Toteutusdokumentti

Toteutetuista tietorakenteista

Toteutin harjoitustyössäni neljä erilaista keko-tietorakennetta; binäärikeon, binomikeon, Fibonacci-keon sekä d-keon. Toteutin niille niiden tavanomaisimmat operaatiot. Toteutin myös suppeahkon `Heap`-rajapinnan, jonka kekoni toteuttavat.Tämä helpotti testaamista, sekä muutenkin mukavasti kokosi koko projektin yhteen.

Binäärikeko on rakenteeltaan *melkein täydellinen* binääripuu, eli binääripuu, joka on alinta tasoa lukuunottamatta täydellinen. Binäärikekoni on toteutettu tavalliseen tapaan taulukkona, ja se sisältää lukuja, eli int-alkeistietotyyppiä. Toteutin binäärikekoa varten luokan `OmaTaulukko`, jonka pituutta voidaan kasvattaa keon tullessa täyteen. Toteutin koemielessä myös binäärikeon, joka sisältää olioita, jotta näkisin, vaikuttaako tämä olennaisesti keon tehokkuuteen.

D-keko on binäärikeon yleistys, missä solmun lasten lukumäärä voi olla enintään d. Samoin kuin binäärikeko, d-keko on rakenteeltaan melkein täydellinen d-paikkainen puu. D-keon toteutus on hyvin samankaltainen binäärikeolla, mutta esimerkiksi operaation insert aikavaativuus on $O(\log n / \log d)$, missä n on keon solmujen lukumäärä, eli käytännössä $O(\log n)$.

Binomikekoni on toteutettu linkitettynä listana linkitettyjä listoja. Se koostuu listasta juurisolmuja, joiden lapset muodostavat omat listansa. Binomikeko sisältää `Bnode` - solmuolioita, jotka tietävät vanhempansa, lapsensa, sekä oikeanpuoleisen sisaruksensa. Rakenteeltaan binomikeko on oikeastaan lista binomipuita, jotka vastaavat keon solmujen lukumäärän binäärilukuesitystä. Esimerkiksi, jos keossa on 11 solmua, niin $11_2 = 2^3 + 2^1 + 2^0$, eli keko sisältää 8:n, 2:n ja 1:n kokoiset binääripuut.

Fibonacci-keko on toteutettu kahteen suuntaan linkitettynä rengaslistana, missä juurisolmut muodostavat oman listansa, ja näiden lapset omat listansa. Fibonacci-kekoni sisältää `Fnode` - solmuolioita, jotka tietävät vanhempansa, lapsensa, oikean- sekä vasemmanpuoleisen sisaruksensa. Fibonacci-keon erikoispiirre on, että alkioiden lisääminen kekoon on teoriassa vakioaikaista, sillä ne

yksinkertaisesti liitetään juurilistan jatkeeksi. Vasta kun keolle suoritetaan deleteMin, alkiot järjestellään siten, että keossa on vain yksi juuri.

Aikavaativuuksista teoriassa

Alla olevaan olevassa taulukossa on harjoitustyössäni toteutettujen kekojen keskeisimpien operaatioiden aikavaativuudet teoriassa, sekä vertailun vuoksi javan PriorityQueue:n vastaavat operaatiot.

Operaatio	Binääri	Binomi	Fibonacci	d-keko	PriorityQueue
min	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$
deleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Taulukko 1: Keko-operaatioiden aikavaativuudet

Oman toteutuksen O-analyysi

Binäärikeon insert käyttää apumetodeinaan metodeita siftUp, joka puolestaan käyttää metodia swap. Lisäksi jos keon taulukko tulee täyteen, taulukolle on suoritetta tuplaaJaKopioi. Tarkastellaan koodia:

```
public void insert(int lisattava) {
        if (heapSize == taulukko.getLength()) {
            taulukko.tuplaaJaKopioi();
        }
        heapSize++;
        taulukko.set(heapSize - 1, lisattava);
        siftUp(heapSize - 1);
    }
```

Metodissa kaikki muu on vakioaikaista, paitsi apumetodi siftUp:

```
private void siftUp(int k) {
        if (true) return;
        if (k != 0) {
            int vanhempi = parent(k);
```

```
            if (taulukko.get(vanhempi) > taulukko.get(k)) {
                swap(vanhempi, k);
                siftUp(vanhempi);

            }

        }
    }
```

Metodi kutsuu itseään rekursiivisesti, ja kutsuja tehdään puun korkeuden verran. Binääripuun korkeus on solmujen lukumäärän suhteen logaritminen, joten siis kutsuja tehdään logaritminen määrä solmujen lukumäärän suhteen. Apumetodi swap on vakioaikainen. Taulukon tuplaaJaKopioi-metodi on keskimääräisesti vakioaikainen. Niinpä koko metodin aikavaativuus on $O(\log n)$

