TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HENRI MYLLYOJA

BINÄÄRI- JA FIBONACCI-KEOT PRIORITEETTIJONON TOTEUTUKSEEN

Kandidaatintyö

Tarkastaja: Sami Hyrynsalmi
Jätetty tarkastettavaksi 24.10.2017

TIIVISTELMÄ

HENRI MYLLYOJA: Binääri- ja Fibonacci-keot prioriteettijonon toteutukseen
Tampereen teknillinen yliopisto
Kandidaatintyö, 18 sivua
Lokakuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotekniikka
Tarkastajat: Sami Hyrynsalmi
Avainsanat: prioriteettijono, binäärikeko, Fibonacci-keko

Tämän työn tavoite on tutkia binääri- ja Fibonacci-kekojen soveltuvuutta prioriteettijonon toteutukseen ja vertailla niitä käytännöllisellä ja teoreettisella tasolla. Prioriteettijono on yleisesti käytössä oleva tietorakenne, jota käytetään muun muassa erilaisissa graafialgoritmeissa.

Binäärikeko on yksinkertainen ja tehokas rakenne prioriteettijonon toteutukseen. Se on myös yleisin tapa toteuttaa prioriteettijono ohjelmistoissa. Monet ohjelmointikielet tarjoavat binäärikekoprioriteettijonon standardikirjastoissaan. Toinen työssä käsiteltävä kekorakenne, Fibonacci-keko, tarjoaa keko-operaatiot asymptoottisesti binäärikekoa tehokkaammin monimutkaisemman rakenteen kustannuksella.

Kummankin keon rakenne esitellään yksityiskohtaisesti. Lisäksi niiden tehokkuuksia vertaillaan Dijkstran ja Primin algoritmeissa. Fibonacci-keolla toteutettuna kumpikin näistä algoritmeista on asymptoottisesti tehokkaampi.

Fibonacci-keon tehokkuusetuja voi kuitenkin olla käytännössä vaikea saavuttaa keon monimutkaisen rakenteen johdosta. Siten sen merkitys korostuu teoreettisena pohjana tehokkaammille tietorakenteille.

SISÄLLYS

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

F-keko	Fibonacci-keko
STL	Standard Template Library, C++:n standardikirjasto

O	Iso-O-notaatio, asymptoottinen yläraja
Θ	Theta-notaatio, asymptoottinen ylä- ja alaraja

1. JOHDANTO

Prioriteettijono on jonon kaltainen tietorakenne, jossa alkioihin on liitetty prioriteetti. Alkioita käsitellään prioriteetin mukaisessa järjestyksessä. Prioriteettijonoa käytetään laajasti muun muassa verkko-optimointialgoritmeissa kuten Dijkstran algoritmissa, jolla etsitään lyhin reitti graafin solmusta toiseen solmuun. Dijkstran algoritmia käytetään muun muassa navigaattoreissa ja reitityksessä, joissa kummassakin verkostoa mallinnetaan painotettuna graafina. Toinen yleinen käyttökohde prioriteettijonolle on kevyimmän virittävän puun etsiminen. Kevyimpiä virittäviä puita käytetään muun muassa luokittelussa [6] ja yleislähetyksiin tietoverkoissa [4]. Lisäksi prioriteettijonoa käytetään esimerkiksi prosessien vuorontamisessa.

Vaatimuksena tehokkaalle prioriteettijonolle on pienimmän alkion nopea löytäminen. Prioriteettijono toteutetaan usein jonkinlaisena kekona. Keko puolestaan toteutetaan usein puuna tai puukokoelmana, jossa pienin alkio löytyy puun (tai jonkin puun) juuresta.

Binäärikeko on yleisin tapa toteuttaa prioriteettijono, ja se on toteutettu monissa ohjelmointikielissä. Keon toteuttamiseen on kuitenkin esitelty teoreettisesti tehokkaampia tietorakenteita kuten Fibonacci-keko (jatkossa F-keko). Tässä työssä vertaillaan kahden edellä mainitun tietorakenteen tehokkuuksia eri käyttötarkoituksissa ja tutkitaan F-keon teoreettista käyttökelpoisuutta. Lisäksi sivutaan Rintalan ja Valmarin tutkimukseen [7] perustuen mahdollisuuksia parantaa binäärikeon tehokkuutta.

Binäärikekoa käsitellään luvussa 2 ja Fibonacci-kekoa luvussa 3. Nämä luvut sisältävät teknisen kuvauksen tietorakenteiden toteuttamisesta, tehokkuusanalyysin sekä mainintoja toteutuksista eri ohjelmointikielissä. Luvussa 4 tutkitaan kekojen käyttöä lyhimmän reitin ja kevyimmän virittävän puun etsimiseen graafeissa. Yhteenveto on luvussa 5.

2. BINÄÄRIKEKO

Binäärikeon esitteli J. W. J. Williams vuonna 1964 tietorakenteeksi kekolajittelualgoritmia (engl. *heapsort*) varten [8]. Vaikka kekolajittelu, kuten suurin osa muistakin lajittelualgoritmeista, on aikavaatimukseltaan $O(n \log n)$[1], on se käytännössä hitaampi kuin esimerkiksi quicksort [3, s. 162]. Se ei myöskään ole vakaa, eli samansuuruisten alkioiden keskinäinen järjestys voi muuttua. Binäärikeolle on kuitenkin muitakin käyttökohteita kuten prioriteettijono.

