio 35. Lasketaan alkiolle indeksi $35 \bmod 7 = 0$. Indeksitaulukon indeksissä 0, on arvo -1, joten yhteentörmäystä ei tapahtunut. Lisätään alkio 35 seuraavaan indeksiin alkiotaulukossa eli indeksiin 2, joka on nykyinen alkioiden lukumäärä ja kasvatetaan alkioiden lukumäärää yhdellä. Next-arvo otetaan indeksitaulukon indeksistä 0, joka on -1. Asetetaan indeksitaulukon indeksiin 0 arvo 2, joka on alkion 35 indeksi arvotaulukossa. Tilanne on tämän lisäyksen jälkeen seuraavanlainen.

0	1	2	3	4	5	6
2	1	-1	-1	-1	-1	-1

Taulukko 4.7: Indeksitaulukko alkion 35 lisäämisen jälkeen

0	1	2	3	4	5	6
hash: 57 next: -1	hash: 50 next: 0	hash: 35 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1

Taulukko 4.8: Alkiotaulukko alkion 35 lisäämisen jälkeen

Lopuksi lisätään vielä alkio 43. Lasketaan alkiolle indeksi $43 \bmod 7 = 1$. Lisätään indeksiin 1 eli tapahtuu yhteentörmäys. Otetaan arvo indeksitaulukon indeksistä 1, joka on 1. Alkiotaulukon indeksissä 1 next-arvo on 0, joten siirrytään alkiontaulukon indeksiin 0. Indeksissä 0 next-arvo on -1, joten voidaan jatkaa. Otetaan alkioiden lukumäärä, joka on 3, talteen ja lisätään alkioiden lukumäärää yhdellä. Lisätään alkio 43 indeksiin 3 ja asettaan next-arvo samaksi kuin indeksitaulukon indeksissä 1, joka on 1. Seuraavaksi päivitetään indeksitaulukon indeksin 1 arvoksi alkion 43 indeksi alkiotaulukossa joka on 3. Tilanne on viimeisen alkion lisäämisen jälkeen seuraavanlainen.

0	1	2	3	4	5	6
2	3	-1	-1	-1	-1	-1

Taulukko 4.9: Indeksitaulukko alkion 43 lisäämisen jälkeen

0	1	2	3	4	5	6
hash: 57 next: -1	hash: 50 next: 0	hash: 35 next: -1	hash: 43 next: 1	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1

Taulukko 4.10: Alkiotaulukko alkion 43 lisäämisen jälkeen

Alkion löytämiseksi tästä rakenteesta käytetään aluksi hajautusavaimen avulla laskettua indeksiä ja tämän jälkeen seurataan next-arvoja, jos tallennettu avain ei ole haettu avain. Esimerkiksi, jos haluttaisiin löytää alkio 57, ensin lasketaan indeksi, johon alkio hajautettaisiin, joka on 1. Tämän jälkeen katsotaan viimeisin arvotaulukon indeksi indeksitaulukosta, joka on 3. Tämän jälkeen tarkastellaan indeksin 3 kohdalta löytyvää alkiota. Alkio ei ole etsitty alkio, joten jatketaan next-arvon osoittamaan indeksiin eli indeksiin 1. Indeksissä 1 ei ole haettua alkiota, joten jatketaan taas next-arvon osoittamaan indeksiin eli indeksiin 0. Indeksin 0 kohdalta löydetään haettu alkio 57.

Alkion poistamista varten tarvitaan kaksi uutta muuttujaa. Ensimmäisessä muuttujassa pidetään yllä vapaiden paikkojen lukumäärää (freeCount), jota kasvatetaan

aina, kun alkio poistetaan. Toisessa muuttujassa pidetään yllä viimeisintä indeksiä alkiotaulukosta, josta alkio on poistettu (freeList). Aluksi nämä muuttujat alustetaan siten, että vapaiden paikkojen lukumäärä on nolla (0) ja viimeisin indeksi on miinus yksi (-1). Kun seuraava alkio poistetaan, asetetaan alkiotaulukosta poistetun alkion indeksin kohdalle next-arvoksi edellinen freeList-arvo ja freeList-arvoksi poistetun alkion indeksi alkiotaulukossa. Vapaat paikat muodostavat siis samanlaisen ketjun kuin rakenteeseen lisätyt alkiot. Alkion poistamisen jälkeen seuraava lisäysoperaatio suoritetaan eri tavalla kuin edellä mainituissa tapauksissa. Jos alkion lisäyksen yhteydessä vapaiden paikkojen lukumäärä on suurempi kuin nolla, lisätään alkio viimeisimpään vapaaseen paikkaan sen sijaan, että alkio lisättäisiin listan loppuun. Uusi alkio lisätään siis freeList-muuttujan osoittaman indeksin kohdalle. Tämän jälkeen freeList-muuttujan arvoksi asetetaan alkion indeksin kohdalla oleva next-arvo, joka kertoo seuraavan vapaan paikan indeksin. Tämän jälkeen alkion indeksin next-arvo asetetaan samalla tavalla kuin normaalissa tapauksessa eli se otetaan indeksitaulukon arvosta.

Jos nyt esimerkiksi haluttaisiin poistaa alkio 50, täytyisi alkio ensin löytää yllä mainitulla algoritmilla. Tämän lisäksi poisto-operaation ajan pidettäisiin yllä ketjussa kulkemisen edellistä indeksiä. Ensin alkiolle laskettaisiin indeksi, joka on 1. Indeksin 1 kohdalta indeksitaulukosta nähdään ketjun viimeisen alkion indeksi alkiotaulukossa, joka on 3. Koska alkio 50 ei ole indeksin 3 kohdalla jatketaan next-arvon perusteella seuraavan indeksin kohdalle, mutta pidetään yllä nykyistä indeksiä muuttujassa. Next-arvon kautta mennään indeksin 1 kohdalle. Alkio 50 löytyy tästä indeksistä, joten alkio voidaan poistaa. Ensin edellisen indeksin kohdalla olevan alkion next-arvoksi asettaan indeksin 1 next-arvo, koska tämä alkio poistetaan. Tämän jälkeen freeCount-muuttujan arvoa kasvatetaan yhdellä. Indeksin 1 kohdalle next-arvoksi asetetaan nykyinen freeList-muuttujan arvo, joka on -1 ja freeList-muuttujan arvoksi asetetaan 1. Tämän lisäksi indeksin 1 kohdalla alkiotaulukossa hash-arvoksi asetetaan -1, avaimen arvoksi asetetaan kyseisen tietotyypin oletusarvo ja alkion arvoksi asetetaan myös kyseisen tietotyypin oletusarvo. Alkion 50 poistamisen jälkeen taulukot näyttäisivät seuraavanlaiselta.

0	1	2	3	4	5	6
2	3	-1	-1	-1	-1	-1

Taulukko 4.11: Indeksitaulukko alkion 50 poistamisen jälkeen

0	1	2	3	4	5	6
hash: 57 next: -1	hash: -1 next: -1	hash: 35 next: -1	hash: 43 next: 0	hash: 0 next: -1	hash: 0 next: -1	hash: 0 next: -1

Taulukko 4.12: Alkiotaulukko alkion 50 poistamisen jälkeen

Tämä on yksinkertaistettu versio ketjutusalgoritmista, jota esimerkiksi C#:n Dictionary-luokka käyttää. Tässä algoritmissa on se etu, että ei tarvitse keksiä sääntöjä taulukon koon kasvattamiselle, koska kun alkiotaulukko tulee täyteen, on taulukko vaihdettava suurempaan. Tällöin alkiotaulukon next-arvot ja indeksitaulukon viimeisimmät indeksit on laskettava uudestaan. Tämä tehdään siten, että luodaan uudet suuremmat taulukot indeksejä ja alkioita varten, kopioidaan vanhan alkiotaulukon alkiot uuteen taulukkoon ja alustetaan uuden indeksitaulukon arvot arvoon -1. Tämän jälkeen käydään lisätyt alkiot läpi alkiotaulukosta ja lasketaan alkiolle uusi indeksi uuden taulukon koolla. Otetaan alkion next-arvoksi arvo indeksitaulukosta uudella indeksillä ja päivitetään tämän jälkeen indeksitaulukon uuden indeksin arvoksi kyseessä olevan alkion indeksi alkiotaulukossa.

Kaksoishajautuksessa lisättävät alkiot lisätään suoraan hajautustaulukkoon eikä lokeroita tai ketjuja käytetä. Toisin kuin lineaarisessa haussa, yhteentörmäyksen tapahtuessa indeksiä ei lisätä yhdellä, vaan avaimelle lasketaan uusi hajautusarvo. Tällä tekniikalla yritetään estää rykelmien muodostuminen taulukkoon. Kaksoishajautuksessa käytetään kahta hajautusfunktiota. Ensimmäisellä funktiolla lasketaan indeksi lisättävälle alkiolle, ja jos tapahtuu yhteentörmäys, toisella funktiolla lasketaan uusi indeksi. Haasteena kaksoishajautuksessa on estää syklinen indeksien laskeminen. Jos $h(x)$ on ensimmäinen hajautusfunktio ja $g_i(x)$ on toinen hajautusfunktio, kun ensimmäinen indeksi lasketaan x:lle funktiolla $h(x)$ ja tapahtuu yhteentörmäys, lasketaan uusi indeksi funktiolla $g_1(x)$. Tällöin funktioiden tulisi olla sellaisia, että $g_i(x)$:n arvo ei ole minkään funktion $g_j(x)$ arvo, jossa $j < i$ tai funktion $h(x)$ arvo. Täydellisessä kaksoishajautuksessa tulisi myös käydä kaikki taulukon indeksit läpi, jos jokaisella $g_i(x)$:n kutsulla tapahtuu yhteentörmäys.

4.3.2 Hajautusrakenteiden vahvuudet

Kuten taulukkorakenteidenkin tapauksessa hajautusrakenteiden vahvuus on se, että ne ovat hajasaantirakenteita. Hajautusrakenteiden tapauksessa tämä on vielä suurempi vahvuus, koska hajasaantirakenteella mitä tahansa voidaan käyttää avaime-

na, kun taas taulukon tapauksessa alkioon päästään vain kokonaislukuindeksin kautta. Toki hajasaantirakenteissakin käytetään taulukkoa perustietorakenteena, jolloin alkioihin päästään kokonaislukuindeksien kautta, mutta hajautusrakenteessa mille tahansa alkiolle voidaan laskea hajautusavain ja sen kautta taulukon indeksi, johon alkio kuuluu. Lisäksi hajautusrakenteen käyttäjän ei tarvitse tietää, kuinka suuri hajautusrakenteen taulukko on, koska hajautusfunktioilla saadaan aina arvo, joka on taulukon rajoissa. Ja yhteentörmäyksien käsittelyn ansiosta ei haittaa, vaikka kahdelle eri alkiolle laskettaisiinkin sama taulukon indeksi.

4.3.3 Hajautusrakenteiden heikkoudet

Hajautusrakenteiden heikkoudeksi voidaan laskea niiden muistinkäyttö. Koska hajautusrakenteet käyttävät taulukkoa perustietorakenteena, ovat ne staattisia tietorakenteita. Tämän vuoksi rakenteelle täytyy varata muisti sen luomisen yhteydessä, ja jos rakenteeseen lisätään huomattavasti vähemmän alkioita verrattuna sille varattuun tilaan nähden, hukataan muistia. Sen lisäksi, että rakenteelle täytyy varata staattisesti muistia, sitä varataan usein esimerkiksi lineaarisen haun ja kaksoishajautuksen tapauksessa hieman ylimääräistä. Tätä kutsutaan täyttösuhteeksi (engl. *Load factor*). Kun hajautusrakenteen taulukko täyttyy, yhteentörmäysten todennäköisyys kasvaa. Tämän vuoksi hajautusrakenteen taulukko kannattaa vaihtaa suurempaan, vaikka taulukko ei olisikaan täynnä. Tämän vuoksi hajautusrakenteessa hukataan aina vähän muistia, jos käytetään lineaarista hakua tai kaksoishajautusta. Lokeroinnin ja ketjutuksen tapauksessa käytetään myös ylimääräistä muistia lokeroille tai ketjuille tai indeksitaulukoille.

Vaikka hajautusrakenteet ovat nopeita ja niiden parhaan tapauksen kompleksisuus kaikille perusoperaatioille on $O(1)$, ne eivät aina ole nopeimpia tietorakenteita. Kuten luvun 5.1 tuloksista voidaan huomata, listarakenteet ovat hajautusrakenteita nopeampia lisäysoperaatiossa. Hajautusavainten ja taulukon indeksin laskemiseen sekä yhteentörmäysten käsittelyyn kuluu aina aikaa. Tästä syystä ne eivät sovellu parhaiten tilanteisiin, joissa lisäysoperaatioissa vaaditaan parasta suorituskykyä.

4.4 Puut

Toisin kuin tutkielmassa aiemmin esitellyt rakenteet, puut (engl. *Tree*) eivät ole lineaarisia rakenteita. Kuten Wood [15, s.141] mainitsee, puut tarjoavat hierarkkisen ta-

van järjestää dataa. Vaikka puut ovat hierarkkisia, voidaan niihin tallennettava data silti joissain tapauksissa tallentaa lineaariseen rakenteeseen, kuten taulukkoon. Esimerkiksi binääripuu voidaan tallentaa taulukkoon seuraavasti. Juurisolmu tallennetaan indeksin 1 kohdalle. Juurisolmun vasen lapsisolmu tallennetaan indeksin $1 * 2$ eli indeksin 2 kohdalle ja juurisolmun oikea lapsisolmu tallennetaan indeksin $1 * 2 + 1$ eli indeksin 3 kohdalle. Tätä jatketaan siten, että solmun vasen lapsisolmu tallennetaan aina indeksin $i * 2$ kohdalle ja solmun oikea lapsisolmu tallennetaan indeksin $i * 2 + 1$ kohdalle, joissa i on solmun indeksi. Tässä luvussa käsitellään binääripuurakennetta.

Puurakenne koostuu solmuista. Puussa on aina vähintään yksi solmu (engl. *Node*), joka on sen juuri. Juurella voi olla lapsisolmuja (engl. *Child node*), joilla taas voi olla omia lapsisolmuja. Jos solmulla ei ole lapsisolmuja kutsutaan sitä lehtisolmuksi (engl. *Leaf node*). Jokainen solmu, joka ei ole puun juurisolmu, muodostaa puun alipuun (engl. *Sub tree*). Esimerkiksi tämän tutkielman voidaan käsittää muodostavan puurakenteen, jossa dokumentti on puun juurisolmu, pääluvut ovat juuren lapsisolmuja ja aliluvut ovat päälukujen lapsisolmuja.

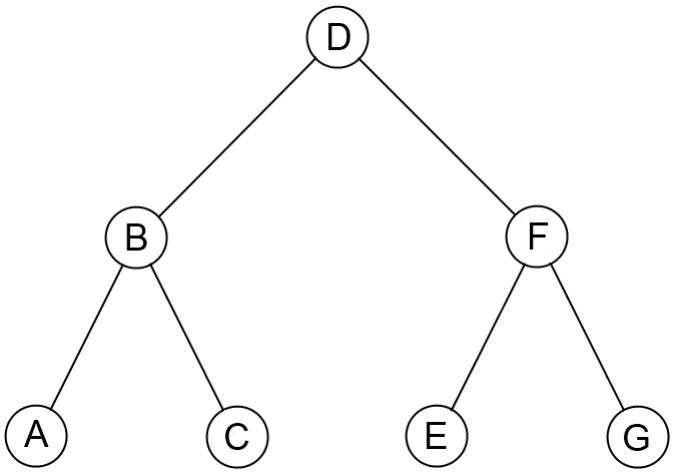
Puut voivat olla säännöllisiä tai epäsäännöllisiä. Säännöllisessä puussa jokaisen solmun lapsisolmujen määrällä on yläraja. Lisäksi säännöllisessä puussa solmut ovat usein järjestyksessä. Esimerkiksi binääripuussa solmulla voi olla maksimissaan kaksi lapsisolmua ja solmut ovat järjestyksessä siten, että vasen lapsisolmu on järjestyksessä aiemmin kuin solmu itse ja oikea lapsisolmu on järjestyksessä solmun jälkeen. Epäsäännöllisessä puussa lapsisolmujen lukumäärää ei ole rajoitettu eivätkä solmut välttämättä ole järjestyksessä.

4.4.1 Binääripuut

Tässä luvussa keskitytään säännöllisiin puihin, ja erityisesti binääripuihin (engl. *Binary search tree*), koska näillä rakenteilla saavutetaan etuja operaatioiden suoritusajan suhteen. Binääripuut ovat säännöllisiä ja järjestettyjä puurakenteita. Binääripuun jokaisessa solmulla on enintään kaksi lapsisolmua. Solmun vasemmassa lapsisolmussa on solmu, jonka arvo ja kaikkien lapsisolmujen arvot ovat järjestyksessä ennen kyseisen solmun arvoa. Lisäksi solmun oikeassa lapsisolmussa on solmu, jonka arvo ja, jonka kaikkien lapsisolmujen arvot ovat järjestyksessä kyseisen solmun arvon jälkeen.

Binääripuiden läpikäyntiin on kolme tapaa, jotka Knuth [8, s.319] mainitsee: esijärjestys (engl. *preorder*), sisäjärjestys (engl. *inorder*) ja jälkijärjestys (*engl. postorder*).

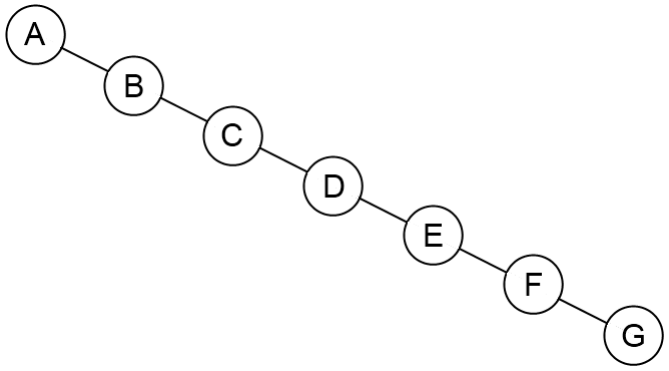
Esijärjestyksessä otetaan ensin arvo solmusta, minkä jälkeen käydään vasen alipuu läpi esijärjestyksessä, ja lopuksi käydään oikea alipuu läpi esijärjestyksessä. sisäjärjestyksessä ensin käydään vasen alipuu läpi järjestyksessä, minkä jälkeen otetaan arvo solmusta, ja tämän jälkeen käydään oikea alipuu läpi järjestyksessä. Jälkijärjestyksessä ensin käydään vasen alipuu läpi jälkijärjestyksessä, minkä jälkeen käydään oikea alipuu läpi jälkijärjestyksessä, ja lopuksi otetaan arvo solmusta.



Kuva 4.3: Esimerkkibinääripuu

Esimerkkibinääripuusta arvot saadaan eri järjestyksessä läpikäyntitavasta riippuen. Esijärjestyksessä arvot ovat: D, B, A, C, F, E, G. Järjestyksessä arvot ovat: A, B, C, D, E, F, G. Jälkijärjestyksessä arvot ovat: A, C, B, E, G, F, D.

Kuvan 4.3 puu on tasapainossa. Binääripuu on tehokkaimmillaan, kun puu on tasapainossa, eli vasen ja oikea alipuu ovat tasapainossa ja samansyvyisiä, kuten Baldwin ja Scragg [1, s.446] mainitsevat. Jos puuta ei tasapainoteta, sen tehokkuus heikkenee. Binääripuu voidaan pitää tasapainossa, joko lisäämällä arvot juuri oikeassa järjestyksessä, tai tasapainottamalla puuta arvojen lisäämisen välissä. Vastaavasti, jos arvot lisätään puuhun huonossa järjestyksessä, tulee puusta epätasapainoinen. Esimerkiksi, jos esimerkkipuun arvot lisättäisiin puuhun järjestyksessä: A, B, C, D, E, F, G, tulisi puusta seuraavan näköinen:



Kuva 4.4: Esimerkkibinääripuu epätasapainossa

On olemassa puurakenteita, jotka huolehtivat omasta tasapainostaan eli tasapainottavat itseään alkioiden lisäyksen ja poiston yhteydessä. Yksi tällainen puurakenne on AVL-puu, joka on saanut nimensä keksiöidensä sukunimien ensimmäisistä kirjaimista Adelson-Velskii ja Landis. AVL-puu pitää itsensä tasapainossa tekemällä tasapainotusta tarvittaessa jokaisen lisäys- ja poisto-operaation jälkeen. Tasapainottaminen suoritetaan puun kierroilla, jotka esitellään luvussa 4.4.2.

4.4.2 Perusoperaatiot binääripuulla

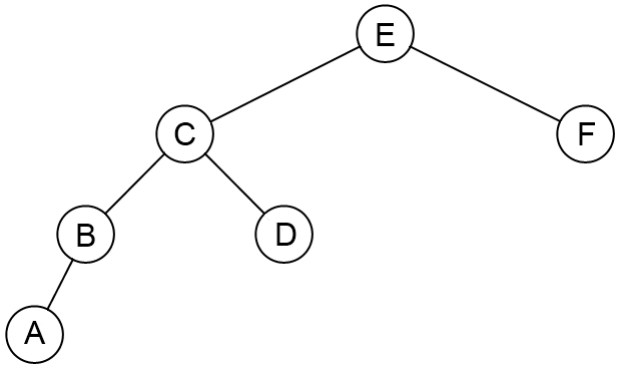
Binääripuu on dynaaminen tietorakenne, joten sille ei tarvitse varata muistia staattisesti sen luomisen yhteydessä. Alkioiden lisäämisen, etsimisen ja poistamisen lisäksi, binääripuun käyttämiseen liittyy vahvasti puun tasapainottaminen. Tasapainoisen binääripuun operaatiot voidaan suorittaa logaritmisessa ajassa ($O(log_2 n)$), kun taas epätasapainoisen binääripuun operaatiot ovat lineaariaikaisia ($O(n)$). Binääripuu voidaan toteuttaa myös taulukkorakenteen avulla, jolloin rakenne olisi staattinen, mutta tutkielmassa käsitellään dynaamista binääripuuta, koska taulukkorakenteita käsiteltiin jo luvussa 4.1.

Kun binääripuuhun lisätään alkio lähdetään binääripuuta käymään läpi sen juuresta. Jos juuri on tyhjä, lisättävä alkio lisätään uuteen juurisolmuun. Jos juurisolmu ei ole tyhjä, verrataan lisättävää alkiota ja juurisolmun alkiota. Jos lisättävä alkio on järjestyksessä juurisolmun alkiota ennen, siirrytään juurisolmun vasempaan alipuuhun. Jos lisättävä alkio on järjestyksessä juurisolmun alkion jälkeen, siirrytään juurisolmun oikeaan alipuuhun. Jos taas alkio on sama kuin solmun alkio, voidaan solmun alkio korvata lisättävällä alkiolla tai jättää alkio lisäämättä. Jos alipuu on tyhjä, tehdään uusi solmu, joka asetetaan juurisolmun vasemmaksi tai oikeaksi lapsisolmuksi vertailun perusteella. Jos alipuu ei ole tyhjä, toistetaan vertailuoperaatio

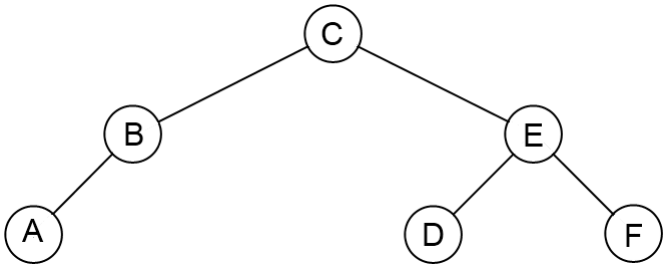
rekursiivisesti alipuun solmuille, kunnes alkiolle löydetään paikka.

Kuten aiemmin mainittiin, lisäysoperaatio voidaan tehdä logaritmisessa ajassa, jos puu on tasapainossa. Koska tasapainoisen puun vasen ja oikea alipuu ovat samansyvyiset, on niissä suuriin piirtein saman verran solmuja. Eli jos lisättävän alkion paikka ei ole juurisolmussa, on sen paikka joko vasemmassa alipuussa tai oikeassa alipuussa. Eli jokaisen vertailuoperaation jälkeen mahdollisten paikkojen määrä lisättävälle alkiolle puolittuu.

Puurakenne ei kuitenkaan pysy tasapainossa, jos alkioita ei lisätä oikeassa järjestyksessä, tai jos puuta ei tasapainoteta lisäysoperaatioiden välissä. Puu voidaan tasapainottaa puun kierroilla (engl. *Tree rotation*). Puun kierrossa puusta etsitään solmu, jonka toinen alipuu on liian paljon korkeampi kuin toinen. Esimerkiksi AVL-puun tapauksessa tätä eroa kutsutaan tasapainokertoimeksi (engl. *Balance factor*) ja se lasketaan vähentämällä vasemman alipuun korkeudesta oikean alipuun korkeus. Jos tasapainokerroin on suurempi kuin kaksi tai vähemmän kuin miinus kaksi, solmu on epätasapainossa. Solmu saadaan tasapainoon kiertämällä puuta solmun suhteen. Jos vasen alipuu on korkeampi kuin oikea alipuu, kierretään puuta vasemmalta oikealle. Jos oikea alipuu on korkeampi kuin vasen alipuu, kierretään puuta oikealta vasemmalle. Kun puuta kierretään, alipuun, joka on kiertosuunnassa, korkeus kasvaa yhdellä. Vastaavasti toisen alipuun korkeus vähenee yhdellä.



Kuva 4.5: Esimerkkibinääripuu tarvitsee kierron oikealle E:n suhteen



Kuva 4.6: Esimerkkibinääripuu puun kierron oikealle jälkeen

Kuvassa 4.5 binääripuu on epätasapainossa solmun E suhteen. Solmun E vasemman alipuun korkeus on 3 ja oikean alipuun korkeus on 1. Puuta tarvitsee siis kiertää oikealle solmun E suhteen. Oikealle kierrossa solmun N suhteen N:n paikan ottaa sen vasen lapsisolmu. Oikeankierron algoritmi on seuraavanlainen:

```
1  protected void RotateRight(BinaryTreeNode<TKey, TValue> node)
2  {
3      BinaryTreeNode<TKey, TValue> parent = node.Parent;
4      bool isRightChild =
5          parent != null ? node == parent.Right : false;
6      var pivot = node.Left;
7      node.Left = pivot.Right;
8      if(pivot.Right != null)
9          pivot.Right.Parent = node;
10     pivot.Right = node;
11
12     node.Parent = pivot;
13     pivot.Parent = parent;
14
15     if (parent == null)
16     {
17         root = pivot;
18         root.Parent = null;
19     }
20
21     else if (isRightChild)
22     {
23         parent.Right = pivot;
24     }
25
26     else
27     {
28         parent.Left = pivot;
```

```
29      }
30  }
```

Oikean kierron algoritmissa kuvan 4.5 puun solmu E annetaan ensimmäiseksi parametriksi. Algoritmissa pivot-muuttujaksi asetetaan E-solmun vasen lapsisolmu, joka on C. E-solmun vasemmaksi lapseksi asetetaan pivot-solmun oikea lapsi, joka on D. Pivot-solmun oikeaksi lapseksi asetetaan E. Koska E on puun juuri, puun juureksi asetetaan pivot-solmu. Tuloksena saadaan kuvan 4.6 mukainen puu.

Vasemman kierron algoritmi on peilikuva oikeankierron algoritmista:

```
1  protected void RotateLeft(BinaryTreeNode<TKey, TValue> node)
2  {
3      BinaryTreeNode<TKey, TValue> parent = node.Parent;
4      bool isRightChild =
5          parent != null ? node == parent.Right : false;
6      var pivot = node.Right;
7      node.Right = pivot.Left;
8      if(pivot.Left != null)
9          pivot.Left.Parent = node;
10     pivot.Left = node;
11
12     node.Parent = pivot;
13     pivot.Parent = parent;
14
15     if (parent == null)
16     {
17         root = pivot;
18         root.Parent = null;
19     }
20
21     else if (isRightChild)
22     {
23         parent.Right = pivot;
24     }
25
26     else
27     {
28         parent.Left = pivot;
29     }
30 }
```

Alkion poistamisessa binääripuusta on kolme tapausta. Jos poistettava alkio on

lehtisolmu, asetetaan solmun vanhempisolmun viite tyhjäksi. Lisäksi jos poistettava solmu on juurisolmu, asetetaan juuri tyhjäksi. Jos poistettavan alkion solmulla on yksi lapsisolmu, asetetaan poistettavan solmun vanhempisolmun lapsiviite osoittamaan poistettavan solmun lapsisolmuun. Lisäksi jos poistettavan alkion solmu on juurisolmu asetetaan juurisolmuksi poistettavan alkion lapsisolmu. Jos poistettavan alkion solmulla on kaksi lapsisolmua, etsitään joko poistettavan solmun vasemman alipuun suurimman arvon solmu tai oikean alipuun pienimmän arvon solmu ja korvataan poistettavan arvon solmun arvo löydetyn solmun arvolla ja poistetaan löydetty solmu. Valintaa vasemman alipuun suurimman ja oikean alipuun pienimmän välillä voidaan vuorotella, jos halutaan, että puu ei päädy helposti epätasapainoon. Jos esimerkiksi valitaan aina oikean alipuun pienin korvattavaksi, pienenee oikea alipuu ja vasen alipuu jää samankokoiseksi, jos poistot osuvat samaan solmuun.

4.4.3 Puiden vahvuudet

Puiden, erityisesti binääripuiden, vahvuudeksi voidaan lukea niihin suoritettavien operaatioiden kompleksisuus. Koska aikavaativuus kaikille perusoperaatioille on tasapainoiselle binääripuulle $O(log_2(n))$, absoluuttinen operaatioihin käytetty aika kasvaa hitaasti puun solmujen määrän lisääntyessä. Toisaalta operaatiot ovat aina vähintään $O(log_2(n))$ aikaisia, kun taas esimerkiksi taulukkorakenteella lisäysoperaatio voi parhaassa tapauksessa olla vakioaikainen ja linkitetyllä listalla se on aina vakioaikainen, kuten luvuissa 4.1 ja 4.2 mainittiin.

Puiden vahvuus on myös se, että ne ovat dynaamisia tietorakenteita. Koska puut koostuvat solmuista, joista on viitteet niiden lapsisolmuihin, ei puille tarvitse varata muistia staattisesti niiden luomisen yhteydessä. Tästä syystä puita käytettäessä ei koskaan hukata muistia käyttämättömän tilan vuoksi.

