Union-find-delete-algoritmien vertailua

Sari Itäluoma

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Kesäkuu 2015

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Sari Itäluoma: Union-find-delete-algoritmien vertailua
Pro gradu -tutkielma, 46 sivua
Kesäkuu 2015

Union-find-algoritmi on tehokas ratkaisu erillisten joukkojen käsittelyongelmaan. Union-find-delete-algoritmissa siihen on lisätty poiston mahdollisuus. Esittelen tässä tutkielmassa useita union-find-algoritmeja ja union-find-delete-algoritmeja ja vertailen niitä teoreettisesti ja kokeellisesti.

Toteutin kolme erilaista union-find-delete-algoritmia, ja testasin niitä satunnaisilla syötteillä. Toteuttamani algoritmit ovat Ben-Amramin ja Yoffen algoritmi, aidon poiston algoritmi ja vajaan poiston algoritmi. Testieni perusteella vaikuttaa siltä, että asymptoottiselta aikavaatimukseltaan huonompi aidon poiston algoritmi toimii keskimäärin yhtä hyvin tai jopa paremmin kuin Ben-Amramin ja Yoffen algoritmi, jossa poisto tehdään vakioajassa.

Avainsanat ja -sanonnat: union-find-delete, union-find, erillisten joukkojen käsittelyongelma, aikakompleksisuus, algoritmien kokeellinen vertailu

Sisällys

1. Johdanto 1
2. Union-find-algoritmi 3
    - 2.1. Erillisten joukkojen käsittelyongelma 3
    - 2.2. Union-find-algoritmi 4
    - 2.3. Union-find-algoritmin tehostaminen 4
        - 2.3.1. Hakupolkujen lyhentäminen 5
        - 2.3.2. Yhdistäminen koon, korkeuden tai arvon perusteella 6
        - 2.3.3. Splicing eli punominen 7
        - 2.3.4. Aikainen yhdistäminen 8
    - 2.4. Smidin union-find-algoritmi 9
3. Union-find-delete-algoritmi 11
    - 3.1. Vajaan poiston algoritmi 11
    - 3.2. Smidin union-find-algoritmi poistolla 13
    - 3.3. Union-find-delete inkrementaalisella kopioinnilla 15
    - 3.4. Alstrupin ja muiden algoritmi 16
    - 3.5. Ben-Amramin ja Yoffen algoritmi 19
    - 3.6. Aidon poiston algoritmi 24
4. Algoritmien teoreettinen tehokkuus 27
    - 4.1. Asymptoottisista aikavaatimuksista ja merkintätavoista 27
    - 4.2. Erillisten joukkojen käsittelyongelman naiivit ratkaisut 27
    - 4.3. Union-find-algoritmien aikavaatimukset 28
    - 4.4. Union-find-delete-algoritmien aika- ja tilavaatimus 31
5. Algoritmien kokeellinen vertailu 34
    - 5.1. Algoritmien vertailu 34
    - 5.2. Testisyötteet 35
        - 5.2.1. Satunnaiset testit 35
        - 5.2.2. Keinotekoiset testit 36
    - 5.3. Testien tulokset 37
        - 5.3.1. Satunnaisten testien tulokset 37
        - 5.3.2. Keinotekoisten testien tulokset 39
    - 5.4. Testien tulosten analysointia 41
6. Yhteenveto 43

Viiteluettelo 45

1. Johdanto

Erillisten joukkojen käsittelyongelmassa halutaan ylläpitää tietoa siitä, mitkä jonkin alkioavaruuden alkiot kuuluvat samaan joukkoon. Ongelma voidaan ratkaista tallentamalla alkiot listaan tai taulukkoon, mutta nämä ratkaisut eivät ole kovin tehokkaita. Ongelman tehokas ratkaisu on union-find-algoritmi (englanniksi myös disjoint set algorithm ja equivalence algorithm), joka tallentaa alkiot puihin ja metsiin. Union-find-algoritmi tallentaa samaan joukkoon kuuluvat alkiot yhteen puuhun. Union-find-algoritmin nimi tulee joukolle tehtävistä operaatioista union (yhdistäminen) ja find (haku). Union yhdistää kaksi operaatiolle annettua joukkoa yhdeksi joukoksi. Find etsii ja palauttaa sen joukon nimialkion, johon operaatiolle annettu alkio kuuluu.

Union-find-algoritmia voidaan käyttää esimerkiksi graafien aputietorakenteena, kun halutaan selvittää tehokkaasti, ovatko jotkin graafin solmut yhteydessä toisiinsa kaarilla. Graafin jokainen solmu on yksi union-find-algoritmin puun solmu. Samassa puussa ovat kaikki sellaiset graafin solmut, joilla on yhteys toisiinsa kaarien kautta. Jos graafiin lisätään uusi kaari solmujen a ja b välille, annetaan union-find-algoritmille komento union(a, b). Union yhdistää solmun a joukon ja solmun b joukon yhdeksi joukoksi, jos ne eivät jo olleet yhteydessä toisiinsa, ja siten samaa joukkoa. Jos taas halutaan selvittää, ovatko solmut a ja b yhteydessä toisiinsa, annetaan union-find-algoritmille komennot find(a) ja find(b). Hakuoperaatiot palauttavat joukkojensa nimet, joita vertaamalla selviää, ovatko solmut yhteydessä toisiinsa. Näin voidaan esimerkiksi selvittää, loisiko kaaren lisääminen graafiin syklin. Union-find-algoritmia voidaan käyttää myös Kruskalin algoritmissa, joka etsii painotetun graafin pienemmän virittävän puun.

Klassisessa union-find-algoritmissa alkioita ei koskaan poistu joukosta, johon alkio on kerran lisätty. Kaplan ja muut [2002] lisäsivät union-find-algoritmiin poiston mahdollisuuden, jolloin algoritmista tuli nimeltään union-find-delete. Poisto-operaatio delete(a) poistaa alkion a puusta.

Tässä tutkielmassa käsittelen tätä union-find-algoritmin laajennusta, jossa algoritmiin on lisätty mahdollisuus poistaa solmuja. Union-find-delete-algoritmia voidaan käyttää samoin kuin union-find-algoritmia pitämään yllä tietoa siitä, mitkä graafin solmut kuuluvat samaan joukkoon, mutta nyt graafista voidaan lisäksi poistaa solmuja tai siirtää niitä toisiin joukkoihin. Union-find-delete-algoritmia käytetään esimerkiksi Mendelson ja muut [2006] prioriteettijonotietorakenteen kanssa. Union-find-delete-algoritmin avulla prioriteettijonoon on toteutettu yhdistämisoperaatio (meld).

Poisto-operaatio voidaan toteuttaa monella eri tavalla. Vertailen tässä työssä teoreettisesti ja kokeellisesti erilaisia union-find-delete-algoritmeja. Kokeellisessa vertailussa mukana on kolme algoritmia. Vertailen algoritmeista sekä aika- että tilavaatimusta.

Kokeellista vertailua varten toteuttamistani algoritmeista yksinkertaisemmin toteutettavat aidon poiston algoritmi ja vajaan poiston algoritmi ovat asymptoottiselta aika- tai tilavaatimukseltaan huonompia kuin Ben-Amramin ja Yoffen [2011] algoritmi. Kokeellisessa testauksessa kuitenkin kaikki kolme algoritmia suoriutuivat lähes yhtä hyvin.

Luvussa 2 esittelen erillisten joukkojen käsittelyongelman, sen joitakin naiiveja ratkaisuja ja tehokkaan ratkaisun, eli union-find-algoritmin. Esittelen myös union-find-algoritmille kehitettyjä tehostustapoja. Luvussa 3 esittelen kuusi erilaista union-find-delete-algoritmia. Luvussa 4 kerron näiden union-find- ja union-find-delete-algoritmien teoreettisista aika- ja tilavaatimuksista. Luvussa 5 esittelen tekemääni kokeellista tutkimusta, jossa ensin toteutin kolme union-find-delete-algoritmia ja testasin niitä kokeellisesti. Lisäksi analysoin saamiani tuloksia ja vertaan niitä teoreettisiin aika- ja tilavaatimuksiin. Luvussa 6 on yhteenveto.

2. Union-find-algoritmi

Tässä luvussa esittelen erillisten joukkojen käsittelyongelman ja sen ratkaisuja. Esittelen joitakin yksinkertaisia ratkaisuja ja kaksi puutietorakenteeseen perustuvaa ratkaisua, joilla ongelma voidaan ratkaista tehokkaasti. Tässä luvussa esittelemieni ratkaisujen aikavaatimuksista kerron luvussa 4, jossa vertailen algoritmien teoreettista tehokkuutta.

2.1. Erillisten joukkojen käsittelyongelma

Erillisten joukkojen käsittelyongelmassa halutaan ylläpitää tietoa siitä, mitkä alkiot kuuluvat keskenään samaan joukkoon. Kun jokin mielivaltainen määrä alkioita on jakautuneena erillisiin joukkoihin ja halutaan selvittää, kuuluvatko jotkin kaksi alkiota samaan joukkoon, joudutaan pahimmassa tapauksessa käymään läpi kaikki alkiot tämän selvittämiseksi. Ongelma voidaan ratkaista tallentamalla johonkin tietorakenteeseen tieto siitä, mihin joukkoon mikin alkio kuuluu. [Cormen *et al.*, 1996]

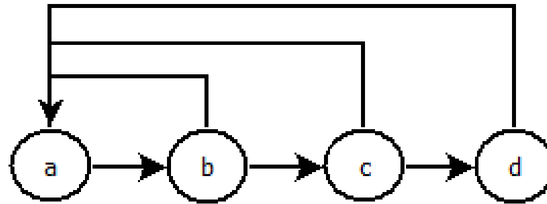
Tärkeimmät joukoille tehtävät operaatiot ovat haku (find) ja yhdistäminen (union). Haku kertoo, mihin joukkoon kyseinen alkio kuuluu. Hakuoperaatiolle annetaan haettava alkio parametrina, ja hakuoperaatio palauttaa sen joukon nimen, johon alkio kuuluu. Yhdistäminen liittää kaksi erillistä joukkoa yhdeksi joukoksi. Yhdistämisoperaatiolle annetaan kaksi alkiota, ja operaatio yhdistää joukot, joihin alkiot kuuluvat.

Erillisten joukkojen käsittelyongelman yksinkertaisia ratkaisuja ovat joukkojen esittäminen taulukkona tai listana. Nämä ratkaisut eivät kuitenkaan ole tehokkaita.

Taulukkoratkaisussa jokainen joukko on omassa taulukossaan. Joukon alkiot ovat peräkkäin taulukossa. Haku voidaan suorittaa tehokkaasti, sillä haku palauttaa taulukon, johon haettu alkio kuuluu. Yhdistämisoperaatio on kuitenkin tehoton. Yhdistäessä joudutaan päivittämään kaikkien yhdistettävien taulukoiden alkioiden tietoja, koska kaikki alkiot pitää kopioida uuteen riittävän suureen taulukkoon. [Mäkinen ja Poranen, 2011]

Cormen ja muut [1996] esittelevät linkitettyyn listaan perustuvan ratkaisun, jossa haku on tehokas ja yhdistäminen voidaan tehdä tehokkaammin kuin taulukkoratkaisussa. Tässä ratkaisussa alkiot ovat yksisuuntaisessa linkitetyssä listassa ja jokaisesta alkiosta on lisäksi osoitin listan ensimmäiseen alkioon, joka on joukkoa edustava nimialkio. Nyt haku on yhtä tehokas kuin taulukkoratkaisussa. Yhdistämisessä linkitetään toinen lista toisen perään. Yhdistämisen yhteydessä joudutaan päivittämään toisen taulukon kaikkien alkioiden osoitin taulukon nimeen. Kun yhdistäminen tehdään koon mukaan siten, että pienempi joukko lisätään suurempaan, joudutaan käymään läpi enintään puolet kaikista alkioista ja päivittämään niiden osoitin joukon nimeen.

Kuvassa 1 on esimerkkijoukko $\{a, b, c, d\}$, jonka alkiot ovat linkitetyssä listassa. Jokaisesta listan solmusta on osoitin seuraavaan solmuun ja lisäksi listan ensimmäiseen alkioon a, joka on joukon nimi.



Kuva 1: Linkitetty listaratkaisu.

2.2. Union-find-algoritmi

Gallerin ja Fisherin [1964] tehokkaampi ratkaisu erillisten joukkojen käsittelyongelmaan on esittää joukot puina. Puuratkaisu on huomattavasti tehokkaampi kuin listaan tai taulukkoon perustuvat ratkaisut. Tässä esitelty union-find-algoritmi on naiivi union-find-algoritmi, jossa yhdistäminen tehdään satunnaisesti. Seuraavassa kohdassa esittelen tehostustapoja algoritmille.

Union-find-algoritmissa joukon, eli puun, jokaisella solmulla on osoitin sitä vastaavaan alkioon. Lisäksi jokaisella solmulla on osoitin vanhempaansa. Puun juurisolmussa oleva alkio on puuta edustava solmu, eli puun nimi. [Tarjan, 1975]

Union-find-algoritmilla on kolme funktiota:

- makeset (luo joukko)
- union (yhdistä)
- find (hae).

Makeset(*a*) luo uuden puun, jonka ainoa alkio on *a*. Solmusta *a* tulee puun juuri ja samalla joukkoa edustava solmu. Juuren vanhempi on juuri itse.

Hakuoperaatiolle annetaan parametrina haettava alkio. Find(*a*) palauttaa sen joukon nimen, jonka jäsen *a* on. Joukon nimi on puun juuri. Hakuoperaatio seuraa annetusta alkiosta lähtien osoittimia vanhempaan, kunnes tullaan puun juureen. Hakuoperaatio palauttaa puun juurialkion.

Yhdistämisoperaatiolle annetaan parametreina kaksi alkiota. Union(*a*, *b*) tekee ensin operaatiot find(*a*) ja find(*b*), joilla selvitetään, mihin puihin alkiot kuuluvat. Jos alkiot kuuluvat eri puihin, puut yhdistetään. Puut yhdistetään siten, että pienemmän puun juuren vanhemmaksi tulee suuremman puun juuri.

Näistä kolmesta perusoperaatiosta voidaan johtaa esimerkiksi lisäysoperaatio insert(*a*, *b*), joka luo alkion *a* ja lisää sen puuhun, jossa alkio *b* on. Lisäysoperaatiossa tehdään peräkkäin makeset(*a*) ja union(*a*, *b*).

2.3. Union-find-algoritmin tehostaminen

Union-find-algoritmin hakuoperaatioita voidaan tehostaa lyhentämällä hakupolkuja. Tämä tehdään nostamalla solmuja ylemmäs puussa, jolloin hakupolku solmusta juureen on lyhempi. Hakupolkuja lyhennetään hakuoperaatioiden yhteydessä. Solmuja nostetaan

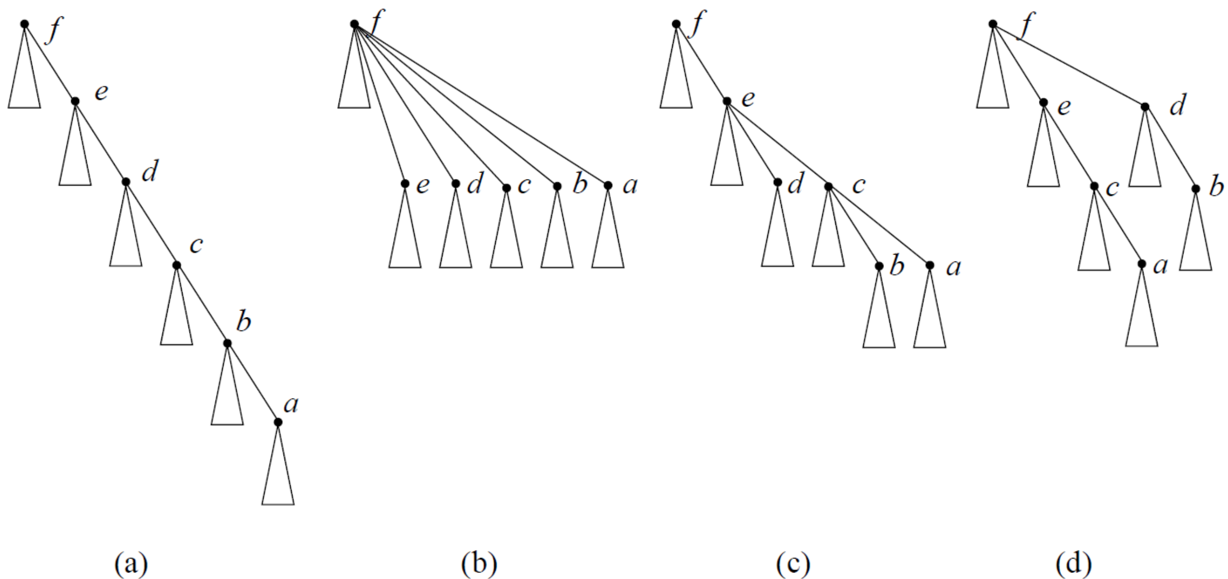
ylemmäs sitä mukaa, kun puussa edetään ylemmäs vanhempiosoittimia pitkin. Tämä tekee hausta hitaamman, mutta tehostaa tulevia hakuja huomattavasti.

Toinen tapa tehostaa hakuja on tehdä puiden yhdistäminen puun koon, korkeuden tai arvon mukaan. Tällöin hakupolut pitenevät mahdollisimman vähän, kun toisen puun juuri liitetään toisen puun juureksi. Edellisessä kohdassa esitellyssä Tarjanin [1975] union-find-algoritmissa näin jo tehdäänkin, koska puut yhdistetään koon mukaan.

Puut voidaan yhdistää myös punotulla yhdistämisellä, joka tekee yhdistämiseen liittyvät hakuoperaatiot lomitetusti ja punoo puut samalla yhteen. Toinen lomitettu yhdistämistapa on aikainen yhdistäminen, jossa molemmissa puissa ei tarvitse tehdä yhdistämiseen liittyviä hakuoperaatioita loppuun asti.

2.3.1. Hakupolkujen lyhentäminen

Hakupolkuja voidaan lyhentää hakuoperaatioiden yhteydessä esimerkiksi *tiivistämällä* (compression), *puolittamalla* (halving) tai *halkaisemalla* (splitting). Tiivistäminen nostaa kaikki hakupolulla olevat solmut suoraan juuren lapsiksi. Puolitus nostaa joka toisen hakupolulla olevan solmun isovanhempansa lapseksi. Halkaisu nostaa jokaisen hakupolulla olevan solmun isovanhempansa lapseksi. Kuvassa 2 on esitetty graafisesti nämä kolme hakupolkujen lyhennystapaa. Kuvan kolmiot ovat mielivaltaisia alipuita.



Kuva 2: Puu alkutilanteesa (a), tiivistetty puu (b), puolitettu puu (c) ja halkaistu puu (d). [Tarjan and van Leeuwen, 1984]

Tiivistäminen lyhentää hakupolkuja tehokkaimmin, koska kaikki hakupolun varrella olevat solmut nostetaan suoraan juuren lapsiksi. Tämä kuitenkin on melko työläs tapa lyhentää hakupolkuja, koska hakupolku joudutaan käymään läpi kahdesti. Puolitus ja halkaisu lyhentävät hakupolkua vähemmän, mutta operaatiot ovat paikallisia, jolloin

operaatiot voidaan tehdä samalla, kun puussa edetään ylöspäin. Kun hakuja tehdään useita, päädytään jossain vaiheessa samaan tilanteeseen kuin tiivistämistä käytettäessä, eli kaikki alkiot ovat suoraan juuren lapsia.

