Totetutusdokumentti

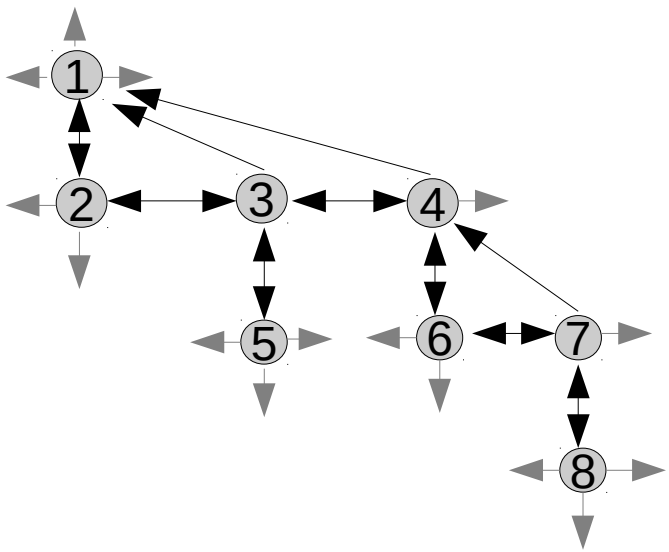
Riikka Huusari

Ohjelman yleisrakenne

Jokainen keko on oma luokkansa, ja sen lisäksi keot toteuttavat rajapinnan Keko, jonka kautta niitä käytetään kekojärjestämisessä ja Dijkstrassa.

Binäärikeko koostuu Solmu-tyyppisestä listasta, ja luvusta heapSize, joka kertoo keon koon. D-ary -keko on rakenteeltaan samanlainen kuin binäärikeko, joskin sillä on lisäksi luku d, joka kertoo solmun lasten maksimimäärä. Luokassa Solmu oleva ominaisuus indeksi viittaa siihen indeksiin, jossa Solmu binäärikeossa tai d-ary -keossa on. Tätä ominaisuutta käytän Dijkstran algoritmin apuna.

Fibonaccikeko ja Binomikeko käyttävät myös luokkaa Solmu apurakenteena, tosin paljon monipuolisemmin kuin edelliset luokat. Jokaisella Solmulla on pointteri edelliseen ja seuraavaan solmuun, kuten myös lapseen ja vanhempaan (tosin yleensä ainakin yksi näistä on null). Binomikeko ja Fibonaccikeko ovat siis linkitettyjä listoja. Molemmilla on Solmu keko, pointteri ensimmäiseen alkioon. Fibonaccikeossa on lisäksi suoraan pointteri minimiin, sekä solmujen lukumäärä (sitä käytetään minimin poistamisen jälkeen kun puita yhdistellään). Luokassa Solmu on ominaisuus marked, jota Binomikeko ei käytä. Se on osa Fibonaccikekoa ja sen decreaseKey -metodia jossa sitä käytetään pitämään kekoa matalana. Luokassa Solmu on myös ominaisuus node, jota käytetään Dijkstran algoritmin kanssa. Seuraavana havainnekuva siitä, miten solmut on linkitetty toisiinsa Binomikeossa (ja Fibonaccikeossa). Harmaat nuolet osoittavat null:iin.



Kekojärjestäminen toimii lisäämällä kaikki alkiot kekoon, ja poistamalla ne sieltä yksitellen. Dijkstran algoritmissa lasken etäisyydet lähtösolmusta kaikkiin solmuihin. En katsonut tarpeelliseksi pitää kirjaa reiteistä, sillä ohjelmani pääajatus on kekojen vertailu eikä reittien etsiminen. Reittikirjanpidon voisi kuitenkin tarvittaessa helposti lisätä.

O-analyysi

D-ary -keko ja **binäärikeko** ovat hyvin samanlaisia. Jälkimmäinen saadaan edellisestä asettamalla lapsien lukumäärän kahdeksi. Seuraavaksi käyn läpi d-ary -keon aika- ja tilavaativuudet. Binäärikeko tulee hoidettua siinä sivussa, sillä sen olen toteuttanut samalla tavalla. n tarkoittaa keon kokoa, ja d solmun lapsien lukumäärää.

Tilavaativuus:

Tallennan kekoni int-tyyppiseen taulukkoon, joka on aluksi pituudeltaan 20. Taulukko kasvaa keon mukana, ja on pahimmillaan kaksinkertainen keon kokoon nähden. Tilavaativuus on siis luokkaa O(n), missä n on keko suurimmillaan. Keon koon ilmaiseva muuttuja ei tähän vaikuta, ja kun nodetaulukko on mukana, tilavaativuus on kaksinkertainen edelliseen nähden; se ei myöskään vaikuta vaativuusluokkaan.

