TEKNILLINEN KORKEAKOULU
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan tutkinto-ohjelma

PYTHONIN STANDARDIKIRJASTO

Katsaus tietorakennetoteutuksiin

Kandidaatintyö

Axel Eirola

Tietotekniikan laitos
Espoo 2009

TEKNILLINEN KORKEAKOULU
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan tutkinto-ohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä: Axel Eirola

Työn nimi:
Pythonin standardikirjasto – Katsaus tietorakennetoteutuksiin

Päiväys: 25. huhtikuuta 2009 **Sivumäärä:** 3 + 28
Pääaine: Ohjelmistotekniikka **Koodi:** T-106
Vastuuopettaja: Professori Lauri Savioja
Työn ohjaaja: Dosentti Ari Korhonen

Tässä kandidaatintyössä tarkastellaan ohjelmointikielen Pythonin sisäänrakennettuja tietorakenteita, keskittyen näiden toteutusten suorituskykyyn. Tietorakenteiden toteutuksista pyrittiin selvittämään suurimmat eroavaisuudet muiden ohjelmointikielten ja kirjastojen toteutuksiin, sekä miten nämä eroavaisuudet vaikuttavat ohjelmoijan työtä.

Suurin osa työstä keskittyy kokeellisten mittausten antamiin tuloksiin, niiden analysointiin, sekä näiden ympärille syntyvään keskusteluun. Samalla käsitellään niitä valintoja ja kompromisseja Pythonin kehittäjät ovat toteutuksessa tehneet, ja miten nämä yhdistyvät heidän kehitysfilosofiassa.

Avainsanat: Python, tietorakenteet, kokeellinen algoritmien analyysi
Kieli: Suomi

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Degree Program of Computer Science and Engineering

ABSTRACT OF BACHELOR'S THESIS

Author: Axel Eirola

Title of thesis:
Python standard library – A data structure overview

Date: April 25, 2009 **Pages:** 3 + 28
Professorship: Software Technology **Code:** T-106
Supervisor: Professor Lauri Savioja
Instructor: Docent Ari Korhonen

This bachelor's thesis inspects the built-in datastructures in the Python programming language, focusing on their performance. The most signifigant diffirences between these datastructures, compared to similar found in other programming languages and library implementations, are studied as well as how these differences affect the developer.

The main part of the work focuses on results from experimental measurements, the analysis of these, and the surrounding discussion. At the same time it touches the choices and compromises made by the developers of Python, and how these combine in their design philosophy.

Keywords: Python, data structures, experimental algorithm analysis
Language: Finnish

Sisältö

Luku 1

Johdanto

Tietotekniikan viimevuotisten kehitysten seuraksena suuri osa ohjelmistotuotannosta on siirtynyt käyttämään korkeamman abstraktiotason omaavia ohjelmointikieliä, kuten Java [6] ja Ruby [10]. Nämä kielet usein nopeuttavat ja yksinkertaistavat ohjelmointityötä ja tuotekehitystä, yleensä suorituskyvyn ja tehokkuuden kustannuksella. Mahdollisia vaikuttajia edistämässä tätä muutosta ovat ohjelmoijien vaatimusten ja kysynnän myötä kasvanut palkkataso, sekä teknisten uudistuksen mukana tulleet suorituskyvyn korotukset, yhdistyneenä hintatason laskuun. Yhdessä nämä mahdollistavat suotuisan kompromissin; lyhempiä kehitysaikoja, ja halvempia tuotantokustannuksia.

Python [15] on dynaamisena ja modulaarisena yksi näistä ns. uuden sukupolven korkean tason ohjelmointikielistä. Se tarjoaa käyttäjilleen samalla korkean abstraktiotason ohessa, myös todella laajan standardikirjaston [16], joka sisältää suurimman osan nykyohjelmoinnissa tarvittavista ominaisuuksista kuten tietokantakyselyitä, kryptografiaa, käyttöliittymiä, merkkijonojen käsittelyä ja paljon muuta. Tämän lisäksi Pythonin suosio perustuu myös automaattiseen muistinkäsittelyyn, dynaamiseen tyypitykseen ja sulavaan luettavuuteen.

Näistä syistä kieltä on hyvin usein kehuttu yksinkertaisuudestaan ja helppokäyttöisyydestään, ja on jopa syrjäyttänyt monessa koulussa päämääräisenä opetuskielenä ennen olleen Javan. Monikäyttöisenä kieli on käytössä myös monella eri teollisuuden alalla, mm. web-kehityksessä, tietokantaratkaisuissa, numeerisessa laskennassa, käyttöliittymissä, elokuvatuotannossa, ja yleisenä skriptauskielenä yhdistämässä erillisiä järjestelmäkokonaisuuksia [3].

Vaikka Python ei ole nykystandardien mukaan hidas skriptauskieleksi, hyödyntäessä monessa tapauksessa valmiiksikäännettyjä C kirjastoja, on sitä vähintään yhtä usein moitittu hitaaksi, kuin kehuttu tehokkaaksi ja käytännölliseksi. Varsinkin verrattuna perinteisempiin käännettäviin alemman tason kieliin kuten FORT-

RAN [2] ja C [1].

Kielen monikäyttöisyys ja kaikkien käyttäjien vaatimuksien tyydyttäminen aiheuttaa vaikeuksia tasavertaisen tehokkaan toteutuksen luomiseen. Varsinkin Pythonissa, jonka kehitysfilosofian [13] mukaan tulee pyrkiä alemman tason tapahtumien piilottamisella sekä yksinkertaisuudella johdattaa käyttäjää ainoaan ilmiselvään tehokkaaseen toteutustapaan. Tämä tarkoittaa esimerkiksi että yksinkertaisessa listassa ei kysytä käyttäjältä monelleko elementille varataan muistia alustaessa, vaan välikääntäjä ja ohjelmistotulkki yrittää parhaansa mukaan tuottaa kompromissin tehokkaimman mahdollisen resurssikäytön saavuttamiseksi.

Vaikka ohjelmointikieli tarjoaa dynaamisen tyypityksen ja suuren standardikirjaston tuottamat mukavuudet, eikä aktiivisesti vaadi käyttäjältä erillistä huomiota näiden asioiden syvempiin toteutuksiin, on oleellista olla tietoinen toteutuksen tarkemmasta käyttäytymistä. Tämä siksi, jotta voitaisiin tarkemmin hyödyntää kielen ja kirjastojen vahvuuksia, sekä välttää heikkouksia. Esimerkiksi jos meillä on tietorakenne josta halutaan usein hakea suurinta elementtiä, voidaan joko käyttää normaalia taulukkoa jolloin haku kestäisi $O(n)$, tai kekoa jolloin haku suoriutuu logaritmisessa ajassa. Todella usein oikean tietorakenteen valitseminen riippuu sen lopullisesta käyttötarkoituksesta, eikä optimaalista tietorakennetta voida luoda tietämättä jotain itse ohjelman ulkopuolisesta ympäristöstä.

Tässä kandidaatintyössä tarkastellaan ensin lyhyesti luvussa 2 joitain työn selvälle ymmärtämiselle oleellisia taustatietoja, kuten Pythonin dynaamisia muuttujatyyppejä jotta tarkemmin ymmärrettäisiin miten ne vaikuttavat laajempien tietorakenteiden käyttöön ja implementaatioon. Tarkastellaan myös ns. *mutable* ja *immutable* tietotyyppien eroja ja vaikutuksia.

Tämän jälkeen siirrytään luvussa 3 työn painopisteeseen, standardikirjaston tietorakenteiden toteutuksiin. Katsaus painotetaan viiteen tärkeimpään valmiiseen tietorakenteeseen: `list`, `bisect`, `dictionary`, `set` ja `heapq`; tässä työssä ei siis oteta huomioon kolmannen osapuolen laatimia, muita tietorakenteita tarjoavia, Python moduuleja tai kirjastoja kuten esimerkiksi SciPy [8]. Mainittuja tietorakenteita lähestytään kolmesta suunnasta. Ensinnäkin tarkastellaan Pythonin referenssidokumenttien kautta mitä tietorakenteilta vaaditaan, ja minkälaisia rajoituksia niille on asetettu. Toiseksi, tutkimalla tietorakennetoteutuksien lähdekoodia pyritään selvittämään tarkemmin mitä tunnettuja algoritmeja ja alemman tason tietorakenteita on hyödynnetty implementaatiossa. Kolmanneksi, suoritetaan koetintestejä (en. *benchmark*) erikokoisille tietorakenteille jotta saataisiin kuva niiden suorituskyvystä.

Tässä työssä siis tarkastetaan varsin yksinkertaisia, ja omalla tavalla peruslaatuisia, tietorakenteita. Tämä johtuen siitä että Pythonin standardiin, kuten moneenkaan muuhun ylemmän tason kielen, ei ole päätetty lisätä monimutkaisia tie-

torakenteita kuten laajoja puurakenteita. Tämä siksi että yleinen toteutus olisi liian monimutkainen ja tehoton verrattuna aina käyttötarkoitukseen tarpeellisen omaavan erillisesti toteutettuun tietorakenteeseen, jotta sitä olisi suotavampaa käyttää.

Luvussa 4 verrataan eri lähestymistapojen tuloksia, ja pohditaan mistä eroavaisuudet syntyvät. Keskitytään myös mihin käyttötarkoituksiin mikäkin tietorakenne on tarkoitettu, ja miten ne soveltuvat niiltä vaadittuilta eri käyttötarkoituksilta.