2.3.2. Yhdistäminen koon, korkeuden tai arvon perusteella

Union-find-algoritmia voi tehostaa myös tekemällä yhdistämisoperaatiot puun *koon* (size), *korkeuden* (height) tai *arvon* (rank) mukaan. Yhdistäminen tehdään liittämällä pienempi, matalampi tai pienempiarvoisempi puu suuremman, korkeamman tai suurempiarvoisen juuren lapseksi. Tätä tapaa käytetään usein yhdessä edellisen alakohdan hakupolkujen lyhennystapojen kanssa.

Puun koko on puun solmujen lukumäärä. Puun solmujen lukumäärästä on helppo pitää kirjaa. Yhdistämisen jälkeen uuden puun koko saadaan laskemalla puiden koot yhteen. Kun pienempi puu laitetaan suuremman puun juuren lapseksi, pienemmän puun alkioiden hakupolut pitenevät yhdellä askelella. Hakupolut pidentyvät kuitenkin vain enintään puolella solmuista.

Kun käytetään yhdistämistä korkeuden mukaan, puun jokaiseen solmuun tallennetaan tieto solmun korkeudesta puussa. Solmun korkeus on solmun ja sen syvimmällä olevan jälkeläisen välinen etäisyys. Puun korkeus on sama kuin juuren korkeus, eli puun syvimmällä olevan lehtisolmun ja juuren välinen etäisyys. Kun yhdistämistä korkeuden mukaan käytetään hakupolkujen lyhentämisen kanssa, solmujen korkeuksien päivittämisestä koituu lisätyötä. Solmujen korkeuksia joudutaan päivittämään jokaisen hakuoperaation yhteydessä, koska haku nostaa solmuja korkeammalle puussa. Hakupolkujen lyhentämisen kanssa sopii paremmin yhdistäminen koon tai arvon mukaan.

Arvo on sama kuin puun korkeus, kun hakupolkujen lyhennysoperaatioiden vaikutus solmujen korkeuksiin jätetään huomioimatta. Kun hakupolkuja lyhennetään nostamalla solmuja ylemmäs, arvoa ei päivitetä vastaamaan korkeutta. Hakupolun lyhennysoperaatio saattaa siis johtaa siihen, että lehtisolmun korkeus on suurempi kuin nolla. Jokaisen solmun arvo on kuitenkin aina pienempi kuin sen vanhemman arvo, koska solmua ei koskaan nosteta vanhempansa vanhemmaksi. Solmu saattaa nousta puussa korkeammalle kuin vanhempansa, mutta ei koskaan esivanhemmaksi. Solmun arvo ei koskaan pienene. Arvoa kasvatetaan vain yhdistämisoperaatioiden yhteydessä. Kun kaksi samanarvoista puuta yhdistetään, uudeksi juureksi arvotun solmun arvoa kasvatetaan yhdellä. Tosin joissain union-find-delete-algoritmeissa solmujen arvoja saatetaan muuttaa myös muissa tilanteissa.

Korkeuden tai arvon perusteella yhdistettäessä puun korkeus ei kasva, paitsi silloin, kun kaksi samankorkuista puuta yhdistetään.

2.3.3. Splicing eli punominen

Tarjan ja van Leeuwen [1984] esittelevät lomitetun yhdistämisoperaation, splicing, jossa yhdistämisoperaatioon liittyvät hakuoperaatiot tehdään lomitetusti. Lomitetut hakuoperaatiot ”punovat” puut yhteen ja samalla puiden solmut nousevat puussa ylemmäs. Lomittelu perustuu Remin algoritmiin ([Dijkstra, 1976] viitattu Tarjanin ja van Leeuwenin [1984] kautta).

Tässä algoritmissa oletetaan, että alkiot ovat järjestyksessä indeksin mukaan, siten että alempana olevan solmun indeksi on aina pienempi kuin ylempänä olevan. Tarjan ja van Leeuwen [1984] esittelevät myös arvon perusteella tehtävän splicing-algoritmin. Lomitettu yhdistäminen saa parametreina alkiot x ja y. Algoritmi lähtee etenemään annetuista solmuista ylöspäin samalla punoen hakupolut kiinni toisiinsa. Hakupoluilla edetään alkioiden vanhempien indeksin mukaan seuraavasti. Jos vanhempi(x) = vanhempi(y), operaatio lopetetaan, koska alkiot ovat jo samassa puussa. Muutoin otetaan solmuista se, jolla on pienempi vanhempi, vaikkapa x, ja laitetaan se suuremman vanhemman lapseksi. Solmuksi x tulee nyt sen entinen vanhempi. Jos x ei tämän jälkeen muuttunut, saavuttiin juureen ja operaatio lopetetaan.

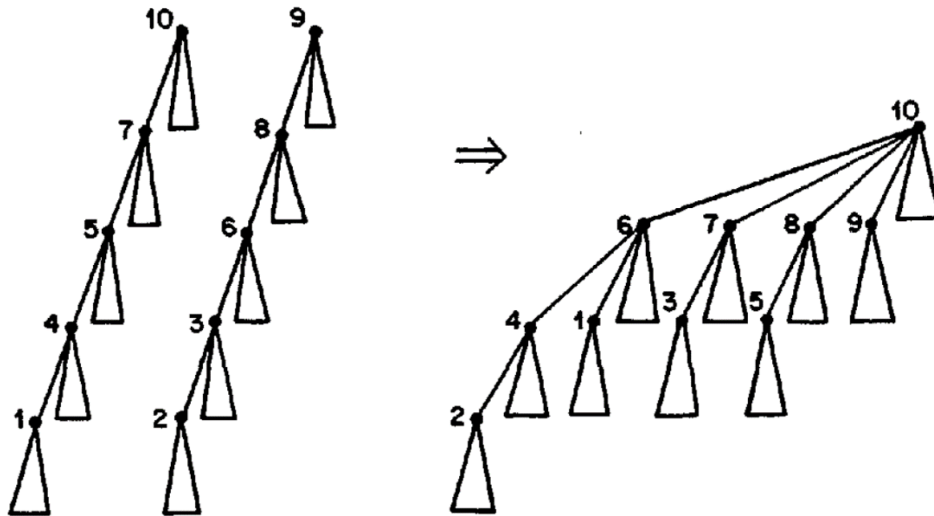
Koodiesimerkissä 1 on esitetty pseudokoodina punomalla toteutettu union. Kuvassa 3 on kaksi puuta, jotka yhdistetään operaatiolla union(1, 2). Kuvan kolmas puu on lopputulos, kun puut on lomitettu. Kuvan kolmiot ovat mielivaltaisia alipuita, jotka siirtyvät vanhempansa mukana.

```
function union(x,y)
    if(parent(x) = parent(y))
        return false

    if(parent(x) < parent(y))
        z = x
        x := parent(x)
        parent(x) := parent(y)
        if(z = x)
          return true

    if(parent(x) > parent(y))
        z = y
        y := parent(y)
        parent(y) := parent(x)
        if(z = y)
            return true
```

Koodiesimerkki 1: Puiden yhdistäminen punomalla.



Kuva 3: Punomalla yhdistäminen eli splicing [Tarjan and van Leeuwen, 1984].

2.3.4. Aikainen yhdistäminen

Goelin ja muiden [2014] mukaan yhdistämisoperaatiota arvon tai korkeuden mukaan voidaan tehostaa entisestään käyttämällä *aikaista yhdistämistä* (early linking). Aikaista yhdistämistä voidaan käyttää silloin, kun puusta ei tallenneta alkioiden lisäksi ylimääräistä tietoa, kuten joukon nimeä tai kokoa. Jos käytetään aikaista yhdistämistä, union ei palauta uuden joukon juurta. Aikaista yhdistämistä voidaan käyttää yhdessä alakohtien 2.3.1. ja 2.3.2. tehostustapojen kanssa.

Yhdistämisoperaatiolle annetaan kaksi alkiota, joille yhdistämisoperaatio tekee hakuoperaatiot ennen yhdistämistä. Jos käytetään aikaista yhdistämistä, hakuoperaatiot tehdään lomitetusti ja puut voidaan yhdistää, kun vain toinen juuri on löytynyt. Näin molemmissa puissa ei välttämättä tarvitse edetä juureen saakka. Esittelen tässä aikaisen yhdistämisen arvon mukaan.

Operaatio union(x, y) tekee hakuoperaatiot find(x) ja find(y). Kun käytetään aikaista yhdistämistä, hakuoperaatiot etenevät askelen kerrallaan ylemmäs puussa solmuista x ja y lähtien. Ylemmäs noustaan aina siitä solmusta, jonka vanhemmalla on pienempi arvo. Jos hakupolulla tullaan tilanteeseen, jossa solmuilla on sama vanhempi, alkiot ovat jo samassa puussa ja operaatio keskeytetään. Jos toinen hakupolku saavuttaa juuren, juuri liitetään toisella hakupolulla olevan solmun lapseksi. Jos molemmilla hakupoluilla tullaan juureen, ja juuret ovat samanarvoiset, satunnainen juuri lisätään toisen lapseksi, ja koko puun juuren arvoa kasvatetaan yhdellä.

Solmujen vanhempien arvot pysyvät edelleen aina suurempina kuin niiden lapsien arvot, koska hakupoluilla edetään aina pienempiarvoisempaan vanhempaan. Pienempiarvoisemman puun hakupolut pitenevät tässä ratkaisussa enemmän verrattuna arvon perusteella yhdistämiseen, koska yhdistäminen saatetaan tehdä aiemmin. Toisaalta tässä tapauk-

sessa koko hakupolkua ei tarvitse toisessa puussa käydä läpi. Uuden puun kokonaiskorkeus sen sijaan ei suurene, paitsi siinä tapauksessa, että yhdistettävien puiden arvot olivat samat.

2.4. Smidin union-find-algoritmi

Smidin ([1990] viitattu Kaplanin ja muiden [2002] kautta) union-find-algoritmi on hieman erilainen ratkaisu erillisten joukkojen käsittelyongelmaan. Smid ei käytä edellisen kohdan hakupolkujen lyhennystapoja, mutta algoritmin tehokkuus on silti samaa luokkaa. Kutsun algoritmia tässä Smidin algoritmiksi.

Smidin algoritmissa puun juuri on joukon nimi, mutta juuressa ei ole alkiota. Alkiot löytyvät puun lehtisolmuista. Sisäsolmuissa tai juuressa ei ole alkioita. Puussa on aina vähintään kaksi solmua, juuri ja lapsisolmu. Puun korkeus on siis aina vähintään 1. Yhden alkion puussa on juurisolmu ja lehtisolmu, jossa on puun ainoa alkio. Jokaisella puun solmulla on osoitin vanhempaan ja lapsilistan ensimmäiseen solmuun. Lapsilista on linkitetty lista. Puun juureen on tallennettuna joukon nimi, juuren lasten lukumäärä ja puun korkeus. Tässä sisäsolmulla tarkoitetaan solmua, joka ei ole juuri eikä lehti. Lisäksi sisäsolmulla, jonka korkeus on h, on vähintään k lasta, joiden korkeus on h–1. Tässä k on jokin vakioarvo. K on ihanteellinen solmun lasten vähimmäismäärä. Kun solmulla on k lasta, siitä tulee pysyvä solmu, jonka lapsia ei siirretä toisen solmun lapsiksi yhdistämisoperaatioiden aikana.

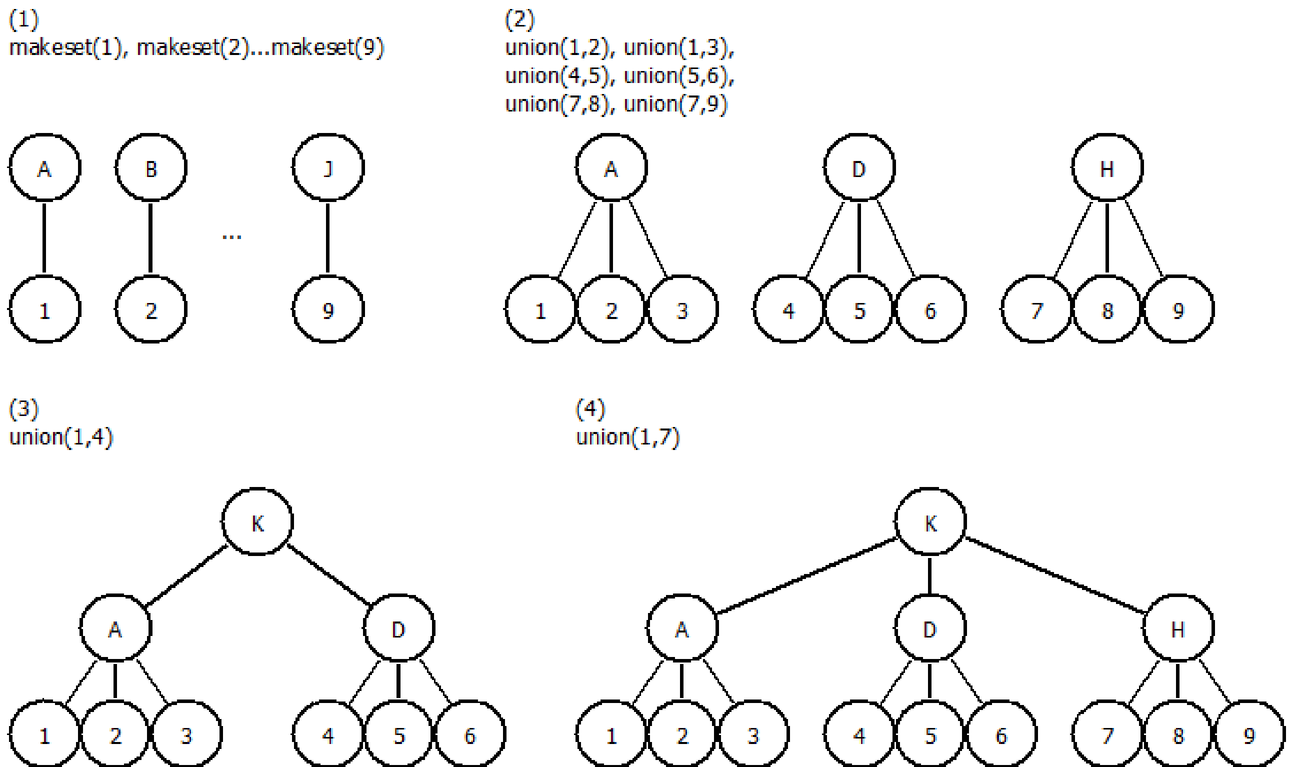
Hakuoperaatio tehdään samalla tavalla kuin kohdan 2.2. union-find-algoritmissa. Hakuoperaatio ei lyhennä hakupolkua. Hakuoperaatiolle annetusta alkiosta lähtien edetään puussa ylöspäin vanhempiosoittimia pitkin, kunnes löydetään puun juuri, eli joukon nimi.

Yhdistämisoperaatiossa union(A, B, C) yhdistetään joukot A ja B, joiden juuret ovat a ja b, joukoksi C. Lisäksi oletetaan, että puun A korkeus on pienempi tai yhtä suuri kuin joukon B korkeus, ja että jos A ja B ovat yhtä korkeita, niin solmulla b on vähintään yhtä monta lasta kuin solmulla a. Yhdistämisoperaatiossa tehdään jokin seuraavista kolmesta vaihtoehdosta:

1. Puun A korkeus on pienempi kuin puun B korkeus. Oletetaan, että v on jokin solmun b lapsi. Tässä tilanteessa on kolme vaihtoehtoa:
    1. Jos solmulla a on vähemmän kuin k lasta, siirretään solmun a kaikki lapset solmun v lapsiksi. Solmu a tuhotaan.
    2. Jos solmulla a on vähintään k lasta ja $h(a) = h(b) - 1$, solmusta a tulee solmun b lapsi. Päivitetään puun juuren, eli solmun b, lapsilukumuuttuja.
    3. Muutoin solmulla a on vähintään k lasta ja $h(a) < h(b) - 1$, jolloin solmusta a tulee solmun v lapsi.

    Lopuksi päivitetään solmuun b uuden joukon nimi.

2. Puut A ja B ovat yhtä korkeita ja solmulla *a* on vähemmän kuin *k* lasta. Solmun *a* kaikki lapset siirretään solmun *b* lapsiksi. Päivitetään puun juuren, eli solmun *b*, lapsilukumuuttuja.
3. Puut A ja B ovat yhtä korkeita ja solmulla *a* on vähintään *k* lasta (eli molempien puiden juurilla on vähintään *k* lasta). Luodaan uusi juuri *c*, jonka lapsiksi tulevat *a* ja *b*. Solmut *a* ja *b* muodostavat nyt solmun *c* lapsilistan. Juuren *c* lapsimääräksi tulee kaksi ja korkeudeksi solmun *a* korkeus lisättynä yhdellä.



Kuva 4: Smidin algoritmin operaatioita, kun *k* = 3.

Kuvassa 4 on joitakin esimerkkioperaatioita Smidin algoritmista. Kuvan puissa ovat alkiot 1–9. Kirjaimilla merkityissä solmuissa ei ole alkiota. Kuvan kohdassa (1) luodaan yhdeksän yhden alkion joukkoa ja kohdassa (2) ne yhdistetään kolmeksi kolmen alkion puuksi. Esimerkeissä *k* = 3, joten solmut A, D ja H ovat täysiä, koska niillä on *k* = 3 lapsisolmua. Kohdassa (3) yhdistetään joukot A ja D, joilla on täysi määrä lapsia, joten luodaan uusi juuri, jonka lapsiksi tulevat A ja D. Tämän uuden puun K korkeus on 2. Kohdassa (4) puuhun K yhdistetään puu H, jonka lapsimäärä on vähintään *k* ja korkeus 1. Puu H lisätään suoraan juuren K lapseksi. Jos puu K olisi ollut vielä korkeampi, H olisi lisätty solmun K ensimmäisen lapsen lapseksi (tai jonkin solmun K lapsen lapseksi, helpointa on kuitenkin edetä ensimmäiseen lapseen). Jos taas solmulla H olisi vähemmän lapsia, sen lapset lisättäisiin solmun A lapsiksi.

3. Union-find-delete-algoritmi

Kaplan ja muut [2002] laajentavat union-find-algoritmia lisäämällä poiston mahdollisuuden. Kutsun laajennettua union-find-algoritmia union-find-delete-algoritmiksi.

Poisto voidaan toteuttaa monella tavalla. Esittelen alakohdissa 3.1.–3.6. erilaisia poistoratkaisuja. Toteutin kolme näistä ratkaisuista ja testaan niitä kokeellisesti. Kokeellisesti tutkimani algoritmit ovat vajaan poiston algoritmi, aidon poiston algoritmi ja Ben-Amramin ja Yoffen algoritmi. Esittelen kokeellista tutkimusta luvussa 5.