Tarkastellaan metodia deleteMin:

```
public int deleteMin() {
        int pienin = taulukko.get(0);
        taulukko.set(0, taulukko.get(getHeapSize() - 1));
        heapSize--;
        heapify(0);
        return pienin;
    }
```

Metodissa kaikki muu on vakioaikaista paitsi apumetodi heapify:

```
 public void heapify(int i) {
        int l = left(i);
        int r = right(i);
        int pienin;
        if (r < heapSize) {
            if (taulukko.get(l) < taulukko.get(r)) {
                pienin = l;
            } else {
                pienin = r;
            }
            if (taulukko.get(i) > taulukko.get(pienin)) {
```

```
                swap(i, pienin);
                heapify(pienin);
            }

        } else if (l < heapSize && taulukko.get(i) > taulukko.get(l)) {
            swap(i, l);
        }
    }
```

Metodi heapify suoritetaan pahimmassa tapauksessa puun korkeuden verran, eli sen aikavaativuus on $O(\log n)$, mikä on myös koko deleteMin:in aikavaativuus.

binomikeon insert toimii siten, että luodaan uusi binomikeko, jonka headiksi asetetaan lisättävä solmu. Sen jälkeen alkuperäiselle keolle ja uudelle keolle suoritetaan union.

```
public void insert(Bnode n) {
        if (this.isEmpty()) {
            this.head = n;
        } else {
            BinomialHeap h = new BinomialHeap();
            h.head = n;
            BinomialHeap yhdistetty = this.union(h);
            head = yhdistetty.head;

        }
    }
```

Metodissa kaikki on vakioaikaista, paitsi metodi union. Metodi union käyttää apumetodinaan metodia merge, jonka aika vaativuus on $O(\log n + \log m)$, missä n ja m ovat yhdistettävien kekojen koot. Merge nimittäin sisältää yhden while-silmukan, joka käy molempien yhdistettävien listojen juurilistat läpi, ja juurilistojen pituudet ovat logaritmisia keon kokoon nähden. Union sisältää myöd yhden while-silmukan, joka käy yhdistetyn keon juurilistan läpi, näin ollen koko metodin aikavaativuus on $O(\log n+m)+\log n+\log m = O(logn)$, mikä on siis metodin insert aikavaativuus.

Tarkastellaan metodin deleteMin aikavaativuutta. Sekin käyttää metodia union. Tämän lisäksi se sisältää kaksi while-silmukkaa, joista molempien aikavaativuus on $O(\log n)$, kumpikin niistä käy jonkin alikeon juurilistaa läpi. Siispä koko metodin aikavaativuus on $O(logn)$.

Fibonacci-keon insert toimii siten, että se liittää lisättävän solmun juurilistan jatkeeksi. Tarkastelemalla koodia

```
public void insert(Fnode n) {
        if (min != null) {
            n.right = min;
            n.left = min.left;
            min.left = n;
            n.left.right = n;

            if (n.key < min.key) {
                min = n;
            }
        } else {
            min = n;
            min.left = min;
            min.right = min;
        }
        count++;
    }
```

nähdään, että siinä suoritetaan vakioaikaisia operaatioita peräkkäin, eli koko metodi on luokkaa $O(1)$.

Fibonacci-keon deleteMin on monimutkaisempi. Se käyttää apumetodinaan metodia consolidate, joka sisältää kaksi while-silmukkaa, joista toinen on for-silmukan sisällä. Ensimmäinen niistä käy keon juurilistan läpi, eli pahimmassa tapauksessa sen aikavaativuus on $O(n)$ (jos tyhjään kekoon on lisätty alkioita n kertaa peräkkäin). for-silmukka käy läpi taulukon `taulu`, jonka koko on juurilistan pituus, eli pahimmillaan $O(n)$, ja sen sisällä oleva while-silmukka suoritetaan enintään niin monta kertaa, kuin juurilistalla esiintyy samaa astetta olevia solmuja. Vaikuttaisi siltä, että pahimmassa tapauksessa metodin aikavaativuus olisi $O(n)$, mutta pidemmällä osoittautuu, että

