58131 Tietorakenteet

kevät 2010

Matti Luukkainen

Lähdemateriaali

- Moniste perustuu monilta osin keväällä 2004 kirjoittamaani luentomateriaaliin, sen Matti Nykäsen 2005-06 täydentämään versioon sekä Jyrki Kivisen 2008 kirjoittamaan luentomateriaaliin

  Suuri osa tekstiä on kuitenkin kirjoitettu uudelleen

- Oma vanhempi materiaalini perustui lähes suoraan kirjan

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

  toiseen painokseen. Nyt kirjasta siis on ilmestynyt 3. painos joka ei kuitenkaan poikkea kovin paljoa edellisestä.

- 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

  Edellisten TiRa-kurssien pseudokoodi noudattaa Cormenin toisen painoksen muotoa

- Kivisen materiaalissa siirryttiin ns. punamustien puiden sijasta käsittelemään AVL-puita, joihin hyvä lähde on

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

- Myös tässä monisteessa tullaan käsittelemään AVL-puita joita ei Cormenissa käsitellä oolenkaan

- Cormenin ulkopuolelta käsitellään myös ns. B+-puu, joka poikkeaa jonkin verran Cormenissa esiteltävästä B-puusta

- Joitakin lähinnä verkkoalgoritmeihin liittyviä todistuksia muotoilen uudelleen hiukan Cormenin raskaahkosta esitystavasta poikkeavasti, käyttäen lähteinä mm. seuraavia

  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

1. Johdanto

- Kaikki epätriviaalit ohjelmat joutuvat tallettamaan ja käsittelemään tietoa suoritusaikanaan

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

- suoritusaikana tiedot tallennetaan *tietorakenteeseen*

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* (ADT)
  - tietorakenteen sisäinen toteutus piilotetaan käyttäjältä
  - abstrakti tietotyyppi näkyy käyttäjille ainoastaan operaatioina minkä avulla tietoa käytetään
  - abstrakti tietotyyppi ei siis ota kantaa siihen miten tieto on koneen muistiin varastoitu

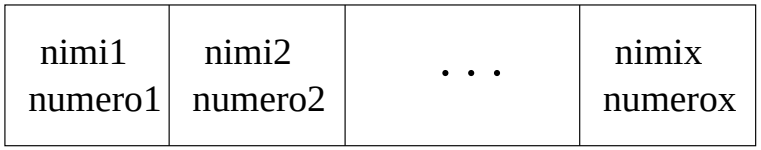
Esim. puhelinluettelo-ohjelmiston tietorakenne

- Tietorakenteen operaatioita ovat ainakin seuraavat:

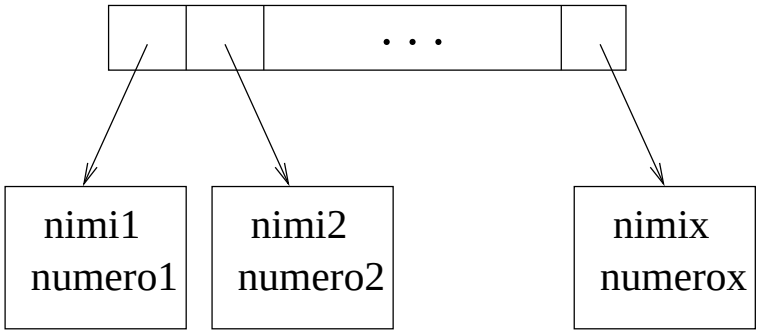
  - **init()** luodaan tyhjä luettelo
  - **add(n,p)** lisää luetteloon nimen n ja sille numeron p
  - **del(n)** poistaa nimen n luettelosta
  - **findNumber(n)** palauttaa luettelosta henkilön n numeron
  - **findName(p)** palauttaa luettelosta numeron p omistajan
  - **update(n,p)** muuttaa n:lle numeroksi p:n
  - **list()** listaa nimi-numero -parit 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 osoittimia nimi/numero-pari-olioihin

- Taulukkoon perustuvat tietorakenteet ovat tehottomia 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?

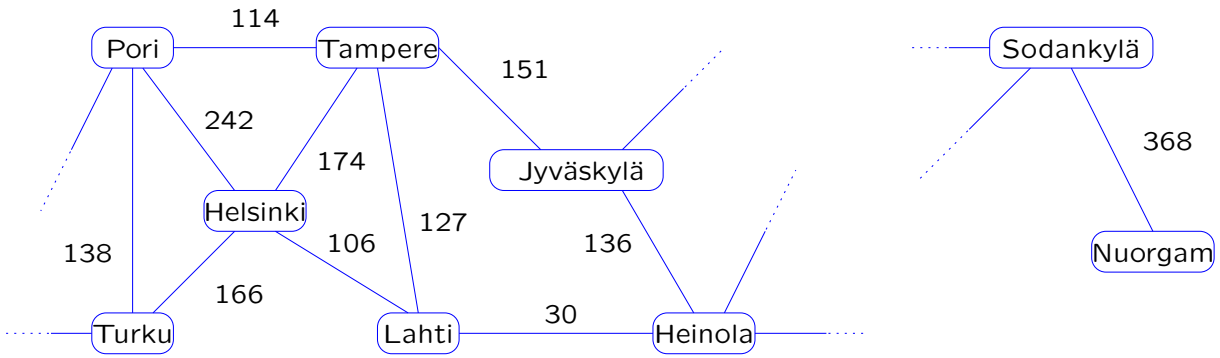
Helsinki–Lahti	106 km
Helsinki–Turku	166 km
Turku–Pori	138 km
Lahti–Tampere	127 km
...	...
Sodankylä–Nuorgam	368 km

$\Longrightarrow$

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 30 paikkakuntaa $\Rightarrow$ $30! \approx 2,7 \cdot 10^{32}$ 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)
  - sekä 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

Miksi pakollinen kurssi Tietorakenteet?

- Puhelinluettelo on erittäin helppo toteuttaa Java API:ssa olevasta kokoelmakehyksestä (Collections Framework) suoraan löytyvien tietorakennetoteutusten (esim. Arraylist, HashMap, TreeMap) avulla

  Miksi siis vaivautua opiskelemaan tietorakenteita 8 op:n verran?

- Vaikka tietorakennetoteutuksia on olemassa, ei niitä välttämättä osata hyödyntää tehokkaasti ilman teoreettista taustatietämystä

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

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

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

- On siis hyvä omata 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 jo havaittiin, pienikin ongelma typerästi ratkaistuna on niin vaikea, ettei nopeinkaan kone siihen pysty
- Vaikka koneet nopeutuvat, käsiteltävät datamäärät kasvavat suhteessa jopa enemmän sillä 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
  - 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: eli käytetään matematiikkaa ja samalla myös opitaan matematiikkaa
- Näistä taidoista hyötyä toisen vuoden kurssilla Laskennan mallit sekä kaikilla Algoritmit ja koneoppiminen -linjan kursseilla, joista erityisesti kurssia Algoritmien suunnittelu ja analyysi 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 sisältö ja tavoitteet

- Ennen kuin menemme asiaan, tarkastellaan kurssin sisältöä ja tavoitteita vielä korkealla tasolla

- Sisältö ja asioiden käsittelyjärjestys

  1. Johdanto: algoritmien oikeellisuus, vaativuus, matemaattisia työkaluja
  2. Perustietorakenteita: pino, jono ja lista, niiden toteutus ja käyttäminen
  3. Hakupuut: eräs tehokas tapa suurten tietomäärien organisoimiseen
  4. Hajautus: toinen tehokas tapa suurten tietomäärien organisoimiseen
  5. Keko: monien algoritmien käyttämä hyödyllinen aputietorakenne
  6. Järjestäminen
  7. Verkot: tallettaminen, perusalgoritmit, polunetsintä, virittävät puut

- 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?
- Oppimitavoitematriisi (ks. kurssisivu) määrittelee tarkemmin, mikä on oleellisinta kurssilla.
  - Sarake *lähestyy oppimistavoitetta* määrittelee suunilleen arvosanaan 1 vaadittavan osaamisen tason
  - Sarake *saavuttaa oppistavoitteet* määrittelee korkeimpiin arvosanoihin vaadittavan oppimisen tason
- Osa asioista (erityisesti algoritmien ja tietorakenteiden toteuttaminen Javalla) on sen luonteista, että niiden hallinta osoitetaan kokeiden sijasta vain laskareissa

Algoritmien oikeellisuus

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

- tällöin operaation **list** yhteydessä nimet täytyy järjestää aakkosjärjestykseen

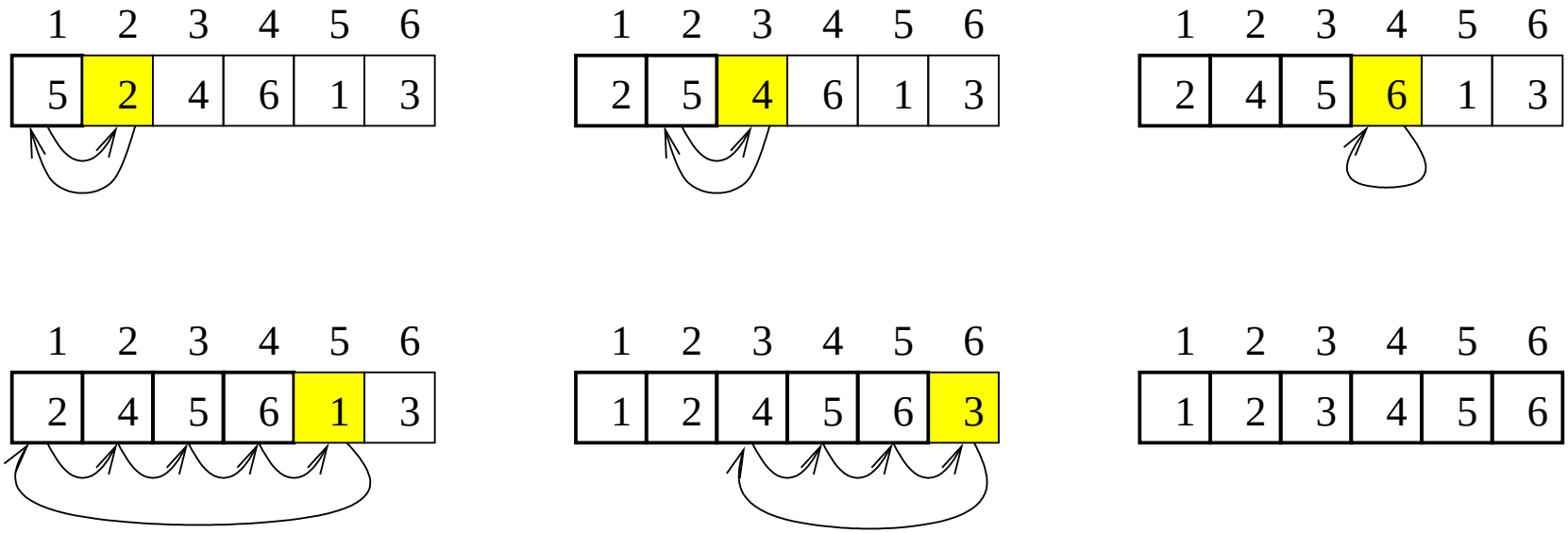
- järjestäminen on yleisemminkin tärkeässä roolissa tietojenkäsittelyssä ja myöhemmin kurssilla tarkastellaan muutamia tehokkaita järjestämisalgoritmeja

- tarkastellaan nyt esimerkkinä kurssilta ohjelmoinnin perusteet tuttua lisäysjärjestämistä (insertion sort), pseudokoodina:

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

- toimintaideana algoritmissa on pitää koko ajan taulukon alkupää $A[1, \ldots, j-1]$ järjestyksessä:
  - alussa $j == 2$ eli alkupää sisältää vain yhden luvun
  - while-loopissa 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 $A[j] = key$ jolle etsitään taulukon alkupäästä oikea paikka



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

  jokaisen **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 loopin suorituksen ajan

- erityisesti invariantti on vielä totta siinäkin vaiheessa kun loopin suoritus loppuu

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

- **for**-loopin alussa $j == 2$, joten invariantti

  jokaisen **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
  - whilen jälkeen alkuosa $A[1, \ldots, j]$ on järjestyksessä
  - taulukon loppuosaan $A[j+1, \ldots, A.length]$ 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 $j = A.length + 1$ ja koska invariantti pysyy voimassa seuraa tästä että $A[1, \ldots, A.length]$ 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!), sensijaan **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 peräkkäisitä osista, joissa kaikissa on oma looppinsa, tarvitaan kaikille loopeille omat invariantit. Yleensä myöhempien looppien invariantit perustuvat oleellisesti siihen tilanteeseen, johon algoritmin syöte on muotoutunut ensimmäisten looppien suorituksessa

Invariantin löytäminen

- Invariantti siis ilmaisee jonkin asiain tilan, joka on voimassa koko loopin suorituksen ajan
- Invariantti tavallaan kiteyttää koko loopin 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 hyödytyn
- Invarianttitodistuksissa lähes suurimmaksi haasteeksi osoittautuukin sopivan invariantin löytäminen
- Ilman syvällistä ymmärtämystä algoritmin toimintaperiaatteesta, on sopivaa invarianttia lähes mahdoton löytää

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 kolmatta osaa, joka sanoo, että loopin 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 sinäänsä tarvitse olla rajattu. Invarianttitodistuksessa kuitenkin tiedetään, että taulukolla on joku koko, eli "taulukon ulkopuolella" olevia lukuja ei tarvitse järjestää ja on oleellista, että loopin suorittaminen loppuu jossain vaiheessa

Huomautuksia monisteessa käytettävästä pseudokoodista

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

    Tästä seurauksena se, että monimutkaiseen parametriin (taulukko tai olio) metodissa tehty muutos näkyy kutsujalle

- Pseudokoodin eroavuuksia Javaan:
  - 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*

Algoritmien vaativuus

- Algoritmin tehokkuudella tai vaativuudella tarkoitetaan sen suoritusaikana tarvitsemia resursseja, esim.
  - laskentaan kuluva aika
  - laskennan vaatima muistitila
  - levyhakujen määrä
  - verkossa toimivilla algoritmeilla 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 lisäysjärjestämisessä syötteen koko on taulukon A alkioiden lukumäärä
- 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 tai if-, while ja for-lauseiden aiheuttaman haarautuminen) suorittaminen vie vakiomäärän tälläisia aikayksiköitä
  - sen sijaan esim. mielivaltaisen kokoisen taulukon järjestäminen ei onnistu vakiomäärällä aikayksiköitä
- Aikavaativuus voidaan esittää 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 olisi 100, kuluttaisi 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, joku luku (int, long, double) tai olioviite (eli käytännössä muistiosoite)
  - oleellista on, että tälläinen muistipaikka on rajatun kokoinen, esim. tietokoneessa 64:llä bitillä esitettävä
  - esim. taulukkoa ei voi tallentaa yhteen muistipaikkaan
  - yhteen tämän määritelmän mukaiseen muistipaikkaan ei voi tallettaa myöskään esim. Pythonin tarjoamaa rajattoman kokoista konaislukutyyppistä arvoa
  - rajattoman kokoisen luvun viemä talletustila nimittäin riippuu luvun pituudesta, eli mielivaltainen tälläinen luku ei voi mahtua mihinkään kiinteän kokoiseen muistipaikkaan

- 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 syötteenkseen 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 suunnilleen yhtä vaikeita kuin pahimmatkin (esim. lisäysjärjestämisellä)
  - monilla algoritmeilla pahimmat tapaukset yleisiä
- Pahimman tapauksen analyysi antaa useimmiten varsin hyvän kuvan algoritmin vaativuudesta. Muutamia poikkeuksiakin on, kuten tulemme huomaamaan

Lisäysjärjestämisen aikavaativuus monimutkaisesti laskettuna

- Oletetaan että rivin i suorittaminen kestää c_i aikayksikköä

  merkitään t_j:llä kuinka monta kertaa while-lausekkeen testi suoritetaan kullakin j:n arvolla ja merkitään n:llä syötteenä olevan taulukon A pituutta

- Seuraavassa on merkitty kullekin koodiriville sen vievien aikayksiköiden määrä sekä tieto kuinka monta kertaa rivi suoritetaan

		aika	suorituskertojen määrä
1	for j = 2 to A.length	c_1	n
2	key = A[j]	c_2	$n-1$
3	// ...	0	$n-1$
4	i = j–1	c_4	$n-1$
5	while i>0 and A[i]>key	c_5	$\sum_{j=2}^{n} t_j$
6	A[i+1] = A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	i = i–1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	A[i+1] = key	c_8	$n-1$

- nyt voimme laskea kuinka kauan suoritukseen kuluu summammalla rivit

- käytetään merkintää $T(n)$ suoritusajasta syötteellä, jonka pituus on n. laskemalla kuvasta rivien suoritusaikojen summat saamme:

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

- *parhaassa tapauksessa* (taulukko on jo valmiina järjestyksessä) **while**-testi tehdään aina vaan kerran, eli $t_j = 1$, ja saamme sievennyksen jälkeen

$$T_{best}(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

- jos ajatellaan toteutetun algoritmin viemää aikaa, vakiot $c_1, \ldots, c_8$ riippuvat ohjelmointikielestä, käytettävästä laitteistosta, laitteiston kuormituksesta ym. itse algoritmin tehokkuutta ajatellen toisarvoisista seikoista

- unohdetaan konkreettiset vakiokertoimet ja merkitään $T_{best}(n) = an + b$ joillain vakioilla a ja b eli $T_{best}(n)$ on lineaarinen funktio syötteen pituuteen n nähden

- *Lisäysjärjestäminen toimii parhaassa tapauksessa lineaarisessa ajassa syötteen pituuteen nähden*: syötteen koon kaksinkertaistuminen kaksinkertaistaa algoritmin suoritusajan

- Kuten pian näemme, on ikävät vakiot tapana "unohtaa" kokonaan ja merkitä parhaan tapauksen aikavaativuus seuraavasti: $T_{best}(n) = \mathcal{O}(n)$.

- *pahimmassa tapauksessa* (taulukko käänteisessä järjestyksessä) kohdan $A[j]$ alkio viedään aina taulukon alkuun, eli $t_j = j$, saamme

$$T_{worst}(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1)$$
$$+ c_7 \sum_{j=2}^{n} (j-1) + c_8(n-1)$$

- Sovelletaan kaavaa $\sum_{i=1}^{n} i = \frac{1}{2} n(n+1)$, ja $T_{worst}(n)$ sievenee pienen pyörittelyn jälkeen muotoon:

$$T_{worst}(n) = \frac{n^2}{2}(c_5 + c_6 + c_7) + \frac{n}{2}(2c_1 + 2c_2 + 2c_4 + c_5 + c_6 + c_7 + 2c_8) -$$
$$(c_2 + c_4 + c_5 + c_8),$$

- Unohtamalla konkreettiset vakiot, voimme merkitä $T_{worst}(n) = an^2 + bn + c$ joillain vakioilla a, b ja c

- *Lisäysjärjestäminen toimii pahimmassa tapauksessa neliöllisessö ajassa syötteen pituuteen nähden*: syötteen koon kaksinkertaistuminen nelinkertaistaa algoritmin suoritusajan

- Pahimman tapauksen aikavaativuuden neliölisyys voidaan merkitä $T_{worst}(n) = \mathcal{O}(n^2)$.

Algoritmien vaativuusluokat

- Algoritmien vaativuuden arvioinnissa vakiot, kuten edellisten sivujen a, b ja $c, c_1, \ldots, c_8$ ovat ohjelmointikieli- ja suoritusympäristökohtaisia

- jatkossa arvioimmekin algoritmien vaativuutta karkeammin, ainoastaan luonnehtimalla mihin *vaativuusluokkaan* algoritmi kuuluu

- edellä jo totesimme, että lisäysjärjestäminen toimii
  - parhaassa tapauksessa lineaarisesti syötteen pituuteen nähden, merkitsimme tätä $T_{best}(n) = \mathcal{O}(n)$
  - ja neliöllisesti pahimmassa tapauksessa, ja tästä käytettiin merkintää $T_{worst}(n) = \mathcal{O}(n^2)$

- Tapana on siis "unohtaa" vakiokertoimet sekä vähemmän merkitsevät osat (kuten pahimmassa tapauksessa ensimmäisen asteen termi) 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 vaikeimpaan

  - $\mathcal{O}(1)$ 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, kuten lisäysjärjestäminen selviää vielä miljoonienkin kokoisista syötteistä

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

Onko algoritmien vaativuuden luokittelu vaativuusluokkiin perustuen mielekäs?

- Kuten näimme, lisäysjärjestäminen toimii pahimmassa tapauksessa ajassa $T_{worst}(n) = \mathcal{O}(n^2)$
- Kurssin aikana tutustumme muutamiin pahimmassa tapauksessa ajassa $T(n) = \mathcal{O}(n \log n)$ toimiviin algoritmeihin
- 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 $T_{lis} = 2n^2$, eli vaikiokerroin olisi vain 2

  ja että aloitteleva ohjelmoija on toteuttanut ajassa $\mathcal{O}(n \log n)$ toimivan lomitusjärjestämisen melko epäoptimaalisesti, ja toteutuksen tarkkaa suoritusaikaa kuvaa funktio $T_{lom} = 50n \log n$
- 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 miljoonan kokoinen taulukko

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

  ja lomitusjärjestäminen suoritetaan koneella, joka suorittaa ainoastaan 10^6 komentoa sekunnissa. A on siis 1000 kertaa nopeampi kone kuin B

- Nyt nopea kone A järjestää hyvin toteutetulla lisäysjärjestämisellä luvut 20000 sekunnissa eli reilussa 5.5 tunnissa

  Tuhat kertaa hitaampi kone B järjestää luvut epäoptimaalisella lomitusjärjestämisellä 1163 sekunnissa, eli alle 20:ssä minuutissa

- Jos järjestettävän taulukon koko olisi 100 miljoonaa, olisi ero vielä radikaalimpi, lisäysjärjestämisellä kuluisi aikaa 23 päivää mutta lomitusjärjestäminen selviäisi alle neljässä tunnissa

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

Funktioiden kertaluokat

- Määritellään nyt matemaattisesti, mitä tarkoitetaan esim. merkinnällä $T(n) = \mathcal{O}(n^2)$.
- Tarkastellaan funktioita $f : \mathbb{N} \to \mathbb{R}$ ja $g : \mathbb{N} \to \mathbb{R}$
- Sanotaan että f kuuluu kertaluokkaan $\mathcal{O}(g)$ jos on olemassa vakiot $d > 0$ ja n_0 siten että kaikilla $n > n_0$ ehto $0 \leq f(n) \leq dg(n)$ 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
- tällöin merkitään $f = \mathcal{O}(g)$

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

  resurssitarpeiden yläraja on g

- 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 lisäysjärjestämiseen:
  - parhaassa tapauksessa algoritmi kulutti aikaa $an + b \leq n(a + b) = \mathcal{O}(n)$, sillä vakioksi voidaan valita $d = a + b$ ja $n_0 = 1$
  - pahimmassa tapauksessa $an^2 + bn + c \leq n^2(a + b + c) = \mathcal{O}(n^2)$ sillä vakioksi voidaan valita $d = a + b + c$ 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 $f(n) \leq 123 * 1$, 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 $n^2 = \mathcal{O}(n^5)$, voidaan merkitä että lisäysjärjestäminen on pahimmassa tapauksessa esim. $\mathcal{O}(n^5)$ mutta tämä ei tietenkään ole kovin järjevää
  - mielekästä onkin etsiä pienintä mahdollista argumenttifunktioita f, jolla algoritmin vaativuus on $\mathcal{O}(f)$

- vaihtoehtoisesti voidaan esittää vaativuudelle *alaraja*, ja tällöin käytössä merkintä Ω

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

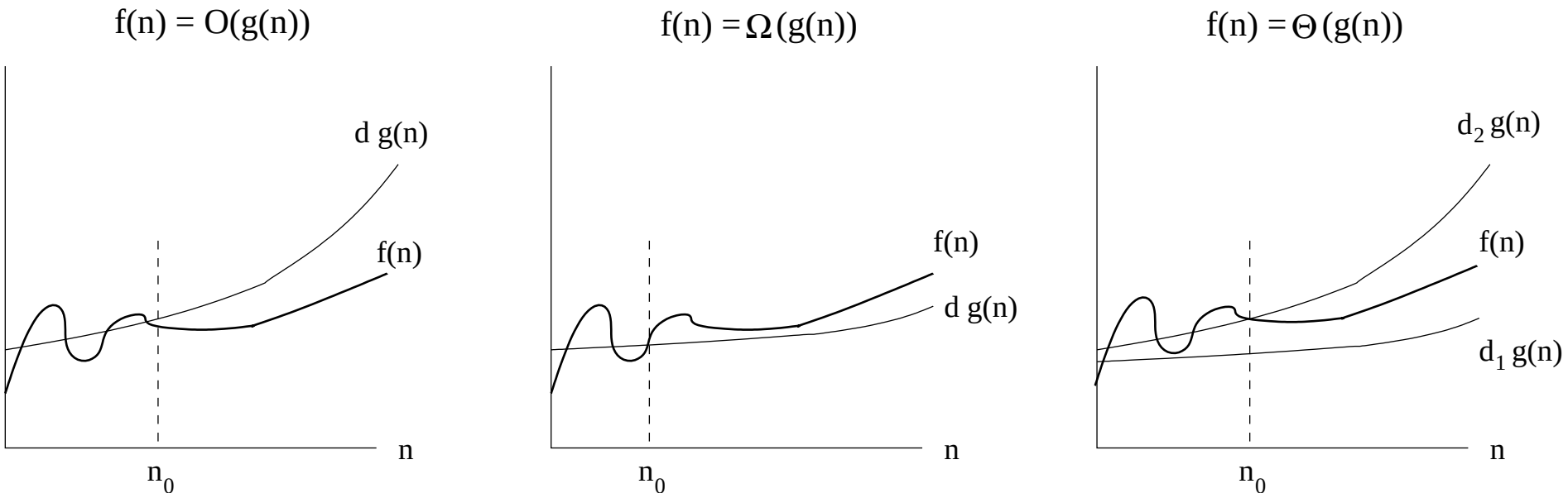
- Esim: valitsemalla $d \leq a$ huomaamme että $dn^2 \leq an^2 \leq an^2 + bn + c$ eli lisäysjärjestäminen on pahimmassa tapauksessa myös $\Omega(n^2)$

  siis ylä- ja alaraja vaativuudelle on sama, eli kyseessä on tarkka arvio, myös tälle tapaukselle on olemassa oma merkintätapa

- g on sekä ylä- että alaraja f:lle, merkitään $f = \Theta(g)$, jos ja vain jos $f = \mathcal{O}(g)$ ja $f = \Omega(g)$
  - nyt olemassa vakiot d_1 ja d_2 siten että funktio $f(n)$ jää kuvaajien $d_1 g(n)$ ja $d_2 g(n)$ väliin
  - f käyttäytyy oleellisesti samoin kuin g, toisin sanoen f:llä sama *asymptoottinen* kasvunopeus kuin g:llä

- Esim: koska lisäysjärjestämisen pahimman tapauksen yläraja $\mathcal{O}(n^2)$ sekä alaraja $\Omega(n^2)$, voimme käyttää merkintää $\Theta(n^2)$

- 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 harhaan johtavan käytön voi ehkäistä käyttämällä tarkempaa merkintää Θ, joka siis rajaa funktion sekä ylä- että alapuolelta. Cormenin kirjassa on tapana käyttää pääosin Θ-merkintää
- Alarajaa merkitsevä Ω ei esiinny käytännössä kovinkaan usein

Sääntöjä algoritmien analysointiin

- yksinkertaisten käskyjen, eli muuttujia käsittelevien sijoitus-, syöttö-, tulostuskäskyjen sekä niitä käsittelevien yksinkertaisten vertailujen ja aritmeettisten operaatioiden aikavaativuus on vakio, eli $\mathcal{O}(1)$

- Jos S_1:n aikavaativuus on $f_1(n)$ ja S_2:n aikavaativuus on $f_2(n)$, 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, että usean peräkkäisen yksinkertaisen käskyn suorituksen aikavaativuus on siis vakio eli $\mathcal{O}(1)$

- ehtolauseen aikavaativuus on $\mathcal{O}$(ehdon testausaika + suoritetun vaihtoehdon aikavaativuus)

- 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 neljää apumuuttujaa

- entä aikavaativuus?

- Merkitään S_{i-j}:llä riviltä i riville j muodostuvaa osaa koodista.
- Syöte, sijoitus ja vertailukäskyjen aikavaativuus vakio eli $\mathcal{O}(1)$. Edellisen sivun sääntöjen 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 vakioiaikainen operaatio. Jos rivin 5 ehto epätosi, suoritetaan rivi 7, joka myös selvästi onnistuu vakioajassa. Eli S_{5-10} aikavativuus $\mathcal{O}(1)$.
- 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 pääteltyä, 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

- aliohjelman/metodin suorituksen aikavaativuus on $\mathcal{O}$(parametrien evaluointiaika + parametrien sijoitusaika + aliohjelman/metodin käskyjen suoritusaika)

- Eli jos edellisen esimerkin metodia kutsutaan, on koko suorituksen aikavaativuus myös $\mathcal{O}(1)$ sillä parametrina on taulukko, ja taulukot välitetään viitteiden tapaan eli "parametrin sijoitusaika" on myös $\mathcal{O}(1)$

- while-silmukan aikavaativuus on $\mathcal{O}(k(f_1 + f_2))$, missä k toistojen määrä f_1 lopetusehdon testausaika ja f_2 silmukan rungon suoritusaika, jos lopetusehdon ja rungon suoritusaika on aina sama

  Näin ei välttämättä ole, sillä usein rungon suoritusaika riippuu jonkin joka kierros uuden arvon saavan muuttujan arvosta

- Oletetaan, että A on järjestetty kokonaislukutaulukko. Mikä on seuraavassa esitetyn peräkkäishaun aikavaativuus?

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

- Ennen ja jälkeen toiston olevat osat suoritetaan selvästi vakioajassa
- Pahimmassa tapauksessa etsittävää lukua ei löydy, joten toisto käy koko taulukko läpi. Tällöin whilen runko-osa suoritetaan $n = A.length$ kertaa

  Toistolauseen runko suoritetaan vakioajassa samoin kahdesta vakioaikaisesta operaatiosta koostuva toistoehto.

  Toisto-osan vaativuus on siis $\mathcal{O}(n)$
- Suoritetaan peräkkäin $\mathcal{O}(1)$, $\mathcal{O}(n)$ ja $\mathcal{O}(1)$ vievät osat, eli koko metodin aikavaativuus on $\mathcal{O}(n)$, suhteessa syötteenä olevan taulukon pituuteen n
- Metodin aikavaativuus on niin ilmeinen, että yleensä sitä ei jaksettaisi näin tarkkaan analysoida

- edellisen esimerkin sisältämän while-loopin aikavaativuus oli helppo laskea sillä whilen runko-osan suoritsaika on jokaisella suorituskerralla sama

- sisäkkäisten looppien tapauksessa analysointi ei yleensä ole näin yksinkertaista, sillä sisemmän loopin suorituskertojen määrä riippuu usein ulomman loopin looppimuuttujasta
  - näin oli lisäysjärjestämisessä, missä jokaisella rungon suorituskerralla vietiin yksi alkio oikealle paikalle koko ajan kasvavaan valmiina järjestyksessä olevana osaan

- yleisen looppeja sisältävän algoritmin aikavaativuuden laskemiseen ei ole olemassa oikeastaan mitään yleispätevää hyödyllistä sääntöä

- paras strategia lienee yrittää arvioida sisimmän loopin runko-osan (joka on yleensä vakioaikainen) suoritusten lukumäärää suhteessa syötteen pituuteen

Esimerkki: Kuplajärjestäminen

Bubble-Sort(A)

```
1   for i = n to 2
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

    taulukon osa $A[2,\ldots,n]$ järjestyksessä ja järjestetyn osan alkiot vähintään yhtä suuria kuin paikan $A[1]$ alkio

    siis koko taulukko 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: toisin kuin lisäysjärjestämisellä aikavaativuus on nyt sama *kaikissa tapauksissa*

Pikakertaus: logaritmi

- Koulumatematiikassa kerrottiin, että $\log_a x$ on sellainen luku z, jolla $a^z = x$, tässä oletetaan että $a > 0$ ja $a \neq 1$
  - Toisin sanoen $\log_a x$ on vastaus kysymykseen "mihin potenssiin a täytyy korottaa jota saadaan x?
  - Esim. paljonko on $\log_2 8$? Jos 2 korotetaan potenssiin 3 saadaan 8, eli $\log_2 = 8$ koska $2^3 = 8$.

- Tietojenkäsittelijöille tärkein on kaksikantainen logaritmi $\log_2 n$, joka kertoo suunilleen
  - montako kertaa n pitää puolittaa, että pääastään alle 2:n
  - luonnollisen luvun n pituuden bitteinä (miinus 1)

- Laskusääntöjä:

$$\begin{aligned} \log_a a &= 1 \\ \log_a(xy) &= \log_a x + \log_a y \\ \log_a(x^y) &= y \log_a x \\ \log_a x &= \frac{\log_b x}{\log_b a} \end{aligned}$$

- Viimeinen edellisen sivun säännöistä 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 $f_A(n) = 7\log_{55} n$, niin molemmat $\mathcal{O}$-analyysin mielessä yhtä vaativia, eli $f_A(n) = \mathcal{O}(\log n)$ ja $f_A(n) = \mathcal{O}(\log n)$

- 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 vaan
  - 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 $j = \log_2(n+1) - 1$

- 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

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
8       else vasen = keski+1
```

- Invariantti:

  jos x on taulukossa niin se on välillä $A[vasen, oikea]$ ja $found = tosi$ 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

- 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 joku 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 joku 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 oikeellisuuteen, tulemme perustelemaan asian myöhemmin kurssilla matemaattisesti

Rekursiiviset metodit ja rekursioyhtälöt

- Tarkastellaan taas yhtä tapaa etsiä haluttu luku järjestetystä taulukosta. Tällä kertaa kyseessä on rekursiivisesti toimiva metodi:

```
Stupid-Find(A,i,x)
1   if i > A.length or A[i]>x
2       return false
3   elsif A[i] = x
4       return true
5   else
6       return Stupid-Find(A,i+1,x)
```

- Metodia siis kutsutaan **Stupid-Find(A,1,x)** kun halutaan tarkastaa löytyykö luku x taulukosta A ja aloitetaan etsintä taulukon kohdasta 1

- Mikä on algoritmin pahimman tapauksen aikavaativuus suhteessa syötteenä olevan taulukon kokoon n?

- Rivien 1-4 aikavaativuus on selvästi $\mathcal{O}(1)$. Entä rekursiivinen kutsu?

