Karelia-ammattikorkeakoulu
Tutkintonimike, Tradenomi (AMK)

Tietorakenteiden ja ohjelmointirakenteiden suorituskykyanalyysi C#-kielessä

Tessa Helenius

Opinnäytetyö, toukokuu 2025

www.karelia.fi

Karelia AMMATTIKORKEAKOULU	**OPINNÄYTETYÖ** **Toukokuu 2025** **Tietojenkäsittelyn koulutus** Tikkarinne 9 80200 JOENSUU +358 13 260 600

Tekijä(t)
Tessa Helenius

Nimeke
Tietorakenteiden ja ohjelmointirakenteiden suorituskykyanalyysi C#-kielessä

Tiivistelmä

Tässä opinnäytetyössä tutkittiin C#-ohjelmointikielen yleisesti käytettyjen tietorakenteiden ja ohjelmointirakenteiden suorituskykyä. Tavoitteena oli selvittää, kuinka erilaiset rakenteet vaikuttavat ohjelman tehokkuuteen. Tutkimuksen lähtökohtana oli lisätä ymmärrystä siitä, miten oikean rakenteen valinta voi parantaa ohjelman suorituskykyä.

Tutkimus toteutettiin laadullisena tapaustutkimuksena, jota täydennettiin käytännön suorituskykymittauksilla. Mittaustyökaluina käytettiin BenchmarkDotNet-kirjastoa sekä Visual Studio Performance Profileria. Testit suunniteltiin yksinkertaisiksi ja toistettaviksi, jotta eri rakenteiden vertaileminen olisi selkeää. Teoriapohja muodostui aiheeseen liittyvästä kirjallisuudesta, artikkeleista ja aiemmista tutkimuksista, joissa käsiteltiin tietorakenteiden ja ohjelmointirakenteiden suorituskykyä sekä suorituskyvyn mittaamiseen vaikuttavia tekijöitä.

Tulosten perusteella havaittiin, että tietorakenteiden rakenteelliset erot vaikuttavat niiden suorituskykyyn. Sanakirja oli nopein hakutoiminnoissa, mutta taulukko oli tehokkain lisäyksissä ja poistoissa. Useimmin esiintyvän ehdon järjestyksellä ei ollut vaikutusta suorituskykyyn, mutta if-else oli rakenteena kevyempi kuin switch-case. LINQ osoittautui hyväksi vaihtoehdoksi silmukoille tietyissä operaatioissa. Rekursio oli rakenteeltaan hieman raskas, mutta binääripuun haussa sen suorituskyky oli lähes yhtä hyvä kuin iteratiivisella ratkaisulla. Tulokset osoittavat, että rakenteiden valinnalla on merkitystä suorituskykyyn ja suorituskykyä kannattaa optimoida käytännön testien kautta, eikä luottaa pelkkään teoriaan. Jatkokehitysideana tutkimusta voisi laajentaa koskemaan esimerkiksi .NET-ympäristön sisäisiä optimointeja tai tutkimusta voisi syventää vertailemalla rakenteiden suorituskykyä täysin eri ohjelmointikielellä.

Kieli suomi	Sivuja 54 Liitteet 6 Liitesivumäärä 7

Asiasanat
C#, tietorakenteet, ohjelmointi, suorituskyky

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
May 2025
Degree Programme in Business Information Technology

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600

Author (s)
Tessa Helenius

Title
Performance Analysis of Data Structures and Programming Structures in C#

Abstract

The aim of this thesis was to study the performance of commonly used data structures and programming structures in the C# programming language. The purpose of this study was to determine how structural choices affect program efficiency and to increase understanding of their practical impact.

The study was conducted as a qualitative case study supported by empirical performance measurements. BenchmarkDotNet and Visual Studio Performance Profiler were used as measurement tools. The tests were designed to be simple and repeatable to allow clear comparisons. The theoretical background of the study is based on literature, articles, and previous studies on programming performance and measurement.

The results provide some support for the idea that structural differences significantly impact performance. For example, dictionaries were the fastest in search-operations, while arrays performed best in insert- and delete-operations. The order of the most common condition had no measurable effect in conditional loops, though if-else was lighter than switch-case. LINQ proved effective in specific operations, and while recursion was slightly heavier, it performed nearly as well as iteration in binary tree searches.

In conclusion, these results suggest that structural choices matter, and that performance should be optimized through practical testing rather than theoretical assumptions. Further research could explore internal optimizations of the .NET environment or compare performance across other programming languages.

Language
Finnish

Pages 54
Appendices 6
Pages of Appendices 7

Keywords

C#, data structures, programming, performance

Sisältö

1 Johdanto 3
2 Opinnäytetyössä käytetyt menetelmät 4
2.1 Tutkimusmenetelmät 4
2.2 Aineiston analyysi 5
2.3 Tutkimuksen teoreettinen viitekehys 8
3 Tietorakenteet 11
3.1 Tietorakenteiden yleiskatsaus 11
3.2 Lista 12
3.3 Sanakirja 13
3.4 Taulukko 14
3.5 Muita tietorakenteita 14
4 Ohjelmointirakenteet ja niiden suorituskyky 17
4.1 Ehtorakenteet 17
4.1.1 If-else 17
4.1.2 Switch-case 18
4.2 Toistorakenteet 18
4.2.1 Silmukkapohjaiset rakenteet 19
4.2.2 Rekursio 20
4.3 LINQ 22
5 Suorituskyvyn mittaamisen perusteet 24
5.1 Mitä suorituskyvyn mittaaminen tarkoittaa 24
5.2 Mittaustekniikat ja työkalut 24
5.2.1 Visual Studio Performance Profiler 25
5.2.2 BenchmarkDotNet 26
5.3 Suorituskyvyn mittaamiseen vaikuttavat tekijät 27
5.4 Mittaustulosten luotettavuus ja virhemahdollisuudet 29
6 Tulokset 31
6.1 Tietorakenteet 31
6.2 Silmukat ja LINQ 34
6.3 Useimmin esiintyvän ehdon järjestyksen vaikutus 36
6.4 Ehtorakenteet ja rekursio 37
6.5 Suorituskyvyn mittaaminen 39
7 Pohdinta 40
7.1 Tutkimustulosten ja teorian vertailu 40
7.1.1 Sanakirja 40
7.1.2 Taulukko 41
7.1.3 Lista 41
7.1.4 Silmukat ja LINQ 42
7.1.5 Useimmin esiintyvän ehdon järjestyksen vaikutus 44
7.1.6 Ehtorakenteet ja rekursio 45
7.2 Suorituskyvyn mittaamiseen vaikuttavat tekijät 47
7.3 Luotettavuus ja eettisyys 48
7.4 Jatkokehitysideat 50
7.5 Tutkimuksen hyödynnettävyys 52
7.6 Oman osaamisen kehittyminen 52
Lähteet 54

Liitteet

Liite 1 Aineistonhallintasuunnitelma
Liite 2 Testitulokset tietorakenteet
Liite 3 Testitulokset silmukat ja LINQ
Liite 4 Testitulokset useimmin esiintyvän ehdon järjestyksen vaikutus
Liite 5 Testitulokset if-else vs. switch-case vs. rekursio
Liite 6 Testitiedostot GitHub-linkki

1 Johdanto

Tämän opinnäytetyön tavoitteena on tutkia, miten erilaiset tietorakenteet ja ohjelmointirakenteet vaikuttavat sovellusten suorituskykyyn C#-ohjelmointikielessä. Työssä keskitytään erityisesti tietorakenteiden, ehtorakenteiden sekä perinteisten silmukkarakenteiden suorituskyvyn luotettavaan mittaamiseen. Tavoitteena on tarjota käytännönläheinen ja selkeä analyysi, joka auttaa ohjelmistokehittäjiä tekemään tietoon perustuvia valintoja. Opinnäytetyön aihe on Karelia-ammattikorkeakoulun tarjoama.

Tutkimus vastaa seuraaviin kysymyksiin: Miten erilaiset tietorakenteet ja ohjelmointivalinnat vaikuttavat sovellusten suorituskykyyn C#-kielessä, onko ehtorakenteissa useimmin esiintyvän ehdon järjestyksellä vaikutusta suorituskykyyn, miten suorituskykyä voidaan luotettavasti mitata ja mitä seikkoja tulee ottaa huomioon suorituskyvyn mittaamisessa. Työn lopputuloksena syntyy yksityiskohtainen analyysi, jossa esitellään suorituskykytestien tulokset ja suositukset tietorakenteiden ja ohjelmointirakenteiden valintaan.

Tämä tutkimus keskittyy erityisesti C#-ohjelmointikieleen ja sen tarjoamiin tietorakenteisiin ja ohjelmointirakenteisiin. Tutkimus on rajattu käsittämään ehtorakenteista if-else- ja switch-case -rakenteet sekä rekursiiviset ratkaisut. Tietorakenteista analysoidaan listoja, sanakirjoja ja taulukoita, mutta teoriaosuudessa sivutaan myös muutamia muita yleisiä tietorakenteita lyhyesti. Lisäksi tutkimuksessa analysoidaan LINQ-kyselyiden suorituskykyä verrattuna perinteisiin silmukkapohjaisiin rakenteisiin. Suorituskykyä mitataan tässä tutkimuksessa ainoastaan Windows-käyttöjärjestelmällä ja .NET 9.0 -versiolla. Suorituskykytestien ja tulosten analysoinnin lisäksi tutkimus sisältää kirjallisuuskatsauksen, joka toimii tutkimuksen teoriapohjana. Kirjallisuuskatsaus tarjoaa kattavan läpileikkauksen tutkimuksessa käsiteltäviin aiheisiin ja aiempiin tutkimuksiin aiheesta.

2 Opinnäytetyössä käytetyt menetelmät

2.1 Tutkimusmenetelmät

Tämä opinnäytetyö toteutettiin laadullisena tutkimuksena. Laadullinen tutkimus vastaa usein kysymyksiin, kuten miten, millainen tai miksi, ja näiden kysymysten avulla pyritään kasvattamaan ymmärrystä tutkimuksen kohteena olevasta ilmiöstä. (Heikkilä 2014, 15.) Ennen kuin kvalitatiivinen tutkimus toteutetaan, tutkijan tulisi miettiä miltä kannalta asiaa tai ilmiötä halutaan tarkastella. Tutkimuksen alkuvaiheessa tulisi määritellä, tutkitaanko asiaa tai ilmiötä kokemuksien vai käsityksien merkitysten valossa. Kokemukset ovat aina jokaisen omakohtaisesti koettuja asioita, kun taas käsitykset ovat usein ikään kuin stereotyyppisiä ajatusmalleja tutkittavasta ilmiöstä. (Vilkka 2021, 94.)

Laadullista tutkimusta voidaan lähestyä useilla eri tavoilla. Laadullisen tutkimuksen lähestymistapoja ovat esimerkiksi tapaustutkimus ja toimintatutkimus. Tutkimuksellinen lähestymistapa kuvaa niitä tapoja, joiden avulla tutkimusongelma yritetään saada ratkaistua. Tutkimuksellinen lähestymistapa ei ole aineistonkeruumenetelmä, vaan tutkimuksellinen ote, jonka perusteella tutkimusta lähdetään toteuttamaan. Tutkimuksellinen lähestymistapa tulisi valita niin, että se palvelee mahdollisimman hyvin tutkimuksen tarkoituksenmukaisuutta. (Puusa & Juuti 2020, 195.)

Tutkimuksellinen lähestymistapa tälle opinnäytetyölle on tapaustutkimus, tarkemmin ilmaistuna rajatun joukon tapaustutkimus, joka keskittyy etukäteen rajattuihin tapauksiin. Koska tutkittavalle kohteelle voidaan asettaa selkeä ja tarkka rajaus, tapaustutkimuksen lähestymistapa sopii valittuun opinnäytetyöhön hyvin. (Vilkka 2021, 125.) Tässä tutkimuksessa jokainen eri testiskenaario on oma tapauksensa. Tutkimuksessa voidaan esimerkiksi tarkastella, miten tietojen haku, lisäys ja poisto toimii eri tietorakenteilla tai

ohjelmointirakenteilla suorituskyvyn näkökulmasta, ja näin ollen jokainen operaatio on erillinen tapaus.

2.2 Aineiston analyysi

Teoreettiseen tietoperustaan tarvittava aineisto hankittiin eri tietokannoista, kirjallisuudesta, tutkimuksista ja artikkeleista. Aineiston hankinnassa keskitytään erityisesti siihen, että kaikki käytettävä kirjallisuus, tutkimukset ja artikkelit ovat luotettavia, ajantasaisia ja relevantteja tutkimuksen kannalta. Lähdeaineiston pääpaino on kirjallisuudessa ja aiemmissa tutkimuksissa; verkkoaineistojen kuten artikkelien, blogien ja muiden tekstien käyttö pyrittiin pitämään vähäisenä. Tutkimuksessa käytettiin mahdollisimman tuoreita lähteitä, mutta vanhempienkin tutkimusten ja kirjallisuuden käyttö koettiin perustelluksi, jos asiasisältö oli edelleen luotettavaa ja ajantasaista.

Tutkimuksessa suoritettiin erilaisia testejä ja mittauksia suorituskyvyn vertailemiseksi BenchmarkDotNet-kirjaston ja Visual Studion Performance Profilerin avulla. Näistä suorituskykytesteistä saatujen tulosten avulla voidaan vastata laadulliselle tutkimukselle tyypillisiin miksi-kysymyksiin, kuten miksi jokin tietorakenne on suorituskyvyltään tehokkaampi kuin toinen. Laadullisen tutkimuksen yksi pääpiirre on se, että tarkoituksena ei ole ainoastaan kertoa, että X on tehokkaampi ratkaisu kuin Y, vaan myös kertoa ja pohtia, miksi näin on, ja mitkä seikat tähän tulokseen vaikuttavat. (Vilkka 2021, 94.)

Testit suoritettiin Windows 11 Home (versio 24H2) -käyttöjärjestelmällä varustetulla kannettavalla, jonka keskeiset tekniset tiedot ja vertailu tyypilliseen nykyaikaiseen kehityskannettavaan on esitetty taulukossa 1.

Taulukko 1. Testikoneen kokoonpano vs. tyypillinen kehityskone. (Yadav 2024.)

Ominaisuus	Testikone	Tyypillinen kehityskone
Käyttöjärjestelmä	Windows 11 Home 242H	Windows 11,macOS, Linux
Suoritin	Intel® Core™ i5-11400H (6 ydintä, 12 säiettä, max 4.5 GHz)	min. Intel i5 / AMD Ryzen 5
Muisti (RAM)	16 Gt DDR4-3200 MHz	min. 16 Gt – suositus 32 Gt
Tallennus	512 Gt + 1 Tt M.2 NVMe PCIe 3.0 SSD	min. 512 Gt – suositus 1 Tt
Visual Studio	17.13.4	Uusin versio
BenchmarkDotNet	0.14.0	Uusin versio
.NET SDK	9.0.101	Uusin versio
.NETCore.App	9.0	Uusin versio

Testikone vastaa kohtuullisen hyvin nykyaikaisen kehityskoneen perustasoa. Valituilla .NET SDK- ja .NETCore-App -versioilla on merkitystä sen verran, että ne sisältävät uusimmat .NET-ympäristön optimoinnit, kuten suorituskykyparannukset ja parannetun roskienkeruun hallinnan, jotka voivat vaikuttaa mittaustuloksiin erityisesti muistinkäytön ja CPU-kuormituksen osalta. Testikoneen prosessori täyttää suositellut minimivaatimukset, ja 16 gigatavun RAM-muisti sekä nopea NVMe SSD -tallennus tekevät siitä kohtuullisen vaihtoehdon ohjelmistokehityksen ja testauksen tarpeisiin. Uusissa kehityskoneissa suositellaan kuitenkin usein jo 32 gigatavun kokoonpanoja RAM-muistin osalta. Yleisissä kehityskoneissa nähdään nykyään enemmän DDR5-muisteja, mutta erot ovat silti marginaalisia tämän tutkimuksen suorituskykytesteissä. (Yadav 2024.) Käytännössä laitteiston ei arvioida aiheuttaneen merkittäviä vääristymiä testituloksiin.

Suorituskykytestejä varten tarvittava lähdekoodi tuotettiin itse Visual Studiossa C#-kielellä, eikä se sisällä salassa pidettävää tai arkaluontoista materiaalia. Saatuja tuloksia verrattiin kirjallisuudesta ja aiemmista tutkimuksista saatuihin havaintoihin. Testausprosessi alkoi testien suunnittelulla ja koodin toimivuuden testauksella. Testejä tehtiin ensin jokaiseen tietorakenteeseen ja ohjelmointirakenteeseen muutamia erilaisia vaihtoehtoja, joista valittiin testauksen kannalta oleellisin lopullinen testidata. Testidataa muokattiin tietorakenteiden osalta vielä kerran testien suorittamisen jälkeen, sillä

koodirakenteeseen oli jäänyt turhaa kopiointia, joka vaikutti suorituskykyyn negatiivisesti. Korjausten jälkeen tietorakenteiden testit suoritettiin uudelleen kaksi kertaa.

Molemmat, sekä BenchmarkDotNet- että Performance Profiler -testit, suoritettiin lopullisissa mittauksissa kaksi kertaa, kahtena eri päivänä. Testien ajon aikana tietokoneen verkkoyhteys katkaistiin, ja kaikki ylimääräiset taustaprosessit, kuten virustorjunta ja palomuurit suljettiin. BenchmarkDotNet-testit keskittyivät muistivarauksen ja suoritusajan mittaamiseen, ja Performance Profilerilla mitattiin pääasiassa prosessorin kuormitusta.

Tässä tutkimuksessa keskityttiin tarkastelemaan prosessorin kuormitusta ensisijaisesti tutkimalla Self CPU -arvoa, koska se kuvaa suoraan, kuinka paljon prosessorin aikaa yksittäinen metodi käyttää ilman siihen liittyviä alikutsuja. Total CPU sen sijaan sisältää myös muiden kutsuttujen metodien kuluttaman ajan, joten se voi antaa hieman laajemman mutta vähemmän tarkan kuvan suorituskyvystä. (Microsoft Learn 2025.)

Testattavat operaatiot ja aineistot

Tietorakenteiden testeissä käytettiin miljoonan alkion taulukkoa, listaa ja sanakirjaa, joihin suoritettiin alkion haku, uuden alkion lisäys sekä poisto alusta, keskeltä ja lopusta. Silmukka- ja LINQ-testeissä käytettiin kolmen miljoonan alkion listaa. Listalta laskettiin parillisten lukujen määrä, etsittiin suurin arvo ja laskettiin kaikkien alkioiden summa eri silmukkarakenteilla (for, foreach, while, do-while) sekä LINQ-kyselyllä.

