TKT20001 Tietorakenteet ja algoritmit

Syksy 2017

Jyrki Kivinen

Sisällys

1. Johdanto ja kertausta: tietoa kurssista, matematiikan kertausta, rekursio
2. Johdanto tietorakenteisiin: algoritmien oikeellisuus, vaativuus
3. Järjestäminen
4. Perustietorakenteita: pino, jono ja lista, niiden toteutus ja käyttäminen
5. Hajautus: toinen tehokas tapa suurten tietomäärien organisoimiseen
6. Hakupuut: eräs tehokas tapa suurten tietomäärien organisoimiseen
7. Keko: monien algoritmien käyttämä hyödyllinen aputietorakenne, mm. kekojärjestäminen
8. Verkot: tallettaminen, perusalgoritmit, polunetsintä, virittävät puut
9. Algoritminsuunnittelumenetelmiä ja kertausta

1. Johdanto ja kertausta

Kurssin asema opetusohjelmassa

- Tietojenkäsittelytieteen aineopintokurssi, 10 op, pääaineopiskelijoille pakollinen
- Esitietoina Johdatus yliopistomatematiikkaan (ennen: Johdatus diskreettiin matematiikkaan) ja Ohjelmoinnin jatkokurssi
- Jatkokurssi Design and Analysis of Algorithms (5 op, syventävät, pakollinen algoritmien, data-analytiikan ja koneoppimisen linjalla, valinnainen muille)
- Tällä kurssilla opittavat tekniikat ja ajattelutavat ovat tarpeen kurssilla Laskennan mallit (8 op, pääaineopiskelijoille pakollinen aineopintokurssi)

Esitietovaatimuksisista

Ohjelmointitaito Pääpaino on yleisillä periaatteilla, jotka esitetään käyttämällä luonnollista kieltä, kuvia, pseudokoodia, Java-ohjelmia jne.

- Yleiset periaatteet ovat riippumattomia ohjelmointikielestä, mutta on tärkeätä pystyä toteuttamaan algoritmit eri kielillä (tällä kurssilla Javalla)
- Koska tarkoituksena on oppia tietorakenteita ja perusalgoritmeja, kurssilla ei saa käyttää näiden valmiita Java-kirjastojen luokkia *ellei* erikseen pyydetä; toisaalta tarkoitus on keskittyä itse algoritmeihin, joten ohjelmissa ei tarvitse varautua virhetilanteisiin, jotka ovat olennaisia ”oikeissa” ohjelmissa
- Java-toteutusta harjoitellaan laskuharjoituksissa ja erikseen Tira-harjoitustyössä

Matematiikka Kurssilla tarvitaan vain vähän varsinaisia matemattisia tietoja (teoreemoja yms.). Diskreetit perusrakenteet kuten verkot (eli graafit) tulevat käyttöön, ja jonkin verran tarvitaan joukko-oppia ja laskutaitoa. Oleellisempaa kuitenkin on, että tietorakenteiden käsittely vaatii samantyyppistä ajattelua kuin (diskreetti) matematiikka

Lähdemateriaali

- Tämä luentomateriaali on uudistettu painos edellisvuosien kalvoista (tekijöinä mm. Patrik Floréen, Matti Luukkainen ja Matti Nykänen)

- Suurin osa käsiteltävistä asioista perustuu kirjaan

  T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms. 3rd ed. MIT Press, 2009.

  Myös kirjan toinen painos on käyttökelpoinen

- Cormenin kirja kattaa suurimman osan kurssilla käsiteltävistä asioista ja onkin suositeltavin teos, jos olet aikeissa ostaa jonkun kirjan

- Suurin ero Cormenin painosten 2 ja 3 välillä on pseudokoodin esitysmuodossa. Tässä monisteessa käytetään kolmannen painoksen pythonmaista pseudokoodia

- Kurssilla käsiteltävistä asioista Cormenissa ei ole AVL-puita, joihin hyvä lähde on

  M. A. Weiss: Data Structures and Algorithm Analysis in Java, 2nd ed., Addison-Wesley, 2007

- Muita lähteitä on mm.

  A. Levitin: Introduction to The Design and Analysis of Algorithms, Addison-Wesley, 2003

  J. Kleinberg, E. Tardos: Algorithm Design, Pearson Addison-Wesley, 2006

  A. Aho, J. Hopcroft, J. Ullman : Data Structures and Algorithms, Addison-Wesley, 1983

- Tässä mainittujen ohella aihepiiristä löytyy runsaat määrät kirjallisuutta ja vaihtelevatasoista materiaalia internetistä, erityisesti koska aihe on tietojenkäsittelytieteen perusasiaa ja siten globaalisti alan yliopistolaitosten kurssivalikoimassa

Miksi pakollinen kurssi Tietorakenteet ja algoritmit?

- Monet tietorakenteet on helppo toteuttaa Javan valmiiden tietorakennetoteutusten avulla. Esim. seuraavassa luvussa esitettävän puhelinluettelon toteuttamiseen Javassa on valmiina luokka `TreeMap`

  Miksi siis vaivautua opiskelemaan tietorakenteita 10 op:n verran?

- Vaikka tietorakennetoteutuksia on olemassa, ei niitä välttämättä osaa hyödyntää tehokkaasti ilman taustatietämystä niiden toimintaperiaatteista

  Esim. kannattaako valita puhelinluettelon toteutukseen `Arraylist`, `HashMap`, `TreeMap` vai jokin muu Javan kokoelmakehyksestä löytyvä tietorakennetoteutus

- Kaikiin ongelmiin ei löydy suoraan valmista tietorakennetta tai algoritmia ainakaan kielten standardikirjastojen joukosta

  Esim. seuraavassa luvussa esitettävä kaupunkien välisten lyhimpien reittien tehokas etsiminen ei onnistu Javan valmiiden mekanismien avulla

- Tarvitaan siis yleiskäsitys erilaisista tietorakenteista ja algoritmeista, jotta vastaantulevia ongelmia pystyy jäsentämään siinä määrin, että ratkaisun tekeminen onnistuu tai lisäinformaation löytyminen helpottuu

- Voisi myös kuvitella, että tietokoneiden jatkuva nopeutuminen vähentää edistyneiden tietorakenteiden ja algoritmien merkitystä

- Kuten lyhimpien polkujen esimerkistä sivulla 32–34 voidaan havaita, pienikin ongelma typerästi ratkaistuna on niin vaikea, ettei nopeinkaan kone siihen pysty

- Vaikka koneet nopeutuvat, käsiteltävät datamäärät kasvavat suhteessa jopa enemmän

- Koko ajan halutaan ratkaista yhä vaikeampia ongelmia (tekoäly, grafiikka, . . .)

- Paradoksaalisesti laitetehojen parantuminen saattaa jopa tehdä algoritmin tehokkuudesta tärkeämpää:

  algoritmien tehokkuuserot näkyvät yleensä selvästi vasta suurilla syötteillä, ja tehokkaat koneet mahdollistavat yhä suurempien syötteiden käsittelemisen

- Muuttuva ympäristö tuo myös uusia tehokkuushaasteita:
  - moniytimisyyden yleistymisen myötä syntynyt kasvava tarve rinnakkaistaa laskentaa
  - hajautettu laskenta
  - reaaliaikainen laskenta
  - mobiililaitteiden rajoitettu kapasitetti
  - muistihierarkiat
  - jne.
- Tällä kurssilla keskitytään kuitenkin klassisiin perusasioihin

Tietorakenteiden opiskelu on monella tavalla "opettavaista"

- Perustietorakenteet ovat niin keskeinen osa tietojenkäsittelytiedettä, että ne pitää tuntea pintaa syvemmältä
- Tietorakenteisiin liittyvät perusalgoritmit ovat opettavaisia esimerkkejä algoritmien suunnittelutekniikoista
- Tietorakenteita ja algoritmeja analysoidaan täsmällisesti: käytetään matematiikkaa ja samalla myös opitaan matematiikkaa
- Näistä taidoista hyötyä toisen vuoden kurssilla Laskennan mallit sekä kaikilla Algoritmit, data-analytiikka ja koneoppiminen -linjan kursseilla, joista erityisesti kurssia Design and Analysis of Algorithms voi pitää Tiran "jatkokurssina"
- Algoritmeihin liittyvästä täsmällisestä argumentoinnista on hyötyä tietysti myös normaalissa ohjelmoinnissa, testaamisessa ym.
- Teoreettisesta lähestymistavasta huolimatta kurssilla on tarkoitus pitää myös käytäntö mielessä:
  - miten opitut tietorakenteet ja algoritmit voidaan toteuttaa esim. Javalla?
  - mikä ohjelmointikielen tarjoamista valmiista tietorakenne- ja algoritmitoteutuksista kannattaa valita oman ongelman ratkaisuun?

Kurssin tavoitteet

- Yleisemmin voidaan todeta, että kurssilla etsitään vastauksia seuraavanlaisiin kysymyksiin:
  - miten laskennassa tarvittavat tiedot organisoidaan tehokkaasti?
  - miten varmistumme toteutuksen toimivuudesta?
  - miten analysoimme toteutuksen tehokkuutta?
  - millaisia tunnettuja tietorakenteita voimme hyödyntää?
- Osaamistavoitteet (ks. kurssisivu) määrittelee tarkemmin, mikä on oleellisinta kurssilla.
- Osa asioista (erityisesti algoritmien ja tietorakenteiden toteuttaminen Javalla) on sen luonteista, että niiden hallinta osoitetaan kokeiden sijasta vain harjoituksissa

Matematiikan kertausta: Logiikka

- Loogisten lausekkeiden totuusarvot määräytyvät seuraavasti (T niin kuin tosi eli true ja F niin kuin epätosi eli false):

A	B	$A \vee B$	$A \wedge B$	$A \Rightarrow B$	$A \Leftrightarrow B$
T	T	T	T	T	T
T	F	T	F	F	F
F	T	T	F	T	F
F	F	F	F	T	T

- Huomaa, että implikaatio $A \Rightarrow B$ siis tarkoittaa $\neg A \vee B$. Esimerkiksi väittämä ”jos kuu on juusto, niin Suomen valuutta on kruunu” on tosi implikaatio

Matematiikan kertausta: Todistusmenetelmiä

- Halutaan todistaa, että (todesta) oletuksesta A seuraa väite B eli $A \Rightarrow B$. Todistamiseen on useita erilaisia tekniikoita. Se, mikä kulloinkin sopii parhaiten, riippuu tilanteesta

- Deduktiivisessa todistuksessa, jota kutsutaan myös suoraksi todistukseksi, oletetaan A ja näytetään, että tästä seuraa B

  Esim. Olkoon $x \in \mathbb{N}$ pariton. Väite: x^2 on pariton.

  Tod.: Jos $x \in \mathbb{N}$ on pariton (oletus), niin $x = 2k+1$ jollain $k \in \mathbb{N}$ (parittomuuden määritelmä). Silloin $x^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1,$ eli x^2 on pariton. $\square$

- Kontrapositiivisessa todistuksessa näytetään että oletuksesta ei B seuraa ei A, eli että $\neg B \Rightarrow \neg A$

  Esim. Väite: Jos $x > 0$ on irrationaaliluku, niin $\sqrt{x}$ on irrationaaliluku.

  Tod.: Osoitetaan, että jos $\sqrt{x}$ on rationaaliluku, niin x on rationaaliluku. Siis $\sqrt{x} = p/q$, joillakin $p, q \in \mathbb{Z}, q \neq 0$, joten $x = \frac{p^2}{q^2}$ ja $p^2, q^2 \in \mathbb{Z}, q^2 \neq 0$. $\square$

- Vastaväitetodistuksessa (reductio ad absurdum) näytetään että oletuksesta $A \wedge \neg B$ seuraa ristiriita. Tekniikkaa kutsutaan myös epäsuoraksi todistamiseksi

  Esim. Väite: Jos $a + b = 2$, niin $a \geq 1$ tai $b \geq 1$.

  Tod.: Oletetaan, että $a + b = 2$ ja tehdään vastaväite $a < 1$ ja $b < 1$. Tällöin $2 = a + b < 1 + 1 = 2$, ristiriita. $\square$

- Miksi vastaväitetodistus toimii? Katsomme mitä tämä loogisesti tarkoittaa

- Osoitamme vastaväitetodistuksessa, että $A \wedge \neg B$ on epätosi. Vertaamme totuusarvotaulukon avulla tätä implikaatioon:

A	B	$A \wedge \neg B$	$A \Rightarrow B$
T	T	F	T
T	F	T	F
F	T	F	T
F	F	F	T

- Näemme siis, että $A \wedge \neg B$ on epätosi täsmälleen silloin kun $A \Rightarrow B$ on tosi

- Katsomme vielä, miten voidaan osoittaa jotain epätodeksi. Helpoin tapa on esittää tapaus, jossa väite on epätosi

  Esim. (Epätosi) väite: Kaikki alkuluvut ovat parittomia.

  Todistetaan epätodeksi: Luku 2 on alkuluku ja on parillinen. Väite ei siis päde. □

Matematiikan kertausta: Logaritmi

- Koulumatematiikasta muistetaan, että $\log_a x$ on sellainen luku z, jolla $a^z = x$. Tässä oletetaan yleensä $a > 0$, $a \neq 1$ ja $x > 0$

- Logaritmi $\log_a n$ siis kertoo, kuinka monta kertaa luku n pitää jakaa a:lla, ennen kuin päästään lukuun 1. Esimerkiksi $\log_2 8$ on 3, koska 8/2/2/2 on 1. Jos lukuun 1 ei päästä tasajaolla, logaritmin loppuosa on desimaaliosa, joka kuvaa viimeistä vaillinaista jakoa. Esimerkiksi $\log_2 9$ on noin 3,17, koska 9/2/2/2 on 1,125, eli kolmen jaon jälkeen on vielä hieman matkaa lukuun 1, mutta neljäs täysi jako veisi jo selvästi alle yhden

- Luonnollinen logaritmi on $\ln x = \log_e x$, missä kantalukuna on Neperin luku $e = \lim_{n\to\infty}(1 + 1/n)^n \approx 2{,}718$

- Kaksikantainen logaritmi $\log_2 n$ kertoo likimäärin luonnollisen luvun n pituuden bitteinä (tarkemmin: luvun $n > 0$ pituus bitteinä on $\lfloor \log_2 n \rfloor + 1$)

- Laskusääntöjä:

$$\begin{aligned} \log_a(xy) &= \log_a x + \log_a y \\ \log_a(x^y) &= y \log_a x \\ \log_a x &= \frac{\log_b x}{\log_b a} \end{aligned}$$

- Usein tietorakenteissa esiintyy tilanne, jossa on tiedossa esim. että $4(n+1) = 2^{j+3}$ ja olisi saatava selville mikä on j:n arvo n:n suhteen. Vakiot voivat olla tietysti mitä vain
  - Aloitetaan ottamalla 2-kantainen logaritmi molemmilta puolilta, tuloksena $\log_2(4(n+1)) = \log_2 2^{j+3}$
  - edellisen sivun laskusääntöjen perusteella vasen puoli yksinkertaistuu seuravasti $\log_2(4(n+1)) = \log_2 4 + \log_2(n+1) = 2 + \log_2(n+1)$
  - ja oikea puoli $\log_2 2^{j+3} = (j+3)\log_2 2 = (j+3) \cdot 1 = j+3$, eli

  on päästy muotoon $2 + \log_2(n+1) = j+3$, joten $j = \log_2(n+1) - 1$

Ohjelmointitaitoa: Rekursio

- Metodi voi tunnetusti sisältää kutsuja muihin metodeihin
- Jos metodi kutsuu itseään, se on rekursiivinen
- Rekursion pitää kuitenkin päättyä joskus (eli ei saa esiintyä kehäpäättelyä): tämä perustapaus (eng. base case) ei kutsu metodia ja takaa sen, että rekursio päättyy
- Tarkastellaan seuraavaa yksinkertaista esimerkkiä: Olkoon x reaaliluku ja n luonnollinen luku. Tiedämme, että

$$x^n = \begin{cases} 1, & \text{kun } n = 0 \\ x \cdot x^{n-1}, & \text{kun } n > 0 \end{cases}$$

- Tässä tapaus $n = 0$ on perustapaus, joka sisältää ratkaisun suoraan
- Tämän voimme kirjoittaa ohjelmaksi:

```
public static double potenssi(double x, int n) {
   if (n == 0)
      return 1.0;
   return x * potenssi(x,n-1);
}
```

- Mitä tapahtuu, jos kutsutaan potenssi(2.0,4)?
- Kaavatasolla tämä siis tarkoittaa että $2^4 = 2 \cdot 2^3 = 2 \cdot (2 \cdot 2^2) = 2 \cdot (2 \cdot (2 \cdot 2^1)) = 2 \cdot (2 \cdot (2 \cdot (2 \cdot 2^0))) = 2 \cdot (2 \cdot (2 \cdot (2 \cdot 1))) = 2 \cdot (2 \cdot (2 \cdot 2)) = 2 \cdot (2 \cdot 4) = 2 \cdot 8 = 16$
- Huomaa, miten rekursio ensin ”laajenee” kunnes päästään tasolle $n = 0$ ja sitten se ”kelautuu takaisin” kun saadaan laskettua arvot yksi kerrallaan
- Ohjelmalla on ns. ajonaikainen pino (eng. run-time stack). Pino-tietorakenteeseen tullaan myöhemmin kurssilla. Tässä vaiheessa riittää ymmärtää, että kyse on asioiden laittamisesta muistiin myöhempää käyttöä varten
- Ajonaikaiseen pinoon laitetaan tietue, joka kertoo parametrien arvot ja mihin kohtaan ohjelmaa palataan kun ollaan valmiit (paluuosoite on koodin joku rivi)
- Tämä kutsutaan aktivaatiotietueeksi (eng. activation record). Enemmän näistä asioista kurssilla Tietokoneen toiminta

- Ajonaikainen pino on Javassa osa Javan virtuaalikonetta (Java Virtual Machine, JVM), joka on konekielinen ohjelma, joka suorittaa Java-ohjelman tavukoodia (eng. byte code)

- Java varaa ajonaikaiselle pinolle tietyn koon, ja jos tämä ylittyy tulee virheilmoitus StackOverflowError

- Kun aktivaatiotietueet aiheutuvat rekursiivisesta metodista, puhumme rekursiopinosta (tältä osalta ajonaikaista pinoa)

- Kutsu potenssi(2.0,4) luo aktivaatiotietueen, jossa on parametrit 2.0 ja 4 sekä paluuosoitteeksi mistä kutsu tuli

- Koska n ei ole 0, suoritetaan rivi

```
return x * potenssi(x,n-1);
```

- Mutta tämä sisältää kutsun potenssi(2.0,3), eli nyt laitetaan pinoon parametrit 2.0 ja 3 ja paluosoitteeksi tämä potenssi(2.0,4):n return-rivi, josta kutsu potenssi(2.0,3):een tuli

- Seuraavaksi potenssi(2.0,2) laittaa pinoon parametrit 2.0 ja 2 ja paluuosoitteeksi potenssi(2.0,3):n return-rivi

- Seuraavaksi potenssi(2.0,1) laittaa pinoon parametrit 2.0 ja 1 ja paluuosoitteeksi potenssi(2.0,2):n return-rivi

- Seuraavaksi potenssi(2.0,0) laittaa pinoon parametrit 2.0 ja 0 ja paluuosoitteeksi potenssi(2.0,1):n return-rivi

- Pinon sisältö on siis seuraava:

2.0	0
2.0	1
2.0	2
2.0	3
2.0	4

- Nyt ohjelma palauttaa arvon 1.0 eikä kutsu enää itseään, eli poistetaan pinosta tietue, jossa on 2.0 ja 0 ja paluuosoitteena potenssi(2.0,1):n return-rivi ja siirrytään sille riville

- Suoritetaan potenssi(2.0,1):n return-rivin lasku 2.0 * 1.0 = 2.0 ja poistetaan pinosta tietue, jossa on 2.0 ja 1 ja paluuosoitteena potenssi(2.0,2):n return-rivi ja siirrytään sille riville

- Suoritetaan potenssi(2.0,2):n return-rivin lasku 2.0 * 2.0 = 4.0 ja poistetaan pinosta tietue, jossa on 2.0 ja 2 ja paluuosoitteena potenssi(2.0,3):n return-rivi ja siirrytään sille riville

- Suoritetaan potenssi(2.0,3):n return-rivin lasku 2.0 * 4.0 = 8.0 ja poistetaan pinosta tietue, jossa on 2.0 ja 3 ja paluuosoitteena potenssi(2.0,4):n return-rivi ja siirrytään sille riville

- Suoritetaan potenssi(2.0,4):n return-rivin lasku 2.0 * 8.0 = 16.0 ja poistetaan pinosta tietue, jossa on 2.0 ja 4 ja paluuosoittena paikka mistä metodi kutsuttiin

- Huomautus: potenssia ei kannata toteuttaa näin, esimerkki oli valittu jotta se olisi helppo selittää. Parempi toteutus käyttää iteraatiota:

```
public static double potenssi(double x, int n) {
   double result = 1;
   for (int j = 1; j <= n; j++)
      result *= x;
   return result;
}
```

- Tämä toteutus vie paljon vähemmän tilaa (ja on nopeampi), koska ei tarvita rekursiopinoa!

- Hyvät ja huonot puolet: Rekursiivinen ohjelma on yleensä helpompi ymmärtää ja ohjelmoida, koodi on lyhyempää. Toisaalta rekursiivinen ohjelma on yleensä hitaampi ja vie enemmän muistia

- Huom: Periaatteessa jokainen rekursio voidaan muuttaa iteraatioksi
- Tulemme kurssin aikana näkemään algoritmeja, joita on helppoja esittää rekursiivisesti, mutta vaikeampia koodata iteratiivisiksi ohjelmiksi
- Muita esimerkkejä: Kertoman määritelmä on

$$n! = \begin{cases} 1 & \text{kun } n = 0 \\ n \cdot (n-1)! & \text{kun } n > 0 \end{cases}$$

- Binomikertoimelle

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

tunnetaan palautuskaava

$$\binom{n}{k} = \begin{cases} 1 & \text{kun } k = 0 \text{ tai } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{kun } 1 \leq k \leq n-1 \end{cases}$$

(jota ei kuitenkaan yleensä tehokkuussyistä kannata suoraan käyttää rekursiivisen laskennan pohjana).

- Katsotaan vielä esimerkki muunlaisesta rekursiosta kuin jonkin numeerisen funktion laskemisesta

- Tehtävänä on tulostaa jonkin n-alkioisen perusjoukon kaikki k-alkioiset osajoukot. Yksinkertaisuuden vuoksi oletetaan, että perusjoukko on $\{\,0,\ldots,n-1\,\}$

- Miten tehtävän voi ratkaista n-alkioisella perusjoukolla, jos osaamme jo ratkaista sen $(n-1)$-alkioisella?

- Joukon $\{\,0,\ldots,n-1\,\}$ kaikki k-alkioiset osajoukot (kun $k>0$) voidaan luetella seuraavasti:
  - Luettele joukon $\{\,1,\ldots,n-1\,\}$ kaikki $(k-1)$-alkioiset osajoukot siten, että niiden eteen on liitetty alkio 0
  - Luettele joukon $\{\,1,\ldots,n-1\,\}$ kaikki k-alkioiset osajoukot

- Jälkimmäinen luettelo jää tyhjäksi tapauksessa $k=n$

- Saamme siis n-alkioisen ongelman suunnilleen palautetuksi $(n-1)$-alkioiseen
- Rekursion toteuttaminen vaatii kuitenkin hieman lisäinformaatiota:
  - mitkä alkiot on jo valittu joukon alkuun
  - mikä on pienin jäljellä oleva alkio
- Kerätään seuraavaksi tulostettava osajoukko taulukkoon A alkio kerrallaan
- Määritellään rekursiivinen algoritmi **tulostaJoukot**(p, q), joka tulostaa kaikki sellaiset k-alkioiset osajoukot, joissa
  - pienimmät p alkiota ovat $A[1], \ldots, A[p]$
  - loput $k-p$ alkiota valitaan joukosta $\{\, q, \ldots, n \,\}$
- Kutsu **tulostaJoukot**(0,0) tuottaa nyt halutun tuloksen

Algoritmi on seuraava:

```
tulostaJoukot(p, q)
1   if p < k
2       A[p + 1] = q
3       tulostaJoukot(p + 1, q + 1)
4       if k − p < n − q
5           tulostaJoukot(p, q + 1)
6   else
7       tulosta joukko { A[1], . . . , A[k] }
```

- n ja k oletetaan määritellyksi jossain muualla
- rivin 4 ehtotestin tarkoitus on estää alkion q jättäminen pois, jos jäljellä on enää täsmälleen tarvittava määrä alkioita

2. Johdanto tietorakenteisiin

- 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-numeroparien tulostaminen aakkosjärjestyksessä
  - ...

- Suoritusaikana tiedot tallennetaan tietorakenteeseen (engl. data structure)

Tietorakenne

- Tietorakenteella tarkoitetaan
  - tapaa miten tieto tallennetaan koneen muistiin, ja
  - operaatioita joiden avulla tietoa päästään käyttämään ja muokkaamaan

- Joskus lähes samasta asiasta käytetään nimitystä abstrakti tietotyyppi (engl. abstract data type, ADT)
  - tietorakenteen sisäinen toteutus piilotetaan käyttäjältä
  - abstrakti tietotyyppi näkyy käyttäjille ainoastaan operaatioina joiden avulla tietoa käytetään
  - abstrakti tietotyyppi ei siis ota kantaa siihen miten tieto on koneen muistiin varastoitu

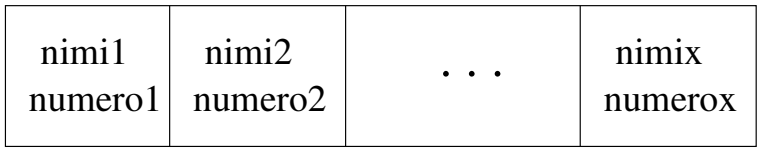
Esim. puhelinluettelo-ohjelmiston tietorakenne

- Tietorakenteen operaatioita ovat ainakin seuraavat:

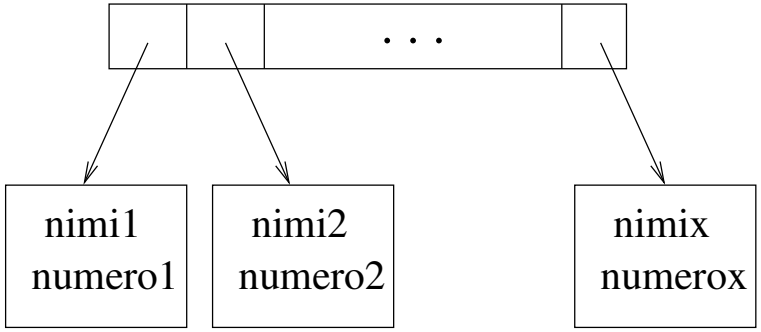
  - **add**(n, p) lisää luetteloon nimen n ja sille numeron p
  - **del**(n) poistaa nimen n luettelosta
  - **findNumber**(n) palauttaa luettelosta henkilön n numeron
  - **findName**(p) palauttaa luettelosta numeron p omistajan
  - **update**(n, p) muuttaa n:lle numeroksi p:n
  - **list**$()$ listaa nimi-numeroparit aakkosjärjestyksessä

- Seuraavassa oletetaan, että yhdellä henkilöllä voi olla ainoastaan yksi puhelinnumero

- Tieto voisi olla talletettu suoraan taulukkoon



- tai taulukosta voi olla viitteitä varsinaisen nimi/numero-parin tallettavaan muistialueeseen



- Javaa käytettäessä olisi luonnollista toteuttaa yksittäinen nimi/numero-pari omana luokkana ja itse puhelinluettelo omana luokkana

  Puhelinluettelon metodeina olisivat edellisen sivun operaatiot ja yhtenä attribuuttina taulukko viitteitä nimi/numero-pari-olioihin

- Taulukkoon perustuvat tietorakenteet ovat tehottomia (suuren) puhelinluettelon toteuttamiseen. Opimme kurssin kuluessa useita parempia tietorakenteita (mm. hakupuut ja hajautusrakenteet) ongelman ratkaisemiseen

Esim. miten löydetään lyhin kahden kaupungin välinen reitti?

- Tarkastellaan toisena esimerkkinä aivan toisentyyppistä ongelmaa

  Annettu: Suomen paikkakuntien väliset suorat maantie-etäisyydet
  Kysymys: paljonko matkaa Helsingistä Nuorgamiin?
  Jatkokysymys: mikä on lyhin reitti Helsingistä Nuorgamiin?

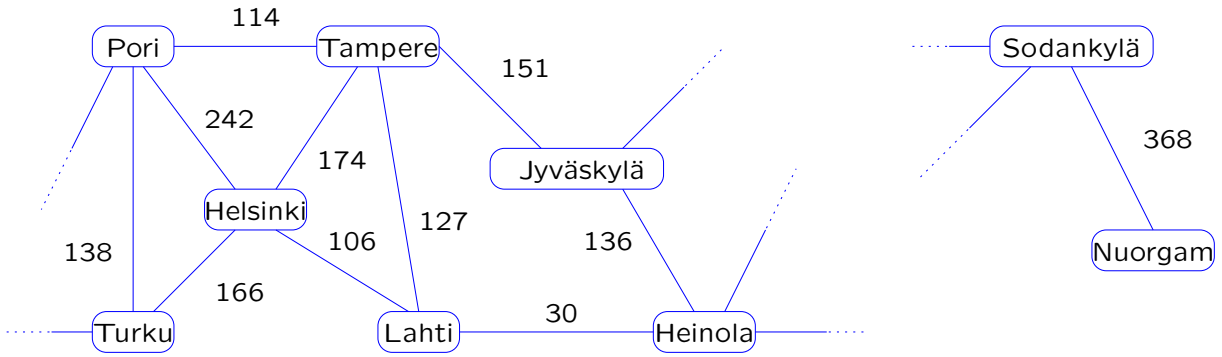
Helsinki–Lahti	106 km
Helsinki–Turku	166 km
Turku–Pori	138 km
Lahti–Tampere	127 km
...	...
Sodankylä–Nuorgam	368 km

$\Longrightarrow$

Helsinki
$\rightarrow$ Lahti
$\rightarrow$ Heinola
...
$\rightarrow$ Sodankylä
$\rightarrow$ Nuorgam

yht. 1328 km

- Tilanteeseen sopiva tietorakenne on painotettu suuntaamaton verkko



- Kysymyksessä on lyhimmän polun etsiminen verkosta

- Jos puhelinluettelo toteutetaan taulukkoon perustuen, on kaikkien operaatioiden toteuttaminen suoraviivaista

- Sen sijaan ei ole ollenkaan selvää, miten löydetään verkosta lyhin polku tehokkaasti

- Ongelma voitaisiin ratkaista raa'alla voimalla:

  käydään järjestyksessä läpi kaikki mahdolliset reitit kaupunkien välillä (eli kaupunkien permutaatiot) ja valitaan niistä lyhin

- Ratkaisu on erittäin tehoton: oletetaan että on olemassa 32 paikkakuntaa $\Rightarrow$ suuruusluokkaa $30! \approx 2,7 \cdot 10^{32}$ mahdollista reittiä!

- Oletetaan, että kone testaa miljoona reittiä sekunnissa $\Rightarrow$ tarvitaan $8,5 \cdot 10^{18}$ vuotta

- Raakaan voimaan perustuva ratkaisu on siis käytännössä käyttökelvoton

- Kurssin loppupuolella esitetään useampikin tehokas algoritmi lyhimpien polkujen etsimiseen

  Eräs näistä, ns. Dijkstran algoritmi käyttää itsessään suorituksen apuna keko-nimistä tietorakennetta

Yhteenveto

- Tietorakenteita tarvitaan
  - suurten tietomäärien hallintaan (puhelinluettelo)
  - ongelmien mallintamiseen (kaupunkien väliset minimaaliset etäisyydet)
  - myös algoritmien laskennan välivaiheiden käsittelyyn (minimaalisten etäisyyksien etsimisessä Dijkstran algoritmin apuna keko)
- Tietorakenteet ja algoritmit liittyvät erottamattomasti toisiinsa:
  - tietorakenteiden toteuttamiseen tarvitaan algoritmeja
  - oikeat tietorakenteet tekevät algoritmista tehokkaan
- Tietty ongelma voidaan usein ratkaista eri tavalla, voi olla joko vaihtoehtoisia tietorakenteita tai tiettyyn operaatioon vaihtoehtoisia algoritmeja
- Tilanteesta riippuu, mikä ratkaisuista on paras
  - kuinka paljon dataa on
  - kuinka usein mitäkin operaatiota suoritetaan
  - mitkä ovat eri operaatioiden nopeusvaatimukset
  - onko muistitilasta pulaa jne.

Algoritmien oikeellisuus

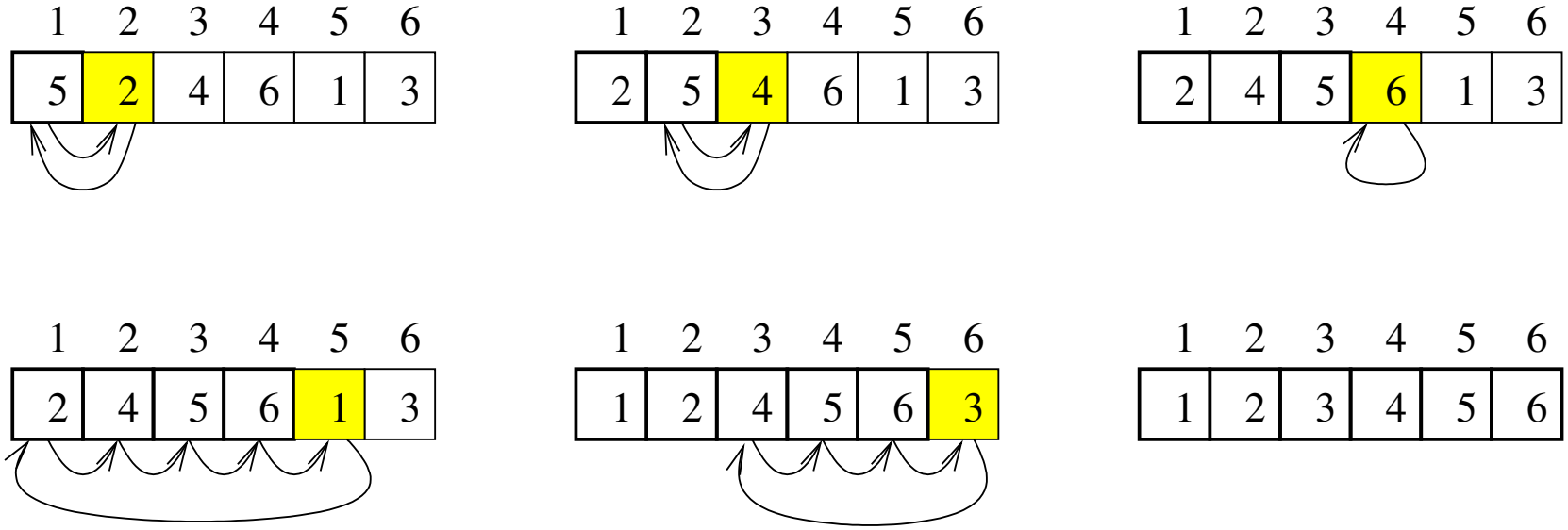
- Yksi tapa toteuttaa puhelinluettelon operaatio **add** (taulukkototeutuksessa) on lisätä uusi nimi-numeropari aina taulukon ensimmäiseen tyhjään paikkaan
- Tällöin operaation **list** yhteydessä nimet täytyy järjestää aakkosjärjestykseen
- Järjestäminen on yleisemminkin tärkeässä roolissa tietojenkäsittelyssä ja myöhemmin kurssilla tarkastellaan muutamia tehokkaita järjestämisalgoritmeja
- Tarkastellaan nyt esimerkkinä kurssilta Ohjelmoinnin perusteet ehkä tuttua **lisäysjärjestämistä** (engl. insertion sort), pseudokoodina:

```
insertion-sort(A)
1   for j = 2 to A.length
        // invariantin paikka
2       x = A[j]
3       // viedään A[j] paikalleen järjestettyyn osaan A[1,. . . ,j-1]
4       i = j–1
5       while i > 0 and A[i] > x
6           A[i+1] = A[i]
7           i = i–1
8       A[i+1] = x
```

- Huom: Toisin kuin esim. Javassa, alkaa taulukon indeksointi ykkösestä

- Toimintaideana algoritmissa on pitää koko ajan taulukon alkupää $A[1, \ldots, j-1]$ järjestyksessä:
  - alussa $j = 2$ eli alkupää sisältää vain yhden luvun
  - while-silmukassa siirretään kohdassa j oleva luku oikeaan kohtaan alkupään järjestyksessä olevaa osaa
  - näin taulukon loppupään luvut viedään yksi kerrallaan alkupään järjestettyyn osaan ja lopulta kaikki luvut ovat järjestyksessä

- Algoritmin toiminta esimerkkisyötteellä, keltaisella merkitty alkio $A[j] = x$ jolle etsitään taulukon alkupäästä oikea paikka



- Algoritmin toimintaidea voidaan ilmaista vielä hiukan täsmällisemmin ns. invariantin avulla:

  > 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

- Invariantteja voidaan käyttää algoritmin oikeellisuustodistuksissa

- Jokin ylläolevan kaltainen väittämä osoitetaan invariantiksi näyttämällä, että

  1. invariantti on voimassa tultaessa ensimmäistä kertaa toistolauseeseen
  2. invariantti pysyy totena jokaisen toistolauseen rungon suorituksen jälkeen

- Askelien 1 ja 2 voimassaolosta seuraa se, että jos invariantti on aluksi tosi, pysyy se voimassa koko silmukan suorituksen ajan

- Erityisesti invariantti on vielä totta siinäkin vaiheessa kun silmukan suoritus loppuu

- Jos invariantti on valittu järkevästi, niin toistolauseen loppuminen yhdessä invariantin voimassaolon kanssa takaa halutun lopputuloksen

- for-silmukan alussa $j = 2$, joten invariantti

  > 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

  on voimassa, koska taulukon tarkasteltavassa osassa $A[1, \ldots, 1]$ on ainoastaan yksi alkio.

- for:in sisällä oleva while vie taulukon alkion $A[j]$ oikealle paikalleen taulukon alkuosaan
  - while:n jälkeen alkuosa $A[1, \ldots, j]$ on järjestyksessä
  - taulukon loppuosaan $A[j+1, \ldots, A.\mathit{length}]$ ei kosketa, eli alkuosassa ovat selvästi täsmälleen ne alkiot, jotka olivat taulukossa alunperin osassa $A[1, \ldots, j]$

- for-lauseen rungon suorituksen jälkeen siis on voimassa

  > taulukon osa $A[1, \ldots, j]$ on järjestyksessä ja sisältää alunperin taulukon osassa $A[1, \ldots, j]$ olleet alkiot

  ja koska for lopussa kasvattaa j:n arvo yhdellä, on invariantti jälleen voimassa

- for-lauseen suorituksen loputtua $j = A.\mathit{length} + 1$ ja koska invariantti pysyy voimassa seuraa tästä että $A[1, \ldots, A.\mathit{length}]$ on järjestyksessä ja sisältää alunperin taulukossa olleet alkiot eli algoritmi toimii oikein

- Koska kyseessä on for-lause, tiedämme varmasti, että sen suoritus päättyy (sillä oletuksella että runko-osa ei jää silmukkaan!), sen sijaan while-toistolause saattaisi jäädä ikuiseen silmukkaan

- Yleisen toistolauseen tapauksessa on siis invariantin voimassa pysymisen lisäksi näytettävä että toistolause pysähtyy

- Edellisen sivun oikeellisuustodistuksen argumentaatio, erityisesti sen osalta miksi invariantti säilyy totena for:in rungossa olevan while:n ansiosta, olisi voitu tehdä formaalimmin muotoilemalla while:lle oma invariantti

- Liian detaljoidulla tasolla tapahtuva argumentaatio ei kuitenkaan liene tarpeen, tärkeintä on ymmärtää tarkasti mitä algoritmi tekee ja tehdä päättelyt tarvittavalla formaaliuden tasolla

- Jos algoritmi koostuu peräkkäisistä osista, joissa kaikissa on oma silmukkansa, tarvitaan kaikille silmukoille omat invariantit. Yleensä myöhempien silmukoiden invariantit perustuvat oleellisesti siihen tilanteeseen, johon algoritmin syöte on muotoutunut ensimmäisten silmukoiden suorituksessa

Invariantin löytäminen

- Invariantti siis ilmaisee jonkin asiain tilan, joka on voimassa koko silmukan suorituksen ajan
- Invariantti tavallaan kiteyttää koko silmukan toimintaidean
- Algoritmin keksijällä onkin yleensä ensin mielessä invariantin kaltainen idea, joka sitten ilmaistaan koodimuodossa
- Jos halutaan todistaa algoritmin oikeellisuus invariantin avulla, on oleellista löytää sellainen invariantti, jonka avulla oikeellisuus voidaan päätellä
- Voitaisiin esim. näyttää, että seuraava on **lisäysjärjestämisen** invariantti:

  jokaisen for:in jokaisen suorituksen alussa taulukon osa $A[1, \ldots, j-1]$ sisältää alunperin taulukon osassa $A[1, \ldots, j-1]$ olleet alkiot

- Tämä invariantti ei kuitenkaan auta todistamaan että **lisäysjärjestäminen** järjestää taulukon, joten se on lähes hyödytön
- Invarianttitodistuksissa lähes suurimmaksi haasteeksi osoittautuukin sopivan invariantin löytäminen
- Ilman syvällistä ymmärtämystä algoritmin toimintaperiaatteesta on sopivaa invarianttia lähes mahdoton löytää

Toinen invarianttiesimerkki

- Tarkastellaan algoritmia tai metodia, joka etsii parametrina saamansa taulukon suurimman arvon

```
suurin(A)
1   s = A[1]
2   for i = 2 to A.length
        // invariantin paikka
3       if A[i] > s
4           s = A[i]
5   return s
```

- Metodin toiminta perustuu sille, että muuttuja s muistaa taulukon alkuosan alkioista suurimman
  - alussa s saa arvokseen taulukon ensimmäisen alkion
  - toistolauseessa verrataan tunnettua suurinta arvoa aina tutkittua aluetta seuraavassa paikassa olevaan ja päivitetään s:n arvo tarvittaessa
  - lopulta on käyty läpi kaikki taulukon alkiot ja palautetaan s joka sisältää koko taulukon suurimman alkion
- Algoritmin ytimenä olevan for-lauseen toimintaa siis kuvaa invariantti

  s:n arvo on sama kuin suurin arvo taulukon osasta $A[1, \ldots, i-1]$

- Jos halutaan olla täsmällisiä, pitää vielä todistaa että edellinen todellakin on invariantti:
  - invariantti on for:in alussa tosi koska $s = A[1]$ ja for:iin mentäessä $i = 2$ eli tarkasteltava väli on $A[1, \ldots, 1]$
  - invariantti pysyy silmukan aikana totena sillä jos uusi tarkasteltava alkio $A[i] \leq s$, on s edelleen suurin tunnettu, tai jos $A[i] > s$, päivitetään myös s
- Kun for lopetetaan, on i:n arvo $A.\,length + 1$, ja koska invariantti on edelleen voimassa, seuraa siitä, että s:n arvo on sama kuin suurin arvo taulukon osasta $A[1, \ldots, n]$ missä n on taulukon pituus eli $A.length$, eli koko taulukosta
- Invarianttien idea saattaa tuntua aluksi käsittämättömältä. Invarianttien takia ei kuitenkaan kenenkään kannata ruveta panikoimaan

Invarianttitodistuksen ja matemaattisen induktion samankaltaisuus

- Invarianttitodistus on melkein kuten induktiotodistus matematiikassa

  1. Induktiotodistuksessahan näytetään ensin että jokin väittämä $P(n)$ pätee alussa, eli esim. arvolla $n = 0$

  2. Sen jälkeen todistetaan, että oletuksesta, että $P(k)$ pätee mielivaltaiselle arvolle k seuraa $P(k + 1)$, eli väittämä pätee myös arvolle $k + 1$

- Siis $P(0)$ on tosi ja induktioaskeleen perusteella myös $P(1)$. Tästä taas seuraa induktioaskeleen perusteella, että $P(2)$ on tosi, jne. . .

- Induktiotodistuksessa ei ole invarianttitodistuksen päätelmää, että silmukan lopussa päädytään haluttuun tulokseen sillä induktiossahan halutaan todistaa, että tietty väittämä pätee kaikilla arvoilla

- Esim. järjestämisalgoritmin taas täytyy toimia oikein tietyn kokoiselle taulukolle. Taulukon koon ei sinänsä tarvitse olla rajattu.
  Invarianttitodistuksessa kuitenkin tiedetään, että taulukolla on jokin koko, eli "taulukon ulkopuolella" olevia lukuja ei tarvitse järjestää ja on oleellista, että silmukan suorittaminen loppuu jossain vaiheessa

Huomautuksia monisteessa käytettävästä pseudokoodista

- Pseudokoodin kirjoitustapa on omaksuttu Cormenin kirjan 3. painoksesta
- Kirjan toisen painoksen pseudokoodinotaatio poikkeaa paikoin uudesta kirjoitustavasta
- Pseudokoodin pitäisi olla ymmärrettävää kaikille esim. Javaa opiskelleille
  - kontrollirakenteina mm. tutut if, for ja while
  - taulukon alkioita indeksoidaan []-merkinnällä: A[i]
  - sijoituslausekkeessa käytetään = ja yhtäsuuruusvertailussa ==
  - kommentit alkavat //-merkinnällä
  - monimutkaiset asiat esitetään "oliona", jonka attribuutteihin viitataan pistenotaatiolla tyyliin A.length, henkilo.nimi, ...
  - metodien parametrien käyttäytyminen kuten Javassa, eli yksinkertaiset (int, double) arvona, monimutkaisista välitetään viite

- Pseudokoodin eroavuuksia Javaan:
  - taulukon paikkojen numerointi alkaa ykkösestä
  - loogiset konnektiivit kirjoitetaan and, or, not
  - pseudokoodin for muistuttaa enemmän Javan for eachia kuin normaalia for-lausetta
  - muuttujien, metodien parametrien tai paluuarvon tyyppejä ei määritellä eksplisiittisesti
  - lohkorakenne esitetään sisennyksen avulla aivan kuten Pythonissa, Javassahan lohkorakenne esitetään merkeillä { ja }
  - pseudokoodi on epäolio-orientoitunutta: eli olioihin liittyy vain attribuutteja, kaikki metodit ovat Javan mielessä staattisia eli saavat syötteet parametreina
  - modernien skriptikielien tyyliin return voi palauttaa monta arvoa
  - pseudokoodissa joitakin komentoja kirjoitetaan tarvittaessa englanniksi tai suomeksi, tyyliin: vaihda muuttujien x ja y arvot keskenään
- Pseudokoodi on tarkoitettu helpottamaan algoritmeista kommunikoimista ihmisten välillä

Algoritmien vaativuus

- Algoritmin tehokkuudella tai vaativuudella tarkoitetaan sen suoritusaikana tarvitsemia resursseja, esim.
  - laskentaan kuluva aika
  - laskennan vaatima muistitila
  - levyhakujen määrä, tarvittavan tietoliikenteen määrä, ...
- Tällä kurssilla tarkastellaan lähinnä aikavaativuutta ja joskus tilavaativuutta
- Yleensä algoritmin suoritukseen kuluu sitä enemmän resursseja mitä suurempi syöte on, esim. järjestäminen vie ilmiselvästi aikaa enemmän miljoonan kuin tuhannen kokoiselle taulukolle
- Resurssintarvetta kannattaakin tarkastella suhteessa algoritmin syötteen kokoon, esim taulukkoja käsittelevien algoritmien tapauksessa syötteen koko on yleensä taulukon koko
- Ei ole aina itsestään selvää, mikä on hyvä valinta algoritmin syötteen kooksi
  - Myöhemmin kurssilla huomaamme, että lyhimpien polkujen etsinnässä hyväksi arvioksi syötteen koosta osoittautuu maantiekartastossa olevien kaupunkien lukumäärä sekä kaupunkien välisten yhteyksien määrä. Algoritmin vaativuus riippuu siis molemmista näistä.

Aikavaativuus tarkemmin ilmaistuna

- Algoritmin aikavaativuudella tarkoitetaan sen käyttämää laskenta-aikaa suhteessa algoritmin syötteen pituuteen
  - laskenta-aikaa mitataan käytettyjen "pienehköjen" aikayksiköiden määrällä
  - yksinkertaisen komennon (esim. vertailu, sijoituslause, laskutoimitus tai if-, while- ja for-lauseiden ehdon testaaminen) suorittaminen vie vakiomäärän tälläisia aikayksiköitä
  - sen sijaan esim. mielivaltaisen kokoisen taulukon järjestäminen ei onnistu vakiomäärällä aikayksiköitä

- Aikavaativuus voidaan ilmaista funktiona, joka kuvaa syötteen pituuden algoritmin käyttämien aikayksiköiden määräksi

- Jonkun algoritmin aikavaativuus syötteen pituuden n suhteen voisi olla esim. $T(n) = 2n^2 + 3n + 7$, eli jos syötteen pituus on 2, kuluttaa algoritmi aikaa 21 yksikköä, jos taas syötteen pituus on 100, kuluttaa algoritmi aikaa 20307 yksikköä

- Kuten pian näemme, ei laskenta-aikaa yleensä ole mielekästä mitata tällä tarkkuudella

Tilavaativuus tarkemmin ilmaistuna

- Algoritmin tilavaativuudella tarkoitetaan sen laskennassa käyttämän aputilan määrää suhteessa algoritmin syötteen pituuteen
  - käytettyä aputilaa mitataan muistipaikoissa
  - voidaan ajatella, että muistipaikka on sen kokoinen tila, johon voidaan tallettaa totuusarvo, merkki, jokin luku (int, long, double) tai olioviite (eli käytännössä muistiosoite)
  - oleellista on, että tälläinen muistipaikka on rajatun kokoinen, esim. tietokoneessa 64 bitillä esitettävä
  - esim. taulukkoa ei voi tallentaa yhteen muistipaikkaan
  - yhteen tämän määritelmän mukaiseen muistipaikkaan ei voi tallettaa myöskään esim. Javan tarjoamaa rajattoman kokoista BigInteger-tyyppistä arvoa
  - rajattoman kokoisen luvun viemä talletustila nimittäin riippuu luvun pituudesta, eli mielivaltainen tälläinen luku ei voi mahtua mihinkään kiinteän kokoiseen muistipaikkaan
  - formaali määritelmä aika- ja tilavaativuudelle nimittäin käsittelee syötteen pituutta bitteinä, ja rajatun kokoista muistipaikkaa vastaa siis vakiomäärä bittejä

- Aikavaativuuden tapaan myös tilavaativuus voidaan esittää funktiona, joka kuvaa syötteen pituuden algoritmin laskennan apuna käyttämien muistipaikkojen määräksi

- Algoritmin syötettä ei lasketa mukaan tilavaativuuteen, eli jos algoritmi saa syötteekseen taulukon jonka koko on n, ei taulukon koolla sinänsä ole merkitystä algoritmin tilavaativuuteen

- Myöskään algoritmin koodia ei huomioida tilavaativuuteen millään tavalla

- Voimme analysoida algoritmin resurssien tarvetta
  - parhaassa tapauksessa
  - keskimääräisessä tapauksessa
  - pahimmassa tapauksessa
- Paras tapaus ei yleensä kiinnosta, sillä lähes kaikki algoritmit toimivat parhaassa tapauksessa hyvin
- Keskimääräisen tapauksen analyysi taas on teknisesti usein vaikea suorittaa ja vaatii tekemään algoritmin syötteestä oletuksia, jotka eivät ehkä ole voimassa
- Keskitymmekin kurssilla pahimman tapauksen analyysiin
  - tiedämme ainakin mihin on varauduttava
  - usein keskimääräiset tapaukset ovat suunnilleen yhtä vaikeita kuin pahimmatkin (esim. **lisäysjärjestämisellä**)
  - monilla algoritmeilla pahimmat tapaukset ovat yleisiä
- Pahimman tapauksen analyysi antaa useimmiten varsin hyvän kuvan algoritmin vaativuudesta. Muutamia poikkeuksiakin on, kuten tulemme huomaamaan

Taulukon suurimman alkion selvittävän algoritmin aikavaativuus monimutkaisesti laskettuna

- Oletetaan että rivin i suorittaminen kestää c_i aikayksikköä. Seuraavassa on merkitty kullekin koodiriville sen vievien aikayksiköiden määrä sekä tieto kuinka monta kertaa rivi suoritetaan

		aika	suorituskertojen määrä
1	s = A[1]	c_1	1
2	// s taulukon alkuosan pienin	0	1
3	for i = 2 to A.length	c_3	n
4	if A[i] > s	c_4	$n-1$
5	s = A[i]	c_5	$0 \ldots n-1$
6	return s	c_6	1

- Huom: Rivi 3 suoritetaan n kertaa, sillä for-ehdon meneminen epätodeksi (i kasvaa arvoon $A.\,length+1$) aiheuttaa yhden suorituskerran
- Rivin 5 suorituslukumäärä siis riippuu algoritmin syötteestä
  - jos jo ensimmäinen luku on suurin, ei riviä suoriteta kertaakaan
  - pahimmassa tapauksessa, eli jos alkiot ovat suuruusjärjestyksessä, suoritetaan rivi $n-1$ kertaa
- Nyt voimme laskea kuinka kauan suoritukseen kuluu summaamalla rivit

- Käytetään merkintää $T(n)$ suoritusajasta syötteellä, jonka pituus on n
- Laskemalla kuvasta rivien suoritusaikojen summa saamme:

  $T(n) = c_1 + c_3 n + c_4(n-1) + c_5 x + c_6$

  rivin 5 suorituskertoja on merkattu x:llä, ja siis pätee $0 \leq x \leq n-1$
- Yleensä algoritmien analyysissä ollaan kiinnostuneita pahimmasta tapauksesta, tällöin rivi 5 suoritetaan $n-1$ kertaa, eli edellinen sievenee muotoon

  $T(n) = (c_3 + c_4 + c_5)n + c_1 + c_6 - c_4 - c_5$
- Jos ajatellaan toteutetun algoritmin suorittamisen viemää aikaa, vakiot $c_1, \ldots, c_6$ riippuvat ohjelmointikielestä, käytettävästä laitteistosta, laitteiston kuormituksesta ym. itse algoritmin tehokkuutta ajatellen toisarvoisista seikoista
- Unohdetaan konkreettiset vakiokertoimet ja merkitään $T(n) = an + b$ joillain vakioilla a ja b eli $T(n)$ on lineaarinen funktio syötteen pituuteen n nähden
- Taulukon suurimman alkion etsiminen toimii lineaarisessa ajassa syötteen pituuteen nähden:

  syötteen koon kaksinkertaistuminen kaksinkertaistaa algoritmin suoritusajan
- Kuten pian näemme, on ikävät vakiot tapana "unohtaa" kokonaan ja merkitä aikavaativuus seuraavasti: $T(n) = \mathcal{O}(n)$

Algoritmien vaativuusluokat

- Algoritmien vaativuuden arvioinnissa vakiot, kuten edellisten sivujen a, b ja $c, c_1, \ldots, c_6$ ovat ohjelmointikieli- ja suoritusympäristökohtaisia
- Jatkossa arvioimmekin algoritmien vaativuutta karkeammin, ainoastaan luonnehtimalla mihin vaativuusluokkaan (engl. complexity class) algoritmi kuuluu
- Edellä totesimme että taulukon suurimman alkion etsiminen toimii lineaarisessa ajassa syötteen pituuteen nähden, merkitsimme tätä $T(n) = \mathcal{O}(n)$
- Tapana on siis "unohtaa" vakiokertoimet sekä vähemmän merkitsevät osat ja olla kiinnostunut ainoastaan onko algoritmin vaativuus esim. lineaarinen, neliöllinen, ...
- Määrittelemme hetken kuluttua tarkemmin mitä vaativuusluokilla matemaattisessa mielessä tarkoitetaan. Ennen sitä vielä perustellaan, miksi tälläinen karkea jaoittelu on mielekästä

- Yleisesti esiintyviä vaativuusluokkia joihin algoritmien tehokkuus luokitellaan ei itseasiassa ole kovin suurta määrää

  Mielenkiintoiset vaativuusluokat helpoimmasta vaativimpaan

  - $\mathcal{O}(1)$ vakio
  - $\mathcal{O}(\log n)$ logaritminen
  - $\mathcal{O}(n)$ lineaarinen
  - $\mathcal{O}(n \log n)$
  - $\mathcal{O}(n^2)$ neliöllinen
  - $\mathcal{O}(n^3)$, $\mathcal{O}(n^4)$ jne. polynominen
  - $\mathcal{O}(2^n)$, $\mathcal{O}(3^n)$ jne. eksponentiaalinen

- Polynomisten ja eksponentiaalisten algoritmien skaalautuvuudessa on merkittävä ero:

  - Eksponentiaaliset ovat käytännöllisiä vain pienillä (n suunnilleen joitakin kymmeniä tai maksimissaan satoja) syötteillä. esim. neliöllinen algoritmi selviää vielä miljoonienkin kokoisista syötteistä

- Huom: Kun tietojenkäsittelijä merkitsee $\log n$ tarkoitetaan silloin yleensä kaksikantaista logaritmia eli $\log_2 n$

- Laskusääntö $\log_a x = \frac{\log_b x}{\log_b a}$ kuitenkin osoittaa, että logaritmin kantaa voi vaihtaa kertomalla logaritmi sopivalla vakiolla, esim. $\log_{16} n = \frac{\log_2 n}{\log_2 16} = \frac{1}{4} \log_2 n$

- Koska kaikki logaritmien kannat ovat vakiokertoimien päässä toisistaan, ei $\mathcal{O}$ merkintöjen yhteydessä yleensä merkitä logaritmille mitään kantalukua

  – Siis esim. jos algoritmin A aikavaativuus on $f_A(n) = 2 \log_7 n$ ja algoritmin B aikavaativuus on $f_B(n) = 7 \log_{55} n$, niin molemmat ovat $\mathcal{O}$-analyysin mielessä yhtä vaativia, eli $f_A(n) = \mathcal{O}(\log n)$ ja $f_B(n) = \mathcal{O}(\log n)$

- Logaritmifunktio kasvaa erittäin hitaasti. Vaativuusluokkaan $\mathcal{O}(\log n)$ kuuluvat algoritmit ovatkin todella paljon parempia kuin vaativuusluokkaan $\mathcal{O}(n)$ kuuluvat. Samoin on esim. vaativuusluokkien $\mathcal{O}(n \log n)$ ja $\mathcal{O}(n^2)$ suhteen

Onko algoritmien vaativuuden luokittelu vaativuusluokkiin perustuen mielekäs?

- Kuten pian näemme, äsken esitelty **lisäysjärjestäminen** toimii pahimmassa tapauksessa neliöllisesti eli ajassa $T(n) = \mathcal{O}(n^2)$
- Kurssin aikana tutustumme muutamiin pahimmassa tapauksessa ajassa $T(n) = \mathcal{O}(n \log n)$ toimiviin järjestämisalgoritmeihin (lomitus- ja kekojärjestäminen)
- Vaativuusluokka siis "unohtaa" kertoimet algoritmin vaativuutta kuvaavasta funktiosta. Onko näin saatu luokittelu mielekäs?
- Oletetaan, että **lisäysjärjestäminen** on toteutettu erittäin optimaalisesti ja sen tarkkaa suoritusaikaa kuvaa funktio $T_{lis}(n) = 2n^2$, eli vakiokerroin olisi vain 2
- Oletetaan lisäksi, että aloitteleva ohjelmoija on toteuttanut ajassa $\mathcal{O}(n \log n)$ toimivan **lomitusjärjestämisen** melko epäoptimaalisesti, ja toteutuksen tarkkaa suoritusaikaa kuvaa funktio $T_{lom}(n) = 50n \log_2 n$
- Jos syöte on pienehkö (luokkaa n<200), toimii **lisäysjärjestäminen** nopeammin
- Koska nykyiset tietokoneet ovat erittäin nopeita, ei pienen syötteen (n<200) suoritusajalla ole merkitystä edes huonosti toteutetulla algoritmilla

- Entä isot syötteet? Oletetaan, että järjestettävänä on 10 miljoonan kokoinen taulukko

- Oletetaan, että **lisäysjärjestäminen** suoritetaan koneella NOPEA, joka suorittaa 10^9 komentoa sekunnissa

- Oletetaan lisäksi, että **lomitusjärjestäminen** suoritetaan koneella HIDAS, joka suorittaa ainoastaan 10^6 komentoa sekunnissa. NOPEA on siis 1000 kertaa nopeampi kone kuin HIDAS

- Nyt kone NOPEA järjestää hyvin toteutetulla $\mathcal{O}(n^2)$ ajassa voimivalla **lisäysjärjestämisellä** luvut 200000 sekunnissa eli reilussa 55 tunnissa

  Tuhat kertaa hitaampi kone HIDAS järjestää epäoptimaalisesti toteutetulla, mutta ajassa $\mathcal{O}(n \log n)$ toimivalla **lomitusjärjestämisellä** noin 11627 sekunnissa, eli alle 3,3:ssa tunnissa

- Eli kuten huomaamme, suurilla syötteillä kahden vaativuusluokan välillä on ratkaiseva ero vakiokertoimista huolimatta

Funktioiden kertaluokat

- Tarkastellaan funktioita $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$ ja $g : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$
- Sanotaan että $f(n)$ kuuluu kertaluokkaan $\mathcal{O}(g(n))$, tai lyhyemmin f kuuluu kertaluokkaan $\mathcal{O}(g)$, jos on olemassa vakiot $d > 0$ ja n_0 siten että kaikilla $n \geq n_0$ ehto $f(n) \leq dg(n)$ on voimassa
- Eli f kuuluu kertaluokkaan $\mathcal{O}(g)$ jos ja vain jos voimme valita
  - jonkin kertoimen d
  - ja kohdan n_0 lukusuoralta

  siten että tämän kohdan jälkeen funktion $f(n)$ kuvaaja pysyttelee funktion $dg(n)$ kuvaajan alapuolella
- Muodollisemmin:

$$\mathcal{O}(g(n)) = \{\, f(n) \mid \exists d, n_0 > 0 : (\forall n \geq n_0 : f(n) \leq dg(n)) \,\}$$

- Tapana on merkitä $f(n) \in \mathcal{O}(g(n))$ tähän tapaan: $f(n) = \mathcal{O}(g(n))$ tai $f = \mathcal{O}(g)$. Lisäksi, jos esimerkiksi $g(n) = n^2$, kirjoitamme suoraan $\mathcal{O}(n^2)$
- Huomaa, että yhtäsuuruusmerkki ei siis ole tavallinen matematiikan yhtäsuuruusmerkki

- Intuitiivinen tulkinta: jos f on ohjelman resurssitarvetta kuvaava funktio ja $f = \mathcal{O}(g)$, niin
  - tarpeeksi suurilla syötepituuksilla $n \geq n_0$
  - toteutuskohtaisiin vakioihin d menemättä

  resurssitarpeiden yläraja on g

- esim. $2n^2 + 3n + 7 = \mathcal{O}(n^2)$ sillä $2n^2 + 3n + 7 \leq 2n^2 + 3n^2 + 7n^2 = 12n^2$ jos $n \geq 1$. Eli määritelmän mukaisiksi vakioiksi voidaan valita $d = 12$ ja $n_0 = 1$

- Palataan ennestään tuttuun taulukon suurimman alkion selvittävään algoritmiin:
  - pahimmassa tapauksessa algoritmi kulutti aikaa

$$\begin{aligned} T(n) &= (c_3 + c_4 + c_5)n + c_1 + c_6 - c_4 - c_5 \\ &\leq (c_3 + c_4 + c_5 + c_1 + c_6 - c_4 - c_5)n \\ &= \mathcal{O}(n), \end{aligned}$$

  sillä vakioksi voidaan valita $d = c_3 + c_4 + c_5 + c_1 + c_6 - c_4 - c_5$ ja $n_0 = 1$

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

- Kannattaa huomata, että määritelmän mukaan mille tahansa vakioarvoiselle funktiolle $f(n)$ on voimassa $f(n) = \mathcal{O}(1)$.

  Jos esim. $f(n) = 123$, niin $f(n) \leq 123 \cdot 1$, eli vakiot on valittu seuraavasti: $d = 123$ ja $n_0 = 0$

- $\mathcal{O}$-merkintää käytetään ylärajan esittämiseen
  - jos jonkin algoritmin pahimman tapauksen vaativuus on esim. $\mathcal{O}(n^3)$ ei se välttämättä tarkoita että pahin tapaus toimii ajassa dn^3 jollekin d, vaan että algoritmi ei toimi ainakaan huonommin kuin ajassa dn^3
  - Esim: Koska $n = \mathcal{O}(n^5)$, voidaan merkitä että taulukon suurimman alkion etsivän algoritmin aikavaativuus on esim. $\mathcal{O}(n^5)$ mutta tämä ei tietenkään ole kovin järkevää
  - mielekästä onkin etsiä pienintä mahdollista argumenttifunktioita f, jolla algoritmin vaativuus on $\mathcal{O}(f)$

- Voidaan esittää vaativuudelle myös alaraja, ja tällöin on käytössä merkintä Ω
- $f = \Omega(g)$ jos on olemassa vakiot $d > 0$ ja n_0, siten että kaikilla $n \geq n_0$ ehto $dg(n) \leq f(n)$ on voimassa
- Esim: $2n^2 + 3n + 7 = \Omega(n^2)$ sillä kun $n > 0$, on $2n^2 + 3n + 7 \geq 2n^2$ jos $n \geq 1$. Eli voidaan valita määritelmän mukaisiksi vakioiksi $d = 2$ ja $n_0 = 1$

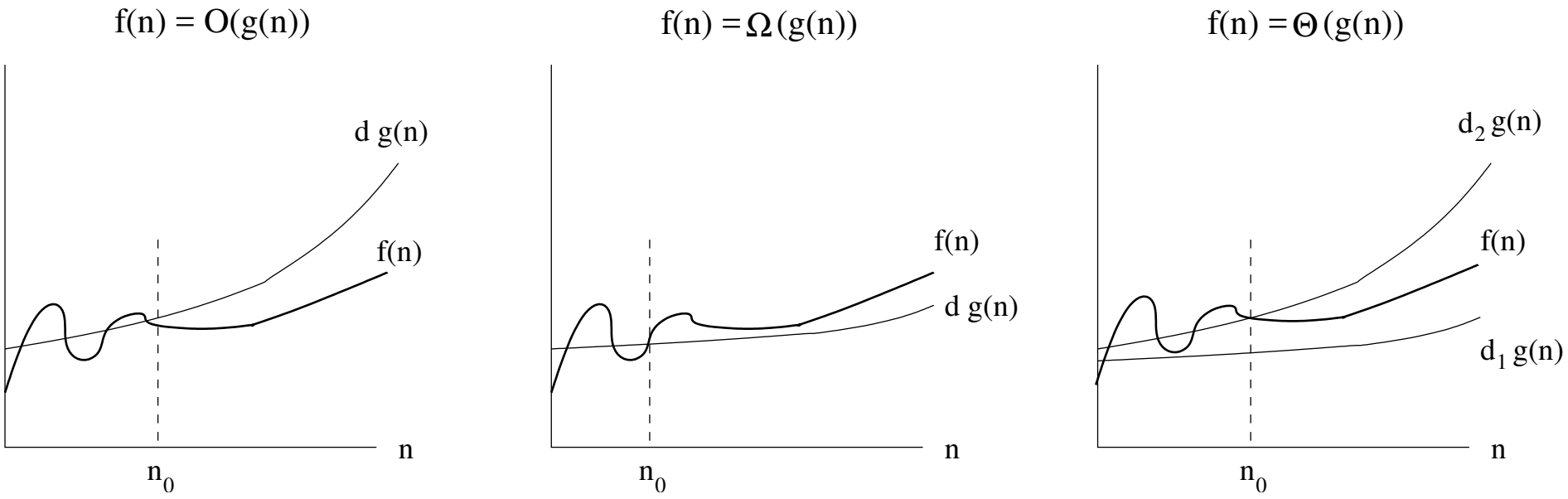
  n^2 on siis funktion $2n^2 + 3n + 7$ ala- ja yläraja

- Jos algoritmin ylä- ja alaraja vaativuudelle on sama, eli kyseessä on tarkka arvio, voidaan käyttää omaa 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, d_2 ja n_0, siten että kaikilla $n \geq n_0$ pätee $d_1 g(n) \leq f(n) \leq d_2 g(n)$
  - f käyttäytyy oleellisesti samoin kuin g, toisin sanoen f:llä sama asymptoottinen kasvunopeus kuin g:llä

- Seuraavat kuvat valaisevat asymptoottisten merkintöjen eroja



- Valtaosassa algoritmikirjallisuutta algoritmien vaativuudet ilmaistaan "ison oon", eli merkinnän $\mathcal{O}$ avulla

- Kuten pari sivua sitten todettiin, $\mathcal{O}$ ilmaisee oikeastaan ainoastaan funktion kasvun asymptoottisen ylärajan, ja merkintää voidaan "väärinkäyttää", esim. sanomalla että **lisäysjärjestämisen** aikavaativuus on $\mathcal{O}(n^5)$

- Tälläisen harhaanjohtavan käytön voi ehkäistä käyttämällä tarkempaa merkintää Θ, joka siis rajaa funktion sekä ylä- että alapuolelta. Cormenin kirjassa on tapana käyttää pääosin Θ-merkintää

- Näytämme vielä esimerkin vuoksi, miten voidaan todistaa suuruusluokkaväittämä vääräksi

- Yleensä helpoin tapa osoittaa jotain vääräksi, on ottaa esimerkki, jossa väite ei päde. Mutta jos haluamme osoittaa, että $n^2 = \mathcal{O}(n)$ ei päde, meidän pitää osoittaa että ei ole olemassa mitään sopivia vakioita d ja n_0, niin että haluttu ehto toimisi.

- Negaatiota sille, että "on olemassa A: B pätee"on "kaikille A: B ei päde"

- Eli
  $\neg(\exists d, n_0 > 0 : \forall n \geq n_0 : f(n) \leq dg(n)) \Leftrightarrow (\forall d, n_0 > 0 : \exists n \geq n_0 : \neg(f(n) \leq dg(n)))$

- Voimme nyt todistaa, että $n^2 = \mathcal{O}(n)$ ei päde.

  Tod.: Olkoot d ja n_0 mielivaltaiset positiiviset luvut. Jos valitaan $n \geq \max\{d+1, n_0\}$, niin pitäisi olla voimassa $n^2 \leq dn$, mutta n on valittu niin, että $n^2 \geq (d+1)n$. Näin ollen $(d+1)n \leq n^2 \leq dn$, eli $d+1 \leq d$, mikä on ristiriita. $\square$

Aika- ja tilavaativuusanalyysi käytännössä

- Käytännön algoritmianalyysissa seuraavassa esiteltävien nyrkkisääntöjen soveltaminen on avainasemassa

- Kolme ensimmäistä aikavaativuusanalysoijan nyrkkisääntöä
  - yksinkertaisten käskyjen (sijoitus, vertailu, tulostus, laskutoimitukset) aikavaativuus on vakio eli $\mathcal{O}(1)$
  - peräkkäin suoritettavien käskyjen aikavaativuus on sama kuin käskyistä eniten aikaavievän aikavaativuus
  - ehtolauseen aikavaativuus on $\mathcal{O}$(ehdon testausaika + suoritetun vaihtoehdon aikavaativuus)

- Tarkennus ja perustelu edellisen sivun toiselle säännölle:
  - Jos S_1:n aikavaativuus on $\mathcal{O}(f_1(n))$ ja S_2:n aikavaativuus on $\mathcal{O}(f_2(n))$, niin peräkkäin suoritettuna niiden aikavaativuus on $\mathcal{O}(f_1(n) + f_2(n))$
  - tämä on helppo näyttää määritelmästä lähtien, mutta on myös intuitiivisesti selvää, että peräkkäisten käskyjen suorituksen vievä aika on samaa suuruusluokkaa kuin käskyjen suoritusajan summa
  - summat yksinkertaistuvat, eli $\mathcal{O}(f_1(n) + f_2(n)) = \mathcal{O}(\max\{f_1(n), f_2(n)\})$
  - tämä johtuu siitä, että S_1 ja S_2 peräkkäin suoritettuna vie korkeintaan saman verran aikaa kuin komennoista hitaampi 2 kertaa suoritettuna
  - Edellistä kahta sääntöä voidaan soveltaa toistuvasti: käskyjonon $S_1, S_2, \ldots, S_k$ aikavaativuus on $\mathcal{O}(f_1 + \ldots + f_k) = max\{\mathcal{O}(f_1), \ldots, \mathcal{O}(f_k)\}$, jos käskyjonon pituus ei riipu syötteen pituudesta
- edellisistä seuraa, esim että usean peräkkäisen yksinkertaisen käskyn suorituksen aikavaativuus on siis vakio eli $\mathcal{O}(1)$

- Tarkastellaan seuraavaa metodia:

```
insertion-smaller(A)  // A on kokonaislukutaulukko
1   x = readInt()  // luetaan syöte käyttäjältä
2   y = readInt()  // luetaan toinen syöte käyttäjältä
3   z = readInt()  // luetaan kolmas syöte käyttäjältä
4   n = A.length
5   if x < y and x < z
6       smallest = x
7   elsif y < z
8       smallest = y
9   else
10      smallest = z
11  A[1] = smallest
12  A[n] = smallest
```

- Tilavaativuus on selvästi vakio $\mathcal{O}(1)$, sillä riippumatta syötteenä olevan taulukon A koosta metodi käyttää viisi apumuuttujaa

- Entä aikavaativuus?

- Merkitään S_{i-j}:llä riviltä i riville j muodostuvaa osaa koodista.
- Syöte, sijoitus ja vertailukäskyjen aikavaativuus vakio eli $\mathcal{O}(1)$. Peräkkäin suoritettavien käskyjen vaativuuden säännön perusteella S_{1-4} ja S_{11-12} aikavaativuus myös $\mathcal{O}(1)$

  Huomioi, että myös taulukon käsittely on vakioaikaista sillä sijoitus tapahtuu taulukon pituudesta riippumatta aina ensimmäiseen ja viimeiseen paikkaan
- Rivin 5 ehtolauseessa on kaksi vakioaikaista testiä, joten ehtojen testaus vakioaikainen. Ehdon toteutuessa suoritetaan vakioaikainen operaatio. Jos rivin 5 ehto epätosi, suoritetaan rivi 7, joka myös selvästi onnistuu vakioajassa. Eli S_{5-10} aikavaativuus $\mathcal{O}(1)$.
- On siis todettu, että metodi koostuu kolmesta peräkkäin suoritettavasta osasta S_{1-4}, S_{5-10} ja S_{11-12} joiden kaikkien aikavaativuus on $\mathcal{O}(1)$

  Soveltamalla peräkkäisten käskyjen sääntöä, saadaan päätelty, että koko metodin aikavaativuus on $\mathcal{O}(1)$
- Eli toisin sanoen, metodin aikana suoritetaan vakiomäärä käskyjä riippumatta sen syötteenä olevan taulukon A koosta
- Näin yksinkertaisen koodin aikavaativuuden analysointiin ei yleensä käytetä näin paljoa vaivaa. Todetaan ainoastaan, että on ilmeistä että koodin aikavaativuus on $\mathcal{O}(1)$. Alussa joitain asioita on kuitenkin hyödyllistä käydä läpi liiankin seikkaperäisesti

- Kannattaa huomata, että vakioajassa toimivassa algoritmissa ei välttämättä suoriteta aina samaa määrää komentoja:

```
insertion-if-nonzero(A, x)  // A on kokonaislukutaulukko
1   n = A.length
2   if x ≠ 0 and n > 3
3       A[n-3] = x
4       A[n-2] = x
5       A[n-1] = x
6       A[n] = x
```

- Jos parametri on x on nolla, ei rivejä 3-6 suoriteta, eli riippuen x:n arvosta komentoja suoritetaan joko 2 tai 6 kpl

- Suoritettavien komentojen määrä pahimmassa tapauksessa on 6, eli riippumaton toisen parametrin eli taulukon A koosta

- Koska komennot ovat yksinkertaisia komentoja, on pahimmassa tapauksessa suoritettavan käskyjonon 1-6 vaativuus $\mathcal{O}(1)$

- Neljäs nyrkkisääntö, aliohjelman suorituksen aikavaativuus:

  jos koodissa kutsutaan aliohjelmaa, on huomioitava aliohjelman suorituksen aikavaativuus, joka on $\mathcal{O}$(parametrien evaluointiaika + parametrien sijoitusaika + aliohjelman käskyjen suoritusaika)

```
1   x = readInt()
2   y = readInt()
3   z = readInt()
4   n = A.length
5   smallest = min(x, y, z)
6   A[1] = smallest
7   A[n] = smallest
```

```
min(x,y,x)
1   if x < y and x < z
2       return x
3   elsif y < z
4       return y
5   else
6       return z
```

- Aliohjelman min aikavaativuus on selvästi $\mathcal{O}(1)$
- Ohjelma koostuu peräkkäisistä $\mathcal{O}(1)$ osista, eli sen aikavaativuus on $\mathcal{O}(1)$

- Viides sääntö: silmukoita sisältävän algoritmin aikavaativuuden arvioiminen
  - arvioi silmukan runko-osan aikavaativuus: $\mathcal{O}(runko)$
  - arvioi kuinka monta kertaa silmukka yhteensä suoritetaan: lkm
  - aikavaativuus on $\mathcal{O}(lkm \cdot runko)$

```
find(A,x)  // A on kokonaislukutaulukko ja x etsittävä luku
1   i = 1
2   while i ≤ A.length
3       if A[i] == x
4           return true
5       i = i+1
6   return false
```

- Silmukan runko koostuu vakioaikaisista komennoista, eli runko vie $\mathcal{O}(1)$
- Runko suoritetaan pahimmassa tapauksessa (etsittävä ei ole taulukossa tai se on viimeisenä) taulukon pituuden verran
- Jos merkitään taulukon pituutta n:llä, silmukan vaativuus on $\mathcal{O}(n \cdot 1) = \mathcal{O}(n)$
- Koko aliohjelman vaativuus siis $\mathcal{O}(n)$ koska silmukan ulkopuoleiset rivit 1 ja 6 vakioaikaisia ja silmukka dominoi aikavaativuutta
- Tilavaativuus on vakio sillä vain 1 apumuuttuja käytössä

- Esimerkin metodin aika- ja tilavaativuudet ovat niin ilmeisiä, että yleensä niitä ei jaksettaisi näin tarkkaan analysoida

- Edellisen esimerkin sisältämän while-silmukan aikavaativuus oli helppo laskea sillä while:n runko-osan suoritusaika on jokaisella suorituskerralla sama

- Sisäkkäisten silmukoiden tapauksessa analysointi ei yleensä ole näin yksinkertaista, sillä sisemmän silmukan suorituskertojen määrä riippuu usein ulomman silmukan silmukkamuuttujasta

Esimerkki: Kuplajärjestäminen

```
bubble-sort(A)
1   for i = A.length to 2  // i:n arvo aluksi taulukon pituus, merkitään sitä n
        // invariantin paikka
2       for j = 1 to i–1
3           if A[j] > A[j+1]
4               // vaihdetaan A[j]:n ja A[j+1]:n sisältöä
5               apu = A[j]
6               A[j] = A[j+1]
7               A[j+1] = apu
```

- Invariantti:

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

  - alussa $i = n$ eli loppuosa on tyhjä ja invariantti siis voimassa
  - invariantti pysyy totena jokaisen ulomman for-lauseen rungon suorituksen jälkeen: ensin kohtaan $A[i]$ viedään alkuosan suurin alkio, sitten i:n arvo vähenee yhdellä
  - lopuksi $i = 2$ eli invariantista seuraa, että taulukon osa $A[2,\ldots,n]$ on järjestyksessä ja järjestetyn osan alkiot vähintään yhtä suuria kuin paikan $A[1]$ alkio

    siis koko taulukko on järjestyksessä

- Sisemmän for-silmukan runko sisältää valinnan ja vakioajan vieviä sijoitusoperaatioita. Runko on siis vakioaikainen, eli vaativuudeltaan $\mathcal{O}(1)$

- Kuinka monta kertaa runko suoritetaan?
  - ensimmäisellä sisemmän silmukan suorituskerralla $i = n$ eli j saa arvot väliltä $1 \ldots n-1$. Runko siis suoritetaan $n-1$ kertaa
  - Seuraavalla kerralla $i = n-1$ eli j saa arvot väliltä $1 \ldots n-2$, joten runko suoritetaan $n-2$ kertaa
  - Seuraavaksi $i = n-2$ eli j saa arvot väliltä $1 \ldots n-3$, joten runko suoritetaan $n-3$ -kertaa
  - ...
  - lopulta $i = 2$ ja j saa enää arvon 1 eli runko suoritetaan ainoastaan kerran
  - runko suoritetaan siis $1+2+\ldots+(n-1) = \sum_{i=1}^{n} i - n = \frac{1}{2}n(n+1) - n$ kertaa, eli aikavaativuus $\mathcal{O}(n^2)$

- Huom: Analyysi hyödyntää tuttua summakaavaa $1+2\ldots+n = \sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$

Esimerkki: Lisäysjärjestämisen aikavaativuus

- Algoritmi on siis seuraava:

```
insertion-sort(A)
1   for j = 2 to A.length
2       x = A[j]
3       // viedään A[j] paikalleen järjestettyyn osaan A[1,. . . ,j-1]
4       i = j–1
5       while i>0 and A[i]>x
6           A[i+1] = A[i]
7           i = i–1
8       A[i+1] = x
```

- Eli toimimme jälleen seuraavasti:
  - arvioi silmukan runko-osan aikavaativuus: $\mathcal{O}(runko)$
  - arvioi kuinka monta kertaa silmukka yhteensä suoritetaan: lkm
  - aikavaativuus on $\mathcal{O}(lkm \cdot runko)$

- Sisemmän silmukan runko-osan aikavaativuus selvästi $\mathcal{O}(1)$
- Kuinka monta kertaa sisempi silmukka (while) suoritetaan? Merkitään seuraavassa taulukon A pituutta n:llä
- Toisin kuin kuplajärjestämisessä, sisemmän silmukan suorituskertojen määrä ei ole sama jokaisella syötteellä
- Jos taulukko on valmiina järjestyksessä, ei sisempää toistoa (while) suoriteta kertaakaan, eli ulomman toiston (for) runko-osa vie siinä tapauksessa vakiomäärän aikaa

  eli parhaassa tapauksessa algoritmi toimii kuten sisempää while:ä ei olisi ja for:in $\mathcal{O}(1)$ runko suoritetaan $n-1$ kertaa
- **Lisäysjärjestämisen** parhaan tapauksen aikavaativuus siis $\mathcal{O}(n)$
- Yleensä parhaan tapauksen aikavaatimus ei ole kovin kiinnostava
- Järjestämisalgoritmit muodostavat joskus poikkeuksen, sillä tietyissä tilanteissa saattaa olla niin, että järjestämisalgoritmin syöte on aina lähes valmiina järjestyksessä
- Tällöin **lisäysjärjestäminen** toimii aina erittäin nopeasti vaikka sen pahimman tapauksen aikavaatimus ei olekaan kilpailukykyinen parhaiden algoritmien kanssa

- Pahimmassa tapauksessa (taulukko käänteisessä järjestyksessä) sisempi while joutuu viemään alkion $A[j]$ joka kerta paikkaan $A[1]$ asti

- Sisemmän silmukan suorituskertojen määrä riippuu ulomman silmukan muuttujan j arvosta, tarkemmin ottaen sisempi silmukka suoritetaan pahimmassa tapauksessa joka kerta $j-1$ kertaa
  - aluksi $j = 2$ eli sisemmän silmukan runko suoritetaan kerran
  - sitten $j = 3$ ja sisemmän silmukan runko suoritetaan 2 kertaa
  - ...
  - lopulta $j = n$ ja sisemmän silmukan runko suoritetaan $n-1$ kertaa
  - sisemmän silmukan runko siis suoritetaan pahimmassa tapauksessa yhteensä $1 + 2 + \ldots + n - 1 = \sum_{i=1}^{n} i - n = \frac{1}{2}n(n+1) - n$ kertaa

- **Lisäysjärjestämisen** pahimman tapauksen aikavaativuus siis $\mathcal{O}(n^2)$

- Valitettavasti algoritmi toimii myös keskimäärin ajassa $\mathcal{O}(n^2)$, eli ei ole normaalilla syötteillä kilpailukykyinen myöhemmin kurssilla esiteltävien algoritmien (keko-, lomitus- ja pikajärjestäminen) kanssa

- Algoritmi käyttää kolmea apumuuttujaa, eli tilavaativuus vakio $\mathcal{O}(1)$

- Kuudes sääntö: rekursiota käyttävien algoritmien aikavaativuuden arvioiminen etenee hyvin samaan tyyliin kuin silmukoita sisältävien algoritmien analyysi
  - arvioi pelkän rekursiivisen aliohjelman runko-osan aikavaativuus: $\mathcal{O}(runko)$
  - arvioi kuinka monta kertaa rekursiokutsu yhteensä suoritetaan: lkm
  - aikavaativuus on $\mathcal{O}(lkm \cdot runko)$

- Esim. tulostetaan taulukon sisältö käänteisesti rekursiivisen metodin avulla

```
recPrint(A,i)
1   if i > A.length
2       return
3   recPrint(A,i+1)
4   println( A[i] )
```

- Tämä on tietenkin typerä tapa taulukon käänteiseen tulostamiseen, mutta saa kelvata yksinkertaisena esimerkkinä rekursiosta

- Pelkän rungon aikavaativuus vakio $\mathcal{O}(1)$

- Jos taulukon koko on n tehdään rekursiivisia kutsuja $n + 1$, eli algoritmin suorituksen aikavaativuus on $\mathcal{O}(n)$

- Toimintaperiaate:
  - metodia kutsutaan recPrint(A,1), eli toisena parametrinä on taulukon ensimmäisen paikan indeksi
  - ennen kuin metodi tulostaa parametrinsa kertomassa indeksissä olevan kohdan taulukosta, se kutsuu rekursiivisesti itseään tulostamaan seuraavassa paikassa olevan alkion
  - esim. jos taulukon A pituus on 3 ja kutsutaan recPrint(A,1), kutsuu metodi itseään rekursiivisesti recPrint(A,2), tämä taas tekee rekursiivisen kutsun recPrint(A,3), joka edelleen kutsuu recPrint(A,4)
  - koska taulukon pituus on 3 ei kutsu recPrint(A,4) tee mitään ja palataan sitä kutsuneeseen metodiin
  - palataan siis metodikutsun recPrint(A,3) riville 4 ja tapahtuu ensimmäinen tulostusoperaatio ja alkio A[3] tulostuu
  - tämän jälkeen palataan metodikutsun recPrint(A,2) riville 4 ja tulostuu alkio A[2] . . .
  - näin käy siten, että ensin tulostetaan taulukon viimeinen alkio, sitten toiseksi viimeinen, . . . , ja lopulta ensimmäinen

- Mikä on algoritmin tilavaativuus? Näyttää siltä, että koodi ei käytä yhtään apumuuttujaa, eli onko tilavaativuus vakio eli $\mathcal{O}(1)$?

- Kuten jo aiemmin todettiin, metodikutsu varaa suoritusaikaisesta pinosta tilaa parametreille ja metodin paikallisille muuttujille (tekninen termi tälle varaukselle on aktivaatiotietue). Yhden metodikutsun pinosta varaaman tilan koko on vakio eli $\mathcal{O}(1)$

- Kun recPrint(A,1) suorittaa rekursiivisen kutsun recPrint(A,2), jää sen itse varaama tila vielä pinoon sillä metodikutsu ei ole ohi ennen kuin rekursiivisesta kutsusta palataan

  Vastaavasti kun recPrint(A,2) kutsuu recPrint(A,3), jää sen varaama tila pinoon, jossa on ennestään jo kutsun recPrint(A,1) tilanvaraus.

  Kun lopulta kutsutaan rekursion päättävä recPrint(A,n), on pinossa $n+1$ kappaletta rekursiivisten kutsujen tilanvarauksia

  Algoritmin tilavaativuus on siis $\mathcal{O}(n)$

- Rekursiivisten algoritmien vaativuus on helppo ilmaista rekursioyhtälönä. Tutustutaan seuraavaksi pintapuoleisesti rekursioyhtälöihin

Rekursioyhtälöt

- Merkitään $T(k)$:lla recPrint-metodikutsun pahimman tapauksen aikavaativuutta silloin kun tarkastettavan taulukonosan pituus on k.

- T voidaan määritellä seuraavasti rekursioyhtälönä:

$$T(k) = \begin{cases} \mathcal{O}(1) & \text{kun } k = 0 \\ T(k-1) + \mathcal{O}(1) & \text{kun } k \geq 1 \end{cases}$$

- Rekursioyhtälö tulee tulkita seuraavasti:
  - jos jäljellä on nollan mittainen osa taulukkoa (eli i:n arvo on mennyt taulukon ohi), suoritetaan rivit 1 ja 2 eli vakioaikainen operaatio
  - Jos jäljellä olevan osan k pituus on vähintään 1
    - tehdään rekursiivinen kutsu, jossa käsitellään taulukon osa jonka pituus $k-1$, tämän aikavaativuus on $T(k-1)$
    - ja suoritetaan tulostusoperaatio, tämän aikavaativuus $\mathcal{O}(1)$

- Selvittääksemme algoritmin aikavaativuuden on laskettava $T(n)$:lle jokin n:stä riippuva arvo

- Oletetaan että c on jokin tarpeeksi suuri vakio ($\mathcal{O}(1)$:han tarkoittaa "jotain" vakioa eli c on jokin tälläinen vakio), ja kirjoitetaan rekursioyhtälö seuraavasti:

$$T(k) = \begin{cases} c & \text{kun } k = 0 \\ T(k-1) + c & \text{kun } k \geq 1 \end{cases}$$

- Aikavaativuus syötteen koolla n saadaan laskemalla rekursioyhtälö auki:

  Rekursioyhtälön mukaan $T(n) = T(n-1) + c$. Soveltamalla edelleen rekursioyhtälöä tiedetään, että $T(n-1) = T(n-2) + c$, ja nämä yhdistämällä saadaan $T(n) = T(n-2) + 2c$, eli

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= T(n-2) + 2c \\ &= T(n-3) + 3c \\ &\cdots \\ &= T(n-i) + ic \\ &= T(n-n) + nc \\ &= T(0) + nc \\ &= c + nc \\ &= \mathcal{O}(n) \end{aligned}$$

- Eli päädytään samaan lopputulokseen kun aiemmassa laskelmassa

- Rekursioyhtälö ratkesi varsin mukavasti aukilaskemalla. Näemme muutaman sivun päästä toisen esimerkin, jossa vastaava tekniikka toimii

- Aina ei näin ole. Kurssilla Design and Analysis of Algorithms opitaan ratkaisemaan rekursioyhtälöitä, joihin käyttämämme menetelmä ei toimi

- Tällä kurssilla on hyvä osata määritellä yksinkertaisia rekursioyhtälöitä, mutta monimutkaisten rekursioyhtälöiden ratkaiseminen ei kuulu kurssiin

- Tarkastellaan seuraavaksi Ohjelmoinnin perusteista tuttua binäärihakua (engl. binary search).

- Binäärihaun ja monien muiden "asioiden puolittamiseen" perustuvien algoritmien analyysissä tarvitaan logaritmeja

Binäärihaku

- Annettuna järjestyksessä oleva taulukko A ja luku x. Onko luku taulukossa?
- Koska taulukko on järjestyksessä, voidaan katsomalla löytyykö etsittävä luku x keskeltä aina puolittaa hakualue
  - jos nimittäin taulukon keskellä on suurempi luku kuin x, on varmaa, että keskikohdan oikealla puolella on ainoastaan x:ää suurempia lukuja ja sieltä ei enää tarvitse etsiä
  - vastaavasti jos keskellä on x:ää pienempi luku, rajautuu hakualue taulukon yläpäähän

```
binary-search(A,x)
1   vasen = 1
2   oikea = A.length
3   found = false
4   while vasen ≤ oikea and not found
5       keski = ⌊ (vasen+oikea) /2 ⌋
6       if A[keski]==x
7           found = true
8       if A[keski]>x
9           oikea = keski−1
10      else vasen = keski+1
```

- Invariantti:

  Jos x on taulukossa niin se on välillä $A[vasen, oikea]$ ja $found =$ true jos ja vain jos x löytyi jo

- Todistetaan, että algoritmi toimii oikein
  - invariantti tosi alussa sillä $vasen = 1$, $oikea = n$ ja $found =$ false
  - pysyy selvästi totena toistolauseen suorituksessa:
    jos x löytyy, asetetaan $found =$ true
    jos $A[keski] > x$ niin myös kaikilla $j > keski$ on $A[j] > x$ eli ei tarvitse etsiä kohdan $keski$ takaa ja voidaan asettaa $oikea = keski - 1$
    else-haara vastaavasti
  - lopuksi joko $vasen > oikea$ ja $found =$ false eli etsittyä ei löytynyt, tai $x = A[keski]$ ja $found =$ true

- Toistolause terminoi koska jokaisella toistolla joko hakualue pienenee tai etsitty alkio löytyy

Binäärihaun aikavaativuus

- Aikavaativuusanalyysin viides nyrkkisääntö sanoo:
  - arvioi silmukan runko-osan aikavaativuus: $\mathcal{O}(runko)$
  - arvioi kuinka monta kertaa silmukka yhteensä suoritetaan: lkm
  - silmukan aikavaativuus on $\mathcal{O}(lkm \cdot runko)$
- Silmukan runko-osan aikavaativuus selvästi $\mathcal{O}(1)$
- Kuinka monta kertaa silmukka suoritetaan pahimmassa tapauksessa?
  - merkitään taulukon pituutta n:llä, aluksi tutkittavana n:n mittainen osa
  - jokaisella while:n suorituskerralla tutkittavan osan pituus puolittuu
  - kuten muistamme $\log_2 n$ kertoo suunilleen montako kertaa n pitää puolittaa, jotta päästään 1:een
- Silmukka suoritetaan noin $\log_2 n$ kertaa, eli binäärihaun aikavaativuus siis $\mathcal{O}(\log n)$
- Koska logaritmin kantaluvulla ei ole merkitystä kertaluokkien suhteen, merkittiin aikavaativuudeksi $\mathcal{O}(\log n)$
- Binäärihaun tilavaativuus on $\mathcal{O}(1)$ sillä algoritmi käyttää kolmea apumuuttujaa

Binäärihaun aikavaativuus rekursioyhtälön avulla

- Edellisen sivun analyysi on matemaattiselta täsmällisyydeltään Tietorakenteisiin riittävä
- Esitetään kuitenkin vielä sama analyysi hieman täsmällisemmin ja käyttämällä rekursioyhtälöitä
- Merkitään $T(k)$:llä binäärihaun pahimman tapauksen aikavaativuutta silloin kuin taulukosta on vielä tutkimatta k paikkaa.
- T voidaan määritellä seuraavasti rekursioyhtälönä:

$$T(k) = \begin{cases} \mathcal{O}(1) & \text{kun } k = 1 \\ T(k/2) + \mathcal{O}(1) & \text{kun } k > 1 \end{cases}$$

- Rekursioyhtälö tulee tulkita seuraavasti:
  - Kun taulukosta on enää jäljellä yhden mittainen osa, sen tutkiminen onnistuu vakioajassa.
  - Jos tutkittavana olevan osan k pituus on yli yksi, menee algoritmilta aikaa vakion verran ja sen lisäksi vielä saman verran kuin $k/2$:n pituisen taulukon osan tutkimiseen kuluu

- Oletetaan yksinkertaisuuden vuoksi että taulukon A pituus n on jokin kahden potenssi, eli $n = 2^j$ jollain kokonaisluvulla j

  ottamalla tästä kaksikantainen logaritmi molemmilta puolilta, saadaan $j = \log_2 n$

  Tämä perusteltiin juuri muutama sivu sitten logaritmiselityksen yhteydessä ja on siis oikeastaan sama asia kuin logaritmin määritelmä.

- Olkoon c jokin tarpeeksi suuri vakio. Korvataan $\mathcal{O}(1)$ edellisen sivun rekursioyhtälössä tällä vakiolla:

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

- Aikavaativuus syötteen koolla n saadaan laskemalla rekursioyhtälö auki:

  Rekursioyhtälön mukaan $T(n) = T(n/2) + c$. Soveltamalla edelleen rekusioyhtälöä tiedetään, että $T(n/2) = T(n/4) + c$, ja nämä yhdistämällä saadaan $T(n) = T(n/4) + c + c$, eli

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + c + c \\ &= T(n/8) + c + c + c = T(n/2^3) + 3c \\ &\dots \\ &= T(n/2^j) + jc \\ &= T(n/n) + jc = (j+1)c = (\log_2 n + 1)c \\ &= \mathcal{O}(\log_2 n) = \mathcal{O}(\log n) \end{aligned}$$

- Toiseksi viimeisen rivin sievennys perustuu siihen, että oletimme taulukon pituuden n olevan jokin kahden potenssi, eli $n = 2^j$ jollain kokonaisluvulla j, josta taas seuraa $j = \log_2 n$

- Koska logaritmin kantaluvulla ei ole merkitystä kertaluokkien suhteen (ks. logaritmikertaussivu), merkittiin aikavaativuudeksi lopulta $\mathcal{O}(\log n)$

- Yksinkertaistimme analyysiä olettamalla että taulukon pituus on jokin kahden potenssi, eli $n = 2^j$, toinen yksinkertaistus jonka teimme oli rekursioyhtälössä missä emme ottaneet huomioon, että oikeasti taulukko ei jakaudu aina täsmälleen puoliksi

- Tekemämme yksinkertaistukset eivät kuitenkaan vaikuta vaativuusanalyysin lopputulokseen

Vaativuusluokkien luonnehdintaa

- Palataan vielä kertauksenomaisesti mielenkiintoisiin vaativuusluokkiin:
  - $\mathcal{O}(1)$ vakio
  - $\mathcal{O}(\log n)$ logaritminen
  - $\mathcal{O}(n)$ lineaarinen
  - $\mathcal{O}(n \log n)$
  - $\mathcal{O}(n^2)$ neliöllinen
  - $\mathcal{O}(2^n)$, $\mathcal{O}(3^n)$ jne. eksponentiaalinen
- Matemaattisessa mielessä vaativuusluokalla, esim. $\mathcal{O}(n^2)$:llä siis tarkoitetaan funktioita $f(n)$, joille jollakin vakiolla d pätee $f(n) \leq dn^2$ tarpeeksi suurilla n
- Eli $\mathcal{O}(n^2)$ on vaativuusluokka johon kuuluvat ne funktiot, joiden kasvu neliöllistä, eli kun n kaksinkertaistuu, funktion arvo nelinkertaistuu
- Seuraavassa katsaus kuhunkin vaativuusluokkaan ja esimerkkejä vaativuusluokkaan kuuluvasta algoritmista. Huomaa, että esitys ei ole kaikin osin matemaattisesti täysin täsmällinen
- Lähteenä on käytetty Antti Laaksosen tekemää osoitteesta http://www.ohjelmointiputka.net löytyvää Algoritmien aakkoset -opasta

$\mathcal{O}(1)$ - vakioaikainen tai tilainen algoritmi

- Vakioaikaisen algoritmin suoritus vie aina yhtä kauan aikaa riippumatta käsiteltävien tietojen määrästä
- Tämä tarkoittaa, että algoritmissa ei saa olla silmukoita, joiden toistokertojen määrä vaihtelisi sen mukaan, miten paljon tietoa algoritmille annetaan
- Vakioaikaiset algoritmit ovat harvinaisia: yleensä kaikki annetut tiedot täytyy käydä edes kerran läpi, jotta algoritmi voi ilmoittaa vastauksen luotettavasti
- Jos tehtävänä on esim. laskea taulukon lukujen summa, algoritmi ei voi saada summaa selville käymättä läpi kaikkia lukuja eikä vakioaikainen algoritmi tule kysymykseen
- Vakioaikaisiakin algoritmeja esiintyy. Esimerkki: Jos taulukossa on joukko lukuja, jotka on järjestetty pienimmästä suurimpaan, on olemassa vakioaikainen algoritmi, joka selvittää taulukon pienimmän luvun

  Algoritmin täytyy vain katsoa, mikä luku on taulukon ensimmäisessä kohdassa, koska se on varmasti pienin, kun luvut ovat kerran järjestyksessä. Tähän kuluu sama aika riippumatta siitä, miten paljon lukuja koko taulukossa on
- Vakiotilaisen algoritmin käyttämän aputilan määrä ei riipu syötteen pituudesta. Vakiotilaisia algoritmeja on paljon, esim. **lisäysjärjestämisen** käyttämä tila (muutama apumuuttuja) ei riipu millään tavalla syötteen pituudesta

$\mathcal{O}(\log n)$ - logaritminen algoritmi

- Logaritmifunktio $\log_2 n$ siis ilmoittaa miten monta kertaa luku n täytyy puolittaa, jotta päästään lukuun 1 (tai alle kakkoseen jos logaritmi ei ole kokonaisluku). Esimerkiksi $\log_2 8$ on 3, koska luku 8 täytyy puolittaa kolmesti, jotta tulos on 1

- Kuten binäärihaun yhteydessä nähtiin, logaritmi myös kertoo kunka monta kertaa taulukko jonka pituus on n on halkaistava kahtia, jotta päädytään yhden kokoiseen taulukkoon

- Eli logaritminen algoritmi pystyy joka askeleella pienentämään ongelman koon puoleen (tai esim. kolmasosaan), kunnes ongelma on niin pieni (noin 1), että sen ratkaisu on selvä

- Logaritmiset algoritmin ovat todella nopeita: vaikka luku n olisi suurikin, sitä ei tarvitse puolittaa kovin monta kertaa, ennen kuin tulos alittaa jo luvun 1

- Esim. jos binäärihaulle annetaan taulukko, jossa on miljoona lukua, algoritmi tarvitsee sen käsittelyyn vain noin 20 askelta, koska $\log_2 1000000$ on noin 20

- Törmäämme jatkossa algoritmeihin joiden tilavaativuus on logaritminen. Koska logaritmi kasvaa hyvin hitaasti, tätä voidaan pitää hyvänä tilavaativuutena

- Muistutus: logaritmien kantaluvuilla ei ole väliä $\mathcal{O}$-analyysissä

$\mathcal{O}(n)$ - lineaarinen algoritmi

- Lineaarinen algoritmi sisältää usein yhden for-silmukan, jossa annetut tiedot käydään läpi:

```
for i = 1 to n
    sum = sum + A[i]
```

- Lineaarisen algoritmin ajankäyttö kasvaa samassa suhteessa kuin käsiteltävän tiedon määrä

- Algoritmi on lineaarinen myös jos se käy syötteen läpi aina esim. kolmeen kertaan tai aina puolet tiedoista

- Esim. seuraava on lineaarinen kahdesta for:ista huolimatta koska ulommainen for suoritetaan 3 kertaa riippumatta syötteen koosta

```
for j = 1 to 3
    for i = 1 to n/2
        sum = sum + A[i]
```

- Lineaarinen algoritmi on aikavaativuuden kannalta nopein mahdollinen algoritmi, jos ongelma on luonteeltaan sellainen, että kaikki annetut tiedot täytyy joka tapauksessa tarkistaa

$\mathcal{O}(n^2)$ - neliöllinen algoritmi

- Neliöllinen algoritmi sisältää usein kaksi sisäkkäistä for-silmukkaa:

```
for i = 1 to n
    for j = 1 to n
        ...
```

- Neliöllinen algoritmi voi käydä läpi kaikki parit, jotka voidaan muodostaa annetuista tiedoista

- Algoritmi voi siis verrata jokaista tietoa jokaiseen tietoon. Tällöin ensimmäinen silmukka valitsee ensimmäisen tiedon ja toinen silmukka valitsee toisen tiedon

- Esimerkki: Jos taulukossa on joukko lukuja, on olemassa neliöllinen algoritmi, joka tarkistaa, onko taulukossa kahta samaa lukua

  Algoritmin ensimmäinen silmukka käy läpi taulukossa olevat luvut ja toinen silmukka tarkistaa, esiintyykö käsiteltävä luku jossain toisessa taulukon kohdassa

- Neliöllisen algoritmin molempia silmukoita ei välttämättä toisteta n kertaa, kuten järjestämisalgoritmien yhteydessä huomattiin. Aikavaativuusanalyysissä on oltava tällöin tarkkana ja yleensä sovellettava sopivaa summakaavaa

$\mathcal{O}(n^3)$ - kuutiollinen algoritmi

- Kuutiollinen algoritmi sisältää usein kolme sisäkkäistä for-silmukkaa:

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            ...
```

- Vastaavasti kuin neliöllinen algoritmi voi käydä läpi kaikki parit, kuutiollinen algoritmi voi käydä läpi kaikki kolmikot, jotka voidaan muodostaa annetuista tiedoista

- Esimerkki: Jos taulukossa on joukko lukuja, on olemassa kuutiollinen algoritmi, joka tarkistaa, voiko taulukosta valita luvut a, b ja c niin, että $a + b = c$

  Ensimmäinen silmukka valitsee luvun a, toinen silmukka valitsee luvun b ja kolmas silmukka valitsee luvun c. Silmukoiden sisällä tarkistetaan, päteekö todella $a + b = c$

$\mathcal{O}(k^n)$ - eksponentiaalinen algoritmi

- Eksponentiaalisen algoritmin suoritus voi haarautua jokaisen käsitellyn syötteen alkion kohdalla k osaan
- Esimerkiksi jos algoritmin aikavaativuus on $\mathcal{O}(2^n)$, algoritmin suoritus voi haarautua n kertaa kahteen osaan. Eksponentiaalinen algoritmi on todella hidas, ellei n ole pieni, koska jokainen haarautuminen kasvattaa algoritmin työmäärää räjähdysmäisesti
- Polynomisissa algoritmeissa for-silmukoiden määrä on kiinteä ja silmukan kesto vaihtelee tiedon määrän mukaan
- Vastaavasti eksponentiaalisissa algoritmeissa voidaan ajatella, että "for-silmukoiden"määrä vaihtelee tiedon määrän mukaan, mutta silmukoiden koko on kiinteä
- Esimerkiksi jos algoritmin aikavaativuus on $\mathcal{O}(2^n)$, siinä on n "for-silmukkaa", joista jokainen käy läpi kaksi lukua
- Käytännössä koodiin ei voi kirjoittaa vaihtuvaa määrää for-silmukoita vaan täytyy käyttää esim. rekursiota

- Esimerkki: Jos taulukossa on joukko lukuja, on olemassa eksponentiaalinen algoritmi, joka etsii tavan jakaa luvut kahteen ryhmään niin, että ryhmien lukujen summat ovat mahdollisimman lähellä toisiaan.

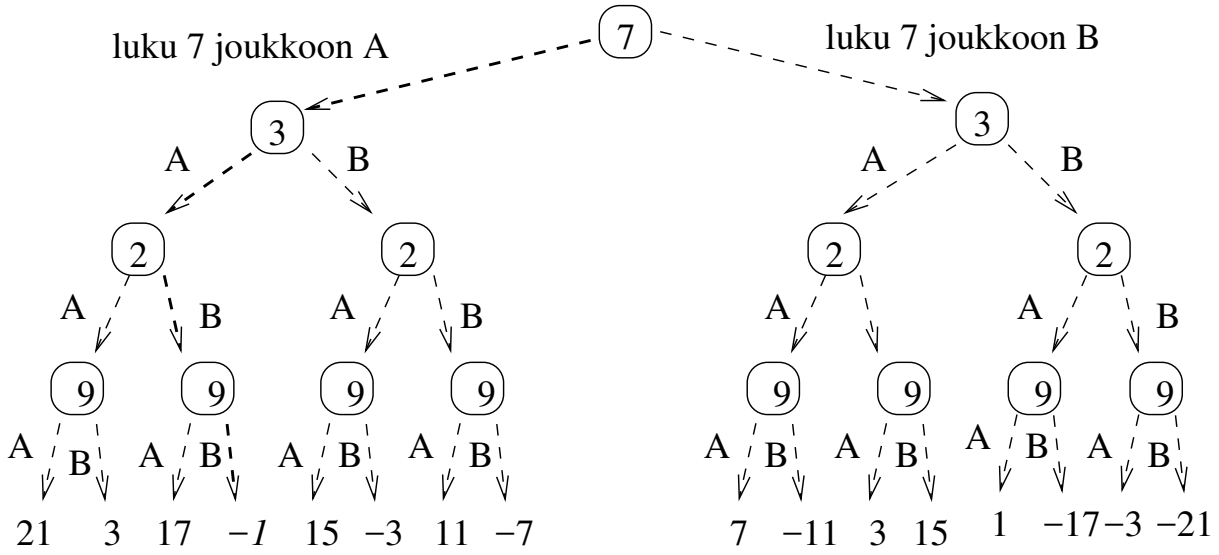
  Algoritmi haarautuu joka luvun kohdalla kahden mahdollisuuden mukaan: joko luku menee ryhmään A tai sitten se menee ryhmään B.

  Algoritmi pitää kirjaa ryhmien summista sekä pienimmän erotuksen tuottavista jaoista.

  Algoritmi haarautuu kahteen osaan joka luvun kohdalla, joten algoritmin aikavaativuus on $\mathcal{O}(2^n)$

- Seuraavalla sivulla oleva esimerkki valottaa algoritmin toimintaa

- Syötteenä luvut 7, 3, 2 ja 9 jotka algoritmi käsittelee tässä järjestyksessä
- Ensimmäinen vaihtoehto on, että luku 7 laitetaan joko joukkoon A tai joukkoon B
- Molemmat näin syntyvät laskentahaarat kokeilevat laittaa luvun 3 joko joukkoon A tai B
- Kuva havainnollistaa, miten laskentahaarojen määrä kaksinkertaistuu jokaisen käsiteltävän luvun kohdalla
- Kun kaikki luvut on käsitelty, voidaan valita syntyneistä jaoista paras (kuvassa -1 jaolla $A = \{7, 3\} B = \{2, 9\}$)



Kasvunopeudet

- Jos käsiteltävien tietojen määrä kasvaa yhdellä, muut esitellyt algoritmityypit toimivat lähes yhtä nopeasti kuin ennenkin, mutta eksponentiaalisen algoritmin ajankäyttö k-kertaistuu

- Edellisessä esimerkissä jos taulukkoon tulee uusi luku, mahdollisia jakotapoja on 32 aiemman 16:n sijasta eli aika kaksinkertaistuu

- Jos käsiteltävien tietojen määrä kaksinkertaistuu:
  - vakioaikaisen algoritmin nopeus ei muutu
  - logaritminen algoritmi tarvitsee yhden lisäaskeleen
  - lineaarinen algoritmi tarvitsee kaksinkertaisen ajan
  - neliöllinen algoritmi tarvitsee nelinkertaisen ajan
  - kuutiollinen algoritmi tarvitsee kahdeksankertaisen ajan
  - $\mathcal{O}(n^k)$-aikainen algoritmi tarvitsee 2^k-kertaisen ajan
  - eksponentiaalisen algoritmin ajankäyttö korottuu toiseen potenssiin

Osaamisen taso algoritmien suhteen

- Kurssilla tulee eteen suuri määrä algoritmeja ja tietorakenteita. Millä tasolla ne tulisi oppia?
- Algoritmisen osaamisen tasoja tai lajeja voisi kuvitella olevan ainakin seuraavat:
  1. osaa lukea monisteesta algoritmin ja kopioida sen lunttilapulle ja lunttilapulta koepaperille
  2. tuntee esim. verkkoalgoritmeja nimeltä, mutta ei oikein tiedä mitä ne tekevät
  3. osaa algoritmin pseudokoodiesityksen ulkoa, mutta ei oikein tiedä miksi algoritmi toimii
  4. ymmärtää mitä algoritmi tekee
  5. osaa toteuttaa algoritmin valitsemallaan ohjelmointikielellä
  6. ymmärtää miksi algoritmi toimii oikein
  7. ymmärtää miten paljon algoritmi käyttää resursseja
  8. osaa soveltaa algoritmia epätriviaalilla tavalla
  9. osaa todistaa matemaattisesti algoritmin vaativuuden
  10. osaa todistaa matemaattisesti algoritmin oikeaksi

- On selvää, että 1 ja 2 ovat sentyylistä "osaamista", millä ei ole juuri arvoa
- Tason 3 osaaminenkin on kyseenalaista
- Tasojen 4 ja 5 osaaminen alkaa jo olla enemmän sinne suuntaan mitä halutaan
- Yliopistotason opiskelussa keskiössä täytyy olla tasot 6-10:
  - ymmärtää miksi algoritmi toimii oikein
  - ymmärtää miten paljon algoritmi käyttää resursseja
  - osaa soveltaa algoritmia epätriviaalilla tavalla
  - osaa todistaa matemaattisesti algoritmin vaativuuden
  - osaa todistaa matemaattisesti algoritmin oikeaksi
- Algoritmien mielekkään matemaattisen analysoinnin ehdoton edellytys on kuitenkin hyvä ymmärrys algoritmin toiminnasta

  Matemaattinen todistus usein ainoastaan tuo täsmällisellä tavalla esiin sen mikä jo tiedetään ymmärryksen tasolla
- On myös olemassa algoritmeja, joista on vaikea nähdä että ne toimivat oikein ja tarvitaan matemaattinen todistus oikeellisuudesta vakuuttumiseksi

Kertauksena algoritmien aikavaativuusanalyysin nyrkkisäännöt

- yksinkertaisten käskyjen aikavaativuus on vakio eli $\mathcal{O}(1)$
- peräkkäin suoritettavien käskyjen aikavaativuus on sama kuin käskyistä eniten aikaavievän aikavaativuus
- ehtolauseen aikavaativuus on $\mathcal{O}$(ehdon testausaika + suoritetun vaihtoehdon aikavaativuus)
- aliohjelman suorituksen aikavaativuus on $\mathcal{O}$(parametrien evaluointiaika + parametrien sijoitusaika + aliohjelman käskyjen suoritusaika)
- silmukoita sisältävän algoritmin aikavaativuuden arvioiminen:
  - arvioi silmukan runko-osan aikavaativuus: $\mathcal{O}(runko)$
  - arvioi kuinka monta kertaa silmukka yhteensä suoritetaan: lkm
  - aikavaativuus on $\mathcal{O}(lkm \cdot runko)$
- rekursiota käyttävien algoritmien aikavaativuuden arvioiminen:
  - arvioi pelkän rekursiivisen aliohjelman runko-osan aikavaativuus: $\mathcal{O}(runko)$
  - arvioi kuinka monta kertaa rekursiokutsu yhteensä suoritetaan: lkm
  - aikavaativuus on $\mathcal{O}(lkm \cdot runko)$

3. Järjestäminen

- Osaamme jo muutamia $\mathcal{O}(n^2)$ ajassa toimivia menetelmiä jotka järjestävät n lukua suuruusjärjestykseen
  - vaihtojärjestäminen (*Ohjelmoinnin perusteet*)
  - lisäysjärjestäminen eli insertion sort (luku 1, sivu 36)
  - kuplajärjestäminen eli bubble sort (luku 1, sivu 73)
- Järjestäminen (engl. sorting), jota kutsutaan myös lajitteluksi, on klassinen tietojenkäsittelyongelma, jota on analysoitu runsaasti
- Järjestäminen on myös hyvä perusesimerkki, jonka yhteydessä voidaan esitellä
  - hajota ja hallitse -menetelmä algoritminsuunnittelussa
  - aikavaativuuden pahin vs. keskimääräinen tapaus

- Opimme tässä luvussa pari tapaa suorittaa järjestäminen ajassa $\mathcal{O}(n \log n)$:
  - lomitusjärjestäminen
  - pikajärjestäminen (keskimäärin)

  Myöhemmin tulee vastaan vielä
  - kekojärjestäminen

- Koska n voi olla hyvin suuri, vaativuusluokkien $\mathcal{O}(n^2)$ ja $\mathcal{O}(n \log n)$ ero on merkittävä.

- Jos järjestämisalgoritmin toiminta perustuu alkioiden vertailuun, algoritmi ei voi toimia nopeammin kuin $\mathcal{O}(n \log n)$. Palaamme tähän

- Järjestämisessä voidaan päästä lineaariseen aikaan tietyissä erikoistapauksissa. Algoritmi ei silloin voi vertailla alkioita vaan toiminta perustuu esim. taulukossa olevien lukujen laskemiseen: laskemisjärjestäminen

Lomitusjärjestäminen

Perusajatuksena on seuraava menetelmä järjestää (mahdollisesti vajaa) korttipakka:

1. Jos pakassa on vain yksi kortti, älä tee mitään
2. Muuten
   - **(a)** Jaa pakka kahteen suunnilleen yhtä suureen osaan A ja B.
   - **(b)** Järjestä osa A soveltamalla tätä menetelmää rekursiivisesti
   - **(c)** Järjestä osa B soveltamalla tätä menetelmää rekursiivisesti
   - **(d)** Lomita nyt järjestyksessä olevat osapakat A ja B siten, että koko pakka tulee järjestykseen

Lomittaminen tapahtuu esim. asettamalla osapakat A ja B kuvapuoli ylöspäin pöydälle ja ottamalla kahdesta näkyvissä olevasta kortista aina pienempi

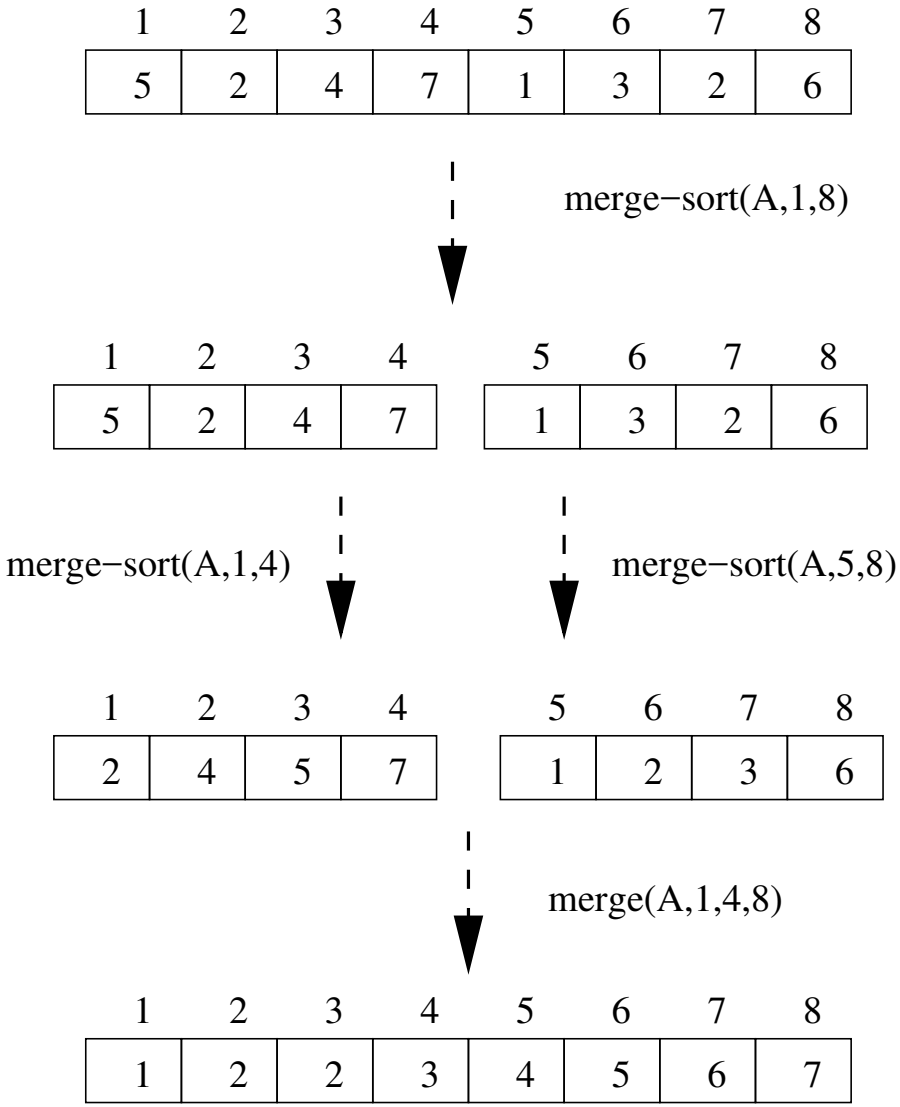
- **Lomitusjärjestäminen** perustuu hajota-ja-hallitse (engl. divide-and-conquer) -tekniikkaan:
  - hajotetaan ongelma pienempiin osaongelmiin
  - hallitaan, eli ratkaistaan osaongelmat rekursiivisesti
  - yhdistetään osaratkaisut siten että saadaan ratkaisu koko ongelmalle

- Taulukko $A[1, n]$ järjestetään kutsumalla **merge-sort**$(A, 1, n)$:

```
merge-sort(A,vasen,oikea)
1   if vasen < oikea
2       keski = ⌊(vasen + oikea)/2⌋
3       merge-sort(A,vasen,keski)
4       merge-sort(A,keski+1,oikea)
5       merge(A,vasen,keski,oikea)
```

- Toimintaidea
  - operaation tehtävänä on järjestää taulukon A osa $A[vasen, oikea]$
  - jos järjestettävän osan pituus on korkeintaan 1 (eli $vasen \geq oikea$), ei tehdä mitään, sillä tällöin haluttu taulukon osa on valmiiksi järjestyksessä (rivin 1 if-ehto)
  - rivillä 2 asetetaan $keski$ käsiteltävän taulukon osan keskikohtaan
  - taulukon osat $A[vasen, keski]$ ja $A[keski + 1, oikea]$ järjestetään kutsumalla niille **merge-sort**-operaatiota rekursiivisesti
  - riville 5 tultaessa siis taulukon osat $A[vasen, keski]$ ja $A[keski + 1, oikea]$ ovat järjestyksessä
  - rivillä 5 kutsutaan operaatiota **merge** joka "lomittaa" osaratkaisut siten että $A[vasen, oikea]$ saadaan järjestykseen

- Esimerkki (jossa on näytetty vain laskennan alku ja loppu):



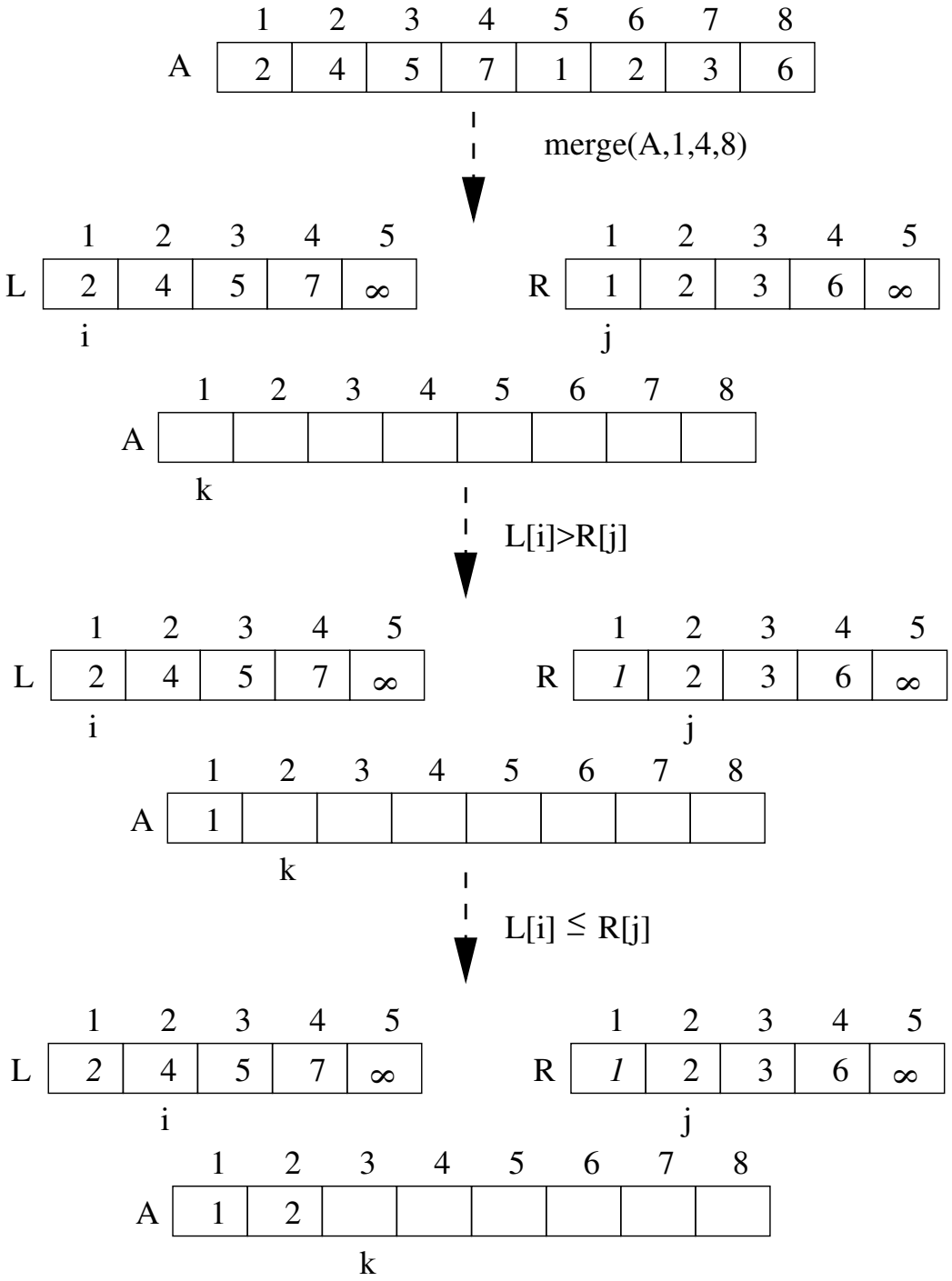
- Järjestyksessä olevien taulukon osien $A[vasen, keski]$ ja $A[keski+1, oikea]$ lomittaminen järjestykseen on helppo tehdä:
  - kopioidaan $A[vasen, keski]$ aputaulukkoon $L[1, n_1]$
  - ja $A[keski+1, oikea]$ aputaulukkoon $R[1, n_2]$
  - laitetaan paikkaan $A[vasen]$ pienin alkioista $L[1]$ ja $R[1]$,
  - jos $L[1]$ laitettiin taulukkoon, niin paikkaan $A[vasen+1]$ laitetaan pienin alkioista $L[2]$ ja $R[1]$

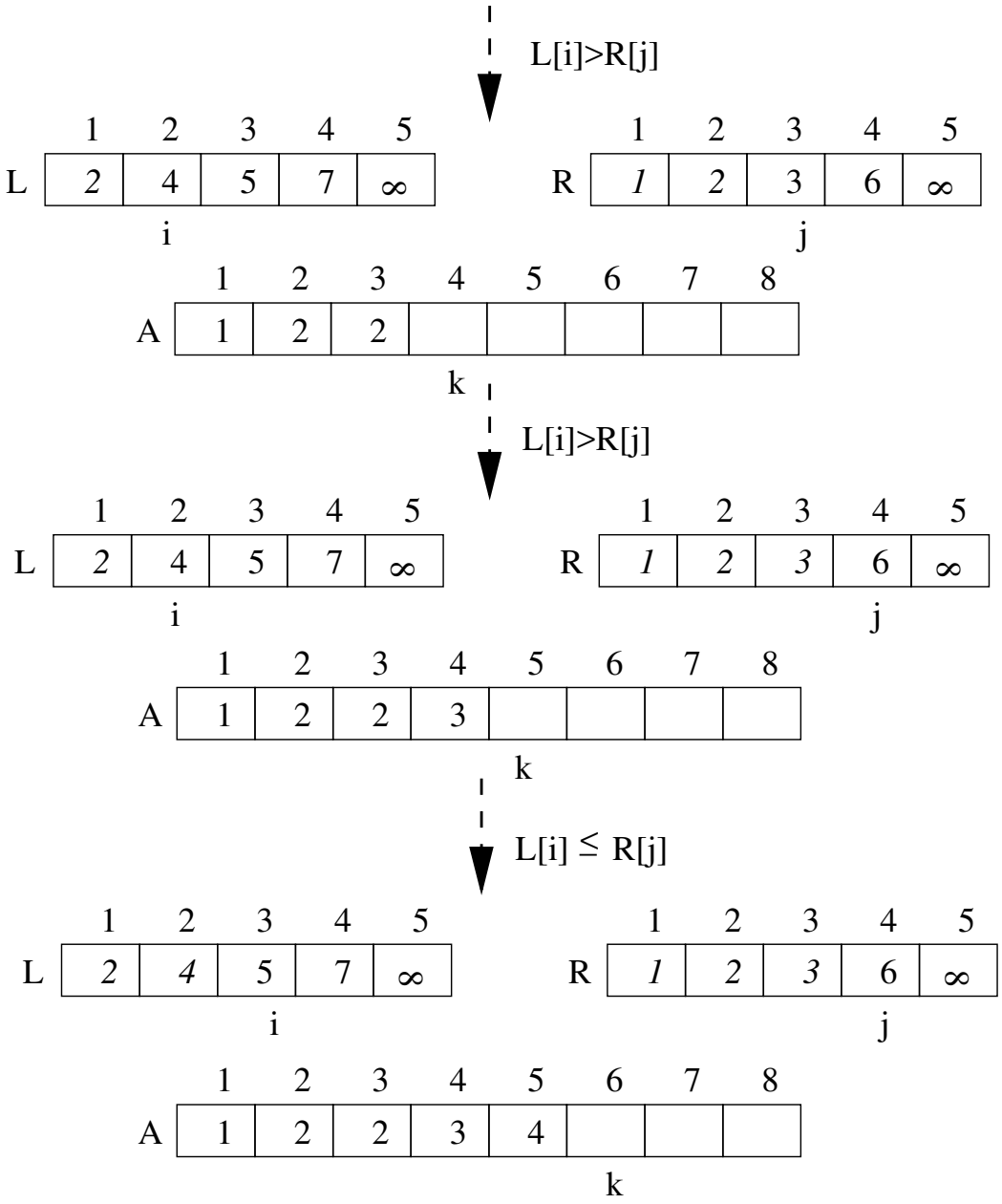
    jos taas $R[1]$ laitettiin taulukkoon, niin paikkaan $A[vasen+1]$ laitetaan pienin alkioista $L[1]$ ja $R[2]$
  - näin jatketaan kunnes kaikki aputaulukoiden alkiot on siirretty taulukkoon A

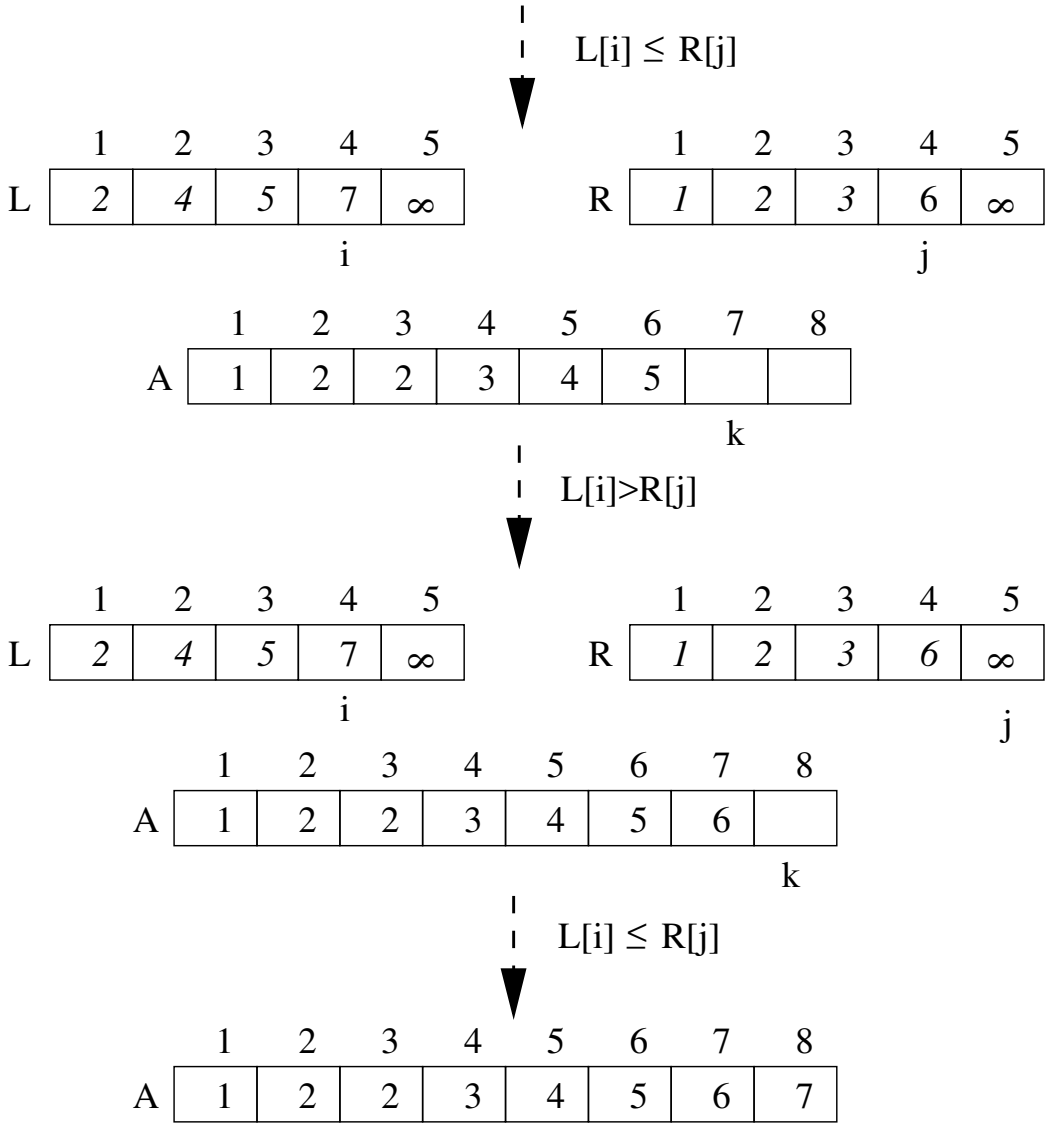
- Algoritmina

```
merge(A,vasen,keski,oikea)
1   n1 = keski−vasen+1
2   n2 = oikea−keski
3   // luodaan taulukot L[1,. . . , n1+1] ja R[1,. . . ,n2+1]
4   for i = 1 to n1
5       L[i] = A[vasen+i−1]
6   L[n1+1] = ∞          // lisätään loppuun kaikkia muita suurempi alkio
7   for j = 1 to n2
8       R[j] = A[keski+j]
9   R[n2+1] = ∞          // lisätään loppuun kaikkia muita suurempi alkio
10  i = 1
11  j = 1
12  for k = vasen to oikea
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i= i+1
16      else
17          A[k] = R[j]
18          j= j+1
```

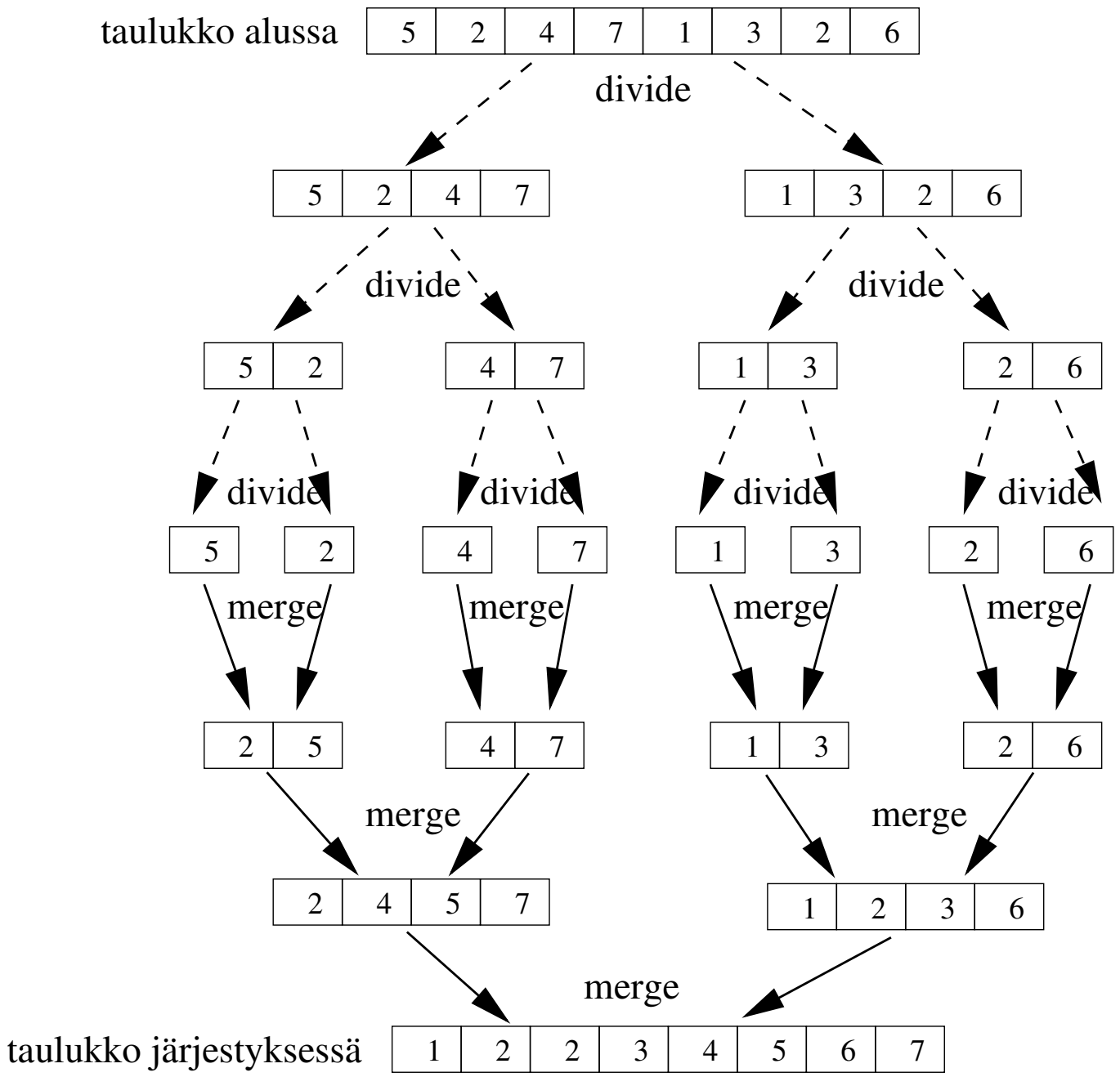
- Esimerkki **merge**-operaation toiminnasta seuraavilla sivuilla:


1 2 3 4 5 6 7 8
A 2 4 5 7 1 2 3 6
merge(A,1,4,8)
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A
k
L[i]>R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1
k
L[i] ≤ R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2
k


L[i]>R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2 2
k
L[i]>R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2 2 3
k
L[i] ≤ R[j]
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
1 2 3 4 5 6 7 8
A 1 2 2 3 4
k



- Seuraavalla sivulla esimerkki koko algoritmin toiminnasta:


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
taulukko järjestyksessä
1 2 2 3 4 5 6 7

- Ennen kuin analysoimme koko **lomitusjärjestämisen** aikavaativuutta, tarkastellaan mikä on **merge**-operaation vaativuus
  - olkoon k lomitettavan taulukonosan pituus eli $k = n_1 + n_2 = oikea - vasen + 1$
  - operaatio luo kaksi aputaulukkoa L:n ja R:n kooltaan yhteensä $n_1 + 1 + n_2 + 1 = k + 2 = \mathcal{O}(k)$
  - rivien 4 ja 7 for-lauseiden vaativuus yhteensä $\mathcal{O}(n_1 + n_2) = \mathcal{O}(k)$
  - rivin 12 for toistetaan k kertaa tehden toisto-osassa vakiomäärä operaatioita, myöhempi for-lause siis myös $\mathcal{O}(k)$
- **merge**-operaation aikavaativuus sekä tilavaativuus siis $\mathcal{O}(k)$, missä k on lomitettavan taulukonosan pituus

- Entä koko **lomitusjärjestämisen** aikavaativuus?
- Tehdään yksinkertaistava oletus: järjestettävän taulukon koko on jokin kahden potenssi, jokainen jako siis puolittaa taulukon kahteen yhtäsuureen osaan
- Käytetään k:n kokoisen taulukon **lomitusjärjestämisen** vaativuudesta merkintää $T(k)$
- k:n kokoisen taulukon **lomitusjärjestämisen** vaativuus on nyt sama kuin kahden $k/2$ kokoisen taulukon järjestämisen vaativuus + taulukon osien lomittaminen
- Yhden kokoisen taulukon **lomitusjärjestämiseksi** ei tarvitse tehdä mitään
- Eli voimme määritellä T:n rekursioyhtälönä:

$$T(k) = \begin{cases} \mathcal{O}(1) & \text{kun } k = 1 \\ T(k/2) + T(k/2) + \mathcal{O}(k) & \text{kun } k > 1 \end{cases}$$

- Taulukon $A[1,n]$ **lomitusjärjestämisen** aikavaativuus saadaan selville ratkaisemalla rekursioyhtälö $T(n)$

- Kirjoitetaan rekursioyhtälö hieman toisin, käyttämättä $\mathcal{O}$-termejä

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

  missä c on sopivasti valittu vakio

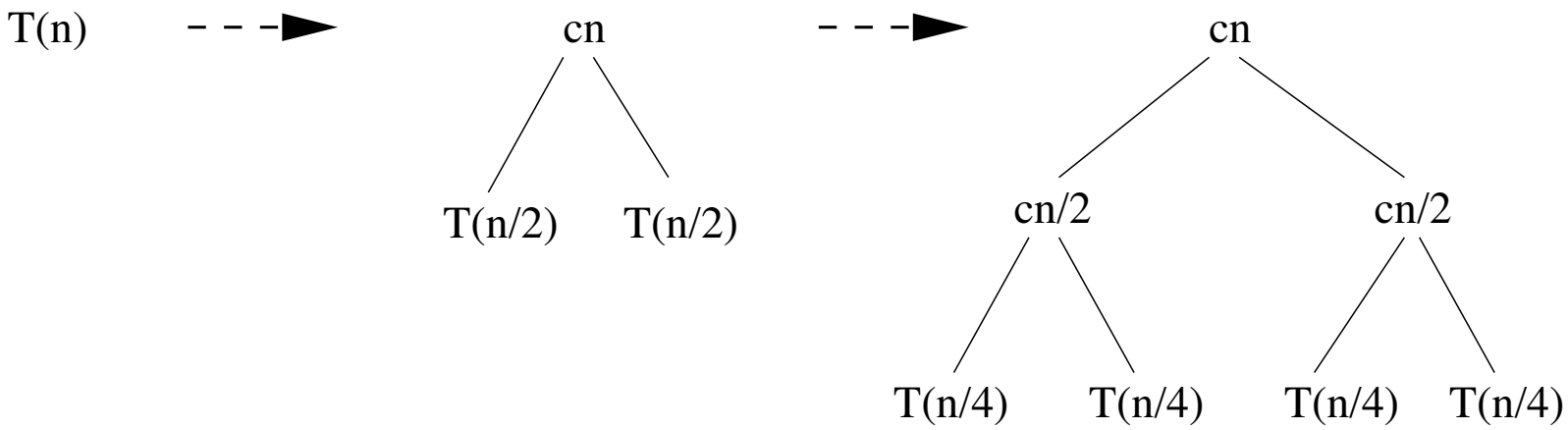
- Käytännössä molemmissa yhtälöissä $\mathcal{O}$:n korvaa oma vakio, valittu c on näistä vakioista suurempi

- Pitää selvittää mikä on n:n pituisen taulukon **lomitusjärjestämisen** aikavaativuus, eli on laskettava $T(n)$:n arvo

- Aletaan laskea auki rekursioyhtälöä:

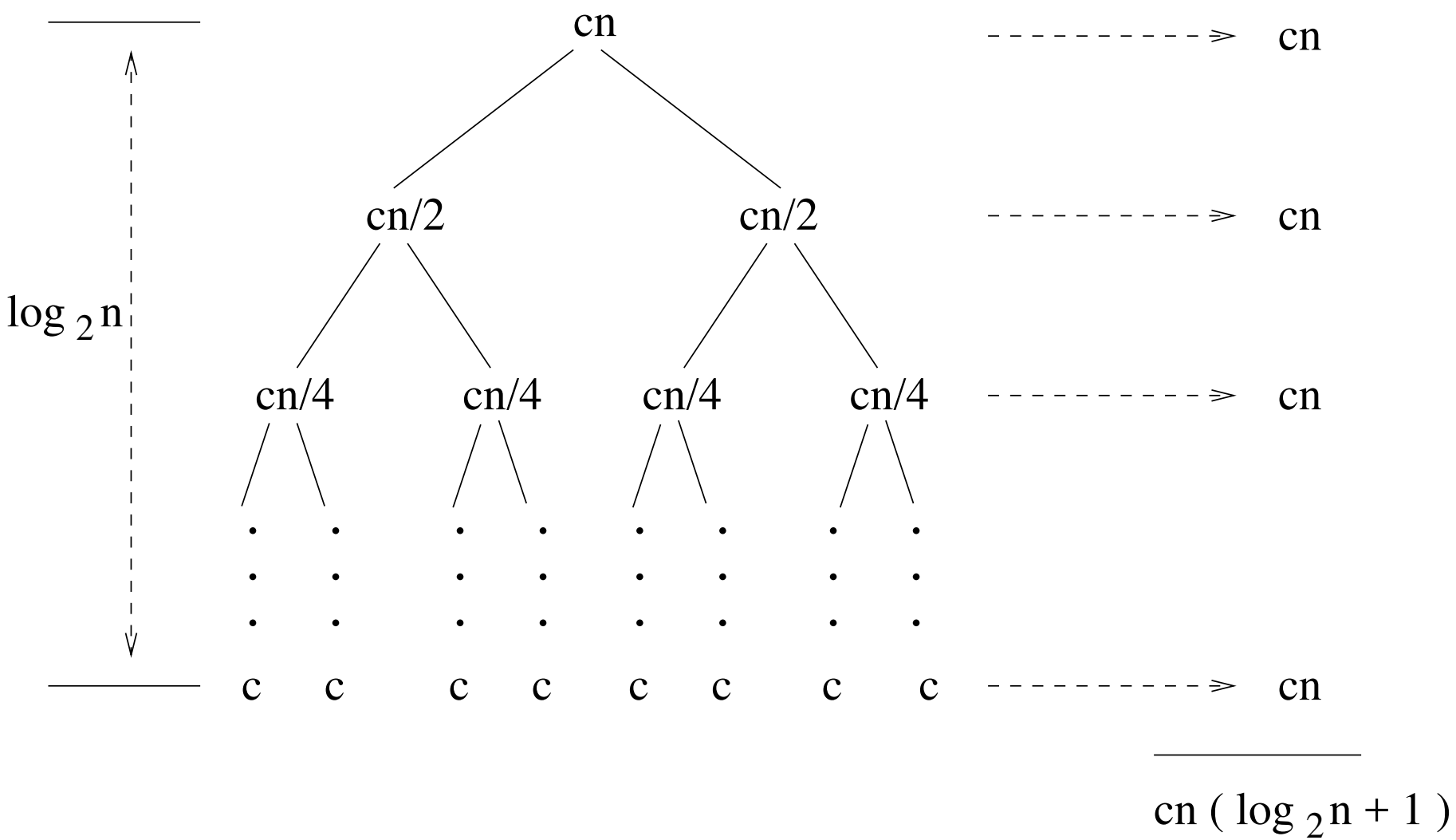
$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + cn \\ &= T(n/4) + T(n/4) + cn/2 + T(n/4) + T(n/4) + cn/2 + cn \\ &\ldots \end{aligned}$$

- Havainnollisempaa on muodostaa rekursiopuu:



- Merkataan rekursiopuun solmuihin kyseisen rekursioinstanssin vaativuus eli lomitukseen menevä aika

- Jatketaan rekursiopuun aukipiirtämistä niin kauan kunnes tullaan yhden kokoisen taulukon järjestämistä vastaaviin lehtisolmuihin


cn
cn/2
cn/2
cn/4
cn/4
cn/4
cn/4
c
log 2 n
cn
cn
cn
cn
cn (log 2 n + 1)

- Huomaamme että rekursiopuun jokaisella tasolla olevien rekursiokutsuinstanssien yhteenlaskettu vaativuus on cn

- Yksi tapa päätellä rekursiotasojen määrä on seuraava:

  Jokainen rekursiokutsu puolittaa syötteensä koon. Syötteen koko on aluksi n. Kun puolitus on tehty $\log_2 n$ kertaa, syötteen koko on enää 1. Eli mukaanlukien alin taso, jossa rekursiokutsua ei enää tapahdu, on rekursiotasoja $\log_2 n + 1$ kappaletta.

- **Lomitusjärjestämisen** aikavaativuus saadaan siis laskemalla yhteen kaikkien rekursiotasojen vaativuus, joka on $cn(\log_2 n + 1)$

- Entä jos n ei ole kakkosen potenssi?
  - silloin $2^p < n < 2^{p+1}$ jollekin positiiviselle kokonaisluvulle p
  - selvästi T on kasvava funktio, joten
$$T(n) \leq T(2^{p+1}) = c2^{p+1}(\log_2 2^{p+1} + 1)$$
  - Koska $2^{p+1} = 2 \cdot 2^p < 2n$, saamme
$$T(n) < 2cn(\log_2(2n) + 1) = 2cn(\log_2 n + 2)$$

- **Lomitusjärjestämisen** aikavaativuus on siis $\mathcal{O}(n \log n)$

- Algoritmin tilavaativuus on $\mathcal{O}(n)$ kuten **merge**-operaatiolla.

  Rekursion syvyys tosin on $\mathcal{O}(\log n)$, mutta **merge**-operaatioita on suoritettavana vain yksi kerrallaan. Siis yksittäisen **merge**-operaation tila $\mathcal{O}(n)$ dominoi koko algoritmin tilavaativuutta.

Iteratiivinen lomitusjärjestäminen

- Edellä esitetyssä versiossa rekursiiviset **merge-sort**-kutsut vain jakavat taulukkoa pienempiin osataulukoihin

- Tämä on turhaa työtä, koska tiedämme ilman muuta, mitä osataulukoita tarvitaan:
  - ensin lomitetaan $A[1]$ ja $A[2]$; sekä $A[3]$ ja $A[4]$; jne.
  - sitten lomitetaan $A[1\,..\,2]$ ja $A[3\,..\,4]$; sekä $A[5\,..\,6]$ ja $A[7\,..\,8]$; jne.
  - ...
  - lomitetaan $A[1\,..\,2^{i-1}]$ ja $A[2^{i-1}+1\,..\,2^i]$; sekä $A[2^i+1\,..\,2^i+2^{i-1}]$ ja $A[2^i+2^{i-1}+1\,..\,2\cdot 2^i]$; jne.
  - ...
  - lomitetaan $A[1\,..\,n/2]$ ja $A[n/2+1\,..\,n]$ (olettaen yksinkertaisuuden vuoksi $n = 2^p$)

- Siis lomituksen vaiheessa i lomitetaan pituutta 2^i olevia osataulukoita

- Ottamalla huomioon, että viimeinen lohko voi jäädä vajaaksi, saadaan seuraava algoritmi:

```
merge-sort2(A)
    pituus = 1 // lomitettavien lohkojen pituus
    while pituus < A.length
        alku = 1 // lomitettavan lohkoparin alkukohta
        while alku + pituus ≤ A.length
            vasen = alku
            keski = alku + pituus - 1
            oikea = min{keski+pituus, A.length} //viimeinen lohko voi olla vajaa
            merge(A,vasen,keski,oikea)
            alku = alku + 2 * pituus
        pituus = 2* pituus
```

- Selvästi **merge**-kutsut dominoivat algoritmin aikavaativuutta ja ovat samat kuin rekursiivisessa toteutuksessa, joten niihin menevä aika on sama
  - aikavaativuus on siis edelleen $\mathcal{O}(n \log n)$
- Entä tilavaativuus?
  - tilavaativuutta dominoi suurin **merge**-kutsussa tarvittava aputaulukko, jonka koko on $n/2$
  - siis tilavaativuus on $\mathcal{O}(n)$
- Rekursiivisen algoritmin muuttaminen iteratiiviseksi ei siis tässä tapauksessa tuottanut parempaa aika- tai tilavaativuusluokkaa

- Järjestämisalgoritmia sanotaan vakaaksi (engl. stable) jos sillä on seuraava ominaisuus:

  Jos kahdella eri tiedolla on sama avain, niin vakaa järjestämisalgoritmi ei muuta näiden kahden tiedon keskenäistä järjestystä

- **Lomitusjärjestäminen** on vakaa toisin kuin myöhemmin esitettävä **kekojärjestäminen** ja kohta esitettävä **pikajärjestäminen**

- Milloin järjestämisalgoritmin vakaudesta on voisi olla hyötyä?
  - Oletetaan, että järjestetään olioita, joilla on attribuutteina *etunimi*, *sukunimi* ja *ikä*
  - Järjestäminen halutaan tehdä ensisijassa sukunimen suhteen
  - Sama sukunimisten järjestyksen ratkaisee etunimen järjestys
  - Jos molemmat nimet ovat samat, ratkaisee järjestyksen ikä

- Oliot saadaan haluttuun järjestykseen seuraavasti

  jarjesta(A, 1, n)
  1 merge-sort(A, 1, n) iän mukaan
  2 merge-sort(A, 1, n) etunimen mukaan
  3 merge-sort(A, 1, n) sukunimen mukaan

- Toimintaperiaate on seuraava
  - rivin 1 jälkeen oliot ovat ikäjärjestyksessä
  - rivin 2 jälkeen oliot ovat etunimen mukaisessa järjestyksessä

    lomitusjärjestämisen vakaus takaa, että saman etunimen omaavat ovat edelleen iän mukaan järjestettynä
  - rivin 3 jälkeen järjestys sukunimien suhteen

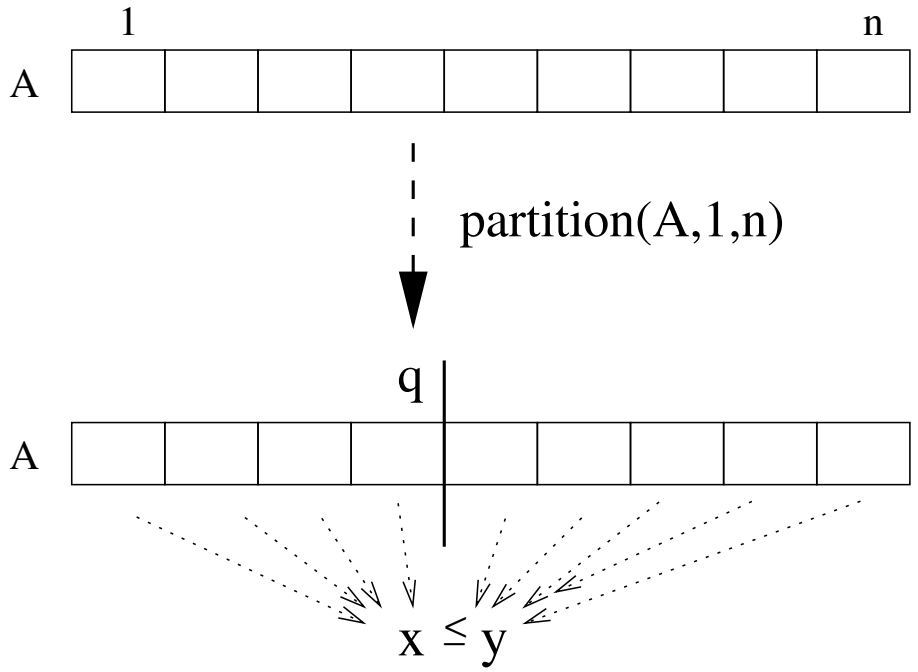
    koska **lomitusjärjestäminen** on vakaa, niin saman sukunimiset ovat edelleen etunimen mukaisessa järjestyksessä ja sekä saman etu- että sukunimen omaavien järjestyksen määrää ikä
- Aikavaativuus on edelleen $\mathcal{O}(n \log n)$ sillä $\mathcal{O}(n \log n)$ aikaa vievä **lomitusjärjestäminen** suoritetaan kolme kertaa peräkkäin
- Jos Javassa halutaan järjestää olioita vastaavalla tavalla, helpoimmalla selvitään määrittelemällä luokalle sopiva `compareTo`-metodi, joka määrittelee mikä on kahden luokan olion keskinäinen järjestys
- Java API:n luokasta `Collections` löytyy valmis staattinen metodi `sort`, jonka avulla voidaan järjestää esim. ArrayList:issa säilytettävät oliot niiden `compareTo`-metodin määrittelemään järjestykseen
- Luokasta `Arrays` löytyy vastaava metodi normaalien taulukoiden järjestämiseen

Pikajärjestäminen (Quicksort) [C.A.R. Hoare, 1962]

- Sovelletaan edelleen hajota-ja-hallitse-periaatetta
- taulukko $A[1, n]$ järjestetään kutsumalla **quicksort**$(A, 1, n)$:

```
quicksort(A,vasen,oikea)
1   if vasen < oikea
2       jako = partition(A,vasen,oikea)
3       quicksort(A,vasen,jako)
4       quicksort(A,jako+1,oikea)
```

- Toimintaidea
  - operaation tehtävänä on järjestää taulukon A osa $A[vasen, oikea]$
  - jos järjestettävän osan pituus on korkeintaan 1 (eli $vasen \geq oikea$), ei tehdä mitään, sillä tällöin haluttu taulukon osa on valmiiksi järjestyksessä (rivin 1 if-ehto)
  - rivillä 2 kutsutaan **partition**-operaatiota joka jakaa taulukon kahteen osaan jakoalkion (engl. pivot) mukaan:



- Jaon jälkeen kaikki alkuosan $A[vasen, jako]$ alkiot ovat korkeintaan yhtä suuria kuin loppuosan $A[jako + 1, oikea]$ alkiot
- Huom: toisin kuin **lomitusjärjestämisessä**, taulukon osat $A[vasen, jako]$ ja $A[jako + 1, oikea]$ eivät ole välttämättä läheskään saman kokoisia
- taulukon osat $A[vasen, jako]$ ja $A[jako + 1, oikea]$ järjestetään kutsumalla niille rekursiivisesti **quicksort**-operaatiota
- Tulosten kokoamisvaihetta ei tarvita, sillä alkuosan ja loppuosan alkiot ovat jo **partition**-operaation jäljiltä keskenään oikeassa järjestyksessä

- **Pikajärjestämisen** tehokkuuden kannalta on oleellista että **partition**-operaatio toimii nopeasti (lineaarisessa ajassa jaettavan taulukonosan koon suhteen) ja tuottaa mahdollisimman tasaisia jakoja

- Käytetään seuraavaa **partition**-operaatiota, jossa on valittu jakoalkioksi vasemmanpuolesin alkio $A[vasen]$

- Cormenissa (luku 7) on jakoalkiona käytetty oikeanpuoleisinta alkiota

- Muitakin vaihtoehtoja on

```
partition(A,vasen,oikea)
1   jakoAlkio = A[vasen]
2   i = vasen − 1
3   j = oikea + 1
4   while i < j
5       repeat j = j − 1 until A[j] ≤ jakoAlkio
6       repeat i = i + 1 until A[i] ≥ jakoAlkio
7       if i<j
8           vaihda A[i] ja A[j]
9   return j
```

- Algoritmin oikeellisuuden toteamiseksi havaitaan, että while-silmukan invariantti on:

  $A[k] \leq jakoAlkio$, kun $vasen \leq k \leq i$, ja $A[k] \geq jakoAlkio$, kun $j \leq k \leq oikea$

  paitsi ehkä viimeisen while-iteraation jälkeen

- Alkutilanteessa invariantti pätee triviaalisti. Lisäksi ensimmäisellä iteraatiolla j liikkuu vasempaan, mutta pysähtyy viimeistään indeksissä $vasen$ (jossa on arvo $jakoAlkio$) ja i liikkuu indeksiin $vasen$

- Kun repeat j -silmukka on suoritettu, on $A[k] \geq jakoAlkio$ kun $j < k \leq oikea$ ja $A[j] \leq jakoAlkio$

- Kun repeat i -silmukka on suoritettu, on $A[k] \leq jakoAlkio$ kun $vasen \leq k < i$ ja $A[i] \geq jakoAlkio$

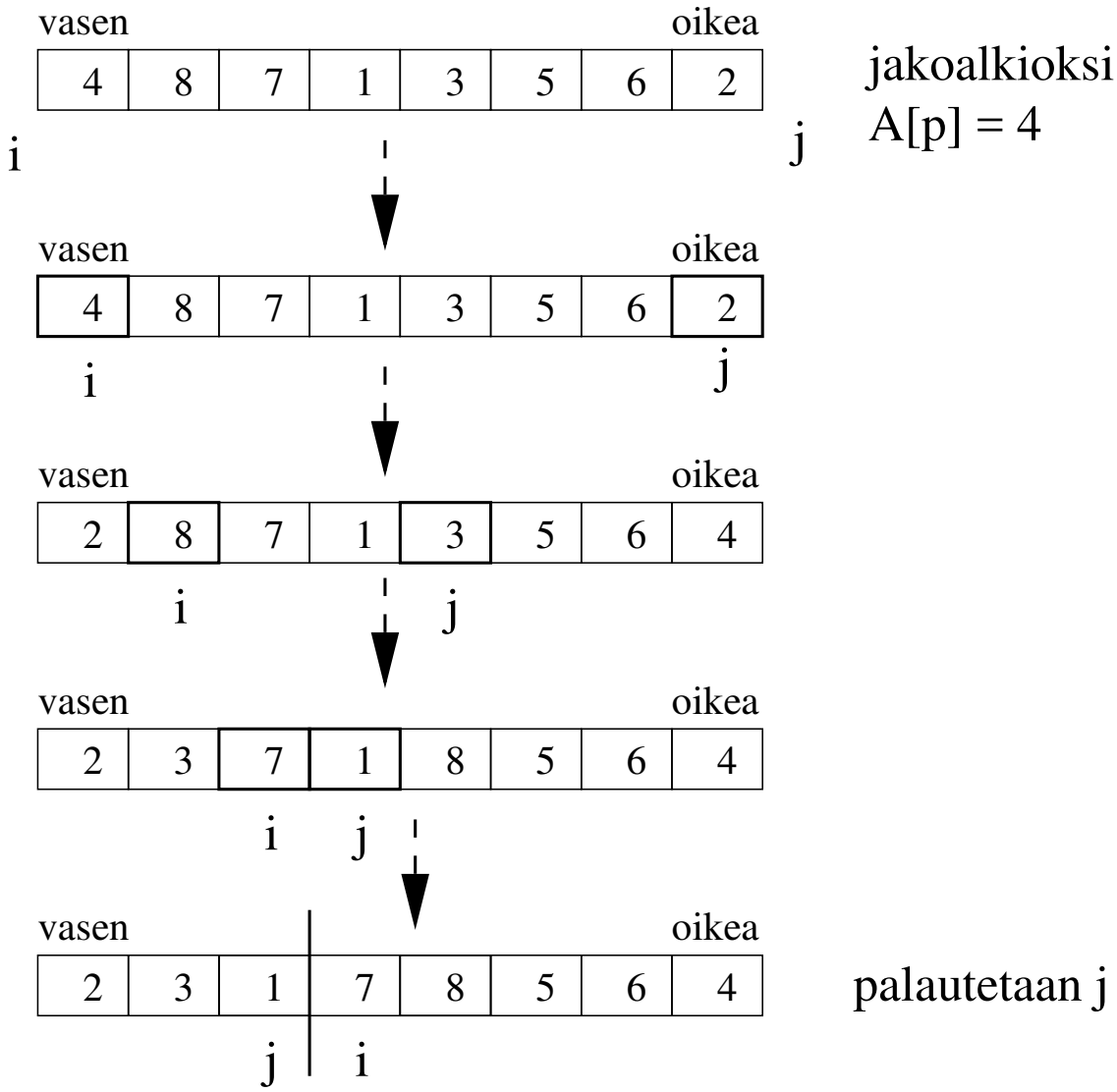
- Jos silmukka ei vielä pääty, vaihdetaan $A[i]$ ja $A[j]$ ja invariantti tulee taas voimaan

- Huomaa, myös, että joka kierroksella indeksit siirtyvät vähintään yhden verran ja että kierroksen jälkeen on olemassa alkio $\leq jakoAlkio$ taulukossa $A[vasen\,..i]$ ja alkio $\geq jakoAlkio$ taulukossa $A[j..\,oikea]$

- Tarkastelemme nyt silmukan viimeistä iteraatiota, jonka lopuksi siis $j \leq i$
- Väitämme, että j on sopiva jakokohta eli

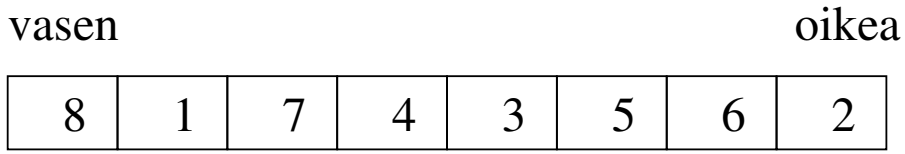
  $A[k] \leq jakoAlkio$, kun $vasen \leq k \leq j$, ja $A[k] \geq jakoAlkio$, kun $j + 1 \leq k \leq oikea$
- Kuten edellä,
  - Kun repeat j -silmukka on suoritettu on $A[k] \geq jakoAlkio$ kun $j < k \leq oikea$ ja $A[j] \leq jakoAlkio$
  - Kun repeat i -silmukka on suoritettu on $A[k] \leq jakoAlkio$ kun $vasen \leq k < i$ ja $A[i] \geq jakoAlkio$
- Siis väitteen jälkimmäinen osa pätee ilman muuta
- Koska $i \geq j$ ja $A[k] \leq jakoAlkio$, kun $vasen \leq k < i$, niin $A[k] \leq jakoAlkio$ pätee kun $k < j$. Lisäksi $A[j] \leq jakoAlkio$, joten myös väitteen ensimmäinen osa pätee

- Esimerkki **partition**-operaation toiminnasta:



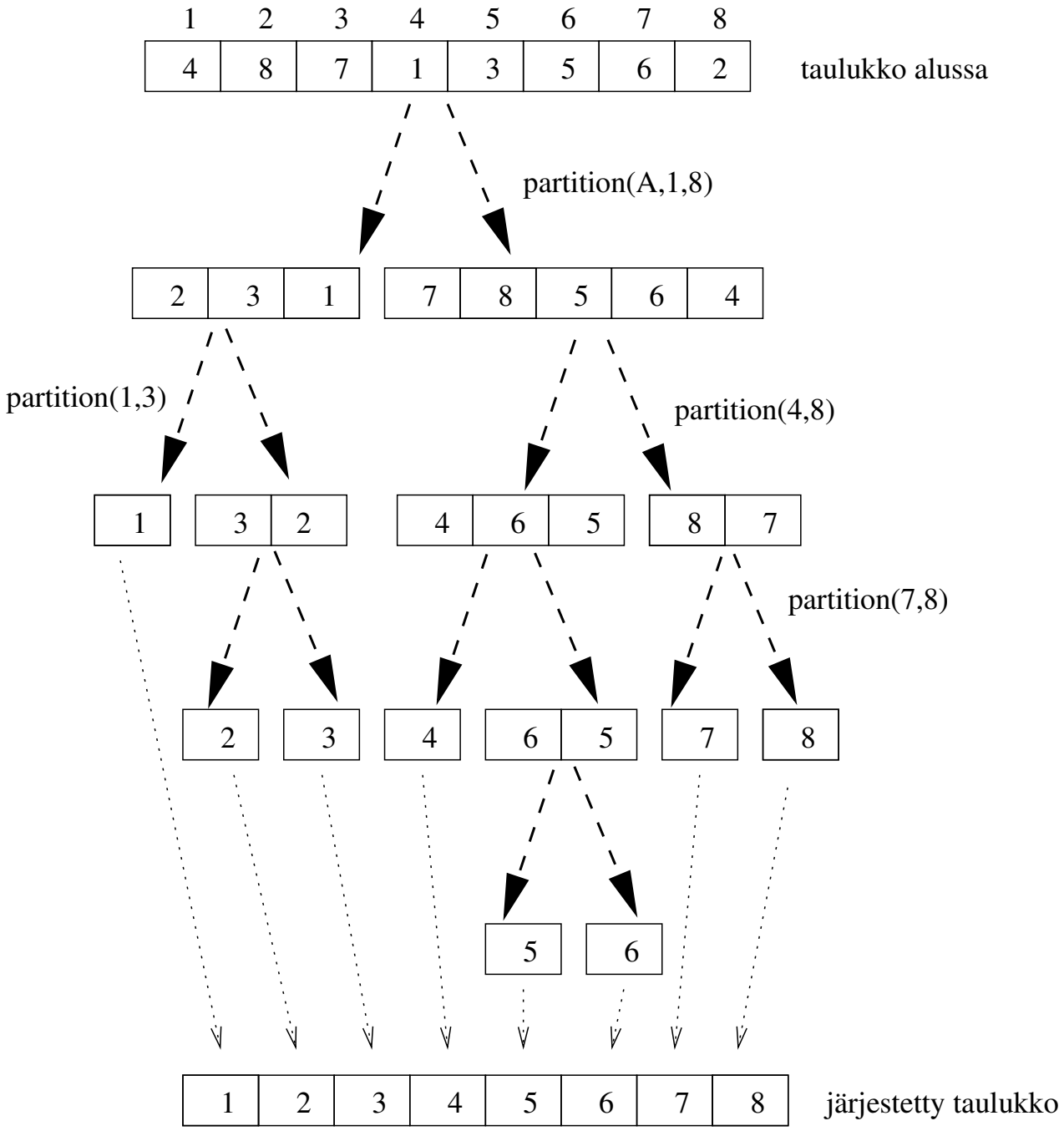
- Selvästi **partition**-operaation aikavaativuus on $\mathcal{O}(k)$ suhteessa käsiteltävän taulukon pituuteen k, aputilaa ei tarvita kuin parin muuttujan verran eli tilavaativuus on $\mathcal{O}(1)$

- Esimerkissämme meillä oli melko hyvä tuuri, taulukko jakautui kohtuullisen tasan kahtia, entä jos kyseessä olisi ollut seuraava taulukko?



- Entä mitä tapahtuu, jos taulukon kaikilla alkioilla on sama arvo?

- Seuraavalla sivulla esimerkki **pikajärjestämisen** toiminnasta


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

- Mikä on **pikajärjestämisen** aikavaativuus?
- Pahimmassa tapauksessa **partition** jakaa taulukon aina kahtia siten että toisessa osassa on vain 1 alkio
- Pahimman tapauksen vaativuus voidaan määritellä seuraavalla rekursioyhtälöllä:

$$T_w(k) = \begin{cases} c & \text{kun } k = 1 \\ T_w(1) + T_w(k-1) + ck & \text{kun } k > 1 \end{cases}$$

  missä c on vakio eli termi ck kuvaa **partition**-operaation vaativuutta k:n mittaiselle taulukon osalle
- Ratkaistaan $T_w(n)$ laskemalla auki rekursioyhtälöä:

$$\begin{aligned} T_w(n) &= T_w(1) + T_w(n-1) + cn \\ &= c + T_w(1) - T_w(n-2) + c(n-1) + cn \\ &= c + c + T_w(1) + T_w(n-3) + c(n-2) + c(n-1) + cn \\ &\dots \\ &= cn + c\sum_{i=1}^{n} i = cn + \frac{cn(n+1)}{2} = \mathcal{O}(n^2) \end{aligned}$$

- Pahimmassa tapauksessa **pikajärjestäminen** toimii neliöisesti, eli on selvästi huonompi kuin **lomitus**- ja myöhemmin esitettävä **kekojärjestäminen**

- Entä parhaassa tapauksessa, missä **partition**-operaatio sattuu aina jakamaan taulukon kahteen yhtäsuureen osaan?

- Parhaan tapauksen vaativuus voidaan määritellä seuraavalla rekursioyhtälöllä:

$$T_b(k) = \begin{cases} c & \text{kun } k = 1 \\ T_b(k/2) + T_b(k/2) + ck & \text{kun } k > 1 \end{cases}$$

- Yhtälö on täsmälleen sama kuin **lomitusjärjestämisen** vaativuutta kuvannut yhtälö, eli parhaan tapauksen aikavaativuus on $\mathcal{O}(n \log n)$

- Käytännössä hyvin toteutettu **pikajärjestäminen** toimii yleensä nopeammin kuin **lomitus**- tai **kekojärjestäminen**

- Huomaa kuitenkin, että tässä esitetty perusversio on hidas, jos taulukko on valmiiksi melkein järjestyksessä, mikä joissain sovelluksissa voi olla yleistä

- Osoittautuu, että keskimääräisen tapauksen vaativuus **pikajärjestämisellä** onkin $\mathcal{O}(n \log n)$. Analyysi löytyy Cormenista, mutta se ei kuulu tämän kurssin vaatimusten piiriin

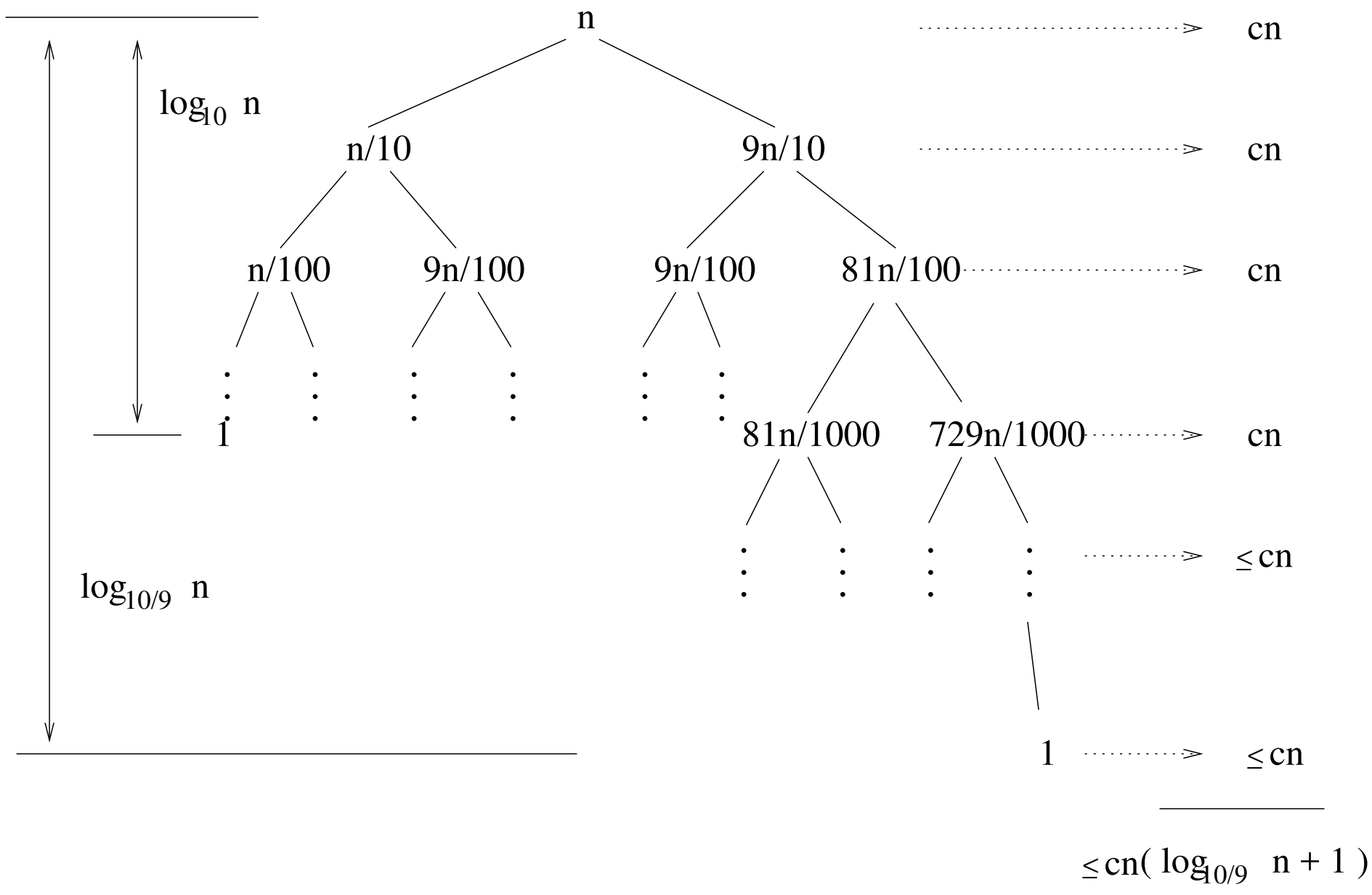
- Analysoidaan vielä tilannetta missä **partition** jakaa alkioita melko huonosti, eli oletetaan että k:n alkion taulukko jakautuisi pahimmillaan siten että pienemmässä osassa on vain $k/10$ alkiota ja suuremmassa $9k/10$

- Näytetään että jopa tässäkin tapauksessa **pikajärjestämisen** vaativuus olisi $\mathcal{O}(n \log n)$

- "Epätasapainoisten jakojen" tapauksen vaativuus $T_u(n)$ voidaan määritellä seuraavalla rekursioyhtälöllä:

$$T_u(k) = \begin{cases} c & \text{kun } k = 1 \\ T_u(k/10) + T_u(9k/10) + ck & \text{kun } k > 1 \end{cases}$$

- Kirjoitetaan rekursioyhtälöä auki rekursiopuumuodossa, solmuihin merkitty nyt rekursiokutsua vastaavan taulukonosan pituus


n
cn
$\log_{10}$ n
n/10
9n/10
cn
n/100
9n/100
9n/100
81n/100
cn
1
81n/1000
729n/1000
cn
$\log_{10/9}$ n
$\leq$ cn
1
$\leq$ cn
$\leq$ cn($\log_{10/9}$ n + 1)

- Saamme siis epätasapainoisten jakojen tapauksen aikavaativuudeksi $T_u(n) \leq cn(\log_{10/9} n + 1) = cn(\frac{\log_2 n}{\log_2 10/9} + 1) \leq cn(7 \cdot \log_2 n + 1) = 7 \cdot cn(\log_2 n \ + \frac{1}{7}) = \mathcal{O}(n \log n)$

- Näinkin epätasapainoinen partitiointi johtaa vielä $\mathcal{O}(n \log n)$ aikavaativuuteen, tosin rekursiotasoja tulee noin 7 kertaa niin monta kuin täysin tasapainoisissa jaoissa

- Yleisemmin jos jakosuhde on aina $\alpha : 1 - \alpha$ missä $0 < \alpha \leq 1/2$ on vakio, niin aikavaativuus on

$$\frac{1}{\log_2(1/(1-\alpha))} cn \log_2 n = \mathcal{O}(n \log n)$$

- Yleensä jakojen tasaisuus tietenkin vaihtelee. Jos esim. joka toinen jako on $\frac{1}{n} : \frac{n-1}{n}$ ja joka toinen $\frac{1}{10} : \frac{9}{10}$, niin edelliseen verrattuna rekursiopuuhun tulee kaksinkertainen määrä tasoja, joista joka toisella tasolla laskenta ei ''etene''

  Tämä tuottaa silti vain vakiokertoimen 2 aikavaativuuteen

- Yleisessä tapauksessa jakosuhteiden vaihtelu ei tietenkään ole näin helposti analysoitavissa. Kuitenkin aikavaativuus $\mathcal{O}(n^2)$ edellyttää, että huonoja jakoja tulee systemaattisesti paljon peräkkäin

  – Satunnaisesti valitulla syötteellä tämä on hyvin epätodennäköistä

- Koska **quicksort** ja **partition** vaativat vain vakiomäärän apumuuttujia, **pikajärjestämisen** tilavaativuus on verrannollinen rekursion maksimisyvyyteen

- Pahimmassa tapauksessa se on edellä esitetyllä toteutuksella $\mathcal{O}(n)$

- Tämä voidaan kuitenkin parantaa arvoon $O(\log n)$ toteutusta optimoimalla

- Ensimmäinen askel on korvata **quicksort**-kutsun päättävä rekursiivinen kutsu eli häntärekursio iteraatiolla:

  quicksort2(A,vasen,oikea)
  1 while vasen $<$ oikea
  2 jako = partition(A,vasen,oikea)
  3 quicksort2(A,vasen,jako)
  4 vasen = jako+1 // quicksort(A,jako+1,oikea) korvataan iteraatiolla

- Monet kääntäjät tekevät tällaisen optimoinnin automaattisesti

- Tämä ei vielä muuta pahimman tapauksen vaativuutta, koska pahimmassa tapauksessa rekursiivisesti käsiteltävä osataulukko $A[vasen\,..\,oikea]$ on edelleen kooltaan $n-1$

- Tilavaativuuteen $\mathcal{O}(\log n)$ päästään valitsemalla pienempi kahdesta osataulukosta rekursion kohteeksi. Suurempi jätetään iteraation käsiteltäväksi, mihin ei kulu lisätilaa:

```
quicksort3(A,vasen,oikea)
1   while vasen < oikea
2       jako = partition(A,vasen,oikea)
3       if jako − vasen < oikea − jako // alkuosa pienempi
4           quicksort3(A,vasen,jako)
5           vasen = jako+1   // quicksort(A,jako+1,oikea) korvataan iteraatiolla
6       else                                  // loppuosa pienempi
7           quicksort3(A,jako+1,oikea)
8           oikea = jako       // quicksort(A,vasen,jako) korvataan iteraatiolla
```

- Algoritmin optimoidun version toimintaperiaate ei ole välttämättä täysin ilmeinen, joten sen toimintaa kannattaa simuloida

- Nyt tilavaativuudeltaan pahin tapaus onkin tasajako, jolloin tilavaativuus toteuttaa seuraavan rekursioyhtälön:

$$\begin{aligned} S(1) &= c \\ S(n) &\leq S(n/2) + c \end{aligned}$$

- Sivulla 89 ratkaisimme saman rekursioyhtälön, ja päädyimme tulokseen $S(n) = \mathcal{O}(\log n)$, eli **quicksort3**:n tilavaativuus pahimmassa tapauksessa ja siis tietysti myös keskimäärin on $\mathcal{O}(\log n)$

Pikajärjestäminen käytännössä

- **Lisäysjärjestäminen** on lyömätön pienillä aineistoilla. **Pikajärjestämistä** kannattaakin viritellä siten että jos järjesteltävän taulukonosan pituus on enää esim. alle 20, järjestetäänkin tämä osa käyttäen **lisäysjärjestämistä**

  quicksort-tuned(A,vasen,oikea)
  1 if oikea $-$ vasen $<$ 20 insertion-sort(A,vasen,oikea)
  2 else
  3 jako = partition(A,vasen,oikea)
  4 quicksort-tuned(A,vasen,jako)
  5 quicksort-tuned(A,jako+1,oikea)

- **partition**-operaatiossa paras valinta jakoalkioksi olisi jaettavan aineiston mediaani, eli suuruudeltaan puolessa välissä oleva alkio. Tällöin jako olisi aina tasainen ja aikavaativuus pahimmassa tapauksessa $\mathcal{O}(n \log n)$, mutta mediaanin selvittäminen ei käytännössä ole tehokasta (joskin mahdollista ajassa $\mathcal{O}(()n)$ ns. BFPRT-algoritmilla)

- Melko hyvä keino on valita jakoalkioksi kolmen mediaani eli mediaani arvoista $A[vasen]$, $A[\lfloor (vasen + oikea)/2 \rfloor]$ ja $A[oikea]$

- Pahinta tapausta (siis jolloin aikaa kuluu $\mathcal{O}(n^2)$) **pikajärjestämisessä** ei voida täysin välttää käyttämällä kolmen mediaania jakoalkion valintaan, mutta pahimman tapauksen esiintymistodennäköisyys on hyvin pieni

Alaraja vertailuihin perustuvalle järjestämiselle

- Edellä esitetyt järjestämisalgoritmit ovat kaikki vertailuihin perustuvia: ne käsittelevät järjestettäviä arvoja vain
  - testaamalla järjestysehtoja $A[i] < A[j]$, $A[i] = A[j]$, $A[i] > A[j]$ jne.
  - vaihdoilla $A[i] \leftrightarrow A[j]$ ja yleisemmin sijoituksilla $x = A[i]$ jne.

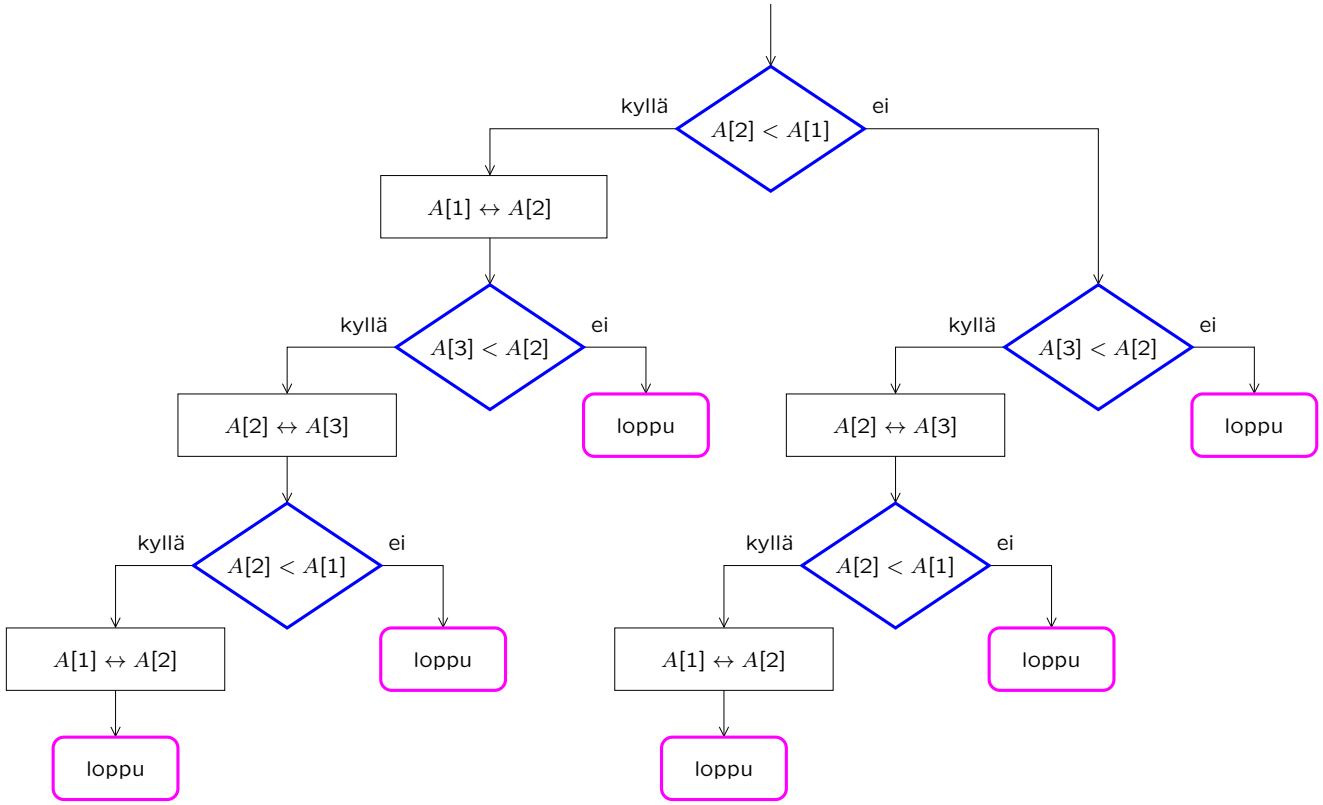
- Esimerkki muusta kuin vaihtoihin perustuvasta algoritmista voisi olla sellainen, joka olettaa alkioiden olevan kokonaislukuja ja esim. käyttää keskiarvoa $(A[1] + A[n])/2$ tai testaa, onko $A[i]$ parillinen tai laskee taulukossa A esiintyvien arvojen lukumääriä

- Vertailuihin perustuva järjestämisalgoritmi suorittaa pahimmassa tapauksessa $\Omega(n \log n)$ vertailua

  (Muistinvirkistys: merkinnällä Ω tarkoitetaan funktion kasvunopeuden alarajaa)

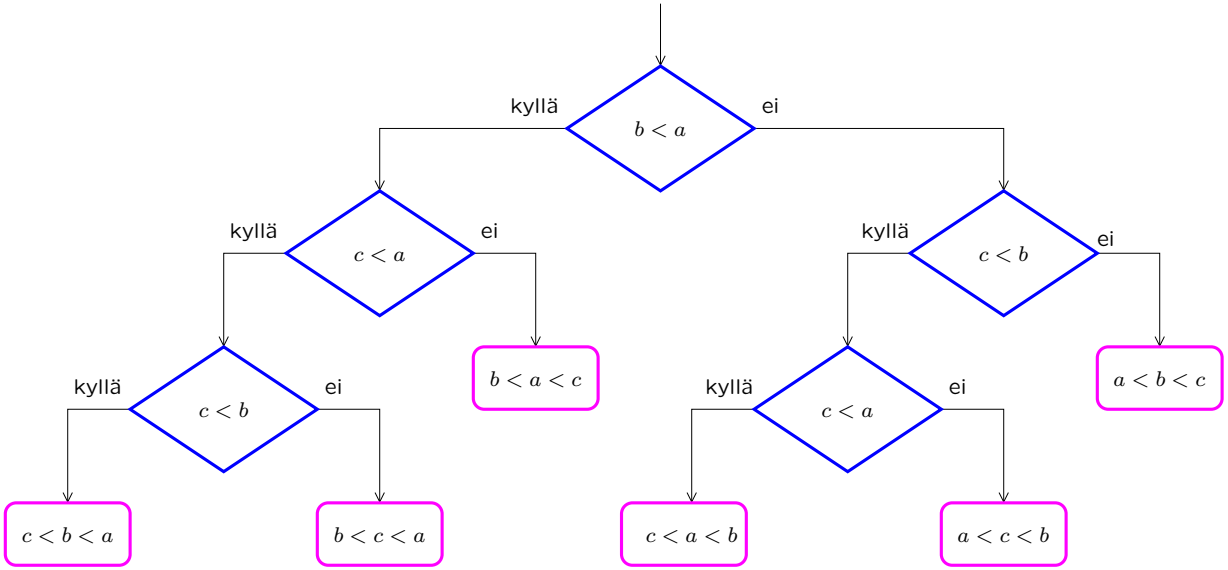
- Lähtökohtana on jokin vertailuihin perustuva järjestämisalgoritmi jollekin kiinteälle syötteen koolle n

- Alla on esimerkkinä vuokaavio **lisäysjärjestämiselle**, kun $n = 3$



- Saamme vuokaavioesityksestä päätöspuun, kun jätämme sijoitusoperaatiot pois ja merkitsemme muuttujien $A[1]$, $A[2]$ ja $A[3]$ alkuperäisiä arvoja a, b ja c

- Testien päätyttyä nähdään, mikä on alkioiden a, b ja c järjestys



- Yksinkertaisuuden vuoksi oletetaan, että alkiot ovat erisuuria

- Yleisesti järjestämisalgoritmia vastaavassa päätöspuussa
  - sisäsolmuina (haarautumakohtina) on ehtotestejä
  - kussakin haarautumassa on kaksi vaihtoehtoa
  - lehtinä (haarojen päätepisteinä) on n alkion järjestyksiä (permutaatioita)
  - puun korkeus on pahimmassa tapauksessa tehtävien vertailujen lukumäärä
- Jotta algoritmi toimisi oikein, jokaiselle syötteen järjestykselle pitää olla (ainakin) yksi lehti
- Siis lehtiä on ainakin $n!$
- Koska haarautumissa on aina kaksi vaihtoehtoa, puun tasolla k (alkukohdasta eli juuresta lukien) on korkeintaan 2^k haaraa
- Siis $n!$ lehden saamiseksi aikaan puun haarautumista pitää jatkaa ainakin syvyydelle (noin) $\log_2 n!$
- Tämä on siis samalla alaraja algoritmin pahimman tapauksen aikavaativuudelle

- Teemme nyt hyvin karkean arvion

$$\begin{aligned}
\log_2(n!) &= \log_2(n(n-1)(n-2)\cdots 3\cdot 2\cdot 1) \\
&= \log_2 n + \log_2(n-1) + \ldots + \log_2 3 + \log_2 2 + \log_2 1 \\
&= \sum_{k=1}^{n} \log_2 k \\
&\geq \sum_{k=\lceil n/2\rceil}^{n} \log_2 k \\
&\geq \lceil n/2\rceil \log_2 \lceil n/2\rceil
\end{aligned}$$

- Siis mikä tahansa vertailuihin perustuva järjestämisalgoritmi tekee pahimmassa tapauksessa ainakin $(n/2)\log_2(n/2) = \Omega(n \log n)$ vertailua

- Siis **kekojärjestämisen** ja **lomitusjärjestämisen** aikavaativuudet ovat vakiokerrointa vaille optimaalisia, kun rajoitutaan vertailuihin perustuviin algoritmeihin

Järjestäminen lineaarisessa ajassa

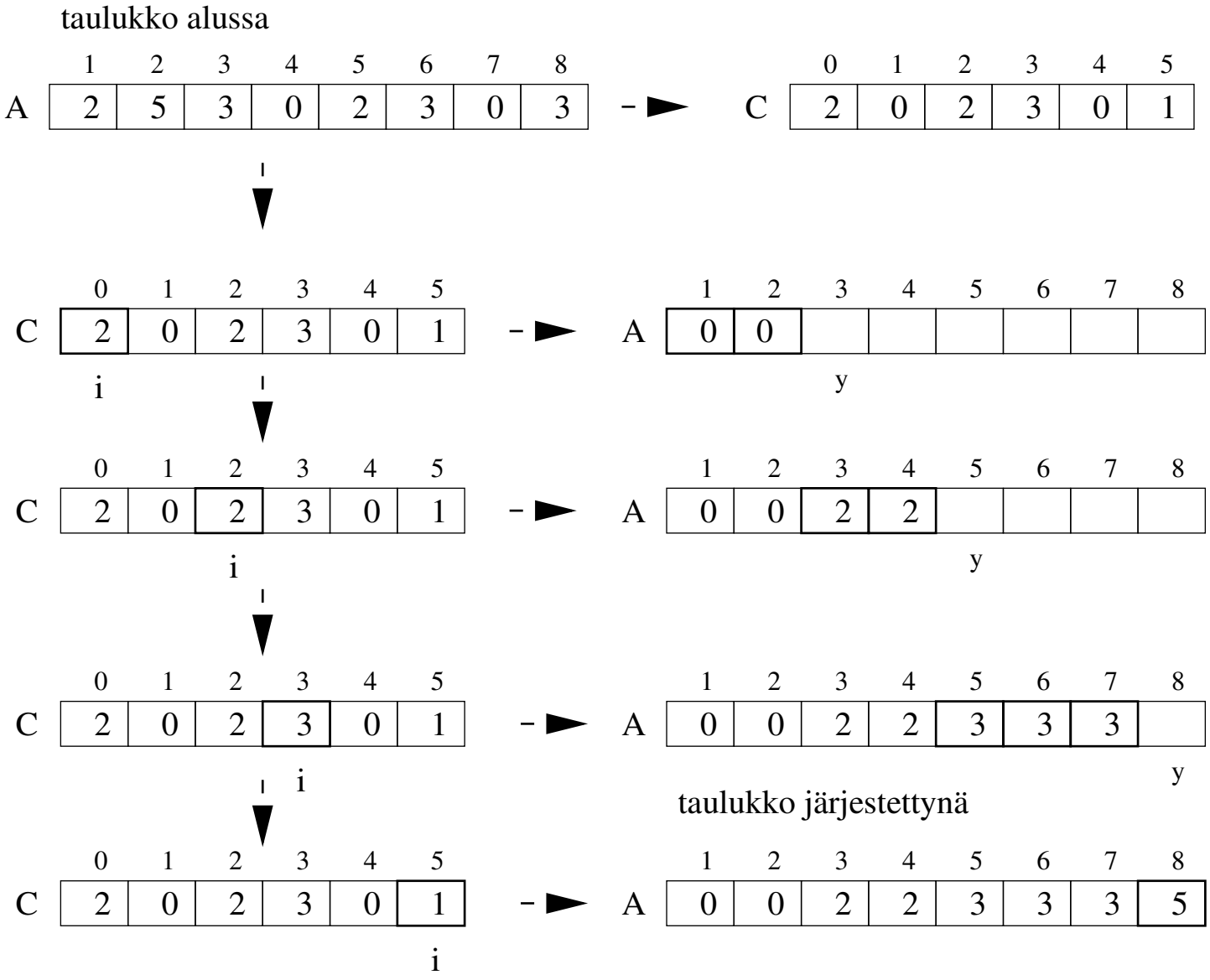
- Tehokkuusraja $\mathcal{O}(n \log n)$ voidaan rikkoa, jos järjestäminen perustuu johonkin muuhun kuin alkioiden keskinäiseen vertailuun
- Oletetaan että järjestettävä aineisto $A[1, n]$ koostuu luvuista joiden arvo on väliltä $0, \ldots, k$
- Yksinkertainen ja tehokas järjestämismenetelmä saadaan aikaan seuraavasti:
  - otetaan käyttöön aputaulukko $C[0, k]$
  - käydään A läpi ja lasketaan kuinka monta kertaa kukin luku esiintyy; luvun x esiintymien määrä talletetaan paikkaan $C[x]$
  - sijoitetaan taulukkoon A ensin $C[0]$ kertaa luku 0, $C[1]$ kertaa luku 1 jne
  - näin taulukossa on samat luvut kuin alussa ja luvut ovat suuruusjärjestyksessä

- Algoritmina:

```
counting-sort1(A,k,n)
1   for i = 0 to k C[i]= 0
2   for j = 1 to n
3       x = A[j]
4       C[x] = C[x]+1
5   y = 1
6   for i = 0 to k
7       for j = 1 to C[i]
8           A[y] = i
9           y = y+1
```

- Kutsutaan algoritmia laskemisjärjestämiseksi (engl. counting sort). Kyseessä ei kuitenkaan ole sama versio laskemisjärjestämisestä kuin mikä löytyy Cormenista

- Esimerkki:


taulukko alussa
A
1 2 3 4 5 6 7 8
2 5 3 0 2 3 0 3
C
0 1 2 3 4 5
2 0 2 3 0 1
i
y
taulukko järjestettynä

- Algoritmi käy kerran läpi taulukon A, kahteen kertaan taulukon C ja tulostaa luvun jokaiseen A:n paikkaan, aikavaativuus siis $\mathcal{O}(n+k)$ ja jos $k = \mathcal{O}(n)$ niin aikavaativuus on lineaarinen järjestettävän aineiston koon suhteen

- Tilavaativuus on luonnollisesti $\mathcal{O}(k)$

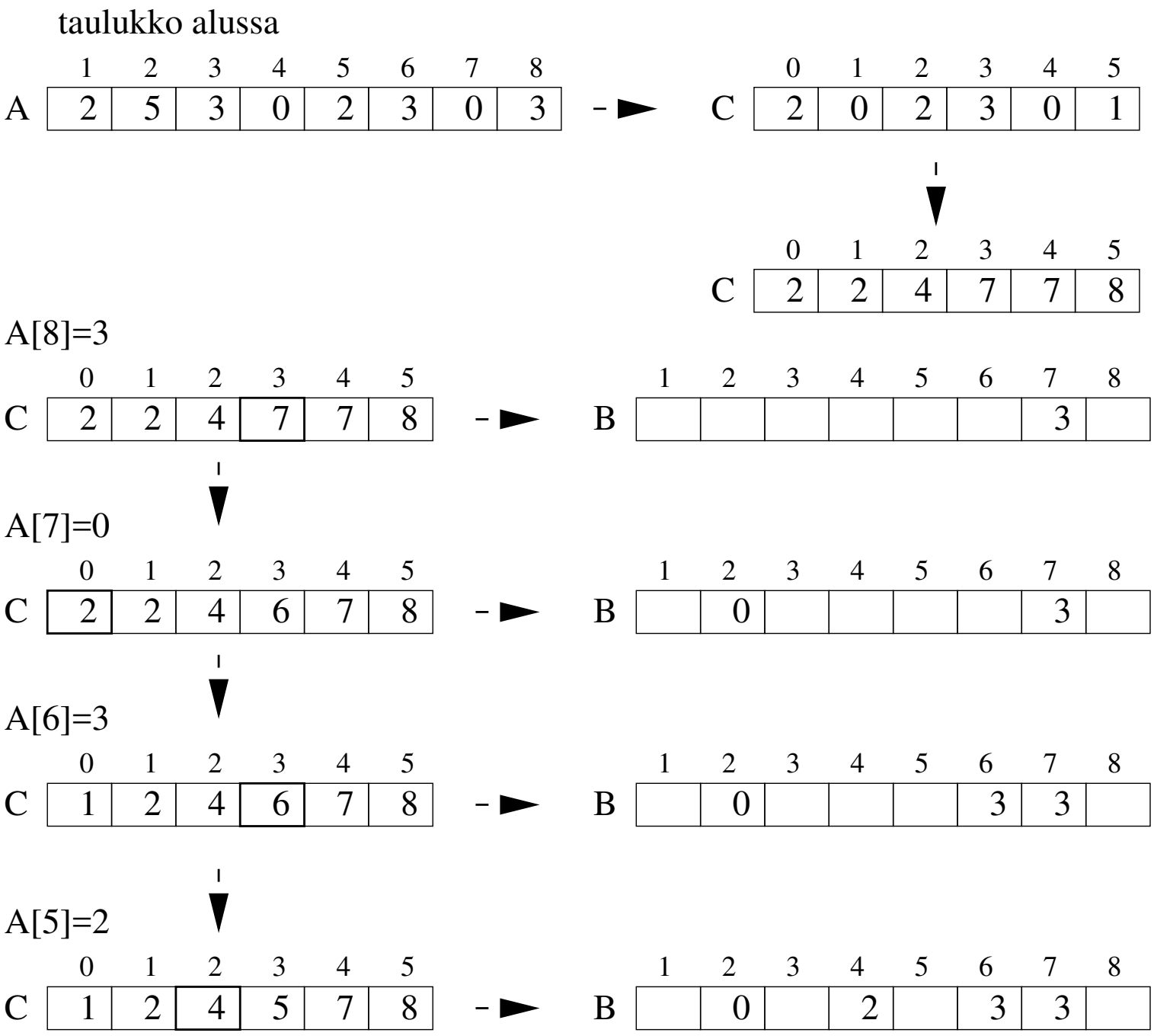
- Jos järjestettäviin alkioihin liittyy muita datakenttiä, juuri esitetty versio **laskemisjärjestämisestä** ei toimi

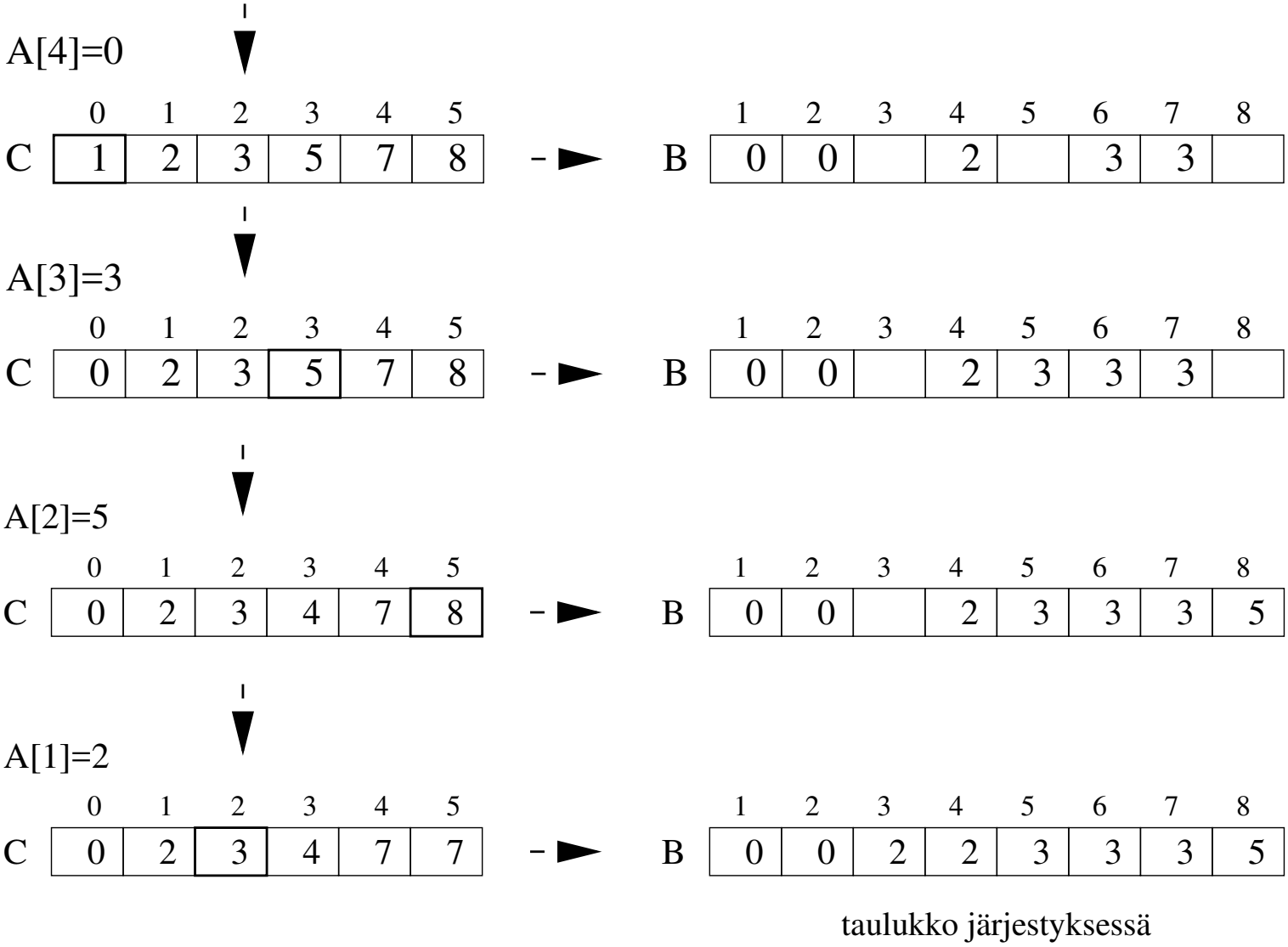
- Esitetään vielä **laskemisjärjestämisestä Cormenin** versio, joka välttää yllä mainitun ongelman

```
counting-sort2(A,k,n)
1   for i = 0 to k C[i]= 0
2   for j = 1 to n
3       x = A[j]
4       C[x] = C[x]+1
5   for i = 1 to k
6       C[i] = C[i]+C[i-1]
7   for j = n downto 1
8       x = A[j]
9       B[C[x]]= x
10      C[x] = C[x]-1
11  for i = 1 to n A[i] = B[i]
```

- Toimintaidea:
  - algoritmi käyttää aputaulukkoja $B[1, n]$ ja $C[0, k]$
  - rivien 3-4 for-lauseen jälkeen $C[i]$ sisältää tiedon kuinka monta lukua i taulukossa A on
  - rivien 5-6 for-lauseen jälkeen $C[i]$ sisältää tiedon kuinka monta korkeintaan yhtä suurta lukua kuin i taulukossa A on

- Toimintaidea, jatkuu:
  - rivien 7-10 for-lause järjestää A:n alkiot taulukkoon B
  - laitetaan ensin paikalleen paikan $A[n]$ alkio x
  - $C[x]$ kertoo kuinka monta korkeintaan x:n suuruista alkiota taulukossa A on
  - x on siis $C[x]$:nneksi suurin A:n alkioista, eli laitetaan x paikkaan $B[C[x]]$
  - vähennetään vielä arvoa $C[x]$ yhdellä jotta seuraava paikalleen laitettava saman suuruinen alkio menee oikealle paikalleen
  - jatketaan laittamalla paikoilleen alkio $A[n-1]$
  - lopuksi kopioidaan järjestetyt alkiot taulukosta B takaisin taulukkoon A
- Algoritmi käy kahteen kertaan läpi molemmat taulukot A ja C sekä kertaalleen läpi taulukon B, aikavaativuus siis $\mathcal{O}(n+k)$
- Aputaulukon B koko on n ja aputaulukon C koko on k eli tilavaativuus $\mathcal{O}(n+k)$
- Edelleen, jos $k = \mathcal{O}(n)$, on molempina vaativuuksina $\mathcal{O}(n)$
- Esimerkki **Cormenin laskemisjärjestämisen** toiminnasta seuraavilla sivuilla:


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 2 4 7 7 8
A[8]=3
0 1 2 3 4 5
C 2 2 4 7 7 8
1 2 3 4 5 6 7 8
B 3
A[7]=0
0 1 2 3 4 5
C 2 2 4 6 7 8
1 2 3 4 5 6 7 8
B 0 3
A[6]=3
0 1 2 3 4 5
C 1 2 4 6 7 8
1 2 3 4 5 6 7 8
B 0 3 3
A[5]=2
0 1 2 3 4 5
C 1 2 4 5 7 8
1 2 3 4 5 6 7 8
B 0 2 3 3



taulukko järjestyksessä

Reikäkorttijärjestäminen (radix sort)

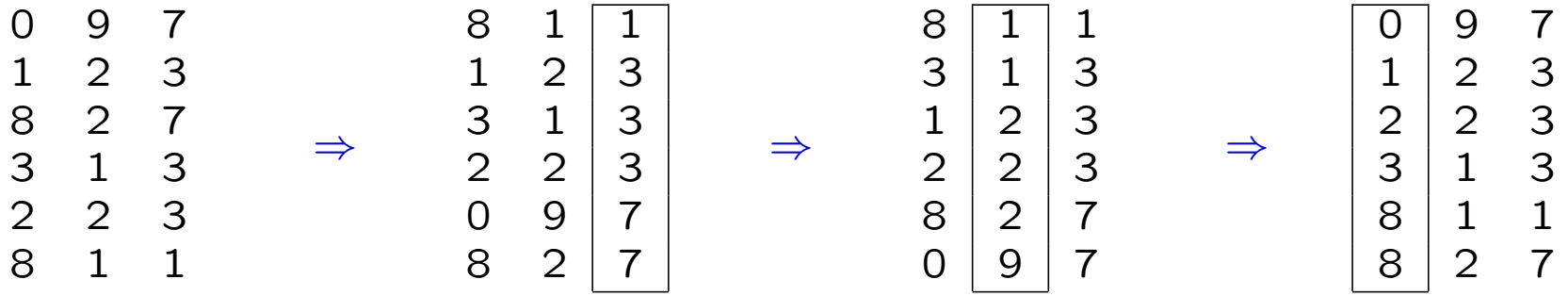
Tarkastellaan esimerkkinä kolmenumeroisten positiivisten kokonaislukujen järjestämistä. Tämä voidaan tehdä seuraavasti:

- Järjestä luvut vähiten merkitsevän numeron mukaan jollain vakaalla algoritmilla (esim. laskemisjärjestäminen)
- Toista sama keskimmäisen numeron mukaan.
- Lopuksi järjestä eniten merkitsevän mukaan.

Kun kunkin vaiheen järjestäminen tehdään vakaasti, lopputuloksena luvut tulevat oikeaan järjestykseen (vrt. s. 126).

Tarvittaessa lisätään etunollia niin, että luvut ovat saman mittaisia.

Esimerkki Järjestetään luvut 97, 123, 827, 313, 223, 811



Yleisemmin olkoon järjestettävänä n avainta $A_1, \ldots, A_n$, missä kukin A_i on m-ulotteinen vektori ja komponenteilla $A_i[j]$ on k mahdollista arvoa, joille on määritelty järjestys.

- Edellisessä esimerkissä $n = 6$, $m = 3$ ja $k = 10$.
- Tätä voidaan soveltaan m-merkkisten merkkijonojen järjestämiseen, jolloin (esim.) $k = 256$.

Aikavaativuudeksi saadaan $O(m(n + k))$:

- Yksittäisen ”sarakkeen” mukaan järjestäminen menee ajassa $O(n + k)$ laskemisjärjestämisellä
- ”Sarakkeita” on m kappaletta.

Yhteenveto järjestämisalgoritmeista

- Aikavaativuus

Aikavaativuus	pahin tapaus	keskim. tapaus	paras tapaus	vakaa
kupla	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	on
lisäys	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	on
lomitus	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	on
keko	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$[a]	ei
pika	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	ei
laskemis	$\mathcal{O}(n + k)$	$\mathcal{O}(n + k)$	$\mathcal{O}(n + k)$	on

[a]Tämä olettaa, että järjestettävänä on n erisuurta alkiota. Jos kaikki alkiot ovat yhtäsuuria, aikavaativuus on $\mathcal{O}(n)$.

- Tilavaativuus
  - **lomitusjärjestäminen** $\mathcal{O}(n)$
  - **pikajärjestäminen** $\mathcal{O}(\log n)$
  - **laskemisjärjestäminen** $\mathcal{O}(n + k)$
  - **muut** $\mathcal{O}(1)$

Järjestämisalgoritmin valinta

- Käytännössä viritelty **pikajärjestäminen** eli lyhyisiin taulukonosiin **lisäysjärjestämistä** käyttävä versio on nopeiten toimiva järjestämisalgoritmi

- Ohjelmointikielten kirjastototeutukset perustuvat useimmiten viriteltyyn **pikajärjestämiseen**

- **Pikajärjestämisen** keveys esim. **lomitusjärjestämiseen** verrattuna johtuu siitä, että tulosten yhdistämisvaihetta ei tarvita ja alkioiden jaon suorittava **partition** on suhteellisen kevyt operaatio, sillä se käy jaettavat alkiot läpi ainoastaan kertaalleen

- Järkevästi toteutetussa **pikajärjestämisessä** pahin tapaus eli neliöisen suoritusajan tuottava tapaus on käytännössä erittäin harvinainen

- Jos on erittäin tärkeää, että pahin tapaus ei toteudu koskaan, esim. turvallisuuskriittisissä järjestelmissä, joissa vasteaikojen on oltava kaikissa tapauksissa ennustettavat, kannattaa käyttää **lomitusjärjestämistä** tai **kekojärjestämistä**, sillä ne ovat pahimmassakin tapauksessa ajassa $\mathcal{O}(n \log n)$ toimivista algoritmeista käytännössä nopeimmat

- Jos **pikajärjestämisen partition**-operaatio onnistuisi valitsemaan jakoalkioksi tarkasteltavan taulukonosan alkioiden mediaanin, olisivat jaot aina tasaisia, eikä pahin tapaus $\mathcal{O}(n^2)$ koskaan toteutuisi
  - yllättäen osoittautuu, että k:n kokoisesta taulukosta on mahdollista löytää mediaani ajassa $\mathcal{O}(k)$, eli **pikajärjestäminen** on mahdollista toteuttaa toimimaan pahimmassakin tapauksessa ajassa $\mathcal{O}(n \log n)$
  - mediaanin selvittäminen lineaarisessa ajassa on kuitenkin käytännössä niin hidas operaatio, että sen käyttäminen **pikajärjestämisen** yhteydessä huonontaisi algoritmin toimintaa liian paljon

- Vakautta edellytettäessä eivät **pika**- ja **kekojärjestäminen** kelpaa, vaan valinta on **lomitusjärjestäminen**

- **Lomitusjärjestäminen** on kehittyneimmistä järjestämisalgoritmeista suoritusajaltaan ennustettavin, sillä se toimii käytännössä kaikenmuotoisilla syötteillä täysin samalla tavalla.

- Javan valmiista järjestämisalgoritmeista taulukoita järjestävä `Arrays.Sort` perustuu **pikajärjestämiseen**, kokoelmia järjestävä `Collections.Sort` taas **lomitusjärjestämiseen**

- **Laskemisjärjestäminen** näyttää lineaarisine aikavaativuuksineen houkuttavalta valinnalta. Käytännössä ongelmaksi osoittautuu järjestettävien alkioiden arvoalueen suuri koko. Jos järjestettävänä on esim. max 20 merkin mittaisia merkkijonoja, arvoalueen on koko luokkaa $(2 \cdot 27)^{20}$, eli **laskemisjärjestäminen** on tilanteessa täysin käyttökelvoton

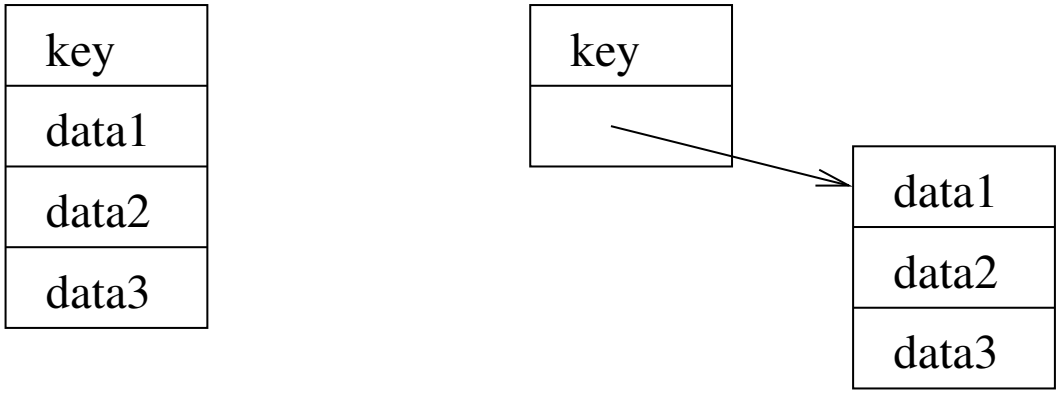
- Jos sovellus koostuu esim. kymmenestätuhannesta pienestä taulukosta jotka on järjestettävä, on paras valinta **lisäysjärjestäminen**

- **Lisäysjärjestäminen** toimii myös parhaiten isoillakin taulukoilla, jos taulukot ovat melkein järjestyksessä

4. Perustietorakenteet: pino, jono ja lista

Abstrakti tietotyyppi joukko

- Usein ohjelma ylläpitää suoritusaikanaan jotain joukkoa tietoalkioita, esim. puhelinluettelo nimi-numeropareja
- Joukon tietoalkiot muodostuvat olioista missä yksi kenttä on ns. avain, jonka perusteella tietueita voidaan hakea ja johon perustuen tietoalkiot voidaan järjestää; esim. puhelinluettelossa nimi on avain
- Muu tieto voi olla olion muissa kentissä tai talletettuna johonkin viitteen takana olevaan olioon



- Jälkimmäinen vaihtoehto on järkevämpi varsinkin silloin, jos tietoalkioilla on kaksi eri avainta: esim. puhelinluettelosta pitää pystyä tekemään hakuja sekä nimen että numeron perusteella . . .

- Useimmiten joukolle (merkitään S:llä) tarvitaan ainakin seuraavia operaatioita:

  - **search**(S, k) jos joukosta löytyy avaimella k varustettu alkio, operaatio palauttaa viitteen alkioon, muuten operaatio palauttaa viitteen NIL
  - **insert**(S, x) lisää joukkoon alkion johon x viittaa
  - **delete**(S, x) poistaa tietueen johon x viittaa

- Joskus ovat tarpeen myös seuraavat operaatiot:

  - **min**(S) palauttaa viitteen joukon pienimpään alkioon
  - **max**(S) palauttaa viitteen joukon suurimpaan alkioon
  - **succ**(S, x) palauttaa viitteen x:ää seuraavaksi suurimpaan alkioon; jos x:n avain oli suurin palauttaa NIL (x on viite)
  - **pred**(S, x) palauttaa viitteen x:ää seuraavaksi pienempään alkioon; jos x:n avain oli pienin palauttaa NIL (x on viite)

- Huomaamme että sivulla 30 mainituista puhelinluettelon operaatioista suurin osa on suoraan tarjolla abstraktin tietotyypin joukko-operaationa

- Operaatio, joka tulostaa nimi-numeroparit aakkosjärjestyksessä, saadaan myös helposti toteutetuksi joukon operaatioiden avulla kysymällä ensin pienin alkio operaatiolla **min** ja sen jälkeen seuraavat järjestyksessä kutsumalla toistuvasti **succ**-operaatiota

- Puhelinluettelossa on kaksi etsimisoperaatiota findNumber ja findName, eli etsi numero nimen perusteella ja etsi nimi nimeron perusteella

- Joukossa on kuitenkin vain yksi etsintäoperaatio, **search**, joka etsii tietyllä avaimella varustetun alkion

- Yhden joukon avulla saadaankin puhelinluettelosta tehtyä vain versio, jossa on joko findNumber tai findName "tehokkaasti" toteutettuna, muttei molempia

- Jos valitaan avaimeksi nimi, niin operaatio findNumber, eli etsi tietyn henkilön puhelinnumero, saadaan tehokkasti toteutettua joukon avulla

- Jos halutaan molemmat operaatiosta findNumber ja findName tehokkaaksi, tarvitaankin kaksi joukkoa (toisessa avaimena nimi, toisessa numero)

- Palaamme aiheeseen muutaman viikon päästä

- Edellä määritellyillä operaatiolla varustettu joukko on abstrakti tietotyyppi, jonka toteuttamiseen on useita vaihtoehtoisia tietorakenteita:
  - taulukko
  - linkitetty lista
  - hakupuu
  - tasapainotettu hakupuu
  - hajautusrakenne
  - keko
- Joukon operaatioiden tehokkuus riippuu suuresti siitä minkä tietorakenteen avulla joukko on toteutettu, ks. seuraava sivu
- Toteutustavan valinnassa siis on ratkaisevaa se mitä operaatioita joukkoa käyttävä sovellus tarvitsee eniten
- Kurssin seuraavien viikkojen ohjelmassa on käsitellä joukon toteutukseen sopivia tietorakenteita

- Operaatioiden aikavaativuus riippuen käytetystä tietorakenteesta:

	taulukko	**järj. taulukko**	**lista**	**järj. lista**	**tasap. puu**	**haj.-taulu**
search	O(n)	O(log n)	O(n)	O(n)	O(log n)	keskim. O(1)
insert	O(1)	O(n)	O(1)	O(n)	O(log n)	O(1)
delete	O(n)	O(n)	O(1)	O(1)	O(log n)	O(1)
min	O(n)	O(1)	O(n)	O(1)	O(log n)	O(n)
max	O(n)	O(1)	O(n)	O(1)	O(log n)	O(n)
succ	O(n)	O(1)	O(n)	O(1)	O(log n)	O(n)
pred	O(n)	O(1)	O(n)	O(1)	O(log n)	O(n)

  lista tarkoittaa tässä kahteen suuntaan linkitettyä järjestämätöntä listaa, järj.taulukko tarkoittaa taulukkoa jossa alkiot on talletettu suuruusjärjestyksessä ja järj.lista linkitettyä rengaslistaa jossa alkiot on talletettu suuruusjärjestyksessä

- Tietorakenteen johon alkioita talletetaan ei välttämättä aina tarvitse olla matemaattisessa mielessä joukko, eli sama alkio voi olla mielekästä pystyä tallentamaan tietorakenteeseen moneen kertaan

- Tietorakennetta jossa voi olla monta samaa alkiota voidaan kutsua esim. kokoelmaksi (engl. collection)

Javan valmiit tietorakennetoteutukset

- Java API:sta löytyy useita lähes edellä määriteltyä joukkoa vastaavia valmiita tietorakenneratkaisuja, esim.:
  - `ArrayList`, dynaaminen taulukko
  - `LinkedList`, linkitetty lista
  - `TreeSet` ja `TreeMap`, tasapainotettu hakupuu
  - `HashSet` ja `HashMap`, hajautustaulu
- Osaa Javan valmiista kalustosta on jo käytetty Ohjelmoinnin jatkokurssilla
- Tällä kurssilla jatkamme Javan tietorakenneluokkiin tutustumista ja tavoitteena on, että kurssin jälkeen kaikki tuntevat Javan valmiiden ratkaisujen takana olevat toteutusperiaatteet niin hyvin, että optimaalisen tietorakenneratkaisun valinta kulloiseenkin sovellustilanteeseen on helppoa

Pino ja jono

- Ennen kuin menemme varsinaiseen lista-tietorakenteeseen, opiskelemme kahta joukon erikoistapausta, pinoa (engl. stack) ja jonoa (engl. queue)

- Pino ja jono eivät toteuta sivun 165 joukko-operaatioita, vaan toimivat niinkuin pino ja jono toimivat normaalielämässä

- Pino ja jono ovat tärkeitä rakenneosasia monissa algoritmeissa, kuten tulemme myöhemmin näkemään

Pino

- Last-in-first-out -periaate, eli toimii kuten Unicafessa lautaspino

- Pino-operaatiot:

  - **push**(S, x) laittaa tietoalkion (johon x viittaa) pinon päälle
  - **pop**(S) palauttaa viitteen pinon päällimmäiseen tietoalkioon ja poistaa sen pinosta
  - **empty**(S) palauttaa true jos pino tyhjä, muuten false

- Oletetaan seuraavassa yksinkertaisuuden vuoksi että pinoon talletettavat alkiot sisältävät ainoastaan kentän key, eli ovat esim. kokonaislukuja

- Eli operaation parametrit yksinkertaistetussa tapauksessa:

  - **push**(S, k) laittaa tietoalkion k pinon päälle
  - **pop**(S) palauttaa pinon päällimmäisen tietoalkion

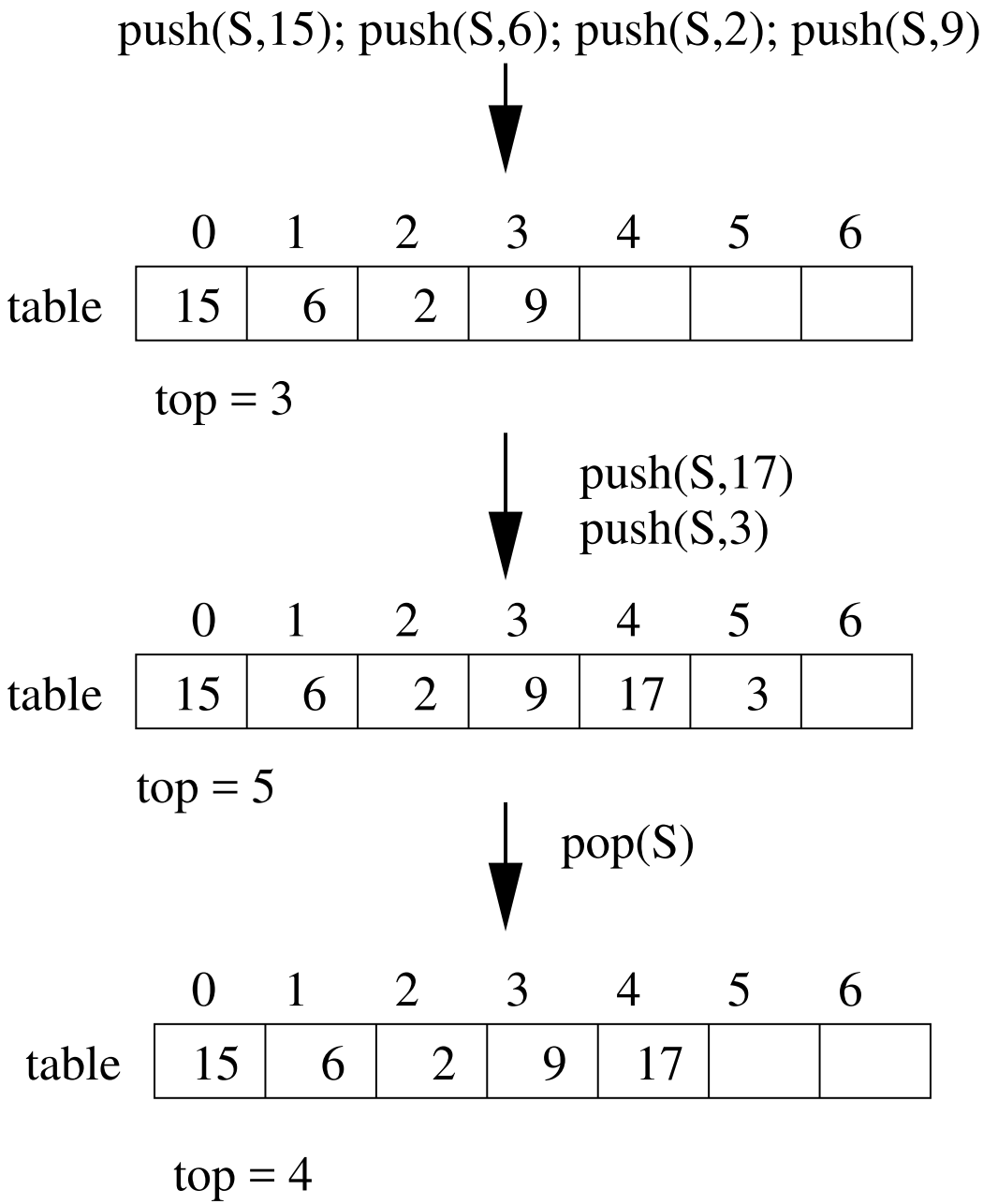
- Jos yläraja pinon koolle tiedetään ennakolta, on pino helppo toteuttaa taulukon avulla
- Esitetään seuraavassa pino pseudokoodin sijaan Javalla
- Määritellään pino luokkana, jolla attribuuttina taulukko `table`, johon alkiot talletetaan (tällä kertaa kokonaislukuja) sekä attribuutti `top`, joka kertoo missä kohtaa taulukkoa on pinon huippu

```
class Stack {
   int[] table;
   int top;
```

- Pinon koko annetaan konstruktorin parametrina. Konstruktori luo taulukon sekä alustaa `top`-attribuutin arvoon -1, joka kertoo, että pinossa ei ole vielä alkioita

```
   Stack(int n){
      top = -1;
      table = new int[n];
   }
```

- Talletetaan pinon alkiot taulukkoon `table[0, ..., n-1]`
- Ensimmäinen alkio laitetaan paikkaan `table[0]`, toinen paikkaan `table[1]`, jne
- `top` siis kertoo huipulla olevan alkion paikan taulukossa


push(S,15); push(S,6); push(S,2); push(S,9)
0 1 2 3 4 5 6
table
15 6 2 9
top = 3
push(S,17)
push(S,3)
0 1 2 3 4 5 6
table
15 6 2 9 17 3
top = 5
pop(S)
0 1 2 3 4 5 6
table
15 6 2 9 17
top = 4

- Operaatioiden toteutus:

```
int pop(){
   int pois = table[top];
   top--;
   return pois;
}

void push(int x){
   top++;
   table[top] = x;
}

boolean empty(){
   return top==-1;
}
```

- Toteutus olettaa, että ennen **pop**-operaation käyttämistä ohjelma testaa että pino ei ole tyhjä

- Kaikki pino-operaatiot sisältävät vain saman vakiomäärän komentoja riippumatta siitä, kuinka monta alkiota pinossa on, operaatiot ovat siis vaativuudeltaan $\mathcal{O}(1)$

- Myös operaatioiden tilavaativuus on $\mathcal{O}(1)$. Tilavaativuudellahan tarkoitettiin operaation aikana käytettyjen apumuuttujien määrää.

  Vaikka pinossa olisikin paljon tavaraa, niin operaatioiden aikana ei apumuuttujia tarvita kuin **pop**:issa, josta siitäkin on tarvittaessa helppo päästä eroon

- Täyteen pinoon alkion laittaminen aiheuttaa ylivuodon (engl. overflow) ja alkion ottaminen tyhästä pinosta aiheuttaa alivuodon (engl. underflow): Javassa poikkeuksen `ArrayIndexOutOfBoundsException`

- On sovelluskohtaista miten tilanteeseen kannattaa varautua

- Yksi mahdollisuus on tietysti hoitaa poikkeus `try-catch`-mekanismin avulla

- Ehkä järkevämpää on toteuttaa metodi, jolla voidaan varmistua, että pino ei ole täysi:

```
boolean full(){
   return top==table.length-1;
}
```

- On myös mahdollista tehdä pinon taulukosta tarpeen mukaan kasvava, eli jos **push**-operaatio toteaa taulukon olevan täynnä, se luo uuden taulukon johon alkuperäiset alkiot kopioidaan

  tämä kuitenkin tekee **push**:in pahimman tapauksen aikavaativuudesta lineaarisen pinon koon suhteen

- Jos pinon tallennusalue toteutettaisiin `ArrayList`:in avulla, tilanne on sama sillä `ArrayList` joutuu tekemään raskaan tilankasvatusoperaation

- Tarkemmassa analyysissä osoittautuu, että jos pinon koko aina kaksinkertaistetaan kasvatustilanteessa, niin vaikutus aikavaativuuteen ei ole loppujenlopuksi kovin dramaattinen:
  - yksittäinen **push**-operaatio vie pahimmassa tapauksessa aikaa $\mathcal{O}(n)$
  - n:n peräkkäisen pino-operaation (joihin saa kuulua myös **push**:in paha tapaus) yhteenlaskettu aikavaativuus on kuitenkin sekin $\mathcal{O}(n)$, eli yhtä komentoa kohti ainoastaan $\mathcal{O}(1)$
  - eli siis vaikka **push** voi joskus viedä paljon aikaa, ovat muut sarjassa suoritetut pino-operaatiot niin halpoja, että ne "maksavat" hankalasta **push**:ista aiheutuneen vaivan

- Tämä johtuu karkeasti ottaen siitä, että kalliiksi tuleva **push**-operaatio ei voi toteutua kuin korkeintaan kerran n:n peräkkäisen operaation aikana
  - kun kallis operaatio taas suoritetaan, taulukkoon tulee runsaasti uutta tilaa
  - ja aikaa vievä **push** voi taas toteutua korkeintaan kerran seuraavan n:n operaation aikana
  - tässä n on taulukon uusi koko eli kaksinkertainen edelliseen nähden
- Tämäntyylistä aikavaativuusanalyysiä kutsutaan tasoitetuksi analyysiksi (engl. amortized analysis), ja on lähinnä syventävien opintojen asioita
- Myös `ArrayList` käyttäytyy näin: silloin tällöin saattaa tulla kallis operaatio, mutta pitkää operaatiosarjaa tarkasteltaessa yhdelle operaatiolle tuleva keskimääräinen osuus on vakioaikainen (jos huomioidaan ainoastaan lisäyksen ja poistot `ArrayListin` lopusta)

Jono

- First-in-first-out, eli toimii kuten jono Unicafessa

- Jono-operaatiot:

  - **enqueue**(Q, k) laittaa tietoalkion k jonon perälle
  - **dequeue**(Q) palauttaa jonon ensimmäisenä olevan tietoalkion ja poistaa sen jonosta
  - **empty**(Q) palauttaa true jos jono tyhjä, muuten false

- Jos jonon koon yläraja tiedetään ennakolta, toteutus onnistuu taulukon avulla

- Toteutus Javalla seuraavaksi

- Määritellään jono luokkana, jolla attribuuttina taulukko `table` johon alkiot talletetaan sekä attribuutit `n` joka kertoo taulukon koon sekä
  - `head`, joka kertoo jonossa ensimmäisenä olevan alkion paikan taulukossa ja
  - `tail`, joka kertoo seuraavaksi jonoon laitettavan alkion paikan

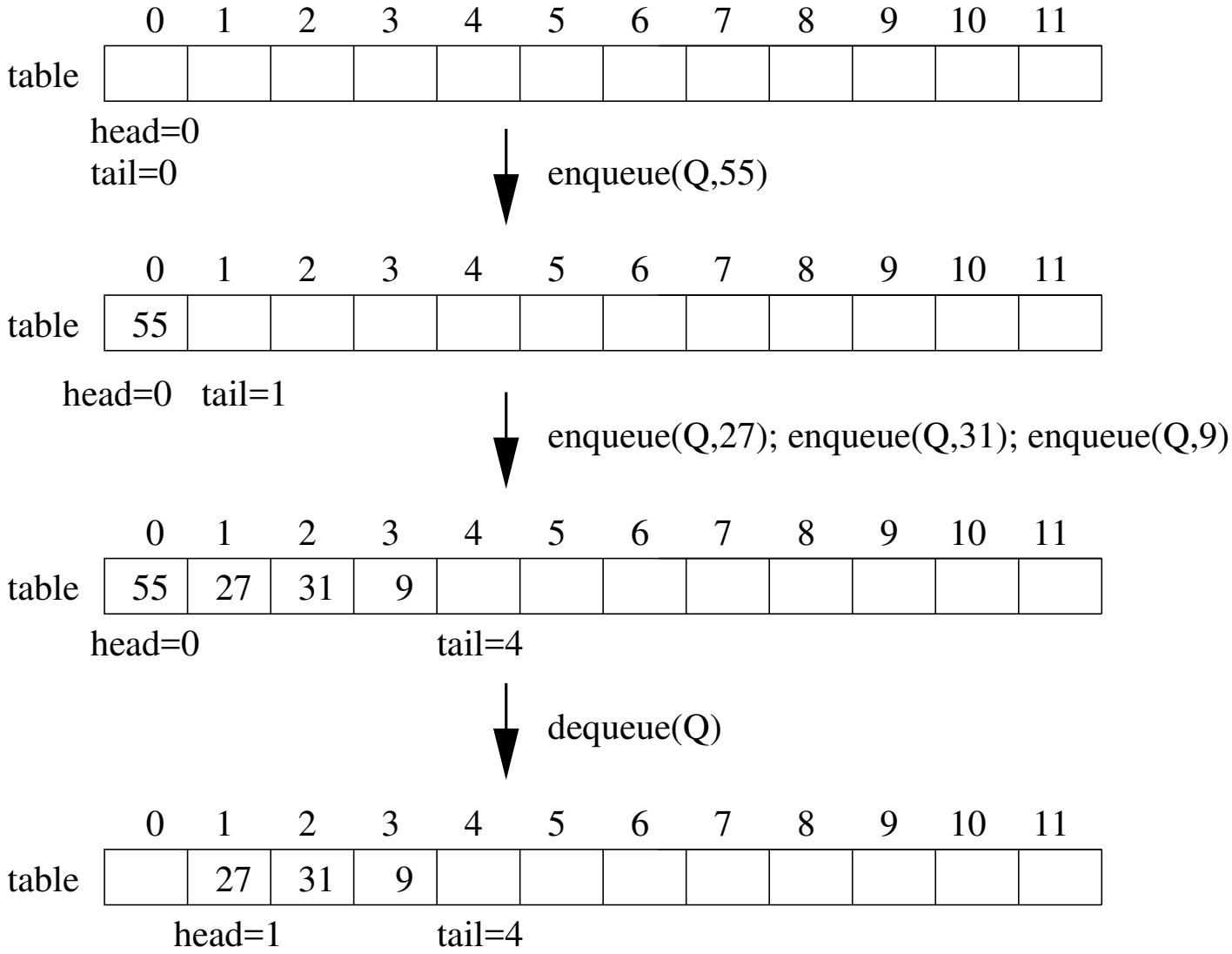
```
class Queue {
   int[] table;
   int head;
   int tail;
   int n;
```

- Konstruktori varaa taulukolle tilan (jonka koko välitetään parametrina) ja alustaa attribuutit sopivasti

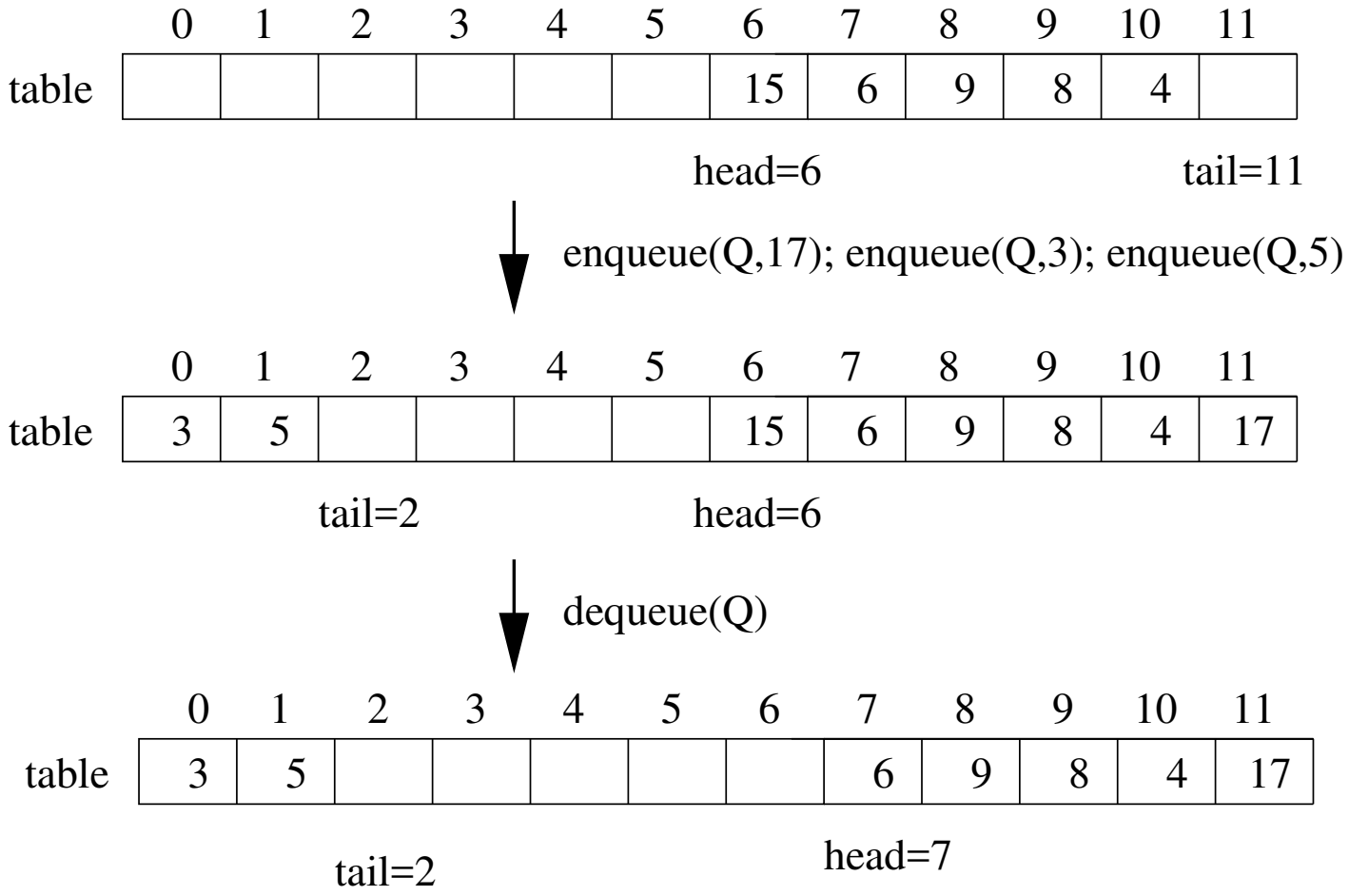
```
   Queue(int size){
      head = 0;   tail = 0;   n = size;
      table = new int[n];
   }
```

- Alussa `head` = `tail` = 0 ja jono on tyhjä
- Seuraavalla sivulla esimerkki jonon toiminnasta

- Jono alkaa täyttyä yllätyksettömällä tavalla taulukon alusta alkaen



- Tilanne muuttuu kiinnostavaksi siinä vaiheessa kun taulukon loppupäässä ei enää ole tilaa, ja jonon häntä pyörähtää taulukon alkupäähän



- Alussa siis `head` $=$ `tail` $= 0$ ja jono on tyhjä
- Jos laitetaan esim. 3 alkiota jonoon ja sen jälkeen otetaan 3 alkiota pois, on `head` $=$ `tail` $= 3$
- Jonon ollessa tyhjä on `head` $=$ `tail`, eli jonon tyhjyyden tarkastus

```
boolean empty(){
   return head == tail;
}
```

- Kannattaa huomioida, että jonoon jonka pituus on n voidaan tallettaa ainoastaan $n-1$ alkiota, sillä muuten tilannetta, jossa jono on tyhjä ei voi erottaa tilanteesta jossa jono on täysi!
- Taulukkoon toteutetulle jonolle on hyvä toteuttaa myös operaatio, jolla testataan onko jono täynnä
- Pienellä miettimisellä huomataan, että jono on täysi, jos `tail`:in seuraava paikka on sama kuin `head`

```
boolean full(){
   int tailnext = tail+1;
   if ( tailnext == n ) tailnext = 0; // jono pyörähtää taas alkuun
   return head == tailnext;
}
```

- Uusi alkio laitetaan jonon perälle eli kohtaan `tail`
- `tail`:ille pitää antaa uusi arvo, eli se joko kasvaa yhdellä tai pyörähtää ympäri, jos seuraava vapaa paikka onkin jonon lopussa

```
void enqueue(int x){
   table[tail] = x;
   tail++;
   if ( tail == n ) tail = 0;
}
```

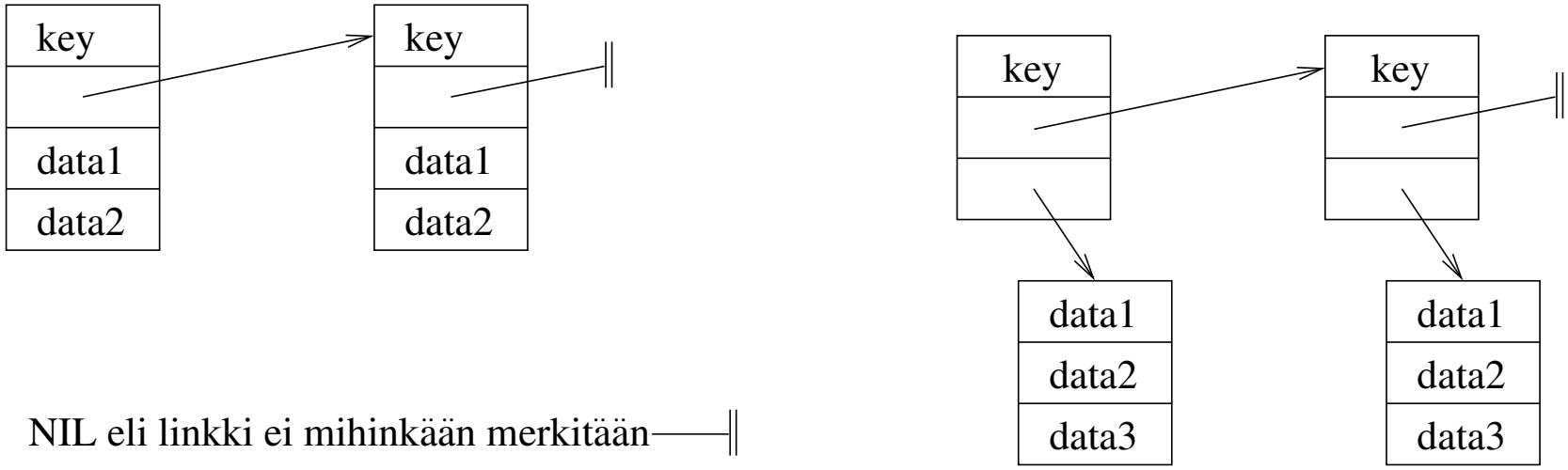
- Alkiot poistetaan jonon edestä eli kohdasta `head`
- Jonon etuosaa merkkaava `head` on päivitettävä uuteen arvoon, eli se joko kasvaa yhdellä tai pyörähtää ympäri, jos jono alkaa taas kerran taulukon alusta

```
int dequeue(){
   int pois = table[head];
   head++;
   if ( head == n ) head = 0;
   return pois;
}
```

- Myös jonon tapauksessa kaikkien operaatioiden aika- ja tilavaativuus on selvästi $\mathcal{O}(1)$

Linkitetty lista

- Linkitetyssä listassa varaamme muistia dynaamisesti tarpeen mukaan, joten alkioiden lukumäärällä ei tarvitse olla ennalta määrättyä ylärajaa
- Lisätään tietoalkiot tallentaviin olioihin attribuutti, joka sisältää viitteen johonkin toiseen tietoalkioon
- Talletettava tieto voi olla myös viitteen takana, eli linkitetty olio voi sisältää pelkästään avaimen, viitteitä linkitetyn rakenteen muihin olioihin sekä viitteen linkitettyä oliota vastaavaan tiedon sisältävään olioon



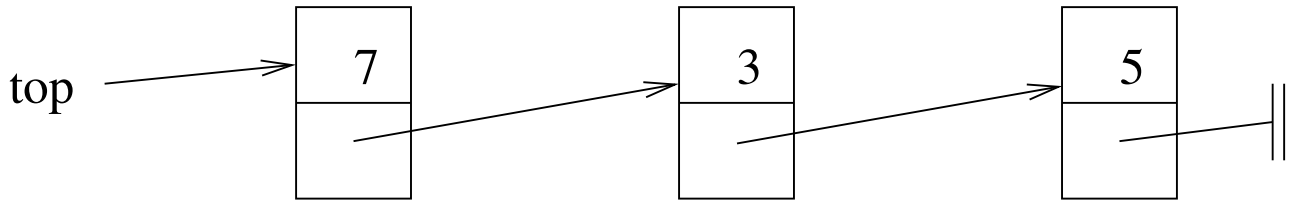
- **Huom.** nykyaikaisilla tietokoneilla taulukoiden käsittely on usein hyvin nopeaa, mutta linkkien käsittely hidasta. Tällaisten rakenteiden vertailussa pelkkään O-aikavaativuuteen tuijottaminen voi johtaa harhaan

- Joka tapauksessa linkitetyn tietorakenteen perusajatus pitää ymmärtää, sitä tarvitaan jatkossa monimutkaisemmissa tietorakenteissa joita ei helposti saa esitetyksi taulukkona

- Esitetään linkitettyyn rakenteeseen perustuva pino ensin pseudokoodina
  - pinossa oleva tieto talletetaan pinosolmu-tyyppiä oleviin olioihin, joilla attribuutit:

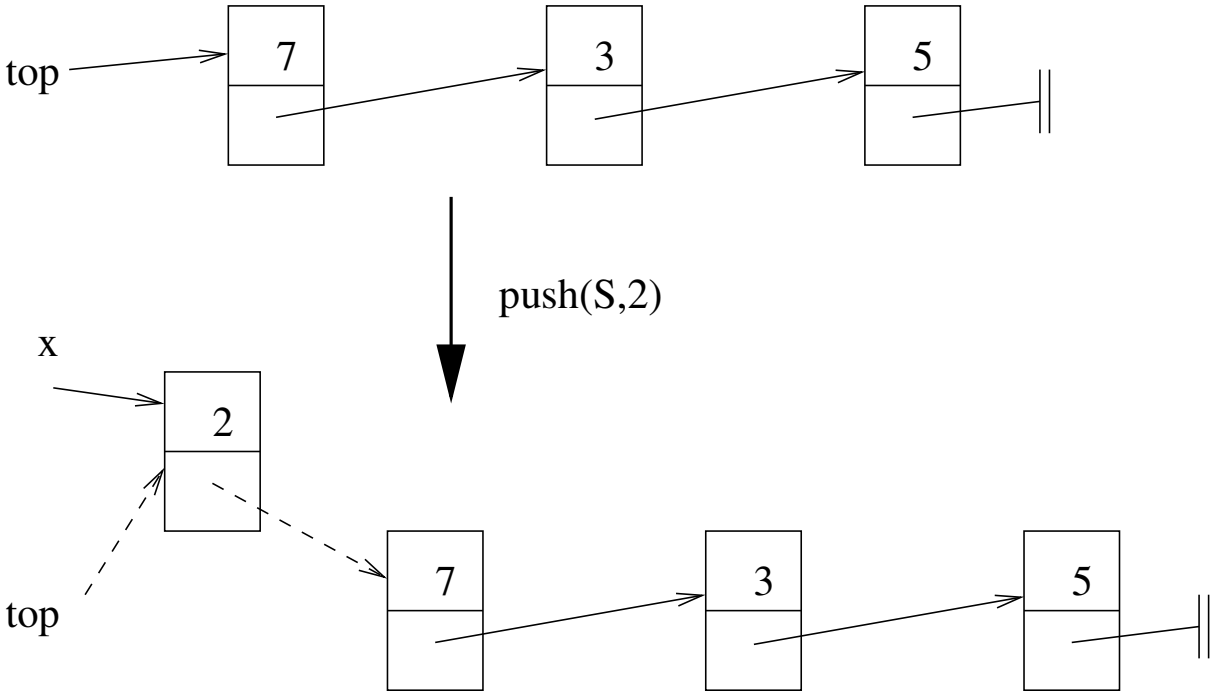
key	pinoon talletettu tietoalkio
$next$	viite toiseen pinosolmu-olioon

  - pinolla (jota merkitään S) on attribuutti top joka on viite siihen pinosolmu-tyyppiä olevaan olioon, joka tallettaa pinon päällä olevan alkion
  - jos $S.top =$ NIL, pino on tyhjä
  - jokaiseen pinosolmun attribuuttiin $next$ talletetaan pinossa seuraavana olevan pinosolmun viite
  - pinon pohjimmaisen pinosolmun $next$-attribuutin arvo on NIL
- Esim: Pino missä huipulla 7, jonka alla 3 ja 5



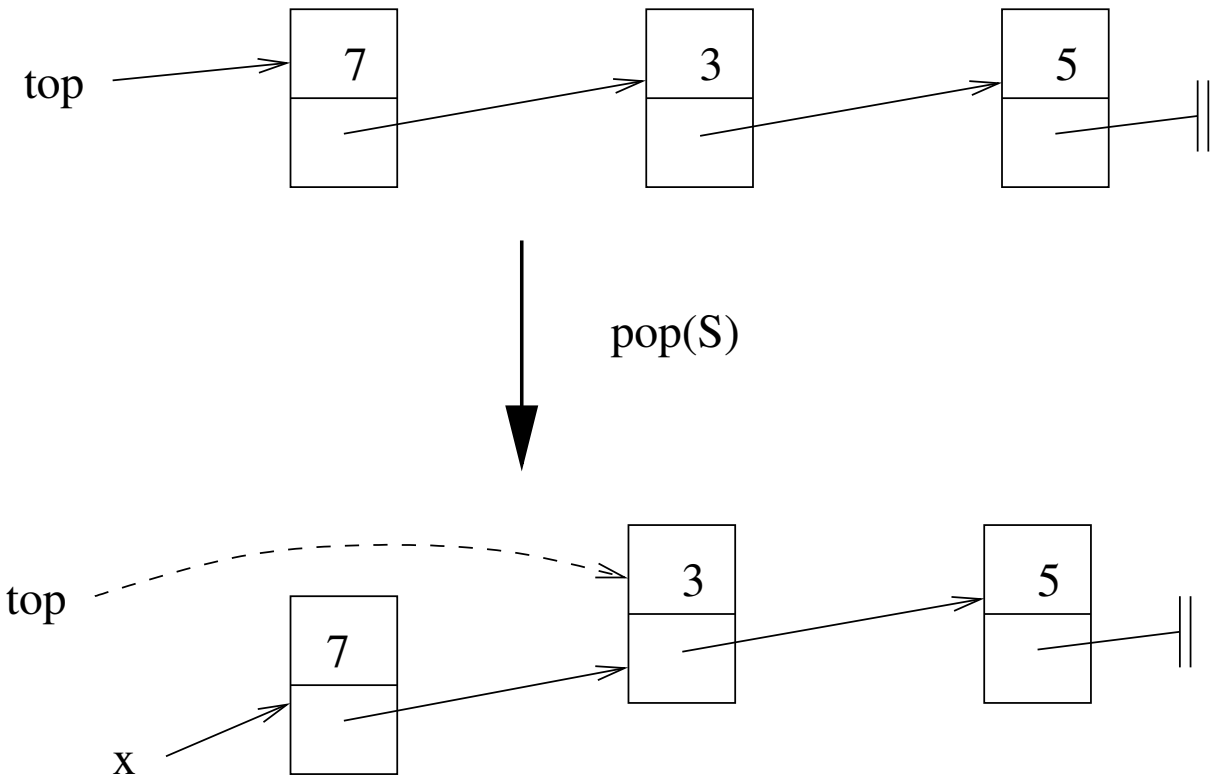
- Seuraavaksi operaatioiden toteutus ja toiminta