- Rekursiivisten metodien aikavaativuus on yleensä helpointa määritellä rekursioyhtälönä

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

- T voidaan määritellä seuraavasti rekursioyhtälönä:

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

- Eli jos jäljellä on nollan mittainen osa taulukkoa, suoritetaan rivit 1 ja 2 eli vakioaikainen operaatio

- Jos jäljellä olevan osan k pituus on vähintään 1
  - parhaassa tapauksessa etsittävä löytyy, mutta huonoimmassa tapauksessa täytyy tehdä rekursiivinen kutsu, jossa syötteenä taulukon osa jonka pituus $k-1$, tämän aikavaativuus on $T(k-1)$
  - vertailu, jonka vaativuus on $\mathcal{O}(1)$ tehdään joka tapauksessa

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

- Oletetaan että c on joku tarpeeksi suuri vakio, ja kirjoitetaan rekursioyhtälö seuraavasti:

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

- Ruvetaan taas laskemaan rekursioyhtälöä auki:
  $T(n) = T(n-1) + c$ ja $T(n-1) = T(n-2) + c$ eli $T(n) = T(n-2) + 2c$

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

- Eli pahimman tapauksen aikavaativuus lineaarinen suhteessa taulukon kokoon niinkuin olettaa saattaa sillä jokainen rekursiokutsu poistaa etsintäalueelta vakiomäärän (eli yhden) alkion

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

- Kurssilla *Tietokoneen toiminta* tullaan oppimaan, että 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 **Find-Stupid(A,1,x)** suorittaa rekursiivisen kutsun **Find-Stupid(A,2,x)**, jää sen itse varaama tila vielä pinoon sillä metodikutsu ei ole ohi ennen kuin rekursiivisesta kutsusta palataan

  Vastaavasti kun **Find-Stupid(A,2,x)** kutsuu **Find-Stupid(A,3,x)**, jää sen varaama tila pinoon, jossa on ennestään jo kutsun **Find-Stupid(A,1,x)** tilanvaraus. Kun lopulta kutsutaan rekursion päättävä **Find-Stupid(A,n,x)**, on pinossa n+1 kappaletta rekursiivisten kutsujen tilanvarauksia

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

- Näennäisestä yksinkertaisuudestaan huolimatta rekursio siis saattaa kätkeä alleen suurta tilankäyttöä ja pahimmassa tapauksessa myös aikavaativuus voi rekursiivisella ratkaisulla olla paljon suurempi kun ilman rekursiota toteutetulla algoritmilla (esim. Fibonaccin lukujen tuottaminen)

- Olemme jo nähneet pariin kertaan miten rekursioyhtälö ratkeaa mukavasti. Aina ei näin ole. Kurssilla *Algoritmien suunnittelu ja analyysi* opitaan ratkaisemaan rekursioyhtälöitä, joihin käyttämämme menetelmä ei toimi

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

- Kalvolla *Pikakertaus: logaritmi* perustelimme minkä takia logaritimien 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 forista 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. Aikavativuusanalyysissä 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 loopattavan alueen koko vaihtelee tiedon määrän mukaan
- Vastaavasti eksponentiaalisissa algoritmeissa voidaan ajatella, että "for-silmukoiden" määrä vaihtelee tiedon määrän mukaan, mutta loopin 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.

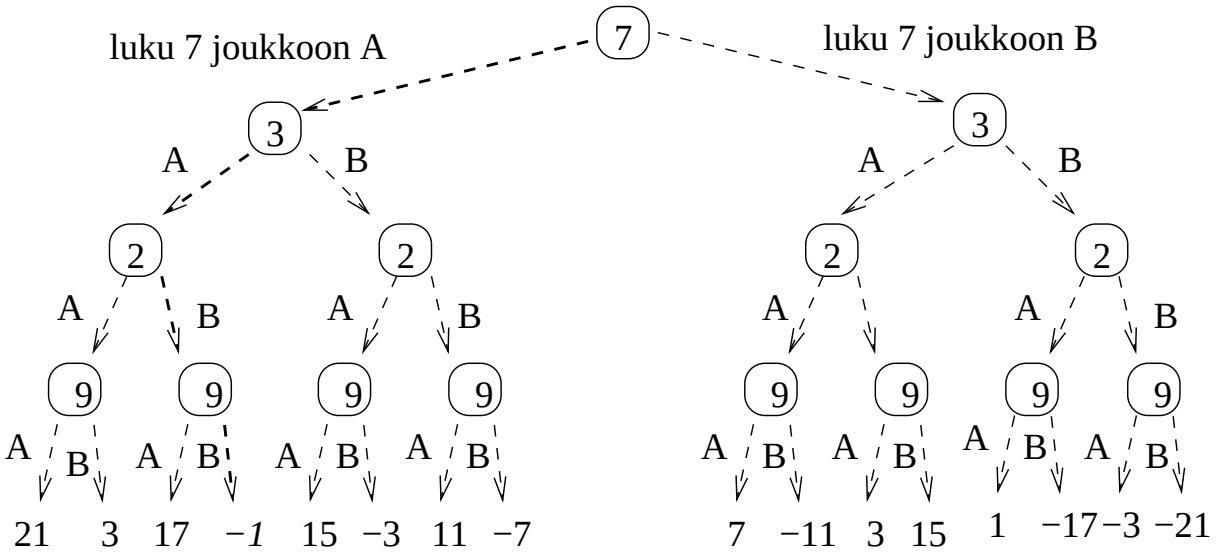
  Algoritmi haarautuu joka luvun kohdalla kahden mahdollisuuden mukaan: joko luku menee ryhmään A tai sitten se menee ryhmään B.

  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 kasvaa räjähdysmäisesti

Osaamisen taso algoritmien suhteen

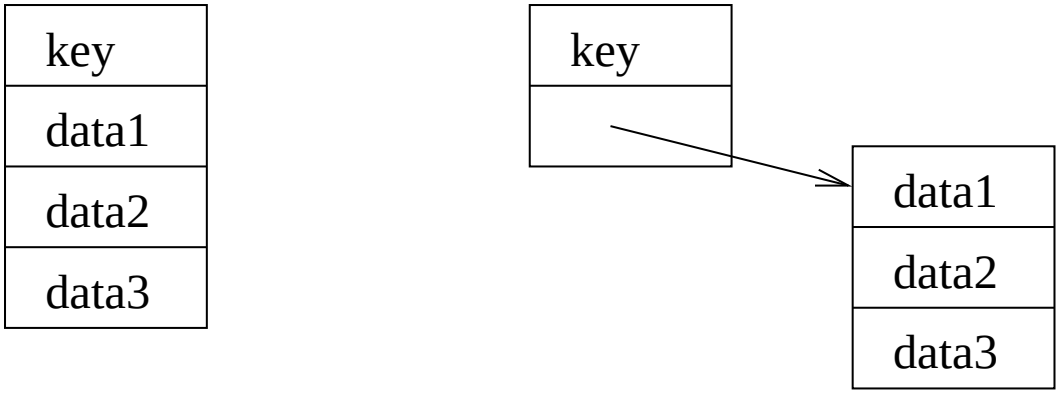
- Kurssilla tulee eteen suuri määrä algoritmeja ja tietorakenteita. Millä tasolla ne tulisi oppia?
- Algoritmisen osaamisen tasoja tai lajeja voisi kuvitella olevan ainakin seuraavat:
  1. osaa lukea monisteesta algoritmin ja kopioida sen lunttilapulle ja lunttilapulta koepaperille
  2. tuntee esim. 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 6-8:
  - ymmärtää miksi algoritmi toimii oikein
  - ymmärtää miten paljon algoritmi käyttää resursseja
  - osaa soveltaa algoritmia epätriviaalilla tavalla
- Tärkeitä ovat myös tasot 9 ja 10, eli:
  - 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

Abstrakti tietotyyppi joukko

- Usein ohjelma ylläpitää suoritusaikanaan jotain joukkoa tietoalkioita, esim. puhelinluettelo nimi-numero–pareja
- 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 viitteen taakse "satelliitti-tiedoksi"



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

- Joukon S käsittelyyn tarvitaan 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
- **min(S)**	palauttaa viitteen joukon pienimpään alkioon
- **max(S)**	palauttaa viitteen joukon suurimpaan alkioon
- **succ(S,x)**	palauttaa viitteen alkiota x seuraavaksi suurimpaan alkioon, jos x oli suurin palauttaa NIL
- **pred(S,x)**	palauttaa viitteen alkiota x seuraavaksi pienempään alkioon, jos x oli pienin palauttaa NIL

- huomaamme että kalvolla 6 mainituista puhelinluettelon operaatioista suurin osa on suoraan tarjolla abstraktin tietotyypin joukko-operaationa

- operaatio, joka tulostaa nimi-numero–parit aakkosjärjestyksessä saadaan myös helposti toteutettua 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

- Joukko-operaatioiden tehokkuus riippuen käytetystä tietorakenteesta:

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

- Lista 1 tarkoittaa järjestämätöntä linkitettyä listaa ja lista 2 järjestettyä linkitettyä listaa

Java API:n kokoelmakehys

- Java API:n kokoelmakehyksestä (component framework) löytyy useita lähes edellä määriteltyä joukkoa vastaavia valmiita tietorakenneratkaisuja

- Tutustumme näihin tarkemmin laskareiden yhteydessä. Seuraavassa kuitenkin jo mainittu yleisimpiä sekä niiden toteutuksessa käytetyt tietorakenteet
  - ArrayList taulukko
  - LinkedList linkitetty lista
  - TreeMap tasapainotettu hakupuu
  - HashMap hajautustaulu

2. Perustietorakenteet: Lista, pino ja jono

- ennen kuin menemme varsinaiseen lista-tietorakenteeseen, opiskelemme kahta joukon "erikoistapausta", pinoa ja jonoa
- pino ja jono eivät toteuta kalvon 74 joukko-operaatioita, vaan toimivat niinkuin pino ja jono toimivat normaalielämässä
- pino ja jono ovat tärkeitä rakenneosasia monissa algoritmeissa, kuten tulemme myöhemmin näkemään
- tämä luku on ylivoimaisesti kurssin helpointa asiaa

Pino

- last-in-first-out–periaate, eli toimii kuten Unicafessa lautaspino

- pino-operaatiot:

  - **push(S,x)** laittaa tietoalkion (mihin 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

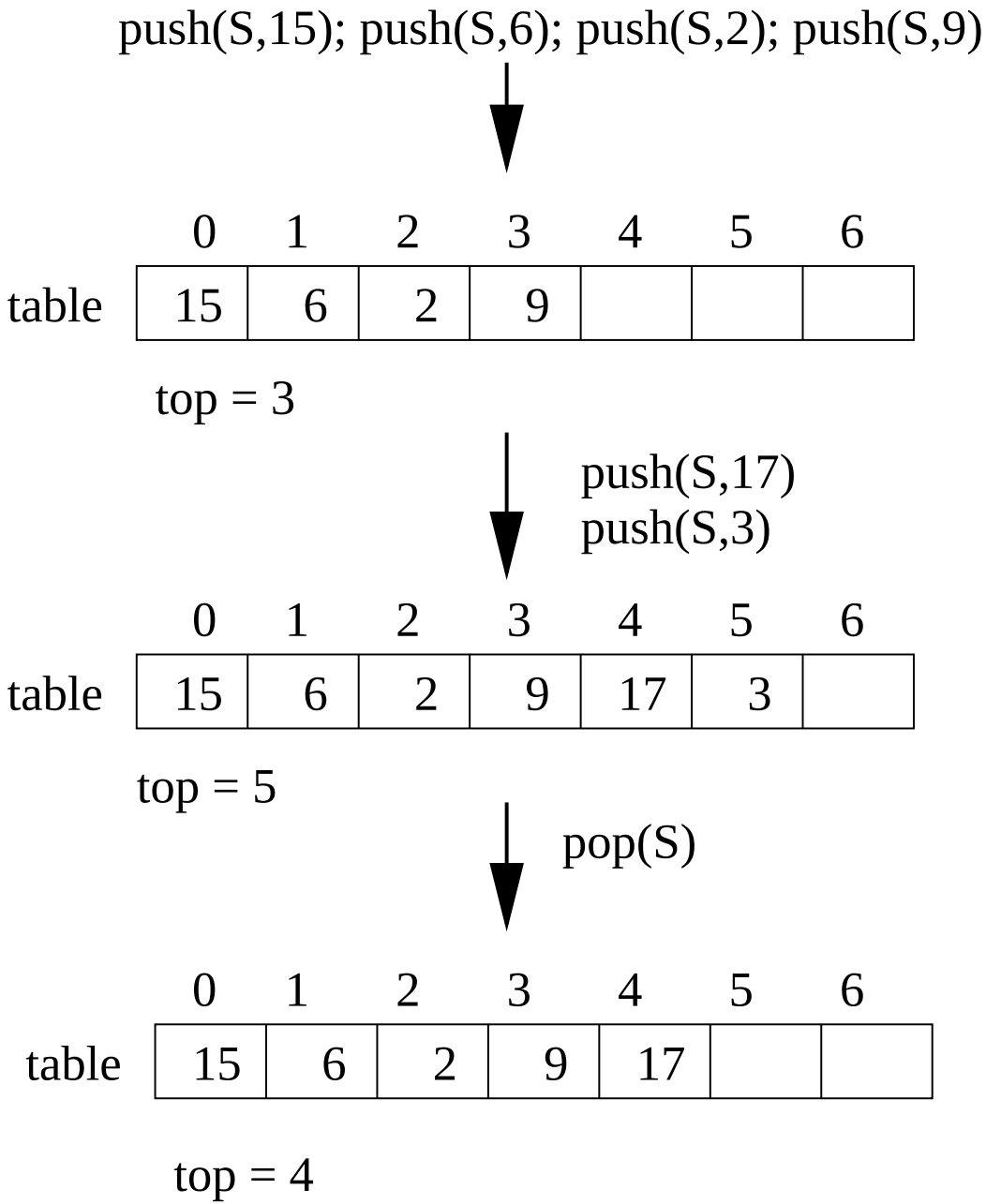
- jos yläraja pinon koolle tiedetään ennakolta, on pino helppo toteuttaa taulukon avulla
- esitetään seuraavassa pino pseudokoodin sijaan Javalla
- määritellään pino luokkana, jolla attribuuttina taulukko `table`, johon alkiot talletetaan (tällä kertaa kokonaislukuja) sekä attribuutti `top`, joka kertoo 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 popissa, josta siitäkin on tarvittaessa helppo päästä eroon

- täyteen pinoon alkion laittaminen aiheuttaa 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 ArrayListin avulla, päädyttään oleellisesti samaan ratkaisuun

- 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 pushin 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, emme käsittele aihetta enempää tällä kurssilla
- huomautuksena vielä se, että edellinen argumentaatio ei ole matemaattisesti validi tapa tehdä tasoitetun vaativuuden analyysiä, tarkoituksena olikin ainoastaan kertoa ideatasolla mistä on kysymys
- 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

- 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

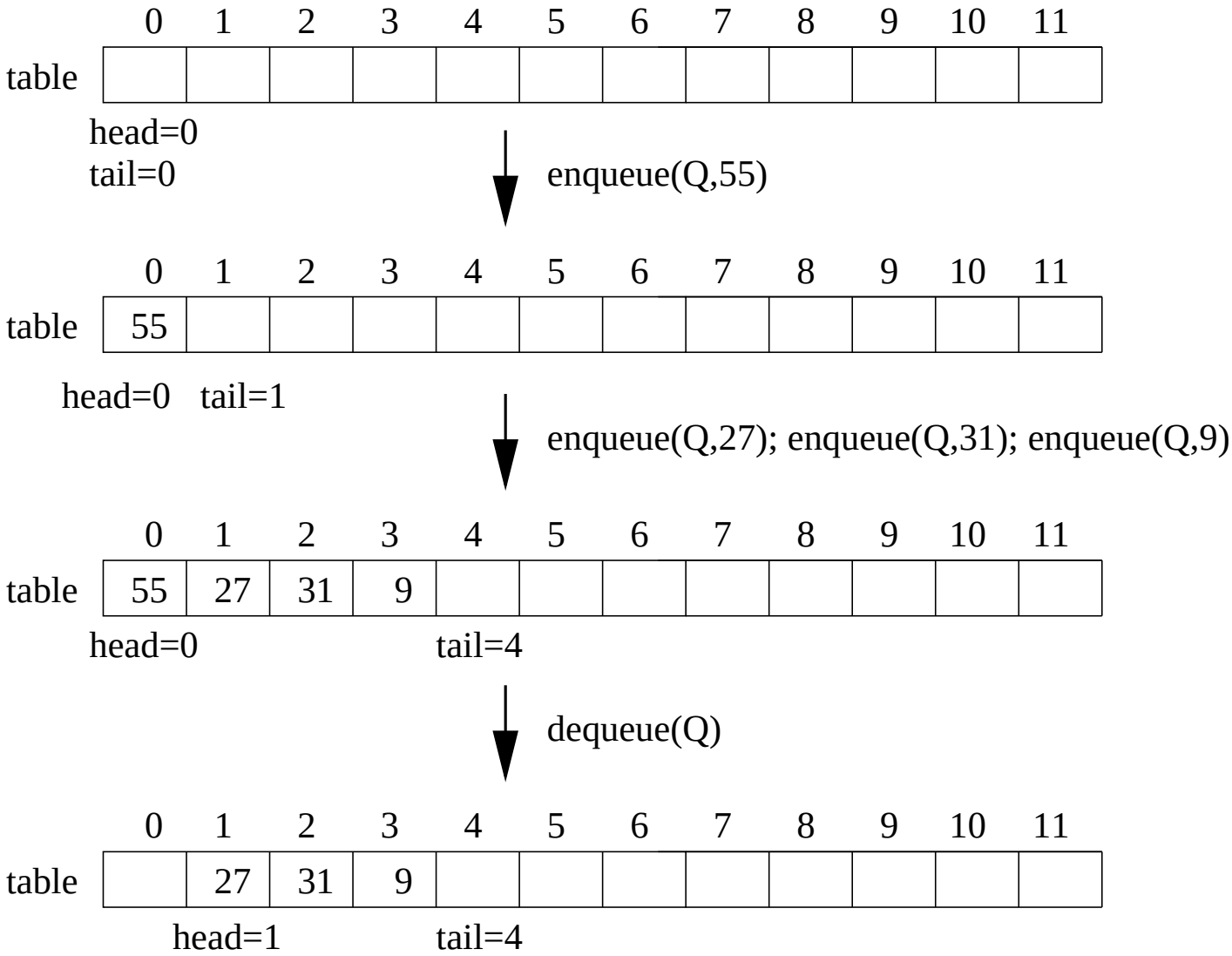
```
class Queue {
   int[] table;
   int head;
   int tail;
   int n;
```

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

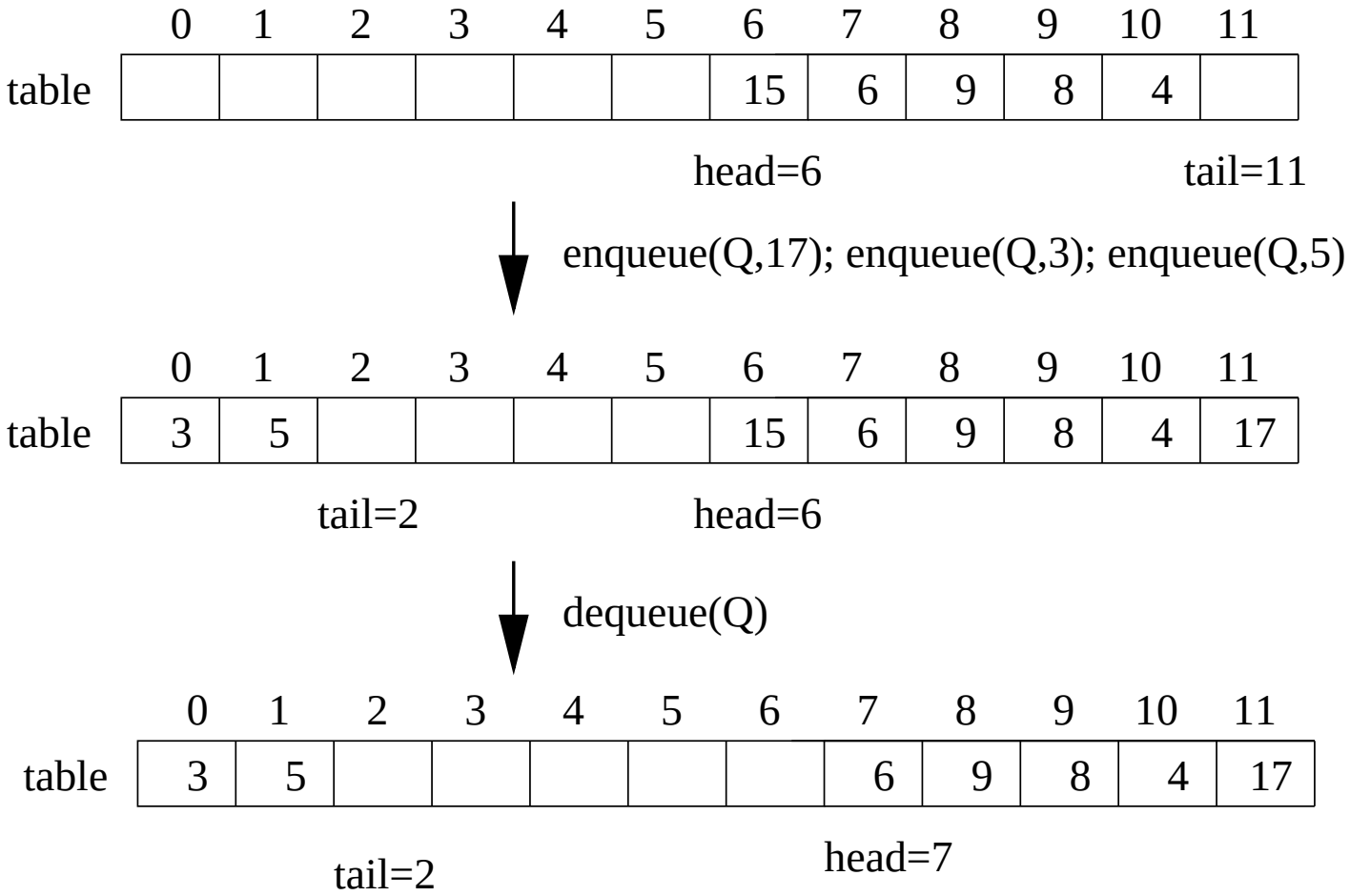
```
   Queue(int size){
      head = 0;   tail = 0;   n = size;
      table = new int[n];
   }
```

- 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` $= 3$, eli
- 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;
}
```

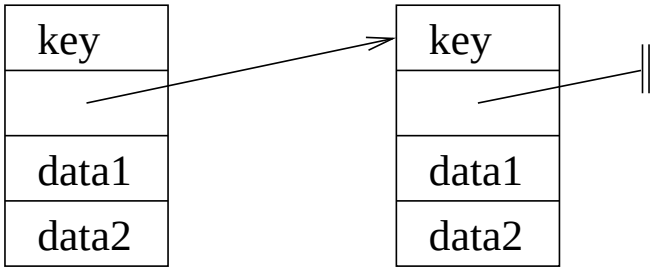
- alkiot poistetaan jonon edestä eli kohdasta `head`
- jonon etuosaa merkkaava `head` on päivitettävä uuteen arvoon, eli se joko kasvaa yhdellä tai pyörähtää ympäri, jos jono alkaa taas kerran taulukon alusta

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

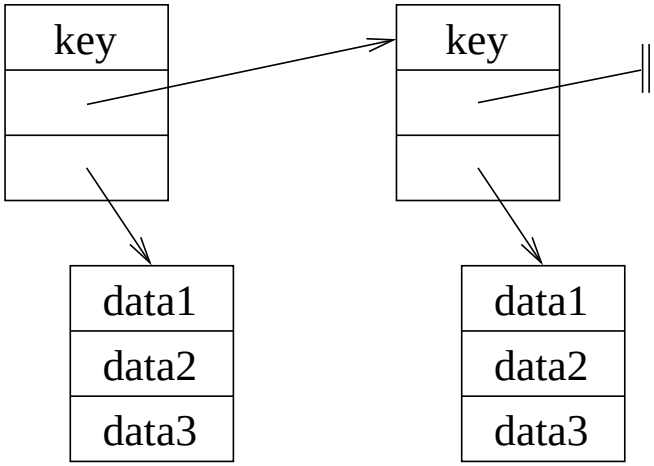
- Myös jonon tapauksessa kaikkien operaatioiden aika- ja tilavaativuus on selvästi $\mathcal{O}(1)$

Linkitetyt rakenteet

- Jos emme tiedä, mikä on pinoon tai jonoon talletettavien alkioiden maksimimäärä, on taulukkototeutus hiukan ongelmallinen
- toisaalta jos tiedämme että esim. pinossa ei voi olla koskaan enempää kuin miljoona alkiota, mutta yleensä alkiomäärä on hyvin vähäinen, joutuu taulukkoon perustuva pino varaamaan tilaa turhan paljon
- parempi ratkaisu saadaan linkitetyllä rakenteella: 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 linkkitetyn rakenteen muihin olioihin sekä viitten linkitettyä oliota vastaavaan tiedon sisältävään olioon



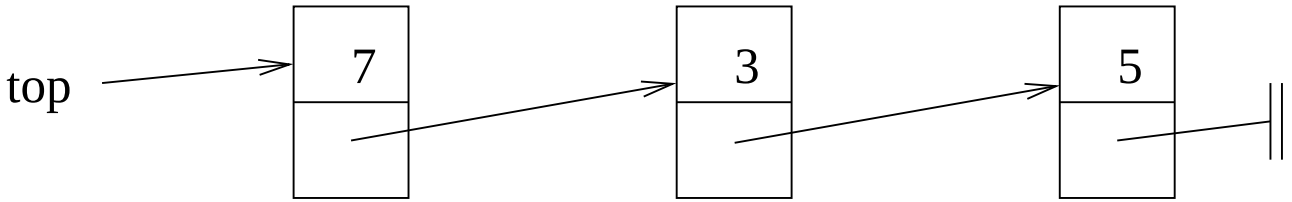
NIL eli linkki ei mihinkään merkitään



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

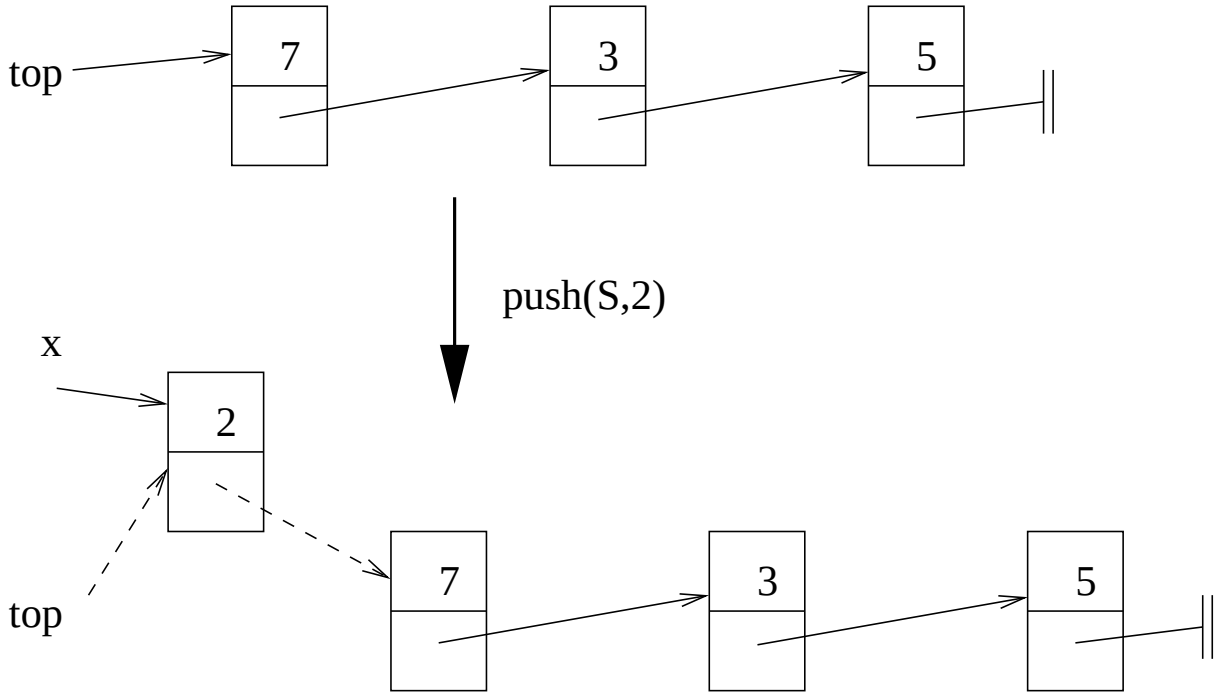
key	pinoon talletettu tietoalkio
next	viite toiseen pinosolmu-olioon

  - Pinolla (jota merkitään S) on attribuutti top joka on viite siihen *pinosolmu*-tyyppiä olevaan olioon, joka tallettaa pinon päällä olevan alkion
  - Jos $S.top = NIL$ on pino tyhjä
  - Jokaiseen pinosolmun atribuuttiin $next$ talletetaan pinossa seuraavana olevan pinosolmun viite
  - pinon pohjimmaisen pinosolmun $next$-atribuutin arvo NIL, eli viite ei mihinkään
- esim: pino missä huipulla 7, jonka alla 3 ja 5



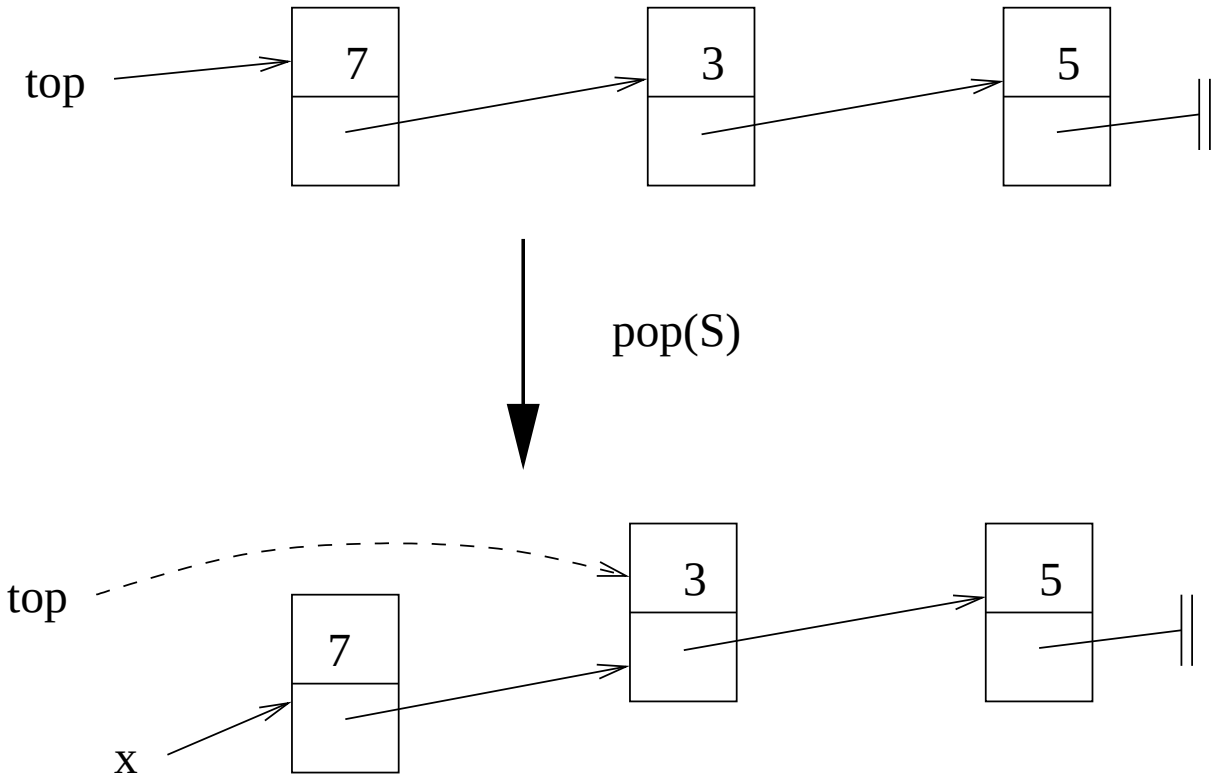
- seuraavaksi operaatioiden toteutus ja toiminta

```
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 25 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: popin 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

- Koska pinosolmuja tuskin käytetään muualla kun pinon sisällä, kannattaa ne toteuttaa sisäluokkana, eli luokkana joka määritellään luokan Pino-sisällä

- Ohjelmoinnin jatkokurssilla ei ilmeisesti sisäluokkia käytetty, mutta niissä ei ole peruskäytön kannalta mitään ihmeellistä.

