58131: Tietorakenteet

Versio: 8. tammikuuta 2007

- Nämä kalvot on laatinut Matti Nykänen kevään ja syksyn 2005 luentoihin.
- Ne pohjautuvat Matti Luukkaisen kevään ja syksyn 2004 kalvoihin.
- Niihin on tehty muutoksia myös Kerttu Pollari-Malmin kevään 2006 luentojen pohjalta, joilla ne olivat oheismateriaalina.

Sisältö

1 Johdanto 1
- 1.1 Oikeellisuus 7
- 1.2 Vaativuus 17
- 1.3 Funktioiden kertaluokat 25
  - 1.3.1 Sääntöjä algoritmien analysointiin 33
  - 1.3.2 Esimerkki: Kuplajärjestäminen 41
  - 1.3.3 Esimerkki: Binäärihaku 43
- 1.4 Logaritmeista 48
- 1.5 Abstrakti tietotyyppi joukko 52

2 Pino, jono ja lista 58
- 2.1 Pino 59
- 2.2 Jono 65
- 2.3 Linkitetyt rakenteet 71
- 2.4 Lista 80
  - 2.4.1 Järjestetty rengaslista 91
  - 2.4.2 Tunnussolmullinen lista 99
- 2.5 Pinon toteutus Javalla 103
- 2.6 Esimerkki pinoa käyttävästä algoritmista 106

3 Hakupuut 109
- 3.1 Binääripuu 119
  - 3.1.1 Binääripuualgoritmien johtamisesta 128
  - 3.1.2 Binäärihakupuut 131
  - 3.1.3 Joukko-operaatioiden toteuttaminen 144
- 3.2 Tasapainoisia puulajeja 165
- 3.3 Punamustat puut 168
  - 3.3.1 Kierrot 178
  - 3.3.2 Värit säilyttävä kierto 185
  - 3.3.3 Alkion lisäys 187
  - 3.3.4 Solmun poisto 203
- 3.4 Monta indeksiä samaan dataan 235
- 3.5 B-puut 239
  - 3.5.1 Avaimen haku 249
  - 3.5.2 Tyhjän puun luonti 254
  - 3.5.3 Avaimen lisäys 255
  - 3.5.4 Avaimen poisto 270
- 3.6 Yleisen puun talletus ja läpikäynti 280
- 3.7 Puut ongelmanratkaisussa 292
  - 3.7.1 Kahdeksan kuningattaren ongelma 293
  - 3.7.2 Permutaatiot 306
  - 3.7.3 Kauppamatkustajan ongelma 313
- 3.8 Pelipuu 322

4 Hajautus 334
4.1 Yhteentörmäykset ylivuotoketjuun 341
4.2 Hajautusfunktion valinta 353
4.2.1 Merkkijonosta luvuksi 355
4.2.2 Jakojäännösmenetelmä 359
4.2.3 Kertolaskumenetelmä 363
4.2.4 Universaalihajautus 367
4.3 Avoin hajautus 372
4.3.1 Operaatiot kokeillen lineaarisesti 376
4.3.2 Neliöinen kokeilu 397
4.3.3 Kaksoishajautus 400
4.3.4 Avoimen hajautuksen analyysistä 406
4.4 Taulukon pidentäminen 409
4.5 Käytännöllisiä huomautuksia 414
5 Keko 418
5.1 Keon käyttökohteita 422
5.2 Binäärikeko 425
5.3 Muita operaatioita 444
5.4 Kahvat keon sisälle 455
6 Järjestäminen 459
6.1 Kekojärjestäminen 460
6.2 Lomitusjärjestäminen 464
6.2.1 Vaativuusanalyysi 478
6.2.2 Ilman rekursiota 487
6.2.3 Listoilla 491
6.3 Pikajärjestäminen 496
6.3.1 Ositus 500
6.3.2 Vaativuusanalyysi 508
6.3.3 Käytännön huomautuksia 517
6.4 Järjestämisen alaraja 520
6.5 Järjestäminen lineaarisessa ajassa 530
6.6 Yhteenveto ominaisuuksista 543
6.7 Järjestäminen algoritmin osana 546
7 Verkot 549
7.1 Käsitteistö 555
7.2 Verkkojen tallettaminen 563
7.2.1 Vieruslistat 564
7.2.2 Vierusmatriisit 570
7.3 Verkon läpikäynti 578
7.3.1 Leveyssuuntainen läpikäynti 579
7.3.2 Syvyyssuuntainen läpikäynti 588
7.3.3 Verkon syklittömyyden tarkastus 601
7.3.4 Topologinen järjestäminen 606

7.3.5 Kriittiset työvaiheet 611
7.3.6 Vahvasti yhtenäiset komponentit 618
7.4 Lyhyimmät polut 631
7.4.1 Ongelmanratkaisusta Dijkstralla 654
7.4.2 Bellmanin ja Fordin algoritmi 660
7.4.3 Floydin algoritmi 668
7.5 Transitiivinen sulkeuma 679
7.6 Verkon virittävät puut 685
7.6.1 Kruskalin algoritmi 691
7.6.2 Union-find-tietorakenne 699
7.6.3 Kruskal ja keko 712
7.6.4 Esimerkki: Ryvästys 714
7.6.5 Primin algoritmi 716
7.6.6 Kruskal vai Prim? 725
7.6.7 Esimerkki: Laboratoriohiiri 726
7.6.8 Oikeellisuudesta 735

1 Johdanto

- Kaikki epätriviaalit ohjelmat joutuvat tallettamaan ja käsittelemään tietoa suoritusaikanaan

- Esim. "puhelinluettelo":
  - numeron lisäys
  - numeron poisto
  - numeron muutos
  - numeron haku nimen perusteella
  - nimen haku numeron perusteella
  - nimi-numero -parien tulostaminen aakkosjärjestyksessä
  - ...

- Suoritusaikana tiedot tallennetaan *tietorakenteeseen*. Sillä tarkoitetaan
  - tapaa miten tieto tallennetaan koneen muistiin, *ja*
  - operaatioita joiden avulla tietoa päästään käyttämään ja muokkaamaan.

- Joskus lähes samasta asiasta käytetään nimitystä *abstrakti tietotyyppi:*
  - tietorakenteen sisäinen toteutus piilotetaan käyttäjältä, ja
  - abstrakti tietotyyppi näkyy käyttäjille ainoastaan operaatioina joiden avulla tietoa käytetään
  - abstrakti tietotyyppi ei siis ota kantaa siihen miten tieto on koneen muistiin varastoitu . . .

- Puhelinluettelon tietorakenne voisi olla esim. seuraavanlainen:

- Oleellisia operaatioita ovat (abstrakti tietotyyppi puhelinluettelo)

  init luo tyhjän puhelinluettelon

  add(n,p) lisää puhelinluetteloon nimen n ja sille numeron p

  del(n) poistaa nimen n puhelinluettelosta

  find(n) palauttaa luettelosta henkilön nimeltä n puhelinnumeron

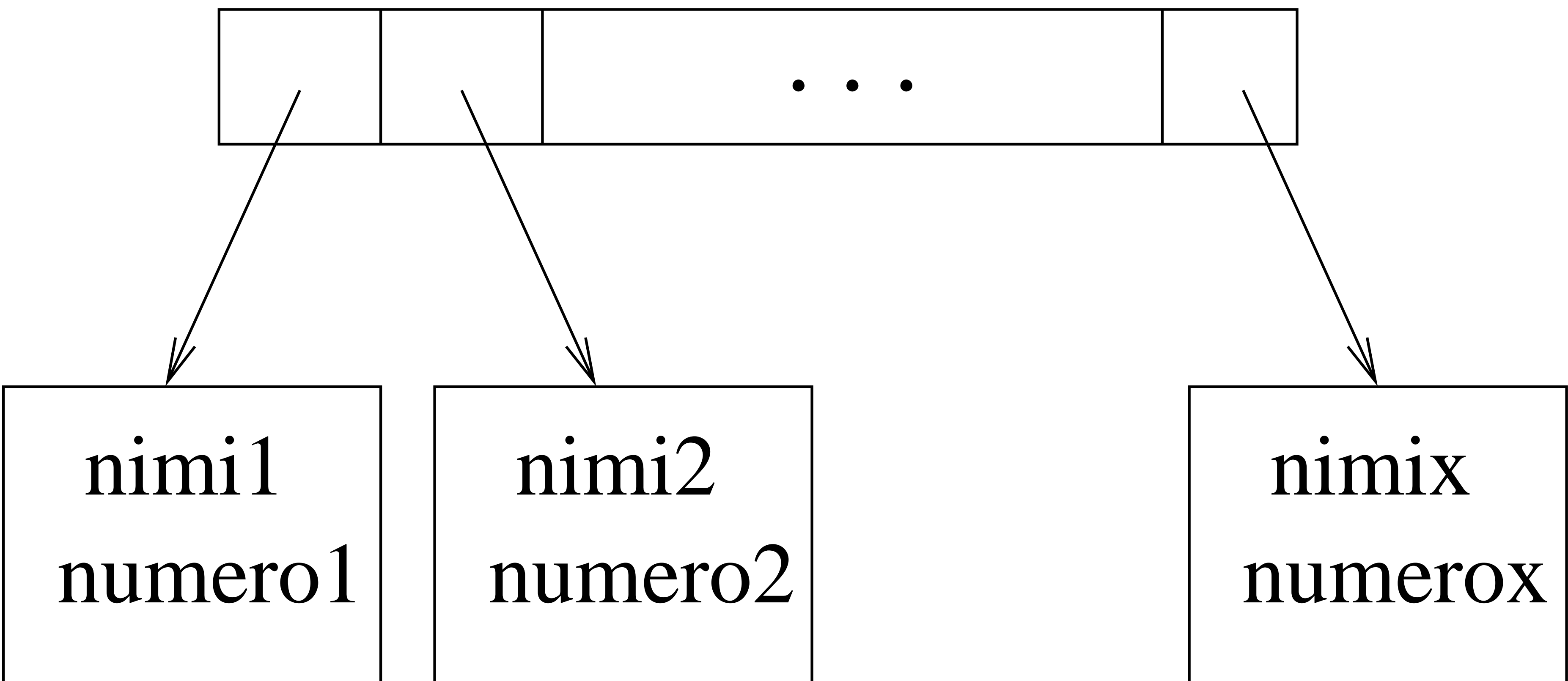
  list listaa puhelinluettelon nimi-numero-parit aakkosjärjestyksessä.

- Tieto voisi olla talletettu taulukkoon

  joko suoraan taulukon alkioihin

  tai siten, että taulukossa on *linkki* varsinaisen nimi/numero-parin tallettavaan muistialueeseen.

nimi1 numero1	nimi2 numero2	. . .	nimix numerox


. . .
nimi1
numero1
nimi2
numero2
nimix
numerox

- Kurssilla tulemme näkemään monia eri vaihtoehtoja valita tietorakenne, jolla on puhelinluettelolle sopivat operaatiot: lista (kalvoilla 2.4), hakupuu (kalvoilla 3), hajautustaulu (kalvoilla 4),. . .

- Kurssin tavoite on osata *valita* eri tilanteisiin sopivat tietorakenteet, ja *perustella* valintansa.
  - Kurssilla esitellään tavallisimpia perustietorakenteita ja niiden käsittelyalgoritmeja, jotta olisi mistä valita.
  - Keskeinen valintakriteeri on *tehokkuus tiedon määrän kasvaessa* (kalvoilla 1.2).
  - Tärkeää on myös osata *toteuttaa* ne oikein.
    - Siksi tarkastellaan ensin lyhyesti algoritmien oikeellisuutta (kalvoilla 1.1).
    - Varsinaiseen käytännön toteutustyöhön tutustutaan erillisenä harjoitustyönä.

1.1 Oikeellisuus

- Yksi mahdollisuus toteuttaa puhelinluettelon **add** operaatio (taulukkototeutuksessa) on lisätä uusi nimi-numero-pari aina taulukon ensimmäiseen tyhjään paikkaan.

- Tällöin operaation **list** yhteydessä nimet täytyy järjestää aakkosjärjestykseen ...

- Järjestäminen on yleisemminkin tärkeässä roolissa tietojenkäsittelyssä ja myöhemmin (kalvoilla 6) kurssilla tarkastellaankin muutamia tehokkaita järjestämisalgoritmeja.

- Tarkastellaan nyt esimerkkinä lisäysjärjestämistä (insertion sort).

- Tällä kurssilla esitetään algoritmit *pseudokoodina*

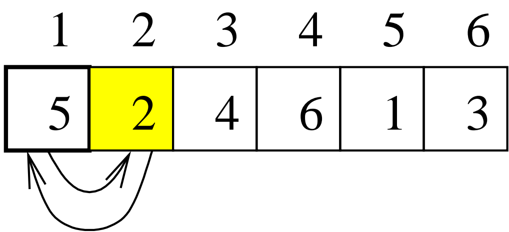
  eli notaationa joka on niin

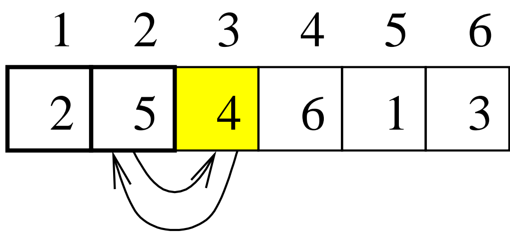
  konkreettinen että tiedetään miten se voitaisiin oikeasti ohjelmoida

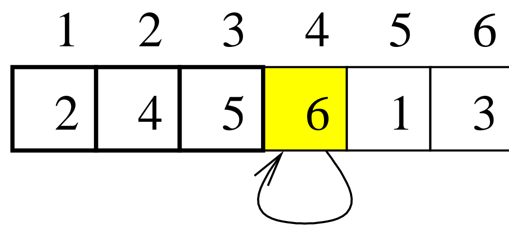
  abstrakti että kaikkia oikean ohjelman yksityiskohtia ei tarvitse esittää.

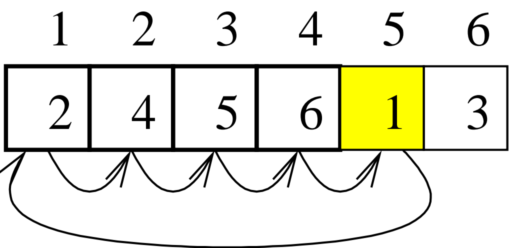
```
1  for j ← 2 to length[A] do
2      key ← A[j]
3      ▷A[j] paikalleen osaan A[1,...,j − 1]
       ▷joka on jo järjestetty
4      i ← j − 1
5      while i > 0 and A[i] > key do
6          A[i + 1] ← A[i]
7          i ← i − 1
8      A[i + 1] ← key
```

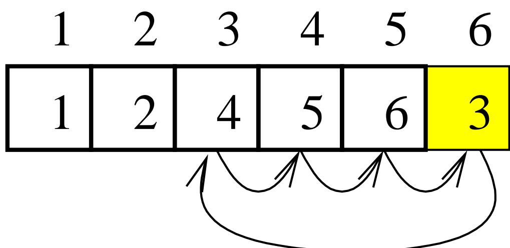
- Algoritmin toiminnasta esimerkkisyötteellä voidaan piirtää kuva.

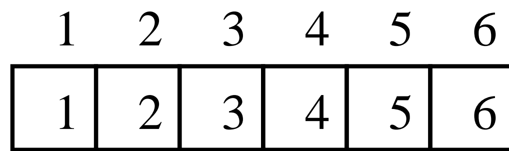

1 2 3 4 5 6
5 2 4 6 1 3


1 2 3 4 5 6
2 5 4 6 1 3


1 2 3 4 5 6
2 4 5 6 1 3


1 2 3 4 5 6
2 4 5 6 1 3


1 2 3 4 5 6
1 2 4 5 6 3


1 2 3 4 5 6
1 2 3 4 5 6

- Silmukan toimintaidea voidaan ilmaista ns. *invariantin* avulla: väitteen, joka
  - on totta silmukan alussa
  - pysyy totena vaikka silmukan kuluessa muuttujien arvot päivittyvätkin.

- Lisäysjärjestämisen ulomman silmukan invariantti on:

  for-lauseen jokaisen suorituksen alussa taulukon osa $A[1, \ldots, j-1]$ on järjestyksessä ja sisältää alunperin taulukon osassa $A[1, \ldots, j-1]$ olleet alkiot.

  - Totta triviaalisti alussa, koska silloin $j = 2$.
  - Pysyy totena kun j kasvaa (yhdellä), koska
    - **for**-silmukan sisältö on *suunniteltu* juuri siten, että se pysyisi totena
    - apuna on **while**-silmukan oma invariantti.

- **while**-silmukan oma invariantti:

  while-lauseen alussa

  alkuosa $A[1,\ldots,i]$ on yhä samanlainen kuin ennen **while**-silmukkaa

  loppuosa $A[i+2,\ldots,j]$

  – on samanlainen kuin $A[i+1,\ldots,j-1]$ oli ennen **while**-silmukkaa

  – koostuu sellaisista alkioista jotka ovat $>$ key.

  – Totta alussa koska $i = j - 1$:

  loppuosa $A[j+1,\ldots,j]$ on *tyhjä!*

  – Säilyy totena kun i pienenee (yhdellä):

    * **alkuosan** viimeinen alkio $A[i]$ siirtyy **loppuosan** eteen alkioksi $A[i+1]$

    * silmukkaehdon nojalla $A[i] >$ key

  ennen kasvattamista.

- Silmukkaan liittyy myös *konvergentti* joka takaa, että se pysähtyy: suure joka
  - pienenee aidosti jokaisella kierroksella
  - tekee silmukkaehdosta epätoden tarpeeksi pienillä arvoilla.

- Lisäysjärjestämisesimerkissämme
  - **while**-sisäsilmukan konvergentti on muuttuja i itse
  - **for**-ulkosilmukan konvergentti on length$[A] - j$.

- Koko algoritmin oikeellisuusperustelu on sen lähdekoodin mukainen yhdistelmä yksittäisiä invariantteja ja konvergentteja:

  1. **while**-sisäsilmukan invariantti ja konvergentti takaavat yhdessä, että silmukan *lopussa* $A[1,\ldots,j-1]$ on saatu ositettua siten, että

     alkuosa $A[1,\ldots,i]$ on yhä paikoillaan

     loppuosa $A[i+1,\ldots,j-1]$ on siirtynyt "yhden paikan eteenpäin" paikoille $A[i+2,\ldots,j]$

     välialkioon $A[i+1]$ voi tallettaa alkion key siten, että koko $A[1,\ldots,j]$ on järjestyksessä.

  2. Tämä välialkion tallennus tehdään rivillä 8.

  3. Rivillä 2 välialkioksi otettiin $A[j]$.

4. Siispä **for**-ulkosilmukan rungon lopussa koko $A[1,\ldots,j]$ on järjestyksessä ja sisältää alunperin taulukon osassa $A[1,\ldots,j]$ olleet alkiot

   eli **for**-ulkosilmukan invariantti *on* totta kun sen indeksiä j kasvatetaan.

5. **for**-ulkosilmukan konvergentti takaa pysähtymisen indeksin arvoon $j = \text{length}[A] + 1$.

6. **for**-ulkosilmukan invariantti antaa silloin halutun lopputuloksen:

   lopuksi koko taulukko $A[1,\ldots,\text{length}[A]]$ on järjestyksessä ja sisältää alunperin siinä olleet alkiot.

- Opiskeluvihje

  algoritmeille ja tietorakenteille:

  – Keskity invariantteihin, älä koodin yksityiskohtiin

  – koska jälkimmäiset voi päätellä edellisistä.

  ohjelmien oikeellisuuteen:
  Tietojenkäsittelijä tarvitsee koodinsa suunnitteluun ja kirjoittamiseen

  – systemaattista

  – huolellista

  – kaikki tapaukset huomioivaa

  päättelytaitoa.

  Samaa, jota matemaatikko tarvitsee laatiessaan omia todistuksiaan.

- Huomautuksia pseudokoodista:
  - Lohkorakenne ilmaistaan sisennyksen avulla (Javassa {...}).
  - Kommenttirivi alkaa $\triangleright$-merkillä.
  - Sijoitus merkitään $x \leftarrow 5$ (Javassa $x = 5$).
  - Taulukon alkioihin viitataan tyyliin $A[i]$.
  - Koosteinen data esitetään olioina/tietueina joilla on kentät:

    esim. jos C on tyyppiä jolla on kentät key ja next, näihin kenttiin viitataan key[C] ja next[C] (esim. Javassa sama tapahtuu C.key ja C.next).
  - **for**-lauseen indeksimuuttujan loppuarvo:

    esim. lauseen
    for $i \leftarrow 1$ **to** 10 ... suorituksen jälkeen
    $i = 11$
  - Jos heti **for**-lauseen aluksi indeksin alkuarvo ylittää loppuarvon, niin silmukkaa ei tehdä kertaakaan.

1.2 Vaativuus

- Algoritmin tehokkuudella tai vaativuudella tarkoitetaan sen suoritusaikana tarvitsemia resursseja:
  - suoritukseen kuluva aika
  - suorituksen vaatima muistitila
  - kommunikoinnin määrä
  - ...
- Tällä kurssilla tarkastellaan lähinnä *aikaa*,

  joskus myös tilaa.
- Resurssintarvetta tarkastellaan *suhteessa syötteen kokoon*

  esim. järjestettäessä koko on taulukon A alkioiden lukumäärä length[A].

- Voimme analysoida resurssien tarvetta

  parhaassa tapauksessa — ei yleensä kiinnosta

  keskimääräisessä tapauksessa — vaatisi todennäköisyyslaskentaa / tilastotiedettä

  pahimmassa tapauksessa.

- Keskitymme kurssilla *pahimman* tapauksen analyysiin:
  - Helpompaa kuin keskimääräisen tapauksen analyysi
  - koska voimme valita algoritmillemme pahimman mahdollisimman syötteen.
  - Joskus keskimääräiset tapaukset ovat suunnilleen yhtä vaikeita kuin pahimmatkin (lisäysjärjestäminen!).
  - Monilla algoritmeilla pahimmat tapaukset myös yleisiä.

- Tarkastellaan jälleen lisäysjärjestämistä kalvoilta 1.1.

- Merkitään $c_i =$ rivin i suorittamiseen kuluva aika.

  – Se oletetaan *vakioksi:*

    jos rivi i onkin mutkikas (esim. aliohjelmakutsu), niin rivin osat pitää laskea erikseen ja tarkemmin.

  – Sen yksikköä (millisekunteja tms.) ei merkitä

    vaan ajetellaan kaikkien c_i olevan samaa yksikköä.

  – Todellinen seinäkelloaika riippuisi monesta algoritmin ulkopuolisesta tekijästä

    (ohjelmointikieli, laitteen ominaisuudet, koneen muu työkuorma,...)

    mutta meitä kiinnostaa nyt itse *algoritmin oma ajantarve* tällaisista vaihtuvista seikoista riippumattomana suureena.

- Merkitään t_j = kuinka monta kertaa **while**-rivin 5 testi suoritetaan kullakin muuttujan j arvolla.

- Tämä rivi on *keskeinen kontrollikohta* algoritmissa: jokainen

  - **for**-ulkosilmukan

  - **while**-sisäsilmukan

  suorituskerta joutuu suorittamaan myös sen.

- Juuri sellaisten rivien suorituskertojen lukumäärää kannattaa tarkkailla.

- Merkitään n = syötteen määrä

  eli tässä algoritmissa taulukon A pituus length[A].

- Lasketaan kuinka monta kertaa eri rivit suoritetaan:

		aika	kertaa
1	**for** $j \leftarrow 2$ **to** length$[A]$ **do**	c_1	n
2	key $\leftarrow A[j]$	c_2	$n-1$
3	$\triangleright \ldots$	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ **and** $A[i] >$ key	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow$ key	c_8	$n-1$

- Laskemalla ne kaikki yhteen saadaan kokonaisajaksi

$$
\begin{aligned}
T(n) = & c_1 \cdot n + \\
& (c_2 + c_4 + c_8) \cdot (n-1) + \\
& c_5 \cdot \sum_{j=2}^{n} t_j + \\
& (c_6 + c_7) \cdot \sum_{j=2}^{n} (t_j - 1).
\end{aligned}
$$

Parhaassa tapauksessa syötetaulukko A on *valmiiksi* järjestyksessä.

- Silloin jokainen **while**-testi tehdään vain kerran, eli

$$t_1 = t_2 = t_3 = \cdots = 1.$$

- Kokonaisajan lauseke saa muodon

$$\begin{aligned} T(n) &= \underbrace{(c_1 + c_2 + c_4 + c_5 + c_8)}_{a} \cdot n \\ &\quad \underbrace{-(c_2 + c_4 + c_5 + c_8)}_{b} \\ &= a \cdot n + b \end{aligned}$$

  joillakin vakioilla a ja b

  joista emme nyt ole kiinnostuneita.

- $T(n)$ on *lineaarinen* funktio:

  suora viiva, jonka kulmakerroin on a.

- Sanomme siis:

  lisäysjärjestäminen toimii <u>parhaassa tapauksessa lineaarisessa</u> ajassa syötteen pituuteen nähden.

Pahimmassa tapauksessa syötetaulukko A onkin *käänteisessä* järjestyksessä.

- Silloin jokainen **while**-silmukka vie alkionsa $A[j]$ koko matkan paikkaan $A[1]$ saakka, eli

$$t_j = j.$$

- Kokonaisajan lauseke saa nyt muodon

$$\begin{aligned} T(n) = & c_1 \cdot n + \\ & (c_2 + c_4 + c_8 - c_6 - c_7) \cdot (n-1) + \\ & (c_5 + c_6 + c_7) \cdot \sum_{j=2}^{n} t_j. \end{aligned}$$

- Käyttämällä *summakaavaa*

$$\sum_{k=p}^{q} k = \underbrace{\frac{p+q}{2}}_{\text{ka.}} \cdot \underbrace{(q-p+1)}_{\text{lkm.}}$$

saadaan

$$\begin{aligned} \sum_{j=2}^{n} t_j &= \frac{2+n}{2} \cdot (n-1) \\ &= \frac{1}{2}(n^2 + n - 2). \end{aligned}$$

- Kokonaisajan lauseke saa nyt muodon

$$T(n) = a \cdot n^2 + b \cdot n + c$$

joillakin vakioilla a, b ja c.

- $T(n)$ on *neliö(ll)inen* funktio:

  paraabeli.

- Sanomme siis:

  lisäysjärjestäminen toimii pahimmassa tapauksessa neliöisessä ajassa syötteen pituuteen nähden.

- Näin olemme selvittäneet lisäysjärjestämisalgoritmin *oleellisen* aikavaatimuksen eri tapauksissa:

  parhaimmillaan lineaarinen

  pahimmillaan neliöinen.

  Tämä kertoo intuitiivisesti sen, *millaista vauhtia laskentatehon tarve kasvaa syötteen koon n kasvaessa*.

1.3 Funktioiden kertaluokat

- Haluamme puhua funktion $f(n)$ *oleellisesta* kasvunopeudesta *suurilla* arvoilla n.

- Tarkastellaan funktioita $f : \mathbb{N} \to \mathbb{R}$ ja $g : \mathbb{N} \to \mathbb{R}$.

- f kuuluu kertaluokkaan $\mathcal{O}(g)$ jos on olemassa vakiot $d > 0$ ja n_0 siten että kaikilla $n > n_0$ ehto $0 \leq f(n) \leq d \cdot g(n)$ on voimassa.

- Eli f on kertaluokassa $\mathcal{O}(g)$ jos ja vain jos voimme valita

  - jonkin kertoimen d

  - ja kohdan n_0 lukusuoralta

  siten että tämä kohdan jälkeen funktion $f(n)$ kuvaaja pysyttelee funktion $d \cdot g(n)$ kuvaajan alapuolella.

- Tällöin merkitään $f = \mathcal{O}(g)$.

- Intuitiivinen tulkinta: jos f on ohjelman resurssitarvetta kuvaava funktio ja $f = \mathcal{O}(g)$, niin

  - tarpeeksi suurilla syötepituuksilla $n > n_0$
  - toteutuskohtaisiin vakioihin d menemättä

  resurssitarpeiden yläraja on g.

- Palataan lisäysjärjestämiseen:

  parhaassa tapauksessa algoritmi kulutti
  aikaa $a \cdot n + b \leq n \cdot (a + b) = \mathcal{O}(n)$, sillä vakioksi voidaan valita $d = a + b$

  pahimmassa tapauksessa
  $a \cdot n^2 + b \cdot n + c \leq n^2 \cdot (a + b + c) = \mathcal{O}\left(n^2\right)$ sillä vakioksi voidaan valita $d = a + b + c$.

- Näin merkinnässä $f = \mathcal{O}(g)$ funktio g on se osa funktiosta f, joka kasvaa rajuimmin:

  - nouseva suora n rajummin kuin vaakasuora b
  - paraabeli n^2 rajummin kuin suora n.

- Jatkossa tulemme analysoimaan algoritmeja $\mathcal{O}$-merkinnän tarkkuudella.

- $\mathcal{O}$-merkintää käytetään *ylärajan* esittämiseen

  - Jos jonkin algoritmin pahimman tapauksen vaativuus on esim. $\mathcal{O}\left(n^3\right)$, niin

    * se ei välttämättä tarkoita että pahin tapaus toimii ajassa $d \cdot n^3$ jollekin d

    * vaan että algoritmi ei toimi ainakaan huonommin kuin ajassa $d \cdot n^3$.

  - Esim: koska $n^2 = \mathcal{O}\left(n^5\right)$, voidaan (ja on ihan oikein) merkitä että lisäysjärjestäminen on pahimmassa tapauksessa $\mathcal{O}\left(n^5\right)$

    mutta kertooko se tarpeeksi lisäysjärjestämisestä?

  - Mielekkäintä onkin etsiä *pien(in)tä* argumenttifunktiota f jolla algoritmin vaativuus on $\mathcal{O}(f)$.

- Vaihtoehtoisesti voidaan esittää vaativuudelle *ala*rajoja. Tällöin käytetään merkintää Ω.

- $f = \Omega(g)$ jos on olemassa vakiot $d > 0$ ja n_0 siten että kaikilla $n > n_0$ ehto $0 \leq d \cdot g(n) \leq f(n)$ on voimassa.

- Esim: valitsemalla $d \leq a$ huomaamme että

$$\begin{aligned} d \cdot n^2 &\leq a \cdot n^2 \\ &\leq a \cdot n^2 + b \cdot n + c \end{aligned}$$

  eli lisäysjärjestäminen on myös pahimmassa tapauksessa $\Omega(n^2)$.

  - siis ylä- ja alarajat vaativuudelle ovat *samat*

  - eli kyseessä on *tarkka* arvio.

  - Myös tälle tapaukselle on olemassa oma merkintätapa.

- g on sekä ylä- että alaraja f:lle, merkitään $f = \Theta(g)$, jos ja vain jos $f = \mathcal{O}(g)$ ja $f = \Omega(g)$

  - nyt olemassa vakiot d_1 ja d_2 siten että funktio $f(n)$ jää kuvaajien $d_1 \cdot g(n)$ ja $d_2 \cdot g(n)$ *väliin*

  - f käyttäytyy oleellisesti samoin kuin g toisin sanoen f:llä sama *asymptoottinen* kasvunopeus kuin g:llä

- Esim: koska lisäysjärjestämisen pahimman tapauksen

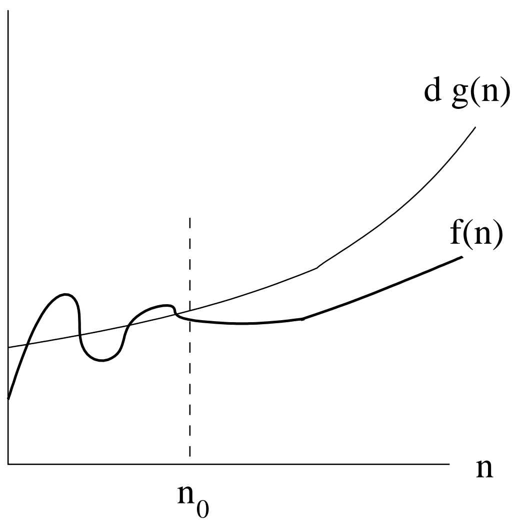
  yläraja on $\mathcal{O}\left(n^2\right)$

  alaraja on $\Omega(n^2)$

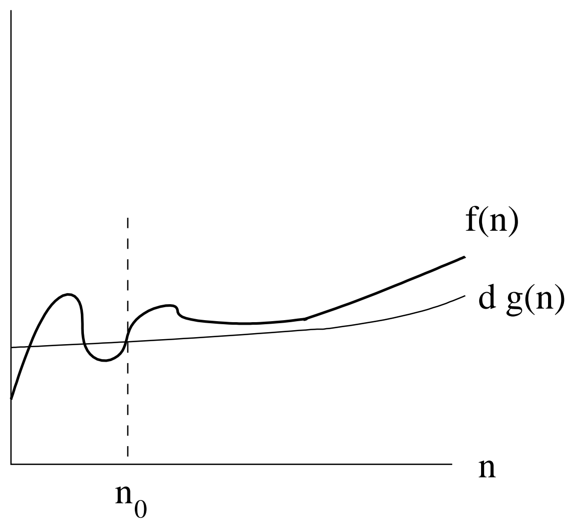
  niin voimme käyttää merkintää $\Theta(n^2)$.

- Seuraavat kuvat valaisevat asymptoottisten merkintöjen eroja.

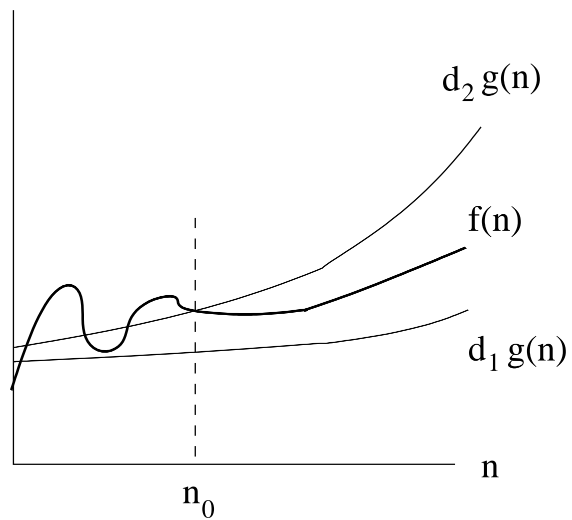
$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$



- T(ällä kurssilla t)ärkeimmät vaativuusluokat (kasvavassa järjestyksessä) syötteen kokoon n nähden ovat

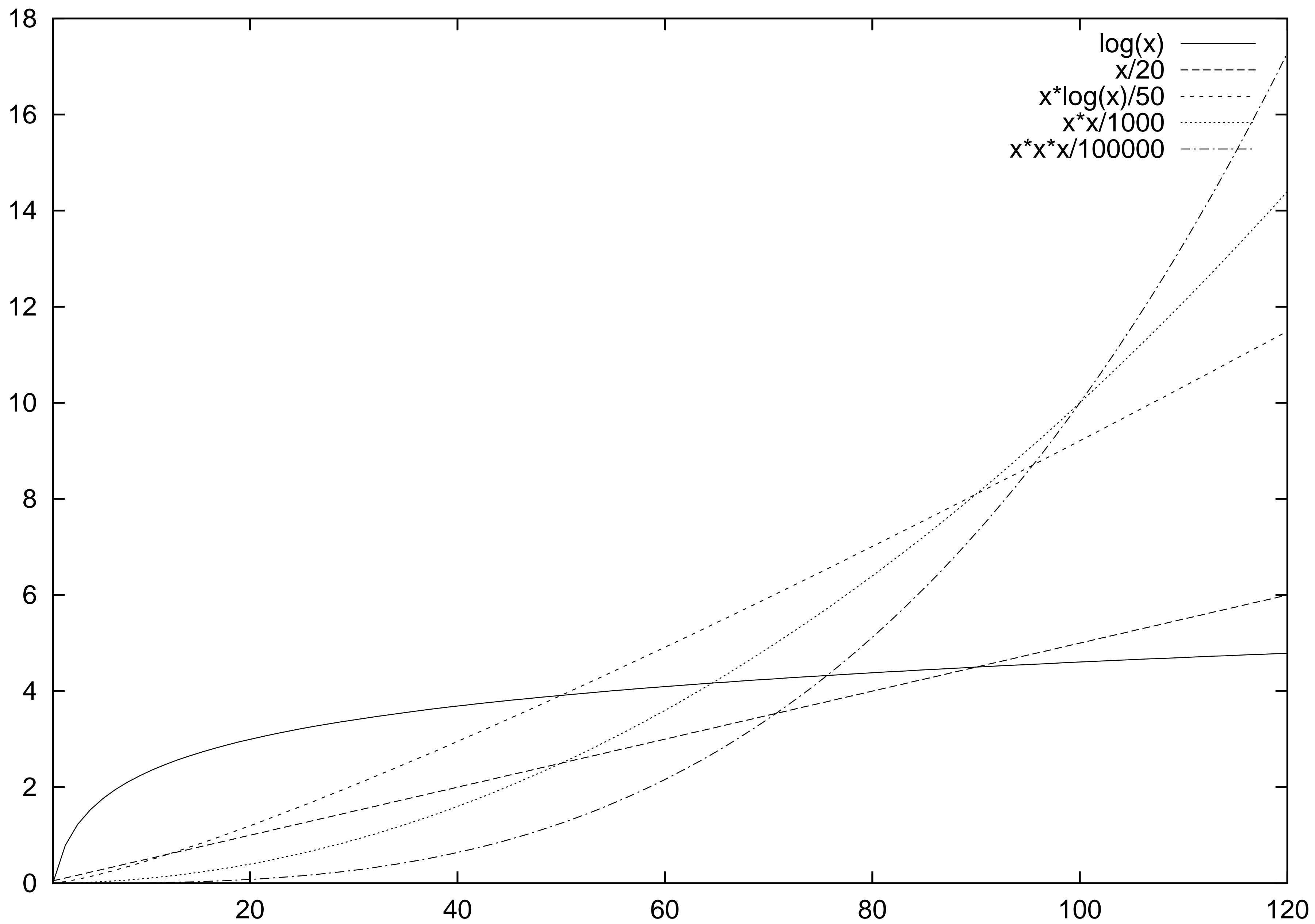
  1. $\mathcal{O}(1)$ (vakio)
  2. $\mathcal{O}(\log(n))$ (logaritminen)
  3. $\mathcal{O}(n)$ (lineaarinen)
  4. $\mathcal{O}(n \cdot \log n)$
  5. $\mathcal{O}\left(n^2\right), \mathcal{O}\left(n^3\right), \ldots$ (polynomi(aali)nen)
  6. $\mathcal{O}(2^n)$ (eksponentiaalinen).

- Seuraava kuva valaisee näiden luokkien suhteita.

  Eksponentiaaliset funktiot on jätetty pois; ne kasvaisivat vielä paljon rajummin.

- Algoritmia pidetään käytännöllisenä vain jos sen aikavaatimus on polynominen.

  (Käytännössä eksponentinkin pitää olla $\lesssim 3$.)


log(x)
x/20
x*log(x)/50
x*x/1000
x*x*x/100000
0
2
4
6
8
10
12
14
16
18
20
40
60
80
100
120

1.3.1 Sääntöjä algoritmien analysointiin

Yksinkertaiset muuttujia käsittelevät perustoimitukset kuten

- sijoituslause $x \leftarrow y$ kun y on yksinkertainen lauseke
- yhden arvon syöttö
- yhden arvon tulostus
- kahden arvon suuruusvertailu
- kahden arvon peruslaskutoimitukset

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

"Perustelu": kutsukaamme yksinkertaisiksi vain sellaisia toimituksia, joihin kuluu syötteen koosta n riippumaton kiinteä määrä suoritettuja konekäskyjä.

Peräkkäiset lauseet $S_1; S_2$ voidaan analysoida seuraavasti:

1. Analysoidaan ne erikseen: olkoon S_1 $\mathcal{O}(f_1(n))$ ja S_2 $\mathcal{O}(f_2(n))$.

   Voimme olettaa että funktiot f_1 ja f_2 eivät saa negatiivisia arvoja.

2. Kokonaisaika on silloin niiden summa

   $$\mathcal{O}(f_1(n)) + \mathcal{O}(f_2(n)) = \mathcal{O}(f_1(n) + f_2(n)).$$

   Perustelu $+$:n siirtämiselle $\mathcal{O}$:n sisään:

   - On d_1 ja n'_1 siten, että S_1 vie korkeintaan $d_1 \cdot f_1(n)$ askelta kun $n \geq n'_1$.
   - Samoin d_2 ja n'_2.
   - Silloin $S_1; S_2$ vie korkeintaan

     $$\underbrace{(d_1 + d_2)}_{d} \cdot (f_1(n) + f_2(n))$$

     askelta kun

     $$n \geq \underbrace{\max(n'_1, n'_2)}_{n_0}.$$

Summat yksinkertaistuvat:

$$\mathcal{O}\left(f_1(n) + f_2(n)\right) = \mathcal{O}\left(\max(f_1(n), f_2(n))\right).$$

Perustelu: $S_1; S_2$ vie korkeintaan

$$2 \cdot \text{hitaampi lauseista } S_1 \text{ ja } S_2$$

askelta — hidastetaan nopeampi yhtä hitaaksi.

Varsinkin: jos $f_2(n) = \mathcal{O}\left(f_1(n)\right)$ niin

$$\mathcal{O}\left(f_1(n) + f_2(n)\right) = \mathcal{O}\left(f_1(n)\right).$$

"Summan rajuimmin kasvava termi" -periaatetta käytettiin jo kalvoilla 1.3.

Ehtolause if E **then** S_1 **else** S_2 vie aikaa

$\mathcal{O}$(ehdon E testauksen viemä aikaa
+
valitun haaran S_1 tai S_2 viemä aika).

Silmukka muotoa

for $i \leftarrow p(n)$ **to** $q(n)$ **do**
    S

voidaan analysoida seuraavasti:

- Muodostetaan rungolle S lauseke $\mathcal{O}(f_S(n,i))$ joka voi riippua nyt

  paitsi syötteen koosta n

  myös nykyisestä kierroksesta i.

- Kääntäen: jos tämä silmukka on toisen silmukan **for** j **. . .** sisällä, niin $p(n,j)$, $q(n,j)$ ja $f_S(n,j,i)$.

- Kokonaisajaksi saadaan summa

$$\mathcal{O}\left(\sum_{i=p(n)}^{q(n)} f_S(n,i)\right).$$

- Tätä summaa *ei saa* max-yksinkertaistaa:

  yhteenlaskettavien lukumäärä ei ole vakio!

- Matematiikassa on sovittu, että *tyhjä summa*

$$\left(\sum_{i=p}^{q} \ldots\right) = 0$$

  - kun $q < p$ eli yhteenlaskettavia ei ole yhtään kappaletta
  - koska 0 on yhteenlaskun $+$ *neutraalialkio* eli se jolla tulos ei muutu.
  - Tämä sopii myös meille tietojenkäsittelijöille...

- Usein $f_S(n, i)$ ei riipu kierroksesta i. Silloin kokonaisaika yksinkertaistuu muotoon

$$\mathcal{O}\left(k \cdot f_S(n)\right)$$

missä

$$k = \max\left(0, q(n) - p(n) + 1\right)$$

on kierrosten lukumäärä.

Silmukka muotoa

> **while** E **do**
> $\quad S$

on yleisessä tapauksessa paljon vaikeampi (jopa mahdoton):

- Tärkeät suureet
  - kierrosten lukumäärä k
  - ehdon E testausaika $f_E(n)$ (tehdään yhteensä $k+1$ kertaa)
  - rungon S suoritusaika $f_S(n)$

  voivat riippua

  sekä syötteen pituudesta n

  että edellisistä kierroksista.

- Jos ne eivät riipu edellisistä kierroksista, niin

$$\mathcal{O}\left(k \cdot (f_E(n) + f_S(n))\right)$$

  askelta.

- Onneksi **while**-silmukoita käytetään tällä kurssilla (ja muussakin ohjelmoinnissa) yleensä

  vaeltamaan eteenpäin tietorakenteessa

  pysähtymään heti halutussa kohdassa eikä vasta tietorakenteen lopussa.

- Esimerkkinä peräkkäishaku:

  Annettuna järjestyksessä oleva taulukko $A[1, \ldots, n]$ ja luku x. Onko luku taulukossa?

  1 $i \leftarrow 1$
  2 **while** $i \leq n$ **and** $A[i] < x$ **do**
  3 $\quad i \leftarrow i + 1$
  4 **return** ($i \leq n$ **and** $A[i] = x$)

- Tällaisia **while**-silmukoita voi käsitellä suoraan:

$$\begin{aligned} k &= \mathcal{O}(n) \\ f_E(n) &= \mathcal{O}(1) \\ f_S(n) &= \mathcal{O}(1) \\ \text{yhteensä} &= \mathcal{O}(n)\,. \end{aligned}$$

Ei-rekursiivinen aliohjelma voidaan analysoida seuraavasti:

Periaatteessa ajatellaan, että jokainen aliohjelmakutsu korvataan ensin aliohjelman lähdekoodilla:

Kutsusta $r(a_1,\ldots,a_m)$ tulee

1	$p_1 \leftarrow a_1$
	$\vdots$
m	$p_m \leftarrow a_m$
$m+1$	S

missä

- $p_1,\ldots,p_m$ ovat parametrien nimet
- S on runko

aliohjelman r koodissa.

Käytännössä rungon S analyysi toistuisi monta kertaa

joten kannattaa muodostaa ensin erillinen lauseke

$$f_S(n, p_1\text{:n koko}, \ldots, p_m\text{:n koko}).$$

1.3.2 Esimerkki: Kuplajärjestäminen

```
1    for i ← n downto 2 do
2        for j ← 1 to i − 1 do
3            if A[j] > A[j + 1] then
4                ▷vaihdetaan A[j] ↔ A[j + 1]
5                apu ← A[j]
6                A[j] ← A[j + 1]
7                A[j + 1] ← apu
```

Invariantti: *Taulukon osa $A[i+1,\ldots,n]$ on järjestyksessä ja järjestetyn osan alkiot vähintään yhtä suuria kuin osan $A[1,\ldots,i]$ alkiot.*

- Alussa $i = n$ eli voimassa itsestään (loppuosa tyhjä).

- Invariantti pysyy totena jokaisen ulomman **for**-lauseen rungon suorituksen jälkeen:

  1. ensin kohtaan $A[i]$ haetaan alkuosan suurin alkio

  2. sitten indeksin i arvo vähenee yhdellä.

- Lopuksi $i = 1$ eli invariantista seuraa:
  - taulukon osa $A[2, \ldots, n]$ järjestyksessä ja järjestetyn osan alkiot vähintään yhtä suuria kuin paikan $A[1]$ alkio
  - siis koko taulukko järjestyksessä.

- Rivien 2–7 sisäsilmukan analyysi:
  - suoritetaan i kertaa
  - runko (eli rivit 3–7) vie $\mathcal{O}(1)$ askelta
  - siis $\mathcal{O}(i)$ askelta.

- Ulkosilmukan analyysi:

$$\mathcal{O}\left(\sum_{i=2}^{n} i\right) = \mathcal{O}\left(n^2\right)$$

  kalvojen 1.1 summakaavalla.

- Huom: toisin kuin kalvojen 1.1 lisäysjärjestämisellä aikavaativuus on nyt sama *kaikissa tapauksissa*.

1.3.3 Esimerkki: Binäärihaku

Annettuna järjestyksessä oleva taulukko $A[1,\ldots,n]$ ja luku x. Onko luku taulukossa?

```
1    vasen ← 1
2    oikea ← n
3    found ← false
4    while vasen ≤ oikea and not found do
5        keski ← (vasen + oikea)/2
6        found ← A[keski] = x
7        if A[keski] > x then
8            oikea ← keski − 1
9        else
10           vasen ← keski + 1
```

Konvergentti: Silmukka pysähtyy, koska jokaisella toistolla

joko etsitty alkio löytyi

tai etsittävä alue A[vasen ... oikea] kutistuu.

Invariantti: jos x on taulukossa niin se on alueella A[vasen...oikea] ja found $= \mathbf{true}$ jos ja vain jos x löytyi jo.

- Tosi alussa sillä vasen $= 1$, oikea $= n$ ja found $= \mathbf{false}$.
- Säilyy totena toistolauseen suorituksessa:
  - jos x löytyy, niin asetetaan found $= \mathbf{true}$
  - jos A[keski] $> x$ niin myös kaikilla $j >$ keski on $A[j] > x$ eli ei tarvitse etsiä kohdan keski takaa ja voidaan asettaa oikea$\leftarrow$keski -1
  - **else**-haara vastaavasti.
- lopuksi joko vasen $>$ oikea ja found $= \mathbf{false}$ eli etsittyä ei löytynyt, tai $x = A$[keski].

- Merkitään $T(k) =$ binäärihaun pahimman tapauksen aikavaativuus silloin kuin taulukosta on vielä tutkimatta k paikkaa.

  Siis $k =$ oikea $-$ vasen $+1$.

- Pahin tapaus on, että x on suurempi kuin mikään taulukon A luku:
  - Silmukka jatkaa loppuun saakka.
  - Alue puolittuu joka kierroksella.

- Oletetaan yksinkertaisuuden vuoksi että taulukon A pituus n on jokin kahden potenssi, eli $n = 2^j$ jollain kokonaisluvulla j.
  - Alueen puolitus helpottuu: eksponentti j vähenee yhdellä.
  - Tämä yksinkertaistus on sallittua:
    * vaikka kaikki syötteen pituudet n eivät olekaan aivan yhtä vaikeita
    * niin on olemassa vaikka kuinka suuria tällaisia vaikeita pituuksia.

- T on määritelty seuraavasti *rekursioyhtälönä:*

$$
\begin{aligned}
T(k) &= T(k/2) + c \qquad \text{kun } k > 1 \\
T(1) &= c
\end{aligned}
$$

missä c on jokin riittävän suuri vakio.

- Binäärihaun aikavaatimus saadaan laskemalla rekursiokaava auki

$$
\begin{aligned}
T(n) &= T(n/2) + c \\
&= (T(n/4) + c) + c \\
&= (T(n/8) + c) + c + c \\
&= T(n/2^3) + 3 \cdot c \\
&\ \ \vdots \\
&= T(n/2^j) + j \cdot c \\
&= (j + 1) \cdot c
\end{aligned}
$$

ja koska $n = 2^j$ niin $j = \log_2(n)$ eli

$$
\begin{aligned}
&= (\log_2(n) + 1) \cdot c \\
&= \mathcal{O}(\log_2(n)) \\
&= \mathcal{O}(\log(n))
\end{aligned}
$$

koska logaritmin kantaluvulla ei ole väliä $\mathcal{O}$-analyysissä.

- Samaa rekursioyhtälötekniikkaa käytetään myös *rekursiivisten* aliohjelmien analyysissä:
  - Haara $T(1)$ vastaa rekursiokutsua, joka ei kutsu itseään.
  - Muu haara $T(k)$ vastaa rekursiokutsua, jossa
    - syötteen pituus on k
    - rekursiivinen kutsu syötteen pituudella $k' < k$ analysoidaan termiksi $T(k')$.

- Nyt voimme sanoa missä mielessä binäärihaku on *parempi algoritmi* samaan tehtävään kuin kalvojen 1.3 peräkkäishaku:
  - binäärihaun pahin tapaus $\mathcal{O}(\log(n))$ on *aidosti parempi* kuin $\mathcal{O}(n)$
  - koska $\log n = \mathcal{O}(n)$ mutta $n \neq \mathcal{O}(\log(n))$
  - eli tarpeeksi suurilla n binäärihausta tulee väistämättä nopeampi kuin peräkkäishausta.

1.4 Logaritmeista

- Lukion matematiikassa määriteltiin (luonnollinen) logaritmi eksponenttifunktion

$$x \mapsto e^x$$

käänteisfunktiona, missä *kantaluku*

$$e = 2.718281828459045\ldots$$

on ns. Neperin vakio. Siis

$$\ln(y) = \text{"se } x \text{ jolla } e^x = y\text{"}.$$

- Tietojenkäsittelyssä kantaluku onkin tavallisesti **2**:

log(n) = "kuinka monta kertaa lukua $n \geq 1$ on jaettava luvulla 2, jotta tulos olisi ≤ 1?"

```
1  k ← 0
2  while n > 1
3       do k ← k + 1
4          n ← n/2
5  return k
```

- Tätä ajatusta käytimme kalvojen 1.3.3 binäärihaun analyysissa:

  1. puolitimme etsittävän alueen A[vasen . . . oikea] pituuden joka kierroksella
  2. kunnes pituus oli < 1.

- Sama puolittamisidea toistuu
  - eri muodoissa
  - monissa tehokkaissa algoritmeissa
  - ja tietorakenteissa.

- Tietojenkäsittelijän näkökulmasta logaritmi siis vastaa kysymykseen:

  ”Kuinka monta kertaa pitää soveltaa puolittamisideaa, ennen kuin päästään niin pieneen tapaukseen, ettei sitä enää tarvita?”

- Otetaan esimerkiksi tulo $a \cdot b$:
  - Kun sitä puolitetaan toistuvasti, niin
    1. ensin puolitetaan k_a kertaa osaa a, kunnes se on ≤ 1
    2. sitten k_b kertaa osaa b, kunnes sekin on ≤ 1.

    Siis yhteensä puolitettiin

    $$k = k_a + k_b$$

    kertaa.
  - Näin nähtiin logaritmien laskusääntö

    $$\log(a \cdot b) = \log(a) + \log(b).$$

- Toinen tapa ajatella 2-kantaista logaritmia log(n) on:

  "Kuinka monta bittiä on luvussa n?"

  Sen bittien tulostus takaperin (pienimmästä suuurimpaan) on:

```
1   while n > 0
2         do if n on pariton
3               then tulosta '1'
4                    n ← n − 1
5               else tulosta '0'
6            n ← n/2
```

- Tällä ajattelutavalla havaitaan, että log(n) *kasvaa, mutta hyvin hitaasti:*

n	bittejä
1 000	10
1 000 000	20
1 000 000 000	30
1 000 000 000 000	40

1.5 Abstrakti tietotyyppi joukko

- Usein ohjelma ylläpitää suoritusaikanaan jotain *joukkoa tietoalkioita*.

  Esimerkiksi kalvojen 1 puhelinluettelo on nimi-numero-parien joukko.

- Joukon tietoalkiot muodostuvat *tietueista*.

- Tietueen yksi kenttä on ns. *avain*.

- Tietueita voidaan hakea ja järjestää avaimen perusteella.

  – Esimerkiksi puhelinluettelossa nimi on avain.

  – Vertaa tietokantataulu ja sen avain.

- Muu tieto voi olla

  – tietueen muissa kentissä, tai

  – viitteen takana "satelliitti-tietueena".

data1
data2
data3
key
key
data1
data2
data3

- Joukon S käsittelyssä voidaan tarvita esimerkiksi seuraavia operaatioita:

  search(S**,**k**)** etsii avainta k: palautetaan viite x sellaiseen tietueeseen, jonka avainkentässä on k.

  insert(S**,**x**)** lisää joukkoon sen tietueen, johon x viittaa

  delete(S**,**x**)** poistaa tietueen johon x viittaa

  (jos siis halutaan poistaa *avain* k, niin on ensin tehtävä **search**, jotta saadaan x)

  min/max(S**)** palauttaa viitteen joukon siihen tietueeseen, jonka avain on pienin/suurin

  pred/succ(S**,**x**)** palauttaa viitteen siihen tietueeseen, jonka avain on järjestyksessä lähin pienempi/suurempi kuin viitatun tietueen x avain.

- Jos kyseistä tietuetta ei ole, niin palautetaan NIL.

- Puhelinluettelonkin operaatiot (paitsi **list**) sisältyvät näihin joukko-operaatioihin.

- Näin määriteltynä joukko on *abstrakti tietotyyppi*, joka toteuttamiseen on useita vaihtoehtoisia tietorakenteita:
  - taulukko (edellyttäen, että

    joko tiedämme etukäteen joukon maksimikoon

    tai osaamme pidentää taulukkoa tarvittaessa — kalvot 4.4)
  - linkitetty lista — kalvot 2.4
  - hakupuu, tasapainotuksella tai ilman — kalvot 3
  - hajautusrakenne — kalvot 4
  - keko — kalvot 5
  - ...

- Joukon operaatioiden tehokkuus riippuu suuresti siitä minkä tietorakenteen avulla joukko on toteutettu.

  – Eri tietorakenteilla eri operaatiot ovat nopeita/hitaita:

    lista1 on epäjärjestyksessä

    lista2 pidetään järjestyksessä avaimen mukaan.

  – Siis toteutustavan valinnassa on ratkaisevaa se mitä operaatioita joukkoa *käyttävä* ohjelma tarvitsee eniten.

- Suuri osa kurssista käsittelee

  – joukon toteutukseen sopivia tietorakenteita

  – ja niiden operaatioiden analysointia

  – jonka tulosten perusteella eri toteutuksien välillä voi valita eri tilanteissa.

	lista1	lista2	tasap.puu	hajautusrak
search	O(n)	O(n)	O(log n)	O(1)
insert	O(1)	O(n)	O(log n)	O(1)
delete	O(1)	O(1)	O(log n)	O(1)
min	O(n)	O(1)	O(log n)	O(n)
max	O(n)	O(1)	O(log n)	O(n)
succ	O(n)	O(1)	O(log n)	O(n)
pred	O(n)	O(1)	O(log n)	O(n)

2 Pino, jono ja lista

- Ennen kuin menemme varsinaiseen tietorakenteeseen **lista** (kalvot 2.4), opiskelemme kahta konkreettista tietorakennetta:

  pinoa (kalvot 2.1)

  jonoa (kalvot 2.2).

- Pino ja jono eivät toteuta kalvojen 1.5 joukko-operaatioita

  vaan ne toimivat kuten pino ja jono toimivat normaalielämässä.

- Pino ja jono ovat kuitenkin tärkeitä rakenneosasia monissa algoritmeissa

  kuten tulemme myöhemmin näkemään.

2.1 Pino

- Last-in-first-out (LIFO) -periaate.

- Pinon S operaatiot:

  push(S**,**x**)** lisäää pinon *päälle* vielä sen tietueen, johon x viittaa

  pop(S**)** poistaa ja palauttaa pinosta sen *päällimmäisen* tietueen

  empty(S**)** tutkii, onko pino tyhjä vaiko ei.

- Oletetaan seuraavassa yksinkertaisuuden vuoksi että pinoon talletettavat tietueet sisältävät vain avainkentän:

  push(S**,**k**)** lisää pinon päälle vielä tietoalkion k

  pop(S**)** poistaa ja palauttaa pinosta sen päällimmäisen tietoalkion.

- Jos pinon koolle tiedetään ennakolta yläraja n, niin pino voidaan toteuttaa helposti taulukon avulla:
  - Talletetaan pinon alkiot tarpeeksi pitkään taulukkoon $S[\underline{1} \ldots n]$.
  - Talletusjärjestys:

$$\begin{aligned} S[1] &= \text{pinon pohjimmainen alkio} \\ S[2] &= \text{pohjimmaisen päällä oleva alkio} \\ S[3] &= \text{edellisen päällä oleva alkio} \\ &\vdots \\ S[\text{top}] &= \text{pinon päällimmäinen alkio} \end{aligned}$$

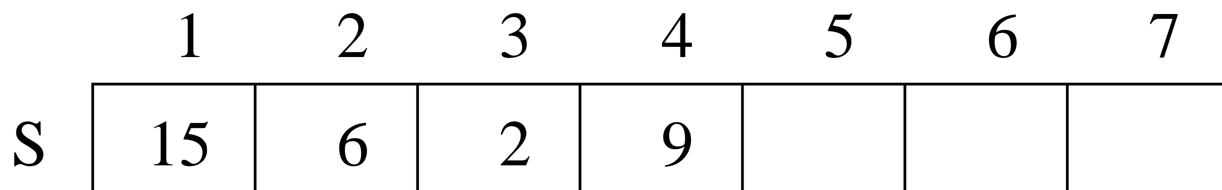
    missä koko pinon attribuutti

$$\begin{aligned} \text{top}[S] &= \text{pinossa olevien alkioiden määrä} \\ &= \text{päällimmäisen alkion indeksi.} \end{aligned}$$

  - Huomaa: *muuttujaan* top liittyy siis oma invariantti

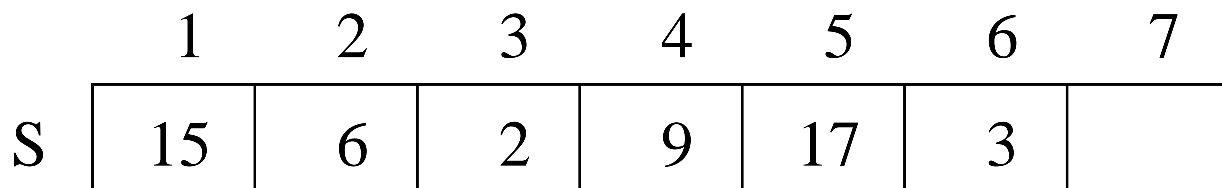
    josta huolehditaan operaatioita toteutettaessa.
  - Alussa siis $\text{top}[S] = 0$ ja pino tyhjä.

push(S,15); push(S,6); push(S,2); push(S,9)



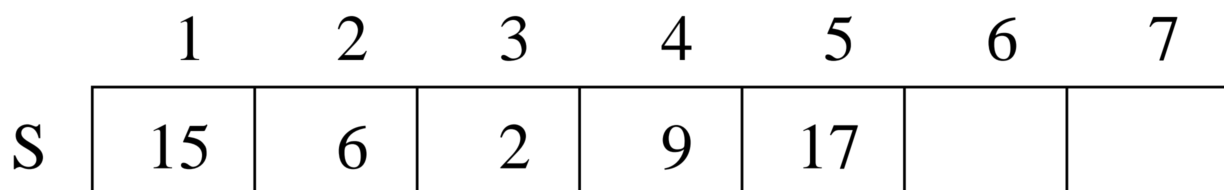
top[S] = 4

push(S,17)
push(S,3)



top[S] = 6

pop(S)



top[S] = 5

- Pino-operaatioiden toteutus:

  empty(S)
  return top[S] $= 0$

  push(S, k)
  top[S] $\leftarrow$ top[S] $+ 1$
  S[top[S]] $\leftarrow k$

  pop(S)
  top[S] $\leftarrow$ top[S] $- 1$
  return S[top[S] $+ 1$]

- Toteutus olettaa käyttävältä ohjelmalta:

  – Operaatiota pop ei kutsuta, jos pino on tyhjä.

  – Operaatiota push ei kutsuta, jos pino on täysi:

    full(S)
    return top[S] $= n$

  – Normaalisti operaatio full ei kuitenkaan kuulu pinon operaatiovalikoimaan.

- Abstraktin tietotyypin toteutusperiaatteet:

  Täysi suojautuminen: toteutus lupaa käsitellä sisäisesti kaikki virhetilanteet.

  \+ Totetusta voi käyttää huolettomasti.

  − Virheentarkistuskoodi joudutaan suorittamaan aina operaatiota käytettäessä — vaikka käyttävä ohjelma takaisikin, ettei virhettä voi syntyä.

  "Design by Contract": *Jos* tietotyyppiä käytetään "sopimuksen mukaan", niin sekin toimii "sopimuksen mukaan".

  − Muunkin ohjelman on noudatettava sopimusehtoja.

  − Ei täysin modulaarista: jos ehdot ovat tiukat, niin muu ohjelma ja juuri tämä tietotyypin toteutustapa tulevat liitetyiksi toisiinsa.

  \+ Virheentarkastus voidaan välttää.

  Edelliset push- ja pop-käyttöehdot ovat esimerkki tällaisesta sopimuksesta.

- Kaikki pino-operaatiot sisältävät vain saman vakiomäärän komentoja riippumatta siitä kuinka monta alkiota pinossa on.

- Operaatiot ovat siis vaativuudeltaan $\mathcal{O}(1)$.

- Pinon käyttö algoritmissa ei siis kasvata sen oleellista aikavaatimusta.

2.2 Jono

- first-in-first-out (FIFO) -periaate
- Operaatiot jonolle Q:

  enqueue(Q,k) lisää tietoalkion k jonon Q *viimeiseksi*

  dequeue(Q) poistaa ja palautttaa jonosta Q sen *ensimmäisen* tietoalkion

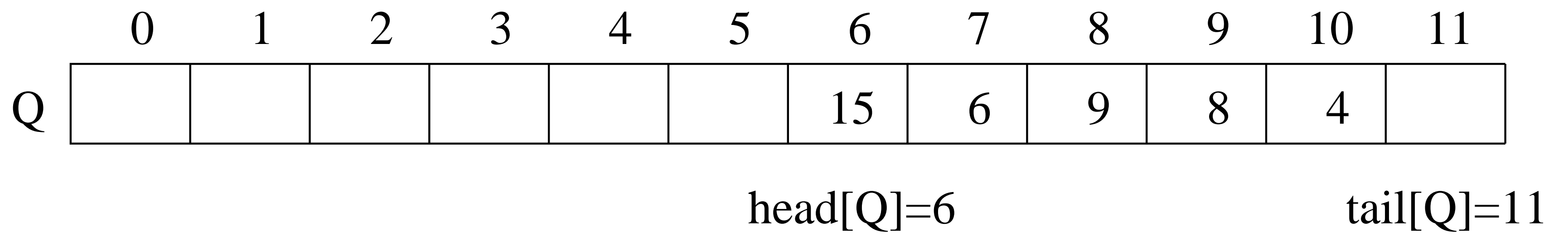
  empty(Q) tutkii onko jono Q tyhjä vaiko ei.

- Jälleen, jos jonon koon yläraja n tiedetään ennakolta, toteutus onnistuu taulukon avulla:
  - Talletetaan jonon alkiot taulukkoon $Q[\underline{0} \ldots n]$.
    - Taulukkoon varataan siis 1 paikka *enemmän* kuin tarpeen!
    - Merkitään varatun taulukon pituutta $m = \text{length}[Q] = n + 1$.
  - Jonolla on attribuutit

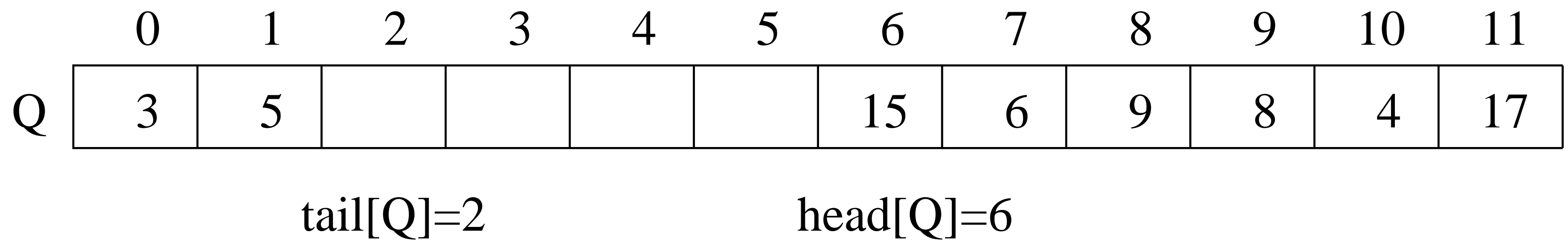
$$\begin{aligned} \text{head}[Q] &= \text{ensimmäisen alkion indeksi} \\ \text{tail}[Q] &= \text{seuraavan vapaan paikan ind.} \end{aligned}$$

  - Niiden mukainen talletusjärjestys on

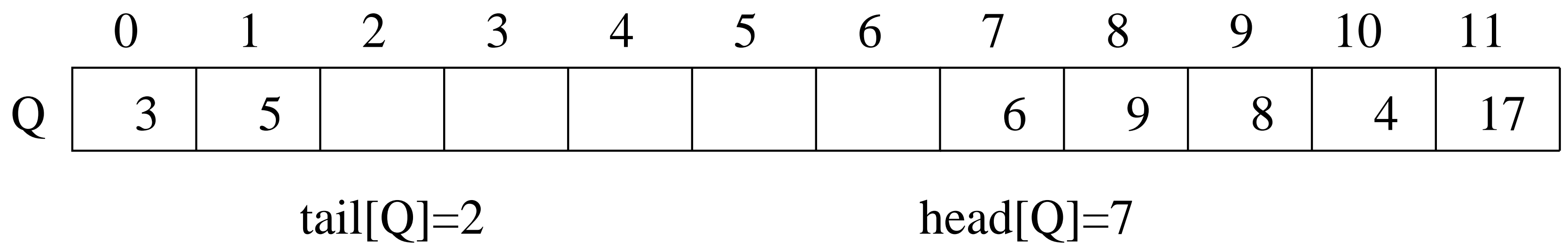
$$\begin{aligned} Q[\text{head}] &= \text{1. alkio} \\ Q[(\text{head}+1) \bmod m] &= \text{2. alkio} \\ Q[(\text{head}+2) \bmod m] &= \text{3. alkio} \\ &\vdots \\ Q[\text{tail}-1] &= \text{viimeinen alkio.} \end{aligned}$$



enqueue(Q,17); enqueue(Q,3); enqueue(Q,5)



dequeue(Q)



- Operaatioiden toteutukset:

  empty(Q)
      return head[Q] $=$ tail[Q]

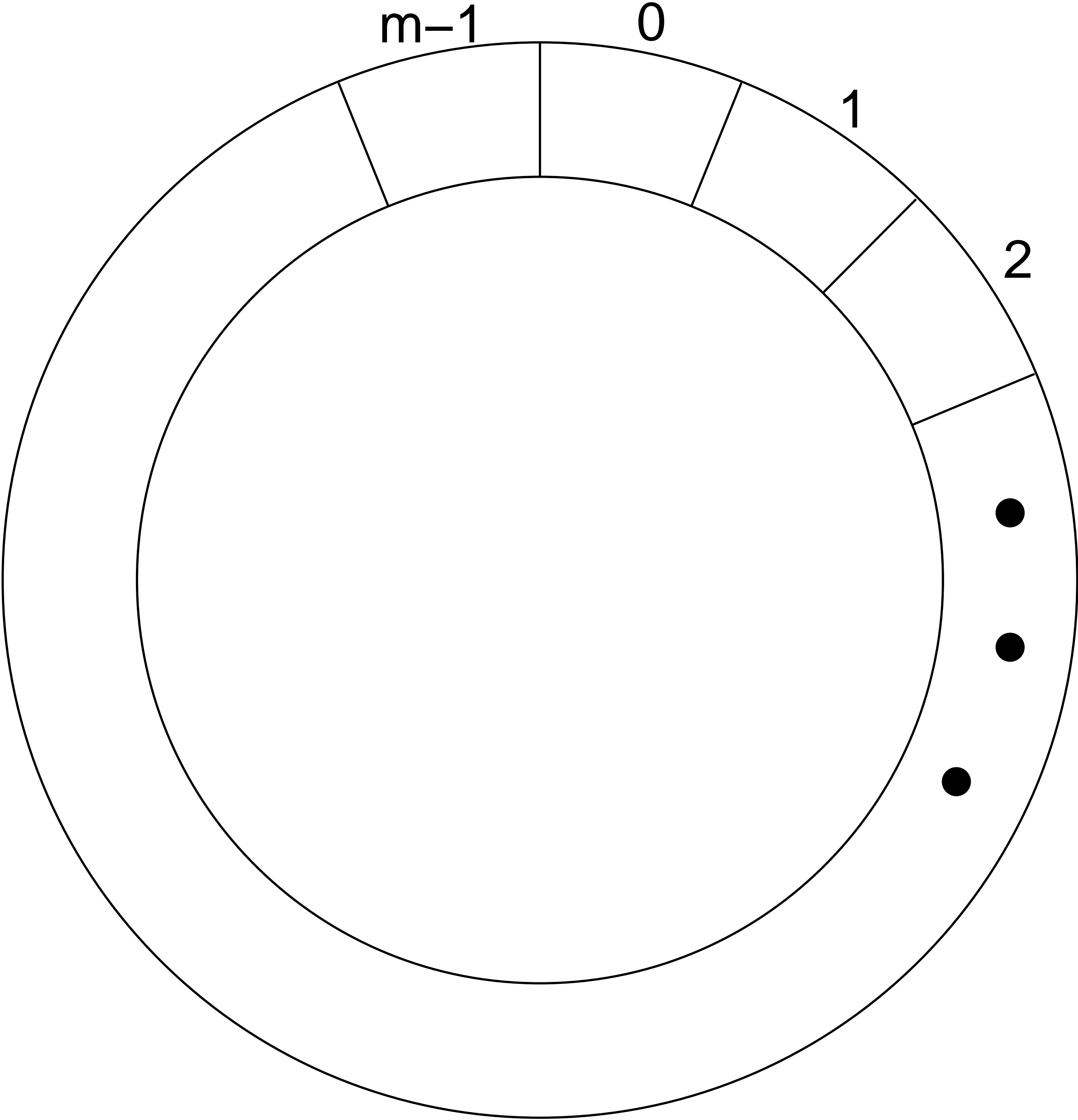
  enqueue(Q,k)
      Q[tail[Q]] $\leftarrow k$
      tail[Q] $\leftarrow$ (tail[Q] $+ 1$) mod m

  dequeue(Q)
      $x \leftarrow Q$[head[Q]]
      head[Q] $\leftarrow$ (head[Q] $+ 1$) mod m
      return x

- Jakojäännösoperaattori mod tekee taulukosta *kehän:* ensimmäisestä paikasta $Q[0]$ tulee viimeisen paikan $Q[m-1]$ seuraaja.

- Sovimme jälleen, että
  - tyhjästä jonosta ei poisteta
  - täyteen jonoon ei lisätä

  alkiota.


m–1
0
1
2

- Mikä on operaatioiden aikavaativuus?
- Miksi **empty(Q)** toimii?
- Mihin tarvittiin lisäpaikkaa?
- Miten tunnistetaan täysi jono?

2.3 Linkitetyt rakenteet

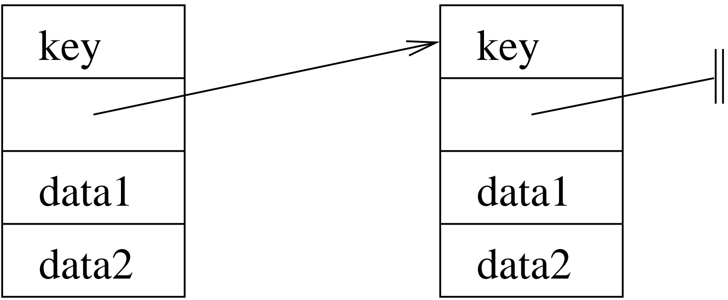
- Jos emme tiedä mikä on pinoon tai jonoon talletettavien alkioiden maksimimäärä, ei taulukkototeutusta voi käyttää.

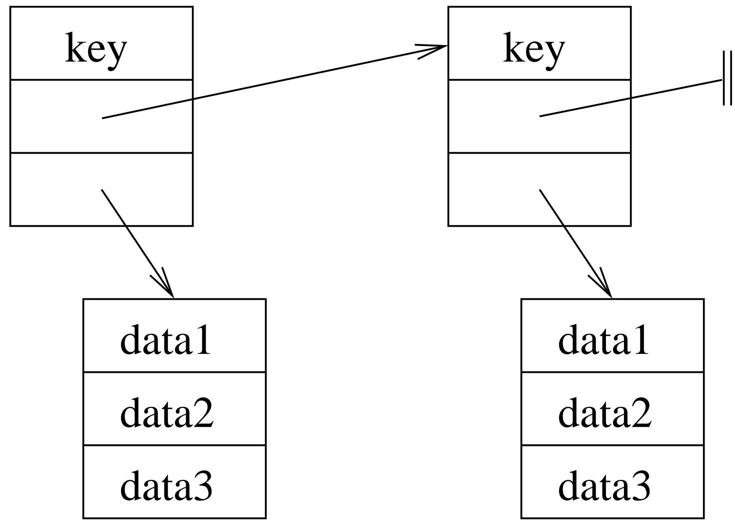
- Ratkaisu saadaan lisäämällä tietoalkiot tallentaviin tietueisiin kenttä joka sisältää *viitteen* johonkin toiseen tietoalkioon.

- Esimerkiksi tietue joka sisältää
  - avaimen
  - viitteen seuraavaan tietueeseen
  - tähän avaimeen liittyvän muun datan

    tai

    viitteen satelliittitietueeseen, jossa on kyseinen muu data.


key
data1
data2
key
data1
data2


key
key
data1
data2
data3
data1
data2
data3

- *Huomaa:* NIL on viite joka ei viittaa mihinkään, kuvassa ⊩

- Kalvojen 2.1 pino voidaan toteuttaa linkitettynä rakenteena seuraavasti:
  - Pinossa oleva tieto talletetaan tyyppiä *pinosolmu* oleviin tietueisiin.

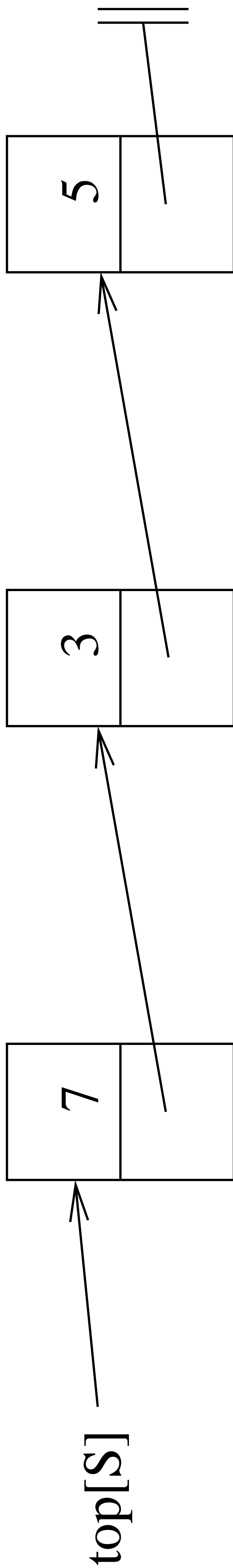
    Sellaisessa tietueessa x on kentät
    - key[x] = pinoon talletettu tietoalkio
    - next[x] = viite tietueen x alla olevaan samanlaiseen tietueeseen

      tai

      NIL jos x on pinon pohjimmainen tietue.
  - Koko pinolla S on attribuutti top[S] = viite päällimmäiseen pinosolmuun

    tai

    NIL jos pino S on tyhjä.

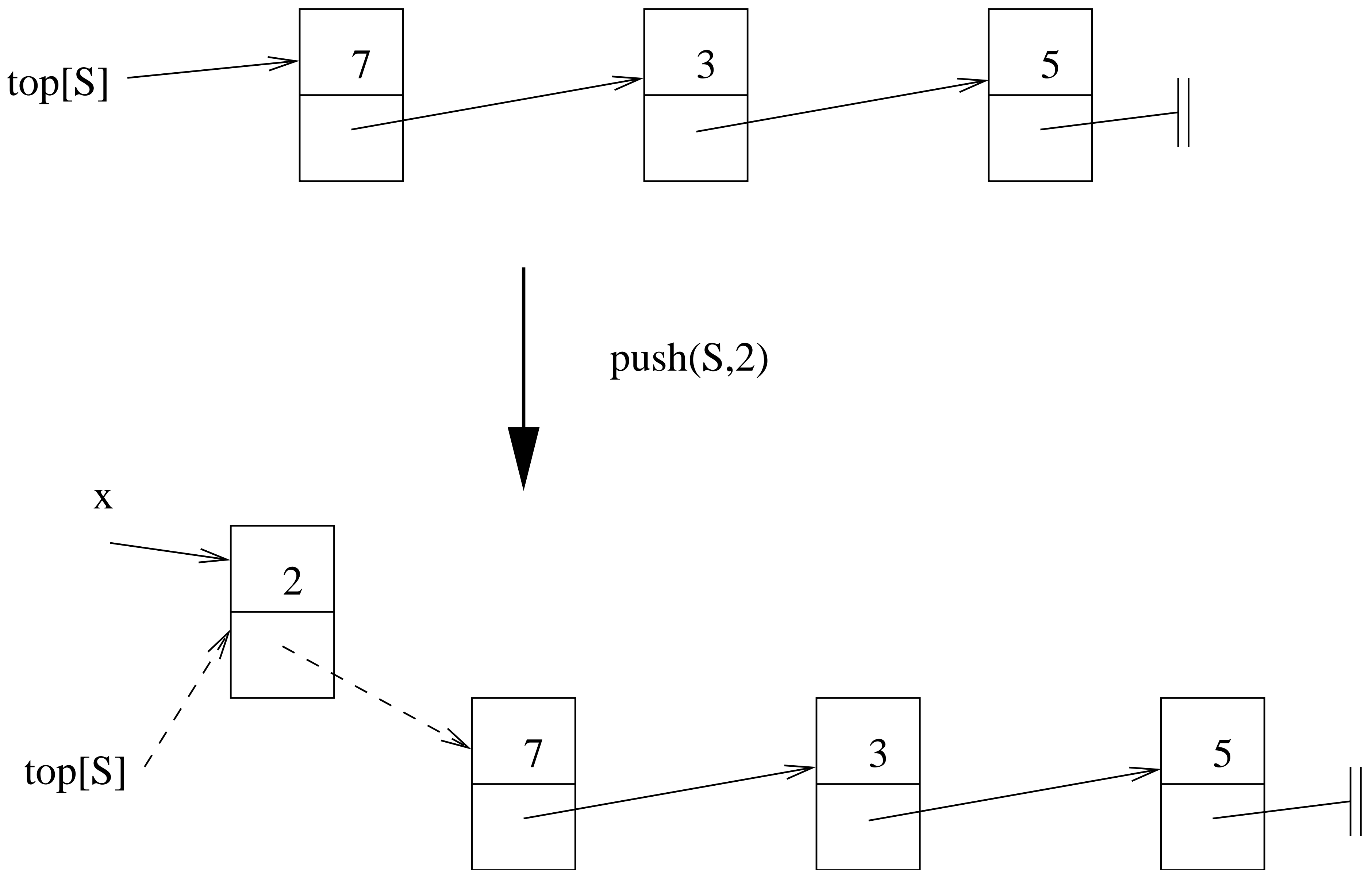

5
3
7
top[S]

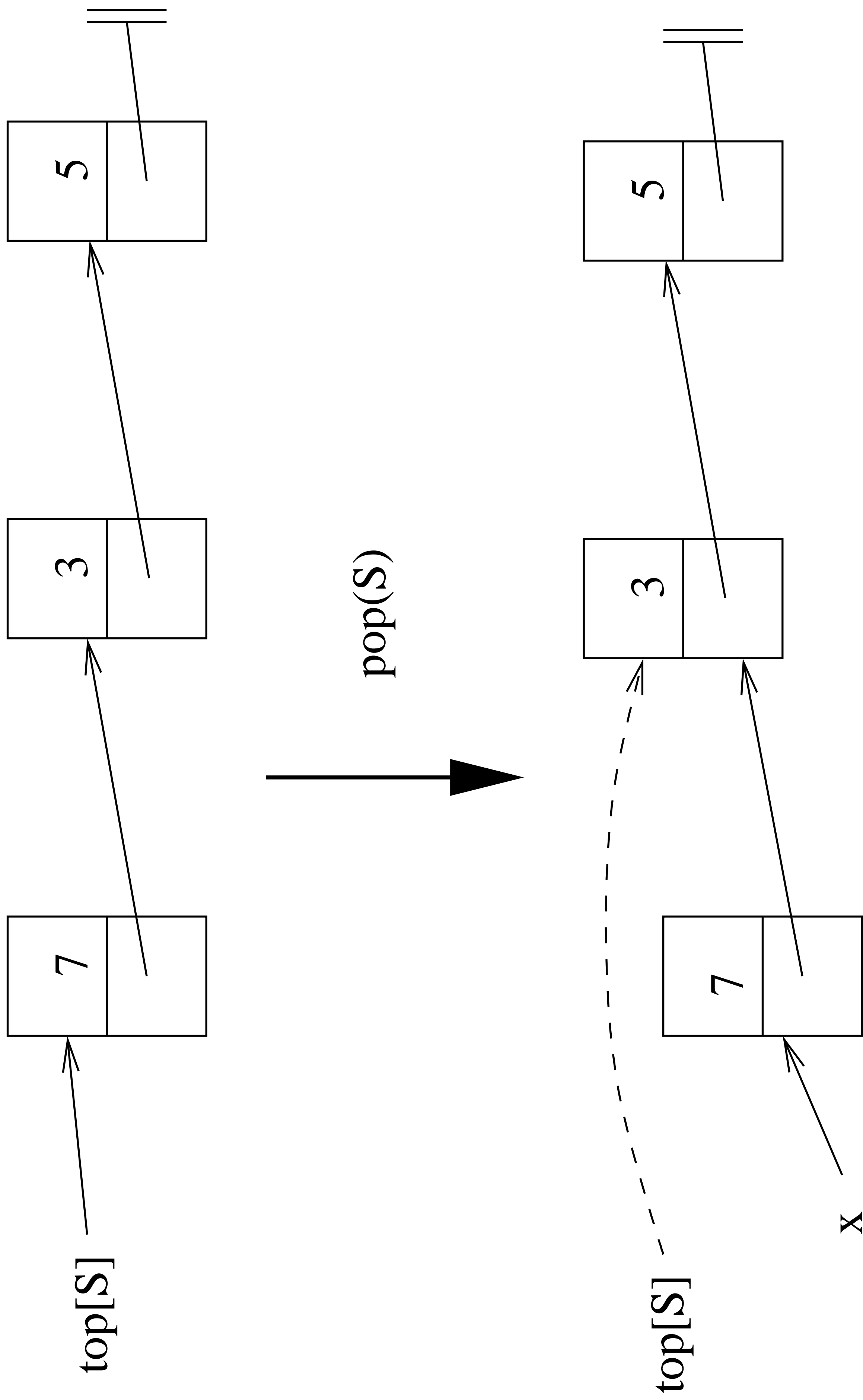
```
empty(S)
    return top[S] = NIL

push(S,k)
    x ← new pinosolmu
    key[x] ← k
    next[x] ← top[S]
    top[S] ← x

pop(S)
    x ← top[S]
    top[S] ← next[x]
    return key[x]
```

- Lisätään pseudokoodiin $\mathcal{O}(1)$ operaatio

  new τ

  - Se palauttaa viitteen sellaiseen muistialueeseen, johon voi tallettaa tyyppiä τ olevan arvon
  - Tämä alue on erillään kaikesta sitä ennen käytetystä muistista.
  - Varaustyön tekee käyttöjärjestelmä.


top[S]
7
3
5
push(S,2)
x
2
top[S]
7
3
5


top[S]
7
3
5
pop(S)
top[S]
x
7
3
5

- *Huomaa:* operaation pop jälkeen sen paikallisen muuttujan x viittaama vanha päällimmäinen tietue jää ”roskaksi”:
  - siihen ei tule enää yhtään viitettä
  - joten ohjelma ei enää näe sitä
  - vaikka se on yhä varattuna ohjelmalle.

- Javassa on *automaattinen roskankeruu*, joka huolehtii sellaisen muistin *vapauttamisesta*.

- Monissa vanhemmissa ohjelmointikielissä ohjelm(oij)an pitää itse vapauttaa varaamansa muisti, kun sitä ei enää tarvita.

  Muuten suoritus täyttää hiljalleen koneen koko muistin.

- Esimerkiksi ohjelmointikielessä `C` tällainen *dynaamisen* muistin varaus/vapautusfunktiopari on `malloc`/`free`.

- Selvästikin pino-operaatiot ovat jälleen vaativuudeltaan $\mathcal{O}(1)$ myös linkitettynä toteutuksena.

- Lisäksi olemme vapautuneet etukäteen asetetusta kiinteästä ylärajasta n pinon koolle.

  Vain muistin määrä on rajana.

- Toisaalta jouduimme olettamaan **new**-mekanismin.

- Myös kalvojen 2.2 jono voidaan toteuttaa linkitettynä rakenteena siten että jono-operaatiot vaativat vakioajan.

  Miten?

2.4 Lista

- Linkitetyn listan avulla voimme helposti toteuttaa kalvoilla 1.5 esitellyn abstraktin tietotyypin joukko.

- Linkitetyistä listoista on monia eri variaatioita.

- Tarkastelemme niistä ensin *molempiin suuntiin linkitettyä listaa*.
  - Nämä suunnat ovat ”*eteen*päin” ja ”*taakse*päin”.
  - Listassa tieto on siis *peräkkäin*.

- Toteutetaan lista kalvojen 2.3 linkitettynä rakenteena.

- Listassa oleva tieto talletetaan tyyppiä *listasolmu* oleviin tietueisiin.

  Sellaisen tietueen x kentät ovat

  - key[x] = tähän solmuun talletettu tietoalkio
  - next[x] = viite seuraavaan tietueeseen

    tai

    NIL jos x on listan viimeinen tietue.
  - prev[x] = viite edelliseen tietueeseen

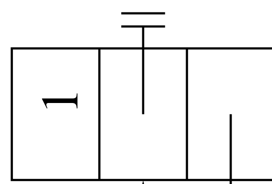
    tai

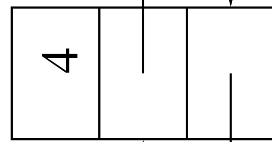
    NIL jos x on listan ensimmäinen tietue.

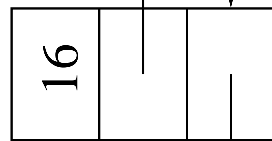
- Koko listalla L on attribuutti head[L] = viite ensimmäiseen tietueeseen

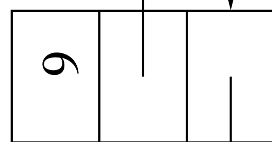
  tai

  NIL jos lista L on tyhjä.


1


4


16


9

head[L]

- Hae listalta L viite ensimmäiseen tietueeseen x, jonka key$[x] = k$.

```
search(L,k)
    x ← head[L]
    while x ≠ NIL and k ≠ key[x] do
        x ← next[x]
    return x
```

Jos sellaista x ei ole, niin palautetaan NIL.

- Operaatiossa search verrataan listan alkioita alusta lähtien etsittävään.

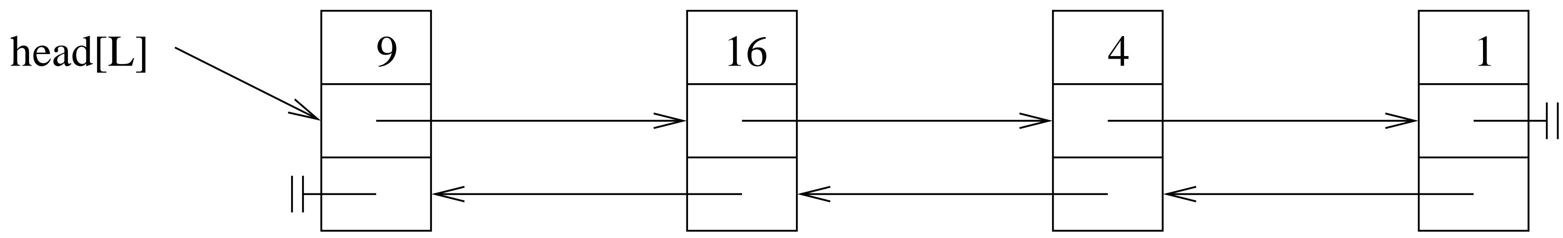
- Jos etsittävää ei löydy, niin käydään kaikki listan alkiot läpi.

- Pahimmassa tapauksessa operaation aikavaativuus on siis $\mathcal{O}(n)$, missä $n =$ listalla olevien alkioiden määrä.

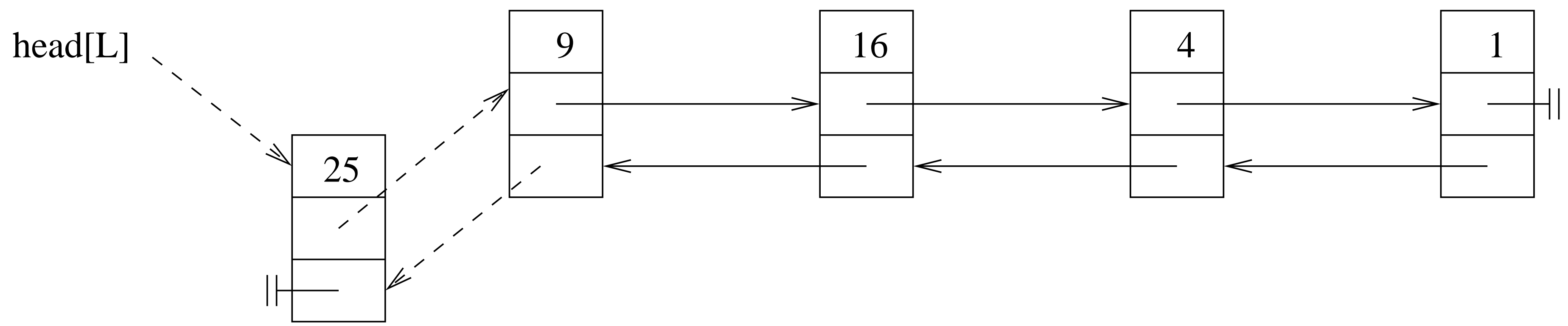
- Mutta uuden alkion k *lisääminen* listan L alkuun on nopeaa:

```
insert(L,k)
    x ← new listasolmu
    key[x] ← k
    next[x] ← head[L]
    prev[x] ← NIL
    if head[L] ≠ NIL then
        prev[head[L]] ← x
    head[L] ← x
```

- Käsitellään vain muutamaa viitettä
  - joiden lukumäärä ei riipu listassa olevien alkioiden lukumäärästä n
  - joten operaation vaativuus on $\mathcal{O}(1)$.



insert(L,25)



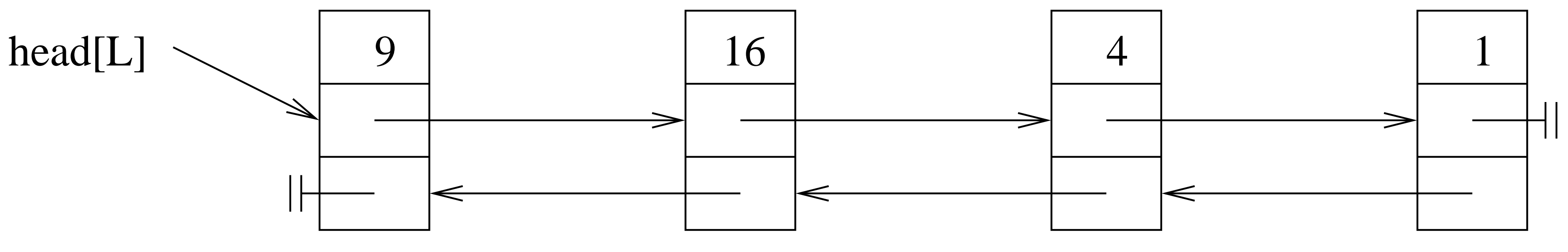
- Tietueen x *poistaminen* listastaan L on yhtä nopeaa:

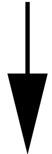
```
delete(L,x)
    y ← prev[x]
    z ← next[x]
    if y = NIL then
        head[L] ← z
    else
        next[y] ← z
    if z ≠ NIL then
        prev[z] ← y
```

- Operaatio delete saa parametrinaan x viitteen poistettavaan tietueeseen.

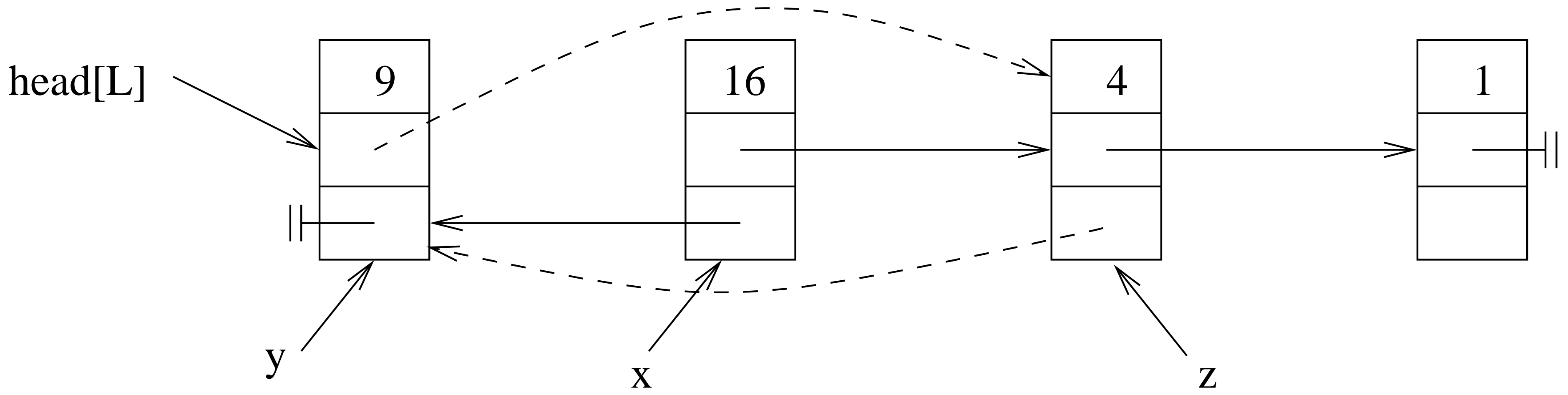
  Jos halutaankin poistaa avaimen k sisältävä tietue, niin se on ensin haettava operaatiolla search:

```
x ← search(L, k)
if x ≠ NIL then
    delete(L, x)
```





delete(L,search(16))



- Entä loput kalvojen 1.5 operaatioista?

- Operaatio min voidaan toteuttaa seuraavasti:

  min(L)
      $x \leftarrow$ head[L]
      $p \leftarrow$ head[L]
      while $p \neq$ NIL **do**
          if key[p] $<$ key[x] **then**
              $x \leftarrow p$
          $p \leftarrow$ next[p]
      return x

- Operaatio käy läpi listan kaikki alkiot, eli vaatii ajan $\mathcal{O}(n)$.

- Operaatio succ voidaan toteuttaa seuraavasti:

```
succ(L, x)
    y ← NIL
    p ← head[L]
    while p ≠ NIL do
        if  key[p] > key[x] and
            (y = NIL or key[p] < key[y])
            then y ← p
        p ← next[p]
    return y
```

- Operaatio käy koko listan läpi, eli aikavaativuus $\mathcal{O}(n)$.

- Loput operaatiot max ja pred toteutetaan samaan tyyliin kuin min ja succ.

- Toteutuksessamme operaatioiden vaativuudet ovat itse asiassa samat kuin kalvojen 1.5 taulukon toteutuksessa nimeltä **lista1**.

  – Lista ei ole kovin tehokas tietorakenne joukon toteutukseen silloin, jos tietoa haetaan avaimen arvolla k.

  + Myöhemmin kurssilla esitellään avainkyselyissä parempia tietorakenteita.

- Hyvä ohjelmoija valitseekin eri joukkojen toteutukset niiden eri käyttöyhteyksien mukaan.

- Tarkkaan ottaen tämä **lista1** *ei* toteutakaan matemaattista käsitettä ”joukko”.

  Miksei?

2.4.1 Järjestetty rengaslista

- Jos talletamme avaimet listalle *järjestyksessä*, niin

  – operaation insert vaativuus huononee luokkaan $\mathcal{O}(n)$

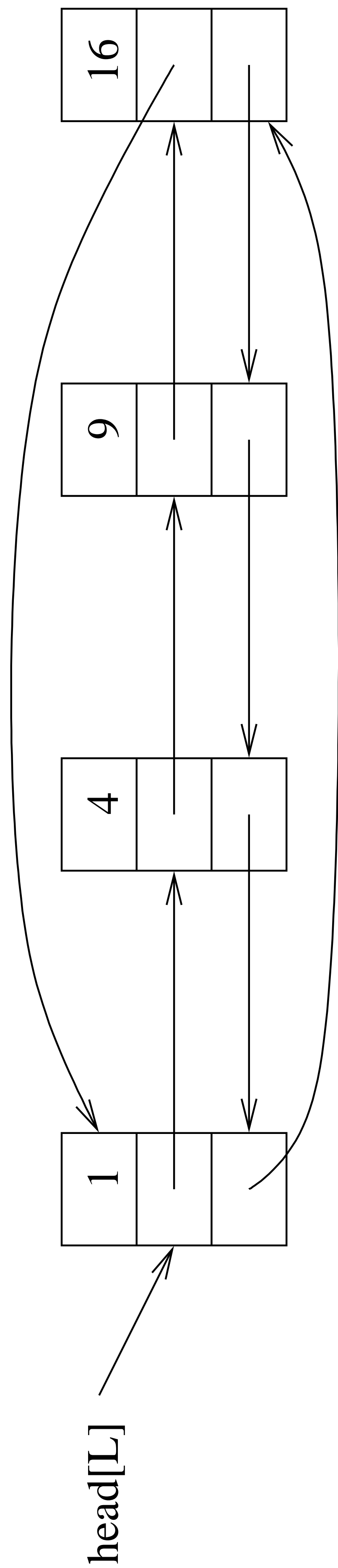
    koska lisäyksessä etsitään uudelle avaimelle järjestyksen mukainen oikea paikka

  + operaatioiden max ja pred vaativuus paranee luokkaan $\mathcal{O}(1)$

    – esimerkiksi käyttäen järjestettyä *rengas*listaa

    – mutta muitakin yhtä tehokkaita järjestettyjä listarakenteita on olemassa.

- Näin saadaan kalvojen 1.5 toteutus nimeltä **lista2**.


head[L]
1
4
9
16

- Lisätään viitteet
  - prev[jonon ensimmäinen alkio] = jonon viimeinen alkio
  - next[jonon viimeinen alkio] = jonon ensimmäinen alkio

  joista johtuu nimitys *rengas*lista.

- Vertaa jakojäännösoperaattorin mod käyttö jonojen taulukkototeutuksessa kalvoilla 2.2.

- Avainten järjestykseen liittyvät operaatiot yksinkertaistuvat ja tehostuvat:

```
min(L)
    return head[L]

max(L)
    if head[L] = NIL
        then return NIL
        else return prev[head[L]]

succ(L, x)
    if next[x] = head[L]
        then return NIL
        else return next[x]

pred(L, x)
    if x = head[L]
        then return NIL
        else return prev[x]
```

- Listan tulostus avainjärjestyksessä tulee elegantiksi ja tehokkaaksi:

```
x ← min[L]
while x ≠ NIL do
    print (key[x])
    x ← succ(L, x)
```

- Tietueen poistaminen järjestetyltä rengaslistalta:

```
delete(L, x)
    y ← prev[x]
    z ← next[x]
    if x = z then
        ▷poistettava on listan ainoa alkio
        head[L] ← NIL
    else
        next[y] ← z
        prev[z] ← y
        if x = head[L] then
            head[L] ← z
```

- Tietueen etsintä järjestetyltä rengaslistalta:

```
search(L, k)
    if head[L] = NIL then
        return NIL
    x ← head[L]
    while next[x] ≠ head[L] and
            k > key[x] do
        x ← next[x]
    if key[x] = k then
        return x
    else
        return NIL
```

- Voimme lopettaa etsinnän heti kun listalla tulee vastaan tietue x jonka avain on vähintään etsittävä avain k.

- Etsinnän pahimman tapauksen aikavaativuus on kuitenkin yhä $O(n)$.

- Järjestetyn rengaslistan operaatioista hankalin on tietueen lisääminen.

- Koska lisättävä tietue on vietävä oikealle paikalleen, kuluu aikaa pahimmassa tapauksessa $O(n)$.

- Operaatio on hiukan hankala myös monien erikseen huomioitavien tapaustensa takia.

- Huomaa tapausten käsittelyjärjestys:

  1. Onko lista L on *tyhjä?*

  2. Muuten listalla L on *ensimmäinen* tietue.

     Vasta nyt tapauksessa voimme käyttää sen kenttiä.

  3. Etsitään oikea lisäyskohta siirtämällä viitettä p listassa eteenpäin:

     (a) Tarvitseeko siirtää kertaakaan?

     (b) Menikö p listan L loppuun?

     (c) Vai lisätäänkö sen eteen?

```
insert(L, k)
    x ← new jonosolmu
    key[x] ← k
    p ← head[L]
    if p = NIL then                        ▷lisäys tyhjään
        next[x] ← x
        prev[x] ← x
        head[L] ← x
    else if k < key[p] then                ▷lisäys alkuun
        next[x] ← p
        prev[x] ← prev[p]
        next[prev[x]] ← x
        prev[p] ← x
        head[L] ← x
    else
        while  next[p] ≠ head[L] and key[p] < k
            do p ← next[p]                 ▷vrt. search
        if k > key[p] then                 ▷lisäys loppuun
            next[x] ← head[L]
            prev[x] ← p
            next[p] ← x
            prev[head[L]] ← x
        else                               ▷lisäys väliin
            next[x] ← p
            prev[x] ← prev[p]
            next[prev[p]] ← x
            prev[p] ← x
```

2.4.2 Tunnussolmullinen lista

- Joitakin operaatioita

  (kuten kalvojen 2.4.1 rengaslistaan lisäystä)

  mutkistaa hieman erikoistapausten (lista tyhjä? käsitelläänkö ensimmäistä/viimeistä tietuetta?) huomioiminen.

- Yksi variaatio linkitetyistä listoista on *tunnussolmullinen rengas*lista L:

  - nyt listan alussa on *aina* tunnussolmu

    eli erikoissolmu jossa ei säilytetä tietoa

  - attribuutti nil[L] osoittaa listan tunnussolmuun

  - listan ensimmäinen alkio löytyy viitteen next[nil[L]] päästä

  - listan viimeinen alkio löytyy viitteen prev[nil[L]] päästä

  - tyhjällä listalla next[nil[L]] = prev[nil[L]] = nil[L].

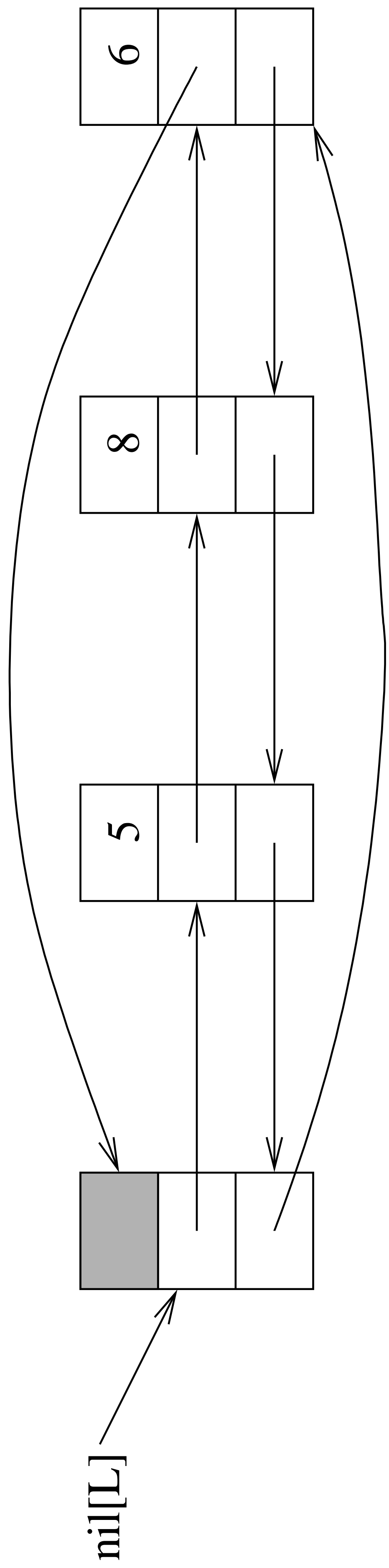
- Käsitellään seuraavassa tunnussolmullisen listan variaatiota joka ei edellytä alkioille suuruusjärjestystä.

- Tietueen poisto tunnussolmullisesta rengaslistalta hoituu erittäin helposti:

  delete(L, x)
  next[prev[x]] $\leftarrow$ next[x]
  prev[next[x]] $\leftarrow$ prev[x]

- Tietueen etsintä tunnussolmullisesta rengaslistasta:

  search(L, k)
  $x \leftarrow$ next[nil[L]]
  while $x \neq$ nil[L] **and** $k \neq$ key[x] **do**
  $x \leftarrow$ next[x]
  if $x =$ nil[L] **then**
  return NIL
  else
  return x


nil[L]
5
8
6

- Tietueen lisääminen:

  insert(L, k)
  $x \leftarrow$ **new** jonosolmu
  key[x] $\leftarrow k$
  next[x] $\leftarrow$ next[nil[L]]
  prev[x] $\leftarrow$ nil[L]
  prev[next[x]] $\leftarrow x$
  next[nil[L]] $\leftarrow x$

- Listoista voi kehittää erilaisia variaatioita:

  - molempiin suuntiin *tai* vain yhteen suuntaan (eli eteenpäin) linkitetty
  - päättyvä *tai* rengaslista
  - tunnussolmulla *tai* ilman
  - ...

2.5 Pinon toteutus Javalla

- Vaikka kurssi on kieliriippumaton, toteutamme esimerkkinä Javalla kalvojen 2.1 abstraktin tietotyypin pino.

- Luokan `Pino` metodit:

  `public void push(Object k)` laittaa pinon päällimmäiseksi tietueeksi olio(viittee)n k

  `public Object pop()` poistaa ja palauttaa pinosta sen päällimmäisen tietueen

  `public boolean empty()` tutkii, onko pino tyhjä vai ei.

- Pinon tietueet toteutetaan luokan `Pino` yksityisen sisäluokan `PinoSolmu` ilmentyminä.

- pinon päällimmäiseen alkioon osoittaa yksityinen `PinoSolmu`-tyyppinen kenttä `top`.

```
public class Pino {
    private class PinoSolmu {
        Object key;
        PinoSolmu next;
        private PinoSolmu(Object k,
                              PinoSolmu seur) {
            key = k;
            next = seur;
        }
    }
    private PinoSolmu top;
    Pino() {
        top = null;
    }
```

- Yksityisen sisäluokan `PinoSolmu` ilmentymät ovat näkyvissä vain luokan `Pino` sisällä.

- Saamme abstraktin tietotyypin:
  - Pinoa koossa pitäviä viitteitä `top`, `next` ja `key` käsittelevät vain `push`, `pop` ja `empty`.
  - Ne ovat siis suojassa ulkomaailmalta.
  - Siis pinoa ei voi rikkoa.

```
    public void push(Object k) {
        PinoSolmu uusi = new PinoSolmu(k,top);
        top = uusi;
    }

    public Object pop() {
        PinoSolmu pois = top;
        top = pois.next;
        return pois.key;
    }

    public boolean empty() {
        return (top == null);
    }
}
```

- Operaation `pop` jälkeen vanha pinon päällimmäinen tietue `pois` jää ilman viitettä

  ja Javan roskankerääjä tulee aikanaan kierrättämään sille varatun muistialueen uusiokäyttöön.

2.6 Esimerkki pinoa käyttävästä algoritmista

- Ohjelmoinnissa tulee helposti virheitä sulkumerkkien kirjoittamisessa:
  - sulkuja on joko liikaa tai liian vähän
  - jollekin alkusululle ei löydy loppusulkua tai päinvastoin
  - erilaiset sulkeet voivat "mennä ristiin": `[(...])`.

- Sulkujen tasapaino syötteenä olevasta merkkijonosta voidaan testata esimerkiksi seuraavasti:

  - luetaan syötemerkkijonon jokainen merkki, yksi kerrallaan, alusta alkaen

  - viedään jokainen vasen sulkumerkki pinoon

  - kun luetaan jokin oikea sulkumerkki, sitä vastaavan vasemman sulkumerkin pitää löytyä pinon päältä, se poistetaan

  - kun tiedosto on luettu, pinon pitää olla tyhjä.

- Oletetaan apufunktio `readnext`(t) joka lukee ja palauttaa seuraavan merkin syötevirrasta t

  esimerkiksi tekstitiedostosta.

- Oletetaan yksinkertaisuuden vuoksi että t saa sisältää *ainoastaan* sulkumerkkejä.

- Pinoa S käyttävä algoritmi sulkujen tasapainon tarkastamiseksi:

$S \leftarrow$ tyhjä merkkipino
while (syötevirta t ei ole loppu) **do**
    $c_2 \leftarrow$ `readnext`(t)
    if c_2 on '(', '[' tai '{' **then**
        push(S, c_2)
    else if empty(S) **then**
        ERROR "Liian vähän alkusulkuja!"
    else
        $c_1 \leftarrow$ pop(S)
        if (c_1 **= '(' and not** c_2 **= ')') or**
            (c_1 **= '[' and not** c_2 **= ']') or**
            (c_1 **= '{' and not** c_2 **= '}')**
        then ERROR "Sulut ristissä!"
if not empty(S) **then**
    ERROR "Liian vähän loppusulkuja!"

- ERROR tulostaa vastaavan virheilmoituksen ja keskeyttää algoritmin suorituksen.

3 Hakupuut

- Hakupuu on listaa huomattavasti edistyneempi tapa toteuttaa kalvojen 1.5 abstrakti tietotyyppi ”joukko”.

- Puurakenteelle on tietojenkäsittelyssä myös muutakin käyttöä:

  - algoritmin suoritusajan analysoinnissa

  - ohjelman laskennan etenemisen kuvailussa

  - ongelmanratkaisun apuvälineenä, esimerkiksi strategiapelien ohjelmoinnissa

  - ...

- Puilla on siis 2 roolia:

  konkreettisina tietorakenteina tietokoneen muistissa

  abstrakteina ongelman hahmottamisen ja analysoinnin apuvälineinä.

- Ennen kuin menemme konkreettisiin hakupuihin, tutustumme yleiseen puihin liittyvään käsitteistöön.

  Tosin eri käsitteiden

  - nimet

  - määritelmien yksityiskohdat

  valitettavasti eroavat eri lähteissä...

- *Puu* on kokoelma *solmuja* ja niitä yhdistäviä *kaaria* siten että:

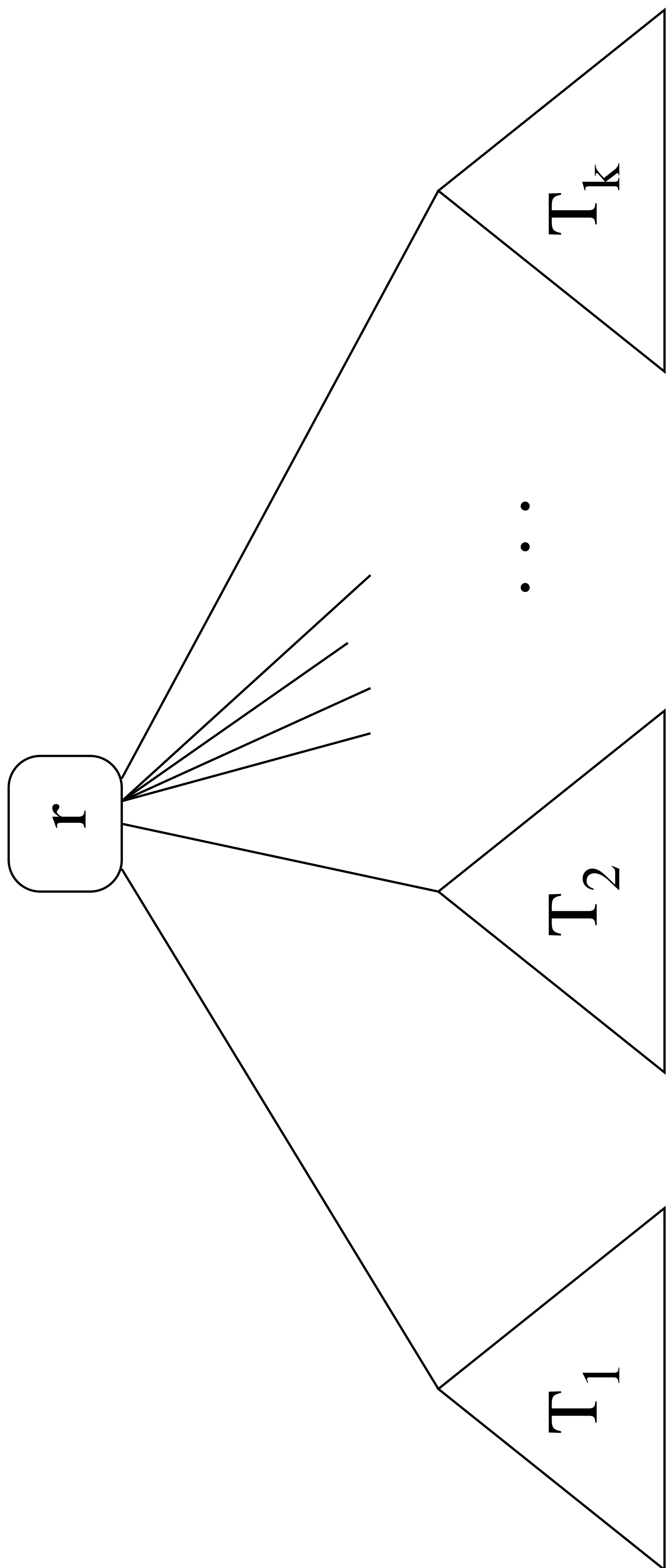
  - kokoelma on tyhjä, tai

  - yksi solmuista r on *juuri*

    * johon kaaret liittävät nolla tai useampia *alipuita* $T_1, \ldots, T_k$

    * jotka itsekin ovat puita.

    Tämä on *itseensä viittaava* (induktiivinen, rekursiivinen) määritelmä.


r
T1
T2
Tk

- Puiden terminologia on valittu mukailemaan *sukupuuta* (miehen mukaan).

- Puiden $T_1, \ldots, T_k$ omat juuret $r_1, \ldots, r_k$ ovat juuren r *lapsia*

  ja r on lastensa $r_1, \ldots, r_k$ *vanhempi* (tai *isä*)

- Jokaisella muulla solmulla kuin juurella on tasan yksi vanhempi:

  tästä seuraa että n-solmuisessa puussa on täsmälleen $n-1$ kaarta.

- Solmu jolla ei ole lapsia on *lehti*.

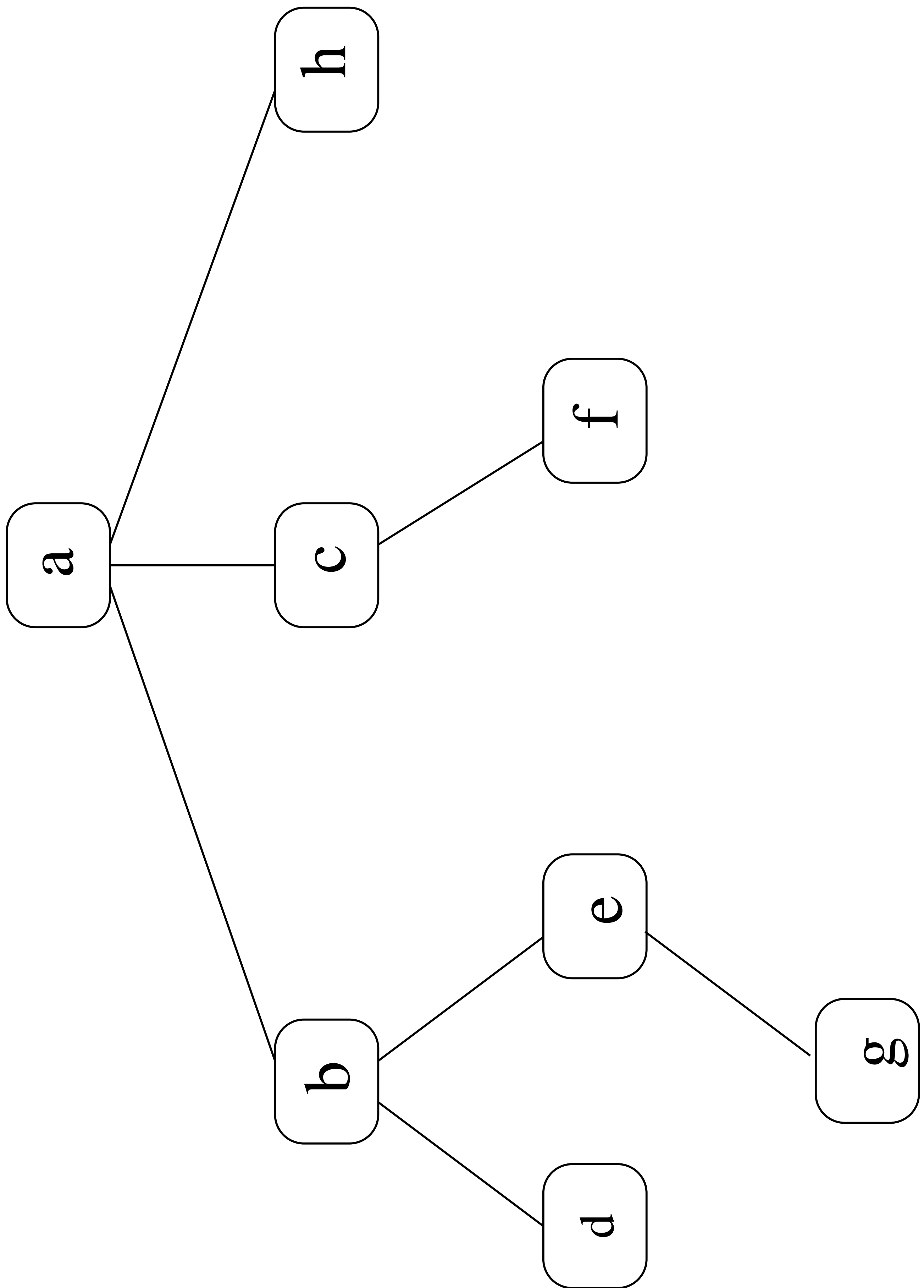
- Solmut joilla on yhteinen vanhempi, ovat *sisaruksia*.

- Sellaiset käsitteet kuin *isovanhempi*, *lapsenlapsi*, *setä* ja *serkku* määräytyvät luonnollisella tavalla.

- Seuraavassa kuvassa
  - koko puun juuri on a, ja sillä on lapset b, c ja h
  - puun lehtiä ovat solmut d, g, f ja h
  - solmut d, e ja f ovat solmun a lapsenlapsia

    ja solmu a on solmujen d, e ja f isovanhempi (tai isoisä)
  - solmut d ja e ovat sisaruksia

    ja niiden serkku on f
  - solmu b on solmun f setä.


a
b
c
d
e
f
g
h

Polku solmusta x_1 solmuun x_k on jono solmuja

$$x_1, x_2, x_3, \ldots, x_k$$

siten että aina seuraava solmu x_{i+1} on edellisen solmun x_i lapsi.

Polkuja voidaan ajatella myös ”takaperin” lapsesta isään.

Polun pituus on sen kaarien lukumäärä $k - 1$.

- Edellisessä kuvassa a, b, e, g on polku solmusta a solmuun g.
- Tämän polun pituus on 3.
- Erikoistapauksena jokaisesta solmusta on polku itseensä, ja sen pituus on 0.

Edeltäjä: Jos on olemassa polku solmusta x_i solmuun x_j, niin solmu x_i on solmun x_j *edeltäjä*.

- Jos $x_i \neq x_j$, niin *aito* edeltäjä.
- Edellisessä kuvassa solmun e edeltäjät ovat e itse, b ja a.
- Niistä vain b ja a ovat aitoja.

Jälkeläinen: Kääntäen, solmu x_j on solmun x_i (aito) *jälkeläinen* eli **seuraaja**.

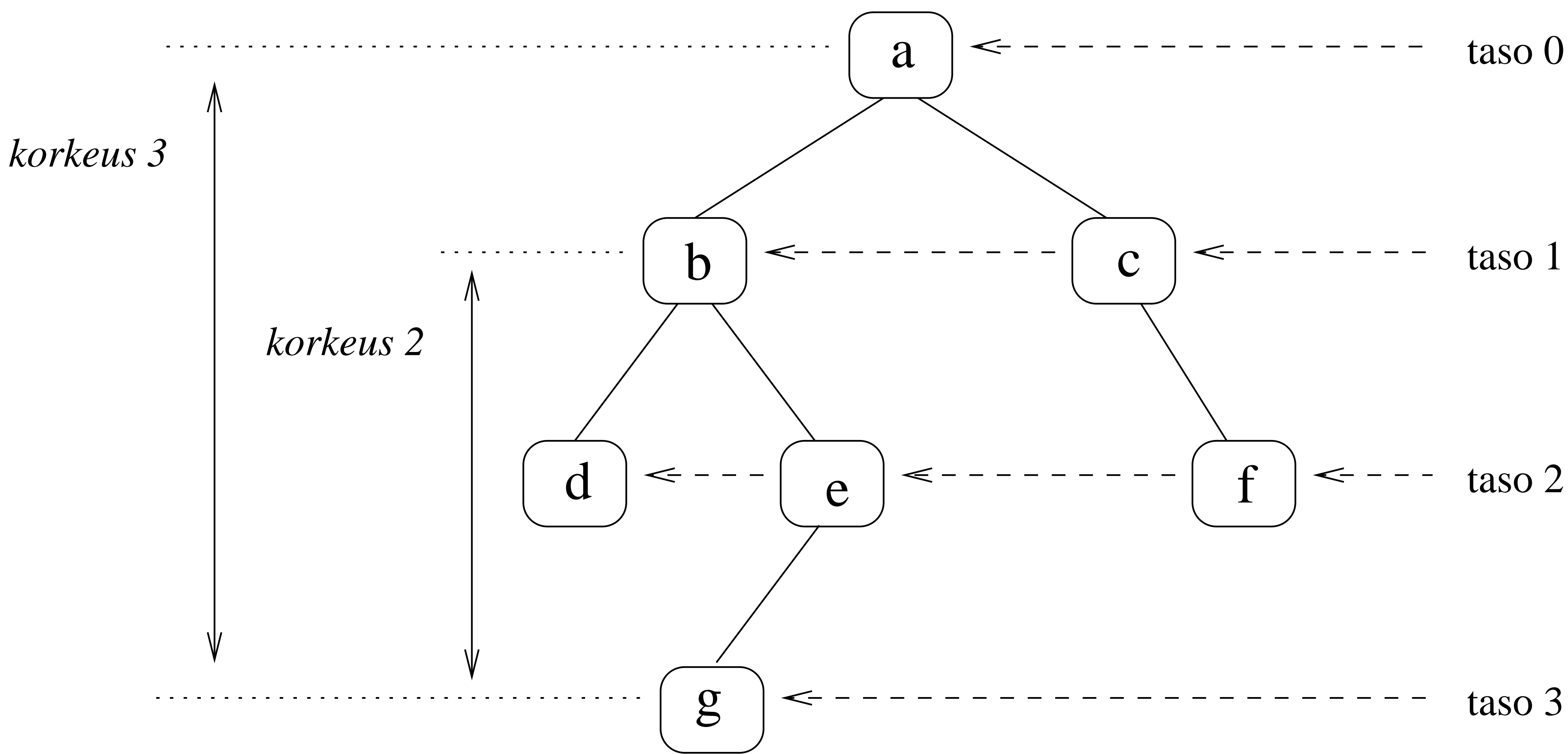
- Edellisessä kuvassa solmun e jälkeläiset ovat e itse ja g.
- Niistä vain g on aito.

