Tietojenkäsittelytiede **29** Lokakuu 2009 sivut 49–70
Toimittaja: Jorma Tarhio

Käytännön kokemus algoritmikirjastojen autuudesta

Antti Valmari
Tampereen teknillinen yliopisto
Ohjelmistotekniikan laitos
Antti.Valmari@tut.fi

Tiivistelmä

Perustietorakenteiden ja -algoritmien mukaantulo ohjelmointikieliin on tehnyt helpoksi saada ne käyttöön ohjelmoimatta niitä itse. On luonnollista kysyä, tarvitseeko niiden toimintaa enää opettaa kuten aikaisemmin. Eikö riitä, että opetetaan niiden tuottamat palvelut sekä miten ne saa käyttöön ja mitä ne maksavat? Tässä kirjoituksessa esitellään ohjelmointikielen C++ mukana tulevaa tietorakenne- ja algoritmikirjastoa. Esimerkkien avulla perustellaan, että sen turvallinen ja tehokas käyttö on hyvin vaikeaa, ellei käyttäjä tunne palveluiden takana olevaa toteutusta edes pääpiirteissään. Kirjoitukseen sisältyy kaunistelematon tositarina siitä, kuinka kirjaston ominaisuudet yllättivät opiskelijan johtamalla hämmästyttävän hitaaseen ohjelmakoodiin, ja kuinka vaikeaa hänen opettajansa — siis minun — oli löytää hitauden syy.

1 Johdanto

Hyvien tietorakenteiden ja algoritmien käytöllä on tunnetusti suuri merkitys tietokoneohjelmien suoritusajalle. Siksi tärkeimmät perusalgoritmit ja -tietorakenteet vakiintuivat osaksi tietojenkäsittelyalan akateemisia oppisisältöjä jo varhain, ainakin ohjelmointitekniikkaan tai teoreettiseen tietojenkäsittelytieteeseen suuntautuneissa koulutusohjelmissa.

Pyörää ei kuitenkaan kannata keksiä toistuvasti uudelleen. Niinpä perusalgoritmeja ja -tietorakenteita alettiin koota kirjastoihin, joista niitä voi ottaa valmiina käyttöön. Kirjastoja alettiin sisällyttää ohjelmointikielten osaksi. Niin tehtiin muun muassa ohjelmointikielen C++ kohdalla. C++-standardi [2] takaa, että kielen kääntäjän mukana tulee kirjasto, jossa on suuri joukko monenlaisia palveluja, mukaan lukien tavallisimpien perusalgoritmien ja -tietorakenteiden tarjoamat palvelut.

Koska perusalgoritmit ja -tietorakenteet ovat saatavana valmiina, herää kysymys, tarvitseeko niitä enää opettaa opiskelijoille kuten aikaisemmin. Eikö riitä, että opetetaan niiden tuottamat palvelut sekä kerrotaan, miten ne saa käyttöön ja mitä niiden käyttö maksaa? Eihän nykyisin tarvitse opettaa sitäkään, miltä aliohjelman kutsu näyttää konekielen tasolla.

Asia ei, ikävä kyllä, ole ongelmaton. Valmiiden algoritmikirjastojen käyttöön liittyy ansoja. Alkusysäyksen tälle kirjoitukselle antoi, kun opintojaksolleni har-

OHJE KIRJAPAINOLLE: B5-arkin vasen- ja yläreuna kohdistetaan A4-arkin vasempaan ja yläreunaan. Näin pitäisi marginaaliksi tulla taitteen puolella noin 33 mm ja muualla noin 22 mm.

joitustyötä tekevä opiskelija otti yhteyttä. Hänen laatimansa ohjelman eräs osa toimi kummallisen hitaasti. Hän uskoi jäljittäneensä syyn C++:n standardikirjaston erääseen toimintoon, jonka kaiken järjen mukaan ei pitäisi olla hidas. Pulman ratkaiseminen ei ollut minulle ihan helppoa — vaikka jälkiviisaana voin todeta, että olisi pitänyt olla.

Luvussa 3 on kerrottu opiskelijan pulma ja kuinka sain sen ratkaistua. Tarina on kerrottu kaikessa raadollisuudessaan, mukaan lukien hölmöt ratkaisuehdotukseni. Toivon, että ongelmanratkaisutyön tunnelma välittyy kertomuksesta aitona. Teksti on alun perin kirjoitettu tapahtumia seuranneen kuukauden aikana. Yksityiskohdat olivat tuoreessa muistissa, joten ne ovat varsin luotettavia. Tosin kaikkien tapahtumien järjestystä en enää muistanut, ja toistin tekemiäni kokeita useammin kuin tekstissä kerrotaan.

Luvussa 2 on C++:n standardikirjaston tietorakenteiden esittely. Sen alkuperäinen tarkoitus oli antaa pulman ja sen ratkaisun edellyttämät taustatiedot, mutta yksityiskohtien määrä kasvoi niin suureksi, että esittely kannatti laajentaa kattamaan keskeiset periaatteet kokonaan. Esittely painottaa kirjastoon liittyviä ongelmia tavallista enemmän ja jättää merkittävät edut vähälle huomiolle. Kirjoittajan kokemuksen mukaan edut tunnetaan hyvin, joten niitä ei tarvitse toistaa. Sen sijaan ongelmista on liian vähän tietoa liikkeellä ottaen huomioon, että niitä ei voi varoa ellei niitä tunne. Ongelmien painottaminen sopii muutenkin tämän kirjoituksen luonteeseen.

Lisätiedoista kiinnostuneille suosittelen erinomaista kirjaa [3]. C++-kielen perusteoksen [6] standardikirjasto-osuutta neuvon välttämään, sillä se on sekava ja jopa vaarallisen epätarkka.

Viimeisessä luvussa kootaan tapahtumien minulle antamat opetukset ja pohditaan, mikä sanoma niillä on tietorakenteiden ja algoritmien opetukselle.

Tässä kirjoituksessa olevat viittaukset C++-standardiin kohdistuvat vuoden 1998 versioon [2], paitsi kun toisin sanotaan. Standardista on lähiaikoina tulossa uusi versio. Moni jäljempänä mainituista pikkuvirheistä on korjattu sen luonnoksissa. Viittaukset Gnu-kirjastoon kohdistuvat GNU ISO C++ Libraryn versioon 3.3.5,

2 C++:n standardikirjaston tietorakenteet

2.1 Säiliöt ja muotit

Ohjelmointikielen C++ määritelmä sisältää laajan osan nimeltä *C++ Standard Library* eli C++:n standardikirjasto. Siihen sisältyy sekä useasti käytettäviä toimintoja, kuten syöttö ja tulostus, että vain toisinaan tarvittavia, kuten kompleksiluvut. Tämän kirjoituksen kannalta olennaista on, että siihen sisältyy kokoelma *säiliöitä* (*container*). Säiliöt ovat käytännössä tiettyjen tietorakenteiden toteutuksia.

C++-standardi määrittelee säiliöt vain niiden rajapintojen avulla, eli kuvaamalla kunkin säiliön tuottamat palvelut ja niiden ajan kulutuksen. Standardi ei siis nimenomaan vaadi, että esimerkiksi `list`-niminen säiliö on toteutettu kaksisuuntaisena linkitettynä listana. Palvelujen ominaisuudet on kuitenkin määritelty niin yksityiskohtaisesti, että yleensä ei ole kuin yksi laajalti tunnettu tietorakenne, jolla vaatimukset voi saavuttaa. Niinpä on käytännöllisesti katsoen varmaa, että `list` on toteutettu kaksisuuntaisena linkitettynä listana.

Säiliö nimeltä `map` tallettaa avaimesta ja muusta tiedosta koostuvia pareja siten,

että niitä voi lisätä, poistaa, etsiä avaimen perusteella ja selata avaimen mukaisessa suuruusjärjestyksessä logaritmisessa ajassa. Se on jokseenkin varmasti toteutettu tasapainotettuna binääripuuna. Gnu C++-kääntäjän mukana tulevassa kirjastossa se on puna-musta puu, ja koodin kommenteissa viitataan kirjaan [1].

Yleiskäyttöisen tietorakennetoteutuksen täytyy jotenkin selvitä siitä, että joskus siihen halutaan tallettaa pieniä kohteita, kuten kokonaislukuja, ja joskus suuria kohteita, esimerkiksi henkilötietueita. Jos tietorakenne tarjoaa mahdollisuuden selata alkioita suuruusjärjestyksessä, kuten `map`, sille täytyy voida kertoa, miten esimerkiksi henkilötietueiden välinen järjestys määräytyy.

C++:n standardikirjastossa nämä on ratkaistu käyttäen *muotteja* (*template*). Hieman yksinkertaistaen, muotti on C++-kieleen sisältyvä funktio tai luokka, joka ottaa parametrikseen ainakin yhden tyypin. Jos henkilötietueita varten on luotu tyyppi `henkilotietue`, niin ohjelmoija voi luoda kokonaislukuja tallettavan listan kirjoittamalla

```
std::list< int > kl_lista;
```

ja henkilötietueita tallettavan listan kirjoittamalla

```
std::list< henkilotietue >
  ht_lista;.
```

Hakemisto, josta opiskelijan tietoja voidaan etsiä opiskelijanumeron perusteella, saadaan määritelmällä

```
std::map< int, henkilotietue >
  opiskelijahakemisto;.
```

Kirjoitin edellä “hieman yksinkertaistaen”, koska muotilla ei todellisuudessa ole pakko olla tyyppiparametreja. Parametreina voi tyyppien sijaan tai lisäksi olla esimerkiksi lukuvakioita. Jäljempänä mainittava `bitset` on esimerkki.

Ohjelmoija voi luoda tyypin `henkilotietue` tekemällä siitä luokan, eli käyttämällä C++:n olio-ohjelmointia tukevia piirteitä. Siinä yhteydessä hän voi määritellä, miten henkilötietueita verrataan operaattorilla “`<`”. Vaihtoehtoisesti vertailutapa voidaan antaa `map`-muotille kolmantena parametrina. Tätä varten C++:ssa on keino naamioida funktio tyypiksi. Funktion voi naamioida myös muuttujaksi ja antaa sen muuttujalle `opiskelijahakemisto` sen luomistapahtuman ainoana parametrina.

Muottien yleisin (kenties ainoa käytössä oleva) toteutustapa muistuttaa makrojen toteutusta. Kääntäjällä on käytettävissään muotin kaikki määritelmät ja toiminnot lähdekooditasolla siten, että parametrina käytettävän tyypin, luvun tai muun tiedon tilalla on jokin muodollinen nimi, esimerkiksi `T`. Kun muottia käytetään ensimmäisen kerran, kääntäjä ikään kuin korvaa sen lähdekielisessä kuvauksessa esiintyvät muodolliset parametrien nimet kutsussa käytetyillä tiedoilla ja kääntää lopputuloksen. Esimerkiksi jokainen muodollisen tyypin `T` esiintymä korvataan kutsussa käytetyn tyypin `henkilotietue` esiintymällä, ja näin saatu koodi käännetään. Tehokkuussyistä korvaaminen ei todellisuudessa tapahdu lähdekooditasolla vaan jossakin väliesitysmuodossa. Jos samaa muottia käytetään uudelleen siten, että ainakin yksi parametri on eri kuin aikaisemmin, niin muotista käännetään uusi kopio. Jos uudessa käytössä on samat parametrit kuin aikaisemmassa, uutta kopiota ei tehdä.