- Sisäluokka kannattaa määritellä yksityiseksi, sillä kenenkään muun kun luokan Pino ei ole tarkoitus tietää sisäluokasta mitään

- Sisäluokan 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 class PinoSolmu {
      int key;
      PinoSolmu next;
      private PinoSolmu(int k, PinoSolmu seur) {
         key = k; next = seur;
      }
   }

   private PinoSolmu top;    // viite pinon päällä olevaan alkioon

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

- huom: yksityisen sisäluokan PinoSolmu ilmentymät ovat käytettävissä ja viitattavissa ainoastaan luokan Pino sisällä

- Näin saamme aikaan oikean abstraktin tietotyypin: pinon sisältöön pääsee käsiksi ainoastaan metodeilla push, pop ja empty

- operaatioiden toteutus ei juuri poikkea aiemmin esitetystä pseudokoodista

```
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
}

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:

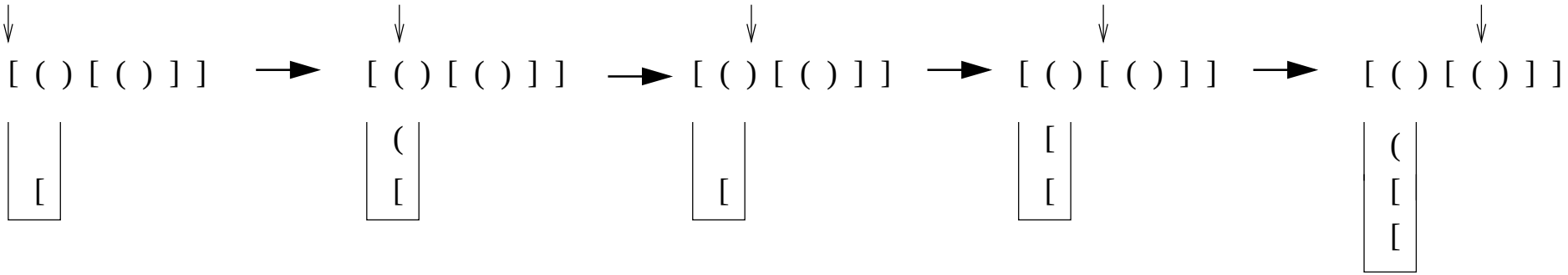
```
Pino pino = new Pino() ;
pino.push(10);
pino.push(15);
if ( !pino.empty() )
    System.out.println(" pinon huipulta: " + pino.pop());
```

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 päälimmäisen 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 = 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() ) // liikaa ( 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

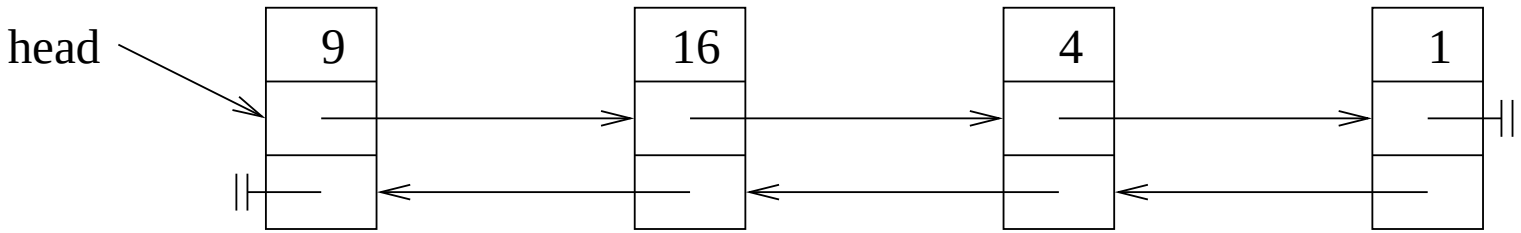
- mikä on algoritmin vaativuus?
- algoritmi käy syötteenä olevan merkkijonon läpi kertaalleen. Jokaisen merkin kohdalla tehdään muutama vertailuoperaatio sekä maksimissaan yksi pino-operaatio
- Pino-operaatioiden vaativuus on $\mathcal{O}(1)$, eli jokaisen merkin kohdalla aikaa kuluu vakiomäärä
- algoritmin 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.
- 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

Lista

- linkitetyn listan avulla on helppo toteuttaa abstrakti tietotyyppi joukko
- linkitetyistä listoista on monia eri variaatioita joista ensin tarkastelemme *kahteen suuntaan linkitettyä listaa*
- listalla oleva tieto talletetaan *listasolmu*-tyyppiä oleviin olioihin, joilla 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 missä 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

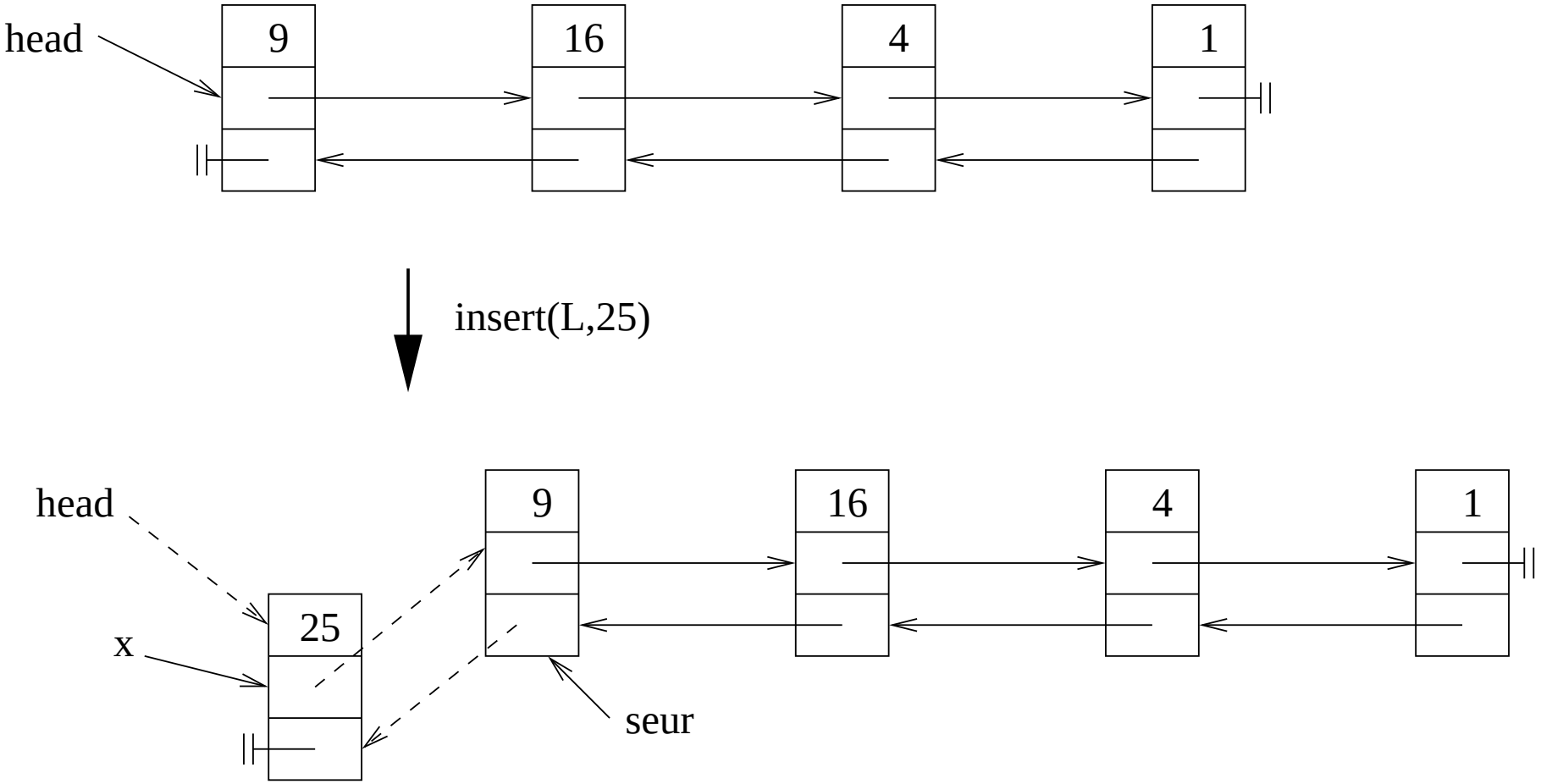
```
search(L,k)
    p = L.head
    while p ≠ NIL and k ≠ p.key      // jos ei olla lopussa tai löydetty etsittyä
        p = p.next                   // jatketaan eteenpäin
    return p
```

- p on apumuuttuja, joka on viite siihen listasolmuun missä etsintä on menossa
- p viittaa aluksi 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

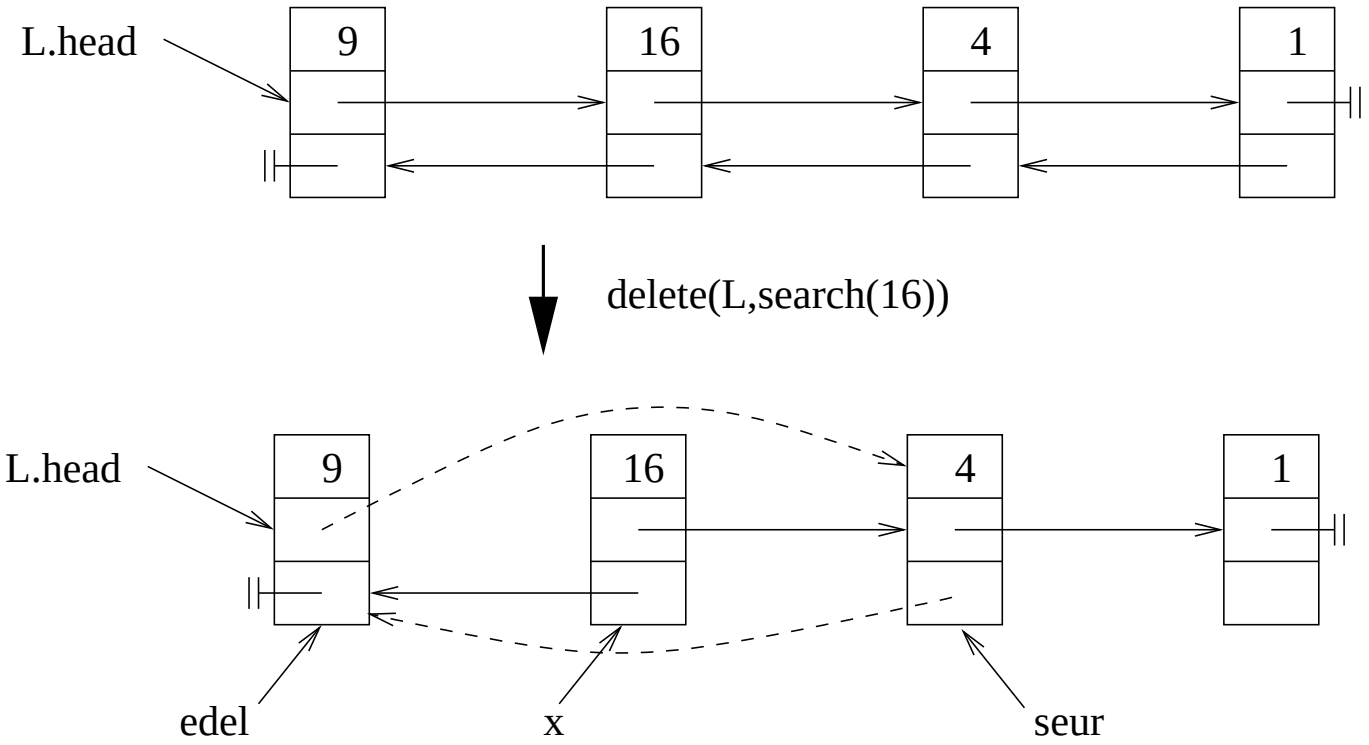
- operaatiossa manipuloidaan viitteitä siten, että x:n yli hypätään, eli x:n edellinen viittaakin suoraan x:n seuraavaan

- 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 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 75 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 linkitettynä listana toteutetun abstraktin tietotyypin joukko 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)	$\mathcal{O}(n)$

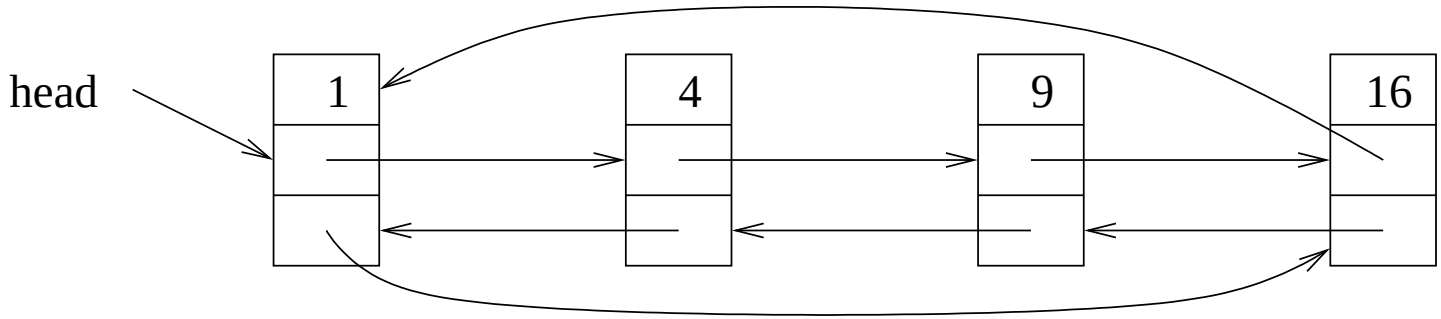
- toteutuksessamme operaatioiden vaativuudet ovat samat kuin sivun 78 taulukon lista1:llä

Rengaslista

- jos talletamme luvut listalle *suuruusjärjestyksessä*, insertin vaativuus huononee luokkaan $\mathcal{O}(n)$ mutta operaatiot min, succ sekä pienillä modifikaatioilla myös operaatiot max ja pred saadaan vakioaikaisiksi, eli päädytään sivun 78 taulukon lista2:n

- 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 operaatioiden max ja pred saamiseksi vakioaikaiseksi on käyttää *rengaslistaa*, missä alkiot on talletettu suuruusjärjestykseen



- 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 huomatan, 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, operaation toteutus jätetään laskareihin
- Kaksisuuntaisesti linkitettynä rengaslistana toteutetun abstraktin tietotyypin joukko 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)	$\mathcal{O}(1)$

- toteutuksessamme operaatioiden vaativuudet ovat samat kuin sivun 78 taulukon lista2:llä
- 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

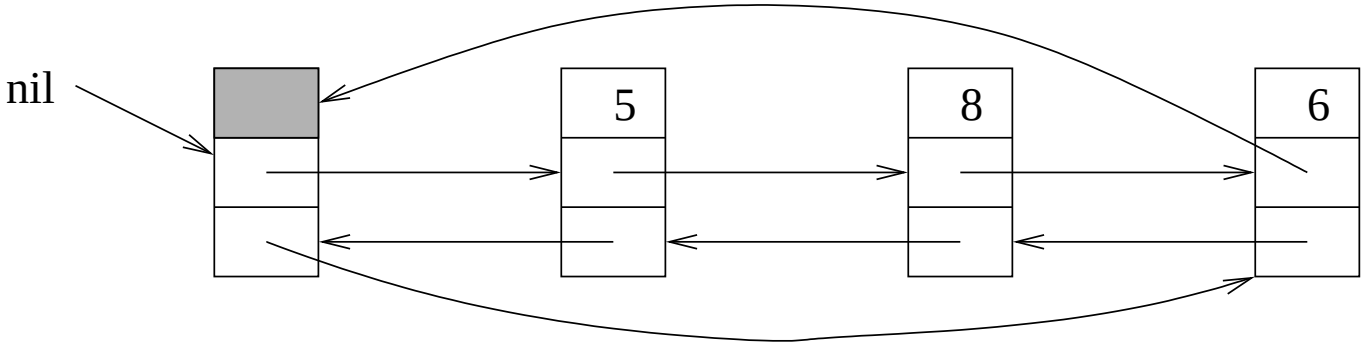
- yksi variaatio linkitetyistä listoista on *tunnussolmullinen rengaslista*
  - nyt listan alussa tunnussolmu, eli solmu missä ei säilytetä tietoa
  - listan L attribuutti L.nil viittaa listan tunnussolmuun
  - listan ensimmäinen alkio löytyy viitteen L.nil.next päästä
  - listan viimeinen alkio löytyy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

- listoista on muitakin variaatioita: tunnussolmuton rengaslista, yhteensuuntaan linkitetyt listat sekä tunnussolmulla että ilman, tavallisena tai rengaslistana . . .

- huom: 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?

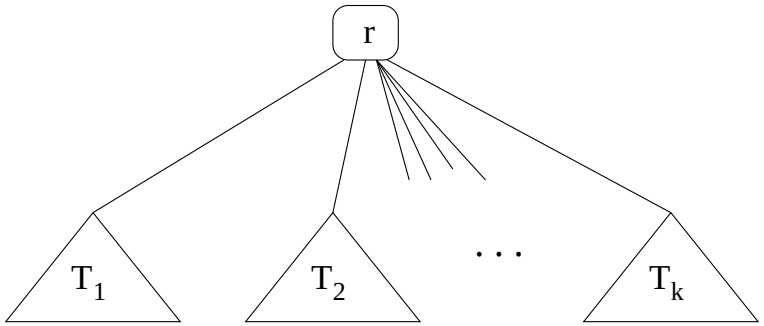
- jos käsiteltävä aineisto on pieni, riittää lista varsin hyvin

- 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

3. Hakupuut

- Hakupuu on listaa huomattavasti edistyneempi tapa toteuttaa abstrakti tietotyyppi joukko
- puurakenteelle on tietojenkäsittelyssä myös muutakin käyttöä, esim. algoritmin suoritusajan analysoinnissa, ohjelman laskennan etenemisen kuvailussa ym.
- ennen kuin menemme hakupuihin, tutustutaan yleiseen puihin liittyvään käsitteistöön
- *Puu* on kokoelma *solmuja* ja niitä yhdistäviä *kaaria*, siten että:
  - kokoelma on tyhjä, tai
  - yksi solmuista r, on *juuri* johon kaaret liittävät nolla tai useampia *alipuita* $T_1, \ldots T_k$, jotka itsekin ovat puita



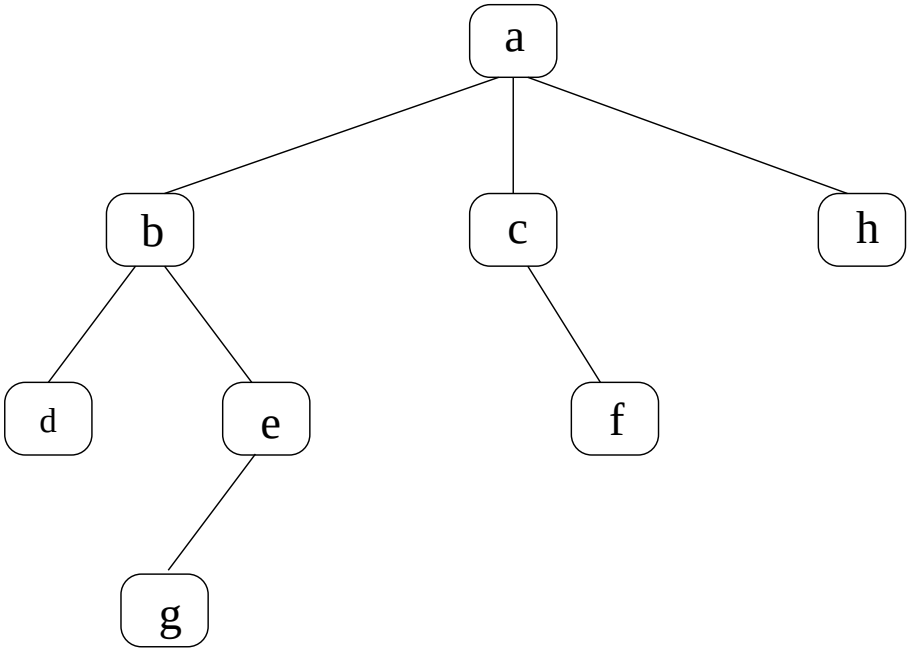
- Edellä annettu puun määritelmä on rekursiivinen, eli puu koostuu juuresta ja siihen liittyvistä alipuista, jotka ovat itsekin puita

- Puulle voidaan antaa myös ei-rekursiivinen määritelmä, jolloin puun katsotaan olevan hieman yleisemmän tietorakenteen eli verkon erikoistapaus. Palaamme verkkopohjaisen puun määritelmään myöhemmin

- Rekursiivisen määritelmän kryptisyydestä huolimatta muutaman esimerkin jälkeen kellään ei liene vaikeuksia ymmärtää mitä puu tarkoittaa

- Määritellään muutamia puihin liittyviä käsitteitä
  - Puiden $T_1, \ldots, T_k$ juuret ovat r:n *lapsia* ja r on lastensa *vanhempi* (tässä r ja T_i viittaavat edellisen kalvon määritelmään ja kuvaan)
  - jokaisella solmulla paitsi juurella on tasan yksi vanhempi
  - solmu jolla ei ole lapsia on *lehti*
  - solmut joilla on yhteinen vanhempi, ovat *sisaruksia*
  - *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 sisältävää puu, on koko puun alipuu ja b on alipuun juuri

- lisää määritelmiä:
  - *polku* 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 kalvon kuvassa a, b, e, g on polku solmusta a solmuun g jonka pituus on 3

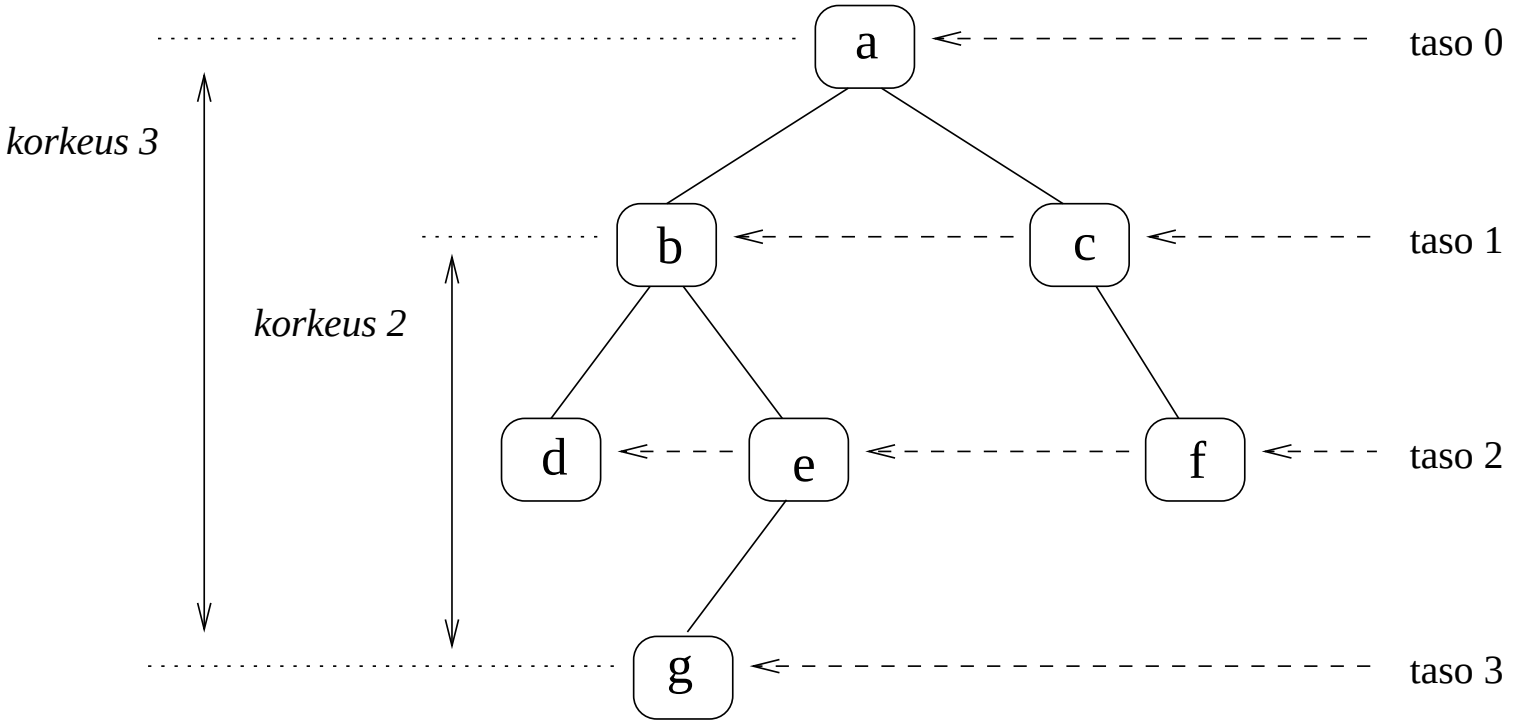
    huom: erikoistapaus, jokaisesta solmusta on nollan pituinen polku itseensä!
  - jos on olemassa polku solmusta x_1 solmuun x_2, sanotaan että x_1 on solmun x_2, *edeltäjä* ja x_2 on x_1:n *jälkeläinen*

    jos $x_1 \neq x_2$ ovat edeltäjyys ja jälkeläisyys *aitoja*

    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* on pisimmän x:stä lehteen vievän polun pituus
  - *puun korkeus* on sen juuren korkeus

- esim: kuvassa solmujen tasot sekä solmun a ja b korkeus



Binääripuu

- jos puun solmuilla on korkeintaan kaksi lasta, on kyseessä *binääripuu*
  - 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 kalvolla 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 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 $2 \times 2^n = 2^{n+1}$. Kaava siis voimassa myös tasolla $n + 1$.

Väitteen 2 todistus:

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

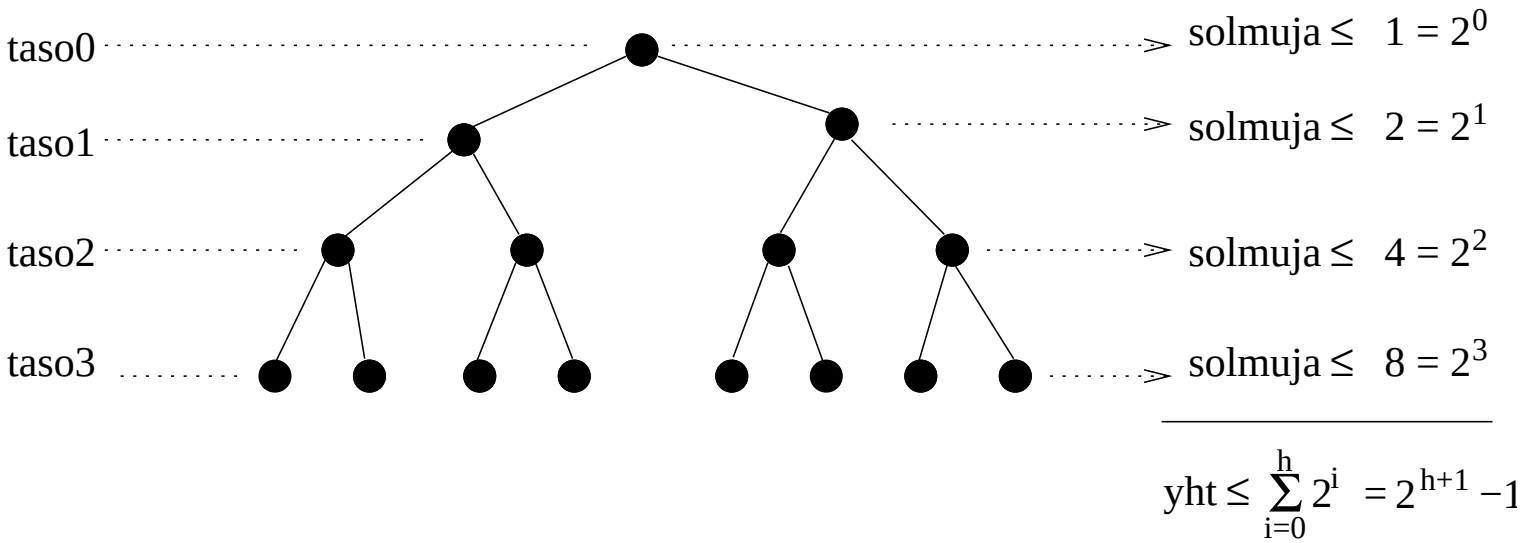
Binääripuun solmujen lukumäärä on summa eri tason solmujen lukumärästä. Edellisen kohdan perusteella tasolla i korkeintaan 2^i solmua. Eli h:n korkuisen puun maksimisolmumäärä on

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

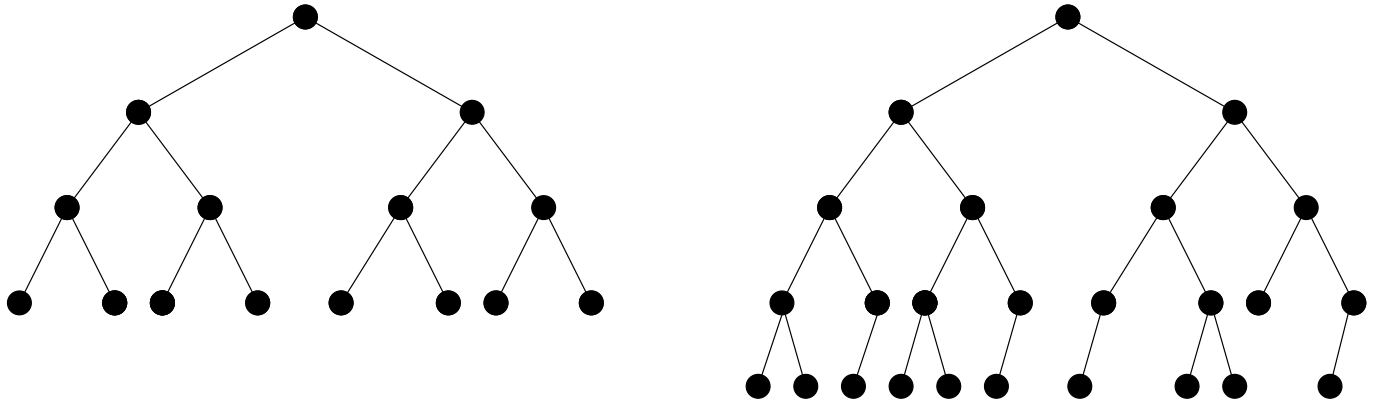
Ensimmäisen viikon toisessa laskaritehtävässä todistimme, että

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

- Edellisen kalvon todistuksien ideaa valottava kuva

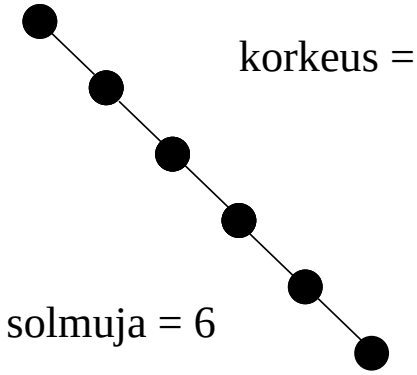


- Binääripuuta, jonka jokaisella solmulla on joko 0 tai 2 lasta, ja jonka jokainen lehti on samalla tasolla, sanotaan täydelliseksi binääripuuksi
- Kuvassa vasemmalla täydellinen binääripuu, eli puu, joka on on "täysin mahdollinen" tietyn korkuinen puu
- Jos puu on viimeistä tasoa lukuunnottamatta täydellisen binääripuun kaltainen, voidaan puuta nimittää melkein täydelliseksi
- kuvassa oikealla melkein täydellinen binääripuu



- lausetta 1 soveltamalla on helppo huomata että h:n korkuisella täydellisellä binääripuulla on täsmälleen 2^h lasta 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 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 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

$$\log_2 (n + 1) - 1 \leq \text{puun korkeus} \leq n - 1$$

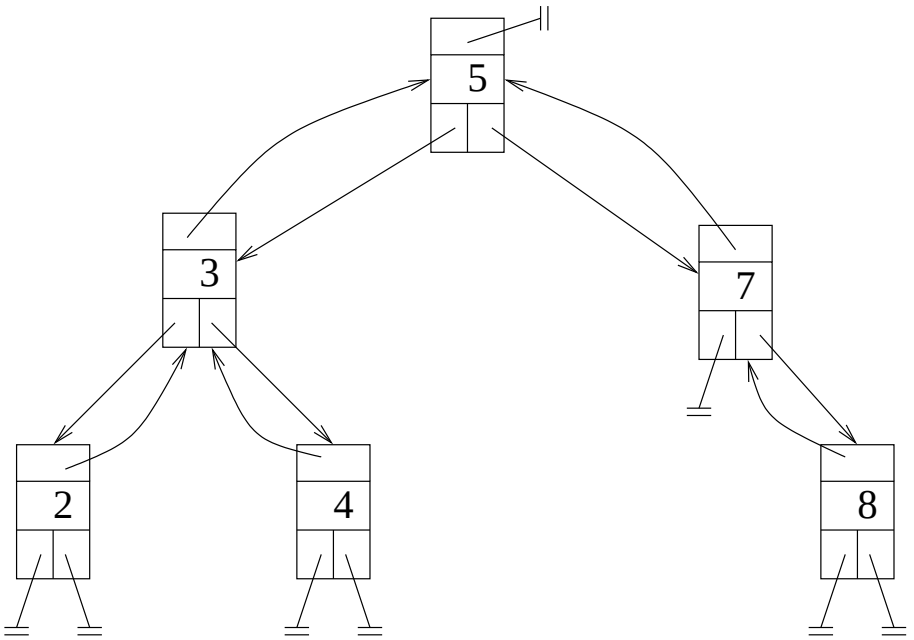
Binäärihakupuu

- toteutetaan kalvon 75 abstrakti tietotyyppi joukko siten että joukossa olevat alkiot talletetaan binääripuun solmuihin
- rajoitumme jälleen yksinkertaistettuun tapaukseen missä talletettavat tietoalkiot sisältävät ainoastaan avaimen
- puu rakentuu puusolmu-olioista, joilla seuraavat kentät:

key	talletettu tietoalkio
left	viite vasempaan lapseen
right	viite oikeaan lapseen
parent	viite vanhempaan

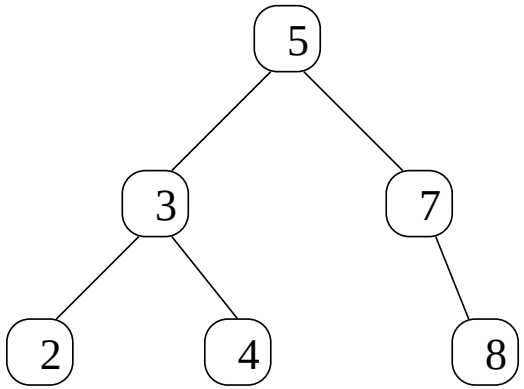
- puulla T attribuutti $T.root$ joka osoittaa juurisolmuun
- jos solmulla x ei ole vasenta lasta, $x.left = NIL$, vastaavasti oikealle lapselle
- puun juurisolmulla parent-kentän arvo NIL

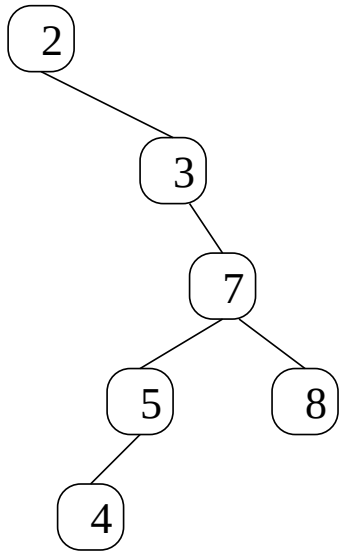
- Binäärihakupuussa avaimet toteuttavat **binäärihakupuuehdon**:
  - jos solmu v on solmun x vasemmassa alipuussa, niin v.key $<$ x.key
  - jos solmu o on solmun x oikeassa alipuussa, niin x.key $<$ o.key
- eli **solmun vasemmassa alipuussa on ainoastaan sitä pienempiä ja oikeassa alipuussa sitä isompia avaimia**
- esim: binäärihakupuu missä alkiot 2, 3, 4, 5 ,7 ja 8



- yleensä piirrämme puut ilman linkkikenttien eksplisiittistä sisältöä

- samansisältöiset, mutta mutta muodoltaan erilaiset binäärihakupuut:





- kun monisteen tässä luvussa puhutaan puusta, tarkoitetaan jatkossa nimenomaan binäärihakupuuta

- puun T alkiot voidaan tulostaa suuruusjärjestyksessä kutsumalla seuraavaksi esitettävää rekursiivista algoritmia parametrilla $T.root$

```
inorder-tree-walk(x)
    if x ≠ NIL
        inorder-tree-walk(x.left)
        print x.key
        inorder-tree-walk(x.right)
```

- algoritmin toiminnan eteneminen syötteenä kuvan vasen puu