```
push(S,uusi)
    x = new pinosolmu        // luodaan pinosolmu
    x.key = uusi             // uusi solmu menee pinon päälle, eli next:iin viite
    x.next = S.top           // aiemmin pinon päällä olleeseen pinosolmuun
    S.top = x                // uusi solmu on tämän jälkeen pinon päällimmäinen
```



- Huom: Kuten sivulla 46 mainittiin, on pseudokoodi epäolio-orientoitunutta, eli operaatiot eivät liity suoraan mihinkään olioon. Kaikki käsiteltävät oliot (kuten nyt pino S) on välitettävä operaatioille parametrina

```
pop(S)
    x = S.top              // pinon uusi huippu on ...
    S.top = x.next         // vanhan huipun x alapuolella ollut pinosolmu
    return x.key           // palautetaan vanhan huipun sisältämä tieto
```



- Huom: **pop**:in jälkeen x:n viittaama pinon vanha huippu jää "orvoksi", esim. Javassa roskankeruumekanismi huolehtisi sen varaaman muistin vapauttamisesta

- Joissain kielissä, kuten C:ssä ja C++:ssa ohjelmoijan on vapautettava itse x:n viittaama muistialue

- Pinon tyhjyyden tarkastaminen on helppoa

  empty(S)
  return S.top==NIL

- Olettaen, että koneesta ei lopu muisti, voi linkitettynä rakenteena toteutettuun pinoon lisätä rajattoman määrän alkiota

- Toisin kuin taulukkoon perustuva, linkitettynä rakenteena toteutettu pino kuluttaa muistia vain sen verran, mitä kullakin hetkellä pinossa olevan tiedon talletukseen vaaditaan (+ olioviitteiden talletukseen kuluva muisti)

- Selvästikin pino-operaatiot ovat jälleen aikavaativuudeltaan $\mathcal{O}(1)$

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

Pinon linkitetty toteutus Javalla

- Katsotaan miten linkitettyyn rakenteeseen perustuva pino toteutetaan Javalla
- Toteutetaan pino luokkana, jolla on seuraavat julkiset metodit

  `public void push(int k)`
  asettaa pinon huipulle luvun k
  `public int pop()`
  palauttaa ja poistaa pinosta päällimmäisen alkion
  `public boolean empty()`
  palauttaa true jos pino tyhjä, muuten false

- Pinosolmut kannattaa toteuttaa omana luokkana

```
class PinoSolmu {
   int key;
   PinoSolmu next;
   PinoSolmu(int k, PinoSolmu seur) {
      key = k; next = seur;
   }
}
```

- PinoSolmu attribuutteina ovat solmuun talletettavaa tietoa varten oleva `int key` sekä `PinoSolmu next`, eli olioviite toiseen PinoSolmu-olioon

- Luokan Pino ainoa attribuutti on `PinoSolmu top`, eli viite pinon päällä olevaan PinoSolmu-olioon

- Pinon konstruktori asettaa `top = null`, sillä pino on aluksi tyhjä

```
public class Pino {
   private PinoSolmu top;    // viite pinon päällä olevaan alkioon

   public Pino() { top = null; }    // alussa pinossa ei ole mitään
```

- Operaatioiden toteutus ei juuri poikkea aiemmin esitetystä pseudokoodista, ensin **push**:

```
   public void push(int k) {
      // uuden solmun x kentille asetetaan arvot konstruktorissa
      // uuden huipun alapuolella siis pinon vanha huippu
      PinoSolmu x = new PinoSolmu(k,top);
      top = x;              // asetetaan uusi solmu x huipuksi
   }
```

- **pop** ja **empty**:

```
public int pop() {
   PinoSolmu x = top;
   top = x.next;          // uusi huippu on vanhan huipun x alla oleva solmu
   return x.key;
}

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

- **pop**-operaation yhteydessä entinen pinon huippusolmu jää ilman viitettä, ja Javan roskankeräysmekanismi tulee aikanaan poistamaan vanhalle huipulle varatun muistialueen

- Pinon käyttö sovelluksissa:

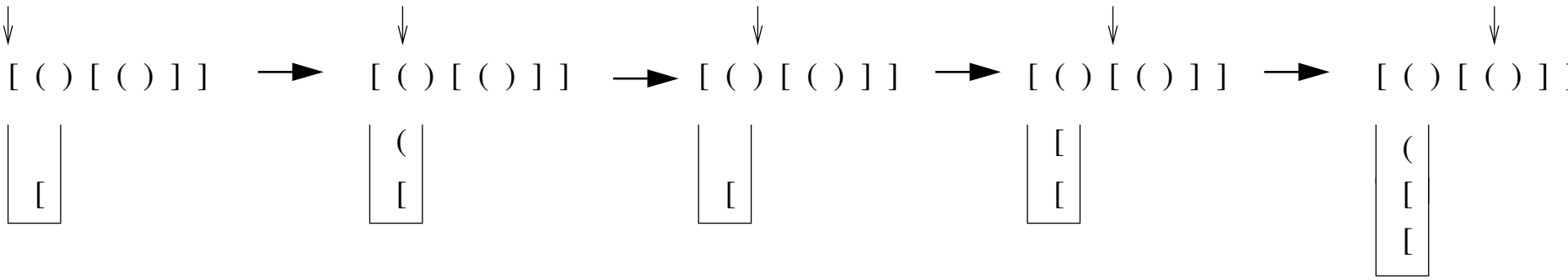
```
Pino pino = new Pino() ;
pino.push(10);
pino.push(15);
while ( !pino.empty() )
    System.out.println(" pinosta otettiin: " + pino.pop());
```

Esimerkki pinoa käyttävästä algoritmista

- Ohjelmoinnissa tulee helposti virheitä sulkumerkkien { } ( ) [ja] kirjoittamisessa:
  - jollekin alkusululle ei löydy loppusulkua tai päinvastoin: ()) tai ((
  - sulut voivat "mennä ristiin": ({)}

- Sulkujen tasapaino syötteenä olevasta merkkijonosta voidaan testata esim. seuraavasti:
  - luetaan syötemerkkijonon merkit yksi kerrallaan, alusta alkaen
  - viedään jokainen alkusulkumerkki eli (, { tai [pinoon
  - kun luetaan jokin loppusulkumerkki eli), } tai] sitä vastaavan alkusulkumerkin pitää löytyä pinon päältä. alkusulku poistetaan pinosta
  - kun tiedosto on luettu, pinon pitää olla tyhjä

- Algoritmin toimintaa syötteellä [( ) [( )]]



- Pinon päällä on aina viimeksi nähty alkusulkumerkki, jonka vastinetta ei ole vielä tullut vastaan

- Eli kun syötteessä tulee vuoroon loppusulkumerkki, on sen oltava pinon päällimmäisen vastine, muuten sulut ovat ristissä

  loppusulkumerkin löytyessä pinon päälimmäinen sulku otetaan pois sillä se on käsitelty, esim. kuvassa kolmas askel

- Jos taas seuraava sulkumerkki onkin alkusulku, menee se pinon päälimmäiseksi ja ruvetaan odottamaan sitä vastaavaa loppusulkua, esim. kuvassa toinen askel

- Syvemmällä pinossa olevat sulut ovat ulompaa sulutustasoa, ja ne käsitellään pinon ylemmän osan käsittelyn jälkeen

- Seuraavassa Javalla toteutettu pinoa käyttävä algoritmi sulkujen tasapainon tarkastamiseksi. Ollaan kiinnostuneita vain suluista ( ) ja { }, algoritmi on helppo laajentaa myös sulut [ ] tarkistavaksi

```
   public boolean tasapainoinen(String mj) {
0     Pino pino = new Pino();                  // pinon talletettavien tyyppi char
1     for ( int i=0; i<mj.length(); i++){
2        char c1 = mj.charAt(i);
3        if ( c1 == '(' || c1 == '{' )
4           pino.push(c1);
5        else if ( c1 == ')' || c1 == '}' ) {
6           if ( pino.empty() ) // liian vähän ( tai {
7              return false;
8           char c2 = pino.pop();
9           if ( c1 == ')' && c2 != '(' )      // väärä sulku
10             return false;
11          else if ( c1 == '}' && c2 != '{' ) // väärä sulku
12             return false;
13       }
14    }
15    if ( !pino.empty() ) // liikaa ( tai {
16       return false;
17    else
18      return true;
   }
```

- for-lauseessa käydään läpi syötteenä oleva merkkijono merkki merkiltä
- Muut kuin sulkumerkit jätetään huomioimatta
- Jos vuorossa oleva merkki on alkusulku eli (tai [, laitetaan se pinoon
- Jos vuorossa oleva merkki on loppusulku eli) tai] tarkastetaan, että sitä vastaava alkusulku löytyy:
  - rivillä 6 tarkastetaan, että edes joku vastinsulku on olemassa
  - rivillä 9 tarkastetaan, että sulun) tapauksessa pinossa (
  - rivillä 11 tarkastetaan, että sulun] tapauksessa pinossa [
- Lopuksi rivillä 15 vielä tarkistetaan, että pino on lopulta tyhjä, eli että jokaiselle alkusululle on löytynyt vastine

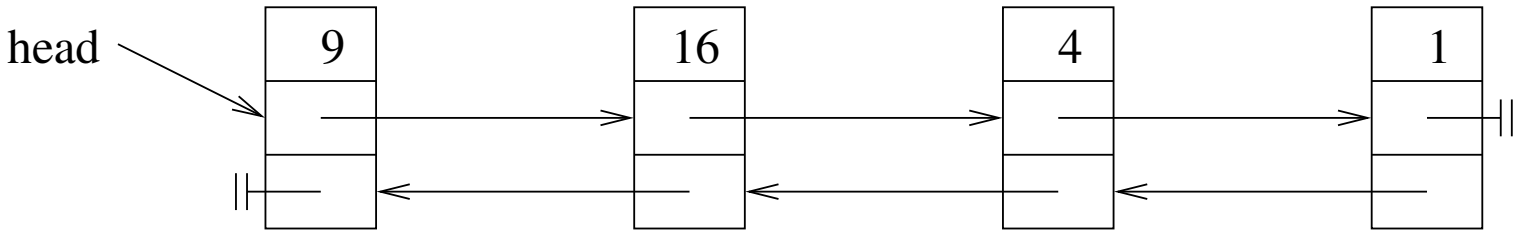
- Mikä on algoritmin vaativuus?
- for-lauseen runko suoritetaan kertaalleen jokaiselle syötemerkkijonon merkille
- Jokaisen merkin kohdalla tehdään muutama vertailuoperaatio sekä maksimissaan yksi pino-operaatio
- Pino-operaatioiden vaativuus on $\mathcal{O}(1)$, eli for-lauseen runko-osa vie vakioajan
- Algoritmin aikavaativuus on siis $\mathcal{O}(n)$ syötteen pituudella n
- Jos syöte koostuu pelkistä suluista ja on muotoa $(((\ldots()\ldots)))$, suoritetaan aluksi **push** operaatioita $n/2$ kappaletta ennen ensimmäistä **pop**:ia. Eli pinossa on pahimmillaan $n/2$ sulkua. Jos syöte on pelkästään $(((\ldots($, suoritetaan n **push** operaatiota
- Pahimmassa tapauksessa algoritmin tilavaativuus siis on $\mathcal{O}(n)$ syötteen pituuteen n nähden
- Algoritmilla on yleisempääkin käyttöä: XML:ssä jokainen elementti avataan ja suljetaan, tyyliin tekstikappale , elementtien täytyy olla samalla tavalla tasapainossa kuin tasapainoisesti sulutettujen merkkijonojen eli pienellä lisävirittelyllä algoritmi saadaan tarkastamaan XML-dokumenttien elementtien tasapaino

Kahteen suuntaan linkitetty lista

- Linkitetyn listan avulla on helppo toteuttaa abstrakti tietotyyppi joukko
- Linkitetyistä listoista on monia eri variaatioita joista nyt tarkastelemme kahteen suuntaan linkitettyä listaa
- Listalla oleva tieto talletetaan listasolmu-tyyppiä oleviin olioihin, joilla on attribuutit:

key	talletetun tietoalkion avain
...	mahdolliset muut tietoa tallettavat attribuutit
$next$	viite seuraavana olevaan listasolmu-olioon
$prev$	viite edellisenä olevaan listasolmu-olioon

- Listalla L on attribuutti $L.head$ joka on viite listasolmu-olioon, joka tallettaa listan alussa olevan tietoalkion. Jos $L.head =$ NIL on lista tyhjä
- Esim. lista, jossa on luvut 9, 16, 4 ja 1



- Alkiota listalta etsivässä **search**-operaatiossa verrataan listan alkioita alusta lähtien etsittävään

- Operaatio palauttaa viitteen etsityn alkion tallettamaan listasolmuun

- Jos etsittyä ei löydy, palautetaan viite NIL

```
search(L,k)
    p = L.head
    while p ≠ NIL and k ≠ p.key      // jos ei olla lopussa tai löydetty etsittyä
        p = p.next                   // jatketaan eteenpäin
    return p
```

- apumuuttuja p viittaa listasolmuun, jossa etsintä on menossa

- aluksi p viittaa ensimmäiseen listasolmuun

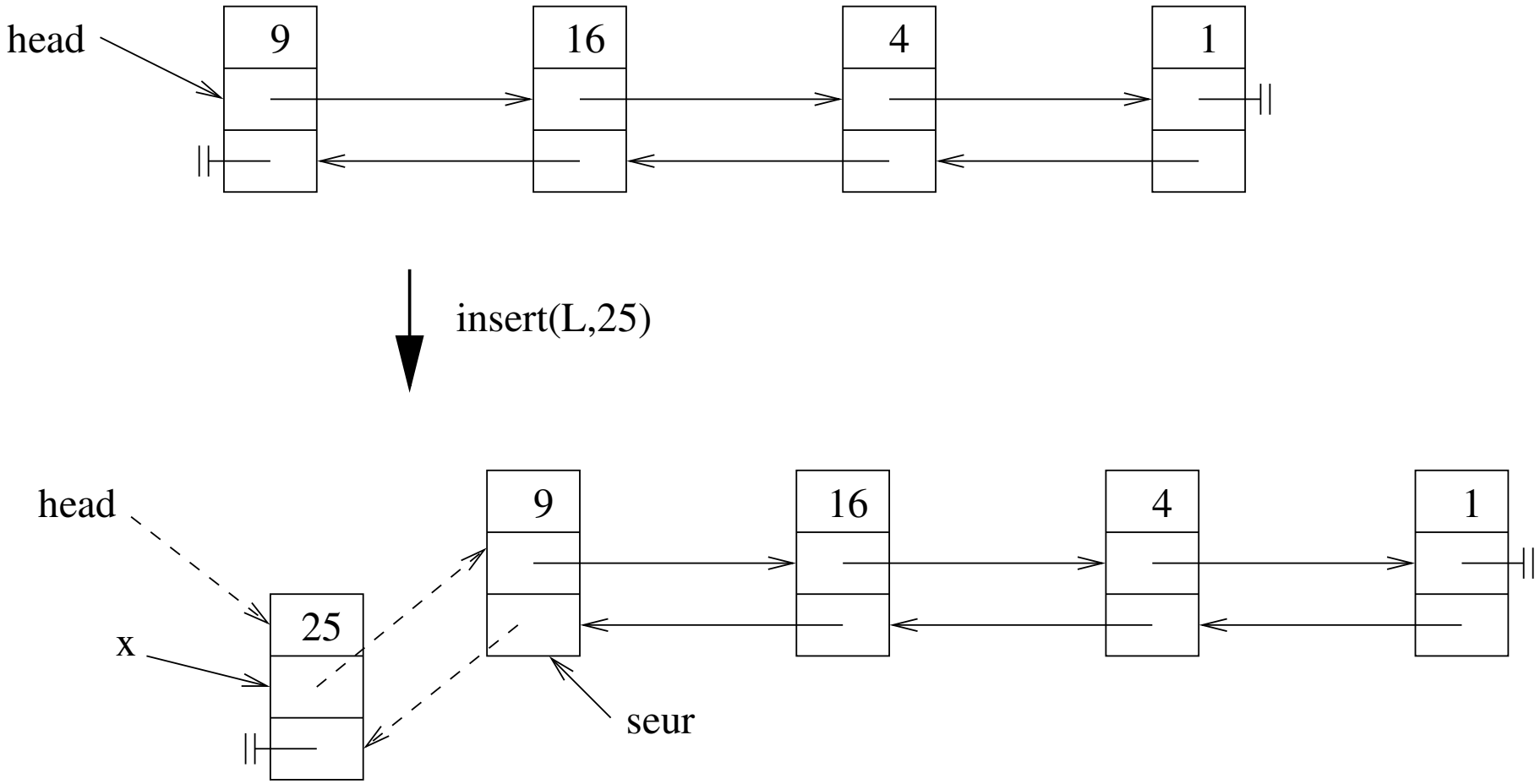
- Listalla edetessä p laitetaan viittaamaan seuraavaan listan solmuun asettamalla sen arvoksi nykyisen solmun p seuraava eli $p.next$

- Jos etsittävää ei löydy tai se on listan viimeisenä, käydään kaikki listan alkiot läpi. Pahimmassa tapauksessa operaation aikavaativuus on siis $\mathcal{O}(n)$, missä n on listalla olevien alkioiden määrä

- Uusi alkio lisätään aina listan alkuun
- Uuden alkion seuraava on siis vanha listan ensimmäinen alkio
- Lisäyksen tapauksessa on oltava huolellinen, että kaikki viitteet laitetaan kuntoon
- Myös erikoistapaus, jossa lisäys tapahtuu tyhjään listaan (eli kun $L.head ==$ NIL operaatiota suoritettaessa) on huomioitava

```
insert(L,k)
    x = new listasolmu
    x.key = k
    x.next = L.head          // uuden seuraava on vanha ensimmäinen
    x.prev = NIL
    if L.head ≠ NIL
        seur = x.next
        seur.prev = x        // uusi on seuraajansa edeltäjä
    L.head = x               // lisätty solmu on listan ensimmäinen
```

- Esimerkki seuraavalla sivulla selventää operaation toimintaperiaatetta



- Riippumatta jonossa olevien alkioiden määrästä, operaation suorittamien komentojen määrä on aina sama, eli operaation aikavaativuus $\mathcal{O}(1)$

- Listalta alkioita poistava **delete**-operaatio saa parametrinaan viitteen poistettavaan solmuun x

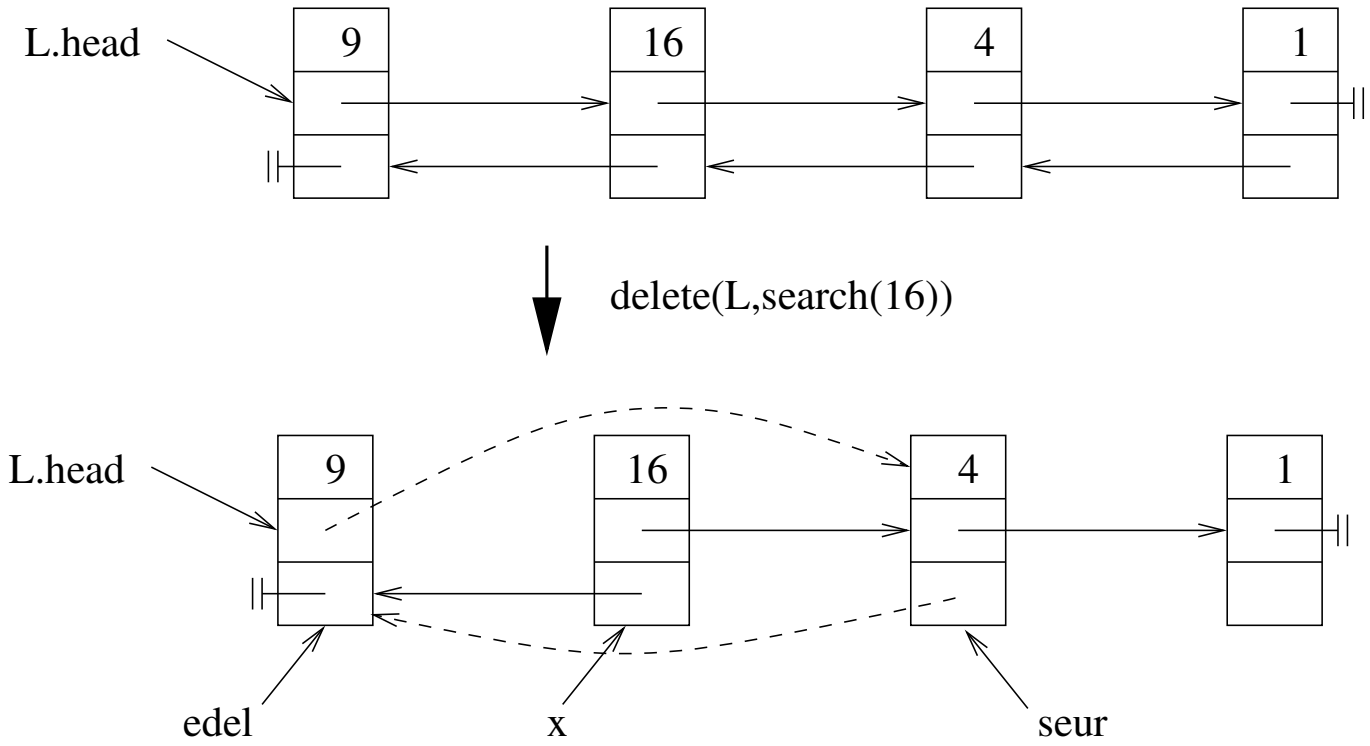
- Operaatiossa manipuloidaan viitteitä siten, että x:n yli hypätään, eli x:n edellinen viittaakin suoraan x:n seuraavaan

- Jälleen on huomioitava erikoistapaukset, eli onko x:llä seuraavaa tai edellistä solmua

```
delete(L,x)
    edel = x.prev
    seur = x.next
    if edel ≠ NIL
        edel.next = seur
    else L.head = seur     // edellistä ei ollut, eli poistettu oli listan ensimmäinen
    if seur ≠ NIL
        seur.prev = edel
```

- Seuraavalla sivulla on esimerkki, jossa poistettavalla on sekä edellinen että seuraava solmu

- On hyödyllistä simuloida myös erikoistapaukset: ei seuraavaa, ei edellistä ja poistettava listan ainoa solmu



- Listan pituudesta riippumatta operaatio suorittaa vakiomäärän komentoja, eli aikavaativuus $\mathcal{O}(1)$

- Huom: **delete**-operaatio saa parametrinaan viitteen poistettavaan solmuun eli jos halutaan poistaa listalta esim. luku 4, on viite poistettavaan solmuun selvitettävä **search**-operaatiolla: **delete**(L,**search**(4))

- Koska **search**-operaation aikavaativuus on $\mathcal{O}(n)$, on myös **delete**(L,**search**(x)):n aikavaativuus $\mathcal{O}(n)$

- Entä loput sivulla 162 mainituista operaatioista: **min**, **max**, **succ** ja **pred**?
  - **min** siis etsii listalta solmun, jossa on pienin avain
  - **succ** taas etsii annettua solmua x seuraavaksi suurimman avaimen omaavan solmun
- **min** voidaan toteuttaa seuraavasti:

```
min(L)
    pienin = L.head                 // pienin tähänasti tunnettu
    p = L.head
    while p ≠ NIL
        if p.key < pienin.key       // löytyikö pienempi kun tähänastinen pienin
            pienin = p
        p = p.next                  // edetään listalla
    return pienin
```

- Operaatiossa käydään läpi koko lista solmu solmulta, muuttuja p vittaa vuorossa olevaan listasolmuun
- Muuttuja pienin muistaa, mikä solmu sisältää siihen mennessä pienimmän vastaantulleen avaimen
- Operaatio käy läpi listan kaikki alkiot ja jokaisen alkion kohdalla tehdään vakiomäärä työtä, eli aikavaativuus selvästi $\mathcal{O}(n)$

- Operaatio **succ**, joka etsii x:stä seuraavaksi suurimman avaimen sisältävän solmun on hiukan hankalampi:

```
succ(L,x)
1   seur = NIL
2   p = L.head
3   while p ≠ NIL
4       if p.key > x.key
5           if seur == NIL or p.key < seur.key
6               seur = p
7       p = p.next
8   return seur
```

- Operaatiossa käydään läpi koko lista solmu solmulta, muuttuja p viittaa läpikäynnissä vuorossa olevaan listasolmuun

- $seur$ on viite solmuun, joka on siihen astisen tietämyksen perusteella x:ää seuraavaksi suurempi

- Aina kun tulee vastaan x:ää suuremman avaimen omaava listasolmu (rivi 4) tarkastetaan (rivi 5), onko tämä parempi kandidaatti etsityksi alkioksi kuin siihen asti parhaaksi luultu $seur$

- Jos syötteenä oleva x on listan suurimman avaimen omaava listasolmu, palauttaa operaatio NIL

- Operaatio **succ** käy koko listan läpi tehden vakiomäärän työtä jokaista solmua kohti, aikavaativuus siis $\mathcal{O}(n)$

- Operaatiot **max** ja **pred** toteutetaan samaan tyyliin kuin **min** ja **succ**

- Operaatiota **succ** tarvitaan kun halutaan käydä listalla olevat tiedot läpi avaimen mukaisessa järjestyksessä. esim. kaikkien tulostus järjestyksessä:

```
x = min(L)
while x ≠ NIL
    print ( x.key )
    x = succ(L,x)
```

- Eli ensin haetaan pienimmän avaimen sisältävä solmu x. Sen jälkeen aina seuraava kunnes kaikki on käyty läpi

- Jokaisen listasolmun osalta sen seuraajan etsiminen operaatiolla **succ** vie aikaa $\mathcal{O}(n)$, eli solmujen läpikäynti avaimeen perustuvassa järjestyksessä vie aikaa peräti $\mathcal{O}(n^2)$ missä n listalla olevien solmujen määrä

- Jos operaatiot **min**, **max**, **succ** ja **pred** ovat tarpeen, ei joukon toteuttamisessa esitetyn kaltaisena linkitettynä listana ole järkeä jos listalle talletettavien alkioiden määrä on suuri

- Esitetyllä tavalla kaksisuuntaisena linkitettynä (järjestämättömänä) listana toteutetun abstraktin tietotyypin kokoelma operaatioiden aikavaativuudet siis ovat

search(S, k)	$\mathcal{O}(n)$
insert(S, x)	$\mathcal{O}(1)$
delete(S, x)	$\mathcal{O}(1)$
min(S)	$\mathcal{O}(n)$
max(S)	$\mathcal{O}(n)$
succ(S, x)	$\mathcal{O}(n)$
pred(S, x)	$\mathcal{O}(n)$

- Jos lista on vain yhteen suuntaan linkitetty, ts. *prev*-osoittimet on jätetty pois, aikavaativuudet ovat muuten samat, mutta **delete**-operaation toteutus muuttuu mutkikkaaksi. Etuna on muistitilan säästö.

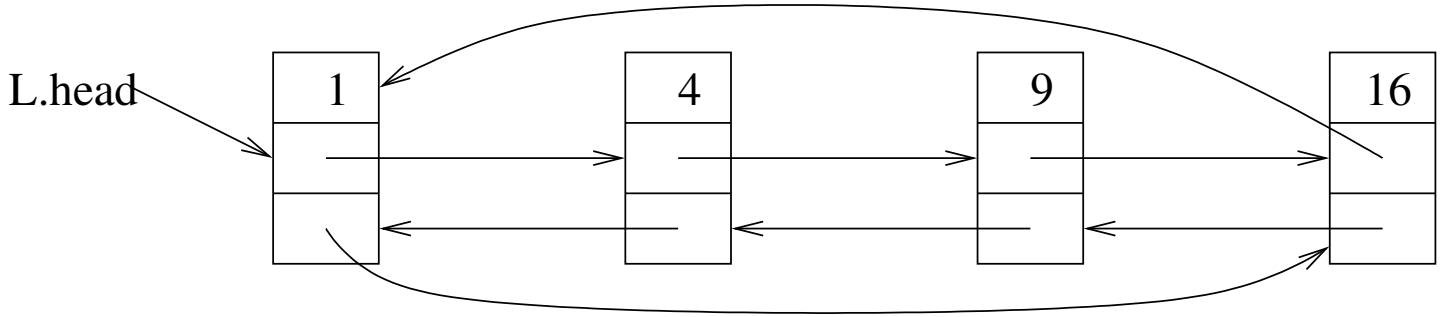
- Koska tietorakenne sallii saman alkiot tallettamisen useaan kertaan, ei kyseessä matemaattisessa mielessä ole joukko vaan pelkkä kokoelma alkoita

Järjestetty lista ja rengaslista

- Jos talletamme luvut listalle suuruusjärjestyksessä, **insert**-operaation vaativuus huononee luokkaan $\mathcal{O}(n)$ mutta operaatiot **min** ja **succ** saadaan vakioaikaisiksi

- **insert** siis ei lisää uusia alkioita aina listan ensimmäiseksi alkioksi, vaan etsii niille oikean paikan siten, että lisäyksen jälkeenkin listan alkiot ovat suuruusjärjestyksessä

- Yksi tapa myös operaatioiden **max** ja **pred** saamiseksi vakioaikaiseksi on tallettaa alkiot suuruusjärjestyksessä kahteen suuntaan linkitetyksi rengaslistaksi



- Jonon ensimmäisen alkion $prev$ viittaa jonon viimeiseen alkioon, jonka $next$ viittaa jonon ensimmäiseen alkioon

- Operaatioiden **min**, **max**, **succ** ja **pred** toteuttaminen järjestetylle rengaslistalle on helppoa:

```
min(L)
    return L.head

max(L)
    if L.head == NIL return NIL
    else return L.head.prev

succ(L,x)
    if x.next == L.head return NIL
    else return x.next

pred(L,x)
    if x.prev == L.head.prev return NIL
    else return x.prev
```

- Edelliset neljä operaatiota ovat selvästi vakioaikaisia sillä olipa lista miten pitkä tahansa, aina suoritetaan sama määrä käskyjä

- Muutama sivu sitten toteutettu listan alkioiden tulostus suuruusjärjestyksessä onnistuu järjestetylle ajassa $\mathcal{O}(n)$

- Alkioiden poistaminen järjestetyltä rengaslistalta:

```
delete(L,x)
    edel = x.prev
    seur = x.next
    if x == seur                    // poistettava on listan ainoa alkio
        L.head = NIL
    else
        edel.next = seur
        seur.prev = edel
    if x == L.head                  // poistettava on listan ensimmäisenä
        L.head = seur
```

- Operaatio ei poikkea paljoa aiemmin esitetystä normaalin listan **delete**:stä
- Erikoistapaukset on jälleen huomioitava
- Jos rengaslistalla on vain yksi listasolmu, on solmu sekä itsensä seuraaja ja edeltäjä, tämä tilanne testataan kolmannella rivillä
- Operaatio on selvästi vakioikainen, sillä listan pituus ei vaikuta suoritettavien käskyjen määrään

- Alkioiden etsintä järjestetyltä rengaslistalta:

```
search(L,k)
1   if L.head == NIL return NIL
2   p = L.head
3   while p.next ≠ L.head and p.key < k
4       p = p.next
5   if p.key== k return p else NIL
```

- Operaatio on hiukan erilainen kuin normaalin listan **search**
  - läpikäynti voidaan lopettaa jos huomataan, että listalla tulee vastaan etsittyä avainta suurempi alkio
  - rengaslista on käyty läpi siinä vaiheessa kun ollaan palaamassa jälleen listan ensimmäiseen solmuun $L.head$
- Rivin 3 ehto lopettaakin läpikäynnin, jos ollaan viimeisen alkion kohdalla tai jos etsittäviä avaimia pienempiä ei enää listalla ole
- Solmu jonka kohdalle etsintä pysähtyy, sisältää etsityn avaimen jos se ylipäätään listalla on. Rivillä 5 vielä tarkistetaan asia
- Vaikka voimmekin lopettaa etsinnän, kun listalla tulee vastaan alkio, joka on suurempi kuin etsittävä avain, etsinnän pahimman tapauksen aikavaativuus on kuitenkin $O(n)$

- Järjestetyn rengaslistan operaatioista teknisesti hankalin on **insert**
- Koska alkio on vietävä oikealle paikalle, kuluu aikaa pahimmassa tapauksessa $O(n)$, sillä alkio voidaan joutua lisäämään listan loppuun
- Operaatio on periaatteessa selkeä, mutta sen toteutus on ikävä monien huomioitavien erikoistapauksien takia
- Kaksisuuntaisesti linkitettynä rengaslistana toteutetun abstraktin tietotyypin kokoelma operaatioiden aikavaativuudet siis ovat

search(S, k)	$\mathcal{O}(n)$
insert(S, x)	$\mathcal{O}(n)$
delete(S, x)	$\mathcal{O}(1)$
min(S)	$\mathcal{O}(1)$
max(S)	$\mathcal{O}(1)$
succ(S, x)	$\mathcal{O}(1)$
pred(S, x)	$\mathcal{O}(1)$

- Eli uhraamalla operaatio **insert**, päästään operaatiossa **min**, **max**, **succ** ja **pred** vakioaikaisuuteen

Tunnussolmullinen lista

- Joitakin operaatiota mutkistaa hieman erikoistapauksen onko poistettava alkio listan alussa huomioiminen

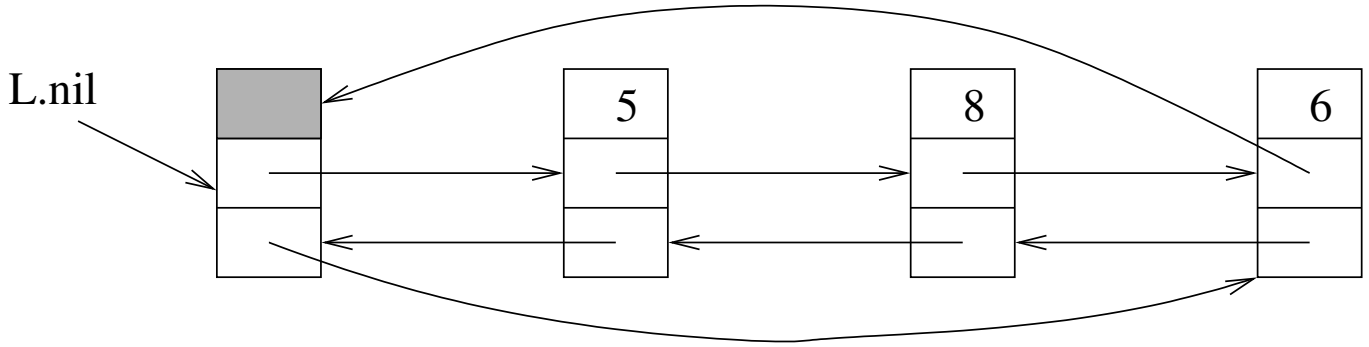
- Yksi variaatio linkitetyistä listoista on tunnussolmullinen rengaslista
  - nyt listan alussa tunnussolmu, eli solmu missä ei säilytetä tietoa
  - listan L attribuutti $L.nil$ viittaa listan tunnussolmuun
  - listan ensimmäinen alkio löytyy viitteen $L.nil.next$ päästä
  - listan viimeinen alkio löytyy viitteen $L.nil.prev$ päästä

- Huom: Jotta seuraavassa esitettävät operaatiot toimisivat, tyhjällä listalla täytyy olla: $L.\,nil\,.\,next = L.\,nil\,.\,prev = L.\,nil$

  eli tyhjässä listassa tunnussolmun seuraavaksi ja edeltäväksi solmuksi on asetettu tunnissolmu itse

- Käsitellään seuraavassa tunnussolmullisen listan variaatiota joka ei edellytä alkioille suuruusjärjestystä

- Esim: Tunnussolmullinen rengaslista missä luvut 5, 8 ja 6



- Alkion poisto tunnussolmullisesta rengaslistalta hoituu erittäin helposti

  delete(L,x)
  seur = x.next
  edel = x.prev
  seur.prev = edel
  edel.next = seur

- Varmista simuloimalla, että operaatio toimii myös erikoistapauksissa (listalla vain yksi alkio, poistettava alussa, poistettava lopussa)

- Alkion lisääminen listan alkuun onnistuu vakioajassa

```
insert(L,k)
    x = new listasolmu
    x.key = k
    seur = L.nil.next       // seuraava on listan vanha ensimmäinen
    x.next = seur
    x.prev = L.nil
    seur.prev = x
    L.nil.next = x          // lisätty on listan ensimmäinen alkio
```

- Sekä poisto- että lisäysoperaatio ovat suoraviivaisia sillä erityistapauksia ei koodissa varvitse huomioida

- Vakuuta itsellesi simuloimalla, että **insert** voimii myös erikoistapauksissa

- Alkion etsintä tunnussolmullisesta rengaslistasta

```
search(L,k)
    p = L.nil.next
    while p ≠ L.nil and k ≠ p.key
        p = p.next
    if p == L.nil
        return NIL
    else return p
```

- Linkitetystä listasta on muitakin variaatioita: tunnussolmuton rengaslista, yhteen suuntaan linkitetty lista sekä tunnussolmulla että ilman, tavallisena tai rengaslistana . . .

- Kuten jo todettiin, määrittelemämme linkitetty lista ei tarkalleen ottaen toteuta joukkoa niin kuin se käsitetään matemaattisesti sillä joukossahan tietty alkio voi esiintyä vaan kertaalleen mutta mikään ei estä saman alkion laittamista useaan kertaan listalle

- Linkitetty lista on yksinkertainen mutta useimmissa tilanteissa suhteellisen tehoton tietorakenne, eli onko listalle ylipäätään käyttöä?

- Jos käsiteltävä aineisto on pieni, riittää linkitetty lista varsin hyvin

- Linkitetty lista on käyttökelpoinen komponentti tietyissä muissa tietorakenteissa kuten hajautusrakenteissa

- Jos listan hitaita operaatioita (esim. **search**) tarvitaan äärimmäisen harvoin ja tilanteissa, joissa suoritusaika ei ole kovin kriittinen, voi lista olla järkevä vaihtoehto sen yleisen keveyden takia

- Yhteenvetona voi todeta, että meillä on 16 eri variaatiota linkitetystä listasta, riippuen seuraavista valinnoista:
  - yhteen suuntaan linkitetty TAI kahteen suuntaan linkitetty
  - tunnussolmullinen TAI tunnussolmuton
  - tavallinen lista TAI rengaslista
  - järjestetty TAI järjestämätön

5. Hajautus

- Tarkastellaan edelleen sivulla 165 esitellyn joukkotietotyypin toteuttamista
- Useissa sovelluksissa riittää että operaatiot **insert**, **delete** ja **search** toimivat nopeasti
  - esim. sivun 30 puhelinluetteloesimerkissä ei puhelinnumeron mukaan järjestetyssä indeksirakenteessa luultavasti ole kovin usein käyttöä operaatioille **min**, **max**, **succ** ja **pred**
  - sen sijaan nimen mukaan järjestetyssä indeksissä operaatioille **min** ja **succ** voi olla käyttöä, jos on tarvetta esim. listata kaikki nimet aakkosjärjestyksessä
- Tapauksissa, joissa **insert**-, **delete**- ja **search**-operaatiot riittävät, hajautusrakenne on varteenotettava tapa joukon toteutukselle

- Hajautusrakenteessa **insert** ja **delete** toimivat vakioajassa ja **search** toimii keskimäärin vakioajassa
- Pahimmassa tapauksessa hajautusrakenteen **search** voi viedä $\mathcal{O}(n)$
  - opimme hieman myöhemmin, miten tasapainoisella hakupuulla kaikki operaatiot voi toteuttaa pahimman tapauksen aikavaatimuksella $\mathcal{O}(\log n)$
- Hajautusrakenteen tehokkuuden kannalta tärkeää on hajautusfunktion valinta
- Hyvän hajautusfunktion löytäminen voi vaatia hieman kokeilua, mutta muuten menetelmä on helppo toteuttaa
- Avainten järjestykseen liittyviä operaatioita **succ**, **pred**, **min** ja **max** ei hajautuksessa erityisemmin tueta, ne voidaan toteuttaa ajassa $O(n)$
  - tasapainoisella hakupuulla nekin vievät vain $O(\log n)$

- Merkitään kaikkien mahdollisten avainten joukkoa U:lla
  - esim. opiskelijarekisterissä tämä olisi kaikki mahdolliset opiskelijanumerot tai puhelinluettelossa kaikki mahdolliset puhelinnumerot tai kaikki mahdolliset nimet

- Otetaan käyttöön m-paikkainen hajautustaulukko (engl. hash table) T, joka on indeksoitu alkaen nollasta

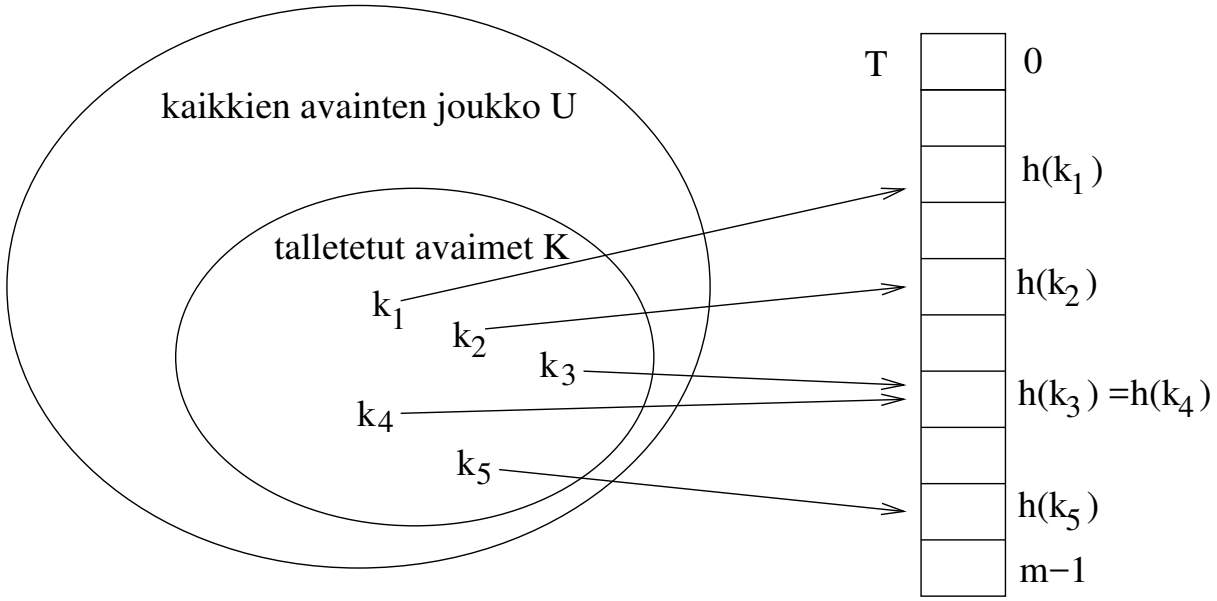
- Koska yleensä U:n koko on erittäin suuri (tai ääretön), on $m < |U|$

- Määritellään hajautusfunktio (engl. hash function) $h : U \to \{0, \ldots, m-1\}$, joka siis kuvaa jokaisen avaimen jollekin kokonaisluvulle väliltä $0, \ldots, m-1$

- Toisin sanoen, hajautusfunktio kuvaa jokaisen avaimen jollekin taulukon T indeksille

- Sanotaan että $h(k_i)$ on avaimen k_i osoite tai kotiosoite

- Ideana on nyt että avain k_i talletetaan taulukon T paikkaan $h(k_i)$, eli $T[h(k_i)] = k_i$



- Kuten kuva yllä kertoo, voi käydä niin että kaksi avainta saavat saman osoitteen eli joillekin avaimille k_3, k_4 voi olla $h(k_3) = h(k_4)$

- Tällöin sanotaan että avainten k_3 ja k_4 välillä sattuu yhteentörmäys (engl. collision)

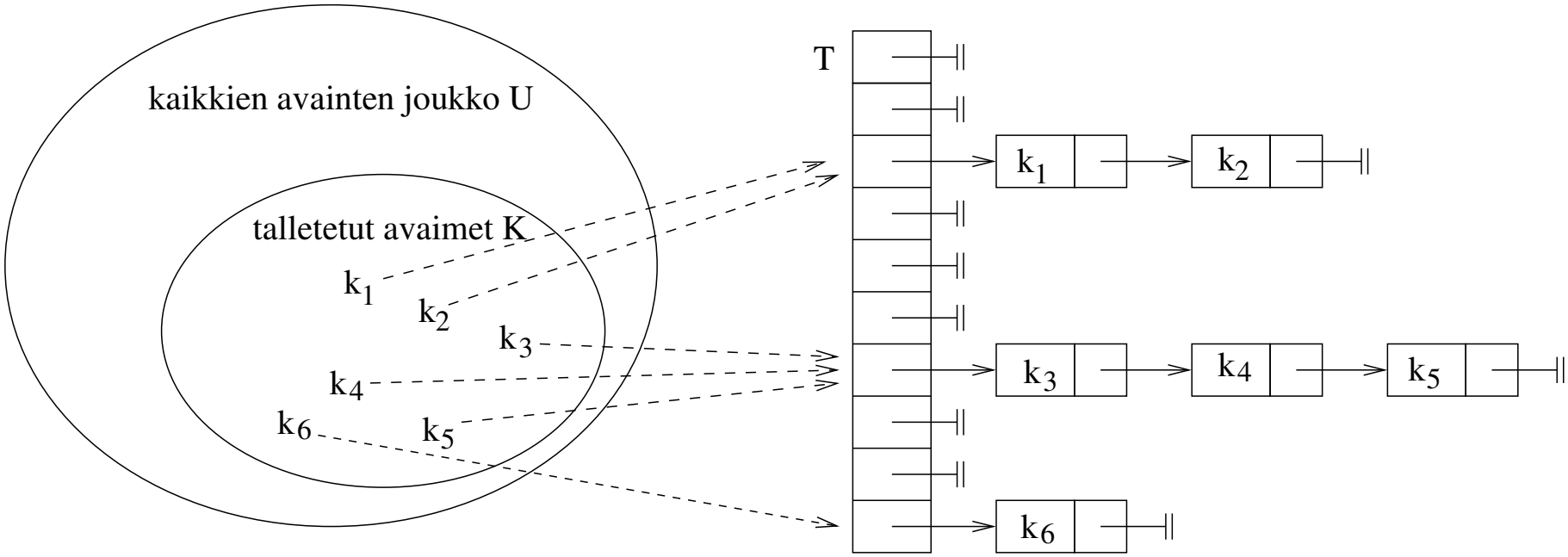
- Oleellista onkin suunnitella hajautusfunktio h sellaiseksi, että yhteentörmäyksiä tapahtuu mahdollisimman vähän. Palaamme hieman myöhemmin hajautusfunktion suunnitteluun

- Oli hajautusfunktio miten hyvä tahansa, ei jokaiselle avaimelle yleensä riitä omaa osoitetta

Yhteentörmäykset ylivuotoketjuun

- Yhteentörmäysten ratkaisemiseen on olemassa kaksi erilaista strategiaa. Käsitellään ensin ketjutus (engl. chaining), jossa yhteentörmäävät avaimet talletetaan linkitettyihin listoihin

- Alla olevassa kuvassa hajautustaulukon alkio i sisältää viitteen osoitteen i saavien avainten sisältämään listan



- Paikkaan $T[i]$ liittyvään listaan talletetaan siis kaikki osoitteen i saavat avaimet

- Esitetään nyt toteutus ketjutuksella varustettuun hajautukseen käyttäen linkitettyjä listoja (ks. luku 3)

- T on nyt taulukko joka on määritelty välille $[0, \ldots, m-1]$, ja jokainen $T[i]$ sisältää viitteen linkitetyn listan ensimmäiseen solmuun

- Alussa hajautusalue on tyhjä eli kaikkien $T[i]$ arvo on NIL

- Avaimen etsiminen hajautusrakenteesta:

  hash-search(T,k)
  1 L = T[h(k)]
  2 x = list-search(L,k)
  3 return x

  - rivillä 1 selvitetään missä taulun indeksiin liittyvässä listassa on avaimen paikka
  - rivillä 2 kutsutaan sivulla 199 määriteltyä listan etsintäoperaatiota selvittämään löytyykö etsittävää avainta
  - operaatio palauttaa viitteen siihen listasolmuun joka sisältää etsityn avaimen

- Avaimen lisääminen hajautusrakenteeseen

  hash-insert(T,k)
  1 L = T[h(k)]
  2 list-insert(L,k)

  – rivillä 1 selvitetään missä taulun indeksiin liittyvässä listassa on avaimen paikka

  – rivillä 2 kutsutaan sivulla 200 määriteltyä listan lisäysoperaatiota laittamaan avain paikoilleen

- Avaimen poistaminen hajautusrakenteesta

  hash-delete(T,x)
  1 L = T[h(x.key)]
  2 list-delete(L,x)

  – syötteenä siis viite poistettavan avaimen sisältämään listasolmuun; jos viitettä ei ole tallessa, on se haettava ensin **hash-search**-operaatiolla

  – rivillä 1 selvitetään missä taulun indeksiin liittyvässä listassa poistettava avain on

  – rivillä 2 kutsutaan sivulla 202 määriteltyä listan poisto-operaatiota, joka poistaa avaimen listalta

- Mikä on operaatioiden aikavaativuus?
- Oletetaan että hajautusfunktion arvon laskeminen vie vakioajan
- Luvusta 3 muistamme että lisäys ja poisto linkitetystä listasta vievät vakioajan (jos kyseessä on kahteen suuntaan linkitetty lista)
- Lisäys ja poisto hajautusrakenteesta vie aikaa $\mathcal{O}(1)$, sillä vakioajassa voimme selvittää minkä taulukon indeksin päästä löytyy oikea ylivuotolista ja sekä avaimen lisäys että poisto listalta ovat vakioaikaisia operaatioita
- Luvusta 2 muistamme että l avainta sisältävältä listalta etsintä vie aikaa $\mathcal{O}(l)$
- Hajautustaulusta etsinnän aikavaativuus siis riippuu ylivuotolistojen pituudesta
- Pahimmassa tapauksessa kaikki avaimet saavat saman osoitteen ja ne on talletettu samalle listalle eli pahimmassa tapauksessa haku n avainta sisältävästä hajautusrakenteesta vie ajan $\mathcal{O}(n)$
- Pahimman tapauksen tilanne on siis sama kuin avaimet olisi talletettu yhteen linkitettyyn listaan; selvästikään hajautusta ei ole tarkoitettu tällaisiin tilanteisiin, eli pahimman tapauksen analyysi ei anna oikeaa kuvaa menetelmästä

- Analysoidaan hakua keskimääräisessä tapauksessa
- Oletetaan että m-kokoiseen hajautustauluun on talletettu n avainta, ja että satunnaisella avaimella on sama todennäköisyys päätyä mille tahansa ylivuotolistalle

  (Hyvin valitulla hajautusfunktiolla tilanne on ainakin suurin piirtein tämä.)
- Hajautustaulun tämänhetkinen täyttösuhde (engl. load factor) on $\alpha = n/m$, joka on samalla keskimääräinen ylivuotoketjun pituus
- Avaimen haku hajautusrakenteesta vie aikaa keskimäärin $\mathcal{O}(1 + \alpha)$
- Todistushahmotelma (jossa todennäköisyyslaskelmat esitetään vain intuitiivisella tasolla):

  Todistus jakaantuu kahteen osaan: tuloksettomaan ja tuloksekkaaseen hakuun.

  Haettu avain voi olla yhtä suurella todennäköisyydellä missä tahansa ylivuotoketjussa. Avaimen k tuloksettomassa haussa käydään läpi ylivuotolista $T[h(k)]$, jonka pituus oletuksen mukaan on keskimäärin α. Kun otetaan vielä huomioon hajautusfunktion laskemiseen kuluva vakioaika saadaan vaatimukseksi $\mathcal{O}(1 + \alpha)$.

  Tuloksellisessa tapauksessa joudutaan keskimäärin käymään läpi puolet tietystä ylivuotolistasta $T[h(k)]$. Ylivuotolistan pituus on keskimäärin α, joten joudutaan tekemään karkeasti ottaen $\mathcal{O}(1 + \alpha/2)$ askelta.

- Yleensä pyritään siihen, että hajautustaulun koko m on suoraan verrannollinen talletettujen avaimien lukumäärän n
- Tällöin $\alpha = n/m = n/dn = \mathcal{O}(1)$ eli ylivuotoketjujen keskimääräinen pituus on vakio hajautusrakenteeseen talletettavien avaimien lukumäärän suhteen
- esim. Javan luokassa `HashMap` täyttösuhde α on oletusarvoisesti korkeintaan 0,75
- Kun $\alpha = \mathcal{O}(1)$, haun keskimääräinen aikavaativuus eli $\mathcal{O}(1 + \alpha)$ siis vakio $\mathcal{O}(1)$
- Suuri α tehostaa muistinkäyttöä mutta lisää yhteentörmäysten käsittelyyn kuluvaa aikaa
- Täyttösuhteen α virittäminen voi tehostaa rakennetta, mutta sen valintaan on vaikea antaa yleispäteviä sääntöjä

- Jos hajautustaulukon koko on suoraan verrannollinen talletettavien avaimien lukumäärään, operaatiot hajautustaulussa ovat keskimäärin vakioaikaisia
  - hakuoperaatio search vie keskimäärin vakioajan jos avaimet ovat jakautuneet tasaisesti ylivuotolistoille
  - **hash-insert** ja **hash-delete** ovat vakioaikaisia myös pahimmassa tapauksessa

- Useissa sovelluksissa tiedetään suunnilleen talletettavien avaimien lukumäärän yläraja. Tällöin on helppo valita hajautusrakenteen koko sopivasti

- Edellä taulukon kussakin paikassa on vain viite listaan (suora ketjutus), mutta vaihtoehtoisesti voi tehdä niin, että taulukossa itsessään varataan tilaa yhdelle avaimelle osoitetta kohti, ja vasta yhteentörmäysten sattuessa aletaan rakentaa linkitettyä listaa (erillinen ketjutus)

Hajautusfunktion valinta

- Tavoitteena on, että hajautusfunktion arvo on laskettavissa nopeasti ja funktio jakaa avaimet tasaisesti hajautusalueelle
- Hajautusfunktio olettaa yleensä että avain on kokonaisluku. Jos avain on kuitenkin esim. merkkijono, se on ensin muutettava kokonaisluvuksi

  olkoon avain merkkijono $a_1 a_2 \ldots a_n$, oletetaan että funktio $ascii(a)$ palauttaa merkin a ascii-koodin, joka on kokonaisluku väliltä 0–127

  eräs tapa muuttaa merkkijono kokonaisluvuksi k on seuraava

  $k = ascii(a_1) + 128 \cdot ascii(a_2) + 128^2 \cdot ascii(a_3) + \ldots + 128^{n-1} \cdot ascii(a_n)$
- Tähän sisältyy muutama ongelma
  - luvusta tulee suuri ja seurauksena saattaa olla aritmeettinen ylivuoto
  - ongelma korjautuu tulkitsemalla luku etumerkittömäksi kokonaisluvuksi ja yhdistämällä se seuraavan sivun jakolaskumenetelmään
  - toisaalta jos merkkijono on pitkä, voi merkkijonoa vastaavan kokonaislukuarvon laskeminen kestää kauan
  - merkkijonon jokaista merkkiä ei ole välttämätöntä huomioida, voidaan esim. ottaa mukaan vain joka toinen merkki, viisi ensimmäistä merkkiä, jne.
- Seuraavassa käytännössä toimivaksi osoittautuneita hajautusfunktioita

Jakojäännösmenetelmä

- $h(k) = k \ mod \ m$

  missä m on alkuluku, joka ei ole lähellä mitään 2:n potenssia

  näin valittu m yleensä jakaa avaimet suhteellisen tasaisesti hajautustauluun vaikka avaimet itsessään olisivat jossain määrin epätasaisesti jakautuneet

  (seuraavalla kalvolla perustellaan näitä ehtoja m:lle)

- Esim. jos haluamme hajautustaulun johon talletetaan noin 2000 avainta, ja sallimme että ylivuotolistoilla on keskimäärin 3 avainta, valitaan esim. $m = 701$ (Lähimmät kakkosen potenssit ovat $2^9 = 512$ ja $2^{10} = 1024$)

- Nyt m on alkuluku joka ei ole lähellä 2:n potenssia ja täyttöasteeksi tulee $2000/701 \approx 2.85$

Siis jakojäännösmenetelmää käytettäessä suositellaan, että hajautustaulun koko m on

- alkuluku
- ei lähellä mitään kakkosen potenssia

Miksi alkuluku:

- oletetaan että $m = pq$, missä p, q ykköstä suurempia kokonaislukuja
- jos hajautetaan avaimet $q, 2q, 3q, \ldots$, niiden kotiosoitteet $h(kq)$ sattuvat kaikki joukkoon $\{p, 2p, 3p, \ldots\}$
- avainten jaollisuus yhteisellä tekijällä q on tilanne johon halutaan varautua

Miksi ei lähellä kakkosen potenssia:

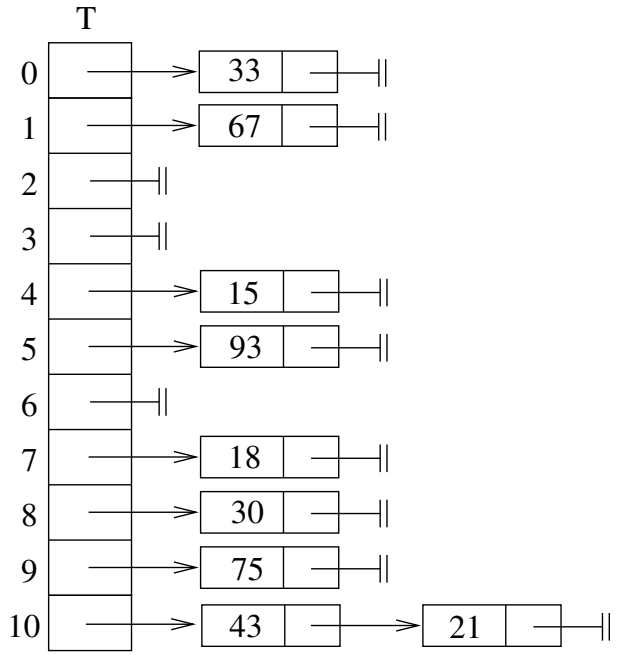
- oletetaan esim. $m = 2^8 - 1$
- hajautetaan 32-bittinen luonnollinen luku x, jonka binääriesitys on $x_1 x_2 \ldots x_{32}$
- on melko helppo nähdä, että $h(x) = x \bmod m$ määräytyy summasta $x_1 \ldots x_8 + x_9 \ldots x_{16} + x_{17} \ldots x_{24} + x_{25} \ldots x_{32}$ (missä siis 32-bittisestä luvusta on lohkottu neljä 8-bittistä ja laskettu ne yhteen)

- **Esimerkki** Valitaan hajautustaulukon kooksi $m = 11$, joten hajautusfunktiona on $h(k) = k \bmod 11$

- Hajautetaan avaimet 75, 21, 43, 15, 18, 33, 30, 67 ja 93, eli nyt

  $h(75) = 9$, $h(21) = 10$, $h(43) = 10$, $h(15) = 4$, $h(18) = 7$,
  $h(33) = 0$, $h(30) = 8$, $h(67) = 1$, $h(93) = 5$

- Tuloksena

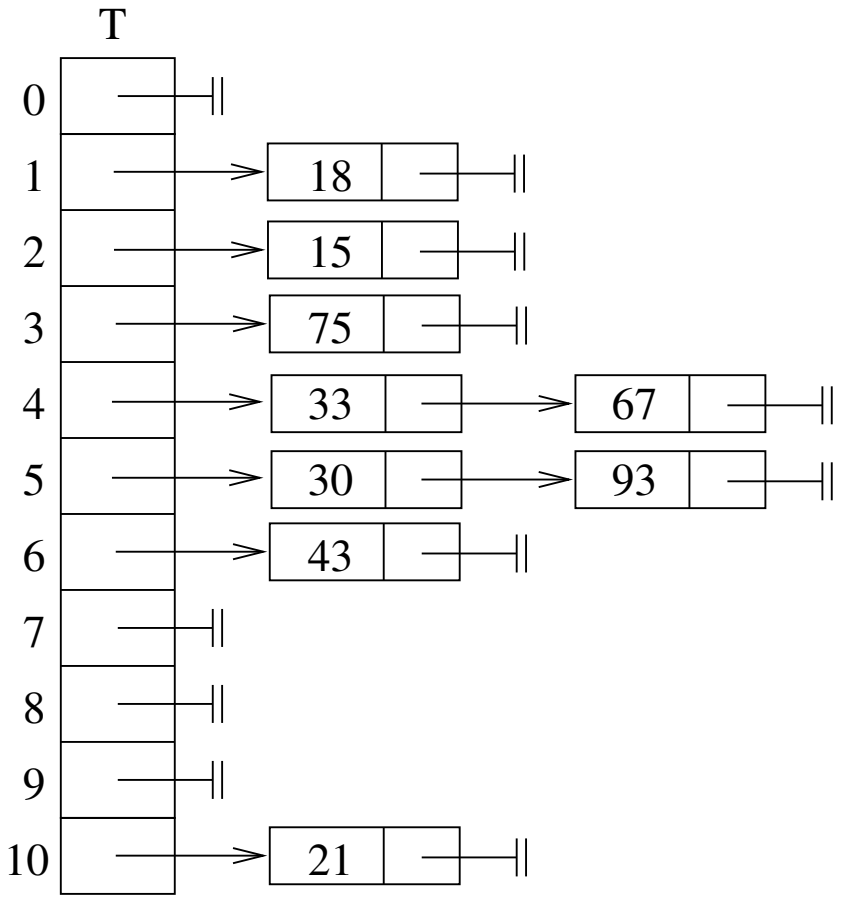


Kertolaskumenetelmä

- Olkoon A jokin luku $0 < A < 1$
- Merkitään $\lfloor x \rfloor$:llä luvun x kokonaislukuosaa ja fr(x):lla luvun x desimaaliosaa, eli esim $\lfloor 3{,}14159 \rfloor = 3$ ja fr(3,14159) = 0,14159
- $h(k) = \lfloor m \cdot \text{fr}(Ak) \rfloor$
- Eräs suositeltu valinta on $A = \frac{\sqrt{5}-1}{2} = 0{,}6180339887\ldots$ (ks. Knuth: *The Art of Computer Programming*)
- Parametri m ei suuremmin vaikuta hajautuksen tasaisuuteen. Voidaan valita esim. $m = 2^p$, jollekin kokonaisluvulle p, jolloin laskenta bittitasolla on helppoa (ks. Cormen luku 11.3.2)
- Esim. oletetaan että käytössä on hajautustaulukko jonka koko on 11, ja hajautusfunktiona $h(k) = \lfloor 11 \cdot \text{fr}(0{,}618 \cdot k) \rfloor$
- Avaimet 75, 43, 21, 15, 18, 33, 30, 67 ja 93, nyt

  $h(75) = 3$, $h(43) = 6$, $h(21) = 10$, $h(15) = 2$, $h(18) = 1$
  $h(33) = 4$, $h(30) = 5$, $h(67) = 4$, $h(93) = 5$

- Tuloksena



- Hajautusfunktio pitäisi valita siten, että hajautusosoitteiden $h(k)$ jakauma olisi mahdollisimman lähellä joukon $\{0, \ldots, m-1\}$ tasaista jakaumaa
- Tätä kriteeriä on mahdoton arvioida, jos ei tiedetä avainten k jakaumaa (ja vaikka tiedettäisiin, se voi olla käytännössä hyvin vaikeaa)
- Ohjelmoija voi tyhmyyttään valita sovellusta ajatellen juuri väärän hajautusfunktion
- Jos tehdään huono valinta hajautusfunktioksi, hajautus toimii jokaisella suorituskerralla hitaasti
- Universaalihajautuksessa valitaan jokaisella ohjelman suorituskerralla satunnainen hajautusfunktio, joka millä tahansa avainten jakaumalla on keskimäärin hyvä
- Huonon hajautusfunktion saaminen edellyttää (hyvin) huonoa tuuria
- Universaalihajautuksessakin voi joskus syntyä sovelluksen käsiteltävään dataan nähden huono hajautusfunktio
- Keskimäärin näin käy kuitenkin erittäin harvoin
- Universaalihajautus toimii odotusarvoisesti nopeasti riippumatta hajautettavien avaimien jakaumasta

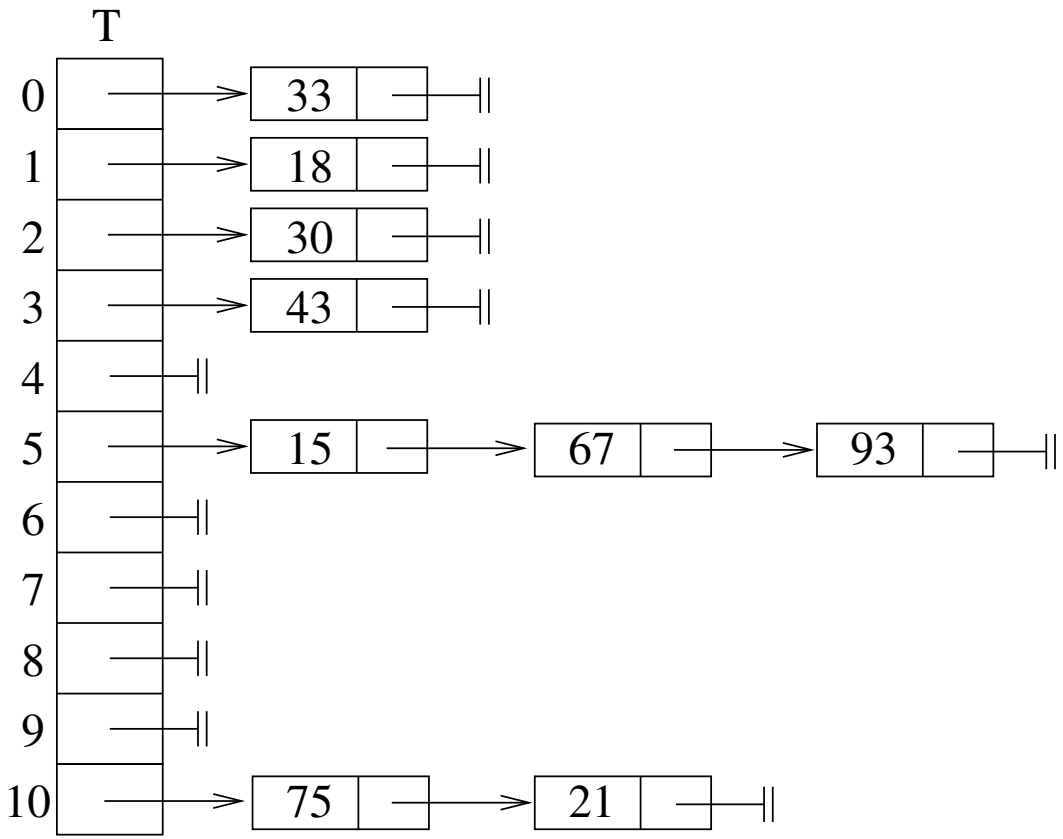
Universaalihajautus

- Valitaan alkuluku p joka on vähintään yhtä suuri kuin talletettavien avainten lukumäärä n
- Valitaan satunnaisesti kaksi kokonaislukua a ja b väliltä $1 \leq a < p$ ja $0 \leq b < p$
- Muodostetaan hajautusfunktio seuraavasti: $h(k) = ((ak + b) \bmod p) \bmod m$
- Perusteluja näin muodostetun hajautusfunktion hyvyydestä löytyy Cormenin luvusta 11.3.3
- Esim. oletetaan jälleen että käytössä on hajautustaulukko jonka koko on 11
- Oletetaan että talletettavia avaimia korkeintaan 53 eli valitaan $p = 53$, ja valitaan satunnaisesti $a = 31$, $b = 17$

  hajautusfunktiona siis $h(k) = ((31 \cdot k + 17) \bmod 53) \bmod 11$
- Avaimet 75, 43, 21, 15, 18, 33, 30, 67 ja 93, nyt

  $h(75) = 10 \bmod 11 = 10$, $h(43) = 25 \bmod 11 = 3$, $h(21) = 32 \bmod 11 = 10$
  $h(15) = 5 \bmod 11 = 5$, $h(18) = 45 \bmod 11 = 1$, $h(33) = 33 \bmod 11 = 0$
  $h(30) = 46 \bmod 11 = 2$, $h(67) = 27 \bmod 11 = 5$ ja $h(93) = 38 \bmod 11 = 5$

- Tuloksena



- Valitsemalla satunnaiset a ja b toisin olisi päädytty eri arvot antavaan hajautusfunktioon, jonka käyttö olisi saattanut johtaa parempaan tulokseen

Uudelleenhajautus

- Kertauksena sivulla 226:
  - jos hajautustaukon koko m valitaan suoraan verrannolliseksi talletettujen avaimien lukumäärän n, ylivuotoketjujen keskimääräinen pituus on $\alpha = n/m = \mathcal{O}(1)$ eli vakio hajautusrakenteeseen talletettavien avaimien lukumäärän suhteen

- Eli jos hajautustaulukon koko on suoraan verrannollinen talletettavien avaimien lukumäärään, ovat operaatiot hajautustaulussa vakioaikaisia

- Jos hajautustaulun koko on vakio, esim. 1000, ja rakenteeseen talletettavien alkioiden määrä n kasvaa rajatta, on yhden ylivuotoketjun keskimääräinen pituus $n/1000$, eli $\mathcal{O}(n)$

- Siis jos talletettavien alkioiden määrä n kasvaa rajatta, mutta taulun koko m on vakio, ylivuotoketjujen keskimääräiseksi pituudeksi tulee $\mathcal{O}(n)$ ja etsintä hajautusrakenteesta vie lineaarisen ajan

- Hajautusrakenteen täyttymisongelma voidaan ratkaista uudelleenhajautuksella
  - kun täyttösuhde n/m ylittää asetetun kynnysarvon $kynnys$ (esim. Javan `HashMap`issä 0,75), varataan hajautustaululle lisäyksen yhteydessä uusi kooltaan kaksinkertainen tila
  - kaikki alkuperäisen rakenteen avaimet talletetaan uuteen hajautusrakenteeseen käyttäen uutta hajautusfunktiota
  - uuden hajautusrakenteen täyttösuhde on $1/2 \cdot kynnys$
  - jos poiston jälkeen rakenteen täyttösuhde on alle $1/4 \cdot kynnys$ voidaan tilaa vapauttaa varaamalla uusi rakenne, kooltaan puolet alkuperäisestä
  - tässäkin tapauksessa kaikki alkuperäisen rakenteen avaimet talletetaan uuteen hajautusrakenteeseen käyttäen uutta hajautusfunktiota
  - uuden hajautusrakenteen täyttösuhde on $1/2 \cdot kynnys$
  - molemmissa tapauksissa uudelleenhajautuksen aikavaatimus on $\mathcal{O}(m)$, missä m alkuperäisen hajautusrakenteen koko

- Monisteen sivuilla 176–177 tarkasteltiin taulukon laajentamisen aikavaativuutta pinon yhteydessä

- Totesimme (ilman täsmällistä perustelua), että jos taulukon koko lisäyksen yhteydessä tarvittaessa kaksinkertaistetaan, ei kasvatus lisää yhden operaation keskimäärin vievää aikavaativuutta kuin vakion verran

- Samantapainen analyysi osoittaa, että vaikka uudelleenhajautus eli hajautustaulun kasvattaminen on raskas operaatio, niin jos ajatellaan pidempiä sarjoja hajautusrakenteelle suoritettuja operaatiota, yhden operaation keskimääräinen aikavaativuus on silloin tällöin tapahtuvasta uudelleenhajautuksesta huolimatta vakio

- Jos uudelleenhajautusta ei suoriteta ja talletettavien alkioiden määrä kasvaa jatkuvasti, **hash-search**-operaation keskimääräinen aikavaativuus ei enää säily vakiona

- Eli ajoittaisesta raskaudestaan huolimatta uudelleenhajautus kannattaa jos hajautusrakenteeseen talletettavien alkioiden lukumäärällä ei ole ylärajaa

Avoin hajautus

- Ketjutuksen rinnalla toinen tapa ratkaista yhteentörmäävien solmujen ongelma on avoin hajautus (engl. open addressing)
- Avoimessa hajautuksessa kaikki avaimet talletetaan hajautustauluun
- Jos paikka $T[h(k)]$ on varattu, etsitään avaimelle k jokin muu paikka taulusta
- Uusikin paikka voi olla varattu, tällöin jatketaan etsimistä edelleen
- Ne paikat joihin avainta k yritetään laittaa muodostavat k:n kokeilujonon (engl. probe sequence)
- Järjestyksessään j:s kokeilu kohdistuu paikkaan

  $h(k, j) = (h'(k) + s(j, k)) \bmod m$,

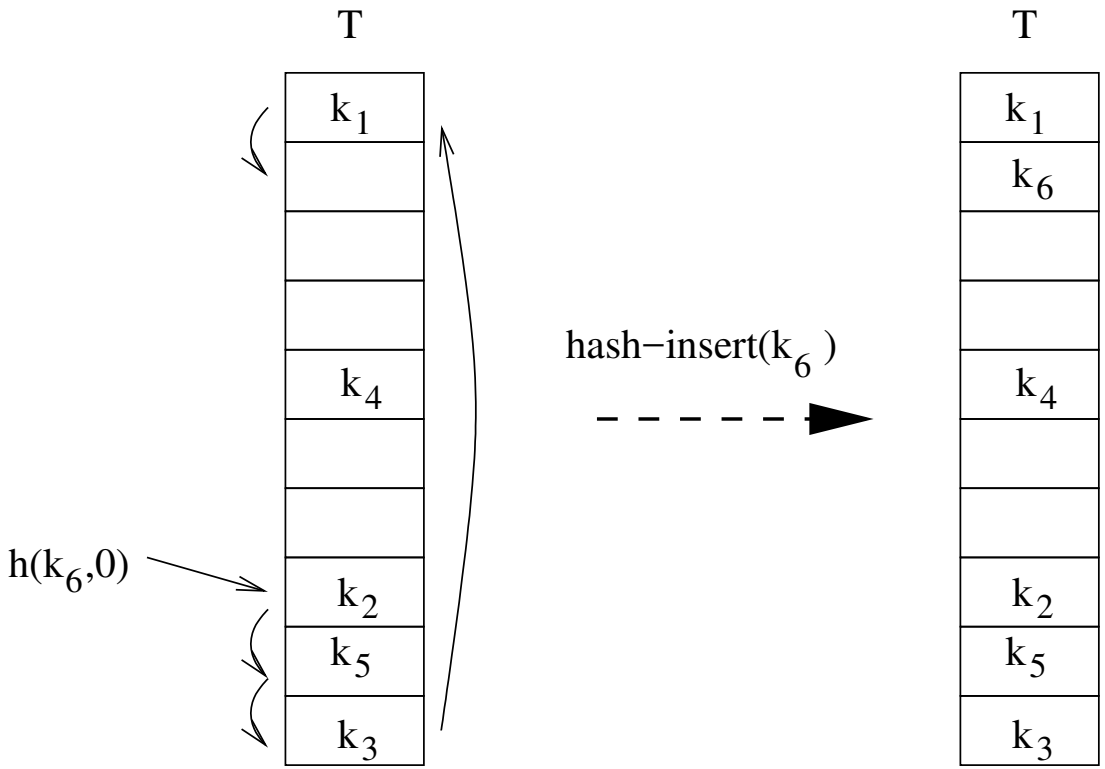
  missä h' on "normaali" hajautusfunktio ja s on kokeilufunktio
- Vaaditaan että jokaiselle avaimelle k kokeilujono $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ muodostaa jonon $0, 1, \ldots, (m-1)$ permutaation, eli toisin sanoen kokeilujonon on kokeiltava jokaista hajautustaulun indeksiä kertaalleen
- Avaimen k lisääminen tapahtuu siten että ensin yritetään laittaa avain paikkaan $h(k, 0)$, jos tämä paikka on täysi, yritetään paikkaa $h(k, 1)$, jne

- Kaikki avaimet talletetaan hajautustauluun joten avaimien maksimimäärä on m
- Oletetaan että jos hajautustaulun paikka $T[i]$ on tyhjä, sillä on erityisarvo NIL
- Ennen kuin katsotaan miten operaatiot toteutetaan avoimessa hajautuksessa, tutustumme yksinkertaisimpaan kokeilustrategiaan, eli lineaariseen kokeiluun (engl. linear probing)
  - kokeilujono on nyt $h(k, j) = (h'(k) + j) \bmod m$, eli ensimmäinen paikka on $h'(k) \bmod m$, toinen $(h'(k) + 1) \bmod m$, kolmas $(h'(k) + 2) \bmod m$, ...
  - esim. jos $m = 10$ ja $h'(k) = 7$ niin kokeilujono olisi $7, 8, 9, 0, 1, 2, 3, 4, 5, 6$
- Avaimen lisääminen hajautustauluun

```
hash-insert(T,k)
1   i = 0
2   repeat
3       j = h(k,i)
4       if T[j] == NIL
5           T[j] = k
6           return true
7       i = i+1
8   until i == m
9   return false
```

- Jos paikka avaimelle löytyy, palauttaa operaatio arvon true, mutta jos mikään kokeilujonon paikka ei ole vapaana, eli hajautustaulu on jo täysi, palauttaa operaatio arvon false

- Operaation toiminta:

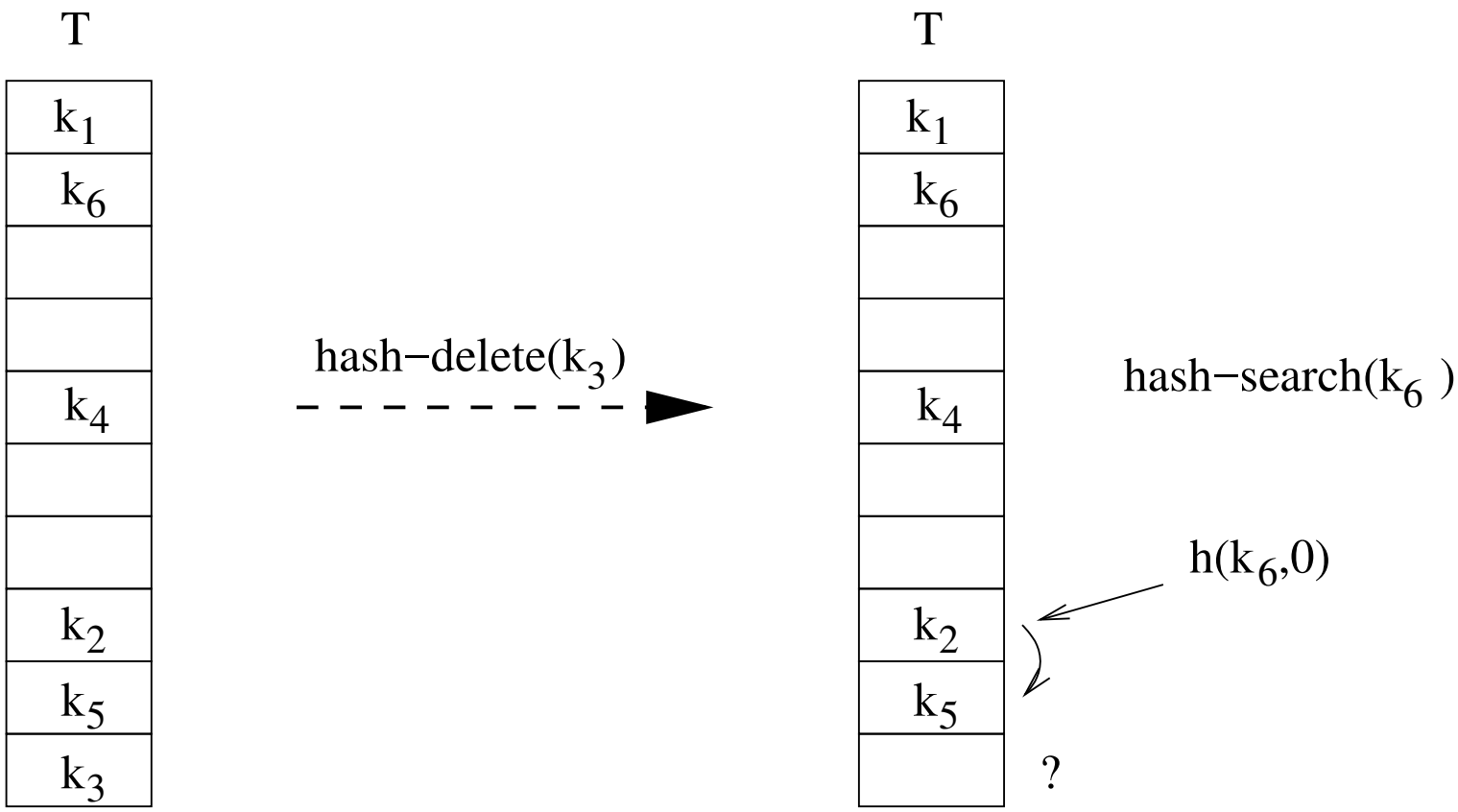


- Avaimen etsiminen hajautustaulusta

```
hash-search(T,k)
1   i = 0
2   repeat
3       j = h(k,i)
4       if T[j] == k return j
5       i = i+1
6   until T[j] == NIL or i == m
7   return NIL
```

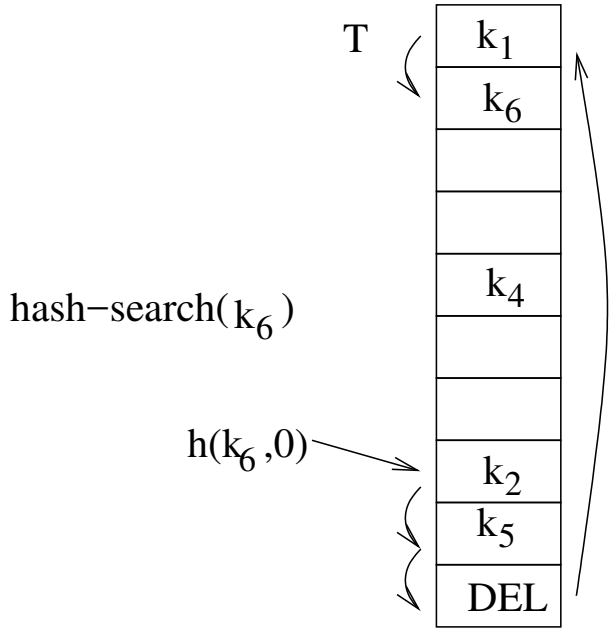
- Etsitään kokeilujonoa $h(k,0), h(k,1), \ldots, h(k,m-1)$ läpi niin kauan kun
  - etsitty avain löytyy,
  - törmätään tyhjään hajautustaulun paikkaan, tai
  - koko kokeilujono on käyty läpi
- Avaimen löytyessä palautetaan avaimen tallettavan taulukon paikan indeksi, muuten NIL

- Avaimen poistamisen yhteydessä ei paikkaa voida jättää vapaaksi (NIL-arvoisiksi), muuten reitti joihinkin avaimiin saattaa katketa:



- Merkitään poiston yhteydessä poistettavan avaimen paikalle erityisarvo DEL niin avaimen etsiminen toimii taas

```
hash-delete(T,k)
1   i = 0
2   repeat
3       j = h(k,i)
4       if T[j] == k
5           T[j] = DEL
6       i = i+1
7   until T[j] == NIL or i == m
```



- Muutetaan vielä lisäysoperaation riviä 5 siten että lisäys voidaan suorittaa taulukon paikkaan jonka arvo on NIL tai DEL

```
hash-insert(T,k)
1   i = 0
2   repeat
3       j = h(k,i)
4       if T[j] == NIL or T[j] == DEL
5           T[j] = k
6           return true
7       i = i+1
8   until i = m
9   return false
```

- Avoimen hajautuksen poisto-operaatio on laiska, se ei vapauta muistialuetta, vaan jättää taulukkoon DEL-arvoisia kohtia jotka täyttävät hajautustaulukkoa

- DEL-kohdat voivat korvautua normaaleilla avaimilla **hash-insert**:in yhteydessä

- Esim. oletetaan että käytössä hajautustaulukko jonka koko 11
- Käytetään kokeilujonofunktiota $h(k,i) = ((k \bmod 11) + i) \bmod 11$ käytössä siis lineaarinen kokeilujono
- Hajautetaan avaimet 75, 43, 21, 15, 18, 33, 30, 66 ja 92
- $h(75,0) = 9$ ja $h(43,0) = 10$, eli kahden ensimmäisen lisäyksen jälkeen

0	1	2	3	4	5	6	7	8	9	10
									75	43

- $h(21,0) = 10$ eli joudumme etsimään 21:lle uuden paikan
  $h(21,1) = ((21 \bmod 11) + 1) \bmod 11 = 0$ joka on vapaa.
- $h(15,0) = 4$ ja $h(18,0) = 7$, tilanne viiden ensimmäisen lisäyksen jälkeen:

0	1	2	3	4	5	6	7	8	9	10
21				15			18		75	43

- $h(33, 0) = 0$ joka jo varattu, eli etsintä jatkuu $h(33, 1) = 1$. Seuraava avain voidaan sijoittaa heti kokeilujonon ensimmäiseen paikkaan $h(30, 0) = 8$. Tilanne tässä vaiheessa:

0	1	2	3	4	5	6	7	8	9	10
21	33			15			18	30	75	43

- $h(66, 0) = 0$ ei käy, $h(66, 1) = 1$ ei vieläkään, mutta $h(66, 2) = 2$ onnistuu

- Lopuksi $h(92, 0) = 4$, ei käy, mutta $h(92, 1) = 5$ vapaa

- Tuloksena

0	1	2	3	4	5	6	7	8	9	10
21	33	66		15	92		18	30	75	43

- Entä jos haluamme 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? Etsintä etenisi paikasta $h(41, 0) = 7$ aina paikkaan $h(41, 7) = 3$ asti jonka jälkeen voidaan todeta että 41 ei ole taulukossa

- Poistetaan taulukosta avaimet 21, 30 ja 75:

0	1	2	3	4	5	6	7	8	9	10
DEL	33	66		15	92		18	DEL	DEL	43

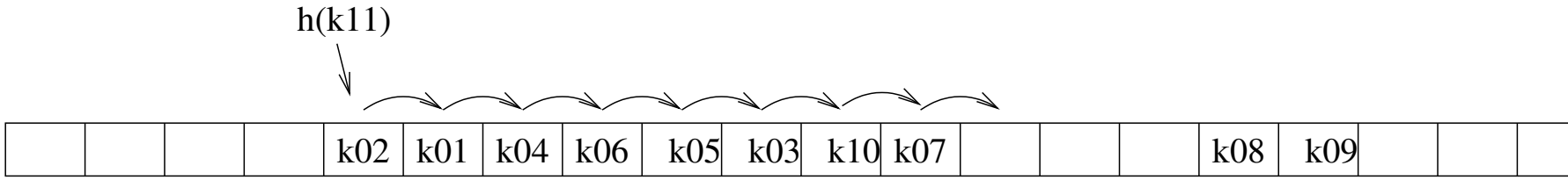
- Edelleen, luvun 41 etsintä etenisi paikasta $h(41, 0) = 7$ aina paikkaan $h(41, 7) = 3$ asti jonka jälkeen voidaan todeta että 41 ei ole taulukossa

- Luvun 29 lisäys löytäisi nyt 29:lle paikan toisella kokeilulla $h(29, 1) = 8$ sillä $T[8]$ =DEL

- Taulukko luvun 29 lisäyksen jälkeen

0	1	2	3	4	5	6	7	8	9	10
DEL	33	66		15	92		18	29	DEL	43

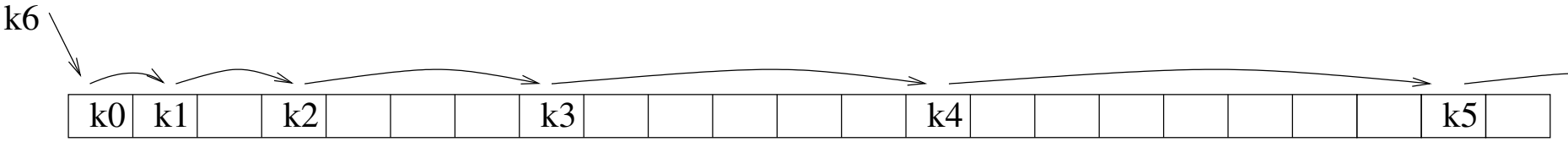
- Lineaarinen kokeilujono on helppo toteuttaa mutta käytännössä aika huono:
  - tauluun tulee usein pitkiä varattuja alueita
  - pitkät varatut alueet kasvavat suuremmalla todennäköisyydellä kuin lyhyet varatut osat:



- Ilmiötä nimitetään primääriseksi kasautumiseksi (engl. primary clustering)
- Varattuja alueita muodostuu aina jonkin verran, koska yhteentörmäyksiä sattuu pienilläkin täyttöasteilla
  - syntymäpäiväparadoksi: jos taulukossa on $\sqrt{2m}$ avainta, todennäköisyys välttyä kokonaan yhteentörmäyksistä on alle $1/2$, vaikka täyttöaste on $\alpha = \sqrt{2m}/m = 2/\sqrt{m} \ll 1$

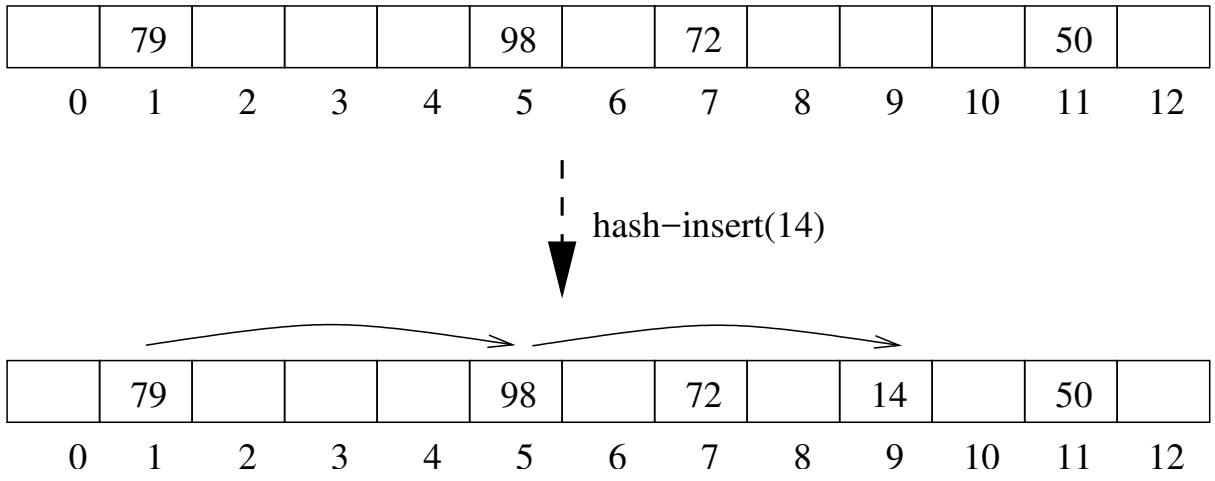
- Neliöinen kokeilu
  - kokeilujono on nyt $h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$, missä h' normaali hajautusfunktio, c_1 ja c_2 vakioita
  - kokeilujonon ensimmäinen paikka on $h'(k) \bmod m$, toinen $(h'(k) + c_1 + c_2) \bmod m$, kolmas $(h'(k) + 2c_i + 4c_2) \bmod m$, jne
  - huom: kaikki valinnat vakioiksi c_1, c_2 ja m eivät tuota kokeilujonoa joka käy taulukon kaikki paikat läpi

    yksi toimiva tapa valita vakiot on asettaa $c_1 = c_2 = 1/2$ ja valita hajautustaulun kooksi m jokin kahden potenssi
  - esim jos $c_1 = c_2 = 1/2$ ja $h'(k_1)$=1 niin kokeilujonon alku olisi $1, 2, 4, 7, 11, 16 \ldots$ eli $h(k,1) = h(k,0) + 1$, $h(k,2) = h(k,0) + 3 = h(k,1) + 2$, $h(k,3) = h(k,0) + 6 = h(k,2) + 3$ ...
- Primääriseltä kasaantumiselta vältytään, mutta nyt vaivana on sekundäärinen kasautuminen (engl. secondary clustering): jos $h(k_1,0) = h(k_2,0)$ niin k_1:n ja k_2:n kokeilujono on sama

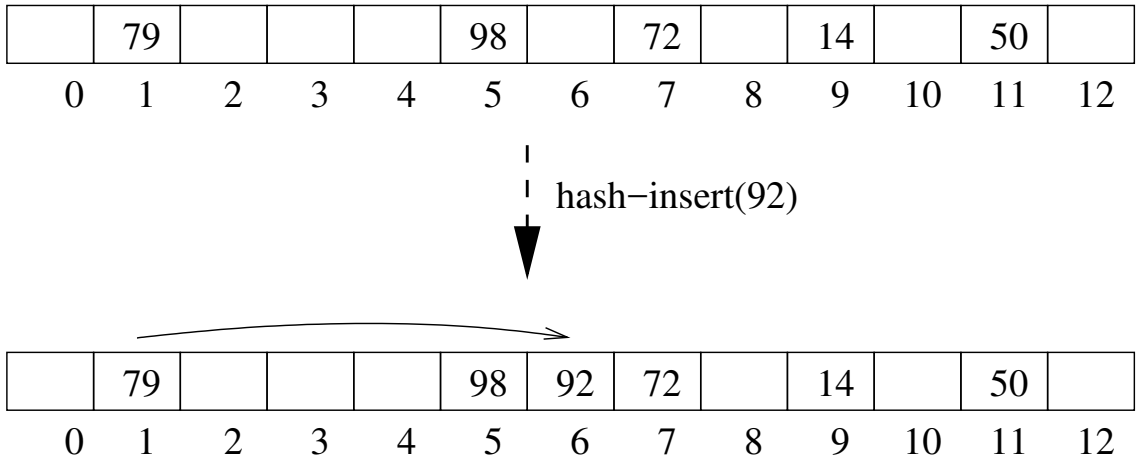


h(k0,0) = h(k1,0) = h(k2,0) = h(k3,0) = h(h4,0) = h(k5,0) = h(k6,0) = 1

- Kaksoishajautus (engl. double hashing)
  - kokeilujono on nyt $h(k,i) = (h'(k) + i \cdot h''(k)) \bmod m$, missä h' ja h'' normaaleja hajautusfunktioita
  - ensimmäinen paikka on $h'(k) \bmod m$, toinen $(h'(k) + h''(k)) \bmod m$, kolmas $(h'(k) + 2 \cdot h''(k)) \bmod m$, jne
  - esim. tarkastellaan 13 paikkaista hajautustaulua ja olkoot $h'(k) = k \bmod 13$ ja $h''(k) = 1 + (k \bmod 11)$
  - lisätään hajautustauluun avain 14
  - $h'(14) = 14 \bmod 13 = 1$, $h''(14) = 1 + (14 \bmod 11) = 4$,
  - kokeilujono on siis $1, 5, 9, 0, 4, 8, 12, \ldots$



- Lisätään hajautustauluun avain 92
  - $h'(92) = 92 \bmod 13 = 1$, $h''(92) = 1 + (92 \bmod 11) = 5$,
  - kokeilujono on siis $1, 6, 11, 3, 8, 0 \ldots$



- Kaksoishajautus ei aiheuta kasaantumista ja on yleensä menetelmistä toimivin
- Funktion h'' valinnassa on kuitenkin oltava huolellinen, arvoilla $h''(x)$ ja m ei saa olla yhteisiä tekijöitä, muuten osa hajautustaulusta voi jäädä käymättä läpi
- Yksi hyvä valinta on edellisessäkin esimerkissä esiintynyt tilanne missä m on alkuluku ja $0 < h''(x) < m$, eli valitaan esim

  $h'(k) = k \bmod m$ ja $h''(k) = 1 + (k \bmod (m-2))$

- Avoimen hajautuksen tapauksessa kaikkien operaatioiden tehokkuus on riippuvainen avaimen löytymisen tehokkuudesta

- Seuraavassa taulukossa tuloksettoman ja tuloksellisen haun keskimääräinen pituus kullakin kokeilujonotyypillä suhteessa täyttösuhteeseen $\alpha = n/m$. Kaavoja ei tässä perustella tarkemmin, niiden taustalla oleva analyysi on esitetty Cormenin luvussa 11.4

	tulokseton	tuloksellinen
kaksoishajautus	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$
neliöinen	$\frac{1}{1-\alpha} - \alpha + \ln\frac{1}{1-\alpha}$	$1 - \frac{2}{\alpha} + \ln\frac{1}{1-\alpha}$
lineaarinen	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$	$\frac{1}{2}(1 + \frac{1}{1-\alpha})$

- Seuraavalla sivulla vielä hakujen keskimääräisiä pituuksia muutamalla konkreettisella täyttösuhteen arvolla

$\alpha = 0{,}5$	tulokseton	tuloksellinen
kaksoishajautus	2	1,38
neliöinen	2,19	1,44
lineaarinen	2,5	1,5

$\alpha = 0{,}75$	tulokseton	tuloksellinen
kaksoishajautus	4	1,85
neliöinen	4,63	2,01
lineaarinen	8,5	2,5

$\alpha = 0{,}9$	tulokseton	tuloksellinen
kaksoishajautus	10	2,55
neliöinen	11	2,88
lineaarinen	50,5	5,5

$\alpha = 0{,}95$	tulokseton	tuloksellinen
kaksoishajautus	20	3,15
neliöinen	22	3,53
lineaarinen	200,5	10,5

- Näyttää siltä ettei ole suurta eroa menetelmien välillä täyttösuhteeseen $0,5$ asti
- Tätä täydemmillä hajautustauluilla lineaarista kokeilujonoa ei kannata käyttää
- Ratkaisu avoimen hajautuksen yhteydessä taulukon täyttyessä on kasvattaa hajautustaulukkoa ja suorittaa alkioille uudelleenhajautus

Käytännöllisiä huomautuksia hajautuksesta

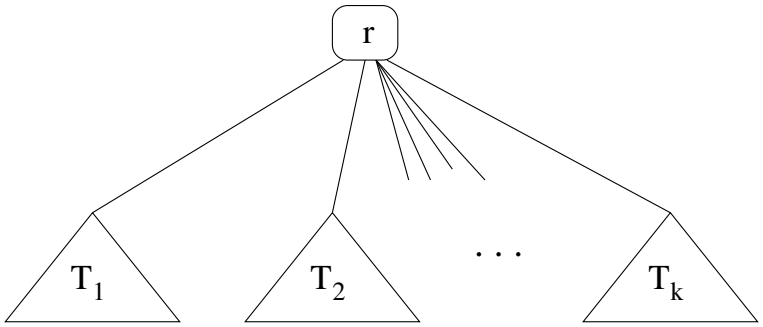
- Hajautusfunktioista jakojäännösmenetelmä on yleensä hyvä
- Usein aineiston kanssa kannattaa tehdä muutamia kokeita ennen kuin käytettävä hajautusfunktio valitaan
- Yhteentörmäysstrategioista ketjutus on yleensä paras ratkaisu
  - toimii vaikka hajautusalue täynnä; poistot aitoja toisin kuin avoimessa hajautuksessa
  - mutta samat tehokkuuteen liittyvät ongelmat kuin linkitetyillä listoilla yleensäkin (s. 185)
- Avoin hajautus on hyvä jos täyttösuhde säilyy pienenä ja jos koko taulukko mahtuu nopeaan muistiin
- Hajautus on yleensä käytännössä tehokkain menetelmä joukko-operaatioiden toteuttamiseen jos operaatioita **min**, **succ**, **max** ja **pred** ei tarvita
- Operaatioiden **min**, **succ**, **max** ja **pred** toteuttaminen hajautusrakenteissa on toivottoman hidasta eli jos näitä operaatioita tarvitaan, kannattaa käyttää pian nähtävää tasapainotettua hakupuuta
- **hash-search**-operaatio toimii vakioaikaisesti keskimääräisessä tapauksessa, eli jos halutaan takuu että pahin tapaus ei koskaan toteudu, on syytä käyttää tasapainoitettua hakupuuta

Java-API:n hajautusrakenteet

- Javassa on kaksi hieman eri tilanteisiin sopivaa hajautusrakennetta (näistä molemmista on myös pari erikoistettua versiota)
  - Rajapinnan `Set` toteuttava `HashSet`
  - Rajapinnan `Map` toteuttava `HashMap`
- `HashSet` tarjoaa operaatiot olion tallettamiseen ja etsimiseen, olio itse toimii hajautusavaimena
  - `HashSet`:in avulla ei siis ole mahdollista etsiä oliota nopeasti minkään attribuutin perusteella, ajassa $\mathcal{O}(1)$ voidaan ainoastaan kysyä onko tietty olio joukossa
- `HashMap`:iin talletetaan avain-olio-pareja, ja etsintä avaimeen perustuen tapahtuu keskimäärin ajassa $\mathcal{O}(1)$
- Huomionarvoista on, että jos itse määritellyn luokan olioita talletetaan `HashSet`:iin tai käytetään `HashMap`:in avaimina, on luokassa toteutettava (tarkemmin sanottuna ylikirjoitettava `Object`-luokassa määritellyt) metodit `int hashCode()` ja `boolean equals(Object param)`, muuten hajautusrakenne ei toimi ennakoidulla tavalla

6. Hakupuut

- Hakupuu (engl. search tree) on listaa huomattavasti edistyneempi tapa toteuttaa abstrakti tietotyyppi joukko
- Puurakenteelle on tietojenkäsittelyssä myös muuta käyttöä, esim. algoritmin suoritusajan analysoinnissa, ohjelman laskennan etenemisen kuvailussa ja ongelmanratkaisussa
- Ennen kuin menemme hakupuihin, tutustutaan yleiseen puihin liittyvään käsitteistöön
- (Juurellinen) puu on kokoelma solmuja (engl. node, vertex) ja niitä yhdistäviä kaaria (engl. edge), siten että:
  - kokoelma on tyhjä, tai
  - yksi solmuista r, on juuri (engl. root) johon kaaret liittävät nolla tai useampia alipuita (engl. subtree) $T_1, \ldots T_k$, jotka itsekin ovat puita



- Edellä annettu puun määritelmä on rekursiivinen, eli puu koostuu juuresta ja siihen liittyvistä alipuista, jotka ovat itsekin puita

- Verkkojen (eli graafien) teoriassa puu tarkoittaa yhtenäistä syklitöntä verkkoa. Jos verkkoteoreettiselle puulle on lisäksi nimetty juuri, se on jokseenkin sama käsite kuin tässä tarkasteltava. Palaamme verkkoihin myöhemmin kurssilla.

- Tietojenkäsittelytieteessä puu piirretään juuri ylöspäin

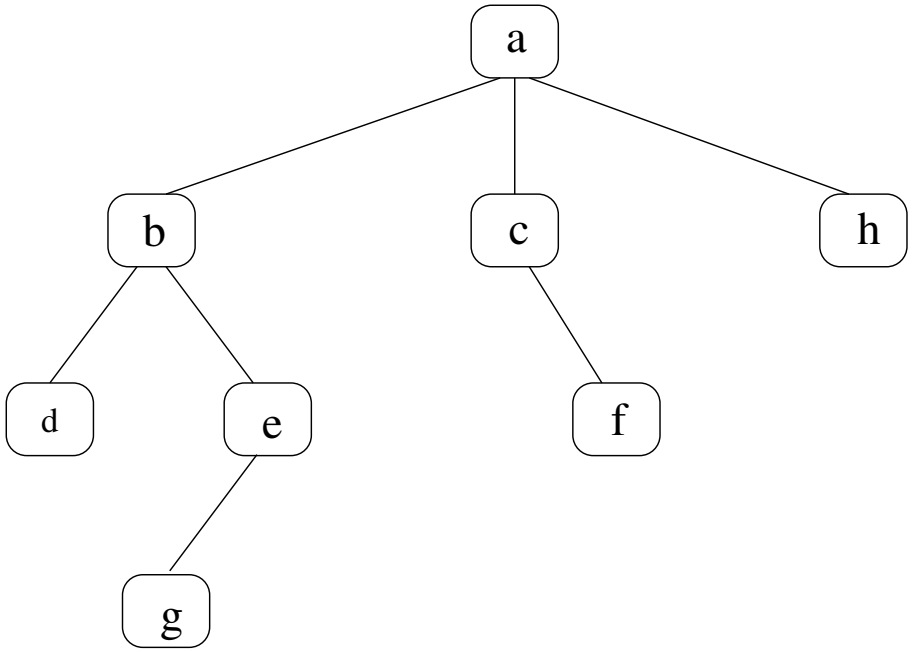
- Jos ei muuta mainita, niin tarkastelemme jatkossa järjestettyjä (ordered) puita, eli alipuiden järjestys vasemmalta oikealle on merkityksellinen

- Rekursiivinen määritelmä ei luultavasti ole helpoin tapa ymmärtää, mistä puissa on kysymys, mutta se on syytä pitää mielessä, sillä sama rekursiivinen rakenne toistuu esim. monissa puita käsittelevissä algoritmeissa

Määritellään muutamia puihin liittyviä käsitteitä

- Alipuiden $T_1, \ldots, T_k$ juuret ovat r:n lapsia (engl. child) ja r on lastensa vanhempi (engl. parent, tässä r ja T_i viittaavat edellisen sivun määritelmään ja kuvaan)
- jokaisella solmulla paitsi juurella on tasan yksi vanhempi
- solmu jolla ei ole lapsia on lehti (engl. leaf)
- solmut joilla on yhteinen vanhempi, ovat sisaruksia (engl. sibling)
- isovanhempi ja lapsenlapsi määräytyvät luonnollisella tavalla

- Esim:



  - puun juuri a, jolla lapset b, c ja h
  - puun lehtiä ovat solmut d, g, f ja h
  - d, e ja f ovat a:n lapsenlapsia, ja a on solmujen d, e ja f isovanhempi
  - d ja e ovat sisaruksia
  - esim. solmusta b alkava d:n, e:n ja g:n sisältävä puu on koko puun alipuu ja b on alipuun juuri

- Lisää määritelmiä:
  - polku (engl. path) solmusta x_1 solmuun x_k on jono solmuja $x_1, x_2, \ldots, x_k$ siten että x_i on x_{i+1}:n vanhempi kun $1 \leq i \leq k-1$

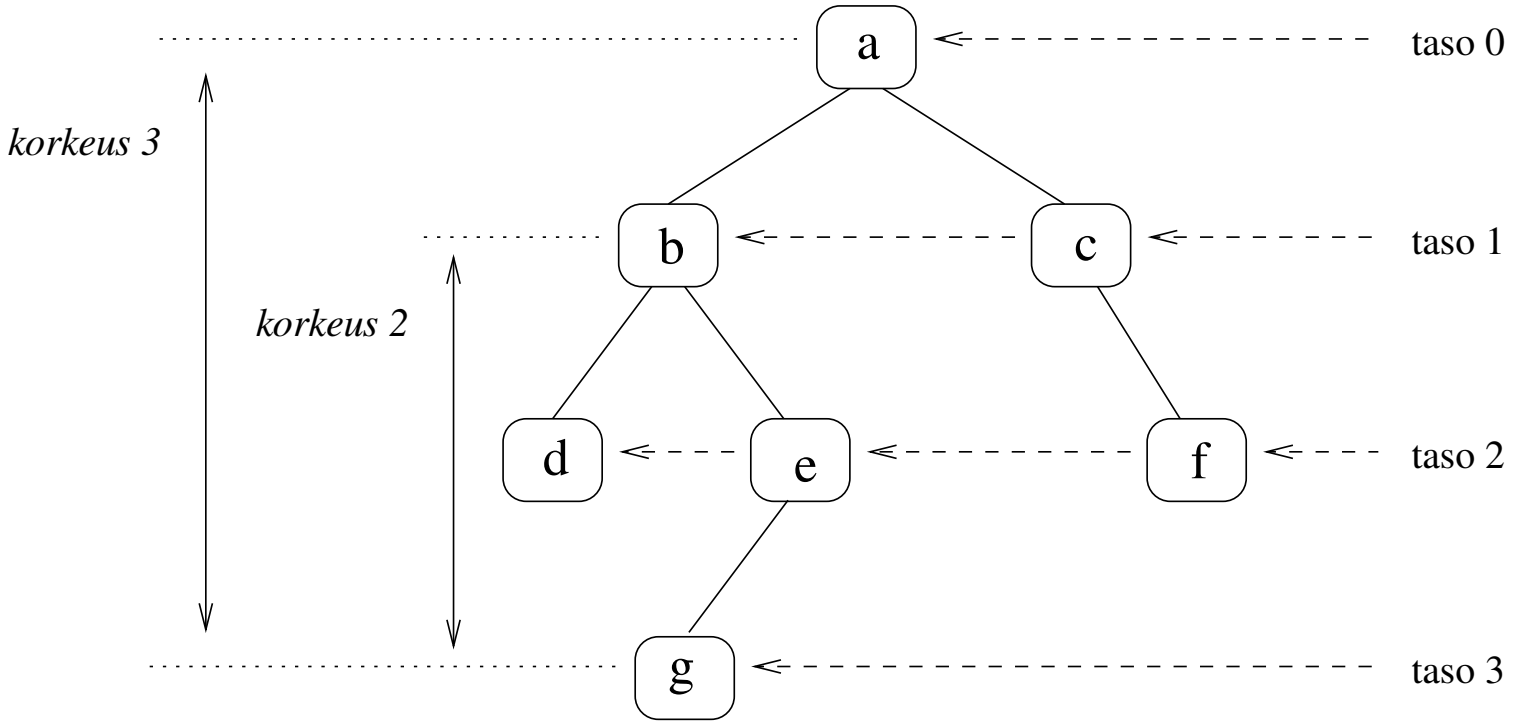
    polun pituus on sen kaarien lukumäärä, esim. edellisen sivun kuvassa a, b, e, g on polku solmusta a solmuun g jonka pituus on 3

    Erikoistapaus: jokaisesta solmusta on nollan pituinen polku itseensä
  - jos on olemassa polku solmusta x_1 solmuun x_2, sanotaan että x_1 on solmun x_2, esi-isä tai edeltäjä (engl. ancestor) ja x_2 on x_1:n jälkeläinen tai seuraaja (engl. descendant)

    Jos $x_1 \neq x_2$, edeltäjyys ja jälkeläisyys ovat aitoja (engl. proper ancestor, proper descendant)

    esim. a, b ja e ovat solmun e edeltäjät, joista a ja b ovat aitoja edeltäjiä, solmun e seuraajia ovat e ja g, joista jälkimmäinen on aito seuraaja
  - solmun x taso (engl. depth) on polun pituus juuresta x:ään, juuren taso on 0. Joissain lähteissä tasosta käytetään termiä syvyys
  - solmun x korkeus (engl. height) on pisimmän x:stä lehteen vievän polun pituus, eli syvimmän lehden syvyys
  - puun korkeus on sen juuren korkeus
- Havainto: Koska juurta lukuunottamatta jokaisella solmulla on tasan yksi vanhempi, kaarten lkm = solmujen lkm − 1

- Esim: Kuvassa solmujen tasot sekä solmun a ja b korkeus



- Puun korkeus on siis sama kuin sen juuren korkeus eli 3

Binääripuu

- Jos puun solmuilla on korkeintaan kaksi lasta, on kyseessä binääripuu (engl. binary tree)
  - binääripuun alipuiden juuria kutsutaan vasemmaksi ja oikeaksi lapseksi, solmun x vasempaan lapseen viitataan $x.left$ ja oikeaan $x.right$
  - solmusta $x.left$ alkavaa puuta kutsutaan solmun x vasemmaksi alipuuksi ja solmusta $x.right$ alkavaa x:n oikeaksi alipuuksi
  - edellisellä sivulla oleva puu on binääripuu, esim. solmun b vasen lapsi on d ja oikea lapsi e, eli $b.left = d$ ja $b.right = e$
- Todistetaan seuraavaksi kaksi tärkeää binääripuita koskevaa tulosta
- **Lause 6.1**: Olkoon T binääripuu jonka korkeus on h. Tällöin

  (1) puun tasolla i on enintään 2^i solmua

  (2) puussa on enintään $2^{h+1} - 1$ solmua

Väitteen 1 todistus:

On siis näytettävä, että binääripuun tasolla $i \geq 0$ on enintään 2^i solmua.

Tehdään todistus induktiolla tason numeron suhteen.

Tasolla 0 on vain juurisolmu, ja $2^0 = 1$ joten kaava voimassa tasolla 1.

Tehdään induktio-oletus, että väite pätee tasolla n, ja osoitetaan, että se pätee myös tasolla $n + 1$.

Induktio-oletuksen mukaan tasolla n siis enintään 2^n solmua. Koska kyseessä on binääripuu, on jokaisella näistä enintään kaksi lasta, siis tason $n + 1$ lapsimäärä on enintään $2 \cdot 2^n = 2^{n+1}$. Väite siis pätee myös tasolla $n + 1$.

Väitteen 2 todistus:

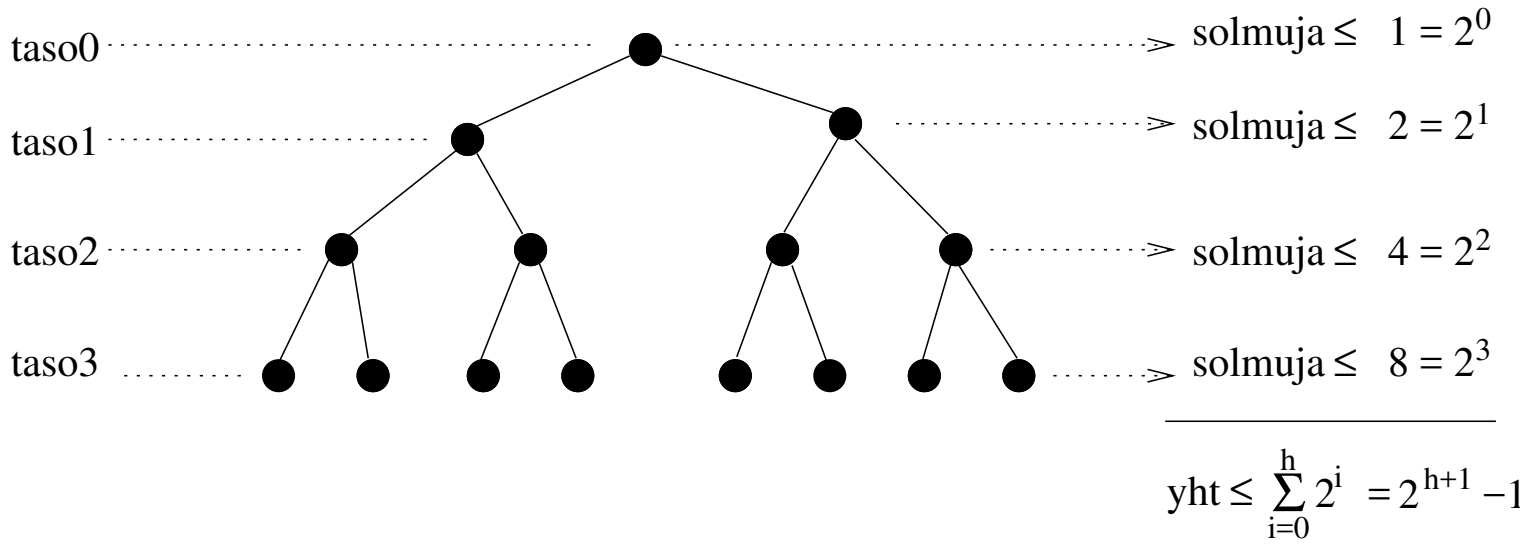
On siis näytettävä, että h:n korkuisessa puussa on enintään $2^{h+1} - 1$ solmua.

Binääripuun solmujen lukumäärä on summa eri tason solmujen lukumärästä. Edellisen kohdan perusteella tasolla i korkeintaan 2^i solmua. Eli h:n korkuisen puun maksimisolmumäärä on

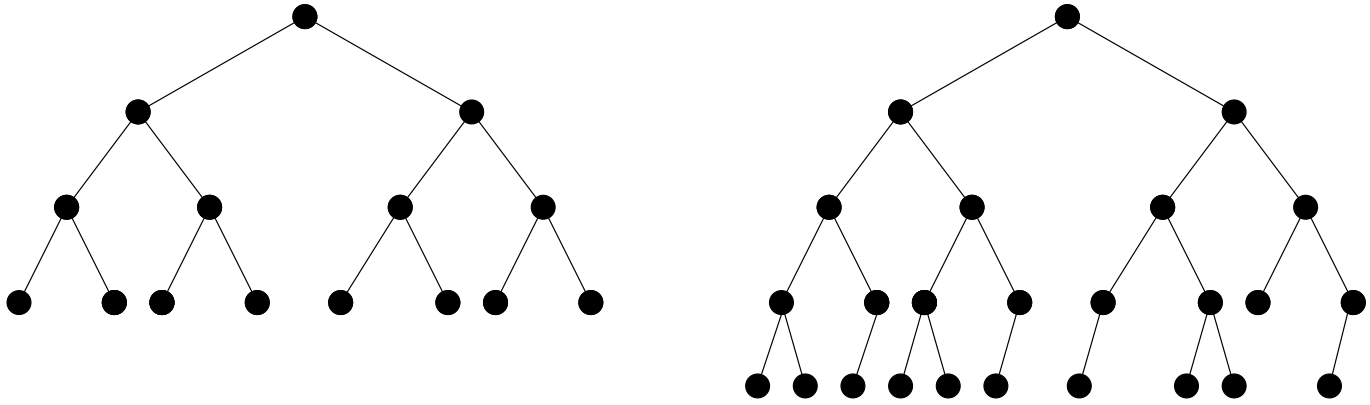
$$2^0 + 2^1 + 2^3 + \ldots + 2^h = \sum_{i=0}^{h} 2^i.$$

Tunnettu laskukaava (harjoitus 1) kertoo, että $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$. □

- Edellisen sivun todistuksien ideaa valottava kuva

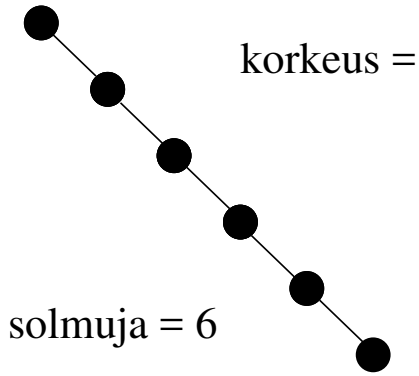


- Binääripuu on täysi (engl. full), jos jokaisella solmulla on joko 0 tai 2 lasta
- Binääripuu on täydellinen (engl. complete), jos se on täysi ja kaikki lehdet ovat samalla tasolla
- Kuvassa vasemmalla täydellinen binääripuu, eli puu, joka on on "täysin mahdollinen" tietyn korkuinen puu
- Jos puu on viimeistä tasoa lukuunnottamatta täydellisen binääripuun kaltainen, voidaan puuta nimittää melkein täydelliseksi
- Kuvassa oikealla melkein täydellinen binääripuu, joka ei ole täysi



- Lausetta 6.1 soveltamalla on helppo huomata, että h:n korkuisella täydellisellä binääripuulla on täsmälleen 2^h lehteä ja $2^{h+1} - 1$ solmua

- Toisaalta binääripuussa, jonka korkeus on, h voi olla vähimmillään $h + 1$ solmua:



- Eli jos puun korkeus on h, niin:

$$h + 1 \leq \text{solmumäärä} \leq 2^{h+1} - 1$$

- Todistetaan seuraavassa vielä täsmällisesti, mikä on puun korkeus suhteessa solmujen lukumäärään

- **Lause 6.2**:

  Olkoon T binääripuu jonka solmujen lukumäärä on n. Tällöin

  (1) puun korkeus on enintään $n - 1$

  (2) puun korkeus on vähintään $\log_2(n + 1) - 1$

  Väitteen 1 todistus:

  jos jokaisella ei-lapsisolmulla on ainoastaan yksi lapsi, on puu kuin lista ja korkeus silloin suurin mahdollinen eli $n - 1$

  Väitteen 2 todistus:

  binääripuun korkeus on pienin mahdollinen jos puu on täydellinen.
  Merkitään puun korkeutta h:lla. Lauseen 6.1 nojalla täydellisen puun solmumäärä n on $2^{h+1} - 1$, eli $n + 1 = 2^{h+1}$.
  Otetaan kaksikantainen logaritmi molemmilta puolilta:
  $\log_2(n + 1) = \log_2 2^{h+1} = (h + 1) \log_2 2 = h + 1$
  Eli $h = \log_2 (n + 1) - 1$. □

- Eli jos puussa on n solmua, niin

$$\log_2 (n + 1) - 1 \leq \text{puun korkeus} \leq n - 1$$

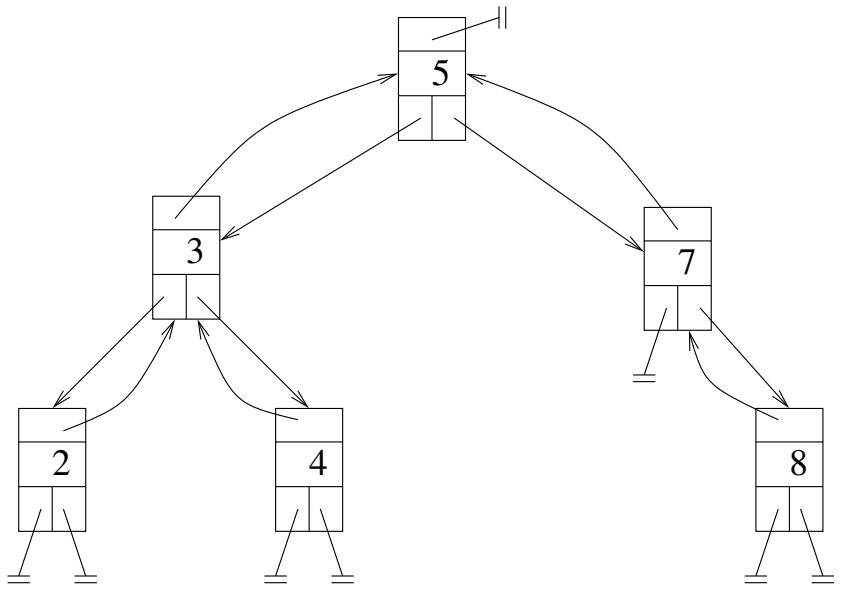
Binäärihakupuu

- Toteutetaan sivun 165 abstrakti tietotyyppi joukko siten, että joukossa olevat alkiot talletetaan binääripuun solmuihin
- Rajoitumme jälleen yksinkertaistettuun tapaukseen, jossa talletettavat tietoalkiot sisältävät ainoastaan avaimen
- Puu rakentuu puusolmu-olioista, joilla on seuraavat kentät:

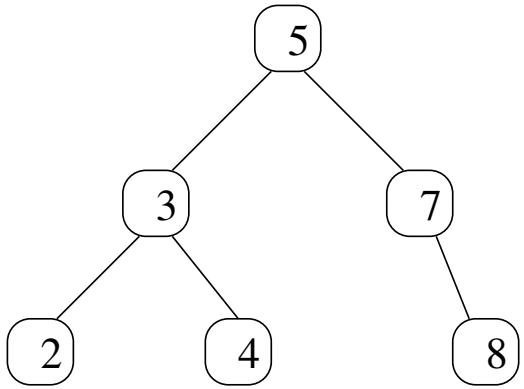
key	talletettu tietoalkio
$left$	viite vasempaan lapseen
$right$	viite oikeaan lapseen
$parent$	viite vanhempaan

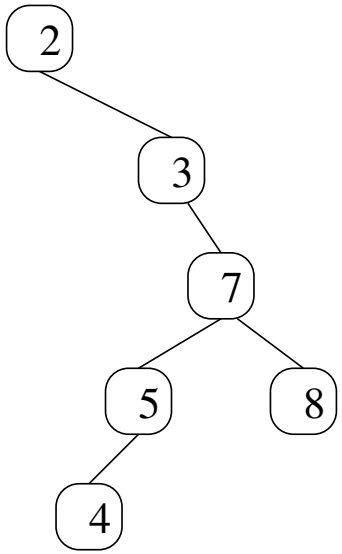
- Puulla T on attribuutti $T.root$ joka osoittaa juurisolmuun
- Jos solmulla x ei ole vasenta lasta, $x.left =$ NIL, vastaavasti oikealle lapselle
- Puun juurisolmulla $parent$-kentän arvo NIL
- $parent$-kenttä ei ole välttämätön, se kuitenkin helpottaa muutaman operaation toteutusta, joten pidämme sen mukana

- Binäärihakupuussa (engl. binary search tree) avaimet toteuttavat binäärihakupuuehdon:
  - jos solmu v on solmun x vasemmassa alipuussa, niin $v.key < x.key$
  - jos solmu o on solmun x oikeassa alipuussa, niin $x.key < o.key$
- Eli solmun vasemmassa alipuussa on ainoastaan sitä pienempiä ja oikeassa alipuussa sitä isompia avaimia
- Muistutus: vasemmalla alipuulla tarkoitetaan solmun vasemmasta lapsesta alkavaa puun osaa, oikea alipuu taas on oikeasta lapsesta alkava osa
- Esim: eräs tapa tallettaa alkiot 2, 3, 4, 5 ,7 ja 8 binäärihakupuuhun



- Yleensä piirrämme puut ilman linkkikenttien eksplisiittistä sisältöä
- Samansisältöiset, mutta mutta muodoltaan erilaiset binäärihakupuut:





- Kun monisteen tässä luvussa puhutaan puusta, tarkoitetaan jatkossa yleensä binäärihakupuuta
- Ryhdymme kohta tarkastelemaan, miten tällaisen rakenteen avulla voidaan toteuttaa tietotyypin *joukko* operaatiot

Puun alkioiden läpikäynti

- Usein on tarvetta operaatiolle, joka käy läpi kaikki puun alkiot
- Alkioiden läpikäynnin voi toteuttaa kulkemalla puun $left$-, $right$- ja $parent$-linkkejä pitkin
- Huomattavasti helpompi tapa on kuitenkin toteuttaa läpikäynti rekursion avulla
- Puun T alkiot voidaan tulostaa suuruusjärjestyksessä kutsumalla seuraavaksi esitettävää rekursiivista algoritmia parametrilla $T.root$
- **tulosta-alkiot**(x)

```
if x ≠ NIL
    tulosta-alkiot(x.left)
    print x.key
    tulosta-alkiot(x.right)
```

- Algoritmin toiminnan eteneminen syötteenä sivun 273 kuvan vasen puu

```
tulosta-alkiot(5)
    tulosta-alkiot(3)
        tulosta-alkiot(2)
            tulosta-alkiot(NIL)
            print(2)                    2
            tulosta-alkiot(NIL)
        print(3)                        3
        tulosta-alkiot(4)
            tulosta-alkiot(NIL)
            print(4)                    4
            tulosta-alkiot(NIL)
    print(5)                            5
    tulosta-alkiot(7)
        tulosta-alkiot(NIL)
        print(7)                        7
        tulosta-alkiot(8)
            tulosta-alkiot(NIL)
            print(8)                    8
            tulosta-alkiot(NIL)
```

edellä esim. kutsu tulosta-alkiot(5) tarkoittaa että operaatiota kutsutaan viitteenään solmu joka sisältää avaimenaan numeron 5

- Algoritmi käy puun läpi sisäjärjestyksessä (engl. inorder)
  - tultaessa solmuun x ensin käsitellään vasen lapsi $x.left$ sitten itse solmu x ja tämän jälkeen oikea lapsi $x.right$
  - vasenta lasta $x.left$ käsitellessä tulostetaan kaikki siitä alkavan alipuun alkiot ja binäärihakupuuehdon perusteella näiden arvo on korkeintaan sama kuin $x.key$:n arvo
  - oikeaa lasta $x.right$ käsitellessä tulostetaan kaikki siitä alkavan alipuun alkiot ja binäärihakupuuehdon perusteella näiden arvo on vähintään yhtä suuri kuin $x.key$:n arvo
  - arvo $x.key$ tulostetaan siis oikeassa kohdassa suuruusjärjestyksen suhteen
  - samanlaisen päättelyn perusteella jokainen alkio tulostetaan oikeassa kohdassa

- Jos puussa on n solmua, algoritmin aikavaativuus on $\mathcal{O}(n)$ sillä metodin runko-osa on selvästi vakioaikainen ja rekursiota kutsutaan jokaiselle solmulle sekä lehtisolmujen olemattomille lapsille eli yhteensä korkeintaan $2n + 1$ kertaa (lehtisolmuja puussa, jossa on n sisäsolmua, voi olla korkeintaan $n + 1$ kpl)

- Algoritmin tilavaativuus
  - 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 algoritmin tilavaativuus on $\mathcal{O}(h)$
  - Lauseen 6.2 perusteella puun korkeus pahimmassa tapauksessa $n-1$, missä n solmujen lukumäärä, eli
  - tilavaativuus on siis pahimmassa tapauksessa $\mathcal{O}(n)$

- Sisäjärjestyksen lisäksi kaksi muuta tärkeää binääripuun läpikäyntistrategiaa ovat esijärjestys (preorder) ja jälkijärjestys (postorder), jotka molemmat on helppo toteuttaa sisäjärjestyksen tapaan rekursiivisina operaationa
  - esijärjestyksessä solmu x käsitellään ensin ja sen jälkeen rekursiivisesti alipuut $x.left$ ja $x.right$
  - jälkijärjestyksessä käsitellään ensin rekursiivisesti alipuut $x.left$ ja $x.right$ ja lopuksi solmu x

- Eli algoritmi on esi- ja jälkijärjestyksessä sama kuin sisäjärjestyksen tapauksessa lukuunottamatta itsensä alkion x käsittelykohtaa

- Puusta voi selvittää monenlaisia mielenkiintoisia asioita sopivilla läpikäynneillä
- Jos halutaan kuljettaa tietoa lehdistä juureen päin, on jälkijärjestys hyvä strategia, eli käsittele lapset ensin, ja vasta sitten solmu itse
- Esim. puun korkeuden selvittäminen
  - solmun korkeus on sen lasten korkeuksien maksimi plus 1
  - lehtisolmujen korkeus on 0
  - algoritmi yksinkertaistuu hieman jos ajatellaan, että olemattomien alipuiden korkeus on -1

```
laske-korkeus(x)
    if x == NIL
        return -1                    // olemattoman alipuun korkeus on -1
    k1 = laske-korkeus(x.left)       // selvitetään vasemman lapsen korkeus
    k2 = laske-korkeus(x.right)      // selvitetään oikean lapsen korkeus
    return max( k1, k2 )+1           // oma korkeus on lasten korkeuden maksimi +1
```

- Eli ensin selvitetään lapsien korkeudet ja palautetaan sitten kysyjälle oma korkeus
- Operaatiolle annetaan parametriksi puun juuri: $korkeus =$ **laske-korkeus**$(T.root)$

- Jos halutaan kuljettaa tietoa juuresta lehtiin päin, on esijärjestys hyvä strategia, eli käsittele solmu ensin, ja vasta sen jälkeen lapset

- Esim. halutaan muuttaa solmujen avaimeksi arvo, joka on vanha arvo + niiden solmujen vanhojen avaimien arvot, jotka sijaitsevat polulla juuresta solmuun

  – kuljetetaan summaa mukana puussa ylhäältä alaspäin

  – huom: puu tuskin säilyy hakupuuna tällaisen operaation seurauksena

  muuta-arvoja(x, summa)
  if x $\neq$ NIL
  x.key = x.key+summa
  muuta-arvoja(x.left, x.key)
  muuta-arvoja(x.right, x.key)

  – kutsutaan operaatiota parametrina juuri ja 0: **muuta-arvoja**($T.root$, 0)

  – juuren avain säilyy entisellään, juuren lapsien avaimien arvoksi tulee juuren avaimen arvo + alkuperäinen avain, jne.