Tällä toteutustavalla tietorakenne saadaan mukautumaan kulloiseenkin tyyppiin ja sen operaatioihin ilman, että suoritettavalle ohjelmalle aiheutuu muuta ylimääräistä kustannusta kuin se, että käännetyssä ohjelmassa voi olla samankaltaista koodia monena kopiona. Siltä varal-

ta, että jokin asia voidaan toteuttaa erityisen tehokkaasti jollekin yleisesti käytetylle tyypille, C++ sisältää mahdollisuuden määritellä muotti siten, että jos tyyppiparametrina on mainittu tyyppi, niin muotille käytetään eri toteutusta kuin muulloin.

C++:n standardikirjaston tietorakenne- ja algoritmiosa tunnetaan yleisesti nimellä *standard template library* ja lyhenteellä *STL*. Nimityksellä on historiallinen tausta. Se ei ole täysin osuva, sillä melkein kaikki muutkin standardikirjaston osat käyttävät muotteja. C++-standardi ei itse käytä nimitystä.

2.2 Standardikirjaston säiliöt ja “melkein säiliöt”

Tämän kirjoituksen esimerkeissä mainitaan melkein kaikki C++:n standardikirjaston säiliöt ja säiliöitä läheisesti muistuttavat muotit, joten kattavuuden nimessä tässä alaluvussa esitellään ne kaikki lyhyesti.

`List` ja `map` on jo mainittu. Jälkimmäisen kaltaisia ovat `multimap`, `set` ja `multiset`. `Multimap` sallii samalle avaimelle monta avaimesta ja muusta tiedosta koostuvaa paria, toisin kuin `map`. `Set` ja `multiset` ovat edellisten kaltaisia, mutta niihin talletetaan pelkkiä avaimia, siis ilman oheistietoa. Nämä neljä säiliötä käyttävät avainten vertailuihin aina toimintoa “<” tai käyttäjän sille antamaa korviketta. Kaksi avainta katsotaan yhtäsuuriksi, jos ja vain jos kumpikaan ei ole toista pienempi.

`Multiset` tallettaa avaimen monesti eikä ainoastaan pidä kirjaa avaimen talletuskertojen määrästä. Näin siksi, että vaikka käytetty vertailutapa julistaisi kaksi avainta yhtäsuuriksi, avaimet eivät välttämättä ole täysin identtiset. Esimerkiksi merkkijonojen järjestyksen vertailussa on joskus mielekästä jättää isojen ja pienten kirjainten ero huomiotta.

C++:n säiliöistä ei voi kovin kauaa puhua mainitsematta *vektoria* `vector`. Se on muuten kuten tavallinen taulukko, mutta siinä on ylimääräisiä toimintoja. Erityisen tärkeää on, että sillä ei ole käyttöönoton yhteydessä määrättyä kiinteää kokoa, vaan sen koko kasvaa automaattisesti sitä mukaa kuin siihen lisätään alkioita. Todennäköinen toteutus toimii siten, että alkiot talletetaan kiinteän kokoiseen taulukkoon, ja kun siitä loppuu tila kesken, varataan uusi, kaksi kertaa edellisen kokoinen kiinteä taulukko, ja siirretään alkiot sinne. Varattua taulukkoa ei vaihdeta pienempään alkioiden määrän vähentyessä.

Lisääminen vektorin loppuun on useimmiten vakioaikaista. Se vie lineaarisen ajan silloin, kun alkiot joudutaan siirtämään pieneksi jääneestä taulukosta uuteen taulukkoon. Tämä tapahtuu edellä kuvatulla toteutuksella niin harvoin, että jos alkioita lisätään yksi kerrallaan alun perin tyhjän taulukon loppuun mikä tahansa määrä, niin ajan kulutus on yhteensä vähemmän kuin se olisi, jos lisäys olisi aina vakioaikaista kolminkertaisella vakion arvolla. Tästä syystä sanotaan, että alkion lisääminen vektorin loppuun on *tasatusti* vakioaikaista.

Käyttäjä voi milloin tahansa antaa luvun ja käskeä, että tilaa pitää varata tai olla varattuna vähintään niin monelle alkiolle. Näin hän voi varmistaa, että alkioiden uudelleensijoittamista ei tule tapahtumaan, jos hän tietää ennakolta alkioiden lopullisen määrän.

Vektoria voi ja on suositeltavaa käyttää melkein kaikkialla, missä muuten käytettäisiin “tavallista”, C-kielestä perittyä taulukkoa.

Vektoreista on olemassa totuusarvojen tallettamiseen erikoistunut versio `vector<bool>`. Se tarjoaa toteutukselle mahdollisuuden pakata yhden totuusarvon

bittiä kohden, kun tavallisen vektorin toteutus varaa välttämättä ainakin yhden tavun kullekin alkiolle. Hintana on indeksoinnin hidastuminen sekä alkion osoitteen käsitteen hämärtyminen, koska osoitteeksi ei enää riitä tavun osoite muistissa, vaan täytyy myös ilmaista bitin indeksi tavun sisällä. C++:n tavanomainen osoitteen käsite ei siis ole käytettävissä. Tästä seuraa, että ei ole taattua, että valmis algoritmi, joka toimii muille vektoreille, toimii myös muotille `vector<bool>`.

Pakka `deque` on kaksipäinen jono. Sen molempiin päihin voi lisätä ja päistä poistaa alkioita nopeasti. Toisin kuin muiden säiliöiden taustalla olevat tietorakenteet, pakan rakenne ei ole algoritmikirjallisuudesta tuttu. Pakka on tyypillisesti toteutettu taulukollisella osoittimia taulukoihin, joissa alkiot ovat. Alkiotaulukot ovat kiinteän kokoisia, mutta niiden määrä kasvaa ja vähenee tarpeen mukaan. Osoitintaulukkoa kasvatetaan tarpeen mukaan kuten vektoria. Tehokas lisääminen alkuun voidaan mahdollistaa käyttämällä osoitintaulukkoa rengaspuskurina. Toinen, Gnu-kirjastossa käytetty vaihtoehto on siirtää osoittimet osoitintaulukon keskialueelle aina kun toinen reuna tulee vastaan ja vastakkaisessa reunassa on runsaasti tilaa.

Standardin kohta 23.2.1 1 lupaa, että pakan päihin lisääminen ja poisto tapahtuvat vakioajassa. Osoitintaulukon kasvattaminen ja osoitinten siirtäminen sen keskialueelle eivät kuitenkaan onnistu vakioajassa, vaan ainoastaan tasatusti vakioajassa. Luvussa 4 tullaan palaamaan kysymykseen, onko standardissa tässä kohdassa pieni virhe vai käyttääkö se tavallisuudesta poikkeavaa tulkintaa suoritusaikaa koskeville väittämille. Virhe on tuskin toteutuksessa, sillä toteutus oli ensin ja otettiin huomioon standardia tehtäessä.

Jono `queue`, pino `stack` ja prioriteettijono `priority_queue` eivät ole tarjolla itsenäisinä säiliöinä, vaan *säiliömuuntimina* (*container adapter*), jotka ottavat sisäänsä esimerkiksi linkitetyn listan ja muuttavat sen rajapinnan jonon tai pinon rajapinnan mukaiseksi. Pinon perustana voi käyttää vektoria listan sijaan, mutta jonon perustana ei voi, koska vektorin alkuun ei voi lisätä eikä alusta voi poistaa alkioita tehokkaasti. Prioriteettijonon kohdalla tilanne on päinvastainen: vektori kelpaa, lista ei. Kaikkien kolmen perustana voi käyttää myös pakkaa.

Kirjallisuus on standardi mukaan lukien epäselvä sen suhteen, ovatko jono, pino ja prioriteettijono säiliöitä. Niille ei ole selaustoimintoja, toisin kuin kiistattomille säiliöille. Toisaalta standardi esittelee ne samassa nipussa samojen otsikoiden alla kuin listan, vektorin ja pakan.

Merkkijonojen käsittelyä varten on muotti `string`. Se mukautuu merkkijonon pituuteen automaattisesti ja huolehtii itse tarvittavan muistin varaamisesta ja vapauttamisesta. Siinä on toimintoja sekä yksittäisten merkkien, osajonojen että kokonaisten merkkijonojen käsittelyyn. Sitä ei ole luokiteltu säiliöksi, vaikka sillä on paljon säiliöiden kanssa yhteisiä ominaisuuksia ja sitä voi laajalti kohdella kuten säiliöitä. Muotin `basic_string` avulla käyttäjä voi jopa luoda merkkijonojen lailla käyttäytyviä tyyppejä, joiden alkiot eivät ole merkkejä vaan käyttäjän valitsemaa tyyppiä. (Kannattaa tosin ottaa huomioon, että `basic_string` on optimoitu tilanteeseen, jossa alkio on pieni olento.)

`Bitset` on kiinteän kokoinen taulukko bittejä. Sillä on monipuolisemmat bitinkäsittelyoperaatiot kuin muotilla `vector<bool>`. Se on C++-standardissa luokiteltu säiliöksi, vaikka, toisin kuin muilla säiliöillä, käyttäjä ei pääse valitsemaan alkioiden tyyppiä, eikä sen kokoa voi muuttaa käytön aikana. (Standardin

uudessa versiossa sen luokittelua aiotaan muuttaa.) `Bitset` täyttää standardin kohdassa 23.1 1 esitetyn säiliön luonnehdinnan huonosti, mutta `basic_string` täyttää sen hyvin.

Säiliöiden kaltaiseksi mainitaan toisinaan myös `valarray`. Se on tarkoitettu matemaattisten vektorien ja matriisien esittämiseen. Käytettävä lukutyyppi on sen muottiparametri. Kirja [3, s. 547] toteaa siitä, että

> `Valarray`-luokkia ei suunniteltu kovin hyvin. Itse asiassa kukaan ei yrittänyt selvittää, toimiiko lopullinen määritelmä. Näin kävi, koska kukaan ei tuntenut olevansa "vastuussa" niistä luokista. Ihmiset, jotka toivat `valarray`:n C++:n standardikirjastoon jättivät komitean kauan ennen standardin valmistumista.

Perustietorakenteet hyvin tunteva lukija saattaa tässä vaiheessa ihmetellä, että alaluku lähenee loppuaan, mutta hajautustaulua ei ole vielä mainittu. Arvoituksen ratkaisu on siinä, että voimassa olevassa C++-standardissa ei ole hajautustaulua ollenkaan. Syynä on kenties se, että hajautustaulu on ristiriidassa säiliöjärjestelmän erään keskeisen piirteen kanssa, kuten seuraavassa alaluvussa tullaan kertomaan. Hajautustaulu on kuitenkin niin tärkeä, että standardin seuraavaan versioon se on tulossa.