```
inorder-tree-walk(5)
    inorder-tree-walk(3)
        inorder-tree-walk(2)
            inorder-tree-walk(NIL)
            print(2)                    2
            inorder-tree-walk(NIL)
        print(3)                        3
        inorder-tree-walk(4)
            inorder-tree-walk(NIL)
            print(4)                    4
            inorder-tree-walk(NIL)
    print(5)                            5
    inorder-tree-walk(7)
        inorder-tree-walk(NIL)
        print(7)                        7
        inorder-tree-walk(8)
            inorder-tree-walk(NIL)
            print(8)                    8
            inorder-tree-walk(NIL)
```

edellä esim. kutsu inorder-tree-walk(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 jokaisessa solmussa käydään tasan kolme kertaa: tultaessa vanhemmasta rekursiokutsulla, palattaessa vasemman alipuun rekursiosta ja palatessa oikean alipuun rekursiosta

- algoritmin tilavaativuus
  - algoritmin edetessä rekursiopinoon on talletuttuna 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 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

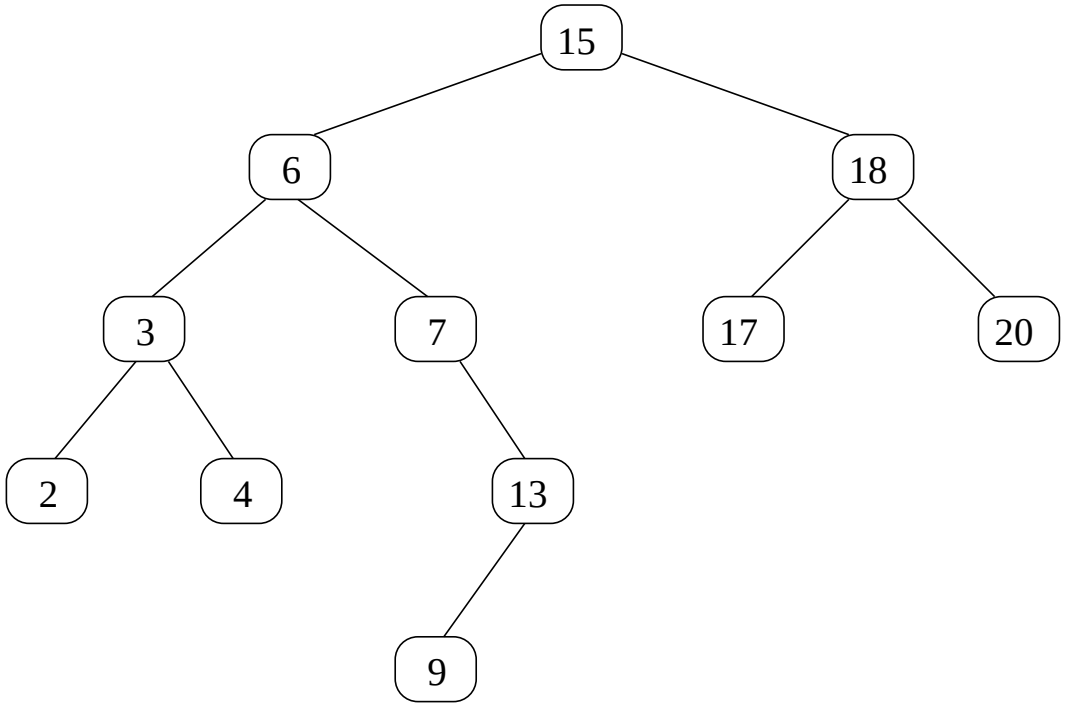
Joukko-operaatioiden toteuttaminen

- 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.
  - Vastaavaasti, jos etsittävä k on suurempi kuin x.key, jatkuu etsintä rekursiivisesti oikeaan alipuuhun.
  - Jos etsittävä solmu on puussa, päättyy algoritmin suoritus siihen, että saavutaan etsittävän solmun kohdalle
  - 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 $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

- search(T.root, 10) etenee $15 \rightarrow 6 \rightarrow 7 \rightarrow 13 \rightarrow 9 \rightarrow NIL$

  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 2 perusteella puun korkeus pahimmassa tapauksessa $n-1$, missä n solmujen lukumäärä, eli searchin 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 $\mathcal{O}(1)$

- kalvon 145 esimerkissä minimin haku etenee polkua $15 \rightarrow 6 \rightarrow 3 \rightarrow 2$ ja maksimin polkua $15 \rightarrow 18 \rightarrow 20$

- annettua alkiota seuraavaksi suurimman etsintä, eli operaation succ toteuttaminen on hiukan hankalampaa

- kalvon 145 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 asken 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 kalvon kuvat liittyvät perusteluun

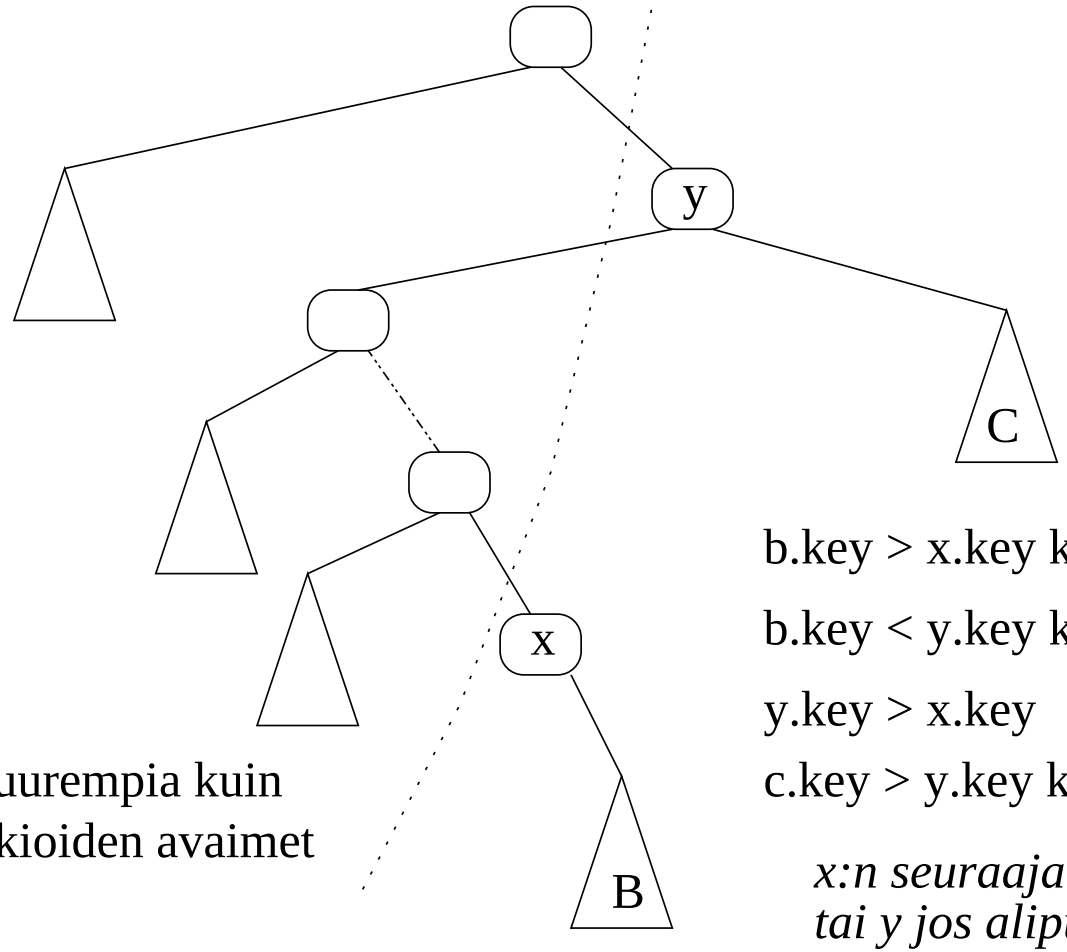
- Jos tämä perustelu (joka on matemaattisen oikeellisuustodistuksen "kansanomaistettu" versio) tuntuu sekavalta, hyppää sen yli suosiolla

- 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 alipuunssa 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 kalvolla 150) tai y:tä suurempia (kuva kalvolla 151), ne eivät siis voi tulla kyseeseen x:n seuraajina
  - y:n vasemmassa alipuussa olevat solmut jotka jäävät x:n "vasemmalle puolele" 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 kalvon 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

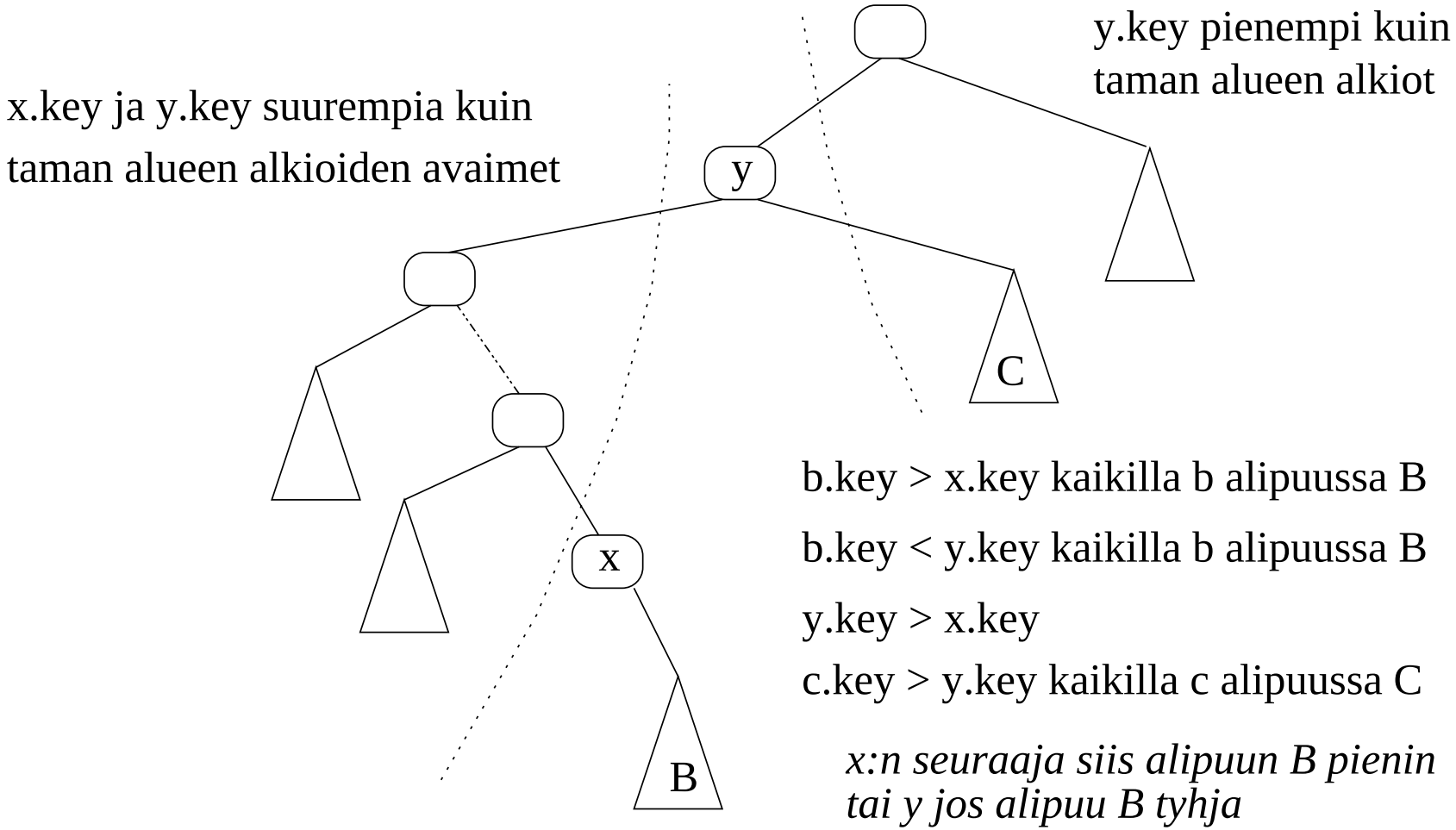
y.key > x.key

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 $\mathcal{O}(1)$
- operaatio pred on symmetrinen succ-operaation kanssa, eli vaatii pahimmassa tapauksessa ajan $\mathcal{O}(h)$ ja tilan $\mathcal{O}(1)$

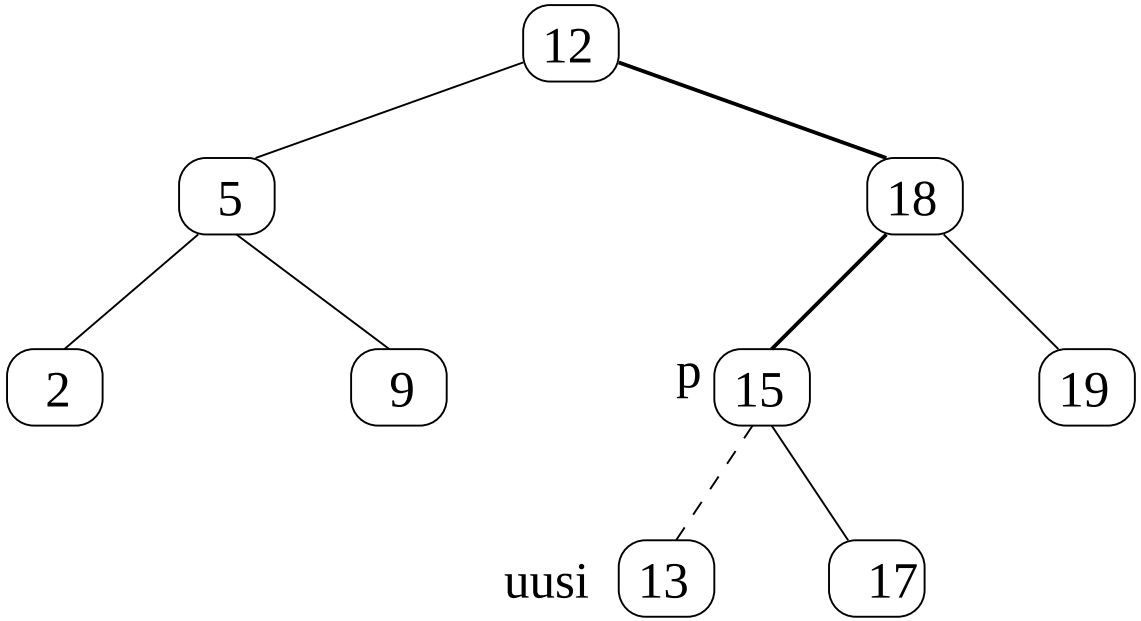
- alkion lisääminen binäärihakupuuhun käy melko helposti

```
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   p = NIL
9   while x ≠ NIL             // etsitään kohta, johon uusi alkio kuuluu
10      p = x
11      if k < x.key
12          x = x.left
13      else x = x.right
14  uusi.parent = p           // viitteet uuden alkion ja sen vanhemman välille
15  if uusi.key < p.key
16      p.left = uusi
17  else p.right = uusi
```

- algoritmin toimintaperiaate
  - riveillä 4-6 käsitellään erikoistapaus, jossa lisättävä on puun ensimmäinen solmu
  - rivien 9-13 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 15-17 uusi alkio laitetaan joko p:n vasemmaksi tai oikeaksi lapseksi riippuen onko uuden solmun avain pienempi vai suurempi kuin p:n avain

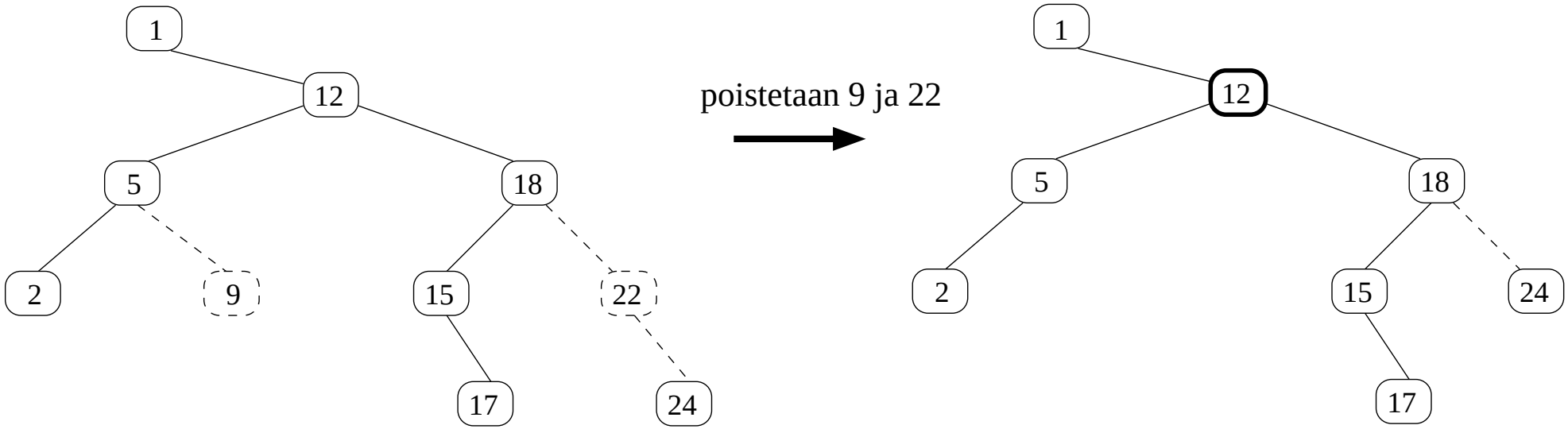
- puu mihin lisätty avain 13, polku juuresta solmuun jonka lapseksi uusi solmu lisätään on tummennettu



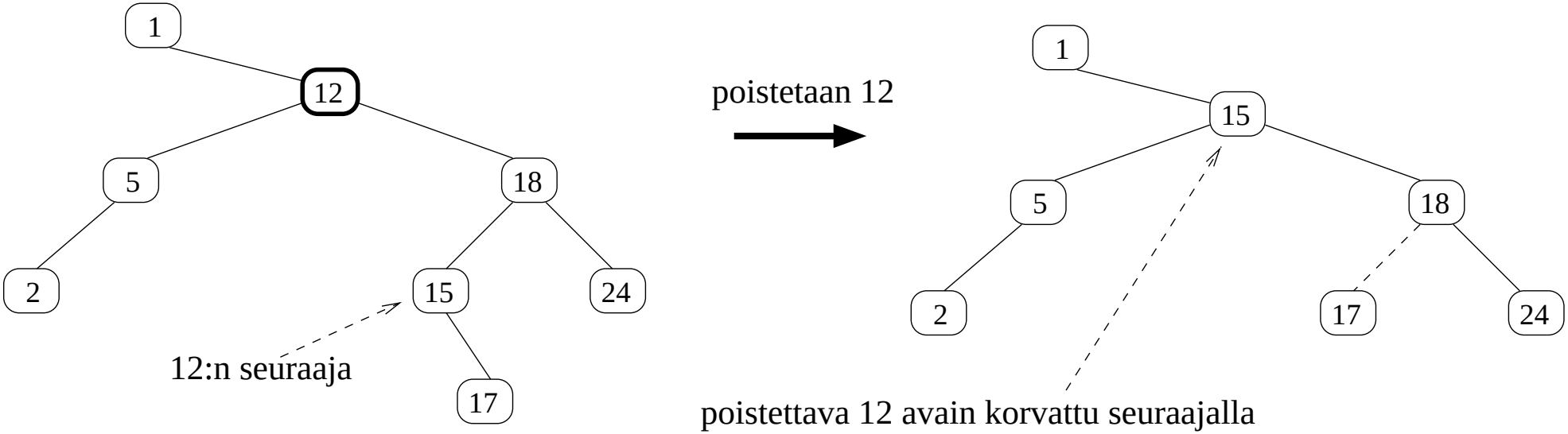
- kuten muillakin tähän asti kohtaamillamme operaatioilla, on lisäyksenkin aikavaativuus $\mathcal{O}(h)$ puun korkeuden h suhteen, sillä pahimmassa tapauksessa lisäys tehdään alimmalla tasolla olevan solmun lapseksi

- tilavaativuus on $\mathcal{O}(1)$, sillä rekursio ei ole käytössä

- 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, on sen seuraajasolmu $seur$ poistettavan oikean alipuun solmuista pienin
  - poistetaan solmu $seur$, se onnistuu helposti sillä solmulla on korkeantaan yksi lapsi
- 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
19      if pois == vanh.left
20          vanh.left = lapsi
21      else vanh.right = lapsi
22      return pois
```