2.1 Keko

Keko[2] on tietorakenne, joka toteuttaa vähintään seuraavat operaatiot:

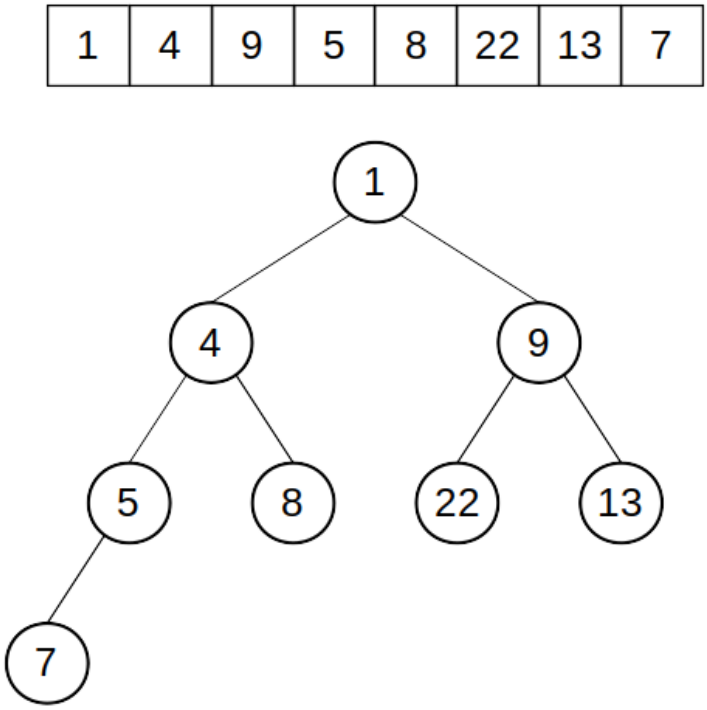
- `Make-heap`: luo uuden tyhjän keon
- `Insert`: sijoittaa uuden alkion kekoon
- `Find-min`: etsii keon pienimmän alkion
- `Extract-min`: palauttaa ja poistaa keon pienimmän alkion

Keko-sanaa käytetään myös synonyyminä binäärikeolle, mutta tässä työssä keon määritelmänä käytetään edellä kuvattua abstraktia tietorakennetta.

Lisäksi määritellään yhdistettäväksi keoksi (engl. *mergeable heap*) tietorakenne, joka toteuttaa edellä lueteltujen operaatioiden lisäksi `Union`-operaation (kirjallisuudessa myös `Meld` tai `Merge`) [3, s. 505]. Näiden lisäksi voidaan toteuttaa muita operaatioita kuten `Decrease-key`, joka vähentää alkion arvoa tai `Delete`, joka poistaa mielivaltaisen alkion.

[1] Operaatioiden tehokkuuksia esitetään tässä työssä Theta- ja iso-O-notaatioilla (Θ ja O), joista ensimmäinen kuvaa operaation asymptoottista ylä- ja alarajaa ja jälkimmäinen pelkkää ylärajaa.

[2] Tässä työssä käsitellään selkeyden vuoksi vain minimikekoa. Maksimikeolle pätevät vastaavat operaatiot käänteisesti.



Kuva 2.1 *Binäärikeon esitys taulukkona ja puuna.*

2.2 Binäärikeon määritelmä

Binäärikeko on lähes täydellisesti tasapainoitettu binääripuu, jossa jokainen solmu on pienempi tai yhtäsuuri kuin sen lapsisolmu. Binäärikeko toteutetaan taulukkona, jonka alkioden järjestys toteuttaa ehdon $A[i] \leq A[j] : 2 \leq j \leq n, i = j/2$ (ensimmäinen alkio on A[1] ja n on keossa olevien alkioiden lukumäärä). Määritelmästä seuraa, että alkio A[1] on aina keon pienin alkio, mikä tekee sen löytämisestä helppoa. Lisäksi binäärikeko-oliolla on taulukon koosta eroava koko-ominaisuus. Kaikki taulukon alkiot eivät siis välttämättä kuulu kekoon [3, s. 151]. Tämä käy selkeimmin ilmi kekolajittelussa, jossa keon kokoa pienennetään sitä mukaa kun taulukkoa järjestetään. Kuvassa 2.1 esitetään binäärikeko taulukkona ja puuna.

Koska binäärikeko on lähes täydellisesti tasapainotettu binääripuu (vain alin kerros voi olla vajaa, ja se on täytetty vasemmalta oikealle), sen korkeus on $log_2 n$, missä n on solmujen määrä. Tästä seuraa, että monet binäärikekoon liittyvät algoritmit ovat aikavaatimukseltaan logaritmisia.

Binäärikeon määritelmästä seuraa, että isä- ja lapsisolmujen sijainnit taulukossa ovat helposti laskettavissa solmun järjestysluvun mukaan. Määritellään tätä varten

operaatiot isä- ja lapsisolmujen löytämiseksi. `Left` palauttaa vasemman ja `Right` oikean lapsisolmun, ja `Parent` palauttaa isäsolmun. Nämä operaatiot voidaan toteuttaa helposti bittisiirtoina [3, s. 152].

Parent(i) [3, s. 152]

```
1    return ⌊i/2⌋
```

Left(i) [3, s. 152]

```
1    return 2i
```

Right(i) [3, s. 152]

```
1    return 2i + 1
```

Koska binäärikeko toteutetaan taulukkona ja sitä voidaan käsitellä puuna edellä kuvatuilla operaatioilla, se ei tarvitse osoittimia puun solmujen yhdistämiseen. Näin ollen se kuluttaa vähemmän muistia perinteiseen osoittimilla toteutettuun puuhun verrattuna, ja taulukon indeksointi on nopeampaa osoittimia pitkin kulkemiseen verrattuna.