Solmun x **taso** eli **syvyys** on polun pituus koko puun juuresta solmuun x.

- Juuren itsensä taso on 0.
- Seuraavassa kuvassa on merkitty edellisen kuvan solmujen tasot.

Solmun x **korkeus** on pisimmän polun pituus solmusta x lehteen.

- Koko *puun korkeus* on sen juuren korkeus.
- Seuraavassa kuvassa on solmujen a ja b korkeudet.
- Solmun c korkeus onkin 1.


a
b
c
d
e
f
g
taso 0
taso 1
taso 2
taso 3
korkeus 3
korkeus 2

3.1 Binääripuu

- Jos puun jokaisella solmuilla on korkeintaan kaksi lasta, niin kyseessä on *binääripuu*.

- Binääripuun alipuiden juuria kutsutaan *vasemmaksi* ja *oikeaksi* lapseksi.

- Solmun x

  vasempaan lapseen viitataan left[x]

  oikeaan right[x].

- Solmusta left[x] alkavaa puuta kutsutaan solmun x *vasemmaksi alipuuksi*

  ja solmusta right[x] alkavaa *oikeaksi* alipuuksi.

- Edellinen kuva on binääripuu. Siinä

$$\text{left}[b] = d$$
$$\text{right}[b] = e.$$

Lause 3.1. *Binääripuun tasolla i on korkeintaan*

$$2^i$$

solmua.

Todistus: Induktiolla tason i suhteen.

- Tasolla 0 on vain juurisolmu, ja

$$2^0 = 1.$$

- Oletetaan sitten, että väite pätee tasolla $i-1$. Osoitetaan sen nojalla, että väite pätee myös tasolla i.

  Induktio-oletuksen mukaan tasolla $i-1$ on korkeintaan

$$2^{i-1}$$

  solmua. Jokaisella niistä on korkeintaan 2 lasta. Siis tasolla i on korkeintaan

$$2 \cdot 2^{i-1} = 2^i$$

  solmua. □

Lause 3.2. *Olkoon T binääripuu jonka korkeus on k. Siinä on yhteensä korkeintaan*

$$2^{k+1} - 1$$

solmua.

Todistus: Binääripuun solmujen kokonaislukumäärä on summa sen eri tasojen solmujen kokonaislukumäärästä

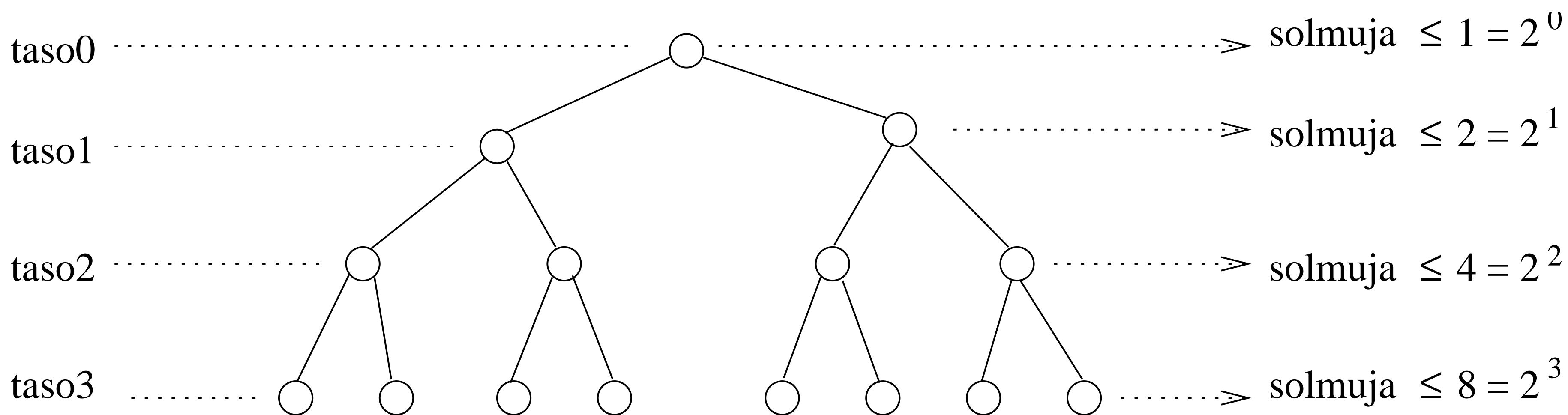
Puussa T on tasot $i = 0, 1, 2, \ldots, k$.

Edellinen lause 3.1 antaa ylärajan kunkin tason i solmujen lukumäärälle.

Siis koko puun solmujen lukumäärällä on yläraja

$$\sum_{i=0}^{k} 2^i = \frac{1 - 2^{k+1}}{1 - 2} = 2^{k+1} - 1$$

missä ensimmäinen askel on *geometrisen sarjan summa*kaava. □



$$\text{yht} \leq \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

- Binääripuu on

aito jos jokaisella solmulla on joko 0 tai 2 lasta

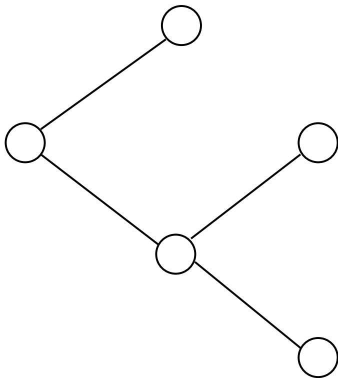
- eli siinä ei ole 1-lapsisia solmuja
- Seuraavan kuvan puista T1 ja T3 ovat aitoja, mutta T2 ei.

täysi jos sen jokainen lehti on samalla tasolla

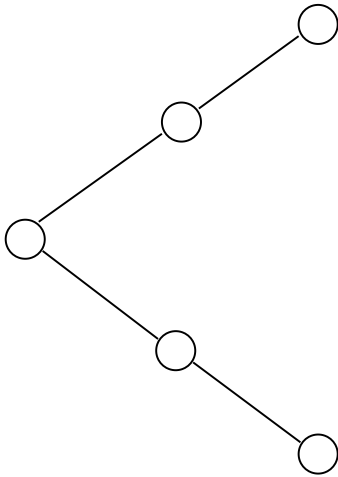
- Seuraavan kuvan puista T2 ja T3 ovat täysiä, mutta T1 ei.

täydellinen jos se on aito ja täysi

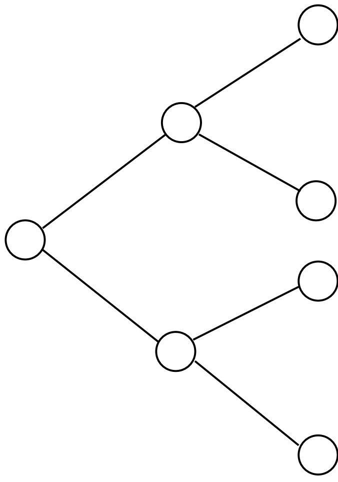
- eli siinä on kaikki ne solmut, jotka tämän korkuisessa binääripuussa voi enimmillään olla.
- Seuraavan kuvan puista vain T3 on täydellinen, kun taas T1 ja T2 eivät ole.



T1



T2



T3

- Siten korkeutta k olevassa täydellisessä binääripuussa

  solmuja on lauseen 3.2 nojalla tasan

  $$2^{k+1} - 1$$

  kappaletta

  ja niistä

  lehtiä on lauseen 3.1 nojalla tasan

  $$2^k$$

  kappaletta.

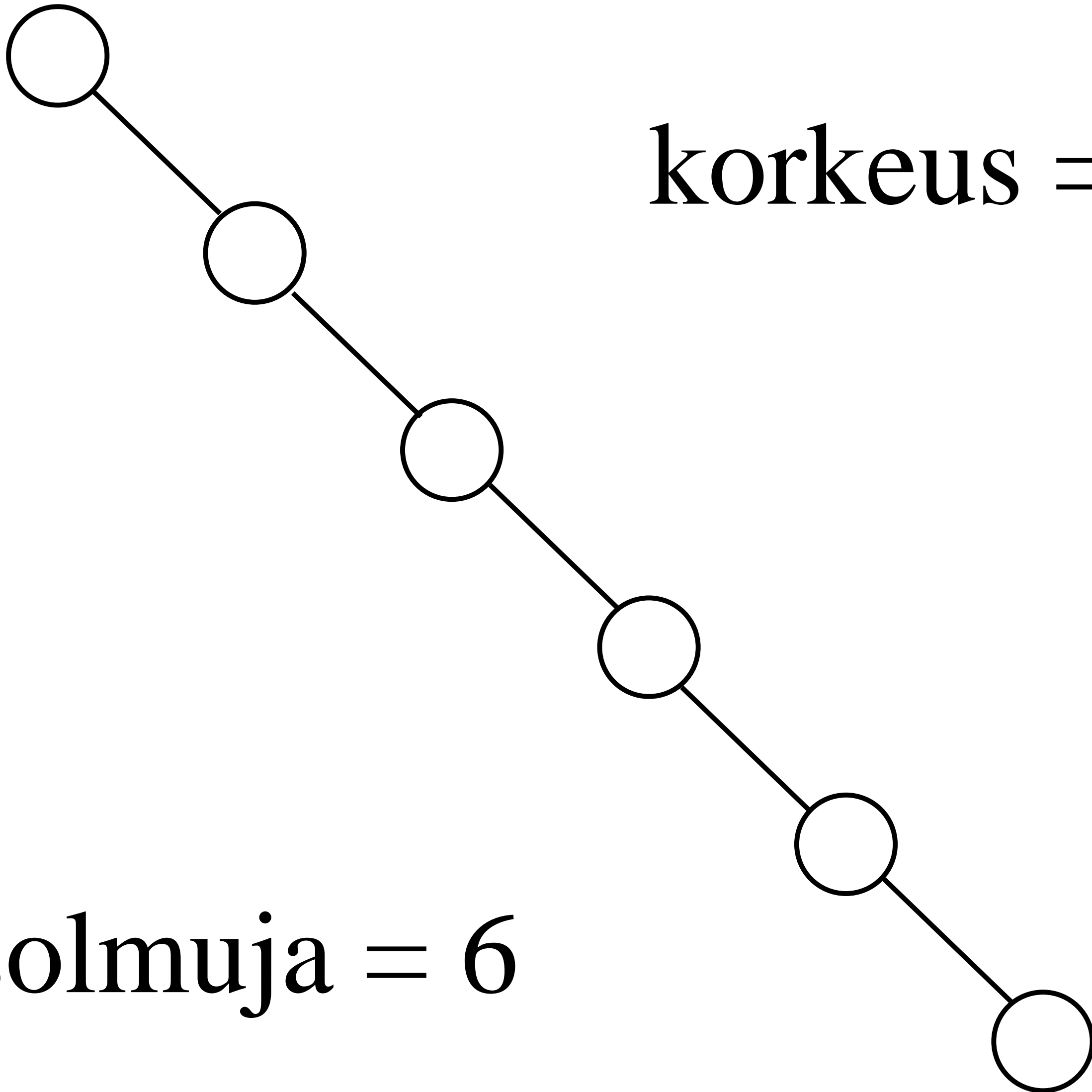
- Toisaalta binääripuussa jonka korkeus on k voi olla vähimmillään

  solmuja vain $k + 1$ kappaletta

  ja niistä

  lehtiä vain 1

  seuraavan kuvan perusteella.


korkeus = 5
solmuja = 6

Lause 3.3. *Jos binääripuussa T on $n > 0$ solmua, niin sen korkeus on*

vähintään $\log_2(n+1) - 1$

(tarvittaessa pyöristettynä ylöspäin)

enintään $n - 1$.

Todistus: Korkeus k on pienimmillään solmujen lukumäärään nähden silloin kun T on täydellinen. Saadaan ehto

$$\begin{aligned} 2^{k+1} - 1 &= n \\ 2^{k+1} &= n + 1 \\ \log_2\left(2^{k+1}\right) &= \log_2(n+1) \\ k + 1 &= \log_2(n+1) \\ k &= \log_2(n+1) - 1. \end{aligned}$$

Korkeus k on suurimmillaan edellisessä kuvassa, ja siellä $k = n - 1$. □

3.1.1 Binääripuualgoritmien johtamisesta

- Binääripuu määriteltiin kalvoja 3 seuraten *induktiivisesti:*

  Perustapaus: NIL on tyhjä eli *pienin* mahdollinen binääripuu.

  Induktio: Jos T_{vasen} ja T_{oikea} ovat *pienempiä* binääripuita, niin myös sellainen otus on binääripuu, jossa on

  – uusi solmu x jonka

  – vasempana alipuuna left[x] on T_{vasen}

  – oikeana alipuuna right[x] on T_{oikea}.

- Näin induktiivisesti määriteltyä rakennetta on luontevaa käsitellä *rekursiolla* määritelmänsä suhteen:

```
PuuRek(x)
1  if olemme perustapauksessa eli x = NIL
2      then return sopiva vakiotulos c
3      else  v_vasen ← PuuRek(left[x])
4            v_oikea ← PuuRek(right[x])
5            u ← itse solmun x tieto
6            return yhdistä välitulokset
                   v_vasen, v_oikea ja u
                   lopputulokseksi
```

- Tätä perusrekursioskeemaa voi muunnella sen mukaan, millainen tulos halutaan laskea:
  - Osat c, u ja yhdistämistapa saadaan halutun tuloksen määritelmästä.
  - Jos välitulosta v_{vasen} ei tarvita rivillä 6, niin rekursiokutsurivi 3 jää pois. (Samoin v_{oikea} ja rivi 4.)
  - Rivit 3–5 voidaan tehdä halutussa järjestyksessä.

- Esimerkki: tarvitaan sellainen algoritmi, jonka

  syötteenä annetaan sellainen binääripuu, jonka jokaisessa solmussa x on kokonaisuluku key[x]

  tuloksena halutaan niiden solmujen y lukumäärä, jossa key[y] on parillinen.

- Edellisellä reseptillä:

```
ParillistenLkm(x)
1  if x = NIL
2      then return 0
3      else  v_vasen ← ParillistenLkm(left[x])
4            v_oikea ← ParillistenLkm(right[x])
5            if key[x] on parillinen
                then u ← 1
                else u ← 0
6            return v_vasen + v_oikea + u
```

3.1.2 Binäärihakupuut

- Toteutetaan kalvojen 1.5 abstrakti tietotyyppi **joukko** siten että joukossa olevat alkiot talletetaan binääripuun solmuihin.

- Rajoitumme jälleen yksinkertaistettuun tapaukseen jossa talletettavat tietoalkiot sisältävät ainoastaan avaimen.

- Puu rakentuu tyyppiä *puusolmu* olevista tietueista.
  - Kukin tietue edustaa yhtä solmua.
  - Tietueen x kentät ovat

$$\begin{aligned} \text{key}[x] &= \text{talletettu avain} \\ \text{left}[x] &= \text{viite vasempaan lapseen} \\ \text{right}[x] &= \text{viite oikeaan lapseen} \\ p[x] &= \text{viite vanhempaan.} \end{aligned}$$

  - Jos viitattavaa solmua (eli tietuetta) ei ole, niin viite on NIL.

- Viite $p[x]$ ei ole välttämätön
  - mutta se tehostaa joitakin operaatioita
  - kuten kalvojen 2.4 2-suuntainen linkitys.

- Koko puulla T on attribuutti root[T] joka viittaa juurisolmuun.

- Puun jokaisessa solmussa y vallitsee *binäärihakupuuehto:*

$$\text{key}[x] < \text{key}[y] < \text{key}[z]$$

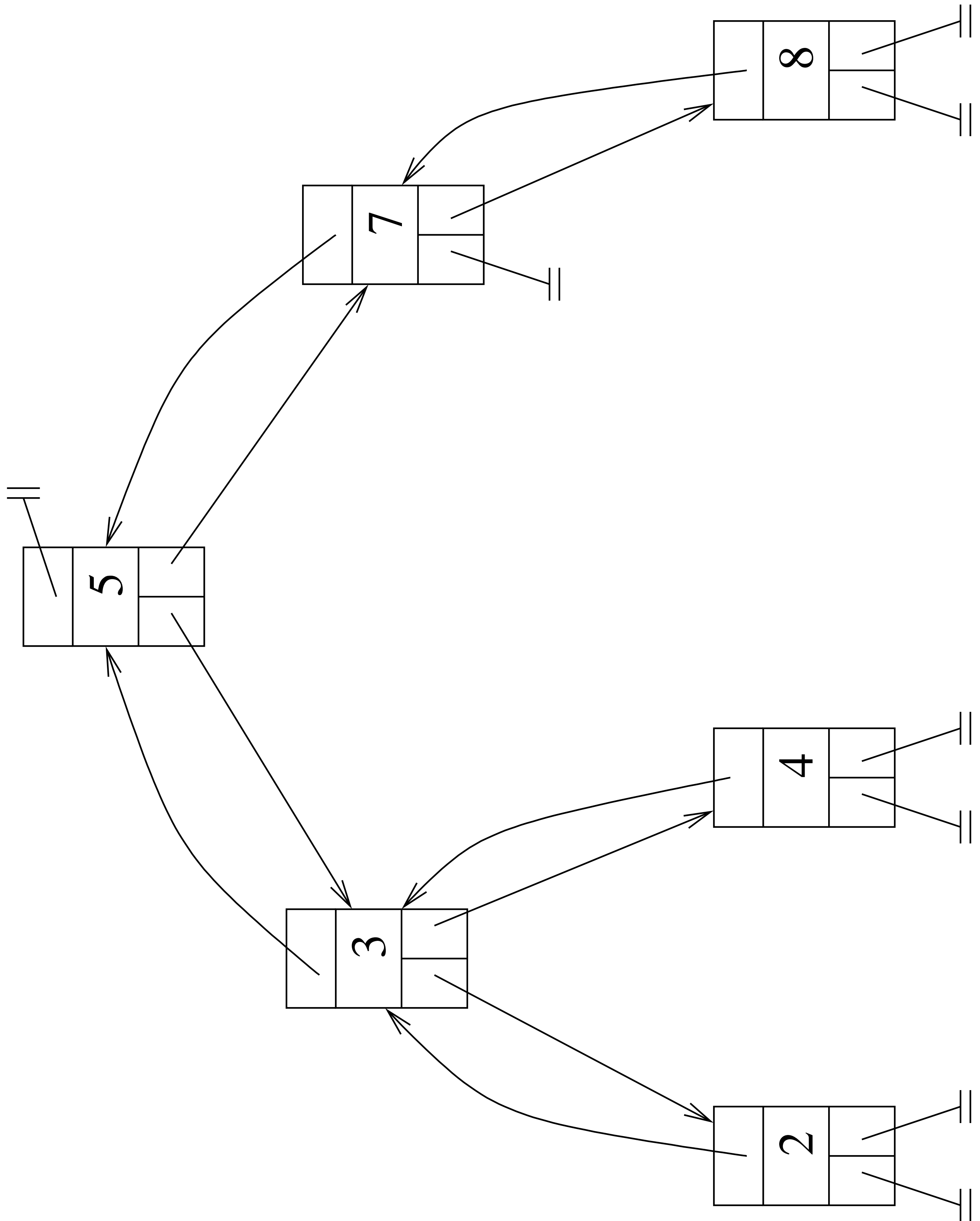
  missä
  - x on mikä tahansa alipuun left$[y]$
  - z on mikä tahansa alipuun right$[y]$

  solmu.

- Vertaa kalvojen 1.3.3 binäärihakuun: solmusta y

  vasemmalle ovat sitä **pienemmät** solmut x

  oikealle suuremmat solmut z.


8
7
5
4
3
2

- Esimerkkinä edellisessä kuvassa on hakupuu, jossa on avaimet

$$2, 3, 4, 5, 7 \text{ ja } 8.$$

- Yleensä jätämme piirroksista pois viitekenttien eksplisiittiset sisällöt.

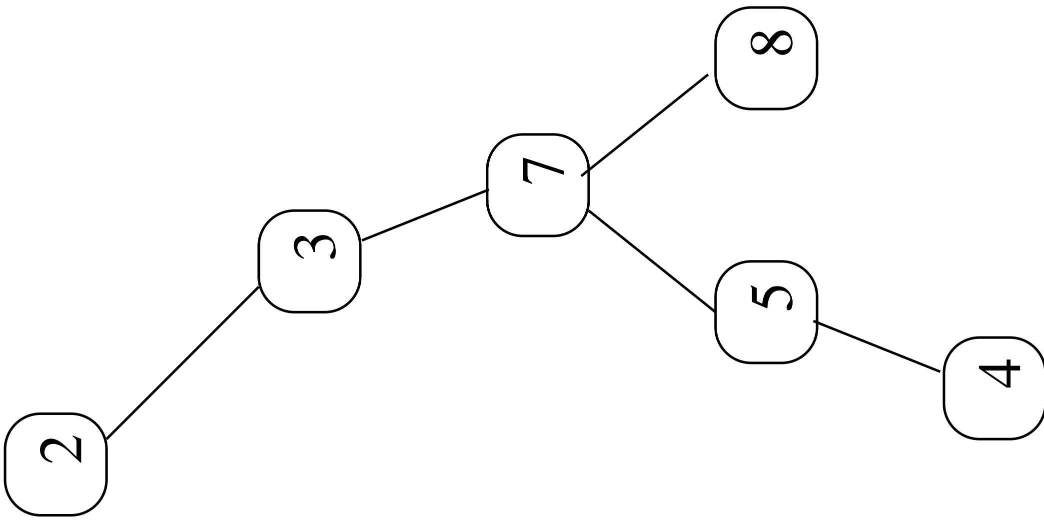
  Näin saadaan seuraavan kuvan vasen piirros.

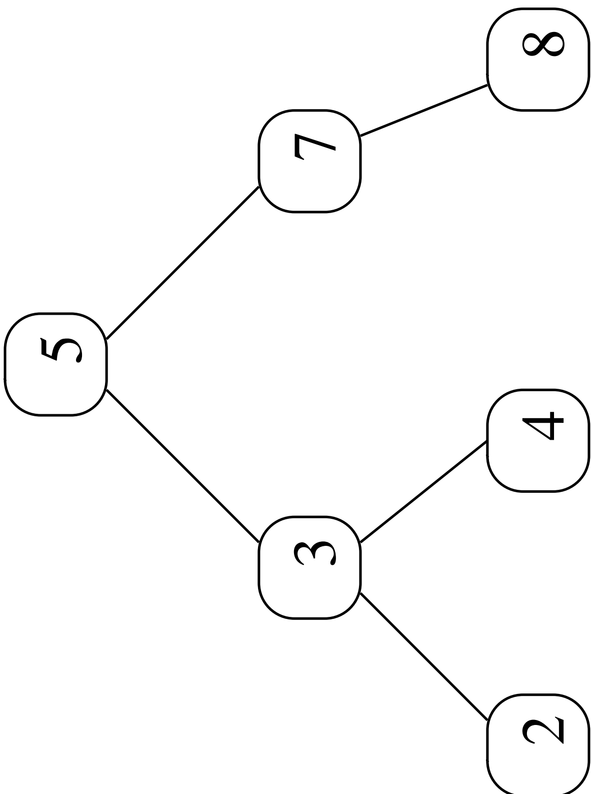
- Samoille avaimille on monta erilaista hakupuuta.

  Esimerkiksi seuraavan kuvan oikea puu sisältää samat avaimet ja toteuttaa binäärihakupuuehdon

  mutta puun muoto on erilainen.

- Myöhemmin käsitellään *tasapainottamista*, jolla pyritään valitsemaan ”tehokkaan muotoinen” puu.


2
3
7
8
5
4


5
7
8
3
4
2

- Puun T alkiot voidaan tulostaa suuruusjärjestyksessä kutsulla

  inorderTreeWalk(root[T])

  missä kutsutaan rekursiivista funktiota

```
inorderTreeWalk(x)
    if x ≠ NIL then
        inorderTreeWalk(left[x])
        print key[x]
        inorderTreeWalk(right[x])
```

- Algoritmi on rakennettu kalvojen 3.1.1 tapaan:

  Tyhjälle puulle ei tarvitse tehdä mitään.

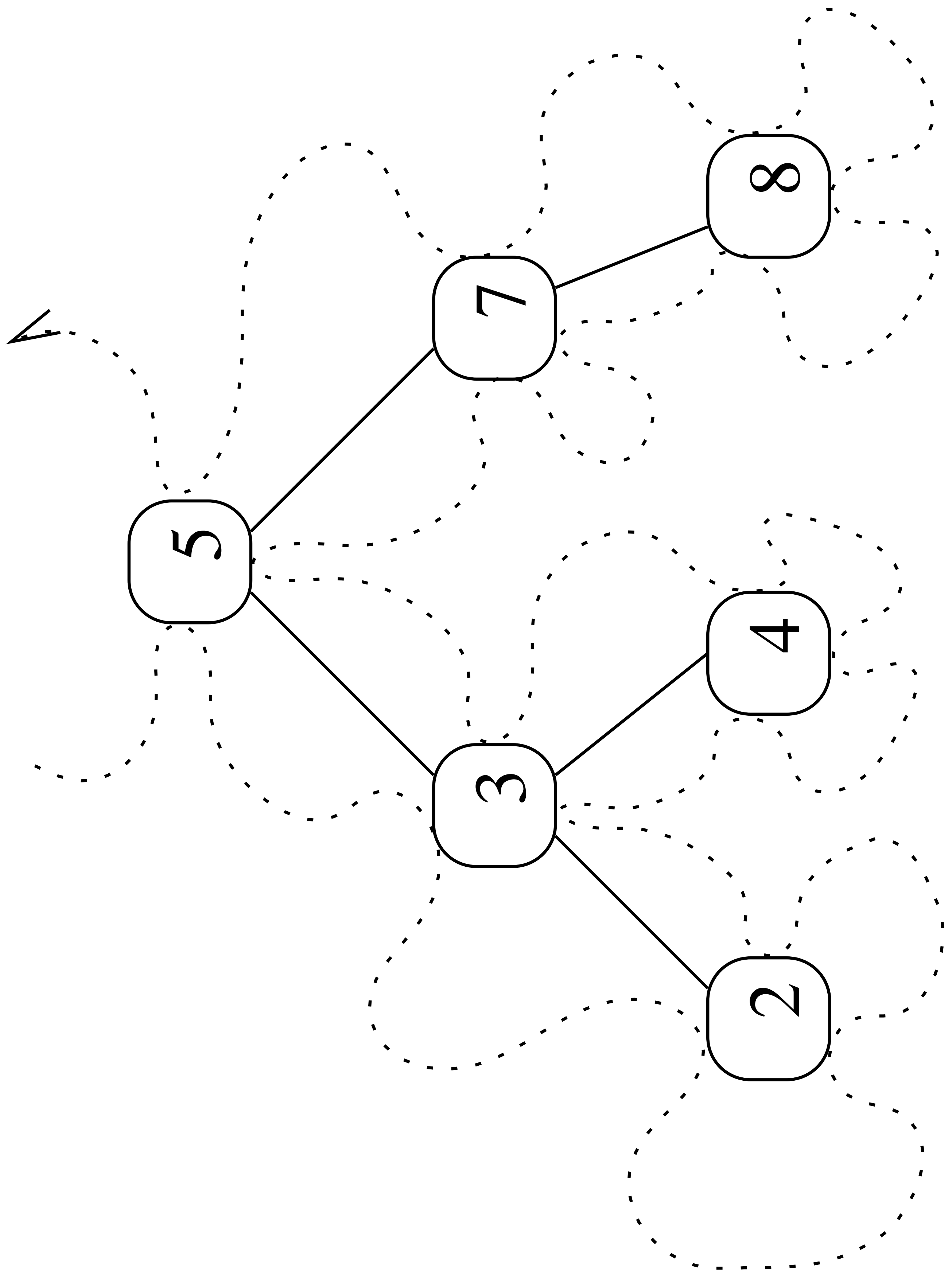
  Solmussa x toimitaan seuraavassa järjestyksessä:

  1. ensin tulostetaan vasen alipuu left[x] rekursiivisesti
  2. sitten tulostetaan juuren sisältö key[x]
  3. lopuksi tulostetaan oikea alipuu right[x] rekursiivisesti.

- Algoritmin eteneminen edellisen kuvan vasemmanpuoleisessa puussa:

```
inorderTreeWalk(5)
    inorderTreeWalk(3)
        inorderTreeWalk(2)
            inorderTreeWalk(NIL)
            print(2)                        2
            inorderTreeWalk(NIL)
        print(3)                            3
        inorderTreeWalk(4)
            inorderTreeWalk(NIL)
            print(4)                        4
            inorderTreeWalk(NIL)
    print(5)                                5
    inorderTreeWalk(7)
        inorderTreeWalk(NIL)
        print(7)                            7
        inorderTreeWalk(8)
            inorderTreeWalk(NIL)
            print(8)                        8
            inorderTreeWalk(NIL)
```

(Kutsu inorderTreeWalk(5) luetaan ”parametri x viittaa solmuun jonka avain on 5”.)


5
7
8
3
4
2

Lause 3.4. *Kutsu* inorderTreeWalk(x) *tulostaa kasvavassa järjestyksessä sen alipuun avaimet, jonka juuri on* x.

Todistus: Binääripuuta käsittelevän rekursiivisen funktion toimintaa on luonteva tarkastella *induktiolla yli puun rakenteen:*

- Jos puu x on tyhjä eli $x = \mathrm{NIL}$, niin tulostettavia avaimia ei ole, eikä funktio myöskään tulosta yhtään mitään.

- Muuten puu x on epätyhjä, eli tietue x on olemassa. Funktio tulostaa

  aluksi vasemman alipuun left[x] sisällön

  sitten itse solmun avaimen key[x]

  lopuksi oikean alipuun right[x] sisällön.

  Induktio-oletuksen nojalla alipuiden avaimet tulostetaan kasvavassa järjestyksessä.

- Binäärihakupuuehdon mukaan key[x] on

  suurempi kuin **vasemman** alipuun suurin

  eli **viimeiseksi** tulostettu

  mutta

  pienempi kuin **oikean** alipuun pienin

  eli **ensimmäiseksi** tulostettu

  avain.

  On siis oikein tulostaa se niiden väliin. $\square$

- Edelliseen kuvaan on piirretty katkoviivana algoritmin kulkema reitti esimerkkipuuta pitkin.

- Algoritmi käsittelee puun **sisäjärjestyksessä:**

  Tultaessa solmuun x

  aluksi käsitellään **vasen** lapsi left[x]

  sitten tulostetaan **solmu**n oma key[x]

  lopuksi oikea lapsi right[x].

- Binääripuun voi käsitellä myös

  esijärjestyksessä jolloin **solmu** käsitelläänkin heti **aluksi**

  jälkijärjestyksessä jolloin **solmu** käsitelläänkin vasta **lopuksi**.

- Olkoon $n =$ koko puun solmujen lukumäärä.

- Algoritmin *aika*vaativuus on $\mathcal{O}(n)$, koska

  jokaiseen solmuun x tullaan tasan kolme kertaa:

  1. aluksi sen vanhemmasta
  2. palattaessa siihen vasemmasta rekursiokutsusta
  3. palattaessa siihen oikeasta rekursiokutsusta.

- Algoritmin *tila*vaativuus:

  – Algoritmin edetessä rekursiopinoon on talletettuna reitti juurisolmusta tarkasteltavaan solmuun.

  – Algoritmin suoritusaikana rekursiopinossa on siis tietoa ”piilossa”.

  – Rekursiopinon koko on pahimmillaan sama kuin puun korkeus h, eli tilavaativuus on $\mathcal{O}(h)$.

- Lauseen 3.3 perusteella puun korkeus eli algoritmin tilavaativuus h on

  vähintään

  $$\Omega(\log_2 n)$$

  enintään

  $$n - 1 = \mathcal{O}(n)$$

  muistipaikkaa.

- Rekursiivisten binääripuualgoritmien analyysissa arvioidaan tavallisesti

  aika summana

  $$\sum_{\text{vieraillut solmut } x} \text{solmussa } x \text{ vietetty aika}$$

  tila summana

  $$\sum_{\text{polun } p \text{ solmut } x} \text{solmun } x \text{ viemä tila}$$

  missä p on puun pisin polku.

3.1.3 Joukko-operaatioiden toteuttaminen

- Avaimen k etsiminen puusta tapahtuu kutsulla

  $$\text{search}(\text{root}[T], k)$$

  missä

  ```
  search(x, k)
      if x = NIL or key[x] = k then
          return x
      if k < key[x] then
          return search(left[x], k)
      else
          return search(right[x], k)
  ```

- Etsintä siis etenee juuresta alaspäin
  - pahimmassa tapauksessa puun korkeuden verran
  - eli aikavaativuus on

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

    missä h on puun korkeus.

- Lauseen 3.3 mukaan *aika*vaativuus on siis

  vähintään

$$\Omega(\log_2 n)$$

  enintään

$$n - 1 = \mathcal{O}(n)$$

  missä $n =$ solmujen lukumäärä.

- Myös rekursiopinon korkeus on pahimmillaan sama kuin puun korkeus, eli myös *tila*vaativuus on sama.

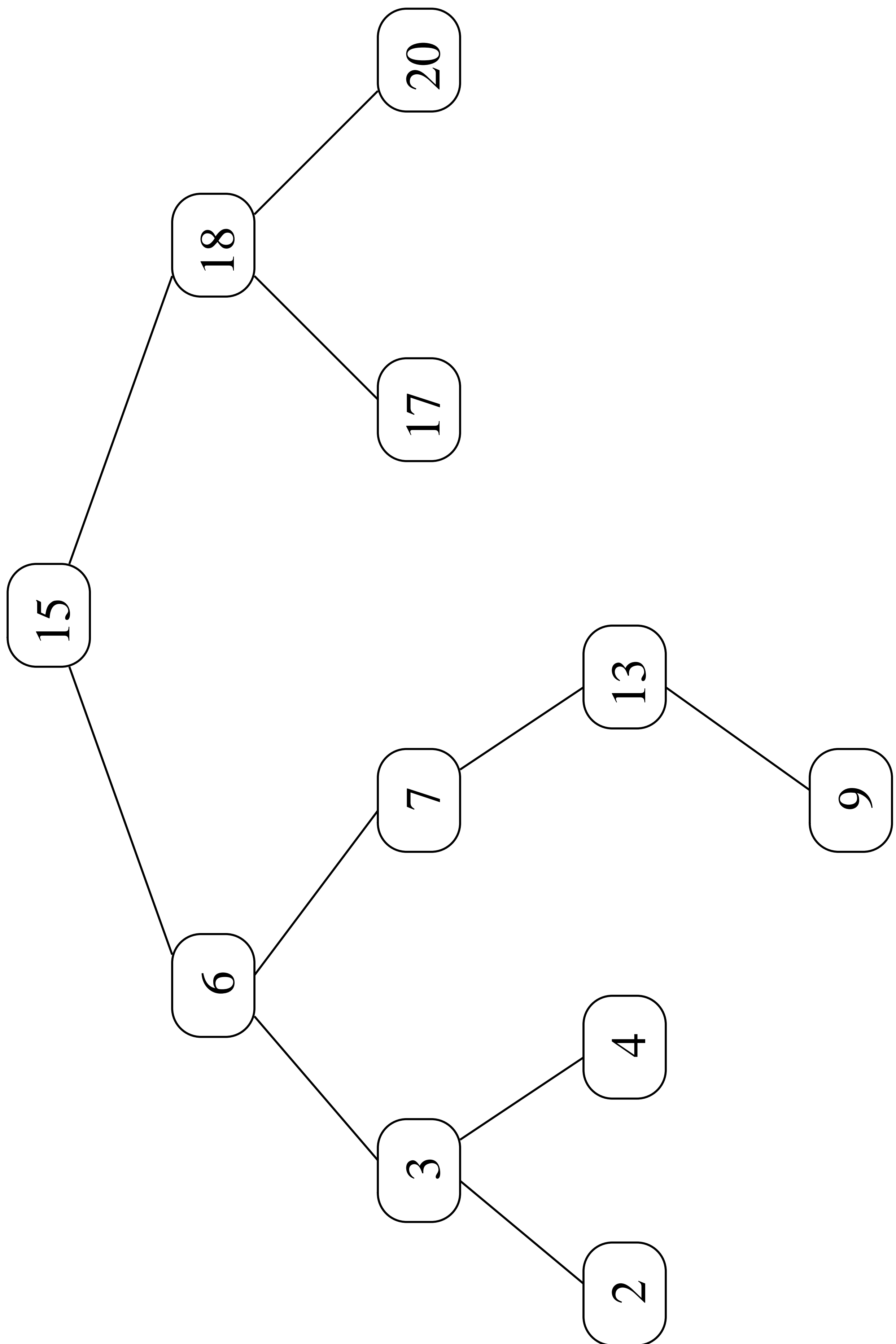
- Seuraavassa kuvassa

  – search(root$[T]$, 13) etenee polkua

$$15 \to 6 \to 7 \to 13$$

  – search(root$[T]$, 10) polkua

$$15 \to 6 \to 7 \to 13 \to 9 \to \text{NIL}\,.$$


15
18
17
20
6
3
2
4
7
13
9

- Avaimen etsiminen on helppo toteuttaa myös ilman rekursiota:

  search(x, k)
      while $x \neq$ NIL **and** key[x] $\neq k$ **do**
          if $k <$ key[x]
              then $x \leftarrow$ left[x]
              else $x \leftarrow$ right[x]
      return x

- Rekursion korvaaminen silmukalla onnistui, koska kaikki kutsut olivat erityistä muotoa

  return search(. . .)

  jota kutsutaan *taka*- eli *häntä*rekursioksi.

- *Aika*vaativuus on edelleen

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

  mutta *tila*vaativuus onkin enää

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

  koska rekursiopinosta päästiin eroon.

- Joissakin ohjelmointikielissä (ei Javassa) tämä optimointi tapahtuu automaattisesti.

- Binäärihakupuuehdosta seuraa suoraan että

  jos puussa kuljetaan aina vain vasemmalle

  niin lopulta päädytään sen *pienimpään* alkioon

  joten

  min(x)
      while left[x] $\neq$ NIL **do**
          $x \leftarrow$ left[x]
      return x

  ja kääntäen

  max(x)
      while right[x] $\neq$ NIL **do**
          $x \leftarrow$ right[x]
      return x

- Aika- ja tilavaatimuudet ovat samat kuin operaatiolla search

  (sen iteratiivisella versiolla).

- Edellisessä kuvassa

  minimin haku etenee polkua

  $$15 \rightarrow 6 \rightarrow 3 \rightarrow 2$$

  maksimin polkua

  $$15 \rightarrow 18 \rightarrow 20.$$

- Operaation succ toteutus on hieman haastavampi:

```
succ(x)
    if right[x] ≠ NIL then
        return min(right[x])
    else
        y ← p[x]
        while y ≠ NIL and x = right[y] do
            x ← y
            y ← p[x]
        return y
```

  Jos erityisesti x on koko puun suurin alkio, niin palautusarvo on $y = \text{NIL}$.

Jos solmun x oikea alipuu on epätyhjä niin solmun seuraaja on sen pienin alkio.

Edellisessä kuvassa solmun 15 seuraaja on sen oikean alipuun pienin, eli solmu 17.

Jos oikea alipuu onkin tyhjä niin

- kuljetaan solmusta x takaisin juurta kohti
- kunnes on otettu yksi askel *oikealle*
- koska niin kauan kuin kuljetaan *vasemmalle*, niin pysytään siinä alipuussa T', jonka oikeanpuolimmainen eli suurin avain key[x] on
- ja halutaan se solmu y, jonka vasen alipuu on T'.

Edellisessä kuvassa solmun 13 seuraaja löytyy paluureittiä

$$13 \xrightarrow{\text{vas.}} 7 \xrightarrow{\text{vas.}} 6 \xrightarrow{\text{oik.}} 15$$

kun taas

$$17 \xrightarrow{\text{oik.}} 18.$$

- Tämä operaation succ toteutus edellyttää isälinkit $p[x]$.

- Operaation succ pahin tapaus vie *aika*a

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

koska voidaan joutua

joko laskeutumaan juuresta aina lehteen

tai nousemaan lehdestä aina juureen

asti, eli koko puun korkeuden h verran.

- *Tila*a tarvitaan

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

muistipaikkaa.

- Operaatio pred on vasen/oikea-symmetrinen.

- Alkion lisääminen binäärihakupuuhun on hieman pidempi mutta suoraviivainen algoritmi:

```
insert(T, k)
1   z ← new puusolmu
2   key[z] ← k
3   left[z] ← NIL
4   right[z] ← NIL
5   x ← root[T]
6   y ← NIL
7   while x ≠ NIL do
8       y ← x
9       if k < key[x]
10          then x ← left[x]
11          else x ← right[x]
12  p[z] ← y
13  if y = NIL
14      then root[T] ← z
15      else if k < key[y]
16          then left[y] ← z
17          else right[y] ← z
```

- Riveillä 1–4 luodaan ja alustetaan uusi puusolmu z lisättävälle avaimelle k.

- Riveillä 5–11 etsitään puusta T sellainen solmu y, jonka lapseksi solmu z voidaan lisätä.

  - Toimitaan kuten hakuoperatiossa search

  - joka nyt epäonnistuu koska k ei vielä esiinny puussa

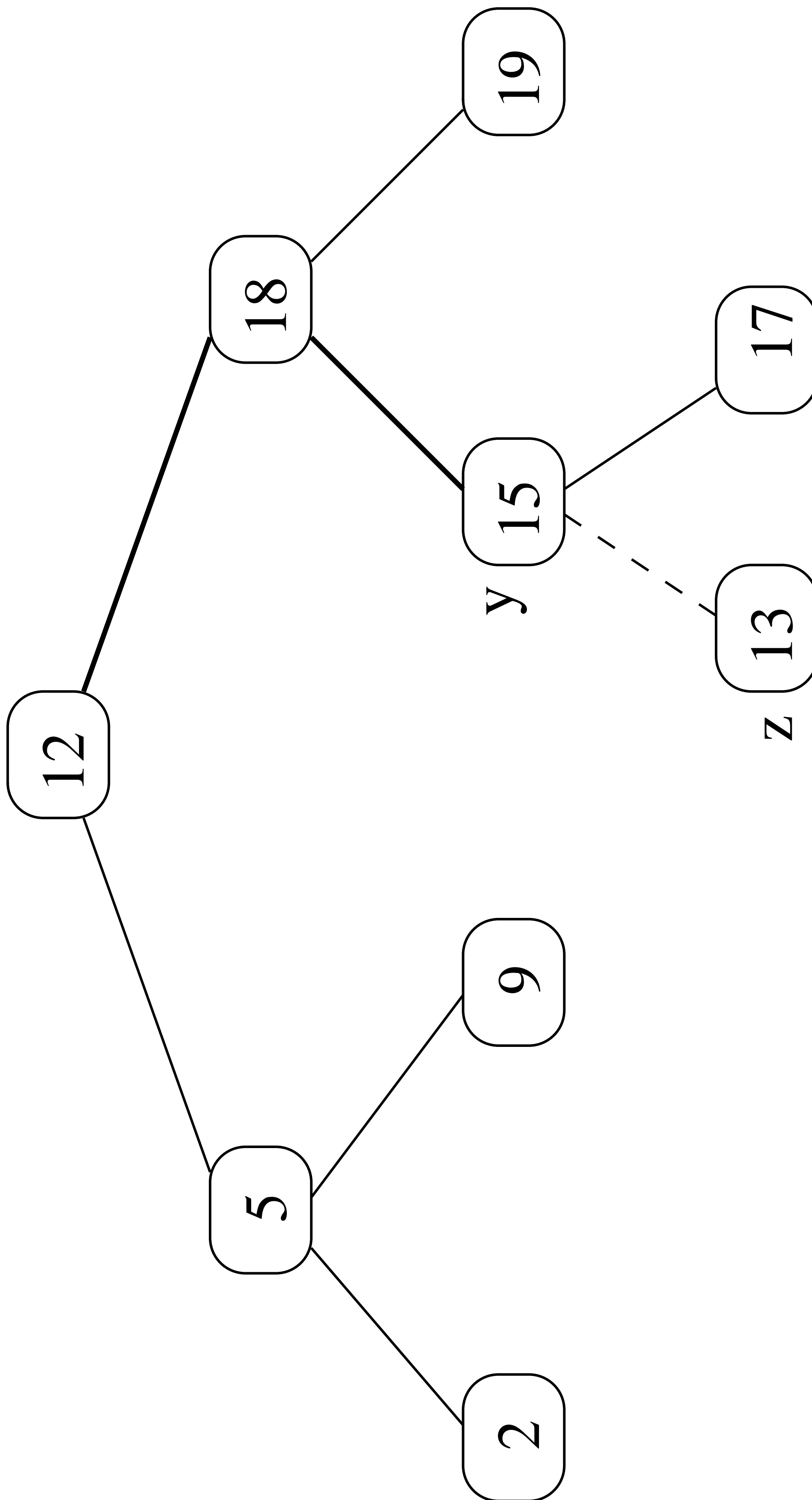
  - samalla muistetaan viitteen x edellinen arvo y (riveillä 6 ja 8).

- Riveillä 13–14 huolehditaan erikoistilanteesta, jossa

  - solmua y ei olekaan olemassa

  - koska koko puu T olikin tyhjä.

- Seuraavassa kuvassa on lisätty 13. Etsintäpolku on **tummennettu**, ja lisätty linkki on katkoviivana.


12
18
19
15
17
13
y
z
5
9
2

- Kuten muillakin tähän asti kohtaamillamme operaatioilla, on lisäyksenkin *aika*vaativuus

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

  – missä $h =$ puun T korkeus

  – sillä pahimmassa tapauksessa lisäys tehdään alimmalla tasolla olevan solmun y lapseksi.

- *Tila*vaativuus on

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

  sillä rekursiopino ei ole käytössä.

- Poisto on puu-operaatioista monimutkaisin:

  argumenttina saadaan puu T ja siitä poistettava solmu z

  tuloksena palautetaan oikeasti poistettu solmu y.

  – Siis voi olla $y \neq z$!

  – Ohjelm(oij)a voi vapauttaa solmun y varaaman muistin.

  – Jos halutaan poistaa täsmälleen solmu z (eikä y), niin solmu y siirretään solmun z tilalle puussa.

- Algoritmi jakautuu kolmeen tapaukseen sen mukaan, montako lasta solmulla z on:

  0: rivit 1–7

  1: rivit 8–18

  2: rivit 19–27.

```
delete(T, z)
1   if left[z] = NIL and right[z] = NIL then
2       w ← p[z]
3       if w = NIL then root[T] = NIL
4       else if z = left[w]
5               then left[w] ← NIL
6               else right[w] ← NIL
7       return z
8   if left[z] = NIL or right[z] = NIL then
9       if left[z] ≠ NIL
10          then x ← left[z]
11          else x ← right[z]
12      w ← p[z]
13      if w = NIL then root[T] = x
14      else if z = left[w]
15              then left[w] ← x
16              else right[w] ← x
17      p[x] ← w
18      return z
19  y ← succ(z)
20  x ← right[y]
21  w ← p[y]
22  if y = left[w]
23      then left[w] ← x
24      else right[w] ← x
25  if x ≠ NIL then p[x] ← w
26  key[z] ← key[y]
27  return y
```

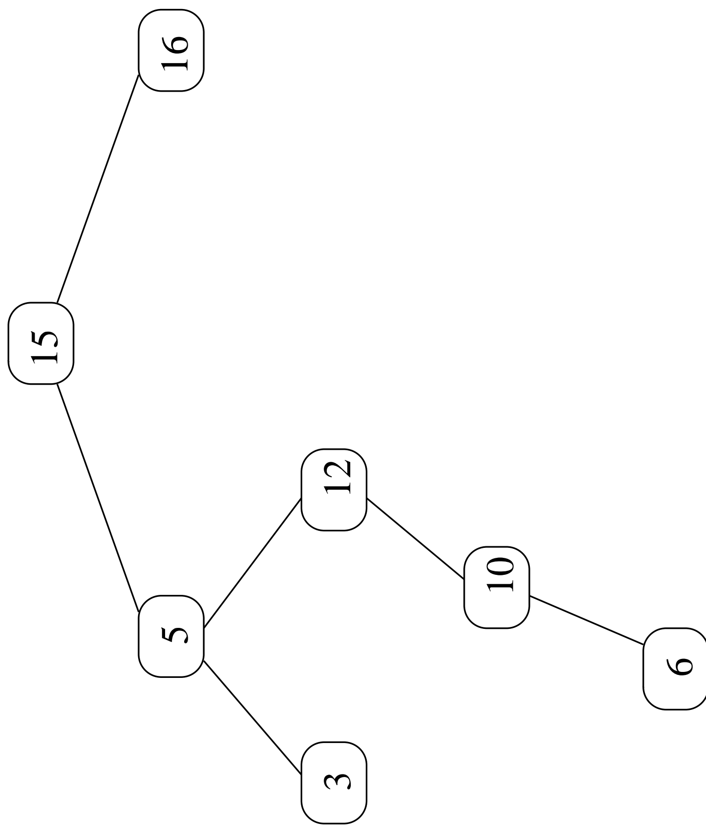
Jos solmu z on lapseton niin se voidaan poistaa, eikä muita muutoksia tarvita.

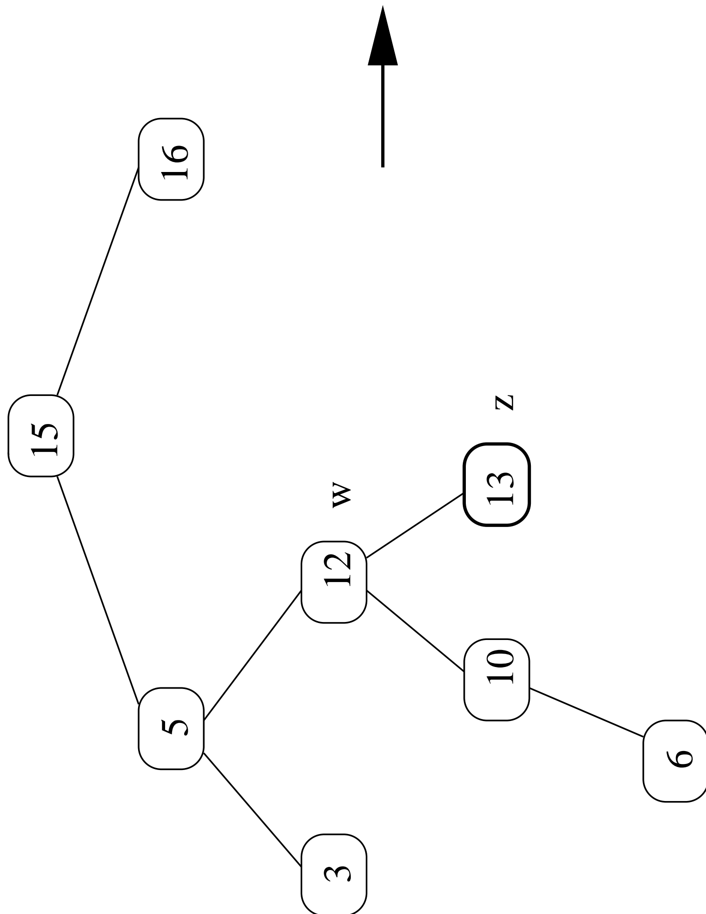
Jos solmulla z on vain yksi lapsi x niin solmu x ottaa solmulta z vapautuvan paikan puussa.

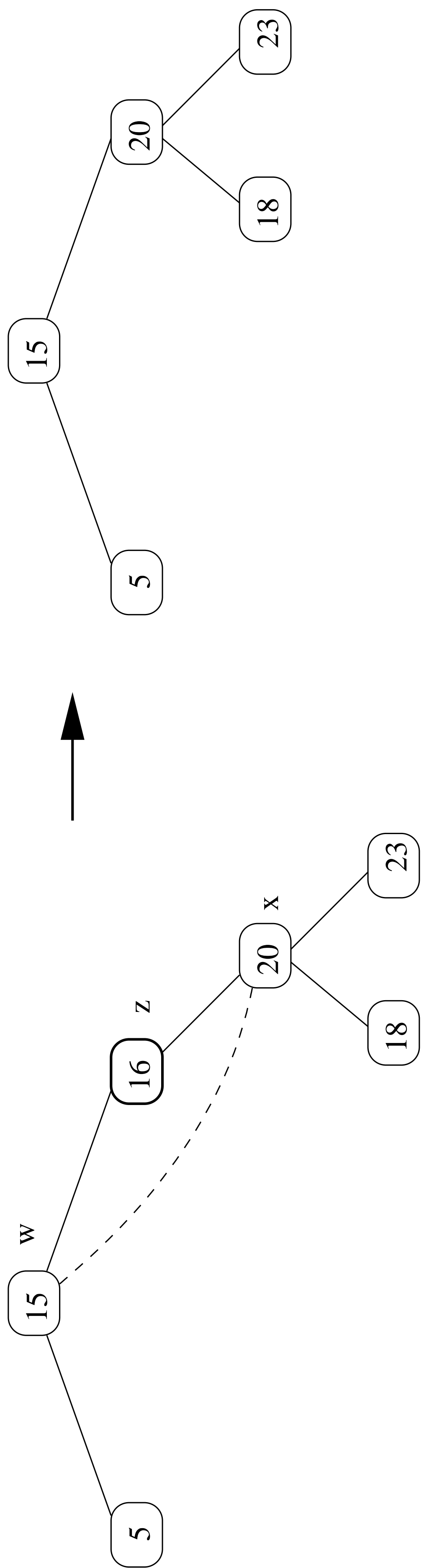
Jos solmulla z on molemmat lapset niin sitä itseään ei voikaan poistaa.

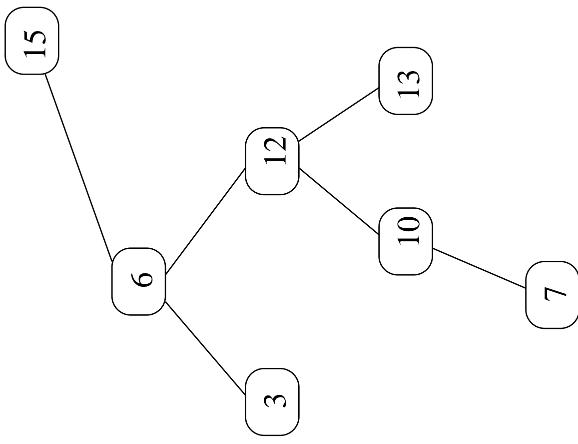
- Sen sijaan poistetaankin sen seuraajasolmu $y = \text{succ}(z)$.
- Solmu $y = \text{min}(\text{right}[z])$ on välttämättä olemassa.
- Solmulla y ei voi olla ainakaan lasta left$[z]$.
- Solmu y voidaan siis poistaa kuten yllä.
- Avain key$[z]$ voidaan korvata avaimella key$[y]$

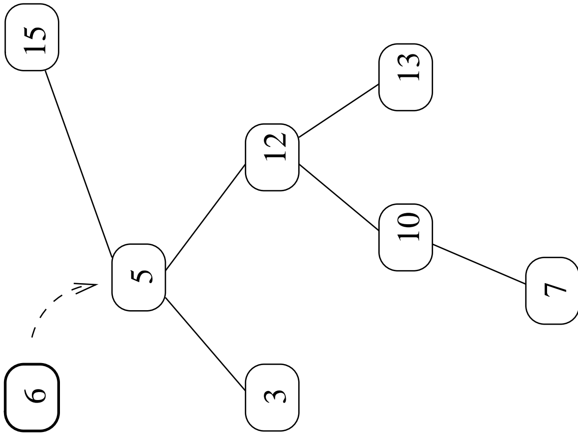
  koska niiden välissä ei ole muita avaimia.

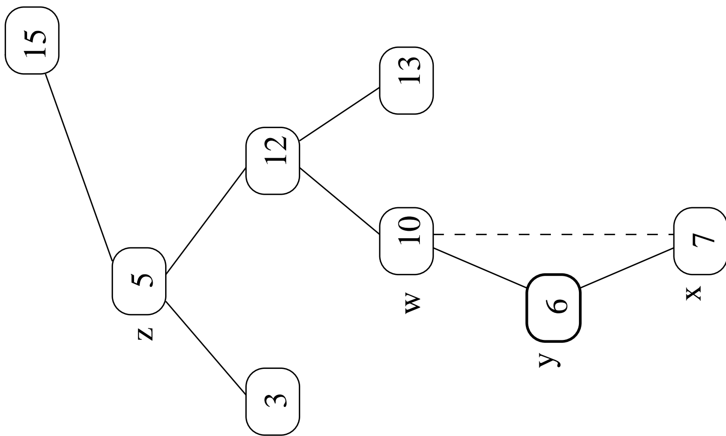

16
15
12
5
10
6
3


16
15
z
w
13
12
5
10
6
3


15
5
20
18
23
w
z
x
15
16
20
5
18
23


15
13
12
6
10
7
3


15
13
12
5
10
7
6
3


15
13
12
5
10
7
z
w
6
x
y
3

- Edellisisissä kuvissa on esitettu 0-, 1- ja 2-lapsisen solmun z poisto.

- Poistonkin *aika*vaativuus on

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

missä $h =$ puun T korkeus:

kuljetaan yksi polku juuresta mahdollisesti lehtisolmuun ja kussakin solmussa suoritetaan vakiomäärä operaatioita.

- Poiston *tila*vaativuus on

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

- Kirjassa Cormen et al. poisto on esitetty hieman eri tavalla:

  - eri tapaukset ovat yhä samat mutta lomittuneet toisiinsa

  - näin saadaan lyhyempi mutta vaikeaselkoisempi pseudokoodi.

- Kalvojen 1.5 abstraktin tietotyypin **joukko** jokainen operaatio siis pystytään tekemään

  ajassa $\mathcal{O}(h)$

  tilassa $\mathcal{O}(1)$

  missä $h =$ puun T korkeus.

- Lauseen 3.3 perusteella n-solmuisen puun korkeus on välillä

$$\log_2(n+1) - 1 \leq h \leq n - 1.$$

- Jos puu on *tasapainoinen*

  – eli jos

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

  – niin puu on oleellisesti parempi toteutus joukolle kuin kalvojen 2.4 listat.

- Hyvin *epä*tasapainoisessa puussa

$$h = \Omega(n)$$

  ja puu on listan luokkaa (mutta monimutkaisempi ohjelmoida).

- Huono puu syntyy vaikkapa lisäämällä tyhjään puuhun T avaimet $1, 2, 3, \ldots, n$ (suuruus)järjestyksessä.

  Tai käänteisessä järjestyksessä.

  Tai patologisen insert/delete-kutsuyhdistelmän takia.

- Ongelma: *miten voisimme taata että hakupuumme pysyvät tasapainoisina?*

3.2 Tasapainoisia puulajeja

AVL-puu (Adel'son, Vel'skiĭ ja Landis)

- Vanhin (1962).
- Esitteli hakupuun rakenteen muokkaukseen *kierron* (rotation)

  jota mekin käytämme kalvoilla 3.3.1.
- Joka solmuun on talletettu siitä alkavan alipuun korkeus.

\+ Intuitiivinen tasapainoehto:

solmun alipuiden korkeuksien ero ≤ 0.

– Algoritmeissa paljon erikoistapauksia:

esimerkiksi poisto ohitetaan useissa oppikirjoissa liian mutkikkaana...

– Yksi operaatio voi tehdä monta kiertoa.

- Se vaikeuttaa käyttämistä muiden tietorakenteiden *toteutusalustana*
- koska kierron yhteydessä voi joutua tekemään muutakin.

Splay-puu tekee kiertoja siten, että tavallisimmin kysyttyjen avainten solmut lähestyvät juurta.

+ Ei lainkaan solmukohtaista tasapainotietoa.

– Ei takaa jokaisen operaation nopeutta:

  - Jos tätä avainta ei ole kysytty pitkään aikaan, niin sen solmun löytäminen saattaa kestää pidempään.

  - Toisaalta suorituskyky koko ohjelman mitassa on hyvä, koska tavallisimmin kysytyt solmut löytyvät nopeasti.

2-3-4-puu ei olekaan binääripuu (kuten nämä muut) vaan tasapaino saavutetaankin solmun lapsien lukumäärää muuttelemalla.

+ Sen yleistys *B-puu* (kalvot 3.5) on erittäin hyvä jos hakupuuta ylläpidetään kovalevyllä.

– Keskusmuistissa NIL-viitteet puuttuviin lapsiin vievät turhaan tilaa.

Punamusta puu on valittu tälle kurssille (kalvot 3.3).

+ Joka solmussa on tasapainotietoa vain 1 bitti.
  - Vähemmän kuin AVL-puussa.
  - Onko enää nykyään tärkeä ominaisuus?

+ Jokainen operaatio tekee vain vakiomäärän kiertoja.
  - Toisin kuin AVL-puussa.
  - Soveltuu paremmin toteutusalustaksi.
  - Sitä voidaankin pitää 2-3-4-puun toteutusalustana.

+ Jokainen operaatio on aina tehokas toisin kuin Splay-puussa.

– Algoritmeissa on monia eri tapauksia.

3.3 Punamustat puut

- Kaikki binäärihakupuun operaatiot siis pystytään toteuttamaan

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

askeleessa, missä $h =$ puun korkeus.

- Jos voimme taata, että korkeus h on lähellä minimiään, eli

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

missä $n =$ puussa olevien solmujen lukumäärä

niin kaikkien operaatioiden aikavaativuus on todella hyvä.

- Intuitiivisesti: aikavaatimus on vain samaa luokkaa kuin numeromerkkejä silloin kun lukumäärä n kirjoitetaan paperille (binäärilukuna).

- *Punamusta* puu on binäärihakupuu, jossa jokaiseen solmuun x on lisätty uusi kenttä

$$\text{color}[x] = \text{solmun } x \textit{ väri}.$$

- Väri on joko *punainen* (RED) tai *musta* (BLACK).

- Rajoittamalla solmujen väritystä poluilla sisäsolmuista lehtiin voidaan taata että mikään polku ei ole yli kaksi kertaa pitempi kuin mikään muu.

  - Näin varmistutaan siitä että puu on riittävän tasapainossa.

    (Osoitamme tämän kohta.)

  - Tasapainoehto on kuitenkin samalla varsin salliva:

    * hyvin monen muotoiset puut kelpaavat

    * joten tasapainoon päästään vähällä lisätyöllä.

- Koko puuhun T liittyy oma nil-solmu nil[T].
  - Sen väri color[nil[T]] = BLACK.
  - Sen muiden kenttien arvoilla ei ole väliä.
    - Niitä voidaan käyttää vapaasti.
    - Tämä yksinkertaistaa erikoistapausten käsittelyä algoritmeissa.
    - Erityiesti voidaan kysyä solmun väriä tarkistamatta ensin onko se olemassa.
  - Kuin kalvojen 2.4.2 tunnussolmu listassa.
  - Jos puun T solmulla x ei ole vasenta lasta left[x], niin silloin

    $$\text{left}[x] = \text{nil}[T]$$

    ja samoin oikealle lapselle right[x] sekä isälle $p[x]$.

- Siis punamustan puun jokainen (eli ainoa) lehti on nil[T].

- Kaikki tieto talletetaan *sisä*solmuihin (eli niihin solmuihin jotka eivät ole lehti(ä)).

- Binäärihakupuu on punamusta puu, jos seuraavat ehdot ovat voimassa:

  1. Jokainen solmu on joko punainen tai musta.

  2. Juurisolmu on musta.

  3. Jokainen lehti (nil-solmu) on musta.

  4. *Jos solmu on punainen, sen molemmat lapset ovat mustia.*

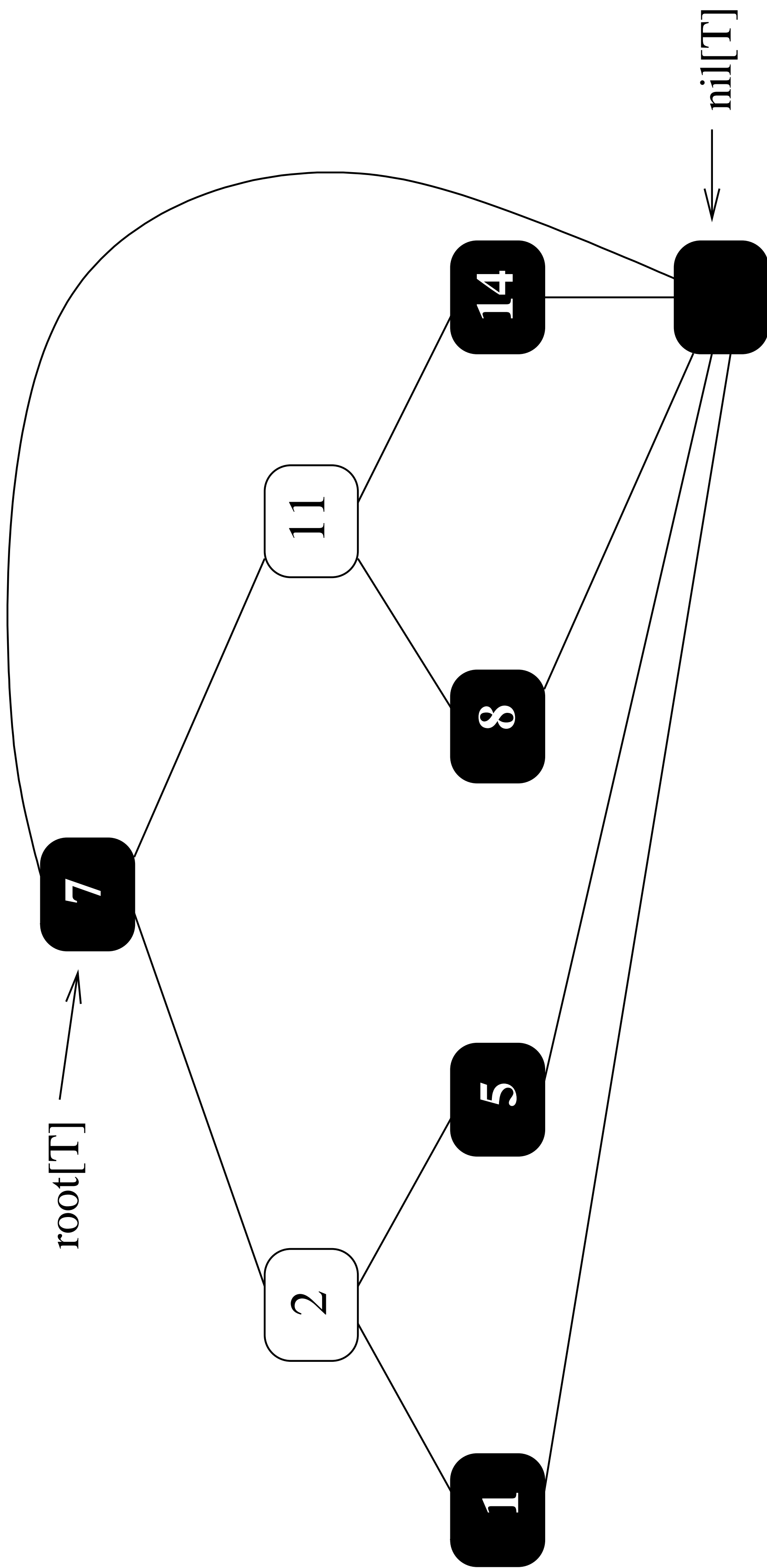
  5. *Jokainen polku tietystä solmusta sen lehtiin sisältää saman määrän mustia solmuja.*

- Seuraavassa kuvassa on eräs punamusta hakupuu T.

- Yleensä emme piirrä kuviin solmua nil[T] vaan

  joko jätämme sen piirtämättä

  tai piirrämme jokaiselle oman NIL-solmun (tai -linkin).


nil[T]
14
11
8
7
root[T]
5
2
1

- Määritellään solmun x *mustakorkeudeksi* mustien solmujen lukumäärä jollain polulla solmusta x lehteen.
  - Solmua x itseään ei lasketa mukaan.
  - Merkitään $\mathrm{bh}(x)$.

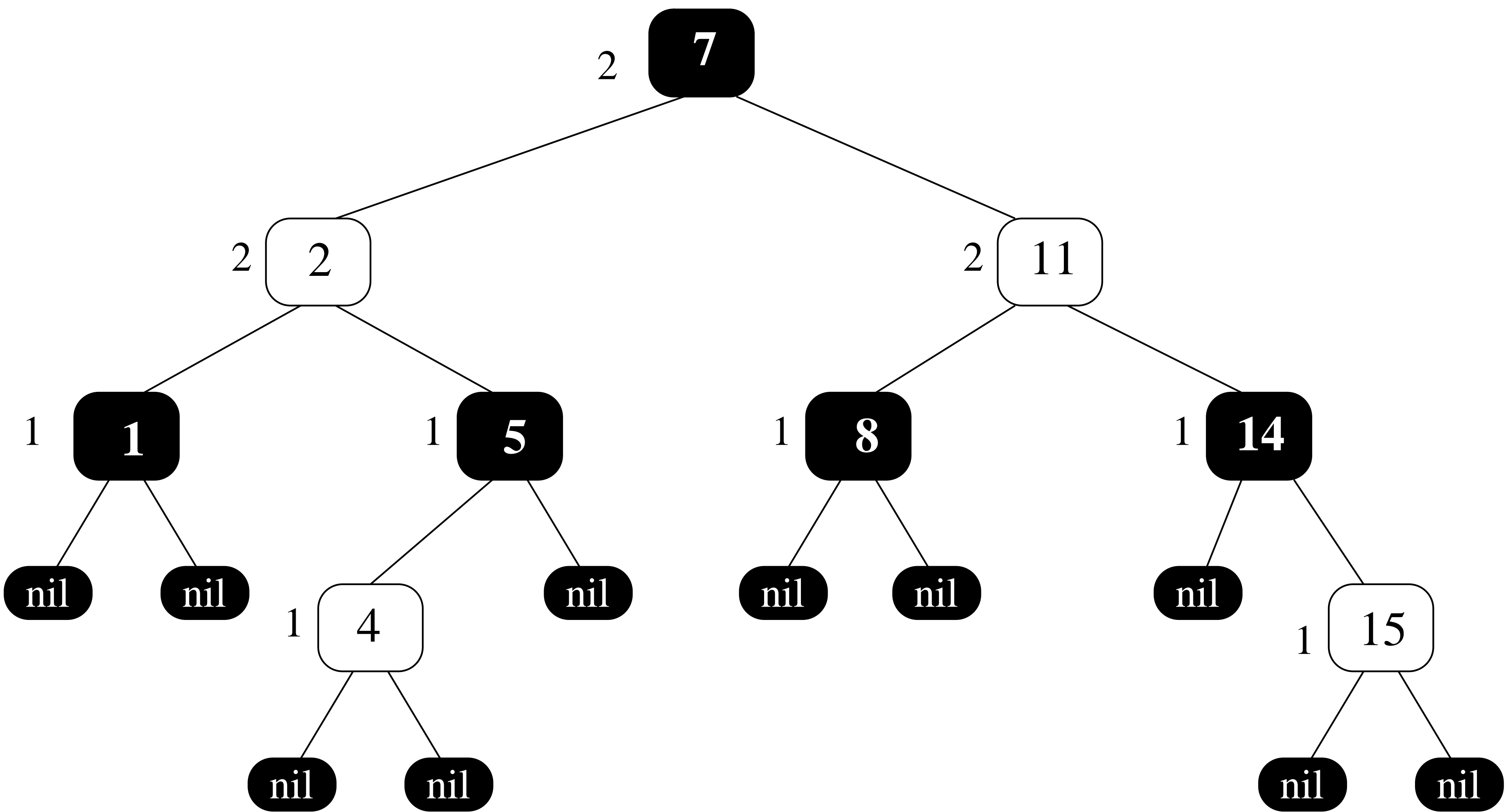
  Punamustaehto 5 takaa, että $\mathrm{bh}(x)$ on hyvin määritelty: tulos on sama riippumatta siitä mitä polkua pitkin se laskettiin.

- Koko puun T mustakorkeus

$$\mathrm{bh}(T) = \mathrm{bh}(\mathrm{root}[T]).$$

- Seuraavassa kuvassa on sama puu, jossa jokaisen sisäsolmun viereen on merkitty sen mustakorkeus.

- Tarkastellaan sitten mustakorkeuden ja sisäsolmujen lukumäärän välistä suhdetta.


2
7
2
2
2
11
1
1
1
5
1
8
1
14
nil
nil
1
4
nil
nil
nil
nil
1
15
nil
nil
nil
nil

Lause 3.5. *Jos x on punamustan puun T solmu, niin solmusta x alkavassa alipuussa on ainakin*

$$2^{\mathrm{bh}(x)} - 1$$

sisäsolmua.

Todistus: Edetään induktiolla solmun x *tavallisen* korkeuden k suhteen.

- Jos $k = 0$, niin x on lehtisolmu nil$[T]$. Silloin sen alipuussa on $2^0 - 1 = 0$ sisäsolmua.

- Jos $k > 0$, niin x on sisäsolmu. Sillä on näin ollen 2 lapsisolmua (erityisesti myös nil$[T]$ on solmu).

  Lapsen mustakorkeus on määritelmän nojalla joko $\mathrm{bh}(x)$ tai $\mathrm{bh}(x) - 1$.

  Koska lapsen korkeus on pienempi kuin isän x, niin induktio-oletuksen mukaan lapsen alipuussa on ainakin $2^{(\mathrm{bh}(x)-1)} - 1$ sisäsolmua.

  Siis isän x puussa on ainakin $2 \cdot (2^{(\mathrm{bh}(x)-1)} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ sisäsolmua. □

- Edellisen aputuloksen nojalla voimme vihdoin päätellä, että punamustapuu on todellakin siinä mielessä tasapainoinen kuin halusimmekin.

Lause 3.6. *Jos punamustalla puulla T on n sisäsolmua, niin sen korkeus on enintään $2 \cdot \log_2(n+1)$.*

Todistus: Olkoon koko puun T korkeus h. Punamustaehdon 4 nojalla sen $\text{bh}(T) \geq h/2$.

Silloin lauseen 3.5 nojalla koko puussa T on ainakin $2^{h/2} - 1$ sisäsolmua.

Saadaan epäyhtälö

$$\begin{aligned}
2^{h/2} - 1 &\leq n \\
2^{h/2} &\leq n + 1 \\
\log_2\left(2^{h/2}\right) &\leq \log_2(n+1) \\
h/2 &\leq \log_2(n+1) \\
h &\leq 2 \cdot \log_2(n+1).
\end{aligned}$$

□

- Edellisestä lauseesta 3.6 seuraa heti:
  - kalvojen 1.5 abstraktin tietotyypin **joukko** voi toteuttaa siten, että
  - sen jokainen operaatio voidaan tehdä ajassa

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

    missä $n =$ joukon alkioiden lukumäärä
  - eli taulukon sarakkeen **tasap. puu** mukaisena
  - käyttämällä toteutuksena punamustia hakupuita.

- ”Enää” on vain huolehdittava että solmun lisäys ja poisto säilyttävät saamansa puun punamustana. . .

3.3.1 Kierrot

- Alkion lisäys tai poisto punamustaan puuhun siis saattaa rikkoa punamustaehtoja.

- Lisäys punamustaan puuhun tehdään

  aluksi samalla tavalla kuin yleiseen hakupuuhun kalvoilla 3.1.3:

  - etsitään lisäyskohta etenemällä puussa *alaspäin*

  - linkitetään uusi solmu löydettyyn kohtaan

  lopuksi uutena *jälkikäsittelyvaihe*ena:

  - palataan puussa takaisin lisäyskohdasta *ylöspäin* ja

  - matkalla *muokataan paikallisesti* puun muotoa ja värejä, jotta punamustaehdot saadaan taas voimaan.

- Poisto noudattaa samaa periaatetta.

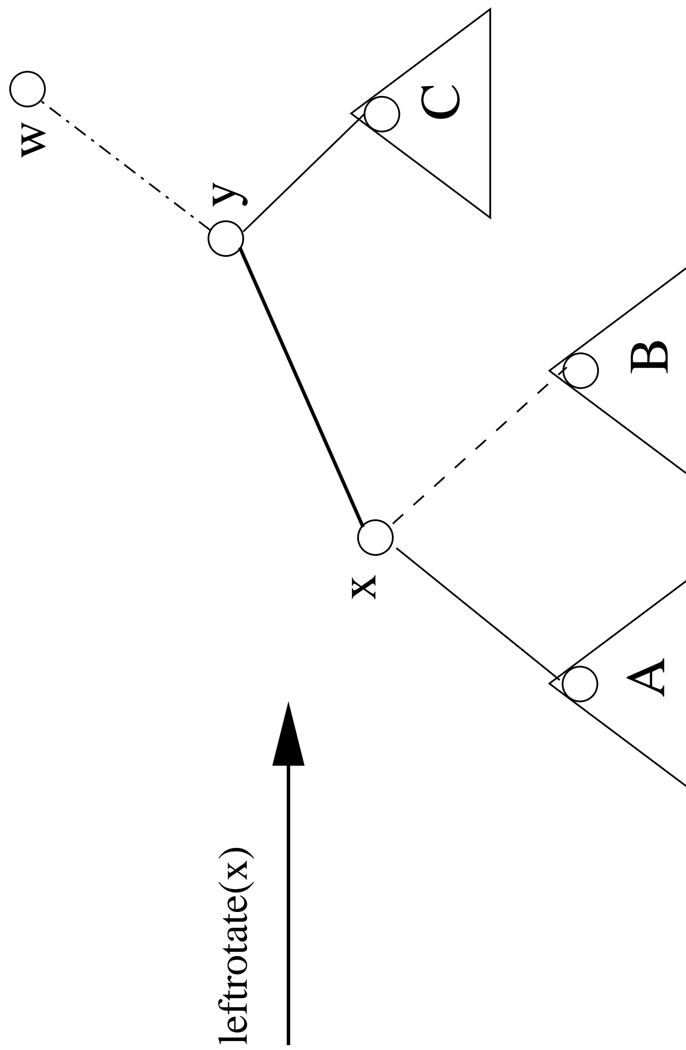
- Tarvitaan siis paikallisia puunmuokkaustoimenpiteitä, jotka
  - säilyttävät binäärihakupuuehdon
  - toimivat ajassa
  $$\mathcal{O}(1).$$
- *Kierrot* ovat tällaisia toimenpiteitä.
- Yhdessä kierrossa puun jokin isä-lapsi-suhde käännetäänkin päinvastaiseksi.
- Punamustaehdoista huolehditaan
  - tekemällä sopivat kierrot sopivissa kohdissa puuta
  - missä sopivat kohdat päätellään solmujen väreistä.

  Kierron jälkeen päivitetään värit vastaamaan sen tulosta.

- Esimerkiksi solmun x *vasen kierto:*
  - Nykyinen isäsolmu x haluaa muuttua lapsensa y vasemmaksi lapseksi.
  - Hakupuuehdon säilymiseksi
    * lapsen y pitää olla right[x]
    * alipuun left[y] pitää siirtyä alipuuksi right[x]

      eli alipuun y tilalle.

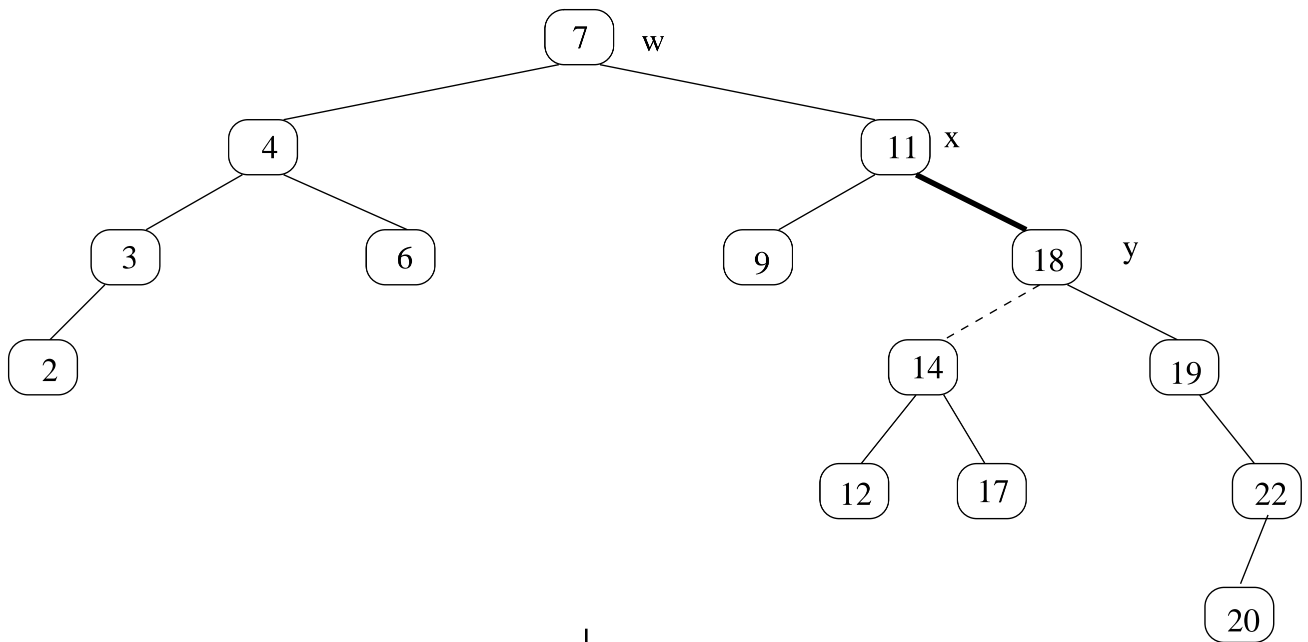
```
leftrotate(T, x)
1   y ← right[x]
2   right[x] ← left[y]
3   if left[y] ≠ nil[T] then p[left[y]] ← x
4   w ← p[x]
5   p[y] ← w
6   if w = nil[T]
7       then root[T] ← y
8       else if left[w] = x
9               then left[w] ← y
10              else right[w] ← y
11  left[y] ← x
12  p[x] ← y
```


w
y
C
B
x
A
leftrotate(x)
w
y
C
B
x
A

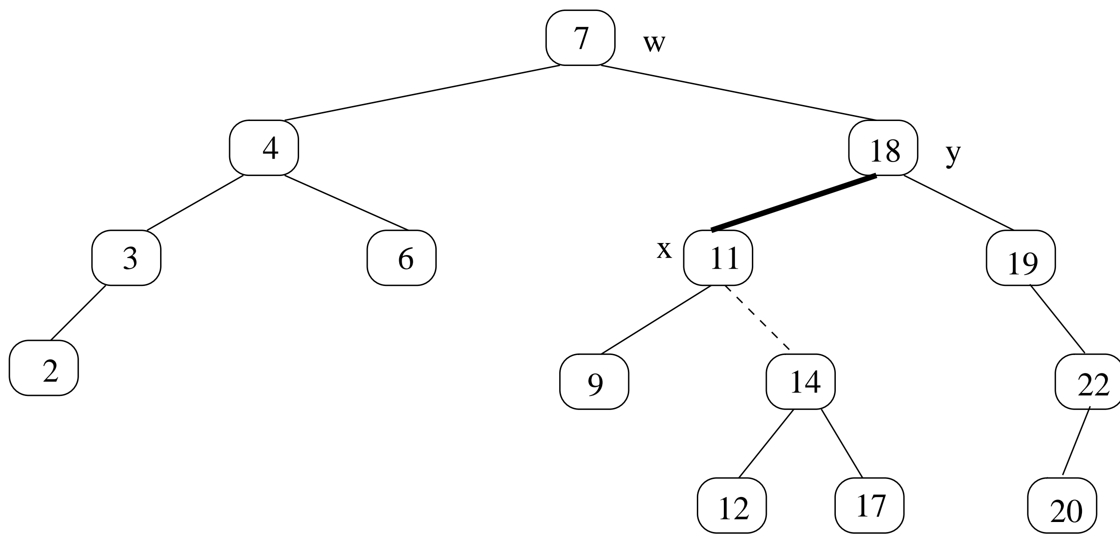
- Seuraavassa kuvassa on konkreettinen esimerkki:
  - Solmujen värejä ei ole merkitty, eikä puu ole tasapainossa.
  - Huomaa miten kierto tasapainoittaa puun epätasapainossa olevaa kohtaa.
  - Hakupuuominaisuus säilyy kierrosta huolimatta.

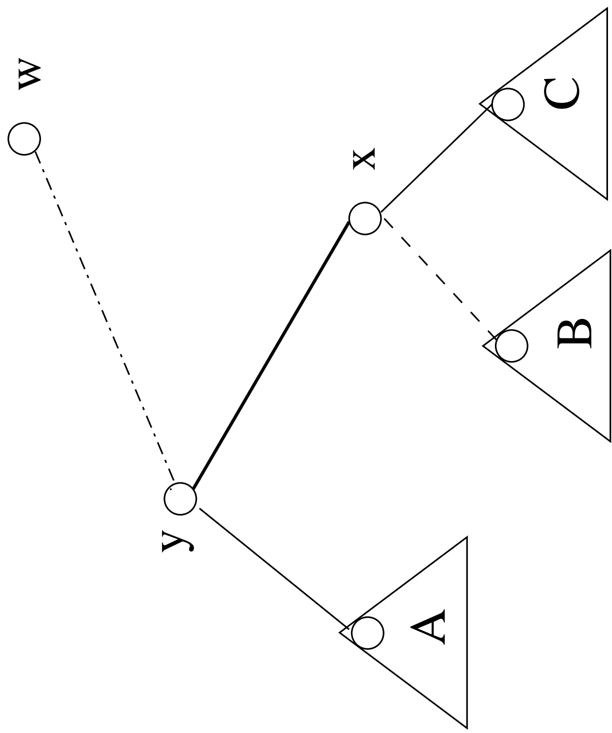
- *Oikea* kierto on vasen/oikea-symmetrinen:

```
rightrotate(T, x)
1   y ← left[x]
2   left[x] ← right[y]
3   if right[y] ≠ nil[T] then p[right[y]] ← x
4   w ← p[x]
5   p[y] ← w
6   if w = nil[T]
7       then root[T] ← y
8       else if left[w] = x
9               then left[w] ← y
10              else right[w] ← y
11  right[y] ← x
12  p[x] ← y
```

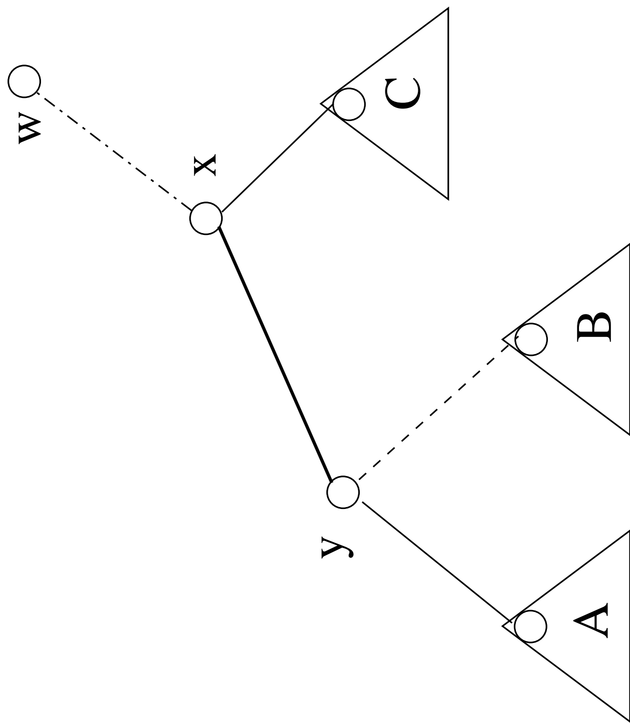


leftrotate(x)




w
x
C
B
y
A

rightrotate(x)


w
x
C
B
y
A

3.3.2 Värit säilyttävä kierto

- Kalvojen 3.3.1 kierto-operaatiot eivät liity väreihin.

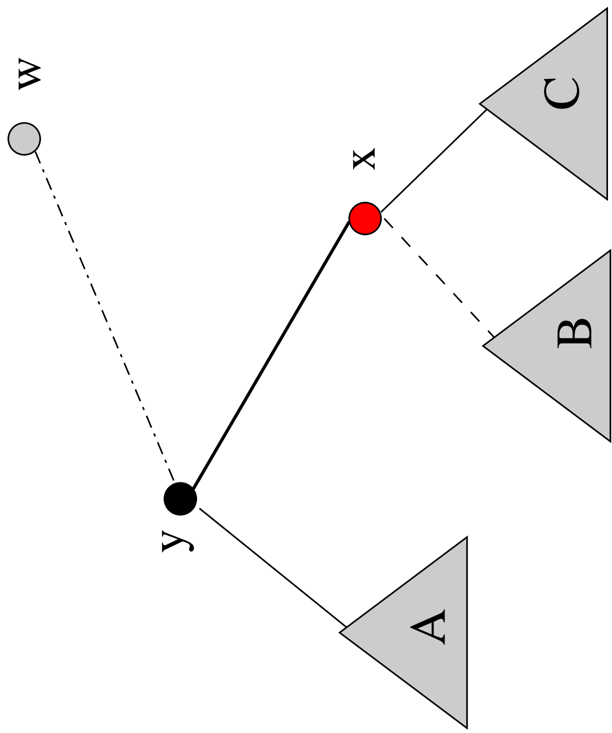
- Samoja kiertoja käytetään myös sellaisissa binäärihakupuissa, joiden tasapainotus ei perustu väreihin.

- Jatkoa varten tarvitaan sellainen kierto, joka ei muuta punamustaehtoa 5.

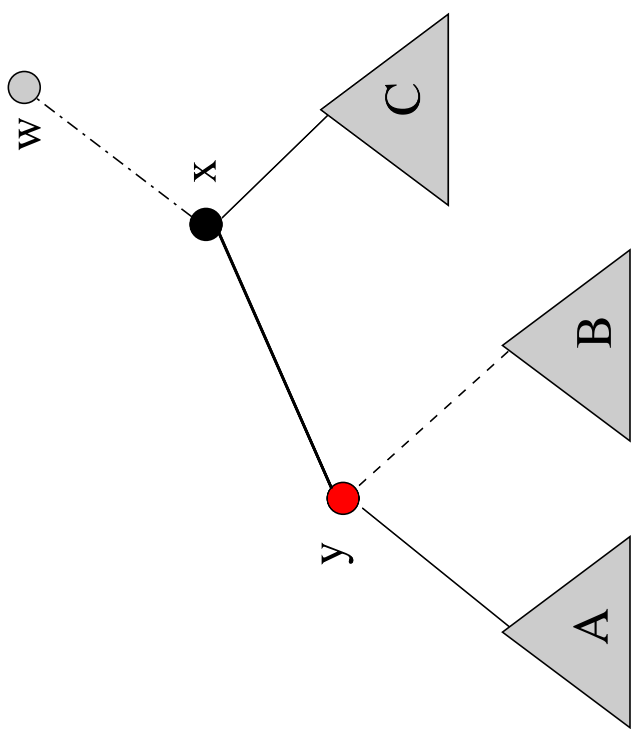
- Jos solmua x kierrettäessä *ylöspäin nouseva lapsi y on punainen*

  niin riittää *vaihtaa solmujen x ja y värit* keskenään.

- Kuvasta näkyy, että mustien solmujen lukumäärä pysyy silloin samana poluilla isäsolmusta w alipuihin A, B ja C.



rightrotate(x)



3.3.3 Alkion lisäys

- Avaimen k lisäys punamustaan hakupuuhun T:

```
insert(T, k)
1   z ← new punamustapuusolmu
2   key[z] ← k
3   left[z] ← nil[T]
4   right[z] ← nil[T]
5   x ← root[T]
6   y ← nil[T]
7   while x ≠ nil[T] do
8       y ← x
9       if k < key[x]
10          then x ← left[x]
11          else x ← right[x]
12  p[z] ← y
13  if y = nil[T]
14      then root[T] ← z
15      else if k < key[y]
16          then left[y] ← z
17          else right[y] ← z
18  color[z] ← RED
19  rbinsertfixup(T, z)
```

- Alkurivit 1–17 ovat samat kuin kalvojen 3.1.3 lisäyksessä tasapainottamattomaan hakupuuhun, paitsi että
  - rivillä 1 luodaankin solmu z punamustatyyppisenä
  - vakioviitteen NIL sijaan käytetäänkin puun T omaa nil-solmua nil[T].
- Rivillä 18 solmu z saa värikseen ”punainen”
  - vaikka se *saattaakin* rikkoa punamustaehdon 4 solmun z isässä y
  - koska ”musta” olisi *varmasti* rikkonut ehdon 5.
- Rivillä 19 ryhdytään korjaamaan tätä ehdon 4 mahdollista rikkoutumista.

```
rbinsertfixup(T, z)
1      while color[p[z]] = RED do
2          if p[z] = left[p[p[z]]] then
3              y ← right[p[p[z]]]
4              if color[y] = RED then
5                  color[p[z]] ← BLACK          ▷1
6                  color[y] ← BLACK             ▷1
7                  color[p[p[z]]] ← RED         ▷1
8                  z ← p[p[z]]                  ▷1
9              else if z = right[p[z]] then
10                     z ← p[z]                 ▷2
11                     leftrotate(T, z)         ▷2
12                 color[p[z]] ← BLACK          ▷3
13                 color[p[p[z]]] ← RED         ▷3
14                 rightrotate(T, p[p[z]])      ▷3
15         else
16-27      ▷rivit 3–14 vasen/oikea-symmetrisesti
28 color[root[T]] ← BLACK
```

- Tasapainotusrutiinin silmukan invariantissa on monta osaa:

  1. Solmu z on punainen.
  2. Kaari $z \to p[z]$ on ainoa kohta puussa, jossa punamustaehto 4 saattaa olla rikki.
  3. Punamustaehto 5 pidetään koko ajan voimassa

  sopivin kierroin ja värien vaihdoin.

- Jos punamustaehto 4 ei ole rikki edes invariantissa 2 (rivi 1), niin silmukasta saadaan poistua.

- Sen jälkeen varmistetaan punamustaehto 2 siltä varalta, että $z = \mathrm{root}[T]$ (rivi 28).

  Sitten koko puu on jälleen kunnossa.

- Tarkastellaan sitten, mitä yhdellä silmukkakierroksella tapahtuu.

- Koska isäsolmu $p[z]$ on punainen (rivi 1), niin sillä on isoisäsolmu $p[p[z]]$.

- Isoisäsolmu $p[p[z]]$ on musta.

- Käsittely jakautuu kahteen vasen/oikea-symmetriseen tapaukseen sen mukaan, onko isäsolmu $p[z]$ isoisäsolmun $p[p[z]]$ *vasen vai oikea* lapsi (rivit 2 ja 15).

- Tarkastellaan esimerkkinä vasen (rivit 3–14).

- Olkoon y solmun z setä (rivi 3).

- Käsittely jakutuu kolmeen tapaukseen

  joita vastaavat koodin osat on merkitty kommenteilla.

1. Setä y on *punainen* (rivi 4):

- Seuraava kuva esittää tämän tapauksen

  joka on samanlainen riippumatta siitä oliko solmu z itse vasen vai oikea lapsi.

- Valutetaan isoisäsolmun $p[p[z]]$ musta väri sen lapsiin $p[z]$ (rivi 5) ja y (rivi 6).

- Isoisäsolmusta $p[p[z]]$ paljastuu sen alkuperäinen punainen pohjaväri (rivi 7).

- Tarkistetaan invariantit:

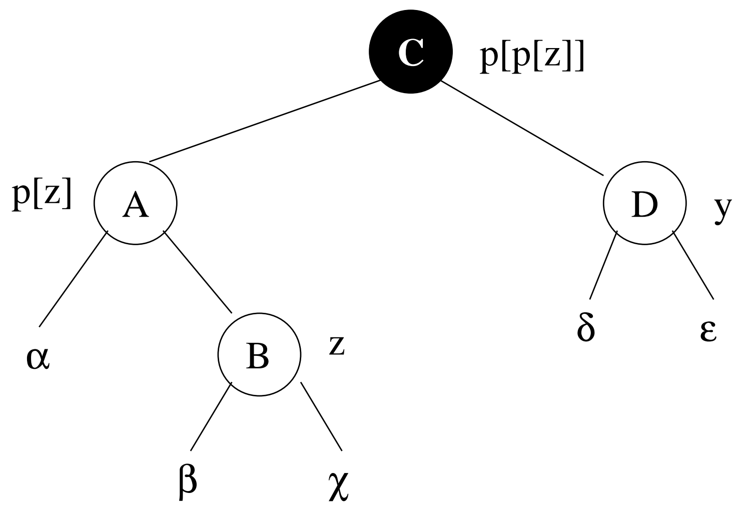
  - Mustien solmujen lukumäärä alaspäin vievillä poluilla ei muutu

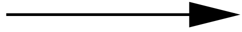
    joten invariantti 3 pysyy voimassa.

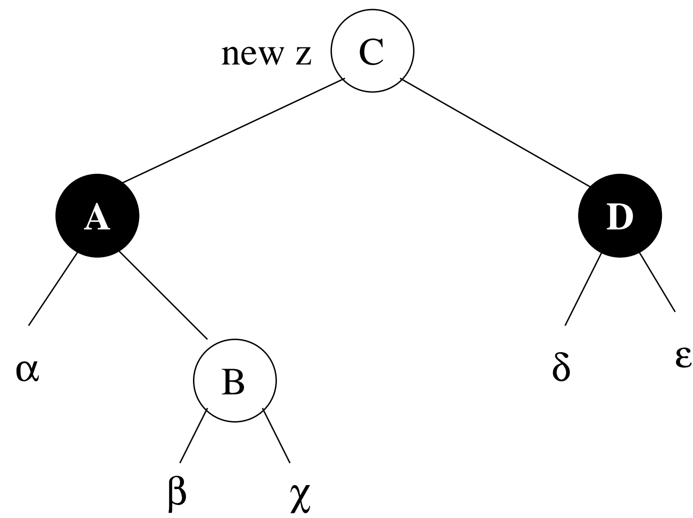
  - Kaikki kuvan kaaret täyttävät nyt punamustaehdon 4.

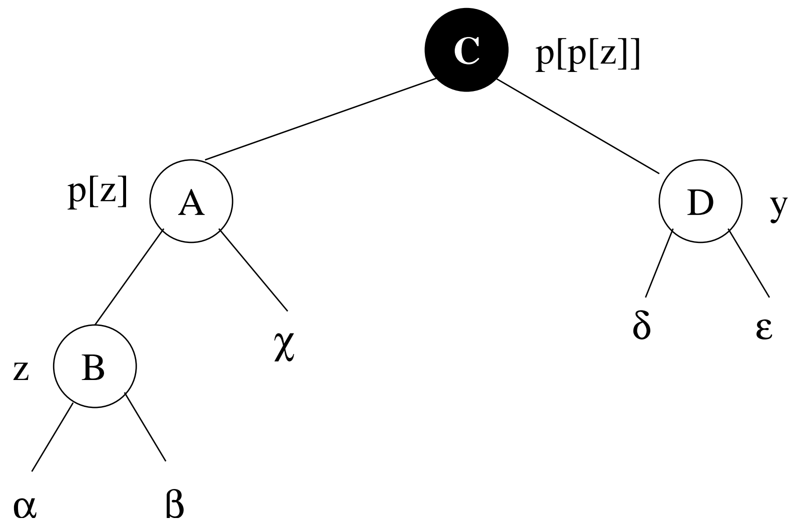
  - Ainoa ehdon 4 mahdollisesti rikkova kaari on isoisäsolmusta ylöspäin $p[p[z]] \to p[p[p[z]]]$.

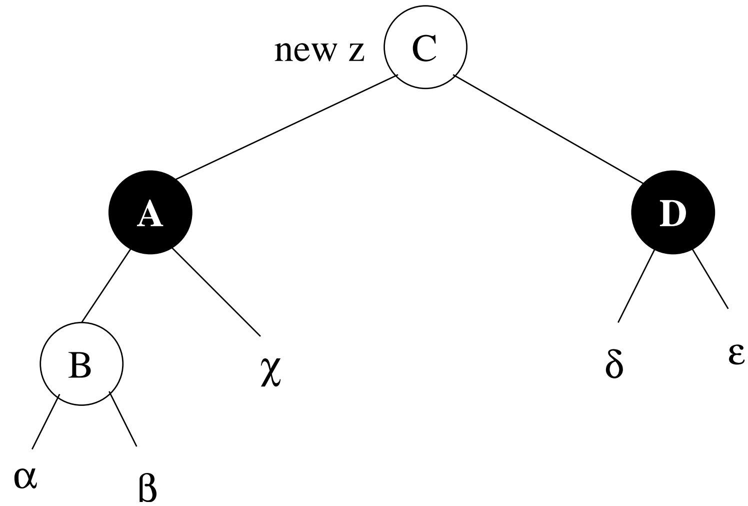
  - Siis jatketaan silmukkaa siten, että seuraava z onkin isoisäsolmu $p[p[z]]$ (rivi 8).


C
p[p[z]]
p[z]
A
D
y
α
B
z
δ
ε
β
χ




new z
C
A
D
α
B
δ
ε
β
χ


C
p[p[z]]
p[z]
A
D
y
z
B
χ
δ
ε
α
β


new z
C
A
D
B
χ
δ
ε
α
β

2. Solmu z on isäsolmun $p[z]$ ja isoisäsolmun $p[p[z]]$ *välissä* (rivi 9).

- Setä y on nyt musta, koska muuten olisimmekin tapauksessa 1.

- Seuraavan kuvan yläosa esittää tämän tapauksen.

- Tämä tapaus palautetaan seuraavaan tapaukseen 3

  johon koodikin ”valuu”.

- Palautus tehdään

  (a) nousemalla solmuun $p[z]$ (rivi 10)

  (b) kiertämällä sitä vasemmalle (rivi 11).

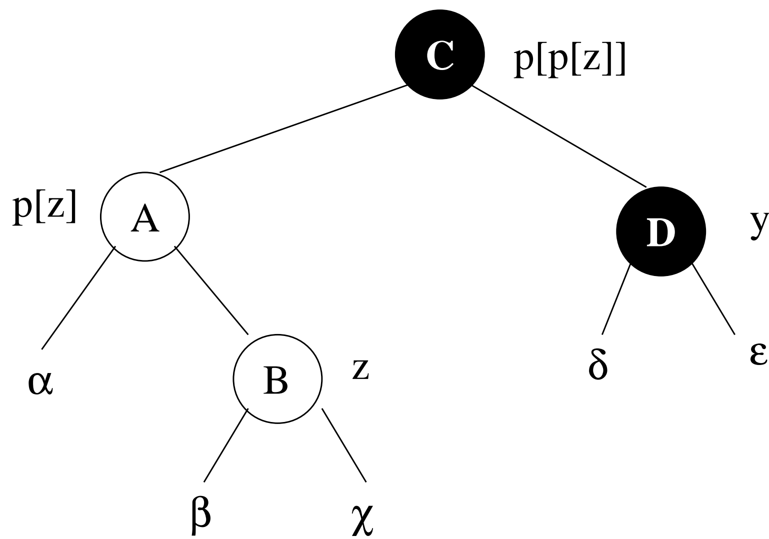
- Punamustaehto 5 säilyy

  – koska tämä on kalvojen 3.3.2 mukainen kierto

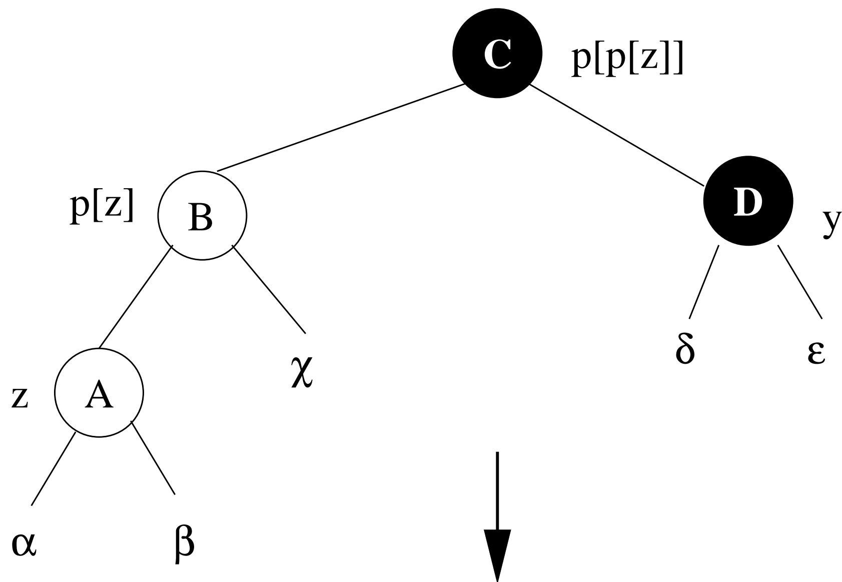
  – silloinkin kun myös $p[z]$ on punainen.

- Ehto 4 on yhä rikki

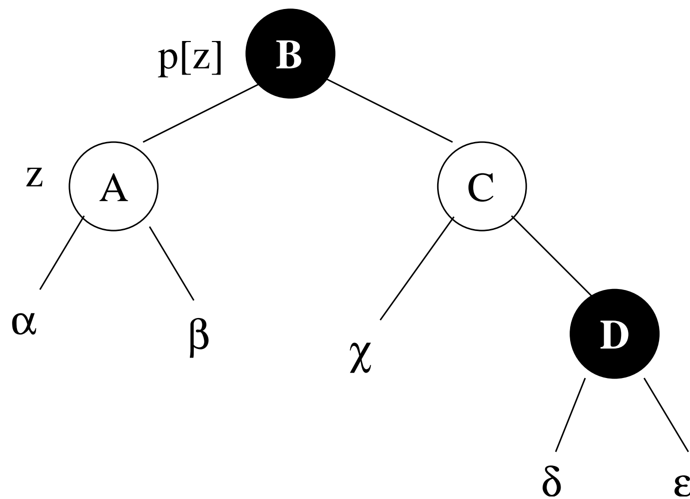
  mutta vain kaarella $p[z] \rightarrow p[p[z]]$.



Tapaus 2



Tapaus 3



3. Solmut z, $p[z]$ ja $p[p[z]]$ ovat tässä *järjestyksessä*.

- Setä y on nyt musta, koska muuten olisimmekin tapauksessa 1.

- Edellisen kuvan alaosa esittää tämän tapauksen.

- Sovelletaan isoisään $p[p[z]]$ kalvojen 3.3.2 punamustaehdon 5 säilyttävää kiertoa

  – riveillä 12–14

  – nostamaan sen punainen lapsi $p[z]$.

- Kierron seurauksena

  – koko *ehdon 4 rikkoutuminen korjautuu* kaarelta $z \to p[z]$

  – puu on jälleen kaikkien punamustaehtojen mukainen

  – tasapainotus voidaan lopettaa.

- Aliohjelman rbinsertfixup(T, z) silmukan konvergenttikin on moniosainen:

  1. Jos silmukan sisällä käsitellään tapausta 1, niin sen seurauksena viite z siirtyy aidosti lähemmäksi puun T juurta

     – rivillä 8

     – joten tämä tapaus ei voi toistua ikuisesti.

  2. Jos silmukan sisällä käsitellään tapausta 3, niin sen seurauksena

     – solmu $p[z]$ mustui rivillä 12

     – joten silmukka päättyy rivillä 2.

  3. Jos silmukan sisällä käsitellään tapausta 2, niin sen seurauksena

     – palaudutaan tapaukseen 3

     – eli silmukka päättyy silloinkin.

- Invariantista nähdään silmukan toiminta:

  Aluksi toistetaan tapausta 1 eli

  – palataan takaisin juurta kohti

  – vaihtaen solmujen värejä polulla

  – pitäen puun muoto ennallaan.

  Lopuksi päädytään

  juureen — ei tehdä enää muuta

  tapaukseen 3 — tehdään yksi kierto

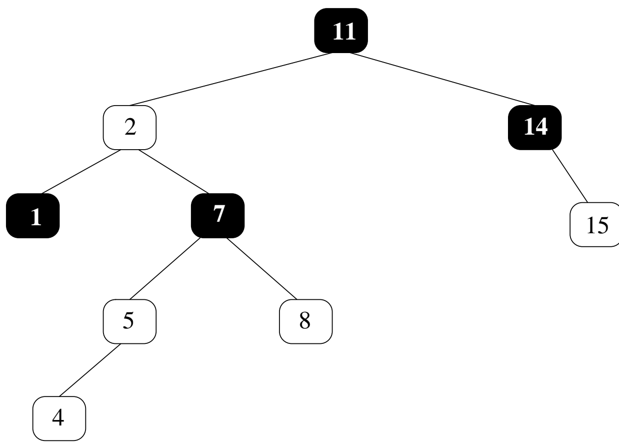
  tapaukseen 2 — tehdään kaksi kiertoa.

- Siis tasapainotus rbinsertfixup(T, z) vie

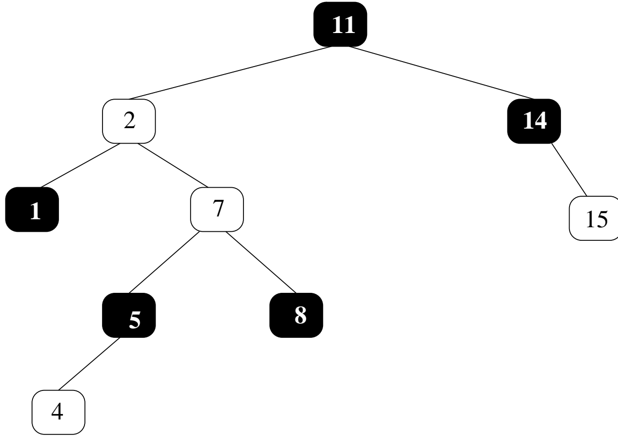
aikaa	$\mathcal{O}(\log n)$
tilaa	$\mathcal{O}(1)$

  eli saman kuin pääohjelmansa insert(T, k)

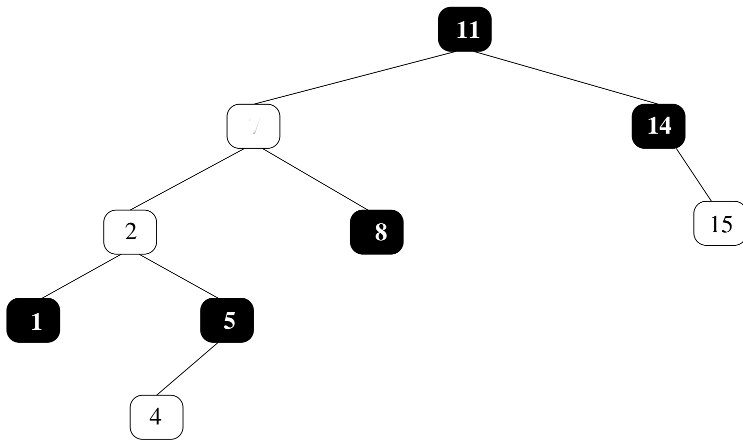
  kiertoja ≤ 2.



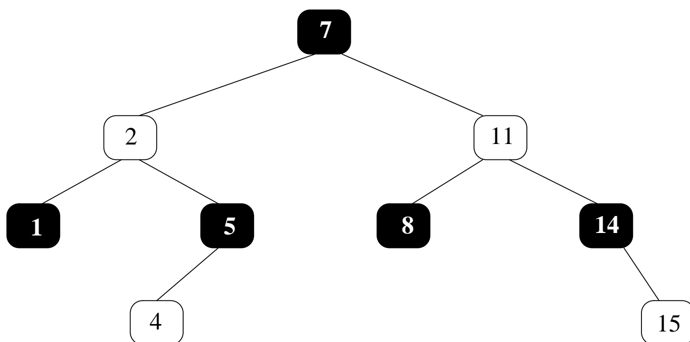
11
2
14
1
7
15
5
8
4



11
2
14
1
7
15
5
8
4



11
14
2
8
15
1
5
4



7
2
11
1
5
8
14
4
15

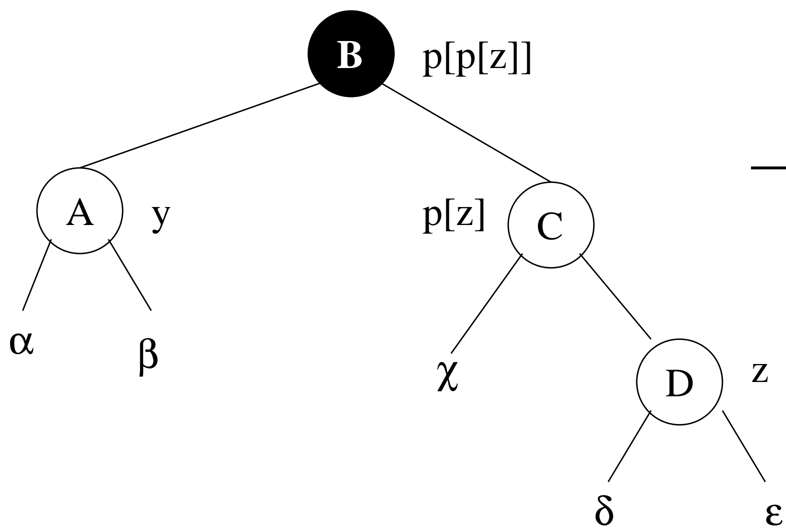
- Edellinen kuva on vielä konkreettinen esimerkki jossa solmun lisääminen puuhun aiheuttaa kaikkien korjaustapausten suorittamisen.

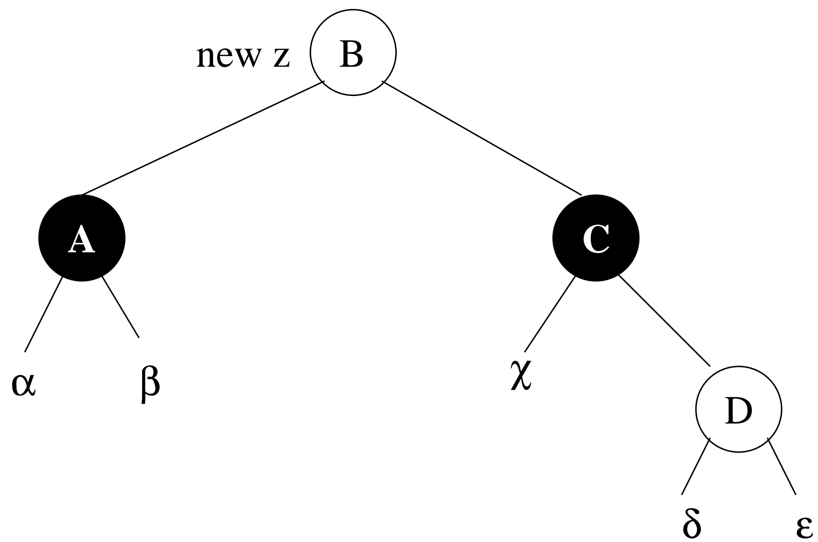
- Esitellään vielä rivit 16–27

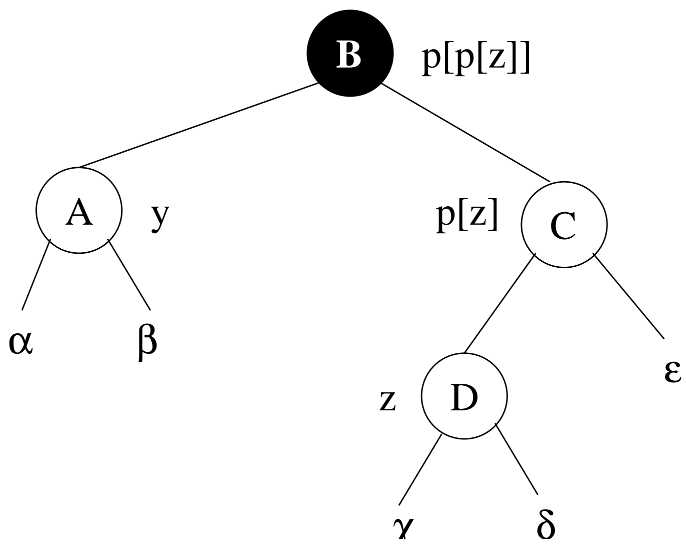
```
16          y ← left[p[p[z]]]
17          if color[y] = RED then
18              color[p[z]] ← BLACK     ▷1'
19              color[y] ← BLACK        ▷1'
20              color[p[p[z]]] ← RED    ▷1'
21              z ← p[p[z]]             ▷1'
22          else if z = left[p[z]] then
23                  z ← p[z]            ▷2'
24                  rightrotate(T, z)   ▷2'
25              color[p[z]] ← BLACK     ▷3'
26              color[p[p[z]]] ← RED    ▷3'
27              leftrotate(T, p[p[z]])  ▷3'
```

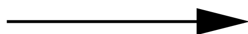
joihin on kommentoitu tapausten 1–3 vasen/oikea-symmetriset tapaukset 1'–3'.

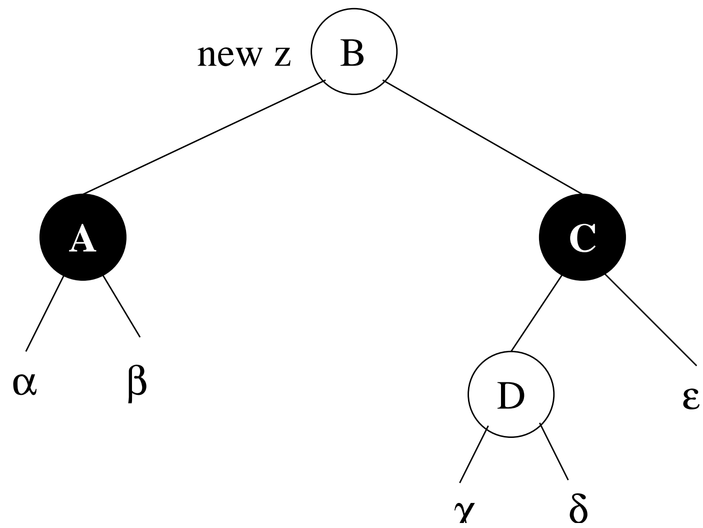
- Seuraavina ovat näistä tapauksista 1'–3' vastaavat esimerkkikuvat kuin edellä.

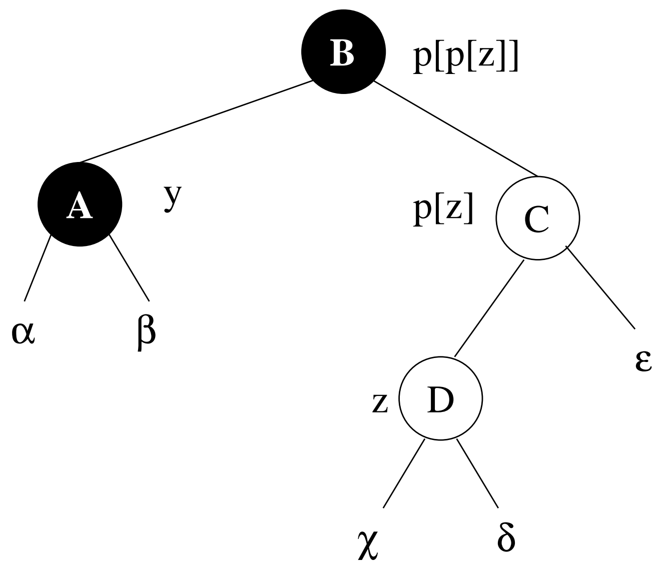

B
p[p[z]]
A
y
p[z]
C
α
β
χ
D
z
δ
ε


new z
B
A
C
α
β
χ
D
δ
ε

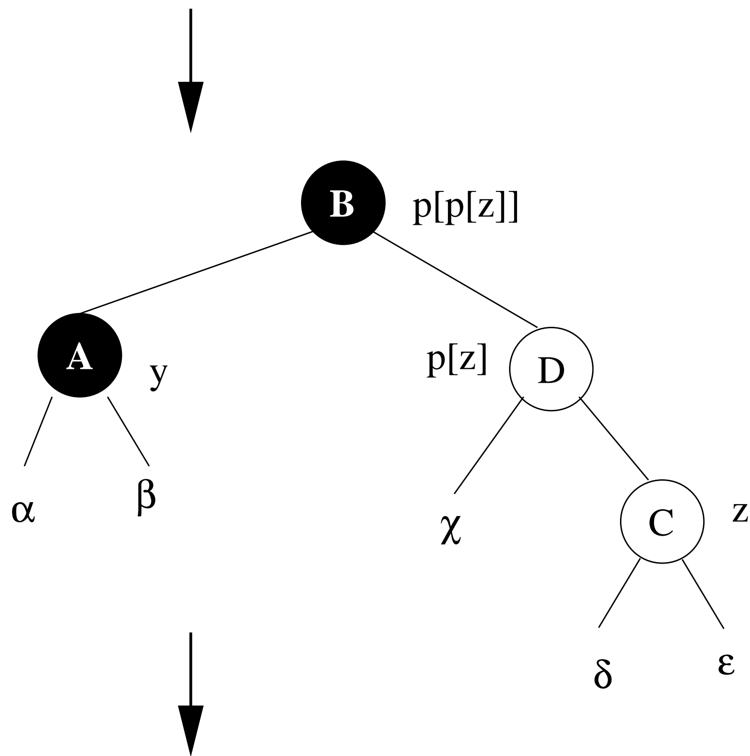

B
p[p[z]]
A
y
p[z]
C
α
β
ε
z
D
γ
δ



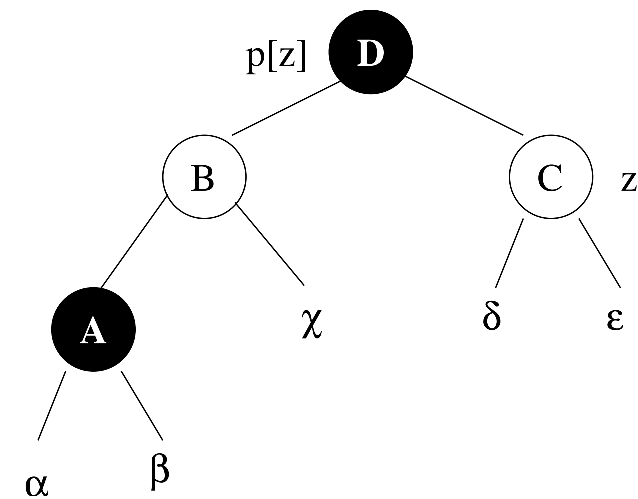

new z
B
A
C
α
β
D
ε
γ
δ



Tapaus 2’



Tapaus 3’



3.3.4 Solmun poisto

- Solmun z poisto punamustasta hakupuusta T suoritetaan kuten poisto tasapainottamattomasta hakupuusta kalvoilla 3.1.3
  - missä NIL on jälleen nil$[T]$
  - vaikka tämä pseudokoodi onkin kirjoitettu tiiviimmin:

    tapaukset "0/1 lasta ($\neq$ NIL$[T]$)" käsitellään yhdessä riveinä 1–13.