- ja vielä kolmas tapaus

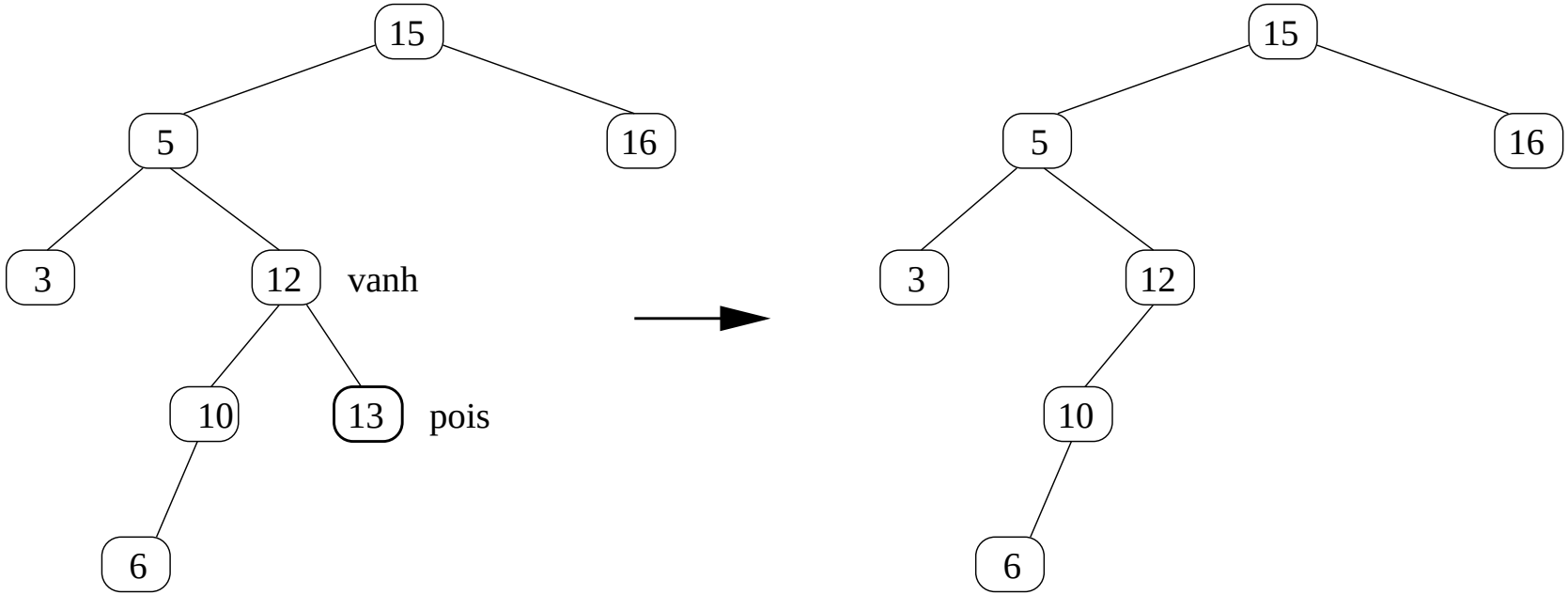
```
delete(T, pois) jatkuu ...
// tapaus 3: poistettavalla kaksi lasta
23  seur = min(pois.right)
24  pois.key = seur.key          // korvataan poistettavan avain seuraajan avaimella
25  lapsi = seur.right
26  vanh = seur.parent
    // korvataan solmu seur sen lapsella
27  if vanh.left == seur
28      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

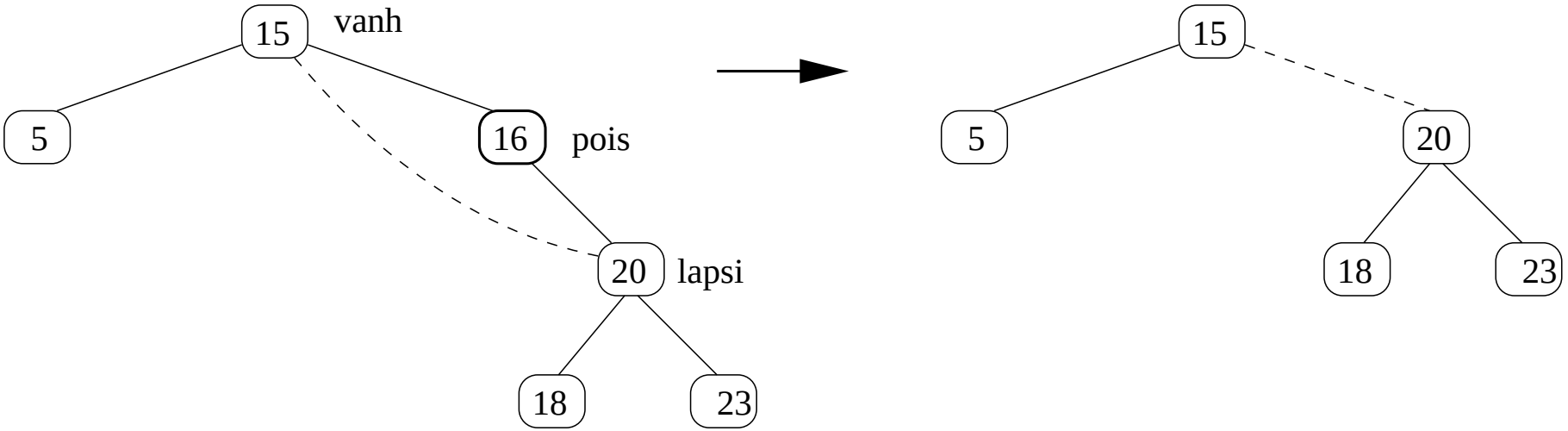
- algoritmi näyttää huomattavasti monimutkaisemmalta mitä se itseasiassa on

- Cormenissa esitetty versio on hieman lyhempi, mutta siinä kolmea tapausta ei ole koodissa eritelty samaan tapaan kuin tässä esitetyssä

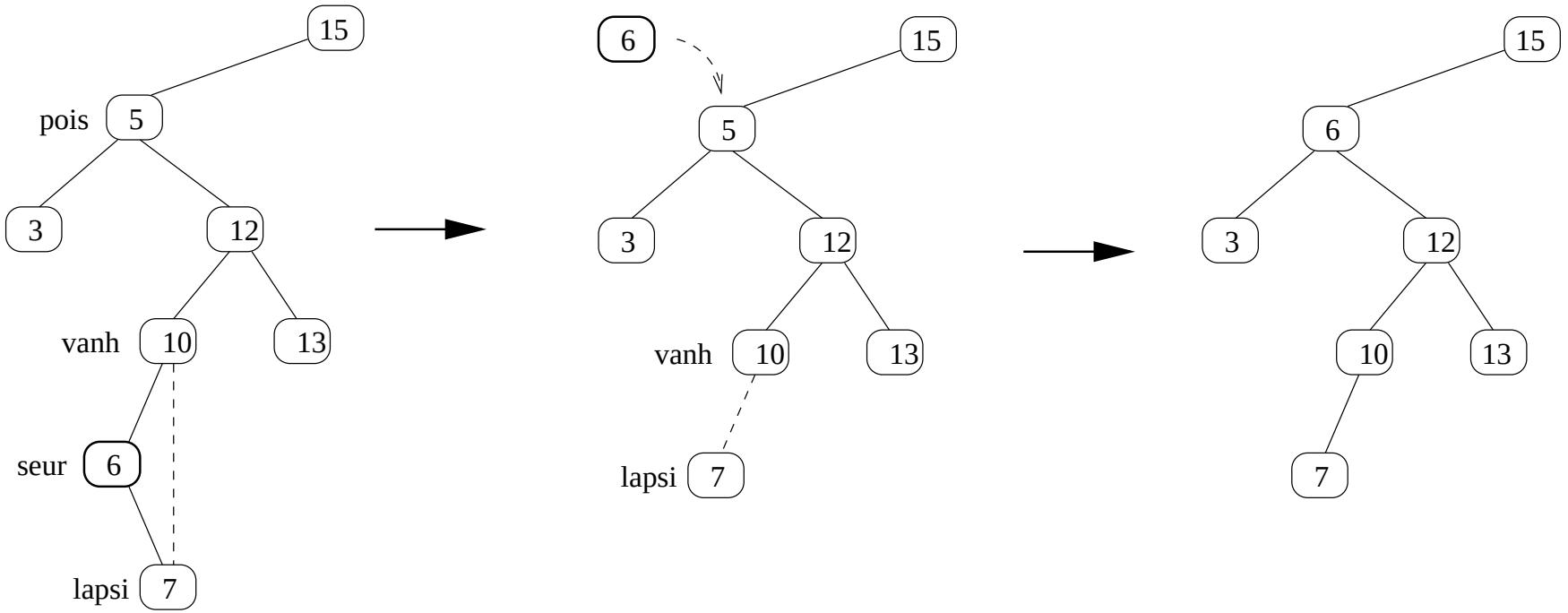
- käydään algoritmin toiminta läpi vielä tapaus tapaukselta
- Tapaus 1: *poistettavalla ei lapsia*, rivit 1-9
  - poistetaan solmu $pois$
  - riveilla 3-5 huomioidaan erikoistapaus missä poistettava on puun ainoa solmu. Tässä tapauksessa puun juureksi asetetaan NIL
  - riveillä 6-8 asetetaan poistuneen 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 18-20 laitetaan lapsi poistuneen tilalle



- Tapaus 3: *poistettavalla kaksi lasta*, rivit 23-31
  - aluksi vaihdetaan solmun $pois$ *avain* sen seuraajan $seur$ avaimeen seuraaja
  - 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 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 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"

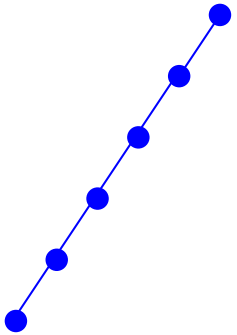
- Tutustumme ensin AVL-puihin. Myöhemmin kurssilla vuorossa B-puut

- Hakunopeuden kannalta paras tilanne on täydellinen puu. 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

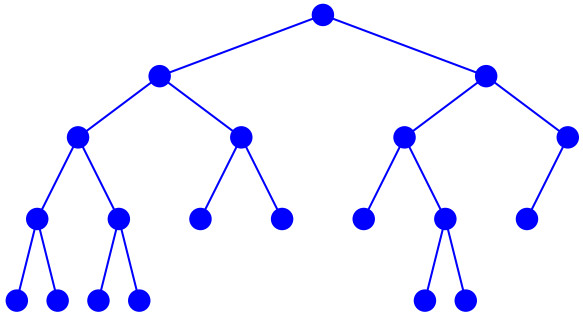
- Pahin tilanne on lineaarinen puu, jossa kaikki oikeat tai kaikki vasemmat alipuut ovat tyhjiä, puun korkeus on tällöin $n-1$

- Intuitiivisesti puu on "tasapainoinen", kun se muistuttaa muodoltaan enemmän täydellistä kuin lineaarista puuta

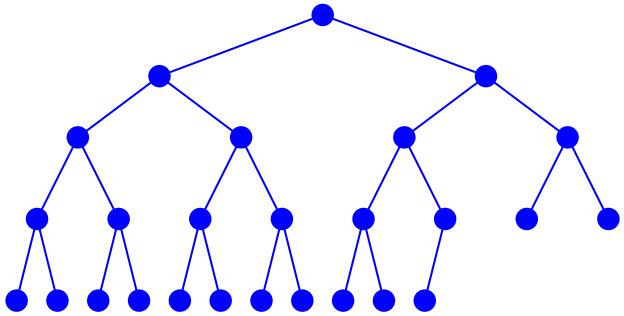
- Eri tavat määritellä tasapainoisuus täsmällisesti johtavat hieman erilaisiin tietorakenteisiin (esim. AVL-puu tai punamustapuu)



lineaarinen



tasapainoinen?



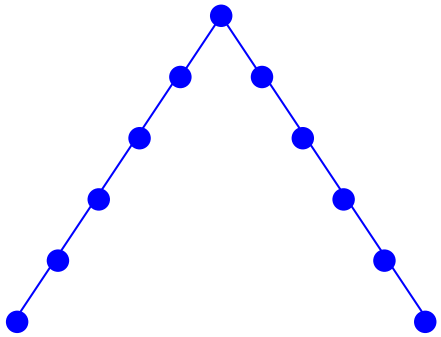
melkein täydellinen

- 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, voidaan puun korkeus laskea 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:

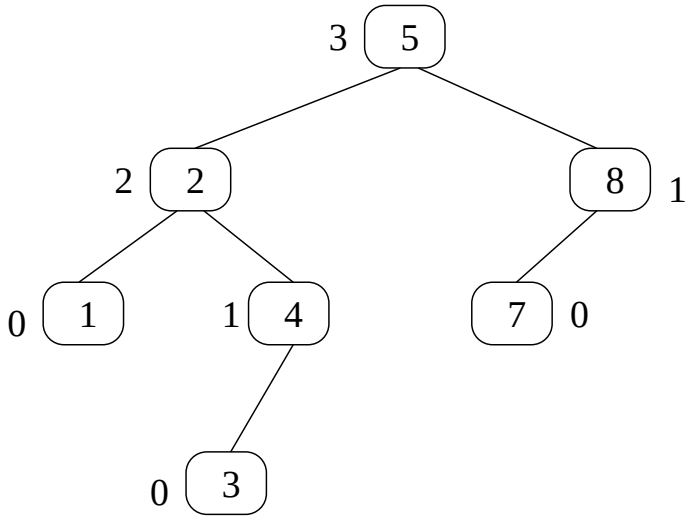
  minkä tahansa solmun vasemman ja oikean alipuun korkeuksien erotus on joko -1, 0 tai 1.

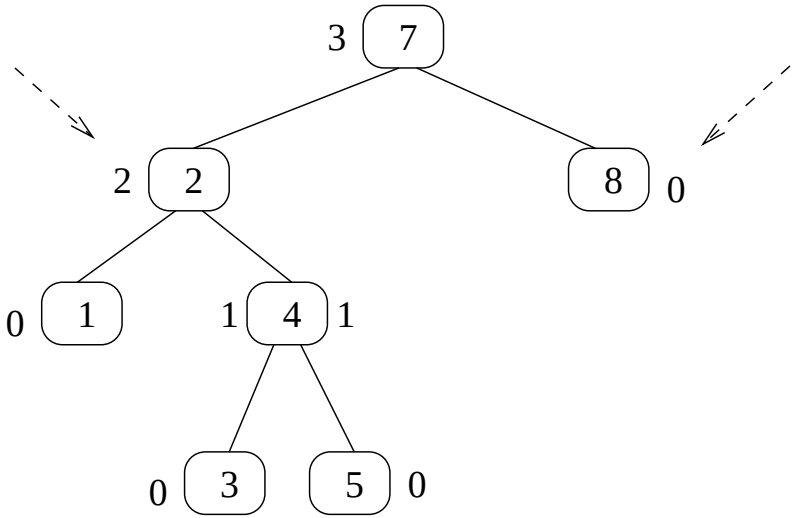
  Toisin sanoen vaaditaan

  $$|\,\textbf{Height(x.left)} - \textbf{Height(x.right)}\,| \leq 1 \qquad \textbf{kaikilla solmuilla x}$$

- 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 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

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{kun } i \geq 2 \end{aligned}$$

- Tarkastellaan lukusarjaa jonka alussa on 0 ja sen jälkeen 2:n potensseja aina kaksi kappaletta peräkkäin kasvavassa järjestyksessä, eli lukusarja on $0, 2^0, 2^0, 2^1, 2^1, 2^2, 2^2, 2^3, 2^3, \ldots$, joka siis on $0, 1, 1, 2, 2, 4, 4, 8, 8, \ldots$

  Jos aloitetaan sarjan lukujen numerointi nollasta, lukusarjan i:s luku on $2^{\frac{i+1}{2}-1}$, esim. viides $2^{\frac{5+1}{2}-1} = 2^3$, myös kuudes on $2^{\frac{6+1}{2}-1} = 2^3$ sillä oletetaan että jakolaskun tulos on aina sen kokonaislukuosa.

- On ilmeistä, että määritelty lukusarja kasvaa Fibonaccin lukuja hitaammin, eli $F_i \geq 2^{\frac{i+1}{2}-1}$. Syy tälle on että lukusarjat alkavat samasta, ja
  - määritellyn lukusarjan i:s luku on suuruudeltaan 2 kertaa lukusarjan $i-2$:s luku
  - Fibonaccin i:s luku taas on vähintään kaksi kertaa Fibonaccin $i-2$:s luku.

- Merkitään AVL-puun korkeutta h:lla ja solmumäärää n:llä.
- Lauseen 3 mukaan siis $n \geq F_{h+3} - 1$. Edellisen kalvon lopussa perusteltiin, että $F_i \geq 2^{\frac{i+1}{2}-1}$, eli $F_{h+3} \geq 2^{\frac{h+4}{2}-1}$
- Eli saadaan puun solmumäärälle n alaraja korkeuden h suhteen

$$n \geq 2^{\frac{h+4}{2}-1}$$

  solmumäärä siis kasvaa eksponentiaalisesti puun korkeuden kasvaessa
- Selvitetään vielä mikä on korkeuden yläraja solmumäärän suhteen.
- Otetaan edellisestä 2-kantainen logaritmi puolittain, ja tuloksena on

$$\log_2 n \geq \frac{h+4}{2} - 1$$

- Kirjoitetaan miellyttävämpään muotoon

$$h \leq 2 \log_2 n - 2$$

- AVL-puun korkeus h siis on pahimmassakin tapauksessa logaritminen puun solmumäärän n suhteen.
- Eli vaikka puussa on hyvin suuri määrä solmuja, ei puu kasva kovin korkeaksi, esim. $2 \log_2 1000000 - 2 \approx 38$, eli miljoonasolmuisen puun korkeus ei ole ainakaan enempää kuin 38.

- Päättelimme äsken, että jos AVL-puussa on n kappaletta solmuja on puun korkeus h korkeintaan $2\log_2 n - 2$

- AVL-puun korkeudelle ylärajalle on mahdollista laskea tarkempikin arvo

- Seuraavassa tehtävä tarkempi laskelma on siis vaihtoehtoinen tapa osoittaa, että AVL-puun korkeus on pahimmassakin tapauksessa logaritminen solmumäärän suhteen

- Jos et ole kiinnostunut tästä tarkemmasta laskelmasta, voit hypätä suoraan noin kalvon 177 puoleen väliin

- Lause 3 siis sanoo että jos AVL-puun korkeus on h, niin siinä on ainakin $F_{h+3} - 1$ solmua, missä F_i on i:s Fibonaccin luku

- Rekursiivisen määritelmän lisäksi Fibonaccin luvuille tunnetaan myös eksplisiittinen kaava

$$F_i = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right)$$

- Lasketaan korkeuden ylärajalle tarkempi arvio tähän kaavaan perustuen

- Olkoon h AVL-puun korkeus ja merkitään n:llä sen solmujen määrää
- Lauseen 3 mukaan siis

$$n \geq F_{h+3} - 1 = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right) - 1$$

- Koska toinen potenssiin korotettava luku $(1-\sqrt{5})/2 \approx -0{,}618$ on itseisarvoltaan ykköstä pienempi, on $((1-\sqrt{5})/2)^{h+3} < 1$, eli voidaan kirjottaa

$$n \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \right) - 1 = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \frac{1+\sqrt{5}}{\sqrt{5}}$$

- Muutetaan edellinen vielä hieman havainnollisempaan muotoon laskemalla likiarvot neliöjuurille ja arvioimalla hieman alaspäin:

$$n \geq \frac{1.618^{h+3}}{2.236} - 1.447$$

- Solmujen lukumäärä n kasvaa siis eksponentiaalisesti korkeuden h funktiona
- Entä mikä on korkeuden yläraja tietyllä solmumäärällä?

- siirrellään termejä

$$n + \frac{1+\sqrt{5}}{\sqrt{5}} \geq \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3}$$

- kerrotaan puolittain $\sqrt{5}$:llä

$$\sqrt{5}n + 1 + \sqrt{5} \geq \left(\frac{1+\sqrt{5}}{2} \right)^{h+3}$$

- Otetaan puolittain $\frac{1+\sqrt{5}}{2}$ kantainen logaritmi

$$\log_{\frac{1+\sqrt{5}}{2}} (\sqrt{5}n + 1 + \sqrt{5}) \geq h + 3$$

- Koska $\sqrt{5} > 1$, voidaan vasenta puolta arvioida ylöspäin (korvaamalla ykkönen $\sqrt{5}$:llä)

$$\log_{\frac{1+\sqrt{5}}{2}} (\sqrt{5}(n+2)) \geq h + 3$$

- Siirretään 3 toiselle puolelle ja hajoitetaan vasemman puolen tulon logaritmi logaritmien summaksi:

$$\log_{\frac{1+\sqrt{5}}{2}} (n+2) + \log_{\frac{1+\sqrt{5}}{2}} \sqrt{5} - 3 \geq h$$

- Eli olemme edenneet seuraavaan tilanteeseen

$$h \leq \log_{\frac{1+\sqrt{5}}{2}}(n+2) + \log_{\frac{1+\sqrt{5}}{2}}\sqrt{5} - 3$$

- Muutetaan logaritmit 2-kantaiseksi

$$h \leq \frac{1}{\log_2 \frac{1+\sqrt{5}}{2}} \log_2(n+2) + \frac{\log_2 \sqrt{5}}{\log_2 \frac{1+\sqrt{5}}{2}} - 3$$

- Kaava näyttää ikävältä, mutta lasketaan logaritmeille laskimella likiarvot ja arvioimalla hiukan ylös, päästään mukavampaan muotoon

$$h \leq 1,44 \log_2(n+2) - 1.328$$

- Puun korkeus h solmujen lukumäärän n suhteen on siis logaritminen, eli $\mathcal{O}(\log n)$

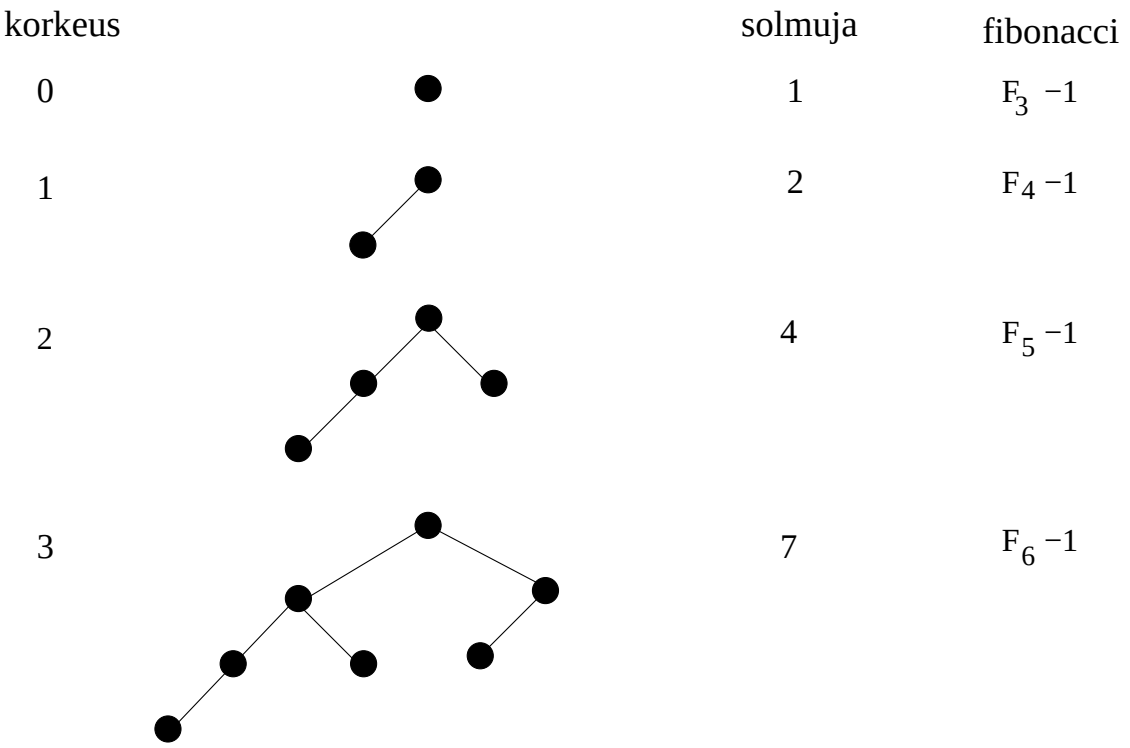
- AVL-puun korkeus kasvaa todella hitaasti, esim:
  - jos solmujen määrä $n = 100$ on korkeus $h \leq 8$
  - jos solmujen määrä $n = 1000000$ on korkeus $h \leq 28$
  - jos solmujen määrä $n = 1000000000$ on korkeus $h \leq 42$

- **Lauseen 3 todistus:**

  On siis näytettävä, että h:n korkuisessa AVL-puussa on vähintään $F_{h+3} - 1$ solmua:

- Fibonaccin lukusarjan 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.

  (tämä voidaan tarvittaessa osoittaa täsmällisesti helpolla induktiotodistuksella)

- jos käytetään pienimmän h:n korkuisen puun solmulukumäärästä merkintää $S_{min}(h)$, voidaan kirjoittaa

$$S_{min}(h) = \begin{cases} 1 & \text{kun } h = 0 \\ 2 & \text{kun } h = 1 \\ S_{min}(h) = S_{min}(h-1) + S_{min}(h-2) + 1 & \text{kun } h \geq 2 \end{cases}$$

- Aiemmin sovittiin, että tyhjän alipuun korkeus on -1. Tyhjässä alipuussa ei ole yhtään solmua, eli $S_{min}(-1) = 0$

- Kun aloitetaan puiden korkeuden laskeminen jo tyhjästä puusta eli -1:stä, niin pienimmän h:n korkuisen puun solmumäärää kuvaava kaava voidaan kirjoittaa

$$S_{min}(h) = \begin{cases} 0 & \text{kun } h = -1 \\ 1 & \text{kun } h = 0 \\ S_{min}(h) = S_{min}(h-1) + S_{min}(h-2) + 1 & \text{kun } h \geq 1 \end{cases}$$

- Pieninsolmuisen h:n korkuisen puun solmumäärälle on nyt annettu rekursiivinen määritelmä ja todistaakseemme lauseen, on näytettävä, että $S_{min}(h) = F_{h+3} - 1$

- Tehdään merkintä $Apu(h) = S_{min}(h) + 1$

- Tarkastellaan $Apu(h)$:n arvoa:

$$\begin{aligned} Apu(-1) &= S_{min}(-1) + 1 = 1 \\ Apu(0) &= S_{min}(0) + 1 = 2 \\ Apu(h) &= S_{min}(h) + 1 \\ &= S_{min}(h-1) + S_{min}(h-2) + 1 + 1 \\ &= S_{min}(h-1) + 1 + S_{min}(h-2) + 1 \\ &= Apu(h-1) + Apu(h-2) \end{aligned}$$

- Muistutuksena $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, . . . , eli kun $i \geq 2$, niin $F_i = F_{i-1} + F_{i-2}$

- Huomaamme, että $Apu(h)$ toteuttaa saman palautuskaavan kuin Fibonaccin lukusarja:
  - $Apu(-1) = 1 = F_2$, $Apu(0) = 2 = F_3$, $Apu(1) = Apu(0) + Apu(-1) = 3 = F_4$, jne
  - eli $Apu(h) = F_{h+3}$
  - Tämä on melko ilmeistä, mutta voidaan tarvittaessa todistaa helposti induktiolla.

- Nyt $S_{min}(h)$:n arvo on helppo ratkaista:

  $Apu(h) = S_{min}(h) + 1$ ja $Apu(h) = F_{h+3}$ eli $S_{min}(h) = F_{h+3} - 1$

- Todistus on kohtuullisen haastava, eikä kannata olla huolissaan vaikkei todistusta täysin ymmärtäisikään

- Kokeeseen ei tule missään tapauksessa näin vaikeaa todistustehtävää

- Todistuksen huolellinen läpikäyminen lienee kuitenkin monella tavoin opettavaista

- **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 (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 logaritminen määrä puun korkeuteen nähden

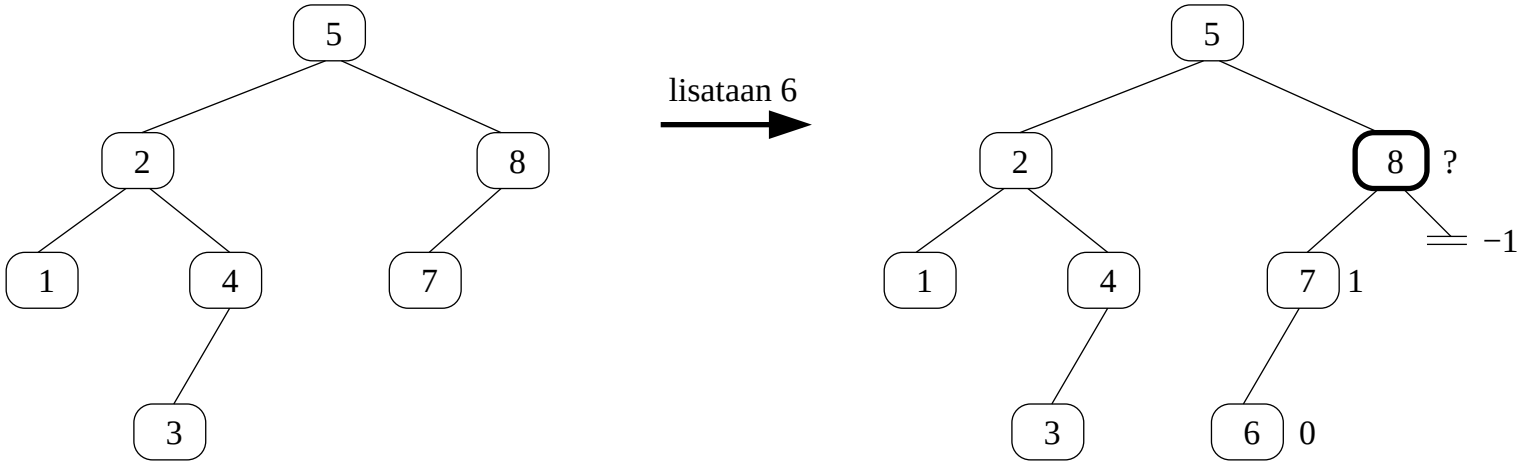
- Kiertoja sovelletaan myös muissa hakupuurakenteissa (punamusta, splay)

- 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

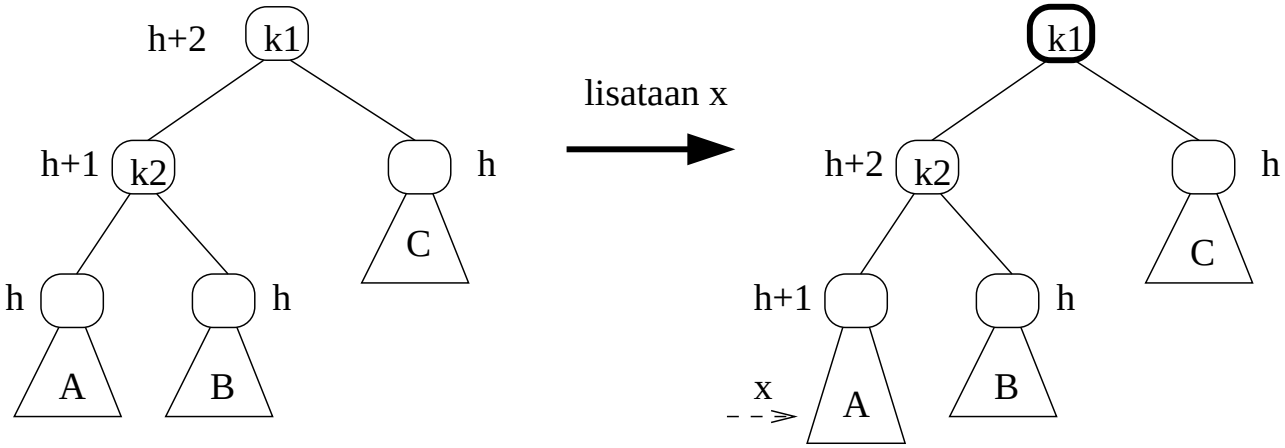
- Periaatteena AVL-puun lisäysoperaatiossa on tehdä ensin lisäys kuten normaaliin binäärihakupuuhun. Jos AVL-ehdon havaitaan menneen rikki lisäyksen yhteydessä, puu korjataan tekemällä sopivat kierto-operaatiot

- 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

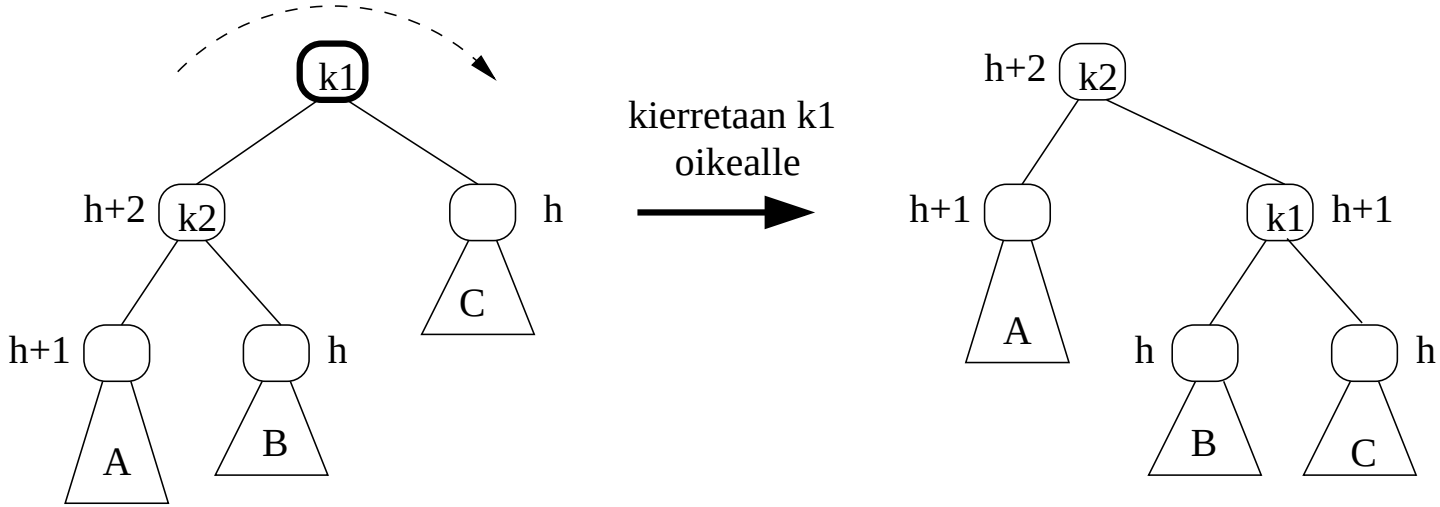
- Tarkastellaan ensin yksinkertaisempaa tapausta epätasapainoon joutumisesta ja siitä toipumisesta

- 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



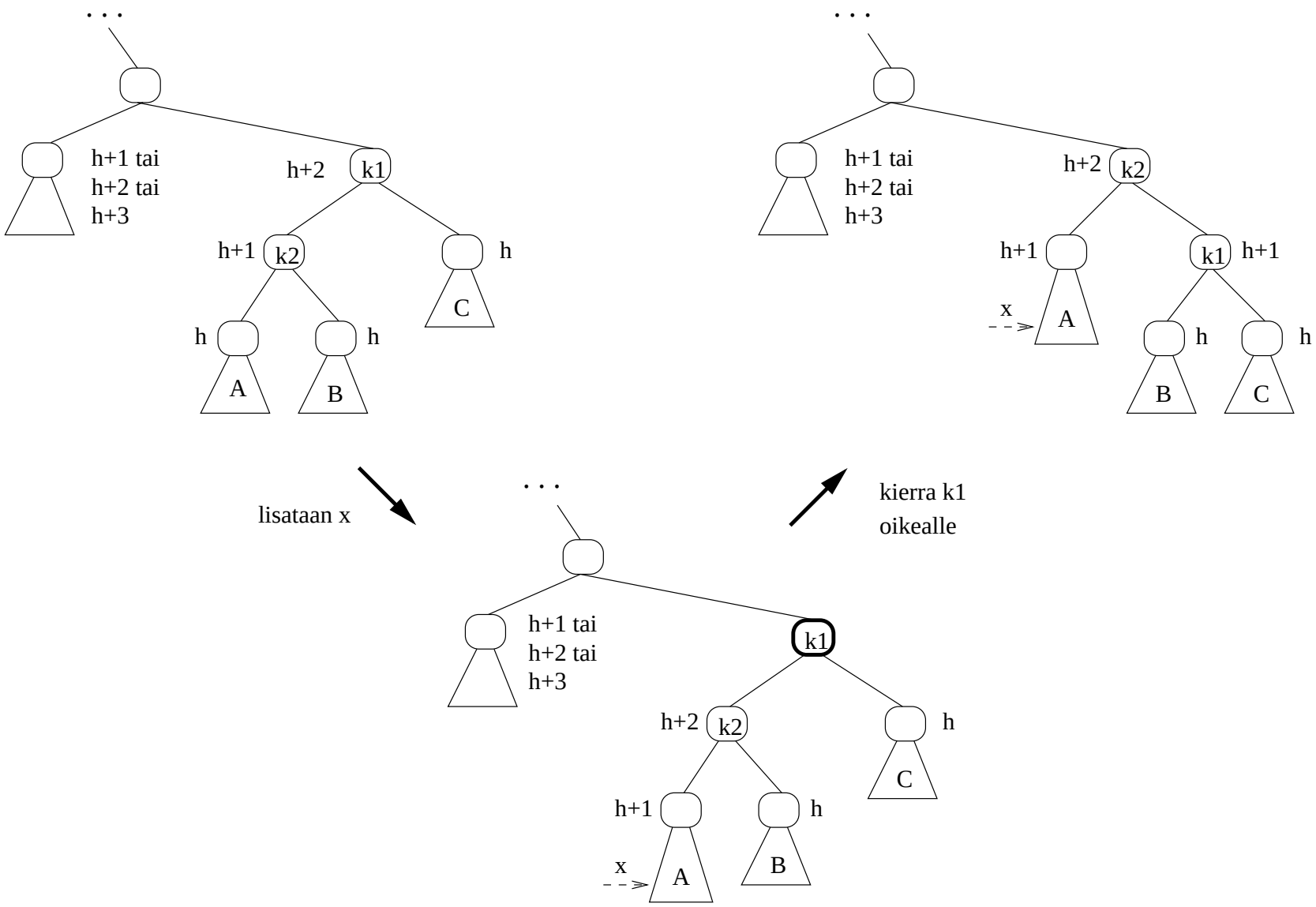
- Epätasapainoa solmuun $k1$ ei olisi syntynyt ellei puun tilanne ennen lisäystä olisi ollut täsmälleen edellisen kalvon vasemmalla olevan kaltainen
  - Jos B:n korkeus olisi ollut $h-1$, niin $k2$ olisi mennyt epätasapainoon. Oletimme kuitenkin, että $k1$ on syvimmällä epätasapainossa oleva solmu
  - 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



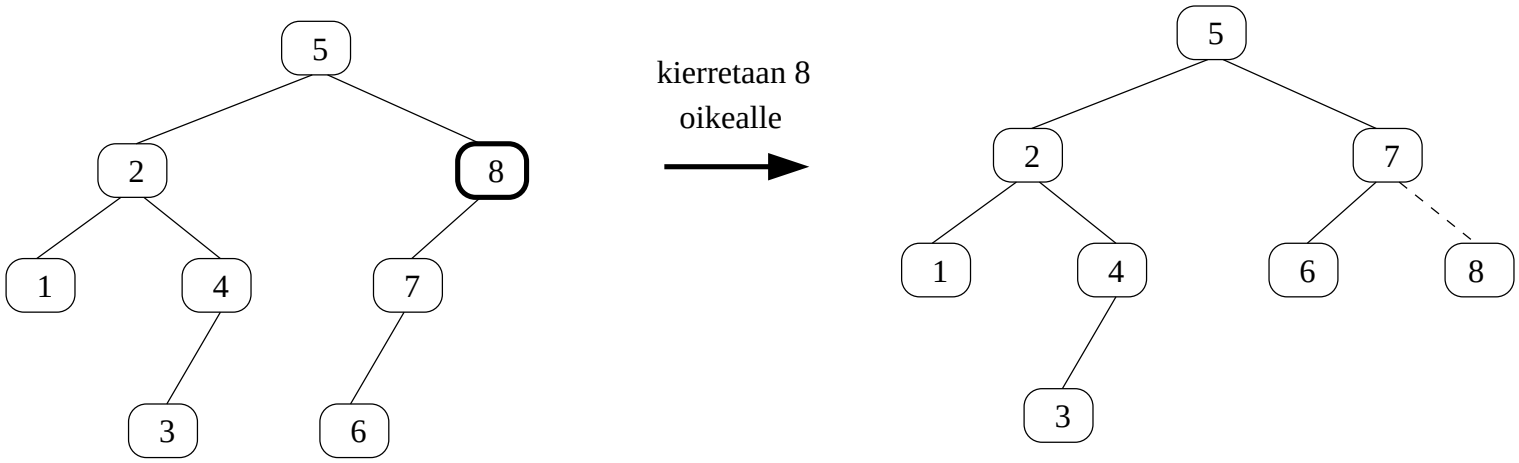
- huomaamme, että kierto palauttaa puun takaisin tasapainoon (ks. seur. kalvo)
  - koko alipuun korkeus on nyt sama kuin ennen lisäysoperaatiota
  - koska oletimme, että $k1$ on alin epätasapainoinen solmu, ei puussa kierron jälkeen enää voi olla epätasapainoisia solmuja

- Epätasapainoisen solmun kierto korjaa koko puun tasapainon:



- Kierto oikealle selvittää myös muutama kalvo sitten aiheuttamamme epätasapainon

- nyt solmu 7 nousee kierrettävän solmun 8 vanhemmaksi

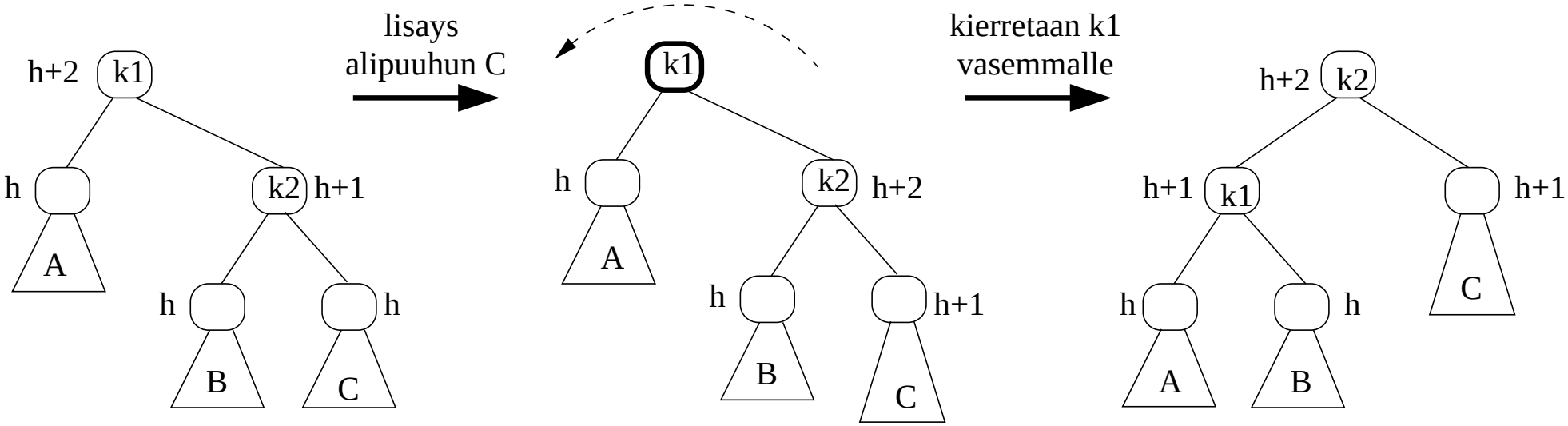


- kierron seurauksena solmusta 7 siis tulee epätasapainoon menneen alipuun uusi juuri

- 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, 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

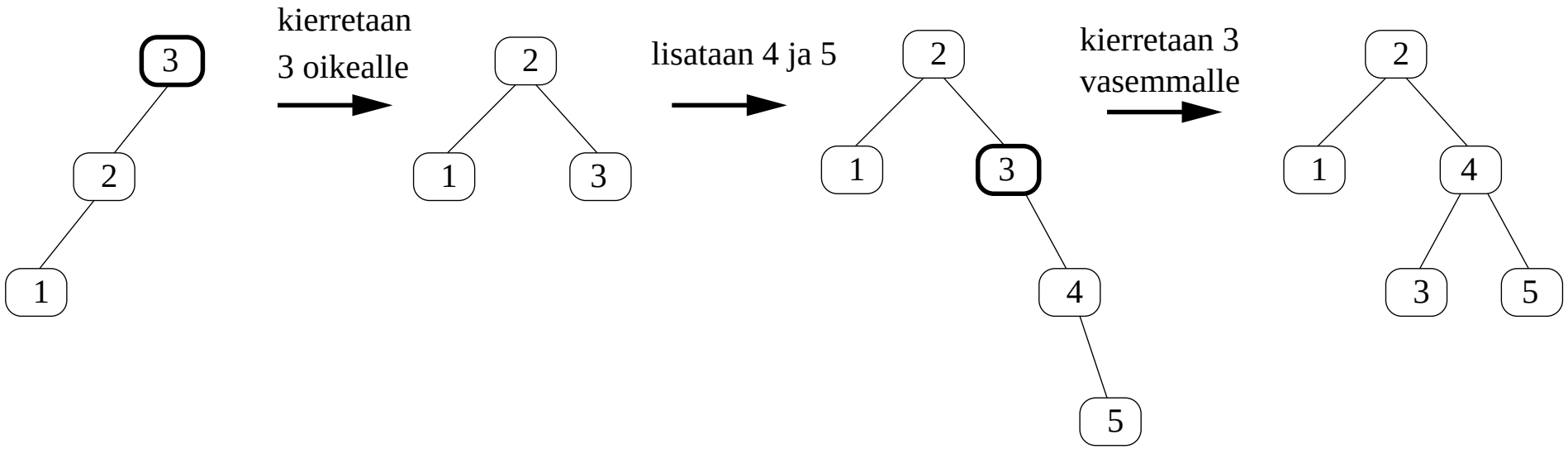
- Riveillä 4-6 asetetaan alipuun uusi juuri vanhempansa oikeaksi tai vasemmaksi lapseksi, riippuen siitä oliko p vasen vai oikea lapsi

- 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
```

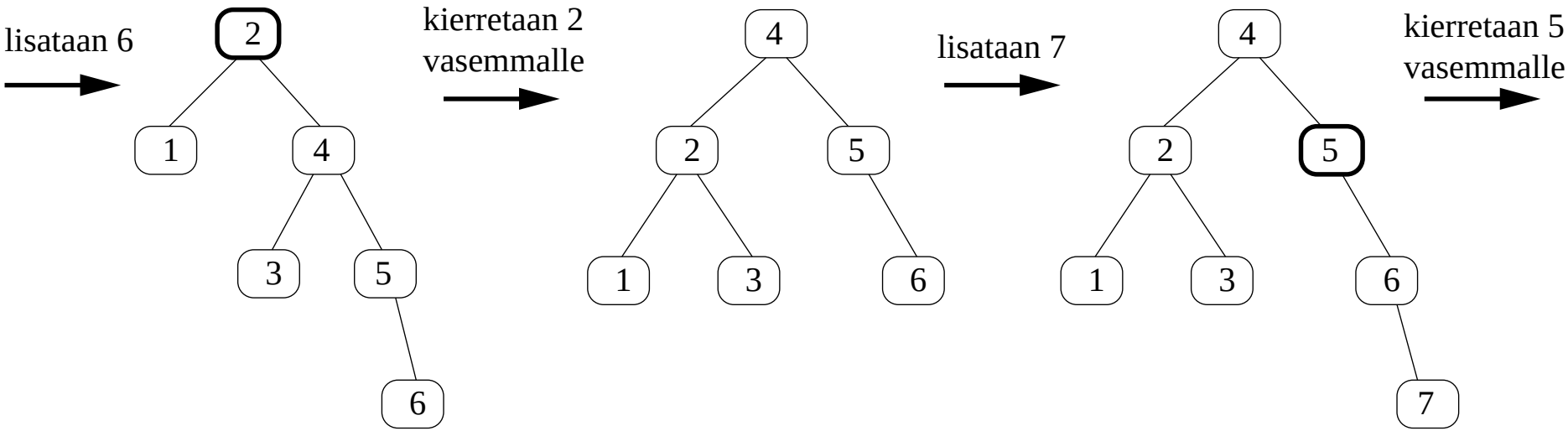
- Nyt siis tiedämme miten puu saadaan tasapainotettua, jos solmuun $k1$ epätasapainon aiheuttama lisäys tehdään
  - $k1$:n oikean lapsen $k2$ oikeaan alipuuhun
  - $k1$:n vasemman lapsen $k2$ vasempaan alipuuhun
- Ensimmäisessä tapauksessa siis kierretään solmua $k1$ vasemmalle ja toisessa tapauksessa 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

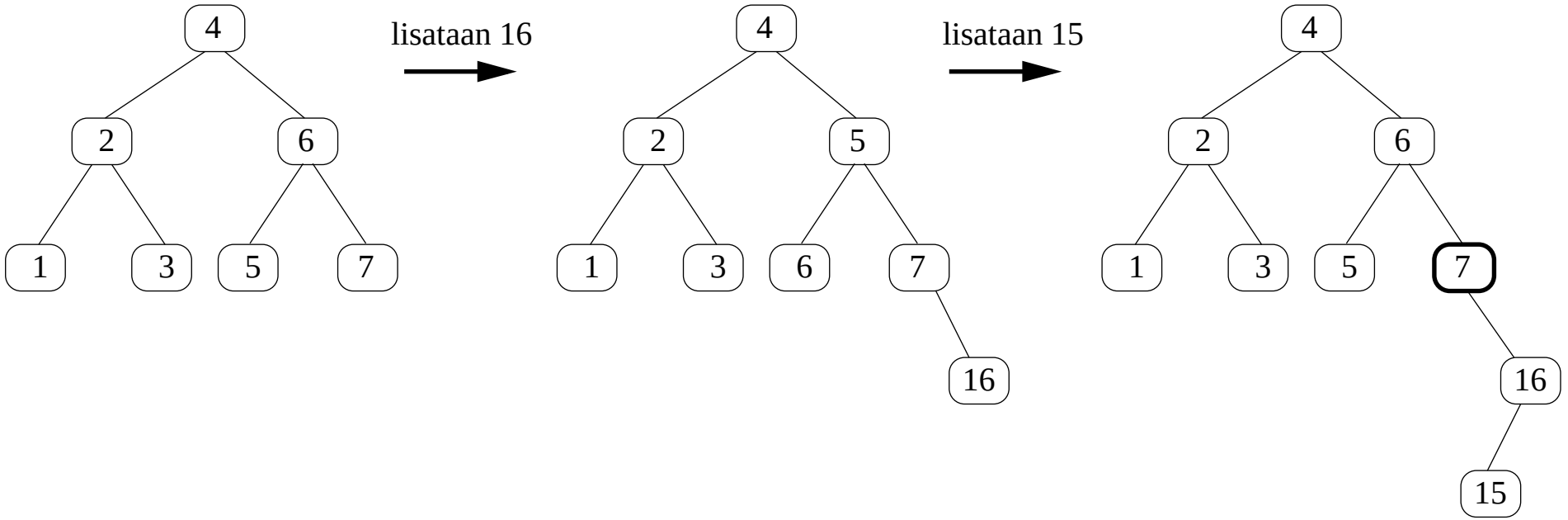


- Jatketaan lisäämällä puuhun 6. Tämä vie juuren, eli solmun 2 epätasapainoon. Jälleen epätasapainon aiheuttaja on lisäys oikean lapsen oikeaan alipuuhun ja kierto vasemmalle korjaa tilanteen.