Vaikka on olemassa monia eri implementaatiota Pythonin standardista eri käyttötarkoituksiin ja eri alustoille kuten Jython [4], IronPython [7], PyS60 [9], jne., niin keskitytään tässä työssä pelkästään alkuperäisen Pythonin kehittäjien luomaan ja laajimmin käytettyyn implementaatioon, CPythoniin. Rajaus on tärkeä siksi että C kielellä implementoitu CPython ei pysty nojaamaan toisen korkeamman tason kielen valmiiksi tarjoamiin tietorakenteisiin tai ominaisuuksiin. Jatkossa viitataan tässä tekstissä Pythonilla aina erityisesti CPython toteutukseen.

Työ rajoitetaan myös Python standardin viimeisimpään 2.x versioon 2.6.1, vaikka uudempi standardi 3.0 on olemassa. Uudempi 3.0 versio on niin uusi että suurin osa kirjallisuudesta ja levityksistä ei ole vielä ehtinyt päivittyä uusimpaan versioon. Sen lisäksi se ei tarjoa mitään merkittäviä uudistuksia tietorakenteisiin, ja siten ei myöskään niiden implementaatioihin, vaan keskittyy enimmäkseen siivoamaan kielen pieniä ärsykkeitä, ja edistämään yleistä luettavuutta.

Luku 2

Tausta

2.1 Tietotyyppien ominaisuuksia

Pythonin ollessa perimillään olio-ohjelmointikieli, sen kaikki muuttujat ja tietorakenteet ovat olion `Object` alityyppejä, joista nämä jakautuvat muutamaan yleiseen aliluokkaan ominaisuuksien perusteella. Ensinnäkin on olemassa "muuttumattomia" (en. *immutable*) oliot joiden arvoa ei pysty muuttamaan luomisen jälkeen. Tämä ei kuitenkaan tarkoita että muuttujan arvo on muuttumaton, sillä nämä oliot ovat yleensä 'säiliö' tyyppisiä jotka vain sisältävät viittauksen muutettavaan arvoon. Tyypillisiä olioita ovat numerot ja merkkijonot, mutta myös 'jäädytetyt' tietorakenteet joihin lisäyksiä ei voida tehdä. Nämä oliot ovat yleensä myös hajautettavia (en. *hashable*), eli sopeutuvat käytettäväksi tietorakenteisiin jotka perustuvat elementtien hajautusarvoihin, yleensä niiden tunnistamisen tai sijoituksen nopeuttamiseksi.

Toiseksi on olemassa "muutettavia" (en. *mutable*) oliot joihin kuuluu kaikki tietorakenteet kuten listat ja joukot. Näiden olioiden arvoja voidaan muuttaa esimerkiksi suurentamalla listaa, tai lisäämällä lisää olioita joukkoon. Tämän ominaisuuden vuoksi näitä olioita ei pystytä hajauttamaan (en. *hash*), eikä järjestää, jolloin niitä voidaan harvoin lisätä tietorakenteen elementiksi.

Lisäksi oliot voivat olla joko järjestettäviä, jolloin oliolle on olemassa funktio joka kertoo onko yksi instanssi pienempi, yhtä suuri, tai suurempi kuin toinen samaa tyyppiä oleva instanssi. Merkkijonot ja reaaliluvut ovat järjestettäviä, kun taas kompleksiluvut eivät ole. Tämä rajoittaa joihinkin tietorakenteihin lisättävien sallittuja elementtejä määrää entisestään.

Suurimpaan osaan tietorakenteista pystyy lisäämään rinnakkain mitä tahansa muuttumaton tyyppiä olevaa oliota, eli listassa on mahdollista olla sekä nume-

roita ja merkkijonoja, ja ‘jäädytettyjä’ (muuttumattomaksi muutettuja) listoja. Tämä asettaa, kuten tulemme näkemään, joitain lisävaatimuksia algoritmien tehokkaalle suoritukselle.

2.2 Operaatiot

Tietorakenteille suoritetaan yleensä, tyypistä riippuen, erilaisia operaatioita, joista tässä käydään läpi seuraavan luvun kannalta oleellisemmat. Luonnollisesti jokaiseen tietorakenteeseen tehdään alkioiden lisäyksiä, ja yleensä vastaavasti myös poistoja. Joillekin tietorakenteille, kuten tietyt keot, tämä riittää, kun taas toisille on todettu hyödylliseksi tarjota laajempi joukko operaatioita. Tietorakenteeseen talletetun tiedon käsiin pääsemiseksi voidaan joko suorittaa poiston yhteydessä palautetun muuttujan käyttäminen, suora nouto tiedossa olevasta indeksistä, tai tietorakenteeseen kohdistuneella haulla jolla haettu alkio löydetään.

Lisäksi halutaan usein suorittaa yksittäisiä elementtejä laajempia operaatioita, kuten tietorakenteen järjestäminen, joka palauttaa järjestetyn tietorakenteen jonkun funktion määrittelemän järjestyksen mukaan. Joukkoja kuvaaville tietorakenteille on usein myös tärkeää verrata kahta erillistä tietorakennetta, tällöin usein suoritetaan matematiikan joukko-opista tuttuja operaatioita kuten leikkauksen ja yhdisteen laskeminen.

Listoille lisäys ja poisto tarkoittavat yksinkertaisesti vain alkion tallentamista tai poistamista annetusta muistipaikasta. Lisättyjä alkioita voidaan hakea suoraan indeksoinnilla jos niiden paikka listassa on tiedossa, tai haulla jolloin lista käydään kokonaan läpi etsien kyseistä alkiota. Hakua vastaavalla tavalla voidaan myös listaa käymällä läpi laskea tietyn alkion esiintymien määrä. Listoja voidaan myös järjestää jolloin kaikki alkiot järjestetään siten että edellinen alkio on pienempi, ja seuraava suurempi.

Järjestetylle listalle voidaan suorittaa samat operaatiot kuten edellä mainitulle listalle, sillä erolla että lisäys on oltava oikealle paikalle. Tämä voidaan varmistaa puolitushaulla joka löytää oikean paikan annetulle alkiolle. Luonnollisesti tietyt listan operaatiot kuten järjestäminen, haku ja esiintymien haku voidaan suorittaa paljon tehokkaammin järjestetylle listalle.

Keolle määritellään vain kaksi operaatiota, molemmat itsestään selviä, alkion lisäys sekä pienimmän alkion poisto. Useampia operaatiota on yleensä käytettävissä, mutta nämä ei esiinny tässä työssä.

Sanakirjoille kaikki kolme operaatiota lisäys, poisto ja haku toimivat kuten listassa, sillä erolla että jokaisen yhteydessä on alkion lisäksi tiedettävä alkiota vastaava

avain jolla alkioon voidaan viitata sanakirjassa.

Joukoille on käytössä, yksinkertaisen lisäyksen ja poiston, lisäksi monta matematiikan joukko-opista tuttua operaatiota. Ensinnäkin voidaan tarkistaa kuuluuko annettu alkio tiettyyn joukkoon, samoin myös tarkistaa onko annettu joukko tietyn joukon osajoukko. Tämän lisäksi voidaan kahdesta joukosta luoda näiden joukkojen yhdiste sekä leikkaus. Kaikki nämä operaatiot käyttäytyvät joukko-opista tutuilla tavoilla.

Luku 3

Tietorakenteet

Ennen kuin käymme käsiksi itse tietorakenteisiin on tärkeätä muistaa että Python on tarkoitettu erittäin monikäyttöiseksi kieleksi, joten tietorakenteet ovat optimoitu laajalle käyttöalalle. Näin ollen ei voida olettaa että ne toimisivat täydellisellä tehokkuudella missään osa-alueessa, mutta että ne suoriutuvat tarpeeksi hyvin kaikissa tilanteissa missä niitä voidaan olettaa käytettävän.

Seuraavissa osioissa keskitytään ensisijaisesti operaatioiden ajankäyttöön ja tietorakenteen tilavaatimukseen tietorakenteen koon funktiona. Mutta tietyissä tilanteissa tarkastellaan myös itse algoritmia, ja alemmalla tasoilla tehtyjä kompromisseja. Esimerkiksi muistin käsittelyn järjestyksistä suorituskyvyn nostamiseksi tietyissä yleisissä käyttöskenaarioissa.

3.1 Menetelmät

Toteutuksien käytännön suorituskyvyn tarkastamiseksi suoritettiin joukko mittauksia yksinkertaisilla koetintesteillä (en. benchmark). Testit suoritettiin mittaamalla yksittäisten operaatioiden ajankäyttöä erikokoisille tietorakenteille. Operaatiot valittiin aina kyseisen tietorakenteen ominaisuuksien ja käyttötarkoitusten mukaisesti, jotta testit vastaisivat mahdollisen hyvin oikean maailman käyttökohteita.

Lisäyksissä lisättiin aina satunnaisesti yksi satunnaisarvoinen kokonaisluku, liukuluku, kompleksiluku tai merkkijono. Näin ollen tietorakenteet sisälsivät samanaikaisesti monta eri tietotyyppiä. Listoihin, ja kekoihin ei lisätty kompleksilukuja jotta vertailuoperaatioita voitaisiin suorittaa ongelmitta.

Poistossa poistettiin aina yksittäinen satunnainen alkio tietorakenteesta; kekoa

lukuun ottamatta, josta aina poistettiin pienin (ylin) alkio. Poiston jälkeen lisättiin välittömästi toinen alkio säilyttämään tietorakenteen koon vakiona seuraavia mittauksia varten. Haussa sekä indeksoinnissa käsiteltiin aina satunnaista, tietorakenteessa olevaa, alkiota. Joukkojen väliset operaatiot suoritettiin aina kahdelle samankokoiselle joukolle.