2.3 Selaimet ja pyrkimys yhtenäisyyteen

C++:n säiliöiden yhteydessä on välttämätöntä mainita *selain* (*iterator*). Se on ikään kuin tarttumakeino säiliön sisällä olevaan alkioon. Selaimia saadaan muun muassa etsintätoimintojen tuloksina, ja niitä voi tallettaa selaintyyppisiin muuttujiin. Alkion tietoja voi lukea ja muuttaa ja alkion voi poistaa selainten avulla. Selaimia on eri lajeja sen mukaan, mitä alkiolle voi tehdä selaimen avulla ja miten selaimen voi siirtää alkiosta toiseen.

Selain edustaa alkion paikkaa säiliössä ajateltuna säiliön tuottaman palvelun (eikä säiliön toteutuksen) kannalta. Vaikka merkkijonojen ja joidenkin säiliöiden tapauksessa alkion paikan voi ilmaista myös indeksillä, selain on eri asia kuin indeksi. Selaimet ovat samannäköisinä käytettävissä sekä useimmille niille säiliöille, joille voi käyttää indeksiä, että useimmille niille, joille ei voi.

Esimerkiksi jokaisen kiistattoman säiliön alkiot voi selata läpi kuten kuvassa 1 on näytetty. Sama toimii myös merkkijonoille. Kuvassa `begin` palauttaa selaimen, joka osoittaa säiliön ensimmäiseen alkioon. Vastaavasti `end` palauttaa selaimen säiliön viimeisen alkion jäljessä välittömästi olevaan tyhjään (kuvitteelliseen) paikkaan. `++` siirtää selaimen alkiosta seuraavaan. Standardin kohta 24.1 8 lupaa näiden toimivan säiliöille tasatussa vakioajassa.[1]

Selainten ansiosta alkioiden paikoista erilaisissa säiliöissä voidaan puhua yhtenäisellä tavalla. Esimerkiksi useimpiin säiliöihin voi lisätä alkion toiminnolla

[1] Kirjoittaja uskoo tässäkin olevan pienen virheen standardissa. Ainakaan Gnu-kirjaston `map` ei täytä lupausta. Toiminto `++` on toteutettu tavallisella isä- ja lapsiosoittimia pitkin etenevällä seuraajan etsinnällä binääripuussa. Se on tunnetusti tasatusti vakioaikainen, kun sitä käytetään vain selaamaan koko puu läpi. Kuitenkin, jos selain osoittaa sopivaan kohtaan puuta, niin kutsumalla toimintoa `++` ja sille vastakkaista toimintoa `--` vuorotellen sadaan mielivaltaisen pitkä toimintojono, jonka jokainen toiminto vie logaritmisesti aikaa.

```
for( selaintyyppi s = sailio.begin(); s != sailio.end(); ++s ){
  tee_jotain( *s );   // *s viittaa selattavaan alkion
}
```

Kuva 1: Säiliön alkioiden selaus.

`sailio.insert( paikka, alkio );`,

missä paikka on ilmaistu selaimena. Niille säiliöille, jotka päättävät alkion paikan itse, parametri `paikka` toimii vihjeenä, jonka avulla oikean paikan löytymistä voi nopeuttaa. Lisääminen keskelle säiliötä on säiliön lajista riippuen vakioaikaista, lineaarista tai siltä väliltä.

Säiliöiden toiminnoissa on muutenkin pyritty yhtenäisyyteen. Esimerkiksi jokaiselta säiliöltä voi kysyä sen sisältämien alkioiden määrän funktiolla `size`:

`if( kl_lista.size() < 100 ){ ... }`

Yhtenäisyys ei kuitenkaan ole aukotonta. Kaikilta muilta säiliöiltä voi kysyä toiminnolla `empty` onko se tyhjä, mutta säiliöltä `bitset` ei voi. (Myös `valarray` tarjoaa toiminnon `size` mutta ei toimintoa `empty`. Sitä ei kuitenkaan yleensä luokitella säiliöksi.) Standardin laatijat ovat kenties ajatelleet, että `empty` on säiliön `bitset` yhteydessä täysin tarpeeton, koska kukin `bitset` on joko aina tyhjä tai aina epätyhjä.

Aikaisemmin mainittiin toinen esimerkki epäyhtenäisyydestä: algoritmi, joka toimii muille vektoreille, ei välttämättä toimi muotille `vector<bool>`. Syynä oli se, että muotin `vector<bool>` toteutus muuttaa osoitteen käsitettä, ja algoritmi saattaa olla riippuvainen siitä, että osoite on kuten C++:ssa yleensä. Jatkossa tulee lisää esimerkkejä epäyhtenäisyydestä mahdollisine syineen.

Koska erilaisten säiliöiden sisältöjä voi selata yhtenäisillä tavoilla, on C++-standardiin voitu sisällyttää suuri joukko algoritmeja, jotka on määritelty vain kerran, mutta joita voi käyttää monenlaisille säiliöille. Esimerkiksi

`find_if( alku, loppu, ehto )`

etsii säiliöstä tai sen osasta ensimmäisen alkion, joka täyttää annetun ehdon. Alku ja loppu ilmaistaan selaimina. Ehto ilmaistaan funktiona tai luokkana, joka ottaa alkion tai viitteen alkioon parametrikseen ja palauttaa totuusarvon. Se voi olla käyttäjän itse kirjoittama. (C++:ssa *viite* eli *reference* on toteutukseltaan osoittimen kaltainen keino antaa kohteelle toinen nimi, jota käytetään kuten tavallista nimeä eikä kuten osoitinta.)

Valmis algoritmi toimii myös käyttäjän itse toteuttamalle tietorakenteelle, jos se tarjoaa algoritmin edellyttämät selaintoiminnot, ja jos alkion tyyppi tarjoaa algoritmin edellyttämät toiminnot, kuten vertailun “`<`”.

Algoritmilla voi olla eri toteutuksia selattavasta säiliöstä riippuen. Hyvä esimerkki on `distance( eka, toka )`, joka palauttaa kahden alkion välisen etäisyyden säiliössä. Niille säiliöille, joita voi indeksoida, `distance` palauttaa `toka`a ja `eka`a vastaavien indeksien erotuksen ja toimii siis nopeasti. Muille säiliöille `distance` selaa säiliötä alkaen paikasta `eka` kunnes paikka `toka` tulee vastaan, ja palauttaa selausaskelten määrän. Suoritusaika on silloin lineaarinen.

Säiliöiden ja algoritmien muodostaman kokonaisuuden eräänä keskeisenä suunnitteluperiaatteena on siis ollut, että säiliöitä voi selata kuvan 1 esittämällä tavalla tehokkaasti. Hajautustaulu on

tässä suhteessa ristiriidassa säiliöjärjestelmän kanssa, sillä sen selaus ei ole tehokasta. Tehokkuutta voi yrittää taata ylimääräisillä aputietorakenteilla tai säätämällä jatkuvasti hajautustaulun kokoa siten, että sen täyttöaste on koko ajan suurehko. Tällöin kuitenkin hajautustaulun vahvin etu — erittäin nopea etsintä ja lisäys — kärsii. Hajautustaulu poikkeaa standardin säiliöistä myös siten, että alkioilla on säiliöissä joko käyttäjän tai alkioiden tyypin määräämä järjestys, mutta hajautustaulussa kumpaakaan järjestystä ei voi ylläpitää.

Yhtenäisyyteen pyrkiminen on johtanut yllättäviinkin seurauksiin. Puolitushaku lienee maailman yleisimmin käytetty esimerkki tehokkaasta algoritmista. C++:ssa se on nimellä `binary_search`. Sitä voi kutsua myös linkitetyille listoille, vaikka niille sitä ei voi toteuttaa tehokkaasti! Standardi lupaa logaritmisen suoritusajan niille säiliöille, joita voi indeksoida. Linkitetyille listoille standardi lupaa, että alkioita verrataan vain logaritminen määrä kertoja, mutta kokonaissuoritusaika saa olla lineaarinen. Standardin laatijat lienevät ajatelleet, että jos listan alkiot ovat monimutkaista tietotyyppiä, niiden vertaaminen saattaa olla paljon hitaampaa kuin listan selaaminen linkkejä seuraamalla, joten puolitushaun tarjoamisesta myös listoille voi olla hyötyä.

Toisaalta standardin laatijat ovat paikoitellen jättäneet loogisesti mielekkään toiminnon tarkoituksella pois niistä säiliöstä, joille sitä ei voi toteuttaa tehokkaasti. Esimerkkinä tästä ovat `push_front`, `pop_front`, `push_back` ja `pop_back`, jotka lisäävät ja poistavat säiliön ensimmäisen ja viimeisen alkion. Standardin kohta 23.1.1 12 sanoo, että ne tarjotaan vain niille säiliöille, joille ne toimivat tasatusti vakioajassa.[2] Vektorille on yllä mainitut `back`-toiminnot mutta ei `front`-toimintoja, koska vektoria voi venyttää ja kutistaa tehokkaasti lopusta, mutta ei alusta.

2.4 Tehokkuus vai turvallisuus?

C++:n standardikirjaston tietorakenne- ja algoritmiosuudessa on pyritty ensisijaisesti erinomaiseen suorituskykyyn. Käytön turvallisuus on ollut toissijaista. Esimerkiksi on täysin käyttäjän vastuulla huolehtia, että toimintoa `pop_back` ei kohdisteta tyhjään säiliöön. Myös on täysin käyttäjän vastuulla huolehtia, että `distance` ei vahingossa selaa säiliön lopun ohi etsiessään paikkaa `toka`. Olisi ollut mahdollista suunnitella säiliöjärjestelmä siten, että nämä toiminnot huomaisivat virhetilanteet. Se olisi kuitenkin lisännyt suoritusaikaa, ja sitä ei haluttu.

Vektorin ja pakan indeksoinnin tapauksessa käyttäjälle on annettu mahdollisuus valita tehokkaan mutta turvattoman ja tehottomamman mutta turvallisemman vaihtoehdon väliltä. Niitä voi indeksoida normaalin hakasulkusyntaksin `A[i]` lisäksi tyyliin `A.at(i)`. Jälkimmäisessä tapauksessa käännetty ohjelma tarkastaa, että indeksi osuu käytössä olevalle alueelle.

Säiliöjärjestelmä hyödyntää tehokkaasti C++:n tyyppijärjestelmän tarjoamia keinoja käytön turvallisuuden lisäämiseen. Säiliöön ei esimerkiksi voi tallettaa muun tyyppistä alkiota kuin mitä säiliötä luotaessa määriteltiin. C++:ssa tyyppien täsmääminen tarkastetaan käännösaikana, joten se ei hidasta ohjelman suoritusta.