Ehtorakenteiden ja rekursion testeissä tarkasteltiin kahta erillistä kokonaisuutta. Ensimmäisessä skenaariossa miljoonalle satunnaiselle pistemäärälle (0–100) määritettiin arvosana kolmella eri tavalla: if-elsellä, switch-casella ja rekursion avulla. Toisessa sarjassa toteutettiin binääripuun haku 200 000 satunnaisella arvolla vertaillen rekursiivista ja iteratiivista hakua. Ehdon järjestyksen testauksessa aineistona oli 10 miljoonaa satunnaista ikää tai syntymäkuukautta. Testeissä tutkittiin, miten suorituskyky muuttuu riippuen siitä, onko useimmin

esiintyvä ehto rakenteen alussa, keskellä vai lopussa. Ikätesteissä aikuisten osuus ja kuukausitesteissä kesäkuun osuus oli määritetty 60 prosenttiin.

2.3 Tutkimuksen teoreettinen viitekehys

Tässä luvussa avataan tutkimuksen teoreettista viitekehystä. Nämä keskeiset termit ja ilmiöt kuvaavat niitä asioita, jotka ovat oleellisia tälle tutkimukselle. Nämä käsitteet toistuvat sekä kirjallisuuskatsauksessa että mittausten analysoinnissa, eli ne muodostavat yhteisen käsitteellisen rungon koko opinnäytetyölle. Termit perustuvat alalla vakiintuneisiin käytäntöihin, jotta samoista asioista puhutaan johdonmukaisesti samoilla nimillä.

Algoritmien aikavaativuusanalyysi: Teoreettinen tapa arvioida, kuinka paljon työtä algoritmi joutuu tekemään, kun käsiteltävän tiedon määrä kasvaa. Se ei mittaa suoritusaikaa sekunteina, vaan tarkastelee sitä, kuinka monta toimintoa, kuten vertailua tai laskutoimitusta, algoritmi tekee. (Benoit, Robert & Vivien 2013, luku 1.) Aikavaativuusanalyysi ei ole riippuvainen siitä, miten algoritmi on teknisesti kirjoitettu, vaan mitä se tekee loogisesti. (Bhargava 2024, luku 1.) Yksi keskeinen algoritmien aikavaativuusanalyysin työkalu on Big O -notaatio, jonka avulla kuvataan, kuinka nopeasti algoritmin työmäärä kasvaa syötteen kasvaessa. Esimerkiksi merkintä O(n) tarkoittaa, että algoritmin työmäärä kasvaa suoraviivaisesti syötteen koon kasvaessa, kun taas O(log n) viittaa huomattavasti hitaampaan logaritmiseen kasvuun. (La Rocca 2021, liite B.) Tässä tutkimuksessa algoritmien aikavaativuusanalyysin käsitteen avaamisen tarkoituksena on antaa lukijalle yleinen ymmärrys algoritmien tehokkuudesta ja siitä, että suorituskykyä voisi mitata myös tästä näkökulmasta, mutta tässä opinnäytetyössä se toimii kuitenkin vain taustateoriana tutkittavaan aiheeseen.

BenchmarkDotNet: Avoimen lähdekoodin .NET-kirjasto, joka on suunniteltu mittaamaan ohjelmakoodien suorituskykyä. Kirjasto mahdollistaa toistettavat ja luotettavat mittaukset sekä auttaa ohjelmoijia analysoimaan ohjelmointiratkaisujen tehokkuutta. (Akinshin 2019.)

C#: C# on Microsoftin kehittämä avoimen lähdekoodin ohjelmointikieli, jota käytetään laajasti esimerkiksi mobiilisovelluksissa, pilvipalveluissa ja pelikehityksessä. C# on osa .NET -ympäristöä, joka tarjoaa valmiita kirjastoja ja työkaluja sovelluskehitykseen. (Griffiths 2024, luku 1.)

Ehtorakenne: Ehtorakenteet ovat tärkeä osa ohjelmointia, ja niiden avulla ohjelman suoritusta voidaan ohjata eri tilanteiden mukaan. Ehtorakenteiden avulla ohjelma tekee päätöksiä ja valitsee, mikä koodilohko suoritetaan seuraavaksi. Ehtorakenteita käytetään silloin, kun halutaan tarkistaa jokin ehto ja toimia sen mukaan, riippuen siitä, onko ehto tosi vai epätosi. Tyypillisimpiä ehtorakenteita ovat if-, if-else ja switch-case -rakenteet. Ehtorakenteiden avulla voidaan tarkastella yhtä tai useampaa ehtoa. (Chen & Huang 2025, 133–135.)

LINQ: Language Integrated Query eli LINQ on C#-kielen ominaisuus, jonka avulla voidaan käsitellä ja hakea tietoa helposti erilaisista lähteistä. LINQ:a voidaan käyttää esimerkiksi tietokannoista, muistissa olevista olioista tai XML-tiedostoista haetun tiedon käsittelyyn. Se helpottaa tiedon suodattamista, järjestämistä ja muokkaamista, kun erillistä SQL-kieltä ei tarvita. (Griffiths 2024, luku 10.)

Lista: Lista on ohjelmoinnissa käytetty kokoelmatyyppi, joka soveltuu tilanteisiin, joissa tietoja halutaan säilyttää järjestyksessä, ja tarvittaessa lisätä ja poistaa tietoja ohjelman ajon aikana. Listan koko kasvaa tai pienenee tarpeen mukaan ilman, että kehittäjän täytyy määritellä tarkkaa kokoa etukäteen. (McGee 2015, luku 10.) C#-kielessä lista kuuluu geneerisiin kokoelmiin, ja se tukee monipuolisia toimintoja, kuten tietojen lajittelua, suodatusta ja läpikäyntiä. (Jamro 2024, luku 6.)

Microbenchmarking: Microbenchmarking tarkoittaa pienten ohjelman osien, esimerkiksi yksittäisten metodien tai algoritmien suorituskyvyn mittaamista. Microbenchmarkingin tarkoituksena on testata mahdollisimman tarkasti ja yksityiskohtaisesti, kuinka nopeasti jokin tietty koodin osa toimii. Microbenchmarkingia käytetään yleensä silloin, kun mitataan suorituskyvyn

näkökulmasta kriittisiä tilanteita, joissa ohjelman nopeus on ratkaiseva tekijä. (Evans, Gough & Newland 2018, luku 5.)

Rekursio: Rekursio tarkoittaa ohjelmoinnissa tilannetta, jossa funktio kutsuu itseään ongelman ratkaisemiseksi. Rekursiossa ongelma jaetaan ensin pienempiin, alkuperäistä ongelmaa muistuttaviin osiin. Tämän jälkeen funktio toistaa itseään niin kauan, kunnes niin sanottu perustapaus on saavutettu. Perustapauksella tarkoitetaan tilannetta, jossa ongelma on jo riittävän pieni ratkaistavaksi ilman uusia funktiokutsuja. (Roberts 2005, luku 1.)

Sanakirja: Sanakirja on tietorakenne, joka tallentaa tietoa avain–arvo-pareina. Jokainen avain on yksilöllinen ja liittyy tiettyyn arvoon, joka voidaan hakea nopeasti avaimen avulla. Sanakirjat tukevat laajasti erilaisia toimintoja, kuten tietojen lisäämistä, poistamista, hakemista ja läpikäyntiä silmukoiden avulla. (McGee 2015, luku 10.)

Suorituskykyanalyysi: Suorituskykyanalyysi tarkoittaa järjestelmällistä tapaa selvittää, mitkä tekijät vaikuttavat ohjelman tehokkuuteen negatiivisesti. Suorituskyvyn analysoinnilla voidaan tunnistaa ongelmien juurisyyt ja löytää parhaiten toimivat ratkaisut suorituskyvyn optimoimiseen. (Rossett 2009, luku 3.)

Taulukko: Taulukko on yksi ohjelmoinnin perusrakenteista, ja se on yksi yksinkertaisimmista tietorakenteista. Taulukossa tiedot tallennetaan peräkkäin, ja jokaisella tiedolla on oma sijaintinsa eli indeksi. Indeksin avulla taulukosta voidaan hakea haluttu alkio nopeasti, koska ohjelma tietää tarkalleen, missä kohtaa muistia tieto sijaitsee. Taulukon koko määritellään etukäteen, eikä sen kokoa voi muuttaa ohjelman suorituksen aikana. (Wengrow 2024, luku 1.)

Tietorakenne: Tietorakenne tarkoittaa tapaa, jolla tieto järjestetään ja tallennetaan ohjelmissa. Tietorakenteen avulla tiedon käsittely, haku ja muokkaus voidaan toteuttaa mahdollisimman tehokkaasti. Oikean tietorakenteen valinta on tärkeä osa ohjelmointia ja algoritmien suunnittelua, ja

se vaikuttaa suoraan ohjelman suorituskykyyn. (Vijayalakshmi Pai 2023, luku 1.)

Toistorakenne: Toistorakenteet eli silmukat ovat ohjelmoinnin perusrakenteita, ja niiden avulla sama koodilohko voidaan suorittaa useita kertoja peräkkäin ilman että samaa koodia tarvitsee kirjoittaa uudelleen. Ne soveltuvat esimerkiksi tilanteisiin, joissa ohjelman pitää suorittaa toistuvia tehtäviä. C#-kielessä käytetyimpiä toistorakenteita ovat while-, do-while-, foreach- ja for-silmukat. Jokainen silmukka toimii hieman eri tavalla, mutta perusidea taustalla on kuitenkin sama, tarkoituksena on suorittaa tiettyä koodilohkoa niin pitkään, kunnes sille määritelty ehto ei enää täyty. (Michaelis 2023, luku 4.)

Performance Profiler: Performance Profiler on Visual Studion sisäänrakennettu työkalu, jonka avulla voidaan mitata ja analysoida sovellusten suorituskykyä. Performance Profiler tarjoaa useita erilaisia mittaustapoja, joita voidaan hyödyntää erilaisten suorituskykyongelmien tunnistamiseen ja optimointiin. Työkalun avulla voidaan esimerkiksi seurata, kuinka paljon prosessorin laskentatehoa tai muistia eri ohjelmanosat käyttävät. (Goldshtein, Zurbalev, & Flatow 2012, luku 2.)

3 Tietorakenteet

3.1 Tietorakenteiden yleiskatsaus

Tietorakenteet ovat tärkeä osa ohjelmointia, sillä niiden avulla voidaan tallentaa ja järjestää tietoa niin, että tiedon käsittely ja hallinta on mahdollisimman tehokasta ja tarkoituksenmukaista. Tietorakenteet määrittävät miten tiedot ovat yhteydessä toisiinsa ja millaisia operaatioita tiedoille voidaan suorittaa. Tällaisia operaatioita voivat olla esimerkiksi tiedon lisääminen, poistaminen ja tiedon hakeminen. (La Rocca 2024, luku 1.)

Hyvin valitut ja tarkoitukseen sopivat tietorakenteet voivat parantaa ohjelman suorituskykyä merkittävästi, ja ne mahdollistavat datan nopean käsittelyn pienemmillä resursseilla. Toisaalta “väärän” tietorakenteen valinta voi hidastaa ohjelman toimintaa, tai pahimmassa tapauksessa aiheuttaa ohjelman kaatumisen. (La Rocca 2024, luku 1.)

Eri tietorakenteilla on erilaisia suorituskykyyn vaikuttavia ominaisuuksia, jotka määrittävät, kuinka tehokkaasti tietoa voidaan hakea, lisätä ja poistaa. (Khan 2018, Data Structures And Writing Optimized Code In C#). Tietorakenteen valinta riippuu hyvin paljon siitä, mitä tarpeita kehittäjällä on sovellukselle ja millaisia resursseja on käytössä. Suorituskyvyn optimointi vaatii aina ymmärrystä siitä, miten eri tietorakenteet käyttäytyvät erilaisissa operaatioissa ja kuinka niiden käyttö vaikuttaa ohjelman tehokkuuteen. (Khan 2018, Data Structures And Writing Optimized Code In C#). Vaikka tehokkaammat tietorakenteet voivat olla suorituskyvyn kannalta parempia, kun käsitellään suuria määriä tietoa, yksinkertaisemmat rakenteet, kuten taulukot ja listat, voivat olla usein kuitenkin täysin riittäviä. (Canning, Broder & Lafore 2022, luku 16.)

3.2 Lista

Tietorakenteena lista mahdollistaa saman tyyppisten tietojen tallentamisen, hakemisen ja hallinnan. Lista on dynaaminen tietorakenne, eli sen koko voi muuttua suorituksen aikana. Lista on myös tyyppiturvallinen vaihtoehto, sillä sen sisältämät tiedot ovat aina määriteltyä tietotyyppiä tai periytyvät samasta kantaluokasta. Listan käyttö perustuu .NET System.Collections-Generic-nimiavaruuteen, ja se toteuttaa IEnumerable-rajapinnan, joka mahdollistaa listan läpikäynnin esimerkiksi silmukoiden avulla. (McGee 2015, luku 10.)

Listojen avulla tietoja voidaan esimerkiksi lisätä, poistaa tai järjestää. Lista on hyödyllinen tilanteissa, joissa tietojen pitää olla hyvin järjestyksessä ja helposti hallittavissa. Sitä voidaan käyttää esimerkiksi numeeristen tietojen tallentamiseen tai monimutkaisempien objektien, kuten luokkien, hallintaan.

Lisäksi listat tukevat polymorfismia eli listaan voidaan tallentaa eri luokkien objekteja, kunhan ne periytyvät samasta kantaluokasta. (Jamro 2024, luku 6.)

C#-kielen geneerinen *List*-luokka tarjoaa paljon hyödyllisiä ominaisuuksia kehittäjälle. Esimerkiksi *add*-metodilla voi lisätä elementtejä ja *remove*-metodilla poistaa niitä. Lisäksi listat tukevat laajasti LINQ-kyselyitä, joiden avulla tietoja voidaan lajitella, ryhmitellä ja suodattaa. (Jamro 2024, Arrays and Lists). Listojen koko kasvaa tarpeen mukaan ilman, että muistia täytyy varata etukäteen. Ne mahdollistavat tehokkaan indeksipohjaisen haun, mutta alkioiden lisääminen tai poistaminen keskeltä listaa on hidasta. Listojen etuna on kuitenkin niiden joustavuus sekä monipuoliset operaatiot, kuten lajittelu ja suodatus, joita taulukot eivät tue suoraan. (Khan 2018, Data Structures And Writing Optimized Code In C#).

3.3 Sanakirja

Sanakirja yhdistää yksilölliset avaimet arvoihin ja mahdollistaa tiedon nopean tallentamisen, hakemisen, päivittämisen ja poistamisen. Sanakirja on osa .NET:n System.Collections.Generic-nimiavaruutta, kuten edellä mainittu listakin, ja se tarjoaa tyyppiturvallisen tavan käsitellä tietoa, sillä sekä avaimen että arvon tietotyypit määritellään etukäteen. (McGee 2015, luku 10.) Sanakirjat sisältävät hyödyllisiä metodeja. Näillä metodeilla voidaan esimerkiksi tarkistaa, onko jokin tietty avain olemassa, hakea avainta vastaava arvo sekä poistaa avain–arvo-pareja. Lisäksi sanakirjan koko sisältö voidaan käydä läpi esimerkiksi foreach-silmukan avulla, mikä mahdollistaa tallennettujen tietojen hyödyntämisen erilaisissa tarkoituksissa. (McGee 2015, luku 10.)

Sanakirjat sopivat erityisesti tilanteisiin, joissa tarvitaan nopeaa hakutoimintoa avaimen perusteella. Näitä rakenteita käytetään silloin, kun tietoa täytyy käsitellä tehokkaasti yksilöllisten avainten perusteella. Sen sijaan, jos arvoa haetaan ilman avainta, suorituskyky voi heikentyä. Sanakirjat soveltuvat tietorakenteena hyvin tilanteisiin, joissa tiedon nopea haku ja tallennus ovat

keskeisiä vaatimuksia. (Khan 2018, Data Structures and Writing Optimized Code In C#).

3.4 Taulukko

Taulukko on ohjelmoinnissa yleisesti käytetty tietorakenne, joka mahdollistaa saman tyyppisten tietojen tallentamisen kiinteään kokoelmaan. Taulukon koko määritetään alustusvaiheessa, eikä sitä voi suoraan muuttaa myöhemmin. Taulukon jokaisella elementillä on indeksi, jonka avulla siihen voidaan viitata ja C#-kielessä indeksit alkavat aina nollasta. (Jamro 2024, luku 3.) Koska taulukoiden koko on kiinteä, ei tietojen lisääminen ja poistaminen onnistu suoraan ohjelman ajon aikana. Tarvittaessa voidaan luoda uusi suurempi tai pienempi taulukko, ja kopioida olemassa olevan taulukon tiedot siihen, mutta tämä ei muuta alkuperäisen taulukon kokoa. Jos tietomäärä ei ole ennalta tiedossa ja joustavuus on tarpeen, kannattaa näissä tilanteissa käyttää taulukoiden sijaan listoja, joissa koon muuttaminen on mahdollista ohjelman ajon aikana. (Jamro 2024, luku 3.)

Taulukot varaavat muistia etukäteen, mikä mahdollistaa nopean pääsyn yksittäisiin alkioihin indeksin perusteella. (Canning ym. 2022, luku 2.) Kuitenkin, jos tietorakenteeseen on tarpeellista lisätä tai poistaa alkioita usein, taulukon käyttäminen voi olla tehottomampaa kuin dynaamisten tietorakenteiden, kuten listojen, hyödyntäminen. Jos esimerkiksi alkion poisto tai lisäys tapahtuu keskellä taulukkoa, muiden alkioiden siirtäminen vie aikaa sekä heikentää suorituskykyä, kun vaaditaan uuden taulukon luonti ja vanhan kopiointi. (Khan 2018, Data Structures And Writing Optimized Code In C#).

3.5 Muita tietorakenteita

Puut ovat hierarkkisia tietorakenteita, joissa tiedot järjestetään solmujen (*nodes*) avulla siten, että jokaisella solmulla voi olla yhteys yhteen tai useampaan alisolmuun, joita kutsutaan myös lapsisolmuiksi. Puiden yläosassa

on yleensä yksi juurisolmu (*root*), josta kaikki muut solmut ovat saavutettavissa. Puut ovat yleisiä tietorakenteita esimerkiksi erilaisissa hakemistoissa. Puu tietorakenteita on olemassa useita erilaisia ja eri käyttötarkoituksiin sopivia, kuten esimerkiksi binääripuu, AVL-puu, punamusta puu, 2–3–4-puu sekä binäärinen hakupuu. (Canning ym. 2022, luku 8.)

Binäärinen hakupuu (*Binary Search Tree, BST*) on yksi puutietorakenne. Sen jokaisella solmulla voi olla enintään kaksi lapsisolmua, vasen ja oikea. Puussa jokaisella solmulla on arvo eli avain, ja solmut on järjestetty niin, että vasemmalla olevat avaimet ovat pienempiä ja oikealla olevat suurempia kuin nykyinen avain. Parhaimmillaan binäärisellä hakupuulla saavutetaan logaritminen aikavaativuus (*log n*), mikä tekee puusta nopean tietorakenteen suurilla tietomäärillä. Puun tehokkuus tietorakenteena riippuu kuitenkin usein siitä, miten tasapainossa puu on. Jos arvoja lisätään nousevassa järjestyksessä, puusta voi tulla vino, ja sen korkeus kasvaa merkittävästi. Tämä taas hidastaa puun toimintaa. (Downey 2017, luku 13.) Arvot kannattaakin lisätä puuhun satunnaisessa järjestyksessä, tämä toimintatapa ei takaa täydellistä tasapainoa mutta johtaa keskimäärin huomattavasti parempaan tasapainoon.