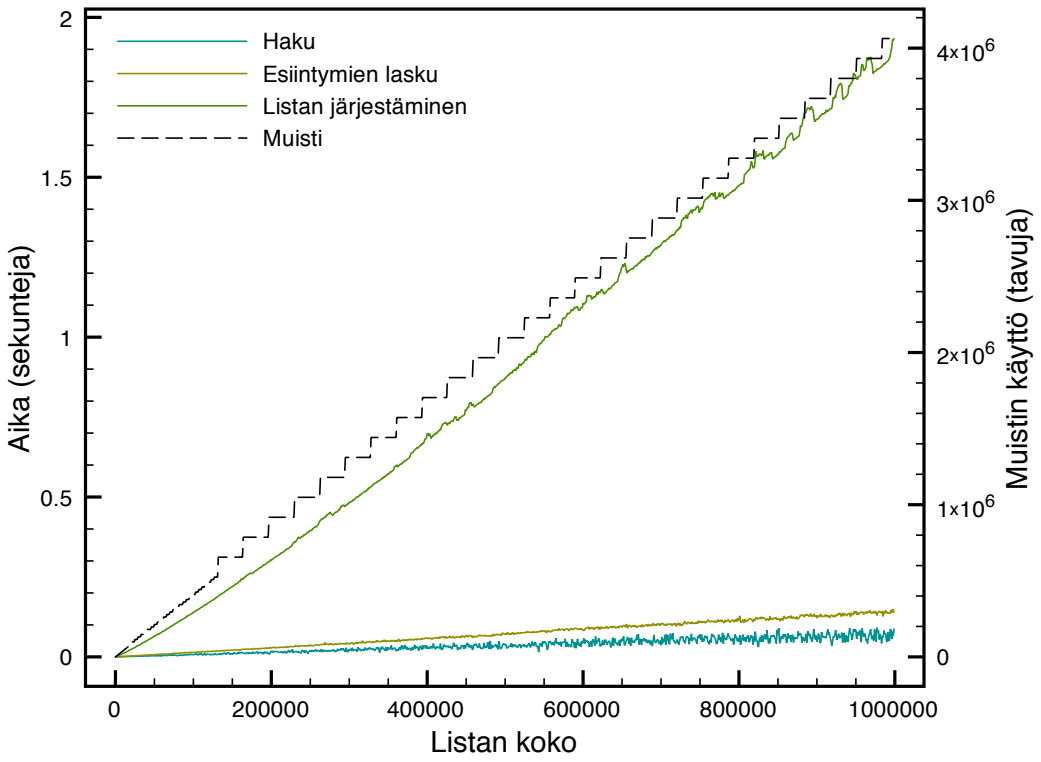
Ajoaikojen lisäksi mitattiin myös samanaikaisesti tietorakenteen koon muistissa. Tämä siis tarkoittaa itse tietorakenteen aiheuttaman ylimääräisen muistikäytön, ei sen sisältämien elementtien käyttämää muistia.

Nämä kaikki mittaukset suoritettiin yhdelle tietorakennetyypille aina samanaikaisesti samalla tietorakenteelle, tietorakenteen ollessa tietyn kokoinen. Ajan säästämiseksi ajoaikamittauksia tehtiin vain noin tuhannella eri tietorakennekoolla, kasvattaen kokoa satunnaisilla lisäyksillä mittausten välillä.

Mittaukset pyrittiin ajamaan mahdollisimman laajalle skaalalle tietorakenteiden kokoja, saadakseen näin mahdollisimman laajan kuvan suorituskyvystä. Aika- ja resurssirajoitusten myötä mittauksen rajoittuivat yleensä miljoonan alkion kokoisiin tietorakenteisiin. Jokainen koetintesti ajettiin kymmenen kertaa, ilman muisti- tai aikarajoituksia, joista keskiarvo on esitetty tässä luvussa.

Kaikki koetintestit[1] ajettiin Intel CoreDuo 1.83 GHz 2 Mt L2 välimuistia, 2 Gt koneella OS X 10.5.6 käyttöjärjestelmällä, Python versiolla 2.6.1. Itse ajanmittaus suoritettiin `timeit` [16, s. 925–926] moduulilla, ja muistinmittaus Heapy [11] kirjastolla (pienien ongelmien takia joukoille `__sizeof()__` metodilla). Regressioanalyysi ja kuvat luotiin Plot [17] ohjelmiston versiolla 0.997.

[1] Käytetty lähdekoodi on saatavissa osoitteesta http://users.tkk.fi/~aeirola/pythonBench09.tar.bz2



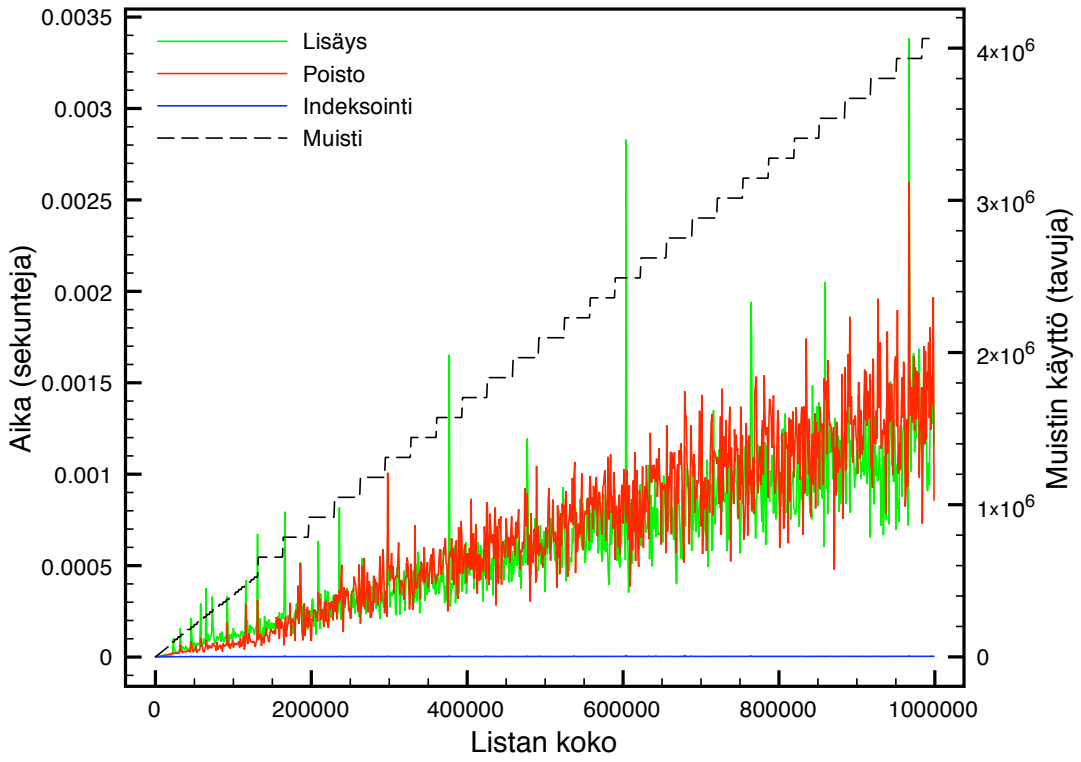
Kuva 3.1: Haun, järjestyksen ja esiintymien laskun aikavaatimus listan koon funktiona.

3.2 Tulokset

3.2.1 `list`

Listat ovat Pythonin sisäänrakennettu vastine perinteisten ohjelmointikielien taulukoille, sisältäen järjestetyn joukon olioita. Listat ovat Pythonissa, kuten taulukot monessa muussa ohjelmointikielenä, tärkeä osa myös muita ylemmällä abstraktiotasolla toimivia tietorakenteita. Esimerkiksi moni rakenne on itse asiassa toteutettuna listana, lisäten vain erillisiä operaatioita tietorakenteen käsittelyyn.

Kuten näemme kuvassa 3.1 listalle vartaan luomisen yhteydessä automaattisesti tietyn (pienen) määrän muistia, joka automaattisesti kasvaa listan täyttyessä. Muistia kasvatetaan suurimmille listoille suuremmissa lohkoissa, perustuen oletukseen että jos listaan on lisätty paljon elementtejä, niin siihen tullaan lisäämään vielä paljon enemmänkin. Muistialuetta kasvatetaan kuitenkin maltilla, verraten muistialueen tuplaamiseen. Näin ollen lista ei kovin herkästi varaa liikaa ylimääräistä muistia, mutta ei kuitenkaan tuhlaa turhaan aikaa liian tiheisiin



Kuva 3.2: Lisäyksen, poiston ja indeksoinnin ajankäyttö listan koon funktiona

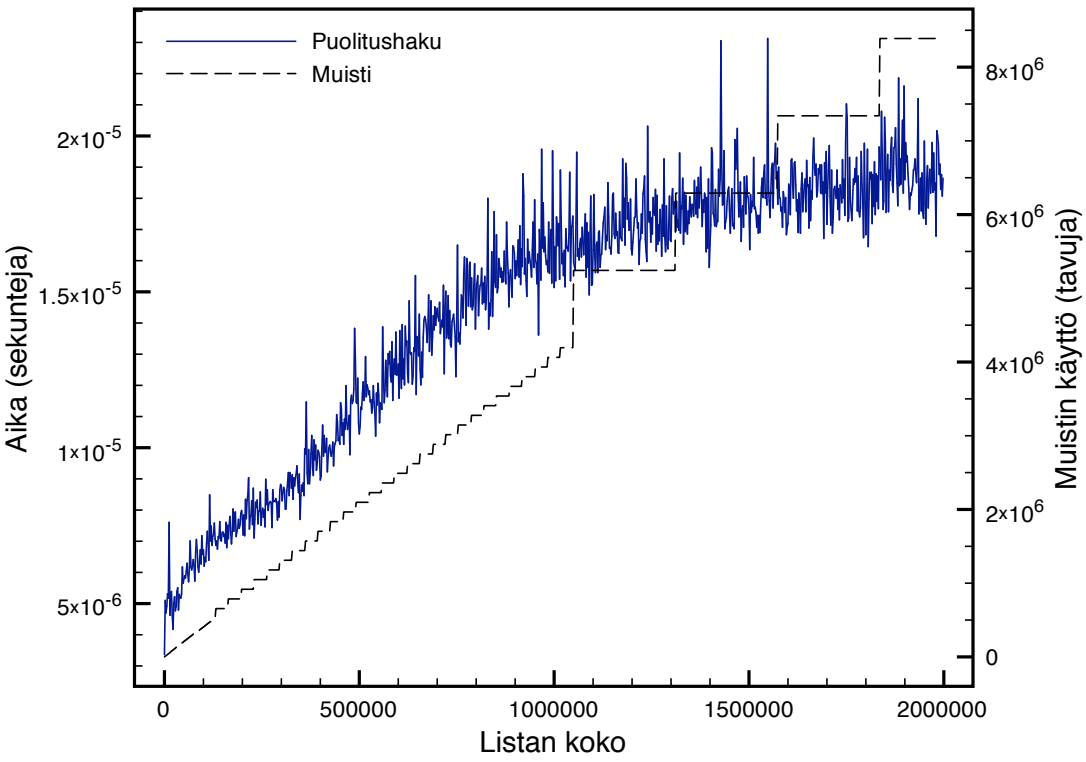
muistinvaraus-operaatioihin.

