View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UEF Electronic Publications

Prioriteettijonon toteuttaminen kekorajapinnan ehdolla Dijkstran algoritmissa

Sami Hyvönen

Pro gradu -tutkielma

ITÄ-SUOMEN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tietojenkäsittelytiede

Kesäkuu 2014

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta, Kuopio
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Opiskelija, Sami Hyvönen: Prioriteettijonon toteuttaminen kekorajapinnan ehdolla Dijkstran algoritmissa
Pro gradu -tutkielma, 89 s.
Pro gradu -tutkielman ohjaajat: FT Matti Nykänen
Kesäkuu 2014

Prioriteettijono on abstrakti tietorakenne, jonka toteuttamiseen keot ovat tutkitusti hyviä. Dijkstran algoritmi vaatii minimiprioriteettijonolta kuitenkin päivitysoperaatiota, jonka toteuttamiseen pitää tietää, missä päin muistia keon solmu on. Tämä toteutetaan normaalisti käyttäen kahvoja, jotka osoittavat suoraan tarvittavaan muistipaikkaan. Tällöin kahva on tietoa, joka sijaitsee keon ulkopuolella, näin rikkoen keon rajapinnan. Tutkiemassa esitellään implisiittinen poistaminen keon rajapinnan säilyttävänä tapana päivittää informaatio ja esitellään sen vaikutuksia operaatioiden suoritusaikoihin Dijkstran algoritmissa. Tutkielmassa näytetään kahvallisten ja kahvattomien tietorakenteiden välinen ero.

Aineisto koostuu satunnais- ja täydellisen kaltaisista verkoista, joissa on tuhannesta kahdeksaan tuhatta solmua. Satunnaisverkoilla suoritusajassa tehokkain ahkera rakenne on binäärikeko ja tehokkain implisiittisen poistamisen rakenne on vasenta suosiva keko. Tiheissä täydellisissä verkoissa tehokkain ahkera rakenne on Fibonacci-keko ja tehokkain implisiittisen poiston rakenne on vasenta suosiva keko. Hakurakenteissa implisiittinen poistaminen on yleensä tehotonta.

Avainsanat: prioriteettijono, keko, rajapinta, Dijkstran algoritmi, toteuttaminen

ACM-luokat (ACM Computing Classification System, 1998 version):
E.1 [Data Structures]: Trees; F.2.2 [Analysis Of Algorithms and Problem complexity]: Nonnumerical Algorithms and Problems – sorting and searching; G.2.2 [Discrete Mathematics]: Graph Theory – graph algorithms;

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Kuopio
School of Computing
Computer Science

Student, Sami Hyvönen: Implementation of Priority Queue with Heap Interface Restriction in Dijkstra's Algorithm
Master's Thesis, 89 p.
Supervisors of the Master's Thesis: PhD Matti Nykänen
June 2014

Priority queue is an abstract data structure which is well implemented by heaps. However Dijkstra's algorithm demands from minimum priority queue an update-operation. Update-operation requires knowing where in memory the node is located. This can be implemented by handles that point directly to the relevant memory location. The handles is information outside of the heap and thus breaks the interface of the heap. This thesis presents implicit removing as a method to preserve the interface of heap when the update-operation is used and presents effects on runtime of the operations in Dijkstra's algorithm. This thesis shows differences between datastructures with and without handles.

Graph material includes random and complete graphs where count of vertices is between 1000 and 8000. In random graphs the most efficient structure is Binary heap and in implicit removing it is Leftist heap. In dense complete graphs the most efficient structure is Fibonacci heap and in implicit removing it is Leftist heap. Binary search trees are not efficient when implemented in implicit removing manner.

Keywords: priority queue, heap, interface, Dijstra's algorithm, implementation

CR Categories (ACM Computing Classification System, 1998 version):
E.1 [Data Structures]: Trees; F.2.2 [Analysis Of Algorithms and Problem complexity]: Nonnumerical Algorithms and Problems – sorting and searching; G.2.2 [Discrete Mathematics]: Graph Theory – graph algorithms;

Esipuhe

Professori Matti Nykäselle kiitos hyvästä tutkielma-aiheesta ja ohjauksesta. Matin varhaisimmat ideat olivat saada tutkielma Dijkstran algoritmin toteuttamisesta funktionaalisella ohjelmointikielellä. Aihe rajautui keskusteluissamme suorituskyvyn mittaamiseen ja mittaamisen tullessa pääaiheeksi funktionaalisuus pääsi unohtumaan. Lopulta haluttiin vertailla tietorakenteita, joita ei ole mahdollista toteuttaa tehokkaasti funktionaalisilla ohjelmointikielillä.

Kuopissa 13. kesäkuuta 2014

Sami Hyvönen

Sisältö

1 Johdanto

Dijkstran algoritmi (Dijkstra, 1959) on julkaisunsa jälkeen ollut hyvin tunnettu ja merkittävissä määrin tutkimuksen kohteena. Se tarjoaa hyvän ympäristön testata prioriteettijonojen toteutuksia (Oberhauser ja Simha, 1995), joista tehokkaimmat perustuvat kekorakenteisiin. Dijkstran algoritmi vaatii päivittämään tietoa prioriteettijonon keskeltä, mutta oppikirjoissa (Knuth, 1973; Tarjan, 1983) jätetään kekorakenteista usein huomioitta algoritmin kannalta oleellinen päivittäminen. Tästä on seurannut, ettei standardikirjastojen, kuten Java (Oracle, 2011), C (*Algorithms-library*, n.d.) tai C++ (*std::priority_queue*, n.d.), kekorakeiden toteutuksissa ole algoritmin toteuttamiseen tehokasta päivitysoperaatiota. Päivitysoperaatio kekorakenteeseen voidaan tehokkaasti toteuttaa käyttämällä kahvoja.

Funktionaalisissa ohjelmointikielissä, kuten Haskell (Peyton-Jones, 2003), kahvat ovat ongelmallisia, sillä kahvat ovat osoittimia muistipaikkoihin ja funktionaaliset ohjelmointikielet abstrahoivat muistin käsittelyn. Tutkielmassa ei esitetä funktionaalista lähestymistapaa ratkaista Dijkstran algoritmi tai kuinka algoritmi ratkaistaan standardikirjastojen avulla, vaan keskitytään vertaamaan prioriteettijonon toteutuksia rajoittumalla rajapintaan, jonka standardikirjastojen keot tyypillisesti toteuttavat.

Prioriteettijonoille esitetään toteutus, jolla Dijkstran algoritmi ratkeaisi tehokkaasti ilman päivitysoperaation käyttöä ja mitataan paljonko erilaiset toteutukset resursseja ratkaisemiseen käyttävät. Tutkielman rajaukseen kuuluvat tietorakenteet, joilla on sekä kahvallinen että kahvaton toteutus ja tästä syystä tutkielmasta on pois jätetty aidosti funktionaalisia tietorakenteita, kuten Funktionaalinen Brodal-puu (Brodal ja Okasaki, 1996). Tutkielman tavoitteena on selvittää onko kahvattomuudesta kohtuutonta haittaa Dijkstran algoritmin tapauksessa eli onko kahvattomuus este tehokkaalle funktionaaliselle Dijkstran algoritmille ja kuinka suuri ero kahvallisten ja kahvattomien prioriteettijonojen välillä on. Tutkielmassa näytetään erot sekä teoreettisesti tarkasteltuna että empiirisesti mittaamalla, kun prioriteettijono toteutetaan binäärikeolla, vasenta suosivalla keolla, AVL-puulla, itseisesti kiertyvällä puulla ja Fibonacci-keolla.

Luvussa 2 määritellään tutkielmassa tarvittavia käsitteitä rakenteille ja niiden tilaehdoille. Lisäksi esitellään Dijkstran algoritmi, ahkerat ja laiskat prioriteettijonot ja verrataan toteuttavien rakenteiden rajapintoja. Luvussa esitellään myös laskennan painopistettä siirtävä menetelmä, implisiittinen poistaminen, jonka tarkoitus on siirtää laskentaa useimmin käytetyltä operaatiolta harvemmin käytetyille.

Luvussa 3 tarkastellaan prioriteettijonon toteutuksia, implisiittisen prioriteettijonon toteutuksia ja sovelletaan implisiittisen poistamisen tekniikkaa myös hakurakenteisiin, joissa sitä ei aiemmin ole hyödynnetty. Tutkielmassa esitellään seuraavat rakenteet: binäärikeko, vasenta suosiva keko, AVL-puu, itseisesti kiertyvä keko ja Fibonacci-keko.

Luvussa 4 esitellään aineisto ja ympäristö mittauksille. Aineisto kootaan satunnais- ja täydellisen kaltaisista verkoista kolmella painofunktiolla. Mittaukset suoritettiin Java-virtuaalikoneella Itä-Suomen yliopiston Laskuri 3 -palvelimella ja aiheutuvia virhelähteitä arvoidaan.

Luvussa 5 esitellään saadut tulokset. Tuloksista ensimmäisenä esitellään operaatiomääriin perustavat tulokset ja suoritusaikojen tuloksia käsitellään näiden jälkeen. Rakenteita verrataan toisiinsa sekä pareittain että ryhmissä.

Luvussa 6 tehdään yhteenveto tutkielmassa esitetyistä asiosta. Lopuksi pohditaan tulosten merkittävyyttä ja aihepiirin jatkotutkimusmahdollisuuksia.

2 Määritelmät

Luvussa esitellään tarvittavat määritelmät, Dijkstran algoritmi ja tarkastellaan prioriteettijonorajapinnan toteutustekniikoita. Luku seuraa pääasiassa Tarjanin (1983) ja Knuthin (1973) kirjoja.

2.1 Verkko, ehdot ja mitat

Tarjanin mukaan *verkko* G koostuu solmujen joukosta V ja kaarien joukosta E. Joukon E kaari tarkoittaa paria, jossa molemmat alkiot kuuluvat solmujen joukkoon V. Verkkoa sanotaan *suuntaamattomaksi*, jos kaikki kaaret joukossa E ovat *järjestämättömiä* pareja $\{u,v\}$, muuten verkkoa kutsutaan *suunnatuksi* ja parit ovat *järjestettyjä* (u,v). Verkko G on painotettu, jos kaikille kaarille on määriteltävissä funktio $paino(u,v)$. Vastaavasti solmuille on määriteltävissä funktioita, kuten yksilöivä funktio $identiteetti(u)$ tai tärkeyttä merkitsevä funktio $prioriteetti(u)$. Olkoon lisäksi $avain(u)$ muodostettu katenoimalla $prioriteetti(u)$ ja $identiteetti(u)$.

Verkossa on *polku* p, joka on jono solmuja joukosta V ja jonka peräkkäisille jäsenille u ja v pätee, että pari (u,v) kuuluu joukkoon E. Polun paino on kaarien painojen summa. Polussa on *sykli*, jos mikä tahansa solmu u esiintyy polussa enemmän kuin yhden kerran. Solmu v on *saavutettavissa* solmusta u, jos on olemassa polku solmusta u solmuun v. Verkko G on *yhtenäinen*, jos kaikki sen solmut ovat saavutettavissa kaikista solmuista.

Suuntaamaton, yhtenäinen ja syklitön verkko on *vapaa puu*. Jos yksi solmu vapaasta puusta valitaan *juurisolmuksi* tai *juureksi*, niin puuta kutsutaan *juurelliseksi puuksi*. Olkoon puulla juuri r ja jokin toinen solmu v. Lisäksi puussa on polku solmusta r solmuun v ja polulla solmu u. Tällöin r on solmun u *esivanhempi* ja solmu v on solmun u *jälkeläinen*. Jos on olemassa kaari (u,v), niin u on solmun v *vanhempi* ja solmu v on solmun u *lapsi*. Lapsetonta solmua kutsutaan *lehdeksi*. Vastaavasti solmua, joka ei ole lehti, sanotaan *sisäsolmuksi*.

Juurellisen puun solmuilla voi olla mielivaltainen määrä alipuita. Alipuiden lukumäärä kutsutaan *lapsiluvuksi*. Juurellista puuta, jonka jokaisen solmun lapsilukua rajoitetaan jollakin luvulla d, kutsutaan *d-puuksi*. Erityisesti *2-puuta* kutsutaan *binääripuuksi*. Binääripuun jokaisella solmulla on korkeintaan kaksi lasta, joita jatkossa kutsutaan *vasemmaksi* ja *oikeaksi*. Lisäksi ne ovat binäärisien alipuiden juuria tai lehtiä.

Seuraavaksi määritellään juurellisen puun solmuille *syvyys*, *korkeus* ja *mataluus*. Määritellään, että juuren syvyys on 0 ja solmu v vanhempaansa yhden yksikön verran syvemmällä. Syvyys merkitsee kaarten lukumäärää polulla solmusta juureen. Binääripuuta kutsutaan *täydeksi*, jos kaikki lehdet ovat samalla syvyydellä.

Määritellään, että lehden korkeus on 0 ja solmu v on korkeinta lastaan yhden yksikön verran korkeammalla. Solmun korkeus merkitsee suurinta kaarten lukumäärää polulla solmusta lehteen. Lisäksi puun korkeus voidaan määritellä juuren korkeutena.

Määritellään, että lehden mataluus on 0 ja solmun v mataluus on yhden yksikön suurempi kuin pienin lapsien mataluuksista. Mataluus merkitsee pienintä kaarten lukumäärää polulla solmusta lehteen.

Seuraavaksi määritellään binääriselle puulle rekursiivisia ehtoja: *hakuehto*, *prioriteettiehto*, *tasapainoehto* ja *suosivuusehto*. Binäärinen puu on *binäärihakupuu*, jos puun solmut täyttävät hakuehdon. Hakuehtoa varten määrätään lasten järjestys: solmun vasemman lapsen avain on pienempi kuin solmun avain ja oikean lapsen avain suurempi kuin solmun avain, sellaisten solmujen tapauksessa, joilla on lapsia.

Määritellään, että solmussa on voimassa hakuehto, kun solmun lapset ovat järjestyksessä ja solmun lapset noudattavat hakuehtoa. Lapseton solmu on hakuehdon mukainen aina ja yksilapsinen solmu on hakuehdon mukainen, jos lapsen järjestys on voimassa.

Hakuehdosta seuraa, että juurellisen puun pienimmän avaimen löytämiseksi on kuljettava polku juuresta järjestysrelaation pienimmän solmun osoittamaan suuntaan ja suurimman avaimen löytämiseksi päinvastaiseen suuntaan. Hakuehdollisen juurellisen puun kaikki solmut ovat haettavissa avaimen arvolla.

Hakuehtoa löyhempi järjestely tapa on prioriteettiehto. Prioriteettiehtoa varten tarvitaan priorisoituneisuus. Solmu on maksimipriorisoitunut, jos sen lapsien prioriteettien arvot ovat pienempiä kuin solmun prioriteetti. Määritellään prioriteettiehto, jonka mukaan solmu on priorisoitunut ja sen lapset ovat prioriteettiehtoisia. Tästä seuraa, että solmun lapsien prioriteetit ovat pienempiä kuin solmun itse ja, että juurisolmu on prioriteetiltaan suurin.

Prioriteettiehto tunnetaan kirjallisuudessa *kekoehtona* (eng. *heap condition*) ja usein puhutaan rakenteessa solmujen avaimista, jolloin avaimen arvo sisältää prioriteetin. Käytännössä on tilanteita, joissa priorisoituminen pitää kääntää toiseen suuntaan eli minimipriorisoitumiseksi.

Tasapainoehtoja voidaan määrätä ainakin korkeuden, painon (Knuth, 1973) ja mataluuden (Haeupler et al., 2009) perusteella. Tarkastellaan tasapainoehtoa korkeuden mukaan binääripuussa määrittelemällä *tasapainotila*. Tasapainotila on solmun vasemman lapsen ja oikean lapsen korkeuksien erotus.

Määritellään *AVL-tasapainoehto* (Knuth, 1973), että solmun tasapainotilan itseisarvo on rajoitettu arvoihin 0 ja 1 sekä lisäksi solmun lapset ovat AVL-tasapainoehtoisia. AVL-ehdosta seuraa, että puun korkeus on rajoitettu logaritmiseksi solmujen määrään nähden.

Binääripuu *suosii vasenta* haaraa, jos vasemman lapsen mataluus on vähintään yhtä suuri kuin oikean lapsen mataluus. Määritellään suosivuusehto, että solmu suosii vasenta haaraa ja sen lapset ovat suosivuusehtoisia. Suosivuus tarkoittaa, että polut kaarten määrällä mitaten voivat olla puun suositussa haarassa pidempiä kuin toisessa haarassa.

Tietorakennetta kutsutaan *konsistentiksi*, jos se täyttää sitä kuvaavat ehdot. Tietorakennetta kutsutaan *ahkeraksi*, jos sen kaikki alkiot ovat määritellyssä konsistentissa tilassa jokaisen operaation jälkeen. Vastavasti, jos tietorakenne ei ole ahkera, sitä kutsutaan *laiskaksi*. Laiska rakenne voidaan palauttaa tarvittaessa konsistenttiin tilaan, mutta jokaisen operaation jälkeen näin ei ole. Tietorakennetta kuvaavat ehdot voivat rikkoutua,

kun tietorakenteen tietoja lisätään, päivitetään tai poistetaan. Tietorakennetta kuvaavat ehdot voidaan palauttaa, kun rakenteen rikkoutuneen ehdot korjataan sopivilla algoritmeilla.

2.2 Algoritmit

Binääripuu voidaan läpikäydä *esi-*, *sisä-* tai *jälkijärjestyksessä*. Järjestykset käyvät ilmi proseduurista BINÄÄRIPUUN LÄPIKÄYNTI (**Algoritmi 1**) (Tarjan, 1983). Solmussa käynnissä esijärjestyksessä käskyt suoritetaan ennen lapsissa käyntiä, sisäjärjestyksessä niiden välissä ja jälkikäynnissä niiden jälkeen. Vastaavasti käänteisen järjestyksen saa vaihtamalla lasten järjestyksen proseduurissa. Eritoten hakupuussa avaimet ovat järjestettyjä, joten niiden tulostaminen sisäjärjestyksessä tuottaa tulosteen, jossa avaimet ovat järjestyksessä.

Algoritmi 1 Algoritmirunko esi-, sisä- ja jälkijärjestyksille.

```
 1: procedure BINÄÄRIPUUN LÄPIKÄYNTI(Solmu u)
 2:     if u on lehti then
 3:         return /* Lehdellä ei ole lapsia */
 4:     end if
 5:     /* Esijärjestyksen käskyt */
 6:     BINÄÄRIPUUN LÄPIKÄYNTI(vasen(u))
 7:     /* Sisäjärjestyksen käskyt */
 8:     BINÄÄRIPUUN LÄPIKÄYNTI(oikea(u))
 9:     /* Jälkijärjestyksen käskyt */
10: end procedure
```

Algoritmien *aika- ja tilavaativuutta* pyritään arvioimaan *asymptoottisilla kertaluokilla*. Arvioinnin tarkoitus on hahmottaa, mitä kertaluokkaa algoritmin suorittamiseen käytettävä aika tai tila olisi suhteessa syötteen suuruuteen. Asymptoottisyys tarkoittaa, että aikavaativuus lähenee laskettua arvoa syötteen määrän suurentuessa ja kertaluokka tarkoittaa, että kiinnostuksen kohteena on itse suhdefunktio $f(n)$. Tulokseksi saatuja kertaluokkia vertaillaan asymptoottisella notaatiolla:

- $\mathrm{O}(f(n))$ Enintään samaa kertaluokkaa kuin $f(n)$.
- $\Theta(f(n))$ Täsmälleen samaa kertaluokkaa kuin $f(n)$.

- $\Omega(f(n))$ Vähintään samaa kertaluokkaa kuin $f(n)$.

Esimerkiksi proseduurin BINÄÄRIPUUN LÄPIKÄYNTI (**Algoritmi 1**) aikavaativuus voidaan perustella lyhyesti. Oletetaan, että välissä tehtävät käskyt ovat aikavaativuudeltaan $O(1)$. Jos algoritmi käy läpi kaikki n solmua ja tekee täsmälleen kerran esi-, sisä- tai jälkikäynnin käskyt, niin aikavaativuus on $\Theta(n)$. Koska $O(n)$ ei ole suurempaa kertaluokkaa kuin $\Theta(n)$, niin jatkossa yksinkertaistetaan tarkastelua käyttämällä vain tarpeeksi tarkkaa vertailua $O(n)$.

Tässä tutkielmassa tarkastellaan pääsääntöisesti pahimman tapauksen aikavaativuuksia, mutta tästä huolimatta jotkin aikavaativuudet ilmoitetaan lähteidensä mukaan *tasattuina aikavaativuuksina* (Sleator ja Tarjan, 1985; Fredman ja Tarjan, 1987). Tasattu aikavaativuus tarkoittaa, että yksittäiset operaatiot saattavat ylittää keskimääräisen aikavaativuuden, mutta tarkastelemalla suurta syötteen kokoa ja operaatioiden sarjaa keskimääräinen operaation aikavaativuus on tuota mainittua kertaluokkaa. Tasattu aikavaativuus kertoo algoritmista enemmän silloin, kun pahimman tapauksen aikavaativuus kuvaa algoritmia huonosti.

Proseduuri BINÄÄRIPUUN LÄPIKÄYNTI (**Algoritmi 1**) yleistyy myös muihin puurakenteisiin, mutta kun puun sijasta läpikäydään verkkoa, niin läpikäyntiin sopii paremmin leveys- tai syvyyssuuntainen läpikäynti. Proseduuri VERKON LÄPIKÄYNTI (**Algoritmi 2**) esittää algoritmirungon verkon leveys- ja syvyyssuuntaiseen läpikäyntiin. Algoritmissa käytettävä tilavaativuudeltaan $O(n)$ apurakenne *jono* tai *pino* muuttaa algoritmin toimintaa seuraavasti. Jos apurakenne Q on jono, niin läpikäynti tapahtuu *leveyssuuntaisena* ja apurakenteen ollessa pino *syvyyssuuntaisena*. Algoritmin aikavaativuus on $O(m+n)$, jossa m on verkon kaarten määrä ja n on verkon solmujen määrä, jos apurakenteeseen lisääminen ja ensimmäisen poistaminen sekä läpikäynnin käskyt ovat aikavaativuudeltaan $O(1)$.

Dijkstran (1959) mukaan *lyhimpien polkujen ongelmassa* tarkastellaan yhtenäistä ja positiivisesti painotettua verkkoa, jossa on n solmua ja m kaarta. Siinä ongelmana on löytää yhdestä solmusta u lähtevät lyhimmät polut verkon muihin solmuihin.

Algoritmi 2 Algoritmirunko leveys- ja syvyyssuuntaiselle läpikäynnille.

```
1:  procedure VERKON LÄPIKÄYNTI(Solmu u)
2:      Apurakenne Q
3:      LISÄÄ UUSI(Q, u)
4:      repeat
5:          v ← POISTA ENSIMMÄINEN(Q)
6:          /* Läpikäynnin solmua v käsittelevät käskyt */
7:          for w ∈ vierussolmu(v) do
8:              /* Läpikäynnin kaaria (v, w) käsittelevät käskyt */
9:              if w ei ole merkitty käydyksi then
10:                 merkitään w käydyksi
11:                 LISÄÄ UUSI(Q, w)
12:             end if
13:         end for
14:     until Q on tyhjä
15: end procedure
```

Dijkstran esittämä ratkaisu lyhimpien polkujen ongelmaan tunnetaan Dijkstran algoritmina. Alkuperäisessä (Dijkstra, 1959) algoritmissa esitellään käytettäväksi kolmea kaarten ja kolmea solmujen joukkoa, joiden operaatiot oletetaan olevan aikavaativuudeltaan $O(1)$. Tällä tavoin ongelmalle on saatu laskettua aikavaativuuden alaraja $O(m+n)$, johon päästään suunnatuilla ja syklittömillä verkoilla. Seuraavassa tarkasteellaan mukautettua Tarjanin (1983) versiota Dijkstran algoritmista (**Algoritmi 3**), joka on suunniteltu yleisemmille (sallii syklit), mutta positiivisesti painotetuille verkoille.

Proseduuri LYHIMMÄT ETÄISYYDET (**Algoritmi 3**) ratkaisee verkon (V,E) yhdestä lähtösolmusta u lyhimmät etäisyydet muihin solmuihin sanakirjarakenteeseen *matka*. Algoritmi pitää muistissa lyhintä löytynyttä etäisyyttä kuhunkin solmuun minimiprioriteettijonossa Q ja algoritmin suoritus koostuu kahdesta vaihdeesta. Ensimmäisessä vaiheessa alustetaan kaikki etäisyydet arvolla ∞, koska solmuihin ei ole tunnettua etäisyyttä. Käytännössä arvoksi ∞ riittää sellainen luku, joka on suurempi kuin kaikkien kaarien $paino(v,w)$-funktion arvojen summa. Lisäksi etäisyys lähtösolmuun nollataan ja prioriteettijonoon lisätään vain lähtösolmu prioriteetilla 0. Toinen vaihe, joka ratkaisee ongelman, koostuu kahdesta sisäkkäisestä silmukasta, jotka esitellään seuraavaksi.

Ulommassa silmukassa tarkastellaan solmua v, joka on lyhimmän löydetyn polun pää-

Algoritmi 3 Proseduuri, joka mukailee Tarjanin versiota Dijkstran algoritmista.

```
procedure LYHYIMMÄT ETÄISYYDET(Verkko (V,E), Solmu u, Etäisyydet matka[1..n])
    Minimiprioriteettijono Q
    for v ∈ V do
        matka[v] ← ∞
    end for
    matka[u] ← 0
    LISÄÄ UUSI(Q,(u,0))
    repeat
        v ← POISTA TÄRKEIN(Q)
        for w ∈ vierussolmu(v) do
            pituus_uusi ← matka[v] + paino(v,w)          ▷ uusi matka
            pituus_vanha ← matka[w]                      ▷ vanha matka
            if pituus_vanha = ∞ then
                LISÄÄ UUSI(Q,(w,pituus_uusi))
                matka[w] ← pituus_uusi
            else if pituus_uusi < pituus_vanha then
                PÄIVITÄ PRIORITEETTI(Q,(w,pituus_uusi),(w,pituus_vanha))
                matka[w] ← pituus_uusi
            end if
        end for
    until Q on tyhjä
end procedure
```

tepiste. Jokaisella kierroksella prioriteettijonosta poistetaan lyhintä etäisyyttä vastaava solmu operaatiolla POISTA TÄRKEIN. Solmua, joka on poistettu, ei koskaan lisätä prioriteettijonoon uudestaan ja kaikki lähtösolmusta u saavutettavat solmut lisätään kertaalleen prioriteettijonoon, josta seuraa, että silmukkaa suoritetaan n kertaa.

Sisemmässä silmukassa tarkastellaan solmusta v lähteviä kaaria (v,w). Lasketaan uusi etäisyys $pituus_{uusi} = matka[v] + paino(v,w)$, jossa $paino(v,w)$ on kaareen (v,w) liittyvä painofunktio ja asetetaan vanha etäisyys muuttujaan $pituus_{vanha} = matka[v]$. Jos kaaren päätepisteeseen w ei ennestään vienyt polkua eli vanha etäisyys on ∞, niin prioriteettijonoon Q lisätään operaatiolla LISÄÄ UUSI solmu w prioriteetilla $pituus_{uusi}$. Jos uusi etäisyys on lyhyempi kuin vanha eli $pituus_{uusi} < pituus_{vanha}$, niin päivitetään prioriteettijonossa Q solmun w prioriteteetti arvosta $pituus_{vanha}$ arvoon $pituus_{uusi}$ operaatiolla PÄIVITÄ PRIORITEETTI. Uuden etäisyyden ollessa vanhaa lyhyempi päivitetään myös matka-muuttuja $matka[w] = pituus_{uusi}$. Jokaiseen solmuun liitetyt kaaret (*vierussolmut*) käydään vain kerran läpi, joten silmukkaa suoritetaan yhteensä m kertaa. Algoritmin lopuksi yhtenäisessä verkossa kaikkilla solmuilla on jokin $matka[w] \neq \infty$.

Proseduurista LYHIMMÄT ETÄISYYDET (**Algoritmi 3**) saadaan edellytykset vaatia prioriteettijonolta operaatioita POISTA TÄRKEIN, LISÄÄ UUSI ja PÄIVITÄ PRIORITEETTI. Operaatioita POISTA TÄRKEIN ja LISÄÄ UUSI kutsutaan n kertaa ja operaatiota PÄIVITÄ PRIORITEETTI kutsutaan korkeintaan $m-n$ kertaa. Aikavaativuus riippuu siis paljolti operaation PÄIVITÄ PRIORITEETTI toteutuksesta.