Pino on yksinkertainen tietorakenne, jonka toiminta perustuu LIFO (*First-In, Last-Out)* -periaatteeseen. LIFO tarkoittaa sitä, että uudet tiedot lisätään aina viimeisenä ja poistetaan ensimmäisenä. Yksinkertaisesti voidaan ajatella, että pinoon lisätään tietoa järjestyksessä päällekkäin, ja poistaminen tapahtuu aina pinon ylimmästä kohdasta, pinon keskelle tai pohjalle ei voida siis suoraan vaikuttaa. (Jamro 2024, luku 5.)

Pinon perusoperaatioihin kuuluvat Push, Pop ja Peek. Push lisää uuden alkion pinon päällimmäiseksi, Pop poistaa ja palauttaa ylimmän alkion, ja Peek tarkastelee pinon päällimmäistä alkiota ilman sen poistamista. Pinoja käytetään ohjelmoinnissa esimerkiksi laskutoimitusten suorittamisessa, myös merkkijonon kääntäminen voidaan toteuttaa pinon avulla. (Jamro 2024, luku 5.) Pinoon voidaan lisätä ja siitä voidaan poistaa alkioita tehokkaasti, mutta yksittäisiin alkioihin ei ole suoraa pääsyä ilman, että aiemmat alkiot poistetaan ensin. (Khan 2018, Data Structures And Writing Optimized Code In C#).

Jono on tietorakenne, jossa uudet alkiot lisätään jonon loppuun ja poistetaan jonon alusta. Jono toimii siis FIFO (*First-In, First-Out*) -periaatteella eli ensimmäisenä lisätty tieto myös poistetaan ensimmäisenä. Jono toimii erityisen hyvin tilanteissa, joissa tiedon käsittely tapahtuu sen saapumisjärjestyksessä, kuten esimerkiksi tulostusjonossa tai muussa vastaavassa. (Jamro 2024, luku 5.)

Jonon perusoperaatioita ovat:

- Enqueue – lisää uuden alkion jonon loppuun
- Dequeue – poistaa ja palauttaa alkion jonon alusta
- Peek – tarkastelee jonon ensimmäistä alkiota poistamatta sitä
- Clear – tyhjentää jonon kaikista alkioista
- Contains – tarkistaa, sisältääkö jono tietyn alkion.

Jonon toimintaa voi havainnollistaa yksinkertaisen esimerkin avulla: Ajatellaan, että asiakas saapuu esimerkiksi pankin palvelupisteelle. Asiakas menee jonon perälle, ja palvelu etenee järjestyksessä niin, että ensimmäisenä jonoon tullut asiakas palvellaan ensin. (Jamro 2024, luku 5.)

Koska jonossa tietojen lisäys tehdään jonon loppuun ja tietojen poisto jonon alusta, jono ei ole paras ratkaisu tietorakenteena tilanteissa, joissa tietoa pitäisi käsitellä satunnaisessa järjestyksessä jonon sisällä. Jonosta ei voi hakea tai poistaa alkioita suoraan rakenteen keskeltä, vaan tietyn alkion etsiminen vaatii koko jonon läpikäymistä, mikä voi heikentää suorituskykyä erityisesti suurilla tietomäärillä. (Khan 2018, Data Structures and Writing Optimized Code In C#).

4 Ohjelmointirakenteet ja niiden suorituskyky

4.1 Ehtorakenteet

Ehtorakenteet eli ehtolauseet ovat ohjelmoinnin perusrakenteita, joiden avulla ohjelman suorituksen kulkua voidaan ohjata kehittäjän asettamien ehtojen perusteella. Ehtolauseet tarkistavat, täyttyykö määritelty ehto, ja tämän perusteella ohjelma päättää, mikä koodilohko suoritetaan seuraavaksi. Ehtorakenteet voivat sisältää hyvin yksinkertaisiakin ehtoja, ja näitä rakenteita käytetään usein tilanteissa, joissa ohjelman täytyy tehdä valintoja esimerkiksi käyttäjän toimien perusteella. (SuomiGameHub 2023.)

Ehtojen avulla ohjelma voi tehdä valintoja ja mukautua erilaisiin tilanteisiin, kuten ohjaamaan sitä, mitä ohjelma tekee seuraavaksi esimerkiksi annetun syötteen tai muuttujan arvon perusteella. Yleisimpiä ehtolauseita C#-kielessä ovat if-else ja switch-case, ja näiden lauseiden avulla aloittelevakin kehittäjä saa rakennettua yksinkertaisia ja tehokkaita koodilohkoja ohjelman suorituksen ohjaamiseen. (Drayton, Albahari & Neward 2003, luku 2.)

4.1.1 If-else

If-else-rakenne on yksi ohjelmoinnin perusrakenteista, jolla voidaan suorittaa tietty koodilohko, jos annettu ehto toteutuu. Rakenne perustuu loogiseen lauseeseen, joka arvioi ehtoa ja suorittaa joko ensimmäisen ehdon mukaisen lohkon (*if*) tai vaihtoehtoisen lohkon (*else*), jos alkuperäinen ehto ei toteudu. C#-kielessä if-else hyväksyy ainoastaan boolean totuusarvon tosi tai epätosi (*true tai false).* Jos if-lauseen perässä oleva ehto on tosi (*true*), suoritetaan tämä lohko, muussa tapauksessa siirrytään seuraavaan lohkoon (*false*). (Drayton ym. 2003, luku 2.)

Esimerkiksi käyttäjän syötteiden vertailu tai päätöksenteko, kuten onko luku suurempi, pienempi vai yhtä suuri kuin toinen luku, voidaan toteuttaa if-else-rakenteella. Monimutkaisempia ehtoja voidaan käsitellä loogisilla operaattoreilla, kuten && (*ja*), || (*tai*) ja ! (*ei*). Näiden operaattoreiden avulla voidaan käsitellä myös tilanteita, joissa lopputulos riippuu useamman kuin yhden ehdon täyttymisestä. (Drayton ym. 2003, luku 2.)

4.1.2 Switch-case

Switch-case-rakenne tarjoaa tehokkaan tavan käsitellä tilannetta, jossa ohjelman tulee valita toiminto useista mahdollisista vaihtoehdoista. Ehtorakenteen toiminta perustuu siihen, että ensin määritellään muuttujan arvo, jonka perusteella valitaan oikea koodilohko suoritettavaksi. Tämän vuoksi switch-case on selkeämpi ja helpommin luettava vaihtoehto tilanteissa, joissa olisi muuten ketjutettava useita if-else lauseita. (Drayton ym. 2003, luku 2.)

Switch-case arvioi muuttujan arvon kerran ja vertaa sitä ennalta määriteltyihin vaihtoehtoihin (case). Jokainen case-lohko päättyy yleensä break-komentoon, joka estää muiden lohkojen suorituksen. Switch-case soveltuu hyvin tilanteisiin, joissa käsitellään ennalta määritettyjä arvoja, kuten kokonaislukuja tai merkkijonoja. Esimerkiksi valikon vaihtoehtojen käsittelyyn switch-case-ehtorakenne on hyvä vaihtoehto. (Drayton ym. 2003, luku 2.)

4.2 Toistorakenteet

C#-kielessä on olemassa neljä itsenäisesti toimivaa silmukkapohjaista rakennetta: for, while, do-while ja foreach. Nämä ovat toistorakenteita, joissa toisto toteutetaan silmukoiden avulla eli silmukassa suoritetaan tiettyä koodilohkoa useita kertoja ja niin kauan, kunnes jokin määritelty ehto muuttuu epätodeksi. (McGee 2015, luku 5.)

Suorituskyvyn kannalta kaikkein yksinkertaisimmat silmukat ovat yleensä tehokkaimpia, kun turhat operaatiot on minimoitu pois, mikä pienentää myös muistinkäyttöä. Suorituskyvyn maksimoimiseksi paras tapa on valita yksinkertainen silmukkarakenne ja välttää ylimääräisiä ja turhia ehtolausekkeita ja funktiokutsuja silmukan sisällä. (Lavieri 2024, luku 3.)

4.2.1 Silmukkapohjaiset rakenteet

For-silmukka sopii erityisesti sellaisiin tilanteisiin, joissa halutaan käydä läpi joukko alkioita ja kun toistojen määrä on etukäteen tiedossa. For-silmukkaa käytetään usein esimerkiksi taulukoiden tai muiden indeksoitujen rakenteiden läpikäyntiin. (McGee 2015, luku 5.) For-silmukan toiminta perustuu siihen, että sitä suoritetaan niin kauan, kun annettu ehto pitää paikkaansa. Mikäli ehtoa ei aseteta oikein, silmukka voi jatkaa loputtomiin ja aiheuttaa niin sanotun ikiluupin, joka aiheuttaa ohjelman jumiutumisen. (Codecademy 2023.)

While-silmukassa tiettyä koodilohkoa suoritetaan toistuvasti niin kauan, kun sille määritelty ehto on tosi. While-silmukka on kätevä erityisesti tilanteissa, joissa toistojen määrä ei ole etukäteen tiedossa. Silmukka jatkaa toistoa niin kauan, kunnes tilanne muuttuu epätodeksi. While-silmukka sopii esimerkiksi sellaisiin tilanteisiin, joissa ohjelman täytyy tarkistaa jatkuvasti jonkun ehdon täyttyminen ennen ohjelman siirtymistä eteenpäin. (McGee 2015, luku 5.) Silmukka tarkistaa ehdon aina ennen suoritusta, ja siksi on mahdollista, ettei sen sisältämää koodia suoriteta kertaakaan, jos ehto ei ole alusta alkaen tosi. (Codecademy 2023.)

Do-while-silmukka on samankaltainen while-silmukan kanssa, sillä erolla, että do-while silmukassa koodi suoritetaan aina vähintään kerran, koska ehtoa tarkastellaan vasta ensimmäisen suorituksen jälkeen. Do-while-silmukkaa kannattaa käyttää tilanteissa, joissa halutaan varmistaa, että silmukan koodilohko tulee suoritetuksi ainakin kerran. (McGee 2015, luku 5.)

Foreach-silmukka toimii erityisesti kokoelmien läpikäynnissä. Se käy automaattisesti läpi kokoelman jokaisen alkion yksi kerrallaan, eli silmukka sopii esimerkiksi taulukoiden, listojen tai muiden kokoelmien läpikäyntiin. (McGee 2015, luku 5.) Foreach-silmukka tekee koodista selkeämpää ja vähentää virhemahdollisuuksia, koska se piilottaa tarpeettomat indeksit ja iteraattorit. Iteraattoreilla tarkoitetaan tässä yhteydessä arvoja läpikäyviä olioita. Foreach-silmukat toimivat sekä taulukoiden että kokoelmien kanssa ja sopivat erityisen hyvin sisäkkäisiin silmukoihin, joissa perinteiset rakenteet ovat alttiimpia virheille. Koska foreach on hyvä silmukka tietojen läpikäyntiin, sitä kannattaa suosia aina silloin, kun kokoelmaa ei tarvitse muokata esimerkiksi poistamalla alkioita. (Bloch 2017, luku 9.)

For-, foreach-, while- ja do-while-silmukat ovat suorituskyvyn kannalta kaikki lähes yhtä tehokkaita, mutta näiden silmukoiden suorituskykyä voidaan myös parantaa vähentämällä toistuvia operaatioita. Esimerkiksi jos silmukan ehto tarkistaa kokoelman pituuden jokaisella kierroksella, tämä tieto kannattaa tallentaa paikalliseen muuttujaan ennen silmukan alkua. Näin vältetään turhat haut ja säästetään aikaa, varsinkin suurilla tietomäärillä. Toinen usein tehokas keino on kääntää silmukan kulkusuuntaa, eli aloitetaan viimeisestä alkiosta ja edetään kohti ensimmäistä alkiota. (Zakas 2010, luku 4.)

4.2.2 Rekursio

Rekursion toiminta perustuu siihen, että jokin toiminto kutsuu itseään uudestaan ja uudestaan, kunnes määritetty ehto täyttyy. Rekursiivinen rakenne pilkkoo monimutkaisen ongelman pienempiin, alkuperäisen ongelman kaltaisiin osiin, joita suoritetaan toistuvasti, kunnes asetettu ehto on tosi. (Rubio-Sanchez 2017, luku 1.1.) Tämä lopetusehto, jota kutsutaan myös nimellä perustapaus, pysäyttää rekursion ja palauttaa lopullisen tuloksen. Lopetusehdon määritteleminen on kriittinen osa rekursiota, sillä ilman sitä ohjelma jatkaisi itsensä kutsumista loputtomiin, mikä väistämättä johtaisi jossain vaiheessa virhetilanteeseen. (Ohjelmointiputka 2003.) Rekursio voidaan jakaa erilaisiin tyyppeihin sen rakenteen perusteella ja näitä tyyppejä ovat esimerkiksi suora

rekursio ja epäsuora rekursio. Suora rekursio tarkoittaa tilannetta, jossa funktio kutsuu itseään, kun taas epäsuorassa rekursiossa useampi funktio kutsuu toisiaan niin sanotusti ketjussa. (Anggoro 2016, luku 7.) Esimerkiksi epäsuorassa rekursiossa funktio A voi kutsua funktiota B ja B puolestaan A:ta. Rakenne voi olla myös monimutkaisempi, jolloin funktio A kutsuu funktio B:tä, B kutsuu funktio C:tä ja lopuksi C kutsuu taas funktio A:ta. Näin muodostuu epäsuora rekursiivinen kierto.

Vaikka rekursio on tehokas tapa ratkaista monia ongelmia, sillä on omat haasteensa. Rekursiiviset rakenteet voivat olla resurssien kannalta tehottomia, sillä joka kerta, kun funktio kutsuu itseään, se vie tilaa ohjelman pinoalueelta. Tämä voi johtaa suorituskyvyn heikkenemiseen tai jopa virheisiin, kuten pinoalueen ylivuotoon (*Stack Overflow*). (Rubio-Sanchez 2017, luku 1.5.) Pinoalueen ylivuoto on mahdollista estää häntärekursion (*Tail Recursion*) avulla. Häntärekursiossa rekursiivinen kutsu on funktion lopussa niin, että sen jälkeen ei jää suoritettavaksi enää muuta koodia. Joissakin ohjelmointikielissä kääntäjä voi optimoida tällaiset tilanteet automaattisesti. Kun rekursiivinen kutsu on viimeinen toiminto, eikä muuta tämän jälkeen tehdä, kääntäjä voi jättää tallentamatta aiemmat kutsut pinoon ja näin estää pinoalueen ylivuodon. (Sweigart 2022, luku 8.) C#-kielessä tätä optimointia ei kuitenkaan tehdä automaattisesti, vaikka se teoriassa onkin mahdollista, joten häntärekursion käyttö voi tehdä koodista vaikealukuisempaa. Käytännössä sama lopputulos on mahdollista saavuttaa selkeämmin käyttämällä silmukkaa.

Yksi rekursion suorituskykyyn liittyvä ongelma suorituskyvyn kannalta on se, että sama laskutoimitus saatetaan joutua tekemään useita kertoja niin sanotusti ”turhaan”. Esimerkiksi rekursiivisessa Fibonacci-laskennassa ohjelma laskee samat väliarvot yhä uudelleen ja uudelleen, koska se ei tallenna automaattisesti mitään aiemmin laskettua muistiin. Tämä toimintatapa hidastaa ohjelman suoritusta huomattavasti. Iteratiivisissa ratkaisuissa sen sijaan edetään vaihe vaiheelta järjestyksessä, ja kaikki tarvittavat arvot tallennetaan muistiin. Tästä syystä silmukkapohjaiset rakenteet ovat usein nopeampia ja käyttävät vähemmän laskentatehoa. (Sweigart 2022, luku 2.)

Vaikka rekursiota ei lopulta käyttäisi ohjelmoinnissa, rekursiivinen ajattelutapa voi olla silti hyödyllinen. Rekursiivinen lähestymistapa voi auttaa kehittäjiä hahmottamaan, miten ongelma voidaan jakaa pienempiin ja tehokkaammin ratkaistaviin osiin. Tämä ajattelutapa voi tuoda uusia ideoita myös siihen, miten voidaan kehittää nopeampi ja parempi ratkaisu myös silmukoita käyttämällä. Rekursio voi toimia apuna suunnittelussa, vaikka varsinainen toteutus tehtäisiinkin silmukkapohjaisten rakenteiden avulla. (Sweigart 2022, luku 2.) Rekursio toimii hyvin tietyissä tilanteissa, mutta suorituskyvyn kannalta iteraatio on usein nopeampi ratkaisu. Varsinkin silloin, jos toistoja on paljon ja käytettävissä olevat resurssit ovat rajalliset. Molemmilla tavoilla voidaan ratkaista samanlaisia ongelmia, mutta rekursio ja iteratiiviset ratkaisut toimivat lähtökohtaisesti eri tavoilla. Lopulta on jokaisen oman harkinnan varassa valita, mikä on tilanteeseen sopivin lähestymistapa. (Sweigart 2022, luku 2.)

4.3 LINQ

LINQ eli Language Integrated Query on C#-kielen ja .NET:n ominaisuus, jonka avulla voidaan käsitellä erilaisia kokoelmia. Sen avulla voidaan hakea, suodattaa, järjestellä ja yhdistellä dataa ilman monimutkaisten silmukoiden tai ylimääräisen koodin kirjoittamista. LINQ toimii minkä tahansa objektin kanssa, joka toteuttaa IEnumerable<T>-rajapinnan System.Collections.Generic-nimiavaruudessa. (Stellman & Greene 2024, luku 9.)

LINQ:a voidaan käyttää kahdella eri tavalla: metodien ketjutuksen avulla tai kyselysyntaksin kautta. Metodien ketjutus mahdollistaa useiden metodien kutsumisen peräkkäin samassa lauseessa, kun taas kyselysyntaksi muistuttaa enemmänkin SQL-kieltä, ja sen avulla saadaan monimutkaiset kyselyt helposti luettavaan muotoon. LINQ-kyselyiden avulla on mahdollista esimerkiksi suodattaa tietoja tiettyjen ehtojen perusteella, järjestää tietueita eri tavoilla, kuten nousevaan tai laskevaan järjestykseen, ja yhdistää kokoelmia yhdeksi kokonaisuudeksi. (Stellman & Greene 2024, luku 9.)

LINQ on kohtuullisen tehokas tapa käsitellä ja suodattaa tietoja, ja sitä käytetään usein sen helppolukuisuuden ja joustavuuden vuoksi. LINQ:n suorituskyky on kuitenkin herättänyt huolta ja sitä on usein kuvattu hitaaksi ja tehottomaksi. (Vickers 2020.) LINQ ei välttämättä ole itsessään hidas, vaan sen tehokkuus riippuu hyvin pitkälti siitä, miten sitä käytetään.