- 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 kalvon vasemmanpuoleisessa kuvassa

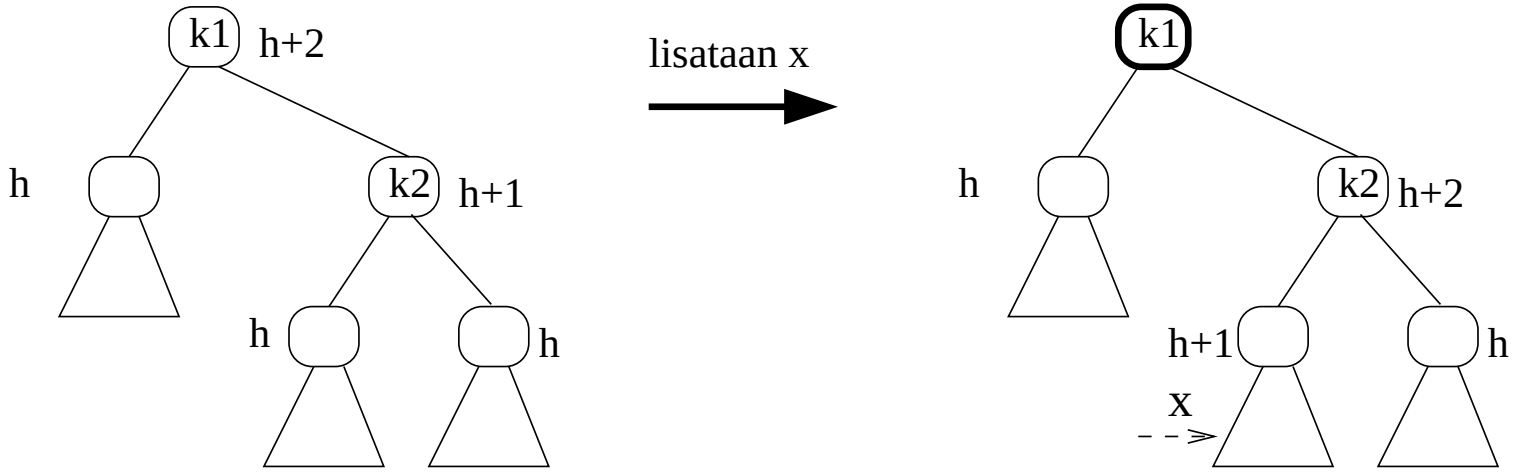


- Jatketaan lisäämällä puuhun 16. Lisäys ei riko tasapainoa
- Kun nyt lisätään puuhun 15 joutuu solmu 7 epätasapainoon
- Epätasapainon aiheuttaa nyt *lisäys oikean lapsen vasempaan alipuuhun*
- tilanne eroaa aiemmista epätasapainon syistä eli *lisäys oikean lapsen oikeaan alipuuhun* ja *lisäys vasemman lapsen vasempaan alipuuhun*, jotka korjautuivat kierto-operaatiolla



- kumpikaan kierto-operaatio ei näytä korjaavan solmun 7 epätasapainoa

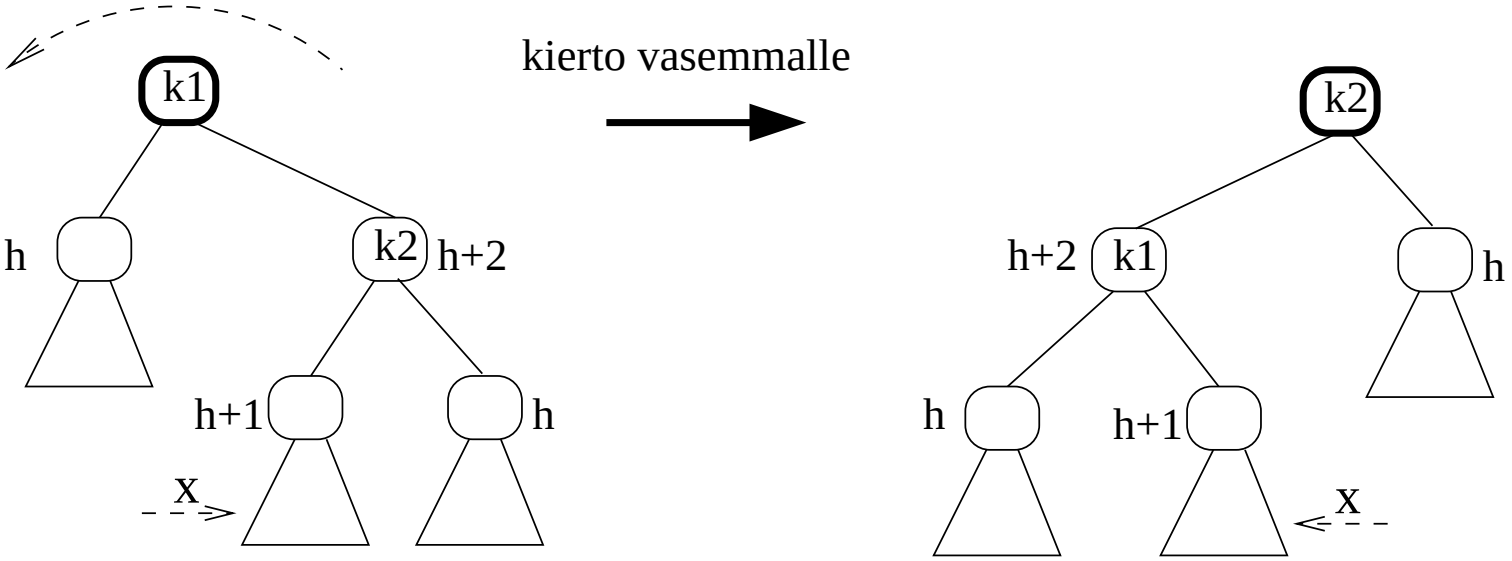
- Tutkitaan tarkemmin tätä tilannetta, johon kierto-operaatio ei näytä tehoavan
- Olkoon $k1$ syvimmällä epätasapainossa lisäyksen jälkeen oleva solmu
- Käsitellään ensin tilannetta, jossa lisätty solmu menee $k1$:n oikean lapsen vasempaan alipuuhun



- Epätasapainoa solmuun $k1$ ei olisi syntynyt ellei puun tilanne ennen lisäystä olisi ollut täsmälleen vasemmalla oleva, tämän voi päätellä kalvon 184 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

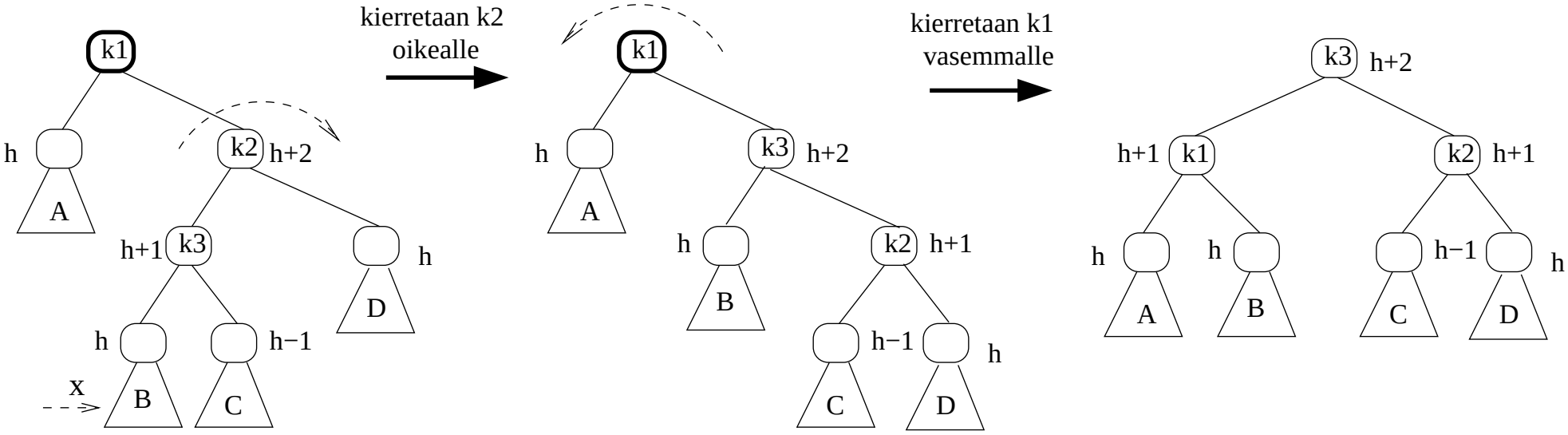


- On keksittävä joku muu ratkaisu

- 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 kalvon kuvassa B:llä
- Tekemällä epätasapainoisen solmun $k1$:n oikealle lapselle $k2$ ensin kierto oikealle ja sitten $k1$:lle kierto vasemmalle, puu menee yllättäen tasapainoon

- Kiertämällä epätasapainoisen lapsi $k2$ oikealle, aikaansaadaan tilanne, jossa epätasapainon syy on $k1$:n oikean lapsen oikeassa alipuussa

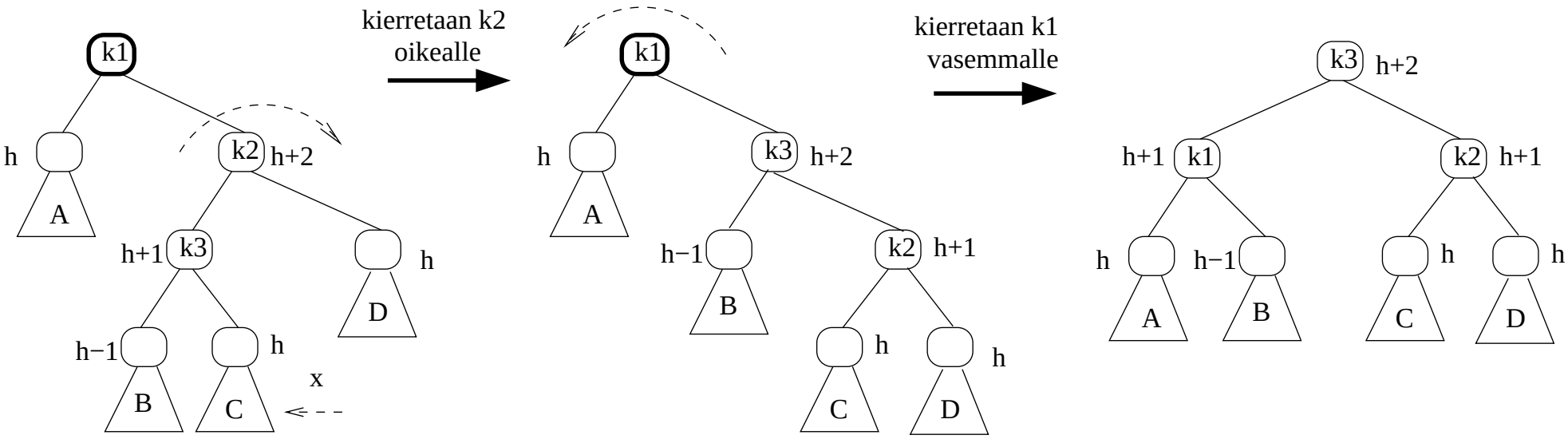
- Tämä on sama tilanne, johon törmäsimme aiemmin, eli epätasapainon korjaa $k1$:n kierto vasemmalle



- 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

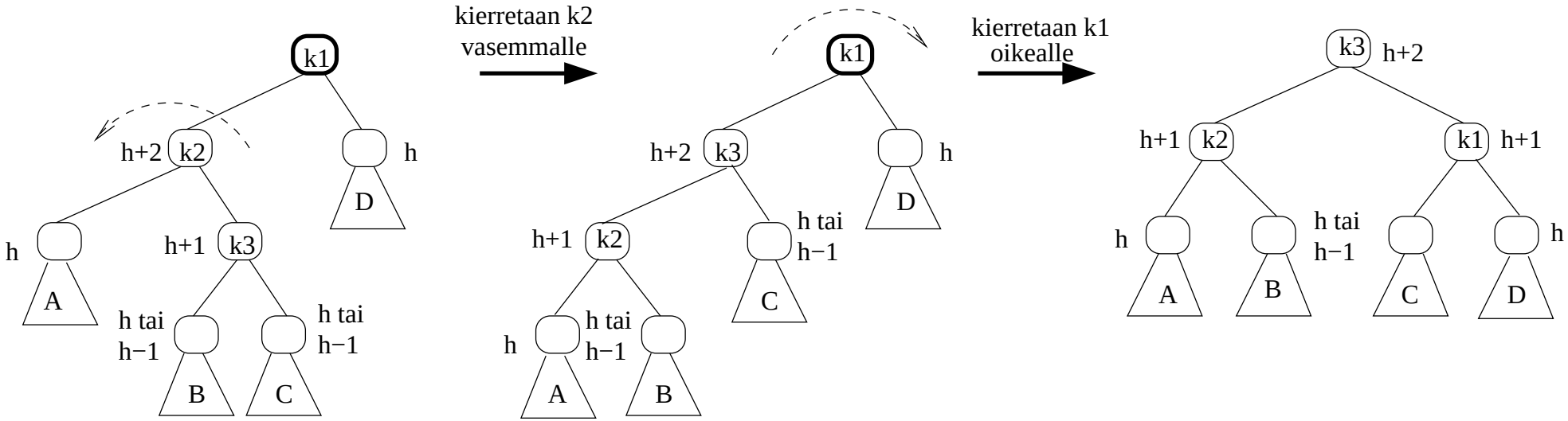
- Seuraavassa vielä näemme, että tilanne, jossa lisäys on tehty $k3$:n oikeaan alipuuhun ratkeaa myös kahdella kierrolla



- 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);
```

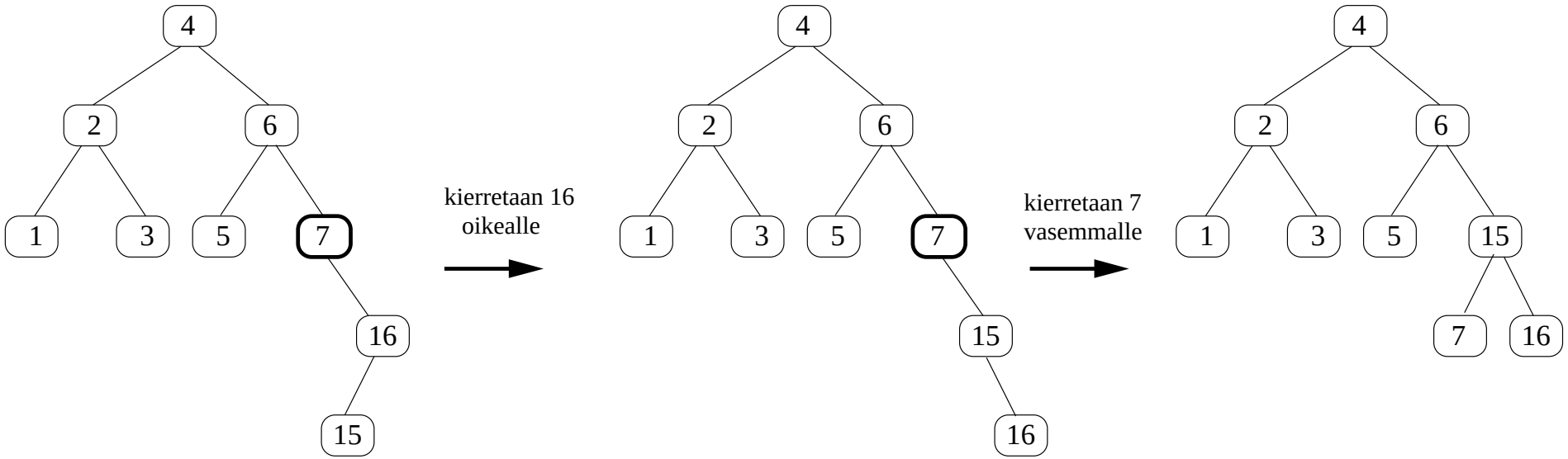
```
LeftRightRotate(k1)
1   k2 = k1.left
2   k1.left = LeftRotate(k2);
3   return RightRotate(k1);
```

- 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

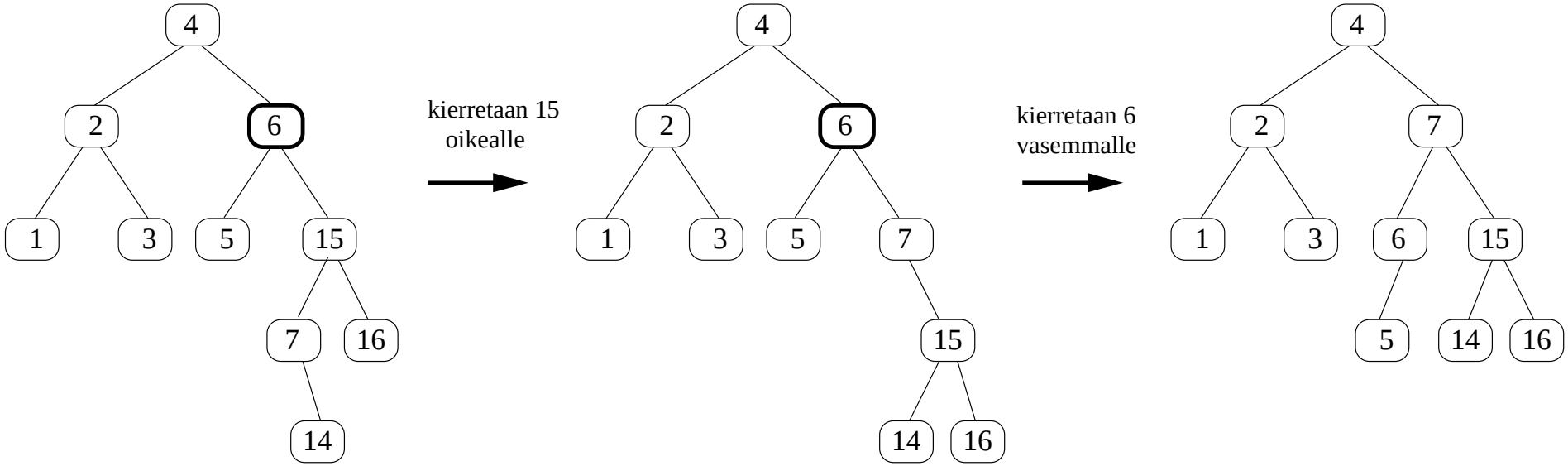
- Ennenkuin esitämme AVL-puuhun lisäyksen pseudokoodina, jatketaan jo aloittamaamme esimerkkiä

- 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 15 vasemmalle
- Näin puun tasapaino palautuu



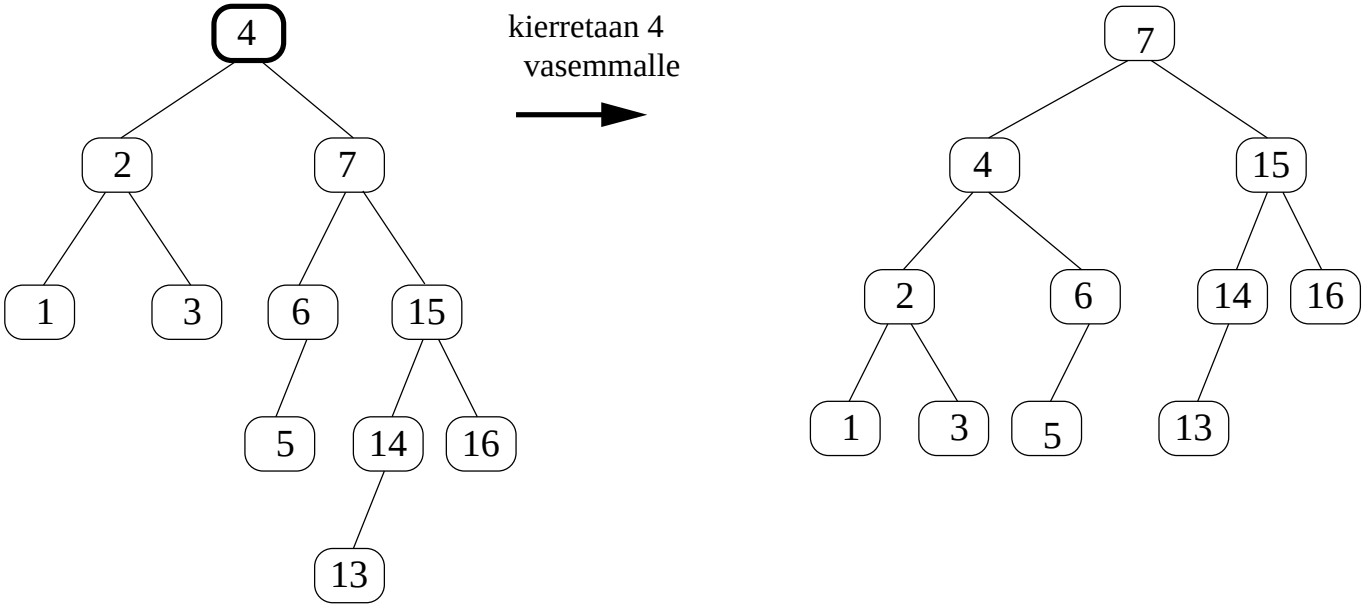
- Lisätään 14. Solmu 6 menee epätasapainoon. Syy epätasapainolle on lisäys oikean lapsen vasempaan alipuuhun.

- 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

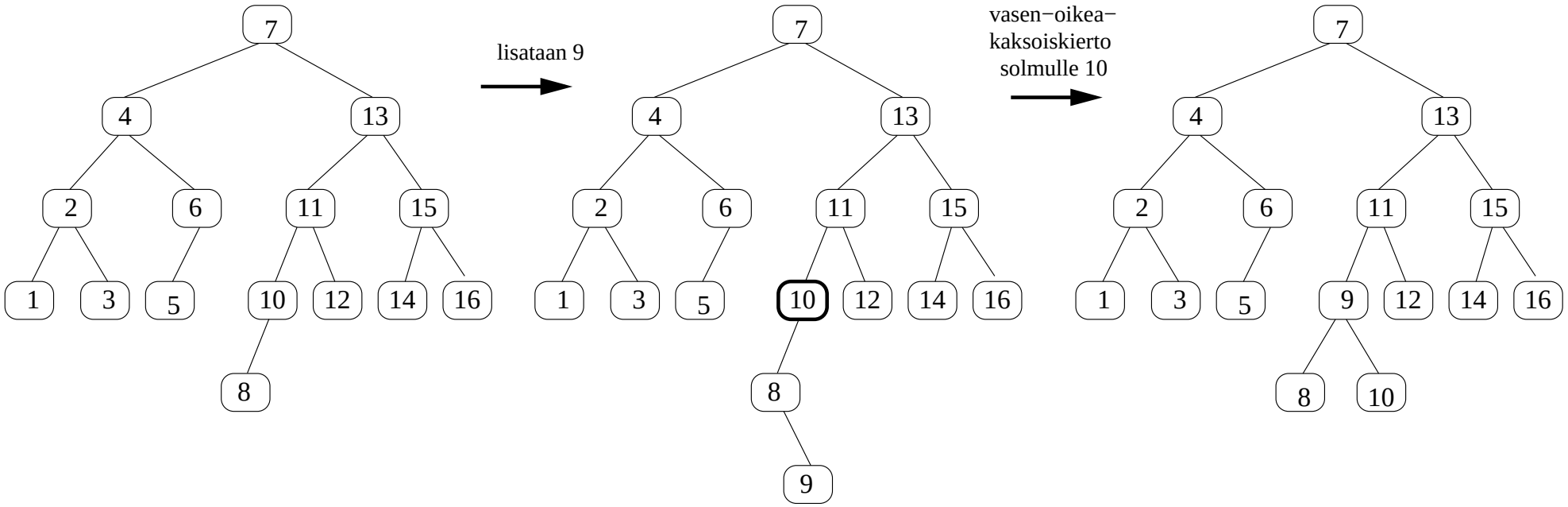
- Kyseessä on siis helpompi tapaus, joka korjautuu yhdellä kierrolla, eli kierretään juurisolmua vasemmalle ja tasapaino palautuu



- Jos jatketaan lisäämällä solmut 12, 11 ja 10, pitää kaikkien lisäyksien yhteydessä tehdä yksi kierto

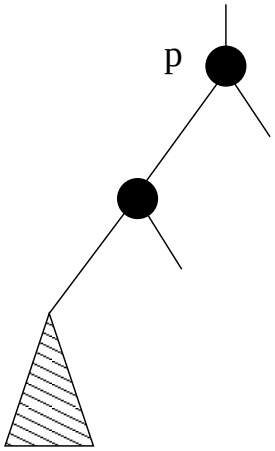
- Edellisten jälkeen solmun 8 voi lisätä puuhun ilman tasapainon rikkoutumista Tuloksena on kuvassa vasemmalla oleva puu

- 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

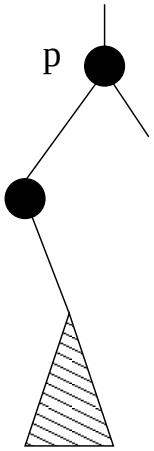


- Esitetään AVL-puun tasapainossa pitävä lisäysoperaatio vielä pseudokoodina
- Lisäys siis tapahtuu seuraavasti
  - lisätään uusi solmu puuhun kuten normaalin binäärihakupuun yhteydessä, asetaan uuden solmun korkeudeksi 0
  - lisäys saattaa viedä puun epätasapainoon
  - epätasapainoon joutuneiden solmujen 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

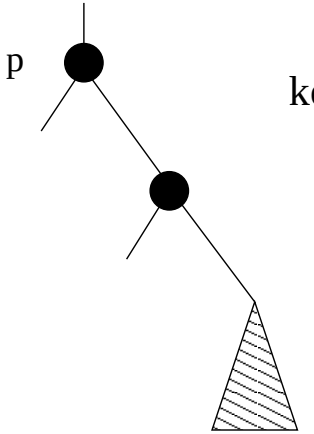


korjaava toimenpide:
rightRotate(p)

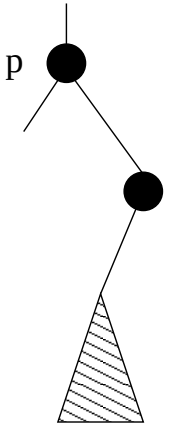


korjaava toimenpide:
leftRightRotate(p)
eli
leftRotate(p.left)
rightRotate(p)

- Jos solmun p epätasapainon syy on sen oikeassa alipuussa, tapaukset ovat:



korjaava toimenpide:
leftRotate(p)

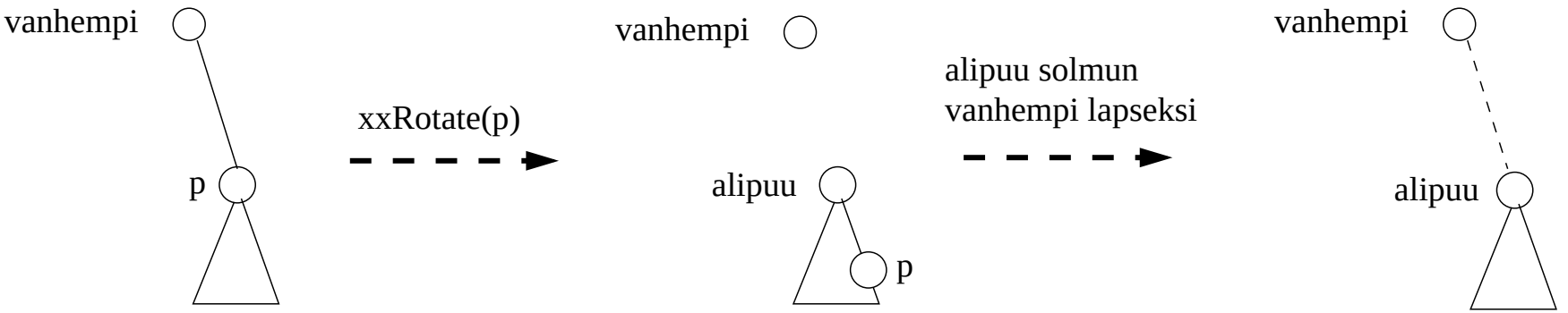


korjaava toimenpide:
rightLeftRotate(p)
eli
rightRotate(p.right)
leftRotate(p)

- esitetään algoritmi ensin korkealla tasolla
- algoritmi käyttää kalvolla 153 esiteltyä normaalin binääripuun lisäysoperaatiota jota on muokattu siltä osin, että se palauttaa viitteen lisättyyn solmuun

```
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 = rotateRight(p)
8            elsif epätasapainon syy vasemman lapsen oikea alipuu
9                alipuu = rotateLeftRight(p)
10           elsif epätasapainon syy oikean lapsen oikea alipuu
11               alipuu = rotateLeft(p)
12           else  // epätasapainon syy oikean lapsen vasen alipuu
13               alipuu = rotateRightLeft(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 epätasapainon
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
35      p = p.parent                // jatketaan kohti juurta
```

```
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 epätasapainon edellisellä sivulla
19      if height(p.right) == height(p.left)+2     // oikea lapsi aiheuttaa epätasapainon
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 on palannut tasapainoon
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, matkan varrella kohdattujen solmujen korkeuskentät päivitetään rivillä 34 ja kiertojen tapauksessa riveillä 17 ja 32
- jos epätasapainoinen solmu löytyy ja kierto-operaatio suoritetaan, on taattua että puu menee tasapainoon ja algoritmi lopettaa (rivit 18 ja 32)
- muussa tapauksessa jatketaan solmujen tasapainon tutkimista aina juureen asti

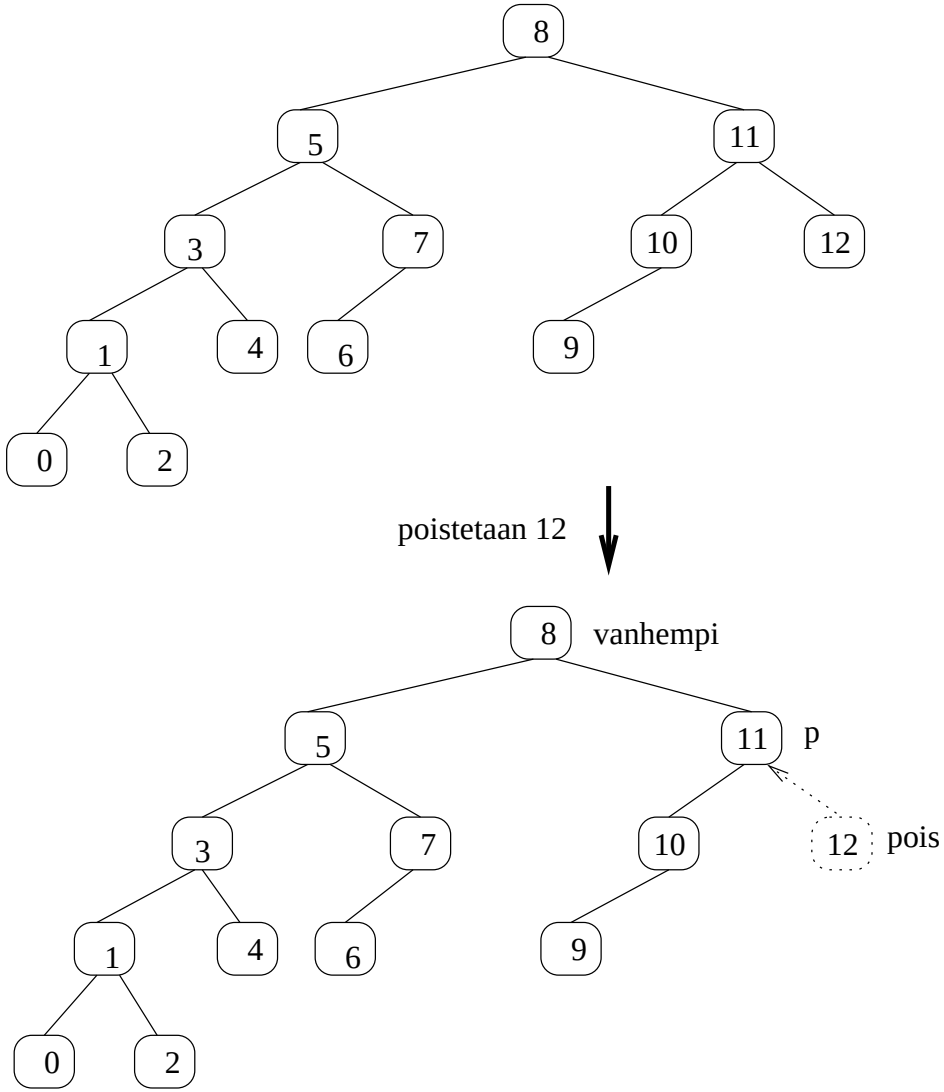
- Analysoidaan algoritmin aika- ja tilavaativuutta
- Käytetään analyysin pohjana kalvon 210 abstraktimpaa versiota
- Ensin kutsutaan normaalia binäärihakupuun lisäysoperaatiota, jonka aikavaativuus on $\mathcal{O}(h)$ puun korkeuhden h suhteen. Olemme osoittaneet aiemmin, että AVL-puun korkeus solmujen 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-insertistä käyttävät ainoastaan vakiomäärän apumuuttujia, eli kokonaisuudessaan algoritmin tilavaativuus on vakio

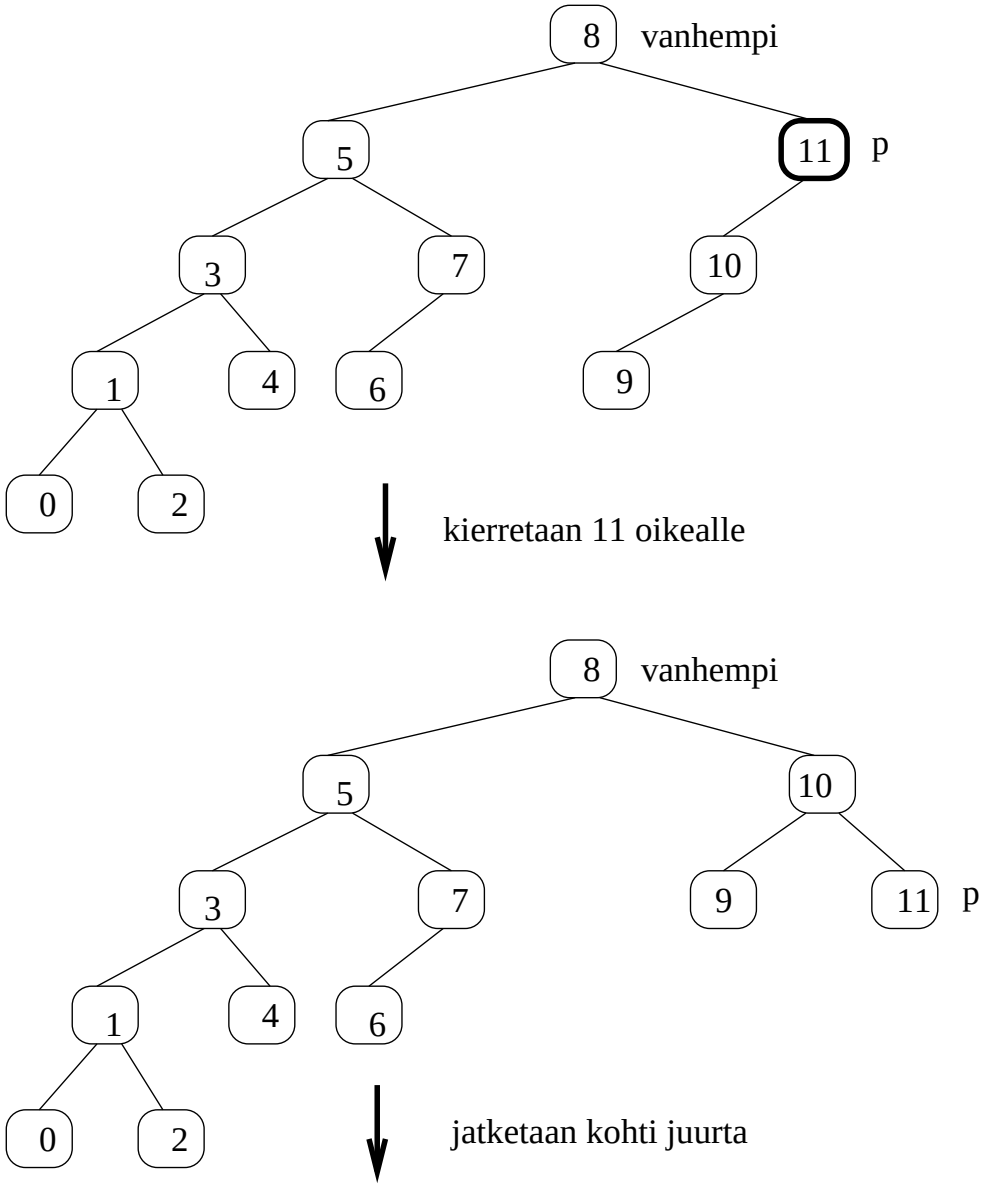
- AVL-delete on hiukan AVL-insertiä monimutkaisempi operaatio
- Solmu poistetaan ensin käyttäen normaalia delete-operaatiota
- Algoritmi käyttää kalvoilla 158-9 esiteltyä normaalin binääripuun poisto-operaatiota, joka palauttaa viitteen siihen solmuun, joka puusta todellisuudessa poistettiin (kaikissa 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