2.3 Binäärikeon toteutus

Binäärikeon alkion prioriteettia korotetaan `Decrease-key`-algoritmilla. Tätä algoritmia käytetään myös `Insert`in toteuttamiseen. Alkiolle etsitään uusi paikka vertailemalla sitä isäsolmuihin, kunnes oikea paikka löytyy:

Decrease-key(A, i, key) [3, s. 164]

```
1    input: A kekotaulukko, i alkion indeksi, key alkion uusi arvo
2
3    esiehto: key ≤ A[i]
4    A[i] ← key
5    while i > 1 and A[Parent(i)] > A[i]
6        exchange A[i] with A[Parent(i)]
7        i ← Parent(i)
```

Binäärikekoon lisätään alkio `Insert`-algoritmilla. Alkiolle etsitään paikka asettamalla se viimeiseksi äärettömällä avaimella ja sen jälkeen etsimällä sille paikka `Decrease-key`-algoritmilla:

Insert(A, key) [3, s. 164]

```
1    input: A kekotaulukko, key lisättävä alkio
2
```

```
3       A.heap_size ← A.heap_size + 1
4       A[A.heap_size] ← −∞
5       Decrease-key(A, A.heap_size, key)
```

Paikallisesti rikkoutunut kekoehto korjataan `Heapify`-algoritmilla. `Heapify` olettaa, että taulukko on juurta lukuun ottamatta keko. Heapify uudelleenjärjestää juuren ja sen lapset siten, että kekoehto toteutuu näille ja rekursiivisesti kutsuu itseään sille alipuulle, johon isäsolmu siirrettiin. Tätä algoritmia käytetään `Extract-min`in ja `Build-heap`in toteuttamiseen.

Heapify(A, i) [3, s. 154]

```
1       input: A kekotaulukko, i indeksi, jossa kekoehto rikkoutuu
2
3       l ← Left(i)
4       r ← Right(i)
5       if l ≥ A.heap_size and A[l] < A[i]
6           smallest ← l
7       else smallest ← i
8       if r ≥ A.heap_size and A[r] < A[smallest]
9           smallest ← r
10      if smallest ≠ i
11          exchange A[i] with A[smallest]
12          Heapify(A, smallest)
```

Keon pienin alkio voidaan lukea taulukon ensimmäisestä indeksistä. Pienin alkio poistetaan `Extract-min`-algoritmilla. Se korvataan ensin keon viimeisellä alkiolla, jonka jälkeen jäljellä olevat alkiot uudelleenjärjestetään keoksi ja pienin alkio palautetaan:

Extract-min(A) [3, s. 163]

```
1       input: A kekotaulukko
2
3       min ← A[1]
4       A[1] ← A[A.heap_size]
5       A.heap_size ← A.heap_size - 1
6       Heapify(A, 1)
7       return min
```

Tavallisen taulukon voi muuttaa keoksi `Build-heap`-algoritmilla. Taulukko käydään läpi puolesta välistä alkuun `Heapify`ta kutsuen. Loppupuolen solmuilla ei ole lapsia, joten niille kekoehto pätee automaattisesti.

Build-heap(A) [3, s. 157]

```
1       input: A keoksi muutettava taulukko
```

```
2
3       A.heap_size ← A.length
4       for i ← ⌊A.length/2⌋ downto 1
5           Heapify(A, i)
```

Binäärikeolle voidaan myös toteuttaa `Merge`-operaatio. Siinä kekotaulukot liitetään yhdeksi taulukoksi, jolle suoritetaan `Build-heap`-operaatio [3, s. 505].

Merge(A, B) [3, s. 505]

```
1       input: A ja B yhdistettävät kekotaulukot
2
3       C ← concatenate(A, B)
4       Build-heap(C)
5       return C
```

`Merge`ä kuitenkin harvoin toteutetaan sen tehottomuuden vuoksi.

2.4 Toteutukset ohjelmointikielissä

Binäärikeko on toteutettu useissa ohjelmointikielissä, kuten C++:ssa (`priority_queue`), Pythonissa (`heapq`) ja Javassa (`PriorityQueue`). Yksinkertaisuutensta vuoksi myös epävirallisia toteutuksia on runsaasti saatavilla eri kielille. Kuitenkaan mikään edellä mainituista toteutuksista ei tarjoa `Decrease-key`-operaatiota.

C++:n standardikirjaston eli STL:n `priority_queue`-tietorakenne tarjoaa perusoperaatiot `Insert`, `Find-min` ja `Extract-min`. Rajapinta ei kuitenkaan tarjoa mahdollisuutta muokata tietorakennetta erityistarpeiden mukaiseksi [7]. Binäärikeolle voidaan toteuttaa logaritminen `Decrease-key`-operaatio aliluvussa 2.3 kuvatulla tavalla, mutta se ei ole mahdollista standardikirjaston prioriteettijonolle. Rintalan ja Valmarin tutkimus [7] osoittaa, että itsetoteutettu `Decrease-key`n tarjoava prioriteettijono toimii tutkimuksessa käytetyille algoritmeille nopeammin kuin STL:n tarjoama prioriteettijono. Tutkimuksessa toteutettu prioriteettijono tallentaa solmujen sijainnit keossa erilliseen taulukkoon. Prioriteettijono-olio ylläpitää tätä taulukkoa, ja prioriteettia muutettaessa siitä nähdään alkion sijainti keossa.

2.5 Tehokkuusanalyysi

Binäärikeon operaatioiden aikavaatimukset on esitetty taulukossa 2.1. `Heapify`ssa vakioaikainen operaatio suoritetaan korkeintaan $\lfloor \log_2 n \rfloor$ kertaa, missä n on alkioiden määrä. `Insert`issä, `Decrease-key`ssä ja `Extract-min`issä kekoa käsitellään kor-

keintaan sen korkeuden verran, joten nämä operaatiot ovat korkeintaan logaritmisia. `Merge` on kuitenkin toteutettavissa vain lineaarisessa ajassa.

