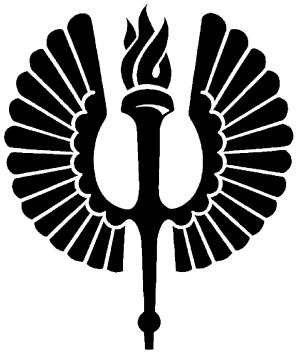


Välimuistitietoisuuden vaikutus k-aristen puiden suorituskykyyn

Riku Rajaniemi

Tietojenkäsittelytiede
Pro gradu -tutkielma
Marraskuu 2017

TULEVAISUUDEN TEKNOLOGIOIDEN LAITOS
TURUN YLIOPISTO

TURUN YLIOPISTO
TULEVAISUUDEN TEKNOLOGIOIDEN LAITOS

RIKU RAJANIEMI: Välimuistitietoisuuden vaikutus k-aristen puiden suorituskykyyn

Pro gradu -tutkielma, 62 sivua
Tietojenkäsittelytiede
Marraskuu 2017

Tietokoneiden keskusmuisti ei ole nopeutunut samaa tahtia prosessorien kanssa. Tämän vuoksi tehokkailla prosessorisiruilla on nykyään nopeaa välimuistia. Välimuistin tehokkaalla hyödyntämisellä voi olla suurempi vaikutus ohjelman suorituskykyyn kuin suoritettavien käskyjen määrällä.

Tässä tutkielmassa vertaillaan kolmen k-arisen puurakenteen suorituskykyä välimuistin tehokkaan hyödyntämisen näkökulmasta. Puista kaksi on tyypillisiä osoittimista koostuvia puurakenteita, ja yksi on uusi syvyystaulukkoon pohjautuva puurakenne. Osoittimista koostuvista puista testataan naiivin version lisäksi muistivarantoa käyttävä versio, joka vähentää vaadittavien muistinvarausoperaatioiden määrää ja parantaa tietorakenteen sisäistä välimuistipaikallisuutta.

Puurakenteiden arviointi suoritetaan käyttäen useita erilaisia muokkaus- ja lukuoperaatioita, ja ennen jokaista testiä välimuisti tyhjennetään vertailukelpoisen lähtötilanteen takaamiseksi.

Testit osoittavat, että osoitinpuiden tapauksessa muistivarannon käyttäminen parantaa puiden suorituskykyä muokkausoperaatioissa 10–80 %, ja että syvyystaulukkopuu suoriutuu lukuoperaatioista osoitinpuita 20–90 % lyhyemmässä ajassa.

Avainsanat: välimuisti, välimuistitietoinen, muistivaranto, k-arinen, puu, osoitin, suorituskyky, syvyystaulukkopuu

UNIVERSITY OF TURKU
DEPARTMENT OF FUTURE TECHNOLOGIES

RIKU RAJANIEMI: The Effect of Cache-Sensitivity on the Performance of k-ary Trees

Master's Thesis, 62 pages
Computer Science
November 2017

The speed and bandwidth of central memory are lagging behind the increasing power of processors. To alleviate this situation recent high performance processors have a fast cache included directly on the processor chip. Using the cache efficiently may have a greater effect on the performance of a program as a whole than the number of instructions executed.

In this thesis, the performance of three different k-ary tree structures is compared keeping the cache-sensitivity in mind. Two of the trees are typical pointer-based tree structures and the third is a novel depth array tree, which instead of node structures holds an array of vaelues and a same length array of depth numbers to represent the shape of the tree. In addition to the naive versions of the pointer-based trees, pooled versions are also tested, which have improved memory allocation performance and memory locality.

Several modification and read operations are tested on these trees and before every test the processor cache is overwritten to ensure a clean starting state for every test.

The tests show that for modification operations the pooled versions of pointer-based trees perform 10–80 % better when compared to the non-pooled versions. Further for read operations the depth array tree performs 20–90 % better in comparison to the pointer trees.

Keywords: cache, cache-sensitive, memory pool, k-ary, tree, pointer, performance, depth array tree

Sisältö

1 Johdanto

Nykyaikaisten prosessorien nopeuden kasvun myötä muistin hitaus on muodostunut tietokoneiden suorituskyvyn merkittäväksi pullonkaulaksi. Yksi haku tietokoneen keskusmuistista kestää tyypillisesti satoja nanosekunteja [21], mikä on merkittävästi enemmän kuin vaikkapa kalliina pidetyn neliöjuurioperaation kesto.

Keskusmuistiväylän hitauden vuoksi prosessoreihin on lisätty nopeita välimuistitasoja, jotta vasta käytettyä dataa ja käskyjä voitaisiin käyttää nopeasti uudelleen. Jos algoritmien ja erityisesti tietorakenteiden suunnittelussa ei kuitenkaan oteta huomioon muistiväylän asettamia rajoituksia, tarpeettomat keskusmuistiin kohdistuvat muistihaut voivat merkittävästi heikentää ohjelman suorituskykyä. [20]

Esimerkki erityisen huonosti välimuistia hyödyntävästä tietorakenteesta on linkitetty lista. Naiivissa toteutuksessa jokainen hyppy seuraavaan listan solmuun aiheuttaa uuden haun keskusmuistista, eikä prosessorin ole mahdollista ennakoivasti noutaa seuraavia solmuja. Samaan tapaan puurakenteet, joissa solmut koostuvat muistiosoittimista muihin solmuihin, voivat aiheuttaa suuren määrän muistihakuja tavallisen käytön aikana.

Välimuistin tehokkaan hyödyntämisen merkitystä suorituskyvylle on informaatioteknologiassa tutkittu jo hyvän aikaa. Esimerkiksi Vesa Hirvisalo (2004) esittelee väitöskirjassaan monipuolisen ja yleistettävissä olevan tavan tutkia ohjelman välimuistisuorituskykyä staattisen ja dynaamisen analyysin yhdistelmän avulla. Tässä metodissa mitattavan ohjelmakoodin sekaan lisätään staattisen analyysin avulla valittuihin sijainteihin koodia, joka puolestaan suorittaa dynaamista analyysiä. Molempien analyysien tulokset yhdistetään lopuksi muotoon, josta voidaan nähdä, missä kohtaa ohjelman suoritusta tapahtuu eniten *muistihuteja* (engl. cache miss). Kyseinen metodi on jätetty tämän tutkielman rajauksen ulkopuolelle, mutta se voisi olla hyvä jatkotutkimuksen aihe.

Aiempi puurakenteiden *välimuistitietoisuuden* (engl. cahce-sensitive) tutkimus, on keskittynyt pääasiassa binäärihakupuiden käyttötehokkuuteen. Esimerkiksi Riku Saikkonen (2009) käsittelee välimuistitietoista solmujen muistisijoittelua binäärihakupuille. Suoritettujen testien tulokset osoittavat, että puiden läpikäyminen nopeutui 30–50 % käytettäessä välimuistitietoista solmujen sijoittelua.

Tässä tutkielmassa vertaillaan kolmen k-arisen puurakenteen suorituskykyä erityisesti välimuistitietoisuuden näkökulmasta. Vertailtavista puurakenteista kaksi on tyypillisiä muistiosoittimista koostuvia puita ja yksi on uusi syvyystaulukkoa käyttävä lineaariseen muistirakenteeseen pohjautuva puu. Kaikki kolme puurakennetta ovat muodoltaan k-arisia puita eli jokaisella solmulla voi olla rajoittamaton määrä lapsisolmuja. Osoitinpuista käytetään testeissä sekä naiivia versiota että muistivarantoa (engl. memory pool) käyttävää versiota, jossa solmuja varten varataan kerralla isompi muistialue.

k-arisia puita käytetään laajalti ohjelmistotuotannossa datan säilytysmuotona. Tietokonepeleissä maailmat koostuvat usein entiteettien hierarkioista – Unity, Unreal Engine ja Ogre3D ovat suosittuja pelimoottoreita, jotka käyttävät k-arisia puita maailmojensa kuvaamiseen. Käyttöliittymien rakennetta kuvataan monesti widgettien hierarkiana (Qt, Tk, wxWidgets, Windows Forms). Edellä mainituissa sovellutuksissa puun tietorakenne on tyypillisesti tässä tutkielmassa esiintyvän lapsitaulukkopuun mukainen sillä erotuksella, että edellä mainituissa käyttökohteissa puun solmut ovat sangen kookkaita ja eivät näinollen pysty hyödyntämään puun läpikäymisessä välimuistia tehokkaasti. Näissä käyttökohteissa puita myös luetaan huomattavasti enemmän kuin muokataan. Tämän tutkielman motivaationa on tutkia voitaisiinko tällöin käyttää välimuistitietoista puurakennetta suorituskyvyn parantamiseksi.

Testien suorituksessa pyritään tyhjentämään prosessorin välimuisti ennen kunkin testin suoritusta, jotta siinä ei olisi valmiiksi puun solmuja. Tällä taataan sama lähtötilanne jokaiselle testille. Testituloksista havaittiin, että muistivarantoa käyttävät versiot osoitinpuista suoriutuvat solmujen lisäyksestä ja poistosta 10–80 % nopeammin kuin muistivarantoa käyttämättömät versiot. Sy-

vyystaulukkopuu puolestaan suoriutuu 20–90 % nopeammin puun lukemisesta kuin osoittimista koostuvat puut.

Luvussa 2 käydään läpi aiempaa tutkimusta ja taustaa puurakenteista, välimuistin tehokkaasta hyödyntämisestä ja suorituskyvyn mittauksesta. Luvussa 3 esitellään vertailtavat puurakenteet ja luvussa 4 vertailtavat operaatiot sekä testausympäristö ja -käytännöt. Testien tulokset kuvaajineen ovat luvussa 5, ja luvun lopussa on myös tulosten analysointi. Lopuksi vielä tehdään yhteenveto luvussa 6.

1.1 Käsitteitä

Näin aluksi on syytä esitellä tutkielmassa käytettyjä keskeisiä käsitteitä. Termeille esitetään myös englanninkielinen vastine, koska monilla termeistä ei ole vielä vakiintunutta suomenkielistä vastinetta.

prosessorin välimuisti
(engl. CPU cache)

Prosessorin sisäinen nopea välimuisti, jota on useimmiten kolmea tasoa: L1, L2, L3. L1-välimuisti on pienin ja nopein – kooltaan kymmeniä kilotavuja ja latenssiltaan alle nanosekunti. L3-välimuisti on suurin ja hitain – kooltaan jotain megatavuja ja latenssiltaan noin kymmeniä nanosekunteja [11] [7]. Myös L3-välimuisti on silti merkittävästi nopeampi kuin keskusmuisti, jonka latenssi on satoja nanosekunteja [21].

muistihaku

Prosessorin tarvitseman datan ja käskyjen noutaminen muistista tai välimuistista rekistereihin. Muistihaun kestoa ei voida etukäteen tietää varmasti, mistä johtuen prosessori voi joutua viivyttämään hausta riippuvaisia komentoja [12].

välimuistihuti
(engl. cache miss)

Jos muistihaun kohde ei löydy joltakin välimuistin tasolta, sen hakemista yritetään seuraavalta. Tällöin on tapahtunut välimuistihuti. Voidaan vielä tarkentaa, että jos dataa ei löytynyt L1-välimuistitasolta, on tapahtunut L1-välimuistihuti. Jos puolestaan data löytyy joltain välimuistitasolta, on tapahtunut *välimuistiosuma* (engl. cache hit).

välimuistilinja

(engl. cache line)

Muistihaku prosessorin välimuistista ja keskusmuistista tapahtuu aina vakiokokoisina paloina, joita kutsutaan välimuistilinjoiksi. Tyypillisiä välimuistilinjan kokoja ovat 32, 64 ja 128 tavua, joista 64 tavua on nykyaikaisilla prosessoreilla yleisin. Jos tietue on suurempi kuin välimuistilinja, on sen noutamiseksi keskusmuistista suoritettava useampi muistihaku. Vastaavasti jos tietue on pienenpi kuin välimuistilinja, sen mukana tulee loppu linjan verran viereistä dataa.

Välimuistilinjoja tehokkaasti käyttävää koodia ja tietueita kutsutaan *välimuistitietoiseksi*.

muistioptimointi

Ohjelman suoritusnopeutta voidaan parantaa pitämällä keskenään relevantti data muistissa lähekkäin, jotta välimuistilinjassa olisi aina mahdollisimman suuri hyötykuorma. Tätä voidaan edesauttaa mm. järjestelemällä tietueen muuttujat niin, että useimmin käytetty informaatio on tietueen alussa, ja pienet arvot on pakattu vierekkäin.

muistivaranto

(engl. memory pool)

Muistivarannolla tarkoitetaan yksinkertaisemmillaan esivarattua muistia. Sen pääasiallinen tarkoitus on vähentää muistinvaraus- ja -alustuskutsujen määrää, mutta sillä on myös suuri positiivinen sivuvaikutus pitää keskenään relevanttia dataa lähekkäin muistiavaruudessa.

2 Taustaa

Tässä luvussa annetaan taustaa hakupuiden erilaisille muistirakenteille, niiden käyttötarkoituksille ja aiemmalle tutkimukselle. Myös välimuistitietoisuuden käsite esitellään tarkemmin alan tutkimuksen kera.

2.1 Puurakenteet

Tietojenkäsittelyssä puulla tarkoitetaan useasta tietueesta – solmuista – koostuvaa tietorakennetta. Solmut on liitetty toisiinsa siten, että yksi solmu voi viitata useampaan muuhun lapsisolmuun. Puussa ei ole silmukoita, eli solmu ei voi viitata solmuun, joka suoraan tai välillisesti viittaa takaisin siihen. Solmuilla on myös hyötykuormaa, jonka sisältö riippuu käyttötarkoituksesta. [8] Tässä tutkielmassa tarkastellaan niin kutsuttuja juurrettuja puita, joissa solmujen välillä on hierarkia; kullakin solmulla (isäsolmu) on 0, 1 tai useampia lapsisolmuja. Tällaista puuta kutsutaan *juurretuksi*. Yleinen käytäntö on piirtää puun juurisolmu, solmu jolla ei ole isäsolmua, ylimmäksi ja sen lapset tämän alapuolelle.

Puun muotoa kuvaamaan käytetään usein termiä *haarautumisaste*. Puuta jonka haarautumisaste on kaksi, eli jonka jokaisella solmulla voi olla korkeintaan kaksi lapsisolmua kutsutaan *binääripuiksi*. Binääripuilla on lukuisia erilaisia käyttötarkoituksia, joista yksi on tiedon hakemisen nopeuttaminen. *Binäärihakupuun* solmut on järjestetty siten, että jokaisen solmun vasemmalle puolelle jää solmun sisältämää avainarvoa pienemmän avaimen omaavat lapsisolmut ja oikealle puolelle suuremman. Esimerkki tällaisesta binäärihakupuusta on *kd*-puu, jota käytetään k-ulotteisen avaruuden jakamiseen kerrallaan yhden akselin mukaan. Solmuun tallennettu arvo on tällöin jonkin ulottuvuuden arvo tässä koordinaatissa. [4] Muita vakioasteisia puun muotoja ovat nelikko- ja kahdeksikkopuu [5], joiden haarautumisasteet ovat neljä ja kahdeksan ja joita käytetään tyypillisimmin kaksiulotteisen ja kolmiulotteisen datan jäsen-

tämiseen.

Vakioasteisen puurakenteen hyviin puoliin kuuluu, että puun solmuille voidaan aina etukäteen tietää sen lapsiosoittimien vaatima tila. Tällaiset puut voidaan myös tallettaa lineaariseen taulukkoon ilman, että solmujen lapsiosoittimia tarvitsee eksplisiittisesti tallentaa. Esimerkiksi keko-rakenteessa taulukkoon tallennetun binääripuun lapset ovat taulukossa paikoilla $2n + 1$ ja $2n + 2$, missä n on solmun oma sijainti [9]. Tällä tavalla pakattuna solmut ovat lähellä toisiaan muistissa, millä on positiivinen vaikutus suorituskykyyn, mihin perehdytään tarkemmin aliluvussa 2.2.