4.4.4 Puiden heikkoudet

Puiden heikkoudeksi voidaan laskea se, että päästäkseen logaritmiseen kompleksisuuteen, ne vaativat tasapainotusta. Jos tavalliseen binääripuuhun lisätään alkiot huonossa järjestyksessä, tulee sen operaatioiden kompleksisuudeksi $O(n)$. Itseään tasapainottavat puut, kuten AVL-puu, huolehtivat puun tasapainotuksesta itse, mutta tasapainotus voi olla raskas toimenpide ja AVL-puun tapauksessa tämä toimenpide suoritetaan jokaisen lisäyksen ja poiston yhteydessä, jos siihen on tarvetta. Tästä syystä puut eivät ole välttämättä suorituskyvyn suhteen optimaalisia tietorakentei-

ta.

Kuten linkitetyn listan tapauksessa, puu koostuu solmuista, joista on viitteitä muihin solmuihin. Binääripuun tapauksessa alkion tallentamiseen puuhun tarvitaan muistia itse alkiolle, puun solmulle sekä kahdelle viitteelle lapsisolmuihin. Joissakin tapauksissa solmuista on viite myös sen vanhempisolmuun, jolloin muistia tarvitaan kolmelle viitteelle. Tällöin jos tallennettavalle alkiolle tarvitaan vähän muistia verrattuna solmuihin ja viitteisiin, eivät puut ole optimaalisia tietorakenteita muistinkäytön suhteen.

4.5 Rajapinnat

Tietorakenneluokkien lisäksi tietorakenteiden luokkakirjastoissa on yleensä määritelty rajapintatyyppejä (engl. *Interface*), joissa on määritelty eri abstraktien tietotyyppien operaatioita, joita tietorakenneluokat toteuttavat. Tietorakenneluokkia yleensä käytetäänkin näiden rajapintojen kautta, kuten rajapintaerotteluperiaate (engl. *Interface Segregation Principle*) ehdottaa. C#:n Systems.Collections- ja Systems.Collection.Generic-nimiavaruuksissa on määritelty rajapintoja, joita tutkielmassa käsiteltävät tietorakenneluokat toteuttavat. Näitä rajapintoja ovat: IEnumerable, ICollecion, IList, ISet ja IDictionary.

IEnumerable-rajapintaa käytetään, kun halutaan vain käydä läpi tietorakenteen alkiot. Rajapinnan kautta voidaan kysyä enumeraattoria GetEnumerator-metodilla. Tätä metodia käytetään implisiittisesti, kun tietorakenne käydään läpi *foreach*-silmukassa. Tämän rajapinnan kautta ei tietorakennetta voida muokata.

ICollection-rajapinnan kautta voidaan tietorakenteeseen kohdistaa perusoperaatiot kuten lisäys (Add-metodi), etsiminen (Contains-metodi) ja poisto (Remove-metodi), sekä kysyä tietorakenteen kokoa Count-ominaisuuden kautta. Contains-metodilla voidaan kysyä, onko tietorakenteessa tietty alkio. Tietorakenne voidaan myös tyhjentää Clear-metodin avulla ja tietorakenteen alkiot voidaan kopioida taulukkoon CopyTo-metodin avulla. ICollection toteuttaa IEnumerable-rajapinnan, joten kaikki IEnumerable-rajapinnan toiminnot voidaan suorittaa ICollection-rajapinnan kautta.

IList on rajapinta indeksoidulle tietorakenteelle. IList toteuttaa ICollection-rajapinnan, joten kaikki ICollection-rajapinnan operaatiot voidaan suorittaa IList-rajapinnan kautta. ICollection-rajapinnassa määriteltyjen metodien lisäksi IList-rajapinnan kautta voidaan lisätä alkio tiettyyn indeksiin Insert-metodilla, kysyä alkion

indeksiä IndexOf-metodilla ja poistaa tietyn indeksin kohdalla oleva alkio RemoveAt-metodilla. IList-rajapinnassa on myös määritelty indekseri, jolloin sitä voidaan käyttää samanlaisella syntaksilla kuin taulukkoa.

ISet on rajapinta joukoille. ISet toteuttaa ICollection-rajapinnan, joten kaikki ICollection-rajapinnan operaatiot voidaan suorittaa ISet-rajapinnan kautta. Sen lisäksi, että ISet-rajapinnassa on ICollection-rajapinnan Add-metodi, on siinä myös toinen Add-metodi, joka palauttaa tiedon siitä, onnistuiko alkion lisäys vai ei. Rajapinnassa on määritelty yleisiä joukkojen operaatiota. ExceptWith-metodilla voidaan joukosta poistaa kaikki parametrina annetussa kokoelmassa olevat alkiot. IntersectWith-metodilla voidaan joukosta tehdä joukon ja parametrina annetun kokoelman leikkaus. IsProperSubsetOf-metodilla voidaan kysyä, onko joukko parametrina annetun kokoelman todellinen osajoukko. IsProperSupersetOf-metodilla voidaan kysyä, onko parametrina annettu kokoelma joukon todellinen osajoukko. Nämä metodit palauttavat epätoden, jos joukko ja parametrina annettua kokoelma sisältävät täsmälleen samanarvoiset alkiot. IsSubSetOf-metodilla voidaan kysyä, onko joukko parametrina annetun kokoelman osajoukko. IsSuperSetOf-metodilla voidaan kysyä, onko parametrina annettu kokoelma joukon osajoukko. Nämä metodit palauttavat toden myös, jos joukko ja metodin parametri sisältävät täsmälleen samanarvoiset alkiot. Overlaps-metodilla voidaan kysyä, leikkaako joukko parametrina annettua kokoelmaa. SetEquals-metodilla voidaan kysyä onko joukossa ja parametrina annetussa kokoelmassa täsmälleen samanarvoisen alkiot. SymmetricExceptWith-metodilla voidaan joukkoa muokata siten, että se sisältää vain alkiot, jotka ovat joukossa tai parametrina annetussa kokoelmassa, mutta ei molemmissa. UnionWith-metodilla voidaan joukosta tehdä joukon ja parametrina annetun kokoelman yhdiste.

IDictionary on rajapinta sanakirjoille. IDictionay-rajapinnan kautta alkiot yhdistetään avaimiin. IDictionary toteuttaa ICollection-rajapinnan, mutta alkioiden sijaan metodeilla käsitellään avain-alkio-pareja. Rajapinnassa on määritelty indekseri, jolla tietorakennetta voidaan käyttää kuten assosiatiivista taulukkoa. Rajapinta ei kuitenkaan vaadi, että rajapinnan toteuttava tietorakenne käyttäisi tallennusrakenteenaan taulukkoa. IDictionary-rajapinnan Add-metodilla voidaan sanakirjaan lisätä alkio. Metodi ottaa parametriksi lisättävän alkion lisäksi avaimen, jolle alkio lisätään. Jos tietorakenteeseen on jo lisätty alkio kyseisellä alkiolla, tulisi tietorakenteen tällöin heittää poikkeus. ContainsKey-metodilla voidaan kysyä, onko tietorakenteessa alkiota parametrina annetulla avaimella. Remove-metodilla voidaan poistaa alkio, joka on liitetty parametrina annettuun avaimeen. TryGetValue-metodilla voidaan yrit-

tää palauttaa alkio, joka on liitetty parametrina annettuun avaimeen. Metodi palauttaa tiedon siitä, löytyikö alkio, ja alkio palautetaan metodin out-parametrina. Alkio voidaan palauttaa myös indekserin avulla, mutta jos kyseisellä avaimella ei ole alkiota, kuuluu tietorakenteen heittää poikkeus.

Rajapinnoilla voidaan siis määritellä eri abstraktien tietotyyppien operaatioita, joita tietorakenneluokat toteuttavat. Rajapinnat kuitenkin määrittelevät vain, mitä operaatioita toteuttavan luokan kuuluu toteuttaa. Rajapinnat eivät ota kantaa toteutustapaan. Esimerkiksi ICollection voidaan toteuttaa taulukkoa käyttävänä listana, linkitettynä listana tai hajautusrakenteena. ISet taas voidaan toteuttaa esimerkiksi hajautusrakenteena tai binääripuuna, joissa avaimet ovat rakenteen alkioita. Samoin IDictionary voidaan toteuttaa hajautusrakenteena tai binääripuuna.

Kun tietorakenneluokkia käytetään rajapintojen kautta, voidaan tietorakenne vaihtaa toiseen toteutukseen koskematta ollenkaan tietorakennetta käyttävään koodiin. Tämä on koodin ylläpidettävyyden kannalta hyvä ratkaisu, mutta tällä voidaan huonontaa sovelluksen suorituskykyä, jos tietorakenne valitaan huonosti. Antti Valmari kirjoittaa artikkelissaan [12] tapauksesta, jossa sovelluksen suorituskyky oli erityisen huono. Hitaus jäljitettiin *while*-silmukkaan, jossa C++:n list-luokkaa käsiteltiin. While-silmukan ehtona oli *kaaret.size() != 0* ja silmukassa poistettiin listasta alkioita. Listan size-metodi oli hidas, koska metodi kävi jokaisen kutsun yhteydessä kaikki listan alkiot läpi alkioiden lukumäärää selvitettäessä. Ongelma kuitenkin poistui, kun koodi käännettiin eri kääntäjällä. Ero oli siis eri kääntäjien list-luokan toteutuksessa. Vaikka tapauksessa ei käytettykään rajapintoja ja ero oli eri kääntäjissä, voidaan tapausta verrata myös rajapintojen käyttöön. Esimerkiksi ICollection-rajapinta voitaisiin toteuttaa siten, että Count-ominaisuus kävisi kaikki tietorakenteen alkiot läpi, tai siten, että alkioiden lukumäärää pidettäisiin yllä kokonaislukumuuttujassa. Jos tietorakenne, joka pitäisi alkioiden lukumäärä yllä, vaihdettaisiin tietorakenteeseen, joka laskisi alkioiden määrään joka kerta uudestaan, voisi suorituskyky huonontua huomattavasti, jos koodi käyttäisi Count-ominaisuutta paljon.

Jos toteutettaisiin adaptoituva tietorakenne, mitä rajapintoja sen tulisi toteuttaa? Ei olisi järkevää yrittää toteuttaa tietorakennetta, joka toteuttaisi kaikki yllä mainitut tietorakenteet. Esimerkiksi ei olisi järkevää toteuttaa IList- ja IDictionary-rajapintaa samassa tietorakenteessa, koska nämä rajapinnat määrittelevät operaatioita, jotka ovat toisiaan poissulkevia. Esimerkiksi IList-rajapinnan kautta on mahdollista lisätä alkio tietyn indeksi kohdalle, kun taas IDictionary-rajapinnalle tällainen operaatio ei ole mielekäs, koska indeksi lasketaan aina hajautusavaimen kautta.

Hyvin erikoistuneiden rajapintojen toteuttaminen adaptoituvassa tietorakenteessa ei ole mielekästä. Erikoistuneille rajapinnoille, kuten IDictionary, on jo olemassa toteutuksia, jotka suoriutuvat rajapinnan operaatioista tehokkaasti. Lisäksi abstraktit tietotyypit, joihin jotkut rajapinnat liittyvät, sisältävät rajoitteita, jotka välttämättä hidastaisivat joitakin operaatioita. Esimerkiksi joukko sisältää rajoitteen, että tietorakenne ei voi sisältää duplikaatteja. Tällöin esimerkiksi lisäysoperaatio ei voisi olla kaikista tehokkain, koska jokaisen lisäysoperaation yhteydessä täytyisi tarkistaa, onko tietorakenteessa jo samanarvoista alkiota.

Olemassa olevista perusrajapinnoista adaptoituvan tietorakenteen tulisi kuitenkin toteuttaa ainakin IEnumerable-rajapinta, koska tietorakenne tulisi aina voida käydä läpi alkio alkiolta. Lisäksi adaptoituva tietorakenne voisi toteuttaa ICollection-rajapinnan, joka on yksinkertaisin tietorakennerajapinta, jonka kautta tietorakenteen sisältöä voidaan muokata. Jos adaptoituvalla tietorakenteella olisi erikoistuneita operaatioita, tulisi näitä varten luoda uusi rajapintatyyppi, joka toteuttaisi IEnumerable- ja ICollection-rajapinnat.

5 Tietorakenteiden suorituskyky perusoperaatioissa

Tässä luvussa käsitellään tutkielmassa käsiteltyjen tietorakenteiden suorituskykyä perusoperaatioiden suhteen. Tietorakenteiden toteutuksia testataan simuloiden testaussovelluksella, jossa perusoperaatioita suoritetaan tietorakenteisiin ja operaatioon käytetty aika mitataan. Tietorakenteiden toteutuksiksi on valittu C#-kielestä geneerinen List-luokka, joka on taulukkototeutus listasta; geneerinen LinkedList-luokka, joka on toteutus linkitetylle listalle; geneerinen HashSet-luokka, joka on toteutus joukolle; Hashtable-luokka, joka on toteutus hajautusrakenteelle, joka käyttää tuplahajautusta törmäysten käsittelyyn ja geneerinen Dictionary-luokka, joka on toteutus hajautusrakenteelle, joka käyttää ketjuttamista törmäysten käsittelyyn, kuten Mitchell mainitsee [11]. Lisäksi testataan itse tehtyä luokkaa binääripuulle.

Testausta varten generointiin testiaineisto etukäteen, joka tallennettiin tekstitiedostoon. Testiaineistoksi generointiin 20 merkin pituisia uniikkeja merkkijonoja, joista otettiin erikokoisia otoksia testaukseen. Testaus suoritettiin siten, että testiaineistosta valittiin otoksia sadan ja tuhannen merkkijonon väliltä sadan merkkijonon välein ja jokaisella otoksella suoritettiin sama testaus. Operaatioihin käytetyn ajan mittaamiseen käytettiin C#:n kirjastosta System.Diagnostics-nimiavaruudesta löytyvää Stopwatch-luokkaa.

Operaatioihin käytetty aika mitattiin siten, että kello laitettiin käyntiin, jonka jälkeen tietorakenteisiin lisättiin otoksen verran alkioita. Otos lisättiin tietorakenteeseen satatuhatta kertaa, jotta kokonaisajasta saataisiin järkevän kokoinen. Tämän jälkeen kello pysäytettiin. Näin saatiin kokonaisaika, jossa oli mukana myös iteraattoreiden toimintaan kulunut aika sekä uusien tietorakenneinstanssien luontiin käytetty aika. Tämän jälkeen kellolla mitattiin aika, joka kului iteraattoreiden toimintaan ja uusien tietorakenneinstanssien luomiseen. Tämä aika vähennettiin kokonaisajasta. Näin saatiin selville vain operaatioihin käytetty aika. Testiajoja suoritettiin jokaiselle operaatiolle kolme kappaletta, jotta saataisiin tietää kuinka paljon eroavaisuuksia ajojen välillä oli, ja että esimerkiksi .NET:n virtuaalikoneen roskien keruu ei vaikuttanut tuloksiin liikaa. Roskien keruun vaikutusta tuloksiin vähennettiin myös siten, että roskien keruu pakotettiin ennen iteraatioiden aloittamista ja odotettiin, kunnes roskien keruu oli valmis. Mittaamiseen käytetyn luokan lähde-

koodi löytyy liitteistä.

Jokaisesta kokonaisajasta laskettiin tämän jälkeen keskimääräinen yhteen operaatioon kulunut aika. Tämä laskettiin siten, että kokonaisaika jaettiin iteraatioiden määrällä ja otoksen koolla. Kaikkien ajojen tuloksista laskettiin tämän jälkeen keskiarvot. Koska yhteen operaatioon kulunut aika on hyvin pieni, esitetään tulokset nanosekunteina. Tulokset esitetään kuvaajina ja taulukkomuotoisina.

5.1 Suorituskyky alkion lisäämisessä

Alkion lisäämisen testauksessa tietorakenteisiin lisättiin eri kokoisia otoksia testiaineistosta. Lisäys suoritettiin käyttämällä kunkin tietorakenteen alkionlisäysmetodia. Alkioita ei lisätty tietorakenteen luomisen aikana antamalla testiaineistoa konstruktoriin, vaikka tällainen konstruktori olisikin ollut olemassa. Näin ei tehty, koska kaikilla tietorakenteilla ei ole tällaista konstruktoria olemassa. Alkiot lisättiin tietorakenteisiin yksitellen eikä esimerkiksi antamalla koko testiaineistoa lisäysmetodiin, vaikka tällainen metodi olisikin ollut olemassa. Näin tuloksista saatiin vertailukelpoisia keskenään.

Alkioiden lisäämisessä List-tietorakenteeseen käytettiin List-luokan Add-metodia, joka ottaa parametriksi yhden lisättävän alkion. Listaan lisättiin alkioita kahdella eri tavalla. Ensimmäisenä listaan lisättiin alkioita, kun se oli luotu oletuskonstruktorilla, eli sille ei oltu annettu kokoa. Tällöin lista luo itselleen nollanmittaisen taulukon alkioille. Tämän seurauksena lista joutui kasvattamaan taulukon kokoa aina, kun tila loppuu. Toiseksi listaan lisättiin alkiot siten, että se luotiin antamalla konstruktoriin testiaineiston koko. Tällöin lista luo halutunkokoisen taulukon alkioille. Tämän seurauksena listan ei tarvinnut kasvattaa taulukon kokoa.

Alkioiden lisäämisessä LinkedList-tietorakenteeseen käytettiin LinkedList-luokan AddLast-metodia, joka ottaa parametriksi yhden lisättävän alkion ja lisää sen listan loppuun. Linkitetylle listalle ei voi antaa alkukokoa, koska se on dynaaminen tietorakenne ja muisti jokaiselle alkiolle varataan sen lisäämisen yhteydessä.

Alkioiden lisäämisessä Dictionary-tietorakenteeseen käytettiin Dictionary-luokan Add-metodia, joka ottaa parametriksi avaimen lisättävälle alkiolle ja lisättävän alkion. Lisäämisessä avaimena käytettiin lisättävää alkiota. Kuten listankin tapauksessa alkioiden lisääminen tehtiin kahdella tavalla. Ensin luomalla tietorakenne oletuskonstruktorilla ja tämän jälkeen antamalla testiaineiston koko aloituskooksi. Kun aloituskokoa ei annettu, joutui sanakirja kasvattamaan sisäisen tietorakenteensa ko-

koa, kun se täyttyi. Kun aloituskoko annettiin, alusti sanakirja taulukon käyttämällä annettua kokoa, mutta täsmälleen annetun koon kokoiseksi. Tietorakenne laskee kooksi lähimmän annettua kokoa suuremman alkuluvun, jotta hajautus toimisi tehokkaammin.

Alkioiden lisäämisessä Hashtable-tietorakenteeseen käytettiin Hastable-luokan Add-metodia, joka ottaa parametriksi avaimen lisättävälle alkiolle ja lisättävän alkion. Lisäämisessä avaimena käytettiin lisättävää alkiota. Lisääminen suoritettiin samalla tavalla kuin Dictionary-luokallekin.

Alkioiden lisäämisessä HashSet-tietorakenteeseen käytettiin HashSet-luokan Add-metodia, joka ottaa parametriksi lisättävän alkion, ja palauttaa tiedon siitä, lisättiinkö alkio vai ei. Koska HashSet on joukko, ei samanarvoista alkiota voi lisätä rakenteeseen kahta kertaa. Jos tietorakenteessa on jo samanarvoinen alkio, ei alkiota lisätä. Kaikki testauksen aikana lisättävät alkiot lisättiin tietorakenteeseen, koska testiaineistossa oli vain uniikkeja merkkijonoja. HashSet-luokalla ei ole konstruktoria, jolle voisi antaa aloituskoon, joten lisäys tehtiin vain käyttämällä oletuskonstruktoria.

Alkioiden lisäämisessä BinarySearchTree-tietorakenteeseen käytettiin BinarySearchTree-luokan Add-metodia, joka ottaa parametriksi avaimen lisättävälle alkiolle ja lisättävän alkion, ja palauttaa tiedon siitä, lisättiinkö alkio vai ei. Lisäämisessä avaimena käytettiin lisättävää alkiota. Binääripuulle ei voi antaa aloituskokoa, koska se on dynaaminen tietorakenne, ja jokaiselle alkiolle varataan muisti sen lisäämisen yhteydessä.

Tuloksista voidaan huomata, että itse tehty binääripuuluokka olisi kaikista huonoin valinta tietorakenteeksi, jos lisäysoperaatioissa vaadittaisiin hyvää suorituskykyä. Tuloksia voidaan selittää sillä, että alkion lisäyksessä on aina aloitettava puun juuresta ja käytävä solmuja läpi niin kauan, että löydetään oikea paikka. Läpikäyntiä joudutaan tekemään niin kauan, että löydetään null-viite solmun siltä puolelta, jonne lisättävä alkio kuuluisi. Binääripuu-luokka ei ole itseään tasapainottava, joten puu ei ollut jokaisen operaation tasapainossa. Taulukossa 5.1 on listattu puun lopullinen korkeus jokaisen otoksen lisäämisen jälkeen. Taulukossa esitetään myös, mikä olisi puun optimaalinen korkeus kyseisen kokoisella aineistolla, eli mikä olisi täydellisesti tasapainossa olevan puun korkeus. Luvuista voidaan huomata, että puu ei ollut lisäyksien jälkeen täysin tasapainossa, mutta ei myöskään pahimman tapauksen mukainen, jossa puun rakenne olisi vastannut linkitettyä listaa.

otoksen koko	puun lopullinen korkeus	optimaalinen korkeus
100	16	7
200	18	8
300	22	9
400	22	9
500	22	9
600	25	10
700	26	10
800	26	10
900	27	10
1000	27	10

Taulukko 5.1: Binääripuun korkeudet ja optimaaliset korkeudet kunkin otoksen lisäämisen jälkeen.

.NET:n tietorakenteista listarakenteet näyttäisivät olevan suorituskykyisimpiä. List-luokka näyttäisi olevan kaikista paras vaihtoehto lisäysoperaation kannalta. Voidaan myös huomata, että aloituskoon antamisella listalle on vaikutusta tietorakenteen suorituskykyyn. Koska tilan loppumisen yhteydessä lista kaksinkertaistaa kokonsa, joudutaan listan kokoa kasvattamaan sitä harvemmin mitä suurempi määrä alkioita on. Erikoistapauksena listan kasvattamisesta on tyhjän listan kasvattaminen, jolloin esimerkiksi C#:n List-luokassa listasta tehdään neljän alkion kokoinen. Sadan alkion listan tapauksessa listaa jouduttiin kasvattamaan kuusi kertaa: 4, 8, 16, 32, 64 ja 128 alkion kokoiseksi. Tuhannen alkion listan tapauksessa jouduttiin listaa kasvattamaan vain yhdeksän kertaa, vaikka alkioita on kymmenkertainen määrä sataan verrattuna: 4, 816, 32, 64, 128, 256, 512 ja 1024 kokoiseksi. Suurimmalle osalle alkioita on lisäysoperaatio vakioaikainen, vaikka listalle ei annettaisikaan aloituskokoa. Aloituskoon antaminen kuitenkin selkeästi parantaa suorituskykyä, koska jokaisen taulukon laajentamisen yhteydessä on kaikki aikaisemmin lisätyt alkion lisättävä uuteen taulukkoon. Suurilla aineistoilla tästä operaatiosta tulee luonnollisesti kalliimpi suorituskyvyn ja myös muistinkäytön suhteen.

Linkitetyn listan lisäysoperaatio on aina vakioaikainen, mutta tulosten mukaan LinkedList-luokka on aina noin kaksi kertaa hitaampi kuin List-luokka, vaikka verrattaisiin tapauksiin, joissa listalle ei annettu aloituskokoa. Tätä eroa voidaan selittää sillä, että linkitetyn listan jokaisessa lisäysoperaatiossa varataan aina muistia listan solmulle, eikä alkiota tai viitettä siihen tallenneta peräkkäisrakenteeseen kuten tau-

lukon tapauksessa, ja päivitetään viimeisen alkion seuraava viite sekä TAIL-viite. Tämä muistin varaaminen ja viitteiden päivittäminen tekee siitä taulukkopohjaista listaa hitaamman tietorakenteen lisäysoperaatiossa.

Hajautusrakenteista Dictionary-luokka näyttäisi olevan kaikista suorituskykyisin. Tässäkin on nähtävissä ero sillä, asetettiinko aloituskoko rakenteelle vai ei. Tapauksissa, joissa aloituskokoa ei annettu, suoritusaika oli keskimäärin puolitoistakertainen verrattuna tapauksiin, joissa aloituskoko annettiin. HashSet-luokan suorituskyky on hyvin lähellä Dictionary-luokan suorituskykyä, kun aloituskokoa ei annettu. Tämä tulos on loogista, sillä HashSet-luokalle ei voi antaa aloituskokoa ja luokka käyttää samanlaista hajautus- ja törmäyksenkäsittelyalgoritmia. Suurimmat erot suorituskyvyssä on Hashtable-luokalla, jonka suoritusaika tapauksissa, joissa aloituskokoa ei annettu, on aina vähintään puolitoistakertainen, ja ajoittain yli kaksinkertainen, verrattuna tapauksiin, joissa aloituskoko annettiin. Hashtable-luokka on siis näistä hajautusrakenteista suorituskyvyltään huonoin.

Dictionary-luokalla, kun aloituskokoa ei annettu, voidaan huomata suurempia suoritusajan kasvuja neljänsadan ja viidensadan alkion välillä sekä yhdeksänsadan ja tuhannen alkion välillä. Tämä sama käyttäytyminen esiintyy myös sadan ja kahdensadan alkion välillä, mutta pienempänä. Nämä suuremman kasvut suoritusajassa johtuvat rakenteen sisäisen taulukon kasvattamisen tarpeesta. Dictionary-luokka kasvattaa sisäistä taulukkoaan, kun tila loppuu, samantapaisesti kuin List-luokka, eli kaksinkertaistamalla edellisen kapasiteetin. Tämän lisäksi kuitenkin etsitään suurempi tai yhtä suuri alkuluku, joksi koko asetetaan. Alkulukua yritetään ensin löytää HashHelpers-luokan staattisesta taulukosta, johon on listattu alkulukuja väliltä 3–7199369, mutta ei kuitenkaan kaikkia alkulukuja tältä väliltä. Jos tarvitaan suurempaa kapasiteettia, niin alkuluku etsitään laskennallisesti. Kun instanssi luodaan sen kooksi asetetaan kolme. Tämän jälkeen tilan loppumisen yhteydessä kapasiteettia kasvatetaan kaksinkertaiseksi eli kuuden alkion kokoiseksi ja etsitään seuraava alkuluku alkulukutaulukosta. Koon kasvatuksen kolmen alkion kokoisesta rakenteesta rakenteeseen, johon mahtuu tuhat alkiota, ovat seuraavat: 7, 17, 37, 89, 197, 431, 919 ja 1931. Tästä voidaan huomata, että sadan alkion kokoiseksi kasvatettaessa koon kasvattamisia joudutaan tekemään viisi kertaa; kahdestasadasta neljäänsataan kasvatuksia joudutaan tekemään kuusi kertaa; viidestäsadasta yhdeksäänsataan kasvatuksia joudutaan tekemään seitsemän kertaa; ja tuhat alkiota sisältävän rakenteen kohdalla kasvatuksia joudutaan tekemään kahdeksan kertaa. Suuremmilla otoksilla suoritusajan kasvu on suurempi, koska on kopioitava enemmän

alkioita ja täytyy tehdä enemmän uudelleenhajautuksia.

vaadittu koko	kasvatuksien määrä	lopullinen kapasiteetti
100	5	197
200	6	431
300	6	431
400	6	431
500	7	919
600	7	919
700	7	919
800	7	919
900	7	919
1000	8	1931

Taulukko 5.2: Dictionary-luokan vaatimat sisäisen taulukon kasvatukset

Tästä voidaan huomata myös, että kun aloituskokoa ei anneta ja rakenteen kasvattaminen tehdään tilan loppumisen yhteydessä, voi lopullinen kapasiteetti olla paljon enemmän kuin mitä alkioiden tallentamiseen tarvitaan. Seuraavassa taulukossa on esitetty lopullinen kapasiteetti, kun vaadittu koko annetaan rakenteelle aloituskooksi.

vaadittu koko	lopullinen kapasiteetti
100	107
200	239
300	353
400	431
500	521
600	631
700	761
800	919
900	919
1000	1103

Taulukko 5.3: Dictionary-luokan lopullinen kapasiteetti, kun aloituskoko annetaan

Hajautusrakenteiden kohdalla operaatioissa tapahtui yhteentörmäyksiä. Tällöin operaatio ei ollut aina täysin vakioaikainen. Yhteentörmäyksien määrää tarkastel-

tiin Dictionary-luokan osalta tapauksissa, joissa aloituskoko annettiin ja tapauksissa, joissa aloituskokoa ei annettu. Tämä suoritettiin lisäämällä kaikki käytetyt otokset tietorakenteeseen ja tarkastelemalla luokan *buckets*-taulukkoa, jossa on ilmoitettu ketjujen viimeiset indeksit. Tähän taulukkoon päästiin käsiksi C#:n *Reflection*-rajapintojen kautta seuraavanlaisella kutsulla:

```
Type dictionaryType = typeof(Dictionary<string, string>);
int[] buckets = (int[])dictionaryType.InvokeMember(
    "buckets",
    System.Reflection.BindingFlags.GetField
    |
    System.Reflection.BindingFlags.Instance
    |
    System.Reflection.BindingFlags.NonPublic,
    null,
    dictionaryInstance,
    null);
```

Tästä taulukosta otettiin talteen taulukon kapasiteetti ja -1-arvoisten indeksien lukumäärä, joka ilmaisee tyhjien ketjujen määrän. Näiden ja otoksien kokojen avulla voitiin laskea keskimääräiset ketjujen pituudet, kun tietorakenteeseen oli lisätty kaikki alkiot. Keskimääräisen ketjun pituus laskettiin jakamalla otoksen koko taulukon kapasiteetin ja tyhjien ketjujen lukumäärän erotuksella.

otoksen koko	taulukon kapasiteetti	tyhjien ketjujen lukumäärä	keskimääräinen ketjun pituus
100	197	121	1,31
200	431	270	1,24
300	431	219	1,42
400	431	166	1,51
500	919	534	1,30
600	919	475	1,35
700	919	428	1,43
800	919	382	1,49
900	919	345	1,57
1000	1931	1147	1,28

Taulukko 5.4: Dictionary-luokan keskimääräiset ketjujen pituudet, kun aloituskokoa ei annettu

otoksen koko	taulukon kapasiteetti	tyhjien ketjujen lukumäärä	keskimääräinen ketjun pituus
100	107	44	1,59
200	239	104	1,48
300	353	150	1,48
400	431	166	1,51
500	521	191	1,52
600	631	235	1,52
700	761	289	1,48
800	919	382	1,49
900	919	345	1,57
1000	1103	444	1,52

Taulukko 5.5: Dictionary-luokan keskimääräiset ketjujen pituudet, kun aloituskoko annettiin

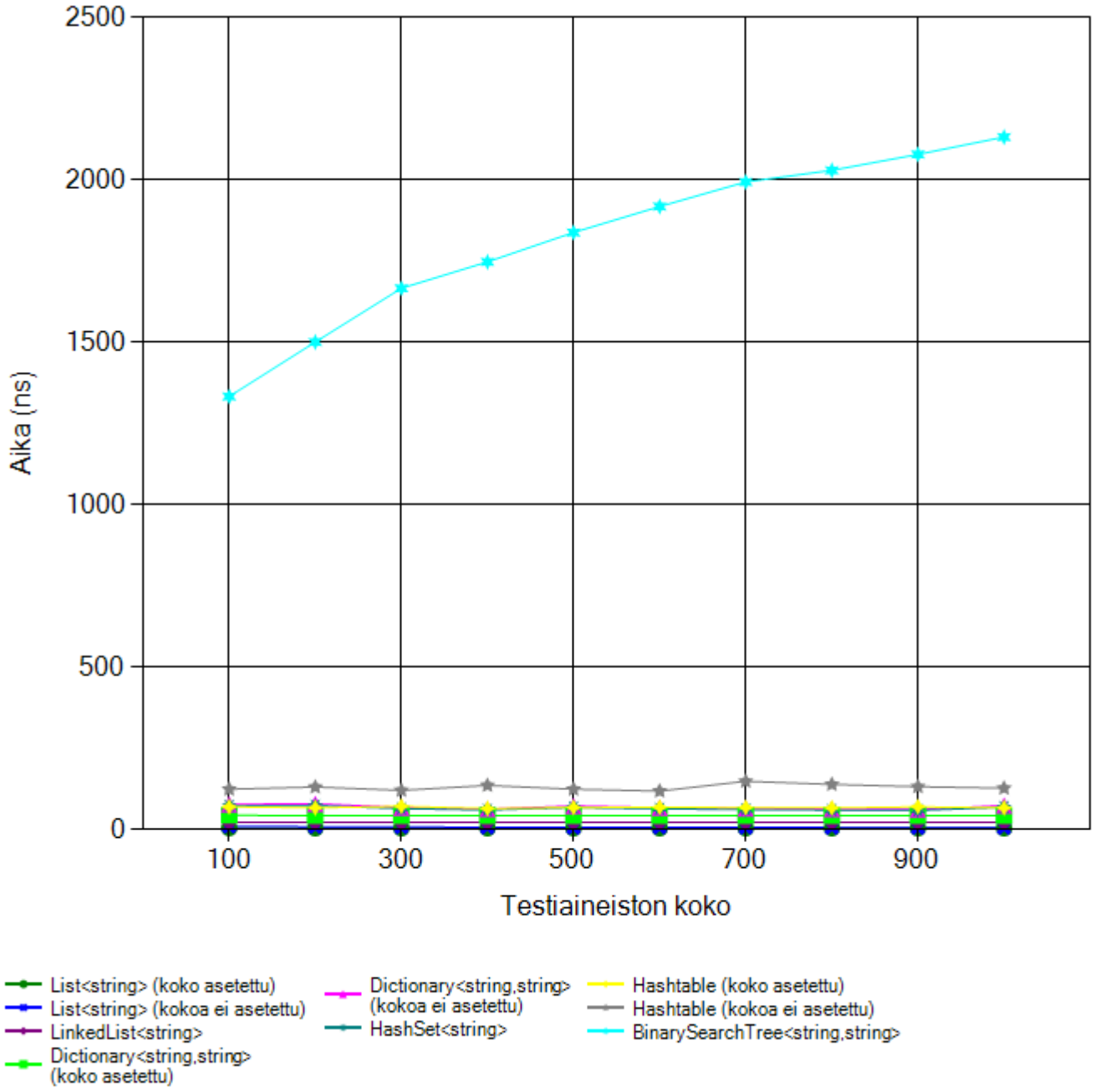
Näistä tuloksista voidaan huomata, että keskimääräiset ketjujen pituudet ovat hieman pidempiä, kun aloituskoko annettiin. Taulukkojen kapasiteetit olivat myös lähempänä otoksien kokoja, kun aloituskoko annettiin, joten taulukkoon jäi vähemmän tyhjää tilaa, jolloin yhteentörmäyksien todennäköisyys oli suurempi.