Jos oletetaan, että solmujen välillä on korkeintaan yksi kaari, niin Dijkstran algoritmi rajaa lopputulokseen vaikuttavien kaarien määrää seuraavasti. Kaaria, jotka osoittavat jo käsiteltyyn solmuun ei käsitellä operaatiolla PÄIVITÄ PRIORITEETTI, sillä polku jo käsiteltyyn solmuun on jo lyhyempi kuin myöhemmin käsiteltävän kaaren kautta kulkeva polku. Sama pätee käsiteltävälle solmulle itselleen, eli solmusta kaaria solmuun itseensä Dijkstran algoritmi ei käsittele. Tällöin kaarien lukumäärä

$$m = \frac{n^2 - n}{2} \in \mathrm{O}(n^2), \text{ jossa } n \text{ on solmujen lukumäärä.}$$

Andrew Goldbergin ja Robert Tarjanin (1996) mukaan käytännössä suurelle osalle erilaisia verkkoja operaation PÄIVITÄ PRIORITEETTI toistokertojen lukumäärä

$$M \in \mathrm{O}(n \log(1 + \frac{m}{n})),$$

jossa n on solmujen ja m kaarien lukumäärä. Dijkstran algoritmin prioriteettijonon päivityskertojen lukumäärä voi olla vähemmän kuin verrannollinen kaarien lukumäärään. Operaatio PÄIVITÄ PRIORITEETTI ei yksinään dominoi suoritusaikaa, jos aikavaativuus on samaa luokkaa kuin operaatioilla LISÄÄ UUSI ja POISTA TÄRKEIN.

Lyhimpien polkujen ongelman voi ratkaista käyttämällä proseduuria VERKON LÄPIKÄYNTI (**Algoritmi 2**) pohjana ja pitää muistissa mahdollisesti useita löytyneitä etäisyyksiä kuhunkin solmuun. Proseduuri LYHIMMÄT POLUT TOISIN (**Algoritmi 4**) on lyhimpien polkujen löytämiseen muokattu versio verkon läpikäynnistä apurakenteen ollessa *minimiprioriteettijono implisiittisellä poistolla*. Algoritmi eroaa Dijkstran algoritmista operaation PÄIVITÄ PRIORITEETTI puuttumisena, mistä seuraa, että operaatiota LISÄÄ UUSI kutsutaan m kertaa ja operaatioita POISTA TÄRKEIN n kertaa, kun se palauttaa vain aiemmin poistamattomia solmuja eli implisiittisesti poistaa etäisyyksiä solmuihin w, joille $matka[w] \leq matka[v]$. Tähän tarkoitukseen tarvitaan implisiittisen poistamisen prioriteettijonoja, jotka hoitavat poistamisen sisäisesti – rajapinnan sisäpuolella.

2.3 Rajapinnat

Tarjanin (1983) mukaan J. W. J. Williams, kekolajittelun keksijä, käytti termiä *keko* (*heap*) varsinaisesta lapsiluvultaan luvulla d rajoitetusta ja kekoehtoisesta puurakenteesta, joka tunnetaan nykyisin *d-kekona*. Knuthin (1973, luku 5.2.3) mukaan prioriteettijono on pinon ja jonon tapainen abstrakti tietorakenne, jonka voi d-keolla tehokkaasti toteuttaa. Kirjallisuudessa esiintyy keko ja prioriteettijono myös muissa merkityksissä ja varsinkin keko-sana saattaa merkitä mitä tahansa prioriteettiehtoista (kekoehtoista) rakennetta ja siihen liitetään myös kaikki informaatio, joka prioriteettijo-

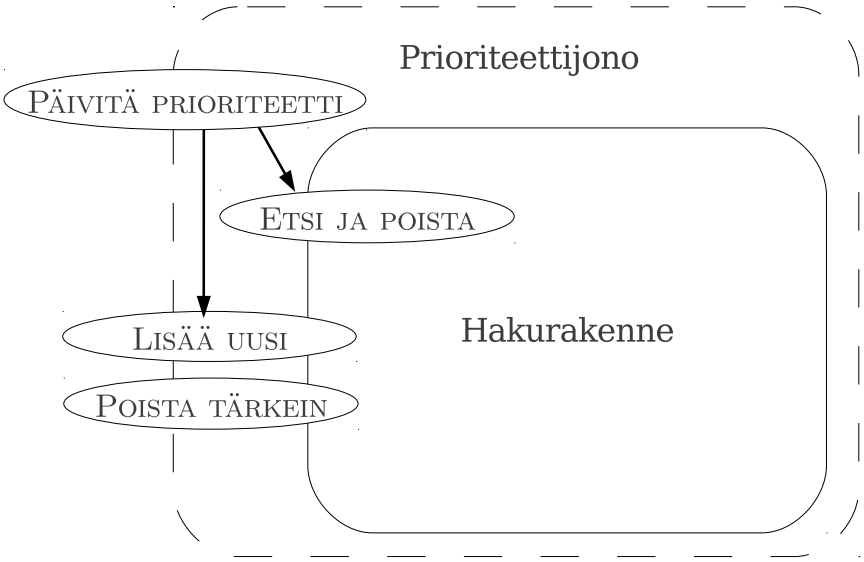
Algoritmi 4 Proseduuri, joka on muokattu verkon läpikäynti algoritmista.

```
procedure LYHIMMÄT ETÄISYYDET TOISIN( Verkko (V,E), Solmu u, Etäisyydet matka[1..n])
    Minimiprioriteettijono implisiittisellä poistolla Q
    for v ∈ V do
        matka[v] ← ∞
    end for
    matka[u] ← 0
    LISÄÄ UUSI(Q,(u,0))
    repeat
        v ← POISTA TÄRKEIN(Q)
        for w ∈ vierussolmu(v) do
            pituus_uusi ← matka[v] + paino(v,w)          ▷ uusi matka
            pituus_vanha ← matka[w]                      ▷ vanha matka
            matka[w] ← pituus_uusi
            if pituus_uusi < pituus_vanha then
                LISÄÄ UUSI(Q,(w,pituus_uusi))
            end if
        end for
    until Q on tyhjä
end procedure
```

non toteuttamiseen tarvitaan. Jatkossa keko merkitsee varsinaista prioriteetiltaan järjestettyä rakennetta ja muu prioriteettijonossa hyödynnettävä informaatio nimetään tarpeen tullen.

Jatkossa prioriteettijonolla tarkoitetaan minimiprioriteettijonoa, jossa priorisoidaan poistojärjestys pienimmän prioriteetin mukaan. Sekaannusten välttämiseksi verkon solmuista käytetään nimeä *solmu* ja prioriteettijonon, joka on puurakenne, solmuista käytetään nimeä *alkio*. Usein tarkasteltaessa kekorakennetta alkion attribuutit prioriteetti ja avain samaistetaan, varsinkin nimellisesti. Alkioiden vertailulla tarkoitetaan niiden prioriteettien vertailua. Jos rakenne on hakurakenne, niin käytetään avaimia, joten alkioiden vertaileminen tarkoittaa niiden avaimien vertailemista.

Edellisessä luvussa havaittiin, että Dijkstran algoritmissa prioriteettijonolta vaadittavat proseduurit ovat LISÄÄ UUSI, POISTA TÄRKEIN ja PÄIVITÄ PRIORITEETTI. Kun keko toteuttaa prioriteettijonon lisäämisen ja poistamisen, se voidaan yleensä tehdä keon oman rajapinnan operaatiolla tehokkaasti. Näistä poiketen alkion prioriteetin päivittä-



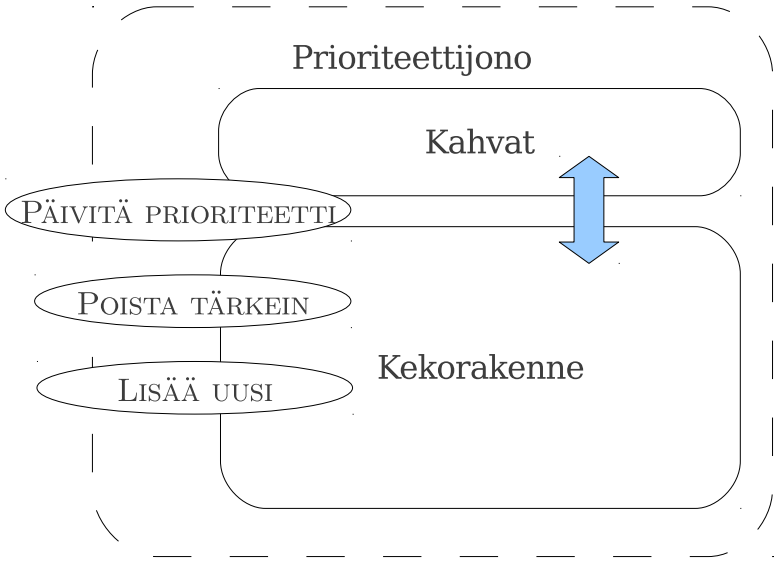
Kuva 1: Prioriteettijonon rajapinta, hakurakenteella.

minen ei kuulu kekorakenteen rajapintaan, sillä se edellyttää alkion paikan tuntemista keon ulkopuolelta käsin tai aikavaativuudeltaan $O(n)$ haun tekemistä keossa, joka ei puolestaan ole tehokasta. Prioriteettijonon rajapinta paketoi tietorakenteen ja sen ominaisuuksien toteuttamiseen tarvittavat tiedot. Tarkastellaan seuraavaksi prioriteettijonon toteuttavien rakenteiden rajapintoja.

Kuvassa 1 on prioriteettijono, jonka rajapinnassa on proseduurit LISÄÄ UUSI, POISTA TÄRKEIN ja PÄIVITÄ PRIORITEETTI. Proseduurit LISÄÄ UUSI ja POISTA TÄRKEIN käsittelevät rakennetta, joten ne on kuvattu myös rakenteen rajapintaan. Proseduuri PÄIVITÄ PRIORITEETTI on kuvattu kutsuvan proseduureja LISÄÄ UUSI ja ETSI JA POISTA. Proseduuri ETSI JA POISTA poistaa etsityn alkion rakenteesta.

Hakuehtoisessa rakenteessa voidaan hakea mikä tahansa alkio. Tällöin PÄIVITÄ PRIORITEETTI voidaan tehokkaasti toteuttaa, vaikka poistamalla vanhaa avainta vastaava alkio (ETSI JA POISTA) ja lisäämällä uusi toiseen paikkaan (LISÄÄ UUSI), jolloin proseduurin toteuttamiseen ei vaadita lisäinformaatiota.

Jos rakenteessa on voimassa hakuehdon sijaan kekoehto, ei mielivaltaista alkioa voida etsiä ja poistaa tehokkaasti. Proseduuri PÄIVITÄ PRIORITEETTI voitaisiin tehdä päi-

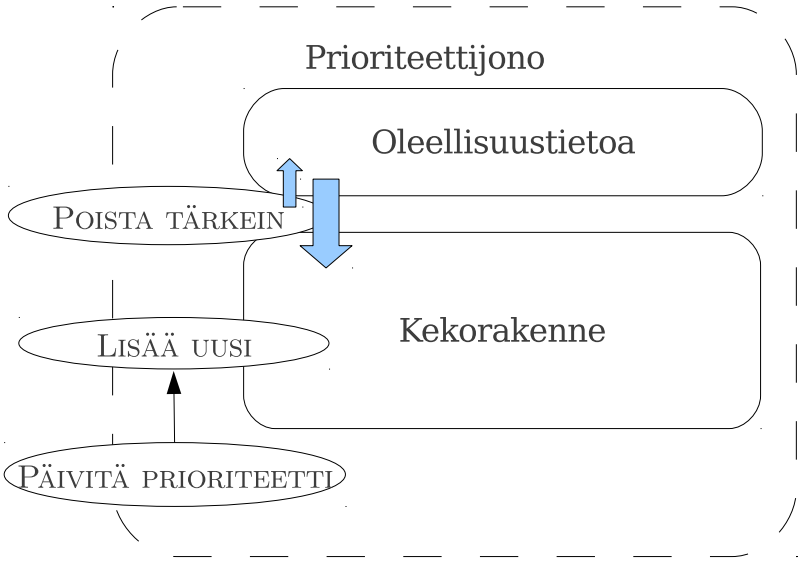


Kuva 2: Prioriteettijonon rajapinta, implisiittisesti kahvoja hyödyntävällä rakenteella.

vittämällä alkion prioriteetti, jos alkion etsimistä voisi nopeuttaa käyttämällä lisäinformaatiota, joka osoittaisi missä kohdassa rakennetta etsittävä alkio on milloinkin algoritmin suorituksen aikana. Tällaista tiedonosoittamiseen perustuvaa tekniikkaa kutsutaan *kahvoiksi*. Kahvoja tarvitaan yksi jokaista solmua kohti eli tilavaativuus on $O(n)$, mutta jokaisella kahvalla päästään suoraan käsiksi tarvittavaan alkioon eli yksittäisen kahvan aikavaativuus on vain $O(1)$.

Kuvassa 2 on prioriteettijono, jonka rajapinnassa on proseduurit LISÄÄ UUSI, POISTA TÄRKEIN ja PÄIVITÄ PRIORITEETTI. Proseduurit LISÄÄ UUSI ja POISTA TÄRKEIN käsittelevät kekoa ja siksi ne on kuvattu myös keon rajapintaan. Proseduuri PÄIVITÄ PRIORITEETTI käsittelee kekoa, mutta tarvitsee kahvoja nopeaan pääsyyn rakenteeseen ja siksi se on kuvattu enemmän kahvoihin kohdistuvaksi. Kaksisuuntainen nuoli kertoo, että kaikki kekoon tehtävät muutokset pitää päivittää myös kahvoihin.

Kompromissina edellä esitettyistä toteutuksista voidaan pitää tekniikkaa, jota kutsutaan implisiittiseksi poistoksi (Tarjan, 1983). Idea on tallentaa implisiittiseen eli sisäiseen käyttöön rajapinnan sisäpuolelle tietoa onko alkiota vastaava solmu jo poistettu rakenteesta eli onko alkio lopputuloksen kannalta *oleellinen* vai *epäoleellinen*.



Kuva 3: Prioriteettijonon rajapinta, implisiittisen poiston toteuttavalla rakenteella.

Kuvassa 3 on prioriteettijono, jonka rajapinnassa on proseduurit LISÄÄ UUSI, PÄIVITÄ PRIORITEETTI ja POISTA TÄRKEIN. Proseduurit LISÄÄ UUSI ja PÄIVITÄ PRIORITEETTI käsittelevät kekoa, mutta ei oleellisuustietoja. Proseduuri POISTA TÄRKEIN päivittää poistoa kohti yhden arvon oleellisuustietoa (pieni nuoli ylöspäin), mutta hyödyntää sitä paljon käsiteltäessä kekoa (suuri nuoli alaspäin).

Implisiittisen poistamisen tekniikka perustuu laiskaan poistamiseen eli mielivaltaisen poistamisen (ETSI JA POISTA) lykkäämiseen. Jos rakenteen kokoaminen osarakenteita yhdistämällä on tehokasta, niin poiston yhteydessä (POISTA TÄRKEIN) rakennetaan rakenne osarakenteistaan uudelleen. Poiston yhteydessä vastaan tulevia epäoleellisia alkioita ei uuteen rakenteeseen yhdistetä, vaan ne poistetaan.

Proseduuri PÄIVITÄ PRIORITEETTI toimii samoin kuin proseduuri LISÄÄ UUSI eli se lisää rakenteeseen uuden alkion aikaisempien alkioiden lisäksi. Yhdistettynä laiskaan lisäämiseen, joka voidaan suorittaa aikavaativuudella $O(1)$, siirtyy laskentataakkaa lisäämisestä poistamiseen. Laiskasta poistamisesta ja ylimääräisistä lisäämisistä seuraa, että rakenteeseen jää epäoleellisia alkioita. Tällöin rakenteen tilavaativuus on $O(m)$.

Tekniikka mahdollistaa proseduurin LYHIMMÄT ETÄISYYDET TOISIN (**Algoritmi 4,**

sivu 12) käyttämisen ja saattaa nopeuttaa algoritmeissa, joissa poistoja tulee huomattavasti lisäyksiä vähemmän, kuten Dijkstran algoritmin joissain tapauksissa. Tutkielmassa selvitetään, missä tapauksissa tekniikkaa kannattaa hyödyntää ja missä määrin siitä koituu haittaa.

3 Prioriteettijonon toteutukset

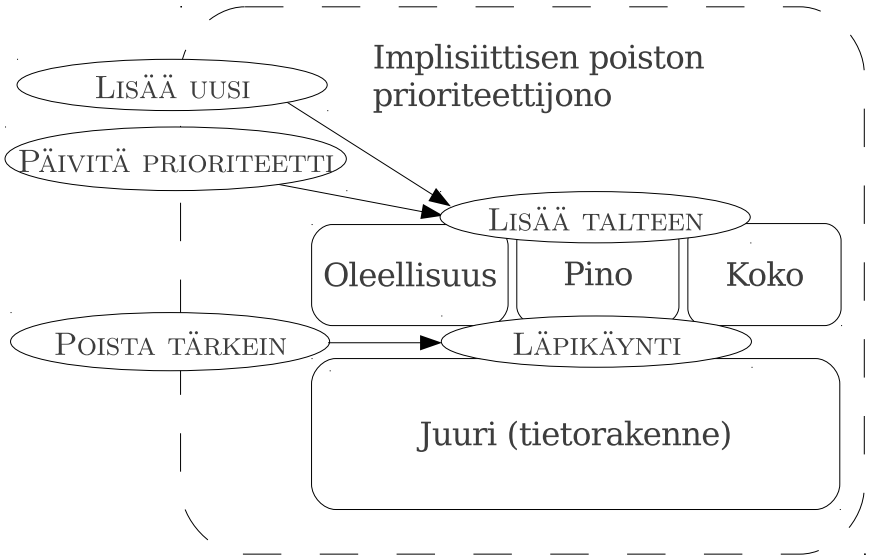
Tässä luvussa käsitellään prioriteettijonon toteuttamista muutamalla eri tavalla. Tarkastelun alla ovat binäärisiin puihin perustuvia perusrakenteita ja kehittyneempää binomipuuta edustaa Fibonacci-keko. Rakenteista käytetään seuraavia vapaasti suomennettuja nimiä:

- *Binäärikeko* (eng. *binary heap*)
- *Vasenta suosiva keko* (eng. *leftist heap*)
- *AVL-puu* (eng. *AVL tree*)
- *Itseisesti kiertyvä puu* (eng. *self-adjusting tree, splay tree*)
- *Fibonacci-keko* (eng. *Fibonacci heap*)

Rakenteiksi on valikoitunut kirjava joukko keko- ja hakurakenteita. Rakenteet edustavat aikavaativuuksiltaan jotkin tarkkoja ja toiset tasattuja. Rakenteita käytetään pohjana implisiittisen poiston prioriteettijonon toteutuksissa, joita vertaillaan sekä keskenään että pareittain ahkeran version kanssa.

Binäärikeosta ei saada implisiittisen poiston versiota, koska ei ole tunnettua tehokasta tapaa yhdistää binäärikekomaisia osarakenteita. Sen sijaan binäärikeosta tarjotaan sekä taulukoitu että linkitetty versio, sillä muut vertailevat algoritmit toimivat vain linkitetysti.

Kuvassa 4 on implisiittisen poiston prioriteettijonolle tässä tutkielmassa käytettävä puskuroitu toteutus. Puskurointi tekee rakenteesta laiskan. Rakenteeseen liittyvät tiedot ovat alkioiden *oleellisuus*, uusien alkioiden *pino*, alkioiden lukumäärä eli *koko* ja tietorakenteen *juuri*. Prioriteettijonon rajapinnan operaatioista LISÄÄ UUSI ja PÄIVITÄ PRIORITEETTI viittaavat proseduuriin LISÄÄ TALTEEN. Ainut implisiittisen poiston rakenteissa muuttuva asia, operaatio POISTA TÄRKEIN, käsitellään rakennekohtaisesti myöhemmin. Seuraavaksi esitellään proseduuri LISÄÄ TALTEEN, jotta sitä ei tarvitse käydä läpi jokaisen rakenteen kohdalla erikseen.



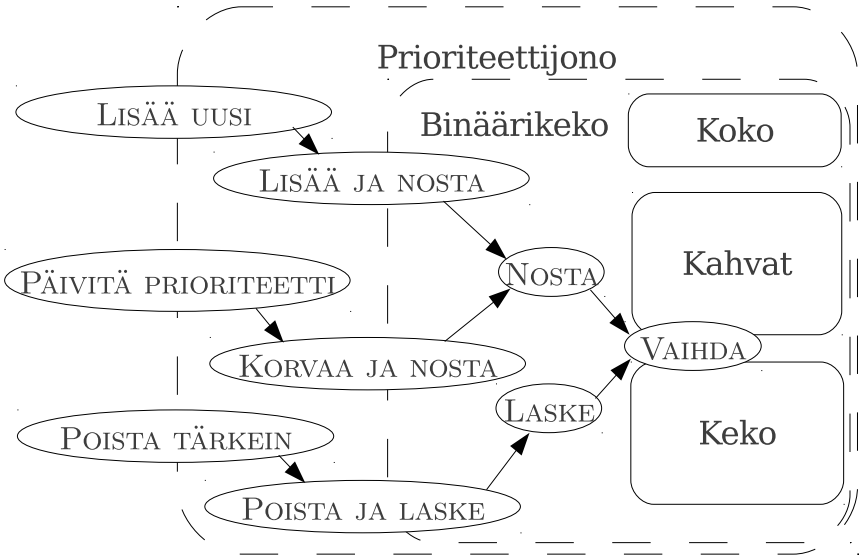
Kuva 4: Implisiittisen poiston prioriteettijonon toteutus.

Proseduuri LISÄÄ TALTEEN (**Algoritmi 5**) vastaa laiskaa lisäämistä. Aluksi merkataan rakenteessa oleva tieto vanhentuneeksi sen oleellisuutta muuttamalla. Proseduuri pitää pinon pienimmän alkion pinon päällä, koska uusimman alkion prioriteetti voi olla pienempi kuin pienin rakenteessa oleva. Muilla pinon alkioilla ei ole järjestystä. Laiska lisääminen on siis aikavaativuudeltaan vakio $O(1)$.

Algoritmi 5 Proseduuri laiskaan lisäämiseen.$O(1)$

```
procedure LISÄÄ TALTEEN(Prioriteettijono (pino, koko, oleellisuus), Alkio u)
    Merkataan oleellisuus[identiteetti(u)] epäoleelliseksi.
    koko ← koko + 1
    if Pinossa pino ei ole jäseniä then                          ▷ pino = null
        pino ← u
    else if prioriteetti(u) < prioriteetti(pino) then            ▷ u pinon päälle
        alla(u) ← pino
        pino ← u
    else                                                         ▷ u pinon toiseksi
        alla(u) ← alla(pino)
        alla(pino) ← u
    end if
end procedure
```

Seuraavissa aliluvuissa perehdytään yksityiskohtaisesti keko- ja hakurakenteiden toteutuksiin. Aliluvut koostuvat operaatioiden LISÄÄ UUSI, PÄIVITÄ PRIORITEETTI ja



Kuva 5: Binäärikeon rakenne.

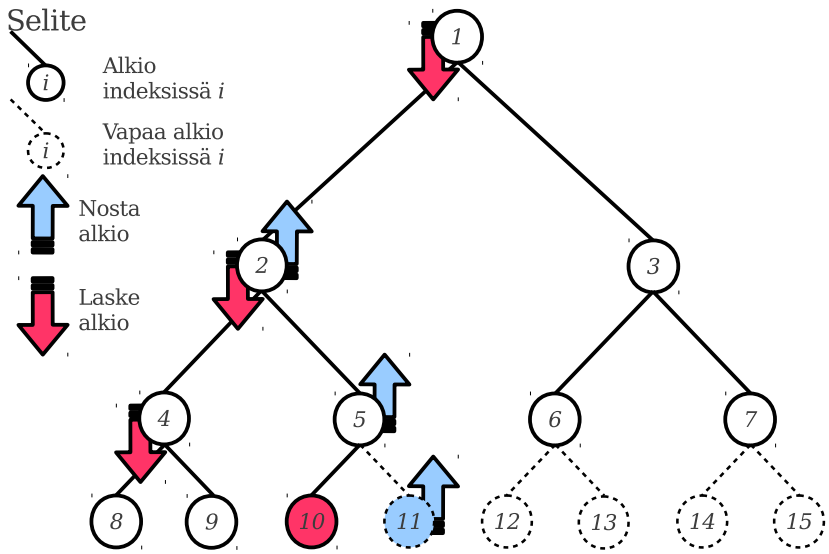
POISTA TÄRKEIN toteutusten selittämisestä ja joidenkin aikavaativuuksien ja konsistettisuuksien perusteluista.

3.1 Binäärikeko

Tarjanin (1983) mukaan d-keolla tarkoitetaan kekoehdollista d-haaraista puuta, johon lisääminen ja poistaminen tehdään *tasojärjestyksessä*. Tasojärjestys on sama kuin leveyssuuntaisen läpikäynnin järjestys d-puuta läpikäydessä.

3.1.1 Taulukoituna

Tässä luvussa on kaksikantainen sovitus Tarjanin (1983) d-kekoalgoritmeista lisättynä kahvojen käytöllä ja prioriteetin päivittämisellä. Kuvassa 5 on binäärikeon rakenne, joka toteuttaa prioriteettijonon rajapinnan. Binäärikeosta tiedetään $keko[1..n]$, alkioiden määrä $koko$ ja $kahvat[1..n]$. Operaatiosta LISÄÄ UUSI vastaa LISÄÄ JA NOSTA, operaatiosta PÄIVITÄ PRIORITEETTI vastaa KORVAA JA NOSTA ja operaatiosta POISTA TÄRKEIN vastaa POISTA JA LASKE.



Kuva 6: Lisääminen ja poistaminen binäärikeossa.

Kuvassa 6 on 10-solmuinen binäärikeko. Siniset nuolet kuvaavat proseduurin LISÄÄ JA NOSTA mahdolliset vaihdot uuden alkion noustessa mahdollisimman ylös puussa. Sitä vastoin, kun rakenteesta poistetaan tärkein eli alkio indeksistä 1 ja se korvataan pohjimmaisella eli alkiolla indeksista 10, punaiset nuolet kuvaavat proseduurin POISTA JA LASKE vaihdot tapauksessa, jossa kuvan alkioiden indeksit olisivat prioriteetteja.

Proseduurilla LISÄÄ JA NOSTA (**Algoritmi 6**) on kuvattu alkion lisääminen binäärikekoon. Kekoon lisääminen tehdään keon pohjalle, josta alkio nostetaan oikealle sijoilleen prioriteettiinsa nähden. Kekoehto korjataan proseduurissa NOSTA (**Algoritmi 7**).

Algoritmi 6 Proseduuri alkion lisääminen ja kekoehdon korjaaminen nostamalla. $O(\log_2 koko)$

```
procedure LISÄÄ JA NOSTA(Binäärikeko (keko[1..n], kahvat[1..n], koko), Alkio u)
    koko ← koko + 1                                          ▷ 0 ≤ koko ≤ n
    keko[koko] ← u
    kahvat[identiteetti(u)] ← koko
    if 0 < ⌊koko/2⌋ then
        NOSTA((Q,H), koko)
    end if
end procedure
```

Proseduuri NOSTA (**Algoritmi 7**) tarkastelee jokaista vaihtoa kohti yhden alkion ja jokainen vaihto kasvattaa alkion korkeutta yhdellä. Kekoehto korjataan vaihtamalla proseduurilla VAIHDA (**Algoritmi 8**) alkio vanhempansa kanssa toistuvasti, kunnes kekoehto on voimassa.

Algoritmi 7 Proseduuri kekoehdon korjaamiseen nostamalla. $O(\log_2 koko)$

```
procedure NOSTA(Binäärikeko (keko[1..n], kahvat[1..n]), Indeksi i)
    p ← ⌊i/2⌋
    while p ≠ 0 and prioriteetti(keko[i]) < prioriteetti(keko[p]) do
        VAIHDA((keko, kahvat), p, i)
        i ← p
        p ← ⌊i/2⌋
    end while
end procedure
```

Proseduuri VAIHDA (**Algoritmi 8**) saa kaksi indeksiä parametrinaan. Proseduuri vaihtaa indeksejä vastaavien alkioiden kahvat ja sisällöt keskenään.