K-ariseksi puuksi kutsutaan puuta, jonka solmulla voi olla korkeintaan k lapsisolmua. Binääripuu on esimerkki k-arisen puun erikoistapauksesta, jossa $k = 2$. k-arisella puulla voidaan kuitenkin myös tarkoittaa puuta, jossa k:n arvoa ei ole ennalta määritelty, jolloin solmuilla voi olla vapaa määrä lapsia, ja k on suurin puussa esiintyvä haarautumisaste.[10]

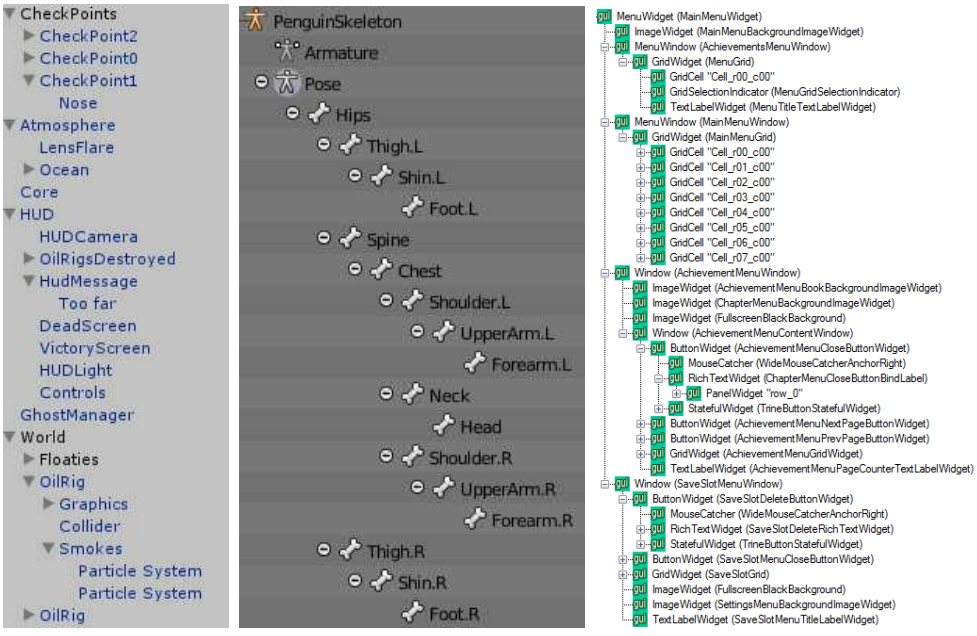
Luvussa 3 esiteltävän tavan lisäksi, k-arinen puu voidaan tallentaa lineaariseen taulukkoon siten, että solmut ovat leveyshakujärjestyksessä, jolloin solmun lasten järjestyslyvut ovat välillä $[kn+1, kn+k]$, jossa n on solmun oma osoite. Jos k-arinen puu on täydellinen, eli sen kaikilla solmuilla paitsi lehtisolmuilla on täydet k lasta [14], on tämä rakenne muistin käytöltään optimaalinen eli se ei hukkaa tilaa.

Useissa datan säilytykseen liittyvissä käyttötarkoituksissa tarvitaan kuitenkin puurakennetta, jolla voi olla rajoittamaton märää lapsisolmuja. Rajoittamattoman lapsimäärän myötä solmujen koko voi myös vaihdella tarpeen mukaan, jolloin vähemmän haarautuvat solmut vievät vähemmän tilaa muistissa kuin, jos puun jokainen solmu olisi puun suurimman solmun kokoinen. Vapaan haarautumisasteensa vuoksi tällaisia puita ei kuitenkaan voida tallentaa lineaariseen taulukkoon ilman lisäinformaatiota kuten vakioasteiset puut. Tämä lisäinformaatio voi olla esimerkiksi jokaiselle solmulle tallennettu isäsolmuosoitin tai, kuten luvussa 3.3 esiteltävässä syvyystaulukkopuussa, kunkin solmun syvyys puun juuresta lukien.

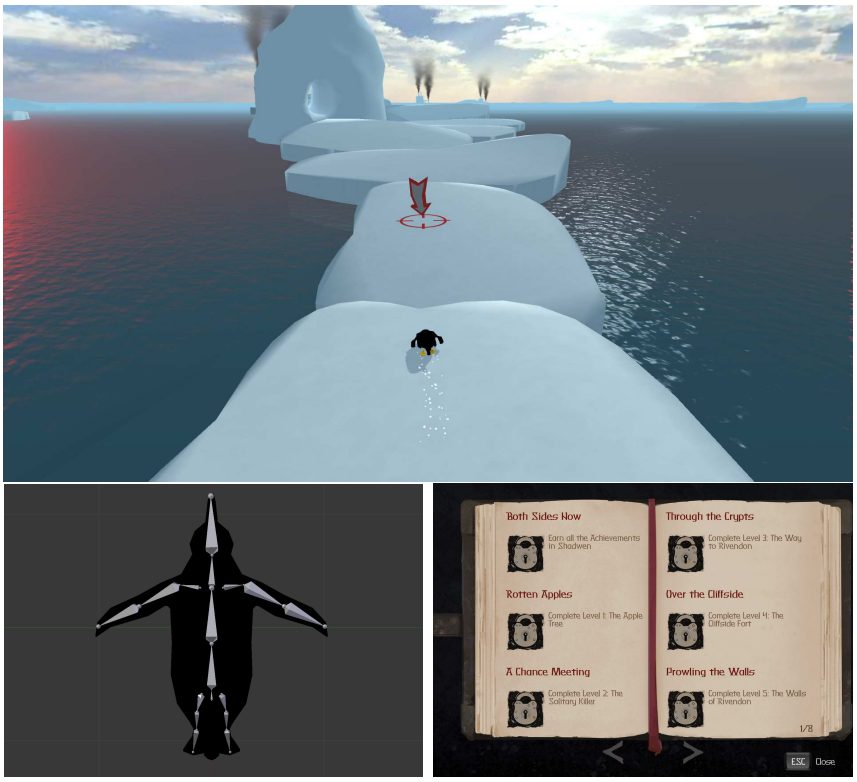
Käytännön esimerkkejä vapaahaarautuvista puista ovat widgettipohjaiset käyttöliittymäympäristöt kuten Qt [1], käyttöjärjestelmien kansiorakenteet, pelimoottoreiden kenttägraafien oliohierarkiat ja tietokoneanimoitujen hahmojen luuhierarkia [6]. Näistä on esimerkkejä kuvassa 1 ja puiden aikaansaamat lopputulokset on esitelty kuvassa 2. Kaikissa edellämainituissa käyttötarkoituksissa sovellutuksen suorituskyky on ensiarvoisen tärkeää, minkä vuoksi myös vapaahaarautuvien puiden suorituskyvyn parantamisen tutkiminen on tärkeää.

Vapaahaarautuvista puista on saatavilla huomattavasti vähemmän tieteellistä tutkimusta kuin vakiohaaratuvista puista. Esimerkiksi tieteelliseen kirjallisuuteen keskittyvä hakukone Google Scholar palauttaa "binary tree" hakusanalla 165 000 osumaa, kun taas "k-ary" palauttaa 25 000 osumaa ja "k-ary tree" 2 700 osumaa.

Riku Saikkonen (2009) keskittyy väitöskirjassaan *Bulk Updates and Cache*



Kuva 1: Ruutukaappauksia puumallisten tietorakenteiden näkymistä tuotantokäytössä olevissa ohjelmistoissa. Vasemmalta lukien: Unity-pelimoottorin kenttägraafi, Blender-3D-mallinnusohjelman hahmomallin animaatioluurakenne, Frozenbyte Oy:n käyttöliittymäympäristön widgettipuu



Kuva 2: Ruutukaappauksissa alkaen ylhäältä Unity pelimoottorilla tehty peli, vasemmalla alhaalla hahmon animaatioluurakenne Blenderissä ja oikealla alhaalla Frozenbyte Oy:n käyttöliittymäympäristöä käyttäen toteutettu valikko

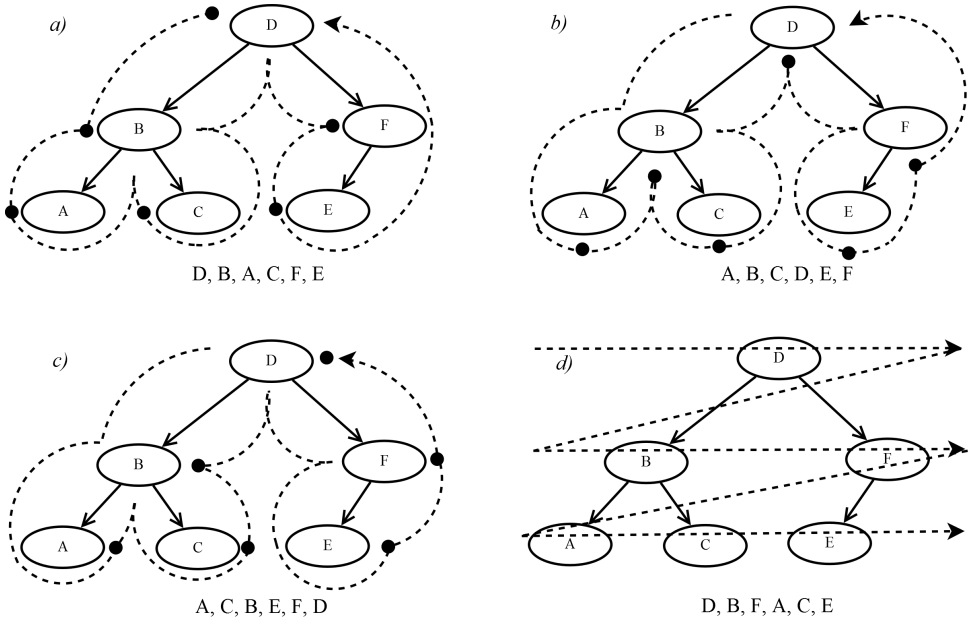
Sensitivity in Search Trees keskittyy punamusta- ja AVL-puiden suorituskyvyn parantamiseen välimuistitietoisuuden ja eräpäivitysten avulla [17]. Eräpäivityksellä tarkoitetaan usean solmun lisäämistä, poistamista tai muuttamista yhdellä operaatiolla. Eräpäivitys on tehokkaampi tapa muokata montaa avainta kerrallaan verrattuna yksi kerrallaan muokkamiseen. Esimerkiksi erälisäysalgoritmit toimivat yleisesti keräämällä monta solmua, joista luodaan valmiiksi tasapainotettu alipuu. Tämä alipuu lisätään olemassa olevaan puuhun, jolle suoritetaan sitten tasapainotusoperaatioita kunnes puu on jälleen tasapainotettu [17]. Välimuistitietoisuuteen paneudutaan tarkemmin luvussa 2.3.

2.2 Puissa kulkeminen

Puussa kulkemisella tarkoitetaan prosessia, jolla voidaan käydä puun jokaisessa solmussa kerran ja vain kerran. Nämä operaatiot useimmiten rekursiivisia, koska itse puurakenteet ovat tyypillisesti rekursiivisesti määriteltyjä. Kaksi tyypillisintä puiden läpikäymistapaa ovat syvyyshaku ja leveyshaku, joita havainnollistetaan kuvassa 3.

Syvyyshaussa vieraillaan ensiksi vuorossa olevan solmun ensimmäisessä lapsisolmussa ja kuljetaan tämän lapsisolmun muodostama alipuu syvyyshakujärjestyksessä, jonka jälkeen kuljetaan loppujen lapsisolmujen muodostamat alipuut rekursiivisesti samalla tavalla. Järjestys, jossa kukin läpikäyty solmu luetaan vaihtelee toteutuksen mukaan. Tässä tutkielmassa käytetyssä syvyyshaussa esimerkiksi isäsolmu luetaan ennen lapsisolmuihin siirtymistä, kuvan 3 graafin (a) eli esijärjestyksen mukaisesti. Kuvan 3 graafin (b) mukainen sisäjärjestys on tyypillisesti käytössä binäärihakupuissa, joissa solmun lapsista toisen arvo on suurempi ja toisen pienenpi kuin solmun oma arvo. Graafin (c) mukainen jälkijärjestys käy solmujen alipuut ennen itse solmua, mikä käytännössä tarkoittaa, että puussa edetään lehdestä juurta kohti.

Kuvan 3 graafin (d) esittämä läpikäymisjärjestys on puolestaan leveyshaku, jossa puu käydään läpi kerroksittain. Tämä vaatii tyypillisesti jonorakenteen,



Kuva 3: Visualisoinnit eri hakutyypeistä. Syvyyshaku (a) esijärjestyksessä, (b) sisäjärjestyksessä ja (c) jälkijärjestyksessä, sekä (d) leveyshaku. Syvyyshakuja vastaavissa kuvaajissa mustat täplät kuvastavat solmujen läpikäyntijärjestystä. Kuvaajien alapuolella on solmujen avaimet haun läpikäymässä järjestyksessä.

johon vastaan tulleiden solmujen lapsisolmut kerätään, ja joissa vieraillaan FIFO-järjestyksessä. Taulukkoon pohjautuvat vakiohaarautuvat puut kuten binääripuut voidaan järjestää lineaarisesti muistiin siten, että solmut ovat muistissa leveyshaun mukaisessa järjestyksessä.

2.3 Välimuisti

Nykyaikaiset tietokoneet käyttävät von Neumann -arkkitehtuuria. Von Neumann -arkkitehtuurin mukaiset tietokoneet koostuvat syöte- ja tulostuslaitteiden lisäksi prosessorista ja muistiväylästä [13]. Prosessori ja muistiväylä yhdessä määrittävät tietokoneen nopeuden.

Siinä missä prosessorien nopeus on kasvanut viime vuosiin asti eksponentiaalisesti, keskusmuistin nopeus on kasvanut vain vähän. Tästä syystä prosessorin nopeus on enää harvoin suorituskykypullonkaula tyypillisille ohjelmil-

le. Verrattuna prosessorin suorittamien käskyjen kestoon, datan noutaminen keskusmuistista kestää huomattavasti pidempään kuin pisimmätkin käskyt [17, 20, 11, 21].

Keskusmuistin suhteellisen hitauden vuoksi nykyaikaisilla prosessoreilla on välimuistia suoraan sirulla, joka on huomattavasti nopeampaa kuin keskusmuisti. Välimuistiakin on useita (tyypillisesti kolme) kerrosta. Kerroksista pienin ja nopein, L1, on kooltaan tyypillisesti joitain kymmeniä kilotavuja. Suurin L3 on hitain ja tyypillisesti kooltaan jotain megatavuja. Jokaisella prosessoriytimellä on tyypillisesti oma L1 välimuisti, kun taas L3 välimuisti on kaikille ytimelle yhteinen [11, 21]. L2 tason välimuistin tehtävä vaihtelee hieman eri prosessorien välillä, mutta tyypillisesti sen koko ja nopeus sijoittuu L1 ja L3 tasojen välimaastoon, ja sitä on joko yksi jokaisella ytimellä (kuten Intel Skylake -prosessorit) tai osa ytimistä saattaa jakaa saman L2 välimuistin (kuten AMD Ryzen -prosessorit).

Välimuististitietoisuudesta (engl. cache-consciousness) on kaksi eri mallia: *välimuistiherkkä* (engl. cache-sensitive) ja *välimuistiailahteleva* (engl. cache-oblivious). Välimuistiherkkä algoritmi tiedostaa ympäristönsä välimuistin todellisen luonteen – välimuistin ja muistiväylän koon – ja pyrkii optimoimaan juuri tätä silmällä pitäen. Välimuistiailahteleva pyrkii optimoimaan välimuistin käyttöä yleisesti pienentämällä resurssien käyttöä mutta ei aseta oletuksia ympäristönsä tarkoille piirteille. [17]

Riku Saikkonen ja Eljas Soisalon-Soininen käsittelevät artikkelissaan *Cache-sensitive Memory Layout for Binary Trees* [15] ja artikkelissaan *Cache-Sensitive Memory Layout for Dynamic Binary Trees* [18] binääripuiden muistisijoittelun välimuistitietoistamista. Artikkeleissa välimuistitietoisuuden lisäämisellä tavallisiin binääripuihin, parannetaan niiden suorituskykyä ilman, että solmujen tietueita tarvitsee muuttaa tai puille suoritettavien algoritmien aikavaatimukset kasvavat. Puiden solmut asetellaan jokaisen muokkauksen jälkeen siten, että solmun isäsolmu tai vähintään yksi lapsisolmu päätyy samaan välimuislinjaan. Tällä saavutetaan jälkimmäisessä artikkelissa 26–36 %:n parempi suorituskyvyn solmuille, joiden koko on 16 tavua välimuistilinjan ollessa

kooltaan 64 tavua [16, 18].