Yksi yleinen suorituskykyongelma liittyy tapaan, jolla kokoelman viimeinen arvoa haetaan. LINQ:n Last()-metodia käytettäessä suorituskyky usein heikkenee, koska metodi saattaa joutua käymään läpi kaikki kokoelman alkiot päästäkseen viimeiseen. Suorituskyvyn kannalta hyvä vaihtoehto on käyttää tavallisia taulukoita ja listoja, jotka tukevat suoraa indeksihakua. Sen sijaan linkitetyt listat ja muut IEnumerable<T>-tyyppiset kokoelmat eivät tarjoa suoraa pääsyä indekseihin. (Alls 2022, luku 7.) Toinen suorituskykyyn liittyvä tekijä on let-avainsanan käyttö LINQ-kyselyissä. Let-avainsanan avulla voidaan tallentaa väliaikaisia arvoja kyselyiden sisällä, mikä saattaa parantaa koodin luettavuutta, mutta samalla suorituskyky kärsii. Avainsanan käyttö voi lisätä muistinkulutusta ja hidastaa kyselyiden suoritusaikaa verrattuna tilanteeseen, jossa arvoa ei tallenneta erilliseen muuttujaan. (Alls 2022, luku 7.)

Ryhmittely (*GroupBy*) on yksi LINQ:n tehokkaimmista ominaisuuksista, mutta myös sen suorituskykyä voidaan optimoida. Parhaimmat tulokset saavutetaan, kun käsiteltävät tiedot muutetaan ensin taulukoksi ennen ryhmittelyä. Tämä nopeuttaa suoritusaikaa verrattuna siihen, että tiedot käsiteltäisiin suoraan listasta. (Alls 2022, luku 7.) Perinteinen for-silmukka tarjoaa usein nopeamman vaihtoehdon tietojen suodatukseen, koska silloin LINQ:n tuomat lisäkäsittelyvaiheet jäävät pois. (Alls 2022, luku 7.)

Vickers (2020) tutki LINQ:n suorituskykyä verrattuna perinteisiin for- ja foreach-silmukoihin BenchmarkDotNet-kirjaston avulla. Testeissä arvioitiin, kuinka nopeasti eri menetelmät suoriutuvat perusoperaatioista, kuten tietojen suodattamisesta ja muuntamisesta. (Vickers 2020.) Tulokset osoittivat, että useimmissa tapauksissa LINQ:n suorituskyky oli hyvin lähellä perinteisiä silmukkapohjaisia rakenteita. Eroja havaittiin kuitenkin siinä, miten kyselyt toteutettiin. Erityisesti tilanteessa, jossa laskettiin tietyt ehdot täyttävien

henkilöiden määrä suoraan LINQ:n avulla, suorituskyky oli huomattavasti heikompi kuin perinteisellä silmukalla. Sen sijaan tietojen suodattaminen ja muokkaaminen sujui lähes yhtä nopeasti riippumatta siitä, käytettiinkö LINQ:a vai for- tai foreach-silmukkaa. (Vickers 2020.) Vickers (2020) teki testien perusteella johtopäätöksen, että LINQ ei yleensä aiheuta merkittäviä suorituskykyongelmia sen maineesta huolimatta, mutta jos koodin nopeus on tärkein kriteeri suorituskyvyn optimoinnissa, kannattaa testejä tehdä tapauskohtaisesti ja tehdä lopulliset valinnat vasta saatujen tulosten perusteella. (Vickers 2020.)

5 Suorituskyvyn mittaamisen perusteet

5.1 Mitä suorituskyvyn mittaaminen tarkoittaa

Suorituskyvyn mittaaminen ohjelmistokehityksessä tarkoittaa sovelluksen toiminnan arvioimista ja analysointia erilaisten numeeristen mittareiden avulla. Suorituskykyä voidaan mitata esimerkiksi suoritusajan, muistin käytön tai prosessorin kuormituksen perusteella. Näitä erilaisia mittareita käytetään ohjelman tehokkuuden arvioimiseen ja jotta voidaan tunnistaa ohjelmassa olevat pullonkaulat tai muut epäkohdat, jotka kaipaavat optimointia. (Goldshtein ym. 2012, luku 1.)

Suorituskyvyn mittaaminen on tärkeää monista eri syistä. Se auttaa varmistamaan, että ohjelma täyttää asetetut tavoitteet esimerkiksi suorituskyvyn osalta. Näitä tavoitteita voivat olla esimerkiksi nopea vasteaika, resurssien tehokas käyttö ja ohjelman vakaa toiminta myös silloin, kun käyttäjiä on useita samaan aikaan. Suorituskyvyn mittaaminen auttaa myös mahdollisten ongelmien havaitsemisessa ja tunnistamisessa, ennen kuin ne aiheuttavat esimerkiksi ohjelman kaatumisen. (Goldshtein ym. 2012, luku 1.)

5.2 Mittaustekniikat ja työkalut

Suorituskyvyn mittaamiseen on olemassa useita erilaisia työkaluja, joiden avulla saadaan monipuolisesti tietoa sovellusten suorituskyvystä. Työkaluissa on hieman eroa sen suhteen, millaisia tuloksia suorituskyvystä saadaan. Osa työkaluista mittaa yleisemmin esimerkiksi prosessorin kuormitusta ja muistinkäyttöä, kun taas toisilla työkaluilla saadaan yksityiskohtaista tietoa yksittäisten koodilohkojen suorituskyvystä. Parhaan lopputuloksen saamiseksi eri työkalujen käyttö yhdessä on järkevää, sillä silloin suorituskykyä voidaan tarkastella useista eri näkökulmista. (Goldshtein ym. 2012, luku 2.)

5.2.1 Visual Studio Performance Profiler

Visual Studio tarjoaa useita erilaisia työkaluja ohjelmien suorituskyvyn mittaamiseen. Mittaustekniikat on suunniteltu sopiviksi erilaisiin tilanteisiin ja ongelmiin, ja niitä voidaan hyödyntää sovelluksen optimoinnissa. Visual Studiosta löytyvän työkalun, Visual Studio Performance Profilerin avulla, suorituskykyä voidaan mitata useilla eri tavoilla:

- Sampling Profiler, näytteisiin perustuva profilointi

Näytteisiin perustuva profilointi on melko kevyt tapa mitata suorituskykyä. Sen toiminta perustuu siihen, että ohjelman suorituksen aikana kerätään tietoa sovelluksen CPU-ajasta ottamalla näytteitä. Menetelmä sopii erityisesti CPU-kuormituksen analysointiin. (Goldshtein ym. 2012, luku 2.)

- Instrumentation Profiler, instrumentointiin perustuva profilointi

Instrumentointiin perustuva profilointi tarjoaa tarkempaa tietoa ohjelman suoritusajoista. Ohjelman koodiin lisätään mittauspisteitä, joihin tallentuu jokaisen menetelmäkutsun kesto ja määrä. Tämä tekniikka mahdollistaa myös I/O-keskeisten ohjelmien suorituskyvyn mittaamisen, sillä se seuraa kaikkia metodeja, ei vain niitä, jotka käyttävät CPU:ta. (Goldshtein ym. 2012, luku 2.)

- Allocation Profiler, muistiprofilointi

Visual Studio Profiler mahdollistaa myös muistin käyttöön liittyvän profiloinnin. Tällä tekniikalla mitataan, kuinka paljon muistia eri metodit käyttävät ja tunnistetaan, mitkä objektit vievät eniten muistia (Goldshtein ym. 2012, luku 2.)

5.2.2 BenchmarkDotNet

Suorituskykyä voidaan mitata myös hyvin pienellä tarkkuudella. Tätä kutsutaan Microbenchmarkingiksi. Microbenchmarking tarkoittaa esimerkiksi yksittäisten metodien, funktioiden tai koodinosien suorituskyvyn analysointia. Microbenchmarking toimii täydennyksenä muille mittaustekniikoille, koska se keskittyy tarkkoihin yksityiskohtiin. Sen avulla voidaan esimerkiksi tarkastella ja analysoida pienempien toteutusten suorituskykyä. (Goldshtein ym. 2012, luku 2.)

BenchmarkDotNet on .NET-kirjasto, joka on suunniteltu sovellusten suorituskyvyn mittaamiseen. Kirjasto hoitaa automaattisesti useimmat suorituskyvyn mittaamiseen liittyvät tehtävät, kuten testiajojen toistamisen, lämmittelyvaiheen erottamisen varsinaisista mittauksista ja ympäristötekijöiden huomioimisen. Kirjaston käyttö vähentää tyypillisiä virhetilanteita, kuten sitä, että testejä ajetaan ilman optimointia. Kirjasto pitää myös huolen siitä, että kaikki mittaukset tehdään yhdenmukaisilla asetuksilla. Näiden lisäksi kirjasto pystyy mukautumaan erilaisiin tilanteisiin ja säätämään mittausparametreja automaattisesti, jotta tulokset olisivat mahdollisimman tarkkoja. (Akinshin 2019, luku 6.)

Tulosten analysointi on tehty kohtuullisen helpoksi BenchmarkDotNetin kautta. Kirjasto laskee keskeiset tilastolliset suureet, tunnistaa poikkeavat arvot ja suorittaa tilastollisia testejä luotettavuuden varmistamiseksi. Kirjaston avulla kehittäjä voi tuoda raportit ohjelmasta haluamassaan formaatissa, BenchmarkDotNet tukee muun muassa CSV-, JSON-, HTML-, XML- ja Markdown-tiedostomuotoja. Vaikka BenchmarkDotNet tekee suorituskyvyn testaamisesta helpompaa ja sujuvampaa, kehittäjän vastuulla on kuitenkin

tulosten analysointi ja mahdollisten optimointien tekeminen. Kirjasto ei siis ole mikään ihmetyökalu, joka ratkaisee suorituskykyongelmat automaattisesti, mutta se tarjoaa luotettavan pohjan näiden ongelmien tunnistamiseen. (Akinshin 2019, luku 6.)

5.3 Suorituskyvyn mittaamiseen vaikuttavat tekijät

Ohjelmien suorituskykyyn vaikuttaa lähdekoodissa tehtyjen koodaamiseen liittyvien valintojen lisäksi moni muukin asia. Voidaan siis sanoa, että suorituskyky on monen asian summa. Ohjelmakoodi toimii aina jossain tietyssä ympäristössä, ja juuri tällä ympäristöllä on merkittävä vaikutus siihen, kuinka nopeasti ja tehokkaasti ohjelma toimii käytännössä. Suorituskykyyn vaikuttavat erityisesti ajonaikainen ympäristö (*runtime*), käännöstyökalut, käyttöjärjestelmä, laitteisto sekä fyysiset olosuhteet. Suorituskyvyn mittaaminen ei ole kovin yksiselitteinen asia, vaan se vaatii tarkempaa ymmärrystä siitä ympäristöstä, jossa mittaukset tehdään. (Akinshin 2019, luku 3.)

.NET:llä on erilaisia ajoympäristöjä, ja kaikilla näillä ympäristöillä on omat toteutuksensa ja optimointinsa. Pienetkin versiomuutokset voivat vaikuttaa suorituskykyyn merkittävästi. Build-konfiguraation (*Debug vs. Release*) valinta vaikuttaa suoraan kirjoitettavan koodin suorituskykyyn. Debug-versiot ovat yleensä hitaampia, koska niissä ei ole automaattista optimointia. Myös tietokoneen käyttöjärjestelmällä on oma roolinsa, kun mietitään suorituskykyyn vaikuttavia tekijöitä. Sen lisäksi, että sama ohjelma voi tuottaa täysin erilaisia tuloksia Windowsilla, Linuxilla ja macOS:lla, jo pelkästään saman käyttöjärjestelmän eri versio voi tuoda suorituskykyyn muutoksia. (Akinshin 2019, luku 3.)

Laitteistojen osalta erityisesti prosessorin tyyppi ja arkkitehtuuri, RAM-muistin määrä ja nopeus sekä nettiyhteyden laatu vaikuttavat suorituskykyyn. Suorituskyvyn mittaamisessa on huomioitava useita sellaisia asioita, joita ei välttämättä tule heti ajatelleeksi. Suorituskykyä mitattaessa ei välttämättä

ymmärretä, että myös laitteistolla testien aikana pyörivät taustajärjestelmät, kuten virustorjuntaohjelma tai näytönsäästäjä, voivat huomaamatta hidastaa ohjelman toimintaa. Myös testiympäristön fyysiset olosuhteet, kuten ilman lämpötila ja kosteusprosentti, voivat vaikuttaa suorituskykyyn. (Akinshin 2019, luku 3.)

Yksi kehittäjiltä usein niin sanotusti ”piiloon jäävä” tekijä on prosessorin *branch prediction* eli haarautumisen ennustaminen. Haarautumisen ennustaminen tarkoittaa sitä, että prosessori arvaa etukäteen, mihin ohjelma tulee seuraavaksi jatkamaan, kun se kohtaa esimerkiksi ehtolauseen tai silmukan. Modernit prosessorit käyttävät nykyään dynaamista haarautumisen ennustamista, jossa prosessori oppii ohjelman käyttäytymisestä suorituksen aikana. Haarautumisen ennustaminen on prosessorin ominaisuus, joka toimii täysin automaattisesti, eikä sen kytkeminen pois käytöstä ole suoraan mahdollista. (Patterson & Hennessy 2020, luku 4.9.)

Haarautumisen ennustaminen toimii yhdessä spekulatiivisen suorituksen kanssa. Spekulatiivinen suoritus tarkoittaa sitä, että prosessori suorittaa etukäteen käskyjä, vaikka ohjelman oikea suoritussuunta ei ole vielä tiedossa. (Kocher, Horn, Fogh, Genkin, Gruss, Haas, Hamburg, Lipp, Mangard, Prescher, Schwarz & Yarom 2019, luku 1.) Tämä tehdään suorituskyvyn parantamiseksi, kun prosessori ei jää odottamaan esimerkiksi hitaan muistin vastausta, vaan arvaa, mikä haaran suunta tulee olemaan ja jatkaa ohjelman suoritusta tämän oletuksen mukaan. Jos ennustus osoittautuu lopulta oikeaksi, työ on tehty etukäteen ja tämä säästää aikaa. Jos taas ennustus menee pieleen, spekulatiivisesti suoritetut käskyt perutaan ja prosessori palauttaa ohjelman tilan siihen pisteeseen, jossa haaran oikea suunta selviää. Tästä ei aiheudu ohjelman toiminnalle mitään varsinaista ”haittaa”, mutta suorituskyky voi kärsiä turhaan tehdystä työstä. (Kocher ym. 2019, luku 2.)

Haarautumisen ennustaminen ja spekulatiivinen suoritus voivat vaikuttaa suorituskykyyn, mutta nämä ominaisuudet eivät ole mitenkään ”hallittavissa” mittaustilanteissa, eli prosessori tekee ennustuksen ja spekulatiivisen suorituksen joka tapauksessa. Vaikka tähän ei pystytä vaikuttamaan, on hyvä

tiedostaa, että prosessorilla on tällaisia automaattisia ominaisuuksia. Tämä aihe ei ole opinnäytetyön tutkimuksen kannalta keskeinen, mutta asian ymmärtäminen auttaa tulosten tulkinnassa, kun ymmärretään, että taustalla olevat automaattiset prosessit voivat vaikuttaa tuloksiin.

Haarautumisen ennustamisen lisäksi on olemassa myös muita niin sanottuja ”taustaprosesseja”, joilla voi olla vaikutusta suorituskykyyn. Esimerkiksi *garbage collection* eli roskienkeruu on yksi tällainen prosessi. Roskienkeruu tarkoittaa tilannetta, jossa ohjelma poistaa automaattisesti muistista turhiksi jääneitä olioita. Kehittäjän ei tarvitse itse huolehtia muistin vapauttamisesta, ja roskienkeruun ansiosta muisti ei myöskään täyty turhaan. Roskienkeruu tapahtuu automaattisesti ohjelman taustalla, mutta sillä voi olla vaikutusta esimerkiksi ohjelman suoritusnopeuteen. Joskus roskienkeruu pysäyttää ohjelman hetkeksi muistin puhdistamisen ajaksi, ja pysäytys voi näkyä viiveenä suorituskykytesteissä. Erilaiset testit voivat käynnistää roskienkeruun eri tavalla, eikä roskienkeruu aktivoidu säännöllisesti tietyissä tilanteissa. (Oaks 2020, luku 5.) Tämän tutkimuksen pääpaino ei ole roskienkeruussa, mutta roskienkeruulla voi olla vaikutusta siihen, miten tarkasti saadaan mitattua oikeita eroja suorituskyvyssä. Työkalut, kuten BenchmarkDotNet, ottavat roskienkeruun automaattisesti huomioon suorituskykytesteissä, jonka vuoksi tulokset pysyvät luotettavina ja vertailukelpoisina roskienkeruusta huolimatta.

5.4 Mittaustulosten luotettavuus ja virhemahdollisuudet

Kuten edellisessä luvussa todettiin, ohjelman suorituskykyyn voivat vaikuttaa monet eri tekijät. Näistä syistä suorituskykymittauksia ei voida pitää täysin luotettavina yksittäisten mittausten perusteella. Yhden testiajon tulos voi poiketa toisesta merkittävästi, vaikka testattava koodi pysyisi täysin samana. Tähän vaikuttavat muun muassa käyttöjärjestelmän taustaprosessit, keskusmuistin hetkellinen kuormitus tai prosessorin lämpötilasta johtuva kellotaajuuden lasku. (Oaks 2014, luku 2.)

Luotettavan kuvan saamiseksi mittauksia tulee suorittaa useita kertoja. Kun testi toistetaan esimerkiksi kymmenen kertaa ja tuloksista lasketaan keskiarvo, saadaan todenmukaisempi kuva koodin todellisesta suorituskyvystä. Näin yksittäiset poikkeamat tai satunnaisesti esiintyvät vaihtelut eivät vääristä kokonaiskuvaa. Mittausten toistaminen auttaa myös tunnistamaan mahdollisia pullonkauloja, jotka eivät tule ilmi yksittäisillä testiajoilla. (Oaks 2014, luku 2.) Testien toistaminen on tärkeää myös siksi, että tämä varmistaa niin sanotun ”lämmittelyvaiheen”. Ohjelmat eivät aina toimi täydellä teholla heti ensimmäisellä käynnistyksellä, ja testien toistamisella huolehditaan siitä, että lämmittelyvaihe on suoritettu ja kaikki ohjelman ajamat optimoinnit on tehty. (Oaks 2014, luku 2.) Useimmat työkalut kuitenkin huolehtivat automaattisesti lämmittelyvaiheen suorittamisesta ja erottavat sen muista varsinaisista mittauksista. (Akinshin 2019, luku 6.)