Suorituskyky näyttäisi olevan huomattavasti tasaisempi tapauksissa, joissa aloituskoko annettiin tietorakenteelle. Tämä näkyy selkeämmin hajautusrakenteilla, joilla tilankasvattaminen on työläämpi operaatio listarakenteeseen verrattuna, koska sisäisen rakenteen eli taulukon kasvattamisen lisäksi on alkioiden indeksit laskettava uudestaan. Dictionary- ja HashSet-luokkien käyttämä ketjutus-yhteentörmäyksienkäsittely näyttäisi olevan nopeampi kuin Hashtable-luokan käyttämä tuplahajautus, varsinkin rakenteen kasvattamisen yhteydessä suoritettavassa uudelleenhajautuksessa. Lisäksi Dictionary- ja HashSet-luokkien tapaukset, joissa aloituskokoa ei annettu, olivat samaa tasoa Hashtable-luokan tapauksien kanssa, joissa Hashtableilmentymälle annettiin aloituskoko. Näiden tulosten perusteella ketjutus on tuplahajautusta parempi vaihtoehto yhteentörmäysten käsittelyyn.

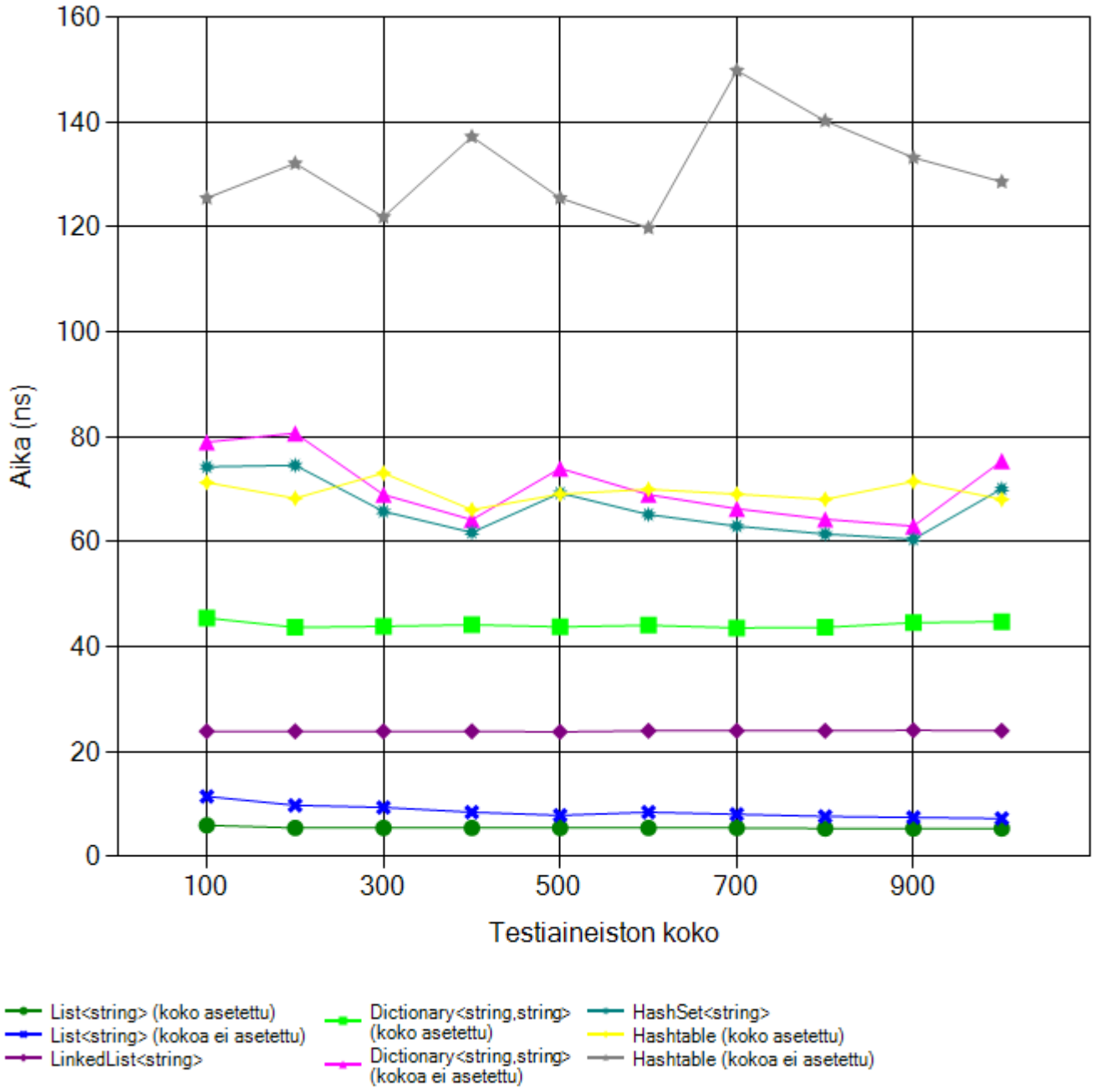
Jos otetaan huomioon, että tulosten laskemisessa testiotokset lisättiin rakenteisiin satatuhatta kertaa, ei pienillä aineistoilla ole juurikaan väliä, minkä tietorakenteen valitsee, jos suorituskyky ei ole erityisen tärkeää.

luokka	miten lisätään	kompleksisuus	erityispiirteet
List	loppuun	$O(1)$	tilan loppuessa $O(n)$
			muualle kuin loppuun $O(n)$
LinkedList	loppuun	$O(1)$	
Dictionary	hajauttamalla	$O(1)$	tilan loppuessa $O(n)$
HashSet	hajauttamalla	$O(1)$	tilan loppuessa $O(n)$
Hashtable	hajauttamalla	O(1)	tilan loppuessa $O(n)$
BinarySearchTree	kohdalleen järjestyksessä	$O(log_2(n))$	valmiiksi järjestyksessä lisäämällä $O(n)$

Taulukko 5.6: Yhteenveto lisäystavoista



Kuva 5.1: Keskimääräinen yhteen lisäysopreaatioon kulunut aika kaikilla tietorakenteilla



Kuva 5.2: Keskimääräinen yhteen lisäysopreaatioon kulunut aika .NET:n tietorakenteilla

	100	200	300	400	500	600	700	800	900	1000
List<string> (koko asetettu)	6,0	5,5	5,5	5,5	5,5	5,5	5,5	5,4	5,4	5,4
List<string> (kokoa ei asetettu)	11,5	9,8	9,4	8,5	7,9	8,5	8,1	7,7	7,5	7,3
LinkedList<string>	23,9	23,9	23,9	23,9	23,8	24,0	24,0	24,0	24,1	24,0
Dictionary<string,string> (koko asetettu)	45,5	43,7	43,9	44,2	43,8	44,1	43,6	43,7	44,6	44,8
Dictionary<string,string> (kokoa ei asetettu)	79,0	80,7	69,0	64,2	74,0	69,0	66,3	64,3	63,0	75,4
HashSet<string>	74,3	74,6	65,8	61,8	69,3	65,2	63,0	61,5	60,5	70,1
Hashtable (koko asetettu)	71,3	68,3	73,1	66,1	69,1	70,0	69,1	68,1	71,5	68,1
Hashtable (kokoa ei asetettu)	125,5	132,1	121,9	137,2	125,5	119,8	149,8	140,2	133,2	128,6
BinarySearchTree<string,string>	1333,0	1500,3	1665,7	1746,8	1837,3	1917,3	1992,7	2028,4	2077,5	2130,8

Taulukko 5.7: Keskimääräinen yhteen lisäysoperaatioon kulunut aika nanosekunteina

5.2 Suorituskyky alkion etsimisessä

Alkion etsimistä tietorakenteista testattiin siten, että aluksi niihin lisättiin testiaineisto, jonka jälkeen tietorakenteesta etsittiin jokainen siihen lisätty alkio. Mittaus suoritettiin vastaavalla tavalla kuin alkion lisäystä mitattaessa. Aika mitattiin siitä, kun jokainen alkio etsittiin tietorakenteesta ja tämä toistettiin satatuhatta kertaa. Tästä ajasta poistettiin tietorakenteiden luomiseen ja täyttämiseen käytetty aika sekä iteraattoreiden toimintaan kulunut aika. Tulokset sisältävät vain siis etsimisoperaatioon käytetyn ajan. Tietorakenteet alustettiin siten, että niille annettiin aloituskooksi testiaineiston koko, jos tällainen konstruktori oli olemassa.

Alkioiden etsimisessä List-luokasta käytettiin List-luokan Contains-metodia, joka ottaa parametriksi yhden alkion. LinkedList-luokasta käytettiin LinkedList-luokan Contains-metodia, joka ottaa parametriksi yhden alkion. Dictionary-luokasta käytettiin Dictionary-luokan ContainsKey-metodia, joka ottaa parametriksi yhden rakenteen avaimen tyyppisen alkion, jota se vertaa alkioiden avaimiin. HashSet-luokasta käytettiin Contains-metodia, joka ottaa parametriksi yhden alkion. Hashtable-luokasta käytettiin ContainsKey-metodia, joka ottaa parametriksi yhden alkion, jota se vertaa alkioiden avaimiin. BinarySearchTree-luokasta käytettiin ContainsKey-metodia, joka ottaa parametriksi yhden rakenteen avaimen tyyppisen alkion, jota se vertaa alkioiden avaimiin.

Luvussa 5.1 huomattiin, että taulukkoa tallennusrakenteena käyttävä List-luokka oli lisäysoperaatiossa ylivoimainen suorituskyvyn suhteen. Etsimisoperaatiossa asia ei kuitenkaan ole näin. List-luokka on pienemmillä otoksilla hitaimpien rakenteiden joukossa ja jo kolmensadan alkion kokoisilla otoksilla List-luokka on kaikista testattavista rakenteista hitain. Tämä johtuu siitä, että etsimisoperaatio on taulukko-

rakenteilla lineaariaikainen operaatio, koska taulukko käydään läpi ensimmäisestä alkiosta lähtien alkio alkiolta siihen asti kunnes etsittävä alkio löydetään tai kunnes on päästy viimeiseen tallennettuun alkioon. Eli kun etsittiin viimeiseksi taulukkoon lisättyä alkiota, käytiin läpi kaikki taulukkoon tallennetut alkiot.

LinkedList-luokka on samaa tasoa List-luokan kanssa etsimisoperaatiossa, joskin se näyttäisi olevan hieman nopeampi. Tämä suorituskykyero on kuitenkin mitätön, kun otetaan huomioon, että etsimisoperaatiot toistettiin satatuhatta kertaa. Linkitettyjen listojen etsimisoperaatio on myös lineaariaikainen, joten oli odotettavaa, että List- ja LinkedList-luokkien suorituskyky etsimisoperaatiossa olisi samaa luokkaa.

BinarySearchTree-luokka pienillä otoksilla on testattavista rakenteista hitain, mutta kahdensadan alkion otoksilla jo samaa luokkaa hitaimman rakenteen eli List-luokan kanssa. Tuhannen alkion otoksilla etsimisoperaatioihin käytetty aika on enää noin kolmannes List-luokan käyttämään aikaan verrattuna. Binääripuuta läpikäytäessä ei tarvitse käydä kaikkia puun alkioita läpi, koska parhaassa tapauksessa, eli kun puu on tasapainossa, jokaisen vertailun jälkeen jää mahdollisia solmuja jäljelle enää puolet edellisestä. Koska binääripuulla on tämä ominaisuus, on myös odotettavaa, että etsimisoperaatio on taulukkorakennetta tehokkaampi, kun alkioiden määrä kasvaa. Taulukossa 5.1 esitettyjen korkeuksien mukaan tuhannen alkion kokoisessa puussa tarvitsi täytyi käydä maksimissaan kaksikymmentäseitsemän puun solmua läpi alkion löytämiseksi, koska jokaisen vertailun jälkeen siirrytään puussa seuraavalle tasolle, ja puun korkeus tuhannella alkiolla oli kaksikymmentäseitsemän. Tämä on huomattavasti vähemmän kuin listarakenteilla, joilla täytyi käydä maksimissaan tuhat alkiota läpi oikean alkion löytämiseksi.

Hajautusrakenteet ovat etsimisoperaatiossa selkeästi nopeimpia. Etsimisoperaatiossa kaikki testattavat hajautusrakenteet ovat hyvin tasaväkisiä. Dictionary-luokka näyttäisi kuitenkin olevan nopein suurimmassa osassa tapauksista. Vaikka HashSet-luokka käyttää samankaltaista ketjutusalgoritmia kuin Dictionary, on se silti hieman hitaampi. HashSet- ja Dictionary-luokat eivät kuitenkaan ole täysin identtisiä sisäiseltä rakenteeltaan etsimisoperaatiota testattaessa. Dictionary-luokalle voitiin antaa aloituskoko, mutta HashSet-luokalle ei. Tämän vuoksi näiden luokkien sisäisien taulukoiden koot eroavat toisistaan, koska Dictionary-luokalle löydettiin jo instanssin luomisen aikana otoksen kokoa lähellä oleva alkuluku aloituskooksi, mutta HashSet-ilmentymä joutui kasvattamaan kokoaan lisäyksien aikana, jolloin lopulliseksi kooksi tuli eri kuin, jos koko olisi annettu sen luomisen aikana. Taulukoista 5.2

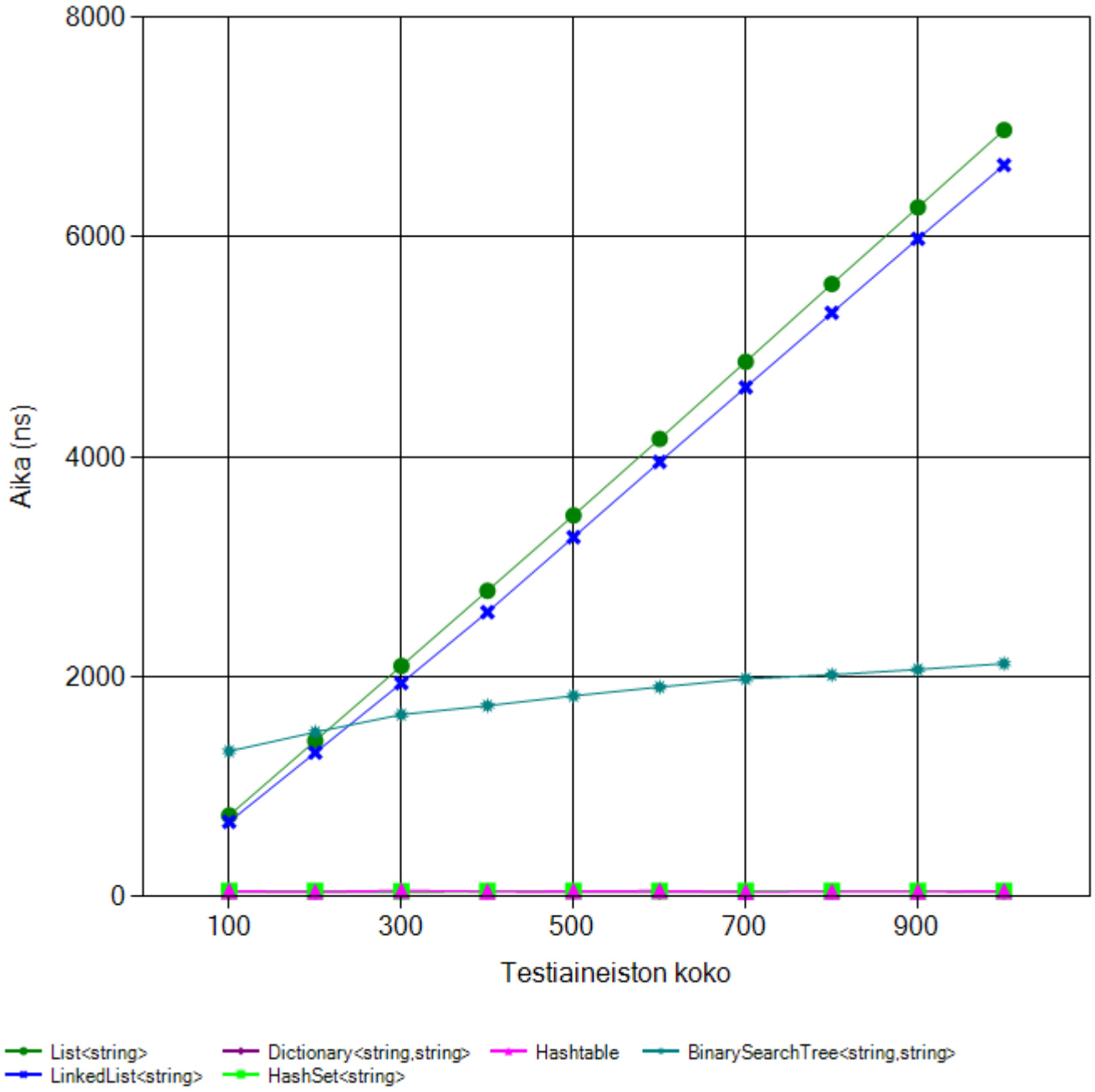
ja 5.3 voi nähdä, kuinka paljon lopulliset taulukoiden koot eroavat, kun aloituskoko annetaan tai jätetään antamatta.

Hashtable, joka on viimeinen testattavista hajautusrakenteista, on suorituskyvyltään epätasaisin. Hashtable käyttää eri hajautusalgoritmia kuin Dictionary ja HashSet, kuten luvussa 4.3 mainittiin. Hashtable-luokan käyttämään kaksoishajautukseen vaikuttaa sisäisen taulukon tilan loppuminen, koska jos alkiolle lasketun indeksin paikalla on jo alkio, lasketaan sille uusi indeksi. Siis, kun alkioiden määrä kasvaa, kasvaa myös todennäköisyys, että lasketun indeksin kohdalla on jo alkio. Tämän vuoksi Hashtable-luokka käyttääkin täyttösuhdetta hyödykseen, kun tarkastellaan, tarvitseeko sisäisen taulukon kokoa kasvattaa. Taulukkoa ei siis koskaan päästetä täysin täyteen. Tämän vuoksi Hashtable myös väistämättä hukkaa hieman muistia.

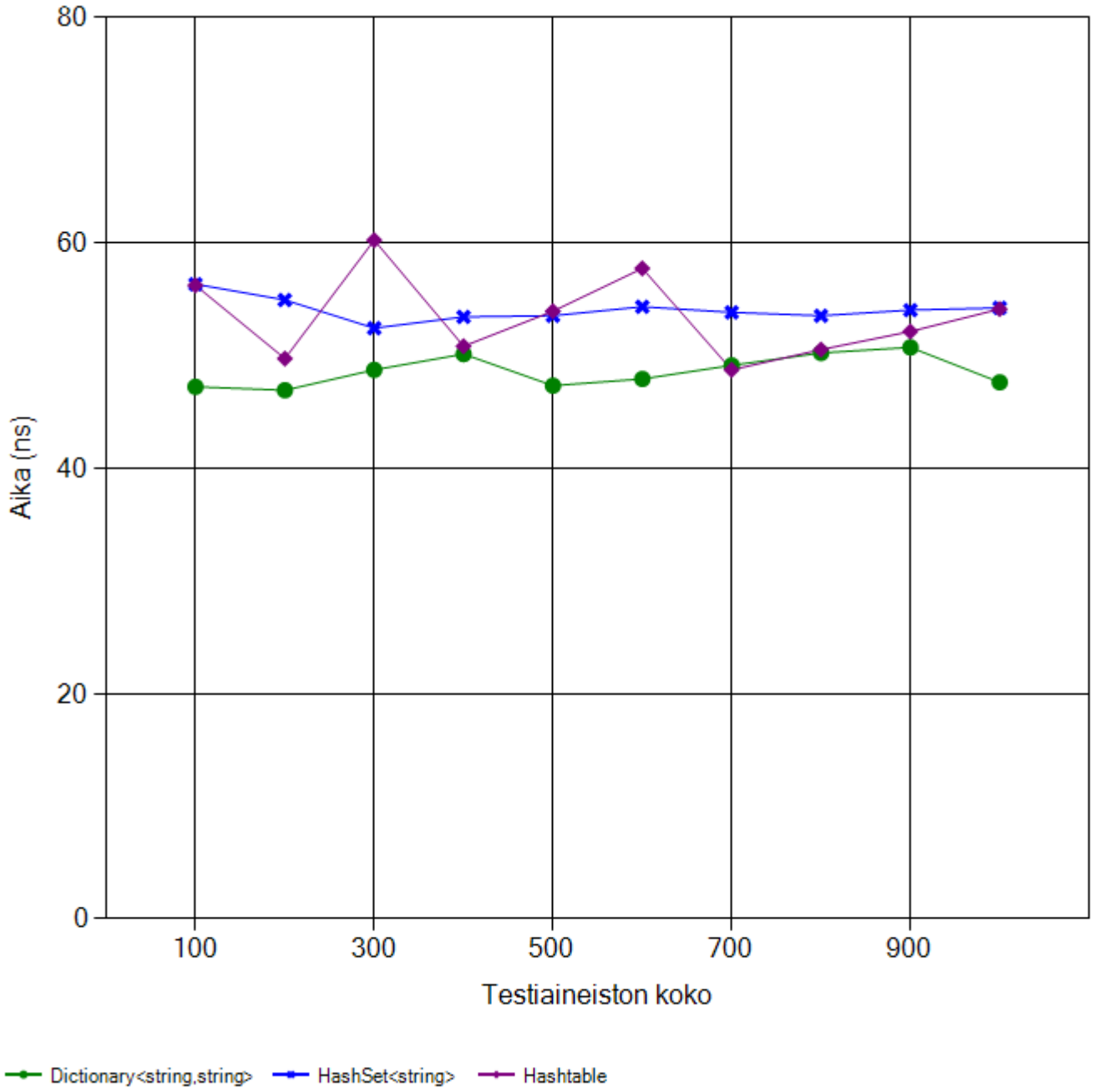
Tuloksista voidaan huomata, että hajautusrakenteiden kompleksisuus on enemmän tai vähemmän vakioaikainen. Tulokset eivät muodosta täydellistä vakiosuoraa, mutta tulokset eivät myöskään kasva tasaisesti otoksien koiden kasvaessa.

luokka	kompleksisuus
List	$O(n)$
LinkedList	$O(n)$
Dictionary	$O(1)$
HashSet	$O(1)$
Hashtable	$O(1)$
BinarySearchTree	$O(log_2(n))$

Taulukko 5.8: Yhteenveto etsimistavoista



Kuva 5.3: Keskimääräinen yhteen etsimisoperaatioon kulunut aika kaikilla tietorakenteilla



Kuva 5.4: Keskimääräinen yhteen etsimisoperaatioon kulunut aika hajautusrakenteilla

	100	200	300	400	500	600	700	800	900	1000
List<string>	741,9	1419,2	2100,7	2783,8	3468,9	4164,3	4865,9	5572,7	6269,6	6972,2
LinkedList<string>	683,2	1312,2	1944,2	2589,6	3271,1	3955,0	4633,5	5310,5	5983,6	6653,4
Dictionary<string,string>	47,2	46,9	48,7	50,1	47,3	47,9	49,1	50,2	50,7	47,6
HashSet<string>	56,3	54,9	52,4	53,4	53,5	54,3	53,8	53,5	54,0	54,2
Hashtable	56,2	49,7	60,2	50,8	53,9	57,7	48,7	50,5	52,1	54,1
BinarySearchTree<string,string>	1325,1	1498,4	1658,0	1739,3	1828,8	1908,5	1982,9	2020,1	2068,9	2121,7

Taulukko 5.9: Keskimääräinen yhteen etsimisoperaatioon kulunut aika nanosekunteina

5.3 Suorituskyky alkion poistamisessa

Alkion poistamista testattaessa toimittiin samalla tavalla kuin alkion etsimistä testattaessakin. Ensin tietorakenteet täytettiin testiotoksilla ja tämän jälkeen jokainen lisätty alkio poistettiin tietorakenteista. Aloituskoko annettiin niille tietorakenteille, joilla tällainen konstruktori oli olemassa. Poistamista testattiin kolmella eri tavalla. Ensin alkiot poistettiin samassa järjestyksessä kuin ne oli lisättykin, eli ensimmäisenä lisätty alkio poistettiin ensimmäisenä ja viimeisenä lisätty alkio poistettiin viimeisenä. Seuraavaksi testattiin tapausta, jossa alkio poistettiin lisäysjärjestystä käänteisessä järjestyksessä, eli viimeisenä lisätty alkio poistettiin ensimmäisenä ja ensimmäisenä lisätty alkio poistettiin viimeisenä. Viimeiseksi testattiin tapausta, jossa alkio poistettiin aina niin sanotusti keskeltä. Eli lista, jossa poistettavat alkiot olivat, järjestettiin siten, että testiotoksesta otettiin keskimmäinen alkio, jonka jälkeen otettiin alkio vuoroin vasemmalta ja oikealta puolelta kulkemalla listan alkua ja loppua kohden. Näin poistettava alkio oli aina niin sanotusti keskellä. Poistettava alkio oli oikeasti rakenteen keskellä vain listarakenteissa. Hajautusrakenteissa tämän ei pitäisi vaikuttaa, koska mihin tahansa alkioon päästään vakioaikaisesti. Binääripuun tapauksessa alkiot ovat rakenteessa järjestyksessä, joten poistettavan alkion paikkaa ei etukäteen tiedetty. Metodi, jolla lisäysjärjestyksestä saatiin järjestys, jossa alkiot tultiin poistamaan keskeltä on seuraavanlainen:

```
1 private List<string> OrderFromCenter(List<string> list)
2 {
3     int center = list.Count / 2;
4     int index = center;
5     int counter = 1;
6     int twoCounter = 0;
7     int osc = 1;
8     List<string> ordered = new List<string>(list.Count);
```

```
9       while (ordered.Count < list.Count)
10      {
11          if(index >= 0 && index < list.Count)
12              ordered.Add(list[index]);
13          index = center + (osc * counter);
14
15          osc = osc * -1;
16          ++twoCounter;
17          if (twoCounter == 2)
18          {
19              ++counter;
20              twoCounter = 0;
21          }
22
23      }
24      return ordered;
25  }
```

Poisto-operaatioon käytettiin List-luokasta Remove-metodia, joka ottaa parametriksi poistettavan alkion. LinkedList-luokasta käytettiin Remove-metodia, joka ottaa parametriksi poistettavan alkion. Dictionary-luokasta käytettiin Remove-metodia, joka ottaa parametriksi poistettavaa alkiota vastaavan avaimen. HashSet-luokasta käytettiin Remove-metodia, joka ottaa parametriksi poistettavan alkion. Hashtable-luokasta käytettiin Remove-metodia, joka ottaa parametriksi poistettavaa alkiota vastaavan avaimen. BinarySearchTree-luokasta käytettiin Remove-metodia, joka ottaa parametriksi poistettavaa alkiota vastaavan avaimen.