Välimuistin käytön analysointi ja suorituskyvyn mittaaminen on vaikeaa, koska ympäristön vaikutus välimuistin sisältöön on lähes väistämätön. Koska ohjelmistot toimivat käyttöjärjestelmän, joka myös tarvitsee tietokoneen resursseja omiin tarpeisiinsa, alaisuudessa, ei välimuistin tila ole lähes koskaan deterministinen ja muuttumaton. Tästä syystä myös ohjelmien suorituskyvyn mittaaminen on haastavaa, koska tietokoneen tila voi helposti poiketa testien suoritusten välillä. [20]

Vesa Hirvisalo (2004) esitteli tavan ohjelmien välimuistin käytön simuloimiseen käyttäen staattisen ohjelmistoanalyysin ja dynaamisen muistianalyysin yhdistelmää. *Staattinen ohjelmistoanalyysi* tarkoittaa ohjelmakoodin analysointia ilman, että ohjelmaa suoritetaan. Sen suuri etu on sen kyky samanaikaisesti antaa tuloksia kokonaisille syötearvojoukoille. Mutta koska ohjelman suoritus on useasti erilainen eri syötteillä, staattinen analyysi on luonteeltaan yleensä approksimoivaa. Tarkkojen tulosten sijaan se antaa tyypillisesti ylä- ja alaraja-arvoja. [20]

Dynaaminen muistianalyysi keskittyy simuloimaan ohjelman suorituksen aikaista muistin tilaa. Modernin muistiarkkitehtuurin monimutkaisuuden vuoksi muistiviittaukset ovat monimutkaisia. Tästä syystä testattavan ohjelman muistioperaatiot yleensä simuloidaan. Yleisin muistisimulaation malli on *jäljitepohjainen simulaatio* (engl. trace-driven simulation), jossa analysoitavasta ohjelmasta kerätään sen suorituksen aikana informaatiota yksittäisistä tapahtumista kuten muistihauista. Tämän analyysin tulosta kutsutaan *jäljitteeksi*. Jäljitepohjainen simulaatio suoritetaan kahdessa vaiheessa. Ensimmäisessä vaiheessa jäljite kerätään. Koska yleinen tietokonelaitteisto ei tue jäljitteiden keräämistä automaattisesti ohjelman normaalin suorituksen aikana, jäljitteitä lähettäviä käskyjä on lisättävä suoritettavan ohjelmakoodin sekaan. Toisessa vaiheessa suoritetaan simulaatio käyttäen ensimmäisessa vaiheessa saatua jäljitettä syötteenä. Tässä simulaatiossa mallinnetaan esimerkiksi hypoteettisen muistijärjestelmän tilaa. Näiden vaiheiden välissä jäljite tyypillisesti pelkistään poistamalla siitä turhaa ja redundanttia dataa sen käytön

tehostamiseksi. [20]

Jäljitepohjainen muistisimulaatio on erityisen hyödyllistä laitteistotason tutkimuksissa, koska samaa jäljitettä voidaan käyttää useita kertoja, ja simulaation tulosta voidaan käyttää vertailuarvona. Jäljitteiden koko voi kuitenkin olla valtava. Sen sijaan ohjelmistoanalyysissä on käytännöllisempää simuloida muistia ohjelman suorituksen aikana, tällöin jäljitettä ei tallenneta vaan simulaatio käyttää sen sitä mukaa, kun se luodaan. [20]

Vesa Hirvisalon [20] esittämä yhdistelmäanalysointi yhdistää välimuistin dynaamisen ja staattisen analyysin siten, että staattinen analyysi nopeuttaa dynaamista analyysiä ja dynaaminen analyysi tarkentaa staattisen analyysin tuloksia. Tässä yhdistelmäanalyysissä on kolme vaihetta: käännösvaihe, suoritusvaihe ja yhdistämisvaihe. Käännösvaiheessa ohjelmaa analysoidaan staattisesti ja muodostetaan koodi, johon on lisätty dynaamisen analyysin edellyttämät käskyt. Suoritusvaiheessa muokattu ohjelma suoritetaan monilla eri syötteillä. Yhdistelmävaiheessa käännösvaiheen staattisen analyysin ja suoritusvaiheen dynaamisen analyysin tulokset yhdistetään ohjelmaksi, josta on poistettu tarpeettomaksi todettuja ja mahdollisesti raskaitakin osia kuten silmukoita. [20]

Eräs toinen välimuistinkäytön simulointiin tarkoitettu työkalu on Insomniac Games –peliyrityksen kehittämä CacheSim [3]. Tällä ohjelmalla pyritään simuloimaan välimuistin tilaa AMD Jaguar-perheen prosessoreissa, jotka ovat käytössä mm. sekä Sony PlayStation 4 ja Microsoft Xbox One pelikonsoleissa [2]. CacheSim on tarkoitus suorittaa lyhyissä purskeissa – yhden ruudunpäivityksen verran – ja datan kerääminen kestää 2–3 minuuttia. Sen avulla pyritään paikantamaan ohjelman suorituksen aikana usein tapahtuvia välimuistihuteja. [3]

Toisin kuin jäljitepohjaiset muistisimulaatiot CacheSim ei edellytä ohjelmakoodin muokkausta, vaan se hyödyntää modernien prosessorien ominaisuutta astua suoritettavan ohjelman läpi yksi käsky kerrallaan – *virheenjäljittäjät* (engl. debugger) käyttävät tätä ominaisuutta. CacheSim analysoi jokaisen

käskyn ja ylläpitää omaa välimuistisimulaatiotaan, josta suorituksen lopuksi muodostetaan raportti eniten muistihuteja aiheuttaneista koodiriveistä. Koska CacheSimin simuloiman prosessorin muistiarkkitehtuuri poikkeaa melko suuresti tyypillisistä tietokoneprosessoreista, sen tulosten hyödyllisyys nykyisellään on rajallinen. Jaguarissa on jokaisella ytimellä 64 Kt L1-välimuistia ja 2 Mt jaettua L2-välimuistia, ja välimuistitasot ovat inklusiivisiä – L1-välimuistien sisältö on kokonaisuudessaan myös L2-välimuistissa. [3]

2.4 Suoriutuskyvyn vertaileminen

Ohjelmien toiminnassa kiinnitetään usein paljon huomiota ohjelmien ominaisuuksiin ja oikeellisuuteen, mutta arkikäytössä näitä merkittävämpi piirre voi olla suorituskyky. Silti ohjelmien ja algoritmejen kehityksen aikana kiinnitetään vain vähän huomiota todelliseen suorituskykyyn. [22]

Suorituskykyä voidaan ajatella olevan kahdenlaista: reagoivuus ja skaalautuvuus. Reagoivuudella tarkoitetaan ohjelman kykyä vastata syötteisiin nopeasti. Skaalautuvuudella tarkoitetaan ohjelman kykyä suoriutua tehtävistään ajallaan, kun sen käyttöaste kasvaa. Käyttöasteen kasvu voi tarkoittaa esimerkiksi syötteen koon tai käyttäjien määrän lisääntymistä. Suorituskyky on mittari ohjelman ajan käytön lisäksi myös sähkön kulutukselle. [19]

Tietojenkäsittelytieteen tutkimuksessa algoritmien suorituskykyä vertaillaan tyypillisesti abstraktisti käyttäen asymptoottista suoritusaikaa. Asymptoottinen suoritusaika kertoo ohjelman suorituskyvyn muutoksesta, kun syötteen koko kasvaa. Se ei kuitenkaan kerro kestääkö jokin operaatio yhden millisekunnin vai yhden sekunnin. Useimmat kuluttajien käyttämät tietokoneohjelmat eivät käsittele riittävän suuria syötteitä, jotta niiden tehokkuutta voitaisiin kuvata tarkasti ottamalla huomioon vain syötteen koko – lajittelualgoritmeissa lasketaan tyypillisesti vain arvojen välisten vertailujen määrää, mutta ei esimerkiksi alkoiden siirtojen määrää. Esimerkiksi matemaattisesti tehoton kuplalajittelu voittaa yksinkertaisuutensa ansiosta pienillä alkiomäärillä monet elegantimmat lajitteluagloritmit todellisessa suoritusajassa.

Asymptoottisen suoritusajan laskeminen onkin järkevää, kun vertaillaan ohjelmien skaalautuvuutta erityisen raskaiden laskujen tai suurien datamäärien käsittelyssä. Tällöin ohjelman todellinen suoritusaika voi olla niin pitkä, ettei ohjelman käyttö ole käytännöllistä. Jotta ohjelman suoritusnopeutta voitaisiin optimoida, on sen suorituskykypiirteitä pystyttävä analysoimaan ennakoivasti asymptoottisen suoritusajan avulla. Tällöin on tärkeää tietää kestääkö ohjelman suoritus tunnin, päivän vai vuoden.

Kun taas vertaillaan ohjelmien kykyä reagoida nopeasti käyttäjän syötteeseen kuluttajille tyypillisillä tietokoneilla, on suoritusajan mittaus järkevintä tehdä käyttäen todellisesta suoritusaikaa asymptoottisen ajan sijaan. Syötteen suuruusluokka on tällöin tyypillisesti tunnettu, ja suoritusajat mitataan millisekunneissa. Tämä sekä yksinkertaistaa että monimutkaistaa suorituskyvyn mittausta; ajan mittaukseen tarvitaan vain kutsu käyttöjärjestelmän todellisen ajan palauttavaan käskyyn; toisaalta suoritusympäristö tekee tuloksista epätarkkoja ja käyttöjärjestelmän ajastimen rajallinen tarkkuus heikentää tulosten tarkkuutta eteenkin pienillä arvoilla. Silti todellinen aika on tarkempi mittari suorituskyvylle kuin asymptoottinen aika, kun vertaillaan kestääkö jossain operaatiossa yksi vai kaksi millisekuntia.

Näiden kahden ääripään – matemaattisen laskennan ja kuluttajasovellusten – välimaastoon asettuu myös joitain tapauksia, kuten videon ja kuvien muokkaus ja pakkaus, jotka ovat raskaita, mahdollisesti minuuttien tai tuntien mittaisia operaatioita, mutta joiden nopeus on silti käyttäjälle keskeistä. Niiden arvioinnissa voidaan hyödyntää eri osa-alueilla sekä asymptoottisen että todellisen suoritusajan laskentaa.

3 Vertailtavat puurakenteet

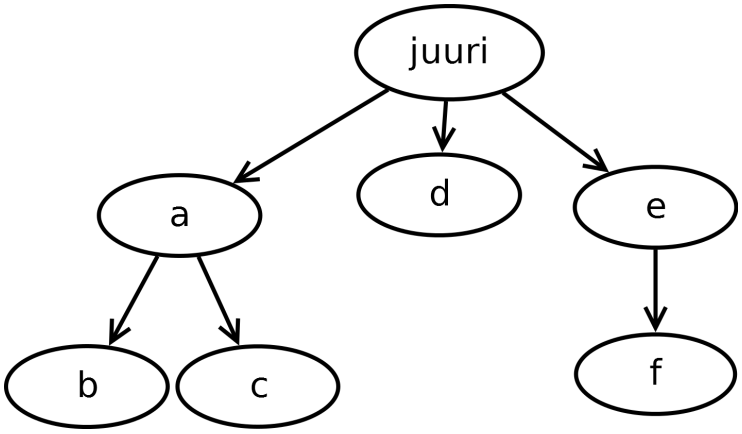
Tässä tutkielmassa vertaillaan kolmea k-arista puurakennetta. Näistä kaksi ovat yleisesti tunnettuja ja käytössä olevia tietorakenteita. Kolmas taulukoista koostuva puu on tiettävästi uusi.

Puut esittävät ohjelmistotuotannossa käytettyjä puurakenteita, joita käytetään datan hierarkiseen järjestämiseen – niillä on data-kenttä mutta ei avainkenttää. Testeissä data-kenttä sisältää neljä 32-bittistä liukulukuarvoa (yhteensä 16 tavua), jotka kuvastavat widgettimäistä kaksiulotteisista sijainneista ja ko'oista muodostuvaa transformaatiohierarkiaa. Tämä datan koko valittiin, koska se on riittävän pieni, jotta puun itsensä aiheuttamat muistihudit eivät täysin peity, mutta riittävän suuri, jotta sen voisi kuvitella olevan käytännössä hyödyllinen.

Osoittimista muodostetun puun hyväksi puoleksi mielletään yleisesti sen muokkaamisen tehokkuus ja helppous. Solmun lisäämis-, poistamis- ja siirto-operaatiot vaativat tyypillisesti vain yhden tai kahden osoittimen muuttamista. Huonona puolena osoittimista koostuvan puun solmujen läpikäyminen on lähes väistämättä rekursiivinen operaatio, ja vaatii epälineaarisesti muistiin tallennettujen solmujen lukemista. Tämä altistaa operaation välimuistihudeille.

Taulukkoon pohjautuvan puun huonoksi puoleksi voidaan mieltää sen muokkaamisen tehottomuus. Muokkausoperaatiot vaativat mahdollisesti isojen muistialueiden siirtelyä. Nykyaikaiset tietokoneet ovat kuitenkin tehokkaita tässä, minkä ansiosta siitä aiheutuva suorituskyvyn menetys osoittautuu yllättävän pieneksi. Toisena hyvänä puolena on, että taulukkoon pohjautuvan puun läpikäymiseen riittää yksinkertainen silmukka.

Osoittimista koostuvan puurakenteen suorituskykyä voidaan parantaa käyttämällä muistivarantoa, eli tallentamalla sen solmut tiiviisti yhtenäisiin muistilohkoihin, jolloin yhteen välimuistilinjaan päätyy todennäköisemmin useita toisiinsa liittyviä solmuja. Testeissä ei olla pyritty edistämään toisiinsa liitty-

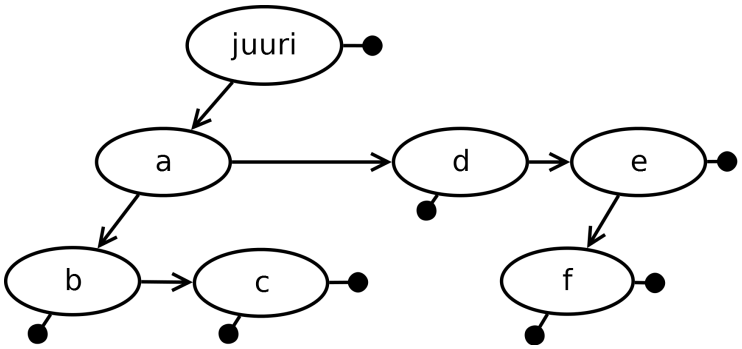


Kuva 4: Mallipuu

vien solmujen läheisyyttä muistivarannossa, vaan solmut on sijoiteltu varantoon mielivaltaisesti lisäysjärjestyksessä.

Taulukkoon pohjatuvan puun linearisuuden ansiosta, sen käsittelyssä on mahdollista hyödyntää prosessorin *rinnakkaislaskentaoperaatioita* (SIMD; Single Instruction, Multiple Data).

Kaikkia kolmea puurakennetta esitellään kuvan 4 mallipuun avulla.



Kuva 5: Vasen-lapsi oikea-sisar -puun muistiesitys. Nuolet ja mustat pallot kuvastavat muistiosoittimia.

3.1 Vasen-lapsi oikea-sisar -puu

Vasen-lapsi oikea-sisar -puu (engl. left-child right-sibling binary tree), jota tässä tutkielmassa kutsutaan luettavuuden vuoksi tästedes binääripuuksi, on rakenteeltaan sama kuin tavanomainen binääripuu, mutta kahden lapsisolmun sijaan solmuissa on viittaukset yhteen lapsi- ja yhteen sisarsolmuun kuvan 5 esittämällä tavalla. Solmuilla voi olla rajoittamaton määrä lapsisolmuja, ja ne löytyvät ensimmäisen lapsisolmun sisarsolmuista koostuvasta linkitetystä listasta.

Tässä tutkielmassa vertaillun binääripuun solmut ovat seuraavan koodin mukaisia.

```
struct Node
{
  Data data;
  Node* left_child;
  Node* right_sibling;
};
```

Solmuilla ei ole osoitinta isäsolmuunsa, mikä tekee puussa juurta kohti kulkemisesta mahdotonta. Solmun isäsolmun voi löytää vain käymällä koko puun läpi juuresta alkaen. Tämä heikentää puun suorituskykyä monissa operaatioissa, jos isäsolmua ja aiempia sisaria ei ole erikseen tallennettu pinoon, mikä ei ole käytännöllistä, jos ohjelman suorituksen aikana viitataan satunnaisiin solmuihin puussa.