Suorituskykytestien tuloksia voidaan tarkastella myös tilastollisesta näkökulmasta. Esimerkiksi t-testi on tilastollinen menetelmä, jolla voidaan arvioida, onko kahden mittauksen välinen ero merkittävä, vai johtuuko se sattumasta. (Tähtinen, Laakkonen & Broberg 2020, 121.) Tämä on erityisen tärkeää silloin, kun vertaillaan eri koodiversioiden suorituskykyä. Pelkkä prosenttilukuna esitetty eroavaisuus ei kerro vielä riittävästi, vaan tarvitaan ymmärrystä siitä, kuinka todennäköisesti eroja on oikeasti olemassa. On myös tärkeää ottaa huomioon, että suuret erot tuloksissa eivät välttämättä aina tarkoita suurta merkitystä käytännön kannalta, eikä pieni ero ole välttämättä yhdentekevä. Esimerkiksi 1 %:n suorituskykyero voi olla merkittävä sovelluksessa, jota käytetään tuhansia kertoja sekunnissa, ja toisaalta 10 %:n ero voi olla täysin merkityksetön, jos toiminto suoritetaan vain kerran päivässä. Mittausten analysoinnissa on hyvä olla sekä tilastollista tarkkuutta että harkintakykyä siitä, mikä on sovelluksen kannalta oleellista. (Oaks 2014, luku 2.)

Suorituskykymittauksissa tulee olla myös tarkkana siitä, mitä oikeasti halutaan mitata. Testikoodi ei saisi sisältää mitään ylimääräistä kuten satunnaislukugeneraattorin toimintaa tai virheenkäsittelyä, joka ei ole osa mitattavaa ominaisuutta. Suorituskykymittausten luotettavuus perustuu siis

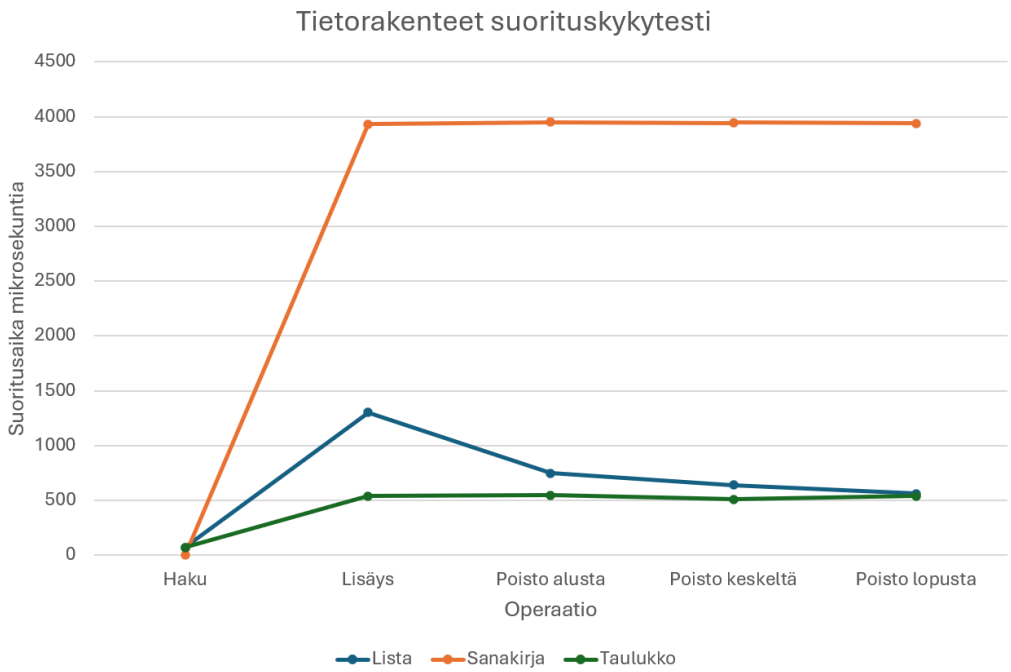
useisiin eri tekijöihin, kuten huolelliseen valmistautumiseen, testien toistettavuuteen ja tulosten kriittiseen tarkasteluun. Luotettavuuden varmistamisella huolehditaan siitä, että suorituskykymittauksista saatu tieto on oikeasti hyödyllistä ja hyödynnettävissä. (Oaks 2014, luku 2.)

6 Tulokset

6.1 Tietorakenteet

Tässä opinnäytetyössä testattavia tietorakenteita olivat lista, sanakirja ja taulukko. Tietorakenteiden suorituskykyä testattiin kolmella eri testiskenaariolla, ja testattaviin operaatioihin kuuluivat tietojen haku, lisäys ja poisto. Tietojen poistossa toteutettiin kolme erilaista esimerkkiä, joissa tietoa poistettiin tietorakenteen alusta, keskeltä ja lopusta.

Alkuperäisessä testiversiossa tietorakenteiden osalta havaittiin, että testit eivät kaikilta osin mitanneet operaation varsinaista aikavaativuutta, koska jokaisessa metodissa luotiin kokonaan uusi kopio alkuperäisestä tietorakenteesta. Tämä vaikutti erityisesti hakutoimintojen testituloksiin, sillä ylimääräinen muistinkäsittely peitti alleen varsinaisen hakuoperaation suorituskyvyn. Testikoodia päivitettiin niin, että hakumetodit suoritettiin suoraan olemassa olevaan alkuperäiseen tietorakenteeseen ilman uuden rakenteen luontia, jolloin saatiin tarkempi kuva varsinaisen hakutoiminnon suorituskyvystä ja tehokkuudesta. Testin perusteella tietorakenteiden suorituskyvyssä oli selkeitä eroja riippuen siitä, millaisia operaatioita suoritettiin (kuva 1).



Kuva 1. Tietorakenteiden suorituskykykaavio.

Taulukko osoittautui nopeimmaksi tietorakenteeksi useimmissa tilanteissa, erityisesti lisäyksissä ja poistoissa ja tästä kiitos kuuluu taulukon yksinkertaiselle rakenteelle. Lista suoriutui testeistä hyvin ja sen aiheuttama muistivaraus oli identtinen taulukon kanssa. Sanakirja oli odotetusti nopein hakutoiminnoissa, mutta lisäyksissä ja poistoissa se oli kaikkein hitain, ja näistä operaatioista aiheutui myös suurempaa muistivarausta (liite 2.)

Sanakirja

Sanakirjan hakumetodi oli ylivoimaisesti nopein kaikista testatuista tietorakenteista, kuten teoreettisesti oli oletettavissakin (liite 2). Lisäys- ja poisto-operaatioissa uusi sanakirja luotiin jokaisen metodin sisällä, koska näissä operaatioissa haluttiin säilyttää *baseDictionary* muuttumattomana, mikä on testauksen kannalta perusteltua. Lisäksi testien aikana jokainen lisäys- ja poisto-operaatio suoritettiin kopioimalla alkuperäinen sanakirja, mikä aiheutti tarpeen suurelle muistivaraukselle ja kasvatti muistinkäyttöä merkittävästi.

Prosessorin kuormituksen osalta *DictionaryTests*-testeissä suurin CPU-kuormitus kohdistui pääasiassa sanakirjan alustamiseen. Alustamisen osuus CPU-ajasta oli keskimäärin jopa 73 %. Alkioiden poisto- ja lisäysoperaatiot

olivat kaikkein kevyimpiä. Nämä operaatiot käyttivät CPU-ajasta niin vähän, että ne eivät päässeet Performance Profilerin tulostaulukkoon. Koko ohjelman muistinkäyttö oli tasaista läpi molempien testien muistinkäytön ollessa noin 146 megatavua.

Tulosten perusteella voidaan sanoa, että sanakirjalla on erinomainen suorituskyky erityisesti hakuoperaatioissa. Poisto- ja lisäysoperaatioissa se ei kuitenkaan yltänyt listan ja taulukon tasolle. Sanakirjan muistivaraus oli selvästi korkeampi kuin listalla ja taulukolla. Tämä viittaa siihen, että hajautustaulun käsittely on muistinkäytön kannalta raskasta, erityisesti silloin, kun suuri määrä tietoa kopioidaan ja rakennetaan uudelleen jokaisella testikierroksella.

Taulukko

Taulukko pärjäsi suorituskykytesteissä tasaisen vahvasti erityisesti suoritusnopeuden osalta (liite 2). Yksi suoritusnopeuteen varmasti positiivisesti vaikuttava asia on *Array.Copy*-menetelmän tehokkuus (Nyingi 2023). Vaikka teoriassa taulukon lisäys- ja poisto-operaatiot rakenteen keskeltä vaativat uuden taulukon luomisen ja kaikkien alkioiden siirtämisen, .NET-ympäristö toteuttaa nämä operaatiot tehokkaasti, mikä osaltaan selittää hyvät tulokset suoritusajoissa. Testien operaatiot olivat selkeitä ja suoraviivaisia, eikä niissä tarvittu dynaamista taulukon koon muuttamista yhdelle rakenteelle, koska jokaisessa lisäys- ja poisto-operaatiossa luotiin uusi taulukko.

Performance Profilerin mittauksissa *ArrayTests*-testit osoittivat hyvin, miten prosessorin kuormitus jakautui eri toimintojen välillä. Testeissä ohjelman muistinkäyttö pysyi molemmilla kerroilla tasaisena noin 67 megatavun tasolla koko testin ajan. Vaikka taulukon suoritusnopeus oli testeissä vertailluista tietorakenteista paras, Performance Profilerin mukaan sen käsittely kuormitti prosessoria yllättävän paljon. CPU-kuormituksessa taulukon testien selvästi suurin osuus kohdistui *System.Array.Copy*-metodiin, joka käytti ensimmäisessä testissä noin 54 % ja toisessa testissä noin 63 % CPU-ajasta. Tämä kertoo siitä, että taulukon käsittelyssä pullonkaulaksi muodostuu tietojen kopiointi uuteen taulukkoon eri operaatioissa, kuten lisäyksissä ja poistoissa. Yksittäisinä

operaatioina tarkasteltuna alkioiden lisäys sekä poisto alusta, keskeltä ja lopusta kuormittivat prosessoria tasaisesti noin 3–4 % verran.

Lista

Listan suorituskyky osoittautui yleisesti melko hyväksi, mutta se ei yltänyt suoritusnopeuden osalta kuitenkaan ihan taulukon tasolle (liite 2). Vaikka alkioiden lisääminen listan loppuun on yleensä tehokasta, alkuperäisessä testissä jokaisesta operaatiosta luotiin uusi lista jo olemassa olevasta listasta, mikä aiheutti lisätyötä roskienkeruulle ja muistinhallinnalle. Tämä ongelma korjattiin uudessa testiversiossa siten, että hakutoiminto suoritettiin suoraan olemassa olevaan listaan ilman uuden listan luontia. Tämän ansiosta *Search*-metodi ei enää aiheuttanut tarvetta lisämuistin varaukselle ja suorituskyky parani huomattavasti. *Delete*-operaatiot vaativat kuitenkin yhä uuden listan luomisen, koska testien luonteen vuoksi alkuperäistä listaa ei haluttu muokata. Listan *Add*-metodi tarvitsi enemmän muistivarausta kuin *Delete*-metodit, koska uuden arvon lisääminen voi vaatia listan taustalla olevan taulukon kasvattamista.

Visual Studio Performance Profilerilla saatiin tietoa muistinkäytöstä sekä prosessorin kuormituksesta. Testien perusteella *ListTests*-testeissä listan luonti kuormitti prosessoria selvästi eniten, noin 47–48 % kokonaisajasta. Toinen merkittävä CPU-resurssien käyttäjä oli listan taustalla toimivan taulukon automaattinen koon kasvatus (*AddWithResize*), jonka CPU-kuormitus oli 17–22 %:n luokkaa. Listalle suoritetut poisto- ja lisäysmetodit aiheuttivat yksittäisinä suorituksina vain pientä prosessorin kuormitusta verrattuna esimerkiksi uuden listan luontiin. Testiajoissa havaittiin myös muistinkäytön vaihtelua ohjelman suorituksen aikana: ensimmäisessä ajossa koko ohjelman muistinkulutus oli noin 33 megatavua, mutta toisessa ajossa se nousi noin 76 megatavuun. Muistinkulutuksen vaihteluun voi olla useita eri syitä, kuten roskienkeruun aktivoituminen ja testikoneen taustaprosessit.

6.2 Silmukat ja LINQ

Silmukka- ja LINQ-testeissä vertailtiin eri ohjelmointirakenteiden suorituskykyä kolmessa eri skenaariossa. Testeissä etsittiin alkioiden joukosta suurin arvo (*MaxValueTests*), laskettiin kaikki arvot yhteen (*SumTests*) ja etsittiin alkioiden joukosta parilliset arvot (*EvenNumberTests*). Suorituskykytestien aikana ohjelman muistinkäyttö pysyi hyvin vakaana kaikissa testiskenaarioissa. Performance Profilerin mittausten perusteella ohjelman muistinkäyttö oli noin 20 megatavua tasaisesti läpi suorituksen, ilman merkittäviä vaihteluita tai piikkejä muistinkäytössä.

Testien perusteella perinteiset silmukkarakenteet (for, foreach, while ja do-while) suoriutuivat testeistä lähes yhtä nopeasti (liite 3). Suoritusajoissa näiden rakenteiden välillä oli ainoastaan joitain kymmeniä mikrosekunteja, mikä on lähes mitätön ero käytännön kannalta. Silmukoiden osalta tulokset olivat odotetun kaltaisia, sillä rakenteena silmukat toimivat keskenään hyvin samankaltaisesti ja suoraviivaisesti.

LINQ oli selvästi silmukoita hitaampi arvojen yhteenlaskussa ja parillisten arvojen hakemisessa (liite 3). Kaikki testit käyttivät samaa *List<int>*-kokoelmaa, mutta suorituskykyeroja syntyi siitä, miten eri rakenteet käsittelevät listaa. Silmukoilla lista käydään läpi suoraan indeksin avulla, mikä on erittäin nopea tapa. Sen sijaan LINQ käyttää IEnumerable-rajapinnan enumeratoria, joka käy listan läpi yksi alkio kerrallaan. LINQ:lla lisäviivettä aiheuttaa lisäksi ylimääräiset funktiokutsut jokaiselle alkiolle. Tämä selittää, miksi LINQ oli hieman hitaampi näissä kahdessa testiskenaariossa silmukoihin verrattuna.

Hieman yllättäen LINQ oli kuitenkin huomattavasti nopeampi suurimman arvon hakemisessa kuin silmukkarakenteet. LINQ on saanut huomattavia parannuksia erityisesti *Max()*-metodiin .NET 7 versiossa, ja nykyään se on jopa 10 kertaa nopeampi kuin aiemmin. (Evtihevich 2022.) Lisäksi LINQ:n *Max()*-metodissa ei myöskään käytetä ylimääräisiä funktiokutsuja, vaan vertailu tehdään arvojen välillä suoraan. Nämä asiat vaikuttavat metodin suorituskykyyn positiivisesti ja ovat luultavasti suurin syy siihen, miksi LINQ:n *Max()*-metodi pärjäsi suorituskykytesteissä suurimman arvon hakemisessa näin hyvin.

Performance Profilerilla tehdyissä CPU-kuormaa ja muistinkäyttöä mittaavissa testeissä tulokset osoittivat, että kaikki silmukkarakenteet olivat CPU-kuormituksen osalta hyvin samankaltaisia. *EvenNumber*-testeissä silmukoiden kuormitus vaihteli välillä 14–17 %:n, *MaxValue*-testeissä 22–23 %:n ja *Sum*-testeissä 15–16 %:n välillä. Näiden perusteella voidaan todeta, että eri silmukkarakenteiden välillä ei ole käytännössä merkittäviä suorituskykyeroja suurten aineistojen käsittelyssä prosessorin kuormituksen näkökulmasta.

Sen sijaan LINQ:n *Count*- ja *Sum*-metodit lisäsivät CPU-kuormitusta huomattavasti parillisten arvojen haussa ja alkioiden yhteenlaskussa. *EvenNumber*-testeissä LINQ käytti keskimäärin 36 % CPU-resursseista, *Sum*-testeissä 32–33 % ja ehkä hieman yllätyksenä *MaxValue*-testeissä vain 6–7 %. *EvenNumber*-testien ja *Sum*-testien osalta tämä selittyy ylimääräisillä funktiokutsuilla, joita LINQ:n sisäinen rakenne toteuttaa. Ylimääräiset funktiokutsut aiheuttavat myös prosessorille ylimääräistä laskentakuormaa silmukkarakenteisiin verrattuna.

6.3 Useimmin esiintyvän ehdon järjestyksen vaikutus

Testeissä tutkittiin, onko ehtojen järjestyksellä vaikutusta ohjelman suorituskykyyn if-else- ja switch-case-rakenteissa. *BirthMonthTests*-testeissä etsittiin kuukauden nimiä satunnaisista syntymäkuukausista ja *AgeGroupTests*-testeissä luokiteltiin henkilöt eri ikäryhmiin. Testidatassa useimmin esiintyvä ehto, kesäkuu syntymäkuukausissa ja aikuiset ikäryhmissä, oli tarkoituksella asetettu esiintymään 60 prosentissa tapauksista.

Tulokset osoittivat, että if-else-rakenteessa suoritusaikojen erot olivat erittäin pieniä riippumatta siitä, sijoitettiinko todennäköisin ehto rakenteen alkuun, keskelle vai loppuun (liite 4). Suoritusnopeudet poikkesivat toisistaan vain muutaman millisekunnin verran, mikä selittyy mittaustarkkuuden sisäisillä vaihteluilla, eikä sillä ole käytännössä merkitystä ohjelman suorituksen kannalta. Sama tulos saatiin myös switch-case-rakenteen osalta. Useimmin

esiintyvän ehdon järjestyksellä ei havaittu olevan vaikutusta, vaan suoritusajat pysyivät lähes samoina ehdon järjestyksestä huolimatta.

Ainoa selvästi havaittava ero löytyi rakenteiden väliltä, ja if-else osoittautui testien perusteella nopeammaksi kuin switch-case. Tämä ero johtuu kuitenkin rakenteiden erilaisesta toimintalogiikasta, ei ehtojen järjestyksestä. If-else-rakenne suorittaa ehtoja järjestyksessä ylhäältä alaspäin ja päättyy heti ensimmäisen ”osuman” jälkeen. Switch-case taas vaatii enemmän ohjausrakenteita ja optimointimahdollisuudet ovat rajallisemmat.

Performance Profilerilla tehdyissä testeissä havaittiin, että switch-case-rakenteet kuormittivat prosessoria enemmän kuin if-else-rakenteet. *BirthMonth*-testeissä Performance Profiler ei löytänyt lainkaan if-else-rakenteiden suorituksia tulostaulukosta. Tämä viittaa siihen, että testissä käytetyt if-else-metodit olivat laskennallisesti niin kevyitä, että ne eivät päässeet erottumaan muiden suoritusten joukosta. Yksittäisten switch-case-metodien CPU-kuormitus oli näissä testeissä noin 8 %:n luokkaa mutta switch-case-rakenteiden kokonaiskuormitus oli noin 31 %. *AgeGroup*-testeissä if-else-metodi löytyi mittauksista, mutta sen CPU-kuormitus jäi vain noin 2 %:iin kokonaiskuormituksesta. Switch-casen yksittäiset metodit kuormittivat prosessoria saman verran kuin *BirthMonth*-testeissäkin, eli noin 8 %, mutta kokonaiskuormitus oli myös *AgeGroup*-testeissä korkeampi, jopa 46 %.

Ehtojen järjestyksellä ei siis ollut merkittävää vaikutusta ohjelman suoritusaikaan kummassakaan rakenteessa. Eroa oli vain joitain millisekunteja riippumatta siitä, missä useimmin esiintyvä ehto sijaitsi. Suorituskykytestien perusteella erot olivat niin pieniä, ettei niillä ole juuri merkitystä suorituskykyyn.