Union-find-algoritmia käytettäessä voidaan ajatella, että alkioavaruus, jossa alkiot ovat jakautuneet joukkoihin, on kiinnitetty. Alkiot ovat aina olemassa, mutta niitä saatetaan yhdistää uusiksi joukoiksi. Union-find-delete-algoritmin tapauksessa alkioavaruus ei ole kiinnitetty, vaan alkioita voidaan poistaa ja lisätä. [Kaplan *et al.*, 2002]

Union-find-delete-algoritmin poisto-operaatio tuhoaa kaiken alkioon liittyvän tiedon. Alkio poistetaan puusta, ja alkio tuhotaan myös kaikkien alkioiden avaruudesta. Poistofunktiolle annetaan parametrina vain poistettava alkio, ei esimerkiksi tietoa siitä, mihin joukkoon alkio kuuluu.

Poistamisen mahdollisuuden lisääminen union-find-algoritmiin mahdollistaa esimerkiksi seuraavien johdettujen operaatioiden käyttämisen:

- Detach(a), joka irrottaa solmun a puusta omaksi joukokseen. Irrotusoperaatiossa tehdään peräkkäin delete(a) ja makeset(a).
- Replace(a, b), joka korvaa solmun a solmulla b. Korvausoperaatiossa tehdään peräkkäin makeset(b), union(b, a) ja delete(a).
- Move(a, b), joka siirtää alkion a joukkoon b. Siirto-operaatiossa tehdään peräkkäin delete(a), makeset(a), union(a, b).

3.1. Vajaan poiston algoritmi

Ensimmäinen Kaplanin ja muiden [2002] esittämä poistotapa on yksinkertaisesti olla tekemättä mitään. Alkio vain merkitään poistetuksi, ja se pysyy puussa muiden alkioiden rinnalla. Tämän ratkaisun huono puoli on se, ettei tilavaatimus pysy enää suhteessa ”elävien” alkioiden lukumäärään.

Tämä yksinkertainen poistotapa on yksi kokeellista tutkimusta varten toteuttamistani algoritmeista. Toteutin algoritmin siten, että poistettava alkio tuhotaan, mutta itse solmu jää puuhun. Solmu on siis ”ontto”. Kutsun algoritmia vajaan poiston algoritmiksi, koska alkio poistetaan, mutta kaikki alkioon liittyvä muistinvaraus ei vapaudu.

Vajaan poiston algoritmi on hyvin samanlainen kuin union-find-algoritmi. Puun rakenne on samanlainen kuin union-find-algoritmissa. Jokaisesta puun solmusta on osoitin alkioon ja vanhempaan, mutta ei lapsisolmuihin.

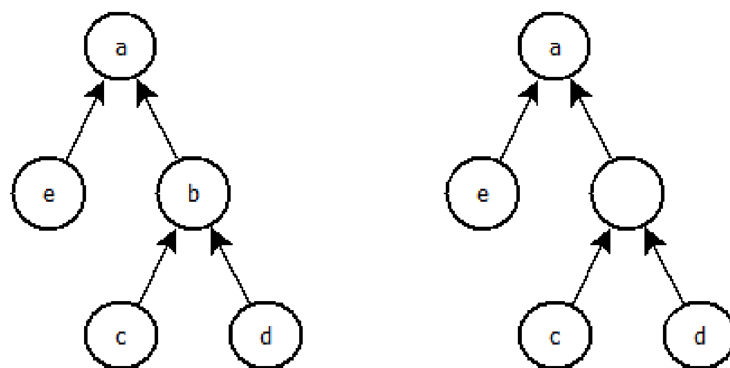
Haku toimii samalla tavalla kuin union-find-algoritmin haku. Haettavasta solmusta edetään ylemmäs puussa vanhempiosoittimia pitkin, kunnes tullaan juureen. Toteuttamani

algoritmi käyttää halkaisua hakupolkujen lyhentämiseen, kun puussa edetään ylemmäs. Lopuksi haku palauttaa juuren.

Myös yhdistäminen tehdään samalla tavalla kuin union-find-algoritmissa. Haku-operaatiolle annetaan kaksi alkiota, ja puut, joihin alkiot kuuluvat yhdistetään. Yhdistämis-operaatio selvittää aluksi, mihin puihin annetut alkiot kuuluvat, tekemällä hakuoperaation molemmille alkioille. Jos alkiot kuuluvat eri puihin, puut yhdistetään. Jos alkiot ovat jo valmiiksi samassa puussa, ei tehdä mitään. Toteuttamassani algoritmissa puut yhdistetään arvon perusteella. Pienempiarvoisemman puun juuren vanhemmaksi tulee suurempiarvoisemman puun juuri. Mikäli juurien arvot ovat samat, uuden puun juureksi tuleva juuri valitaan satunnaisesti. Toinen juuri lisätään toisen lapseksi. Uuden juuren arvoa kasvatetaan yhdellä.

Poisto-operaatio poistaa alkion ja solmun viittauksen alkioon, jolloin puuhun jää ontto solmu. Koska poistettavan alkion solmulla ei ole tietoa lapsistaan, kaikkea alkioon liittyvää tietoa ei voida poistaa. Jos solmu poistettaisiin kokonaan, sen lapsisolmujen yhteys juureen katkeaisi. Lapsisolmuihin ei päästä päivittämään tietoa uudesta vanhemmasta, paitsi käymällä läpi kaikkien puiden solmut.

Kun puuhun jää onttoja solmuja, osoittimien määrä ei vähene solmuja poistettaessa. Koko rakenteen tilavaatimus pienenee hieman, koska itse alkion muistipaikka vapautuu. Hakupolku pysyy yhtä pitkänä kuin ennen poistoa. Puun rakenne on siis aluksi kevyt, mutta jokainen poisto jättää jälkeensä kaksi ylimääräistä osoitinta: osoittimen vanhempaan ja *null*-osoittimen alkio-osoittimen tilalle.



Kuva 5: Osoittimet vajaan poiston algoritmin puussa ja puu alkion *b* poistamisen jälkeen.

Kuvassa 5 on kaksi esimerkkipuuta vajaan poiston algoritmista. Kuvassa ympyrät ovat solmuja ja nuolet osoittimia. Solmujen sisältämät alkiot ovat selkeyden vuoksi solmujen sisällä, vaikka todellisuudessa solmusta on osoitin alkioon. Kuvan vasemmanpuoleisessa

puussa on tilanne luonti- ja yhdistämisoperaatioiden jälkeen. Oikeanpuoleinen puu kuvaa tilannetta poisto-operaation jälkeen.

3.2. Smidin union-find-algoritmi poistolla

Kaplan ja muut [2002] laajentavat kohdassa 2.4. esiteltyä Smidin [1990] union-find-algoritmia lisäämällä siihen poisto-operaation. Tässäkin algoritmissa alkiot tallennetaan puun lehtisolmuihin.

Poisto-operaation lisäämiseksi puun sisäsolmut luokitellaan seuraaviin kolmeen luokkaan: solmu on *lyhyt*, *voitolla* tai *häviöllä* (short, gaining, losing). Solmu v on lyhyt, jos sen vanhemman korkeus on enemmän kuin yhtä suurempi kuin solmun korkeus, eli $h(p(v)) > h(v) + 1$, jossa $p(v)$ on solmun v vanhempi ja $h(v)$ on solmun v etäisyys lehdestä. Jos solmu ei ole lyhyt, se on joko voitolla tai häviöllä. Lisäksi määritellään, että jokaisella sisäsolmulla, jonka korkeus on suurempi kuin yksi, on täsmälleen yksi voitolla oleva lapsisolmu.

Jokaisella puun solmulla on kolme osoitinta lapsiin. Solmulla on osoitin voitolla olevaan lapseen, listaan häviöllä olevista lapsista ja listaan lyhyistä lapsista. Lisäksi solmulla on laskurimuuttuja, jossa on solmun ei-lyhyiden lasten lukumäärä. Puulle määritellään lisäksi jokin vakioarvo k, joka on ihanteellinen solmun lasten vähimmäismäärä. Kun solmulla on k lasta, siitä tulee pysyvä solmu, jonka lapsia ei siirretä toisen solmun lapsiksi yhdistämis-operaatioiden aikana.

Näistä määritelmistä seuraa, että puu täyttää seuraavat ehdot:

- juurella ei ole lyhyitä lapsia
- juurella, jonka korkeus $h > 1$, on vähintään kaksi lasta
- voitolla olevalla solmulla on vähintään k lasta
- häviöllä olevalla solmulla, jonka korkeus $h > 0$, on vähintään kaksi lasta.

Hakuoperaatio find(x), seuraa vanhempiosoittimia ylöspäin puussa ja palauttaa puun juuresta löytyvän puun nimen. Hakupolkua ei lyhennetä.

Yhdistämisoperaatiossa union(A, B, C) yhdistetään joukot A ja B, joiden juuret ovat a ja b, joukoksi C. Lisäksi oletetaan, että puun A korkeus on pienempi tai yhtä suuri kuin joukon B korkeus, ja, että jos A ja B ovat yhtä korkeita, niin solmulla b on vähintään yhtä monta lasta kuin solmulla a. Yhdistämisoperaatiossa tehdään jokin seuraavista kolmesta vaihto-ehdosta.

1. Puun A korkeus on pienempi kuin puun B korkeus. Oletetaan, että v on jokin solmun b lapsi. Nyt on kolme vaihtoehtoa:
    1. $h(a) < h(v) - 1$. Solmusta a tehdään solmun v lyhyt lapsi.
    2. $h(a) = h(v) - 1$. Jos solmulla a on vähemmän kuin k lasta, siirretään solmun a kaikki lapset solmun v lyhyiksi lapsiksi. Solmu a hylätään. Jos taas solmulla a on vähintään k lasta, solmusta a tehdään solmun v häviöllä oleva lapsi.

3. $h(a) = h(v)$. Jos solmulla a on vähemmän kuin k lasta, solmun a lapset siirretään solmun v häviöllä oleviksi lapsi. Jos taas solmulla a on vähintään k lasta, solmusta a tehdään solmun b häviöllä oleva lapsi.

   Lopuksi päivitetään solmuun b uuden joukon nimi, ja päivitetään solmun b tai v lapsilukulaskurit.

2. Puut A ja B ovat yhtä korkeita ja solmulla a on vähemmän kuin k lasta. Solmun a kaikki lapset siirretään solmun b lapsiksi. Häviöllä olevien lasten listat yhdistetään ja solmun a voitolla olevasta lapsesta tulee häviöllä oleva lapsi. Päivitetään puun juuren, eli solmun b, lapsilukumuuttuja.
3. Puut A ja B ovat yhtä korkeita ja solmulla a on vähintään k lasta (eli molempien puiden juurilla on vähintään k lasta). Luodaan uusi juuri c, jonka lapsiksi tulevat a ja b. Solmusta b tehdään juuren voitolla oleva lapsi, ja solmusta a tehdään häviöllä oleva lapsi. Solmut a ja b muodostavat nyt solmun c lapsilistan. Juuren c lapsimääräksi tulee kaksi ja korkeudeksi solmun a korkeus lisättynä yhdellä.

Poisto-operaatio delete(x) säilyttää puun aiempien määrittelyjen mukaisena. Oletetaan aluksi, että puussa, josta x poistetaan, ei ole lyhyitä solmuja. Poisto-operaatio käyttää rekursiivista funktiota delete′, joka poistaa puusta sellaisen solmu w, joilla ei ole lapsisolmuja. Funktio delete′ saattaa luoda uuden lapsettoman solmun, jolloin funktiota kutsutaan uudelleen. Aluksi funktiota delete′ sovelletaan poistettavaan solmuun x, ja sen jälkeen häviöllä oleviin solmuihin, joilla ei ole jäljellä yhtään lapsisolmua. Seuraavassa on funktion delete′ määrittely. Poistettava solmu on w, ja sen vanhempi on v. Funktiossa delete′ on kolme vaihtoehtoista tilannetta:

1. Solmu v on häviöllä. Poista w vanhempansa lapsilistasta. Jos tämän jälkeen solmulla v on jäljellä yksi lapsi, tämä ainoa lapsi siirretään solmun v voitolla olevan sisaruksen lapseksi ja v poistetaan, eli kutsutaan funktiota delete′(v).
2. Solmu v on voitolla. Solmulla v on häviöllä oleva sisarus u. Jonkin solmun u häviöllä olevan lapsen ja solmun w paikat vaihdetaan keskenään. Nyt solmun w vanhempi on häviöllä, ja jatketaan kuten kohdassa 1.
3. Solmu v on puun juuri. Nyt on kolme vaihtoehtoa:
   1. Jos w on juuren ainoa lapsi, joukko tyhjenee, kun w poistetaan. Poistetaan sekä v että w.
   2. Jos w on lehti, mutta se ei ole juuren ainoa lapsi, poistetaan vain w.
   3. Jos w ei ole lehti ja solmu w ja sen voitolla oleva sisarus ovat juuren ainoat lapset, w ja v poistetaan. Solmun w voitolla olevasta sisaruksesta tulee puun uusi juuri.

Seuraavassa kuvaan poisto-operaation, kun se laajennetaan toimimaan myös puussa, jossa on lyhyitä solmuja. Poisto aloitetaan kulkemalla poistettavasta solmusta x ylöspäin puussa ja selvittämällä, onko hakupolulla lyhyitä solmuja, eli onko solmulla x lyhyttä esivanhempaa. Tämän jälkeen tehdään jokin seuraavista:

1. Jos löytyi lyhyt esivanhempi, olkoon solmua x lähinnä oleva esivanhempi v. Poistetaan x, ja jos x oli vanhempansa ainoa lapsi, poistetaan myös vanhempi. Peräkkäisiä solmuja poistetaan niin kauan, kunnes tullaan solmuun, jolla on enemmän kuin yksi lapsi tai kunnes on poistettu solmu v.
2. Jos solmulla x ei ole lyhyttä esivanhempaa, suoritetaan yllä oleva funktio delete' simuloidusti ilman, että päivityksiä tehdään. Tässä vaiheessa tarkistetaan, onko jollakin poiston kohteeksi joutuvista solmuista lyhyt lapsi tai onko jollain poistettavan solmun voitolla olevalla sisaruksella lyhyt lapsi. Nyt on kaksi vaihtoehtoa:
   1. Jos lyhyitä lapsia ei löytynyt, suoritetaan funktio delete' aiemmin kuvatulla tavalla.
   2. Jos löytyi jokin lyhyitä lapsia omistava solmu v, seurataan tästä solmusta osoittimia alaspäin lehteen asti. Tämän löydetyn lehden ja solmun x paikkaa vaihdetaan keskenään. Nyt solmulla x on lyhyt esivanhempi, ja poisto-operaatio tehdään kohdan 1 mukaisesti.

3.3. Union-find-delete inkrementaalisella kopioinnilla

Kaplan ja muiden [2002] union-find-delete-algoritmissa poisto toteutetaan *inkrementaalisella kopioinnilla* (incremental copying). Poisto voidaan toteuttaa tällä menetelmällä mihin tahansa puumuotoiseen union-find-algoritmiin.

Inkrementaalisella kopioinnilla toteutetussa union-find-delete-algoritmissa poisto toteutetaan samalla tavalla kuin vajaan poiston algoritmissa. Poistettava alkio vain merkitään poistetuksi ja ontto solmu jää puuhun. Onttoja solmuja kuitenkin karsitaan puusta aika ajoin, joten puun muistinvaraus on suhteessa alkioiden määrään.

Jokaista joukkoa edustaa yksi tai kaksi puuta. Joukkoa S edustaa joko puu S_n yksinään tai puut S_n ja S_o. Puussa S_n pidetään yllä tietoa siitä, kuinka monta alkiota puusta on poistettu, eli montako onttoa solmua puussa on. Molemmilla puilla on lista puussa olevista ”elävistä” alkioista, eli alkioista, joita ei ole merkitty poistetuiksi. Näitä listoja kutsutaan tässä nimillä $L(S_n)$ ja $L(S_o)$. Joukkoa S edustavasta nimialkiosta on osoitin puiden S_n ja S_o nimialkioihin, eli juuriin, ja toisinpäin.

Kun joukko luodaan luontioperaatiolla makeset(x), joukkoa edustaa vain yksi puu S_n, jonka ainoaksi alkioksi tulee x. Lisäksi alkio lisätään listaan $L(S_n)$.

Kun joukot A ja B yhdistetään joukoksi C operaatiolla union(A, B, C), molemmat joukkoa edustavat puut yhdistetään. Todellisuudessa tehdään siis kaksi yhdistämisoperaatiota, union(A_n, B_n) ja union(A_o, B_o). Lisäksi alkiolistat $L(A_n)$ ja $L(B_n)$ sekä $L(A_o)$ ja $L(B_o)$ yhdistetään listoiksi $L(C_n)$ ja $L(C_o)$. Puun C_n poistettujen alkioiden lukumäärän laskuri päivitetään. Lopuksi asetetaan vielä joukkoa C edustava nimialkio osoittamaan puita C_n ja C_o edustaviin alkioihin ja puita edustavat nimialkiot osoittamaan joukkoa edustavaan nimialkioon.

Hakuoperaatio find(x) seuraa vanhempiosoittimia ylemmäs puussa, kunnes tullaan puun juureen, josta on osoitin koko joukkoa edustavaan nimialkioon. Operaatio palauttaa tämän koko joukkoa edustavan alkion.

Kun alkio poistetaan operaatiolla delete(x), suoritetaan ensin operaatio find(x). Nyt tiedossa on, mihin joukkoon x kuuluu ja kummassa puussa, S_n vai S_o, solmu x on. Solmu merkitään poistetuksi ja poistetaan myös puun alkiolistasta. Jos x on puussa S_n, lisätään puun onttojen solmujen määrää yhdellä. Tämän jälkeen tarkistetaan, onko puun S_n solmuista vähintään ¼ onttoja. Jos näin on, puun S_n nimeksi annetaan S_o ja aloitetaan uusi puu S_n.

