Lause 5.1.1. *Epätyhjässä täydellisessä binääripuussa on 2^d solmua syvyydellä d.*

Todistus. Induktiolla yli syvyyden $0 \leq d \leq$ korkeus:

- Syvyydellä $d = 0$ on vain juuri, ja niitä on vain $1 = 2^0$ kappaletta.

- Syvyys $d+1$ kasvaa edellisestä syvyydestä d:

  Aitouden nojalla jokaisella syvyyden d solmulla on joko 0 tai 2 lasta.

  Täyteyden nojalla jokainen syvyyden d solmu on sisäsolmu.

  Siis jokaisella syvyyden d solmulla on 2 lasta.

  Induktion nojalla syvyydellä d on yhteensä 2^d solmua.

  Siis syvyydellä $d+1$ on näiden 2^d solmun $2 \cdot 2^d = 2^{d+1}$ lasta. □

Lause 5.1.2. *Epätyhjän täydellisen binääripuun*

$$\textit{korkeus} = \log_2(\textit{lehtien lukumäärä}).$$

Todistus. Lauseen 5.1.1 nojalla

$$\begin{aligned}\text{lehtien lukumäärä} &= 2^{\text{lehtien syvyys}} \\ &= 2^{\text{puun korkeus}}\end{aligned}$$

josta saadaan

$$\log_2(\text{lehtien lukumäärä}) = \text{puun korkeus}$$

ottamalla logaritmit molemmilta puolilta kuten kalvoilla 3.5. □

- Rekursiokutsupuun

  solmussa on luku

  $$c \cdot \frac{n}{2^d} \tag{4}$$

  missä

  - d on solmun syvyys
  - n on koko algoritmin saaman syötelistan pituus

  koska jokainen kutsu puolittaa oman syötteensä 2 lapselleen.

  lehdessä tämä luku on

  $$c \cdot 1$$

  koska rekursio pysähtyy kun ei enää voi puolittaa.

  Tässä c on vakio kaavasta (3).

- Halutaan siis lauseke *kaikkien puussa olevien lukujen summa*lle suureen n suhteen.

- Lasketaan summa *syvyystaso kerrallaan*.

- Syvyystasolla d olevien lukujen summa on

  $$\underbrace{\text{yhtälö (4)}}_{\text{luku solmussa}} \cdot \underbrace{\text{lause 5.1.1}}_{\text{solmujen määrä}} = c \cdot n. \tag{5}$$

- Lehtiä on n kappaletta, koska jokainen lajiteltava alkio päätyy omaan lehteensä.

  Silloin syvyystasoja $0 \leq d \leq$ korkeus on

  $$\log_2(n) + 1 \tag{6}$$

  kappaletta lauseen 5.1.2 nojalla.

- Kaikkien puussa olevien lukujen summa on

  $$\text{yhtälö (5)} \cdot \text{yhtälö (6)} = \mathcal{O}(n \cdot \log(n)).$$

- Toisaalta jokaisessa rekursiokutsussa kuluu

$$\Omega(|L|)$$

askelta jo pelkkään aineiston jakamiseen kalvoilta 5.1.1.

- Samalla rekursiopuupäättelyllä nähdään siitä kokonaisajaksi

$$\Omega(n \cdot \log(n))$$

askelta.

- Siis lomitusjärjestämisen kokonaisajantarve on

$$\Theta(n \cdot \log(n))$$

askelta.

5.1.4 Lomitusjärjestämisen tilantarve

- Jos algoritmi saa syötteensä listana, jonka osoittimia saa muokata, niin sen apurutiineille kalvoilta 5.1.1 ja 5.1.2 riittää

$$\mathcal{O}(1)$$

muistipaikkaa aputilaksi.

- Muuten algoritmi joutuu tekemään työ*kopion* (viimeistään) kalvojen 5.1.2 yhdistämisvaiheessa.

  Kopioon tarvitaan

$$\mathcal{O}(n)$$

muistipaikkaa.

- Rekursiivinen algoritmi tarvitsee rekursiokutsupinosta muistipaikkoja *sisäkkäisten rekursiokutsujen lukumäärän* verran.