Koska alkeisoperaatioiden, lisäysten, poistojen ja noutojenon, ajoajat ovat kertaluokkaa nopeampia kun muut operaatiot, käsitellään ne erillisessä kuvassa 3.2. Huomataan kuitenkin miten alkioiden haku sekä lukumäärän lasku suoriutuu lineaarisessa ajassa. Oletuksella että nämä operaatiot ovat niin harvoin käytettyjä, niin erillistä indeksiä alkioiden lukumääristä tai sijainneista ei ylläpidetä.

Listan järjestäminen näyttäisi, nopealla katsauksella, suoriutuvan lineaarisessa ajassa, vaikka ei ole olemassa tunnettua yleistä vakioajassa toimivaa järjestysalgoritmia. Tarkastaen implementaation lähdekoodia [14, `/Objects/listsort.txt`] huomataan että järjestysalgoritmissa on käytetty 'adaptoituvaa, vakaata, ja luonnollista lomituslajittelua' (en. *mergesort*) [5, Luku 2], saavuttaen näin aikavaatimuksen $O(n \log n)$.

Tutkiessamme listan lisäys-, poisto- sekä indeksointi-operaatioita tarkastellaan kuvaa 3.2 joka antaa paremman näkymän tilanteesta. Huomataan heti että indeksointi listasta toteutuu vakioajassa, viitaten vahvasti taulukon olevan listan toteutuksen takana. Nähdään myös että lisäys (satunnaiseen indeksiin), ja samoin pois-

to vaatii lineaarisen ajan, viitaten myös siihen että lista on toteutettu taulukkona. Taulukkototeutus johtaa siihen että kaikki lisätyjenn, tai poistettujen, elementin jälkeen tulevat elementit on siirrettävä askeleen eteenpäin, tai taaksepäin. Tästä johtuen ajoajoissa esiintyy voimakasta värähtelyä kun siirrettävien alkioiden lukumäärä on verrannollinen satunnaisesti valittuun lisäys-/poistokohtaan. Tämän huomaa myös tarkistamalla lähdekoodia [14, `/Objects/listobject.c:220-250`]. Luonnollisesti lisäys listan loppuun, joka saattaa olla yleisin tapaus, suoriutuisi keskimäärin vakioajassa.



Kuva 3.3: Puolitushakualgoritmin aikavaatimus

3.2.2 `bisect`

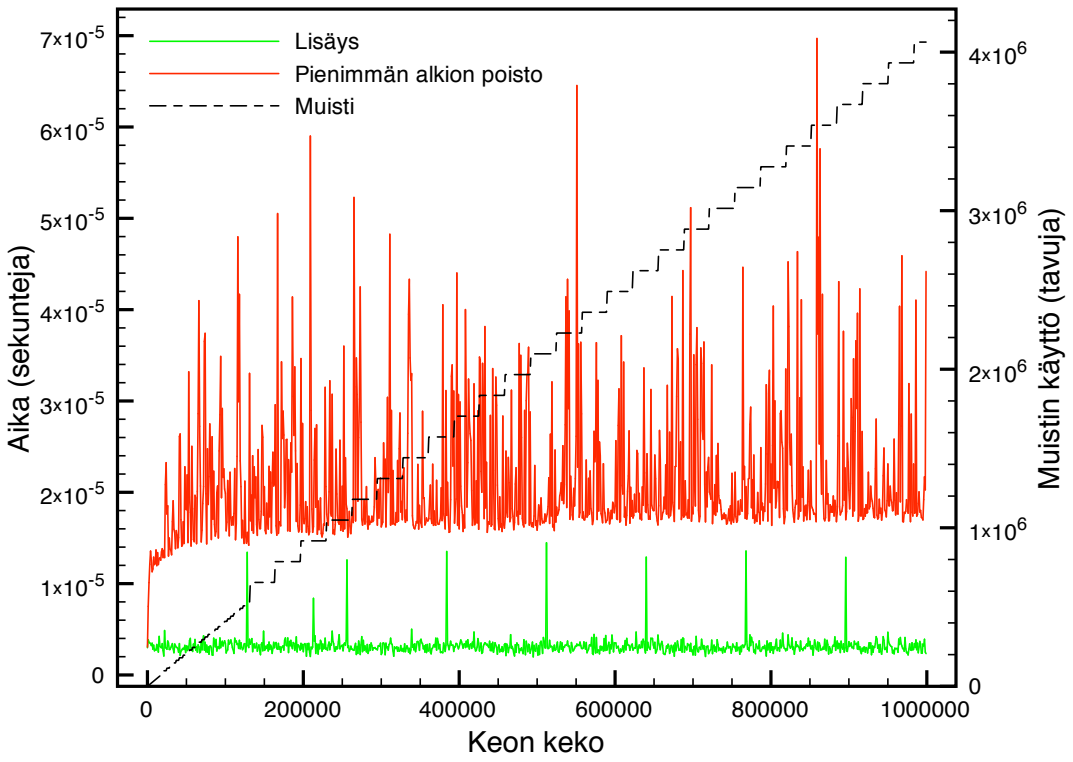
Vaikka ei suoranaisesti erillinen tietorakenne, niin edellisessä aliluvussa olleella listalla toimiva `bisect` tarjoaa järjestetyn listan, jossa jokainen lisättävä alkio sijoitetaan järjestyksen ylläpitävään kohtaan. Koska muut operaatiot, ja kuten huomataan myös muistinkäyttö, periytyvät suoraan listan ominaisuuksista (nähtävissä kuvassa 3.2) riittää tässä tarkastella vain lisäykseen liittyvän puolitushaun kustannusta. Järjestyksessä olevalle listalle voidaan kuitenkin suorittaa huomattavasti tehokkaampi puolitushaku verrattuna yleiseen listaan.

Kuten nimestä saattaa jo päätellä, hyödyntää lisäysalgoritmi varsin perinteistä puolitushakua [14, `/Lib/bisect.py:24-43`] löytääkseen listasta sopivan kohdan lisätä elementin. Huomattavaa on että tämä algoritmi on toteutettu puhtaalla Pythonilla, siis käyttämättä alemman tason C ohjelmointikieliä, joka teoriassa johtaa heikompaan suorituskykyyn.

Tarkastellessa kuvaa 3.3, nähdään miten puolitushaku suoriutuu järjestetyn listan kasvaessa. Lisäyshän käyttäytyy, kuten edellisessä kappaleessa todettiin, lineaarisesti listan pituuden kanssa, kun taas puolitushaku on selvästi tehokkaammassa

$O(\log n)$ luokassa. Yllättäen, kuvassa näin ei kuitenkaan vaikuttaisi käyvän ennen miljoonan alkion kokoa, vaan kasvu näyttää pienemmillä ko'oilla enemmänkin lineaariselta.

Vaikka puolitushaun aikavaatimus kasvaakin alussa hieman jyrkemmin kuin puhdas logaritminen, on listaan lisäyksen vaatima aika kuitenkin selvästi yli puolitushaun. Verraten kuvia 3.2 ja 3.3 huomataan itse lisäyksen vaikuttavan kokonaisaikaa huomattavasti enemmän yhteiseen lisäysoperaatioon kuin itse lisäyspaikan haku. Näin ollen puolitushakualgoritmin ylimääräinen optimointi ei antaisi huomattavaa kokonaishyötyä suorituskykyyn.