Käytön turvallisuuden kannalta merkittävä linjanveto on ollut, että kun alkio talletetaan säiliöön, säiliö ottaa siitä ko-

[2] Vuoden 1998 standardi lupaa vakioajan, mutta siinä on kiistatta virhe, koska lisäys vektorin loppuun on vain tasatusti vakioaikainen. Sana “tasatusti” on lisätty standardin uuteen versioon.

```
if(
  opiskelijahakemisto.insert(
    std::make_pair( opiskelijanro, henkilotiedot )
  ).second
){
  ... // opiskelijanro ei ollut käytössä, tiedot lisättiin
}else{
  ... // opiskelijanro oli käytössä, tietoja ei lisätty
}
```

Kuva 2: Opiskelijan tietojen lisäys, paitsi jos opiskelijanumero on jo käytössä.

pion ja huolehtii itse sen vaatiman muistin varaamisesta ja vapauttamisesta. Toinen mahdollisuus olisi ollut, että säiliöihin talletettaisiin osoittimia tai viitteitä alkioihin. Alkioiden muistin hallinta olisi silloin käyttäjän vastuulla. Lisäksi käyttäjä voisi vahingossa esimerkiksi sotkea säiliön `map` sisäisen järjestyksen muuttamalla siellä olevan alkion avainta ja jättämällä kertomatta säiliölle, että teki niin.

Koska säiliöllä on alkiosta kopio, avaimen muuttaminen säiliöltä salaa ei onnistu noin vain. Säiliön rajapinta estää säiliössä olevan alkion avaimen muuttamisen. Käyttäjä voi kiertää tätä rakentamalla avaimet ja niiden vertailun siten, että varsinainen tieto on säiliön ulkopuolella, ja säiliölle annettava avain ainoastaan kertoo, mistä se löytyy. Tämän voi tehdä helposti osoittimien avulla. Käyttäjä joutuu kuitenkin silloin toteuttamaan vertailutoiminnon itse. Se pienentää riskiä, että hän valitsee tämän ratkaisun ymmärtämättä, mitä on tekemässä.

2.5 Onko säiliöiden käyttö helppoa?

Pyrkimys toimintojen yhtenäisyyteen on tehnyt niiden käytöstä paikoitellen kömpelöä. Esimerkiksi usein tarvitaan erityistä paritusmekanismia. Kun aikaisemman esimerkin opiskelijahakemistoon lisätään uusi opiskelija, joudutaan hänen opiskelijanumerostaan ja henkilötiedoistaan tekemään pari, kuten kuvassa 2 on näytetty.

Joskus suositellaan, että toimintojen paluuarvot pitäisi välittää funktion tuloksena eikä viiteparametreina. `Map`-säiliön `insert` palauttaa sekä paikan, johon alkio lisättiin, että tiedon lisäyksen onnistumisesta. Lisäys ei onnistu, jos samalla avaimella on jo alkio. `Insert` palauttaa siis kaksi tietoa. Jotta ne voisi välittää funktion tuloksena, on ne yhdistetty pariksi. Paikkatieto on sen kentässä `first` ja onnistumistieto kentässä `second`. Kuvassa 2 on hyödynnetty onnistumistietoa. Jos olisi haluttu hyödyntää molempia tietoja, olisi toiminnon `insert` tuloksen vastaanottamiseksi pitänyt luoda muuttuja tyyppiä

```
std::pair< paikkatyyppi, bool >,
```

missä `paikkatyyppi` on

```
std::map< int, henkilotietue >
   ::iterator.
```

Toisaalta on myös tilanteita, joissa säiliön käyttö on todella yksinkertaista. Jos halutaan muuttaa opiskelijan tietoja siten, että jollei kyseessä oleva opiskelijanumero ole käytössä, opiskelija samalla lisätään, se onnistuu käskyllä

```
opiskelijahakemisto[ nro ] =
  henkilotiedot;.
```

Valitettavasti yksinkertaisuuteen kätkeytyy ansa. Se houkuttelee esimerkiksi tulostamaan opiskelijan tietoja tyyliin

```
std::cout << oh[i].nimi <<
  oh[i].osoite << oh[i].puhnro;,
```

missä `oh` on opiskelijahakemisto. Tällöin opiskelija etsitään opiskelijahakemistosta uudelleen jokaista tulostettavaa tietoa varten. Koska säiliöstä `map` etsimiseen menee merkittävä aika, on tämä ajan tuhlausta. Se voidaan välttää luomalla löydettyyn opiskelijaan viite ja käyttämällä tulostukseen sitä:

```
henkilotietue &opisk = oh[i];
std::cout << opisk.nimi <<
  opisk.osoite << opisk.puhnro;.
```

Merkki "`&`" on tärkeä, sillä jos se puuttuu, ohjelma ottaakin löydetyn opiskelijan tiedoista viitteen sijasta kopion, mikä on hidasta, jos tietoja on paljon.

Moninkertaisen etsinnän ongelma ratkaisuineen koskee vähäisemmässä määrin myös pakkaa ja merkkijonoa, ja ehkä jopa vektoria.

Käyttäjän on syytä olla tietoinen myös siitä, että osalle standardikirjaston palveluista on valittu yksinkertainen mutta tehoton algoritmi monimutkaisen mutta huomattavasti tehokkaamman sijaan. Tämä koskee esimerkiksi merkkijonon etsimistä toisen merkkijonon sisältä sekä ensimmäisen sellaisen alkion etsimistä, joka kuuluu annettuun joukkoon alkioita.

Joskus merkitykselliseksi nousee myös se, että käyttäjä ei voi laajentaa säiliöitä. Säiliötä `priority_queue` ei voi valjastaa Dijkstran algoritmin toteutukseen (katso [1, s. 530]), koska se ei sisällä toimintoa DECREASE-KEY eikä sen mahdollistamiseksi tarvittavia lisäpiirteitä. Säiliötä `map` kaltaisineen ei voi laajentaa esimerkiksi intervallipuuksi kirjan [1] luvun 15 mukaisesti, koska käyttäjä ei pääse lisäämään tarvittavia ylläpitotoimintoja säiliön toteutukseen. Standardihan sallii, että `map` ei edes ole binääripuu.

2.6 Loitsut, sokkosäännöt ja selainten vanheneminen

Kaiken kaikkiaan C++:n säiliöiden käyttämiseksi tehokkaasti ja turvallisesti on osattava paljon asioita ainakin "loitsujen" ja "sokkosääntöjen" tasolla. *Loitsu* on Tampereen teknillisen yliopiston ohjelmistotekniikan opiskelijoiden ja opettajienkin kielenkäytössä tekstinpätkä, joka on pakko kirjoittaa ohjelmaan jotta se kääntyisi ja toimisi alkuunkaan mielekkäästi, mutta josta ei yritetäkään ymmärtää, miksi se on sellainen kuin on. Opiskelija voi ottaa esimerkiksi kuvan 2 rivit 2, 3 ja 4 loitsuna, jossa hän korvaa sanat `opiskelijahakemisto`, `opiskelijanro` ja `henkilotiedot` omilla sanoillaan ja kopioi muun merkki merkiltä.

Sokkosääntö on uudissana, jota ehdotan tarkoittamaan sääntöä, jonka noudattaminen on tarpeellista vaikka kääntäjä ja ensimmäiset koeajot eivät siihen pakota, ja jonka syytä ei ymmärretä. Käsky muistaa "`&`" edellä olleessa esimerkissä voi olla sellainen. Sokkosääntö eroaa loitsusta siten, että loitsun noudattamatta jättämistä ei voi olla huomaamatta. Sokkosäännöistä tulee pian lisää esimerkkejä.

Jos jonkin säännön noudattamista vaaditaan toistuvasti silloinkin, kun sen noudattaminen ei ole tarpeellista, kutsun sitä Raamatusta poimitulla ilmauksella *vanhinten perinnäissääntö*. Vanhinten perinnäissääntö voi syntyä esimerkiksi siten, että ohjelmoinnin alkeiskurssilla globaalien muuttujien käyttö kielletään ehdottomasti, jotta opiskelijat oppisivat käyttämään aliohjelmien parametreja. Kiellon ankara valvonta takaa, että se jää opiskelijoiden mieleen, mutta mikään ei takaa,

```
s1 = sailio.begin();
while( s1 != sailio.end() ){
  s2 = s1; ++s2;
  if( s2 != sailio.end() && *s1 == *s2 ){ sailio.erase( s2 ); }
  else{ s1 = s2; }
}
```

Kuva 3: Välittömien kopioiden poistaminen säiliöstä.

että heidän mieleensä jää myös, että kielto oli vain tätä opintojaksoa varten. Opiskelijoiden mieleen jää vain, että vaikka kääntäjä hyväksyy globaalit muuttujat, niitä ei saa milloinkaan käyttää.

Esimerkiksi seuraavien sääntöjen syiden ymmärtäminen vaatii taustatietoja, joten niistä voi tulla sokkosääntöjä. Vaikka puolitushakua voi käyttää listoille, älä tee niin. Älä lisää vektoriin mitään, ei edes sen perään, kesken vektorin selauksen. Jos kokeilet perään lisäystä, kaikki toimii hienosti, mutta älä silti lisää. Sen sijaan poistaminen kesken selauksen on sallittua, kunhan poistettava alkio sijaitsee selaimen senhetkisen kohdan jälkeen. Pakan tapauksessa sekä lisääminen että poistaminen ovat kiellettyjä kesken selauksen, paitsi poistaminen pakan jommasta kummasta päästä, paitsi että selauskohdassa olevaa alkiota ei silloinkaan saa poistaa. Listasta poistaminen ja lisääminen kesken selauksen on sallittua, paitsi että selauskohdassa olevaa alkiota ei saa poistaa.

Kuva 3 valottaa kesken selaamisen tapahtuvaa poistamista koskevien sääntöjen merkitystä. Siinä näytetty ohjelmanpätkä poistaa säiliöstä alkioiden välittömät kopiot. Säiliölle `multiset` se on hyvä, joskaan ei suositeltavin tapa. Listalle on oma palvelu, joka tekee saman asian, mutta yhtä hyvin voi käyttää kuvan koodia. Vektoreille kuvan koodi toimii, mutta on tehoton. Pakoille kuvan koodi on kielletty, ja tuotti eri koeajoissa oikean tuloksen, väärän tuloksen, ohjelman kaatumisen ja ohjelman jäämisen ikuiseen silmukkaan. Kuvan toiminnon saa aikaan myös standardikirjaston funktiolla `unique`, joka siirtää kopiot loppuun, ja poistamalla loppuun siirretyt eri käskyllä. Sitä ei kuitenkaan voi käyttää `multiset`:ille.

Kesken selauksen tapahtuvan lisäämisen ja poistamisen esimerkki havainnollistaa, että osa säännöistä vaikuttaa epäjohdonmukaisilta, jos niitä ajattelee vain säiliön tarjoaman palvelun kautta. Epäjohdonmukaisia sääntöjä on vaikea muistaa. On selvää, että vaikeuksia voi seurata, jos poistetaan juuri se alkio, jonka kohdalla selaus on. Mutta miksi lisäys tai muiden alkioiden poisto on toisinaan kiellettyä? Ja miksi säännöt ovat niin erilaiset eri säiliöille?