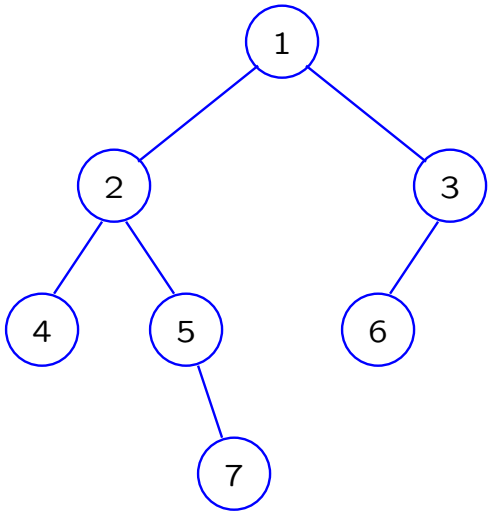
- Läpikäynti esijärjestyksessä vastaa verkossa syvyyssuuntaista läpikäyntiä

- Joissain sovellustilanteissa puun läpikäyntijärjestyksellä ei ole väliä

- Läpikäynnit on mahdollista toteuttaa myös ilman rekursiota vakiotilassa olettaen, että kahden osoittimen yhtäsuuruusvertailu on mahdollista

- Yhteenvetona toistetaan vielä eri puiden läpikäynnit: sisäjärjestys, esijärjestys, ja jälkijärjestys

```
Inorder-Tree-Walk(x)
    if x ≠ NIL
        Inorder-Tree-Walk(x.left)
        käsittele(x)
        Inorder-Tree-Walk(x.right)

Preorder-Tree-Walk(x)
    if x ≠ NIL
        käsittele(x)
        Preorder-Tree-Walk(x.left)
        Preorder-Tree-Walk(x.right)

Postorder-Tree-Walk(x)
    if x ≠ NIL
        Postorder-Tree-Walk(x.left)
        Postorder-Tree-Walk(x.right)
        käsittele(x)
```



Binääripuun solmut

sisäjärjestyksessä: 4, 2, 5, 7, 1, 6, 3
esijärjestyksessä: 1, 2, 4, 5, 7, 3, 6
jälkijärjestyksessä: 4, 7, 5, 2, 6, 3, 1

Joukko-operaatioiden toteuttaminen

- Avaimen k etsiminen puusta tapahtuu kutsumalla rekursiivista operaatiota **search**$(T.\,root, k)$

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

  – etsintä alkaa juuresta, eli aluksi $x = T.root$

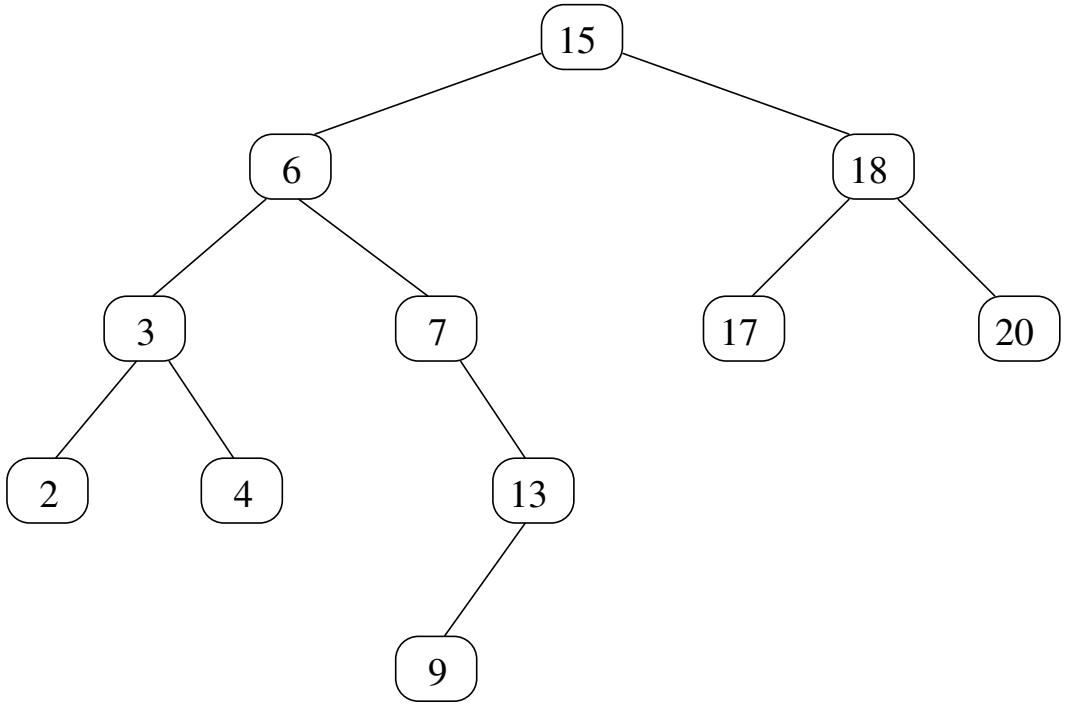
  – jos $x.key$ on suurempi kuin etsittävä k, niin binäärihakupuuehdon perusteella k ei voi olla kuin x:n vasemmassa alipuussa. Etsintä siis jatkuu rekursiivisesti vasempaan alipuuhun

  – vastaavasti, jos etsittävä k on suurempi kuin $x.key$, jatkuu etsintä rekursiivisesti oikeaan alipuuhun

  – jos etsittävä solmu on puussa, päättyy rekursion eteneminen etsittävään solmuun, jonka osoite palautetaan

  – jos etsittävää ei puussa ole, päädytään kohtaan, jossa etsittävä olisi: kohdasta löytyy ainoastaan "olematon alipuu", jota toteutuksessamme edustaa viite NIL

- Esim:



- **search**$(T.\,root, 13)$ etenee polkua $15 \to 6 \to 7 \to 13$

- **search**$(T.\,root, 10)$ etenee $15 \to 6 \to 7 \to 13 \to 9 \to$ NIL

  etsintä siis päätyy solmun 9 oikeaan olemattomaan alipuuhun, jossa avaimen 10 täytyisi olla jos se ylipäätään olisi puussa

- Etsintä siis etenee juuresta alaspäin, pahimmassa tapauksessa puun korkeuden verran, eli aikavaativuus $\mathcal{O}(h)$ missä h puun korkeus, sillä jokaisen solmun kohdalla tehdään vakiomäärä työtä

  Lauseen 6.2 perusteella puun korkeus pahimmassa tapauksessa $n-1$, missä n solmujen lukumäärä, eli **search**-operaation pahin tapaus vie aikaa $\mathcal{O}(n)$

- Rekursiopinon korkeus pahimmillaan sama kuin puun korkeus, eli myös tilavaativuus $\mathcal{O}(n)$

- Avaimen etsiminen on helppo toteuttaa myös ilman rekursiota

```
search(x,k)
    while x ≠ NIL and x.key ≠ k
        if k < x.key
            x = x.left          // siirretään x viittaamaan vasempaan lapseen
        else x = x.right        // siirretään x viittaamaan oikeaan lapseen
    return x
```

- Aikavaativuus on edelleen $\mathcal{O}(n)$ mutta koska rekursiopinosta on päästy eroon, tilavaativuus onkin $\mathcal{O}(1)$

- Binäärihakupuuehdosta seuraa suoraan, että kulkemalla mahdollisimman paljon vasemmalle päädymme pienimpään puussa olevaan alkioon

  seuraavassa operaatio, joka palauttaa viitteen solmun x alipuun pienimpään alkioon

  min(x)
  ```
  while x.left ≠ NIL
      x = x.left
  return x
  ```

- Vastaavasti, maksimialkio löytyy menemällä oikealle niin kauan kuin mahdollista:

  max(x)
  ```
  while x.right ≠ NIL
      x = x.right
  return x
  ```

- Kuten ei-rekursiivisella **search**:illa, sekä **min**- että **max**-operaatioiden pahimman tapauksen aikavaativuus puun korkeuden h suhteen on $\mathcal{O}(h)$ ja tilavaativuus $\mathcal{O}(1)$

- Sivun 283 esimerkissä minimin haku etenee polkua $15 \to 6 \to 3 \to 2$ ja maksimin polkua $15 \to 18 \to 20$

- Annettua alkiota seuraavaksi suurimman etsintä, eli operaation **succ** toteuttaminen on hiukan hankalampaa

- Sivun 283 kuvassa esim. solmun 15 seuraaja on 17, joka on sen oikean alipuun pienin alkio. Sama sääntö, eli seuraaja on oikean alipuun pienin alkio, näyttää pätevän myös esim. solmulle 6

- Esim. solmulla 13 ei ole oikeaa alipuuta, sen seuraaja onkin 15, joka löytyy kulkemalla puussa ylöspäin

  samoin on esim. solmun 4 suhteen, sen seuraaja 6 löytyy kulkemalla ylöspäin

- Oletetaan, että tarkastelun alla on solmun x seuraaja
  - binäärihakupuuehdosta seuraa, että x:n oikeassa alipuussa $x.right$ olevat alkiot ovat x:ää suurempia
  - toisaalta x:ää suurempi on myös sellainen esi-isä y, joka löytyy kulkemalla x:stä kohti juurta ja johon saavutaan kun otetaan ensimmäinen askel ylös oikealle
  - näin löytyvä alkio y on alipuun $x.right$ alkioiden jälkeen x:ää seuraavaksi suurempi alkio
  - eli x:n seuraaja on alipuun $x.right$ pienin alkio jos alipuu ei ole tyhjä. Jos taas alipuu on tyhjä, on seuraaja kuvatulla tavalla löytyvä y

- Perustellaan vielä hieman täsmällisemmin miksi seuraaja löytyy kuvatulla tavalla. Seuraavien kahden sivun kuvat liittyvät perusteluun

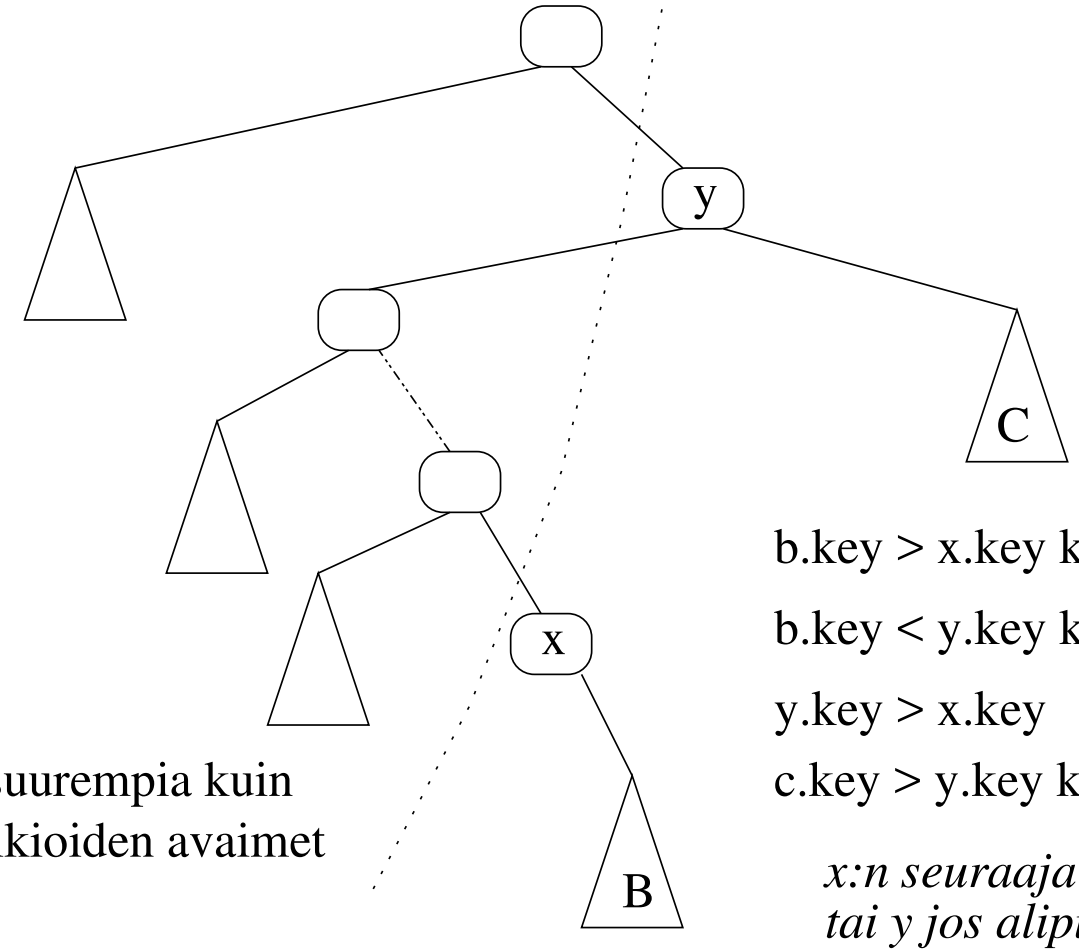
- Jos tämä perustelu (joka on matemaattisen oikeellisuustodistuksen "kansanomaistettu" versio) tuntuu sekavalta, hyppää sen yli suosiolla

- Olkoon x solmu, jonka seuraajaa etsimme, ja y sen esi-isä, joka löytyy kulkemalla kohti juurta siihen asti kunnes on kuljettu yksi askel oikealle

- Binäärihakupuuehdon perusteella
  - x:n alipuussa B olevat solmut ovat x:ää suurempia
  - y ja sen oikeassa alipuussa C olevat solmut ovat x:ää suurempia
  - y:n vanhemmassa ja sen toisessa alipuussa olevat solmut ovat joko solmua x pienempiä (kuva sivulla 288) tai y:tä suurempia (kuva sivulla 289), ne eivät siis voi tulla kyseeseen x:n seuraajina
  - y:n vasemmassa alipuussa olevat solmut jotka jäävät x:n "vasemmalle puolelle" ovat x:ää pienempiä, eli eivät voi tulla kyseeseen x:n seuraajina
  - B:ssä olevat solmut ovat y:tä pienempiä, eli jos B on epätyhjä, on sen pienin alkio x:n seuraaja
  - jos B on tyhjä, täytyy x:n seuraajan olla y

- Edellisen sivun päättelyä tukeva kuva, tapaus jossa y on vanhempansa oikea lapsi

tilanne, jossa y on vanhempansa oikea lapsi



x.key ja y.key suurempia kuin taman alueen alkioiden avaimet

b.key > x.key kaikilla b alipuussa B

b.key < y.key kaikilla b alipuussa B

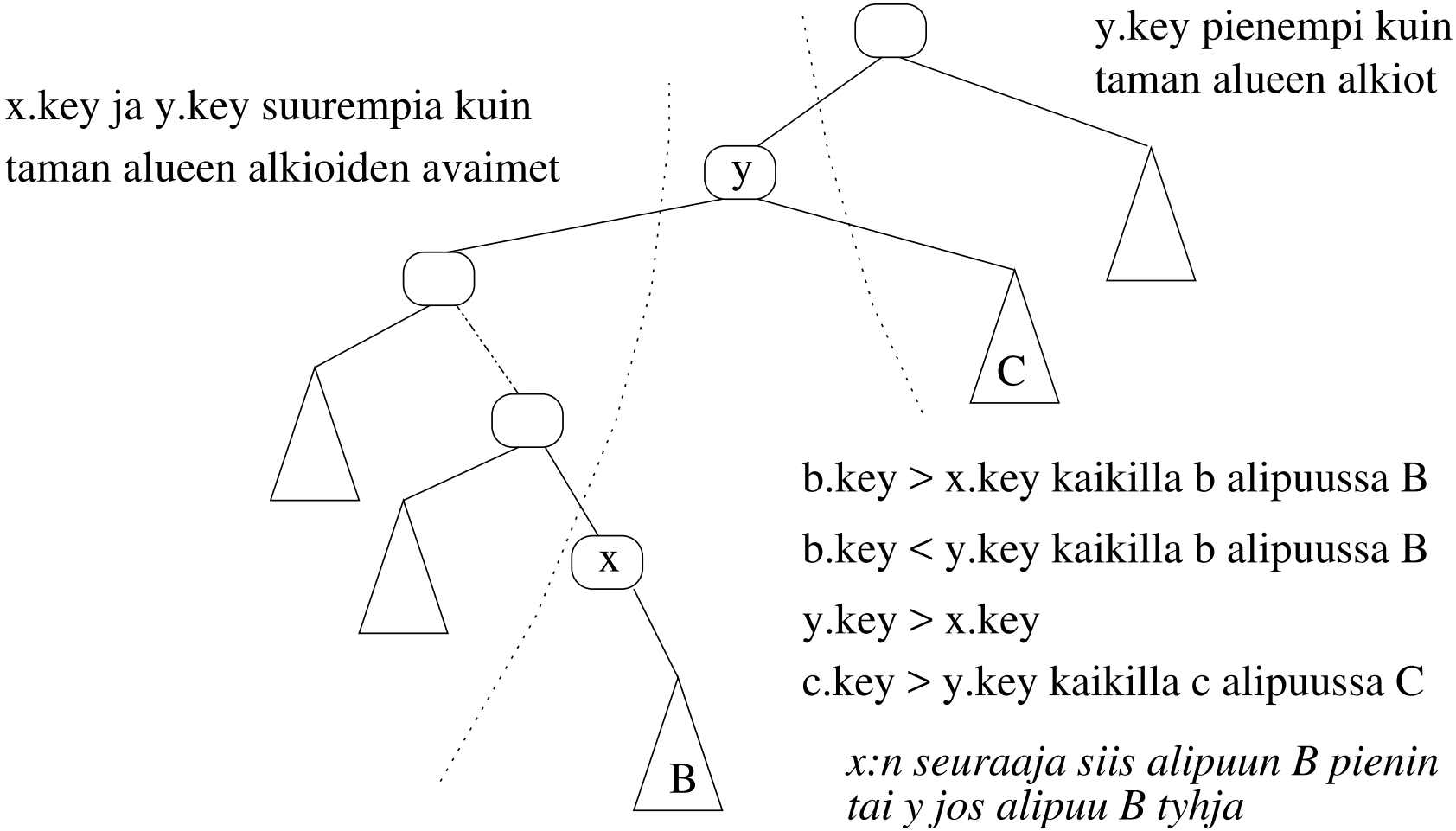
y.key > x.key

c.key > y.key kaikilla c alipuussa C

x:n seuraaja siis alipuun B pienin tai y jos alipuu B tyhja

- Toinen tapaus, eli y on vanhempansa vasen lapsi:

tilanne jossa y on vanhempansa vasen lapsi



- Algoritmi seuraavassa:

```
succ(x)
1   if x.right ≠ NIL
2       return min(x.right)
3   y = x.parent
4   while y ≠ NIL and x == y.right
5       x = y
6       y = x.parent
7   return y
```

- Kuten todettiin, algoritmin toiminta jakautuu kahteen eri tapaukseen
  - jos solmun x oikea alipuu on epätyhjä, on solmun seuraaja oikean alipuun pienin alkio
  - jos oikea alipuu on tyhjä, solmun x seuraaja on siis esi-isä y joka löytyy siten että palataan puussa kohti juurta niin kauan kunnes tehdään yksi paluuaskel "yläviistoon oikealle"
- **succ**-operaation pahin tapaus vie aikaa $\mathcal{O}(h)$ puun korkeuden h suhteen, sillä voidaan joutua menemään juuresta aina koko puun korkeuden verran alas tai palata lehdestä aina juureen asti, tilavaativuus on selvästi $\mathcal{O}(1)$
- **pred**-operaatio on symmetrinen **succ**-operaation kanssa, eli vaatii pahimmassa tapauksessa ajan $\mathcal{O}(h)$ ja tilan $\mathcal{O}(1)$

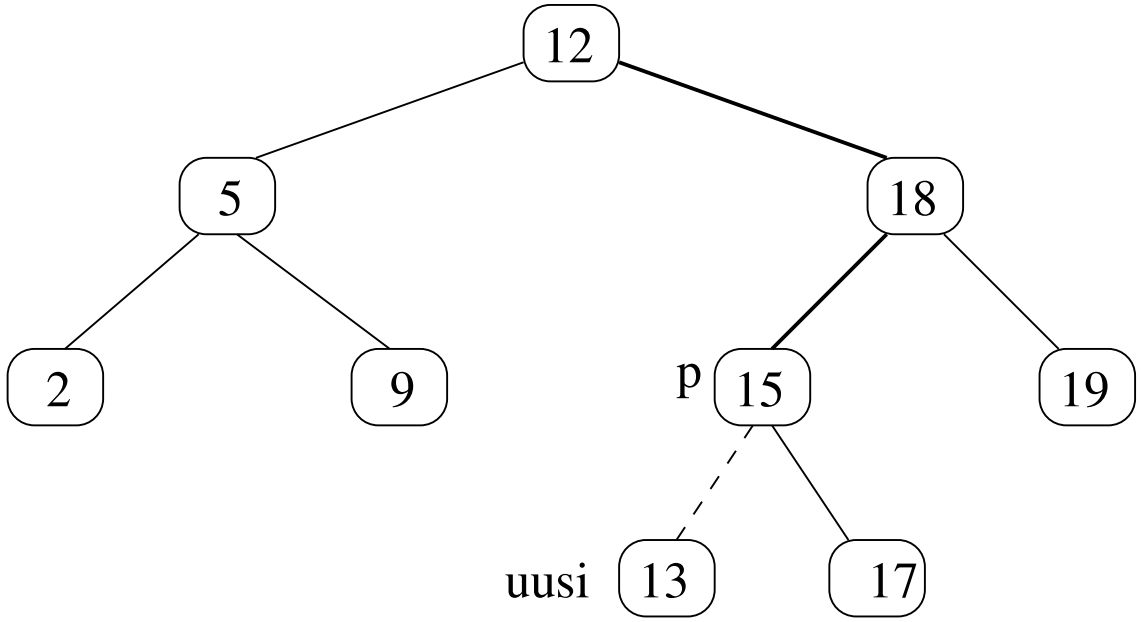
- Alkion lisääminen binäärihakupuuhun käy melko helposti

```
insert(T,k)
1   uusi = new puusolmu
2   uusi.key = k
3   uusi.left = uusi.right = uusi.parent = NIL
4   if T.root == NIL          // jos puu on tyhjä, tulee uudesta solmusta juuri
5       T.root = uusi
6       return
7   x = T.root
8   while x ≠ NIL             // etsitään kohta, johon uusi alkio kuuluu
9       p = x
10      if uusi.k < x.key
11          x = x.left
12      else x = x.right
13  uusi.parent = p           // viitteet uuden alkion ja sen vanhemman välille
14  if uusi.key < p.key
15      p.left = uusi
16  else p.right = uusi
```

- Algoritmin toimintaperiaate
  - riveillä 4-6 käsitellään erikoistapaus, jossa lisättävä on puun ensimmäinen solmu
  - rivien 8-12 while-toistolause etsii uudelle alkiolle paikan
    - uusi alkio on lisättävä puuhun siten, että binäärihakupuuehto ei mene rikki
    - paikka löytyy kulkemalla puussa alaspäin search-operaation tapaan, eli haaraudutaan joko oikeaan tai vasempaan alipuuhun riippuen lisättävän avaimen arvosta
    - etsinnässä käytetään apuna kahta viitettä:

      x on viite solmuun, jonka kohdalla paikan etsintä on menossa ja p viite tämän vanhempaan
    - kun etsinnässä päädytään tilanteeseen, jossa $x =$ NIL, tiedetään, että p on solmu, jonka alle uusi solmu voidaan laittaa rikkomatta binäärihakupuuehtoa
  - eli toistolauseen jälkeen p viittaa alkioon jonka lapseksi uusi alkio lisätään
  - riveillä 14-16 uusi alkio laitetaan joko p:n vasemmaksi tai oikeaksi lapseksi riippuen onko uuden solmun avain pienempi vai suurempi kuin p:n avain

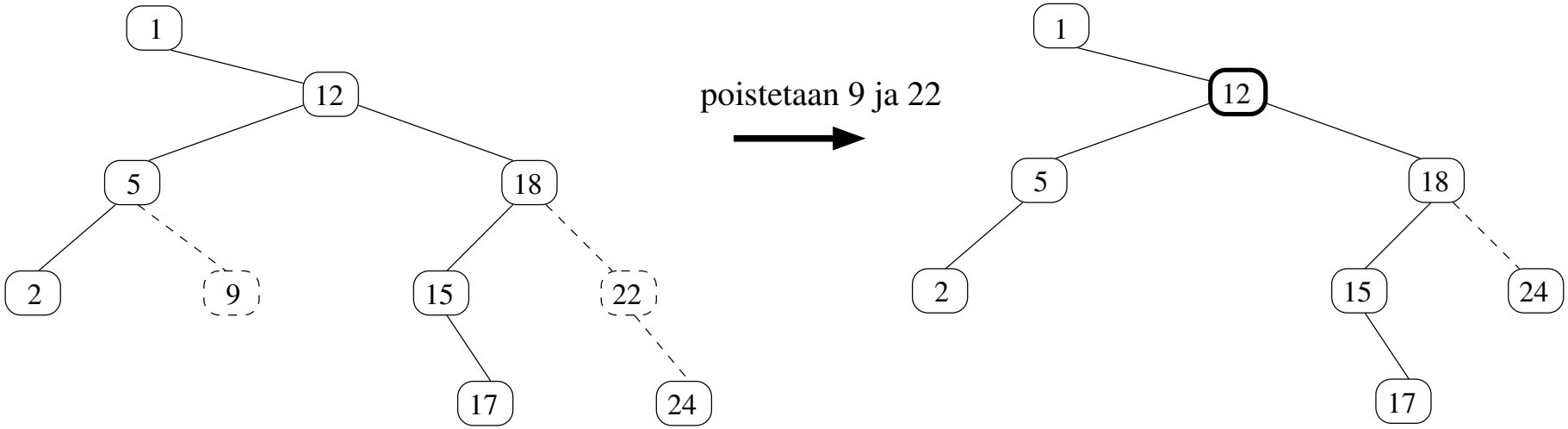
- Puu mihin lisätty avain 13, polku juuresta solmuun jonka lapseksi uusi solmu lisätään on tummennettu



- Kuten muillakin tähän asti kohtaamillamme operaatioilla, lisäyksen aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen, sillä pahimmassa tapauksessa lisäys tehdään alimmalla tasolla olevan solmun lapseksi ja tällöin operaation aikavaativuutta dominoiva while joudutaan suorittamaan h kertaa

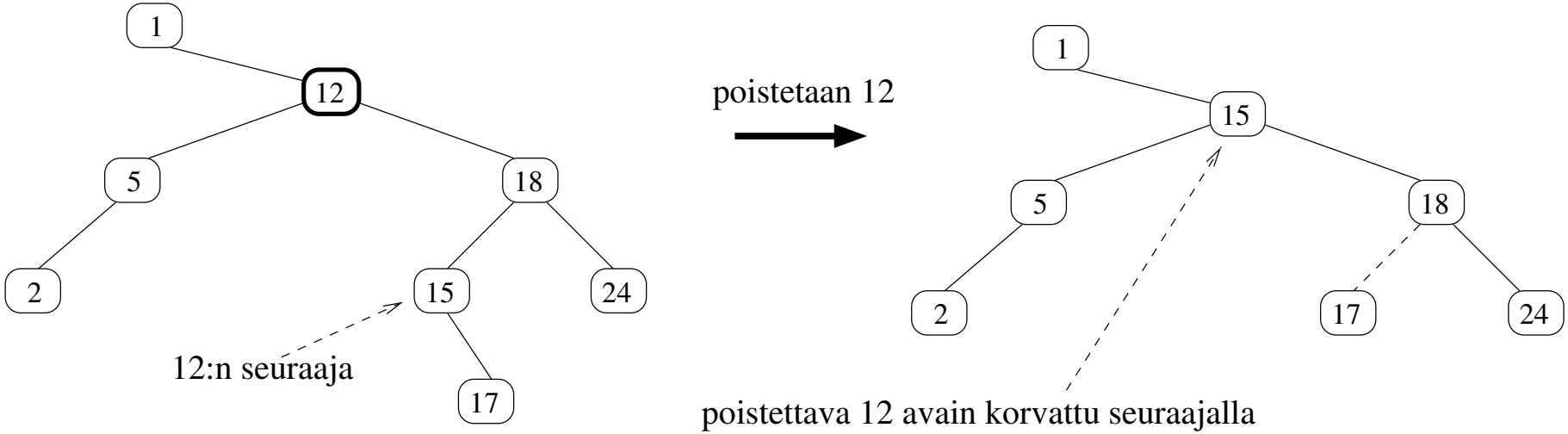
- Tilavaativuus on $\mathcal{O}(1)$, sillä rekursio ei ole käytössä ja apumuuttujia on vain muutama

- Poisto on puun operaatioista monimutkaisin. Operaatio jakautuu kolmeen tapaukseen
  - solmu, jolla ei ole lapsia on helppo poistaa: alla olevasta kuvasta esim. 9 voidaan poistaa laittamalla 5:n oikeaksi lapseksi NIL
  - solmu, jolla on tasan yksi lapsi on lähes yhtä helppo poistaa: kuvassa solmu 22 saadaan poistettua laittamalla sen lapsi 24 suoraan vanhemman 18 lapseksi, eli yksilapsinen solmu poistetaan korvaamalla se lapsella
  - kaksilapsinen solmu, esim. kuvassa 12 on ongelmallinen tapaus



- Kaksilapsinen solmu poistetaan puusta seuraavasti:
  - vaihdetaan poistettavan ja sitä seuraavaksi suurimman avaimen omaavan solmun $seur$ sisältö
  - koska poistettavalla on molemmat lapset, sen seuraajasolmu $seur$ on poistettavan oikean alipuun solmuista pienin
  - poistetaan solmu $seur$, se onnistuu helposti sillä solmulla on korkeintaan yksi lapsi (ei voi olla vasenta lasta)

- Kuvassa solmun 12 seuraaja on solmu 15. Sen sisältö viedään poistettavaan solmuun ja poistetaan vanha 15 puusta.



- Parametrina operaatiolla on viite poistettavaan solmuun $pois$

```
delete(T,pois)
1   if pois.left == NIL and pois.right == NIL
// tapaus 1: poistettavalla ei lapsia
2       vanh = pois.parent
3       if vanh == NIL                      // poistettava on puun ainoa solmu
4           T.root = NIL
5           return pois
6       if pois == vanh.left
7           vanh.left = NIL
8       else vanh.right = NIL
9       return pois
10  if pois.left == NIL or pois.right == NIL
// tapaus 2: poistettavalla on yksi lapsi
11      if pois.left ≠ NIL
12          lapsi = pois.left
13      else lapsi = pois.right
14      vanh = pois.parent
15      lapsi.parent = vanh
16      if vanh == NIL                      // poistettava on juuri
17          T.root = lapsi
18          return pois
```

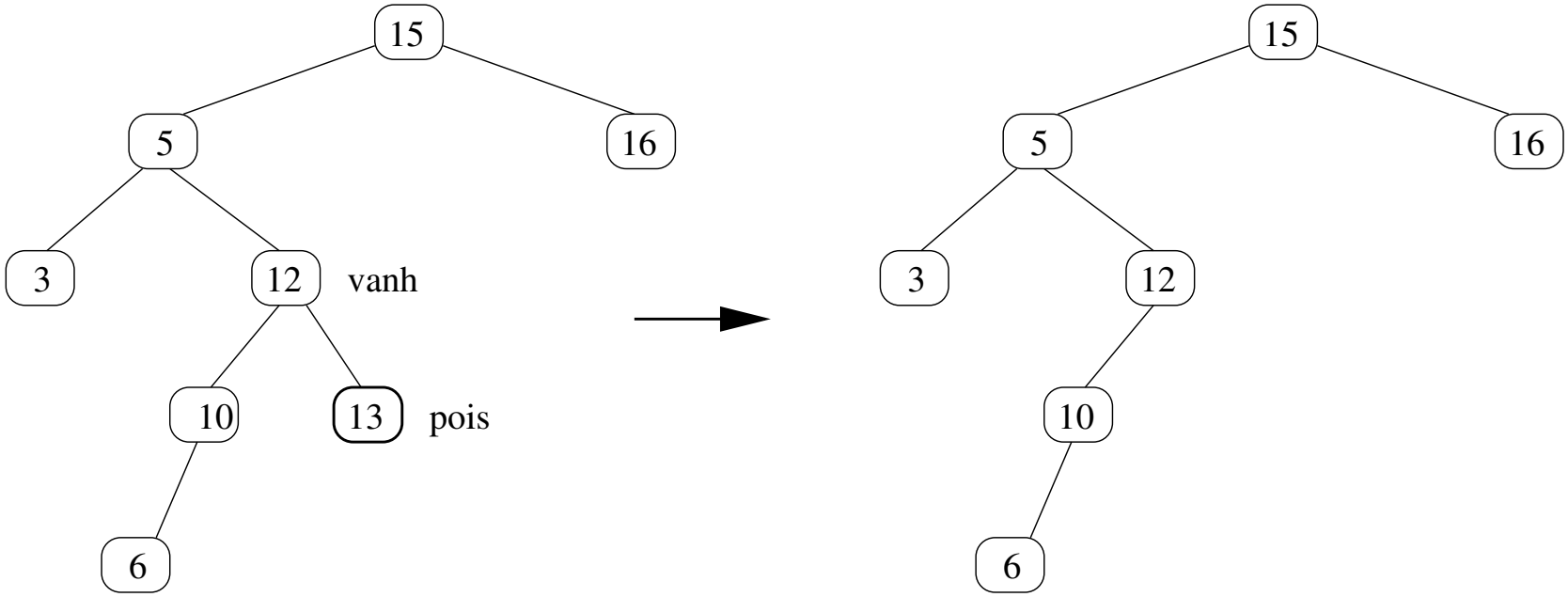
delete(T, pois) jatkuu seuraavalla sivulla...

delete(T,pois) jatkuu edelliseltä sivulla...

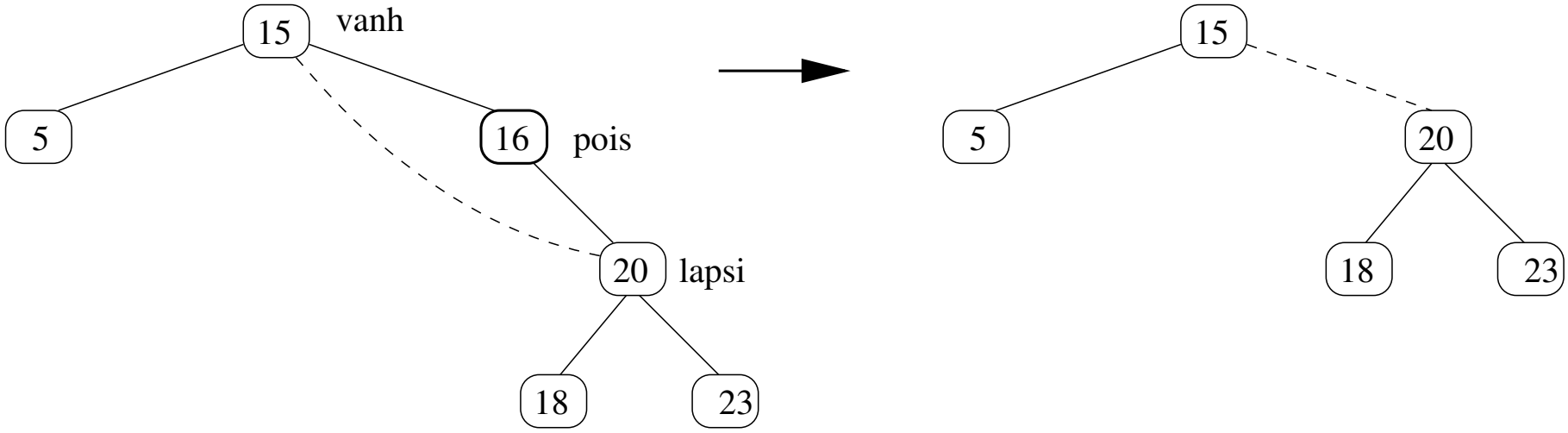
```
19      if pois == vanh.left
20          vanh.left = lapsi
21      else vanh.right = lapsi
22      return pois
// tapaus 3: poistettavalla kaksi lasta
23  seur = min(pois.right)
24  pois.key = seur.key        // korvataan poistettavan avain seuraajan avaimella
25  lapsi = seur.right
26  vanh = seur.parent         // korvataan solmu seur sen lapsella
27  if vanh.left == seur
28      vanh.left = lapsi
29  else vanh.right = lapsi
30  if lapsi ≠ NIL
31      lapsi.parent = vanh
32  return seur
```

- Operaatio palauttaa viitteen siihen solmuun joka todellisuudessa poistettiin jotta kutsuja voi tarvittaessa vapauttaa muistitilan. Kolmannessa tapauksessahan poistettava solmu ei ole sama kuin parametrina annettu
- Algoritmi näyttää huomattavasti monimutkaisemmalta mitä se itseasiassa on
- Cormenissa esitetty versio on hieman lyhempi, mutta siinä kolmea tapausta ei ole koodissa eritelty samaan tapaan kuin tässä esitetyssä

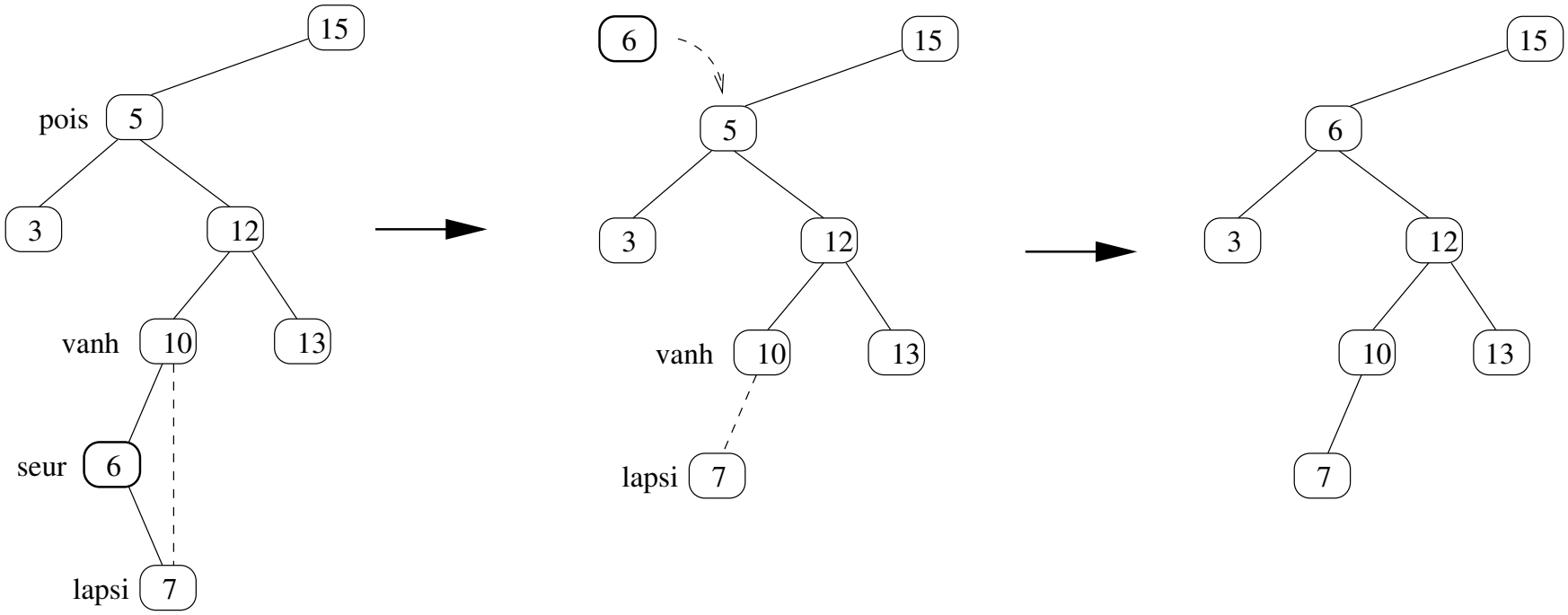
- Käydään algoritmin toiminta läpi vielä tapaus tapaukselta
- Tapaus 1: poistettavalla ei lapsia, rivit 1-9
  - poistetaan solmu $pois$
  - riveilla 3-5 huomioidaan erikoistapaus missä poistettava on puun ainoa solmu. Tässä tapauksessa puun juureksi asetetaan NIL
  - riveillä 6-8 asetetaan poistuneen solmun tilalle sen vanhempaan NIL-viite



- Tapaus 2: poistettavalla yksi lapsi, rivit 10-22
  - korvataan poistettava $pois$ ainoalla lapsellaan $lapsi$
  - riveillä 11-13 selvitetään, onko poistettavan lapsi oikea vai vasen
  - riveillä 16-18 huomioidaan tapaus, missä on poistettava on puun juuri. Tällöin poistettavan lapsesta tulee uusi juuri
  - riveillä 19-21 laitetaan $lapsi$ poistuneen tilalle



- Tapaus 3: poistettavalla kaksi lasta, rivit 23-31
  - aluksi vaihdetaan solmun $pois$ avain sen seuraajan $seur$ avaimeen
  - solmu $seur$ mistä korvaava avain löytyi poistetaan korvaamalla se lapsellaan $lapsi$, tämä tapahtuu riveillä 25-29
  - on varmaa, että solmulla $seur$ on korkeintaan oikea lapsi, joten sen poistaminen on helppoa
  - jos korvaava solmu $lapsi$ ei ole NIL, laitetaan se osoittamaan uuteen vanhempaansa riveillä 30-31



- **delete** on muuten vakioaikainen, mutta tapauksessa kolme joudutaan kutsumaan operaatiota **min**, jonka aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen

- Täten poiston pahimman tapauksen aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen. Tilavaativuus on $\mathcal{O}(1)$ sillä käytettyjen apumuuttujien määrä on vakio

- Huom: Operaatio **delete**$(T, pois)$ siis poistaa puusta solmun sisällön, tapauksessa 3 solmun pois muistialue jää vielä käyttöön sisältäen kuitenkin toisen avaimen

- Binäärihakupuun kaikkien operaatioiden (**search**, **insert**, **delete**, **max**, **min**, **succ** ja **pred**) pahimman tapauksen aikavaativuus on siis $\mathcal{O}(h)$, missä h on puun korkeus

- Kaikki operaatiot pystyttiin tekemään siten että tilavaativuus on vain $\mathcal{O}(1)$

- Lauseen 6.2 perusteella n-solmuisen puun korkeus h vaihtelee välillä $\log_2(n+1) - 1 \leq h \leq n - 1$

- Eli jos puu on tarpeeksi tasapainoinen (eli mahdollisimman paljon täydellistä binääripuuta muistuttava) on operaatioiden aikavaativuus $\mathcal{O}(\log n)$ ja puu on oleellisesti listaa parempi joukon toteutuksessa

- Kun taas hyvin epätasapainoisessa puussa operaatioiden aikavaativuus on $\mathcal{O}(n)$ ja puu on siis jopa listaa huonompi tapa abstraktin tietotyypin joukko toteuttamiseen

  - huono puu syntyy esim. lisäämällä puuhun solmut $1, \ldots, n$ suuruusjärjestyksessä tai käänteisessä järjestyksessä

  - toisaalta huono puu voi syntyä myös patologisen insert/delete-suoritusyhdistelmän takia

- Logaritmisen ja lineaarisen aikavaativuuden ero on huikea:

  - esim. $\log_2 16777216 = 24$ ja $\log_2 4294967296 = 32$, logaritmi siis kasvaa todella hitaasti

- Haasteeksi nouseekin miten voisimme pitää puun tasapainoisena siten, että operaatioiden vaativuus saataisiin pysymään logaritmisena

- Puun tasapainossa pitäminen on vielä tapahduttava siten, että toimenpide itse ei ole niin vaativa että se tekisi puusta käyttökelvottoman

- Esim. rakentamalla puu jokaisen lisäyksen ja poiston jälkeen huolellisesti alusta uudelleen todennäköisesti pitäisi puun tasapainossa, mutta tekisi lisäyksestä ja poistosta toivottoman hitaan

- Eli puu on osattava pitää tasapainoisena siten, että tasapainossa pitäminen ei muuta operaatioiden aikavaativuutta huonompaan suuntaan kuin korkeintaan vakiokertoimen verran

Tasapainoiset hakupuut

- Tavoitetta toteuttaa joukko-operaatiot ajassa $\mathcal{O}(\log n)$ voidaan lähestyä eri tavoin:

  AVL-puut: historiallisesti ensimmäinen (1962) ja toteutukseltaan yksinkertaisin

  punamustat puut: AVL-puun idean tehostettu versio, käytetään esim. Javan standardikirjastoissa.

  B-puut: tehokkaita levymuistia käytettäessä

  splay-puut: ei takaa tasapainoisuutta, mutta operaatioiden aikavaativuus silti $O(\log n)$ per operaatio (tasoitettu aikavaativuus) kun tarkastellaan koko operaatiojonoa

  skip-lista: ei puurakenne, vaan usean listan hierarkia, aikavaativuus $O(\log n)$ odotusarvoisesti

  treap: satunnaisuutta käyttävä puun ja keon (heap) yhdistelmä, "odotusarvoisesti tasapainoinen"

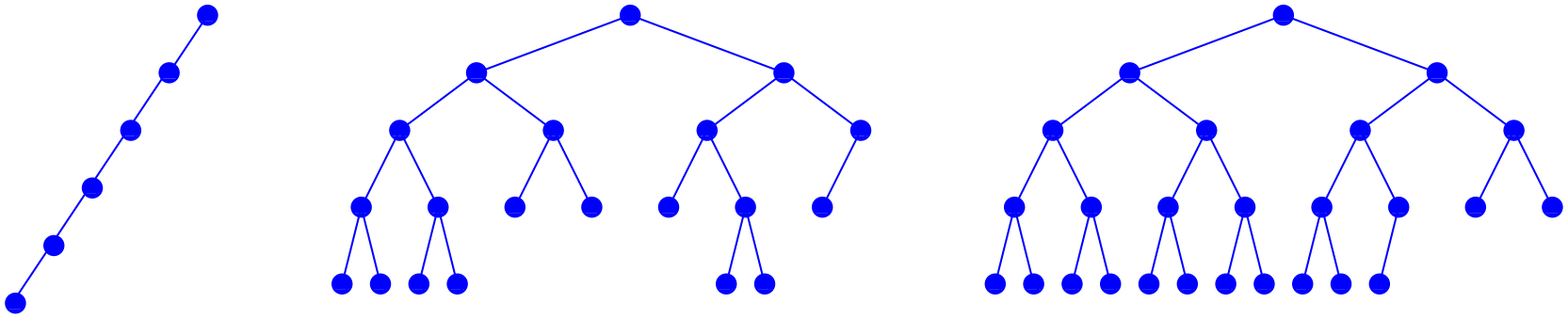
- Tutustumme tässä AVL-puihin

- Hakunopeuden kannalta paras tilanne on täydellinen puu (ks. sivu 268). Tällöin n-alkioisen puun korkeus on $\log_2(n+1)-1$. Puuta on kuitenkin vaikea pitää edes suunnilleen täydellisenä, kun siihen lisätään ja siitä poistetaan alkioita

- Pahin tilanne on lineaarinen puu, jossa kaikki oikeat tai kaikki vasemmat alipuut ovat tyhjiä, puun korkeus on tällöin $n-1$

- Intuitiivisesti puu on "tasapainoinen", kun se muistuttaa muodoltaan enemmän täydellistä kuin lineaarista puuta

- Eri tavat määritellä tasapainoisuus täsmällisesti johtavat hieman erilaisiin tietorakenteisiin (esim. AVL-puu tai punamustapuu)



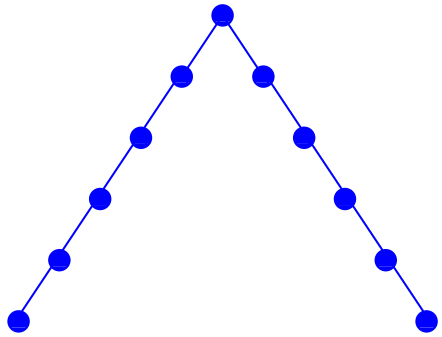
lineaarinen | tasapainoinen? | melkein täydellinen

- Teknisesti ottaen haluamme määritellä jonkin täydellisyyttä hieman heikomman tasapainoehdon, joka
  - on helpompi pitää voimassa kuin täydellisyys mutta
  - takaa kuitenkin, että puun korkeus on $\mathcal{O}(\log n)$

- Ylläpidon helpottamiseksi siis löysennämme ehtoa niin, että ei vaadita puun olevan lähes täydellinen (eli maksimaalisesti tasapainoinen), vaan riittää että puun korkeus solmujen lukumäärän n suhteen on $d \cdot \log n$ jollakin vakiolla d

- Tasapainoehdon intuitiivinen merkitys on yleensä karkeasti, että jokaisen solmun vasen ja oikea alipuu ovat jossain mielessä suunnilleen samankokoiset

- Huomaa, että pelkästään juuren tarkasteleminen ei riitä:



Juuren vasen ja oikea alipuu näyttävät samankokoisilta, mutta puun korkeus on $\lfloor n/2 \rfloor$ (eli jakolaskun n/2:n kokonaislukuosa) eli puu ei ole tasapainoinen

AVL-puut [Georgi Adelson-Velski ja Jevgeni Landis, 1962]

- AVL-puita Java-koodeineen esitetään Weissin kirjan luvussa 4.4.
  - Monisteessa esitetään AVL-puun **insert**-operaatiosta ei-rekursiivinen versio, toisin kuin Weissin kirjassa
  - Weissin kirja, kuten useimmat lähteet, jättää operaation **delete** kokonaan määrittelemättä

- Normaalin binääripuun solmussa olevien attribuuttien key, $left$, $right$ ja $parent$ lisäksi jokaisessa AVL-puun solmussa on kenttä $height$, joka ilmoittaa solmun korkeuden.

- Muistin virkistykseksi:
  - Solmun korkeus on pisimmän siitä lehteen vievän polun pituus
  - Erityisesti lehden korkeus on 0.
  - Puun (tai alipuun) korkeudella tarkoitetaan sen juuren korkeutta.

- Edellisen kanssa on yhteensopivaa, että tyhjän binääripuun NIL korkeudeksi määritellään -1

- Olettaen että solmuilla on $height$-attribuutit joille on rakennusvaiheessa asetettu oikeat arvot, voidaan solmusta x alkavan alipuun korkeus kysyä funktiolla

  Height(x)
  ```
  if x== NIL
      return -1
  else return x.height
  ```

  eli tyhjän puun (jota edustaa viite NIL) korkeus on -1 ja muuten puun korkeus selviää juurisolmun height-attribuutista

- AVL-puulta vaaditaan, että se toteuttaa seuraavan tasapainoehdon:

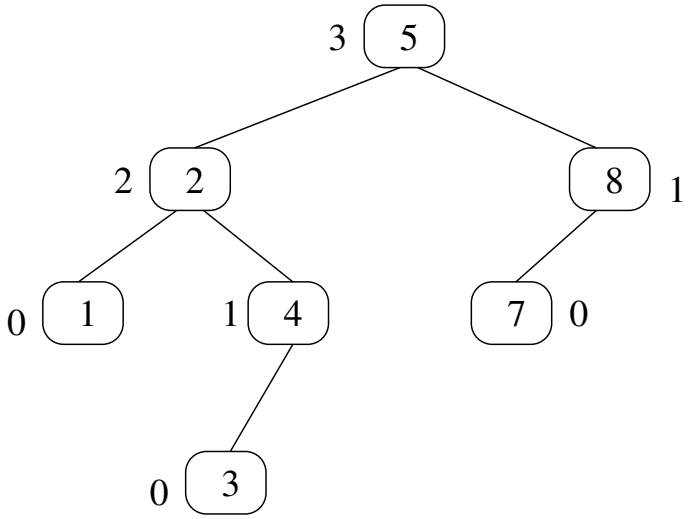
  > minkä tahansa solmun vasemman ja oikean alipuun korkeuksien erotus on joko -1, 0 tai 1.

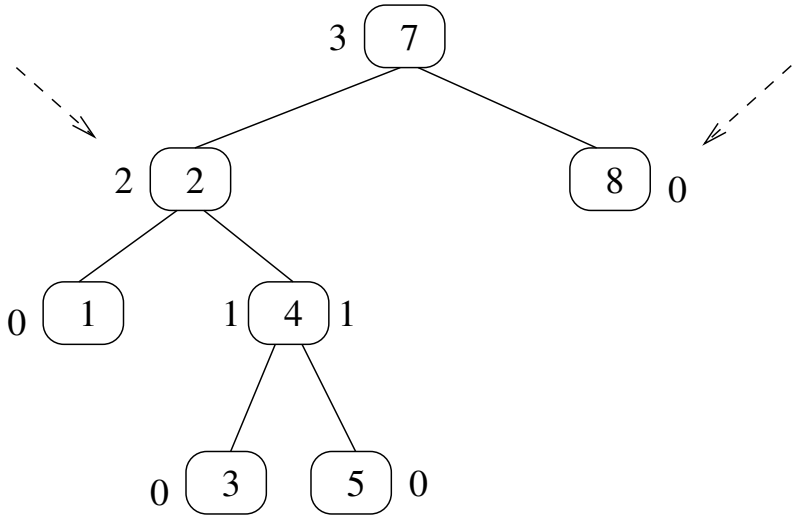
  Toisin sanoen vaaditaan

  $$|\,\textbf{Height}(x.\,left) - \textbf{Height}(x.\,right)\,| \leq 1 \qquad \text{kaikilla solmuilla } x$$

- Haluamme osoittaa, että
  - jos tasapainoehto on voimassa, niin puun korkeus on $\mathcal{O}(\log n)$
  - jos tasapainoehto rikkoutuu avaimen lisäyksen tai poiston yhteydessä, se voidaan saada taas voimaan ajassa $\mathcal{O}(\log n)$ puuta sopivasti muokkaamalla

- Allaoleviin samansisältöisiin puihin on merkitty solmujen korkeudet





- Vasemmanpuoleinen puu toteuttaa AVL-tasapainoehdon, oikeanpuoleinen ei toteuta, sillä juurisolmun 7 alipuiden korkeusero on 2

- **Lause 6.3**: Jos AVL-puun korkeus on h, niin siinä on ainakin $F_{h+3} - 1$ solmua, missä F_i on i:s Fibonaccin luku

- Ennen lauseen todistusta tarkastellaan sen seurauksia

  Fibonaccin luvut $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ määritellään palautuskaavalla

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{i+2} &= F_{i+1} + F_i \qquad \text{kun } i \geq 0. \end{aligned}$$

  Fibonaccin luvuille tunnetaan myös eksplisiittinen kaava

$$F_i = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right).$$

  Kaavan voi johtaa esim. generoivien funktioiden avulla, mutta tämä tekniikka ei kuulu kurssin alueeseen. Kaava on kuitenkin helppo todeta oikeaksi induktiolla.

Luku $(1+\sqrt{5})/2 \approx 1{,}618$ tunnetaan kultaisena suhteena. Koska toinen potenssiin korotettava luku $(1-\sqrt{5})/2 \approx -0{,}618$ on itseisarvoltaan ykköstä pienempi, termi $((1-\sqrt{5})/2)^i$ pienenee nopeasti, kun i kasvaa. Itse asiassa

$$F_i = \left[\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^i \right],$$

missä $[x]$ tarkoittaa reaalilukua x pyöristettynä lähimpään kokonaislukuun.

Lauseen mukaan pienin solmujen lukumäärä korkeutta h olevassa AVL-puussa on

$$F_{h+3} - 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \approx \frac{1{,}618^{h+3}}{2{,}236} - 1.$$

Solmujen lukumäärä kasvaa siis eksponentiaalisesti korkeuden funktiona

Olkoon puolestaan $H(n)$ suurin n-solmuisen AVL-puun korkeus. Merkitään $\alpha = (1+\sqrt{5})/2$ ja $\beta = (1-\sqrt{5})/2$. Selvästi jos $H(n) = h$, niin

$$\begin{aligned} n \geq S(h) = F_{h+3} - 1 &= \frac{1}{\sqrt{5}}(\alpha^{h+3} - \beta^{h+3}) - 1 \\ &\geq \frac{1}{\sqrt{5}}(\alpha^{h+3} - |\beta|^3) - 1. \end{aligned}$$

Tästä seuraa $\sqrt{5}(n+1+|\beta|^3/\sqrt{5}) \geq \alpha^{h+3}$. Ottamalla logarimit molemmin puolin saadaan h ratkaistua:

$$h \leq \frac{\log_2(n+1+|\beta|^3/\sqrt{5})}{\log_2 \alpha} + \frac{\log_2 \sqrt{5}}{\log_2 \alpha} - 3 \leq 1{,}4405 \log_2(n+1{,}1056) - 1{,}3277.$$

Siis $H(n) = O(\log n)$ eli puun korkeus kasvaa logaritmisesti.

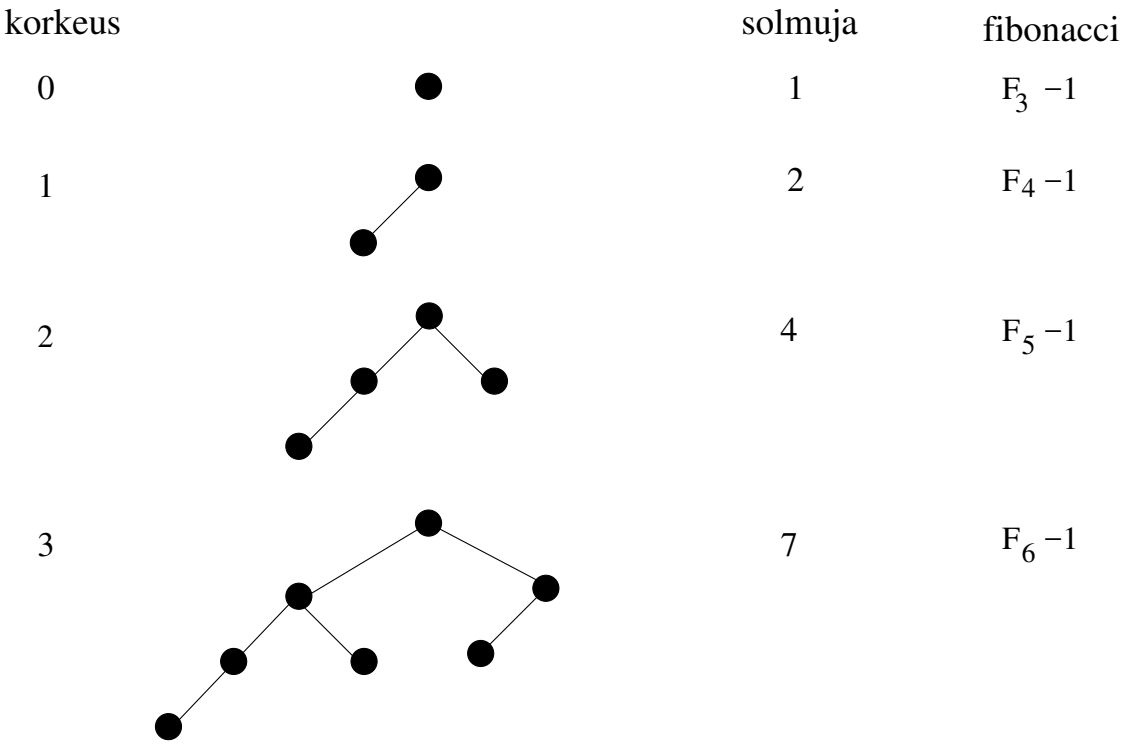
Esimerkki: Jos $n = 100$, niin yllä oleva kaava antaa tulokseksi $h \leq 8{,}266$, eli 100-solmuinen AVL-puu on korkeintaan korkeudeltaan 8 ja jos $n = 1000000$, niin $h \leq 27{,}384$, eli korkeus on korkeintaan 27

- **Lauseen 6.3 todistus:**

  On siis näytettävä, että h:n korkuisessa AVL-puussa on vähintään $F_{h+3} - 1$ solmua:

  Fibonaccin lukujonon alku siis: $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$.

  Eli esim. 3:n korkuisessa AVL-puussa pitäisi olla vähintään $F_6 - 1$ eli 7 solmua



Kuvaa tarkastelemalla huomaamme, että esim. pienin 3:n korkuinen AVL-puu voidaan muodostaa ottamalla juurisolmu ja asettamalla sen alipuiksi pienin mahdollinen 2:n korkuinen ja pienin mahdollinen 1:n korkuinen AVL-puu

Pienimmän mahdollisen 3:n korkuinen AVL-puun solmulukumäärä on siis pienimmän mahdollisen 2:n ja 1:n korkuisen puiden solmulukumäärän summa plus yksi (eli juurisolmu)

Sama pätee mille tahansa korkeudelle:

pienimmän h:n korkuisen puun solmujen lukumäärä on pienimmän $h-1$:n ja pienimmän $h-2$:n korkuisen puun solmulukumäärä + juurisolmu

Todistetaan nyt induktiolla lauseen väittämä, eli että h:n korkuisessa AVL-puussa on vähintään $F_{h+3}-1$ solmua

Perustapaus: tyhjän puun korkeus on -1 ja tosiaan $F_2-1=0$; vastaavasti jos puun korkeus on 0, puussa on yksi solmu (juuri) ja tosiaan $F_3-1=1$

Oletetaan, nyt että kaava pitää paikkansa h:n korkuiselle tai sitä matalammille puulle ja näytetään että tästä seuraa että se pätee myös $h+1$:n korkuiselle puulle, eli että $h+1$:n korkuisessa AVL-puussa on vähintään $F_{h+4}-1$ solmua.

Kuten edellä todettiin, saadaan pienin mahdollinen $h+1$:n korkuinen puu liittämällä juurisolmun lapsiksi pienin mahdollinen h:n ja $h-1$:n korkuinen puu.

Tälläisen puun solmulukumäärä on lasten alipuiden solmulukumäärä plus juuri.

Induktio-oletuksen perusteella alipuiden solmulukumäärät ovat $F_{h+3}-1$ ja $F_{h+2}-1$

Eli yhteensä pienimmän $h+1$:n korkuisen puun solmulukumäärä on
$F_{h+3}-1+F_{h+2}-1+1 = F_{h+3}+F_{h+2}-1$

Fibonaccin lukusarjalle pätee $F_{h+3}+F_{h+2} = F_{h+4}$, eli pienimmän $h+1$:n korkuisen puun solmulukumäärä on $F_{h+4}-1$, eli kaava kaava on todistettu oikeaksi. □

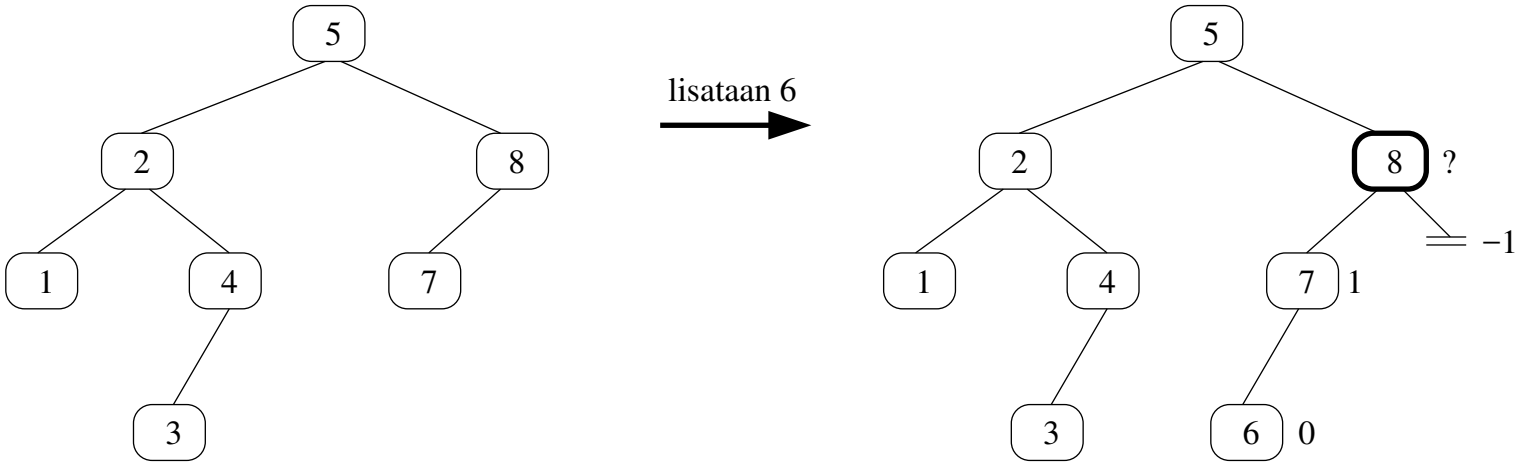
- Johtopäätös: jos binäärihakupuu saadaan toteuttamaan AVL-puun tasapainoehto, niin joukko-operaatiot sujuvat ajassa $O(\log n)$
- Ongelma: miten lisäys ja poisto suoritetaan rikkomatta tasapainoehtoa?
- Ongelman ratkaisu on soveltaa lisäyksen tai poiston jälkeen sopivia kiertoja (engl. rotation), joilla mahdollisesti rikkoutunut tasapainoehto saadaan taas voimaan
- Kierto on paikallinen, vakioaikainen operaatio, joka nostaa joitain alipuita ylemmäs ja painaa joitain alemmas
- Kierto korjaa paikallisen epätasapainon siten, että binäärihakupuuehto pysyy voimassa
- Osoittautuu, että lisäysoperaation yhteydessä tarvitaan korkeintaan kaksi kiertoa pitämään puu tasapainossa. Poiston yhteydessä kiertoja saatetaan joutua tekemään lineaarinen määrä puun korkeuteen nähden (logaritminen solmujen lukumäärään nähden)
- Kiertoja sovelletaan myös muissa hakupuurakenteissa (punamusta, splay)

- Lisäys- ja poisto-operaatioiden yhteydessä on siis pidettävä huolta, että puu säilyy AVL-puuna

- Tarkastellaan ensin lisäysoperaatiota

- Kuvan vasemmanpuoleiseen puuhun voitaisiin lisätä esim. avain 0 rikkomatta AVL-ehtoa

- Esim. avaimen 6 lisääminen taas rikkoo AVL-ehdon sillä solmu 8 menee epätasapainoon



- Epätasapainoon menevällä solmulla on vasen alipuu jonka korkeus 1, mutta oikeata alipuuta ei ole. Olemattoman alipuun korkeudeksi sovittiin -1, joten alipuiden korkeuksien erotus on 2

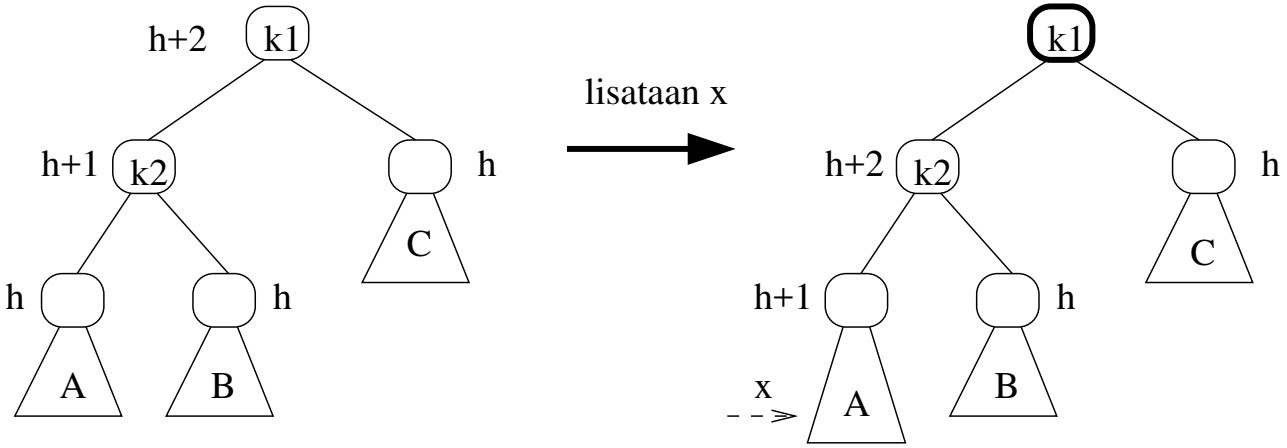
- Periaatteena AVL-puun lisäysoperaatiossa on tehdä ensin lisäys kuten normaaliin binäärihakupuuhun. Jos AVL-ehdon havaitaan menneen rikki lisäyksen yhteydessä, puu korjataan tekemällä sopivat kierto-operaatiot

- Koska lisäys vaikuttaa ainoastaan lisätyn solmun esi-isien korkeuteen, epätasapainoon lisäyksen takia mahdollisesti menevät solmut löytyvät reitiltä lisätystä solmusta puun juureen

- Tarkempi analyysi paljastaa, että lisäyksessä syntyvälle epätasapainolle voi olla 2 erilaista syytä ja näiden kanssa symmetriset 2 tapausta

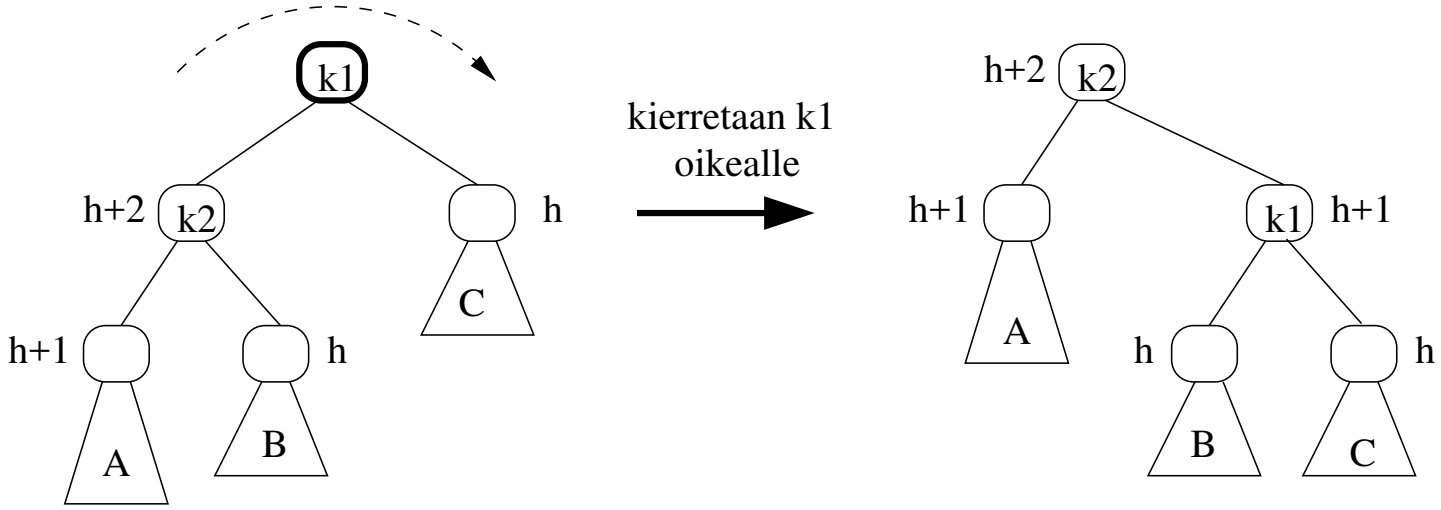
- Tarkastellaan ensin yksinkertaisempaa tapausta epätasapainoon joutumisesta ja siitä toipumisesta

- Tutkitaan tilannetta, missä AVL-puuhun lisätään solmu x siten että lisäys aiheuttaa epätasapainon
- Olkoon $k1$ syvimmällä epätasapainossa lisäyksen jälkeen oleva solmu
- Käsitellään ensin tilannetta, jossa lisätty solmu menee $k1$:n vasemman lapsen $k2$ vasempaan alipuuhun, kuvaan merkitty myös alipuiden korkeuksia



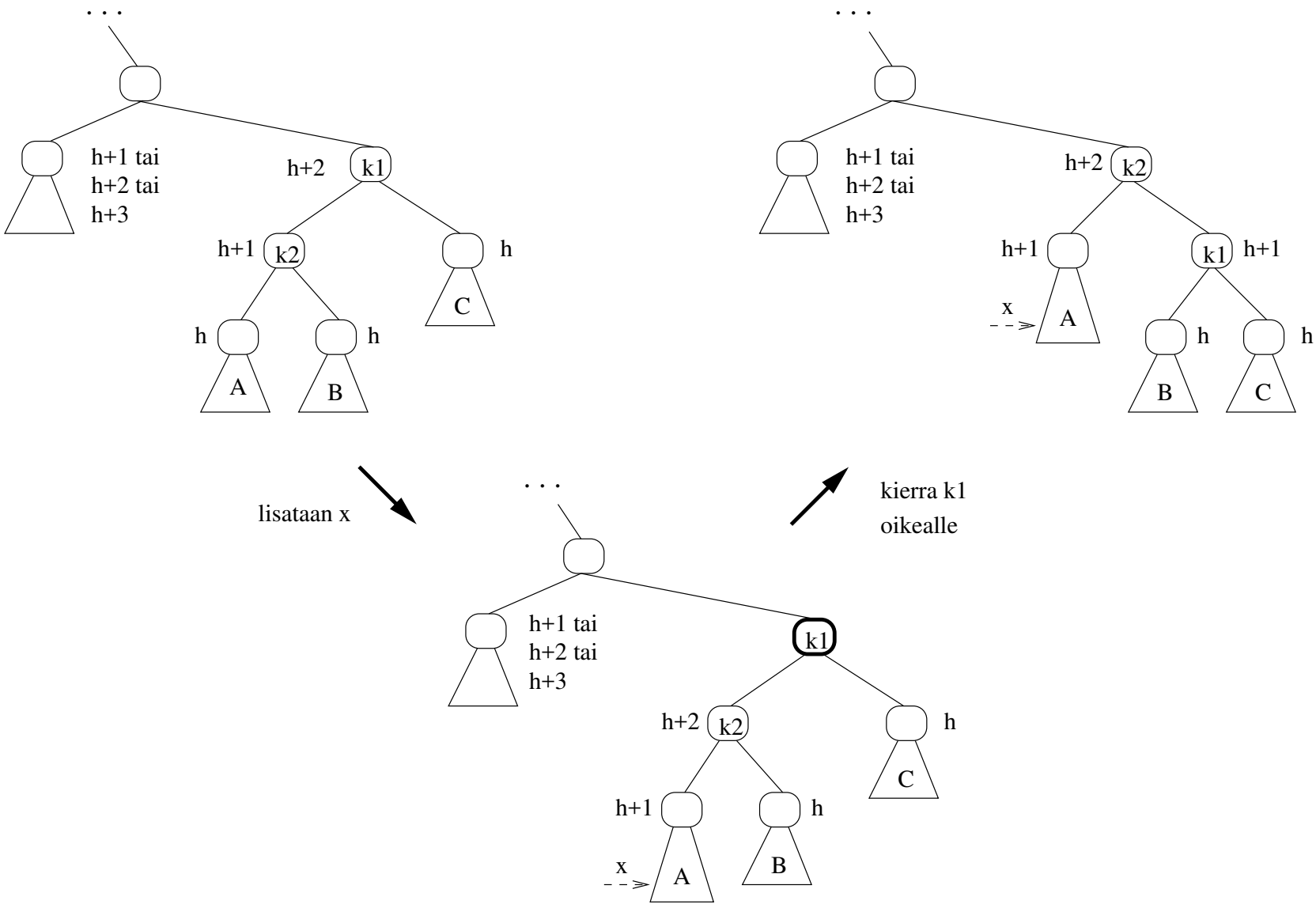
- Epätasapainoa solmuun $k1$ ei olisi syntynyt ellei puun tilanne ennen lisäystä olisi ollut täsmälleen edellisen sivun vasemmalla olevan kaltainen
  - Jos B:n korkeus olisi ollut $h-1$, niin $k2$ olisi mennyt epätasapainoon. Oletimme kuitenkin, että $k1$ on syvimmällä epätasapainossa oleva solmu
  - B:n korkeus ei voinut olla myöskään $h+1$ sillä silloin $k1$ olisi ollut epätasapainossa jo ennen lisäystä
  - Jos A:n korkeus olisi ollut $h-1$ ei epätasapainoa olisi syntynyt
  - A:n korkeus ei voinut olla myöskään $h+1$ sillä silloin $k1$ olisi ollut epätasapainossa jo ennen lisäystä
  - Alipuiden A:n ja B:n korkeuksien siis täytyi ennen lisäystä olla h
  - C:n korkeus ei olisi voinut olla $h-1$, sillä silloin $k1$ olisi ollut epätasapainossa jo ennen lisäystä
  - C:n korkeus ei olisi voinut myöskään ollut $h+1$ sillä siinä tapauksessa epätasapainoa ei olisi syntynyt
  - myös alipuun C korkeuden täytyi ennen lisäystä olla h

- Tasapaino palautetaan tekemällä kierto oikealle epätasapainossa olevan solmun $k1$ suhteen
  - $k1$:sta tulee sen vasemman lapsen $k2$ oikea lapsi
  - $k2$:n oikea alipuu B siirtyy $k1$:n vasemmaksi alipuuksi
  - kierto oikealle säilyttää puun binäärihakupuuna sillä alipuu B pysyy edelleen solmun $k1$ vasemmalla puolella ja $k2$:n oikealla puolella



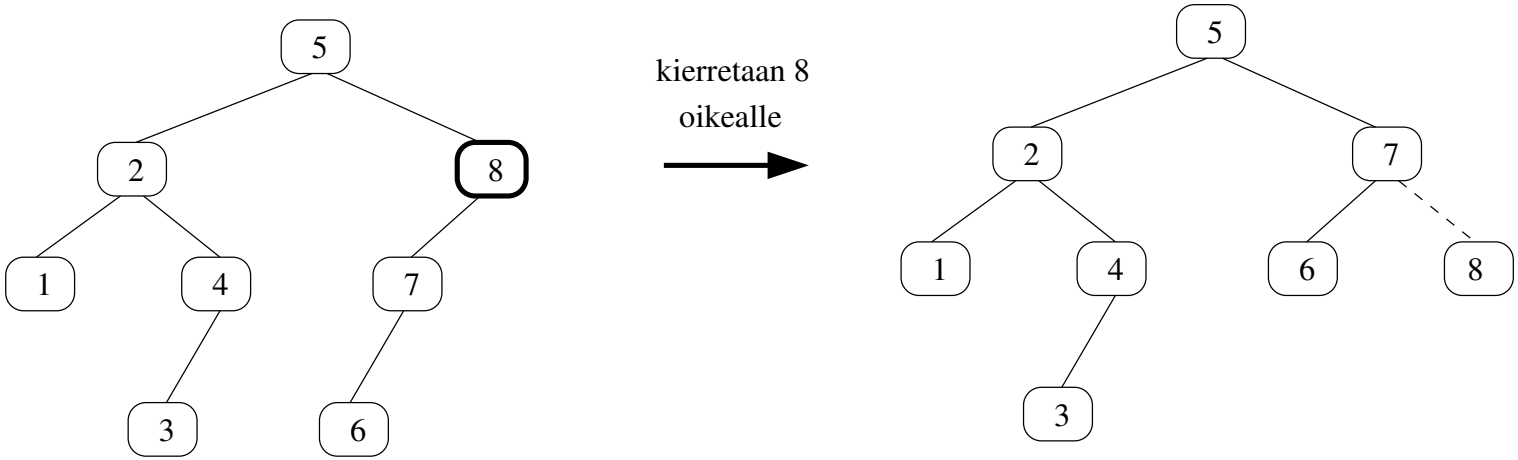
- Huomaamme, että kierto palauttaa puun takaisin tasapainoon (ks. seur. sivu)
  - koko alipuun korkeus on nyt sama kuin ennen lisäysoperaatiota
  - koska oletimme, että $k1$ on alin epätasapainoinen solmu, ei puussa kierron jälkeen enää voi olla epätasapainoisia solmuja

- Epätasapainoisen solmun kierto korjaa koko puun tasapainon:



- Kierto oikealle selvittää myös muutama sivu sitten aiheuttamamme epätasapainon

- Nyt solmu 7 nousee kierrettävän solmun 8 vanhemmaksi

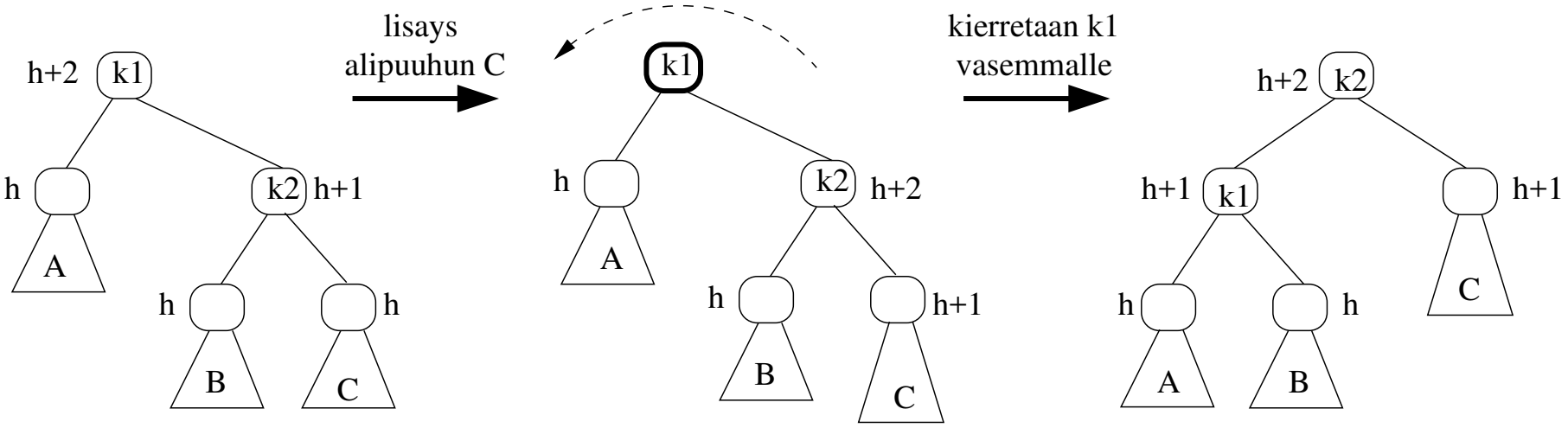


- Kierron seurauksena solmusta 7 siis tulee epätasapainoon menneen alipuun uusi juuri

- Kierron yhteydessä kierretyn alipuun uusi juuri on siis kiinnitettävä vanhempansa lapseksi

- Huomataan, että tilanne, jossa solmun $k1$ epätasapainon aiheuttaa lisäys sen oikean lapsen $k2$ oikeaan alipuuhun, on symmetrinen edellisen kanssa

- Solmun $k1$ joka siis on syvimmällä oleva epätasapainoon joutunut solmu kiertäminen vasemmalle palauttaa puun tasapainoon



- Kierto-operaatioiden toteutus on suhteellisen suoraviivainen

```
RightRotate(k1)
1   k2 = k1.left
2   k2.parent = k1.parent
3   k1.parent = k2
4   k1.left = k2.right
5   k2.right = k1
6   if k1.left ≠ NIL
7       k1.left.parent = k1
8   k1.height = max( Height(k1.left), Height(k1.right) ) +1
9   k2.height = max( Height(k2.left), Height(k2.right) ) +1
10  return k2
```

- Riveillä 2-5 $k2$:sta tulee alipuun uusi juuri, $k1$:stä sen oikea alipuu ja $k1$ saa vasemmaksi alipuuksi $k2$:n oikean alipuun

- Rivillä 6-7 asetetaan siirtyneelle alipuulle oikea vanhempi jos alipuu ei ollut tyhjä

- Riveillä 8 ja 9 päivitetään solmujen korkeuden muistavat $height$-attribuutit

- Viimeisellä rivillä palautetaan kierretyn alipuun uusi juuri, näin on tehtävä, jotta alipuu saadaan laitettua vanhempansa lapseksi

- Kierto-operaatio on selvästi vakioaikainen: puun koosta riippumatta suoritettavia koodirivejä on saman verran. Myös tilavaativuus on vakio sillä käytössä on ainoastaan 2 apumuuttujaa

- Analyysissä on tietysti huomioitava myös kierron käyttämät operaatiot **Height** ja **max** (funktio, joka palauttaa argumenteista suurimman), jotka ovat selvästi vakioaikaisia ja -tilaisia

- Kierto-operaation kutsujan vastuulle siis jää liittää kierretty alipuu vanhempansa lapseksi

- Kutsu voisi esim. tapahtua seuraavasti:

```
    ...
1   if solmu p epätasapainossa
2       vanhempi = p.parent
3       alipuu = RightRotate(p)
4       if vanhempi.left == p
5           vanhempi.left = alipuu
6       else vanhempi.right = alipuu
    ...
```

- Kierron seurauksena p siis ei ole enää alipuun juuri

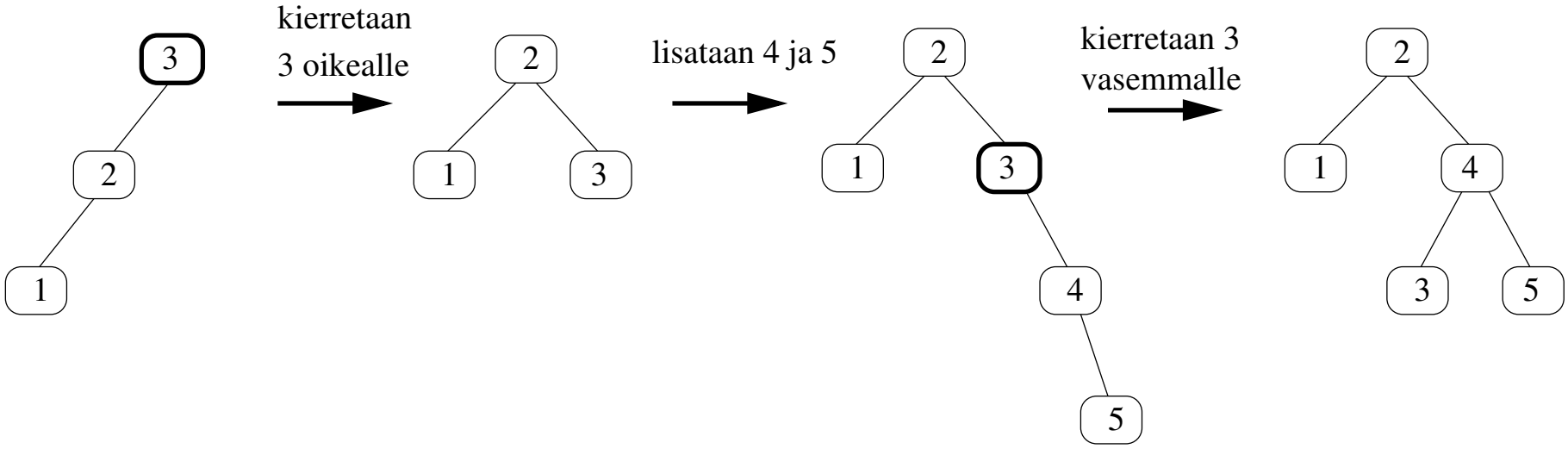
- Riveillä 4-6 asetetaan alipuun uusi juuri vanhempansa oikeaksi tai vasemmaksi lapseksi sen mukaan, oliko p vasen vai oikea lapsi

- Kierto vasemmalle on edellisen kanssa symmetrinen

```
LeftRotate(k1)
1   k2 = k1.right
2   k2.parent = k1.parent
3   k1.parent = k2
4   k1.right = k2.left
5   k2.left = k1
6   if k1.right ≠ NIL
7       k1.right.parent = k1
8   k1.height = max( Height(k1.left), Height(k1.right) ) +1
9   k2.height = max( Height(k2.left), Height(k2.right) ) +1
10  return k2
```

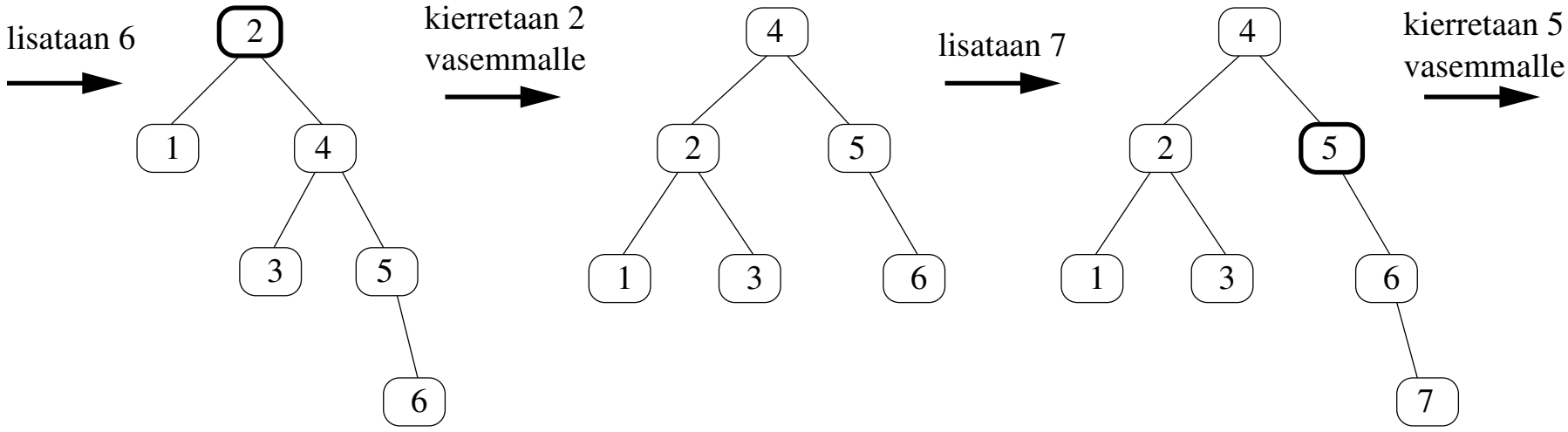
- Nyt siis tiedämme miten puu saadaan tasapainotettua, jos solmuun $k1$ epätasapainon aiheuttama lisäys tehdään
  - $k1$:n oikean lapsen $k2$ oikeaan alipuuhun
  - $k1$:n vasemman lapsen $k2$ vasempaan alipuuhun
- Ensimmäisessä tapauksessa siis kierretään solmua $k1$ vasemmalle ja toisessa tapauksessa oikealle. Yksi kierto-operaatio palauttaa puun tasapainoon
- Ennen kuin tarkastelemme muita tapoja epätasapainon syntytapoja, käydään läpi konkreettinen esimerkki

- Puuhun lisätään aluksi 3, 2 ja 1. Viimeinen lisäys aiheuttaa epätasapainon solmuun 3
- Epätasapainon aiheuttama lisäys tehtiin epätasapainossa olevan solmun 3 vasemman lapsen vasempaan alipuuhun. Edellä päätellyn perusteella ongelma ratkeaa kiertämällä epätasapainoista solmua 3 oikealle
- Lisätään puuhun vielä 4 ja 5. Jälkimmäinen lisäys aiheuttaa epätasapainon solmuun 3
- Tällä kertaa tilanteesta selvitään vasemmalle kierrolla, sillä epätasapainon aiheuttaja oikean lapsen oikeassa alipuussa

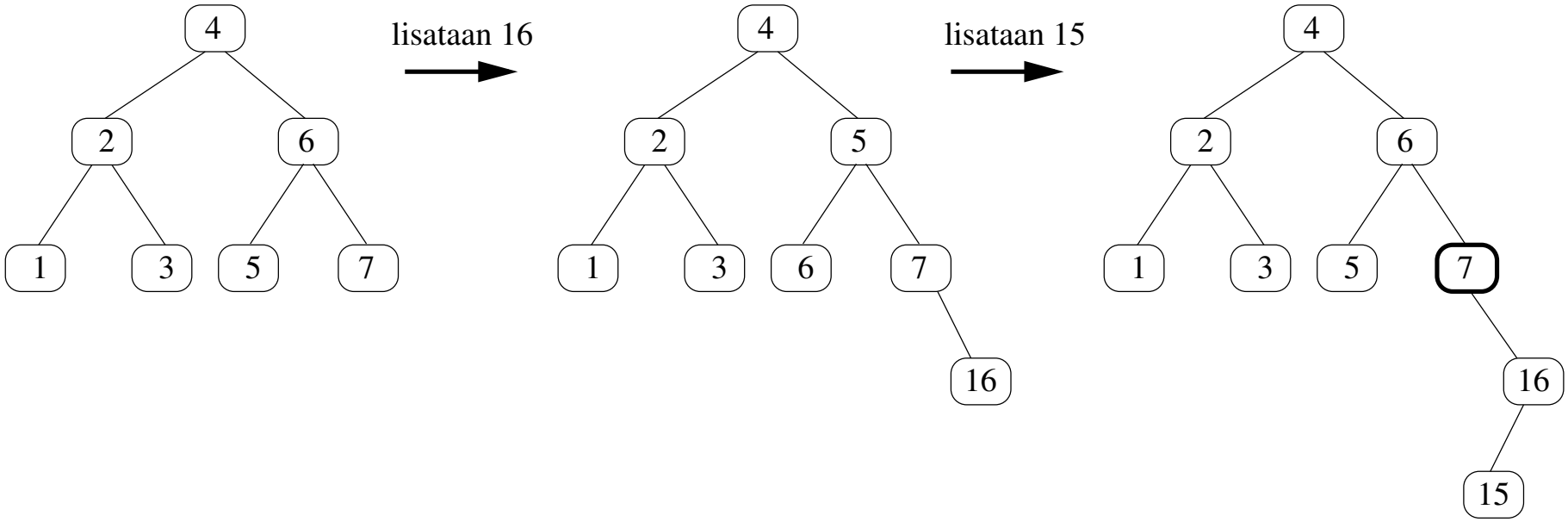


- Jatketaan lisäämällä puuhun 6. Tämä vie juuren, eli solmun 2 epätasapainoon. Jälleen epätasapainon aiheuttaja on lisäys oikean lapsen oikeaan alipuuhun ja kierto vasemmalle korjaa tilanteen.

- Puuhun lisätään vielä 7. Tämä vie solmun 5 epätasapainoon ja korjaus on kierto vasemmalle. Kierron jälkeen lopputuloksena on täydellinen, eli maksimaalisen tasapainoinen avaimet 1, . . . , 7 sisältävä puu, joka on esitetty seuraavalla sivun vasemmanpuoleisessa kuvassa

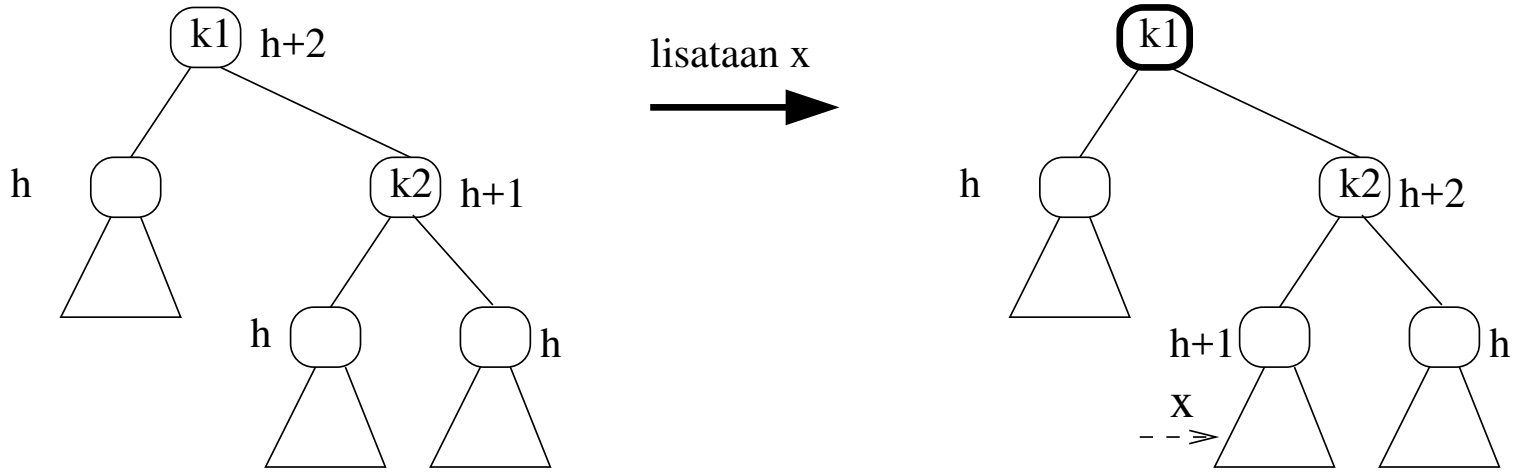


- Jatketaan lisäämällä puuhun 16. Lisäys ei riko tasapainoa
- Kun nyt lisätään puuhun 15 joutuu solmu 7 epätasapainoon
- Epätasapainon aiheuttaa nyt lisäys oikean lapsen vasempaan alipuuhun
- Tilanne eroaa aiemmista epätasapainon syistä eli lisäys oikean lapsen oikeaan alipuuhun ja lisäys vasemman lapsen vasempaan alipuuhun, jotka korjautuivat kierto-operaatiolla



- Kumpikaan kierto-operaatio ei näytä korjaavan solmun 7 epätasapainoa

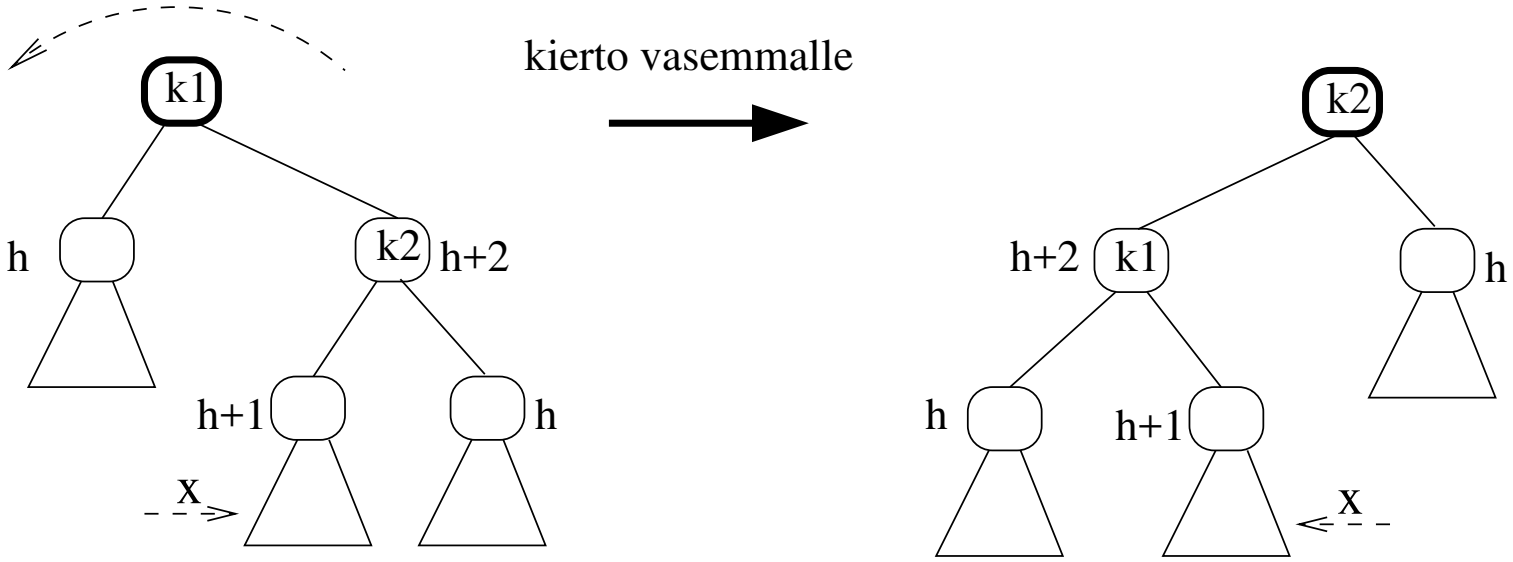
- Tutkitaan tarkemmin tätä tilannetta, johon kierto-operaatio ei näytä tehoavan
- Olkoon $k1$ syvimmällä epätasapainossa lisäyksen jälkeen oleva solmu
- Käsitellään ensin tilannetta, jossa lisätty solmu menee $k1$:n oikean lapsen vasempaan alipuuhun



- Epätasapainoa solmuun $k1$ ei olisi syntynyt, ellei puun tilanne ennen lisäystä olisi ollut täsmälleen vasemmalla oleva, tämän voi päätellä sivun 320 tapaan

- Solmun $k1$ kierto oikealle ei ainakaan voi ratkaista ongelmaa, sillä se ainoastaan pahentaa epätasapainoa

- Seuraavasta huomaamme, että $k1$:n kierto vasemmalle ainoastaan muuttaa tilanteen peilikuvaksi: alipuu pysyy epätasapainossa, sillä $k2$:sta tulee epätasapainoinen ja syynä on se, että uusi solmu x on sen vasemman lapsen oikeassa alipuussa

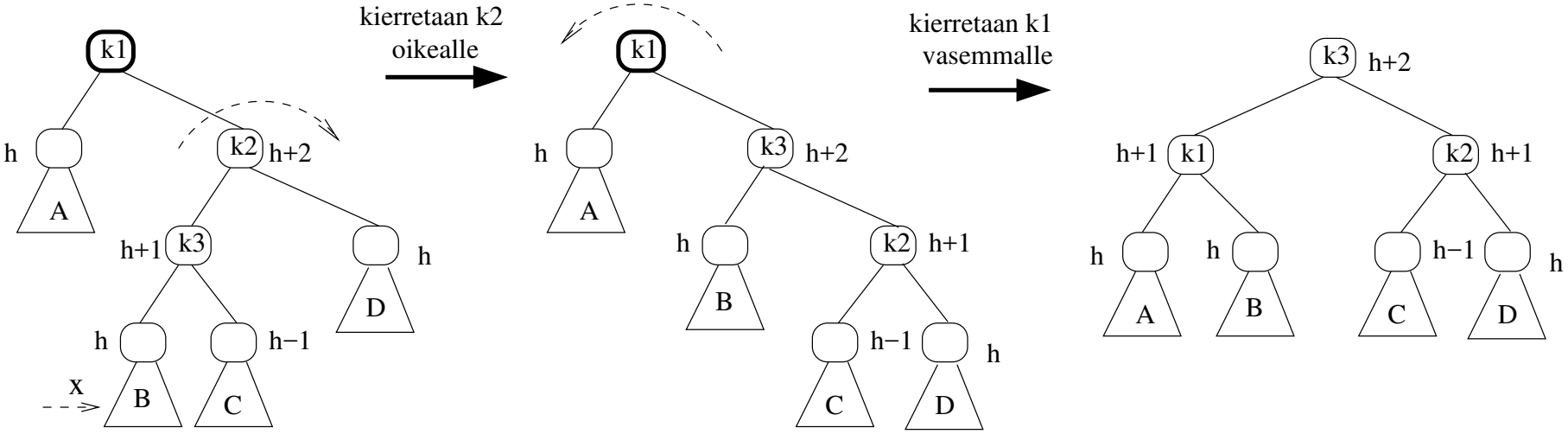


- On keksittävä jokin muu ratkaisu

- Solmun $k1$ epätasapainon siis aiheuttaa sen oikean lapsen $k2$ vasempaan alipuuhun tehty lisäys. Merkitään tämän juurta $k3$:lla
- Lisäys voi olla tehty joko $k3$:n oikeaan tai vasempaan alipuuhun
- Osoittautuu, että kumpikin näistä tapauksista korjautuu samalla tekniikalla
- Tarkastellaan ensin tilannetta, jossa lisäys on tehty $k3$:n vasempaan alipuuhun, jota merkitään seuraavan sivun kuvassa B:llä
- Tekemällä epätasapainoisen solmun $k1$:n oikealle lapselle $k2$ ensin kierto oikealle ja sitten $k1$:lle kierto vasemmalle saadaan puu tasapainoon

- Kiertämällä epätasapainoisen lapsi $k2$ oikealle luodaan tilanne, jossa epätasapainon syy on $k1$:n oikean lapsen oikeassa alipuussa

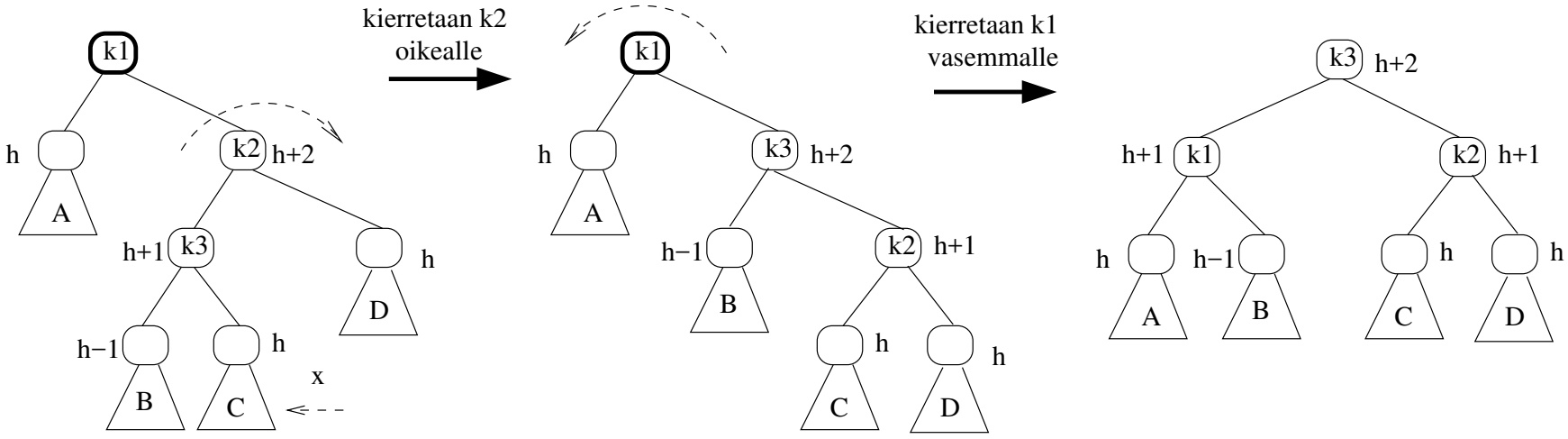
- Tämä on sama tilanne, johon törmäsimme aiemmin, eli epätasapainon korjaa $k1$:n kierto vasemmalle



- Kahden kierto-operaation seurauksena koko alipuun korkeus on sama kuin ennen lisäystä

- Koska siis oli oletettu, että $k1$ on alin epätasapainoinen solmu, on kiertojen jälkeen puu taas tasapainossa

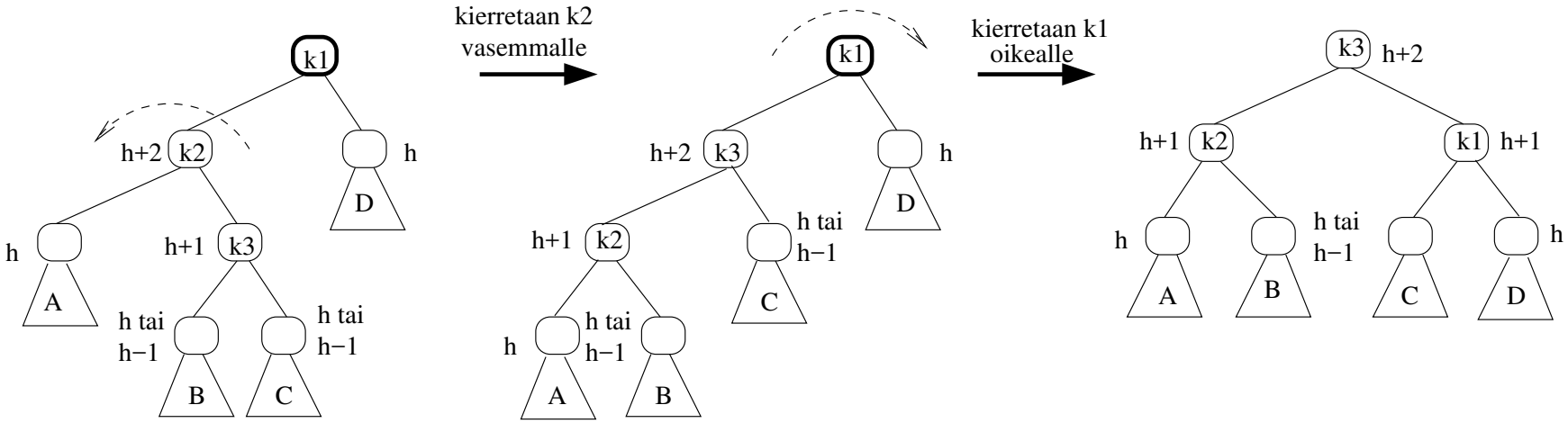
- Seuraavassa vielä näemme, että tilanne, jossa lisäys on tehty $k3$:n oikeaan alipuuhun ratkeaa myös kahdella kierrolla



- Eli jos solmun $k1$ epätasapainon aiheuttaa sen oikean lapsen $k2$ vasempaan alipuuhun tehty lisäys, puu tasapainoittuu tekemällä ensin oikea kierto $k2$:lle ja sitten vasen kierto $k1$:lle

- Näiden operaatioiden muodostamaa kokonaisuutta sanotaan oikea-vasen-kaksoiskierroksi

- Jäljelle jää vielä edellisen tapauksen peilikuva
- Jos solmun $k1$ epätasapainon aiheuttaa sen vasemman lapsen $k2$ oikeaan alipuuhun tehty lisäys, puu tasapainoittuu tekemällä ensin vasen kierto $k2$:lle ja sitten oikea kierto $k1$:lle
- Näiden operaatioiden muodostamaa kokonaisuutta sanotaan vasen-oikea-kaksoiskierroksi
- Seuraavassa vielä näemme miten tilanne, jossa lisäys on tehty $k3$:n oikeaan alipuuhun, ratkeaa kahdella kierrolla



- Kaksoiskierto-operaatioiden toteuttaminen on helppoa, riittää kutsua normaalia kiertoa oikeille solmuille

  RightLeftRotate(k1)
  1 k2 = k1.right
  2 k1.right = RightRotate(k2);
  3 return LeftRotate(k1);

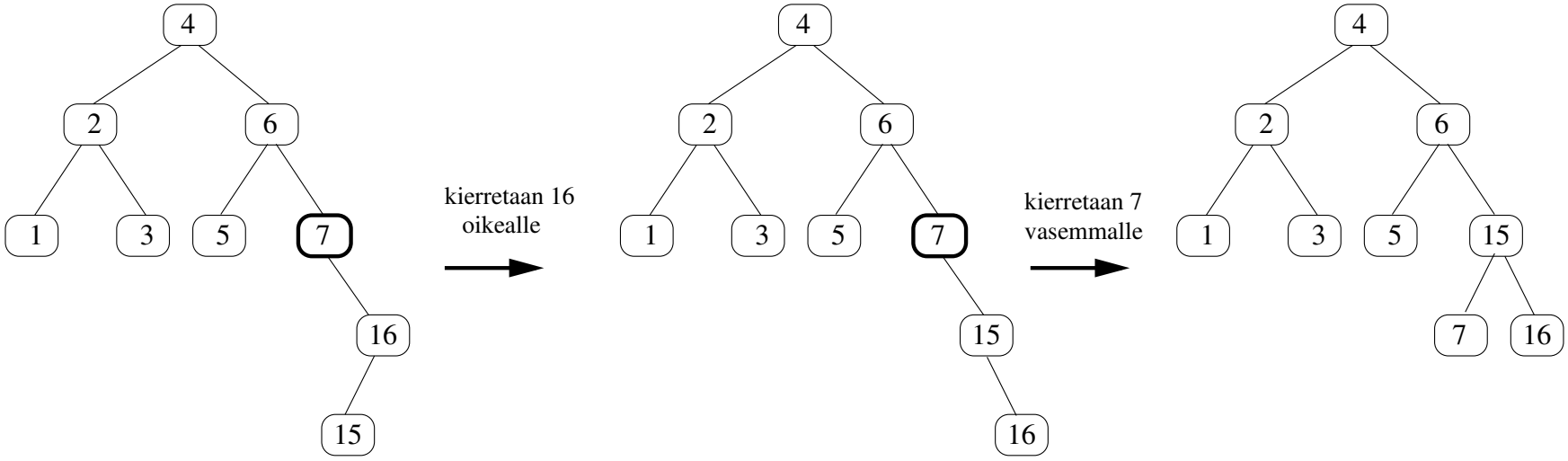
  LeftRightRotate(k1)
  1 k2 = k1.left
  2 k1.left = LeftRotate(k2);
  3 return RightRotate(k1);

- Myös kaksoiskierrossa on alipuun uusi juuri palautettava, jotta kutsuja pystyy laittamaan sen vanhempansa lapseksi

- Koska yksittäiset kierto-operaatiot ovat vakioaikaisia ja vakiotilaisia, on selvää, että myös kaksoiskierrot vain vakioaikaisia ja vakiotilaisia operaatioita

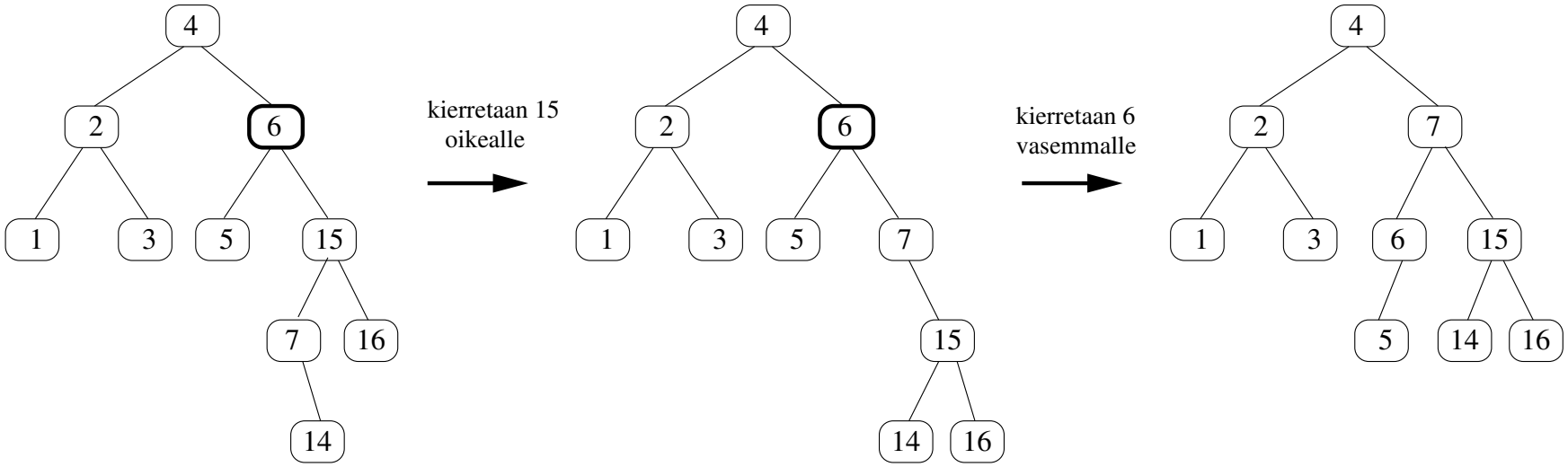
- Ennenkuin esitämme AVL-puuhun lisäyksen pseudokoodina, jatketaan jo aloittamaamme esimerkkiä

- Jäimme tilanteeseen, missä solmujen 15 ja 16 lisäys aiheutti solmuun 7 epätasapainon
- Epätasapaino johtuu oikean lapsen vasempaan alipuuhun suoritetusta lisäyksestä
- Äsken havaitun perusteella oikea-vasen-kaksoiskierto tasapainottaa puun, eli
  - Ensin kierretään oikeaa lasta 16 oikealle, ja sen jälkeen
  - kierretään epätasapainossa olevaa solmua 7 vasemmalle
- Näin puun tasapaino palautuu



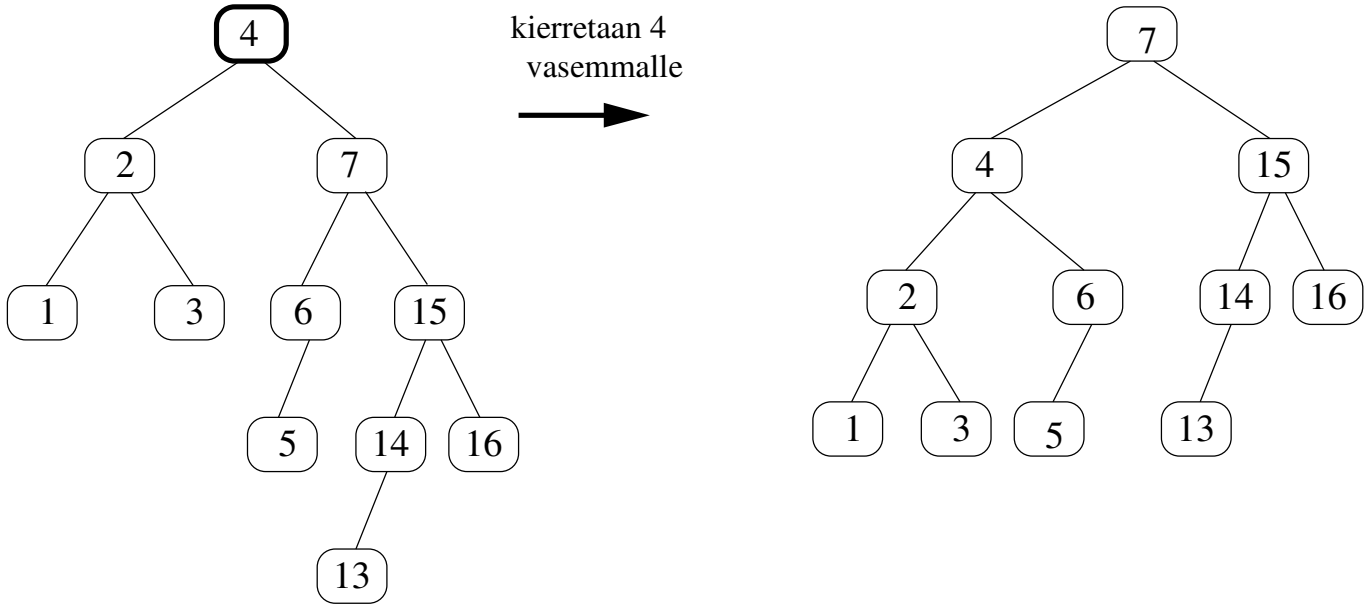
- Lisätään 14. Solmu 6 menee epätasapainoon. Syy epätasapainolle on lisäys oikean lapsen vasempaan alipuuhun.