- Tämä lukumäärä on kalvojen 5.1.3.2

$$\textit{rekursiokutsupuun korkeus} + 1$$

- Tämä suure laskettiin lomitusjärjestämisen rekursiokutsupuulle yhtälössä (6):

$$\log_2(n) + 1.$$

- Siis lomitusjärjestäminen tarvitsee

$$\Theta(\log(n))$$

muistipaikkaa rekursiokutsupinosta.

5.2 Pikajärjestäminen

- *Pikajärjestäminen* (quicksort) lienee *käytännössä tehokkain* yleinen järjestämisalgoritmi.

- Hyvä menetelmä jos *aineisto tulee taulukkona joka pitää järjestää.*

  – Taulukko voidaan lajitella *paikallaan* (in place, in situ).

    Eli taulukon lisäksi tarvitaan vain rekursiokutsupinosta tilaa.

  – Muutenkin kilpailukykyinen menetelmä.

- Ei suoritusaika*takuu*ta:

  Normaalisti toimii $\Theta(n \cdot \log(n))$ askeleessa.

  Pahimmillaan voi viedä $\mathcal{O}(n^2)$ askelta.

- Pikajärjestäminen saadaan kalvojen 2.1 hajoita ja hallitse -periaatteella. (Vertaa kalvojen 5.1 lomitusjärjestämiseen.)

- Syötetaulukon $A[p \ldots r]$ jako osiin:
  - Vaihdellaan sen alkioita keskenään
  - kunnes on olemassa sellainen indeksi $p \leq q \leq r$ jolla pätee:
  - alkuosan $A[p \ldots q-1]$ alkiot ovat *korkeintaan* yhtä suuria kuin *jako-* eli *napa-alkio* (pivot) $A[q]$
  - loppuosan $A[q+1 \ldots r]$ alkiot ovat *vähintään* yhtä suuria kuin $A[q]$.

  Sitten lajitellaan alku- ja loppuosa.

- Osatuloksia ei tarvitse yhdistellä:
  - $A[q]$ on jo oikealla paikallaan
  - alkuosa järjestyy paikallaan, samoin loppuosa.

  $\mathcal{O}$-vakio lienee siis pieni.

procedure quicksort(A: **taulukkoviite**, p, r: $\mathbb{N}$)
  if $p < r$ **then**
    q := partition(A, p, r);
    quicksort($A, p, q-1$);
    quicksort($A, q+1, r$)
  end if.

function partition(A: **taulukkoviite**, p, r: $\mathbb{N}$): $\mathbb{N}$
1: Valitse jollakin strategialla jokin alkioista $A[p \ldots r]$ jakoalkioksi x;
2: Vaihtele alkioita $A[p \ldots r]$ siten, että jaolta vaaditut ehdot tulevat voimaan;
3: **return** se indeksi q johon rivillä 1 valittu jakoalkio x päätyi rivin 2 vaihtojen seurauksena.

- Rivillä 1 kannattaisi valita sellainen x jonka q on mahdollisimman *keskellä* taulukkoa $A[p \ldots r]$.

- Mutta funktio partition saa viedä vain

$$\mathcal{O}(r-p) \tag{7}$$

  askelta.

5.2.1 Aineiston partitiointi

- Kehitetään eräs sellainen funktio partition(A, p, r) joka toteuttaa yhtälön (7).

- Jaetaan syöte $A[p \ldots r]$ 4 osaan:
  1. Osan $A[p \ldots i]$ alkiot ovat $\leq x$.
  2. Osan $A[i+1 \ldots j-1]$ alkiot ovat $> x$.
  3. Osan $A[j \ldots r-1]$ alkioita ei ole vielä käsitelty.
  4. Osa $A[r] = x$ on jakoalkio. Se ei muutu.

- Tämä 4-jako olkoon silmukkainvariantti.

- Konvergentti olkoon:

  Joka kierroksella käsitellään osan 3 seuraava alkio $A[j]$ osaan 1 tai 2.

function partition(A: **taulukkoviite**, p, r: $\mathbb{N}$): $\mathbb{N}$
1: x := $A[r]$;
2: i := $p-1$;
3: **for all** j := p **to** $r-1$ **do**
4: **if** $A[j] \leq x$ **then**
5: i := $i+1$;
6: Vaihda alkiot $A[i]$ ja $A[j]$ keskenään
7: **end if**
8: **end for;**
9: Vaihda alkiot $A[i+1]$ ja $A[r]$ keskenään;
10: **return** $i+1$.