- Tasapainotukseen rbdeletefixup(T, x) ryhdytään (riveillä 12 ja 23)
  - poiston jälkeen
  - jos poistettu solmu oli **musta**

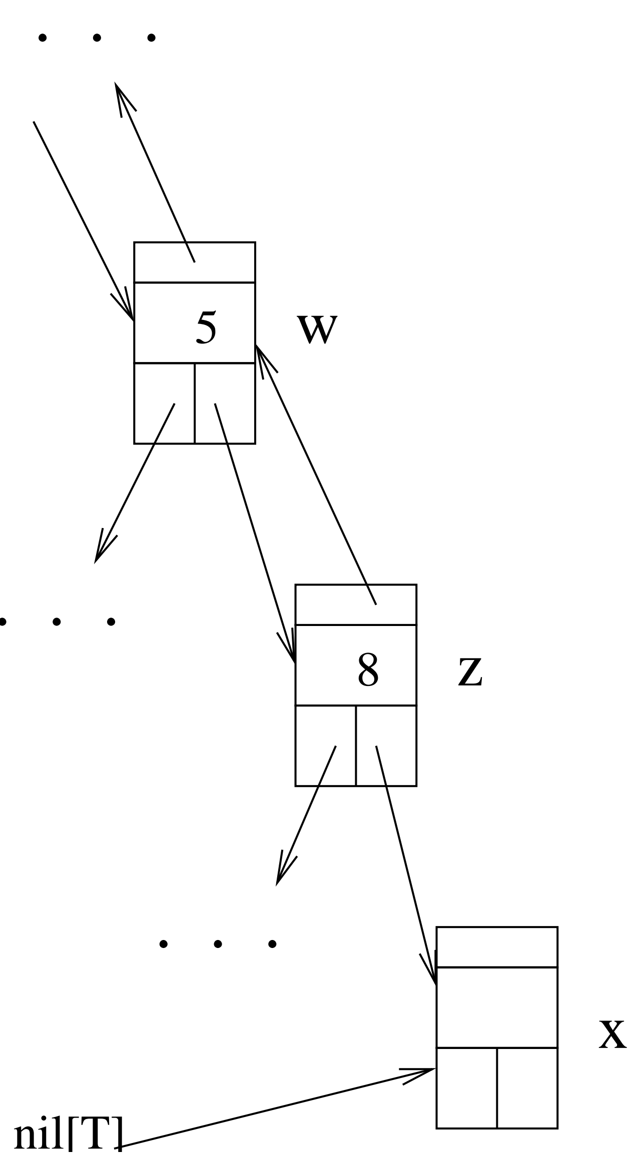
    (z rivillä 11, y rivillä 22)
  - alkaen siitä solmusta x joka otti poistetun solmun paikan.

```
delete(T, z)
1   if left[z] = nil[T] or right[z] = nil[T] then
2       if left[z] ≠ nil[T]
3           then x ← left[z]
4           else x ← right[z]
5       w ← p[z]
6       if w = nil[T] then root[T] = x
7       else if z = left[w]
8           then left[w] ← x
9           else right[w] ← x
10      p[x] ← w
11      if color[z] = BLACK
12          then rbdeletefixup(T, x)
13      return z
14  y ← succ(z)
15  x ← right[y]
16  w ← p[y]
17  if y = left[w]
18      then left[w] ← x
19      else right[w] ← x
20  p[x] ← w
21  key[z] ← key[y]
22  if color[y] = BLACK
23      then rbdeletefixup(T, x)
24  return y
```

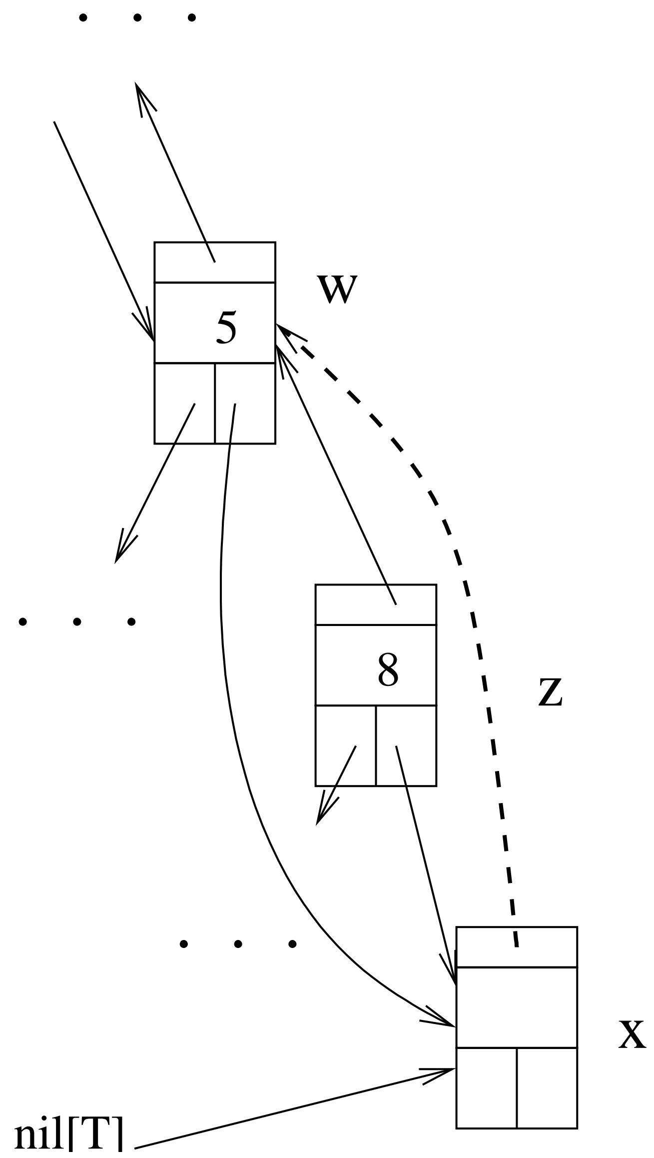
- Tätä solmua x *ei välttämättä* ole olemassa!
  - Jos esimerkiksi z on puun T ainoa solmu...
  - Silti riveillä 10 ja 20 päivitetään kentän $p[x]$ sisältöä.
  - Tämä *on* sallittua, koska kenttä

$$p[\underbrace{\mathsf{nil}[T]}_{x}]$$

    on olemassa.
  - Näin saadaan poiston tasapainotuksen erilaiset lähtötilanteet (x on/ei ole olemassa) keskenään samanlaisiksi
    * jolloin niitä ei tarvitse käsitellä tapaus tapaukselta
    * vasen/oikea-symmetrisesti
    * mikä onkin keskeinen syy erillisen solmun nil$[T]$ käyttöönotolle.



delete(z)



- Mustan solmun poistaminen puusta rikkoi punamustaehdon 5:

  polku poistokohdan kautta lehteen nil$[T]$ sisältää 1 vähemmän mustia solmuja kuin puun T muut polut.

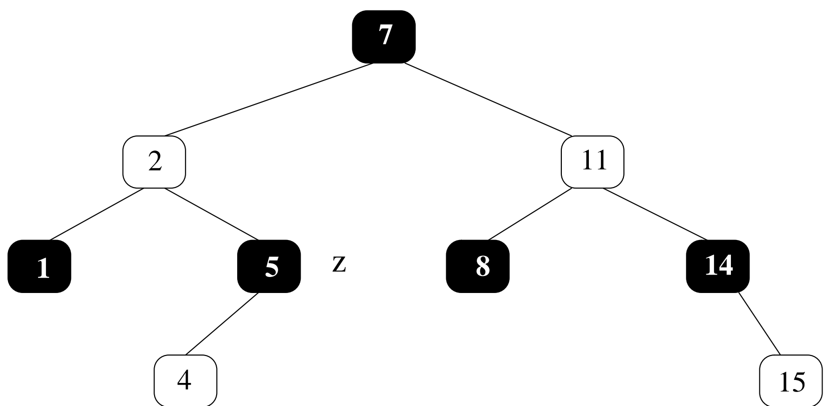
- Tasapainotuksen invariantti on ehdollinen:

  Jos solmussa x olisikin vielä *yksi **musta** enemmän* kuin siinä oikeasti on

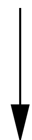
  niin ehto 5 olisi voimassa.

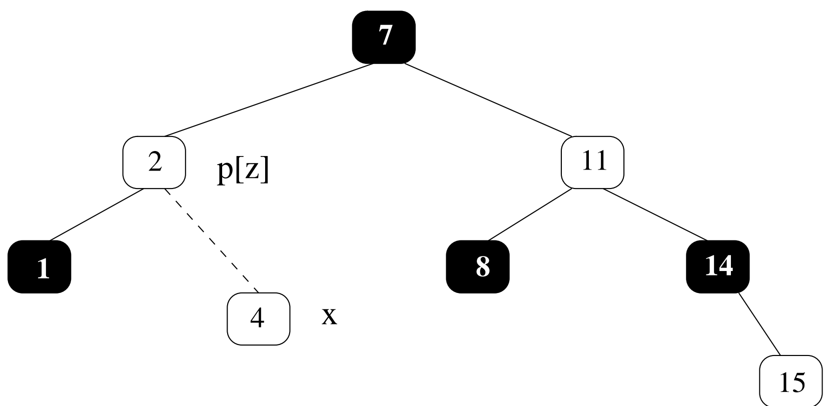
- Toisin sanoen viite x kantaa mukanaan *yhtä ylimääräistä mustaa*

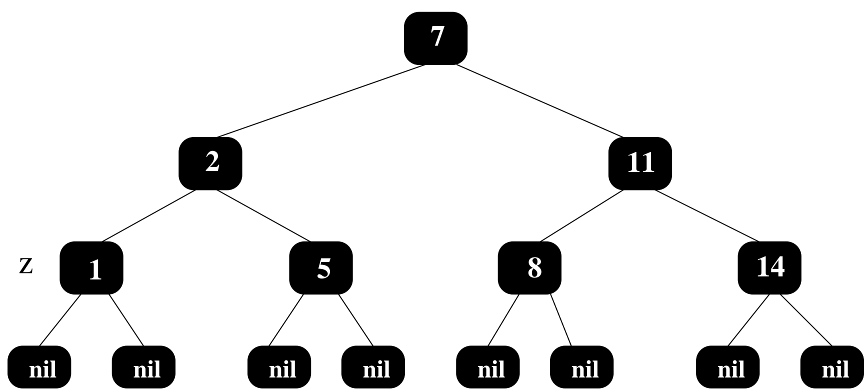
  jolle tasapainotus etsii sopivaa sijoituspaikkaa.



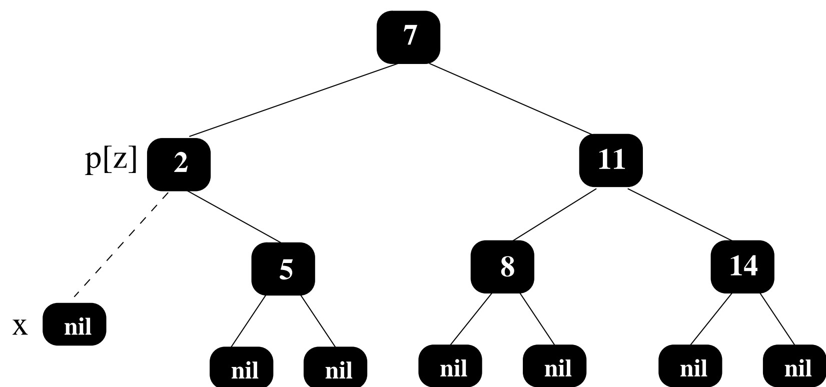
delete(z)







delete(z)



```
rbdeletefixup(T, x)
1      while x ≠ root[T] and color[x] = BLACK do
2          if x = left[p[x]] then
3              w ← right[p[x]]
4              if color[w] = RED then
5                  color[w] ← BLACK                ▷1
6                  color[p[x]] ← RED               ▷1
7                  leftrotate(T, p[x])             ▷1
8                  w ← right[p[x]]                 ▷1
9              if color[left[w]] = BLACK and
                       color[right[w]] = BLACK then
10                 color[w] ← RED                  ▷2
11                 x ← p[x]                        ▷2
12             else if color[right[w]] = BLACK then
13                     color[left[w]] ← BLACK      ▷3
14                     color[w] ← RED              ▷3
15                     rightrotate(T, w)           ▷3
16                     w ← right[p[x]]             ▷3
17                 color[w] ← color[p[x]]          ▷4
18                 color[p[x]] ← BLACK             ▷4
19                 color[right[w]] ← BLACK         ▷4
20                 leftrotate(T, p[x])             ▷4
21                 x ← root[T]                     ▷4
22-41      else ▷rivit 3–21 vasen/oikea-symm.
42     color[x] ← BLACK
```

- Tasapainotussilmukka päättyy (rivillä 1) jos x viittaa

  punaiseen solmuun koska se voidaan värjätä mustaksi

  (rivillä 42)

  juureen root$[T]$ koska kaikki polut kulkevat juuren kautta.

- Silmukka jakautuu kahteen vasen/oikea-symmetriseen haaraan (rivi 2) sen mukaan, onko x

  vasen (rivit 3–21 joka esitellään nyt)

  oikea (rivit 23–41 jotka esitetään myöhemmin)

  lapsi.

- Rivillä 3 viite w asetetaan osoittamaan solmun x *veljeen*.

- Tämä veli $w \neq$ NIL$[T]$:
  - alipuusta x poistettu sisäsolmu oli musta
  - joten myös alipuussa w täytyy olla jokin musta sisäsolmu
  - koska yhteinen isäsolmu $p[x] = p[w]$ toteuttaa punamustaehdon 5

    kun otetaan huomioon solmun x ylimääräinen musta.

- Vasen haara jakautuu edelleen neljään tapaukseen veljen w perhetilanteen mukaan.

1. Veli w on *punainen* (rivit 4–8):

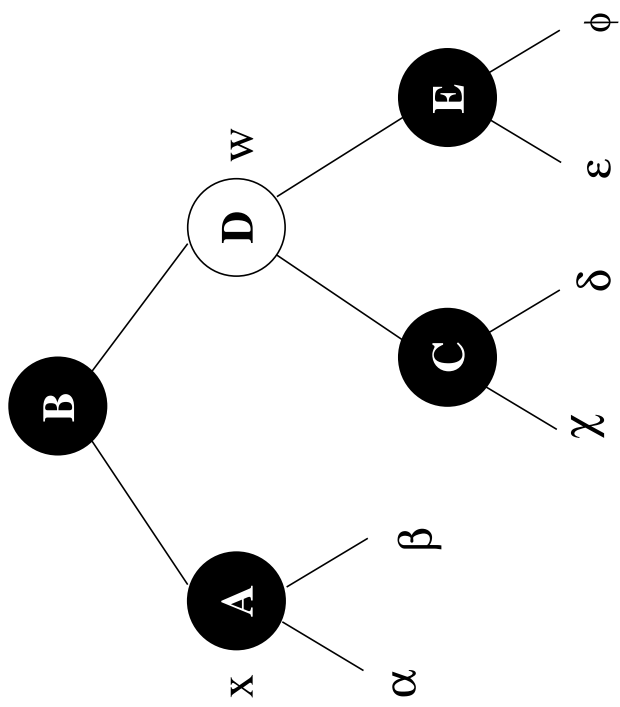
- Silloin
  - isäsolmu $p[x]$ ja
  - veljen w lapsisolmut

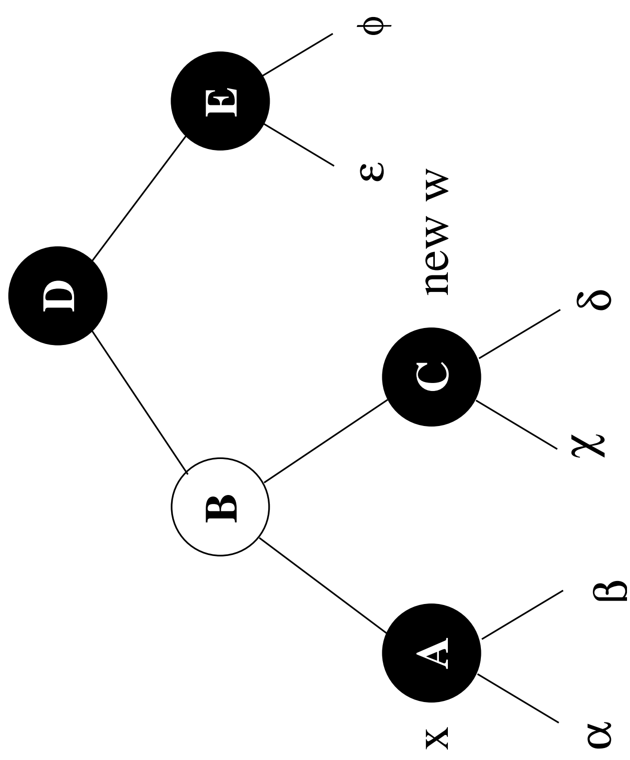
  ovat mustia punamustaehdon 4 nojalla.

- Tavoite:
  - Saada solmulle x jokin musta veli...
  - eli palauttaa tämä tapaus muihin tapauksiin 2–4...
  - joihin koodi "valuu".

- Solmu left$[w]$
  - on musta ehdon 4 nojalla
  - saadaan solmun x uudeksi veljeksi kiertämällä isää $p[x]$ vasemmalle kalvojen 3.3.2 tapaan punamustaehto 5 säilyttäen (rivit 5–7)
  - valitaan solmun x uudeksi veljeksi.

- Tämä veljen vaihto säilyttää myös punamustaehdon 4:
  - Punastuva isäsolmu $p[x]$ päätyy mustien solmujen ympäröimäksi
  - seuraavan kuvan mukaisesti.
- Kierron jälkeen siirretään w osoittamaan tähän uuteen veljeen
  - joka löytyykin kierron seurauksena viitteellä right[$p[x]$] (rivi 8)
  - jotta myös w olisi oikein, kun ''valutaan'' tapauksiin 2–4.


B
D
w
A
x
C
E
α
β
χ
δ
ε
φ


D
E
B
C
A
x
new w
α
β
χ
δ
ε
φ

2. Mustan veljen w *molemmat lapset ovat myös mustia* (rivit 9–11):

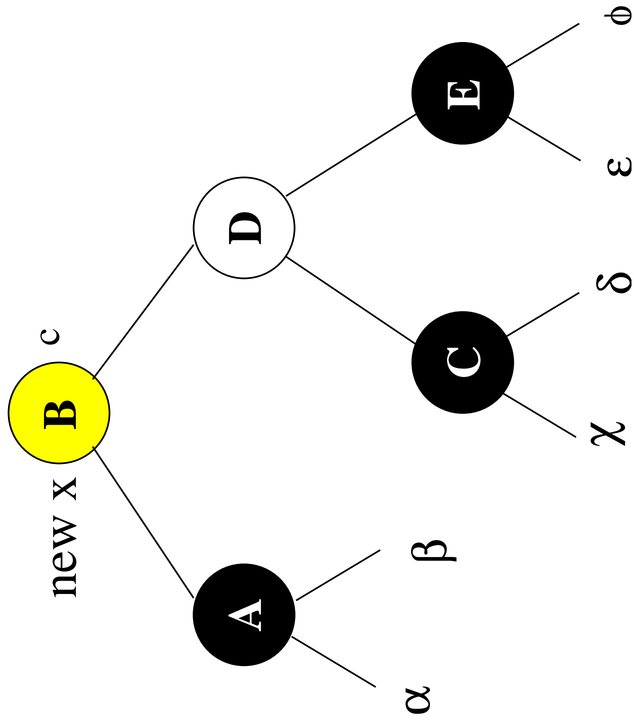
- Isä $p[x]$ voi olla kumpaa väriä c tahansa.
- Tavoite: nostaa viite x nykyisestä solmusta sen isään $p[x]$ (rivi 11).
- Punamustaehto 5 säilyy
  - jos veli w punastuu (rivi 10)
  - koska isään $p[x]$ tulee

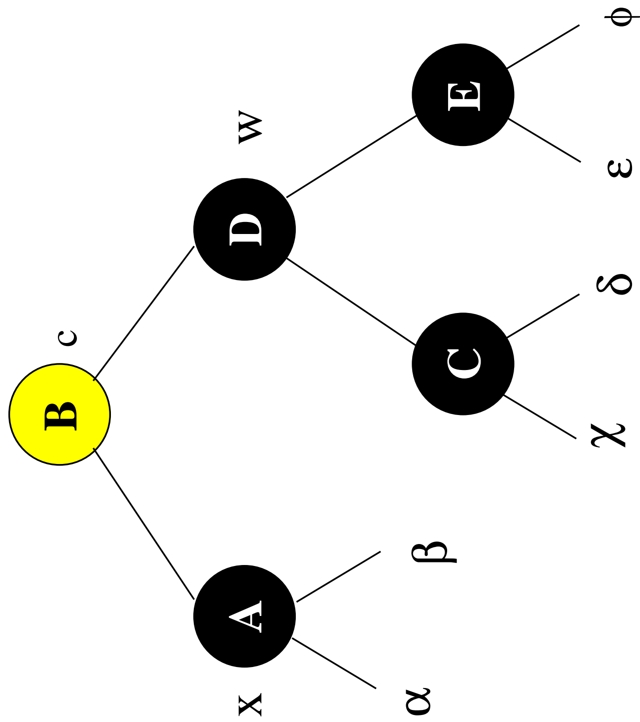
    vasemmalta nouseva viite x *ja sen yksi ylimääräinen musta*

    oikealta se musta joka oli veljessä w.
  - Veljen w punastuminen ei riko punamustaehtoa 4, koska sen

    lapset ovat mustia

    isään viittaa x eli sekin on vähintään ylimääräisesti musta.
- Sitten aloitetaan seuraava silmukkakierros.


new x
c
B
A
D
C
E
α
β
χ
δ
ε
φ

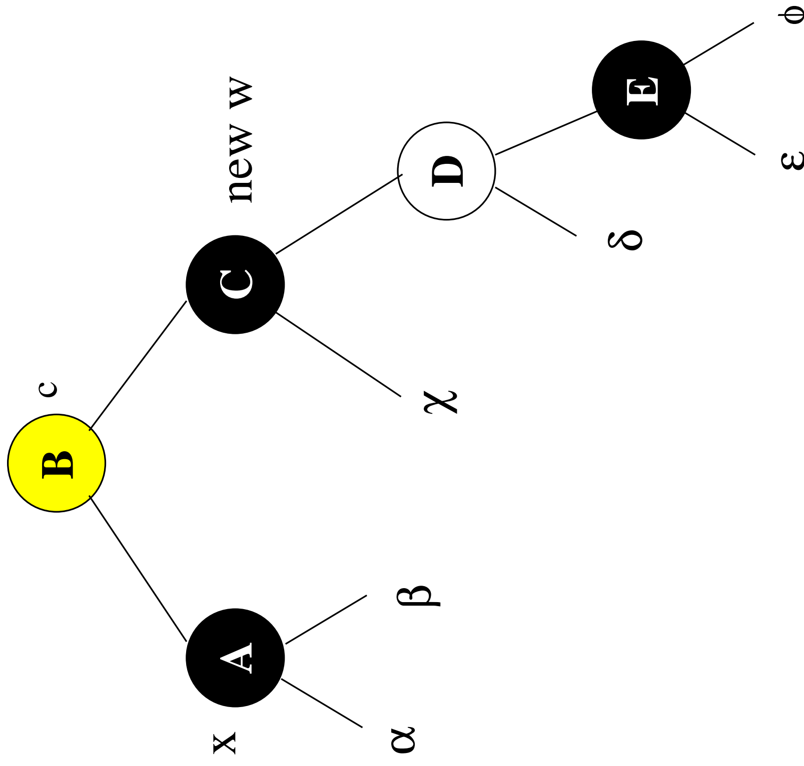

x
c
w
B
A
D
C
E
α
β
χ
δ
ε
φ

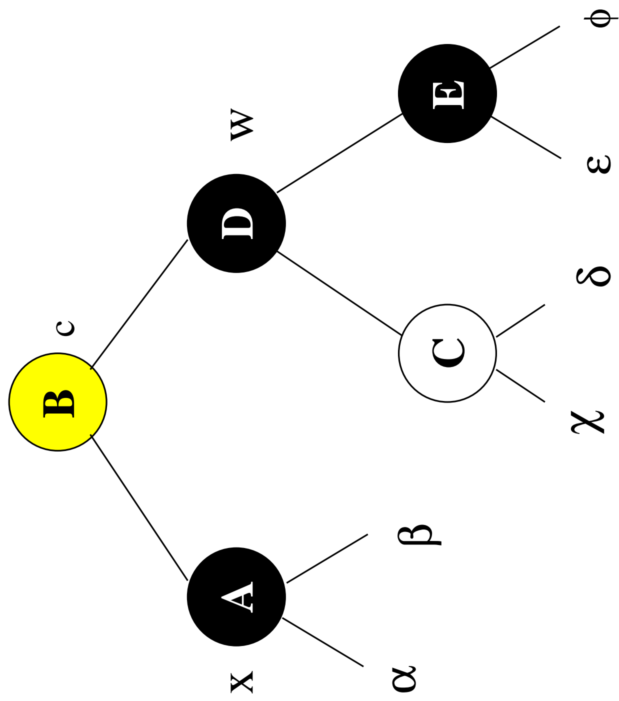
3. Mustan veljen w lapsista (rivit 12–16)

vasen eli solmua x läheisempi on *punainen*

oikea eli kaukaisempi onkin *musta:*

- Isä $p[x]$ voi olla kumpaa väriä c tahansa.
- Tavoite: saada veljen punainen lapsi kauemmas
  - eli palauttaa tämä tapaukseen 4
  - johon koodi ”valuu”.
- Kierretään veljeä w oikealle kalvojen 3.3.2 tapaan punamustaehto 5 säilyttäen (rivit 13–15).
- Kierto vaihtaa myös solmulle x veljen
  - entiseksi punaiseksi veljenpojaksi
  - joka mustui kierrossa
  - joten päivitetään viite w (rivi 16).
- Punamustaehdon 4 säilyminen näkyy seuraavasta esimerkkikuvasta.

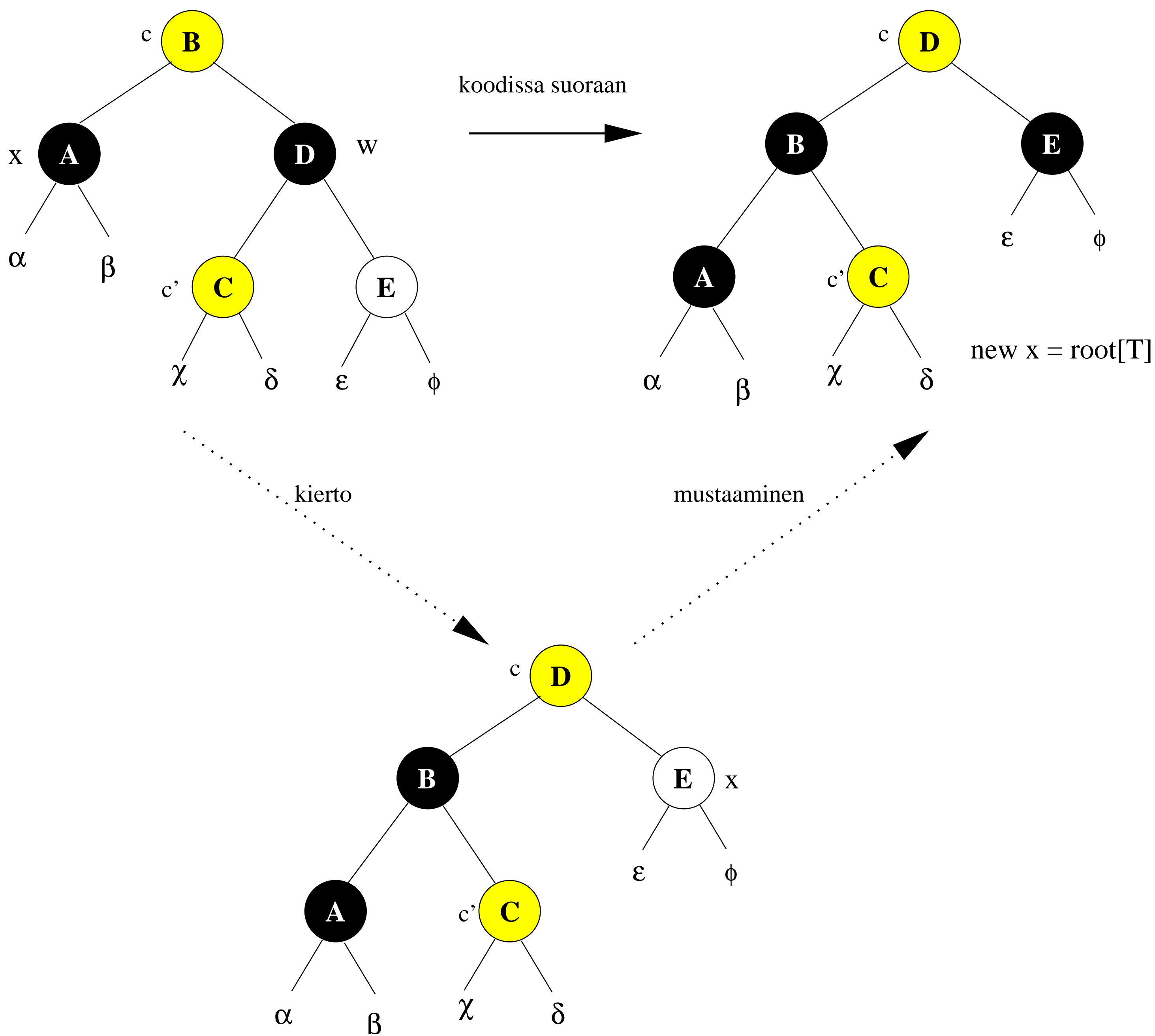

B
c
C
new w
D
E
A
x
α
β
χ
δ
ε
φ


B
c
D
w
E
C
A
x
α
β
χ
δ
ε
φ

4. Mustan veljen w *oikea lapsi on punainen* (rivit 17–21):

- Isä $p[x]$ voi olla kumpaa väriä c tahansa.
- Veljen w toinen lapsi left$[w]$ saa olla kumpaa väriä c' tahansa.
- Tavoite:
  - *päästä kokonaan eroon* viitteen x kuljettamasta ylimääräisestä mustasta
  - eli saada haluttu lopputulos:
    - alkuperäiset punamustaehdot täyttävä puu
    - ilman että ehdossa 5 tarvitsisi enää laskea "+1" viitteen x kohdalla
  - eli lopettaa koko päivityssilmukka (rivi 21).

- Tavoite voidaan saavuttaa 2 vaiheessa:
  - Ensin kierretään isää $p[x]$ vasemmalle kalvojen 3.3.2 tapaan *vaikka nouseva lapsi w onkin musta*.
  - Kierron seuraukset näkyvät seuraavasta kuvasta:
    * Ongelmasolmun x isä $p[x]$ mustui, eli *alkuperäinen ylimääräinen musta poistui*.
    * Alipuu left$[w]$ siirtyi yhden mustan solmun alta toisen alle, eli sen musta syvyys säilyi samana.
    * Alipuu right$[w]$ menetti yläpuoleltaan mustan solmun, joten siihen pitäisi laittaa *uusi ylimääräinen musta*.
  - Uudesta ylimääräisestä mustasta on kuitenkin helppoa päästä eroon:
    * Se on punaisessa solmussa, joka voidaan siis värjätä itse mustaksi.
    * Värjäys varmistaa samalla punamustaehdon 4.


c
B
x
A
D
w
α
β
c'
C
E
χ
δ
ε
ϕ
koodissa suoraan
c
D
B
E
ε
ϕ
A
c'
C
α
β
χ
δ
new x = root[T]
kierto
mustaaminen
c
D
B
E
x
ε
ϕ
A
c'
C
α
β
χ
δ

- Rivit 17–21 toteuttavat suoraan
  - näiden vaiheiden lopputuloksen
  - silmukan pysäyttämisen ”voimakeinoin”

  kuten kirjassa.

- Vaiheittain eteneminen voitaisiin toteuttaa
  - poistamalla rivi 19
  - muuttamalla riviksi

    21 $x \leftarrow$ right$[w]$ ▷4

  - jatkamalla silmukkaa normaalisti rivin 1 kautta riville 42.

- Tasapainotussilmukan pysähtyminen:
  - Tapaus 4 pysähtyy varmasti (rivi 21).
  - Tapaus 3 pysähtyy, koska se palautuu tapaukseen 4.
  - Tapauksessa 2 on kaksi vaihtoehtoa sen isäsolmun $p[x]$ värin mukaan, johon viite x nousee:

    Punainen (tai juuri): silmukka päättyy rivillä 1.

    Musta (eikä juuri): silmukka jatkuu, mutta viite x on *noussut* nyt lähemmäksi juurta root$[T]$.
  - Tapaus 1 palautuu

    joko pysähtyviin tapauksiin 4 tai 3

    tai silmukkatapaukseen 2:
    * Isäsolmu $p[x]$ on nyt punainen.
    * Siis silmukka päättyy, vaikka viite x onkin *laskeutunut* puussa alaspäin.

- Tasapainotusalgoritmin pahin tapaus on siis

$$\underbrace{2, 2, 2, 2, 2, 2, \ldots, 2}_{\text{puun korkeus } -1 \text{ kpl.}}, 1, 3, 4.$$

- Siitä nähdään tutut resurssitarpeet

$$\begin{aligned} \mathcal{O}(\log n) & \quad \text{silmukkakierrosta, missä} \\ n = & \text{ sisäsolmujen lukumäärä} \\ \mathcal{O}(1) & \quad \text{muistipaikkaa} \\ \leq 3 & \quad \text{kiertoa} \end{aligned}$$

  eli ei hidasta oleellisesti poistoa.

- Esimerkkikuvia:

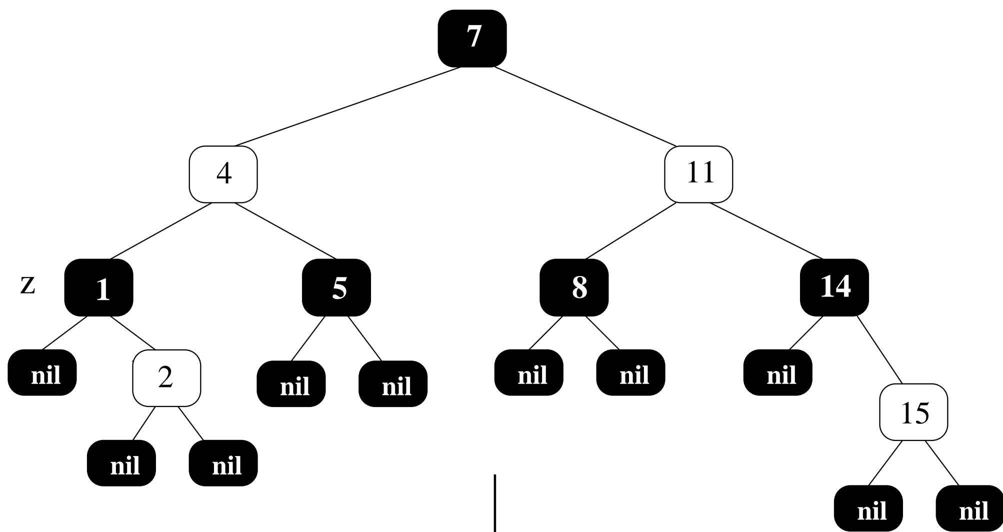
  1. Yhtään silmukkakierrosta ei tarvita, koska poistuvan solmun z paikan ottaa punainen solmu x.

  2. Silmukassa toistetaan tapauksen 2 mustaa vaihtoehtoa, kunnes viite x osoittaa juureen root[T].

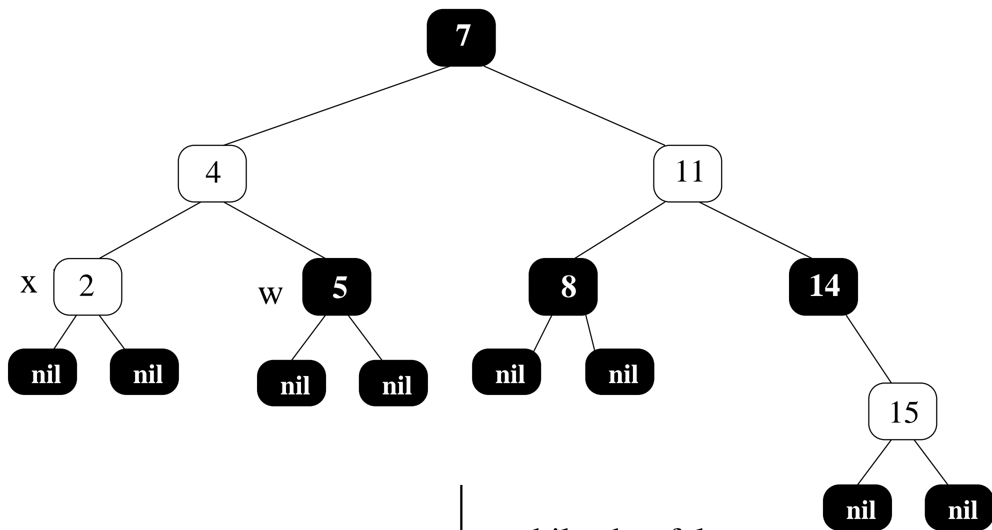
  3. Poisto aiheuttaa pahimman tapauksen.

     – Tapaus 2 esiintyy aluksi 1 kerran.

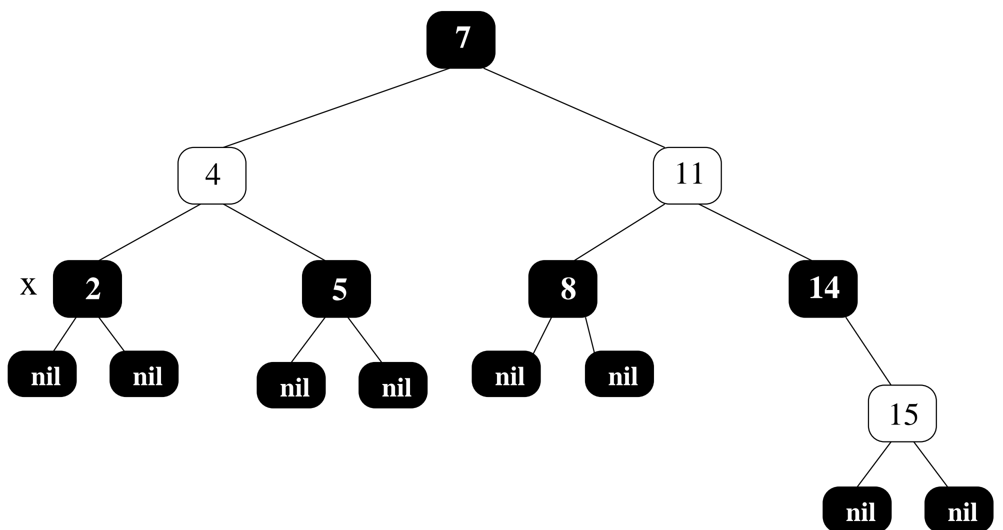
     – 2 kuvaa.

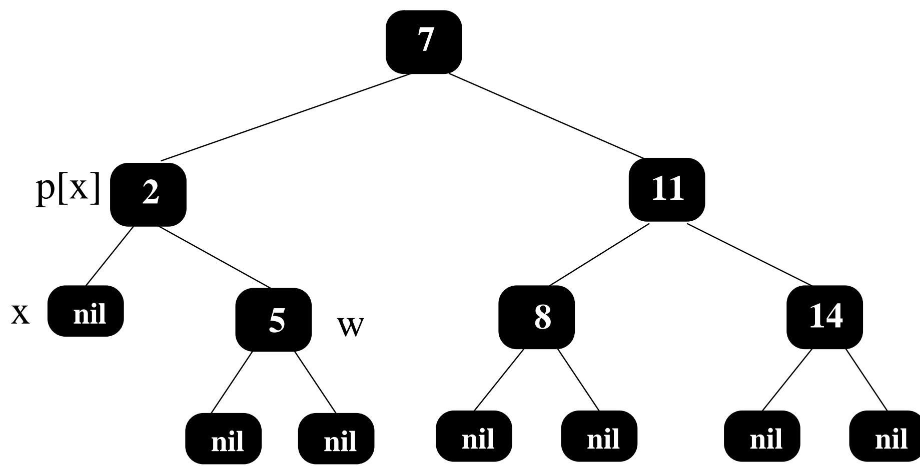


delete(z)

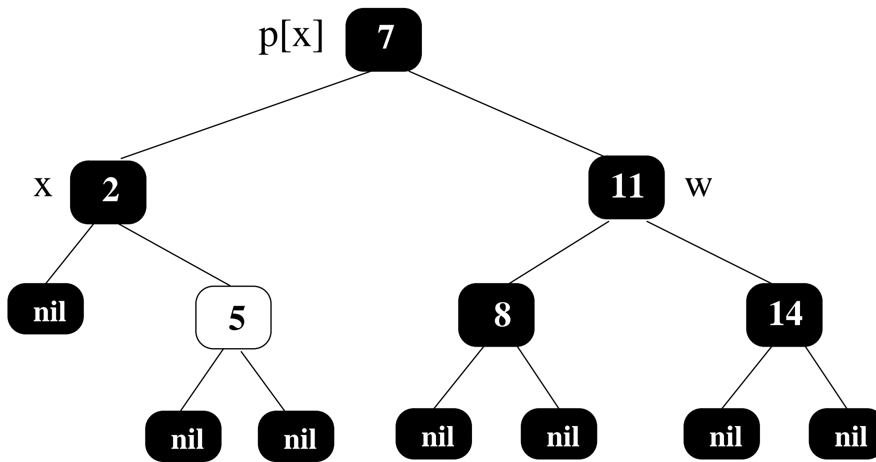


while ehto false

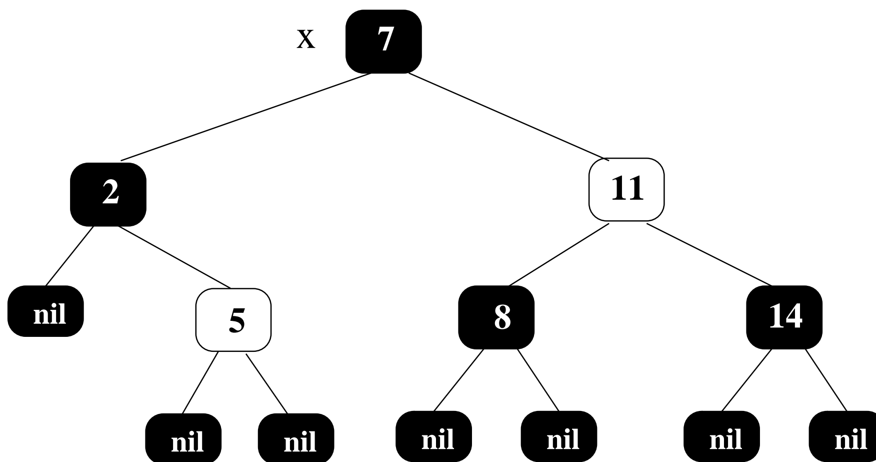


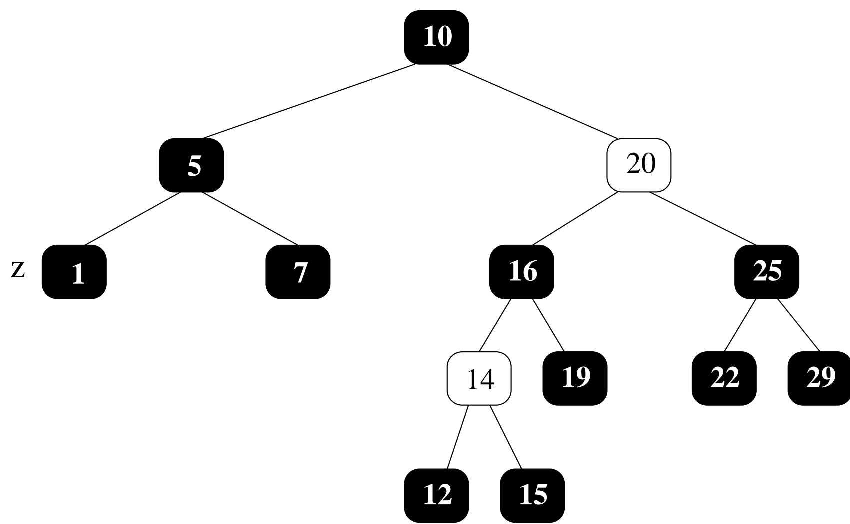


tapaus 2

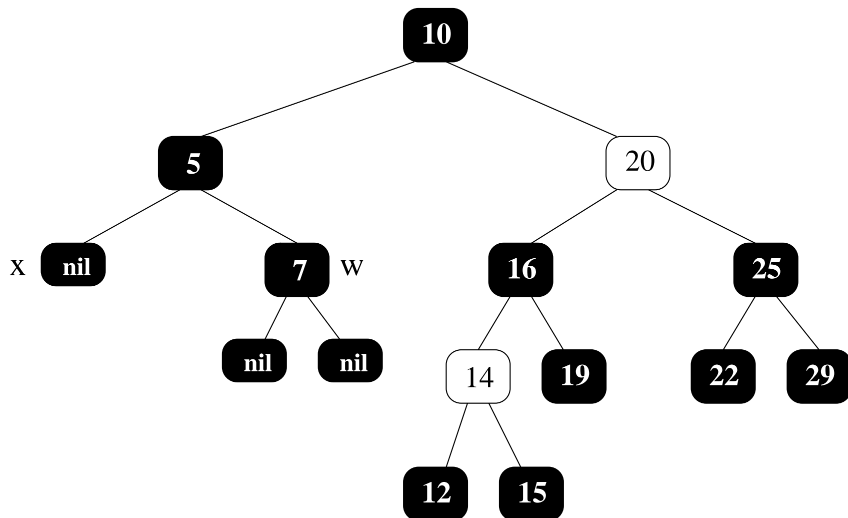


tapaus 2

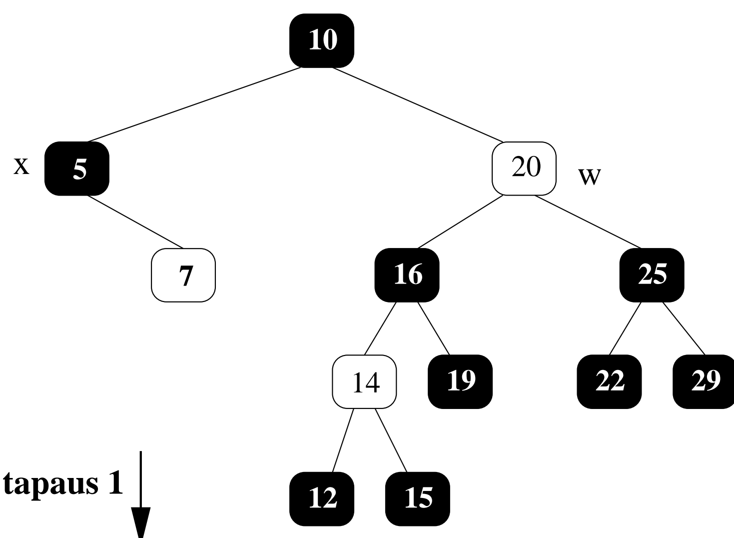


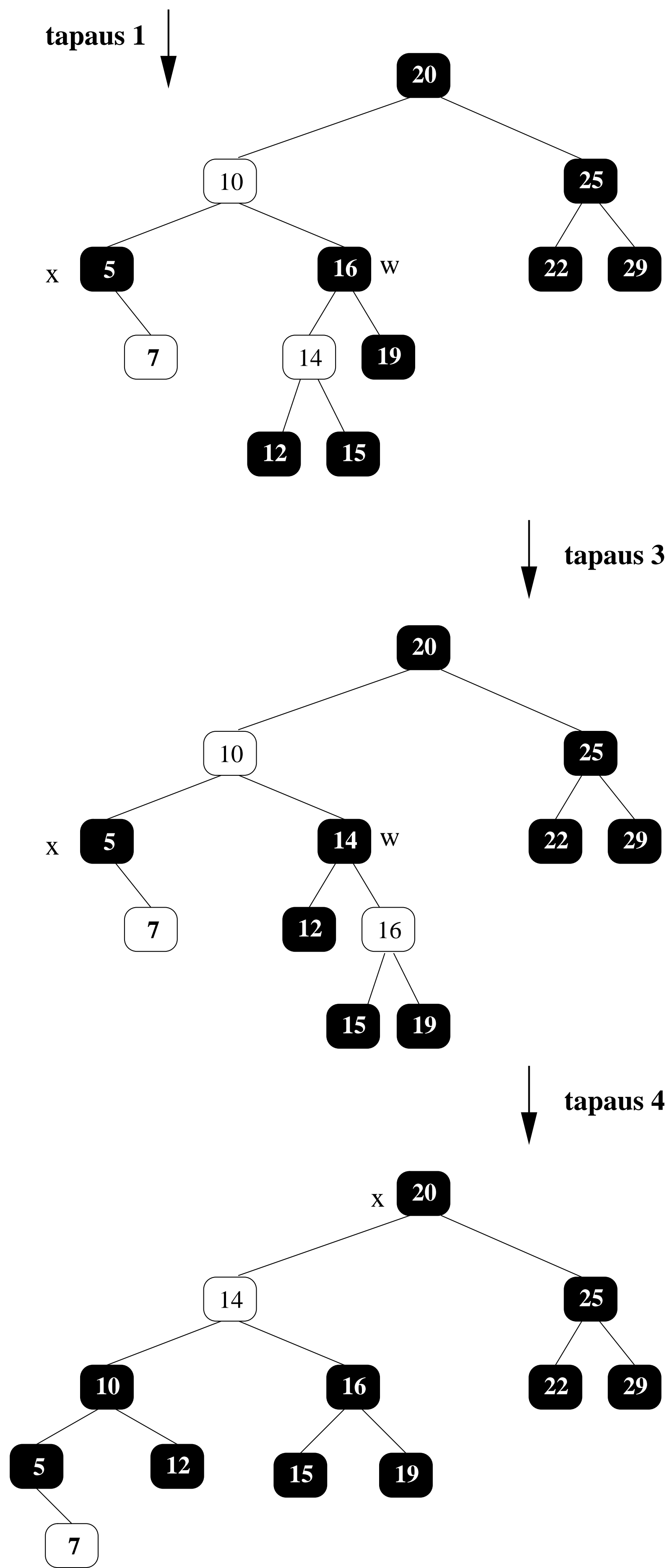


delete(z)



tapaus 2




tapaus 1
20
10
25
x
5
16
w
22
29
7
14
19
12
15
tapaus 3
20
10
25
x
5
14
w
22
29
7
12
16
15
19
tapaus 4
x
20
14
25
10
16
22
29
5
12
15
19
7

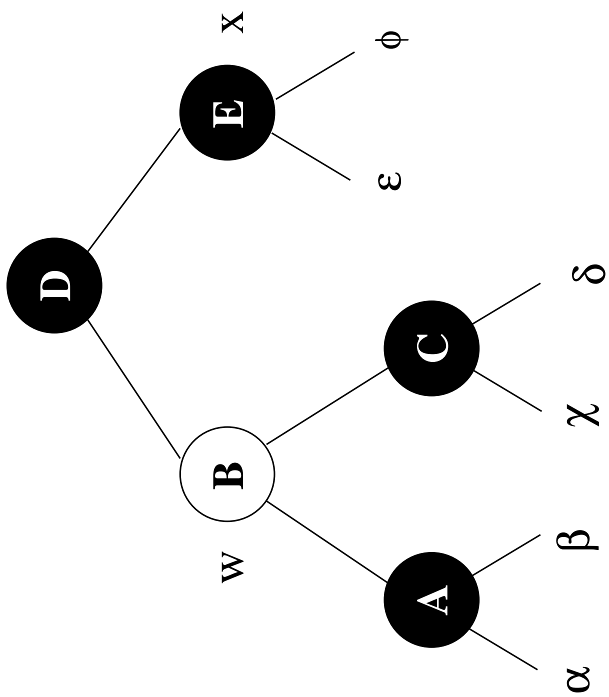
- Esitetään vielä täydellisyyden vuoksi vasen/oikea-symmetrisen **else**-haaran koodi:

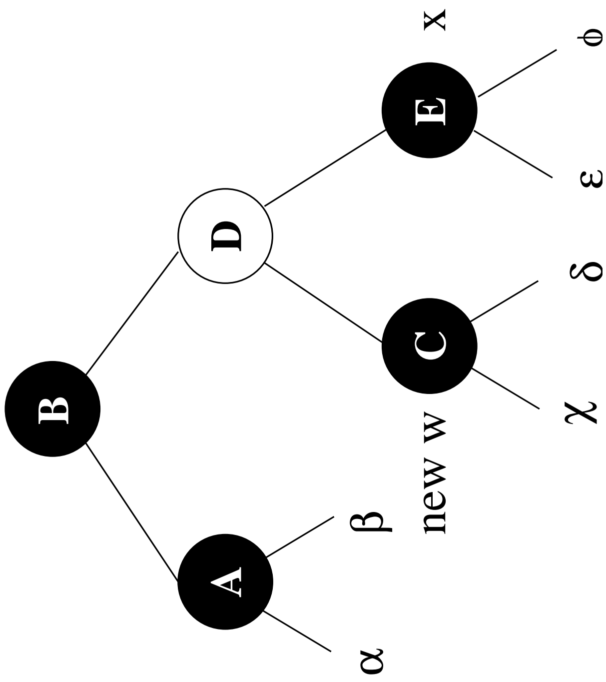
```
23 w ← left[p[x]]
24 if color[w] = RED then
25     color[w] ← BLACK                ▷1'
26     color[p[x]] ← RED               ▷1'
27     rightrotate(T, p[x])            ▷1'
28     w ← left[p[x]]                  ▷1'
29 if color[left[w]] = BLACK and
          color[right[w]] = BLACK then
30     color[w] ← RED                  ▷2'
31     x ← p[x]                        ▷2'
32 else if color[left[w]] = BLACK then
33         color[right[w]] ← BLACK     ▷3'
34         color[w] ← RED              ▷3'
35         leftrotate(T, w)            ▷3'
36         w ← left[p[x]]              ▷3'
37     color[w] ← color[p[x]]          ▷4'
38     color[p[x]] ← BLACK             ▷4'
39     color[left[w]] ← BLACK          ▷4'
40     rightrotate(T, p[x])            ▷4'
41     x ← root[T]                     ▷4'
```

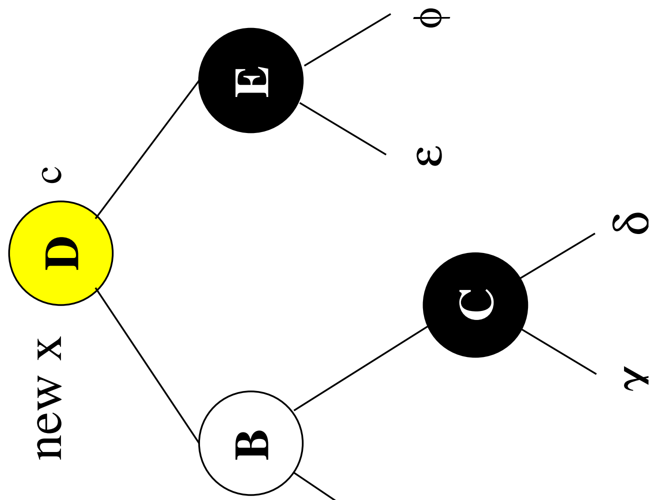
- Kommentein on osoitettu tapausten 1–4 vasen/oikea-symmetriset muodot 1'–4'.

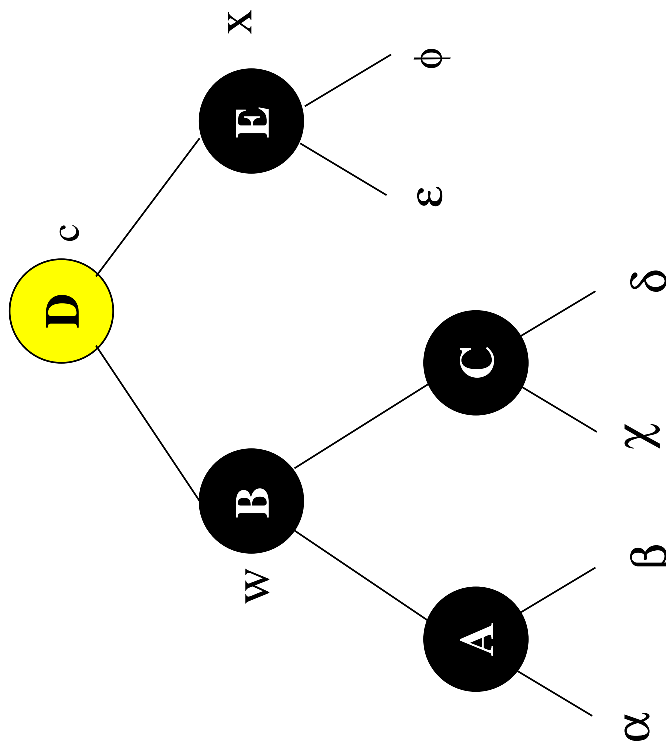
- Jokaisesta muodosta 1'–4' esitetään vielä sitä vastaava esimerkkikuva.

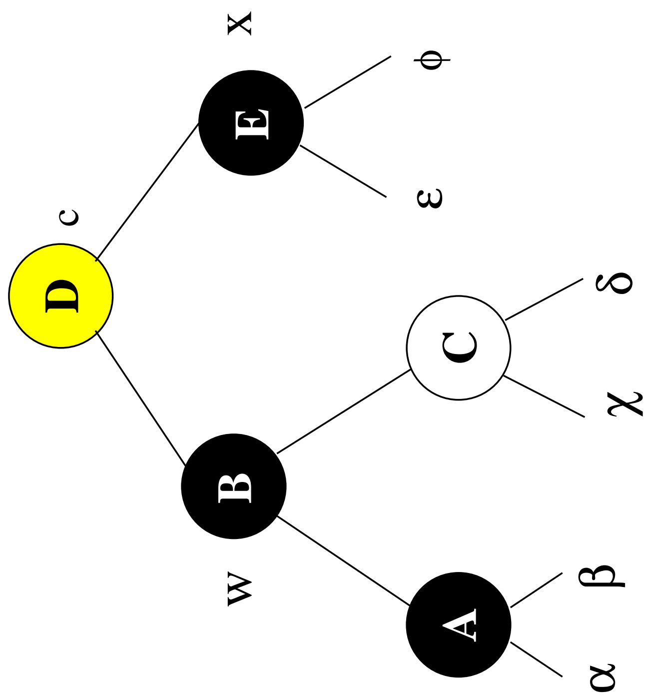
- Muodon 4' esimerkkikuvassa on kuitenkin vain lopputulos, ei välivaihetta.


D
B
E
A
C
w
x
α
β
χ
δ
ε
φ

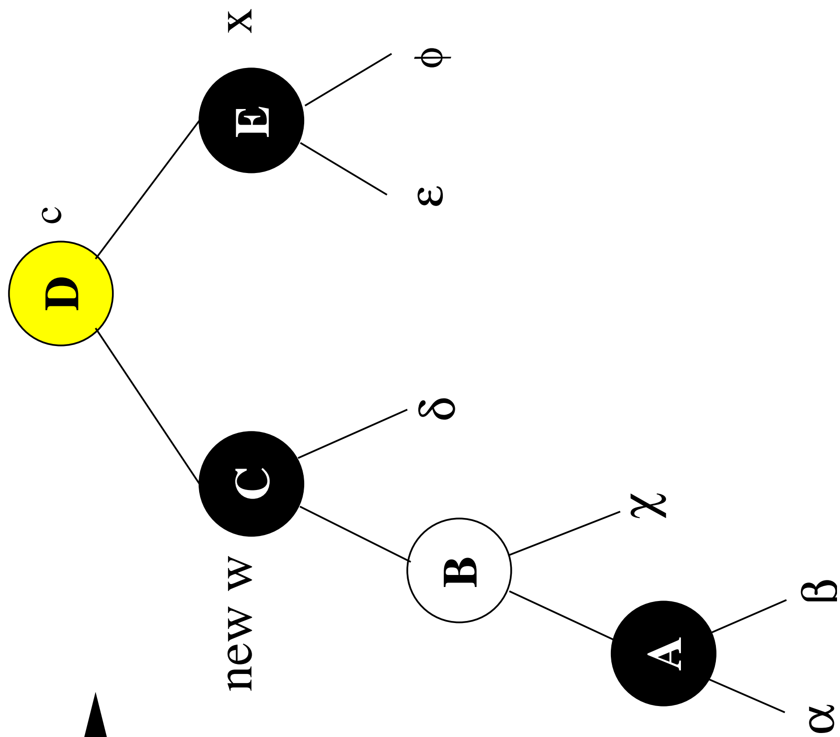

B
A
D
C
E
new w
x
α
β
χ
δ
ε
φ


new x
c
D
B
E
A
C
α
β
γ
δ
ε
φ

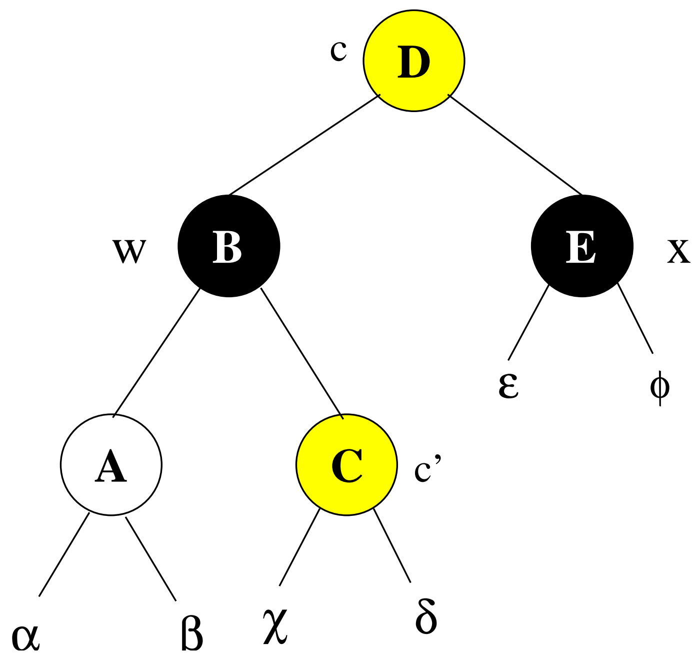

c
x
w
D
B
E
A
C
α
β
χ
δ
ε
φ

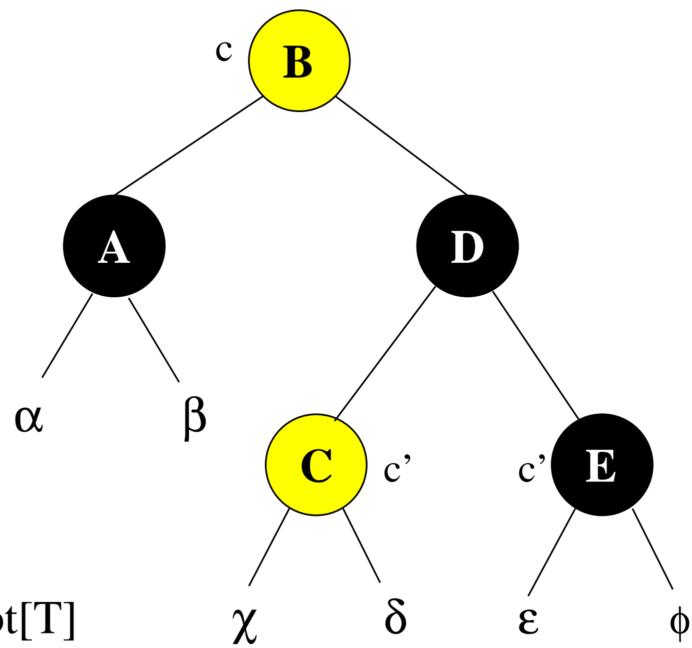


D
c
E
x
B
w
C
A
α
β
χ
δ
ε
φ



D
c
E
x
C
new w
B
A
α
β
χ
δ
ε
φ





new x = root[T]

3.4 Monta indeksiä samaan dataan

- Edellä käsiteltiin yksinkertaist(ettu)a tilannetta, jossa puuhun talletettiin vain avaimen arvo.

- Usein talletettava tietoalkio sisältää enemmän:
  - useita *erilaisia* avaimia
  - lisätietokenttiä.

- Esimerkiksi puhelinnumerotiedustelusta kysytään
  - nimeen liittyvää puhelinnumeroa — nimet ovat avaimia
  - numeroon liittyvää nimeä — numerotkin ovat avaimia
  - näillä avaimilla löydettyä osoitetta — lisätietona.

- Luonteva ratkaisu on silloin *erottaa* toisistaan

tietueet jotka esittävät kaikki yhteen entiteettiin liittyvät tietokentät

puhelinluettelossa yhden henkilön tietue sisältää kentät

- nimi
- puhelinnumero
- osoite
- ...

indeksit jotka ovat (esimerkiksi) hakupuita avainkenttien suhteen:

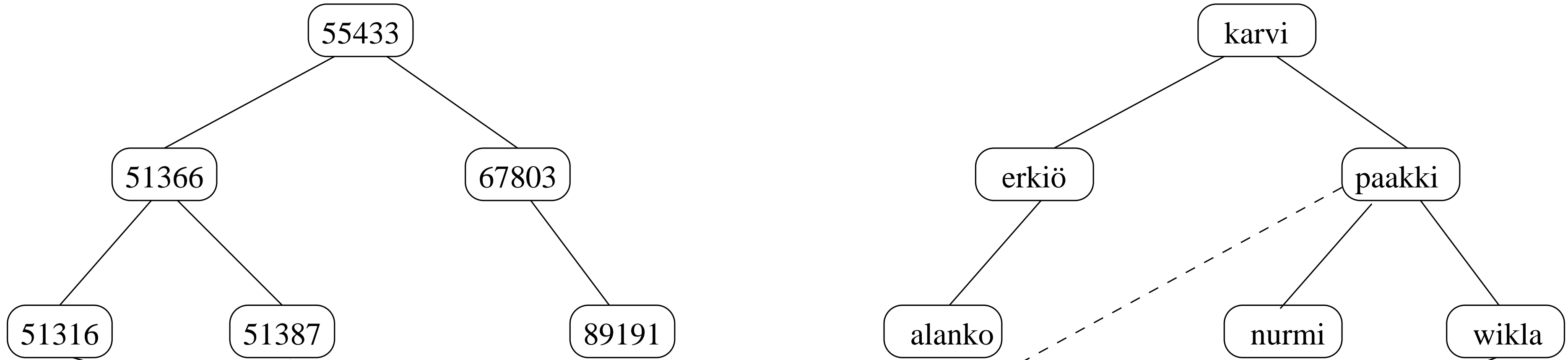
jokaisessa solmussa on *loogisesti*

- vastaava avaimen arvo
- viite vastaavaan tietueeseen.

Fyysisesti riittää pelkkä viite: avaimen arvo voidaan lukea viitteen kautta.

indeksi 1: puhelinnumerolla

indeksi 2: sukunimellä



51387
paakki
D240b

51316
wikla
C231

. . . huoneen nro: . . .

henkilöitä vastaavat tietueet

- Yhteen tietueeseen voi silloin viitata monesta eri indeksistä

  eli samaa tietoa käsitellä monen eri avaimen suhteen.

- Tietorakenteiden näkökulmasta

  tietokannat ovat ohjelmistoja, jotka ylläpitävät kovalevyllä tällaisia tietueita ja niiden indeksejä

  tietokantasuunnittelu analysoi tietojenkäsittelyongelmaa löytääkseen sopivat tietueet, kentät, indeksit. . .

  . . . **eheysehdot** ovat ohjelmointiperiaatteita, jotka pitävät tietorakenteen kunnossa, kuten

  ”jos poistat henkilöä vastaavan tietueen, niin muista poistaa myös siihen viittaavat avainsolmut nimi- ja puhelinnumeroindekseistä”.

3.5 B-puut

- Vanhassa tutkinnossa pääaineopiskelijoille pakollisella kurssilla *Tietokannan hallinta* esitellään niiden sukulaiset B^+-puut.

- Kalvojen 3.3 punamustat hakupuut ovat erinomaisia *keskusmuistissa:*

  – Jokainen solmu on *pienikokoinen*, tyypillisesti kymmeniä tavuja.

  – Ohjelma voi varata haluamansa kokoisia muistin palasia (operaatiolla **new**).

- *Kovalevy* on kuitenkin erilaista muistia:

  – Varataan *levylohkoja*, joiden *vakiokoko* riippuu kovalevystä, ei ohjelmasta.

  – Lohkon lukeminen levyltä on *paljon hitaampaa* kuin sen käsittely keskusmuistissa.

  – Mutta yhteen lohkoon mahtuu *paljon* tietoa, tyypillisesti useita kilotavuja.

- Siksi kovalevyllä säilytettävissä kalvojen 3.4 indekseissä käytetäänkin yleensä sellaisia hakupuita, joiden *yksi solmu täyttää kokonaisen levylohkon*.

- Lohko täytetään *haarautumalla useampaan kuin vain kahteen lapseen*.

  – Puu ei siis olekaan *binääri*puu.

  – Puun haaratumisasteella on kiinteä yläraja, joka riippuu lohkon koosta.

- Toiminta tehostuu:

  – Toisaalta keskusmuistiin luetun solmun käsittely hidastuu.

  – Toisaalta luetaan vähemmän solmuja.

  – Lukeminen on *paljon* (kymmeniä tuhansia kertoja) käsittelyä hitaampaa.

- Sama tilanne on lievempänä myös keskusmuistin ja *prosessorin välimuistin* (cache) välillä.

- B-puu T on seuraavanlainen puu:
  - Sillä on juuri root[T].
  - Jokaisella solmulla x on seuraavat kentät:
    * $n[x] =$ *tällä hetkellä* solmuun x talletettujen avainten lukumäärä

      joka voi siis vaihdella

    * nämä talletetut avaimet avainjärjestyksessä

      $$\text{key}_1[x] < \text{key}_2[x] < \cdots < \text{key}_{n[x]}[x]$$

    * leaf[x] $=$ **true** jos ja vain jos solmu x on *lehti* eli lapseton

    * jos leaf[x] $=$ **false** niin $n[x] + 1$ viitettä

      $$c_1[x], c_2[x], c_3[x], \ldots c_{n[x]+1}[x]$$

      sen lapsisolmuihin

      jotka ovat myös olemassa.

- Avainten ja viitteiden välillä vallitsee kalvojen 3.1.2 binäärihakupuuehdon laajennus $n[x]+1$-haaraiseen solmuun:

$$k_1 < \mathsf{key}_1[x] < k_2 < \mathsf{key}_2[x] < k_3 < \ldots < k_{n[x]} < \mathsf{key}_{n[x]}[x] < k_{n[x]+1}$$

  missä jokainen k_i on mielivaltainen avain alipuusta $c_i[x]$.

  – Binäärihakupuusolmun x ainoa avain $\mathsf{key}[x]$ jakoi muut avaimet kahteen osaan: vasempaan pienemmät ja oikeaan suuremmat.

  – Nyt $n[x]$-alkioinen *järjestetty avaintaulukko* jakaa muut avaimet $n[x]+1$ osaan vastaavasti:

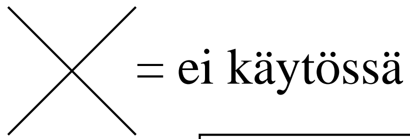
    * $c_1[x] = \cdots < \mathsf{key}_1[x]$

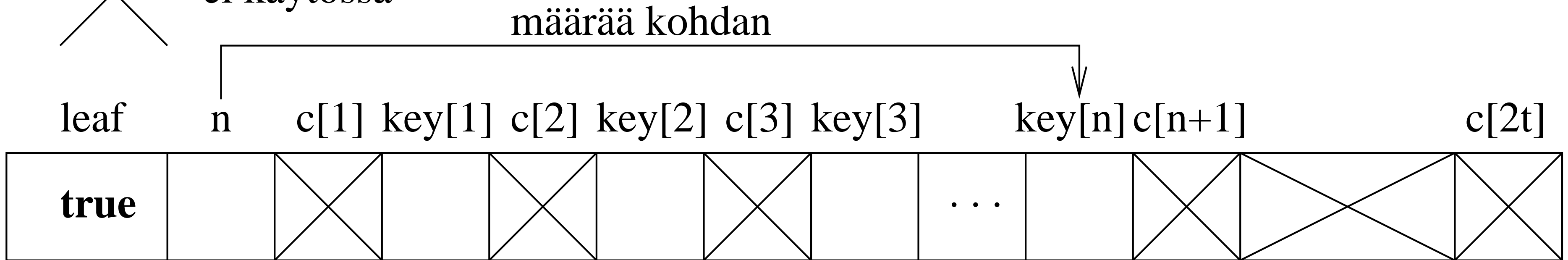
    * $c_2[x] = \mathsf{key}_1[x] < \cdots < \mathsf{key}_2[x]$

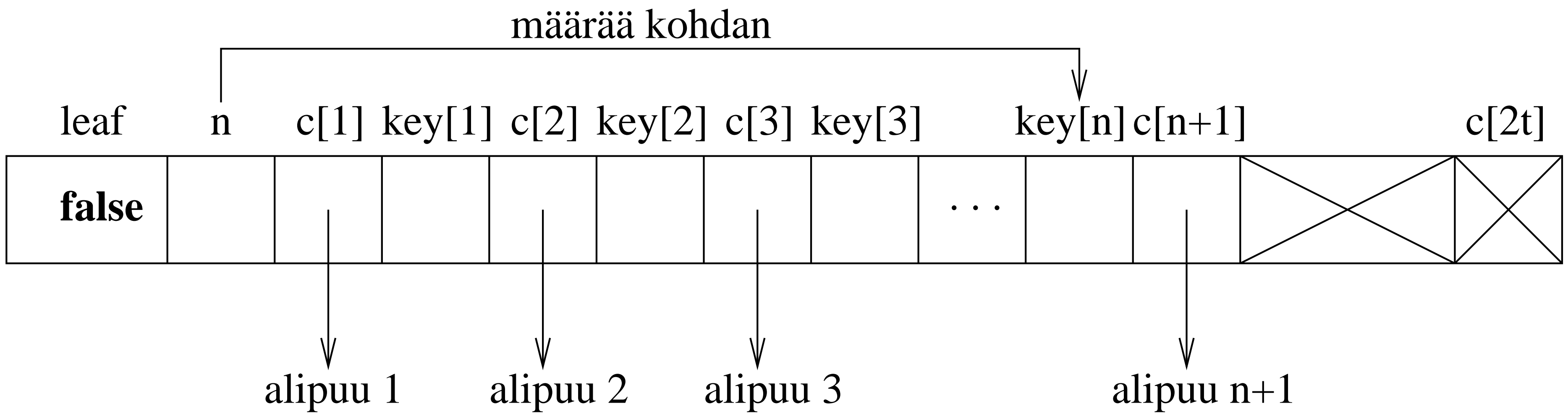
    * $c_3[x] = \mathsf{key}_2[x] < \cdots < \mathsf{key}_3[x]$

    * ...

    * $c_{n[x]+1}[x] = \mathsf{key}_{n[x]}[x] < \ldots.$


= ei käytössä


määrää kohdan
leaf
n
c[1]
key[1]
c[2]
key[2]
c[3]
key[3]
key[n]
c[n+1]
c[2t]
true
. . .


määrää kohdan
leaf
n
c[1]
key[1]
c[2]
key[2]
c[3]
key[3]
key[n]
c[n+1]
c[2t]
false
. . .
alipuu 1
alipuu 2
alipuu 3
alipuu n+1

- B-puun tasapainoehdot:
  - Kaikki lehdet ovat samalla syvyydellä joka on puun korkeus h.
  - Jokaisen solmun x avainten lukumäärää säätelee *vakio* $t \geq 2$*:*

    $$t - 1 \leq n[x] \leq 2 \cdot t - 1.$$

    Jos siis leaf$[x] = \textbf{false}$, niin solmulla x on $t \ldots 2 \cdot t$ lasta.
  - root$[T]$ *on poikkeus* (myös algoritmeissa): joko

    $$1 \leq n[\text{root}[T]] \leq 2 \cdot t - 1$$

    tai puu on kokonaan tyhjä.
- Puuta tasapainotetaan (ei kierroin ja värein vaan) *lisäämällä ja vähentämällä lapsien lukumäärää* sen solmuissa näiden rajojen sisällä.

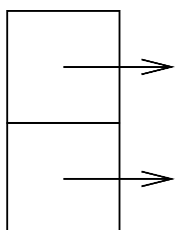
- Vakio t valitaan levylohkon koon mukaan.

- Yksinkertaisin tapaus $t = 2$ on *2-3-4-puu:*
  - Jokaisella solmulla on 2, 3 tai 4 lasta.
  - Tai 0, jos solmu on lehti.
  - Seuraavassa kuvassa on 2-3-4- ja kalvojen 3.3 punamustien solmujen yhteys.

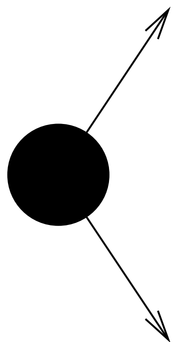
- Sitä seuraavassa kuvassa
  - jokaisessa solmussa on 1000 avainta
  - puun T korkeus $h = 2$

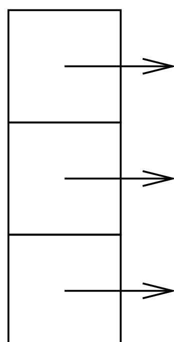
    jolloin jokainen haku tutkii ≤ 3 levylohkoa
  - puussa T on maksimimäärä solmuja
  - puussa T on hieman yli *miljardi* avainta!

(Kuva 18.3 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

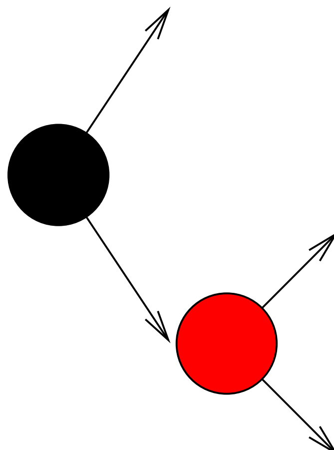


on

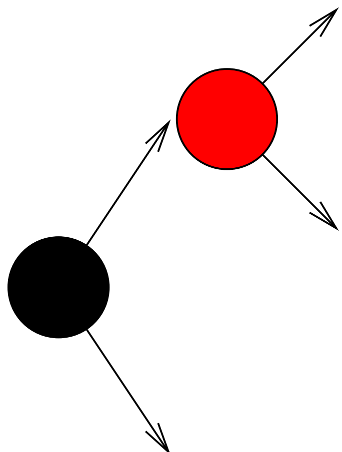


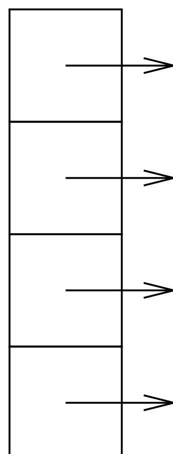


on

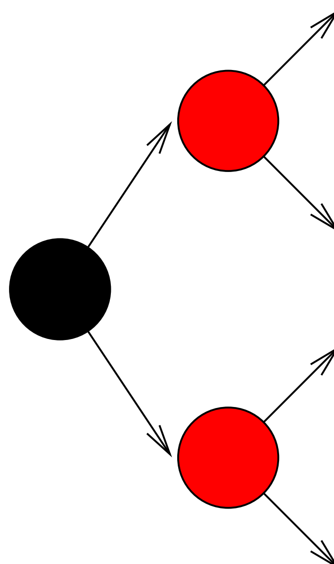


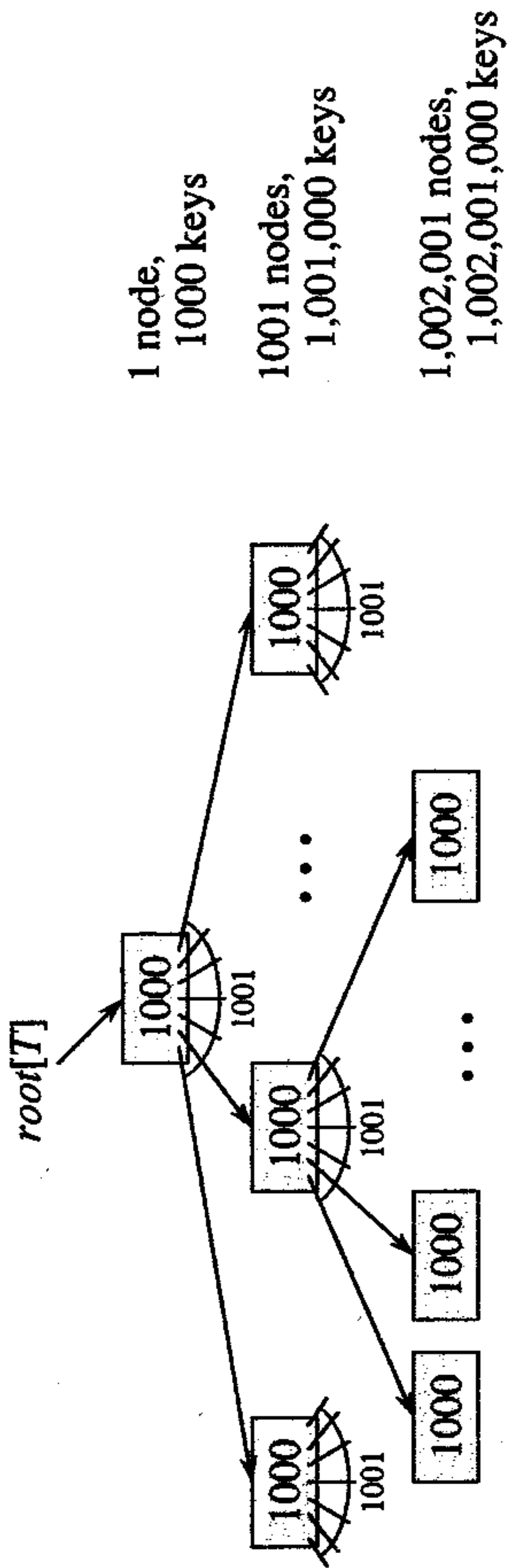
tai





on




root[T]
1000
1001
1 node,
1000 keys
1001 nodes,
1,001,000 keys
1,002,001 nodes,
1,002,001,000 keys

- Voidaan osoittaa (vastaavasti kuin binäärihakupuille) että B-puun korkeus

$$h \leq \log_t \left(\frac{n+1}{2} \right)$$

kun B-puussa on $n \geq 1$ avainta.

- Siis myös B-puun korkeus h on logaritminen
  - mutta kantalukuna onkin minimihaarautumisaste t
  - eikä vain 2 kuten tasapainoisessa binäärihakupuussa.

3.5.1 Avaimen haku

- Avaimen k haku B-puusta T tapahtuu rekursiokutsulla

$$\text{BTreeSearch}(\text{root}[T], k)$$

missä

```
BTreeSearch(x, k)
1   i ← 1
2   while i ≤ n[x] and k > key_i[x] do
3       i ← i + 1
4   if i ≤ n[x] and key_i[x] = k then
5       return (x, i)
6   if leaf[x] then
7       return NIL
8   else
9       DiskRead(c_i[x])
10      return BTreeSearch(c_i[x], k)
```

- Rekursiokutsun tuloksena saadaan
  - pari (x, i) missä $\text{key}_i[x] = k$ (rivit 4–5)
  - NIL (rivit 6–7) jos puussa T ei ole yhtään sellaista solmua x, jossa esiintyisi avain k.

- Algoritmi on suora yleistys binäärisistä useampihaaraisiin hakupuihin:

  - Riveillä 1–3 haetaan se kohta i, jossa avain k voisi esiintyä

    * nykyisessä solmussa x

    * alipuussa $c_i[x]$

    johon edetään rekursiivisesti rivillä 10.

  - Ennen rekursiota haetaan rivillä 9 viitteen $c_i[x]$ osoittama levylohko keskusmuistiin.

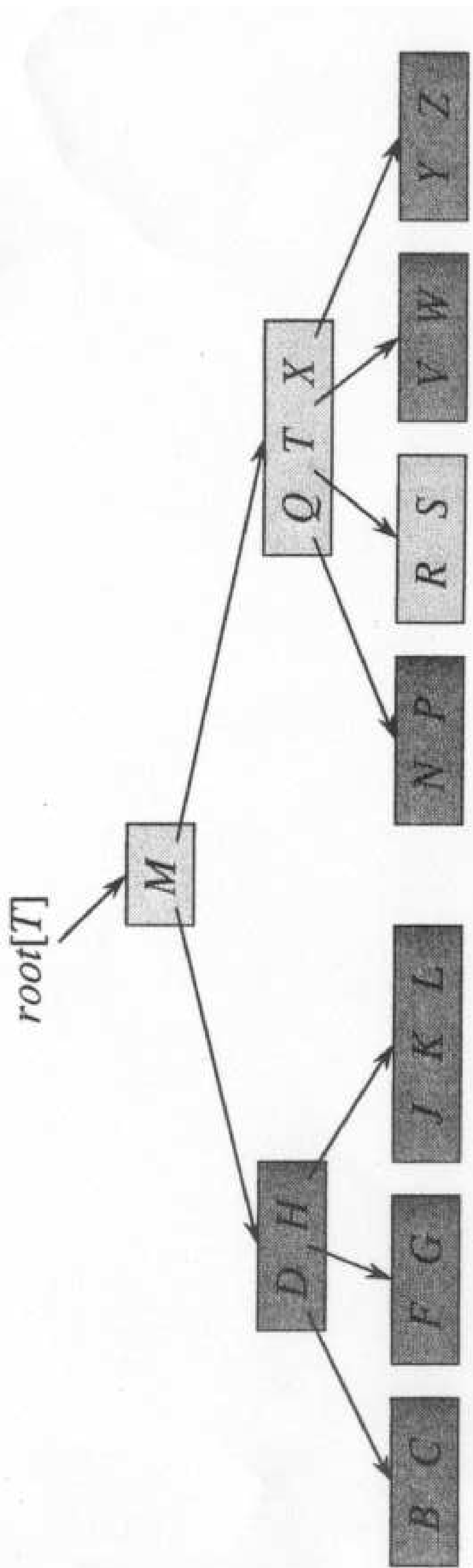
    * Algoritmi siis olettaa, että solmu x on luettu keskusmuistiin.

    * Erityisesti siis root$[T]$ oli valmiiksi keskusmuistissa.

- Seuraavassa kuvassa haetaan avainta R.

  Käsitellyt solmut on piirretty vaaleammalla harmaalla.

  (Kuva 18.1 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)



root[T]
M
D H
Q T X
B C
F G
J K L
N P
R S
V W
Y Z

- B-puualgoritmeissa kannattaa lohkojen siirrot keskusmuistin ja levyn välillä merkitä näkyviin, koska

  – ne ovat operaatioista raskaimmat

  – algoritmi ei saisi pitää levylohkoja turhaan keskusmuistissa

    * Nämä ovat levy*tietokanta*-algoritmeja, joten niiden ei pitäisi turhaan

      · kopioida tietokantaa levyltä keskusmuistiin

      · lukita levylohkoja odottamaan päivittämistä, jolloin toiset rinnakkain suoritettavat operaatiot eivät pääse lukemaan niitä.

    * Oletamme algoritmeissamme, että (Java-)roskankerääjä hävittää keskusmuistiin luetun levylohkon varaaman keskusmuistitilan ja lukituksen sitten kun lohkoa ei enää käsitellä.

- Haku vie ajan

$$\mathcal{O}\left(t \cdot h\right).$$

  - Vakiokerroin t johtuu rivien 1–3 *peräkkäis*hausta (kalvoilta 1.3.1).
  - Se pienentyy arvoon

    $$\log_2 t$$

    jos käytetäänkin *binääri*hakua (kalvoilta 1.3.3).
  - Toisaalta käytännössä levyoperaatiot määräävät suoritusajan, ei haku pienessä palassa keskusmuistia.

- Funktion BTreeSearch(x, k) takarekursio voidaan korvata silmukalla (kalvoilta 3.1.3).

  Silloin nähdään, että funktio pitää muistissa vain yhtä levylohkoa x kerrallaan.

3.5.2 Tyhjän puun luonti

- Esitetään tyhjä B-puu T tyhjänä levylohkona:

  BTreeCreate(T)
  1 $x \leftarrow$ **new** B-puusolmu keskusmuistiin
  2 leaf[x] $\leftarrow$ **true**
  3 $n[x] \leftarrow 0$
  4 DiskWrite(x)
  5 root[T] $\leftarrow x$

- Rivillä 5 kirjoitetaan levylle sellainen solmu x, jonka sisältöä on muutettu keskusmuistissa.

- Merkitään siis nämäkin operaatiot näkyviin algoritmeissamme.

3.5.3 Avaimen lisäys

- Avaimen k lisäys B-puuhun T on vaikeampaa kuin binääripuuhun:
  - Emme voi luoda uutta solmua pelkästään avainta k varten

    koska se rikkoisi minimihaarautumisastetta t.
  - Sen sijaan avain k on sijoitettava johonkin jo olemassa olevaan solmuun.
  - Päätämme: sijoitetaan uusi avain aina lehteen.
  - Mutta entä jos tähän lehteen ei enää mahdukaan enempää avaimia

    koska avainten maksimimäärä $2 \cdot t - 1$ yhdessä solmussa ylittyisi?

- Oletetaan, ettei avain k vielä esiinny puussa T.

- Täysi solmu voidaan *halkaista* kahdeksi mahdollisimman vajaaksi solmuksi:

  Ensimmäiset $t-1$ avainta

  $$\mathsf{key}_1[y], \mathsf{key}_3[y], \ldots, \mathsf{key}_{t-1}[y]$$

  ja t viitettä

  $$c_1[y], c_2[y], \ldots, c_t[y]$$

  jäävät nykyiseen solmuun y.

  Viimeiset $t-1$ avainta

  $$\mathsf{key}_{t+1}[y], \mathsf{key}_{t+2}[y], \ldots, \mathsf{key}_{2\cdot t-1}[y]$$

  ja t viitettä

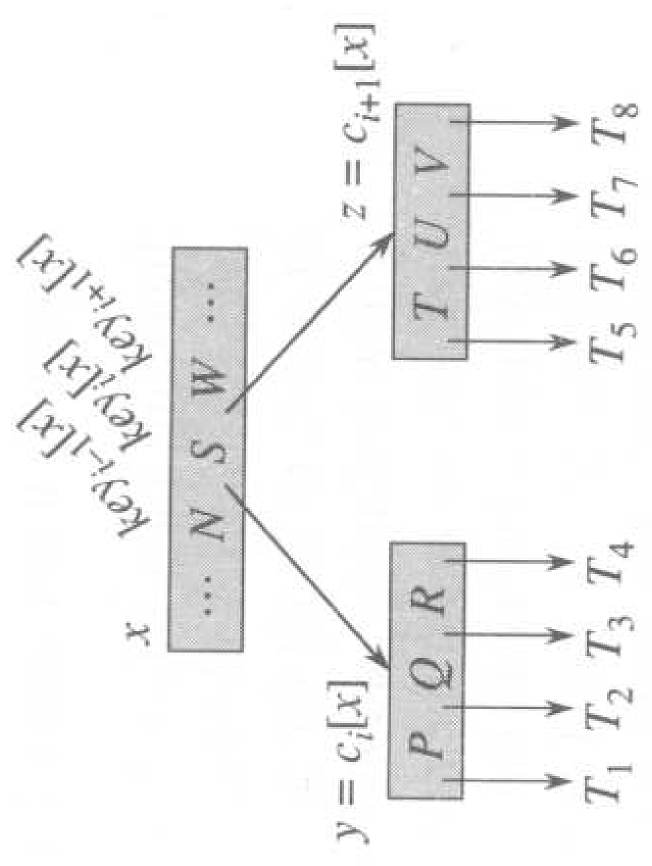
  $$c_{t+1}[y], c_{t+2}[y], \ldots, c_{2\cdot t}[y]$$

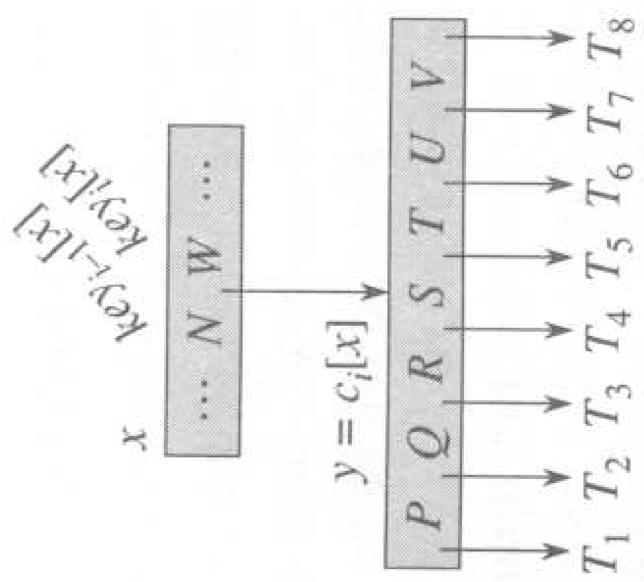
  muodostavat uuden solmun z.

  Keskimmäinen avain $\mathsf{key}_t[y]$ nousee isäsolmuun x erottamaan peräkkäisiä alipuita y ja z.

- Seuraavassa esimerkkikuvassa halkaistaan isäsolmun x i:s lapsi y.

  (Kuva 18.5 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)


x
key_{i-1}[x]
key_i[x]
key_{i+1}[x]
... N S W ...
y = c_i[x]
z = c_{i+1}[x]
P Q R
T U V
T_1 T_2 T_3 T_4
T_5 T_6 T_7 T_8


x
key_{i-1}[x]
key_i[x]
... N W ...
y = c_i[x]
P Q R S T U V
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8

- Tässä pseudokoodissakin argumentit x, i ja y ovat kuten edellisessä esimerkkikuvassa.

BTreeSplitChild(x, i, y)

1 $z \leftarrow$ **new** B-puusolmu keskusmuistiin
2 leaf[z] $\leftarrow$ leaf[y]
3 $n[z] \leftarrow t - 1$
4 **for** $j \leftarrow 1$ **to** $t - 1$ **do**
5   key$_j[z] \leftarrow$ key$_{t+j}[y]$
6 **if not** leaf[y] **then**
7   **for** $j \leftarrow 1$ **to** t **do**
8         $c_j[z] \leftarrow c_{t+j}[y]$
9 $n[y] \leftarrow t - 1$
10 **for** $j \leftarrow n[x] + 1$ **downto** $i + 1$ **do**
11     $c_{j+1}[x] \leftarrow c_j[x]$
12 $c_{i+1}[x] \leftarrow z$
13 **for** $j \leftarrow n[x]$ **downto** i **do**
14   key$_{j+1}[x] \leftarrow$ key$_j[x]$
15 key$_i[x] \leftarrow$ key$_t[y]$
16 $n[x] \leftarrow n[x] + 1$
17 DiskWrite(y)
18 DiskWrite(z)
19 DiskWrite(x)

- Riveillä 1–3 luodaan uusi solmu z.

- Riveillä 4–9 siirretään solmuun z solmun y jälkimmäinen puolisko.

- Riveillä 10–12 lisätään viite z solmuun x.

- Riveillä 13–16 lisätään avain $\mathrm{key}_t[y]$ solmuun x.

- Huomaa: solmun x olemassaolevia
  - viitteitä (rivit 10–11) ja
  - avaimia (rivit 13–14)

  siirretään yksi paikka eteenpäin

  jotta uudelle viitteelle ja avaimelle saadaan tilaa oikealle kohdalleen.

- Rivillä 17 kirjoitetaan puolittunut solmu y levylle.

- Rivillä 18 kirjoitetaan uusi solmu z levylle.
  - Käytännössä solmulle z varataan tällöin uusi aikaisemmin käyttämätön levylohko.

    (Samoin kuin operaatio **new** keskusmuistissa.)
  - Olkoon b tämän varatun lohkon osoite levyllä.

- Rivillä 19 kirjoitetaan muuttunut isäsolmu x levylle.
  - Käytännössä viitteenä $z = c_{i+1}[x]$ (rivi 12) kirjoitetaan b.

- Halkaisuun kuluu

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

  askelta (viitteiden ja avainten siirtelyyn).

- Halkaisun aikana muistissa on 3 levylohkoa x, y ja z.

- Halkaisualgoritmi BTreeSplitChild(x, i, y) olettaa seuraavaa:

  1. Solmut x ja y ovat jo keskusmuistissa.

  2. Solmu x ei itse ole täysi

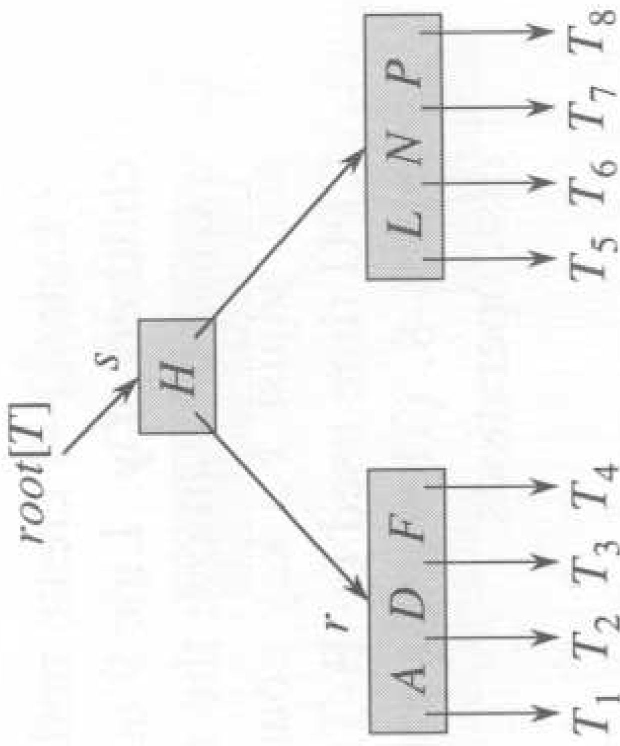
     – jotta sitä itseään ei tarvitse jakaa

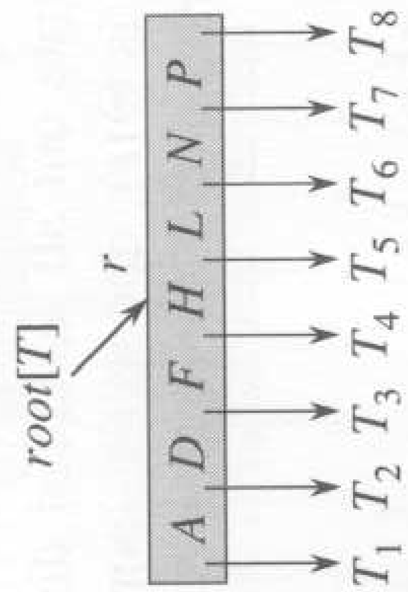
     – kun siihen lisätään uusi avain ja viite.

- Lisäysalgoritmin pääohjelma takaa halkaisuoletuksen 2 jakamalla juuren jos tarpeen:

```
BTreeInsert(T, k)
1  if n[root[T]] = 2 · t − 1 then
2       s ← new B-puusolmu
3       c_1[s] ← root[T]
4       root[T] ← s
5       leaf[s] ← false
6       n[s] ← 0
7       BTreeSplitChild(s, 1, c_1[s])
8  BTreeInsertNonfull(root[T], k)
```

(Kuva 18.6 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)


root[T]
s
H
r
A D F
L N P
T_1 T_2 T_3 T_4
T_5 T_6 T_7 T_8


root[T]
r
A D F H L N P
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8

- Varsinainen rekursiivinen lisäysalgoritmi voi silloin tehdä saman halkaisuoletuksen 2:

BTreeInsertNonfull(x, k)
1 $i \leftarrow n[x]$
2 **if** leaf[x] **then**
3 **while** $i \geq 1$ **and** $k < \text{key}_i[x]$ **do**
4 $\text{key}_{i+1}[x] \leftarrow \text{key}_i[x]$
5 $i \leftarrow i - 1$
6 $\text{key}_{i+1}[x] \leftarrow k$
7 $n[x] \leftarrow n[x] + 1$
8 DiskWrite(x)
9 **else**
10 **while** $i \geq 1$ **and** $k < \text{key}_i[x]$ **do**
11 $i \leftarrow i - 1$
12 $i \leftarrow i + 1$
13 DiskRead($c_i[x]$)
14 **if** $n[c_i[x]] = 2 \cdot t - 1$ **then**
15 BTreeSplitChild($x, i, c_i[x]$)
16 **if** $k > \text{key}_i[x]$ **then**
17 $i \leftarrow i + 1$
18 BTreeInsertNonfull($c_i[c], k$)

- Jos solmu x on lehti (rivi 2), niin
  1. tehdään tilaa avaimelle k (rivit 3–5)
  2. talletetaan k paikoilleen (rivit 6–8).

- Jos solmu x ei ole lehti:
  1. Etsitään sellainen alipuu $c_i[x]$ johon k voidaan lisätä (rivit 10–12).
  2. Luetaan tämä alipuun $c_i[x]$ juurisolmu keskusmuistiin (rivi 13)

     jotta halkaisuoletus 1 saadaan voimaan.
  3. Jos tämä juurisolmu $c_i[x]$ on täynnä (rivi 14), niin
     (a) halkaistaan se (rivi 15)
     (b) jotta rekursiossa (rivi 18) pätee halkaisuoletus 2
     (c) valitaan haljenneista paloista avaimelle k sopiva (rivit 16–17).

- Seuraavassa esimerkkisarjakuvassa $t = 3$ ja muuttuneet solmut on varjostettu vaaleammin:

  (b) Avaimelle B on vielä tilaa lehdessä.

  (c) Avaimelle Q ei ole tilaa lehdessä $RSTUV$, joten se halkaistaan ensin.

  (d) Lisätään avainta L, mutta juuri $GMPTX$ on täynnä.

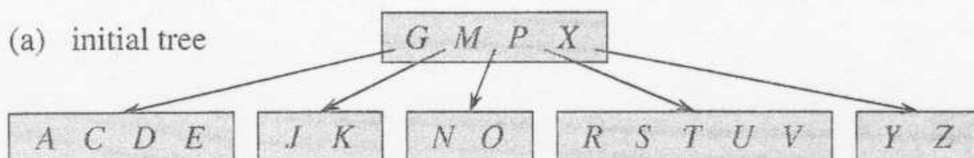
  - Halkaistaan juuri pääohjelmassa.

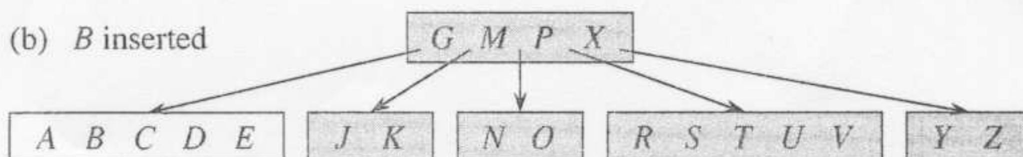
  - Juuren halkaisu on ainoa tapa, jolla B-puu kasvaa korkeutta.

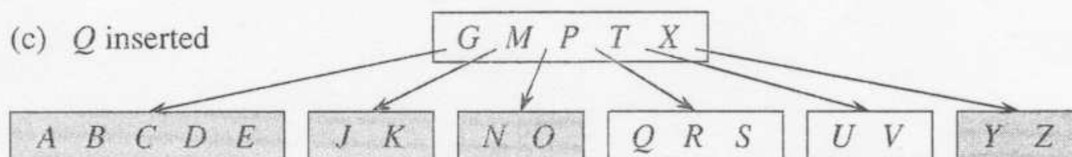
  - Juuren halkaisun jälkeen tehdään normaali lisäys.

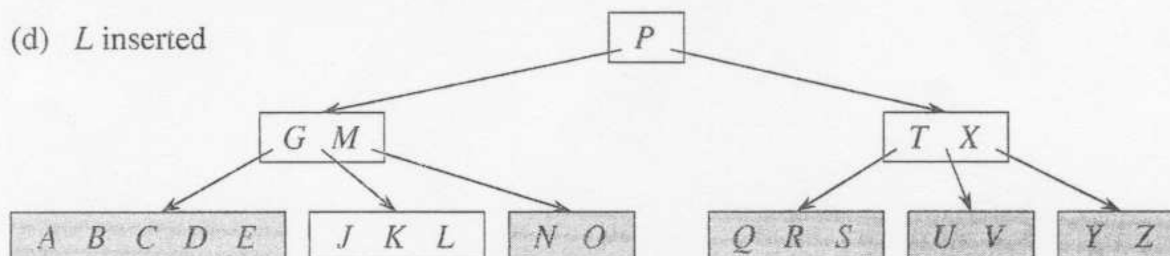
  (e) Samoin kuin **(c)**.

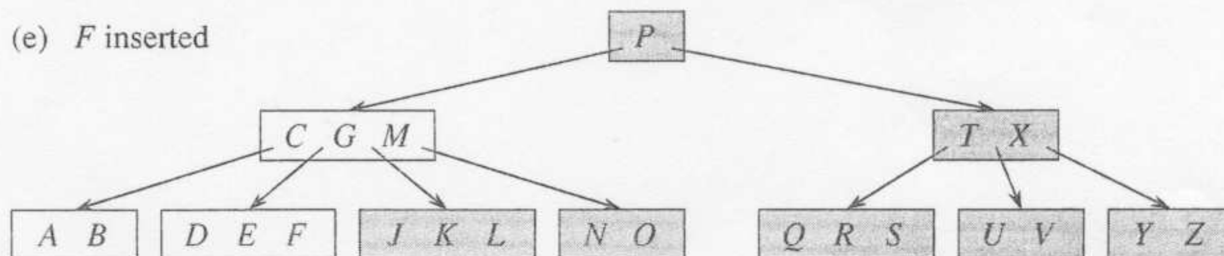
  (Kuva 18.7 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)


(a) initial tree
G M P X
A C D E
J K
N O
R S T U V
Y Z


(b) B inserted
G M P X
A B C D E
J K
N O
R S T U V
Y Z


(c) Q inserted
G M P T X
A B C D E
J K
N O
Q R S
U V
Y Z


(d) L inserted
P
G M
T X
A B C D E
J K L
N O
Q R S
U V
Y Z


(e) F inserted
P
C G M
T X
A B
D E F
J K L
N O
Q R S
U V
Y Z

- Lisäysalgoritmin ajantarve on

$$\mathcal{O}(t \cdot h)$$

  missä vakiokerroin t johtuu avainten sekä viitteiden siirtelystä keskusmuistissa.

- Kuten haussakin (kalvoilla 3.5.1), myös nyt käytännön suoritusajan määrää käsiteltyjen levylohkojen lukumäärä

$$\mathcal{O}(h).$$

  Keskusmuistissa tehty työ on niin paljon nopeampaa.

- Muistissa pidetään korkeintaan 3 levylohkoa kerrallaan

  halkaisua varten.

- Tämä lisäysalgoritmi halkoo täysiä solmuja *jo edetessään* lisäyskohtaan.

  – Täysi solmu halkaistaan *varmuuden vuoksi*

    vaikka vielä ei tiedetäkään olisiko se välttämätöntä.

  + Ei tarvitse palata käsittelemään samaa levylohkoa.

- Voitaisiin myös toimia päinvastoin:

  1. Ensin edetään rekursiolla suoraan lisäyskohtaan

  2. sitten *vasta palatessa* halkaistaan ne solmut, jotka ylittivät suurimman sallitun koon.

  + Solmu halkaistaan vain jos se on välttämätöntä.

  – Täytyy palata käsittelemään samaa levylohkoa.

- Halkaisu

  edetessä on parempi levytallennuksessa

  palatessa on parempi keskusmuistissa.

- Kalvojen 3.3 punamustat puut ovatkin kalvojen 3.5 kuvan mukainen 2-3-4-puiden toteutus siten, että tasapainotus

  - eli 2-3-4-solmujen halkaisu

  - eli kierrot ja värien vaihdot

  tehdäänkin vasta palatessa puuta takaisin juurta kohden.

- Vastaavasti myös punamustista puista voi johtaa jo edetessä tasapainottavan version

  mutta silloin tehtävien kiertojen lukumäärä ei enää olekaan vakio / operaatio.

  (Versio kuvattu sivuilla 219–228 kirjassa R. Sedgewick: *Algorithms. 2nd Ed.* Addison-Wesley 1988.)

3.5.4 Avaimen poisto

- B-puusta T poistetaan avain k

  ei sitä solmua jonka avain on k, koska yhdessä solmussa on nyt monta eri avainta.

- Kun solmusta poistuu yksi sen avaimista, sen viitteet ja muut avaimet täytyy järjestellä uudelleen.

- Poistoalgoritmissa huolehditaan *etukäteen*, että käsiteltävällä solmulla on enemmän kuin minimimäärä $t - 1$ avaimia.

- Tyydytään kuvailemaan eri tapaukset yleistasolla tarkan pseudokoodin sijasta.

  Yksityiskohtia olisi niin paljon.

1. Jos avain k on lehtisolmussa x, niin poista se sieltä.

   Tämä on näin helppoa juuri siksi että $n[x] \geq t$ (tai erikoistapauksena $x = \text{root}[T]$).

2. Jos avain k on sisäsolmussa x, niin seuraavat 3 vaihtoehtoa kattavat kaikki tilanteet:

   (a) Jos avainta k edeltävä solmu y on minimiä suurempi ($n[y] \geq t$), niin:

   - Etsi ja poista rekursiivisesti alipuusta y avaimen k edeltäjä k' = alipuun y oikeanpuoleisimman (lehti)solmun oikeanpuoleisin avain.

   - Korvaa solmussa x avain k avaimella k'.

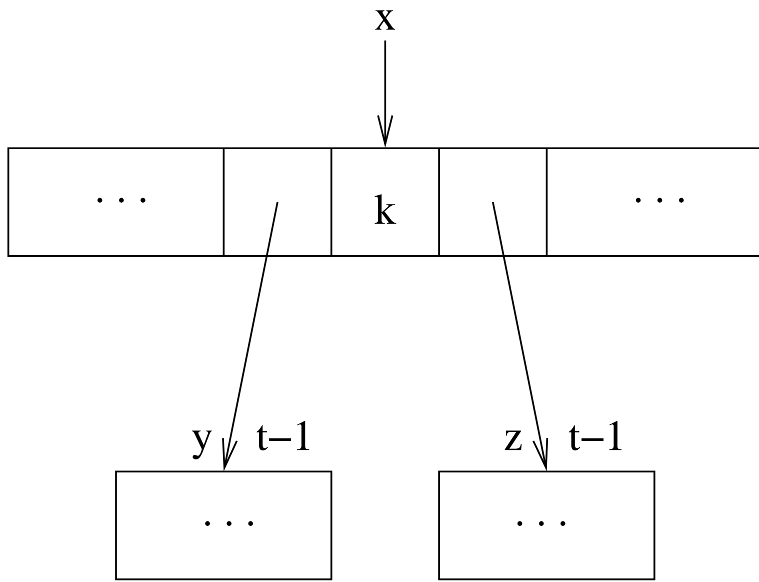
   (b) Edellinen vaihtoehto 2a vasen/oikea-symmetrisesti.

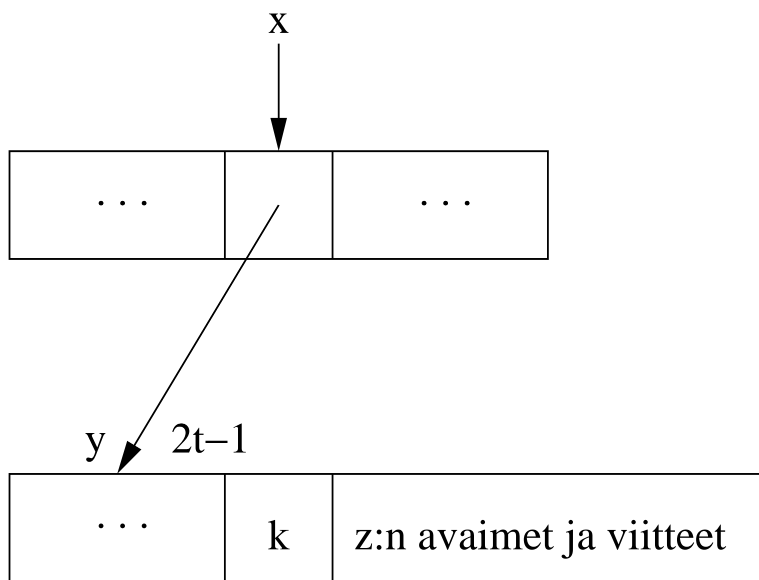
   (c) Muuten avainta k sekä edeltävä solmu y että seuraava solmu z ovat minimikokoisia ($n[y] = n[z] = t - 1$).

   i. Liitä solmun y sisällön perään ensin avain k ja sitten solmun z sisältö.

      Poista solmusta x avain k ja viite z.

   ii. Poista rekursiivisesti avain k solmusta y, joka kasvoi (maksimikokoiseksi) askeleessa 2(c)i.


x
. . .
k
. . .
y t−1
z t−1
. . .
. . .

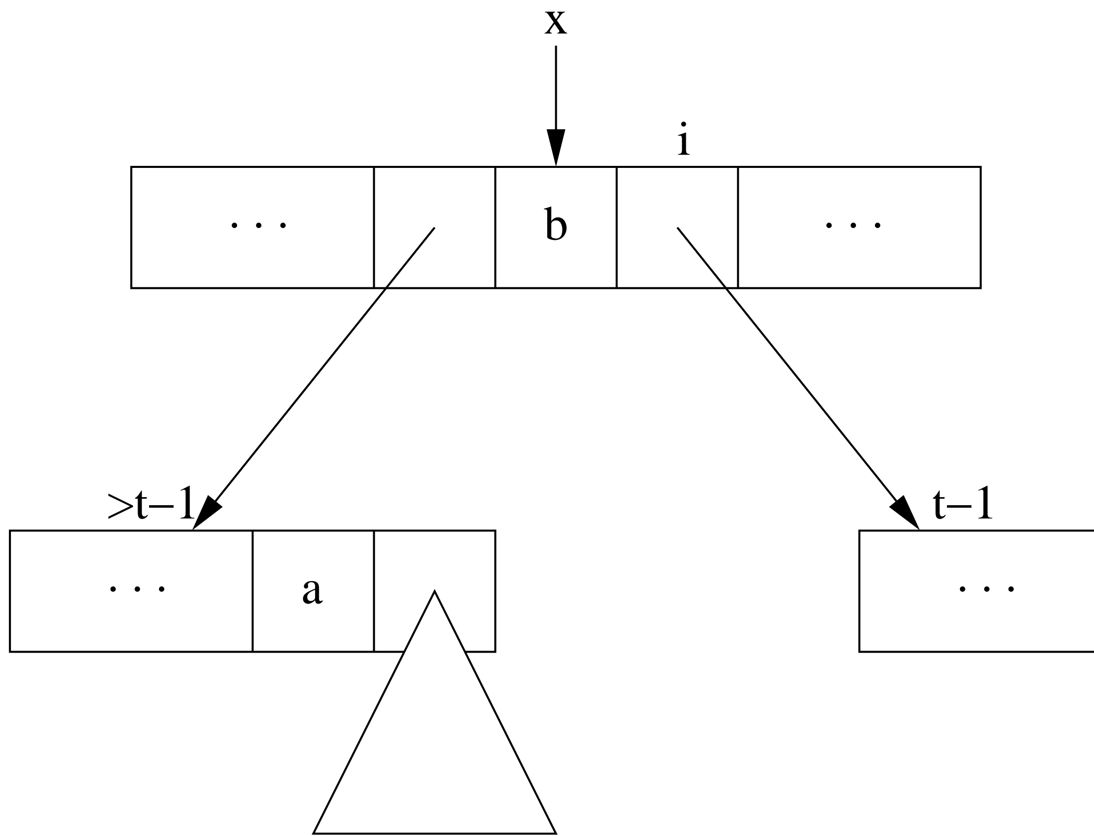

x
. . .
. . .
y 2t−1
. . .
k
z:n avaimet ja viitteet
z poistuu

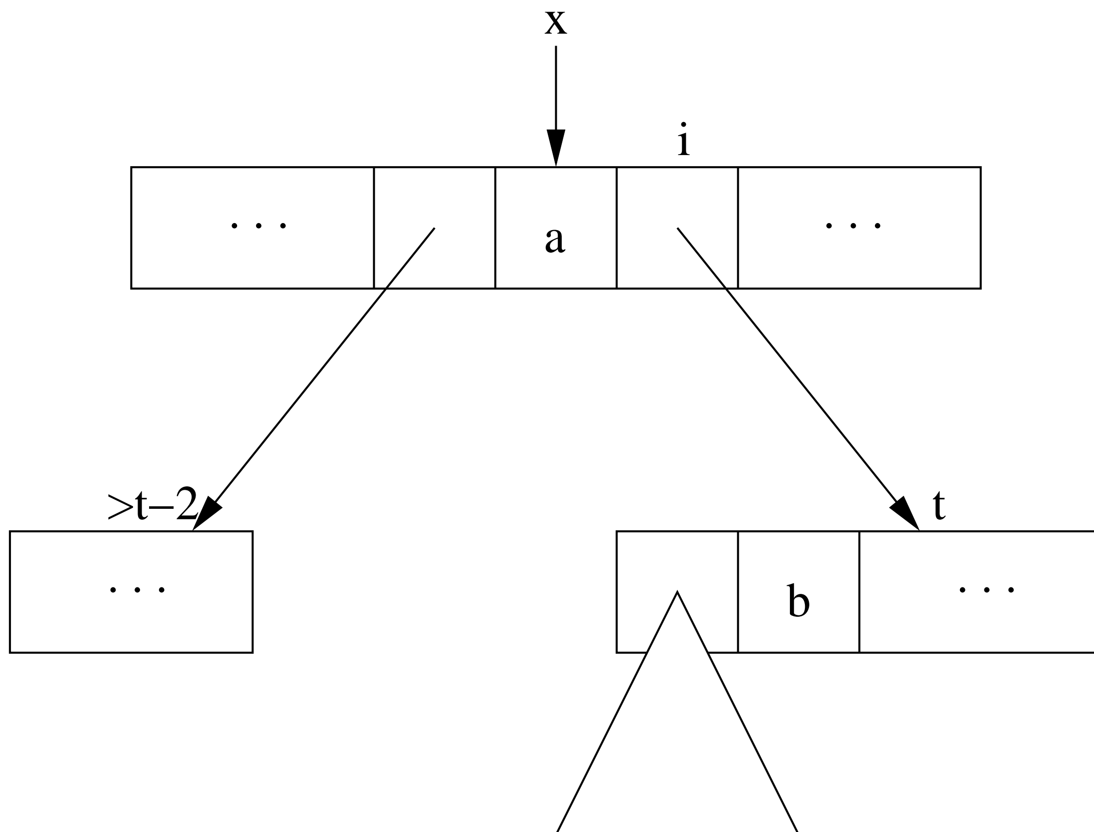
3. Muuten sisäsolmu x ei sisällä avainta k, joten poisto jatkuu rekursiivisesti.

- Olkoon $c_i[x]$ se, johon pitää edetä.
- Jos $c_i[x]$ on minimikokoinen ($n[c_i[x]] = t - 1$), niin sitä pitää kasvattaa ennen rekursiota.
  (a) Jos solmulla $c_i[x]$ on vasen naapuri $c_{i-1}[x]$ joka on minimikokoa suurempi ($n[c_{i-1}[x]] \geq t$), niin siirrä
    - kohtaa i edeltävä avain ($\text{key}_{i-1}[x]$) solmun $c_i[x]$ alkuun
    - naapurin viimeinen avain ($\text{key}_{i-1}[n[c_{i-1}[x]]]$) sen tilalle
    - naapurin viimeinen viite ($c_{i-1}[n[c_{i-1}[x] + 1]]$) solmun $c_i[x]$ alkuun.

    Oikealle $c_{i+1}[x]$ symmetrisesti.

  (b) Muuten yhdistä solmu $c_i[x]$ jomman kumman (tai ainoan) naapurinsa kanssa kuten askeleessa 2c.


x
i
. . .
b
. . .
>t−1
t−1
. . .
a
. . .


x
i
. . .
a
. . .
>t−2
t
. . .
b
. . .

- Erikoistapaus:
  - *Jos* käsitellään juurta $x = \text{root}[T]$ ...
  - jossa on vain 1 avain ($n[x] = 1$) ja ...
  - päädytään yhdistämään sen ainoat lapset $c_1[x]$ ja $c_2[x]$ askeleissa 2c ja 3b.
  - *niin* silloin vanha juuri x korvautuu kasvaneella lapsellaan $c_1[x]$ ja ...
  - B-puun korkeus pienenee (yhdellä).

- Seuraavassa esimerkkisarjakuvassa $t = 3$ ja muuttuneet solmut ovat vaaleampia:

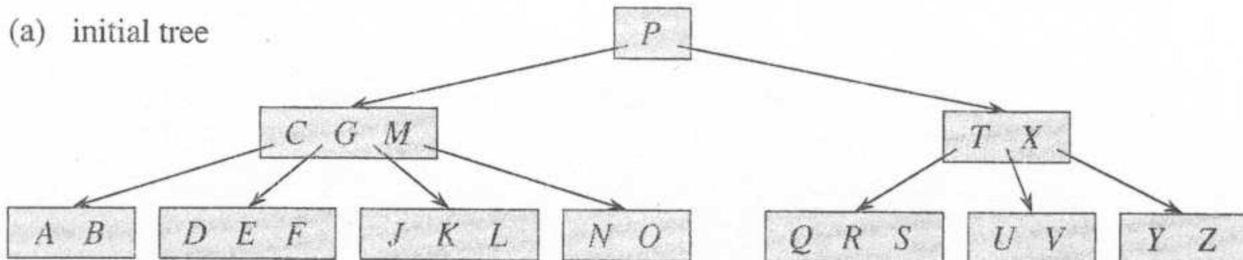
  (b) Avaimen F poisto on helppo tapaus 1.

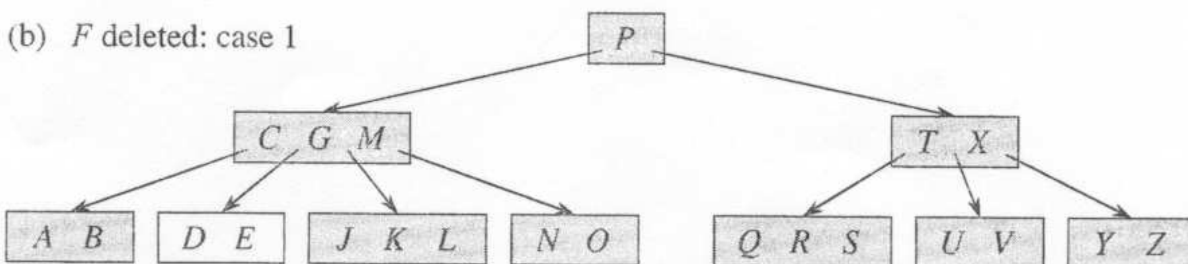
  (c) Avaimen M poisto on tapaus 2a:

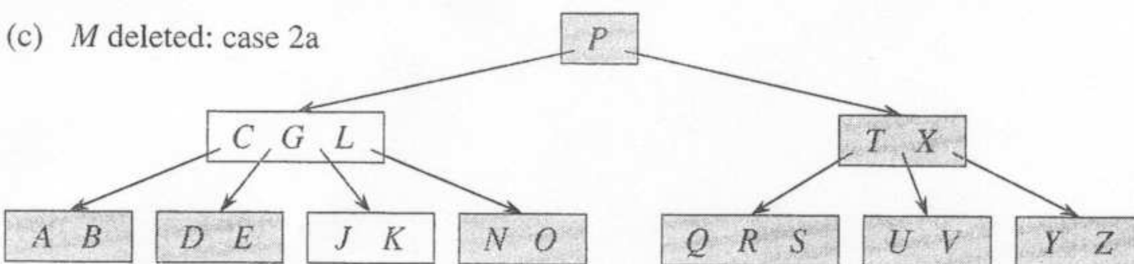
  edeltävä avain L ottaa sen paikan.

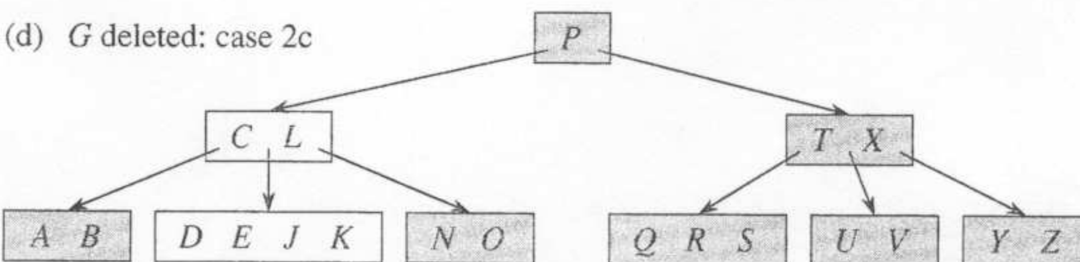
  (d) Avaimen G poisto on tapaus 2c:

  ensin rakennetaan väliaikainen solmu $DEGJK$, johon voidaan edetä.









(Kuva 18.8 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

- Esimerkkisarjakuva jatkuu:

  (e) Avaimen D poisto johtaa tapaukseen 3b:

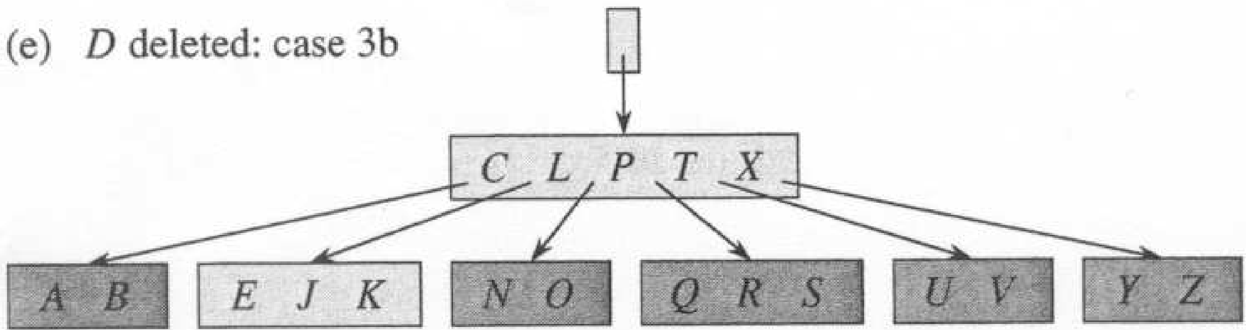
  - Ensin rakennetaan solmu $CLPTX$, jotta siihen voidaan edetä.
  - Varsinainen poisto tapahtuu vasta sen alapuolella.