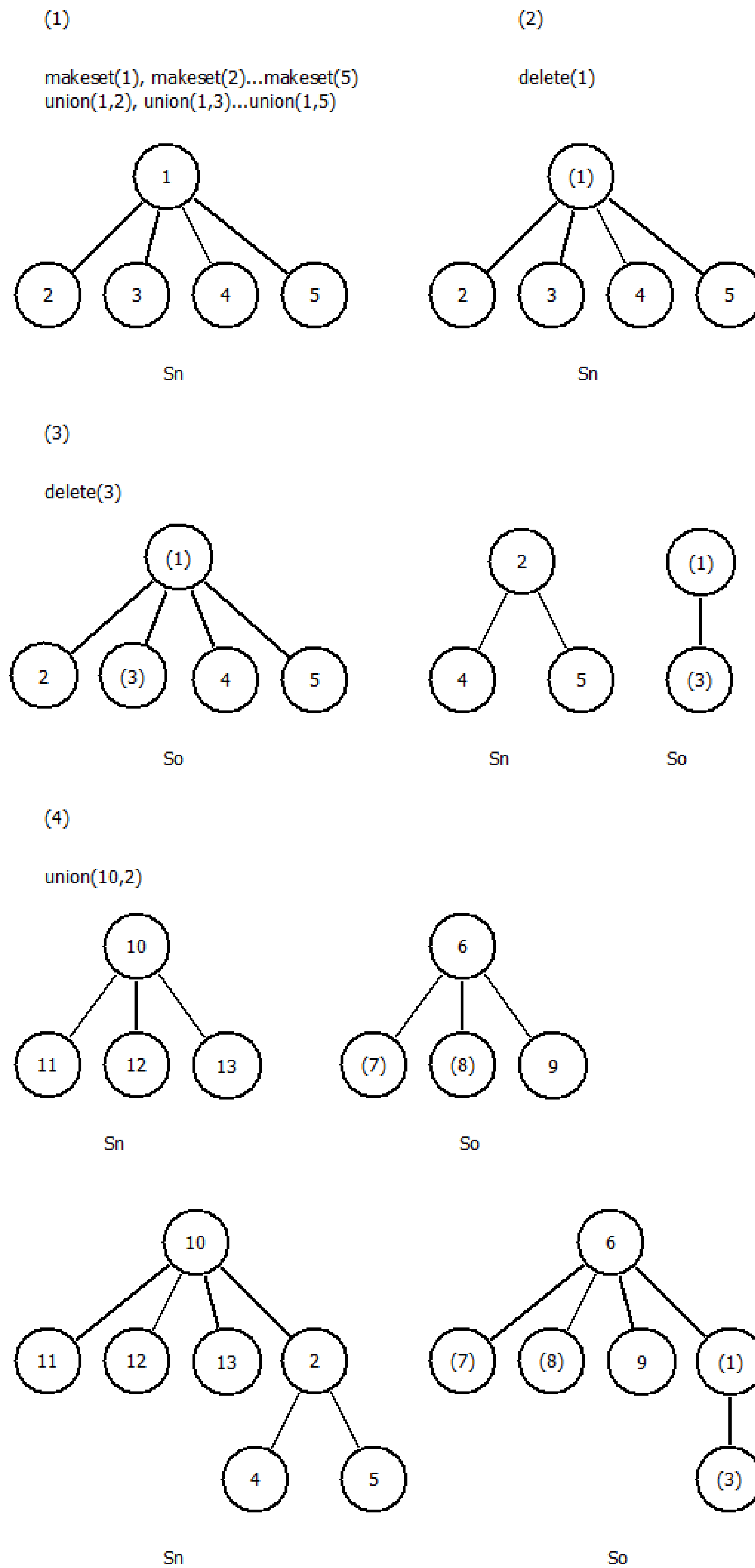
Poiston jälkeen tarkistetaan, onko lista L(S_o) tyhjä, eli onko puussa S_o vain onttoja solmuja. Jos lista L(S_o) ei ole tyhjä, siitä siirretään neljä (tai kaikki solmut, jos listassa on alle neljä solmua) solmua puuhun L(S_n). Jos listassa L(S_o) on tämän jälkeen alle neljä solmua, siirretään nekin puuhun S_n. Siirtäminen tehdään poistamalla alkiot listasta L(S_o) luomalla ne uudelleen makeset-operaatiolla ja lisäämällä ne sitten yhdistämisoperaatiolla puuhun S_n.

Algoritmi toimii, koska jos puun S_n alkioista ¼ on onttoja, puussa S_o on oltava vain onttoja solmuja. Tämän takia joukon puu S_o voidaan tuhota poistotilanteessa, jos joukon S_n alkioista neljännes on onttoja.

Kuvassa 6 on esimerkkipuita inkrementaalisella kopioinnilla toteutetusta union-find-delete-algoritmista. Kuvassa alkiot ovat solmujen sisällä olevia kokonaislukuja. Suluissa olevat alkiot ovat poistetuiksi merkittyjä alkioita. Kuvan kohdassa (1) olevassa puussa on tilanne, jossa on luotu viisi alkiota ja yhdistetty ne yhdeksi joukoksi. Joukon kaikki alkiot ovat joukossa S_n, koska yhtään alkiota ei ole vielä poistettu. Puun alkiot 1–5 ovat kaikki listassa L(S_n). Kun kohdassa (2) tästä puusta poistetaan alkio 1, solmu merkitään tyhjäksi, alkio poistetaan listasta L(S_n) ja alkioiden lukumäärää puussa vähennetään yhdellä. Nyt alle neljännes puun solmuista on onttoja, joten muuta ei tehdä. Kohdassa (3) puusta poistetaan alkio 3. Ensin alkio merkitään poistetuksi. Nyt puun solmuista yli neljännes on onttoja, joten puulle annetaan nimeksi S_o ja siitä siirretään enintään neljä alkiota puuhun S_n. Tässä tapauksessa siirretään kolme alkiota, koska alkioita on alle neljä. Viimeisessä kohdassa (4) on toinen joukko, joka on jakautunut kahdeksi puuksi. Toisessa puussa on luotu alkiot 6–13, yhdistetty ne samaan puuhun ja sitten poistettu alkiot 7 ja 8. Kun kohdan (3) joukko yhdistetään tähän uuteen joukkoon, niiden puut yhdistetään toisen joukon vastaavien puiden kanssa. Kuvan viimeinen puu kuvaa tätä yhdistettyä joukkoa.

3.4. Alstrupin ja muiden algoritmi

Alstrup ja muut [2014] esittelevät union-find-delete-algoritmin, jossa poisto on toteutettu tehokkaammin kuin kahdessa edellisessä algoritmissa. Tässä toteutuksessa poisto onnistuu vakioajassa, koska poistaminen tehdään paikallisilla operaatioilla, eikä puussa edetä juureen asti.



Kuva 6: Union-find-delete-algoritmi inkrementaalisella kopioinnilla toteutettuna.

Alstrupin ja muiden [2014] algoritmin lähtökohtana on kohdassa 3.1. esitelty vajaan poiston algoritmi. Vajaan poiston algoritmin ongelmia ovat tilanhaaskaus ja se, ettei hakuoperaation tehokkuus ole enää suhteessa alkioiden määrään. Alstrupin ja muiden algoritmi ratkaisee nämä ongelmat tekemällä poisto-operaatioiden yhteydessä paikallisia siistimisoperaatioita, jotka poistavat onttoja solmuja, ja nostamalla solmuja ylemmäs puussa.

Jotta onttoja solmuja voidaan poistaa, Alstrupin ja muiden algoritmissa jokaisella puun solmulla on vanhempiosoittimen lisäksi osoittimet sisaruksiin ja lapsilistan ensimmäiseen solmuun. Lapsilistat ovat kaksisuuntaisia linkitettyjä listoja.

Luonti-, haku- ja yhdistämisoperaatiot tehdään samalla tavalla kuin union-find-algoritmissa. Yhdistäminen tehdään arvon perusteella. Poisto-operaatio merkitsee alkion poistetuksi, jolloin puuhun jää ontto solmu. Tämän jälkeen poistetun alkion ympäristössä tehdään *siistimistä* (tidy-up) ja *oikaisuja* (short-cut).

Algoritmissa oletetaan, että seuraavat ehdot täyttyvät

- juurisolmun vanhempiosoitin osoittaa solmuun itseensä, eli $parent(x) = x$
- lehtisolmun lapsiosoitin osoittaa solmuun itseensä, eli $child(x) = x$
- kaikissa lapsilistoissa ovat ensin ei-lehtisolmut ja sen jälkeen kaikki lehtisolmut.

Viimeisestä kohdasta johtuen solmuja joudutaan joskus siirtämään lapsilistan loppuun. Esimerkiksi, jos solmulla x on yksi lapsi y, ja y poistetaan puusta, solmusta x tulee lehti, ja se siirretään vanhempansa lapsilistassa viimeiseksi solmuksi.

Algoritmissa määritellään myös, että puu on *siisti* (tidy), kun seuraavat ehdot täyttyvät:

- jokaisella ontolla solmulla, joka ei ole juuri, on vähintään kaksi lasta
- lehtisolmu ei ole koskaan ontto, ja lehden arvo on aina 0.

Poisto-operaatio delete(x) saattaa aiheuttaa sen, ettei puu ole siisti. Kun poisto-operaatio on merkinnyt poistetun alkion solmun ontoksi, nyt onton solmun ympäristöä siistitään seuraavasti:

1. Jos x on sisäsolmu (ei juuri tai lehti), jolla on vain yksi lapsi, solmu voidaan ohittaa ja poistaa puusta. Solmun x ainoa lapsi siis nostetaan isovanhempansa lapseksi.
2. Jos x on ontto lehti, x poistetaan puusta. Nyt solmun x vanhempi saattaa vielä rikkoa puun siisteyssääntöä seuraavilla tavoilla:
   1. Jos solmun x vanhemmasta tuli lehtisolmu ja se ei ole ontto, sen arvoksi asetetaan 0.
   2. Jos taas solmun x vanhempi on ontto, ja sillä on vain yksi lapsi, se voidaan ohittaa ja poistaa.

Puhdistusoperaation jälkeen varmistetaan vielä, että puu pysyy matalana nostamalla solmuja ylemmäs puussa. Näitä nostoja tai oikaisuja tehdään vakiomäärä. Puussa kuljetaan ensin vanhempiosoittimia pitkin ylöspäin neljä askelta (tai kunnes tullaan juureen), ja saavutaan solmuun z. Jos solmulla z on lapsenlapsia, tehdään seuraava oikaisuoperaatio

neljä kertaa. Oikaisu ottaa solmun z lapsenlapsenlapsenlapsen (tai vastaan tulleen lehtisolmun ennen kuin päästään neljänteen polveen) ja nostaa sen solmun z lapseksi. Jos tämän operaation tuloksena solmusta x tuli ontto solmu, jolla on vain yksi lapsi, se voidaan poistaa.

3.5. Ben-Amramin ja Yoffen algoritmi

Ben-Amramin ja Yoffen [2011] algoritmi on kehittäjiensä mukaan hieman yksinkertaisempi ja (vakiokertoimien tasolla) tehokkaampi versio Alstrupin ja muiden [2014] union-find-delete algoritmista. Ben-Amramin ja Yoffen algoritmin asymptoottinen aikavaatimus on sama kuin Astrupin ja muiden algoritmissa.

Ben-Amram ja Yoffe [2011] määrittelevät, että puu voi olla joko *täysi* (full) tai *vajaa* (reduced). Puu on vajaa, jos puussa on vain yksi solmu tai, jos puun korkeus on 1, puun juuren arvo on 1 ja lapsisolmujen arvot ovat 0. Täydessä puussa jokaisella solmulla joko ei ole yhtään lapsisolmua tai sillä on vähintään kolme lapsisolmua. Ben-Amram ja Yoffe määrittelevät lisäksi, että puu on aina joko täysi tai vajaa. Tästä seuraa, että vähintään neljän solmun kokoiset puut ovat täysiä ja alle neljän solmun kokoiset puut ovat vajaita.

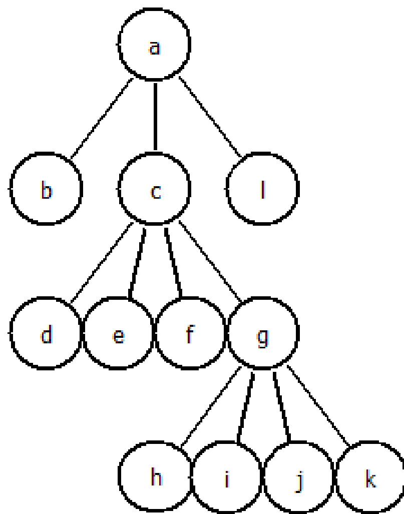
Kun luodaan uusi solmu operaatiolla makeset(x), luodaan uusi yhden solmun puu. Puun ainoasta solmusta tulee puun juuri. Solmulle luodaan osoitin alkioon x, vanhempaan ja kaksisuuntaiseen linkitettyyn listaan lapsista. Juuren osoitin vanhempaan on juuri itse, eli p(x) = x. Lisäksi puulle luodaan kaksi listaa puun solmuista: syvyysjärjestyslista ja sisäsolmulista. Syvyysjärjestyslista on kaksisuuntainen renkaaksi linkitetty lista puun kaikista alkioista syvyysjärjestyksessä. Sisäsolmulista on kaksisuuntainen linkitetty lista sisäsolmuista, jotka ovat juuren lapsia. Juuri on syvyysjärjestyslistan ainoa solmu. Lapsilista ja sisäsolmulista ovat tyhjiä.

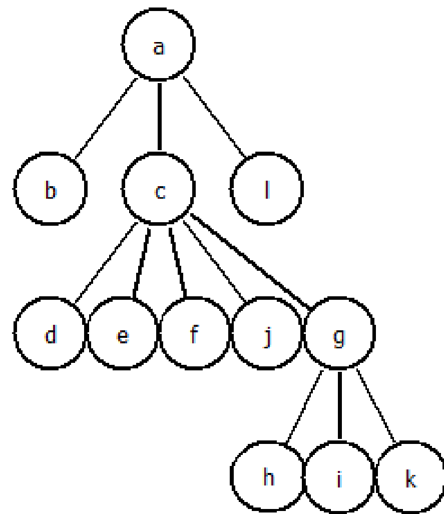
Hakuoperaatio find(x) etenee solmusta x vanhempiosoittimia pitkin ylöspäin puussa ja nostaa samalla solmuja ylemmäs puussa käyttämällä halkaisua. Tämä lyhentää hakupolkuja ja nopeuttaa tulevia hakuja. Halkaisussa solmun vanhemmaksi tulee sen isovanhempi, eli jokainen hakupolulla oleva solmu (ja sen alipuu) nousee puussa yhden tason ylemmäs. Hakuoperaatio tarkistaa halkaisun yhteydessä, jääkö ylemmäs nostettavan solmun vanhemmalle alle kolme lapsisolmua, ja jos näin on, nostaa myös ne yhtä tasoa ylemmäs. Näin puu pysyy täyden puun määritelmän mukaisena, eli jokaisella solmulla, joka ei ole lehti, on vähintään kolme lasta.

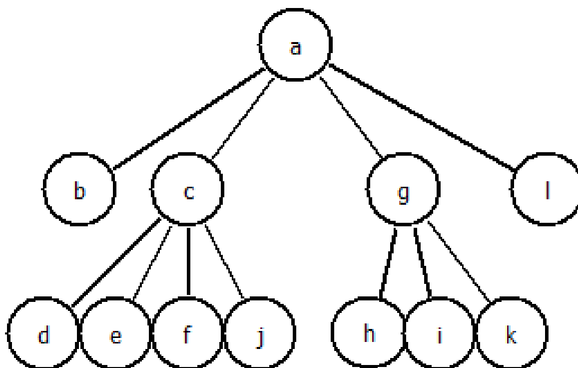
Algoritmissa käytetään funktiota relink(x), joka nostaa solmun x yhtä tasoa ylemmäs puussa. Funktiota relink(x) käytetään hakuoperaation yhteydessä hakupolun halkaisuun ja poisto-operaation yhteydessä puun tasapainoisena pitämiseen. Relink(x) nostaa solmun x vanhempansa oikeanpuoleiseksi tai vasemmanpuoleiseksi sisarukseksi riippuen siitä, onko solmulla x oikeanpuoleista sisarusta:

1. Jos solmulla x on oikeanpuoleinen sisarus, solmusta x tulee vanhempansa vasemmanpuoleinen sisarus, eli se nostetaan isovanhempansa lapsilistaan ennen vanhempaansa.
2. Jos solmulla x ei ole oikeanpuoleista sisarusta, solmusta x tulee vanhempansa oikeanpuoleinen sisarus, eli se nostetaan isovanhempansa lapsilistaan vanhempansa perään. Tässä tapauksessa puun syvyysjärjestys ei muutu, joten syvyysjärjestyslistaa ei tarvitse muuttaa.

Funktio relink(x) muuttaa solmun x osoittimen vanhempaan, poistaa solmun vanhempansa lapsilistasta, lisää solmun isovanhempansa lapsilistaan ennen tai jälkeen vanhempaansa ja päivittää syvyysjärjestyslistaa tarvittaessa.







Kuva 7: Ben-Amramin ja Yoffen algoritmin funktio relink(x).

Kuvassa 7 on esimerkki funktion relink toiminnasta. Kuvassa ei selkeyden vuoksi ole kaikkia osoittimia, vaan vain viivat, jotka osoittavat lapsi–vanhempi-suhteet. Kuvan kohdassa (1) on puun alkutilanne, jolle tehdään operaatio find(j). Hakuoperaatio nostaa

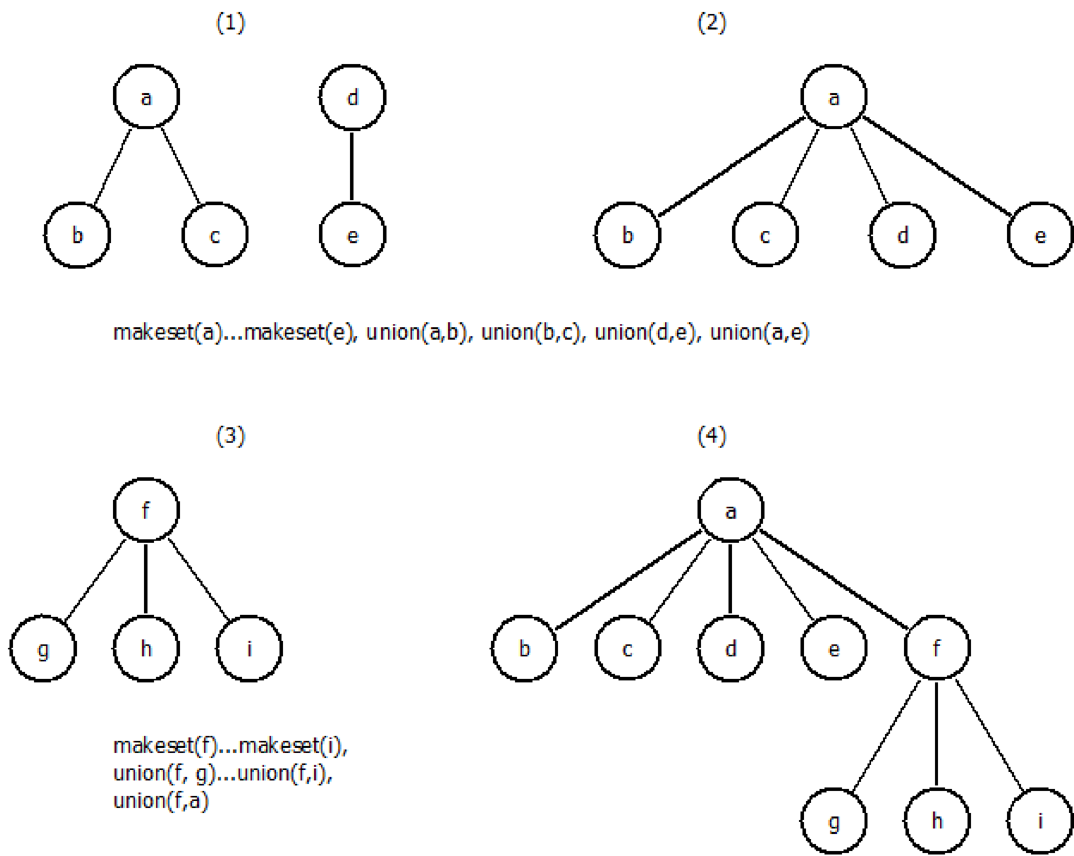
kaikki hakupolulla olevat solmut isovanhempansa lapseksi. Tässä tapauksessa tehdään operaatiot relink(j) ja relink(g). Solmulla c ei ole isovanhempaa, joten nostot päättyvät tähän. Solmulla j on oikeanpuoleinen sisarus, joten se nostetaan isovanhempansa lapsilistaan ennen vanhempaansa. Tämä muuttaa syvyysjärjestystä, joten syvyysjärjestyslistaa on päivitettävä. Seuraavaksi nostetaan solmun j entinen vanhempi g. Solmulla g ei ole oikeanpuoleista sisarusta, joten se nostetaan vanhempansa oikealle puolelle. Nyt syvyysjärjestys ei muutu, joten syvyysjärjestyslistaa ei tarvitse muuttaa. Lopuksi haku palauttaa solmun a.