Operaatioiden aikavaativuudet:

Apumetodi heapifyWithNode:

```
     heapifyWithNode(int key, boolean onNode) {
1       for (key has lapsi l) {
2           lapset.add(l);
3       }
4       indeksi = etsiPienin(lapset);
5       if (indeksi != -1 && keko[indeksi] < keko[key]){
6           vaihda(keko[indeksi],keko[key]);
7           if (onNode) {
8               vaihda(node[indeksi],node[key]);
9               heapifyWithNode(indeksi, true);
10          } else {
11              heapifyWithNode(indeksi, false);
12          }
13      }
```

Metodi etsiPienin käy kaikki lapset läpi ja palauttaa pienimmän indeksin. Metodin alussa käydään siis solmun lapset, joita on d kappaletta, läpi kaksi kertaa. Seuraavaksi, jos pienimmän lapsen arvo on pienempi kuin nykyisen solmun arvo, niiden paikkoja vaihdetaan ja kutsutaan metodia uudestaan. Tätä tapahtuu korkeintaan keon korkeuden verran, ja keon korkeus on log n/log d. Siispä koko operaation aikavaativuudeksi tulee O(d * (log n/log d)). Taaskaan nodetaulukko ei vaikuta aikavaativuusluokkaan.

findMin:
Tämä tapahtuu vakioajassa, sillä pienin alkio on aina keon huipulla.

deleteMin:
Tässä metodissa vaihdetaan ensin (vakioajassa) keon ensimmäisen ja viimeisen alkion paikat, pienennetään keon kokoa, ja kutsutaan heapify:tä. Siispä O(d * (log n/log d)).

decreaseKey:
Tämä metodi toimii kuten heapify siinä mielessä että se vaihtaa lapsen ja vanhemman arvot niin

kauan kunnes kekoehto ei ole enää rikki. Sen ei tarvitse kuitenkaan etsiä oikeaa kohtaa josta se vaihtaisi arvon; solmulla on vain yksi vanhempi. Siispä aikavaativuus on vain O(log n/log d).

insert:
Insert kasvattaa taulukon kaksinkertaiseksi jos se täyttyy. Tämä operaatio tapahtuu kuitenkin hyvin harvoin, joten sen vaikutus käytännössä tasoittuu olemattomiin. Itse insert tapahtuu kasvattamalla muuttujaa heapSize ja asettamalla keon viimeiseksi uuden arvon. Tämän jälkeen kekoehto korjataan kuten metodissa decreaseKey. Siispä aikavaativuusjälleen O(log n/log d).

Fibonaccikeossa tilavaativuus riippuu täysin keon solmujen lukumäärästä, sillä keko muodostetaan linkitettynä listana solmuja ja niitä ei ole koskaan ylimääräisiä. (Java huolehtii roskista.)
Tilavaativuus on siis luokkaa O(n).

Aikavaativuudet:

findMin:
Tämä hoituu vakioajassa, sillä minulla on pointteri pienimpään solmuun.

insert:

```
  insert(Solmu s) {
1         if (keko == null) {
2             keko = s;
3             min = s;
4         } else {
5             keko.setSeuraava(s);
6            //päivitetään muutkin pointterit
7             if (s.getArvo() < min.getArvo()) {
8                 min = s;
9             }
10        }
11        solmuja++;
12    }
```

Selvästi nähdään, että lisäys tapahtuu vakioajassa.

DeleteMin
Tämä operaatio toimii kolmessa osassa: ensin poistetaan minimialkio ja lisätään sen lapset listaan. Minimialkio löydetään vakioajassa. Tämän jälkeen lisätään lapset listaan, ja joudutaan käymään kaikki lapset läpi jotta pointterit voidaan päivittää. Tämä vaihe kestää siis O(d)=O(log n).

Seuraavaksi yhdistetään keon puut niin, ettei tämän jälkeen ole kahta samanasteista puuta. Apurakenteena on Solmu-taulukko, jossa on linkit indeksin asteisiin solmuihin. Jokainen keon alkio käydään läpi ja tarvittaessa yhdistetään se samanasteisen jo löydetyn alkion kanssa ja lisätään uudestaan läpikäytäviin solmuihin. Jos juurisolmuja on ennen operaatiota m kappaletta, kestää tämä vaihe O(log n +m). Tasoitetulla analyysillä kuitenkin saadaan tämä vaihe tippumaan O(log n):n (http://en.wikipedia.org/wiki/Fibonacci_heap).