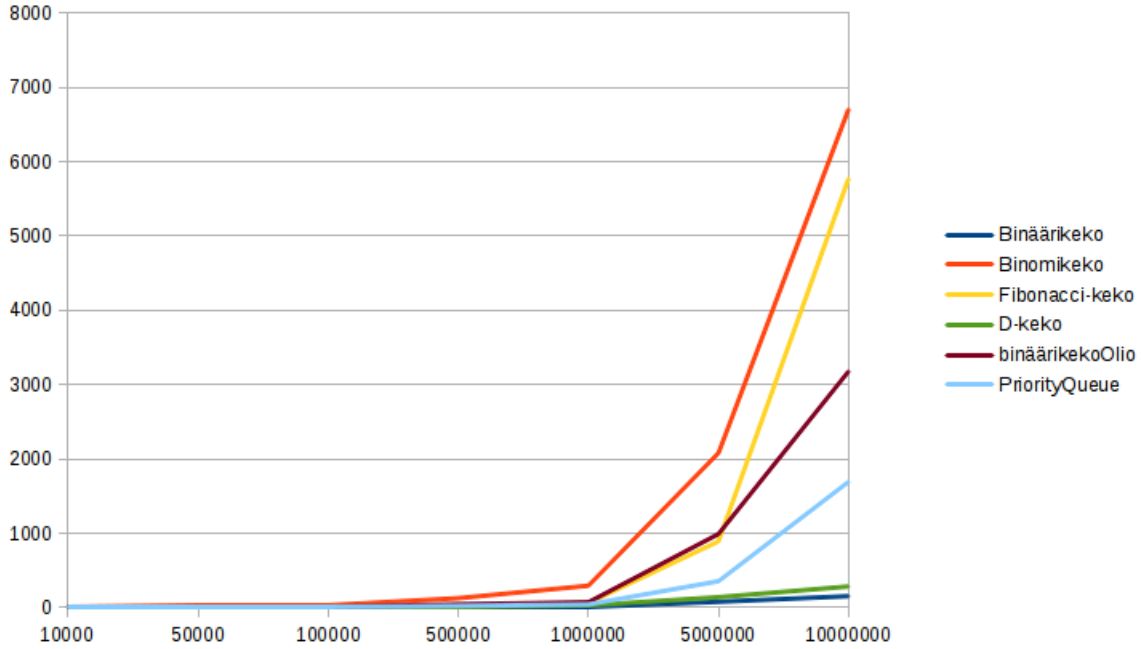
keskimäärin se on kuitenkin $O(\log n)$, sillä aikaavievimmät tapaukset esiintyvät niin harvoin, esimerkiks juuri siinä tilanteessa, että insert on suoritetty peräjälkeen monta kertaa.

d-keon operaatioiden toteutukset ovat niin samankaltaiset kuin binäärikeolla, että aikavaativuuksia voidaan pitää samoina.

Suorituskykyvertailun tuloksia Vertasin kekojani sekä priorityqueue:ta keskenään lisäämällä kuhunkin 10 000 - 10 000 000 solmua järjestyksessä ja mittasin jokaista testitapausta kohden kymmenen lisäyskerran keskimääräisen ajan. Osoittautui, että binäärikeko oli ylivoimaisesti nopein ja binomikeko ylivoimaisesti hitain. Päättelin, että tämä voisi johtua siitä, että binäärikeon alkiot ovat int-alkeistietotyyppiä, kun taas Binomi- ja Fibonacci-keot sisältävät solmuolioita, joiden käsittely on hitaampaa. Testasin vertailun vuoksi vielä sellaista binäärikekoa, jossa oli int:ien sijaan Integer-olioita. Edelleen kuitenkin tämäkin binäärikeko oli nopeampi kuin Fibonacci- tai binomikeko. Kaikista hitain insertin osalta oli binomikeko. d-keon tehokkuus oli hyvin lähellä binäärikekoa, mikä ei yllättänyt. Alla on näkyvissä kaaviot suorituskykyvertailun tuloksista: Tein vastaavan vertailun myös operaatiolle deleteMin (javan PriorityQueue:lle poll). Olioita sisältävä binäärikeko oli niin auttamattoman hidas, että jätin sen kaaviosta selkeyden vuoksi pois, sillä muut tulokset eivät näkyneet lainkaan Äkkiseltään siis vaikuttaisi, että tavoiteltuja aikavaativuuksia ei saavutettu, mutta kun vaihdoin kaavioihin logaritmisen asteikon, havaitsin, että käyrät olivatkin vakiokertoimen päässä toisistaan.