Puurakenteen vaatiman tallennustilan kokovaatimus on pienehkö, koska jokaisessa solmussa on itse arvon lisäksi vain kaksi osoitinta. Puu olisi mahdollista koostaa tavanomaisen binääripuun tavoin suoraan taulukkoon luvussa 2.1 esitellyllä tavalla siten, että osoitteessa n olevan solmun lapset olisivat taulukkossa osoitteissa $2n + 1$ ja $2n + 2$, poistaen tarpeen osoittimille. Tällöin puu voi kuitenkin helposti vaatia eksponentiaalisesti kasvavan määrän harvaan käytettyä muistia, koska vaikka k-arinen puu, jota binääripuu kuvastaa, olisi matala, jokainen rinnakkainen sisarsolmu lisää binääri-

puun syvyyttä yhdellä. Esimerkiksi kuvan 5 puu on pienestä koostaan huolimatta tavannoisena binääripuuna tulkittuna 5 kerrosta syvä, ja vaatisi siis $2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 1 + 2 + 4 + 8 + 16 = 31$ solmun edestä tilaa taulukkomuotoon tallennettuna vain 7 solmulle. Leveät k-ariset puut tuottaisivat hyvin epätasaisesti täytettyjä taulukoita, joiden kokoa olisi vaikea arvioida ennakkoon. Lisäksi koska juurella ei ole sisarsolmuja eli binääripuuna ajateltuna vasenta lasta, olisi sellaisenaan puusta muodostettu taulukko aina vähintään puoliksi tyhjä.

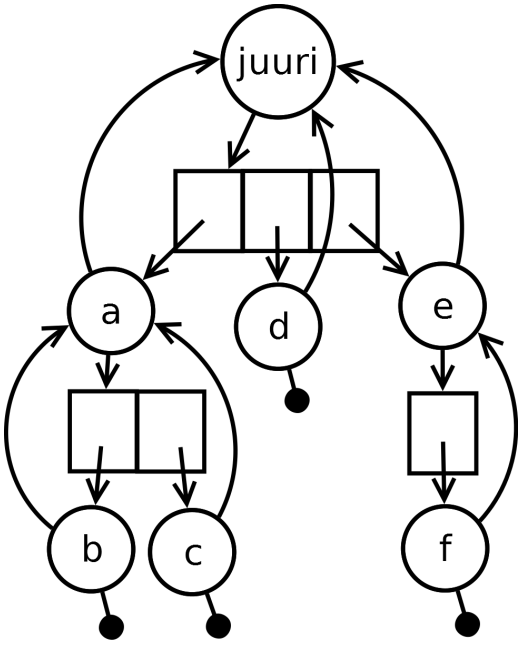
Kun puun solmut alustetaan muistivarannosta eikä satunnaisesti muistikeosta, päätyvät peräkkäin alustetut solmut vierekkäin muistissa. Jos puu on pieni tai se alustetaan suurin piirtein samassa järjestyksessä kuin, jossa se on tarkoitus myöhemmin lukea, voi muistivarannon käyttäminen mahdollistaa useamman kuin yhden relevantin solmun päätymisen samaan välimuistilinjaan, joka on tyypillisesti nykyaikaisilla tietokoneilla 64 tavua. Tämä vähentää muistihutien todennäköisyyttä, mikä puolestaan nopeuttaa puun läpikäymistä.

Testeissä binääripuun solmujen koko on 32 tavua, joista osoittimet ovat yhteensä 16 tavua ja data-kentän koko on 16 tavua. Tämän ansiosta yhteen tyypillisesti 64 tavun kokoiseen välimuistilinjaan mahtuu kaksi kokonaista solmua. Relatiivisilla osoittimilla solmun kokoa voisi pienentää entisestään.

Koska tämän puurakenteen lapsisolmut ovat solmun lapsen sisarista koostuvassa linkitetyssä listassa, voi solmun lapsien läpikäymisestä aiheutua yhtä monta muistihutia kuin solmulla on lapsia. Muistivarantoa käyttämällä voidaan mahdollisesti välttää ainakin puolet näistä.

3.2 Lapsitaulukkopuu

Lapsitaulukkopuussa voi kulkea juurta kohti, ja solmun kaikki lapset ovat aina kahden osoittimen päässä vanhemmastaan, mikä vähentää tarvetta muistihauille verrattuna binääripuuhun. Koska tyypilliset taulukkotietorakenteet koostuvat osoittimen lisäksi myös taulukon nykyisen alkiomäärän ja kapasi-



Kuva 6: Lapsitaulukkopuun muistiesitys. Nuolet ja mustat pallot kuvastavat muistiosoittimia, ympyrät solmuja ja neliöt solmujen omistamia lapsiosoitintaulukoita.

teetin kertovista kokonaisluvusta, on solmujen koko melko suuri. Koska tyypillisissä taulukkototeutuksissa taulukon käyttämä muistialue on muualla muistissa kuin itse solmu, on sen noutamiseksi suoritettava muistihaku ennen kuin lapsisolmuja voidaan tarkastella. Puun muuttuessa lapsisolmujen isäsolmuosoittimen tilaa on ylläpidettävä erikseen solmun lapsiosoittimien lisäksi.

Tässä tutkielmassa vertaillun lapsitaulukkopuun solmut ovat seuraavan koodin mukaisia.

```
struct Node
{
  Data data;
  Node* parent;
  Array<Node*> children;
};
```

Kuvan 6 kaltaisessa pienessä puussa lapsitaulukkopuu kuluttaa suhteessa suuren määrän ylimääräistä muistia osoittimiin ja taulukoihin. Leveämmässä

puussa, jossa lapsisolmutaulukoilla on enemmän jäseniä, muistia tuhlautuu suhteessa vähemmän solmua kohden. Tällöin myös muistihakujen määrä vähenee verrattuna binääripuuhun, koska yhden solmun jokainen lapsisolmuosoitin on yhdessä taulukossa toisin kuin binääripuussa, jossa jokaisen lapsisolmuosoittimen noutamiseksi on tehtävä uusi muistihaku. Molemmissa puurakenteissa itse lapsisolmun noutaminen vaatii kuitenkin vielä yhden muistihaun.

Kun puun solmut alustetaan muistivarantoon eikä satunnaisesti muistikeosta, päätyvät peräkkäin alustetut solmut vierekkäin muistissa. Jos puu on pieni tai se alustetaan suurin piirtein samassa järjestyksessä kuin, jossa se on tarkoitus myöhemmin lukea, voi muistivarannon käyttäminen mahdollistaa useamman kuin yhden relevantin solmun päätymisen samaan välimuistilinjaan. Tämä vähentää muistihutien todennäköisyyttä, mikä puolestaan nopeuttaa puun läpikäymistä.

Testeissä lapsitaulukkopuun solmujen koko on 48 tavua, mikä koostuu 16 tavun kokoisesta data-kentästä, 8-tavuisesta isäsolmuosoittimesta ja 24-tavuisesta taulukkoalkiorakenteesta, josta puolestaan sekä alkiomäärä, kapasiteetti että osoitin käyttävät jokainen 8 tavua. Tämän vuoksi yhteen 64-tavuiseen välimuistilinjaan mahtuu vain yksi kokonainen solmu. Pienemmällä data-kentällä tai pienemmillä, relatiivisilla osoittimilla solmun kokoa voitaisiin pienentää tarpeeksi, jotta useampi kuin yksi solmu mahtuisi yhteen välimuistilinjaan. Voitaisiin myös harkita taulukon koon tai isäsolmun pitämistä lapsitaulukon ensimmäisenä jäsenenä, mutta tämä pakottaisi jokaisen puun solmun varaamaan itselleen lapsitaulukon, vaikkei niillä olisi lapsisolmuja. Taulukon muistialue voitaisiin myös varata muistivarannosta välimuistikäyttäytymisen parantamiseksi, mutta tätä ei ole tehty näissä testeissä.

Jos jossain lapsitaulukkopuun käyttötapauksessa solmuilla olisi tyypillisessti pieni määrä lapsisolmuja, voitaisiin muistilinja täyttää lapsisolmuosoittimilla 64-tavuun asti. Tämä voisi tapahtua esimerkiksi siten, että taulukoon koon ollessa vähemmän kuin 5, solmun muistin jälkimmäinen osuus tulkittaisiin lapsisolmuosoittimiksi seuraavan koodin osoittamalla tavalla.

```
struct Node
{
  Data data;
  Node* parent;
  int child_count;
  union
  {
    struct
    {
      int data_capacity;
      Node** data_pointer;
    };
    struct
    {
      Node* child_node_0;
      Node* child_node_1;
      Node* child_node_2;
      Node* child_node_3;
    };
  };
};
```

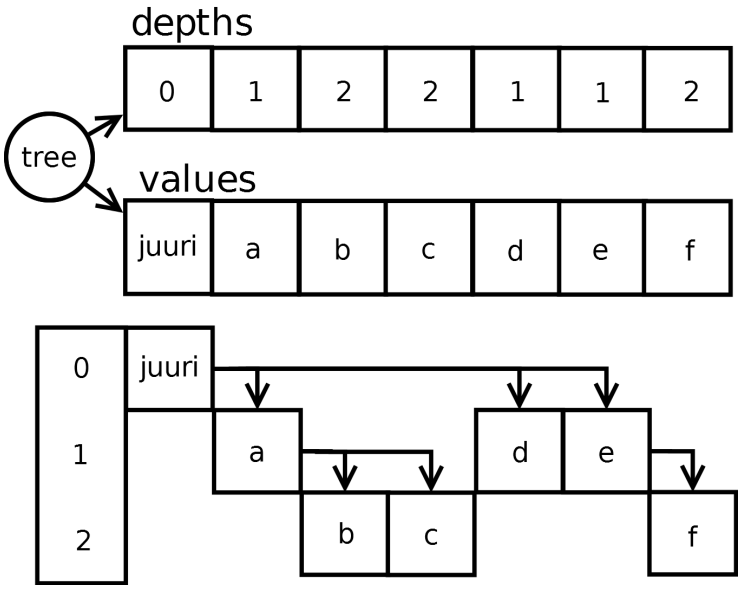
Tällä rakenteella vältettäisiin pienillä lapsisolmumäärillä yksi muistihuti, joka aiheutuu taulukon osoittaman muistin noutamisesta. Huonona puolena tälle ratkaisulle on, että aina lapsisolmuja käsiteltäessä olisi tarkastettava, solmun lapsien määrä, jotta voidaan tietää, miten solmun muistia on tulkittava. Silti muistihuteihin verrattuna tästä aiheutuvat suorituskykyhaitat olisivat todennäköisesti melko pienet. Tässä tutkielmassa ei olla vertailtu tätä versiota lapsitaulukkopuusta, koska tyypillisesti käytettyt versiot lapsitaulukkopuusta eivät ole toteutettu näin – sen sijaan monet tehokkaat merkkijonotietorakenteet käyttävät tätä vastaavaa toteutusta.

3.3 Syvyystaulukkopuu

Tässä tutkielmassa käytettävän syvyystaulukkopuun toteutus vastaa seuraavaa tietorakennetta.

```
struct Tree
{
  Array<int>  depths;
  Array<Data> data;
};
```

Poiketen luvuissa 3.1 ja 3.2 esitellyistä puurakenteista, tässä tietorakenteessa solmua ei vastaa mikään konkreettinen tietorakenne. Puu koostuu kuvan 7 mukaisesti kahdesta erillisestä taulukosta: `depths`-taulukossa on kunkin solmun syvyys puussa ja `data` taulukossa on kunkin solmun arvo. Tässä puussa solmun voidaan ajatella olevan indeksi, joka viittaa samanaikaisesti sekä `depths` että `data` taulukkoon.



Kuva 7: Ylempi kuva vastaa syvyystaulukkopuun koostumusta muistissa. Alempi kuva kertoo, miten `depths` ja `data` taulukot tulkitaan puuksi. Alemman kuvan nuolet kuvastavat vain loogista vanhemmuussuhdetta solmujen välillä, eikä niitä vastaa mikään konkreettinen arvo muistissa.

Osoittimien sijaan puun topologian määrittää järjestys. Solmun jälkeläissolmuja ovat ne kaikki ne indeksit, joiden `depth`-arvo on suurempi kuin sen, ja jotka ovat taulukossa solmun jälkeen ennen seuraavaa solmua, jonka `depth` -arvo on piempi kuin tämän. Vastaavasti solmun isäsolmu on ensimmäinen tätä edeltävä pienemmän `depth`-arvon omaava solmu. Taulukoiden sisäinen järjestys määrittää puun topologian.

Syvyystaulukkopuun järjestyksen voi tiivistää yhteen rekursiivinen sääntöön: jokaisen solmun jälkeläiset ovat aina välittömästi tämän jälkeen taulukossa. Tämä johtaa siihen, että puun solmut ovat aina syvyyshakujärjestyksessä. Tätä ajatusta selvennetään kuvan 7 alaosassa.

Voidaan ajatella, että tässä puussa solmun käsitettä vastaa taulukon indeksi. Jos solmusta haluttaisiin tehdä konkreettinen käsite, voitaisiin puu koostaa taulukollisesta solmutietueita, joissa olisi `depth` numero ja `data` data-kenttä. Tätä voitaisiin kuvastaa koodilla seuraavasti.

```
struct Node
{
  int  depth;
  Data data;
};
struct Tree
{
  Array<Node>  nodes;
};
```

Tässäkin toteutuksessa `nodes` taulukon olisi oltava syvyyshakujärjestyksessä. Suorituskykyvyn vuoksi puu kuitenkin tallennetaan kahteen erilliseen taulukkoon ilman konkreettisia solmutietueita.

Syvyyksien ilmaisemiseen voidaan käyttää mitä tahansa numerotyyppiä. Suuribittiset numerot (32- ja 64-bittiset) mahdollistavat syvemmät puurakenteet. Vähäbittiset numerot (8- ja 16-bittiset) parantavat puun suorituskykyä, koska yhteen välimuistilinjaan mahtuu tällöin enemmän syvyysarvoja, ja SIMD-rinnakkaislaskentaoperaatiot pystyvät käsittelemään niitä useamman yhtä ai-

kaa.

Jos syvyyksien ilmaisemiseen käytetään 16-bittisiä numeroita, voi puun syvyys yhä olla suuruudeltaan yli 65 000 tasoa. Koska yksi syvyysarvo on tällöin vain 2 tavua, mahtuu yhteen 64-tavuiseen välimuistilinjaan 32 solmun edestä puun topologiaa. Jos haetaan vain tiettyä lapsisolmua esimerkiksi syvyyden perusteella, milloin data-kentän arvoa ei tarvitse hakea, voi puun selaaminen nopeutua merkittävästi. Vaikka haussa tarvittaisiin data-kentän arvo, myös niitä mahtuu yhteen välimuistilinjaan maksimaalinen määrä (testeissä 4 kappaletta), koska niiden välillä ei ole tarpeetonta dataa.

Luvuissa 3.1 ja 3.2 esitellyistä puurakenteissa, data-kentän läsnäolo solmutietueessa hidastaa puun selaamista, myös silloin kun arvoa ei tarvita puun läpikäymiseen, koska se vie tilaa välimuistilinjalta. Esimerkki tapauksesta jossa puuta selataan ilman tarvetta data-kentän arvolle, on esimerkiksi luvussa 5.5 testattu lehtisolmuun siirtyminen, toinen esimerkki voisi olla kaikkien puun lehtisolmujen hakeminen.

4 Puurakenteiden kokeellinen vertailu

Tässä luvussa esitellään luvun 3 tietorakenteiden suorituskykymittaamiseen käytetyt operaatiot ja määritellään testausympäristö. Näiden lisäksi selvitetään testauksen valmistelussa suoritettavat toimenpiteet puhtaan alkutilan takaamiseksi.

4.1 Testatut operaatiot

Puiden erilaisten muistirakenteiden vertailussa käytetään seuraavia operaatioita:

- lehtisolmun lisääminen,
- alipuun poisto,
- alipuun siirtäminen,
- solmun hakeminen järjestysluvulla,
- siirtyminen juuresta lehteen,
- puun syvyyden laskeminen,
- solmujen lukumäärän laskeminen,
- solmun etsiminen data-kentän perusteella ja
- solmujen arvokenttien summaus rekursiivesti.

Jokaiselle operaatiolle on oma alilukunsa, jossa operaatio selitetään tarkemmin.