- Voitaisiin parantaa jakoalkion x valintastrategiaa:

  Viimeisen valinta on huono strategia:

  Jos A on jo valmiiksi lajiteltu?

  Jakoalkio laidassa on hyvä ohjelmoinnissa:

  Se ei ole muiden alkioiden siirtelyssä tiellä.

  Hyvä strategia voi aloittaa vaihtamalla valitsemansa paremman jakoalkion taulukon laitaan.

- Ositus 3 osaan

  $\langle s, t\rangle$:= partition3(A, p, r);

  missä

  1. osa $A[p \ldots s-1]$ on $< x$
  2. osa $A[s \ldots t]$ on $= x$
  3. osa $A[t+1 \ldots r]$ on $> x$

  välttäisi rekursion keskiosalla 2.

5.2.2 Pikajärjestäminen parhaimmillaan

- Tarkastellaan pikajärjestämisen aikavaatimusta kun kalvojen 5.2.1 partitiointi onnistuu *aina jakamaan aineiston tasan*.

- Rajoitutaan syötepituuksiin $n = 2^k - 1$:
  - Jakoalkio x jää pois rekursiosta.
  - Rekursioihin jää yhteensä
    $$n - 1 = 2^k - 2 = 2 \cdot (2^{k-1} - 1)$$
    muuta alkiota.
  - Yhteen rekursioon jää niistä puolet eli
    $$2^{k-1} - 1 = \frac{n-1}{2}$$
    alkiota.

procedure quicktime(n: **syötepituus** $r - p + 1$)
  if $n > 1$ **then**
    $2 \cdot$ quicktime $\left(\frac{n-1}{2}\right) + \mathcal{O}(n)$
  end if.

$$U(n) = \begin{cases} \mathcal{O}(1) & \text{kun } n = 1 \\ 2 \cdot U\left(\frac{n-1}{2}\right) + \mathcal{O}(n) & \text{kun } n > 1 \end{cases}$$

- Vertaa lomitusjärjestämisen lausekkeeseen $T(n)$ kalvoilta 5.1.3.

- Siitä arvaus: Onko myös
  $$U(n) = \mathcal{O}(n \cdot \log(n))?$$

- Sijoitetaan arvaus kaavaan ja sievennetään:

  Kyllä on!

- Tilantarve saadaan kalvojen 5.1.4 tapaan:

  $\Theta(\log(n))$.

5.2.3 Pikajärjestäminen pahimmillaan

- Tarkastellaan pikajärjestämisen ajantarvetta kun *partitiointi jättää aina toisen osista tyhjäksi*.

- Kalvojen 5.2.1 menetelmällä näin käy kun
  - jakoalkion arvo $x = A[r]$ on aina osataulukon $A[p \ldots r]$ *suurin*
  - eli esimerkiksi kun koko aineisto A on jo alun perin oikeassa *järjestyksessä*.

- Kun partitioidaan n alkiota, niin se vie(nee) $\Omega(n)$ askelta.

- Silloin tämä erikoistapaus vie $\Omega(n^2)$ askelta.

procedure slowtime(n: **syötepituus** $r-p+1$)
  if $n > 1$ **then**
    slowtime$(n-1) + \Omega(n)$
  end if.

$$V(n) = \begin{cases} \Omega(1) & \text{kun } n = 1 \\ V(n-1) + \Omega(n) & \text{kun } n < 1 \end{cases}$$
$$= \sum_{1 \le m \le n} \Omega(m)$$
$$= \Omega(n^2)$$

kalvojen 3.4 tapaan.

- Osoitetaan vielä että myös aivan *pahin mahdollinen* tapaus on ajantarpeeltaan tätä samaa luokkaa

$$\mathcal{O}(n^2).$$

procedure worsttime(n: **syötepituus** $r-p+1$)
  if $n > 1$ **then**
    $\max_{1 \le m \le n-1}$(worsttime$(n-m)$ +
    worsttime$(m-1)) + \mathcal{O}(n)$
  end if.

$$W(n) = \begin{cases} \mathcal{O}(1) & \text{kun } n \le 1 \\ \max_{1 \le m < n}(W(n-m) & \\ \qquad + W(m-1)) & \\ \quad + \mathcal{O}(n) & \text{kun } n > 1. \end{cases}$$

- Arvataan siis että

$$W(n) = \mathcal{O}(n^2) \le c \cdot n^2$$

  jollakin vakiolla $c > 0$.