Algoritmi 8 Proseduuri alkioiden keskinäiseen vaihtamiseen. $O(1)$

```
procedure VAIHDA(Binäärikeko (keko[1..n], kahvat[1..n]), Indeksi a, Indeksi b)
    kahvat[a], kahvat[b] ← b, a
    keko[a], keko[b] ← keko[b], keko[a]
end procedure
```

Proseduurilla KORVAA JA NOSTA (**Algoritmi 9**) kuvataan proseduuri alkion prioriteetin päivittämiseen. Alkion prioriteetin pienentämiseksi tulee ensiksi etsiä alkio rakenteesta. Kahvan indeksoimassa paikassa olevan alkion prioriteetti päivitetään uuteen arvoon ja mahdollinen kekoehdon rikkoutuminen korjataan kuten lisäämisen tapauksessa proseduurilla NOSTA.

Proseduurilla POISTA JA LASKE (**Algoritmi 10**) kuvataan tärkeimmän alkion poistaminen keosta ja kekoehdon korjaaminen sen jälkeen. Keon päällimmäinen alkio on prioriteetiltaan pienin ja keolle saadaan uusi juuri keon perimmäisimmästä alkiosta. Kekoehto korjataan proseduurilla LASKE (**Algoritmi 11**).

Proseduuri LASKE tarkastelee jokaista vaihtoa kohti korkeintaan kaksi alkiota ja jokainen vaihto kasvattaa alkion syvyyttä yhdellä, kun pienempi tarkastelluista alkiosta

Algoritmi 9 Proseduuri alkion prioriteetin korottamiseksi ja kekoehdon korjaamiseksi. $O(\log_2 koko)$

```
procedure KORVAA JA NOSTA(Binäärikeko (keko[1..n], kahvat[1..n], koko), Alkio u, Alkio v)
    i ← kahvat[identiteetti(u)]
    prioriteetti(keko[i]) ← prioriteetti(u)
    if 0 < ⌊i/2⌋ then
        NOSTA((Q,H),i)
    end if
end procedure
```

Algoritmi 10 Proseduuri tärkeimmän alkion poistamiseen keosta ja kekoehdon korjaamiseen laskemalla. $O(\log_2 koko)$

```
procedure POISTA JA LASKE(Binäärikeko (keko[1..n], kahvat[1..n], koko))
    vastaus ← keko[1]
    kahvat[identiteetti(vastaus)] ← ∞
    keko[1] ← keko[koko]
    kahvat[identiteetti(keko[1])] ← 1
    keko[koko] ← null
    koko ← koko − 1
    if 1 < koko then
        LASKE((Q,H),1)
    end if
    return vastaus
end procedure
```

vaihdetaan laskettavan alkion kanssa. Kekoehto korjataan vaihtamalla alkio toistuvasti prioriteetiltaan pienimmän lapsen (vasemman tai oikean) kanssa, kunnes kumpikaan lapsi ei riko kekoehtoa.

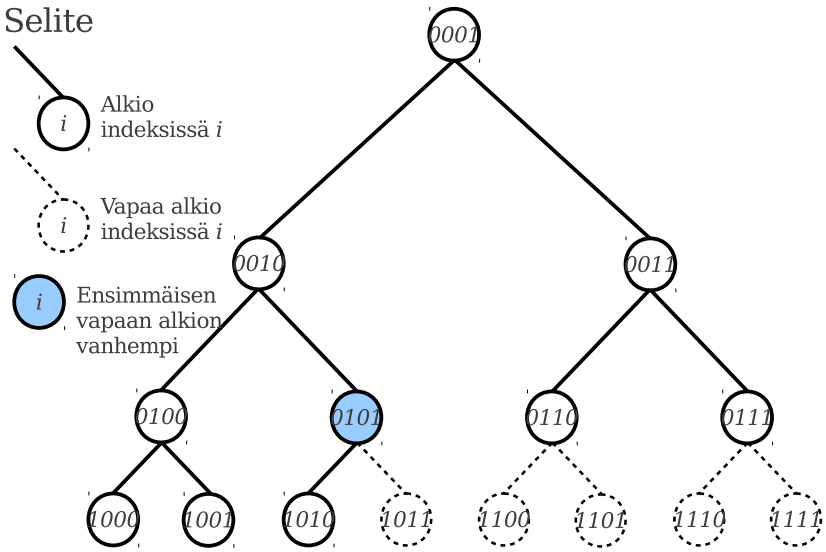
Algoritmi 11 Proseduuri kekoehdon korjaamiseen laskemalla. O($\log_2 koko$)

procedure LASKE(Binäärikeko ($keko[1..n], kahvat[1..n], koko$), Indeksi i)
    Indeksi j ▷ j on lapsen indeksi, jos alkiolla on lapsi
    $p_j = \min\{p_i, p_{2i}, p_{2i+1}\}$, jossa $p_j = prioriteetti(keko[j])$
    while $j \leq koko$ **and** $p_j < p_i$ **do**
        VAIHDA($(Q,H), i, j$)
        $i \leftarrow j$
        $p_j = \min\{p_{2i}, p_{2i+1}\}$, jossa $p_j = prioriteetti(keko[j])$ ja $j \leq koko$
    end while
end procedure

3.1.2 Linkitettynä

Linkitetty binäärikeko vastaa vertailussa paremmin muita linkitettyjä rakenteita antamalla hieman tasoitusta taulukoituun binääriseen kekoon nähden. Kuten taulukoidussa toteutuksessa tiedetään keon *juuri*, alkioiden lukumäärän *koko* ja $kahvat[1..n]$. Päivittäminen onnistuu kahvoilla, mutta rakenteen ongelma on, kuinka saavutetaan viimeinen alkio tehokkaasti, kun rakenteeseen lisätään tai siitä poistetaan alkioita. Nyt, toisin kuin taulukoidussa toteutuksessa, ei päästä käsiksi viimeiseen vapaaseen paikkaan ($juuri + koko$), koska ei tiedetä missä päin muistia se sijaitsee.

Kuvassa 7 on kymmenalkioinen binäärikeko, jossa alkiot on indeksoitu yhdestä eteenpäin binääriesityksenä. Koska binäärikeot täytetään tasojärjestyksessä, tulee rakenteesta löytää seuraavaksi vapaan paikan (indeksin $i = 1011_2$) vanhempi – sininen solmu. Tutkimalla hieman rakenteen indeksien binääriesitystä havaitaan seuraavia asioita. Olkoon indeksin i binäärilukuesitys $i = 2^{l-1}b_1 + 2^{l-2}b_2 + ... + 2^{l-k}b_k + ... + 2^0 b_l$, jossa l on lukuesityksen bittimäärä. Binäärilukuesityksen ensimmäinen ykkösbitti b_1 siirtyy yhden vasemmalle jokaista uutta alkiota kohti, jotka kasvattavat puun korkeutta eli samalla syvyydellä olevilla alkioilla on samassa paikassa lukuesitystään ensimmäinen ykkösbitti. Ensimmäistä ykkösbittiä seuraavat bitit b_k (kun $k > 1$) kertovat alkion



Kuva 7: Binäärikeko 10 solmua binääri-indeksillä.

paikasta. Jos $b_k = 0$, niin etsittävä alkio sijaitsee polun $[v_1, ..., v_{k-1}]$ alkiosta v_{k-1} vasemmalla. Jos $b_k = 1$, niin alkio sijaitseekin alkiosta v_{k-1} oikealla. Tässä päästäisiin hieman helpommalla, jos indeksit noudattaisivat Braunin puuta (Hoogerwoord, 1983).

Havainnoista edellä saadaan muotoiltua proseduuri alkion löytämiseksi. Proseduurilla ETSI ALKIO INDEKSILLE (**Algoritmi 12**) voidaan etsiä alkio kulkemalla indeksin johdattelevaa polkua pitkin. Proseduuri saa parametrinaan juurialkion *juuri* ja etsittävän alkion indeksin i, kun $0 < i \leq n$. Alustetaan alkio v juurella, muuttuja hb kokonaisluvulla, jonka arvo on sama kuin korkeimman bitin arvo luvusta i ja muuttuja t arvolla $i - hb$. Jaetaan muuttujan hb arvo kahdella merkitsemään seuraavaa bittiä. Siirrytään seuraavaksi suorittamaan silmukkaa niin kauan kuin hb on erisuuri kuin nolla. Jos muuttujan t arvo on pienempi kuin muuttujan hb, niin muuttujan hb osoittama bitti luvussa t on nolla, jolloin päivitetään alkio v vasempaan lapseensa. Muuten muuttujan hb osoittama bitti luvussa t on yksi ja se on samalla muuttujan t korkein bitti, jolloin päivitetään alkio v oikeaan lapseensa ja muuttuja t arvoksi $t - hb$. Silmukan lopussa päivitetään hb arvoon $hb/2$, joka tarkoittaa seuraavaksi pienimmän bitin tutkimista seuraavalla kieroksella. Silmukkaa suoritetaan $\lceil \log_2 i \rceil$ kertaa, koska joka kierroksella muuttuja hb käy läpi yhden bitin luvun i kaksikantaisesta esityksestä. Proseduurin

lopuksi palautetaan alkio v.

Algoritmi 12 Proseduuri alkion löytämiseksi indeksin perusteella.

```
procedure ETSI ALKIO INDEKSILLE(Alkio juuri, Indeksi i)
    v ← juuri
    hb ← min{2^x | i ≤ 2^x}
    t ← i − hb
    hb ← hb/2
    while hb ≠ 0 do
        if t < hb then
            v ← vasen(v)
        else
            v ← oikea(v)
            t − hb
        end if
        hb ← hb/2
    end while
    return v
end procedure
```

Alkion vanhempi saadaan samalla proseduurilla kutsumalla sitä parametrilla $i/2$. Kuvan 7 sinisen alkion laskenta suoritetaan seuraavasti. Kutsutaan proseduuria parametreilla *juuri* ja $i = 1011_2/2 = 101_2$. Alustetaan muuttujat $v_1 = juuri$, $hb_0 = 100_2$ ja $t_1 = i - hb_0 = 101_2 - 100_2 = 1$. Lisäksi asetetaan $hb_1 = hb_0/2 = 10_2$. Tarkistetaan, että silmukkaehto pätee eli $hb_1 = 10_2 \neq 0$ ja ehtolauseesta $t_1 = 1 < 10_2 = hb_1$ seuraa, että asetetaan $v_2 = vasen(v_1)$, mentiin siis vasemmalle. Päivitetään $hb_2 = hb_1/2 = 1$ ja $t_2 = t_1$. Testataan, onko silmukkaehto $hb_2 \neq 0$ tosi, mutta nyt $t_2 = 1 < 1 = hb_2$ se onkin epätosi. Edetäänkin oikeaan haaraan ja päivitetään arvot $v_3 = oikea(v_2)$, $t_3 = t_2 - hb_2 = 1 - 1 = 0$ ja $hb_3 = hb_2/2 = 1/2 = 0$. Silmukkaehto $hb_3 \neq 0$ ei enää täyty, joten jatketaan silmukan jälkeistä suoritusta ja palautetaan alkio $v_3 = oikea(vasen(juuri))$.

Proseduurit LISÄÄ JA NOSTA sekä POISTA JA LASKE toteutetaan taulukon indeksiin *koko* viittaamisen sijaan linkitettyinä versioina LISÄÄ JA NOSTA LINKITETTYNÄ ja POISTA JA LASKE LINKITETTYNÄ etsimällä indeksille vastaava alkion vanhempi proseduurilla ETSI ALKIO INDEKSILLE vastaavalla parametreilla $koko/2$. Lisääminen tai poistaminen kohdistetaan muuttujan *koko* ollessa parillinen vasempaan lapseen ja sen ollessa pariton oikeaan lapseen. Proseduurit NOSTA ja LASKE korvataan vastaavilla proseduureilla NOSTA LINKITETTYNÄ ja LASKE LINKITETTYNÄ, jotka parametrisoi-

daan kyseisellä alkiolla a ja lausekkeiden $\lfloor i/2 \rfloor$, $2i$ ja $2i+1$ sijaan käytetään vastaavia linkkeja *vanhempi*(a), *vasen*(a) ja *oikea*(a). Linkitettyjen proseduurien ideat ovat samoja kuin taulukoidut vastineensa, joten algoritmeja ei esitetä.

3.2 Vasenta suosiva keko

Tarjanin (1983) mukaan Clark A. Cranen keksimä rakenne on linkitetty binääripuu kekoehdolla ja suosivuusehdolla. Rakenne on kehitetty kahden kekoehtoisen puun tehokasta yhdistämistä varten.

Binääripuut voidaan helposti yhdistää, jos edes toinen juuren lapsista on vapaa. Tällöin toinen puu asetetaan vain vapaalle paikalle. Tämä pätee aina puun lehtitasolla ja sille päästään nopeimmin, kun kuljetaan alkioiden määrältään lyhintä polkua pitkin. Tähän alipuiden vertailemiseen sopiva mittari on *mataluus*. Vasenta suosivassa keossa vasen alipuu on vähintään yhtä matala kuin oikea.

Kekoehtoisten binääripuiden yhdistämisessä puista prioriteetiltaan pienempi valitaan kerrallaan mukaan yhdistettyyn puuhun. Molempien lapsien ollessa varattuja valitaan oikea lapsi *tarkasteltavan alipuun juureksi*. Tarkasteltavan alipuun juurta verrataan yhdistettävän puun juureen ja näistä prioriteettiehdon mukainen otetaan lapseksi yhdistettyyn puuhun ja toisesta tulee uusi yhdistettävä puu, jota pyritään yhdistämään edellisen alipuuhun.

Jotta samanaikaisilta prioriteetin ja mataluuden vertailuilta vältyttäisiin vasenta suosivassa keossa yhdistäminen tehdään kahdessa vaiheessa. Ensimmäisessä hyödynnetään tietoa, että oikea alipuu on mahdollisesti matalampi ja kuljetaan vain oikean reunan polkua puita yhdistellen. Toisessa vaiheessa, kun puut on yhdistetty palataan muuttamaan mahdollisesti muuttuneet alkioiden mataluudet ja varmistetaan, että vasemman lapsen mataluus on vähintään oikean lapsen mataluus.

Kuvassa 8 on vasenta suosivat keot u ja v, jotka yhdistetään. Aluksi havaitaan, että keon v juuri a on pienempi kuin keon u juuri c. Valitaan yhdistetyksi puuksi a ja yhdis-

tettäväksi puuksi c. Ensimmäisessä vaiheessa tarkastellaan tilannetta sopisiko alkio c alkion a oikeaksi lapseksi. Alkion a oikeana lapsena on alkio i, joka olkoon suurempi kuin c, joten se saa väistyä ja sen tilalle sijoitetaan alkio c. Seuraavaksi siirrytään tilanteeseen, jossa alkiota i koetetaan sijoittaa edellisen muutetun alkion c oikealle puolelle. Nyt lisättävä alkio i on suurempi kuin tarkasteltava alkio d, joten alkioita ei vaihdeta keskenään. Sijoitetaan alkio i alkion d oikeaksi lapseksi. Tässä tilanteessa alkiolla d ei ole oikean puoleista alipuuta ja niinpä alkiosta i tehdään sellainen. Toisessa vaiheessa on keot yhdistetty, mutta uusi rakenne ei vielä suosi vasenta. Palaamalla muuttunutta polkua pitkin (kuvassa tummennettu), päivittämällä mataluudet (alkion muuttuja), korjaamalla rikkoutunut suosivuusehto (kuvassa taitettu nuoli) vaihtamalla vasemman ja oikean lapsen paikkaa saadaan lopuksi vasenta suosiva keko.

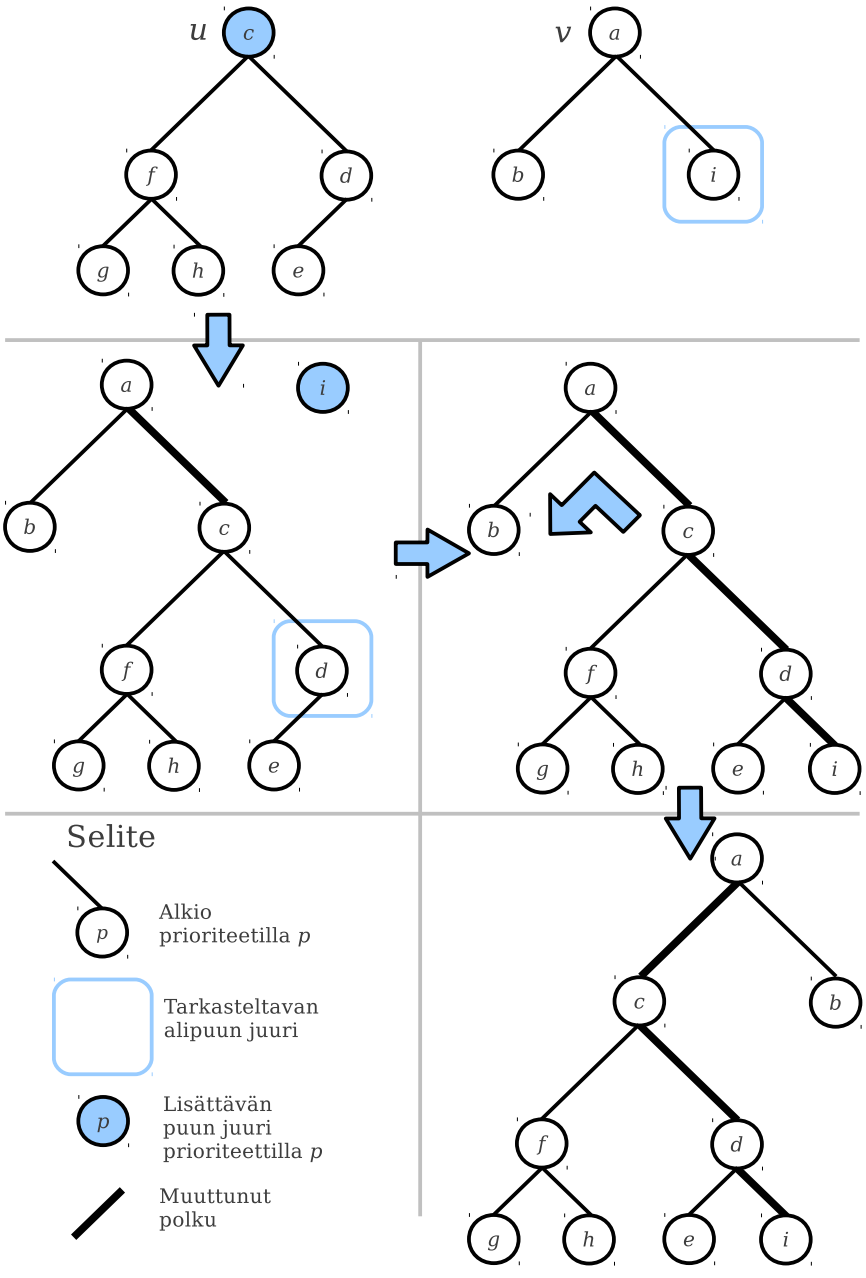
3.2.1 Ahkerasti

Kuvassa 9 on vasenta suosivan keon rakenne, joka toteuttaa prioriteettijonon rajapinnan. Kuvasta näkee kuinka keskeinen proseduuri YHDISTÄ on vasenta suosivassa keossa, koska sitä kutsutaan kaikissa tapauksissa. Vasenta suosivasta keosta tiedetään puun *juuri*, alkioiden lukumäärä *koko* ja kahvat $kahvat[1..n]$. Operaatiosta LISÄÄ UUSI vastaa LISÄÄ YHDISTÄEN, operaatiosta PÄIVITÄ PRIORITEETTI vastaa POISTA JA LISÄÄ YHDISTÄEN ja operaatiosta POISTA TÄRKEIN vastaa POISTA YHDISTÄEN.

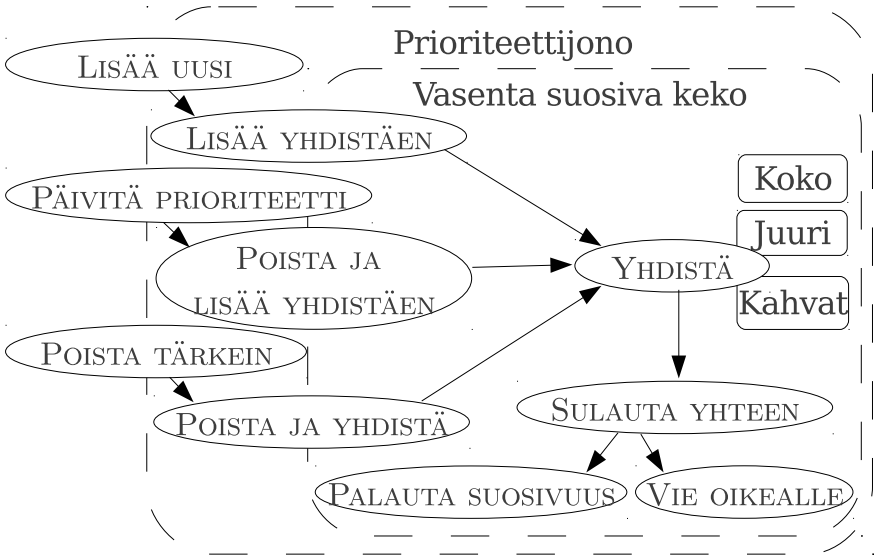
Proseduurilla LISÄÄ YHDISTÄEN (**Algoritmi 13**) lisätään uusi alkio kekoon yhdistämällä vanha keko ja uusi alkio uudeksi yhtä alkiota suuremmaksi keoksi. Uutta alkiota käsitellään kuin se olisi toinen puu, joka yhdistetään nykyisen puun kanssa proseduurilla YHDISTÄ.

Algoritmi 13 Proseduuri uuden alkion lisämiseen vasenta suosivaan kekoon. $O(\log koko)$

```
procedure LISÄÄ YHDISTÄEN(Vasenta suosiva keko (juuri, kahvat[1..n], koko),
Alkio u)
    koko ← koko + 1
    kahvat[identiteetti(u)] ← u
    juuri ← YHDISTÄ(juuri, u)
end procedure
```



Kuva 8: Vasenta suosivien kekojen u ja v yhdistäminen vaiheittain.



Kuva 9: Vasenta suosivan keon rakenne.

Yhdistäminen tapahtuu proseduurilla YHDISTÄ (**Algoritmi 14**). Jos jompi kumpi yhdistettävistä puista on tyhjä, niin yhdistämisen tulos on toinen yhdistettävistä puista. Muussa tapauksessa suurempijuurinen puu sulautetaan pienempijuuriseen proseduurilla SULAUTA YHTEEN (**Algoritmi 15**).

Algoritmi 14 Proseduuri, joka palauttaa kahden vasenta suosivan keon yhdisteen. $O(\log koko)$

```
procedure YHDISTÄ(Alkio u, Alkio v)
    if u = null then
        return v
    else if v = null then
        return u
    else if prioriteetti(u) < prioriteetti(v) then
        return SULAUTA YHTEEN(u, v)
    else
        return SULAUTA YHTEEN(v, u)
    end if
end procedure
```

Proseduuri SULAUTA YHTEEN (**Algoritmi 15**) saa yhdistettäväksi puut u ja v. Puu v yhdistetään puuhun u. Proseduuri VIE-OIKEALLE palauttaa oikeareunaisimman alkion t yhdistetystä puusta yhdistämisen jälkeen. Proseduuri PALAUTA SUOSIVUUS korjaa

suosivuusehdon ja päivittää mataluudet matkalla alkiosta t puun juureen u.

Algoritmi 15 Proseduuri yhdistää kaksi vasenta suosivaa kekoa. O(log *koko*)

```
procedure SULAUTA YHTEEN(Alkio u, Alkio v)
    Alkio t ← VIE-OIKEALLE(u, v)        ▷ Puu yhdistetyn puun oikeassa alipuussa
    return PALAUTA SUOSIVUUS(vanhempi(u), t)
end procedure
```

Proseduurilla VIE-OIKEALLE (**Algoritmi 16**) yhdistetään puut kekoehto säilyttäen. Yhdistettävän puun juurta verrataan tarkasteltavan alkioon ja vaihdetaan mikäli juuri on tarkasteltavaa pienempi. Muuten tarkastellaan seuraavaa oikean puoleista lasta. Jollei tarkasteltavaa alipuuta ole, niin yhdistettävästä puusta tehdään yhdistetyn puun alipuu.

Algoritmi 16 Proseduuri, joka kulkee oikeaa laitaa yhdistettävää puuta ja valitsee joka välissä pienimmän alkion jatkamaan yhdistelmäpuuta. O(log *koko*)

```
procedure VIE-OIKEALLE(Alkio u, Alkio v)
    while u:lla on oikealla lapsi do
        r ← oikea(u)
        if prioriteetti(r) < prioriteetti(v) then
            u ← r
        else
            Asetetaan alkion u oikeaksi lapseksi v.
            u ← v
            v ← r
        end if
    end while
    Asetetaan alkion u oikeaksi lapseksi v.
    return u;
end procedure
```

Suosivuus palautetaan proseduurilla PALAUTA SUOSIVUUS (**Algoritmi 17**). Proseduuri palaa yhdistettyä polkua pitkin ja päivittää muuttuneet mataluudet alkioihin. Havaittu suosivuusehdon rikkoutuminen korjataan vaihtamalla alipuut keskenään.

Prioriteetin päivittämiseen ei vasenta suosivassa keossa kannata käyttää nostamista, sillä puun korkeus ei ole logaritminen alkioiden määrään nähden. Prioriteetin päivittäminen tehdään alkion paikallisena poistamisena ja uuden alkion lisäämisenä. Kuvassa 10 paikallinen poisto kohdistuu vasemman puun viimeiseen alkioon g. Alkion g alipuiden yhdistelmä on tyhjä ja sen mataluus on siis pienempi kuin alkion g mataluus.

Algoritmi 17 Proseduuri, joka korjaa suosivuusehdon nousemalla yhdistettyä polkua pitkin juurelle. O(log *koko*)

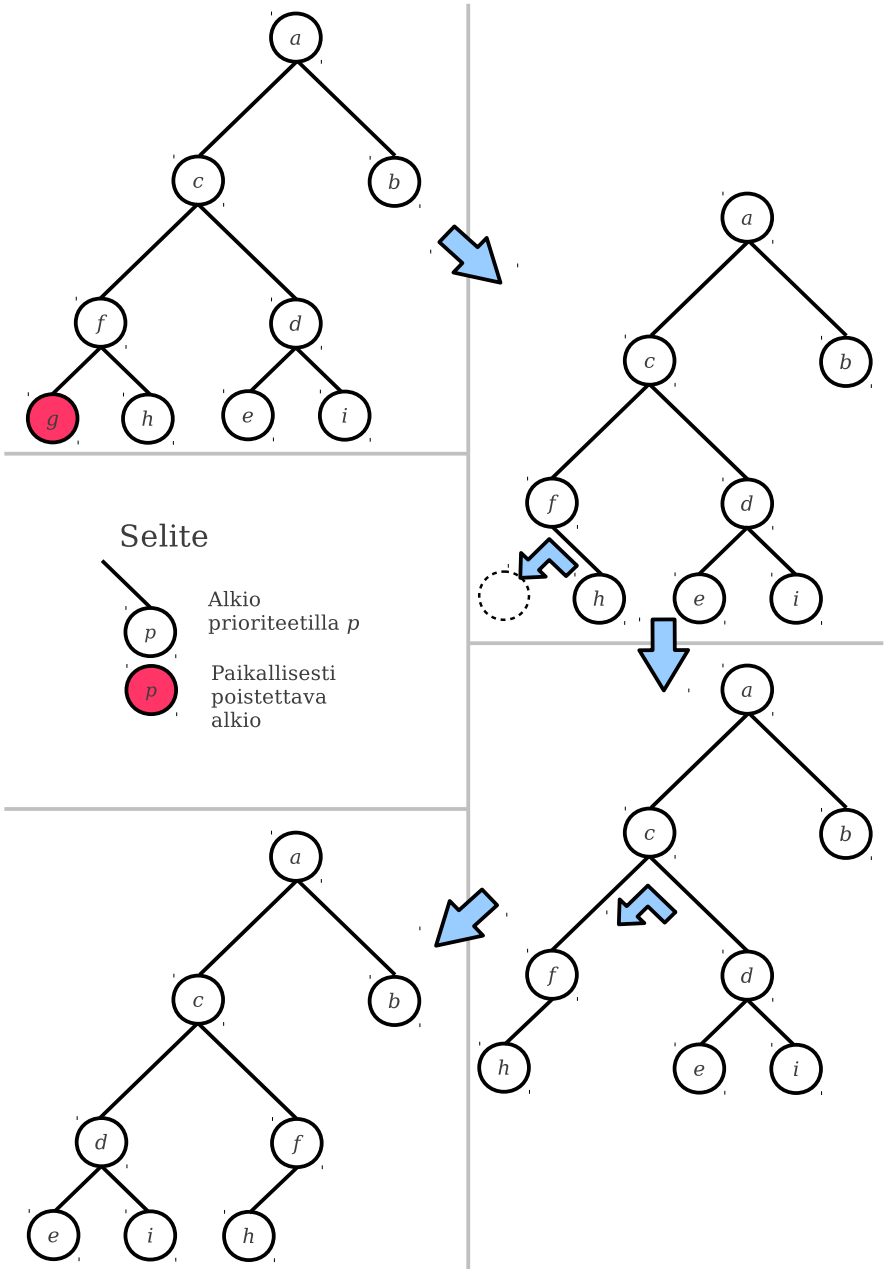
```
procedure PALAUTA SUOSIVUUS(Alkio u, Alkio v)
    repeat
        if mataluus(vasen(v)) < mataluus(oikea(v)) then
            Vaihdetaan keskenään vasen(v) ja oikea(v)
        end if
        v ← vanhempi(v)
    until u = v
    return u
end procedure
```

Alkion g poistaminen johtaa ketjureaktioon, jossa mataluuden pienentyminen toistuu saaden aikaan suosivuusehdon rikkoutumisia. Suosivuusehto rikkoutuu ensin alkiossa f ja sitten alkiossa c. Alkion a mataluus ei muutu eikä suosivuusehto rikkoudu. Ketjureaktio pysähtyisi alkioon a riippumatta sen sijainnista keossa. Ketjureaktio pystyy jatkumaan vain niin pitkään kuin alipuut ovat yhtä matalia.