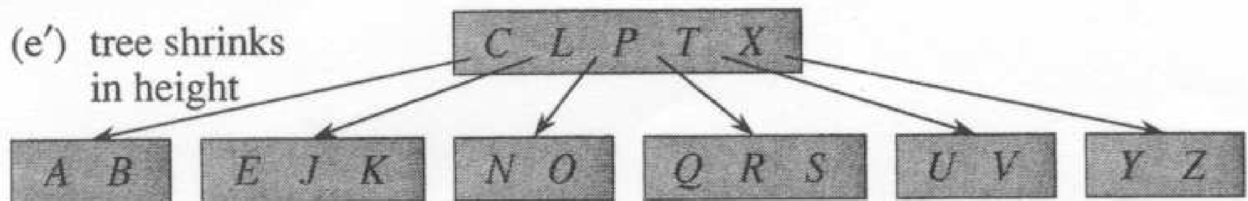
  (e') edellinen kuva **(e)** johtaa erikoistapaukseen:

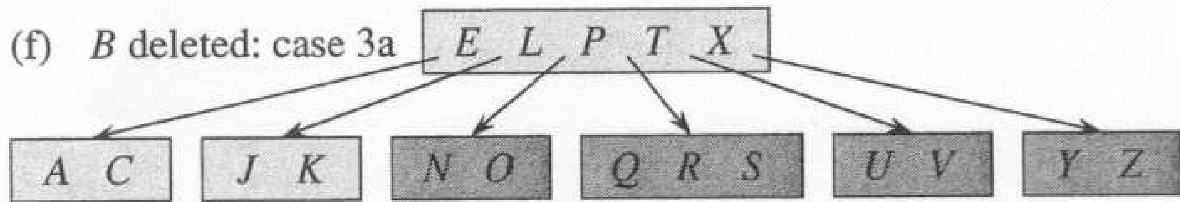
  - root[T] tyhjeni
  - ja korvautui solmulla $CLPTX$.

  (f) Avaimen B poisto on tapaus 3a symmetrisesti:

  - Solmu AB ottaa loppuunsa isänsä seuraavan avaimen C.
  - Isä saa sen tilalle seuraavan lapsensa EJK ensimmäisen avaimen E.
  - Jos käsiteltäisiin sisäsolmuja, niin solmu ABC saisi vielä loppuunsa solmun EJK ensimmäisen viitteen.


(e) D deleted: case 3b
C L P T X
A B
E J K
N O
Q R S
U V
Y Z


(e′) tree shrinks in height
C L P T X
A B
E J K
N O
Q R S
U V
Y Z


(f) B deleted: case 3a
E L P T X
A C
J K
N O
Q R S
U V
Y Z

- Resurssitarpeet ovat samanlaiset

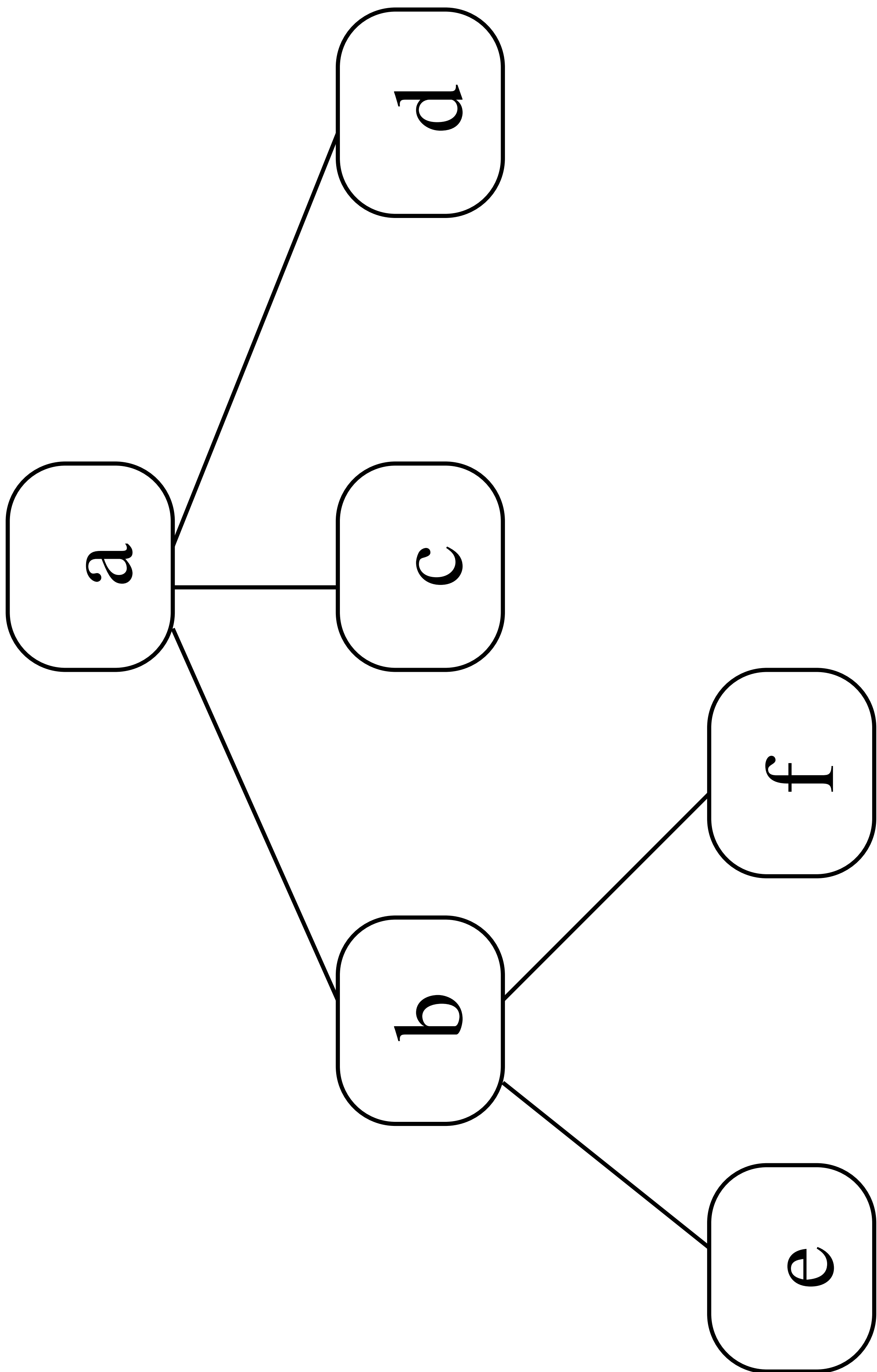
$\mathcal{O}(t \cdot h)$	askelta
$\mathcal{O}(1)$	lohkoa yhtä aikaa muistissa

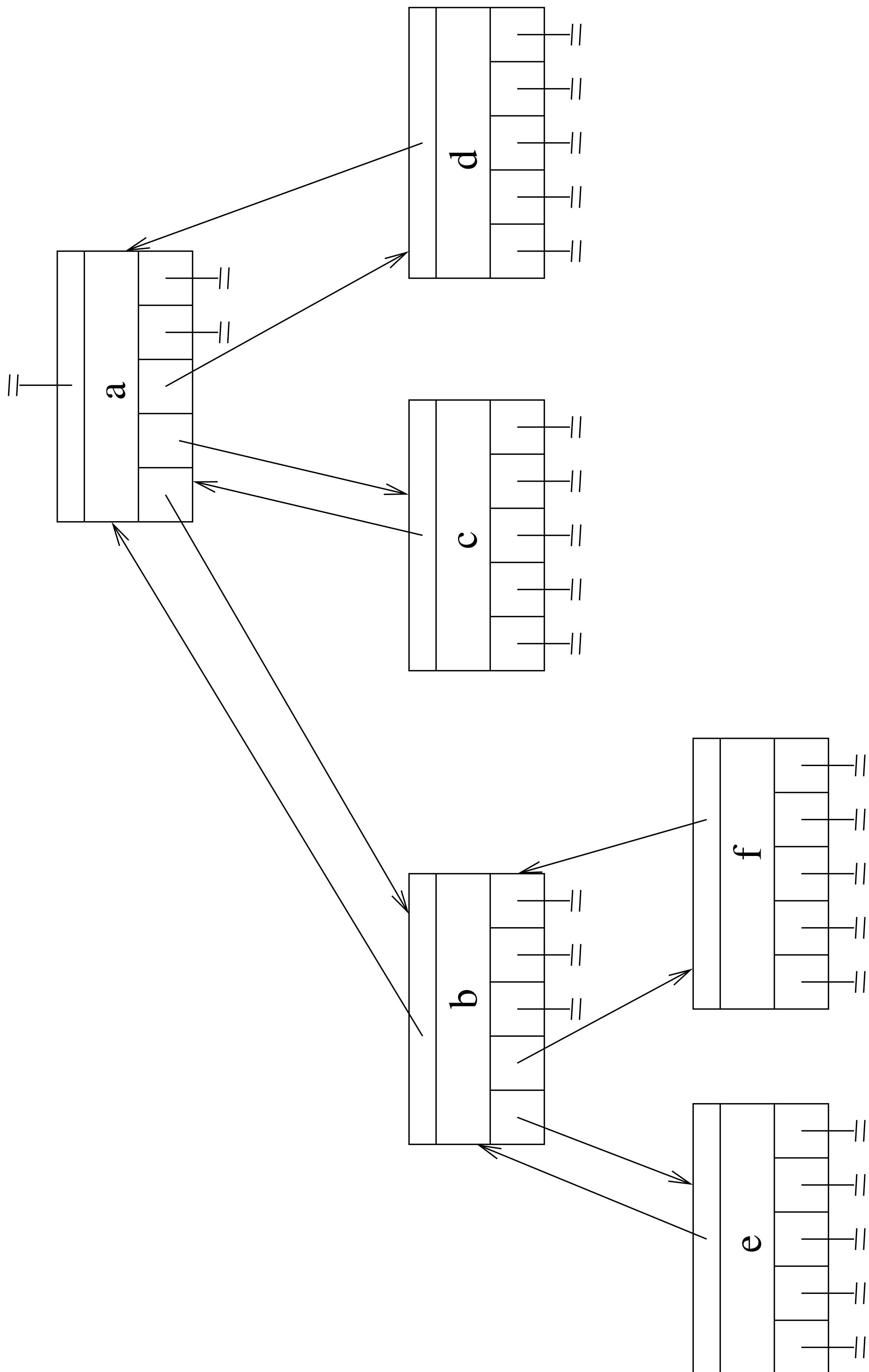
kuin kalvojen 3.5.3 lisäyksessäkin.

3.6 Yleisen puun talletus ja läpikäynti

- Jo kalvojen 3 alussa mainittiin, että puilla on monenlaista käyttöä tietojenkäsittelyssä.
- Kalvojen 3.5 B-puut osoittivat, että kaikki puut eivät suinkaan ole *binääri*puita.
- Miten siis voimme tallettaa yleisen puun?
- Jos tiedämme mikä on solmun suurin mahdollinen haarautumisaste, voimme tallettaa solmuun viitteet kaikkiin mahdollisiin lapsiin.

  Näin toimittiin juuri B-puissa; niissä maksimiaste oli $2 \cdot t$.
- Seuraavissa esimerkkikuvissa on
  - 3-haarainen puu
  - sen muistiesitys maksimiasteena 5.


a
b
c
d
e
f


d
a
c
f
b
e

- Rakenne tuhlaa paljon muistia tyhjiin viitekenttiin.

- Rakenne ei salli solmuja joilla on enemmän kuin 5 lasta.

- Parempi ratkaisu yleisen puun tallettamiseen saadaankin tallettamalla jokaiseen solmuun x seuraavat kentät:

  – key[x] = talletettu avain (kuten aiemminkin)

  – last[x] = onko x sisaruksistaan viimeinen eli sukupuujärjestyksessä nuorin

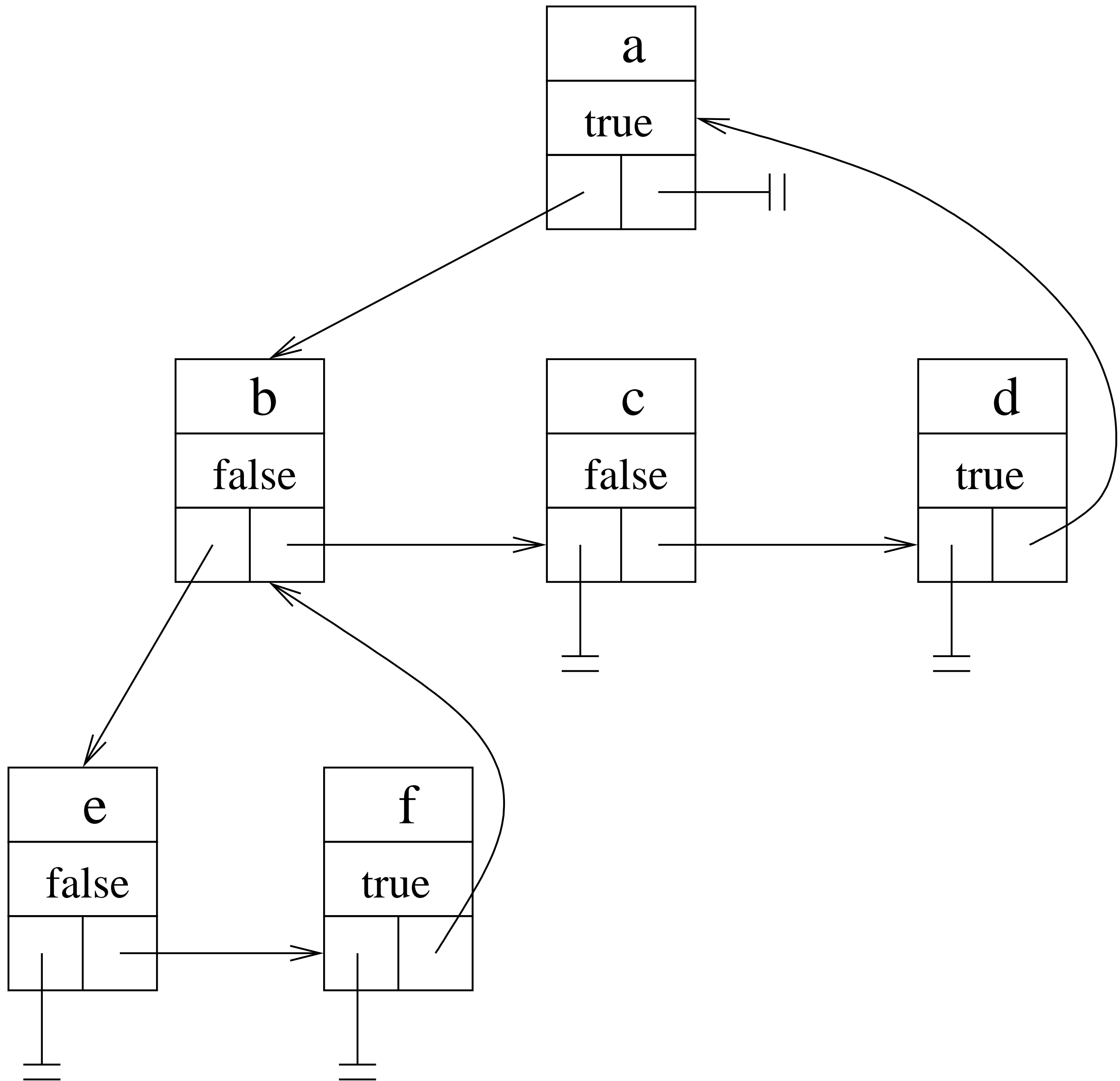
  – child[x] = viite solmun x ensimmäiseen lapseen

  – viite next[x] =

    seuraavaan (nuorempaan) sisarukseen kun last[x] = $\mathbf{false}$

    isään kun last[x] = $\mathbf{true}$.

- Edellinen esimerkkipuu näin talletettuna.


a
true
b
false
c
false
d
true
e
false
f
true

- Jokaisessa solmussa x on siis lista child[x] sen lapsista.
  - Muistia ei tuhlaudu turhiin viitekenttiin.
  - Puun haarautumisaste ei ole rajoitettu.
  - Solmu x osallistuu myös isänsä lapsilistaan

    eli omaan sisaruslistaansa

    viitteellä next[x].

- Solmusta x päästään sen isään etsimällä isäviite sisaruslistan lopusta:

parent(x)
    while not last[x] **do**
        $x \leftarrow$ next[x]
    return next[x]

  - Tässä toteutustavassa säästetään muistia mutta hidastetaan isään siirtymistä.
  - Vaihtoehtoisesti voidaan liittää jokaiseen solmuun x aikaisemmin käytetty oma isäviite $p[x]$.

- Solmun x lapsilista voidaan läpikäydä seuraavasti:

  Ensimmäinen lapsi y saadaan viitteenä

  firstchild(x)
  return child[x]

  Seuraavaan lapseen z siirrytään edellisestä lapsesta y operaatiolla

  nextchild(y)
  return if last[y] **then** NIL **else** next[y]

  Viimeinen lapsi u on se jolla nextchild(u) = NIL.

- Lauseessa 3.4 tulostimme binäärisen hakupuun T avaimet suuruusjärjestyksessä käsittelemällä se *sisä*järjestyksessä:

  1. vasen alipuu
  2. itse solmu
  3. oikea alipuu.

- Yleisten puiden tapauksessa luontevat käsittelyjärjestykset ovat

  esijärjestys:

  1. itse solmu
  2. solmun lapset listan järjestyksessä

  jälkijärjestys:

  1. solmun lapset listan järjestyksessä
  2. itse solmu

  kuten kalvoilla 3.1.2.

Esijärjestys on algoritmina

preOrderTreeWalk(x)
    käsittele key[x]
    $y \leftarrow$ firstchild(x)
    while $y \neq$ NIL **do**
        preOrderTreeWalk(y)
        $y \leftarrow$ nextchild(y)

- Alkukutsu

  preOrderTreeWalk(root[T])

  toimii vain kun root[T] $\neq$ NIL.

- Esimerkkipuumme solmut esijärjestyksessä lueteltuina:

$$a, b, e, f, c, d.$$

Jälkijärjestys on algoritmina

```
postOrderTreeWalk(x)
    y ← firstchild(x)
    while y ≠ NIL do
        postOrderTreeWalk(y)
        y ← nextchild(y)
    käsittele key[x]
```

- Alkukutsu

  postOrderTreeWalk(root[T])

  toimii vain kun root[T] $\neq$ NIL.

- Esimerkkipuumme solmut jälkijärjestyksessä lueteltuina:

$$e, f, b, c, d, a.$$

- Esi-, sisä- ja jälkijärjestys ovat puun *syvyys*suuntaisia läpikäyntejä:

  solmun lapset käsitellään ennen sen sisaruksia.

Leveyssuuntainen läpikäynti toimii toisin.

- Puu käydään läpi *taso tasolta*

  1. juuri

  2. juuren lapset

  3. juuren lapsenlapset

  4. ...

  ja tason sisällä vasemmalta oikealle.

- Puumme solmut leveyssuuntaisesti:

$$a, b, c, d, e, f.$$

- Pidetään yllä keskeneräisten töiden muistilistaa Q:
  - Sisältönä ovat ne solmut jotka on jo nähty mutta ei vielä käsitelty.
  - Seuraavaksi otetaan käsittelyyn listan ensimmäinen solmu.
  - Käsiteltävän solmun lapset viedään listan loppuun.
  - Siis muistilistaksi käy kalvojen 2.2 jono.
- Algoritmina

$Q \leftarrow$ aluksi tyhjä solmujono
enqueue(Q, root[T])
while not isempty(Q) **do**
    $x \leftarrow$ dequeue(Q)
    käsittele key[x]
    $y \leftarrow$ firstchild(x)
    while $y \neq$ NIL **do**
        enqueue(Q, y)
        $y \leftarrow$ nextchild(y)

- Jälleen oletamme että root[T] $\neq$ NIL.

3.7 Puut ongelmanratkaisussa

- Tutustutaan puiden käyttöön
  - ongelmanratkaisun
  - algoritmikehityksen

  apuneuvoina.

- Yksi puiden tärkeistä käyttötavoista on
  - ongelmanratkaisussa tapahtuvan laskennan etenemisen kuvaaminen
  - jotta voimme olla varmoja, että ratkaisu todellakin tutkii kaikki mahdolliset vaihtoehdot.

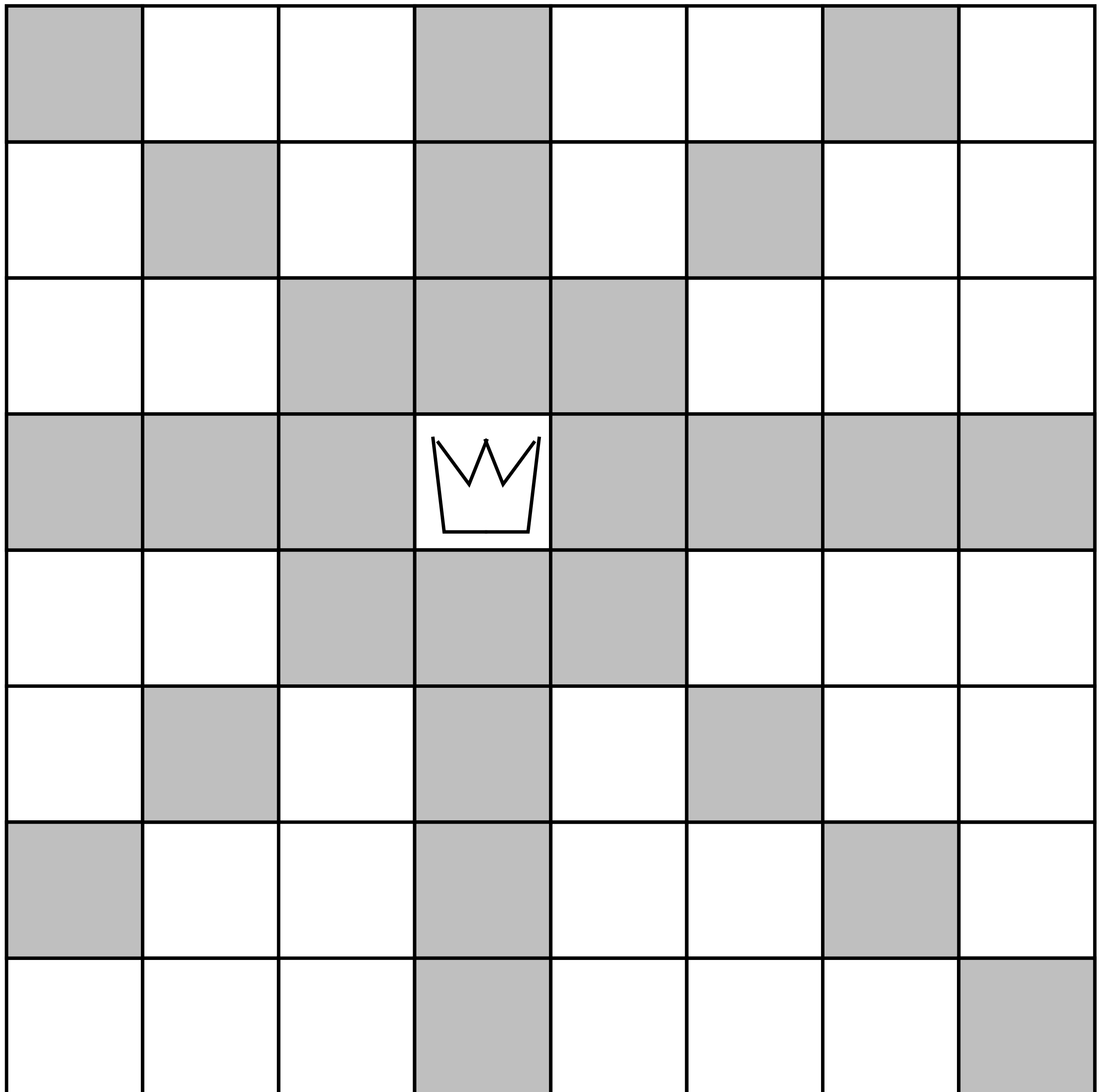
3.7.1 Kahdeksan kuningattaren ongelma

- Miten voimme sijoittaa shakkilaudalle 8 kuningatarta siten, että ne eivät uhkaa toisiaan?

- Kuningatar uhkaa samalla rivillä, sarakkeella sekä diagonaalilla olevia ruutuja.

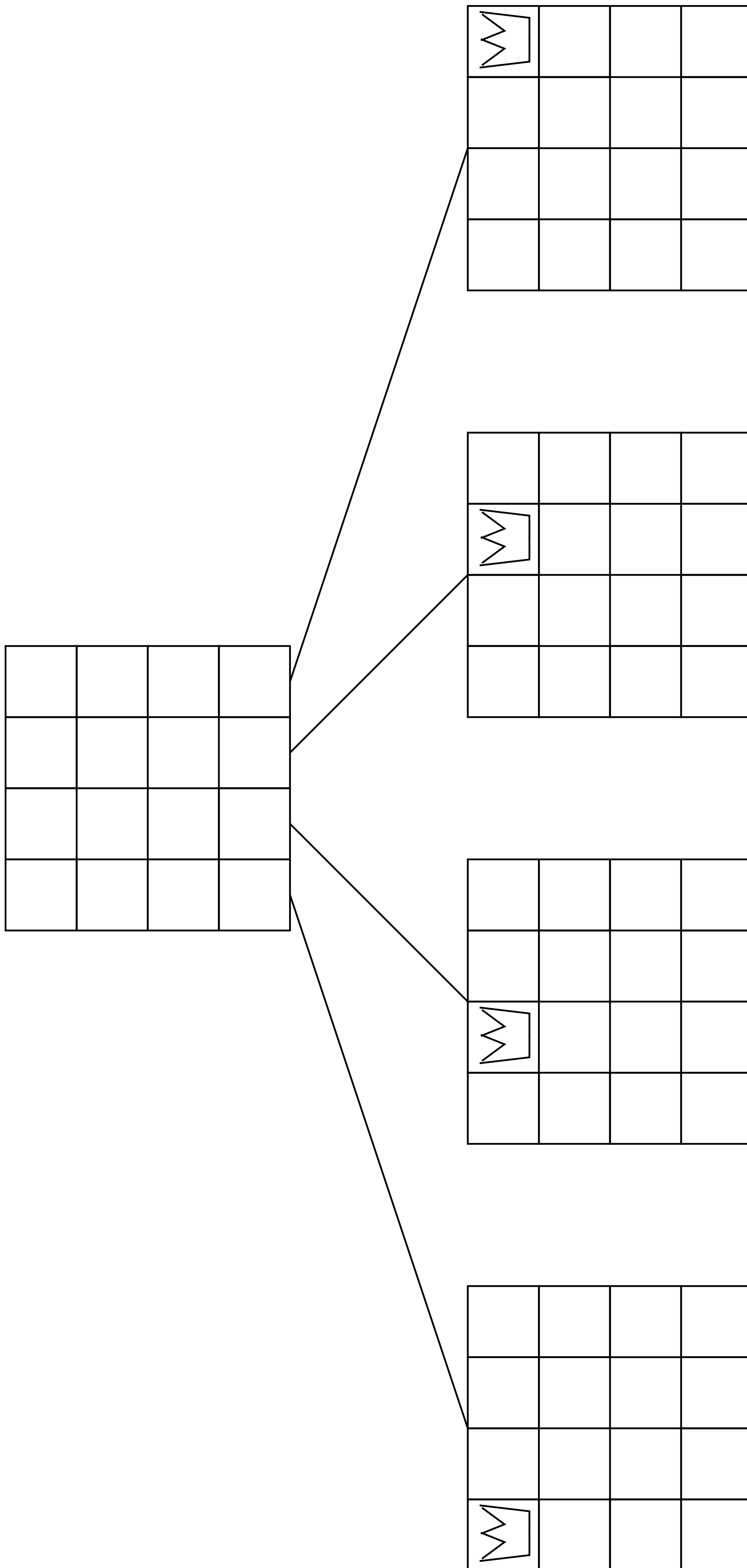
  Seuraavan esimerkkikuvan varjostettuja ruutuja.

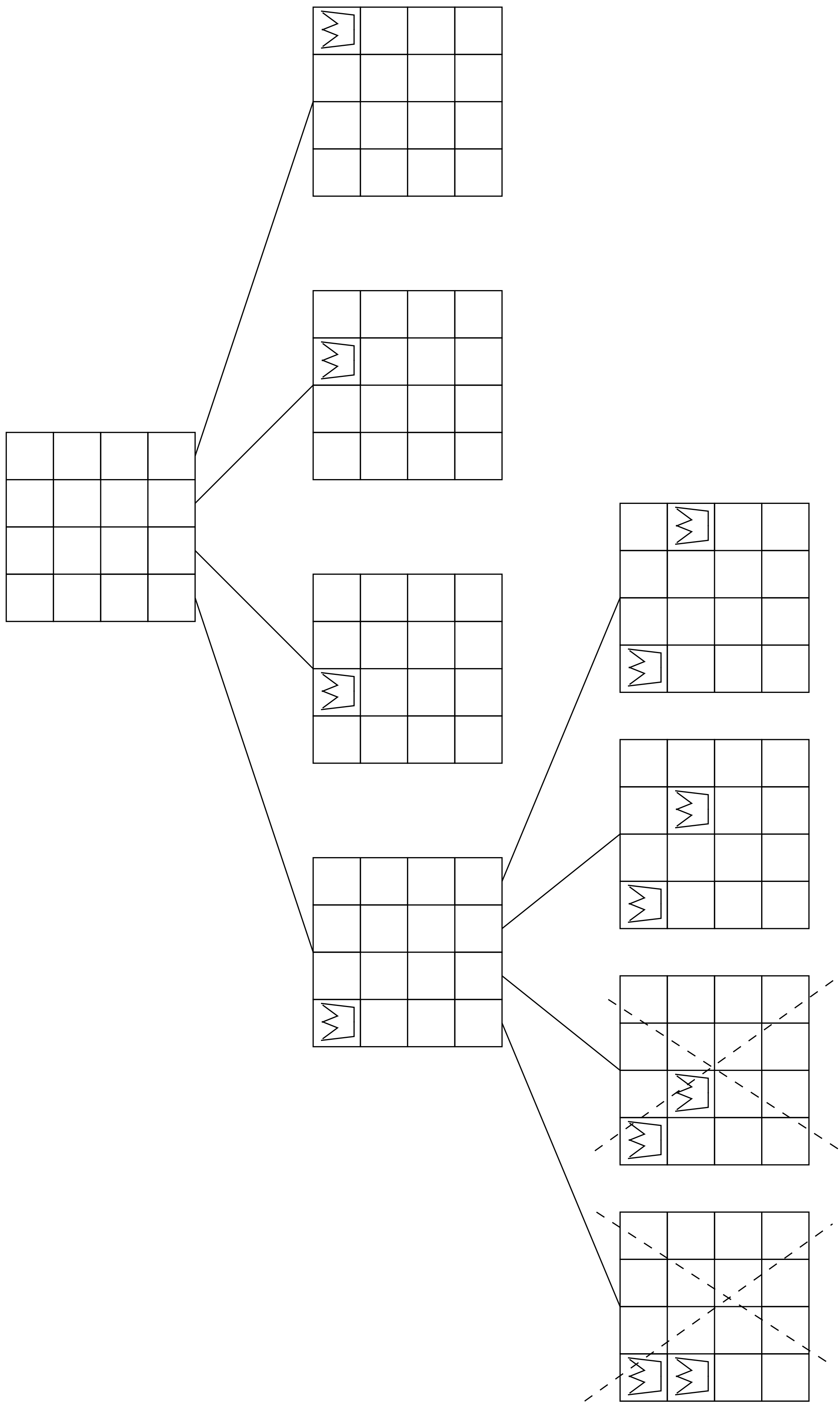
- Yleistetty versio ongelmasta: miten saamme sijoitettua n kuningatarta $n \times n$-kokoiselle shakkilaudalle?

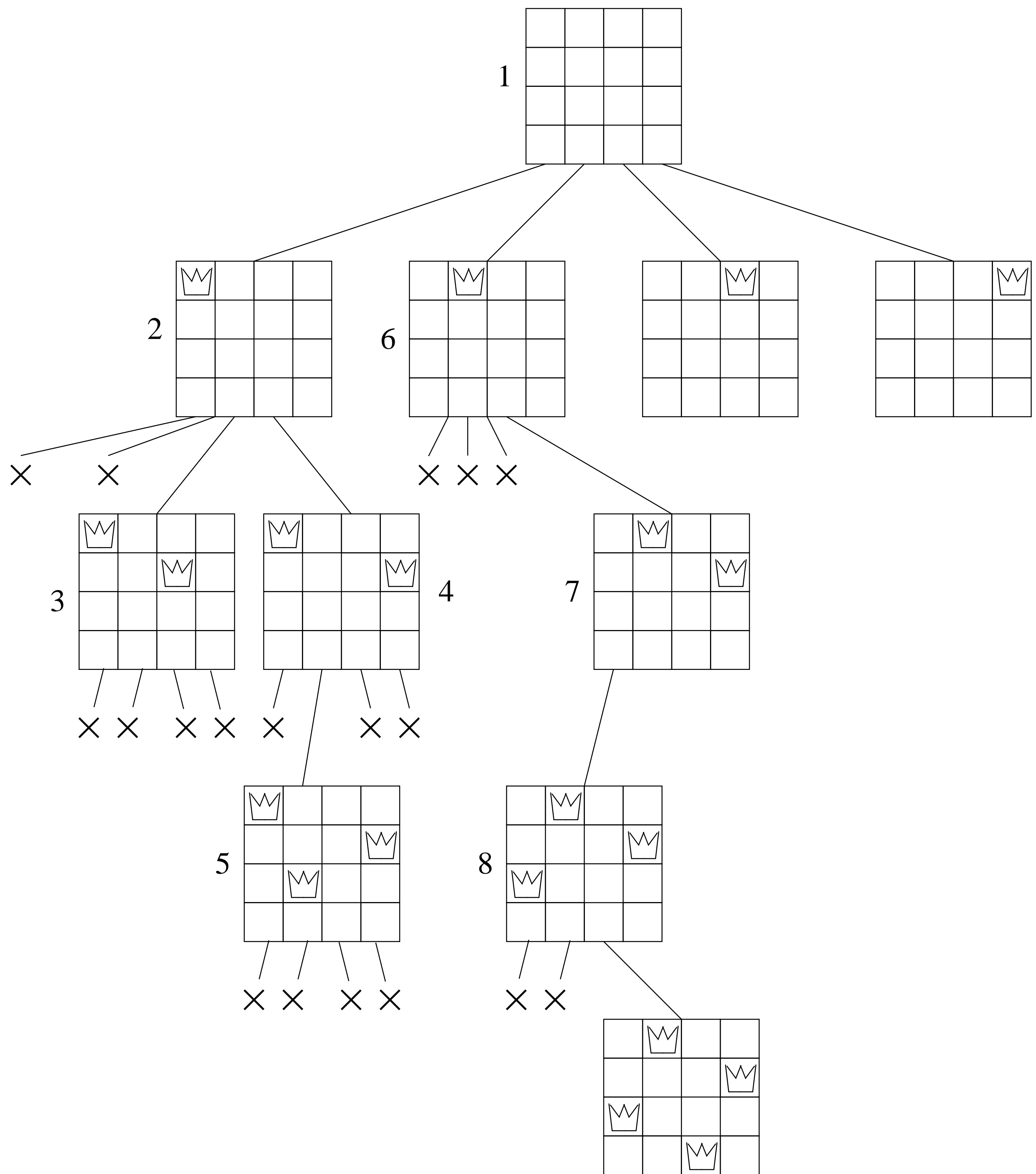


- Etsitään oikea kuningatarasetelma systemaattisesti kun $n = 4$:
  - Aloitetaan tyhjältä laudalta.
  - jokainen kuningatar on omalla rivillään, jokainen rivi lopulta käytössä.
  - Asetetaan rivin 1 kuningatar — 4 eri mahdollisuutta.
  - Seuraavaksi asetetaan rivin 2 kuningatarta:
    - Yritetään sarakkeita 1,2,3,4 vasemmalta oikealle.
    - Huomaamme: olemme muodostamassa puuta, joka kuvaa erilaiset ratkaisumahdollisuudet.
    - Kaksi vasemmanpuoleisinta yritystä ovat jo nyt tuhoon tuomittuja, eikä niitä siis kannata enää jatkaa.
  - Jatketaan puun piirtämistä riveille 3,4 kunnes löytyy ratkaisu.

	1	2	3	4	
1					← kuningatar 1
2					← kuningatar 2
3					← kuningatar 3
4					← kuningatar 4






1
2
6
3
4
7
5
8

- Kuvassa puun solmut on numeroitu kalvojen 3.6 *esijärjestyksessä*
  - eli puu on ajateltu piirretyksi esijärjestyksessä
  - ja yhdeksäntenä piirretty solmu on siis ratkaisua vastaava pelitilanne.

- Kun laudan koko n kasvaa, tulee puusta varsin suuri.

- Kuitenkaan *koko puun ei tarvitse olla talletettuna* muistiin samalla kertaa:

  riittää että muistissa on ainoastaan reitti juuresta parhaillaan tutkittavaan solmuun.

- Voimme etsiä ratkaisun n kuningattaren ongelmaan suorittamalla ratkaisupuun läpikäynnin esijärjestyksessä ilman että ratkaisupuuta on missään vaiheessa olemassa!

- Ratkaisupuuta käytettiin *algoritmin suunnittelussa*.

- Yhtä pelilautaa esittää $n \times n$ alkion totuusarvotaulukko table missä

  table$[x][y] =$ onko rivin x sarakkeessa y kuningatar vaiko ei.

- Oletetaan että käytössä on apufunktio check(table) =

  onko taulukon table kuvaama pelilauta yhä täydennettävissä ratkaisuksi vaiko ei?

  – Jos taulukossa table on jo kaikki n kuningatarta, niin apufunktio vastaa onko se ratkaisu vaiko ei.

  – Apufunktio saa vastata **false** vasta silloin, kun on *varmaa*, ettei taulukosta table voi mitenkään enää saada ratkaisua.

  – Ratkaisu löytyy sitä nopeammin, mitä aikaisemmin apufunktio uskaltaa vastata **false**.

- Aluksi

  1. laitetaan $n \times n$ taulukon table jokaisen ruudun arvoksi $\mathbf{false}$

     eli aloitetaan tyhjältä pelilaudalta

  2. kutsutaan rekursiivista ratkaisunhakufunktiota putqueen(table, 1).

- Tämä hakufunktio on

```
putqueen(table, r)
1  if not check(table) then return false
2  if r = n + 1 then
3       print(table)
4       return true
5  for s ← 1 to n do
6       table2 ← table ▷ laudasta uusi kopio
7       table2[r][s] ← true
8       if putqueen(table2, r + 1) then
9            return true
10 return false
```

- Karsitaan pois sellainen pelilauta table jota ei voi laajentaa ratkaisuksi (rivillä 1).

- Jos lautaa ei tarvitse enää laajentaa, niin

  - tulostetaan se vastauksena

  - ilmoitetaan kutsujalle ”kyllä löytyi”

  (riveillä 2–4).

- Muuten s käy läpi kaikki sarakkeet (rivillä 5).

- Yritetään sijoittaa nykyisen rivin r kuningatar sarakkeelle s (riveillä 6–7).

- Tarkistetaan rekursiivisesti (rivillä 8) voiko näin täydennetyn laudan table2 ratkaista.

- Jos tällä sarakkeella s voi, niin vastataan heti ”kyllä voi” (rivillä 9).

- Jos millään sarakkeella s ei voinut, niin vastataan lopuksi ”ei voi” (rivillä 10).

- Algoritmi käy läpi puun T, joka ei ole missään vaiheessa rakennettuna muistiin.

  Tällaista puuta sanotaan *implisiittiseksi puuksi*.

- Jos puu T olisi kokonaan muistissa, olisi sen koko valtava:

  $$\mathcal{O}\left(1 + n + n^2 + n^3 + \cdots + n^n\right)$$

  solmua.

- Nyt muistissa onkin vain polku puun T juuresta nykyiseen solmuun.

  Tilavaativuus pysyy kohtuullisena

  $$\mathcal{O}\left(n \cdot m\right)$$

  missä

  $$\begin{aligned} n - 1 &= \text{polun maksimipituus} \\ m &= \text{yhden pelilaudan vaatima tila} \\ &= \mathcal{O}\left(n^2\right) \text{ ratkaisussamme.} \end{aligned}$$

- Aikavaativuus sen sijaan on suuri:

  vaikka kaikkia solmuja ei tarvitsekaan käydä läpi

  silti läpikäytävien solmujen määrä voi kasvaa eksponentiaalisesti ongelman koon n suhteen.

- Käytännön ajantarpeen voi (noin) puolittaa huomaamalla vasen/oikea-symmetrian:

  ensimmäisestä rivistä riittää tutkia vain sen vasen puolisko.

- Käyttämästämme ongelmanratkaisutekniikasta käytetään nimitystä *branch-and-bound:*

  Haaraudutaan ratkaisua etsittäessä yhä syvemmälle eri vaihtoehtojen puussa.

  Rajataan etsintää leikkaamalla haara pois kun se osoittautuu mahdottomaksi.

3.7.2 Permutaatiot

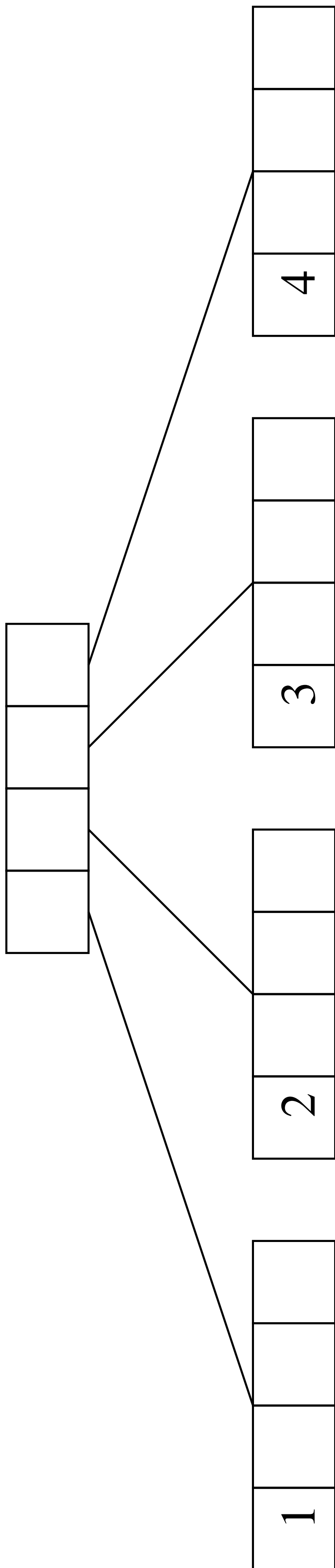
- Kalvojen 3.7.1 ratkaisustrategiaa voimme käyttää myös generoimaan lukujen $1, 2, 3, \ldots, n$ kaikki *permutaatiot*.

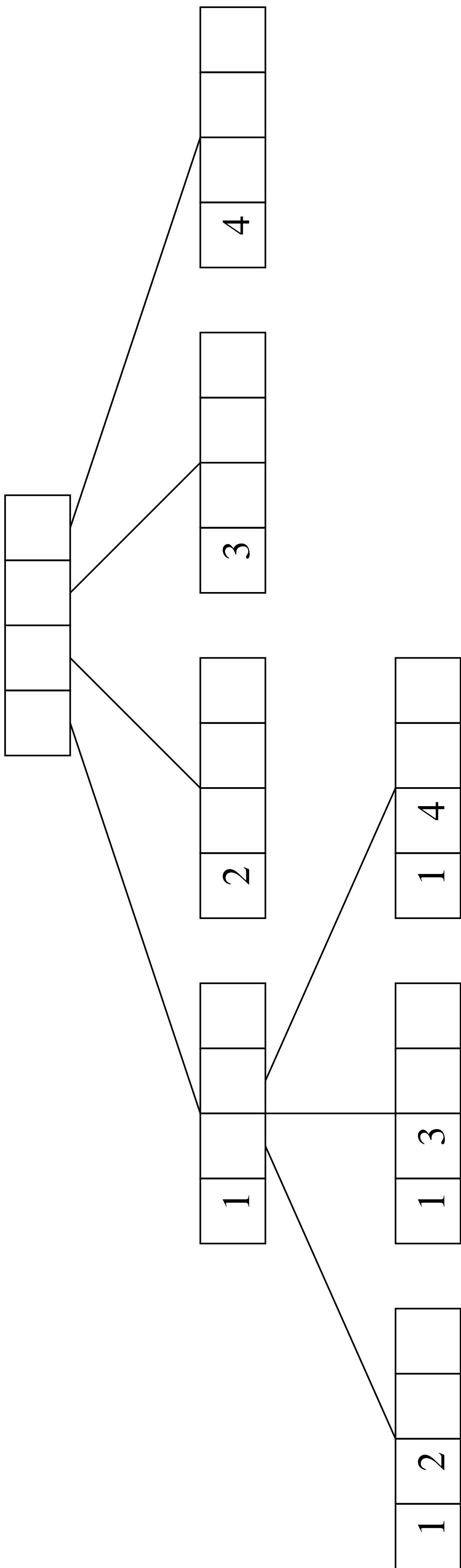
- Tarkastellaan permutaatioita kun $n = 4$:
  - Permutaation ensimmäinen luku voi mikä tahansa luvuista $1, 2, 3, 4$.
  - vasen haara jatkuisi siten että seuraava numero voi olla joku joukosta $2, 3, 4$:

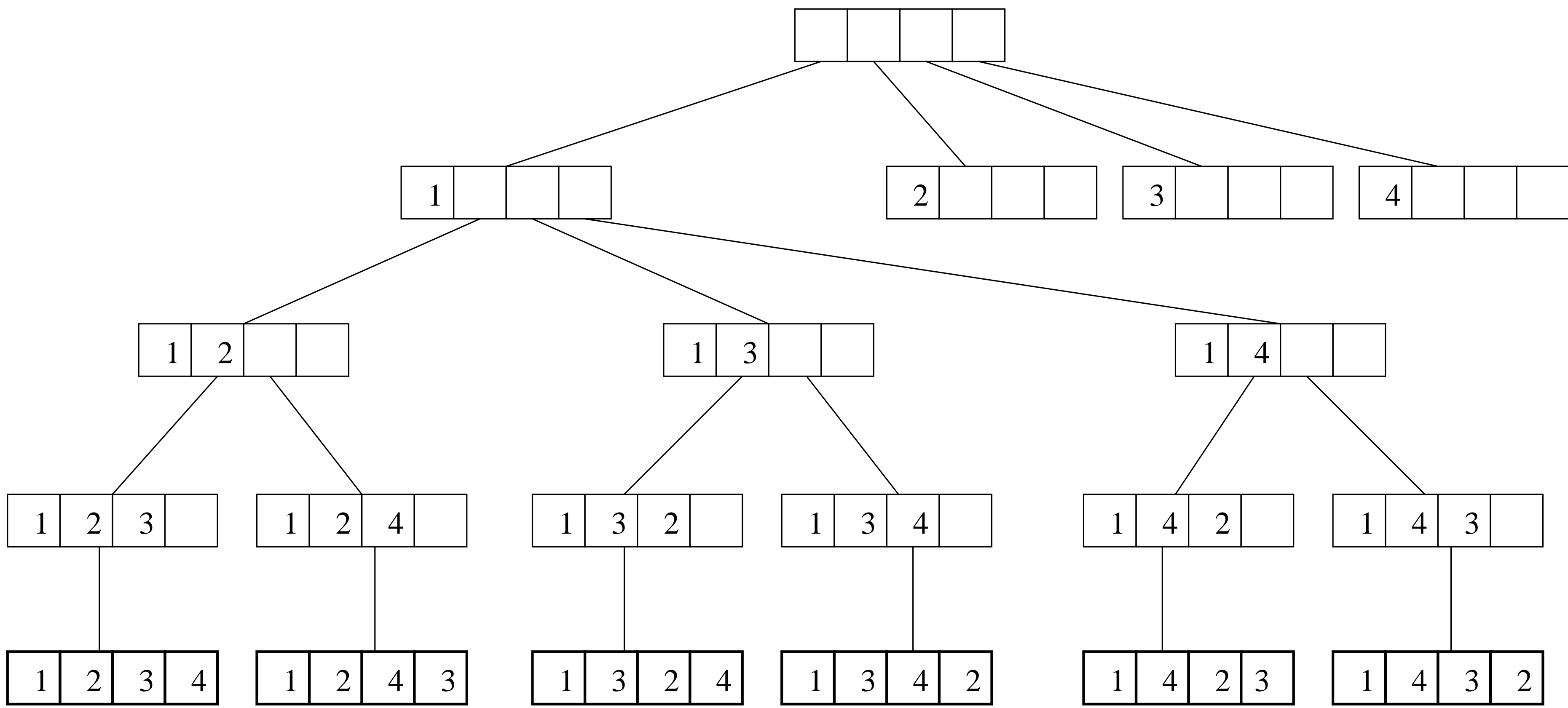
    luku 1 on jo käytetty, sillä se aloittaa permutaation.
  - Piirretään permutaatiopuuta pitemmälle.

- Valmiit permutaatiot löytyvät siis puun lehdistä.

- Jos lehdet generoidaan esijärjestyksessä, niin permutaatiot saadaan sanakirjajärjestyksessä.


1
2
3
4


4
3
2
1
1 4
1 3
1 2


1
2
3
4
1 2
1 3
1 4
1 2 3
1 2 4
1 3 2
1 3 4
1 4 2
1 4 3
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2

- Algoritmi permutaatioiden generoimiseen:
  - Käytetään totuusarvotaulukkoa used$[1 \dots n]$.
  - used$[i]$ = onko lukua i jo käytetty permutaatiossa.
  - Alkuarvona siis used$[i] = \mathbf{false}$ kaikilla indekseillä i.
  - Käytetään myös taulukkoa table$[1 \dots n]$.
  - Permutaation j:s luku on table$[j]$.
  - Kutsutaan generate(table, used, 1) missä

```
generate(table, used, k)
1   if k = n + 1 then print(table)
2   else for i ← 1 to n do
3           if not used[i] then
3               used2 ← used
4               used2[i] ← true
5               table2 ← table
6               table2[k] ← i
7               generate(table2, used2, k + 1)
```

- Operaation toimintaidea:
  - Rivillä 1 tarkistetaan onko permutaatio jo valmis; jos on niin tulostetaan se.

    Parametri k ilmaisee, monettako lukua ollaan nyt lisäämässä.
  - Jos ei vielä ole, niin jatketaan permutaatiota kaikilla luvuilla i joita vielä ole käytetty (rivit 2–3).
  - Vuorollaan kukin käyttämätön luku i
    1. merkitään käytetyksi (rivit 3–4)
    2. liitetään permutaatioon (rivit 5–6)
    3. laajennetaan valmiiksi rekursiivisesti (rivi 7).
- Erona n kuningattaren ongelmaan on siis tällä kertaa se, että puun generoimista ei lopeteta missään vaiheessa, sillä haluamme tulostaa *kaikki* permutaatiot.
- Kyseessä on siis branch-and-bound ilman bound-mahdollisuutta.

Tilavaativuus on yhä varsin kohtuullinen

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

sillä yhden rekursiotason viemä tila on

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

ja rekursiotasoja on n kpl.

Aikavaativuus on

$$\mathcal{O}\left(1+n+n^2+\ldots+n^n\right)=\mathcal{O}\left(n^{n+1}\right).$$

3.7.3 Kauppamatkustajan ongelma

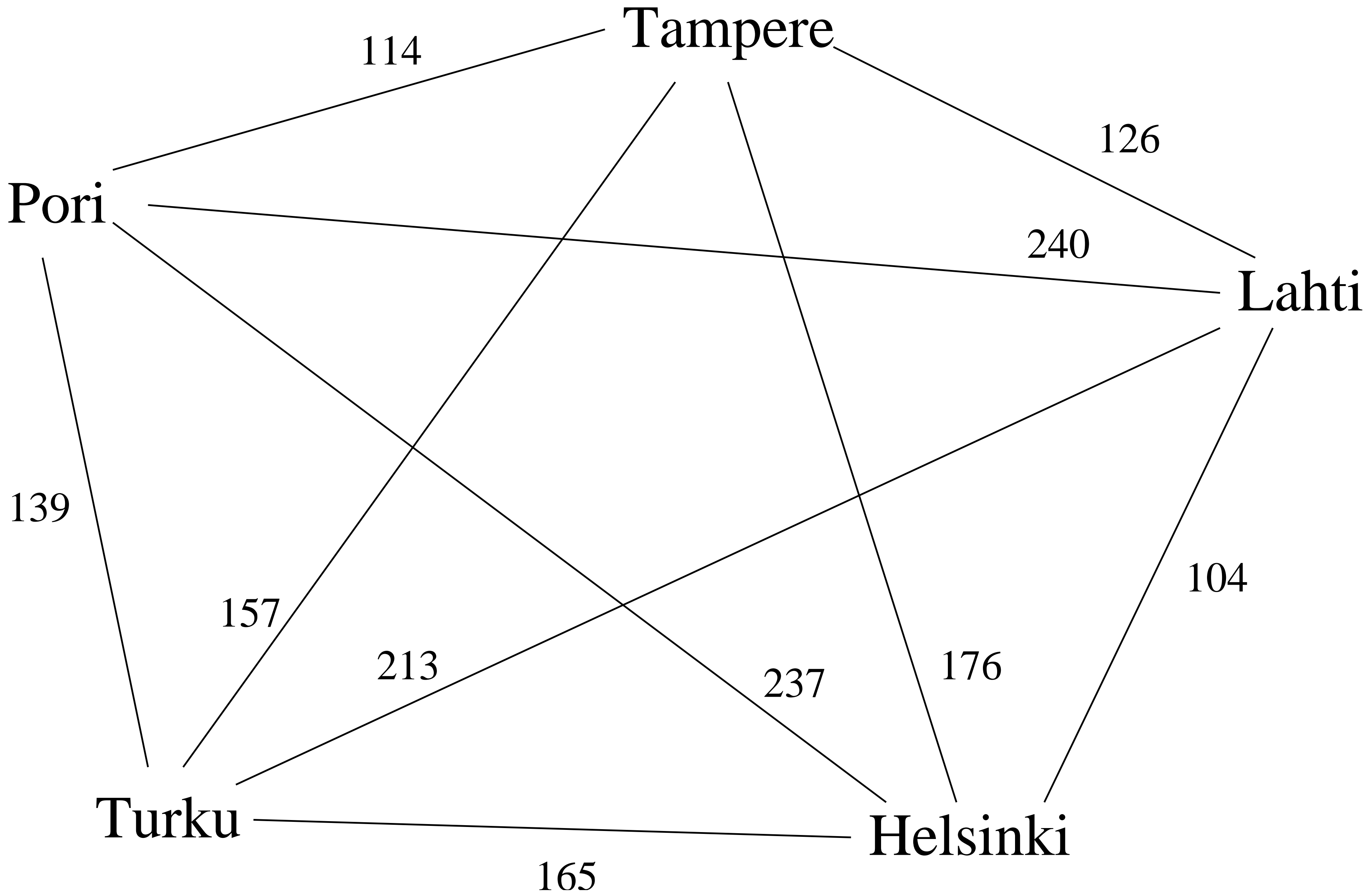
- Helsingissä asuvan kauppamatkustajan täytyy vierailla Lahdessa, Turussa, Porissa ja Tampereella.

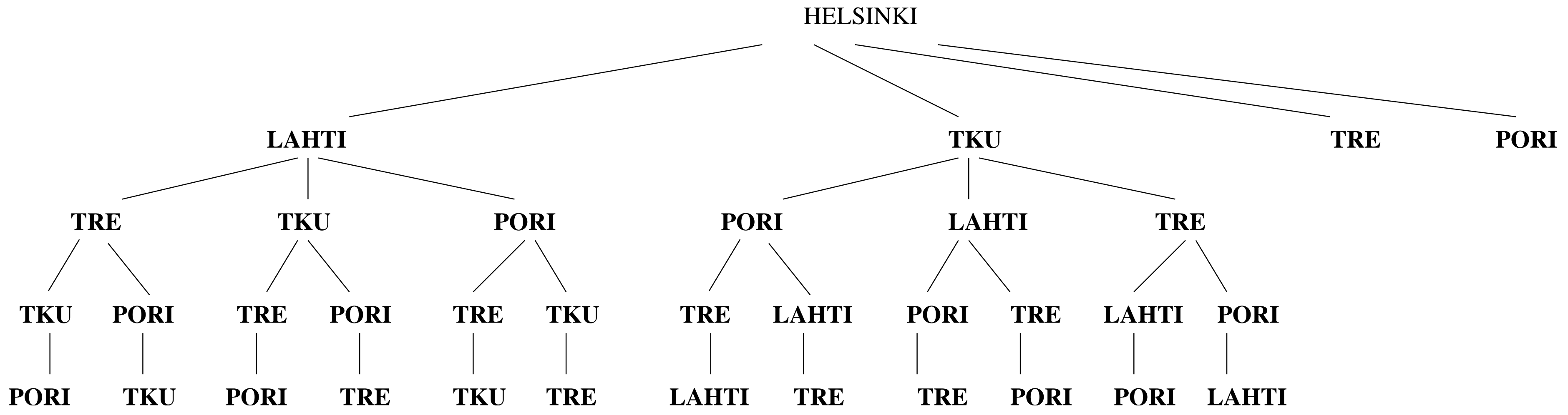
- Matkakulujen minimointi bisneksessä on tärkeää: mikä on lyhin reitti joka
  - alkaa Helsingistä
  - päättyy Helsinkiin
  - sisältää yhden vierailun kussakin kaupungissa?

- Huomaamme että mahdolliset reitit ovat kaupunkijonon Turku, Tampere, Pori, Lahti permutaatiot.

- Voimme siis käyttää ratkaisussa samaa periaatetta kuin permutaatioiden tulostuksessa kalvoilla 3.7.2:

  näin siis saamme systemaattisesti generoitua kaikki mahdolliset reitit.


Tampere
114
126
Pori
240
Lahti
139
157
213
237
176
104
Turku
Helsinki
165


HELSINKI
LAHTI
TKU
TRE
PORI
TRE
TKU
PORI
PORI
LAHTI
TRE
TKU
PORI
TRE
PORI
TRE
TKU
TRE
LAHTI
PORI
TRE
LAHTI
PORI
PORI
TKU
PORI
TRE
TKU
TRE
LAHTI
TRE
TRE
PORI
PORI
LAHTI

- Reitin pituus kannattaa laskea heti generoinnin yhteydessä:
  - Palatessamme etsintäpuuta ylöspäin muistamme mikä oli parhaan kyseistä kautta kulkevan reitin pituus.
  - Uutta reittiä etsittäessä nykyinen haara voidaan karsia, jos se kasvaa ainakin yhtä pitkäksi kuin paras aiemmin löydetty reitti — branch and bound, jälleen.
  - Kuten kuvassa keskeneräinen ratkaisu

    $$\text{Helsinki} \xrightarrow{237} \text{Pori} \xrightarrow{240} \text{Lahti} \xrightarrow{213} \text{Turku}$$

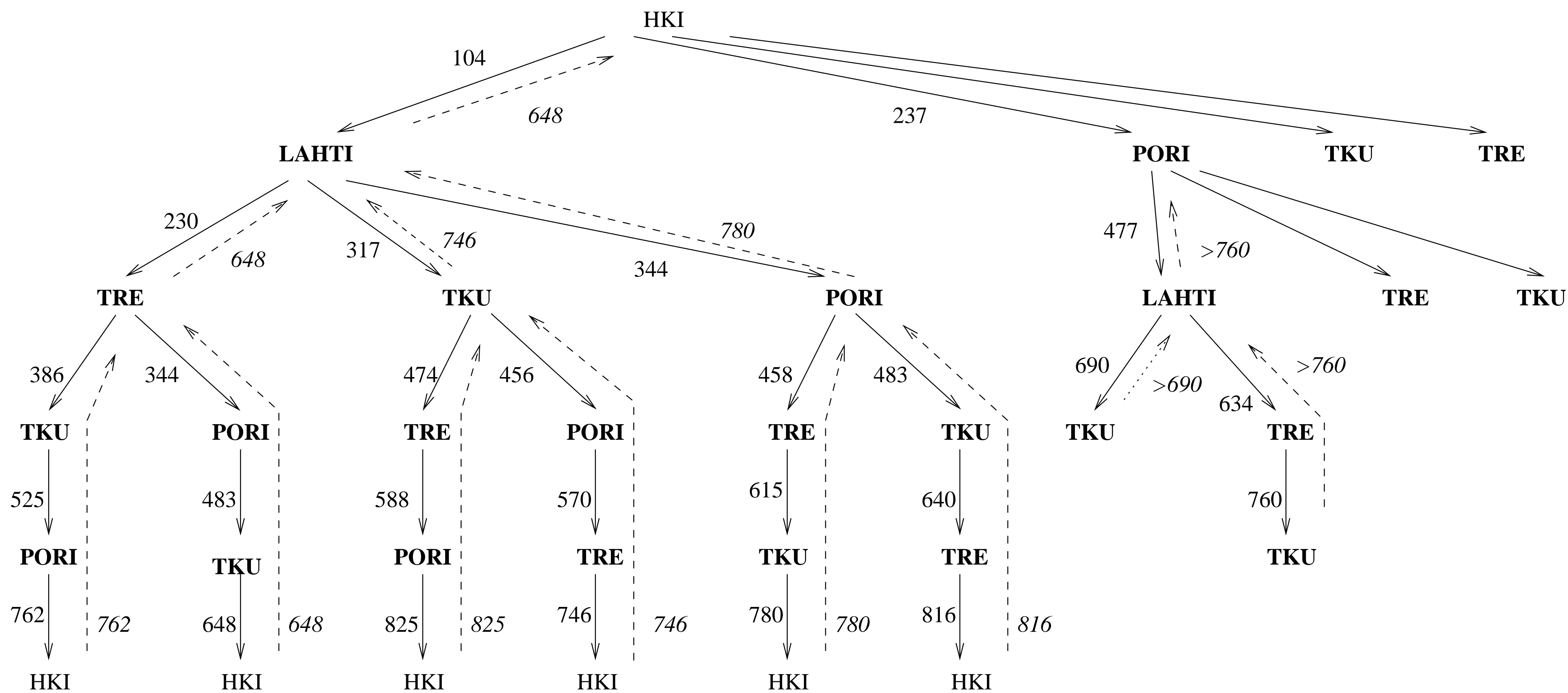
    on jo nyt pitempi (690 km) kuin paras siihen mennessä löydetty valmis ratkaisu

    $$\text{Helsinki} \xrightarrow{104} \text{Lahti} \xrightarrow{126} \text{Tampere} \xrightarrow{114} \text{Pori} \xrightarrow{139} \text{Turku} \xrightarrow{165} \text{Helsinki}$$

    (648 km).

- Koko etsintäpuun läpikäytyämme saamme tietoon lyhyimmän reitin pituuden.

- Samalla toki kannattaa merkitä muistiin minkä kaupunkien kautta reitti kulkee.


HKI
104
648
237
LAHTI
PORI
TKU
TRE
230
648
317
746
780
344
477
>760
TRE
TKU
PORI
LAHTI
TRE
TKU
386
344
474
456
458
483
690
>690
634
>760
TKU
PORI
TRE
PORI
TRE
TKU
TKU
TRE
525
483
588
570
615
640
760
PORI
TKU
PORI
TRE
TKU
TRE
TKU
762
762
648
648
825
825
746
746
780
780
816
816
HKI
HKI
HKI
HKI
HKI
HKI

- Algoritmihahmotelma:
  - Numeroidaan kaupungit $1, 2, 3, \ldots, n$.

    Esimerkissämme $n = 5$.
  - Lähtökaupunki on 1.

    Esimerkissämme Helsinki = 1.
  - Välimatkataulukko dist$[1 \ldots n][1 \ldots n]$:

    dist$[p][q]$ = dist$[q][p]$ = kaupunkien p ja q välinen lyhyin etäisyys (> 0).
  - Totuusarvotaulukko visited$[1 \ldots n]$ kertoo missä kaupungeissa on jo vierailtu tutkittavalla polulla.
  - Alustus on siis visited$[i] = \mathbf{false}$ jokaiselle indeksille i.
  - Rekursiivinen reitinhaku alkaa kutsulla

    $$\text{gen}(+\infty, \text{visited}, 1, 1)$$

    jonka tuloksena saadaan lyhyimmän reitin pituus.

```
gen(best, length, visited, cur, k)
1  if k = n then return length + dist[cur][1]
2  mybest ← +∞
3  for i ← 2 to n do
4      if not visited[i] then
5          vis2 ← visited
6          vis2[i] ← true
7          if length + dist[cur][i] < best then
8              newp ← gen(best,
                          length + dist[cur][i],
                          vis2, i, k + 1)
9              if newp < mybest then
10                 mybest ← newp
11             if newp < best then
12                 best ← newp
13 return mybest
```

best = parhaan jo löydetyn reitin pituus.

- Käytetään rivillä 7 karsimaan hakua.
- Alkuarvo saadaan parametrina: paras niistä reiteistä, jotka eivät laajenna nykyistä reittiä.
- Pidetään ajan tasalla riveillä 11–12.

newp = nykyisen reitin nykyinen laajennus (rivit 3–6 ja 8).

cur = nykyisen reitin nykyinen kaupunki.

Päivitetään rivin 8 rekursiokutsussa vastaamaan nykyistä laajennusta.

mybest = paras nykyistä reittiä laajentamalla saatu ratkaisu.

- Alustetaan rivillä 2, kun yhtään laajennusta ei vielä ole kokeiltu.
- Pidetään ajan tasalla 9–10.

$k =$ kuinka monessa kaupungissa on jo käyty.

- Jos kaikki kaupungit on käyty (rivi 1), niin suljetaan rengas palaamalla lähtökaupunkiin.
- Muuten kasvatetaan rivin 8 rekursiokutsussa.

length = kuinka pitkä reitin tähän asti tutkittu osa on.

Nykyisen reitin pidennykset lasketaan riveillä 1, 7 ja 8.

Tilavaativuus kohtuullinen

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

olettaen että yksi rekursiotaso voidaan tallettaa tilassa $\mathcal{O}(m)$.

Aikavaativuus pahimmillaan *eksponentiaalinen* kaupunkien lukumäärän n suhteen.

- Ongelmalle ei tiedetä alle eksponentiaalista ratkaisualgoritmia.
- Itse asiassa on vahvoja syitä epäillä, että tehokasta eli *polynomista ei liene* olemassakaan...
- ...mutta sitova todistus puuttuu yhä.
- Kyseessä on ns. NP-täydellinen ongelma, aiheesta enemmän kurssilla *Laskennan vaativuus*.

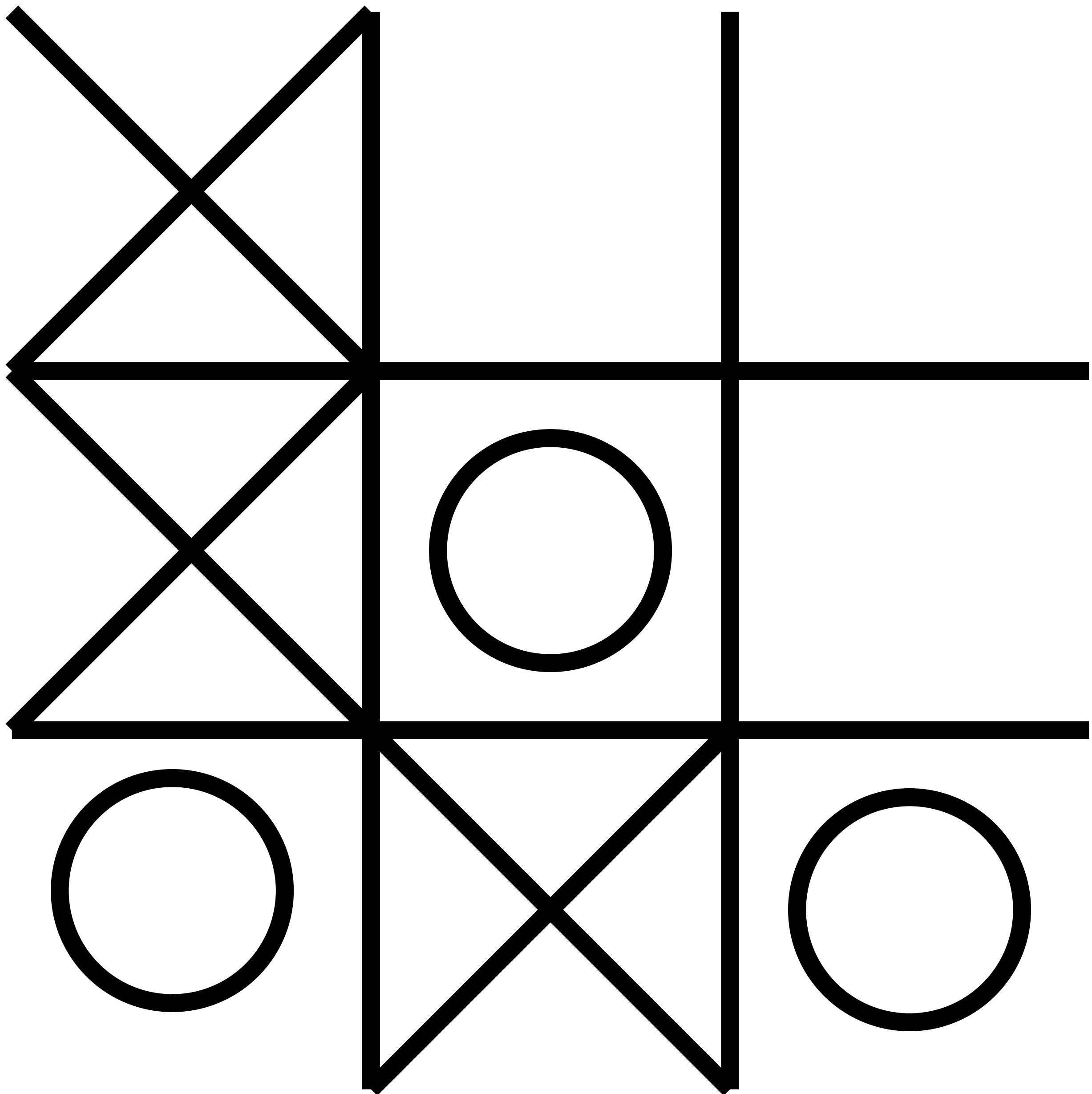
3.8 Pelipuu

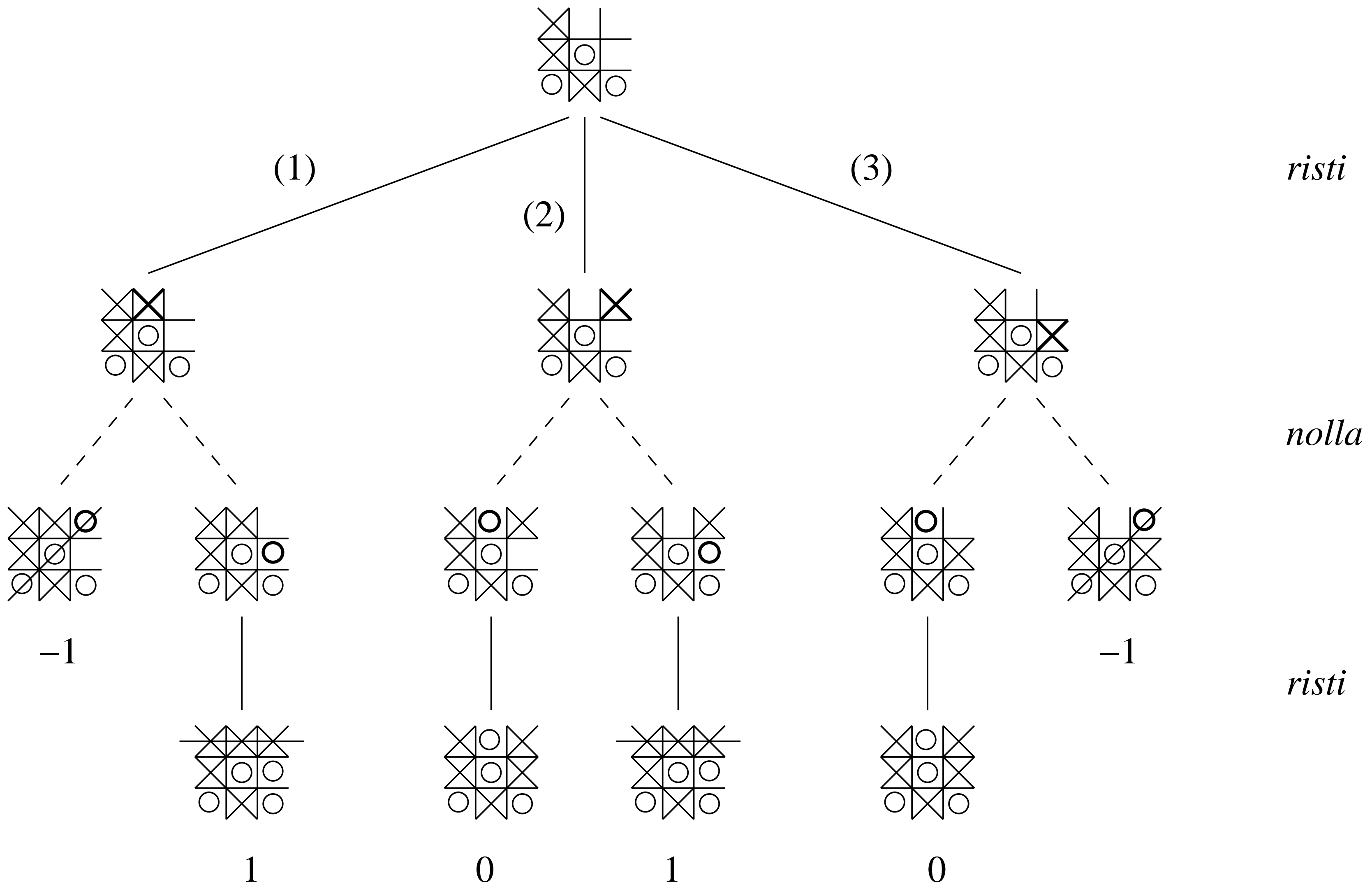
- Tietokone pelaa risti-nollaa ihmistä vastaan 3×3 ruudukossa.
- On ristin vuoro, mitä tietokoneen kannattaa tehdä seuraavassa tilanteessa?
- Tietokone rakentaa päätöksensä tueksi *pelipuun*.
- On tehtävä siis valinta 3 mahdollisen siirron suhteen.
- Pelipuuhun on kirjattu auki kaikki

  ristin (tietokoneen) mahdolliset siirrot nyt

  nollan (ihmisvastustajan) mahdolliset vastasiirrot sen jälkeen

  loppupelit näistä siirtopareista eteenpäin.




(1)
(2)
(3)
risti
nolla
risti
–1
–1
1
0
1
0

- Pelin lopputilanteet = pelipuun loppusolmut.

- Jokaiseen loppusolmuun on merkattu tilanteen arvo ristin kannalta:

$$
\begin{aligned}
+1 &= \text{voitto} \\
\pm 0 &= \text{tasapeli} \\
-1 &= \text{tappio.}
\end{aligned}
$$

- Siis minkä siirron tietokone valitsee?

  (1) johtaa lopulta jomman kumman voittoon

  (tasapeli on mahdoton)

  (2) johtaa joko tasapeliin tai ristin voittoon

  (nolla ei voi voittaa)

  (3) johtaa joko tasapeliin tai nollan voittoon

  (risti ei voi voittaa).

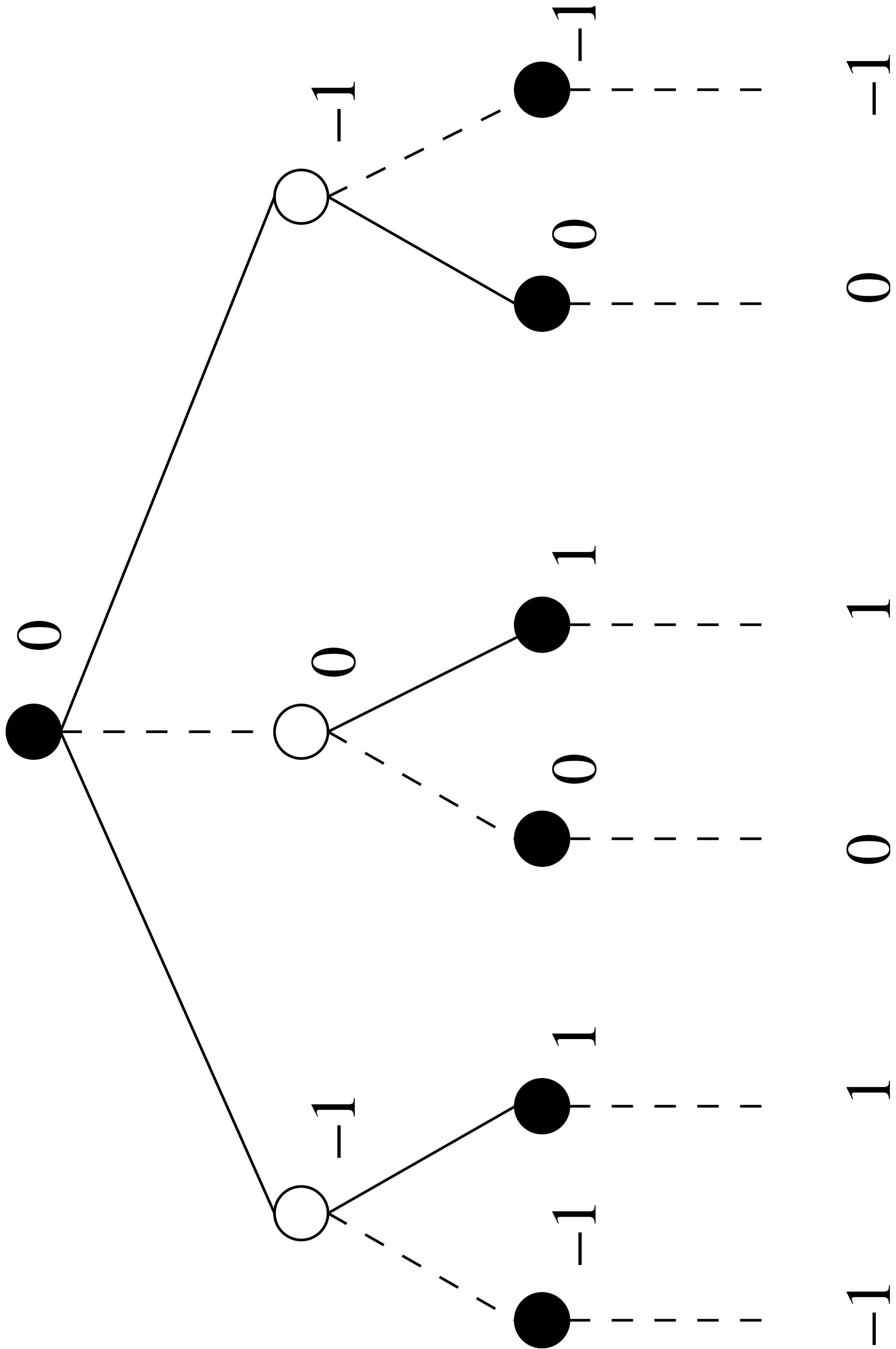
- Järkevintä on valita **(2)**:
  - on yhä mahdollista voittaa

    (jos nolla pelaa kehnosti)
  - mutta ei ainakaan voida enää hävitä

    (vaikka nolla pelaisikin vaikka kuinka hyvin).

- Strategiana on valita parhaan arvon (voitto $+1$, tasapeli ± 0 tai tappio -1) tuottava haara
  - kun *oletetaan että vastustajani pelaa mahdollisimman hyvin*
  - eli valitsee samasta puusta itselleen parhaan (eli minulle huonoimman) haaran.

- Seuraavassa pelipuu piirrettynä hiukan abstraktimmin.


0
−1
0
−1
−1
1
0
1
0
−1
−1
1
0
1
0
−1

- Pelipuu evaluoidaan lähtien lehdistä edeten juureen:

  Mustat solmut

  - edustavat ristin omaa siirtovuoroa
  - ovat ns. *max*-solmuja
  - saavat arvokseen lapsen jolla on *suurin* arvo.

  Valkoiset solmut

  - edustavat vastustajan eli nollan siirtovuoroa
  - ovat ns. *min*-solmuja
  - saavat arvokseen lapsen jolla on *pienin* arvo.

- Ristin siirtoa vastaa se lapsi, josta juuren max-solmu sai arvonsa.

  (Yhtä maksimaalisen hyviä lapsia voi olla monta.)

- Kuten edellisissä esimerkeissämme (kalvoilla 3.7)

  nytkään ei tarvitse luoda koko pelipuuta eksplisiittisesti muistiin

  vaan riittää generoida yksi polku kerrallaan.

- Seuraavat vuorottain rekursiiviset operaatiot suorittavat pelipuun evaluoinnin.

- Aluksi kutsutaan

  risti(nykyinen pelitilanne)

  ja vastauksena saadaan paras varma pelin lopputulos.

ristiv(v)

1 **if** peli on ohi **then**
2   **if** risti voitti **then return** $+1$
3   **if** nolla voitti **then return** -1
4   **return** ± 0
5 mybest $\leftarrow -\infty$
6 **for each** solmun v lapsi w **do**
7   newval $\leftarrow$ nolla(w)
8   **if** newval $>$ mybest **then**
9   mybest $\leftarrow$ newval
10 **return** mybest

nolla(v)

1 **if** peli on ohi **then**
2   **if** risti voitti **then return** $+1$
3   **if** nolla voitti **then return** -1
4   **return** ± 0
5 myworst $\leftarrow +\infty$
6 **for each** solmun v lapsi w **do**
7   newval $\leftarrow$ risti(w)
8   **if** newval $<$ myworst **then**
9   myworst $\leftarrow$ newval
10 **return** myworst

- Rivillä 1 tutkitaan, onko peli jo ohi:
  - lauta täynnä
  - jommalla kummalla kolmen suora.

  Silloin palautetaan vastaava arvo (rivit 2–4).

- Jos peli jatkuu vielä, niin
  1. käymme läpi kaikki mahdolliset siirrot (rivi 6)
  2. evaluoimme miten peli etenee tämän siirron seurauksena (rivi 7).

- Funktio risti(v) palauttaa parhaan löytämänsä siirron antaman arvon (rivit 5, 8 ja 9).

- Funktio nolla(v) huonoimman.

- Esitetty pelipuun evaluointimenetelmä kulkee kirjallisuudessa nimellä *min-max-* (tai minimax-)algoritmi.

- Risti-nollassa pelipuut ovat vielä kohtuullisen kokoisia, eli parhaan mahdollisen siirron valinta vie kohtuullisen ajan.

  (Huomaa: jotkut puun haarat ovat symmetrisiä eikä keskenään ”samanlaisista” tarvitse tutkia kuin yksi vaihtoehto.)

- Esimerkiksi shakissa tilanne onkin aivan toinen:

  pelipuut ovat niin suuria että niiden läpikäynti kokonaisuudessaan on mahdotonta

- Tällöin paras mitä voidaan tehdä on
  - generoida pelitilanteita vain tiettyyn syvyyteen asti
  - arvioida katkaisusyvyydessä kesken jäävän pelitilanteen arvoa jotenkin

    (jäljellä olevat omat/vastustajan nappulat, asetelma laudalla, ym . . .)
  - karsia hakua branch-and-bound-tyyppisesti

    ns. alfa-beta-karsinnalla.
- Silloin tietokoneen

$$\text{pelitaito} \approx \text{syvyys} + \text{arviointitapa}.$$

- Lisää kurssilla *Tekoäly*.
- On myös pelejä, joiden puun haarautumisaste eli siirtovaihtoehtojen määrä on liian suuri, jotta päästäisiin järkeviin syvyyksiin järkevässä ajassa.

  Esimerkiksi lautapelissä Go on $\approx$ 350 eri aloitusvaihtoehtoa. . .

4 Hajautus

- Tarkastellaan edelleen kalvoilla 1.5 määritellyn joukkotietotyypin toteuttamista.

- Useissa sovelluksissa riittää että operaatiot **insert**, **delete** ja **search** toimivat *yleensä hyvin nopeasti*.

- Avainten *järjestystä* käyttäviä operaatioita **min**/**max** ja **pred**/**succ** ei tarvita.

  (Tai tarvitaan niin harvoin, ettei niitä kannata nopeuttaa.)

- Kalvojen 3.4 puhelinluettelossa tuskin tarvitaan ”seuraavaksi suurempaa puhelinnumeroa” tms.

- Sen sijaan puhelinluettelon indeksissä nimen mukaan (aakkos)järjestysoperaatioita voidaan tarvita.

- Jos **insert**, **delete** ja **search** riittävät, niin *hajautusrakenne* on varteenotettava joukon toteutustapa.

- Perusidea:

  - koska avaimia ei tarvitse tallettaa järjestyksessä

  - niin avaimet voidaan tulkita (etumerkittöminä kokonais)*lukuina*

  - joilla saa tehdä *laskutoimituksia*

  - joiden tuloksia voi käyttää *taulukkoindekseinä*.

- Kalvojen 3 hakupuihin verrattuna

  + operaatiot vievät vain *vakioajan*

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

  mutta vain ”jos kaikki on vielä hyvin”. . .

  – hajautustaulussa voi olla *ruuhkaa:*

    – se voi olla (lähes) täysi

    – operaatiot voivat keskittyä vain pieneen osaan siitä.

  Silloin operaatiot hidastuvatkin samaan *lineaariseen* aikaan

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

  (missä n on avainten lukumäärä) kuin kalvojen 2.4 listoilla.

- **Uhraamme** varman takuun suoritusajasta

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

  kaikissa olosuhteissa

  saadaksemme normaaliolosuhteissa parhaan mahdollisen.

- Olkoon

  $U =$ kaikkien mahdollisten avainten joukko.

- Otetaan käyttöön

  $$T[0 \ldots m-1] = \text{hajautustaulukko}$$

  missä

  $$m = \text{taulukon pituus}$$

  johon avaimet talletetaan.

- Nyt

  $$m \ll |U|$$

  eli mahdollisia avaimia on *paljon enemmän kuin* taulukkopaikkoja.

  – Jopa ääretön määrä.

  – Jos olisi $|U| \leq m$, niin jokaiselle avaimelle $k \in U$ riittäisi oma yksityinen taulukkopaikka $T[k]$.

- Määritellään *hajautusfunktio*

$$h\colon U \to \{0, 1, 2 \ldots, m-1\}$$

  - eli hajautusfunktio h kuvaa jokaisen avaimen k jollekin kokonaisluvulle $h(k)$ väliltä $0, 1, 2, \ldots, m-1$

  - eli jollekin taulukon T indeksille.

- Sanotaan: $h(k_i)$ on avaimen k_i *(koti)osoite*.

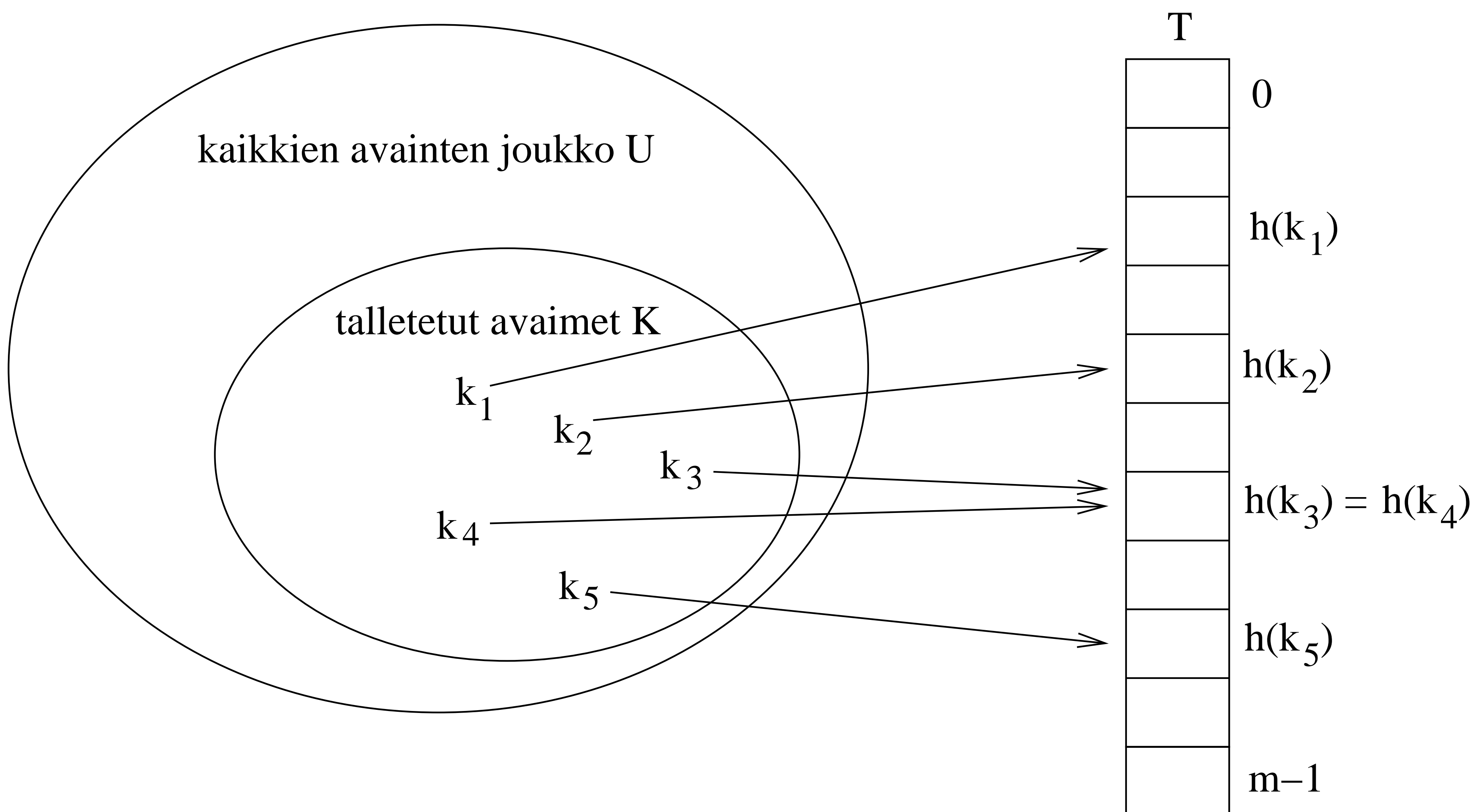
  Ideana on tallettaa avain k_i kostiosoitteensa $h(k_i)$ mukaiseen taulukkopaikkaan $T[h(k_i)]$.

- Kuten seuraava kuva kertoo, voi kaksi avainta saada saman osoitteen

  - eli joillekin avaimille k_3, k_4 voi olla $h(k_3) = h(k_4)$.

  - Sanotaan: avainten k_3 ja k_4 välillä sattuu *yhteentörmäys*.

  - Nämä yhteentörmäykset on se ruuhkaisuus, joka voi hidastaa hajautusta.


T
0
kaikkien avainten joukko U
h(k1)
talletetut avaimet K
h(k2)
k1
k2
k3
h(k3) = h(k4)
k4
k5
h(k5)
m–1

- Oleellista onkin suunnitella hajautusfunktio h sellaiseksi, että yhteentörmäyksiä tapahtuu mahdollisimman vähän.

- Palataan hieman myöhemmin hajautusfunktion suunniteluun.

- Mutta olipa hajautusfunktio miten hyvä tahansa

  ei jokaiselle paikalle yleensä riitä omaa osoitetta

  kun talletettavien alkioiden määrä n kasvaa tarpeeksi suureksi

  (viimeistään kun $n > m$).

- Tarvitaan siis väistämättä jonkinlainen *yhteentörmäysten käsittelystrategia*.

4.1 Yhteentörmäykset ylivuotoketjuun

- Yksi tapa ratkaista yhteentörmäykset on tallettaa yhteentörmäävät alkiot linkitettyihin listoihin.

- Tämä lienee käytännössä useimmiten paras tapa.

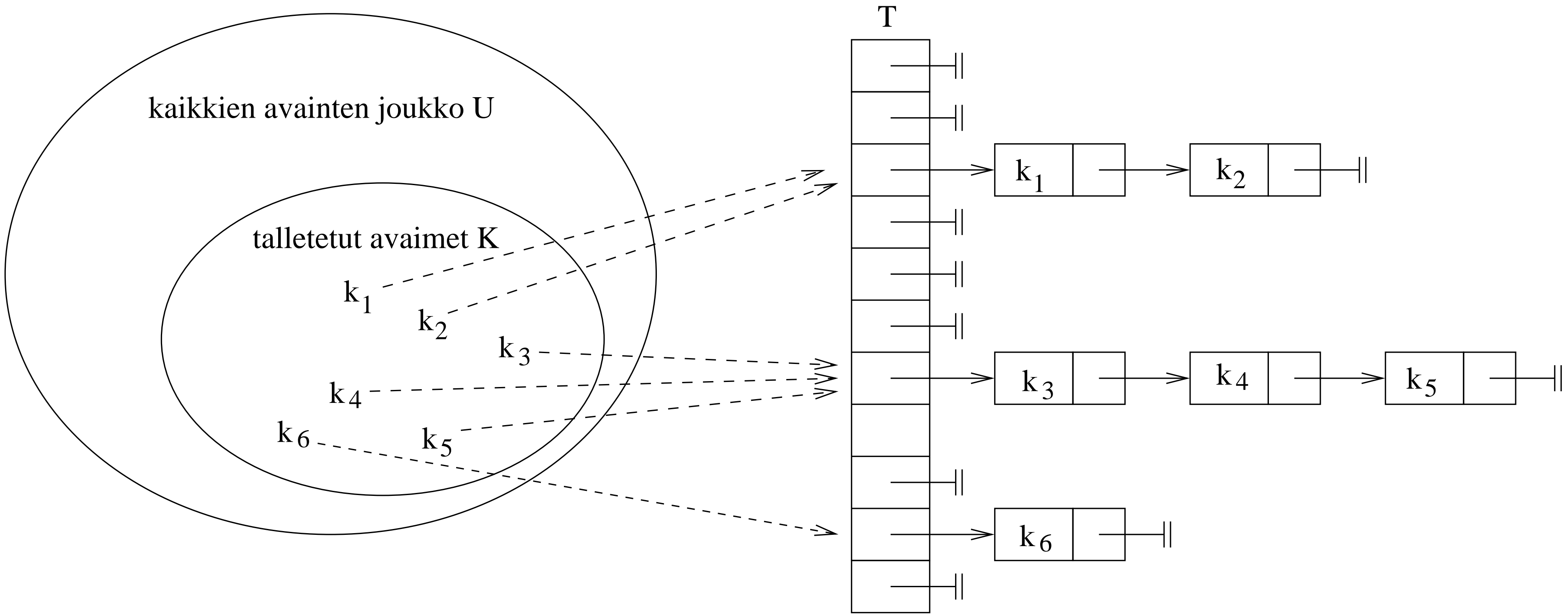
- Kirjallisuudessa tätä yhteentörmäysten ratkaisustrategiaa sanotaan *ketjutukseksi*

  (yleensä chaining, mutta joissakin kirjoissa open hashing).

- Seuraavassa kuvassa hajautustaulukon alkio $T[i]$ sisältää sen listan

  johon laitetaan kaikki ne tallettavat avaimet

  joiden kotiosoitteeksi tulee i.


T
kaikkien avainten joukko U
talletetut avaimet K
k1
k2
k3
k4
k6
k5
k1
k2
k3
k4
k5
k6

- Esitetään nyt toteutus ketjutuksella varustettuun hajautukseen käyttäen kalvoilla 2.4 esiteltyjä linkitettyjä listoja.

- Nyt taulukon $T[0 \ldots m-1]$ kukin alkio $T[i]$ on siis viite oman listansa alkuun.

- Alussa hajautusalue on tyhjä eli jokainen $T[i] = \text{NIL}$.

- Tietueen etsiminen hajautusrakenteesta T avaimella k:

  hashSearch(T, k)
  1 $L \leftarrow T[h(k)]$
  2 $x \leftarrow$ listSearch(L, k)
  3 **return** x

  Rivillä 1 selvitetään se taulun T lista L, joka voi sisältää avaimen k.

  Rivillä 2 kutsutaan listan hakuoperaatiota hakemaan avainta k listasta L.

- Tietueen x lisääminen hajautusrakenteeseen T on vastaavasti:

  hashInsert(T, x)
  1 $L \leftarrow T[h(\text{key}[x])]$
  2 listInsert(L, x)

  Rivillä 1 selvitetään se taulun T lista L, joka voi sisältää avaimen key[x].

  Rivillä 2 kutsutaan listan lisäysoperaatiota lisäämään tietue x listaan L.

- Samoin tietueen poistaminen:

  hashDelete(T, x)
  1 $L \leftarrow T[h(\text{key}[x])]$
  2 listDelete(L, x)

  Rivillä 1 selvitetään se taulun T lista L, joka voi sisältää tietueen x.

  Rivillä 2 kutsutaan poisto-operaatiota poistamaan tietue x listasta L.

  Tarvitaan myös lista L, koska x voi olla sen ensimmäinen tietue.

- Mikä on operaatioiden aikavaativuus?

- Oletetaan että hajautusfunktion h arvon laskeminen vie vakioajan

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

- Kalvoilta 2.4 muistamme että myös listoissa tietueen lisäys ja poisto vievät vakioajan

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

 (kunhan valitaan sopivat toteutustapa).

- Lisäys ja poisto vievät siis vakioajan

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

 koska niissä

 1. lasketaan hajautusfunktion h arvo
 2. indeksoidaan tuloksella taulukkoa T
 3. sovelletaan taulukkopaikan sisältöön listaoperaatiota.

- Kalvoilta 2.4 muistamme että j alkiota sisältävältä listalta L etsintä vie lineaarisen ajan

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

listan pituuden j suhteen.

- Hajautustaulusta etsinnän aikavaativuus siis riippuu eri ylivuotolistojen pituudesta.

Pahimmassa tapauksessa

- kaikki taulukon n avainta saavat saman osoitteen i

  (tai pääosa avaimista)

- eli päätyvät samaan listaan $T[i]$

- josta haku vie siis lineaarisen ajan

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

- Silloin hajautus käyttäytyy kuin lista.

- Hajautus pyrkii parempaan, joten pahin tapaus ei anna oikeudenmukaista kuvaa.

Keskimääräisen tapauksen arvio:

- Oletetaan, että kun hajautusfunktio h "valitsee" taulukkopaikkaa nykyiselle avaimelle k_i, niin
  - jokainen paikoista $T[0 \ldots m-1]$ on yhtä todennäköinen, eli h
    - jakaa avaimet k *tasaisesti* taulukkoon T
    - on *hyvä* hajautusfunktio
  - rippumatta aikaisemmille avaimille $\ldots, k_{i-3}, k_{i-2}, k_{i-1}$ valituista paikoista eli h ei välitä taulukosta T.
- Silloin jokaisen listan $T[i]$ pituuden *odotusarvo* on
$$\alpha = \frac{n}{m}$$
missä n on talletettujen avainten lukumäärä (perustelu sivuutetaan).
- Suuretta α kutsutaan *täyttösuhteeksi*.

Odotusarvo on todennäköisyyslaskennan käsite, jonka intuitio on seuraava:

- Meillä on

  q **eri vaihtoehtoa** joilla on eri

  todennäköisyydet

  $$p_1 + p_2 + p_3 + \cdots + p_q = 1$$

  ja

  joka vaihtoehdolla oma arvonsa

  $$v_1, v_2, v_3, \ldots, v_q$$

  meidän näkökulmastamme.

- Paljoko arvoa voimme *odottaa* saavamme?

  $$p_1 \cdot v_1 + p_2 \cdot v_2 + p_3 \cdot v_3 + \cdots + p_q \cdot v_q.$$

- Jos heitämme nopan, ja saamme 1 EUR/silmä, niin voimme odottaa voittavamme pelistä

  $$\frac{1}{6}\cdot 1+\frac{1}{6}\cdot 2+\frac{1}{6}\cdot 3+\frac{1}{6}\cdot 4+\frac{1}{6}\cdot 5+\frac{1}{6}\cdot 6 = 3.50\,\mathrm{EUR}.$$

Lause 4.1. *Avaimen hakuajan odotusarvo on $\mathcal{O}(1+\alpha)$.*

Todistus: Oletusten mukaan haettu avain k voi olla yhtä suurella todennäköisyydellä missä tahansa ylivuotoketjussa $T[i]$.

Tuloksettomassa haussa käydään läpi lista $T[h(k)]$. Sen pituus on oletuksen mukaan α alkiota. Kun otetaan vielä huomioon hajautusfunktion laskemiseen ja taulukkoindeksointiin kuluva vakioaika, niin saadaan vaatimukseksi $\mathcal{O}(1+\alpha)$.

Tuloksellinen haku on hankalampi, ja tarkka analyysi sivuutetaan. Perusidea:

Todennäköisyys ryhtyä käymään läpi listaa $T[i]$ on nyt verrannollinen sen pituuteen, ja haettava tietue voi sijaita yhtä todennäköisesti missä tahansa listan $T[i]$ kohdassa.

Näistä voidaan johtaa, että tyypillisesti käydään läpi noin puolet listasta $T[i]$, jossa on tyypillisesti noin α tietuetta, siis $\mathcal{O}(1+\alpha/2)$ askelta. □

- Ohjelmoijan lukutapa lauseelle 4.1:

  1. Jos tunnetaan (tai arvioidaan) ennalta

     $n =$ suurin yhtä aikaa talletettavien avainten lukumäärä

  2. niin valitaan vakio

     $$\begin{aligned} \alpha &= \text{ketjun pituuden odotusarvo} \\ &= \text{haluttu tyypillinen suorituskyky} \end{aligned}$$

  3. ja varataan taulukolle T

     $$m = \left\lceil \frac{n}{\alpha} \right\rceil$$

     muistipaikkaa (ylöspäin pyöristettynä).

  4. Silloin meillä on lupa olettaa, että operaatiot vievät vakioajan

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

     tai sitten hajautusfunktio h ei ole tehtäviensä tasalla...

- Edellä hajautustaulukko sisälsi ainoastaan viitteet listojen alkuun.

  Kirjallisuudessa tätä ratkaisua kutsutaan *suoraksi* ketjutukseksi.

- Toinen mahdollisuus on tallettaa hajautustaulupaikkaan $T[i]$

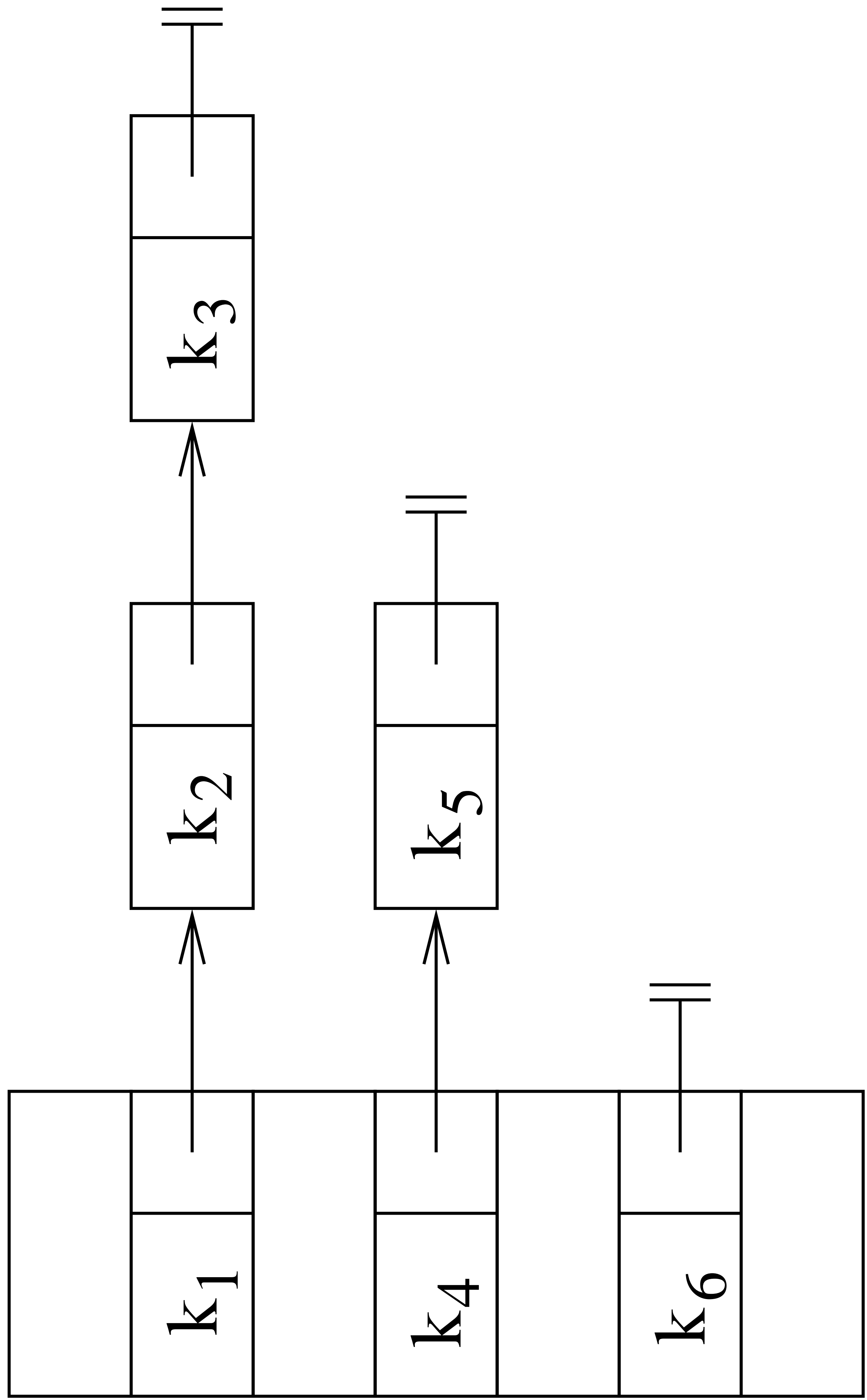
  - ensimmäinen tietue (joka voi myös puuttua)
  - viite muut tietueet sisältävään listaan

  seuraavan kuvan mukaan.

  Tätä ratkaisua kutsutaan *erilliseksi* ketjutukseksi.

- Erillisen ketjutuksen perustelu:

  - Jos halutaan hyvä suorituskyky $\alpha \approx 1$
  - niin varataan $m \approx n$ taulukkopaikkaa
  - joista useimmissa tietueita on ≈ 1.


T
k1
k2
k3
k4
k5
k6

4.2 Hajautusfunktion valinta

- Tavoitteet:
  - hajautusfunktion h arvon tulee olla laskettavissa nopeasti
  - h jakaa avaimet tasaisesti hajautusalueelle $T[0 \ldots m-1]$.

- h olettaa yleensä että avain k on (etumerkitön) kokonaisluku.

- Jos k on kuitenkin vaikkapa merkkijono, niin

  1. muunnetaan se ensin kokonaisluvuksi

     $$k' = g(k)$$

     jollakin (nopealla) muunnoksella g.

  2. sitten hajautetaan tämä muunnettu luku k'

  eli otetaankin hajautusfunktioksi yhdistetty funktio

  $$h(g(k)).$$

- Kun näin löydetään oikea hajautustaulun paikka, niin sitten etsitään, löytyykö alkuperäinen avain k ylivuotoketjusta.

  Siis ketjua selatessa vertaillaan merkkijonoja.

- Sekä funktiolla g että funktiolla h on tärkeää, että se
  - erottelee avaimia hyvin toisistaan
  - riippuu (lähes) kaikista syötteensä biteistä.

4.2.1 Merkkijonosta luvuksi

- Tarkastellaan ensin muunnoksen g valintaa.

- Olkoon avainmerkkijono

$$k = a_1 a_2 a_3 \ldots a_p$$

ja funktio

$$\text{ascii}(a) = \text{merkin } a \text{ ASCII-koodi } 0 \ldots 127$$

(7-bittinen merkistö).

- Summa

$$g(k) = \sum_{i=1}^{p} \text{ascii}(a_i) \qquad (1)$$

merkkien koodeista?

+ riippuu jokaisen merkin jokaisesta bitistä

– tulos on pieni

$$g(k) \leq p \cdot 127$$

joten se pakkaa merkkijonon k ehkä liiankin tiiviisti luvuksi.

- Ensimmäisen, keskimmäisen ja viimeisen merkin muodostama $3 \cdot 7 = 21$-bittinen luku

$$g(k) = \text{ascii}\,(a_1) + \text{ascii}\left(a_{\lceil \frac{p}{2} \rceil}\right) \cdot 128 + \text{ascii}\,(a_p) \cdot 128^2? \quad (2)$$

  – ei riipu edes jokaisesta merkistä

  + erilaisia tulosvaihtoehtoja

$$2^{7 \cdot 3} = 2^{21} = 128^3 = 2\,097\,152$$

  joka riittänee tyypillisimmille taulukon T pituuksille m

  + nopeasti laskettavissa *bittioperaatioilla:*

    – kakkosen potenssilla kertominen

$$x \cdot 2^y$$

    on luvun x *siirtämistä* y *bittiä vasemmalle*.

    – yhteenlaskettavat luvut eivät mene siirtojensa jälkeen päällekkäin

    joten '+' onkin bittitason ''tai''-operaatio.

- Ideoista (1) ja (2) on useita variaatioita, kuten

$$g(k) = \sum_{i=1}^{n} \text{ascii}(a_i) \cdot 128^{i-1}$$

eli tulkitaankin *koko* merkkijono binääriluvuksi — tulos ylittää laskentatarkkuuden

$$g(k) = \text{ascii}(a_1) + \text{ascii}(a_2) + \text{ascii}(a_3)$$

eli otetaankin kolme *ensimmäistä* merkkiä — vain $3 \cdot 127 + 1 = 382$ eri arvoa

$$g(k) = \text{ascii}(a_1) + \text{ascii}(a_2) \cdot 128 + \text{ascii}(a_3) \cdot 128^2$$

eli tulkitaankin ne binääriluvuksi — entäpä sukunimet `Virta`, `Virtanen`, `Virkkunen`,...?

- Muitakin ideoita muunnokseksi g voi kokeilla...

  ...ja myös kritisoida.

- Ideat (1) ja (2) voi myös yhdistää:

```
1   b[0...2] ← 0
2   for i ← 1 to n do
3       b[i mod 3] ← b[i mod 3] XOR a_i
4   return b[0] + b[1] · 128 + b[2] · 128^2
```

  - Aputaulukkopaikkaan $b[j]$ kootaan merkkien $a_{j+3\cdot i}$ sisältämä informaatio (rivit 1–3).

  - Rivillä 3 käytettiin bittitason operaatiota "joko-tai" (eXclusive OR).

    - Se on kuin '+' jossa muistinumero unohdetaan:

$$\begin{array}{ccc} 0 & 1 & 1 \\ 1 & 1 & 0 \\ \hline 1 & 0 & 1 \end{array}$$

    - "Summa" riippuu koottujen merkkien kaikista biteistä...

    - ...mutta pysyy 7-bittisenä.

  - Rivi 4 yhdistää aputaulukon yhdeksi luvuksi (bittioperaatioilla).

4.2.2 Jakojäännösmenetelmä

- Siirrytään esittelemään käytännössä toimivaksi osoittautuneita hajautusfunktioita h.

- Jakojäännösmenetelmä on

$$h(k) = k \bmod m$$

missä taulukon pituus m

  - on *alkuluku*
    - eli jakautuu tasan vain itsellään ja ykkösella.
    - ensimmäiset alkuluvut ovat

$$2, 3, 5, 7, 11, 13, \ldots$$

  - ei ole lähellä mitään kakkosen potenssia

$$2^p$$

  - koska muuten $h(k) \approx$ "lohko avain k p bitin paloihin ja laita ne päällekkäin" (kuten kalvoilla 4.2.1).

- Esimerkiksi:

  Haluamme hajautustaulun johon talletetaan noin

  $$n \approx 2000$$

  tietuetta.

  Sallimme ylivuotolistojen pituuden odotusarvoksi

  $$\leq 3$$

  tietuetta.

  Valitsemme taulukon pituudeksi

  $$m = 701$$

  ja saamme täyttöasteeksi

  $$\alpha = \frac{2000}{701} \approx 2.85.$$

- Seuraavassa kuvassa täytetään hajautustaulukkoa, jossa on

$$m = 11$$

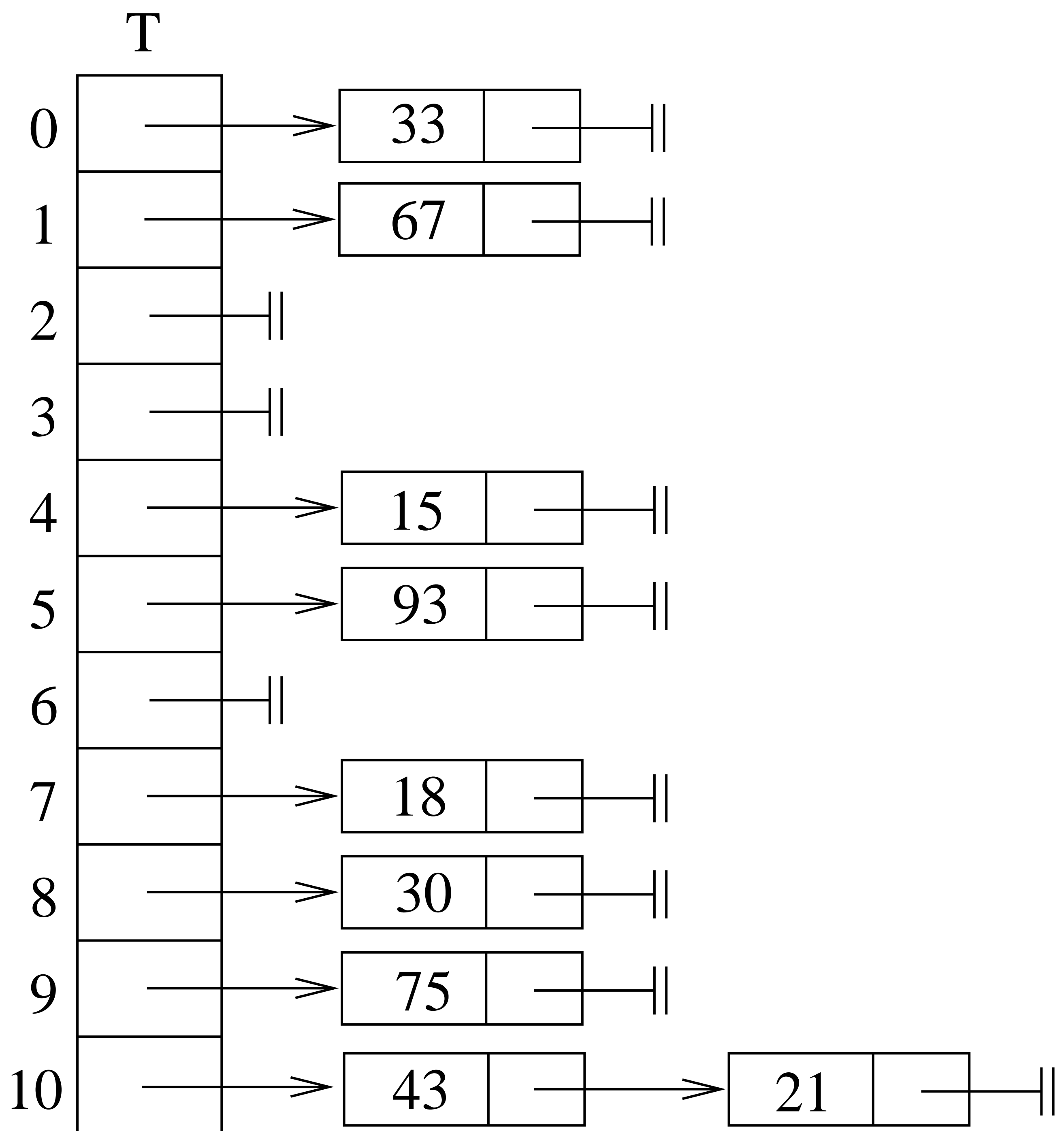
paikkaa. Siis hajautusfunktiona on

$$h(k) = k \bmod 11$$

- Talletetaan avaimet

k	$h(k)$
75	9
43	10
21	10
15	4
18	7
33	0
30	8
67	1
93	5

(Päätetään lisätä uusi tietue aina listan loppuun; listan järjestyksellähän ei ole hajautuksessa merkitystä.)


T
0
1
2
3
4
5
6
7
8
9
10
33
67
15
93
18
30
75
43
21

4.2.3 Kertolaskumenetelmä

- Valitaan reaalilukuvakio

$$0 < A < 1.$$

- Merkitään

$$\begin{aligned} \text{fr}(x) &= \text{reaaliluvun } x \text{ desimaaliosa} \\ &= x - \lfloor x \rfloor \end{aligned}$$

missä

$$\lfloor x \rfloor = \text{reaaliluvun } x \text{ kokonaisosa.}$$

- Esimerkiksi

$$\begin{aligned} \text{fr}(3.14159) &= 3.14159 - \lfloor 3.14159 \rfloor \\ &= 3.14159 - 3 \\ &= 0.14159. \end{aligned}$$

- Kertolaskumenetelmässä hajautusfunktio on

$$h(k) = \lfloor m \cdot \text{fr}(A \cdot k) \rfloor .$$

- Etuna kalvojen 4.2.2 jakojäännösmenetelmään on se, että taulukon pituuden m saa nyt valita vapaasti.

- Hyvä valinta olisi *kultaisen leikkauksen* suhdeluku

$$A = \frac{\sqrt{5}-1}{2} \approx 0.6180339887\ldots$$

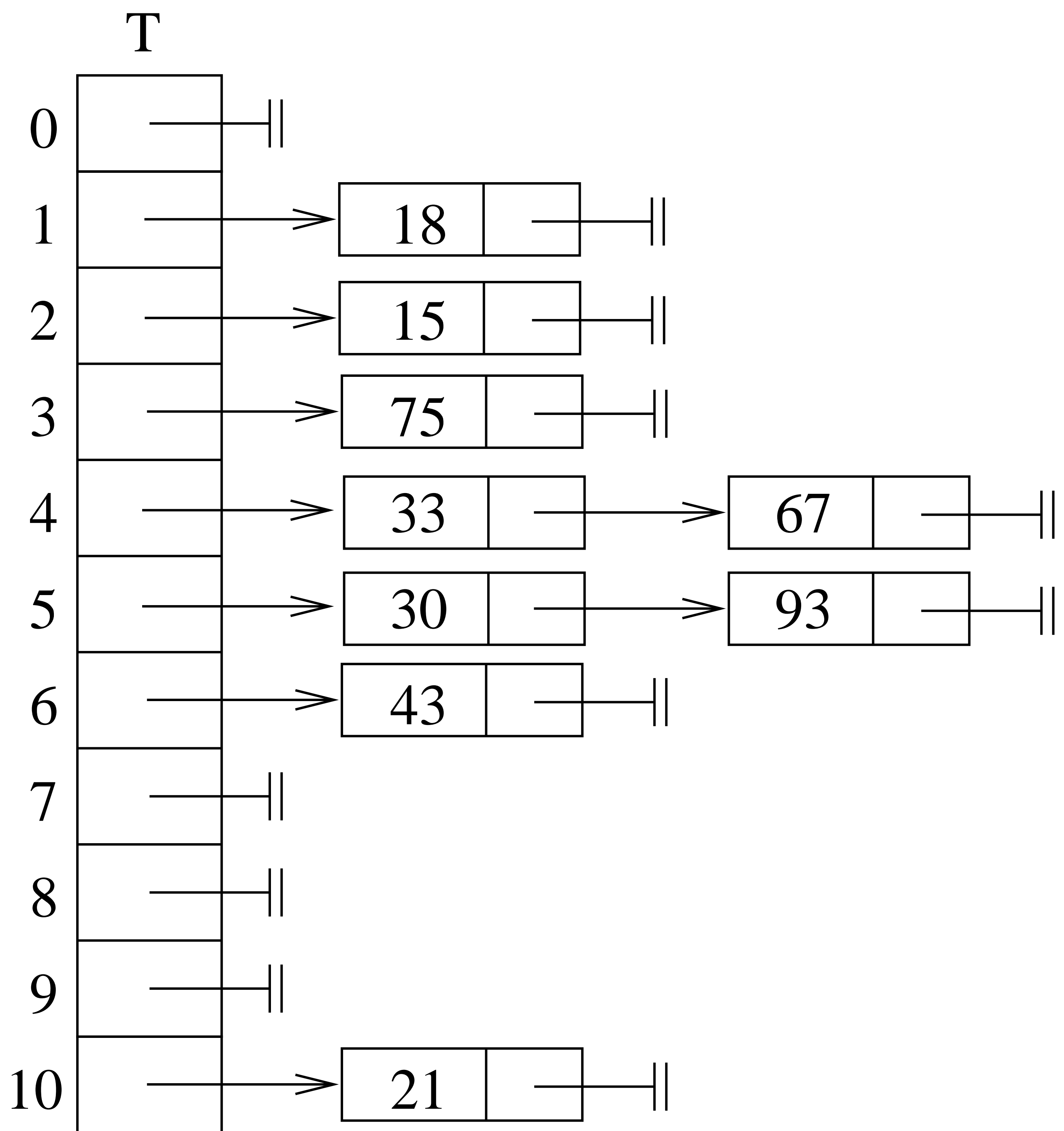
joka johtaa ns. *Fibonacci*-hajautukseen.

(D.E. Knuth: *The Art of Computer Programming. Vol. 3: Sorting and Searching. 2nd Ed.* Addison Wesley, 1998, sivut 518–519.)

- Tehdään kalvojen 4.2.2 esimerkki uudelleen hajautusfunktiona nyt

$$h(k) = \lfloor 11 \cdot \mathrm{fr}(0.618 \cdot k) \rfloor .$$

k	$h(k)$
75	3
43	6
21	10
15	2
18	1
33	4
30	5
67	4
93	5


T
0
1
18
2
15
3
75
4
33
67
5
30
93
6
43
7
8
9
10
21

4.2.4 Universaalihajautus

- Valitaan alkuluku

$$p \geq n.$$

- Valitaan *satunnaisesti* kokonaisluvut

$$1 \leq a < p$$
$$0 \leq b < p.$$

- Hajautusfunktioksi tulee silloin

$$h(k) = ((a \cdot k + b) \bmod p) \bmod m. \qquad (3)$$

- Järkeily satunnaisuuden käytön takana:
  - Ohjelmoija voi valita vääränlaisen hajautusfunktion h
    - koska hän ei tiedä etukäteen tarpeeksi hyvin, minkälaiset ovat ohjelman käsittelemän *datan tilastolliset ominaisuudet* — minkälaisia avaimia on paljon/vähän,...
    - jolloin ohjelma toimii *joka ajokerralla* hitaasti.
  - Jos sen sijaan ohjelma aina käynnistyessään *arpoo* tälle ajokerralle uuden hajautusfunktion h, niin se
    - kyllä *joskus* arpoo itselleen sellaisen h, joka ei sovikaan dataan
    - mutta (toivottavasti) *harvoin* — eikä ainakaan joka ajokerta!
    - Silloin hajautukseen kuluvan ajan "odotusarvo on pieni".

- Järkeily juuri hajautusfunktio*perhe*en (3) käytön takana:

  Kiinnitetään mielivaltainen avainpari $x \neq y$.

  Kysytään: kuinka todennäköistä on arpoa sellainen funktio, jolla $h(x) = h(y)$, eli jolla ne törmäävät?

  Osoittautuu: tämä todennäköisyys $\frac{1}{m}$ ei riipu avaimista x, y.

  Johtopäätös 1: tämä perhe kohtelee kaikkia avainpareja x, y tasavertaisesti.

  Johtopäätös 2: annetulle datalle D on vain vähän sellaisia h, joilla D aiheuttaa paljon yhteentörmäyksiä.

- Tarkempia perusteluja löytyy

  – Karvin monisteen luvun 7.2 kohdasta 3

  – Cormenin kirjan luvusta 11.3.3

  mutta sivuutetaan ne.

- Tehdään jälleen kalvojen 4.2.2 esimerkki arvoilla

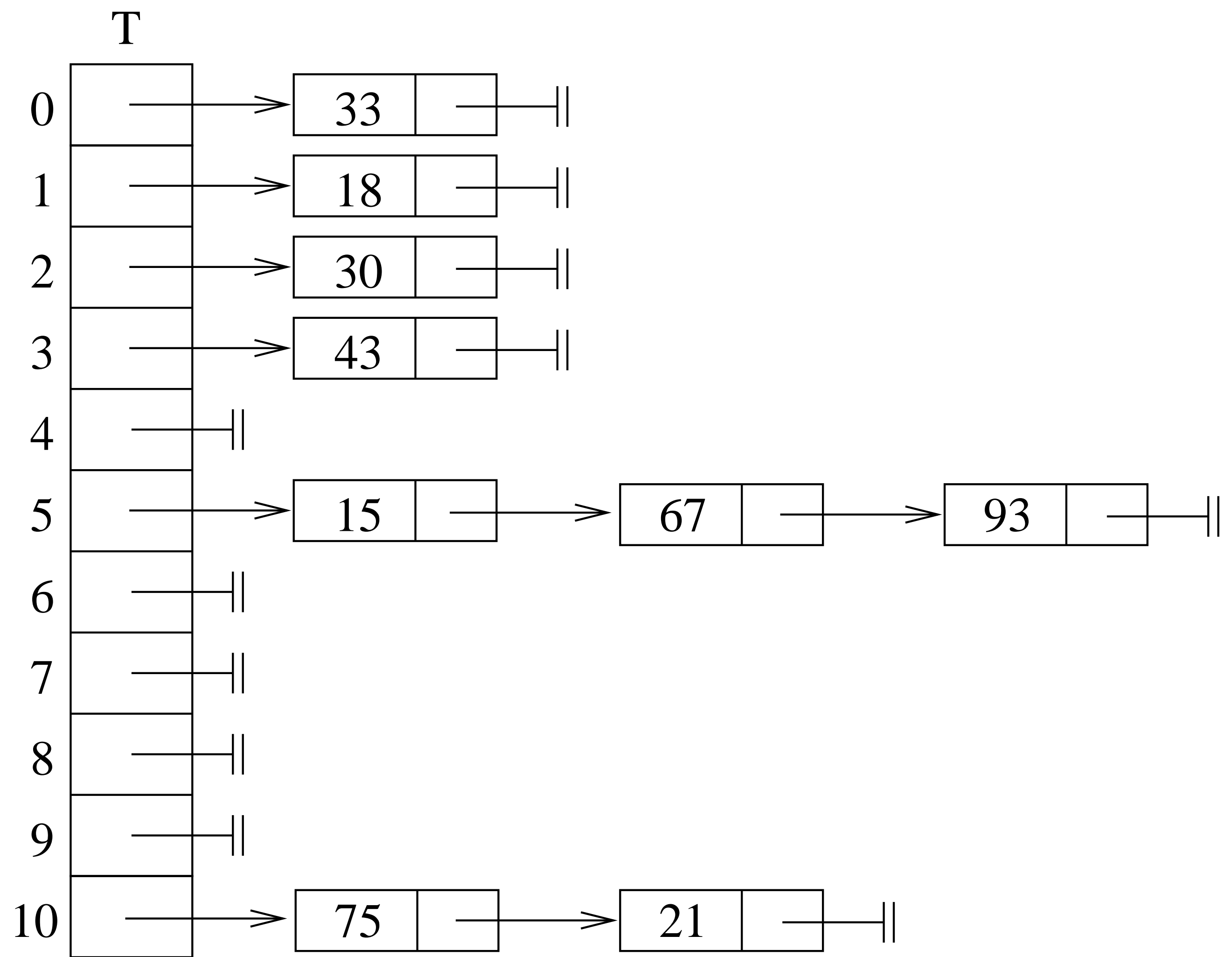
$$p = 53$$
$$a = 31$$
$$b = 17$$

eli hajautusfunktiolla

$$h(k) = (\underbrace{(31 \cdot k + 17) \bmod 53}_{\star}) \bmod 11.$$

k	$\star$	$h(k)$
75	10	10
43	25	3
21	32	10
15	5	5
18	45	1
33	33	0
30	46	2
67	27	5
93	38	5

- Arpomalla a ja b toisin olisi voitu päätyä
  - eri arvot antavaan hajautusfunktioon h
  - jonka käyttö olisi saattanut johtaa parempaan tulokseen.