- Todennetaan arvaus sijoittamalla se lausekkeeseen.

- Tarvitaan funktion

$$f(m) = c \cdot (n-m)^2 + c \cdot (m-1)^2$$

  suurin arvo välillä $1 \le m \le n-1$.

- Funktio $f(m)$ on ylöspäin aukeava paraabeli.

- Siis se saa suurimman arvonsa jommassa kummassa päätepisteessä:

$$f(1) = c \cdot (n-1)^2$$
$$f(n-1) = c \cdot (1 + (n-2)^2)$$
$$f(1) > f(n-1).$$

- Nyt arvaus on todennettu jos

$$W(n) \le c \cdot (n-1)^2 + \mathcal{O}(n)$$
$$= c \cdot n^2 - c \cdot (2 \cdot n - 1) + \mathcal{O}(n)$$
$$\le c \cdot n^2.$$

- Tämä onnistuu valitsemalla niin suuri c että on suurempi kuin termissä $\mathcal{O}(n)$ piileksivä $c' \cdot n$.

- Siis pikajärjestämisen aikatehokkuus *voi romahtaa* yksinkertaisten algoritmien (kuten kalvojen 3.4 lisäyslajittelun) tasolle.

5.2.4 Pikajärjestäminen tyypillisesti

- Ajantarve on myös *keskimäärin* sama

$$\mathcal{O}(n \cdot \log(n))$$

  kuin kalvojen 5.2.2 paras tapaus.

- Kääntäen: kalvojen 5.2.3 pahin tapaus

$$\mathcal{O}(n^2)$$

  on harvinainen.

- Todennäköisyyslaskelmat ohitetaan.

- Syitä hyvään käytökseen:
  - huono jako + hyvä jako $\approx$ hyvä jako.
  - Mikään vakiojako*suhde* p ei ole huono:
    * Jos n alkion aineisto jakautuu aina $p \cdot n$ ja $(1-p) \cdot n$ alkion osiin
    * niin pikajärjestäminen vie edelleen ajan $\mathcal{O}(n \cdot \log(n))$.

5.2.5 Pikajärjestämisen tilantarve

- Ei vaadi aputilaa jos syöteaineiston A tallennusrakennetta saa käyttää.

- Perusversio vaatii rekursiopinotilaa

  vähintään $\Omega(\log(n))$ kalvojen 5.2.2

  enintään $\mathcal{O}(n)$ kalvojen 5.2.3

  esimerkillä.

- Voidaan viilata tarvitsemaan

  $$\mathcal{O}(\log(n))$$

  pinopaikkaa:

  1. Ensin se *normaali* rekursiokutsu joka saa *lyhyemmän* osataulukon.
  2. Sitten *pitempi* kalvojen 4.6 *taka*rekursiivisena kutsuna eli toistona.

5.2.6 Jakoalkion valintastrategioita

Kolmesta keskimmäinen (median of three): valitse keskimmäinen arvoista

vasen laita laita $A[p]$

oikea laita $A[q]$

keskipiste $A\left[\frac{q-p+1}{2}\right]$.

- Ehkäisee kalvojen 5.2.3 ongelman valmiiksi järjestetyllä aidosti kasvavalla A.

Satunnainen: poimi satunnaislukugeneraattorilla indeksi $p \leq q \leq r$.

- Tarvitsee generaattoria.
- Ohjaa kohti kalvojen 5.2.4 hyvää keskimäräistä käyttäytymistä.

Edellisten yhdistelmä: poimi satunnaislukugeneraattorilla 3 indeksiä $p \leq i, j, k \leq r$ ja valitse keskimmäinen arvoista $A[i], A[j], A[k]$.