Prioriteetin päivittäminen tehdään proseduurilla POISTA JA LISÄÄ YHDISTÄEN (**Algoritmi 18**), joka on Tarjanin (1983, luku 3.2) mukaan mahdollista logaritmisella aikavaativuudella, mutta lähde ei sisällä toteutusta. Tutkielmassa laaditun toteutuksen aluksi hyödynnetään kahvaa, jotta löydetään käsiteltävä alkio t. Jos t on *juuri* tai uusi prioriteetin arvo ei ole vanhemman prioriteetin arvoa pienempi, niin päivitetään prioriteetti paikallaan. Muuten tehdään paikallinen poisto päivittämällä kahvan arvo uuteen alkioon, yhdistämällä alkion t vasen ja oikea alipuu keskenään ja asettamalla yhdistetyn puun juurialkio t' alkion t paikalle.

Paikallinen poisto saattaa aiheuttaa ketjureaktion, jossa mataluuden muutos alkioiden t ja t' välillä saattaa muuttaa alkion t vanhemman v_1 mataluutta, joka vaikuttaa vastaavasti vanhempiinsa v_x ja saattaa rikkoa suosivuusehdon. Mataluuksia tarvitsee korjata ja suosivuusehto palauttaa polulla alkiosta t' juureen ensimmäiseen alkioon asti, jonka mataluus ei muutu. Koska alkioiden määrä ei kaikilla poluilla ole logaritminen rakenteen alkioiden määrän nähden, niin tarkastellaan seuraavaksi tilanteet aikavaativuuden perustelemiseksi.

Alkion v_x mataluuden muutostapauksia on kaksi, joissa kummassakin on kolme tar-



Kuva 10: Paikkallinen poisto vasenta suosivassa keossa.

Algoritmi 18 Proseduuri alkion prioriteetin päivittämiseksi korvaamalla tai tarvittaessa poistamalla ja lisäämällä uusi.O(log *koko*)

procedure POISTA JA LISÄÄ YHDISTÄEN(Vasenta suosiva keko $(juuri, kahvat[1..n])$, Alkio u, Alkio v)
  $t \leftarrow kahvat[identiteetti(v)]$
  if $t = juuri$ **or** $prioriteetti(u) > prioriteetti(vanhempi(t))$ **then**
    $prioriteetti(t) \leftarrow prioriteetti(u)$ ▷ korvaaminen ei muuta kekoa
  else
    $kahvat[identiteetti(v)] \leftarrow u$
    $t' \leftarrow$ YHDISTÄ$(vasen(t), oikea(t))$ ▷ alkion t paikallinen poistaminen
    Asetetaan alkion t tilalle alkio t'
    Korjataan suosivuusehto ensimmäiseen alkioon asti, jonka mataluus ei muutu.
    $juuri \leftarrow$ YHDISTÄ$(juuri, u)$ ▷ uuden lisääminen
  end if
end procedure

kasteltavaa kohtaa. Ehdot tapauksien alussa ovat, että mataluus alussa kasvaa eli $mataluus(t) < mataluus(t')$ tai $mataluus(v_x)$ pienentyy korkeintaan yhdellä keon koon pienentyessä yhdellä eli $mataluus(oikea(t)) = mataluus(t) - 1 \leq mataluus(t')$.

Ensiksi tarkastellaan mataluuden kasvua koskevat ehdot.

- Jos alkion v_x oikean lapsen mataluus kasvaa ja suosivuusehto säilyy, niin alkion mataluus kasvaa saman verran kuin oikean lapsen mataluus kasvoi.
- Jos alkion v_x oikean lapsen mataluus kasvaa ja suosivuusehto rikkoutuu, niin alkion mataluus kasvaa lasten mataluuksien erotuksen verran ja suosivuusehto korjataan vaihtamalla lasten paikat.
- Jos alkion v_x vasemman lapsen mataluus kasvaa, niin sillä ei ole vaikutusta alkion mataluuteen. Algoritmi pysähtyy tässä tapauksessa.

Koska oikeanpuoleisia lapsia on vasenta suosivan keon poluilla logaritminen määrä, mataluuden kasvuun liittyvät tapaukset päättyvät logaritmisella määrällä alkiotarkasteluja.

Toiseksi tarkastellaan mataluuden pienentymistä koskevat kohdat.

- Jos alkion v_x oikean lapsen mataluus pienentyy, niin alkion mataluus pienentyy saman verran kuin oikean lapsen mataluus pienentyi.

- Jos alkion v_x vasemman lapsen mataluus pienentyy ja suosivuusehto säilyy, niin se ei vaikuta alkion mataluuteen. Algoritmi pysähtyy tässä tapauksessa.

- Jos alkion v_x vasemman lapsen mataluus pienentyy ja suosivuusehto rikkoutuu, niin alkion mataluus pienentyy lasten mataluuksien erotuksen verran. Tapaus toteutuu vain, jos oikea ja vasen lapsi olivat olleet yhtä matalia.

Oikean lapsen tapaus on logaritminen kuten edellä todettiin. Yhtä matalien alipuiden tapauksessa puiden alkiomäärä kasvaa eksponentiaalisesti suhteessa puun mataluuteen, joten polulla tarkasteltavien alkioiden määrä on oltava logaritminen suhteessa kokorakenteen alkioiden määrään. Paikallinen poistaminen on aikavaativuudeltaan $\mathrm{O}(\log koko)$.

Algoritmin POISTA JA LISÄÄ YHDISTÄEN lopuksi paikallisen poiston jälkeen lisätään uusi alkio proseduurilla YHDISTÄ. Algoritmin aikavaativuus on kokonaisuudessaan $\mathrm{O}(\log koko)$.

Tarjanin (1983, luku 3.2) mukaan tärkeimmän alkion poistaminen keosta tehdään proseduurilla POISTA JA YHDISTÄ (**Algoritmi 19**). Proseduuri YHDISTÄ palauttaa kekoehtoisen vasenta suosivan keon, joten tärkeimmän poistamisessa riittää, että yhdistetään juuren vasen alipuu oikean alipuun kanssa uudeksi keoksi ja asetetaan se uudeksi *juureksi*. Operaation aikavaativuus on seuraus proseduurin YHDISTÄ aikavaativuudesta eli $\mathrm{O}(\log koko)$.

Algoritmi 19 Proseduuri tärkeimmän alkion poistamiseksi keosta.

```
procedure POISTA JA YHDISTÄ(Vasenta suosiva keko (juuri, kahvat[1..n], koko))
    vastaus ← juuri
    kahvat[identiteetti(juuri)] ← null
    juuri ← YHDISTÄ(vasen(vastaus), oikea(vastaus))
    koko ← koko − 1
    return vastaus
end procedure
```

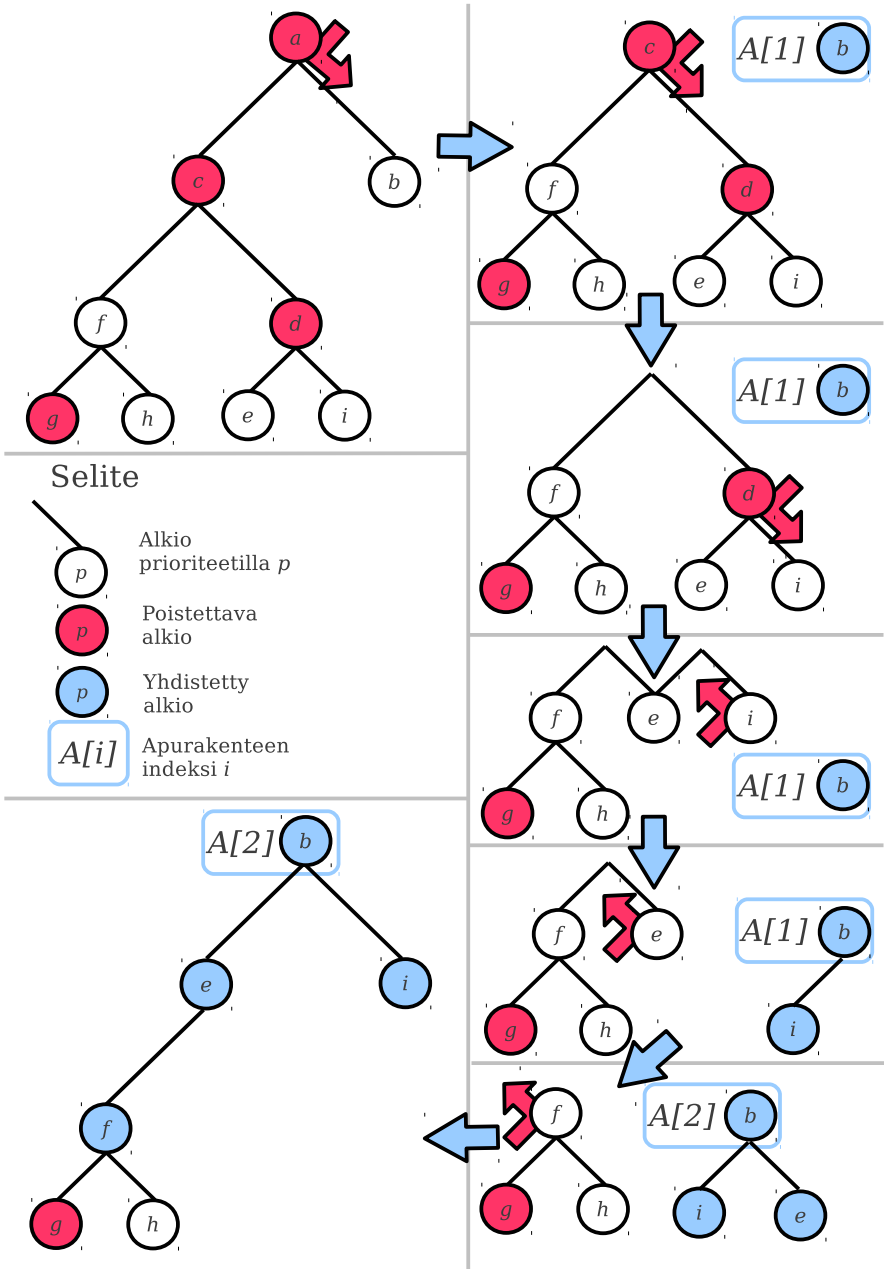
3.2.2 Implisiittisellä poistolla

Tässä luvussa esitetään hieman Tarjanin (1983) esityksestä poikkeava tapa toteuttaa tärkeimmän alkion poistaminen. Tarjan on saanut esittämälleen algoritmille aikavaativuudeksi $O(k \max\{1, \log(koko/(k+1))\})$, jossa k on uusien alkioiden ja keon päällä sijaitsevien epäoleellisten alkioiden lukumäärä. Keon ollessa tyhjä keon rakentaminen pinosta (puskuri) tehdään samoin kuin *kekoaminen* (eng. *heapify*, $O(k)$). Jos keon päällä ei juuren lisäksi ole epäoleellisia alkioita toiminta ei eroa siitä, mitä proseduuri POISTA JA YHDISTÄ tekee.

Rakenteesta tunnetaan puun *juuri*, uusien alkioiden pino *pino*, $oleellisuus[1..n]$ ja lisäksi tarvitaan apurakenne $A[1..\log m]$. Tarjanin versiossa käytetään apurakenteena linkitettyä listaa ja apuoperaationa laiskaa lisäämistä (eng. *lazy-meld*). Linkitetty lista korvataan seuraavassa logaritmisen mittaisella taulukolla, jonka indeksi vastaa siinä esiintyvän puun mataluutta. Yhdistettävän puun kanssa yhtä matalan puun löytäminen taulukosta on nopeaa ja yhdistäminen on tehokkaampaa kuin yksitellen alkioiden lisääminen yhdistämällä. Algoritmi tekee yhtä monta yhdistäoperaatiota kuin Tarjanin versio, mutta yhdistää parhaimman tapauksen (yhtä matalia) puita keskenään. Algoritmin aikavaativuuteen optimoinnin ei pitäisi vaikuttaa.

Kuvassa 11 on implisiittinen poistaminen keon epäoleellisten alkioiden poistamisen osalta. Epäoleellisten alkioiden poistaminen etenee juuresta alkaen järjestyksessä oikealta vasemmalle syvyyshakuna. Alkion ollessa oleellinen, kuten b, se yhdistetään taulukon indeksissä $mataluus(b) = 1$ vastaavan puun kanssa. Ensimmäinen taulukkoon lisätty on b ja seuraava i. Näiden yhdistelmä on edelleen mataluudeltaan 1. Yhdistettynä alkion e kanssa yhdistelmän mataluus kasvaa arvoon 2, joka tulee yhdistelmän uudeksi indeksiksi. Samoin puun f mataluus on 2, joten lopputuloksena b ja f yhdistetään vasenta suosivaksi keoksi.

Proseduurilla PÄÄLLYKSEN LÄPIKÄYNTI (**Algoritmi 20**) tehdään vasenta suosivaan kekoon tärkeimmän alkion poistaminen. Keon päältä poistetaan epäoleellisia alkioita proseduurilla PUHDISTA KEKO, koska niillä ei ole merkitystä ja ensimmäinen oleel-



Kuva 11: Implisiittinen poistaminen vasenta suosivassa keossa.

linen alkio seuraavalla kerralla löytyisi ilman lineaarista etsintää. Pinossa odottavat alkiot lisätään kekoon proseduurilla TALLETUKSEN PURKU ja merkataan oleellisiksi. Lopuksi apurakenteen puut yhdistetään vasenta suosivaksi keoksi proseduurilla YHDISTÄ TAULUKOSTA.

Algoritmi 20 Proseduuri läpikäymään keko ja palauttamaan pienin. $O(k \max\{1, \log(koko/(k+1))\})$, jossa k on uusien alkioiden ja keon päällä sijaisevien epäoleellisten alkioiden lukumäärä.

```
procedure PÄÄLLYKSEN LÄPIKÄYNTI(Vasenta suosiva keko (juuri, pino, oleellisuus[1..n], koko))
    koko ← koko − 1
    vastaus ← min(juuri, pino)
    Merkataan oleellisuus[identiteetti(vastaus)] epäoleelliseksi.
    Olkoon A[1..log m] aputaulukko.
    PUHDISTA KEKO(juuri, oleellisuus, A)
    if vastaus = pino then
        TALLETUKSEN PURKU(alla(pino), oleellisuus, A)
    else
        TALLETUKSEN PURKU(pino, oleellisuus, A)
    end if
    juuri ← YHDISTÄ TAULUKOSTA(A, koko)
    return vastaus
end procedure
```

Proseduurilla PUHDISTA KEKO (**Algoritmi 21**) käydään läpi keon päällimmäiset alkiot. Epäoleelliset alkiot poistetaan ja oleelliset alkiot yhdistetään apurakenteeseen A proseduurilla YHDISTÄ TAULUKKOON (**Algoritmi 22**). Proseduuri ei matkaa syvemmälle kuin ensimmäisiin oleellisiin alkiohin, joten kaikkia epäoleellisuuksia ei keosta poisteta kerralla.

Proseduuri YHDISTÄ TAULUKKOON (**Algoritmi 22**) yhdistää puun ja taulukossa olevan yhtä matalan puun keskenään. Jos yhdistelmän mataluus on suurempi kuin d, niin yhdistelmä yhdistetään seuraavan vastaavan mataluuden indeksissä olevan puun kanssa ja tätä toistetaan kunnes mataluus ei enää kasva. Toistoa voi tulla korkeitaan logaritminen määrä taulukon puissa olevien alkioiden lukumäärään nähden, koska taulukon pituus on rajattu ja yhtä matalien puiden yhdistymisessä yhdistetyn puun mataluus ei ole pienempi kuin yhdistettävien. Toistoa tulee kuitenkin harvoin, koska jokaisessa yhdistettävässä puussa on vähintään $2^{mataluus}$ alkiota.

Algoritmi 21 Proseduuri poistamaan epäoleelliset alkiot keon päältä. $O(k \max\{1, \log(koko_{keko}/k+1))\})$, jossa k on epäoleellisten alkioiden lukumäärä keon päällä ja $koko_{keko}$ alkioiden määrä keossa.

```
procedure PUHDISTA KEKO(Alkio t, Oleellisuus oleellisuus[1..n], Apurakenne A[1..log m])
    Seuraava alkio s ← null
    while t ≠ null do
        if Alkio t on oleellinen then
            s ← vanhempi(t)
            vanhempi(t) ← null
            YHDISTÄ TAULUKKOON(t, A)
        else if Alkiolla t on oikea lapsi then        ▷ Jos on oikea, niin on vasenkin
            vanhempi(vasen(t)) ← vanhempi(t)
            vanhempi(oikea(t)) ← vasen(t)
            s ← oikea(t)
            koko ← koko − 1
        else if Alkiolla t on vasen lapsi then
            vanhempi(vasen(t)) ← vanhempi(t)
            s ← vasen(t)
            koko ← koko − 1
        else
            s ← vanhempi(t)
            koko ← koko − 1
        end if
        t ← s
    end while
end procedure
```

Algoritmi 22 Proseduuri yhdistämään puu vastaavan mataluuden taulukon puuhun.

```
procedure YHDISTÄ TAULUKKOON(Alkio t, Apurakenne A[1..log m])
    vanhempi(t) ← null                                  ▷ varmistus
    d ← mataluus(t)
    while Apurakenteessa paikassa A[d] on alkio do
        t ← YHDISTÄ(A[d], t)
        A[d] ← null
        d ← mataluus(t)
    end while
    A[d] ← t
end procedure
```

Produurissa TALLETUKSEN PURKU (**Algoritmi 23**) käydään läpi uusien alkioiden pino ja lisätään alkiot taulukkoon A muodostaen puita proseduurilla YHDISTÄ TAULUKKOON (**Algoritmi 22**), missä taulukon indeksi vastaa puun mataluutta.

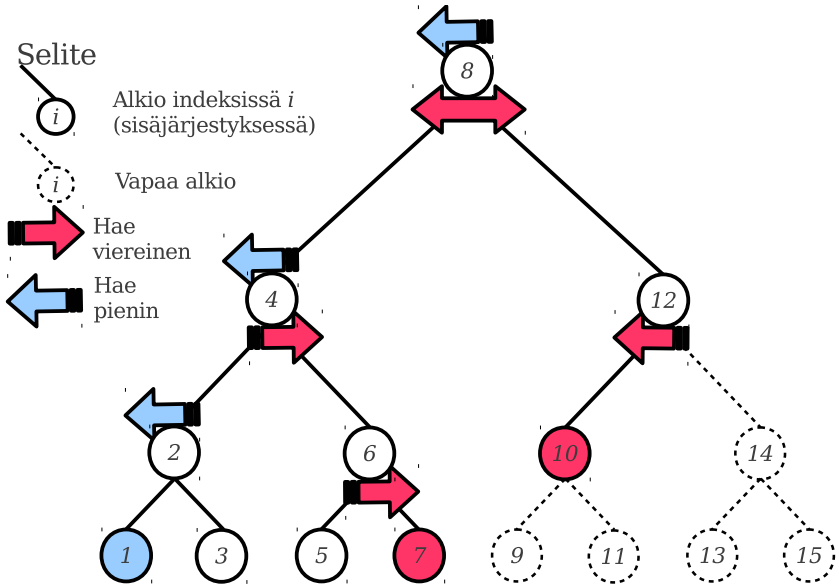
Algoritmi 23 Proseduuri pinoon tallennettujen alkioiden purkamiseen taulukkoon. $O(koko_{pino} \max\{1, \log koko_A/k+1\})$, jossa $koko_{pino}$ on pinon koko, $koko_A$ on alkioiden määrä apurakenteessa ja k on puiden määrä apurakenteessa.

```
procedure TALLETUKSEN PURKU(Alkio pino, Oleellisuus oleellisuus[1..n], Apurakenne A[1..log m])
    while Pinossa pino on jäseniä do                    ▷ pino ≠ null
        t ← pino
        pino ← alla(pino)
        alla(t) ← null
        YHDISTÄ TAULUKKOON(t, A)
    end while
end procedure
```

Taulukosta A yhdistetään puut uudeksi keoksi proseduurissa YHDISTÄ TAULUKOSTA (**Algoritmi 24**) ja keon pienimmäksi alkioksi saadaan uusi *juuri*. Tälle proseduurille ei esitetä aikavaativuutta monimutkaisen luonteensa vuoksi.

Algoritmi 24 Proseduuri yhdistämään aputaulukon puut.

```
procedure YHDISTÄ TAULUKOSTA(Apurakenne A[1..log m], Alkiomäärä koko)
    juuri ← null
    for i ∈ [1, log(koko + 1)] do
        if A[i] ≠ null then
            juuri ← YHDISTÄ(juuri, A[i])
            A[i] ← null
        end if
    end for
    return juuri
end procedure
```



Kuva 12: Hakuja hakupuuhun.

3.3 AVL-puu

Knuthin (1973, luku 6.2.3) mukaan G. M. Adel'son-Vel'skiin ja E. M. Landisin kehittämä rakenne on binääripuu, jossa on voimassa haku- ja tasapainoehto. Koska kyseessä on hakupuu, niin on tarvetta käyttää avainta yksilöimään alkiot. Hakuehto mahdollistaa alkion etsimisen avaimen perusteella ja tasapainoehto pitää rakenteen korkeuden ja sitä kautta perusoperaatioiden aikavaativuudet logaritmisina.

Hakuehdosta seuraa, ettei kekorakenteista poiketen prioriteetiltaan pienin alkio ole puun juurena, vaan äärimmäisenä siinä suunnassa, johon hakuehto sen pakottaa. Tässä suunnaksi on valittu vasen. Kuvassa 12 on pienimmän prioriteetin haku (sinisellä) ja juuren viereisien alkioiden haut (punaisella). Viereinen tarkoittaa (sisäjärjestyksessä) *edeltäjää* tai *seuraajaa*.

Tasapainoehto korjataan rikkoutuessaan *kierroilla*, jotka eivät riko hakuehtoa. Kiertoja tarvitaan lisäämisessä tapahtuneen rikkoutumisen korjaamiseen korkeintaan kaksi ja poiston jälkeiseen korjautumiseen korkeintaan logaritminen määrä alkioiden määrään nähden.

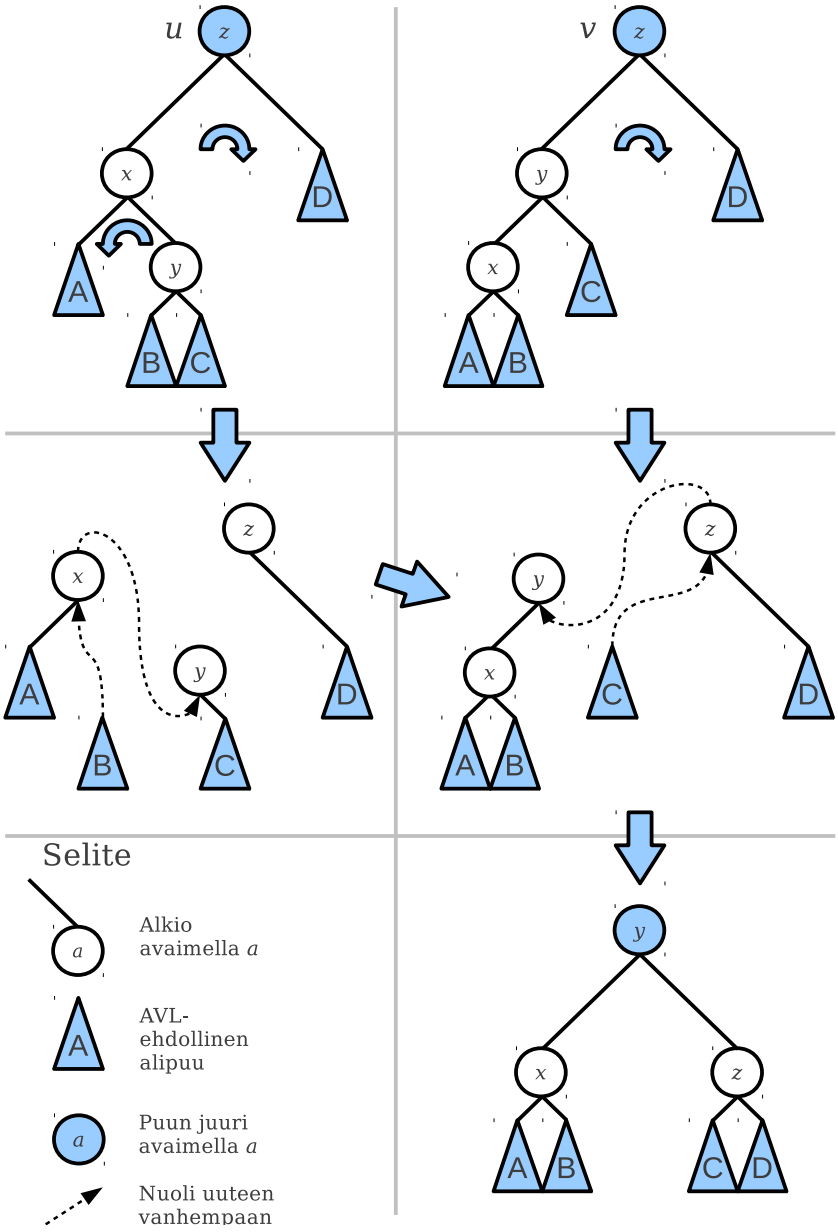
Tarkastellaan seuraavaksi tilanteet, joissa tasapainoehdon korjaaminen esiintyy. Kuvassa 13 on kaksi binääripuuta u ja v. Olkoon alkion z alipuut AVL-ehtoisia eli alkio z on jälkijärjestyksessä ensimmäinen alkio, joka saattaa rikkoa tasapainoehdon. Jos alkion z tasapainotila, mikä on alipuiden korkeuksien erotus, on -1, 0, tai 1, niin se ei riko tasapainoehtoa ja puut ovat tasapainoisia. Jos alkion z tasapainotila on -2, niin tasapaino voidaan korjata korkeintaan kahdella kierrolla (kuvassa kaarinuolet). Tasapainotilan arvolla 2 tarkasteltaisiin puiden u ja v peilikuvia, joten kierrot tehtäisiin peilikuvina. Pienempiä tasapainotilan arvoja kuin -2 tai suurempia arvoja kuin 2 ei käsitellä, koska yhden alkion lisääminen tai poistaminen kasvattaa tai pienentää tilaa yhdellä, joten aina päädytään ennemmin korjaamaan ehto muuten.