Jos asiaa ajattelee säiliöiden toteutuksen kautta, niin säännöt muuttuvat ymmärrettäviksi. Lisääminen ja poistaminen ovat kiellettyjä silloin, kun ne voivat rikkoa niitä tietoja, joilla alkion seuraaja löydetään selattaessa. Sanotaan, että selain on *vanhentunut* tällaisen toiminnon jälkeen. Vektorin tapauksessa tämä tieto on alkion osoite muistissa. Alkion lisääminen saattaa muuttaa jokaisen vanhan alkion osoitteen, koska se aiheuttaa vektorin koko sisällön siirtämisen muistialueesta toiseen silloin, kun vanhasta muistialueesta loppuu tila kesken. Poistamisen yhteydessä muistialuetta ei vaihdeta, mutta poistetun alkion perässä olevia alkioita siirretään aikaisemmaksi täyttämään poistetun alkion jättämä aukko. Siksi poistaminen ei van-

henna selainta, jos ja vain jos se on poistokohdan etupuolella.

Linkitetyn listan alkion seuraaja ja edeltäjä löydetään alkion yhteyteen talletettujen osoittimien avulla. Kun alkio lisätään tai poistetaan, muita alkioita ei tarvitse koskaan siirtää muistissa. Niinpä osoittimet säilyvät voimassa.

Poisto pakan keskeltä edellyttää aukon sulkemista siirtämällä alkioita sen jommalta kummalta puolelta. Standardin kompleksisuusvaatimus 23.2.1.3 5 pakottaa valitsemaan sen puolen, jolla on vähemmän siirrettävää. Minkään selaimen säilyminen voimassa ei siis ole taattua, eli jokainen selain vanhenee. Poisto reunasta ei aiheuta siirtotarvetta. Lisäys reunaan saattaa aiheuttaa sen aputaulukon siirtämisen uuteen muistialueeseen, jonka kautta alkiotaulukot löydetään. Siksi lisäys reunaan vanhentaa kaikki selaimet, mutta poisto reunasta vain ne, jotka kohdistuvat poistettavaan alkioon.

Vastaava pätee toimintojen nopeuksille. Puolitushaun nopeuseroista eri säiliöillä ei voi saada minkäänlaista vihjettä puolitushaun tuottaman palvelun kautta, sillä se ei riipu säiliöstä. Palvelun näkökulmasta nopeusero on asia, joka täytyy vain muistaa. Jos sen sijaan ajatellaan puolitushaun toimintaperiaatetta, nopeusero saa välittömästi selityksen. Tehokas puolitushaku edellyttää kykyä etsiä alkio nopeasti numeerisesti ilmoitetusta paikasta. Vektori ja pakka tarjoavat sellaisen palvelun, mutta lista ei.

Loitsujen osaamisen tarve on sikäli ikävää, että jos loitsun puuttuminen aiheuttaa käännösvirheen tai virheen ensimmäisessä koeajossa, niin syyn selvittäminen ja oikean loitsun löytäminen voi joskus olla melkoista hapuilua ja kestää kauan. Se myös nostaa kynnystä ottaa säiliöitä ja niihin liittyviä algoritmeja käyttöön. Se ei kuitenkaan johda epäluotettaviin ohjelmiin.

Sokkosäännöt ovat tässä suhteessa kiusallisempia, sillä niiden rikkomisesta ei välttämättä saa ajoissa palautetta. Aluksi kaikki näyttää hyvältä, mutta isoilla syöteaineistoilla ohjelma toimii oudon hitaasti, tai kaatuu joskus harvoin omituisesti. Onneksi standardikirjaston säiliöiden toteutuksessa käytettävien tietorakenteiden tunteminen muuttaa monia sokkosääntöjä järjellä pääteltäviksi asioiksi. Silti loitsujen ja sokkosääntöjen osaamisen tai toteutuksen ymmärtämisen tarve pienentää sitä hyötyä, jota kirjastojen käyttöönotolla tavoitellaan: tehokkaiden ja luotettavien ohjelmien tekemistä helposti.

3 Käytännön kokemus

3.1 Ongelma ilmenee

Eräänä marraskuisena perjantaina löysin sähköpostilaatikostani seuraavan viestin:

> Moi,
>
> Huomasin tässä yötä myöten [tietorakenteiden jatkokurssin] merkeissä koodaillessani (mielestäni) varsin erikoisen ominaisuuden standardikirjaston lista-rakenteesta ja ajattelin että saattaisit olla törmännyt samaan… tai jos et niin ehkä kiinnostaisi siltikin. Huomasin nimittäin että listan funktiot `front()` ja/tai `pop_front()` ovat nauretavan hitaita. En ole varma kumpi niistä on se varsinainen paholainen mutta `pop`pia toki olettaisin. Mutta linkitetyssä listassahan molempien pitäisi olla nopeita operaa-

> tioita. Tässä kuitenkin hieman tilastoa ...
>
> Huomattava hyppäys kaarien tuhoamisessa, hmmm? Kun ne kerta saadaan luotua ja tulostettua saman 12 sekunnin aikana kun muukin käsittely tapahtuu, miten voi kestää yli minuutti niiden tuhoamisessa? Kaaret olivat listassa ja tuhoaminen näytti tältä (`pop`in kutsuminen oli tarpeeton, mutta järjen mukaan sen ei pitäisi kovin kauaa kestää):

```
while( kaaret.size() != 0 ) {
   delete kaaret.front();
   kaaret.pop_front(); }
```

(Toiminnolla `delete` vapautetaan dynaamisesti varattu muistinpala. `Front` palauttaa viitteen listan ensimmäiseen alkioon.) Viesti jatkui kuvailulla siitä, kuinka hän oli toisenlaisella ratkaisulla saanut kaarten tuhoamisen puristettua sekunnin osaan ja koko ohjelman suoritusajan kahteen sekuntiin, sekä standardikirjaston listan huonouden päivittelyllä.

Vastasin seuraavana maanantaina.

> C++-standardi lupaa, että `front()` ja `pop_front()` ovat vakioaikaisia listoille (ja kaikille muillekin, joille ne ovat tarjolla). Tästä ja muista syistä en helposti usko, että C++:n lista olisi niin huono. Hitauden todellinen syy on muualla. ...

Asiaa kunnolla miettimättä tarjosin hitaudelle kahta vaihtoehtoista selitystä. Ensimmäinen oli, että aika kuluu muistin vapautuksessa toiminnolla `delete`. Minuun on iskostettu vanhinten perinnäissääntö, että vältä `delete`:n käyttöä, sillä se saattaa olla hidas. Monen kokoisten vapautettujen muistilohkojen hallintahan on tunnetusti vaikeaa. Lisäksi on moneen kertaan havaittu, että yleisimmät syöttö- ja tulostustoiminnot ovat huomattavan hitaita ainakin sillä kääntäjällä, jolla opiskelija oli kokeensa tehnyt. Siksi pidin juuri ja juuri mahdollisena, että `delete` olisi noin huono. Omituiselta se kyllä tuntui.

Huvittavaa asiassa on, että kaarten tuhoaminen oli tarpeetonta. Kaarten tuhoamisvaihe oli ohjelmassa vastauksen tulostamisen jälkeen, ja sen jälkeen oli ainoastaan muiden tietorakenteiden tuhoamista. Muistin vapauttamisesta sen jälkeen kun kaikki muu on tehty ei ole hyötyä, sillä ohjelman lopetettua ajonaikainen järjestelmä vapauttaa joka tapauksessa kaiken ohjelmalle varatun muistin. Muistin vapauttamistoiminto on ohjelmointikielissä sitä varten, että voidaan kierrättää muistia, eli jos myöhemmin tarvitaan lisää muistia toiseen tarkoitukseen, voidaan käyttää aikaisemmin vapautettua muistia.

Jos muistia jää vahingossa vapauttamatta, sanotaan, että *muistia vuotaa*. Jos ohjelman on tarkoitus toimia pitkäaikaisesti, kuten kännykän tai www-palvelimen on, niin muistivuodot voivat aiheuttaa muistin loppumisen kesken ja saman — tai jonkin toisen! — ohjelman kaatumisen. Tällaiset virheet ovat kiusallisia ja vaikeita jäljittää. Siksi niihin on kiinnitetty erityistä huomiota Tampereen teknillisen yliopiston opetuksessa [4]. Eräs keino on ollut, että alkeiskurssien harjoitustöiden pitää lopuksi yrittää vapauttaa kaikki varaamansa muisti, ja kurssilla käytössä oleva kääntäjä lisää ohjelman loppuun tarkastuksen, joka ilmoittaa, jäikö muistia vapauttamatta. Kenties opiskelijalle oli jäänyt vanhinten perinnäissäännöksi, että ohjelman pitää aina lopuksi vapauttaa kaikki varaamansa muisti, tai kenties hän halusi tietoisesti jatkaa käytäntöä kehittyäkseen ohjelmoijana.

Toinen ehdotukseni oli, että koska tietue poistetaan sen ollessa vielä listassa, `pop_front` sekoaa jotenkin. Ehdotus oli typerä. Opiskelijan antamassa ohjelmanpätkässä oleva `delete` ei poista listan tietuetta, vaan muualla sijaitsevan tietueen, jonka osoite on listan tietueessa. Listan sisäiselle rakenteelle ei tapahdu mitään, joten `pop_front` ei voi seota tästä syystä.

Lopuksi totesin

> Mitä yleisemmin tulee C++-standardikirjaston säiliöiden tehokkuuteen, niin tietty terve epäluulo on paikallaan. Lukaisepa [linkki].

Linkki vei julkaisuun [7] Tietojenkäsittelytieteen seuran www-sivulla. Julkaisussa oli todettu, että `map` vie yllättävän paljon muistia, ja analysoitu, mistä se johtuu.

Opiskelijalta tuli samana iltapäivänä vastaus, jossa hän ihastuttavan diplomaattisesti selitti, että `pop_front` ei voi seota. Se sai minut ensimmäisen kerran miettimään asiaa kunnolla. Noin tunnin kuluttua vastasin hänelle:

> Nyt vasta huomaan, että listassasi on osoittimia eikä alkioita. Se, mitä kirjoitin, on siis soopaa.
>
> Kokeilin tällaista:
>
> …
>
> ```
> while( kaaret.size() != 0 ){
>   kaaret.pop_front();
> }
> ```
>
> …
>
> Ei yhtään `delete`ä. “Lista selattu” ja sitä edeltävät tulostuvat silmänräpäyksessä, mutta “Lista purettu” antaa odottaa itseään. Siis `pop_front` *on* hidas.
>
> Selityksenä saattaa silti olla muistin vapauttamisen hitaus, sillä `pop_front`issa lienee sisällä piilevä `delete`. Mutta siitä huolimatta tulos on huolestuttava. Kiitos, kun toit sen tietooni.
>
> `pop_back` oli rannekellomittauksessani jopa hieman hitaampi, mutta se voi selittyä kohinalla. Sen sijaan seuraava on vaikeampi selittää:
>
> 40 000 kierrosta 11,5 s
> 80 000 kierrosta noin 165 s

Loppuhuomautuksellani tarkoitin, että ajan kulutus näyttää riippuvan aineiston koosta dramaattisesti, paljon voimakkaammin kuin lineaarisesti. Hitauden syyksi ei siis riitä, että jokin yksittäinen toiminto — vaikka juuri `delete` — vie kauan aikaa, vaan sen ajan kulutuksen *yhdelle* listan alkiolle täytyy riippua siitä, paljonko listassa on (tai on ollut) alkioita *kaikkiaan*.