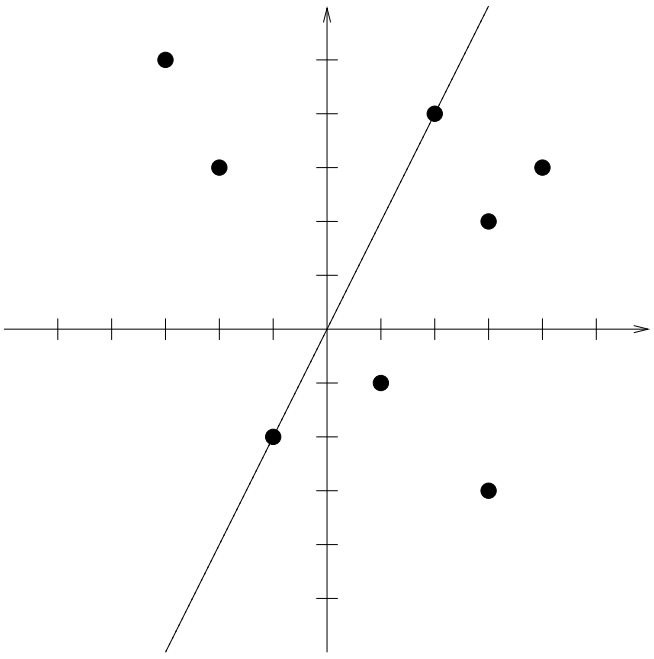
5.3 Järjestäminen algoritmin osana

- Tehokas järjestäminen on usein käyttökelpoinen *aliohjelmana* muiden ongelmien ratkaisualgoritmeissa.

- Erityisesti (muokatun) *syötteen* järjestäminen etukäteen helpottaa usein lopputyötä.

- Esimerkkinä seuraava *laskennallisen geometrian* (computational geometry) ongelma:

  - Syötteenä annetaan pisteet $p_1, p_2, p_3, \ldots, p_n$ koordinaatistossa kokonaislukupareina $p_i = \langle p_i.x, p_i.y \rangle \in \mathbb{Z} \times \mathbb{Z}$.
  - Kysytään ovatko mitkään kaksi pistettä samalla origon $\langle 0, 0 \rangle$ kautta kulkevalla suoralla.
  - Oletetaan yksinkertaistuksena että jokainen $x_i \neq 0$.



- Origon ja pisteen p kautta kulkevan suoran yhtälö on

$$y = \text{kulmakerroin}(p) \cdot x$$

missä

$$\text{kulmakerroin}(p) = \frac{p.y}{p.x}$$

joten pisteet p_i ja p_j ovat samalla origon kautta kulkevalla suoralla täsmälleen kun

$$\text{kulmakerroin}(p_i) = \text{kulmakerroin}(p_j).$$

- Ongelma saa siis muodon:

"Toistuuko luvuissa

$$\begin{aligned} K = \text{kulmakerroin}(p_1),\\ \text{kulmakerroin}(p_2),\\ \text{kulmakerroin}(p_3),\\ \vdots\\ \text{kulmakerroin}(p_n) \end{aligned}$$

mikään arvo?"

Muunnos vie

$$\mathcal{O}(n)$$

askelta (emme laske bittejä kalvojen 3.5 tapaan).

Perusmenetelmä tekee jokaiselle luvulle vertailut

$$\begin{aligned} \text{kulmakerroin}(p_i) \text{ vs } & \text{kulmakerroin}(p_{i+1}),\\ & \text{kulmakerroin}(p_{i+2}),\\ & \text{kulmakerroin}(p_{i+3}),\\ & \vdots\\ & \text{kulmakerroin}(p_n) \end{aligned}$$

sen jälkeisiin lukuihin.

Vie

$$\mathcal{O}(n^2)$$

askelta.

Nopeampi menetelmä on:

1. Ensin lajittele (nopeasti) luvut K.
2. Sitten katso toistuuko lajitelluissa luvuissa sama arvo 2 kertaa peräkkäin.

Vie vain

$$\mathcal{O}(n \cdot \log(n) + n) = \mathcal{O}(n \cdot \log(n))$$

askelta.

- Ratkaisussa on vielä ongelma:

$$\text{kulmakerroin}(p_i) \in \mathbb{Q}$$

joten eikö $\mathbb{Z}$-aritmetiikka riitäkään?