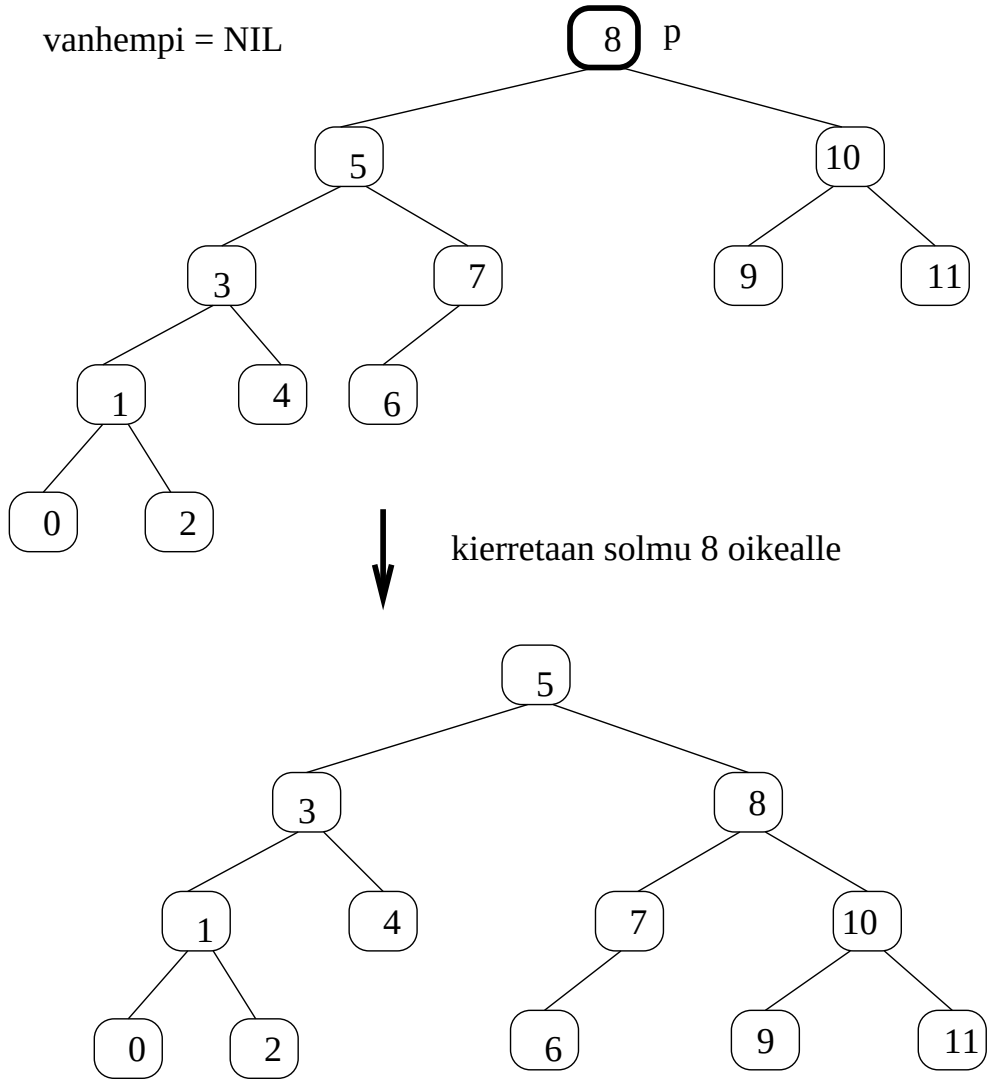
```
AVL-delete(T,x)
1   pois = delete(T,x)
2   p = pois.parent
3   while p ≠ NIL
4       if p epätasapainossa
5           vanhempi = p.parent
6           if epätasapainon syy vasemman lapsen vasen alipuu
7               alipuu = rotateRight(p)
8           elsif epätasapainon syy vasemman lapsen oikea alipuu
9               alipuu = rotateLeftRight(p)
10          elsif epätasapainon syy oikean lapsen oikea alipuu
11              alipuu = rotateLeft(p)
12          else  // epätasapainon syy oikean lapsen vasen alipuu
13              alipuu = rotateRightLeft(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 algoritmi tarkastaa, 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 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 delete ja kierto-operaatioiden tilavaativuus on todettu jo aiemmin vakioksi, samoin muutkin osat ALV-deletessä 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 tuskin käytännössä koskaan on 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

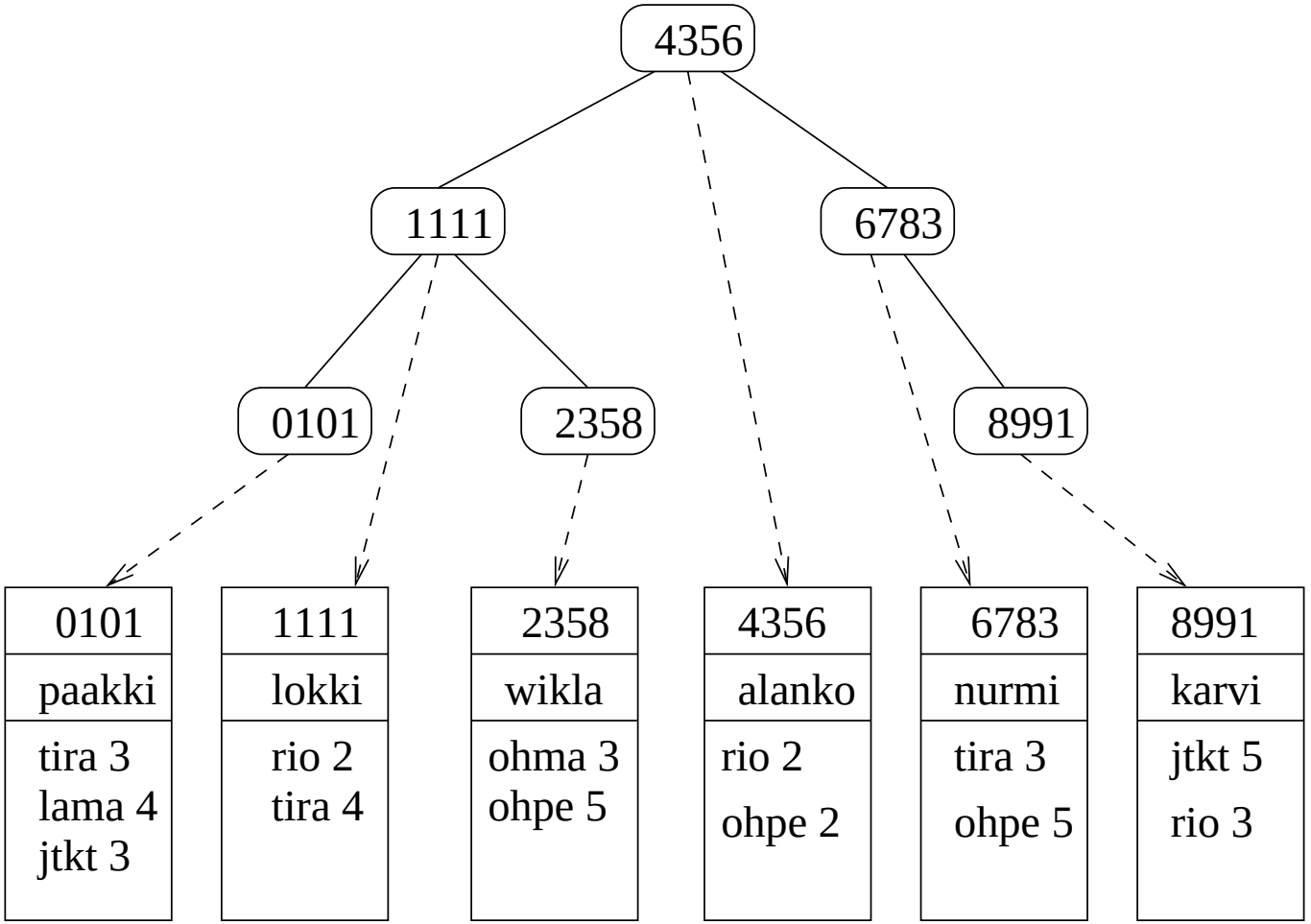
- Jos organisoitavaa dataa on paljon, paras ratkaisu on tallettaa muu data omana olionaan ja lisätä puusolmuihin avaimen lisäksi viite muun datan tallettavaan olioon

- puusolmu muodostuu tällöin kentistä:

key	talletettu avain
data	viite avaimeen liittyvän tiedon tallettavaan olioon
left	viite vasempaan lapseen
right	viite oikeaan lapseen
p	viite vanhempaan

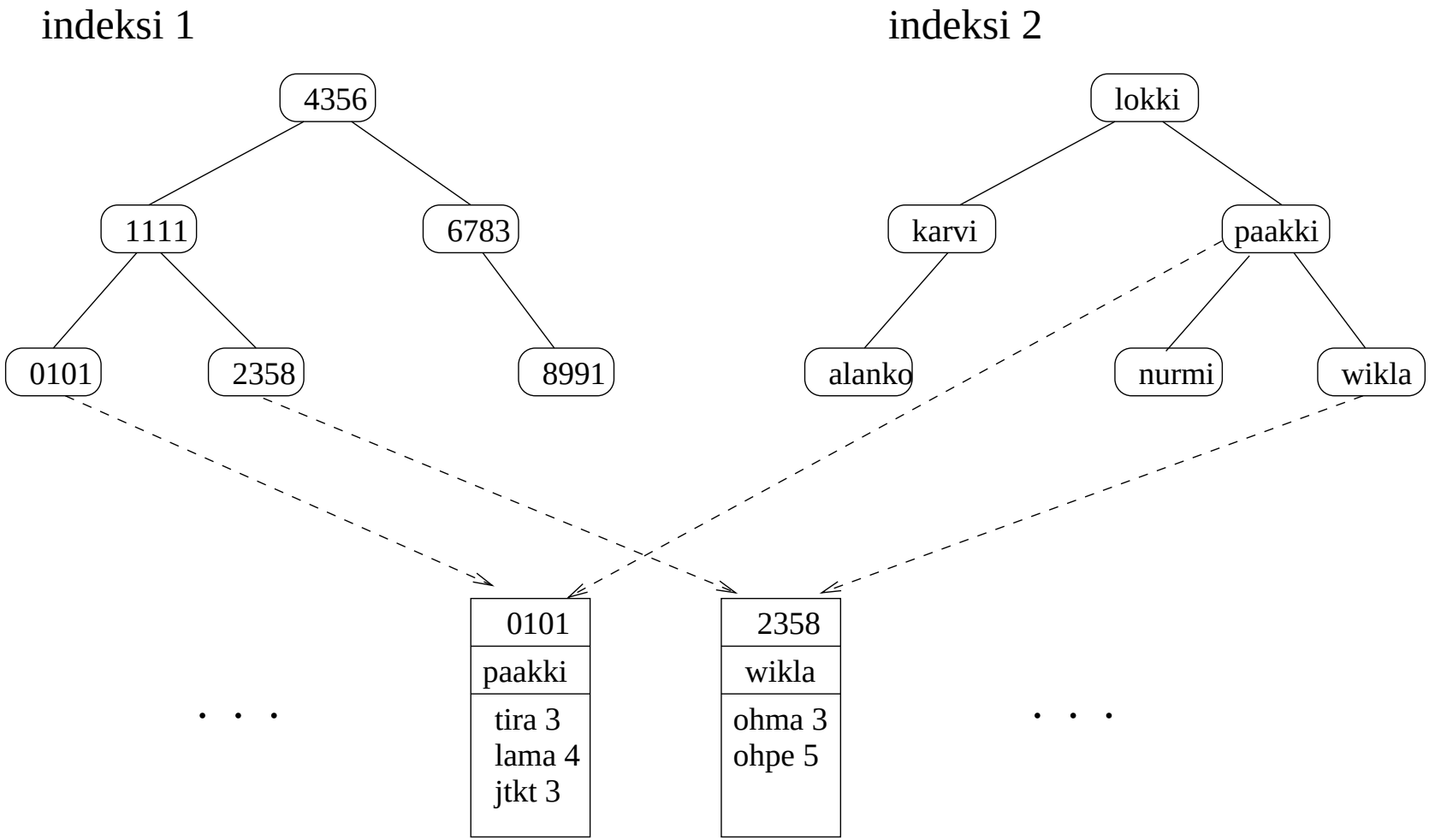
- näin puu toimii indeksirakenteena, jonka avulla talletettuun tietoon on mahdollista tehdä nopeita hakuja avaimen perusteella

- 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

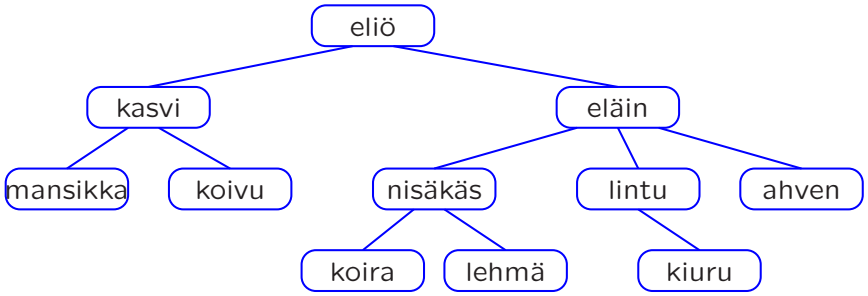
- TreeMap taas toimii edellisten sivujen indeksirakenteiden tapaan, eli TreeMap:iin talletetaan avain-dataolio–pareja, ja etsintä avaimeen perustuen on tehokas

- Esim. edellisten kalvojen 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

- TreeSet:iin tutustumme viikon 6 laskareissa ja TreeMap:iin viikon 7 laskareissa

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:



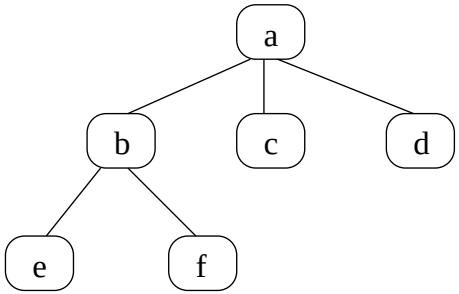
- Muutaman kalvon päästä tutustumme puiden käyttöön ongelmanratkaisussa
- Kaikki hyödylliset puut eivät siis suinkaan ole binääripuita tai hakupuita
- Miten voimme tallettaa yleisen puun?

- jos tiedämme mikä on solmun maksimihaarautumisaste, voimme tallettaa solmuun viitteet kaikkiin mahdollisiin lapsiin

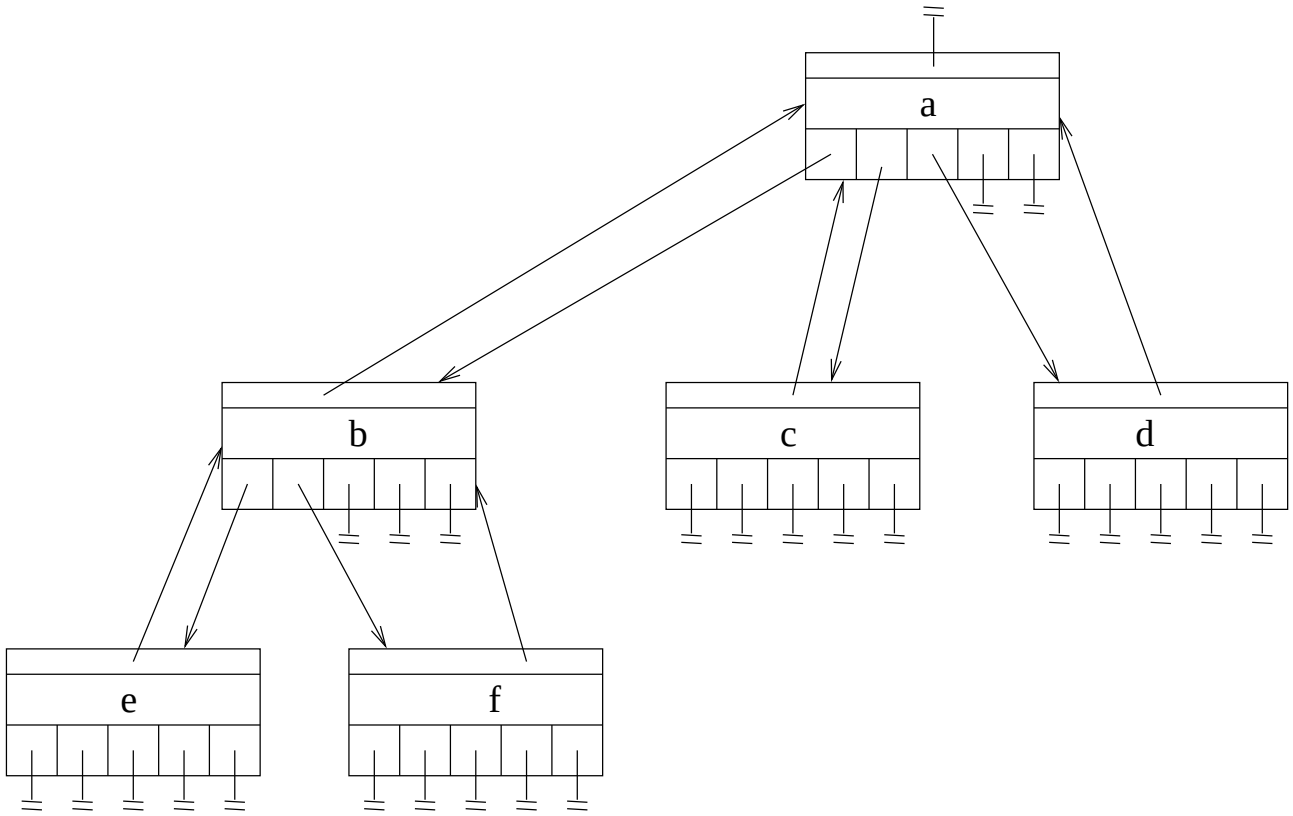
- eli puusolmu muodostuu tällöin kentistä:

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

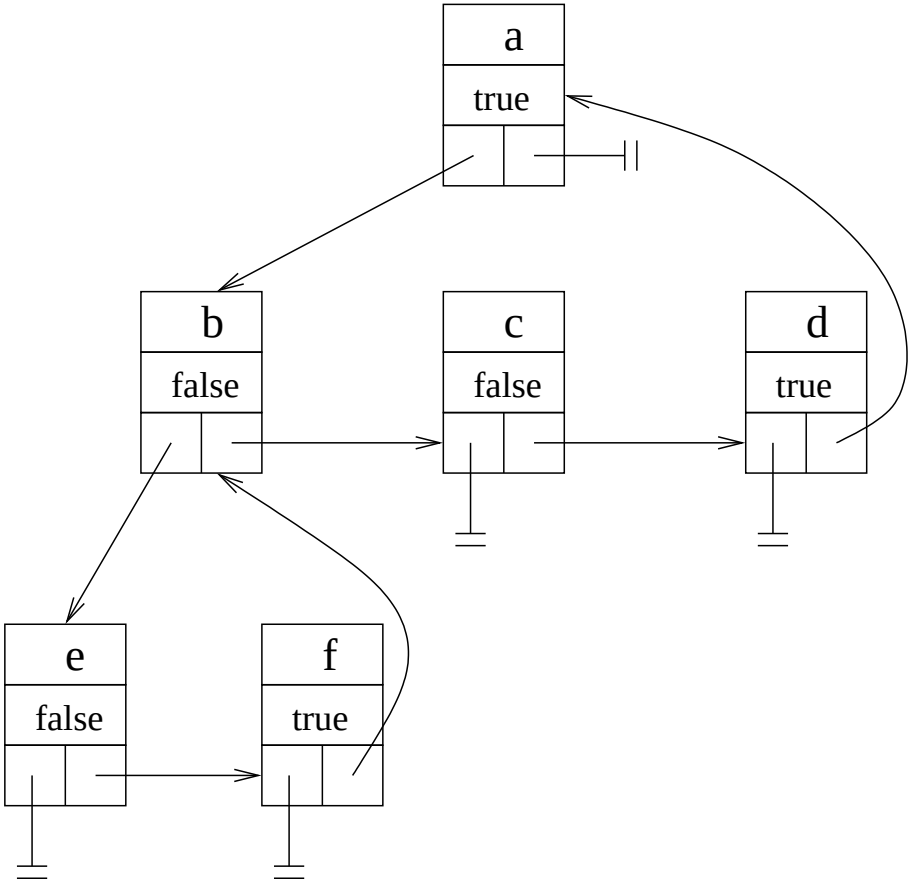
- Näin saataisiin lapsimäärästä joustava, ja turhaa tilaa ei varattaisi kohtuuttoman paljoa

- 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 edelliseksi löydetty lapsi y

```
nextchild(y)
    if y.last == true return NIL
    else return y.next
```

- viimeisen lapsen jälkeen operaatio palauttaa NIL

- binäärihakupuun yhteydessä saimme tulostettua 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

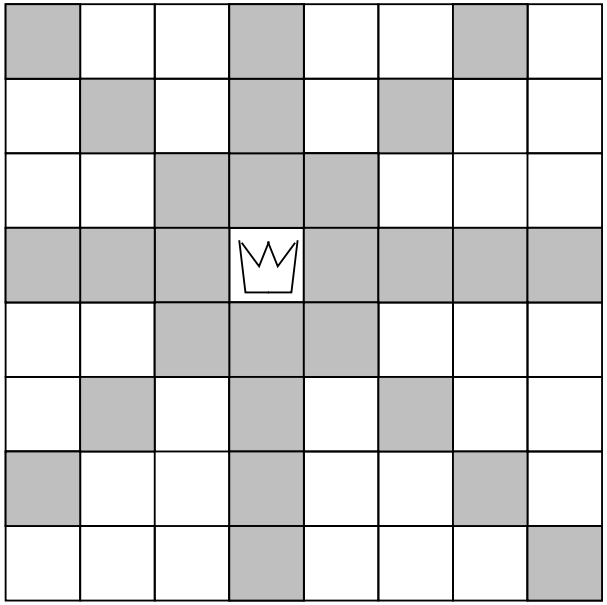
```
postorder-tree-walk(x)
    y = firstchild(x)
    while y ≠ NIL
        postorder-tree-walk(y)
        y = nextchild(y)
    print x.key
```

- 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

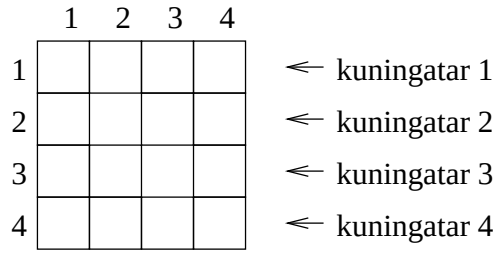
Puut ongelmanratkaisussa: kahdeksan kuningattaren ongelma

- Yksi puiden tärkeistä käyttötavoista on ongelmanratkaisussa tapahtuvan laskennan etenemisen kuvaaminen
- *kahdeksan kuningattaren ongelma:* miten voimme sijoittaa shakkilaudalle 8 kuningatarta sitten että ne eivät uhkaa toisiaan?
- kuningatar uhkaa samalla rivillä, sarakkeella sekä diagonaalilla olevia ruutuja

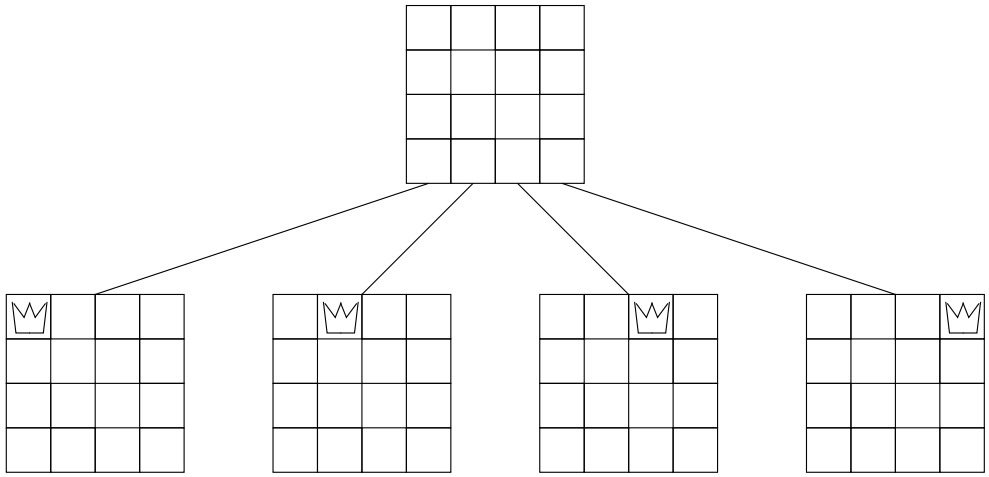


- yleistetty version ongelmasta: miten saamme sijoitettua n kuningatarta $n \times n$ -kokoiselle shakkilaudalle

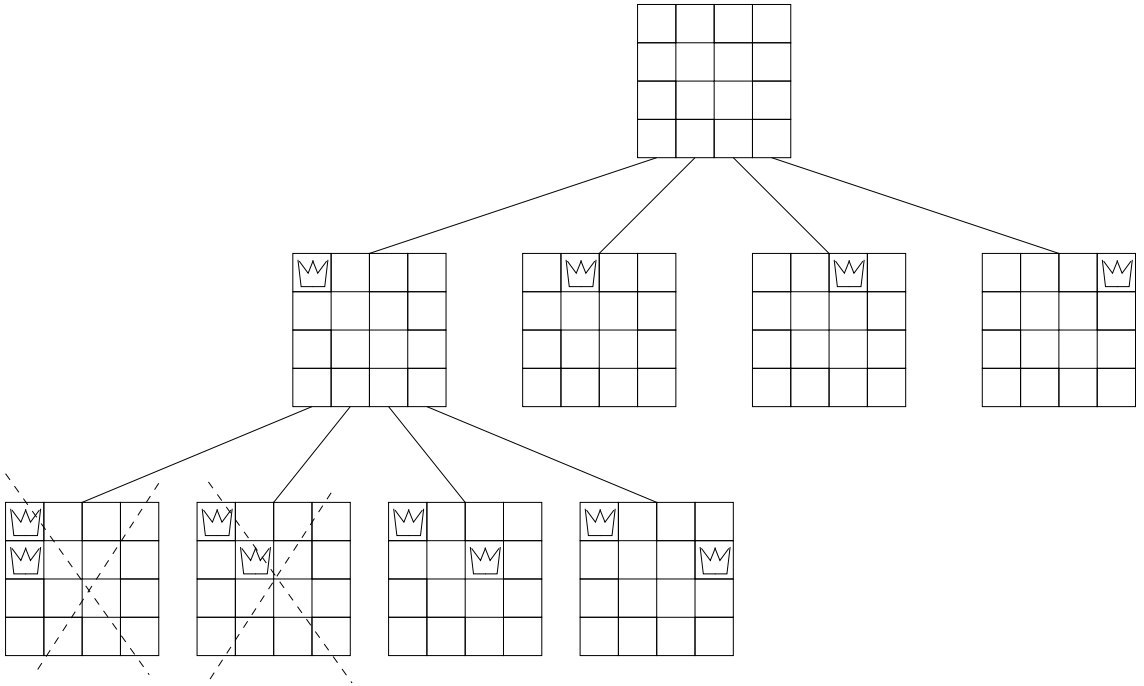
- tarkastellaan ensin tapausta missä $n = 4$. selvästikin jokaisella rivillä täytyy olla tasan 1 kuningatar:



- etsitään oikea kuningatarasetelma systemaattisesti
  - aloitetaan tyhjältä laudalta
  - tämän jälkeen asetetaan kuningatar riville 1
  - neljä eri mahdollisuutta:



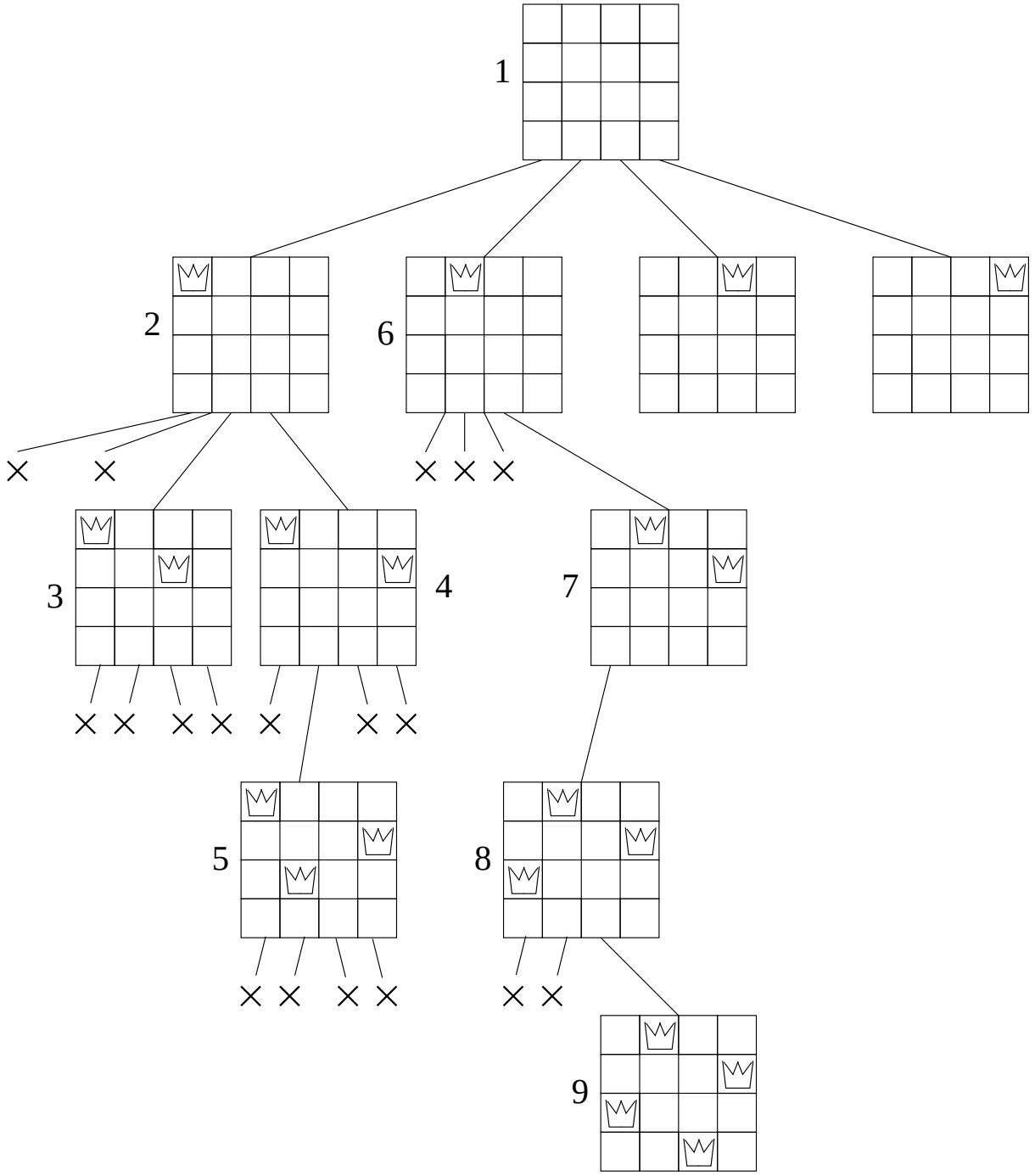
- Seuraavaksi tarkastellaan miten kuningattaret voidaan asettaa riville 2. Aloitetaan vasemmanpuoleisesta 1 rivin valinnasta



- huomaamme että olemme muodostamassa puuta, joka kuvaa erilaisia ratkaisumahdollisuuksia

- kaksi vasemmanpuoleisinta yritystä ovat tuhoon tuomittuja, eikä enää kannata tutkia mitä niissä haaroissa tapahtuu

- seuraavalla kalvolla 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 funktio check(table)
  - funktio palauttaa true jos sen parametrina saama pelitilanne voidaan vielä täydentää ratkaisuksi tai on jo ratkaisu kuningatarongelmaan
  - jos pelilaudalla on toisiaan uhkaavia kuningattaria, operaatio palauttaa false

- 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 = table
8       table2[x,row] = true
9       putqueen(table2,row+1)
```

- operaation toiminta parametreilla $table, row$
  - operaatio tarkastaa ensin (rivi 1) edustaako $table$ pelilautaa mikä 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 sensijaan on suuri, sillä vaikka kaikkia solmuja ei tarvisekkaan käydä läpi, kasvaa läpikäytävien solmujen määrä kuitenkin eksponentiaalisesti n:n suhteen

- tälläisestä implisiittisen puun läpikäyntimenetelmästä käytetään nimitystä *peruuttava etsintä* (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 $O(n^2)$, eli jokainen funktion rungon suoritus kuluttaa taulukkojen kopiointiin aikaa n kertaa $O(n^2)$, eli $O(n^3)$
- taulukon kuljettaminen parametrina ei ole oikeastaan tarpeen: riittää että pidetään rakennuksen alla oleva kuningatarasetelma globaalina muuttujana olevassa taulukossa
- oletetaan, että table 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
```

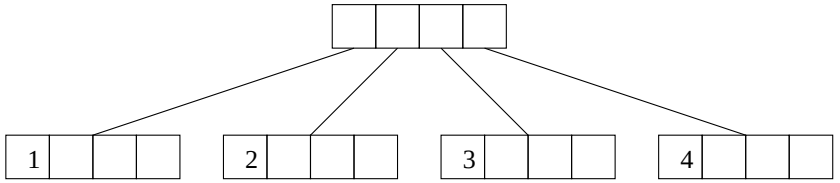
- funktiorungon yhden suorituksen aikavaativuus on nyt funktion `check` suoritusaika plus $O(n)$
- algoritmin kokonaisaikavaativuus on siis solmujen lukumäärä kertaa funktiorungon suoritusaika
- algoritmin tilavaativuus pienenee, sillä edellisessä versiossa jokainen rekursiivinen kutsu talletti oman kopionsa asetelmasta ja vei tilaa $O(n^2)$ ja koska rekursiivisia kutsuja voi olla kerrallaan menossa n kpl tilavaativuus oli $O(n^3)$
- uudessa versiossa jokainen rekursiokutsu vie tilaa ainoastaan vakion verran, eli koko algoritmin tilavaativuus on $O(n)$
- 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);
   }
}
```

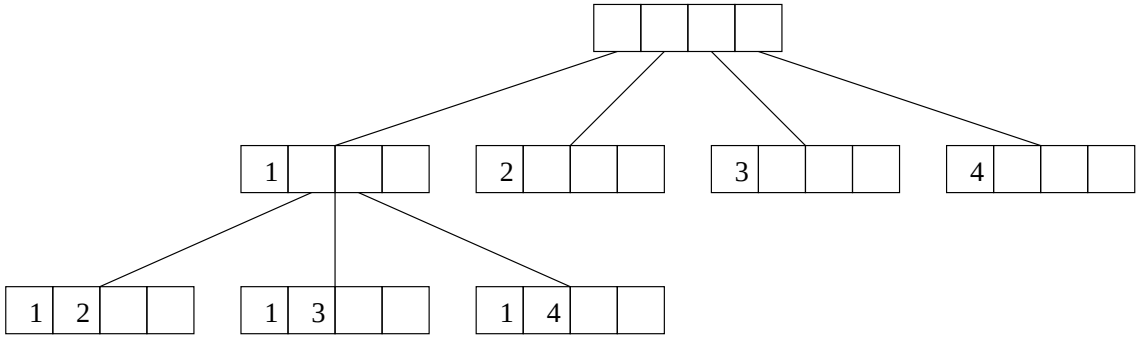
Permutaatioiden generoiminen

- samaa ratkaisustrategiaa voimme käyttää myös seuraavaan ongelmaan: miten voidaan generoida lukujen $1, 2, \ldots, n$ kaikki permutaatiot?