Kuvassa 13 esitetään puiden u ja v tasapainon korjaaminen, kun tasapainoehto rikkoutuu alkiossa z, tasapainotilan arvolla -2. Tarkastellaan alkion z vasemman lapsen tasapainotilaa. Arvoilla 0 tai -1 tarkastellaan tilannetta, kuten puussa v, jonka korjaamiseksi tarvitaan yksi kierto oikealle (kaarinuoli). Tasapaino korjataan liittämällä alipuu C alkion z vasemmalle ja alkio z alkion y oikealle, ja näin saatu puu on alkion y suhteen tasapainossa. Kun alkion z vasemman lapsen tasapainotila on 1, tarkastellaan tapausta u, joka korjataan kahdella kierrolla (yksi vasemmalle ja toinen oikealle). Ensiksi liitetään alipuu B alkion x oikealle ja alkio x alkion y vasemmalle puolelle. Tällä tavoin korjattuna tilanne on sama kuin puun v tapauksessa ja sille tehdään sama korjaus, jolla saadaan tasapainoehdon täyttävä puu.

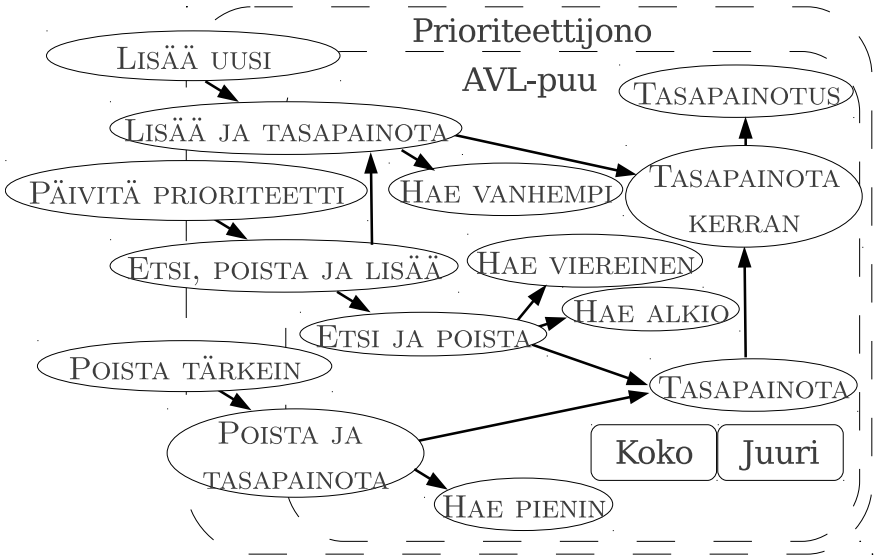
3.3.1 Ahkerasti

Knuthin kirjasta (1973, luku 6.2.3) löytyy huolellisesti laadittu analyysi ja perustelut AVL-puun ehtojen ja logaritmisten vaativuuksien pitävyydelle lisäyksen ja poistamisen yhteydessä. Rakenteesta tiedetään puun *juuri* alkio ja *koko*. Puun korkeus h on rajoitettu $\log_2 koko \leq h \leq \log_\phi koko$, jossa $\phi = \frac{\sqrt{5}+1}{2}$ on kultainen leikkaus ja aikavaativuuksien osoittaminen perustuu näihin rajoihin.

Kuvassa 14 on prioriteettijonon rajapinta, jonka AVL-puu toteuttaa. Kuvasta nähdään,



Kuva 13: AVL-puiden *u* ja *v* tasapainottavat kierrot vaiheittain.



Kuva 14: AVL-puun rakenne.

että operaatiot koostuvat erilaisista hauista ja tasapainoittavista proseduureista. Operaatiosta LISÄÄ UUSI vastaa proseduuri LISÄÄ JA TASAPAINOTA, operaatiosta PÄIVITÄ PRIORITEETTI vastaa proseduuri ETSI, POISTA JA LISÄÄ ja operaatiosta POISTA TÄRKEIN vastaa proseduuri POISTA JA TASAPAINOTA.

Proseduuri LISÄÄ JA TASAPAINOTA (**Algoritmi 25**) lisää uuden alkion AVL-puuhun ja korjaa tasapainoehdon. Epätyhjään puuhun lisäämisessä alkiolle haetaan vanhempi proseduurilla HAE VANHEMPI (**Algoritmi 26**). Alkion lisääminen tapahtuu avaimen arvosta riippuen vanhemman vasemmaksi tai oikeaksi lapseksi hakuehto säilyttäen, jonka jälkeen korjataan tasapainoehto proseduurilla TASAPAINOTA KERRAN (**Algoritmi 27**).

Proseduuri HAE VANHEMPI (**Algoritmi 26**) etsii alkion, jonka lapseksi voisi lisätä hakuehtoon sopivan alkion. Viimeinen sopiva vanhempi palautetaan.

Proseduuri TASAPAINOTA KERRAN (**Algoritmi 27**) palauttaa tasapainoehdon voimaan lisäämisen jälkeen. Alkioita tarkastetaan puussa ylöspäin kulkemalla, kunnes päädytään juureen tai löydetään rike, joka voidaan korjata proseduurilla TASAPAINOTUS (**Algoritmi 28**).

Algoritmi 25 Proseduuri uuden alkion lisäämiseen AVL-puuhun. O(log *koko*)

```
procedure LISÄÄ JA TASAPAINOTA(AVL-puu (juuri, koko), Alkio v )
    koko ← koko + 1
    if juuri = null then
        juuri ← v
    else
        t ← HAE VANHEMPI(juuri, avain(v))
        if avain(v) < avain(t) then
            Asetetaan alkio v alkion t vasemmaksi lapseksi.
        else
            Asetetaan alkio v alkion t oikeaksi lapseksi.
        end if
        t ← TASAPAINOTA KERRAN(juuri, t)
        if vanhempi(t) = null then
            juuri ← t
        end if
    end if
end procedure
```

Algoritmi 26 Proseduuri alkion *u* vanhemman hakemiseen. O(log *koko*)

```
procedure HAE VANHEMPI(Alkio u,Avain k)
    t ← u
    repeat
        u ← t
        if k < avain(t) then
            t ← vasen(u)
        else
            t ← oikea(u)
        end if
    until t = null
    return u
end procedure
```

Algoritmi 27 Proseduuri lisäyksen jälkeiseen tasapainottamiseen AVL-puussa. O(log *koko*)

```
procedure TASAPAINOTA KERRAN(Alkio t)
    d ← tasapainotila(t)
    while vanhempi(t) ≠ null or |d| < 2 do
        t ← vanhempi(t)
        d ← tasapainotila(t)
    end while
    if d = −2 then
        t ← TASAPAINOTUS(t, vasen(t), oikea, d)
    else if d = 2 then
        t ← TASAPAINOTUS(t, oikea(t), vasen, d)
    end if
    return t
end procedure
```

Proseduuri TASAPAINOTUS (**Algoritmi 28**) tekee yksi tai kaksi kiertoa parametrinaan saamansa alkion ympäristössä. Puun on tältä osin tasapainoinen.

Algoritmi 28 Proseduuri tasapainotukseen. O(1)

```
procedure TASAPAINOTUS(Alkio t, Alkio lapsi, Suunta s, Luku d )
    if tasapainotila(lapsi) · d < 0 then
        Kierto vastakkaiseen suuntaan kuin s juurena lapsi.
        Kierto suuntaan s juurena t.
    else
        Kierto suuntaan s juurena t.
    end if
    return Kierron jälkeinen alipuun juuri.
end procedure
```

Proseduuri ETSI, POISTA JA LISÄÄ (**Algoritmi 29**) muokkaa avaimen arvoa ja paikkaa hakupuussa. Aluksi poistetaan vanha alkio puusta proseduurilla ETSI JA POISTA (**Algoritmi 30**) ja lisätään uusi alkio samalle solmulle aiempaa pienemmällä prioriteetilla proseduurilla LISÄÄ JA TASAPAINOTA (**Algoritmi 25**).

Proseduuri ETSI JA POISTA (**Algoritmi 30**) poistaa avainta vastaavan alkion puusta. Aluksi etsitään avainta vastaava alkio proseduurilla HAE ALKIO (**Algoritmi 31**). Alkion tasapainotilasta riippuen etsitään korkeammasta alipuusta alkiolle edeltäjä tai seuraaja proseduurilla HAE VIEREINEN (**Algoritmi 32**). Haettu alkio korvaa paikallisesti vanhan alkion, mutta puun lyheneminen saattaa rikkoa tasapainoehdon ylempänä puus-

Algoritmi 29 Proseduuri avaimen arvon muuttamiseen poistamalla ja lisäämällä. O(log *koko*)

```
procedure ETSI, POISTA JA LISÄÄ(AVL-puu Q, Alkio u, Alkio v)
    ETSI JA POISTA(Q, avain(u))
    LISÄÄ JA TASAPAINOTA(Q, v)
end procedure
```

sa. Korjataan tasapainoehto polulla juureen proseduurilla TASAPAINOTA (**Algoritmi 35**) ja korvataan mahdollisesti vaihtunut *juuri* uudella.

Algoritmi 30 Proseduuri etsii ja poistaa prioriteettijonosta avainta vastaavan alkion. O(log *koko*)

```
procedure ETSI JA POISTA(AVL-puu (juuri), Avain k)
    u ← HAE ALKIO(juuri, k)
    if tasapainotila(u) < 0 then
        Suunta s ← oikea.
        Alkio x ← vasen(u)
        Alkio y ← oikea(u)
    else
        Suunta s ← vasen.
        Alkio x ← oikea(u)
        Alkio y ← vasen(u)
    end if
    t ← HAE VIEREINEN(x, s)
    Asetetaan alkion t suunnan s puollelle alipuu y.
    Korvataan alkio u alkiolla t.
    t ← TASAPAINOTA(t)
    if vanhempi(t) = null then
        juuri ← t
    end if
end procedure
```

Proseduuri HAE ALKIO (**Algoritmi 31**) etsii hakurakenteesta avainta vastaavan alkion. Proseduuri olettaa, että puusta etsittävä alkio on puussa.

Proseduuri HAE VIEREINEN (**Algoritmi 32**) etsii viereisen alkion annetussa suunnassa ja muodostaa puun, jossa tämä viereinen on juurena. Puuta käydään alas annettuun suuntaan kunnes suunnassa ei ole alipuuta. Kyseinen alkio on viereinen, se tulee korvata mahdollisella lapsellaan ja alkiosta tehdään puun juuri.

Proseduuri POISTA JA TASAPAINOTA (**Algoritmi 33**) poistaa tärkeimmän. Aluksi hae-

Algoritmi 31 Proseduuri hakee puusta alkiosta t alas päin alkion avaimeltaan k. $O(\log koko)$

```
procedure HAE ALKIO(Alkio t, Avain k)
    while k ≠ avain(t) do
        if k < avain(t) then
            t ← vasen(t)
        else if avain(t) < k then
            t ← oikea(t)
        else
            return t
        end if
    end while
end procedure
```

Algoritmi 32 Proseduuri alkion u viereisen alkion hakemiseen. $O(\log koko)$

```
procedure HAE VIEREINEN(Alkio u, Suunta s)
    if Alkiolla u ei ole lasta suunnassa s then
        return u
    end if
    t ← u
    while Alkiolla t on lapsi suunnassa s do
        t ← Alkion t lapsi suunnassa s.
    end while
    Korvataan alkio t lapsellaan v.                    ▷ v on ainut lapsi
    Asetetaan alkio u alkion t lapseksi suunnan s vastakkaiselle puolelle.
    Korjataan tasapainoehto polulla alkiosta v alkioon t.
    return t
end procedure
```

taan prioriteetiltaan pienin proseduurilla HAE PIENIN (**Algoritmi 34**). Alkio korvataan sen oikealla alipuulla ja mahdollisesti rikkoutunut tasapainoehto korjataan käymällä polku alkiosta juureen proseduurilla TASAPAINOTA (**Algoritmi 35**).

Algoritmi 33 Proseduuri tärkeimmän poistamiseen ja puun uudelleen tasapainottamiseen. O($\log koko$)

```
procedure POISTA JA TASAPAINOTA(AVL-puu (juuri, koko) )
    vastaus ← HAE PIENIN(juuri)
    if vastaus = juuri then
        juuri ← oikea(juuri)
    else
        v ← vanhempi(vastaus)
        Asetetaan alkion v vasemmaksi lapseksi alkio oikea(vastaus).
        t ← TASAPAINOTA(v)
        if vanhempi(t) = null then
            juuri ← t
        end if
    end if
    koko ← koko − 1
    return vastaus
end procedure
```

Proseduuri HAE PIENIN (**Algoritmi 34**) hakee avaimeltaan pienimmän alkion hakupuusta. Hakupuussa pienin sijaitsee vasemmalla.

Algoritmi 34 Proseduuri etsii pienimmän alkion. O($\log koko$)

```
procedure HAE PIENIN(Alkio t)
    while vasen(t) ≠ null do
        t ← vasen(t)
    end while
    return t
end procedure
```

Proseduuri TASAPAINOTA (**Algoritmi 35**) korjaa poiston jälkeisen puun AVL-ehdon mukaiseksi. Korjaaminen tapahtuu kutsumalla toistuvasti proseduuria TASAPAINOTA KERRAN polulla muuttuneesta alkiosta juureen. Jokainen kutsukerta etsii seuraavan AVL-ehdon rikkoutumiskohdan ja korjaa sen kierrolla.

Algoritmi 35 Proseduuri AVL-puun tasapainottamiseen poiston jälkeen. O(log *koko*)

```
procedure TASAPAINOTA(Alkio t)
    while vanhempi(t) ≠ null do
        t = TASAPAINOTA KERRAN(vanhempi(t))
    end while
    return t
end procedure
```

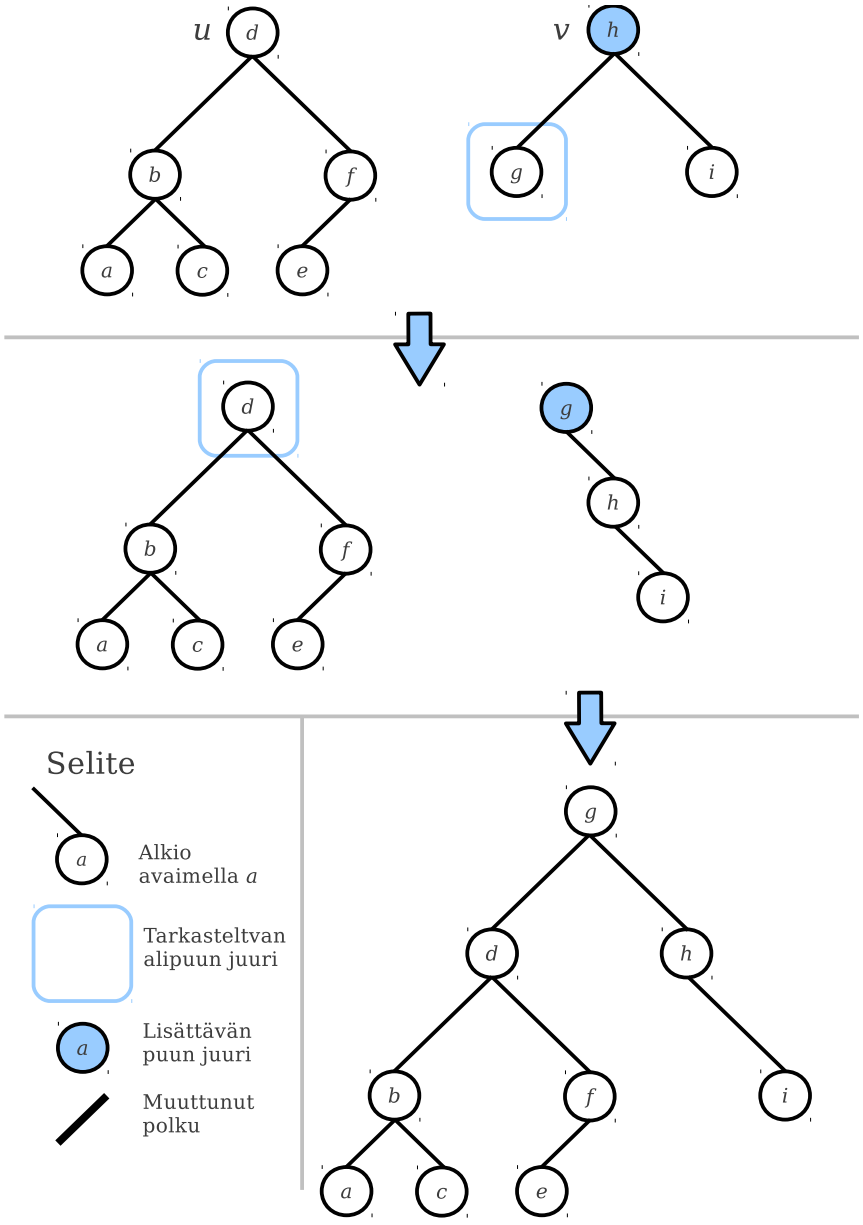
3.3.2 Implisiittisellä poistolla

Seuraava algoritmi hyödyntää Clark. A. Cranen keksimää menetelmää avaimiltaan eri väleiltä olevien AVL-puiden yhdistämiseen aikavaativuudeltaan logaritmisesti (Knuth, 1973). AVL-puuta käytetään tässä samaan tapaan kuin vasenta suosivaa kekoa: kaapaistaan puusta pois epäoleellisia alkioita ja jäljelle jääneet puut juurinaan oleellisia alkioita yhdistetään suhteellisen nopealla tavalla.

Kaksi AVL-puuta voidaan tehokkaasti yhdistää, jos ne eivät ole avainväleiltään päällekkäiset ja niiden korkeudet tunnetaan. Yhdistäminen tapahtuu korkeamman puun suhteen. Pienemmästä puusta etsitään viereinen alkio (lähin lehti) ja asetetaan sen alipuuksi pienempi puu. Tämän jälkeen etsitään korkeammasta puusta suunnilleen yhtä korkea alipuu toiseksi alipuuksi. Lopuksi yhdistelmä asetetaan korvatun puun paikalle ja tasapainoehto korjataan tästä ylöspäin kuin lisäämisen tapauksessa.

Kuvassa 15 on esimerkki AVL-puiden yhdistämisestä. Esimerkkipuiden pieni korkeusero johtaa hyvin yksinkertaiseen yhdistämiseen.

Implisiittinen poistaminen aiheuttaa hakupuussa erilaisen läpikäynnin kuin kekojen tapauksessa. Siinä, missä keossa läpikäynniksi riittää poistaa epäoleelliset alkiot keon päältä, jäisi hakupuussa vastaavalla tavalla tehtynä etsimättä pienin oleellinen alkio, joka keon tapauksessa löytyy yhdistämisen jälkeen. Pienimmän oleellisen alkion löytämiseksi täytyy edetä sisäjärjestyksessä ja kaikki ennen oleellista alkiota löytyneet epäoleelliset poistetaan puusta. Toistaalta yksin sisäjärjestyksessä eteneminen ei vastaavalla tavalla poista alkioita keskeltä jonoa, mikä on mahdollista ja pahimmassa tapauksessa jopa suotavaa, sillä epäoleelliset alkiot eivät samalla tavalla hautaudu vain



Kuva 15: Kahden AVL-puun yhdistäminen.

syvemmälle kekoon kuin vasenta suosivan puun tapauksessa, vaan päätyvät kasvattamaan tasapainoisen puun kaikkien operaatioiden aikavaativuutta yhtä lailla.

Ensimmäisessä vaiheessa etsitään pienin oleellinen alkio hakupuusta. Lähtien puun pienimmästä alkiosta edetään puuta tuhoten sisäjärjestyksessä, kunnes saavutetaan ensimmäinen alkio, joka ei ole poistettava. Poistetuiksi voisi ajatella tässä vaiheessa jonon alkupäätä.

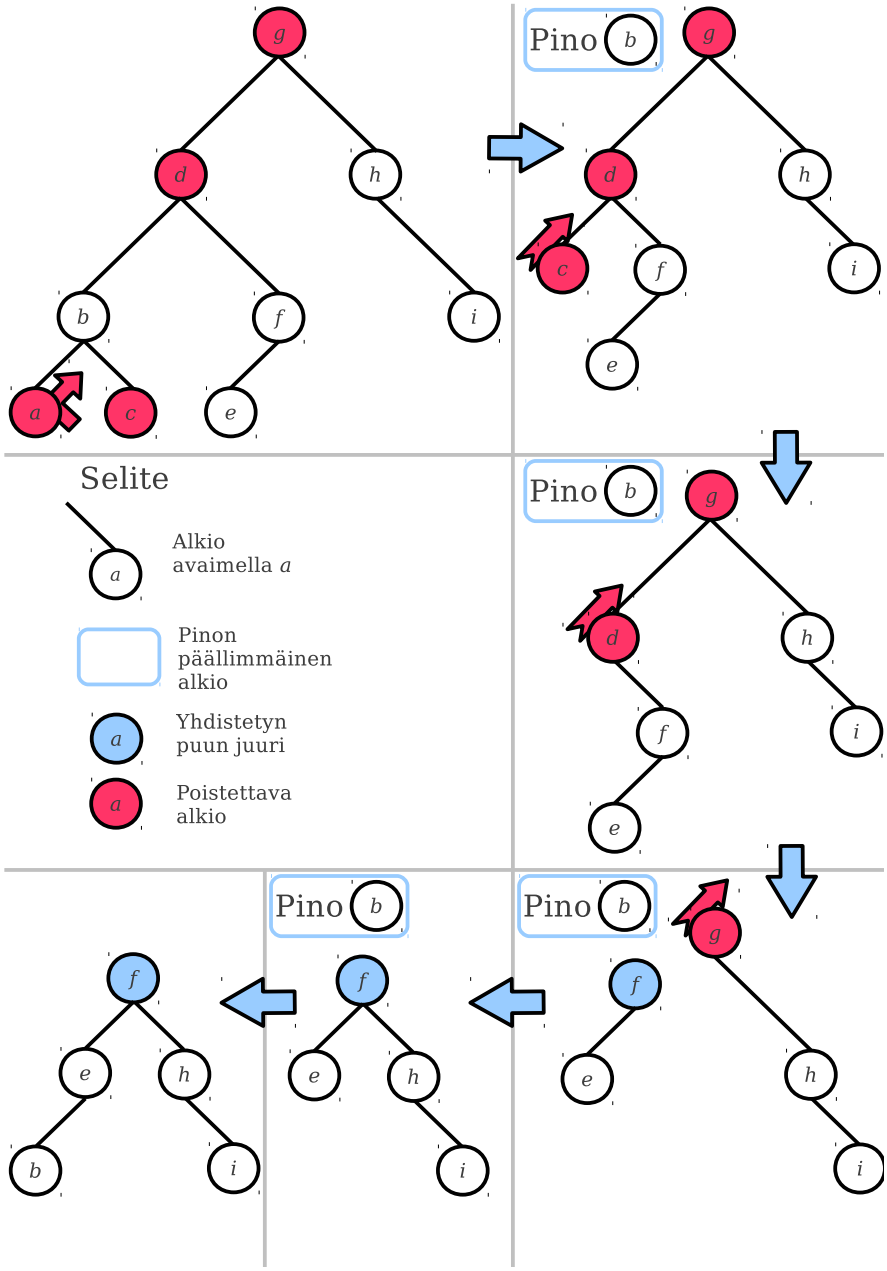
Toisessa vaiheessa noustaan muuttunutta polkua kohti juurta. Muuttuneen polun alkioilla ei ole vasempia lapsia, sillä ensimmäisessä vaiheessa ne vapautettiin. Jos alkio polulla on oleellinen, niin lisätään se pinoon myöhemmin lisättäväksi puuhun ja epäoleellinen alkio poistetaan. Alipuut oikealla käsitellään samantapaisella läpikäyntialgoritmilla kuin keon tapauksessa, joka kouraisee epäoleelliset alkiot pois puun päältä ja yhdistää löytyneet alipuut, joilla on oleelliset alkiot juurinaan. Läpikäynnin jälkeiseen puuhun lisätään pinossa odottavat alkiot.

Kuvassa 16 on esimerkki läpikäynnistä. Esimmäisessä vaiheessa (taittonuoli) puusta poistetaan epäoleellisia alkioita sisäjärjestyksessä matkalla kohti ensimmäistä oleellista alkiota. Toisessa vaiheessa (lovettu nuoli) puun vasenta laitaa noustaan ylös kohti juurta, ja oikeat alipuut tyhjennetään päällisin puolin epäoleellisista alkioista.

AVL-puusta implisiittisellä poistolla tiedetään hakupuun *juuri*, alkioiden *pino*, alkioiden lukumäärä eli *koko* ja oleellisuustieto $oleellisuus[1..n]$. Proseduurilla TASAPAINOITTAVA LÄPIKÄYNTI (**Algoritmi 36**) poistetaan keosta pienin oleellinen alkio ja muodostetaan läpikäynnin jälkeen tasapainoinen AVL-puu.

Proseduurissa OLEELLISEN ETSINTÄ (**Algoritmi 37**) käydään alkioita sisäjärjestyksessä läpi kunnes saavutetaan ensimmäinen oleellinen alkio. Läpikäynnin ohessa vapautetaan epäoleellisia alkioita puusta ja se saattaa tuhota puun tasapainoehdon.

Proseduuri OLEELLISTEN YHDISTÄMINEN (**Algoritmi 38**) läpikäy polun pienimmästä oleellisesta alkiosta vanhaan juureen ja pistää talteen oleelliset alkiot. Oikeanpuoleisista alipuista poistetaan epäoleelliset alkiot päältä proseduurilla PUHDISTA PUU.



Kuva 16: Läpikäynnit AVL-puussa.

Algoritmi 36 Proseduuri tärkeimmän alkion implisiittiseen poistamiseen AVL-puusta. $O(k \log koko)$, jossa k on epäoleellisten alkioiden lukumäärä keossa ennen ensimmäistä oleellista alkiota.

```
procedure TASAPAINOITTAVA LÄPIKÄYNTI( AVL (juuri, pino, oleellisuus, koko))
    koko ← koko − 1
    minimi ← ETSI PIENIN(juuri)
    vastaus ← min(minimi, pino)
    Merkataan oleellisuus[identiteetti(vastaus)] epäoleelliseksi.
    minimi ← OLEELLISEN ETSINTÄ(minimi, oleellisuus, koko)
    juuri ← OLEELLISTEN YHDISTÄMINEN(minimi, oleellisuus, koko)
    if vastaus = pino then
        juuri ← TALLETETTUJEN LISÄÄMINEN(alla(pino), oleellisuus)
    else
        juuri ← TALLETETTUJEN LISÄÄMINEN(pino, oleellisuus)
    end if
    return vastaus
end procedure
```

Algoritmi 37 Proseduuri pienimmän oleellisen alkion etsimiseen. $O(k)$, jossa k on epäoleellisien alkioiden määrä puussa sisäjärjestyksessä ennen ensimmäistä oleellista alkiota.

```
procedure OLEELLISEN ETSINTÄ(Alkio t, Oleellisuus b[1..n], Alkiomäärä koko)
    while t ≠ null do
        if Alkiolla t on vasen lapsi then
            s ← vasen(t)
        else if Alkio t on oleellinen then
            break
        else if Alkiolla t on oikea lapsi then
            s ← oikea(t)
            vanhempi(s) ← vanhempi(t)
            vasen(vanhempi(t)) ← s                ▷ jos alkiolla t on vanhempi
            koko ← koko − 1
        else
            s ← vanhempi(t)
            Poista vasen lapsi alkiolta s.        ▷ jos alkiolla t on vanhempi
            koko ← koko − 1
        end if
        t ← s
    end while
    return t
end procedure
```

Algoritmi 38 Proseduuri puun puhdistamiseen. $O(k \max\{1, \log(koko/(k+1))\})$, jossa k on epäoleellisten alkioiden lukumäärä, jotka ovat saavutettavissa syvyyshaulla.

```
procedure OLEELLISTEN YHDISTÄMINEN(Alkio t, juuri, Oleellisuus oleellisuus[1..n], Alkiomäärä koko)
    juuri ← null
    while t ≠ null do
        s ← vanhempi(t)
        if Alkiolla t on oikea lapsi then
            r ← oikea(t)
            Irroita r omaksi puuksi.
            PUHDISTA PUU(juuri, r, oleellisuus, koko)
        end if
        Irroita t itsenäiseksi alkioksi.
        koko ← koko − 1
        if Alkio t on oleellinen then
            LISÄÄ TALTEEN(pino, koko, oleellisuus, t)
        end if
        t ← s
    end while
    return t
end procedure
```

Proseduurissa PUHDISTA PUU (**Algoritmi 39**) käydään alkioita puun päältä epäoleelliset poistaen ja oleelliset yhdistäen *juuren* kanssa proseduurilla YHDISTÄ PUUT.