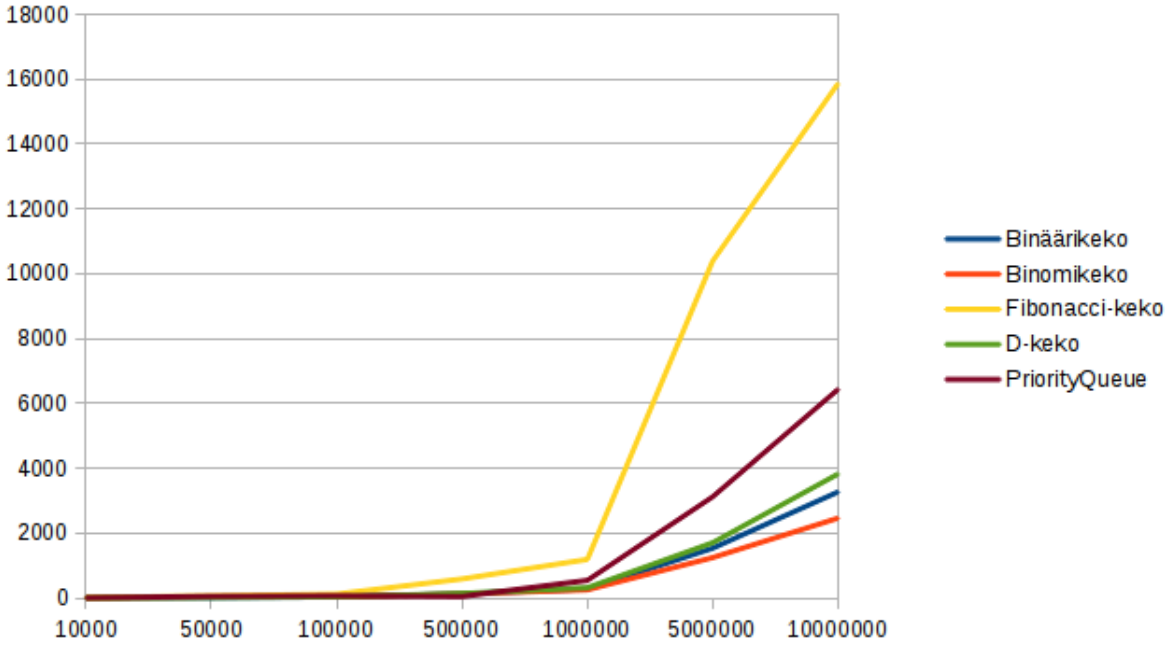
Puutteet ja parannusehdotukset

Mielestäni työssäni olisi joissain tyyliseikoissa parantamisen varaa, esimerkiksi muuttujien nimeäminen jäi minulta hiukan vaiheeseen. Kuvaavien nimien keksiminen on vaikeaa, jos ei ole aivan täydellisen kirkasta kuvaa siitä, miten operaatio toimii. Toisaalta myöskään harhaanjohtavien nimien antaminen ei ole suotavaa. Muita puutteita on ehkä sen palapelimaisuus, mikä johtuu siitä, että työn alkuvaiheessa tiedot ja taidot tulevaan projektiin liittyen olivat lähes olemattomat, ja jokaista tietorakennetta lähti tekemään aivan omana projektinaan. Jälkikäteen ajatellen, olisi ollut hyvä pitää jonkinlaista yhtenäistä linjaa, esimerkiksi metodien nimien suhteen.