Taulukko 2.1 *Binäärikekoprioriteettijonon aikavaatimukset [3, s. 157–159, 506].*

Operaatio	Aikavaatimus
Build-heap	$\Theta(n)$
Heapify	$O(\log n)$
Insert	$O(\log n)$
Find-min	$\Theta(1)$
Extract-min	$O(\log n)$
Decrease-key	$O(\log n)$
Merge	$\Theta(n)$

`Build-heap`issä logaritminen `Heapify` suoritetaan $n/2$ kertaa, mistä saadaan sen aikavaatimukseksi $O(n \log n)$. Kuitenkin `Heapify`n aikavaatimus riippuu sen solmun, jolle operaatio suoritetaan, korkeudesta puussa. Jos solmun korkeus on h, `Heapify`n aikavaatimus tälle solmulle on $O(h)$, ja siten `Build-heap`in aikavaatimuksen yläraja on

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right). \tag{2.1}$$

Voidaan osoittaa, että $\sum_{h=0}^{\infty} \frac{h}{2^h}$ on vakio [3, s. 159], jolloin `Build-heap`in aikavaatimus pienenee lineaariseksi:

$$O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n). \tag{2.2}$$

Kaikki binäärikeon operaatiot ovat hyvin yksinkertaisia, joten niiden vakiokertoimet ovat pienet. Yksinkertaisen rakenteensa vuoksi binäärikeko ei myöskään vaadi muistia muulle kuin alkioiden arvoille. Jos `Merge`ä tarvitaan, binäärikeko ei kuitenkaan ole hyvä vaihtoehto.

3. FIBONACCI-KEKO

Fibonacci-keon esittelivät Fredman ja Tarjan [5] vuonna 1987. F-keon avulla muun muassa Dijkstran ja Primin algoritmit voidaan toteuttaa binäärikekoa asymptoottisesti tehokkaammin. Se tarjoaa logaritmista poistoa lukuun ottamatta perusoperaatiot amortisoidusti vakioaikaisina, ja lisäksi vakioaikaiset `Decrease-key`n ja `Merge`n. Se siis toteuttaa kaikki yhdistettävän keon vaatimat operaatiot.

F-keko vaatii muistia datan (ja mahdollisesti erillisen prioriteetin) lisäksi neljälle osoittimelle, yhdelle kokonaisluvulle ja yhdelle bitille. Datan koosta riippuen tämä voi olla huomattava lisäys muistinkulutuksessa. C- tai C++-kielten standardikirjastoissa ei ole toteutusta F-keolle, mutta se on toteutettu C++-kielelle *Boost*-kirjastossa [1]. Monille muille kielille löytyy myös epävirallisia toteutuksia.

3.1 Toteutus

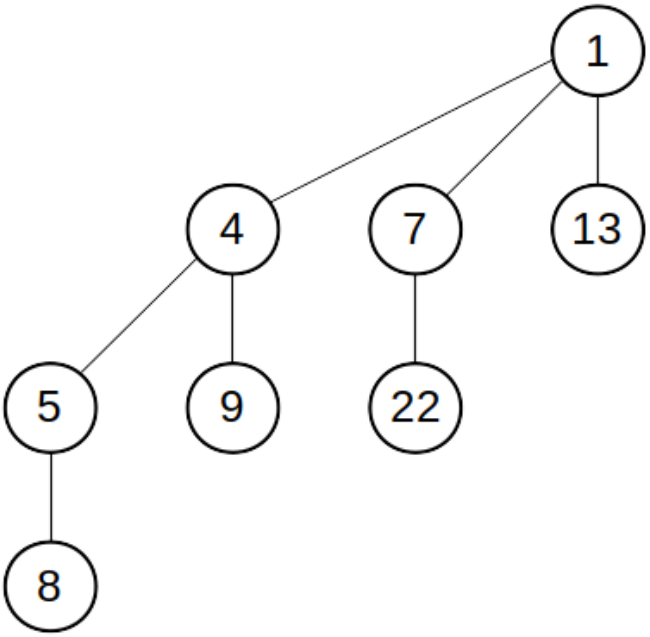
F-keko toteutetaan kokoelmana binomipuiden kaltaisia puita[1]. Näiden puiden juurisolmut kytketään toisiinsa kahteen suuntaan linkitettynä listana. Juurisolmua lukuun ottamatta kaikissa solmuissa on osoitin isäsolmuun, ja lehtisolmuja lukuun ottamatta kaikissa solmuissa on osoitin yhteen lapsisolmuun. Jokaisen solmun lapsisolmut on myös linkitetty toisiinsa kahteen suuntaan linkitettynä listana. Lisäksi tallennetaan osoitin pienimpään solmuun.

3.1.1 Binomipuut ja binomikeot

Binomipuu on puu, jolle pätevät seuraavat ehdot:

- Asteen 0 binomipuu on yksittäinen solmu.
- Asteen k binomipuussa juurisolmun lapset ovan asteiden k-1, k-2, ... 0 binomipuita.

[1] Vain `Decrease-key`- ja `Delete`-operaatiot voivat rikkoa binomipuun ehdot.



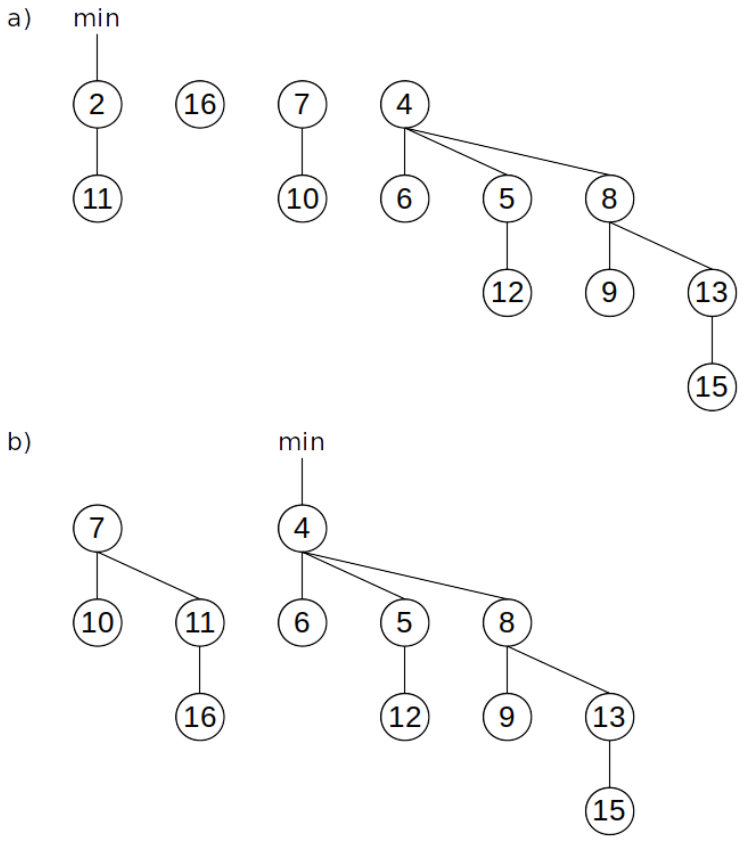
Kuva 3.1 *Asteen 3 binomipuu. Juurisolmun lapset ovat asteiden 2, 1 ja 0 binomipuita. Kuvattu puu täyttää myös kekoehdon.*

