5.4.1.4 Taulukosta keoksi

- Nyt voimme ohjelmoida kalvojen 5.4.1.1 tavan muuntaa syötteenä saatu taulukko $A[1 \ldots n]$ keoksi.
- Kuljetaan puun solmuja numerointijärjestyksessä *taakse*päin alkaen ensimmäisestä sisäsolmusta.
- Silloin solmua käsiteltäessä sen lapset täyttävät jo kekoehdon.
- Siis riittää käyttää kalvojen 5.4.1.3 aliohjelmaa.

procedure buildHeap($A[1 \ldots n]$: **taulukkoviite**)
  m := n;
  for i := $\lfloor \frac{n}{2} \rfloor$ **down to** 1 **do**
    heapify(A, i)
  end for.

- Aputilan tarve on kaikkiaan

$$\mathcal{O}(1). \tag{10}$$

- Koska yhtälön (9) mukaan yhden aliohjelmakutsun aikavaatimus riippuu korkeudesta h, niin summataan yli silmukassa läpikäytyjen korkeuksien.
- Koska puu on miltei täydellinen, niin sen korkeus on enintään

$$\lfloor \log(n) \rfloor .$$

- Korkeudella h on enintään

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$

solmua.

- Aikavaatimukselle saadaan summa

$$\begin{aligned} \sum_{h=0}^{\lfloor \log(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot h &= \mathcal{O}\left(n \cdot \frac{1}{2} \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= \mathcal{O}(n) \end{aligned} \tag{11}$$

aputuloksen (12) nojalla.

- Tarvitsemme aputuloksen

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2. \tag{12}$$

- Yhtälö (12) voidaan osoittaa seuraavasti:

  Koulumatematiikasta tunnemme äärettömän *geometrisen sarjan* summan

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$$

  kun $|x| < 1$. Kohdistamme sen molempiin puoliin operaation

$$x \cdot \mathbf{D}_x \left(\sum_{h=0}^{\infty} x^h \right) = x \cdot \mathbf{D}_x \left(\frac{1}{1-x} \right).$$

  Vasen puoli saa muodon

$$\begin{aligned} x \cdot \sum_{h=1}^{\infty} \mathbf{D}_x \left(x^h \right) &= x \cdot \sum_{h=1}^{\infty} h \cdot x^{h-1} \\ &= \sum_{h=0}^{\infty} h \cdot x^h \end{aligned}$$

  joka on mitä etsimme arvolla $x = \frac{1}{2}$.

- Yhtälön (12) oikea puoli saa muodon

$$\begin{aligned} x \cdot \mathbf{D}_x \left(\frac{1}{1-x} \right) &= x \cdot \frac{-1}{(1-x)^2} \cdot -1 \\ &= \frac{x}{(1-x)^2} \\ &= 2 \end{aligned}$$

  arvolla $x = \frac{1}{2}$.

- Lopuksi pitää vielä tarkistaa että geometrisen sarjan derivointi termeittäin oli sallittua arvollamme $x = \frac{1}{2}$.

  Oli, koska geometrinen sarja suppenee peräti *tasaisesti* jokaisella suljetulla välillä

$$[-k, +k] \subsetneq (-1, +1),$$

  siis myös pisteemme $x = \frac{1}{2}$ ympäristössä.

5.4.2 Kekojärjestämisalgoritmi

- Kootaan lopuksi kaikki yhteen:

  procedure heapsort(A: **taulukkoviite**)
  1: buildHeap(A) kalvoilta 5.4.1.4;
  2: **while** $m > 1$ **do**
  3: Vaihda taulukkopaikkojen $A[1]$ ja $A[m]$ sisällöt keskenään;
  4: $m := m - 1$;
  5: heapify($A, 1$) kalvoilta 5.4.1.3
  6: **end while.**

- Kokonaisaika on

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

  yhtälöistä (9) ja (11).

- Aputilan kokonaistarve on

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

  yhtälöistä (8) ja (10).