- Ratkaisu ongelmaan on siis oikea-vasen-kaksoiskierto, eli ensin oikeaa lasta 15 kierretään oikealle ja sen jälkeen solmua 6 vasemmalle



- Lisätään 13. Puun juuri, eli solmu 4 menee epätasapainoon. Syy epätasapainolle on lisäys oikean lapsen oikeaan alipuuhun

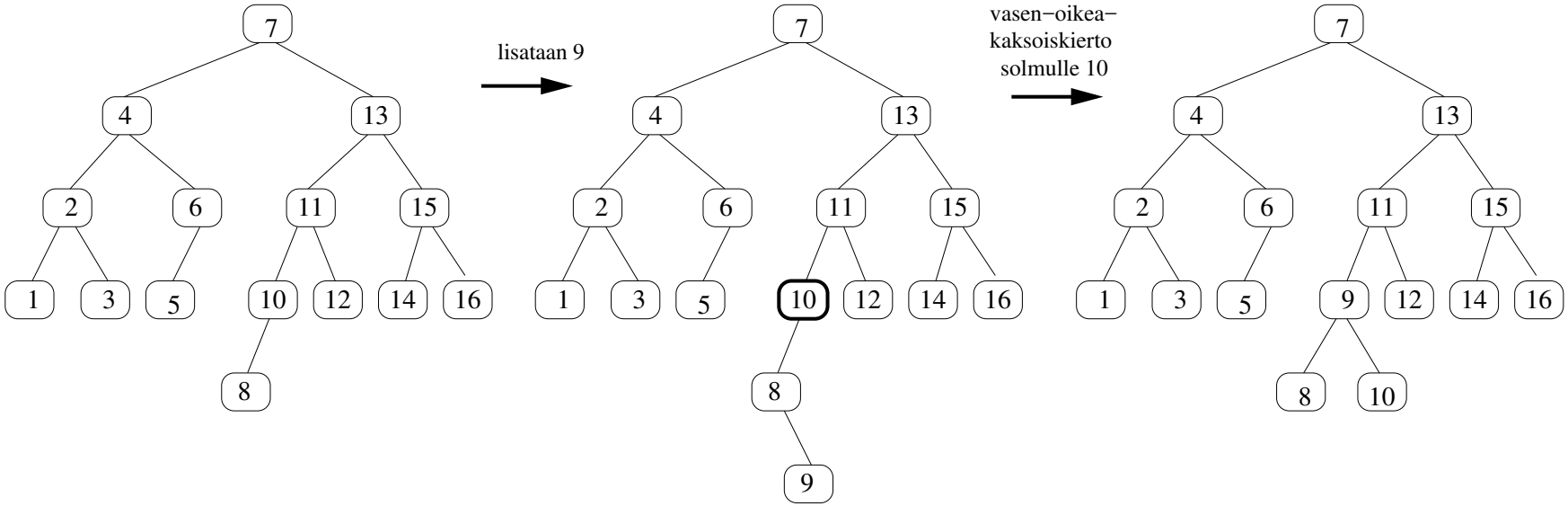
- Kyseessä on siis helpompi tapaus, joka korjautuu yhdellä kierrolla, eli kierretään juurisolmua vasemmalle ja tasapaino palautuu



- Jos jatketaan lisäämällä solmut 12, 11 ja 10, pitää kaikkien lisäyksien yhteydessä tehdä yksi kierto

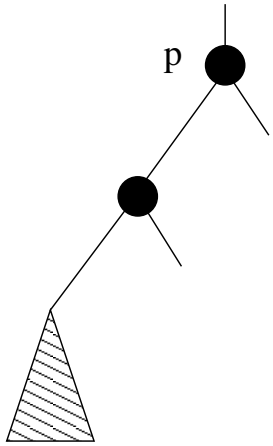
- Edellisten jälkeen solmun 8 voi lisätä puuhun ilman tasapainon rikkoutumista Tuloksena on kuvassa vasemmalla oleva puu

- Jos tähän vielä lisätään solmu 9, solmu 10 menee epätasapainoon. Syynä sen vasemman lapsen oikeaan alipuuhun tehty lisäys. Tilanne korjautuu vasen-oikea-kaksoiskierrolla, eli ensin 8 vasemmalle ja sen jälkeen 10 oikealle

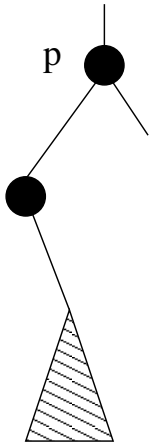


- Esitetään AVL-puun tasapainossa pitävä lisäysoperaatio vielä pseudokoodina
- Lisäys siis tapahtuu seuraavasti
  - lisätään uusi solmu puuhun kuten normaalin binäärihakupuun yhteydessä, asetaan uuden solmun korkeudeksi 0
  - lisäys saattaa viedä puun epätasapainoon
  - epätasapainoon joutuneiden solmujen täytyy sijaita polulla lisätystä solmusta juureen
  - lähdetään kulkemaan tätä polkua ylöspäin ja jos löydetään epätasapainoinen solmu, tehdään tarvittavat kierto-operaatiot
  - aiemmin tehtyjen havaintojen perusteella riittää että palautetaan alimpana puussa oleva epätasapainoinen solmu (eli se joka tulee ensimmäisenä vastaan reitillä lisätystä juureen) tasapainoon
  - eli jos ensimmäisenä vastaantuleva epätasapainotilanne korjataan, menee puu tasapainoon
  - lisäys on voinut muuttaa lisätyn esivanhempien korkeutta, joten kuljettaessa juurta kohti on matkalla vastaan tulevien solmujen korkeuskentät päivitettävä
- Seuraavalla sivulla vielä kootusti epätasapainotilanteessa tarvittavat kierto-operaatiot

- Jos solmun p epätasapainon syy on sen vasemmassa alipuussa, tapauksia kaksi

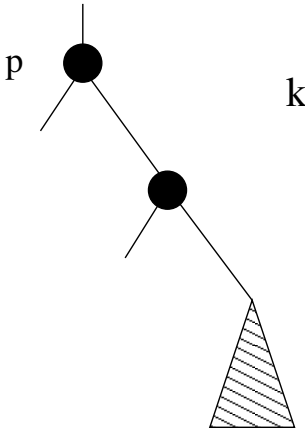


korjaava toimenpide:
RightRotate(p)

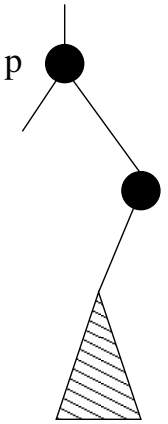


korjaava toimenpide:
LeftRightRotate(p)
eli
LeftRotate(p.left)
RightRotate(p)

- Jos solmun p epätasapainon syy on sen oikeassa alipuussa, tapaukset ovat:



korjaava toimenpide:
LeftRotate(p)

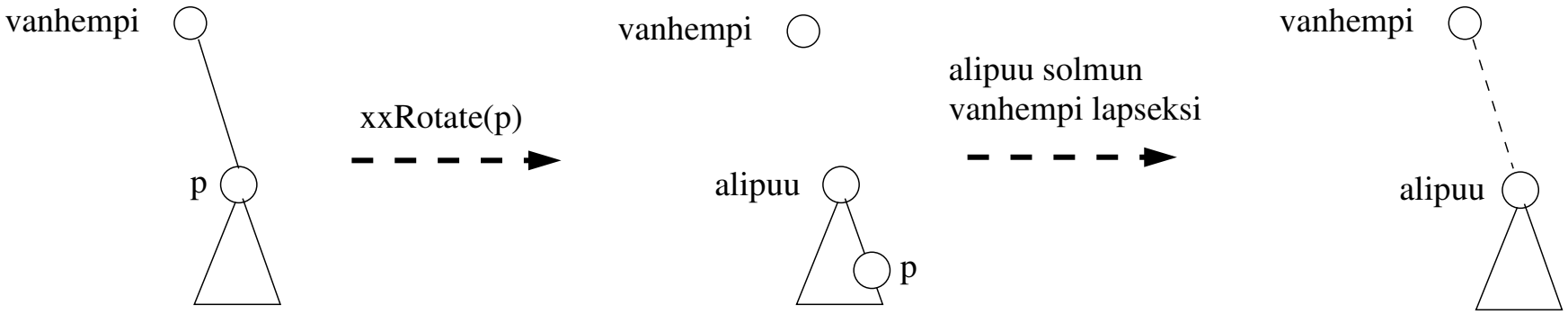


korjaava toimenpide:
RightLeftRotate(p)
eli
RightRotate(p.right)
LeftRotate(p)

- Esitetään algoritmi ensin korkealla tasolla. Algoritmi käyttää sivulla 291 esiteltyä tasapainottamattoman binääripuun lisäysoperaatiota jota on muokattu siltä osin, että se palauttaa viitteen lisättyyn solmuun ja asettaa $uusi.height = 0$

```
AVL-insert(T,k)
1   uusi = insert(T,k)
2   p = uusi.parent
3   while p ≠ NIL
4       if p epätasapainossa
5           vanhempi = p.parent
6           if epätasapainon syy vasemman lapsen vasen alipuu
7               alipuu = RightRotate(p)
8           elsif epätasapainon syy vasemman lapsen oikea alipuu
9               alipuu = LeftRightRotate(p)
10          elsif epätasapainon syy oikean lapsen oikea alipuu
11              alipuu = LeftRotate(p)
12          else  // epätasapainon syy oikean lapsen vasen alipuu
13              alipuu = RightLeftRotate(p)
14          aseta alipuu solmun vanhempi lapseksi
                tai jos p oli juuri, niin tee alipuu:sta uusi juuri
15          vanhempi.height = max( Height(vanhempi.left),
                                   Height(vanhempi.right))+1
16          return
17      p.height = max( Height(p.left), Height(p.right) ) +1
18      p = p.parent
```

- Rivillä 2 viite p saa arvokseen lisätyn solmun vanhemman
- Toistolauseessa tarkastetaan onko p tasapainossa. Jos on, niin p saa arvokseen vanhempansa, tämä tapahtuu rivillä 18, eli jatketaan tasapainon tarkastamista yhtä askelta lähempää puun juurta
- Toistolausetta jatketaan niin kauan kunnes p on NIL eli ollaan käyty kaikki solmut lisätyn ja juuren välillä läpi tai kunnes tehdään tasapainoitusoperaatio
- Jos p on epätasapainossa, haarautuu käsittely neljään tapaukseen joissa kussakin tehdään kierto- tai kaksoiskierto-operaatio
- Kierron seurauksena p muuttaa paikkaa ja aiempi p:n vanhempi täytyy liittää kierron seurauksena syntyneeseen alipuun juureen



- Seuraavalla sivulla vielä pseudokoodi detaljitasolla esitettynä

```
AVL-insert(T,k)
1   uusi = insert(T,k)
2   p = uusi.parent
3   while p ≠ NIL
4       if height(p.left) == height(p.right)+2    // vasen lapsi aiheutti epätasap.
5           vanhempi = p.parent
            // onko syy vasemman lapsen vasemmassa vai oikeassa alipuussa?
6           if height(p.left.left) > height(p.left.right)
7               alipuu = RightRotate(p)
8           else
9               alipuu = LeftRightRotate(p)
10          if vanhempi == NIL
11              T.root = alipuu
12          elsif vanhempi.left == p
13              vanhempi.left = alipuu
14          else
15              vanhempi.right = alipuu
16          if vanhempi ≠ NIL
17              vanhempi.height =max(   Height(vanhempi.left),
                                        Height(vanhempi.right))+1
18          return                  // kierrot tehty, eli puu on palannut tasapainoon
19      if height(p.right) == height(p.left)+2
            // tilanne jossa oikea lapsi aiheuttaa epätasapainon seuraavalla sivulla
```

- **AVL-insert**(T,k) jatkuu...

```
        // vasen lapsi aiheutti epätasapainon edellisellä sivulla
19      if height(p.right) == height(p.left)+2     // oikea lapsi aiheuttaa epätasap.
20          vanhempi = p.parent
            // onko syy vasemman lapsen oikeassa vai vasemmassa alipuussa?
21          if height(p.right.right) > height(p.right.left)
22              alipuu = LeftRotate(p)
23          else
24              alipuu = RightLeftRotate(p)
25          if vanhempi == NIL
26              T.root = alipuu
27          elsif vanhempi.left == p
28              vanhempi.left = alipuu
29          else
30              vanhempi.right = alipuu
31          if vanhempi ≠ NIL
32              vanhempi.height =max(   Height(vanhempi.left),
                                        Height(vanhempi.right))+1
33          return // kierrot tehty, eli puu tasapainossa ja korkeus sama kuin ennen
34      p.height = max(Height(p.left), Height(p.right) )+1
35      p = p.parent // jatketaan kohti juurta
```

- Rivit 5-16 hoitavat tilanteen, jossa solmun p epätasapaino johtuu sen vasemmasta alipuusta
  - Rivillä 7 käsitellään tilanne jossa epätasapainon syy vasemman lapsen vasen alipuu
  - Rivillä 9 tilanne jossa epätasapainon syy vasemman lapsen oikea alipuu
  - Rivit 10-16 laittavat kierretyn alipuun sen vanhemman lapseksi
  - Rivi 10 huomioi heti erikoistapauksen, jossa kierrettiin puun juurta
- Riveillä 19-32 tilanteen, jossa solmun p epätasapainon syy oikeassa alipuussa
  - Rivillä 21 tilanne jossa epätasapainon syy oikean lapsen oikea alipuu
  - Rivillä 23 tilanne jossa epätasapainon syy oikea lapsen vasen alipuu
  - Rivit 25-30 laittavat kierretyn alipuun sen vanhemman lapseksi
- Noustaessa lisätystä kohti juurta päivitetään matkan varrella kohdattujen solmujen korkeuskentät rivillä 34 ja kiertojen tapauksessa riveillä 17 ja 32
- Jos epätasapainoinen solmu löytyy ja kierto-operaatio suoritetaan, on taattua että puu menee tasapainoon, korkeus ei ole muuttunut, ja algoritmi lopettaa (rivit 18 ja 32)
- Muussa tapauksessa jatketaan solmujen tasapainon tutkimista aina juureen asti

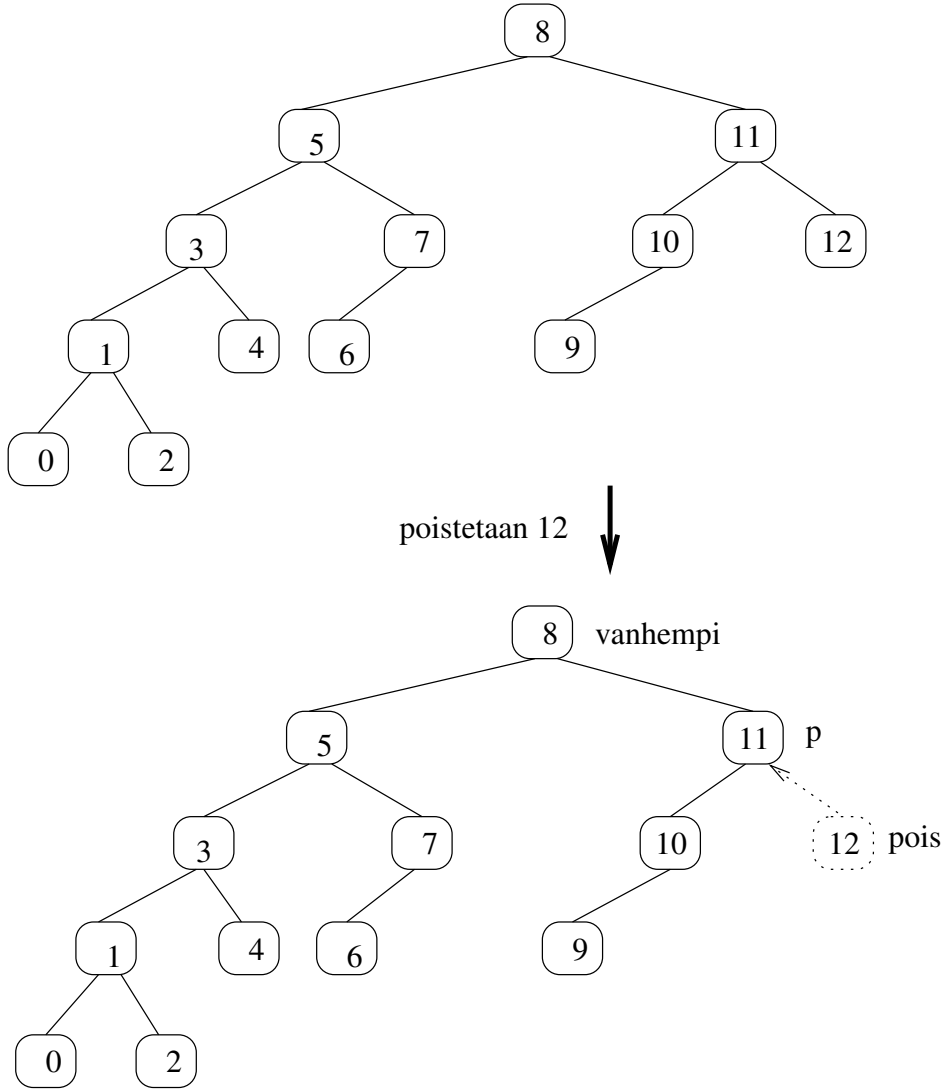
- Analysoidaan algoritmin aika- ja tilavaativuutta
- Käytetään analyysin pohjana sivun 344 abstraktimpaa versiota
- Ensin kutsutaan normaalia binäärihakupuun lisäysoperaatiota, jonka aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen. Olemme osoittaneet aiemmin, että AVL-puun korkeus solmujen lukumäärän n suhteen on $\mathcal{O}(\log n)$, joten insert vie aikaa siis logaritmisesti
- **AVL-insert** suorittaa maksimissaan kaksi kierto-operaatiota, jotka molemmat ovat vakioaikaisia
- Pahimmassa tapauksessa algoritmi myös kulkee reitin lisätystä solmusta juureen, eli puun korkeuden verran, tämä on vaativuudeltaan myös $\mathcal{O}(\log n)$
- Eli algoritmi koostuu kahdesta $\mathcal{O}(\log n)$ aikaa vievästä osasta, siispä kokonaisaikavaativuus $\mathcal{O}(\log n)$
- Binäärihakupuun insertin ja kierto-operaatioiden tilavaativuus on todettu jo aiemmin vakioksi, samoin muutkin osat **AVL-insert**:istä käyttävät ainoastaan vakiomäärän apumuuttujia, eli kokonaisuudessaan algoritmin tilavaativuus on vakio

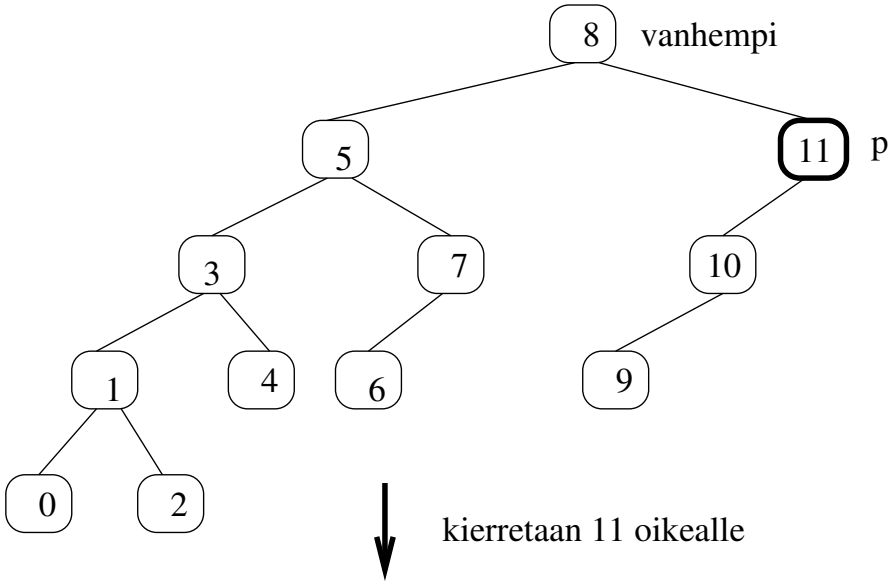
- **AVL-delete** on hiukan **AVL-insert**:iä monimutkaisempi operaatio
- Solmu poistetaan ensin käyttäen normaalia **delete**-operaatiota
- Algoritmi käyttää sivuilla 296–297 esiteltyä normaalin binääripuun poisto-operaatiota, joka palauttaa viitteen siihen solmuun, joka puusta todellisuudessa poistettiin (kaikissa tapauksissahan kyseessä ei ole sama solmu, jonka sisältö haluttiin poistaa)
- Poistetun solmun $parent$-kenttä sisältää edelleen viitteen poistetun vanhempaan
- Poistetun vanhempi on alin solmu, jonka tasapaino on voinut häiriintyä poisto-operaation seurauksena
- Algoritmi etenee poistetun vanhemmasta ylöspäin juureen asti ja korjaa matkalla vastaan tulleet epätasapainoisuudet
- Erona **AVL-insert**-operaatioon on nyt se, että ensimmäisenä vastaantulevan epätasapainoisen solmun tasapainoitus ei välttämättä palauta koko puuta takaisin tasapainoon
- Epätasapainoisten etsimistä on jatkettava joka tapauksessa juureen asti
- Esitetään algoritmista vain abstrakti versio

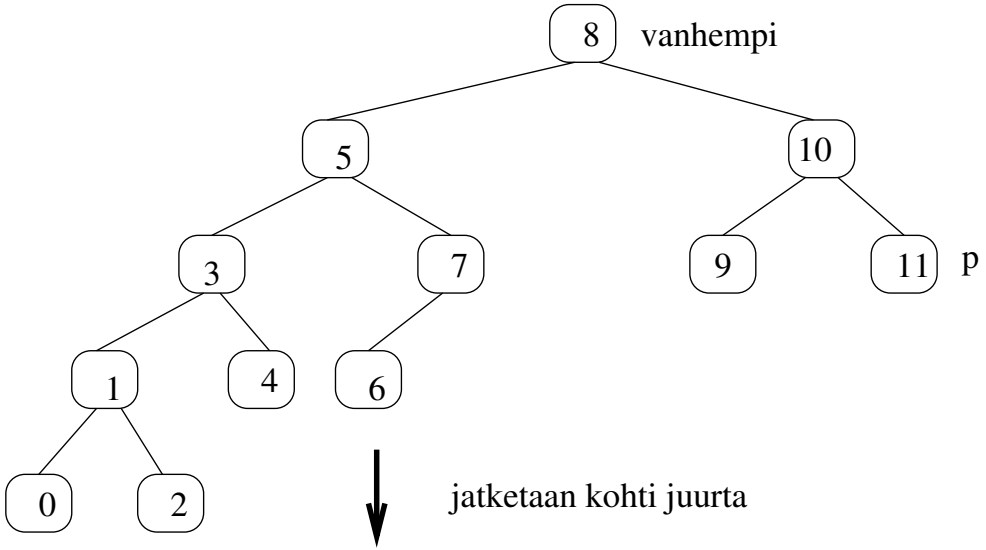
```
AVL-delete(T,x)
1   pois = delete(T,x)
2   p = pois.parent
3   while p ≠ NIL
4       if p epätasapainossa
5           vanhempi = p.parent
6           if epätasapainon syy vasemman lapsen vasen alipuu
7               alipuu = RightRotate(p)
8           elsif epätasapainon syy oikean lapsen oikea alipuu
9               alipuu = LeftRotate(p)
10          elsif epätasapainon syy vasemman lapsen oikea alipuu
11              alipuu = LeftRightRotate(p)
12          else  // epätasapainon syy oikean lapsen vasen alipuu
13              alipuu = RightLeftRotate(p)
14          if p oli puun juuri
15              T.root = alipuu
16              return
17          aseta alipuu solmun vanhempi lapseksi
18          p = vanhempi
19      else
20          p.height = max( Height(p.left), Height(p.right) ) +1
21          p = p.parent
```

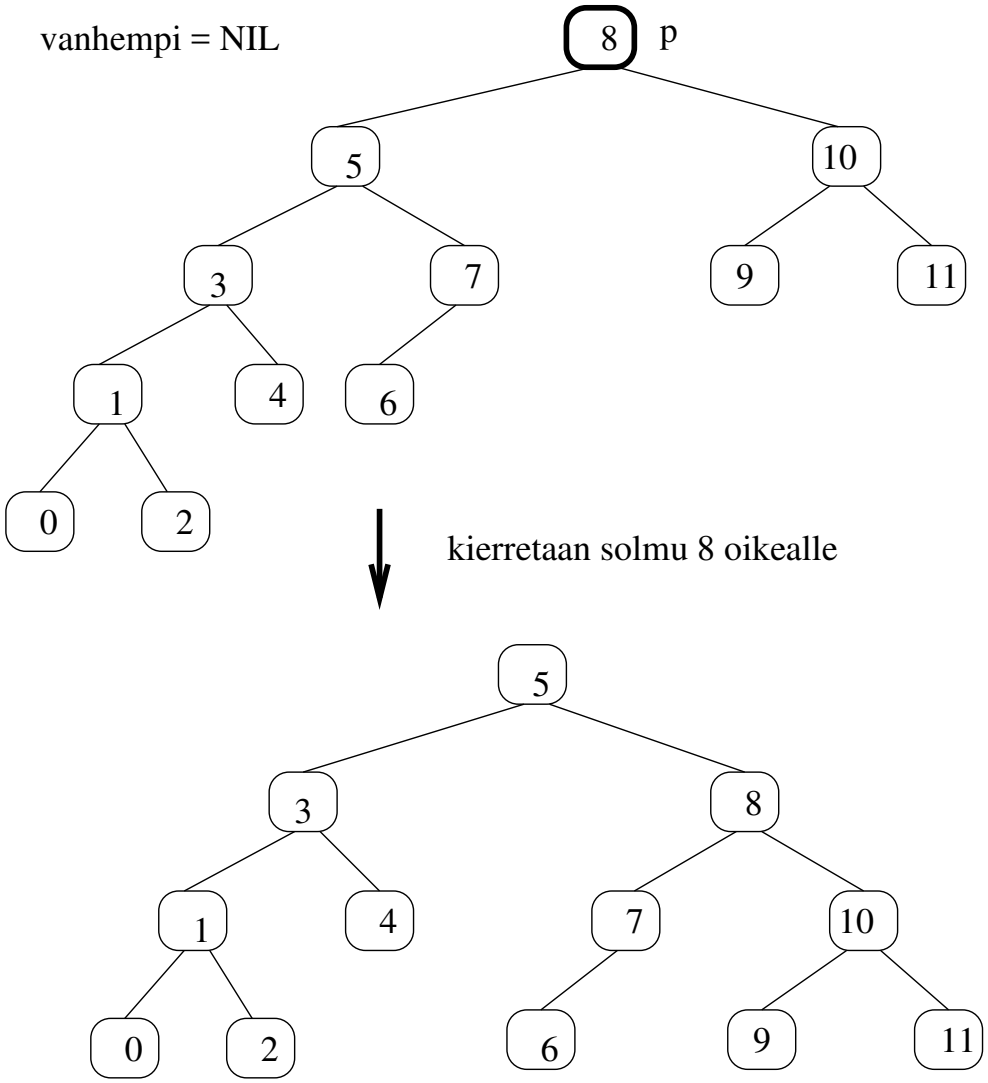
- Alussa siis kutsutaan normaalia binäärihakupuun **delete**-operaatiota, näin saadaan tietoon se solmu $pois$, joka todellisuudessa poistettiin puusta
- Rivillä 2 laitetaan apuviite p viittaamaan poistetun vanhempaan
- Rivillä 4 tarkastetaan, onko solmu johon p viittaa mennyt epätasapainoon
- Jos on, niin epätasapaino korjataan riveillä 5-19
  - riveillä 6-13 tutkitaan minkä tyyppisestä epätasapainosta on kyse ja tehdään korjaava (mahdollisimman yksinkertainen) kierto-operaatio
  - kierron seurauksena p:n alkuperäiselle vanhemmalle $vanhempi$ tulee uusi lapsi: solmu $alipuu$, jonka kierto-operaatio palauttaa
  - riveillä 14-16 erikoistapaus, jossa kierrettävä solmu oli puun juuri
- Tapahtui tasapainoitus tai ei, jatkuu epätasapainoisten solmujen etsintä p:n vanhemmasta, etsintää jatketaan niin kauan kunnes kaikki solmut matkalla poistetusta puun juuren on käyty läpi
- Poisto-operaatio saattaa muuttaa poistettujen edeltäjien korkeuksia tämän takia noustaessa poistetusta kohti juurta, matkan varrella kohdattujen solmujen korkeuskentät päivitetään rivillä 20
- Kierto-operaatiot päivittävät kierrettyjen alipuiden korkeuskentät, joten kierron yhteydessä algoritmin ei tarvitse koskea korkeuskenttiin

- Tarkastellaan algoritmin toimintaa esimerkin avulla




8
vanhempi
5
11
p
3
7
10
1
4
6
9
0
2
kierretaan 11 oikealle


8
vanhempi
5
10
3
7
9
11
p
1
4
6
0
2
jatketaan kohti juurta


vanhempi = NIL
8
p
5
10
3
7
9
11
1
4
6
0
2
kierretaan solmu 8 oikealle
5
3
8
1
4
7
10
0
2
6
9
11

- Analysoidaan algoritmin aika- ja tilavaativuutta

- Ensin kutsutaan normaalia binäärihakupuun poisto-operaatiota, jonka aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen. Olemme osoittaneet aiemmin, että AVL-puun korkeus solmujen lukumäärän n suhteen on $\mathcal{O}(\log n)$, joten **delete** vie aikaa siis logaritmisesti

- **AVL-delete** suorittaa pahimmassa tapauksessa puun korkeuden verran kierto-operaatioita kulkiessaan poistetusta solmusta juureen

- Kierto-operaatiot ovat vakioaikaisia, eli tasapainon korjaukseen kuluu aikaa enintään $\mathcal{O}(\log n)$

- Algoritmi siis koostuu kahdesta $\mathcal{O}(\log n)$ aikaa vievästä osasta, eli kokonaisaikavaativuus $\mathcal{O}(\log n)$

- Eli vaikka operaatio onkin kiertojen takia hieman raskaampi kuin normaali poisto, ei tasapainon ylläpitäminen muuta operaation aikavaativuuden kertaluokkaa

- Binäärihakupuun poisto- ja kierto-operaatioiden tilavaativuus on todettu jo aiemmin vakioksi, samoin muutkin osat **AVL-delete**:ssä käyttävät ainoastaan vakiomäärän apumuuttujia, eli kokonaisuudessaan algoritmin tilavaativuus on vakio

Yhteenveto AVL-puista:

- AVL-tasapainoehto takaa puun korkeuden $\mathcal{O}(\log n)$, missä n on alkioiden lukumäärä
- Siis **search** toimii aina ajassa $\mathcal{O}(\log n)$
- Tasapainoehtoa pidetään yllä tekemällä vakioiaikaisia kiertoja hakupolulla
- Pahimmassa tapauksessa tarvitaan yksi kierto-operaatio kussakin hakupolun solmussa
- Edellisestä seuraa, että kaikki AVL-puuna toteutetun joukon operaatiot toimivat pahimmassakin tapauksessa ajassa $\mathcal{O}(\log n)$
- Tasapainottamattomaan puuhun verrattuna AVL-puut tarvitsevat kuhunkin solmuun ylimääräisen $height$-laskurin, jolle käytännössä riittää 8 bittiä sillä puun korkeus on tuskin käytännössä koskaan yli 256
- Periaatteessa muistia voitaisiin hieman säästää tallentamalla vain vasemman ja oikean alipuun korkeuksien ero (-1, 0 tai $+1$) eli tällöin tarvittaisiin vain 2 bittiä

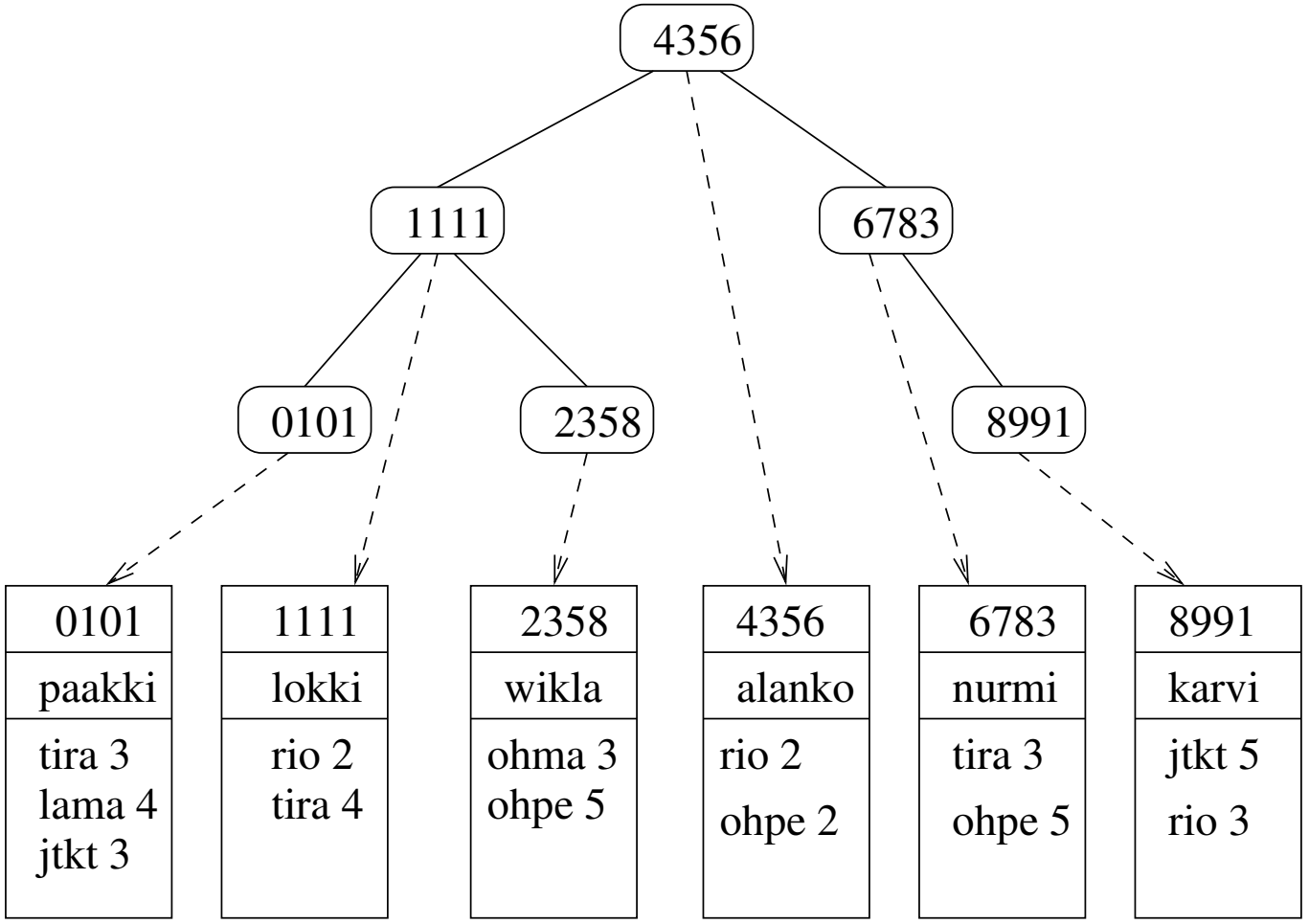
Monta indeksiä

- Olemme käsitelleet yksinkertaistettua tilannetta, jossa puun solmuihin ei ole talletettu muuta kuin avaimen arvo
- Jos organisoitavaa dataa on paljon, paras ratkaisu on tallettaa muu data omana olionaan ja lisätä puusolmuihin avaimen lisäksi viite muun datan tallettavaan olioon
- Puusolmu muodostuu tällöin kentistä:

key	talletettu avain
$data$	viite avaimeen liittyvän tiedon tallettavaan olioon
$left$	viite vasempaan lapseen
$right$	viite oikeaan lapseen
$parent$	viite vanhempaan

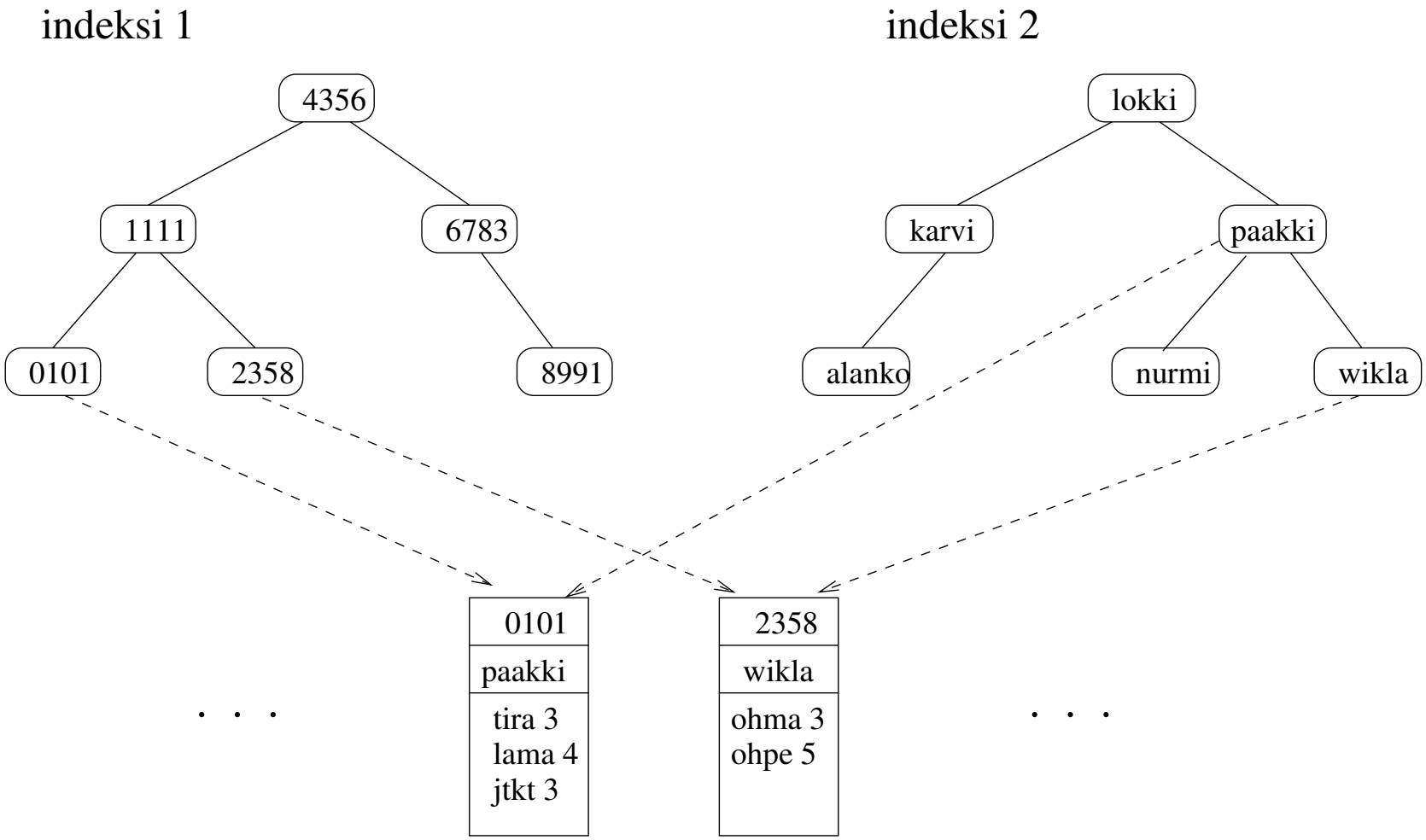
- Näin puu toimii indeksirakenteena, jonka avulla talletettuun tietoon on mahdollista tehdä nopeita hakuja avaimen perusteella

- Esim. opiskelijarekisteri, jossa binäärihakupuu toimii indeksirakenteena opiskelijanumeron suhteen:



- Nyt siis opiskelijan tietojen haku opiskelijanumeron perusteella on nopeaa

- Entä jos haluamme nopeat haut myös nimen perusteella?
- Lisätään samalle datalle toinen indeksirakenne, joka mahdollistaa nopeat haut nimeen perustuen



- Javan valmiista tietorakennetoteutuksista `TreeSet` ja `TreeMap` perustuvat tasapainoisiin binäärihakupuihin

- `TreeSet`:iin talletetaan olioita, joille on määritelty suuruusjärjestys (Tämä tapahtuu joko toteuttamalla ns. `Comparable`-rajapinta tai määrittelemällä järjestyksen antava metodi)

- Oliot on talletettu `TreeSet`:iin niille määritellyssä järjestyksessä, ja olioiden läpikäynti järjestyksessä on nopeaa

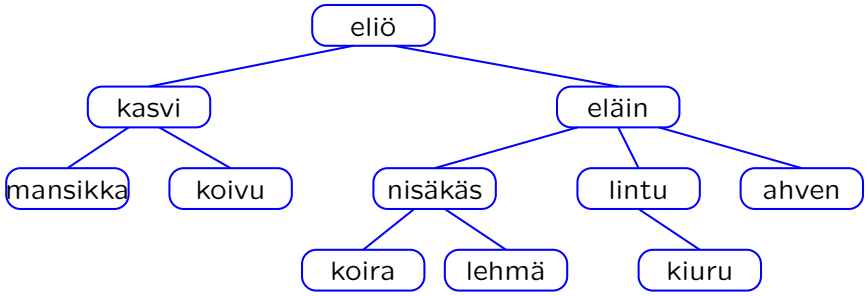
- `TreeSet`:issä olevia olioita ei pysty hakemaan nopeasti mihinkään olion attribuuttiin (esim. opiskelijanumero) perustuen. Nopea, eli $\mathcal{O}(\log n)$ suhteessa talletettujen olioiden määrään n, on ainoastaan testi onko tietty olio `TreeSet`:issä

- `TreeMap` taas toimii edellisten sivujen indeksirakenteiden tapaan, eli `TreeMap`:iin talletetaan avain-dataolio -pareja, ja etsintä avaimeen perustuen on tehokas

- Esim. edellisten sivujen opintorekisteriesimerkin toteutus onnistuisi helposti kahden `TreeMap`:in avulla, toisessa olisi avaimena opiskelijanumero ja toisessa opiskelijan nimi, molemmissa opiskelijan tiedot talletettaisiin erilliseen olioon, joka löytyisi nopeasti sekä nimen että opiskelijanumeron perusteella

Yleisen puun talletus ja läpikäynti

- Kuten jo puuluvun alussa mainittiin, on puille monenlaista käyttöä tietojenkäsittelyssä
- Puita käytetään esim. yleisesti erilaisten hierarkioiden esittämiseen tietojenkäsittelyssä ja muualla:



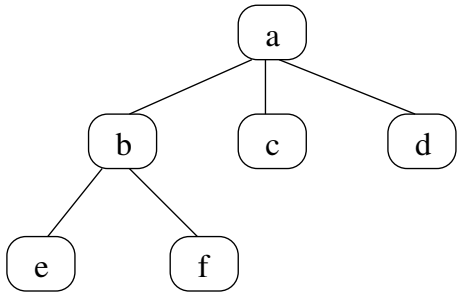
- Muutaman sivun päästä tutustumme puiden käyttöön ongelmanratkaisussa
- Kaikki hyödylliset puut eivät siis suinkaan ole binääripuita tai hakupuita
- Miten voimme tallettaa yleisen puun?

- Jos tiedämme mikä on solmun maksimihaarautumisaste, voimme tallettaa solmuun viitteet kaikkiin mahdollisiin lapsiin

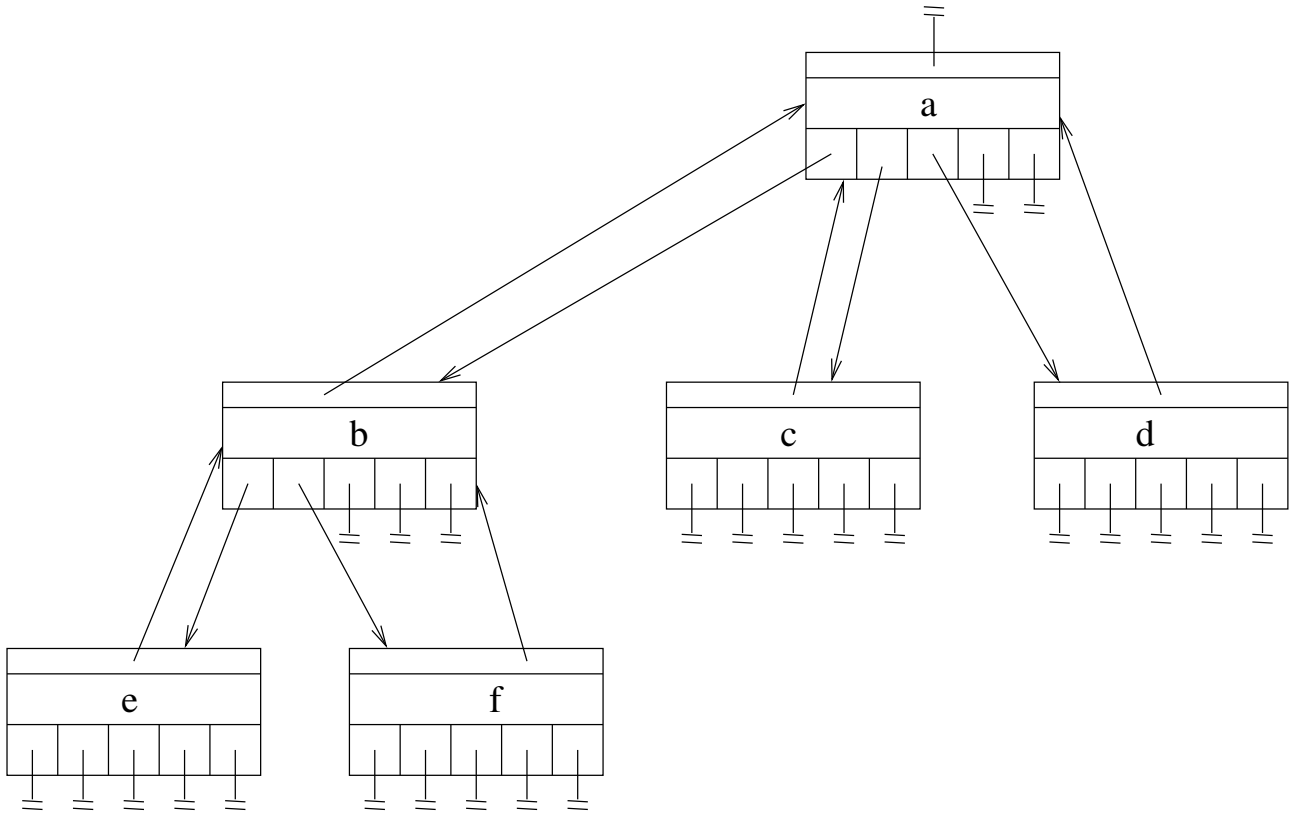
- Eli puusolmu muodostuu tällöin kentistä:

key	talletettu avain
$c1$	viite 1. lapseen
$c2$	viite 2. lapseen
...	
ck	viite k:nteen lapseen
p	viite vanhempaan

- Esimerkki allaolevan puun tallettamisesta seuraavalla sivulla



- Puu talletettuna käyttäen puusolmuja joissa haarautumisaste on 5



- Huomaamme että rakenne tuhlaa paljon muistia tarpeettomiin linkkikenttiin

- Toisaalta voi käydä myös niin että johonkin solmuun tulisikin enemmän lapsia kuin 5

- Esim. Javassa voisimme tallettaa solmun lapset `ArrayList`:iin, joka siis käytännössä on vaihtuvamittainen taulukko

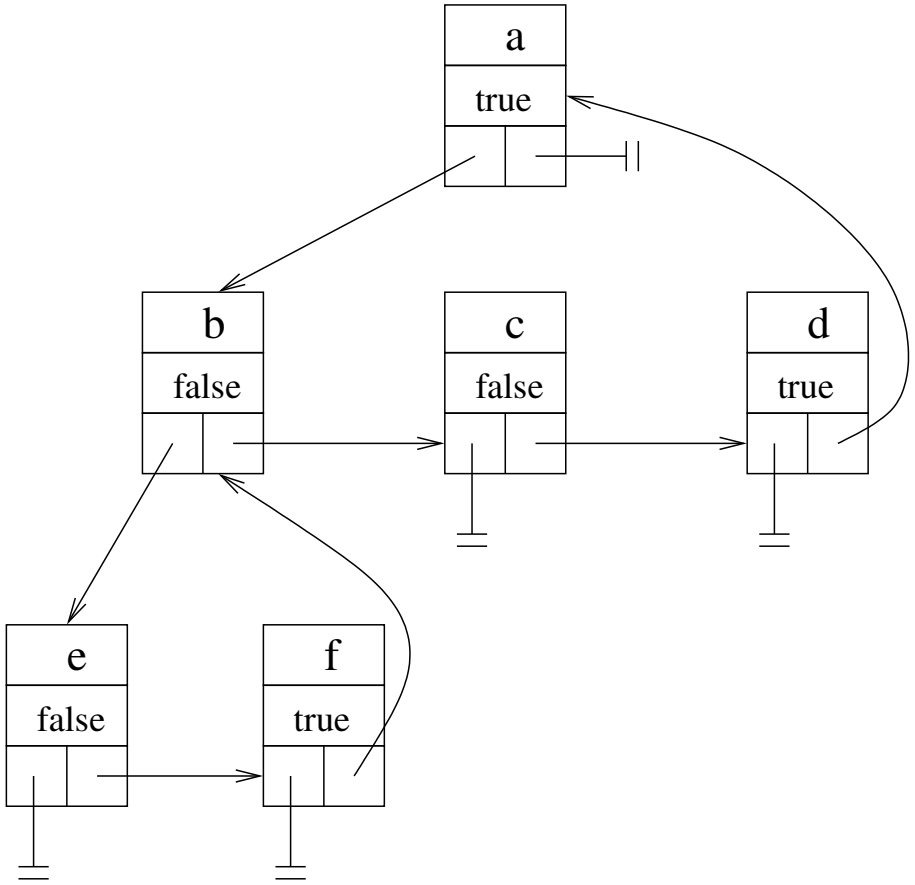
- Näin saataisiin lapsimäärästä joustava, ja turhaa tilaa ei varattaisi kohtuuttoman paljoa

- Muistin käytön kannalta parempi ratkaisu yleisen puun tallettamiseen on kuitenkin seuraava

- Puusolmun kentät

key	talletettu avain
$last$	bitti jonka arvo on true, jos kyseessä on sisaruksista viimeinen
$child$	viite 1. lapseen
$next$	viite seuraavaan sisarukseen (jos $last$ = false), tai vanhempaan (jos $last$ = true)

- Esimerkkipuumme talletettaisiin seuraavasti:



- Muistia ei tuhlaudu turhiin linkkikenttiin ja toisaalta puun haarautumisaste ei ole rajoitettu

- Solmusta x päästään vanhempaan kulkemalla $next$-linkkejä, kunnes on ohitettu sisarus jolla $last$ = true

  parent(x)
  ```
      while x.last == false
          x = x.next
      return x.next
  ```

- Viite solmun x ensimmäiseen lapseen on helppo selvittää

  firstchild(x)
  ```
      return x.child
  ```

- Muut lapset saadaan kutsumalla toistuvasti seuraavaa operaatiota parametrina edellinen löydetty lapsi y

  nextchild(y)
  ```
      if y.last == true return NIL
      else return y.next
  ```

- Viimeisen lapsen jälkeen operaatio palauttaa NIL

- Binäärihakupuun yhteydessä saimme tulostetuksi puun solmut suuruusjärjestyksessä käymällä puun läpi sisäjärjestyksessä, eli ensin vasen lapsi, sitten solmu itse ja lopulta oikea lapsi

- Yleisten puiden kohdalla mielekkäät läpikäyntitavat ovat esijärjestys ja jälkijärjestys

- Esijärjestyksessä käsittelemme ensin solmun ja tämän jälkeen lapset

- Esimerkkipuumme solmut esijärjestyksessä lueteltuna: a, b, e, f, c, d

- Algoritmina

  preorder-tree-walk(x)
  ```
      print x.key
      y = firstchild(x)
      while y ≠ NIL
          preorder-tree-walk(y)
          y = nextchild(y)
  ```

- Kutsu **preorder-tree-walk**$(T.\,root)$ tulostaa nyt puun sisällön esijärjestyksessä. Huom: operaatio ei toimi tyhjälle puulle!

- Jälkijärjestyksessä käsittelemme ensin lapset ja tämän jälkeen solmun itsensä
- Esimerkkipuumme solmut jälkijärjestyksessä lueteltuna: e, f, b, c, d, a
- Algoritmina

```
postorder-tree-walk(x)
    y = firstchild(x)
    while y ≠ NIL
        postorder-tree-walk(y)
        y = nextchild(y)
    print x.key
```

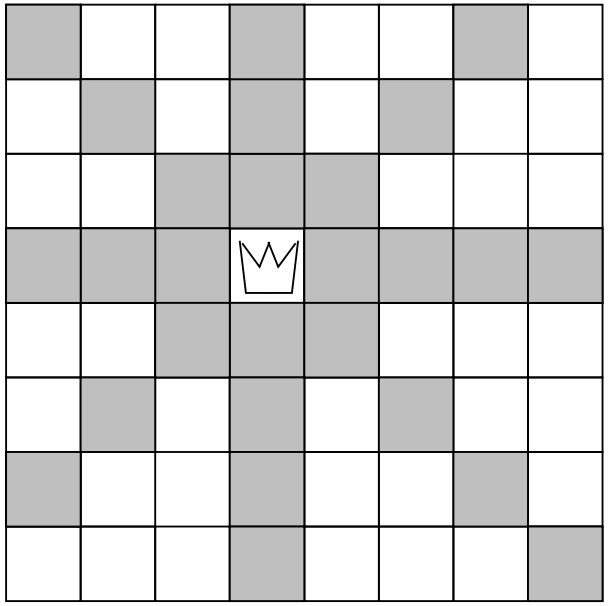
- Toki muitakin tapoja puun läpikäynnille on, esim. leveyssuuntainen läpikäynti missä puun alkiot käydään läpi taso kerrallaan, alkaen juuresta

- Esimerkkipuumme solmut leveyssuuntaisesti lueteltuna: a, b, c, d, e, f

```
levelorder-tree-walk(x)
    Q = tyhjä solmujono
    enqueue(Q, T.root)
    while not empty(Q)
        x = dequeue(Q)
        print x.key
        y = firstchild(x)
        while y ≠ NIL
            enqueue(Q,y)
            y = nextchild(y)
```

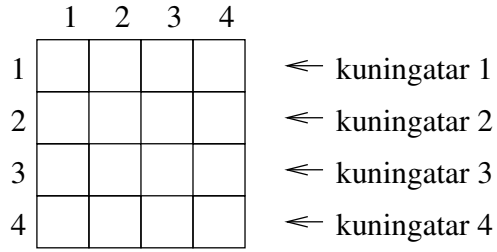
Puut ongelmanratkaisussa: kahdeksan kuningattaren ongelma

- Yksi puiden tärkeistä käyttötavoista on ongelmanratkaisussa tapahtuvan laskennan etenemisen kuvaaminen
- Kahdeksan kuningattaren ongelma: miten voimme sijoittaa shakkilaudalle 8 kuningatarta siten että ne eivät uhkaa toisiaan?
- Kuningatar uhkaa samalla rivillä, sarakkeella sekä diagonaalilla olevia ruutuja

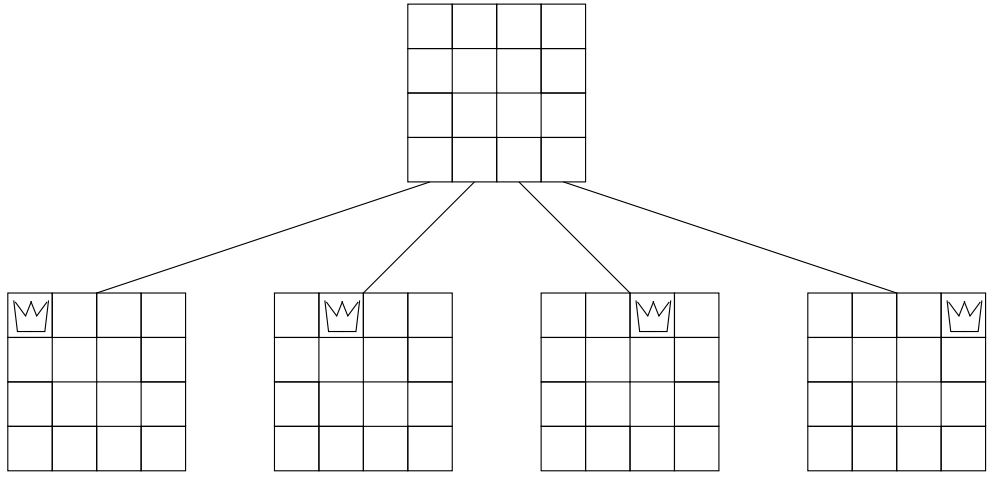


- Yleistetty version ongelmasta: miten saamme sijoitettua n kuningatarta $n \times n$ -kokoiselle shakkilaudalle

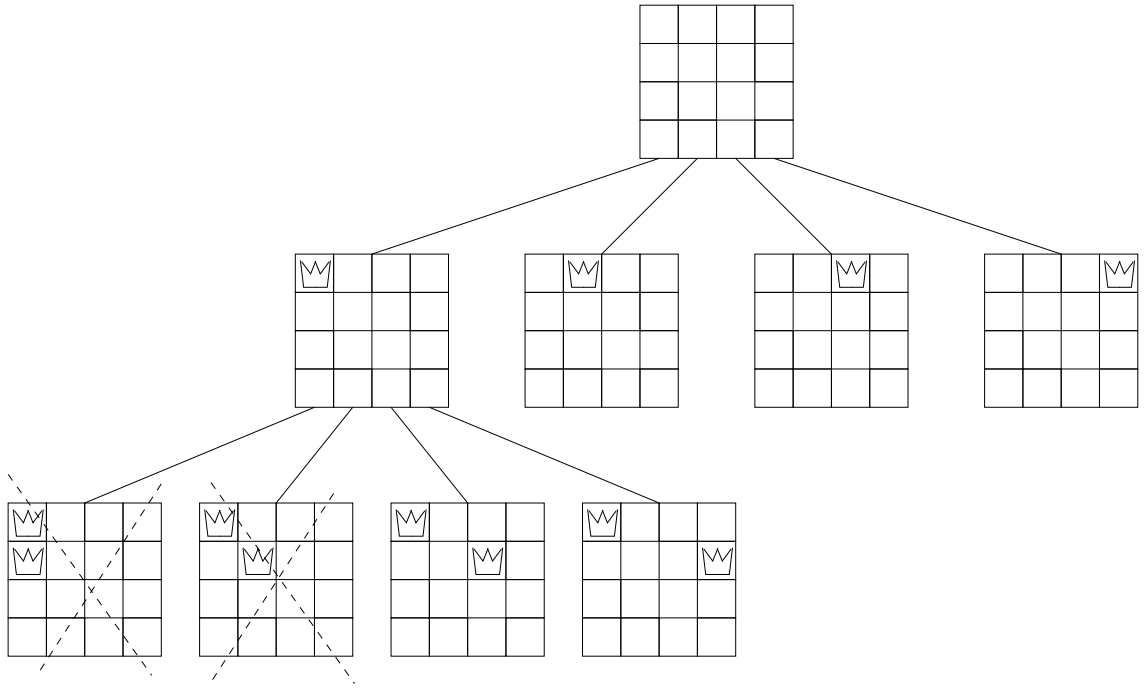
- Tarkastellaan ensin tapausta missä $n = 4$. Selvästi jokaisella rivillä täytyy olla tasan 1 kuningatar:



- Etsitään oikea kuningatarasetelma systemaattisesti
  - aloitetaan tyhjältä laudalta
  - tämän jälkeen asetetaan kuningatar riville 1
  - neljä eri mahdollisuutta:



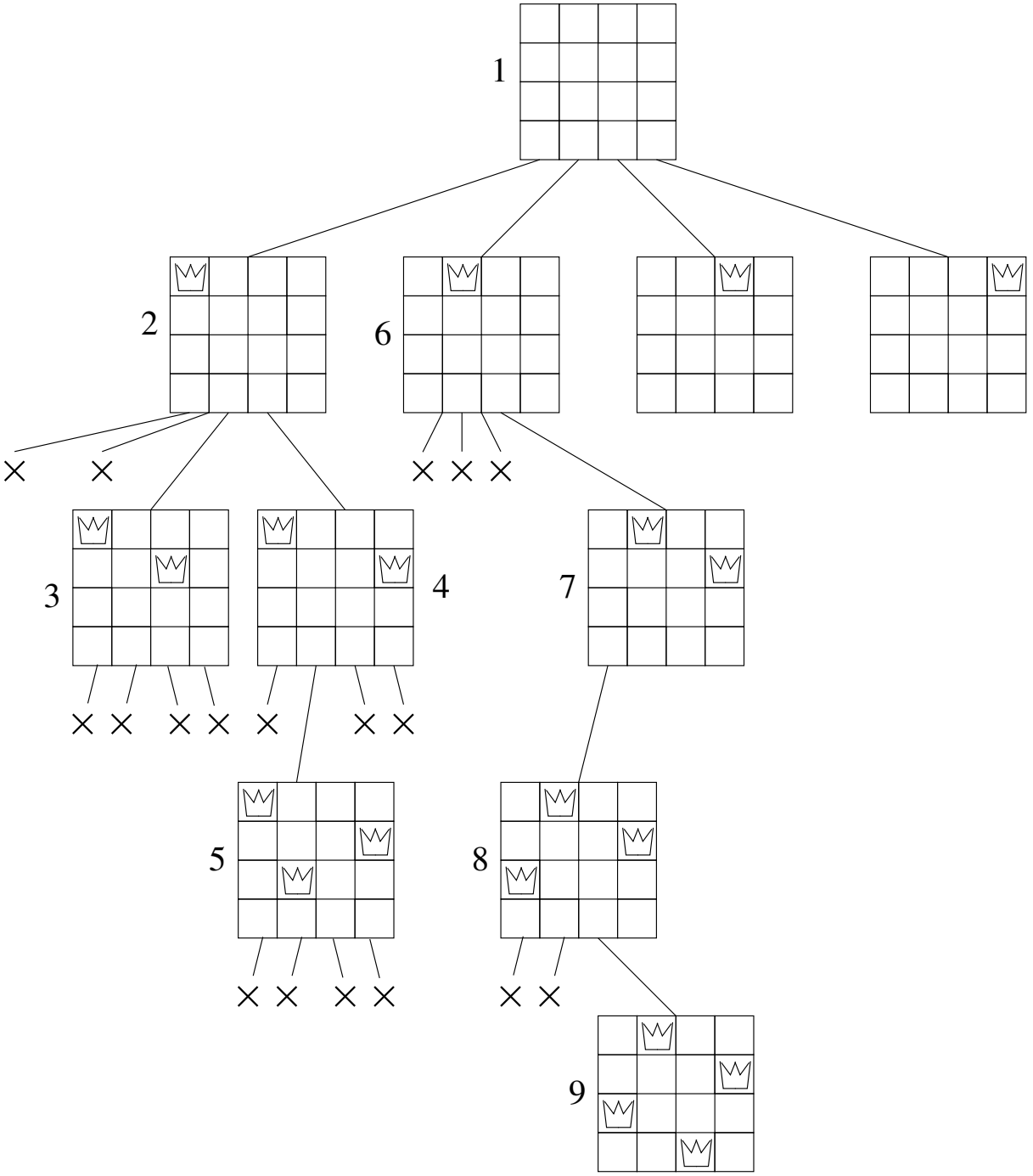
- Seuraavaksi tarkastellaan miten kuningattaret voidaan asettaa riville 2. Aloitetaan vasemmanpuoleisesta 1 rivin valinnasta



- Huomaamme että olemme muodostamassa puuta, joka kuvaa erilaisia ratkaisumahdollisuuksia

- Kaksi vasemmanpuoleisinta yritystä ovat tuhoon tuomittuja, eikä enää kannata tutkia mitä niissä haaroissa tapahtuu

- Seuraavalla sivulla ratkaisun löytymiseen asti piirretty ratkaisupuu


1
2
6
3
4
7
5
8
9

- Kuvassa puun solmut on numeroitu esijärjestyksessä, ja yhdeksäs solmu on siis ratkaisua vastaava pelitilanne
- Kun laudan koko n kasvaa, tulee puusta varsin suuri
- Huomionarvoista on kuitenkin se että koko puun ei tarvitse olla talletettuna muistiin
- Itseasiassa riittää että muistissa on ainoastaan reitti juuresta parhaillaan tutkittavaan solmuun

- Voimme etsiä ratkaisun n:n kuningattaren ongelmaan suorittamalla ratkaisupuun läpikäynnin esijärjestyksessä ilman että ratkaisupuuta on missään vaiheessa olemassa

- Talletetaan pelitilanne $n \times n$ -taulukkoon:
  - oletetaan että pelilautaa esittää $n \times n$ -taulukko $table$
  - jos pelilaudan kohdassa (x, y) on kuningatar, on $table[x, y] =$ true
  - muuten $table[x, y] =$ false

- Oletetaan että käytössä on metodi **check**($table$)
  - metodi palauttaa true jos sen parametrina saama pelitilanne on mahdollinen ratkaisu tai voidaan vielä täydentää ratkaisuksi
  - jos pelilaudalla on toisiaan uhkaavia kuningattaria, operaatio palauttaa false

- Valitaan ensin vakio n, eli pelilaudan koko on $n \times n$

- Aluksi laitetaan $n \times n$ taulukon $table$ kaikkien ruutujen arvoksi false, ja kutsutaan **putqueen**($table$,1)

- **putqueen**(table,row)

```
1   if check(table) == false
2       return
3   if row == n+1
4       print(table)
5       return
6   for x = 1 to n
7       table2 = luoKopio( table )
8       table2[x,row] = true
9       putqueen(table2,row+1)
```

- Operaation toiminta parametreilla ($table$, row):
  - operaatio tarkastaa ensin (rivi 1) edustaako $table$ pelilautaa joka voi johtaa ratkaisuun tai on jo ratkaisu (rivi 3)
  - jos kyseessä on ratkaisu, tulostetaan pelilauta (rivit 3-5)
  - muussa tapauksessa tutkitaan kaikki tavat asettaa kuningatar riville row
  - luodaan uusi asetelma tauluun $table2$ ja rekursiivinen kutsu (rivi 9) tarkastaa johtaako tämä asetelma ratkaisuun

- Algoritmi käy läpi puun mikä ei ole missään vaiheessa rakennettuna muistiin; tällaista puuta sanotaan implisiittiseksi puuksi

- Jos puu olisi kokonaan muistissa, olisi sen koko valtava: $1 + n + n^2 + n^3 + \ldots + n^n$

- Koska nyt muistissa on korkeintaan puun korkeudellinen (eli n kpl) solmuja, on tilavaativuus $\mathcal{O}(n^3)$, sillä jokainen rekursiokutsu vaatii tilaa shakkilaudan verran eli $\mathcal{O}(n^2)$, tilavaativuus ei siis ole kohtuuton

- Aikavaativuus sen sijaan on suuri, sillä vaikka kaikkia solmuja ei tarvisekaan käydä läpi, kasvaa läpikäytävien solmujen määrä kuitenkin eksponentiaalisesti n:n suhteen

- Tälläisestä implisiittisen puun läpikäyntimenetelmästä käytetään nimitystä peruuttava etsintä (engl. backtracking): umpikujaan jouduttaessa palataan puussa sellaiseen ylempään solmuun, johon vielä liittyy kokeilemattomia vaihtoehtoja

- Esitetty algoritmi kuljettaa muodostettavaa kuningatarasetelmaa rekursiivisten kutsujen parametrina
- Taulukko kopioidaan jokaisen rekursiivisen kutsun yhteydessä rivillä 7
- $n \times n$-kokoisen taulukon kopiointiin kuluu aikaa $\mathcal{O}(n^2)$, eli jokainen funktion rungon suoritus kuluttaa taulukkojen kopiointiin aikaa n kertaa $\mathcal{O}(n^2)$, eli $\mathcal{O}(n^3)$
- Taulukon kuljettaminen parametrina ei ole oikeastaan tarpeen: riittää että pidetään rakennuksen alla oleva kuningatarasetelma globaalina muuttujana olevassa taulukossa
- Oletetaan, että *table* on kuten edellä, mutta kyseessä on globaali muuttuja, kuningatarasetelma löytyy kutsumalla seuraavaa funktiota parametrilla 1:

```
putqueenv2(row)
1   if check(table) == false
2       return
3   if row == n+1
4       print(table)
5       return
6   for x = 1 to n
7       table[x,row] = true
8       putqueenv2(row+1)
9       table[x,row] = false
```

- Funktiorungon yhden suorituksen aikavaativuus on nyt funktion **check** suoritusaika plus $\mathcal{O}(n)$

- Algoritmin kokonaisaikavaativuus on siis solmujen lukumäärä kertaa funktiorungon suoritusaika

- Algoritmin tilavaativuus pienenee, sillä edellisessä versiossa jokainen rekursiivinen kutsu talletti oman kopionsa asetelmasta ja vei tilaa $\mathcal{O}(n^2)$ ja koska rekursiivisia kutsuja voi olla kerrallaan menossa n kpl tilavaativuus oli $\mathcal{O}(n^3)$

- Uudessa versiossa jokainen rekursiokutsu vie tilaa ainoastaan vakion verran, eli koko algoritmin tilavaativuus on $\mathcal{O}(n)$

- Globaalien muuttujien käyttöä ei yleisesti pidetä kovin hyvänä ideana

- Jos käytössä on olio-ohjelmointikieli, voidaan "globaali" muuttuja esittää myös ohjelmistoteknisessä mielessä tyylikkäästi tekemällä globaalisti saatavilla olevasta datasta olion attribuutti

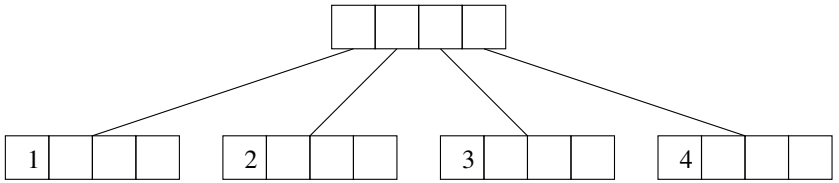
- Java-luonnos ratkaisusta seuraavalla sivulla

```
public class Queens{
   boolean[][] table;
   int n;

   public Queens(int n){ this.n = n; this.table = new boolean[n]; ... }

   private boolean check(){ ... } // ei parametria sillä näkee attribuutin table

   public void putQueen(int row){
       if ( check( ) == false ) return;
       if ( row = n+1 ) { tulosta( this.table ); return; }
       for ( int x=1; x<=n; x++ ) {
           this.table[row][x] = true;
           putQueen(row+1);
           this.table[row][x] = false;
       }
   }
}

public class PaaOhjelma{
   public static void main(String[] args) {
       Queens ongelma = new Queens(8);
       ongelma.putQueen(1);
   }
}
```

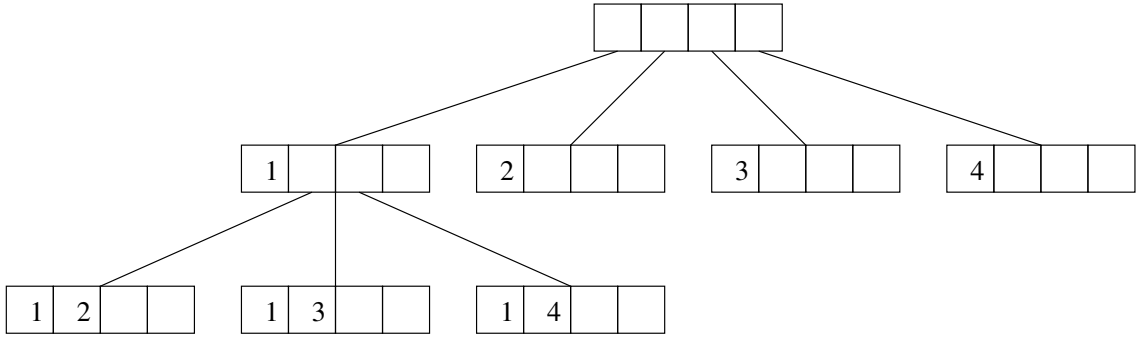
Permutaatioiden generoiminen

- Samaa ratkaisustrategiaa voimme käyttää myös seuraavaan ongelmaan: miten voidaan generoida lukujen $1, 2, \ldots, n$ kaikki permutaatiot?

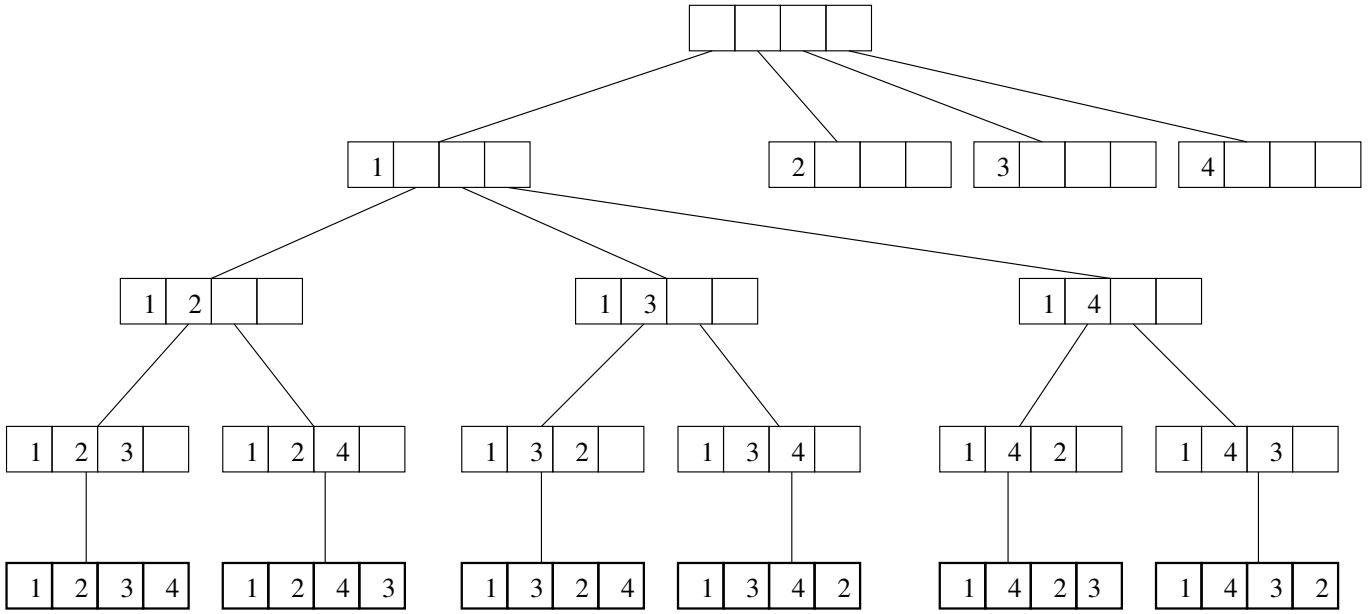
- Tarkastellaan permutaatioita luvuille $1, 2, 3, 4$
  - permutaation ensimmäinen luku voi alkaa mikä tahansa yo. luvuista:



  - vasen haara jatkuisi siten että seuraava numero voi olla joku joukosta $2, 3, 4$; luku 1 on jo käytetty sillä se aloittaa permutaation



- Seuraavassa permutaatiopuu hieman pitemmälle piirrettynä



- Valmiit permutaation löytyvät siis puun lehdistä, ja jos lehdet generoidaan esijärjestyksessä saadaan permutaatiot suuruusjärjestyksessä

- Algoritmi permutaatioiden generoimiseen
  - alustetaan n-paikkainen totuusarvoinen taulukko $used$ siten että jokaisen alkion arvo on false
  - $used$-taulukko kertoo mitkä luvuista on jo käytetty permutaatiossa
  - oletetaan että $table$ on n-paikkainen taulukko minkä alkiot ovat tyyppiä `int`
  - kutsutaan **generate**($table$,$used$,1)

```
generate(table,used,k)
1   if k == n+1 print(table)
2   else for i = 1 to n
3            if used[i] == false
4                 used2 = luoKopio( used )
5                 used2[i] = true
6                 table[k] = i
7                 generate(table, used2, k+1)
```

- Algoritmin toimintaidea
  - rivillä 1 tarkistetaan onko permutaatio jo generoitu, jos on niin permutaatio tulostetaan
  - jos permutaatio ei ole vielä valmis, niin jatketaan permutaatiota kaikilla luvuilla jotka eivät vielä ole käytettyjä (rivit 2-3)
  - riveillä 4-6 käyttämätön luku lisätään permutaatioon, merkataan luku käytetyksi (taulukkoon $used2$) ja rekursiivinen kutsu (rivillä 7) jatkaa kyseistä haaraa alaspäin

- Erona kuningatar-ongelmaan siis tällä kertaa on se että puun generoimista ei lopeteta missään vaiheessa sillä haluamme tulostaa kaikki permutaatiot

- Edelleen tilavaativuus on varsin kohtuullinen, $\mathcal{O}(n^2)$, sillä yhden rekursiotason viemä tila on $\mathcal{O}(n)$ ja rekursiotasoja on n kpl

- Puun solmumäärän yläraja on $1 + n + n^2 + \ldots + n^n = \mathcal{O}(n^n)$, yhden solmun käsittely vie aikaa $\mathcal{O}(n^2)$, joten algoritmin aikavaativuus on $\mathcal{O}(n^{n+2})$

- Samoin kuin kuningatarongelmassa, ei nytkään ole välttämätöntä kuljettaa taulukkoa funktiokutsujen parametrina

- Muuttamalla taulukot $table$ ja $used$ globaaleiksi muuttujiksi, saadaan yhden solmun käsittelyaika lineaariseksi ja koko algoritmin aikavaativuus on $\mathcal{O}(n^{n+1})$

- Edellisellä sivulla todetaan solmujen lukumäärän ylärajan $1 + n + n^2 + \ldots + n^n$ olevan kertaluokkaa $\mathcal{O}(n^n)$, tämä ei ole välttämättä täysin ilmeistä joten perustellaan miksi on näin

- Todetaan ensin, että kun $n \geq 1$, pätee

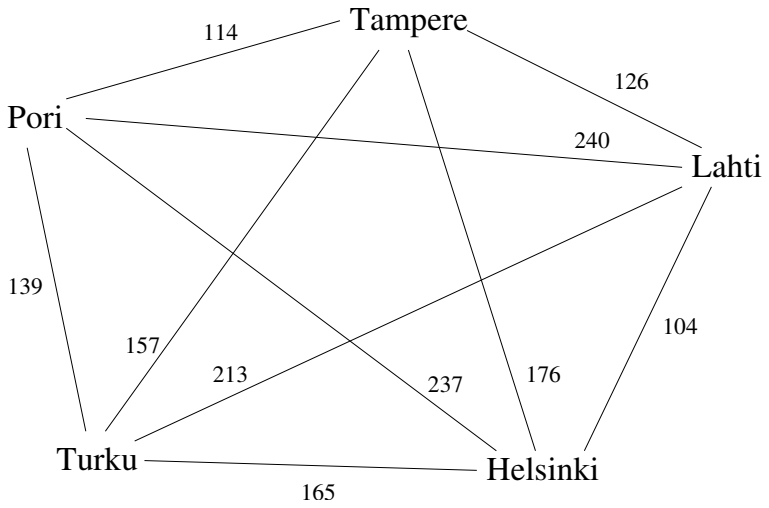
$$\begin{aligned} 1 + n + n^2 + \ldots + n^{n-1} &\leq \underbrace{n^{n-1} + \ldots + n^{n-1}}_{n \text{ kpl}} \\ &= n \cdot n^{n-1} \\ &= n^n. \end{aligned}$$

  Siis

$$1 + n + n^2 + \ldots + n^{n-1} + n^n \leq n^n + n^n = 2n^n.$$

Kauppamatkustajan ongelma

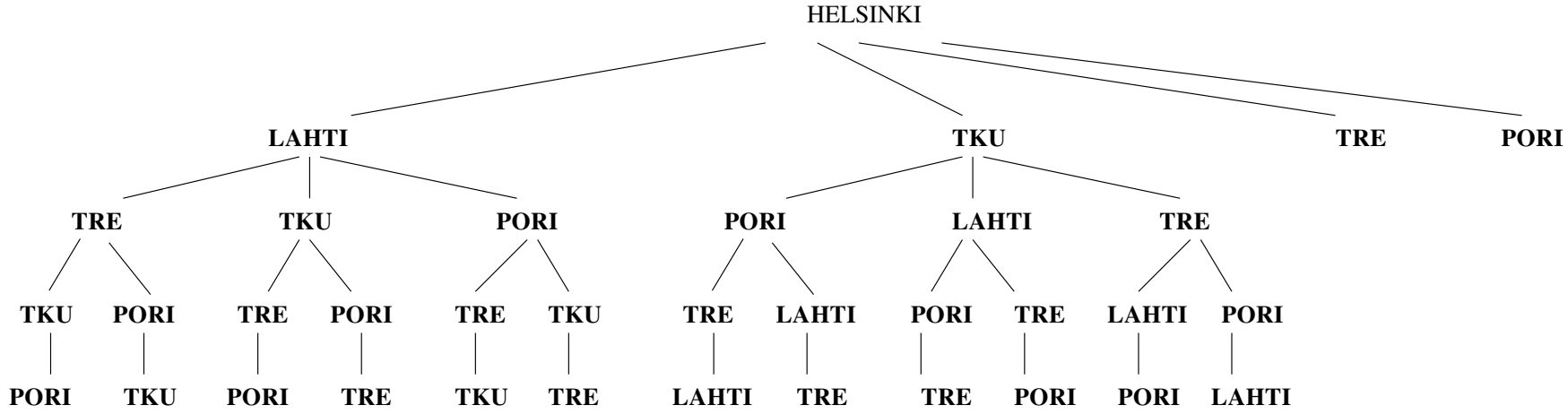
- Helsingissä asuvan kauppamatkustajan täytyy vierailla Lahdessa, Turussa, Porissa ja Tampereella



- Kulujen minimointi bisneksessä on tärkeää: mikä on lyhin reitti joka alkaa Helsingistä ja päättyy Helsinkiin ja sisältää yhden vierailun kussakin kaupungissa?

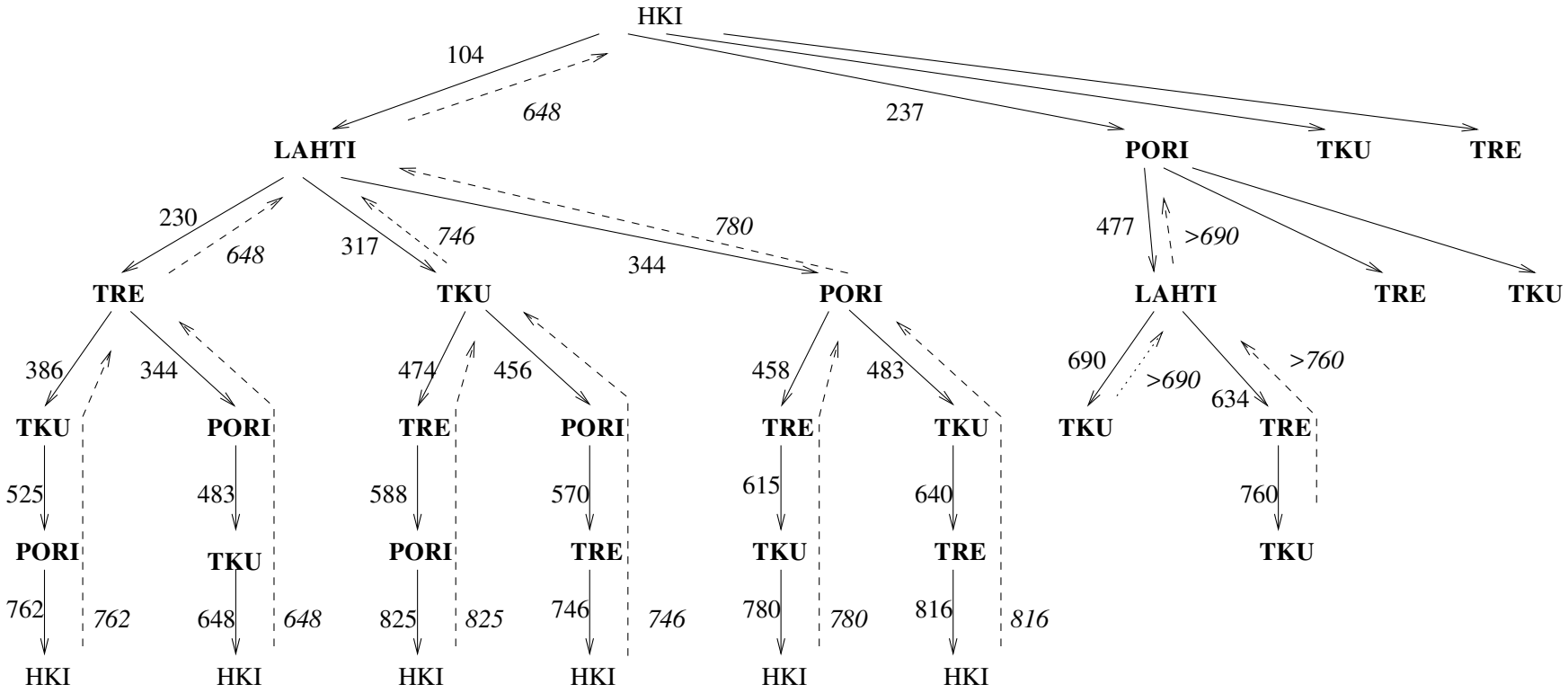
- Huomaamme että mahdolliset reitit ovat kaupunkien jonon Turku, Tampere, Pori, Lahti permutaatiot

- Voimme siis käyttää ratkaisussa samaa periaatetta kuin permutaatioiden tulostuksessa:



- Näin siis saamme systemaattisesti generoiduksi kaikki mahdolliset reitit

- Reitin pituus kannattaa laskea heti generoinnin yhteydessä:



- Palatessamme etsintäpuuta ylöspäin muistamme mikä oli parhaan kyseistä kautta kulkevan reitin pituus
- Uutta reittiä etsittäessä ei kannata enää jatkaa jos tiedämme että kyseinen reitti tulee joka tapauksessa olemaan pitempi kuin paras aiemmin löydetty reitti esim. kuvassa Helsinki $\rightarrow$ Pori $\rightarrow$ Lahti $\rightarrow$ Turku
- Koko etsintäpuun läpikäytyämme saamme tietoon lyhimmän reitin pituuden, samalla toki kannattaa merkitä muistiin minkä kaupunkien kautta reitti kulkee

- Algoritmihahmotelma
  - oletetaan että kaupunkeja on n kappaletta, Helsinki on kaupunki numero 1
  - kaksiulotteinen taulukko $dist$ kertoo kaupunkien välimatkat, esim. $dist$[1,3] sisältää Helsingin ja kaupungin numero 3 välimatkan
  - n-paikkainen totuusarvoinen taulukko $visited$ kertoo missä kaupungeissa on jo vierailtu tutkittavalla polulla
  - alustetaan $visited[i]$ = false jokaiselle i:lle
  - tulos kerätään globaaliin muuttujaan $best$ joka alustetaan arvolla ∞
  - kutsutaan **tsp**(0,$visited$,1,1)

```
tsp(length,visited,current,k)
1   if k == n
2       if length + dist[current,1] < best
3           best = length + dist[current,1]
4       return
5   for i = 2 to n
6       if visited[i] == false and length + dist[current,i] < best
7           visited2 = luoKopio( visited )
8           visited2[i] = true
9           tsp(length + dist[current,i],visited2,i,k + 1)
10  return
```

- Algoritmin toimintaidea
  - parametrit:
    $best$ parhaan jo löydetyn reitin pituus
    $length$ kuinka pitkä reitin tähän asti tutkittu osa on
    $visited$ missä kaupungeissa on jo käyty
    $current$ nykyisen kaupungin numero
    k kuinka monessa kaupungissa on jo käyty
  - rivillä 1 huomataan jos koko reitti on jo generoitu ja se on lyhempi kuin paras reitti tähän mennessä: palautetaan tässä tapauksessa reitin pituus (tähän asti käydyn osan pituus + matka Helsinkiin)
  - jos reitti ei ole vielä valmis, niin jatketaan reittiä kaikilla kaupungeilla joissa ei vielä ole käyty (rivit 5-10)
  - rekursiokutsu rivillä 9 tutkii mikä on kaupungilla i jatkuvan reitin pituus
  - uutta reittiä ei tutkita jos se on jo tässä vaiheessa toivottoman pitkä
  - Huomautus: oikeastaan $visited$[1] ei tule käyttöön ollenkaan

- Käytetty ongelmanratkaisutekniikka muistuttaa läheisesti kuningatarongelmassa käytettyä peruuttavaa etsintää jossa peruutetaan kun törmätään puun haarassa umpikujaan

- Nyt toimimme hieman kehittyneemmin, eli jos huomataan, että joku puun haara ei voi johtaa parempaan ratkaisuun kuin tunnettu paras ratkaisu, jätetään haara tutkimatta.

- Menetelmä kulkee nimellä branch-and-bound

- Tilavaativuus kohtuullinen $\mathcal{O}(n^2)$ sillä yksi rekursiotaso vie tilaa $\mathcal{O}(n)$

- Aikaa algoritmi vie eksponentiaalisesti tutkittavien kaupunkien määrään nähden

- Huom: esim. reitti Helsinki $\rightarrow$ Lahti $\rightarrow$ Tampere $\rightarrow$ Pori $\rightarrow$ Turku $\rightarrow$ Helsinki on samanpituinen myös päinvastaiseen suuntaan kuljettuna

- Sama pätee jokaiselle reitille, algoritmimme siis oikeastaan tutkii jokaisen erilaisen reitin kahteen kertaan

- Vaikka optimoisimme algoritmia siten että tämä epäkohta poistuisi, pysyy aikavaativuus silti eksponentiaalisena

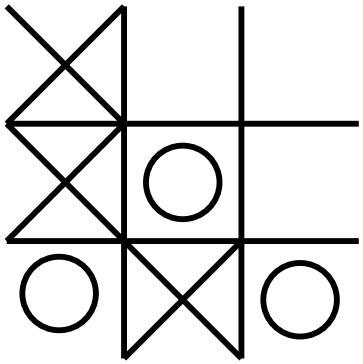
- Kauppamatkustajan ongelmalle ei tiedetä parempia kuin eksponentiaalisessa ajassa toimivia ratkaisualgoritmeja

- Toisaalta ei ole pystytty todistamaan ettei nopeaa (polynomisessa ajassa toimivaa) algoritmia ole olemassa . . .

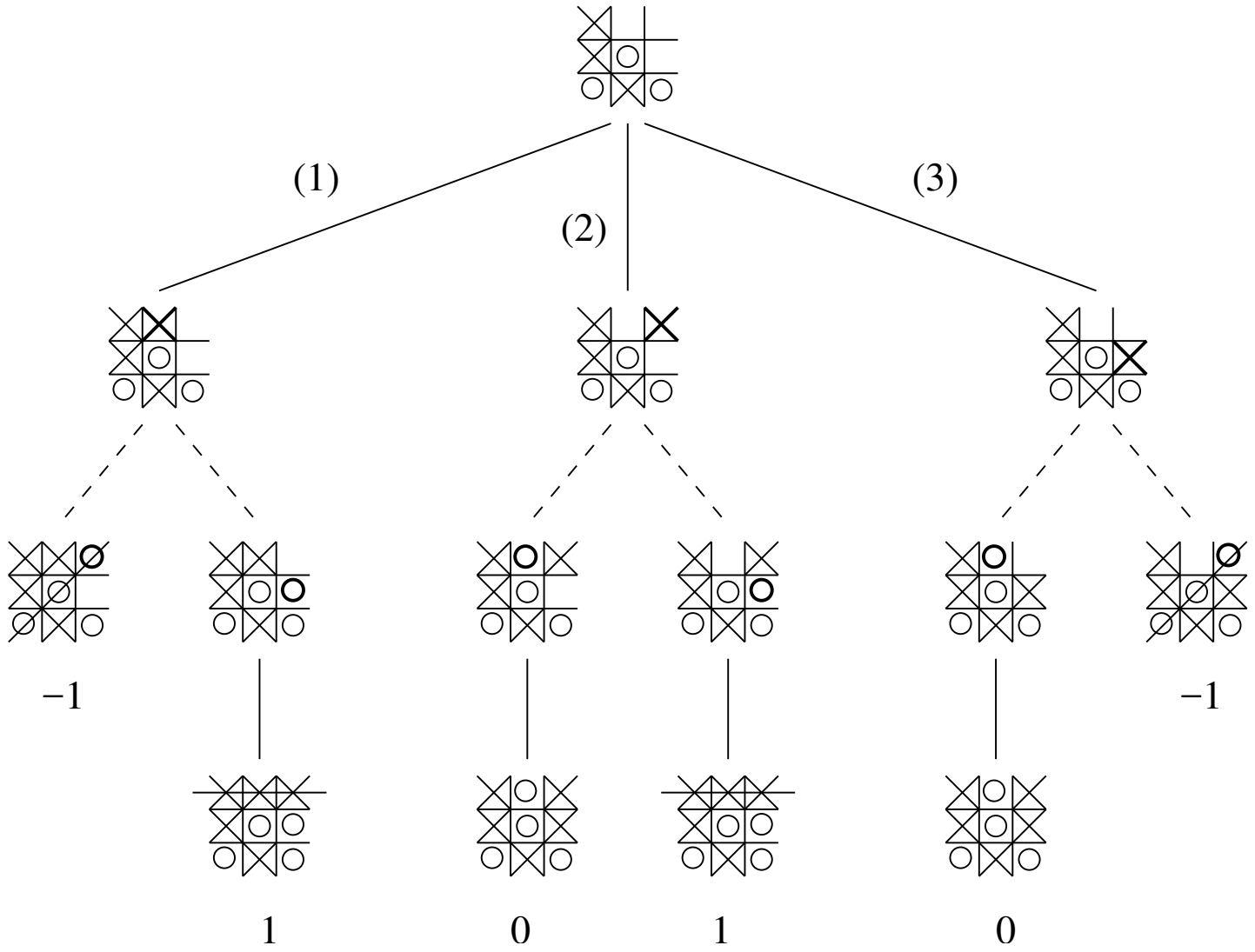
- Kyseessä on ns. NP-täydellinen ongelma, aiheesta hieman enemmän kurssilla Laskennan mallit

Pelipuu

- Tietokone pelaa risti-nollaa ihmistä vastaan
- On ristin vuoro, mitä tietokoneen kannattaa tehdä seuraavassa tilanteessa?



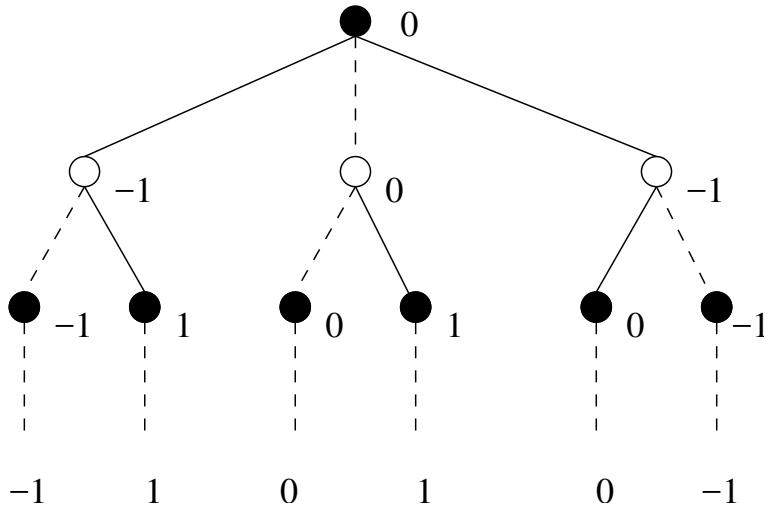
- Tietokone rakentaa päätöksensä tueksi pelipuun, ks. seuraava sivu


(1)
(2)
(3)
risti
nolla
risti
−1
−1
1
0
1
0

- On siis tehtävä valinta kolmen mahdollisen siirron välillä
- Pelipuuhun on kirjattu auki myös kaikki mahdolliset nollaa pelaavan siirrot, eli miten nolla voisi vastata kunkin ristin siirron jälkeen
- Ja edelleen, miten peli voisi jatkua kahden siirtovuoron jälkeen
- Lopputilanteita vastaaviin pelipuun lehtisolmuihin on merkattu tilanteen arvo ristin kannalta: voitto 1, tasapeli 0 ja tappio -1
- Siis minkä siirron tietokone tekee?
  - valinta (1) johtaa lopulta joko nollan tai ristin voittoon
  - valinta (2) johtaa joko tasapeliin tai ristin voittoon
  - valinta (3) johtaa joko tasapeliin tai nollan voittoon

  eli järkevintä valita siirto (2), voittomahdollisuus jää mutta on varmaa ettei ainakaan hävitä
- Strategiana on valita parhaan arvon (voitto 1, tasapeli 0, tappio -1) tuottava haara siten että oletetaan että vastustaja pelaa mahdollisimman hyvin

- Seuraavassa pelipuu piirrettynä hiukan abstraktimmin



- Ristin vuoroa vastaavat solmut ovat mustia ja nollan vuoroa vastaavat valkoisia
- Pelipuu evaluoidaan lähtien lehdistä edeten juureen
  - mustat solmut ovat max-solmuja, ne saavat arvokseen lapsen jolla suurin arvo
  - valkoiset solmut ovat min-solmuja, saaden arvokseen lapsen jolla pienin arvo
- Ristin siirtoa vastaa se lapsi minkä arvon juuren max-solmu perii
- Kuten edellisissä esimerkissämme, ei nytkään ole tarvetta luoda pelipuuta eksplisiittisesti muistiin, riittää generoida yksi polku kerrallaan

- Seuraavassa rekursiiviset operaatiot suorittavat pelipuun evaluoinnin, aluksi kutsutaan operaatiota risti parametrina meneillään olevaa pelitilannetta vastaava solmu

```
risti(v)
1   if v:llä ei lapsia tai peli jo ohi
2       if ristillä kolmen suora return 1
3       if nollalla kolmen suora return -1
4       return 0
5   mybest = −∞
6   for kaikilla v:n lapsilla w eli tilanteilla joihin nykyisestä asetelmasta päästään
7       newval = nolla(w)
8       if newval > mybest mybest = newval
9   return mybest
```

```
nolla(v)
1   if v:llä ei lapsia tai peli jo ohi
2       if ristillä kolmen suora return 1
3       if nollalla kolmen suora return -1
4       return 0
5   myworst = ∞
6   for kaikilla v:n lapsilla w eli tilanteilla joihin nykyisestä asetelmasta päästään
7       newval = risti(w)
8       if newval < myworst myworst = newval
9   return myworst
```

- Rivillä 1 siis huomaamme jos siirtoja ei enää ole ja palautamme pelitilannetta vastaavan arvon (rivit 2-4)

- Jos peli jatkuu vielä, käymme läpi kaikki mahdolliset siirrot (rivi 6) ja evaluoimme miten peli etenee tämän siirron seurauksena (rivi 7)

- **risti**-operaatio palauttaa parhaan lapsensa arvon ja **nolla** palauttaa huonoimman lapsensa arvon

- Esitetty pelipuun evaluointimenetelmä kulkee kirjallisuudessa nimellä min-max-algoritmi

- Risti-nollassa pelipuut ovat vielä kohtuullisen kokoisia, eli siirron valinta vie kohtuullisen ajan (huom: jotkut puun haarat ovat symmetrisiä eikä "samanlaisista" tarvitse tutkia kuin yksi vaihtoehto)

- Useimmissa kiinnostavissa peleissä, esim. shakissa, tilanne on aivan toinen, pelipuut ovat niin suuria, että niiden läpikäynti kokonaisuudessaan on mahdotonta

- Tällöin paras mitä voidaan tehdä, on generoida pelitilanteita tiettyyn syvyyteen asti

- Jos pelipuuta ei voida rakentaa valmiisiin tilanteisiin (voitto, häviö, tasapeli) asti, ongelmaksi nouseekin se mikä on pelipuun lehtisolmuissa olevien pelitilanteiden arvo

- Tähän on toki mahdollista kehitellä erilaisia arviointitapoja (jäljellä olevat omat/vastustajan nappulat, asetelma laudalla, y.m.)

7. Keko

- Tarkastellaan vielä yhtä tapaa toteuttaa sivulla 165 määritelty tietotyyppi joukko

- Tällä kertaa emme kuitenkaan toteuta normaalia operaatiovalikoimaa, vaan olemme kiinnostuneita ainoastaan kolmesta operaatiosta:

  - **heap-insert**(A, k) lisää joukkoon avaimen k
  - **heap-min**(A) palauttaa joukon pienimmän avaimen arvon
  - **heap-del-min**(A) poistaa ja palauttaa joukosta pienimmän avaimen

- Nämä operaatiot tarjoavaa kekoa sanotaan minimikeoksi (engl. minimum heap)

- Toinen vaihtoehto, eli maksimikeko (engl. maximum heap) tarjoaa operaatiot:

  - **heap-insert**(A, k) lisää joukkoon avaimen k
  - **heap-max**(A) palauttaa joukon suurimman avaimen arvon
  - **heap-del-max**(A) poistaa ja palauttaa joukosta suurimman avaimen

- Näitä kolmea operaatiota sanotaan (maksimi/minimi) keko-operaatioiksi

- Keskitymme tässä luvussa ensisijaisesti maksimikekoon ja sen toteutukseen

- Huom: Keoksi kutsutaan myös muistialuetta, josta suoritettavien ohjelmien ajonaikainen muistinvaraaminen tapahtuu. Tällä tietojenkäsittelyn "toisella" keolla ei ole mitään tekemistä tietorakenteen keko kanssa

- Pystyisimme luonnollisesti toteuttamaan operaatiot käyttäen jo tuntemiamme tietorakenteita:
  - käyttämällä järjestämättömän listan **insert**, **delete** ja **max**-operaatioita **heap-insert** olisi vakioaikainen mutta **heap-del-max** veisi aikaa $\mathcal{O}(n)$
  - käyttämällä järjestetyn rengaslistan **insert**, **delete** ja **max**-operaatioita **heap-insert** veisi aikaa $\mathcal{O}(n)$ ja **heap-del-max** olisi $\mathcal{O}(1)$
  - käyttämällä AVL-puun **insert**, **delete** ja **max**-operaatioita sekä **heap-insert** että **heap-del-max** veisivät aikaa $\mathcal{O}(\log n)$

- Seuraavassa esittämämme keko-tietotyypin toteutuksessa sekä **heap-del-max** (tai minimikeossa **heap-del-min**) että **heap-insert** toimivat ajassa $\mathcal{O}(\log n)$ ja **heap-max** (tai minimikeossa **heap-min**) toimii vakioajassa $\mathcal{O}(1)$

- Herää kysymys, mihin tarvitsemme näin spesialisoitunutta tietorakennetta?
- Eikö riitä, että käyttäisimme muokattua tasapainoitettua hakupuuta, sillä näin saavutettu keko-operaatioiden vaativuus olisi $\mathcal{O}$-analyysin mielessä sama kuin kohta esitettävällä varsinaisella kekototeutuksella?
- Tulemme kurssin aikana näkemään algoritmeja, jotka käyttävät aputietorakenteenaan kekoa ja näiden algoritmien tehokkaan toteutuksen kannalta keko-operaatioiden tehokkuus on oleellinen
- Vaikka keko ei tuokaan kertaluokkaparannusta operaatioihin, on se toteutukseltaan hyvin kevyt, ja käytännössä esim. AVL-puuhun perustuvaa toteutusta huomattavasti nopeampi
- Keko on ohjelmoijalle kiitollinen tietorakenne siinä mielessä, että toteutus on hyvin yksinkertainen toisin kuin esim. tasapainoisten hakupuiden toteutukset
- Keosta on erilaisia versioita, kuten binomikeko ja Fibonacci-keko (joita emme käsittele täällä)

- Minimikeon avulla saamme toteutettua tehokkaasti prioriteettijonon:
  - **heap-insert** vie jonottajan jonoon, avain vastaa jonottajan prioriteettia (mitä pienempi numeroarvo sitä korkeampi prioriteetti)
  - **heap-del-min** ottaa jonosta seuraavaksi palveltavaksi korkeimman prioriteetin omaavan jonottajan
- Keon avulla saamme myös toteutettua tehokkaan kekojärjestämisalgoritmin
- Myös verkkoalgoritmien yhteydessä (luvussa 8) löydämme käyttöä keolle

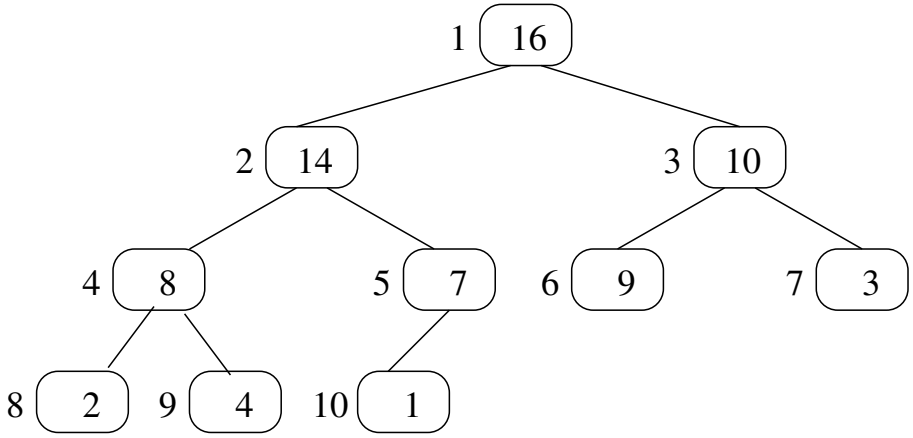
Maksimikeon toteuttaminen

- Keko kannattaa ajatella binääripuuna, joka on talletettu muistiin taulukkona
- Binääripuu on maksimikeko jos

  (K1) kaikki lehdet ovat kahdella vierekkäisellä tasolla k ja $k+1$ siten että tason $k+1$ lehdet ovat niin vasemmalla kuin mahdollista ja kaikilla k:ta ylempien tasojen solmuilla on kaksi lasta

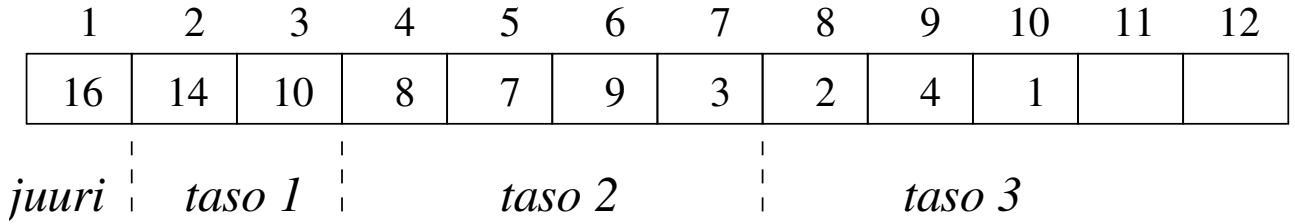
  (K2) jokaiseen solmuun talletettu arvo on suurempi tai yhtäsuuri kuin solmun lapsiin talletetut arvot

- Seuraava binääripuu on maksimikeko (solmun vasemmalla puolella on sen järjestysnumero)



- Keko-ominaisuuden (k1) ansiosta puu voidaan esittää taulukkona, missä solmut on lueteltu tasoittain vasemmalta oikealle

- Yllä oleva puu taulukkoesityksenä:



- Käytännössä keko kannattaa aina tallentaa taulukkoa käyttäen

- Kekotaulukkoon A liittyy kaksi attribuuttia
  - $A.length$ kertoo taulukon koon
  - $A.heap\text{-}size$ kertoo montako taulukon paikkaa (alusta alkaen) kuuluu kekoon
  - Huom: Taulukossa A voi siis kohdan $heap\text{-}size$ jälkeen olla "kekoon kuulumattomia" lukuja

- Keon juurialkio on talletettu taulukon ensimmäiseen paikkaan $A[1]$

- Taulukon kohtaan i talletetun solmun vanhemman sekä lapset tallettavat taulukon indeksit saadaan selville seuraavilla apuoperaatioilla:

  parent(i)
  return $\lfloor i/2 \rfloor$

  left(i)
  return 2i

  right(i)
  return 2i+1

- Vaikka varsinaisia viitteitä ei ole, keossa liikkuminen on todella helppoa
  - tarkastellaan edellisen sivun esimerkkitapausta
  - juuren $A[1]$ vasen lapsi on paikassa $A[2 \cdot 1] = A[2]$ ja oikea lapsi paikassa $A[2 \cdot 1 + 1] = A[3]$
  - solmun $A[5]$ vanhempi on $A[\lfloor 5/2 \rfloor] = 2$, vasen lapsi $A[10]$ mutta koska $\mathit{heap\text{-}size}[A] = 10$, niin oikeaa lasta ei ole

- Voimme lausua kekoehdon (K2) nyt seuraavasti:

  kaikille $1 < i \leq \mathit{heap\text{-}size}$ pätee $A[\mathit{parent}(i)] \geq A[i]$

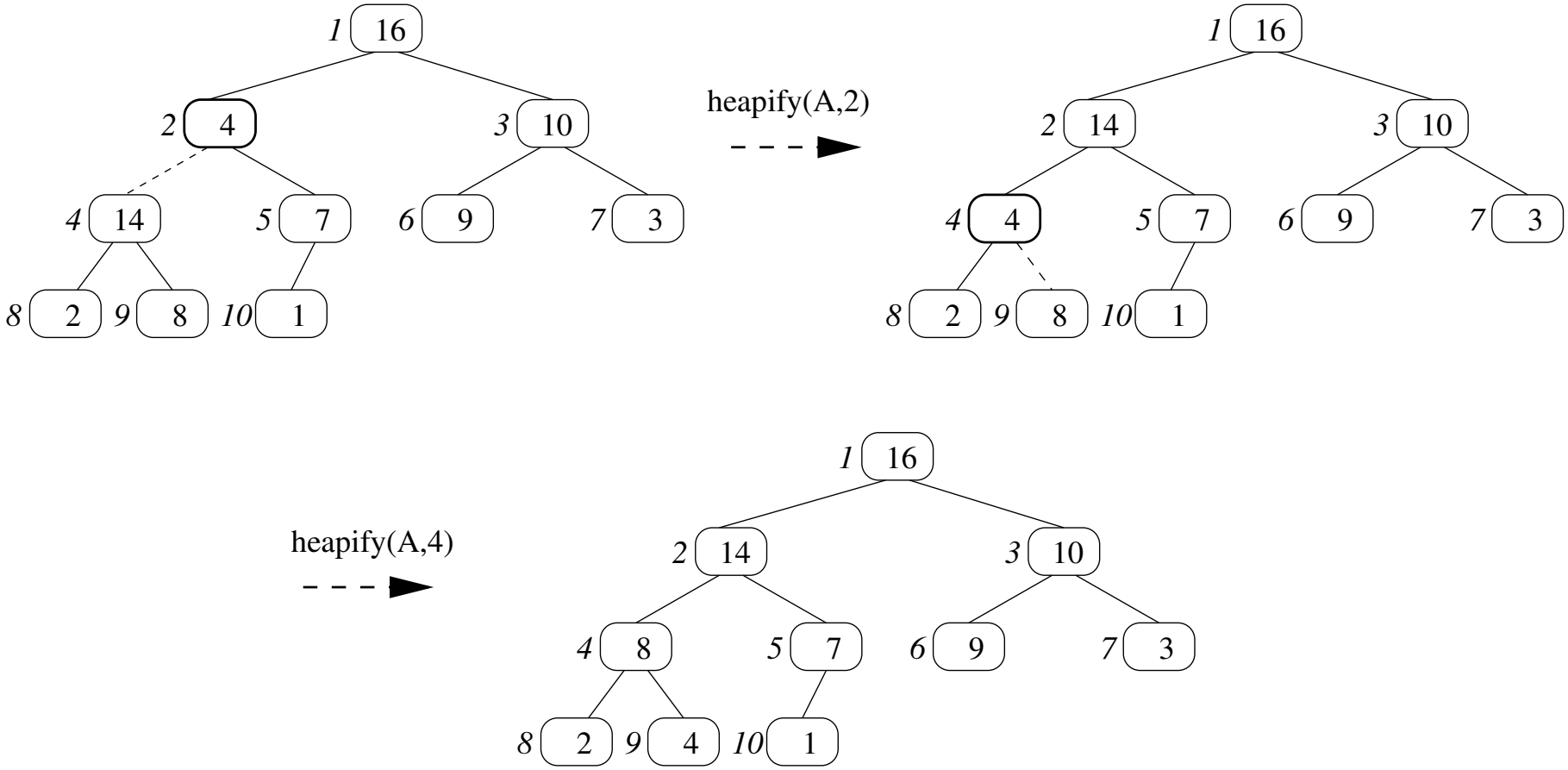
- Ennen varsinaisia keko-operaatioita toteutetaan tärkeä apuoperaatio **heapify**
- Operaatio korjaa kekoehdon, jos se on rikki solmun i kohdalla
- Oletus on, että solmun i vasen ja oikea alipuu toteuttavat kekoehdon
- Kutsun **heapify**(A, i) jälkeen koko solmusta i lähtevä alipuu toteuttaa kekoehdon
- Toimintaperiaate on seuraava:
  - Jos $A[i]$ on pienempi kuin toinen lapsistaan, vaihdetaan se suuremman lapsen kanssa
  - Jatketaan rekursiivisesti alas muuttuneesta lapsesta

- **heapify**:n parametreina ovat taulukko A ja indeksi i
  - oletuksena siis on että $left(i)$ ja $right(i)$ viittaavat jo kekoja olevien alipuiden $A[left(i)]$ ja $A[right(i)]$ juuriin
  - operaatio kuljettaa alkiota $A[i]$ alaspäin, kunnes alipuusta jonka juurena $A[i]$ on tulee keko

```
heapify(A,i)
1   l = left(i)
2   r = right(i)
3   if r ≤ A.heap-size
4       if A[l] > A[r] largest = l
5       else largest = r
6       if A[i] < A[largest]
7           vaihda A[i] ja A[largest]
8           heapify(A,largest)
9   elsif l == A.heap-size and A[i]<A[l]
10          vaihda A[i] ja A[l]
```

- Jos molemmat lapset ovat olemassa (rivi 3), vaihdetaan tarvittaessa $A[i]$:n arvo lapsista suuremman arvoon ja kutsutaan lapselle **heapify**-operaatiota
- Jos vain vasen lapsi on olemassa ja tämän arvo suurempi kuin $A[i]$:n, vaihdetaan arvot keskenään (rivit 9 ja 10)

- Seuraava kuvasarja valottaa operaation toimintaa



- **heapify**-operaation suoritusaika riippuu ainoastaan puun korkeudesta, rekursiivisia kutsuja tehdään pahimmassa tapauksessa puun korkeuden verran

- n-alkioisen keon korkeus selvästi $\mathcal{O}(\log n)$ sillä keko on lähes täydellinen binääripuu

- n alkiota sisältävälle keolle tehdyn **heapify**-operaation pahimman tapauksen aikavaativuus on siis $\mathcal{O}(\log n)$

- Rekursion takia operaation tilavaativuus on $\mathcal{O}(\log n)$

- Operaatio on helppo kirjoittaa myös ilman rekursiota, jolloin se toimii vakiotilassa

- Kekoehdosta (K2) seuraa suoraan että keon maksimialkio on talletettu paikkaan $A[1]$

- Operaatio **heap-max** siis on triviaali ja vie vakioajan

```
heap-max(A)
    return A[1]
```

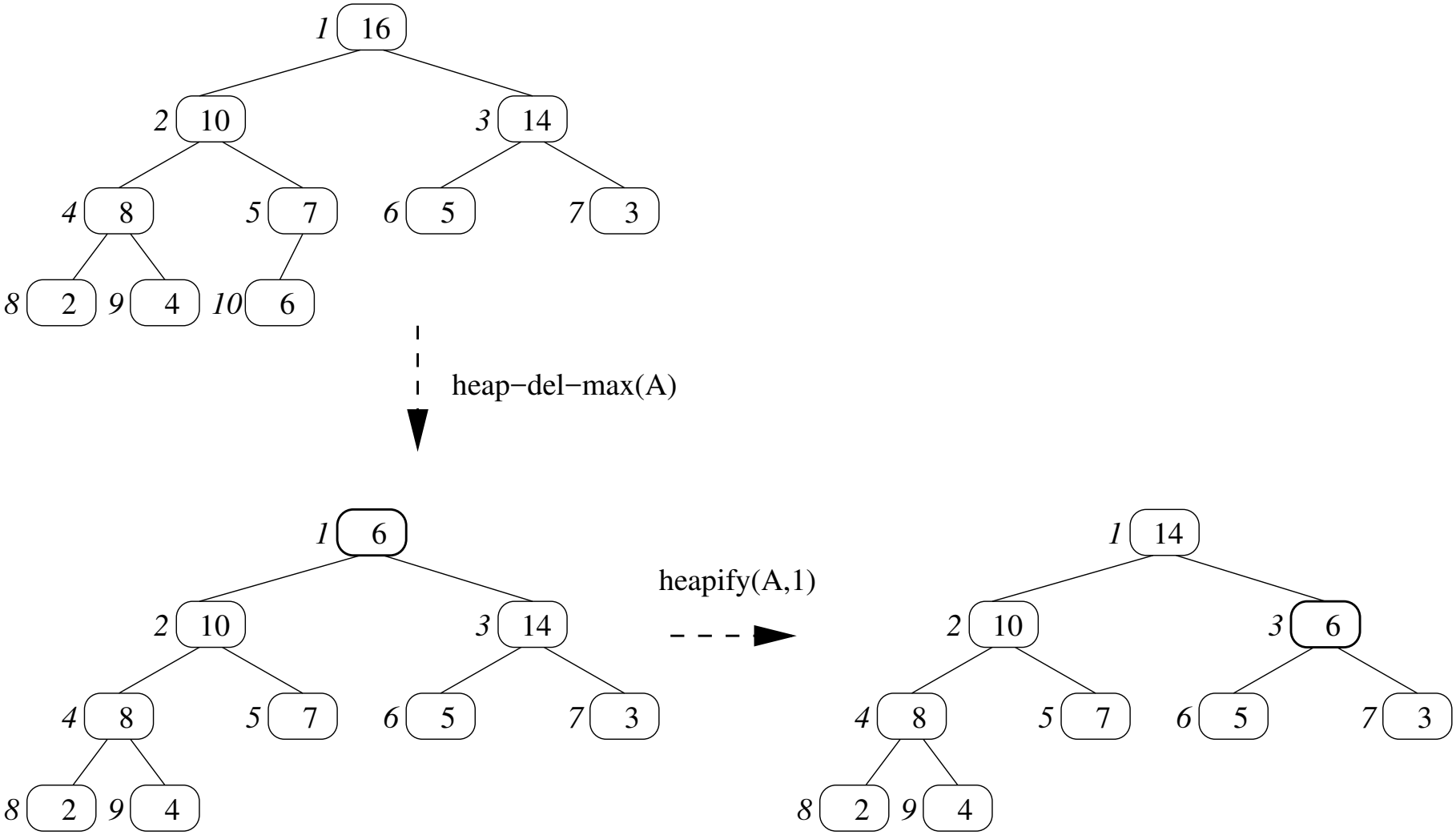
- Alkion poistaminen keosta

  heap-del-max(A)
  1 max = A[1]
  2 A[1] = A[A.heap-size]
  3 A.heap-size = A.heap-size -1
  4 heapify(A,1)
  5 return max

- Toimintaidea
  - operaatio palauttaa kohdassa $A[1]$ olleen avaimen
  - keon viimeisessä paikassa oleva alkio $A[A.\,heap\text{-}size]$ viedään poistetun alkion tilalle ja keon kokoa pienennetään yhdellä (rivit 2 ja 3)
  - keko on muuten kunnossa mutta kohtaan $A[1]$ siirretty avain saattaa rikkoa keko-ominaisuuden, kutsutaan **heapify** operaatiota korjaamaan tilanne

- Operaation aikavaativuus sama kuin **heapify**:llä, eli $\mathcal{O}(\log n)$

- Esimerkki **heap-del-max**-operaation toiminnasta

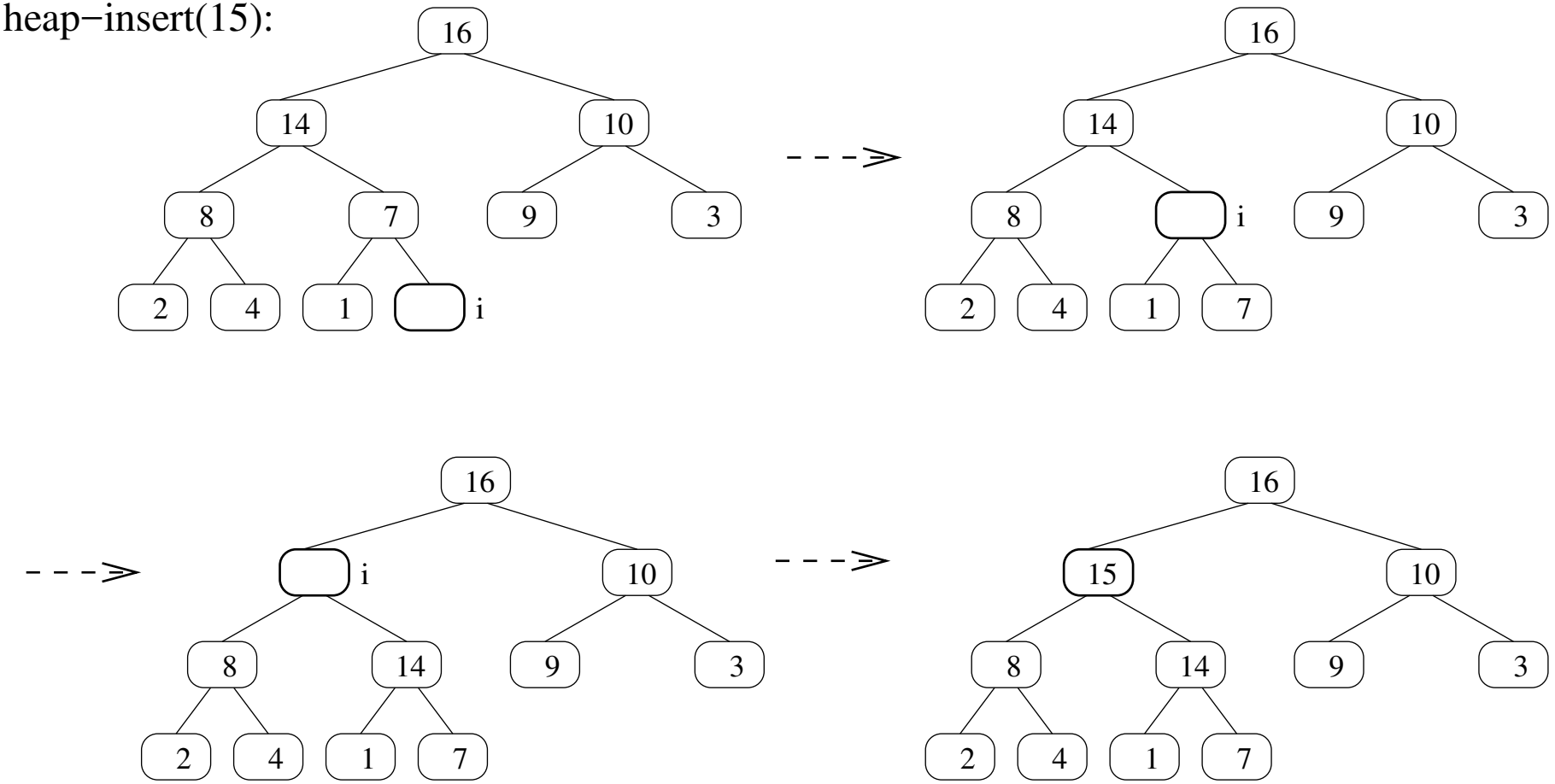


- Alkion lisääminen kekoon

```
heap-insert(A,k)
1   A.heap-size = A.heap-size+1
2   i = A.heap-size
3   while i>1 and A[parent(i)] < k
4       A[i] = A[parent(i)]
5       i = parent(i)
6   A[i] = k
```

- Toimintaidea
  - kasvatetaan keon kokoa yhdellä solmulla eli tehdään paikka uudelle avaimelle
  - kuljetaan nyt keon uudesta solmusta ylöspäin ja siirretään arvoja samaan aikaan yhtä alemmas niin kauan kunnes uudelle alkiolle löydetään paikka joka ei riko keko-ominaisuutta (K2)
- Pahimmassa tapauksessa lisättävä avain viedään puun juureen ja näin käydessä puun korkeudellisen verran avaimia on valutettu alaspäin
- Operaation aikavaativuus siis on $\mathcal{O}(\log n)$
- Seuraavalla sivulla esimerkki operaation toiminnasta

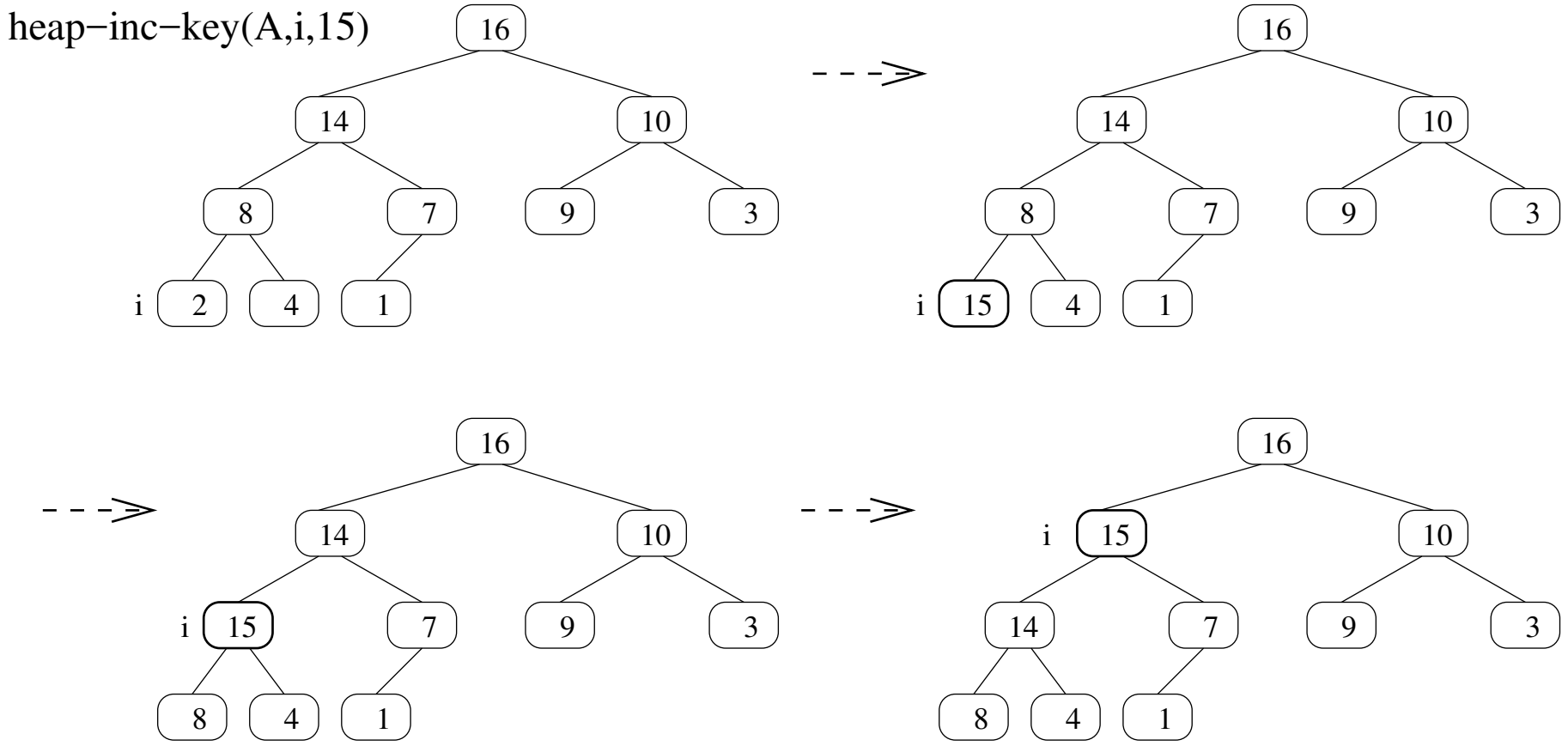
heap−insert(15):