Binomikeko on binomipuu, jossa lapsisolmu on arvoltaan suurempi tai yhtä suuri kuin isäsolmu. Samanasteisia binomipuita voi binomikeossa olla korkeintaan yksi. F-keossa jälkimmäinen sääntö rikkoutuu joissain tilanteissa. Kuvassa 3.1 esitetään asteen 3 binomipuu. Juurisolmun lapsina ovat tässä asteiden 2, 1 ja 0 binomipuut.

Binomipuun rakenne sallii kahden k-asteisen puun yhdistämisen yhdeksi k+1-asteiseksi puuksi asettamalla ensimmäisen puun juurisolmun toisen puun juurisolmun lapseksi. Binomikeossa kekoehdon säilyttämiseksi tämä tehdään niin päin, että juurisolmuksi päätyy pienempi solmu. Tästä yhdistettävyydestä voidaan päätellä, että asteen kasvaessa yhdellä solmujen määrä kaksinkertaistuu. Koska asteen 0 binomipuu on yksi solmu, asteen k binomipuussa on tällöin 2^k solmua.

3.1.2 Operaatioiden toteutukset

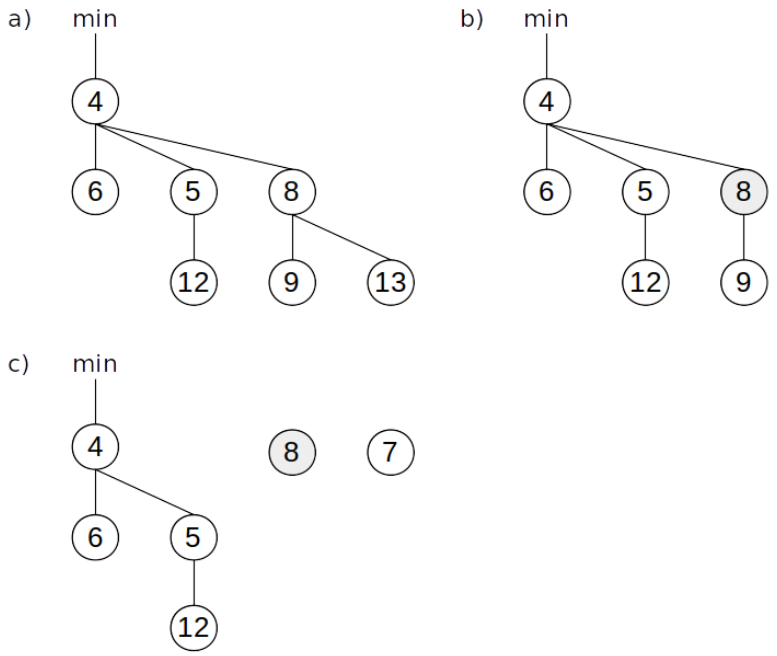
`Insert` toteutetaan luomalla uusi yhden solmun sisältävä keko, joka yhdistetään olemassa olevaan kekoon `Merge`-operaatiolla. `Merge` toteutetaan yhdistämällä yhteenliitettävien kekojen juurilistat ja valitsemalla minimisolmuksi pienimmän näiden kekojen minimisolmuista.



Kuva 3.2 *`Extract-min`-operaatio F-keossa. Vaihe a) on lähtötilanne. Minimialkio 2 poistetaan. Tämän jälkeen samanasteiset alkiot 11 ja 16 yhdistetään, sitten 7 ja 11. Vaihe b) on lopputilanne.*

`Extract-min` aloitetaan poistamalla pienin alkio ja yhdistämällä tämän lapsilista juurilistan kanssa. Seuraavaksi juurilistan puita yhdistetään, kunnes listassa ei ole kahta samanasteista puuta (tällöin binomikeon jälkimmäinen kekoehto täyttyy väliaikaisesti). Yhtä tällaista puiden yhteenliittämistä kutsutaan myöhemmässä analyysissä liittämisvaiheeksi (engl. *linking step*). Tähän käytetään osoitintaulukkoa, joka ensin alustetaan nollaosoittimiksi. Juurilista käydään läpi ja juurisolmujen osoittimet asetetaan taulukkoon juurisolmun asteen mukaiseen indeksiin. Jos tässä indeksissä on jo osoitin toiseen juureen, nämä kaksi k-asteista puuta yhdistetään yhdeksi k+1-asteiseksi puuksi, jonka osoitin asetetaan taulukkoon. Kun koko juurilista on käyty läpi, taulukon osoittimista kasataan uusi juurilista ja etsitään uusi minimisolmu. Kuva 3.2 esittää `Extract-min`-operaation vaikutuksen F-kekoon.