T
0
1
2
3
4
5
6
7
8
9
10
33
18
30
43
15
67
93
75
21

4.3 Avoin hajautus

- Kalvojen 4.1 ylivuotoketjutuksen rinnalla toinen tapa ratkaista yhteentörmäävien tietueiden ongelma on *avoin hajautus* (eng. open addressing).

- Avoimessa hajautuksessa *kaikki* tietueet talletetaan itse hajautustauluun.

- jos kotipaikka $T[h(k)]$ on varattu, niin avaimelle k etsitään jokin muu paikka taulusta.

- Uusikin paikka voi olla varattu, tällöin jatketaan etsimistä edelleen.

- Ne paikat joihin avainta k yritetään laittaa, muodostavat avaimen k *kokeilujonon* (engl. probe sequence).

- Siis taulun sisäinen kokeilujono ottaa ulkoisen ylivuotoketjun roolin.

- Jos avaimen k kaikki aikaisemmat kokeilut $0, 1, 2, \ldots, j-1$ ovat epäonnistuneet, niin j:s kokeilu kohdistuu paikkaan

$$\begin{aligned} h(k,j) &= (h'(k) + s(j,k)) \bmod m \\ h'(k) &= \text{tavallinen hajautusfunktio} \\ s(j,k) &= \text{kokeilufunktio.} \end{aligned}$$

- Vaaditaan: Jokaiselle avaimelle k kokeilujono

$$h(k,0), h(k,1), h(k,2), \ldots, h(k,m-1)$$

on taulukkopaikkojen

$$0, 1, 2, \ldots, m-1$$

jokin permutaatio

eli kokeilujonon on kokeiltava jokaista hajautustaulun indeksiä kertaalleen.

- Avain k talletetaan kokeilujononsa ensimmäiseen vapaaseen paikkaan $T[h(k,j)]$.

- Kalvojen 4.1 ylivuotoketjutukseen verrattuna:

  - Hajautus on nyt ”avointa” siinä mielessä, että jokainen taulukkopaikka osallistuu jokaiseen kokeilujonoon

    eikä tallennustila enää jakaudukaan erillisiin listohin.

  – Taulun $T[0 \ldots m-1]$ tallennuskapasiteetilla on nyt kiinteä yläraja: m tietuetta.

    – Taulu ei siis mukaudukaan tietueiden lukumäärään

    – vaan ilmoittaa ”täynnä” ja lisäys epäonnistuu.

  – Epäonnistuvat haut hidastuvat ajan myötä: jokainen paikka $T[i]$ kuuluu jokaiseen jonoon $T[h(k,j)]$.

  + Taulu on tiiviimmin muistissa: listojen viitekenttiä ei enää tarvita.

  + Toteutus ei tarvitse operaatiota **new**.

- Siis avoin hajautus on hyödyllinen jos
  - muistinhallinta täytyy tehdä staattisesti
  - hakuoperaatiot tyypillisesti onnistuvat.

- Esimerkkinä käyttöjärjestelmän prosessitaulu:
  - Kun luodaan uusi prosessi, niin sille annetaan yksikäsitteinen tunnusnumero

    joka on luonteva hajautusavain prosessin muut tiedot sisältävälle tietueelle.
  - Yhtä aikaa olemassa olevien prosessien lukumäärällä on usein kiinteä yläraja
    * jonka muuttaminen vaatii muutoksia käyttöjärjestelmään (tai vähintään uudelleenkäynnistyksen)
    * josta saadaan hajautustaululle koko.
  - Operaatioissa kuten ”pysäytä prosessi numero p” viitataan (yleensä) vain olemassa oleviin prosesseihin p.

4.3.1 Operaatiot kokeillen lineaarisesti

- Oletetaan, että jos hajautustaulun paikka $T[i]$ on tyhjä, niin sillä on erityisarvo NIL.

- Ennen kuin katsotaan miten operaatiot toteutetaan avoimessa hajautuksessa, tutustumme yksinkertaisimpaan kokeilustrategiaan, eli *lineaariseen kokeiluun*.

- Kokeillaan kotiosoitteesta $h'(k)$ alkaen taulukkopaikkoja
  - eteenpäin peräkkäin
  - pyörähtäen taulukon lopusta alkuun.

  Vertaa kalvojen 2.2 taulukkototeutus jonolle.

- Kokeilujono on siis

$$h(k, j) = (h'(k) + j) \bmod m$$

eli

$$\begin{gathered}(h'(k) + 0) \bmod m, \\ (h'(k) + 1) \bmod m, \\ (h'(k) + 2) \bmod m, \\ \vdots \\ (h'(k) - 1) \bmod m.\end{gathered}$$

- Esimerkiksi jos $m = 10$ ja $h'(k) = 7$, niin kokeilujono olisi

$$7, 8, 9, 0, 1, 2, 3, 4, 5, 6.$$

- Tietueen x lisääminen hajautustauluun T:

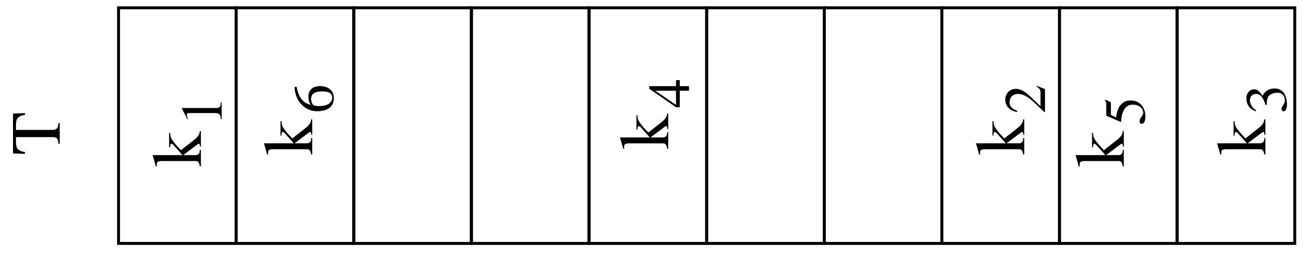
  hashInsert(T, x)
  1 $k \leftarrow$ key[x]
  2 $i \leftarrow 0$
  3 **repeat**
  4 $\quad j \leftarrow h(k, i)$
  5 $\quad$ **if** $T[j] =$ NIL **then**
  6 $\quad\quad T[j] \leftarrow x$
  7 $\quad\quad$ **return** true
  8 $\quad i \leftarrow i + 1$
  9 **until** $i = m$
  10 **return** false

- Lisäysoperaatio palauttaa

  true jos tietue x mahtui hajautustauluun

  false jos taulu T oli jo täynnä.

  Eli koko kokeilujono on käyty läpi, eikä tietue x mahtunut mihinkään (rivi 9).



hash–insert(k_6)

T

k_1				k_4			k_2	k_5	k_3

h(k_6,0)

- Avainta k vastaavan tietueen etsiminen hajautustaulusta T:

```
hashSearch(T, k)
1   i ← 0
2   repeat
3       j ← h(k, i)
4       if T[j] ≠ NIL and key[T[j]] = k
5           then return j
6       i ← i + 1
7   until T[j] = NIL or i = m
8   return NIL
```

- Etsitään kokeilujonoa

$$h(k,0), h(k,1), h(k,2), \ldots, h(k,m-1)$$

läpi niin kauan, kunnes

1. etsitty avain löytyy (rivi 4),
2. törmätään tyhjään hajautustaulun paikkaan (rivin 7 ensimmäinen ehto), tai
3. koko kokeilujono on käyty läpi (rivin 7 jälkimmäinen ehto).

- Jos avain k löytyy, niin palautetaan löytöpaikan indeksi j (rivi 4).

- Jos ei löytynyt, niin palautetaan NIL (rivi 8).

- Miksi saamme lopettaa etsinnän myös tapauksessa 2?

  - Jos avaimella k olisi talletettu tietue, niin se olisi talletettu kokeilujonon *ensimmäiseen tyhjään* paikkaan.

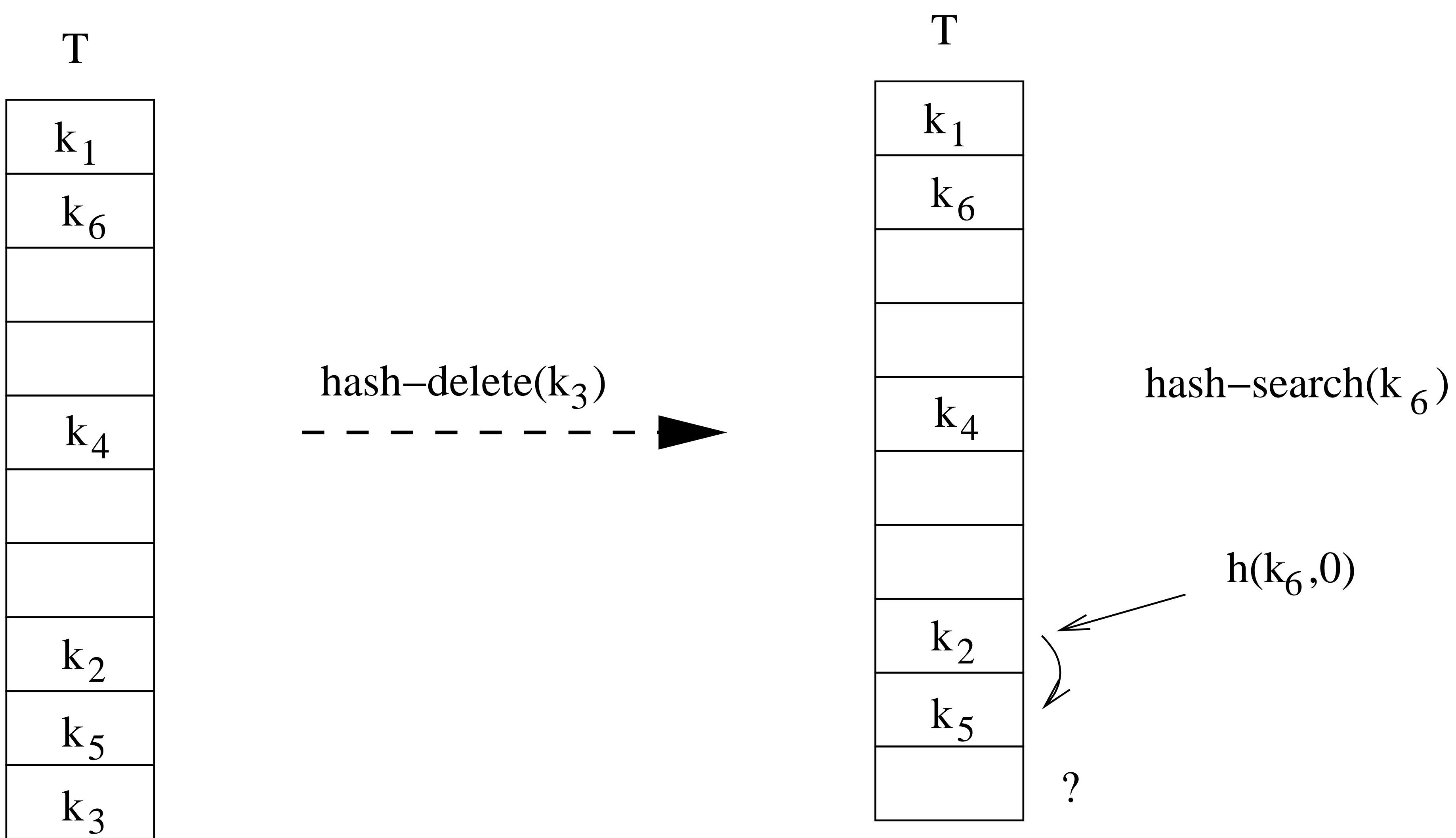
  - Siis tyhjä paikka kokeilujonossa tarkoittaa, että kokeilujonon *käytetty osuus* on nyt käyty läpi.

- Mutta tällöinhän avaimen poistamisen yhteydessä ei paikkaa voidakaan jättää vapaaksi (arvoon NIL):

  - muuten etsintäreittireitti joihinkin avaimiin saattaa katketa

  - kun niihin vievä käytetty osuus katkeaa.

Esimerkki seuraavana kuvana.



T
k1
k6
k4
k2
k5
k3
hash−delete(k3)
T
k1
k6
k4
k2
k5
hash−search(k6)
h(k6,0)
?

- Merkitäänkin poiston yhteydessä poistettavan alkion paikalle erityisarvo DEL.

- Silloin taulukkopaikan arvo

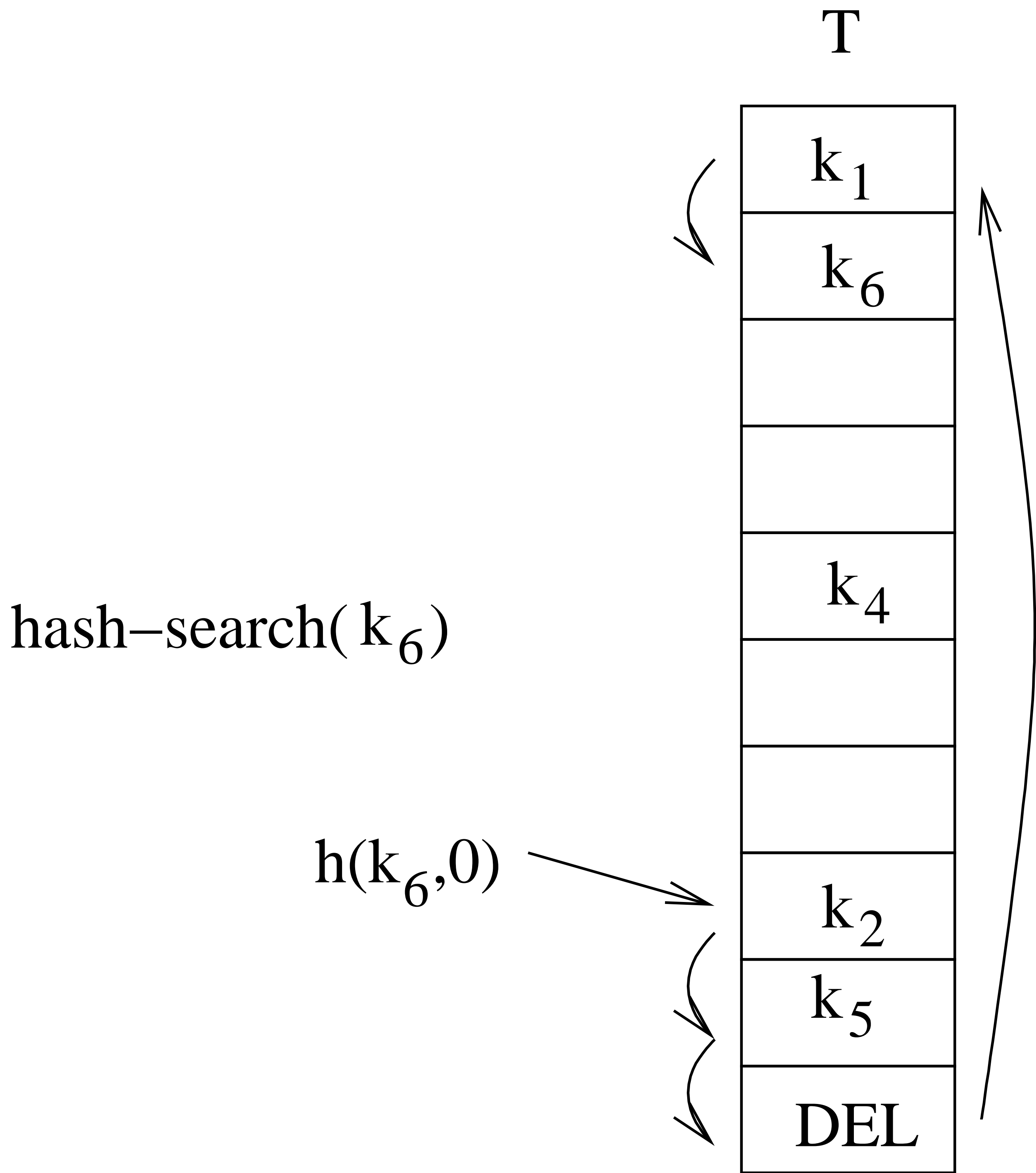
  NIL = tässä paikassa ei ole vielä koskaan ollut mitään sisältöä

  DEL = tässä paikassa ei juuri nyt ole mitään sisältöä, mutta joskus on ollut.

- Avainta k vastaavan tietueen poisto hajautustaulusta T saa silloin muodon:

  hashDelete(T, k)
  1 $i \leftarrow 0$
  2 **repeat**
  3 $\quad j \leftarrow h(k, i)$
  4 $\quad$ **if** $T[j] \neq$ NIL **and** key$[T[j]] = k$
  5 $\quad\quad$ **then** $T[j] \leftarrow$ DEL
  6 $\quad i \leftarrow i + 1$
  7 **until** $T[j] =$ NIL **or** $i = m$

- Nyt avaimen etsiminen toimii taas: seuraavassa kuvassa on edellisen kuvan ongelma korjattu.


T
k_1
k_6
k_4
hash–search(k_6)
h(k_6,0)
k_2
k_5
DEL

- Muutetaan vielä lisäysoperaation riviä 5 siten että lisäys voidaan suorittaa kokeilujonon ensimmäiseen vapaaseen paikkaan $T[j]$

  olipa sen arvo sitten NIL tai DEL.

```
hashInsert(T, x)
1   k ← key[x]
2   i ← 0
3   repeat
4       j ← h(k, i)
5       if T[j] = NIL or T[j] = DEL then
6           T[j] ← x
7           return true
8       i ← i + 1
9   until i = m
10  return false
```

- Avoimen hajautuksen poisto-operaatio on siis *laiska*:

  - se ei tyhjennä ja vapauta muistialuetta

  - vaan jättää taulukkoon DEL-arvoisia kohtia, jotka täyttävät hajautustaulukkoa.

- Nämä DEL-kohdat
  - voivat korvautua normaaleilla alkioilla lisäyksen yhteydessä
  - täytyy käydä läpi haun yhteydessä.

- Haku- ja poistoalgoritmeissa täydennetään rivin 4 **if**-ehto muotoon

$$T[j] \neq \text{NIL} \textbf{ and } \underline{\textbf{and } T[j] \neq \text{DEL}} \textbf{ and } \text{key}[T[j]] = k$$

koska myöskään kenttää key[DEL] ei ole määritelty.

- Esimerkissämme hajautustaulun T koko on $m = 11$ paikkaa.

- Käytetään lineaarista kokeilujonofunktiota

$$h(k,i) = ((k \bmod 11) + i) \bmod 11$$

- Hajautetaan avaimet

  75, 43, 21, 15, 18, 33, 30, 66 ja 92

  tässä järjestyksessä.

- Kahdessa ensimmäisessä lisäyksessä

$$h(75,0) = 9$$
$$h(43,0) = 10$$

  ja tulos on seuraavan kuvan ylin taulukko.

0	1	2	3	4	5	6	7	8	9	10
									75	43

0	1	2	3	4	5	6	7	8	9	10
21				15			18		75	43

0	1	2	3	4	5	6	7	8	9	10
21	33			15			18	30	75	43

0	1	2	3	4	5	6	7	8	9	10
21	33	66		15	92		18	30	75	43

- Kolmannessa lisäyksessä

$$h(21,0) = 10$$

jonka annoimme jo avaimelle 43.

- Joudumme siis etsimään avaimelle 21 uuden paikan.

- Kokeilujonon seuraava paikka

$$\begin{aligned} h(21,1) &= ((21 \bmod 11) + 1) \bmod 11 \\ &= 0 \end{aligned}$$

on yhä vapaa, joten käytämme sen.

- Seuraavat kaksi lisäystä

$$h(15,0) = 4$$
$$h(18,0) = 7$$

mahtuvat vapaille kotipaikoilleen.

- Tilanne näiden viiden lisäyksen jälkeen on edellisen kuvan toinen taulukko.

- Kuudes lisäys

$$h(33, 0) = 0$$

osuu jo varattuun paikkaan, joten etsintä jatkuu seuraavasta paikasta

$$h(33, 1) = 1$$

joka on tyhjä.

- Seitsemäs lisäys

$$h(30, 0) = 8$$

voidaan tehdä heti kokeilujononsa ensimmäiseen paikkaan.

- Tilanne tässä vaiheessa on edellisen kuvan kolmas taulukko.

- Kahdeksannessa lisäyksessä joudutaan käymään läpi paikat

$$h(66,0) = 0$$
$$h(66,1) = 1$$
$$h(66,2) = 2$$

  ennen kuin tilaa löytyy.

- Viimeisessä lisäyksessä käydään läpi paikat

$$h(92,0) = 4$$
$$h(92,1) = 5.$$

- Tuloksena on edellisen kalvon neljäs taulukko.

- Entä jos haluaisimme vielä lisätä avaimen 29?

$$h(29, 0) = 7$$

on varattu ja vapaa löytyy vasta 8. kokeilulla:

$$h(29, 7) = 3!$$

- Entä jos etsisimme taulukosta lukua 41?

1. Etsintä etenisi paikasta

$$h(41, 0) = 7$$

2. aina paikkaan

$$h(41, 7) = 3$$

asti

3. jonka jälkeen voidaan todeta että 41 ei ole taulukossa.

- Poistetaan edellisen kuvan viimeisestä taulukosta avaimet

21, 30 ja 75

ja saadaan seuraavan kuvan ylempi taulukko.

- Avaimen 41 etsintä etenisi edelleen paikasta $h(41, 0) = 7$ aina paikkaan $h(41, 7) = 3$ asti jonka jälkeen voidaan todeta että 41 ei ole taulukossa.

- Avaimen 29 lisäys löytää nyt sille paikan toisella kokeilulla

$$h(29, 1) = 8$$

koska

$$T[8] = \text{DEL}\,.$$

Saadaan seuraavan kuvan alempi taulukko.

0	1	2	3	4	5	6	7	8	9	10
DEL	33	66		15	92		18	DEL	DEL	43

0	1	2	3	4	5	6	7	8	9	10
DEL	33	66		15	92		18	29	DEL	43

- Lineaarinen kokeilujono on
  - \+ helppo toteuttaa
  - − käytännössä aika huono.
    - tauluun tulee usein pitkiä varattuja alueita
    - jotka vielä kasvavatkin suuremmalla todennäköisyydellä kuin lyhyet varatut alueet.

      Esimerkki seuraavassa kuvassa.
    - Ilmiötä nimitetään *ensisijaiseksi (eli primääriseksi) kasautumiseksi* (engl. primary clustering).

- Intuitiivinen syy:
  - Paikkaan $T[p]$ törmääville q avaimelle varataan alue

    $$T[p], T[p+1], T[p+2], \ldots, T[p+q-1]$$
  - jolloin alueelle kuuluvat muut tietueet pakotetaan alueen perään

    $$T[p+q], T[p+q+1], T[p+q+2], \ldots.$$

h(k11)

k02 k01 k04 k06 k05 k03 k10 k07 k08 k09

4.3.2 Neliöinen kokeilu

- Kalvojen 4.3.1 ensisijainen kasautuminen vältetään sillä, että *kokeilujono ei olekaan peräkkäin* taulukossa.

- Eräs tapa on *toisen asteen* lauseke

$$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$$

sopivilla kertoimilla c_1 ja c_2.

- Silloin kokeilujono alkaa

$$\begin{gathered}(h'(k) + 0 \cdot c_1 + 0 \cdot c_2) \bmod m \\ (h'(k) + 1 \cdot c_1 + 1 \cdot c_2) \bmod m \\ (h'(k) + 2 \cdot c_1 + 4 \cdot c_2) \bmod m \\ (h'(k) + 3 \cdot c_1 + 9 \cdot c_2) \bmod m \\ \vdots\end{gathered}$$

- Jos esimerkiksi $c_1 = c_2 = 1$ ja $h'(k_1) = 5$ (ja $m > 25$), niin kokeilujonon alku olisi

$$5, 7, 11, 17, 25, \ldots$$

- Siis taulukossa aletaankin nyt hyppiä *epätasaisin* välein

  kun taas kalvojen 4.3.1 lineaarisessa kokeilussa hypättiin aina seuraavaan paikkaan.

Ohjelmointiongelma on valita

- vakiot c_1 ja c_2
- taulun pituus m

siten, että kokeilujono todellakin käy läpi koko taulukon T.

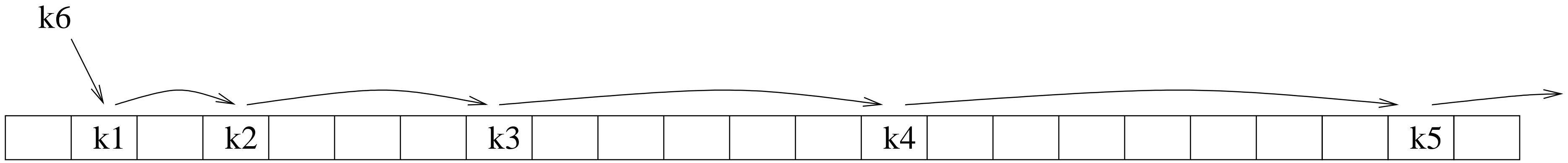
Suorituskykyongelma on *toissijainen (eli sekundäärinen)* kasautuminen:

samaan kotipaikkaan hajautuvien avainten

$$h(k_1, 0) = h(k_2, 0)$$

kokeilujonotkin ovat samat.

Esimerkki seuraavassa kuvassa.



$$h(k1,0) = h(k2,0) = h(k3,0) = h(h4,0) = h(k5,0) = h(k6,0) = 1$$

4.3.3 Kaksoishajautus

- Kalvojen 4.3.2 toissijainenkin kasautuminen vältetään sillä, että *koko kokeilujono* riippuu avaimesta k

  (eikä vain sen alkukohta $h'(k)$ kuten aiemmin).

- Vasta nyt hyödynnämme täysin hajautuksen avoimmuutta!

- Kokeilujono on nyt

  $$h(k,i) = (h'(k) + i \cdot h''(k)) \bmod m$$

  eli

  - ”aloita paikasta $h'(k)$
  - etene $h''(k)$ paikan välein”.

- Hypitään siis taulukossa jälleen *tasavälein* kuten kalvojen 4.3.1 lineaarisessa kokeilussa

  mutta välin *pituus* riippuukin nyt avaimesta.

- Kokeilujono alkaa nyt

$$(h'(k) + 0 \cdot h''(k)) \bmod m$$
$$(h'(k) + 1 \cdot h''(k)) \bmod m$$
$$(h'(k) + 2 \cdot h''(k)) \bmod m$$
$$\vdots$$

- Tarkastellaan esimerkkinä 13-paikkaista hajautustaulua T ja hajautusfunktioita

$$h'(k) = k \bmod 13$$
$$h''(k) = 1 + (k \bmod 11)$$

  kun taulussa on jo avaimet

$$79, 72, 98, 50.$$

- Lisätään vielä avain 14:

$$\begin{aligned} h'(14) &= 14 \bmod 13 \\ &= 1 \\ h''(14) &= 1 + (14 \bmod 11) \\ &= 4 \end{aligned}$$

  joten kokeilujono on

$$1, 5, 9, 0, 4, 8, 12, 3, 7, 11, 2, 6, 1.$$

- Lisätään näin saatuun saatuun tauluun vielä avain 27:

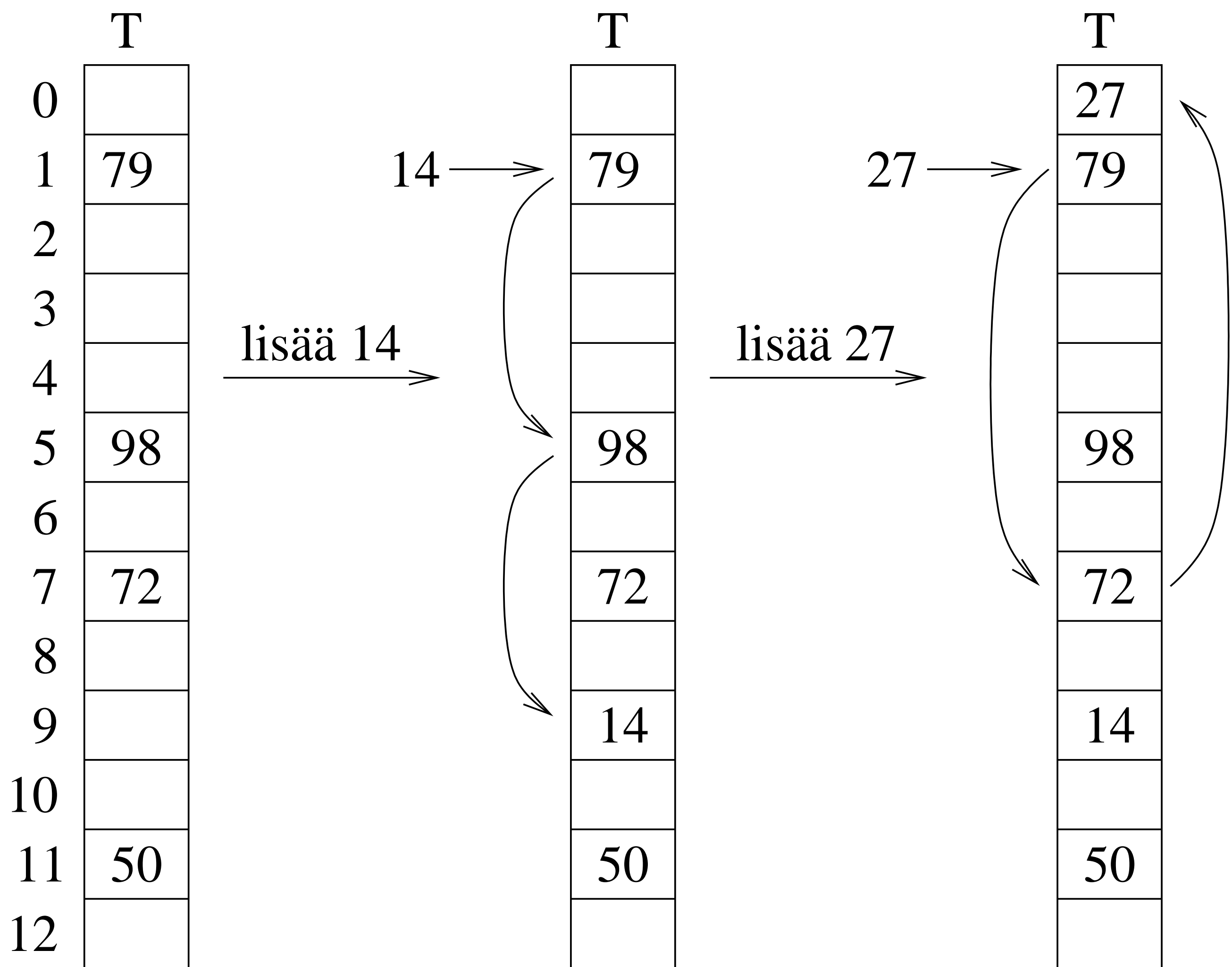
$$
\begin{aligned}
h'(27) &= 27 \bmod 13 \\
&= 1 \\
h''(27) &= 1 + (27 \bmod 11) \\
&= 6
\end{aligned}
$$

joten kokeilujono onkin

$$1, 7, 0, 6, 12, 5, 11, 4, 10, 3, 9, 2, 8.$$

- Siis lisätyt avaimet 14 ja 27
  - törmäävät yhteen kotiosoitteessa 1
  - seuraavat siitä alkaen eri kokeilujonoja.

- Seuraava kuva esittää lisäysten etenemisen.


T
T
T
0
1
2
3
4
5
6
7
8
9
10
11
12
79
98
72
50
lisää 14
14
79
98
72
14
50
lisää 27
27
27
79
98
72
14
50

- Kaksoishajautuksen ideana onkin:
  - Valitse hajautusfunktiot h' ja h''.
  - Silloin kahdella eri avaimella k_1, k_2 lienee

    $$h'(k_1) \neq h'(k_2) \quad \text{tai} \quad h''(k_1) \neq h''(k_2).$$

  - Silloin ne saavat eri kokeilujonot.
  - Näin vältetään kumpikin kasautuminen.

- Se onkin yleensä avoimen hajautuksen menetelmistä toimivin.

- Funktion h'' valinnassa on kuitenkin oltava huolellinen:
  - täytyy olla $h''(k) \neq 0$ jokaiselle avaimelle k
  - millään arvolla $h''(k)$ ei saa olla yhteisiä tekijöitä taulukon pituuden m kanssa
  - muuten osa hajautustaulusta voi jäädä käymättä läpi.

- Yksi hyvä valinta on edellisessäkin esimerkissä käytetty valinta:
  - m on alkuluku
  - $0 < h''(k) \underline{\leq m}$, esimerkiksi

    $$h''(k) = \underline{1+}(k \bmod m')$$

    missä $m' < m - 1$, vaikkapa $m' = m - 2$.

- Toinen mahdollisuus on
  - m pidetään luvun 2 potenssina

    $$m = 2^b$$

  - $h''(k)$ pidetään parittomana

    $$h''(k) = 2 \cdot h'''(k) + 1$$
    $$h'''(k) = \text{haluttu hajautusfunktio.}$$

4.3.4 Avoimen hajautuksen analyysistä

- Avoimen hajautuksen tapauksessa kaikkien operaatioiden tehokkuus on riippuvainen avaimen löytymisen tehokkuudesta.

- Seuraavassa taulukossa on tuloksettoman ja tuloksellisen haun askelmäärän odotusarvo kullakin kokeilujonotyypillä suhteessa täyttösuhteeseen

$$\alpha = \frac{n}{m} < 1.$$

tyyppi	tulokseton	tuloksellinen
4.3.3	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}$
4.3.2	$\frac{1}{1-\alpha} - \alpha + \ln \frac{1}{1-\alpha}$	$1 - \frac{2}{\alpha} + \ln \frac{1}{1-\alpha}$
4.3.1	$\frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right)$

- Tarkastellaan taulukon antamia tietoja muutamalla konkreettisella täyttösuhteen α arvolla.

$\alpha = 0.5$	tulokseton	tuloksellinen
4.3.3	2	1.38
4.3.2	2.19	1.44
4.3.1	2.5	1.5
$\alpha = 0.75$		
4.3.3	4	1.85
4.3.2	4.63	2.01
4.3.1	8.5	2.5
$\alpha = 0.9$		
4.3.3	10	2.55
4.3.2	11	2.88
4.3.1	50.5	5.5
$\alpha = 0.95$		
4.3.3	20	3.15
4.3.2	22	3.53
4.3.1	200.5	10.5

- Näyttää siltä että täyttöasteeseen $\alpha = 0.5$ asti suurta eroa menetelmien välillä ei ole
  - koska kasautuminen ei vielä vaivaa
  - koska puolet taulukosta on vielä tyhjänä.
- Tätä täydemmillä hajautustauluilla lineaarista kokeilujonoa ei enää kannata käyttää.
- Neliöinen kokeilu ja kaksoishajaus käyttäytyvät suunnilleen samoin.
- Tuloksettomat haut hidastuvat rajusti.

4.4 Taulukon pidentäminen

- Kalvojen 4.3 avoimessa hajautuksessa lisäys täyteen hajautusrakenteeseen on mahdotonta.

- Kalvojen 4.1 ketjutusratkaisussa hajautusrakenne toimii hajautusalueen täytyttyäkin.

- Molemmissa tapauksissa voimme *pidentää taulukkoa* kesken suorituksen:

  avoimessa hajautuksessa saadaksemme tilaa uusille tietueille

  ketjutuksessa jos valittu taulun pituus m osoittautuukin myöhemmin turhan pieneksi tietueiden lukumäärään n nähden.

- Pidennyksen vaiheet:

  1. Luodaan uusi aluksi tyhjä hajautustaulu $T'[0 \ldots m']$ jonka pituus $m' > m$.

  2. Käydään läpi nykyisen hajautustaulun $T[0 \ldots m]$ kaikki n tietuetta x, ja tehdään jokaiselle niistä operaatio hashInsert(T', x).

  3. Ryhdytään käyttämään taulua T' taulun T sijasta.

- Pidennys on raskas operaatio:

$$\Omega(m + n) = \Omega(n)$$

  askelta vaiheessa 2.

- Siis kannattaa pidentää

  – vain harvoin

  – eli kerralla reilummin.

- Osoittautuu hyväksi strategiaksi valita

$$m' = 2 \cdot m$$

eli *kaksinkertaistaa* taulun pituus joka pidennyskerralla.

- Intuitiivinen perustelu:
  - Jokainen operaatio hashInsert(T, x), joka ei vielä pidennä alkuperäistä taulukkoa T, maksakoon *3* EURoa.
  - 1 EURo kuluu tähän lisäykseen ilman pidennystä.
  - 2 EURoa *talletetaan pankkiin* pidennystä varten; pankissa pidetään aina vähintään 1 EURo / lisätty tietue.
  - Kun tulee aika pidentää taulukkoa, niin pankissa on $\geq 2 \cdot n$ EURoa
    - joista n EURolla maksetaan pidennysvaihe 2
    - ja loput $\geq n$ EURoa jäävät pankkiin pesämunaksi seuraavaa pidennystä odottamaan.

- Siis pidennyksen lisärasite onkin vain

  1 yksikkö <u>/ lisäysoperaatio</u>.

- Perustelu osoittaa siis, että
  - vaikka osa lisäysoperaatioista muuttuukin huomattavan hitaiksi

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

  - niin koko ohjelman suoritus hidastuu vain *vakio*kertoimen verran.

- Perustelu on esimerkki *tasoitetusta vaativuusanalyysistä:*
  - jos jokin operaatio uhrautuu tekemään raskaan lisätyön
  - jonka seurauksena monet muut operaatiot sujuvat nopeammin
  - niin on vain reilua, että muutkin operaatiot osallistuvat lisätyön kustannuksiin.

- Samalla tavalla voidaan perustella taulukon *lyhennys*strategia (jos muistia on säästettävä):

  – Kun täyttöaste putoaa

  $$\alpha \leq \frac{1}{4}$$

  – niin siirrytään käyttämään *puolitettua* taulukkoa

  $$T[0 \ldots \underbrace{\left\lfloor \frac{m}{2} \right\rfloor}_{m'}].$$

  – Myös puolitusehtoa

  $$\alpha \leq \frac{1}{3}$$

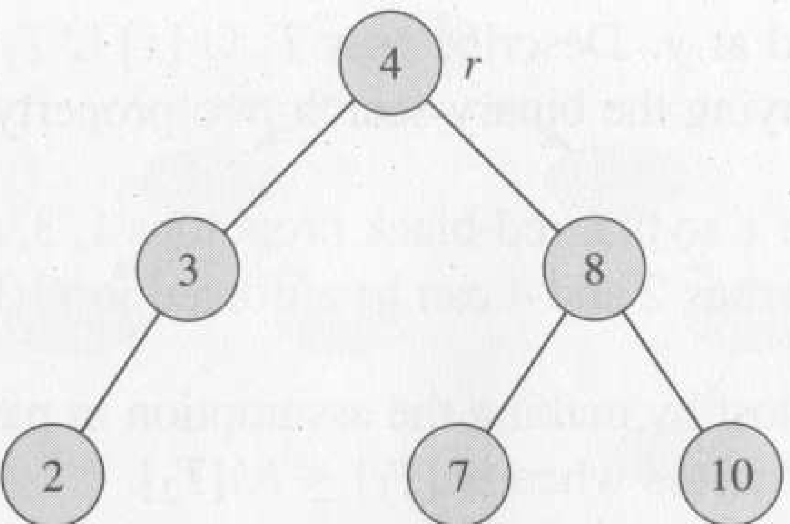
  voi käyttää:

  + muistia säästyy aggressiivisemmin

  – vakiokerroin kasvaa.

4.5 Käytännöllisiä huomautuksia

- *Hajautus on yleensä paras* menetelmä joukko-operaatioiden toteuttamiseen, jos
  - on helppo keksiä sopiva muunnosfunktio g avaimesta hajautettavaksi luvuksi (kalvojen 4.2.1 tapaan)
  - järjestystä käyttäviä operaatioita **min**/**max** ja **pred**/**succ** ei tarvita
  - voidaan hyväksyä, että vaikka operaatiot ovatkin tavallisesti nopeita, niin joskus mahdollisesti hyvinkin hitaita
  - voidaan hyväksyä, että hajautustaululle varattu mukautuu talletettavien tietueiden määrään viiveellä, eikä heti
  - ei tarvita perusoperaatiota ”tee toinen kopio tästä tietorakenteesta”.
- Muuten kannattaa käyttää tasapainoitettua puuta (esimerkiksi kalvoilta 3.3).



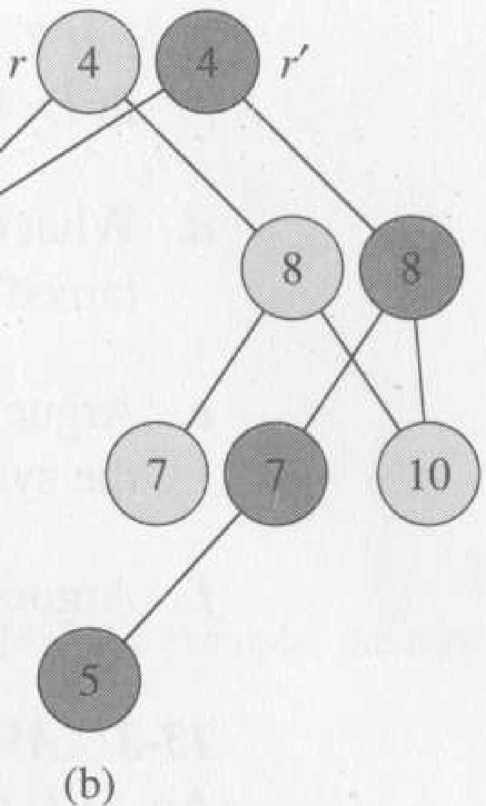


Figure 13.8 **(a)** A binary search tree with keys 2, 3, 4, 7, 8, 10. **(b)** The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root r', and the previous version consists of the nodes reachable from r. Heavily shaded nodes are added when key 5 is inserted.

- Edellinen kuva esittää periaatteen, miten puuta ei tarvitsekaan kopioida kokonaan.

  (Kuva 13.8 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

- Hajautusfunktioista kalvojen 4.2.2 *jakojäännösmenetelmä* on yleensä hyvä.

- Usein aineiston kanssa kannattaa tehdä muutamia kokeita eri hajautusfunktiokandidaateilla ennen kuin lopullinen valitaan.

- Jos

  - et voi tehdä kokeita, tai

  - kaikki keksimäsi funktiot osoittautuvat niissä huonoiksi

  niin kalvojen 4.2.4 universaali hajautus auttaa.

- Yhteentörmäysstrategioista kalvojen 4.1 *ketjutus* on yleensä paras ratkaisu:
  - toimii hajautusalueen täytyttyäkin
  - poistot aitoja, toisin kuin kalvojen 4.3 avoimessa hajautuksessa.

- Avoin hajautus on hyvä jos
  - täyttösuhde α voidaan pitää pienenä
  - haut yleensä onnistuvat.

  Esimerkki: käyttöjärjestelmän prosessitaulu.

  - Tietokoneen käynnistyessä varataan kiinteän kokoinen taulu:
    * Jos taulu täyttyy, niin uusia prosesseja ei enää voi käynnistää.
    * Yleensä prosesseja on käynnissä vain murto-osa varatusta maksimista.
    * Aina käsitellään käynnissä olevaa prosessia (ainoa poikkeus on, jos se onkin ehtinyt tällä välin kuolla).

5 Keko

- Tunnetaan myös nimellä ”kasa”.

- Tarkastellaan vielä yhtä tapaa toteuttaa kalvoilla 1.5 määritelty tietotyyppi joukko.

- Tällä kertaa emme kuitenkaan toteuta sen normaalia operaatiovalikoimaa.

- Olemme kiinnostuneita ainoastaan kolmesta operaatiosta:

  heapInsert(A**,**k**)** lisää kekoon A avaimen k

  heapMin(A**)** palauttaa keosta A sen *pienimmän* avaimen

  heapDelMin(A**)** poistaa ja palauttaa keosta A sen pienimmän avaimen.

- Nämä operaatiot tarjoavaa kekoa sanotaan *minimikeoksi* (engl. minheap)

- Toinen vaihtoehto, eli *maksimikeko* (engl. maxheap) taas tarjoaa operaatiot:

  heapInsert(A**,**k**)** lisää kekoon A avaimen k

  heapMax(A**)** palauttaa keosta A sen *suurimman* avaimen

  heapDelMax(A**)** poistaa ja palauttaa keosta A sen suurimman avaimen.

- Näitä kolmea operaatiota sanotaan (maksimi/minimi)*keko-operaatioiksi*.

- Keskitymme tässä luvussa ensisijaisesti maksimikekoon ja sen toteutukseen.

- Pystyisimme luonnollisesti toteuttamaan operaatiot käyttäen jo tuntemiamme tietorakenteita:

  Järjestämättömän listan (kalvot 2.4) operaatioilla insert, delete ja max

  heapInsert olisi vakioaikainen $\mathcal{O}(1)$

  heap(Del)Max olisivat lineaarisia $\mathcal{O}(n)$.

  Järjestetyn rengaslistan (kalvot 2.4.1) operaatioilla insert, delete ja max

  heap(Del)Max olisivat vakioaikaisia $\mathcal{O}(1)$

  heapInsert olisi lineaarinen $\mathcal{O}(n)$

  eli päinvastoin.

  Punamustien puiden (kalvot 3.3) operaatioilla insert, delete ja max

  kaikki keko-operaatiot olisivat logaritmisia $\mathcal{O}(\log n)$.

- Pystymme yhdistelemään niiden parhaat puolet: esittelemme tietotyypin keko toteutuksen, jossa

  heapMax on vakioaikainen $\mathcal{O}(1)$

  heapDelMax ja heapInsert ovat logaritmisia $\mathcal{O}(\log n)$.

- Mihin tarvitsemme näin erikoistunutta tietorakennetta?

  Eikö tasapainoitetettu hakupuu riittäisi, onhan sen tehokkuus melkein yhtä hyvä?

  Vain **heapMax** olisi hieman hitaampi.

- Tulemme kurssin aikana näkemään algoritmeja, jotka käyttävät aputietorakenteenaan kekoa.

  Niissä keon tehokkuus on oleellista.

- Vaikka keko ei tuokaan kertaluokkaparannusta operaatioihin, on se käytännössä hakupuuta huomattavasti nopeampi.

5.1 Keon käyttökohteita

- Keon avulla saamme toteutettua tehokkaasti *prioriteetti*jonon:
  - Kalvojen 2.2 tavallinen jono oli kaupan kassa:
    - uusi asiakas asettuu jonon perään
    - jonosta palvellaan aina ensimmäistä asiakasta.
  - Prioriteettijono on taas *ensiapupoliklinikka:*
    - potilaita palvellaan kiireellisyysjärjestyksessä
    - jos sisään tulevan potilaan vammat ovat vakavia, niin hän ohittaa jonossa vähemmän kiireelliset.
  - Operaatiossa **heapInsert** avain k on sisään tulevan potilaan kiireellisyys.
  - Operaatio **heapDelMax** ottaa kiireellisimmän potilaan hoitoon.

- Keon avulla saamme toteutettua erittäin tehokkaan järjestämisalgoritmin:
  - Syöte olkoon taulukkona $B[1 \ldots n]$.
  - Seuraava algoritmi järjestää taulukon B alkiot suuruusjärjestykseen käyttäen aluksi tyhjää kekoa A aputietorakenteena:

    heapSort(B, n)
    1 **for** $i \leftarrow 1$ **to** n **do**
    2   heapInsert($A, B[i]$)
    3 **for** $i \leftarrow 1$ **to** n **do**
    4     $B[i] \leftarrow$ heapDelMin(A)

  - Algoritmin aikavaativuus on

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

    eli aidosti parempi kuin neliöiset lisäys- ja kuplajärjestäminen (kalvoilta 1.1–1.3.2).

  - Palaamme tarkemmin kekojärjestämiseen kalvoilla 6.1.

- Prioriteettijonon käyttökohteita ovat esimerkiksi

  käyttöjärjestelmät kun valitaan seuraavaa prosessia suoritukseen

  verkkoalgoritmit joita esitellään kalvoilla 7.

5.2 Binäärikeko

Abstraktisti kekoa kannattaa ajatella binääripuuna (kalvot 3.1) joka täyttää seuraavan *kekoehdon:*

1. Isäsolmuun talletettu avain on aina *vähintään* yhtä suuri kuin sen lapsisolmuhin talletetut avaimet.

- Silloin koko puun juurisolmussa on aina suurin mahdollinen avain operaatiota **heapDelMax** varten.
- Toteutamme siis nyt *maksimi*kekoa.
- *Minimi*keossa ehto on kääntäen *korkeintaan*.
- Haluttu suorituskyky saavutetaan pitämällä puu tasapainossa

  eli sen korkeus logaritmisena

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

  solmujen eli avainten lukumäärän n suhteen.

Konkreettisesti tämä puu kannattaa tallettaa *taulukkona* linkityksen sijaan.

Määrittelemme puun tasapainoehdon siten, että taulukkototeutus on vaivatonta:

2. Puu on *mahdollisimman* täydellinen, eli
    - kaikki muut tasot paitsi alin ovat täynnä

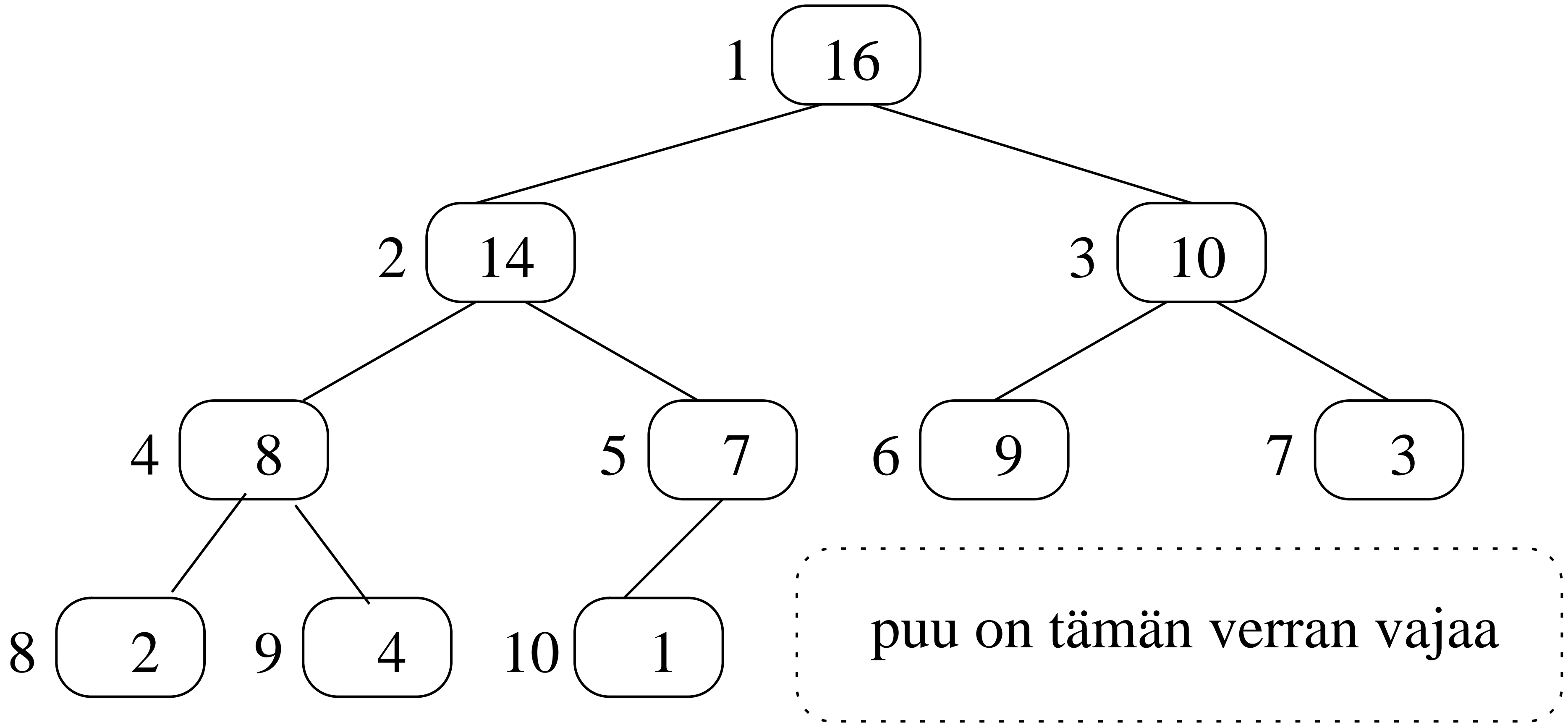
      eli niillä on maksimimäärä solmuja
    - alin taso on täytetty puun vasemmasta reunasta alkaen

      eli vain puun oikea alareuna saa olla vajaa.

- Näin saadaan *binäärikeko*.
- Seuraavassa kuvassa on esimerkkipuu, jossa solmun

  sisällä on sen avain

  vieressä on sen taulukkopaikka.
- Taulukkopaikat on jaettu ylläolevan täyttöehdon 2 mukaisessa järjestyksessä.


1 16
2 14
3 10
4 8
5 7
6 9
7 3
8 2
9 4
10 1
puu on tämän verran vajaa

- Kekotaulukkoon A liittyy kaksi attribuutti:

  length$[A]$ = taulukon pituus

  heapSize$[A]$ = keossa olevien avainten lukumäärä.

- Taulukon alkuosa

  $$A[\underline{1} \ldots \text{heapSize}[A]]$$

  on tällä hetkellä käytössä.

- Huomaa: indeksointi alkaa ykkösestä!

- Taulukon loppuosa

  $$A[\text{heapSize}[A] + 1 \ldots \text{length}[A]]$$

  on tällä hetkellä tyhjänä odottamassa uusia avaimia.

- Tietenkin koko ajan

  $$\text{heapSize}[A] \leq \text{length}[A]$$

  jotta talletettu keko pysyy taulukon A sisällä.

- Keon juurialkio on talletettu taulukon ensimmäiseen paikkaan $A[1]$.

- Puu talletetaan taulukkoon siten, että paikkaan $A[i]$ talletetusta solmusta pääsee sen isään ja lapsiin seuraavilla indeksin i laskutoimituksilla:

parent(i)
  return $\lfloor \frac{i}{2} \rfloor$
left(i)
  return $2 \cdot i$
right(i)
  return $2 \cdot i + 1$

  – Juuressa (ja vain siellä) on

$$\text{parent}[1] = 0.$$

  – Jos

$$\text{left}(i) > \text{heapSize}[A]$$

  niin paikan i solmulla ei ole vasenta lasta.

  – Sama ehto pätee myös oikealle lapselle.

- Vaikka varsinaisia linkkejä ei ole, on keossa liikkuminen todella helppoa:
  - Edellisen kuvan esimerkkipuun taulukkototeutus on seuraavana kuvana.
  - Solmun $A[5]$

    vanhempi on paikassa

$$\begin{aligned} A[\text{parent}(5)] &= A\left[\left\lfloor\frac{5}{2}\right\rfloor\right] \\ &= A[2] \end{aligned}$$

    vasen lapsi on paikassa

$$\begin{aligned} A[\text{left}(5)] &= A[2 \cdot 5] \\ &= A[10] \end{aligned}$$

    oikea lapsi olisi paikassa

$$\begin{aligned} A[\text{right}(5)] &= A[2 \cdot 5 + 1] \\ &= A[11] \end{aligned}$$

    mutta sellaista solmua ei ole, koska

$$\begin{aligned} \text{heapSize}[A] &= 10 \\ &< 11. \end{aligned}$$

1	2	3	4	5	6	7	8	9	10	11	12
16	14	10	8	7	9	3	2	4	1		

juuri ¦ *taso 1* ¦ *taso 2* ¦ *taso 3*

- Tämä solmujen tallennusjärjestys saadaan *bittioperaatioina* siten, että kun puussa käännytään

  vasemmalle niin indeksin i perään liitetään bitti $\mathbf{0}$

  oikealle niin bitti $\mathbf{1}$

  eli indeksi $i =$ polku sitä vastaavaan solmuun.

- Seuraava kuva havainnollistaa tätä ajatusta käyttäen esimerkkipuutamme.

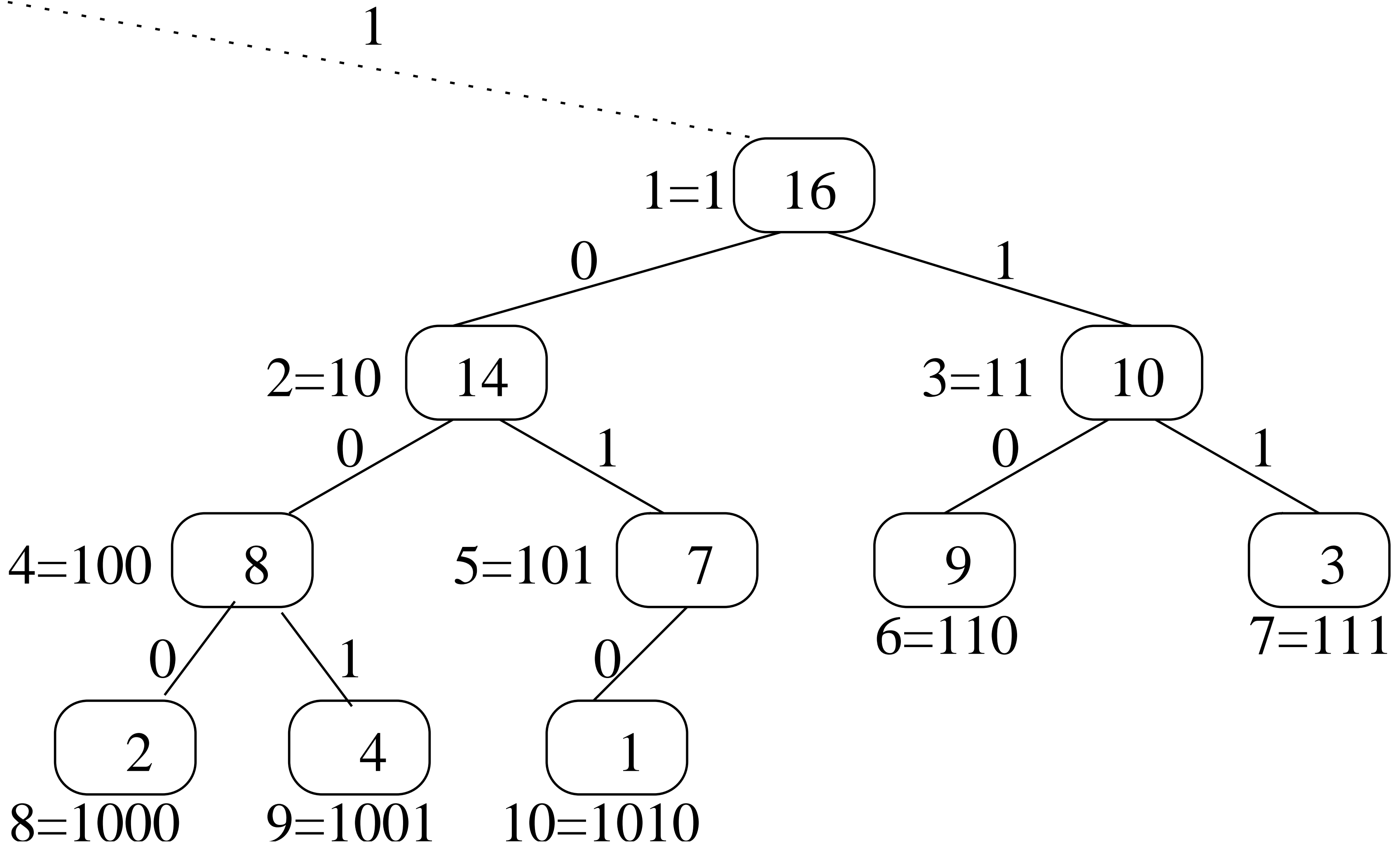
- Nyt voimme lausua kekoehdon 1 seuraavasti:

  kaikille indekseille

  $$1 < i \leq \text{heapSize}[A]$$

  pätee

  $$A[\text{parent}[i]] \geq A[i]. \tag{4}$$


1
1=1
16
0
1
2=10
14
3=11
10
0
1
0
1
4=100
8
5=101
7
9
3
6=110
7=111
0
1
0
2
4
1
8=1000
9=1001
10=1010

- Ennen kuin voimme toteuttaa varsinaiset keko-operaatiot, toteutamme seuraavan apuoperaation:

  - Parametreina saadaan taulukko A ja sen luvallinen indeksi $1 \leq i \leq$ heapSize$[A]$.

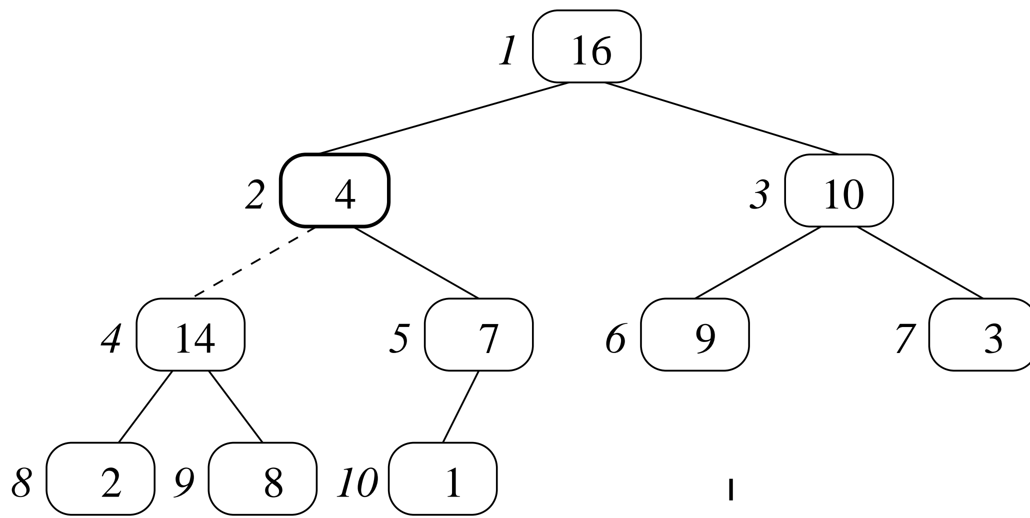
  - Oletuksena on että lapsista left(i) ja right(i) alkavat alipuut ovat jo kekoja ja taulukossa A.

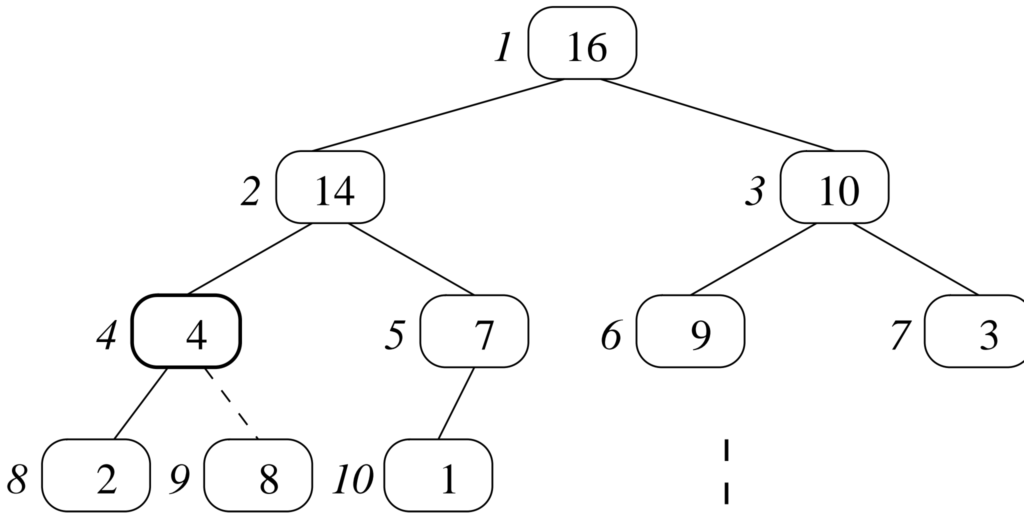
  - Operaation jälkeen myös indeksistä i alkava alipuu on keko ja taulukossa A.

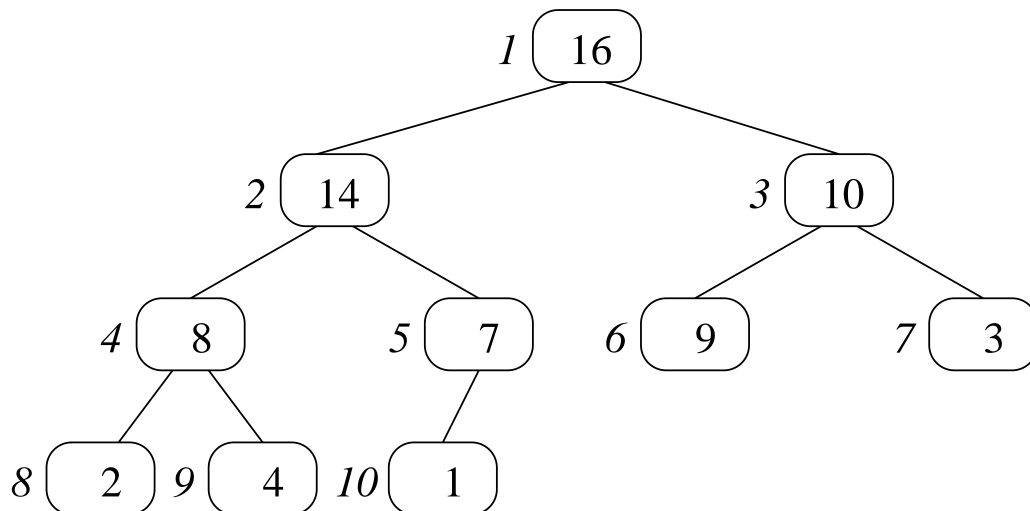
  - Toisin sanoen, myös $A[i]$ on päätynyt sopivaan paikaan kyseisessä alipuussa.

heapify(A, i)
1 $l \leftarrow$ left(i)
2 $r \leftarrow$ right(i)
3 **if** $r \leq$ heapsize$[A]$ **then**
4   **if** $A[l] > A[r]$ **then** $j \leftarrow l$ **else** $j \leftarrow r$
5   **if** $A[i] < A[j]$ **then**
6   vaihda $A[i] \leftrightarrow A[j]$
7   heapify(A, j)
8 **else if** $l =$ heapSize$[A]$ **and** $A[i] < A[l]$
9   **then** vaihda $A[i] \leftrightarrow A[l]$

- Rivillä 3 tutkitaan ovatko molemmat lapset olemassa.
  - Täyttöjärjestyksen 2 nojalla tämä on sama ehto kuin tutkia onko oikea lapsi r olemassa.
  - Rivillä 4 $j =$ lapsista l ja r suurempi.
  - Rivillä 5 tarkistetaan täyttävätkö $A[i]$ ja $A[j]$ kekoehdon 1 muodossa (4).
  - Jos eivät, niin
    * vaihdetaan ne keskenään (rivi 6)
    * jatketaan keon korjailua (rivi 7).
- Jos vain vasen lapsi l on olemassa, niin tehdään sen suhteen sama työ kuin edellä (rivit 8–9).
- Seuraava kuvasarja valottaa operaation toimintaa.


1
16
2
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
heapify(A,2)


1
16
2
14
3
10
4
4
5
7
6
9
7
3
8
2
9
8
10
1
heapify(A,4)


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1

- Operaation heapify suoritusaika riippuu ainoastaan puun korkeudesta:
  - rekursiivisia kutsuja tehdään pahimmassa tapauksessa puun korkeuden verran
  - n-alkioisen keon korkeus on selvästi

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

    sillä keko on lähes täydellinen binääripuu
  - operaation pahimman tapauksen aikavaativuus on siis logaritminen.

- Kekoehdosta 1 seuraa suoraan että keon maksimialkio on talletettu paikkaan A[1], joten operaatio

  heapMax(A)
      return $A[1]$

  on triviaali ja vie vakioajan $\mathcal{O}(1)$.

- Maksimialkion poistaminen epätyhjästä keosta A:

heapDelMax(A)
1 $p \leftarrow A[1]$
2 $A[1] \leftarrow A[\text{heapSize}[A]]$
3 $\text{heapSize}[A] \leftarrow \text{heapSize}[A] - 1$
4 heapify($A, 1$)
5 **return** p

  - Palauttaa kohdassa $A[1]$ olleen avaimen (rivit 1 ja 5).

  - Keon viimeisessä paikassa oleva alkio $A[\text{heapSize}[A]]$ viedään poistetun alkion $A[1]$ tilalle (rivi 2).

  - Keon koko pienennetään yhdellä (rivi 3).

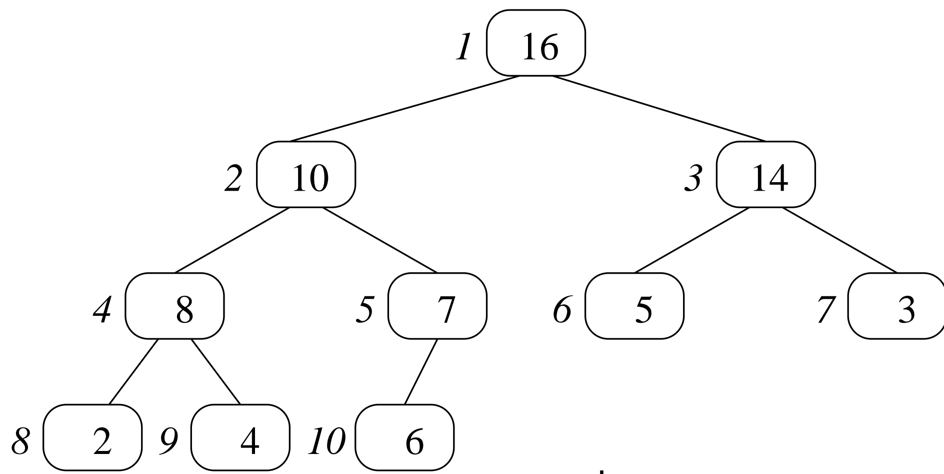
  - Keko on muuten kunnossa, mutta kohtaan $A[1]$ siirretty avain saattaa rikkoa keko-ominaisuuden:

    kutsutaan operaatiota heapify korjaamaan tilanne (rivi 4).

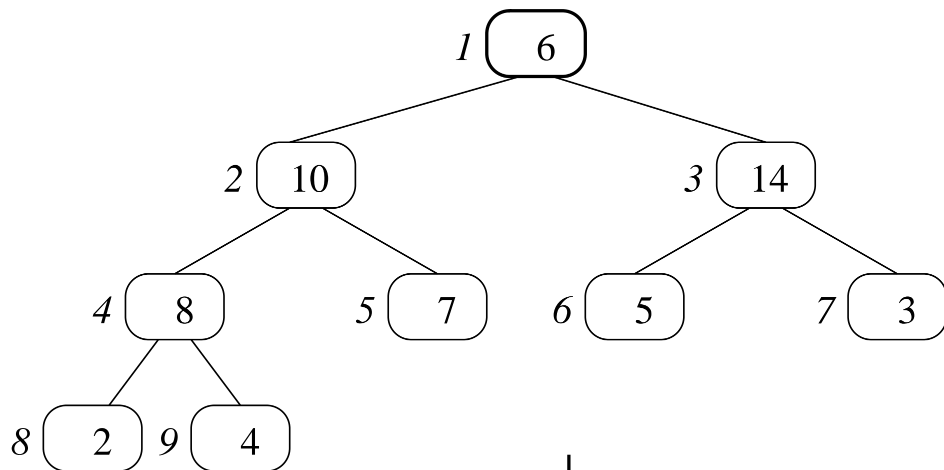
- Poisto-operaation aikavaativuus on sama kuin apuoperaatiolla heapify, eli

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

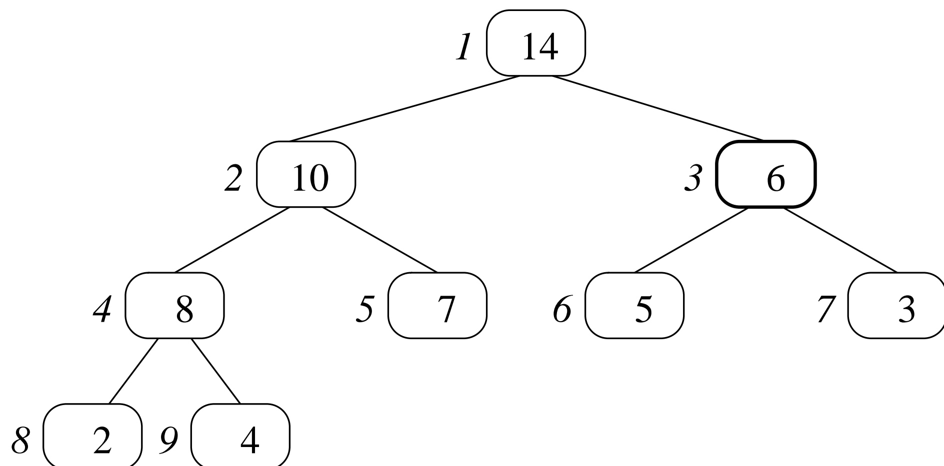
- Seuraava kuvasarja on esimerkki poisto-operaation toiminnasta.



1 16
2 10
3 14
4 8
5 7
6 5
7 3
8 2
9 4
10 6
heapDelMax(A)



1 6
2 10
3 14
4 8
5 7
6 5
7 3
8 2
9 4
heapify(A,1)



1 14
2 10
3 6
4 8
5 7
6 5
7 3
8 2
9 4

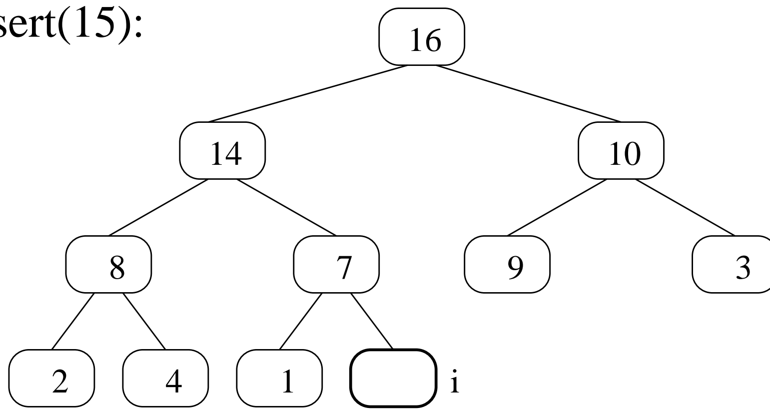
- Avaimen k lisääminen kekoon A:

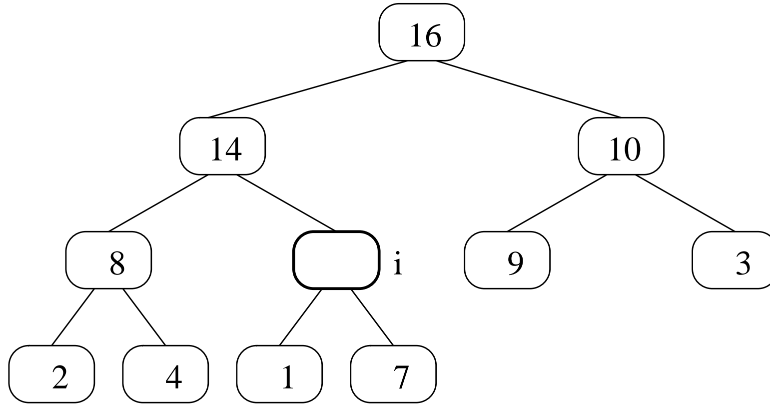
```
heapInsert(A, k)
1  i ← heapSize[A] + 1
2  heapSize[A] ← i
3  while i > 1 and A[parent(i)] < k do
4      A[i] ← A[parent(i)]
5      i ← parent(i)
6  A[i] ← k
```

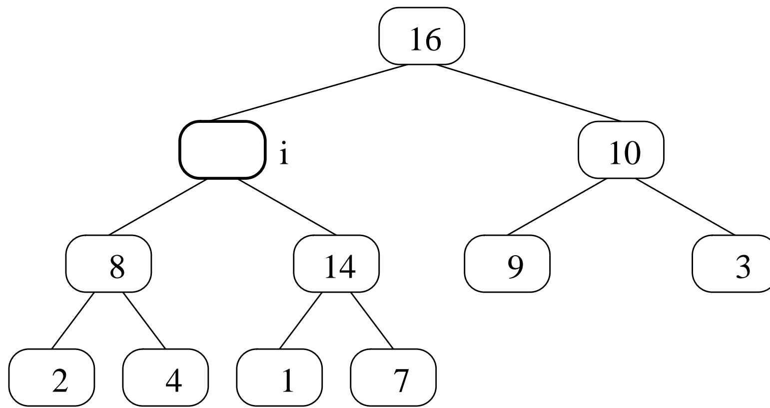
  - Kasvatetaan keon kokoa yhdellä solmulla eli tehdään paikka i uudelle avaimelle (rivit 1–2).
  - Nostetaan tätä tyhjää paikkaa i puussa ylöspäin (rivit 3–5)
    * siirtämällä sen tieltä täyden isäpaikan sisältö alaspäin (rivi 4)
    * kunnes se päätyy kohtaan, jossa kekoehto 1 jälleen pätee muodossa (4).
  - Noston aikana tyhjän paikan i avaimeksi oletetaan k

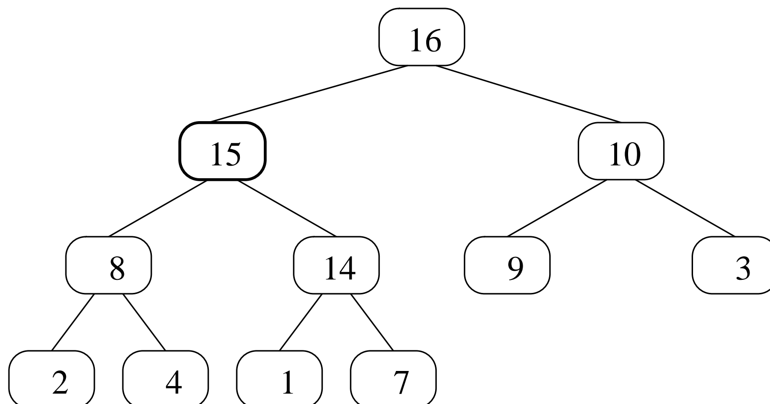
    ja noston loputtua (rivillä 6) asetetaan näin oikeastikin.

heapInsert(15):









- Edellisellä kalvolla on esimerkki lisäysoperaation toiminnasta.

- Pahimmassa tapauksessa avain lisätään juureen, jolloin puun korkeudellisen verran avaimia on valutettu alaspäin.

- Operaation aikavaativuus on siis logaritminen

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

avainten lukumäärän n suhteen.

- Lisäysoperaatio olettaa, että taulukko A ei vielä ole täynnä, eli

$$\text{heapSize}[A] < \text{length}[A].$$

- Jos taulukko A onkin täynnä, niin voidaan soveltaa kalvoilla 4.4 esitettyä dynaamista pidentämistä.

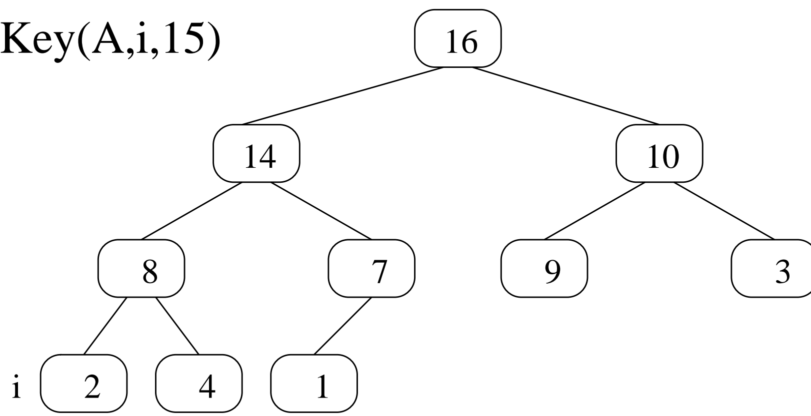
5.3 Muita operaatioita

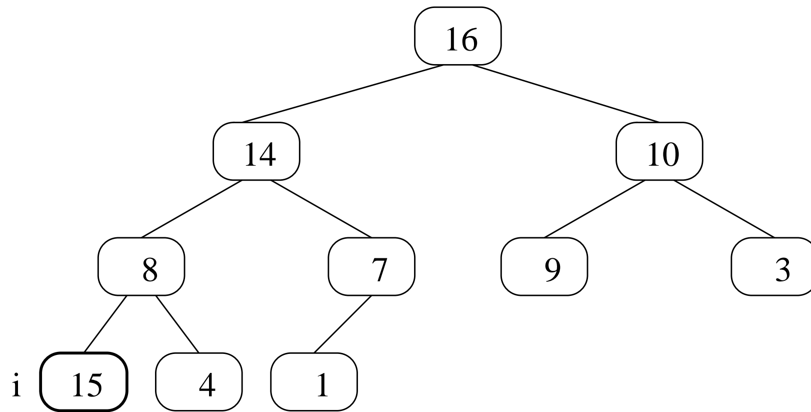
- Jotkut sovellukset tarvitsevat operaatiota, joka *kasvattaa* kekopaikassa $A[i]$ olevan avaimen uuteen arvoon $k > A[i]$:

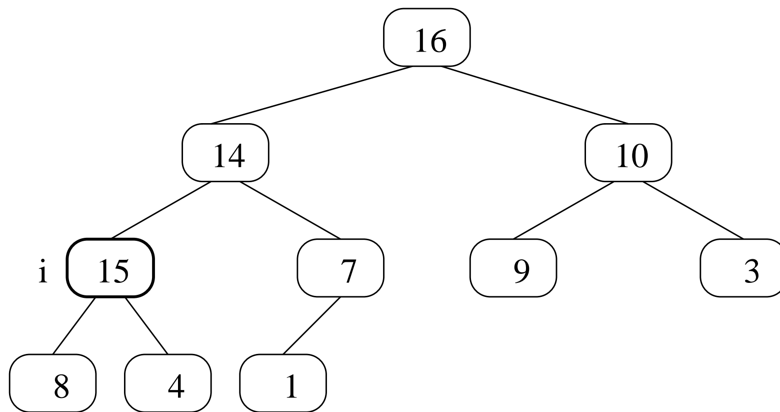
  heapIncKey(A, i, k)
  1 **if** $k > A[i]$ **then**
  2     $A[i] \leftarrow k$
  3   **while** $i > 1$ **and** A[parent(i)] $< A[i]$ **do**
  4   vaihda $A[i] \leftrightarrow A$[parent(i)]
  5         $i \leftarrow$ parent(i)

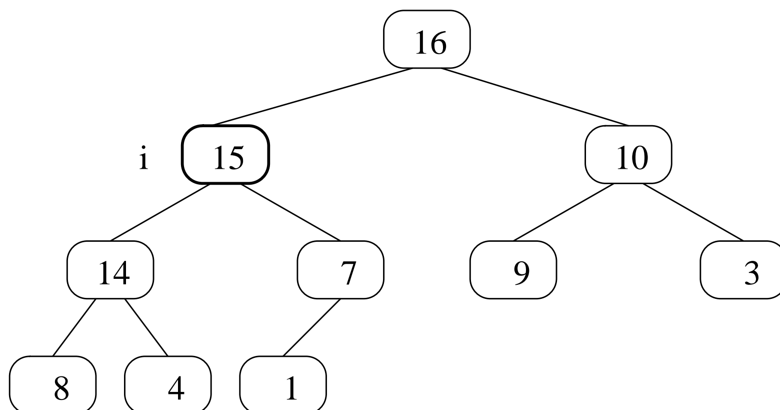
  - Jos yritetään pienentää avaimen arvoa, operaatio ei tee mitään (rivi 1)
  - Kopioidaan keon kohtaan i uusi avaimen arvo (rivi 2).
  - Jos kasvanut $A[i]$ rikkoi kekoehdon 1 muodossa (4), niin
    - vaihdetaan $A[i]$ isänsä kanssa (rivi 4)
    - kunnes ehto on jälleen kunnossa (rivit 5 ja 3).

heapIncKey(A,i,15)









- Jos täytyykin *pienentää* keon A paikassa i olevaa avainta:

  heapDecKey(A, i, k)
  1 **if** $k < A[i]$ **then**
  2 $A[i] \leftarrow k$
  3 heapify(A, i)

- Jos keosta A halutaan poistaa sen paikassa i oleva avain, joka ei siis olekaan suurin:

  heapDel(A, i)
  1 heapIncKey($A, i, +\infty$)
  2 heapDelMax(A)

  - Toimintaperiaate:

    1. Ensin tehdään siitä ''väkivalloin'' suurin.

    2. Sitten poistetaan se normaalisti.

- Tämä tapa kuitenkin siirtelee *kahta* keon avainta:

  1. poistettava avain nousee ensimmäiseen paikkaan $A[1]$

  2. ja korvautuu viimeisellä avaimella A[heapSize[A]] jonka

     – taulukkopaikka vapautetaan

     – arvo laskeutuu uudelle paikalleen taulukossa.

- Riittää siirrellä *yhtä* avainta:

  1. korvaa suoraan poistettava $A[i]$ vapautuvalla avaimella A[heapSize[A]]

  2. ja siirrä korvaava avain uudelle paikalleen

     nostamalla ylöspäin jos se kasvoi

     laskemalla alaspäin jos väheni.

- Syötteenä annetuista n avaimesta voidaan luoda keko lisäämällä ne aluksi tyhjään kekoon yksi kerrallaan.
  - Aikavaatimus olisi
$$\mathcal{O}(n \cdot \log n).$$
  - Esimerkkinä rivit 1–2 kalvojen 5.1 kekojärjestämisalgoritmissa.

- Kuitenkin seuraava tapa tekee saman säästäen

  tilaa: avaimet voivat olla suoraan taulukossa A — niiden alkuperäisellä järjestyksellä ei ole väliä

  aikaa: aikavaatimus onkin vain lineaarinen
$$\mathcal{O}(n).$$

buildHeap(A)
1 heapSize[A] $\leftarrow$ length[A]
2 **for** $i \leftarrow \left\lfloor \frac{\text{length}[A]}{2} \right\rfloor$ **downto** 1 **do**
3 heapify(A, i)

- Tavan perustelu:
  - Kalvojen 5.2 apufunktiota heapify(A, i) saa kutsua, kunhan paikasta $A[i]$ alkavan kekopuun molemmat alipuut $A[\text{left}(i)]$ ja $A[\text{right}(i)]$ ovat jo kekoja.
  - Kutsun tuloksena koko paikasta $A[i]$ alkava alipuu on nyt itsekin keko.
  - Siis käydäänkin läpi

    puun solmut alhaalta ylöspäin

    vastaavat taulukkopaikat lopusta alkuun

    $$A[\text{length}[A]], A[\text{length}[A]-1], \ldots, A[1].$$

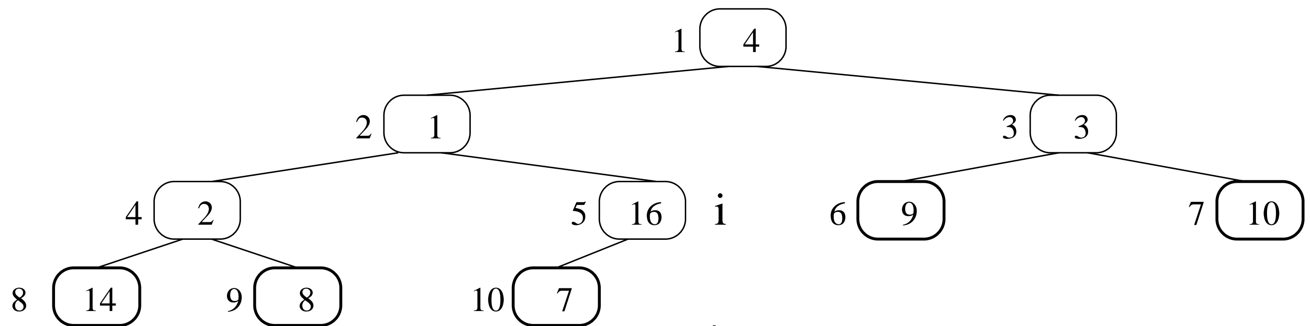
  - Näistä paikoista ensimmäinen puolisko

    $$\left\lceil \frac{\text{length}[A]}{2} \right\rceil \text{ kappaletta}$$

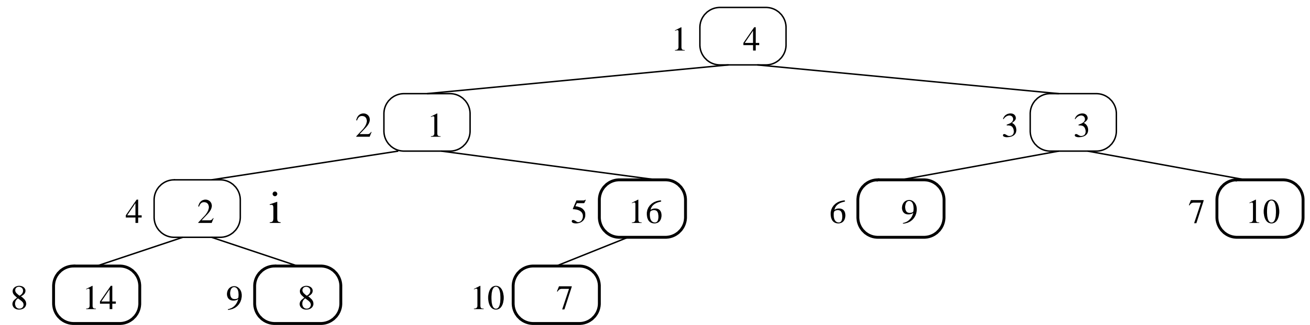
    ovat kuitenkin lapsettomia solmuja, joten ne voidaankin hypätä yli.

- Seuraava kuvasarja valottaa tavan toimintaa.

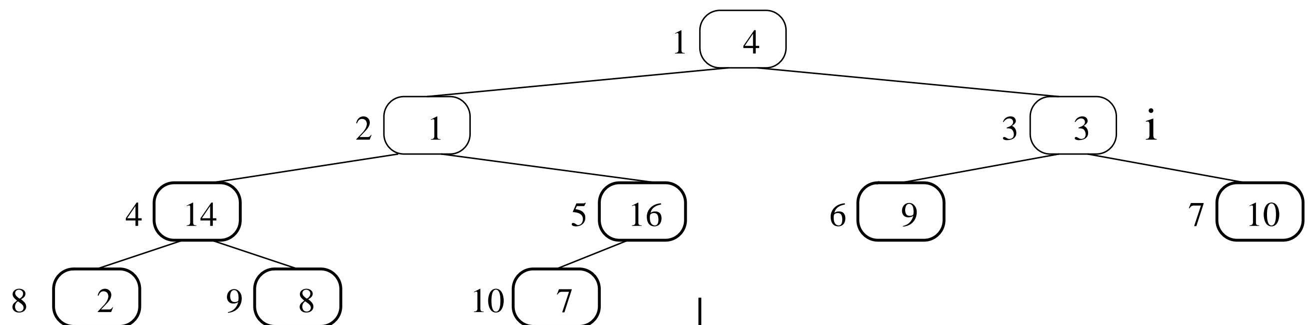
	1	2	3	4	5	6	7	8	9	10
alussa	4	1	3	2	16	9	10	14	8	7



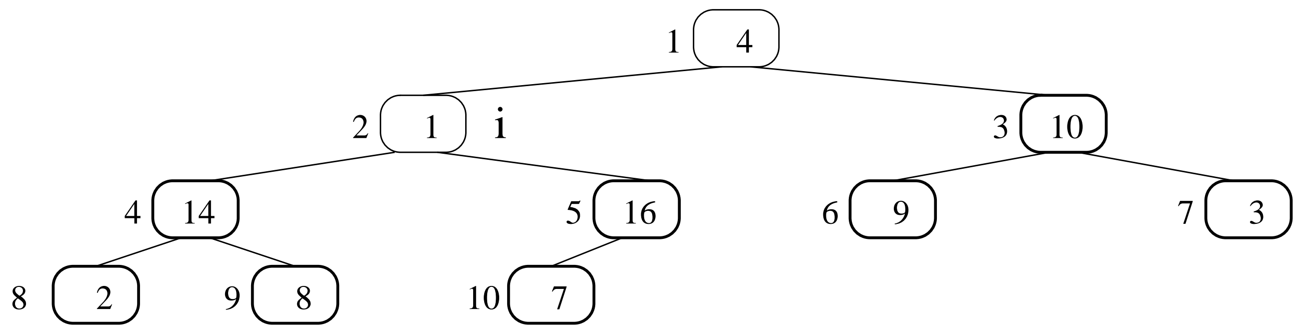
heapify(5)



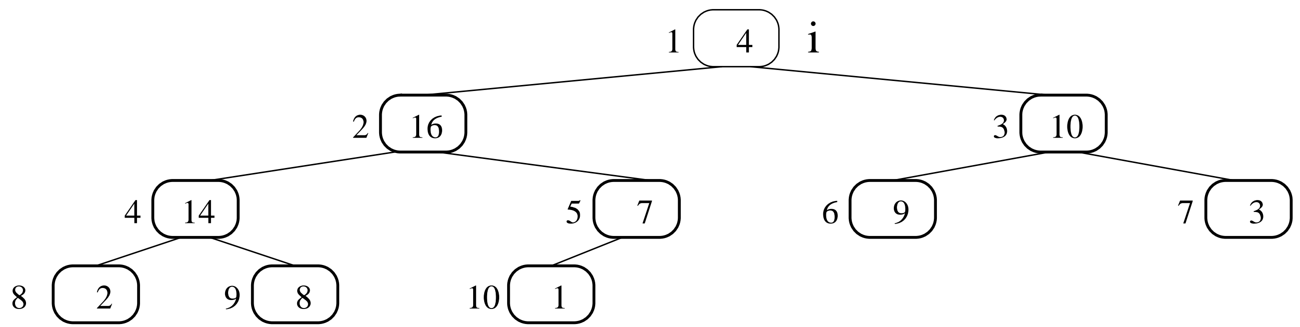
heapify(4)



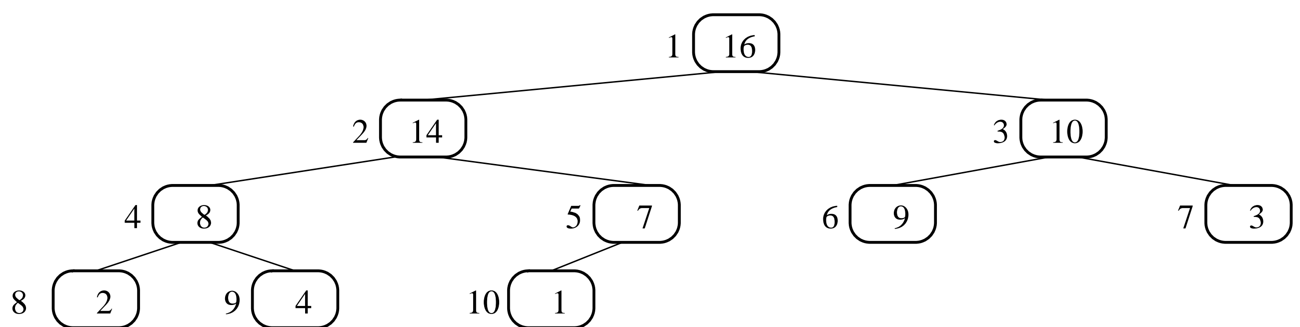
heapify(3)



heapify(2); heapify(5)



heapify(1); heapify(2); heapify(4)



	1	2	3	4	5	6	7	8	9	10
tuloksena	16	14	10	8	7	9	3	2	4	1

- Entäs lineaarisuus? Eikö tämäkin olekaan sama

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

  - silmukkahan suoritetaan $\left\lfloor \frac{n}{2} \right\rfloor$ kertaa
  - joka kerralla tehdään $\mathcal{O}(\log n)$ kutsu heapify(A, i)?

- Tehdäänkin tarkempi analyysi:
  - Yhden kutsun askelmäärä on $\mathcal{O}(h)$ missä $h =$ solmun $A[i]$ korkeus kekopuussa.
  - Binääripuussa korkeudella h on $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$ solmua.

    (Voidaan todistaa induktiolla yli korkeuden h; sivuutetaan.)
  - Kekopuu on melkein tasapainoinen, joten sen (juuren) korkeus on $\lfloor \log n \rfloor$.

- Näin askelmäärälle saadaan yläraja

$$\sum_{h=0}^{\lfloor \log n \rfloor} h \cdot \left\lceil \frac{n}{2^{h+1}} \right\rceil < \sum_{h=0}^{\lfloor \log n \rfloor} h \cdot \frac{n}{2^h}$$

(Pyöristys ylöspäin voidaan poistaa karkealla arviolla: $\lceil y \rceil \leq 2 \cdot y$ kun $y \geq \frac{1}{2}$.)

$$= n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} h \cdot \left(\frac{1}{2}\right)^h$$

$$< n \cdot \underbrace{\sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^h}_{=2}.$$

- Käytimme yllä lopuksi aputulosta

$$\sum_{h=0}^{\infty} h \cdot x^h = \frac{x}{(1-x)^2} \qquad \text{kun } |x| < 1 \qquad (5)$$

arvolla

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

- Aputulos (5) voidaan johtaa lukiomatematiikan tiedoin seuraavasti:
  - Äärettömän geometrisen sarjan summa on
    $$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \qquad \text{kun } |x| < 1.$$
    (Käytimme sen äärellistä tapausta jo lauseen 3.2 todistuksessa.)
  - Derivoidaan se muuttujan x suhteen:
    $$\sum_{k=1}^{\infty} k \cdot x^{k-1} = \frac{1}{(1-x)^2}.$$
  - Kerrotaan se muuttujasymbolilla x:
    $$\sum_{k=1}^{\infty} k \cdot x^{k} = \frac{x}{(1-x)^2}.$$
  - Lisätään siihen $0 \cdot x^0 = 0$:
    $$\sum_{k=0}^{\infty} k \cdot x^{k} = \frac{x}{(1-x)^2}.$$
- Tämä on esimerkki *generoivien funktioiden* ja *formaalien potenssisarjojen* käytöstä algoritmianalyysissä.

5.4 Kahvat keon sisälle

- Monissa käytännön sovelluksissa, kuten käytettäessä kekoa prioriteettijonoa, keottavat alkiot sisältävät muitakin kenttiä kuin pelkän avaimen.

- Tällöin itse kekoon kannattanee tallettaa vain pareja

  - avain

  - viite lisätiedot tallettaneeseen datatietueeseen

  (tai jopa avainkin voi olla datatietueessa).

- Vertaa kalvojen 3.4 erottelu hakupuun ja datatietueen välillä.

- Jos esimerkiksi toteuttaisimme käyttöjärjestelmässä prosessien skedulointijonon käyttäen kekoa, niin koko prosessikuvaajaa ei kannattane viedä kekoon.

- Tällaisessa käytännön tilanteessa keko-operaatioiden parametrit kannattanee valita seuraavasti:

  heapInsert(A,x,k) lisää kekoon A datatietue(viitte)en x avaimella k.

  heapMax(A) palauttaa viitteen siihen keon A datatietueeseen, jonka avain on suurin.

  heapDelMax(A) poistaa ja palauttaa sen.

  heapIncKey(x,k') kasvattaa datatietue(viitte)eseen x liittyvän avaimen uudeksi arvoksi k'.

- Jotta operaatio **heapIncKey** saadaan toteutettua tehokkaasti, niin datatietueista on myös oltava *viite takaisin* vastaavaan kekoalkioon.

- Tätä viitettä takaisin kutsutaan joskus *kahvaksi* (engl. handle).

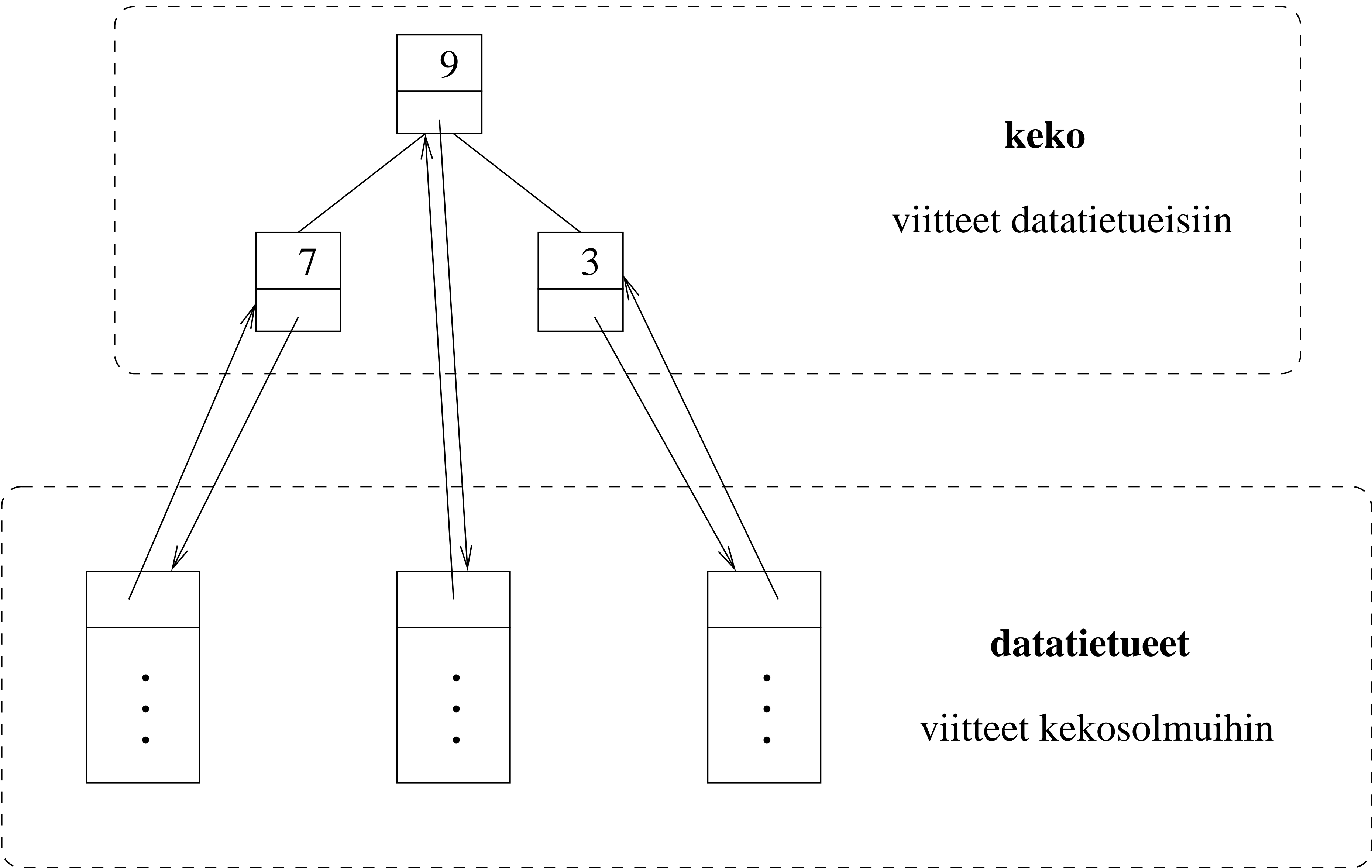
- Muistin organisointi näyttää esimerkiksi seuraavan kuvan mukaiselta.

- Datatietueen x kahva handle[x] takaisin sitä vastaavaan kekoalkioon
  - on käytännössä kekoalkion nykyinen indeksi i kekotaulukossa A, eli
  - täyttää siis invariantin ”kekotaulukkopaikan A[handle[x]] viite osoittaa takaisin tämän samaan tietueeseen x itseensä”
  - jota täytyy siis ylläpitää kaikissa keko-operaatioissa aina kun alkiot vaihtavat paikkaa kekotaulukon A sisällä.

- Silloin datatietuetta käyttävä operaatio **heapIncKey**(x, k') palautuu kalvojen 5.3 taulukkoa käyttäväksi operaatioksi heapIncKey(A, handle[x], k')

  joka vuorostaan pitää yllä tätä invarianttia rivillään 4.

  Muut operaatiot vastaavasti.


9
7
3
keko
viitteet datatietueisiin
datatietueet
viitteet kekosolmuihin

6 Järjestäminen

- Osaamme jo muutamia neliöisessä ajassa
$$\mathcal{O}\left(n^2\right)$$
toimivia menetelmiä, jotka järjestävät n lukua suuruusjärjestykseen:

  lisäysjärjestäminen kalvoilta 1.1

  kuplajärjestäminen kalvoilta 1.3.2.

- Tarkastellaan tässä luvussa muutamaa menetelmää joilla järjestämisessä päästään asymptoottisesti parempaan aikaan
$$\mathcal{O}\left(n \cdot \log n\right).$$

- Eräs menetelmä olisi tietenkin

  1. luoda luvuista tasapainoinen (kuten kalvojen 3.3 punamustaan) hakupuu

  2. tulostaa se sisäjärjestyksessä

  mutta riittäisivätkö yksinkertaisemmat aputietorakenteet?

6.1 Kekojärjestäminen

- Eräs menetelmä saadaan korvaamalla hakupuu kekopuulla.

- Kalvoilla 5.1 hahmottelimme jo idean miten kekoa voidaan käyttää saamaan ajassa

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

  toimiva järjestämisalgoritmi.

- Esitetään tässä käytännön tasolla hieman tehokkaammin toimiva versio kekojärjestämisestä.

- Kalvoilla 5.2 esittelimme
  - apuoperaation heapify(A, i)
  - sen avulla tehdyn nopean muunnoksen buildHeap(A) mielivaltaisesta taulukosta A keoksi.

- Kekojärjestäminen tapahtuu niitä käyttäen seuraavasti:

```
heapSort(A)
1  buildHeap(A)
2  while heapSize[A] > 1 do
3      vaihda A[1] ↔ A[heapSize[A]]
4      heapSize[A] ← heapSize[A] − 1
5      heapify(A, 1)
```

  - Aineistosta A tehdään ensin *maksimi*keko (rivi 1).

  - Vaihdetaan keskenään keon suurin ja viimeinen alkio (rivi 3).

  - Suurin alkio siirtyy oikealle paikalleen.

  - Pienennetään kekoa, jotta sitä ei enää käsitellä uudelleen (rivi 4).

  - Palautetaan kekoehto 1 voimaan myös paikassa $A[1]$ (rivi 5).

  - Toistetaan kunnes keko koostuu vain paikasta $A[1]$ jossa nyt on syötteen pienin alkio (rivi 2).

Aikavaativuus on jo todettu.

Tilavaativuus:

- Operaatio heapify kutsuu rekursiivisesti itseään pahimmillaan kekopuun korkeuden

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

  verran.

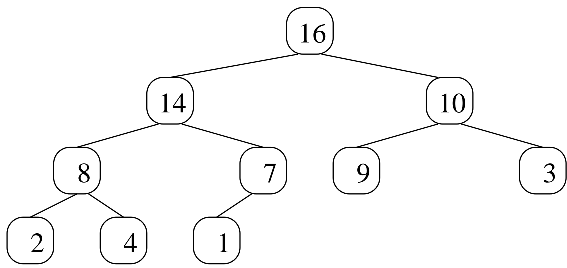
- Operaation rekursio on
  - kalvojen 3.1.3 takarekursiota
  - muutettavissa iteraatioksi

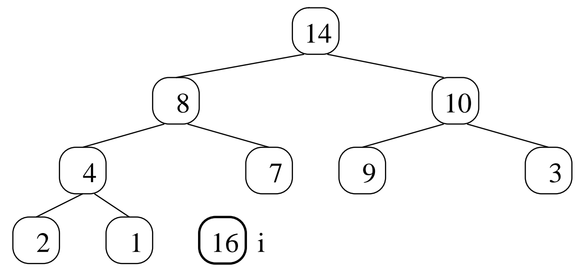
  jolloin sen tilantarve on vakio.

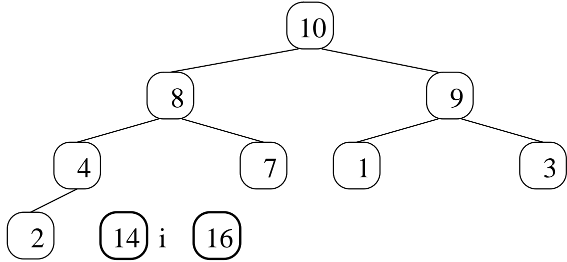
- Muutkin kekojärjestämisen toimet onnistuvat vakiotilassa

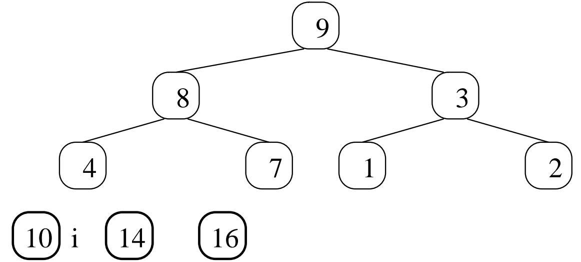
$$\mathcal{O}(1).$$

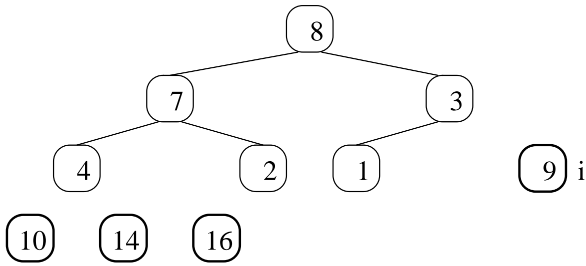
- Seuraavassa kuvassa on esimerkki.

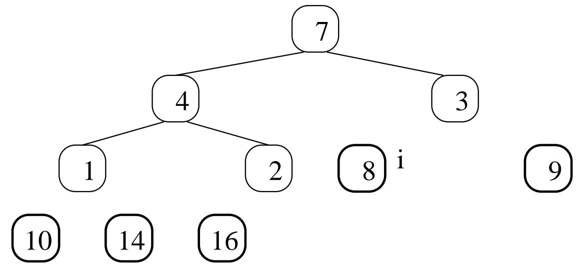


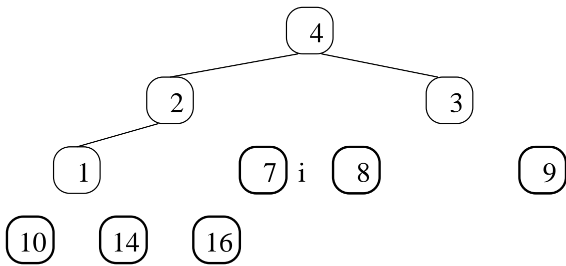


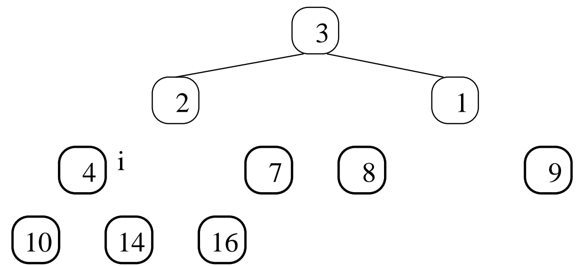


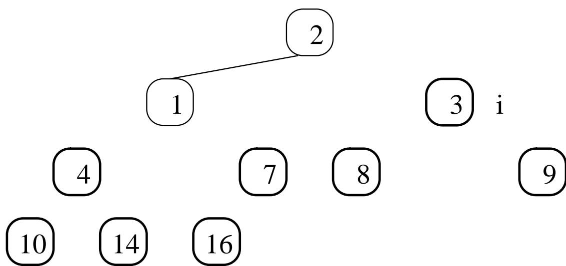


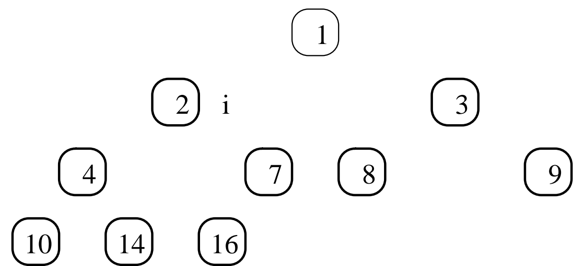












tuloksena

1	2	3	4	7	8	9	10	14	16

6.2 Lomitusjärjestäminen

- Lomitusjärjestäminen perustuu *hajoita-ja-hallitse* (engl. divide-and-conquer) tekniikkaan:

  hajoitetaan ongelma pienempiin osaongelmiin

  hallitaan eli ratkaistaan osaongelmat *rekursiivisesti*

  yhdistetään osaratkaisut siten että saadaan ratkaisu koko ongelmalle.

- Syötetaulukko $A[1 \ldots n]$ järjestetään kutsumalla

  $$\text{mergeSort}(A, 1, n)$$

  missä rekursiivinen kutsu

  $$\text{mergeSort}(A, p, r)$$

  järjestää taulukon osan $A[p \ldots r]$.

- Tämä rekursiivinen funktio on

mergeSort(A, p, r)
1 **if** $p < r$ **then**
2 $\quad q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor$
3 $\quad$ mergeSort(A, p, q)
4 $\quad$ mergeSort($A, q+1, r$)
5 $\quad$ merge(A, p, q, r)

ja sen toiminta voidaan perustella seuraavasti.

- Jos käsiteltävän osan pituus

$$\ell = r - p + 1 \leq 1$$

alkiota, niin

  - ei tehdä mitään (rivi 1)
  - koska sehän on jo järjestyksessä.
  - Tämä on funktion (tyhjä) ei-rekursiivinen haara.

- Muuten rivillä 2 asetetaan q käsiteltävän taulukon osan keskelle.
  - Jälleen käytetään tuttua ideaa *puolittaa* syöte.
  - Vertaa esimerkiksi kalvojen 1.3.3 binäärihaku.

- Taulukon

  alkuosa $A[p \ldots q]$ (rivi 3)

  loppuosa $A[q + 1 \ldots r]$ (rivi 4)

  järjestetään rekursiivisesti.

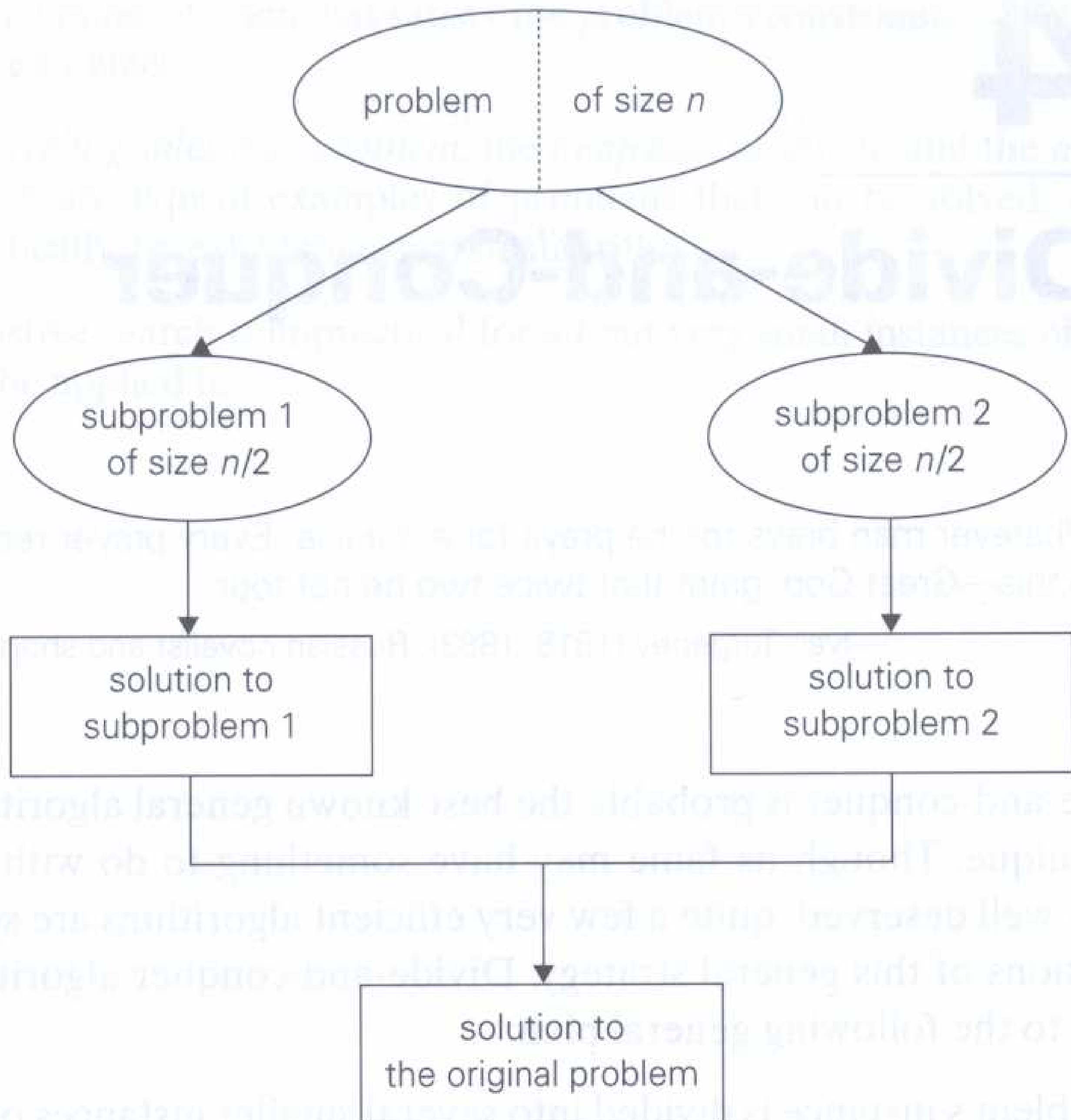
- Rivillä 5 kutsutaan operaatiota merge joka

  lomittaa nämä kaksi osaa yhdeksi järjestetyksi osaksi $A[p \ldots r]$

  lineaarisessa ajassa

$$\mathcal{O}(\ell).$$

(Kuva 4.1 kirjasta A. Levitin: *Introduction to the Design & Analysis of Algorithms.* Addison-Wesley, 2003.)


problem
of size n
subproblem 1
of size n/2
subproblem 2
of size n/2
solution to
subproblem 1
solution to
subproblem 2
solution to
the original problem

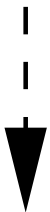
1	2	3	4	5	6	7	8
5	2	4	7	1	3	2	6

merge−sort(A,1,8)

1	2	3	4
5	2	4	7

5	6	7	8
1	3	2	6

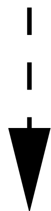
merge−sort(A,1,4)



merge−sort(A,5,8)

1	2	3	4
2	4	5	7

5	6	7	8
1	2	3	6



merge(A,1,4,8)

1	2	3	4	5	6	7	8
1	2	2	3	4	5	6	7

- Lineaarisen lomituksen intuitio:
  - Olkoon meillä 2 *järjestettyä* pelikorttipakkaa kuvapuoli ylöspäin.
  - Verrataan niiden päällimmäisiä kortteja.
  - Korteista pienempi siirretään tulospakkaan kuvapuoli alaspäin.
  - Suurempi jää oman pakkansa päälle.
  - Näin jatketaan, kunnes kaikki kortit ovat tulospakassa.
  - Silloin tulospakka
    - koostuu alkuperäisten pakkojen korteista
    - on samassa järjestyksessä

      (kun se käännetään ympäri kuvapuoli ylöspäin)
    - on muodostettu lineaarisessa ajassa korttien lukumäärän suhteen.

- Sovelletaan tätä korttipakkamentelmää *taulukoihin* seuraavasti:
  - Kopioidaan lomitettavat osat (riittävän pitkiin) aputaulukoihin

$$L[1 \ldots q-p+1] = A[p \ldots q]$$
$$R[1 \ldots r-q] = A[q+1 \ldots r].$$

  - Aputaulukon

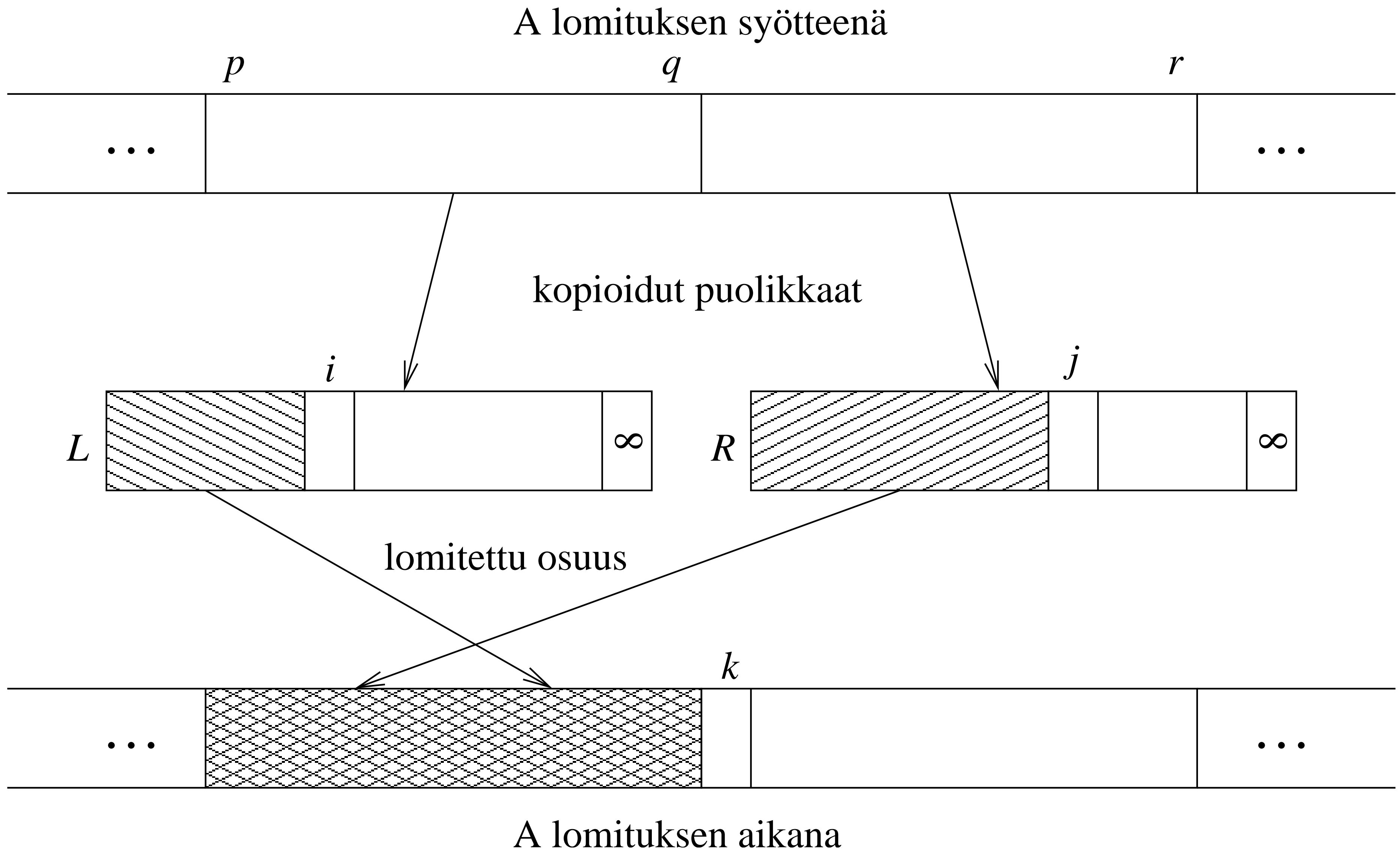
    alkuosa $L[1 \ldots i-1]$ on jo lomitettu tulostaulukkoon

    loppuosa $L[i \ldots]$ odottaa vielä lomittamista.