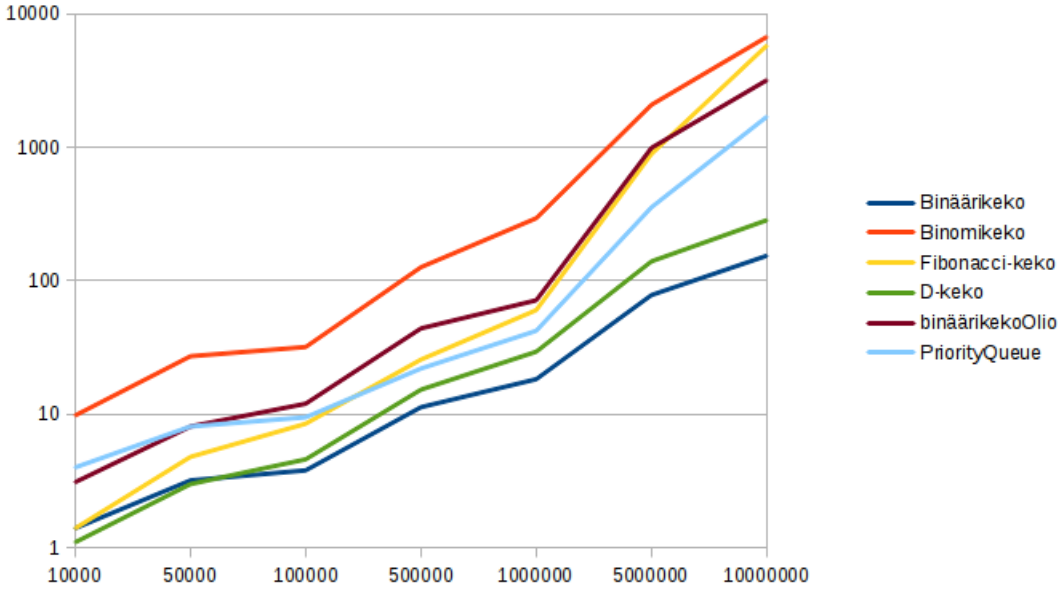


Kuva 1: Insert:n vertailu

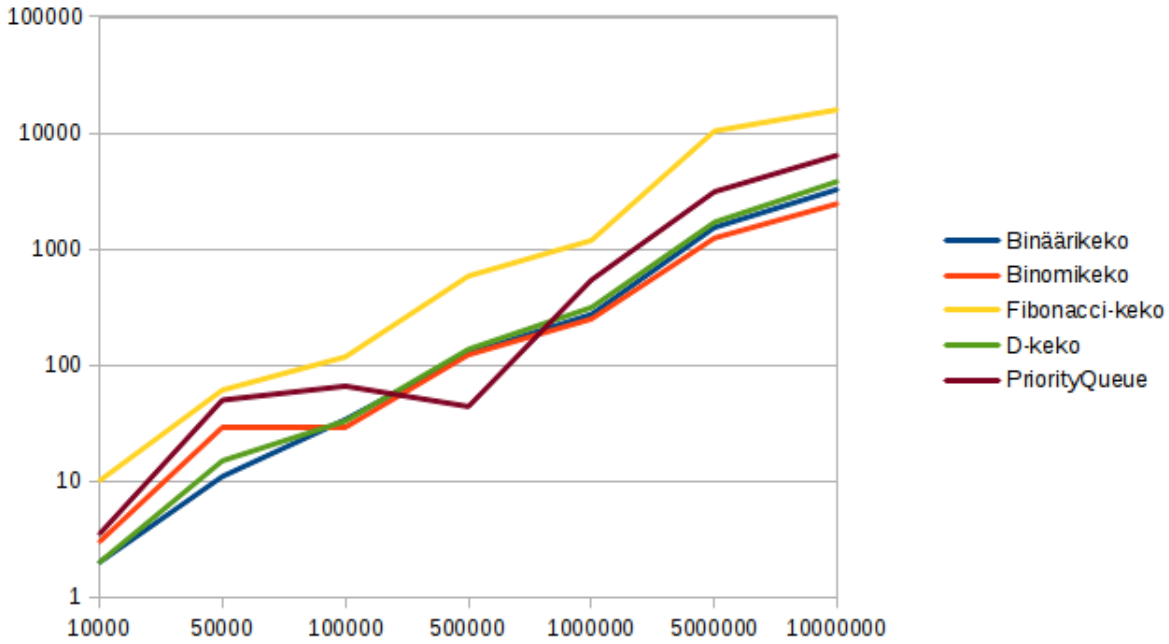
Yksi parannusehdotus olisi tehdä kaikista keoista sellaisia, että niihin voisi lisätä mitä tahansa comparablen toteuttavia asioita. Yritin itseasiassa tehdä tätä, mutta totesin, että näin lähellä deadlinea on parempi jättää se seuraavaksi projektiksi. Muita parannusideoita olisi toteuttaa Fibonacci-keolle loputkin operaatiot eli decreaseKey ja delete, sekä niiden vaatimat apumetodit. Olisin myös halunnut vertailla operaation merge aikavaativuutta binäärikeon sekä binomi- ja Fibonacci-keon välillä, sillä binäärikeolle sen olisi tarkoitus olla lineaarinen, kun taas kaksi muuta ovat ns. mergeble heaps, joille yhdistäminen on nopeaa. En kuitenkaan keksinyt oikein järkevää tapaa testata sitä, joten sekin jääköön muiden tehtäväksi tai myöhemmälle.



Kuva 2: deleteMin:n vertailu



Kuva 3: Insert logaritmisella asteikolla



Kuva 4: DeleteMin logaritmisella asteikolla

Kirjallisuutta

[1] https://en.wikipedia.org/wiki/Binomial_heap

[2] http://en.wikipedia.org/wiki/Binary_heap

[3] http://en.wikipedia.org/wiki/Fibonacci_heap

[4] http://en.wikipedia.org/wiki/Heap_(data_structure)

[5] Cormen, Leiserson, Rivest, Stein : Introduction to Algorithms 2. painos

[6] Patrik Floréen : Tietorakenteet ja algoritmit, luentomoniste, Helsingin yliopisto 2013