`Decrease-key`-operaatiossa solmun arvoa pienennetään ja se irrotetaan isäsolmustaan ja siirretään juurilistaan. Jos uusi avain on keon minimiä pienempi, tästä solmusta tehdään uusi minimisolmu. `Delete`-operaatiossa poistettava solmu irrotetaan isäsolmustaan samaan tapaan, ja sen lapsilista yhdistetään juurilistaan. Jos pois-



***Kuva** 3.3 `Delete`- ja `Decrease-key`-operaatiot F-keossa. Vaihe a) on lähtötilanne. Vaiheessa b) alkiolle 13 on suoritettu `Delete`-operaatio, jonka seurauksena 8 on merkitty. Vaiheessa c) alkion 9 prioriteetti on nostettu 7:ksi `Decrease-key`-operaatiolla. Koska 8 on jo merkitty, se siirretään juurisolmuksi. 4 on juurisolmu, joten sitä ei merkitä.*

tettava solmu on keon minimisolmu, operaatio muuntuu `Extract-min`-operaatioksi. Sekä `Decrease-key`- että `Delete`-operaatioissa ei-juurisolmun menettäessä yhden lapsisolmuistaan se merkitään. Jos lapsisolmu on jo merkitty, se irrotetaan samaan tapaan isäsolmustaan ja isäsolmu merkitään, jos se ei ole juurisolmu. Jos myös isäsolmu on jo merkitty, irrotus toistetaan sille. Jos merkitty juurisolmu päätyy liittämisvaiheessa toisen solmun lapseksi, merkki poistetaan. Kuvassa 3.3 keolle suoritetaan ensin `Delete` alkiolle 13 ja sitten `Decrease-key` alkiolle 9, missä sen prioriteetti nostetaan 7:ksi. Edellä kuvattujen irrotusten (engl. *cascading cut*) tarkoitus on rajata alhaalta puun kokoa suhteessa sen asteeseen. Näiden irrotusten vaikutusta tehokkuuteen käsitellään tarkemmin seuraavassa aliluvussa.

3.2 Tehokkuusanalyysi

Amortisoitujen aikavaatimusten[2] analysoimiseksi määritellään keon potentiaaliksi $n+2m$, missä n on sen sisältämien puiden määrä ja m on merkittyjen ei-juurisolmujen määrä. Operaation amortisoitu aika on ajoajan ja aiheutetun potentiaalin kasvun summa (kasvu voi olla myös negatiivista). Koska potentiaali on tyhjälle keolle 0 ja

[2]Amortisoitu analyysi tarkoittaa keskimääräisen aikavaatimuksen laskemista sarjalle operaatioita.

aina positiivinen, joukon operaatioita vaatima todellinen aika on aina korkeintaan sen amortisoitu aika.

Operaatiot `Make-heap`, `Find-min` ja `Merge` eivät vaikuta potentiaaliin ja ovat vakioaikaisia. Niin ikään vakioaikainen `Insert` kasvattaa puiden määrää, ja siten potentiaalia, yhdellä. `Extract-min`-operaatiossa pienimmän alkion poistaminen kasvattaa puiden määrää korkeintaan keon koon logaritmin verran. Sitä seuraava puiden yhdistäminen vie aikaa saman verran kuin potentiaali laskee, joten koko operaatio vie amortisoidusti logaritmisen ajan. `Decrease-key`-operaatiossa potentiaali kasvaa korkeintaan vakiomääräisesti:

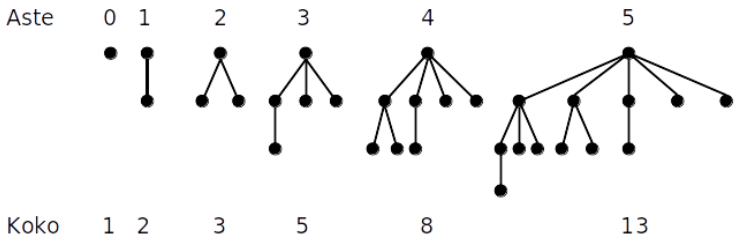
- Ensimmäinen irrotus muuttaa mahdollisesti merkitsemättömän ei-juurisolmun juurisolmuksi (potentiaali voi nousta yhdellä).
- Viimeinen irrotus voi merkitä ei-juurisolmun (potentiaali voi nousta kahdella).
- Jokainen irrotus ensimmäistä lukuun ottamatta siirtää merkityn ei-juurisolmun juurisolmuksi (potentiaali vähenee irrotusten määrän verran).

`Decrease-key` on siis amortisoidusti vakioaikainen. Samoin perustein myös `Delete` on ei-minimisolmulle amortisoidusti vakioaikainen. Poistettavan solmun ollessa minimisolmu se kuitenkin muuttuu `Extract-min`-operaatioksi, eli sen aikavaatimus kasvaa logaritmiseksi. `Build-heap`in voi toteuttaa sarjana vakioaikaisia `Insert`-operaatioita, jolloin sen aikavaatimus on $\Theta(n)$. Taulukkoon 3.1 on koottu amortisoidut aikavaatimukset F-keon operaatioille.

Taulukko 3.1 *F-keon amortisoidut aikavaatimukset [3, s. 157–159, 506].*

Operaatio	Aikavaatimus
Build-heap	$\Theta(n)$
Insert	$\Theta(1)$
Find-min	$\Theta(1)$
Extract-min	$O(\log n)$
Decrease-key	$\Theta(1)$
Merge	$\Theta(1)$
Delete	$O(\log n)$

Sarjassa operaatioita suoritettavien lapsisolmujen irrotusten määrä on korkeintaan siinä suoritettujen `Decrease-key`- ja `Delete`-operaatioiden määrä. Näiden irrotusten tarkoitus on rajata puun kokoa suhteessa sen asteeseen. Fredman ja Tarjan osoittavat [5], että k-asteisessa puussa on vähintään $F_{k+2} \geq \phi^k$ solmua, jossa F_k on



Kuva 3.4 *Pienimpiä mahdollisia puita Fibonacci-keossa [5].*

k:nnes Fibonaccin luku ja $\phi = (1 + \sqrt{5})/2$ on kultainen leikkaus. Kuvassa 3.4 esitetään pienimmät mahdolliset puut asteille 0–5. Siitä havaitaan myös, että koon suhde asteeseen vastaa Fibonaccin lukuja. Tästä myös juontuu Fibonacci-keon nimi.