- Jotkut sovellukset tarvitsevat keko-operaatiota, joka kasvattaa annetussa indeksissä olevan avaimen arvoa

```
heap-inc-key(A,i,newk)
1   if newk > A[i]
2       A[i] = newk
3       while i>1 and A[parent(i)] < A[i]
4           vaihda A[i] ja A[parent(i)]
5           i = parent(i)
```

- Toimintaidea
  - jos yritetään pienentää avaimen arvoa, operaatio ei tee mitään
  - kopioidaan keon kohtaan i uusi avaimen arvo (rivi 2)
  - jos kasvatettu avain rikkoo keko-ominaisuuden (K2), vaihdetaan sen arvo vanhemman kanssa niin monta kertaa kunnes oikea paikka löytyy (rivit 3-5)
- Pahimmassa tapauksessa lehdessä olevaa avainta muutetaan ja muutettu avain joudutaan kuljettamaan aina puun juureen saakka
- Operaation aikavaativuus on $\mathcal{O}(\log n)$
- Seuraavalla sivulla esimerkki operaation toiminnasta


heap−inc−key(A,i,15)
16
14
10
8
7
9
3
i
2
4
1
16
14
10
8
7
9
3
i
15
4
1
16
14
10
i
15
7
9
3
8
4
1
16
i
15
10
14
7
9
3
8
4
1

- Vastaavasti voidaan pienentää annetussa indeksissä olevan avaimen arvoa

  heap-dec-key(A,i,newk)
  1 if newk $<$ A[i]
  2 A[i] $=$ newk
  3 heapify(A,i)

- Nyt voidaan siis vaihtaa avainta indeksissä i joko **heap-inc-key**$(A, i, newk)$:lla tai **heap-dec-key**$(A, i, newk)$, sen mukaan onko $newk$ pienempi tai suurempi kuin $A[i]$

Kekojärjestäminen

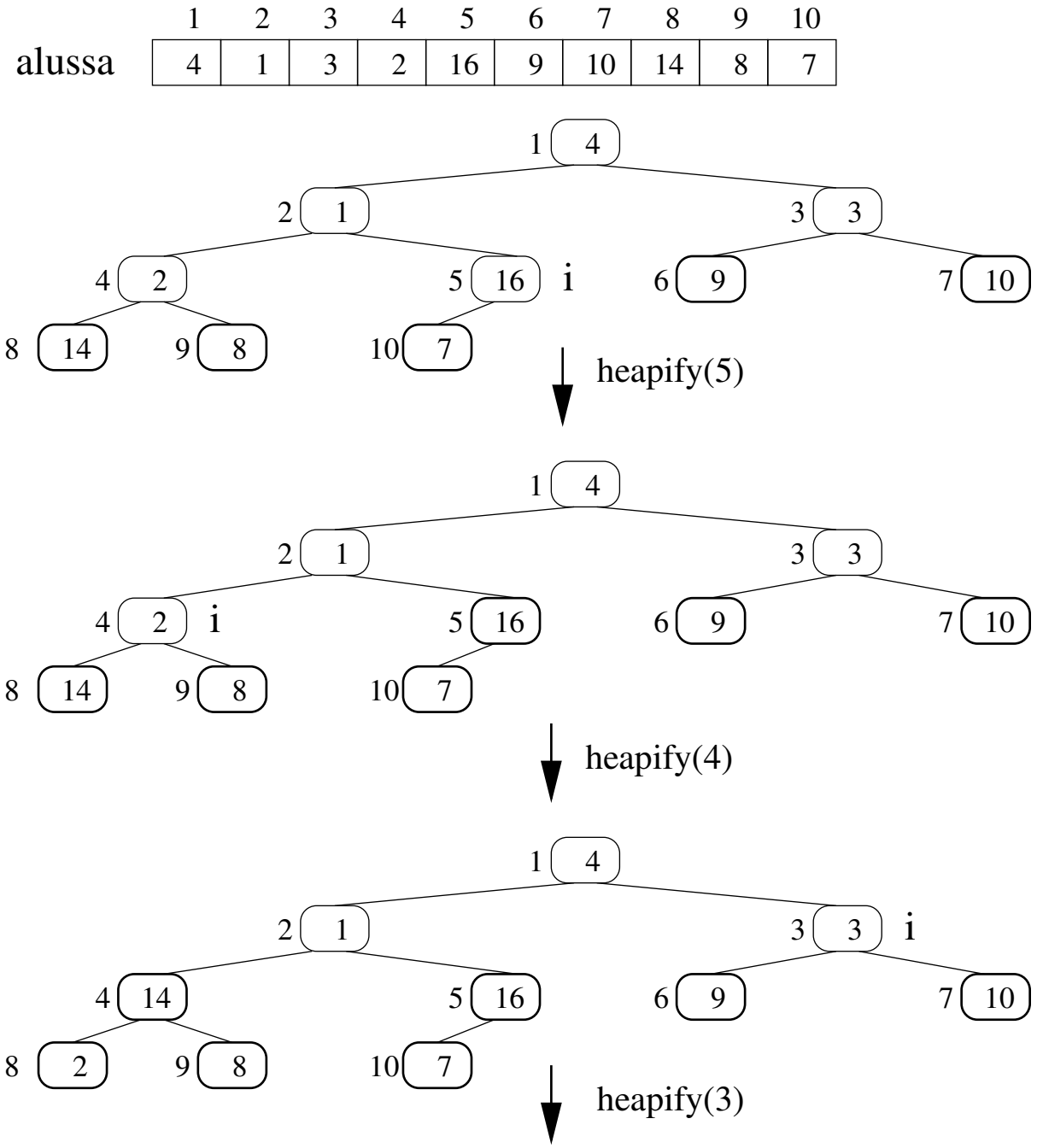
- Oletetaan että A on n-paikkainen kokonaislukutaulukko
- Seuraava algoritmi järjestää A:n alkiot suuruusjärjestykseen käyttäen kekoa H aputietorakenteena

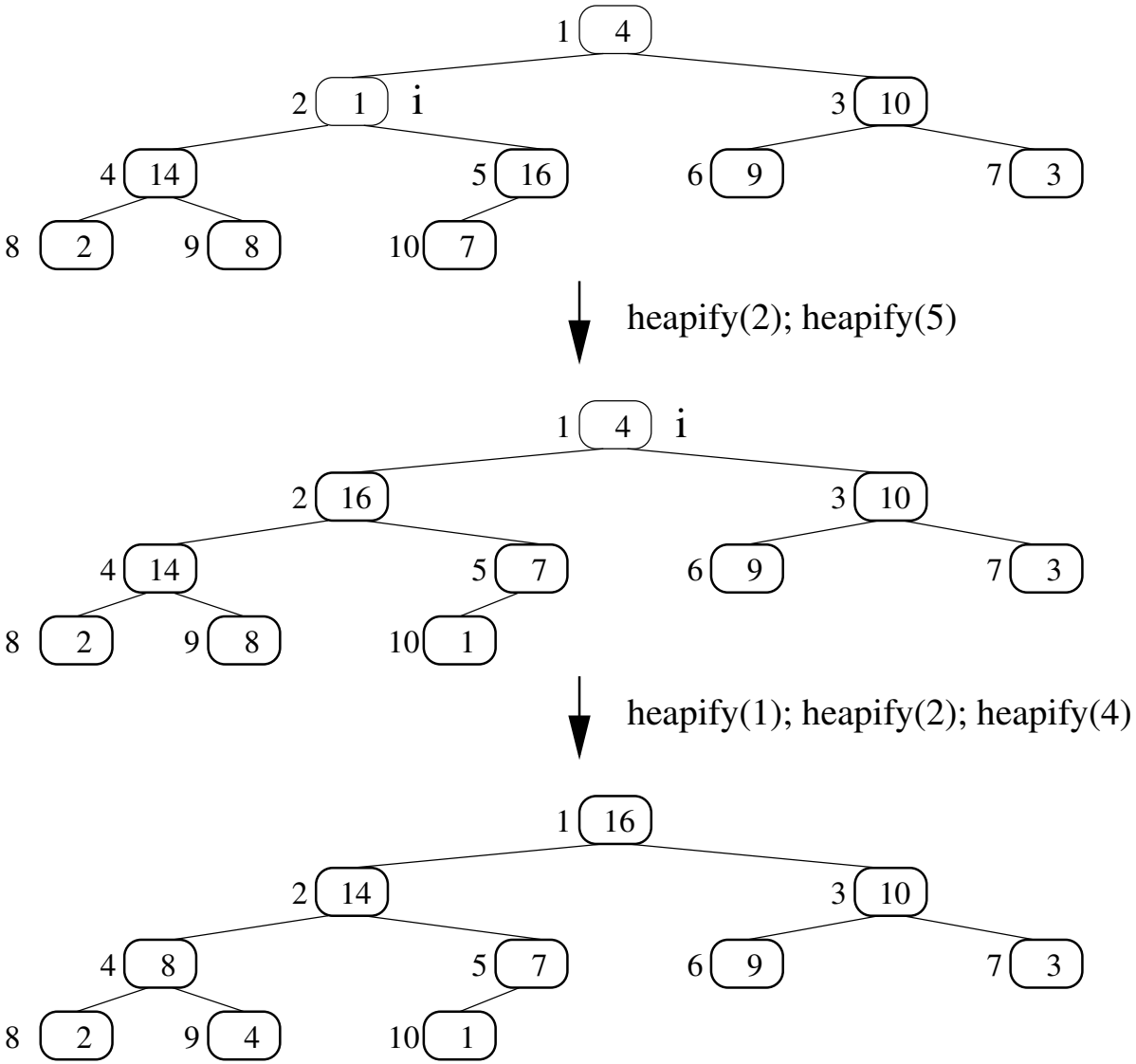
```
sort-with-heap(A,n)
1   for i = 1 to n
2       heap-insert(H,A[i])
3   for i = n downto 1
4       A[i] = heap-del-max(H)
```

- Algoritmin aikavaativuus on $\mathcal{O}(n \log n)$, sillä **heap-insert** ja **heap-del-max** vievät $\mathcal{O}(\log n)$ ja molempia kutsutaan n kertaa

- Mutta voimme toimia fiksummin: operaation **heapify** avulla on helppo rakentaa mistä tahansa taulukosta A keko:
  - lehdet, eli paikossa $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ olevat yhden alkion alipuut ovat jo kekoja
  - kutsutaan **heapify** lapselliselle kekosolmulle alkaen $A[\lfloor n/2 \rfloor]$:sta aina juureen $A[1]$ asti
  - näin taulukko A muuttuu keoksi
  - for-silmukan invariantti on siis: kaikilla $i < j \leq A.\,length$ solmusta j lähtevä alipuu toteuttaa kekoehdon

build-heap(A)
```
1   A.heap-size = A.length
2   for i = ⌊A.length/2⌋ downto 1
3       heapify(A,i)
```


1 2 3 4 5 6 7 8 9 10
alussa
4 1 3 2 16 9 10 14 8 7
1 4
2 1
3 3
4 2
5 16
i
6 9
7 10
8 14
9 8
10 7
heapify(5)
1 4
2 1
3 3
4 2
i
5 16
6 9
7 10
8 14
9 8
10 7
heapify(4)
1 4
2 1
3 3
i
4 14
5 16
6 9
7 10
8 2
9 8
10 7
heapify(3)



tuloksena

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

- **Kekojärjestäminen** tapahtuu seuraavasti

```
heap-sort(A)
1   build-heap(A)
2   for i = A.length downto 2
3       vaihda A[1] ja A[i]
4       A.heap-size = A.heap-size −1
5       heapify(A,1)
```

- Toimintaidea
  - aineistosta tehdään ensin maksimikeko, näin suurin alkio on kohdassa $A[1]$
  - vaihdetaan keskenään keon ensimmäinen ja viimeinen alkio
  - näin saamme yhden alkion vietyä taulukon loppuun "oikealle" paikalleen
  - pienentämällä keon kokoa yhdellä huolehditaan vielä että viimeinen alkio ei enää kuulu kekoon
  - paikkaan $A[1]$ viety alkio saattaa rikkoa keko-ominaisuuden
  - huolehditaan vielä että keko-ominaisuus säilyy kutsumalla **heapify**$(A, 1)$
  - toistetaan samaa niin kauan kun keossa on alkiota

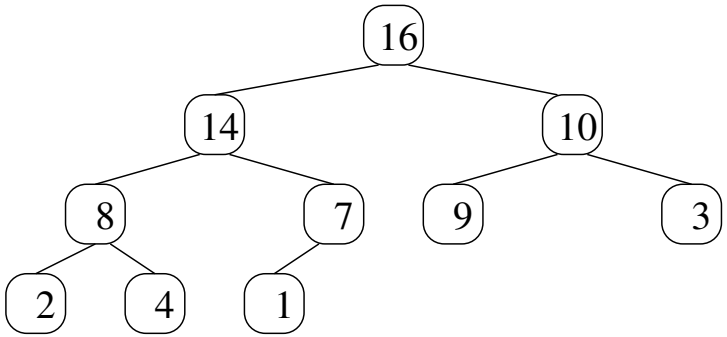
- **Kekojärjestämisen** aikavaativuus
  - **heapify**:n aikavaatimus on $\mathcal{O}(\log n)$ keolle jossa n alkiota
  - **build-heap**-operaatiossa kutsutaan $n/2$ kertaa **heapify** keolle, jossa on korkeintaan n alkiota, siis **build-heap** käyttää aikaa korkeintaan $\mathcal{O}(n \log n)$
  - **heap-sortissa** kutsutaan vielä $n - 1$ kertaa **heapify**-operaatiota
  - kokonaisuudessaan **kekojärjestämisen** aikavaativuus on siis $\mathcal{O}(n \log n)$
- Tilavaativuus
  - **heapify** kutsuu rekursiivisesti itseään pahimmillaan keon korkeudellisen verran, operaatio on kuitenkin helppo toteuttaa myös ilman rekursiota jolloin sen tilantarve on vakio
  - muutkaan **kekojärjestämisen** toimet eivät aputilaa tarvitse, siis tilavaativuus $\mathcal{O}(1)$

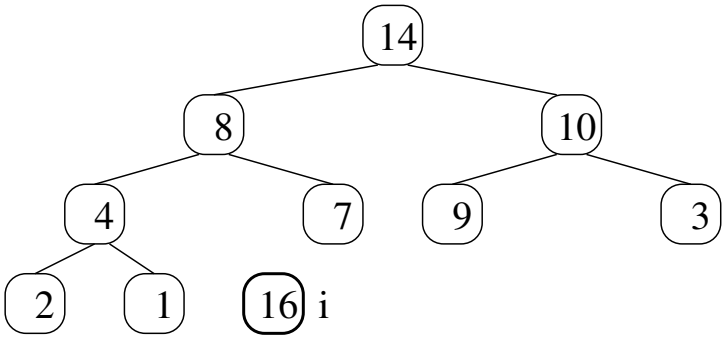
- Tarkempi analyysi paljastaa että operaation **build-heap** aikavaativuus onkin oikeastaan vain $\mathcal{O}(n)$
  - Operaation **heapify**(A, i) aikavaativuus on $\mathcal{O}(h(i))$, missä $h(i)$ on solmun i korkeus puussa
  - Keon korkeus on selvästi $k = \lfloor \log_2 n \rfloor$
  - Keon tasolla j on korkeintaan 2^j solmua ja näiden korkeus on $k - j$ tai $k - j - 1$, eli korkeintaan $k - j$
  - Operaation **build-heap** aikavaativuus on siis $\mathcal{O}(\sum_i h(i))$, mutta edellisen perusteella

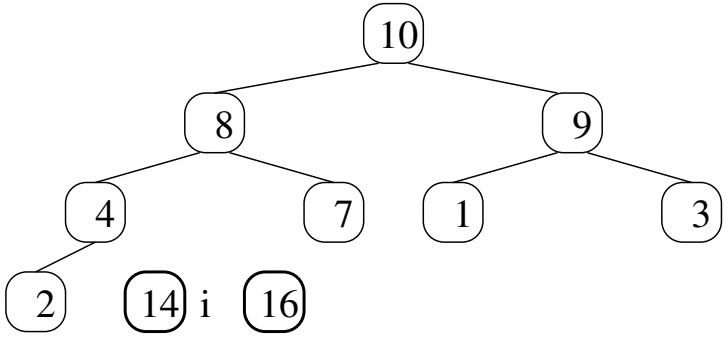
$$\sum_i h(i) \leq \sum_{j=0}^{k} 2^j (k-j) = \sum_{j=0}^{k} 2^{k-j} \cdot j = 2^k \sum_{j=0}^{k} j(\frac{1}{2})^j < 2^k \sum_{j=0}^{\infty} j(\frac{1}{2})^j \leq 2n,$$

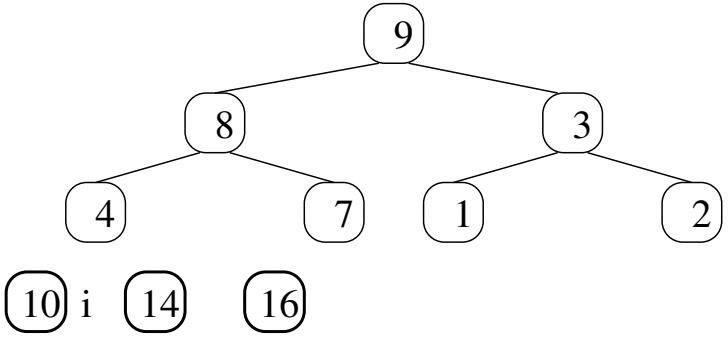
    koska $2^k \leq n$ ja kaavakirjasta näemme, että $\sum_{j=0}^{\infty} j(\frac{1}{2})^j = 2$

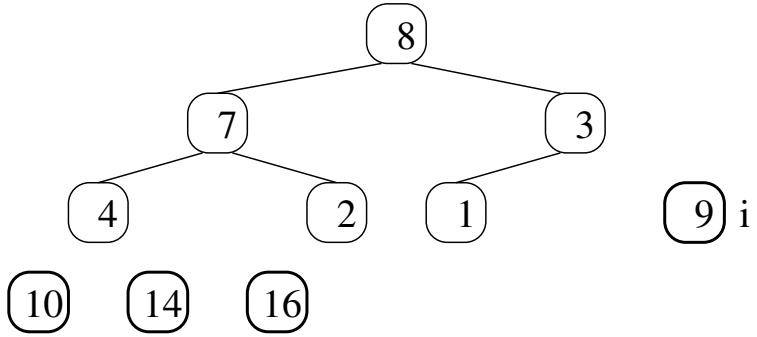
- Seuraavilla sivuilla on esimerkki **kekojärjestämisestä**

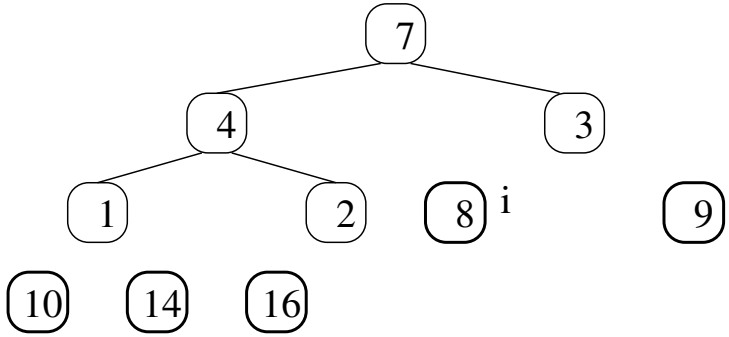

16
14
10
8
7
9
3
2
4
1

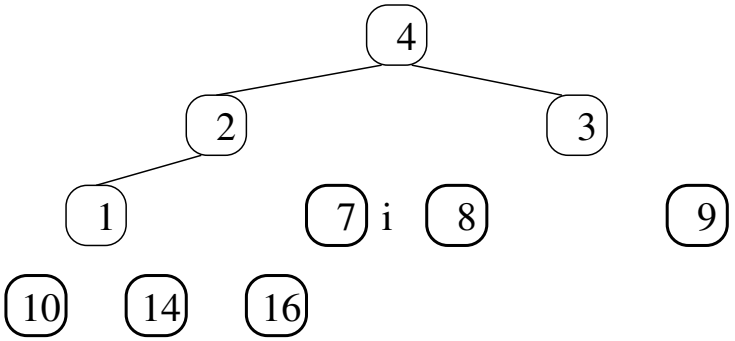

14
8
10
4
7
9
3
2
1
16
i

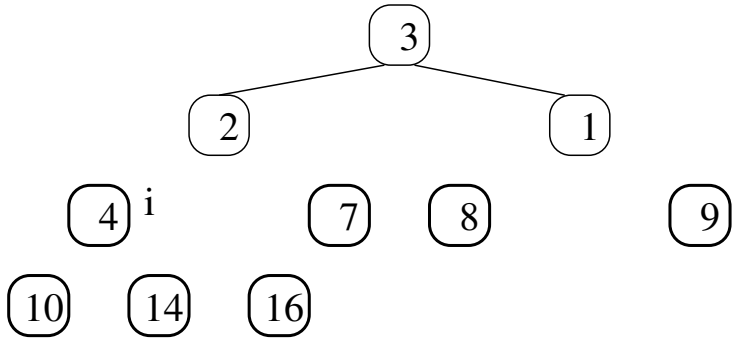

10
8
9
4
7
1
3
2
14
i
16

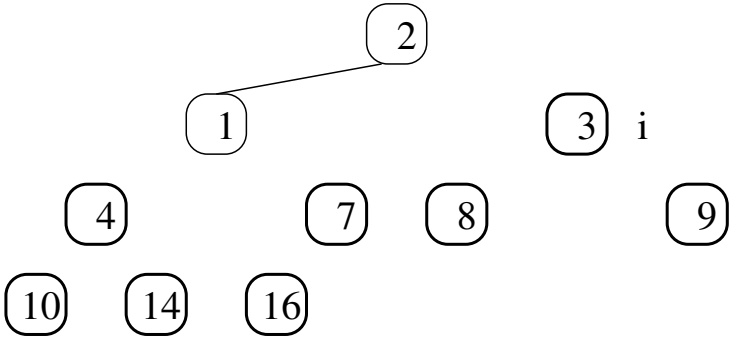

9
8
3
4
7
1
2
10
i
14
16

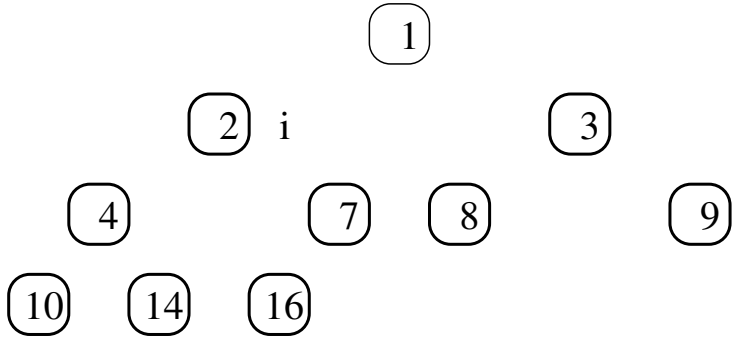

8
7
3
4
2
1
9
i
10
14
16


7
4
3
1
2
8
i
9
10
14
16









tuloksena

1	2	3	4	7	8	9	10	14	16

- Mistä johtuu, että **pikajärjestäminen** toimii käytännössä useimmiten nopeammin kuin **kekojärjestäminen**?

- **Kekojärjestäminen** vaihtaa hyvin usein täysin eri puolilla järjestettävää taulukkoa olevien alkioiden arvoja, **pikajärjestäminen** taas pysyttelee useimmiten pitemmän aikaa pienemmässä osassa taulukkoa

- Tällä on suuri merkitys käytännössä, sillä jos muistiviittaukset keskittyvät tietyllä ajanjaksolla pieneen osaan taulukkoa, on todennäköisempää että taulukon tarvittava osa löytyy välimuistista

- Välimuistiin tehtävien muistihakujen viemä aika on merkittävästi pienempi verrattuna siihen, että tieto jouduttaisiin hakemaan keskusmuistista

- Asian merkitys korostuu vielä enemmän, jos koko järjestettävä taulukko ei mahdu kerralla keskusmuistiin, vaan sijaitsee osittain kiintolevyn swap-osiossa

- Käytännössä on myös osoittautunut, että **kekojärjestäminen** suorittaa keskimäärin monta kertaa enemmän vertailu- ja sijoitusoperaatioita kuin **pikajärjestäminen**

- Ei ole mitenkään ilmeistä mistä tämä seikka johtuu.

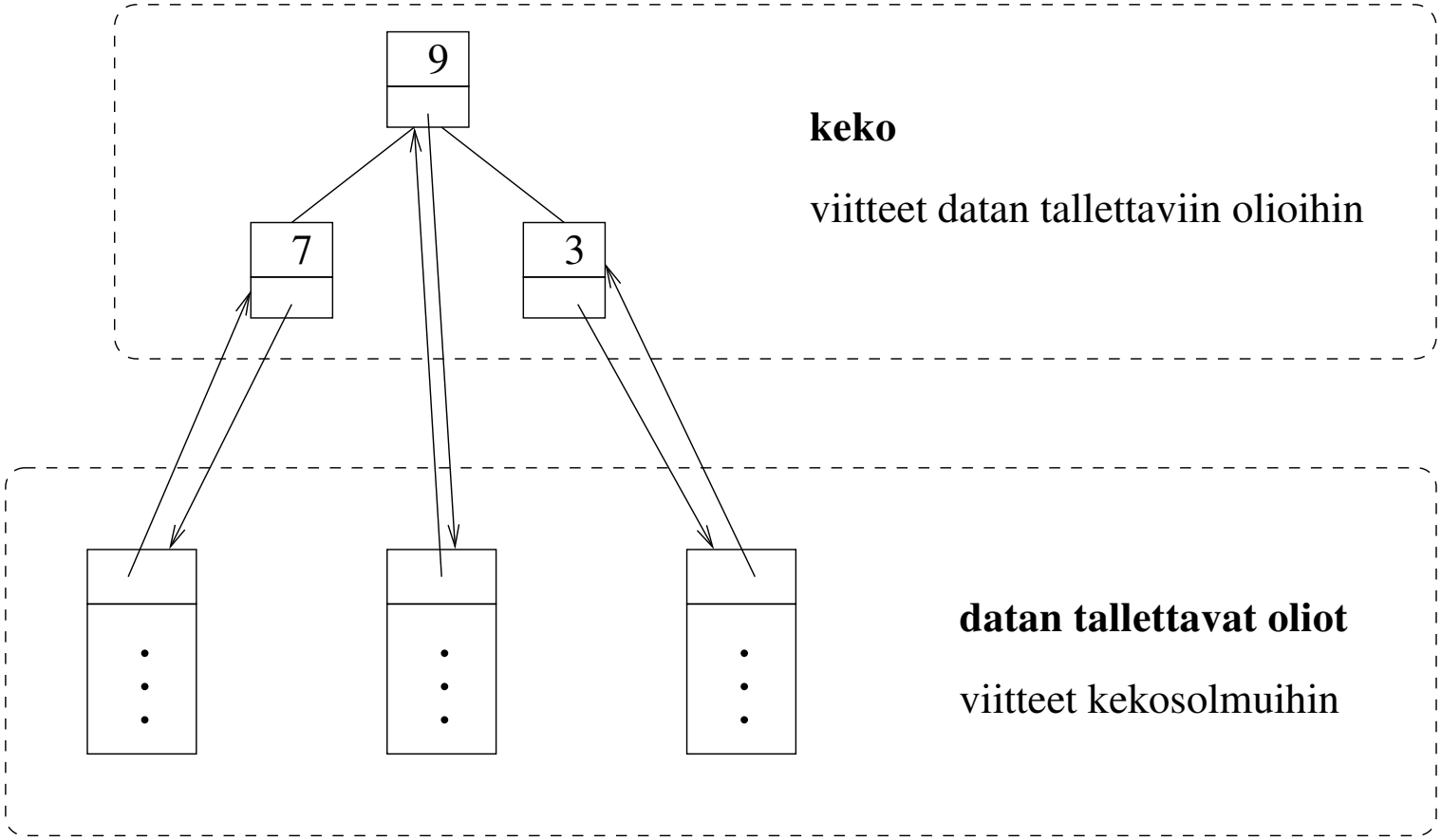
Keko käytännössä

- Monissa käytännön sovelluksissa, esim. käytettäessä kekoa prioriteettijonona, keottavat alkiot sisältävät muitakin attribuutteja kuin pelkän avaimen
- Tällöin itse kekoon ei välttämättä kannata tallettaa muuta kuin avaimet sekä viitteet avaimeen liittyvään muun datan tallettavaan olioon
- Tällaisessa käytännön tilanteessa keko-operaatioiden parametrit kannattanee valita seuraavasti

- **heap-insert**(A, x, k)	lisää kekoon olion x, jolla avain k
- **heap-max**(A)	palauttaa viitteen olioon jolla on avaimena keon maksimiarvo
- **heap-del-max**(A)	palauttaa viitteen olioon jolla on avaimena keon maksimiarvo ja poistaa tämän avaimen keosta
- **heap-inc-key**$(x, newk)$	kasvattaa olion x avainta antaen sille uuden arvon $newk$
- **heap-dec-key**$(x, newk)$	pienentää olion x avainta antaen sille uuden arvon $newk$

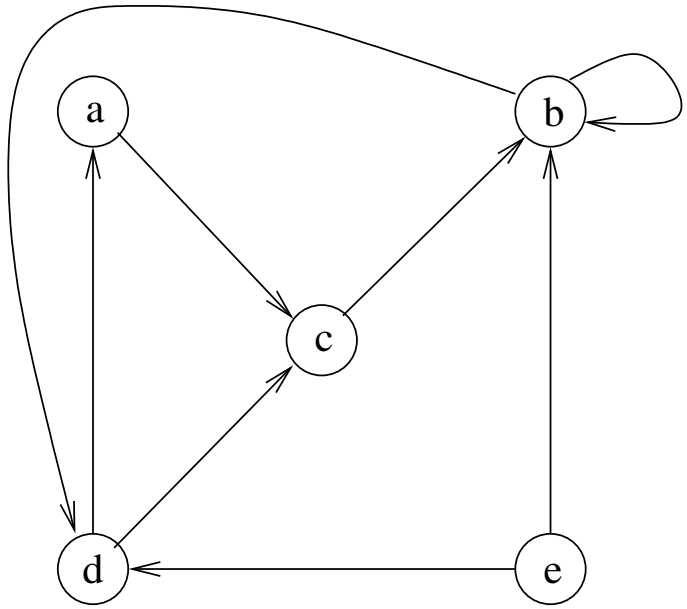
- Jotta operaatio **heap-inc-key** saadaan toteutettua tehokkaasti datan tallettavissa olioissa on myös oltava viite vastaavaan kekoalkioon
- Käytännössä viitteet siis ovat kekotaulukon indeksejä eli kokonaislukuja

- Muistin organisointi näyttää esim. seuraavalta:

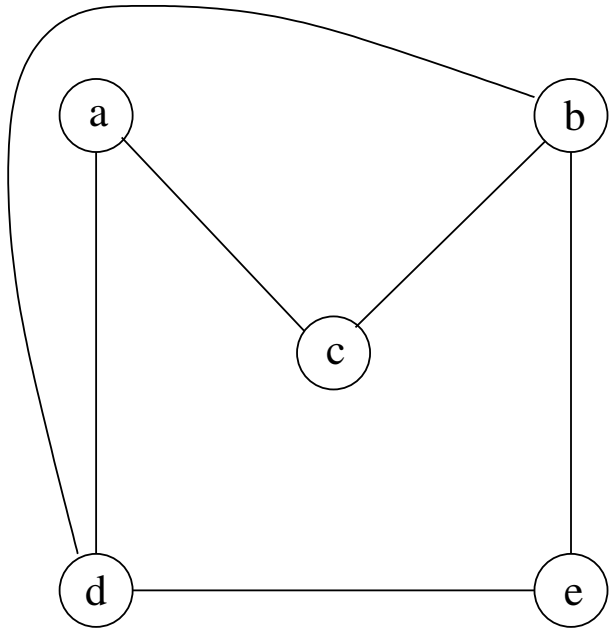


8. Verkot

- Verkko (engl. graph) koostuu solmuista (engl. vertex, node) ja niitä yhdistävistä kaarista (engl. edge)
- Verkkoja on kahta päätyyppiä
- Suunnatuissa verkoissa (engl. directed graph) kaarilla (engl. edge, arc) on suunta
- Esim:



- Suuntaamattomien verkkojen (engl. undirected graph) kaarilla ei ole suuntaa
- Esim:



- Verkoilla on paljon sovelluksia tietojenkäsittelyssä (ja muualla)
- Tutustutaan ensin verkon käsitteistöön, sen jälkeen katsotaan muutamia verkkojen sovelluksia ja tutustutaan tyypillisimpiin verkkoalgoritmeihin

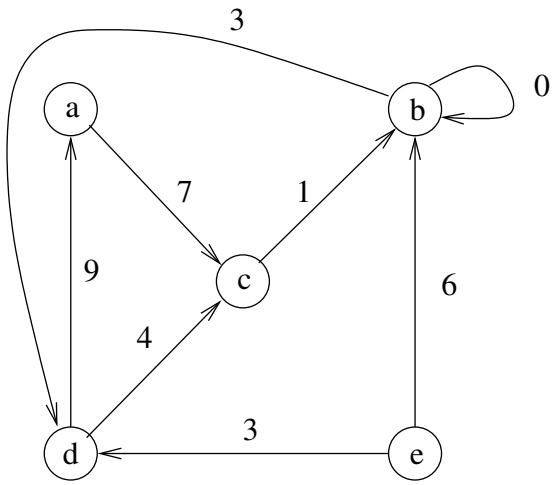
Käsitteistö

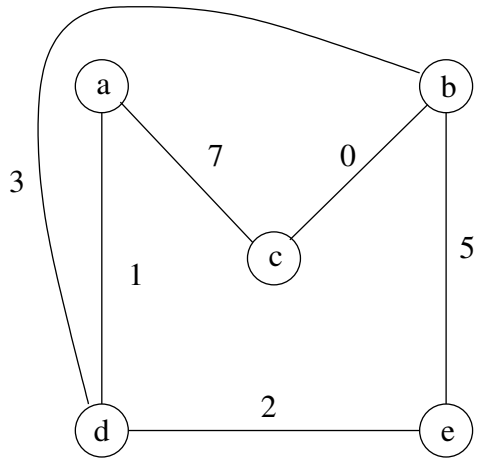
- Formaalisti verkko G esitetään parina (V, E), missä
  - V on solmujen joukko
  - E on kaarien joukko, $E \subseteq V \times V$
- Merkinnät: G niin kuin graph, V vertex ja E edge
- Kaaret ovat siis pareja (u, v) missä u ja v ovat solmuja
- Suunnatussa verkossa $(u, v) \in E$ jos solmusta u on kaari solmuun v
  - tällöin u on kaaren lähtösolmu ja v kaaren maalisolmu
  - solmua v sanotaan solmun u vierussolmuksi (engl. adjacent vertex)
  - suunnatun verkon kaarista käytetään usein myös merkintää $u \to v$
- Sivun 431 suunnatussa verkossa siis esim. solmun e vierussolmut ovat b ja d, sillä $e \to b$ ja $e \to d$

  solmun b vierussolmut ovat d ja solmu itse, sillä $b \to d$ ja $b \to b$

- Esim: sivun 431 kuvan suunnatun verkon formaali määritelmä:
  - $V = \{a, b, c, d, e\}$
  - $E = \{(a, c), (b, b), (b, d), (c, b), (d, a), (d, c), (e, b), (e, d)\}$
- Edellä mainittiin että verkon kaaret muodostavat joukon, eli kahden solmun välillä ei määritelmän mukaan voi olla kahta samaan suuntaan kulkevaa kaarta
  - joissakin sovelluksissa tilanne poikkeaa tästä, ja on mielekästä sallia, että kahden solmun välillä on useita kaaria (ns. moniverkko; engl. multigraph)
- Suunnatun verkon solmujen u ja v välillä voi sen sijaan olla kaaret molempiin suuntiin $u \to v$ ja $v \to u$
- Suuntaamattomassa verkossa kaarten joukko E on symmetrinen (engl. symmetric), eli jos $(u, v) \in E$ niin myös $(v, u) \in E$
  - merkitsemme myös suuntaamattoman verkon kaaria joskus $u \to v$
  - jos $(u, v) \in E$ sanotaan että solmut u ja v ovat vierekkäisiä, (engl. adjacent) eli v on u:n vierussolmu ja u on v:n vierussolmu
- Esim: sivun 432 suuntaamaton verkko formaalisti määriteltynä:
  - $V = \{a, b, c, d, e\}$
  - $E = \{(a, c), (c, a), (b, d), (d, b), (c, b), (b, c), (d, a), (a, d), (e, b), (b, e), (e, d), (d, e)\}$

- Usein verkon kaariin liitetään paino (engl. weight)





- Tässä kaaripainot ovat kokonaislukuja, mutta näin ei välttämättä tarvitse olla
- Kaaripainon käsite määritellään funktiona $w : E \to \{0, 1, 2 \ldots\}$
- Eli funktio w liittää jokaiseen kaareen painon, esim. kuvan suunnatussa verkossa $w(a, c) = 7$, $w(e, d) = 3$ jne.
- Painotetun verkon kaarista $(u, v) \in E$ käytetään myös merkintää $u \overset{w(u,v)}{\to} v$, eli esimerkissämme on kaari $a \overset{7}{\to} c$
- Kaaripainoilla voidaan ilmaista esim. solmujen välisiä etäisyyksiä, niiden välisen yhteyden hintaa ym., painon ei siis välttämättä tarvitse olla kokonaisluku

- Solmujono $v_1, v_2, \ldots, v_n$ on polku (engl. path) solmusta v_1 solmuun v_n, jos verkossa on kaaret $v_1 \to v_2, v_2 \to v_3, \ldots, v_{n-1} \to v_n$

- Jos solmusta u on polku solmuun v, käytetään merkintää $u \rightsquigarrow v$

- Jos $u \rightsquigarrow v$, sanotaan että solmu v on saavutettavissa (engl. reachable) solmusta u

- Polun pituus on polkuun liittyvien kaarien lukumäärä (kaikilla u on pituutta 0 oleva polku $u \rightsquigarrow u$)

- Painotetussa verkossa polun paino on polun kaarien yhteenlaskettu paino
  - Huom: Painotetussa verkossa polun painoa kutsutaan usein myös polun pituudeksi (ja näin mekin teemme)

- Polku on yksinkertainen (engl. simple), jos kukin solmu esiintyy polussa vain kerran, paitsi että viimeinen ja ensimmäinen saavat olla sama solmu

- Yksinkertainen polku on sykli eli kehä (engl. cycle), jos viimeinen ja ensimmäinen solmu ovat samat

- Sivun 435 suunnatun painotetun verkon polkuja:
  - $e \xrightarrow{3} d \xrightarrow{4} c \xrightarrow{1} b$ on yksinkertainen syklitön polku jonka pituus on 3 ja paino 8
  - $d \xrightarrow{9} a \xrightarrow{7} c \xrightarrow{1} b \xrightarrow{3} d$ on sykli jonka pituus on 4 ja paino 20
  - $c \xrightarrow{1} b \xrightarrow{0} b \xrightarrow{0} b \xrightarrow{3} d$ on polku jonka pituus 4, paino 4 ja joka sisältää kaksi sykliä
- Suunnattu verkko on syklitön (engl. acyclic) jos se ei sisällä yhtään sykliä
- Verkon ei välttämättä tarvitse olla yhtenäinen, eli verkko voi koostua useista erillisistä osista
- Suuntaamaton verkko on yhtenäinen (engl. connected), jos $u \rightsquigarrow v$ kaikilla $u, v \in V$
- Suunnattu verkko on vahvasti yhtenäinen (engl. strongly connected), jos $u \rightsquigarrow v$ kaikilla $u, v \in V$. ”Vahva” korostaa, että kaarten suuntauksia on kunnioitettava

- Huom: Englannin kielessä syklittömästä suunnatusta verkosta käytetään lyhennettä DAG (directed acyclic graph)

- Vapaa puu (engl. free tree) on suuntaamaton verkko, joka on sekä syklitön että yhtenäinen. Puussa minkä tahansa kahden solmun välillä on tasan yksi yksinkertainen polku

- Juurellinen puu (engl. rooted tree) on usein luontevaa esittää muodostamalla vastaava vapaa puu ja suuntaamalla sitten kaaret juurta kohti

Esimerkkejä verkoista ja verkko-ongelmista

- **Tietokoneverkon yhtenäisyys**:
  solmut: tietokoneita
  kaaret: tietoliikenneyhteyksiä; ei suuntausta, ei yleensä painoja
  ongelma: mitkä yhteydet ovat sellaisia, että niiden katkeaminen jakaisi verkon kahteen toisistaan eristettyyn osaan

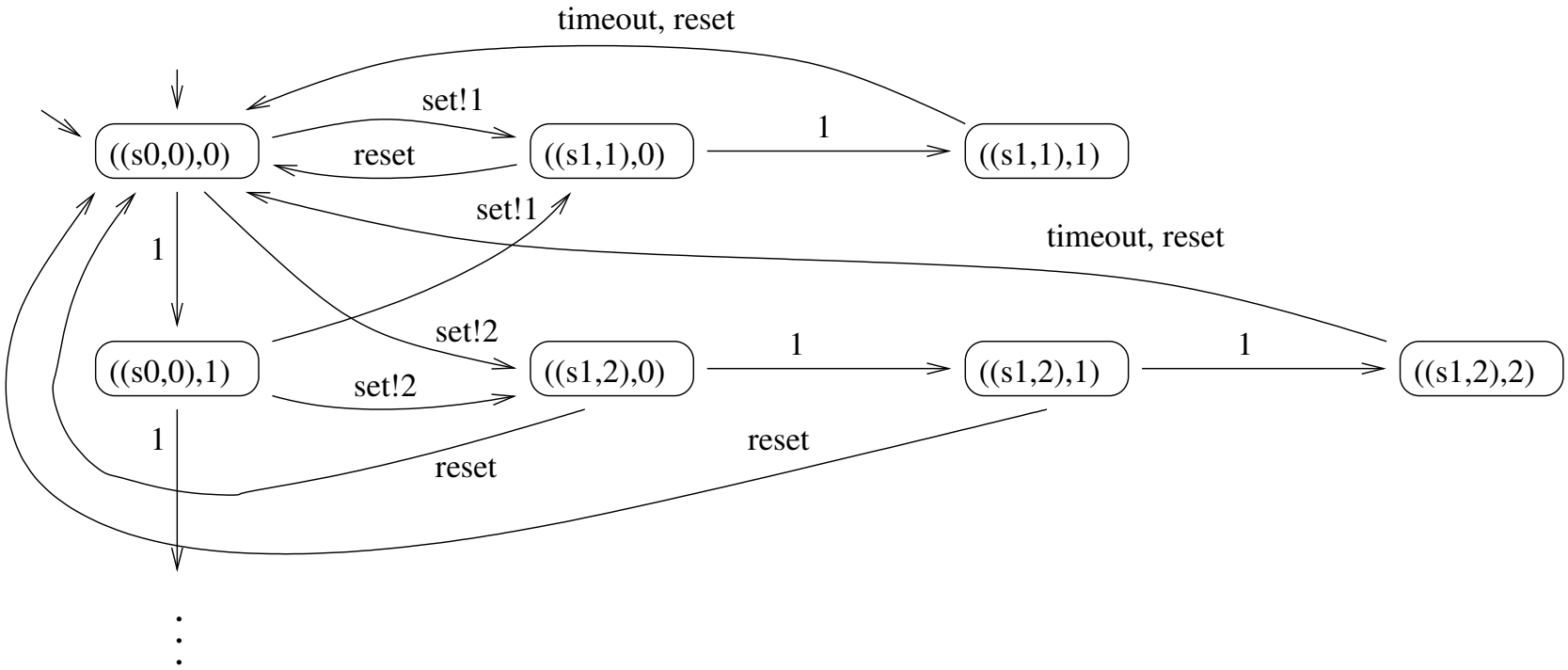
- **Robotin navigointi:**
  solmut: sopivalla tarkkuustasolla esitettyjä maantieteellisiä sijainteja
  kaaret: tunnettuja väyliä; ei suuntausta, ei painoja
  ongelma: ohjaa robotti paikasta A paikkaan B

- **Maantieverkosto:**
  solmut: kaupunkeja
  kaaret: maanteitä; ei suuntausta
  painot: kaupunkien välisiä etäisyyksiä
  ongelma: mikä on lyhin reitti kaupungista A kaupunkiin B

- **Logistiikkaverkosto**:
  solmut: varastoja
  kaaret: olemassaolevia kuljetusreittejä; ei suuntausta
  painot: useasta tavaralajista tieto, kuinka paljon sitä voidaan tietyssä ajassa kuljettaa mitäkin reittiä pitkin
  ongelma: miten saadaan halutut määrät tavaroita kulkemaan eri varastojen välillä

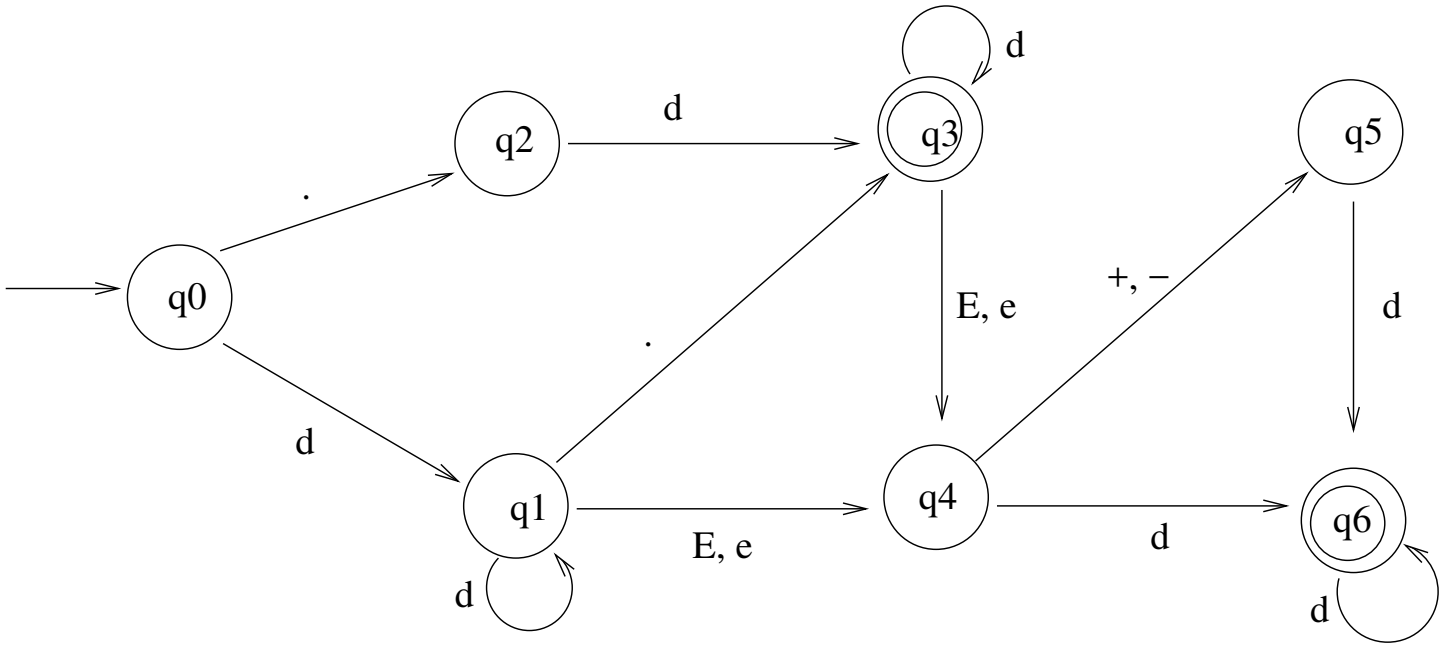
- **Tilasiirtymäverkko:**
  - solmut: reaaliaikaisen järjestelmän tiloja
  - kaaret: tilojen välisiä siirtymiä; suunnattu
  - painot: mikä ulkoinen tapahtuma aiheuttaa minkäkin tilasiirtymän
  - ongelma: voiko jokin tapahtumajono johtaa johonkin epätoivottuun tilaan tai tapahtumajonoon

- Esimerkki tilasiirtymäverkosta



- **Äärellinen automaatti** (kurssilla Laskennan mallit):
  - solmut: abstraktin automaatin laskennan tilanteita
  - kaaret: abstraktin automaatin laskenta-askelia
  - painot: merkkejä
  - ongelma: Voiko tilasta A kulkea tilaan B siten, että kuljettujen kaarten painoista muodostuu haluttu merkkijono

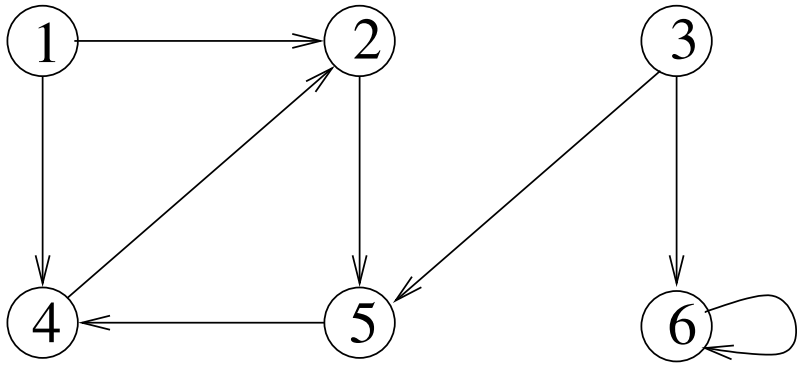
- Esimerkki äärellisestä automaatista, joka määrittelee etumerkittömän Javan float -tyyppisen vakion syntaktisesti oikean muodon

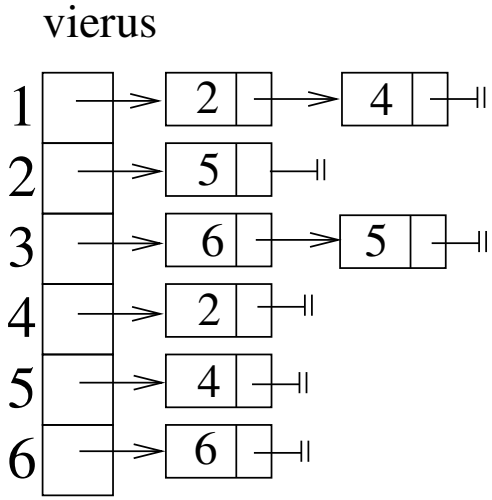


Verkkojen tallettaminen

- Tarkastellaan seuraavassa tapoja verkon $G = (V, E)$ esittämiselle tietokoneohjelmassa
- Merkitään solmujen lukumäärää $|V|$ ja kaarien lukumäärää $|E|$
- Yleensä käytetään jompaa kumpaa kahdesta talletustavasta:
  - vieruslistat (engl. adjacency lists)
  - vierusmatriisi (engl. adjacency matrix)
  - on myös tilanteita, joissa verkko on mielekkäämpi tallettaa jossain muussa muodossa tai verkkoa ei edes kannata tallentaa etukäteen
- Vieruslistaesityksessä verkko $G = (V, E)$ esitetään taulukkona $vierus$ joka sisältää $|V|$ kappaletta linkitettyjä listoja, yhden kullekin verkon solmulle
- Jokaiselle solmulle $u \in V$ lista $vierus[u]$ sisältää kaikki ne solmut joihin u:sta on kaari

- Esim: suunnattu verkko ja sen vieruslistaesitys



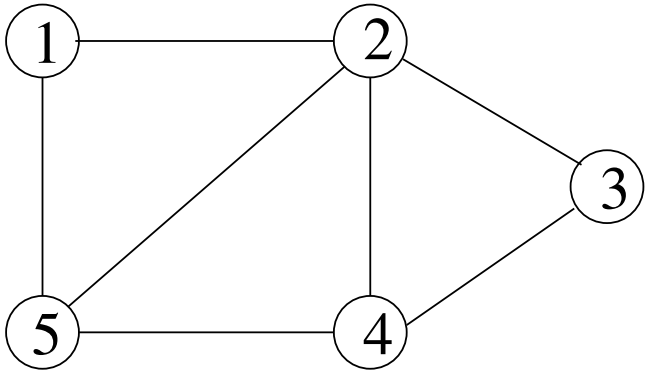


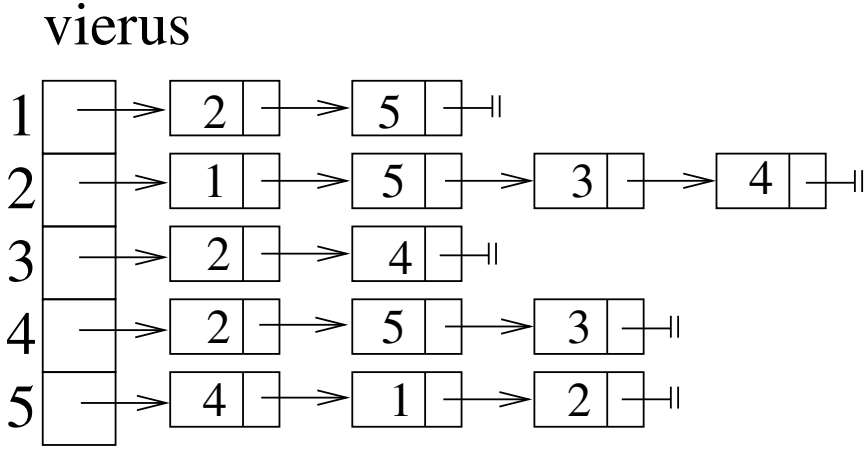
- Suunnatun verkon vieruslistojen yhteenlaskettu pituus on $|E|$ sillä jokainen kaari on talletettu kertaalleen yhteen vieruslistoista

- Koko vieruslistaesitys vie suunnattujen verkkojen tapauksessa tilaa $\mathcal{O}(|E| + |V|)$, sillä kaarien lisäksi varataan luonnollisesti tila taulukolle $vierus$

- Javassa vieruslistat voitaisiin esittää taulukollisena LinkedList-olioita

- Esim: suuntaamaton verkko ja sen vieruslistaesitys

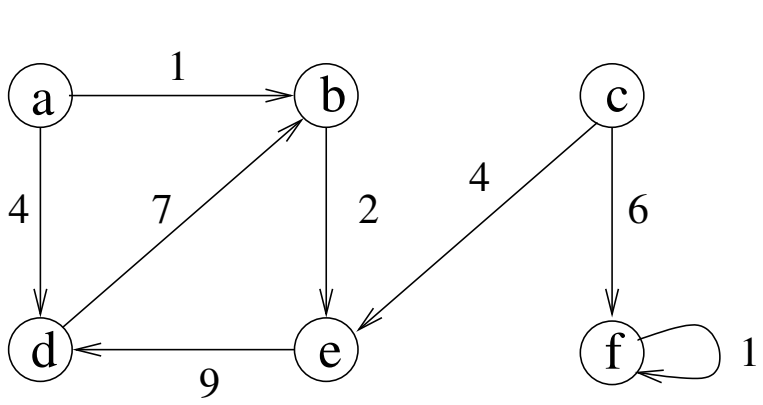


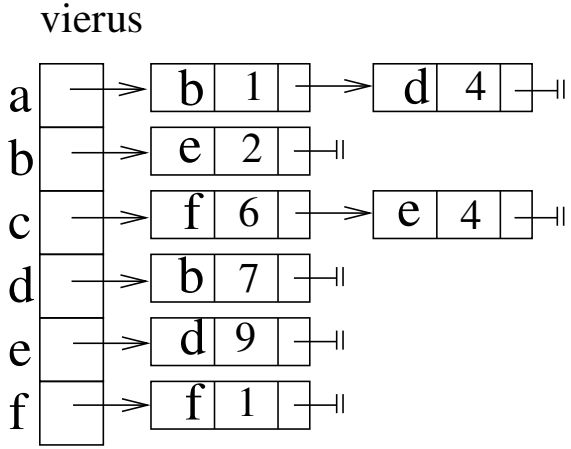


- Suuntaamattoman verkon vieruslistojen yhteenlaskettu pituus $|E|$ on kaksi kertaa kaarien lukumäärä, sillä jokainen kaari on talletettu kahteen vieruslistaan

- Koko vieruslistaesitys vie suuntaamattomien verkkojen tapauksessa tilaa $\mathcal{O}(|V| + 2 \cdot |E|) = \mathcal{O}(|V| + |E|)$

- Myös kaarien painot voidaan tallentaa vieruslistarakenteeseen:





- Vieruslistaesityksen hyvä puoli on siis kohtuullinen tilantarve joka on $\mathcal{O}(|E| + |V|)$, eli lineaarinen suhteessa solmujen ja kaarten määrään

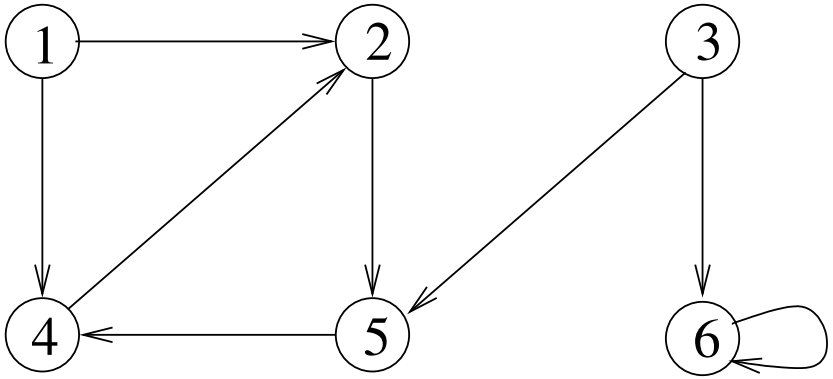
- Huonona puolena taas se että tieto onko verkossa kaarta $u \to v$ ei ole suoraan saatavilla, vaan vaatii vieruslistan $vierus[u]$ läpikäynnin

- Pahimmillaan tämä operaatio vie aikaa $\Omega(|V|)$ sillä solmusta u voi olla pahimmassa tapauksessa kaari kaikkiin verkon solmuihin

- Verkon $G = (V, E)$ vierusmatriisiesityksessä oletetaan että solmut on numeroitu, eli esim. $V = \{1, 2, \ldots, n\}$

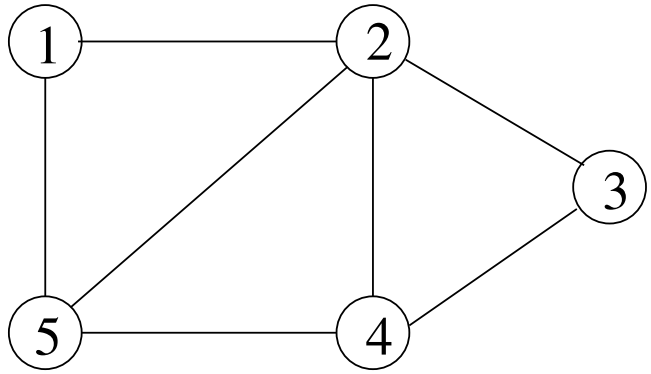
- Vierusmatriisi on $n \times n$-matriisi A, missä $A[i, j] = 1$ jos $(i, j) \in E$ ja muuten $A[i, j] = 0$

- Esimerkki suunnatusta verkosta:



	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**

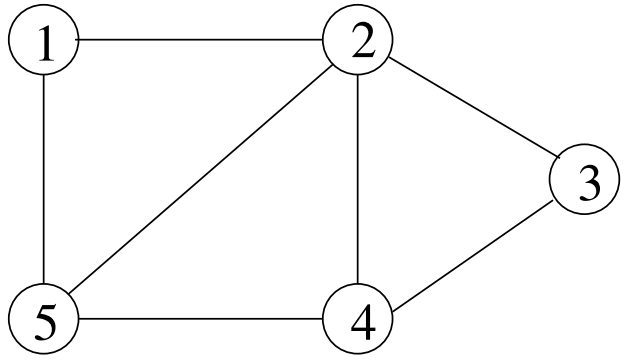
- Esimerkki suuntaamattomasta verkosta:



	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, esim. koska $(2,5) \in E$, niin $A[2,5] = 1$ ja $A[5,2] = 1$

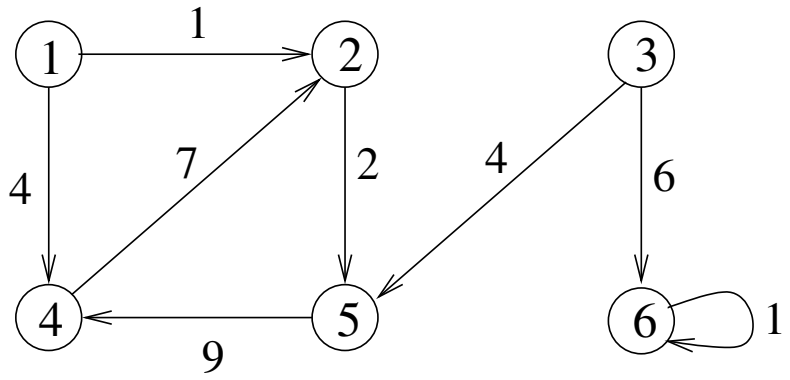
- Suuntaamattoman verkon tapauksessa riittäisikin siirtymämatriisista puolikas:



1	2	3	4	5	
0	**1**	0	0	**1**	1
	0	**1**	**1**	**1**	2
		0	**1**	0	3
			0	**1**	4
				0	5

- Usein on vieläpä käytössä rajoitus että suuntaamattomassa verkossa kaaret muotoa (i, i) eivät ole sallittuja, jos näin on, ei vierusmatriisin diagonaalia (eli alkioita $A[1, 1], A[2, 2], \ldots$) myöskään tarvita

- Kaaripainojen tallettaminen vierusmatriisiin on vaivatonta:



	1	2	3	4	5	6
1	∞	**1**	∞	**4**	∞	∞
2	∞	∞	∞	∞	**2**	∞
3	∞	∞	∞	∞	**4**	**6**
4	∞	**7**	∞	∞	∞	∞
5	∞	∞	∞	**9**	∞	∞
6	∞	∞	∞	∞	∞	**1**

- Painotetun verkon vierusmatriisissa periaatteena siis on asettaa $A[i, j] = w(i, j)$ jos $(i, j) \in E$ ja muuten $A[i, j] = \infty$ tai $A[i, j] = 0$
  - jos kaaren $i \xrightarrow{x} j$ paino kuvaa reitin i:stä j:hin pituutta tai kustannusta, on ääretön luonnollinen valinta olemattomien kaarien merkintään
  - jos kaari taas kuvaa reitin kapasiteettia, on luonnollinen valinta olemattomien kaarien merkintään nolla

- Hyvänä puolena vierusmatriisissa on se että tietyn kaaren olemassaolo selviää matriisista vakioajassa

- Toisaalta solmun kaikkien kaarien selvittämiseen kuluu aikaa aina $\mathcal{O}(|V|)$ vaikka kaaria olisikin vain yksi

- Toinen huono puoli vierusmatriisissa on tilan tarve, matriisin koko on kaarien lukumäärästä riippumatta aina $|V| \times |V|$

- Verkkoa sanotaan harvaksi jos kaaria on suhteellisen vähän, esim. vain kaksi kertaa solmujen määrä
  - Kaarien määrä on tällöin $|E| = \mathcal{O}(|V|)$, ja vieruslistana esitetty verkko vie tilaa $\mathcal{O}(|E| + |V|) = \mathcal{O}(|V|)$ kun taas vierusmatriisiesityksen tilantarve on tähän verrattuna neliöinen $\mathcal{O}(|V| \times |V|)$

- Isot harvat verkot siis kannattanee tallentaa vieruslistoja käyttäen

- Osa verkkoalgoritmeista tosin olettaa että verkko on talletettu esim. käyttäen vierusmatriiseja, eli verkon talletusmuoto riippuu paljolti myös verkon käyttötarkoituksesta

Verkon muunlaiset esitystavat

- Vieruslista- ja vierusmatriisiesitys olettavat, että verkon solmut ja kaaret on eksplisiittisesti generoitu koneen muistiin
- Usein tämä ei ole tarpeen tai tarkoituksenmukaista
- Verkosta voi olla olemassa jokin muunlainen esitysmuoto ja verkko "verkkona" on olemassa vain ohjelmoijan taustalla olevana mentaalisena mallina
- Esim. tehtävänä on etsiä löytyykö merkkeinä kuvatusta labyrintista X:llä merkitystä kohdasta reittiä ulos

```
####.####
#..#....#
#.#.###.#
#....X..#
#########
```

- Lienee mielekästä tulkita mahdolliset sijaintipaikat eli pisteet solmuiksi
  - jokaisen vierekkäisen pistettä edustavan solmun välille tulee kaari
  - labyrintin seinät eli #-merkit taas ovat kaarettomia kohtia
- Labyrintti kannattanee tallettaa koneen muistiin kaksiulotteisena taulukkona jossa esim. 1 merkitsee seinätöntä kohtaa ja 0 seinää:

  000010000
  011011110
  010100010
  011111110
  000000000

- Solmut vastaavat nyt taulukon kohtia, esim. yläreunan aukko labyrintista ulos on taulukon kohta lab[0,4]. Ensimmäinen indeksi siis tarkoittaa riviä ja toinen saraketta
- Kaarien olemassaolo selviää nyt suoraan taulukosta, eli esim
  - kohdasta lab[3,5] (paikka mistä aloitetaan, eli missä on alussa X) on kaari kohtaan lab[3,4] ja lab[3,6] koska molemmissa kohdissa taulukossa on 1
  - kohdasta lab[1,1] kaari kohtiin lab[1,2] ja lab[2,1]
- Verkkoa ei siis kannattane esittää vieruslista- tai vierusmatriisiesityksenä koska taulukosta lab selviää kaikki kaaria koskeva informaatio

Verkon läpikäynti

- Tyypillistä verkkoa käytettäessä on että halutaan kulkea verkossa systemaattisesti vieraillen kaikissa solmuissa tai ainakin kaikissa tietystä solmusta saavutettavissa olevissa solmuissa

- Solmujen läpikäyntiin on kaksi perusstrategiaa:
  - leveyssuuntainen läpikäynti (engl. breadth-first search), ja
  - syvyyssuuntainen läpikäynti (engl. depth-first search)

- Algoritmit tuottavat myös erilaista lisäinformaatiota verkon rakenteesta, joten niitä käytetään muissakin verkkoalgoritmeissa esiproseduurina tai rakennusalustana

- Toimivat sekä suunnatuille että suuntaamattomille verkoille (mutta tulosten tulkinta voi poiketa)

Leveyssuuntainen läpikäynti

- Verkon $G = (V, E)$ leveyssuuntaisessa läpikäynnissä (engl. breadth-first search) tutkitaan mitkä verkon solmuista ovat saavutettavissa annetusta aloitussolmusta $s \in V$

- Läpikäynti etenee uusiin solmuihin "taso kerrallaan", eli ensin etsitään mitkä solmut saavutetaan s:stä yhden pituista polkua käyttäen, tämän jälkeen edetään solmuihin jotka ovat saavutettavissa s:stä kahden mittaista polkua käyttäen, jne.

- Algoritmin sivutuotteena selvitetään mikä on polun pituus aloitussolmusta s kuhunkin läpikäynnin aikana löydettyyn solmuun v. Tieto talletetaan taulukkoon $distance$

- 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 lyhintä polkua $s \rightsquigarrow v$
  - puun kaaret talletetaan taulukkoon $tree$, ja taulukon alkio $tree[v]$ kertoo mistä solmusta lyhin polku solmuun v saapuu

- Algoritmin kirjanpitoa varten verkon solmuihin tarvitaan vielä kolmaskin taulukko, $color$, jossa pidetään kirjaa solmujen "väreistä"

  taulukon alkio $color[v]$ kertoo onko läpikäynti jo löytänyt solmun v

  - solmut joita läpikäynti ei ole vielä löytänyt ovat valkoisia, eli niille $color[v] =$ white
  - jos läpikäynti on ehtinyt solmuun v, mutta solmusta u edelleen vieviä kaaria ei ole vielä käsitelty, niin solmun väriksi asetetaan harmaa $color[v] =$ gray
  - kun solmusta v lähtevät kaaret on käsitelty (jolloin koko solmu v on käsitelty), asetetaan sen väriksi musta, eli $color[v] =$ black

- Aluksi kaikille paitsi aloitussolmulle s merkitään $color[v] =$ white

- Aloitussolmulle s merkitään aluksi $color[s] =$ gray

- Algoritmi käyttää aputietorakenteenaan jonoa Q joka on aluksi tyhjä; jonossa ovat tietyllä hetkellä ne solmut jotka läpikäynti on jo löytänyt, mutta joiden naapurisolmuja ei vielä ole käsitelty

- Rivien 1-7 alustusvaiheen jälkeen aloitussolmu s on merkitty harmaaksi ja laitettu jonoon

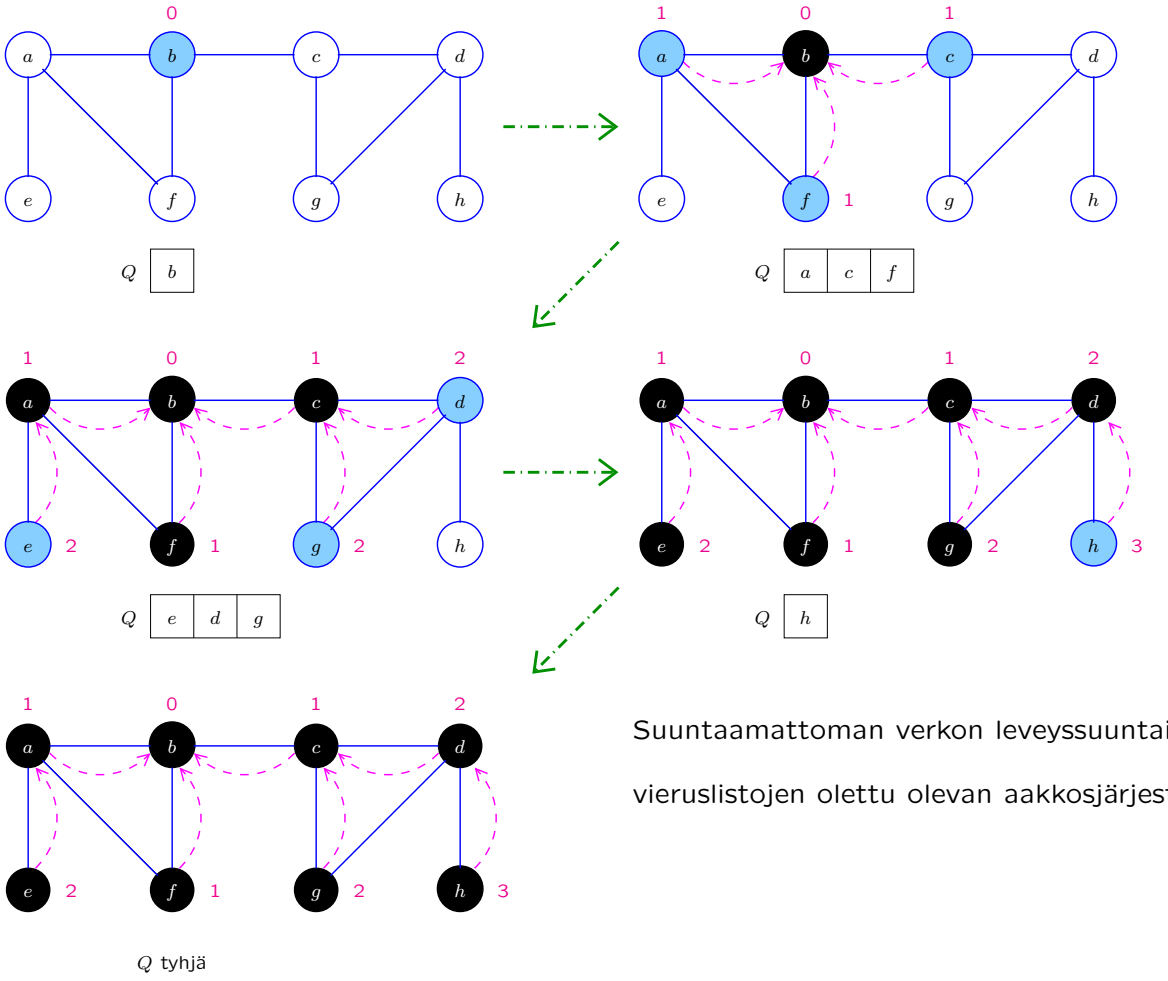
- Rivien 8-16 toimintaperiaate
  - aloitussolmu s otetaan jonosta, ja kaikki sen vierussolmut laitetaan jonoon
  - vierussolmujen etäisyydeksi päivitetään 1, niitä leveyssuuntaispuussa edeltäväksi solmuksi asetetaan s ja merkitään solmut harmaiksi
  - solmu s on nyt käsitelty ja se muuttuu mustaksi
  - tämän jälkeen niin kauan kun jonossa on solmuja, otetaan käsittelyyn jonon alussa oleva solmu u
  - laitetaan jonoon ne u:n vierussolmut, joita etsintä ei ole vielä kohdannut, eli joille $color[v] =$ white ja
  - päivitetään jonoon laitettujen etäisyys- ja leveyssuuntaispuutietoa sekä merkitään ne harmaiksi
  - kun solmu u on käsitelty valmiiksi se merkitään mustaksi

- Algoritmi

```
BFS(G,s)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3       distance[u] = ∞
4       tree[u] = NIL
5   color[s] = gray
6   distance[s] = 0
7   enqueue(Q,s)
8   while ( not empty(Q) )
9       u = dequeue(Q)
10      for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
11          if color[v]==white                   // solmua v ei vielä löydetty
12              color[v] = gray
13              distance[v] = distance[u]+1
14              tree[v] = u
15              enqueue(Q,v)
16      color[u] = black
```

- Algoritmi toimii sekä suunnatuilla että suuntaamattomilla verkoilla

- Esimerkki algoritmin toiminnasta seuraavalla sivulla (harmaat solmut ovat kuvassa sinisiä)



Suuntaamattoman verkon leveyssuuntainen läpikäynti

vieruslistojen olettu olevan aakkosjärjestyksessä

- Algoritmin suorituksen jälkeen lyhin polku $s \rightsquigarrow v$ saadaan selville seuraavasti:
  - $tree[v]$ kertoo minkä solmun kautta lyhin polku $s \rightsquigarrow v$ saapuu solmuun v
  - solmuun $tree[v]$ lyhin polku saapuu solmun $tree[tree[v]]$ kautta, jne
  - laitetaan pinoon $tree[v]$, $tree[tree[v]]$, $tree[tree[tree[v]]]$ ja tulostetaan pinon sisältö
  - näin saadaan tulostettua polulla koko polku $s \rightsquigarrow v$ alusta loppuun

- Algoritmina:

```
shortest-path(G,v)
1   u = tree[v]
2   while u ≠ s
3       push(S,u)
4       u = tree[u]
5   print( "lyhin polku solmusta s solmuun v")
6   while not empty(S)
7       u = pop(S)
8       print(u)
```

- Leveyssuuntaisen läpikäynnin aikavaativuus:
  - alustukseen (rivit 1-6) kuluu aikaa $\mathcal{O}(|V|)$
  - koska jonoon laitettava solmu värjätään harmaaksi eikä väri enää muutu takaisin valkoiseksi, takaa rivin 11 testi että jokainen solmu laitetaan jonoon vain kerran
  - jokainen solmu siis myös poistetaan jonosta korkeintaan kerran
  - enqueue ja dequeue-operaatiot voidaan toteuttaa ajassa $\mathcal{O}(1)$, eli kokonaisuudessaan jono-operaatioihin kuluu aikaa $\mathcal{O}(|V|)$
  - kunkin solmun vieruslista käydään läpi ainoastaan silloin kuin solmu poistetaan jonosta, eli korkeintaan kerran
  - vieruslistojen yhteispituus on $\mathcal{O}(|E|)$, eli yhteensä vieruslistojen läpikäyntiin käytetään aikaa korkeintaan $\mathcal{O}(|E|)$
- Kokonaisuudessaan aikaa siis kuluu $\mathcal{O}(|V| + |V| + |E|)$ eli $\mathcal{O}(|V| + |E|)$
- Tilavaativuus algoritmilla on $\mathcal{O}(|V|)$ sillä pahimmassa tapauksessa aloitussolmusta on kaari kaikkiin verkon solmuihin, ja tässä tapauksessa jono Q tulisi sisältämään kaikki verkon solmut; myös aputaulukot $color$, $tree$ ja $distance$ kuluttavat tilaa $\mathcal{O}(|V|)$

Leveyssuuntaisen läpikäynnin oikeellisuus

- Tarkastelemme nyt algoritmin oikeellisuuden osoittamista esimerkkinä ajattelutavasta, josta on apua hankalampien algoritmien ymmärtämisessä
- Väitämme, että algoritmin suorituksen jälkeen kaikilla solmuilla u pätee, että
  - jos u on saavutettavissa solmusta s, niin u on musta ja $distance[u]$ on lyhimmän polun pituus $s \rightsquigarrow u$
  - muuten u on valkoinen ja $distance[u] = \infty$.

- Värityksiä koskeva osa algoritmin toiminnasta on helppo todeta oikeaksi
- Algoritmista nähdään suoraan, että while-silmukassa pätee invariantti
  - jos solmu on valkoinen, se ei ole käynytkään jonossa
  - jos solmu on harmaa, se on parhaillaan jonossa
  - jos solmu on musta, se on poistettu jonosta
  - jos solmu on musta, sen viereiset solmut ovat harmaita tai mustia
- Erityisesti algoritmin päättyessä
  - harmaita solmuja ei ole, koska jono on tyhjä
  - lähtösolmu on musta
  - jos $s \rightsquigarrow u$ ja u olisi valkoinen, niin polku hyppäisi jossain kohdassa mustasta valkoiseksi

  $\Rightarrow$ kaikki saavutettavissa olevat solmut ovat mustia

- Tehdään toisaalta vastaoletus, että algoritmi värittää harmaaksi ainakin yhden solmun, joka ei ole saavutettavissa solmusta s
- Olkoon v näistä algoritmin suoritusjärjestyksessä ensimmäinen harmaaksi väritettävä
- Ennen kuin v voidaan värittää harmaaksi, algoritmin on pitänyt värittää harmaaksi jokin solmu u, jolla $v \in vierus[u]$
- Koska oletuksen mukaan v oli ensimmäinen ”väärin” väritetty, solmu u on saavutettavissa
- Mutta oletuksen $v \in vierus[u]$ mukaan myös v on nyt saavutettavissa; ristiriita
- Siis solmu väritetään algoritmin kuluessa harmaaksi, jos ja vain jos se on saavutettavissa solmusta s
- Koska kaikki harmaat solmut tulevat mustiksi ennen suorituksen loppua, niin värien osalta algoritmi toimii väitetyllä tavalla

- Polkujen pituuksien tarkastelemiseksi olkoon $D(u)$ lyhimmän polun pituus lähtösolmusta solmuun u
- Lisäksi merkitään $D(u) = \infty$, jos polkua ei ole
- Väitämme siis, että lopuksi $distance[u] = D(u)$ kaikilla u
- On helppo nähdä, että algoritmi säilyttää invariantin $distance[u] \geq D(u)$ kaikilla u
- Aluksi $distance[u] = \infty$ ja invariantti selvästi pätee
- Kun algoritmi myöhemmin päivittää $distance[v] = distance[u] + 1$, niin $(u, v) \in E$
- Tällöin $D(v) \leq D(u) + 1$, ja yhtäsuuruus pätee, jos jokin lyhin polku $s \rightsquigarrow v$ kulkee solmun u kautta
- Jos siis $distance[u] \geq D(u)$, niin $distance[v] = distance[u] + 1 \geq D(u) + 1 \geq D(v)$
- Tämä perustuu siihen, että algoritmin laskema arvo $distance[u]$ on jonkin polun pituus $s \rightsquigarrow u$

- Ongelmaksi jää osoittaa, että algoritmi todella löytää lyhimmän polun
- Algoritmin **BFS** tapauksessa tämä on melko suoraviivaista, koska (kuten kohta perustellaan) algoritmi löytää solmut arvon $D(u)$ mukaan kasvavassa järjestyksessä
- Myöhemmin kohtaamme vastaavan tilanteen painotetuissa verkoissa, jolloin dynamiikka on monimutkaisempi

- Analysoimme arvojen $distance[u]$ laskemista jakamalla suorituksen vaiheisiin: Vaihe k päättyy, kun viimeisen kerran muutetaan mustaksi solmu u, jolla $distance[u] = k - 1$

- Lisäksi sovimme, että vaihe 0 koostuu alustuksista ennen while-silmukan alkua

  Todistamme induktiolla arvon k suhteen, että vaiheen k päättyessä

  - jos $D(u) < k$, niin u on musta (eli poistunut jonosta)
  - jos $D(u) = k$, niin u on harmaa (eli parhaillaan jonossa)
  - jos $D(u) > k$, niin u on valkoinen (eli ei ole vielä ollut jonossa)
  - jos u on harmaa tai musta, niin $distance[u] = D(u)$

- Alustusten jälkeen väite selvästi pätee

- Oletetaan nyt, että väite pätee vaiheen k päättyessä (induktio-oletus)
- Siis jonossa on tasan ne solmut u, joilla $D(u) = k$
- Määritelmän mukaan $D(v) = k + 1$, jos ja vain jos
  - $v \in vierus[u]$ jollain u, jolla $D(u) = k$, ja
  - $v \notin vierus[w]$, jos $D(w) < k$
- Täsmälleen nämä solmut viedään jonoon vaiheen $k + 1$ aikana:
  - kaikki ehdon $D(u) = k$ täyttävät vieruslistat $vierus[u]$ käydään läpi
  - jos $v \in vierus[w]$ missä $D(w) < k$, niin induktio-oletuksen mukaan w on käynyt jonossa aiemmilla kierroksilla, jolloin v on muutettu harmaaksi
- Siis ehto pätee vaiheen $k + 1$ jälkeen

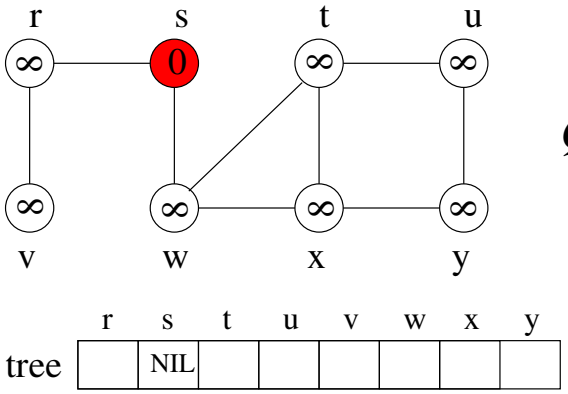
- Algoritmin päättyessä edellä todetun mukaan
  - kaikki lähtösolmusta saavutettavat solmut ovat mustia ja
  - kaikille mustille solmuille u pätee $distance[u] = D(u)$
- Toisaalta muut kuin saavutettavat solmut u ovat valkoisia, ja niillä on voimassa alkuasetus $distance[u] = \infty$
- Siis lopuksi $distance[u] = D(u)$ kaikilla u, kuten haluttiin
- Tarkempi tarkastelu osoittaa, että algoritmin toiminnan kannalta emme oikeastaan tarvitse harmaaksi merkitsemistä, vaan voisimme merkitä solmut suoraan mustiksi, kun niihin tullaan

- Algoritmi olisi nyt tällainen:

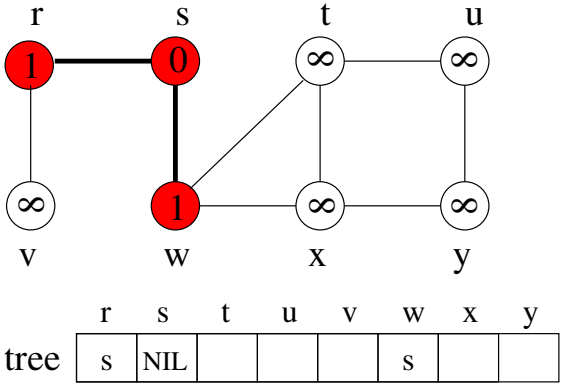
```
BFS2(G,s)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3       distance[u] = ∞
4       tree[u] = NIL
5   color[s] = black
6   distance[s] = 0
7   enqueue(Q,s)
8   while ( not empty(Q) )
9       u = dequeue(Q)
10      for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
11          if color[v]==white                   // solmua v ei vielä löydetty
12              color[v] = black
13              distance[v] = distance[u]+1
14              tree[v] = u
15              enqueue(Q,v)
```

- Käydään vielä yksi esimerkki läpi käyttäen tätä algoritmia (mustat solmut ovat värillisillä sivuilla punaisia)

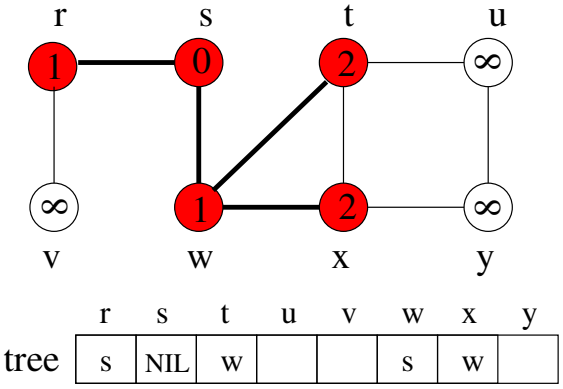
- Taulukon $tree$ alkio $tree[v]$ siis kertoo mistä solmusta lyhin polku lähtösolmusta solmuun v saapuu



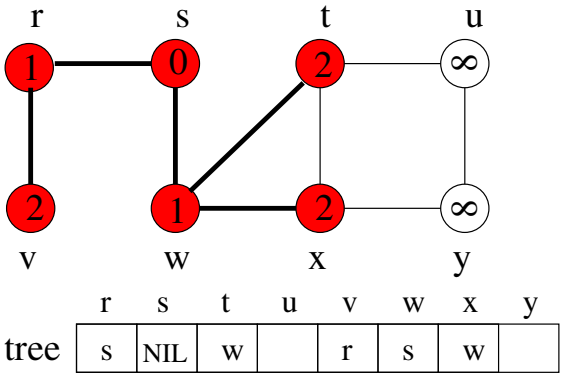
Q | s |



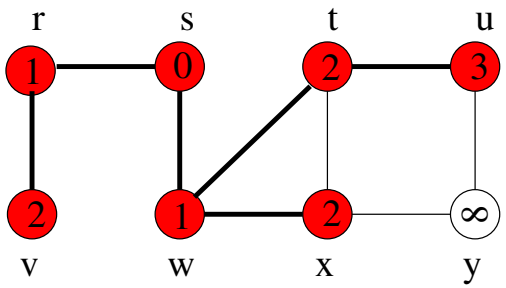
Q | w | r |



Q | r | t | x |

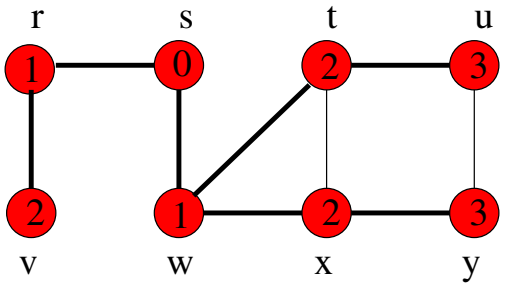


Q | t | x | v |



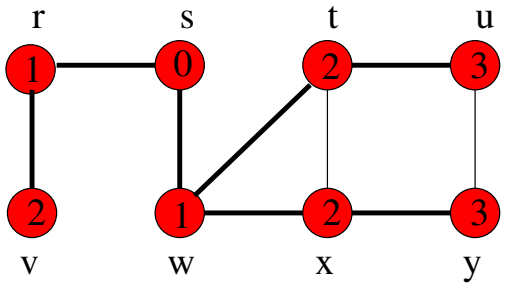
Q | x | v | u |

	r	s	t	u	v	w	x	y
tree	s	NIL	w	t	r	s	w	

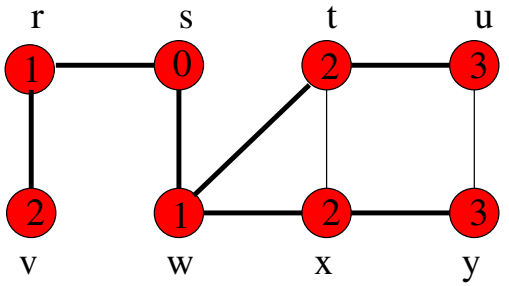


Q | v | u | y |

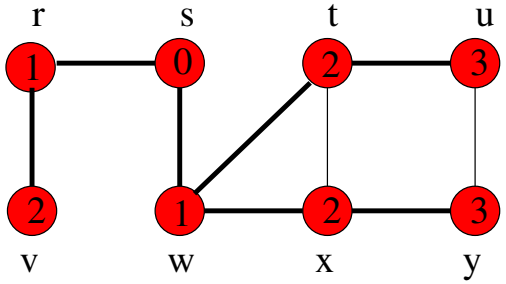
	r	s	t	u	v	w	x	y
tree	s	NIL	w	t	r	s	w	x



Q | u | y |



Q | y |



Q

Syvyyssuuntainen läpikäynti

- Toinen verkkojen perusläpikäyntitavoista on syvyyssuuntainen läpikäynti
- Strategiana on nyt edetä aloitussolmusta s yhtä polkua niin pitkälle kuin mahdollista
- Kun tullaan solmuun josta ei enää päästä uusiin, vielä tutkimattomiin solmuihin, peruutetaan tutkitulla polulla lähimpään sellaiseen solmuun josta lähtee vielä tutkimaton haara
- Näin löydetään kaikki solmusta s saavutettavissa olevat solmut
- Syvyyssuuntainen etsintä saadaan aikaan korvaamalla leveyssuuntaisen etsinnän jono pinolla
- Jos halutaan käydä läpi kaikki verkon solmut ja verkossa on solmuja jotka eivät ole saavutettavissa solmusta s, valitaan yksi saavuttamattomissa olevista solmuista ja käynnistetään uusi läpikäynti

- Myös syvyyssuuntainen läpikäynti värjää solmuja samalla tavalla kuin leveyssuuntaisessa läpikäynnissä:
  - solmut joita ei ole löydetty ovat valkoisia
  - kun solmu löydetään, se asetetaan harmaaksi
  - kun solmun kaikkien vierussolmujen käsittelystä on palattu, tulee solmusta musta
- Harmaa väri siis tarkoittaa, että solmu on jo löydetty, mutta sen käsittely ei ole vielä kokonaisuudessaan ohi

- Seuraavassa algoritmi, joka selvittää aloitussolmusta s saavutettavissa olevat solmut

```
DFS(G,s)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3   DFS-visit(G,s)

DFS-visit(G,u)
4   color[u] = gray
5   for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
6       if color[v]==white                   // solmua v ei vielä löydetty
7           DFS-visit(G,v)
8   color[u] = black
```

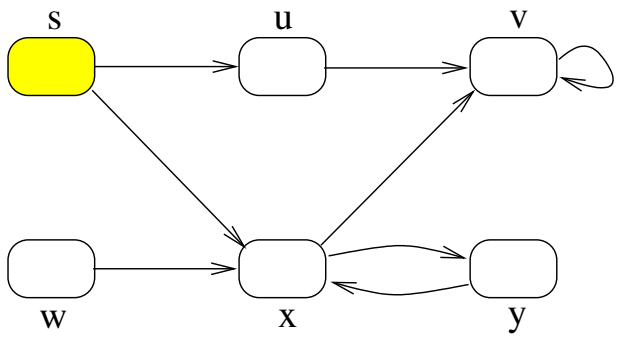
- Harmaa ei ole tässäkään tarpeen algoritmin toiminnan kannalta, eli solmu voitaisiin värjätä heti mustaksi

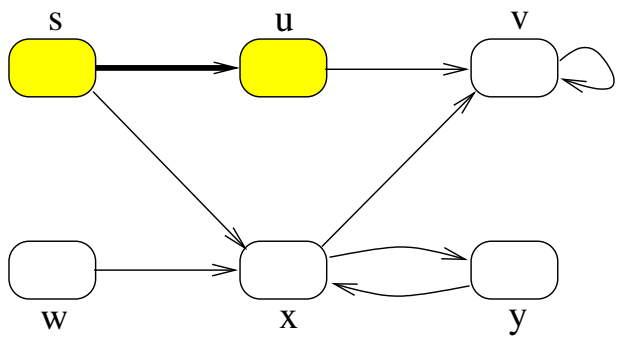
- Tarvitsemme harmaata väriä kohta esitettävässä syvyyssuuntaisen läpikäynnin sovelluksessa (syklittömyyden tarkastus), joten jätämme sen algoritmiin

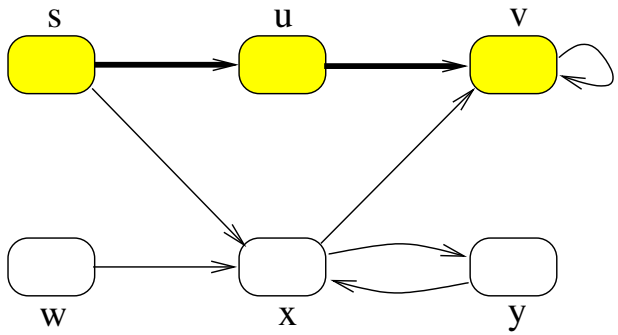
- Tulemme taas osoittamaan tarkasti algoritmin oikeellisuuden. Sitä varten tarvitsemme lisämuuttujia, mutta palaamme siihen myöhemmin

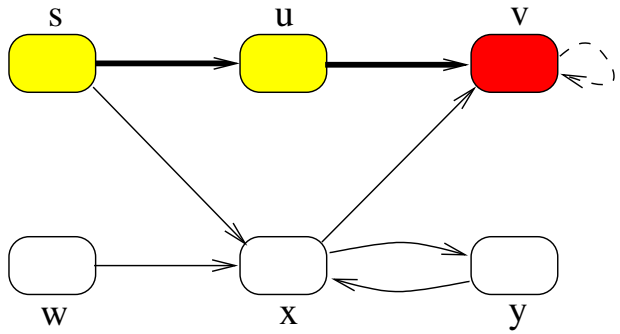
- Toimintaperiaate:
  - alustusvaiheessa kaikki solmut merkataan löytymättömiksi eli valkoisiksi
  - läpikäynti aloitetaan kutsumalla **DFS-visit** aloitussolmulle s
  - kun läpikäynti etenee solmuun, merkataan että solmu on löydetty ja että sen käsittely on kesken eli solmu muuttuu harmaaksi (rivi 4)
  - jokaiselle solmun vierussolmulle jota ei ole vielä löydetty, eli valkoisille solmuille, kutsutaan rekursiivisesti **DFS-visit**:iä (rivit 5-7)
  - kun kaikki vierussolmut on käsitelty, merkataan solmu käsitellyksi eli mustaksi (rivi 8) ja rekursiivinen funktio päättyy

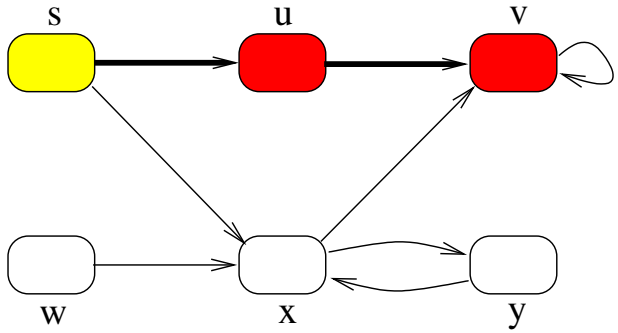
- Esimerkki algoritmin toiminnasta seuraavalla sivulla. Värillisessä kuvassa harmaat solmut ovat keltaisia ja mustat punaisia

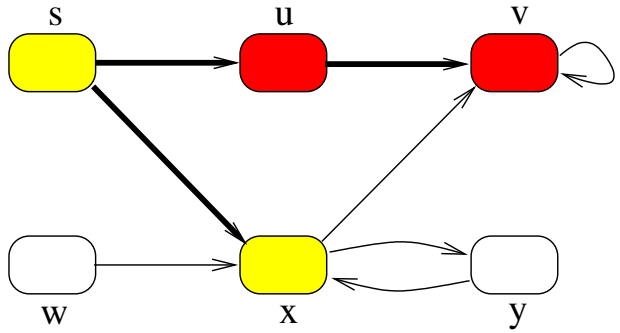

s
u
v
w
x
y

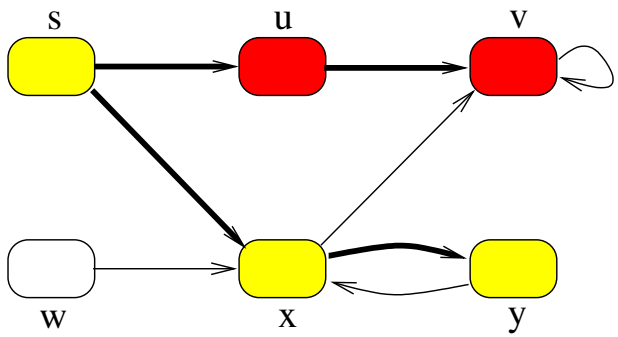

s
u
v
w
x
y

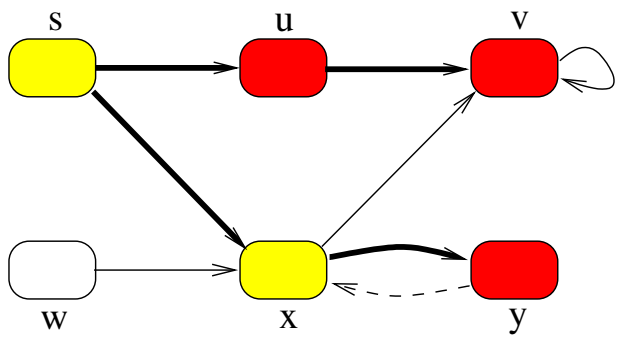

s
u
v
w
x
y

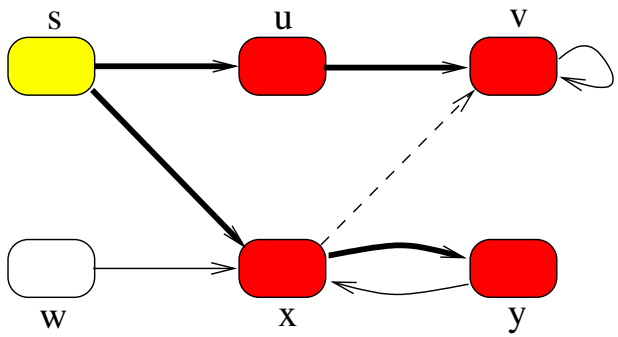

s
u
v
w
x
y

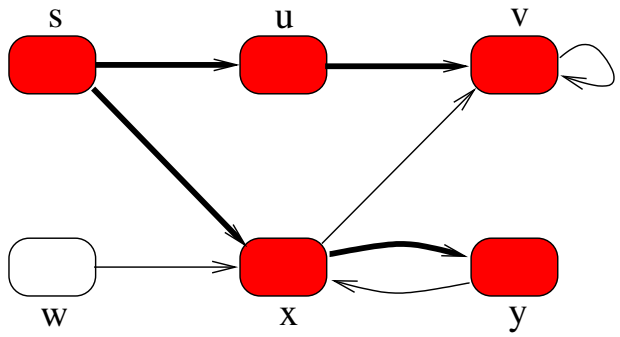

s
u
v
w
x
y


s
u
v
w
x
y


s
u
v
w
x
y


s
u
v
w
x
y


s
u
v
w
x
y


s
u
v
w
x
y

- Algoritmi siis merkkaa ensin $color[v] =$ gray ja lopulta $color[v] =$ black kaikille aloitussolmusta s saavutettavissa oleville solmuille
  - solmu v pysyy harmaana niin kauan kuin haku etenee solmuissa, jotka ovat v:stä saavutettavissa
  - kun kaikki v:stä saavutettavat solmut on löydetty ja käsitelty, värjätään v mustaksi
- Kaaret joita pitkin läpikäynti on edennyt, eli syvyyssuuntaispuun kaaret, on kuvassa paksunnettu
- Syvyyssuuntainen läpikäynti siis löysi solmut seuraavassa järjestyksessä: s, u, v, x, y
- Solmua w ei saavuteta aloitussolmusta, eli lopussa edelleen $color[w] =$ white
- Koko verkolle tehtävä syvyyssuuntainen läpikäynti tapahtuu seuraavasti:

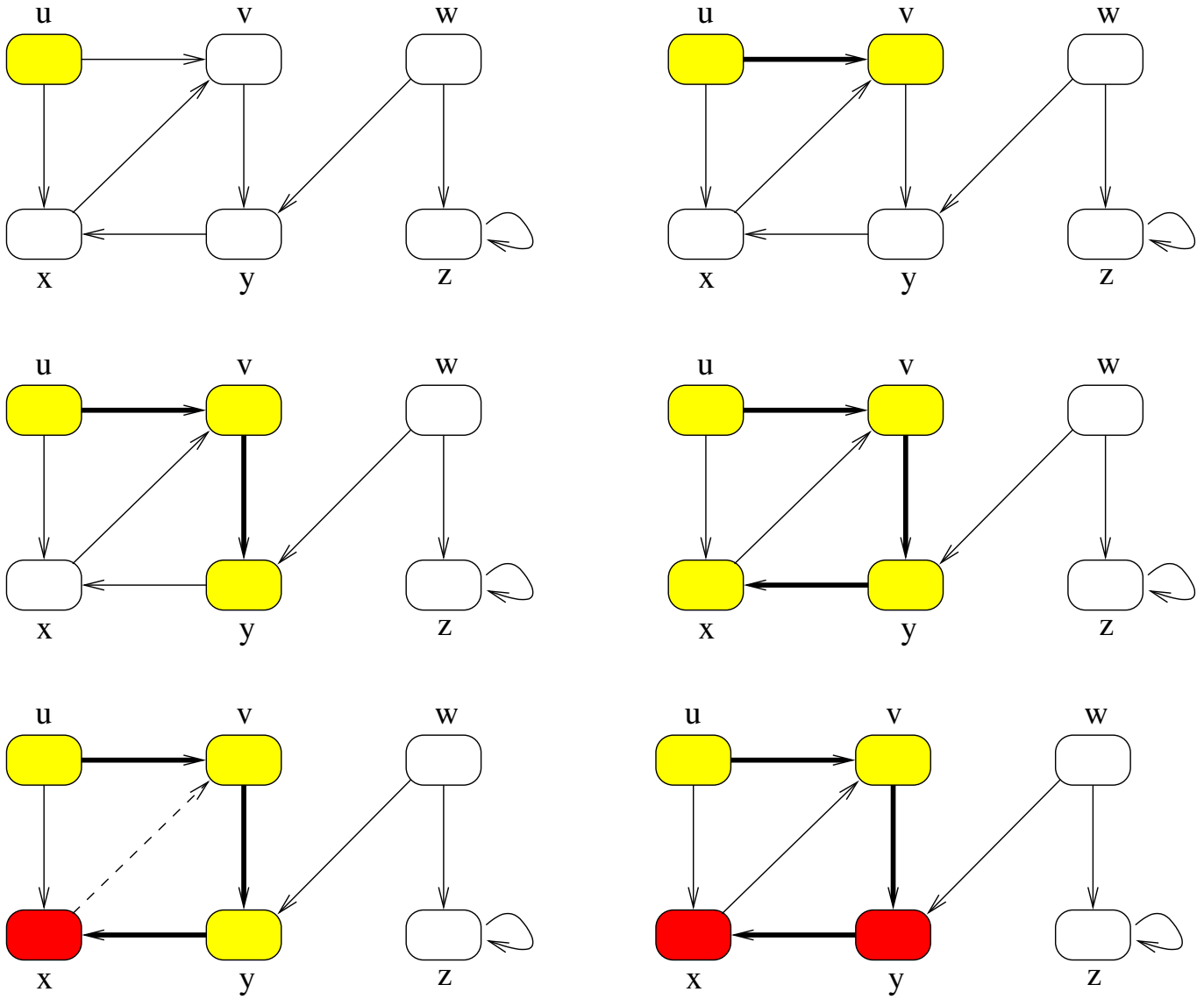
```
DFS-all(G)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3   for jokaiselle solmulle u ∈ V
4       if color[u]==white
5           DFS-visit(G,u)
```

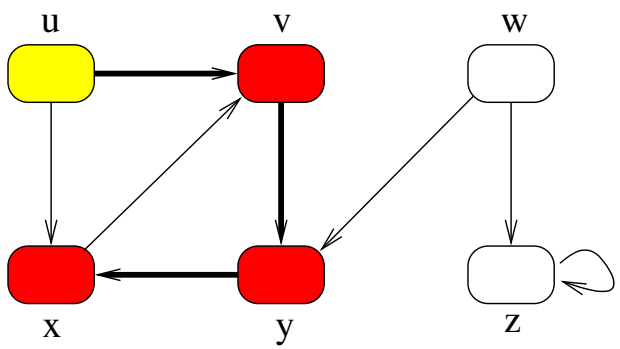
- Koko verkolle tehtävän syvyyssuuntaisen läpikäynnin aikana muodostuu verkon syvyyssuuntainen metsä (engl. depth-first forest). Tämä on kokoelma syvyyssuuntaispuita (engl. depth-first tree). Kukin puu joka koostuu niistä kaarista, joita pitkin läpikäynti eteni aiemmin löytymättömiin solmuihin

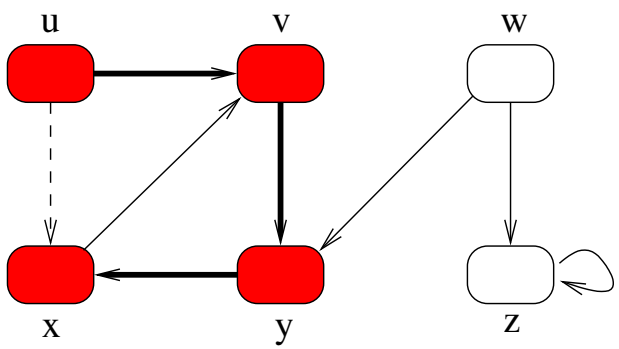
- Syvyyssuuntainen metsä ei ole yksikäsitteinen, vaan riippuu valitusta solmujen ja vieruslistojen järjestyksestä

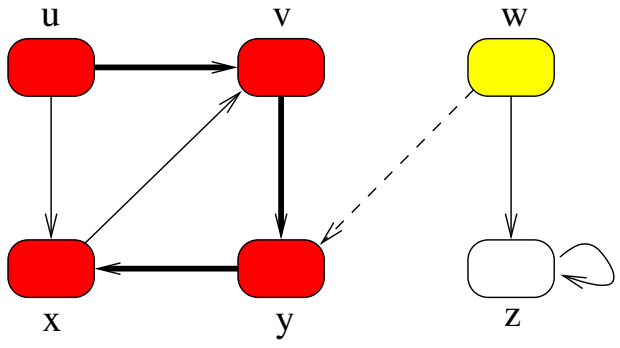
- Kuten leveyssuuntaisen läpikäynnin yhteydessä, syvyyssuuntaispuun kaaret olisi tarvittaessa helppo kirjata algoritmin yhteydessä esim. erilliseen taulukkoon $tree$, johon asetettaisiin $tree[v] = u$ jos läpikäynti eteni solmusta u solmuun v

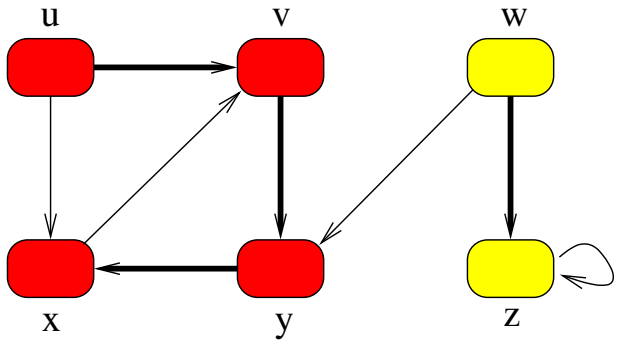
- Seuraavassa esimerkki algoritmin toiminnasta:

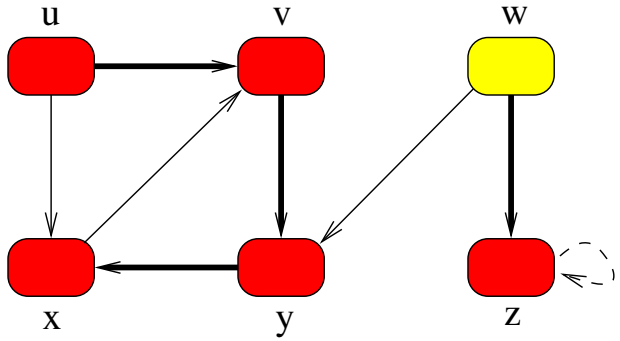


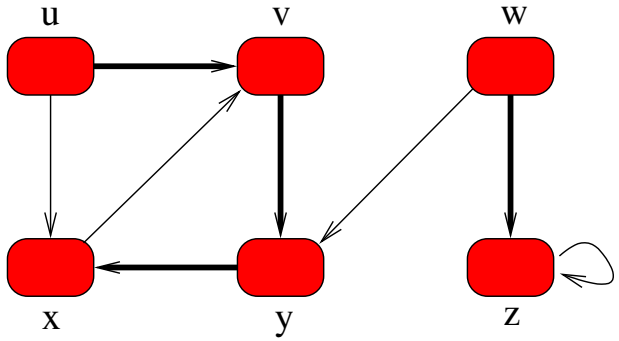

u
v
w
x
y
z


u
v
w
x
y
z


u
v
w
x
y
z


u
v
w
x
y
z


u
v
w
x
y
z


u
v
w
x
y
z

- Syvyyssuuntaisen läpikäynnin vaativuus
  - taulukon $color$ alustus vie aikaa $\mathcal{O}(|V|)$
  - operaatio **DFS-visit** kutsutaan (korkeintaan) kerran jokaiselle solmulle, sillä operaatiota kutsutaan ainoastaan solmuille v, joilla $color[v] =$ white, ja heti kutsun jälkeen asetetaan $color[v] =$ gray eikä väri enää muutu missään vaiheessa valkoiseksi
  - yhteensä **DFS-visit**-operaation kutsuja siis enintään $|V|$ kappaletta
  - **DFS-visit**:in for-lause käy jokaisen solmun vieruslistan läpi, eli for-osa toistetaan yhteensä $|E|$ kertaa
  - kokonaisuudessaan aikaa siis kuluu $\mathcal{O}(|V| + |V| + |E|)$ eli $\mathcal{O}(|V| + |E|)$
  - tilavaativuus algoritmilla on $\mathcal{O}(|V|)$ sillä pahimmassa tapauksessa aloitussolmusta pääsee yhtä polkua pitkin kaikkiin muihin solmuihin ja tällöin sisäkkäisiä rekursiivisia **DFS-visit**-kutsuja tehdään $|V|$ kappaletta; myös aputaulukko vie tilaa $\mathcal{O}(|V|)$
- Aivan kuten leveyssuuntaisen läpikäynnin tapauksessa myös syvyyssuuntaisen läpikäynnin algoritmi toimii sellaisenaan niin suunnatuilla kuin suuntaamattomillakin verkoilla

Kaarten luokittelu

- Verkon kaaret voidaan luokitella neljään eri luokkaan sen perusteella, miten kaaret käyttäytyvät syvyyssuuntaisen läpikäynnin suhteen

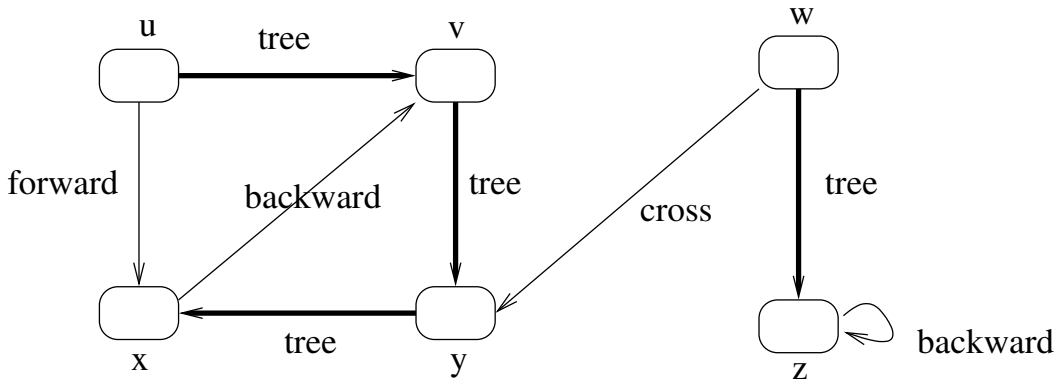
- Läpikäynti etenee puunkaaria (engl. tree arc) pitkin uusiin vielä löytymättömiin solmuihin, eli puunkaari kohdistuu valkoiseen solmuun

- Takautuva kaari (engl. backward arc) kohdistuu taaksepäin syvyyssuuntaispuussa, eli takautuva kaari ilmenee kun algoritmi yrittää edetä solmuun joka on jo löydetty, mutta jonka käsittely on kesken, eli takautuva kaari kohdistuu harmaaseen solmuun

- Etenevä kaari (engl. forward arc) kohdistuu eteenpäin syvyyssuuntaispuussa, eli etenevä kaari ilmenee kun algoritmi yrittää edetä solmuun, joka on nykyisen solmun jälkeläinen mutta löydetty ja käsitelty (eli musta) jo jotain nykyisen solmun muuta jälkeläistä tutkittaessa

- Poikittaiskaari (engl. cross arc) kulkee jo löydettyyn eli mustaan solmuun joko
  - kahden eri syvyyssuuntaispuun välillä, tai
  - syvyyssuuntaispuun sisällä sellaisten solmujen välillä joista kumpikaan ei ole toisensa jälkeläinen puussa

- Alla edellisen esimerkin verkon kaaret luokiteltuina



- Läpikäynti etenee aluksi valkoisia solmuja pitkin reittiä $u \to v \to y \to x$, kaikki nämä harmaasta valkoiseen solmuun kohdistuvat ovat puunkaaria

- Kun ollaan solmussa x, ainoa kaari kohdistuu harmaaseen solmuun v, joka siis on jo löydetty mutta jonka vierussolmuja ei ole käsitelty loppuun, $x \to v$ on siis takautuva kaari

- Kun läpikäynti on peruuttanut takaisin solmuun u, tutkitaan kaari $u \to x$ joka kohdistuu u:n seuraajaan syvyysuuntaispuussa, $u \to x$ siis on etenevä kaari

- Kaari $w \to y$ on kahden eri syvyyssuuntaispuussa sijaitsevan solmun välinen, eli kyseessä on poikittaiskaari

- Huom: Kaarten luokittelu riippuu siitä, mistä solmusta läpikäynti aloitetaan
- Suuntaamattomassa verkossa kaaret jaetaan puukaariin ja takautuviin kaariin: takautuvia ja eteneviä ei voi erottaa, ja poikittaisia ei voi esiintyä

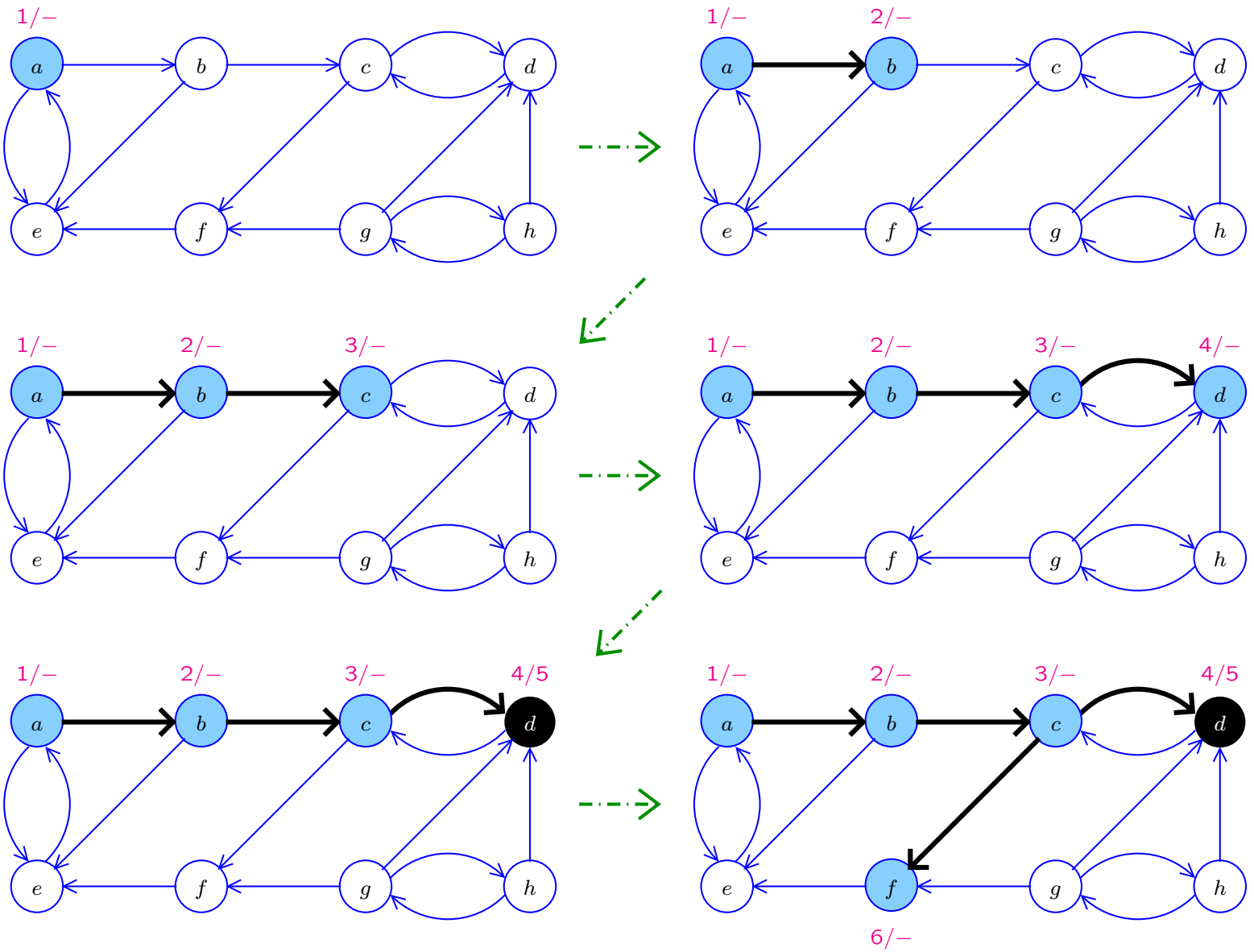
Valkopolkulause

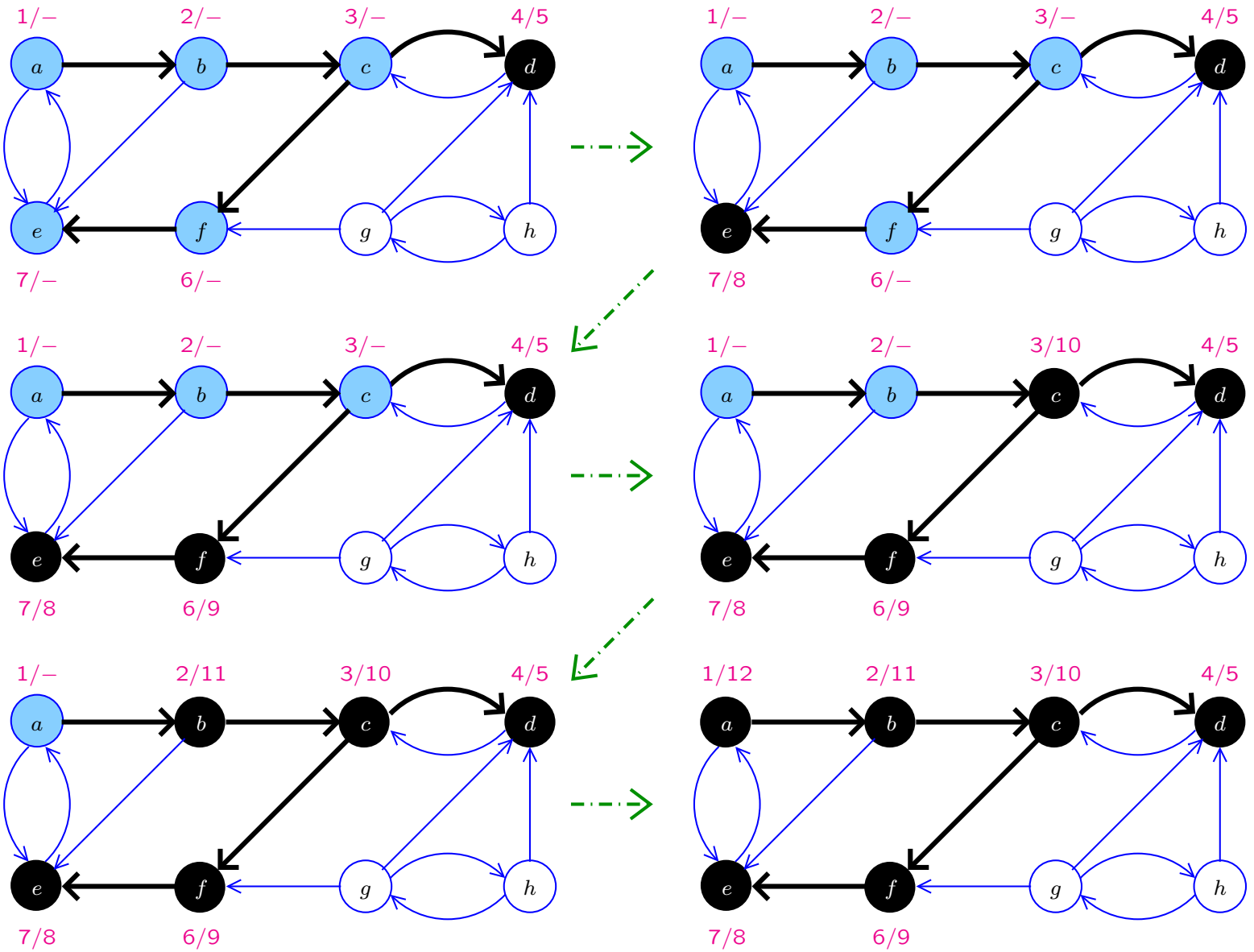
- Algoritmin toiminnan tarkempaa tarkastelua varten käytämme siitä versiota, johon on lisätty kaksi apumuuttujaa
  - löytymishetki $d[u]$ (discovery): milloin solmu muuttui harmaaksi
  - päättymishetki $f[u]$ (finish): milloin solmu muuttui mustaksi
- Aikaa mitataan kaikkiaan tehtyjen värinmuutosten määrällä

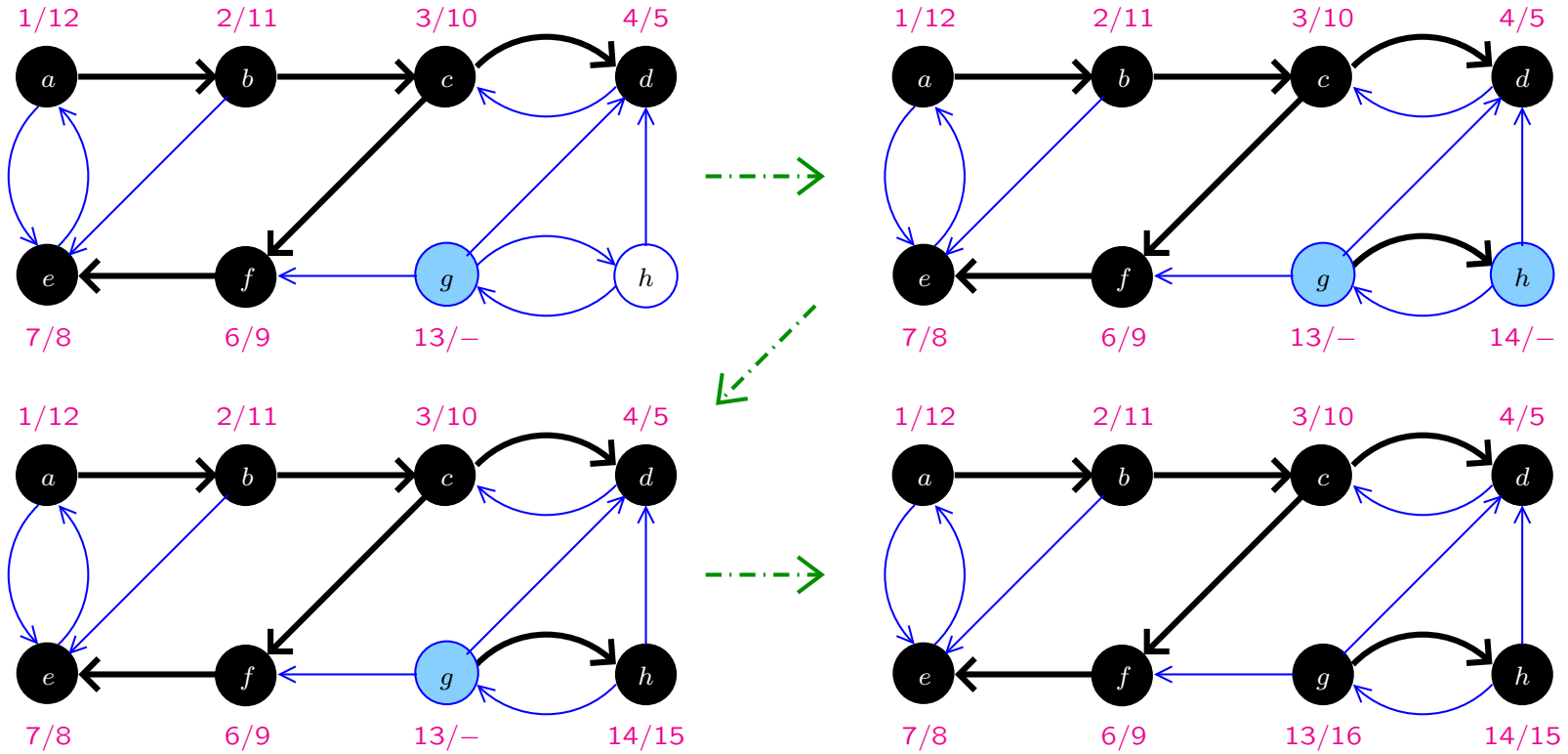
- Algoritmi on nyt seuraavanlainen:

```
DFS-all2(G)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3   time = 0
4   for jokaiselle solmulle u ∈ V
5       if color[u]==white
6           DFS-visit2(G,u)

DFS-visit2(G,u)
7   color[u] = gray
8   time = time + 1
9   d[u] = time
10  for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
11      if color[v]==white                   // solmua v ei vielä löydetty
12          DFS-visit2(G,v)
13  color[u] = black
14  time = time +1
15  f[u] = time
```

- Esimerkki suunnatusta tapauksesta seuraavilla kolmella sivulla: solmujen d- ja f-arvot on merkitty muodossa $d[u]/f[u]$; edetyt kaaret (eli puunkaaret) on vahvennettu


1/−
a
b
c
d
e
f
g
h
1/−
2/−
a
b
c
d
e
f
g
h
1/−
2/−
3/−
a
b
c
d
e
f
g
h
1/−
2/−
3/−
4/−
a
b
c
d
e
f
g
h
1/−
2/−
3/−
4/5
a
b
c
d
e
f
g
h
1/−
2/−
3/−
4/5
a
b
c
d
e
f
g
h
6/−


1/−
2/−
3/−
4/5
a
b
c
d
e
f
g
h
7/−
6/−
1/−
2/−
3/−
4/5
a
b
c
d
e
f
g
h
7/8
6/−
1/−
2/−
3/−
4/5
a
b
c
d
e
f
g
h
7/8
6/9
1/−
2/−
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
1/−
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9


1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
13/−
1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
13/−
14/−
1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
13/−
14/15
1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
13/16
14/15

- **DFS-visit2**:ta tarkastelemalla näemme suoraan seuraavan "sulkumerkkiteoreeman"

- **Lause 8.1**: Mille tahansa syvyysuuntaispuun solmuille u ja v pätee jokin seuraavista:

  1. $d[u] < d[v] < f[v] < f[u]$, ja solmu v on solmun u jälkeläinen
  2. $d[v] < d[u] < f[u] < f[v]$, ja solmu u on solmun v jälkeläinen
  3. $d[u] < f[u] < d[v] < f[v]$ tai $d[v] < f[v] < d[u] < f[u]$, ja solmuista kumpikaan ei ole toisen jälkeläinen.