Yhdistäminen union-operaatiolla tehdään puiden juurien arvojen mukaan. Yhdistämisoperaatio on hieman erilainen täysille ja vajaille puille. Jos molemmat puut ovat vajaita, toinen puista valitaan satunnaisesti uuden puun juureksi. Toisen puun kaikki solmut lisätään toisen puun juuren lapsiksi. Jos toinen puista on vajaa, vajaan puun kaikki solmut lisätään täyden puun juuren lapsiksi. Jos molemmat puut ovat täysiä, juuri, jolla on pienempi arvo, lisätään suuremman arvoisen juuren lapseksi. Jos täydet puut ovat samanarvoisia, toinen puiden juurista valitaan satunnaisesti uuden puun juureksi. Tämän jälkeen uuden puun juuren listat päivitetään vastaamaan muuttunutta tilannetta. Toisen puun syvyysjärjestyslista lisätään uuden juuren syvyysjärjestyslistaan. Uuden juuren lapsilistaan lisätään toisen puun juuri, tai vajaan puun tapauksessa kaikki puun alkiot. Lisäksi, jos molemmat puut olivat täysiä, toisen puun juuri lisätään uuden juuren sisäsolmulistaan. Lopuksi uuden juuren arvoa kasvatetaan yhdellä, jos yhdistettävien puiden arvot olivat samat.

Kuvassa 8 on esimerkkejä yhdistämisoperaatiosta. Kuvassa ei selkeyden vuoksi ole kaikkia osoittimia, vaan vain viivat, jotka osoittavat solmujen lapsi–vanhempi-suhteet. Kuvan kohdassa (1) on kaksi vajaata puuta. Kohdassa (2) vajaat puut on yhdistetty operaatiolla union(a, e). Yhdistämisoperaatio tekee molemmille alkioille hakuoperaation, joten ei ole väliä, vaikka yhdistämisoperaatiolle annetaan lehtialkio. Toisen puun kaikki alkiot siirretään suoraan juuren lapsiksi, koska molemmat puut olivat vajaita. Yhdistämisen jälkeen puu on täysi. Kohdassa (3) luodaan toinen täysi puu ja yhdistetään se puun a kanssa. Nyt molemmat puut ovat täysiä, ja molempien puiden arvo on 1. Satunnaisesta juuresta, tässä a, tulee uusi juuri. Toinen puu asetetaan uuden juuren lapseksi sellaisenaan. Kohdassa (4) on yhdistetty puu. Kohdan (4) puulla on nyt yksi solmu sisäsolmulistassa, solmu f.

Poisto-operaatio delete(x) tarkistaa ensin, onko poistettava alkio lehtisolmu. Lehtisolmun poistaminen on suoraviivaista. Solmu poistetaan ensin syvyysjärjestyslistasta ja vanhempansa lapsisolmulistasta. Tämän jälkeen solmun alkio ja kaikki osoittimet poistetaan. Jos poistettava solmu ei ole lehti, etsitään puusta jokin lehtisolmu ja vaihdetaan poistettavan solmun alkio ja lehtisolmun alkio keskenään. Lopuksi poistetaan lehtisolmu.

Kun vajaasta puusta poistetaan solmu, poistettava solmu on joko juuri tai lehti. Jos poistettava solmu on juuri, vaihdetaan juuren alkion ja jonkin sen lapsisolmun alkion paikkaa keskenään, ja poistetaan lehtisolmu.

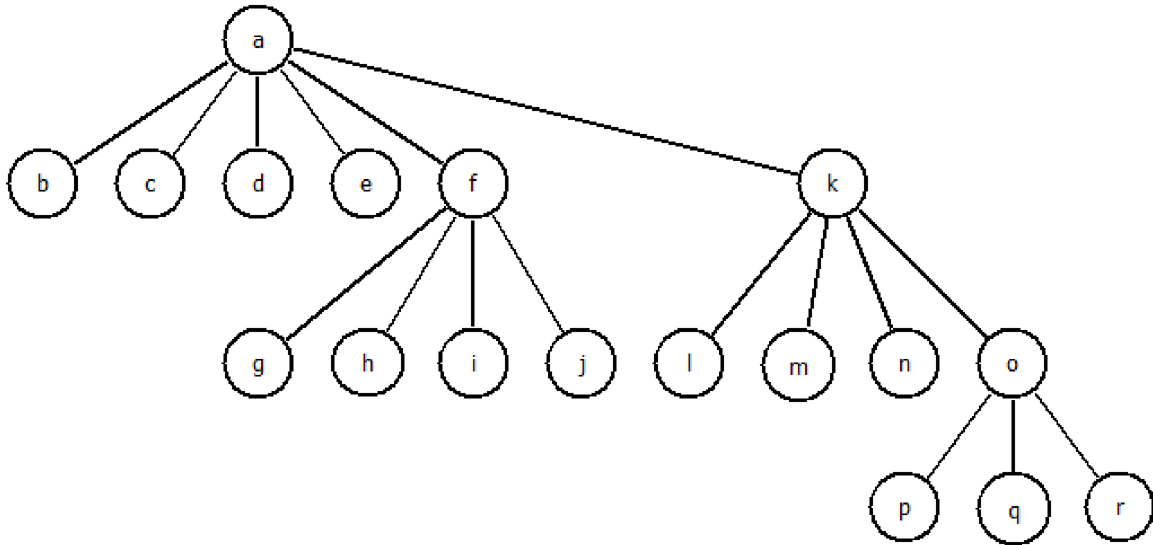


Kuva 8: Union-operaatioita Ben-Amramin ja Yoffen algoritmilla.

Kun täydestä puusta poistetaan solmu, joka ei ole lehti, syvyysjärjestyslistan avulla löydetään helposti jokin lehtisolmu jollakin seuraavista ehdoista:

1. Jos poistettava alkio on juuri, syvyysjärjestyslistassa juurta edeltävä solmu on lehtisolmu. Poistetaan kuvan 9 puusta solmu *a*, joka on juuri. Syvyysjärjestyslistassa juurta edeltävä solmu on *r*, joka on lehtisolmu. Juuren ja löydetyn lehden alkiot vaihdetaan keskenään, ja lehti poistetaan. Poiston jälkeen puun juuri on *r* ja solmulla *o* on jäljellä kaksi lasta.
2. Jos solmulla on oikeanpuoleinen sisarus, sisarusta edeltävä solmu syvyysjärjestyslistassa on lehtisolmu. Poistetaan kuvan 9 puusta solmu *f*, jolla on oikeanpuoleinen sisarus *k*. Solmua *k* edeltää syvyysjärjestyslistassa solmu *j*. Solmu *j* on solmun *f* alipuun oikeanpuoleisin lehti. Alkioiden *f* ja *j* paikkaa vaihdetaan keskenään ja lehtisolmu poistetaan. Nyt solmun *f* tilalla on *j*. Solmulla *j* on kolme lasta, ja sen oikeanpuoleinen sisarus on edelleen solmu *k*.
3. Jos solmulla ei ole oikeanpuoleista sisarusta eikä solmu ole juuri, poistettavaa solmua edeltävä solmu syvyysjärjestyslistassa on lehtisolmu. Poistetaan kuvan 9 puusta solmu *k*. Haetaan syvyysjärjestyslistasta solmua *k* edeltävä alkio, joka

on *j*. Alkiot *k* ja *j* vaihdetaan keskenään ja lehtisolmu poistetaan. Puussa solmun *a* viimeinen lapsi on nyt *j*.



Kuva 9: Esimerkkipuu Ben-Amramin ja Yoffen algoritmista. Puun solmujen aakkosjärjestys on sama kuin solmujen syvyysjärjestys.

Poiston jälkeen tehdään paikallisia tasoitustoimenpiteitä, jotka nostavat solmuja ylemmäs puussa. Tämä tehdään, jotta puu pysyisi tasapainoisena, eli täyden puun määritelmän mukaisena. Jos poistettava alkio on *x*, ja poistetun alkion vanhempi on *y*, tehdään toinen seuraavista tasoitustoimenpiteestä:

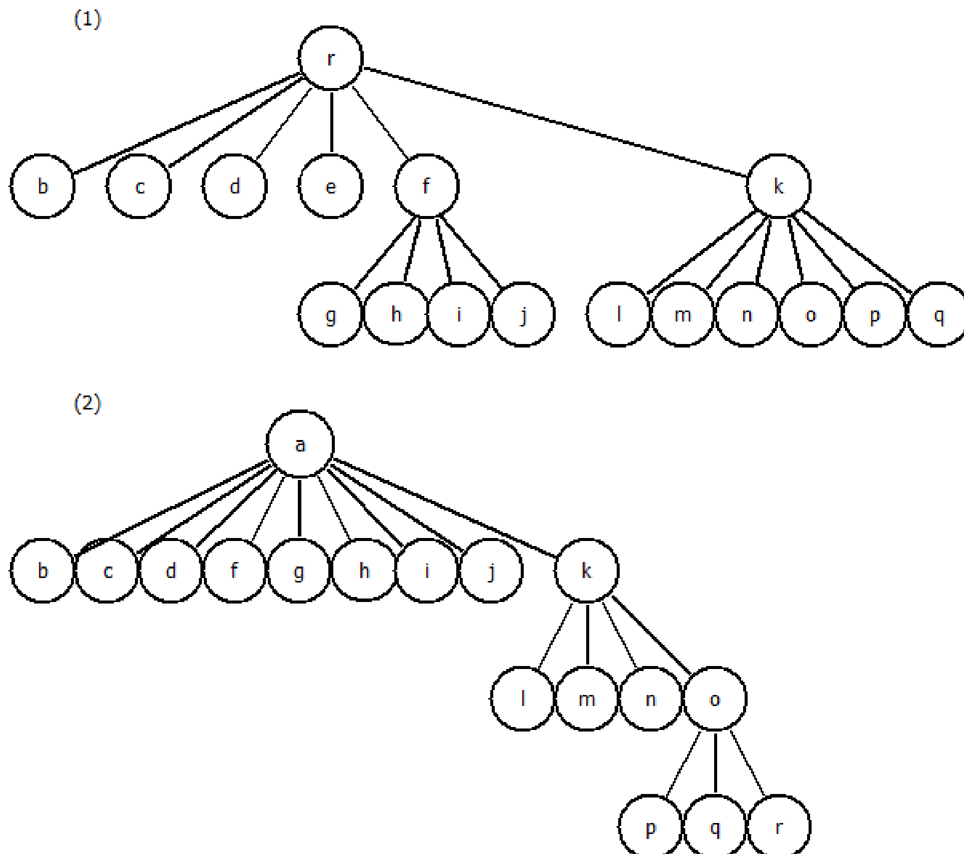
1. Jos *y* ei ole juuri, nostetaan solmun *y* kaksi oikeanpuoleisinta lasta solmun *y* vanhemman lapsiksi. Jos solmulle *y* jäisi tämän jälkeen alle kolme lasta, nostetaan loputkin solmun *y* lapset tasoa ylemmäs.
2. Jos *y* on juuri, etsitään puun sisäsolmulistasta solmun *y* jokin lapsi *z*. Sisäsolmulistassa ovat sellaiset juuren lapsisolmut, jotka eivät ole lehtiä. Nostetaan solmun *z* kolme oikeanpuoleisinta lasta yhtä tasoa ylemmäs, eli juuren lapsiksi. Jos solmulle *z* jäisi tämän jälkeen alle kolme lasta, nostetaan loputkin solmun *z* lapset tasoa ylemmäs ja poistetaan *z* sisäsolmulistasta. Jos sisäsolmulista tyhjeni, puun kaikki solmut ovat suoraan juuren lapsia, eikä solmuja voi nostaa ylemmäs puussa.

Solmun nostaminen ylemmäs ei erota solmua alipuustaan, vaan nostettavan solmun jälkeläiset nousevat solmun mukana. Nostettavat lapsisolmut ovat aina lapsilistan viimeisiä solmuja, joten kun sellainen nostetaan funktiolla relink, solmu nostetaan isovanhempansa lapsilistaan vanhemman oikealle puolelle. Tällöin syvyysjärjestyslistaa ei tarvitse muuttaa.

Kuvan 10 kohdassa (1) on tilanne sen jälkeen, kun kuvan 9 puusta on poistettu solmu *a*. Ensin on haettu lähin lehti *r*, toiseksi vaihdettu alkioita, kolmanneksi poistettu lehtisolmu ja lopuksi tehty paikallisia tasoitustoimenpiteitä. Paikalliset tasoitustoimenpiteet tehdään

siis solmulle *o*, joka on poistetun solmun vanhempi. Molemmat solmun *o* lapset on nostettu tasoa ylemmäs solmun *o* sisaruksiksi.

Kuvan 10 kohdassa (2) on tilanne sen jälkeen, kun kuvan 9 puusta on poistettu solmu *e*. Solmun *e* on lehti, joten poistaminen on suoraviivaista. Tasoitustoimenpiteet tehdään juureen, koska *e* oli suoraan juuren lapsi. Sisäsolmulistasta on haettu solmu *f* ja nostettu sen kaikki lapset tasoa ylemmäs.



Kuva 10: Solmun *a* tai *e* poistaminen Ben-Amramin ja Yoffen algoritmissa.

3.6. Aidon poiston algoritmi

Ajatus aidon poiston algoritmista tuli Erkki Mäkiseltä [2012] tutkimusprojektikurssilta, jolla käsiteltiin union-find-delete-algoritmeja. Aidon poiston algoritmi toimii samoin kuin vajaan poiston algoritmi, mutta lisäksi jokaisella solmulla on tieto lapsisolmuista. Lapsisolmut tallennetaan kaksisuuntaiseen linkitettyyn listaan. Jokaisella solmulla on osoitin alkioon, vanhempaansa, lapsilistan ensimmäiseen solmuun sekä vasemmanpuoleiseen ja oikeanpuoleiseen sisarukseensa.

Hakuoperaatio kulkee puussa ylemmäs vanhempiosoittimia pitkin ja palauttaa puun juuren. Hakuoperaatio lyhentää hakupolkua käyttämällä halkaisua.

Puiden yhdistäminen tapahtuu arvon perusteella samoin kuin kahdessa muussakin algoritmissa. Ensin haetaan yhdistettävien puiden juuret hakuoperaatiolla. Juuri, jolla on suurempi arvo, on uuden puun juuri. Toisen puun juuri lisätään uuden juuren lapsilistaan,

ja sen vanhemmaksi asetetaan uusi juuri. Jos juurien arvot ovat samat, toinen juuri lisätään toisen lapseksi ja uuden juuren arvoa kasvatetaan yhdellä.

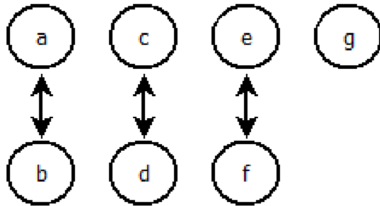
Poisto-operaatio delete(x) poistaa sekä alkion että solmun ja kaikki solmuun liittyvät osoittimet. Jos poistettava alkio on juuri, sen ensimmäinen lapsi nostetaan uudeksi juureksi. Solmun x lasten vanhempiosoittimet päivitetään osoittamaan uuteen juureen. Vanhan juuren lapsilista lisätään solmun x lapsilistaan. Jos solmu x ei ole juuri, sen lapset siirretään sen vanhemman lapsiksi. Solmun x lasten vanhempiosoittimet päivitetään osoittamaan uuteen vanhempaan. Solmun x lapsilista lisätään uuden vanhemman lapsilistaan.

Aidon poiston algoritmissa poistaminen on työlästä, koska lapsisolmujen vanhempiosoittimia joudutaan muuttamaan. Pahimmassa tapauksessa puu on mahdollisimman matala ja poistettava alkio on juuri, jolloin joudutaan päivittämään kaikkien puun solmujen tietoja.

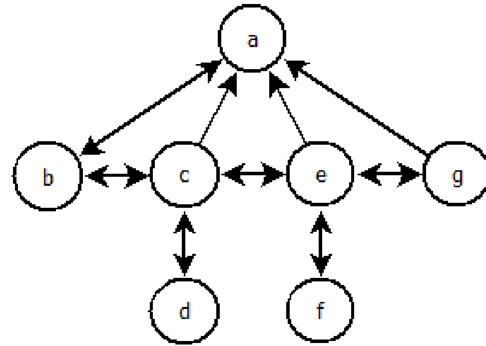
Hieman tehokkaampi tapa tehdä poisto olisi seurata poistettavasta alkiosta lähtien lapsiosoittimia alaspäin puussa, kunnes tullaan lehtisolmuun. Tämän jälkeen lehtisolmun ja poistettavan solmun alkiot vaihdettaisiin keskenään, ja poistettaisiin sitten lehtisolmu. Nyt pahimmassakin tapauksessa jouduttaisiin käymään läpi vain puun korkeuden verran solmuja.

Kuvassa 11 on esimerkkipuita aidon poiston algoritmista. Kuvassa ympyrät ovat solmuja ja nuolet osoittimia. Solmujen sisältämät alkiot ovat selkeyden vuoksi solmujen sisällä, vaikka todellisuudessa solmusta on osoitin myös alkioon. Kuvan kohdassa (1) on ensin luotu seitsemän yhden alkion puuta, ja sitten yhdistetty niitä kahden alkion puiksi. Kohdassa (2) nämä pienet puut on yhdistetty yhdeksi suuremmaksi. Kohdassa (3) on poistettu alkio c, jolloin sen lapset on nostettu isovanhempansa lapsiksi. Solmun c ainoa lapsi d on lisätty juuren viimeiseksi lapseksi. Kohdassa (4) on poistettu juuri a. Solmun a ensimmäisestä lapsesta b on tullut uusi juuri. Solmun a lapset on lisätty solmun b lapsilistaan. Jos solmulla b olisi ollut lapsia jo ennestään, ne olisivat lapsilistassa ennen solmun a lapsia.

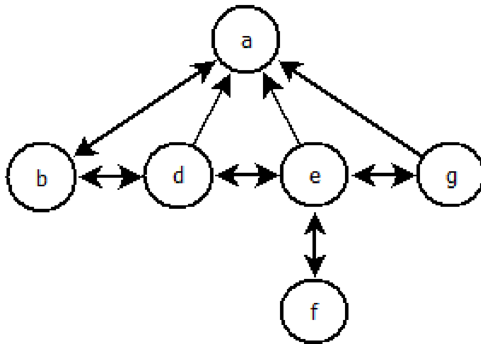
(1)
makeset(a)...makeset(g),
union(a,b), union(c,d), union(e,f)



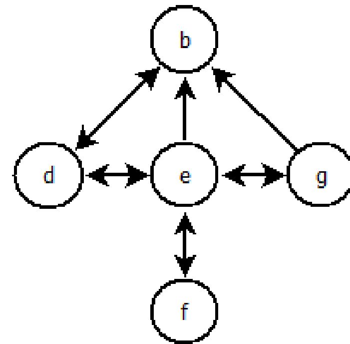
(2)
union(a,c), union(a,e), union(a,g)



(3)
delete(c)



(4)
delete(a)



Kuva 11: Operaatioita aidon poiston algoritmissa.