Viimeisenä päivitetään minimi-pointteri, eli käydään kaikki juurisolmut läpi. Tämä kestää O(log n), sillä keko on juuri tiivistetty ja juurisolmuja on korkeintaan tuo log n.

Kaikenkaikkiaan operaation aikavaativuus on O(log n).

decreaseKey

```
 decreaseKey(Solmu s, int d) {
1         if (d < s.getArvo() && d > 0) {
```

```
2            s.setArvo(d);
3            //jos kekoehto rikkoutuu
4            if (s.getParent() != null && s.getParent().getArvo() > d) {
5                //irroitetaan solmu vanhemmastaan ja lisätään kekoon
6                Solmu p = leikkaaJaLiita(s);
7                //jos parent ei ole merkattu, se merkataan. jos se on
8                //merkattu, sekin leikataan pois
9                boolean jatkuu = true;
10               while (p != null && jatkuu) {
11                   if (p.getParent() == null) {
12                       jatkuu = false;
13                   } else if (p.eiOleMerkitty) {
14                       p.mark();
15                       jatkuu = false;
16                   } else if (p.onMerkitty) {
17                       p = leikkaaJaLiita(p);
18                   }
19               }
20           }
```

decreaseKey leikkaa solmuja puusta pois, ja lisää niitä kekoon uusiksi juurisolmuiksi, kunnes vastaan tuleva solmu ei ole marked. Siis operaation aikavaativuus on O(k), missä k on uusien puiden määrä. Tasoitetussa analyysissä aikavaativuus on kuitenkin vakio.

Kuten fibonaccikeossa, myös **binomikeossa** tilavaativuus on O(n), eli riippuu suoraan solmujen lukumäärästä.

Aikavaativuudet:

apumetodi merge:
Merge tavallaan laskee yhteen kaksi kekoa. Se aloittaa kummankin keon läpikäymisen pienimmästä päästä. Jos toisen keon senhetkisen juurisolmun aste on pienempi kuin toisen, lisätään tuo pienempi uuteen kekoon, ja siirrytään käsittelemään seuraavaa. Jos taas molemmilla on sama aste, lisätään toinen toisen lapseksi, ja lisätään uusi puu uuteen kekoon. Lisäksi kokoajan tarkkaillaan, onko lisättävä puu samanasteinen kuin uudessa keossa viimeisenä oleva puu. Jos puut ovat samanasteisia, ne yhdistetään. Kun keot ovat n:n pituisia (tai pidempi keko on), on aikavaativuus operaatiolla luokkaa O(log n).

findMin:
Metodi käy läpi kaikki juurisolmut ja palauttaa niistä pienimmän, siis vaativuus on O(log n).

deleteMin:
Metodi etsii ensin pienimmän alkion, poistaa sen ja muodostaa uuden keon sen lapsista. Sen jälkeen uusi keko ja alkuperäinen yhdistetään operaatiolla merge. Aikavaativuus on selvästi O(log n).

Insert:
Tässä metodissa luodaan uusi yhden alkion kokoinen keko ja yhdisteään se lisättävään kekoon operaatiolla merge. Aikavaativuus myös tässä O(log n).

decreaseKey:
DecreaseKey toimii vähän kuten binäärikeossa, ja on vaativuudeltaan O(log n). Se vaihtaa solmun arvon, ja vaihtaa sen jälkeen niin kauan arvoja parentinsa kanssa, kunnes kekoehto on jälleen voimassa.

Puutteet ja parannusehdotukset

Fibonaccikekoon jäi jokin bugi, jota en pitkänkään etsimisen jälkeen onnistunut paikantamaan. Käsittääkseni se laittaa aina välillä Dijkstran algoritmin yhteydessä jonkin Solmun edelliseksi tai seuraavaksi Solmuksi Solmun itsensä, jolloin ohjelman suoritus jää junnaamaan paikoilleen. Jostain syystä tämä bugi ei kuitenkaan vaikuta mitenkään kekojärjestämiseen, eivätkä JUnit-testinikään sitä huomaa. Ihan pikkuisilla verkoilla virhe tapahtuu harvoin, mutta mitä isompi verkko on, sitä todennäköisemmin virhe tapahtuu.