□

- Lause sanoo, että kutsujen alku- ja loppumisajat vastaavat alku- ja loppusulkuja hyvinmuodostetussa lausekkeessa

- Seuraava valkopolkulause (White-path Theorem) pätee suunnatuissa ja suuntaamattomissa verkoissa

- Se tulee käyttöön myöhemmin

- **Lause 8.2**: Solmu v tulee solmun u jälkeläiseksi syvyyssuuntaisessa metsässä, jos ja vain jos kutsun **DFS-visit**(G, u) alkaessa on olemassa pelkistä valkoisista solmuista koostuva polku $u \rightsquigarrow v$.

- **Todistus:**

  $\Rightarrow$: Kun v on u:n jälkeläinen, olkoon $(u, w_1, \ldots, w_k, v)$ puukaaria pitkin kulkeva polku $u \rightsquigarrow v$. Sulkumerkkiteoreeman mukaan $d[u] < d[w_1] < \ldots < d[w_k] < d[v]$, joten kutsun **DFS-visit**(G, u) alkaessa kaikki polun solmut ovat valkoisia

$\Leftarrow$: Olkoon $(u, w_1, \ldots, w_k, v)$ valkoisista solmuista koostuva polku, kun **DFS-visit**(G, u) alkaa. Tehdään vastaoletus, että ainakin yksi polun solmuista ei tule u:n jälkeläiseksi. Olkoon v näistä ensimmäinen. Olkoon lisäksi w solmua v edeltävä solmu polulla. Nyt siis w on joko u tai u:n aito jälkeläinen. Sulkumerkkiteoreeman mukaan, kun w on u:n jälkeläinen (ottamalla huomioon erikoistapaus $w = u$), pätee

$$d[u] \leq d[w] < f[w] \leq f[u].$$

Koska v:hen tullaan u:n jälkeen, mutta ennen kuin w on käsitelty loppuun, niin $d[u] < d[v] < f[w]$. Mutta tästä siis seuraa, että $d[u] < d[v] < f[u]$, eli sulkumerkkiteoreeman mukaan vain tapaus

$$d[u] < d[v] < f[v] < f[u]$$

on mahdollinen. Mutta tämä tarkoittaa, että v on sittenkin u:n jälkeläinen; ristiriita. □

- Jos et ymmärtänyt tätä valkopolku-osuutta, niin suurta vahinkoa ei ole tapahtunut...

Verkon syklittömyyden tarkastus

- Joskus voi olla tarvetta testata onko annetussa suunnatussa verkossa sykliä, eli onko kyseessä syklitön suunnattu verkko

- Pienellä muutoksella voimme käyttää syvyyssuuntaisen läpikäynnin algoritmia syklittömyyden testaamiseen:
  - oletetaan että ollaan tutkimassa solmun u vieruslistaa $vierus[u]$
  - jos vieruslistalta löytyy solmu v jolle $color[v] =$ gray, tiedämme että v:n käsittely on kesken, ja
  - solmusta v johtaa polku solmuun u
  - nyt siis on olemassa polku $v \rightsquigarrow u \rightarrow v$, eli verkossa on sykli
  - syklin olemassaolo siis havaitaan jos syvyyssuuntaispuussa on takautuva kaari

- Tarvittava muutos on siis testi löytyykö tutkittavan solmun vieruslistalta solmu v, jolle $color[v] =$ gray

- Seuraavalla sivulla algoritmi palauttaa true jos verkossa on sykli ja muuten false

```
DFS-cycles(G)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3   for jokaiselle solmulle u ∈ V
4       if color[u]==white
5           if DFS-search-cycles(G,u) == true
6               return true
7   return false

DFS-search-cycles(G,u)
9   color[u] = gray
10  for jokaiselle solmulle v ∈ vierus[u]   // kaikille u:n vierussolmuille v
11      if color[v] == gray                 // löytyi sykli, voidaan lopettaa
12          return true
13      if color[v]==white                  // solmua v ei vielä löydetty
14          if DFS-search-cycles(G,v) == true
15              return true
16  color[u] = black
17  return false
```

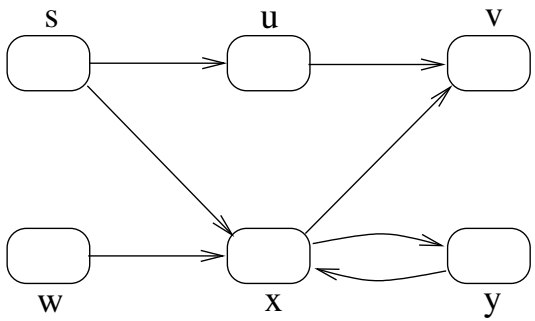
- Jos siis tutkittaessa solmun u vieruslistaa $vierus[u]$ löytyy solmu v jolle $color[v] =$ gray, tiedämme että v:n käsittely on kesken, ja solmusta v johtaa polku solmuun u eli on olemassa polku $v \rightsquigarrow u \rightarrow v$, eli verkossa on sykli

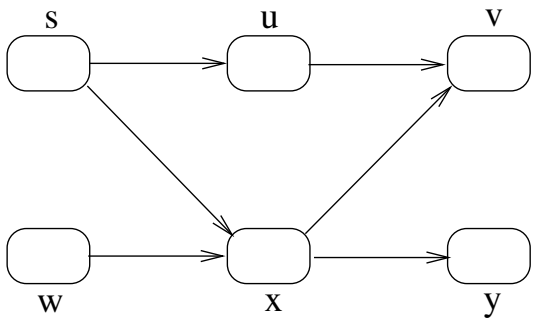
- Perustellaan seuraavassa vielä asia toisinpäin eli, jos verkossa on sykli, algoritmi huomaa syklin olemassaolon
  - Oletetaan, että verkossa on sykli ja olkoon v läpikäynnissä ensimmäisenä vastaantuleva syklin solmu
  - Koska v on osa sykliä, on olemassa solmu u jolle $v \rightsquigarrow u \rightarrow v$, eli u on syklissä oleva solmu josta on kaari v:hen
  - Kun läpikäynti tulee solmuun v, ei oletuksen mukaan u:ta eikä muita syklin solmuja ole vielä löydetty ja koska $v \rightsquigarrow u$, niin läpikäynti etenee solmuun u
  - Kun u:n vierussolmuja tutkitaan, havaitaan, että u:sta on kaari harmaaseen solmuun v, eli algoritmi löytää syklin

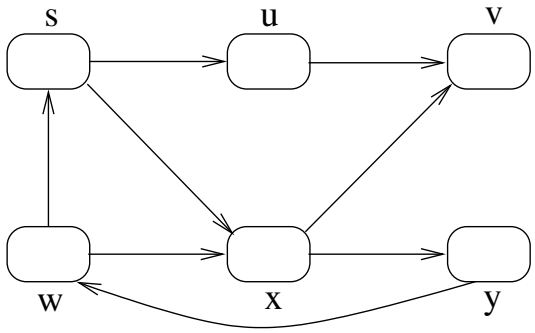
- Algoritmi siis löytää syklin jos ja vain jos verkossa on sykli, eli algoritmi toimii oikein

- Aika- ja tilavaativuus algoritmilla on tietenkin sama kuin modifioimattomalla syvyyssuuntaisella läpikäynnillä

- Esim: miten syklien etsintä toimisi seuraavissa verkoissa?

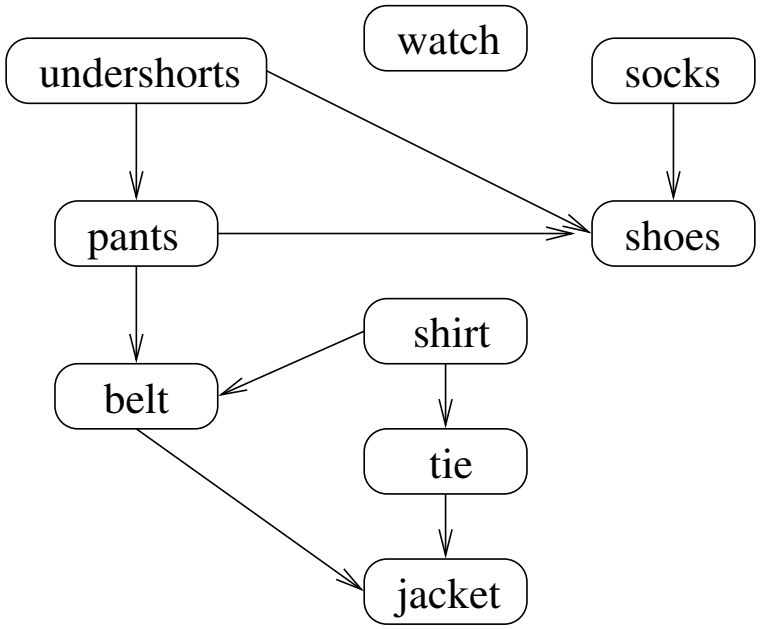






Topologinen järjestäminen

- Syklittömien suunnattujen verkkojen (engl. Directed Acyclic Graph, DAG) avulla voimme kuvata tapahtumien välisiä riippuvuuksia
- Alla oleva verkko sisältää pukeutumiseen kannalta oleelliset riippuvuudet:

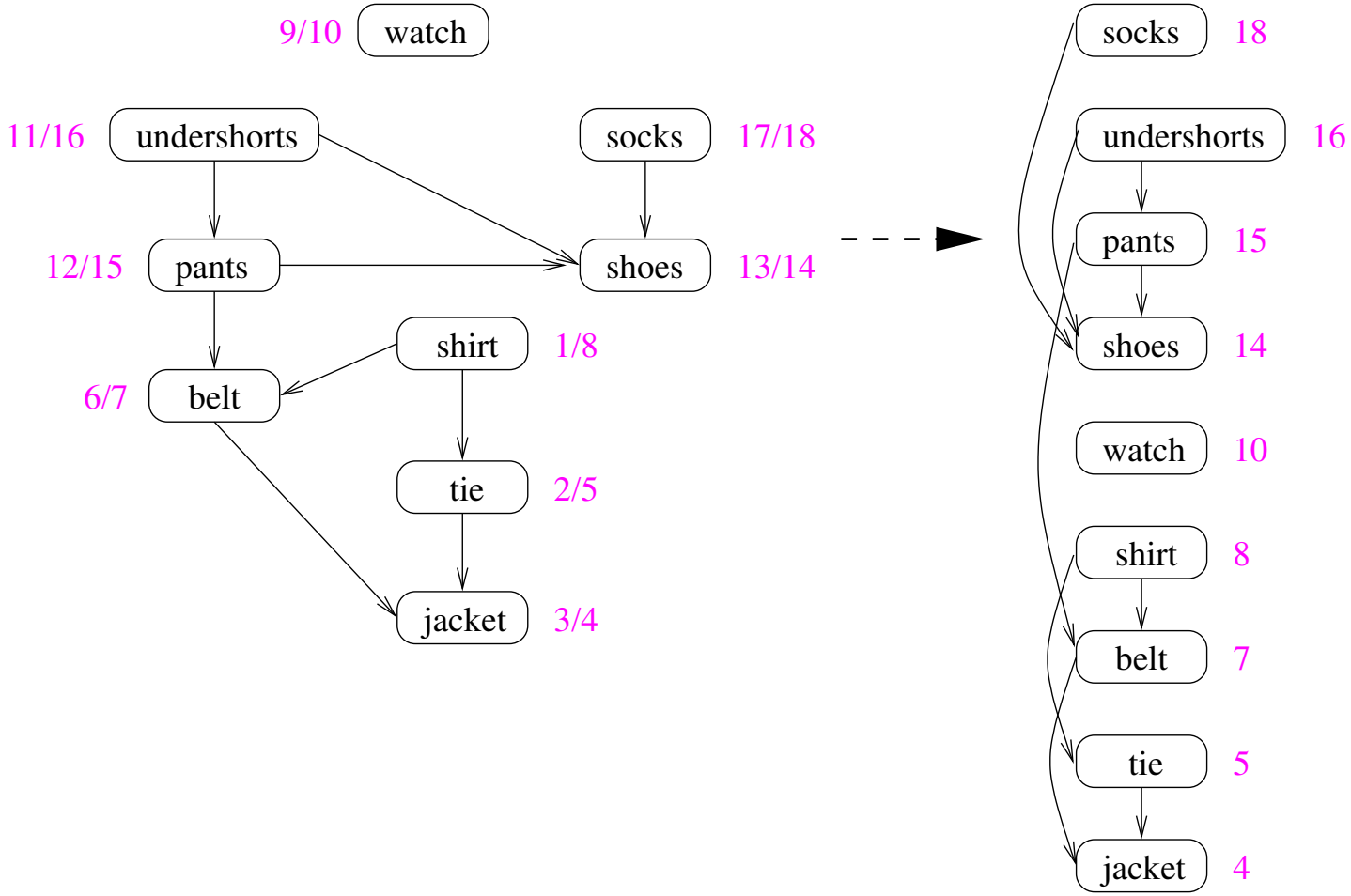


- Eli esim. sukat on laitettava ennen kenkiä koska on kaari socks $\rightarrow$ shoes
- Kello voidaan laittaa käteen missä vaiheessa tahansa koska sillä ei ole mitään riippuvuutta

- Herää kysymys voisimmeko järjestää asiat sellaiseen lineaariseen järjestykseen että voisimme pukea vaatekappaleen kerrallaan siten että mikään riippuvuuksista ei rikkoudu, eli
  - jos riippuvuusverkossa on kaari $u \to v$, tulee u järjestyksessä ennen v:tä
- Tällaista järjestystä kutsutaan topologiseksi järjestykseksi
- Asia hoituu helposti käyttäen apuna syvyyssuuntaista läpikäyntiä

  Topological-Sort(G)
  kutsutaan DFS-all(G)
  kun solmun v käsittely on ohi eli solmu muuttuu mustaksi
  lisätään se pinoon P
  return P
- Operaation jälkeen solmut ovat pinossa P järjestettynä siten että pinon päällä on viimeiseksi käsitelty solmu, eli solmu v_n, johon ei varmuudella kohdistu yhtään kaarta eli jolla ei ole yhtään riippuvuutta
- Pinossa toisena on solmu v_{n-1} johon on olemassa kaari eli riippuvuus korkeintaan solmusta v_n, jne.
- Solmut ovat siis pinossa P topologisessa järjestyksessä

- Seuraavassa esimerkkimme topologisesti järjestettynä. Solmujen yhteyteen on merkitty solmujen d- ja f-arvot on merkitty muodossa $d[u]/f[u]$; solmujen järjestys pinossa siis on käänteinen f-numerojärjestys ja f-arvot on merkitty kuvaan



- Perustellaan algoritmin oikeellisuus vielä hieman tarkemmin

- Algoritmi toimii oikein, jos millekään solmuille v, w, missä v on algoritmin tuottamassa järjestyksessä solmun w edellä, ei ole olemassa kaarta $w \to v$

- Oletetaan, että kaari $w \to v$ on olemassa, ja näytetään, että tästä seuraa ristiriita

  koska v on solmua w edellä algoritmin tuottamassa järjestyksessä, sen käsittely on päättynyt myöhemmin kuin w:n käsittely

  – jos w:n käsittely olisi aloitettu v:n löytymisen jälkeen, olisi verkossa polku $v \rightsquigarrow w$ sekä kaari $w \to v$ eli verkossa olisi sykli. Tämä on ristiriita, sillä oletus on, että verkko on syklitön

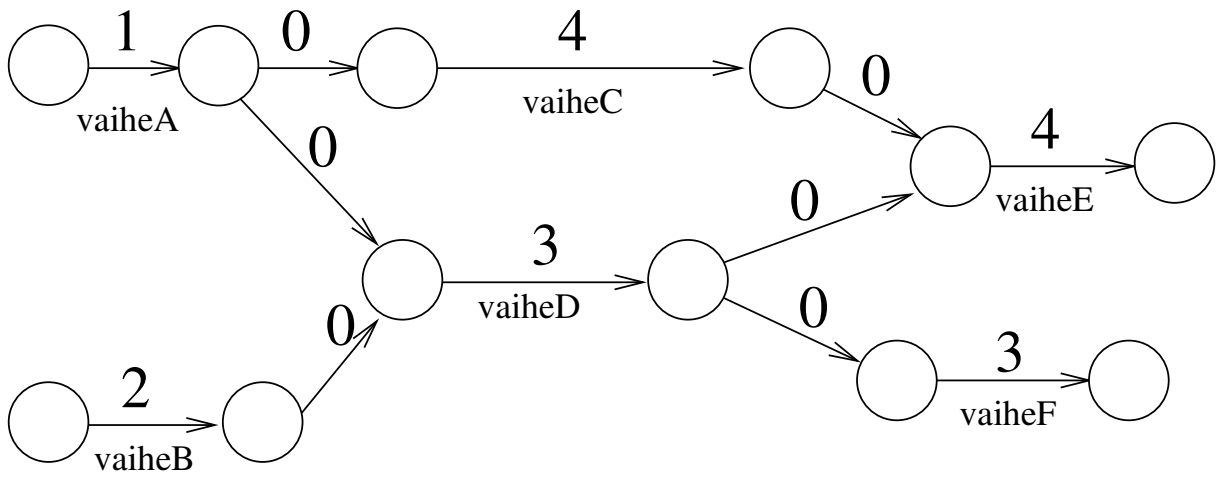
  – jos w:n käsittely olisi aloitettu ennen v:n löytymistä, olisi läpikäynti kaaren $w \to v$ seurauksena edennyt solmuun v ja v:n käsittelyn olisi täytynyt päättyä ennen w:n käsittelyn päättymistä. Tämä taas on ristiriita solmujen oletetun järjestyksen takia

- On siis osoitettu, että kaaren $w \to v$ olemassaolo johtaisi ristiriitaan, eli topologisen järjestyksen rikkovaa kaarta ei voi olla olemassa, eli topologinen järjestäminen toimii oikein

Topologisen järjestämisen sovellus: kriittiset työvaiheet

- Ohjelmistoprojektissa on tiettyjä määräaikoja eli deadlineja
  - dokumenttien ja komponenttien deadlinet, katselmuksia, asiakasdemoja
- Deadlineilla on osittainen järjestys:
  - käyttöliittymä on saatava valmiiksi ennen asiakasdemoa
  - käyttöliittymää ei voida aloittaa ennen kuin tietokantaliittymä on valmis
  - tietokantaliittymää voi testata käyttöliittymästä riippumatta
- Eri vaiheilla on kestoja
  - käyttöliittymän aloittamisesta sen valmistumiseen kuluu (arviolta) 4 viikkoa
- Ero topologiseen järjestämiseen:
  - lisätään eri vaiheille kestot ja sallitaan eri vaiheiden tekeminen rinnakkain
- Mallissa tehdään yksinkertaistuksia, eikä oteta huomioon esim. että
  - joitakin vaiheita ei voida tehdä yhtä aikaa, koska käytettävissä on vain yksi henkilö, joka pystyy tekemään niitä
  - joitakin vaiheita ei voida tehdä yhtä aikaa, mutta niiden keskinäisellä järjestyksellä ei ole merkitystä

- Ongelma: halutaan selvittää, kauanko projektiin vähintään kuluu aikaa
- Ongelma mallinnetaan suunnattuna verkkona:
  - Jokaisesta vaiheesta tehdään kaari, jonka päätepisteinä ovat vaiheen alku ja loppu ja painona vaiheen kesto
  - Jos kahdella vaiheella määrätty järjestys, tehdään kaari, jonka lähtösolmuna on ensimmäisen vaiheen loppu ja maalisolmuna jälkimmäisen vaiheen alku
  - Lasketaan tämän verkon pisin polku (siis painavin polku)
- Esim.
  - vaiheA kestää kuukauden, vaiheB 2 kuukautta, vaiheC 4 kuukautta, ...
  - vaiheA:lla ja vaiheB:llä ei riippuvuutta, vaiheA tehtävä ennen vaiheC:tä, ...



- Talletetaan kustakin solmusta lähtevän painavimman polun paino taulukkoon $heaviest$

- Solmusta u alkavan polun suurin mahdollinen paino $heaviest[u]$ voidaan määritellä seuraavasti:
  - $heaviest[u] = 0$, jos solmusta ei ole kaarta eteenpäin
  - muuten paino on

$$heaviest[u] = \max\{w(u,v) + heaviest[v] \mid \text{solmusta } u \text{ on solmuun } v \text{ kaari, jonka paino on } w(u,v)\}$$

- Laskusääntöä ei voi käyttää, jos verkossa on sykli, mutta tällöin projekti on myös mahdoton

- Solmun u luku $heaviest[u]$ voidaan laskea vasta sitten, kun on laskettu sen kaikkien seuraajasolmujen v luvut $heaviest[v]$

- Solmut on siis käsiteltävä käänteisessä topologisessa järjestyksessä

- Solmun u luku $heaviest[u]$ voidaan laskea sillä hetkellä, kun u viedään topologisen järjestyksen laskevassa algoritmissa pinoon P, eli kun solmu värjätään mustaksi

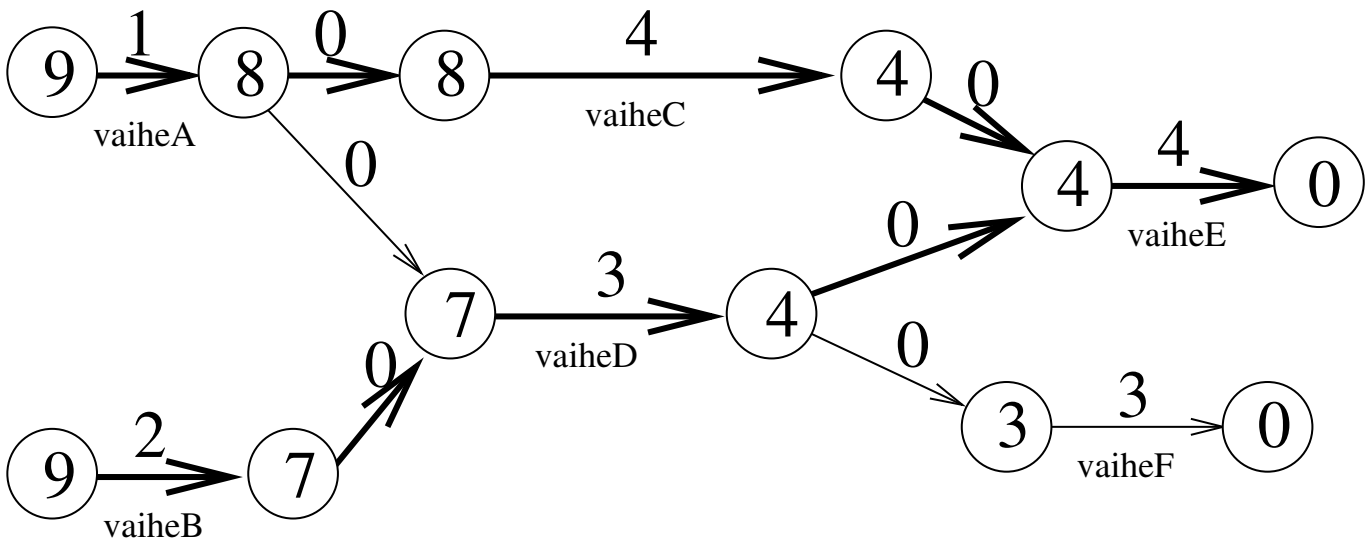
- Projektin kokonaiskesto on

$$t = \max_{u \in V} heaviest[u]$$

- Käytännössä halutaan tietää myös, mitkä projektin poluista ovat kriittisiä: sellaisia, joiden myöhästyminen venyttää koko projektin kestoa
  - Kriittisiä ovat ainakin ne vaiheiden alkusolmut u, joiden $heaviest[u] = t$
  - Jos solmu u on kriittinen, niin sen seuraajasolmuista v ovat kriittisiä ne, joille $heaviest[u] = w(u, v) + heaviest[v]$

- Kriittiset polut voidaan tulostaa syvyyssuuntaisella läpikäynnillä sen jälkeen, kun kunkin solmun $heaviest$-arvo on ensin laskettu
  - lähtösolmuina ovat ne solmut u, joille ehto $heaviest[u] = t$ pätee
  - kaari (u, v) kuuluu kriittiselle polulle, jos $heaviest[u] = w(u, v) + heaviest[v]$

- Esimerkki kriittisten polkujen löytämisestä
  - jokaiseen solmuun on merkitty sen *heaviest*-arvo
  - kriittiset polut on vahvennettu



Pisin yksinkertainen polku painotetussa verkossa

- Pisin tarkoittaa tässä siis painavin
- Ongelma ei ole mielekäs ilman polun yksinkertaisuusoletusta, jos verkossa on positiivinen sykli
- Yleisessä tapauksessa ongelma on NP-täydellinen, mistä seuraa, että ongelmalle ei tunneta polynomista ratkaisualgoritmia
- Edellä kuitenkin näimme, että kun kyseessä on DAG, tämä vastaa kriittisten polkujen etsintää
- Kirjoitamme tässä vielä algoritmin auki DAGille

```
max-path(G)
    for jokaiselle solmulle u ∈ V
        color[u] = white
    for jokaiselle solmulle u ∈ V
        if color[u] == white
            DFS-visit3(u)
    return max{ h[u] | u ∈ V }
```

```
DFS-visit3(u)
    color[u] = gray
    h[u] = 0
    for jokaiselle solmulle v ∈ vierus[u]
        if color[v] == white
            DFS-visit3(v)
        if h[u] < w(u,v) + h[v]
            h[u] = w(u,v) + h[v]
```

Verkon vahvasti yhtenäiset komponentit

- Suunnattua verkkoa $G = (V, E)$ sanotaan vahvasti yhtenäiseksi (engl. strongly connected), jos kaikilla solmuilla $u, v \in V$ on olemassa polut $u \rightsquigarrow v$ ja $v \rightsquigarrow u$

- Eli vahvasti yhtenäisen verkon kaikki solmut ovat saavutettavia toisistaan

- Jos verkko ei ole vahvasti yhtenäinen, se voidaan jakaa erillisiin vahvasti yhtenäisiin komponentteihin (engl. strongly connected components) $V_1, V_n, \ldots, V_n$ seuraavasti:

  1. jokaisella solmulla u pätee $u \in V_i$ tasan yhdellä i

  eli kukin solmu kuuluu tasan yhteen vahvasti yhtenäiseen komponenttiin

  2. jos $u \in V_i$ ja $v \in V_i$, niin $u \rightsquigarrow v$ ja $v \rightsquigarrow u$

  eli vahvasti yhtenäisen komponentin kaikki solmut ovat toisistaan saavutettavissa

  3. jos $u \in V_i$ ja $v \in V_j$, missä $i \neq j$, niin ei ole olemassa molempia poluista $u \rightsquigarrow v$ ja $v \rightsquigarrow u$

  eli kahden eri vahvasti yhtenäisen komponentin solmut eivät ole molemmat toisistaan saavutettavia

- Matemaattisemmin ilmaistuna voidaan sama sanoa käyttäen käsitteitä ekvivalenssirelaatio ja ekvivalenssiluokka

- Voidaan merkitä $u \sim v$, jos verkossa G on sekä polku $u \rightsquigarrow v$ että polku $v \rightsquigarrow u$; $\sim$ on nyt ekvivalenssirelaatio eli toteuttaa ehdot

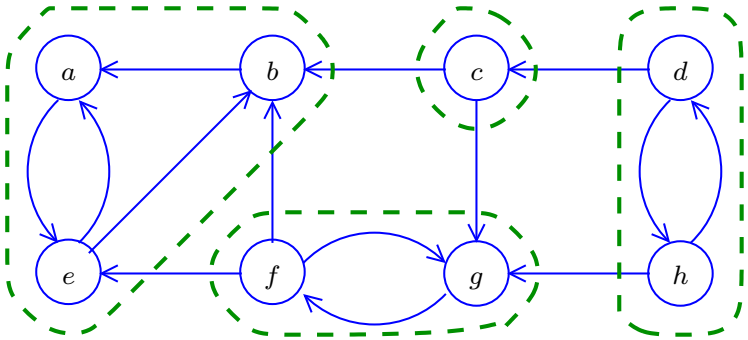
  1. $u \sim u$ kaikilla u (refleksiivisyys)
  2. jos $u \sim v$ niin $v \sim u$ (symmetrisyys)
  3. jos $u \sim v$ ja $v \sim w$ niin $u \sim w$ (transitiivisuus)

- Tästä seuraa, että solmut voidaan jakaa sen suhteen ekvivalenssiluokkiin V_i, jolloin
  - jokaisella u pätee $u \in V_i$ tasan yhdellä i
  - jos $u \in V_i$ ja $v \in V_i$, niin $u \sim v$
  - jos $u \in V_i$ ja $v \in V_j$, missä $i \neq j$, niin $u \not\sim v$

- Näitä ekvivalenssiluokkia V_i sanotaan siis verkon vahvasti yhtenäisiksi komponenteiksi

- Verkko on siis vahvasti yhtenäinen, jos sillä on vain yksi vahvasti yhtenäinen komponentti

- Tarkastellaan seuraavaa suunnattua verkkoa:



- Verkon vahvasti yhtenäiset komponentit ovat

$$\begin{aligned} V_1 &= \{a, b, e\} \\ V_2 &= \{c\} \\ V_3 &= \{d, h\} \\ V_4 &= \{f, g\} \end{aligned}$$

- Verkkoa, jonka solmut ovat vahvasti yhtenäiset komponentit ja kaaret yhteydet komponettien välillä, kutsutaan komponenttiverkoksi

- Vaikka verkon vahvasti yhtenäisten komponenttien tuntemisen hyödyllisyys ei ole päällepäin kovin ilmeinen seikka, on vahvasti yhtenäisten komponenttien selvittämiselle paljon käyttöä erilaisissa sovellustilanteissa

- Vahvasti yhtenäiset komponentit voidaan muodostaa seuraavalla algoritmilla:

  Strongly-Connected-Components(G)

  1. Suorita verkon $G = (V, E)$ syvyyssuuntainen läpikäynti.

  Kun solmun v käsittely on ohi, eli asetetaan $color[v] =$ black, lisätään se pinoon P (eli pinon päällimmäisenä on solmu, jonka f-arvo on suurin)

  2. Muodosta verkon G transpoosi $G^{\mathsf{T}} = (V^{\mathsf{T}}, E^{\mathsf{T}})$, missä $V^{\mathsf{T}} = V$ ja $E^{\mathsf{T}} = \{\, (v, u) \mid (u, v) \in E \,\}$.

  Verkon transpoosi on siis muuten sama verkko, mutta kaaret on käännetty päinvastaiseen suuntaan

  3. Suorita verkon G^{T} syvyyssuuntainen läpikäynti alkaen pinon P huipulla olevasta alkiosta.

  Jokainen verkon G^{T} syvyyssuuntaisen läpikäynnin aikana muodostunut syvyyssuuntaispuu on verkon G vahvasti yhtenäinen komponentti.

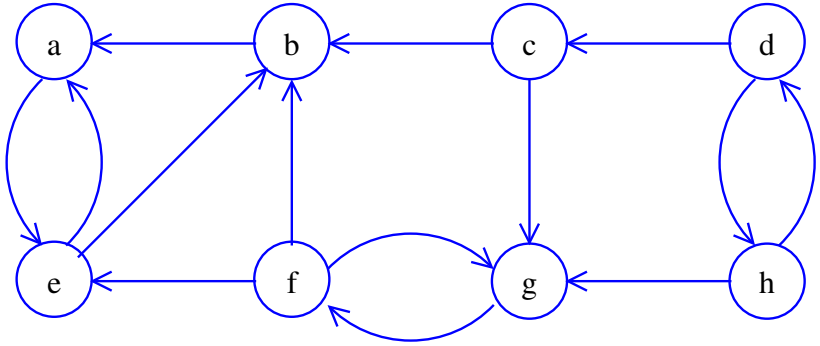
  Aina kun yksi puu on tullut läpikäydyksi, seuraava **DFS-visit**(G^{T}, u) alkaa pinosta P ensimmäisenä löytyvästä vielä läpikäymättömästä solmusta u (eli f-arvoltaan suurimmasta vielä läpikäymättömästä solmusta)

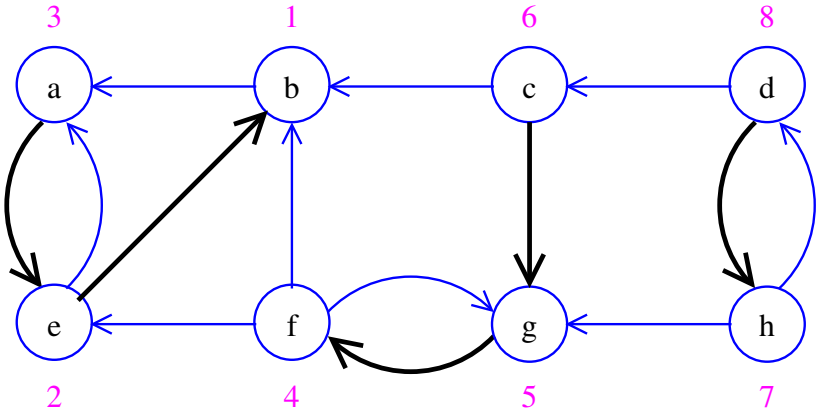
- Koska verkon transpoosi voidaan muodostaa ajassa $\mathcal{O}(|\,V\,|+|\,E\,|)$, vahvasti yhtenäiset komponentit selvittävän algoritmin pahimman tapauksen aikavaativuus on $\mathcal{O}(|\,V\,|+|\,E\,|)$ ja tilavaativuus on $\mathcal{O}(|\,V\,|)$ kuten syvyyssuuntaisella läpikäynnillä

- Tarkastellaan esimerkkiä:

- Vasemmalla verkko G jonka vahvasti yhtenäiset komponentit halutaan selvittää

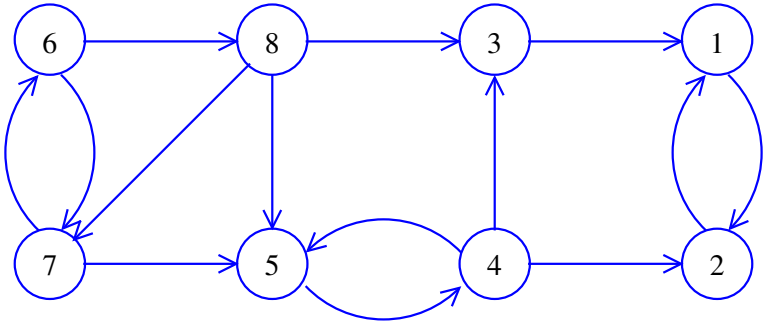
- Oikealla verkon G syvyyssuuntaisen läpikäynnin tulos, solmut on numeroitu siinä järjestyksessä missä niiden käsittely valmistui, eli missä järjestyksessä ne laitettiin pinoon P

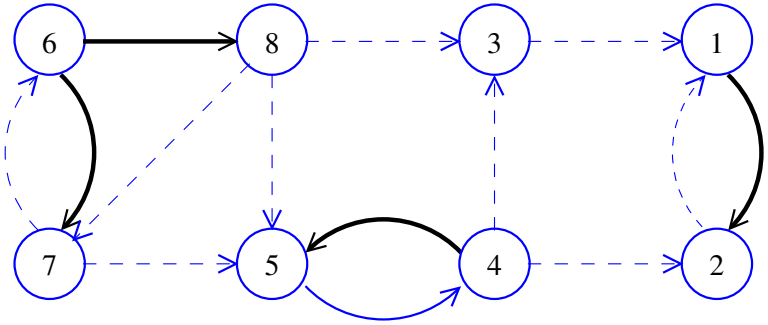




- Vasemmalla transpoosiverkko G^{T}, solmut on numeroitu siinä järjestyksessä missä ne otetaan pinosta, joka siis on järjestys, missä transpoosin syvyyssuuntainen läpikäynti etenee

- Oikealla läpikäynnin tulos, verkon G vahvasti yhtenäiset komponentit muodostuvat läpikäynnin aikana muodostuneista syvyyssuuntaispuiden solmuista





- Algoritmi on suhteellisen yksinkertainen, mutta päältäpäin on vaikea nähdä, että algoritmi todella toimii

- Todistetaan seuraavaksi, että algoritmi toimii oikein

- Algoritmin toiminta perustuu seuraavaan havaintoon:

- **Lause 8.3**: Jos $A \neq B$ ovat kaksi vahvasti yhtenäistä komponenttia ja $(u, v) \in E$ joillain $u \in A$ ja $v \in B$, niin

$$\max_{x \in A} f[x] > \max_{y \in B} f[y].$$

  Siis jos komponentista A on kaari komponenttiin B, niin vaiheessa 1 komponentin B läpikäynti loppuu ennen kuin komponentin A läpikäynti.

- **Todistus**: Kaksi tapausta sen mukaan, kumman komponentin läpikäynti alkaa ensin

  1. Oletetaan ensin, että komponentin A läpikäynti alkaa solmusta $x \in A$, kun kaikki komponentin B solmut ovat vielä valkoisia. Koska $x \rightsquigarrow u \to v \rightsquigarrow y$ millä tahansa $y \in B$ ja kutsun **DFS-visit**(G, x) alkaessa kaikki nämä solmut ovat valkoisia, valkopolkulauseen mukaan jokainen $y \in B$ tulee solmun x jälkeläiseksi. Siis $f[y] < f[x]$ kaikilla $y \in B$

  2. Oletetaan nyt, että komponentin B läpikäynti alkaa solmusta $y \in B$, kun kaikki komponentin A solmut ovat vielä valkoisia. Koska komponenttiverkko on syklitön, solmusta y ei ole polkua mihinkään komponentin A solmuun. Siis kaikista komponentin B solmuista tulee mustia, ennen kuin mikään komponentin A solmu on edes harmaa. □

- **Lause 8.4**: Algoritmi **Strongly-Connected-Components** tuottaa oikein verkon vahvasti yhtenäiset komponentit.

- **Todistus**: Todistamme induktiolla k:n suhteen, että k ensimmäistä vaiheessa 3 tuotettua puuta ovat verkon G vahvasti yhtenäisiä komponentteja.

  Tapaus $k = 0$ on triviaali.

  Oletetaan, että ensimmäiset $k - 1$ puuta ovat oikein ja tarkastellaan puuta numero k.

  Olkoon y puun numero k juuri, ja olkoon B se vahvasti yhtenäinen komponentti, johon y kuuluu.

  Induktio-oletuksen mukaan aiemmat puut eivät sisältäneet komponentin B solmuja, joten kaikki komponentin B solmut ovat kutsun **DFS-visit**(G^{T}, y) alkaessa valkoisia. Valkopolkulauseen mukaan ne tulevat solmun y jälkeläisiksi transpoosin syvyyssuuntaisessa puussa.

Pitää vielä osoittaa, että solmun y jälkeläisiksi ei tule yhtään solmua x, missä x kuuluu johonkin toiseen komponenttiin $A \neq B$.

Ei-valkoisista solmuista ei tietenkään enää tehdä kenenkään jälkeläisiä. Olkoon $A \neq B$ jokin vielä valkoisista solmuista koostuva komponentti.

Koska valitaan **DFS-visit** alkamaan f-arvoltaan suurimmasta vielä läpikäymättömästä solmusta, pätee, että $f[y] > f[x]$ kaikilla $x \in A$. Edellisen lauseen mukaan verkossa G ei voi olla kaarta komponentista A komponenttiin B. Siis transpoosissa G^{T} ei ole kaaria komponentista B komponenttiin A.

Täten solmun y jälkeläisiksi tulee tasan kaikki komponentin B solmut. □

- Todistus siis osoittaa, että algoritmi toimii oikein vaikka algoritmin toiminnan perusteita onkin hiukan vaikea nähdä päältäpäin

- Vahvasti yhtenäiset komponentit etsivä algoritmi onkin hyvä esimerkki tilanteessa, jossa matematiikkaa tarvitaan pelkän intuition lisäksi vakuuttamaan algoritmin oikeellisuudesta

Lyhimmät polut painotetussa verkossa

- Tarkastellaan painotettuja verkkoja ja tulkitaan kaaren paino sen yhdistämien solmujen etäisyydeksi
- Kaaripainon voi tulkita myös esim. ajaksi, joka kuluu matkustettaessa kaaren yhdistävien solmujen välinen matka
- Tehtävänä on löytää annetusta solmusta s lyhin etäisyys, eli vähiten painava polku kaikkiin muihin solmuihin
- Ongelma on keskeinen esim. tietoliikenneverkkojen reitityksessä

- Tarkastelemme kahta eri versiota ongelmasta löytää lyhimmät polut annetusta lähtösolmusta kaikkiin muihin verkon solmuihin:
  - jos kaarten painot voivat olla mielivaltaisia, ongelma ratkeaa ajassa $\mathcal{O}(|V||E|)$ soveltamalla **Bellmanin-Fordin algoritmia**
  - jos kaarten painot oletetaan ei-negatiivisiksi, ongelma ratkeaa ajassa $\mathcal{O}((|V|+|E|)\log|V|)$ soveltamalla **Dijkstran algoritmia**
- Tapaukseen, jossa halutaan lyhin polku lähtösolmusta u vain yhteen annettuun kohdesolmuun v, ei tunneta yleisessä tapauksessa tehokkaampaa ratkaisua kuin laskea samalla kaikki lyhimmät polut lähtösolmusta
- Lisäksi
  - lyhimmät polut kaikkien $|V|^2$ solmuparin välille voidaan kerralla laskea ajassa $\mathcal{O}(|V|^3)$ soveltamalla **Floydin-Warshallin algoritmia**, joka sallii myös negatiiviset kaaripainot

- Olkoon $G = (V, E)$ suunnattu verkko ja s eräs verkon solmu
- Oletetaan että jokaiseen kaareen $(u, v) \in E$ on liitetty pituus $w(u, v)$
- Oletetaan lisäksi
  - jos $(u, v) \notin E$ niin $w(u, v) = \infty$ (jos solmuja ei yhdistä kaari, niiden välisen yhteyden pituus on ääretön)
  - kaikilla v pätee $w(v, v) = 0$ (jokaisesta solmusta matka itseensä on nolla)
- Huomautus: Jos negatiiviset painot ovat sallittuja, voi esiintyä negatiivisen painoinen sykli: kiertämällä sykliä mielivaltaisen monta kertaa saataisiin mielivaltaisen pieni ("$-\infty$") polun pituus
- **Bellman-Fordin** ja **Dijkstran** algoritmit pitävät yllä aputaulukkoja $distance$ ja $path$, joihin talletetaan tietoa verkon solmuista
  - $distance[v]$ kertoo mikä on solmun v etäisyysarvio solmusta s
  - $path[v]$ on solmua v edeltävä solmu lyhimmällä toistaiseksi tunnetulla polulla $s \rightsquigarrow v$

- Tarvitsemme molemmissa algoritmeissa samat aliohjelmat: alustus ja päivitysoperaatio, jota kutsutaan kaaren löysäämiseksi (relaxation)

Initialise-Single-Source(G,s)
1 for kaikille solmuille v $\in$ V
2 distance[v] = ∞
3 path[v] = NIL
4 distance[s] = 0

Relax(u,v,w)
1 if distance[v] > distance[u] + w(u,v)
2 distance[v] = distance[u]+w(u,v)
3 path[v] = u

- Löysääminen voi tuntua omituiselta termiltä, mutta se viittaa siihen, että operaation lopputuloksena oleva ehto $distance[v] \leq distance[u] + w(u, v)$ on "relaksoitunut"

- Ongelmana on, että kun relaksaatio-operaatiossa muutetaan $distance[v]$, niin solmusta v alkaviin kaariin (v, r) voi syntyä jännitteitä, eli tilanteita

  $distance[r] > distance[v] + w(v, r)$, joita pitää taas löysätä

- Algoritmit eroavat siinä, missä järjestyksessä suoritetaan löysäämisoperaatiota

- **Dijkstran algoritmi** järjestää operaatiot niin tehokkaasti, että kutakin kaarta tarvitsee löysätä vain kerran. Tämän onnistuminen voidaan kuitenkin taata vain, kun verkossa ei ole negatiivisia painoja

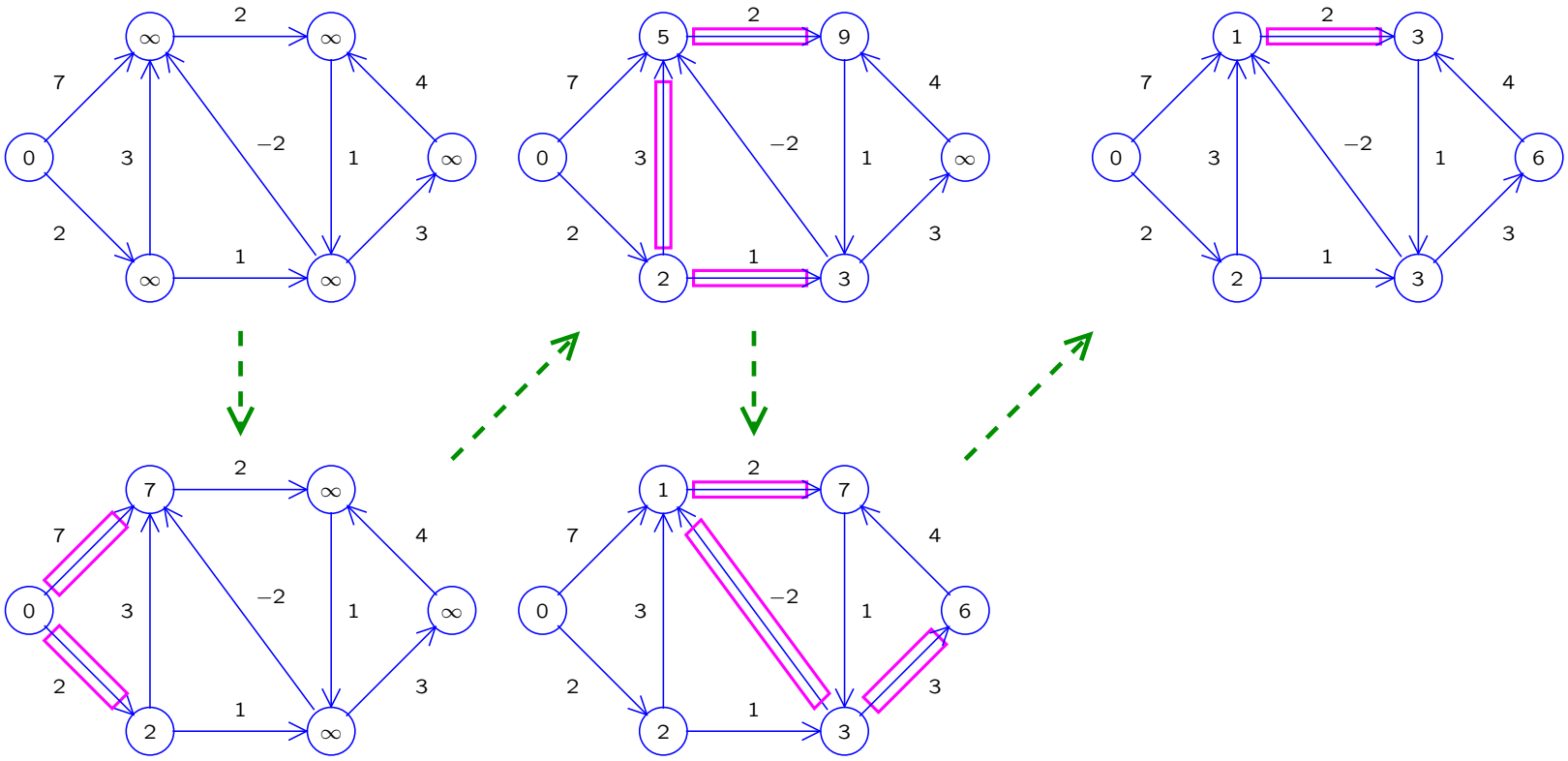
Bellmanin-Fordin algoritmi [Richard Bellman ja Lester Ford Jr., 1958]

- Tässä algoritmissa löysäämisjärjestystä ei yritetäkään optimoida: kaikki kaaret käydään järjestyksessä läpi useita kertoja, kunnes tulos on valmis

- Kuten pian näemme, $|V| - 1$ läpikäyntiä riittää, ellei verkossa ole negatiivisen painoisia syklejä

- Jos tämän ajan jälkeen on jännitteitä jäljellä, algoritmi palauttaa false merkiksi negatiivisen painoisesta syklistä

```
Bellman-Ford(G,w,s)
1   Initialise-Single-Source(G,s)
2   for i = 1 to |V| - 1
3       for jokaiselle kaarelle (u, v) ∈ E
4           Relax(u,v,w)
5   for jokaiselle kaarelle (u, v) ∈ E
6       if distance[v] > distance[u] + w(u,v)
7           return false
8   return true
```


0
∞
∞
∞
∞
2
7
4
3
−2
1
2
1
3
0
5
9
2
3
2
7
4
3
−2
1
2
1
3
0
1
3
2
3
2
7
4
3
−2
1
2
1
3
0
7
∞
∞
2
7
4
3
−2
1
2
1
3
0
1
7
2
7
4
3
−2
1
2
1
3
0
6
3
2
3
2
7
4
3
−2
1
2
1
3

- Seuraava lemma sanoo oleellisesti, että $|V|-1$ iteraatiota riittää kaikkien jännitteiden löysäämiseen, ellei verkossa ole negatiivisen painoisia kehiä:

- **Lemma 8.5**: Jos
  - solmusta s on polku solmuun v ja
  - solmusta s ei voi saavuttaa mitään negatiivisen painoista sykliä,

  niin algoritmin **Bellman-Ford** suorituksen päättyessä $distance[v] = \delta(s, v)$, missä $\delta(s, v)$ on lyhimmän polun $s \rightsquigarrow v$ paino.

- **Todistus:** Oletusten mukaan jokin yksinkertainen polku $\pi = (s, v_1, \ldots, v_{k-1}, v)$ on lyhin polku $s \rightsquigarrow v$. Merkitään $s = v_0$ ja $v = v_k$. Nyt $\pi_i = (v_0, \ldots, v_i)$ on lyhin polku $s \rightsquigarrow v_i$ kaikilla $1 \leq i \leq k$.

  Selvästi aina pätee, että $distance[u] \geq \delta(s, u)$. Huomaa, että $distance[u]$ ei koskaan kasva millään u, eli jos ehto $distance[u] = \delta(s, u)$ kerran tulee voimaan, se pysyy voimassa.

Osoitetaan induktiolla indeksin i suhteen, että kun for-silmukkaa rivillä 2 on iteroitu i kertaa, pätee $distance[v_i] = \delta(s, v_i)$. Koska $k \leq |V| - 1$, tästä seuraa väite.

Tapaus $i = 0$ pätee selvästi.

Oletetaan, että $distance[v_i] = \delta(s, v_i)$ pätee iteraation i jälkeen. Kun on seuraavan kerran suoritettu **Relax**(v_i, v_{i+1}, w), pätee

$$\begin{aligned} \delta(s, v_{i+1}) &\leq distance[v_{i+1}] \\ &\leq distance[v_i] + w(v_i, v_{i+1}) \\ &= \delta(s, v_i) + w(v_i, v_{i+1}) \\ &= \delta(s, v_{i+1}), \end{aligned}$$

missä viimeinen askel perustuu oletukseen, että $s \rightsquigarrow v_i \to v_{i+1}$ on lyhin polku $s \rightsquigarrow v_{i+1}$. □

- Seuraavat kaksi lemmaa perustelevat, että algoritmin lopussa tehtävä testi menee oikein

- **Lemma 8.6**: Jos solmusta s ei voi saavuttaa negatiivisen painoisia kehiä, **Bellman-Ford** palauttaa true.

- **Todistus:** Edellisen lemman mukaan lopuksi pätee $distance[v] = \delta(s, v)$ kaikilla $v \in V$. Siis kaikilla $u \in V$ saadaan

$$\begin{aligned} distance[v] &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \\ &= distance[u] + w(u, v). \end{aligned}$$

□

- **Lemma 8.7**: Jos verkossa on sykli $c = (v_0, v_1, \ldots, v_k)$, missä $v_k = v_0$ ja
  - v_0 on saavutettavissa solmusta s ja
  - syklin paino $w(c) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$ on negatiivinen,

  niin **Bellman-Ford** palauttaa false.

- **Todistus:** Tehdään vastaoletus, että **Bellman-Ford** palauttaa true. Tällöin erityisesti $distance[v_{i+1}] \leq distance[v_i] + w(v_i, v_{i+1})$ kaikilla i. Saadaan

$$\begin{aligned} distance[v_k] &\leq distance[v_{k-1}] + w(v_{k-1}, v_k) \\ &\leq distance[v_{k-2}] + w(v_{k-2}, v_{k-1}) + w(v_{k-1}, v_k) \\ &\ldots \\ &\leq distance[v_0] + \sum_{i=1}^{k} w(v_{i-1}, v_i). \end{aligned}$$

  Koska $v_0 = v_k$ ja $distance[v_0] < \infty$, pätee $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$, mikä on ristiriita oletuksen $w(c) < 0$ kanssa. □

- Edellisten kolmen lemman perusteella
  - **Bellman-Ford** palauttaa true, jos ja vain jos solmusta s ei voi saavuttaa negatiivisen painoista sykliä ja
  - tällöin lopuksi pätee $distance[v] = \delta(s, v)$ kaikilla $v \in V$

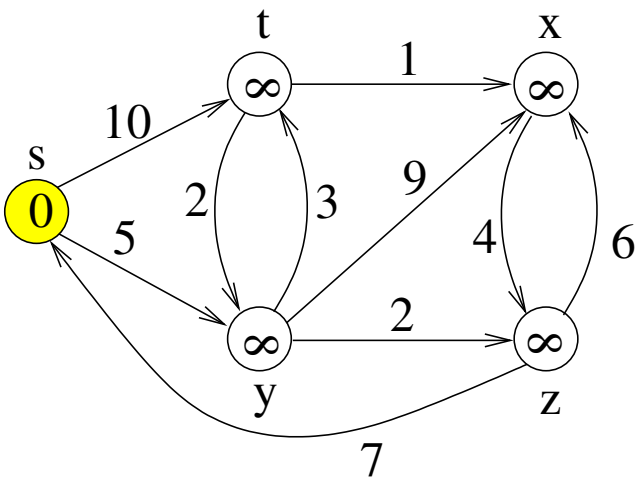
Dijkstran algoritmi [Edsger Dijkstra, 1956]

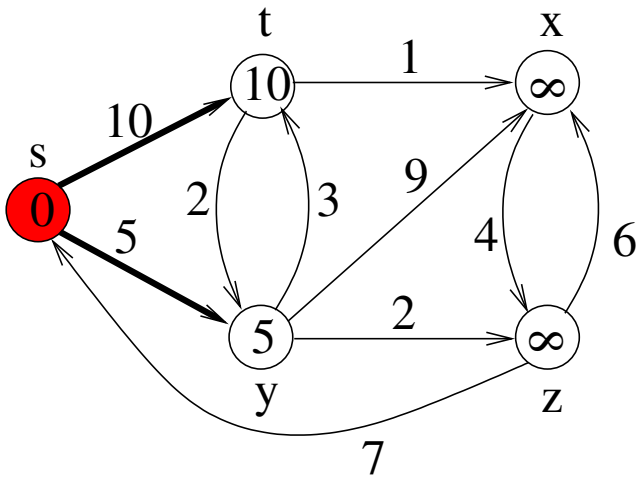
- **Dijkstran algoritmi** pitää yllä joukkoa S, joka muodostuu solmuista joiden lyhin etäisyys solmuun s on jo selvitetty
- Algoritmi valitsee toistuvasti solmun $u \in V \setminus S$, jonka etäisyysarvio solmuun s on pienin
  - valittu solmu u lisätään joukkoon S, ja
  - kaikkien solmun u vierussolmujen v etäisyysarvio solmuun s sekä polkutieto $path[v]$ päivitetään

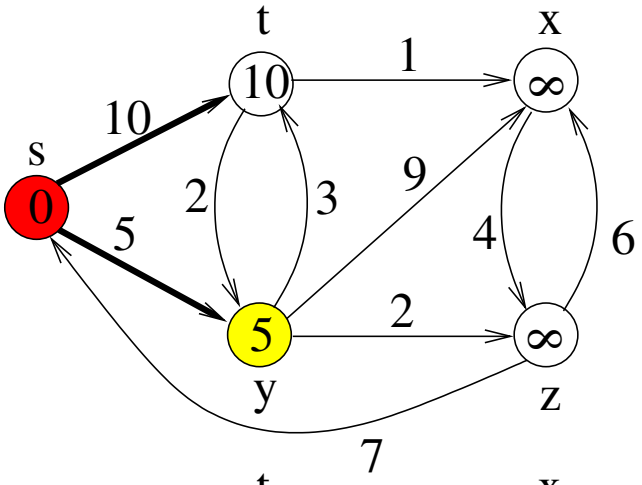
- Algoritmi osittain abstraktissa muodossa:

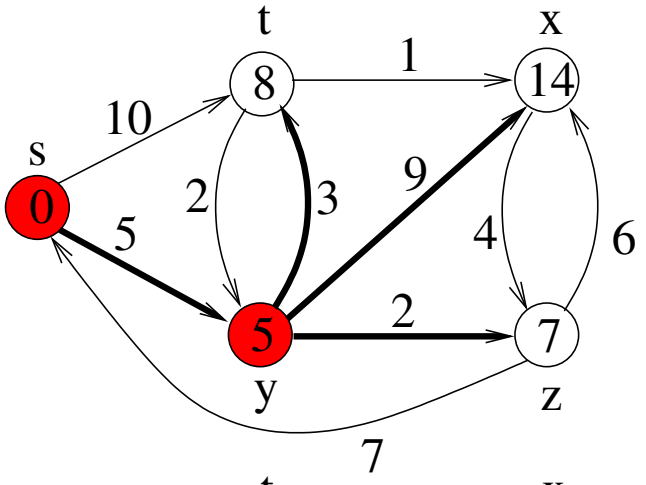
```
Dijkstra(G,w,s)
1   Initialise-Single-Source(G,s)
2   S = ∅
3   while ( kaikki solmut eivät vielä ole joukossa S )
4       valitse solmu u ∈ V \ S, jonka etäisyysarvio lähtösolmuun s on lyhin
5       S = S ∪ {u}
6       for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
7           Relax(u,v,w)
```

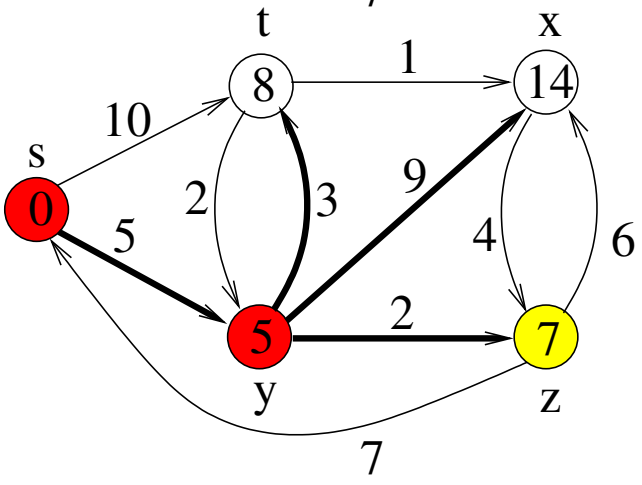
- Seuraavilla sivuilla on esimerkki algoritmin toiminnasta
  - joukkoon S lisätyt alkiot tummanharmaita (värikuvassa punaisia)
  - käsittelyvuorossa oleva alkio vaaleanharmaa (värikuvassa keltainen)

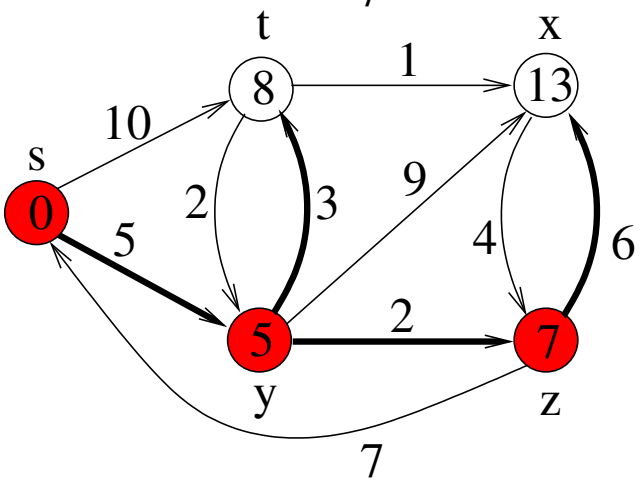

t
x
s
y
z
∞
∞
0
∞
∞
10
1
2
3
9
4
6
5
2
7

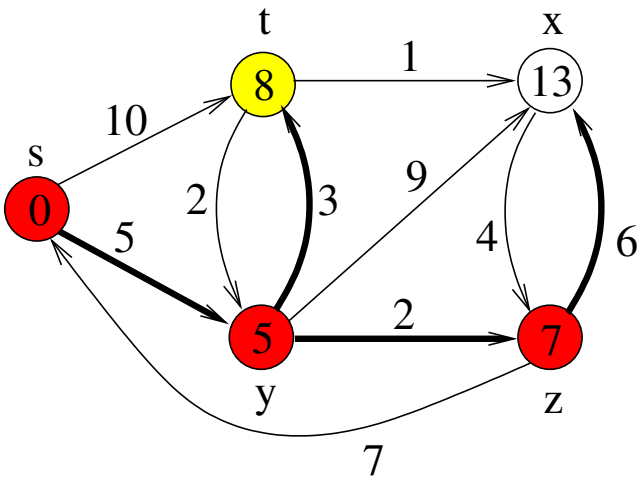

t
x
s
y
z
10
∞
0
5
∞
10
1
2
3
9
4
6
2
7

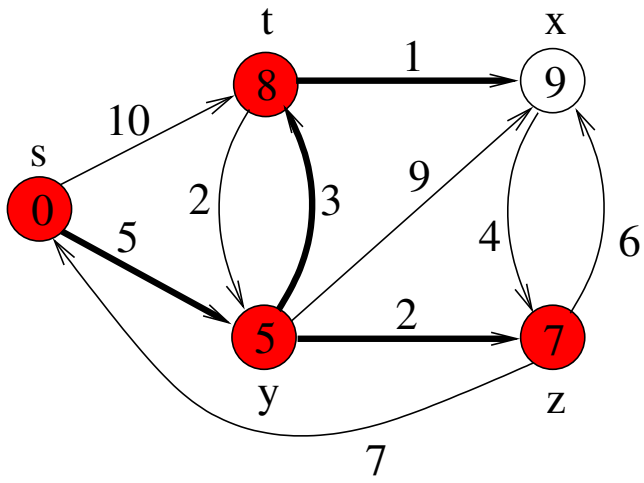

t
x
s
y
z
10
∞
0
5
∞
10
1
2
3
9
4
6
2
7

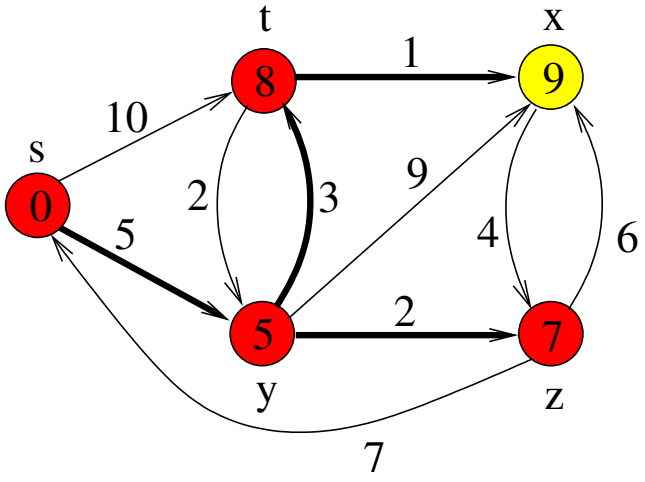

t
x
s
y
z
8
14
0
5
7
10
1
2
3
9
4
6
2
7

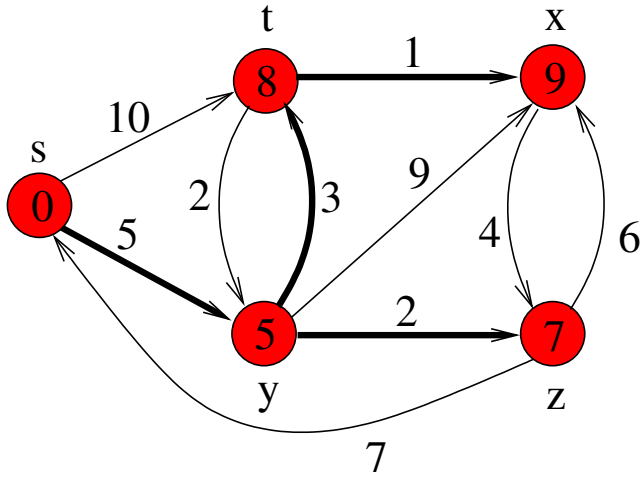

t
x
s
y
z
8
14
0
5
7
10
1
2
3
9
4
6
2
7


t
x
s
y
z
8
13
0
5
7
10
1
2
3
9
4
6
2
7









- Tieto lyhimmistä poluista on kuvattu paksunnettuina kaarina

- Miten algoritmin rivi 4, jossa on valittava solmu $u \in V \setminus S$, jonka etäisyys lähtösolmuun s on lyhin, voidaan toteuttaa tehokkaasti?
  - yksi mahdollisuus olisi tietysti käydä läpi kaikki solmut joukosta $V \setminus S$
- Tehokkaampaan ratkaisuun päästään käyttämällä aputietorakenteena minimikekoa H
  - solmut $v \in V \setminus S$ pidetään keossa
  - solmun v avaimena sen etäisyysarvion $distance[v]$ arvo
  - näin seuraavaksi käsiteltävä solmu saadaan nopeasti operaatiolla **heap-delete-min**
  - jos valitun solmun u jonkin vierussolmun v etäisyysarviota pienennetään, kutsutaan sille **heap-decrease-key**-operaatiota, joka asettaa solmun v taas oikealle paikalle keossa

- Algoritmi joka hyödyntää minimikekoa

```
Dijkstra-with-heap(G,w,s)
1   Intialise-Single-Source(G,s)
2   S = ∅
3   for kaikille solmuille v ∈ V
4       heap-insert(H,v,distance[v])
5   while not empty(H)
6       u = heap-del-min(H)
7       S = S ∪ {u}
8       for jokaiselle solmulle v ∈ vierus[u]    // kaikille u:n vierussolmuille v
9           Relax(u,v,w)
10          heap-decrease-key(H,v,distance[v])
                    // ei tee mitään, jos distance[v] ei ole muuttunut
```

- Huom: Joukkoa S ei oikeastaan enää tarvita, sillä joukossa S ovat täsmälleen ne alkiot jotka eivät ole keossa H

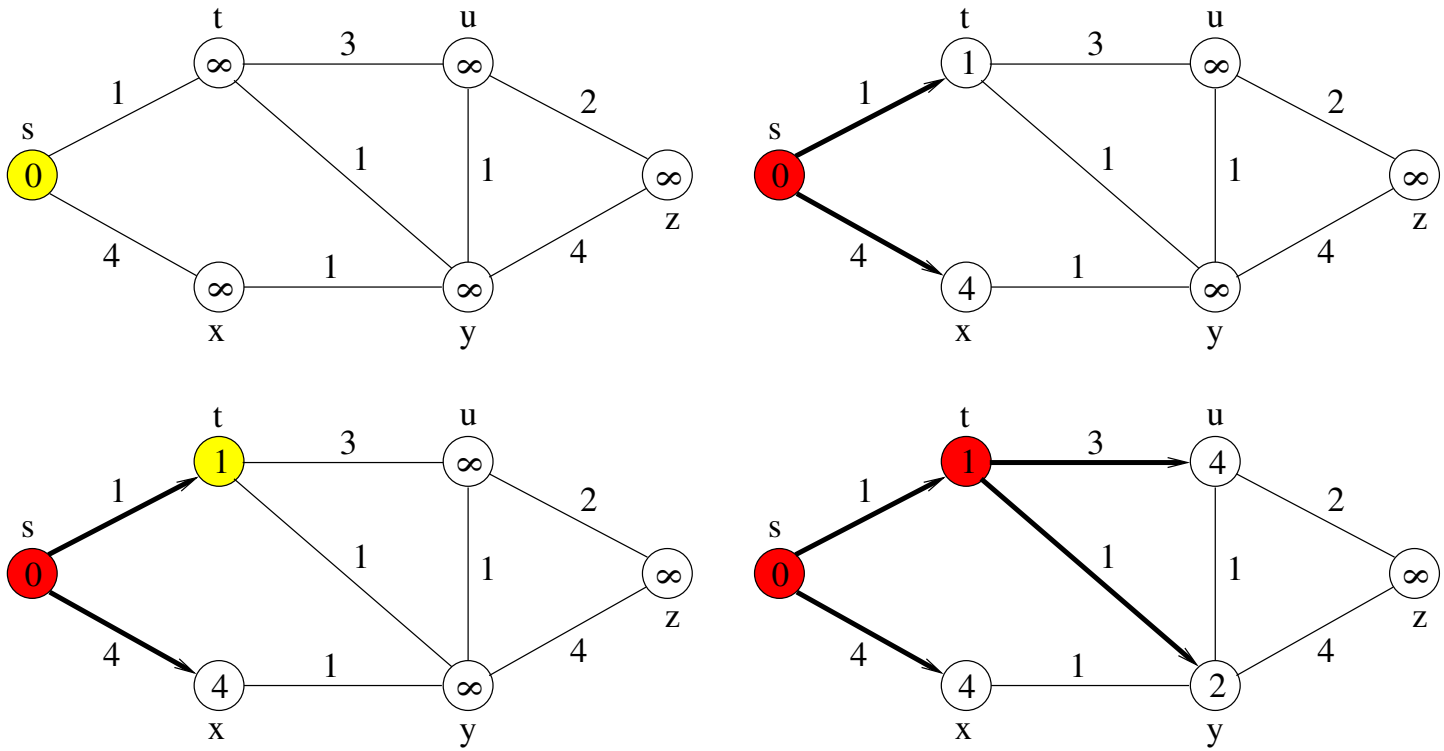
- Pidetään kuitenkin S mukana sillä se selkeyttää pian esitettävää oikeellisuustodistusta

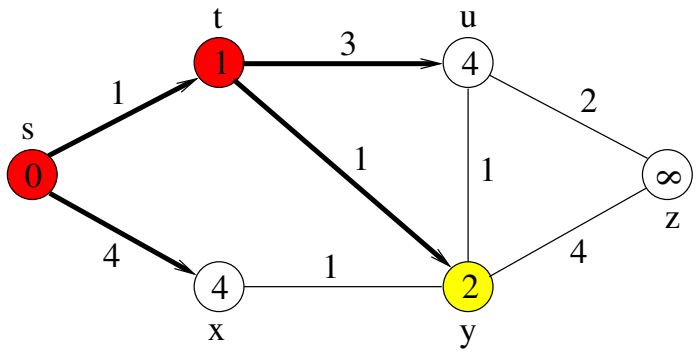
- Algoritmin suorituksen jälkeen lyhin polku $s \rightsquigarrow v$ saadaan selville seuraavasti:
  - $path[v]$ kertoo minkä solmun kautta lyhin polku $s \rightsquigarrow v$ saapuu solmuun v
  - solmuun $path[v]$ lyhin polku saapuu solmun $path[path[v]]$ kautta, jne
  - laitetaan pinoon $path[v]$, $path[path[v]]$, $path[path[path[v]]]$ ja tulostetaan pinon sisältö
  - näin saadaan tulostettua polulla koko polku $s \rightsquigarrow v$ alusta loppuun

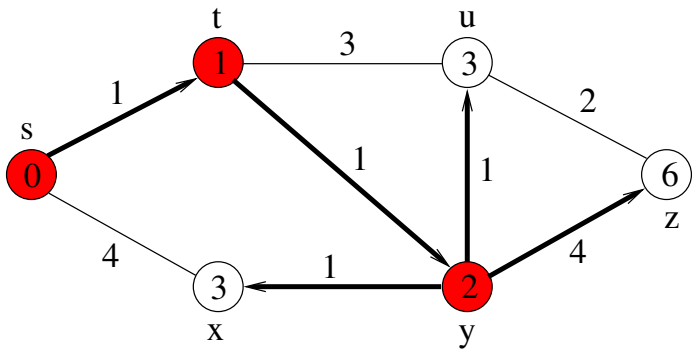
- Algoritmina:

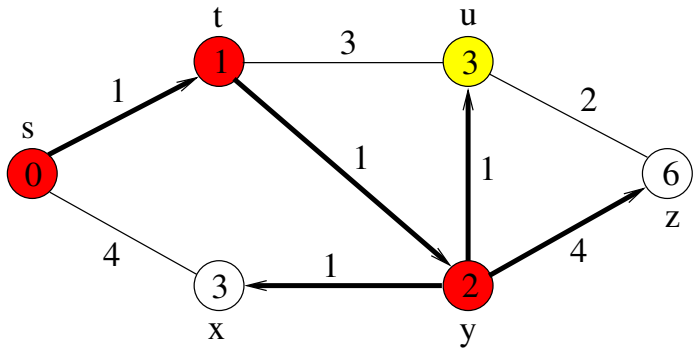
```
shortest-path(G,v)
1   u = path[v]
2   while u ≠ s
3       push(pino,u)
4       u = path[u]
5   print("lyhin polku solmusta s solmuun v kulkee seuraavien solmujen kautta:")
6   while not empty(pino)
7       u = pop(P)
8       print(u)
```

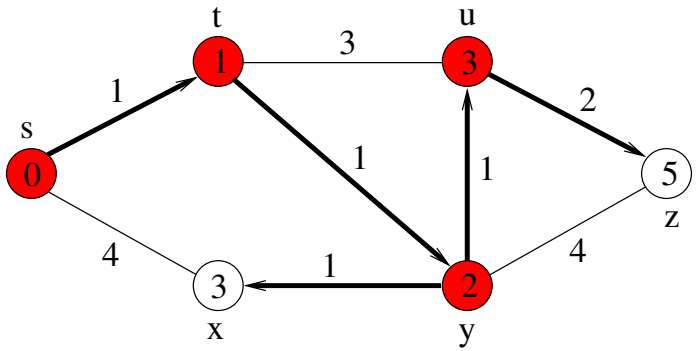
- Toinen esimerkki **Dijkstran algoritmin** toiminnasta, tällä kertaa kyseessä suuntaamaton verkko

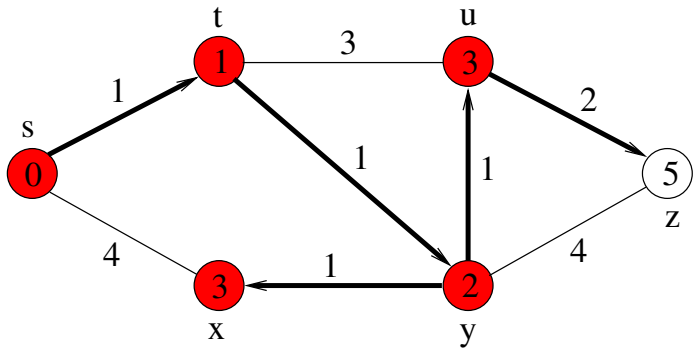


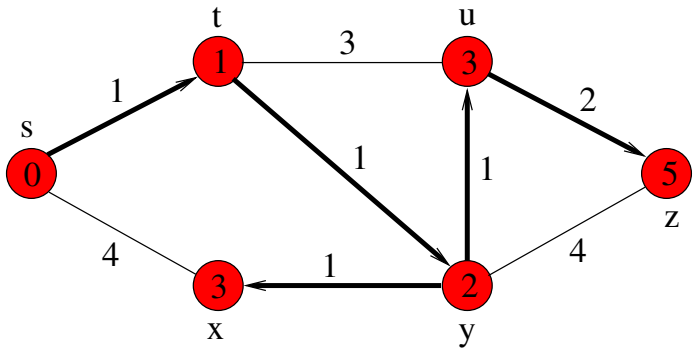

t
u
s
x
y
z
0
1
4
2
∞
3
1
1
2
4
1
4


t
u
s
x
y
z
0
1
3
2
6
3
1
1
2
4
1
4


t
u
s
x
y
z
0
1
3
2
6
3
1
1
2
4
1
4


t
u
s
x
y
z
0
1
3
2
5
3
1
1
2
4
1
4


t
u
s
x
y
z
0
1
3
2
5
3
1
1
2
4
1
4


t
u
s
x
y
z
0
1
3
2
5
3
1
1
2
4
1
4

- **Dijkstran algoritmin** vaativuus
  - rivien 1-2 alustustoimet vievät aikaa selvästi $\mathcal{O}(|V|)$
  - käytössä siis minimikeko, ja kutsutaan keko-operaatioita **heap-insert**, **heap-del-min** ja **heap-decrease-key**
  - keko-operaatioiden vaativuus on $\mathcal{O}(\log n)$ jos keossa n alkiota
  - riveillä 3-4 tehdään $|V|$ kappaletta **heap-insert**-operaatioita, aikaa siis kuluu $\mathcal{O}(|V| \log |V|)$; arvot ovat 0 ja ∞, joten keon voi myös luoda ajassa $\mathcal{O}(|V|)$
  - rivien 5-10 toistolauseessa $\mathcal{O}(\log |V|)$ aikaa vievää operaatiota **heap-del-min** kutsutaan kullekin solmulle kerran, eli yhteensä $|V|$ kertaa
  - koska jokainen solmu v lisätään joukkoon S vain kertaalleen, käydään kukin vieruslista läpi täsmälleen kerran
  - jokaista kaarta siis tutkitaan **Relax**-aliohjelmassa kerran, eli $\mathcal{O}(\log |V|)$ aikaa vievää **heap-decrease-key**-operaatiota kutsutaan maksimissaan $|E|$ kertaa
  - yhteensä toistolauseessa kuluu aikaa $\mathcal{O}((|E| + |V|) \log |V|)$, joka on samalla koko algoritmin aikavaativuus
  - algoritmin alussa kaikki solmut ovat keossa ja tämän jälkeen keko alkaa pienentyä solmujen siirtyessä samalla joukkoon S
  - tilavaativuus on siis selvästi $\mathcal{O}(|V|)$

- Joskus riittää tietää pelkästään kahden solmuparin s ja v välinen lyhin polku
- Otetaan lähtösolmuksi s ja suoritetaan algoritmia kunnes solmu v lisätään joukkoon S
- Tämän jälkeen voidaan lopettaa sillä lyhin polku $s \rightsquigarrow v$ on jo selvinnyt
  - itseasiassa ei ole $\mathcal{O}$-analyysin mielessä helpompaa etsiä lyhintä polkua solmusta s yhteen solmuun v kuin solmusta s kaikkiin solmuihin
- **Dijkstran algoritmi** noudattaa ns. ahnetta (engl. greedy) strategiaa:
  - rivillä 6 valitaan aina käsiteltäväksi se solmu mikä on lähimpänä aloitussolmua s
  - ahneudella tarkoitetaan tässä sitä että algoritmi pyrkii joka hetkellä juuri silloin "parhaalta" vaikuttavaan ratkaisuun
  - ei ole itsestäänselvää että ahne strategia tuottaa nimenomaan lyhimmät polut, mutta **Dijkstran algoritmin** tapauksessa näin on
  - todistus perustuu seuraavaan havaintoon: kaikille solmuille $v \in S$ pätee: $distance[v]$ on sama kuin lyhimmän polun $s \rightsquigarrow v$ pituus
  - eli kun solmu on lisätty joukkoon S, sen lyhin etäisyys aloitussolmuun on selvinnyt
- Perustellaan seuraavassa tarkemmin miksi **Dijkstran algoritmi** toimii oikein

- Näytetään, että

  (1) solmuille $v \in S$ pätee: $distance[v]$ on sama kuin lyhimmän polun $s \rightsquigarrow v$ pituus

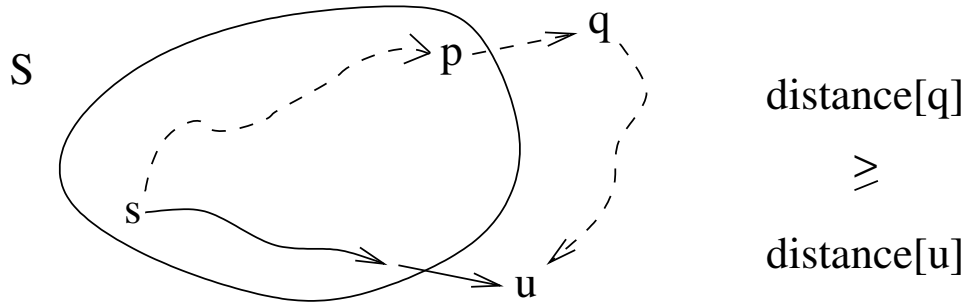
  (2) solmuille $v \in V \setminus S$ pätee: $distance[v]$ on lyhimmän tunnetun, eli käytännössä joukon S solmujen kautta kulkeva polun $s \rightsquigarrow v$ pituus

- Väittämät ovat voimassa alustuksen jälkeen:
  - alussa $S = \{s\}$ ja $distance[s] = 0$, ja koska kaikki kaaripainot positiivisia, ei lyhempää polkua solmuun s ole olemassa
  - algoritmi päivittää s:n vierussolmujen v etäisyysarvioiksi $w(s, v)$, eli selvästi kaikkien solmujen $v \in V \setminus S$ etäisyysarvio on alussa lyhin tunnettu etäisyys

- Oletetaan nyt, että väittämät ovat voimassa tietyllä suoritushetkellä, ja näytetään että ne säilyvät voimassa kun uusia solmuja lisätään joukkoon S

- Olkoon u seuraavaksi käsittelyvuorossa oleva eli joukkoon S lisättävä solmu
  - Käsittelyvuorossa oleva solmu on siis joukon S ulkopuolisista solmuista se, jonka etäisyysarvio on pienin
  - Kun solmu u lisätään joukkoon S ei sen etäisyysarviota enää muuteta
  - eli jotta väittämä (1) pysyisi voimassa, on lisättävän solmun etäisyysarvion jo oltava sama kuin lyhimmän polun $s \rightsquigarrow u$ pituus

- Voiko verkossa olla vielä lyhyempää polkua s:stä u:hun kuin joukon S kautta kulkeva $distance[u]$:n pituinen polku?
  - oletetaan, että verkossa olisi vielä lyhempi polku solmuun u
  - tämän polun täytyisi käydä jossain vaiheessa joukon S ulkopuolella
  - jaetaan tämä polku osiin: $s \rightsquigarrow p \rightarrow q \rightsquigarrow u$ missä q on polun ensimmäinen solmu, joka on joukon S ulkopuolella



  - nyt $distance[q] = distance[p] + w(p, q) \geq distance[u]$ sillä algoritmi valitsi solmun u käsittelyyn ennen q:ta
  - polun loppuosan $q \rightsquigarrow u$ pituuden pitäisi olla negatiivinen, jotta polku olisi todella lyhyempi kuin $distance[u]$. **Dijkstran algoritmi** kuitenkin käsittelee ainoastaan tilanteita, joissa kaarten paino on ei-negatiivinen
  - eli kun solmu u lisätään joukkoon ei koko verkossa voi olla polkua $s \rightsquigarrow u$ jonka pituus olisi pienempi kuin $distance[u]$
  - väittämä (1) siis säilyy voimassa uusia solmuja lisättäessä

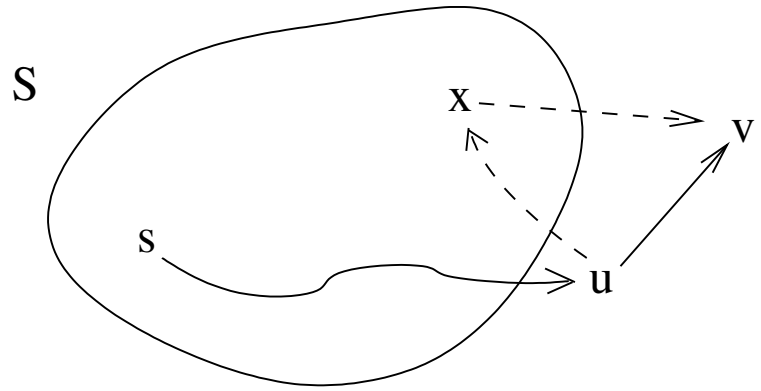
- On vielä osoitettava, että algoritmi päivittää oikein etäisyysarviot taulukkoon $distance$, eli että väittämä (2) säilyy voimassa uusia solmuja joukkoon S lisättäessä

- Olkoon edelleen u seuraavaksi joukkoon S lisättävä solmu, ja tarkastellaan u:n mielivaltaista vierussolmua v joka ei vielä ole joukossa S

- Jos polku $s \rightsquigarrow u \rightarrow v$ on lyhempi kuin aiemmin tunnettu lyhin polku $s \rightsquigarrow v$, asettaa algoritmi **Relax**-aliohjelmassa v:lle uuden etäisyysarvion

- Onko näin asetettu uusi etäisyysarvio pienin etäisyys polulle $s \rightsquigarrow v$, joka voidaan tässä suoritusvaiheessa tietää, eli jossa kaikki solmut polun varrella kuuluvat joukkoon S?

- Ainoa mahdollisuus, että näin ei olisi, on alla kuvattu tilanne, jossa v:hen kulkisi vielä lyhempi polku $s \rightsquigarrow u \rightarrow x \rightarrow v$, eli missä solmusta u palattaisiin vielä johonkin joukon S solmuun x



- Kuvatun kaltaista polkua ei kuitenkaan voi olla olemassa, sillä koska x laitettiin joukkoon S ennen solmua u, ei lyhin polku solmuun x voi kulkea u:n kautta

- Algoritmi siis päivittää oikein joukon S ulkopuolisten solmujen etäisyysarviot eli väittämä (2) säilyy voimassa uusia solmuja lisättäessä

- On siis osoitettu, että

  (1) solmuille $v \in S$ pätee: $distance[v]$ on sama kuin lyhimmän polun $s \rightsquigarrow v$ pituus

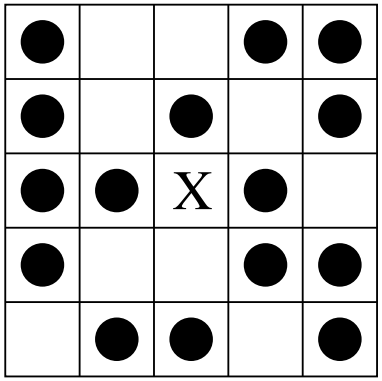
  (2) solmuille $v \in V \setminus S$ pätee: $distance[v]$ on lyhimmän tunnetun, eli käytännössä joukon S solmujen kautta kulkeva polun $s \rightsquigarrow v$ pituus

  ovat voimassa algoritmin suorituksen alussa ja pysyvät voimassa kun solmuja lisätään joukkoon S

- Lopussa kaikki solmut ovat joukossa S, eli väittämästä (1) seuraa, että algoritmi löytää lyhimmän polun kaikkiin solmuihin

Esimerkki ongelmanratkaisusta Dijkstralla: vankilapako

- 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



- Vanki on aluksi vankilan keskipaikassa $B[m][m]$, joka on käytävää
- Tavoitteena on päästä vankilasta sen jonkin reunan yli vapauteen
- Vankilassa saa liikkua seuraavasti:
  - Nykyisestä paikasta voi siirtyä mihin tahansa naapuripaikkaan, mutta ei kulmittain.
  - Käytävillä voi hiiviskellä, mutta niillä ei kannata maleksia turhaan
  - Muuriin täytyy ensin kaivautua ennen kuin siihen voi kulkea

- Pakosuunnitelmassa on tärkeintä, että ei kaiveta yhtään enempää kuin aivan välttämätöntä.
- Myös hiiviskelyä pitäisi välttää, jos se on mahdollista
- Mallinnetaan pakoreitti lyhyimpänä polkuna verkossa G
  - $V =$ vankilan paikat sekä lisäpaikka t vapaudessa
  - Kaari $(p, q) \in E$ jos ja vain jos paikka q on paikan p naapuripaikka
  - Jokaisen reunapaikan naapurina on myös lisäpaikka t
- Kaarilla on painot niiden maalisolmun (kohdepaikan) q mukaan:
  - Jos q on käytävllä, $w(p, q) = 1$
  - Jos q on muurilla $w(p, q) = (2 \cdot m + 1)^2$ eli suurempi kuin koko vankilan pinta-ala (ilman aloituspaikkaa)
  - Jos q on vapaudessa eli $q = t$, $w(p, q) = 0$
- Näillä kaaripainoilla yksikin kaivautuminen johtaa suurempaan etäisyyteen kuin pisinkään hiiviskely

- Eräs mahdollisimman hyvä pakosuunnitelma saadaan siis seuraavasti:
  - Suoritetaan **Dijkstran algoritmia** verkossa G alkusolmusta $B[m][m]$ lähtien, kunnes solmu t liitetään joukkoon S
  - Tällöin pakosuunnitelma saadaan seuraamalla taulukon $path$ polkutietoja solmusta t taaksepäin ja kääntämällä saadun polun järjestys:

$$s \to \ldots \to path[path[t]] \to path[t] \to t$$

- Verkkoa G ei tarvitse tallentaa erikseen, vaan vieruslistat lasketaan algoritmin suorituksen aikana nykyisen solmun indekseistä i, j:
  - Solmun $B[i][j]$ mahdolliset vierussolmut ovat $B[i+1][j]$, $B[i-1][j]$, $B[i][j+1]$ ja $B[i][j-1]$
  - Indeksoinnin ylitys tarkoittaa lisäsolmua t
  - Kaaripaino katsotaan kohdesolmusta

- Vankilapako on (yli)yksinkertaistettu version VLSI-piirisuunnittelusta, jossa komponentti sijoitetaan piisirulle valitun ruudukon pisteisiin ja minimoitavia suureita ovat mm. yhteyksien pituus ja kytkentöjen risteyskohtien lukumäärä
  - Käytännössä VLSI-suunnittelun kriteerit ovat monimutkaisempia ja algoritmisesti hankalia

Kaikki lyhimmät polut

- Merkintöjen yksinkertaistamiseksi oletetaan, että solmujoukko on $V = \{1, \ldots, n\}$, missä n on solmujen lukumäärä
- Halutaan muodostaa $n \times n$-matriisi D, missä $D[i, j]$ on lyhimmän polun paino $i \rightsquigarrow j$, eli halutaan selvittää jokaisen solmuparin välisen lyhimmän polun paino
- Jos kaarten painot ovat ei-negatiivisia, ongelma voidaan ratkaista suorittamalla **Dijkstran algoritmi** n kertaa
- Olettamalla prioriteettijono toteutetuksi kekona saadaan aikavaativuudeksi $\mathcal{O}(|V| \cdot (|V| + |E|) \log |V|) = \mathcal{O}((|V|^2 + |V||E|) \log |V|)$. Jos verkko on tiheä (kaaria on melkein jokaisen solmun välillä) eli $|E| = \mathcal{O}(|V|^2)$, tämä on $\mathcal{O}(|V|^3 \log |V|)$
- **Floydin-Warshallin** algoritmi tai **Floydin** algoritmi (Robert Floyd, 1962, aikaisemmin Bernard Roy, 1959, ja myös Stephen Warshall, 1962) ratkaisee ongelman ajassa $\mathcal{O}(|V|^3)$, mikä siis tiheillä verkoilla on asymptoottisesti parempi kuin **Dijkstran** toistaminen
- Lisäksi **Floyd-Warshall** sallii negatiiviset painot (mutta ei negatiivisia syklejä), on helppo toteuttaa ja vakiokertoimet ovat pieniä

- Verkko oletetaan annetuksi vierusmatriisina $A[1\,..\,n, 1\,..\,n]$. Oletetaan $A[i,j] = \infty$, jos $(i,j) \notin E$. Jos verkko on vieruslistamuodossa, on tieto kaarista konvertoitavissa matriisimuotoon ajassa $\mathcal{O}(|V|^2)$

- Algoritmi laskee välituloksenaan etäisyysmatriiseja $D^0, D^1, D^2, \ldots D^n$

- Matriisin D^k alkio $D^k[i,j]$ pitää yllä tietoa siitä mikä on solmujen i, j lyhin etäisyys, jos niitä yhdistävä polku käyttää ainoastaan solmuja $1, 2, \ldots, k$, eli
  - $D^0[i,j]$ kertoo mikä on solmujen i ja j etäisyys jos niiden välisellä polulla ei ole mitään solmua, toisin sanoen, mikä on i:n ja j:n välisen mahdollisen suoran kaaren pituus eli $D^0[i,j] = A[i,j]$
  - $D^1[i,j]$ kertoo mikä on solmujen i ja j pienin etäisyys jos niiden välisellä polulla käydään korkeintaan solmussa 1
  - $D^2[i,j]$ kertoo mikä on solmujen i ja j pienin etäisyys jos niiden välisellä polulla käydään korkeintaan solmuissa 1 ja 2 (molemmissa tai vain toisessa tai ei kummassakaan)
  - ...
  - $D^n[i,j]$ kertoo mikä on solmujen i ja j pienin etäisyys siten, että niiden välisellä polulla voidaan käydä missä tahansa verkon solmuista

- Eli matriisi D^n kertoo halutun lopputuloksen kaikille solmuille $i, j \in V$
- Miten matriisit $D^0, D^1, D^2, \ldots D^n$ saadaan laskettua?
  - D^0 siis saadaan suoraan siirtymämatriisista $D^0[i,j] = A[i,j]$
- Entä D^1 joka kertoo mikä on solmujen i ja j pienin etäisyys jos niiden välisellä polulla käydään korkeintaan solmussa 1?
  - solmujen i ja j pienin etäisyys silloin kun niiden välinen polku käy korkeintaan solmussa 1 on selvästi joko kaaren $i \to j$ paino tai polun $i \to 1 \to j$ pituus
  - jälkimmäisessä vaihtoehdossa polku $i \rightsquigarrow j$ siis kulkee solmun 1 kautta
  - eli $D^1[i,j] = min\{D^0[i,j], D^0[i,1] + D^0[1,j]\}$
- Entä D^2 joka kertoo mikä on solmujen i ja j pienin etäisyys jos niiden välinen polku saa käydä solmuissa 1 ja 2?
  - $D^1[i,j]$ kertoo pienimmän etäisyyden jos välissä käydään korkeintaan solmussa 1
  - vielä lyhempi polku voi löytyä, jos kuljetaan solmun 2 kautta $i \rightsquigarrow 2 \rightsquigarrow j$, tämä ei tosin ole välttämättä lyhempi kuin jo tunnettu solmua 2 hyödyntämätön polku $i \rightsquigarrow j$
  - eli $D^2[i,j] = min\{D^1[i,j], D^1[i,2] + D^1[2,j]\}$

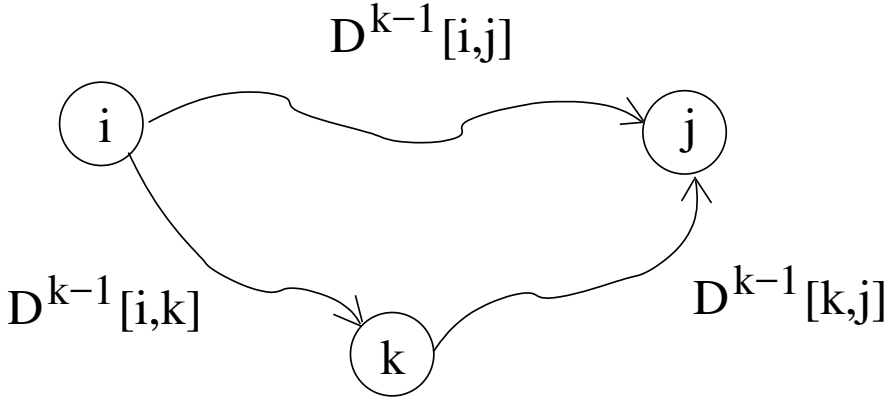
- $D^2[i,j]$ siis pitää sisällään lyhimmän polun i ja j välillä, joka voi olla joko
  - $i \to j$
  - $i \to 1 \to j$
  - $i \to 2 \to j$
  - $i \to 1 \to 2 \to j$
  - $i \to 2 \to 1 \to j$
- Huomattavaa on, että polku voi kulkea solmun 2 kautta kolmella eri tavalla
  - joko käymättä solmussa 1 tai kulkemalla ensin tai lopuksi solmun 1 kautta
  - tieto lyhimmästä korkeintaan solmun 1 kautta kulkevasta polun $i \rightsquigarrow 2$ pituudesta on huomioitu laskettaessa $D^1[i,2]$, polku on joko $i \to 2$ tai $i \to 1 \to 2$
  - vastaavasti lyhimmän korkeintaan solmun 1 kautta kulkevan polun $2 \rightsquigarrow j$ pituus on laskettu $D^1[2,j]$ siten että on huomioitu mahdollinen solmun 1 kautta kulkeminen
  - huom: polku $i \to 1 \to 2 \to 1 \to j$ ei voi tulla kyseeseen, sillä silloin $1 \to 2 \to 1$ olisi negatiivinen sykli, ja algoritmi ei toimi näissä tapauksissa
- Algoritmi siis näyttää käyvän läpi kaikki vaihtoehtoiset polut laskiessaan alkioiden $D^2[i,j]$ arvot

- Edellisestä yleistämällä saamme laskusäännön matriisin D^k alkioiden $D^k[i,j]$ arvoille:

  $D^k[i,j] = min\{\ D^{k-1}[i,j],\ D^{k-1}[i,k] + D^{k-1}[k,j]\ \}$

- Eli lyhin polku $i \rightsquigarrow j$ joka käyttää solmuja $1, 2, \ldots, k$ on sama joka ei käytä solmua k ollenkaan tai $i \rightsquigarrow k \rightsquigarrow j$ missä k:ta pienempiä solmuja ei ole käytetty

- Kuvana:



- D^0 saadaan suoraan siirtymämatriisista ja matriisin D^k kaikki arvot voidaan edellisen matriisin D^{k-1} avulla

- Näin on helppo muodostaa algoritmi joka selvittää matriisit numerojärjestyksessä päätyen matriisiin D^n joka on haluttu lopputulos

- Tästä saamme algoritmin ensimmäisen version

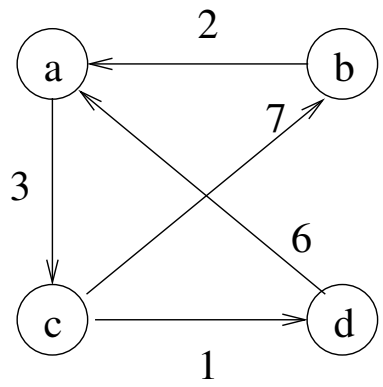
Floyd-Warshall-v1(A)
```
1   for i = 1 to n
2       for j = 1 to n
3           if i == j
4               D^0[i,j] = 0
5           else D^0[i,j] = A[i,j]
6   for k = 1 to n
7       for i = 1 to n
8           for j = 1 to n
9               D^k[i,j] = min { D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j] }
```

- Kolmen sisäkkäisen for-lauseen takia aikavaativuus on selvästi $\mathcal{O}(n^3)$

- Tilavaativuus on tässä versiossa $\mathcal{O}(n^3)$, sillä $\mathcal{O}(n^2)$:n kokoisia matriiseja on käytössä n kappaletta. Kuten pian havaitaan, apumatriisien D^k käyttöä voidaan tehostaa ja tilavaativuus saadaan putoamaan neliöiseksi

- Tarkastellaan seuraavilla sivuilla esimerkkiä algoritmin toiminnasta, selvyyden vuoksi solmut on nimetty aakkosin a, b, c ja d, matriisi D^1 vastaa lyhimpiä korkeintaan a:n kautta kulkevia polkuja, jne

- Verkko ja sitä vastaava siirtymämatriisi



$$D^0 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

- Matriisia D^1 laskettaessa selvitetään, lyhentääkö a:n kautta kulkeminen joidenkin solmujen välistä polkua

$$D^0 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array} \qquad D^1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

- Huomataan, että $b \rightsquigarrow a \rightsquigarrow c$ ja $d \rightsquigarrow a \rightsquigarrow c$ ovat lyhempiä kuin aiemmin tunnetut a:ta käyttämättömät polut

- Matriisia D^2 laskettaessa, huomataan, että b:n kautta kulkeminen lyhentää ainoastaan polkua $c \rightsquigarrow a$

$$D^1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array} \qquad D^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

- Matriisia D^3 laskettaessa selvitetään lyhentääkö c:n kautta kulkeminen joidenkin solmujen välistä polkua

$$D^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array} \qquad D^3 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

- Huomaamme, että esim. $a \rightsquigarrow c \rightsquigarrow b$ ja $a \rightsquigarrow c \rightsquigarrow d$ ovat lyhempiä kuin aiemmin tunnetut c:tä käyttämättömät polut

- Matriisia D^4 laskettaessa siis selvitetään lyhentääkö d:n kautta kulkeminen joidenkin solmujen välistä polkua, eli tässä vaiheessa on kaikki vaihtoehdot otettu huomioon ja algoritmi on selvittänyt halutun lopputuloksen

$$D^3 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array} \qquad D^4 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

- Tilantarve saadaan helposti pudotetuksi luokkaan $\mathcal{O}(n^2)$ toteamalla, että matriisin D^k laskemiseen ei enää tarvita matriiseja $D^0, D^1, \ldots, D^{k-2}$

- Riittää siis pitää muistissa kaksi viimeisintä matriisia $D = D^k$ ja $D' = D^{k-1}$

- Tästäkin voidaan säästää puolet: tarvitaan vain yksi matriisi D, jonka päivitys on $D[i,j] = \min\{\, D[i,j], D[i,k] + D[k,j] \,\}$

- Tämä seuraa siitä, että solmuun k rajautuvissa poluissa ei ole väliä, sallitaanko k välisolmuna, eli kaikilla i, j ja k pätee

$$D^k[i,k] = D^{k-1}[i,k] \qquad \text{ja} \qquad D^k[k,j] = D^{k-1}[k,j]$$

- Saadaan ajassa $\mathcal{O}(n^3)$ ja työtilassa $\mathcal{O}(n^2)$ toimiva algoritmi:

Floyd-Warshall-v2(A)

```
1   for i = 1 to n
2       for j = 1 to n
3           if i == j
4               D[i,j] = 0
5           else D[i,j] = A[i,j]
6   for k = 1 to n
7       for i = 1 to n
8           for j = 1 to n
9               if D[i,k] + D[k,j] < D[i,j]
10                  D[i,j] = D[i,k] + D[k,j]
```

- Algoritmin yhteydessä on mahdollista selvittää lyhimpien polkujen painojen lisäksi lyhimmät polut

- Jätämme tämän harjoitustehtäväksi

Transitiivinen sulkeuma

- Suunnatun verkon $G = (V, E)$ transitiivinen sulkeuma (engl. transitive closure) on verkko $G^* = (V, E^*)$, missä

  $(u, v) \in E^*$ jos ja vain jos verkossa G on polku $u \rightsquigarrow v$

- Esimerkki: Jos G on vahvasti yhtenäinen, niin $E^* = V \times V$

- Yleisemmin jos u ja v kuuluvat samaan vahvasti yhtenäiseen komponenttiin, niin kaikilla $w \in V$ pätee

  $(u, w) \in E^*$ jos ja vain jos $(v, w) \in E^*$
  $(w, u) \in E^*$ jos ja vain jos $(w, v) \in E^*$

- Tämän perusteella verkon transitiivinen sulkeuma voidaan laskea
  - muodostamalla ensin komponenttiverkko G^{SCC} ja
  - laskemalla sitten komponenttiverkon transitiivinen sulkeuma

- Tämä säästää laskentaa, jos G^{SCC} sisältää paljon vähemmän solmuja kuin G

- Transitiivinen sulkeuma voidaan laskea ajassa $\mathcal{O}(|\,V\,|^3)$ soveltamalla **Floydin-Warshallin algoritmia**:

  1. Asetetaan kaikille kaarille $(u, v) \in E$ jokin äärellinen paino, esim. $w(u, v) = 1$
  2. Suoritetaan Floydin-Warshallin algoritmi
  3. Nyt $(u, v) \in E^*$, jos ja vain jos $D[u, v] < \infty$

- Algoritmia voidaan hieman virtaviivaistaa: Pidetään kirjaa ainoastaan siitä, onko $D[i, j]$ äärellinen vai ääretön

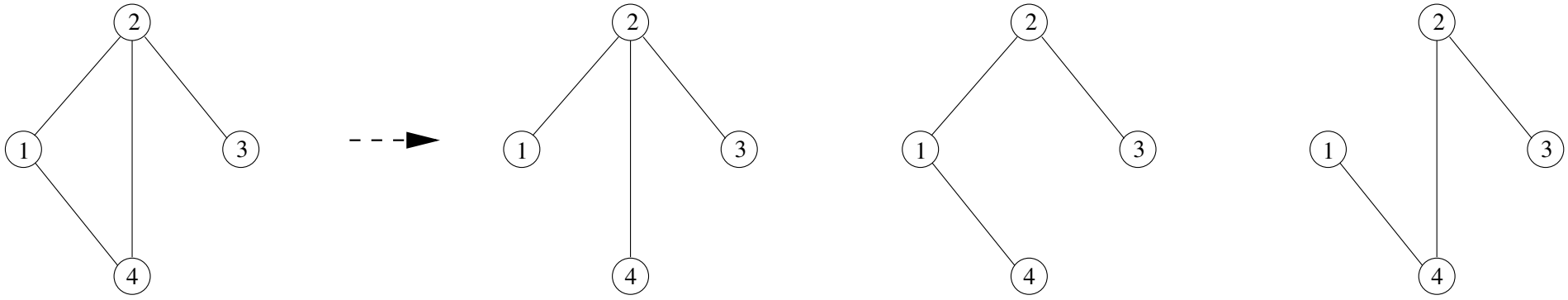
- Saadaan seuraava algoritmi:

```
Transitive-Closure(A)
1   for i = 1 to n
2       for j = 1 to n
3           if i == j or A[i,j] < ∞
4               T[i,j] = 1
5           else T[i,j] = 0
6   for k = 1 to n
7       for i = 1 to n
8           for j = 1 to n
9               if T[i,k] = 1 and T[k,j] = 1
10                  T[i,j] = 1
```

- Ajassa $\mathcal{O}(n^3)$ saadaan taulukko T, missä $T[i,j] = 1$, jos $(i,j) \in E^*$, ja $T[i,j] = 0$, muuten

Verkon virittävät puut

- Olkoon $G = (V, E)$ suuntaamaton yhtenäinen verkko
  - verkon yhtenäisyydellä tarkoitamme että kaikki verkon solmut ovat saavutettavissa toisistaan, eli
  - verkossa ei ole erillisiä osia
- Verkon G virittävä puu (engl. spanning tree) on G:n yhtenäinen syklitön aliverkko joka sisältää kaikki G:n solmut
- Huomio: virittävässä puussa on $|V| - 1$ kaarta
- Verkko ja sen virittäviä puita

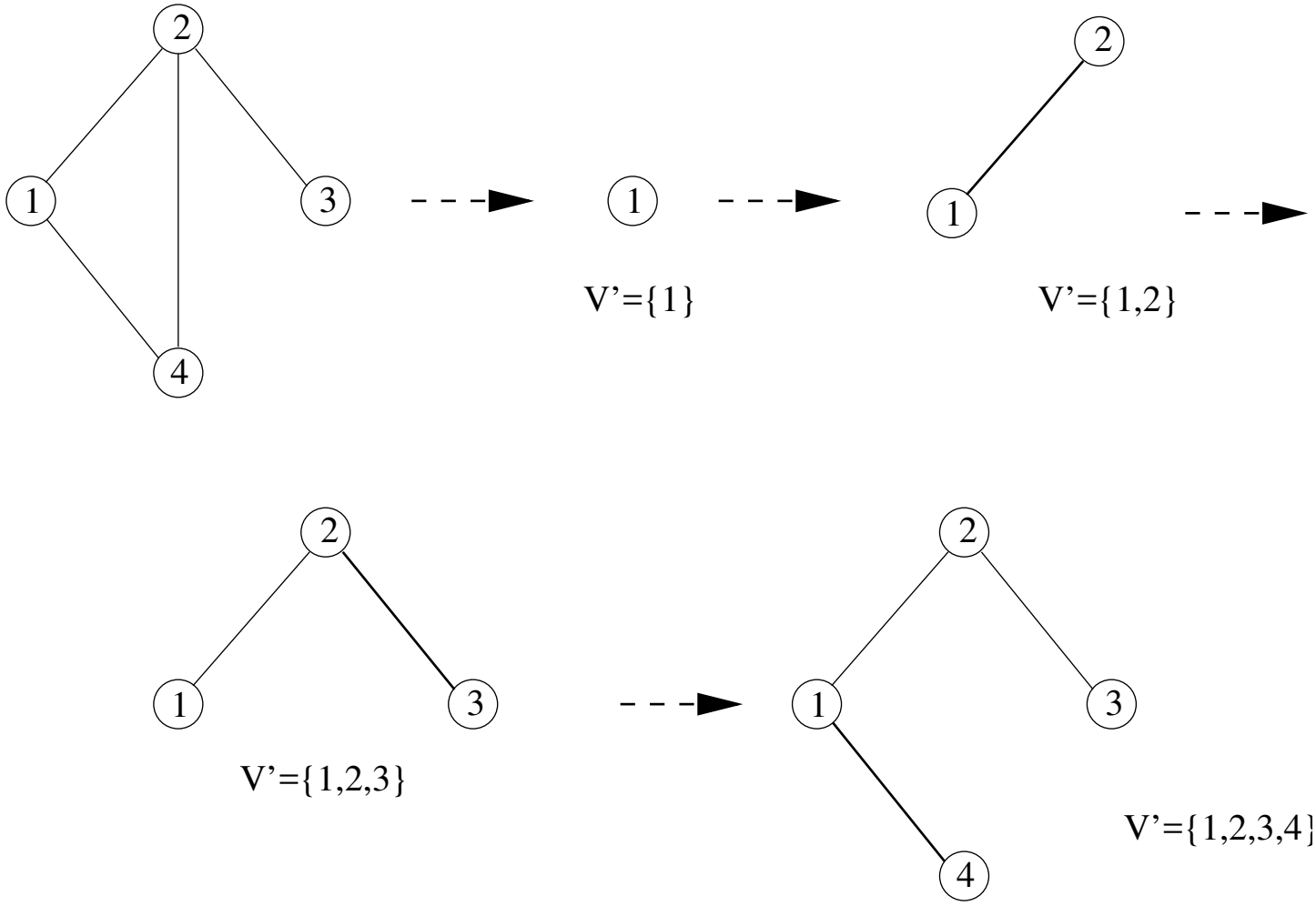


- Seuraavassa on yksinkertainen algoritmi, joka muodostaa verkolle $G = (V, E)$ jonkin virittävän puun

- Algoritmin lopussa joukossa T olevat kaaret muodostavat virittävän puun

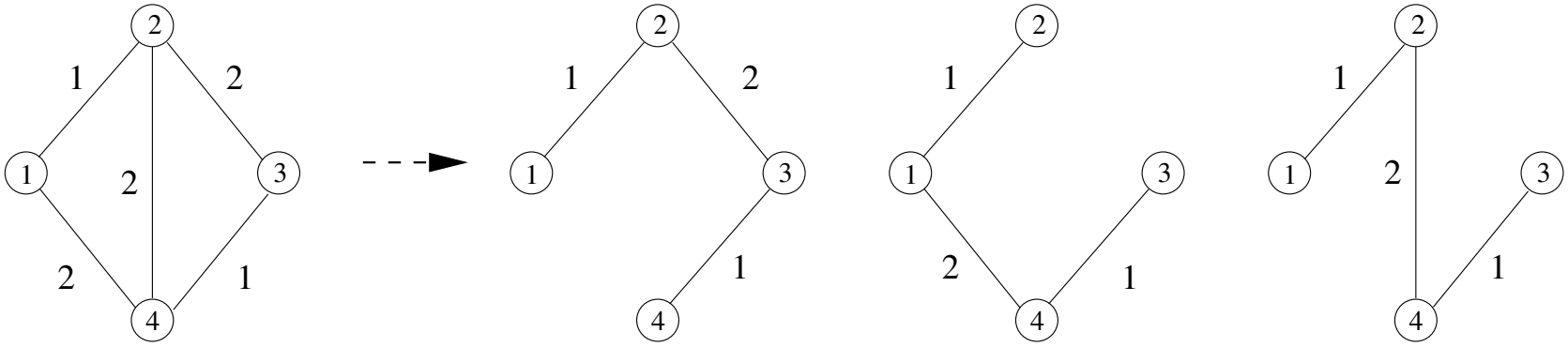
spanning-tree(G)
1 valitaan mielivaltainen solmu $v \in$ V
2 V'= $\{v\}$
3 T = $\emptyset$
4 while V' $\neq$ V
5 valitse kaari $(u, v) \in E$ siten että $u \in V'$ ja $v \in V \setminus V'$
6 V'= V' $\cup$ $\{v\}$
7 T = T $\cup$ $\{(u, v)\}$

- Esimerkki algoritmin toiminnasta:



- Virittäviä puita hyödynnetään useissa sovelluksissa
- Esim. tietokoneverkkoprotokollissa ja hajautetuissa algoritmeissa koneet joutuvat usein jakamaan tietoa keskenään, tämä hoidetaan muodostamalla virittävä puu ja käyttämällä sitä sanomien välittämiseen
- Äsken esitetty algoritmi muodostaa verkosta jonkin virittävän puun. Jos kyseessä on painotettu verkko, olemme yleensä kiinnostuneita pienimmän virittävän puun muodostamisesta
- Olkoon $G = (V, E)$ suuntaamaton yhtenäinen painotettu verkko, jonka kaaripainot määrää funktio w
- Verkon G pienin (tai minimaalinen) virittävä puu (engl. Minimum Spanning Tree, MST) on G:n virittävistä puista se jonka kaaripainojen summa on pienin

- Yhdellä verkolla voi olla useita pienimpiä virittäviä puita:



- Yleensä riittää että pystytään muodostamaan jokin minimaalisista virittävistä puista

- Pienimmän virittävän puun muodostamiseen esitetään kurssilla kaksi algoritmia:
  - **Primin algoritmi**
  - **Kruskalin algoritmi**

- Yleisperiaate on sama kuin aikaisemmin esitetyssä algoritmissa jonkun virittävän puun laskemiseen: puuhun lisätään kaaria yksi kerrallaan niin, että ei synny sykliä

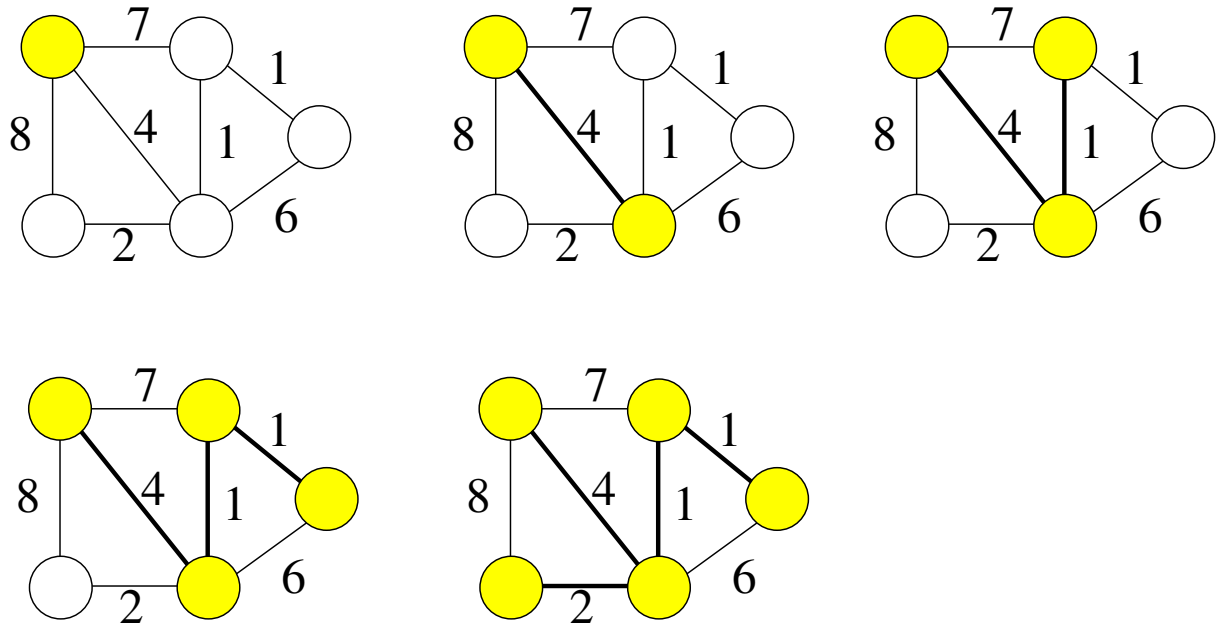
- Nyt kuitenkin kaaria lisätään puuhun määrätyssä järjestyksessä jotta virittävästä puusta saadaan pienin

Primin algoritmi [Vojtech Jarník, 1930, Robert C. Prim keksi uudestaan 1957 ja Edsger Dijkstra keksi uudestaan 1959]

- Algoritmin toimintaperiaate on sivun 560 algoritmin mukainen
  - rakenteilla on virittävä puu, jota kasvatetaan solmu kerrallaan
  - kun kaikki solmut on lisätty, on virittävä puu valmis
- Puuhun lisättäviä solmuja ei valita mielivaltaisesti
- Lisättäväksi solmuksi valitaan se, joka on lähimpänä jotakin jo puussa olevaa solmua
- Tälläisen solmun ja sen puuhun yhdistävän kaaren valinta näyttää aina suoritushetkellä parhaalta valinnalta
- Aina tietyllä hetkellä parhaalta näyttävän valinnan tekemistä sanotaan ahneeksi toimintastrategiaksi (engl. greedy), ja ahneita valintoja tekevää algoritmia ahneeksi algoritmiksi
- Ei ole selvää, että ahne valinta tuottaa aina optimaalisen lopputuloksen. Kuten pian näemme, **Primin algoritmin** kohdalla tilanne on kuitenkin näin
- Seuraavalla sivulla algoritmin hahmotelma

- Puun muodostaminen aloitetaan jostain verkon solmusta r. Tämä aloitussolmu voidaan valita vapaasti
- Algoritmi kerää pienimmän virittävän puun muodostavat kaaret joukkoon T
- Virittävän puun ulkopuolella olevista solmuista pidetään kirjaa joukossa H
- Aluksi T on tyhjä ja H sisältää kaikki solmut
- Ensimmäisellä kierroksella r lisätään virittävään puuhun, eli poistetaan joukosta H
- Tämän jälkeen jokaisella kierroksella algoritmi lisää virittävään puuhun solmun, joka on lähimpänä jotain virittävässä puussa jo olevaa solmua
  - Lisättävä solmu on siis se, johon kohdistuu painoltaan pienin niistä kaarista, jotka yhdistävät jotain virittävään puuhun jo kuuluvaa ja jotain siihen kuulumatonta eli joukossa H olevaa solmua
  - uuden solmun virittävään puuhun yhdistävä kaari lisätään joukkoon T
  - lisätty solmu poistetaan joukosta H
- Kun joukko H on tyhjä, pienin virittävä puu on valmis ja koostuu joukon T kaarista

- Ennen yksityiskohtaisempaa algoritmiesitystä tarkastellaan esimerkkiä
- Algoritmi aloittaa vasemman yläreunan solmusta, tummennetut kaaret muodostavat pienimmän virittävän puun



- Solmut siis liittyvät virittävään puuhun yksitellen, siten että jotain virittävässä puussa jo olevaa solmua lähimpänä oleva solmu liitetään puuhun kullakin kierroksella
- Algoritmin tehokkaan toteutuksen kannalta onkin oleellista että lähimpänä puun valmista osaa oleva solmu löytyy nopeasti

- Algoritmi käyttää aputaulukkoa $distance$, johon on talletettuna jokaiselle solmulle sen lyhin etäisyys johonkin jo virittävävässä puussa olevaan, eli joukon H ulkopuolella olevaan solmuun (eli ei etäisyyttä r:stä niin kuin Dijkstran algoritmissa)

- Aluksi asetetaan $distance[r] = 0$ ja kaikille muille solmuille $distance[v] = \infty$

- Jokaisella kierroksella lisätään puuhun se solmu u, jolla $distance[u]$ on pienin niistä solmuista, jotka kuuluvat joukkoon H

- Joukko H toteuteutetaan minimikekona siten, että avainkenttänä toimii $distance[v]$ eli solmun etäisyys johonkin virittävässä puussa jo olevaan solmuun
  - operaation **heap-del-min**(H) avulla saadaan siis selville nopeasti solmu, josta on lyhin kaari jo virittävässä puussa olevaan solmuun

- Kun virittävään puuhun lisätään uusi solmu u, käydään u:n vieruslista läpi
  - Jos vieruslistan solmu v on vielä keossa H ja $w(u, v) < distance[v]$, niin $distance[v]$-arvoa pienennetään $w(u, v)$:ksi
  - Näin siis solmun v pienin etäisyys jo virittävässä puussa olevaan solmuun saadaan pidettyä ajan tasalla
  - Jotta vierussolmu, jonka $distance$-arvo muuttuu, pysyisi keossa H oikealla paikalla, kutsutaan sille **heap-decrease-key**-operaatiota

- Algoritmilla on käytössä myös aputaulukko $parent$
- Jos solmu u ei vielä ole virittävässä puussa, kertoo $parent[u]$ sen virittävässä puussa olevan solmun, josta on lyhin kaari solmuun u
- Alussa asetetaan kaikille solmuille $parent[u] = NIL$, sillä virittävässä puussa ei ole vielä yhtään solmua
- Kun virittävään puuhun lisätään uusi solmu u
  - virittävään puuhun tulee kaari $(parent[u], u)$, eli se lisätään joukkoon T
  - käytäessä u:n vieruslistaa läpi
    jos vieruslistan solmu v on vielä keossa H ja $w(u, v) < distance[v]$, arvon $distance[v]$ pienennyksen lisäksi asetetaan $parent[v] = u$, sillä u on virittävän puun solmu, josta on lyhin kaari solmuun v
- **Primin algoritmi** seuraavalla sivulla
  - Algoritmin syötteenä on verkko G ja sen kaaripainot määrittelevä funktio w sekä solmu r josta virittävän puun muodostaminen aloitetaan
  - Algoritmi palauttaa joukon T joka sisältää virittävän puun kaaret

```
Prim(G,w,r)
1   T = ∅
2   for kaikille solmuille v ∈ V
3       distance[v] = ∞
4       parent[v] = NIL
5   distance[r] = 0
6   for kaikille solmuille v ∈ V
7       heap-insert(H,v,distance[v])
8   while not empty(H)
9       u = heap-del-min(H)
10      if parent[u] ≠ NIL                    // ensimmäisen solmun lisäys ei tuo
11          T = T ∪ { (parent[u],u ) }        // virittävään puuhun kaarta
12      for jokaiselle solmulle v ∈ vierus[u]
13          if solmu v on vielä keossa H ja w(u,v) < distance[v]
14              parent[v] = u
15              distance[v] = w(u,v)
16              heap-decrease-key(H,v,distance[v])
17  return T
```

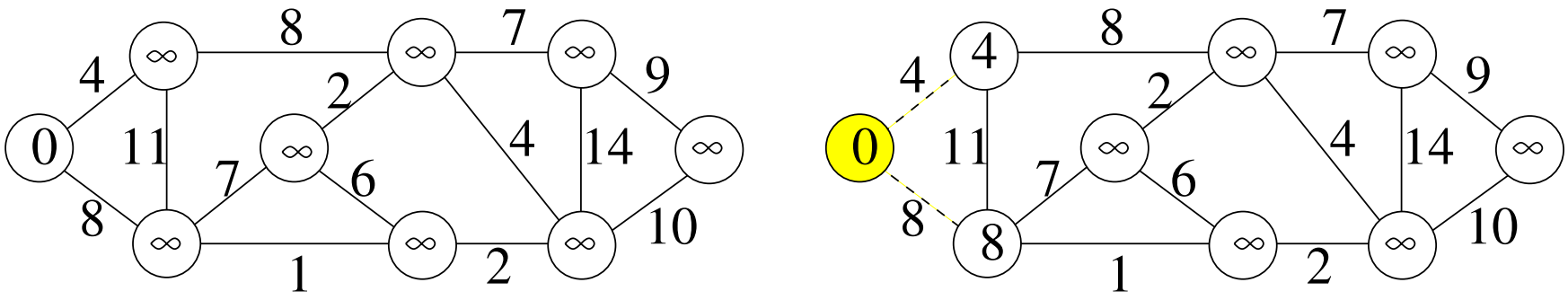
- Vaikka algoritmin toimintaperiaate onkin hyvin selkeä, mutkistavat toteutusyksityiskohdat algoritmia jossain määrin

- Algoritmin toimintaidea vielä kerran:
  - aluksi virittävä puu on tyhjä ja asetetaan kaikille solmuille paitsi r:lle etäisyyden $distance$ arvoksi ääretön (rivi 3)
  - myös lyhin kaari, joka yhdistää solmun virittävään puuhun on tässä vaiheessa tuntematon (rivi 4) kaikille solmuille
  - riveillä 6-7 solmut laitetaan minimikekoon H, avaimena siis solmun pienin etäisyys jostain virittävän puun solmusta joka on aluksi kaikille paitsi lähtösolmulle ääretön
  - ensimmäisessä toistossa tulee valituksi solmu r, joka siis poistuessaan keosta liittyy virittävään puuhun
  - kun virittävään puuhun liitetään solmu, pidetään voimassa ominaisuutta

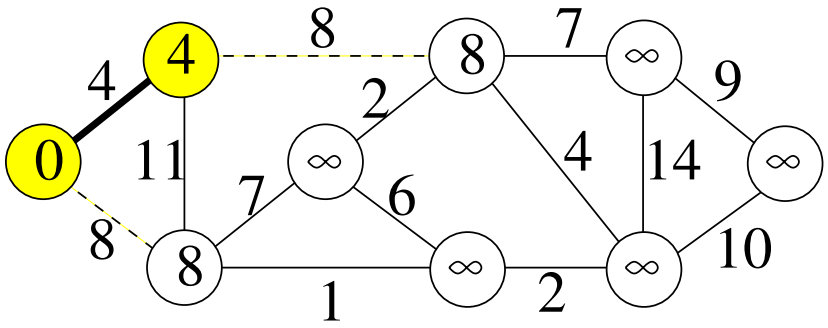
    jokaisen keossa olevan solmun v arvona $distance[v]$ on sen kaaren paino, joka on kevyin niiden kaarten joukossa jotka yhdistävät v:n konstruktion alla olevaan virittävään puuhun;

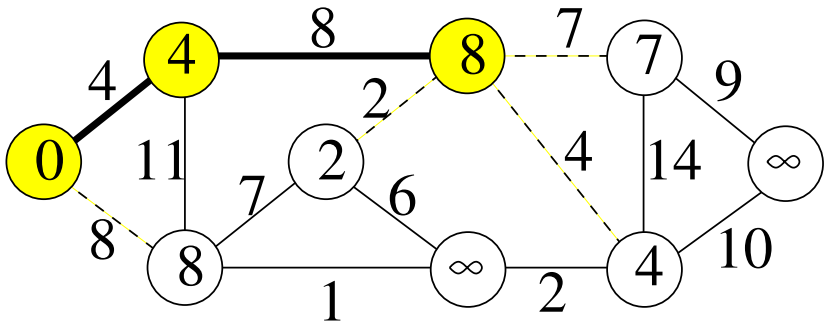
    kyseessä oleva kaari on $(parent[v], v)$
  - tämän jälkeen otetaan keosta käsittelyyn solmu jonka etäisyys virittävään puuhun on pienin ja liitetään solmu virittävään puuhun
  - riveillä 12-16 huolehditaan edelleen, että yllä oleva ominaisuus virittävän puun ulkopuolisten solmujen $distance$- ja $parent$-arvoille pysyy voimassa
  - jatketaan niin kauan kun solmuja on keossa jäljellä

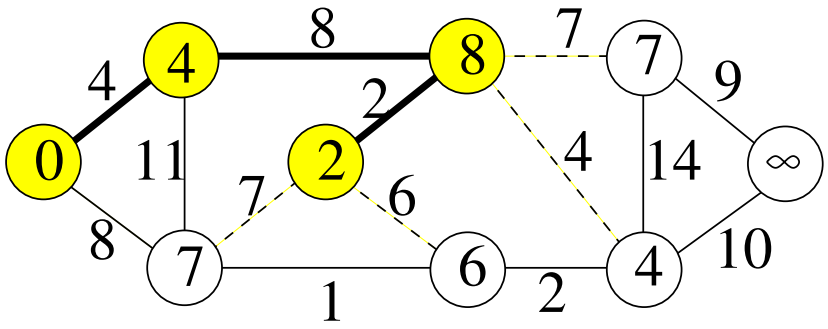
- Seuraavassa esimerkki algoritmin toiminnasta
- Taulukoiden $distance$ ja $parent$ informaatio on merkitty suoraan solmujen yhteyteen, myöskään keon H sisältöä ei eksplisiittisesti näytetä, kekoon kuuluvat kaikki värjäämättömät solmut
- Rivien 1-7 alustusvaiheen jälkeinen tilanne vasemmalla, eli kaikille paitsi aloitussolmulle on merkitty arvoksi $distance = \infty$
- Ensimmäisenä keosta poistetaan aloitussolmu, joka on värjätty oikeanpuoleisessa kuvassa, jossa on rivien 12-16 aikana suoritettujen keosta poistetun solmun vierussolmujen $distance$- ja $parent$- arvojen päivityksen jälkeinen tilanne
- $parent$-arvo, eli lyhin kaari virittävässä puussa olevaan solmuun on ilmaistu katkoviivallisena kaarena

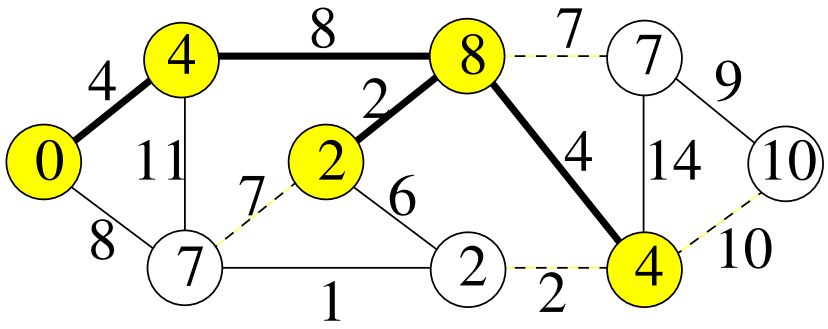


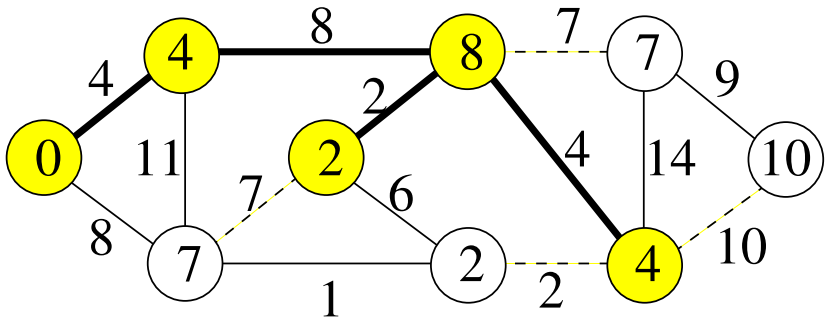
- Algoritmi jatkaa valitsemalla solmun, jonka $distance$-arvo on pienin
- Joukkoon T lisätään valitun solmun u rakenteilla olevaan puuhun yhdistävä kaari $(parent[u], u)$, joka on kuvassa merkitty tummennetulla
- Lisäyksen jälkeen päivitetään lisätyn solmun vierussolmujen $distance$- ja $parent$-arvot

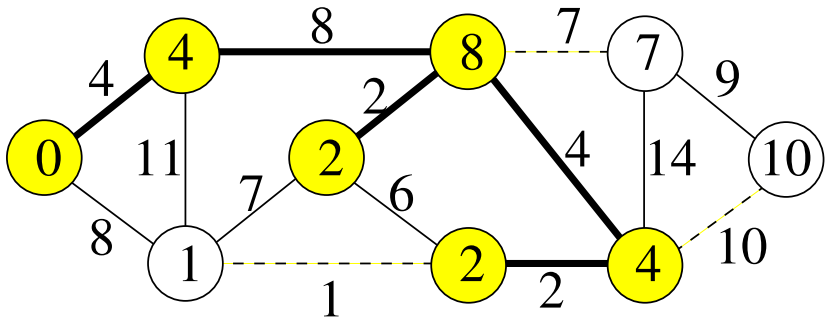


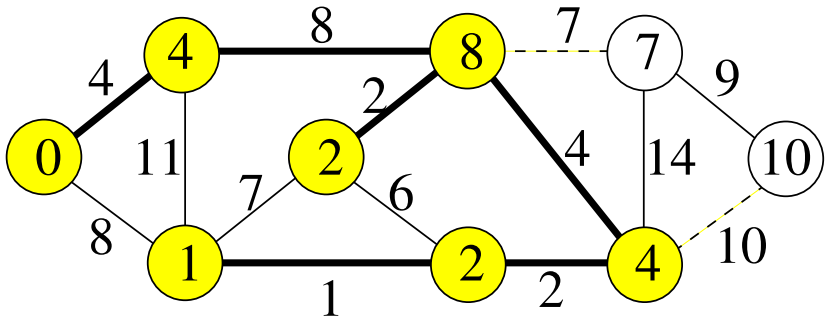


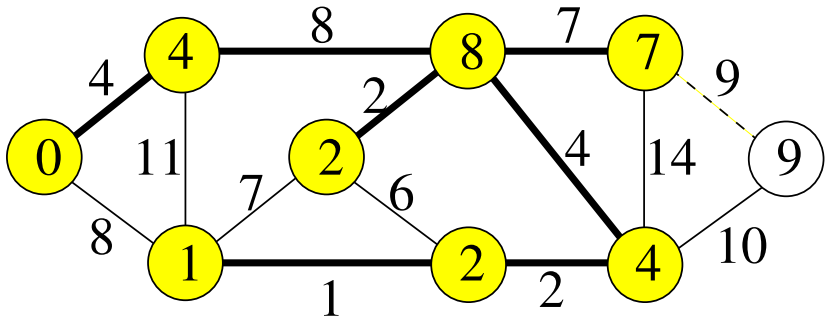


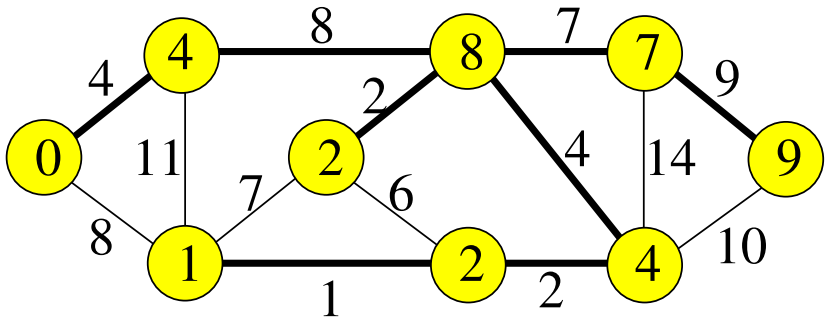



0
4
7
2
4
8
7
10
11
8
7
6
4
14
9
10
2
1
2


0
4
1
2
4
8
7
10
11
8
7
6
4
14
9
10
2
1
2


0
4
1
2
4
8
7
10
11
8
7
6
4
14
9
10
2
1
2


0
4
1
2
4
8
7
9
11
8
7
6
4
14
9
10
2
1
2


0
4
1
2
4
8
7
9
11
8
7
6
4
14
9
10
2
1
2

- **Primin algoritmin** aikavaativuus:
  - rivien 1-5 alkutoimet vievät aikaa $\mathcal{O}(|V|)$
  - algoritmi käyttää kekoa jossa pahimmillaan $|V|$ alkiota, kuten toivottavasti muistamme, kaikkien keko-operaatioiden aikavaatimus on $\mathcal{O}(\log |V|)$
  - riveillä 6-7 kutsutaan $|V|$ kertaa operaatiota **heap-insert**, eli aikaa kuluu $\mathcal{O}(|V| \log |V|)$; kuten Dijkstran algoritmissa, arvot ovat yksi nolla ja muut ääretön, joten tämä voitaisiin tehdä lineaarisessa ajassa
  - rivien 8-16 toistolauseessa operaatio **heap-del-min** suoritetaan kertaalleen jokaiselle solmulle, ja tähän kuluu aikaa $\mathcal{O}(|V| \log |V|)$
  - sisempi toistolause riveillä 12-16 suoritetaan $\mathcal{O}(|E|)$ kertaa sillä suuntaamattomassa verkossa kaikkien vieruslistojen yhteispituus on $2 \times |E|$
  - sisemmässä toistolauseessa suoritetaan **heap-decrease-key**-operaatio korkeintaan kertaalleen jokaisen kaaren yhteydessä, eli tästä koituva vaiva on $\mathcal{O}(|E| \log |V|)$
  - näin saamme **Primin algoritmin** aikavaativuudeksi $\mathcal{O}(|V| \log |V| + |E| \log |V|) = \mathcal{O}((|E| + |V|) \log |V|) = \mathcal{O}(|E| \log |V|)$ koska yhtenäisessä verkossa solmuja ei voi olla kuin korkeintaan yksi enemmän kuin kaaria
- Aputietorakenteena keko, ja alussa kaikki $|V|$ solmua ovat keossa eli tilavaativuus $\mathcal{O}(|V|)$

- Jos verkko on tiheä, eli $|E| = \Theta(|V|^2)$, voimme saada aikaan ajassa $\mathcal{O}(|V|^2)$ toimivan algoritmin

- Voimme etsiä solmun, jolla on pienin $distance$-arvo, lineaarisessa ajassa

```
Prim-Dense(G,w,r)
1   S = {r}
2   distance[r] = 0
3   for kaikille solmuille v ∈ V \ {r}
4       distance[v] = w(r,v) // ääretön, jos kaari puuttuu
5       parent[v] = r
6   while S ≠ V
7       valitse u ∈ V \ S, jolla distance[u] on pienin
8           T = T ∪ {(parent[u],u)}
9       for jokaiselle solmulle v ∈ vierus[u]
10          if v ∉ S and w(u,v) < distance[v]
11              parent[v] = u
12              distance[v] = w(u,v)
13  return T
```

Primin algoritmin oikeellisuus

- **Primin algoritmi** noudattaa ahnetta strategiaa: lisätään aina virittävään puuhun lähimpänä oleva puun ulkopuolinen solmu
- Ei ole aivan itsestään selvää että algoritmi tuottaa nimenomaan pienimmän virittävän puun
- Todistetaan, että **Primin algoritmi** todella tuottaa pienimmän virittävän puun
- Todistuksen idea:
  - Vaihdetaan joku algoritmin tuottaman virittävän puun kaari toiseksi
  - Osoitetaan, että näin saatavan virittävän puun paino ei voi olla pienempi kuin alkuperäisen virittävän puun paino

- Osoitetaan ensin seuraava aputulos, jonka avulla voidaan todistaa, että **Primin algoritmi** (ja seuraavaksi esitettävä **Kruskalin algoritmi**) tuottaa pienimmän virittävän puun

- **Lemma 8.8.:** Oletetaan suuntaamattomasta yhtenäisestä verkosta $G = (V, E)$, että

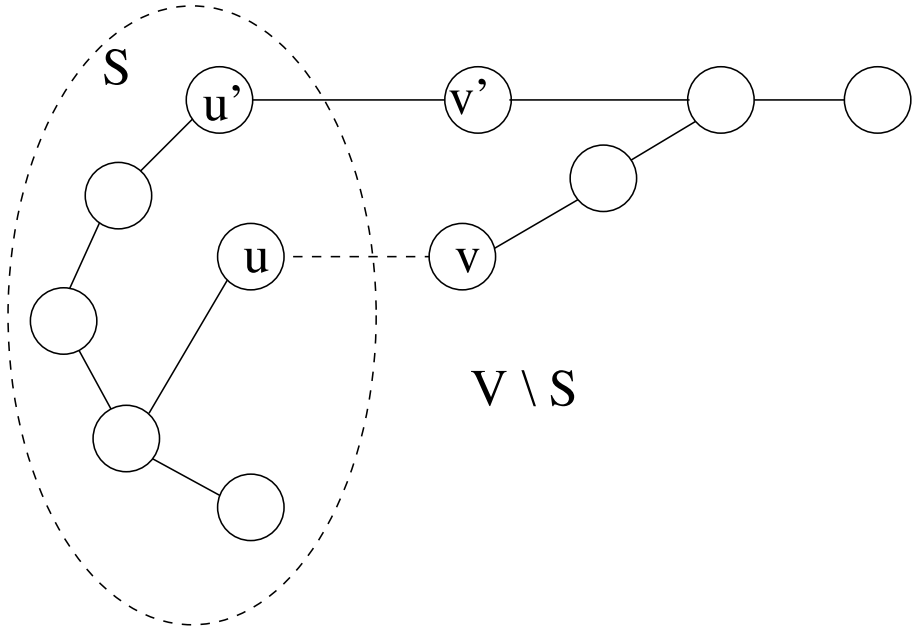
  1. sen solmut on jaettu kahteen epätyhjään osajoukkoon S ja $V \setminus S$
  2. kaari (u, v) kulkee osajoukosta toiseen eli $u \in S$ ja $v \in V \setminus S$
  3. kaari (u, v) on painoltaan pienin kaikista tällaisista osajoukkojen S ja $V \setminus S$ välisistä kaarista
  4. kaarijoukko $F \subseteq T$, jollekin pienimmälle virittävälle puulle (V, T)
  5. ei ole kaaria F:ssä, jotka kulkevat osajoukosta toiseen, eli S:n ja $V \setminus S$:n välillä

  Nyt $F \cup \{(u, v)\} \subseteq T'$ jollekin pienimmälle virittävälle puulle (V, T').