  - Vastaavasti $R[1 \ldots j-1]$ ja $R[j \ldots]$.
  - Tulostaulukon $A[p \ldots r]$

    alkuosa $A[p \ldots k-1]$ koostuu näistä aputaulukoiden alkuosien alkioista lomitettuina järjestykseen

    loppuosa $A[k \ldots r]$ odottaa vielä täyttämistään.


A lomituksen syötteenä
p
q
r
...
...
kopioidut puolikkaat
i
j
L
∞
R
∞
lomitettu osuus
k
...
...
A lomituksen aikana

- Algoritmi ylläpitää näitä invariantteja
  - siirtämällä pienemmän alkioista $L[i]$ ja $R[j]$ paikkaan $A[k]$
  - päivittämällä indeksejä k sekä joko i tai j vastaavasti.

```
merge(A, p, q, r)
1   n1 ← q − p + 1
2   for i ← 1 to n1 do
3       L[i] ← A[p + i − 1]
4   L[n1 + 1] ← +∞
5   n2 ← r − q
6   for j ← 1 to n2 do
7       R[j] ← A[q + j]
8   R[n2 + 1] ← +∞
9   i ← 1
10  j ← 1
11  for k ← p to r do
12      if L[i] ≤ R[j] then
13          A[k] ← L[i]
14          i ← i + 1
15      else
16          A[k] ← R[j]
17          j ← j + 1
```

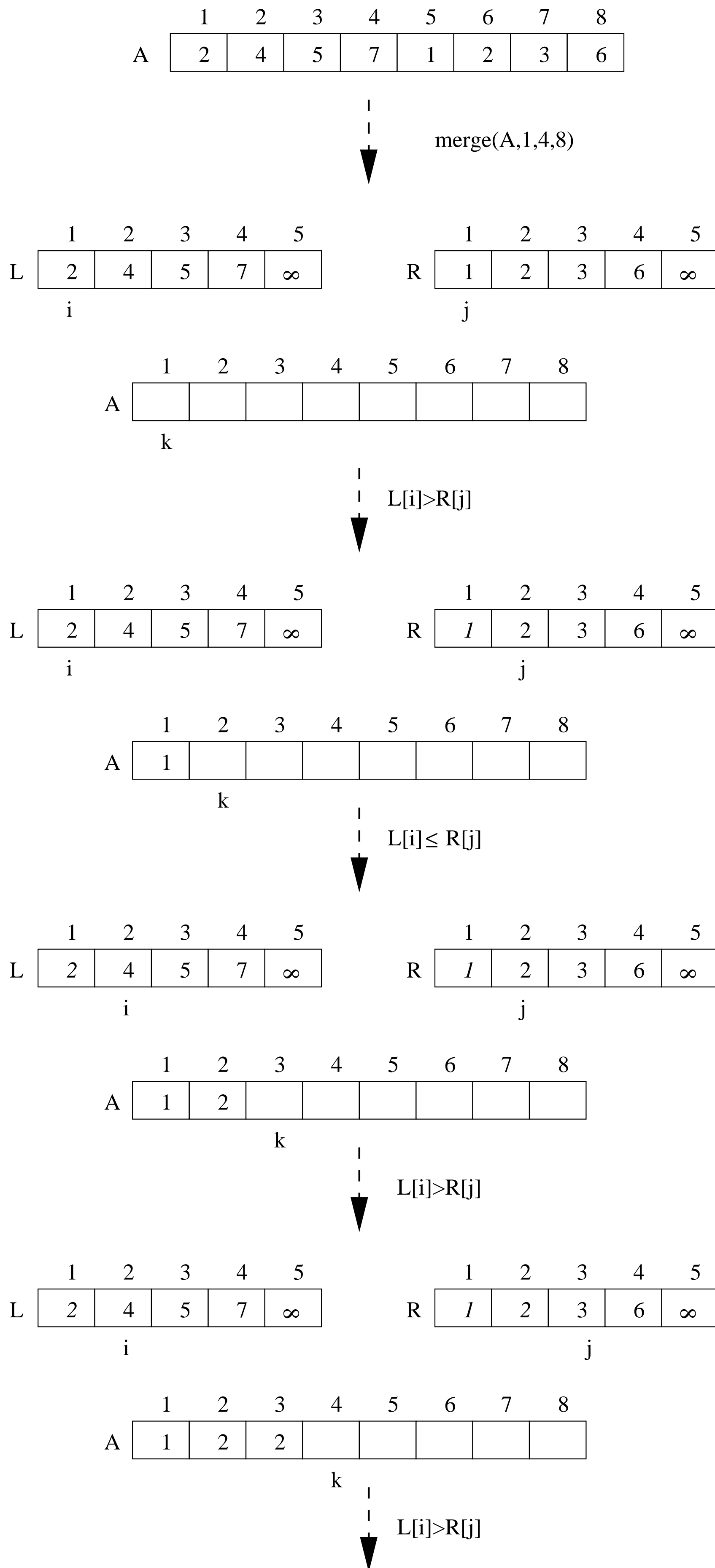
- Riveillä 1–4 luodaan aputaulukko

$$L[1 \ldots \underbrace{q - p + 1}_{n_1} \underline{+1}].$$

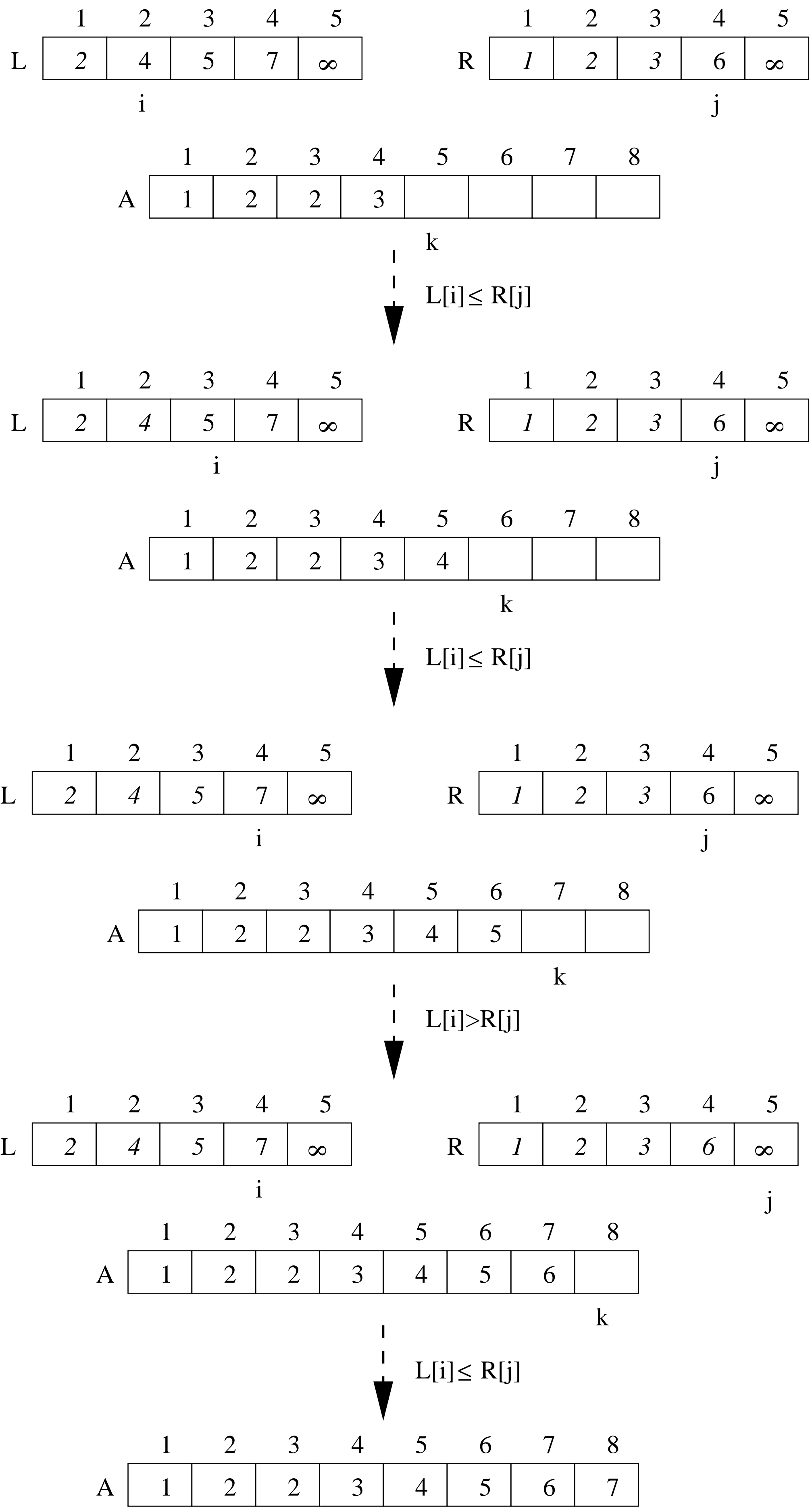
  <u>Lisäpaikka</u>an laitetaan rivillä 4 *rajapyykki* (sentinel) $+\infty$ takaamaan, ettei edetä ohi taulukon viimeisen alkion.

- Riveillä 5–8 vastaavasti aputaulukko R.

- Rajapyykit yksinkertaistavat alkioista $L[i]$ ja $R[j]$ pienemmän valintaa rivillä 12:
  - Jos koko L on jo siirretty, niin
    - $i = n_1 + 1$
    - $L[i] = +\infty$
    - valitaankin $R[j]$.
  - Vastaavasti aputaulukossa R.
  - Ilman rajapyykkejä pitäisi testata taulukon L loppuminen **if**-ehdossa.



A
merge(A,1,4,8)
L
R
i
j
k
L[i]>R[j]
L[i]≤ R[j]
L[i]>R[j]
L[i]>R[j]


L
R
A
i
j
k
L[i]≤ R[j]
L[i]>R[j]
1 2 3 4 5
1 2 3 4 5 6 7 8
2 4 5 7 ∞
1 2 3 6 ∞
1 2 2 3 4 5 6 7

- Edellinen kuvasarja antaa esimerkin lomituksen toiminnasta.

- Seuraava kuva näyttää miten taulukko

  hajoaa osiin rekursiolla

  osat yhdistyvät lomitusoperaatioilla.

- Lomituslajittelu on helppo toteuttaa *vakaa*ksi

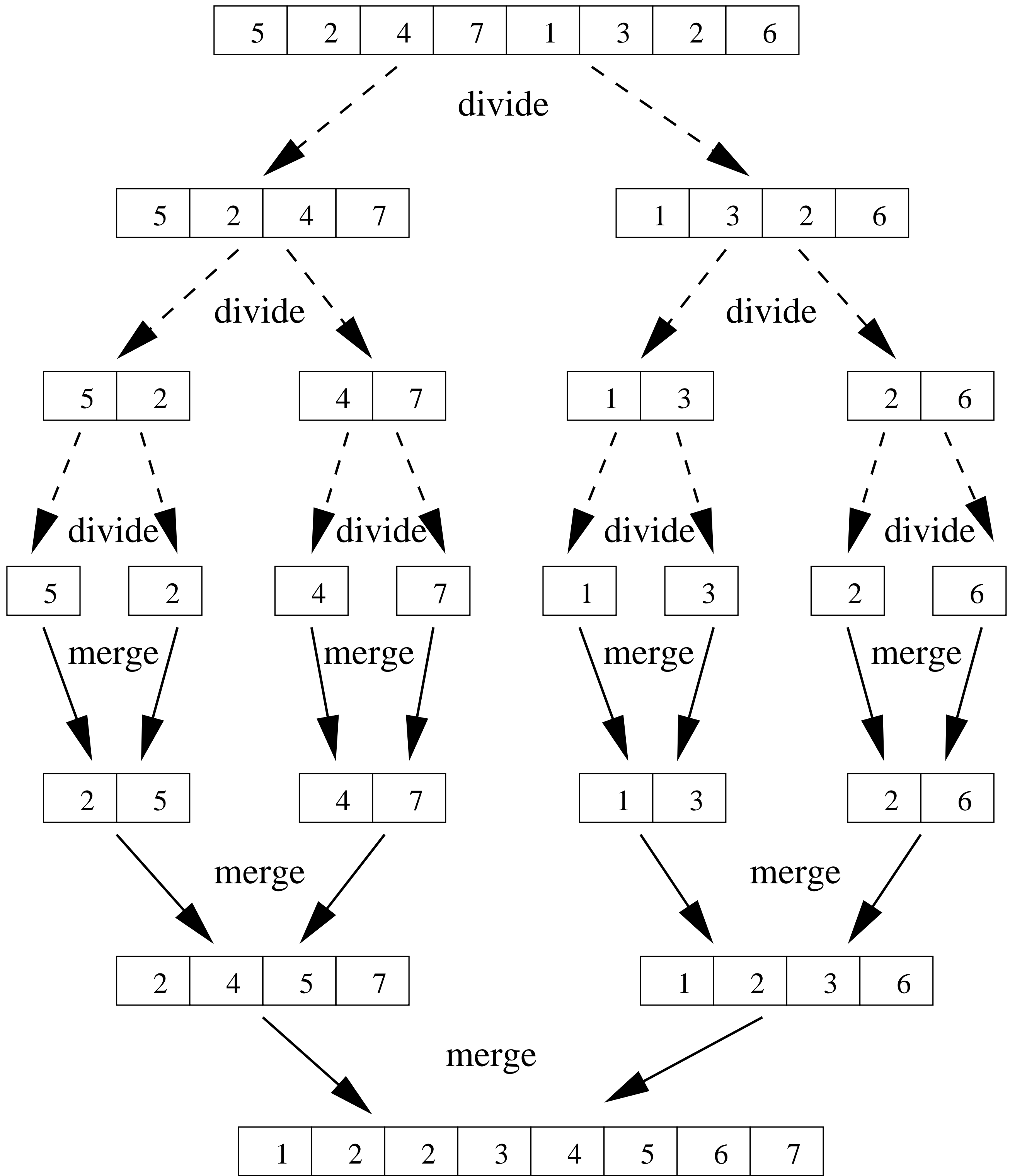
  – eli sellaiseksi, että se säilyttää yhtä suurten syötealkioiden alkuperäisen keskinäisen järjestyksen: syötteessä

  $$\ldots, 5, \ldots, 5, \ldots$$

  oikeanpuoleinen viitonen ei vahingossa ohita järjestämisen aikana vasemmanpuoleista

  – jakamalla taulukko tähän tapaan vasempaan ja oikeaan palaseen

  – lomittamalla ne tähän tapaan siten, että otetaan ensin vasemmasta palasesta jos vain voidaan.


taulukko alussa
5 2 4 7 1 3 2 6
divide
5 2 4 7
1 3 2 6
divide
divide
5 2
4 7
1 3
2 6
divide
divide
divide
divide
5
2
4
7
1
3
2
6
merge
merge
merge
merge
2 5
4 7
1 3
2 6
merge
merge
2 4 5 7
1 2 3 6
merge
1 2 2 3 4 5 6 7
taulukko järjestyksessä

6.2.1 Vaativuusanalyysi

- Ennen kuin analysoimme koko lomitusjärjestämisen aikavaativuutta, varmistetaan että näin toteutettu merge on todellakin aikavaatimukseltaan lineaarinen:

  - Olkoon $\ell = r - p + 1$ lomitettavan taulukon osan pituus.

  - Aputaulukoiden L ja R täyttäminen (rivit 1–4 ja 5–8) vie

    tilaa yhteensä $\ell + 2 = \mathcal{O}(\ell)$ taulukkopaikkaa

    koska lomitettava taulukon osa jaetaan niihin

    aikaa $\mathcal{O}(\ell)$ askelta ($\mathcal{O}(1)$ / paikka).

  - Rivien 11–17 silmukassa

    * toistetaan ℓ kertaa

    * $\mathcal{O}(1)$ askeleen sisältöä

    eli $\mathcal{O}(\ell)$ askelta.

- Yhteensä siis merge vie

  aikaa

  tilaa

  lineaarisen määrän $\mathcal{O}(\ell)$, kuten luvattiinkin.

- Entä koko lomitusjärjestämisen aikavaativuus?

- Tehdään yksinkertaistava oletus:

  – järjestettävän taulukon koko n on jokin kahden potenssi 2^p

  – jolloin jokainen jako siis puolittaa taulukon kahteen yhtäsuureen osaan

  – eli pienentää eksponenttia p yhdellä.

  Sama yksinkertaistus tehtiin jo kalvojen 1.3.3 binäärihaun analyysissä.

- Binäärihaun analyysissä käytimme myös seuraavaa rekursioyhtälön ideaa:
  - Otetaan käyttöön merkintä

    $T(k) = k$ paikkaa pitkän taulukonosan vaatima aika.

  - Kirjoitetaan lomitusjärjestämisen koodin perusteella rekursioyhtälö

    $$T(1) = \mathcal{O}(1)$$
    $$T(k) = 2 \cdot T(k/2) + \mathcal{O}(k) \qquad \text{kun } k > 1.$$

    * Ylempi yhtälö vastaa algoritmin ei-rekursiivista, alempi taas rekursiivista haaraa.
    * Rekursiivisessa haarassa algoritmi kutsuu itseään 2 kertaa
    * kummallakin rekursiokutsukerralla puolet lyhyemmällä taulukolla.
    * Lisäksi tehdään edellä lineaariseksi todettu lomitustyö.

- Korvataan selkeyden vuoksi tässä yhtälössä $\mathcal{O}$-merkintä sen takaamalla riittävän suurella vakiolla c:

$$T(1) = c$$
$$T(k) = 2 \cdot T(k/2) + c \cdot k \qquad \text{kun } k > 1.$$

- Aletaan laskea tätä yhtälöä auki kun $n = 2^p$:

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + c \cdot n \\ &= 2 \cdot (2 \cdot T(n/4) + c \cdot n/2) + c \cdot n \\ &= 4 \cdot T(n/4) + 2 \cdot c \cdot n \\ &= 4 \cdot (2 \cdot T(n/8) + c \cdot n/4) + 2 \cdot c \cdot n \\ &= 8 \cdot T(n/8) + 3 \cdot c \cdot n \\ &= 8 \cdot (2 \cdot T(n/16) + c \cdot n/8) + 3 \cdot c \cdot n \\ &= 16 \cdot T(n/16) + 4 \cdot c \cdot n \end{aligned}$$

ja havaitaan, että q korvauksen jälkeen on

$$= 2^q \cdot T(n/2^q) + q \cdot c \cdot n.$$

Tämä päättyy, kun $n = 2^q$, eli $q = p$, jolloin

$$= 2^p \cdot \underbrace{T(1)}_{c} + p \cdot c \cdot n.$$

Toisin sanoen

$$= n \cdot c + (\log n) \cdot c \cdot n.$$

- Havainnollisempaa on muodostaa *rekursiopuu:*

  – Jokaisessa solmussa on yksi rekursiokutsu

  $$\text{mergeSort}(A, p, r).$$

  – Sen lapsina ovat sen synnyttämät rekursiokutsut

  $\text{mergeSort}(A, p, q)$ ja $\text{mergeSort}(A, q+1, r)$

  missä

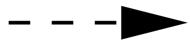
  $$q = \left\lfloor \frac{p+r}{2} \right\rfloor.$$

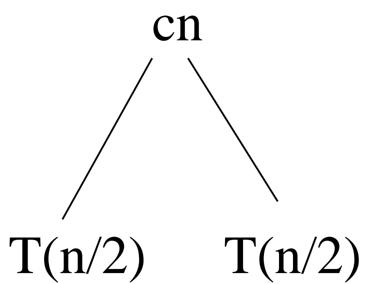
- Nyt kun lasketaan aika-arviota, niin

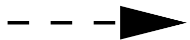
  – kutsujen sijasta kirjoitetaankin solmuihin vastaava T-lauseke

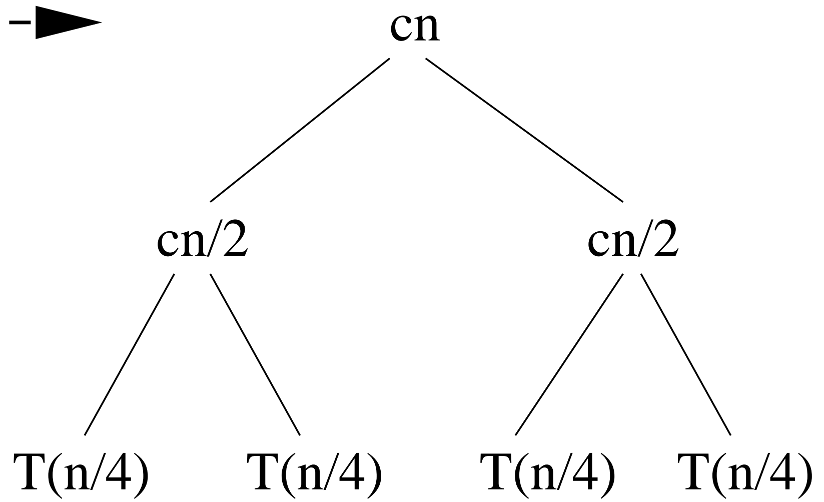
  – puun kasvatus vastaa askeisiä korvauksia kaavaan.

T(n)









- Jatketaan rekursiopuun kasvatusta, kunnes tullaan yhden kokoisen taulukon järjestämistä vastaaviin lehtisolmuihin.

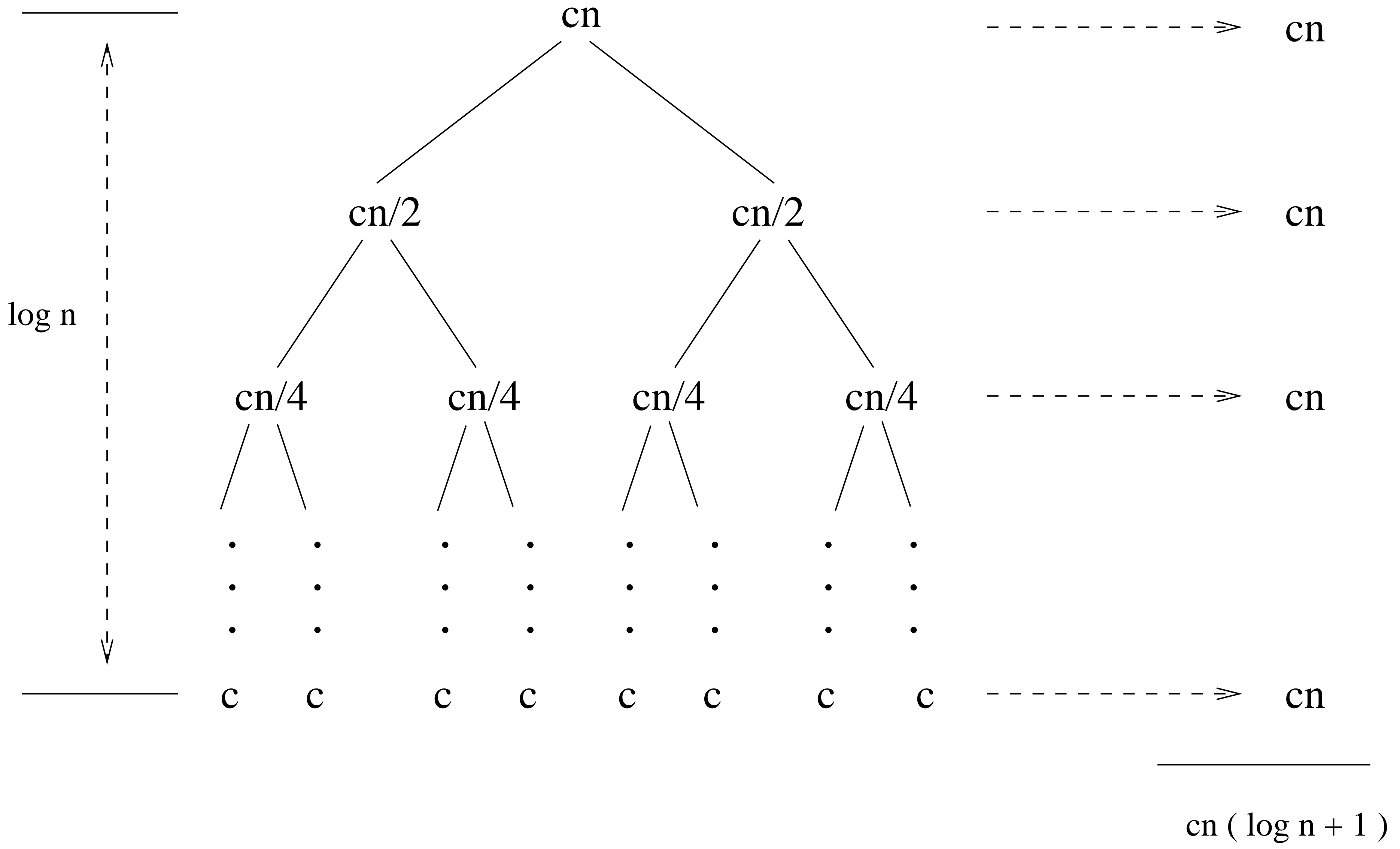
- Huomaamme että rekursiopuun jokaisen tason t vaativuus on $c \cdot n$:
  - tasolla t on 2^t solmua
  - jokaisessa solmussa on kulunut aikaa

$$c \cdot n/2^t \qquad \text{askelta.}$$

- Rekursiopuun muoto on seuraava:
  - Se on täydellinen binääripuu, koska oletimme syötteen pituudeksi $n = 2^p$.
  - Siinä on n lehteä, yksi jokaiselle syötepaikalle $A[i]$.
  - Sen korkeus on

$$p = \log n$$

    ja tasoja t on $p + 1$ kappaletta.



cn
cn/2
cn/2
cn/4
cn/4
cn/4
cn/4
c c c c c c c c
log n
cn
cn
cn
cn
cn (log n + 1)

- Summataan yhteen kaikki nämä tasot: rekursion kokonaisajantarve on

$$c \cdot n \cdot (\log n + 1)$$

- Molemmilla tavoilla on (tietenkin) saatu sama tulos: lomituslajittelun ajantarve on

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

askelta.

- Tämän toteutuksen aputilantarve on

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

muistipaikkaa, koska

  – suurin lomitus tarvitsee aputilaa koko syötteelle

  – samaa aputilaa voi käyttää uudelleen eri lomituksissa

  – sisäkkäisiä rekursiokutsuja on rekursiopuun korkeuden verran.

6.2.2 Ilman rekursiota

- Lomitusjärjestäminen käyttää rekursiota varsin yksinkertaisesti:

  eteenpäin mentäessä vain jakamaan syötettä yhä pienempiin palasiin

  takaisin palatessa vain lomittamaan näitä keskenään saman kokoisia (nyt järjestettyjä) palasia.

- Niinpä se voidaankin tehdä ilman rekursiota:

  Aloitetaankin suoraan yhden alkion mittaisista peruspalasista.

  Joka silmukkakierroksella yhdistellään saman kokoisia naapuripalasia:

  1. ensimmäinen ja toinen palanen
  2. kolmas ja neljäs palanen
  3. viides ja kuudes palanen

  ja niin edelleen.

- Periaate näkyy tästä kuvasta, jossa
  - palasten rajat on merkitty [hakasuluin]
  - alustuksena jokainen alkio on oma palasensa
  - kierrokset ovat
    1. yhdistä vierekkäiset 1 alkion palaset 2 alkion palaseksi
    2. vierekkäiset 2 alkion palaset 4 alkion palaseksi
    3. vierekkäiset 4 alkion palaset 8 alkion palaseksi.

```
 5   7   8   2   4   6   1   9   3   0
[5] [7] [8] [2] [4] [6] [1] [9] [3] [0]
[5   7] [2   8] [4   6] [1   9] [0   3]
[2   5   7   8] [1   4   6   9] [0   3]
[1   2   4   5   6   7   8   9] [0   3]
[0   1   2   3   4   5   6   7   8   9]
```

- Kuvasta näky myös, että viimeinen palanen
  - saattaa jäädä ilman paria
  - ja muita lyhyemmäksi
  - mutta aikanaan sekin lomittuu edellisen palasen kanssa.

- Algoritmina

```
1  u ← 1
2  while u < n do
3      v ← 1
4      while v + u ≤ n do
5          merge(A, v, v + u − 1,
               min(v + 2 · u − 1, n))
6          v ← v + 2 · u
7      u ← 2 · u
```

$u =$ tällä kierroksella yhdisteltävien palasten pituus

(paitsi kierroksen viimeinen palanen voi olla lyhyempikin)

$v =$ seuraavan tällä kierroksella yhdistettävän parin alkukohta.

Aputilaa tarvitaan jälleen lineaarisesti:

- lomitustila
- sekä kaksi muuttujaa u ja v
- mutta ei enää rekursiopinoa.

Aikaa tarvitaan yhä samat

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

- ulkosilmukan joka kierroksella $u = 1, 2, 4, 8, \ldots$
- sisäsilmukka käy taulukon A läpi lomittaen pareja, siis

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

- ulkosilmukalla on kierroksia

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

Algoritmi ja sen analyysi etenevät siis samassa järjestyksessä kuin rekursiopuun summaus tasoittain kalvoilla 6.2.1.

6.2.3 Listoilla

- Lomitusjärjestäminen on erityisen sopivaa kalvojen 2.4 linkitetyille listoille:
  - Lomitus voidaan tehdä listasolmujen viitekenttiä muokkaamalla
  - muuta aputilaa ei siihen tarvita.

- Muunnetaan kalvojen 6.2.2 algoritmi käyttämään listoja.

- Valitaan sellainen listarakenne, jossa kalvojen 6.2 pelikorttipakkaintuition perusoperaatiot
  - kortin otto pakan alusta
  - kortin pano pakan loppuun

  ovat vakioaikaisia $\mathcal{O}(1)$:

  kalvojen 2.2 *jono*rakenteen linkitetty toteutus!

- Kahden jonon L ja R lomitus lineaarisessa ajassa

$$\mathcal{O}(\text{jonon } L \text{ pituus} + \text{jonon } R \text{ pituus})$$

annetun jonon B loppuun on korttipakkaintuition mukaisesti

merge(B, L, R)
1 enqueue($L, +\infty$)
2 enqueue($R, +\infty$)
3 **while** first(L) $\neq +\infty$ **or**
  first(R) $\neq +\infty$ **do**
4   **if** first(L) $\leq$ first(R) **then**
5   enqueue(B, dequeue(L))
6   **else**
7   enqueue(B, dequeue(R))

missä uusi jono-operaatio first

- palauttaa mutta ei poista epätyhjän jonon ensimmäisen alkion
- voidaan toteuttaa vakioaikaisena $\mathcal{O}(1)$.

- Algoritmi saa nyt muodon

```
1   n ← syötejonon A pituus
2   u ← 1
3   while u < n do
4       B ← tyhjä jono
5       repeat
6           L ← take(A, u)
7           R ← take(A, u)
8           merge(B, L, R)
9       until empty(A)
10      A ← B
11      u ← 2 · u
```

- Riveillä 6 ja 7 käytetään aliohjelmaa

```
take(A, u)
1   C ← tyhjä jono
2   t ← 0
3   while t < u and not empty(A) do
4       enqueue(C, dequeue(A))
5       t ← t + 1
6   return C
```

- Tämä aliohjelma take(A, u)
  - ottaa jonosta A sen u ensimmäistä alkiota tulosjonoksi C
  - samassa järjestyksessä
  - tai jos A on liian lyhyt, niin sen kaikki alkiot
  - toimii vakioajassa / silmukkakierros.
- Koko algoritmin samana pysynyt aikavaativuus

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

voidaan todeta kuten kalvoilla 6.2.2.

- Lomitusjärjestäminen näyttää selkeämmältä, kun käytetään listoja eikä taulukoita:
  - Indeksointi korvautui operaatioilla ”lue seuraava” ja ”liitä edellisiin”.
  - Lomitusjärjestäminen siis käy läpi syötettään luontevasti *peräkkäin*.
  - Sitä vastoin kalvojen 6.1 kekojärjestäminen hyödynsi oleellisella tavalla taulukkoindeksoinnin mahdollistamaa *hajasaantia*.

- Lomitusjärjestäminen onkin erinomainen vaihtoehto silloin, kun järjestäminen täytyykin suorittaa käyttäen työmuistina peräkkäis*tiedostoja* keskusmuistin sijaan.

  (Tosin nykyään tämä ei enää ole niin yleistä kuin aikaisemmin.)

6.3 Pikajärjestäminen

- Sovelletaan edelleen hajoita-ja-hallitse-periaatetta.

- Taulukko $A[1 \ldots n]$ järjestetään kutsulla

$$\text{quickSort}(A, 1, n)$$

rekursiiviseen funktioon

```
quickSort(A, p, r)
1   if p < r then
2       q ← partition(A, p, r)
3       quickSort(A, p, q)
4       quickSort(A, q + 1, r)
```

- Jos järjestettävä taulukon alue $A[p \ldots r]$ on alle 2-paikkainen, niin se on jo järjestyksessä (rivi 1).

- Rivillä 2 tehdään *ositus*, jonka tuloksena

  alkuosan $A[p \dots q]$ alkiot ovat *korkeintaan yhtä suuria* kuin

  loppuosan $A[q+1 \dots r]$ alkiot.

  Katso seuraava kuva!

- Nämä osat voivat olla keskenään hyvinkin *eri kokoisia*

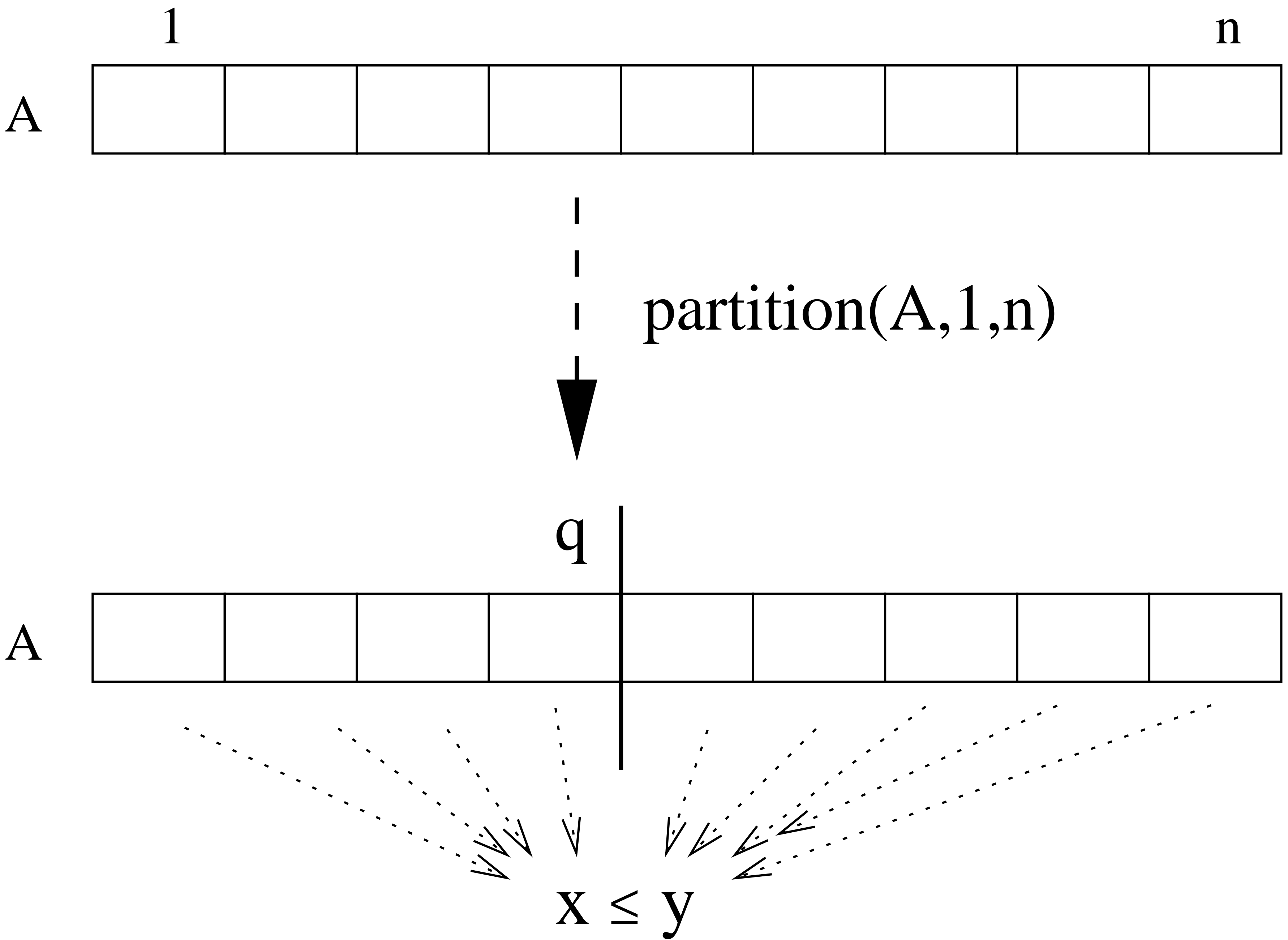
  – eli osituksen valitsema jakoindeksi q ei välttämättä olekaan keskellä aluetta $A[p \dots r]$

  – toisin kuin kalvojen 6.2 lomitusjärjestämisessä.

- Osituksen jälkeen järjestetään nämä osat erikseen rekursiivisesti (rivit 3 ja 4).

- Sen jälkeen alue $A[p \dots r]$ on järjestyksessä.

  Erillistä osien yhdistelyvaihetta (kuten lomitusta) ei nyt tarvita, sillä osituksen seurauksena alkuosa saa edeltää loppuosaa.


1
n
A
partition(A,1,n)
q
A
x ≤ y

- Pikajärjestämisen tehokkuuden kannalta on oleellista että ositusoperaatio partition

  toimii nopeasti eli *lineaarisessa ajassa* $\mathcal{O}(\ell)$ jaettavan alueen $A[p \ldots r]$ pituuden $\ell = r - p + 1$ suhteen

  tämä on *välttämätön* ehto — muuten "pika"järjestäminen toimisi *kaikilla* syötteillä hitaasti

  jakaa tasaisesti tämän alueen alku- ja loppuosiin

  tämä on *toivottavaa* mutta ei välttämätöntä — epätasainen jako aiheuttaa, että juuri *tällä* syötteellä pikajärjestäminen toimiikin hitaasti.

- Edellä "hitaasti" tarkoittaa kauemmin kuin

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

  joka on pikajärjestämisen(kin) ihannetapaus.

- Siis pikajärjestäminen *ei takaa* ihannetapausta!

  Toisin kuin kalvojen 6.1 ja 6.2 keko- ja lomitusjärjestämiset.

6.3.1 Ositus

- On monia erilaisia ositusalgoritmeja.
- Tarkastellaan niistä seuraavaa:

partition(A, p, r)
1 $a \leftarrow A[p]$
2 $i \leftarrow p - 1$
3 $j \leftarrow r + 1$
4 **while** true **do**
5 **repeat** $i \leftarrow i + 1$ **until** $A[i] \geq a$
6 **repeat** $j \leftarrow j - 1$ **until** $A[j] \leq a$
7 **if** $i < j$ **then** vaihda $A[i] \leftrightarrow A[j]$
8 **else return** j

(Se on Quicksort-algoritmin keksijän C.A.R. Hoaren oma alkuperäinen ositusmenetelmä.)

- Rivillä 1 valitaan *jakoalkio*ksi (engl. pivot) vasemmanpuoleisimman alkion $A[p]$ arvo a.

- Tavoitteena on, että osituksen jälkeen

  alkuosan kaikki alkiot ovat $\leq a$

  loppuosan kaikki alkiot ovat $\geq a$.

- Kuljetaan ositettavaa aluetta $A[p \dots r]$ yhtä aikaa

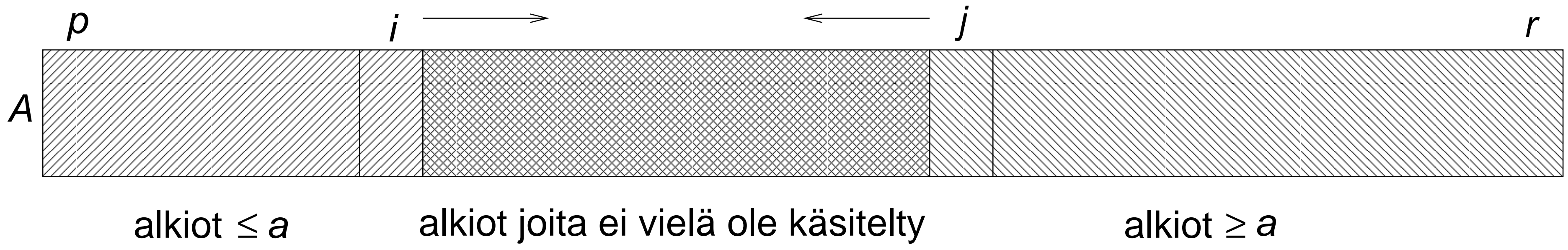
  alusta loppuun päin indeksillä i

  lopusta alkuun päin indeksillä j.

- Rivin 5 silmukka vie indeksin i *ensimmäiseen* kohtaan, jossa $A[i] \geq a$.

  Eli ohittaa kaikki ne $A[i]$ joiden täytyy kuulua alkuosaan.

- Rivi 6 vastaavasti indeksin j *viimeiseen* kohtaan, jossa $A[j] \leq a$.


p
i
j
r
A
alkiot ≤ a
alkiot joita ei vielä ole käsitelty
alkiot ≥ a

- Kun indeksit kohtaavat (tai ohittavat) toisensa, niin ositus on valmis:

  alkuosa on $A[p \ldots j]$: kaikki alkiot $\leq a$

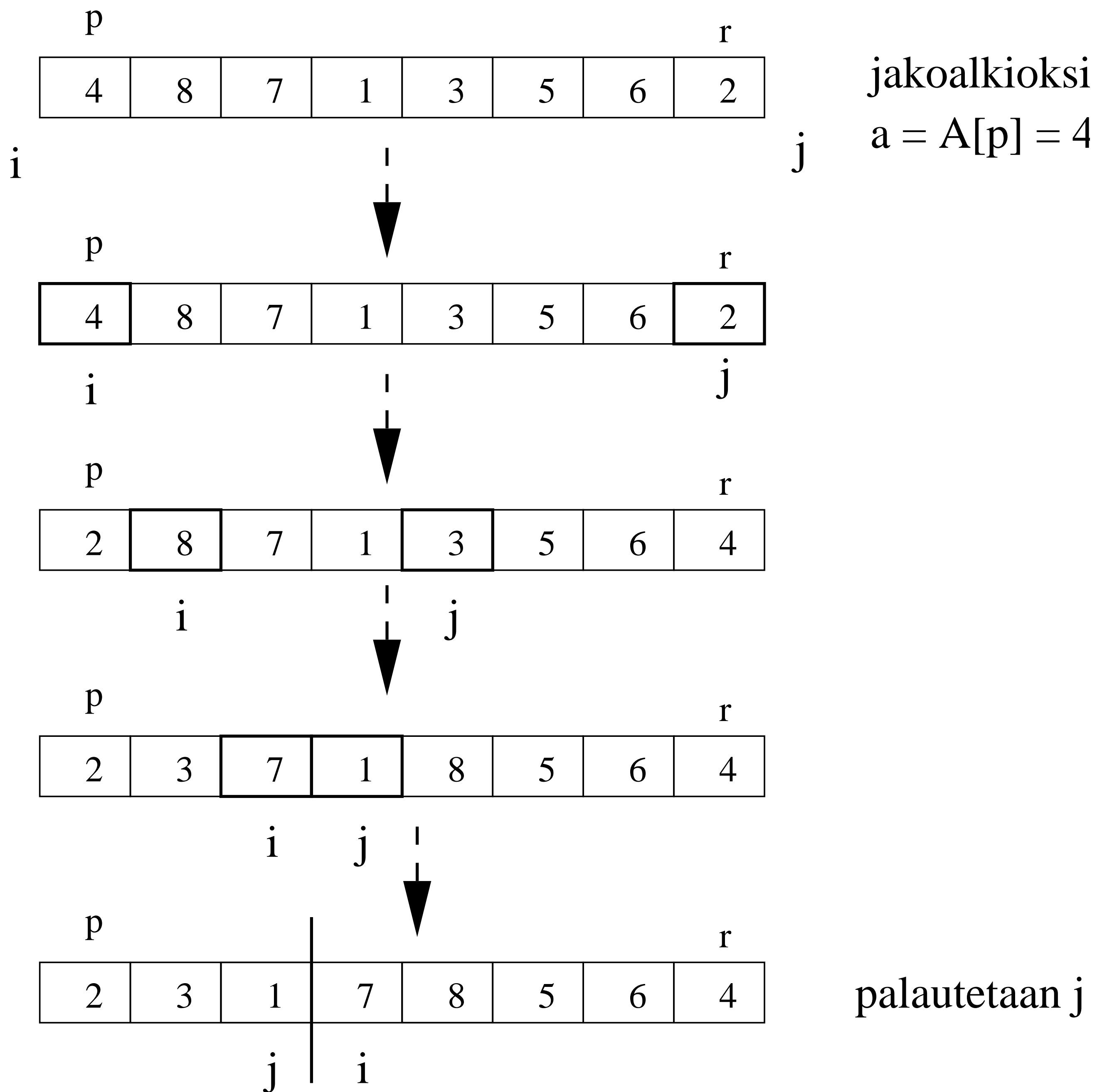
  loppuosa on $A[j+1 \ldots r]$: kaikki alkiot $\geq a$

  (rivi 8).

- Muuten mahdollistetaan indeksien etenemisen jatkaminen

  vaihtamalla keskenään etenemisen pysäyttäneet alkiot $A[i]$ ja $A[j]$ (rivi 7).

- Seuraavassa kuvassa on esimerkki tämän ositusalgoritmin toiminnasta.


p
r
4 8 7 1 3 5 6 2
i
j
jakoalkioksi
a = A[p] = 4
p
r
4 8 7 1 3 5 6 2
i
j
p
r
2 8 7 1 3 5 6 4
i
j
p
r
2 3 7 1 8 5 6 4
i j
p
r
2 3 1 7 8 5 6 4
j i
palautetaan j

- Selvästi tämän partition-operaation aikavaativuus on lineaarinen

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

suhteessa ositettavan alueen pituuteen

$$\ell = r - p + 1.$$

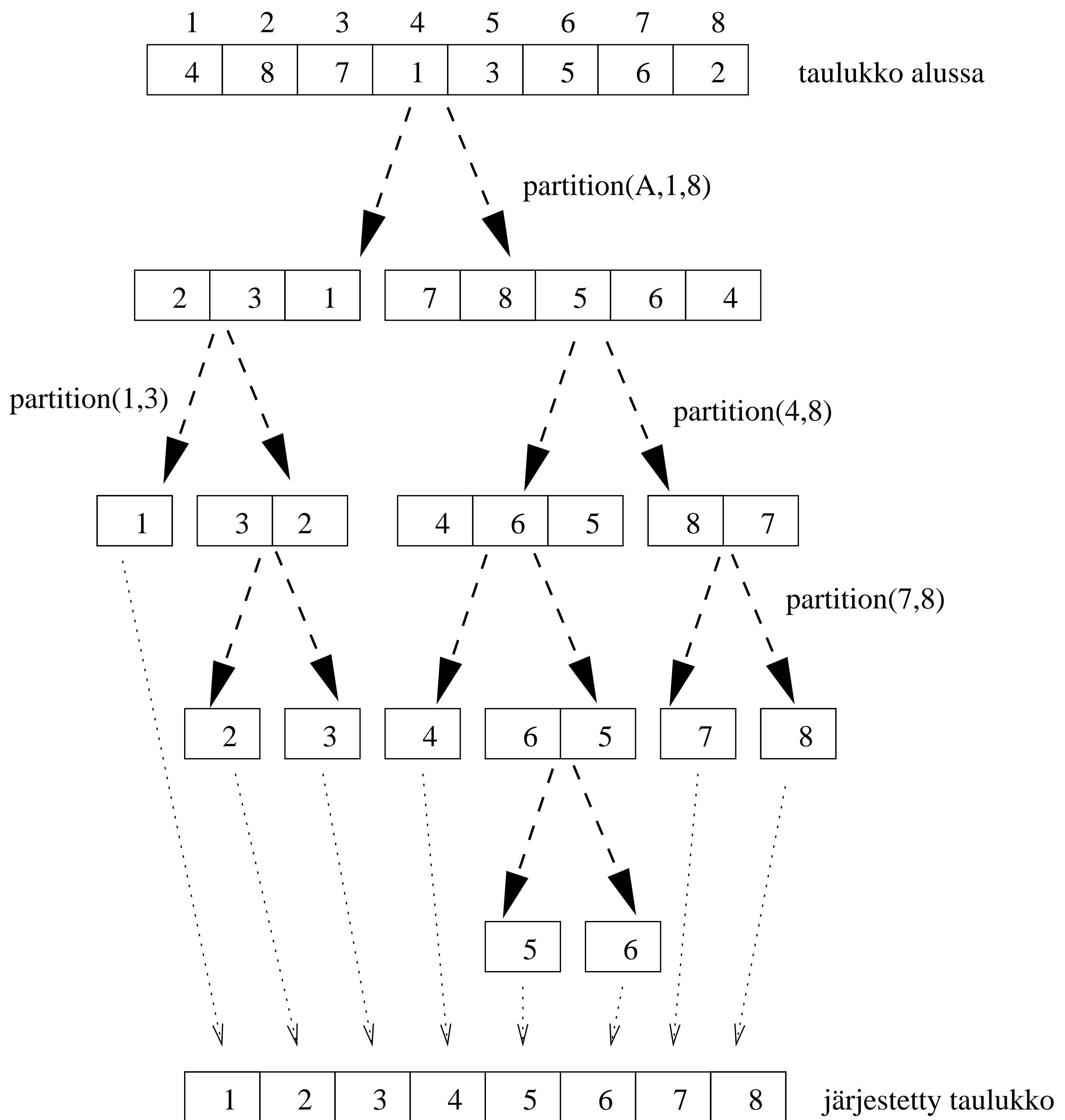
- Aputilaa tarvitaan vain parin muuttujan verran, eli tilavaativuus on

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

- Esimerkissämme meillä oli melko hyvä tuuri:
  - Taulukko jakautui kohtuullisen tasan kahtia.
  - Entä jos kyseessä olisi ollut seuraavan kuvan taulukko?

- Sen jälkeen on vielä esimerkkikuva koko pikajärjestämisen toiminnasta.

p							r
8	1	7	4	3	5	6	2


1 2 3 4 5 6 7 8
4 8 7 1 3 5 6 2
taulukko alussa
partition(A,1,8)
2 3 1
7 8 5 6 4
partition(1,3)
partition(4,8)
1
3 2
4 6 5
8 7
partition(7,8)
2
3
4
6 5
7
8
5
6
1 2 3 4 5 6 7 8
järjestetty taulukko

6.3.2 Vaativuusanalyysi

- *Pahimmassa* tapauksessa ositus jakaa ℓ-alkioisen alueen kahtia siten, että toisessa osassa on vain 1 alkio.

- Esimerkiksi edellä esitetylle partition-toteutukselle käy näin, jos syöte on jo valmiiksi oikeassa järjestyksessä.

- Pahimman tapauksen vaativuus $T_w(n)$ voidaan siis määritellä seuraavalla rekursioyhtälöllä:

$$
\begin{aligned}
T_w(1) &= c \\
T_w(k) &= T_w(1) + T_w(k-1) + c \cdot k \\
&\qquad \text{kun } k > 1.
\end{aligned}
$$

- Siinä c on vakio

  eli termi $c \cdot k$ kuvaa ositusoperaation vaativuutta.

- Lasketaan auki rekursioyhtälöä:

$$\begin{aligned}
T_w(n) &= T_w(1) + T_w(n-1) + c \cdot n \\
&= c + T_w(1) - T_w(n-2) + c \cdot (n-1) \\
&\qquad + c \cdot n \\
&= c + c + T_w(1) + T_w(n-3) + \\
&\qquad c \cdot (n-2) + c \cdot (n-1) + c \cdot n \\
&\vdots \\
&= c \cdot n + c \cdot \sum_{i=1}^{n} i \\
&= c \cdot n + \frac{c \cdot n \cdot (n+1)}{2} \\
&= \mathcal{O}\left(n^2\right).
\end{aligned}$$

- Pahimmassa tapauksessa pikajärjestäminen toimii *neliöisesti*

  eli on selvästi huonompi kuin lomitus- ja kekojärjestäminen.

- Entä *parhaassa* tapauksessa

  jossa ositus sattuu jakamaan taulukon aina kahteen yhtäsuureen osaan?

- Parhaan tapauksen vaativuus $T_b(n)$ voidaan määritellä seuraavalla rekursioyhtälöllä:

$$T_b(1) = c$$
$$T_b(k) = T_b(k/2) + T_b(k/2) + c \cdot k \quad \text{kun } k > 1.$$

- Yhtälö on täsmälleen sama kuin lomitusjärjestämisen vaativuutta kuvannut yhtälö kalvoilla 6.2.1

  eli parhaan tapauksen vaativuus on sama

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

- Pikajärjestämisen pahin tapaus on kuitenkin *erittäin* harvinainen.

- Käytännössä pikajärjestäminen toimii yleensä *paremmin* kuin lomitus- tai kekojärjestäminen.

- *Keskimääräisen* tapauksen vaativuus pikajärjestämisellä onkin sama

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

  kuin paras tapaus.

- Ohitamme tällä kurssilla todistuksen vaatimat todennäköisyystarkastelut.

- Intuitiona analysoidaan kuitenkin tilannetta missä partition jakaisi alkioita melko huonosti:

  m alkion taulukko jakautuisi osiin

$$\frac{1}{10}m \qquad \text{ja} \qquad \frac{9}{10}m$$

  eli suhteessa

$$10\% \qquad \text{ja} \qquad 90\%.$$

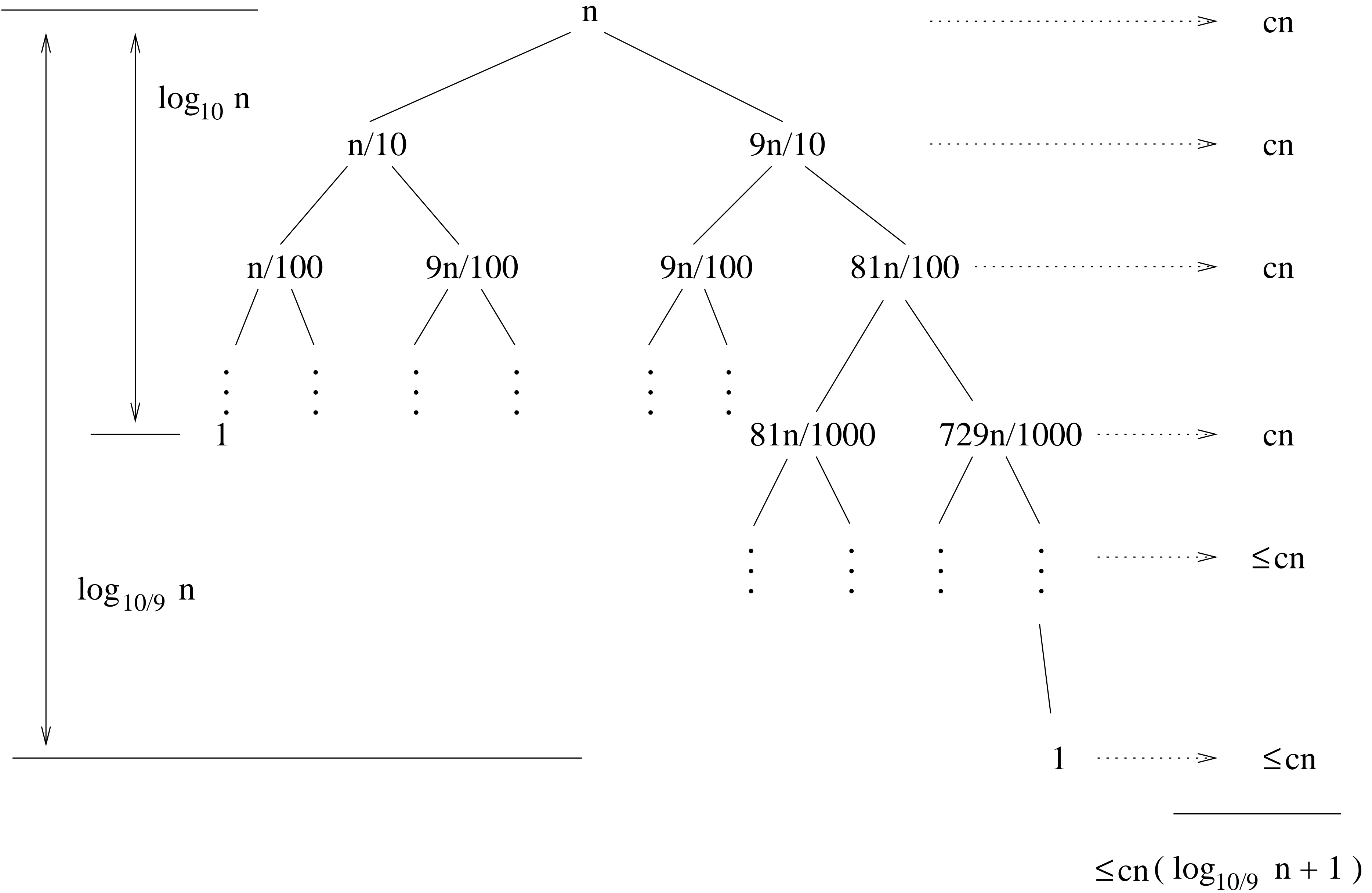
- Näytetään että jopa tässäkin tapauksessa pikajärjestämisen vaativuus olisi

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

- ”Epätasapainoisten jakojen” tapauksen vaativuus $T_u(n)$ voidaan määritellä seuraavalla rekursioyhtälöllä:

$$T_u(1) = c$$
$$T_u(k) = T_u(k/10) + T_u(9 \cdot k/10) + c \cdot k$$
$$\text{kun } k > 1.$$

- Kirjoitetaan rekursioyhtälöä auki kalvojen 6.2.1 rekursiopuumuodossa:
  - Seuraavassa kuvassa puun solmuihin on merkitty nyt rekursiokutsua vastaavan taulukonosan pituus k.
  - Puun ylimmät tasot ovat tasapainossa ja työmäärä / taso $= c \cdot n$.
  - Sen jälkeen puu alkaa vinoutua suuremman osan suuntaan ja työmäärä / taso $\leq c \cdot n$.


n
n/10
9n/10
n/100
9n/100
9n/100
81n/100
1
81n/1000
729n/1000
1
cn
cn
cn
cn
≤cn
≤cn
≤cn (log10/9 n + 1)
log10 n
log10/9 n

- Saamme siis epätasapainoisten jakojen tapauksen aikavaativuudeksi

$$\begin{aligned}
T_u(n) &\leq c \cdot n \cdot \left(\log_{10/9} n + 1\right) \\
&= c \cdot n \cdot \left(\frac{\log n}{\log(10/9)} + 1\right) \\
&\approx c \cdot n \cdot (\underbrace{6.57881347896\ldots}_{\text{merkitään } a} \cdot \log n + 1) \\
&= \mathcal{O}\left(n \cdot \log n\right).
\end{aligned}$$

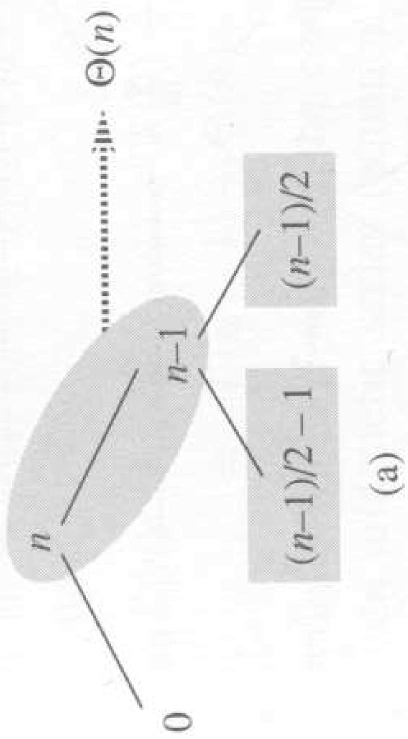
- Tosin rekursio*tasoja* tulee noin a kertaa enemmän kuin täysin tasapainoisissa jaoissa.

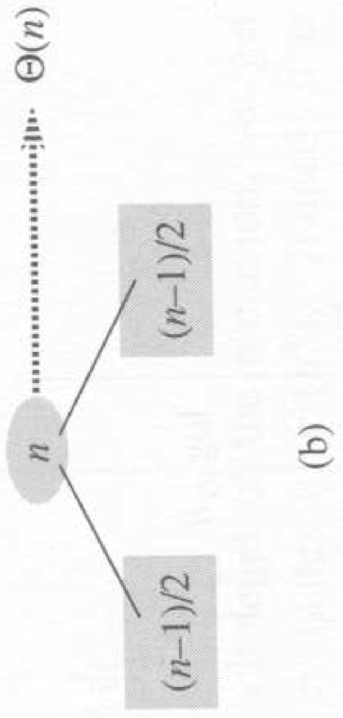
- Samoin käy (eri kertoimella a) kaikilla kiinteillä jaoilla. (1%/99%,...)

- Lisäksi: jos huonoaositusta seuraa hyvä, niin yhteisvaikutus on lähes yhtä hyvä.

  (Kuva 7.5 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

- Siis pahin tapaus vaatii että lähes koko ajan tehdään hyvin vino jako. Harvinaista!


0
n
n−1
(n−1)/2 − 1
(n−1)/2
Θ(n)
(a)


n
(n−1)/2
(n−1)/2
Θ(n)
(b)

- Pikajärjestämisen *tila*vaativuus on sama kuin rekursion syvyys pahimmillaan, eli
  - rekursiopuun korkeus
  - parhaassa ja keskimääräisessä tapauksessa logaritminen

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

  - pahimmassa lineaarinen

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

- Pikajärjestäminen voidaan toteuttaa myös siten, että rekursiosyvyys on *aina*

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

  - Rivien 3 ja 4 rekursiokutsut saa tehdä kummassa järjestyksessä tahdomme.
  - Tehdään siis ensin se, joka käsittelee *pienempää* osista.
  - Jälkimmäinen (eli suuremman osan) kutsu taas on kalvojen 3.1.3 takarekursiota, eli korvattavissa silmukalla.

6.3.3 Käytännön huomautuksia

- Kalvojen 1.1 lisäysjärjestäminen on lyömätön pienillä aineistoilla.

- Pikajärjestämistä kannattaakin virittellä siten, että

  kun järjesteltävän alueen pituus on enää (esimerkiksi) ≤ 20 alkiota

  niin vaihdetaankin lisäysjärjestämiseen.

  quickSort(A, p, r)
  1 **if** $r - p < 20$ **then** insertionSort(A, p, q)
  2 **else**
  3 $q \leftarrow$ partition(A, p, r)
  4 quickSort(A, p, q)
  5 quickSort($A, q + 1, r$)

- Osituksessa paras valinta jakoalkioksi olisi jaettavan aineiston *keskiluku* eli *mediaani:*
  - silloin jako olisi mahdollisimman tasainen
  - eli pikajärjestämisen pahinkin tapaus olisi

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

- On myös olemassa lineaarinen algoritmi mediaanin valitsemiseksi.
  - Se ratkaisee yleisemmän ongelman "valitse järjestyksessä k:s alkio" josta mediaani saadaan arvolla

$$k = \frac{n}{2}.$$

  - Se on selitetty lukuna 9.3 kirjassa T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.
  - Se perustuu samoihin ideoihin kuin itse pikajärjestäminenkin, esimerkiksi aineiston osittamiseen (kalvoilta 6.3.1).

- Tämän lineaarisen mediaanialgoritmin vakiokerroin on kuitenkin niin suuri, ettei pikajärjestäminen olisikaan enää käytännössä nopeampi kuin keko- ja lomitusjärjestäminen (kalvot 6.1 ja 6.2).

- Melko hyvä keino on valita jako-alkioksi mediaani alueen arvoista

$$\underbrace{A[p]}_{\text{eka}} \qquad \underbrace{A\left[\left\lfloor\frac{p+r}{2}\right\rfloor\right]}_{\text{keskimmäinen}} \qquad \underbrace{A[r]}_{\text{vika.}}$$

- Toinen tapa on
  - valita jakoalkio alueelta satunnaisesti
  - koska pahin tapaus on epätodennäköinen.

- Pahin tapaus pikajärjestämisessä
  - voidaan tehdä niin harvinaiseksi, ettei se esiinny käytännössä juuri koskaan
  - ei ole tehokkaasti vältettävissä kokonaan.

6.4 Järjestämisen alaraja

(Myös kurssilla *Johdatus diskreettiin matematiikkaan*.)

- Osoitetaan nyt, että syötealkioiden

$$d_1, d_2, d_3, \ldots, d_n$$

järjestäminen vie vähintään

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

askelta tietyllä oletuksella.

- Tämä oletus on: Ainoa operaatio, jonka mikään järjestämisalgoritmi α voi syötteilleen a_k tehdä, on niiden keskinäinen vertailu

$$d_i \lesseqqgtr d_j. \qquad (6)$$

Toisin sanoen, syötealkioiden d_i tyyppi on *abstrakti* eikä α voi sitä hyödyntää.

- Piirretään syötealkioiden lukumäärälle n seuraavanlainen binääripuu T_n:
  - Jokaisessa sisäsolmussa on jokin vertailu (6).
  - Sisäsolmun s lapset ovat vertailun (6) tulokset **kyllä / ei**.
  - Puun juuri saadaan simuloimalla algoritmia α sen alusta alkaen, kunnes se tekee ensimmäisen vertailun (6).
  - Solmun s lapset saadaan jatkamalla simulaatiota
    * isäsolmun vertailusta (6) eteenpäin
    * vertailun mahdollisilla tuloksilla.
  - Simulaation päättyminen synnyttää lehtisolmun.

    Lehtisolmussa α tulostaa syötteelle vertailuin päättelemänsä oikean järjestyksen, kuten

$$d_{17}, d_2, d_{53}, \ldots, d_1.$$

- Puu T_n

  kuvaa kaikki algoritmin α suoritusreitit
  kun syötteen pituus on n alkiota

  päätöspuuna joka esittää eri suoritusreitit
  algoritmin α tekeminä polkuina
  vertailusta seuraavaan

  on binääripuu koska vertailulla on
  2 tulosvaihtoehtoa

  on äärellinen koska äärettämässä haarassa
  algoritmi α jäisi ikuiseen silmukkaan

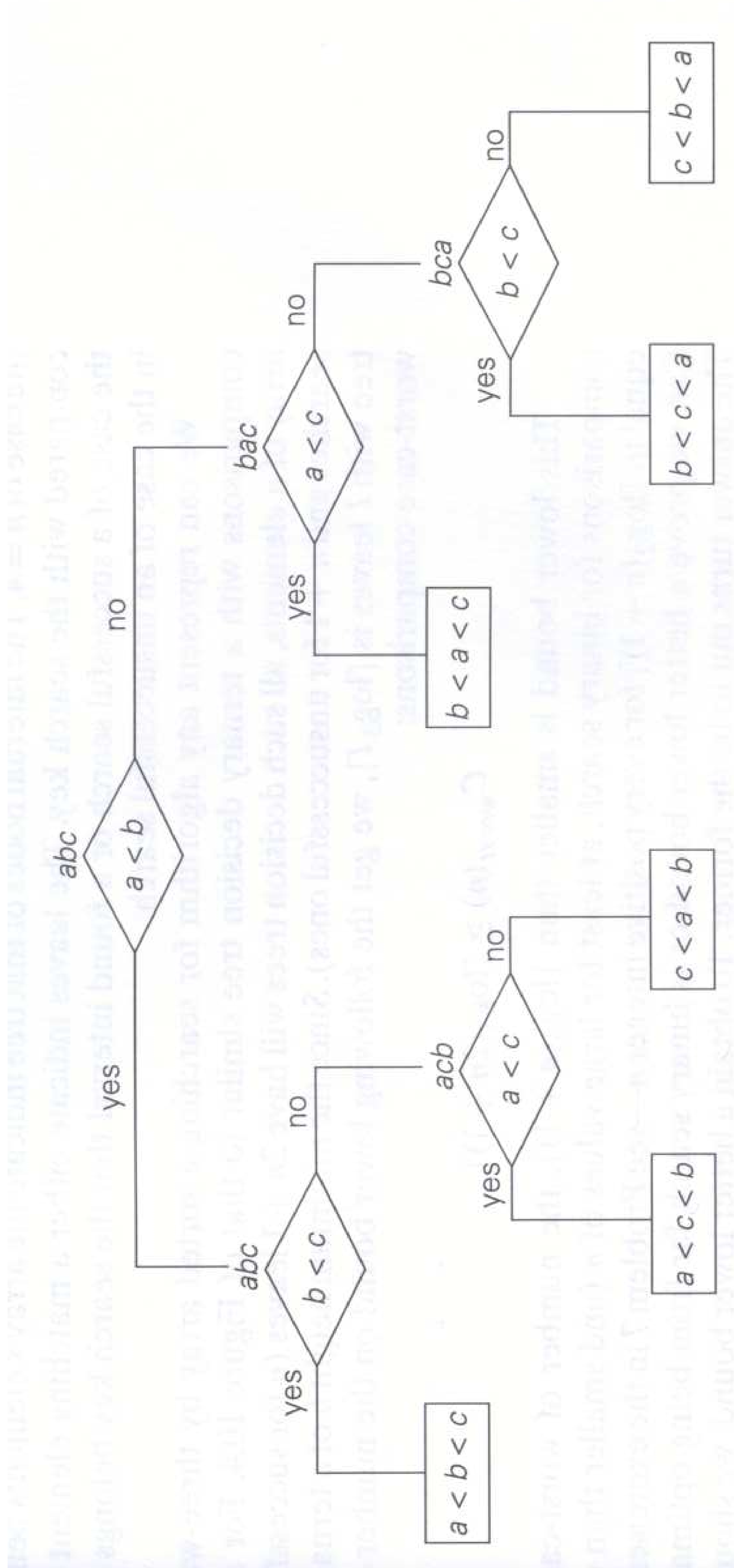
  sisältää vähintään $n!$ lehteä koska
  algoritmin α täytyy pystyä tulostamaan
  syötteen jokainen eri permutaatio.

- Seuraavassa kuvassa on kalvojen 1.1
  lisäysjärjestämisen puu T_3

  jossa syötteet eli taulukon A alkutilanne on
  merkitty

$$d_1 = a \qquad d_2 = b \qquad d_3 = c.$$

(Kuva 10.3 kirjasta A. Levitin: *Introduction to the Design & Analysis of Algorithms.* Addison-Wesley, 2003.)


abc
a < b
yes
no
abc
b < c
bac
a < c
yes
no
yes
no
a < b < c
acb
a < c
b < a < c
bca
b < c
yes
no
yes
no
a < c < b
c < a < b
b < c < a
c < b < a

- Riittää osoittaa, että jokaisen puun T_n korkeus on vähintään

$$\Omega(n \cdot \log n). \qquad (7)$$

- Silloin saadaan vahva tulos:
  - otetaanpa mikä tahansa vertailuihin (6) perustuva järjestämisalgoritmi α
  - ja mikä tahansa syötteiden lukumäärä n
  - niin löytyy syöte $d_1, d_2, d_3, \ldots, d_n$, jolla α seuraa puun T_n pisintä polkua
  - eli tekee vähintään (7) vertailua
  - eli vie vähintään niin monta askelta.

- Tämä on *informaatioteoreettinen alaraja:*

  algoritmin α täytyy kerätä syötteestä vähintään näin paljon informaatiota, jotta sen tulostus olisi varmasti oikein.

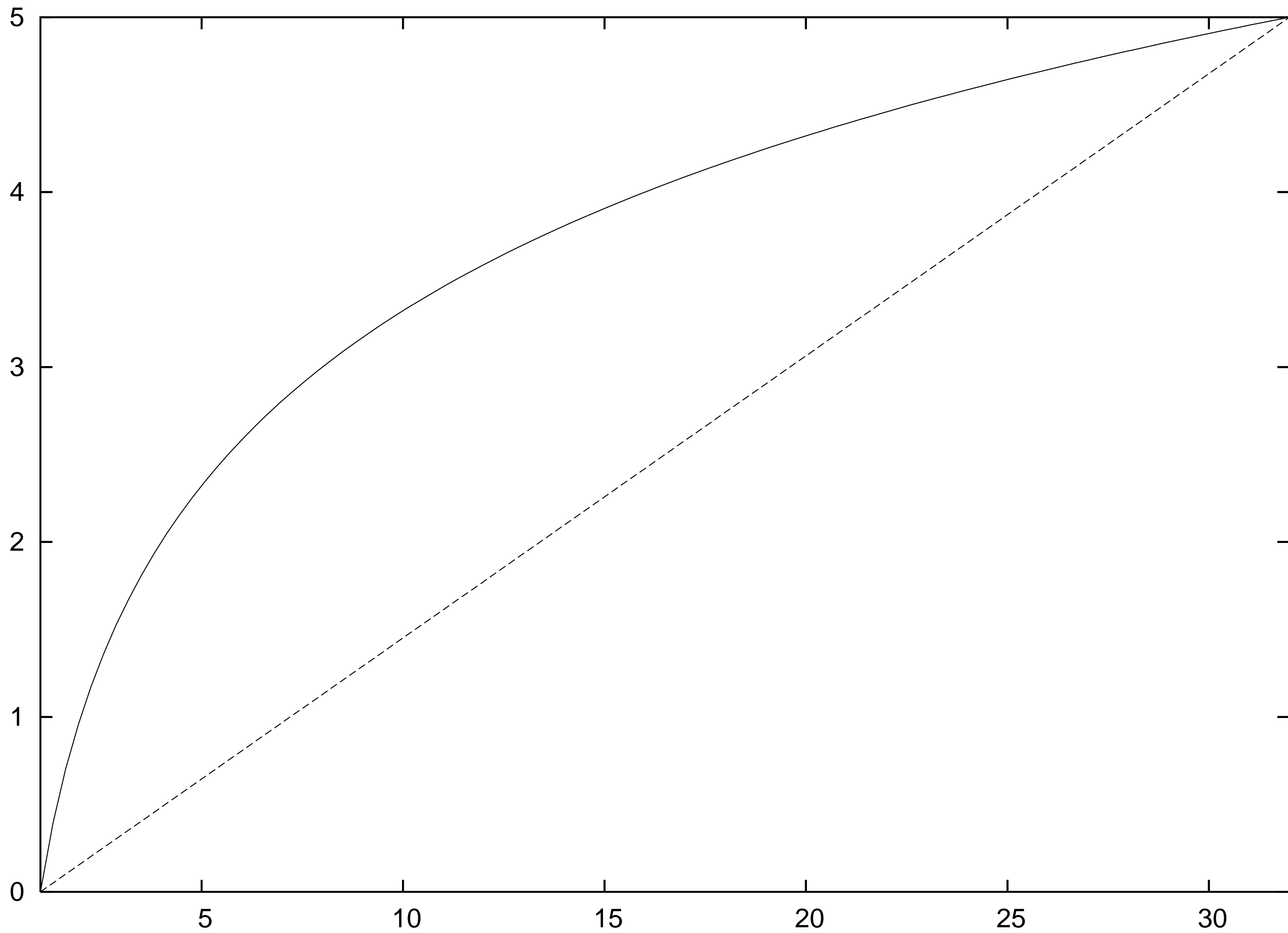
- Tuloksen (7) perustelu:
  - Binääripuussa T_n on enemmän kuin $n!$ solmua, koska jo lehtisolmuja on näin monta.
  - Lauseen 3.3 nojalla puun T_n korkeus on vähintään

$$\log(n!) - 1.$$

  - Arvioidaan

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot 3 \cdot \dots \cdot n) \\ &= \log(1) + \log(2) + \log(3) \\ &\quad + \dots + \log(n) \\ &\geq \frac{\overbrace{(n-1)}^{\text{kanta}} \cdot \overbrace{\log(n)}^{\text{korkeus}}}{2}\end{aligned}$$

missä arvio nähdään seuraavasta kuvasta.


0
1
2
3
4
5
5
10
15
20
25
30

- Kuvassa
  - ylempi käyrä on $y = \log(x)$ välillä
    $$1 \leq x \leq n$$
    (kuvassa $n = 32$)
  - alempi suora kulkee käyrän alkupisteestä loppupisteeseen
  - arvioitava summa koostuu käyrän y-arvoista välin kokonaislukukohdissa
    $$x = 1, 2, 3, \ldots, n$$
  - alaraja on (aritmeettinen) summa suoran y-arvoista samoissa pisteissä.

- Oleellisesti siis
  - näimme kuvasta, että käyrän alle jäävä pinta-ala $\geq$ suoran alle jäävä(n kolmion) pinta-ala
  - käytimme integraalilaskentaa.

- Vastaavalla päätöspuutekniikalla voidaan osoittaa esimerkiksi, että
  - kysymys ”esiintyykö avain x järjestetyssä taulukossa $A[1 \ldots n]$” vaatii ainakin logaritmisen

$$\Omega(\log n)$$

    määrän vertailuja
  - kalvojen 1.3.3 binäärihakua parempaa algoritmia ei voi olla
    * koska se tekee *kaikkiaan* vain saman verran askeleita
    * eli ei tee vertailujen lisäksi mitään tarpeetonta.
  - binäärihaku on *optimaalisen tehokas*.

- Tällaisista alarajoista voidaan päätellä, että jokin on *mahdotonta* käyttäen *ristiriitatodistusta:*

  Väite: On mahdotonta suunnitella sellaista kalvojen 3 mukaista hakupuulajia β, johon lisääminen veisi aina *aidosti vähemmän kuin logaritmisen* ajan sen solmujen lukumäärän suhteen.

  Vastaväite: Sellainen β onkin olemassa.

  Ristiriita: Mutta silloinhan algoritmi

  1. lisää syöte $d_1, d_2, d_3, \ldots, d_n$ aluksi tyhjään β-puuhun

  2. tulosta näin rakennetun β-puun avaimet lauseen 3.4 sisäjärjestyksessä

  olisi mahdottoman nopea järjestämisalgoritmi α.

6.5 Järjestäminen lineaarisessa ajassa

- Kaikki tähän asti tuntemamme järjestämisalgoritmit perustuvat järjestettävän aineiston lukujen keskinäiseen vertailuun (6).

- Kalvoilla 6.4 todistettiin niille aikavaativuuden alaraja (7).

- Koska kalvojen 6.1–6.2 keko- ja lomitusjärjestäminen toimivat pahimmassakin tapauksessa näin nopeasti, niin ne ovat optimaalisen tehokkaita.

- Tehokkuusraja (7) voidaan kuitenkin rikkoa, jos järjestäminen perustuukin *johonkin muuhun* kuin alkioiden keskinäiseen vertailuun (6).

- Vastaavasti kuin kalvojen 4 hajautus oli tehokkaampaa kuin kalvojen 3 hakupuut:

  vertailujen sijasta käytettiinkin indeksointia.

- Oletetaan esimerkiksi, että järjestettävä aineisto $A[1 \ldots n]$ koostuu luvuista
  - joiden arvot ovat väliltä $0, 1, 2, \ldots, k$
  - missä k on niin pieni, että voidaan kayttää aputaulukkoa $C[0 \ldots k]$.

- Yksinkertainen ja tehokas järjestämismenetelmä saadaan aikaan seuraavasti:

  1. lasketaan aputaulukkoon

     $C[i] =$ montako kertaa aineistossa esiintyy luku i itse

  2. tulostetaan sitten taulukkoon A

     (a) ensin $C[0]$ kappaletta lukua 0

     (b) sitten $C[1]$ kappaletta lukua 1

     (c) sitten $C[2]$ kappaletta lukua 2

     ja niin edelleen.

  Näin taulukossa on samat luvut kuin alussa, ja suuruusjärjestyksessä!

- Algoritmina:

```
countingSort1(A, k, n)
1   for i ← 0 to k do C[i] ← 0
2   for j ← 1 to n do
3       x ← A[j]
4       C[x] ← C[x] + 1
5   y ← 1
6   for i ← 0 to k do
7       for j ← 1 to C[i] do
8           A[y] ← i
9           y ← y + 1
```

- Algoritmi
  - käy kerran läpi syötetaulukon A
  - kahteen kertaan aputaulukon C
  - tulostaa luvun jokaiseen taulukon A paikkaan

  joten sen aikavaativuus on siis $\mathcal{O}(n+k)$.

- Jos $k = \mathcal{O}(n)$, niin aikavaativuus onkin *lineaarinen* järjestettävän aineiston koon suhteen.

- Aputilavaativuus on luonnollisesti $\mathcal{O}(k)$.

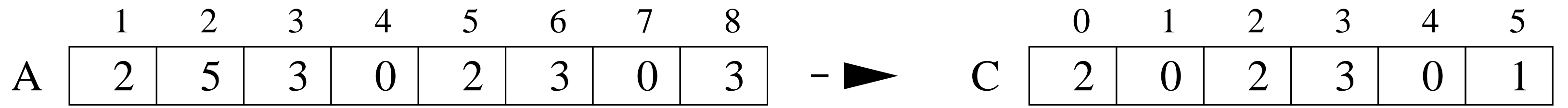
- Esimerkki seuraavana kuvana.

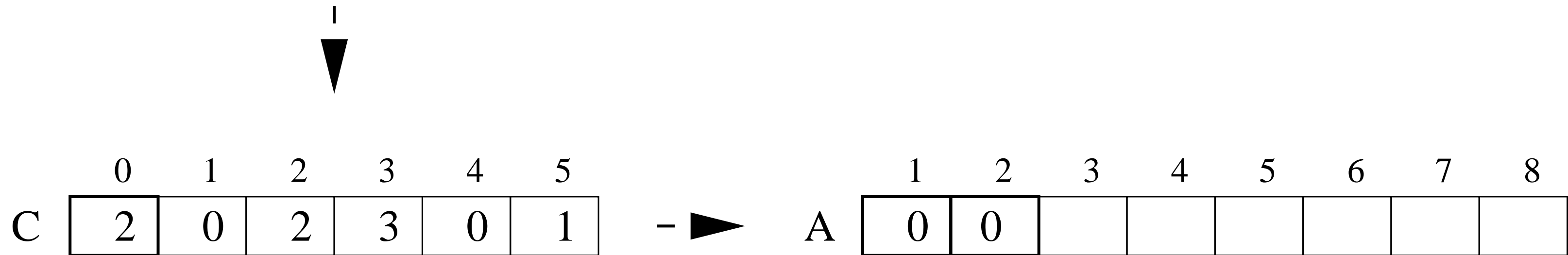
- Kutsutaan algoritmia *laskemisjärjestämiseksi* (counting sort).

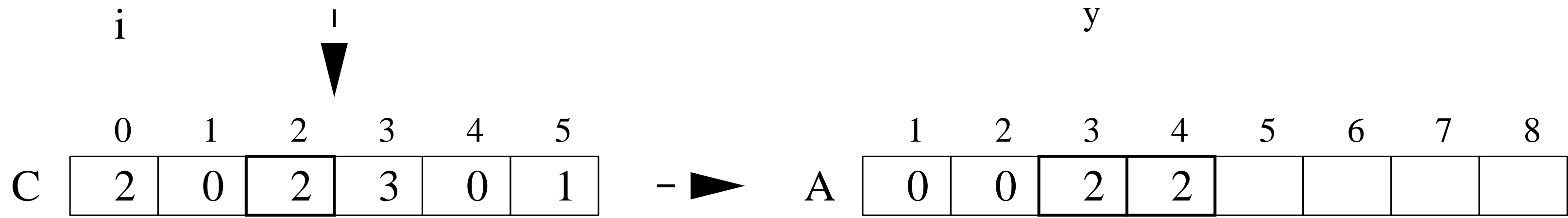
- Kyseessä ei kuitenkaan ole sama versio laskemisjärjestämisestä joka löytyy Cormenista ja Karvista.

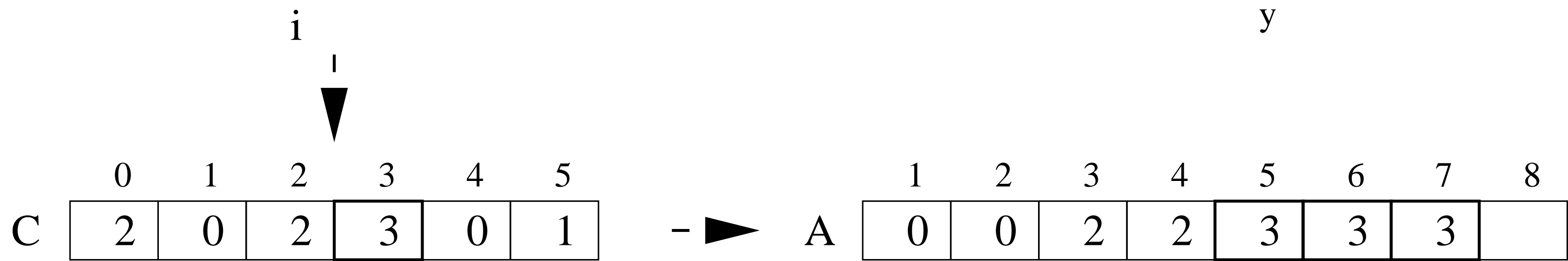
- Jos järjestettäviin alkioihin liittyy muitakin datakenttiä kuin pelkkä luku, niin tämä algoritmi ei toimikaan

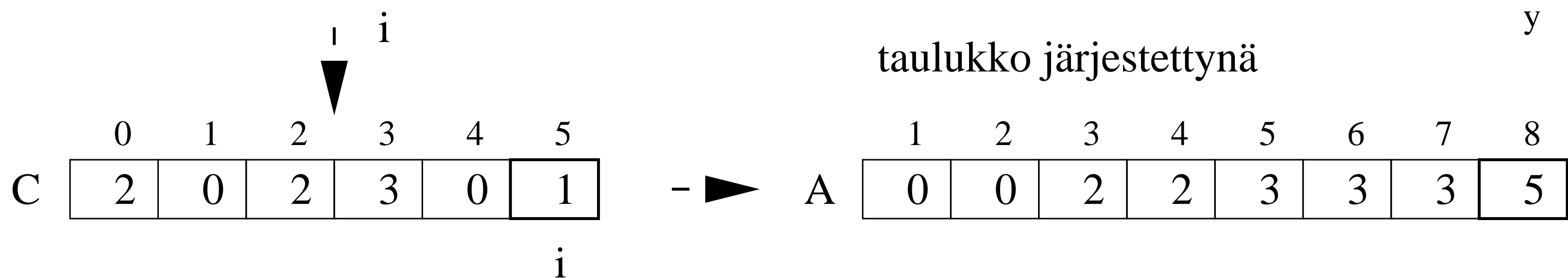
  koska nämä alkioiden datakentät eivät tulostukaan.


taulukko alussa
1 2 3 4 5 6 7 8
A 2 5 3 0 2 3 0 3
0 1 2 3 4 5
C 2 0 2 3 0 1


0 1 2 3 4 5
C 2 0 2 3 0 1
1 2 3 4 5 6 7 8
A 0 0


i
y
0 1 2 3 4 5
C 2 0 2 3 0 1
1 2 3 4 5 6 7 8
A 0 0 2 2


i
y
0 1 2 3 4 5
C 2 0 2 3 0 1
1 2 3 4 5 6 7 8
A 0 0 2 2 3 3 3


i
y
taulukko järjestettynä
0 1 2 3 4 5
C 2 0 2 3 0 1
1 2 3 4 5 6 7 8
A 0 0 2 2 3 3 3 5
i

- Esitetään vielä laskemisjärjestämisestä Cormenin versio joka välttää yllä mainitun ongelman.

countingSort2(A, k, n)
1 **for** $i \leftarrow 0$ **to** k **do** $C[i] \leftarrow 0$
2 **for** $j \leftarrow 1$ **to** n **do**
3 $\quad x \leftarrow A[j]$
4 $\quad C[x] \leftarrow C[x] + 1$
5 **for** $i \leftarrow 1$ **to** k **do**
6 $\quad C[i] \leftarrow C[i] + C[i-1]$
7 **for** $j \leftarrow n$ **downto** 1 **do**
8 $\quad x \leftarrow A[j]$
9 $\quad B[C[x]] \leftarrow x$
10 $\quad C[x] \leftarrow C[x] - 1$
11 **for** $i \leftarrow 1$ **to** n **do** $A[i] \leftarrow B[i]$

- Rivit 1–4 ovat kuten edellä.

- Rivien 5–6 silmukan jälkeen

$$
\begin{aligned}
C[i] &= \text{montako kertaa aineistossa} \\
&\quad\ \text{esiintyy jokin luku} \leq i \\
&= \text{viimeinen taulukkoindeksi} \\
&\quad\ \text{johon saa laittaa luvun} \leq i.
\end{aligned}
$$

- Rivien 7–10 silmukka
  - täyttää toista aputaulukkoa $B[1 \ldots n]$
  - josta jokaiselle luvulle x on varattu oma, oikean kokoinen alue

    $$B[C[x-1]+1 \ldots C[x]]$$

    missä nämä C-arvot ovat ne, jotka vallitsivat ”rivillä $6\frac{1}{2}$” juuri ennen tätä silmukkaa
  - käy läpi syötettä sen lopusta lukien
  - täyttää näitä alueita niiden lopusta lukien.

- Tämä lopusta alkuun -järjestys pitää algoritmin vakaana (kalvojen 6.2 mielessä).

- Lopuksi rivillä 11 tulos on syntynyt taulukkoon B.

- Algoritmi käy läpi
  - taulukot A ja C kahteen kertaan
  - taulukon B kertaalleen

  joten aikavaativuus on

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

- Aputaulukon B pituus on n.

  Aputaulukon C pituus on $k+1$.

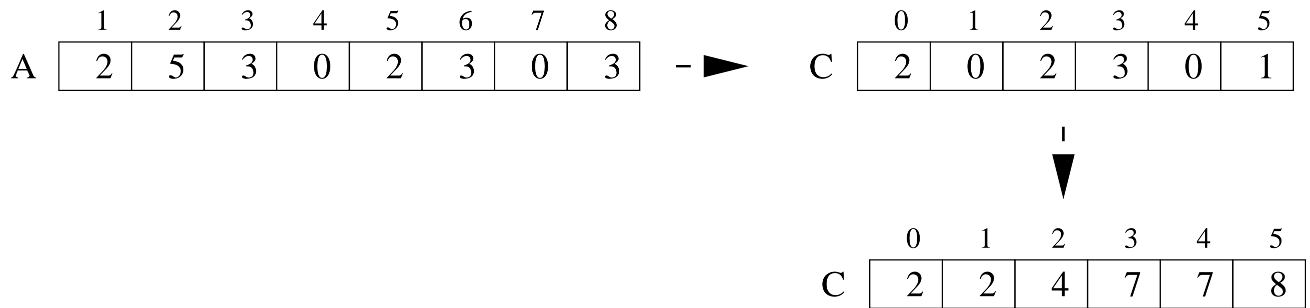
  Aputilavaativuus on siis sama

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

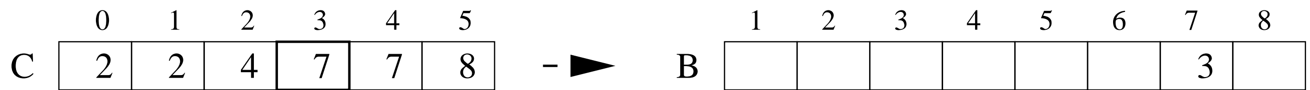
- Edelleen, jos $k = \mathcal{O}(n)$, niin molempina vaativuuksina on $\mathcal{O}(n)$.

- Esimerkki Cormenin laskemisjärjestämisen toiminnasta seuraavilla kahdella kalvolla.

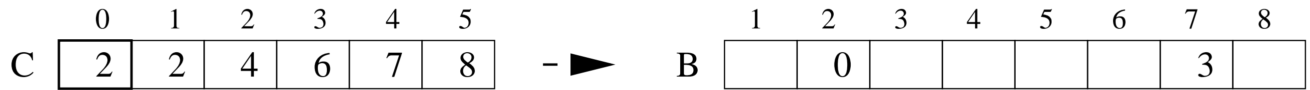
taulukko alussa



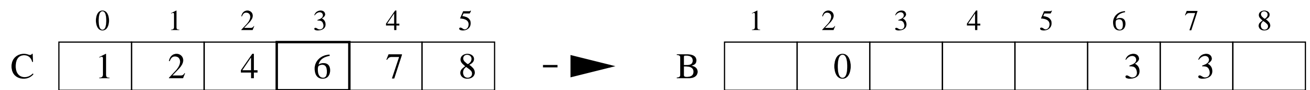
A[8]=3



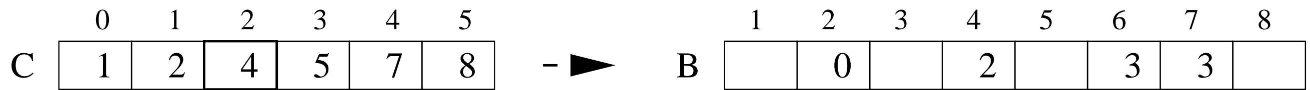
A[7]=0



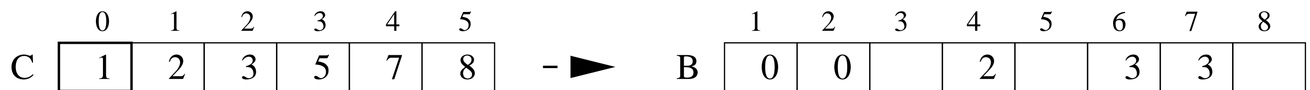
A[6]=3



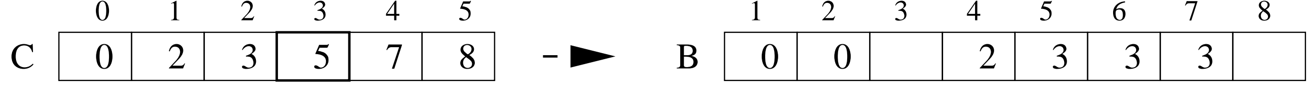
A[5]=2

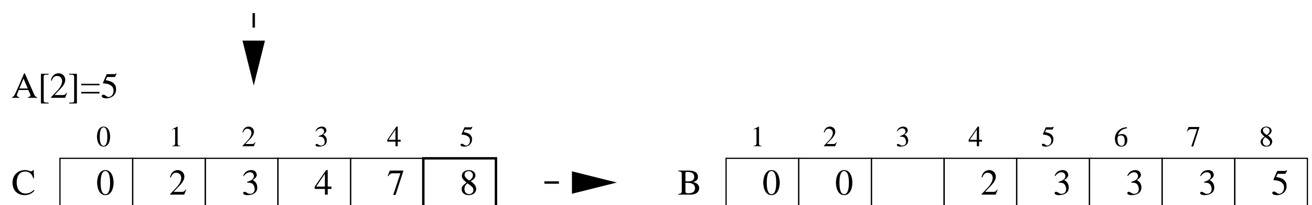


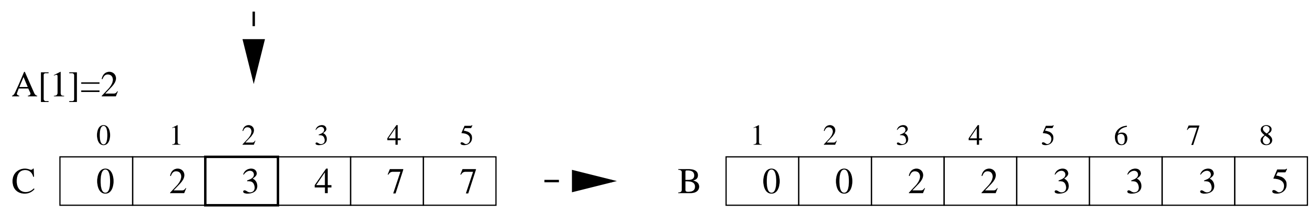
A[4]=0



A[3]=3




A[2]=5
0 1 2 3 4 5
C 0 2 3 4 7 8
B 0 0 2 3 3 3 5
1 2 3 4 5 6 7 8


A[1]=2
0 1 2 3 4 5
C 0 2 3 4 7 7
B 0 0 2 2 3 3 3 5
1 2 3 4 5 6 7 8
taulukko järjestyksessä

- Eräs käyttökohde on kalvojen 6.3 pikajärjestämisen vakauttaminen.

- Pikajärjestäminen on vakaa, jos kalvojen 6.3.1 ositus säilyttää yhtä suurten alkioiden keskinäisen järjestyksen.

- Tehdään siis ositus Cormenin laskemisjärjestämisellä:

  1. Valitaan jakoalkio a.

  2. Määritellään järjestettäville alkioille x $k = 3$ väriä:

$$\text{väri}(x) = \begin{cases} 0 & \text{jos } x < a \\ 1 & \text{jos } x = a \\ 2 & \text{jos } x > a. \end{cases}$$

  3. Laskemisjärjestetään ositettava alue näillä väreillä.

  4. Samalla selviävät rekursiivisesti järjestettävien osien "$x < a$" ja $x > a$" alku- ja loppukohdat.

  5. Osaa "$x = a$" ei tarvitse järjestää.

+ Vakauden lisäksi pystymme näin välttämään jakoalkion a kanssa yhtä suurten alkioiden järjestämisen uudelleen ja uudelleen.

– Jouduimme ottamaan käyttöön lineaarisen aputilan.

– Jouduimme käymään ositettavan aineiston läpi toisenkin kerran.

- Samalla hoitui toinenkin pikajärjestämisen ongelma:
  - Keskenään yhtä suuret alkiot aiheuttavat pahimman tapauksen seuraavasti.
  - Kun jokin niistä valitaan jakoalkioksi, niin muut ositetaan keskenään samaan osaan.
  - Jos jakoalkion kanssa yhtä suuria alkioita on paljon, niin tuloksena on siis epätasapainoinen ositus.
  - Kun rekursiolla edetään yhtä suurten alkioiden yhteiseen osaan, niin ongelma toistuu.
  - Nyt kuitenkin yhtäsuuret alkiot voidaankin jättää pois rekursiosta, eikä tätä ongelmaa enää ole.
  - Hintana ongelman poistamisesta kuitenkin oli, että nyt joudumme käymään ositettavan alueen kaksi kertaa läpi.

6.6 Yhteenveto ominaisuuksista

Aikavaativuus $\mathcal{O}(\ldots)$

algoritmi	pahin	keskim.	paras
kupla (1.3.2)	n^2	n^2	n^2
lisäys (1.1)	n^2	n^2	n
keko (6.1)	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$
lomitus (6.2)	$n \cdot \log n$	$n \cdot \log n$	$n \cdot \log n$
pika (6.3)	teoriassa $n \cdot \log n$ käytännössä n^2	$n \cdot \log n$	$n \cdot \log n$
laskemis (6.5)	$n + k$	$n + k$	$n + k$

Pikajärjestäminen on hyvä valinta, paitsi jos

- järjestettävä aineisto on pieni ($\lessapprox 20$, 1.1)
- jokaisen aineiston pitää järjestyä nopeasti (6.1 / 6.2)
- aineistoista tiedetään jotakin erityistä (kuten 6.5).

Aputilavaativuus:

lomitusjärjestämisellä riippuu toteutuksesta:

muokattavilla listoilla

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

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

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

laskemisjärjestämisellä $\approx$ lineaarinen $\mathcal{O}(n+k)$

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

muilla vakio $\mathcal{O}(1)$.

Erityisesti **kekojärjestäminen** on sekä nopea ja tilaa säästävä.

Jos pitää olla vakaa niin näistä algoritmeista

- lomitus-
- lisäys-
- laskemis-

järjestämiset soveltuvat sellaisinaan.

Muita täytyy muokata (jolloin esim. ajan- tai muistintarve kasvaa).

Kekojärjestäminen on inkrementaalinen:

1. Se voidaan alustaa lineaarisessa ajassa.
2. Alustuksen jälkeen voidaan suorittaa operaatio ”anna järjestyksessä seuraava alkio” logaritmisessa ajassa.

Kaikkia operaatioita ei ole pakko tehdä!

Seuraavilla kalvoilla 6.7 hyödynnetään tätä mahdollisuutta.

6.7 Järjestäminen algoritmin osana

- Järjestämisalgoritmin käyttö aliohjelmana helpottaa monien muiden algoritmien tekemistä.

- Otetaan esimerkkinä seuraava ongelma:

  ”Esiintyykö syötteessä $A[1 \ldots n]$ jokin alkio b useammin kuin kerran?”

  (Vai ovatko kaikki syötealkiot keskenään erisuuria?)

  Tätä syötettä ei siis saada järjestettynä.

Ilman aputietorakenteita neliöisessä ajassa $\mathcal{O}\left(n^2\right)$:

for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        if $A[i] = A[j]$ **then return** true
return false

Tasapainoisella hakupuulla kalvoilta 3 ajassa $\mathcal{O}(n \cdot \log n)$:

```
T ← aluksi tyhjä hakupuu
for i ← 1 to n do
    if search(root[T], A[i]) = NIL then
        insert(T, A[i])
    else
        return true
return false
```

Hajautustaululla kalvoilta 4 — algoritmi sama kuin yllä (olettaa kokonaislukusyötteet):

tyypillisesti nopeammin — jopa $\mathcal{O}(n)$

mahdollisesti hitaasti — jopa $\mathcal{O}\left(n^2\right)$.

Järjestämällä syöte ensin $\mathcal{O}(n \cdot \log n)$:

```
järjestä A
for i ← 1 to n − 1 do
    if A[i] = A[i + 1] then return true
return false
```

Ei vaadi erillistä tietorakennetta.

Virittelemällä järjestysalgoritmia — tällä kertaa kalvojen 6.1 kekojärjestämistä:

```
buildHeap(A)
while heapSize[A] > 1 do
    b ← heapDelMin(A)
    if b = heapMin(A) then return true
return false
```

- Ajantarve on

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

missä $m =$ monesko syötealkio suuruusjärjestyksessä on ensimmäinen joka toistuu.

- Pahin tapaus (mikään alkio ei toistu) on yhä $m = n$.

- Usein kuitenkin $m \ll n$.

Ongelman alaraja onkin kalvojen 6.4 tapaan sama

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

kuin järjestämiselläkin.

7 Verkot

- *Verkko* (engl. graph) koostuu

  solmuista (engl. node, vertex)

  kaarista (engl. edge, arc) jotka yhdistävät solmuja toisiinsa.

  Joskus käytetään myös nimitystä ”särmä”.

- Verkkoja on kahta päätyyppiä:

  Suuntaamattomien verkkojen kaarilla ei ole suuntaa

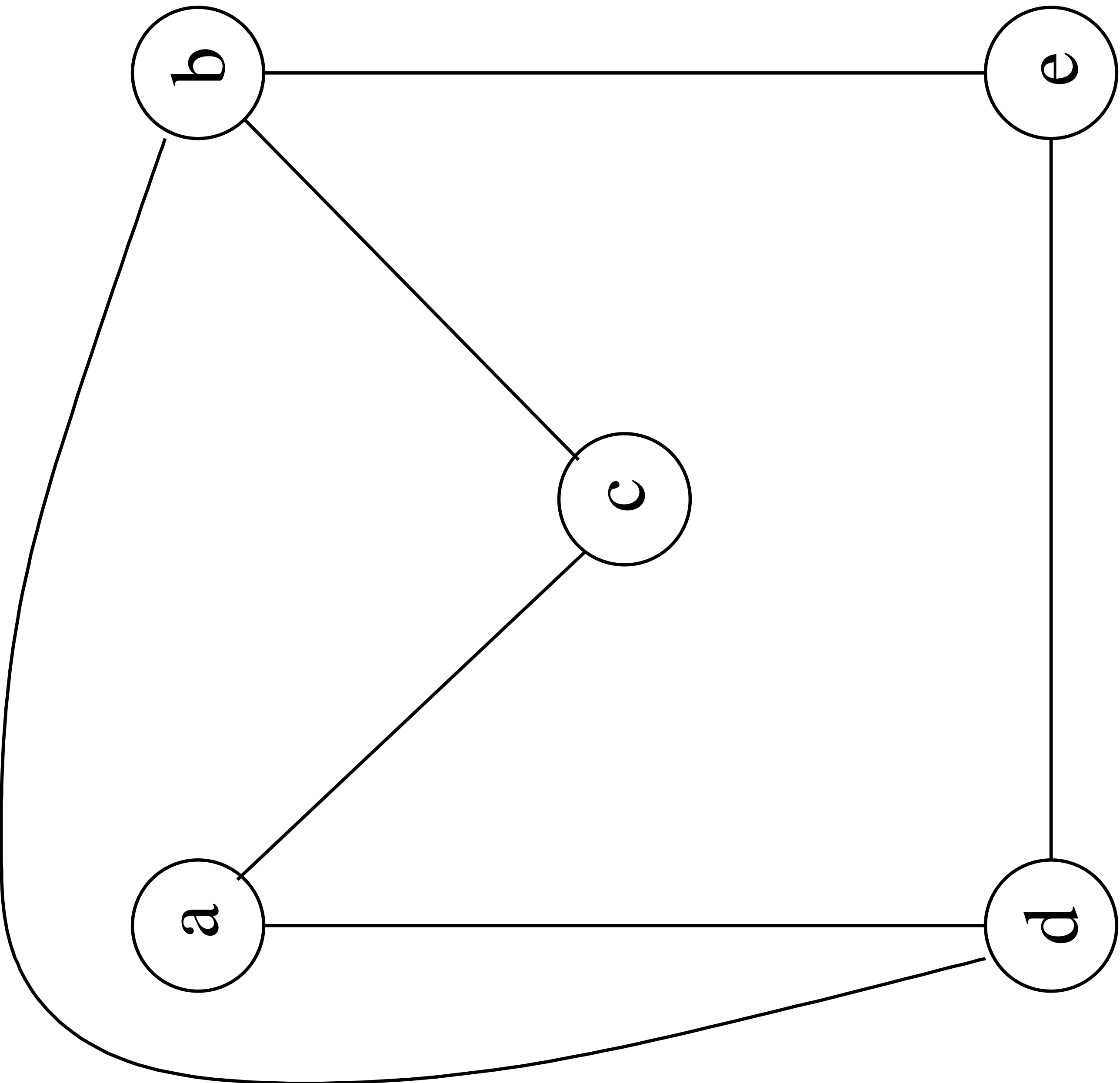
  ne esitetään piirroksina, joissa

  solmut ovat ympyröitä

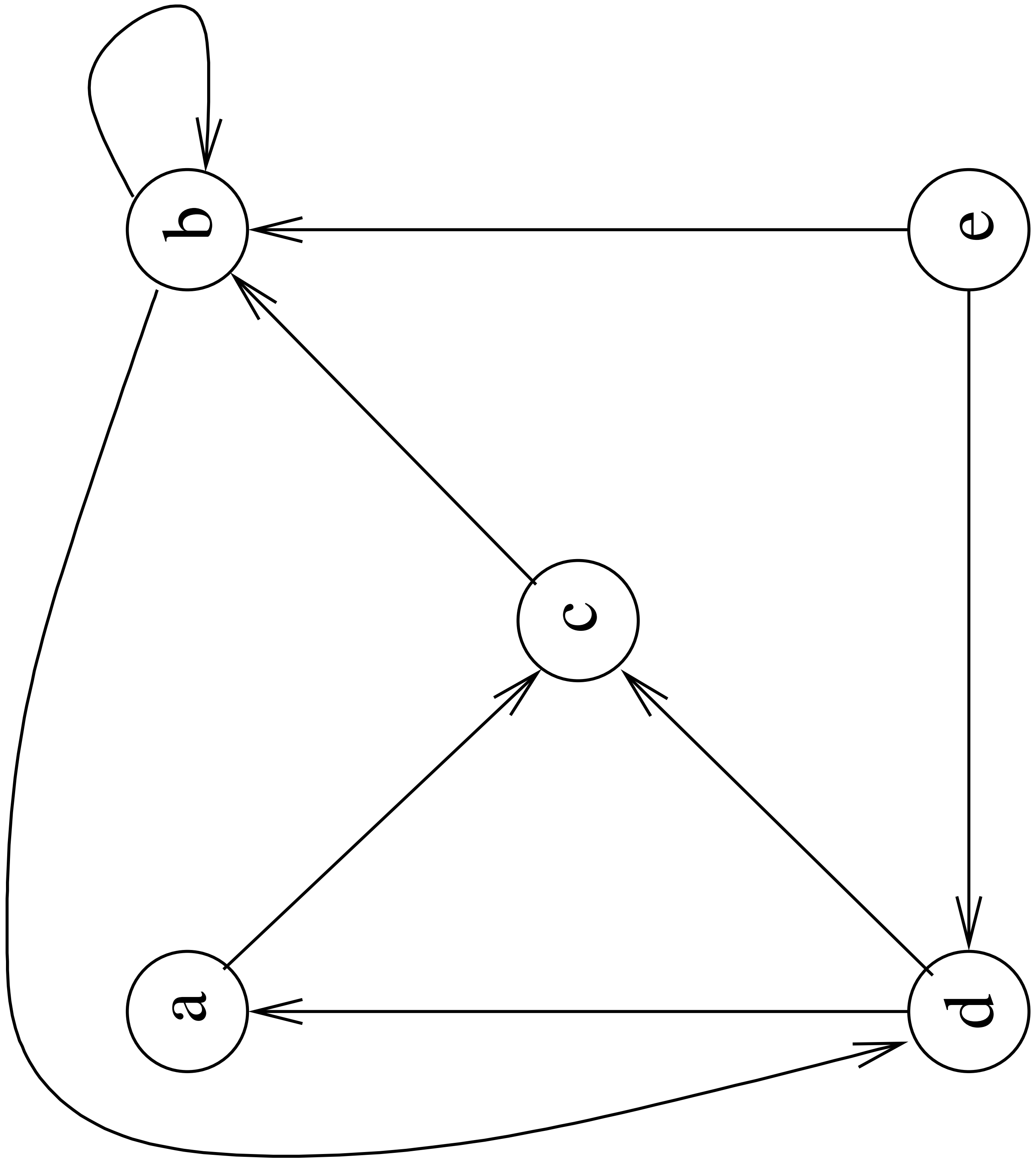
  kaaret (mahdollisesti kaarevia) viivoja niiden välillä.

  Suunnatuissa verkoissa kaarilla on suunta

  eli viivoilla nuolenkärjet.



b
e
c
a
d


b
e
c
a
d

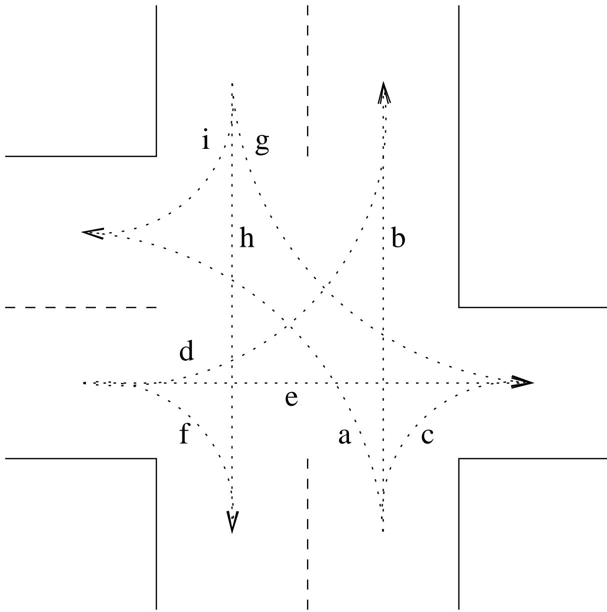
- Verkoilla on paljon sovelluksia tietojenkäsittelyssä.

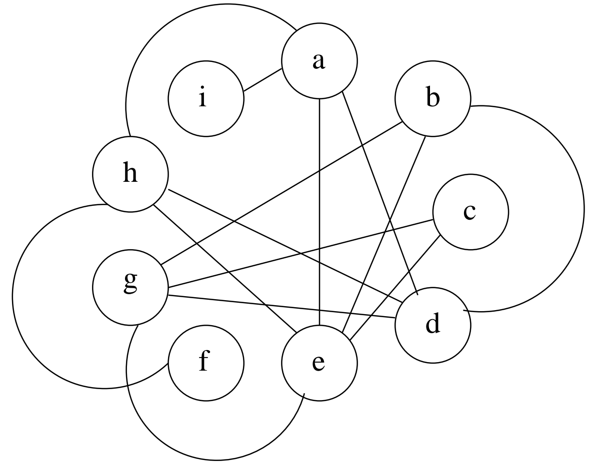
- Otetaan esimerkiksi katuverkko:
  - Seuraavassa kuvassa on yksi risteys.
  - ”Miten Lauttasaaresta pääsee Herttoniemeen?”
    * Mallinnetaan suunnattuna verkkona, jossa

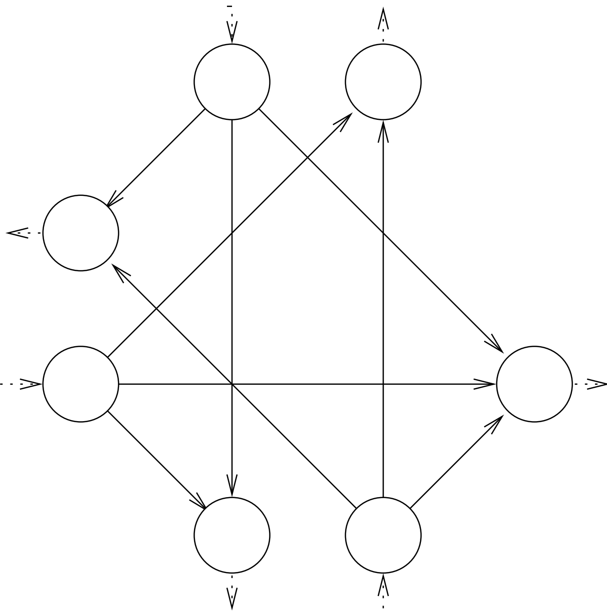
      solmut ovat liikennevirtojen jakautumis- ja yhtymäkohdat

      kaaret nämä virrat.
    * Etsitään reitti kaaria pitkin.
  - ”. . . lyhyintä reittiä?”
    * Lisätään jokaiselle kaarelle sen pituus.
    * Etsitään reitti, jossa kaarten pituuksien summa on mahdollisimman pieni.

  Tutustutaan esimerkiksi näihin algoritmeihin.


i
g
h
b
d
e
f
a
c


a
i
b
h
c
g
d
f
e



- Verkoilla voitaisiin mallintaa myös monia muita katuverkon ongelmia:
  - ”Montako autoa yhteensä pääsee tunnissa Lauttasaaresta Herttoniemeen?”
    * Laitetaan kaaren k ”pituudeksi” sen maksimisiirtokapasiteetti ”n_k autoa tunnissa”.
    * Lasketaan autojen maksimivirtaus.
    * Samalla selviävät liikenteen pullonkaulat: ne kaaret, joilla virtaus on kapasiteettinsa ylärajalla.
  - ”Miten järjestät risteyksen liikennevalot niin, ettei kolareita satu?”

    Solmuina ovatkin nyt liikennevirrat.

    Suuntaamattomina kaarina toisensa leikkaavat virrat.

    Väritetään solmut siten että kaaren päät ovat eri värisiä.

    Samanväriset virrat voi päästää yhtä aikaa risteykseen.

7.1 Käsitteistö

- Formaalisti verkko G esitetään parina (V, E), missä
  - V on solmujen joukko
  - E on kaarien joukko.
- Kaaret ovat siis pareja (p, q) missä p ja q ovat solmuja.
- Suunnatussa verkossa $(p, q) \in E$ jos solmusta p on kaari solmuun q.
  - Tällöin p on kaaren *lähtösolmu* ja q kaaren *maalisolmu*.
  - Solmua q sanotaan solmun p *vierussolmuksi*.
  - Suunnatun verkon kaarista käytetään usein myös merkintää

$$p \rightarrow q.$$

- Kalvojen 7 suunnatussa esimerkkiverkossa
  - solmun e vierussolmut ovat siis b ja d
  - solmun a ainoa vierussolmu on c.
- Sen verkon formaali määritelmä on

$$\begin{aligned} V &= \{a, b, c, d, e\} \\ E &= \{(a, c), (b, b), (b, d), (c, b), \\ &\qquad (d, a), (d, c), (e, b), (e, d)\}. \end{aligned}$$

- Suuntaamattomassa verkossa kaarten joukko E on *symmetrinen*, eli jos $(u,v) \in E$ niin myös $(v,u) \in E$.
  - Merkitsemme suuntaamattoman verkon kaaria u—v.
  - Jos $(u,v) \in E$, niin
    * sanotaan että solmut u ja v ovat *vierekkäisiä*, (engl. adjacent)
    * eli v on solmun u vierussolmu ja u on solmun v vierussolmu.

- Kalvojen 7 suuntaamaton esimerkkiverkko formaalisti määriteltynä:

$$
\begin{aligned}
V &= \{a,b,c,d,e\} \\
E &= \{(a,c),(c,a),(b,d),(d,b),(c,b),(b,c), \\
&\qquad (d,a),(a,d),(e,b),(b,e),(e,d),(d,e)\}.
\end{aligned}
$$

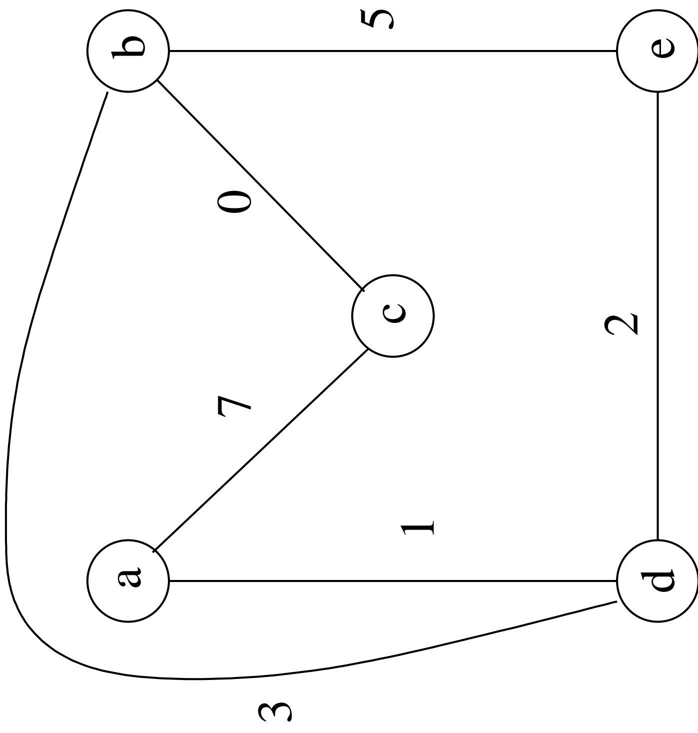
- Usein verkon kaariin liitetään *paino* (engl. weight).

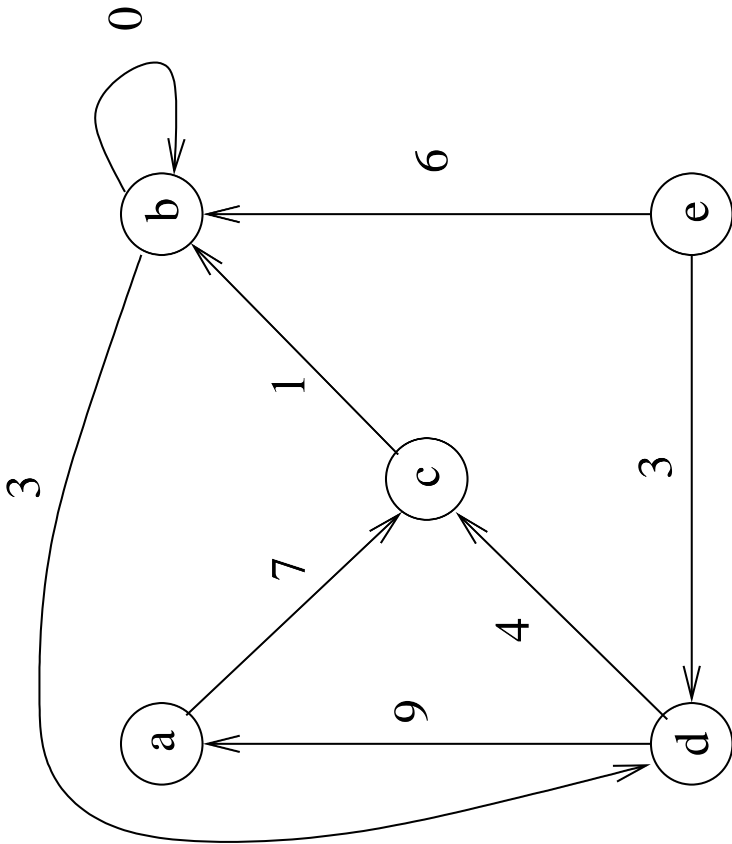
- Kaaripainon käsite voidaan määritellä funktiona
$$w\colon E \mapsto \mathbb{R}.$$
(Myös muita kuin reaalilukupainoja voi toki käyttää.)

- Funktio w liittää jokaiseen kaareen (p, q) sen painon $w(p, q)$.

- Esimerkiksi seuraavan kalvon suunnatussa verkossa
$$w(a, c) = 7$$
$$w(e, d) = 3 \qquad \text{jne.}$$

- Painotetun verkon kaarista $(u, v) \in E$ käytetään myös merkintää
$$u \xrightarrow{w(u,v)} v$$
eli esimerkissämme on kaari
$$a \xrightarrow{7} c \qquad \text{jne.}$$


b
5
e
0
c
7
2
a
1
d
3


0
6
b
e
3
1
c
3
7
4
a
9
d

- Solmujono $v_1, v_2, v_3, \ldots, v_n$ on *polku*

  lähtösolmusta v_1

  kohdesolmuun v_n

  jos verkko sisältää peräkkäiset kaaret

$$v_1 \rightarrow v_2$$
$$v_2 \rightarrow v_3$$
$$\vdots$$
$$v_{n-1} \rightarrow v_n.$$

- Joskus käytetään merkintää

$$v_1 \rightsquigarrow v_n.$$

- Silloin sanotaan että solmu v_n on *saavutettavissa* solmusta v_1.

- *Polun pituus* on polkuun liittyvien kaarien lukumäärä.

- Painotetussa verkossa koko *polun paino* on polun kaarien yhteenlaskettu paino.

- Polku on *yksinkertainen*, jos kukin solmu esiintyy polussa vain kerran.

  - Poikkeus: lähtö- ja kohdesolmut saavat olla samat.

  - Silloin kyseessä on *sykli*.

- Edellisen kuvan suunnatun painotetun verkon polkuja:

$$e \xrightarrow{3} d \xrightarrow{4} c \xrightarrow{1} b$$

  on yksinkertainen syklitön polku jonka pituus on 3 kaarta ja paino 8 yksikköä, ja

$$d \xrightarrow{9} a \xrightarrow{7} c \xrightarrow{1} b \xrightarrow{3} d$$

  on sykli jonka pituus on 4 ja paino 20, kun taas

$$c \xrightarrow{1} b \xrightarrow{0} b \xrightarrow{0} b \xrightarrow{3} d$$

  on polku jonka pituus on 4, paino 4 ja joka *sisältää* kaksi sykliä.

- Englannin kielessä *suunnatusta syklittömästä verkosta* käytetään joskus substantiivia *DAG* (Directed Acyclic Graph)

- Sieltä on suomalaiseen ATK-slangiin kotiutunut *dägi*.

7.2 Verkkojen tallettaminen

- Tarkastellaan seuraavassa tapoja verkon $G = (V, E)$ tallettamiselle tietokoneen muistiin.

- Merkitään

  solmujen lukumäärää $|V|$

  kaarien lukumäärää $|E|$.

  - Joukoilla itseisarvomerkintä $|\ldots|$ tarkoittaa niiden mahtavuutta.

  - Äärellisillä joukoilla mahtavuus = alkioiden lukumäärä.

- Keskeisiä vaihtoehtoisia talletustapoja on kaksi:

  vieruslistat (engl. adjacency lists)
  kalvoilla 7.2.1

  vierusmatriisit (engl. adjacency matrices)
  kalvoilla 7.2.2.

7.2.1 Vieruslistat

- *Vieruslistaesityksessä* verkko $G = (V, E)$ esitetään taulukkona Adj
  - joka sisältää $|V|$ kappaletta linkitettyjä listoja
  - yhden kullekin verkon solmulle.