Testeissä käytetty solmujen hyötykuorma-arvo on kevyt matriisimainen rakenne, joka kuvastaa sijaintia ja kokoa yhteensä neljällä 32-bittisellä liukulukuarvolla. Tällä pyritään esittämään puumuotoista transformaatiomatriisihierarkiaa, jollaisia voi nähdä esimerkiksi pelimoottoreissa. Tässä operaatiossa matriisien arvot lasketaan yhteen siten, että juuresta lähtien solmun matriisi kerrotaan yhteen sen lapsien matriisien kanssa, ja kunkin laskutoimituksen tulos kerrotaan kunkin lapsisolmun omien lapsien matriisien kanssa.

4.2 Välimuistin tyhjennys

Ennen jokaista testattavaa operaatiota prosessorin välimuisti tyhjennetään. Tyhjennys suoritetaan hakemalla keskusmuistista suuri määrä dataa kunnes prosessorin välimuisti on oletettavasti tyhjennetty merkityksellisestä datasta.

Välimuistin tyhjennys vie huomattavasti suuremman osan testiohjelman suoritusajasta kuin itse operaatioiden suoritus. Prosessorin näkökulmasta testiohjelma pääasiassa siirtelee dataa edestakaisin keskusmuistissa, ja toisinaan suorittaa joitain laskutoimituksia.

4.3 Testausympäristö

Testit suoritettiin tietokoneella, jossa on Intel i5-2500k prosessori (4 ydintä, 64 Kt L1-välimuistia jokaiselle ytimelle, 2 kappaletta 256 Kt kahden ytimen kesken jaettua L2-välimuistia, ja 6 Mt kaikkien ytimien kesken jaettua L3-välimuistia, maksimi kellotaajuus 3,3 GHz) ja 16 Gt DDR3-1333 muistia. Testikoneen käyttöjärjestelmänä toimii Microsoft Windows 7. Testiohjelma käyttää yhtä säiettä.

4.4 Operaatioiden suoritusaikojen määrääminen

Testatut operaatiot suoritetaan yksitellen, ja suoritusajat määrätään erikseen kullekin operaatiolle ja havainnot tallennetaan muistiin. Testin lopuksi suodatetaan havaintoja siten, että mediaanihavainnon ympäriltä otetaan mukaan 80 % kaikista havainnoista, jonka jälkeen määrätään näiden keskiarvo. Maksimi ja minimi määrätään keskiarvoina 10 % ylempiä ja alempia arvoja.

Tuloksien esittämisessä käytetään keskiarvoa mediaanin sijaan, koska merkittävä osa operaatioista kestää vähemmän aikaa kuin Windowsin tarkkuusajastimen pienin aikayksikkö. Operaatioita ei voida suorittaa useita kertoja

peräkkäin yhtä mittausta varten, koska ensimmäisen suorituksen jälkeen tietue on prosessorin välimuistissa, ja se on tyhjennettävä puhtaan alkutilanteen varmistamiseksi.

Kaikki mitattavien operaatioden vaatimat valmistelut hoidetaan ennen välimuistin tyhjennystä ja ajastuksen alkua. Esimerkiksi siirto-operaatio vaatii valitun solmun lisäksi isäsolmun, jonka lapsisolmuksi kyseinen solmu on tarkoitus siirtää. Tällöin sekä isäsolmun että valitun solmun osoite haetaan valmiiksi, jotta aikamittauksessa otettaisiin huomion vain itse operaation suoritus.

Tilanne, jossa kaikki tiettyä operaatiota varten vaadittava valmistelu on tehty, mutta jossa puusta ei ole mitään välimuistissa, ei ole lainkaan realistinen. Testit päätettiin kuitenkin suorittaa tällä tavalla, koska, jotta puille tehtäviä operaatioita voitaisiin vertailla, on niistä aiheutuvien välimuistihutien tapahduttava aikamittauksen aikana.

Testilaitteen ajastimen pienin aikayksikkö on 0,3198 mikrosekuntia, mikä asettaa rajoja lyhyiden mittausaikojen tarkkuudelle. Tätä pyritään korjaamaan suorittamalla testin monta kertaa ja ottamalla tuloksista keskiarvo.

4.5 Testien suorittaminen

Testit toistetaan puille, joiden koot ovat väliltä 10–25000 solmua. Jokainen yksittäinen operaatio toistetaan 1000 kertaa peräkkäin. Kaikkien operaatioiden kellotuksen jälkeen koko testi ajetaan samalle puun koolle uudelleen 10 kertaa. Puu kootaan joka toistolla eri muotoiseksi.

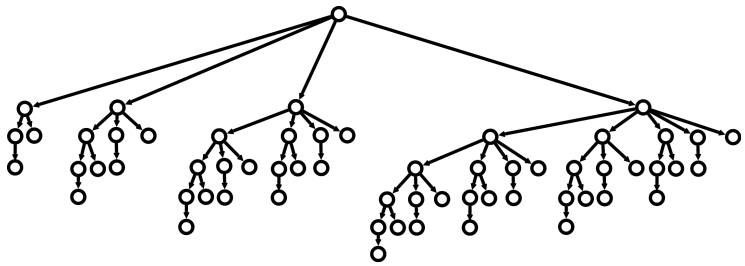
Suurella toistojen määrällä pyritään parantamaan suoritusajaltaan lyhyiden testien mittaustulosten tarkkuutta. Windowsin tarkkuusajastimen pienin aikayksikkö on suuruusluokaltaan satoja nanosekuntteja. Tästä syystä erityisesti lyhyet mittaukset on toistettava useita kertoja, jolloin keskiarvosta voidaan saada jokseenkin totuutta vastaava tulos.

4.6 Testipuiden muodostaminen

Keskenään vertailukelpoisten tulosten saavuttamiseksi puiden muodostukseen käytetään pseudosatunnaisuutta. Tällöin puiden solmujen data-kentät ovat erilaiset peräkkäisissä testeissä, mutta ne ovat samat jokaiselle eri puurakenteelle.

Puiden muodostuksessa käytetään satunnaista mutta hallittua algoritmia, koska täydestä satunnaisuudesta voisi helposti seurata epätavallisen syviä puita; jos jossain puun haarassa on enemmän solmuja kuin muissa, on todennäköisintä, että satunnaiseen solmuun lisätty solmu päätyy myös tähän haaraan. Näissä testeissä pyritään mallintamaan sellaisia puita, joiden muoto on enemmän leveä kuin syvä, kuten ihmisten rakentamat hierarkiat kuvassa 1.

Puiden topologia muodostetaan kahdessa vaiheessa, joista ensimmäinen luo säännölisen fraktaalimaisen puun (tarkemmin luvussa 5.1), ja jälkimmäinen on satunnainen, jossa puuhun lisätään satunnaisiin kohtiin lehtisolmuja. Näissä testeissä ensimmäiset 70 % solmuista lisätään ensin säännöllisesti ja loput lisätään satunnaisesti. Tällä pyritään mukailemaan luonnollista puurakennetta, jossa solmuilla on vaihteleva määrä lapsia, ja jotkin haarat ovat syvempiä kuin toiset. Tämä epäsäännöllisyys myös estää kääntäjää ja prosessoria optimoimasta puun selaamista, mikä voisi olla mahdollista, jos puu olisi vaikkapa säännöllinen binääri- tai kahdeksikkopuu.



Kuva 8: Testattavan puun topologia muistuttaa fraktaalia. Todellisuudessa testeissä käytetyn puun sisarsolmut ovat keskenään sekoitettu, jotta puu ei ole aina säännöllisesti syvempi vasemmalta ja lyhyempi oikealta.

Solmujen luonnin säännöllisessä vaiheessa juurisolmulle lisätään lapsisolmuja kuvan 8 näyttämällä tavalla kunnes testin vaatimasta puun koosta 70 % on saavutettu. Juuren lapsien omien lapsisolmujen lukumäärä määräytyy tämän järjestyksen mukaan; juuren ensimmäisellä lapsella on kolme, toisella neljä, kolmannella viisi lasta jne. Näiden solmujen kaikilla jälkeläissolmuilla puolestaan on omia lapsisolmuja yksi vähemmän kuin omalla vanhemmallaan ja yksi vähemmän jokaista aiemmin olemassa olevaan sisarusta kohden.

5 Testitulokset

Jokaisen puutyypin suoritusajat kullekin operaatiolle on seuraavassa koottu aina yhteen kuvaan. Kunkin operaation kuvasta on jätetty pois ne puutyypit joita kyseinen operaatio ei koske.

Kuvissa käytetään tilan säästämiseksi puista lyhyitä nimityksiä. Binääri on luvun 3.1 vasen-lapsi oikea-sisar -puu, lapsitaulukko on luvun 3.2 lapsitaulukkopuu, syvyystaulukko on luvun 3.3 syvyystaulukkopuu ja varanto tarkoittaa, että puun kanssa on käytetty luvussa 1.1 esiteltyä muistivarantoa.

Kaikissa kuvaajissa vaaka-akseli vastaa solmujen määrää puussa. Pystyakseli vastaa suoritusaikaa mikrosekunneissa. Vaaka-akseli on logaritminen, joten kiihtyvää vauhtia kasvavat käyrät kuvastaa todellisuudessa lineaarista kasvua.

Viivoilla olevat pisteet ovat keskiarvo keskimmäisestä 80 %:sta mittaustuloksia. Kuvaajiin on merkitty myös minimi ja maksimi keskiarvoina ylimmästä ja alimmasta 10 %:sta mittaustuloksia.

Kaikki testit valmistellaan etukäteen siten, että operaatioiden vaatimat argumentit haetaan, minkä jälkeen välimuisti pyyhitään. Esimerkiksi lehtisolmun lisääminen valmistellaan seuraavasti.

```
test_add_leaf_node(tree)
{
  node_count := count_nodes(tree);
  parent_index := get_pseudo_random_number(0, node_count - 1);
  parent_node := depth_search_node(tree, parent_index);
  new_node_data := make_pseudo_random_data();
  flush_cache();
  start_time := get_time();
  add_leaf_node(tree, parent_node, new_node_data);
  end_time := get_time();
  record_time("add_leaf_node", end_time - start_time);
}
```

Testattava operaatio ja sen argumentit vaihtelevat testikohtaisesti, mutta muuten kaava pysyy samana. Itse operaation suorittava funktio myös vaihtelee eri puutietorakenteiden välillä. Koska syvyystaulukkopuu ei käytä solmuolioita, annetaan sille testeissä sen sijaan argumentiksi solmua vastaava indeksi.

Useammassa testissä käytetään joitain toistuvia funktioita. Näiden pseudokoodi on kerätty tähän.

- taulukon koon muuttaminen

```
resize(array, new_count)
{
  if(new_count > array.capacity)
  {
    new_capacity := next_power_of_two(new_count);
    new_data := allocate_memory(size_of(Data_Type) * new_capacity
   );

    for(i := 0; i < array.count; i += 1)
    {
      new_data[i] = array.data[i];
    }
    deallocate_memory(array.data);
    array.data = new_data;
    array.capacity = new_capacity;
  }
  array.count = new_count;
}
```

- syvyystaulukkopuun solmun viimeisen lapsen hakeminen

```
find_last_child(tree, parent_index)
{
  parent_depth := tree.depths[parent_index];
  index := parent_index + 1;
  while((tree.depths.count > index) and (parent_depth > tree.
   depths[index]))
  {
```

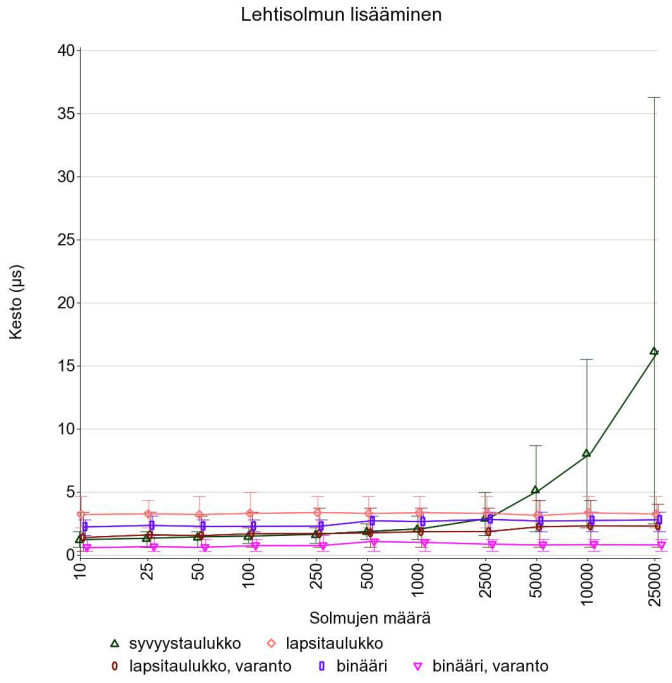
```
    index += 1;
  }
  return index - 1;
}
```

5.1 Lehtisolmun lisääminen

Lehtisolmun lisäämisessä puuhun lisätään yksi satunnaisella data-kentällä luotu solmu jonkin satunnaisen olemassa olevan solmun lapsisolmuksi. Solmu on luotu ja isäsolmu valittu sekä osoitin siihen haettu ennen operaation suoritusta. Operaatio suoritetaan seuraavan pseudokoodin osoittamalla tavalla.

- binääripuu

```
add_leaf_node(tree, parent_node, new_node_data)
{
  new_node_pointer := allocate_node(tree, new_node_data);
```



Kuva 9: Lehtisolmun keskimääräinen lisäämisaika puun koon funktiona.

```
  if(not parent_node.left_child)
  {
    parent_node.left_child = new_node_pointer;
  }
  else
  {
    sibling := parent_node.left_child;
    while(sibling.right_sibling)
    {
      sibling = sibling.right_sibling;
    }
    sibling.right_sibling = new_node_pointer;
  }
};
```

- lapsitaulukkopuu

```
add_leaf_node(tree, parent_node, new_node_data)
{
  new_node_pointer := allocate_node(tree, new_node_data);
  new_node_count := parent_node.children.count + 1;
  resize(parent_node.children, new_node_count);
  parent_node.children[new_node_count - 1] = new_node_pointer;
};
```

- syvyystaulukkopuu

```
add_leaf_node(tree, parent_index, new_node_data)
{
  node_count := tree.data.count;
  resize(tree.data, node_count + 1);
  resize(tree.depths, node_count + 1);
  last_child := find_last_child(tree, parent_index);
  for(i := node_count; i > last_child; i -= 1)
  {
    tree.data[i + 1] = tree.data[i];
    tree.depths[i + 1] = tree.depths[i];
  }
  tree.data[last_child + 1] = new_node_data;
  tree.depths[last_child + 1] = tree.depths[parent_index] + 1;
```

```
};
```

Osoittimista koostuville puille tämä on yksinkertainen operaatio: varataan muistista tilaa, alustetaan solmun muuttujat, ja lisätään tulevaan isäsolmuun osoitin. Binääripuussa solmun lisäystä varten on vielä selattava isäsolmun lapsisolmun sisaria, kunnes solmun haluttu lisäyspaikka löytyy. Syvyystaulukkopuussa lehtisolmujen paikka voi olla missä tahansa päin puun taulukkoa, joten solmun lisäämiseksi keskelle taulukkoa on sen jälkeen taulukossa olevia solmuja siirrettävä muistissa yhden alkion eteenpäin. Puiden suoritusajat voi nähdä kuvassa 9.

Molemmat osoittimista koostuvat puurakenteet suoriutuvat solmun lisäyksestä vakioajassa. Syvyystaulukkopuuhun solmun lisäämisen kustannus kasvaa lineaarisesti solmujen määrän kasvaessa. Puun koon ollessa alle 2500 solmua syvyystaulukkopuu on nopeampi kuin muistivarantoa käyttämättömät versiot osoitinpuurakenteista.

Syvyystaulukkopuun suurimman ja pienimmän suoritusajan ero kasvaa myös lineaarisesti, koska lähempänä tietorakenteen alkupäätä solmun poisto on huomattavasti kalliimpaa kuin tietorakenteen lopussa.

Erot osoittimista koostuvien puiden suorituskyvyn välillä selittyvät solmujen koon ja niiden luonnin vaatimalla vaivalla. Muistivarannoidun binääripuun ei tarvitse kuin kirjottaa pari muuttujaa ennalta varattuun muistiin. Muistivarantoa käyttämättömän osoitinpuun tarvitsee varata muistia, alustaa taulukkomuuttuja ja mahdollisesti varata vielä taulukkomuuttujalle erikseen muistia.