4. Algoritmien teoreettinen tehokkuus

Tässä luvussa vertailen algoritmien teoreettista tehokkuutta. Ensimmäisessä kohdassa kerron yleisesti aika- ja tilavaatimuksista ja niiden merkintätavoista. Toisessa kohdassa kerron erillisten joukkojen käsittelyongelman naiivien ratkaisujen ja union-find-algoritmin aikavaatimuksista. Kolmannessa kohdassa kerron toteuttamieni kolmen union-find-delete-algoritmin aikavaatimuksista.

4.1. Asymptoottisista aikavaatimuksista ja merkintätavoista

Merkinnöillä $O(g)$, $\Theta(g)$ ja $\Omega(g)$ kuvataan algoritmien asymptoottista aikavaatimusta. Näillä merkinnöillä annetaan algoritmin kasvun tyyppi, jossa kertoimia ei huomioida. Merkinnällä $O(g)$ annetaan algoritmin kasvun yläraja ja merkinnällä $\Omega(g)$ alaraja. Merkintä $\Theta(g)$ rajaa algoritmia sekä ylhäältä että alhaalta. [Cormen *et al.*, 1996]

Joistakin esittelemistäni algoritmeista on tiedossa myös algoritmin tasoitettu aikavaatimus. Tasoitetuissa aikavaatimuksissa käytetään Ackermannin funktion käänteisfunktiota, jonka määrittelyn annan tässä. Ackermannin funktio on räjähdysmäisen nopeasti kasvava funktio, jonka määritelmä on:

$$A(m, n) = \begin{cases} n + 1 & \text{jos } m = 0 \\ A(m-1, 1) & \text{jos } m > 0 \text{ ja } n = 0 \\ A(m-1, A(m, n-1)) & \text{jos } m > 0 \text{ ja } n > 0. \end{cases}$$

Ackermannin funktion käänteisfunktio taas kasvaa äärimmäisen hitaasti. Sen määritelmä on:

$$\alpha(m, n) = \min \{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}.$$

4.2. Erillisten joukkojen käsittelyongelman naiivit ratkaisut

Mäkisen ja Porasen [2011] esittelemän taulukkoratkaisun hakuoperaatio voidaan tehdä vakioajassa $O(1)$, mutta yhdistäminen on tehotonta. Joukko on esitetty taulukkona $R[1\ldots n]$, ja i on jokin taulukon alkio. Hakuoperaatio find(i) voidaan suorittaa vakioajassa, sillä haku palauttaa kyseisen taulukon, eli joukon. Yhdistämisoperaatio union(R, S) sen sijaan joutuu päivittämään molempien joukkojen kaikkien alkioiden tietoja. Taulukkoratkaisun pahimman tapauksen aikavaatimus on siis $O(n^2)$.

Cormenin ja muiden [1996] listaratkaisussa joukko on linkitetty lista. Lisäksi jokaisella alkiolla on osoitin joukon nimenä toimivaan ensimmäiseen alkioon. Hakuoperaation aikavaatimus on $O(1)$. Yhdistäminen sen sijaan on tehotonta, ja sen aikavaatimus on $O(n^2)$. Yhdistämisoperaatio lisää toisen listan toisen perään. Tämän jälkeen päivitetään toisen listan kaikkien alkioiden osoitin nimialkioon viittaamaan uuden listan nimialkioon. Algoritmin asymptoottinen aikavaatimus on $O(m^2)$. Algoritmin tasoitettu aikavaatimus on $\Theta(m)$, jossa m on kaikkien makeset-, find- ja union-operaatioiden määrä.

Cormen ja muut [1996] parantavat algoritmia tekemällä yhdistämisen koon mukaan siten, että pienempi lista lisätään aina suuremman listan perään. Nyt yhdistämisoperaation aikavaatimus on $\Omega(n)$. Tämän algoritmin aikavaatimus on $O(m + n \log n)$.

4.3. Union-find-algoritmien aikavaatimukset

Tarjan ja van Leeuwen [1984] analysoivat erillisten joukkojen käsittelyongelman puuratkaisun eli union-find-algoritmin ratkaisujen asymptoottisia aikavaatimuksia. Naiivi ratkaisu yhdistää puita satunnaisesti, eikä hakupolkuja lyhennetä hakujen yhteydessä. Tarjanin ja van Leeuwenin mukaan parhaat parannukset tehokkuuteen saadaan, kun puita yhdistetään koon tai arvon mukaan, kun hakupolkuja lyhennetään hakujen yhteydessä. Tarjan ja van Leeuwen esittelevät myös muita tehostamistapoja, mutta ne ovat pahimman tapauksen analyysissä huonompia kuin mainitut tavat.

Union-find-algoritmissa luomisen ja yhdistämisen aikavaatimus on $O(1)$. Yhdistäminen voidaan tehdä vakioajassa, koska toinen puu liitetään toisen juuren lapseksi, eikä muita osoittimia tarvitse päivittää. Ainoa muutettava osoitin on toisen puun juuren vanhempiosoitin.

Haun aikavaatimus riippuu hakupolulla olevien alkioiden määrästä. Kun puita yhdistetään satunnaisesti, pahimmassa tapauksessa kaikki alkiot ovat puussa listamaisesti allekkain, eli hakupolulla on n alkiota. Haun asymptoottinen aikavaatimus on siis $O(n)$. Naiivin algoritmin asymptoottinen aikavaatimus on $O(n + nm) = O(nm)$, jossa m on hakuoperaatioiden määrä ja n alkioiden määrä.

Ensimmäinen Tarjanin ja van Leeuwenin [1984] esittelemä tapa lisätä tehokkuutta lyhentämällä hakupolkuja on tehdä yhdistäminen puun koon tai arvon mukaan. Pienempi puu lisätään suuremman puun juuren lapseksi, tai pienempiarvoinen juuri lisätään suurempiarvoisen juuren lapseksi. Tästä seuraa, että hakupolun pituus on enintään $\log_2 n$. Kun yhdistetään koon tai arvon mukaan, algoritmin asymptoottinen aikavaatimus on $\Theta(n + m \log n)$.

Toinen Tarjanin ja van Leeuwenin [1984] käyttämä tapa lyhentää hakupolkuja on puiden muokkaaminen hakujen yhteydessä. Kun suoritetaan operaatiota find(x), hakupolkua tiivistetään, puolitetaan tai halkaistaan matkalla kohti juurta. Näistä esimerkiksi tiivistäminen on työläämpää kuin kaksi muuta, koska kaikki hakupolun alkiot siirretään juuren lapsiksi ja tästä syystä hakupolku joudutaan käymään läpi kahdesti. Toisaalta juuri siksi tiivistäminen tehostaa tulevia hakuja parhaiten, koska alkiot siirretään mahdollisimman lähelle juurta. Puolitus ja halkaisu tehdään paikallisilla operaatioilla, joten ne voidaan tehdä samalla, kun hakupolkua edetään ylemmäs. Kaikki kolme tapaa ovat kuitenkin yhtä tehokkaita, kun hakuoperaatioita tehdään paljon. Kun hakuoperaatioita tehdään vähän (kun $m < n$, eli hakuoperaatioita on vähemmän kuin alkioita), hakupolun lyhennystapojen aikavaatimuksissa on eroja. Kun käytetään jotain näistä kolmesta

hakupolun lyhennystavasta ja hakuja tehdään paljon, algoritmin asymptoottinen aikavaatimus on $\Theta(m \log_{(1+m/n)} n)$.

Tehokkaimmin algoritmi toimii, kun hakuoperaatioiden yhteydessä käytetään jotain hakupolun lyhennysmetodia ja puiden yhdistäminen tehdään koon tai arvon perusteella. Kun näitä tehostustapoja käytetään yhdessä, algoritmin aikavaatimukseksi saadaan $\Theta(m\ \alpha(m,n))$, kun $m \geq n$, tai $\Theta(n + m\ \alpha(n,n))$, kun $m < n$, joissa $\alpha()$ on Ackermannin funktion käänteisfunktio. Mikä tahansa yhdistelmä näistä kahdesta yhdistämistavasta ja hakupolkujen lyhennystavasta tekee algoritmista asymptoottisesti optimaalisen. [Tarjan and van Leeuwen, 1984]

Tarjanin ja van Leeuwenin [1984] analyysin mukaan splicing, eli yhdistäminen punomalla, on asymptoottisen aikavaatimuksen kannalta huonompi vaihtoehto verrattuna union-find-algoritmiin, kun käytetään yhdistämistä arvon mukaan ja jotain hakupolun lyhennystapaa. Kuitenkin Patwaryn ja muiden [2010] kokeellisen testauksen mukaan splicing toimii tehokkaammin kuin mikään muu union-find-algoritmi, huolimatta heikommasta asymptoottisesta aikavaatimuksesta.

Taulukoissa 1 ja 2 on koottu yhteen Tarjanin ja van Leeuwenin [1984] analysoimien algoritmien aikavaatimukset. Taulukossa 1 ovat aikavaatimukset, kun hakuoperaatioita tehdään paljon (kun $m \geq n$). Taulukossa 2 aikavaatimukset, kun hakuoperaatioita tehdään suhteellisen vähän (kun $m < n$).

	Naiivi yhdistäminen	Yhdistäminen koon tai arvon mukaan
Naiivi haku	$\Theta(mn)$	$\Theta(m \log n)$
Tiivistäminen	$\Theta(m \log_{(1+m/n)} n)$	$\Theta(m\ \alpha(m,n))$
Halkaisu	$\Theta(m \log_{(1+m/n)} n)$	$\Theta(m\ \alpha(m,n))$
Puolitus	$\Theta(m \log_{(1+m/n)} n)$	$\Theta(m\ \alpha(m,n))$
Splicing	$\Theta(m \log_{(1+m/n)} n)$	-

Taulukko 1: Union-find-algoritmin aikavaatimukset pahimmassa tapauksessa, kun $m \geq n$.

	Naiivi yhdistäminen	Yhdistäminen koon tai arvon mukaan
Naiivi haku	$\Theta(mn)$	$\Theta(n + m \log n)$
Tiivistäminen	$\Theta(n + m \log n)$	$\Theta(n + m\ \alpha(n,n))$
Halkaisu	$\Theta(n \log m)$	$\Theta(n + m\ \alpha(n,n))$
Puolitus	$\Omega(n + m \log n)$, $O(n \log m)$	$\Theta(n + m\ \alpha(n,n))$
Splicing	$\Theta(n + m \log m)$	-

Taulukko 2: Union-find-algoritmin aikavaatimukset pahimmassa tapauksessa, kun $m < n$.

Smidin [1990] union-find-algoritmissa hakuoperaation aikavaatimus on $O(\log n / \log k)$ ja yhdistämisen aikavaatimus on $O(k)$, jossa n on alkioiden lukumäärä ja k on jokin kiinnitetty vakioarvo. Arvo k määrää algoritmissa, montako lasta puun solmulla tulee olla, jotta se olisi ”täysi”. Puun solmuilla saattaa olla solmuja enemmänkin kuin k kappaletta, mutta yhdistämisoperaatiot pyrkivät saamaan solmujen lapsimääräksi vähintään k. Jos valitaan $k = 2$, Smidin algoritmin aikavaatimus on sama kuin union-find-algoritmin perusversiossa, kun yhdistäminen tehdään koon tai arvon mukaan. Suuremmilla k:n arvoilla algoritmi on tehokkaampi. Smidin algoritmin asymptoottiseksi aikavaatimukseksi saadaan siis $O(m \log_k n)$, jossa m on operaatioiden määrä.

Smidin algoritmin tilavaatimus on suhteessa alkioiden määrään, eli $O(n)$. Algoritmissa kaikki alkiot ovat puun lehtisolmuissa, joten puussa on aina ”ylimääräisiä” solmuja, jotka eivät sisällä alkioita. Tällaisia alkiottomia solmuja on kuitenkin aina vähemmän kuin lehtisolmuja. Solmujen kokonaislukumäärä on siis pienempi kuin $2n$, jossa n on alkioiden lukumäärä. Jokaisella solmulla on vakiomäärä osoittimia muihin solmuihin.

Tässä esiteltyjen puutietorakenteeseen perustuvien algoritmien (Smidin union-find-algoritmi ja union-find-algoritmi, jossa yhdistäminen tehdään arvon tai koon mukaan ja hakupolkuja lyhennetään hakujen yhteydessä) asymptoottiset aikavaatimukset ovat samaa luokkaa. Molempien aikavaatimusten kasvu on logaritmista. Jos Smidin algoritmissa valitaan suuri k:n arvo, sen tehokkuus on parempi kuin union-find-algoritmin perusversiolla.

Smidin algoritmin tilavaatimus on hieman suurempi kuin union-find-algoritmin perusversion, koska union-find-algoritmi tallentaa vain osoittimet vanhempiin, kun taas Smidin

algoritmissa solmulla on tieto myös lapsistaan. Smidin algoritmin tilavaatimus on kuitenkin vain vakiokertoimen verran suurempi, ja molempien algoritmien tilavaatimus on lineaarinen.

4.4. Union-find-delete-algoritmien aika- ja tilavaatimus

Vajaan poiston algoritmin luonti-, poisto- ja yhdistämisoperaatio voidaan tehdä vakioajassa $O(1)$. Poisto-operaatio poistaa vain alkion ja jättää puuhun onton solmun, jossa on edelleen osoitin vanhempaan. Poisto ei hae puun juurta, vaan se on paikallinen operaatio. Haun aikavaatimus riippuu puun korkeudesta. Koska puut yhdistetään arvon perusteella ja hakupolkuja halkaistaan haun yhteydessä, hakuoperaation aikavaatimus on sama kuin union-find-algoritmilla, eli $O(\log \mathrm{N})$, jossa N on solmujen lukumäärä, eli luontioperaatioiden määrä. Koko algoritmin asymptoottinen aikavaatimus on siis $O(m \log \mathrm{N})$, jossa m on operaatioiden määrä. Tehokkuus ei riipu alkioiden määrästä n, vaan solmujen luontioperaatioiden määrästä N, koska solmujen määrä puussa ei koskaan vähene. Algoritmin tilavaatimus on $O(\mathrm{N})$.

Smidin algoritmiin lisätyn poisto-operaation aikavaatimus riippuu puun korkeudesta. Yksittäinen rekursiivisen funktion delete′ kutsu voidaan tehdä vakioajassa, mutta kutsukertojen määrä riippuu puun korkeudesta. Poisto-operaation aikavaatimus on siis $O(\log_k n)$, jossa k on jokin vakioarvo, [Kaplan *et al.*, 2002]. Algoritmin aikavaatimus on $O(m \log_k n)$. Algoritmin tilavaatimus on $O(n)$, koska alkioiden poistaminen tuhoaa poistetun solmun kokonaan, ja alkioita sisältämättömiä sisäsolmuja on aina vähemmän kuin alkioita.

Kaplanin ja muiden [2002] inkrementaalisella kopioinnilla toteutetun poiston aikavaatimus riippuu union-find-algoritmin toteutuksesta. Haun, yhdistämisen ja luomisen aikavaatimukset ovat samat kuin union-find-algoritmilla. Poiston aikavaatimus on $O(t_f(n) + t_i(n))$, eli haun aikavaatimus ja insert-operaation eli lisäämisen aikavaatimus yhteenlaskettuna. Jos käytetään union-find-algoritmin perusversiota, jossa yhdistetään arvon mukaan ja hakupolkuja lyhennetään hakujen yhteydessä, saadaan poiston aikavaatimukseksi $O(\log n)$. Algoritmin aikavaatimus on siis $O(m \log n)$. Algoritmin tilavaatimus on lineaarinen, koska onttoja solmuja on aina alle puolet kaikista solmuista. Joukko on jaettu kahteen puuhun S_n ja S_o, joista joukossa S_n onttoja solmuja on enintään ¼ ja joukossa S_o onttoja solmuja on enintään ½.

Alstrupin ja muiden [2014] union-find-delete-algoritmissa luonti- ja yhdistämisoperaatio voidaan tehdä vakioajassa. Hakuoperaation pahimman tapauksen aikavaatimus $O(\log n)$ ja tasoitettu aikavaatimus on $O(\alpha_{\lceil M/N \rceil}(n))$. Poiston aikavaatimus on $O(1)$. Koko algoritmin aikavaatimus on siis $O(m \log n)$.

Ben-Amramin ja Yoffen algoritmilla operaatioiden aikavaatimukset ovat samat kuin union-find-algoritmilla, jossa yhdistäminen tehdään arvon perusteella ja hakujen yhteydessä hakupolkua halkaistaan. Haun asymptoottinen aikavaatimus on $O(\log n)$ ja tasoitettu aikavaatimus on $O(\alpha(n))$, jossa α on Ackermannin funktion käänteisfunktio [Ben-

Amram and Yoffe, 2011]. Yhdistäminen ja poisto voidaan tehdä vakioajassa. Poisto vajaasta puusta voidaan tehdä vakioajassa, koska vajaassa puussa on joko vain juuri tai juuren kaikki jälkeläiset ovat suoraan juuren lapsia. Lehtisolmun poistaminen onnistuu vakioajassa. Jos poistettava alkio on juuri, jokin lehtisolmu löytyy aina vakioajassa hakemalla juuren jokin lapsi. Jos taas puu on täysi, lehtisolmun poistaminen onnistuu edelleen vakioajassa. Jos täydestä puusta ollaan poistamassa sisäsolmua, jokin lehtisolmu löydetään syvyysjärjestyslistan ja lapsisolmulistojen avulla vakioajassa. Näin myös poisto täydestä puusta onnistuu vakioajassa. Myös poistojen yhteydessä tehtävät tasoitus-operaatiot suoritetaan vakioajassa. Ben-Amramin ja Yoffen algoritmin aikavaatimus on siis sama kuin union-find-algoritmin asymptoottinen aikavaatimus eli $O(m \log n)$, ja tasoitettu aikavaatimus on $O(\alpha(n))$. Algoritmin tilavaatimus on $O(n)$.

Aidon poiston algoritmin operaatioiden aikavaatimukset ovat samat kuin union-find-algoritmilla. Luonti- ja yhdistämisoperaatio tehdään vakioajassa. Haun aikavaatimus riippuu puun korkeudesta, eli molempien pahimman tapauksen aikavaatimus on $O(\log n)$. Poisto-operaation aikavaatimus on $O(n)$. Pahimmassa tapauksessa puu on mahdollisimman matala, eli kaikki n alkiota ovat suoraan juuren lapsia, ja poistettava alkio on juuri. Poisto-operaatio nostaa juuren ensimmäisen lapsen uudeksi juureksi. Nyt joudutaan päivittämään kaikkien jäljelle jääneiden $n–1$ alkion vanhempiosoittimet, poistamaan uusi juuri lapsilistasta ja lisäämään uuteen juureen osoitin lapsilistaan. Yhteensä tehdään siis $(n–1)+2+1$ operaatiota, eli poiston aikavaatimus on $O(n)$. Aidon poiston algoritmin asymptoottinen aikavaatimus on siis $O(mn)$. Aidon poiston algoritmin tilavaatimus on $O(n)$. Aidon poiston algoritmissa jokaisella puun alkiolla on vakiomäärä osoittimia muihin alkioihin, ja alkion poistaminen tuhoaa sekä alkion itsensä että nämä osoittimet.