6.4 Ehtorakenteet ja rekursio

Opinnäytetyön tavoitteena oli vertailla rekursiivisten ratkaisujen suorituskykyä perinteisiin ehtorakenteisiin (if-else ja switch-case). Tutkimuksen tavoitteena oli erityisesti selvittää, kuinka suuri vaikutus rekursion luonteella on suorituskyvyn

kannalta verrattuna suoraviivaisempiin rakenteisiin. Tutkimuksen ja testauksen pääpaino oli ehtolauseiden ja rekursion vertailussa, mutta tutkimukseen päätettiin tehdä vielä yksi lisätesti, joka arvioi rekursion suorituskykyä sille tyypillisemmässä tehtävässä eli binääripuun läpikäynnissä. Tätä varten toteutettiin erillinen *SearchRecursiveIterative*-testiluokka, jossa oli sekä rekursiivinen että iteratiivinen hakutoiminto binääripuulle. Testien avulla haluttiin nähdä, miten rekursio suoriutuu sille niin sanotusti luonnollisemmasta tehtävästä.

Testeissä vertailtiin kolmen erilaisen ohjelmointirakenteen suorituskykyä: if-else, switch-case ja rekursio. *GetGrade*-testeissä taulukon arvoille annettiin arvosanat ehtojen perusteella. Tulokset osoittivat, että if-else ja switch-case olivat suoritusajaltaan lähes yhtä nopeita, suoritusaikojen jäädessä noin 225 mikrosekuntiin (liite 5). Sen sijaan rekursio oli selkeästi hitaampi, suoritusaikojen ollessa yli 5 000 mikrosekuntia. Eron suurin syy on rekursion toimintatavassa. Jokainen rekursiivinen ehtotarkistus kutsuu metodia uudelleen, ja jokainen kutsu vaatii hieman lisää muistia ja prosessoritehoa. Kun käsiteltäviä arvoja on paljon, laskennallinen kuorma kasvaa nopeasti. Toisessa testissä (*SearchRecursiveIterative*) verrattiin iteratiivista ja rekursiivista hakualgoritmia binääripuussa (liite 5). Ensimmäisellä testikerralla iteratiivinen haku oli hieman nopeampi, mutta toisessa mittauksessa rekursiivinen haku oli nopeampi. Erot olivat kuitenkin niin marginaalisia, että ne selittyvät koneen taustaprosessien aiheuttamilla vaihteluilla.

Performance Profilerilla tehdyissä mittauksissa saatiin tietoa ehtorakenteiden ja rekursion prosessorin kuormituksesta. Testien perusteella tulos oli hyvin selkeä, ja ehtorakenteet kuluttivat huomattavasti vähemmän prosessorin resursseja rekursioon verrattuna. Esimerkiksi *GetGrade*-testeissä rekursiivinen ratkaisu käytti melkein 90 % prosessorin laskentatehosta, kun if-else ja switch-case käyttivät kumpikin vain noin 4 %. *SearchRecursiveIterative*-testissä binääripuu täytettiin 200 000 arvolla, ja sekä iteratiivinen että rekursiivinen haku ajettiin 10 000 kertaa per suorituskerta. Tulokset osoittivat, että rekursio käytti edelleen hieman enemmän CPU-resursseja (42–44 %) kuin iteratiivinen ratkaisu (34–37 %), mutta tässä testissä ero ei ollut kovin merkittävä.

6.5 Suorituskyvyn mittaaminen

Tässä tutkimuksessa suorituskykyä mitattiin BenchmarkDotNet-kirjaston sekä Visual Studio Performance Profilerin avulla. BenchmarkDotNetin avulla saatiin tarkkaa tietoa suoritusajoista sekä muistivarauksista. Visual Studio Performance Profiler täydensi tuloksia mittaamalla prosessorin kuormitusta ja ohjelman muistinkäyttöä. Molemmat työkalut täydensivät toisiaan, ja niiden käyttäminen yhdessä muodosti luotettavan kokonaisuuden suorituskyvyn analysoimiseksi.

Testiajoissa hyödynnettiin BenchmarkDotNet-kirjaston *warmup-, iteration-* ja *invocation count* -asetuksia, joilla pyrittiin varmistamaan mittausten toistettavuus ja luotettavuus. Jokaisessa testissä ajettiin vähintään viisi lämmittelykierrosta (*warmup*), joiden tarkoituksena oli vakauttaa .NET-ympäristön suorituskyky ennen varsinaisia mittauksia. Varsinaisten mittausten määrä vaihteli 10–15 mittauskierroksen välillä (*iterations*). Yksittäisellä mittauskierroksella testimetodeja kutsuttiin useita kertoja (*invocations*) mittaustarkkuuden parantamiseksi, invocation count -asetukset vaihtelivat 1 024–4 096 välillä. Lisäksi osassa testeistä käytettiin vähimmäissuoritusaikaa (*MinIterationTime*), joka varmisti riittävän pitkän testisuorituksen satunnaisvaihtelun minimoimiseksi.

Performance Profiler tarjosi tietoa koko ohjelman muistinkäytöstä sekä eri metodien CPU-ajasta. Performance Profiler esitti mittaustulokset selkeästi visuaalisena kuviona sekä listana, jossa eniten prosessoria kuormittavat metodit oli listattu CPU-ajan perusteella suurimmasta pienimpään.

Kokonaisuudessaan suorituskykyä mitattiin mahdollisimman luotettavasti hyödyntämällä kahta toisiaan täydentävää työkalua, toistamalla testit kahteen kertaan ja kontrolloimalla testausympäristö mahdollisimman vakioiduksi. Tulokset analysoitiin vertaamalla niitä sekä keskenään että aiempaan teoriaan, jolloin saatiin kokonaiskuva suorituskyvystä eri näkökulmista. Molemmat testikerrat olivat tulosten osalta hyvin samankaltaisia, eikä suuria heittoja tai eroja saatu muistinkäytön, muistivarauksen, suoritusnopeuden tai CPU-

kuormituksen osalta testikertojen välille. Poikkeuksena tässä oli tietorakenteiden osalta lista, jonka muistinkulutuksessa oli vaihtelua testikertojen välillä reilu 40 megatavua.

7 Pohdinta

7.1 Tutkimustulosten ja teorian vertailu

Tässä opinnäytetyössä keskityttiin vertailemaan erilaisten C#-ohjelmointikielen tietorakenteiden ja ohjelmointirakenteiden suorituskykyä teoreettisen tietoperustan ja konkreettisten suorituskykytestien perusteella.

Tutkimuksessa vastattiin asetettuihin tutkimuskysymyksiin suorituskykytestien ja tulosten analyysin avulla. Saatujen tulosten pohjalta voidaan esittää hyviä havaintoja, kriittisiä arvioita sekä jatkokehitysideoita.

7.1.1 Sanakirja

Tutkimuksen teoreettisen viitekehyksen mukaan sanakirja on tehokas tietojen hakutoiminnoissa ja sen tehokkuus perustuu sanakirjan taustalla toimivaan hajautustauluun. (Khan 2018, Data Structures and Writing Optimized Code In C#). Myös tämän tutkimuksen tulosten perusteella sanakirja oli odotetusti tietorakenteista nopein hakuoperaatioissa, ja tätä tulkintaa vahvistivat myös BenchmarkDotNetin testitulokset.

Sen sijaan lisäys- ja poisto-operaatiot olivat selkeästi hitaampia ja muistin kannalta hieman raskaampia, mikä osittain selittyy testikoodin rakenteella, jossa uusi sanakirja luotiin jokaisen operaation yhteydessä. Tulosten pohjalta voidaan tehdä päätelmiä siitä, että suurin osa sanakirjan prosessorin kuormituksesta ei syntynyt yksittäisistä lisäys- tai poistometodeista, vaan taustalla tapahtuvasta sanakirjan kopioinnista ja alustuksesta, jotka vaativat merkittävästi CPU-aikaa.

Tulokset osoittivat, että sanakirja on rakenteellisesti tehokas, mutta sen kokonaiskuormitus voi kasvaa huomattavasti riippuen siitä, miten sitä käytetään ja alustetaan testitilanteessa.

7.1.2 Taulukko

Testitulokset osoittivat, että taulukko oli nopein tietorakenne kaikissa muissa operaatioissa, paitsi tietojen haussa, jossa sanakirja vei voiton. Teoriassa taulukon heikkoutena pidetään sen kiinteää kokoa sekä sitä, että poistoissa ja lisäyksissä alkioiden siirto rakenteen keskeltä tai lopusta heikentää suorituskykyä. (Khan 2018, Data Structures And Writing Optimized Code In C#). Testiskenaariot eivät kuitenkaan vahvistaneet tätä havaintoa, vaan taulukko suoriutui itseasiassa alkioiden lisäyksistä ja poistoista kaikista tietorakenteista nopeimmin, riippumatta siitä, poistettiinko alkio rakenteen alusta, keskeltä vai lopusta. Tuloksista esiin nouseva taulukon hyvä suorituskyky suoritusnopeuden osalta antaa viitteitä siitä, että yksinkertainen tietorakenne voi olla erittäin tehokas, kunhan käyttötapaukset valitaan oikein.

Tulosten pohjalta voidaan tehdä havaintoja myös *Array.Copy*-metodin todellisesta tehokkuudesta. Vaikka metodi on itsessään tehokas, prosessorin kokonaiskuormitus kasvaa silloin kun sitä käytetään usein. Esimerkiksi kun rakenteen muokkaus vaati taulukon uudelleenkopiointia lähes jokaisessa lisäys- ja poisto-operaatiossa, se näkyi CPU-kuormituksen kasvuna. Yhteenvetona voidaan todeta, että taulukoiden suorituskyky on kohtuullisen hyvä suoritusaikojen osalta, mutta etenkin alkioiden lisäämisessä ja poistamisessa tarvitaan koko taulukon kopiointia, mikä näkyy CPU-kuormituksen kannalta kalliina *Array.Copy*-kutsuina.

7.1.3 Lista

Listojen kohdalla suorituskykytestien tulokset tukivat osittain aiemmin teoriassa esiin tulleita havaintoja. Teoriassa nostettiin esiin erityisesti se, miten listat

tarjoavat tietorakenteena dynaamisuutta ja joustavuutta, mutta voivat kuormittaa muistia erityisesti silloin, kun kokoelmaa kasvatetaan tai operoidaan listan keskellä. (Khan 2018, Data Structures And Writing Optimized Code In C#). Tulokset osoittivat, että suoritusnopeuden osalta lista ei pärjännyt taulukolle missään operaatiossa, mutta se oli kuitenkin huomattavasti nopeampi lisäyksissä ja poistoissa kuin sanakirja.

Muistivarauksen osalta listan ja taulukon tulokset samankaltaiset, ja molempien varaama muisti oli 3.81 megatavua. Performance Profilerin mukaan koko ohjelman muistikäyttö kuitenkin vaihteli huomattavasti enemmän listan tapauksessa (33–76 MB), mikä viittaa esimerkiksi roskienkeruun vaikutuksiin. Tätä vaihtelua ei havaittu samassa määrin taulukon ja sanakirjan kohdalla, mikä tukee osittain aiempaa näkemystä siitä, että lista voi olla muistinkäytön kannalta raskaampi vaihtoehto erityisesti kasvavissa rakenteissa. (Khan 2018, Data Structures And Writing Optimized Code In C#). Tuloksista voidaan tehdä johtopäätöksiä siitä, että vaikka lista on usein hyvä kompromissi suoritusnopeuden ja joustavuuden osalta, muistin hallinnan kannalta taulukko voi olla varmempi valinta, mikäli tietorakenteen koko on ennustettavissa ja muuttuu harvoin.

7.1.4 Silmukat ja LINQ

Silmukkarakenteiden osalta teoria piti erinomaisesti paikkansa: kaikki testatut silmukat (for, foreach, while ja do-while) suoriutuivat testeistä hyvin tasaisesti ja tehokkaasti. Käytännössä rakenteiden erot sekä suoritusnopeuden että prosessorin kuormituksen osalta olivat niin vähäisiä, että niillä ei voida katsoa olevan juuri merkitystä ohjelman toimintaan näiden testien perusteella. Tulokset kuitenkin vahvistivat niitä käsityksiä, joita teoriasta oli havaittu. Kuten Zakas (2010) toi ilmi, for-, foreach-, while- ja do-while-silmukat ovat suorituskyvyltään erittäin samankaltaisia (Zakas 2010, luku 4). Saatujen tulosten perusteella on mahdotonta laittaa silmukoita suorituskyvyn näkökulmasta niin sanottuun paremmuusjärjestykseen. Silmukoiden valinnassa on suositeltavaa huomioida ohjelman käyttötarkoitus ja tarkastella ohjelmaa kokonaisuutena. Koska

suorituskykyerot silmukoiden välillä olivat vähäisiä, rakenteen valinnassa kannattaa miettiä esimerkiksi koodin luettavuutta ja sitä, mikä rakenne sopii parhaiten työn alla olevaan ongelmaan.

Sen sijaan LINQ:n kohdalla tulokset olivat yllättäviä ja kiinnostavia. Vaikka teoriassa LINQ:lla on hidas ja tehoton maine, testit osoittivat, että sen suorituskyky on vahvasti riippuvainen siitä, miten LINQ:a käytetään. (Vickers 2020.) Suurimman arvon hakemisessa käytetty LINQ:n *Max()*-metodi oli yllättäen selvästi nopeampi kuin silmukkarakenteet mutta *Count*- ja *Sum*-metodit olivat silmukoita selvästi hitaampia sekä myös prosessorin kuormituksen osalta raskaampia. *Max()*-metodi on saanut uusimmissa .NET-versioissa parannuksia, jotka kävivät ilmi myös testituloksista. LINQ:n *Max()*-metodi suorittaa ainoastaan arvojen vertailun, mikä on laskennallisesti kevyt operaatio. Tämä näkyi tuloksissa huomattavasti pienempänä CPU-kuormituksena LINQ:n muihin metodeihin verrattuna.

Testit osoittivat, että perinteiset silmukat (for, foreach, while ja do-while) ovat tasaisen vahvoja suorituskyvyn näkökulmasta ja soveltuvat erityisesti isompien tietomäärien käsittelyyn. LINQ tarjoaa kehittäjille ehkä hieman selkeämmän ja helpommin luettavan koodin, mutta etenkin suurien tietomäärien kanssa sen suorituskyky voi jäädä perinteisiä silmukoita heikommaksi riippuen LINQ:n käyttämistä metodeista. Poikkeuksena kuitenkin havaittiin, että valmiit LINQ:n funktiot, kuten suurimman arvon etsiminen *Max()*-metodin avulla, voivat tietyissä tapauksissa olla suorituskyvyn näkökulmasta jopa nopeampia kuin silmukat.

Tulosten perusteella voidaan todeta, että LINQ:n suorituskyky on siis vahvasti riippuvainen siitä, mitä metodeja käytetään. Osa LINQ:n toiminnoista on hyvin optimoituja, kun taas toiset aiheuttavat lisää kuormaa etenkin prosessorille, esimerkiksi ylimääräisten funktiokutsujen vuoksi. Tuloksista voidaan tehdä johtopäätöksiä siitä, että LINQ:a ei kannata suoraan leimata silmukoita huonommaksi vaihtoehdoksi, mutta sen toimintalogiikka on tärkeää ymmärtää ennen rakenteen valintaa. Testaamalla eri rakenteiden suorituskykyä saadaan viime käden tietoa suoritusajoista, muistinkäytöstä sekä prosessorin

kuormituksesta. Testien jälkeen kehittäjän on helpompi tehdä päätöksiä siitä, millaisia rakenteita ohjelmassa kannattaa käyttää.

7.1.5 Useimmin esiintyvän ehdon järjestyksen vaikutus

Ehtorakenteiden osalta tutkittiin, onko useimmin esiintyvät ehdon järjestyksellä vaikutusta suorituskykyyn. Testit tuottivat jopa yllättävän selkeän tuloksen, kun ehtojen järjestyksellä ei havaittu olevan vaikutusta suorituskykyyn. Vaikka teoriassa haarautumisen ennustamisen ja spekulatiivisen suorituksen on oletettu vaikuttavan, mittaustulokset kuitenkin osoittivat, että vaikutus on lähes olematon tai liian pieni havaittavaksi näissä testiskenaarioissa. (Kocher ym. 2019, luku 1.) Vaikka ehdon järjestyksellä ei havaittu olevan merkitystä, ehtorakenteen valinnalla sen sijaan oli merkittävä vaikutus *AgeGroup*- ja *BirthMonth*-testeissä sekä suoritusaikaan että prosessorin kuormitukseen. Suorituskykyä tarkastellessa erot suoritusnopeudessa if-elsen ja switch-casen välillä ovat teoreettisesti merkittäviä, kun if-else on noin 25 kertaa nopeampi, mutta käytännön eroista ei voida tehdä täysin yksiselitteistä johtopäätöstä. Mikäli operaatio suoritetaan vain muutamia kertoja, ei eroja suoritusnopeudessa huomaa todennäköisesti ohjelman käytössä. Jos taas operaatio toistetaan esimerkiksi 1 000 kertaa silmukassa, ero voi olla jo yli 50 sekuntia, jolloin myös käytännön suorituskykyeroja voidaan pitää if-elsen ja switch-casen välillä merkittävänä.

Performance Profilerilla tehdyt testit antoivat viitteitä siitä, että if-else on kevyempi vaihtoehto prosessorin kuormituksen kannalta. If-else-rakenteet kuormittivat prosessoria huomattavasti vähemmän, kun switch-case, mutta koska *BirthMonth*-testeistä ei saatu mittausarvoja if-else-rakenteesta, tulosta ei voida pitää täysin varmana, vaikka molempien testien havainnot tähän viittaavat. Oletettavasti if-else-rakenteiden CPU-kuormitus oli niin kevyttä, että se ei näkynyt Performance Profilerin tuloksissa ollenkaan.

Tuloksista ei voida kuitenkaan tehdä yksiselitteistä tulkintaa siitä, että if-else olisi aina nopeampi ja parempi ratkaisu. Kun *GetGrade*-testeissä if-elseä ja

switch-casea verrattiin rekursiivisen ratkaisun kanssa, if-elsen ja switch-casen suoritukset olivat erittäin tasaisia niin suoritusajan kuin prosessorin kuormituksenkin osalta. Tämä selittyy luultavasti *GetGrade*-testien yksinkertaisemmalla rakenteella, kun testissä käsitellään vain viittä loogista ehtoa, jotka etenevät suoraviivaisesti järjestyksessä ja arvosanojen jakauma on tasaisempi. Aluksi erojen epäiltiin johtuvan siitä, että if-elsen ja switch-casen vertailussa esimerkiksi *AgeGroup*-testeissä oli käytössä apumetodi, mutta *BirthMonth*-testeissä ei tällaista ylimääräistä apumetodia ollut, joten erot suorituskyvyssä eivät voineet johtua ainakaan pelkästään tästä.