Poisto-operaatiossa binääripuu-luokka on hitain, kun alkiot poistettiin lisäysjärjestyksessä. Poisto-operaatio onkin binääripuulle operaatioista monimutkaisin. Jos poistettava alkio ei ole lehtisolmu eli, jos solmulla on lapsisolmuja, pitää poisto-operaation lisäksi päivittää useita viitteitä tai siirtää solmuja. Jos poistettava solmu on lehtisolmu, täytyy vain poistaa solmun vanhempisolmun viite solmuun. Jos poistettavalla solmulla on yksi lapsisolmu täytyy tämä solmu asettaa poistettavan solmun tilalle. Jos poistettavalla solmulla on kaksi lapsisolmua, on otettava joko vasemman alipuun oikeanpuoleisin solmu tai oikean alipuun vasemmanpuoleisin solmu, ja siirrettävä tämä poistettavan solmun tilalle.

Kun alkiot poistettiin käänteisessä järjestyksessä, binääripuu-luokka on hitain sadan ja kahdensadan kokoisilla otoksilla. Kolmensadan kokoisilla otoksilla suoritusaika on samaa luokkaa listarakenteiden kanssa, ja suoritusaika kasvaa sitä hi-

taammin, mitä suuremmaksi otokset tulevat. Kun alkiot poistettiin keskeltä, binääripuu-luokka on hitain neljänsadan alkion kokoisiin otoksiin asti.

Listarakenteet suoriutuivat poisto-operaatiosta samankaltaisesti tapauksissa, joissa alkiot poistettiin käänteisessä järjestyksessä, ja kun alkiot poistettiin keskeltä. Suorituskyky kuitenkin eroaa List- ja LinkedList-luokilla huomattavasti, kun alkiot poistettiin lisäysjärjestyksessä. Tässä tapauksessa poistettava alkio oli aina listan ensimmäinen alkio. Alkio siis löydettiin aina vakioaikaisesti. Kuitenkin List-luokka näyttäisi toimivan tässä silti lineaariaikaisesti ja LinkedList-luokka vakioaikaisesti. Tämä ero johtuu siitä, että linkitetyllä listalla tarvitsi vain päivittää viite ensimmäiseen alkioon, kun taas taulukkoa käyttävän listan tapauksessa kaikkia seuraavia alkiota siirrettiin yhden askeleen taaksepäin listassa, koska listaan ei saa jäädä tyhjiä paikkoja. Tämän vuoksi poisto-operaatio on taulukkoa käyttävällä listalla aina lineaariaikainen. Esimerkiksi jos poistettava alkio olisi k:nnes alkio ja listassa olisi n alkiota, niin ensin tarvittaisiin k operaatiota alkion löytämiseen ja tämän jälkeen täytyy vielä siirtää $n - k$ kappaletta alkioita listassa taaksepäin. Jos taas poistettava alkio on listan viimeinen alkio, tarvitaan n kappaletta operaatioita alkion löytämiseen.

Nyt jos vertaillaan tuloksia, joissa alkiot poistettiin lisäysjärjestyksessä, tuloksiin, joissa alkiot poistettiin käänteisessä järjestyksessä, voidaan huomata List-luokan suoritusajoissa selkeitä eroja. Sadan alkion otoksilla poisto-operaatioihin lisäysjärjestyksessä kului lähes viisi kertaa niin kauan kuin käänteisessä järjestyksessä. Tuhannen alkion otoksilla ero on jo noin kaksikymmentäkertainen. Vaikka molemmissa tapauksissa operaatio on lineaariaikainen, niin listan alkioiden siirtäminen askeleen taaksepäin on silti huomattavasti nopeampi toimenpide kuin kaikkien alkioiden läpikäynti yksitellen oikean alkion löytämiseksi. Kun alkiot poistettiin niin sanotusti keskeltä List-luokan suoritusajat vastasivat suurin piirtein ääripäiden keskiarvoja.

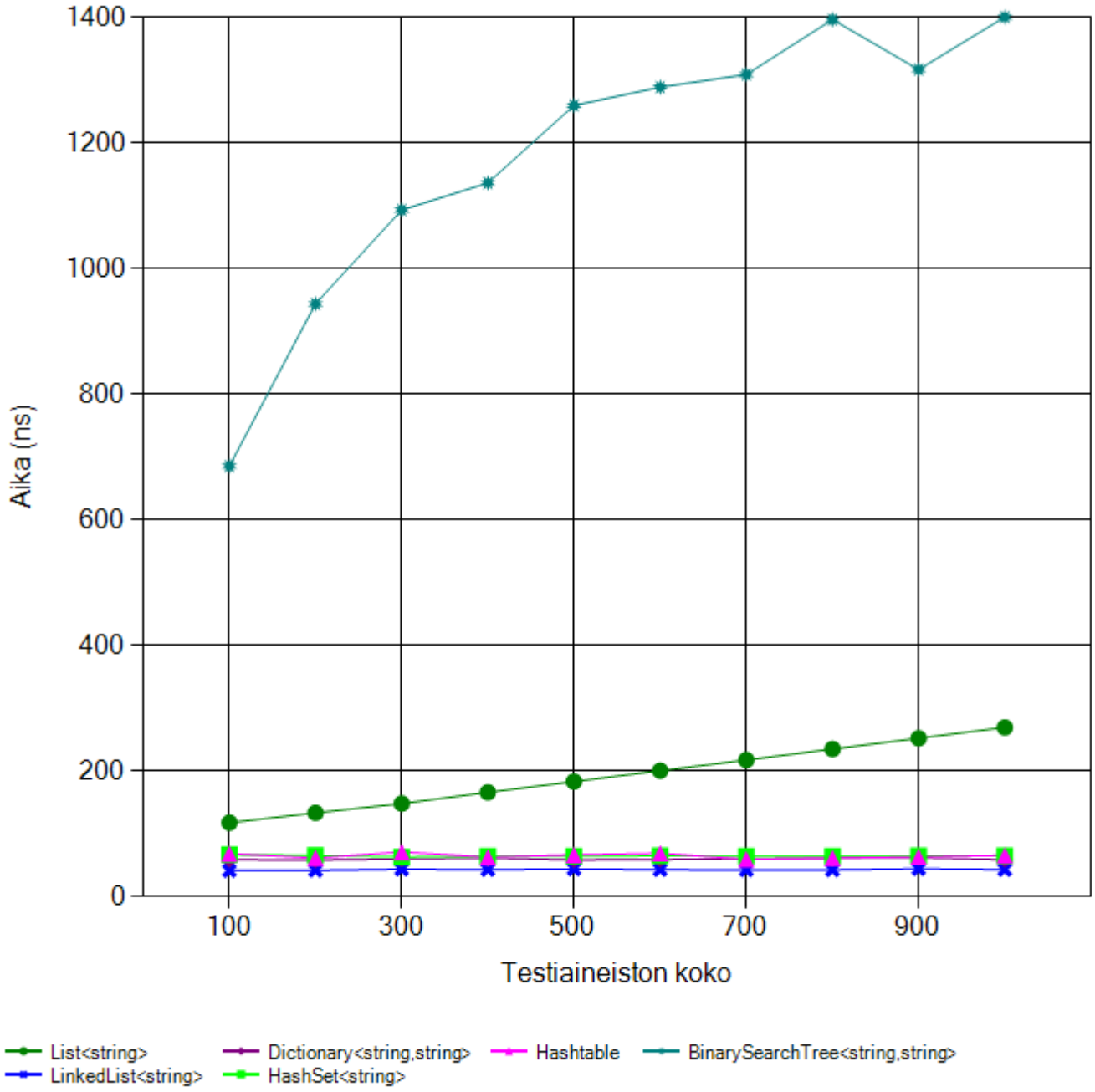
Suurin ero suoritusajoissa eri tapausten välillä on linkitetyllä listalla. Kun alkiot poistettiin lisäysjärjestyksessä, linkitetty lista oli kaikista testattavista rakenteista nopein, ja muissa tapauksissa kaikista hitain rakenne suuremmilla otoksilla. Sadan alkion otoksilla ero on noin kuusitoistakertainen, ja tuhannen alkion otoksilla ero on yli sataviisikymmentäkertainen.

Hajautusrakenteet ovat suurimmassa osassa tapauksista nopeimpia rakenteita, poislukien tapaus, jossa aina poistettiin ensimmäisenä lisätty alkio, jolloin linkitetty lista oli nopeampi. Poistojärjestys ei juurikaan vaikuttanut hajautusrakenteiden

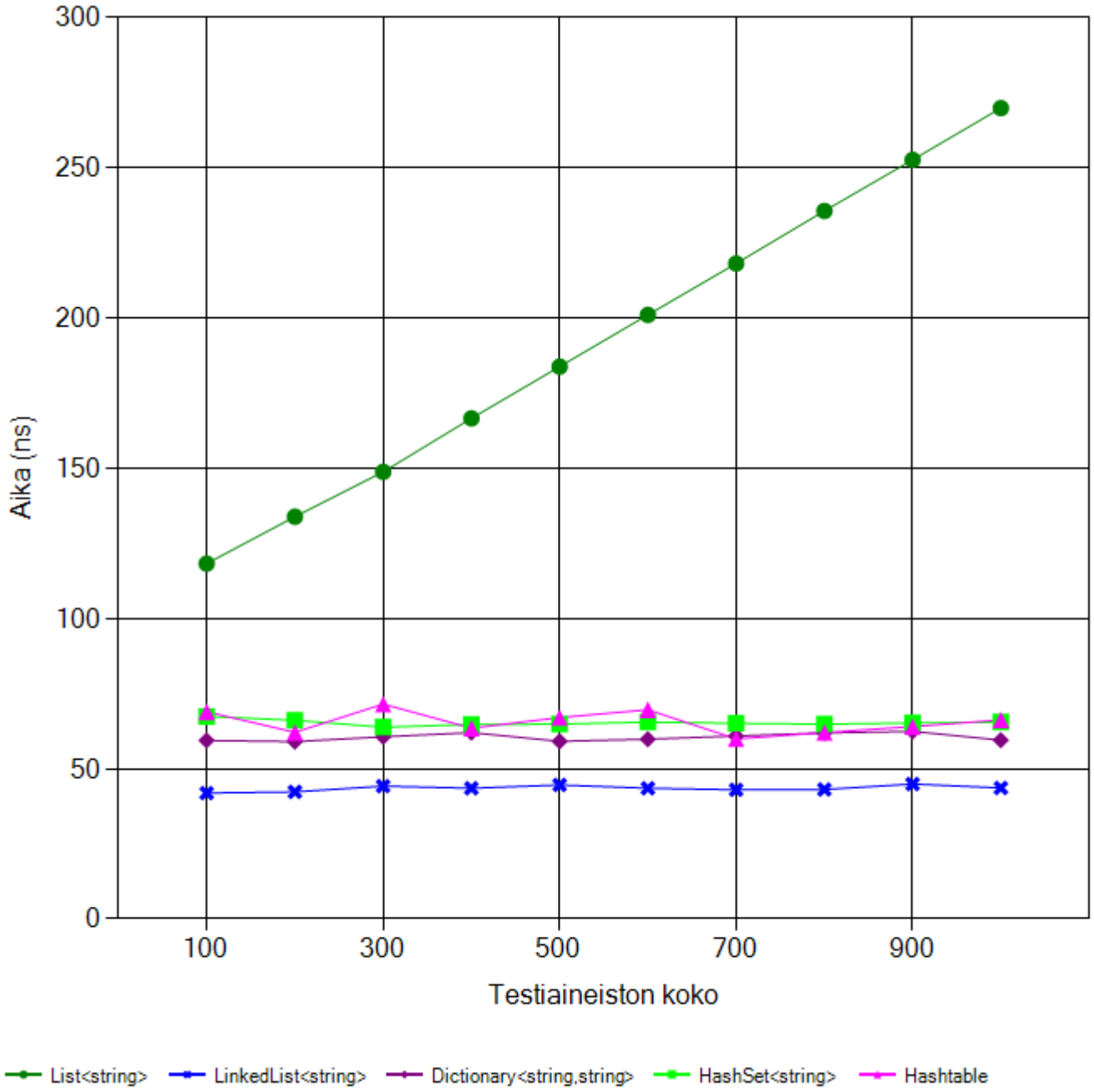
toimintaan. Pieniä eroja voidaan kuitenkin huomata Dictionary- ja HashSet-luokkien suoritusajoissa. Kun alkiot poistettiin lisäysjärjestyksessä näiden luokkien suoritusajat olivat hieman korkeampia kuin muissa tapauksissa. Käänteisessä järjestyksessä poistaessa suoritusajat ovat pienimpiä, ja keskeltä poistaessa suoritusajat ovat näiden väliltä. Tälle käyttäytymiselle on järkevä selitys. Molemmat luokat käyttävät samankaltaista ketjutusalgoritmia, jossa yhteentörmäyksien tapahtuessa viimeisimmäksi lisätyn alkion indeksi asetetaan indeksitaulukkoon ja next-arvoksi asetetaan ennen tätä lisätyn alkion indeksi. Uusi alkio asetetaankin niin sanotusti aina ketjun alkuun. Tämän vuoksi alkiot löytyivät nopeammin, kun alkiot poistettiin käänteisessä järjestyksessä, koska viimeisimmäksi lisätty alkio löydetään aina ketjun alusta. Vastaavasti lisäysjärjestyksessä poistaessa poistamiseen menee kauemmin, koska ensimmäisenä lisätty alkio on aina ketjun viimeinen alkio. Hashtable-luokalla ei näin suuria eroja ole eri tapausten välillä. Kaikki hajautusrakenteet ovat tulosten mukaan vakioaikaisia poisto-operaatiossa kaikissa kolmessa tapauksessa. Hashtable on suorituskyvyltään ailahtelevin, mutta suorituskykyä voidaan silti kutsua vakioaikaiseksi.

luokka	kompleksisuus
List	$O(n)$
LinkedList	$O(n)$
Dictionary	$O(1)$
HashSet	$O(1)$
Hashtable	$O(1)$
BinarySearchTree	$O(log_2(n))$

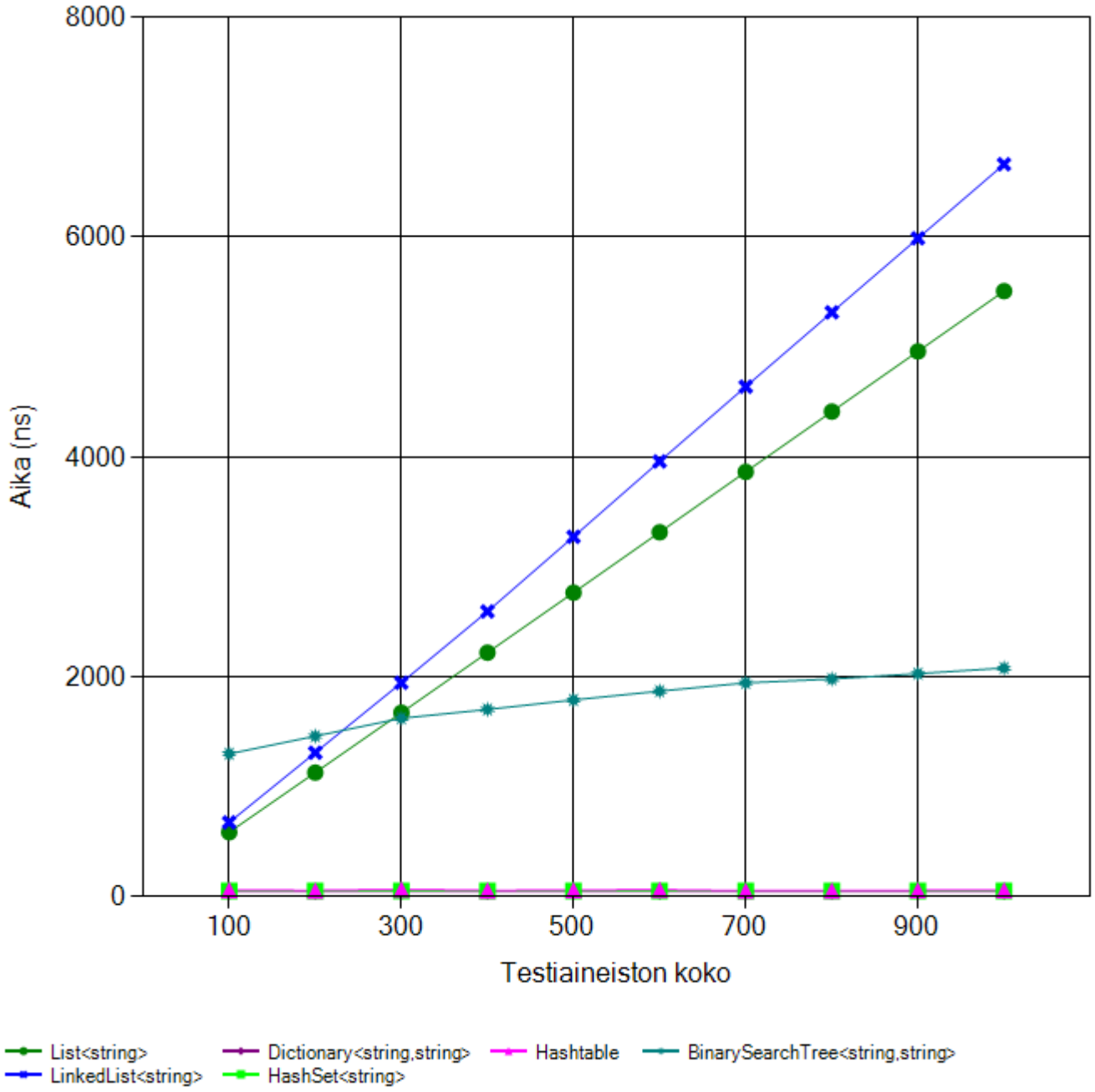
Taulukko 5.10: Yhteenveto poistotavoista



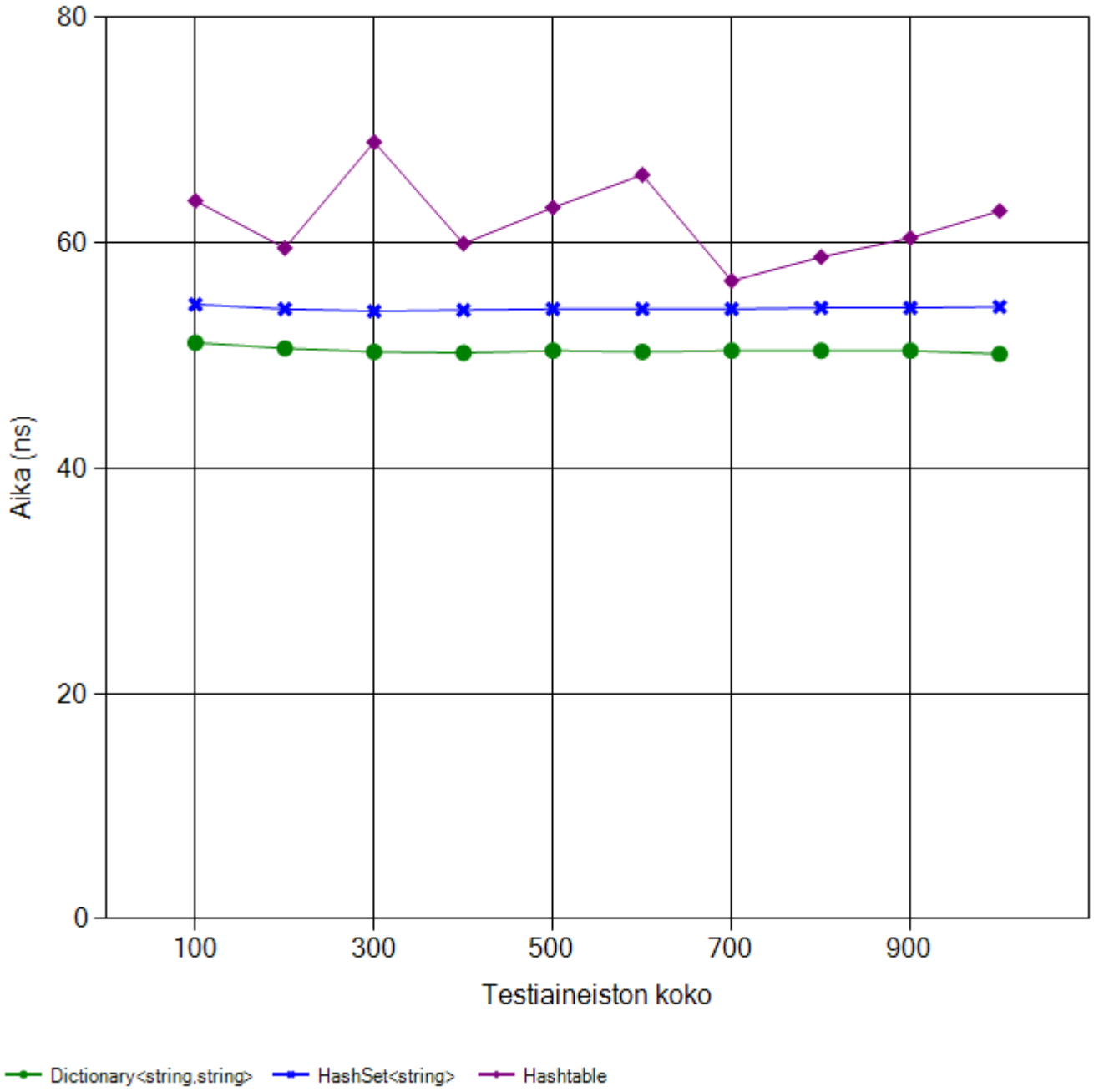
Kuva 5.5: Keskimääräinen yhteen poisto-operaatioon kulunut aika kaikilla tietorakenteilla, kun alkiot poistettiin lisäysjärjestyksessä



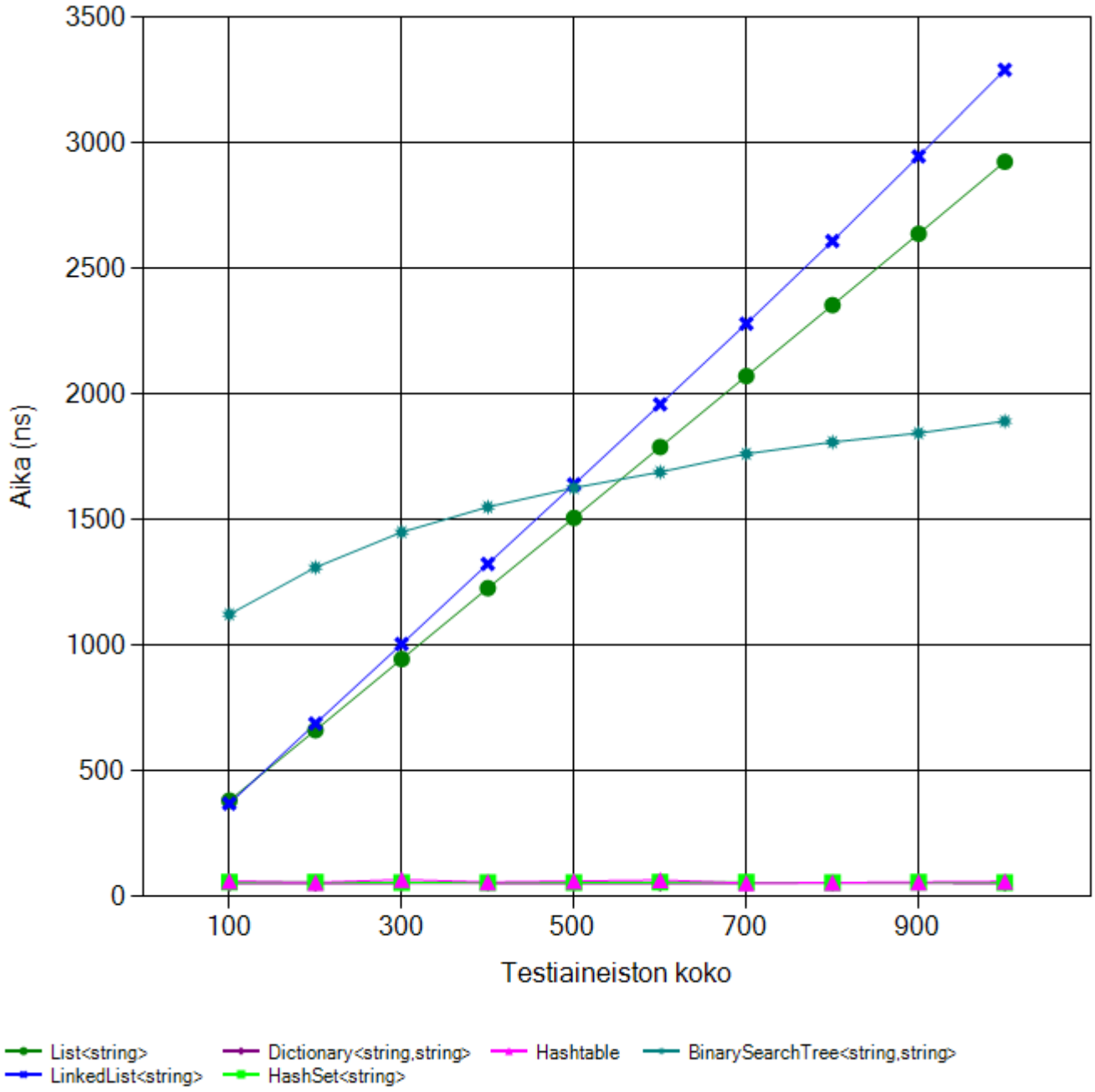
Kuva 5.6: Keskimääräinen yhteen poisto-operaatioon kulunut aika .NET:n tietorakenteilla, kun alkiot poistettiin lisäysjärjestyksessä



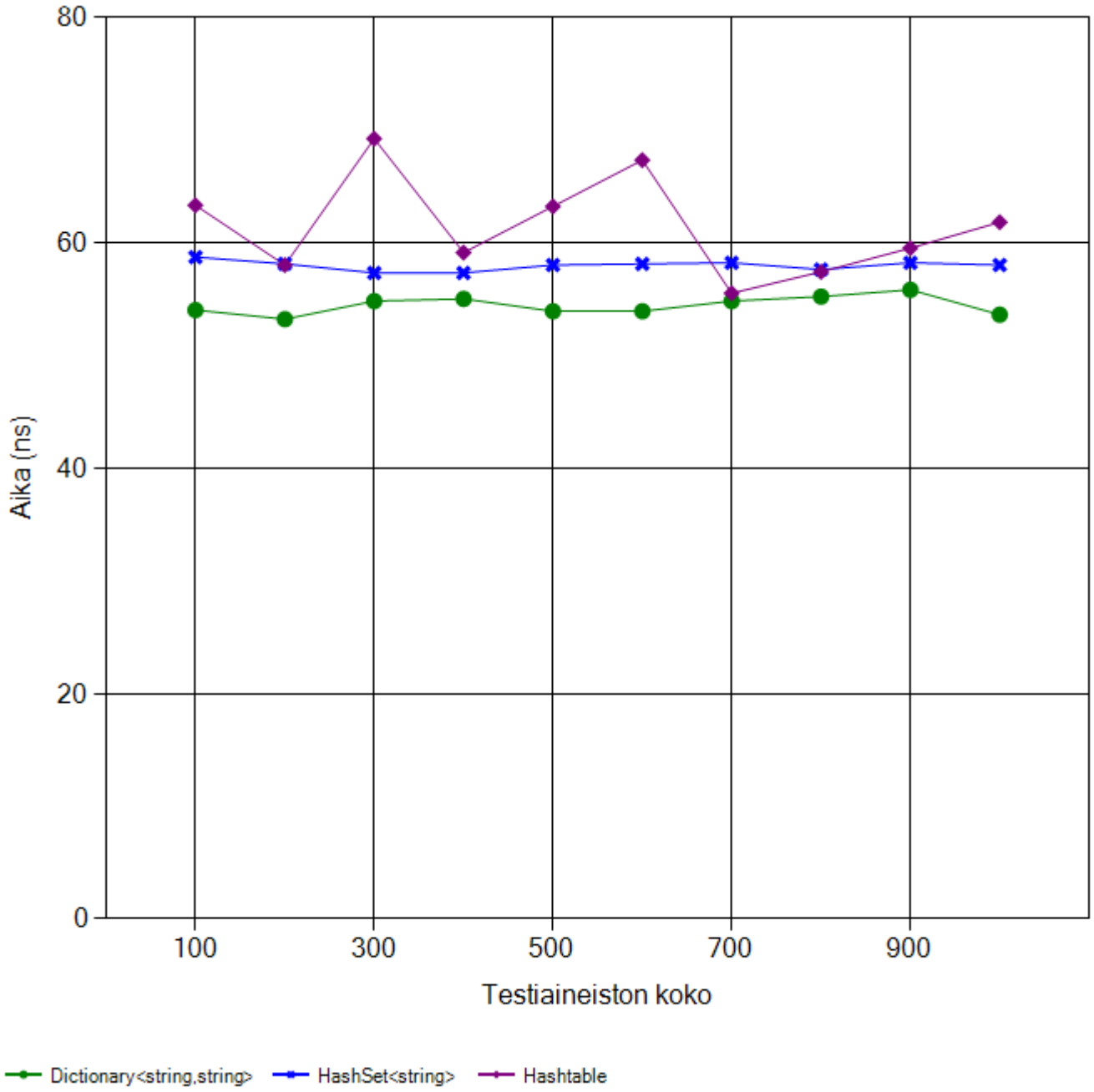
Kuva 5.7: Keskimääräinen yhteen poisto-operaatioon kulunut aika kaikilla tietorakenteilla, kun alkiot poistettiin käänteisessä järjestyksessä



Kuva 5.8: Keskimääräinen yhteen poisto-operaatioon kulunut aika hajautusrakenteilla, kun alkiot poistettiin käänteisessä järjestyksessä



Kuva 5.9: Keskimääräinen yhteen poisto-operaatioon kulunut aika kaikilla tietorakenteilla, kun alkiot poistettiin keskeltä



Kuva 5.10: Keskimääräinen yhteen poisto-operaatioon kulunut aika hajautusrakenteilla, kun alkiot poistettiin keskeltä

	100	200	300	400	500	600	700	800	900	1000
List<string>	118,2	133,8	148,7	166,5	183,7	200,9	218,0	235,5	252,5	269,7
LinkedList<string>	41,9	42,3	44,2	43,5	44,6	43,5	43,0	43,1	44,9	43,6
Dictionary<string,string>	59,4	59,0	60,6	62,0	59,1	59,8	60,9	61,9	62,4	59,5
HashSet<string>	67,4	66,1	63,8	64,7	64,8	65,5	65,1	64,8	65,2	65,5
Hashtable	68,9	62,1	71,5	63,5	67,0	69,6	59,9	62,0	64,0	66,2
BinarySearchTree<string,string>	685,6	943,6	1093,0	1135,9	1259,4	1288,3	1308,5	1395,8	1316,4	1399,9

Taulukko 5.11: Keskimääräinen yhteen poisto-operaatioon kulunut aika nanosekunteina, kun alkiot poistettiin lisäysjärjestyksessä

	100	200	300	400	500	600	700	800	900	1000
List<string>	589,1	1131,1	1675,8	2221,5	2766,9	3316,1	3865,1	4412,8	4960,5	5507,8
LinkedList<string>	678,2	1312,0	1947,5	2596,0	3274,9	3959,9	4639,4	5314,8	5988,1	6661,4
Dictionary<string,string>	51,1	50,6	50,3	50,2	50,4	50,3	50,4	50,4	50,4	50,1
HashSet<string>	54,5	54,1	53,9	54,0	54,1	54,1	54,1	54,2	54,2	54,3
Hashtable	63,7	59,5	68,9	59,9	63,1	66,0	56,6	58,7	60,4	62,8
BinarySearchTree<string,string>	1302,0	1462,5	1625,0	1705,0	1791,5	1871,8	1946,2	1981,6	2030,4	2081,5

Taulukko 5.12: Keskimääräinen yhteen poisto-operaatioon kulunut aika nanosekunteina, kun alkiot poistettiin käänteisessä järjestyksessä

	100	200	300	400	500	600	700	800	900	1000
List<string>	383,5	663,3	946,1	1228,2	1507,7	1789,7	2071,6	2353,9	2638,1	2924,3
LinkedList<string>	371,5	689,8	1006,1	1324,8	1641,1	1959,3	2280,0	2608,9	2945,8	3289,8
Dictionary<string,string>	54,0	53,2	54,8	55,0	53,9	53,9	54,8	55,2	55,8	53,6
HashSet<string>	58,7	58,1	57,3	57,3	58,0	58,1	58,2	57,6	58,2	58,0
Hashtable	63,3	58,0	69,2	59,1	63,2	67,3	55,5	57,4	59,5	61,8
BinarySearchTree<string,string>	1124,5	1311,0	1451,3	1551,0	1627,0	1690,0	1762,6	1809,3	1845,0	1892,5

Taulukko 5.13: Keskimääräinen yhteen poisto-operaatioon kulunut aika nanosekunteina, kun alkiot poistettiin keskeltä

5.4 Onko jo olemassa parasta tietorakennetta?

Luvuissa 5.1, 5.2 ja 5.3 esitettyjen mittaustulosten perusteella ei voida yksiselitteisesti valita yhtä tai yhdenlaista tietorakennetta, joka olisi muita parempi kaikissa tilanteissa. Näiden tulosten perusteella adaptoituvalle tietorakenteelle voisi olla tarvetta. Lisäysoperaatiossa listarakenteet olivat kaikista tehokkaimpia, ja listarakenteista taulukkoa käyttävä List-luokka oli kaikista tehokkain. List-luokka oli LinkedList-luokkaa tehokkaampi myös tapauksissa, joissa aloituskokoa ei annettu, ja lista joutui kasvattamaan sisäisen taulukkonsa kokoa lisäyksien välissä tilan loppumisen yhteydessä. Taulukon kasvatus -operaatioita ei kuitenkaan tapahtunut alkioi-

den määrään suhteutettuna paljon, jos alkioiden määrä oli riittävän suuri, koska taulukon tila kaksinkertaistettiin joka kasvatusoperaation yhteydessä. Eli mitä suurempi alkioiden määrä oli, sitä harvemmin taulukkoa tarvitsi kasvattaa. Kasvattaminen saattaa kuitenkin jättää listaan paljon käyttämätöntä muistia. Esimerkiksi, jos listaan tarvitsisi lisätä enää yksi alkio ennen taulukon kasvattamista, olisi lähes puolet taulukosta käyttämätöntä tilaa. Tätä varten List-luokalla onkin luvussa 4.1 mainittu TrimExcess-metodi, joka vapauttaa käyttämättömän muistin kopioimalla taulukon alkiot pienempään taulukkoon.