Taulukkoon 3 on koottu tässä työssä esiteltyjen union-find-delete-algoritmien aikavaatimukset. Taulukossa riveinä ovat esitellyt algoritmit. Kahdessa ensimmäisessä sarakkeessa ovat algoritmien asymptoottiset aikavaatimukset ja tasoitetut aikavaatimukset. Lopuissa neljässä sarakkeessa ovat yksittäisten operaatioiden aikavaatimukset. Kaikkien algoritmien tasoitettua aikavaatimusta ei ole esitelty kirjallisuudessa. Nämä on merkitty taulukossa viivalla.

Näistä ratkaisuista kaikkien paitsi aidon poiston algoritmin asymptoottinen aikavaatimus on logaritminen. Kaikkien algoritmien yhdistäminen, haku ja luominen toimivat keskenään yhtä tehokkaasti. Erot algoritmien välillä näkyvät poisto-operaation sarakkeessa. Parhaiten toimivat algoritmit ovat Alstrupin ja muiden algoritmi ja siitä johdettu Ben-Amramin ja Yoffen algoritmi, joissa poisto tehdään vakioajassa. Samoin vajaan poiston algoritmissa poisto tehdään vakioajassa, mutta tässä hakuoperaation tehokkuus on heikompi, koska se riippuu solmujen lukumäärästä, eikä alkioiden lukumäärästä, kuten muissa algoritmeissa.

Näistä algoritmeista vain Alstrupin ja muiden algoritmista ja Ben-Amramin ja Yoffen algoritmista tiedetään algoritmin tasoitettu aikavaatimus. Kokeellisessa vertailussa ovat

mukana Ben-Amramin ja Yoffen algoritmin lisäksi vajaan poiston algoritmi ja aidon poiston algoritmi. Kokeellisella vertailulla pyrin tutkimaan, millaiset näiden kahden muun algoritmin tasoitetut aikavaatimukset olisivat.

	Asymptoottinen aikavaatimus	Tasoitettu aikavaatimus	Union	Find	Delete	Makeset
Vajaan poiston algoritmi	$O(m \log N)$	-	$O(1)$	$O(\log N)$	$O(1)$	$O(1)$
Smidin algoritmi poistolla	$O(m \log_k n)$	-	$O(k)$	$O(\log_k n)$	$O(\log_k n)$	$O(1)$
Inkrementaalinen kopiointi	$O(m \log n)$	-	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Alstrupin ja muiden algoritmi	$O(m \log n)$	$O(m\ \alpha_{\lceil M/N \rceil}(n))$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
Ben-Amramin ja Yoffen algoritmi	$O(m \log n)$	$O(m\ \alpha(m,n))$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
Aidon poiston algoritmi	$O(mn)$	-	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$

Taulukko 3: Union-find-delete algoritmien aikavaatimuksia.

5. Algoritmien kokeellinen vertailu

Tässä luvussa esittelen tekemääni kokeellista tutkimusta. Vertailen kolmea eri toteutusta union-find-delete-algoritmista. Toteutin luvussa 3 esitellyistä union-find-delete-algoritmeista vajaan poiston algoritmin, aidon poiston algoritmin ja Ben-Amramin ja Yoffen algoritmin. Toteutin algoritmit Java-ohjelmointikielellä.

Ben-Amramin ja Yoffen algoritmi käyttää hakupolun lyhennystapana halkaisua ja yhdistää puut arvon mukaan. Käytän siksi samoja tehostustapoja myös kahdessa muussa toteuttamassani algoritmissa. Algoritmeissa solmu ja alkio eivät ole sama asia. Puun jokaisesta solmusta on osoitin sitä vastaavaan alkioon.

Vertailtaviksi algoritmeiksi valikoituivat Ben-Amramin ja Yoffen algoritmi, aidon poiston algoritmi ja vajaan poisto algoritmi. Näistä aidon poiston algoritmi ja vajaan poiston algoritmi ovat hyvin helppo toteuttaa. Ben-Amramin ja Yoffen algoritmi on monimutkaisempi.

Aidon poiston algoritmin asymptoottinen aikavaatimus on huono, kun taas kaksi muuta algoritmia ovat asymptoottiselta aikavaatimukseltaan erittäin tehokkaita. Aidon poiston algoritmin tasoitettu aikavaatimus ei ole tiedossa, joten on kiinnostavaa vertailla algoritmeja kokeellisesti. Tutkimukseni lähtöajatus olikin vertailla, pärjääkö yksinkertaisempi aidon poiston algoritmi vertailussa asymptoottiselta aikavaatimukseltaan paremmalle Ben-Amramin ja Yoffen algoritmille, ja toisaalta vertailla, pärjääkö enemmän tilaa vievä Ben-Amramin ja Yoffen algoritmi tilavaatimusvertailussa kahdelle muulle algoritmille, joiden rakenne on hyvin yksinkertainen. Vertailen näitä algoritmeja kokeellisesti sekä aika- että tilavaatimuksen perusteella.

5.1. Algoritmien vertailu

Vertailen algoritmien aikavaatimuksia laskemalla osoittimille tehtäviä operaatioita. Lasken yksittäiseksi operaatioksi tässä osoittimen viittauksen muutoksen, osoittimen luomisen tai osoittimen poiston. Algoritmit tekevät muitakin operaatioita, mutta nämä operaatiot osoittimille tehdään algoritmeissa aina, riippumatta toteutuksesta.

Algoritmien tilavaatimuksia vertailen laskemalla osoittimia ja alkioita. Lasken alkioiden muistipaikat mukaan tilavaatimukseen, vaikka jokaisen algoritmin lopussa alkioita on yhtä monta, kun algoritmit saavat saman syötteen. Lasken alkiot mukaan, koska silloin tuloksista näkyy, että vajaan poiston algoritmin tilavaatimus pienenee hieman poiston yhteydessä, vaikka poisto-operaatio ei vapauta kaikkea solmun varaamaa muistia.

Toteuttamissani algoritmeissa alkio ja solmu eivät ole sama asia, vaan jokaisesta solmusta on osoitin alkioon. Lisäksi jokaisessa algoritmissa kaikilla solmuilla on osoitin solmun vanhempaan. Puiden juurien tapauksessa vanhempiosoitin osoittaa solmuun itseensä.

Vajaan poiston algoritmissa osoitin vanhempaan ja osoitin alkioon ovat rakenteen ainoat osoittimet. Poisto jättää jälkeensä onton solmun, jolla on edelleen osoitin vanhempaan ja *null*-osoitin alkioon. Itse alkio poistetaan, ja sen muistipaikka vapautuu.

Aidon poiston algoritmissa jokaisella solmulla on samoin osoittimet vanhempaan ja alkioon. Lisäksi solmulla on osoitin lapsilistaan. Lapsilista on linkitetty lista, eli jokaisella lapsisolmulla on osoitin lapsilistan seuraavaan solmuun, eli oikeanpuoleiseen sisarukseen. Aidon poiston algoritmissa jokainen solmu tarvitsee tilaa neljälle osoittimelle ja alkiolle.

Ben-Amramin ja Yoffen algoritmissa osoittimia on enemmän kuin kahdessa muussa algoritmissa. Jokaisella solmulla on osoitin alkioon, vanhempaan, lapsilistaan, oikeanpuoleiseen ja vasemmanpuoleiseen sisarukseen sekä osoitin puun syvyysjärjestyksessä edelliseen ja seuraavaan solmuun. Lisäksi puulla saattaa olla yksisuuntainen linkitetty sisäsolmulista. Sisäsolmulistassa ovat juuren sellaiset lapset, jotka ovat sisäsolmuja. Jokaisella sisäsolmulistan solmulla on osoitin seuraavaan sisäsolmulistan alkioon. Lisäksi juurella on osoitin sisäsolmulistaan. Yhteensä yhden solmun tilavaatimus on siis seitsemän osoitinta ja alkio. Lisäksi jokaisella puulla on osoitin syvyysjärjestyslistaan ja sisäsolmulistaan.

5.2. Testisyötteet

Testaan algoritmejani sekä satunnaisilla testisyötteillä että keinotekoisilla syötteillä. Satunnaisissa testeissä vaihtelen poisto-operaation todennäköisyyttä eri syötteissä. Keinotekoiset testit ovat pahimman tapauksen testejä, joissa kaikki alkiot poistetaan, mutta poistojärjestys vaihtelee.

5.2.1. Satunnaiset testit

Testaan algoritmeja näennäissatunnaisilla syötteillä, joissa esiintyy luonti-, yhdistys-, haku- ja poisto-operaatioita tietyllä todennäköisyydellä.

Muutan eri syötteissä poisto-operaatioiden todennäköisyyttä. Muiden operaatioiden todennäköisyydet ovat keskenään samat. Kaikissa satunnaisissa syötteissä luodaan aluksi 100 kappaletta yhden alkion joukkoja luontioperaatiolla. Tämän jälkeen tehdään kaikkia neljää operaatiota satunnaisesti. Käytän satunnaislukugeneraattorina Javan Math.random()-metodia. Syötearpojani arpoo ensin todennäköisyyksien mukaan, mikä operaatio tehdään ja sen jälkeen mihin alkioihin operaatio kohdistetaan. Satunnaissyötteen koko on 20 000 operaatiota.

Testiaineistoissani poisto-operaation todennäköisyys on 0–25 %. Jokaisessa testiaineistossa poiston todennäköisyys on viisi prosenttiyksikköä suurempi kuin edellisessä. Yhdessä testiaineistossa on neljä kappaletta 20 100 operaation (100 luontioperaatiota ja 20 000 satunnaista operaatiota) syötettä. Suurin poiston todennäköisyys on 25 %, koska tätä suuremmilla poiston todennäköisyydellä on hyvin todennäköistä, että kaikki alkiot poistetaan. Tämä johtuu siitä, että luontioperaation esiintymistodennäköisyys olisi silloin pienempi kuin poisto-operaation todennäköisyys. Testiaineistojen operaatioiden todennäköisyydet ovat taulukossa 4.

	Poiston todennäköisyys	Luomisen todennäköisyys	Yhdistämisen todennäköisyys	Haun todennäköisyys
Testiaineisto 1	0 %	0,333 %	0,333 %	0,333 %
Testiaineisto 2	5 %	31,666 %	31,666 %	31,666 %
Testiaineisto 3	10 %	30 %	30 %	30 %
Testiaineisto 4	15 %	28,333 %	28,333 %	28,333 %
Testiaineisto 5	20 %	26,666 %	26,666 %	26,666 %
Testiaineisto 6	25 %	25 %	25 %	25 %

Taulukko 4: Operaatioiden todennäköisyydet satunnaisissa testiaineistoissa.

5.2.2. Keinotekoiset testit

Testaan algoritmeja myös keinotekoisilla testeillä. Testeissä kaikki alkiot yhdistetään samaan puuhun, ja sen jälkeen kaikki alkiot poistetaan. Testit eroavat toisistaan poistettavien alkioiden järjestyksessä.

Ben-Amramin ja Yoffen algoritmin pahin tapaus on tilanne, jossa puun kaikki alkiot poistetaan, ja poistettava alkio on aina alkio, jolla on lapsia. Lehtisolmun ja sisäsolmun poistamisessa ei kuitenkaan ole suurta eroa. Kun poistettava alkio on sisäsolmussa, joudutaan etsimään jokin lehtisolmu ja vaihtamaan alkioiden paikkaa, minkä jälkeen lehtisolmu poistetaan.

Aidon poiston algoritmin pahin tapaus on tilanne, jossa kaikki alkiot poistetaan puusta, joka on mahdollisimman matala, ja poistettava alkio on aina juuri. Mahdollisimman matalassa puussa kaikki puun lapset ovat suoraan juuren lapsia.

Vajaan poiston algoritmin pahin tapaus on yksinkertaisesti kaikkien alkioiden poistaminen. Poistojärjestyksellä ei ole merkitystä, koska juuren poistaminen toimii samalla tavalla kuin muidenkin solmujen. Juuren tilalle ei voida etsiä solmua, joka ei ole ontto, koska juurella ei ole tietoa jälkeläisistään eikä puussa voida liikkua alaspäin kohti lehtiä.

Testiaineistoissa luodaan aluksi 5 000 yhden solmun puuta. Tämän jälkeen puut yhdistetään yhdeksi matalaksi puuksi, jossa kaikki puun solmut ovat suoraan juuren lapsia. Lopuksi kaikki alkiot poistetaan. Pahimmassa tapauksessa poistettava alkio on aina juuri. Testeissä pienennän vähitellen juuren poistojen määrää verrattuna muiden solmujen poistojen määrään.

Aidon poiston algoritmin pahin tapaus, eli matala puu, josta poistetaan aina juuri, on yksi mahdollinen pahin tapaus Ben-Amramin ja Yoffen algoritmille. Tämä on myös vajaan poiston algoritmin pahin tapaus, koska kaikki alkiot poistetaan.

Ensimmäisessä testissä poistettava alkio on aina juuri. Muissa testeissä poistettava alkio on juuri joka toinen/neljäs/kahdeksas/kahdeskymmenes/neljäskymmenes kerta ja muutoin jokin muu alkio. Viimeisessä testissä poistettava alkio ei ole koskaan juuri (paitsi viimeinen alkio, joka on yksinäinen solmu).

5.3. Testien tulokset

Alakohdissa 5.3.1. ja 5.3.2. esittelen satunnaisten testien ja keinotekoisten testien tulokset. Alakohdassa 5.3.3. analysoin saamiani tuloksia.

5.3.1. Satunnaisten testien tulokset

Satunnaisten syötteiden tulokset ovat taulukoissa 5 ja 6. Taulukossa 5 on algoritmien aikavaatimus ja taulukossa 6 algoritmien tilavaatimus. Taulukoiden ensimmäisessä sarakkeessa on poiston esiintyvyystodennäköisyys syötteessä. Muissa sarakkeissa ovat algoritmien mukaan lajitellut aika- ja tilavaatimukset. Taulukot 5 ja 6 sisältävät samojen syötteiden tulokset samassa järjestyksessä.

Kuvissa 12 ja 13 ovat taulukoiden 5 ja 6 keskiarvojen kuvaajat. Keskiarvot on laskettu siten, että saman poiston todennäköisyyden syötteistä on laskettu keskiarvo. Kuvan 12 kaavion pystyakselilla on operaatioiden määrä. Samoin kuvan 13 kaavion pystyakselilla on tilayksiköiden määrä. Kaavioiden vaaka-akselilla on syötteiden poiston todennäköisyys.

Vajaan poiston algoritmi näyttää aikavaatimuksia vertailemalla olevan algoritmeista tehokkain. Sen aikavaatimus on kaikilla satunnaisilla syötteillä pienin. Aidon poiston algoritmi on lähes yhtä tehokas, koska sen aikavaatimus on vähän alle kaksinkertainen verrattuna vajaan poiston algoritmiin. Ben-Amramin ja Yoffen algoritmin aikavaatimus on vähän yli kaksinkertainen verrattuna aidon poiston algoritmiin. Vajaan poiston algoritmiin verrattuna Ben-Amramin ja Yoffen algoritmin aikavaatimus on noin nelinkertainen.

Aidon poiston algoritmi on satunnaisilla syötteillä myös tilavaatimuksen osalta parempi kuin Ben-Amramin ja Yoffen algoritmi. Aidon poiston algoritmin tilavaatimus on noin puolet Ben-Amramin ja Yoffen algoritmin tilavaatimuksesta.

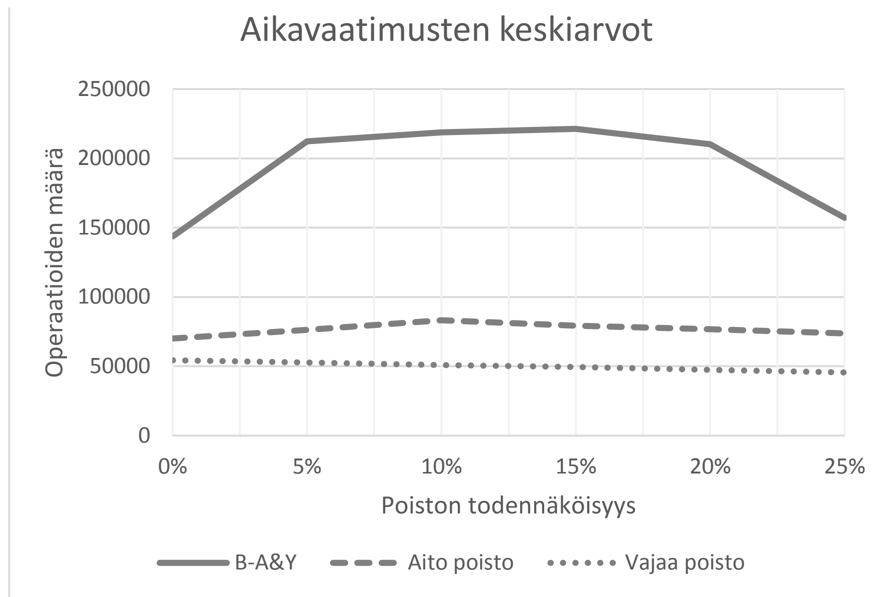
Vajaan poiston algoritmin heikko kohta on tilavaatimuksen jatkuva kasvaminen. Tämä näkyy kuvasta 13, jossa ovat algoritmien tilavaatimukset, kun syötteen kaikki operaatiot on tehty. Kun poiston todennäköisyys on suuri, lähes kaikki alkiot poistetaan. Kun lähes kaikki alkiot poistetaan, aidon poiston algoritmin ja Ben-Amramin ja Yoffen algoritmin tilavaatimus lähenee nollaa. Vajaan poiston algoritmin tilavaatimus pienenee, mutta se ei koskaan vapauta kaikkea varaamaansa muistia.

Poiston todennäköisyys	Ben-Amram ja Yoffe	Aito poisto	Vajaa poisto
0 %	137879	69781	54158
0 %	145083	69580	54328
0 %	147969	70377	54408
5 %	216614	78210	53005
5 %	211481	80281	52654
5 %	208449	70316	52290
10 %	224771	83272	50931
10 %	210894	79707	50629
10 %	220172	86795	51111
15 %	227867	90752	49431
15 %	223052	72937	49317
15 %	212673	74231	49368
20 %	222170	80200	47637
20 %	200799	76784	46860
20 %	207449	73023	47356
25 %	157563	73535	45323
25 %	163140	74501	45577
25 %	150095	73283	45712

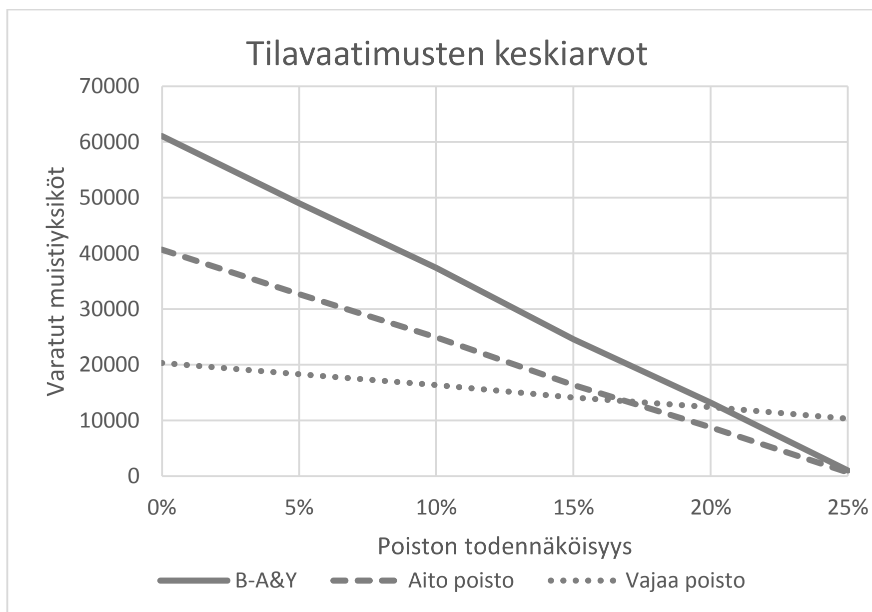
Taulukko 5: Algoritmien aikavaatimukset satunnaisilla syötteillä.