- Kirjoitetaan auki algoritmin testi

$$\begin{aligned} \text{kulmakerroin}(p) &\leq \text{kulmakerroin}(q)\\ \frac{p.y}{p.x} &\leq \frac{q.y}{q.x} \end{aligned}$$

josta ristiin kertomalla saadaan

$$\begin{cases} p.y \cdot q.x \leq p.x \cdot q.y & \text{kun } p.x \cdot q.x > 0 \\ p.y \cdot q.x \geq p.x \cdot q.y & \text{kun } p.x \cdot q.x < 0 \end{cases}$$

joka on $\mathbb{Z}$-aritmetiikkaa.

- Järjestää saa *minkä tahansa järjestysrelaation* suhteen.

Järjestetään siis tämän relaation suhteen.

- Tämän vuoksi valmiit kirjastorutiinit järjestämiseen usein ottavatkin käytettävän vertailurutiinin parametrina:

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

- Jos ohjelmointikielen tyyppijärjestelmä sallii, niin toinen tapa on määritellä
  - uusi tyyppi `piste`
  - sille oma järjestys `piste::compareTo(`x`:piste)`
  - jota valmis kirjastorutiini `sort` sitten automaagisesti käyttää.

- Java-kieli toimii juuri näin.

Mutta kalvoilla 1.3 näimme, että periytyminen ei ole luontevin tapa ilmaista juuri tätä ohjelmointitapaa...

5.4 Kekojärjestäminen

- *Keko-* eli *kasa*järjestäminen (heapsort) perustuu erityiseen *tietorakenteeseen*
  - jossa seurataan nykyisen sisällön joko

    pienintä — *minimi*keko

    suurinta — *maksimi*keko

    alkiota.
  - josta seuratun alkion poisto on nopeaa

    $$\mathcal{O}(\log(n)).$$
  - jonka alustaminen n alkiolla on nopeampaa kuin aineiston lajittelu.
- Tämä keko- eli kasarakenne (heap) on hyödyllinen myös muiden tietorakenteiden ja algoritmien osana.

- Kekojärjestämisen perusmenetelmä:

  Alusta *minimi*keko H syöteaineistolla;
  while keko H ei ole vielä tyhjä **do**
  Poista keosta H sen pienin alkio b;
  Tulosta b nykyisen vastauksen perään
  end while.
- Voi siis *tulostaa vastauksen alkua "asiakkaalle" jo ennen kuin loppu on laskettu valmiiksi.*

  Kalvojen 5.2 pika- ja kalvojen 5.1 lomitusjärjestämisellä tämä "asiakaslähtöisyys" on mutkikkaampaa toteuttaa.
- Ajantarve on luvattu

  $$\mathcal{O}(n \cdot \log(n))$$

  askelta (kunhan H on kuten luvattiin).
- Tilantarvetta ei voi arvioida ennen kuin H on kuvailtu tarkemmin.

- Keon H toteutus hieman tarkemmin:
  - Aineisto saadaan taulukkona

    $$A[1 \ldots n].$$
  - Taulukon alkupäätä $A[1 \ldots m]$ voidaan käyttää $m \leq n$ alkion kekona H.

    Vastaus kehittyy loppupäähän $A[m+1 \ldots n]$.

    Eli järjestäminen tehdään *paikallaan* kuten kalvojen 5.2 pikajärjestämisessä.
  - Seurattava alkio on ensimmäisessä paikassa $A[1]$.
- Menetelmä hieman tarkemmin:

  1: Muunna syötetaulukko $A[1 \ldots n]$ *maksimi*keoksi;
  2: **for all** $m := n$ **down to** 2 **do**
  3: Poista suurin alkio $b = A[1]$ keosta $A[1 \ldots m]$;
  4: Vie b nyt keolta vapautuneeseen ensimmäiseen vastauspaikkaan $A[m]$
  5: **end for.**

- Esitetään vielä lisätoivomus:

  Voisiko rivien 1 ja 3 keko-operaatiot tehdä *vakio*tilassa?