5.2 Alipuun poistaminen

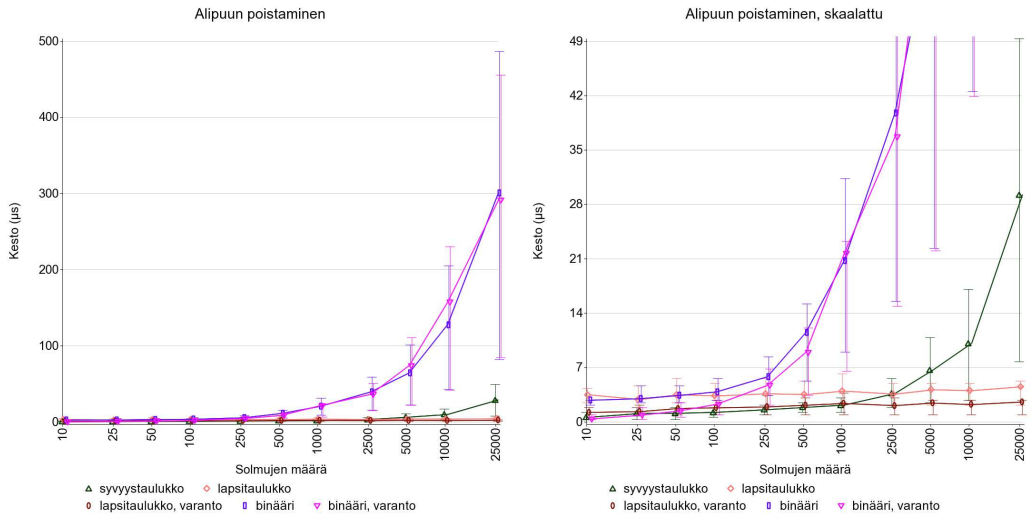
Tässä testissä puussa oleva satunnainen solmu alipuineen poistetaan puusta. Poistettava solmu on valittu ja osoitin siihen haettu ennen operaation suoritusta. Solmun isäsolmua ei kuitenkaan ole haettu valmiiksi, mikä ilmenee erityisesti binääripuun heikkona suorituskykynä, koska siinä ei ole mahdollista kulkea juurta kohti.

Poisto-operaatioiden suoritusajat eri puurakenteille on esitetty kuvassa 10. Osoitinpuille tämä testi ei vaadi juurikaan enempää kuin parin osoittimien ylikirjoittamista. Syvyystaulukkopuussa sen sijaan on tämä vaatii muistilohkojen siirtelyä.

Operaatio suoritetaan puille seuraavien pseudokoodifunktioiden mukaisesti.

- binääripuu

```
remove_node(tree, node)
{
  find_and_disconnect(tree.root, node);
```



Kuva 10: Alipuun keskimääräinen poistamisaika puun koon funktiona. Oikeanpuoleisessa kuvaajassa on y-akselia on skaalattu, jotta puiden väliset erot pienillä solmumäärillä erottuisivat paremmin.

```
  recursively_deallocate_nodes(tree, node);
};
find_and_disconnect(node, node_to_find)
{
  if(node.left_child == node_to_find)
  {
    node.left_child = node_to_find.right_sibling;
    return true;
  }
  else if(node.right_sibling == node_to_find)
  {
    node.right_sibling = node_to_find.right_sibling;
    return true;
  }
  else
  {
    if(node.left_child)
      if(find_and_disconnect(node.left_child, node_to_find))
        return true;
    if(node.right_sibling)
      if(find_and_disconnect(node.right_sibling, node_to_find))
        return true;
    return false;
  }
}
recursively_deallocate_nodes(tree, node)
{
  if(node.left_child)
    recursively_deallocate_nodes(tree, node.left_child);
  if(node.right_sibling)
    recursively_deallocate_nodes(tree, node.right_sibling);
  deallocate_node(tree, node);
}
```

- lapsitaulukkopuu

```
remove_node(tree, node)
{
  for(i := 0; i < node.parent.children.count; i += 1)
```

```
  {
    if(node.parent.children[i] == node)
    {
      erase(node.parent.children, i);
      break;
    }
  }
  recursively_deallocate_nodes(tree, node);
}
recursively_deallocate_nodes(tree, node)
{
  for(i := 0; i < node.children.count; i += 1)
  {
    recursively_deallocate_nodes(tree, node.children[i]);
  }
  deallocate_node(tree, node);
}
```

- syvyystaulukkopuu

```
remove_node(tree, node_index)
{
  last_child := find_last_child(tree, node_index);
  child_count := last_child - node_index;
  for(i := last_child; i < tree.data.count; i += 1)
  {
    tree.data[i - child_count] = tree.data[i + 1];
    tree.depths[i - child_count] = tree.depths[i + 1];
  }
  tree.data.count -= child_count + 1;
  tree.depths.count -= child_count + 1;
};
```

Osoitinpuu suoriutuu solmun poistosta vakioajassa. Syvyystaulukkopuun suoritusaika kasvaa lineaarisesti solmujen määrän kasvaessa samalla tapaa kuin solmun lisäyksessä. Solmun lisäyksestä poiketen binääripuun suoritusaika kasvaa lineaarisesti solmujen määrän kasvaessa. Binääripuun heikko suorituskyky johtuu siitä, että binääripuun solmuilla ei ole viittausta isäsolmuun, joka on

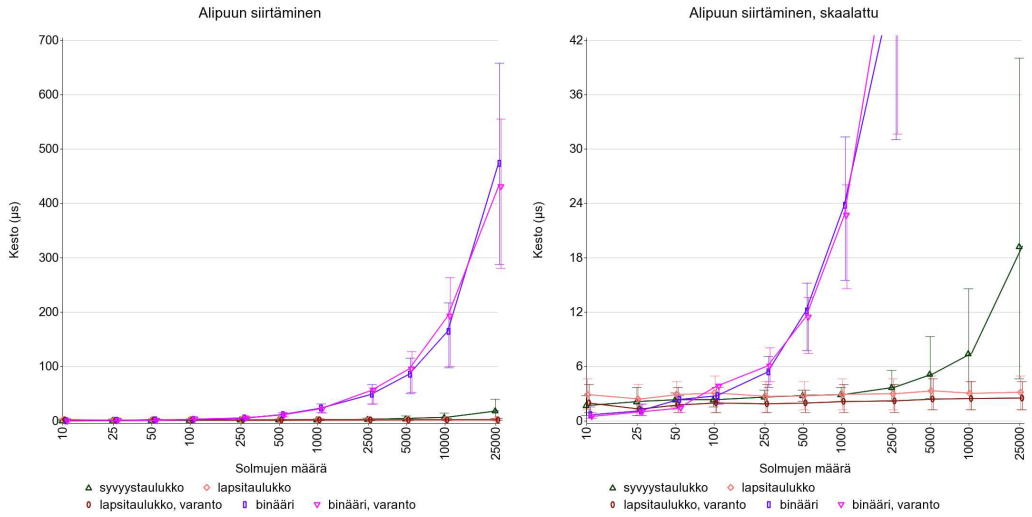
siis etsittävä jokaisen poiston yhteydessä juuresta alkaen.

5.3 Alipuun siirtäminen

Tässä testissä puussa oleva satunnainen solmu alipuineen siirretään toisen satunnaisen solmun lapseksi. Solmu ja sen tuleva isäsolmu on valittu ja osoitin niihin haettu ennen operaation suoritusta. Itse siirrettävän solmun aiempaa isäsolmua ei ole haettu valmiiksi, mikä ilmenee erityisesti binääripuun heikkona suorituskykynä, koska siinä ei ole mahdollista kulkea juurta kohti. Solmut on valittu siten, ettei solmua voida yrittää siirtää itsensä lapsihierarkiaan, jotta puuhun ei synny silmukoita.

Puiden suoritusajat voi nähdä kuvassa 11. Osoitinpuille tämä testi ei vaadi juurikaan enempää kuin parin osoittimien ylikirjoittamista. Syvyystaulukkopuussa sen sijaan on tämä vaatii muistilohkojen siirtelyä ja kopioimista.

Puille suoritettavat operaatiot ovat seuraavien pseudokoodien mukaiset.



Kuva 11: Alipuun keskimääräinen siirtämisaika puun koon funktiona. Oikeanpuoleisessa kuvaajassa on y-akselia on skaalattu, jotta puiden väliset erot pienillä solmumäärillä erottuisivat paremmin.

- binääripuu

```
move_node(tree, node, new_parent_node)
{
  find_and_disconnect(tree.root, node);
  if(not parent_node.left_child)
  {
    parent_node.left_child = new_node_pointer;
  }
  else
  {
    sibling := parent_node.left_child;
    while(sibling.right_sibling)
    {
      sibling = sibling.right_sibling;
    }
    sibling.right_sibling = new_node_pointer;
  }
}
```

- lapsitaulukkopuu

```
move_node(tree, node, new_parent_node)
{
  for(i := 0; i < node.parent.children.count; i += 1)
  {
    if(node.parent.children[i] == node)
    {
      erase(node.parent.children, i);
      break;
    }
  }
  new_node_count := new_parent_node.children.count + 1;
  resize(new_parent_node.children, new_node_count);
  new_parent_node.children[new_node_count - 1] = node;
}
```

- syvyystaulukkopuu

```
move_node(tree, node_index, new_parent_index)
```

```
{
  last_child := find_last_child(tree, node_index);
  child_count := last_child - node_index;
  new_place := find_last_child(tree, new_parent_index) + 1;
  parent_depth := tree.depths[new_parent_index];
  node_depth := tree.depths[node_index];
  depth_difference := parent_depth - node_depth + 1;
  for(i := node_index; i < last_child + 1; i += 1)
  {
    tree.depths[i] += depth_difference;
  }
  if(new_place < node_index)
  {
    rotate_array(tree.data, new_place, node_index, last_child);
    rotate_array(tree.depths, new_place, node_index, last_child);
  }
  else
  {
    rotate_array(tree.data, node_index, last_child, new_place);
    rotate_array(tree.depths, node_index, last_child, new_place);
  }
}
```

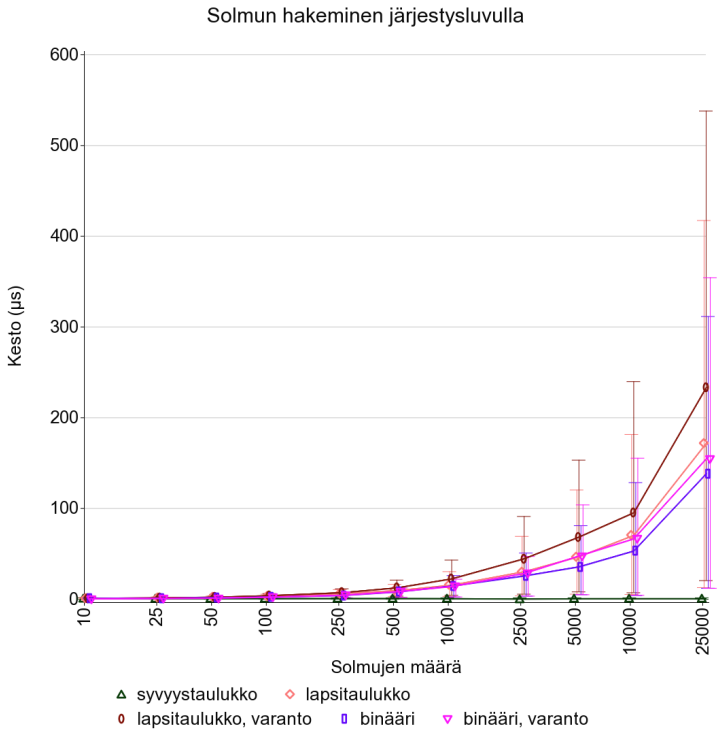
Lapsitaulukkopuu suoriutuu solmun siirrosta vakioajassa. Syvyystaulukkopuun ja binääripuun suoritusaika kasvaa lineaarisesti solmujen määrän kasvaessa. Binääripuun suorituskykyä haittaa jälleen tarve etsiä solmun isäsolmu ennen, kun solmu voidaan poistaa sen lapsista.

5.4 Solmun syvyyshaku järjestysluvulla

Tässä testissä etsitään puusta tietty solmu syvyyshaun mukaisella järjestysluvulla. Tätä operaatiota on käytetty tässä tutkielmassa mitattavien operaatioiden argumentteina käytettävien satunnaisten solmujen hakemiseen eri puurakenteista. Puiden suoritusajat voi nähdä kuvassa 12. Operaatiot ovat seuraavien pseudokoodien mukaiset.

- binääripuu

```
get_node(tree, node_index)
{
  n := node_index;
  pointer_to_n := get_pointer(n);
  return recursive_depth_search(root, pointer_to_n)
```



Kuva 12: Järjestysluvulla solmun keskimääräinen noutamisaika puun koon funktiona.

```
}
recursive_depth_search(node, pointer_to_n)
{
  n := fetch(pointer_to_n);
  if(n == 0)
    return node;
  n -= 1;
  set(pointer_to_n, n);
  if(node.left_child != null)
  {
    result := recursive_depth_search(node.left_child,
   pointer_to_n)
    if(result)
      return result;
  }
  if(node.right_sibling != null)
  {
    result := recursive_depth_search(node.right_sibling,
   pointer_to_n)
    if(result)
      return result;
  }
  return null;
}
```

- lapsitaulukkopuu

```
get_node(tree, node_index)
{
  n := node_index;
  pointer_to_n := get_pointer(n);
  return recursive_depth_search(root, pointer_to_n)
}
recursive_depth_search(node, pointer_to_n)
{
  n := fetch(pointer_to_n);
  if(n == 0)
    return node;
  n -= 1;
```

```
  set(pointer_to_n, n);
  for(i := 0; i < node.child_array.count; i += 1)
  {
    result := recursive_depth_search(node.child_array[i],
   pointer_to_n);
    if(result != null)
      return result;
  }
  return null;
}
```

- syvyystaulukkopuu

```
get_node(tree, node_index)
{
  return node_index;
}
```

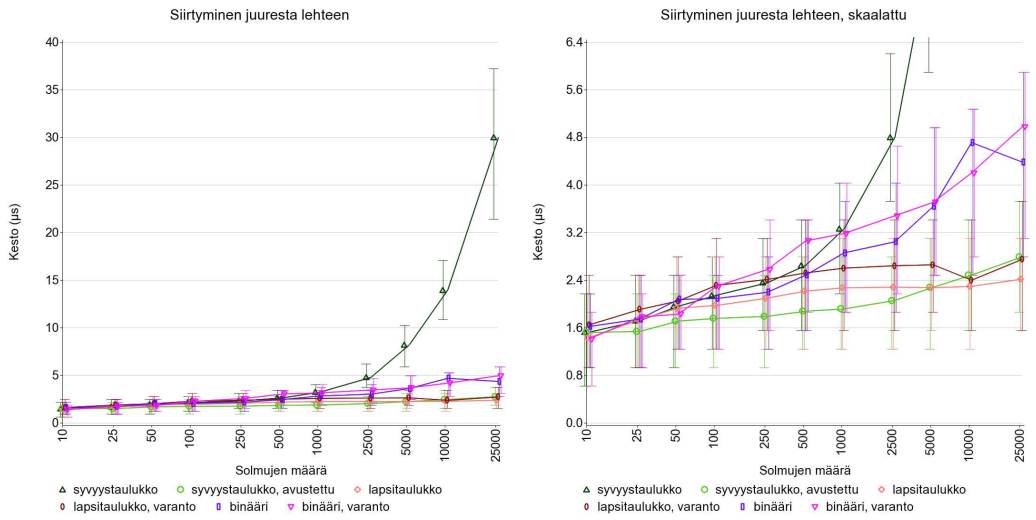
Osoitinpuiden syvyyshakujärjestykseessä selaamisen suoritusaika kasvaa lineaarisesti. Syvyystaulukkopuussa solmujen syvyyshakujärjestyksessä selaaminen on vakioaikainen indeksihaku, koska solmut ovat aina syvyyshakujärjestyksessä taulukossa.