Yksi mahdollinen syy löytyi Tanin ym. (2012) tutkimuksesta, jossa todettiin, että kääntäjä voi optimoida switch-case-rakenteen taulukkomuotoon (*jump table*). Tätä kutsutaan epäsuoraksi haarautumiseksi. Epäsuora haarautuminen voi vaikeuttaa prosessorin haarautumisen ennustamista, kun seuraava suorituslohko määräytyy taulukkoindeksin perusteella eikä lineaarisesti niin kuin if-else -rakenteessa. Tämä voi kuitenkin tehdä switch-case-rakenteesta nopeamman, mutta toisaalta myös lisätä prosessorin kuormitusta. (Tan, Liu, Zhang, Tong & Cheng 2012.) Asian varmistaminen ei kuitenkaan ole kovin yksinkertaista, mutta koska *AgeGroup*- ja *BirthMonth*-testeissä haaroja oli reilusti enemmän ja jakauma oli epätasainen, syntyy vahva epäily siitä, että epäsuora haarautuminen on ollut käytössä. Vaikka näiden ehtolauseiden keskinäinen vertailu ei ollut tutkimuksen keskiössä, on näin selkeät erot suorituskyvyssä kuitenkin hyvä ottaa huomioon, kun tehdään päätöksiä ohjelman rakenteesta.

7.1.6 Ehtorakenteet ja rekursio

If-elseä ja switch-casea verrattiin viimeisissä testeissä myös rekursion kanssa. Rekursion osalta tulokset vastasivat teoriassa esiin nousseita oletuksia: rekursio oli selkeästi hitaampi ja raskaampi perinteisiin ehtorakenteisiin verrattuna. Rubio-Sanchez nosti kirjassaan esiin sen, että rekursiiviset ratkaisut eivät välttämättä ole kovin tehokkaita (Rubio-Sanchez 2017, luku 1.5). Tämä väite piti *GetGrade*-testeissä paikkansa. *GetGrade*-testiskenaarioissa

taulukossa oleville arvoille haettiin niitä vastaavat arvosanat annettujen ehtojen mukaisesti. Rekursion hitaus *GetGrade*-testeissä selittyy sillä, että jokainen ehtotarkistus aiheuttaa uuden metodikutsun, mikä kasvattaa kutsupinoa nopeasti. .NET ei myöskään tee automaattisesti häntärekursio-optimointeja, mikä tarkoittaa sitä, että jokainen kutsu vie enemmän tilaa pinosta. Testiskenaario ei ollut rekursiolle kovin ”tyypillinen” eikä rekursio varmasti ollut näissä testeissä paras tämänkaltaisen ongelman ratkaisemiseen. Tämä tuli ilmi suurena prosessorin kuormituksena sekä hitaampana suoritusaikana.

Toinen testiskenaario oli rekursiolle luonnollisempi ja tyypillisempi. Siinä suoritettiin tietojen haku binääripuusta sekä rekursiivisesti että iteratiivisesti. Näissä testeissä rekursiivinen ja iteratiivinen ratkaisu toimivat molemmat erittäin samankaltaisesti, sillä molempien suoritusnopeus sekä prosessorin kuormitus olivat hyvin lähellä toisiaan. Binääripuun haussa rekursio pääsi siis toimimaan niin sanotusti rekursiolle tyypillisessä tehtävässä, ja rekursio on enemmän ”luonnollisempi” tapa toteuttaa puun läpikäynti. Jokaisessa solmussa tehdään sama operaatio ja siirrytään vasempaan tai oikeaan alisolmuun, puun toimintalogiikka sopii hyvin rekursion perusperiaatteisiin, jossa ongelma pilkotaan pienempiin samankaltaisiin osiin. Voidaan siis sanoa, että *GetGrade*-testeissä rekursiota käytettiin sellaiseen tehtävään, johon se ei sovellu kovin hyvin, kun taas binääripuun haussa rekursio ikään kuin vastasi ongelman rakennetta. Tämän vuoksi erot rakenteiden välillä suoritusajan sekä prosessorin kuormituksen osalta jäivät erittäin vähäisiksi *SearchRecursiveIterative*-testissä.

GetGrade-testeissä rekursio kulutti huomattavasti enemmän prosessoritehoa verrattuna if-else- ja switch-case-rakenteisiin, jotka suoriutuivat samasta tehtävästä hyvin pienellä prosessorin kuormituksella. Tämä vahvistaa ajatusta siitä, että kun on tarve käsitellä suuria tietomääriä tai käyttää paljon toistoa, perinteiset ehtorakenteet, kuten if-else ja switch-case, ovat nopeampia ja kevyempiä prosessorin kannalta rekursiiviseen toteutukseen verrattuna. Toisaalta jos kyseessä rekursiolle luonteva käyttökohde, rekursiivinen ratkaisu voi tarjota kuitenkin lähes yhtä hyvän suorituskyvyn kuin iteratiivinen ratkaisu.

Johtopäätöksenä voidaan todeta, että rekursio on varsin käyttökelpoinen ratkaisu pienissä ja selkeästi rajatuissa ongelmissa, kuten binääripuun haussa, jossa se tarjoaa luonnollisen tavan toteuttaa hakualgoritmi. Tutkimuksessa tehty lisätesti, jossa verrattiin binääripuun rekursiivista ja iteratiivista hakua osoitti kuitenkin, että myös tällaisessa rekursiolle tyypillisemmässä käyttötarkoituksessa rekursiivinen ratkaisu kuormitti prosessoria hieman enemmän. Tämä selittyy rekursion sisäisellä toimintalogiikalla, jossa rekursio kasvattaa kutsupinoa jokaisella funktiokutsulla, mikä kuormittaa prosessoria enemmän kuin yksinkertainen silmukkarakenne, joka ei käytä kutsupinoa samalla tavalla.

7.2 Suorituskyvyn mittaamiseen vaikuttavat tekijät

Vaikka tutkimuksessa pyrittiin toteuttamaan mittaukset mahdollisimman huolellisesti, on tärkeää tarkastella myös mittausten rajoitteita ja haasteita kriittisesti. Teoriassa suositellaan, että suorituskykymittaukset toistetaan vähintään 10 kertaa, jotta yksittäisistä ajoista johtuvat satunnaisvaihtelut voidaan sulkea pois (Oaks 2014, luku 2). Tässä tutkimuksessa varsinaiset mittaukset tehtiin kahteen kertaan kahtena eri päivänä, mikä parantaa tulosten luotettavuutta verrattuna yksittäisiin ajoihin, mutta suositeltu toistomäärä jää kuitenkin kauas tästä.

Yksi tutkimuksen vahvuus oli kuitenkin se, että tutkimuksessa käytettiin kahta eri mittaustyökalua, joista BenchmarkDotNet on suunniteltu erityisesti tarkkaan suorituskykymittaukseen. BenchmarkDotNetin ominaisuudet, kuten lämmittelyvaiheen automaattinen huomioon ottaminen ja poikkeamista ilmoittaminen, vähentävät mittausvirheitä. Samalla on kuitenkin huomioitava, että BenchmarkDotNetin mahdollistama tarkkuus mittauksissa ei kuitenkaan poista tarvetta toistaa testit riittävän monta kertaa.

Visual Studio Performance Profiler puolestaan tarjosi laajemman näkökulman ohjelman kokonaissuorituskykyyn. Sen avulla pystyttiin tarkastelemaan muun muassa prosessorin kuormitusta ja yksittäisten funktioiden aiheuttamaa CPU-

kuormitusta koko ohjelman suorituksen ajalta. Tämä näkökulma oli erityisen hyödyllinen esimerkiksi rekursion ja ehtorakenteiden vertailussa. Performance Profilerista saadut testitulokset täydensivät BenchmarkDotNetin tuloksia, mutta myös sen kohdalla tuloksia on tarkasteltava kriittisesti. Raskas prosessorin kuormitus ei välttämättä tarkoita käytännössä merkittävää viivettä ohjelman suoritukseen, mikäli esimerkiksi ajettava koodi on suoritusajaltaan äärimmäisen nopea.

Mittausympäristön hallinta oli tutkimuksessa hyvin toteutettu, kun turhat prosessit suljettiin ja testit ajettiin mahdollisimman vakaassa tilassa. Tästä huolimatta mittauksiin on silti voinut vaikuttaa esimerkiksi prosessorin haarautumisen ennustaminen tai spekulatiivinen suoritus, jotka tapahtuvat täysin automaattisesti ja joihin ei pystytä testauksissa vaikuttamaan (Kocher ym. 2019). Nämä taustaprosessit voivat aiheuttaa satunnaisvaihtelua tuloksiin, erityisesti mitattaessa ohjelmanosia, joiden suoritusaika on hyvin lyhyt.

Testikoodi ei sisältänyt virheenkäsittelyä tai satunnaisuutta, mikä on hyvä käytäntö suorituskykymittauksissa. BenchmarkDotNetin mittauksissa havaittiin, että roskienkeruu aktivoitui joissain testeissä. Kirjasto ottaa roskienkeruun huomioon testeissä, mutta tämän tutkimuksen painopiste ei ollut roskienkeruun analysoinnissa. Tämän vuoksi ei voida tehdä tarkempia johtopäätöksiä siitä, miten paljon roskienkeruu vaikutti yksittäisten testien suoritusaikoihin tai muistinkäyttöön. Mittaustuloksiin voi sisältyä pieniä vaihteluita roskienkeruun aktivoitumisen seurauksena, mutta tarkempia vaikutuksia näissä testeissä on vaikea osoittaa kyseisessä kontekstissa.

7.3 Luotettavuus ja eettisyys

Tämän tutkimuksen luotettavuus perustuu huolelliseen suunnitteluun, tiedonhankintaan sekä analyysiin, jotka on toteutettu laadullisen tutkimuksen periaatteiden mukaisesti. Laadullisessa tutkimuksessa luotettavuutta voidaan mitata erityisesti tutkimuksen uskottavuuteen perustuen. Tutkimuksessa käytetään useita eri lähteitä ja verrataan suorituskykytestien tuloksia aiheesta jo

olemassa oleviin tutkimuksiin ja kirjallisuuteen. Näin voidaan varmistaa, että tulokset ovat mahdollisimman luotettavia eivätkä perustu sattumanvaraisuuteen. (Merriam & Tisdell 2015, luku 9.) Tutkimuksen luotettavuuden kannalta on tärkeää myös osata jättää omat ennakkoasenteet ja odotukset syrjään ja olla itsekriittinen tutkimuksen teossa (Vilkka 2021, 94).

Tutkimuksen luotettavuutta tarkasteltaessa on tärkeää huomioida paitsi mittausmenetelmien tarkkuus ja toistettavuus, myös koko tutkimusprosessin avoimuus, johdonmukaisuus ja läpinäkyvyys. Tässä opinnäytetyössä suorituskykytestien suunnittelu, toteutus ja analyysi on tehty tietoisesti niin, että mahdollisimman monet ulkoiset tekijät on rajattu testauksen ulkopuolelle. Tällä on pyritty varmistamaan, etteivät esimerkiksi taustalla olevat prosessit vaikuta testituloksiin. Testikoodi on toteutettu alusta loppuun itse, ja toteutus on tehty sellaiseksi, että testien toistaminen ja kooditiedostojen lataaminen GitHubista on tehty mahdollisimman helpoksi. Testikoodit on lisensoitu MIT lisenssillä, joka tarjoaa laajat oikeudet testauksessa käytettyjen koodien käyttöön.

Testien toistettavuutta pyrittiin vahvistamaan ajamalla kaikki testit kahteen kertaan eri ajankohtina ja vertaamaan tuloksia keskenään. Erojen ollessa pieniä voidaan pitää todennäköisenä, että mittaukset ovat kohtuullisen luotettavia. Lisäksi mittauksessa käytetyt työkalut, BenchmarkDotNet-kirjasto sekä Performance Profiler -lisäosa tarjosivat luotettavan testausympäristön testien ajoa varten. Esimerkiksi BenchmarkDotNet-testeissä ajettiin ensin aina käyttäjän itse määrittelemä määrä lämmittelykierroksia, jotta kaikki optimoinnit on varmasti suoritettu ennen varsinaisia testiajoja. Tämän lisäksi BenchmarkDotNet varoitti käyttäjää, jos mittaustulosten luotettavuus on kärsinyt esimerkiksi liian lyhyen testiajon vuoksi.

Suorituskyvyn mittaamisen osalta voidaan kuitenkin todeta, että vaikka tutkimuksen tulokset antavat kattavan kuvan tietorakenteiden ja ohjelmointirakenteiden suorituskyvystä, mittausten toistojen vähäisyys rajoittaa tulosten yleistettävyyttä. Mikäli tutkimus toistettaisiin useammalla koneella, useampina ajankohtina ja suuremmalla määrällä testikierroksia, luotettavuus ja tulosten yleistettävyys paranisivat merkittävästi. Näin ollen mittauksia voidaan

pitää suuntaa antavina ja käytännön kannalta hyödyllisinä, mutta ei täysin kattavina.

Tässä tutkimuksessa eettisyydestä pyrittiin huolehtimaan muun muassa niin, että tutkimuksessa ei pyritty osoittamaan pelkästään ennalta oletettuja tuloksia todeksi, vaan myös ristiriitaiset ja yllättävät tulokset esitettiin ja analysoitiin avoimesti. Erityisenä mainintana tässä tutkimuksessa otettiin osittain huomioon myös saavutettavuus, mikä jää usein teknisissä tutkimuksissa vähemmälle huomiolle. Suorituskykytestien visuaalisessa esittämisessä käytettiin värikoodeja; vihreällä paras tulos ja punaisella heikoin; mutta niiden rinnalle otettiin myös symbolit kuvaamaan parasta ja heikointa tulosta, jotta myös esimerkiksi värisokeat pystyvät lukemaan tuloksia ongelmitta. Tämä tukee saavutettavuusperiaatteita (*WCAG 2.1*), joissa korostetaan tietosisällön saatavuutta eri käyttäjäryhmille niin, että esimerkiksi värikoodaus ei ole ainoa keino välittää informaatiota. (W3C 2024.)

7.4 Jatkokehitysideat

Tämä opinnäytetyö keskittyi rajattujen tietorakenteiden ja ohjelmointirakenteiden suorituskyvyn tarkasteluun C#-kielessä. Tulosten ja pohdinnan pohjalta nousi esiin useita jatkokehitysideoita, joita tulevissa tutkimuksissa voitaisiin hyödyntää tarkemman ja laajemman suorituskykyanalyysin tekemiseksi:

- Tietorakenteiden laajempi analysointi

Tässä tutkimuksessa tarkasteltiin kolmea yleistä tietorakennetta: taulukkoa, listaa ja sanakirjaa. Jatkossa tutkimusta voisi laajentaa koskemaan muita tietorakenteita, kuten esimerkiksi jonoa, pinoa, linkitettyä listaa ja niin edelleen. Näillä tietorakenteilla on erilainen toimintalogiikka ja erilaisia ominaisuuksia suorituskyvyn kannalta, joten tietorakenteiden vertailu voisi tuoda syvällistä analyysia oikean rakenteen valinnasta eri ohjelmointitilanteissa.

- Tutkimuksen laajentaminen muihin ohjelmointikieliin

Yksi mielenkiintoinen jatkotutkimuksen aihe voisi olla vastaavien suorituskykytestien tekeminen jollakin toisella ohjelmointikielellä, kuten esimerkiksi Javalla tai Pythonilla. Tämä mahdollistaisi mielenkiintoisen vertailun siitä, miten eri ohjelmointikieli vaikuttaa esimerkiksi muistinkäyttöön, silmukoiden suorituksiin ja rekursion optimointiin. Olisi myös mielenkiintoista tietää, toimivatko .NET optimoinnit eri tavalla eri kielissä.

- Roskienkeruun huomioon ottaminen suorituskyvyn mittauksessa

.NET:n eri versiot sisältävät jatkuvia parannuksia muun muassa roskienkeruun toimintaan, ja nämä parannukset vaikuttavat suoraan suorituskykyyn ja muistinhallintaan. Yksi hyvä tulevaisuuden jatkokehitysidea voisi olla tarkempi keskittyminen siihen, millainen vaikutus roskienkeruulla on suorituskykyyn. Tutkimuksessa voisi selvittää myös millaisia optimointeja eri .NET -versiot ovat tuoneet roskienkeruun osalta.

- Algoritmien aikavaativuus vs. käytännön suorituskyky

Tässä tutkimuksessa keskityttiin pääasiassa käytännön suorituskykyyn BenchmarkDotNetin ja Visual Studio Performance Profilerin avulla. Yksi hyödyllinen ja mielenkiintoinen jatkotutkimus voisi olla käytännön tutkimuksen ja algoritmien aikavaativuuden vertailu. Mikäli käytännön tieto yhdistettäisiin teoreettiseen näkökulmaan ja arvioitaisiin algoritmien aikavaativuutta esimerkiksi Big O -notaatiolla, voitaisiin tutkia, missä määrin teoreettinen aikavaativuus vastaa mitattua suorituskykyä ja missä tilanteissa syntyy poikkeamia.

- Modernien prosessorien optimointien vaikutukset suorituskykyyn

Tutkimusta voisi jatkossa laajentaa tarkastelemalla myös prosessorin optimointeihin liittyviä ilmiöitä, kuten haarautumisen ja epäsuoran haarautumisen ennustamista sekä spekulatiivista suoritusta. Nämä optimoinnit vaikuttavat erityisesti ehtorakenteiden ja silmukoiden suoritustehoon

nykyaikaisilla prosessoreilla, joten näiden optimointien huomioonottaminen suorituskyvyn arvioinnissa voisi tuoda tutkimukseen mielenkiintoisen uuden näkökulman.

7.5 Tutkimuksen hyödynnettävyys

Tästä tutkimuksesta saadut tulokset ovat merkityksellisiä erityisesti ohjelmistokehityksen näkökulmasta, sillä tutkimus tarjoaa konkreettista tietoa C#-kielen eri tietorakenteiden ja ohjelmointirakenteiden suorituskyvystä. Monet ohjelmoijat tekevät valintoja rakenteiden välillä omaan aiempaan kokemukseen tai teoreettiseen tietoon perustuen, mutta tämän tutkimuksen suorituskykytestit tuovat aiheeseen käytännön näkökulman, jota voi hyödyntää sovelluskehityksessä. Ammatillisessa kontekstissa tutkimus on hyödyllinen ja toimii tukena esimerkiksi ohjelmistoalan opiskelijoille ja kehittäjille. Tutkimus tarjoaa perustellun ja testatun näkemyksen siitä, millaisiin tilanteisiin tietyt rakenteet soveltuvat parhaiten suorituskyvyn näkökulmasta.

Yhteiskunnallisesti tutkimuksen merkitys näkyy siinä, että tehokas ohjelmakoodi on yhä tärkeämpää esimerkiksi resurssienhallinnan ja energiatehokkuuden näkökulmasta. Tämä on tärkeää esimerkiksi silloin, kun samaa ohjelmaa käyttää suuri määrä ihmisiä. Hyvin optimoitu koodi voi parantaa myös käyttökokemusta ja auttaa ohjelmia toimimaan jouhevammin ilman ylimääräisiä viiveitä. Lisäksi tutkimus toimii hyvänä esimerkkinä siitä, miten suorituskykyä voidaan arvioida selkeästi ja luotettavasti erilaisia työkaluja, kuten BenchmarkDotNetiä ja Visual Studio Performance Profileria, käyttäen. Näin ollen tämä opinnäytetyö voi toimia jatkossa pohjana tai inspiraationa vastaaville tutkimuksille ja opinnäytetöille.