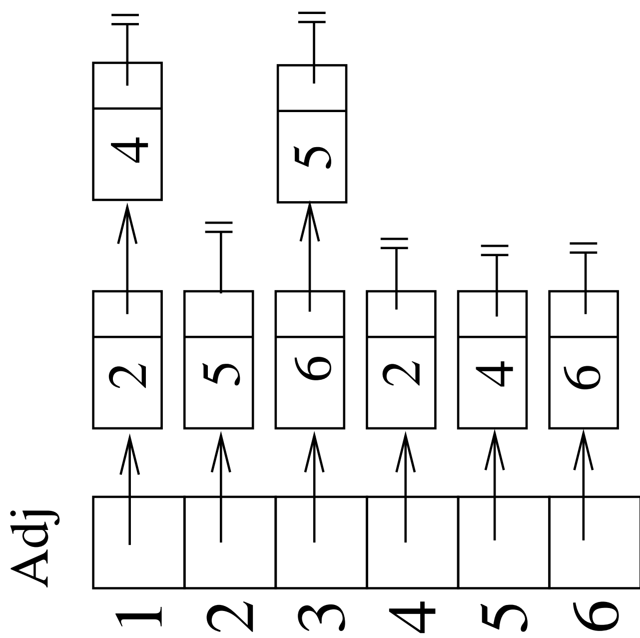
- Oletamme nyt, että solmut on numeroitu

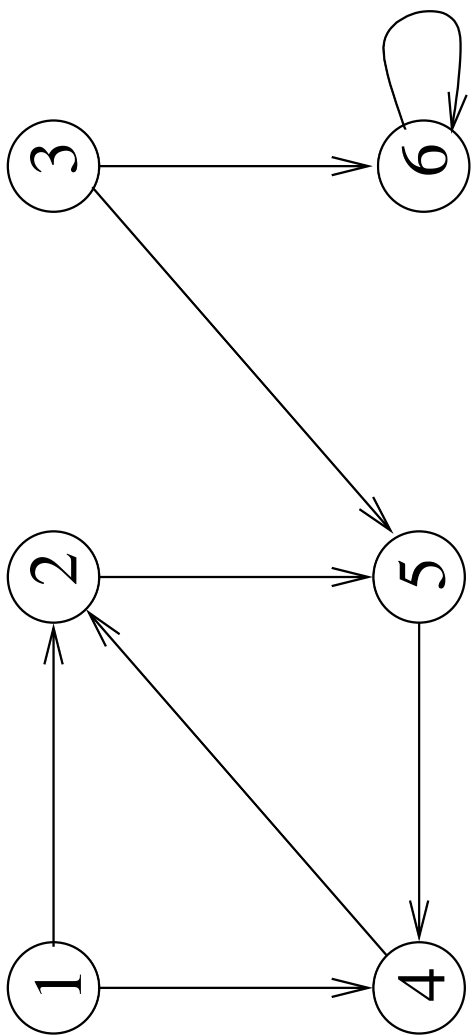
  $$1, 2, 3, \ldots, |V|$$

  ja käytämme solmun numeroa taulukkoindeksinä.

- Siis jokaiselle solmulle $u \in V$ lista Adj$[u]$ sisältää kaikki ne solmut joihin solmusta u on kaari.

- Seuraavana kalvona on suunnattu verkko ja sen vieruslistaesitys.


Adj
1
2
3
4
5
6
2
4
5
6
5
2
4
6


1
2
3
4
5
6

- Suunnatun verkon vieruslistojen yhteenlaskettu pituus on $|E|$:

  jokainen kaari on talletettu kertaalleen yhteen vieruslistoista.

- Koko vieruslistaesitys vie suunnattujen verkkojen tapauksessa tilaa

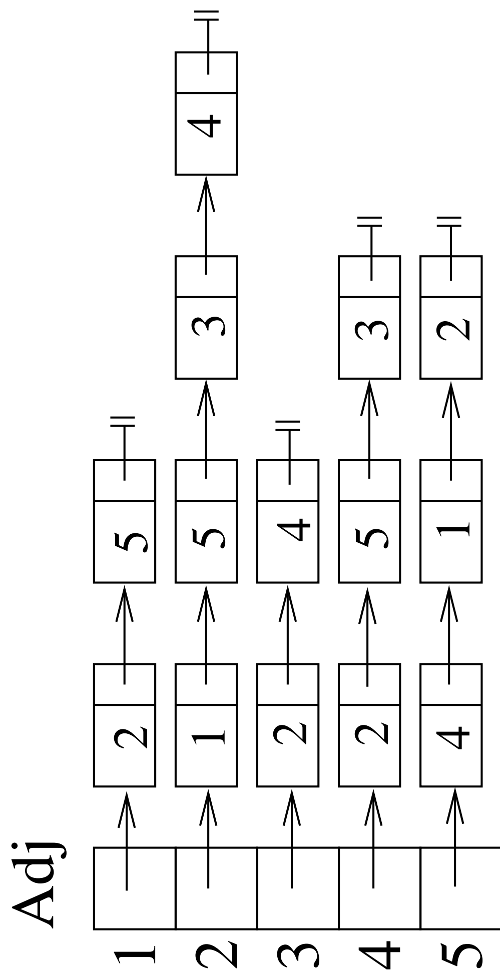
  $$\mathcal{O}\left(|E| + |V|\right):$$

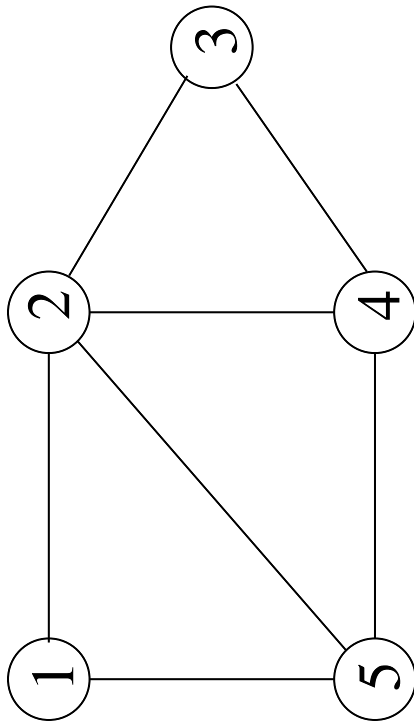
  kaarien lisäksi varataan luonnollisesti tila taulukolle Adj.

- Seuraavassa kuvassa on suuntaamaton verkko ja sen vieruslistaesitys.

- Suuntaamattoman verkon vieruslistojen yhteenlaskettu pituus on $|E|$ on kaksi kertaa kaarien lukumäärä:

  jokainen kaari (p, q) on talletettu kummankin päätesolmunsa p ja q vieruslistaan.


Adj
1
2
3
4
5


1
2
3
4
5

- Koko vieruslistaesitys vie suuntaamattomienkin verkkojen tapauksessa tilaa

$$\mathcal{O}\left(|V| + 2 \cdot |E|\right) = \mathcal{O}\left(|V| + |E|\right).$$

- Myös kaarien painot voidaan tallentaa vieruslistarakenteeseen seuraavan kuvan mukaisesti.

- Vieruslistaesityksen **hyvä** puoli on siis kohtuullinen eli *lineaarinen* tilavaativuus suhteessa solmujen ja kaarten määrään.

- **Huono** puoli on taas se, että tieto onko verkossa kaarta $u \to v$ ei ole suoraan saatavilla, vaan vaatii vieruslistan Adj$[u]$ läpikäynnin.

- Pahimmillaan tämä operaatio vie aikaa $\mathcal{O}\left(|V|\right)$, sillä solmusta u voi olla pahimmassa tapauksessa kaari kaikkiin verkon solmuihin.

- Onneksi useimmat verkkoalgoritmit käyvät läpi solmun u kaikki vierussolmut v samalla kertaa, eli koko vieruslistan Adj$[u]$.

Adj

a → (b, 1) → (d, 4)
b → (e, 2)
c → (f, 6) → (e, 4)
d → (b, 7)
e → (d, 9)
f → (f, 1)

7.2.2 Vierusmatriisit

- Myös verkon $G = (V, E)$ *vierusmatriisiesityksessä* oletetaan, että solmut on numeroitu taulukkoindeksointiin sopivasti:

$$V = \{1, 2, 3, \ldots, n\}.$$

- Vierusmatriisi on $n \times n$-matriisi A, jossa

$$A[i][j] = \begin{cases} 1 & \text{jos } (i,j) \in E \\ 0 & \text{muuten.} \end{cases}$$

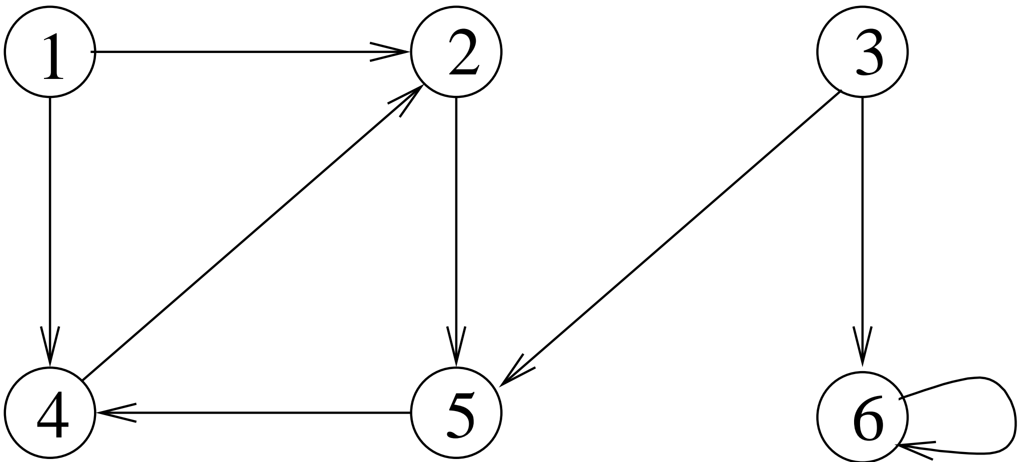
  (Myös totuusarvoja **true**/**false** voi käyttää.)

- Seuraavina kuvina on esimerkit

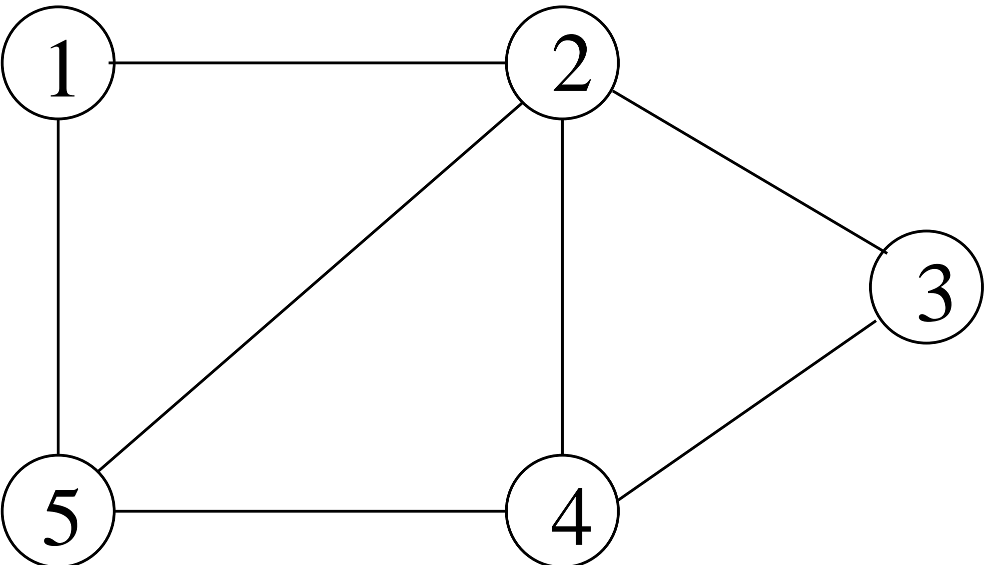
  1. suunnatusta

  2. suuntaamattomasta

  verkosta ja sen vierusmatriisista.



	1	2	3	4	5	6
1	0	**1**	0	**1**	0	0
2	0	0	0	0	**1**	0
3	0	0	0	0	**1**	**1**
4	0	**1**	0	0	0	0
5	0	0	0	**1**	0	0
6	0	0	0	0	0	**1**



	1	2	3	4	5
1	0	**1**	0	0	**1**
2	**1**	0	**1**	**1**	**1**
3	0	**1**	0	**1**	0
4	0	**1**	**1**	0	**1**
5	**1**	**1**	0	**1**	0

- Suuntaamattoman verkon tapauksessa jokainen kaari on rekisteröity *kahteen* kertaan vierusmatriisiin:

  koska edellisessä kuvassa $(2,5) \in E$, niin $A[2][5] = A[5][2] = 1$.

- Suuntaamattoman verkon tapauksessa riittäisikin siirtymämatriisista *puolikas*

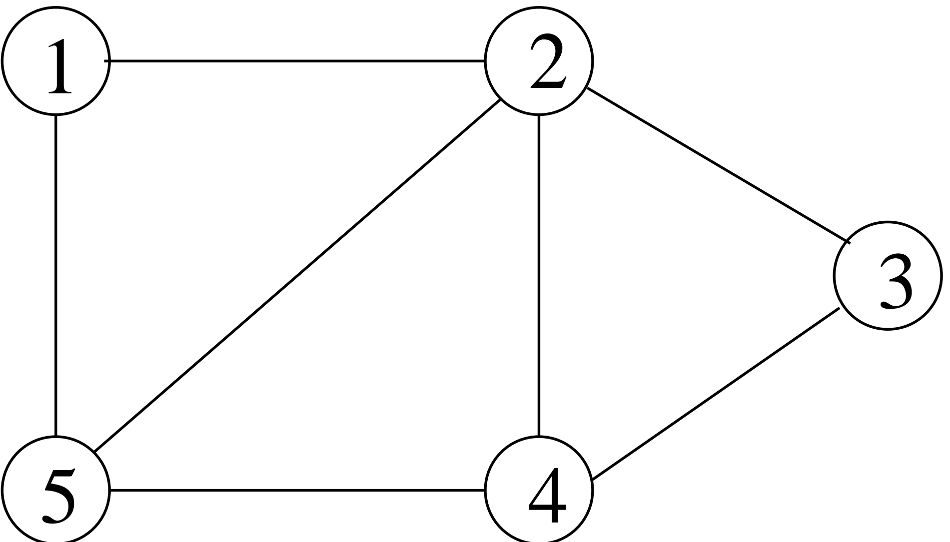
  seuraavan kuvan mukaisesti.

- Joskus ei edes sallita kaaria muotoa (i,i), eli solmusta i takaisin itseensä.

  Silloin ei myöskään tarvita vierusmatriisin *diagonaalia*, eli alkioita

$$A[1][1], A[2][2], A[3][3], \ldots, A[n][n].$$

	1	2	3	4	5
1	*0*	**1**	0	0	**1**
2		*0*	**1**	**1**	**1**
3			*0*	**1**	0
4				*0*	**1**
5					*0*



- Vierusmatriisia voi käyttää myös kaaripainojen talletuspaikkana

  seuraavan kuvan mukaisesti.

- Painotetun verkon vierusmatriisissa periaatteena on siis asettaa

$$A[i][j] = \begin{cases} w(i,j) & \text{jos } (i,j) \in E \\ \infty \text{ tai } 0 & \text{muuten.} \end{cases}$$

Ääretön on luonteva valinta, jos kaaripaino kuvaa

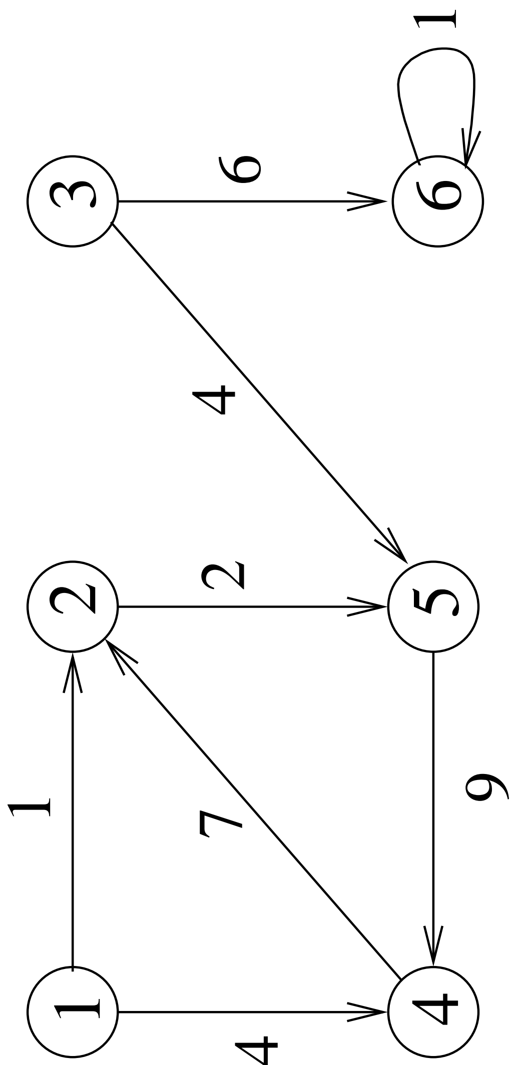
  - kaaren pituutta

  - sen kulkemisen kustannusta

  tai muuta suuretta, jossa ∞ tarkoittaa "älä käytä tätä kaarta".

Nolla jos se kuvaa kapasiteettia tms.

	1	2	3	4	5	6
1	∞	**1**	∞	**4**	∞	∞
2	∞	∞	∞	∞	**2**	∞
3	∞	∞	∞	∞	**4**	**6**
4	∞	**7**	∞	∞	∞	∞
5	∞	∞	∞	**9**	∞	∞
6	∞	∞	∞	∞	∞	**1**



Hyvä puoli vierusmatriisissa on se että kaaren olemassaolon näkee matriisista suoraan.

Huono puoli on taas tilan tarve:

matriisin koko on kaarien lukumäärästä riippumatta aina neliöinen

$$n \times n = \mathcal{O}\left(n^2\right).$$

- Verkkoa sanotaan *harvaksi* jos kaaria on suhteellisen vähän
  - eli vain lineaarinen määrä

    $$|E| = \mathcal{O}(n)$$

  - kun taas maksimimäärä olisi neliöinen

    $$|E| = \mathcal{O}\left(n^2\right).$$

- Kalvojen 7.2.1 vieruslistaesitys pystyy säästämään tilaa kun verkko on harva, matriisi ei.

- On kuitenkin muutamia verkkoalgoritmeja, jotka olettavat matriisiesityksen, koska ne tutkivat verkkoa *hajasaantina*.

7.3 Verkon läpikäynti

- Tyypillistä verkkoa käytettäessä on, että halutaan kulkea verkossa systemaattisesti vieraillen kaikissa (jostakin aloitussolmusta saavutettavissa olevissa) solmuissa.

- Läpikäyntiin on kaksi perusstrategiaa:

  leveyssuuntainen läpikäynti (engl. breadth-first search) kalvoilla 7.3.1

  syvyyssuuntainen läpikäynti (engl. depth-first search) kalvoilla 7.3.2.

- Tutustuimme niiden vastineisiin puissa kalvoilla 3.6.

7.3.1 Leveyssuuntainen läpikäynti

- Verkon $G = (V, E)$ *leveyssuuntaisessa läpikäynnissä* tutkitaan, mitkä verkon solmuista ovat saavutettavissa annetusta aloitussolmusta $s \in V$.

- Läpikäynti etenee uusiin solmuihin "taso kerrallaan":

  1. Ensin käsitellään kaikki ne solmut, joihin pääsee solmusta s *yhtä* kaarta pitkin.

  2. Sitten ne, joihin täytyy kulkea *kahta* pitkin.

  3. Sitten *kolmea*,...

- Algoritmin sivutuotteena saadaan näin jokaiselle kohdatulle solmulle v laskettua attribuutti

  $d[v]$ = lyhyimmän polun $s \rightsquigarrow v$ pituus.

- Toisena sivutuotteena algoritmi muodostaa verkkoa läpikäydessään *leveyssuuntaispuuta* (engl. breadth-first tree):
  - Puu kertoo, mitä reittiä läpikäynti on edennyt kuhunkin solmuun v.
  - Algoritmin suorituksen jälkeen puun polku solmusta s solmuun v vastaa verkon (jotakin) lyhyintä polkua
$$s \rightsquigarrow v.$$
  - Puun kaaret talletetaan verkon solmuihin käyttäen viiteattribuuttia
$$p[v] = \text{se solmu } u \text{ josta tänne tultiin.}$$
  - Silloin verkon lyhyin polku voidaan lukea
$$v \leftarrow p[v] \leftarrow p[p[v]] \leftarrow p[p[p[v]]] \leftarrow \ldots \leftarrow s.$$
  - Siis puu esittää polut *takaperin*.

- Algoritmin omaa kirjanpitoa varten verkon solmuihin tarvitaan vielä kolmaskin attribuutti:

$$\text{color}[v] = \text{solmun } v \textit{ väri}.$$

Valkoinen: Vielä ei ole edetty solmuun v saakka.

Harmaa: On edetty solmuun v itseensä saakka

mutta vielä ei ole edetty solmusta v eteenpäin siitä lähteviä kaaria pitkin.

Musta: Myös solmusta v lähtevät kaaret on käsitelty

eli solmu v on kokonaan käsitelty.

- Aluksi

aloitussolmu s on harmaa

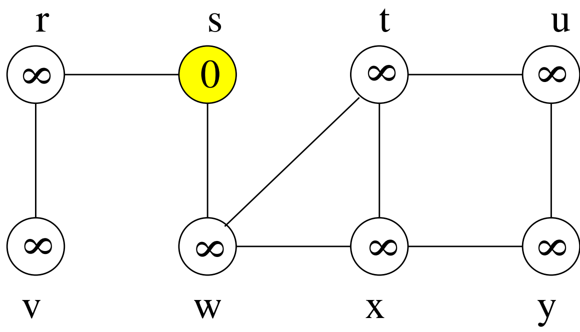
kaikki muut solmut valkoisia.

- Algoritmi käyttää aputietorakenteenaan kalvojen 2.2 *jonoa* Q:
  - $Q =$ ne solmut, jotka ovat tällä hetkellä harmaita
  - Jonokuri takaa, että seuraavaksi käsitellään aina se harmaa solmu, johon edettiin ensimmäiseksi.
  - Se puolestaan takaa, että seuraavaksi pidennetään aina (jotakin) lyhyintä polkua.
  - Polku pidentyy
    * haarautumalla nykyisen päätesolmunsa valkoisiin vierussolmuihin
    * yhdellä uudella kaarella / haara.
- Algoritmin syötteenä on
  - verkko $G = (V, E)$
  - kalvojen 7.2.1 vieruslistaesityksessä
  - aloitussolmu $s \in V$.

```
BFS(G, s)
1    for jokaiselle solmulle u ∈ V do
2        color[u] ← white
3        d[u] ← ∞
4        p[u] ← NIL
5    color[s] ← gray
6    d[s] ← 0
7    enqueue(Q, s)
8    while not empty(Q) do
9        u ← dequeue(Q)
10       for jokaiselle solmulle v ∈ Adj[u] do
11           if color[v] = white then
12               color[v] ← gray
13               d[v] ← d[u] + 1
14               p[v] ← u
15               enqueue(Q, v)
16       color[u] ← black
```

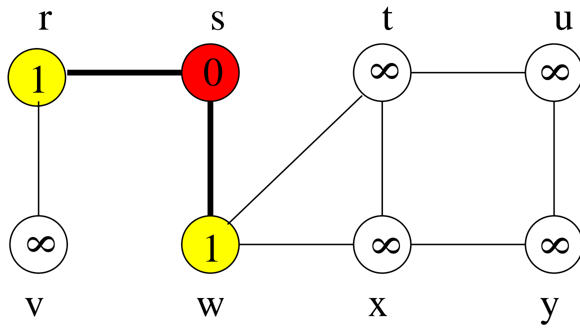
- Algoritmi toimii sekä suunnatuilla että suuntaamattomilla verkoilla.

- Seuraavat 2 kalvoa ovat suoritusesimerkki:
  - Harmaat solmut on merkitty keltaisella, mustat punaisella.
  - Solmuun u on merkitty sen nykyinen $d[u]$.



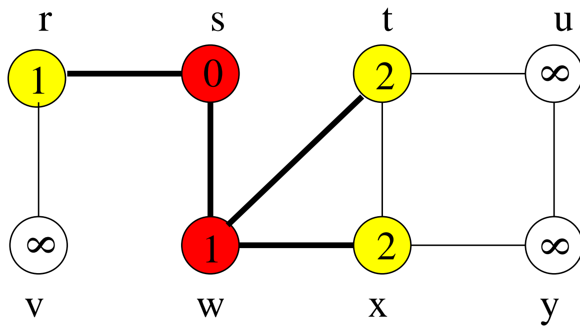
Q

s



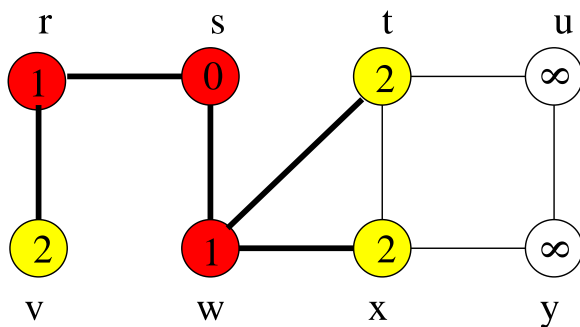
Q

w	r



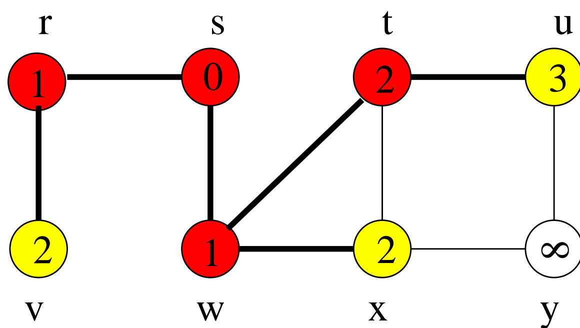
Q

r	t	x



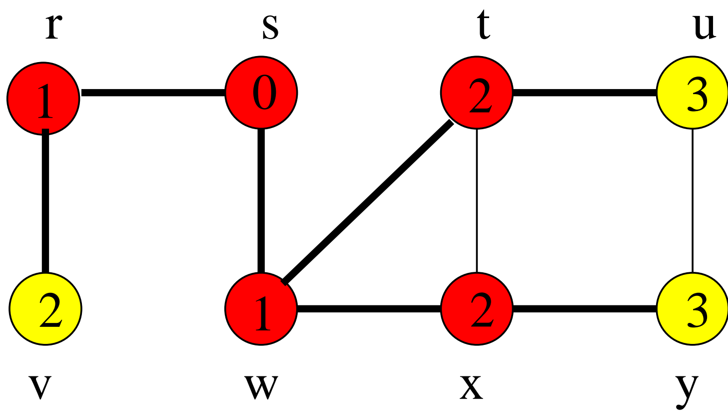
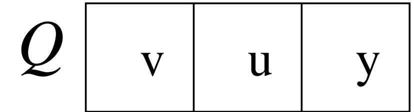
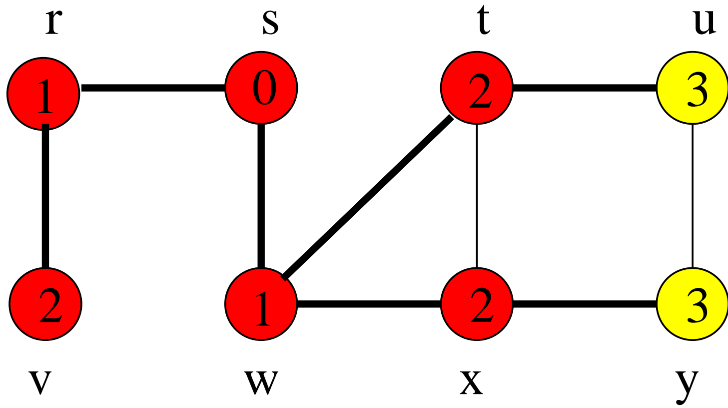
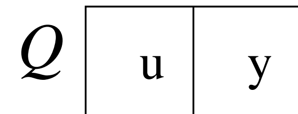
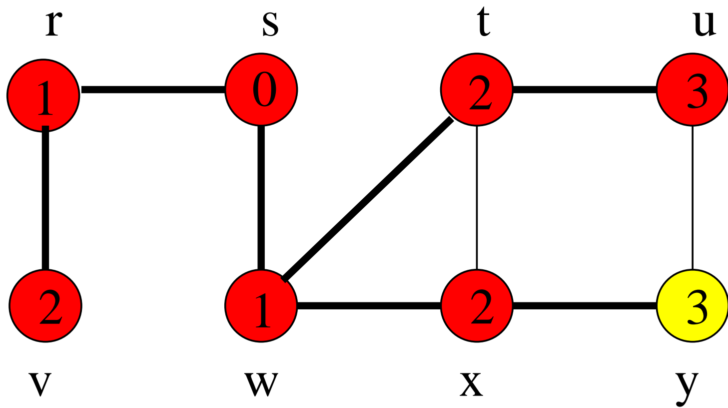
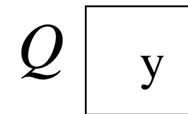
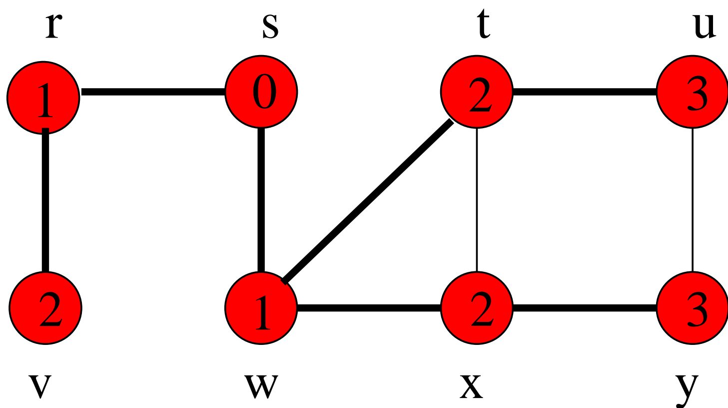
Q

t	x	v



Q

x	v	u


r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y

Q
v
u
y

r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y

Q
u
y

r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y

Q
y

r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y

Q

- Aikavaativuus:
  - Alustukseen (rivit 1–6) kuluu aikaa $\mathcal{O}(|V|)$.
  - Vain valkoinen solmu voi harmaantua, ja tummunut solmu ei valkaistu uudelleen. Siis jokainen solmu käy jonossa Q korkeintaan kerran.
  - Jokainen jono-operaatio voidaan toteuttaa toimimaan vakioajassa $\mathcal{O}(1)$. Yhteensä kaikkiin jono-operaatioihin kuluu siis aikaa $\mathcal{O}(|V|)$.
  - Kunkin solmun vieruslista käydään läpi ainoastaan yhden kerran: silloin kuin solmu poistetaan jonosta (ja mustuu).
  - Kaikkien vieruslistojen yhteispituus on $\mathcal{O}(|E|)$. Yhteensä niiden kaikkien läpikäyntiin kuluu siis aikaa $\mathcal{O}(|E|)$.

  Yhteensä siis

$$\mathcal{O}(|V| + |E|)$$

  eli *lineaarinen* suhteessa kalvojen 7.2.1 vieruslistaesityksen kokoon nähden.

- Tilavaativuus:
  - Algoritmi tarvitsee aputilaa jonolle Q (sekä vakiotilan apumuuttujilleen u ja v).
  - Jonon Q pituus voi pahimmillaan olla

$$|V| - 1:$$

    jos aloitussolmusta s on kaari kaikkiin muihin(kin) solmuihin.
  - Siis tilavaativuus on

$$\mathcal{O}(|V|)$$

    eli lineaarinen solmujen lukumäärään nähden.

7.3.2 Syvyyssuuntainen läpikäynti

- Myös verkon *syvyyssuuntainen* läpikäynti tutkii, mitkä solmut v voidaan saavuttaa aloitussolmusta s.

- Algoritmin löytämällä polulla

$$s \rightsquigarrow v$$

  ei nyt ole välitöntä intuitiivista tulkintaa

  (kuten ”lyhyys” oli kalvojen 7.3.1 leveyssuuntaisessa läpikäynnissä):

  Algoritmi vain

  – pyrkii syöteverkossa aina rohkeasti eteenpäin uusiin solmuihin mitä tahansa reittiä pitkin

  – raportoi nämä ”opportunistisesti” käyttämänsä reitit.

- Syvyyssuuntainen läpikäynti tehdäänkin useimmiten *osana* jotakin toista verkkoalgoritmia:
  - Sen valitsemista reiteistä saadaan informaatiota syöteverkon rakenteesta.
  - Saatua informaatiota voidaan käyttää varsinaisessa algoritmissa.

- *Edetään* aloitussolmusta s alkaen samaa polkua pitkin uusiin solmuihin niin kauan kuin mahdollista:
  - Vasta kun tullaan solmuun, josta ei enää pääse uusiin solmuihin, niin sitten *peruutetaan*
  - tutkitulla polulla edelliseen solmuun, josta lähtee vielä tutkimaton haara, ja aletaan seurata sitä.

  Tämä tehdään *rekursiolla*.

- Näin löydetään kaikki solmusta s saavutettavissa olevat solmut.

- Läpikäynnin edetessä solmuja väritetään samaan tapaan kuin leveyssuuntaisessa läpikäynnissä:

  Valkoinen: Vielä ei ole edetty solmuun v saakka.

  Harmaa: Solmu v on löytynyt ja tutkittavalla polulla.

  Musta: Kaikki tavat jatkaa solmusta v on nyt käyty läpi

  joten läpikäynti on jo peruuttanut pois solmusta v.

- Samalla algoritmi *aikaleimaa* jokaisen solmun v uusilla attribuuteilla:

  $d[v]$ = löytymis- eli harmaantumishetki
  $f[v]$ = peruutus- eli mustumishetki.

  Nämä attribuutit ovat sitä informaatiota, jota syvyyssuuntainen läpikäynti kerää syöteverkosta.

- Seuraavassa algoritmi joka selvittää aloitussolmusta s saavutettavissa olevat solmut.

- Algoritmi käyttää aikaleimaamiseen globaalia muuttujaa

$$\begin{aligned}\text{time} &= \text{algoritmin oma ''kellonaika''} \\ &= \text{sen tekemien värinvaihtojen määrä.}\end{aligned}$$

DFS(G, s)
1 **for** jokaiselle solmulle $u \in V$ **do**
2   color[u] $\leftarrow$ white
3 time $\leftarrow$ 0
4 DFSvisit(s)

DFSvisit(u)
1 color[u] $\leftarrow$ gray
2 time $\leftarrow$ time+1
3 $d[u]$ $\leftarrow$ time
4 **for** jokaiselle solmulle $v \in$ Adj[u] **do**
5   **if** color[v] = white **then** DFSvisit(v)
6 color[u] $\leftarrow$ black
7 time $\leftarrow$ time+1
8 $f[u]$ $\leftarrow$ time

- Seuraavana kalvona on esimerkki algoritmin toiminnasta.

- Algoritmi siis värjää

  1. ensin harmaaksi (kuvassa keltainen)

  2. sitten mustaksi (kuvassa punainen)

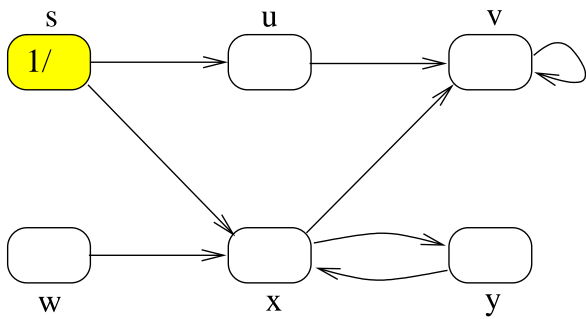
  kaikki aloitussolmusta s saavutettavissa olevat solmut.

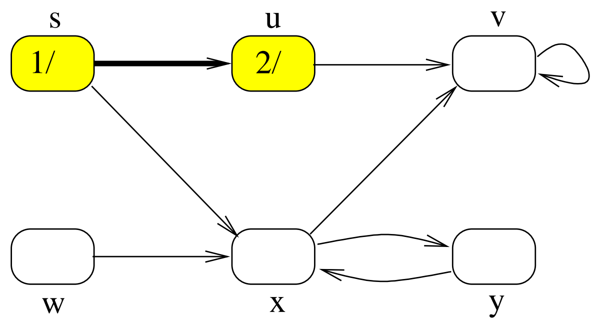
- Kuvassa on **paksunnettu** ne kaaret, joita pitkin läpikäynti on edennyt

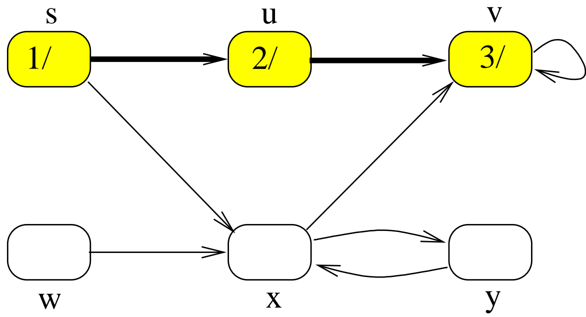
  – eli algoritmin käyttämien polkujen muodostama *puu*

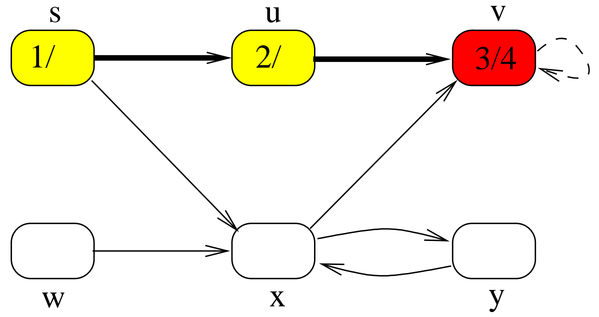
  – joka olisi voitu tallettaa attribuutteihin $p[v]$ kuten leveyssuuntaisessa läpikäynnissä.

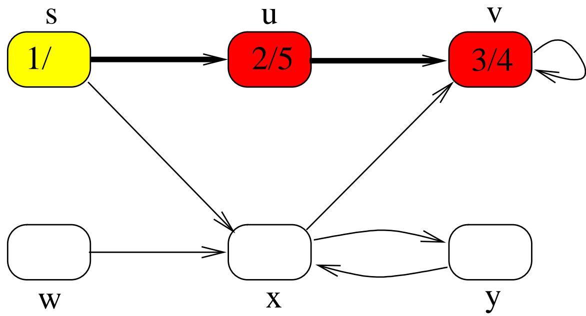
- Jokaiseen solmuun u on merkitty attribuuttiarvot $d[u]/f[u]$ sitä mukaa kun ne saadaan asetettua.

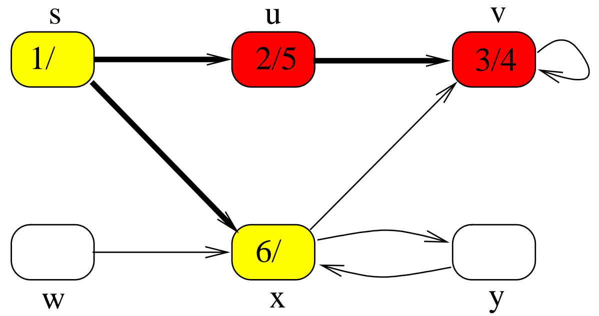

s
u
v
w
x
y
1/

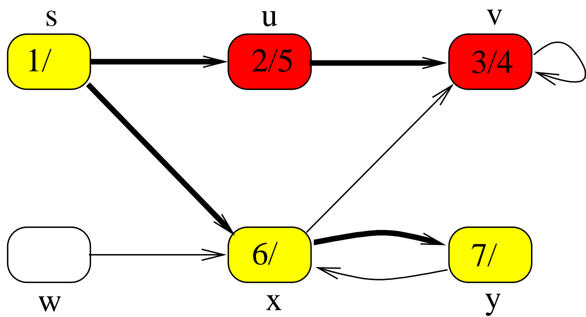

s
u
v
w
x
y
1/
2/

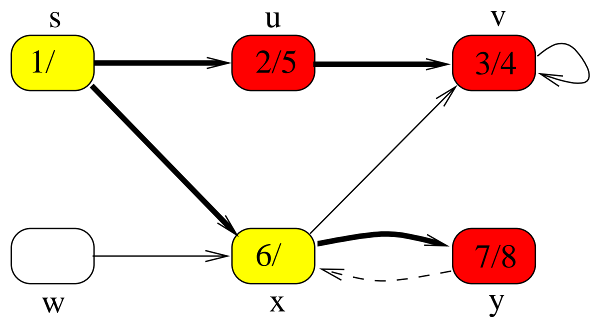

s
u
v
w
x
y
1/
2/
3/

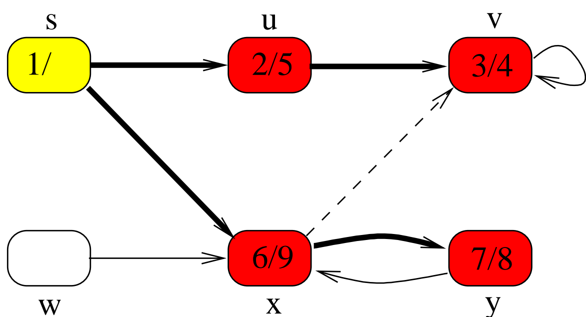

s
u
v
w
x
y
1/
2/
3/4

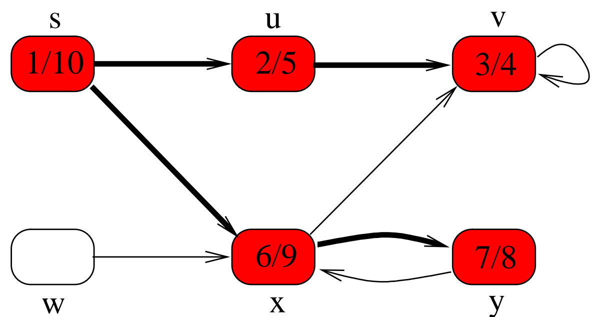

s
u
v
w
x
y
1/
2/5
3/4


s
u
v
w
x
y
1/
2/5
3/4
6/


s
u
v
w
x
y
1/
2/5
3/4
6/
7/


s
u
v
w
x
y
1/
2/5
3/4
6/
7/8


s
u
v
w
x
y
1/
2/5
3/4
6/9
7/8


s
u
v
w
x
y
1/10
2/5
3/4
6/9
7/8

- Suorituksen jälkeen solmulle lasketut attribuutit siis ilmaisevat, milloin sen käsittely alkoi ja päättyi.

- Syvyyssuuntainen läpikäynti siis löysi esimerkkimme solmut seuraavassa järjestyksessä:

$$s, u, v, x, y.$$

- Solmua w ei saavuteta aloitussolmusta, w jää valkeaksi.

- Jos halutaan käydä läpi kaikki verkon solmut

  - ja verkossa on tällaisia solmuja w, jotka eivät ole saavutettavissa solmusta s,

  - niin valitaan jokin niistä, ja käynnistetään siitä alkaen uusi etsintä,

  - joka aloitetaan vanhoilla väreillä,

    jotta se ei kävisi uudelleen läpi jo aikaisemmissa etsinnöissä läpikäytyjä solmuja.

    Olihan esimerkissämmekin kaari $w \to x$.

- Koko verkolle tehtävä syvyyssuuntainen läpikäynti tapahtuu siis seuraavasti:

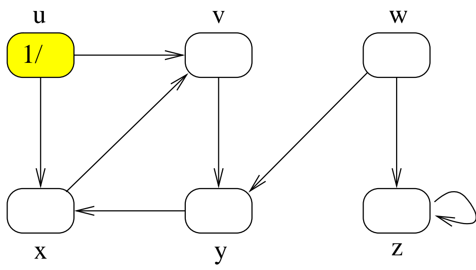
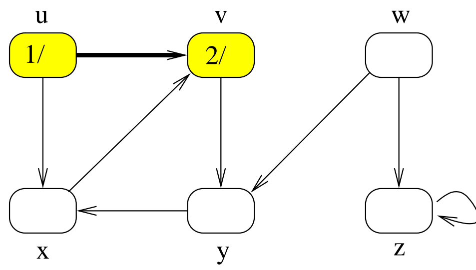
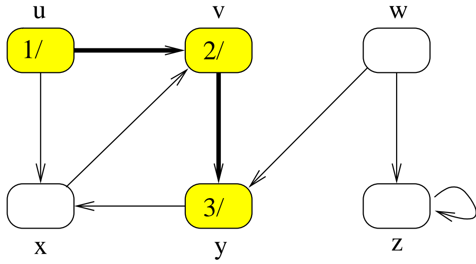
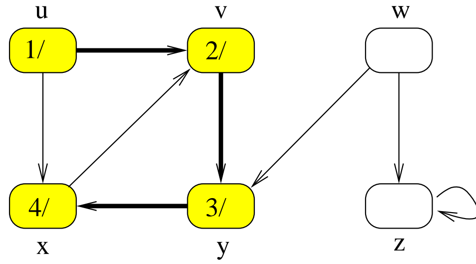
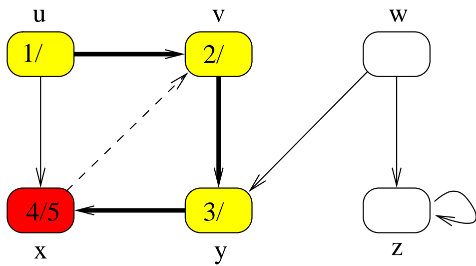
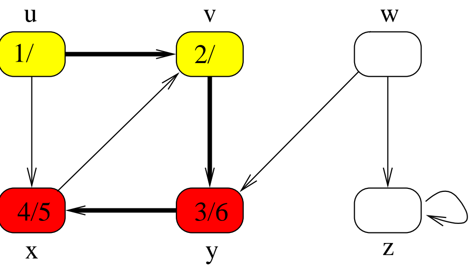
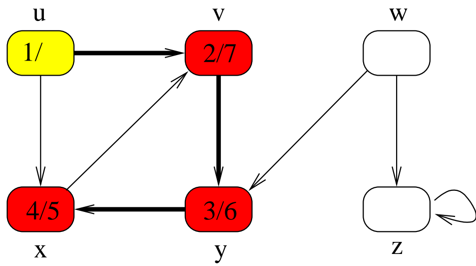
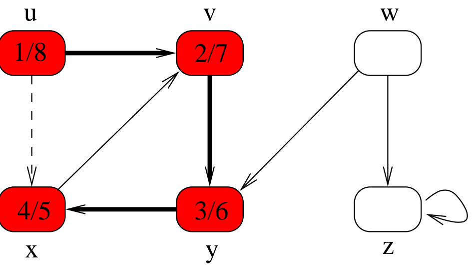
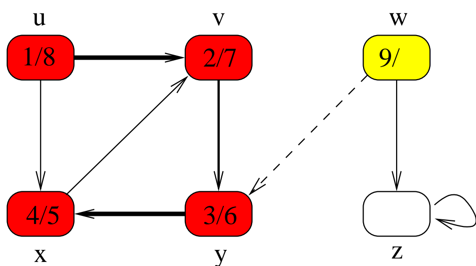
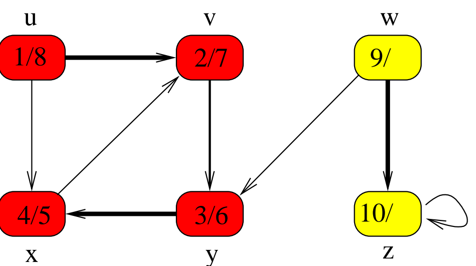
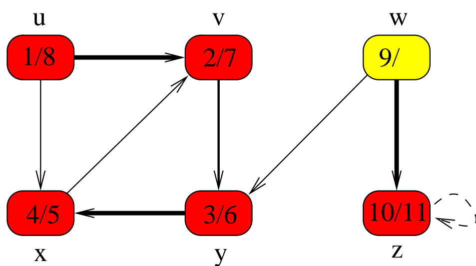
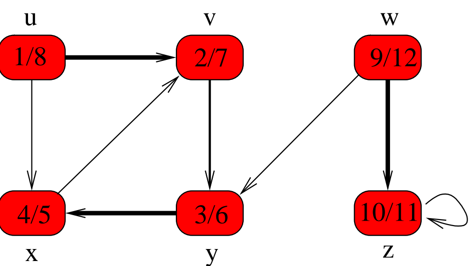
  DFSall(G)
  1 **for** jokaiselle solmulle $u \in V$ **do**
  2 color[u] $\leftarrow$ white
  3 time $\leftarrow$ 0
  4 **for** jokaiselle solmulle $u \in V$ **do**
  5 **if** color[v] = white **then** DFSvisit(u)

- Seuraavana kalvona on esimerkki tämän algoritmin suorituksesta.

- Pääohjelmassa tehdään rekursiokutsut

  1. DFSvisit(u), jonka aikana mustuvat myös solmut v, x ja y

  2. DFSvisit(w), jonka aikana mustuu myös solmu z.


u
v
w
1/
x
y
z

u
v
w
1/
2/
x
y
z

u
v
w
1/
2/
3/
x
y
z

u
v
w
1/
2/
4/
3/
x
y
z

u
v
w
1/
2/
4/5
3/
x
y
z

u
v
w
1/
2/
4/5
3/6
x
y
z

u
v
w
1/
2/7
4/5
3/6
x
y
z

u
v
w
1/8
2/7
4/5
3/6
x
y
z

u
v
w
1/8
2/7
9/
4/5
3/6
x
y
z

u
v
w
1/8
2/7
9/
4/5
3/6
10/
x
y
z

u
v
w
1/8
2/7
9/
4/5
3/6
10/11
x
y
z

u
v
w
1/8
2/7
9/12
4/5
3/6
10/11
x
y
z

- Aikavaativuus:

  - Alustustoimet, eli solmujen värjääminen valkoiseksi, vie aikaa $\mathcal{O}(|V|)$.

  - Operaatiota DFSvisit(u) kutsutaan (korkeintaan) kerran jokaiselle solmulle u:

    1. u on kutsuhetkellä valkoinen

    2. u harmaantuu heti kutsun aluksi

    3. u jää lopulta mustaksi.

    Yhteensä siis kutsuja tehdään $\mathcal{O}(|V|)$ kappaletta.

  - Kutsun DFSvisit(u) aikana käydään läpi solmun u vieruslista Adj[u].

    Siis rivien 4–5 silmukka suoritetaan koko läpikäynnin aikana yhteensä $\leq |E|$ kertaa.

  Yhteensä siis jälleen

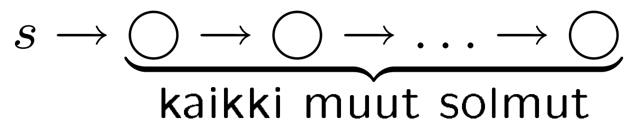
$$\mathcal{O}(|V| + |E|)$$

  eli *lineaarinen* suhteessa kalvojen 7.2.1 vieruslistaesityksen kokoon nähden.

- Tilavaativuus:

  – Algoritmin aputila koostuu sisäkkäisten rekursiokutsujen vaatimasta pinosta.

  – Pahimmassa tapauksessa verkossa on polku



  ja läpikäynti etenee juuri sitä pitkin peruuttamatta.

  – Silloin sisäkkäisiä rekursiokutsuja tehdään

$$\mathcal{O}(|V|)$$

  kappaletta.

- Myös syvyyssuuntaisen läpikäynnin algoritmi toimii sellaisenaan niin suunnatuilla kuin suuntaamattomillakin verkoilla.

- Lopuksi esimerkkinä d- ja f-attribuuttien keräämästä informaatiosta syöteverkon *kaarten luokittelu*.

  Varjostetut kaaret ovat **puukaaria**:

  ne, joita pitkin läpikäynnin rekursio etenee.

  B-kaari $x \to z$ kulkee *taaksepäin* puussa:

$$d[z] \leq d[x] < f[x] \leq f[z].$$

  F-kaari $s \to w$ kulkee *eteenpäin* puussa:

$$d[s] < d[w] < f[w] < f[s].$$

  C-kaari kulkee *sivuttain*

  saman puun sisällä kuten $w \to x$

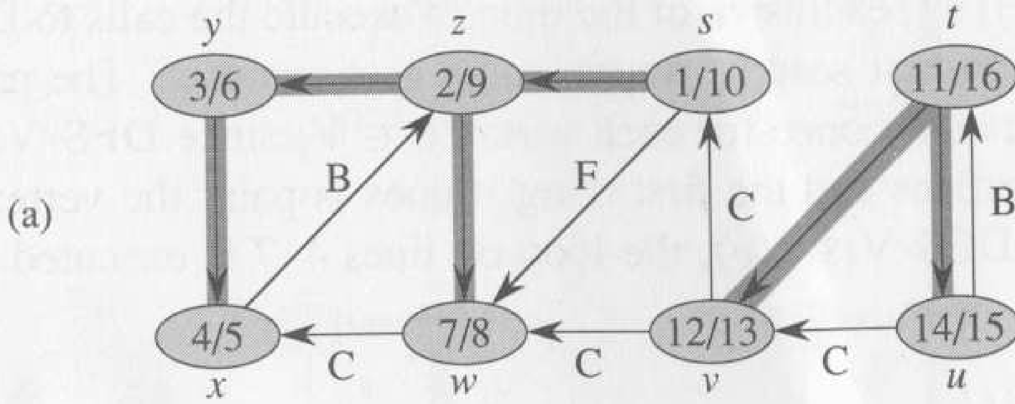
  puusta toiseen kuten $v \to w$.

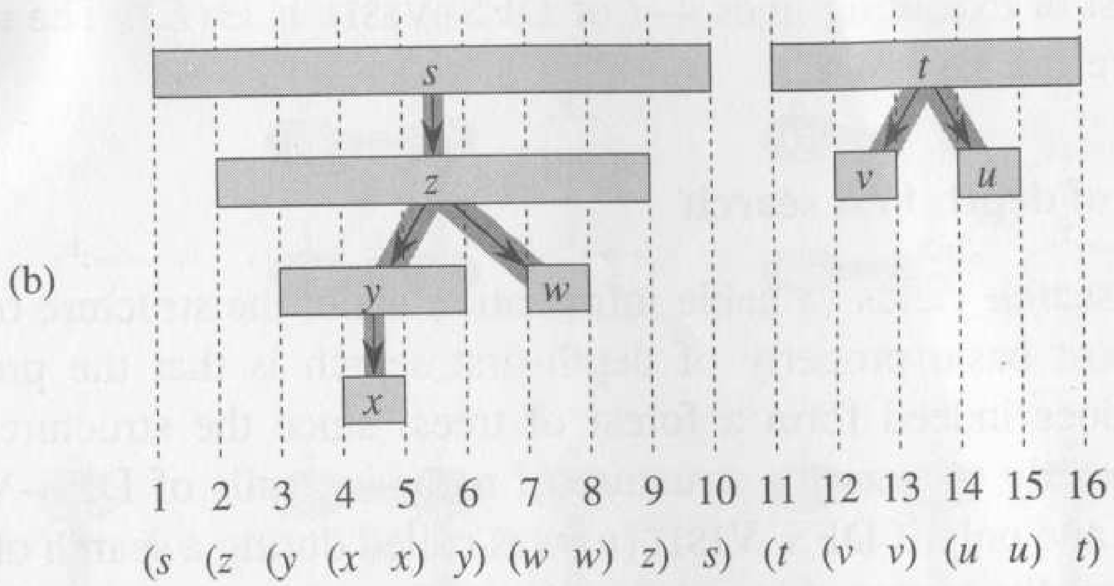
  Molemmissa tapauksissa

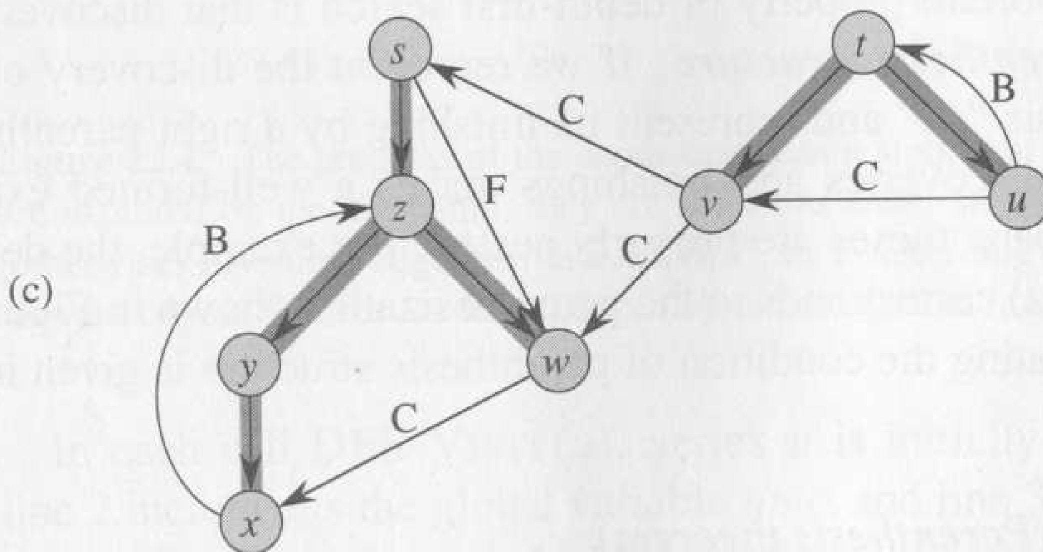
$$f[\text{kohdesolmu}] < d[\text{lähtösolmu}].$$

  Ne voidaan vielä erotella puun juuren perusteella.

  (Kuva 22.5 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
(a)
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u


s
t
z
v
u
(b)
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)


s
t
C
B
C
z
F
v
u
B
C
(c)
y
w
C
x

7.3.3 Verkon syklittömyyden tarkastus

- Joskus voi olla tarvetta testata
  - onko syötteenä annettu suunnattu verkko $G = (V, E)$ DAG
  - vai onko siinä jokin sykli.

Lause 7.1. *Suunnatussa verkossa on sykli täsmälleen silloin kun sen mielivaltainen syvyyssuuntainen läpikäynti kohtaa harmaan solmun.*

Todistus:

$\Leftarrow$: Syvyyssuuntaisessa läpikäynnissä solmu on harmaa täsmälleen silloin kun se on rekursiopinossa.

Jos siis rekursion aikana kohdataan jokin jo harmaa solmu v, niin verkossa on sykli

$$v \to \underbrace{\bigcirc \to \bigcirc \to \ldots \to \bigcirc}_{\text{pino solmun } v \text{ päällä}} \to v$$

jota pitkin solmu v kohdattiin uudelleen.

$\Rightarrow$: Tarkastellaan sitä syvyyssuuntaisen läpikäynnin hetkeä, jolloin kohdataan (ja väritetään harmaaksi) ensimmäisen kerran jokin sellainen solmu v, joka kuuluu johonkin sykliin

$$v \rightarrow \bigcirc \rightarrow \bigcirc \rightarrow \ldots \rightarrow u \rightarrow v.$$

Sillä hetkellä kaikki muutkin tämän polun solmut ovat vielä valkoisia.

Kun rekursio jatkaa solmusta v eteenpäin, niin se värjää kaikki tämän valkoisen polun solmut (”valkopolkulause”).

Perustelu: Mikä olisi niistä ensimmäinen, joka voisi muka jäädä valkoiseksi?

Huomaa: Rekursion ei kuitenkaan ole pakko kulkea juuri tätä polkua pitkin!

Erityisesti siis solmu u harmaantuu.

Siis kun solmua u käsitellään, niin sillä on harmaa vierussolmu, nimittäin v. $\square$

- Toisin sanoen:
  - Verkossa on sykli jos ja vain jos sen kalvojen 7.3.2 syvyyssuuntaisessa läpikäynnissä löytyy B-kaari.
  - Ei ole väliä, missä järjestyksessä edetään, kunhan edetään syvyyssuunnassa.

- Voimme siis käyttää syvyyssuuntaisen läpikäynnin algoritmia pienellä muutoksella syklittömyyden testaamiseen:
  - Onko nykyisen solmun vieruslistalla harmaa solmu?
  - Solmun löytymisajan ja käsittelyn päättymisajan tallentavat attribuutit d ja f ovat nyt tarpeettomia.

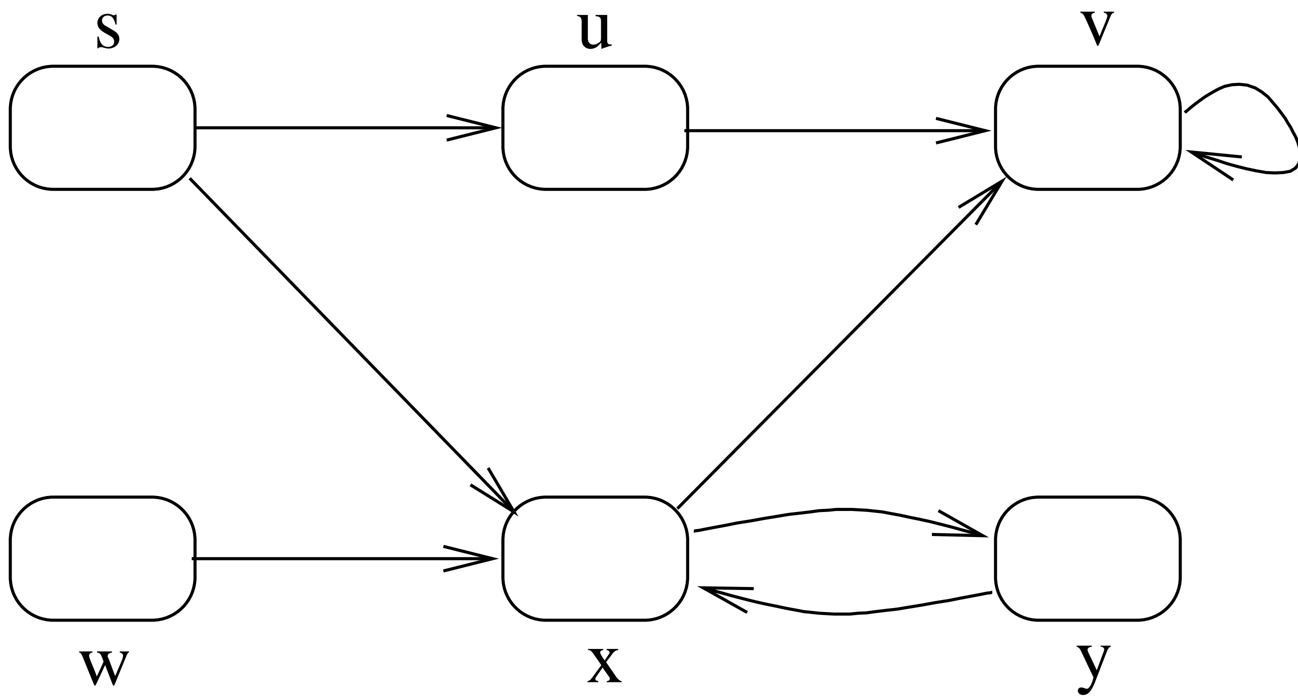
- *Huomaa:* **musta**n solmun v löytyminen solmun u vieruslistasta ei tarkoita syklin olemassaoloa:

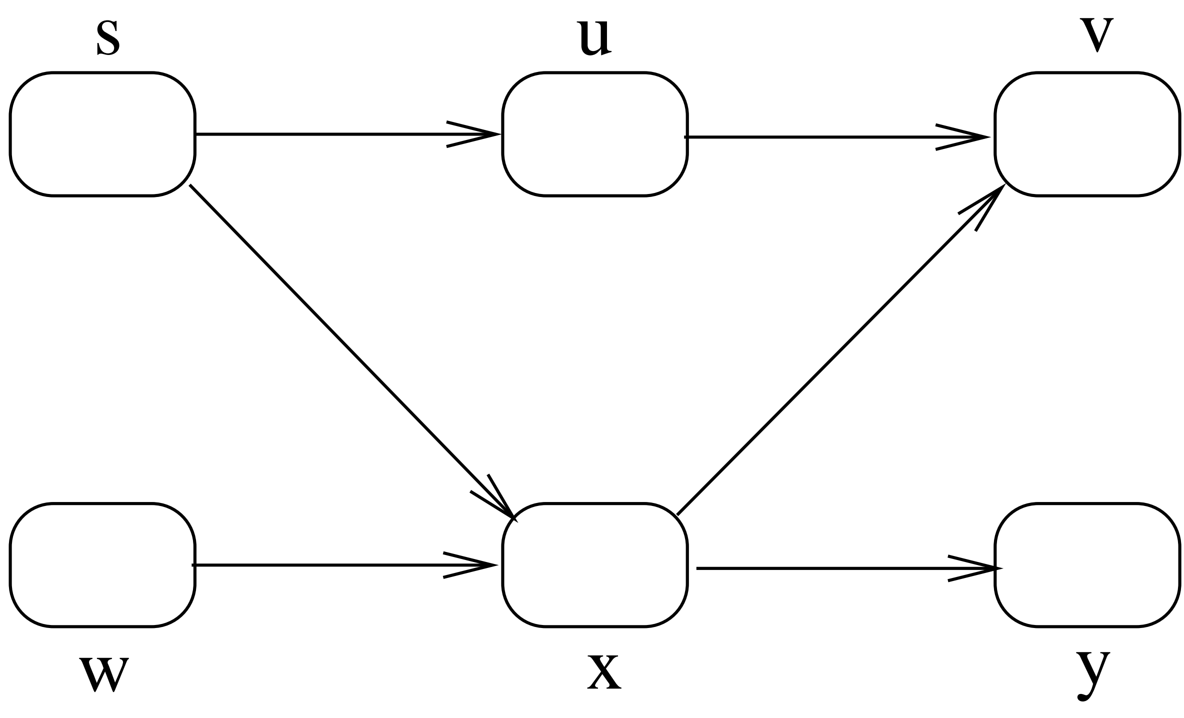
  polkua $v \leadsto u$ ei voi olla olemassa, sillä siinä tapauksessa v ei olisi musta!

```
DFScyclic(G)
1   for jokaiselle solmulle u ∈ V do
2       color[u] ← white
3   for jokaiselle solmulle u ∈ V do
4       if color[u] = white then
5           if DFScvisit(u) then return true
6   return false
```

```
DFScvisit(u)
1   color[u] ← gray
2   for jokaiselle solmulle v ∈ Adj[u] do
3       if color[v] = gray then return true
4       if color[v] = white then
5           if DFScvisit(v) then return true
6   color[u] ← black
7   return false
```

- Esimerkiksi miten syklien etsintä toimisi seuraavissa tapauksissa?


s
u
v
w
x
y


s
u
v
w
x
y

7.3.4 Topologinen järjestäminen

- Syklittömien suunnattujen verkkojen avulla voimme kuvata vaikkapa tapahtumien välisiä riippuvuuksia.

- Esimerkiksi seuraava kaavio sisältää onnistuneen pukeutumiseen kannalta oleelliset riippuvuudet:

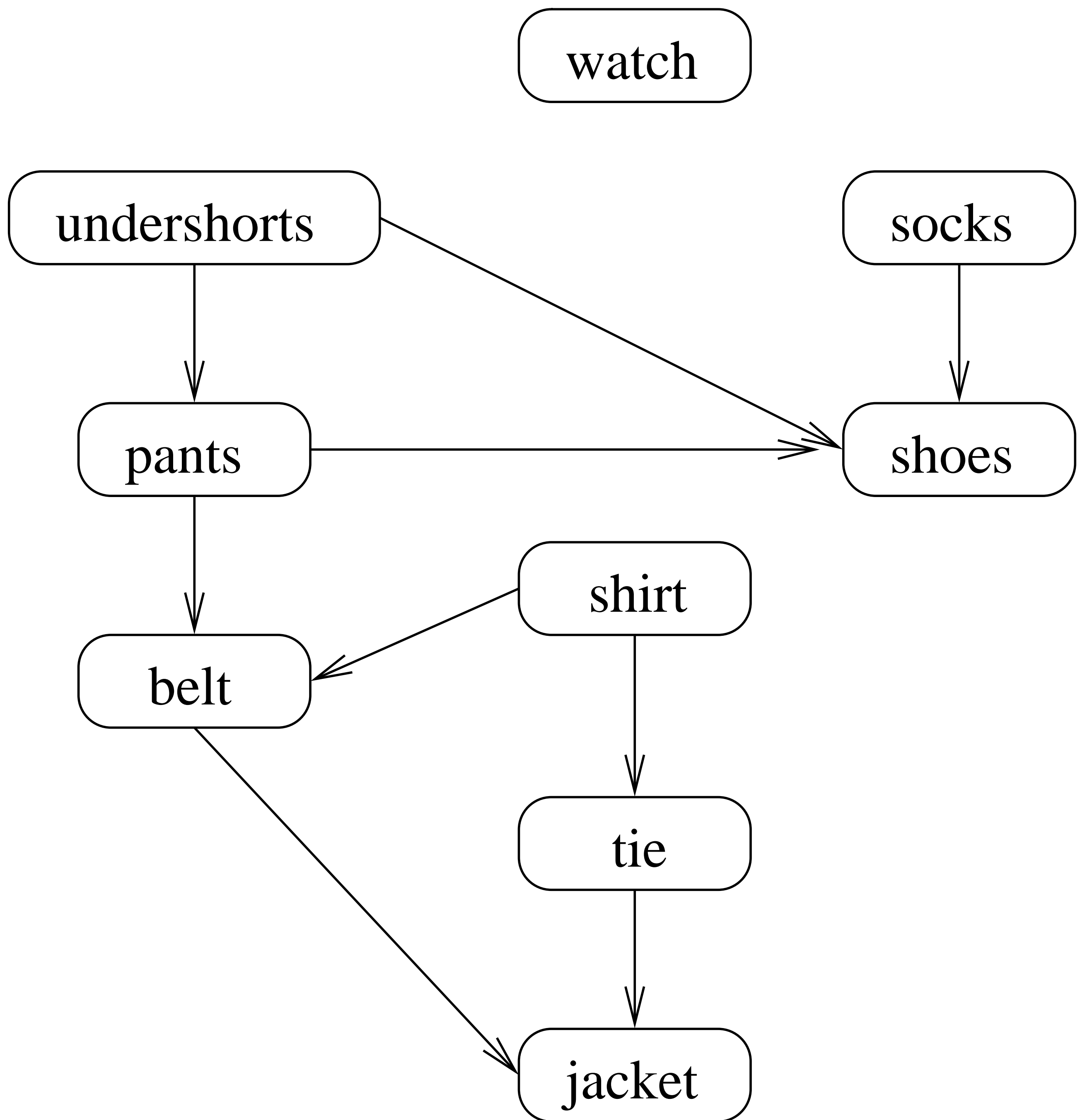
  - Sukat on laitettava ennen kenkiä, solmio ja vyö ennen takkia, jne.

  - Voisimmeko järjestää asiat sellaiseen lineaariseen järjestykseen, että voisimme pukea vaatekappaleen kerrallaan siten että mikään riippuvuuksista ei rikkoudu?

- Siis siten, että jos riippuvuusverkossa on kaari

  $$u \to v$$

  niin solmu u tulee järjestyksessä ennen solmua v.


watch
undershorts
socks
pants
shoes
shirt
belt
tie
jacket

- Tällaista järjestystä kutsutaan *topologiseksi järjestykseksi*.

- Asia hoituu helposti käyttäen apuna syvyyssuuntaista läpikäyntiä:

  1. Aluksi tuloslista L on tyhjä.

  2. Kutsutaan sellaista DFSall(G)

     – joka vie käsiteltävän solmun u listan L alkuun

     – sillä hetkellä kun u mustuu.

  3. Lopuksi lista L koostuu syöteverkon G solmuista topologisessa järjestyksessä.

     – L listaa solmut käsittelyn *päättymisajan f suhteen käänteisessä* järjestyksessä

     – eli solmu p on ennen solmua q listalla L täsmälleen silloin kun

     $$f[p] > f[q]$$

     – joten kaari $x \to y$ pakottaa solmun x solmun y eteen.

- Näin listassa L

  viimeiseksi tulee jokin sellainen solmu v_n, josta ei ole kaarta mihinkään muuhun solmuun

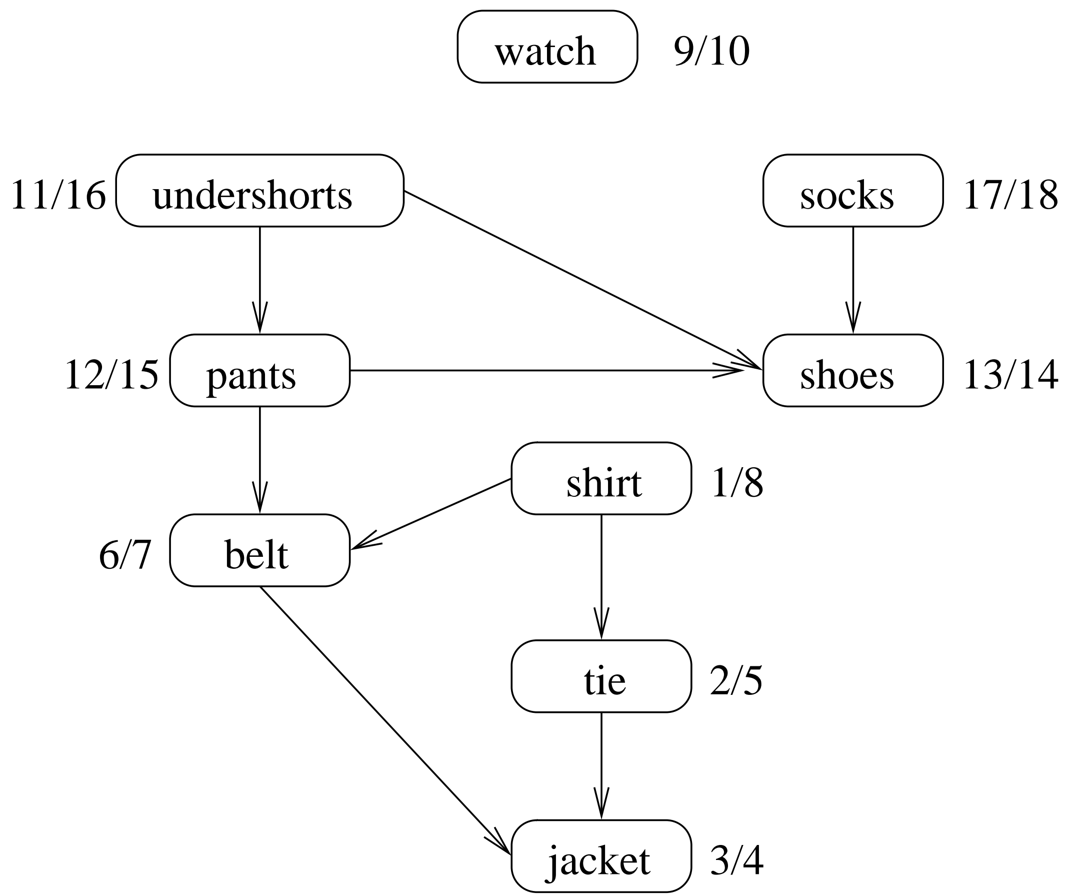
  toiseksi viimeiseksi solmuksi v_{n-1}, josta on kaari korkeintaan solmuun v_n

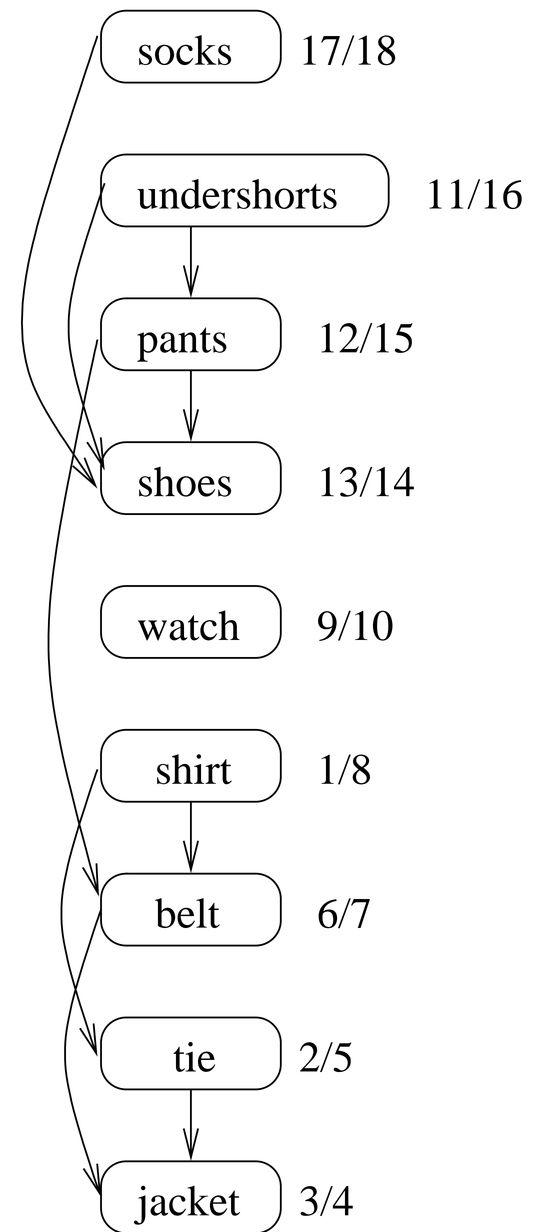
  yleisesti solmusta v_i voi olla kaaria korkeintaan sen jälkeisiin solmuihin

$$v_{i+1}, v_{i+2}, v_{i+3}, \ldots, v_n.$$

- Seuraavassa kuvassa on esimerkkimme eräässä topologisessa järjestyksessä.

- Topologinen järjestys ei ole yksikäsitteinen
  - vaan riippuu solmujen syvyyssuuntaisen läpikäynnin järjestyksestä.
  - Esimerkiksi kello voitaisiin pukea missä välissä tahansa.


watch
9/10
11/16
undershorts
socks
17/18
12/15
pants
shoes
13/14
shirt
1/8
6/7
belt
tie
2/5
jacket
3/4


socks
17/18
undershorts
11/16
pants
12/15
shoes
13/14
watch
9/10
shirt
1/8
belt
6/7
tie
2/5
jacket
3/4

7.3.5 Kriittiset työvaiheet

- (Ohjelmisto)projektissa on tiettyjä *määrähetkiä:*
  - dokumenttien ja komponenttien määräpäiviä
  - (laadunvarmistus)katselmuksia
  - asiakasdemoja
  - ...
- Määrähetkillä on *osittainen järjestys:*
  - Käyttöliittymä(n ensimmäinen versio) on saatava valmiiksi ennen asiakasdemoa.
  - Käyttöliittymää ei voida aloittaa ennen kuin tietokantaliittymä on valmis.
  - ...

- Eri vaiheilla on *kestoja:*
  - Käyttöliittymän aloittamisesta sen valmistumiseen kuluu (arviolta) 4 viikkoa.
  - ...
- Verrattuna edellisten kalvojen 7.3.4 pukeutumisesimerkkiin nyt
  - lisätään eri vaiheille kestot
  - sallitaan niiden tekeminen rinnakkain.
- Tässä mallinnuksessa tosin unohdetaan käytännön rajoitteista esimerkiksi

  resurssit kuten

  > ”X on ainoa työtekijämme, joka hallitsee sekä käyttöliittymä- että verkkoyhteysohjelmoinnin.”

  disjunktiiviset kuten

  > ”Siispä käyttöliittymä on ohjelmoitava *joko* ennen *tai* jälkeen verkkoyhteyden, mutta ei yhtä aikaa.”

- Halutaan tietää (tarjouskilpailua varten) kauanko projektiin *vähintään* kuluu.

- Mallinnetaan suunnattuna verkkona:

  jokainen vaihe kaarena

  $$\text{vaiheen alku} \xrightarrow{\text{vaiheen kesto}} \text{vaiheen loppu}$$

  eri vaiheiden järjestys kaarena

  $$\text{edellisen loppu} \xrightarrow{0} \text{seuraavan alku.}$$

- Halutaan laskea tämän verkon *painavin polku*.

- Määritellään ''solmusta u alkavan polun suurin mahdollinen paino'' seuraavasti:

  $$h[u] = \max\left(0, \max_{u \xrightarrow{c} v \in \mathsf{Adj}[u]} (c + h[v])\right). \quad (8)$$

  - 0 jos solmusta u ei jatketa eteenpäin.

  - Eri jatkomahdollisuudet ovat

    $$u \xrightarrow{c} v \underbrace{\leadsto \ldots}_{h[v]}.$$

- Laskusääntö (8) on mieletön, jos
  - jonkin luvun $h[u]$ laskemiseksi tarvittaisiin sen luvun arvoa itseään
  - verkossa olisi sykli

$$u \rightsquigarrow u$$

  - koko projekti olisi mahdoton.

- Solmun luku $h[u]$ voidaan laskea vasta sen jälkeen, kun on ensin laskettu kaikkien seuraajasolmujen luvut $h[v]$.

- Toisin sanoen: solmut u täytyy käydä läpi *käänteisessä topologisessa* järjestyksessä.

- Kalvojen 7.3.4 algoritmissa solmun u luku $h[u]$ voidaan laskea sillä hetkellä, kun u vietäisiin järjestyksen L alkuun.

- Projektin kokonaiskesto on

$$t = \max_{u \in V} h[u].$$

- Käytännössä halutaan tietää myös mitkä projektin poluista ovat *kriittisiä:*

  sellaisia, joiden myöhästyminen venyttää koko projektin kestoa.

  – Kriittisiä ovat ainakin ne (vaiheiden alku)solmut u, joiden

  $$h[u] = t. \tag{9}$$

  – Jos solmu u on kriittinen, niin sen seuraajasolmuista v ovat kriittisiä ne

  * joista luku $h[u]$ olisi voitu saada säännöllä (8)

  * joille pätee ehto

  $$d[u] = w(u, v) + d[v]. \tag{10}$$

  – Kriittiset polut voidaan siis tulostaa kalvojen 7.3.2 syvyyssuuntaisella läpikäynnillä

  * lähtösolmuina ne u joille ehto (9)

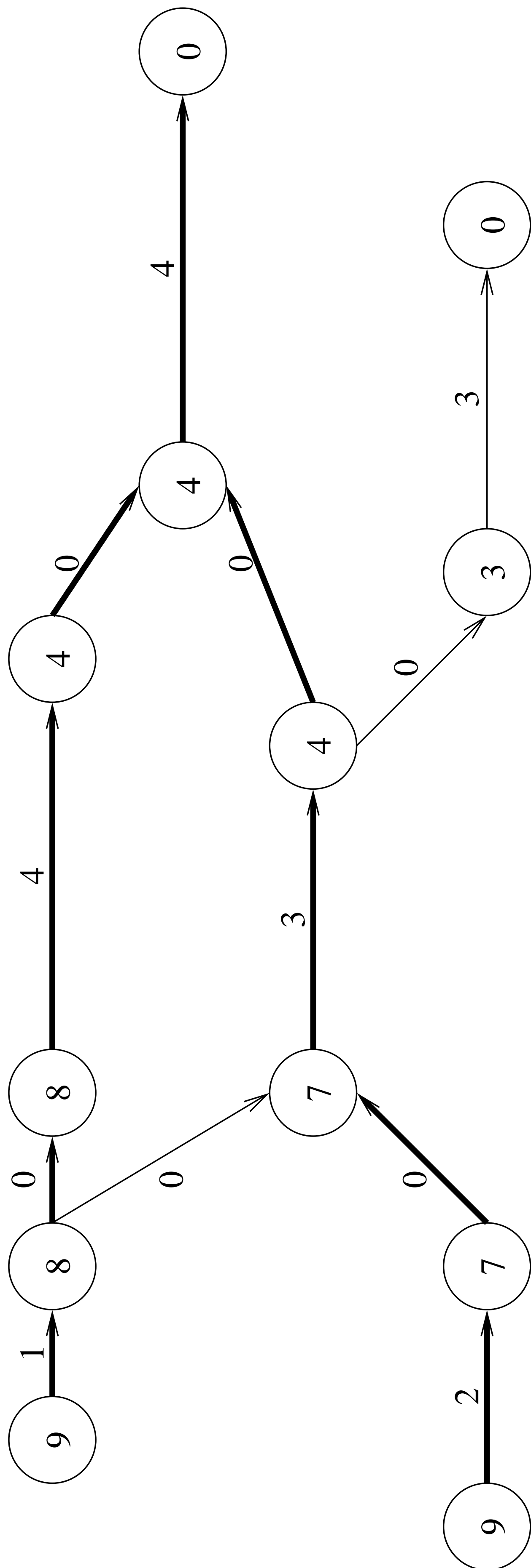
  * kaarina ne $u \xrightarrow{w(u,v)} v$ joille ehto (10)

  pätee.

- Seuraavassa kuvassa on esimerkki pienen projektin verkosta.

- Jokaiseen solmuun on laskettu sen luku h.

- Kriittiset polut on **tummennettu**.



7.3.6 Vahvasti yhtenäiset komponentit

- Suunnatun verkon G jako sen *vahvasti yhtenäisiin komponentteihin* (strongly connected components, SCC) jakaa sen solmut V eri komponentteihin seuraavalla periaatteella:

  Solmut u ja v kuuluvat samaan komponenttiin C

  jos ja vain jos

  verkossa G pääsee solmusta u solmuun v *ja takaisin*.

- Seuraavassa kuvassa (a) on tummennettu esimerkkiverkon G vahvasti yhtenäiset komponentit.

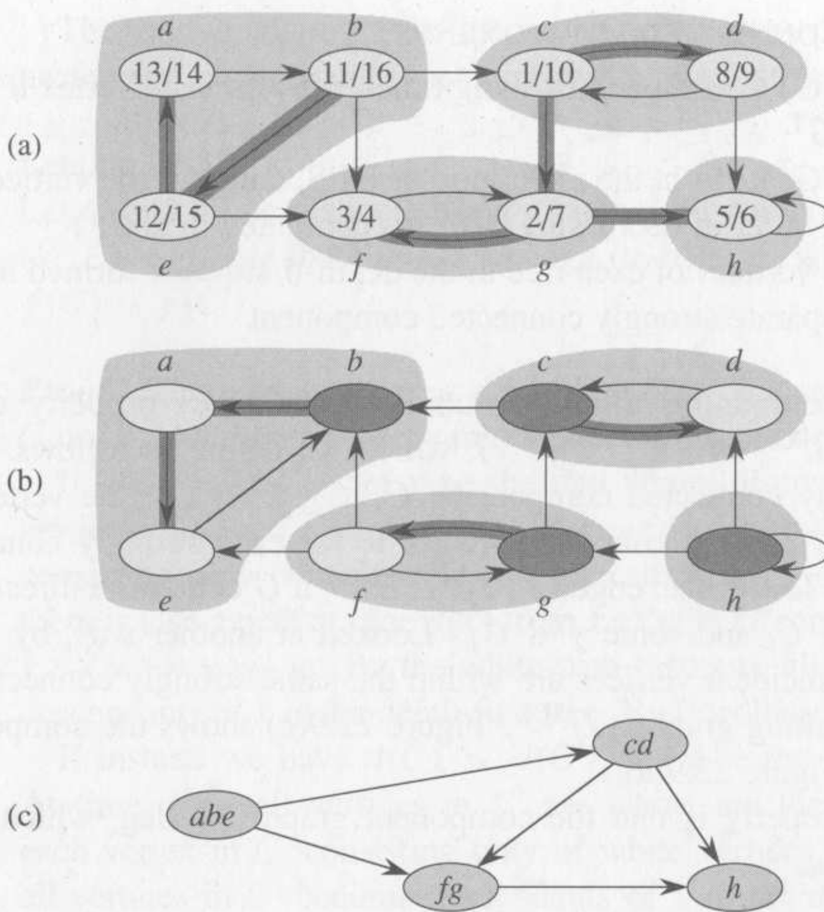


Figure 22.9 **(a)** A directed graph G. The strongly connected components of G are shown as shaded regions. Each vertex is labeled with its discovery and finishing times. Tree edges are shaded. **(b)** The graph G^{T}, the transpose of G. The depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS is shown, with tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^{T}. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

(Kuva 22.9 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

- Verkon G vahvasti yhtenäiset komponentit muodostavat suunnatun verkon G^{SCC} seuraavasti:

  Solmuina ovat verkon G eri komponentit, eli solmujen joukot.

  Kaari $C \to C'$ komponentista toiseen tarkoittaa, että

  - komponentissa C on solmu u, ja
  - komponentissa C' on solmu v, joilla
  - alkuperäisessä verkossa G oli kaari $u \to v$.

  Kaaria luodaan vain yksi, vaikka solmuja u ja v olisikin useita.

- Edellisessä kuvassa (c) on esitetty esimerkkiverkon G "kutistus" komponenttiverkokseen G^{SCC}.

- Komponenttiverkko G^{SCC} on syklitön:

  Jos olisi kehä

  $$C_1 \to C_2 \to C_3 \to \ldots \to C_k \to C_1$$

  niin kaikki sen komponentit $C_1, C_2, C_3, \ldots, C_k$ yhtyisivätkin yhdeksi yhteiseksi komponentiksi.

- Komponenttiverkon G^{SCC} muodostaminen on usein tärkeä esivaihe muissa verkko-ongelmissa.

- Seuraavalla kalvolla on algoritmi, joka listaa
  - syöteverkon G eri komponenttien C sisällöt

    eli kuhunkin komponenttiin C kuuluvat syöteverkon G solmut
  - nämä komponentit C kalvojen 7.3.4 topologisessa järjestyksessä

    komponenttiverkon G^{SCC} suhteen.

1. Järjestä syöteverkon G solmut topologisesti (kalvoilta 7.3.4)

$$u_1, u_2, u_3, \ldots, u_{|V|} \tag{11}$$

mutta älä välitäkään nyt B-kaarista (kalvoilta 7.3.2).

2. Luo syöteverkon G *transpoosi* G^T*:* verkko, jonka

   solmut ovat samat, mutta

   kaarten suunta onkin käännetty päinvastaiseksi.

   Edellinen kuva (b) esittää esimerkkiverkon transpoosin.

3. Käy läpi transpoosi G^T syvyyssuuntaisesti (kalvoilta 7.3.2) aloittaen rekursiot järjestyksessä (11).

   - Kukin syvyyssuntainen puu käsittää täsmälleen yhden komponentin.

   - Samalla nämä komponentit tulevat käsiteltyä halutussa järjestyksessä.

- Askel 1 järjestää syöteverkon G solmut järjestykseen (11), jossa

  B-kaaret (kalvoilta 7.3.2) vievät *taakse*päin

  muut kaaret *eteen*päin.

- Seuraavassa kuvassa on edellisen esimerkkikuvan verkko

  - solmut näin järjestettyinä vasemmalta oikealle

  - B-kaaret **tummennettu**ina.

- Transpoosiin G^T siirtyminen (askel 2) tarkoittaa, että näitä kaaria kuljetaankin *takaperin* askeleessa 3.

- Kun askeleessa 3 alkaa syvyyssuuntaisen puun luova rekursio järjestyksen (11) solmusta u_i, niin se

  hyppää eteenpäin järjestyksessä jotakin B-kaarta

  $$u_i \leftarrow u_j \tag{12}$$

  pitkin, missä siis $j \geq i$

  palaa takaisin kohden juurisolmuaan u_i muita kaaria

  $$u_i \rightarrow \bigcirc \rightarrow \cdots \rightarrow \bigcirc \rightarrow \bigcirc \rightarrow \bigcirc \rightarrow u_j \tag{13}$$

  pitkin.

- Jokin paluureitti (13) on aina olemassa, koska (12) on B-kaari.

- Huomaa: B-kaari (12) ja paluureitti (13) muodostavat syklin!

- Paluureittiä (13) ei kuitenkaan voi palata ”liian pitkälle takaisin” ohi juurisolmusta u_i, koska
  - sitä aikaisemmat solmut
  $$u_1, u_2, u_3, \ldots, u_{i-1} \tag{14}$$
  ovat jo mustia
  - ne on jo käsitelty jossakin aikaisemmassa rekursiossa.

- Siis tämä rekursio käy läpi täsmälleen ne solmut u_j, jotka
  - muodostavat syklejä joihin kuuluu u_i
  - kuuluvat samaan komponenttiin kuin u_i.

- Komponenttien listautuminen topologisesti seuraa mustuudesta (14).

1. Edellisessä kuvassa askel 3 alkaa ensimmäisellä solmulla b.

2. Siinä ainoa vaihtoehto on B-kaari $b \leftarrow a$.

3. Solmusta a ainoa vaihtoehto on paluu $a \leftarrow e \leftarrow b$.

4. Näin saadaan listattua aiemman esimerkkikuvan (b) ensimmäinen komponentti abe.

5. Askel 3 jatkuu solmusta c, koska solmut b, a ja e ovat nyt mustuneet

   (kuten kuvaan on merkitty).

6. Siis B-kaarta $c \leftarrow b$ ei enää voi seurata, koska b on jo musta.

7. Ainoaksi vaihtoehdoksi jää siis B-kaari $c \leftarrow d$, ja saadaan seuraava komponentti cd

8. Samoin loput komponentit fg ja h.

Aikaa tarvitaan lineaarisesti:

$$\mathcal{O}(|V| + |E|).$$

(kalvojen 7.2.1 vieruslistaesityksessä).

- Askeleet 1 ja 3 ovat syvyyssuuntaisia läpikäyntejä.
- Myös askel 2 voidaan tehdä lineaarisesti.

Aputilaa tarvitaan myöskin lineaarisesti:

- syvyyssuuntaisten läpikäyntien rekursiopinolle
- askeleen 1 tuottamalle järjestykselle (11)
- transpoosille G^T.

- On myös olemassa sellainen algoritmi, joka
  - listaa komponentit jo askeleen 1 aikana

    eli käänteisessä topologisessa järjestyksessä
  - eikä siis tarvitse transpoosia G^T.
- Tämä
  - ajan
  - tilan

  säästö on käytännössä merkittävä.
- Tämä mutkikkaampi algoritmi ohitetaan tällä kurssilla.

  (Se löytyy vaikkapa sivuilta 481–483 kirjasta R. Sedgewick: *Algorithms, 2nd Ed.* Addison-Wesley, 1988.)
- Sen pseudokoodi on kuitenkin täydellisyyden vuoksi seuraavana kalvona.

```
1    S ← aluksi tyhjä pino
2    for jokaiselle verkon solmulle u do
3        d[u] ← −∞
4    for jokaiselle verkon solmulle u do
5        if d[u] = −∞ then
6            dummy ← SCCvisit(u)
```

```
SCCvisit(u)
1    time ← time+1
2    d[u] ← time
3    m ← d[u]
4    push(S, u)
5    for jokaiselle solmulle v ∈ Adj[u] do
6        if d[v] = −∞ then
7            m′ ← SCCvisit(v)
8        else
9            m′ ← d[v]
10       m ← min(m, m′)
11   if m = d[u] then
12       ▷ Tulostetaan nyt löytynyt komponentti:
13       repeat
14           p ← pop(S)
15           d[p] ← +∞
16           tulosta p
17       until p = u
18   return m
```

7.4 Lyhyimmät polut

- Tarkastellaan painotettuja verkkoja ja tulkitaan kaaren paino sen yhdistämien solmujen *etäisyydeksi*.

- Kaaripainon voi tulkita myös vaikkapa ajaksi, joka kuluu matkustettaessa tuo kaaren yhdistämien solmujen välinen matka.

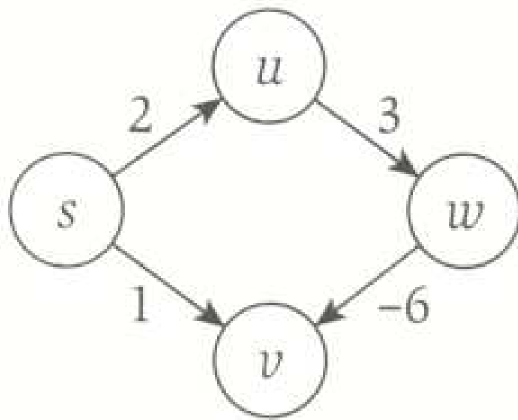
- Tehtävänä on löytää *yhdestä* annetusta solmusta s lyhyin etäisyys, eli vähiten painava polku, *kaikkiin muihin* solmuihin.

- Ongelma on keskeinen esim. tietoliikenneverkkojen reitityksessä.

- Esitellään ratkaisuksi *Dijkstran algoritmi*, joka olettaa että *kaaripainot ovat* ≥ 0.

- Seuraava kuva (a) antaa esimerkin, jossa oletus ei päde, ja algoritmi vastaisi väärin.

  (Kuva 6.21 kirjasta J. Kleinberg, E. Tardos: *Algorithm Design.* Addison-Wesley, 2005.)



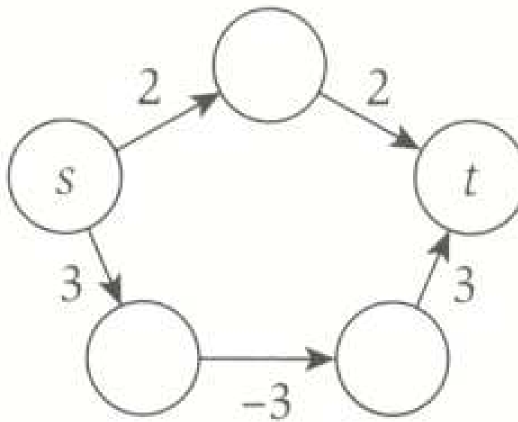


Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest s-t path.

- Muita lyhyimpien polkujen algoritmeja ovat esimerkiksi

  Bellmanin ja Fordin (kalvoilla 7.4.2) joka

  - ratkaisee saman tehtävän kuin Dijkstra
  - mutta hitaammin
  - koska sallii jopa yhteispainoltaan negatiiviset kehät.

  Floydin (kalvoilla 7.4.3) joka

  - laskee koko *välimatkataulukon* eli lyhyimmät etäisyydet *jokaisesta* solmusta jokaiseen muuhun solmuun
  - sallii kyllä negatiiviset kaaripainot
  - mutta ei salli negatiivisia kehiä
  - vaan huomauttaa, jos syötteessä on niitä.

- Edellinen kuva (b) osoittaa, että kaaripainojen kasvattaminen voi vaihtaa lyhyimmän polun toiseksi, eikä siis sovi negatiivisten kaaripainojen eliminointiin.

- Olkoon $G = (V, E)$ suunnattu verkko ja s eräs sen solmu.

- Oletetaan, että jokaiseen kaareen $(u, v) \in E$ on liitetty paino $w(u, v)$, joka on ei-negatiivinen reaaliluku.

- Oletetaan lisäksi:

  – Jos $(u, v) \notin E(G)$ niin $w(u, v) = \infty$, eli jos solmuja ei yhdistä kaari, niin niiden välinen suora etäisyys on ääretön.

  – $w(v, v) = 0$, eli etäisyys jokaisesta solmusta itseensä on nolla.

- Eräs tapa hahmottaa Dijkstran algoritmi:
  - Solmut ovat kaupunkeja, kaaret teitä naapurikaupunkien välillä.
  - Kuningas haluaa lähettää pääkaupungistaan s viestin maansa kaikkiin kolkkiin.
  - Viestinviejät kulkevat
    * kaupungista toiseen teitä pitkin
    * keskenään samalla nopeudella

$$c \text{ km/h}.$$

  - Kuningas lähettää yhtä aikaa pääkaupungista s viestinviejän sen jokaiseen naapurikaupunkiin.
  - Kuningas päivää näihin viesteihin niiden yhteisen lähettämishetken t.

- Dijkstran algoritmin hahmotus, jatkoa...
  - Kun viestinviejä x saapuu kaupunkiin u, niin hän kysyy sen asukkailta ”Oletteko jo kuulleet kuninkaan viestin?”

    Ei: silmänräpäyksessä x

    * kuuluttaa viestin kaikille asukkaille
    * värvää heistä uudet viestinviejät

      yhden jokaiselle tästä kaupungista vievälle tielle
    * lähettää heidät matkaan.

    Kyllä (eli joku toinen viestinviejä y on ehtinyt tänne ensin): x ei tee mitään.

    Nyt x saa levätä, työ on hänen osaltaan ohi.
  - Kaupungin u etäisyys pääkaupungista s on silloin tietenkin

    $$c \cdot (t_u - t)$$

    missä t_u on se hetki, jolloin sen asukkaat kuulivat kuninkaan viestin ensimmäisen kerran.

- Nyt Dijkstran algoritmin voi hahmottaa tämän viestinviennin simulaationa:
  - Simulaation kiinnostavat ajanhetket ovat nämä eri kellonajat t_u.
  - Kun ensimmäinen sanansaattaja x saapuu kaupunkiin u, niin voidaan laskea, milloin hänen värväämänsä sanansaattaja z saavuttaa oman kohdekaupunkinsa v:

    $$t_u + \frac{w(u,v)}{c}.$$

  - Näillä laskelmilla voidaan pitää yllä ja parantaa arvioita $d_v =$ milloin kuninkaan viesti saavuttaa kaupungin v.
  - Simulaatio etenee valitsemalla aina seuraavaksi pienimmän d_v, missä kaupunki v ei vielä ole kuullut kuninkaan viestiä, ja siirtymällä kellonaikaan $t_v = d_v$.
  - Laskutoimitusten helpottamiseksi voimme vielä valita

    nopeudeksi $c = 1$ ja päiväykseksi $t = 0$.

- Dijkstran algoritmi pitää yllä joukkoa $S =$ ne solmut
  - joiden lyhyin etäisyys solmusta s on jo selvitetty
  - jotka on jo käsitelty
  - jotka on jo poistettu aputietorakenteesta.

- Jokaisella solmulla v on kaksi attribuuttia:
  - $d[v] =$ solmun v tällä hetkellä *pienin tunnettu etäisyys* solmusta s
  - $p[v] =$ se joukon S solmu, joka edeltää solmua v tällä hetkellä parhaalla tunnetulla polulla

$$\underbrace{\overbrace{s \leadsto p[v]}^{\text{joukossa } S} \xrightarrow{w(p[v],v)} v}_{\text{kokonaispaino } d[v]} .$$

- Alussa

$$\begin{aligned} p[v] &= \text{NIL} && \text{kaikilla } v \in V \\ d[s] &= 0 \\ d[v] &= \infty && \text{kaikilla muilla } v \in V \setminus \{s\} . \end{aligned}$$

- Algoritmi valitsee rivillä 9 toistuvasti sellaisen solmun $u \in V \setminus S$, jonka $d[u]$ on pienin.
  - Valittu solmu u lisätään joukkoon S.
  - Kaikkien solmun u vierussolmujen v attribuutit $d[v]$ ja $p[v]$ päivitetään ottamaan huomioon myös nyt avautunut uusi mahdollinen polku

$$s \overset{d[u]}{\leadsto} u \xrightarrow{w(u,v)} v.$$

  - Toisin sanoen asetetaan

$$d[v] \leftarrow \min(d[v], d[u] + w(u, v)).$$

1. Ensimmäisessä toistoaskeleessa tulee valituksi aloitussolmu s, sillä $d[s] = 0$ kun taas muut ∞.

   - Solmu s siis lisätään joukkoon S.

   - Jokaiselle solmun s vierussolmulle v asetetaan uusi etäisyysarvio

$$\begin{aligned} d[s] + w(s,v) &= 0 + w(s,v) \\ &= w(s,v) \end{aligned}$$

     eli sama kuin etäisyys solmusta s solmuun v.

   - Attribuuttien $p[v]$ arvoksi tulee s, sillä lyhyin polku solmusta s solmuun v saapuu itse solmusta s.

2. Toisessa toistoaskeleessa tulee valituksi se solmu u jonka etäisyys solmusta s on lyhyin, koska sen $d[u]$ on nyt pienin.

   - Solmu u siis lisätään joukkoon S.

   - Jokaiselle solmun u vierussolmuille v:

     – Kysytään, onko sen nykyinen etäisyysarvo

     $$d[v] \text{ suurempi kuin } d[u] + w(u, v)$$

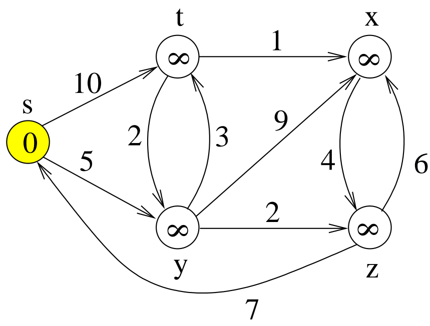
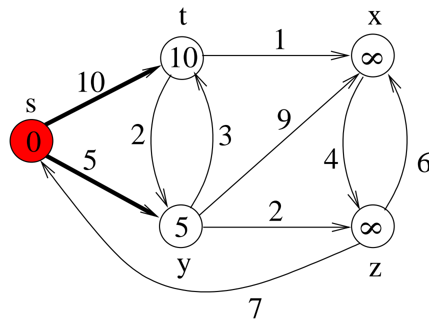
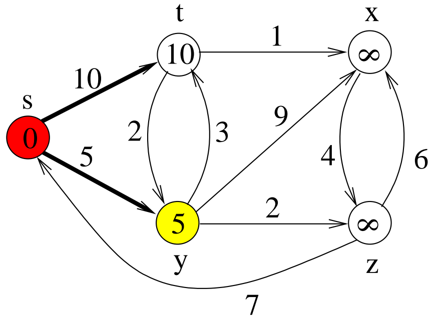
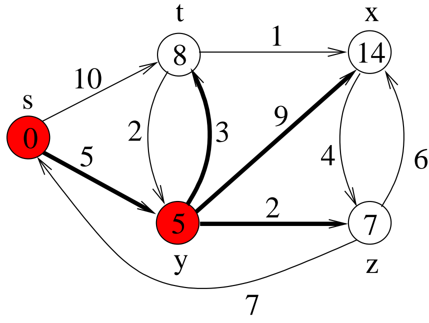
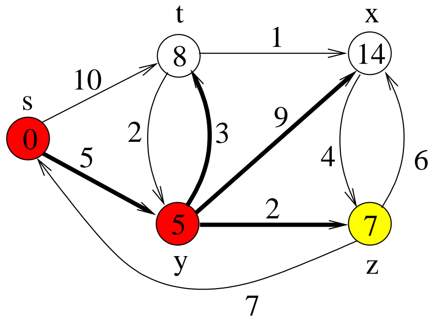
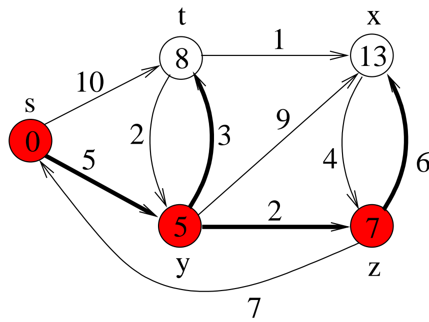
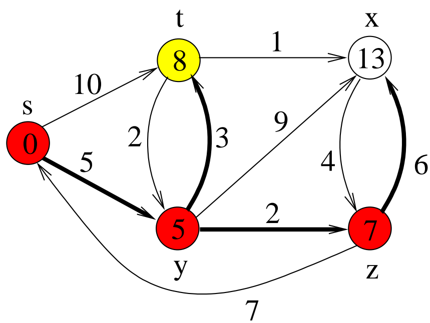
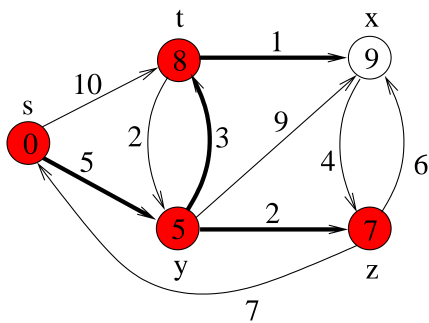
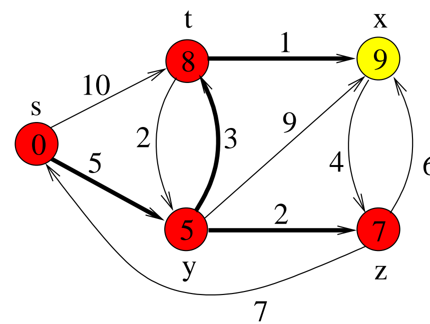
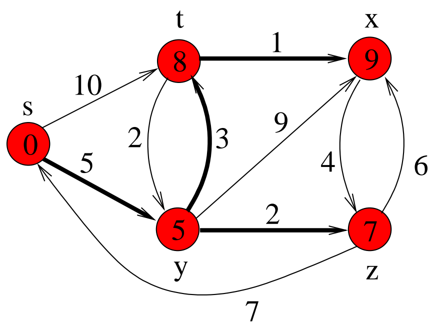
     eli nyt ilmennyt uusi mahdollisuus.

     – Jos on, niin päivitetään

       * $d[v]$ tähän uuteen pienempään mahdollisuuteen

       * $p[v]$ tulemaan sitä vastaavasta solmusta u.

3. Sama jatkuu kolmannessa toistoaskeleessa. . .


s t x y z
0 ∞ ∞ ∞ ∞
10 1 2 3 9 4 6 5 2 7

s t x y z
0 10 ∞ 5 ∞
10 1 2 3 9 4 6 5 2 7

s t x y z
0 10 ∞ 5 ∞
10 1 2 3 9 4 6 5 2 7

s t x y z
0 8 14 5 7
10 1 2 3 9 4 6 5 2 7

s t x y z
0 8 14 5 7
10 1 2 3 9 4 6 5 2 7

s t x y z
0 8 13 5 7
10 1 2 3 9 4 6 5 2 7

s t x y z
0 8 13 5 7
10 1 2 3 9 4 6 5 2 7

s t x y z
0 8 9 5 7
10 1 2 3 9 4 6 5 2 7

s t x y z
0 8 9 5 7
10 1 2 3 9 4 6 5 2 7

s t x y z
0 8 9 5 7
10 1 2 3 9 4 6 5 2 7

- Algoritmi käyttää aputietorakenteena kalvojen 5 *minimi*kekoa H:
  - Keko H koostuu solmuista $u \in V \setminus S$.
  - Solmun u kekoavaimena on sen etäisyysarvioattribuutin $d[u]$ nykyinen arvo.
  - Näin operaatio ”valitse seuraava käsiteltävä solmu u” saadaan nopeaksi.
  - Kun valitun solmun u vierussolmun v etäisyysarviota $d[v]$ päivitetään, niin sen
    * kekoavain pienenee
    * pitää siirtyä vastaavasti keossa H lähemmäksi juurta.
  - Tähän tarvitaan kalvojen...
    * 5.3 kaltaista operaatiota heapDecreaseKey($H, v, d[v]$)
    * 5.4 kahvaa solmua v vastaavaan kekoalkioon.

```
Dijkstra(G, w, s)
1   for kaikille solmuille v ∈ V do
2       d[v] ← ∞
3       p[v] ← NIL
4   d[s] ← 0
5   S ← ∅
6   for kaikille solmuille v ∈ V do
7       heapInsert(H, v, d[v])
8   while not empty(H) do
9       u ← heapDelMin(H)
10      S ← S ∪ {u}
11      for jokaiselle solmulle v ∈ Adj[u] do
12          if d[v] > d[u] + w(u, v) then
13              d[v] ← d[u] + w(u, v)
14              p[v] ← u
15              heapDecreaseKey(H, v, d[v])
```

- Dijkstran algoritmi voidaan toteuttaa myös ilman kahvoja (yksityiskohdat sivuutetaan).

  + Kahvattomat keot löytyvät usein aliohjelmakirjastoista valmiina.

    (Esimerkiksi ohjelmointikielen `C++` standardikirjastosta `STL`.)

  – Tilantarve kasvaa.

- Algoritmin suorituksen jälkeen lyhyin polku $s \leadsto v$ on

$$s \to \ldots \to p[p[v]] \to p[v] \to v$$

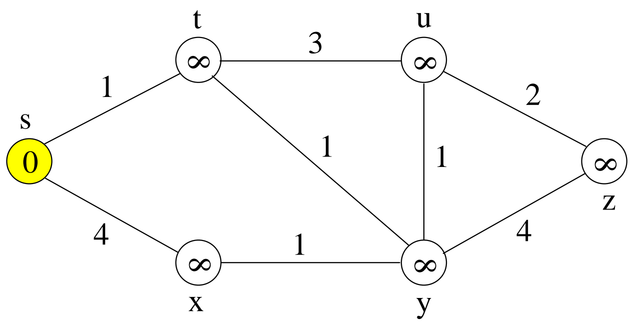
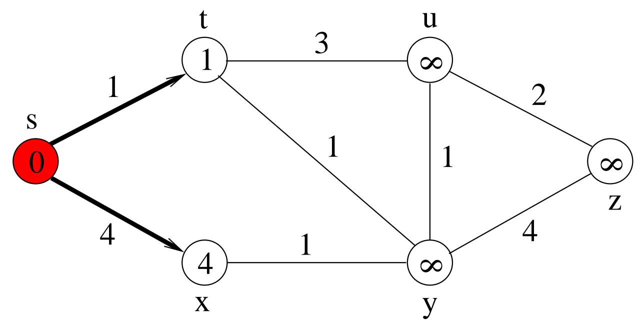
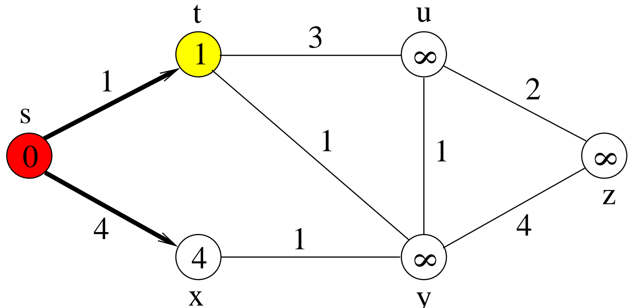
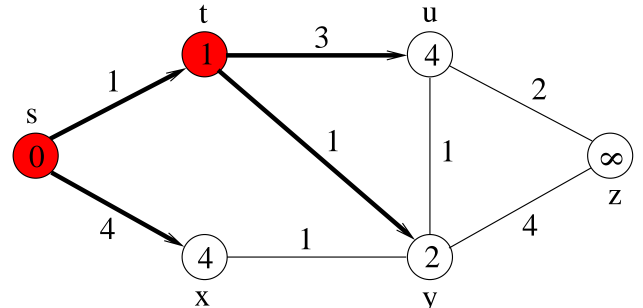
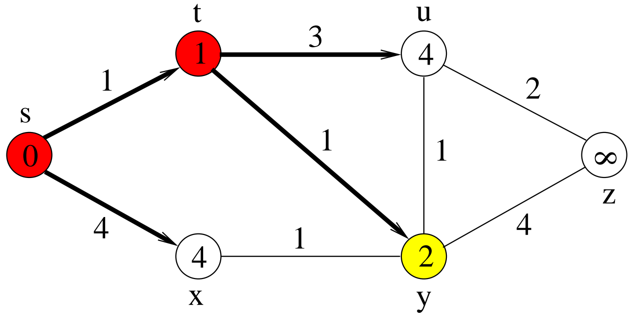
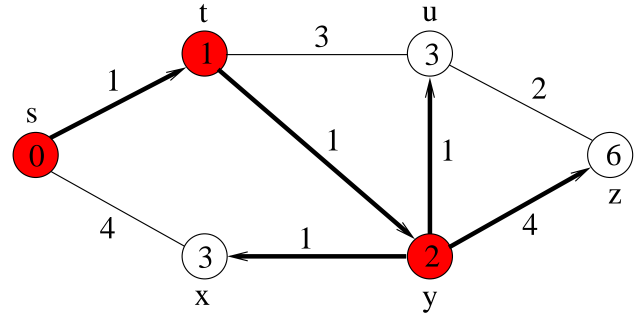
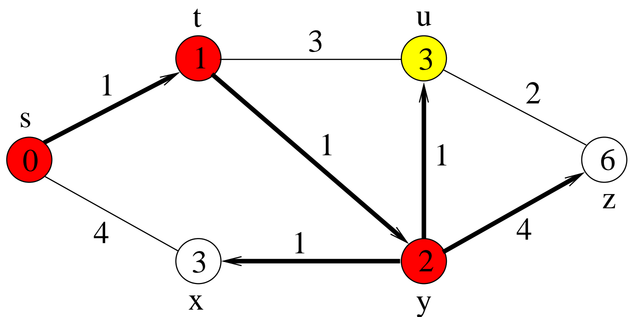
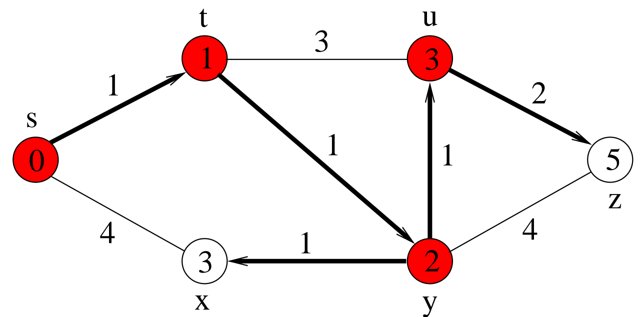
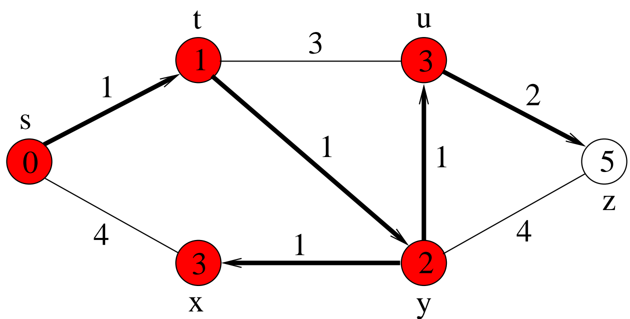
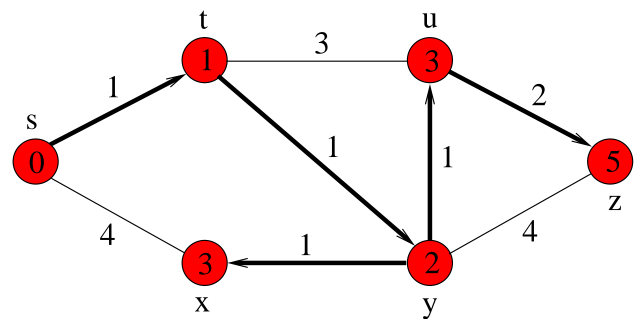
ja se voidaan

  - lukea takaperin p-attribuutteja seuraten
  - kääntää oikein päin käyttäen kalvojen 2.1 pinoa S

algoritmilla

```
shortestPath(G, v)
    repeat
        push(S, v)
        v ← p[v]
    until v = NIL
    repeat
        print(pop(S))
    until empty(S)
```

- Seuraava kuvasarja on toinen esimerkki Dijkstran algoritmin toiminnasta, tällä kertaa kyseessä on suuntaamaton verkko.


t
u
3
1
2
s
0
1
1
∞
∞
∞
z
4
1
4
∞
∞
x
y

t
u
3
1
2
s
1
∞
1
1
0
∞
z
4
1
4
4
∞
x
y

t
u
3
1
∞
2
s
1
1
0
∞
z
4
1
4
4
∞
x
y

t
u
3
1
4
2
s
1
1
0
∞
z
4
1
4
4
2
x
y

t
u
3
1
4
2
s
1
1
0
∞
z
4
1
4
4
2
x
y

t
u
3
1
3
2
s
1
1
0
6
z
4
1
4
3
2
x
y

t
u
3
1
3
2
s
1
1
0
6
z
4
1
4
3
2
x
y

t
u
3
1
3
2
s
1
1
0
5
z
4
1
4
3
2
x
y

t
u
3
1
3
2
s
1
1
0
5
z
4
1
4
3
2
x
y

t
u
3
1
3
2
s
1
1
0
5
z
4
1
4
3
2
x
y

Dijkstran algoritmin aikavaativuus:

- Rivien 1–5 alustustoimet vievät aikaa selvästi
$$\mathcal{O}(|V|).$$
- Keon H operaatiot heapInsert (rivi 7), heapDelMin (rivi 9) ja heapDecreaseKey (rivi 15) vievät aikaa
$$\mathcal{O}(\log(\underbrace{\text{keon alkioiden lukumäärä}}_{\leq |V|})).$$
- Riveillä 6–7 tehdään $|V|$ kappaletta operaatioita heapInsert, aikaa siis kuluu
$$\mathcal{O}(|V| \cdot \log(|V|)).$$
  - Sama keon H alustus voitaisiin suorittaa myös kalvojen 5.3 operaatiolla buildHeap ajassa $\mathcal{O}(|V|)$
  - mutta tämä parantaisi tehokkuutta vain käytännössä, ei teoriassa...

Aikavaativuus jatkuu...

- Rivien 8–15 toistolauseessa operaatiota heapDelMin kutsutaan kullekin solmulle kerran, eli yhteensä $|V|$ kertaa.

- Koska jokainen solmu u lisätään joukkoon S vain kertaalleen, niin jokainen vieruslista Adj$[u]$ käydään läpi täsmälleen kerran.

- Siis jokaista kaarta

  $$u \xrightarrow{w(u,v)} v$$

  tutkitaan rivillä 12 vain kerran, eli operaatiota heapDecreaseKey kutsutaan $\leq |E|$ kertaa.

- Yhteensä rivien 8–15 toistolauseessa kuluu aikaa

  $$\mathcal{O}\left((|E| + |V|) \cdot \log(|V|)\right)$$

  ja se kattaa myös rivien 1–7 alustukset.

Keon vaikutuksesta aikavaativuuteen:

- Jos käytetäänkin *Fibonaccin* kekoa, niin teoreettinen aikavaativuus paranee muotoon

$$\mathcal{O}\left(|E| + |V| \cdot \log |V|\right).$$

- Käytännössä tämä parannus ei liene useinkaan tarpeen, koska
  - ohjelmointi mutkistuu ja vakiokertoimet kasvavat
  - tyypillisessä käytössä Dijkstran algoritmin aikavaativuus lienee lähellä lineaarista

    $$\mathcal{O}\left(|E| + |V|\right)$$

    jolloin keko ei juurikaan vaikuta.

- Dijkstran algoritmi toimii näet sitä tehokkaammin, mitä nopeammin solmujen v arvot $d[v]$ putoavat alaspäin.
  - Mitä pienempi $d[v]$, sitä useampi kaari $u \xrightarrow{w(u,v)} v$ voidaan sivuuttaa.
  - ja tämä lienee usein mahdollista.

Tilavaativuus:

- Alussa kaikki solmut ovat keossa H.
- Tämän jälkeen keko H alkaa pienentyä solmujen siirtyessä samalla joukkoon S
- Tilavaativuus on siis selvästi yhteensä

$$\mathcal{O}\left(|V|\right).$$

- Kahvattomilla keoilla se onkin $\mathcal{O}\left(|E|\right)$.

Ahneudesta: Dijkstran algoritmi noudattaa ns. *ahneen algoritmin* (greedy algorithm) strategiaa:

- Rivillä 9 valitaan aina käsiteltäväksi se solmu u, joka on tällä hetkellä lähimpänä aloitussolmua s.
- Ahneudella tarkoitetaan tässä sitä että algoritmi pyrkii joka hetkellä juuri silloin "parhaalta" vaikuttavaan ratkaisuun.
- Miksi muka lyhytnäköinen ahneus juuri nyt johtaisi aikanaan parhaaseen mahdolliseen lopputulokseen?

Oikeellisuudesta: Todistus. . .

- löytyy sekä Karvin monisteesta että Cormenin kirjasta, sivuutetaan tässä yksityiskohdat.

- perustuu invarianttiin

  rivien 8–15 toistolauseen alussa kaikille solmuilla $v \in S$ pätee, että laskettu arvio $d[v]$ on sama kuin todellinen lyhyin etäisyys $s \rightsquigarrow v$

  eli että kun solmu on lisätty joukkoon S, niin sen todellinen lyhyin etäisyys aloitussolmusta s on jo selvinnyt.

- Invariantti pätee ensimmäisen kierroksen jälkeen, koska silloin

$$S = \{s\}$$
$$d[s] = 0.$$

- Invariantin säilyminen kierrokselta seuraavalle johtuu siitä, että kielsimme negatiiviset kaaripainot.

Invariantin säilymisestä: Olkoon u seuraava käsiteltävä solmu.

- Voiko sen kautta pienentää jonkin jo käsitellyn solmun u' arvoa $d[u']$?
  - Nyt $d[u'] \leq d[u]$, koska u' valittiin ennen kuin u.
  - Silloin polku $s \overset{d[u]}{\leadsto} u \overset{e}{\leadsto} u'$ voisi parantaa arvoa $d[u']$ vain, jos sen loppuosan pituus olisi $e < 0$.
- Onko verkossa vielä lyhyempää polkua $s \leadsto u$ kuin $d[u]$?
  - Jaetaan lyhyin sellainen osiin $s \overset{d[p]}{\leadsto} p \xrightarrow{w(p,q)} q \overset{e}{\leadsto} u$ missä q on ensimmäinen solmu joukon S ulkopuolella.
  - Polulla on sellainen q, koska $s \in S$ mutta $u \notin S$.
  - Nyt $d[q] = d[p] + w(p,q) \geq d[u]$, koska u valittiin ennen kuin q.
  - Jälleen pitäisi olla $e < 0$.

- Joskus meille riittää tietää pelkästään kahden solmun s ja v välinen lyhyin polku.

- Edellisen invariantin nojalla ajetaan Dijkstran algoritmia lähtösolmuna s
  - siihen asti kunnes solmu v lisätään joukkoon S
  - ja siihen voidaan lopettaa, sillä lyhyin polku

    $$s \rightsquigarrow v$$

    on jo selvinnyt.

7.4.1 Ongelmanratkaisusta Dijkstralla

- Tarkastellaan seuraavaa ”vankilapako-ongelmaa”:
  - Syötteenä saadaan vankilan pohjapiirros matriisina $B[0 \ldots 2 \cdot m][0 \ldots 2 \cdot m]$.
  - Jokainen paikka $B[i][j]$ on joko *käytävää* tai *muuria*.

<table>
<tr><td>●</td><td></td><td></td><td>●</td><td>●</td></tr>
<tr><td>●</td><td></td><td>●</td><td></td><td>●</td></tr>
<tr><td>●</td><td>●</td><td>?</td><td>●</td><td></td></tr>
<tr><td>●</td><td></td><td></td><td>●</td><td>●</td></tr>
<tr><td></td><td>●</td><td>●</td><td></td><td>●</td></tr>
</table>

- Olemme aluksi vankilan keskipaikassa $B[m][m]$ (joka on käytävää).

- Tavoitteena on päästä vankilasta sen jonkin reunan yli vapauteen.

- Voimme liikkua vankilassa seuraavasti:
  - Nykyisestä paikasta voi siirtyä mihin tahansa naapuripaikkaan, ei kulmittain.
  - Käytävillä voi hiiviskellä, mutta niillä *ei kannata maleksia* turhaan.
  - Muuriin täytyy ensin *kaivautua* ennen kuin siihen voi kulkea.