3.2 Ongelma syvenee

Jatkoin asian tutkimista seuraavana aamuna. Käytin `pop_front`:in sijaan `pop_back`:iä. Selvitin koeohjelman avulla, että listan rakentaminen ja purkaminen veivät taulukossa 1 mainitut ajat. Samanlainen taulukko koostui nollista muutaman harvan ykkösen säestämänä, kun listan sijasta käytettiin säiliötä `set` taikka taulukollista osoittimia, joiden päähän luodaan ja päästä vapautetaan tietue.

Toivoin, että saisin jotain vihjettä ongelman syystä, jos selvittäisin ajan kulutuksen kertaluokan syötteen koon funktiona. Se onnistui helposti funktiolaskimella ottamalla taulukon 1 ensimmäisen ja viimeisen sarakkeen luvuista logaritmit ja tekemällä niille regressioanalyysi. Aineisto

Taulukko 1: Mittaustuloksia kannettavassa tietokoneessa sekunteina.

	koe 1		koe 2		koe 3		keskiarvo
n	luonti	purku	luonti	purku	luonti	purku	purku
10 000	0	0	0	1	0	0	0
20 000	0	8	0	9	0	8	8
30 000	0	30	0	30	0	30	30
40 000	0	62	0	62	0	61	62
50 000	0	103	0	103	0	103	103
60 000	0	153	0	152	0	153	153
70 000	0	212	0	212	0	212	212

noudattaa likimain lakia

$$\ln t \;=\; 0{,}43 + 2{,}6 \ln \frac{n}{10\,000}$$

eli

$$t \;=\; 1{,}5\Big(\frac{n}{10\,000}\Big)^{2{,}6} \;.$$

Riippuvuus näyttää siis pahemmalta kuin neliöllinen, mutta lievemmältä kuin kuutiollinen. Se on omituinen laki. On tosin otettava huomioon, että pienissä mittaustuloksissa on vain yksi merkitsevä numero, joten tulos ei ole kovin tarkka. Tein myöhemmin uuden laskelman toisella tietokoneella saaduista mittaustuloksista. Sain varsin tarkasti kuutiollisen riippuvuuden, mutta sitäkään ei voi pitää varmana tuloksena, sillä viimeisen havainnon poistaminen pienensi eksponenttia huomattavasti.

Muistin, että säiliöiden toteutustavasta johtuen niiden lähdekoodi on jossakin kannettavan tietokoneeni uumenissa. Olin joskus aiemmin etsinyt oikean hakemiston. Se oli työlästä silloin ja oli työlästä nyt. Pääsin kuitenkin lukemaan listan toteutusta. Siellä ollut kommentti lupasi, että `pop_back` toimii vakioajassa.

Lähdekoodin seuraaminen oli työlästä, koska toiminnot oli pilkottu moneen aliohjelmaan, jotka sijaitsivat hujan hajan monessa suuressa tiedostossa. Listojen toteutus oli jaettu kahteen tiedostoon, joissa oli yhteensä 1543 riviä. `Pop_back` näytti tältä:

```
void
pop_back()
{
  iterator __tmp = end();
  this->erase(--__tmp);
}
```

Toisin sanoen, `pop_back` ei tee muuta kuin kutsuu käyttäjällekin tarjolla olevaa yleistä poistotoimintoa listan viimeiselle alkiolle.

`Erase` oli eri tiedostossa. Niitä oli itse asiassa kaksi, mutta oikea löytyi, kun hieman etsi. Se on näytetty täydellisenä kommentteineen (joita ei siis ole) kuvassa 4. Tuttu kaksisuuntaisesta linkitetystä listasta pois linkittäminen on siitä helposti tunnistettavissa. `Static_cast` ja lopussa oleva `iterator` ovat tyyppimuunnoksia, joiden ei luulisi kauaa aikaa vievän. Miksi niitä tarvitaan on asiallinen kysymys, mutta en paneutunut siihen, koska arvelin, että se tuskin veisi hitauden syyn selvittämistä eteenpäin. Jäljelle jäivät funktioiden `_Destroy` ja `_M_put_node` kutsut.

`_Destroy` ei löytynyt listan toteutuksesta, joten sen jäljittäminen kävi niin työlääksi, että päätin kokeilla ensin muita keinoja löytää hitauden syy. Jälkikäteen ymmärsin, että `_Destroy` huolehtii listaan

```
template<typename _Tp, typename _Alloc>
  typename list<_Tp,_Alloc>::iterator
  list<_Tp,_Alloc>::
  erase(iterator __position)
  {
    _List_node_base* __next_node = __position._M_node->_M_next;
    _List_node_base* __prev_node = __position._M_node->_M_prev;
    _Node* __n = static_cast<_Node*>(__position._M_node);
    __prev_node->_M_next = __next_node;
    __next_node->_M_prev = __prev_node;
    _Destroy(&__n->_M_data);
    _M_put_node(__n);
    return iterator(static_cast<_Node*>(__next_node));
  }
```

Kuva 4: Gnu C++-kääntäjän lista-säiliön `erase(paikka)`. © 2001, 2002 Free Software Foundation, Inc.

talletetun alkion purkamisesta. Kuten alaluvussa 2.4 kerrottiin, lista ei talleta sille annettua alkiota sellaisenaan, vaan ottaa siitä kopion ja tallettaa sen. Jos alkio on kokonaislukutyyppiä, niin kopioimiseen ei liity mitään erityistä. Alkio voi kuitenkin olla myös oliotyyppiä. Oliotyypille voi määritellä *rakentajan* (*constructor*), joka suoritetaan aina, kun olio luodaan. Vastaavasti voi määritellä *purkajan* (*destructor*), joka suoritetaan aina, kun olio lakkaa olemasta. Normaalisti purkajaa kutsutaan automaattisesti kun oliolle varattu muisti vapautetaan. Valitettavasti automatiikkaan liittyy tehohäviötä. `Erase` välttää sen kutsumalla purkajaa itse. `_Destroy` tekee sen.

`_M_put_node` löytyi listan toteutuksesta. Se ei kuitenkaan sisältänyt muuta kuin kutsun aliohjelmaan, joka ei sijainnut listan toteutuksessa. Nimen perusteella se liittyi muistin hallintaan. Arvelin, että sen etsiminen ja tulkitseminen olisi työlästä. Toiminnon tehtävä kävi kuitenkin selväksi: se vapauttaa listan solmulle varatun muistin.

Tilanne tuntui käsittämättömältä. Listasta itsestään ei ollut löytynyt selitystä. Tutkimatta jättämäni aliohjelmat sijaitsivat listan toteutuksen ulkopuolella, joten oli vaikea keksiä, miten niiden suoritusajat voisivat riippua listan alkioiden määrästä. Listan esitystavan muistissa pitäisi olla niille tuntematon. Niiden ei pitäisi edes tietää, että niitä kutsutaan tällä kertaa listasta. Toki voi spekuloida, että niissä on virhe, joka kasvattaa suoritukseen menevää aikaa jo tehtyjen suoritusten määrän funktiona. Silloin suoritusaika kasvaisi listan purkamisen edetessä, ja kokonaissuoritusaika voisi olla mittausten mukainen. Mutta miksi ilmiö ei toistunut `set`:illä? Se tuntui todella omituiselta, mutta parempaakaan selitystä ei ollut tarjolla.

Etsin hetken netistä, olisiko Gnu C++-kääntäjän käyttämän kirjaston toteutuksessa yleisesti tunnettu, hitauden aiheuttava virhe. Mitään ei löytynyt. Olin tehnyt mittauksia sekä työpaikan verkkoympäristössä että kannettavalla tietokoneellani saaden samankaltaiset tulokset. Kannettavassa tietokoneessa oli kääntäjästä monta vuotta vanha versio, mutta työpaikalla

tuore. Hitausilmiö oli niin dramaattinen, että jos sen syynä todella oli virhe kirjastossa, niin oli merkillistä, että sitä ei ollut huomattu ja korjattu kääntäjän vanhan ja uuden version välillä kuluneina vuosina. Virhe kirjaston toteutuksessa tuntui siis väärältä selitykseltä.

Illalla asensin kotonani olevaan pöytäkoneeseen Microsoft Visual C++ 2008 Express Edition -kääntäjän. Jo ensimmäinen koe paljasti, että kun ohjelma käännettiin sillä, niin hitausilmiö katosi. Se oli vahva viittaus siihen, että Gnu C++-kirjastossa olisi sittenkin virhe. Tämä teki tilanteesta entistä vaikeamman ymmärtää. Tiedossani olevat tosiseikat eivät tuntuneet millään sopivan yhteen.

3.3 Ongelma ratkeaa

Samana iltana tai seuraavana aamuna hoksasin uudistaa nopeusmittaukset siten, että alkiot sijoitettiin yhden listan sijasta kahteen listaan, joista kummankin pituus oli puolet aikaisemmissa kokeissa käytetystä. Esimerkiksi 40 000 alkion tapauksessa ajan kulutus ei ollut uudessa kokeessa suunnilleen sama kuin aikaisemmassa kokeessa, vaan suunnilleen kaksi kertaa aikaisemman kokeen ajan kulutus 20 000 alkion listalla. Tulos oli sama jokaisella kokeillulla alkioiden määrällä. Uusi koe osoitti siis vakuuttavasti, että hidastuminen oli sidoksissa listojen pituuksiin eikä aliohjelmien `pop_back`, `_Destroy` ja `_M_put_node` kutsujen määrään. Tämä viittasi vahvasti siihen, että syy ei ollut piilossa aliohjelmien `_Destroy` ja `_M_put_node` sisällä.

Mutta missä syy sitten oli? Kuvan 4 koodissa ei näyttänyt olevan muitakaan mahdollisia paikkoja.

Oli pakko päästä lisäämään listan toteutukseen ajanmittauskomentoja ja kommentoimaan siitä rivejä pois kokeiden jatkamiseksi. Mieleeni muistui, että C++-ohjelman kääntäminen alkaa esikäsittelyvaiheella, joka yhdistää eri tiedostoista peräisin olevat ohjelman osat yhdeksi ja muun muassa avaa makrot. Pelkän esikäsittelijän ajamiseksi on olemassa erillinen komento `cpp`. Kenties sen lopputulos on lukukelpoinen, muokattavissa ja edelleen käännettävissä.