  Tämä raja koskee *kaikkea* lisämuistia:
  - rekursiopinoa
  - muuta aputilaa.
- Silloin kekojärjestäminen (taulukossa) olisi sellainen järjestämisalgoritmi, jonka tilantarve olisi vain

  syöteaineiston vaatima tila $+ \mathcal{O}(1)$

  muistipaikkaa.

5.4.1 Keko

- Perusmalli: binärikeko (binary heap).
- Tavoite:
  - ylläpidä $m \leq n$ alkion maksimikekoa
  - taulukon $A[1 \ldots n]$ alussa
  - siten että suurimman alkion $A[1]$ poisto vie

    $$\mathcal{O}(\log(m))$$

    askelta.
- Vaiheet:
  1. Mallinna keko kalvojen 5.1.3.1 (mahdollisimman) täydellisenä binääripuuna.
  2. Toteuta tällainen binääripuu taulukolla käyttämättä kalvojen 4.1 osoittimia.

5.4.1.1 Keko binääripuuna

- Tarkastellaan binääripuuta T jonka jokaisessa m solmussa on yksi järjestettävistä alkioista.
- T on keko*järjestyksessä* (heap order) jos sen jokainen solmu s täyttää (maksimi)keko*ehdon* (heap property):

  Isäsolmussa s oleva *avain*alkio $s.\text{key}$ on *vähintään* yhtä suuri kuin sen lapsisolmuissa olevat avainalkiot.

  Jos lasta ei ole, niin sen key $= -\infty$.
- Silloin
  - puun T juuressa r on se kaikkein suurin alkio $r.\text{key}$, joka on seuraavana poistovuorossa
  - seuraavaksi suurimmat alkiot odottavat juuren r lapsisolmuissa vasen(r) ja oikea(r) isäalkion poistumista ja juureen pääsyä.

- Intuitio:

  Tarvittavat muutokset kekoon lienevät nopeita, koska tarvittava tietokin lähellä.
- Oivallus:
  - Jos T on muuten kekojärjestyksessä
  - paitsi että juuressa r onkin

    $$r.\text{key} < \max(\text{vasen}(r).\text{key}\ \ \text{oikea}(r).\text{key})$$

  - niin vaihdetaan keskenään $r.\text{key}$ ja suurempi lasten avaimista key.
  - Silloin virhe kekojärjestyksessä siirtyy vastaavaan alipuuhun.

    (Tai katoaa.)
- Siis jälleen kalvojen 2.1 hajoita ja hallitse -ratkaisu:

  Palautetaan suurempi ongelma pienempään.

Suurimman avainalkion poisto keosta:

1. Poistettava alkio $r.\text{key}$ löytyy puun T juuresta r.
2. Valitaan puun T mielivaltainen lehti l ja poistetaan se.

   (Jos $r = l$ niin keko tyhjentyy.)
3. Korvataan alkio $r.\text{key}$ alkiolla $l.\text{key}$.
4. Korvauksen jälkeen juuri r voi rikkoa kekojärjestyksen:

   Palautetaan kekojärjestys toistamalla edellistä oivallusta.

Puun muuntaminen kekojärjestykseen:

Lehtisolmu on jo kekojärjestyksessä.

Sisäsolmu voidaan muuntaa kun sen lapsisolmut on muunnettu:

Sovelletaan sisäsolmuun edellistä oivallusta.

Näin muunnos voi edetä puussa lehdistä juureen päin.

- Edellisen oivalluksen seurauksena
  - avainten vaihdot etenevät puussa T
  - juuresta alkavaa polkua pitkin
  - mahdollisesti jopa lehteen saakka
  - ja jokaisessa solmussa haara valitaan avainten perusteella.
- Tehokkuus pahimmassakin tapauksessa vaatii siis
  - että lehdet ovat mahdollisimman lähellä juurta
  - eli että puu T on mahdollisimman matala.
- Tämä on ensimmäinen esimerkkimme binääripuun *tasapainoehdosta* joilla turvataan puun poluille pituus

$$\mathcal{O}(\log(\text{puun sisällön määrä}))$$

tehokkuuden takaamiseksi.