LinkedList-luokka oli suurimmassa osassa tapauksia List-luokkaa hitaampi. LinkedList oli ainoastaan alkion etsimisessä ja alkion poistamisessa listan alusta List-luokkaa nopeampi. Linkitetyn listan ei tarvinnut lisäysoperaatioin aikana koskea aiemmin lisättyihin alkioihin, koska jokaiselle lisättävälle alkiolle varattiin muistia erikseen, eikä aiemmin lisättyjä alkioita tarvinnut koskaan kopioida muualle tilan loppumisen yhteydessä. Linkitetty lista käyttää taulukkoa enemmän muistia, koska jokaista alkiota varten luokaan solmu-olio, josta on viite listassa seuraavaan solmuun. Lisäksi listassa on viitteet ensimmäiseen ja viimeiseen alkioon. Jos käyttämätön muisti vapautetaan List-luokassa TrimExcess-metodilla, on List-luokka LinkedList-luokkaa tehokkaampi muistinkäytön suhteen, mutta huolimattomasti käytettynä aiheuttaa enemmän töitä virtuaalikoneen roskienkeruulle taulukon kasvattamisen yhteydessä.

Binääripuu-luokka oli lisäysoperaatiossa kaikista hitain. Se oli myös hitain muissa operaatioissa pienillä otoksilla, mutta suuremmilla otoksilla logaritmisen kompleksisuutensa ansiosta se oli listarakenteita nopeampi. Lisäksi binääripuu käyttää linkitettyä listaa enemmän muistia, koska jokaiselle alkiolle luodaan solmu-olio, josta on viitteet sen oikeaan ja vasempaan lapsisolmuun. Lisäksi BinarySearchTree-luokan tapauksessa solmuissa oli myös vanhempiviite solmun vanhempisolmuun.

Taulukkoa käyttävä lista voitaisiin myös saada logaritmiseksi alkion etsimisessä, jos taulukon alkiot pidettäisiin järjestyksessä. Tämä tarkoittaisi kuitenkin sitä, että alkion lisäyksestä tulisi samalla pahimmassa tapauksessa lineaariaikainen operaatio. Ensin täytyisi löytää oikea paikka alkiolle. Tämä voitaisiin tehdä logaritmisessa ajassa käyttämällä puolitushakua. Tämän jälkeen täytyisi kuitenkin siirtää järjestyksessä seuraavia alkioita yhden askeleen eteenpäin, jotta alkio voitaisiin lisätä löydettyyn paikkaan. Jos löydetty paikka olisi listan ensimmäinen, jouduttaisiin kaikkia alkioita siirtämään yhden askeleen eteenpäin. Parhaimmassa tapauksessa lisäysoperaatio olisi logaritminen, jos alkion paikaksi tulisi listan viimeinen ja va-

paata tilaa olisi vielä taulukossa.

Hajautusrakenteet olivat kaikissa operaatioissa vakioaikaisia. Ne olivat silti lisäyksessä listarakenteita hitaampia, koska hajautusavainten laskemiseen ja yhteentörmäysten käsittelyyn kuluu aikaa. Muissa operaatioissa hajautusrakenteet olivat selkeästi nopeimpia rakenteita, pois lukien tapaus, jossa alkiot poistettiin lisäysjärjestyksessä, jossa linkitetty lista oli nopeampi. Hashtable-luokka oli muita hajautusrakenteita hitaampi ja suorituskyvyltään ailahtelevampi. Hashtable ei myöskään ole tyyppiturvallinen, koska avainten ja arvojen tyyppi on aina object. Avaimiksi ja arvoiksi kelpaavat siis minkä tahansa tyyppiset alkiot. Jos tämänkaltaista toiminnallisuutta ei tarvita, kannattaa valita Dictionary-luokka Hashtable-luokan sijaan. Dictionary-luokka on myös muistinkäytön suhteen tehokkaampi, koska sen sisäisen taulukon täyttyminen ei vaikuta hajautukseen niin paljon kuin Hashtable-luokalla. Hashtable-luokka hukkaakin muistia sen vuoksi, että se laajentaa sisäistä taulukkoaan, kun sen täyttösuhde ylittyy. Dictionaryn taulukko voidaan täyttää kokonaan ennen kuin sitä tarvitsee kasvattaa.

Jos rakenteelta tarvitaan joukon ominaisuus, eli rakenteeseen ei saa tallentaa saman arvoista alkiota kuin kerran, on HashSet-luokka hyvä vaihtoehto. Hajautusavaimen laskemiseen ja yhteentörmäysten käsittelyyn kuluu vähemmän aikaa kuin kaikkien listan alkioiden läpikäynti, kun tarkastellaan, voiko alkiota lisätä joukkoon, kun alkioita on listassa riittävän monta. Jos listassa olisi vain muutama alkio, saattaisi lista olla nopeampi, mutta alkioiden määrän lisääntyessä alkioiden läpikäyntiin tarvittava aika kasvaisi lineaarisesti. Kun taas HashSet-luokan tapauksessa alkion löytämiseen tarvittava aika on aina riippumaton alkioiden määrästä.

Yleisesti kaikkiin tapauksiin parasta tietorakennetta ei siis löydetty, mutta kaikille operaatioille löydettiin hyvä tietorakenne. Tietorakenteen hyvyys riippuukin hyvin käyttötarpeesta ja siitä, mikä missäkin tapauksessa on tärkeää. Jos käyttöprofiili on sellainen, että alkioita lisätään rakenteeseen paljon ja lisäyksistä täytyy suoriutua nopeasti, on taulukkoa käyttävä lista paras vaihtoehto. Taulukkoa käyttävä lista on myös hyvä vaihtoehto, jos muistinkäyttöön pitää kiinnittää huomiota. Jos taas rakenteesta pitää löytää tiettyjä alkioita nopeasti, listarakenteet eivät ole hyvä vaihtoehto, koska etsimisoperaatio on listoilla lineaariaikainen. Lisäksi jos rakenteeseen kohdistetaan paljon poisto-operaatioita, eivät listarakenteet ole hyvä vaihtoehto, koska poisto-operaatio on lineaariaikainen. Jos poisto-operaatiot sen sijaan kohdistuisivat aina listan ensimmäiseen alkion olisi linkitetty lista paras vaihtoehto.

Hajautusrakenteet ovat erinomaisia, jos alkiot täytyy löytää nopeasti, koska ra-

kenteeseen tallennettujen alkioiden määrä ei vaikuta operaatioiden suoritusaikaan. Hajautusrakenteet ovat kuitenkin lisäysoperaatiossa hieman hitaampia kuin tavalliset listat. Ne tarvitsevat myös hieman enemmän muistia joko täyttösuhteen vuoksi tai erillisen indeksitaulukon vuoksi. Ne eivät ole siis muistinkäytön suhteen tehokkaimpia rakenteita. Lisäksi, jos hajautusrakenteesta poistetaan paljon alkioita, jää niiden sisäisten taulukoiden koko kuitenkin ennalleen. Ja koska testattavilla luokilla ei ollut List-luokan TrimExcess-metodia vastaavaa toiminnallisuutta, ne voivat hukata muistia käyttämättömän tilan vuoksi. Käyttämätöntä tilaa voi jäädä myös, jos rakenteille ei anneta aloituskokoa, kuten taulukoista 5.2 ja 5.3 voidaan nähdä.

Jos taas on tärkeää, että rakenteeseen tallennetut alkiot ovat aina järjestyksessä, eivät listat tai hajautusrakenteet tarjoa tähän hyvää vaihtoehtoa. Tällöin vaihtoehdoksi jäävät järjestetyt rakenteet kuten binääripuut. Binääripuuhun lisättäessä alkio menee aina järjestyksessä oikealle paikalleen. Alkioiden lisääminen on kuitenkin hidasta, koska lisättävän alkion tai sen avaimen järjestystä jo tallennettuihin täytyy verrata, ja tämän vertailun perusteella valita joko vasen tai oikea alipuu, johon jatkaa. Lisäysoperaatio, kuten muutkin operaatiot, on siis aina parhaassa tapauksessa, eli jos puu on täydellisesti tasapainossa, logaritminen ja pahimmassa tapauksessa lineaarinen.

Useissa ohjelmointikielissä kuten C#:ssa on luokka kirjastoja, joista löytyy monenlaisia tietorakenteita eri tarkoituksiin, koska yleisesti parasta tietorakennetta kaikkiin tapauksiin ei ole. Joissain muissa ohjelmointikielissä on kuitenkin tehty erilaisia päätöksiä. Esimerkiksi Lua-ohjelmointikielessä perustietorakenne, Luan manuaalin [6] mukaan, atomisten tyyppien lisäksi on assosiatiivinen taulukko (*table*). Tämä tietorakenne on vastaavanlainen kuin esimerkiksi C#:n Dictionary-luokka, jossa alkiot voidaan indeksoida minkä tyyppisten avaimen avulla tahansa. Tämä tyyppi onkin ainoa kielessä oleva tietorakenne, jonka avulla voidaan määritellä myös muita tyyppejä. Lua-kielessä syntaksi *muuttuja["alkio"]* on sama kuin kirjoitettaisiin *muuttuja.alkio*.

6 LazyHashList — Ehdotus adaptoituvasta tietorakenteesta

Kuten luvussa 1 mainittiin, adaptoituva tietorakenne on rakenne, joka pystyy muuttamaan sisäistä tallennusrakennettaan tai käyttäytymistään tilanteen mukaan tehokkaammaksi. Adaptoituva tietorakenne olisi hyödyllinen varsinkin tilanteissa, joissa ei tiedetä ennalta varmasti, mitä kaikkia operaatioita tietorakenteeseen tullaan kohdistamaan, mutta halutaan, että tietorakenne suoriutuisi operaatioista tehokkaasti. Kuten luvussa 4.5 mainittiin, adaptoituvan tietorakenteen tulisi toteuttaa IEnumerable- ja ICollection-rajapinnat, mutta ei välttämättä erikoistuneempia rajapintoja, koska niille on jo olemassa tehokkaita toteutuksia.

Luvussa valitaan List- ja Dictionary-luokat, jotka ovat luvussa 5 esitettyjen mittaustulosten perusteella tehokkaimmat, ja pyritään yhdistämään näiden parhaat ominaisuudet yhteen tietorakenneluokkaan, joka toimisi kaikissa operaatioissa tehokkaasti. Luvussa esitellään menetelmä, jossa hajautusta hyödyntämällä saadaan etsimis- ja poisto-operaatioista tehokkaammat kuin List-luokalla. Hajautusta pyritään kuitenkin välttämään lisäysoperaatiossa, jotta se saataisiin pidettyä nopeana.

6.1 LazyHashList-luokan määrittely

Adaptoituvuutta voidaan lähestyä useammasta näkökulmasta. Jos halutaan tietorakenne, joka toteuttaa ICollection-rajapinnan, mutta ei haluta erikoistuneempaa käyttäytymistä, voitaisiin adaptoituvan tietorakenteen toteutusta lähestyä yhdistämällä toiminnallisuuksia jo olemassa olevista tietorakenteista. Jos tarkastellaan luvussa 5 esitettyjä mittaustuloksia, voidaan ensin valita tehokkaimmat tietorakenteet kunkin operaation kohdalta. Lisäysoperaatiossa taulukkolista oli tehokkain, etsimisoperaatiossa tehokkain rakenne oli ketjutusta käyttävä hajautusrakenne, joka oli myös poisto-operaatiossa tehokkain suurimmassa osassa tapauksista. Ainoastaan lisäysjärjestyksessä poistettaessa linkitetty lista oli tehokkaampi.

Jos tarkastellaan näitä tietorakenteita, voidaan huomata, että ne eivät ole suoraan yhteensopivia toistensa kanssa. Listarakenteeseen voidaan lisätä duplikaatteja eikä listarakenteessa alkioihin liitetä avaimia, jolloin sanakirjat ja listarakenteet eivät

voi sellaisinaan korvata toisiaan. Lisäksi taulukkoa tallennusrakenteena käyttävästä rakenteesta ei voida tehdä linkitettyä rakennetta muuten kuin kopioimalla kaikki taulukon alkiot linkitetyn rakenteen solmuihin.

Hajautusrakenteet kuitenkin käyttävät taulukkoa tallennusrakenteena, joten pienin muutoksin taulukkolista ja hajautusrakenne voitaisiin yhdistää. Tavoitteena listarakenteen ja hajautusrakenteen yhdistämisessä on säilyttää listarakenteen nopea lisäysoperaatio suurimmassa osassa tapauksista ja samalla nopeuttaa alkionetsimisoperaatiota hajauttamisen keinoin lineaarisesta kompleksisuudesta lähemmäksi vakioaikaista kompleksisuutta. Poisto-operaatiotakin voidaan nopeuttaa hajauttamisen keinoin.

Tietorakenteet voidaan yhdistää siten, että lopputuloksena saadaan tietorakenneluokka, joka toteuttaa ICollection-rajapinnan. Hajautusrakenteiden IDictionary- tai ISet-rajapintoja ei voida toteuttaa siksi, että duplikaattien salliminen nopeuttaa lisäysoperaatiota, koska samanarvoisen alkion olemassaoloa tietorakenteessa ei tarvitse tarkistaa. Listan IList-rajapintaakaan ei voida toteuttaa, koska IList sallii alkion lisäämisen minkä tahansa indeksin kohdalle. Tämä ei ole mahdollista, sillä mielivaltaisesti sijoittamisessa seuraavia alkioita täytyisi siirtää yhden askeleen eteenpäin ja tämä hajottaisi hajautuksessa käytettävän ketjutusalgoritmin, koska next-arvot eivät enää osoittaisi oikeiden indeksien kohdalle.

Lisäysoperaatio toteutettaisiin ICollection-rajapinnan Add-metodilla. Lisäysoperaatio pyrittäisiin säilyttämään nopeana siten, että uusi alkio lisättäisiin aina listan loppuun ilman hajauttamista aina, kun se olisi mahdollista. Alkiota ei lisättäisi listan loppuun vain silloin, kun alkiotaulukosta olisi tila loppu ja listasta olisi poistettu ainakin yksi hajautettu alkio. Tässä tapauksessa uusi alkio lisättäisiin vapaan indeksin kohdalle ja hajautettaisiin. Näin lisäysoperaatio saataisiin pidettyä mahdollisimman useassa tapauksessa lähes yhtä tehokkaana kuin taulukkolistalla.

Alkion etsiminen toteutettaisiin ICollection-rajapinnan Contains-metodilla. Alkion etsimisessä käytettäisiin joko hajauttamista tai lineaarista alkioiden läpikäyntiä alkioiden määrästä riippuen. Jos alkioita olisi lisätty tietorakenteeseen vain vähän voitaisiin alkiot vain käydä lineaarisesti läpi, koska hajauttamiseen kuluu aina aikaa, jolloin pienillä määrillä alkioita alkioiden läpikäynti voi olla aivan yhtä nopeaa. Eri vaihtoehtojen kokeilemisen tuloksena hajautusrajaksi luokalle annettiin neljä alkiota. Pienemmillä alkioiden määrällä hajautuksen käyttäminen oli hitaampaa. Tällä asetuksella on kuitenkin enemmän vaikutusta, kun alkioita on tallennettu tietorakenteeseen vähän. Tällä asetuksella ei siis ole hyvin suurta vaikutusta tut-

kielmassa suoritettaviin mittauksiin, koska testiotokset ovat vähintään sadan alkion kokoisia.

Jos alkioita on taas paljon, voidaan alkiot hajauttaa alkion etsimisen yhteydessä. Tällöin ensimmäiseen alkion etsimisoperaatioon saattaisi kulua enemmän aikaa kuin alkioiden läpikäyntiin, mutta tämä saattaisi nopeuttaisi seuraavia etsimisoperaatioita. Ensimmäiseen alkion etsimisoperaatioon kuluisi varmasti enemmän aikaa, kuin alkioiden läpikäyntiin, jos kaikki alkiot hajautettaisiin, mutta tämä todennäköisesti nopeuttaisi kaikkia seuraavia etsimisoperaatioita, jos tietorakennetta ei muutettaisi etsimisoperaatioiden välissä.

Hajautus voitaisiin kuitenkin toteuttaa myös siten, että alkioita käytäisiin läpi ja alkioita hajautettaisiin samalla, mutta tätä tehtäisiin vain niin kauan, kunnes etsittävä alkio löytyisi. Tällöin ensimmäisen etsimisoperaation yhteydessä ei hajautettaisi kaikkia alkioita, jos etsittävä alkio ei olisi listan viimeinen tai alkiota ei löytyisi ollenkaan. Tällöin ensimmäinen etsimisoperaatio ei välttämättä olisi niin raskas kuin kaikkien alkioiden hajauttaminen, mutta seuraaville etsimisoperaatioillekin saattaisi jäädä hajautettavaa, jos etsittävää alkiota ei olisi hajautettu edellisen etsimisoperaation yhteydessä. Täytyisi siis päättää, annettaisiinko ensimmäisen etsimisoperaation kestää kauemmin, vai yritettäisiinkö hajauttamiseen kuluvaa aikaa jakaa useammalle etsimisoperaatiolle. Tällaisessa hajauttamisessa täytyisi pitää yllä tietoa hajautettujen alkioiden lukumäärästä, joka samalla ilmaisisi indeksin, jossa olisi ensimmäinen alkio, jota ei olisi hajautettu. Tietorakenneluokalle valittiin tämä hajautustapa.

Hajautettujen alkioiden määrän muistaminen olisi hyödyllistä myös siksi, että etsimisoperaatioiden välissä tietorakenteeseen saatettaisiin lisätä uusia alkioita, joita ei heti hajautettaisi. Lisäysoperaatioiden jälkeen etsimisoperaatiossa käytettäisiin ensin hajautustaulukoita, ja jos alkiota ei tällä löydettäisi, etsittäisiin alkiota lineaarisesti alkaen hajautettujen alkioiden määrää ilmaisevan indeksin kohdalta. Edelleen ei-hajautettujen alkioiden määrästä riippuen alkiot voitaisiin vain käydä läpi tai niitä voitaisiin samalla hajauttaa.

Alkion poistaminen toteutettaisiin ICollection-rajapinnan Remove-metodilla. Alkion poistaminenkin voitaisiin pitää suhteellisen nopeana, koska ICollection-rajapinnan Remove-metodin tarvitsee poistaa vain ensimmäinen samanarvoinen alkio tietorakenteesta. Tällöin parhaassa tapauksessa tarvitsisi löytää vain ensimmäinen hajautettu samanarvoinen alkio ja merkitä alkion paikka vapaiden paikkojen ketjuun. Pahimmassa tapauksessa samanarvoista alkiota ei löydy hajautuksella eikä

myöskään hajauttamattomien alkioiden joukosta, jolloin pahimmassa tapauksessa täytyisi mennä kaikki rakenteen alkion läpi, jos ainuttakaan alkiota ei olisi vielä hajautettu.

Jos poistettavaa alkiota ei löytyisi hajautuksella, voitaisiin hajauttamattomia alkiota kulkea läpi järjestyksessä, ja kun poistettava alkio löytyisi, siirtää seuraavia alkioita yhden askeleen taaksepäin, jotta hajauttamattomien alkioiden väliin ei jäisi tyhjiä paikkoja. Alkioiden läpikäymisen yhteydessä voitaisiin myös suorittaa alkioiden hajauttamista, kunnes poistettava alkio löydettäisiin. Tällöin seuraavat alkion etsimis- ja poisto-operaatiot saattaisivat olla tehokkaampia. Jos alkioita hajautettaisiin vain siihen asti, kunnes poistettava alkio löytyisi, voitaisiin hajautusta taas jakaa useammalle operaatiolle.

Tällaista tietorakennetta voitaisiin kutsua laiskaksi hajautuslistaksi. Tutkielmassa toteutetun tietorakenne luokan nimeksi annettiin LazyHashList.

6.2 LazyHashList-luokan suorituskyky verrattuna List- ja Dictionary-luokkiin

LazyHashList-luokan suorituskyvyn mittaaminen suoritettiin samalla tavalla kuin luvussa 5 suoritetut mittaukset. Testiaineistona käytettiin samoja valmiiksi generoituja merkkijonoja ja testiotoksia otettiin sadan ja tuhannen alkion väliltä. Luokan suorituskykyä verrattiin List- ja Dictionary-luokkiin, joiden käyttäytymistä käytettiin mallina LazyHashList-luokan toteutuksessa.

Lisäysoperaation suorituskyvystä voidaan huomata, että LazyHashList-luokka suoriutuu lisäysoperaatiosta jopa hieman nopeammin kuin List-luokka, kun aloituskoko annetaan. Jos aloituskokoa ei anneta, LazyHashList-luokka suoriutuu lisäysoperaatiosta List-luokkaa hitaammin. Tämä ero voidaan selittää sillä, että uusi koko saadaan List-luokalle kaksinkertaistamalla vanha koko, mutta LazyHashList-luokka ensin kaksinkertaistaa vanhan koon ja etsii tämän jälkeen seuraavan alkuluvun alkulukutaulukosta. Lisäksi LazyHashList-luokassa luodaan kaksi taulukkoa, yksi taulukko indeksille ja toinen taulukko alkioille. Lisäksi koon kasvattamisen yhteydessä edellinen hajautus menetetään ja indeksitaulukon arvot alustetaan arvoon -1 ja arvoalkioiden next-arvot alustetaan arvoon -1. LazyHashList on lisäksi selvästi Dictionary-luokkaa nopeampi lisäysoperaatiossa, koska alkioita ei hajauteta lisäysoperaation yhteydessä. Lisäysoperaatio LazyHashList-luokalla näyttäisikin olevan vakioaikaista.

Etsimisoperaatiossa LazyHashList on List-luokkaa selvästi nopeampi, mutta hieman Dictionary-luokkaa hitaampi. Tätä tulosta voidaan selittää laiskalla hajautuksella. Kun etsitään ensimmäistä alkiota, ei ole vielä hajautettu ainuttakaan alkiota. Tässä ensimmäinen alkio hajautetaan. Toista alkiota etsittäessä ensimmäinen alkio on hajautettu, joten ensin etsitään hajautetuista alkioista. Koska toista alkiota ei ole vielä hajautettu, niin siirrytään etsimään alkiota lineaarisesti aloittaen toisen alkion kohdalta. Tässä toinen alkio hajautetaan. Kun etsitään listan loppupäässä olevia alkioita, on kaikki edelliset alkiot jo hajautettu. Tällöin ensin etsitään hajautuksen avulla, ja koska alkiota ei löydetä, siirrytään etsimään lineaarisesti. Listan loppupuolella lineaarisesti etsiminen on palon nopeampaa kuin List-luokalla, koska suurin osa listan alkioista voidaan ohittaa hajautuksen ansiosta. Tämä käyttäytyminen selittää myös sen, että LazyHashList-luokka on Dictionary-luokkaa hitaampi, koska Dictionary-luokassa kaikki alkiot on aina hajautettu, eikä alkioita tarvitse etsimisoperaation yhteydessä hajauttaa. Tulosten perusteella LazyHashList-luokan suorituskyky näyttäisi vakioaikaiselta, mutta Dictionary-luokkaa suuremmalla kertoimella.

Poisto-operaatiossa LazyHashList-luokka oli hitain, kun alkioita poistettiin lisäysjärjestyksessä. Tällöin hajautusta ei voida hyödyntää, koska poisto-operaation yhteydessä poistettavaa alkiota edeltävät alkiot hajautetaan, mutta poistettavaa alkiota ei hajauteta. Tässä tapauksessa poistettavaa alkiota seuraavia alkioita siirretään yhden askeleen verran taaksepäin arvotaulukossa ja viimeisen alkion next-arvo asetetaan arvoon -1, hash-arvo asetetaan arvoon -1 ja arvoksi asetetaan tyypin oletusarvo. Operaatiossa suoritetaan List-luokkaan verrattuna enemmän operaatioita, joka selittää hitaampaa suorituskykyä. Tässä tapauksessa poisto-operaatio on lineaariaikainen.

Kun alkioita poistettiin käänteisessä järjestyksessä, LazyHashList-luokka on selkeästi List-luokkaa nopeampi. Tässä tapauksessa hajautusta voidaan hyödyntää aina paitsi ensimmäisen poisto-operaation yhteydessä. Ensimmäisen poiston yhteydessä kaikki listan alkiot hajautetaan, jolloin kaikki seuraavat poisto-operaatiot voivat hyödyntää hajautusta. Seuraavien poisto-operaatioiden yhteydessä alkioita ei siirretä taaksepäin, vaan alkio asetetaan poistetuksi lisäämällä se poistettujen alkioiden ketjuun. Viimeisen alkion poistamisen jälkeen rakenne alustetaan uudelleen, jonka jälkeen lisäysoperaatiot voidaan taas suorittaa lisäämällä alkio listan loppuun ilman hajautusta. LazyHashList-luokka on tässä tapauksessa Dictionary-luokkaa hitaampi, ensimmäisen poisto-operaation yhteydessä suoritettavan hajau-

tuksen vuoksi. Tässä tapauksessa poisto-operaatio on keskimäärin vakioaikaista.

Kun alkiot poistettiin keskeltä LazyHashList-luokka on edelleen List-luokkaa nopeampi, mutta operaatio ei ole enää keskimäärin vakioaikaista. Tämä selittyy sillä, että ensimmäisen poisto-operaation yhteydessä noin puolet listan alkioista hajautetaan ja muita alkioita siirretään yhden askeleen verran taaksepäin. Tämän jälkeen joka toisessa poisto-operaatiossa alkio merkitään poistetuksi ja lisätään poistettujen alkioiden ketjuun. Muissa poisto-operaatiossa etsitään ensin hajautetuista alkioista ja tämän jälkeen etsitään lineaarisesti. Etsittävä alkio on kuitenkin aina ensimmäinen alkio lineaarisen etsimisen yhteydessä ja seuraavia alkioita siirretään yhden askeleen verran taaksepäin. Poisto-operaatiosta tulee näin keskimäärin lineaariaikainen, mutta pienemmällä kertoimella kuin, jos alkiot poistettaisiin aina listan alusta.

LazyHashList-luokassa onnistuttiin siis säilyttämään List-luokan suorituskyky lisäysoperaatiossa ja parantamaan etsimis- ja poisto-operaatioiden suorituskykyä List-luokkaan verrattuna huomattavasti suurimmassa osassa tapauksista. Dictionary-luokan suorituskykyyn ei ylletty etsimis- ja poisto-operaatioissa, mutta tämä ei olisi ollut mahdollista ilman lisäysoperaation hidastamista. Luokassa onnistuttiin siis toteuttamaan tutkielmassa esitetyn adaptoituvan tietorakenteen määritelmä suurimmassa osassa tapauksista.