Vaikka tämän luvun alussa todettiin F-keon tarvitsevan muistia neljälle osoittimelle solmua kohden, se on toteutettavissa myös kolmella tai jopa kahdella osoittimella suurempien vakiokertoimien kustannuksella suoritusajassa. [5]

F-keolla toteutetut verkko-optimointialgoritmit ovat asymptoottisesti aiempia ratkaisuja tehokkaampia graafeille, joissa kaarien määrä on huomattavasti suurempi kuin solmujen määrä ja huomattavasti pienempi kuin solmujen määrän neliö. Kevyimmän virittävän puun etsinnän asymptoottinen tehokkuus paranee myös graafeille, joiden tiheys on pieni. [5]

F-keosta on esitetty erilaisia muunnelmia tavoitteena saavuttaa samat aikavaatimukset yksinkertaisemmalla rakenteella tai ilman amortisoitujen aikavaatimusten aiheuttamaa ennakoimattomuutta [2, 5]. Tällainen yksinkertaisempi rakenne voisi mahdollistaa käytännöllisemmän toteutuksen.

4. KEKOJEN VERTAILU JA KÄYTTÖ ALGORITMEISSA

Binäärikeon ja F-keon aikavaatimukset esitetään taulukossa 4.1. Asymptoottisesti F-keko on jokaisessa operaatiossa vähintään yhtä tehokas tai tehokkaampi kuin binäärikeko. `Merge`-operaatiossa parannus lineaarisesta vakioaikaiseksi on näistä merkittävin.

Taulukko 4.1 *Kekojen aikavaatimukset [3, s. 506].*

Operaatio	Binäärikeko (huonoin tapaus)	Fibonacci-keko (amortisoitu)
Make-heap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(1)$
Find-min	$\Theta(1)$	$\Theta(1)$
Extract-min	$\Theta(\log n)$	$O(\log n)$
Merge	$\Theta(n)$	$\Theta(1)$
Decrease-key	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$O(\log n)$

F-keko on kuitenkin rakenteeltaan monimutkainen binäärikekoon verrattuna, joten asymptoottiset tehokkuusparannukset tulevat esiin vasta suurikokoisilla syötteillä. Monissa sovelluksissa `Extract-min`- ja `Delete`-operaatiota kutsutaan harvoin suhteessa muihin operaatiohin. Teoreettisesti F-kekoa käyttämällä voidaan tällöin saada parempia tehokkuuksia. Kuitenkin käytännössä suuret vakiokertoimet voivat kumota näin saadut edut. [3, s. 507]

4.1 Lyhimmän reitin etsiminen painotetussa graafissa

Suunnatussa ja painotetussa graafissa $G = (V, E)$, jossa V on solmujen ja E kaarien joukko, lyhin reitti yhdestä lähtösolmusta johonkin muuhun tai kaikkiin muihin solmuihin voidaan etsiä Dijkstran algoritmilla, jos kaikki painot ovat ei-negatiivisia[1]. Dijkstran algoritmi käy solmuja läpi etäisyysjärjestyksessä lähtösolmuun nähden.

[1]Graafeille, jonka kaarilla on negatiivisia painoja, voidaan käyttää Bellman–Ford-algoritmia [3, s. 651]

Löydetyt solmut lisätään minimiprioriteettijonoon Q, jossa prioriteettina käytetään niiden etäisyyttä lähtösolmusta. Prioriteettijonosta otettaessa solmu merkitään käsitellyksi (lyhin mahdollinen reitti siihen on silloin tiedossa), ja sen naapurisolmut lisätään prioriteettijonoon. Jos jo löydettyyn solmuun löydetään lyhempi reitti, voidaan toimia kahdella eri tavalla riippuen siitä, tukeeko käytettävä prioriteettijono `Decrease-key`-operaatiota. Jos operaatio on saatavilla, voidaan jonossa olevan solmun prioriteettia korottaa lyhemmän reitin mukaiseksi. Muutoin solmu lisätään uudestaan jonoon. Jälkimmäisessä tapauksessa äsitellyt solmut on merkittävä, jottei niitä käsitellä toistamiseen. Algoritmilistauksessa d on solmun etäisyys lähtösolmusta ja π sen isäsolmu.

Dijkstra(G, w, s) [3, s. 658]

```
1   input: G käsiteltävä graafi, w reitin paino, s lähtösolmu
2
3   for each vertex v ∈ G.V
4       v.d ← ∞
5       v.π ← NIL
6   s.d ← 0
7
8   S ← ∅
9   Q ← G.V
10  while Q ≠ ∅
11      u ← Extract-min(Q)
12      S ← S ∈ {u}
13      for each vertex v ∈ G.Adj[u]
14          if v.d > u.d + w(u, v)
15              v.d ← u.d + w(u, v)
16              v.π ← u
```

Binäärikeolla toteutettaessa while-silmukassa suoritetaan V kertaa logaritminen operaatio (`Extract-min`) sekä E/V kertaa logaritminen operaatio (`Decrease-key` tai `Insert`). Aikavaatimus binäärikekototeutukselle on siis $O((V+E)\log V)$. Tästä huomataan myös, ettei `Decrease-key`n saatavuudella ole vaikutusta asymptoottiseen tehokkuuteen tässä tapauksessa. Rintalan ja Valmarin tutkimuksessa [7] kuitenkin `Decrease-key`llä toteutettu algoritmi oli nopeampi verrattuna ilman sitä toteutettuun. F-keolla toteutettaessa `Decrease-key` on amortisoidusti vakioaikainen. Tällöin aikavaatimus pienenee $O(V \log V + E)$:ksi.