Poiston todennäköisyys	Ben-Amram ja Yoffe	Aito poisto	Vajaa poisto
0 %	60732	40488	20244
0 %	60669	40446	20223
0 %	61704	41136	20568
5 %	48969	32646	18281
5 %	49158	32772	18338
5 %	48825	32550	18313
10 %	37368	24912	16360
10 %	37512	25008	16436
10 %	37314	24876	16362
15 %	24066	16044	14058
15 %	25758	17172	14378
15 %	23895	15930	13945
20 %	13158	8772	12342
20 %	13365	8910	12457
20 %	13005	8670	12395
25 %	1161	774	10431
25 %	1188	792	10348
25 %	783	522	10227

Taulukko 6: Algoritmien tilavaatimukset satunnaisilla syötteillä.



Kuva 12: Satunnaisten syötteiden aikavaatimusten keskiarvot.



Kuva 13: Satunnaisten syötteiden tilavaatimusten keskiarvot.

5.3.2. Keinotekoisten testien tulokset

Keinotekoiset testit noudattavat teoreettisia pahimman tapauksen aikavaatimuksia. Pahimmassa tapauksessa, kun kaikki alkiot poistetaan ja poistettava alkio on aina juuri, Ben-Amramin ja Yoffen algoritmi on huomattavasti tehokkaampi kuin aidon poiston algoritmi. Vajaan poiston algoritmi taas näyttää toimivan vielä moninkertaisesti tehokkaammin kuin Ben-Amramin ja Yoffen algoritmi.

Taulukossa 7 ovat algoritmien aikavaatimukset, kun kaikki alkiot poistetaan. Ensimmäisen sarakkeen prosenttiluku kertoo, kuinka suuri osa poistoista osuu juureen. Ben-

Amramin ja Yoffen algoritmin aikavaatimus vaihtelee aineistossa vain vähän. Sen aikavaatimus on melkein sama, kun kaikki poistot osuvat juureen tai kun kaikki poistot osuvat muihin alkioihin. Aidon poiston algoritmin aikavaatimus pahimmassa tapauksessa sen sijaan on valtava, lähes 80-kertainen, verrattuna Ben-Amramin ja Yoffen algoritmiin. Vajaan poiston algoritmissa ei ole merkitystä, missä järjestyksessä alkiot poistetaan, joten sen aikavaatimus on näissä testeissä aina sama. Ben-Amramin ja Yoffen algoritmin aikavaatimus on noin viisinkertainen verrattuna vajaan poiston algoritmiin. Vajaan poiston algoritmi toimii siis tehokkaammin, mutta ero ei ole kovin suuri.

Ben-Amramin ja Yoffen algoritmin aikavaatimus taulukossa 7 pienenee hieman, sitä mukaa kun poistoista pienempi osa osuu juureen. Tämä johtuu siitä, että kun poistettava alkio on lehti, poistaminen on suoraviivaista, mutta kun poistettava alkio on sisäsolmu, joudutaan lisäksi etsimään jokin lehtisolmu ja vaihtamaan alkioiden paikkaa keskenään. Ero näiden kahden tapauksen välillä on kuitenkin mitättömän pieni.

Aidon poiston aikavaatimus pienenee nopeasti, kun suurempi osa poistoista osuu ei-juuriin. Kuitenkin vielä silloinkin, kun joka neljäskymmenes poisto osuu juureen, aidon poiston algoritmin aikavaatimus on suurempi kuin Ben-Amramin ja Yoffen algoritmin aikavaatimus.

Vertailun vuoksi taulukon 7 viimeisellä rivillä on myös tapaus, jossa edelleen kaikki alkiot poistetaan, mutta poistettava alkio on aina lehti. Tässä tapauksessa aidon poiston algoritmin aikavaatimus on pienempi kuin Ben-Amramin ja Yoffen algoritmilla, kuten satunnaisissakin testeissä. Aikavaatimusten erot tässä viimeisessä testissä ovat lähes samanlaisia kuin satunnaisissa testeissä.

Vajaan poiston algoritmin heikkous näkyy taulukossa 9, jossa ovat tilavaatimukset sen jälkeen, kun kaikki alkiot on poistettu. Vajaan poiston algoritmi varaa edelleen muistia kahden osoittimen verran jokaista luontioperaatiota kohti, kun taas muut algoritmit eivät varaa ollenkaan muistia, kun kaikki alkiot on poistettu.

Poistoista juuria	Ben-Amram & Yoffe	Aito poisto	Vajaa poisto
kaikki	159 975	12 547 498	29 997
1/2	152 486	6 304 996	29 997
1/4	148 744	3 183 746	29 997
1/8	146 857	1 623 121	29 997
1/20	145 732	686 746	29 997
1/40	145 357	374 621	29 997
ei yhtään	144 982	59 996	29 997

Taulukko 7: Aikavaatimus, kun kaikki alkiot poistetaan. Poistettava alkio on joko juuri tai ei-juuri.

Poistoista juuria	Ben-Amram & Yoffe	Aito poisto	Vajaa poisto
0–100 %	0	0	10 000

Taulukko 8: Tilavaatimus, kun kaikki alkiot poistetaan.

Kaikissa näissä testeissä poistetaan kaikki alkiot, joten Ben-Amramin ja Yoffen algoritmin ja aidon poiston algoritmin tilavaatimus on lopussa nolla. Vajaan poiston algoritmi sen sijaan jättää aina jälkeensä osoittimia, joten sen tilavaatimus lopussa on kaksi kertaa luontioperaatioiden määrä, eli tässä tapauksessa 10 000. Pahimmassa tapauksessa vajaan poiston algoritmi jättää jälkeensä varattua muistia, vaikka kaikki alkiot on poistettu.

Keinotekoiset testit eivät kuvasta todellisuutta kovin hyvin, koska on hyvin epätodennäköistä, että poisto-operaatio osuisi usein nimenomaan juureen tai muuhun alkioon, jolla on paljon lapsia. Toisaalta testeissä käytetty tilanne, jossa kaikki puun alkiot ovat suoraan juuren lapsia, vaikuttaa hyvinkin todennäköiseltä tilanteelta myös todellisissa käyttötilanteissa. Puu muokkautuu nopeasti tämän muotoiseksi, kun hakuja tehdään paljon.

5.4. Testien tulosten analysointia

Keinotekoisten pahimman tapauksen testien tulokset vahvistavat teoreettisia aikavaatimuksia. Toisaalta satunnaisten testien mukaan algoritmit toimivat kaikki lähes yhtä tehokkaasti.

Keinotekoisissa testeissä näkyy selvästi, että pahimmassa tapauksessa Ben-Amramin ja Yoffen algoritmi ja vajaan poiston algoritmi ovat parempia kuin aidon poiston algoritmi. Pahimmassa tapauksessa aidon poiston algoritmi kuluttaa paljon aikaa solmujen vanhempiosoittimien päivittämiseen. Sen sijaan satunnaisten testien perusteella aidon poiston algoritmi näyttäisi toimivan jopa paremmin kuin Ben-Amramin ja Yoffen algoritmi. Ben-Amramin ja Yoffen algoritmin aikavaatimus satunnaisissa testeissä on vähän yli kaksinkertainen verrattuna aidon poiston algoritmiin. Tämä johtuu siitä, että Ben-Amramin ja Yoffen algoritmissa on enemmän osoittimia toisiin alkioihin, jolloin useampia osoittimia joudutaan muuttamaan, kun solmu poistetaan, tai jos sen paikka vaihtuu. Aidon poiston algoritmin tapauksessa lehtisolmun poistaminen on nopeampaa kuin minkä tahansa Ben-Amramin ja Yoffen algoritmin alkion poistaminen. Aidon poiston algoritmin pahin tapaus on todella epätodennäköinen, joten sen vaikutukset tasoittuvat, kun operaatioita tehdään satunnaisesti.

Alkion poistaminen aidon poiston algoritmilla on työläintä silloin, kun poistettava alkio on alkio, jolla on enemmän kuin yksi lapsi. Kun poistettava alkio on lehti tai yhden alkion puu, poistaminen tehdään vakioajassa. Ben-Amramin ja Yoffen algoritmin tapauksessa ei ole väliä, onko poistettavalla alkiolla lapsia vai ei: poistaminen tehdään aina vakioajassa. Kun alkioita poistetaan satunnaisesti, on todennäköisempää, että poisto-operaatio osuu alkioon, jolla on enintään yksi lapsi, kuin alkioon, jolla on useita lapsia. Tämä johtuu siitä,

että puussa on monilapsisia solmuja aina vähemmän kuin yhden lapsen solmuja tai lehtisolmuja. Tästä johtuen aidon poiston algoritmi toimii keskimäärin yhtä hyvin kuin Ben-Amramin ja Yoffen algoritmi.

Molemmat testaustavat vahvistavat teoreettisia tilavaatimuksia. Ben-Amramin ja Yoffen algoritmin ja aidon poiston algoritmin tilavaatimukset ovat lineaarisia, eli suoraan verrannollisia alkioiden määrään. Ben-Amramin ja Yoffen algoritmi käyttää enemmän muistia kuin aidon poiston algoritmi, koska algoritmi pitää yllä useita listoja puun alkioista, jolloin osoittimia tallennetaan enemmän. Vajaan poiston algoritmin tilavaatimus on suhteessa luontioperaatioiden määrään. Algoritmi varaa edelleen muistia, vaikka kaikki alkiot poistetaan. Aineistossa, jossa tapahtuu paljon muutoksia, eli alkioita poistetaan ja lisätään usein, vajaan poiston algoritmin tilavaatimus kasvaa jatkuvasti.

Kuvassa 13 aidon poiston algoritmin ja Ben-Amramin ja Yoffen algoritmin tilavaatimuskäyrät lähestyvät nollaa, kun poiston todennäköisyys on suuri, jolloin lähes kaikki alkiot poistetaan. Vajaan poiston algoritmin tilavaatimus ylittää kahden muun algoritmin tilavaatimuksen, kun poistoja tehdään paljon. Vaikka poistoja tehtäisiin vain harvoin, vajaan poiston algoritmin tilavaatimus kasvaa jossain vaiheessa suuremmaksi kuin kahden muun algoritmin tilavaatimus, kun algoritmia ajetaan pitkään.

Vajaan poiston algoritmin tilavaatimus ylittää muiden algoritmien tilavaatimuksen jossain vaiheessa, kun algoritmia ajetaan riittävän pitkään. Jos poistoja tehdään vain vähän tai jos algoritmia käytetään vain vähän aikaa, vajaan poiston algoritmi toimii hyvin. Jos algoritmi on käytössä pitkään tai jos alkioita poistetaan ja luodaan usein, paremmin toimiva ratkaisu on jokin sellainen union-find-delete-algoritmi, joka käyttää muistia suhteessa alkioiden määrään. Testieni perusteella vaikuttaisi siltä, että aidon poiston algoritmi olisi paras näistä kolmesta union-find-delete-algoritmista. Sen keskimääräinen aikavaatimus on pienempi kuin Ben-Amramin ja Yoffen algoritmissa ja se on yksinkertaisempi toteuttaa.

Aidon poiston algoritmin poisto-operaation aikavaatimus on $O(n)$, koska kun poistetaan ei-lehtisolmu, poistettavan solmun ensimmäisestä lapsesta tulee uusi juuri ja poistettavan alkion kaikkien muiden lasten vanhempiosoittimet on päivitettävä. Poiston asymptoottisesta aikavaatimuksesta saisi paremman, jos poisto toteutettaisiin samantyyppisesti kuin Ben-Amramin ja Yoffen algoritmissa, eli etsimällä jokin lehtisolmu ja vaihtamalla sitten poistettavan solmun alkio lehtisolmun alkion kanssa ja poistamalla lopuksi lehtisolmu. Poistettavasta solmusta kuljettaisiin puussa alaspäin aina solmun ensimmäiseen lapseen, kunnes löydettäisiin lehtisolmu. Tällä toteutuksella poisto-operaation aikavaatimus riippuisi suoraan puun korkeudesta, eli poiston aikavaatimukseksi saataisiin $O(\log n)$. Todellisuudessa hakupolku olisi vieläkin lyhyempi, koska puiden yhdistäminen lisää puun juuren lapsilistan viimeiseksi alkioksi. Kun alaspäin kuljettaisiin aina ensimmäiseen lapseen, olisi polkukin lyhempi. Olisikin kiinnostavaa vertailla kokeellisesti, miten tämä muutos vaikuttaisi ajoaikoihin.

6. Yhteenveto

Tässä tutkielmassa vertailin erilaisia erillisten joukkojen käsittelyongelmaan kehitettyjä algoritmeja. Vertailin näitä algoritmeja teoreettisesti. Toteutin kolme algoritmia, joissa on myös poiston mahdollisuus, ja vertailin niitä kokeellisesti. Kokeelliseen testaukseen valitsemani algoritmit ovat asymptoottisten aikavaatimusten perusteella helposti laitettavissa paremmuusjärjestykseen. Algoritmien tasoitettua aikavaatimusta ei kuitenkaan ole analysoitu kuin yhdestä toteuttamastani algoritmista. Kokeellisella vertailulla pyrin selvittämään, mikä algoritmeista on keskimäärin paras, kun sitä käytetään satunnaisilla syötteillä.

Toteuttamani algoritmit olivat Ben-Amramin ja Yoffen algoritmi, aidon poiston algoritmi ja vajaan poiston algoritmi. Näistä Ben-Amramin ja Yoffen algoritmin asymptoottinen aikavaatimus on parempi kuin kahden muun algoritmin. Tekemäni pahimman tapauksen testit vahvistavat tämän. Aidon poiston algoritmin poisto-operaation asymptoottinen aikavaatimus on $O(n)$, kun taas Ben-Amramin ja Yoffen algoritmin ja vajaan poiston algoritmin tapauksessa poiston aikavaatimus on $O(1)$.

Tekemieni satunnaisten testiaineistojen perusteella näyttää siltä, että kaikki kolme algoritmia toimivat keskimäärin yhtä tehokkaasti. Algoritmien aikavaatimukset erosivat testieni perusteella toisistaan vain vakiokertoimen verran.

Kokeellisten aikavaatimusten perusteella algoritmien paremmuusjärjestys olisi: vajaan poiston algoritmi, aidon poiston algoritmi ja viimeisenä Ben-Amramin ja Yoffen algoritmi. Vajaan poiston algoritmin huono puoli on sen muistinvaraus, joka ei koskaan pienene nollaan, vaikka alkioita poistettaisiin. Ben-Amramin ja Yoffen algoritmin aikavaatimus oli testeissäni vähän yli kaksinkertainen verrattuna aidon poiston algoritmiin, ja aidon poiston algoritmin aikavaatimus oli noin puolitoistakertainen verrattuna vajaan poiston algoritmiin.

Kokeellisissa testeissäni algoritmit ovat samassa paremmuusjärjestyksessä myös tilavaatimuksen osalta. Vajaan poiston algoritmin muistinvaraus on näin pienillä operaatiomäärillä vielä pienempi kuin kahden muun algoritmin, paitsi siinä tapauksessa, että lähes kaikki alkiot poistetaan suorituksen aikana. Vajaan poiston algoritmi kuitenkin häviää kahdelle muulle algoritmille, kun poistoja tehdään paljon. Muistinvarauksen kannalta aidon poiston algoritmi on parempi kuin Ben-Amramin ja Yoffen algoritmi, mutta ero näiden kahden algoritmin välillä on pieni.

Testieni perusteella tulin siihen tulokseen, että käyttökelpoisin vertailemistani algoritmeista on aidon poiston algoritmi. Aidon poiston algoritmi toimii keskimäärin paremmin kuin Ben-Amramin ja Yoffen algoritmi, vaikka sen asymptoottinen aikavaatimus on huonompi. Lisäksi aidon poiston algoritmi on yksinkertaisempi toteuttaa. Joissain tilanteissa myös vajaan poiston algoritmi on riittävän hyvä, huolimatta sen kasvavasta tilantarpeesta.

Jatkotutkimuskohteena olisi kiinnostavaa testata aidon poiston algoritmin muunnosta, jossa poiston asymptoottinen aikavaatimus on $O(\log n)$. Lisäksi algoritmeja voisi verrata mittaamalla niiden todellista suoritusaikaa.

Viiteluettelo

[Alstrup *et al.*, 2014] Stephen Alstrup, Mikkel Thorup, Inge Li Gørtz, Theis Rauhe, and Uri Zwick, Union-find with constant time deletions. *ACM Transactions on Algorithms* **11**, 1 (2014), Art. No. 6.

[Ben-Amram and Yoffe, 2011] Amir Ben-Amram and Simon Yoffe, A simple and efficient union-find-delete algorithm. *Theoretical Computer Science* **412**, 4–5 (2011), 487–492.

[Cormen *et al.*, 1996] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1996.

[Dijkstra, 1976] E.W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[Galler and Fisher, 1964] Bernard A. Galler and Michael J. Fisher, An improved equivalence algorithm. *Communications of the ACM* **7**, 5 (1964), 301–303.

[Goel *et al.*, 2014] Ashish Goel, Sanjeev Khanna, Daniel H. Larkin and Robert E. Tarjan, Disjoint set union with randomized linking. *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms* (2014), 1005–1017.

[Kaplan *et al.*, 2002] Haim Kaplan, Nira Shafrir and Robert E. Tarjan, Union-find with deletions. *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Mathematics* (2002), 19–28.

[Mendelson *et al.*, 2006] Ran Mendelson, Robert E. Tarjan, Mikkel Thorup and Uri Zwick, Melding priority queues. *ACM Transactions on Algorithms* **2**, 4 (2006), 535–556.

[Mäkinen, 2012] Erkki Mäkinen, henkilökohtainen tiedonanto, 2012.

[Mäkinen ja Poranen, 2011] Erkki Mäkinen ja Timo Poranen, Algoritmit. Tampereen yliopisto, Informaatiotieteiden yksikkö, Informaatiotieteiden yksikön raportteja **1/2011**, Kesäkuu 2011.

[Patwary *et al.*, 2010] Md. Mostofa Ali Patwary, Jean Blair and Fredrik Manne, Experiments on union-find algorithms for the disjoint-set data structure. *Experimental Algorithms*, Springer Berlin Heidelberg (2010), 411–423.

[Smid, 1990] M. Smid, A data structure for the union-find problem having good single-operation complexity. *ALCOM: algorithms review, Newsletter of the ESPRIT II Basic Research action program project no.* **3075**, 1, 1990.

[Tarjan, 1975] Robert E. Tarjan, Efficiency of good but not linear set union algorithm. *Journal of the ACM* **22** (1975), 215–225.

[Tarjan and van Leeuwen, 1984] Robert E. Tarjan and Jan van Leeuwen, Worst-case analysis of set union algorithms. *Journal of the ACM* **31** (1984), 245–281.