Proseduurissa YHDISTÄ PUUT (**Algoritmi 40**) yhdistetään kaksi AVL-puuta keskenään. AVL-puut eivät saa olla avainväliltään päällekkäiset, mutta ehto on voimassa, kun puut ovat AVL-puun alipuita. Tuloksena saadaan tasapainoinen AVL-puu.

Algoritmi 39 Proseduuri puun puhdistamiseen.

```
procedure PUHDISTA PUU(Alkio juuri, Alkio t, Oleellisuus oleellisuus[1..n], Alkiomäärä koko)
    Seuraava alkio s ← null
    while t ≠ null do
        if Alkio t on oleellinen then
            s ← vanhempi(t)
            vanhempi(t) ← null
            juuri ← YHDISTÄ PUUT(juuri, t)
        else if Alkiolla t on vasen lapsi then
            s ← vasen(t)
            Irroita vasen(t)
            vanhempi(vasen(t)) ← vanhempi(t)
            if Alkiolla t on oikea lapsi then
                vanhempi(oikea(t)) ← vanhempi(t)
                vanhempi(s) ← oikea(t)
            else
                vanhempi(s) ← vanhempi(t)
            end if
            Irroita t itsenäiseksi alkioksi.                    ▷ vapauta t
            koko ← koko − 1
        else if Alkiolla t on oikea lapsi then
            s ← oikea(t)
            vanhempi(s) ← vanhempi(t)
            Irroita t itsenäiseksi alkioksi.                    ▷ vapauta t
            koko ← koko − 1
        else
            s ← vanhempi(t)
            Irroita t itsenäiseksi alkioksi.                    ▷ vapauta t
            koko ← koko − 1
        end if
        t ← s
    end while
end procedure
```

Algoritmi 40 Proseduuri kahden AVL-puun yhdistämiseen. O(log *koko*)

procedure YHDISTÄ PUUT(Alkio *a*,Alkio *b*)
Nimetään puista *a* ja *b* korkeammaksi puu *k* ja toiseksi puuksi *m* ▷ vertaa *korkeuksia*
Olkoon *s* suunta, jonka puolella puu *k* on verrattuna puuhun *m* ▷ vertaa *avaimia*
Olkoon s' suunnan *s* vastakkainen suunta
$u \leftarrow$ HAE VIEREINEN(m, s)
Asetetaan alkion *u* suunnan s' puolelle alkio *m*
Lasketaan uusi *korkeus* alkiolle *u*
$t \leftarrow k$
repeat
$t \leftarrow$ alkion *t* lapsi suunnassa s'
$v \leftarrow vanhempi(t)$
until $korkeus(t) \leq korkeus(u)$
Asetetaan alkion *u* suunnan *s* puoleiseksi lapseksi alkio *t*.
Asetetaan alkion *v* suunnan s' puoleiseksi lapseksi alkio *u*.
Korjataan tasapainoehto polulta alkiosta *v* alkioon *k*.
return Tasapainotetun puun juuri
end procedure

3.4 Itseisesti kiertyvä puu

Itseisesti kiertyvä puu on Sleatorin ja Tarjanin (1985) kehittämä binäärinen hakupuu. Itseisesti kiertyminen tarkoittaa rakenteen muodon sopeuttamista hakuihin ilman alkioiden talletettuja korkeus-, mataluus- tai painoattribuutteja.

Polku juuresta alkioon on *hakupolku* (eng. *access path*). Itseisesti kiertyminen perustuu hakupolun muokkaamiseen haun yhteydessä. AVL-puun tapaan kaikissa itseisesti kiertyvän puun operaatioissa alkio paikannetaan hakemalla.

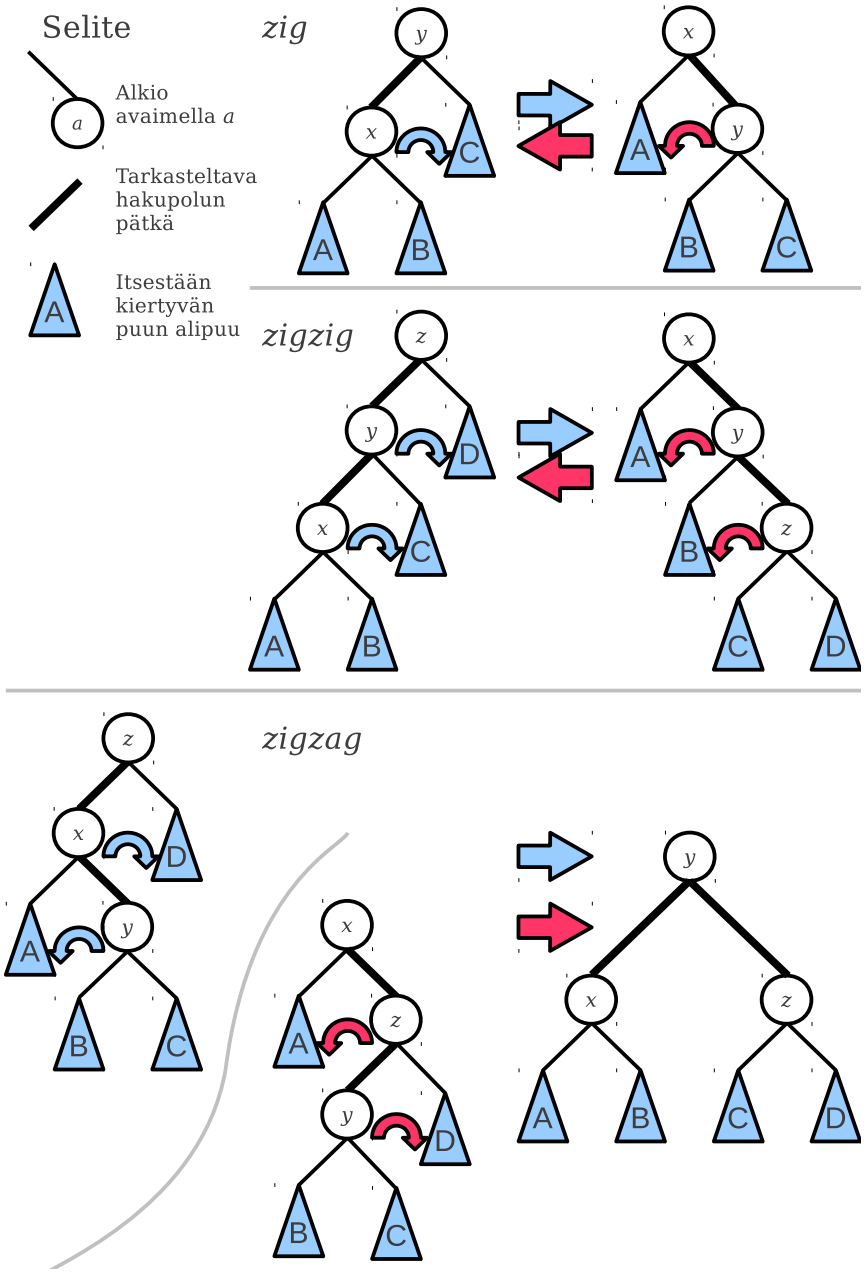
Hakupolkua tarkastellaan ensisijaisesti kahden, mutta tarvittaessa yhden yksikön mittaisissa pätkissä haetusta alkiosta juureen päin. *Tarkastelupolun* mitta on vakio ja sen *kiertyminen* tapahtuu yhdellä tai kahdella kierrolla, joiden aikavaativuus on $O(1)$. Hakupolun kiertymisen aikavaativuus on täten verrannollinen hakupolun pituuteen d. Hakupolun pituus d ei välttämättä ole logaritminen alkioiden määrän *koko* suhteen, mutta keskimäärin jokainen alkio on saavutettavissa logaritmisella määrällä kiertoja. Kiertymisen seurauksena on haetun alkion nouseminen tarkastelupolun päähän alipuun juurialkioksi rikkomatta hakuehtoa.

Tarkastellaan seuraavaksi tilanteet, miten eri tarkastelupolun pätkissä kiertyminen tapahtuu. Kuvassa 17 on yksi yhden mittaisen tarkastelupolun pätkän tarkastelu ja kaksi kahden mittaisen pätkän tapausta. Tapauksissa *zig* ja *zigzig* siniset kaarinuolet näyttävät kierrot, jotka nostavat alkion x juureksi. Punaiset kaarinuolet näyttävät kierrot tapauksiin, jotka päätyvät vastakkaiseen lopputulokseen. Tapauksessa *zigzag* sekä siniset että punaiset kaarinuolet päätyvät samaan lopputulokseen – alkio y nousee juureksi.

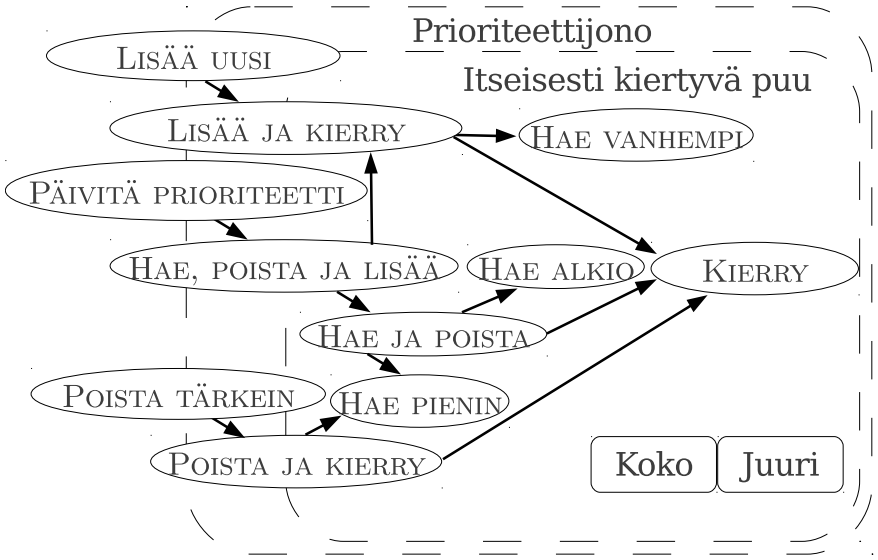
3.4.1 Ahkerasti

Rakenteesta tiedetään puun *juuri*-alkio ja alkioiden lukumäärä *koko*. Kuten AVL-puun tapauksessa, kyseessä on hakupuu, joten on tarvetta käyttää avaimia yksilöimään alkiot.

Kuvassa 18 on prioriteettijonon rajapinta, jonka itseisesti kiertyvä puu toteuttaa. Ku-



Kuva 17: Itseisesti kiertyvän puun kiertyminen eri tilanteissa *zig*, *zigzig* ja *zigzag*.



Kuva 18: Itseisesti kiertyvän puun rakenne.

vasta nähdään, että erilaisia hakuja on AVL-puun tapaan paljon, mutta AVL-puusta poiketen tasapainottavia proseduureja ei ole. Lisäksi nähdään, että proseduuri KIERRY on rakenteessa keskeinen. Operaatio LISÄÄ UUSI tehdään proseduurilla LISÄÄ JA KIERRY, operaatio PÄIVITÄ PRIORITEETTI tehdään proseduurilla HAE, POISTA JA LISÄÄ ja operaatio POISTA TÄRKEIN tehdään proseduurilla POISTA JA KIERRY.

Proseduuri LISÄÄ JA KIERRY (**Algoritmi 41**) hakee lisättävälle alkiolle vanhemman proseduurilla HAE VANHEMPI (**Algoritmi 26, sivu 44**), lisää uuden alkion sen lapseksi ja kierryttää lisätyn alkion uudeksi juureksi proseduurilla KIERRY (**Algoritmi 42**).

Proseduurilla KIERRY (**Algoritmi 42**) muokataan hakupolkua kierroilla, jotta alkion hakuaika lyhenisi. Kiertyminen saa haetun alkion nousemaan puun juureksi.

Proseduuri HAE, POISTA JA LISÄÄ (**Algoritmi 43**) lisää uuden alkion, kierryttää sen juureksi, etsii vanhan alkion, poistaa sen puusta ja kierryttää poistetun alkion isän puun juureksi.

Proseduuri POISTA JA KIERRY (**Algoritmi 44**) hakee puusta minimialkion proseduurilla ETSI PIENIN (**Algoritmi 34, sivu 48**), poistaa minimin ja kierryttää poistetun alkion

Algoritmi 41 Proseduuri uuden alkion lisäämiseen itseisesti kiertyvään puuhun. O(d), jossa d on hakupolun pituus.

```
procedure LISÄÄ JA KIERRY(Itseisesti kiertyvä puu (juuri, koko), Alkio v)
    koko ← koko + 1
    if juuri = null then
        juuri ← v
    else
        t ← HAE VANHEMPI(juuri, avain(v))
        if avain(v) < avain(t) then
            Asetetaan alkio v alkion t vasemmaksi lapseksi.
        else
            Asetetaan alkio v alkion t oikeaksi lapseksi.
        end if
        juuri ← KIERRY(t)
    end if
end procedure
```

Algoritmi 42 Proseduuri alkion t kierryttämiseen puun juureksi. O(d), jossa d on hakupolun pituus.

```
procedure KIERRY(Alkio t)
    while onko alkiolla t vanhempaa do
        s ← puoli, jolla t on vanhempaansa.
        if onko vanhempi vanhempaansa puolella s then
            t ← ZIGZIG(s, t)
        else if onko vanhemmalla vanhempaa then
            t ← ZIGZAG(s, t)
        else
            t ← ZIG(s, t)
        end if
    end while
    return t
end procedure
```

Algoritmi 43 Proseduuri alkion päivittämiseen itseisesti kiertyvässä puussa. O(d), jossa d on hakupolun pituus.

```
procedure HAE, POISTA JA LISÄÄ(Itseisesti kiertyvä puu (juuri, koko), Alkio u,
Alkio v)
    LISÄÄ JA KIERRY(v)                                          ▷ v on juuri
    t ← ETSI ALKIO(avain(u))
    s ← vanhempi(t)
    vanhempi(oikea(t)) ← null
    if alkiolla t on oikea alipuu then
        t' ← KIERRY(ETSI PIENIN(oikea(t)))
        Asetetaan alkion t' vasemmaksi lapseksi alkio vasen(t).
    else
        t' ← vasen(t)
    end if
    Asetetaan alkion s lapsen t paikalle alkio t'.
    juuri ← KIERRY(s)
end procedure
```

vanhemman uudeksi juureksi proseduurilla KIERRY (**Algoritmi 44**).

Algoritmi 44 Proseduuri tärkeimmän alkion poistamiseen itseisesti kiertyvästä puusta. O(d), jossa d on hakupolun pituus.

```
procedure POISTA JA KIERRY(Itseisesti kiertyvä puu (juuri, koko))
    vastaus ← ETSI PIENIN(juuri)
    if vastaus = juuri then
        juuri ← oikea(vastaus)
    else
        t ← vanhempi(vastaus)
        Asetetaan alkio oikea(vastaus) alkion t vasemmaksi lapseksi.
        juuri ← KIERRY(t)
    end if
    koko ← koko − 1
    return vastaus
end procedure
```

3.4.2 Implisiittisellä poistolla

Seuraava algoritmi hyödyntää Sleatorin ja Tarjanin (1985) kehittämää menetelmää itseisesti kiertyvien puiden yhdistämiseen. Yhdistettävien puiden avainvälit eivät saa olla päällekkäiset. Olkoon pienempien avaimien puu u ja suurempien v. Vain toista puuta riittää muokata. Puusta u etsitään maksimi ja kierretään se juureksi. Juuren oikeak-

si lapseksi asetetaan puu v. Vastaavasti voidaan muokattavaksi puuksi valita v ja etsiä siitä minimi, joka kierretään juureksi. Juuren vasemmaksi lapseksi asetetaan tällöin u.

Kuvassa 15 sivulla 50 esitetty esimerkki AVL-puiden yhdistämisestä noudattaa samoja välivaiheita kuin vastaavien itseisesti kiertyvien puiden. Kuvassa minimi puusta v kierretään puun juureksi ja puu u liitetään sen vasemmaksi alipuuksi.

Implisiittinen poistaminen tehdään AVL-puun tapaan kaksivaiheisella algoritmilla. Ensimmäisessä vaiheessa haetaan puun minimialkio sisäjärjestyksessä ja samalla poistetaan puusta vastaan tulevat epäoleelliset alkiot. Löydetty alkio kierretään puun juureksi. Toisessa vaiheessa käydään läpi puun oikeanpuoleisia alkioita, joista epäoleelliset poistetaan ja oleelliset pistetään talteen pinoon. Näiden alkioiden vasemmista alipuista poistetaan epäoleelliset alkiot siten, että juureltaan oleelliset alipuut yhdistetään järjestyksessä keskenään.

Tärkeimmän alkion poistaminen tehdään proseduurilla KIERRYTTÄVÄ LÄPIKÄYNTI (**Algoritmi 45**). Algoritmi jakaa saman idean ja lähes saman toteutuksen kuin AVL-puun tapauksessa (**Algoritmi 36**). Poikeuksena on minimialkion kierryttäminen juureksi ja proseduurin OLEELLISTEN YHDISTÄMINEN tekeminen juuresta lehteen päin. Muut proseduurit toimivat AVL-puun tapaan.

Proseduurissa OLEELLISTEN YHDISTÄMINEN (**Algoritmi 46**) laskeudutaan juuresta pitkin oikeanpuoleisia lapsia. Jos alkiolla on vasen lapsi, sen epäoleelliset alkiot käydään syvyyshaulla poistamassa ja juureltaan oleelliset alipuut yhdistetään juureen.

3.5 Fibonacci-keko

Fibonacci-keko on Fredmanin ja Tarjanin (1987) kehittämä parannus binomikeolle tarkoituksenaan pienentää aikavaativuuksia mm. Dijkstran algoritmin käyttämän PÄIVITÄ PRIORITEETTI -proseduurin osalta. Myöhemmin Fibonacci-kekoakin on paranneltu paremmilla kertoimilla mm. Driscoll, Gabow, Shrairman ja Tarjan (1988) sekä Brodal, Lagogiannis ja Tarjan (2012). Fibonacci-keon toiminnan ymmärtämiseksi tutustutaan

Algoritmi 45 Proseduuri tärkeimmän alkion implisiittiseen poistamiseen itseisesti kiertyvästä puusta. O($k \cdot koko$), jossa k on epäoleellisten alkioiden lukumäärä keossa.

```
procedure KIERRYTTÄVÄ LÄPIKÄYNTI(Itseisesti kiertyvä puu (juuri, pino, oleellisuus[1..n], koko))
    koko ← koko − 1
    minimi ← ETSI PIENIN(juuri)
    vastaus ← min(minimi, pino)
    Merkataan oleellisuus[identiteetti(vastaus)] epäoleelliseksi.
    minimi ← OLEELLISEN ETSINTÄ(minimi, oleellisuus, koko)
    juuri ← KIERRY(minimi)
    juuri ← OLEELLISTEN YHDISTÄMINEN(juuri, oleellisuus, koko)
    if vastaus = pino then
        juuri ← TALLETETTUJEN LISÄÄMINEN(alla(pino), oleellisuus)
    else
        juuri ← TALLETETTUJEN LISÄÄMINEN(pino, oleellisuus)
    end if
    return vastaus
end procedure
```

Algoritmi 46 Proseduuri puun puhdistamiseen. O($k \cdot koko$), jossa k on syvyyshaulla saavutettavien epäoleellisten alkioiden lukumäärä.

```
procedure OLEELLISTEN YHDISTÄMINEN(Alkio juuri, Oleellisuus oleellisuus[1..n], Alkiomäärä koko)
    t ← juuri
    juuri ← null
    while t ≠ null do
        s ← oikea(t)
        if Alkiolla t on vasen lapsi then
            l ← vasen(t)
            Irroita l omaksi puuksi.
            PUHDISTA PUU(juuri, l, oleellisuus, koko)
        end if
        Irroita t itsenäiseksi alkioksi.
        koko ← koko − 1
        if Alkio t on oleellinen then
            LISÄÄ TALTEEN(pino, koko, oleellisuus, t)
        end if
        t ← s
    end while
    return t
end procedure
```

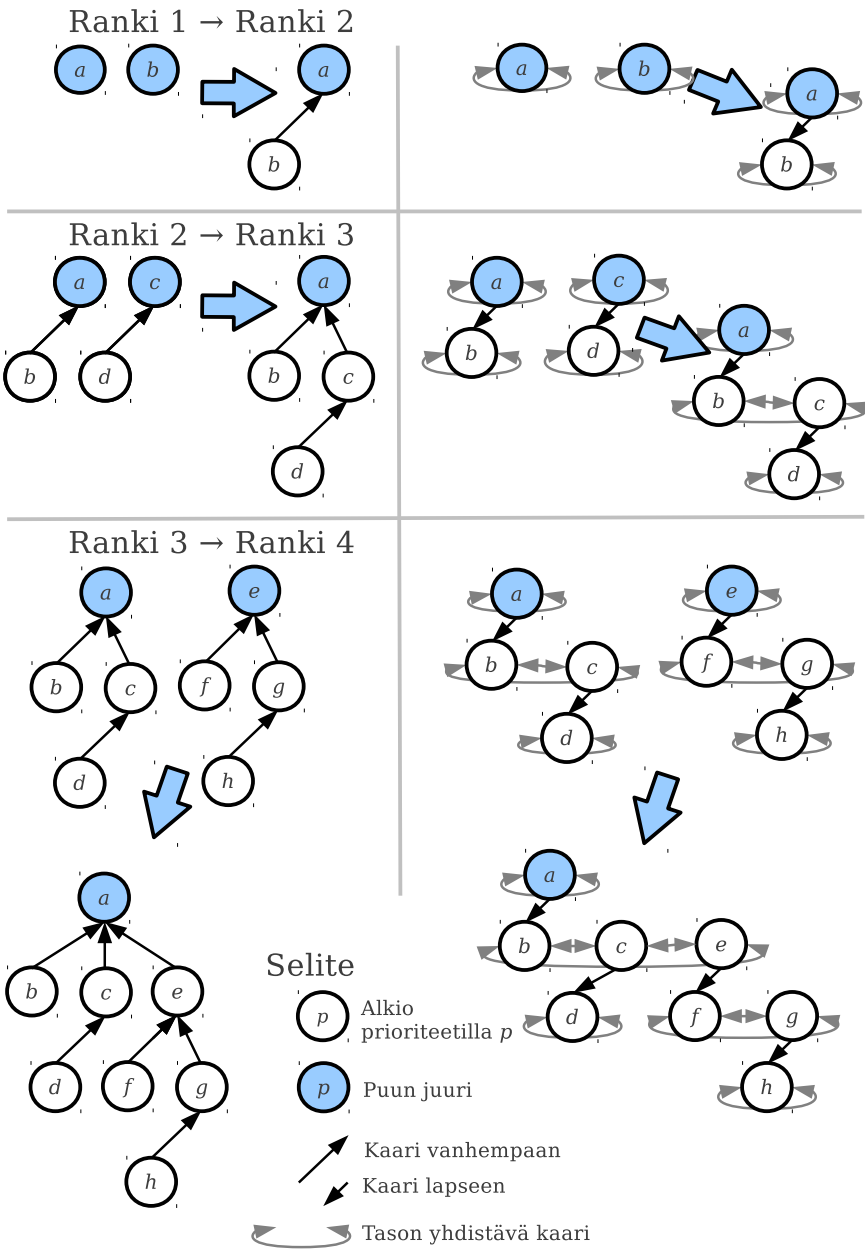
aluksi Jean Vuilleminin (1978) kehittämään *Binomikekoon* (eng. *binomial heap*).

Pienin binomipuu on yhden alkion kokoinen. Binomipuu määritellään rekursiivisesti kahden pienemmän, mutta yhtäsuuren binomipuun *linkitykseksi*. Puiden yhtäsuuruudella binomipuun tapauksessa tarkoitetaan puita, joilla on sama *ranki*. Siksi alkioille annetaan vastaavanlainen ominaisuus ranki, joka on $lapsiluku + 1$ ja lehden ranki on 1. Puun ranki kasvaa linkityksessä yhdellä. Binomipuu saa nimensä alkioiden määrästä tasoittain, joka noudattaa binomijakaumaa.

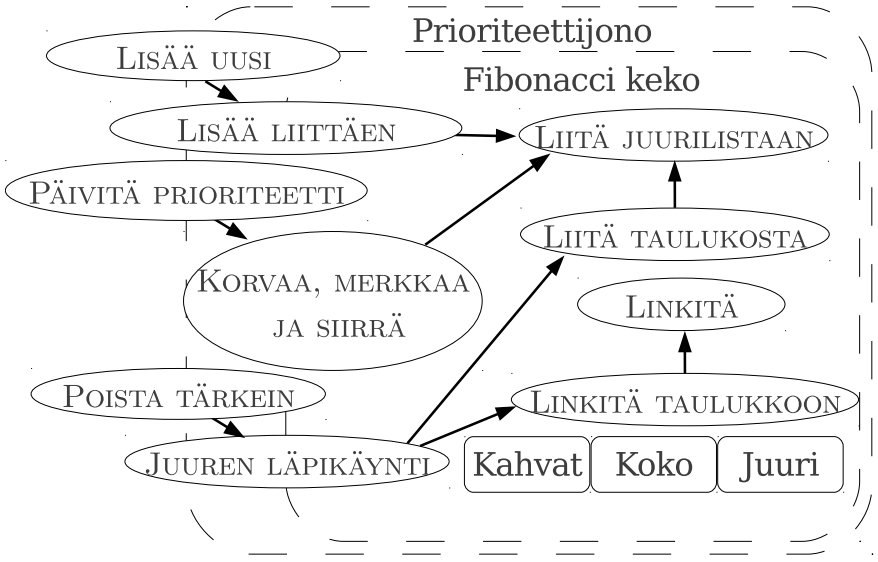
Fredmanin ja Tarjanin (1987) mukaan binomikeko muodostuu *juurilistasta*, joka on kahteen suuntaan linkitetty lista prioriteettiehtoisia erisuuria binomipuita. Binomikeossa ei siis ole kahta saman suuruista binomipuuta, sillä ne olisi tullut linkittää yhdeksi suuremmaksi binomipuuksi.

Kuvassa 19 on binomipuita rankiltaan yhdestä neljään. Kuvan vasemmassa reunassa on kuvattu kaaret alkioiden vanhempiin, kun oikeassa reunassa näkyvät kaaret lapsiin ja lapsien kesken kahteen suuntaan linkitettynä listana. Rankin arvoltaan yksi binomipuiden linkittäminen tapahtuu yksinkertaisesti valitsemalla prioriteetiltaan tärkeämpi juureksi ja toinen sen lapseksi. Vastaavasti linkittäminen tapahtuu rankin arvolla kaksi ja korkeammilla rankin arvoilla.

Fibonacci-keossa käytetään laiskaa lisäämistä, jossa juurilistaan lisätään uusi alkio. Poiston yhteydessä puut yhdistetään vastaavan rankin puun kanssa, vasenta suosivan keon tapaan apurakenteessa. Binomikeosta poiketen yhdistettävät puut eivät ole aina binomipuita, koska prioriteetin päivittämisen yhteydessä puuta käsitellään niistä alipuita poistaen. Puu, jonka ranki pysyy samana, vaikka sitä on poistettu maksimaalisesti alkioita, sisältää Fibonaccin luvun F_{ranki} verran alkioita. Tästä saadaan rakenteen nimi Fibonacci-keko ja apurakenteen koolle yläraja $\log_\phi koko$, jossa $\phi = \frac{\sqrt{5}+1}{2}$ on sama kuin kahden peräkkäisen Fibonaccin luvun suhteen raja-arvo.



Kuva 19: Binomisien puiden linkityksiä rankilta 1 rankiin 4.



Kuva 20: Fibonacci-keon rakenne.