5.4.3 Prioriteettijono

- Kalvojen 5.4.1 kekotietorakenteesta saadaan helposti *prioriteettijono* (priority queue):
  - Tehtävien töiden jono kuten kalvoilla 4.3, mutta seuraavaksi otetaankin aina *kiireellisin* työ.
  - Työn kiireellisyys eli *prioriteetti* annetaan silloin, kun työ saapuu jonoon.

    Seuraavaksi ryhdytään aina korkeimman prioriteetin omaavaan työhön.

- Esimerkiksi seuraavaksi prosessoriaikaa saavan prosessin valinta käyttöjärjestelmän ytimessä.

- Prioriteettijonon perusoperaatiot:
  - heapInsert(S, x) *lisää*
    - prioriteettijonoon S alkion x.
    - jonka prioriteetti on avainkenttänä x.key.
  - heapMaximum(S)
    - *palauttaa poistamatta* prioriteettijonosta S sellaisen alkion y
    - jonka prioriteetti y.key on mahdollisimman suuri.
  - heapExtractMax(S)
    - *poistaa ja* palauttaa
    - saman alkion y kuin heapMaximum(S).

- Prioriteettijonon toteutus kekona:
  - Oletetaan:
    - osaamme varata prioriteettijonolle S etukäteen tarpeeksi suuren taulukon $A[1 \ldots n]$
    - ettei sen käytössä oleva alkuosa $A[1 \ldots m]$ koskaan jää liian pieneksi.

    Myöhemmin palataan siihen, miten tämä oletus voidaan poistaa.
  - heapMaximum(A) palauttaa taulukkopaikan $A[1]$ sisällön.
  - heapExtractMax(A) saadaan muunnoksena kalvon 5.4.2 algoritmin riveistä 3–5.
  - heapInsert(A, x) tehdään lisäämällä seuraava lehti:

    $m := m + 1$;
    $A[m] := x$;
    shiftup(A, m).

- Operaatio shiftup(A, i) on sukua kalvojen 5.4.1.3 algoritmille:
  - Taulukko A täyttää muuten kekoehdon
  - paitsi että solmu $A[i]$ ja sen *isäsolmu* A[parent(i)] voivat rikkoa sen.

  Silloin rike voidaan siirtää puussa *ylös*päin.

  procedure shiftup(A: **taulukkoviite**, i: **indeksi**)
  if $i > 1$ **and** $A[i]$. key $> A$[parent(i)]. key **then**
  Vaihda näiden taulukkoalkioiden $A[i]$ ja A[parent(i)] sisällöt keskenään;
  shiftup(A, parent(i))
  end if.

- Aliohjelma voidaan tehdä
  - toistorakenteena (kuten kalvoilla 4.6)
  - ilman turhia vaihtoja (samoin kuin kalvojen 5.4.1.3 algoritmikin)
  - ajassa

    $$\mathcal{O}(\text{puun korkeus}) = \mathcal{O}(\log(m)).$$

- Usein sallitaan myös prioriteettijonossa olevan alkion prioriteetin
  - nostaminen (tärkeämmäksi)
  - laskeminen (vähemmän tärkeäksi).

- Operaatio setPriority(A, i, p) asettaa kekotaulukon A paikassa i olevan alkion uudeksi prioriteetiksi p.

  procedure setPriority(A: **taulukkoviite**, i: **indeksi**, p: **prioriteetti**)
  q := $A[i]$. key;
  $A[i]$. key := p;
  if $p < q$ **then**
  heapify(A, i)
  else
  shiftup(A, i)
  end if.

- Mistä prioriteettijonon käyttäjä tietää
  - sen taulukkopaikan i
  - jossa *juuri nyt* on
  - se kekoalkio x jonka prioriteettia halutaan muuttaa?

  Alkion x paikka i keossa vaihtelee!

- Usein kekokirjasto tarjoaakin alkioon x *kahvan* (handle) joka tietää sen.