	100	200	300	400	500	600	700	800	900	1000
List<string> (koko asetettu)	6,0	5,5	5,5	5,5	5,5	5,5	5,5	5,4	5,4	5,4
List<string> (kokoa ei asetettu)	11,5	9,8	9,4	8,5	7,9	8,5	8,1	7,7	7,5	7,3
Dictionary<string,string> (koko asetettu)	45,5	43,7	43,9	44,2	43,8	44,1	43,6	43,7	44,6	44,8
Dictionary<string,string> (kokoa ei asetettu)	79,0	80,7	69,0	64,2	74,0	69,0	66,3	64,3	63,0	75,4
LazyHashList<string> (koko asetettu)	5,4	5,2	5,1	5,2	5,2	5,2	5,2	5,1	5,2	5,1
LazyHashList<string> (kokoa ei asetettu)	30,4	17,7	21,8	17,7	25,1	21,8	19,4	17,6	16,3	25,9

Taulukko 6.1: Keskimääräinen yhteen lisäysoperaatioon kulunut aika nanosekunteina

	100	200	300	400	500	600	700	800	900	1000
List<string>	741,9	1419,2	2100,7	2783,8	3468,9	4164,3	4865,9	5572,7	6269,6	6972,2
Dictionary<string,string>	47,2	46,9	48,7	50,1	47,3	47,9	49,1	50,2	50,7	47,6
LazyHashList<string>	96,5	95,0	95,4	95,9	95,5	95,8	95,1	95,3	96,2	96,3

Taulukko 6.2: Keskimääräinen yhteen etsimisoperaatioon kulunut aika nanosekunteina

	100	200	300	400	500	600	700	800	900	1000
List<string>	118,2	133,8	148,7	166,5	183,7	200,9	218,0	235,5	252,5	269,7
Dictionary<string,string>	59,4	59,0	60,6	62,0	59,1	59,8	60,9	61,9	62,4	59,5
LazyHashList<string>	123,0	174,8	225,3	278,3	327,5	379,5	431,7	481,5	532,7	583,6

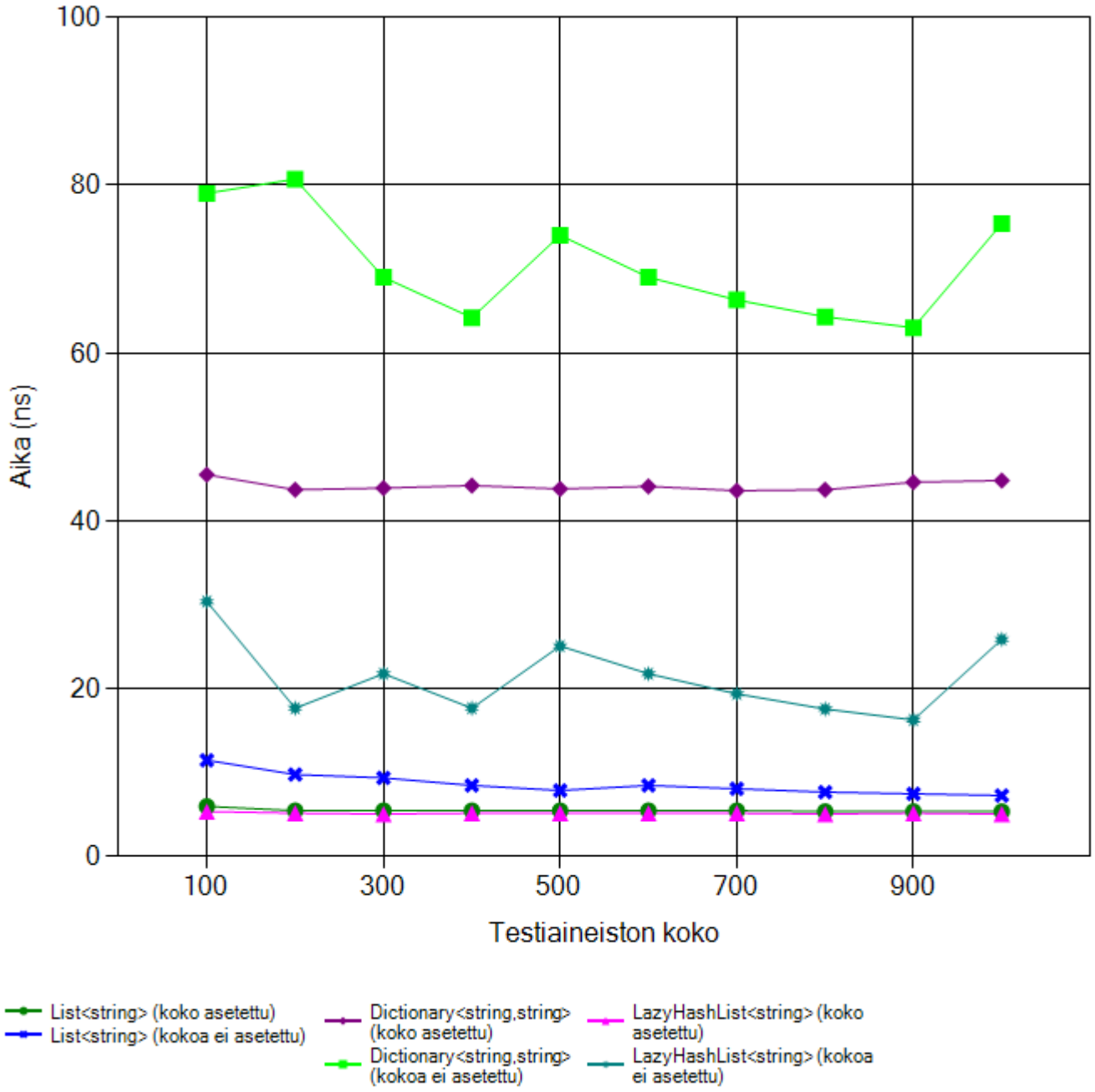
Taulukko 6.3: Keskimääräinen yhteen poisto-operaatioon kulunut aika nanosekunteina, kun alkiot poistettiin lisäysjärjestyksessä

	100	200	300	400	500	600	700	800	900	1000
List<string>	589,1	1131,1	1675,8	2221,5	2766,9	3316,1	3865,1	4412,8	4960,5	5507,8
Dictionary<string,string>	51,1	50,6	50,3	50,2	50,4	50,3	50,4	50,4	50,4	50,1
LazyHashList<string>	135,8	135,9	134,9	135,2	135,2	136,3	136,5	135,3	135,0	136,2

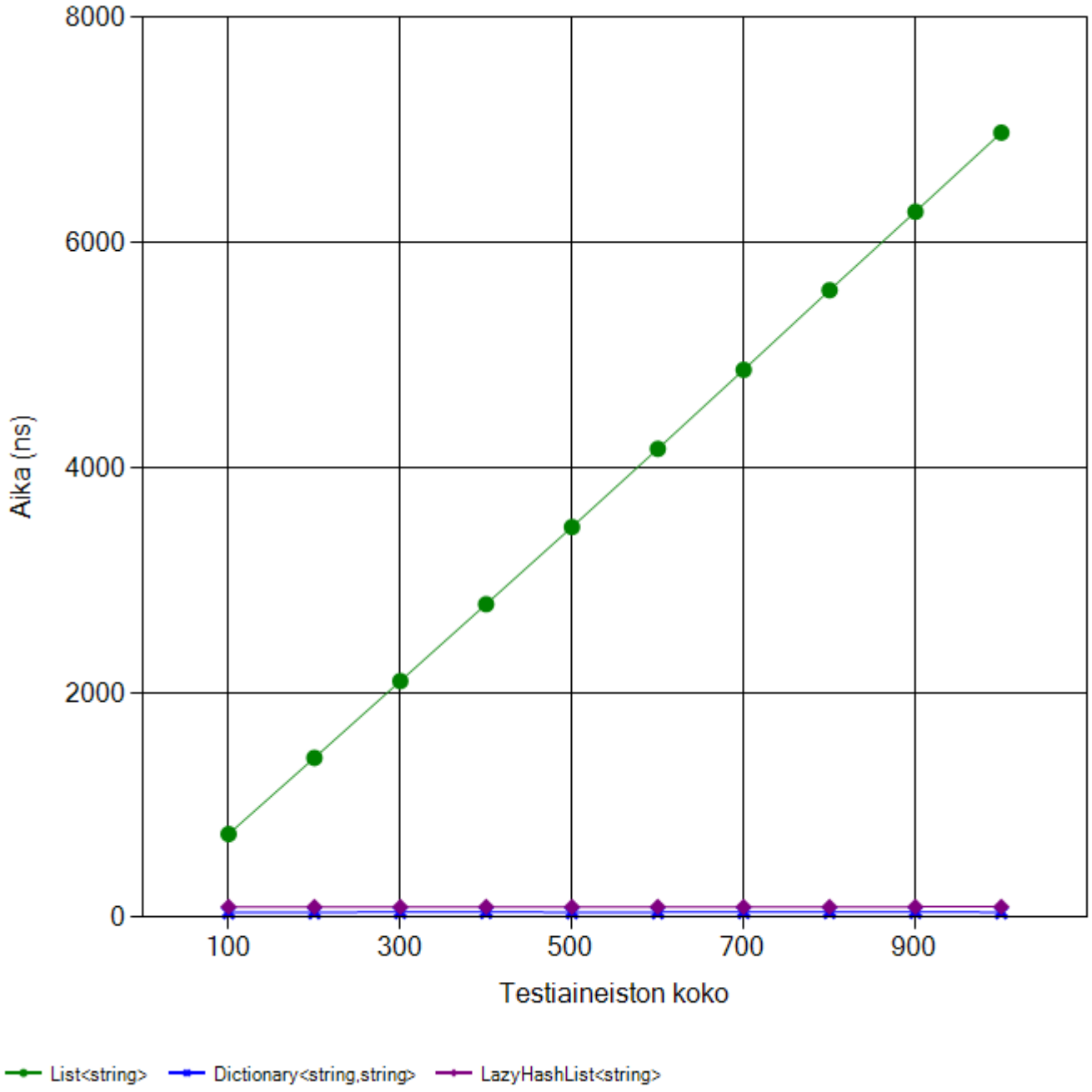
Taulukko 6.4: Keskimääräinen yhteen poisto-operaatioon kulunut aika nanosekunteina, kun alkiot poistettiin käänteisessä järjestyksessä

	100	200	300	400	500	600	700	800	900	1000
List<string>	383,5	663,3	946,1	1228,2	1507,7	1789,7	2071,6	2353,9	2638,1	2924,3
Dictionary<string,string>	54,0	53,2	54,8	55,0	53,9	53,9	54,8	55,2	55,8	53,6
LazyHashList<string>	120,6	133,5	145,8	159,6	171,6	184,5	197,4	210,5	223,8	237,2

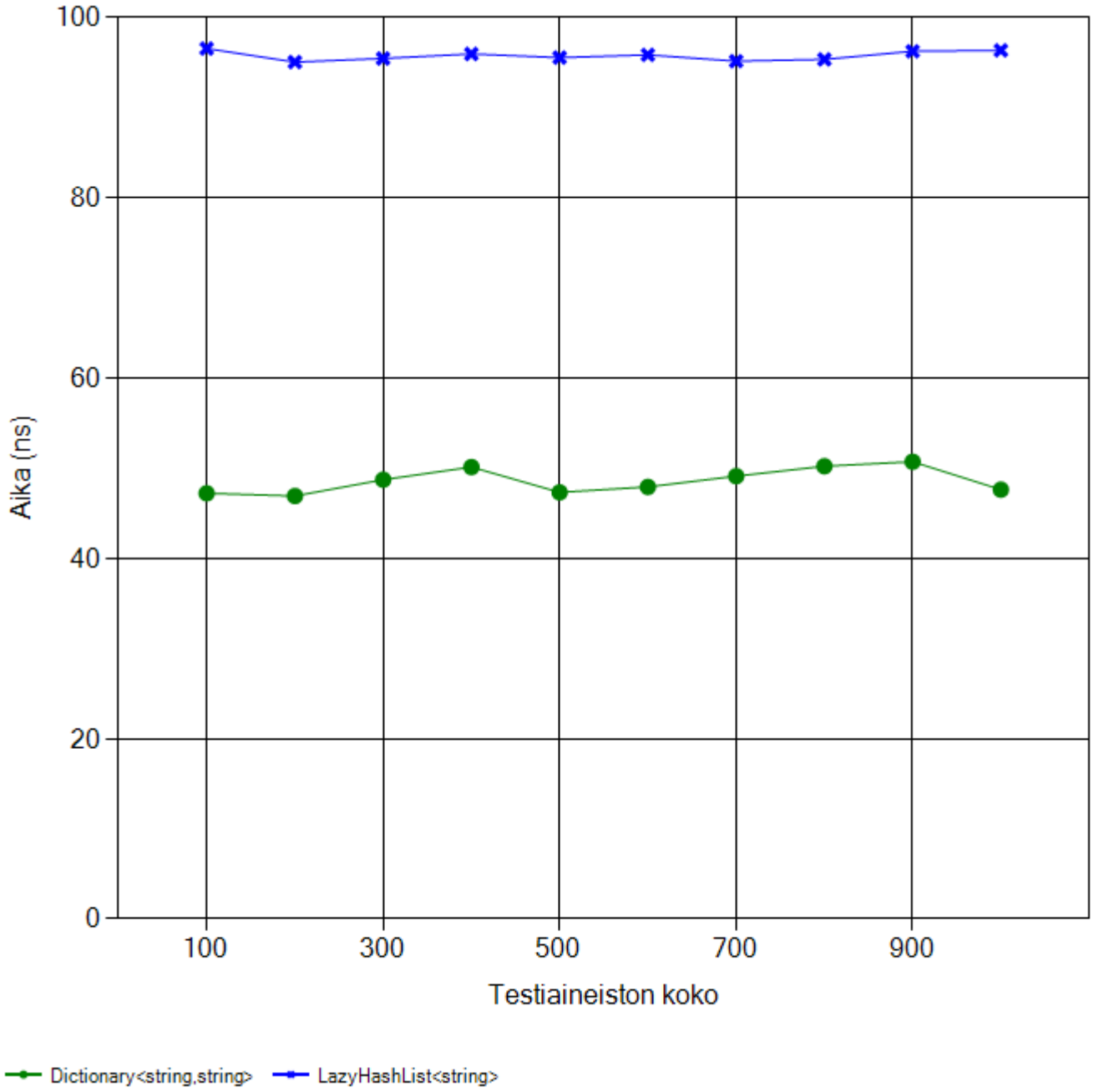
Taulukko 6.5: Keskimääräinen yhteen poisto-operaatioon kulunut aika nanosekunteina, kun alkiot poistettiin keskeltä



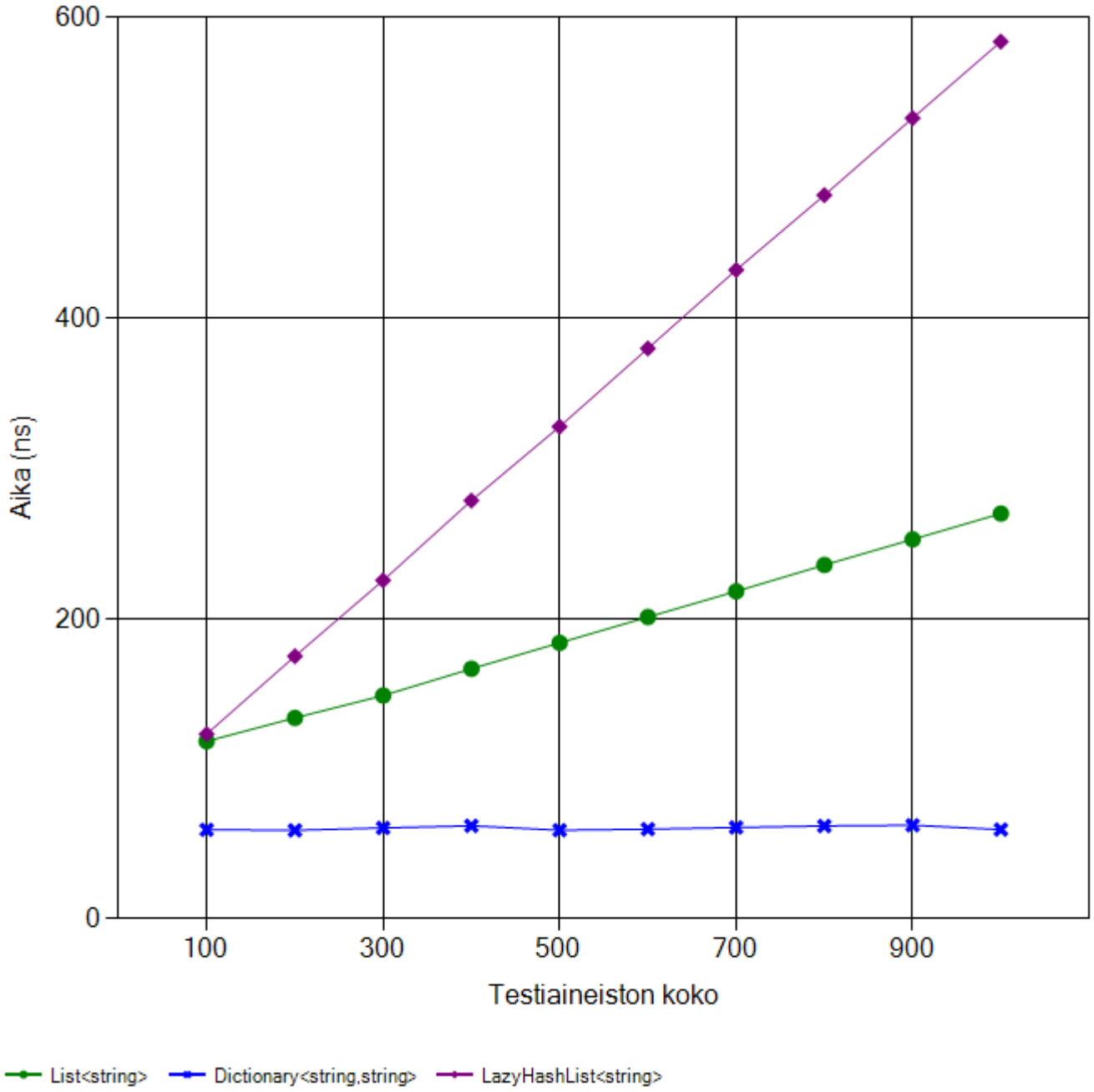
Kuva 6.1: Keskimääräinen yhteen lisäysoperaatioon kulunut aika luokilla: List, Dictionary ja LazyHashList



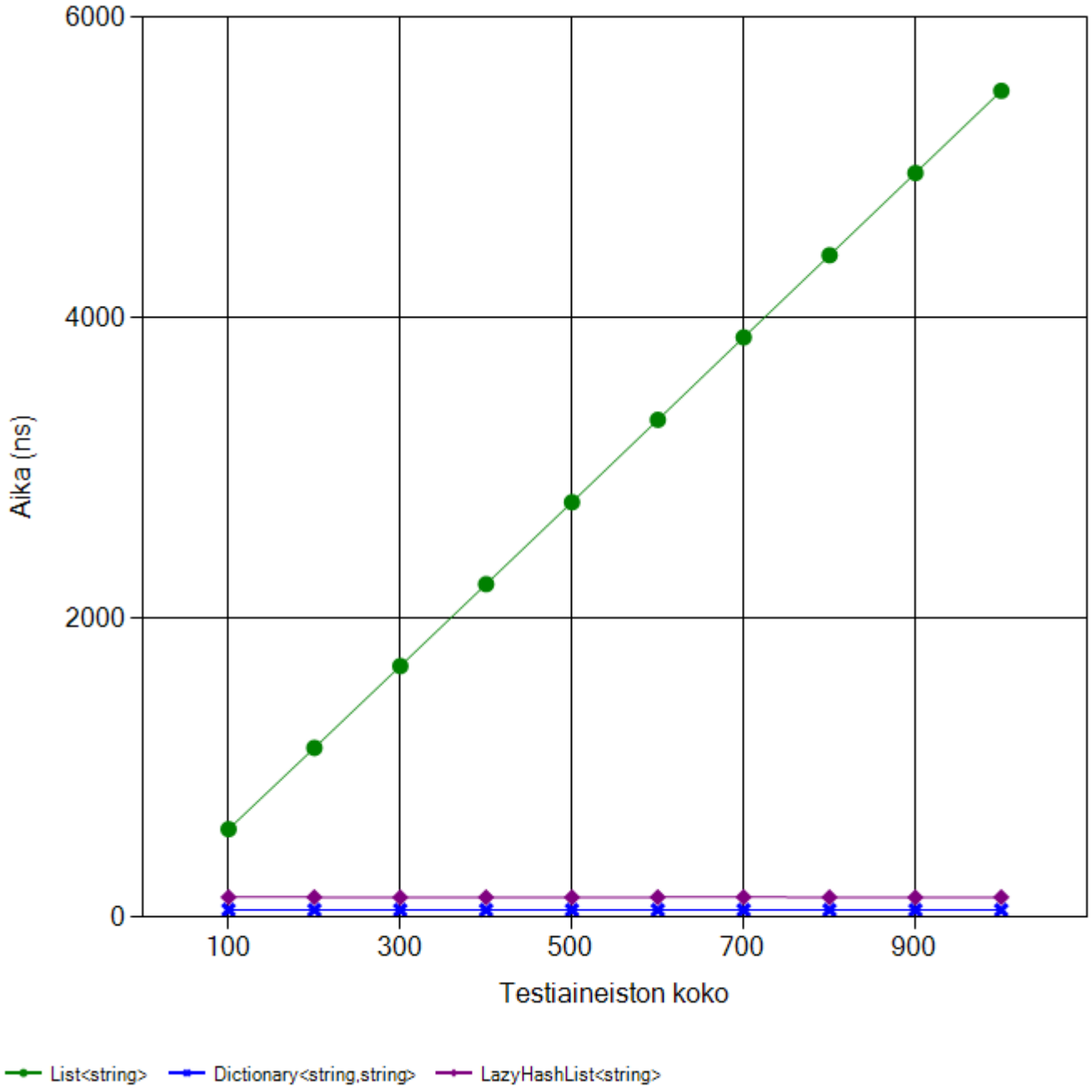
Kuva 6.2: Keskimääräinen yhteen etsimisoperaatioon kulunut aika luokilla: List, Dictionary ja LazyHashList



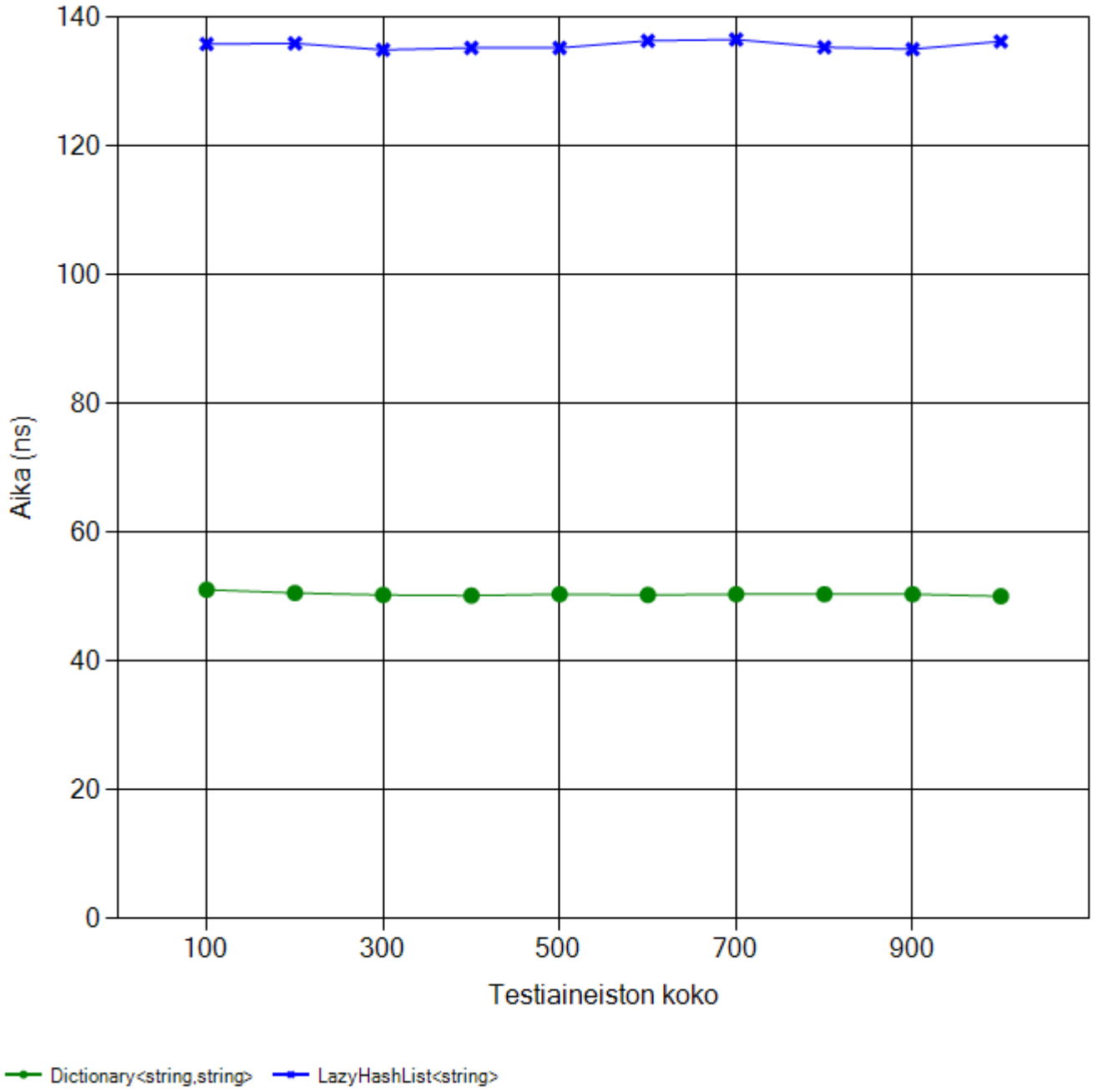
Kuva 6.3: Keskimääräinen yhteen etsimisoperaatioon kulunut aika luokilla: Dictionary ja LazyHashList



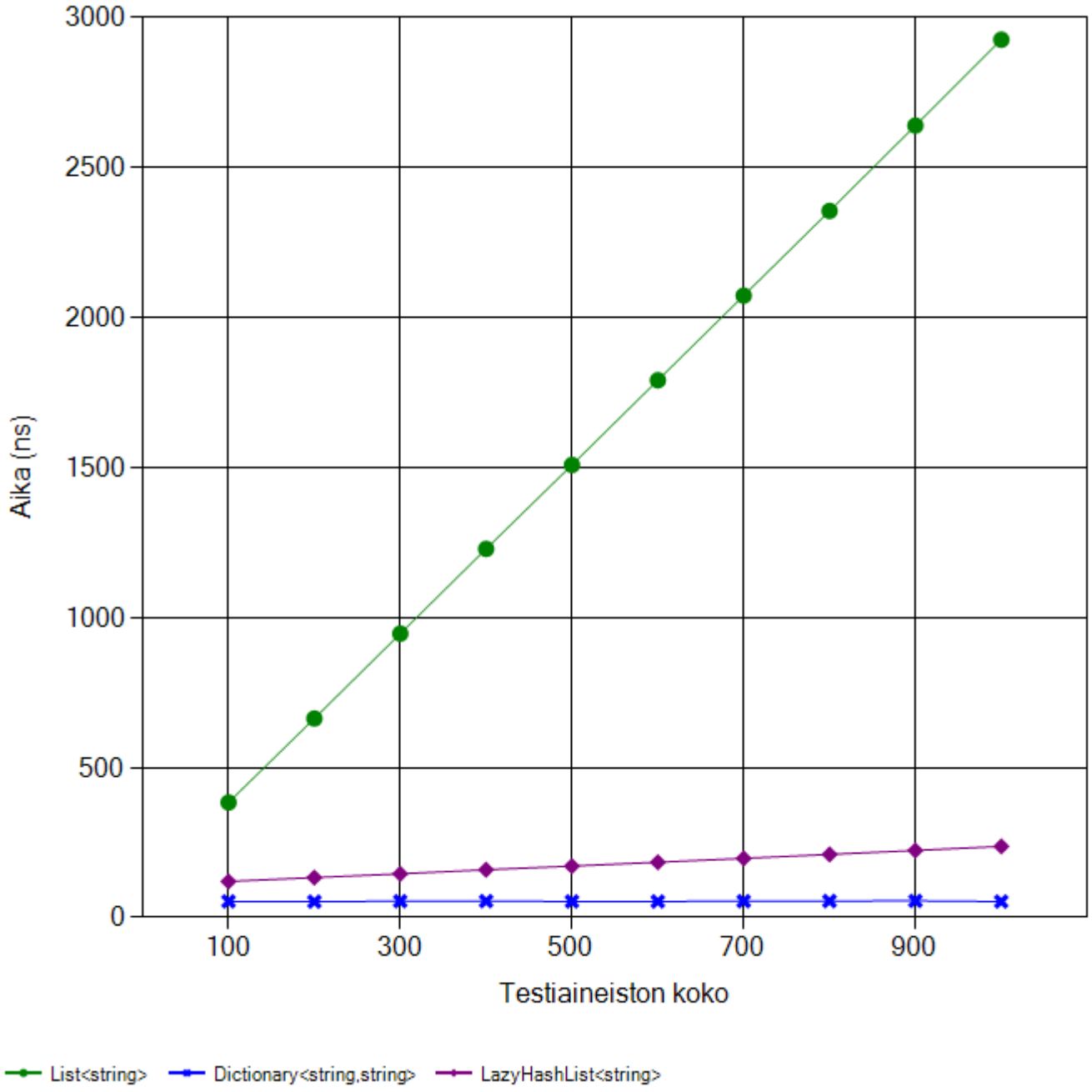
Kuva 6.4: Keskimääräinen yhteen poisto-operaatioon kulunut aika, kun alkiot poistettiin lisäysjärjestyksessä luokilla: List, Dictionary ja LazyHashList