- Pakosuunnitelmassa on tärkeintä, ettei kaiveta yhtään enempää kuin aivan välttämätöntä.

  Myös hiiviskelyä pitäisi välttää, jos mahdollista.

- Voisimme soveltaa kalvojen 3.7 peruuttavaa etsintää hyvän pakosuunnitelman löytämiseksi
  - mutta algoritmi olisi luultavasti hidas
  - koska mahdollisia pakoreittejä on paljon.

- Mallinnetaan sen sijaan pakoreitti lyhyimpänä polkuna verkossa G:

  – $V =$ vankilan paikat sekä lisäpaikka t eli ”vapaudessa”.

  – $p \xrightarrow{w} q \in E$ jos ja vain jos paikka q on paikan p naapuripaikka.

    Jokaisen reunapaikan p naapurina on myös lisäpaikka t.

  – Kaaripainon $w \geq 0$ määrää kohdepaikka q:

    käytävällä $w = 1$

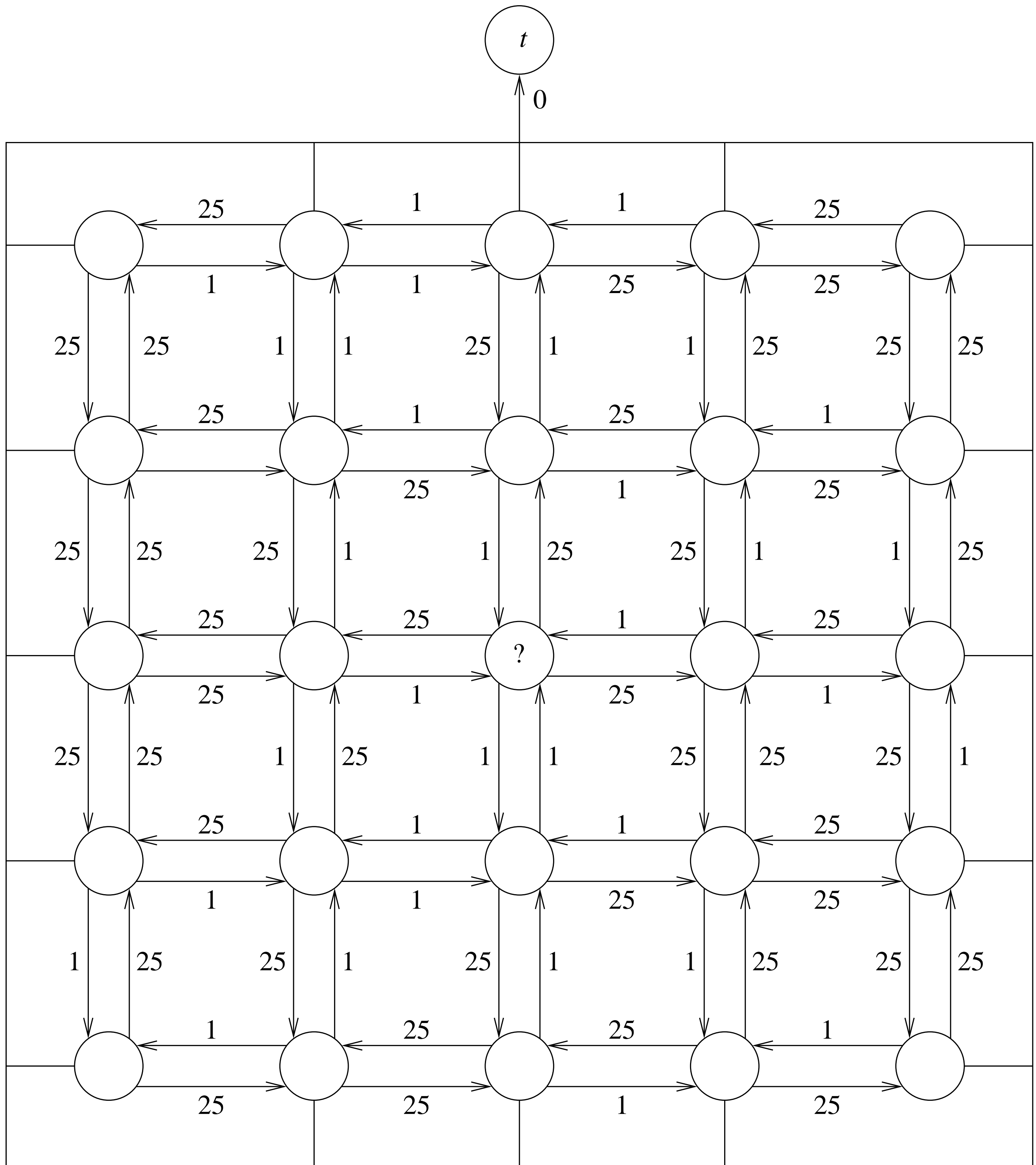
    muurilla $w = (2 \cdot m + 1)^2$ eli vankilan koko pinta-ala

    reunalla $w = 0$ eli silloin kun $q = t$.

    Tällöin yksikin kaivautuminen johtaa suurempaan etäisyyteen kuin pisinkään hiiviskely.

  – Ei tarvitse muistaa kaivettuja kohtia:

    lyhyin polku on yksinkertainen (kun kaaripainot ≥ 0).


t
0
25 1 1 25
1 1 25 25
25 25 1 1 25 1 1 25 25 25
25 1 25 1
25 1 25
25 25 25 1 1 25 25 1 1 25
25 25 1 25
?
25 1 25 1
25 25 1 25 1 1 25 25 25 1
25 1 1 25
1 1 25 25
1 25 25 1 25 1 1 25 25 25
1 25 25 1
25 25 1 25

- (Jokin) mahdollisimman hyvä pakosuunnitelma löytyy siis
  - ajamalla Dijkstran algoritmia
  - verkossa G
  - alkusolmusta $s = B[m][m]$ lähtien
  - kunnes keon H päälle tulee t
  - jolloin pakosuunnitelma on polku

$$s \to \ldots \to p[p[t]] \to p[t] \to t.$$

- Verkkoa G ei tarvitse tallettaa erikseen:
  - Solmun $B[i][j]$ mahdolliset seuraajasolmut ovat $B[i \pm 1, j]$ ja $B[i, j \pm 1]$.
  - Indeksoinnin ylitys tarkoittaa lisäsolmua t.
  - Kaaripaino katsotaan kohdesolmusta.

Siis vieruslistat voidaan *laskea* algoritmin edetessä indekseistä i, j.

- Kaaripainot voidaan esittää suoremminkin:
  - Käytetäänkin luku*pareja* $\langle a, b \rangle$ missä
    * $a =$ muuri-
    * $b =$ käytävä-

    paikkojen lukumäärä polulla.
  - Kaaripainot w pareina

    käytävällä $w = \langle 0, 1 \rangle$

    muurilla $w = \langle 1, 0 \rangle$

    reunalla $w = \langle 0, 0 \rangle$.
  - Kekovertailussa $\langle a, b \rangle \leq \langle c, d \rangle$ täsmälleen kun

    $$a < c \textbf{ or } (a = c \textbf{ and } b \leq d).$$
  - Polun pituus

    $$s \xrightarrow{\langle k_1, m_1 \rangle} p_1 \xrightarrow{\langle k_2, m_2 \rangle} p_2 \xrightarrow{\langle k_3, m_3 \rangle} \ldots$$

    summataan komponenteittain

    $$\langle k_1 + k_2 + k_3 + \ldots, m_1 + m_2 + m_3 + \ldots \rangle .$$

7.4.2 Bellmanin ja Fordin algoritmi

- Tarkastellaan yhä samaa ongelmaa kuin Dijkstran algoritmissakin (kalvoilla 7.4)...

- ...mutta sallien nyt myös *negatiiviset* kaaripainot.

- Määritellään: Jos verkossa on sellainen polku

$$s \to \bigcirc \to \bigcirc \to \cdots \to \underbrace{u \xrightarrow{w_1} \bigcirc \xrightarrow{w_2} \bigcirc \xrightarrow{w_3} \cdots \xrightarrow{w_k} u}_{\text{kehä}} \to \bigcirc \to \bigcirc \to \cdots \to v \quad (15)$$

  jossa kehän yhteenlaskettu kaaripaino onkin negatiivinen eli

$$w_1 + w_2 + w_3 + \cdots + w_k < 0$$

  niin sovitaan solmun v etäisyydeksi lähtösolmusta s

$$d[v] = -\infty.$$

  - Sinnehän päästään mielivaltaisen halvalla
  - kiertämällä kehä tarpeeksi monesti.

- Algoritmin toimintaperiaate:
  - Jos solmun u todellinen etäisyys alkusolmusta s on äärellinen, niin vastaava polku

    $$s \rightsquigarrow u$$

    on yksinkertainen.
  - Yksinkertaisella polulla on

    $$\leq |V| - 1 \quad \text{kaarta.}$$

  - Päivitetään siis etäisyydet $d[u]$ yhteensä

    $$|V| \quad \text{kertaa}$$

  - Jos viimeisellä päivityskerralla $d[u]$...

    pysyy samana niin ollaan jo löydetty todellinen äärellinen etäisyys ja vastaava polku

    $$u \leftarrow p[u] \leftarrow p[p[u]] \leftarrow \cdots \leftarrow s.$$

    pienenee yhä niin u on negatiivisella kehällä (15)

    joten pitääkin lopuksi korjata $d[v] \leftarrow -\infty$ kaikille solmuille joihin on polku $u \rightsquigarrow v$.

```
1   for  jokainen solmu v ∈ V do
2       d[v] ← +∞
3       p[v] ← NIL
4   d[s] ← 0
5   for  |V| − 1 kertaa do
6       for  jokainen kaari u –w→ v ∈ E do
7           if  d[v] > d[u] + w then
8               d[v] ← d[u] + w
9               p[v] ← u
    ▷ Jälkikäsittelyvaihe alkaa:
10  for  jokainen kaari u –w→ v ∈ E do
11      if  d[v] > d[u] + w then
12          korjaa(v)

procedure korjaa(x)
13  if  d[x] ≠ −∞ then
14      d[x] ← −∞
15      for  jokainen solmu y ∈ Adj[x] do
16          korjaa(y)
```

- Alustukset (rivit 1–4) ovat kuten Dijkstran algoritmissakin.

- Päivityssääntökin (rivit 7–9) on sama.

- Toisaalta kaaret käydään läpi mielivaltaisessa järjestyksessä (riveillä 6 ja 10).

- Ensin tehdään $|V| - 1$ päivityskierrosta (rivit 5–9).

- Seuraavassa kuvassa on algoritmin etenemisestä esimerkki
  - jossa jälkikäsittelyvaihetta (rivejä 10–12) ei tarvitakaan
  - koska verkossa ei ole negatiivisia kehiä.

(Kuva 24.4 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

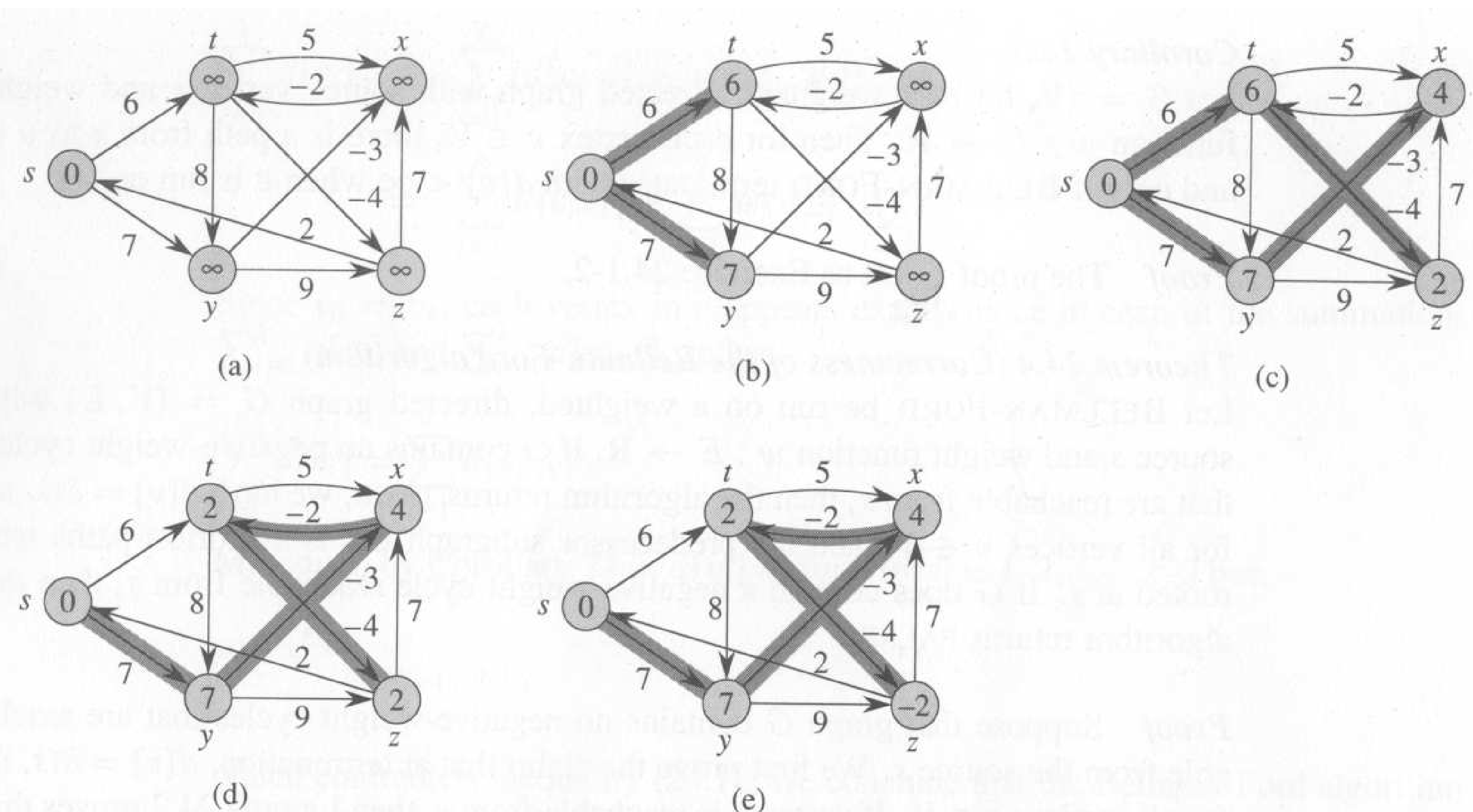


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

- Jälkikäsittelyvaihe (rivit 10–12)
  - otetaan käyttöön, jos halutaan käsitellä verkossa mahdollisesti olevat negatiiviset kehät
  - suorittaa rivit 6–7 vielä $|V|$. kerran (riveinä 10–11)
  - tunnistaa negatiiviselle kehälle kuuluvan solmun v (rivillä 11)
  - korjaa (rivillä 12) solmusta v saavutettaville solmuille etäisyydeksi $-\infty$
  - käyttää korjaamiseen kalvojen 7.3.2 syvyysläpikäynnin versiota (rivit 13–16).

- Algoritmin ajantarve on

  $$\mathcal{O}\left(|V| + |V| \cdot |E| + (|V| + |E|)\right) = \mathcal{O}\left(|V| \cdot |E|\right)$$

  eli huonompi kuin Dijkstran

  koska nyt emme voikaan valita

  ahneesti ja keolla

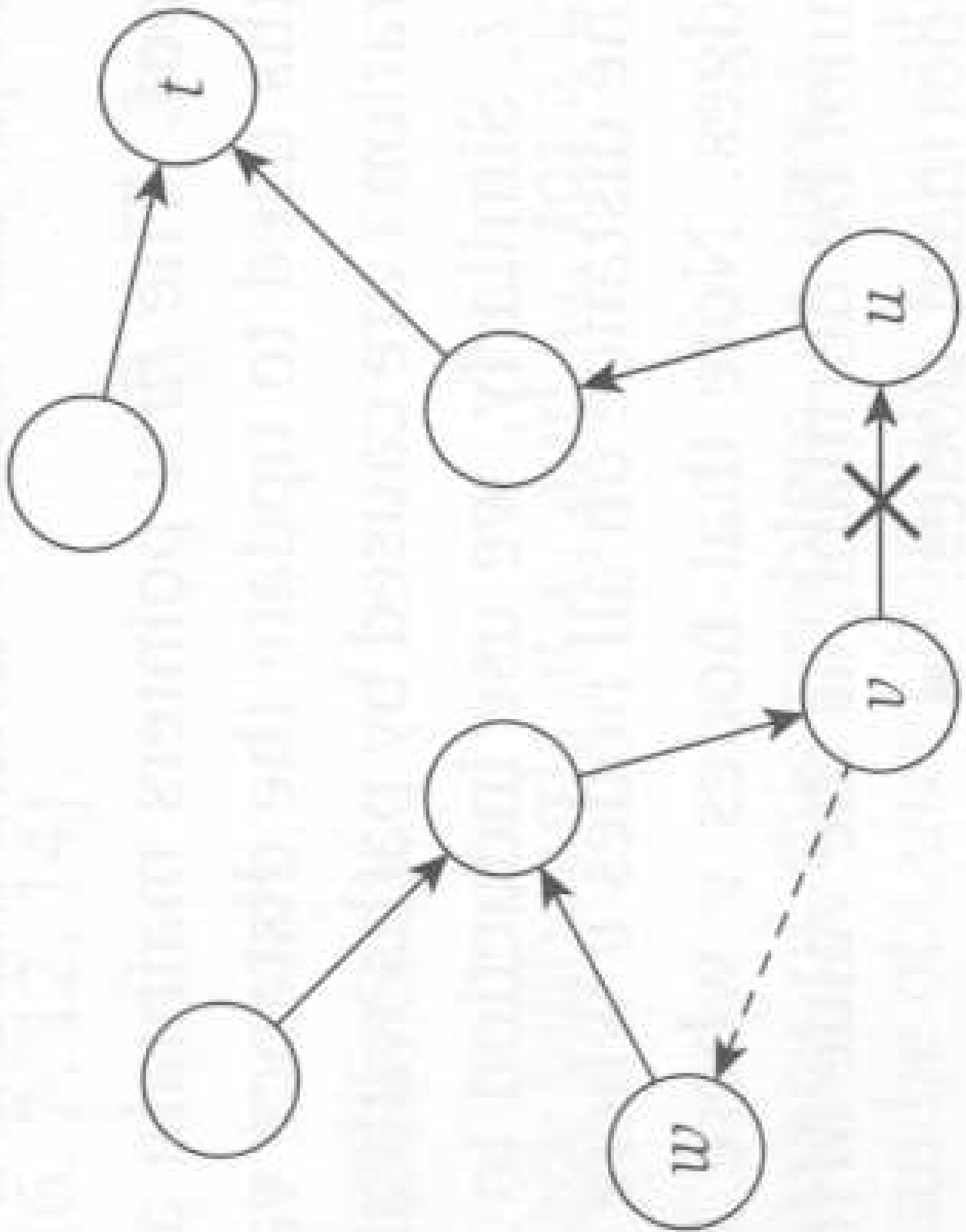
  seuraavaa solmua v, joka lopullinen arvo $d[v]$ olisi nyt selvinnyt

  vaan joudummekin päivittämään kaikkia $d[v]$ yhteensä $|V|$ kertaa.

- Seuraava kuva osoittaa, että jos solmun korjattu etäisyys on $d[v] = -\infty$, niin vastaavaan p-viiteketjuun ei voikaan enää luottaa:

  se onkin voinut kääntyä alkusolmun s sijasta löydettyyn negatiiviseen kehään!

  (Kuva 6.26 (b) kirjasta J. Kleinberg, E. Tardos: *Algorithm Design.* Addison-Wesley, 2005.)



t
u
v
w

7.4.3 Floydin algoritmi

- Halutaan laskea *matriisi* $D[1 \ldots |V|][1 \ldots |V|]$ jossa

  $D[i][j] =$ lyhyimmän polun pituus solmusta i solmuun j.

- Negatiiviset

  kaaret sallitaan mutta

  kehät kielletään.

- Vertaa tiekarttojen välimatkataulukot:

  lyhyin ajomatka kaupungista i kaupunkiin j.

- Esimerkki algoritmista, jossa kalvojen 7.2.2 vierusmatriisit ovat luontevin verkkojen esitystapa.

- Syötteenä saadaan siis vierusmatriisi

$$A[i][j] = \begin{cases} w & \text{jos on kaari } i \xrightarrow{w} j \in E \\ +\infty & \text{jos välillä ei ole kaarta.} \end{cases}$$

- Idea: Lasketaankin apumatriisit
  - $D^{(k)}[i][j]$ = lyhyimmän sellaisen polun pituus solmusta i solmuun j, jonka aikana *ei käydä väli*solmuissa

$$k+1, k+2, k+3, \ldots, |V|$$

  - järjestyksessä

$$k = 0, 1, 2, 3, \ldots, |V|.$$

  - Päätesolmut i ja j saavat olla $\geq k$.

- Ensimmäinen apumatriisi saadaan syötematriisista

$$D^{(0)}[i][j] = \begin{cases} 0 & \text{kun } i = j \\ A[i][j] & \text{kun } i \neq j \end{cases}$$

eli nollaamalla syötematriisin diagonaali.

- Seuraava apumatriisi $D^{(k)}$ voidaan muodostaa edellisestä apumatriisista $D^{(k-1)}$:

  $D^{(k)}[i][j] =$ pienempi luvuista

  $D^{(k-1)}[i][j]$**:** jos lyhyin polku *ei käy yhtään kertaa* uudessa välisolmussa k

  - vaan pysyttelee vanhoissa välisolmuissa

  $$1, 2, 3, \ldots, k-1$$

  $D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$**:** jos se *käy yhden kerran* solmussa k vanhojen lisäksi.

  - Yksi käynti riittää: koska negatiivisia kehiä ei sallita, niin solmun k toistaminen ei lyhennä polkuja.

  - Seuraava kuva esittää polun rakenteen.

    (Kuva 25.3 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

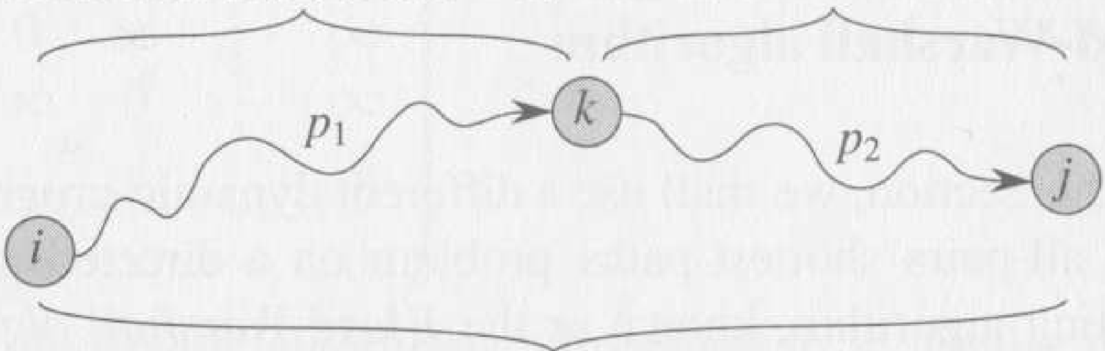


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

```
1   for jokainen solmupari i, j do
2       D[i][j] ← if i = j then 0 else A[i][j]
3   for jokainen solmu k ← 1, 2, 3, ..., |V| do
4       D' ← D
5       for jokainen solmupari i, j do
6           D[i][j] ← min(D'[i][j],
                          D'[i][k] + D'[k][j])
```

- Algoritmi käyttää vain kahta apumatriisia:

$$D = D^{(k)} \quad \text{ja}$$
$$D' = D^{(k-1)}$$

- Itse asiassa jopa D' on turha, sillä

$$D^{(k)}[i][k] = D^{(k-1)}[i][k] \quad \text{ja}$$
$$D^{(k)}[k][j] = D^{(k-1)}[k][j]$$

  - koska lyhyin polku $i \rightsquigarrow k$ ei tarvitse loppusolmuaan k välisolmunaan
  - koska muutenhan olisi negatiivinen kehä $k \rightsquigarrow k$
  - ja samoin k vs. j
  - joten rivillä 6 voi myös lukea samaa D.

- Seuraavina kuvina on

  1. esimerkkiverkko

  2. sen apumatriisit.

- Apumatriisikuvan 2 vasen puolisko esittää apumatriisit $D^{(k)}$

- Kuvan oikea puolisko taas esittää apumatriisit $\Pi^{(k)}$, joiden avulla voidaan myöhemmin vastata kysymyksiin

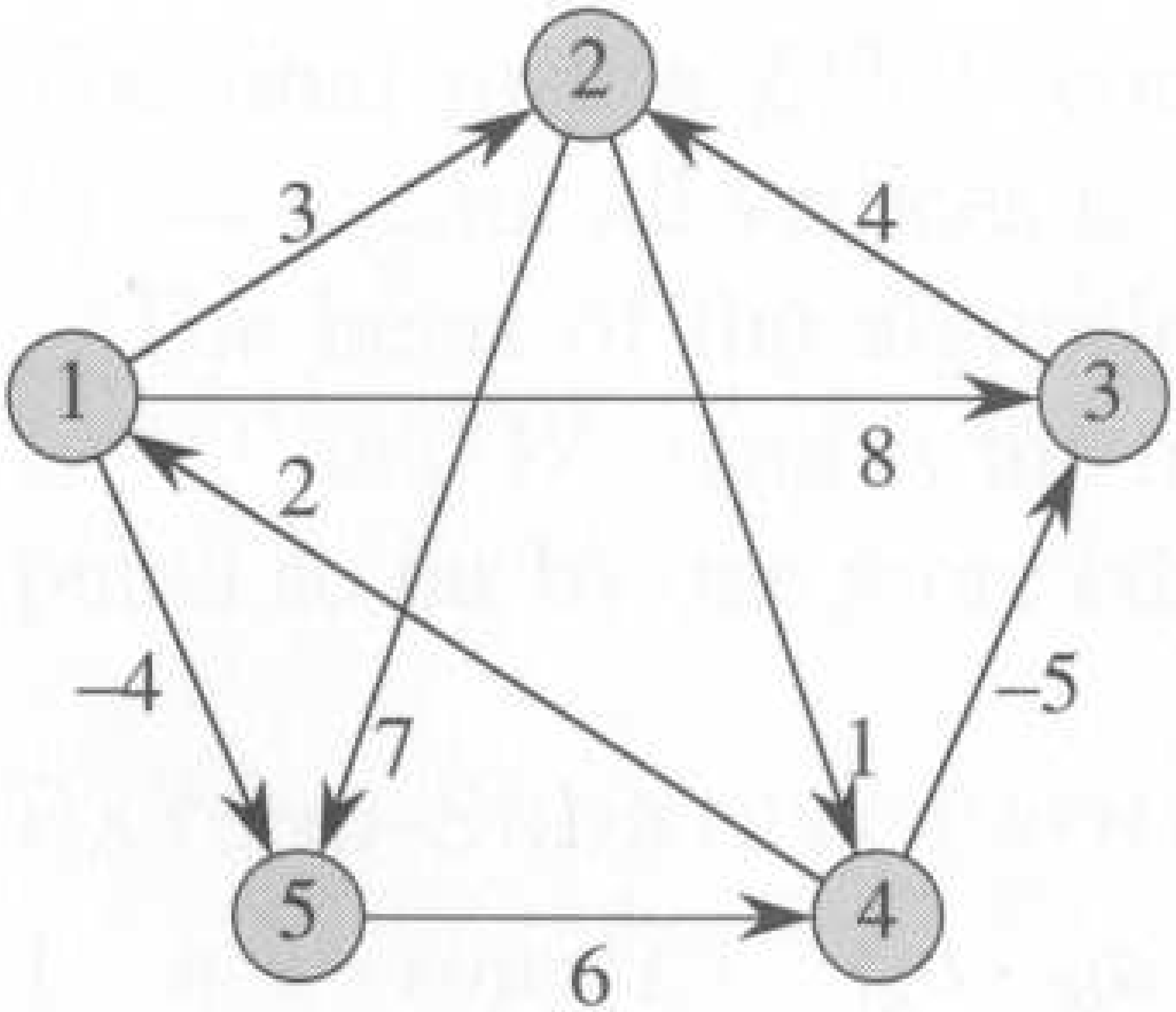
  ”Mikä on (jokin) lyhyin polku solmusta i solmuun j?”

- Määritellään samoin kuin edellä

  $\Pi[i][j] =$ toiseksi viimeinen solmu sillä polulla lähtösolmusta i loppusolmuun j jonka pituus on $D[i][j]$.

- Silloin löydetty lyhyin polku solmusta i solmuun j voidaan jäljittää takaperin

$$j \leftarrow \underbrace{\Pi[i][j]}_{q_1} \leftarrow \underbrace{\Pi[i][q_1]}_{q_2} \leftarrow \underbrace{\Pi[i][q_2]}_{q_3} \leftarrow \cdots \leftarrow i.$$



(Kuva 25.1 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 25.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

(Kuva 25.4 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)

- $\Pi[i][j]$...

  alustetaan arvolla

  NIL **jos** ollaan diagonaalilla (eli $i = j$) **tai** verkossa ei ole kaarta solmusta i solmuun j

  i **jos** ei olla diagonaalilla **ja** verkossa on kaari solmusta i solmuun j.

  päivitetään arvoon $\Pi[k][j]$ aina silloin kun $D[i][j]$ muuttuu.

- Siirrytään yhteen matriisiin D ja lasketaan myös polkuviitteet Π:

```
1  for jokainen solmupari i, j do
2      D[i][j] ← if i = j then 0 else A[i][j]
3      Π[i][j] ← if i = j or A[i][j] = +∞
                                then NIL else i
4  for jokainen solmu k ← 1, 2, 3, ..., |V| do
5      for jokainen solmupari i, j do
6          e ←D[i][k] + D[k][j]
7          if e < D[i][j] then
8              D[i][j] ← e
9              Π[i][j] ← Π[k][j]
```

- Algoritmin vaatima

  aika on

$$\mathcal{O}\left(|V|^3\right)$$

  askelta

  tila on

$$\mathcal{O}\left(|V|^2\right)$$

  muistipaikkaa tulosmatriisille D

  - koska siirryimme vain yhteen matriisiin
  - eli sama kuin syöteverkon vierusmatriisille A.

  sekä $\mathcal{O}(1)$ apumuistipaikkaa.

- Polkukysymykseen vastaaminen (takaperin):

procedure printPath(i, j)
1 **repeat**
2   **print** j
3     $j \leftarrow \Pi[i][j]$
4 **until** $j =$ NIL

- Floydin algoritmi myös *huomaa*, jos sen syötteessä onkin jokin negatiivinen kehä (15):
  - Silloin siinä on myös jokin *yksinkertainen* negatiivinen kehä.
  - Olkoon solmu u sellaisella kehällä.
  - Jos tämä kehä on 1-kaarinen eli

    $$u \xrightarrow{<0} u$$

    niin se voidaan huomata jo rivin 2 alustuksen yhteydessä.
  - Muuten se on monikaarinen. Silloin algoritmin lopussa on

    $$D[u][u] < 0$$

    samasta syystä kuin kalvoilla 7.4.2:

    Rivin 4 ulkosilmukan toisto $|V|$ kertaa huomioi kaikki yksinkertaiset kehät.

7.5 Transitiivinen sulkeuma

- Suunnatun painottamattoman verkon G *transitiivinen sulkeuma* (transitive closure) on sellainen verkko $\mathsf{TC}(G)$, että

  verkossa $\mathsf{TC}(G)$ on kaari $p \to q$

  täsmälleen silloin kun

  verkossa G on polku $p \leadsto q$.

- Esimerkiksi p ja q kuuluvat samaan kalvojen 7.3.6 vahvasti yhtenäiseen komponenttiin täsmälleen silloin kun $\mathsf{TC}(G)$ sisältää molemmat kaaret $p \to q$ ja $q \to p$.

  (Tämä olisi kuitenkin hitaampi tapa muodostaa komponentit.)

- Ja kääntäen, verkon G transitiivisen sulkeuman voisi muodostaa myös syklittömän komponenttiverkon G^{SCC} transitiivisen sulkeuman kautta.

  (Tällä saattaisi säästää aikaa, jos G^{SCC} olisi paljon pienempi kuin G.)

- Esimerkki:
  - Syötteenä annetaan kokoelma *epäyhtälöitä*
    $$x_i \leq x_j$$
    yli muuttujasymboleiden $x_1, x_2, x_3, \ldots, x_n$.
  - Halutaan tietää, mitä muita epäyhtälöitä
    $$x_p \leq x_q$$
    annetuista voidaan päätellä.
  - Epäyhtälö
    $$x_p \leq x_q$$
    voidaan päätellä jos ja vain jos on epäyhtälöketju
    $$x_p \leq x_{r_1} \leq x_{r_2} \leq x_{r_3} \leq \ldots \leq x_{r_m} \leq x_q.$$
  - Siis laaditaan verkko G, jonka

    solmut ovat muuttujasymbolien indeksit

    kaaret $i \to j$ esittävät syöte-epäyhtälöitä $x_i \leq x_j$

    ja lasketaan sen TC(G).

- Olemme jo nähneet algoritmin transitiivisen sulkeuman laskemiseksi!
  - Solmusta i on *jokin* polku solmuun j

    täsmälleen silloin kun

    solmusta i on *lyhyin* polku solmuun j.
  - Siis lisätään verkon kaarille keinotekoiset painot

    $$i \xrightarrow{1} j$$

    ja lasketaan lyhyimmät etäisyydet kaikkien solmujen välillä.
  - Eli käytetään Floydin algoritmia kalvoilta 7.4.3.
  - Kaari $p \to q$ kuuluu transitiiviseen sulkeumaan

    täsmälleen silloin kun

    tulosmatriisissa on

    $$D[p][q] < +\infty. \tag{16}$$

- Floydin algoritmia voidaan vielä yksinkertaistaa laskettaessa pelkkää transitiivista sulkeumaa:
  - Emme olekaan oikeasti kiinnostuneita lyhyimmistä poluista emmekä niiden pituuksista, joten voimme jättää pois apumatriisin Π.
  - Voimme jopa korvata etäisyysmatriisin D *totuusarvo*matriisilla

    $$T[p][q] = \text{pätisikö ehto (16) vaiko ei?}$$

    eli aloittaa alkuperäisen painottamattoman verkon vierusmatriisista A.

- Näin saadaan *Warshall*in algoritmi.
  - Hän todisti, että transitiivinen sulkeuma voidaan laskea eräänlaisena *matriisitulona*

    $$\underbrace{A \cdot A \cdot A \cdot \ \ldots \ \cdot A}_{|V| \text{ kpl.}}.$$

  - Floyd johti tuloksesta molemmat algoritmit.

```
1   for jokainen solmupari i, j do
2       T[i][j] ← i = j or A[i][j]
3   for jokainen solmu k ← 1, 2, 3, ..., |V| do
5       for jokainen solmupari i, j do
6           T[i][j] ← T[i][j] or
                          (T[i][k] and T[k][j])
```

- Vertaa Floydin algoritmin ensimmäiseen versioon

  mutta ilman apumatriisia D'.

- Resurssitarpeetkin ovat asymptoottisesti samat

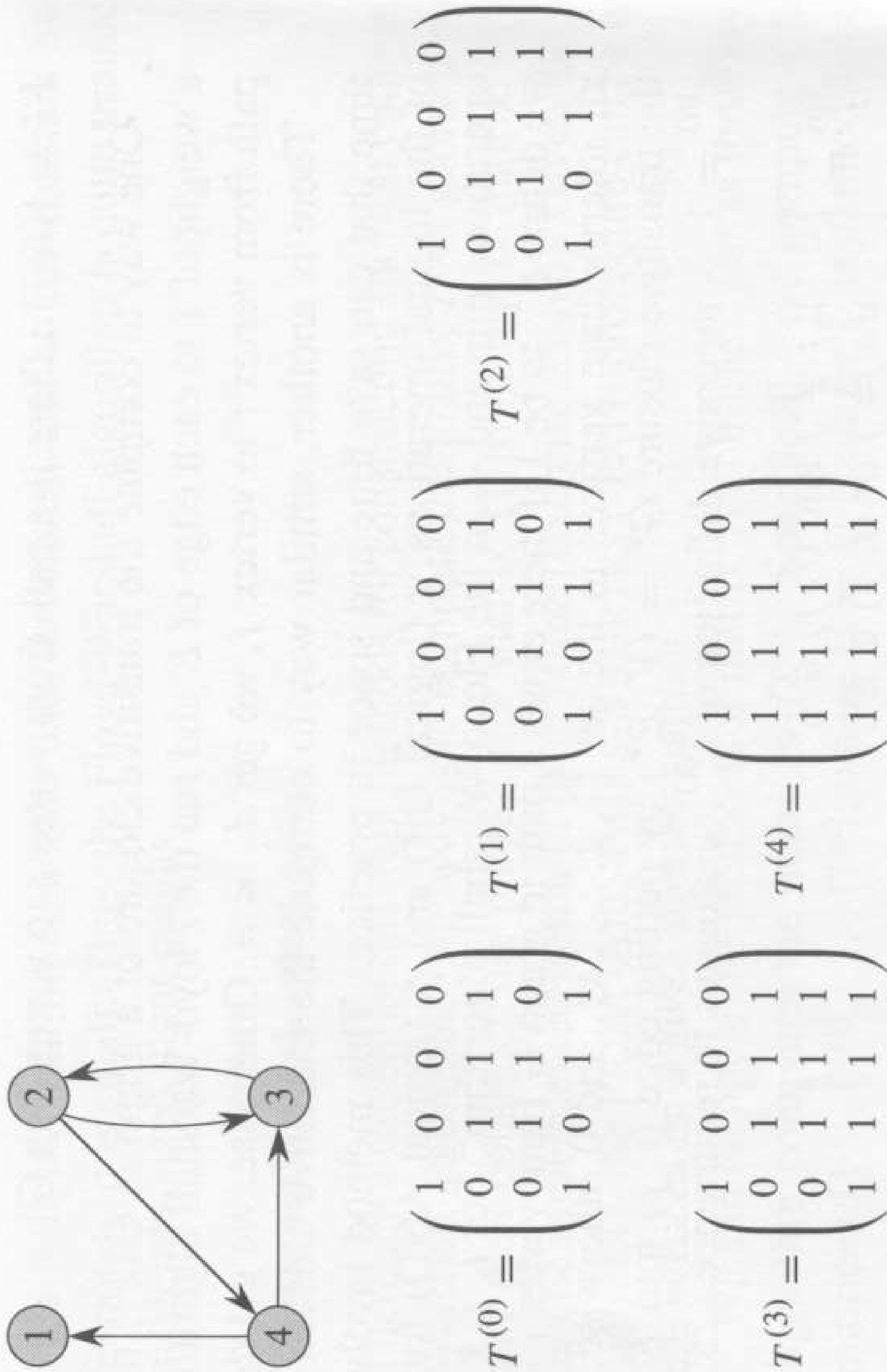
  mutta muistia tarvitaankin enää 1 *bitti* / solmupari.

- Rivin 6 merkitys ehdon (16) mukaan:

  "$T[i][j]$ on äärellinen, jos $T'[i][j]$ **tai sekä** $T'[i][k]$ **että** $T'[k][j]$ ovat".

- Seuraava kuva on esimerkki.

  (Kuva 25.5 kirjasta T.H. Cormen et al.: *Introduction to Algorithms, 2nd Ed.* MIT Press, 2001.)



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

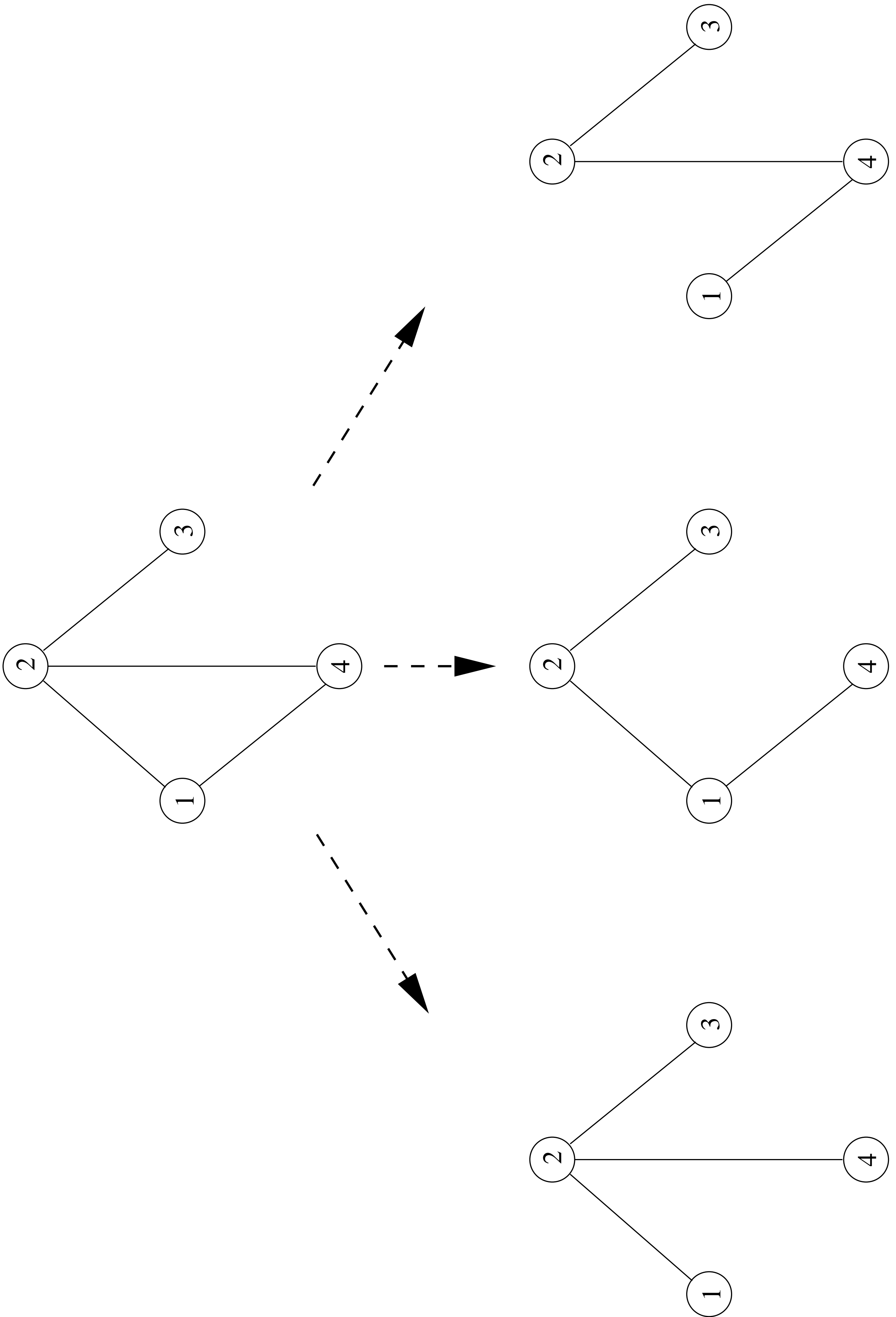
$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 25.5 A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

7.6 Verkon virittävät puut

- Olkoon suuntaamaton verkko $G = (V, E)$ *yhtenäinen*, eli
  - kaikki verkon solmut ovat saavutettavissa toisistaan
  - verkossa ei ole erillisiä osia.

- Verkon G *virittävä puu* T (spanning tree) sisältää
  - kaikki samat solmut
  - vain sen verran kaaria, että solmut pysyvät yhä juuri ja juuri saavutettavissa toisistaan:
    * Jos verkosta T poistettaisiin vielä yksikin kaari, niin se hajoaisi kahteen erilliseen osaan.
    * Verkossa T ei ole kehiä, koska kehästä voitaisiin poistaa vielä kaari, eikä verkko T silti vielä hajoaisi kahteen erilliseen osaan.



3
2
4
1
3
2
4
1
3
2
4
1
3
2
4
1

- Edellisessä kuvassa on yhtenäinen verkko ja sen kaikki erilaiset virittävät puut.

- Seuraava yksinkertainen algoritmi muodostaa verkolle $G = (V, E)$ jonkin virittävän puun T.

- T muodostetaan kaarijoukkona $E' \subseteq E$.

  spanningTree(G)
  1 $E' \leftarrow \emptyset$
  2 **for** $|V| - 1$ kertaa **do**
  3 Valitse jokin kaari u—$v \in E \setminus E'$ siten että kaarijoukossa $E' \cup \{u—v\}$ ei ole kehää
  4 $E' \leftarrow E' \cup \{u—v\}$

- Keskeinen kysymys: Miksi rivin 3 valita voidaan joka kierroksella tehdä?

  Eli ettei algoritmi voi ajautua umpikujaan väärien valintojensa seurauksena.

- Virittäviä puita käytetään monissa sovelluksissa.

- Esimerkiksi tietokoneverkkoprotokollissa ja hajautetuissa algoritmeissa koneet joutuvat usein jakamaan tietoa keskenään:

  solmuina ovat koneet

  kaarina ovat tietoliikennepiuhat koneesta toiseen

  virittävä puu on tämän tietokoneverkon "runkoverkko":

  reititstö, jota pitkin *ainakin* voidaan siirtää sanomia mistä tahansa koneesta mihin tahansa toiseen koneeseen.

- Äsken esitetty algoritmi muodostaa verkosta *jonkin* virittävän puun.

- Jos kyseessä on painotettu verkko, niin olemme yleensä kiinnostuneita *minimaalisesta* virittävästä puusta.

- Tietokoneverkossa kaaripainot voivat olla esimerkiksi kyseisen piuhan ylläpitokustannukset tms.

- Olkoon $G = (V, E)$ suuntaamaton yhtenäinen painotettu verkko.

- Verkon G *minimaalinen virittävä puu* (engl. Minimal Spanning Tree, MST) on verkon G kaikista virittävistä puista T se jonka kaaripainojen summa

$$\sum_{u \overset{w}{\text{---}} v \in T} w$$

  on pienin.

- Minimaalisia virittäviä puita voi olla useita.

- Esittelemme seuraavassa kaksi algoritmia minimaalisen virittävän puun muodostamiseen:

  Kruskalin kalvoilla 7.6.1

  Primin kalvoilla 7.6.5

  (Ne on nimetty keksijöidensä mukaan.)

- Näiden kahden algoritmin
  - yleisperiaate on aikaisemman painottamattoman algoritmin laajennus:
    - valitaan kaaria varoen kehää
    - valintajärjestys määräytyy nyt kaaripainoista
  - yksityiskohdat eroavat hieman toisistaan
  - oikeellisuus voidaan perustella lähtien samoista kummankin takana olevista periaatteista.

- Oikeellisuuteen palataan kalvoilla 7.6.8.

7.6.1 Kruskalin algoritmi

- Algoritmi alkaa muodostamaan virittävää puuta siten, että muodostusvaiheessa koossa on *useita erillisiä paloja* puusta.

- Merkitsemme attribuutilla

  $P[v] =$ se pala, johon solmu v kuuluu.

- Algoritmi

  – loppuu, kun nämä *palat ovat yhdistyneet* yhdeksi yhtenäiseksi palaksi

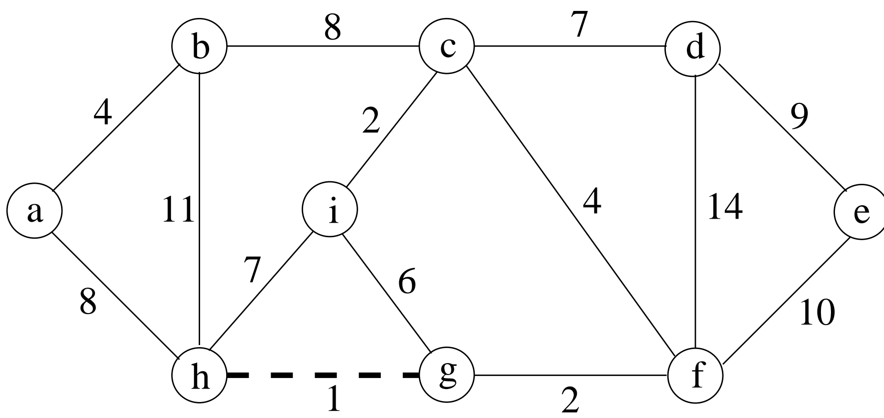
  – joka on etsitty pienin virittävä puu.

- Algoritmi kerää virittävän puun kaaria joukkoon A, ja palauttaa lopuksi tämän joukon.

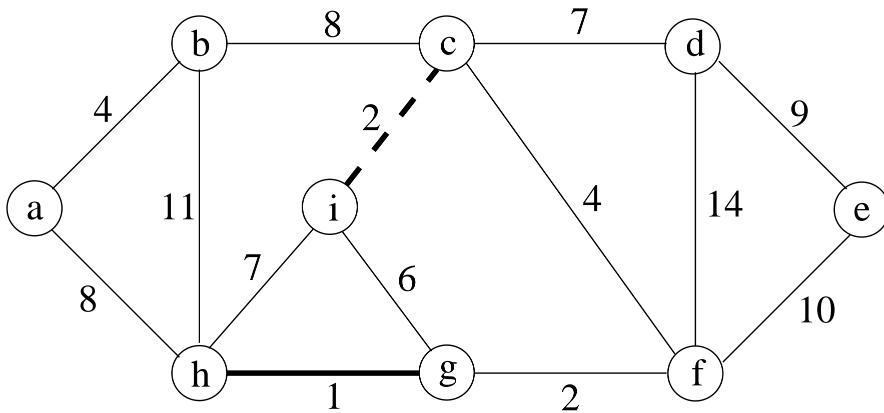
- Algoritmin lopussa verkon $G = (V, E)$ minimaalinen virittävä puu siis on (V, A).

```
Kruskal(G)
1   A ← ∅
2   for kaikille solmuille v ∈ V do
3       Solmu v muodostaa aluksi yksin
            oman palansa P[v]
4   Järjestä kaaret u —w— v ∈ E painon w mukaan
        kasvavaan järjestykseen
5   while eri palojen lukumäärä > 1 do
6       Ota järjestyksessä seuraava kaari u —w— v
7       if P[u] ja P[v] ovat eri palat then
8           A ← A ∪ {u —w— v}
9           Yhdistä P[u] ja P[v] yhdeksi palaksi
10  return A
```

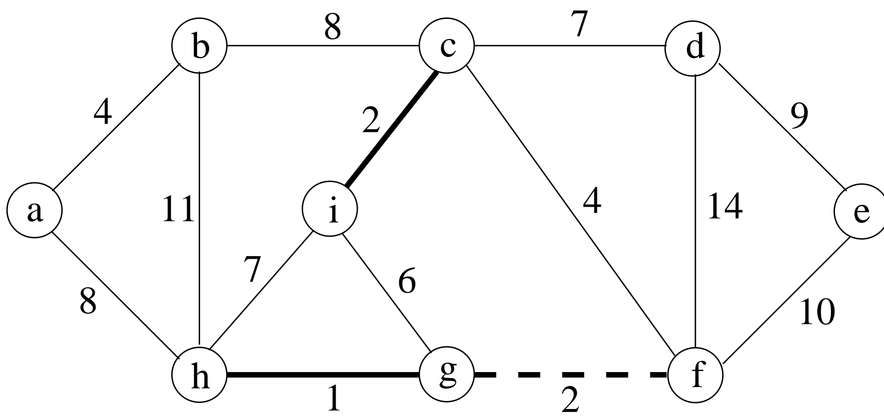
- Rivin 5 ehto voidaan tehdä nopeasti pitämällä yllä erillistä laskuria:
  - Alussa eri paloja on $|V|$ kappaletta.
  - Jokainen rivin 9 suoritus vähentää eri palojen lukumäärää yhdellä.
- Seuraava kuvasarja on esimerkki algoritmin toiminnasta.



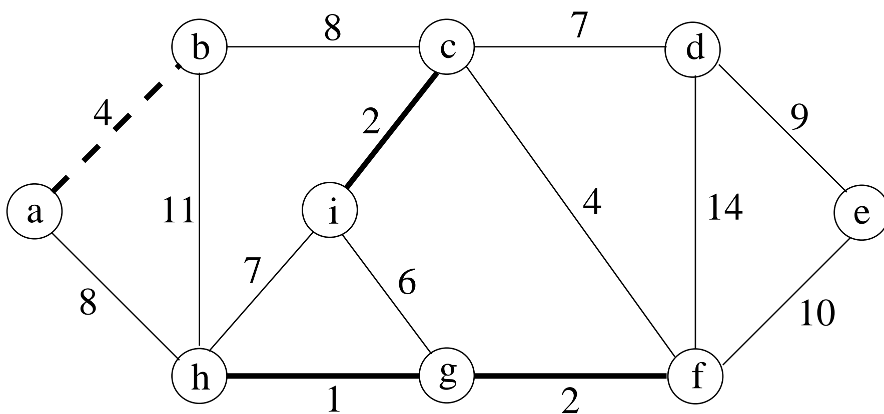
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
6
4
14
9
10
1
2



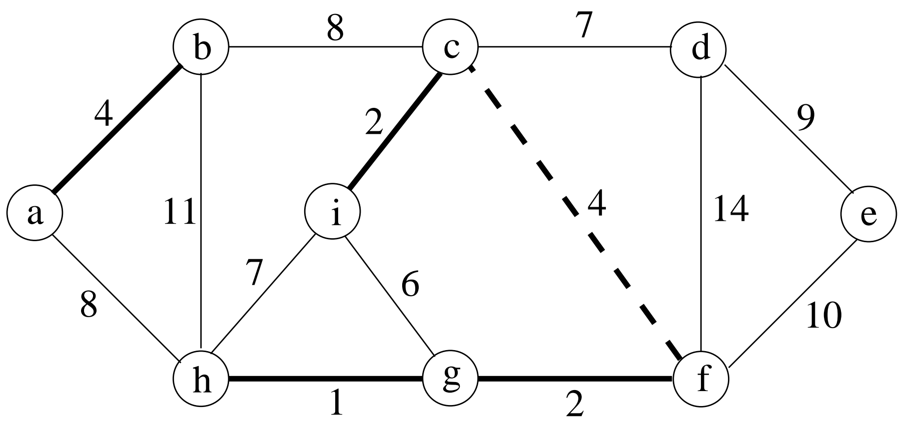
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
6
4
14
9
10
1
2



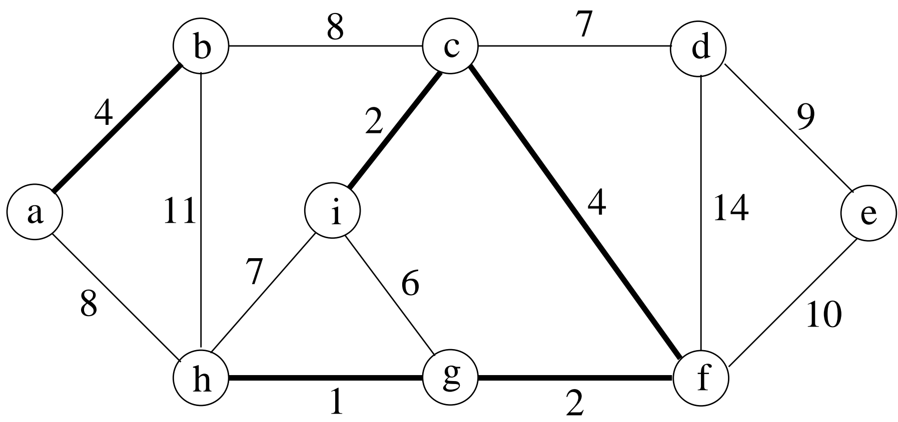
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
6
4
14
9
10
1
2



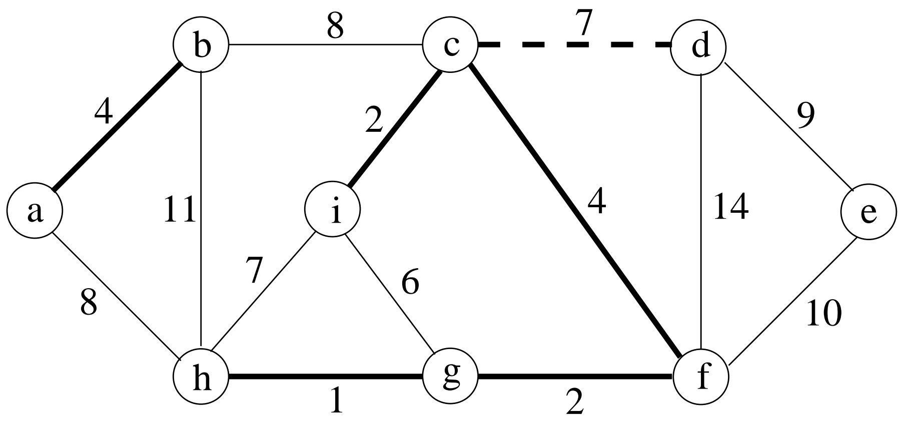
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
6
4
14
9
10
1
2



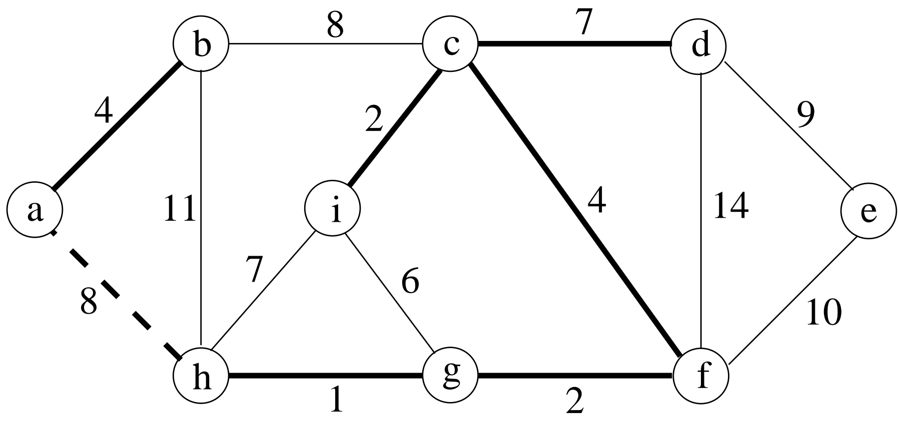
b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2



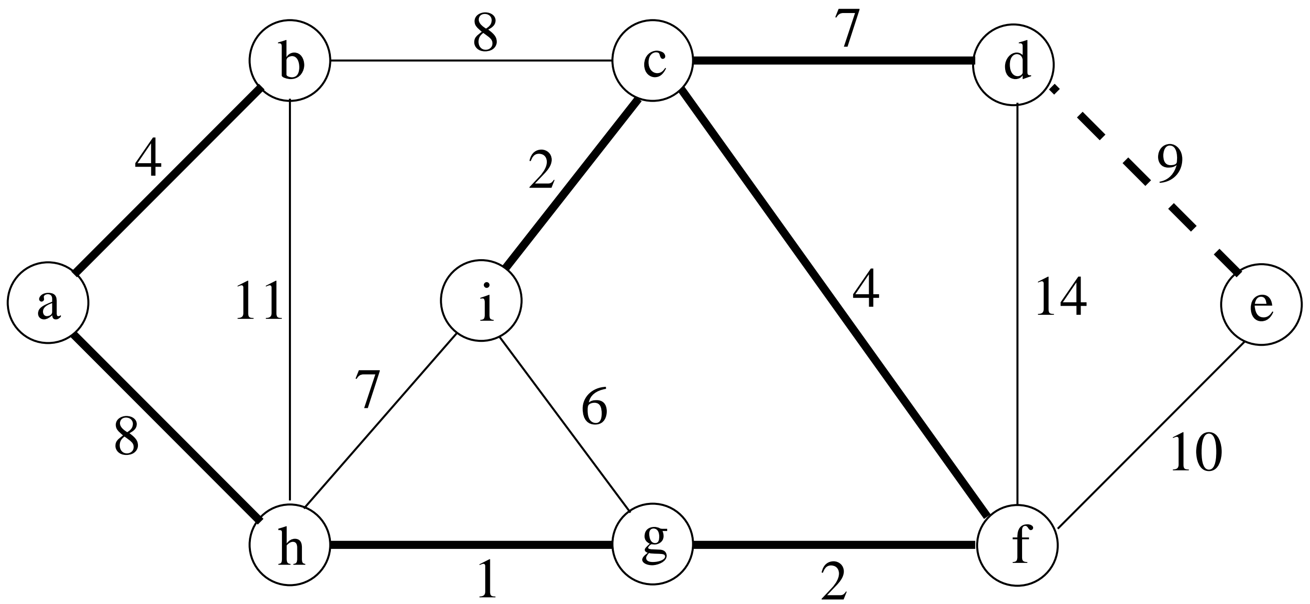
b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2

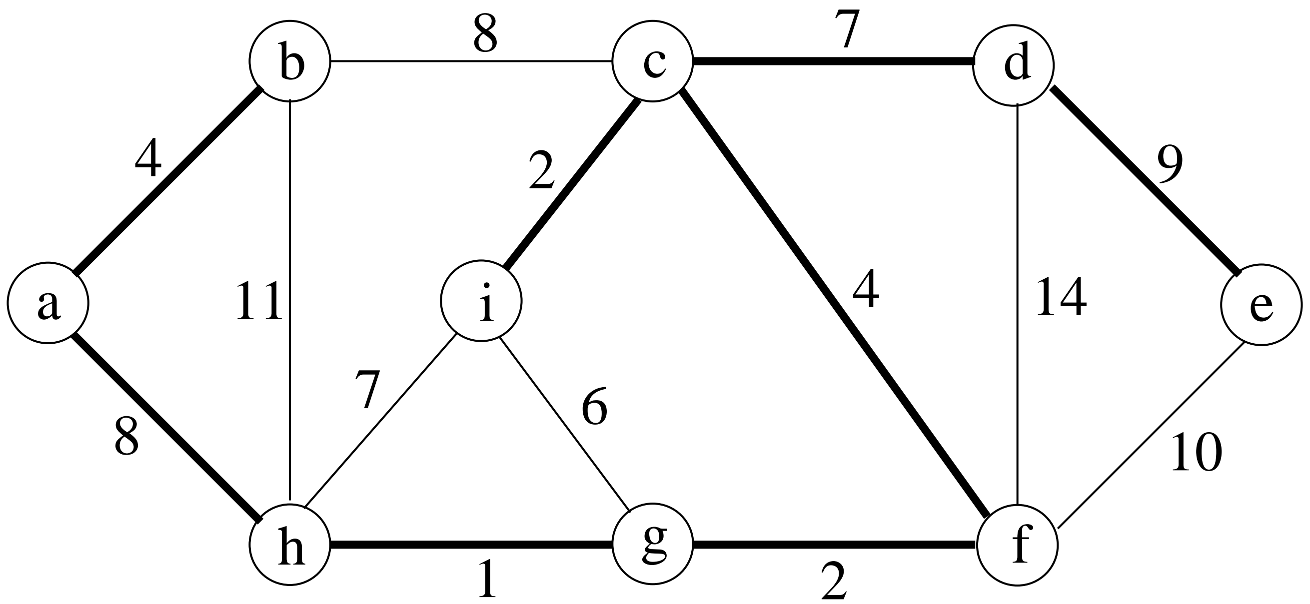


b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2



b
c
d
a
i
e
h
g
f
8
7
4
2
9
11
4
14
7
6
8
10
1
2


a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
6
1
2
4
14
9
10


a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
6
1
2
4
14
9
10

- Kruskalin algoritmi
  - on yksinkertainen hahmottaa
  - mutta ei toteuttaa tehokkaasti.

- Tehokkaassa toteutuksessa on erityisesti huomioitava se, miten solmuihin v liittyvä pala-attribuutti $P[v]$ toteutetaan.

- Helppo mutta hidas tapa palan toteuttamiseen:
  - Talletetaan palatieto $|V|$-paikkaiseen taulukkoon.
  - Alussa kunkin solmun pala olkoon oma lukunsa.
  - Rivin 7 vertailu on nyt helppo, ja hoituu järkevästi toteutettuna vakioajassa.
  - Rivillä 9 on yhdistettävä kaksi palaa.
  - Asetetaan $P[q] := P[u]$ jokaiselle sellaiselle solmulle q, jolla oli ennen $P[q] = P[v]$.

- Tällöin koko algoritmin aikavaativuus on:
  - Alkutoimet riveillä 1–3 vievät aikaa
  $$\mathcal{O}(|V|).$$
  - Rivin 4 järjestäminen vie ajan
  $$\mathcal{O}(|E| \cdot \log |E|)$$
  (kalvot 6.4).
  - Rivin 5 **while**-ehto suoritetaan

    vähintään $|V| - 1$ kertaa:
    * Puussa A on näin monta kaarta.
    * Rivin 7 ehto toteutuu näin monta kertaa
    * Jokaisella kerralla täytyy käydä läpi koko palatietotaulukko rivinä 9 — vie $\mathcal{O}(|V|)$ askelta.

    enintään $|E|$ kertaa: Puuhun A voidaan tarvita verkon painavin kaari.
  - Saamme siis vaativuudeksi yhteensä
  $$\mathcal{O}\left(|E| \cdot \log |E| + |V|^2\right).$$

- Käyttämällä palojen toteuttamiseen tämän taulukon sijasta *union-find*-tietorakennetta päästään algoritmissa aikavaativuuteen

$$\mathcal{O}\left(|E| \cdot \log |E|\right)$$

  eli rivin 4 järjestäminen alkaa hallita aikavaativuutta.

- Kun vielä huomioidaan, että

$$|E| \leq |V|^2$$

  josta taas seuraa

$$\log |E| \leq 2 \cdot \log |V|$$

  niin saamme union-find-rakenteen avulla toteutetun Kruskalin algoritmin aikavaativuuden muotoon

$$\mathcal{O}\left(|E| \cdot \log |V|\right).$$

- Aputilaa algoritmi tarvitsee
  - vastaukselleen A yhteensä $\mathcal{O}\left(|V|\right)$
  - paloilleen P yhteensä $\mathcal{O}\left(|V|\right)$
  - rivin 4 järjestämiseen siihen valittavan algoritmin mukaan.

7.6.2 Union-find-tietorakenne

- Tästä rakenteesta on hyötyä muissakin kuin vain kalvojen 7.6.1 Kruskalin algoritmissa.

- Ongelman yleinen muotoilu on pitää yllä kokoelmaa joukkoja seuraavien operaatioiden suhteen:

Yksiön luonti makeSet(x) tekee uuden joukon, jonka ainoa alkio on tämä x.

Jokainen alkio kuuluu koko ajan tasan yhteen joukkoon — joukot ovat siis *erillisiä* (disjoint sets).

Joukon haku find(x) palauttaa sen joukon, johon tämä alkio x kuuluu.

Jokaisella joukolla on oma *edustaja:* jokin joukon alkio, johon muut sen alkiot viittaavat.

Joukkojen yhdistäminen union(x, y) liittää yhteen ne joukot, joiden edustajat ovat $x \neq y$.

Tunnetaan myös nimellä ”merge”.

- Kruskalin algoritmissa palat $P[v]$ toteutetaan tällaisena joukkokokoelmana:
  - Jokainen solmu on aluksi oma yksiönsä (rivit 2–3).
  - Solmut ovat eri joukoissa jos ja vain jos niillä on eri edustajat (rivi 7).
  - Nämä eri joukot yhdistetään edustajiensa välityksellä (rivi 9).

KruskalUnionFind(G)
1 $A \leftarrow \emptyset$
2 **for** kaikille solmuille $v \in V$ **do**
3 makeSet(v)
4 Järjestä kaaret $u \overset{w}{—} v \in E$ painon w mukaan kasvavaan järjestykseen
5 **while** eri palojen lukumäärä > 1 **do**
6 Ota järjestyksessä seuraava kaari $u \overset{w}{—} v$
7 **if** find(u) $\neq$ find(v) **then**
8 $A \leftarrow A \cup \left\{u \overset{w}{—} v\right\}$
9 union(find(u), find(v))
10 **return** A

- Union-find-rakenne tarvitsee (solmu)tietueeseen x kaksi attribuuttia:

  Viite $p[x]$ sen edustajaan:

  – Jos $p[x] = x$, niin x on yhä itse oman joukkonsa S edustaja.

  – Muuten $p[x] =$ se tietue y

    * jonka hyväksi x luopui edustuksestaan

    * silloin kun tietueen x edustamaa joukkoa yhdistettiin

    * sellaiseen joukkoon, joka edustaja silloin oli y.

  Pituus rank$[x]$ pisimmälle p-viiteketjulle

$$\bigcirc \underbrace{\xrightarrow{p} \bigcirc \xrightarrow{p} \bigcirc \xrightarrow{p} \cdots \xrightarrow{p}}_{\text{rank}[x] \text{ viitettä}} x$$

  joka täytyy kulkea, kun halutaan löytää tämä *edustaja*tietue x.

- Seuraavassa kuvassa on esimerkkijoukot

$$\{a, b, c, d, e, f, g\}, \{h, i, j, k\} \text{ ja } \{l\}$$

joiden

edustajaviitteet p on esitetty nuolilla

edustajien pituusattribuutit ovat

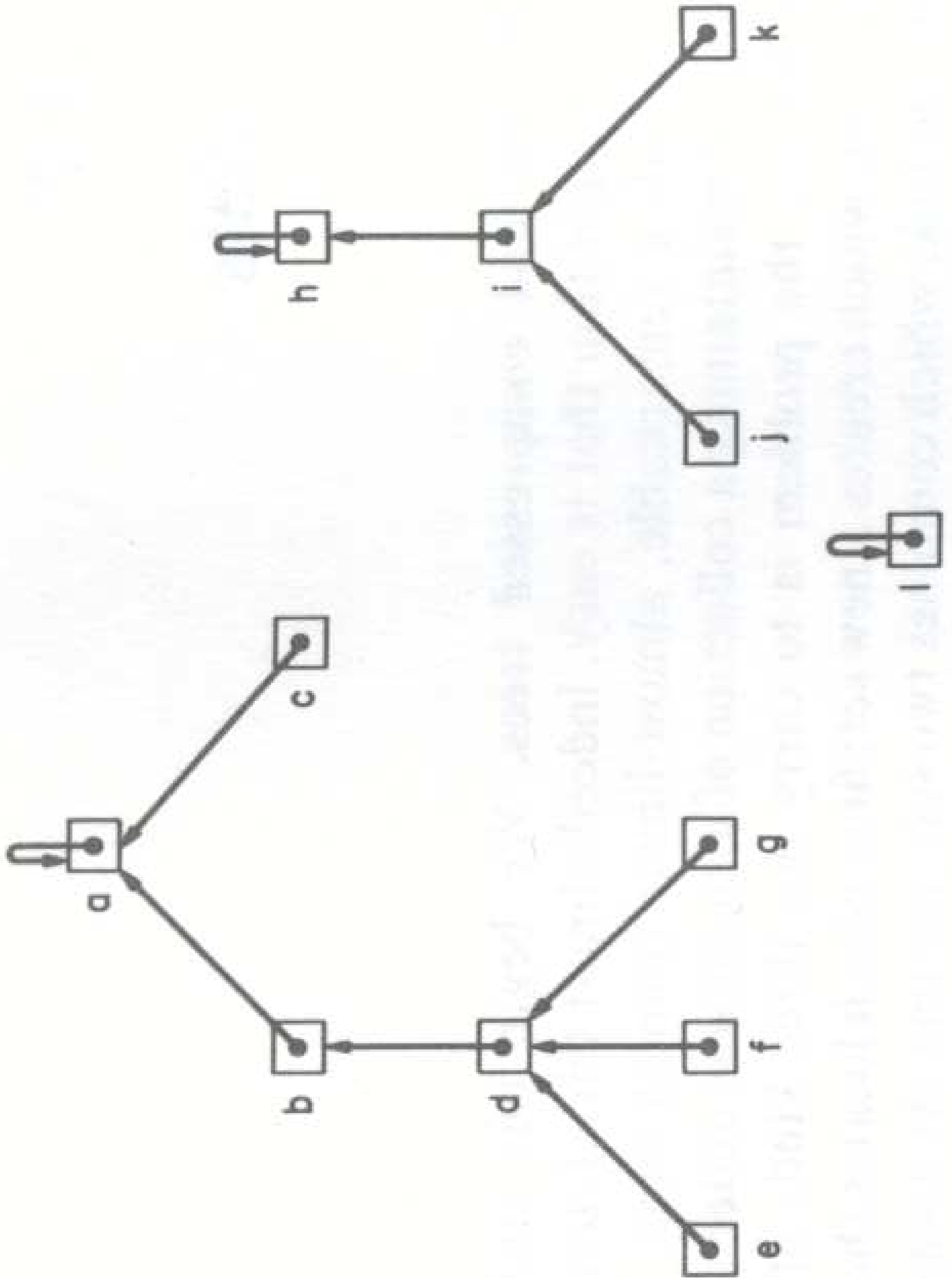
$$\text{rank}[a] = 3$$
$$\text{rank}[h] = 2$$
$$\text{rank}[l] = 0.$$

- Tietueen alustus on

makeSet(x)
  $p[x] \leftarrow x$
  rank[x] $\leftarrow 0$

kuten kuvan tietueella l.

(Kuva 2.1 kirjasta R.E. Tarjan: *Data Structures and Network Algorithms.* SIAM, 1983.)


a
b
c
d
e
f
g
h
i
j
k
l

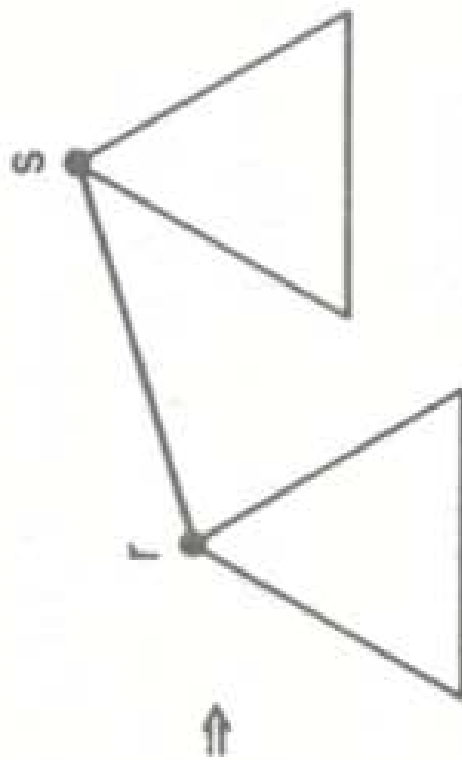
- Ensimmäinen tehostusidea on *yhdistäminen p-ketjujen pituutta minimoiden* (''Union by Rank''):

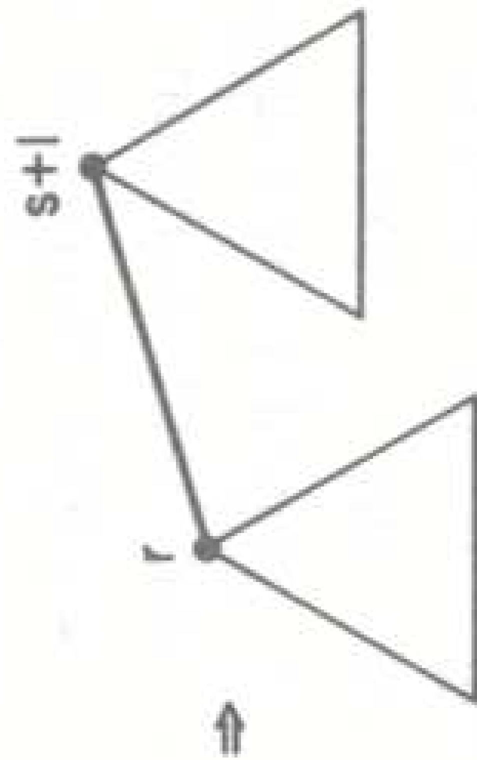
union(x, y)
  if rank[x] $<$ rank[y] **then**
    $p[x] \leftarrow y$
  else if rank[x] $>$ rank[y] **then**
    $p[y] \leftarrow x$
  else
    $p[x] \leftarrow y$
    rank[y] $\leftarrow$ rank[y] $+ 1$

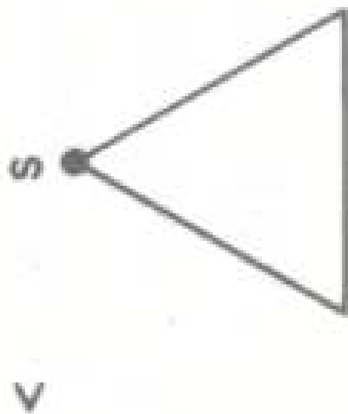
- Eripituiset yhdistetään (seuraava kuva (a))
  - pienempi suurempaan
  - jonka pituus ei muutu.

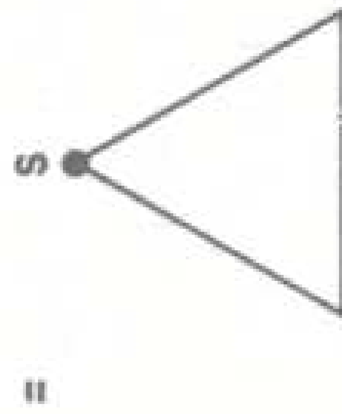
- Samanpituiset yhdistetään (seuraava kuva (b))
  - toinen toiseen
  - jonka pituus kasvaa yhdellä.

(Kuva 2.3 kirjasta R.E. Tarjan: *Data Structures and Network Algorithms.* SIAM, 1983.)









- Edellisestä kuvasta voidaan nähdä, että joukon

$$2^{\text{rank[edustaja]}} \leq \text{tietueiden lukumäärä.}$$

- Olemme tehostaneet Kruskalin algoritmia:
  - Jokainen operaatio find(z) kulkee tietueen z p-ketjua, kunnes tavoitetaan sen edustaja.
  - Siis ylläolevan nojalla kunkin operaation aikavaativuus on

    $$\mathcal{O}\left(\log |V|\right).$$

  - Siis rivit 7 ja 9 voidaan toteuttaa samassa ajassa.
  - Siis myös rivien 5–9 silmukka voidaan toteuttaa samassa ajassa

    $$\mathcal{O}\left(|E| \cdot \log |V|\right)$$

    kuin rivin 4 järjestäminen.

- Mutta voimme jatkaa yhä tehostamista toisella idealla:

  tiivistetään ketju samalla kun se kuljetaan (''Path Compression'').

- Rekursiivinen muotoilu on

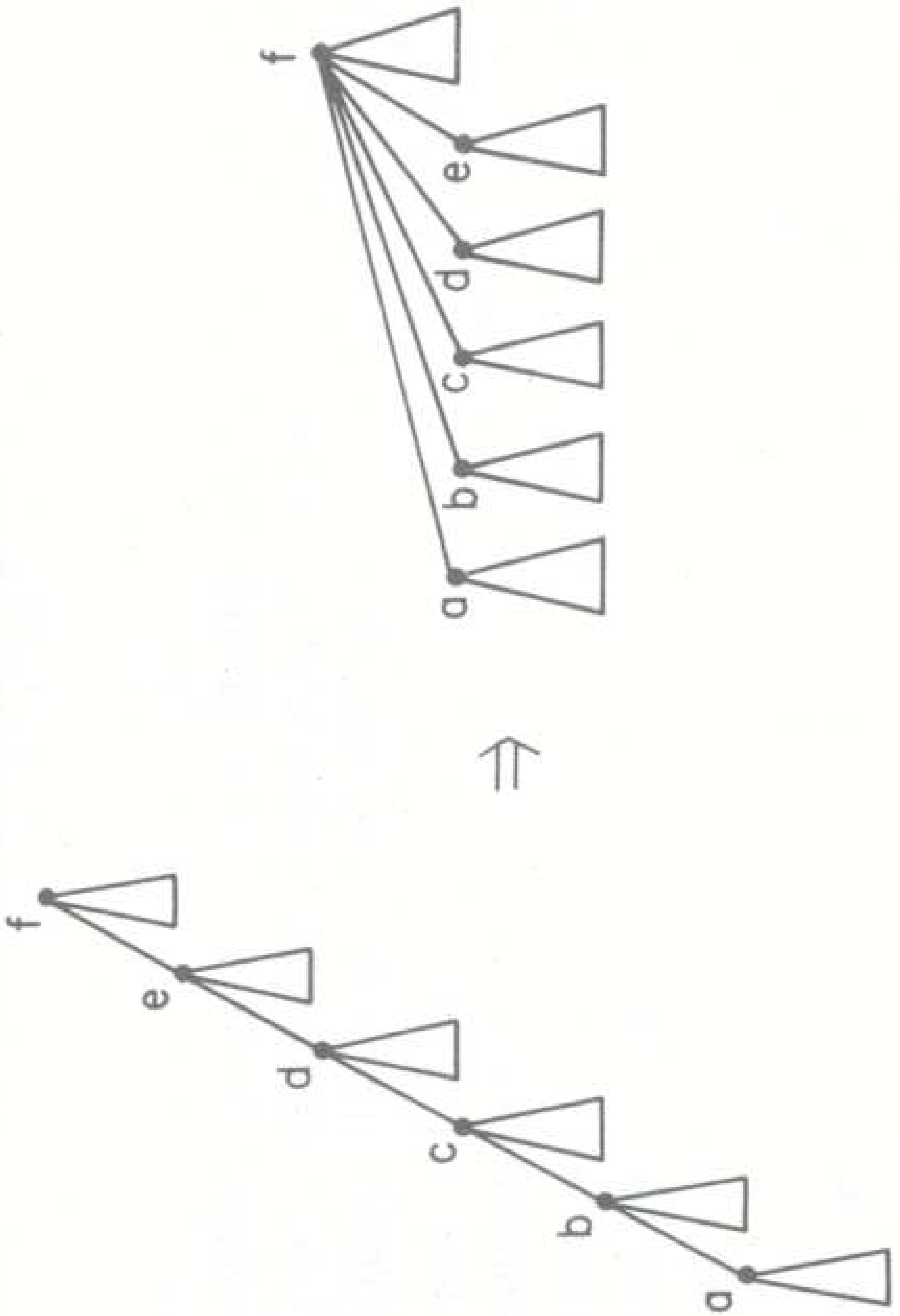
  find(x)
  if $x \neq p[x]$ **then**
  $p[x] \leftarrow$ find($p[x]$)
  return $p[x]$

  eli ''jos tämä x ei itse ole edustaja, niin päivitetään sen $p[x]$ osoittamaan *suoraan* sen edustajaan y, kunhan y ensin löydetään p-ketjun päästä''.

- Iteratiivinen muotoilu kulkee p-ketjun kahdesti:

  1. löytääkseen edustajan y
  2. päivittääkseen kuljetun ketjun viitteet suoraan löydettyyn y.

(Kuva 2.2 kirjasta R.E. Tarjan: *Data Structures and Network Algorithms.* SIAM, 1983.)


f
e
d
c
b
a
f
e
d
c
b
a

- Edellinen kuva esittää operaation find(a) vaikutuksen:

  1. rekursio ketjua pitkin löytää lopulta sen edustajan f

  2. rekursiosta palatessa päivitetään kuljetun ketjun viitteet

     $p[e], p[d], p[c], p[b]$ ja lopulta myös $p[a]$

     osoittamaan suoraan löydettyyn edustajaan f.

- Kenttää rank[f] ei muuteta:

  – todellisten p-ketjupituuksien laskeminen olisi vaivalloista

  – Kenttä tarkoittaakin nyt siis

    ”kuinka pitkä pisin ketju oikein olisikaan, ellemme tiivistäisi niitä”.

- Aikavaativuusanalyysi on yksityiskohdiltaan hyvin tekninen

  ja sivuutetaan.

- Yksi operaatio find(a) näkee lisävaivaa
  - joka nopeuttaa muita operaatioita

    $$\text{find}(b), \text{find}(c), \text{find}(d)$$

  - joten on reilua jakaa lisävaiva kaikkien hyötyjien kesken

    (samoin kuin kalvoilla 4.4).

- Lopputulos: Kun tehdään
  - yhteensä m operaatiota, joista
  - $n \leq m$ on tietueen luonteja, ja
  - $\leq n - 1$ on yhdistämisiä

  niin kokonaisuudessaan aikaa kuluu

  $$\mathcal{O}\left(m \cdot \alpha(m, n)\right)$$

  missä funktio α kasvaa, mutta *hyvin* hitaasti.

- Union-find-rakenne onkin *käytännössä vakioaikainen:*
  - Tietueiden lukumäärä $n =$ tarvittavien muistipaikkojen lukumäärä.
  - Yhden operaation osuus koko työstä on

    $$\alpha(m, n) \leq 4$$

    kaikilla käytännössä mahdollisilla n.
  - Esimerkiksi

    $$\alpha(m, n) \geq 6$$

    vasta kun n lähestyy tunnetun maailmankaikkeuden kaikkien alkeishiukkasten lukumäärää...

- Kruskalin algoritmissa on siis
  1. kaarten järjestämisvaihe (rivi 4)

     joka on se aikaa vievin osuus.
  2. jälkikäsittelyvaihe (rivit 5–9)

     joka on käytännössä lineaarinen.

7.6.3 Kruskal ja keko

- Kruskalin algoritmissa (kalvot 7.6.1)

  1. ensin järjestetään syöteverkon kaaret kasvavaan järjestykseen painon mukaan

  2. sitten jälkikäsitellään niitä siinä järjestyksessä

     kunnes pienin virittävä puu A on koossa.

- Jälkikäsittelyvaihe voi siis päättyä jo *ennen* kuin kaikki järjestetyt kaaret on käsitelty.

- Silloin on luontevaa hyödyntää kekojärjestämisen (kalvot 6.1) inkrementaalisuutta (kalvot 6.6–6.7):

  1. Tehdään kaarista minimikeko H painon suhteen.

  2. Otetaan kaaret jälkikäsittelyyn tästä keosta H.

KruskalUnionFindWithHeap(G)

```
1   A ← ∅
2   for kaikille solmuille v ∈ V do
3       makeSet(v)
4   H ← buildHeap(E)
5   while eri palojen lukumäärä > 1 do
6       u —w— v ← heapDelMin(H)
7       if find(u) ≠ find(v) then
8           A ← A ∪ {u —w— v}
9           union(find(u), find(v))
10  return A
```

- Hyöty aikaisempaan versioon verrattuna:
  - Alustusrivi 4 vie vain $\mathcal{O}(|E|)$ askelta.
  - Rivillä 6 tehdään kekojärjestämistä rinnan Kruskalin kanssa

    jolloin ei aina tarvitsekaan järjestää kaikkia kaaria kokonaan.

- Tästä alkaen käytetään abstraktia tietotyyppiä "keko"

  (aiemmin käytettiin sen konkreettista taulukkototeutusta).

7.6.4 Esimerkki: Ryvästys

- Syötteenä annetaan tason pisteet

$$\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \langle x_3, y_3 \rangle, \ldots, \langle x_n, y_n \rangle.$$

- Ne pitää jakaa kahteen eri luokkaan A ja B siten, että nämä luokat ovat mahdollisimman kaukana toisistaan.

- Tarkemmin sanoen: siten, että lyhyinkin etäisyys luokan A pisteestä luokan B pisteeseen on mahdollisimman suuri.

- Tämä on eräs tapaus *ryvästyksestä* (clustering).

- Siinä Kruskalin algoritmi (kalvoilta 7.6.1) osoittautuu juuri sopivaksi.

(Esimerkki on peräisin kirjan J. Kleinberg, E. Tardos: *Algorithm Design.* Addison-Wesley, 2005 luvusta 4.7.)

Algoritmi:

1. Muodosta suuntaamaton verkko G, jonka

   solmuina ovat annetut pisteet

   kaarina $p \overset{d_{pq}}{\text{---}} q$ ovat kaikki pisteparit $p \neq q$

   painoina d_{pq} ovat pisteiden p ja q välinen etäisyys.

2. Suorita Kruskalin algoritmia, mutta lopeta jo kun *eri palojen lukumäärä* $= 2$ (rivillä 5).

Perustelu: Jokaisella silmukkakierroksella 2 huolehditaan, ettei ainakaan näiden pisteiden p ja q välinen etäisyys d_{pq} ole se pienin, joka erottaa luokat A ja B toisistaan.

7.6.5 Primin algoritmi

- Myös Primin algoritmi käyttää ahnetta strategiaa.

- Ero Kruskaliin (kalvot 7.6.1):
  - Algoritmin suorituksen aikana on olemassa *vain yksi* palanen P
  - jota kasvatetaan vähitellen kokonaiseksi virittäväksi puuksi.

- Syötteet ovat
  - suuntaamaton verkko G
  - sen kaaripainot määrittelevä funktio w
  - jokin sen solmu r, josta virittävän puun muodostaminen aloitetaan.

- Solmuun v liittyy kaksi attribuuttia:

  avain key$[v]$ = pienimmän sellaisen kaaren

  $$u \overset{c}{\text{——}} v$$

  paino c, joka liittää vielä nyt ulkopuolella olevan solmun v palaseen P

  vanhempi $p[v]$ = tämä liitossolmu u palasen P sisäpuolella.

- Palasen P ulkopuoliset v pidetään minimikeossa H (kalvoilta 5) järjestettynä avainkentän key$[v]$ mukaan.

- Lopuksi virittävän puun kaaret ovat

  $$A = \{(v, p[v]) : v \in V \setminus \{r\}\}.$$

  – Algoritmissa ei siis muodosteta eksplisiittisesti virittävän puun kaarijoukkoa A

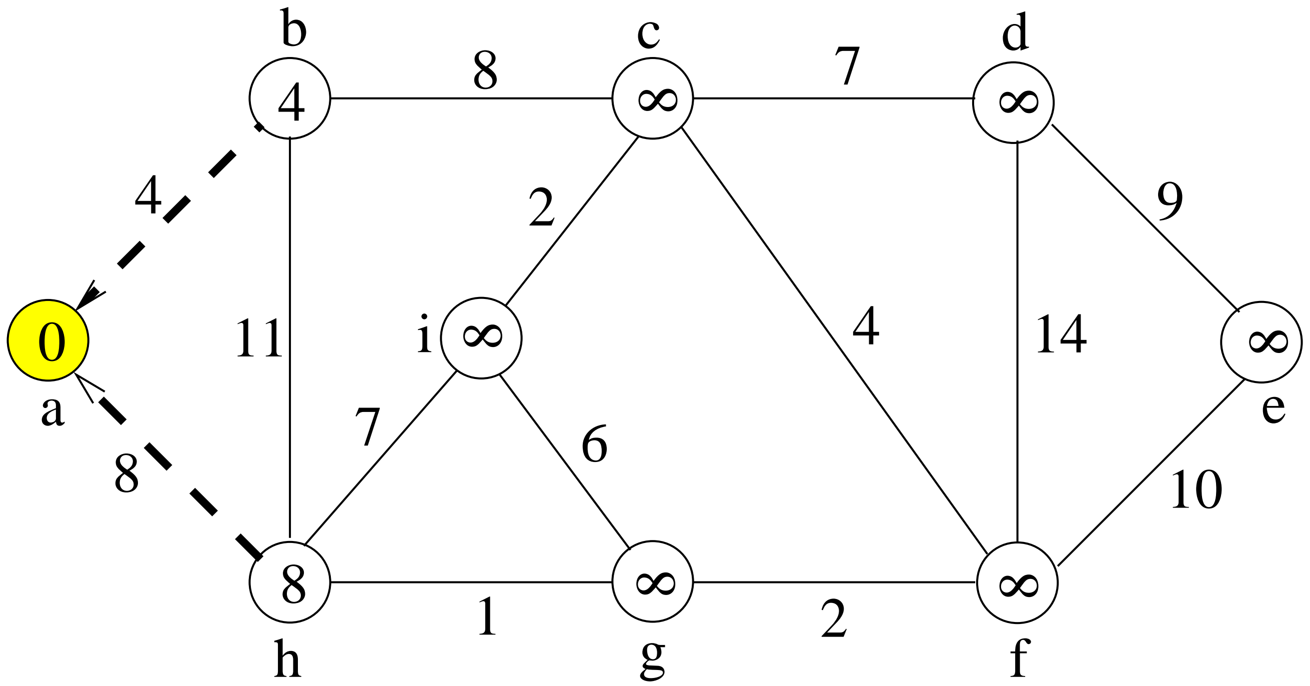
  – vaan ylläpidetään sitä solmuattribuuteissa p.

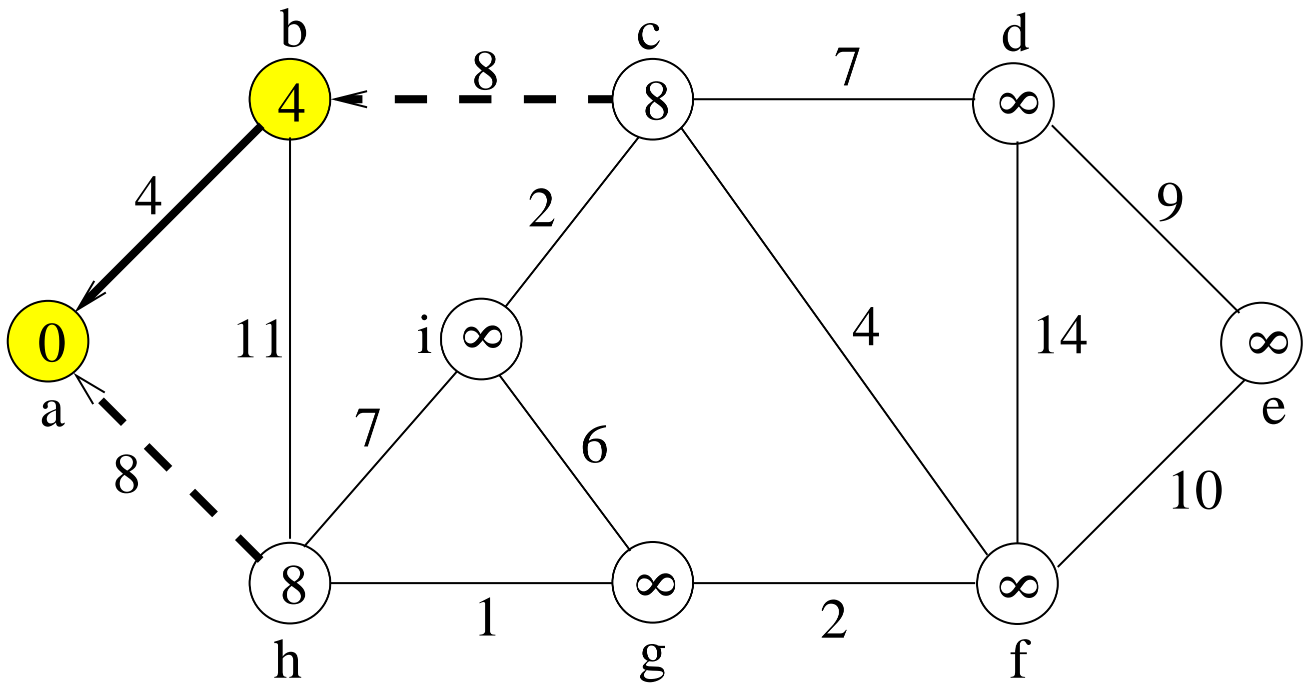
```
Prim(G, w, r)
1   for kaikille solmuille v ∈ V do
2       key[v] ← ∞
3       p[v] ← NIL
4   key[r] ← 0
6   for kaikille solmuille v ∈ V do
7       heapInsert(H, v, key[v])
8   while not empty(H) do
9       u ← heapDelMin(H)
11      for jokaiselle solmulle v ∈ Adj[u] do
12          if solmu v on vielä keossa H and
                    key[v] > w(u, v) then
13              key[v] ← w(u, v)
14              p[v] ← u
15              heapDecreaseKey(H, v, key[v])
```

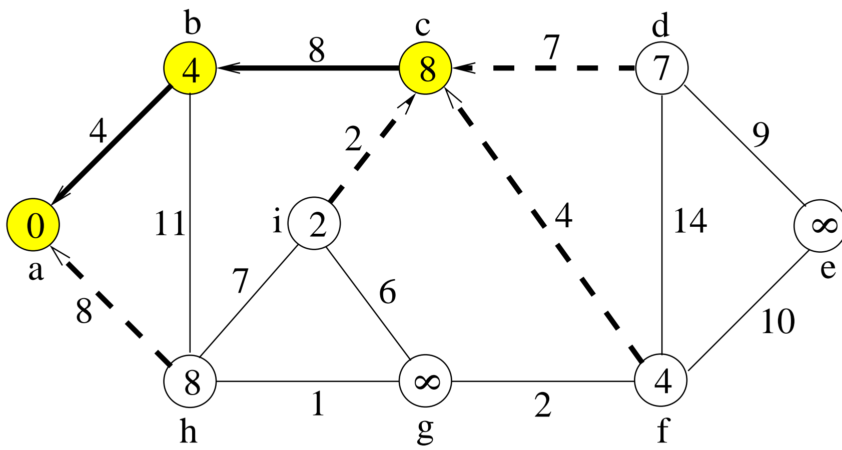
- Vertaa kalvojen 7.4 Dijkstran algoritmiin:
  - Rivit on numeroitu molemmissa samoin.
  - Rivillä 13 Prim ei laskekaan koko polun pituutta $d[v]$, vaan ainoastaan liittävän kaaren.
  - Rivin 12 **if**-lauseessa testataan myös, ettei solmua v ole vielä käsitelty, jotta ei synnytetä kehää.

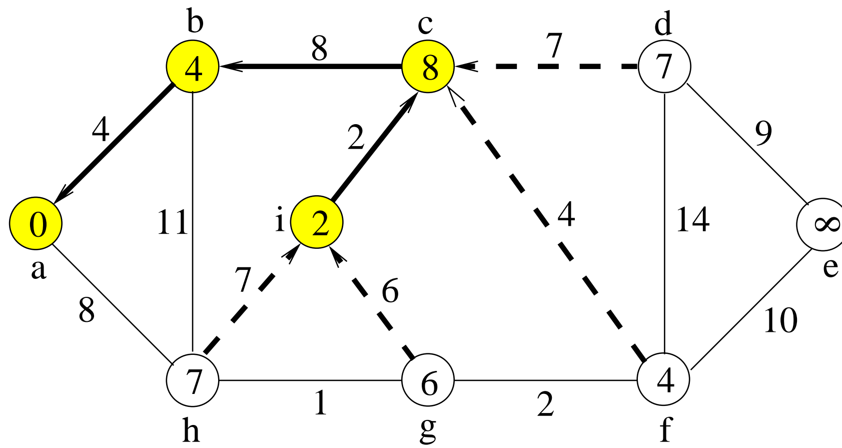
- Primin algoritmin toimintaidea:
  - Aluksi puu P on tyhjä, ja asetetaan kaikille muille solmuille kuin r avaimeksi ääretön (alustusrivit 1–6).
  - Pääsilmukkarivien 7–13 ensimmäisessä toistossa tulee valituksi solmu r, joka siis poistuessaan keosta H implisiittisesti liittyy puuhun P.
  - Aina kun virittävään puuhun P on näin liitetty solmu u, pidetään voimassa invarianttia:

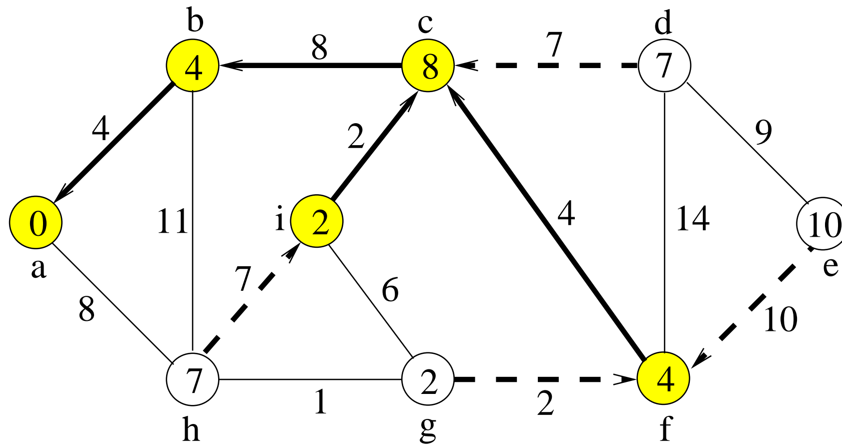
    Puun P ulkopuolella olevan solmun v avainkenttänä on key$[v] =$ pienimmän sellaisen kaaren paino, joka liittäisi solmun v puuhun P.
  - Keosta H otetaan käsittelyyn aina sellainen solmu u, jonka avain key$[u]$, eli etäisyys virittävään puuhun P, on pienin, ja liitetään se virittävään puuhun P.
  - Seuraava kuvasarja on esimerkki algoritmin toiminnasta.

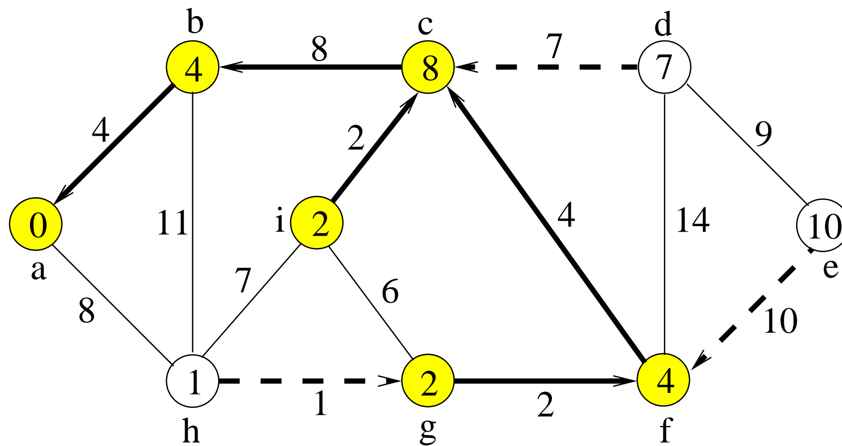

b
c
d
4
8
∞
7
∞
4
2
9
0
11
i
∞
4
14
∞
a
7
6
e
8
10
8
∞
∞
1
2
h
g
f

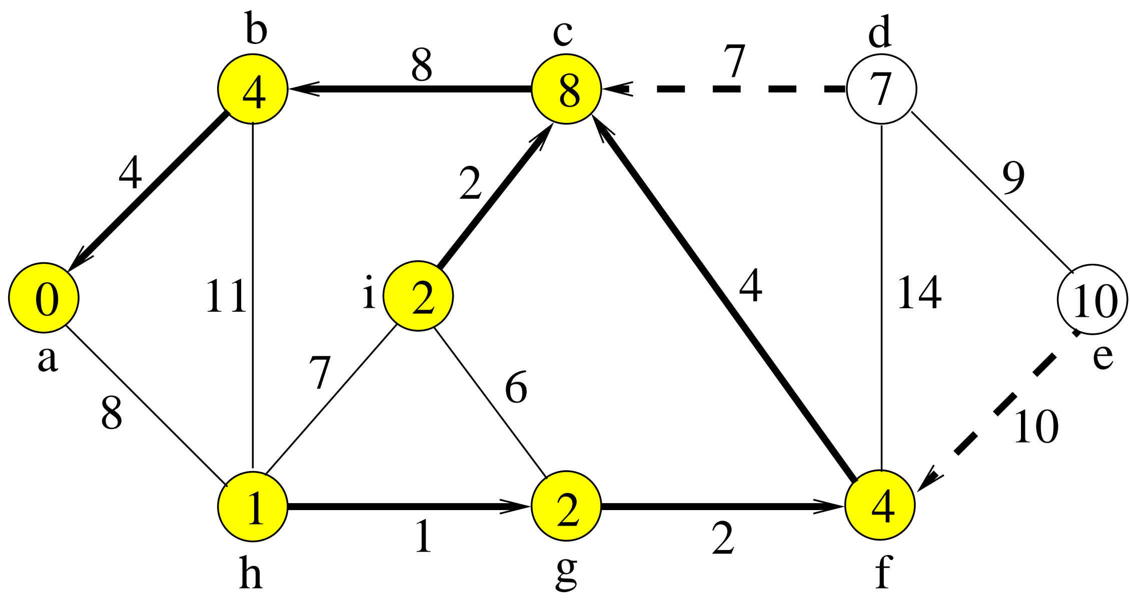

b
c
d
4
8
8
7
∞
4
2
9
0
11
i
∞
4
14
∞
a
7
6
e
8
10
8
∞
∞
1
2
h
g
f

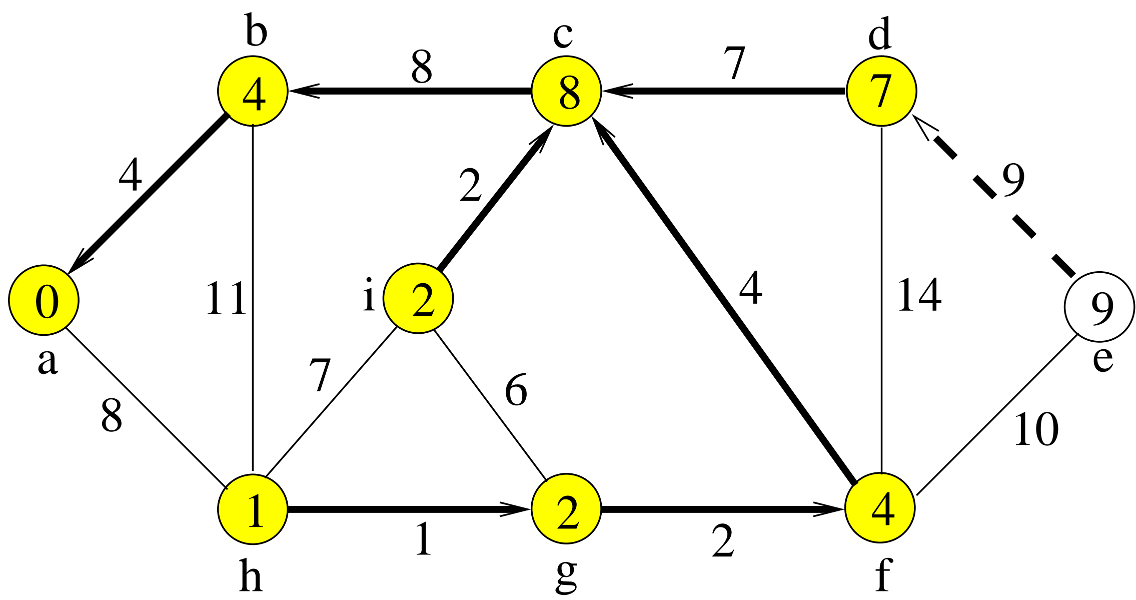

b c d
4 8 8 7 7
4 2 9
0 11 i 2 4 14 ∞
a e
8 7 6 10
8 ∞ 4
1 2
h g f

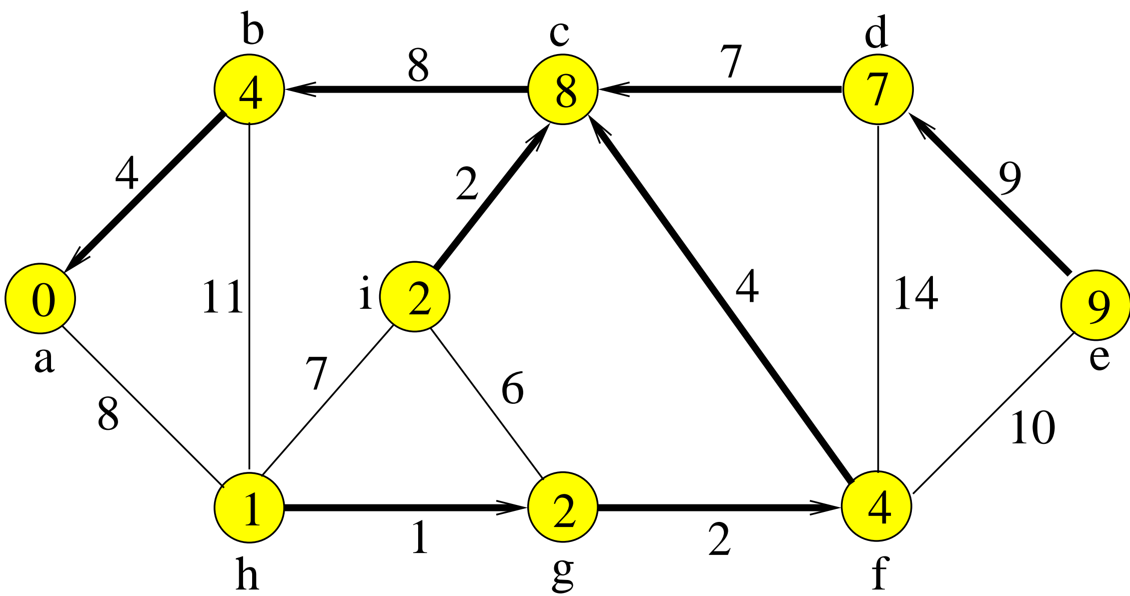

b c d
4 8 8 7 7
4 2 9
0 11 i 2 4 14 ∞
a e
8 7 6 10
7 6 4
1 2
h g f


b c d
4 8 8 7 7
4 2 9
0 11 i 2 4 14 10
a e
8 7 6 10
7 2 4
1 2
h g f


b c d
4 8 8 7 7
4 2 9
0 11 i 2 4 14 10
a e
8 7 6 10
1 2 4
1 2
h g f


b
c
d
8
7
4
8
7
4
2
9
0
11
i
2
4
14
10
a
e
7
6
8
10
1
2
4
1
2
h
g
f


b
c
d
8
7
4
8
7
4
2
9
0
11
i
2
4
14
9
a
e
7
6
8
10
1
2
4
1
2
h
g
f


b
c
d
8
7
4
8
7
4
2
9
0
11
i
2
4
14
9
a
e
7
6
8
10
1
2
4
1
2
h
g
f

Aikavaativuus:

- Rivien 1–6 alkutoimet voidaan tehdä ajassa $\mathcal{O}(|V|)$.
- Rivien 7–13 ulommassa toistolauseessa jokainen solmu poistetaan kertaalleen keosta. Aikaa tähän kuluu yhteensä

$$\mathcal{O}(|V| \cdot \log |V|) .$$

- Rivien 9–13 sisempi toistolause suoritetaan kahdesti jokaiselle kaarelle (kerran kumpaankin suuntaan), eli

$$\mathcal{O}(|E|)$$

  kertaa.
- Sisemmässä toistolauseessa suoritetaan korkeintaan kertaalleen $\mathcal{O}(\log |V|)$ operaatio heapDecreaseKey

  (johon tarvitaan kalvojen 5.4 kahvoja).
- Saamme yhteensä

$$\mathcal{O}(|E| \cdot \log |V|)$$

  eli saman kuin Kruskalin algoritmilla.

Tilavaativuus: Aputietorakenteena on keko, ja se sisältää aluksi kaikki solmut, eli sen tilavaativuus on

$$\mathcal{O}(|V|).$$

7.6.6 Kruskal vai Prim?

- Algoritmit näyttävät $\mathcal{O}$-analyysin valossa yhtä hyviltä.

- Käytännössä Prim on yleensä parempi:

  - Keon osuudesta Primin ajankäyttöön voidaan päätellä samat asiat kuin Dijkstran (kalvoilla 7.4).

  - Kruskalin ajankäyttöä hallitsee järjestäminen

    jonka alarajatodistusta kalvoilta 6.4 voi yleistää myös keskimääräiseen tapaukseen.

- Mutta Kruskalin algoritmi muuttuukin nopeammaksi, jos

  - joku muu antaakin kaaret valmiiksi järjestyksessä

  - kaaret voidaankin järjestää yleistä alarajaa nopeammin.

7.6.7 Esimerkki: Laboratoriohiiri

- Tutkijat ovat panneet hiiren juoksemaan sille tutun labyrintin läpi tarkkaillakseen sen oppimiskykyä.

- Tutkijat näkevät joihinkin kohtiin labyrintistä

  mutta jotkut paikat ovat heidän katseiltaan piilossa.

- Hiirtä alkaa tällainen yksityisyyden loukkaus ärsyttää.

  Niinpä se haluaa suunnitella labyrintin läpi sellaisen reitin, jossa kahden peräkkäisen piilopaikan välinen julkinen matka on aina mahdollisimman lyhyt.

  – Tämä *ei* ole sama kuin lyhyin reitti, joka saattaa olla vaikka koko ajan näkyvillä.

  – ”Parempi virsta väärää kuin vaaksa vaaraa.”

f

s

- Ajatellaan seuraavaa suuntaamatonta verkkoa G_1:
  - Solmut ovat
    * aloituspaikka s
    * lopetuspaikka f
    * jokainen piilopaikka.
  - Kaari

    $$p \overset{c}{—} q$$

    tarkoittaa
    * "paikasta p pääsee paikkaan q (ja päinvastoin) kokonaan näkyvissä kulkevaa reittiä pitkin, ja
    * lyhyimmän sellaisen reitin pituus on c ruutua".

- Oletetaan verkko G_1 yhtenäiseksi.

  (Muuten labyrintissä on paikkoja, joihin hiiri ei pääse.)

Väite: Hiiren kannattaa

- muodostaa verkon G_1 pienin virittävä puu T_1
- kulkea solmusta s solmuun f suorinta sellaista polkua

  $$\mathcal{P}_1 = s \leadsto f$$

  joka kulkee pitkin tätä puuta T_1.

Perustelu: Olkoon

$$\mathcal{P}' = s \leadsto f$$

jokin vielä varjoisampi polku kuin $\mathcal{P}_1$.

Poistetaan polulta $\mathcal{P}_1$ sellainen kaari

$$p \overset{c}{\text{---}} q$$

jonka paino c on suurin. Silloin puu T_1 hajoaa kahdeksi palaksi, joista toisessa on s ja toisessa on f.

Perustelu jatkuu... Koska polku $\mathcal{P}'$ yhdistää nämä solmut, niin sillä on jokin kaari

$$u \overset{d}{—} v$$

joka yhdistää nämä palat.

Koska $\mathcal{P}'$ on aidosti varjoisampi kuin $\mathcal{P}_1$, niin polun $\mathcal{P}'$ jokainen kaaripaino on aidosti pienempi kuin polun $\mathcal{P}_1$ suurin kaaripaino. Erityisesti siis $d < c$.

Jos nyt puussa T_1 kaari

$$p \overset{c}{—} q$$

korvataan kaarella

$$u \overset{d}{—} v$$

niin saadaan toinen virittävä puu T' verkolle G_1, ja näin saadun puun T' kaarien yhteispaino on pienempi kuin puun T_1.

Mutta tämähän olisi ristiriidassa sen kanssa, että puu T_1 on minimaalinen. Siis polkua $\mathcal{P}'$ ei voi olla olemassa. □

- Saadaan seuraava algoritmi:

  1. Muodosta labyrintista verkko G_1.
  2. Suorita verkossa G_1 Primin algoritmia (kalvoilta 7.6.5)
     - lähtien solmusta $r = s$
     - kunnes keosta nostetaan solmu f.
  3. Vastaa näin laskettu verkon G_1 polku

     $$s \xleftarrow{p} \bigcirc \xleftarrow{p} \bigcirc \xleftarrow{p} \cdots \xleftarrow{p} f$$

     siten, että jokainen kaari

     $$\bigcirc \xleftarrow{p} \bigcirc$$

     laajennetaan vastaavaksi näkyvissä kulkevaksi lyhyimmäksi reitiksi.

- Perustoteutus olisi

  1. laske labyrintin kaikkien ruutujen välimatkataulukko Floydin algoritmilla (kalvoilta 7.4.3)
  2. rajoitu taulukossa verkon G_1 solmuihin, kun suoritat Primin algoritmia.

- Voimme myös upottaa vaiheen 1 vaiheeseen 2:

  vasta kun (jos) Primin algoritmi tarvitsee suojaisan labyrinttiruudun p vieruslistaa Adj[p], niin lasketaan se kalvojen 7.3.1 leveyssuuntaisella läpikäynnillä.

- Kutsutaan labyrinttiruutua u

  verkkopaikaksi jos se on verkon G_1 solmu

  labyrinttipaikaksi jos ei.

- Yleistetään labyrinttipaikan u attribuutteja siten, että eri leveysläpikäynnit eivät lue vahingossa toistensa attribuutteja.

  – colour[u] = se verkkopaikka, josta tämä läpikäynti lähti

    jotta attribuutteja ei tarvitse alustaa uudelleen

  – $p[u]$ = varataan eri läpikäynneille eri muistipaikat

    jotta saadaan vaiheen 3 reitti.

```
Alusta colour[u] = NIL kaikilla
    labyrinttipaikoilla u
Alusta minimikeko H kaikilla verkkopaikoilla
    lähtöpaikan avaimena d[s] = 0
    mutta muilla d[u] = +∞
while heapMin(H) ≠ f do
    u ←heapDelMin(H)
    d[u] ← 0
    enqueue(Q, u)
    while not empty(Q) do
        v ← dequeue(Q)
        for jokainen paikan v vieruspaikka t
            do if t on labyrinttipaikka then
                if colour[t] ≠ u then
                    d[t] ← d[v] + 1
                    p[t] ← new p[v]
                    colour[t] ← u
                    enqueue(Q, t)
            else if t on keossa H avaimenaan
                    d[t] > d[v] + 1 then
                d[t] ← d[v] + 1
                p[t] ← u
                heapDecreaseKey(H, t, d[t])
```

return polku $s \xleftarrow{p} \bigcirc \xleftarrow{p} \bigcirc \xleftarrow{p} \cdots \xleftarrow{p} f$

- Perustoteutus toimisi ajassa

$$\mathcal{O}(\underbrace{N^3}_{\text{Floyd}} + \underbrace{K^2 \cdot \log K}_{\text{Prim}})$$

missä

$$N = \text{labyrintin ruutujen}$$
$$K = \text{verkon } G_1 \text{ solmujen}$$

lukumäärä.

- Vaihtoehtoinen toteutuksemme taas ajassa

$$\mathcal{O}(N \cdot \text{Prim}).$$

- Käytännössä vaihtoehtoinen lienee parempi, koska
  - K lienee paljonkin pienempi kuin N
  - algoritmi pysähtyy heti päästessään solmuun f
  - Primin algoritmi toimii usein pahinta aikavaatimustaan nopeammin
  - leveysläpikäynti voi pysähtyä jo ennen kaikkia N ruutua.

7.6.8 Oikeellisuudesta

- Palataan käsittelemään

  Kruskalin (kalvoilta 7.6.1)

  Primin (kalvoilta 7.6.5)

  algoritmien oikeellisuutta.

- Molemmat etenevät ahneella strategialla.

- Kalvojen 7.6.7 väite perusteltiin ahneille algoritmeille tyypillisellä *vaihtoargumentilla:*

  - Vaihdettiin pienimmän virittävän puun yksi kaari toiseksi

  - jolloin saatiin muka vielä pienempi virittävä puu

  - ja päädyttiin näin ristiriitaan.

- Samanlaisella argumentoinnilla voidaan perustella myös seuraava yleisempi tulos.

Lause 7.2. *Oletetaan suuntaamattomasta yhtenäisestä verkosta G, että sen*

1. *kaikki kaaripainot ovat keskenään erisuuria*

2. *solmut on jaettu kahteen osajoukkoon*

   $$S \text{ ja } U = V \setminus S$$

3. *kaari*

   $$e = v \overset{c}{\text{———}} w$$

   kulkee osajoukosta toiseen, eli

   $$v \in S \text{ mutta } w \in U$$

4. *kaari e on painoltaan c pienin kaiksta tällaisista osajoukkojen S ja U välisistä kaarista.*

Silloin kaari e kuuluu jokaiseen verkon G pienimpään virittävään puuhun T.

Todistus: Olkoon T verkon G sellainen virittävä puu, joka ei sisällä kaarta e. Osoitetaan, että verkolla G on toinen virittävä puu T', jonka kaaripainojen summa on vielä pienempi. Silloin T ei siis olekaan pienin virittävä puu.

- Puussa T on polku $\mathcal{P} = v \rightsquigarrow w$

  koska T on virittävä puu verkolle G.

- Lähdetään kulkemaan tätä polkua $\mathcal{P}$ solmusta $v \in S$ alkaen.

- Jossakin vaiheessa kohdataan polulta $\mathcal{P}$ ensimmäinen kaari

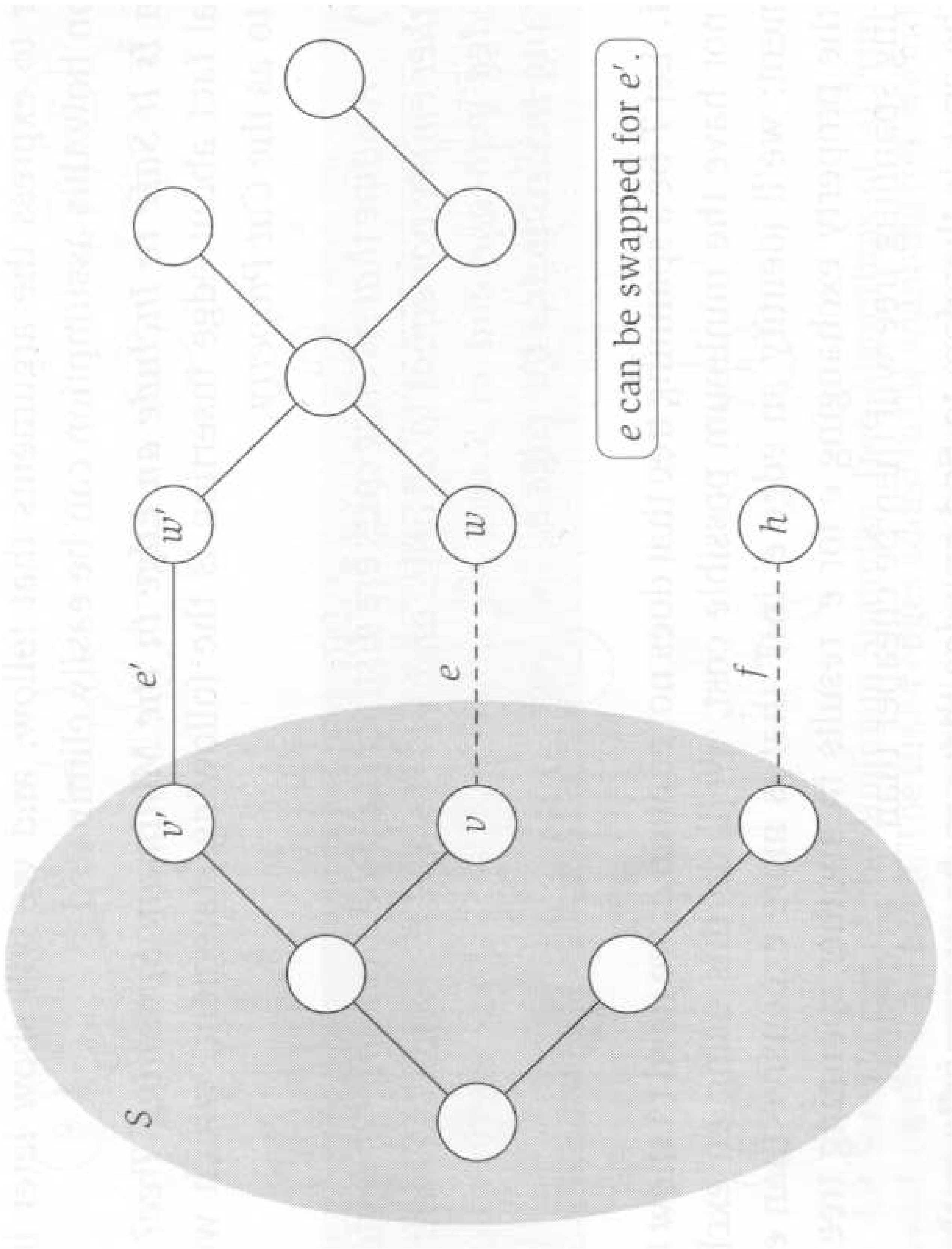
$$e' = v' \overset{c'}{\text{———}} w'$$

  joka kulkee osajoukosta toiseen, eli

$$v' \in S \text{ mutta } w' \in U$$

- Puu T' saadaan vaihtamalla kaaret e ja e' keskenään. □

(Kuva 4.10 kirjasta J. Kleinberg, E. Tardos: *Algorithm Design.* Addison-Wesley, 2005.)


S
v'
w'
e'
v
w
e
f
h
e can be swapped for e'.

Kruskalin algoritmi voidaan nyt osoittaa oikeaksi seuraavasti:

- Tarkastellaan sitä hetkeä, kun algoritmi päättää lisätä kaaren

$$e = v \overset{c}{\text{---}} w$$

  tulosjoukkoonsa A.

- Valitaan osajoukoksi $S = P[v]$, eli se palanen, johon solmu v kuuluu.

- Koska $P[w] \neq P[v]$, niin $w \notin S$.

- Siis voidaan soveltaa Lausetta 7.2.

- Siis kaaren e lisääminen tulosjoukkoon A on välttämätöntä.

- Algoritmin päättyessä A on puu:

  – Se on ohjelmoitu välttämään syklejä.

  – Jos A koostuisi lopussakin kahdesta (tai monesta) eri palasesta $P[v] \neq P[w]$, niin mihin olisi kadonnut se kaari e palasesta toiseen, jonka c on pienin?

Primin algoritmi voidaan nyt osoittaa oikeaksi vastaavalla järkeilyllä:

- Valitaan osajoukoksi S ne solmut, jotka on jo poistettu keosta H.
- Silloin seuraava kaari valitaan täsmälleen Lauseen 7.2 vaatimusten mukaisesti.
- Tulos A pysyy koko ajan puuna

  joka laajenee kattamaan koko verkon G.

Oletus 1 erisuurista kaaripainoista ei ole välttämätön:

- samanlaiset johtopäätökset voitaisiin tehdä ilmankin
- mutta monimutkaisemmin.