Kuva 3.4: Lisäyksen ja pienimmän alkion poiston ajoajat keossa

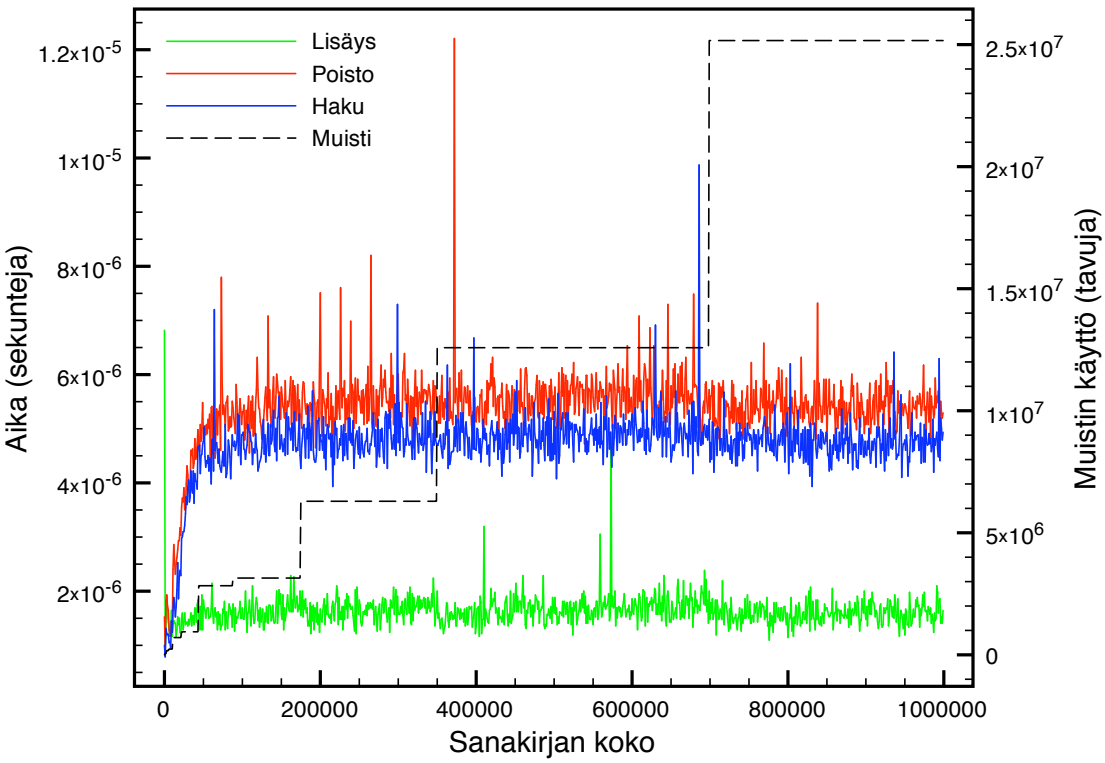
3.2.3 `heapq`

Jos halutaan tietorakennetta joka aina pidetään järjestyksessä, ja josta sen lisäksi aina vain haetaan pienin (tai suurin) alkio, niin tulisi käyttää prioriteettijonoa. Pythonissa näille on tarjoilla valmis keko-toteutus rakennettuna tavallisen taulukon, ei siis listan, päälle. Tämä tarkoittaa että keko ei ole sinänsä erillinen tietorakenne, mutta enemminkin algoritmi joka käsittelee taulukkoa, varsin vastaavasti kun yllä kuvattu puolitushakualgoritmi. Keko on, kuten myös edellisen osion tietorakenne, toteutettu suoraan Pythonilla, luottaen sen suorituskykyyn alemman tason ohjelmointikieltä käyttämättä.

Koska tietorakenne asettaa enemmän rajoitteita sen käytölle (aina järjestyksessä, vain pienimmän poisto), on luonnollista että sen pitäisi suoriutua nopeammin kuin vastaavat tietorakenteet ilman näitä rajoituksia, kuten listat. Harvinaisen yllätyksettömästi nähdään kuvassa 3.4 miten sekä lisäys että pienimmän alkion poisto suoriutuvat logaritmisessa ajassa, eli suoraan verrannollisesti keon syvyyteen.

Verraten siis yllä käsiteltyihin listoihin perustuviin tietorakenteisiin, suoriutuu

keko huomattavasti paremmin sille tarkoitetussa tehtävässään. Tästä nähdään miten yksinkertaisesti oikea tietorakenteen valinta vaikuttaa tehokkuuteen, paljon enemmän kuin toteutuksen optimointi.

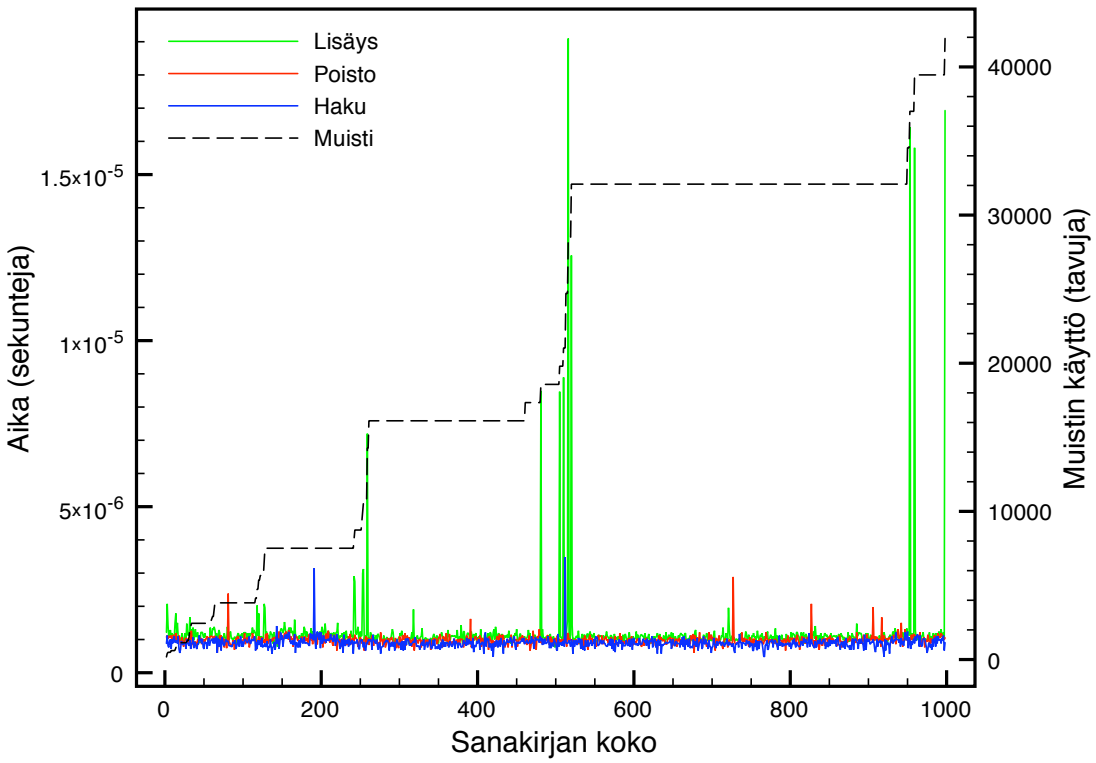


Kuva 3.5: Sanakirjan poiston, lisäyksen ja haun ajoaikoja

3.2.4 `dictionary`

Jos listojen järjestetyn tallennuksen, missä jokainen elementti luetteloidaan kasvavalla numerolla, sijaan halutaan pystyä hakemaan elementtejä erillisen avaimen perusteella on suositeltavaa käyttää hajautustauluja, tai sanakirjoja (en. *dictionary*), kuten Pythonissa on nimetty. Näissä tietorakenteissa elementit eivät ole minkäänlaisessa yleisesti havaittavissa olevassa järjestyksessä, vaan esiintyvät avain-arvo-pareina, jossa arvon saa haettua nopeasti tietämällä vastaavan avaimen.

Tarkastellessa tietorakenteen suorituskykyä kuvassa 3.5, huomataan ensinnäkin miten tasaisen logaritmisesti kaikkien operaatioiden suoritusajat kasvavat tietorakenteen koon kasvaessa yli sadantuhannen, ja tällöinkin pysyen mikrosekuntien luokissa, poiketen tästä ainoastaan muistialueen koon muutoksissa kuten näemme myöhemmin. Koska nämä sanakirjat esiintyvät myös usein pienempinä esiintyminä Pythonin yleisessä toteutuksessa (esim. oliot ja funktioparametrit [12, s. 296–298]) se on tietoisesti päätetty pitää yksinkertaisena ilman liiallisia optimisaatioita erikoistapauksille joiden lisääminen hidastaisi yleisten tapausten suori-

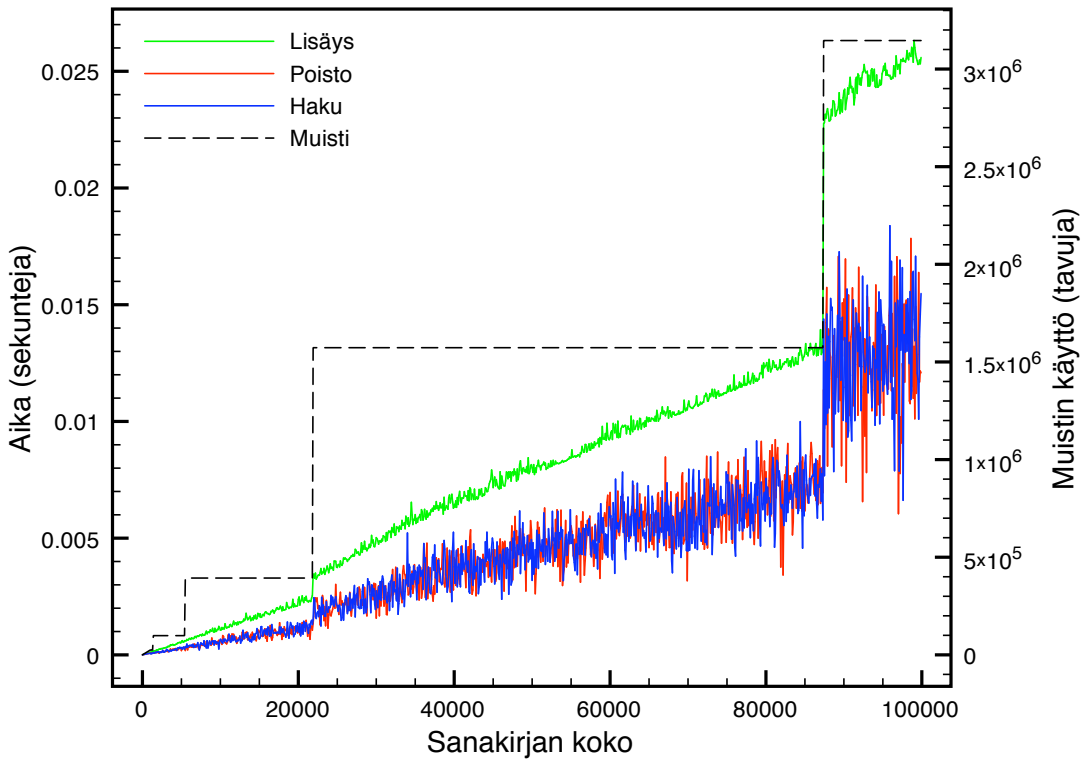


Kuva 3.6: Sanakirjan poiston, lisäyksen ja haun ajoaikoja pienemmällä asteikolla

tuskykyä, ja samoin koko Pythonin suorituskykyä.