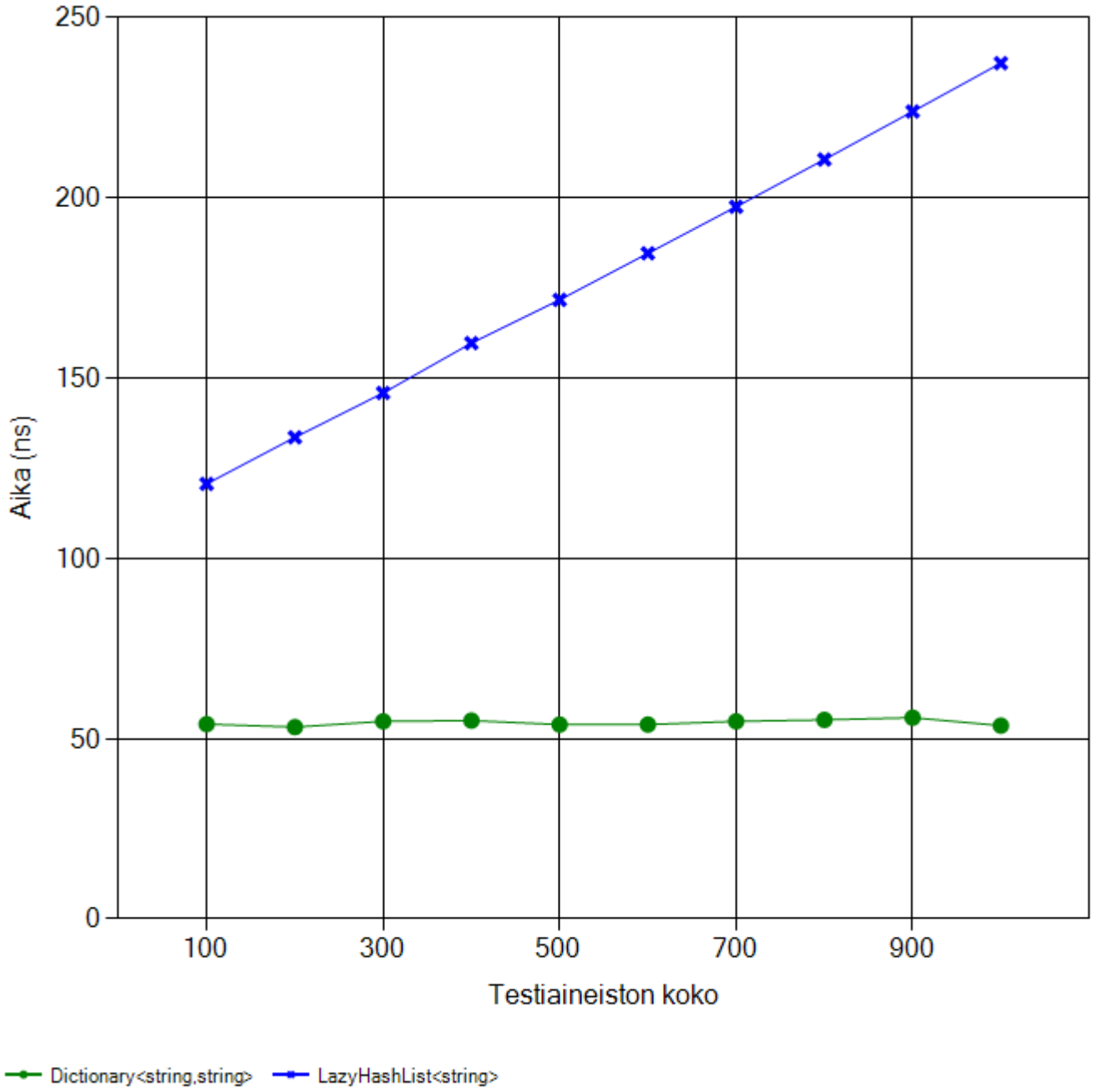
Kuva 6.5: Keskimääräinen yhteen poisto-operaatioon kulunut aika, kun alkiot poistettiin käänteisessä järjestyksessä luokilla: List, Dictionary ja LazyHashList



Kuva 6.6: Keskimääräinen yhteen poisto-operaatioon kulunut aika, kun alkiot poistettiin käänteisessä järjestyksessä luokilla: Dictionary ja LazyHashList



Kuva 6.7: Keskimääräinen yhteen poisto-operaatioon kulunut aika, kun alkiot poistettiin keskeltä luokilla: List, Dictionary ja LazyHashList



Kuva 6.8: Keskimääräinen yhteen poisto-operaatioon kulunut aika, kun alkiot poistettiin keskeltä luokilla: Dictionary ja LazyHashList

7 Aiheita jatkotutkimukseen

Luvussa 6 esiteltiin ehdotus adaptoituvasta tietorakenteesta sekä analysoitiin sen suorituskykyä eri tilanteissa. Toteutuksella onnistuttiin pääsemään kohtuullisen hyvään suorituskykyyn suurimmassa osassa tapauksia, mutta toteutus ei silti ollut täydellisesti adaptoituva. Esimerkiksi luokan suorituskyky osoittautui huonoksi, jos tietorakenteesta poistettiin aina ensimmäinen alkio, kun alkioita ei oltu vielä hajautettu. Lisäksi alkioiden etsiminen tietorakenteesta on rajoittunutta. Tietorakenteelta voidaan vain kysyä, onko tietorakenteessa jo samanarvoinen alkio antamalla alkio Contains-metodille parametriksi.

Jatkotutkimuksessa voitaisiin tutkia, voitaisiinko alkion etsimistä parantaa siten, että alkioita voitaisiin etsiä alkioiden ominaisuuksien perusteella, eikä pelkästään alkion olemassaololla tietorakenteessa, eli olisiko mahdollista vaihtaa indeksointiperustetta ajonaikaisesti käyttämään alkioiden eri ominaisuuksia, jolloin tietorakenteesta saataisiin entistä adaptoituvampi. Esimerkiksi tietorakenteeseen voitaisiin tallentaa seuraavanlaisen luokan ilmentymiä:

```
1  public class FooBar
2  {
3      public int Foo
4      {
5          get;
6          set;
7      }
8
9      public string Bar
10     {
11         get;
12         set;
13     }
14 }
```

Nyt alkioita voitaisiin haluta etsiä Bar-ominaisuuden perusteella. Voitaisiin tutkia, voisiko C#-kielen Func-tietotyyppiä käyttää tässä apuna. Func-tietotyyppi toimii funktio-osoittimen tavoin. Esimerkiksi jos FooBar-luokan ilmentymästä haluttaisiin saada Foo-ominaisuuden arvo, määriteltäisiin seuraavanlainen Func-tyyppinen funk-

tio:

```
1 Func<FooBar,int> func = input => input.Foo;
```

Tässä ensimmäinen geneerinen parametri kertoo funktion parametrin tyypin ja toinen geneerinen parametri kertoo funktion paluuarvon tyypin. Yllä määritelty funktio siis ottaisi FooBar tyyppisen parametrin ja palauttaisi int-tyyppisen arvon. Tässä tapauksessa arvo olisi parametrina annetun FooBar-olion Foo-ominaisuuden arvo. Vastaavasti alkioita voitaisiin haluta etsiä Bar-ominaisuuden avulla. Tällöin ei voitaisi käyttää samaa Func-tyyppistä funktiota, koska halutaan eri ominaisuus ja Bar-ominaisuuden arvo on string eikä int. Tässä haasteena olisi nykyisen valintafunktion tallentaminen, koska paluuarvo saattaa muuttua.

8 Yhteenveto

Tutkielmassa tarkasteltiin eri tietorakenteita ja niiden suorituskykyä tietorakenteiden perusoperaatioissa. Tutkielmassa tarkasteltiin tietorakenteita, jotka toteuttavat abstraktien tietotyyppien kokoelma, joukko ja sanakirja ominaisuudet. Toteutuksiksi valittiin C#-kielen Systems.Collections-nimiavaruudesta List-, LinkedList-, HashSet-, Dictionary- ja Hashtable-luokat sekä itse tehty binääripuuluokka.

Luokkien suorituskykyä analysoitiin ensin tarkastelemalla tietorakenneluokkien toteuttamien operaatioiden kompleksisuutta algoritmianalyysin keinoin luvussa 4. Tämän jälkeen luokkien suorituskykyä testattiin simuloinnin avulla. Testausta varten luotiin testiaineistoksi satunnaisia yksilöllisiä merkkijonoja, jotka tallennettiin tekstitiedostoon. Testiaineistosta otettiin erikokoisia otoksia, joilla testattiin luokkien suorituskykyä lisäys-, etsimis- ja poisto-operaatioissa. Testauksen tulokset esitettiin kuvaajina ja taulukkomuotoisina, ja luokkien operaatioita analysoitiin tulosten perusteella luvussa 5.

Mittaustulosten perusteella testatuista tietorakenneluokista ei löydetty yhtä toteutusta, joka olisi suoriutunut parhaiten kaikista operaatioista. Tämän perusteella pääteltiin, että adaptoituvalle tietorakenteelle voisi olla käyttöä. Tulosten perusteella valittiin luokat, jotka olivat suoriutuneet hyvin eri operaatioista. Testatuista luokista parhaiten suoriutuneet luokat olivat List ja Dictionary. Näiden luokkien pohjalta yritettiin toteuttaa tietorakenneluokka, joka yhdistäisi luokkien hyvät ominaisuudet erityisesti suorituskyvyn suhteen.

Luokka toteutettiin siten, että siinä pyrittiin yhdistämään List-luokan hyvä suorituskyky lisäysoperaatiossa ja Dicitionary-luokan hyvä suorituskyky etsimis- ja poisto-operaatioissa. Toteutuksessa onnistuttiin siten, että hyvä suorituskyky onnistuttiin säilyttämään lisäysoperaatiossa ja samalla suorituskykyä onnistuttiin parantamaan etsimis- ja poisto-operaatioissa List-luokkaan verrattuna. Dictionary-luokan suorituskykyyn etsimis- ja poisto-operaatioissa ei kuitenkaan ylletty, koska ei haluttu uhrata lisäysoperaation hyvää suorituskykyä. Etsimis- ja poisto-operaatiota parannettiin hajautuksen keinoin. Hajautuksessa käytettiin samaa ketjutusalgoritmia, jota Dictionary-luokassa käytetään. Näissä operaatioissa alkioita hajautettiin laiskasti. Tämä tarkoittaa sitä, että alkioita hajautettiin vain niin etsittävään alkioon

asti. Lisäksi jos alkio oli jo hajautettu, ei operaation aikana hajautettu ollenkaan alkioita. Tämä tekniikka nopeutti keskimääräistä etsimis- ja poisto-operaatiota suurimmassa osassa tapauksia. Operaatioiden kompleksisuus säilyi silti lineaarisena, mutta pienemmällä kertoimella kuin List-luokassa.

Lähteet

[1] Douglas Baldwin, Gregg Scragg, *Algorithms and Data Structures : The Science of Computing*, Charles River Media, 2004, saatavilla ebrarystä <URL:http://site.ebrary.com/lib/jyvaskyla/docDetail.action?docID=10066529>, viitattu: 30.1.2011.

[2] Richard Connor, *Abstract data type*, Encyclopedia of Computer Science, 4th edition, 2003, saatavilla verkosta <URL:http://portal.acm.org/citation.cfm?id=1074100.1074101>, viitattu: 17.10.2010.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, McGraw-Hill Book Company, 1990.

[4] Jay Earley, *Toward an understanding of data structures*, International Conference on Management of Data, Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control, 1970, saatavilla verkosta <URL:http://portal.acm.org/citation.cfm?id=1734663.1734665>, viitattu: 25.9.2010.

[5] Michael T. Goodrich, Roberto Tamassia, *Data Structures and Algorithms in Java*, Second edition, John Wiley & Sons, Inc., 2001.

[6] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, *Lua 5.1 Reference Manual*, saatavilla verkosta <URL:http://www.lua.org/manual/5.1/manual.html>, viitattu: 21.8.2011.

[7] Jeffrey H. Kingston, *Algorithms and Data Structures: Design, Correctness, Analysis*, International Computer Science Series, University of Sidney, Addison Wesley Publishing Company, 1990.

[8] Donald E. Knuth, *The art of computer programming: Volume 1 Fundamental algorithms*, Third Edition, Standford University, Addison Wesley Publishing Company, 1997.

[9] Microsoft, *MSDN: Array.Copy Method* , saatavilla verkosta <URL:http://msdn.microsoft.com/en-us/library/z50k9bft.aspx>, viitattu 13.8.2011.

[10] Microsoft, *MSDN: System.Collections Namespaces, .NET Framework 4*, saatavilla verkosta <URL:http://msdn.microsoft.com/en-us/library/gg145035.aspx>, viitattu: 30.7.2011.

[11] Scott Mitchell, *An Extensive Examination of Data Structures Using C# 2.0*, MSDN 2005, saatavilla verkosta<URL:http://msdn.microsoft.com/en-us/library/ms379571(VS.80).aspx>, viitattu: 26.7.2011.

[12] Antti Valmari, *Käytännön kokemus algoritmikirjastojen autuudesta*, Tampereen teknillinen yliopisto, Ohjelmistotekniikan laitos, saatavilla verkosta <URL:http://www.tkts.fi/lehti/a29/valmari.pdf>, viitattu: 13.11.2011.

[13] Mark Allen Weiss, *Algorithms, data structures, and problem solving with C++*, Addison-Wesley Publishing Company, Inc., 1996.

[14] Niklaus Wirth, *Algorithms & Data Structures*, Prentice-Hall, Inc., 1986.

[15] Derrick Wood, *Data Structures, Algorithms, and Performance*, University of Waterloo, Addison-Wesley Publishing Company, 1993.

Liitteet

Mittaustulosten kokonaisajat

	100	200	300	400	500	600	700	800	900	1000
List<string> (koko asetettu)	61	108	170	217	279	340	386	434	495	549
List<string> (kokoa ei asetettu)	116	201	283	343	395	515	564	617	675	727
LinkedList<string>	240	481	721	962	1187	1447	1693	1937	2176	2409
Dictionary<string,string> (koko asetettu)	476	907	1353	1817	2245	2711	3115	3568	4102	4579
Dictionary<string,string> (kokoa ei asetettu)	812	1674	2104	2622	3754	4205	4718	5230	5754	7616
HashSet<string>	760	1512	2002	2511	3508	3970	4473	4991	5499	7077
Hashtable (koko asetettu)	707	1358	2179	2631	3432	4169	4804	5409	6386	6758
Hashtable (kokoa ei asetettu)	1249	2625	3643	5465	6247	7152	10431	11144	11916	12774
BinarySearchTree<string,string>	13589	30677	51014	71333	93743	117392	142428	165529	190669	217426
LazyHashList<string> (koko asetettu)	55	105	152	209	256	313	360	412	463	513
LazyHashList<string> (kokoa ei asetettu)	298	347	642	697	1239	1293	1346	1395	1447	2562

Taulukko 8.1: Kokonaisajat lisäysoperaatiossa millisekunteina. Ensimmäinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string> (koko asetettu)	60	109	165	218	271	327	381	434	486	539
List<string> (kokoa ei asetettu)	115	193	283	336	392	508	564	617	674	724
LinkedList<string>	237	477	715	952	1199	1436	1671	1911	2153	2403
Dictionary<string,string> (koko asetettu)	444	857	1299	1742	2165	2612	3019	3456	3971	4431
Dictionary<string,string> (kokoa ei asetettu)	779	1584	2053	2539	3672	4108	4608	5101	5621	7500
HashSet<string>	734	1479	1959	2456	3438	3879	4375	4885	5419	6963
Hashtable (koko asetettu)	716	1378	2213	2675	3500	4246	4895	5509	6502	6892
Hashtable (kokoa ei asetettu)	1259	2652	3660	5499	6286	7201	10505	11247	12019	12903
BinarySearchTree<string,string>	13189	29664	49445	69078	90873	113798	138024	160645	185125	210929
LazyHashList<string> (koko asetettu)	54	103	155	207	259	310	362	413	466	515
LazyHashList<string> (kokoa ei asetettu)	307	358	661	713	1264	1313	1366	1417	1472	2609

Taulukko 8.2: Kokonaisajat lisäysoperaatiossa millisekunteina. Toinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string> (koko asetettu)	60	110	164	219	270	326	379	432	487	541
List<string> (kokoa ei asetettu)	115	194	284	337	391	508	564	617	671	725
LinkedList<string>	239	477	715	954	1191	1436	1672	1919	2165	2399
Dictionary<string,string> (koko asetettu)	445	855	1298	1741	2164	2610	3012	3454	3969	4431
Dictionary<string,string> (kokoa ei asetettu)	779	1586	2056	2540	3673	4112	4607	5104	5626	7499
HashSet<string>	736	1486	1958	2452	3445	3884	4381	4893	5408	6990
Hashtable (koko asetettu)	717	1359	2186	2629	3437	4182	4820	5414	6414	6777
Hashtable (kokoa ei asetettu)	1258	2651	3665	5498	6290	7205	10514	11253	12026	12910
BinarySearchTree<string,string>	13211	29676	49455	69210	90976	113919	138012	160653	185131	210883
LazyHashList<string> (koko asetettu)	52	104	156	207	258	309	363	410	464	515
LazyHashList<string> (kokoa ei asetettu)	307	359	662	712	1267	1319	1368	1420	1472	2610

Taulukko 8.3: Kokonaisajat lisäysoperaatiossa millisekunteina. Kolmas ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	7230	27977	62401	110496	172407	248521	339210	443999	562493	694508
LinkedList<string>	6761	26085	58114	103282	163356	237123	324159	424691	538431	665240
Dictionary<string,string>	474	936	1460	2003	2365	2884	3435	4017	4563	4768
HashSet<string>	559	1089	1563	2123	2657	3238	3740	4253	4831	5383
Hashtable	565	1004	1820	2049	2720	3493	3441	4077	4721	5451
BinarySearchTree<string,string>	13313	30091	50019	69927	91934	115156	139653	162512	187240	213255
LazyHashList<string>	966	1905	2871	3832	4773	5739	6668	7622	8649	9631

Taulukko 8.4: Kokonaisajat etsimisoperaatiossa millisekunteina. Ensimmäinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	7253	28028	62458	110568	172457	248581	339297	443962	562646	695056
LinkedList<string>	6764	26127	58121	103286	163097	236725	323682	424046	537631	664369
Dictionary<string,string>	470	939	1460	2007	2364	2872	3438	4020	4560	4746
HashSet<string>	559	1090	1565	2126	2657	3237	3745	4250	4825	5379
Hashtable	568	1002	1818	2048	2718	3492	3416	4073	4723	5440
BinarySearchTree<string,string>	13292	30074	49904	69841	91768	114890	139279	162178	186881	212860
LazyHashList<string>	962	1894	2854	3821	4746	5725	6560	7606	8629	9624

Taulukko 8.5: Kokonaisajat etsimisoperaatiossa millisekunteina. Toinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	7774	29146	64203	112988	175474	252472	343332	449479	567664	702081
LinkedList<string>	6972	26519	58740	104182	164205	238058	325204	425794	539512	666410
Dictionary<string,string>	473	936	1462	2007	2361	2871	3436	4009	4553	4769
HashSet<string>	571	1114	1592	2163	2712	3307	3809	4340	4928	5495
Hashtable	553	978	1776	2004	2652	3401	3367	3980	4620	5327
BinarySearchTree<string,string>	13147	29741	49301	68942	90618	113479	137471	160142	184491	210406
LazyHashList<string>	966	1903	2860	3858	4801	5777	6738	7641	8702	9627

Taulukko 8.6: Kokonaisajat etsimisoperaatiossa millisekunteina. Kolmas ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	1195	2714	4486	6711	9245	12127	15358	18914	22765	27058
LinkedList<string>	422	850	1366	1712	2251	2575	3033	3467	4147	4328
Dictionary<string,string>	599	1193	1843	2507	2993	3628	4312	5011	5682	6019
HashSet<string>	686	1348	1954	2635	3298	4000	4653	5272	5974	6664
Hashtable	676	1221	2108	2499	3294	4106	4137	4887	5681	6516
BinarySearchTree<string,string>	6789	18665	32408	44963	62378	76649	90868	110683	117327	138637
LazyHashList<string>	1230	3487	6742	11112	16325	22731	30148	38464	47874	58289

Taulukko 8.7: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin lisäysjärjestyksessä. Ensimmäinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	1176	2652	4460	6607	9158	11999	15201	18821	22723	26878
LinkedList<string>	422	850	1353	1709	2276	2578	3031	3496	4142	4321
Dictionary<string,string>	598	1196	1837	2508	2995	3626	4317	5016	5677	6025
HashSet<string>	676	1326	1921	2598	3260	3947	4572	5214	5906	6575
Hashtable	677	1222	2108	2503	3290	4110	4135	4889	5673	6514
BinarySearchTree<string,string>	6735	18551	32264	44712	61972	76081	90106	109959	116670	137862
LazyHashList<string>	1217	3483	6752	11118	16382	22732	30195	38473	47879	58323

Taulukko 8.8: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin lisäysjärjestyksessä. Toinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	1174	2661	4441	6656	9154	12030	15221	18776	22693	26966
LinkedList<string>	412	835	1263	1795	2170	2679	2966	3380	3833	4441
Dictionary<string,string>	585	1152	1772	2419	2882	3505	4162	4826	5484	5818
HashSet<string>	660	1292	1871	2527	3160	3840	4442	5057	5734	6397
Hashtable	714	1285	2223	2617	3462	4312	4309	5101	5925	6816
BinarySearchTree<string,string>	7044	19399	33694	46629	64566	79160	93820	114341	121424	143468
LazyHashList<string>	1242	3518	6787	11171	16421	22843	30306	38628	48084	58459

Taulukko 8.9: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin lisäysjärjestyksessä. Kolmas ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	5883	22618	50272	88899	138357	199040	270661	353211	446755	550922
LinkedList<string>	6784	26227	58405	103830	163830	237730	324956	425553	539358	666509
Dictionary<string,string>	527	1045	1564	2081	2613	3134	3651	4179	4699	5241
HashSet<string>	566	1124	1684	2245	2811	3380	3943	4510	5076	5648
Hashtable	651	1215	2103	2455	3228	4042	4074	4811	5567	6411
BinarySearchTree<string,string>	13036	29251	48732	68296	89598	112350	136268	158489	182786	208030
LazyHashList<string>	1360	2729	4044	5394	6754	8191	9508	10871	12211	13680

Taulukko 8.10: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin käänteisessä järjestyksessä. Ensimmäinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	5891	22616	50263	88821	138341	198943	270577	353036	446463	550723
LinkedList<string>	6775	26197	58299	103643	163629	237407	324679	424980	538805	666044
Dictionary<string,string>	527	1048	1565	2085	2615	3126	3676	4197	4714	5109
HashSet<string>	566	1125	1677	2248	2812	3378	3924	4510	5076	5646
Hashtable	652	1221	2104	2443	3215	4032	4042	4800	5556	6407
BinarySearchTree<string,string>	12962	29160	48603	67941	89302	112000	135838	158131	182209	207663
LazyHashList<string>	1355	2716	4041	5411	6755	8158	9580	10795	12117	13629

Taulukko 8.11: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin käänteisessä järjestyksessä. Toinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	5898	22630	50290	88860	138344	198908	270437	352817	446108	550697
LinkedList<string>	6787	26295	58571	104041	163769	237638	324634	425028	538629	665861
Dictionary<string,string>	478	940	1400	1863	2331	2791	3259	3722	4188	4669
HashSet<string>	503	998	1492	1991	2489	2988	3491	3990	4484	4983
Hashtable	609	1132	1995	2286	3022	3803	3762	4470	5192	6012
BinarySearchTree<string,string>	13061	29339	48914	68363	89823	112576	136591	158953	183204	208749
LazyHashList<string>	1359	2708	4053	5416	6775	8191	9570	10794	12113	13543

Taulukko 8.12: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin käänteisessä järjestyksessä. Kolmas ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	3843	13283	28429	49216	75429	107433	145082	188408	237485	292500
LinkedList<string>	3709	13753	30209	53076	82115	117563	159610	208873	265267	329054
Dictionary<string,string>	545	1072	1654	2214	2713	3258	3870	4443	5061	5388
HashSet<string>	588	1162	1716	2291	2898	3482	4077	4606	5234	5798
Hashtable	642	1177	2100	2398	3202	4087	3931	4650	5434	6267
BinarySearchTree<string,string>	11342	26426	43917	62571	81995	102153	124328	145820	167242	190547
LazyHashList<string>	1214	2679	4390	6393	8600	11100	13831	16858	20211	23810

Taulukko 8.13: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin keskeltä. Ensimmäinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	3833	13272	28378	49111	75389	107370	145009	188283	237407	292372
LinkedList<string>	3702	13805	30100	52865	81909	117400	159527	208534	264888	328622
Dictionary<string,string>	543	1072	1654	2215	2713	3260	3856	4442	5047	5392
HashSet<string>	586	1160	1714	2289	2899	3482	4074	4603	5231	5798
Hashtable	641	1176	2099	2387	3194	4079	3927	4639	5401	6251
BinarySearchTree<string,string>	11214	26165	43436	61889	81183	101190	123156	144465	165739	188900
LazyHashList<string>	1208	2679	4397	6403	8604	11110	13869	16903	20171	23735

Taulukko 8.14: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin keskeltä. Toinen ajo.

	100	200	300	400	500	600	700	800	900	1000
List<string>	3828	13240	28346	49059	75336	107337	144950	188255	237390	292405
LinkedList<string>	3734	13828	30244	53037	82147	117717	159663	208720	265212	329256
Dictionary<string,string>	533	1047	1621	2167	2657	3185	3788	4351	4955	5286
HashSet<string>	588	1165	1724	2297	2901	3485	4075	4607	5236	5794
Hashtable	616	1127	2029	2311	3085	3942	3787	4480	5220	6023
BinarySearchTree<string,string>	11178	26066	43264	61654	80879	100853	122666	143947	165158	188302
LazyHashList<string>	1196	2650	4333	6351	8539	10995	13744	16767	20055	23601

Taulukko 8.15: Kokonaisajat poisto-operaatiossa millisekunteina, kun alkiot poistettiin keskeltä. Kolmas ajo.

LazyHashList-luokka

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Diagnostics;
6
7  namespace HH.Gradu.CSharp.DataStructures
8  {
9      [DebuggerDisplay("Count = {Count}")]
10     public class LazyHashList<TValue> : ICollection<TValue>
11     {
12
```

```
13          public LazyHashList() : this(0, null)
14          {
15
16          }
17
18          public LazyHashList(IEqualityComparer<TValue> comparer)
19              : this(0, comparer)
20          {
21          }
22
23          public LazyHashList(int capacity) :this(capacity, null)
24          {
25          }
26
27          public LazyHashList(IEnumerable<TValue> source)
28              : this(source, null)
29          {
30
31
32          }
33
34          public LazyHashList(
35              IEnumerable<TValue> source,
36              IEqualityComparer<TValue> comparer)
37              : this(source.Count(), comparer)
38          {
39              foreach (TValue item in source)
40              {
41                  this.Add(item);
42              }
43          }
44
45          public LazyHashList(int capacity, IEqualityComparer<TValue> comparer)
46          {
47              if (comparer == null)
48              {
49                  this.comparer = EqualityComparer<TValue>.Default;
50              }
51              else
52              {
53                  this.comparer = comparer;
54              }
```

```
55
56              int prime = Helper.GetPrime(capacity);
57              buckets = new int[prime];
58              entries = new HashEntry[prime];
59              freeList = -1;
60              freeCount = 0;
61              count = 0;
62              version = 0;
63              hashThreshold = 0;
64              InitBuckets();
65          }
66
67          private void InitBuckets()
68          {
69              for (int i = 0; i < buckets.Length; i++)
70              {
71                  buckets[i] = -1;
72                  entries[i].Next = -1;
73              }
74          }
75
76          private int[] buckets;
77
78          private HashEntry[] entries;
79
80          private int count;
81          private int hashCount;
82
83          private int freeCount;
84          private int freeList;
85
86          private int version;
87
88          private int hashThreshold;
89
90          private IEqualityComparer<TValue> comparer;
91
92          public int Count
93          {
94              get
95              {
96                  return count - freeCount;
```

```
97              }
98          }
99
100         public void Add(TValue value)
101         {
102             if (count == buckets.Length)
103             {
104                 if (freeCount > 0)
105                 {
106                     AddToFree(value);
107                 }
108                 else
109                 {
110                     Expand();
111                 }
112             }
113             else
114             {
115                 entries[count++].Value = value;
116             }
117
118             this.version++;
119         }
120
121         public bool Remove(TValue value)
122         {
123             bool removed = false;
124             if (hashCount > 0)
125             {
126                 removed = RemoveFromHash(value);
127             }
128             if (!removed)
129             {
130
131                 if (count − hashCount <= hashThreshold)
132                 {
133                     removed = RemoveLinear(value);
134                 }
135                 else
136                 {
137                     removed = HashAndRemoveLinear(value);
138                 }
```

```
139
140             }
141             if (removed && Count == 0)
142             {
143                 //all entries removed
144                 Clear();
145             }
146             return removed;
147         }
148
149         private bool HashAndRemoveLinear(TValue value)
150         {
151             int hash, index;
152             for (int i = hashCount; i < count; i++)
153             {
154                 if (this.comparer.Equals(this.entries[i].Value, value))
155                 {
156                     this.count--;
157                     if (i < this.count)
158                     {
159                         Array.Copy(this.entries, i + 1, this.entries, i, this.count - i);
160                     }
161                     this.entries[count].Hash = -1;
162                     this.entries[count].Value = default(TValue);
163                     this.entries[count].Next = -1;
164                     this.version++;
165                     return true;
166                 }
167                 hash = this.comparer.GetHashCode(this.entries[i].Value) & int.MaxValue;
168                 index = hash % this.buckets.Length;
169                 this.hashCount++;
170                 this.entries[i].Hash = hash;
171                 this.entries[i].Next = this.buckets[index];
172                 this.buckets[index] = i;
173
174             }
175             return false;
176         }
177
178         private bool RemoveLinear(TValue value)
179         {
180             for (int i = hashCount; i < count; i++)
```

```
181             {
182                 if (this.entries[i].Hash != -1 && comparer.Equals(this.entries[i].Value, value))
183                 {
184                     this.count--;
185                     if (i < this.count)
186                     {
187                         Array.Copy(this.entries, i + 1, this.entries, i, this.count - i);
188                     }
189                     this.entries[count].Hash = -1;
190                     this.entries[count].Value = default(TValue);
191                     this.entries[count].Next = -1;
192                     this.version++;
193                     return true;
194                 }
195             }
196             return false;
197         }
198 
199         private bool RemoveFromHash(TValue value)
200         {
201             int hash = this.comparer.GetHashCode(value) & int.MaxValue;
202             int index = hash % this.buckets.Length;
203             int prev = -1;
204             for (int i = this.buckets[index]; i >= 0; i = this.entries[i].Next)
205             {
206                 if (this.entries[i].Hash == hash
207                     &&
208                     this.comparer.Equals(value, this.entries[i].Value)
209                     )
210                 {
211                     if (prev < 0)
212                     {
213                         this.buckets[index] = this.entries[i].Next;
214                     }
215                     else
216                     {
217                         this.entries[prev].Next = this.entries[i].Next;
218                     }
219                     this.entries[i].Hash = -1;
220                     this.entries[i].Next = this.freeList;
221                     this.entries[i].Value = default(TValue);
222                     this.freeList = i;
```