- Yksi toteutustapa on:
  - Kekotaulukossa A onkin kalvojen 4.1 osoittimia kekoalkiotietueisiin t.
  - Tietueessa t on (myös) kenttä t. handle.
  - Keko-operaatiot ylläpitävät jokaiselle tietueelle t invarianttia

    "$A[t.\text{handle}]$ osoittaa takaisin tietueeseen t".
  - Käyttäjä saa muuttaa prioriteettikenttää t. key.

5.5 Järjestämisalgoritmien työhönottohaastattelu

- Aikavaatimus?

  Lomitusjärjestäminen (kalvoilta 5.1): $\Theta(n \cdot \log(n))$.

  Pikajärjestäminen (kalvoilta 5.2): Tyypillisesti $\mathcal{O}(n \cdot \log(n))$ mutta pahimmillaan $\mathcal{O}(n^2)$.

  Kekojärjestäminen (kalvoilta 5.4): $\Theta(n \cdot \log(n))$.

- Suosikkitietorakenne?

  L: Lista jonka osoittimia saa muuttaa.

  P: Kumpi vain.

  K: Taulukko jota saa muuttaa.

- Rekursiopinon tarve?

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

  P: Perusversio tyypillisesti $\mathcal{O}(\log(n))$ mutta pahimmillaan $\mathcal{O}(n)$.
  Ohjelmoitavissa siten, että $\mathcal{O}(\log(n))$.

  K: $\mathcal{O}(1)$.

- Muun aputilan tarve?

  L: $\mathcal{O}(1)$ suosikkitietorakenteella, muuten $\mathcal{O}(n)$.

  P: $\mathcal{O}(1)$.

  K: $\mathcal{O}(1)$.

- Erikoistaito?

  L: Helppo ohjelmoida *vakaa*ksi eli *stabiili*ksi (stable): säilyttämään *yhtä suurten* alkioiden keskinäisen järjestyksen.

  P: Käytännössä tehokkain ja suosituin.

  K: Inkrementaalinen jo perusversiona.

5.6 Järjestäminen vertailuin

- Osoitetaan että n alkion järjestäminen vie *ainakin*

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

  askelta...

- ...*jos ainoa* syötealkioille a_i sallittu operaatio on niiden *vertailu* toisiinsa kysymyksin

  $$\text{"Onko } a_i \leq a_j\text{?"} \qquad (13)$$

  (Muut vertailut '$<$', '$=$', '$\geq$', '$\neq$' ja '$>$' tehtävissä 2 kysymyksellä.)

- Käytetään *päätöspuita* (decision tree):

  kalvojen 5.1.3.1 binääripuu jonka

  sisäsolmussa on kahdesta syötealkiosta esitetty kysymys (13).

  haarautuminen on vastauksen mukaan

  kyllä vasemmalle

  ei oikealle.

  lehtisolmussa pysähtyy johtopäätökseen "syötealkioiden (jokin) oikea järjestys on

  $$a_3 \leq a_{57} \leq a_7 \leq \cdots \leq a_{14}\text{".}$$

- Päätöspuun T kasvatus:

  – valitaan järjestämisalgoritmi α

  – valitaan syötepituus n

  – juureksi *tuntematon* syöte

  $$a_1, a_2, a_3, \ldots, a_n \qquad (14)$$

  – solmun lapset lasketaan *simuloimalla* algoritmia α seuraavaan kysymykseen.

- Lehdessä l on tuntemattoman syötteen (14) sellainen uudelleenjärjestely eli *permutaatio*

  $$a_{\pi(1)}, a_{\pi(2)}, a_{\pi(3)}, \ldots, a_{\pi(n)}$$

  joka *sopii yhteen* niiden vastausten kanssa, jotka ovat polulla puun T juuresta tähän lehteen l:

  Jos polulla olevista vastauksista seuraa

  $$a_i < a_j$$

  niin silloin permutaatiossa π pitää olla

  $$\pi^{-1}(i) < \pi^{-1}(j)$$

  eli "syötearvon a_i pitää olla ennen syötearvoa a_j".