Tuumasta toimeen. Esikääntäjä antoi yli 30 000 riviä pitkän listauksen tutkittavakseni. Riveistä suurin osa aiheutui loitsusta `#include<iostream>`, jolla otetaan käyttöön C++:n syöttö- ja tulostustoiminnot. Onneksi mielenkiintoiset osat oli helppo löytää etsimällä toimintojen nimien perusteella. Listauksen kääntäminen edelleen suorituskelpoiseksi koodiksi alkoi sujua, kun kääntäjän käsikirjasta löytyi, että tiedoston nimen lopuksi pitää asettaa `.ii` eikä `.cc`. Kommentoin kokeeksi eri rivejä pois kuvan 4 koodista, ja sain muun muassa ohjelman juuttumaan ikuiseen silmukkaan. Kommenteilla oli siis vaikututusta, joten olin löytänyt oikean aliohjelman. Jälkikäteen varmistin sen vielä lisäämällä testitulostuksen.

Kun kommentoin aliohjelmien `_Destroy` ja `_M_put_node` kutsut pois ja kokeilin, niin hitausilmiö säilyi.

Sitten aloin lisätä ajanmittaustoimintoja jäljittääkseni, missä aika oikein kului. Mittaaminen perustui kellon katsomiseen C-kielen kirjastoista löytyvällä aliohjelmalla `time` ennen ja jälkeen tutkittavan ohjelman osan sekä aikaeron summaamiseen keräilymuuttujaan. Käyttämäni kello antoi ajan vain sekunnin resoluutiolla, mutta järkeilin, että suurin osa sekuntien vaihtumisista osuu siihen ohjelman osaan, jossa ohjelma viettää suurimman osan ajasta. Mutta vaikka otin kuvan 4 koodin haarukkaan niin kattavasti kuin mahdollista, en saanut minuutteja kestävästä suorituksesta kiinni kuin muutaman sekunnin.

```
size_type
size() const { return distance(begin(), end()); }
```

Kuva 5: Funktion `size` toteutus.

Olin aiemmin saanut eräältä tutkimusapulaiselta tietää toisesta, vaikeammin käytettävästä, mikrosekunnin resoluutioon yltävästä tavasta katsoa kelloa. Sitä on vaikea löytää ellei tiedä sen olemassaolosta, sillä se sijaitsee eri kirjastossa kuin `time`, ja sen nimi `gettimeofday` antaa väärän kuvan sen luonteesta. Muunsin ajan mittauksen käyttämään sitä. Pettymykseksenі sekään ei saanut kiinni kuin murto-osan ajasta.

Tuloksessa ei tuntunut olevan järkeä. Päätin ottaa ajanmittauksen piiriin kaiken, minkä vaan pystyn, mukaan lukien sellaisetkin toiminnot, joissa ei pitäisi kulua kuin silmänräpäyksen murto-osa. Katsoin uudelleen silmukkaa, jonka olin mitannut käyttävän kohtuuttomasti aikaa:

```
while( kaaret.size() != 0 ){
  kaaret.pop_back();
}
```

Täytyi siis lisätä ajan mittaus funktioon `size`. Sitä varten etsin sen toteutuksen. Löysin sen, mikä näkyy kuvassa 5. Silloin totuus iski mieleeni kuin salama.

3.4 Mistä oli kyse?

Kuvasta 5 näkyy, että `size` hakee lopputuloksen kysymällä sitä `distance`-funktiolta, jolle se antaa argumenteiksi listan ensimmäisen alkion paikan ja viimeisen alkion jäljessä tulevan paikan. Koko löydetään siis laskemalla etäisyys listan ensimmäisestä alkiosta listan loppuun. Koska listoja ei voi indeksoida, `distance` toimii listoille selaamalla ensimmäiseksi annetusta paikasta toisena annettuun paikkaan ja laskemalla, montako askelta tarvitaan. Toisin sanoen, `size` tuottaa listan koon selaamalla listan läpi ja laskemalla sen alkiot.

Listan selaamiseen menee listan kokoon verrannollinen aika. Koska ongelmallinen silmukka kutsuu `size`-funktiota aina ennen alkion poistamista ja lopuksi tyhjälle listalle, ensimmäinen kutsu selaa n alkiota, toinen $n-1$, seuraava $n-2$ ja niin edelleen nollaan asti, missä n on listan alkioiden määrä ennen purkamista. Yhteensä tämä tekee $\frac{1}{2}n(n+1)$, mikä on neliöllinen määrä. Kun alkioita on 10 000, tuottaa kaava noin 50 000 000. Hitaus oli saanut selityksen. Varmistuin selityksen pätevyydestä korvaamalla testin `size() != 0` samaa tarkoittavalla mutta eri tavalla toimivalla testillä `!empty()`, jolloin hitausilmiö katosi täydellisesti.

Täsmällistä syytä sille, että ajan kulutuksen kasvu oli mittausten mukaan neliöllistä jyrkempi, en ole selvittänyt. Ilmiö ei kuitenkaan ole odottamaton. Ensiksi, mittaus ei ollut erityisen tarkka. Pienetkin muutokset ajoissa muuttavat lain eksponenttia huomattavasti. Toiseksi, nykyaikaisen tietokoneen muisti koostuu useista kerroksista, joista prosessoria lähinnä olevassa on pieni määrä nopeaa muistia, ja mitä kauemmas prosessorista mennään, sitä enemmän ja sitä hitaampaa muistia kerroksessa on. Mitä isompi syöte, sitä suuremman osan laskentaa ohjelma joutuu tekemään ylempien kerrosten hitaan muistin avulla. Tämä jyrkentää suoritusajan kasvua.

Koska Microsoftin kääntäjällä käännettynä koeohjelma toimi nopeasti, on pakko päätellä, että siellä `size` oli toteutettu toisella tavalla. Välittömästi tulee mieleen, että listan kantatietueen yh-

teyteen on talletettu tieto listan koosta eli siellä olevien alkioiden määrästä. On helppoa ja nopeaa kasvattaa tai vähentää kokotietoa yhdellä aina kun listaan lisätään tai sieltä poistetaan alkio. Kun listan kokoa kysytään, tieto saadaan välittömästi katsomalla kokotietomuuttujasta.

Näin dramaattinen ero näin keskeisen toiminnon suoritusajassa herättää kysymyksen: kumpi toimi oikein, Gnu-hankkeen vai Microsoftin kääntäjä? Mitä standardi sanoo? Standardin taulukossa 65 palstassa “complexity” lukee yleensä “compile time”, “constant” tai “linear”, mutta toiminnon `size` ja muutaman muun kohdalla lukee “(Note A)”. Taulukon alla lukee “Those entries marked ‘(Note A)’ should have constant complexity”. Tämä pitänee tulkita siten, että standardi ei suoranaisesti vaadi vakioaikaisuutta, mutta suosittelee sitä.

Jälkiviisaana oli helppo huomata, että kirja [3] korostaa toistuvasti, että säiliön tyhjyyden testaamiseen `empty` on suositeltavampi kuin `size`, koska `size` saattaa olla hitaampi.

Tässä vaiheessa olisi helppo ottaa näkökanta, että koska kokotiedon ylläpitäminen lisää listan muistinkulutusta ja hidastaa listan toimintoja vain hieman, olisi järkevää vaatia standardissa, että `size` toimii vakioajassa. Microsoftin kirjasto olisi siis tältä osin parempi kuin Gnu-hankkeen. Keskustelin kokemuksestani C++-asiantuntija Matti Rintalan kanssa. Hän huomautti, että listoille on toiminto `splice`, jolla voi leikata listasta osan ja liittää sen johonkin kohtaan toista (tai samaa) listaa. Jos listan kokoa ei ylläpidetä erikseen, toiminnon `splice` toteuttamiseksi riittää linkittää siirrettävä listan osa kummastakin päästään pois paikasta, jossa se on, ja paikkaan, johon se siirretään. Se onnistuu vakioajassa. Jos kuitenkin kokoa ylläpidetään ja siirto tapahtuu listasta toiseen, on siirrettävän osan koko saatava selville, jotta kummankin listan koko voidaan päivittää. Siirrettävä osa (tai poiston jälkeen jäänyt lista) on siis selattava läpi, eikä `splice` voi toimia vakioajassa.

On siis valittava: `size` vai `splice`. Kumman tahansa niistä saa toimimaan vakioajassa, mutta ei molempia. Toisesta tulee lineaariaikainen.

Kirja [3, s. 171] lupaa, että `splice` toimii vakioajassa. Se on siis Gnu:n kannalla Microsoftia vastaan. Standardi sanoo, etttä `splice` toimii vakioajassa silloin, kun siirto tapahtuu saman listan sisällä, ja muutoin lineaarisessa ajassa. Riippuen siitä miten edellä mainittu “(Note A)” tulkitaan, standardi joko sallii molemmat tai on Microsoftin puolella Gnu:ta vastaan. Rintalan mukaan standardin uuden version yhteydessä käydyssä keskustelussa on väitetty, että ohjelmoijat joka tapauksessa olettavat toiminnon `size` olevan vakioaikainen. Edellä kuvattujen tapahtumien jälkeen en voi olla tästä eri mieltä!

En kuitenkaan yhdy johtopäätökseen, että `size`:n tulee olla vakioaikainen. Tyhjyyden testaamiseen on vakioaikainen `empty`. Koon selvittäminen selaamalla ei harvoin tehtynä vie kohtuuttomasti aikaa verrattuna siihen, mitä on jo kulunut listan rakentamiseen. Perustelu, että `size`:n hitaus on ohjelmoijalle ansa, ontuu sikäli, että säiliöjärjestelmässä on monia muita yhtä ikäviä ansoja. Ohjelmoijan on siis joka tapauksessa hankittava paljon tietoa voidakseen käyttää C++:n säiliöitä turvallisesti ja tehokkaasti. `Size`-ansan korjaaminen parantaisi tilannetta kovin vähän. Vakioaikainen `splice` on ainutlaatuinen palvelu, jonka vain listat pystyvät tarjoamaan. Vaikka sitä tarvittaisiin vain harvoin, sen menettäminen on sääli.

Standardin valinta ei tosin tarkoita, että vakioaikainen `splice` on aina ehdotto-

masti menetetty. Käyttäjä voi tilanteesta riippuen saada sen takaisin yhdistämällä erilliset listansa yhdeksi suureksi listaksi, jossa osalistojen rajat osoitetaan selaimilla. Monella valmiilla algoritmilla voidaan käsitellä listan osaa yhtä kätevästi kuin kokonaista listaa. Valitettavasti tämä ei päde esimerkiksi järjestämiseen. Valmiit järjestämisalgoritmit edellyttävät kykyä indeksoida, joten niitä ei voi käyttää listoille. Sen vuoksi listoille on oma, säiliökohtainen järjestämistoiminto. Sitä ei voi kuitenkaan kutsua listan osalle. Saadaanko vakioaikainen `splice` tällä tavalla takaisin riippuu siis siitä, mitä osalistoille tarvitsee tehdä.

Ehkä listoille pitäisi olla kaksi eri säiliölajia, joista toinen ylläpitää ja toinen ei ylläpidä kokotietoa. Tälle tielle lähdettäessä on kuitenkin vaarana, että löytyy lukuisia muitakin perusteita jakaa säiliölajeja kahtia, jolloin niiden määrä kasvaa hallitsemattomaksi.