3.5.1 Ahkerasti

Fibonacci-keon toteuttamiseksi rakenteesta tiedetään pienin alkio *juuri*, $kahvat[1..n]$ ja apurakenteena käytetään taulukkoa $A[1..\log_\phi n]$, jossa $\phi = \frac{\sqrt{5}+1}{2}$ ja n on solmujen lukumäärä verkossa. Jokaisella alkiolla on muuttujat $ranki = lapsiluku + 1$ ja *merkki* bitti, joka kertoo onko puulta poistettu alipuu.

Kuvassa 20 on prioriteettijonon rajapinta, jonka Fibonacci-keko toteuttaa. Kuvasta nähdään, että juurilistaa käsitellään lisäämisessä ja päivittämisessä ilman linkittämistä. Vasta tärkeimmän poistamisessa alkiot linkitetään käyttäen apuna taulukkoa ja taulukosta muodostetaan uusi juurilista. Prioriteettijonon proseduurin LISÄÄ UUSI toteuttaa proseduuri LISÄÄ LIITTÄEN, proseduurin PÄIVITÄ PRIORITEETTI toteuttaa proseduuri KORVAA, MERKKAA JA SIIRRÄ ja proseduurin POISTA TÄRKEIN toteuttaa proseduuri JUUREN LÄPIKÄYNTI.

Proseduuri LISÄÄ LIITTÄEN (**Algoritmi 47**) liittää uuden alkion prioriteettijonoon. Alkio liitetään juurilistaan yleisemmällä proseduurilla LIITÄ JUURILISTAAN.

Proseduuri LIITÄ JUURILISTAAN (**Algoritmi 48**) liittää alkion juurilistaan. Jos juuri-

Algoritmi 47 Proseduuri uuden alkion lisäämiseen. O(1)

```
procedure LISÄÄ LIITTÄEN(Fibonacci-keko (juuri, kahvat[1..n], koko), Alkio v)
    koko ← koko + 1
    kahvat[identiteetti(v)] ← v
    LIITÄ JUURILISTAAN(juuri, v)
end procedure
```

taso on tyhjä, niin alkiosta tulee *juuri*. Jos uuden alkion prioriteetti on pienempi kuin *juuren*, niin siitä tulee uusi *juuri*.

Algoritmi 48 Proseduuri alkion liittämiseksi juurilistaan. O(1)

```
procedure LIITÄ JUURILISTAAN(Alkio juuri, Alkio v)
    vanhempi(v) ← null                ▷ Juurilistassa ei alkiolla ole vanhempia.
    if juuri = null then
        juuri ← v
    else
        Liitetään alkio v juurilistaan.
        if prioriteetti(v) < prioriteetti(juuri) then juuri ← v
        end if
    end if
end procedure
```

Proseduuri KORVAA, MERKKAA JA SIIRRÄ (**Algoritmi 49**) päivittää alkion prioriteetin arvon Fibonacci-keossa. Käytetään kahvaa alkion löytämiseen ja asetetaan sille uusi prioriteetti. Jos alkion vanhemman prioriteetti on suurempi kuin alkion, niin toistetaan seuraavaa. Alkio irroitetaan vanhemmastaan, poistetaan mahdollinen merkki alkiosta, liitetään alkio osaksi juurilistaa proseduurilla LISÄÄ JUURILISTAAN (**Algoritmi 48**), pienennetään vanhemman rankia ja merkkaamaton vanhempi merkataan. Mikäli vanhempi oli jo ennestään merkattu toistetaan edellistä vaihetta kunnes joku esivanhempi saadaan merkattua tai saavutetaan juuritaso. Jos päivitetyn alkion prioriteetti osoittautuu keon pienimmäksi, siitä tulee uusi *juuri*. Aikavaativuus on tasoitettu O(1).

Proseduuri JUUREN LÄPIKÄYNTI (**Algoritmi 50**) poistaa pienimmän ja etsii uuden pienimmän *juureksi* linkittämällä juurilistan alkiot. Irroitetaan alkiot juurilistasta ja linkitetään ne osaksi apurakennetta A proseduurilla LINKITÄ TAULUKKOON (**Algoritmi 51**). Kun kaikki juurilistan alkiot on läpikäyty, niin käydään läpi apurakenne A proseduurilla LIITÄ TAULUKOSTA (**Algoritmi 53**).

Algoritmi 49 Proseduuri alkion prioriteetin päivittämiseen Fibonacci-keossa. Aikavaativuus tasoitettu $O(1)$.

```
procedure KORVAA, MERKKAA JA SIIRRÄ(Fibonacci-keko (juuri, kahvat[1..n], koko), Alkio u, Alkio v)
    t ← kahvat[identiteetti(v)]
    prioriteetti(t) ← prioriteetti(u)
    if alkiolla t on vanhempi and prioriteetti(t) < prioriteetti(vanhempi(t)) then
        repeat
            v ← vanhempi(t)
            Poistetaan alkio t vanhempansa lapsilistasta.
            Poista merkki alkiosta t.
            LIITÄ JUURILISTAAN(juuri, t)
            ranki(v) ← ranki(v) − 1
            t ← v
        until alkiota v ei ole merkattu or alkiolla v ei ole vanhempaa
        if alkiolla v on vanhempi then
            Merkataan alkio v.
        end if
    end if
    if prioriteetti(t) < prioriteetti(juuri) then
        juuri ← t
    end if
end procedure
```

Algoritmi 50 Proseduuri tärkeimmän poistamiseksi Fibonacci-keosta. Aikavaativuus tasoitettu $O(\log koko)$.

```
procedure JUUREN LÄPIKÄYNTI(Fibonacci-keko (juuri, kahvat[1..n], koko))
    vastaus ← juuri
    Olkoon apurakenne A[1.. log_φ n].
    while juurilistassa on jäseniä do
        t ← poistettu alkio juurilistasta.
        LINKITÄ TAULUKKOON(t, A)
    end while
    LIITÄ TAULUKOSTA(A)
    kahvat[identiteetti(vastaus)] ← null
    koko ← koko − 1
    return vastaus
end procedure
```

Proseduuri LINKITÄ TAULUKKOON (**Algoritmi 51**) liittää alkion osaksi apurakennetta A. Linkitetään apurankenteesta mahdollisesti löytynyt saman rankin alkio yhteen proseduurilla LINKITÄ (**Algoritmi 52**). Jos taulukosta löytyy tyhjä indeksi, niin asetetaan muodostunut puu siihen.

Algoritmi 51 Proseduuri linkittämään saman rankiset alipuut yhteen, missä ranki merkitsee paikkaa aputaulukossa. $O(\log_\phi koko)$.

```
procedure LINKITÄ TAULUKKOON(Alkio t, Apurakenne A[1..log_φ n])
    r ← ranki(t)
    while A[r] ≠ null do
        if prioriteetti(t) < prioriteetti(A[r]) then
            t ← LINKITÄ(t, A[r])
        else
            t ← LINKITÄ(A[r], t)
        end if
        A[r] ← null
        r ← ranki(t)
    end while
    A[r] ← t
end procedure
```

Proseduuri LINKITÄ (**Algoritmi 52**) linkittää alkiot toisiinsa. Pienempi tulee juureksi, sen ranki kasvaa yhdellä vastaamaan suurempaa puuta ja suurempi lisätään sen lapsilistaan.

Algoritmi 52 Proseduuri linkittämään kaksi saman rankin puuta yhteen. $O(1)$.

```
procedure LINKITÄ(Alkio x, Alkio y)
    Lisää alkion x lapsilistaan alkio y.
    ranki(x) ← ranki(x) + 1
    Poista merkki alkiosta y.
    return x
end procedure
```

Proseduuri LIITÄ TAULUKOSTA (**Algoritmi 53**) kerää puut juurilistaan käymällä läpi apurakenteen A. Liitetään alkio $A[i]$ proseduurilla LIITÄ JUURILISTAAN (**Algoritmi 48**) ja vapautetaan aputaulukon solu. Tämä luo uuden juurilistan ja tyhjentää aputaulukon.

Algoritmi 53 Proseduuri uuden juurilistan luomiseen liittämällä alkiot apurakenteesta. $O(\log_\phi koko)$.

```
procedure LIITÄ TAULUKOSTA(Apurakenne A[1..log n])
    for i ∈ [1, log_φ koko] do
        LIITÄ JUURILISTAAN(A[i])
        A[i] ← null
    end for
end procedure
```

3.5.2 Implisiittisellä poistolla

Fredman ja Tarjan (1987) eivät tyydy Fibonacci-keon kahvalliseen toteutukseen, vaan artikkeli sisältää myös kolme implisiittisen poiston toteutusta Fibonacci-keon yhteydessä. Tässä käytetään niistä helpointa.

Tuttuun tapaan rakenteesta tiedetään minimialkio *juuri*, uusien alkioiden *pino*, $oleellisuus[1..n]$ ja apurakenne $A[1..\log_\phi m]$. Alkion ei tässä tapauksessa tarvitse tietää vanhempaansa, mikä nopeuttaa juurilistan ja lapsilistojen liittämistä toisiinsa. Tärkeimmän poistamiseen käytetään proseduuria PINNAN LÄPIKÄYNTI.

Proseduurin PINNAN LÄPIKÄYNTI (**Algoritmi 54**) ja proseduurin JUUREN LÄPIKÄYNTI (**Algoritmi 50**) erot ovat hyvin pienet. Aikaisemmassa riitti juurilistan läpikäynti, nyt joudutaan epäoleellisen alkion tapauksessa liittämään sen lapsilista osaksi juurilistaa, mikä saattaa pidentää läpikäyntiä. Talletuksen purku tehdään taulukkoon A poistamalla alkio pinosta ja linkittämällä se paikoilleen proseduurilla LINKITYS (**Algoritmi 52**). Edellisestä rakenteesta tuttuun tapaan uusi *juuri* muodostetaan taulukon A puiden juuret yhteen liittämällä proseduurilla LIITÄ TAULUKOSTA (**Algoritmi 53**).

Algoritmi 54 Proseduuri tärkeimmän poistamiseksi Fibonacci-keosta implisiittisellä poistolla. Aikavaativuus tasoitettu $O(k \max\{1, \log(koko/(k+1))\})$, jossa k on poistettujen alkioiden määrä.

```
procedure PINNAN LÄPIKÄYNTI(Prioriteettijono (juuri, b[1..n], koko))
    koko ← koko − 1
    vastaus ← min(juuri, pino)
    Merkataan oleellisuus[identiteetti(vastaus)] epäoleelliseksi.
    Olkoon apurakenne A[1.. log_φ m].
    while Juurilistassa on jäseniä do
        t ← poistettu alkio juurilistasta.
        if Alkio t on oleellinen then
            LINKITYS(t, A)
        else
            Liitetään lapsi(t)-lista juurilistaan.
            koko ← koko − 1
        end if
    end while
    if vastaus = pino then
        TALLETUKSEN PURKU(alla(pino), oleellisuus, A)
    else
        TALLETUKSEN PURKU(pino, oleellisuus, A)
    end if
    LIITÄ TAULUKOSTA(A)
    H[identiteetti(vastaus)] ← null
    return vastaus
end procedure
```

4 Koeasetelma

Luvussa esitellään Dijkstran algoritmin (**Algoritmi 3, sivu 9**) testaamisen tarvittavan verkkoaineiston generointitavat ja kokeissa käytettävä ympäristö. Tutkielmassa rajoitutaan *täydellisiin* ja *satunnaisiin* verkkoihin, joissa on korkeintaan yksi kaari kahden solmun välissä.

Verkkoja voidaan luokitella topologisten ominaisuuksien ja kaarien painofunktion mukaan. Verkon topologia määrittelee solmuja yhdistävien kaarten määrän ja paikat painofunktion määritellessä kaarten painot. Dijkstran algoritmi olettaa, ettei kaarten painot ole negatiivisia lukuja.

4.1 Topologiat

Suurinta kaarien määrää tutkielmassa edustaa täydellinen verkko (eng. *complete graph*), missä kaikista solmuista on kaari jokaiseen solmuun. Tutkielman aineistossa myös täydellisistä verkoista kaarien määrää rajoitetaan prosenttiosuuteen p, koska aidosti täydellisiä verkko on harvassa ongelmassa.

Tilansäästämiseksi täydellisen verkon yksi solmu asetetaan *aloituspisteeksi* ja verkko generoidaan *täydellisen kaltaiseksi aloituspisteen näkökulmasta*. Täydellisen kaltainen aloituspisteen näkökulmasta tarkoittaa, että lähtien aloituspisteestä Dijkstran algoritmi kutsuu vastaavia operaatioita saman määrän ja samassa järjestyksessä kuin täydellisen verkon tapauksessa. Jostain muusta solmusta kuin aloituspisteestä aloittaminen ei tuota samaa lopputulosta. Täydellisen kaltaisien verkojen nimet alkavat tutkielmassa kirjaimella C.

Verkot on esitetty vieruslistana, mutta seuraavassa käytetään matriisiesitystä havainnollistamaan millaista ehtoa täydellisen kaltaisien verkkojen generointiin käytetään. Täydellisen kaltainen verkko on matriisiesityksenä yläkolmiomatriisi, jossa solmujen indeksit vastaavat rivi- ja sarakenumeroita. Kaarten määrän vähentäminen tehdään poistamalla kaaret, jotka vastaavat kolmiota matriisiesityksen oikeassa yläkulmassa.

Poistettavan kolmion pinta-ala vastaa $(n^2-n)(1-p)/2$ kaarta ja jäljelle jäävien kaarien lukumäärä on $m = p(n^2-n)/2$. Täydellisen kaltainen verkko säilyttää yhtenäisyyden, jos kaarten lukumäärä $n \leq m$.

Satunnaisverkko tarkoittaa topologialtaan määrittelemätöntä verkoa. Satunnaisverkkojen nimet alkavat tutkielmassa kirjaimella *R*. Tutkielmassa satunnaisverkot generoidaan arpomalla erisuuria kokonaislukuja r väliltä $[1, n(n-1)]$. Olkoon

$$a = \left\lceil \frac{r}{n-1} \right\rceil \tag{1}$$

$$b = r \mod (n-1) \tag{2}$$

$$c = \begin{cases} b+1 & \text{kun } a \leq b \\ b & \text{muuten} \end{cases} \tag{3}$$

jossa n on solmujen määrä. Tällöin (a,c) on luvun r yksilöimä kaari. Erisuuret kokonaisluvut varmistettiin täysisyklisellä pseudosatunnaislukugeneraattorilla annetulla välillä. Pseudosatunnaislukugeneraattori parametrisoitiin täysisyklisyyden takaavilla parametreilla (Hull ja Dobell, 1962). Kaarten määrän rajoittaminen satunnaisverkoissa saattaa aiheuttaa epäyhtenäisyyttä verkossa.

4.2 Painofunktiot

Seuraavaksi esitellään kolme painofunktiota: pahimpaan tapaukseen räätälöity painofunktio w_f (*fixed,F*), sen satunnainen permutaatio w_p (*permutation,P*) ja satunnainen painofunktio w_r (*random,R*).

Painofunktio $w_f(i,j)$ varmistaa, että jokainen verkon kaari täydellisessä verkossa käsitellään Dijkstran algoritmilla, eli jokainen löytynyt uusi kaari lyhentää polkua sol-

muun. Käsittelyjärjestysriippumaton painofunktio $w_f(i,j)$ on

$$w_f(i,j) = \begin{cases} 2(n-i)-1 & \text{kun } i+1 < j \\ 2(n-j)-1 & \text{kun } j < i-1 \\ 1 & \text{muuten} \end{cases} \quad (4)$$

jossa n on solmujen lukumäärä, i on lähtöpiste ja j on päätepiste sekä $i, j \in [1,n]$. Painojen joukko on parittomat luvut väliltä $[1, 2n-3]$.

Painofunktio $w_r(i,j)$ kuvaa satunnaispainofunktiota, joka palauttaa arvoista i ja j riippumattoman tasajakautuneen parittoman luvun väliltä $[1, 2n-3]$. Kaarten painot ovat samoja lukuja kuin painofunktion $w_f(i,j)$ tapauksessa, mutta niiden jakauma on erilainen.

Painofunktion $w_p(i,j)$ jakauma on samanlainen kuin painofunktion $w_f(i,j)$, mutta painot ovat i ja j arvosta riippumattomat parittomat luvut väliltä $[1, 2n-3]$. Painofunktio on satunnainen permutaatio pahimman tapauksen kaarifunktiosta $w_f(i,j)$.

4.3 Aineisto

Rakenteiden testaamiseen generoidaan kuusi kategoriaa verkkoja, joista käytetään nimiä *CF*, *CP*, *CR*, *RF*, *RP* ja *RR*, ensimmäisen kirjaimen viitatessa topologiaan ja toisen painofunktioon. Solmujen lukumäärät ovat tuhannet välillä $[1000, 8000]$ ja kaarten lukumäärää rajoitetaan prosenttiosuudella p mahdollisesta kaarten määrästä, prosenttiosuuksilla 25% ja 95%. Verkkoja generoidaan jokaiseen kategoriaan $N = 61$ kappaletta.

4.4 Suoritusympäristö

Ohjelma on ohjelmoitu Java-kielellä versiolle 1.6 ja Javan automaattisesta muistin hallinnasta riippuen operaatioon käytetty aika saattaa sisältää satunnaisesti roskankerää-

jän operaatioita, suorituksen aikaisia JIT-kääntäjän (*eng. Just In Time-compiler*) suoritusaikaa ja tietokoneen muuta kuormitusta. Suoritusaika ei sisällä aikaa, joka kuluu muistin allokoitiin, sillä allokoinnit ja luokkien alustukset on sijoitettu mittausvälien ulkopuolelle. Näitä virhelähteitä korjataan suorittamalla ohjelmaa viisi kertaa yhdellä verkolla ja käyttämällä suoritusajan mediaania arviona suorituksen kestosta.

Kokeet suoritettiin Itä-Suomen Yliopiston Laskuri 3 -koneella. Koneessa on Intel XEON E3-1245 suoritin, neljä fyysistä x86-käskykantaista suoritinydintä ja tuki kahdeksalle säikeelle Hyper Thread -ominaisuudella. Suorittimien maksimikellotaajuus on 3,3 GHz ja suorittimella on kahdeksan megatavua L3-välimuistia. Koneessa on kahdeksan gigatavua ECC-tuettua DDR3-1333 muistia.

Kokeet suoritettiin jakamalla ennalta generoidut verkot kolmelle itsenäiselle prosessille ja jokaiselle käytettäväksi annettiin yksi gigatavu muista. Jokainen prosessi pyrki suorittamaan sille osoitetut verkot satunnaisessa järjestyksessä ja satunnaistetulla järjestyksellä prioriteettijonoja. Jokainen suorituskerta sisälsi lämmittelyajon, joka mittasi verkon operaatioiden lukumäärän ja viisi kappaletta ajanmittausajoja, joiden mediaani esitetään operaation suoritusaikana.

Suoritusajat mitattiin jokaiselle operaatioille lisääminen, poistaminen ja päivittäminen erikseen, ei koko ohjelman suoritusaikaa. Epäyhtenäiset verkot suoritettiin loppuun, jostakin saavuttamattomasta solmusta eteen päin ja algoritmin suoritus päättyi vasta sitten, kun kaikkiin solmuihin on löytynyt lyhin polku.

Osa CF-verkoista oli sen verran suuria, ettei yksi gigatavu muistia riittänyt kokeen suorittamiseen. Kokeista epäonnistuneet verkot suoritettiin uudelleen kolmella prosessilla ja kahdella gigatavulla muistia. Tässäkin tapauksessa osalle verkoista muisti ei riittänyt ja viimeinen erä suoritettiin kahdella prosessilla ja kolmella gigatavulla muistia. Välimuistin määrän vaihtelu saattaa näkyä tuloksissa.

5 Tulokset

Tässä luvussa esitellään kokeiden tulokset ja verrataan niitä saman kaltaiseen Oberhauserin ja Simhan (1995) kokeeseen. Tulokset esitetään operaatioiden lukumäärien summina tai suoritusaikojen mediaanien summina ja tuloksissa esitetään 95%-luottamusvälit. Tuloksia havainnollistavissa kuvissa käytetään rakenteista nimistä seuraavia lyhenteitä:

BH taulukoitu binäärikeko

LBH linkitetty binäärikeko

LH vasenta suosiva keko

ILH implisiittisen poistamisen vasenta suosiva keko

AVL AVL-puu

IAVL implisiittisen poistamisen AVL-puu

SH itseisesti kiertyvä puu

ISH implisiittisen poistamisen itseisesti kiertyvä puu

FH Fibonacci-keko

IFH implisiittisen poistamisen Fibonacci-keko

Tuloksista ensimmäisenä esitellään keko- ja hakurakenteiden erot erilaisten verkkotyyppien suorittamisessa. Toiseksi verrataan rakenteita pareittain ja ryhmissä harvoilla satunnaisverkkoilla. Kolmanneksi verrataan rakenteita pareittain ja ryhmissä harvoilla täydellisen kaltaisilla verkoilla ja valikoidaan suorituskykyisimmät vertailuun tiheillä täydellisen kaltaisilla verkoilla. Tavoitteena on löytää suorituskykyisimmät algoritmit erilaisille verkoille.

5.1 Verkkotyypit

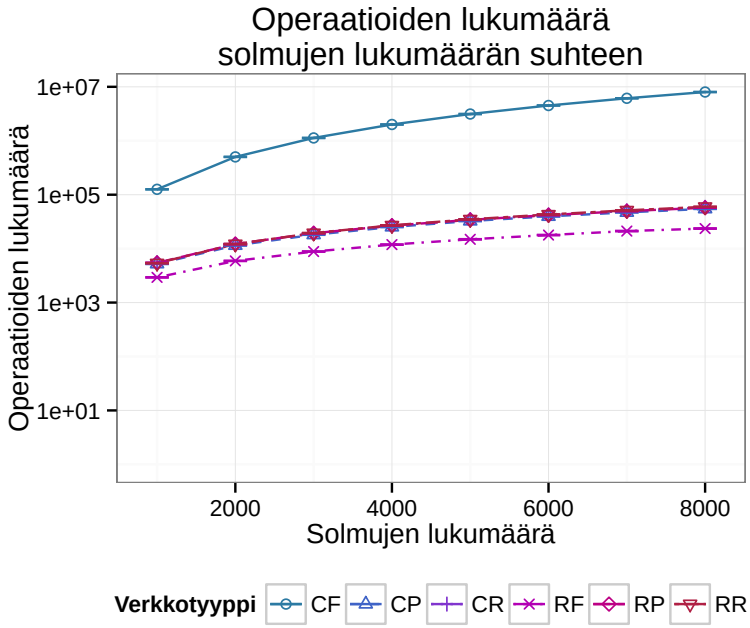
Kuvassa 21a on binäärikeon operaatioiden suorituskertojen summat eri verkkotyypeillä. Kuvasta nähdään, että verkkotyypit eivät erotu RR-verkoista, lukuun ottamatta CF-verkkoja ja RF-verkkoja. CF-verkkojen suorittaminen vaatii muita enemmän operaatioita ja RF-verkot muita vähemmän. Kuvassa 21b on AVL-puun operaatioiden suorituskertojen summat eri verkkotyypeillä. Kuvasta nähdään, että verkkotyypit eivät erotu RR-verkoista, lukuun ottamatta CF-verkkoja ja RF-verkkoja. CF- ja RF-verkkojen suorittaminen vaatii muita enemmän operaatioita.

Kuvia 21a ja 21b vertailemalla nähdään lisäksi, etteivät erot keko- ja hakurakenteiden välillä ole suuret. Kuvien erot RF-verkoissa johtuvat siitä, etteivät kekorakenteet käytä hakurakenteiden tapaan avaimia. Hakurakenteet vaativat avaimilta tarkan järjestyksen, mikä haittaa hakurakenteita RF-verkoissa, kun puolestaan kekorakenteet hyötyvät pelkän prioriteetin vertailusta samoissa verkoissa. CF-verkkojen operaatioiden suorituskerroissa keko- tai hakurakenteet eivät eroa. Kuvien muiden verkkojen samankaltaisuus perustellaan Goldbergin ja Tarjanin (1996) teoreettisella tuloksella: Dijkstran algoritmi käyttäytyy suurimmalla osalla satunnaisverkoista kertaluokkaa pienemmällä lukumäärällä operaatioita kuin pahimmassa tapauksessa.

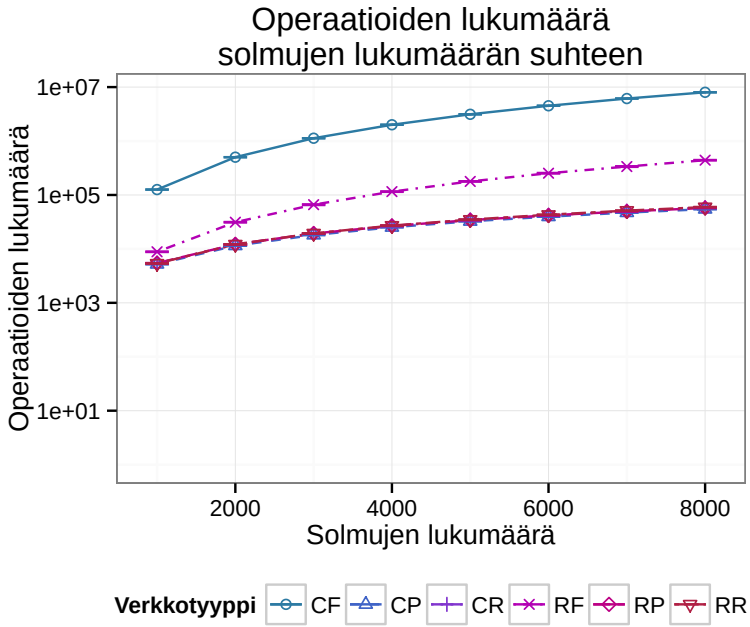
Tutkielmassa keskitytään tarkastelemaan vain CF-verkkoja ja RR-verkkoja verkkojen CP, CR, RF ja RP sijaan. Operaatioiden lukumäärät ovat kaikilla haku- ja kekorakenteilla tarkalleen samat CF-verkoissa ja RR-verkoissa likimäärin samat. RR-verkot lisäksi kattavat muut tapaukset paitsi RF-verkot. Koska operaatioiden lukumäärillä on vahva vaikutus suoritusaikaan, niin RF-verkkojen suoritusaikoja ei vertailla haku- ja kekorakenteiden kesken.

5.2 RR-verkot

Seuraavaksi vertaillaan pareittain rakenteita RR-verkoilla kaarten määrän ollessa pieni eli $p = 25\%$. Pienestä kaarimäärästä seuraa, että suoritusajoissa korostuvat suori-



(a) Binäärikekoon kohdistuvien operaatioiden (lisäys, poisto ja päivitys) lukumäärä suhteessa solmujen lukumäärään erilaisissa verkoissa.



(b) AVL-puuhun kohdistuvien operaatioiden (lisäys, poisto ja päivitys) lukumäärä suhteessa solmujen lukumäärään erilaisissa verkoissa.

Kuva 21: Verkkotyyppien vertailu binäärikeolla ja AVL-hakupuulla.

tusympäristöstä johtuvat vaihtelut. Tuloksissa keskitytään suurien solmu- ja kaarimäärien tarkasteluun.

Kuvassa 22a on vertailu taulukoitua (BH) ja linkitettyä (LBH) binäärikekoa. Suurella solmujen määrällä taulukoitu on nopeampi kuin linkitetty. Linkitetty joutuu tekemään vähän enemmän töitä poistoissa ja lisäyksissä etsiessään solmun vanhempaa.

Kuvassa 22b on vasenta suosivien kekojen vertailu. Suurella solmujen määrällä implisiittinen poisto (ILH) on nopeampi kuin kahvoja käyttävä (LH), joka johtuu siitä, että päivittämisen toisessa vaiheessa yksittäisten alkioiden lisääminen suureen puuhun on tehotonta ja implisiittinen poistaminen välttää tätä työtä.

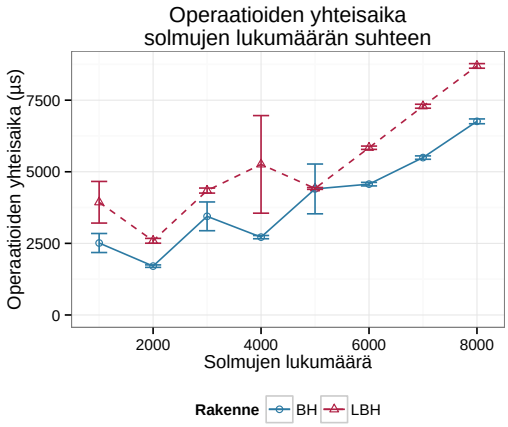
Kuvassa 22c on AVL-puiden vertailu. Ahkera AVL-puu (AVL) vie puolet implisiittisen poiston AVL-puun (IAVL) käyttämästä ajasta suuremmilla solmujen määrällä. Implisiittinen poisto jättää epäoleellisia alkioita puuhun, jotka kasvattavat puun kokoa.