- Jos puusta T puuttuu jokin permutaatio σ niin käänteinen yhteensopivuusehto

  $$\text{jos } \sigma^{-1}(i) < \sigma^{-1}(j) \text{ niin } b_i < b_j$$

  antaa vastaesimerkin: syötteen

  $$b_{\sigma(1)}, b_{\sigma(2)}, b_{\sigma(3)}, \ldots, b_{\sigma(n)}$$

  jolla *algoritmi* α *erehtyy* eli joka päätyy väärään lehteen.

- Siis puussa T pitää olla *ainakin* yhtä monta lehteä kuin n alkiolla on permutaatioita.

- Nyt binääripuussa T

$$\begin{aligned}\text{korkeus} &\geq \log_2(\text{lehtien määrä}) \\ &\quad \text{lauseesta 5.1.2} \\ &\geq \log_2(n!) \\ &\quad \text{edellisestä järkeilystä} \\ &= \Omega(n \cdot \log(n)) \\ &\quad \text{yhtälöstä (2).}\end{aligned}$$

- Siis jollakin n alkion syöteaineistolla algoritmi α vie ainakin

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

askelta.

- Koska α oli mielivaltainen
  - oikein toimiva
  - kysymyksiä (13) käyttävä
  - järjestämisalgoritmi

niin väite seuraa

5.6.1 Muita alarajoja

- Alarajoja on vaikeampi todistaa kuin ylärajoja:

  Ylärajaan riittää kirjoittaa *jokin* algoritmi joka toimii niin nopeasti.

  Alarajaan pitää osoittaa että *jokainen* algoritmi — vaikka kuinka nerokas — jää joskus hitaaksi.

- Järjestämisen lisäksi tunnetaan esimerkiksi seuraavat alarajat:
  - Alkion haku järjestetystä n-alkioisesta taulukosta:

    $$\Omega(\log(n)).$$

    Kalvojen 2.4.2 binäärihaku oli optimaalinen.
  - Toistuuko jokin alkio tässä n-alkioisessa järjestämättömässä aineistossa?

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

    Teimme tämän kalvoilla 5.3.

5.6.2 Järjestäminen vertailuitta

- Jos alkioille saa tehdä muutakin kuin kysymyksiä (13) niin järjestäminen onnistuu nopeamminkin.

- Oletetaan siis että syöteaineisto

$$a_1, a_2, a_3, \ldots, a_n$$

koostuu *numeroista*.

- Silloin voimme lisäksi
  1. indeksoida taulukkoja
  2. tehdä laskutoimituksia.

5.6.2.1 Laskemisjärjestäminen

- *Laskemis*järjestäminen (counting sort) soveltuu, kun syötetaulukko

$$A[1 \ldots n]$$

sisältää pelkästään niin *pieniä* numeroita $0, 1, 2, \ldots, k$

- että voimme käyttää aputaulukkoa

$$C[0 \ldots k]$$

jota voimme indeksoida syötealkioilla $A[j]$.

Riveillä 2–4 $C[i]$ on niiden syötealkioiden lukumäärä jotka $= i$.

Riveillä 5–7 $C[i]$ on niiden syötealkioiden lukumäärä jotka $\leq i$.

- Lisäksi käytämme tulostaulukkoa

$$B[1 \ldots n]$$

johon kokoamme vastauksen.

Riveillä 8–11 laskureina $C[i]$.

procedure cs(A: **taulukko**, B: **taulukkoviite**)

```
 1: Varaa aputaulukko C ja nollaa se;
 2: for all j := 1 to n do
 3:     C[A[j]] := C[A[j]] + 1
 4: end for;
 5: for all i := 1 to k do
 6:     C[i] := C[i] + C[i − 1]
 7: end for;
 8: for all j := n down to 1 do
 9:     B[C[A[j]]] := A[j];
10:     C[A[j]] := C[A[j]] − 1
11: end for.
```

- Aikaa ja tilaa kuluu

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

  mikä on

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

  järkevillä k.