- **Todistus:** Jos $(u, v) \in T$, niin väite on selvä. Oletamme nyt, että (u, v) ei kuulu väitteessä mainittuun pienimpään virittävään puuhun T.

  Nyt verkossa $(V, T \cup \{(u, v)\})$ on sykli, joka sisältää kaaren (u, v). Koska tämä kaari kulkee joukkojen S ja $V \setminus S$ välillä, syklillä on oltava toinen kaari (u', v'), joka myös kulkee joukkojen S ja $V \setminus S$ välillä. Koska F:ssä ei ole kaaria osajoukkojen välillä, tiedämme että $(u', v') \notin F$.

Poistetaan T:stä (u', v') ja lisätään (u, v). Lopputulos on virittävä puu (V, T'), missä $T' = T \cup \{(u, v)\} \setminus \{(u', v')\}$. Koska (u, v) oli pienin kaari joukkojen S ja $V \setminus S$ välillä, tiedämme, että $w(u, v) \leq w(u', v')$.



Uuden puun painoksi saamme siis

$$\sum_{e \in T'} w(e) = \sum_{e \in T} w(e) + w(u, v) - w(u', v') \leq \sum_{e \in T} w(e).$$

Mutta T oli pienin virittävä puu, joten myös T':n on oltava, eli (u, v) kuuluu pienimpään virittävään puuhun T'. □

- Huomautus: Siis $w(u, v) = w(u', v')$

- **Primin algoritmi** on helppo osoittaa oikeaksi Lemmaan 8.8 vedoten
- Tarkastellaan sitä hetkeä, kun algoritmi päättää lisätä kaaren $e = (u, v)$ joukkoon T
- Valitaan nyt joukoksi S ne solmut, jotka on jo poistettu keosta H
- Kaari e on nyt kevyin kaari joukkojen S ja $V \setminus S$ välillä, eli sen täytyy lemman perusteella kuulua johonkin minimaaliseen virittävään puuhun
- Jokaisen algoritmin valitseman kaaren siis täytyy kuulua verkon minimaaliseen virittävään puuhun
- Algoritmi pitää huolen siitä, että T pysyy koko ajan puuna ja lopuksi se kattaa kaikki verkon G solmut
- Huomautus: Jos on samanpainoisia kaaria, pienin virittävä puu ei ole välttämättä yksikäsitteinen

Kruskalin algoritmi [Joseph Kruskal, 1956]

- Algoritmin suorituksen aikana pidetään yllä ns. virittävää metsä (erillisiä puita), joita vähitellen yhdistellään
- Jotta emme menisi sekaisin liiallisista puista, puhumme tässä paloista
- Aluksi verkon jokainen solmu on oma palansa eikä paloihin kuulu lainkaan kaaria
- Jokaisessa vaiheessa valitaan painoltaan pienin kaari, joka yhdistää kaksi tähän asti erillistä palaa. Näin palojen määrä pienenee yhdellä ja valittu kaari kuuluu pienimpään virittävään puuhun
- Kun jäljellä on vain yksi pala, on se verkon pienin virittävä puu
- Kaaret järjestetään ensin painonsa mukaiseen kasvavaan järjestykseen
- Jokaisella kierroksella tutkitaan järjestyksessä seuraavaa kaarta (u, v):

  jos solmut u ja v kuuluvat tällä hetkellä eri paloihin, kaari (u, v) otetaan mukaan minimaaliseen virittävään puuhun ja u:n ja v:n sisältävät palat yhdistetään

- Algoritmin ensimmäisessä versiossa tieto siitä mihin palaan solmut kuuluvat on talletettu taulukkoon $pala$

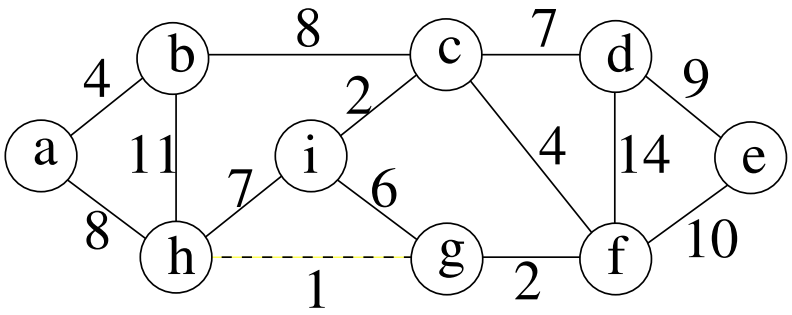
- Algoritmi kerää virittävän puun kaaria joukkoon T ja palauttaa lopuksi joukon

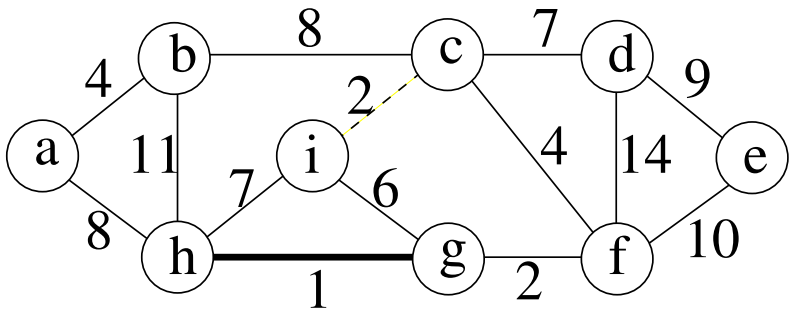
- Algoritmin lopussa verkon pienin virittävä puu siis koostuu joukon T kaarista

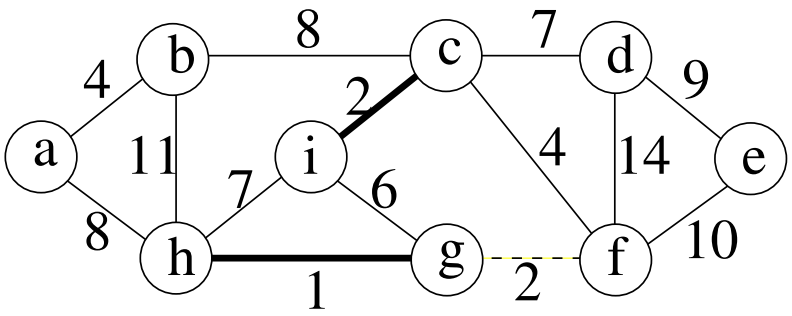
```
Kruskal(G,w)
1   T = ∅
2   for kaikille solmuille v ∈ V
3       solmu v muodostaa oman palan pala[v]
4   järjestetään kaaret painon w mukaan kasvavaan järjestykseen
5   for jokaiselle kaarelle (u,v) ∈ E kasvavassa järjestyksessä
6       if pala[u] ≠ pala[v]
7           T = T ∪ {(u,v)}
8           yhdistä pala[u] ja pala[v]
9   return T
```

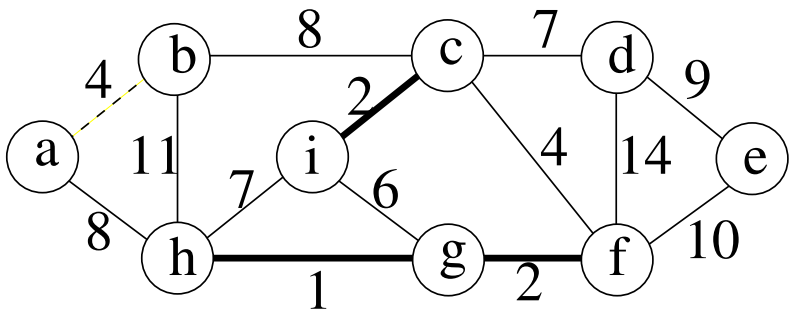
- Esimerkki algoritmin toiminnasta seuraavalla sivulla:

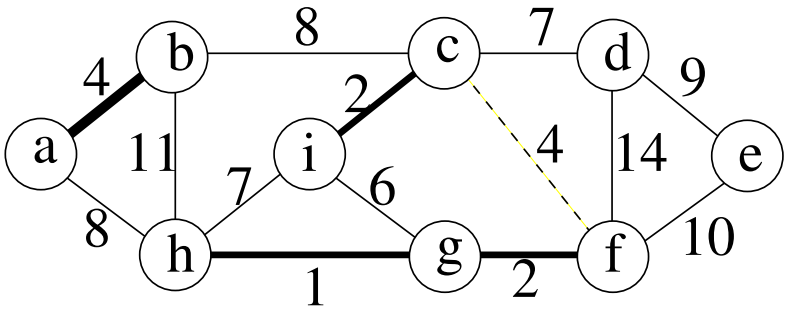
- Aluksi jokainen solmu muodostaa oman palansa
- Paloja yhdistellään siten, että aina pyritään yhdistämään lähimpänä toisiaan olevat palat
  - algoritmi on järjestänyt kaaret kasvavaan pituusjärjestykseen joten kaarien järjestys määrää palojen yhdistelyjärjestyksen
- Paloja yhdistävä kaari (kuvassa paksunnettu) tulee osaksi minimaalista virittävää puuta

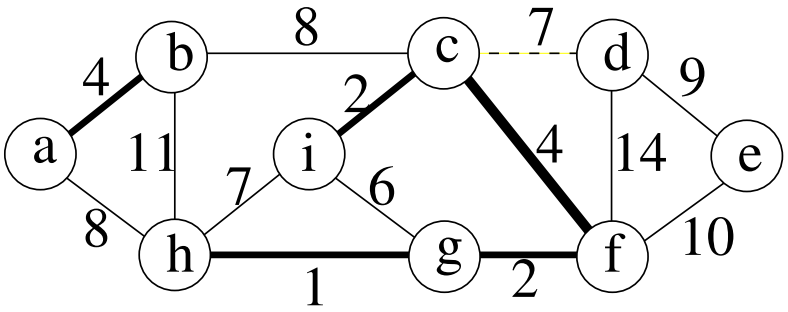


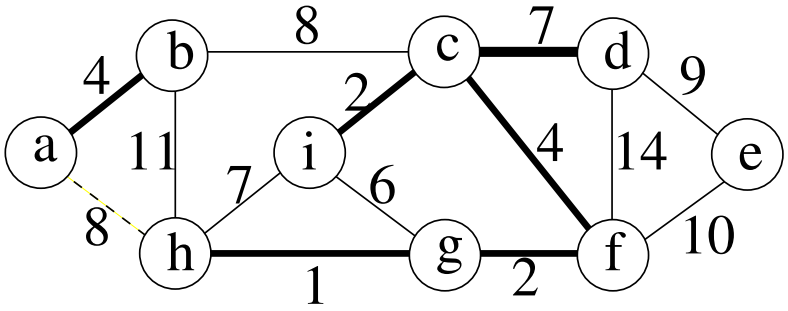


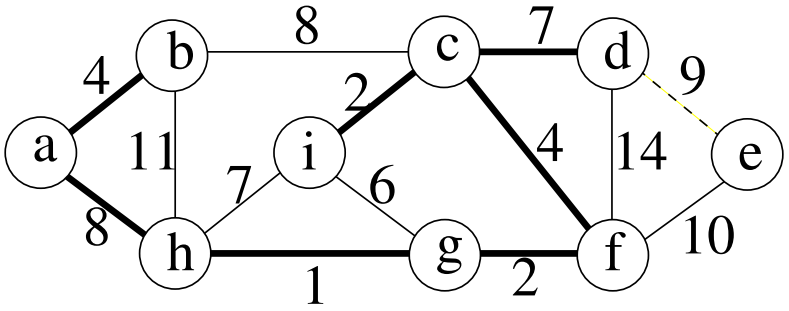


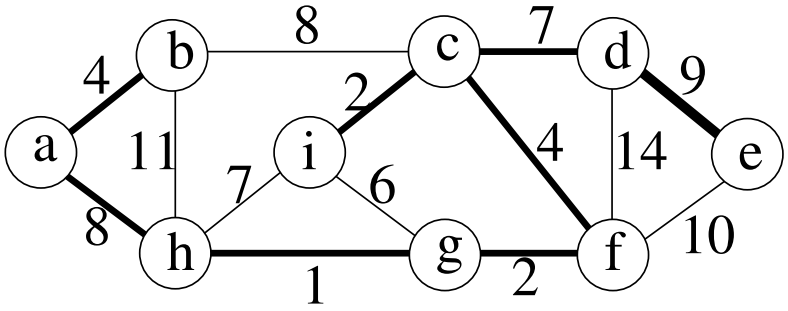



a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
10
7
6
1
2


a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
10
7
6
1
2


a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
10
7
6
1
2


a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
10
7
6
1
2


a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
14
9
10
7
6
1
2

- Algoritmi itsessään on hyvin yksinkertainen mutta sen toteuttaminen tehokkaasti ei välttämättä ole aivan yksinkertaista
- Jotta toteutuksesta tulee tehokas, on erityisesti kiinnitettävä huomiota siihen miten solmuihin liittyvä palainformaatio toteutetaan
- Suoraviivainen tapa palainformaation tallettamiseen siis on käyttää $|V|$-paikkaista taulukkoa $pala$
  - oletetaan että palat on numeroitu, ja asetetaan alussa kunkin solmun palaksi oma luku
  - rivin 6 vertailu on nyt helppo ja hoituu järkevästi toteutettuna vakioajassa
  - rivillä 8 on yhdistettävä kaksi palaa yhdeksi
  - riittää kun asetetaan kaikille solmuille v minkä palanumerona on $pala[v]$ uudeksi palanumeroksi $pala[u]$
  - palat tallettava taulukko on siis käytävä läpi ja näin rivin 8 aikavaativuus on luokkaa $\mathcal{O}(|V|)$

- Koko algoritmin aikavaativuus:
  - oletetaan edellä kuvailtu suoraviivainen tapa toteuttaa palat
  - alkutoimet riveillä 1-3 vievät aikaa $\mathcal{O}(|V|)$
  - kaarten järjestäminen suuruusjärjestykseen, eli rivin 4 suorittaminen onnistuu ajassa $\mathcal{O}(|E| \log |E|)$
  - for-lause käy läpi kaikki $|E|$ kaarta
  - rivin 6 ehto toteutuu $|V| - 1$ kertaa, ja jokaisella näistä kerroista täytyy siis suorittaa $\mathcal{O}(|V|)$ vievä palat yhdistävä operaatio
  - for-silmukan kokonaissuoritusaika on siis $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$
  - saamme koko algoritmin aikavaativuudeksi $\mathcal{O}(|E| \log |E| + |V|^2)$
- Käyttämällä kohta esiteltävää union-find-tietorakennetta palojen toteuttamiseen päästään algoritmissa aikavaativuuteen $\mathcal{O}(|E| \log |E|)$

Kruskalin algoritmin oikeaksi todistus

- Myös **Kruskalin algoritmi** noudattaa ahnetta strategiaa:
  - jokaisessa vaiheessa yhdistetään kaksi toisiaan lähimpänä olevaa palaa
  - eikä ole aivan itsestään selvää että algoritmi tuottaa nimenomaan pienimmän virittävän puun
- **Primin algoritmin** tapaan **Kruskalin algoritmin** oikeellisuus perustuu suoraan Lemmaan 8.8
- Tarkastellaan sitä hetkeä, kun algoritmi päättää lisätä kaaren $e = (u, v)$ tulosjoukkoonsa T
- Valitaan osajoukoksi S ne solmut jotka kuuluvat samaan palaan kuin solmu u, ja F on ne kaaret, jotka ovat jo T:ssä. Koska $pala[u] \neq pala[v]$, niin $v \notin S$
- Lemman perusteella $F \cup \{ e \}$ on nyt osaa jotakin verkon G minimaalista virittävää puuta
- Algoritmin päättyessä T on puu ja siihen on lisätty vain sellaisia kaaria, jotka lemman perusteella kuuluvat minimaaliseen virittävään puuhun
- Kuten edellä **Primin algortimin** tapauksessa, jos on samanpainoisia kaaria, pienin virittävä puu ei ole välttämättä yksikäsitteinen

Union-find-tietorakenne

- Union-find-rakenne on tarkoitettu sovelluksiin, joissa seuraavat operaatiot on pystyttävä toteuttamaan tehokkaasti:
  - **make-set**(x): luo uusi yksialkioinen joukko
  - **find**(x): selvitä, mihin joukkoon parametrina annettu alkio x kuuluu
  - **union**(x, y): liitä yhteen kaksi joukkoa
- Joukot ovat erillisiä eli jokainen alkio kuuluu koko ajan tasan yhteen joukkoon
- Mikään joukko ei voi sisältää kahta tai useampaa samaa alkiota
- Joukon nimenä käytetään joukon edustajaa eli yhtä joukon alkiota. Joukon edustaja pysyy samana niin kauan kuin joukkoa ei muuteta

- Operaatioiden tarkempi määrittely:
  - **make-set**(x): luo uusi yksialkioinen joukko. Joukko sisältää alkion x ja sen nimeksi tulee x
  - **find**(x): selvitä, mihin joukkoon parametrina annettu alkio x kuuluu. Palauta tämän joukon nimi (ts. joukon edustaja)
  - **union**(x, y): liitä yhteen kaksi joukkoa, joiden nimet ovat x ja y. Operaatiota kutsutaan vain, jos $x \neq y$. Yhdistetyn joukon nimeksi tulee joko x tai y

  Kruskalin algoritmissa palat voidaan toteuttaa tällaisina joukkoina:
  - Aluksi luodaan oma joukko jokaista solmua varten
  - Kun halutaan selvittää, kuuluvatko solmut u ja v eri paloihin, tutkitaan, onko **find**(u) $\neq$ **find**(v)
  - Palat $pala[u]$ ja $pala[v]$ yhdistetään operaatiolla **union**(**find**(u), **find**(v))

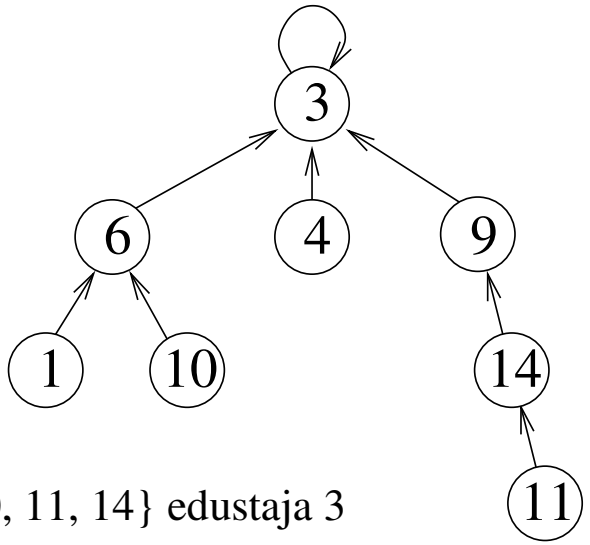
- **Kruskalin algoritmi** union-find-rakenteen avulla

```
Kruskal2(G,w)
1   T = ∅
2   for kaikille solmuille v ∈ V
3       make-set(v)
4   järjestetään kaaret (u,v) ∈ E painon mukaan kasvavaan järjestykseen
5   while joukkojen lukumäärä > 1
6       ota järjestyksessä seuraava kaari (u,v) ∈ E
7       if find(u) ≠ find(v)
8           T = T ∪ {(u,v)}
9           union( find(u), find(v) )
10  return T
```

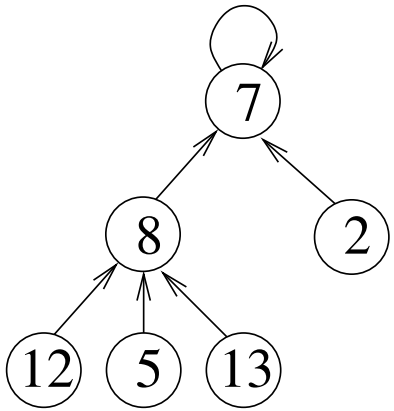
- Algoritmin tehokkuus perustuu siihen, että union-find-rakenne voidaan toteuttaa niin, että jono peräkkäisiä **union**- ja **find**- operaatiota voidaan suorittaa selvästi nopeammin kuin vastaavat operaatiot, jos palat toteutettaisiin taulukon avulla

Union-find-rakenteen toteutus

- Union-find-rakenne toteutetaan puiden avulla
- Jokaista joukkoa kuvaa yksi puu
- Puun juuressa on sen edustaja eli se alkio, joka antaa joukolle nimen
- Puussa jokaisesta solmusta on linkki sen vanhempaan, mutta ei lapsiin. Juuren vanhempi on juuri itse

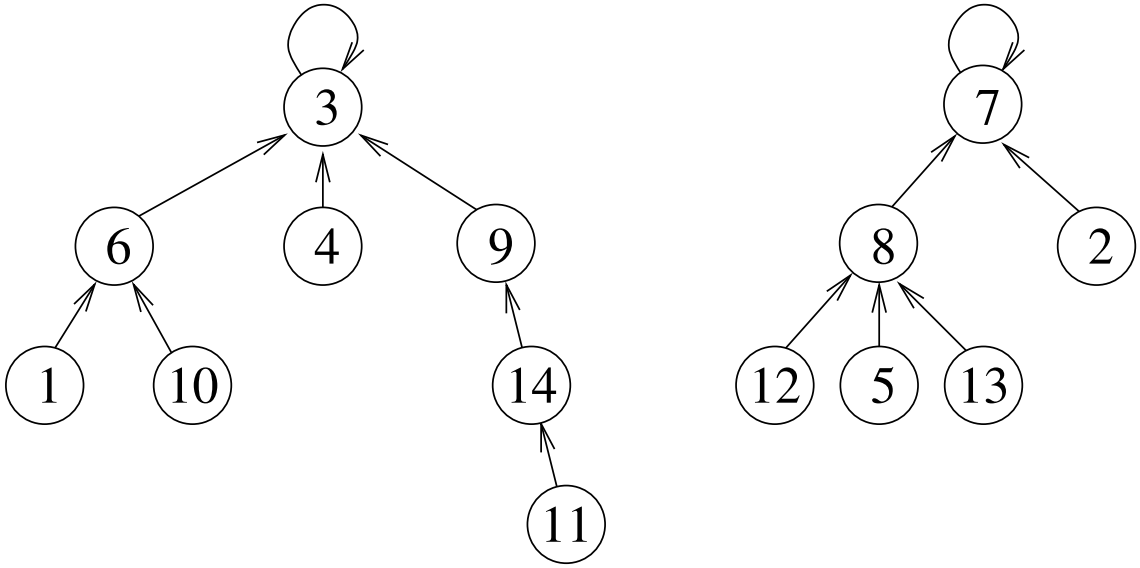


joukko {1, 3, 4, 6, 9, 10, 11, 14} edustaja 3



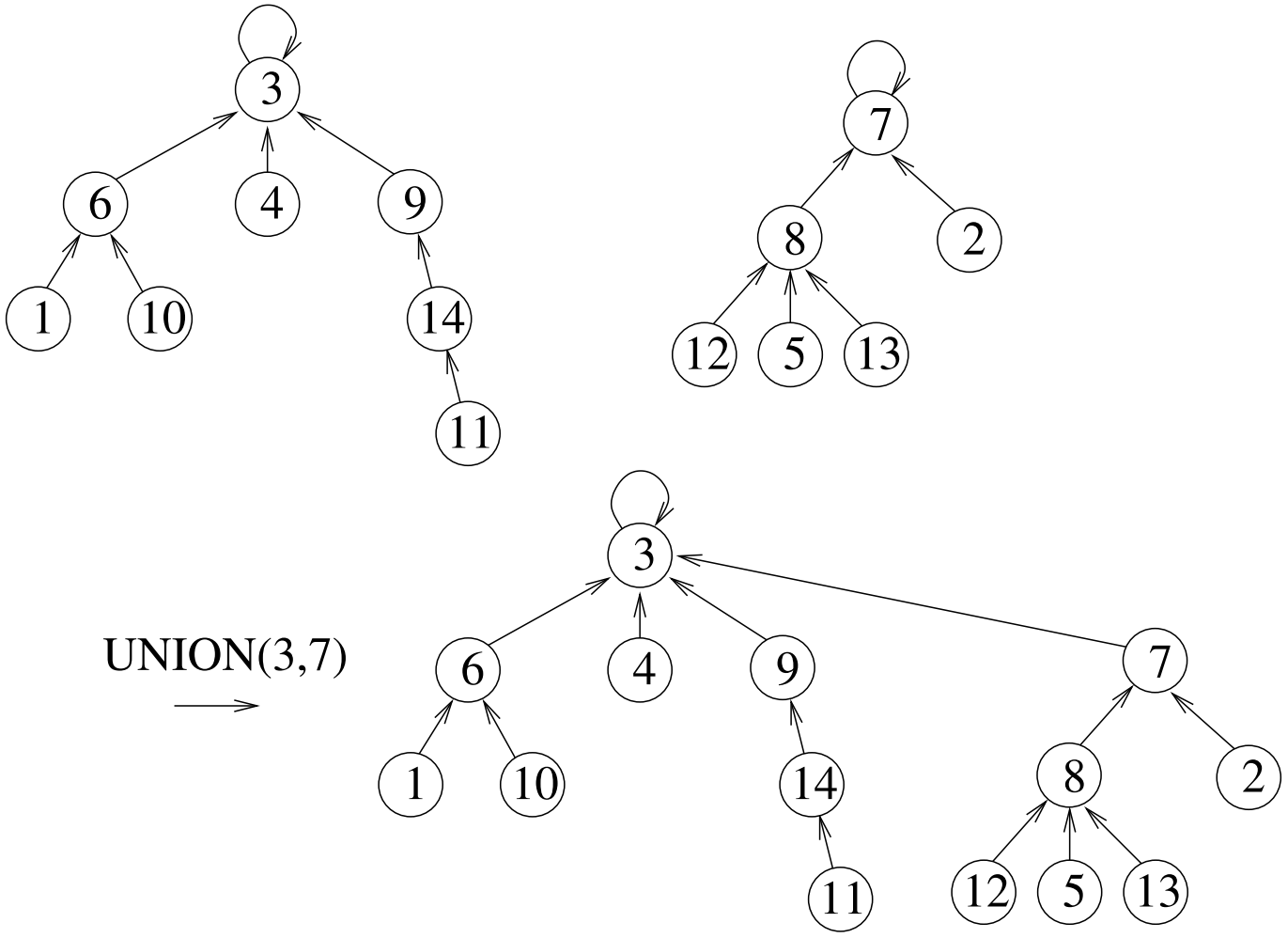
joukko {2, 5, 7, 8, 12, 13} edustaja 7

- Yksinkertaiset operaatiot: **find**
  - Operaatio **find**(x) lähtee solmusta x ja kulkee puussa ylöspäin nykyisen solmun vanhempaan niin kauan, kunnes tullaan juurisolmuun
  - Solmu x löydetään suoraan, sillä joukkoja kuvaavat puut on yleensä toteutettu taulukon avulla, missä kunkin indeksin arvona on tätä indeksiä vastaavan solmun vanhempi

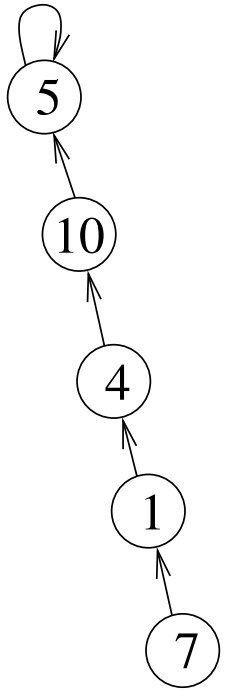


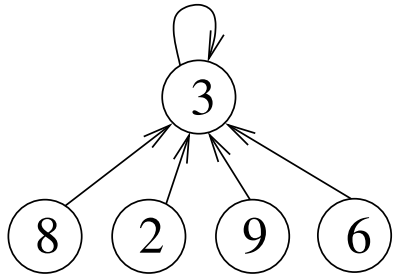
1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	7	3	3	8	3	7	7	3	6	14	8	8	9

- Yksinkertaiset operaatiot: **union**
  - Yhdistettävistä puista toisen juuri tulee toisen juuren lapseksi



- Puun pitäminen matalana
  - Operaatio **find**(x) kulkee koko matkan solmusta x joukkoa kuvaavan puun juureen saakka
  - **find**-operaation aikavaativuus on $\mathcal{O}(h)$, missä h on joukkoa kuvaavan puun korkeus
  - Pahimmassa tapauksessa puun jokaisella alkiolla on vain yksi lapsi, parhaassa tapauksessa taas kaikki puun muut solmut ovat juuren lapsia





- Jotta **find**-operaatiot pysyisivät tehokkaana, pyritään siihen, että joukkoja kuvaavat puut pysyisivät mahdollisimman matalina
- Tämä onnistuu muuttamalla **union**- ja **find**- operaatioita sopivasti
- Tapoja on useita, seuraavassa esitetään tapa, jolla pieni muutos **union**-operaatioon takaa joukkoa esittävien puiden korkeuden logaritmisuuden
- **Union**-operaatio suoritetaan aina siten, että matalamman puun juuri asetetaan korkeamman puun juuren lapseksi
- Näin yhdistetyn puun korkeus on sama kuin korkeamman puun ennen yhdistämistä, tai yksi korkeampi, jos alkuperäiset puut ovat yhtä korkeita
- Jotta **union**-operaatiota tehdessä voitaisiin päätellä, kumpi puu on korkeampi, on jokaisen puun juurisolmulla r attribuutti $r.korkeus$ joka on sen puun korkeus, jonka juurisolmu on r

- Induktiolla voidaan todistaa, että jos joukkoa kuvaavassa puussa on n solmua ja **union**-operaatiot on toteutettu edellisillä sivuilla kuvatulla tavalla, niin puun korkeus on $\mathcal{O}(\log n)$

- Näin m union-find -operaatiota voidaan suorittaa ajassa $\mathcal{O}(m \log n)$

- Tietorakennetta voidaan vielä tehostaa suorittamalla **find**-operaation yhteydessä poluntiivistys: Kun on löydetty polku puun juureen, niin oikaistaan kaikki osoittimet osoittamaan suoraan juureen

- Tällöin seuraavat **find**-operaatiot nopeutuvat

- Huomaa, että korkeus-arvoja ei päivitetä, joten ne menettävät tarkan vastaavuutensa polunpituuksiin

- Seuraavalla sivulla on operaatioiden pseudokoodiesitys

```
make-set(G,x)
1   x.p = x
2   x.korkeus = 0

union(x,y)
1   if x.korkeus < y.korkeus
2       x.p = y
3   elsif x.korkeus > y.korkeus
4       y.p = x
5   else
6       x.p = y
7       y.korkeus = x.korkeus + 1

find(x)
1   z = x
2   while z.p ≠ z
3       z = z.p
4   y = z
5   z = x
6   while z.p ≠ z
7       r = z.p
8       z.p = y
9       z = r
10  return y
```

- Tasapainotusta ja poluntiivistystä käytettäessä voidaan osoittaa, että m union-find -operaatiota n alkion perusjoukossa vie vain ajan $\mathcal{O}(m\alpha(n))$, missä α on erittäin hitaasti kasvava funktion: karkeasti arvioiden $\alpha(n) \leq 4$, kun $n \leq 10^{80}$ (ks. Cormen luku 21.4.)

- Voidaan myös todistaa hiukan löysempi yläraja $\mathcal{O}(m \log_2^* n)$, missä $\log^* n$ määritellään seuraavasti:

$$\begin{aligned} \log^{(0)} n &= n \\ \log^{(i)} n &= \log(\log^{(i-1)} n), \text{ kun } \log^{(i-1)} n > 0 \\ \log^* n &= \min\{\, i \geq 0 \mid \log^{(i)} n \leq 1\}. \end{aligned}$$

- Esimerkiksi

$$\log_2^* 2^{65536} = \log_2^* 2^{2^{2^2}} = 5,$$

joten $\log_2^* n \leq 5$, kun $n \leq 2 \cdot 10^{19728}$

Union-find-rakenteen avulla toteutetun Kruskalin algoritmin aikavaativuus

- **Kruskalin algoritmissa** tarvittavat $\mathcal{O}(|E|)$ **find**-operaatiota voidaan suorittaa ajassa $\mathcal{O}(|E|\alpha(|V|))$
- **Union**- ja **make-set**- operaatiot voidaan suorittaa vakioajassa. **Kruskalin algoritmissa** molempia tarvitaan $\mathcal{O}(|V|)$-kappaletta
- Edellisillä sivuilla kuvatulla tavalla toteutettua union-find -rakennetta käyttävän **Kruskalin algoritmin** aikavaativuus on kaarten järjestämiseen käytettävä $\mathcal{O}(|E|\log|E|)$ + rivin 5-9 while-silmukkaan kuluva $\mathcal{O}(|E|\alpha(|V|))$ eli yhteensä $\mathcal{O}(|E|(\log|E| + \alpha(|V|)))$
- Koska yhtenäisessä verkossa solmuja ei voi olla kuin korkeintaan yksi enemmän kuin kaaria, sievenee aikavaativuus muotoon $\mathcal{O}(|E|\log|E|)$
- Kuitenkin $|E| \leq |V|^2$, joten

$$|E|\log|E| \leq |E|\log(|V|^2) = 2 \cdot |E|\log|V| = \mathcal{O}(|E|\log|V|)$$

- Aikavaativuus voidaan siis kirjoittaa muotoon $\mathcal{O}(|E|\log|V|)$

- Usein virittävä puu tulee valmiiksi, ennen kuin kaikki kaaret on käsitelty
- Tällaisissa tapauksissa algoritmia voidaan jonkin verran tehostaa käyttämällä kekoa, sen sijaan että heti järjestetään kaaret
- Pahimman tapauksen aikavaativuus on kuitenkin sama kuin ennen

Esimerkki virittävien puiden sovelluksesta

- Muita sovelluksia löytyy harjoitustehtävistä
- Halutaan osittaa verkon $G = (V, E)$ solmut kahteen luokkaan V_1 ja V_2 siten, että luokkien välinen pienin etäisyys

  $$d(V_1, V_2) = \min \{ w(u, v) \mid u \in V_1, v \in V_2 \}$$

  on mahdollisimman suuri
- (Muistetaan, että osituksessa $V_1 \cap V_2 = \emptyset$ ja $V_1 \cup V_2 = V$)
- Intuitiivinen tulkinta on, että
  - $w(u, v)$ on jokin mitta alkioiden u ja v erilaisuudelle
  - alkiot halutaan jakaa kahteen mahdollisimman selvästi toisistaan erottuvaan luokkaan

- Annettu ongelma on erikoistapaus ryvästämisestä (engl. clustering), joka on tärkeä menetelmä data-analyysissä

- Halutaan osittaa perusjoukko X rypäisiin (engl. clusters) $X_1, \ldots, X_k$ siten, että
  - eri rypääseen kuuluvilla u ja v erilaisuus $d(u, v)$ on mahdollisimman suuri ja
  - samaan rypääseen kuuluvilla u ja v erilaisuus $d(u, v)$ on mahdollisimman pieni

- Tavoite voidaan täsmentää eri tavoin, mutta yleensä ongelma on laskennallisesti hankala

- Tässä tarkasteltu erikoistapaus ratkeaa kuitenkin helposti muodostamalla pienin virittävä puu

- Väitämme, että ongelma ratkeaa seuraavalla algoritmilla:
  1. Muodosta verkolle pienin virittävä puu (V, T)
  2. Olkoon e joukon T painoltaan suurin kaari
  3. Valitse luokiksi V_1 ja V_2 metsän $(V, T - \{ e \})$ yhtenäiset komponentit

- Tulos on sama, kuin jos ajettaisiin **Kruskalin algoritmia**, mutta lopetettaisiin juuri ennen viimeisen kaaren lisäämistä virittävään puuhun

- Todetaan ensin, että muodostettujen luokkien V_1 ja V_2 pienin etäisyys on

$$\min\{\, w(u,v) \mid u \in V_1, v \in V_2 \,\} = w(e)$$

- Valitaan mitkä tahansa $u \in V_1$ ja $v \in V_2$
- Nyt $T' = (T - \{\, e \,\}) \cup \{\, (u,v) \,\}$ on virittävä puu, jonka paino on

$$w(T') = w(T) - w(e) + w(u,v)$$

- Koska T on pienin virittävä puu, on oltava $w(u,v) \geq w(e)$
- Siis e on luokkia V_1 ja V_2 yhdistävistä kaarista painoltaan pienin
- Todetaan sitten, että millä tahansa solmujen osituksella (U_1, U_2) pätee

$$\min\{\, w(u,v) \mid u \in U_1, v \in U_2 \,\} \leq w(e)$$

- Tämä seuraa suoraan siitä, että ainakin yksi puun T kaari yhdistää joukkoja U_1 ja U_2, ja e on puun T kaarista painoltaan suurin
- Edellisistä kahdesta toteamuksesta seuraa, että algoritmin tuottama ositus (V_1, V_2) maksimoi lyhimmän etäisyyden

$$\min\{\, w(u,v) \mid u \in V_1, v \in V_2 \,\}$$

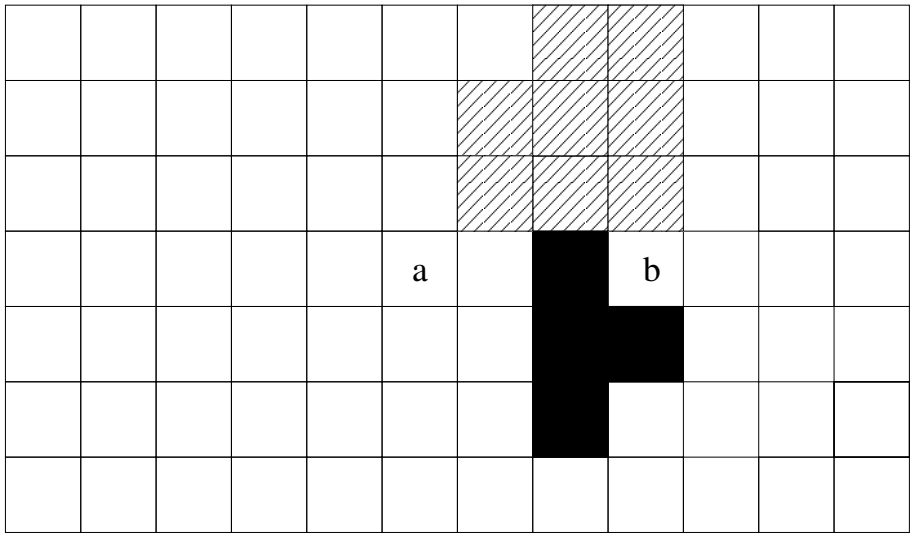
□

Kruskal vai Prim

- Voidaan osoittaa, että molemmat algoritmit toimivat negatiivisilla kaarenpainoilla
- Molempien algoritmien aikavaativuudet ovat $\mathcal{O}(|E| \log |V|)$
- Käytännössä **Prim** on yleensä parempi:
  - Avaimen arvon pienennystä tarvitaan yleensä vain pienelle osalle kaarista
  - **Kruskalin algoritmissa** järjestämisen keskimääräinen vaatimus ei ole pahinta tapausta parempi
  - Tiheissä verkoissa **Primin** aikavaativuus on $\mathcal{O}(|V|^2)$
- **Kruskal** tulee kuitenkin **Primiä** nopeammaksi, jos kaaret saadaankin valmiiksi järjestyksessä tai kaaret voidaan järjestää yleistä alarajaa nopeammin

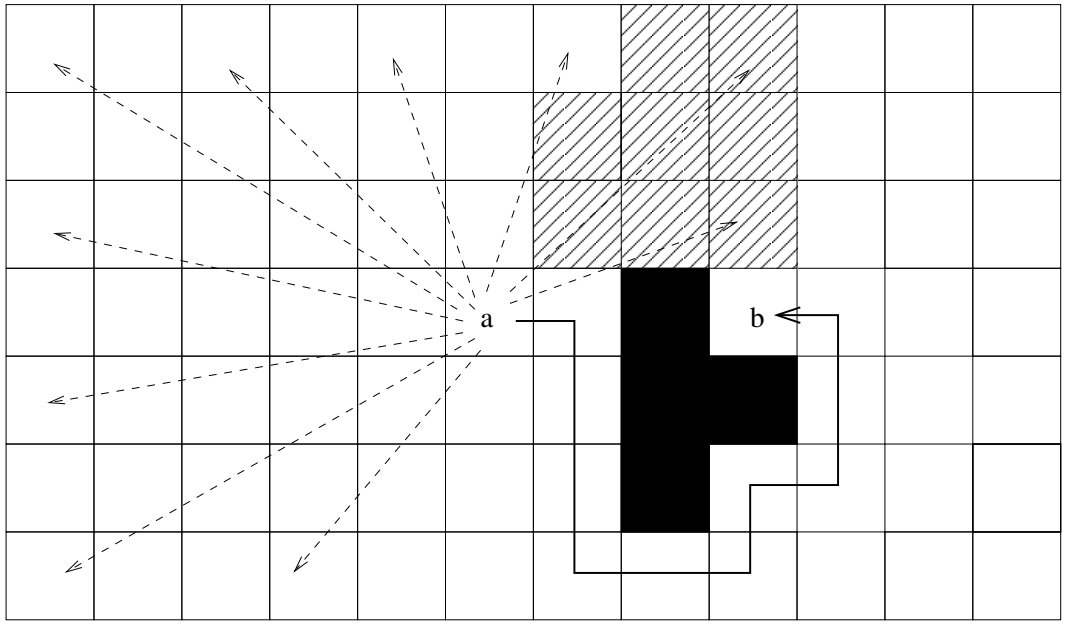
Lyhin kahden solmun välinen polku

- Haluamme etsiä lyhimmän polun alla olevan ruudukon kohdasta a kohtaan b
  - vierekkäisten (toistensa sivuilla, ylä- ja alapuolella olevien) valkoisten ruutujen välinen etäisyys on 1
  - mustien ruutujen läpi ei voi kulkea
  - raidalliset ruudut ovat vaikeakulkuista maastoa: niiden läpi voi kulkea, mutta ”etäisyys” raidalliseen ruutuun on 5



- Tulkitaan ruudukko verkkona siten, että ruudut ovat solmuja ja vierekkäisiä ruutuja vastaavien solmujen välillä on kaari

- Lyhin polku voidaan selvittää **Dijkstran algoritmilla** (ks. sivut 524–538)
- Lyhin polku, jonka pituus on 11 on merkitty kuvaan



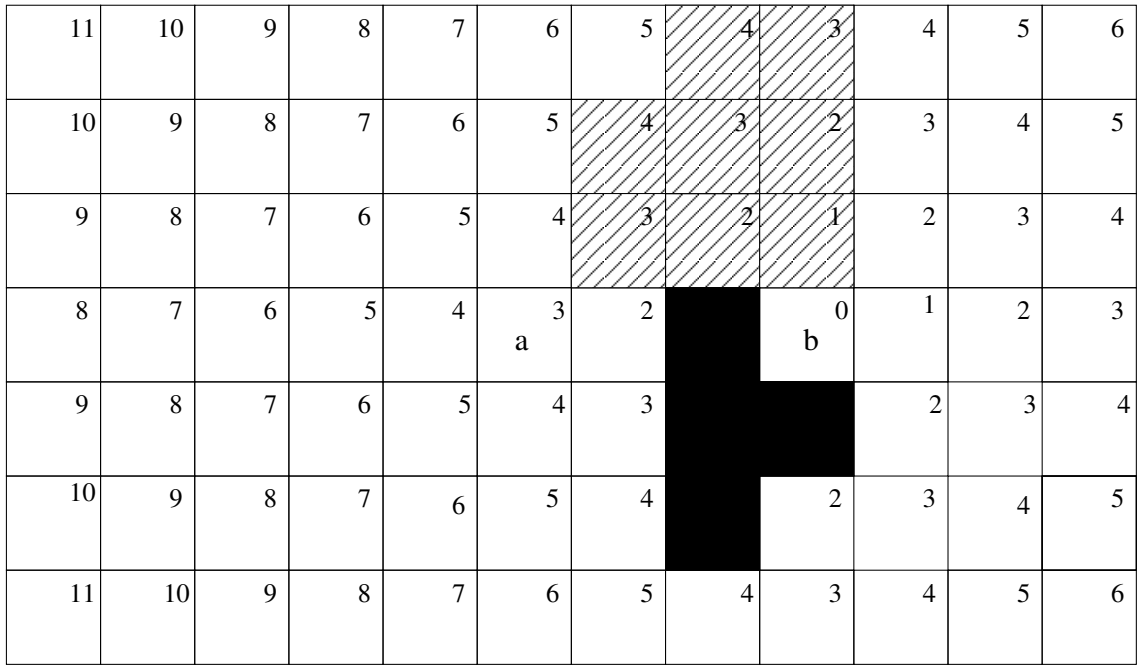
- **Dijkstran algoritmin** huono puoli on, että se ei huomioi millään tavalla, että kohdesolmu b on lähtöpaikan oikealla puolella
- **Dijkstra** etenee tasaisesti kaikkiin suuntiin ja tutkii samalla myös lyhimmät polut muihinkin solmuihin, myös täysin väärässä suunnassa oleviin
- Lyhimmän polun $a \rightsquigarrow b$ löytymisen kannalta **Dijkstra** siis tekee yleensä turhaa työtä

- **Dijkstran algoritmi** siis toimii seuraavasti:
  - ylläpidetään joukkoa S joka koostuu niistä solmuista joiden etäisyys lähtösolmuun s on selvitetty
  - jokaiselle solmulle v pidetään yllä etäisyysarviota $distance[v]$, joka kertoo lyhimmän tunnetun polun $s \rightsquigarrow v$ pituuden
  - alussa $distance[s] = 0$ ja muille solmuille $distance[v] = \infty$
  - aluksi S on tyhjä
  - algoritmi käsittelee solmuja yksi kerrallaan lisäten joukkoon S aina solmun jonka etäisyysarvio on pienin
  - kun solmu v tulee käsitellyksi tarkastetaan sen vierussolmujen etäisyysarviot
- Algoritmi lopettaa kun kaikki verkon solmut on lisätty joukkoon S eli tunnetaan pienin polku lähtösolmusta s kaikkiin muihin solmuihin
- Jos ollaan kiinnostuneita ainoastaan kahden solmun a ja b välisestä lyhimmästä polusta, **Dijkstra** siis tutkii samalla myös b:stä poispäin johtavia polkuja
- "Ohjaamalla" algoritmin toimintaa siten, että se suosii kohdesolmuun b päin johtavia polkuja, on yleensä mahdollista löytää lyhin polku nopeammin kuin **Dijkstran algoritmilla**

A*-algoritmi

- **A*-algoritmin** voi ajatella **Dijkstran algoritmin** laajennuksena, joka koko ajan arvioi mikä tutkimattomista solmuista näyttää olevan osa lyhintä polkua solmujen a ja b välillä
- Tätä varten algoritmi arvioi etäisyyden jokaisesta solmusta maalisolmuun b
- Arvio voidaan tehdä monella tavalla
- Seuraavalla sivulla olevaan kuvaan on merkitty etäisyysarviot, joiden arvo on lyhin mahdollinen etäisyys solmujen välillä huomioimatta millään tavalla esteitä tai vaikeakulkuista reittiä
- Arvion tulee olla helposti laskettavissa
  - esimerkkitapauksessamme kohdesolmu b on ruudussa $(4, 9)$ neljä alas, 9 oikealle
  - etäisyysarvio ruudukon kohdassa (i, j) olevasta solmusta kohdesolmuun b on $|(i - 4) + (j - 9)|$
  - esim. lähtösolmu on paikassa $(4, 6)$, joten sen etäisyysarvio on $|(4 - 4) + (6 - 9)| = |-3| = 3$

- Etäisyysarviot jokaisesta solmusta kohdesolmuun b on merkitty ruudun oikeaan yläkulmaan



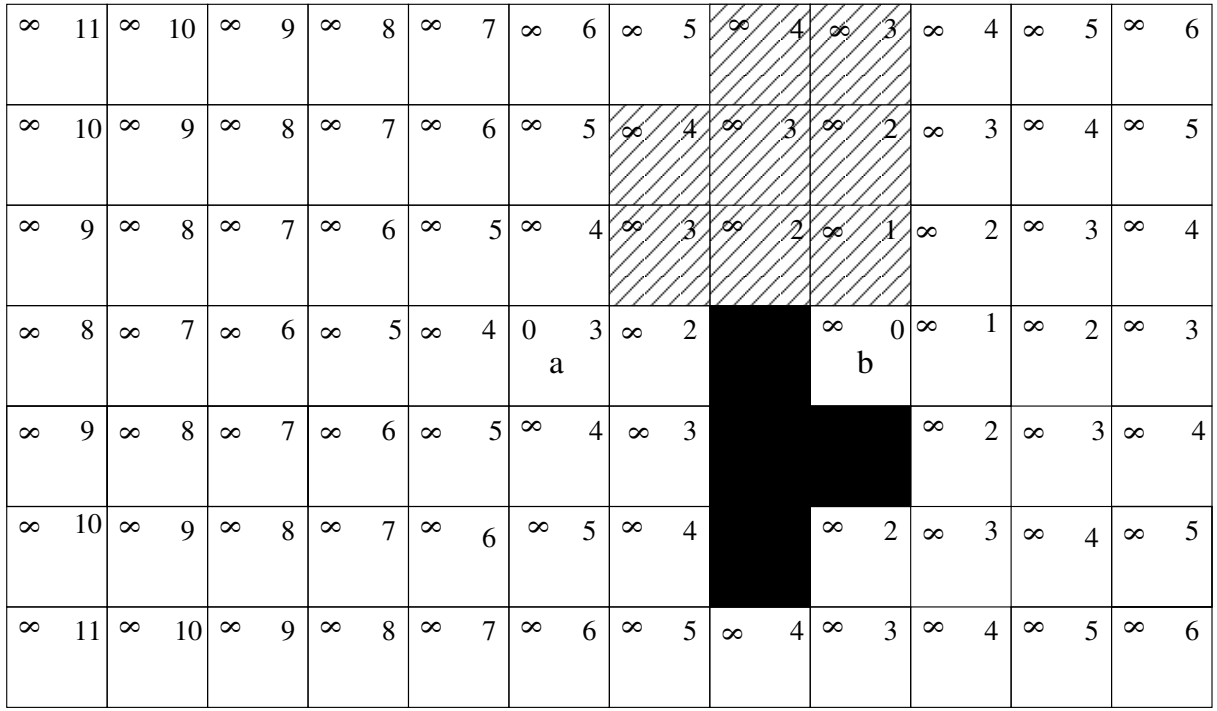
- Todellisuudessa arvioita ei tarvitse laskea kaikille solmuille etukäteen vaan riittää, että ne selvitetään tarpeen vaatiessa

- **Dijkstran algoritmin** tapaan jokaiselle solmulle ylläpidetään myös etäisyysarviota lähtösolmuun a

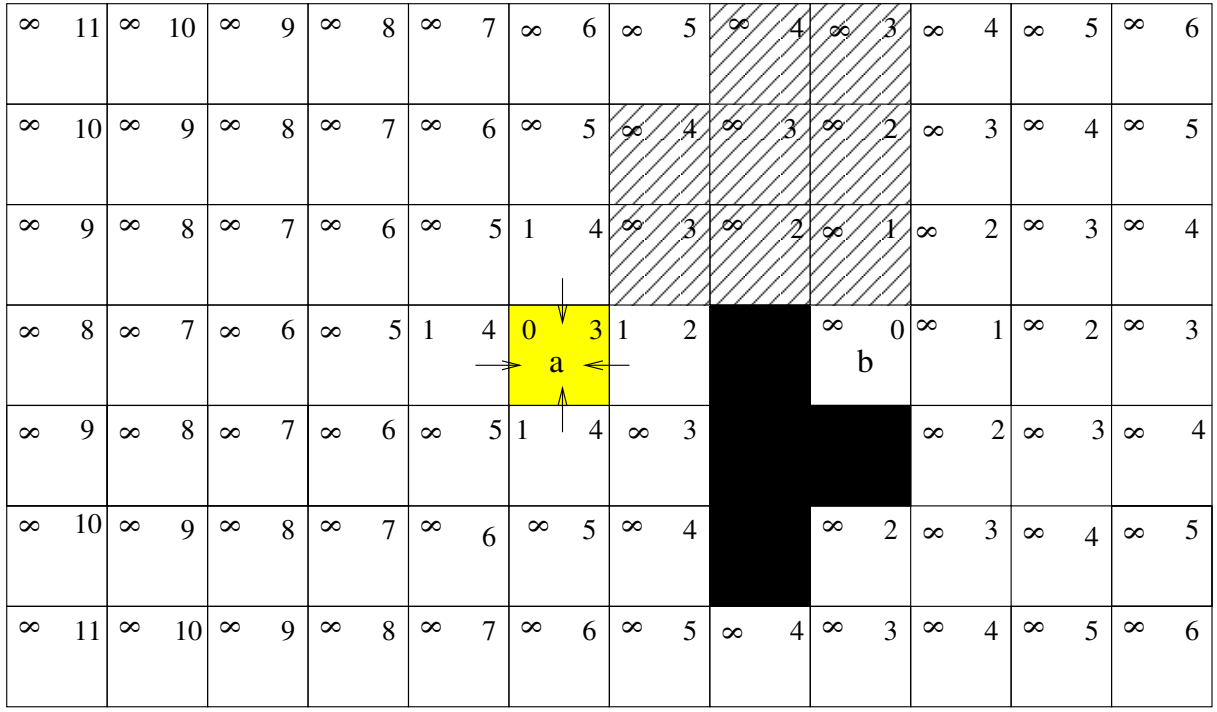
- **A*** siis ylläpitää jokaiselle solmulle v kahta tietoa
  - $alkuun[v]$: lähtösolmusta solmuun v johtavan polun $a \rightsquigarrow v$ etäisyysarvio (tähän mennessä tiedossa oleva lyhin etäisyys, kuten **Dijkstran algoritmissa**)
  - $loppuun[v]$: solmusta v maalisolmuun johtavan polun $v \rightsquigarrow b$ etäisyysarvio
- Algoritmi pitää kirjaa jo käsitellyistä solmuista joukon S avulla
  - alussa S on tyhjä
  - aluksi asetetaan kaikille solmuille v paitsi lähtösolmulle $alkuun[v] = \infty$ ja $alkuun[a] = 0$

    eli alussa etäisyysarvio lähtösolmusta muihin solmuihin on tuntematon
  - algoritmi käsittelee solmuja yksi kerrallaan lisäten joukkoon S aina solmun v jolle summa $alkuun(v) + loppuun(v)$ on pienin
  - kun solmu v tulee käsitellyksi, sen vierussolmujen etäisyysarviot lähtösolmuun päivitetään tarvittaessa (kuten **Dijkstran algoritmissa**)
- Algoritmi lopettaa kun se on käsitellyt kohdesolmun b
- **Dijkstran algoritmin** tapaan algoritmi muistaa kullekin solmulle mistä lyhin polku siihen saapui

- Lähtötilanteessa kaikkien paitsi lähtösolmun etäisyysarvio $alkuun$ on ääretön
- Alussa käsiteltyjen solmujen joukko S on tyhjä
- Joukkoon S lisätään aina solmu v jolle summa $alkuun(v) + loppuun(v)$ on pienin, eli alussa lisättäväksi valitaan lähtösolmu



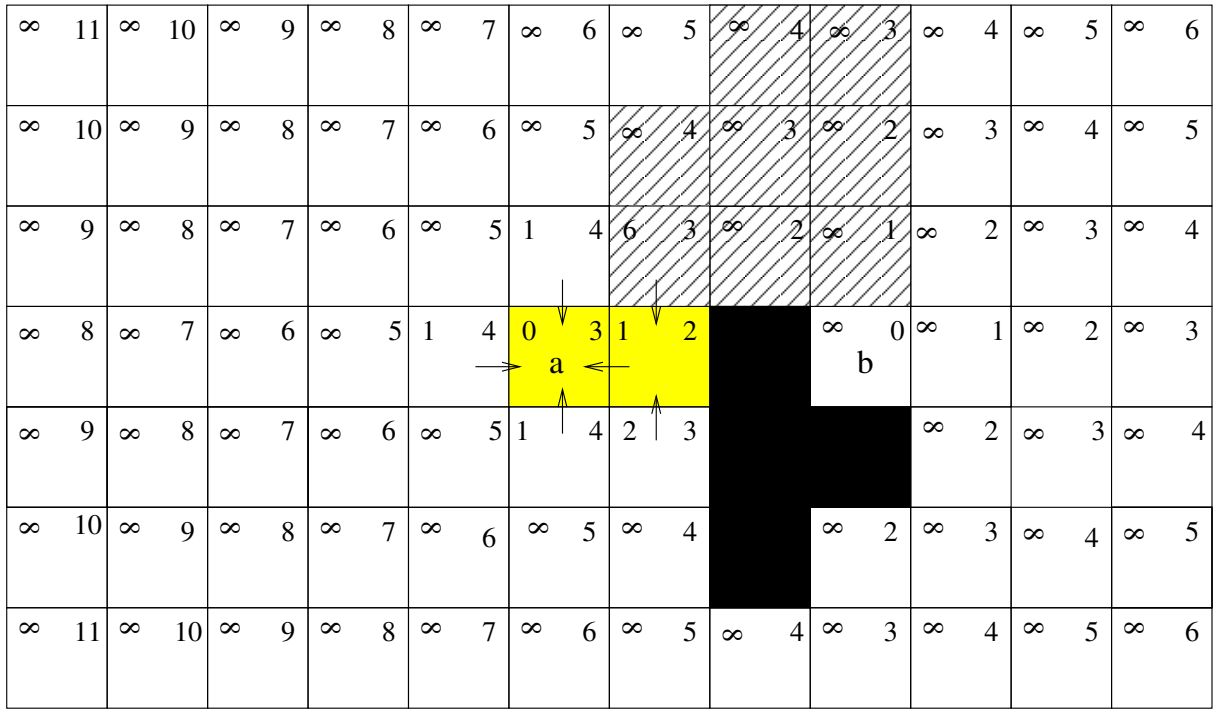
- Lähtösolmu on nyt käsitelty eli viety joukkoon S, kuvassa se on muutettu harmaaksi
- Lähtösolmun vierussolmujen v arvot $alkuun[v]$ on päivitetty
- Vierussolmuihin on myös merkitty että niihin saavuttiin alkusolmusta; tätä tietoa hyväksikäyttäen pystytään lopuksi generoimaan etsitty lyhin polku



- Jälleen käsittelyyn valitaan solmu v jolle summa $alkuun(v) + loppuun(v)$ on pienin, eli lähtösolmun oikealla puolella oleva solmu

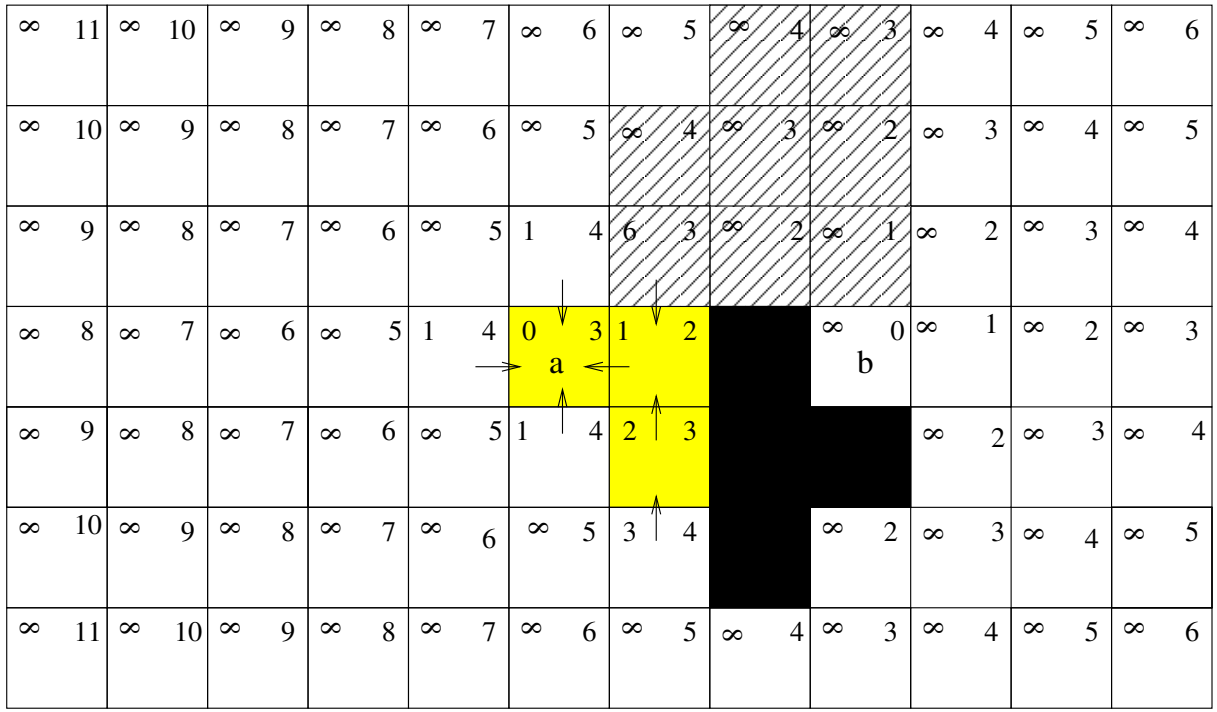
- Huom: valitun solmun yläpuolella on vaikeakulkuinen maasto, joten kaari yläpuolella olevaan solmuun on pituudeltaan 5

- Solmun käsittelyn jälkeen verkossa on 4 solmua joilla $alkuun(v) + loppuun(v) = 5$, eli algoritmi valitsee seuraavaksi käsittelyyn jonkun näistä



- Oletetaan, että algoritmi valitsee juuri käsitellyn solmun alapuolella olevan solmu seuraavaksi käsiteltäväksi

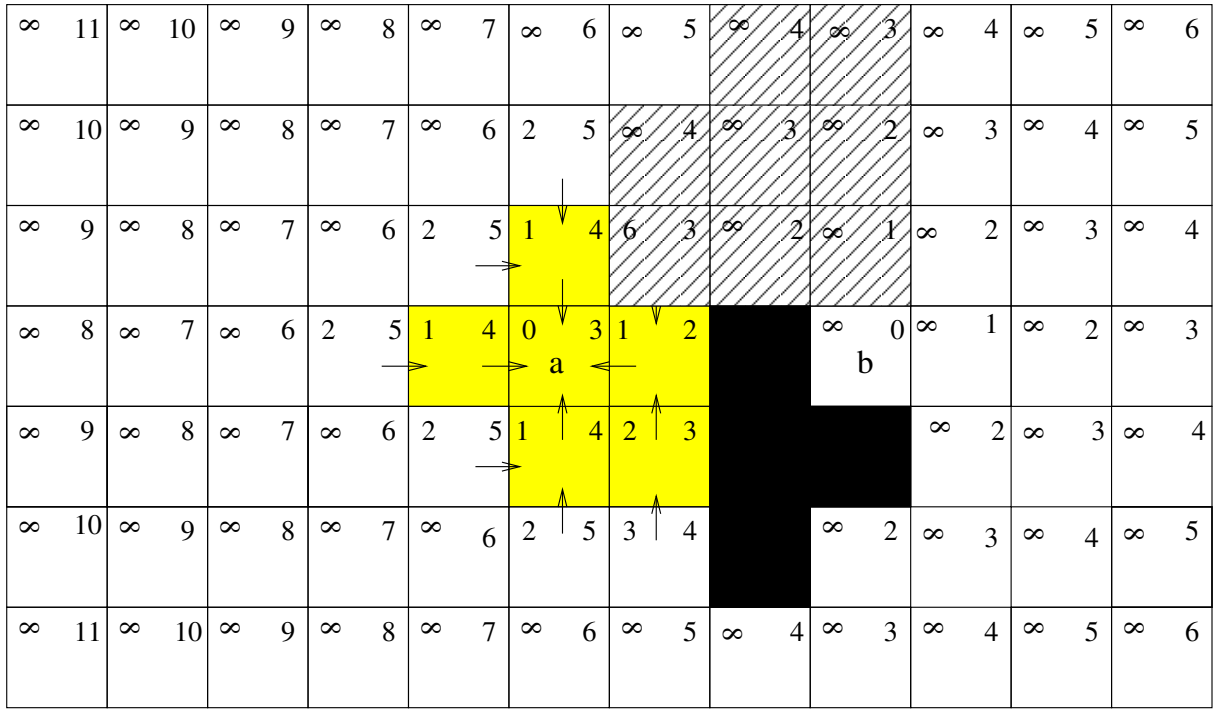
- Päivityksen jälkeen tilanne näyttää seuraavalta



- Jälleen käsittelyyn valitaan solmu v jolle summa $alkuun(v) + loppuun(v)$ on pienin

- Verkossa on edelleen 3 solmua joille $alkuun(v) + loppuun(v) = 5$, eli vaikka silmämääräisesti huomaamme, että ne ovat väärässä suunnassa, algoritmi tutkii ne ennen kuin lähtee oikeaan suuntaan

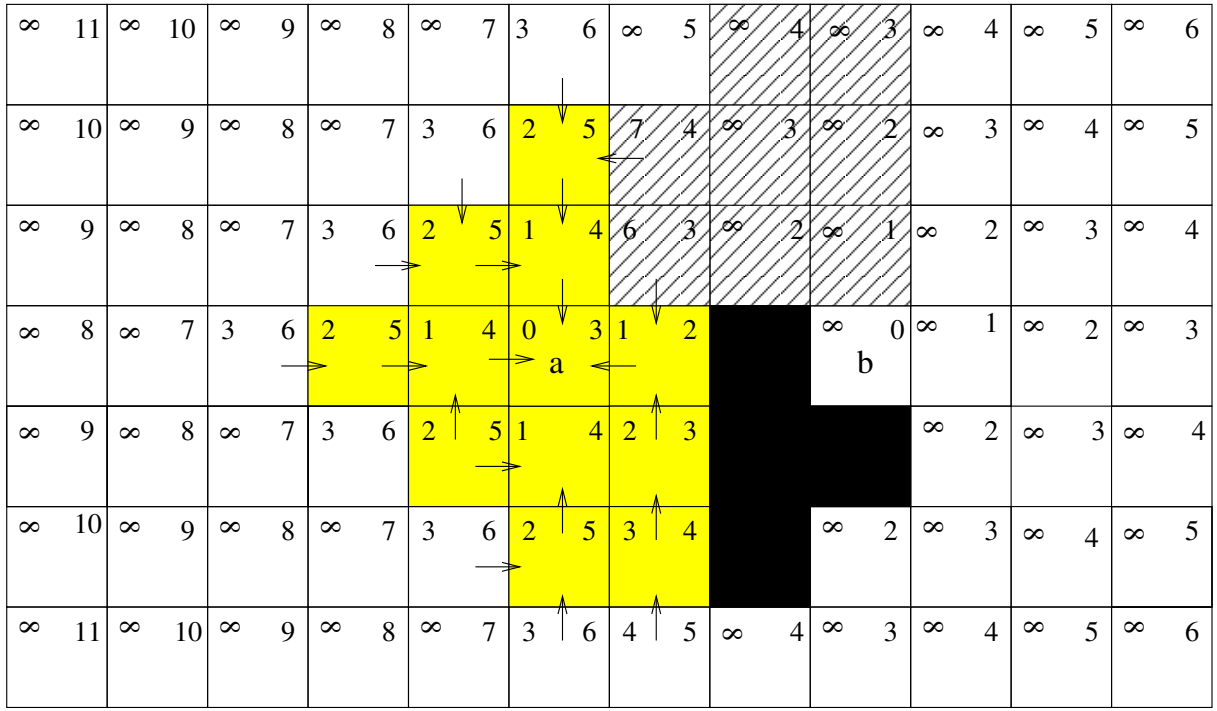
- Seuraavassa tilanne kolmen "väärässä suunnassa" olevan solmun tutkimisen jälkeen



- Verkossa on nyt 6 solmua joille $alkuun(v) + loppuun(v) = 7$

- Vaikka osa näistä on väärässä suunnassa, algoritmi tutkii ne seuraavissa askeleissa

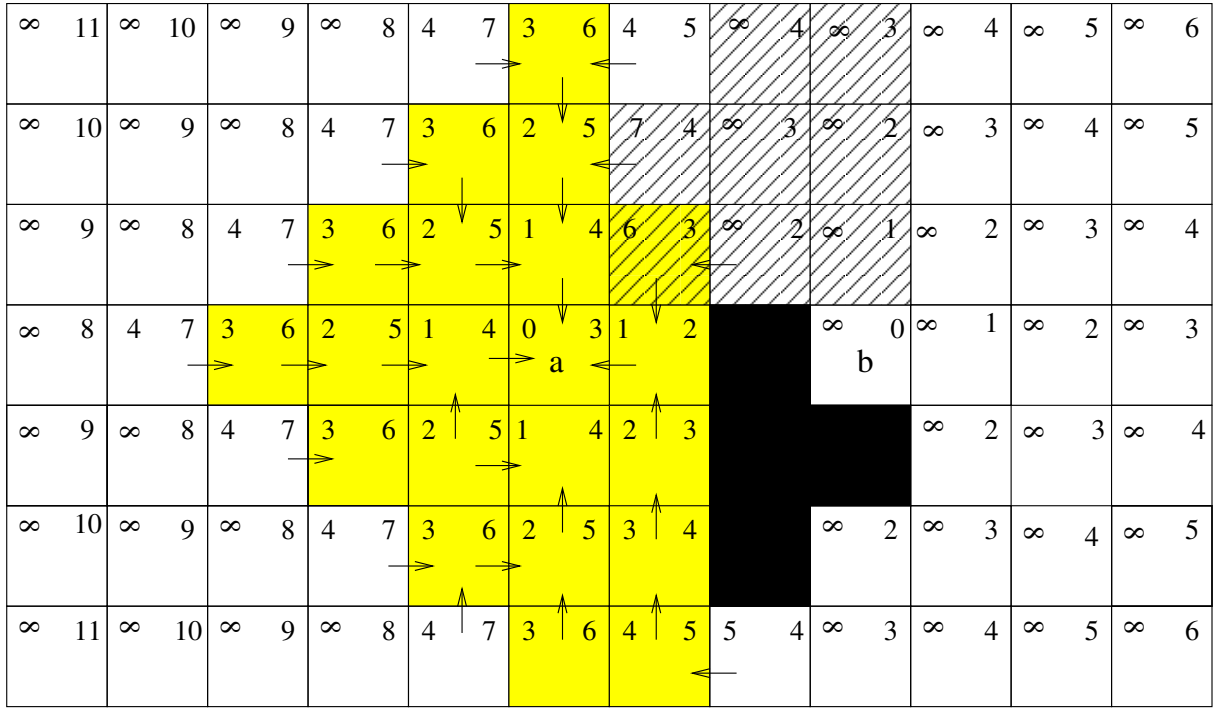
- Päädytään seuraavaan tilanteeseen



- Seuraavissa vaiheissa vuorossa olevat solmut, joille $alkuun(v) + loppuun(v) = 9$, näitä solmuja on 9

- Tutkitaan nämä solmut

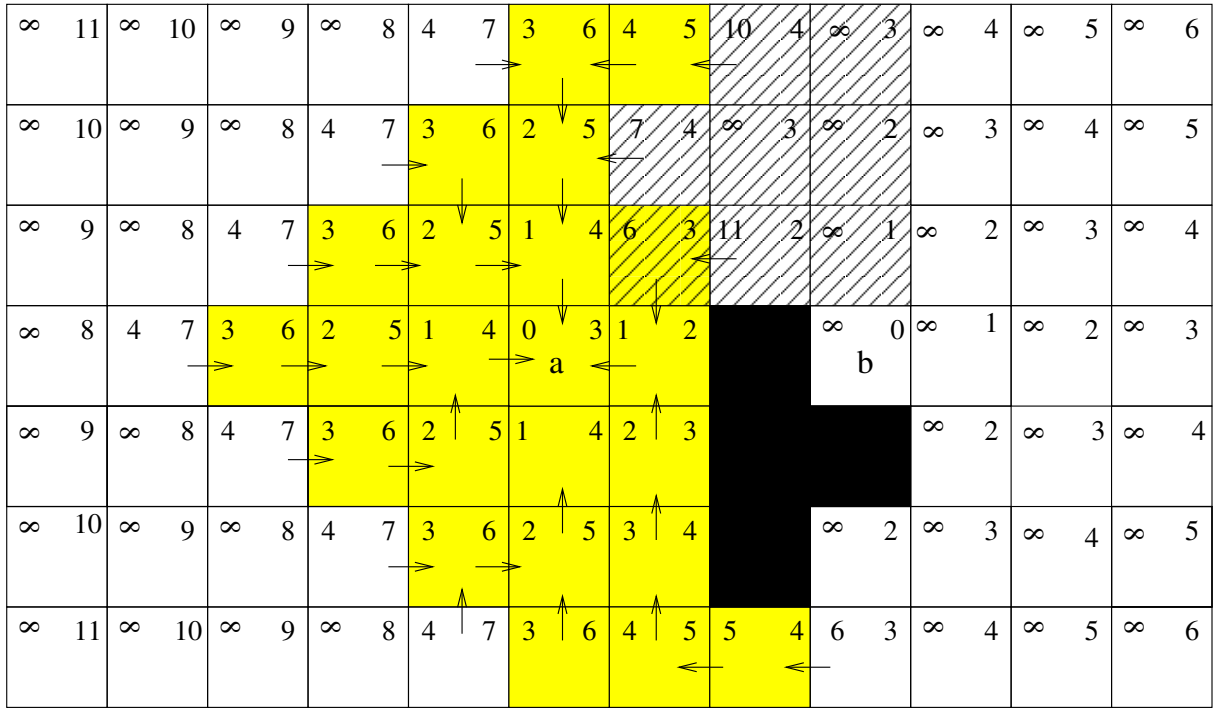
- Päädytään seuraavaan tilanteeseen



- Huomaamme, että algoritmi alkaa vihdoin kääntää etsintää oikeaan suuntaan sillä nyt "väärässä suunnassa" oleville solmuille $alkuun(v) + loppuun(v) = 11$

- Seuraavissa vaiheissa siis käsitellään ne 2 solmua, joille $alkuun(v) + loppuun(v) = 9$

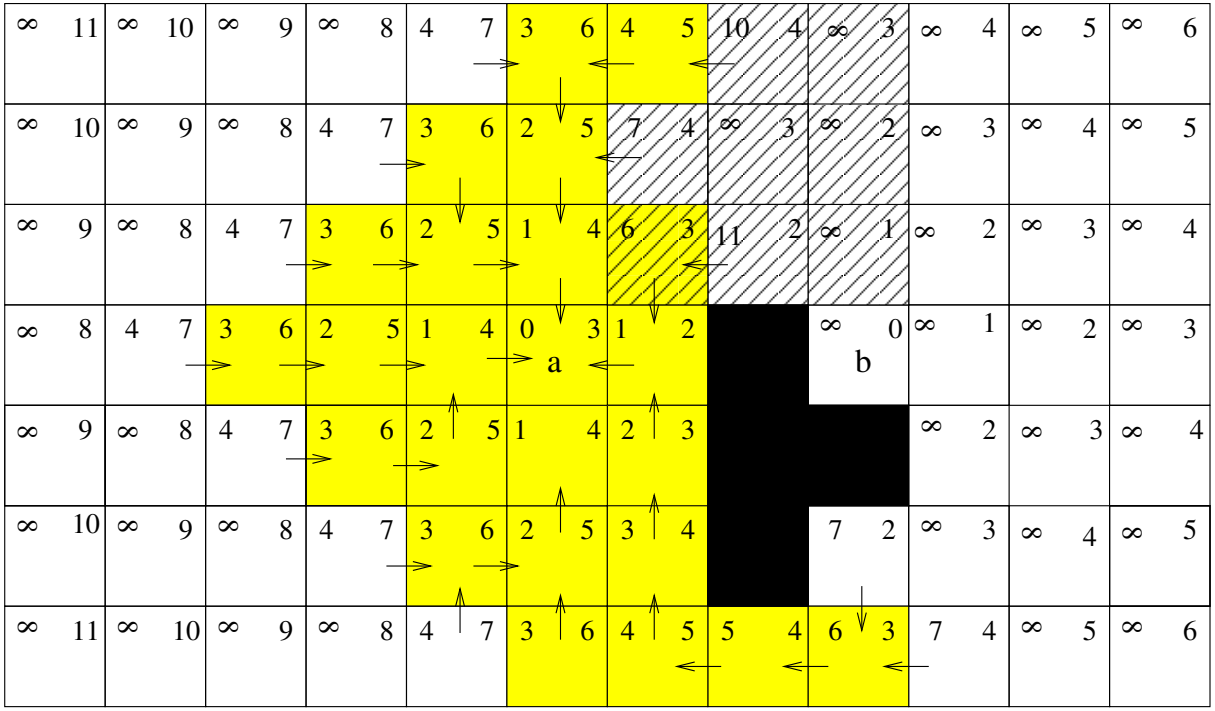
- Päädytään seuraavaan tilanteeseen



- Jälleen löytyy uusi solmu, jolle $alkuun(v) + loppuun(v) = 9$

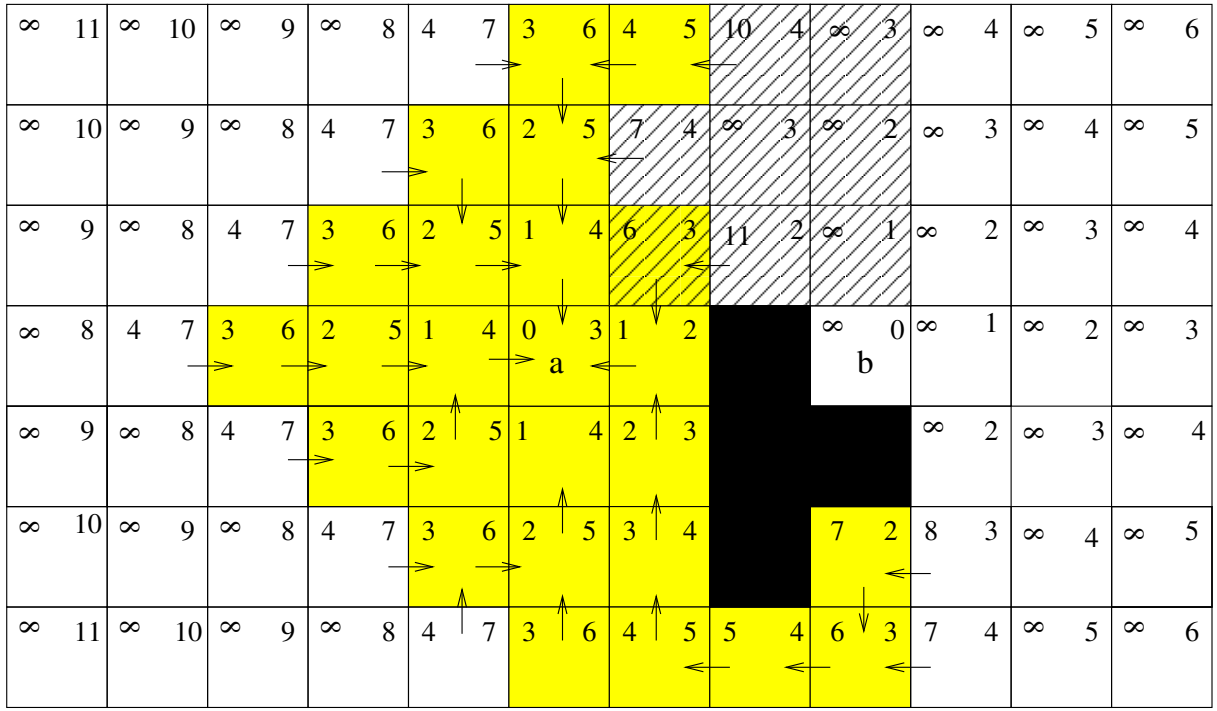
- Valituksi tuleva solmu on askeleen lähempänä maalisolmua, joten algoritmi jatkaa oikeaan suunntaan

- Päädytään seuraavaan tilanteeseen



- Jälleen löytyy uusi solmu, jolle $alkuun(v) + loppuun(v) = 9$
- Algoritmi valitsee solmun käsittelyyn

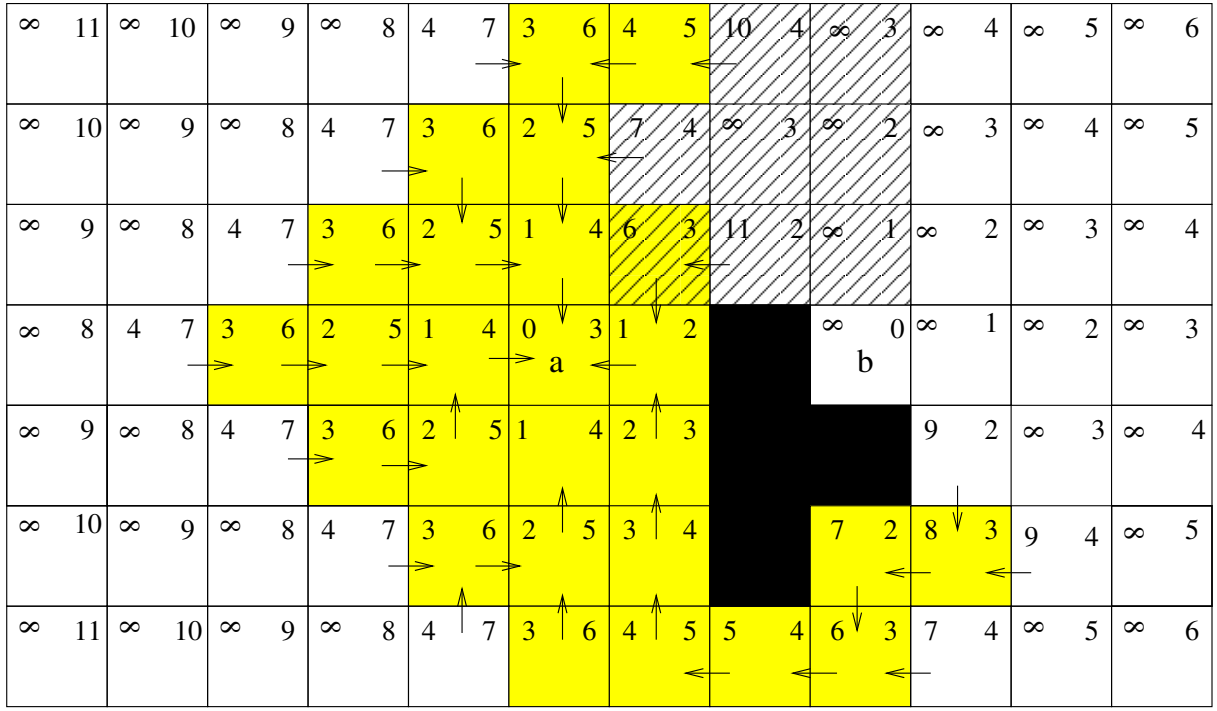
- Päädytään seuraavaan tilanteeseen



- Seuraavana käsittelyvuorossa yksi solmuista, jolle $alkuun(v) + loppuun(v) = 11$

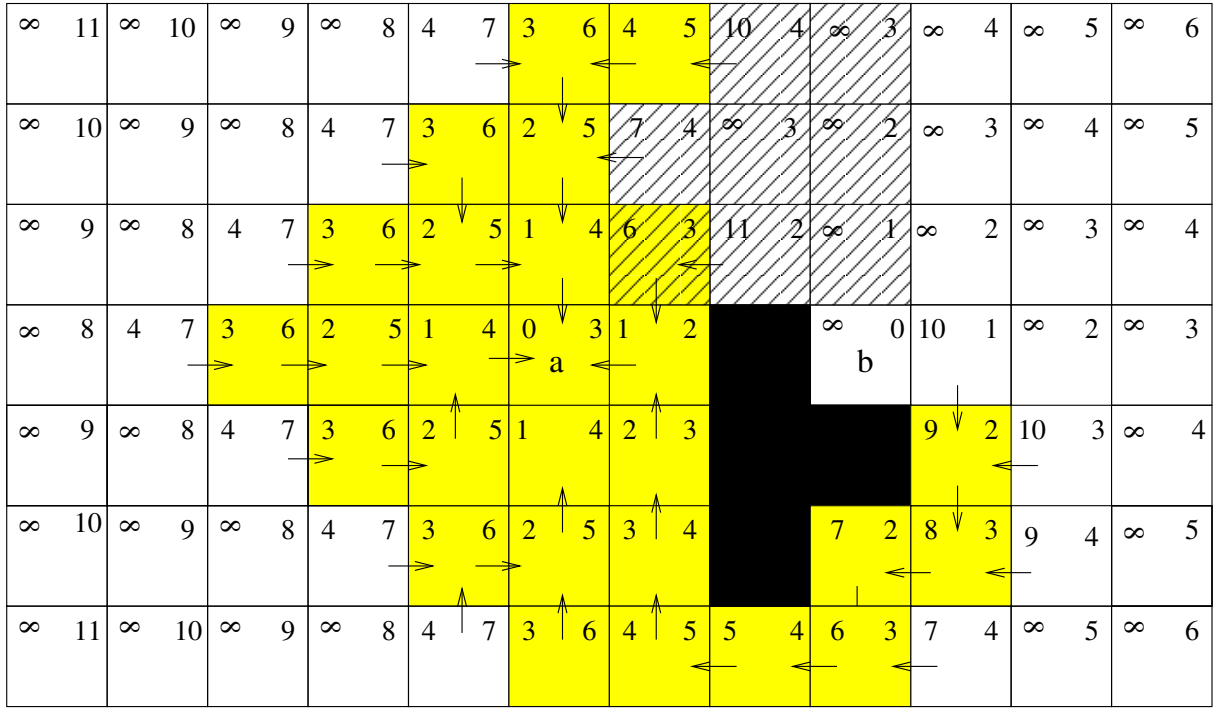
- Oletetaan, että algoritmi valitsee näistä sattumalta käsittelyyn lähempänä maalia olevan

- Päädytään seuraavaan tilanteeseen



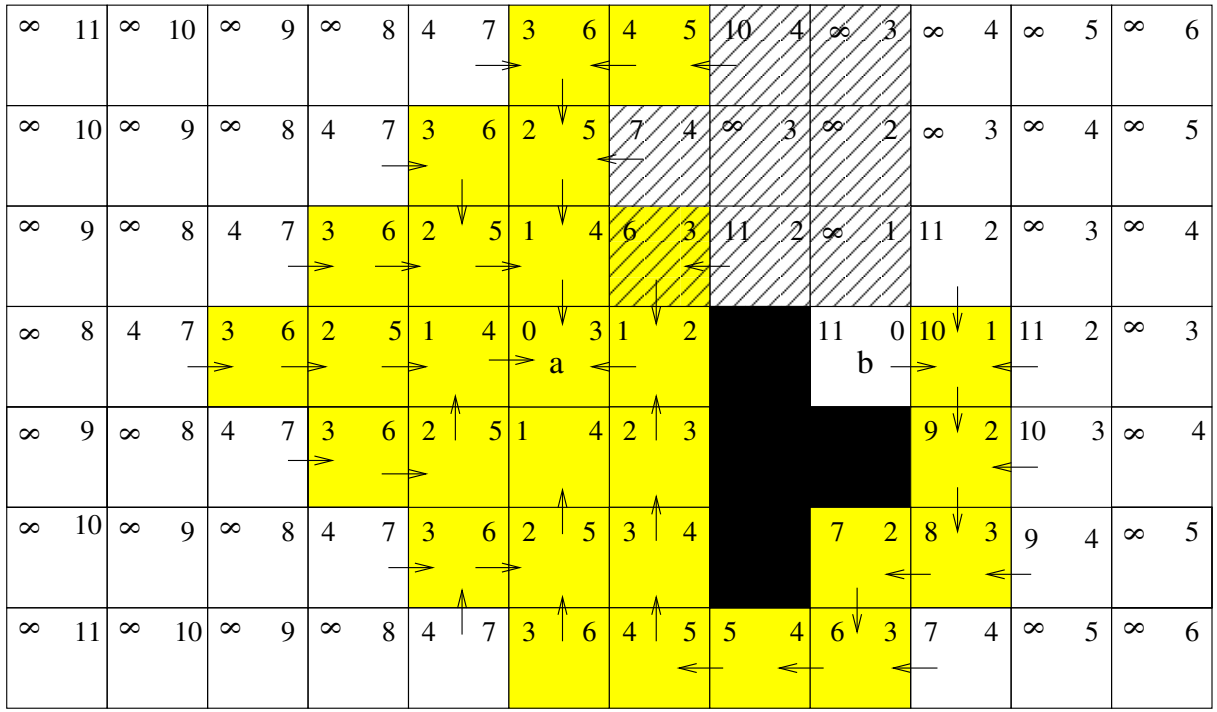
- Jälleen käsitteyvuorossa yksi solmuista, jolle $alkuun(v) + loppuun(v) = 11$
- Oletetaan, että algoritmi valitsee jälleen näistä maalia lähimpänä olevan

- Päädytään seuraavaan tilanteeseen



- Oletetaan taas, että algoritmi sattumalta valitsee käsittelyyn lähempänä maalisolmua olevan solmun

- Maalisolmu b löytyy vihdoin



- Kun maalisolmu tulee lisätyksi joukkoon S, algoritmi voi lopettaa ja lyhin polku on löytynyt

- Kuten huomaamme, algoritmi löytää lyhimmän polun huomattavasti nopeammin kuin **Dijkstran algoritmi**, jonka olisi täytynyt tutkia verkon kaikki polut joiden pituus on korkeintaan 11

- Seuraavalla sivulla algoritmi pseudokoodimuodossa

```
Astar(G,w,a,b)
// G tutkittava verkko, a lähtösolmu, b kohdesolmu ja w kaaripainot kertova funktio
1   for kaikille solmuille v ∈ V
2       alkuun[v] = ∞
3       loppuun[v] = arvioi suora etäisyys v ⇝ b
4       polku[v] = NIL
5   alkuun[a] = 0
6   S = ∅
7   while ( solmu b ei ole vielä joukossa S )
8       valitse solmu u ∈ V \ S, jolle alkuun[v]+loppuun[v] on pienin
9       S = S ∪ {u}
10      for jokaiselle solmulle v ∈ vierus[u]     // kaikille u:n vierussolmuille v
11          if alkuun[v] > alkuun[u] + w(u,v)
12              alkuun[v] = alkuun[u] + w(u,v)
13              polku[v] = u
```

- Lyhin polku muodostuu nyt taulukkoon $polku$, ja se on tulostettavissa samaan tapaan kuin **Dijkstran algoritmin** yhteydessä

- Jos oletetaan, että etäisyysarvio $loppuun[v]$ on laskettavissa vakioajassa, algoritmin pahimman tapauksen aikavaativuus on sama kuin **Dijkstran algoritmilla** eli $\mathcal{O}((|E| + |V|) \log |V|)$, jos toteutuksessa käytetään minimikekoa joukossa $V \setminus S$ olevien solmujen tallettamiseen

- Etäisyysarvio solmusta v maalisolmuun, eli arvo $loppuun(v)$, voidaan laskea monilla eri tavoilla sovelluksen mukaan
- Esimerkissämme oli käytössä ns. Manhattan-etäisyys, eli oletettiin että eteneminen voi tapahtua vain ylös, alas ja sivuille
- Muita mahdollisuuksia esim.
  - diagonaalinen etäisyys joka sallii myös siirtymisen "väli-ilmansuuntiin"
  - euklidinen etäisyys eli viivasuora etäisyys
- Etäisyysarvio voi olla erilainen eri osissa verkkoa
- Jos polun loppuosan etäisyysarvioksi määritellään kaikille solmuille $loppuun(v) = 0$ toimii **A*** täsmälleen kuten **Dijkstran algoritmi**!

- Terminologiasta:
  - kirjallisuudessa etäisyysarviota $loppuun(v)$ kutsutaan heuristiikkafunktioksi ja siitä merkitään usein $h(v)$
  - alkumatkan etäisyysarviosta $alkuun(v)$ taas käytetään usein merkintää $g(v)$
  - joukossa S olevia jo käsiteltyjä solmuja sanotaan suljetuiksi solmuiksi
  - jo löydettyjä, mutta ei vielä joukkoon S vietyjä solmuja sanotaan avoimiksi solmuiksi

- On huomioinarvoista, että algoritmi löytää varmasti lyhimmän polun vain jos heuristiikkafunktion arvo eli loppuosan etäisyysarvio ei ole millekään solmulle suurempi kuin solmun todellinen etäisyys maalisolmusta ja heuristiikkafunktio on monotoninen
  - heuristiikkafunktio on monotoninen, jos mille tahansa vierekkäisille solmuille u ja v pätee $h(u) \leq w(u, v) + h(v)$
  - jos etäisyysarvio liioittelee joidenkin solmujen etäisyyksiä, algoritmi löytää jonkun polun, mutta polku ei välttämättä ole lyhin
  - tälläiset polut löytyvät keskimäärin nopeammin kuin lyhimmät polut
  - algoritmia on siis mahdollisuus joissain tilanteissa nopeuttaa, jos polun ei tarvitse olla täysin optimaalinen pituuden suhteen

- **A*-algoritmi** on ylikurssia, eli ei kuulu koealueeseen
- **A*** on kuitenkin suosittu aihe Tietorakenteiden ja algoritmien harjoitustyössä
- Algoritmia käsitellään jonkin verran kurssilla Johdatus tekoälyyn

- Lisätietoa **A***:sta:
  - `www.policyalmanac.org/games/aStarTutorial.htm`
  - `theory.stanford.edu/~amitp/GameProgramming/`
  - Russell and Norvig: *Artifical Intelligence, a Modern Approach*

Vuo verkossa

- Eräs tärkeä sovellus verkoille on virtauksen eli *vuon* optimointi. Tämäkin on kuitenkin ylikurssia, eli tästäkään aiheesta ei tule tenttikysymyksiä

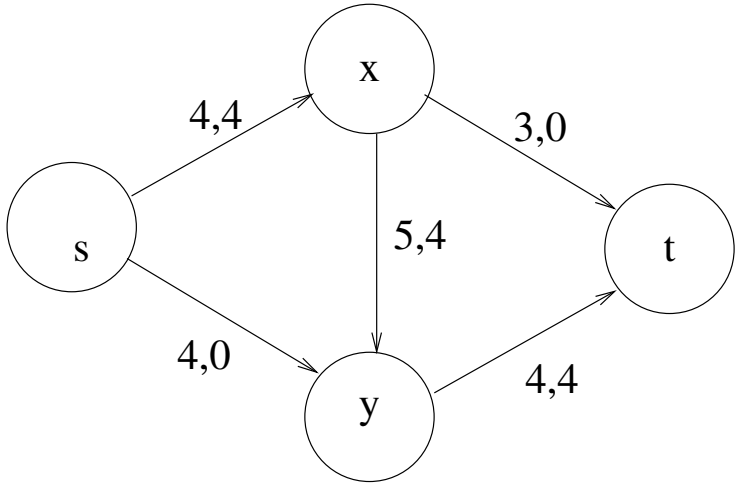
- Vuoverkko (engl. flow network) $G = (V, E)$ on suunnattu verkko, joiden kaarilla $(u, v) \in E$ on kapasiteetti (engl. capacity) $c(u, v) \geq 0$. Jos $(u, v) \notin E$, asetamme $c(u, v) = 0$. Verkossa on lähde eli alkusolmu (engl. source) s ja kohde eli loppusolmu (engl. sink) t

- Vuo (engl. flow) on sellainen funktio $f : V \times V \to R$, että

$$\begin{aligned} \text{kaikille } u, v \in V : f(u, v) \leq c(u, v) \quad &, \quad \text{eli vuo ei ylitä missään kapasiteettia,} \\ \text{kaikille } u, v \in V : f(u, v) = -f(v, u) \quad &, \quad \text{eli vuo kulkee yhteen suuntaan, ja} \\ \text{kaikille } u \in V - \{s, t\} : \sum_{v \in V} f(u, v) = 0 \quad &, \quad \text{eli vuota ei synny eikä tuhoudu} \\ & \quad \text{paitsi alku- ja loppusolmussa} \end{aligned}$$

- Vuon arvo on tällöin $\sum_{v \in V - \{s\}} f(s, v) = \sum_{v \in V - \{t\}} f(v, t)$

- Luonnollinen kysymys on nyt: mikä on vuoverkon maksimivuo?
- Esimerkkejä tällaisista ongelmista on jonkin aineen virtaus putkissa, tavarantoimitus eri toimenpisteiden välillä ja tiedonsiirtokapasiteetti verkon yli
- Tarkastelemme nyt yleisellä tasolla, miten tätä voidaan laskea. Ajatelkaamme, että meillä on verkko, jossa on jo jokin vuofunktio olemassa (triviaalisti nollavuo on vuo)
- Huomaamme, että jos löytyy s:stä t:hen sellainen polku, että sen jokaisella kaarella on kapasiteettia jäljellä, niin voisimme lisätä vuota tätä reittiä pitkin



- Tässä verkossa kaaren ensimmäinen numero on kapasiteetti ja toinen vuo. Huomaamme, että vuota ei voi lisätä, mutta että verkossa ei kulje maksimivuota

- Jos kuitenkin viemme vuon 3 kulkemaan polkua (s, y, x, t), jossa ajattelemme vuota y:stä x.ään negatiivisena vuona x:stä y:hyn, niin verkon vuo kasvaa arvolla 3

- Kutsumme täydennysreitiksi (engl. augmenting path) sellaista suuntaamatonta polkua verkossa, että vuota voi lisätä, eli jokaiselle oikeaan suuntaan kuljetulla kaarella on kapasiteettia jäljellä ja jokaisella takaperin kuljetulla kaarella vuo ei mene negatiiviseksi

- Voidaan osoittaa, että kun täydennysreittejä ei enää ole, vuo on maksimaalinen

- Tämä menetelmä kutsutaan Fordin-Fulkersonin menetelmäksi

- Lisätietoja: Cormenin luku 26

Verkkoalgoritmien yhteenveto tenttiä varten

- Olkoon verkossa $G = (V, E)$ n solmua ja m kaarta (eli $n = |V|$ ja $m = |E|$)

Algoritmi	Aikavaativuus	Tilavaativuus
BFS	$\mathcal{O}(n + m)$	$\mathcal{O}(n)$
DFS	$\mathcal{O}(n + m)$	$\mathcal{O}(n)$
Bellman-Ford	$\mathcal{O}(nm)$	$\mathcal{O}(n)$
Dijkstra	$\mathcal{O}((n + m) \log n)$	$\mathcal{O}(n)$
Floyd-Warshall	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$
Prim	$\mathcal{O}(m \log n)$ tai $\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Kruskal	$\mathcal{O}(m \log n)$	$\mathcal{O}(m)$

- Läpikäyntien suoria sovelluksia (eli aika- ja tilavaativuudet samat kuin läpikäynneillä):
  - Lyhin polku painottomassa verkossa = **BFS**
  - Sylkien löytäminen = **DFS** kohtaa harmaan solmun
  - Topologinen järjestäminen DAGissa = **DFS** + järjestäminen käänteisessä f-arvojärjestyksessä
  - Kriittiset työvaiheet DAGissa = pisin polku DAGissa = topologinen järjestäminen + pituuden laskeminen f-arvojärjesteyksessä = **DFS** + pituus siitä eteenpäin on selvä, kun solmu muuttuu mustaksi
  - Vahvasti yhtenäiset komponentit = **DFS** + verkon transponointi + transpoosiverkon **DFS**, läpikäynnin alkusolmuksi valitaan aina jäljellä olevista se jolla on suurin f-arvo
- Algoritmit soveltuvat kaikki sekä suunnattuihin että suuntaamattomiin verkkoihin, paitsi DAG-sovellukset sekä **Prim** ja **Kruskal**, koska pienin virittävä puu on määritelty vain suuntaamattomalle verkkolle

- Lyhin polku painotetussa verkossa:
  - Mikään näistä (**Bellman-Ford**, **Dijkstra**, **Floyd-Warshall**) ei toimi, jos on negatiivinen sykli
  - **Dijkstra** ei toimi negatiivisille painoille, muut kylläkin
- Pisin yksinkertainen polku painotetussa verkossa
  - Tämä on siis kriittisten polkujen etsintä, kun on DAG
  - Yleisessä tapauksessa ongelma on NP-täydellinen, josta seuraa, että ongelmalle ei tunneta polynomista ratkaisualgoritmia
  - Kirjoitamme tässä algoritmin auki, kun kyseessä on DAG

```
max-path(G)
    for jokaiselle solmulle u ∈ V
        color[u] = white
    for jokaiselle solmulle u ∈ V
        if color[u] == white
            DFS-visit3(u)
    return max{ h[u] | u ∈ V }
```

```
DFS-visit3(u)
    color[u] = gray
    h[u] = 0
    for jokaiselle solmulle v ∈ vierus[u]
        if color[v] == white
            DFS-visit3(v)
        if h[u] < w(u,v) + h[v]
            h[u] = w(u,v) + h[v]
```

9. Algoritminsuunnittelumenetelmiä

- Tämä luku on lähinnä kertausta kurssin aikana tavatuista algoritminsuunnittelumenetelmistä

- Tässä yhteydessä on hyvä muistuttaa invariantin käsitteestä, josta on iloa myös perusohjelmoinnissa turhien virheiden välttämiseksi. Ylipäätään ohjelmoinnissa on aina syytä tarkastella alku- ja lopetusehtoja, erikoistapauksia jne., ja invariantit ovat hyviä työkaluja erinäisten for- ja while-silmukoiden hallitsemiseksi

Raaka voima (Brute force)

- Helppo lähestymistapa on käydä kaikki mahdolliset tapaukset läpi
- Tämä menetelmä on yleensä tehoton, mutta käyttökelpoinen erityisesti pienikokoisille ongelmille tai jos muuta ei voida tehdä ongelman luonteen takia
- Esimerkki, jossa raaka voima on hyvä lähestymistapa:
  - Etsimme pienintä avainta järjestämättömästä linkitetystä listasta
  - Tässä käymme alkiot läpi yksitellen alusta lähtien, pidetään kirjaa tähän asti pienemmästä avaimesta, ja lopetamme kun lista loppuu
  - Tällaista hakumenetelmää kutsutaan peräkkäishauksi (engl. sequential search)
- Esimerkki, jossa raaka voima ei ole hyvä lähestymistapa:
  - Haluamme löytää tiheästä verkosta lyhimmän polun solmusta s solmuun t
  - Generoidaan kaikki mahdolliset polut s:stä t:hen ja laskemme kullekin polun pituus
  - Valitaan lyhin
- Tällaista ”kaikkia vaihtoehtoja luetteleva” menettelyä kutsutaan joskus leikkisästi British Museum -etsinnäksi.

Ahne algoritmi (Greedy algorithm)

- Edetään vaiheittain, kussakin vaiheessa siitä kohdasta katsottuna parhaaseen vaihtoehtoon. Kyse on siis paikallisesta optimoinnista
- Olemme nähneet tästä menetelmästä useita esimerkkejä:
  - **Dijkstran**, **Kruskalin** ja **Primin** algoritmit ovat tällaiset ja näissä tapauksissa menetelmä tuottaa todistettavasti optimitulokset
- Yleisesti ottaen tällä menetelmällä pääsee lokaaliin optimiin, joka voi olla globaalia optimia huonompi
- Optimointitehtävissä puhutaan usein naapuruushausta (engl. hill climbing): haluamme korkeimmalle kukkulalle, joten suuntaamme siihen suuntaan, jossa nousu on jyrkintä
- Esimerkki: Haluamme kaupungissa kävellä paikasta A paikkaan B
  - Otamme joka kadunkulmassa sen vaihtoehdon joka näyttää vievän parhaiten paikan B suuntaan

Peruuttava etsintä (Backtracking)

- Edetään kunnes tullaan umpikujaan ja palataan taaksepäin yrittämään uudestaan toista vaihtoehtoa
- Tätä menetelmää tarkastelimme luvun 3 lopussa
- Optimointitehtävissä pystytään myös karsimaan (prune) pitkin matkaa huonoja vaihtoehtoja (eli jätämme ne pois menemättä niihin syvemmin) ja menetelmää kutsutaan silloin nimellä branch-and-bound
- Peruuttavat algoritmit perustuvat usein rekursion käyttöön eli pinoon

Hajota ja hallitse (Divide and conquer)

- Tämä on tavallinen ongelman lähestymistapa tietojenkäsittelyssä: jaetaan ongelma pienempiin samanlaisiin ongelmiin (''hajota''), joita yhdistämällä (''hallitse'') saadaan lopullinen vastaus
- Tässäkin edetään rekursiivisesti: osaongelmat jaetaan puolestaan rekursiivisesti pienempiin osaongelmiin
- Huomaa: Kaikki rekursio ei ole hajota ja hallitse: tässä viitataan tapauksiin, jossa jaetaan ongelma vähintään kahdeksi (samantyyppiseksi) osaongelmaksi
- Esimerkkejä tällaisista algoritmeista on lomitusjärjestäminen ja pikajärjestäminen
- Binäärihaku on sukua tähän lähestymistapaan, mutta siinä vaan jaetaan ongelma pienemmäksi eikä tarvitse tehdä mitään yhdistämisoperaatiota lopussa. Tällaista tekniikkaa kutsutaan joskus nimellä yksinkertaista ja hallitse

Dynaaminen ohjelmointi (Dynamic programming)

- Rekursio voi monesti olla tehoton, joten voidaan tallentaa osaongelmien tulokset taulukkoon. Osatulosten taulukointia kutsutaan nimellä dynaaminen ohjelmointi

- Esimerkkejä tästä on mm. Fibonaccin lukujen iteratiivinen laskeminen parin apumuuttujan avulla sekä **Floydin-Warshallin algoritmi** (tai transitiivisen sulkeuman laskeminen), jossa päivitetään kaksiulotteista taulukkoa (matriisia)

Algoritmien suunnittelu

- Lisää algoritmien suunnittelusta seuraa kurssilla Design and Analysis of Algorithms
- Tällä kurssilla päätavoite on ymmärtää esitettyjen algoritmien ideat ja osata soveltaa niitä hieman muuntaen
- Käytännössä algoritmit
  - toteutetaan tietokonelaitteistolla ja
  - liittyvät osaksi sovellusohjelmaa
- Tällä kurssilla on lähinnä tarkasteltu asymptoottista pahimman tapauksen aikavaativuutta ("$\mathcal{O}$-notaatio"), joka kertoo yksinkertaisessa muodossa algoritmin skaalautuvuuden

- Sovelluksessa tarvitaan muutakin:
  - Kuinka isoja syötteet todella ovat? Helppoja vai vaikeita?
  - Mikä on käyttäjien tarvitsema/haluama riittävä suorituskyky? Mikä oikeasti on järjestelmän pullonkaula?
  - Jos algoritmia pitää tosissaan ruveta viilaamaan, miten se suhtautuu käytettävissä olevaan suoritusympäristöön (rinnakkaistuminen, välimuistin käyttö jne.)?
  - Käytännön tekijät: ohjelmiston ylläpidettävyys ja laajennettavuus, virheistä toipuminen jne.

Tietorakenteet ja algoritmit

abstrakti tietotyyppi: mitä datalle halutaan tehdä

tietorakenne: miten se tehdään (talletusrakenne, operaatioden toteutus)

- Kaikki ei ole ihan yksinkertaista:
  - tietorakenteiden toteuttamisessa voidaan tarvita ei-triviaaleja algoritmeja (esim. AVL-puiden tasapainotus)
  - algoritmien tehokkaassa toteuttamisessa voidaan tarvita ei-triviaaleja tietorakenteita (esim. keko ja union-find)

Millaisia asioita olemme oppineet

- Toisin sanoen mitä kurssista olisi hyvä muistaa vielä ensi vuonna
- Perustietorakenteita ja -algoritmeja, jotka on hyvä osata jopa koodata melko sujuvasti:
  - pino, jono, linkitetty lista, puu
  - syvyyssuuntainen ja leveyssuuntainen läpikäynti
- Yleisesti tarvittavia tietorakenteita ja algoritmeja, joita ei yleensä tarvitse (tai kannata) itse koodata, mutta keskeiset periaatteet pitää tuntea:
  - hakemistorakenteet: puu vs. hajautus
  - järjestäminen: $\mathcal{O}(n)$ vs. $\mathcal{O}(n \log n)$ vs. $\mathcal{O}(n^2)$; pikajärjestämisen vaikeat tapaukset

- Mallinnustekniikoita: etenkin puiden ja verkkojen käyttäminen ongelmanratkaisun mallina
  - tekoäly: pelipuut, puun läpikäynnit, verkko-algoritmit, graafiset mallit (verkkopohjaisia todennäköisyysmalleja)
  - tietoliikenne: esim. reitittäminen **Dijkstran algoritmilla**
  - ohjelmointi ja ohjelmointikielet: jäsennyspuu, verkkopohjaiset kaavioesitykset
  - tietojenkäsittelyteoria: puut ja verkot abstrakteissa laskennan malleissa

Loppusanat

- Kurssi on toivottavasti ollut teille antoisa ja hyödyllinen
- Muistakaa kurssipalautteen antamista. Erityisen arvokkaita ovat suositukset, miten kurssia voisi parantaa

KIITOS!