5.4.1.2 Täydellinen binääripuu taulukossa

- Binääripuu T on *melkein* (almost) täydellinen, jos
  - se on muuten täydellinen
  - paitsi että sen viimeinen syvyystaso voi olla vajaa

    (eli osa sille kuuluvista lehtisolmuista voi puuttua)
  - tämä viimeinen syvyystaso on täytetty vasemmasta reunasta alkaen

    (eli tyhjät paikat ovat tason oikeassa reunassa).
- Sijoitetaan tällaisen m-solmuisen puun T solmut taulukkoon $A[1 \ldots m]$ siten, että
  - kaaria ei tarvitse esittää
  - vaan ne voidaan laskea taulukkoindekseillä.

- Liitetään puun T jokaiseen kaareen *bitti*

  0 jos kaari vie isäsolmusta *vasempaan* lapseen

  1 jos *oikeaan*.
- Solmun s *nimi* olkoon seuraava bittijono:

  Juuren nimi olkoon 1.

  Muun solmun s nimi saadaan liittämällä sen isäsolmun t nimen perään niiden välisen kaaren $t \xrightarrow{b} s$ bitti b.

  Siis syvyydellä d olevan solmun s nimi on bittijono

$$1 b_1 b_2 b_3 \cdots b_d$$

  joka antaa polun puun T juuresta solmuun s:

  Bitti b_i kertoo käännytäänkö polun i. solmusta vasemmalle vai oikealle.

- Solmut siis numeroidaan
  - taso kerrallaan ylhäältä alkaen
  - tason sisällä vasemmalta oikealle.
- Solmun s indeksi taulukossa $A[1 \ldots m]$ on sen nimi tulkittuna 2-kantaisena (eli binääri)lukuna.
- Jos solmun s indeksi on i, niin sen

  vasemman lapsen indeksi on

$$\text{left}(i) = 2 \cdot i$$

  eli laitetasolla `siirrä yhden bitin verran vasemmalle`.

  oikean lapsen indeksi on

$$\text{right}(i) = \text{left}(i) + 1.$$

  isäsolmun indeksi on

$$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

  eli laitetasolla `siirrä yhden bitin verran oikealle`.

- Lasketun indeksin j pitää pysyä taulukossa $A[1 \ldots m]$:

  Muuten sitä vastaavaa solmua ei ole.

- Juurisolmulla $A[1]$ ei ole isäsolmua.

  Kaikilla muilla solmuilla

  $$A[2], A[3], A[4], \ldots, A[m]$$

  on.

- Jos solmulla $A[i]$

  – ei ole vasenta lasta

  – niin ei myöskään oikeaa

  koska puu T on melkein täydellinen.

- Melkein täydellisyyden nojalla

  – poistettava lehti on $A[m]$

  – lisättävä lehti on $A[m+1]$.

  Keon koko m muuttuu vastaavasti yhdellä.

5.4.1.3 Kekoehdon ylläpito

- Nyt voimme ohjelmoida kalvojen 5.4.1.1 oivalluksen.

- Jos taulukossa $A[1 \ldots m]$ indeksin $1 \leq i \leq m$ lapsista

  – left(i)

  – right(i)

  alkavat alipuut täyttävät jo kekoehdon

  niin kekoehto saadaan voimaan koko indeksistä i alkavaan alipuuhun rekursiivisesti.

procedure heapify($A[1 \ldots m]$: **taulukkoviite**, i: **indeksi**)
  l := left(i);
  r := right(i);
  if $l \leq m$ **and** $A[l]$.key $> A[i]$.key **then**
    $j := l$
  else
    $j := i$
  end if;
  if $r \leq m$ **and** $A[r]$.key $> A[j]$.key **then**
    $j := r$
  end if;
  if $j \neq i$ **then**
    Vaihda taulukkopaikkojen $A[i]$ ja $A[j]$ sisällöt keskenään;
    heapify(A, j)
  end if.

- Kyseessä on kalvojen 4.6 takarekursio eli korvattavissa silmukalla siten, että muistintarve on vain

  $$\mathcal{O}(1). \quad (8)$$

- Jos indeksi i on puussa korkeudella h, niin kutsu heapify(A, i) vie

  $$\mathcal{O}(h) \quad (9)$$

  askelta, koska se kulkee indeksin i alipuussa alaspäin.

- Nyt

  $$h = \mathcal{O}(\log(\text{puun solmujen lukumäärä}))$$

  koska koko puu on melkein täydellinen.