5.5 Siirtyminen juuresta lehteen

Tässä testissä siirrytään solmu kerrallaan juuresta lehteen päin ennalta määrättyä reittiä pitkin. Reitin määrittää on pseudosatunnainen numerosarja, jonka mukaan valitaan aina jokin nykyisen solmun lapsista. Tällä operaatiolla pyritään simuloimaan solmujen lasten hajasaantia, koska puiden eriävät rekenteetn edellyttävät hyvin erilaista lasten läpikäymistä; lapsitaulukkopuussa lasten tietyn lapsen hakeminen on yksi suora muistioperaatio, binääripuussa se edellyttää linkitetyn listan läpikäymistä ja syvyystaulukkopuussa joudutaan käymään potentiaalisesti pitkäkin lista alkioita läpi ennen kun saavutetaan solmun seuraava lapsi. Syvyystaulukkopuusta on tässä operaatiossa testattavana myös versio, jolla on lisäinformaationa taulukko, jossa on jokaisen solmun seuraavan sisarsolmun indeksi. Puiden suoritusajat voi nähdä kuvassa 13. Operaatiot ovat seuraavien pseudokoodien mukaiset.

- binääripuu

```
travel_to_leaf(tree, random)
{
```



Kuva 13: Puun juuresta lehteen keskimääräinen kulkuaika puun koon funktiona. Oikeanpuoleisessa kuvaajassa on y-akselia on skaalattu, jotta puiden väliset erot pienillä solmumäärillä erottuisivat paremmin.

```
  current := tree.root;
  child_count := count_children(current);
  while(child_count > 0)
  {
    child := random(0, child_count);
    current = get_nth_child(current, child);
    child_count := count_children(current);
  }
  return current;
}
count_children(tree, node)
{
  current := node.left_child;
  count := 0;
  while(current != null)
  {
    count += 1;
    current = node.right_sibling;
  }
  return count;
}
get_nth_child(tree, node, n)
{
  current := node.left_child;
  while(n > 0)
  {
    n -= 1;
    current = node.right_sibling;
  }
  return current;
}
```

- lapsitaulukkopuu

```
travel_to_leaf(tree, random)
{
  current := tree.root;
  while(current.child_array.count > 0)
  {
```

```
    child := random(0, current.child_array.count);
    current = current.child_array[child];
  }
  return current;
}
```

- syvyystaulukkopuu

```
travel_to_leaf(tree, random)
{
  current := 0;
  child_count := count_children(current);
  while(child_count > 0)
  {
    child := random(0, child_count);
    current = get_nth_child(tree, current, child);
    child_count = count_children(tree, current);
  }
  return current;
}
count_children(tree, node_index)
{
  child_depth = tree.depths[node_index] + 1;
  current := node_index + 1;
  count := 0;
  while(tree.depths[current] >= child_depth)
  {
    if(tree.depths[current] == child_depth)
    {
      count += 1;
    }
    current += 1;
  }
  return count;
}
get_nth_child(tree, node_index, n)
{
  child_depth = tree.depths[node_index] + 1;
  current := node_index + 1;
```

```
  count := 0;
  while(true)
  {
    if(tree.depths[current] == child_depth)
    {
      count += 1;
      if(count == n)
        break;
    }
    current += 1;
  }
  return current;
}
```

Juuresta satunnaiseen lehteen siirtymisen suoritusaika kasvaa osoitinpuilla logarimisesti puun kokoon nähden. Syvyystaulukkopuulla suoritusaika on lineaarinen. Yksittäisen solmun tietyn lapsisolmun etsiminen on linearisesti lapsien määrän mukaan hidastuva operaatio, jossa joudutaan käymään läpi jokainen lapsenlapsisolmu etsittäessä solmun seuraavaa suoraa lapsisolmua.

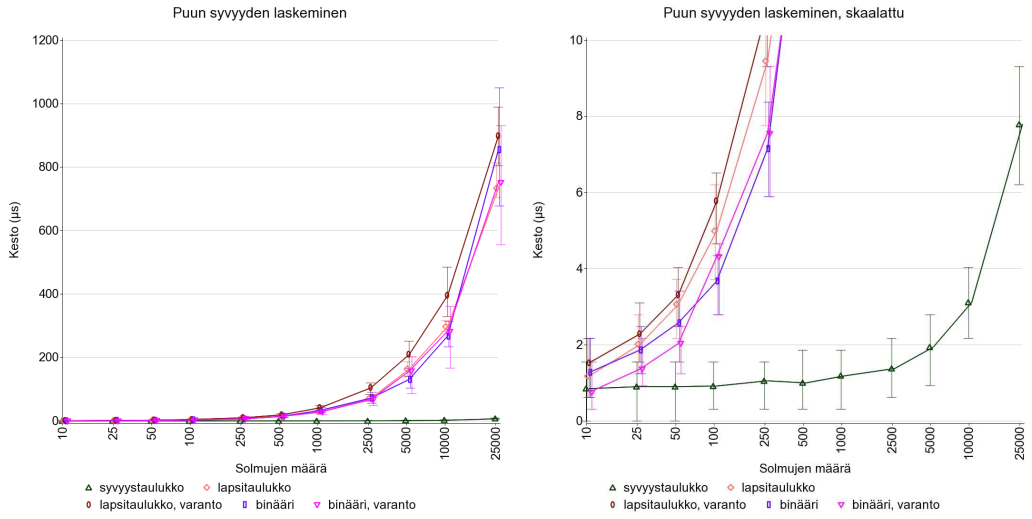
Tämä operaatio suoritettiin syvyystaulukkopuulle myös aputietorakenteen kanssa, mikä tekee operaatiosta sille logaritmisen. Tämä aputietorakenne mahdollistaa *count_children* ja *get_nth_child* funktioiden suorittamisen ilman, että on vierailtava jokaisessa lapsenlapsisolmussa. Tämän aputietorakenteen kanssa syvyystaulukkopuu suoriutuu tästä operaatiosta logaritmisessa ajassa ja nopeammin tai yhtä nopeasti kuin osoitinpuut. Tämä aputietorakenne on käytännössä puun oman syvyystaulukon mittainen taulukko, johon on tallennettu jokaiselle puun solmulle seuraavan sisarsolmun indeksi. Tämä aputietorakenne on mahdollista muodostaa lineaarisessa ajassa, joten sen käyttö on hyödyllistä mikäli syvyystaulukkopuusta on tarkoitus tehdä vähintään kaksi tällaisia hakua ilman, että puu rakenne muuttuu hakujen välissä. Aputietorakennetta on kuitenkin mahdollista päivittää osissa, jos puurakenteen tarvitsee muuttua usein.

5.6 Puun syvyyden laskeminen

Laskemalla puun suurin syvyys voidaan määrittää esimerkiksi tarvittavan väliaikaisen muistin suuruus, joillekin operaatioille, jotka vaativat välituloksia jokaiselta puun tasolta. Yksi esimerkki tästä on luvussa 5.9 testattu rekursiivinen summaus.

Puun suurimman syvyyden laskeminen vaatii kaikissa tapauksissa koko puun läpikäymisen. Osoitinpuissa tämä edellyttää rekursiivista operaatiota, jossa kunkin solmun syvyys lasketaan kasvattamalla rekursiovaiheessa laskuria yhdellä. Syvyystaulukkopuussa tämä vaatii vain syvyystaulukon nopean läpikäymisen. Puiden suorituskyky tässä on nähtävissä kuvassa 14.

Osoitinpuissa suurimman syvyyden laskeminen on lineaarinen operaatio, joka vaatii koko puun läpikäymistä rekursiivisesti. Syvyystaulukkopuulle tämä operaatio on myös lineaarinen, mutta koska solmujen syvyydet ovat lineaarisesti muistissa, voidaan sen läpikäymisessä hyödyntää tehokkaita SIMD-vektorioperaatioita. Tämän ansiosta operaatio on syvyystaulukkopuulle noin



Kuva 14: Puun syvyyden keskimääräinen laskemisaika puun koon funktiona. Oikeanpuoleisessa kuvaajassa on y-akselia on skaalattu, jotta puiden väliset erot pienillä solmumäärillä erottuisivat paremmin.

100 kertaa nopeampi kuin osoitinpuille.

Kääntäjän on mahdollista tuottaa SIMD-optimoitu lopputulos automaattisesti, mutta tätä ei ole helppo taata. Esimerkiksi Microsoftin Visual C++-kääntäjän 32-bittinen versio tuotti syvyystaulukkopuun tälle operaatiolle SIMD-optimoidun lopputuloksen mutta 64-bittinen versio ei. Lisäksi kääntäjän tuottama tulos ei ole aina yhtä tehokas kuin manuaalisesti optimoitu; testiohjelman käsinoptimoitu versio on 6–10 % nopeampi kuin kääntäjän tuottama.

Operaatiot ovat seuraavien pseudokoodien mukaiset. Syvyystaulukosta esitellään myös SIMD-optimointia mukaileva pseudokoodi.

- binääripuu

```
find_max_depth(tree)
{
  return recurse_max_depth(tree.root);
}
recurse_max_depth(node)
{
  child := node.left_child;
  max_child_depth := 0;
  while(child != null)
  {
    child_depth := recurse_max_depth(child)
    if(child_depth > max_child_depth)
      max_child_depth = child_depth;
    child = child.right_sibling;
  }
  return max_child_depth + 1;
}
```

- lapsitaulukkopuu

```
find_max_depth(tree, random)
{
  return recurse_max_depth(tree.root)
}
recurse_max_depth(node)
```

```
{
  max_child_depth := 0;
  for(i := 1; i < node.child_array.count; i += 1)
  {
    child_depth := recurse_max_depth(child)
    if(child_depth > max_child_depth)
      max_child_depth = child_depth;
  }
  return max_child_depth + 1;
}
```

• syvyystaulukkopuu

```
find_max_depth(tree)
{
  max_depth := 0;
  for(i := 0; i < tree.depths.count; i += 1)
  {
    if(max_depth < tree.depths[i])
      tree.depths = tree.depths[i];
  }
  return max_depth;
}
```

• SIMD-optimoitu syvyystaulukkopuu (*depth* arvo on tässä esimerkissä yhden tavun kokoinen, ja käytetty SIMD-taso tukee 128 bitin eli 16 tavun käsittelyä kerrallaan)

```
find_max_depth(tree)
{
  data_pointer := tree.depths.data_pointer;
  data_left := tree.depths.count;
  result := 0;

  // Etene yksi tavu kerrallaan, kunnes data_pointer on 16:lla
  // jaollinen. Tämä on edellytys SIMD-operaatioiden toiminnalle.
  while(modulo(data_pointer, 16) != 0)
  {
    if(data_left == 0)
```

```
    return result;
  if(result < *data_pointer)
    result = *data_pointer)
  data_pointer += 1;
  data_left -= 1;
}

// Alusta 64 tavun edestä muistia neljään muuttujaan
max_0 := zero_16_bytes();
max_1 := zero_16_bytes();
max_2 := zero_16_bytes();
max_3 := zero_16_bytes();

// Vertaile 64 syvyysarvoa kerrallaan
while (data_left >= 64)
{
  in_0 := load_16_bytes(data_pointer + 16 * 0);
  in_1 := load_16_bytes(data_pointer + 16 * 1);
  in_2 := load_16_bytes(data_pointer + 16 * 2);
  in_3 := load_16_bytes(data_pointer + 16 * 3);
  max_0 = bytewise_max(max_0, in_0);
  max_1 = bytewise_max(max_1, in_1);
  max_2 = bytewise_max(max_2, in_2);
  max_3 = bytewise_max(max_3, in_3);
  data_pointer += 64;
  data_left -= 64;
}

// Vertaile 16 syvyysarvoa kerrallaan
while(data_left >= 16)
{
  in_0 := load_16_bytes(data_pointer + 16 * 0);
  max_0 = bytewise_max(max_0, in_0);
  data_pointer += 16;
  data_left -= 16;
}

// Vertaile välitulokset keskenään jotta tulos saadaan
// yhteen 16:n syvyysarvon muuttujaan
```

```
  max_0 = bytewise_max(max_0, max_1);
  max_2 = bytewise_max(max_2, max_3);
  max_0 = bytewise_max(max_0, max_2);

  // Vertaile max_0 muuttujan sisältämät 16 arvoa keskenään,
  // jotta tulos saadaan muuttujan viimeiseen tavuun
  max_0 = bytewise_max(max_0, bytewise_shift(max_0, 8));
  max_0 = bytewise_max(max_0, bytewise_shift(max_0, 4));
  max_0 = bytewise_max(max_0, bytewise_shift(max_0, 2));
  max_0 = bytewise_max(max_0, bytewise_shift(max_0, 1));
  if(result < get_last_byte(max_0))
    result = get_last_byte(max_0);

  // Vertaile 16:lla jaoton loppuosa
  // (Tämä ja alussa tehty vertailu voitaisiin jättää pois
  // algoritmista edellyttämällä, että depths-taulukko on tasattu
  // alkamaan 16 tavulla jaollisesta muistiosoittesta, ja
  // varaamalla aina 16:lla jaollinen määrä syvyysarvoja
  // kerrallaan.)
  while (data_left > 0)
  {
    if (result < *data_pointer)
      result = *data_pointer;
    data_pointer += 1;
    data_left -= 1;
  }
  return result;
}
```

Tämä testi ei ole suoraan vertailukelpoinen osoitinpuiden ja syvyystaulukkopuiden välillä, koska operaatiot on toteutettu hyvin eri tavoin. Se kuitenkin toimii esimerkkinä keinoista, joita syvyystaulukkopuun kanssa voi soveltaa syvyyksien ja ennen kaikkea data-kenttien, mikäli data on SIMD-suorittimen ymmärtämässä muodossa, käsittelemisessä.

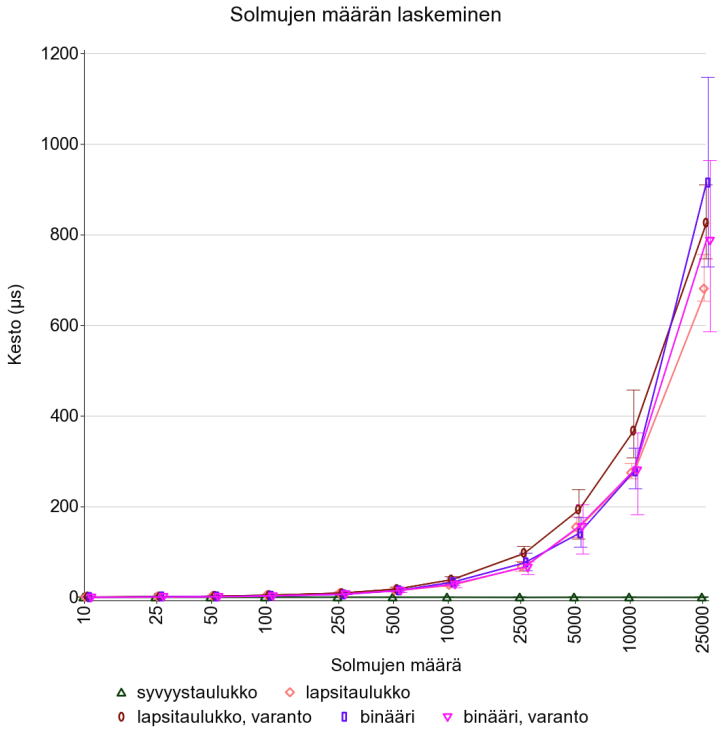
5.7 Solmujen lukumäärän laskeminen

Tässä testissä puun solmujen määrä lasketaan käymällä koko puu läpi. Solmujen arvoilla ei ole vaikutusta tulokseen.

Osoitinpuissa solmujen määrän laskeminen on lineaarinen operaatio, joka vaatii koko puun läpikäymistä rekursiivisesti. Puiden suorituskyky tässä on nähtävissä kuvassa 15. Syvyystaulukkopuulle tämä operaatio on vakioaikainen, koska se perustuu taulukkorakenteeseen, joka tietää aina oman kokonsa.