Mielenkiintoista on myös miten Pythonin kehittäjät ovat päätyneet, vastoin yleisiä käytäntöjä , käyttämään epätasavälistä (en. *non-uniform*) hajautusfunktioita [18] (esim. kokonaislukujen hajautusarvo on kokonaisluku itse), tämä aiheuttaa helposti tiheitä joukkoja hajautustaulukossa, joissa voidaan tehokkaasti hyödyntää välimuistiosumia jatkuvien hakujen tilanteissa. Tämä kuitenkin asettaa tiukempia vaatimuksia hajautustörmäysten käsittelyyn, sillä niitä esiintyy luonnollisesti enemmän.

Kuvasta 3.5 nähdään myös miten muistinkäyttö kasvaa vastaavalla tavalla lineaarisesti kuin aiemmin käsitellyssä listassa, viitaten että jonkinlainen ennaltamäärätyn täyttöasteen kohdissa kasvava taulukko on tietorakenteen takana. Hajautustaulukon luonteen omaavana tulee taulukon koon muuttaminen helposti kalliiksi alkioiden hajautusarvojen taulukkoon nähden päivittämisen, ja sitä myötä niiden pakollinen siirtyminen. Seuraus tästä nähdään selvänä kuvassa 3.6, jossa lisäysten aikavaatimus kasvaa moninkertaisesti kohdissa (kuvassa selvimmin keskikohdassa, sanakirjan koolla 500) joissa muistia kasvatetaan.



Kuva 3.7: Sanakirjan poiston, lisäyksen ja noudon ajoaikoja vakioarvoisella hajautusfunktiolla

Muistinkäytön pienempi porrastus juuri ennen isoa kasvua ei ole toteutuksen ominaisuus, vaan monen ajon keskiarvon tuottama seuraus. Koska hajautustaulusta poistetut elementit merkitään pelkästään poistetuiksi, jotta hajautusketjua ei jouduttaisi rakentamaan uusiksi, kasvaa taulukon täyttöaste vaihtelevasti riippuen montako poistetun elementin paikkaa voidaan uudelleenkäyttää. Tämän takia muistin uudelleenallokointi ei aina tapahdu samalla tietorakenteen koolla.

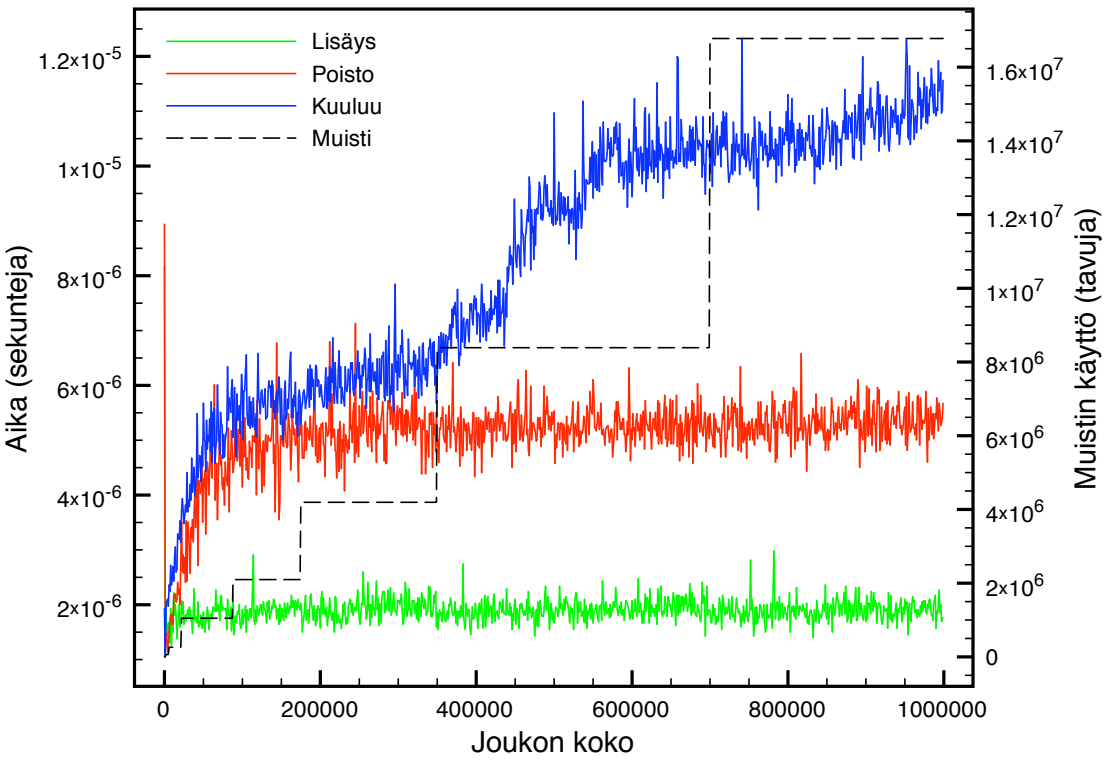
Tässä siis ilmenee iso heikkous Pythonin yksinkertaistusfilosofiassa, sillä mitään ilmeistä tapaa luoda näitä tietorakenteita käyttäjän määrittelemällä aloituskoolla. Tämä johtaa siihen että ei voida hyödyntää mahdollisesti käytettävissä olevaa tietoa tietorakenteen tulevasta käytöstä. Esimerkiksi jos halutaan rakentaa Suomen laajuisen henkilörekisterin josta voidaan sosiaaliturvatunnuksella hakea henkilötietoja, tiedettäisiin että sopiva aloituskoko olisi noin viisi miljoonaa elementtiä, mutta koska tätä tietoa ei pystytä käyttämään, tulee tietorakenteen luomiseen kulutettu aika kasvaa turhien uudelleenjärjestyksien myötä.

Toteutuksen rakentuessa hajautustaulun periaatteen päälle, nojaa se raskaasti

avain-elementtien hajautusarvoihin. Tästä seuraa että jos avaimen oliolle tai tietotyypille ei ole määritelty tarpeeksi hyvä hajautusfunktio, niin tulee väistämättä sattumaan hajautustörmäyksiä (en. *hash collision*), joiden seurauksena suorituskyky heikkenee.

Tarkastellakseen miten Pythonin sanakirja pärjää tässä tilanteessa luotiin keinotekoinen rakennelma missä tietorakenteeseen lisätään yksinkertaisia olioita joiden hajautusfunktio palauttaa aina vakioarvon 0. Testin tulokset näkyvät kuvassa 3.7. Kuten huomataan, jää suorituskyky heti selvästi heikommaksi kuin normaaleilla oliolla. Tämä heikkous ei kuitenkaan ole mitenkään Pythonille ominainen, vaan perustavanlaatuinen ominaisuus hajautusmenetelmissä. Huolestuttavaa sen sijaan on miten nopeasti operaatioiden suoritusajat kasvavat, sillä teoriassa hajautustaulu degeneroituu yksinkertaiseksi listaksi vakiohajautusarvolla, kun taas Pythonissa kasvu näyttäisi oleva huomattavasti suurempi, varsinkin hajautustaulun laajenemisen jälkeen.

Nämä isot loikkaukset ajankäytössä johtuvat aiemmin mainitusta välimuistiosumista, pienillä koilla suurempi osa aina läpikäytävästä listasta mahtuu välimuistiin, mutta kun taulukon koko kasvaa, ei välimuistiin enää mahdu yhtä suuri osa listasta. Näin ollen suorituskyky kärsii selvästi enemmän suuremmilla taulukoilla jotka eivät enää mahdu testialustan prosessorin 2 Mt välimuistiin.