7.6 Oman osaamisen kehittyminen

Opinnäytetyön tekemisen aikana osaaminen kehittyi valtavasti, erityisesti suorituskyvyn mittaamiseen ja C#-kielen käytäntöihin liittyen. Vaikka oma lähtötaso ohjelmoinnissa ei ollut erityisen vahva, tutkimuksen edetessä käsitys

testauksen suunnittelusta ja toteuttamisesta vahvistui. Erityisesti suorituskykytestien suunnittelu, toteutus ja tulosten tulkinta lisäsivät varmuutta omaan tekemiseen. Testien laajuutta jouduttiin kuitenkin rajaamaan vastaamaan omaa osaamistasoa, mikä näkyy esimerkiksi testeissä käsiteltyjen rakenteiden valinnassa.

Kirjoittamisen osalta työskentely sujui luontevasti, ja erityisesti tiedonhakutaidot kehittyivät opinnäytetyön prosessin aikana. Laaja ja erittäin tekninen lähdeaineisto toi kuitenkin haasteita siihen, miten rajata vain oleellinen tieto tutkimuksen osalta, mutta lopulta teoriaosuudesta saatiin muodostettua ehjä ja hyvin perusteltu kokonaisuus. Työn tekeminen vaati pitkäjänteisyyttä, kärsivällisyyttä ja kykyä arvioida kriittisesti omia valintoja. Opinnäytetyön prosessin aikana opin soveltamaan teoriasta nostettuja havaintoja käytännön testaustilanteisiin. Yhteenvetona voidaan sanoa, että opinnäytetyö tarjosi hienon mahdollisuuden vahvistaa omaa teknistä osaamista sekä kehittää tutkimuksellista työskentelyotetta ja itsereflektointia.

Lähteet

Akinshin, A. 2019. Pro .NET Benchmarking: The Art of Performance Measurement. Apress. O'Reilly. 31.1.2025.

Alls, J. 2022. High-Performance Programming in C# and .NET. Packt Publishing. O'Reilly. 20.3.2025.

Anggoro, W. 2016. Functional C#. Packt Publishing. O'Reilly. 23.4.2025.

Bhargava, A. 2024. Grokking Algorithms, 2nd edition. Manning Publications. O'Reilly. 9.4.2025.

Bloch, J. 2017. Effective Java, 3rd edition. Addison-Wesley Professional. O'Reilly. 28.3.2025.

Canning, J., Broder, A. & Lafore, R. 2022. Data Structures & Algorithms in Python. Addison-Wesley Professional. O'Reilly. 6.5.2025.

Chen, Y. & Huang, L. 2025. MATLAB Roadmap to Applications. Springer, Singapore. https://link.springer.com/book/10.1007/978-981-97-8788-3 11.4.2025.

Codecademy. 2024. Loops. https://www.codecademy.com/resources/docs/c-sharp/loops 4.2.2025.

Downey, A.B. 2017. Think Data Structures. O'Reilly Media, Inc. O'Reilly. 7.5.2025.

Drayton, P., Albahari, B. & Neward, T. 2003. C# in a Nutshell, Second Edition. O'Reilly Media, Inc. O'Reilly. 28.1.2025.

Evans, B.J., Gough, J. & Newland, C. 2018. Optimizing Java. O'Reilly Media, Inc. O'Reilly. 10.4.2025.

Evtihevich, M. 2022. How has LINQ performance enhanced in .NET 7? PVS-Studio. https://pvs-studio.com/en/blog/posts/csharp/1011/ 13.5.2025.

Goldshtein, S., Zurbalev, D. & Flatow, I. 2012. Pro .NET Performance. Apress. O'Reilly. 5.2.2025.

Griffiths, I. 2024. Programming C# 12. O'Reilly Media, Inc. O'Reilly. 10.4.2025.

Heikkilä, T. 2014. Tilastollinen tutkimus. Helsinki: Edita.

Jamro, M. C# Data Structures and Algorithms. 2024. Packt Publishing. O'Reilly. 24.1.2025.

Khan, O.M.A. 2018. C# 7 and .NET Core 2.0 High performance. Packt Publishing. O'Reilly. 12.2.2025.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. & Yarom, Y. 2019. Spectre Attacks: Exploiting Speculative Execution. 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 1–19. https://ieeexplore.ieee.org/document/8835233/authors#authors 22.4.2025.

La Rocca, M. 2021. Advanced Algorithms and Data Structures. Manning Publications. O'Reilly. 8.4.2025.

La Rocca, M. 2024. Grokking Data Structures. Manning Publications. O'Reilly. 29.1.2025.

Lavieri, E. 2024. High Performance with Java. Packt Publishing. O'Reilly. 19.3.2025.

McGee, P. 2015. C#: A Beginner's Guide. McGraw-Hill. O'Reilly. 30.1.2025.

Merriam, S. & Tisdell, E. 2015. Qualitative Research: A Guide to Design and Implementation, 4th edition. Jossey-Bass. O'Reilly. 15.2.2025.
Michaelis, M. 2023. Essential C# 12.0. 8th edition. Addison-Wesley Professional. O'Reilly. 13.4.2025.
Microsoft Learn. 2025. Analyze performance by using CPU profiling in the Performance Profiler. https://learn.microsoft.com/en-us/visualstudio/profiling/cpu-usage?view=vs-2022 23.5.2025.
Nyingi, J. 2023. What is the best way to copy an array? DEV. https://dev.to/j0nimost/what-is-the-best-way-to-copy-an-array-1ch4 16.5.2025.
Oaks, S. 2014. Java Performance: The Definite Guide. O'Reilly Media, Inc. O'Reilly. 26.3.2025.
Oaks, S. 2020. Java Performance, 2nd edition. O'Reilly Media, Inc. O'Reilly. 16.4.2025.
Ohjelmointiputka. 2004. Rekursio-opas. https://www.ohjelmointiputka.net/oppaat/opas.php?tunnus=rekursio 30.1.2025.
Patterson, D.A. & Hennessy, J.L. 2020. Computer Organization and Design MIPS Edition, 6th edition. Morgan Kaufmann. O'Reilly. 17.4.2025.
Puusa, A., Juuti, P. & Aaltio, I. 2020. Laadullisen tutkimuksen näkökulmat ja menetelmät. Helsinki: Gaudeamus.
Roberts, E. 2005. Thinking Recursively with Java. Wiley. O'Reilly. 11.4.2025.
Rossett, A. 2009. First Things Fast: A Handbook for Performance Analysis, 2nd edition. Pfeiffer. O'Reilly. 12.4.2025.
Rubio-Sanchez, M. 2017. Introduction to Recursive Programming. CRC Press. O'Reilly. 25.1.2025.
Stellman, A. & Greene, J. 2024. Head First C#, 5th Edition. O'Reilly. 3.2.2025
SuomiGameHub. 2023. Koodin toiminnan ohjailu ehtolauseilla. YouTube-video. https://www.youtube.com/watch?v=oEAcQrf7zzE&t=1s. 29.1.2025.
Sweigart, A. 2022. The Recursive Book of Recursion. No Starch Press. O'Reilly. 25.3.2025.
Tan, M., Liu, X., Zhang, J., Tong, J. & Cheng, X. 2012. Compiler-Assisted Value Correlation for Indirect Branch Prediction. Chinese Journal of Electronics. IEEE. 414–418. https://ieeexplore.ieee.org/document/10197434 19.5.2025.
Tähtinen, J., Laakkonen, E. & Broberg, M. Tilastollisen aineiston käsittelyn ja tulkinnan perusteita. 2020. Turun yliopiston kasvatustieteiden laitos. https://urly.fi/2wou 25.3.2025.
Vickers, J. 2020. Is Using LINQ in C# Bad for Performance? Medium. https://medium.com/swlh/is-using-linq-in-c-bad-for-performance-318a1e71a732 17.3.2025.
Vijayalakshmi Pai, G.A. 2023. A Textbook of Data Structures and Algorithms, Volume 1. Wiley-ISTE. O'Reilly. 16.4.2025.
Vilkka, H. 2021. Tutki ja kehitä. Jyväskylä: PS-kustannus. 18.2.2025.
W3C. 2025. Use of Color. Web Accessibility Initiative WAI. https://www.w3.org/WAI/WCAG22/Understanding/use-of-color.html 20.5.2025.
Wengrow, J. 2024. A Common-Sense Guide to Data Structures and Algorithms in JavaScript, Volume 1. Pragmatic Bookself. O'Reilly. 12.4.2025.

Yadav, R.K. 2024. The Ultimate Laptop Buying Guide for Coders in 2025. DEV. https://dev.to/rajeshkumaryadavdotcom/the-ultimate-laptop-buying-guide-for-coders-in-2025-38la 12.5.2025.

Zakas, N.C. 2010. High Performance JavaScript. O'Reilly Media, Inc. O'Reilly. 27.3.2025.

Aineistonhallintasuunnitelman pohja

Aineistonhallintasuunnitelma

Suunnitelman tekijä(t): Tessa Helenius
Opinnäytetyön nimi: Tietorakenteiden ja ohjelmointikäytänteiden suorituskykyanalyysi C#-kielessä
Suunnitelma laadittu pvm: 7.2.2025

1. Aineiston yleiskuvaus

1.1 Aineiston kuvaus: Kerättävä tai olemassa oleva aineisto

Kerro lyhyesti tai tee taulukko tai luettelo keräämästäsi ja tuottamastasi tai olemassa olevasta aineistosta. Rasti oikea/oikeat vaihtoehto/vaihtoehdot
Kerään aineiston itse

- o haastatteluiden
- o kyselyiden
- o havainnoinnin avulla
- x muulla tavoin, millä? - Mittaamalla

Opiskelija tuottaa itse opinnäytetyössä analysoitavan ja suorituskykymittauksissa käytettävän lähdekoodiaineiston. Teoriaosuudessa aineistoa kerätään vapaasti käytössä olevista lähdemateriaaleista ja aiemmista tutkimuksista.

1.2 Aineiston laadun varmistaminen

Käsittelen aineistoa huolellisesti siten, että alkuperäinen tietosisältö säilyy eikä se muutu.

2. Eettiset periaatteet, lainsäädäntö ja henkilötietojen käsittely

2.1. Henkilötiedot ja tietosuojan huomioiminen
-
2.2 Päävastuu henkilötietojen käsittelystä eli rekisterinpitäjyys
-
2.3 Tietosuojan edellyttämät ilmoitukset
-
2.4 Tutkittavien informointi ja suostumus

Opinnäytetyöhön ei liity tiedonkeruuta ulkopuolisilta, kuten haastatteluja, kyselyitä tms.

2.5 Tutkimuslupa
Tarvitsen tutkimuslupaa: ei

2.6 Miten hallinnoit käyttämäsi, tuottamasi ja jakamasi aineiston oikeuksia?

Opiskelijan opinnäytetyötä varten tuottama lähdekoodi kulkee opinnäytetyön mukana, ja se on lisensoitu avoimella MIT-lisenssillä. Tämä mahdollistaa sen vapaan käytön, muokkaamisen ja jakelun, myös kaupallisessa tarkoituksessa, edellyttäen, että alkuperäinen lisenssiteksti ja tekijänoikeusilmoitus säilytetään.

2.7 Eettistä ennakkoarviointia edellyttävät tutkimusasetelmat opinnäytetöissä
Tarvitseeko opinnäytetyösi tutkimus eettisen ennakkoarvioinnin eettistä ennakkoarviointia?
-Ei

3. Aineiston dokumentointi

3.1. Aineiston dokumentointi
Tuotettu lähdekoodimateriaali ja saadut tulokset järjestetään kronologiseen järjestykseen, jokainen lähdekoodi ja jokainen testi omaan kansioonsa Karelian OneDrive pilvipalveluun.

4. Tallentaminen ja tietoturva opinnäytetyöprosessin aikana

Kerro tässä, minne aineisto tallennat ja miten sen varmuuskopioit opinnäytetyöprosessin aikana. Miten huolehdit, että ulkopuoliset eivät pääse aineistoon käsiksi? Suositus: Karelia-amk:n verkkoasema.

Kaikki aineisto tallennetaan Karelian tarjoamaan OneDriveen opiskelijan omaan kansioon

5. Aineisto opinnäytetyön valmistuttua: tuhoaminen, säilyttäminen tai mahdollinen jatkokäyttö ja avaaminen

Aineisto on oleellinen osa opinnäytetyön sisältöä, joten kaikki lähdekoodit ja muut materiaalit ovat vapaasti käytettävissä ja kulkevat opinnäytetyön mukana

6. Tehtävät ja vastuut

Kerro tässä lyhyesti, miten edellisissä vastauksissa kuvatut tehtävät ja vastuut on jaettu opinnäytetyössäsi: Opiskelija työstää opinnäytetyötä yksin, eli vastuu aineiston säilyttämisestä, tallentamisesta, hävittämisestä ym. On opiskelijalla itsellään.

Tietorakenteet		Paras tulos on merkitty ✓-symbolilla sekä vihreällä värillä			
		Heikoin tulos on merkitty ✗-symbolilla sekä punaisella värillä			
	Suoritusaika mikrosekunteinta µs	**Testit 1 & 2**			
Testi pvm	8.5.-25	9.5.-25	**Muistivaraus**		**Tiedosto**
Operaatio	**Search 1**	**2**			
Lista	✗ 70.74	69.85	0		***ListTests.cs***
Sanakirja	✓ 0.006684	0.007017	0		***DictionaryTests.cs***
Taulukko	70.72	69.54	0		***ArrayTests.cs***
	Add 1	**2**			
Lista	1283.8	1451.08	11.44 MB		***ListTests.cs***
Sanakirja	3928.78	✗ 3945.42	22.18 MB		***DictionaryTests.cs***
Taulukko	✓ 515.38	579.80	11.44 MB		***ArrayTests.cs***
	DeleteFromStart 1	**2**			
Lista	745.5	747.38	3.81 MB		***ListTests.cs***
Sanakirja	3941.96	✗ 3969.90	22.18 MB		***DictionaryTests.cs***
Taulukko	✓ 526.95	591.32	3.81 MB		***ArrayTests.cs***
	DeleteFromMiddle 1	**2**			
Lista	632.62	640.59	3.81 MB		***ListTests.cs***
Sanakirja	3931	✗ 3967.97	22.18 MB		***DictionaryTests.cs***
Taulukko	517.38	✓ 502.07	3.81 MB		***ArrayTests.cs***
	DeleteFromEnd 1	**2**			
Lista	530.49	597.90	3.81 MB		***ListTests.cs***
Sanakirja	3912.95	✗ 3967.55	22.18 MB		***DictionaryTests.cs***
Taulukko	✓ 523.63	574.51	3.81 MB		***ArrayTests.cs***

Silmukat ja LINQ		Paras tulos on merkitty ✓-symbolilla sekä vihreällä värillä							
		Heikoin tulos on merkitty ✗-symbolilla sekä punaisella värillä							
	Suoritusaika mikrosekunteinta µs	**Testit 1 & 2**							
Testi pvm	5.5.-25	6.5.-25	**Muistivaraus**	5.5.-25	6.5.-25	**Muistivaraus**	5.5.-25	6.5.-25	**Muistivaraus**
Metodi	**MaxWithFor 1**	**2**		**SumWithFor 1**	**2**		**CountWithFor 1**	**2**	
	1491.2	1493.0	0	1484	1476	0	1663	1666	0
	MaxWithForeach 1	**2**		**SumWithForeach 1**	**2**		**CountWithForeach 1**	**2**	
	1514.1	✗ 1520.3	0	1509	1504	0	1517	✓ 1514	0
	MaxWithWhile 1	**2**		**SumWithWhile 1**	**2**		**CountWithWhile 1**	**2**	
	1471.5	1474.8	0	1465	✓ 1458	0	1644	1730	0
	MaxWithDoWhile 1	**2**		**SumWithDoWhile 1**	**2**		**CountWithDoWhile 1**	**2**	
	1469.8	1475.5	0	1468	1465	0	1837	1840	0
	MaxWithLinq 1	**2**		**SumWithLinq 1**	**2**		**CountWithLinq**	**2**	
	347.3	✓ 346.1	0	✗ 3528	3516	40 B	3871	✗ 3873	40 B
Tiedosto	***MaxValueTests.cs***			***SumTests.cs***			***EvenNumberTests.cs***		
	MaxValue-testeissä etsittiin suurin luku alkioiden joukosta			*Sum-testeissä laskettiin alkiot yhteen*			*EvenNumber-testeissä etsittiin alkioista parilliset luvut*		

Ehdon järjestyksen vaikutus if-else & switch-case				Paras tulos on merkitty ✓ -symbolilla sekä vihreällä värillä			
				Heikoin tulos on merkitty ✗ -symbolilla sekä punaisella värillä			
	Suoritusaika mikrosekunteinta µs Testit 1 & 2						
Testi pvm	5.5.-25	6.5.-25	**Muistivaraus**		5.5.-25	6.5.-25	**Muistivaraus**
Metodi	**IfCommonFirst 1**	**2**			**IfAdultFirst 1**	**2**	
	2253	2248	2 B		2254	2253	2 B
	IfCommonMiddle 1	**2**			**IfAdultLast 1**	**2**	
	2244	✓ 2240	2 B		2247	✓ 2244	2 B
	IfCommonLast 1	**2**			**SwitchAdultFirst 1**	**2**	
	2247	2245	2 B		50566	49850	40 B
	SwitchCommonFirst 1	**2**			**SwitchAdultLast 1**	**2**	
	✗ 59555	55780	44 B		50824	✗ 52182	40 B
	SwitchCommonMiddle	**2**					
	59401	55648	44 B				
	SwitchCommonLast 1	**2**					
	56184	59266	44 B				
Tiedosto	***BirthMonthTests.cs***				***AgeGroupTests.cs***		

If-else vs. switch-case vs. rekursio			Paras tulos on merkitty ✓ -symbolilla sekä vihreällä värillä		
			Heikoin tulos on merkitty ✗-symbolilla sekä punaisella värillä		
	Suoritusaika mikrosekunteinta µs		**Testit 1 & 2**		
Testi pvm	5.5.-25	6.5.-25		9.5.-25	11.5.-25
Metodi	**IfElse 1**	**2**		**IterativeSearch 1**	**2**
	225.0	226.3		0.005805	0.005219
	SwitchCase 1	**2**		**RecursiveSearch 1**	**2**
	✓ 224.3	224.6		✗ 0.007021	✓ 0.004177
	Recursive 1	**2**			
	✗ 5425.8	5253.2			
Tiedosto	***GetGradeTests.cs***			***SearchRecursiveIterative.cs***	

Tästä linkistä pääset GitHub-arkistoon, josta löydät kaikki tässä tutkimuksessa käytetyt testikoodit:
https://github.com/TessaHelenius/Performance-Tests.git