4 Mitä tästä opin?

Tapahtumat antoivat minulle monta opetusta.

Ensiksi, luotin liikaa opiskelijan viestin mukana tulleisiin tietoihin. Koska hän epäili vain silmukan vartalossa olevia lauseita, en minäkään kiinnittänyt huomiota silmukan ehtoon ennen kuin vartalo oli vakuuttavasti paljastunut viattomaksi. Oikea vastaus olikin kysymyksessä esitetyn rajauksen ulkopuolella.

Toisaalta periaatteen “epäile aina kaikkea” tinkimätön noudattaminen lisää jokaisen yksittäisen ongelmanratkaisun työmäärää. Voi olla, että yrittämällä välttää liiallista vainoharhaisuutta olin jo etukäteen säästänyt sen lisätyön, jonka nyt jouduin tekemään huolimattomuuteni vuoksi. Huomion kohdistaminen tasattuun suoritusaikaan on paikallaan tässäkin asiassa.

Toiseksi, C++-standardi ei ollutkaan sellainen järkkymättömän luotettava ja yksityiskohtainen totuuden lähde kuin olin luullut. Siitä paljastui pikkuvirheitä sekä virheelliseltä vaikuttavia yksityiskohtia, joita en pysty osoittamaan virheiksi. Yksityiskohtia koskeviin kysymyksiin oli usein vaikea löytää vastauksia, kuten kuuluuko `bitset` samaan säiliöiden ryhmään `set`:in kanssa, vai muodostaako se yksinään oman ryhmänsä. Kompleksisuustiedot jäivät usein puuttumaan. Joskus vastauksen saamiseksi oli tehtävä rohkeita tulkintoja. Gnu-hanke ja Microsoft olivat vastakkaista mieltä siitä, kumman tulee olla nopea, `size`:n vai `splice`:n. Standardi on niin epäselvä, että kummankaan ei voi väittää olevan väärässä, joskin Microsoftin valinta on varmemmalla pohjalla.

Standardin kohta 23.1 2 sanoo, että kaikki kompleksisuusvaatimukset koskevat talletettuihin olioihin kohdistuvien toimintojen määrää. Sitä havainnollistetaan esimerkillä, että jos kopioidaan vektorillinen kokonaislukuvektoreita, niin kopiointi on lineaarista, vaikka jokaisen alkiona olevan kokonaislukuvektorin kopiointi vie itsessäänkin lineaarisen ajan. Näin lukijaa muistutetaan siitä tosiasiasta, että ajan kulutus riippuu myös seikoista, jotka eivät ole säiliön vaan sen käyttäjän vallassa.

Olen keskustellut Rintalan kanssa muutaman kerran, tarkoittaako tämä lisäksi sitä, että niiden toimenpiteiden kustannuksia ei oteta lainkaan huomioon, jotka eivät kohdistu säiliöön talletettuihin alkioihin. Tämä liittyy edellä mainittuun kysymykseen siitä, tekeekö standardi virheen luvatessaan pakan päihin lisäämisen olevan vakioaikaista. Pakan todennäköisessä toteutuksessa on taulukollinen osoittimia taulukoihin, joissa alkiot ovat. Lisääminen edellyttää toisinaan osoitin-

taulukon koko sisällön siirtämistä uuteen muistialueeseen, koska vanha täyttyi. Se vie alkioiden määrän suhteen lineaarisesti aikaa. Tulkinnan mukaan tätä aikaa ei kuitenkaan otettaisi huomioon lisäämisen kompleksisuudessa, koska osoitintaulukon siirrossa ei kosketa talletettuihin alkioihin. Tällöin pakan päihin lisääminen todella olisi vakioaikaista siinä merkityksessä, minkä standardi sanalle antaa, mutta merkitys poikkeaisi olennaisesti tavallisesti käytetystä merkityksestä.

C++-standardissa on valtava määrä monimutkaisia yksityiskohtia, joten ei ole noloa, että niissä on virheitä. Standardin seuraavassa versiossa tullaan korjaamaan monta virhettä. Standardointikomitea on tehnyt korkealaatuista työtä. Olisi kohtuutonta vaatia täydellisyyttä. Käyttäjän on kuitenkin hyvä tiedostaa, että jos jokin seikka vaikuttaa kummalliselta, niin sekin mahdollisuus on olemassa, että standardissa on siinä kohdassa virhe.

Kolmanneksi, oli hyödyllistä nähdä esimerkki siitä, miten ennalta arvaamattomia valintoja standardointityössä joudutaan tekemään. Se vähensi intoani moittia standardin yksityiskohtia typeriksi. Vakioaikainen `size` on helppo toteuttaa, ja niin on myös vakioaikainen `splice`. Niiden välillä on ristiriita, mutta se paljastuu vasta, kun ne yritetään sisällyttää samaan kokonaisuuteen. Silloin pitää tehdä valinta tietoisena siitä, että asialla on käyttäjille suuri merkitys, mutta ilman tietoa siitä, kumpi vaihtoehto on heille parempi.

Neljänneksi, tässä kirjoituksessa kerrotut seikat osoittavat minusta vakuuttavasti, että vielä ei osata tehdä laajaa, äärimmäiseen tehokkuuteen pyrkivää tietorakenne- ja algoritmiabstraktiota, joka ei *vuoda*. Abstraktion vuotaminen tarkoittaa, että jonkin tärkeän seikan hallitsemiseksi on ymmärrettävä abstraktion takana olevaa toteutusta. Sitä on käsitelty kirjoituksessa [5].

Tietojenkäsittelyssä on abstraktioita, jotka eivät vuoda, tai ainakin vuotavat niin vähän, että siitä ei tarvitse välittää. Tietokoneen rakenne rekisterien tasolla ilmaistuna on esimerkki. Ohjelmoijien ei tarvitse tietää, ovatko mikropiirit toteutetut CMOS-teknologialla vai jollakin muulla, puhumattakaan siitä, että heidän tarvitsisi ymmärtää niitä kvanttimekaniikan lainalaisuuksia, joihin mikropiirissä olevan transistorin toiminta perustuu. Ohjelmoijan ei yleensä tarvitse tietää mitään edes siitä, miten lausekkeen laskeminen on toteutettu pinon avulla.

Näyttää kuitenkin siltä, että yritys hallita C++:n säiliöt abstraktioina, tuntematta lainkaan sisäistä toteutusta, on tuhoon tuomittu. Ohjelmoijan on mahdotonta pitää mielessään kaikkia säiliöiden turvallisen ja tehokkaan käytön edellyttämiä asioita, ellei hän tunne toteutusta pääpiirteissään. Tämän kirjoituksen pääesimerkki liittyi ainoastaan ohjelmien nopeuteen. Kuitenkin, kuten edellä mainittiin, esimerkiksi selainten vanhenemista koskevat säännöt ovat sekavia, ja niiden huomiotta jättäminen voi johtaa kiusallisiin virhetoimintoihin.

C++:n säiliöiden muodostama kokonaisuus on yritetty tehdä helpommin hallittavaksi jakamalla säiliöt ryhmiksi. Valitettavasti niiden asioiden määrä, jotka pätevät ryhmän jokaiselle jäsenelle, on pieni. Kaikki indeksoitavat säiliöt ovat niin sanottuja *ketjusäiliöitä* (*sequence*), mutta listaa ei voi indeksoida, vaikka sekin on ketjusäiliö. Kaikkien muiden ketjusäiliöiden alkuun voidaan lisätä tehokkaasti, mutta vektorin ei. Listasta ja vektorista saa poistaa selauskohdan jäljessä sijaitsevan alkion kesken selauksen, mutta pakasta ei. Hajautustaulu sopisi muuten loistavasti samaan ryhmään kuin `map`, mutta, kuten edellä todettiin, se ei ylläpidä alkioi-

den järjestystä. Poikkeusten vuoksi asioita ei voi tallettaa mieleensä ryhmittäin, joten ryhmittely auttaa muistamista aika rajallisesti. Toisaalta se myös lisää muistamisen tarvetta, koska on muistettava ryhmän nimi ja kokoava ominaisuus.

Abstraktiota voi muuttaa vuotamattomuuden suuntaan vähentämällä lupauksia, jotka abstraktio takaa. Esimerkiksi selaimia koskevat säännöt saisi helposti muistettaviksi muuttamalla ne muotoon, että selain vanhenee aina kun säiliöön lisätään tai sieltä poistetaan jotain, tai (niiden säiliöiden tapauksessa, jotka ylläpitävät käyttäjän määräämää järjestystä) kun säiliön alkioiden järjestystä muutetaan. Silloin kuitenkin menetetään lupauksia, joista voi olla käyttäjälle suurta hyötyä, kuten kuvan 4 yhteydessä nähtiin.

Kenties äärimmäiseen tehokkuuteen pyrkivistä tietorakenne- ja algoritmiabstraktioista ei voikaan saada vuotamattomia ilman kohtuuttomia uhrauksia.

Koska säiliöabstraktio vuotaa ainakin nykyisellään, on syytä harkita, kannattaako yrittääkään opettaa säiliöiden käyttöä opettamatta niiden taustalla olevia tietorakenteita. Tietorakenteiden perinteisen opettamisen puolesta puhuu myös se, että siinä opitaan muussakin ohjelmoinnissa hyödyllisiä tekniikoita sekä monimutkaisten ohjelmien toimimaan saamisessa tarvittavaa ajattelutapaa, kuten huomion kiinnittämistä invariantteina säilyviin asioihin. Näistä syistä olen sitä mieltä, että vielä ei ole tullut aika heittää pakollisille, perinteisen tyylisille tietorakenteiden ja algoritmien opintojaksoille hyvästit.

Kiitokset

C++-asiantuntija Matti Rintalan kanssa käymistäni keskusteluista ja hänen tästä kirjoituksesta antamistaan huomautuksista oli paljon hyötyä.

Viitteet

[1] Cormen, Thomas H. & Leiserson, Charles E. & Rivest, Ronald L.: *Introduction to Algorithms*. The MIT Press, 1990. xix+1028 s.

[2] International Standard ISO/IEC 14882, Programming languages — C++, First edition 1998-09-01. xxvi+748 s.

[3] Josuttis, Nicolai M.: *The C++ Standard Library, A Tutorial and Reference*. Addison–Wesley 1999. xx+799 s.

[4] Rintala, Matti: Tutnew — työkalu C++:n dynaamisen muistinhallinnan testaamiseen. *Tietojenkäsittelytiede* 17 Toukokuu 2002, ss. 8–23.

[5] Spolsky, Joel: The Law of Leaky Abstractions. Blogi *Joel on Software*, 11.11.2002, http://www.joelonsoftware.com/articles/LeakyAbstractions.html

[6] Stroustrup, Bjarne: *The C++ Programming Language, Third Edition*. Addison-Wesley 1997, x+910 s.

[7] Valmari, Antti: Mitä pieni Rubikin kuutio opetti minulle tietorakenteista, informaatioteoriasta ja satunnaisuudesta. *Tietojenkäsittelytiede* 20 Joulukuu 2003, ss. 53–78.