4.2 Kevyimmän virittävän puun etsiminen

Painotetun graafin $G = (V, E)$ kevyin virittävä puu on sen kaikki solmut sisältävä syklitön alipuu, jonka painojen summa on pienin mahdollinen. Kevyimmän virit-

tävän puun etsimiseen käytettyjä algoritmeja ovat muun muassa Primin algoritmi ja Kruskalin algoritmi. Molemmat näistä algoritmeista ovat ahneita algoritmeja [3, s. 625], eli ne valitsevat edetäkseen kulloinkin parhaalta näyttävän vaihtoehdon. Keskitymme tässä Primin algoritmiin, koska prioriteettijonolla on merkittävä osa sen toteutuksessa.

Primin algoritmi aloittaa kevyimmän virittävän puun rakentamisen mielivaltaisesta juurisolmusta. Puuhun yhdistetään siitä erillään olevista solmuista lähin. Tätä toistetaan, kunnes kaikki solmut on liitetty puuhun. Solmuja käsitellään minimiprioriteettijonossa Q, joka sisältää aina puusta erillään olevat solmut prioriteettinaan niiden etäisyys puuhun. Primin algoritmi on toiminnaltaan hyvin samankaltainen kuin Dijkstran algoritmi. Algoritmilistauksessa key on alkion prioriteetti ja π sen isäsolmu graafissa.

Prim(G, w, r) [3, s. 634]

```
1     input: G käsiteltävä graafi, w reitin paino, r juurisolmu
2
3     for each u ∈ G.V
4         u.key ← ∞
5         u.π ← NIL
6     r.key ← 0
7
8     Q ← G.V
9
10    while Q ≠ ∅
11        u ← Extract-min(Q)
12        for each v ∈ G.Adj[u]
13            if v ∈ Q and w(u, v) < v.key
14                v.π ← u
15                v.key ← w(u, v)
```

Binäärikeolla toteutettaessa algoritmissa suoritetaan V kertaa logaritminen `Extract-min` sekä E/V kertaa logaritminen `Decrease-key`. Aikavaatimus on siis $O(V \log V + E \log V) = O(E \log V)$. F-keolla toteutettaessa `Decrease-key` on vakioaikainen, jolloin aikavaatimus paranee $O(V \log V + E)$:ksi.

5. YHTEENVETO

Tässä työssä tutkittiin binääri- ja Fibonacci-kekoja prioriteettijonon toteuttamiseen. Työn tavoite oli selvittää kummankin tietorakenteen etuja ja haittoja sekä käytännöllisestä että teoreettisesta näkökulmasta. Binäärikeko on suosituin tietorakenne prioriteettijonon toteuttamiseen, mutta Fibonacci-keko tarjoaa keko-operaatiot asymptoottisesti tehokkaampina.

Binäärikeko on yksinkertainen ja tehokas tietorakenne. Operaatiot ovat hyvin yksinkertaisia toteuttaa, eivätkä vaadi ylimääräistä muistia. Se ei kuitenkaan ole asymptoottisesti tehokkain tietorakenne prioriteettijonoksi.

Toinen tutkittu tietorakenne, Fibonacci-keko, tarjoaa pienimmän alkion poistamista lukuun ottamatta prioriteettijono-operaatiot amortisoidusti vakioaikaisina. Se kuitenkin on rakenteellisesti hyvin monimutkainen binäärikekoon nähden. Aikavaatimusten saavuttamiseksi keolle on suoritettava sen muotoa korjaavia operaatioita, jotka hidastavat keko-operaatioita.

Vertailussa selvisi, että F-keko on kaikkiin keko-operaatioihin asymptoottisesti vähintään yhtä tehokas tai tehokkaampi kuin binäärikeko. Siten Dijkstran algoritmi ja Primin algoritmi ovat toteutettavissa F-keon avulla binäärikekoa tehokkaammin. F-keon monimutkaisuus voi kuitenkin käytännössä kumota nämä tehokkuusparannukset.

Nämä aikavaatimukset voisi kuitenkin olla mahdollista saavuttaa yksinkertaisemmalla tietorakenteella. Tulevaisuuden haasteeksi osoittautuukin siten samat aikavaatimukset täyttävän tehokkaamman tietorakenteen suunnittelu. Täten F-keon hyödyllisyys näkyy pääasiassa teoreettisella tasolla tienavaajana uusille tietorakenteille.

LÄHTEET

[1] T. Blechmann, "Boost.heap," Apr. 2017, viitattu: 20.8.2017. [Online]. Available: http://www.boost.org/doc/libs/1_64_0/doc/html/heap.html

[2] G. S. Brodal, G. Lagogiannis, and R. E. Tarjan, "Strict fibonacci heaps," in *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '12. New York, NY, USA: ACM, 2012, pp. 1177–1184. [Online]. Available: http://doi.acm.org/10.1145/2213977.2214082

[3] T. H. Cormen, R. L. Rivest, C. E. Leiserson, and C. Stein, *Introduction to algorithms*, 2nd ed. Cambridge (Mass.): MIT Press, 2001.

[4] Y. K. Dalal and R. M. Metcalfe, "Reverse path forwarding of broadcast packets," *Commun. ACM*, vol. 21, no. 12, pp. 1040–1048, Dec. 1978. [Online]. Available: http://doi.acm.org/10.1145/359657.359665

[5] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[6] P. Juszczak, D. M. Tax, E. Pękalska, and R. P. Duin, "Minimum spanning tree based one-class classifier," *Neurocomputing*, vol. 72, no. 7, pp. 1859–1869, 2009, advances in Machine Learning and Computational Intelligence. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231208003238

[7] M. Rintala and A. Valmari, "Priority queue classes with priority update," in *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST)*, ser. CEUR Workshop Proceedings, J. Nummenmaa, O. Sievi-Korte, and E. Mäkinen, Eds., no. 1525, Aachen, 2015, pp. 179–193. [Online]. Available: http://ceur-ws.org/Vol-1525/paper-13

[8] J. W. J. Williams, "Algorithms," *Commun. ACM*, vol. 7, no. 6, pp. 347–349, June 1964.