Kuvassa 22d on itseisesti kiertyvän puun vertailut. Implisiittinen poistaminen (ISH) on suurempaa kertaluokkaa kuin ahkera itseisesti kiertyvä puu (SH). Implisiittisessä poistossa puu kiertyy oikealle kyljelleen, joka pahimmassa tapauksessa vastaa vektoria. Tällöin aikaisemmin muodostetut alipuut jäävät hyödyntämättä ja pahimman tapauksen aikavaativuus toteutuu jokaisen poiston yhteydessä. Sama pääsee rakenteessa toistumaan, koska puun alkiot ovat järjestyksessä, kun ne lisätään pinoon ja otetaan pinosta, jolloin rakenteesta kasataan uudestaan vektorimainen.

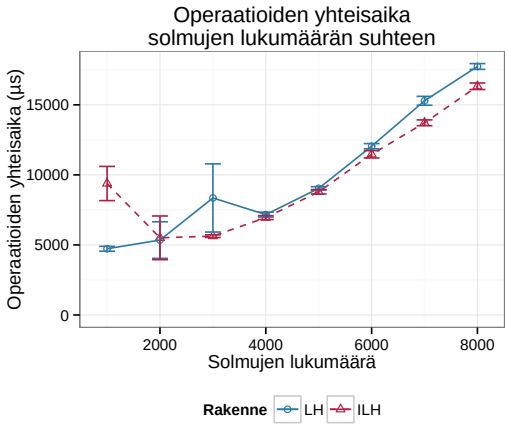
Kuvassa 22e on Fibonacci-kekojen vertailu. Ahkera Fibonacci-keko (FH) on tässä tapauksessa implisiittistä poistoa (IFH) nopeampi ja ero on vakion luokkaa.

Verrataan seuraavaksi ahkeria rakenteita ja implisiittisen poistamisen rakenteita ryhmissä keskenään. Kuvassa 23a on ahkerien rakenteiden vertailu. Tehokkaimmat rakenteet ovat binäärikeot (BH,LBH), joita seuraa Fibonacci-keko (FH) ja vasenta suosiva keko (LH) lähinnä eri kertoimella. Hakurakenteet (AVL,SH) käyttävät huomattavasti enemmän aikaa.

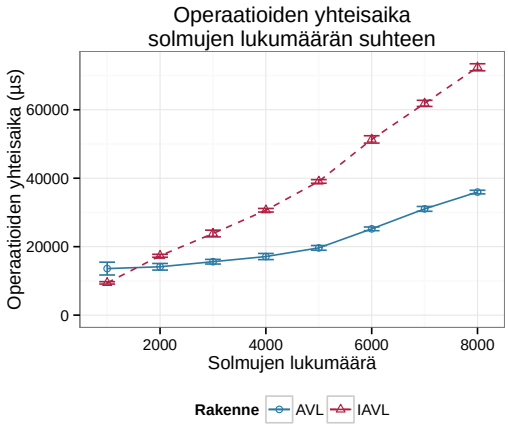
Kuvassa 23b verrataan implisiittisen poiston rakenteita toisiinsa ja referenssirakentei-



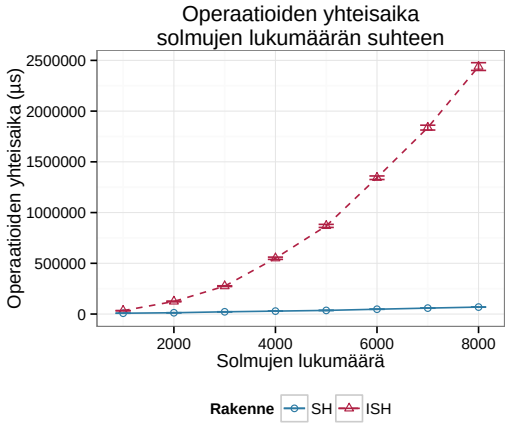
(a) Binäärikeko toteutusten vertailu RR-verkoilla.



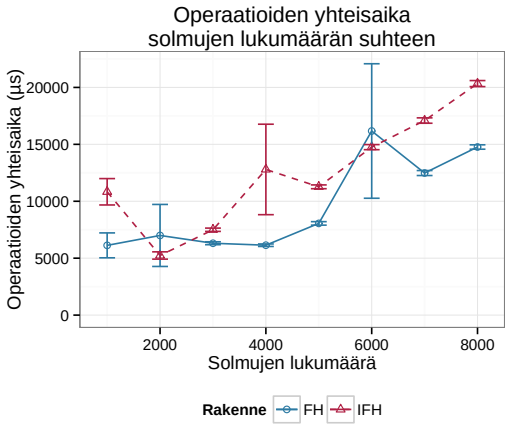
(b) Vasenta suosivan keon vertailu RR-verkoilla.



(c) AVL-puun vertailu RR-verkoilla.



(d) Itseisesti kiertyvän puun vertailu RR-verkoilla.



(e) Fibonacci-kekojen vertailu RR-verkoilla.

Kuva 22: Prioriteettijonojen pareittaiset vertailut RR-verkoilla.

siin eli binäärikekoihin (BH,LBH). Järjestyksessä suurella solmujen määrällä tehokkaimpina ovat ahkerat binäärikeot (BH,LBH). Implisiittisen poiston menetelmistä vasenta suosiva keko (ILH) on tässä Fibonacci-kekoa (IFH) tehokkaampi. Implisiittisen poistamisen AVL-puu (IAVL) on vertailussa tehottomin.

5.3 CF-verkot

Vaihdetaan seuraavaksi tarkasteltava verkko pahimman tapauksen verkkoihin (CF-verkkoihin). CF-verkot ovat lähes identtisiä ja hyvin harvinaisia, joten tuloksia pitää suhtautua hieman varauksellisemmin. Näissä verkoissa kaarten määrä ja päivitysoperaation tehokkuus voimakkaasti dominoi ja implisiittinen poistaminen tyhjentää rakenteen ja rakentaa sen uudestaan pinosta. Samaan tapaan kuin edellä käydään ensin läpi tulokset pareittain ja sitten ryhmissä. Ensin esitetään tulokset pienellä prosenttiosuudella $p = 25\%$.

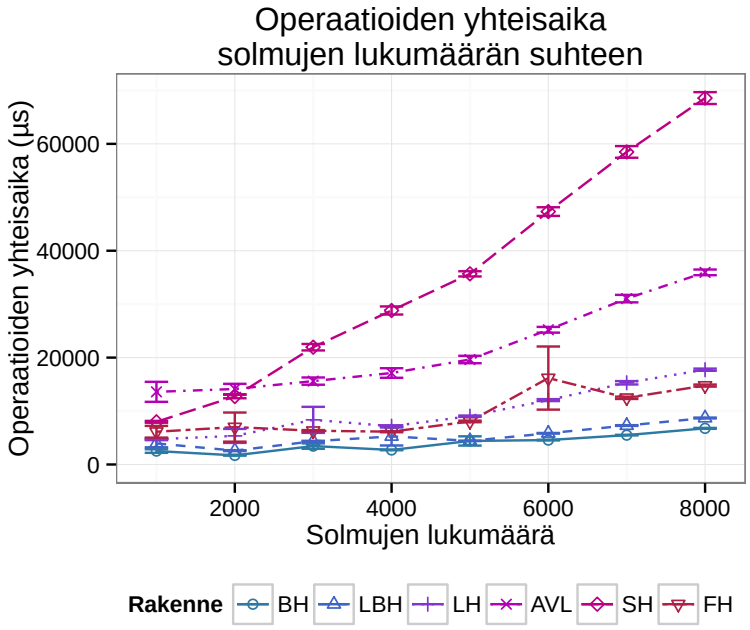
Kuvassa 24a verrataan taulukoitua (BH) ja linkitettyä (LBH) binäärikekoa. Solmujen määrän kasvaessa ero rakenteiden suoritusajoissa katoaa. Päivitysoperaatio on hyvin saman tapainen molemmissa tapauksissa ja muut erot ovat merkityksettömän pieniä.

Kuvassa 24b on vasenta suosivien kekojen vertailu (LH, ILH). Solmujen määrän kasvaessa ero rakenteiden suoritusajoissa katoaa. Tarjanin (1983) mukaan rakenne tekee tässä tapauksessa lineaarisen työn alkiomäärään nähden.

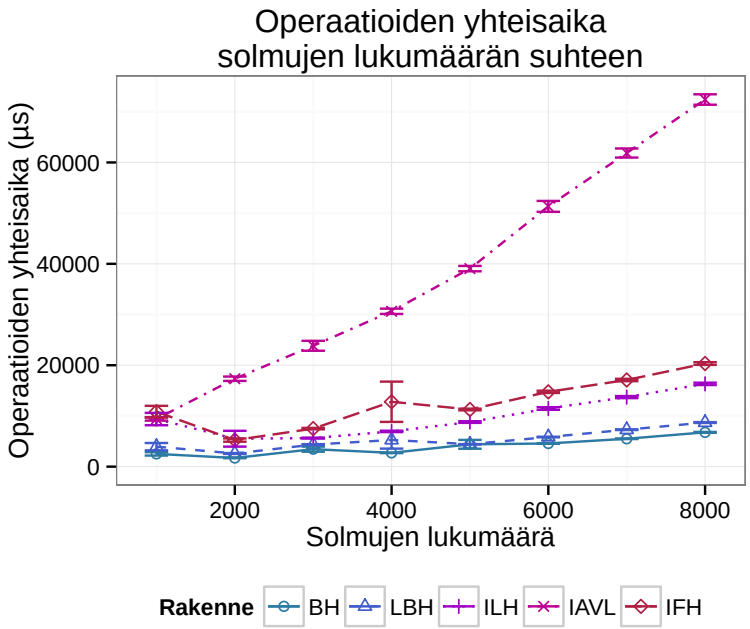
Kuvassa 24c on AVL-puiden vertailu. Implisiittinen poistaminen AVL-puussa (IAVL) on nopeampaa kuin sen ahkera päivittäminen (AVL).

Kuvassa 24d on itseisesti kiertyvän puun vertailut. Implisiittinen poistaminen itseisesti kiertyvässä puussa (ISH) on nopeampi kuin sen ahkera päivittäminen (SH). Implisiittisen poistamisen ero ahkeraan on kertoimen luokkaa. Yhdessä edellisen kuvan kanssa nähdään, että itseisesti kiertyvällä puulla ja AVL-puulla on vain kertoimen verran eroa.

Kuvassa 24e on Fibonacci-kekojen vertailu. Ahkera Fibonacci-keko (FH) on tässä tapauksessa implisiittistä poistoa (IFH) nopeampi ja ero on kertoimen luokkaa.



(a) Ahkeran suorituksen rakenteiden vertailu RR-verkoilla.



(b) Implisiittisen poiston toteuttavien rakenteiden vertailu RR-verkoilla.

Kuva 23: Ryhmittäiset vertailut RR-verkoilla.

Fibonacci-keko on suunniteltu suurelle määrälle kaaria, joten implisiittinen poistaminen ei saa tässä vastaavaa etua kuin hakupuiden tapauksissa.

Kuvassa 25a on implisiittisten kekorakenteiden vertailu. Vasenta suosiva keko (ILH) pärjää paremmin kuin Fibonacci-keko implisiittisellä poistolla (IFH).

Kuvassa 25b on binäärikekojen (BH,LBH), vasenta suosivan keon (LH) ja Fibonacci-keon (FH) vertailu. Vasenta suosiva keko on muita tehottomampi. Binäärikeko- ja Fibonacci-kekorakenteiden erosta ei voi tämän perusteella sanoa mitään.

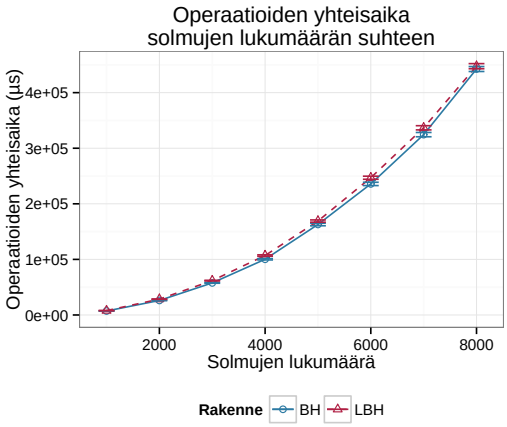
Kuvassa 25c verrataan Fibonacci-kekoa (FH) ja binäärikekoja (BH,LBH) suuremmalla kaarien prosenttiosuudella $p = 95\%$. Fibonacci-keko on kolmikon ahkerista rakenteista tehokkain suurella kaarimäärällä.

5.4 Vertailu muihin tuloksiin

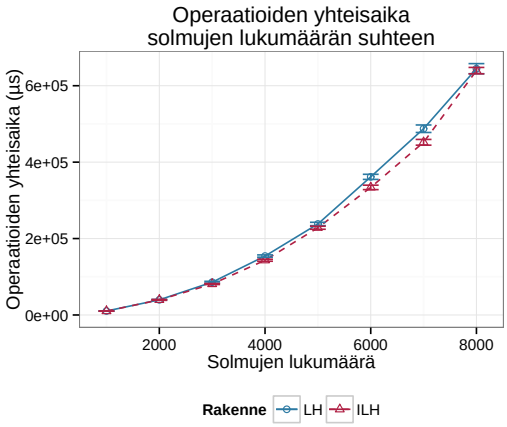
Oberhauser ja Simha (1995) ovat artikkelissaan verranneet binäärikekoa, Fibonacci-kekoa, itseisesti kiertyvää kekoa ja muutamia muita kekoja myös Dijkstran algoritmilla. Heidän menetelmänsä eroaa muutamissa kohdissa tutkielmassa esitystä tavasta. Verkon generointitapa eroaa siinä, että solmujoukkoon arvotaan kaaria kunnes riittävä prosenttiosuus on täytetty ja verkon yhtenäisyys on varmistettu. Verkon generointitavalla ei aiemman esitetyn tuloksen perusteella ei ole juuri merkitystä.

Suoritusaikojen mittaustapa on lähes samanlainen kuin tutkielmassa esitetty. Koska heidän toteutuksensa oli C-kielellä ei roskan kerääjä tai JIT-kääntäjä aiheuta heidän tuloksiinsa suurempaa vaihtelua. Tuloksensa he ovat esittäneet operaation suoritusaikojen keskiarvoina tutkielmassa esitettyjen mediaanien summien sijaan.

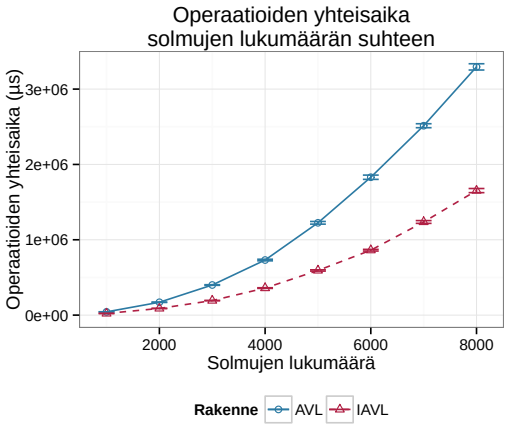
Tulokset ovat varsin saman suuntaisia kuin satunnaisella verkolla tässä tutkielmassa. Itseisesti kiertyvä puu oli yksi tehottomimmista rakenteista ja binäärikeko yksi tehokkaimmista, Fibonacci-keon sattuessa näiden väliin.



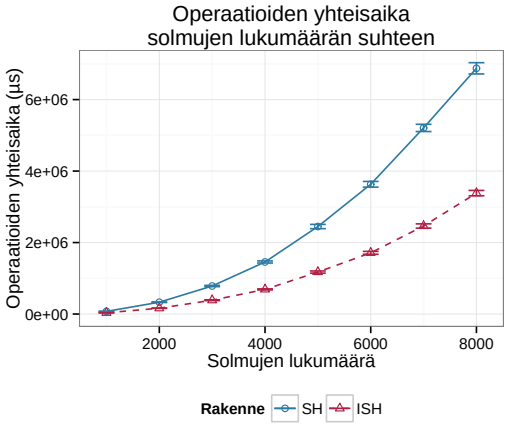
(a) Binäärikekototeutusten vertailu CF-verkoilla.



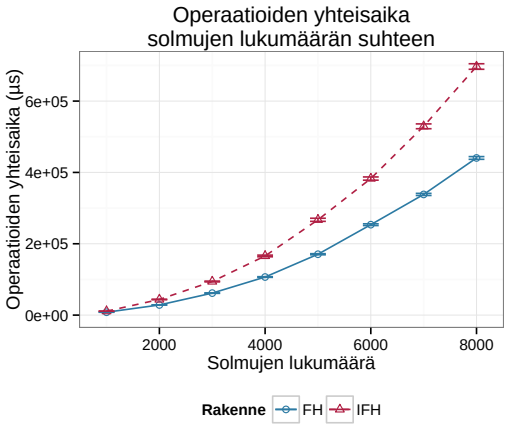
(b) Vasenta suosivan keon vertailu CF-verkoilla.



(c) AVL-puun vertailu CF-verkoilla.

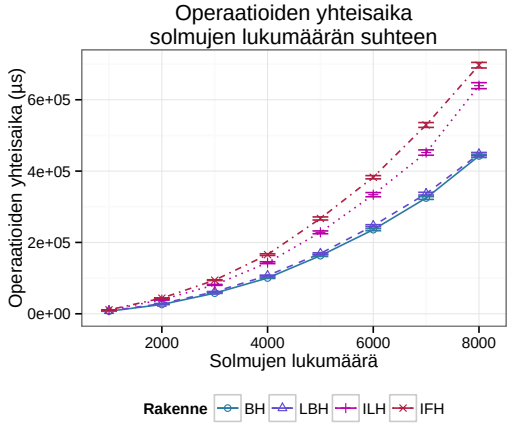


(d) Itseisesti kiertyvän puun vertailu CF-verkoilla.

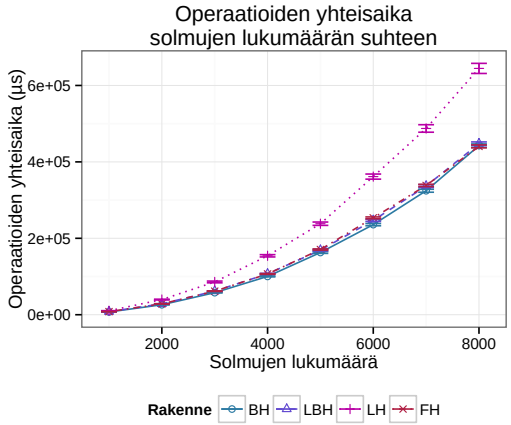


(e) Fibonacci-kekojen vertailu CF-verkoilla.

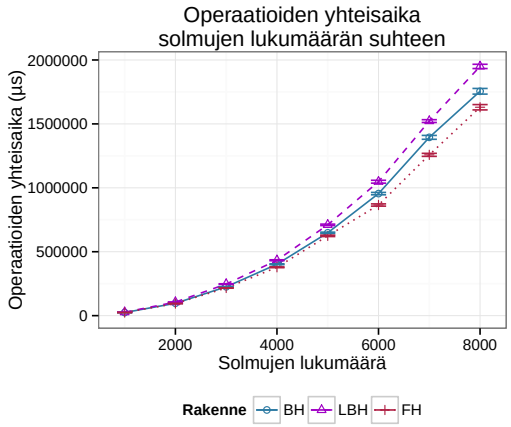
Kuva 24: Parittaiset vertailut CF-verkoilla.



(a) Implisiittisen poiston toteuttavien rakenteiden vertailu CF-verkossa.



(b) Kahvallisten kekojen vertailu CF-verkossa ($p = 25\%$).



(c) Fibonacci-keon ja binäärikeon vertailu CF-verkossa ($p = 95\%$).

Kuva 25: Ryhmittäiset vertailut CF-verkoilla.

6 Yhteenveto

Luvussa 2 määriteltiin verkko, polku ja puut. Puille määriteltiin ehtoja, joiden avulla kuvataan tietorakenteen konsistentti tila. Luvussa määriteltiin Dijkstran algoritmi, joka ratkaisee lyhimpien etäisyyksien ongelman apunaan prioriteettijono, jolta vaadittiin kolmea operaatiota LISÄÄ UUSI, PÄIVITÄ PRIORITEETTI ja POISTA TÄRKEIN. Dijkstran algoritmista esiteltiin toinen versio, jossa hyödynnetään implisiittisen poistamisen mahdollistamaa laiskuutta ja siirrettiin suoritusajan painopistettä useimmin hyödynnetyltä päivitys-operaatiolta harvemmin käytetylle poisto-operaatiolle. Luvussa tarkasteltiin prioriteettijonon rajapintaa ja verrattiin sen toteuttavien rakenteiden rajapintoja.

Luvussa 3 esiteltiin prioriteettijonon rajapinnan toteuttavia tietorakenteita ja niille mahdollisia implisiittisen poiston vastineita. Rakenteiksi valikoitui kirjallisuudesta entuudestaan tuttuja keko- ja hakurakenteita. Valikoidut rakenteet ovat binäärikeko, vasenta suosiva keko, AVL-puu, itseisesti kiertyvä puu ja Fibonacci-keko. Implisiittisen poiston ideaa sovellettiin kekorakenteista, joissa sitä kirjallisuudessa oli käytetty, hakurakenteisiin, joissa sitä ei aiemmin ollut esitetty.

Luvussa 4 esiteltiin aineisto ja ympäristö prioriteettijonojen suoritusaikojen mittaamiseen Dijkstran algoritmissa. Aineisto koottiin satunnais- ja täydellisen kaltaisista verkoista kolmella painofunktiolla, jotka olivat räätälöity w_f, sen satunnainen permutaatio w_p ja tasajakautunut satunnainen w_r. Koeympäristönä toimi Java-virtuaalikone yliopiston palvelimella Laskuri 3:lla.

Luvussa 5 esiteltiin tulokset graafisesti ja pohdittiin havaittujen erojen syitä. Ensimmäiseksi todettiin tulos satunnaisten verkkojen käyttäytymisen samankaltaisuudesta vertaamalla keko- ja hakurakenteiden operaatiomääriä eri verkkotyypeillä. Samankaltaisuudesta johtuen tulosten esittely rajattiin satunnais- ja pahimman tapauksen verkkoihin. Tuloksissa etsittiin tehokkainta tietorakennetta eri verkkotyyppeille.

Seuraavasssa vertailussa käytettiin suhteellisesti harvempia satunnaisverkkoja. Rakenteita vertailtiin pareittain implisiittisen poistamisen kanssa ja vertailua jatkettiin ryh-

missä ahkerat ja laiskat keskenään. Kekorakenteet implisiittisellä poistolla olivat kertoimen erolla ahkeriin nähden hitaampia. Kekorakenteille uudelleen kokoaminen, epäoleellisia alkioita sinne jättäen, on suhteellisen tehokas tapa toteuttaa implisiittinen poistaminen. Hakurakenteissa implisiittisen poistamisen erot ahkeraan olivat kertaluokkaa suuremmat. Hakurakenteiden uudelleen kokoaminen epäoleellisia alkioita sekaan jättäen ei ole tehokas tapa toteuttaa implisiittinen poistaminen. Harvojen satunnaisverkkojen tehokkain prioriteettijonon toteutus on binäärikeko.

Suhteellisen harvoja pahimman tapauksen verkkoja käytettiin seuraavassa vertailussa. Rakenteita vertailtiin pareittain implisiittisen poistamisen kanssa ja vertailua jatkettiin ryhmissä ahkerat ja laiskat keskenään. Kekorakenteiden ahkerat ja laiskat versiot eivät eronneet toisistaan lukuun ottamatta Fibonacci-kekoa, joka on suunniteltu pahinta tapausta varten. Hakurakenteilla uuden puun rakentamiseen alusta menee vähemmän aikaa kuin ahkeraan päivittämiseen. Implisiittisen poistamisen vasenta suosiva keko on tehokkain implisiittisen poistamisen prioriteettijonon toteutus. Tulosten tarkastelua jatkettiin tiheillä pahimman tapauksen verkoilla, joilla saatiin näkyvä ero Fibonacci-keon ja binäärikeon välille. Tiheissä pahimman tapauksen verkoissa tehokkain prioriteettijonon toteutus on Fibonacci-keko.

Dijkstran algoritmilla käytännössä ratkaistaan ongelmia, jotka kuvataan harvoilla verkoilla. Implisiittisen poistamisen hyödyt edellyttävät tiheää verkkoa, jossa päivitysoperaatioita tulee paljon ennemmän kuin poisto-operaatioita. Lisäksi Dijkstran algoritmin kyky vähentää päivitysoperaation tarvetta satunnaisverkoissa tarkoittaa, että hyvin harvoin tarvitaan aikavaativuudeltaan parempaa prioriteettijonon toteutusta kuin mitä binäärikeko yksinkertaisuudessaan tarjoaa. Implisiittinen poistamisen tarve tulee ennemmin tilanteista, joissa kyky osoittaa tietoa tai muuttaa koneen tilaa tarkasti on rajoittunut, kuten funktionaalisen ohjelmoinnin kanssa. Tutkielmalla voidaan perustella jatkotutkimusta kehittää kahvattomien prioriteettijonojen toteutuksia kekoehdolla, kuten funktionaalista vasenta suosivaa kekoa.

Viitteet

Algorithms-library (n.d.), `http://en.cppreference.com/w/cpp/algorithm`. Haettu: 15.10.2013.

Brodal, G. S., Lagogiannis, G. ja Tarjan, R. E. (2012), 'Strict Fibonacci Heaps', *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing* pp. 1177–1184.
URL: *http://doi.acm.org/10.1145/2213977.2214082*

Brodal, G. S. ja Okasaki, C. (1996), 'Optimal Purely Functional Priority Queues', *Journal of Functional Programming* **6**(6), 839–857.

Dijkstra, E. W. (1959), 'A Note on Two Problems in Connexion with Graphs', *Numerische Mathematik* **1**, 269–271.

Driscoll, J. R., Gabow, H. N., Shrairman, R. ja Tarjan, R. E. (1988), 'Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation', *Communications of the Association for Computing Machinery* **13**(11), 1343–1354.

Fredman, M. ja Tarjan, R. (1987), 'Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms', *Journal of the Association for Computing Machinery* **34**(3), 596–615.

Goldberg, A. V. ja Tarjan, R. E. (1996), Expected Performance of Dijkstra's Shortest Path Algorithm, Technical report, NEC Research Institute Report.

Haeupler, B., Sen, S. ja Tarjan, R. E. (2009), 'Rank-Balanced Trees', *Workshop on Algorithms and Data Structures* pp. 351–362.

Hoogerwoord, R. R. (1983), 'A Logarithmic Implementation of Flexible Arrays', *Conference on Mathematics of Program Construction* pp. 191–207.

Hull, T. E. ja Dobell, A. R. (1962), 'Random Number Generator', *SIAM Review* **4**(3), 230–254.

Knuth, D. (1973), *The Art of Computer Programming. volume 3, Sorting and Searching*, Addison-Wesley, Reading, MA.

Oberhauser, G. ja Simha, R. (1995), 'Fast Data Structures for Shortest Path Routing', *IEEE Communications* pp. 1597–1601.

Oracle (2011), 'PriorityQueue (Java Platform SE 6)', `http://docs.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html`. Haettu: 15.10.2013.

Peyton-Jones, S. (2003), *Haskell 98 language and libraries :the revised report*, Cambridge University Press, Cambridge U.K.
URL: *http://www.worldcat.org/isbn/9780521826143*

Sleator, D. D. ja Tarjan, R. E. (1985), 'Self-Adjusting Binary Search Trees', *Journal of the Association for Computing Machinery* **32**(3), 652–686.

std::priority_queue (n.d.), `http://en.cppreference.com/w/cpp/container/priority_queue`. Haettu: 15.10.2013.

Tarjan, R. E. (1983), *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.

Vuillemin, J. (1978), 'A Data Structure for Manipulating Priority Queues', *Communications of the Association for Computing Machinery* **21**(4), 309–315.