```
223                     this.freeCount++;
224                     this.version++;
225                     return true;
226                 }
227                 prev = i;
228             }
229             return false;
230         }
231 
232         private void AddToFree(TValue value)
233         {
234             int hash = this.comparer.GetHashCode(value) & int.MaxValue;
235             int index = hash % buckets.Length;
236             int valueIndex = freeList;
237 
238             this.freeList = this.entries[valueIndex].Next;
239             this.freeCount--;
240 
241             this.entries[valueIndex].Hash = hash;
242             this.entries[valueIndex].Next = this.buckets[index];
243             this.entries[valueIndex].Value = value;
244             this.buckets[index] = valueIndex;
245 
246         }
247 
248         public bool Contains(TValue value)
249         {
250             if (count <= hashThreshold)
251             {
252                 return SearchLinear(0,value);
253             }
254             else
255             {
256                 bool found = false;
257                 if (hashCount > 0)
258                 {
259                     found = HashSearch(value);
260                 }
261                 if (!found)
262                 {
263                     if (this.count - this.hashCount <= 4)
264                     {
```

```
265                         found = SearchLinear(hashCount, value);
266                     }
267                     else
268                     {
269                         found = HashAndSearchLinear(value);
270                     }
271                 }
272                 return found;
273             }
274         }
275 
276         private bool HashAndSearchLinear(TValue value)
277         {
278             int hash, index;
279             for (int i = hashCount; i < count; i++)
280             {
281                 hash = this.comparer.GetHashCode(this.entries[i].Value) & int.MaxValue;
282                 index = hash % this.buckets.Length;
283                 this.hashCount++;
284                 this.entries[i].Hash = hash;
285                 this.entries[i].Next = this.buckets[index];
286                 this.buckets[index] = i;
287                 if (
288                     (this.comparer.GetHashCode(value) & int.MaxValue) == hash
289                     &&
290                     this.comparer.Equals(this.entries[i].Value, value)
291                     )
292                 {
293                     return true;
294                 }
295             }
296             return false;
297         }
298 
299         private bool SearchLinear(int startIndex, TValue value)
300         {
301             for (int i = startIndex; i < this.count; i++)
302             {
303                 if (this.comparer.Equals(this.entries[i].Value, value))
304                 {
305                     return true;
306                 }
```

```
307             }
308             return false;
309         }
310 
311         private bool HashSearch(TValue value)
312         {
313             int hash = this.comparer.GetHashCode(value) & int.MaxValue;
314             int index = hash % this.buckets.Length;
315             for(int i=this.buckets[index]; i>=0; i = this.entries[i].Next)
316             {
317                 if (
318                     this.entries[i].Hash == hash
319                     &&
320                     this.comparer.Equals(this.entries[i].Value, value)
321                     )
322                 {
323                     return true;
324                 }
325             }
326             return false;
327         }
328 
329 
330         private void Expand()
331         {
332             int newLength = Helper.GetPrime(count * 2);
333 
334             int[] b = new int[newLength];
335             HashEntry[] e = new HashEntry[newLength];
336             Array.Copy(this.entries, 0, e, 0, this.count);
337             for (int i = 0; i < b.Length; i++)
338             {
339                 b[i] = -1;
340                 e[i].Hash = 0;
341                 e[i].Next = -1;
342             }
343 
344             this.buckets = b;
345             this.entries = e;
346             this.hashCount = 0;
347         }
348 
```

```
349         public void Clear()
350         {
351             for (int i = 0; i < count; i++)
352             {
353                 this.buckets[i] = -1;
354                 this.entries[i].Hash = 0;
355                 this.entries[i].Next = -1;
356                 this.entries[i].Value = default(TValue);
357             }
358             this.count = 0;
359             this.hashCount = 0;
360             this.freeCount = 0;
361             this.freeList = -1;
362         }
363 
364         public void CopyTo(TValue[] array, int arrayIndex)
365         {
366             if (array != null && array.Rank != 1)
367             {
368                 throw new ArgumentException("Arrays Rank has to be 1");
369             }
370             for (int i = 0; i < count; i++)
371             {
372                 array[arrayIndex++] = this.entries[i++].Value;
373             }
374         }
375 
376         public bool IsReadOnly
377         {
378             get { return false; }
379         }
380 
381         public IEnumerator<TValue> GetEnumerator()
382         {
383             return new LazyHashList<TValue>.Enumerator(this);
384         }
385 
386         System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
387         {
388             return new LazyHashList<TValue>.Enumerator(this);
389         }
390 
```

```
391         private bool Equals(TValue first, TValue second)
392         {
393             if (first == null)
394             {
395                 return second == null;
396             }
397             return this.comparer.Equals(first, second);
398         }
399 
400         [DebuggerDisplay("Hash = {Hash}, Next = {Next}, Value = {Value}")]
401         private struct HashEntry
402         {
403             public int Hash;
404             public TValue Value;
405             public int Next;
406         }
407 
408         public struct Enumerator : IEnumerator<TValue>
409         {
410 
411             public Enumerator(LazyHashList<TValue> lazyHashList)
412             {
413                 this.version = lazyHashList.version;
414                 this.source = lazyHashList;
415                 this.current = default(TValue);
416                 this.index = 0;
417             }
418 
419             private LazyHashList<TValue> source;
420             private int version;
421             private TValue current;
422             private int index;
423             public TValue Current
424             {
425                 get { return current; }
426             }
427 
428             public void Dispose()
429             {
430 
431             }
432 
```

```
433             object System.Collections.IEnumerator.Current
434             {
435                 get
436                 {
437                     if (this.index == 0 || this.index == this.source.count + 1)
438                     {
439                         throw new InvalidOperationException("current value is not available");
440                     }
441                     return this.Current;
442                 }
443             }
444 
445             public bool MoveNext()
446             {
447                 LazyHashList<TValue> list = this.source;
448                 if (this.version == list.version && this.index < list.count)
449                 {
450                     while (list.entries[index].Hash == -1)
451                     {
452                         index++;
453                     }
454                     if (this.index >= list.count)
455                     {
456                         this.index = this.source.count + 1;
457                         this.current = default(TValue);
458                         return false;
459                     }
460                     current = list.entries[index].Value;
461                     index++;
462                     return true;
463                 }
464                 if (this.version != list.version)
465                 {
466                     throw new InvalidOperationException(
467                         "Collection was modified during enumeration");
468                 }
469                 this.index = this.source.count + 1;
470                 this.current = default(TValue);
471                 return false;
472             }
473 
474             public void Reset()
```

```
475                 {
476                     if (this.version != this.source.version)
477                     {
478                         throw new InvalidOperationException(
479                             "Collection was modified during enumeration");
480                     }
481                     this.index = 0;
482                     this.current = default(TValue);
483                 }
484 
485             }
486 
487         }
488 }
```

Helper-luokka

```
1  internal class Helper
2  {
3      internal static readonly int[] primes = new int[]
4      {
5                  3,
6                  7,
7                  11,
8                  17,
9                  23,
10                 29,
11                 37,
12                 47,
13                 59,
14                 71,
15                 89,
16                 107,
17                 131,
18                 163,
19                 197,
20                 239,
21                 293,
22                 353,
23                 431,
24                 521,
```

```
    631,
    761,
    919,
    1103,
    1327,
    1597,
    1931,
    2333,
    2801,
    3371,
    4049,
    4861,
    5839,
    7013,
    8419,
    10103,
    12143,
    14591,
    17519,
    21023,
    25229,
    30293,
    36353,
    43627,
    52361,
    62851,
    75431,
    90523,
    108631,
    130363,
    156437,
    187751,
    225307,
    270371,
    324449,
    389357,
    467237,
    560689,
    672827,
    807403,
    968897,
    1162687,
```

```
            1395263,
            1674319,
            2009191,
            2411033,
            2893249,
            3471899,
            4166287,
            4999559,
            5999471,
            7199369
    };

    internal static int GetPrime(int min)
    {
        if (min < 0)
        {
            throw new ArgumentException("min cannot be negative.");
        }
        for (int i = 0; i < Helper.primes.Length; i++)
        {
            int num = Helper.primes[i];
            if (num >= min)
            {
                return num;
            }
        }
        for (int j = min | 1; j < 2147483647; j += 2)
        {
            if (Helper.IsPrime(j))
            {
                return j;
            }
        }
        return min;
    }

    internal static bool IsPrime(int candidate)
    {
        if ((candidate & 1) != 0)
        {
            int num = (int)Math.Sqrt((double)candidate);
            for (int i = 3; i <= num; i += 2)
```

```
109                 {
110                     if (candidate % i == 0)
111                     {
112                         return false;
113                     }
114                 }
115                 return true;
116             }
117             return candidate == 2;
118         }
119 }
```

BinarySearchTree-luokka ja TraversalOrder-enumeraatio

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4
5  namespace HH.Gradu.CSharp.DataStructures
6  {
7      public class BinarySearchTree<TKey, TValue>
8      {
9
10         public BinarySearchTree()
11         {
12             if (typeof(TKey).GetInterface("IComparable") == null)
13                 throw new Exception(string.Format(
14                     "Key type {0} is not IComparable and no comparer was given",
15                     typeof(TKey)));
16
17             this.root = null;
18             this.nodeCount = 0;
19             this.lastRemoveLeft = false;
20         }
21
22         public BinarySearchTree(IComparer<TKey> keyComparer)
23         {
24             this.root = null;
25             this.keyComparer = keyComparer;
26             this.nodeCount = 0;
27             this.lastRemoveLeft = false;
```

```
28          }
29
30          public BinarySearchTree(TKey rootKey, TValue rootValue)
31          {
32              if (typeof(TKey).GetInterface("IComparable") == null)
33                  throw new Exception(string.Format(
34                      "Key type {0} is not IComparable and no comparer was given",
35                      typeof(TKey)));
36              this.root = new BinaryTreeNode<TKey, TValue>(rootKey, rootValue);
37              this.nodeCount = 1;
38              this.lastRemoveLeft = false;
39          }
40
41          public BinarySearchTree
42          (
43          TKey rootKey,
44          TValue rootValue,
45          IComparer<TKey> keyComparer
46          )
47          {
48              this.keyComparer = keyComparer;
49              this.root = new BinaryTreeNode<TKey, TValue>(rootKey, rootValue);
50              this.nodeCount = 1;
51              this.lastRemoveLeft = false;
52          }
53
54          protected readonly IComparer<TKey> keyComparer;
55
56          protected BinaryTreeNode<TKey, TValue> root;
57
58          private bool lastRemoveLeft;
59
60          private int nodeCount;
61
62          public int NodeCount
63          {
64              get { return nodeCount; }
65          }
66
67          public int Height
68          {
69              get
```

```
70              {
71                  return GetHeight(root);
72              }
73          }
74
75          protected int Compare(TKey key1, TKey key2)
76          {
77              if (keyComparer == null)
78                  return Compare((IComparable)key1, (IComparable)key2);
79              return keyComparer.Compare(key1, key2);
80          }
81
82          protected int Compare(IComparable key1, IComparable key2)
83          {
84              return key1.CompareTo(key2);
85          }
86
87          public bool Add(TKey key, TValue value)
88          {
89              bool newNodeAdded = false;
90              BinaryTreeNode<TKey,TValue> nodeForBalancing = null;
91              if (root == null)
92              {
93                  root = new BinaryTreeNode<TKey, TValue>(key, value);
94                  nodeForBalancing = root;
95                  newNodeAdded = true;
96              }
97              else
98              {
99                  BinaryTreeNode<TKey, TValue> node = root;
100                 int compareValue;
101                 while (true)
102                 {
103                     compareValue = Compare(key, node.Key);
104                     if (compareValue == 0)
105                     {
106                         node.Value = value;
107                         nodeForBalancing = node;
108                         break;
109                     }
110                     else if (compareValue < 0)
111                     {
```

```
112                         if (node.Left == null)
113                         {
114                             node.Left = new BinaryTreeNode<TKey, TValue>(key, value);
115                             node.Left.Parent = node;
116                             nodeForBalancing = node;
117                             newNodeAdded = true;
118                             break;
119                         }
120                         else
121                         {
122                             node = node.Left;
123                         }
124                     }
125                     else
126                     {
127                         if (node.Right == null)
128                         {
129                             node.Right = new BinaryTreeNode<TKey, TValue>(key, value);
130                             node.Right.Parent = node;
131                             nodeForBalancing = node;
132                             newNodeAdded = true;
133                             break;
134                         }
135                         else
136                         {
137                             node = node.Right;
138                         }
139                     }
140                 }
141
142             }
143             if (newNodeAdded)
144                 ++nodeCount;
145             AfterAdd(newNodeAdded, nodeForBalancing);
146             return newNodeAdded;
147         }
148
149         protected virtual void AfterAdd
150         (
151         bool newNodeAdded,
152         BinaryTreeNode<TKey, TValue> nodeForBalancing
153         )
```

```
154         {
155
156         }
157
158         public IEnumerable<TValue> GetValues(TraversalOrder order)
159         {
160             if (root == null)
161                 return new TValue[0];
162
163             return root.GetValues(order, nodeCount);
164         }
165
166         public IEnumerable<KeyValuePair<TKey, TValue>> GetItems(TraversalOrder order)
167         {
168             if(root == null)
169                 return new KeyValuePair<TKey, TValue>[0];
170
171             return root.GetItems(order, nodeCount);
172         }
173
174         public bool ContainsKey(TKey key)
175         {
176             BinaryTreeNode<TKey, TValue> node = root;
177             int compareValue;
178             while (node != null)
179             {
180                 compareValue = Compare(key, node.Key);
181                 if (compareValue == 0)
182                     return true;
183                 else if (compareValue < 0)
184                     node = node.Left;
185                 else
186                     node = node.Right;
187             }
188
189             return false;
190         }
191
192         protected virtual int GetHeight(BinaryTreeNode<TKey, TValue> node)
193         {
194             if (node == null)
195                 return 0;
```

```
196             int leftHeight = GetHeight(node.Left);
197             int rightHeight = GetHeight(node.Right);
198 
199             return Math.Max(leftHeight, rightHeight) + 1;
200 
201         }
202 
203         public bool Remove(TKey key)
204         {
205             BinaryTreeNode<TKey, TValue> node = root;
206             BinaryTreeNode<TKey, TValue> parent = null;
207             BinaryTreeNode<TKey, TValue> nodeForBalancing = null;
208             //false left, true right
209             bool parentChildDir = false;
210             TValue removedValue = default(TValue);
211             int compareValue;
212             bool removed = false;
213             while (node != null)
214             {
215                 compareValue = Compare(key, node.Key);
216                 if (compareValue == 0)
217                 {
218                     //Found node to remove
219                     removed = true;
220                     removedValue = node.Value;
221                     //Removing a leaf node
222                     if (node.Left == null && node.Right == null)
223                     {
224                         RemoveNodeWithoutChildren(node, parent, parentChildDir);
225                         nodeForBalancing = node;
226                     }
227                     //node has only one child
228                     else if ((node.Left != null) ^ (node.Right != null))
229                     {
230                         RemoveNodeWithOneChild(node, parent, parentChildDir);
231                         nodeForBalancing = node;
232                     }
233                     //Node has both children
234                     else
235                     {
236                         nodeForBalancing = RemoveNodeWithBothChildren(node);
237                     }
```

```
238                     break;
239                 }
240                 else if (compareValue < 0)
241                 {
242                     parent = node;
243                     node = node.Left;
244                     parentChildDir = false;
245                 }
246                 else
247                 {
248                     parent = node;
249                     node = node.Right;
250                     parentChildDir = true;
251                 }
252             }
253             if (removed)
254                 --nodeCount;
255             AfterRemove(removed, nodeForBalancing);
256             return removed;
257         }
258 
259         private BinaryTreeNode<TKey, TValue> RemoveNodeWithBothChildren
260         (
261         BinaryTreeNode<TKey, TValue> node
262         )
263         {
264             BinaryTreeNode<TKey, TValue> replacement = null;
265             BinaryTreeNode<TKey, TValue> replacementParent = null;
266             BinaryTreeNode<TKey, TValue> replacementChild = null;
267             //Alternate between largest of left subtree and smallest of right subtree replacement
268             //This still doesn't guarantee balancing but is better than nothing
269             if (lastRemoveLeft)
270             {
271                 //Get left most from right subtree
272 
273                 replacement = node.Right;
274                 replacementParent = node;
275                 replacementChild = replacement.Right;
276                 while (replacement.Left != null)
277                 {
278                     replacementParent = replacement;
279                     replacement = replacement.Left;
```

```
280                     replacementChild = replacement.Right;
281                 }
282                 if (replacementParent == root)
283                     replacementParent.Right = replacementChild;
284                 else
285                     replacementParent.Left = replacementChild;
286                 if(replacementChild != null)
287                     replacementChild.Parent = replacementParent;
288                 lastRemoveLeft = false;
289             }
290             else
291             {
292                 //Get right most from left subtree
293                 replacement = node.Left;
294                 replacementParent = node;
295                 replacementChild = replacement.Left;
296                 while (replacement.Right != null)
297                 {
298                     replacementParent = replacement;
299                     replacement = replacement.Right;
300                     replacementChild = replacement.Left;
301                 }
302                 if (replacementParent == root)
303                     replacementParent.Left = replacementChild;
304                 else
305                     replacementParent.Right = replacementChild;
306                 if(replacementChild != null)
307                     replacementChild.Parent = replacementParent;
308                 lastRemoveLeft = true;
309             }
310             node.Value = replacement.Value;
311             node.Key = replacement.Key;
312             replacement.Left = replacement.Right = null;
313 
314             return replacement;
315         }
316 
317         private void RemoveNodeWithOneChild
318         (
319         BinaryTreeNode<TKey, TValue> node,
320         BinaryTreeNode<TKey, TValue> parent,
321         bool parentChildDir
```

```
322         )
323         {
324             var child = node.Left ?? node.Right;
325             if (parent == null)
326             {
327                 //Removing root
328                 root = child;
329                 child.Parent = null;
330             }
331             else if (parentChildDir)
332             {
333                 parent.Right = child;
334                 child.Parent = parent;
335             }
336             else
337             {
338                 parent.Left = child;
339                 child.Parent = parent;
340             }
341 
342             node.Left = node.Right = null;
343         }
344 
345         private void RemoveNodeWithoutChildren
346         (
347         BinaryTreeNode<TKey, TValue> node,
348         BinaryTreeNode<TKey, TValue> parent,
349         bool parentChildDir
350         )
351         {
352             if (parent == null)
353             {
354                 //removing root
355                 root = null;
356             }
357             else
358             {
359                 //remove parent's right child
360                 if (parentChildDir)
361                     parent.Right = null;
362                 //remove parent's left child
363                 else
```

```
364                     parent.Left = null;
365                 }
366             }
367
368             protected virtual void AfterRemove
369             (
370             bool removed,
371             BinaryTreeNode<TKey, TValue> nodeForBalancing
372             )
373             {
374
375             }
376
377             public void Clear()
378             {
379                 root = null;
380                 nodeCount = 0;
381             }
382
383             public void Balance()
384             {
385                 var inorder = GetItems(TraversalOrder.InOrder).ToArray();
386                 root = null;
387                 AddChildren(inorder);
388             }
389
390             private void AddChildren(KeyValuePair<TKey,TValue>[] items)
391             {
392                 var center = items.Length / 2;
393                 Add(items[center].Key, items[center].Value);
394
395                 if (center != 0)
396                 {
397                     var left = new KeyValuePair<TKey, TValue>[center];
398                     Array.Copy(items, 0, left, 0, center);
399                     AddChildren(left);
400                 }
401                 if (center != items.Length - 1)
402                 {
403                     var right = new KeyValuePair<TKey, TValue>[items.Length - (center + 1)];
404                     Array.Copy(items, center + 1, right, 0, right.Length);
405                     AddChildren(right);
```

```
406                 }
407
408
409             }
410
411             protected void RotateRight(BinaryTreeNode<TKey, TValue> node)
412             {
413                 BinaryTreeNode<TKey, TValue> parent = node.Parent;
414                 bool isRightChild = parent != null ? node == parent.Right : false;
415                 var pivot = node.Left;
416                 node.Left = pivot.Right;
417                 if(pivot.Right != null)
418                     pivot.Right.Parent = node;
419                 pivot.Right = node;
420
421                 node.Parent = pivot;
422                 pivot.Parent = parent;
423
424                 //node was root
425                 if (parent == null)
426                 {
427                     root = pivot;
428                     root.Parent = null;
429                 }
430                 //node was parent's right child
431                 else if (isRightChild)
432                 {
433                     parent.Right = pivot;
434                 }
435                 //node was parent's left child
436                 else
437                 {
438                     parent.Left = pivot;
439                 }
440             }
441
442             protected void RotateLeft(BinaryTreeNode<TKey, TValue> node)
443             {
444                 BinaryTreeNode<TKey, TValue> parent = node.Parent;
445                 bool isRightChild = parent != null ? node == parent.Right : false;
446                 var pivot = node.Right;
447                 node.Right = pivot.Left;
```

```
448             if(pivot.Left != null)
449                 pivot.Left.Parent = node;
450             pivot.Left = node;
451 
452             node.Parent = pivot;
453             pivot.Parent = parent;
454 
455             //node was root
456             if (parent == null)
457             {
458                 root = pivot;
459                 root.Parent = null;
460             }
461             //node was parent's right child
462             else if (isRightChild)
463             {
464                 parent.Right = pivot;
465             }
466             //node was parent's left child
467             else
468             {
469                 parent.Left = pivot;
470             }
471 
472         }
473 
474         #region Node class
475         protected class BinaryTreeNode<TNodeKey, TNodeValue>
476         {
477 
478             internal BinaryTreeNode(TNodeKey key, TNodeValue value)
479             {
480                 this.key = key;
481                 this._value = value;
482             }
483 
484             private TNodeKey key;
485             internal TNodeKey Key
486             {
487                 get
488                 {
489                     return key;
```

```
490             }
491             set
492             {
493                 key = value;
494             }
495         }
496 
497 
498         private TNodeValue _value;
499         internal TNodeValue Value
500         {
501             get
502             {
503                 return _value;
504             }
505             set
506             {
507                 _value = value;
508             }
509         }
510 
511         private BinaryTreeNode<TNodeKey, TNodeValue> left;
512         internal BinaryTreeNode<TNodeKey, TNodeValue> Left
513         {
514             get
515             {
516                 return left;
517             }
518             set
519             {
520                 left = value;
521             }
522         }
523 
524         private BinaryTreeNode<TNodeKey, TNodeValue> right;
525         internal BinaryTreeNode<TNodeKey, TNodeValue> Right
526         {
527             get
528             {
529                 return right;
530             }
531             set
```

```
532                 {
533                     right = value;
534                 }
535             }
536 
537             private BinaryTreeNode<TNodeKey, TNodeValue> parent;
538             internal BinaryTreeNode<TNodeKey, TNodeValue> Parent
539             {
540                 get { return parent; }
541                 set { parent = value; }
542             }
543 
544             internal IEnumerable<KeyValuePair<TNodeKey, TNodeValue>> GetItems
545             (
546             TraversalOrder order,
547             int nodeCount
548             )
549             {
550                 List<KeyValuePair<TNodeKey, TNodeValue>> result =
551                     new List<KeyValuePair<TNodeKey, TNodeValue>>(nodeCount);
552                 switch (order)
553                 {
554                     case TraversalOrder.PreOrder:
555                         GetItemsPreOrder(result);
556                         break;
557                     case TraversalOrder.InOrder:
558                         GetItemsInOrder(result);
559                         break;
560                     case TraversalOrder.PostOrder:
561                         GetItemsPostOrder(result);
562                         break;
563                 }
564                 return result;
565             }
566 
567             private void GetItemsPostOrder(List<KeyValuePair<TNodeKey, TNodeValue>> result)
568             {
569                 if (left != null)
570                     left.GetItemsPostOrder(result);
571                 if (right != null)
572                     right.GetItemsPostOrder(result);
573                 result.Add(new KeyValuePair<TNodeKey, TNodeValue>(key, _value));
```

```
574             }
575
576             private void GetItemsInOrder(List<KeyValuePair<TNodeKey, TNodeValue>> result)
577             {
578                 if (left != null)
579                     left.GetItemsInOrder(result);
580                 result.Add(new KeyValuePair<TNodeKey, TNodeValue>(key, _value));
581                 if (right != null)
582                     right.GetItemsInOrder(result);
583             }
584
585             private void GetItemsPreOrder(List<KeyValuePair<TNodeKey, TNodeValue>> result)
586             {
587                 result.Add(new KeyValuePair<TNodeKey, TNodeValue>(key, _value));
588                 if (left != null)
589                     left.GetItemsPreOrder(result);
590                 if (right != null)
591                     right.GetItemsPreOrder(result);
592             }
593
594             internal IEnumerable<TNodeValue> GetValues(TraversalOrder order, int nodeCount)
595             {
596                 List<TNodeValue> result = new List<TNodeValue>(nodeCount);
597                 switch (order)
598                 {
599                     case TraversalOrder.PreOrder:
600                         GetValuesPreOrder(result);
601                         break;
602                     case TraversalOrder.InOrder:
603                         GetValueInOrder(result);
604                         break;
605                     case TraversalOrder.PostOrder:
606                         GetValuesPostOrder(result);
607                         break;
608                 }
609                 return result;
610             }
611
612
613             private void GetValuesPreOrder(List<TNodeValue> resultList)
614             {
615                 resultList.Add(_value);
```

```
616                     if (left != null)
617                         left.GetValuesPreOrder(resultList);
618                     if (right != null)
619                         right.GetValuesPreOrder(resultList);
620                 }
621 
622                 private void GetValueInOrder(List<TNodeValue> resultList)
623                 {
624                     if (left != null)
625                         left.GetValueInOrder(resultList);
626                     resultList.Add(_value);
627                     if (right != null)
628                         right.GetValueInOrder(resultList);
629                 }
630 
631                 private void GetValuesPostOrder(List<TNodeValue> resultList)
632                 {
633                     if (left != null)
634                         left.GetValuesPostOrder(resultList);
635                     if (right != null)
636                         right.GetValuesPostOrder(resultList);
637                     resultList.Add(_value);
638                 }
639 
640             }
641             #endregion
642     }
643 
644     public enum TraversalOrder : byte
645     {
646         PreOrder = 0,
647         InOrder = 1,
648         PostOrder = 2
649     }
650 }
```

ActionMeasurer- ja MeasurementResult-luokat

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
```

```
4  using System.Diagnostics;
5  using System.Threading;
6
7  namespace HH.Gradu.CSharp.Measurement
8  {
9      public class ActionMeasurer
10     {
11         public MeasurementResult Measure<TCollection,TValue>(
12             string measurementName,
13             int actionIterations,
14             Action<TCollection, TValue> action,
15             Func<TCollection> newCollection,
16             IEnumerable<TValue> testSet,
17             CancellationToken cancelToken,
18             bool newCollectionAfterEveryAction = false
19             )
20         {
21             GC.Collect();
22             GC.WaitForFullGCComplete(Timeout.Infinite);
23             int iterationCounter = 0;
24             Stopwatch watch = new Stopwatch();
25             TimeSpan span;
26             TimeSpan total;
27             TCollection collection;
28             watch.Start();
29             while (iterationCounter++ < actionIterations)
30             {
31                 if (cancelToken.IsCancellationRequested)
32                     break;
33                 collection = newCollection();
34
35                 foreach (var item in testSet)
36                 {
37                     if (cancelToken.IsCancellationRequested)
38                         break;
39
40                     action(collection, item);
41
42                     if (newCollectionAfterEveryAction)
43                         collection = newCollection();
44                 }
45
```

```
46              }
47              watch.Stop();
48              total = watch.Elapsed;
49              watch.Reset();
50
51              GC.Collect();
52              GC.WaitForFullGCComplete(Timeout.Infinite);
53
54              iterationCounter = 0;
55              Action<TCollection, TValue> dummyAction = (col, val) => { };
56              watch.Start();
57              while (iterationCounter++ < actionIterations)
58              {
59                  if (cancelToken.IsCancellationRequested)
60                      break;
61                  collection = newCollection();
62
63                  foreach (var item in testSet)
64                  {
65                      if (cancelToken.IsCancellationRequested)
66                          break;
67
68                      dummyAction(collection, item);
69
70                      if (newCollectionAfterEveryAction)
71                          collection = newCollection();
72                  }
73
74              }
75              watch.Stop();
76              span = total − watch.Elapsed;
77              TimeSpan avg = new TimeSpan(span.Ticks / actionIterations);
78              return new MeasurementResult(
79                          measurementName,
80                          typeof(TCollection),
81                          span,
82                          avg,
83                          actionIterations,
84                          testSet.Count());
85          }
86
87      }
```

```
88
89      public class MeasurementResult
90      {
91
92          public MeasurementResult(
93              string name,
94              Type type,
95              TimeSpan total,
96              TimeSpan avg,
97              int numberOfIterations,
98              int actionsPerIteration
99              )
100         {
101             this.name = name;
102             this.collectionType = type;
103             this.totalTime = total;
104             this.avgTime = avg;
105             this.iterations = numberOfIterations;
106             this.actionsPerIteration = actionsPerIteration;
107             this.avgMillisecondsPerOperation =
108             (decimal)((decimal)total.TotalMilliseconds
109                         /
110                         ((decimal)numberOfIterations * (decimal)actionsPerIteration));
111         }
112
113         private Type collectionType;
114
115         public Type CollectionType
116         {
117             get { return collectionType; }
118             set { collectionType = value; }
119         }
120
121
122         private string name;
123
124         public string Name
125         {
126             get { return name; }
127         }
128
129         private TimeSpan totalTime;
```

```
130
131         public TimeSpan TotalTime
132         {
133             get { return totalTime; }
134         }
135
136         private TimeSpan avgTime;
137
138         public TimeSpan AvgTime
139         {
140             get { return avgTime; }
141         }
142
143         private decimal avgMillisecondsPerOperation;
144
145         public decimal AvgMillisecondsPerOperation
146         {
147             get { return avgMillisecondsPerOperation; }
148         }
149
150         private int iterations;
151
152         public int Iterations
153         {
154             get { return iterations; }
155         }
156
157         private int actionsPerIteration;
158
159         public int ActionsPerIteration
160         {
161             get { return actionsPerIteration; }
162             set { actionsPerIteration = value; }
163         }
164
165
166     }
167 }
```