- Vastaukset kerätään riveillä 8–11 takaperin.

- Algoritmi on vakaa: viimeinen syötealkio kopioidaan viimeiseen mahdolliseen paikkaansa. (Vertaa kalvot 5.5.)

5.6.2.2 Kantalukujärjestäminen

- *Kantaluku*järjestäminen (radix sort) järjestää *suuretkin etumerkittömät kokonais*luvut

  $$a_1, a_2, a_3, \ldots, a_n.$$

  seuraavin vaihein:

1. Kirjoita luvut ruutupaperille omille riveilleen
   - syötejärjestyksessä $a_1, a_2, a_3, \ldots$ alekkain
   - oikea reuna tasaten
   - täyttäen etunollilla vasemmatkin reunat tasan.
2. Aloita oikeanpuoleisimmasta sarakkeesta.
3. Järjestä rivit paperilla tämän *yhden* sarakkeen perusteella.

   Tee se vakaasti: vaihtamatta kahden sellaisen rivin keskinäistä järjestystä, joilla on tässä sarakkeessa sama numeromerkki.

4. Siirry yksi sarake vasemmalle ja jatka vaiheesta 3.
5. Lopeta kun kaikki sarakkeet on käyty näin läpi.

- Luonteva valinta vaiheessa 3 on kalvojen 5.6.2.1 laskemisjärjestäminen arvolla

  $$\begin{aligned} k &= \text{numeromerkkien lukumäärä} \\ &= 10 \quad \text{kymmen- ja} \\ &= 2 \quad \text{binäärilukujärjestelmässä} \end{aligned}$$

  ja niin edelleen.

- Kantaluvun k saamme valita itse!

- Aikaa kuluu

  $$\mathcal{O}(\text{rivien määrä} \cdot \text{rivin pituus}) = \mathcal{O}(n \cdot \log_k \max \text{syöteluvuista } a_i)$$

  askelta kalvojen 3.5 mukaan.

5.6.2.3 Lokerikkojärjestäminen

- Myös syöteaineiston

  $$a_1, a_2, a_3, \ldots, a_n.$$

  todennäköisyys*jakauman* tunteminen voi nopeuttaa järjestämistä.

- Siihen perustuu *lokerikko*järjestäminen (bucket sort).

- Oletetaan että syötealkiot
  - ovat *reaali*lukuja $0.0 \le a_i < 1.0$
  - on poimittu *tasaisesta* jakaumasta.

    Intuitiivisesti: kun poimitaan a_i, niin todennäköisyys että se osuu väliin $p \le \cdots < q$ on välin *pituus* $q - p$.

- Jaetaan siis arvoalue yhtä suuriin lokeroihin $B[0 \ldots n-1]$ joiden pituus on $\frac{1}{n}$.

- Pudotetaan jokainen syötealkio a_i kokonsa mukaiseen lokeroon.

- Jokainen lokero on lista siihen pudonneista syötealkioista.

- Jokainen lista on todennäköisesti (hyvin) lyhyt.

- Lyhyt lista voidaan lajitella jollakin helpolla

  $$\mathcal{O}(\text{listan pituus}^2)$$

  askeleen algoritmilla.

  (Esimerkiksi kalvojen 3.4 lisäyslajittelun listaversiolla kalvoilta 4.7.)

- Aikavaatimuksen oletusarvoksi saadaan

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

  (todennäköisyyslaskelmat sivuutetaan).

for all i := 1 **to** n **do**
  Lisää syötealkio a_i (aluksi tyhjään) listaan $B[\lfloor n \cdot a_i \rfloor]$
end for;
for all i := 0 to $n-1$ **do**
  Lajittele lista $B[i]$
end for;
Liitä listat $B[0], B[1], B[2], \ldots, B[n-1]$ tässä järjestyksessä peräkkäin vastaukseksi siten, että liittämiseen kuluu kokonaisuudessaan $O(n)$ askelta.