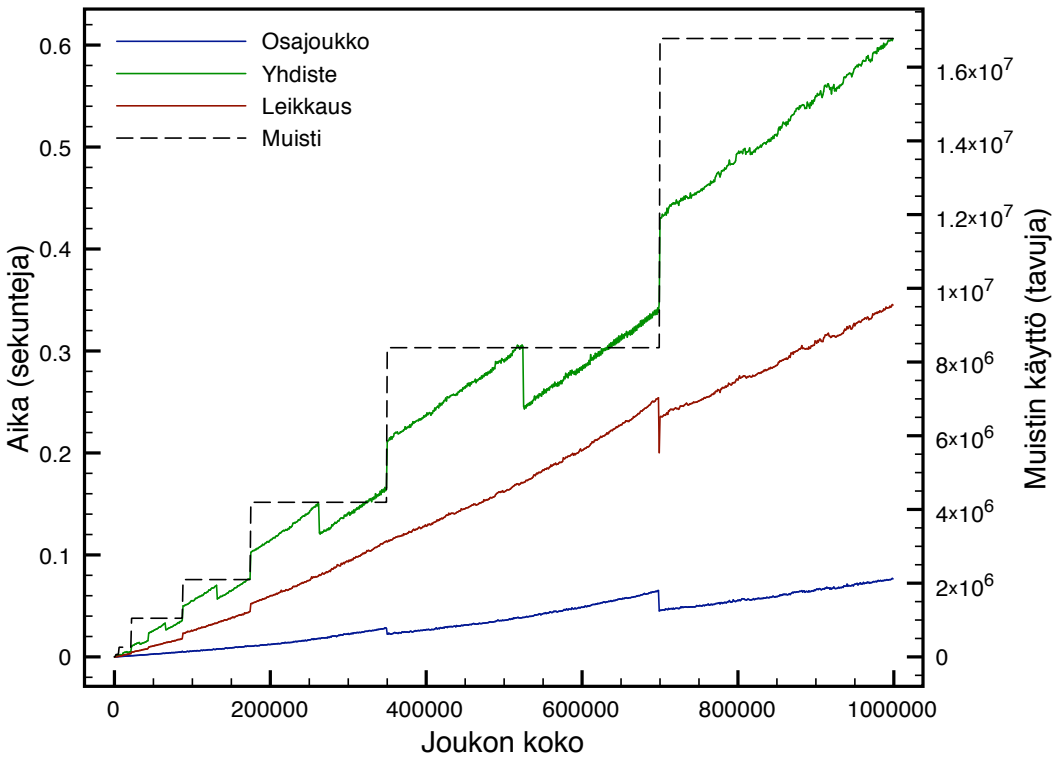
Kuva 3.8: Joukon alkeisoperaatioiden ajankäyttö koon funktiona

3.2.5 `set`

Jos halutaan mallintaa elementtejä yleisenä joukkona, ilman erityistä tietoa elementtien välisestä järjestyksestä tai avaimista joilla joukon elementtejä voitaisiin hakea, niin Python tarjoaa tähänkin erityisesti tarkoitetun tietorakenteen. Jällen kerran lähdekoodia tutkimalla [14, `/Object/setobject.c:4`] huomataan miten Pythonin joukot (en. *set*) toimivat tietyllä tavalla kuten sanakirjat joista on karsittu pois arvot, tallentaen sen sijaan elementit pelkästään avaimina.

Joukoille on ominaista, yksittäisten elementtien manipuloinnin lisäksi, suorittaa joukkojen välisiä vertailuja. Tämän takia jaetaan tarkastelu kahteen osaan; alkeisoperaatioihin, jotka kuvaavat yksittäisten elementtien käsittelyä, sekä yhdisteoperaatioihin, jotka kuvaavat joukoille kokonaisuudessa suoritettavia operaatioita.

Verraten joukon alkeisoperaatioita kuvassa 3.8 sanakirjan vastaaviin, nähdään miten yhteinen, hajautukseen perustuva, toteutusidea antaa vastaavanlaiset logaritmiset aikavaatimukset; jopa kertoimet näyttäisivät olevan identtiset. Ainoa merkittävästi eroava ominaisuus on elementin haun, tai kuten joukoille nimettynä ‘kuuluu’ ($\in$), operaation aikavaatimuksen yllättävä kasvu hajautustaulukon

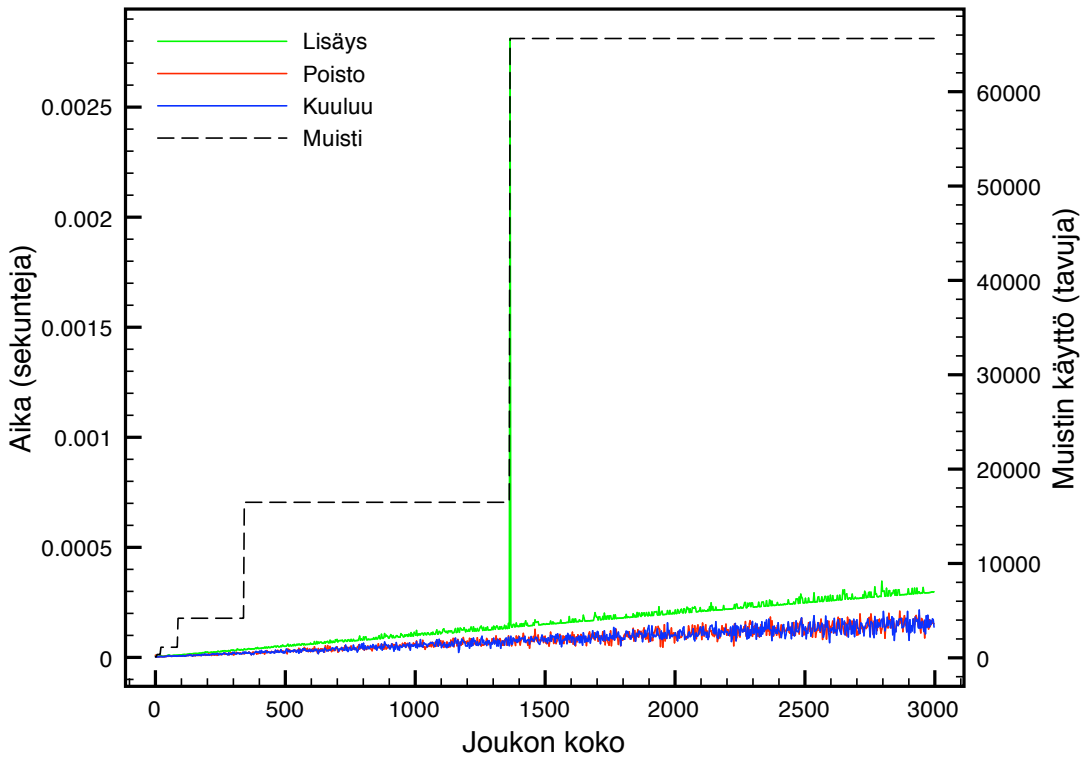


Kuva 3.9: Joukon yleisoperaatioiden ajankäyttö koon funktiona

kasvaessa. Kuuluu operaation aikavaatimus käyttäytyy pienemmillä tietorakenteilla samalla tavalla logaritmisesti kuin poisto operaatio, mutta alkaa suuremmilla ko'oilla vaatimaan enemmän aikaa kuin poisto operaatio tai vastaava hakuoperaatio sanakirjoille. Tälle kasvulle ei löytynyt lähdekoodistakaan selkeätä selitystä.

Joukkojen toteutus ei kuitenkaan ole täysin identtinen sanakirjojen toteutuksen kanssa. Tarkastellessa muistikäyttöä kuvassa 3.8, huomataan että se eroaa sanakirjojen dynaamisesta muistiallokoinnista kuvassa 3.5. Tämä viittaisi siihen että hajautustaulukon uudelleentoteutuksessa ollaan vähintäänkin otettu huomioon joukkojen ja sanakirjojen eroavat käyttötilanteet, soveltaen ja optimoiden toteutusta tarjotakseen parhainta mahdollista suorituskykyä yleiselle joukolle.

Suorittaessamme vastaavanlaisen katsauksen joukon yleisoperaattoreille kuvasta 3.9, emme voi tehdä samoja vertauksia sanakirjan suorituskykyyn, vastaavanlaisien operaattoreiden puuttuessa siitä. Sen sijaan voimme verrata alkeisoperaattoreiden logaritmiseen aikavaativuuteen ja operaatioiden luonteeseen. Sillä kaikki kolme yleisoperaatiota käytännössä vertaavat jokaisen joukon elementin jokaiseen

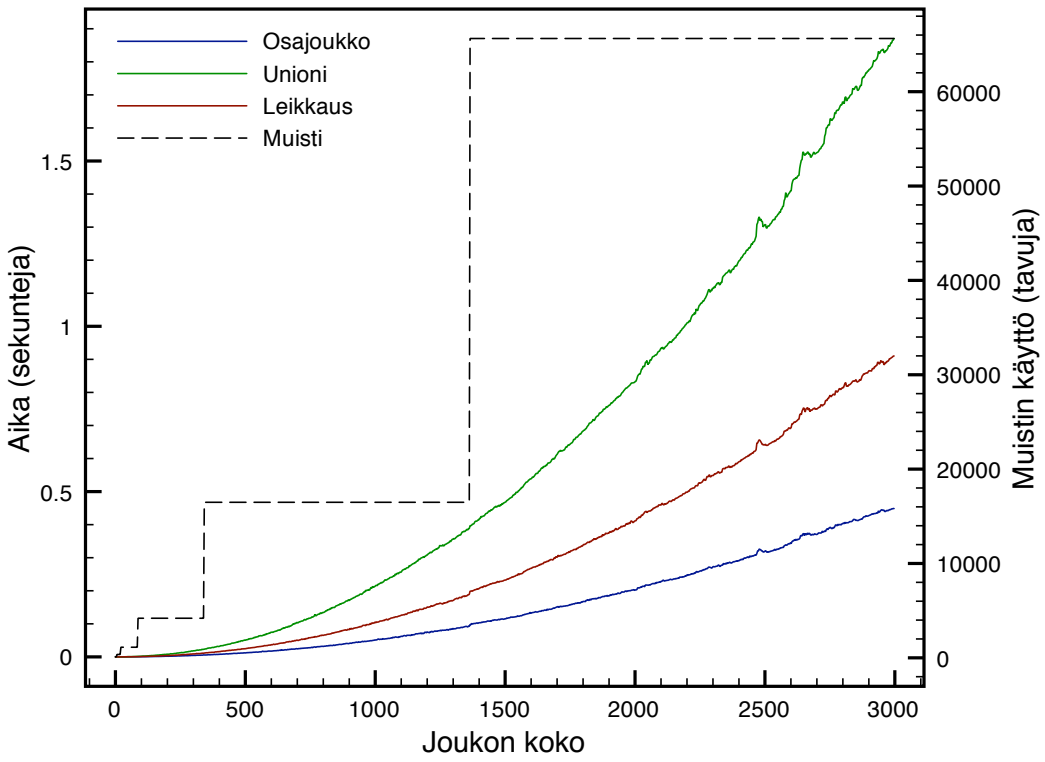


Kuva 3.10: Vakiohajautusarvojoukon alkeisoperaatioiden ajankäyttö koon funktiona

toisen joukon elementtiin, esimerkiksi leikkauksessa tarkastetaan jokaiselle alkiolle kuuluuko se jo toiseen joukkoon jotta tiedetään kuuluuko se lisätä leikkaukseen vai ei. Näin ollen voidaan arvioida aikavaatimuksen olevan $O(n \log n)$, joka yhtenee kuvan yleisen piirteiden kanssa.