5.8 Solmun etsiminen data-kenttiä vertailemalla

Tässä testissä etsitään puusta jokin solmu vertailemalla sen data-kenttää haettavan data-kentän kanssa. Tämä voi vaatia vaihtelevasti koko puun lä-



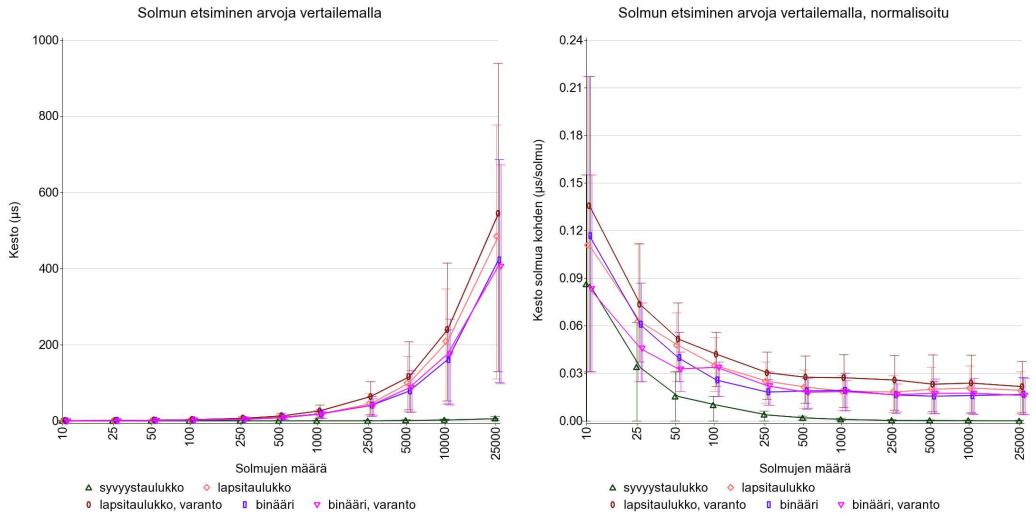
Kuva 15: Solmujen lukumäärän keskimääräinen laskemisaika puun koon funktiona.

pikäymistä tai haku voi päättyä ensimmäiseen solmuun.

Osoitinpuusta solmun etsiminen vertaamalla solmun arvoa etsittävään arvoon on lineaarinen operaatio, joka vaatii puun läpikäymistä rekursiivisesti. Puiden suorituskyky tässä on nähtävissä kuvassa 16. Syvyystaulukkopuulle tämä operaatio on myös lineaarinen, mutta vertailu on helposti tehtävissä yhdellä silmukalla.

Binääripuu suoriutuu tästä tehtävästä nopeammin kuin lapsitaulukkopuu oletettavasti sen pienemmän solmukoon ansiosta – tällöin välimuistiin on nopeampi ladata ja sinne myös mahtuu enemmän solmuja. Muistivarantoa käyttävän lapsitaulukko puun huonommalle suorituskyvylle muistivarantoa käyttämättömään versioon verrattuna, ei ole tiedossa selitystä.

Muistivarantoa käyttävien osoitinpuiden arvovertailu olisi myös mahdollista tehdä lineaarisesti käymällä läpi muistivarannossa olevia muistialueita. Mutta tällöin joudutaan varmistamaan, että löydetty arvo on vielä puussa, eikä sitä ole poistettu. Tämä voitaisiin saavuttaa ylikirjoittamalla poistetut solmut muistivarantoon hautakiviarvoilla. Tällöinkin etsintä on suoritettava ko-



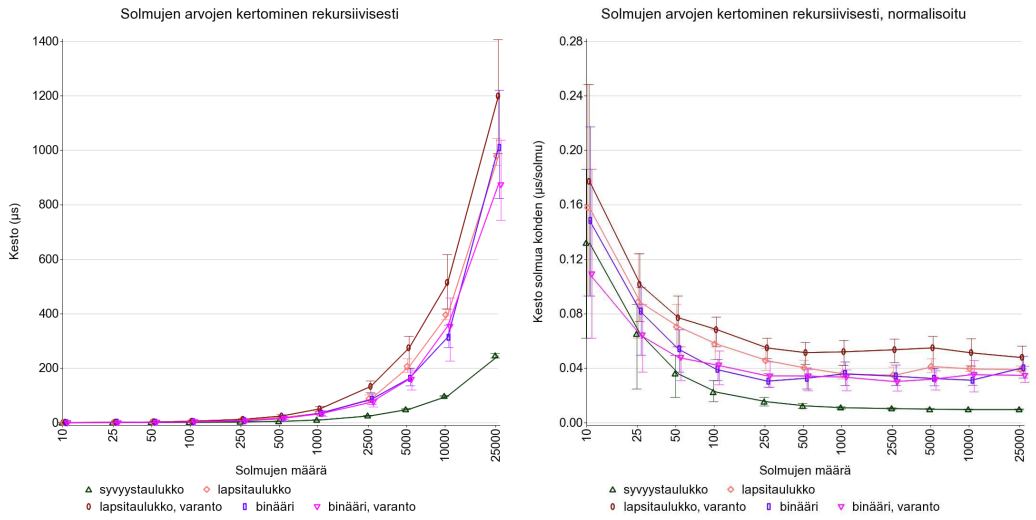
Kuva 16: Data-kentän perusteella solmun keskimääräinen etsimisaika puun koon funktiona. Oikeanpuoleisen kuvaajan y-akseli on normalisoitu solmujen määrän suhteen puiden välisten suorituskykyerojen korostamiseksi.

ko muistivarannon kapasiteetille, koska olemassa olevat solmut eivät välttämättä ole peräkkäin muistissa, eikä niiden kokonaismäärää tiedetä etukäteen.

5.9 Solmujen data-kenttien summaus rekursiivisesti

Tässä testissä kaikkien solmujen arvo summataan yhteen niiden lapsisolmujen arvojen kanssa rekursiivisesti, kunnes saavutaan lehtisolmuun, jolloin tulos kirjataan muistiin. Tämä operaatio vaatii aina koko puun läpikäymisen.

Solmujen arvojen summaus rekursiivisesti on kaikilla puilla lineaarinen operaatio, koska se vaatii puun kaikissa solmuissa käymistä. Kaikkien puiden suoritusajat tässä operaatiossa on nähtävillä kuvasta 17. Osoitinpuilla tämä tapahtuu luontevimmin rekursiivisella funktiolla. Syvyystaulukkopuulla tämä on suoritettavissa yksittäisellä silmukalla. Syvyystaulukkopuulla tämä vaatii eksplisiittistä muistia välituloksille, jotka rekursiivessa toteutuksessa säilyvät implisiittisesti funktiokutsupinossa. Tämän muistin voi varata pinosta suorituskyvyn parantamiseksi, kunhan puun syvyys ja arvon koko on riittävän



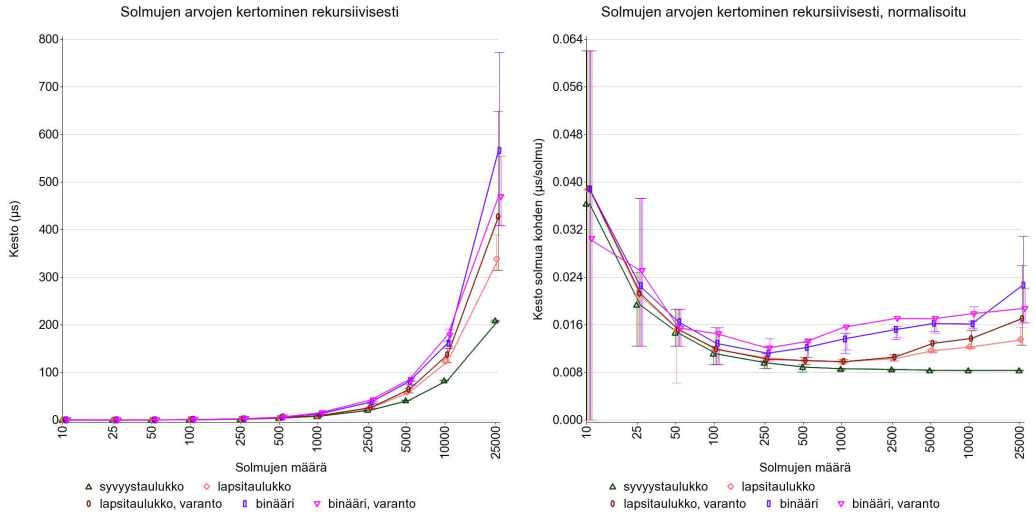
Kuva 17: Puun keskimääräinen solmujen data-kenttien rekursiivinen summausaika puun koon funktiona. Oikeanpuoleisen kuvaajan y-akseli on normalisoitu solmujen määrän suhteen puiden välisten suorituskykyerojen korostamiseksi.

pieni pinon ylivuodon välttämiseksi.

Binääripuu suoriutuu tästä tehtävästä nopeammin kuin lapsitaulukkopuu oletettavasti sen pienemmän solmukoon ansiosta – tällöin välimuistiin on nopeampi ladata ja sinne myös mahtuu enemmän solmuja. Muistivarantoa käyttävän lapsitaulukkopuun huonommalle suorituskyvylle naiiviin versioon verrattuna, ei ole tiedossa selitystä; muistirakenne ja algoritmit ovat täysin samat lukuunottamatta solmujen asettelua muistissa.

Kuvasta 18 voidaan havaita, että kaikki puut suoritutuvat solmujen läpikäynnistä nopeammin, kun tämä testi toistetaan useita kertoja peräkkäin ilman välimuistin pyyhkimistä välissä. Tästä voidaan päätellä, että erityisesti osoittimista koostuvat puut hyötyvät arvojen olemisesta välimuistissa.

Syvyystaulukkopuun suoritusaika ei muutu suuresti verrattuna ensimmäiseen ajokertaan, mistä voidaan päätellä syvyystaulukkopuun käyttävän välimuistia tehokkaasti jo ensimmäisellä ajokerralla.



Kuva 18: Sama informaatio kuin kuvassa 17, mutta toisella peräkkäisellä suorituskerralla.

5.10 Tulosten analysointi

Suoritettujen testien tuloksista voidaan nähdä, että osoittimista koostuvat puut pääsääntöisesti toimivat vakioajassa puita muokattaessa, ja että käyttämällä muistivarantoa niiden suorituskykyä voidaan parantaa havaittavasti.

Syvyystaulukkopuun suorituskyky puuta muokattaessa on lineaarinen, mutta riittävän pienillä puilla (alle 1000–2000 solmua) sen suorituskyky voi olla jopa parempi kuin osoitinpuilla.

Puun selaamiseen keskittyvissä operaatioissa syvyystaulukkopuu on huomattavasti tehokkaampi kuin osoittimista koostuvat puut. Kaksi todennäköistä syytä tälle ovat yksinkertaisemman tietorakenteen mahdollistamat yksikertaisemmat funktiot ja lineaarisen muistialueen käytöstä seuraava muistin tiiviys.

Poikkeuksena tähän on juuresta lehtisolmuun matkustaminen, jossa syvyystaulukkopuun sisarsolmujen etäisyys toisistaan vaatii lapsisolmujen yli hyppimistä. Tämäkin ongelma on helposti ratkaistavissa aputietorakenteella, johon on kirjattu jokaisen solmun seuraavan sisaren indeksi.

Käyttökohteesta riippuen voi siis olla järkevää käyttää kumpaa tahansa puutyyppiä. Jos käyttökohde vaatii suurien puiden rakentamista, mutta ei juurikaan lukemista, osoitinpuut ovat järkevä valinta.

Jos puu on tarkoitus rakentaa vain kerran, ja sitä muokataan harvoin verrattuna lukemiseen, syvyystaulukkopuu on hyvä vaihtoehto.

Jos puu on suuri, ja sitä muokataan usein, mutta lukemisen on oltavata tehokasta, voi olla järkevintä käyttää osoitinpuuta puun rakentamiseen ja datan säilyttämiseen, ja koostaa puusta lukemista varten syvyystaulukkopuu väliaikaiseksi kiihdytysrakenteeksi.

6 Yhteenveto

Tässä tutkielmassa vertailtiin kolmen puutietorakenteen suorituskykyä. Kaksi näistä olivat tyypillisiä osoittimista koostuvia puurakenteita. Kolmas puu koostuu kahdesta taulukosta, joissa ei ole eksplisiittistä solmun käsitettä, vaan solmua vastaa indeksi taulukoissa. Osoittimista koostuvista puista testattiin myös muistia optimaalisemmin käyttävät muistivarantolliset versiot.

Suoritettujen testien tuloksista käy ilmi, että

- muistivarannon käyttäminen osoitinpuilla parantaa niiden suorituskykyä muokkausoperaatioissa 10–80 %, mutta ei vaikuta merkittävsti lukuoperaatioissa,
- syvyystaulukkopuu on muokkausoperaatioissa tehokkuudeltaan samaa luokka kuin osoittimista koostuvat puut, jos solmujen määrä on alle 2000 solmua, ja
- syvyystaulukkopuun selaaminen on pääsääntöisesti 20–90 % tehokkaampaa kuin osoittimista koostuvien puiden – poikkeuksena siirtyminen juuresta lehteen, mitä voi kuitenkin tarvittaessa nopeuttaa aputietorakenteella.

Syvyystaulukkopuu voidaan rakentaa suurillekin puille nopeasti, jos solmut lisätään syvyyshakujärjestyksessä, koska uudet solmut voidaan tällöin lisätä puun taulukoiden loppuun. Tämän ansiosta sillä voisi olla käyttöä kiihdytysrakenteena osoittimista koostuvalle puulle tilanteessa, jossa puun selaamisen on oltava mahdollisimman nopeaa. Tällöin solmujen määrällä ei olisi merkitystä puun muodostamisen tehokkuuden kannalta.

Eri puutyypeistä olisi myös mahdollista yhdistää hybridipuu, jonka yksi tai useampi ensimmäinen kerros olisi osoittimista koostuvaa puuta, ja lehtiosuus koostuisi taulukkopohjaisista alipuista. Tällöin taulukkopuun koko olisi help-

po pitää alle 2000 solmussa ja osoitinpuussa osoittimia pitkin hyppimisen tarve vähenisi taulukkopuun syvyyden verran.

Tässä tutkielmassa sivuttiin vain pikaisesti rinnakkaislaksentaoperaatioiden käyttöä taulukkopohjaisen puun kanssa syvyyksien vertailuun. Joissakin tapauksissa rinnakkaislaskentaoperaatioita voisi olla mahdollista käyttää myös arvojen vertailuun ja muokkaamiseen.

Viitteet

[1] Alan Ezust, Paul Ezust, *An Introduction to Design Patterns in C++ with Qt 4*, www.prenhallprofessional.com/perens, (2006).

[2] Anand Lal Shimpi, *AMD's Jaguar Architecture: The CPU Powering Xbox One, PlayStation 4, Kabini & Temash*, http://www.anandtech.com/show/6976/amds-jaguar-architecture-the-cpu-powering-xbox-one-playstation-4-kabini-temash/3, (2013).

[3] Andreas Fredriksson, *Cold, Hard Cache – Insomniac Games' Cache Simulator*, Game Developers Conference, (2017).

[4] B.C. Ooi, K. J. McDonnel, R. Sacks-Davis., *Spatial kd-tree: An indexing mechanism for spatial databases*, Proc. 11th International Conference on Computer Software and Applications, (1987).

[5] D. Mehta, S. Sahni, *Handbook of Data Structures and Applications*, (2004).

[6] David H. Eberly, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, (2015).

[7] David Levinthal, *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*, Intel, (2009).

[8] Donald Knuth, *The Art of Computer Programming: Fundamental Algorithms, Third Edition*, (1997).

[9] Guy Jacobson, *Space-Efficient Static Trees and Graphs*, IEEE Symposium on Foundations of Computer Science, (1989).

[10] J.A. Storer, *An Introduction to Data Structures and Algorithms*, (2001).

[11] Jeremy Jones, *5 Caches*, https://www.scss.tcd.ie/jeremy.jones/CS3021/5 caches.pdf, (2016).

[12] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, Konrad Lai, *Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching*, ICS '02 Proceedings of the 16th international conference on Supercomputing, (2002).

[13] John von Neumann, *The First Draft Report on the EDVAC*, University of Pennsylvania, (1945).

[14] Paul E. Black, *perfect k-ary tree*, https://xlinux.nist.gov/dads/HTML/perfectKaryTree.html, (2017).

[15] R. Saikkonen, E. Soisalon-Soininen, *Cache-sensitive Memory Layout for Binary Trees*, (2008).

[16] Riku Saikkonen, *Online Bulk Deletion*, (2007).

[17] Riku Saikkonen, *Bulk Updates and Cache Sensitivity in Search Trees*, väitöskirja, Teknillinen korkeakoulu, (2009).

[18] Riku Saikkonen, Eljas Soisalon-Soininen, *Cache-Sensitive Memory Layout for Dynamic Binary Trees*, (2016).

[19] Smith C.U. and Williams L.G, *Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software*, Boston, MA, Addision Wesley, (2002).

[20] Vesa Hirvisalo, *Using Static Program Analysis to Compile Fast Cache Simulators*, väitöskirja, Teknillinen korkeakoulu, (2004).

[21] Wang, David Ng, Spencer Jacob, Bruce, *Memory Systems*, Morgan Kaufmann, (2010).

[22] Xian Gan, *Software Performace Test*, seminaarityö, Helsingin Yliopisto, (2006).