Selittämättömäksi jää kuitenkin radikaalit suoritusajan loikkaukset hajautustaulun uudelleen rakentamisen myötä muistin kasvupisteissä, sekä yhdisteelle toiset loikkaukset näiden välissä. Yksi selitys yhdisteen käyttäytymiselle löytyy sen toteutuksesta, jossa ensimmäisen joukon kopion uudelleenrakentaminen suuremmaksi päätetään joukkojen koiden perusteella. Jos joukot ovat tarpeeksi pieniä ne mahdutetaan samaan ahtaampaan taulukkoon jolloin hajautustörmäyksiä tapahtuu useammin. Tällöin olisi mahdollista että muidenkin operaatioiden käyttäytymiset selittyisi vastaavasti.

Koska joukot perustuvat, kuten sanakirjatkin, sinne lisättyjen elementtien hajautusfunktioiden toimivuuteen, on ilmeistä että suorituskyky heikkenee samankaltaisesti huonon hajautusfunktion myötä. Vakioarvoinen hajautusfunktion omaavien elementtien lisäys joukkoon aiheuttaa jokaiselle operaatiolle hajautustör-



Kuva 3.11: Vakiohajautusarvojoukon yleisoperaatioiden ajankäyttö koon funktiona

mäyksen jokaisen joukossa olevan elementin kanssa.

Kuvassa 3.10 nähdään miten tämä hajautustörmäysten ketju vaikuttaa alkeisoperaatioiden surituskykyyn. Huomataan miten tasaisesti lisäys, poisto ja elementin haku joukosta kasvavat suoraan joukon koon mukaan. Tämä johtu siitä että hajautustaulukko degeneroituu listaksi, hajautustörmäysten aiheuttamaksi ketjuksi, jonka läpi on iteroitava kunnes sopiva paikka löytyy. Yksinäinen piikki muistinkorotuksen yhteydessä näyttää hajautustaulun raskaan uudelleenluomisen, varsinkin monilla hajautustörmäyksillä. Tämä piikki ei näy muilla kasvuilla sillä mittaukset suoritettiin, aikaa säästääkseen, vain joka kolmannen lisäyksen kohdalla.

Koska jokainen alkeisoperaattori vaatii lineaarisen ajan vakiohajautusarvolla, ei ole yllättävää että joukkojen yhdisteoperaattorit vaativat neliöllisen ajan samoilla joukoilla. Kuten nähdään kuvassa 3.11 tämä toteutuu varsin tarkasti, sillä jokainen elementti ensimmäisessä joukossa verrataan, tavalla tai toisella, toisen joukon elementtiin sillä koko lista on käytävä aina läpi.

Tuloksista huomataan miten Pythonin joukkojen tietorakenne eroaa olennaisesti sitä tunnetummasta union-find rakenteesta [5, Luku 21 (nimellä *disjoint-set*)]. Kun union-find rakenne suoriutuu haku- sekä yhdisteoperaatioissa huomattavasti tehokkaammin $O(\log n)$ ajassa, on tässä kappaleessa käsitellyt joukot huomattavasti tasapainoisemmin optimoitu, mahdollistaen kaikkien operaatioiden suorituksen mielekkäässä ajassa. Tästä koituu kuitenkin helposti tiettyjä ongelmatilanteita, esimerkiksi Kruskalin algoritmissa [5, Osio 23.2] nojataan tehokkaisiin yhdisteoperaatioihin jolloin se toimii tehokkaasti union-find rakenteella, mutta hidastuu huomattavasti Pythonin joukkojen selvästi lineaarisella yhdisteoperaatiolla.

Luku 4

Yhteenveto

Yhdistäen edellisessä luvussa käsiteltyjä tuloksia, huomataan miten lähellä toteutus on pysynyt johdannossa mainittua kehitysfilosofiaa [13]. Ensinnäkin, tietorakenteet ovat suunniteltu yksinkertaisiksi, keskittyen yleisen käyttötarkoituksen toimivuuteen, turhaan monimutkaistamatta yleistä tapausta harvinaisten erikoistapausten vuoksi. Näin ollen valtaosa käytöstä toimii huoletta, ja erikoistapauksetkin tarpeeksi hyvin, vaikka erikoisia ongelmatapauksia onkin aina olemassa. Toisaalta, on tietorakenteita kuitenkin toteutettu käytännöllisyyttä silmällä pitäen. Painottaen suorituskykyä oikean maailman ongelmien tehokkaaseen ratkontaa, jättäen liiallisen 'puhtauden' ja pelkistämisen pois. Kolmanneksi, kaikilla tietorakenteilla on selkeät, toisistaan hyvin rajatut, käyttötarkoitukset. Tämän takia on yleensä selkeää nähdä mikä tietorakenne soveltuu parhaiten tietyn edessä olevan ongelman juuri oikeaan toteutukseen.

Tästä huolimatta toteutukset eivät ole täysin puutteettomia; tekemällä toteutuksesta helppokäyttöisemmän, karsiintuu siitä väistämättä jotain, tosin ei välttämät kovin tärkeitä, ominaisuuksia. Esimerkkinä, osiossa 3.2.4, esillä olleen tietorakenteen aloituskoon määrittelyn puute. Tämä estää arvokkaan ulkopuolisen informaation lisäämisen ohjelmaan, joka puolestaan johtaa heikompaan suorituskykyyn.

Lisäksi, osiossa 3.2.5, törmättiin Pythonin joukkojen toteutuksen aikavaatimusten käyttäytyvän oleellisesti erilaisesti yleisemmin joukko-rakenteena toimivan union-find rakenteeseen verrattuna. Tämä eroavaisuus voisi tuottaa merkittäviä tehokkuushäviöitä jos toteutusta hyödyntäessä oletetaan tiettyjä ominaispiirteitä tuntematta itse toteutusta tarkemmin.

Pienistä epätäydellisyyksistä huolimatta, suurin osa käyttäjien tarpeista täyttyvät tarpeeksi hyvin, tuottamatta ylimääräistä päänvaivaa. Ja pienten ongelmien kasvaessa joissain käyttökohteissa suuremmiksi ongelmiksi, on aina mahdollista,

yksinkertaisempia elementtejä käyttäen, tai uudelleen määrittelemällä, rakentaa juuri tiettyyn käyttökohteeseen täydellisen toteutuksen tietorakenteesta.

Nämä kaikki ominaisuudet tekevätkin Pythonista niin tehokkaan ja soveltuvan monen eri käyttöalojen tarpeisiin, kun mihin se on jo lyhyen vajaan kahdenkymmenen vuoden historian aikana asettautunut. Näin ollen tarjoten käyttäjilleen tasapainoisen ja yleisesti toimivan ohjelmointikielen, sekä tietorakenteidensa että yleisen kieliasun puolesta.

Kirjallisuutta

[1] *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, joulukuu 1999.

[2] *Draft International Standard ISO/IEC 1539-1:2004(E): Information technology — Programming languages — Fortran Part 1: Base Language*. International Organization for Standardization, toukokuu 2004.

[3] Python success stories, 2009. URL http://www.python.org/about/success/.

[4] J. Baker et al. The Jython Project. URL http://jython.org.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest ja Clifford Stein. *Introduction to algorithms, Second Edition*. MIT press, 2001.

[6] James Gosling. *The Java Language Specification*. Addison-Wesley Professional, 2000.

[7] J. Hugunin. IronPython: A fast Python implementation for .NET and Mono. *PyCON 2004 International Python Conference*, 2004.

[8] E. Jones, T. Oliphant, P. Peterson et al. SciPy: Open source scientific tools for Python. URL http://www.scipy.org.

[9] J. Laurila, M. Marchetti ja E. Smartt. Python for S60–PyS60, Nokia. URL http://www.pys60.org/.

[10] Yukihiro Matsumoto. *Ruby Programming Language*. Addison Wesley Publishing Company, 2002.

[11] Sverker Nilsson. Heapy: A Memory Profiler and Debugger for Python. 2006.

[12] Andy Oram ja Greg Wilson. *Beautiful Code*. O'Reilly Media, Inc., 2007.

[13] Tim Peters. The Zen of Python. URL http://www.python.org/dev/peps/pep-0020/. Saatavilla kirjoittamalla Python tulkkiin `import this`.

[14] Python Software Foundation. CPython 2.6.1 source tarball, 2009. URL http://www.python.org/ftp/python/2.6.1/Python-2.6.1.tar.bz2.

[15] Guido van Rossum ja Fred L. Drake, Jr. *Python Language Reference Manual*, helmikuu 2009. URL http://docs.python.org/2.6/reference/.

[16] Guido van Rossum ja Fred L. Drake, Jr. *Python library reference*, helmikuu 2009. URL http://docs.python.org/2.6/library/.

[17] Michael Wesemann. Plot, a scientific 2D plotting program for Mac OS X, 2007. URL http://plot.micw.eu/.

[18] Andrew C. Yao. Uniform hashing is optimal. *Journal of the ACM (JACM)*, 32(3):687–693, 1985.