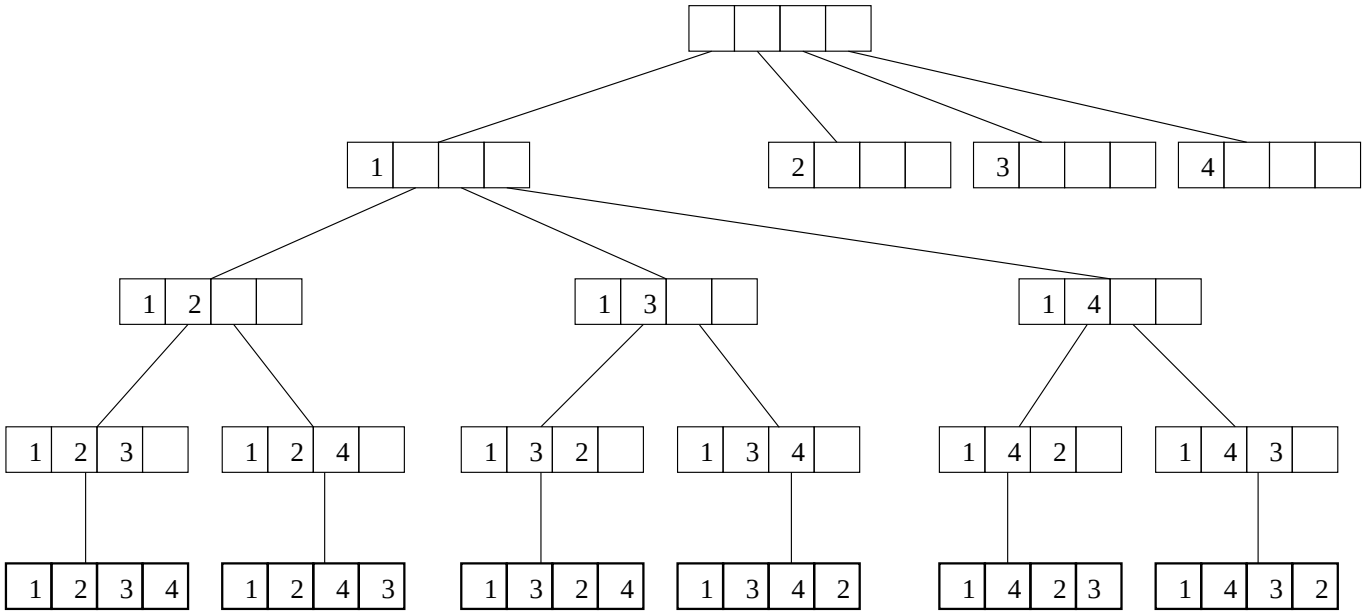
- tarkastellaan permutaatioita luvuille $1, 2, 3, 4$
  - permutaation ensimmäinen luku voi alkaa mikä tahansa yo. luvuista:



  - 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
3               used2 = used
4               used2[i] = true
5               table2 = table
6               table2[k] = i
7               generate(table2, 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 3-6 käyttämätön luku lisätään permutaatioon (taulukkoon table2), 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
- on selvää, että n^{n-1} on alle puolet n^n:stä, eli $n^{n-1} \leq \frac{1}{2}n^n$, siispä

  $1 + n + n^2 + \ldots + n^{n-1} + n^n \leq 1 + n + n^2 + \ldots + n^{n-2} + \frac{1}{2}n^n + n^n$

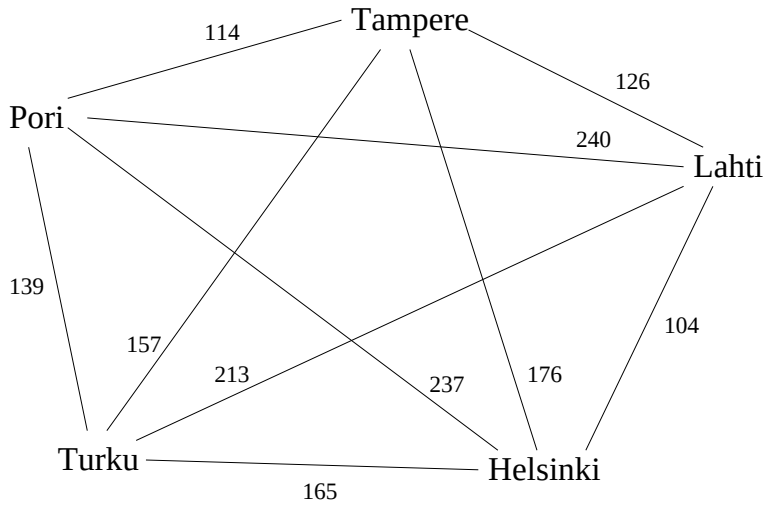
  samoin n^{n-2} on alle puolet n^{n-1}:stä, n^{n-3} n^{n-2}:sta ... eli $n^{n-i} \leq \frac{1}{2^i}n^n$, joten

  $$\begin{aligned} 1 + n + n^2 + \ldots + n^n &\leq 1 + \tfrac{1}{2^n}n^n + \tfrac{1}{2^{n-1}}n^n + \ldots + \tfrac{1}{2^2}n^n + \tfrac{1}{2^1}n^n + \tfrac{1}{2^0}n^n \\ &= 1 + n^n(\tfrac{1}{2^n} + \tfrac{1}{2^{n-1}} + \ldots + \tfrac{1}{2^2} + \tfrac{1}{2^1} + \tfrac{1}{2^0}) \\ &= 1 + n^n \textstyle\sum_{i=0}^{n} (\frac{1}{2})^i \\ &\leq 1 + n^n \textstyle\sum_{i=0}^{\infty} (\frac{1}{2})^i \\ &= 1 + n^n \tfrac{1}{1-\frac{1}{2}} = 1 + 2n^n = \mathcal{O}(n^n) \end{aligned}$$

- edellä hyödynnettiin kaavaa $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ kun $x < 1$.

Kauppamatkustajan ongelma

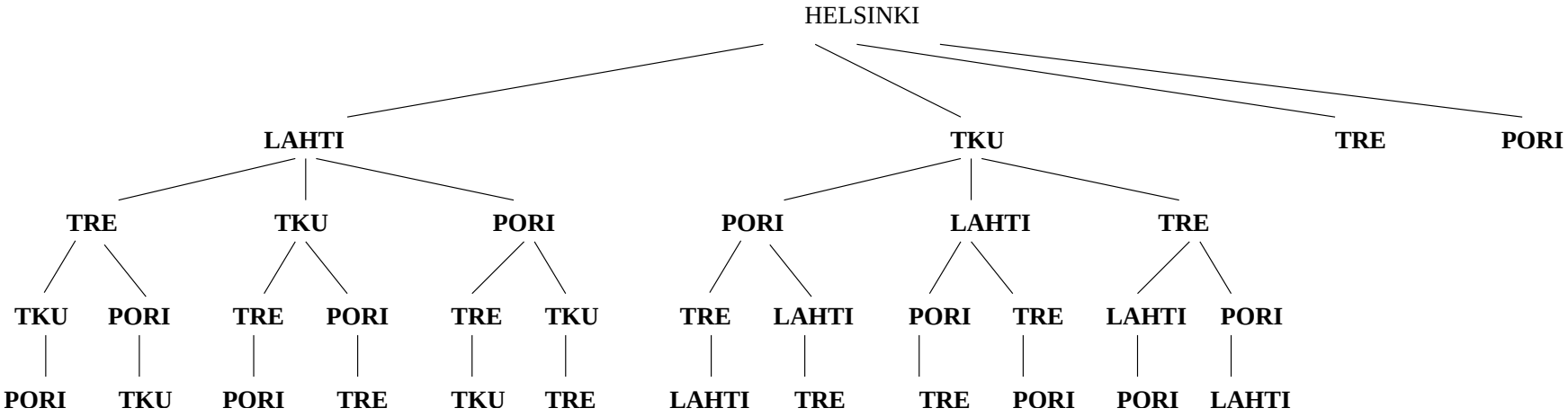
- Helsingissä asuvan kauppamatkustajan täytyy vierailla Lahdessa, Turussa, Porissa ja Tampereella



- kulujen minimointi bisneksessä on tärkeää: mikä on lyhin reitti joka alkaa Helsingistä ja päättyy Helsinkiin ja sisältää yhden vierailun kussakin kaupungissa?

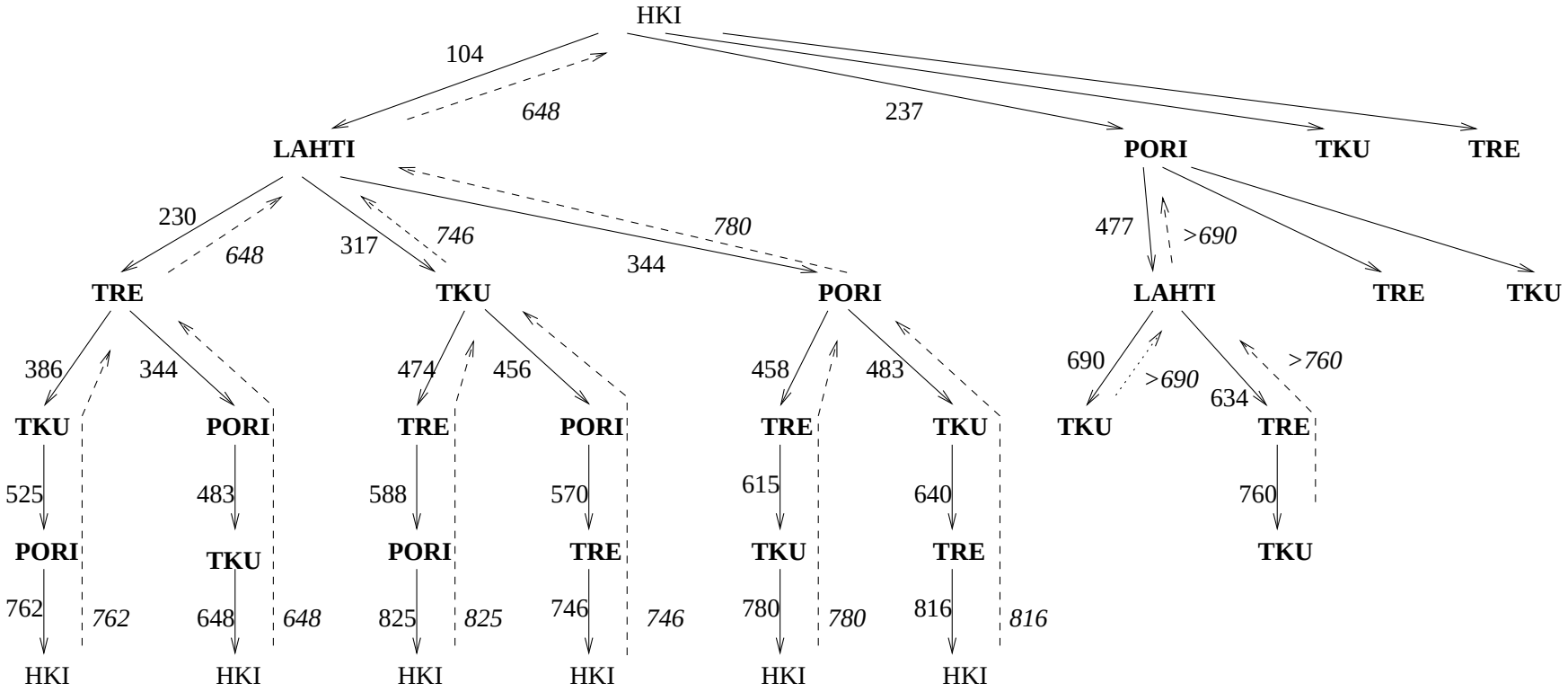
- huomaamme että mahdolliset reitit ovat kaupunkien jonon Turku, Tampere, Pori, Lahti permutaatiot

- voimme siis käyttää ratkaisussa samaa periaatetta kuin permutaatioiden tulostuksessa:



- näin siis saamme systemaattisesti generoitua 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
  - kutsutaan $gen(\infty, 0, visited, 1, 1)$

```
gen(best,length,visited,cur,k)
1   if k == n+1 return length+dist[cur,1]
2   mybest = ∞
3   for i = 2 to n
4       if visited[i] == false
5           vis2 = visited
6           vis2[i] = true
7           if length+dist[cur,i]< best
8               newp = gen(best,length+dist[cur,i],vis2,i,k+1)
9               if newp<mybest mybest = newp
10              if newp<best best = newp
11  return mybest
```

- algoritmin toimintaidea
  - parametrit:
    best parhaan jo löydetyn reitin pituus
    length kuinka pitkä reitin tähän asti tutkittu osa on
    visited missä kaupungeissa on jo käyty
    cur nykyisen kaupungin numero
    k kuinka monessa kaupungissa on jo käyty
  - rivillä 1 huomataan jos koko reitti on jo generoitu: 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 3-4)
  - rekursiokutsu rivillä 8 tutkii mikä on kaupungilla i jatkuvan reitin pituus
  - uutta reittiä ei tutkita jos se on jo tässä vaiheessa toivottoman pitkä
  - lopulta palautetaan lyhimmän löytyneen reitin pituus

- 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 saman pituinen 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

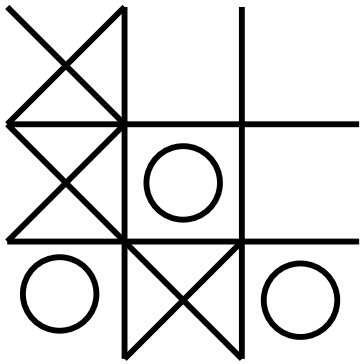
- kauppamatkustajan ongelmalle ei tiedetä parempia kuin eksponentiaalisessa ajassa toimivia ratkaisualgoritmeja

- toisaalta ei ole pystytty todistamaan ettei nopeaa (polynomisessa ajassa toimivaa) algoritmia ole olemassa . . .

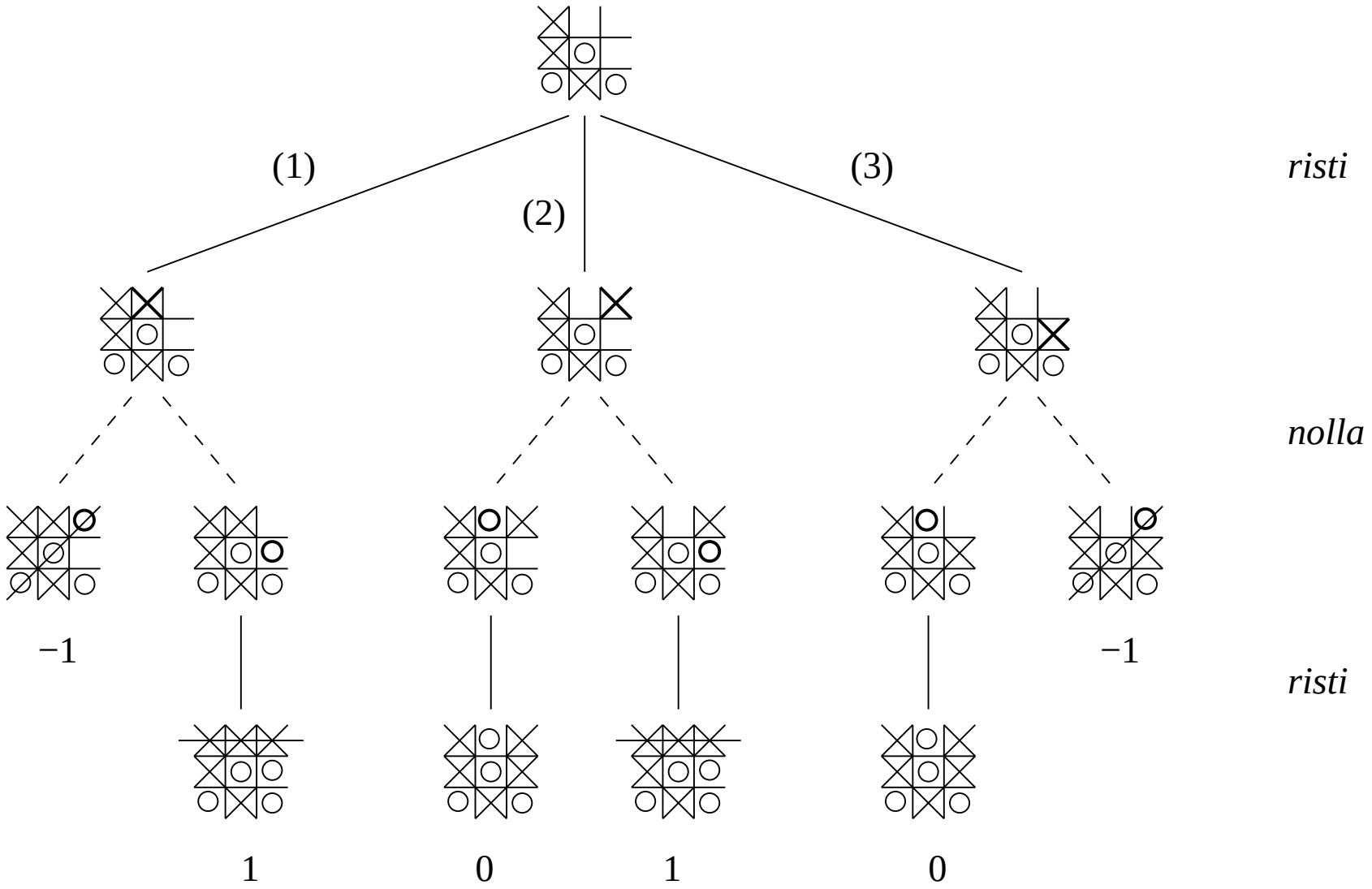
- kyseessä on ns. NP-täydellinen ongelma, aiheesta hieman enemmän kurssilla Laskennan mallit

Pelipuu

- tietokone pelaa risti-nollaa ihmistä vastaan
- on ristin vuoro, mitä tietokoneen kannattaa tehdä seuraavassa tilanteessa?



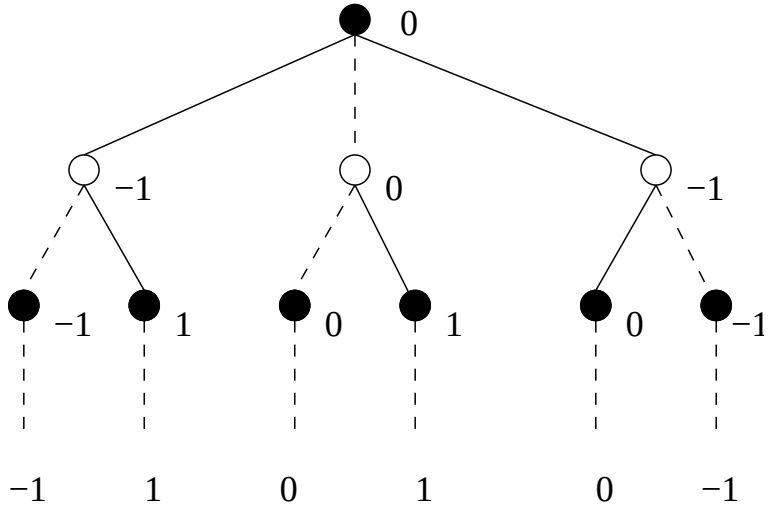
- tietokone rakentaa päätöksensä tueksi *pelipuun*, ks. seuraava kalvo


(1)
(2)
(3)
risti
nolla
risti
−1
−1
1
0
1
0

- tehtävä siis valinta kolmen mahdollisen siirron suhteen
- 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 valkoinen pelaa mahdollisimman hyvin*

- seuraavassa pelipuu piirrettynä hiukan abstraktimmin



- ristin vuoroa vastaavat solmut ovat mustia ja nollan vuoroa vastaavat valkoisia
- pelipuu evaluoida 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
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
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 peleissä, kuten 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, ym . . .)

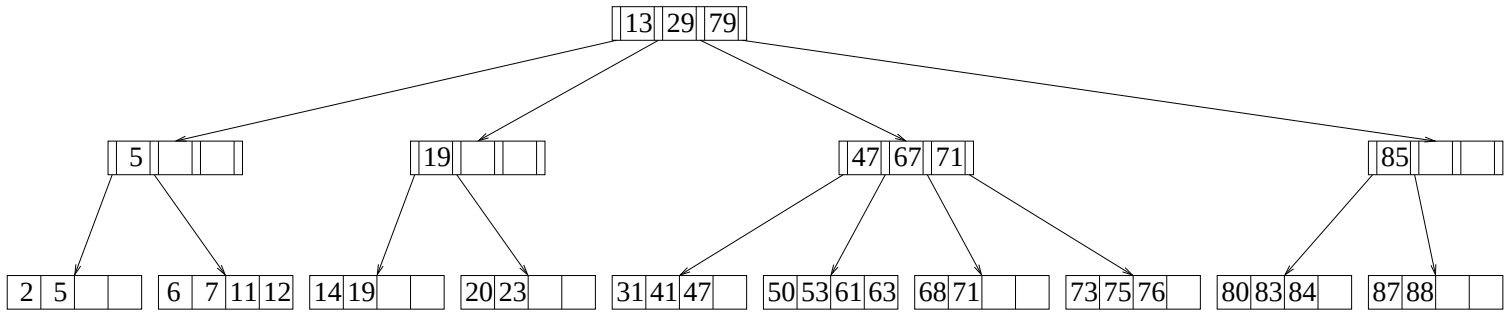
4. Levymuistiin talletetun tiedon organisoiminen

- Tasapainoitetut binäärihakupuut, kuten AVL-puu toimivat hyvin, jos kaikki käsiteltävä tieto mahtuu keskusmuistiin

- Jos käsiteltävä tieto joudutaan pitämään levymuistissa, tilanne muuttuu
  - tiedon hakeminen keskusmuistista on nopeaa, aikaa esim. kokonaisluvun kokoisen tiedon hakemiseen kuluu reilusti alle mikrosekunti
  - tiedon hakeminen levyltä vie huomattavasti kauemmin, esim. kokonaisluvun kokoisen tiedon hakemiseen menee ehkä 10 millisekuntia
  - tiedon haku levyltä voi olla jopa miljoona kertaa hitaampaa kuin haku keskusmuistista
  - levyltä ei kannata hakea pieniä määriä tietoa, sillä käytännössä samassa ajassa saadaan kerralla haettua isompi määrä, esim. neljä kilotavua joka vastaa noin tuhannen int-muuttujan viemää tilaa

- jos tieto talletetaan levymuistiin, on suorituskyvyn kannalta oleellista, että levyltä lukuoperaatioiden määrä on mahdollisimman pieni

- **B-puu** on tasapainoinen hakupuu, joka ottaa huomioon levymuistin erityisluonteen
  - ideana on tehdä puista mahdolisimman matalia, jotta hakupolku olisi mahdolisimman lyhyt
  - puu saadaan matalaksi, jos solmuilla voi olla useita lapsia ja jos yhteen solmuun talletetaan yhden data-alkion sijasta useampia, esim. tuhat data-alkiota
  - koska levyltä kannattaa lukea kerralla suuri määrä dataa, on tälläisen ison solmun lukeminen levyltä suhteessa tehokkaampaa kuin monen pienen solmun yksittäinen lukeminen levyltä
- B-puita tai oikeastaan erästä sen varianttia, B+-puuta käsitellään erillisessä materiaalissa, joka löytyy kurssisivulta
- B+-puu kuuluu toisen välikokeen koealueeseen
- Alla esimerkki B-puusta



5. Hajautus

- Tarkastellaan edelleen sivulla 75 määritellyn joukkotietotyypin toteuttamista
- useissa sovelluksissa riittää että operaatiot *insert*, *delete* ja *search* toimivat nopeasti
  - esim. sivun 226 opiskelijarekisteriesimerkissä on epätodennäköistä että opiskelijanumeron mukaan järjestetyssä indeksirakenteessa on mitään käyttöä operaatioille min, max, succ ja prev
  - myöskään puhelinluettelon numeroiden suhteen järjestetty indeksi ei näitä operaatioita tarvinne
  - sensijaan (opiskelijarekisterissä tai puhelinluettelossa) nimen mukaan järjestetyssä indeksissä operaatioille min ja succ voi olla käyttöä jos on tarvetta esim. listata kaikki nimet aakkosjärjestyksessä
- tapauksissa, joissa insert, delete ja find operaatiot riittävät, *hajautusrakenne* on varteenotettava tapa joukon toteutukselle

- Hajautusrakenteessa insert ja delete toimivat vakioajassa ja search toimii keskinmäärin vakioajassa
  - tasapainoisella puulla kaikki operaatiot pahimmassakin tapauksessa $\mathcal{O}(\log n)$
- pahimmassa tapauksessa hajautusrakenteen seach-operaatio voi viedä aikaa $\mathcal{O}(n)$
- hajautusrakenteen tehokkuuden kannalta erittäin tärkeä seikka on hajautusfunktiolla
- 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 ne vievät vain $O(\log n)$

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

- otetaan käyttöön m-paikkainen *hajautustaulukko* T, joka on indeksoitu alkaen nollasta

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

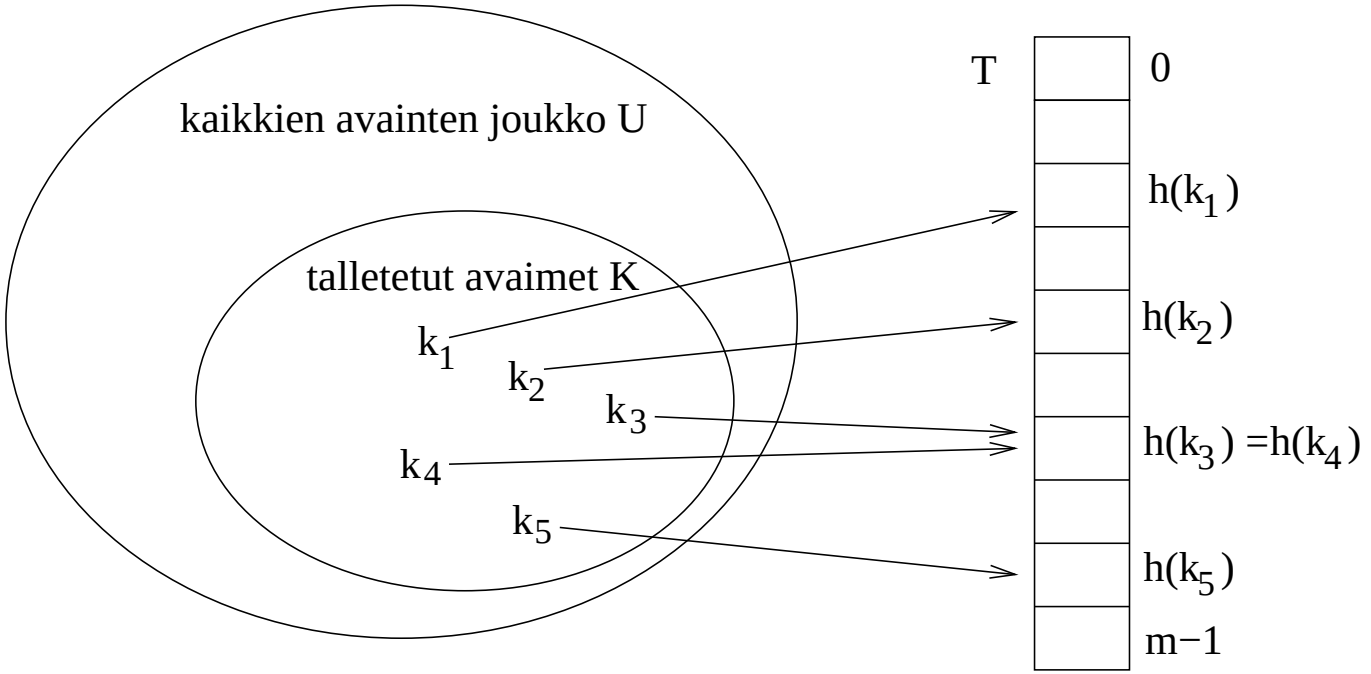
- määritellään *hajautusfunktio* $h : U \to \{0, \ldots, m-1\}$

  eli hajautusfunktio 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 $h(k_i)$

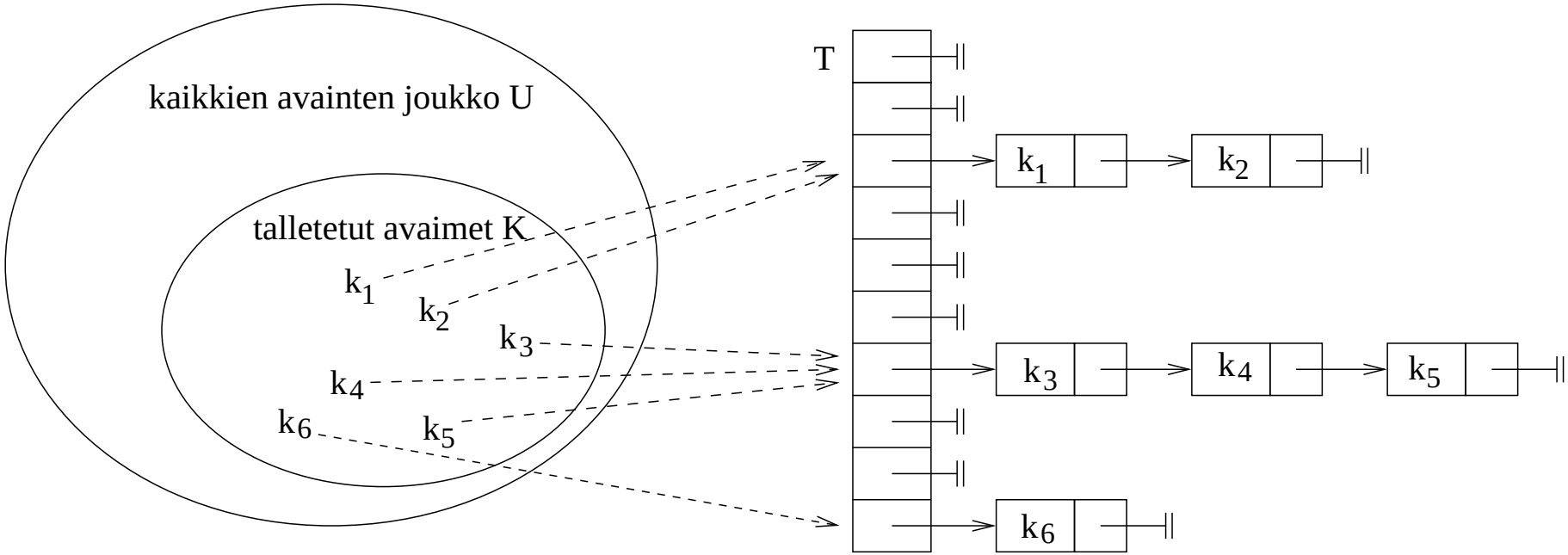


- kuten kuva yllä kertoo, voi käydä niin että kaksi avainta saavat saman osoitteen eli joillekin avaimille k_3, k_4 voi olla $h(k_3) = h(k_4)$
- tällöin sanotaan että avainten k_3 ja k_4 välillä sattuu *yhteentörmäys*
- oleellista onkin että hajautusfunktio h suunnitellaan sellaiseksi että yhteentörmäyksiä tapahtuu mahdollisimman vähän, palataan hieman myöhemmin hajautusfunktion suunniteluun
- oli hajautusfunktio miten hyvä tahansa, ei jokaiselle avaimelle yleensä riitä omaa osoitetta

Yhteentörmäykset ylivuotoketjuun

- yhteentörmäysten ratkaisemiseen on olemassa kaksi erilaista strategiaa, käsitellään ensin **ketjutus**, 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. kalvot 108-113)

- 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 kalvolla 109 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 kalvolla 110 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 kalvolla 112 määriteltyä listan poisto-operaatiota, joka poistaa avaimen listalta

- mikä on operaatioiden aikavaativuus?
- oletetaan että hajautusfunktion arvon laskeminen vie vakioajan
- luvusta 2 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 hajautustauluun on talletettu n avainta, ja että avaimet ovat jakautuneet tasaisesti eri ylivuotolistoille**

  tilanne on yleensä tämä jos käytetty hajautusfunktio on järkevästi valittu
- hajautustaulun tämänhetkinen *täyttösuhde* on $\alpha = n/m$, joka on samalla *keskinmääräinen ylivuotoketjun pituus*
- **Avaimen haku hajautusrakenteesta vie aikaa keskimäärin $\mathcal{O}(1+\alpha)$**

  todistushahmotelma:

  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 α avainta. Kun otetaan vielä huomioon hajautusfunktion laskemiseen kuluva vakioaika saadaan vaatimukseksi $\mathcal{O}(1+\alpha)$.

  Tuloksellisen haun tarkka analyysi vaatii todennäköisyyslaskennan osaamista niin paljon, että sivuutamme sen. Karkeasti ottaen tuloksellisessa tapauksessa joudutaan 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, esim. $m \approx n/3$, silloin on $n = \mathcal{O}(m)$
- tällöin $\alpha = n/m = \mathcal{O}(m)/m = \mathcal{O}(1)$ eli ylivuotoketjujen keskimääräinen pituus on vakio hajautusrakenteeseen talletettavien avaimien lukumäärän suhteen
- kun $\alpha = \mathcal{O}(1)$, haun keskimääräinen aikavaativuus eli $\mathcal{O}(1 + \alpha)$ siis vakio $\mathcal{O}(1)$
- eli jos tiedämme että talletettavia avaimia on korkeintaan esim. kolminkertainen määrä hajautustaulun kokoon nähden, vie hakuoperaatio keskinmäärin ajan $\mathcal{O}(1)$
- **jos hajautustaulukon koko on suoraan verrannollinen talletettavien avaimien lukumäärään ovat operaatiot hajautustaulussa vakioaikaisia**
  - hakuoperaatio search vie keskimäärin vakioajan **jos avaimet ovat jakautuneet tasaisesti ylivuotolistoille**
  - insert ja delete ovat vakioaikaisia myös pahimmassa tapauksessa
- useissa sovelluksissa tiedetään suunnilleen talletettavien avaimien lukumäärän yläraja, tällöin on helppo valita hajautusrakenteen koko sopivasti
  - Helsingin Yliopistossa opiskelijoita noin 30000
  - Suomessa asukkaita reilu 5 miljoonaa
  - ...

Hajautusfunktion valinta

- tavoitteena on että hajautusfunktion arvo tulee olla laskettavissa nopeasti ja että funktio jakaa avaimet tasaisesti hajautusalueelle
- hajautusfunktio olettaa yleensä että avain on kokonaisluku, jos avain on kuitenkin esim. merkkijono, on se 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

  $k = ascii(a_1) + 128 \times ascii(a_2) + 128^2 \times ascii(a_3) + \ldots + 128^{n-1} \times ascii(a_n)$
- 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**
  - $h(k) = k \ mod \ m$

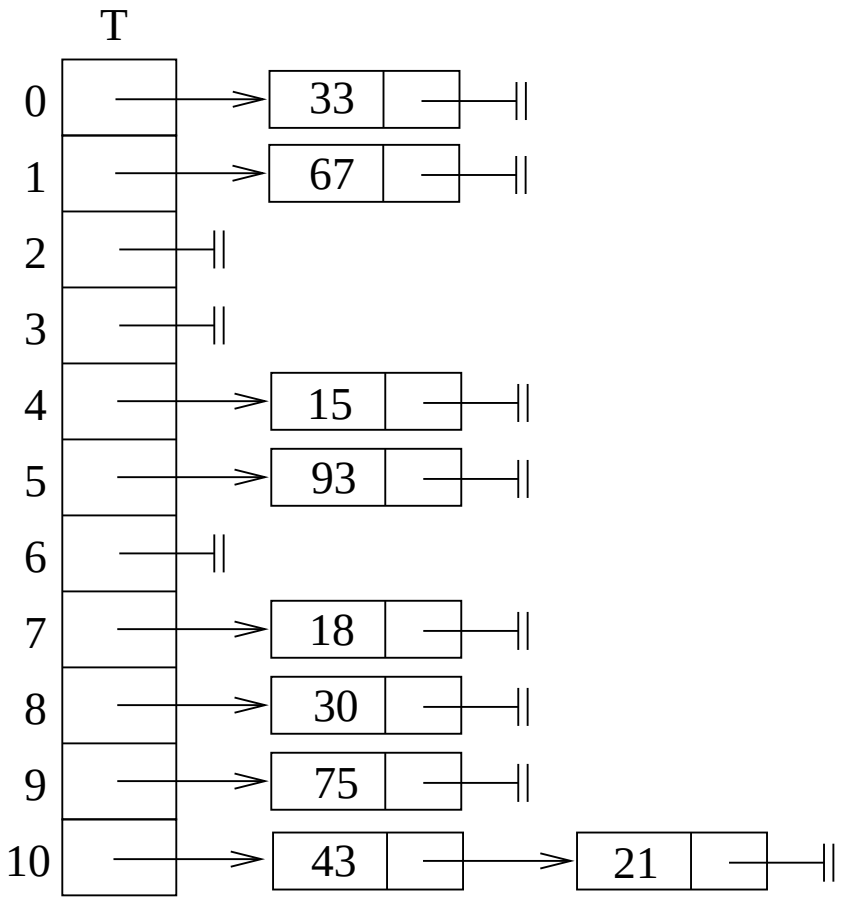
    missä m on alkuluku, joka ei ole lähellä mitään 2:n potenssia

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

    jos m olisi jaollinen luku eli $m = rs$ jollakin r ja s ja talletettavat avaimet (tai suuri osa niistä) sattuisi olemaan r:n monikertoja eli muotoa ir jollakin i, niin jakojäännöksellä olisi vain s eri arvoa ja koko hajautustaulukko ei täyttyisi. m:n siis kannattaa olla alkuluku. Tarkempi analyysi paljastaa vielä, että valinta on parempi jos m ei ole lähellä mitään 2:n potenssia.
  - esim. jos haluamme hajautustaulun johon talletetaan noin 2000 avainta, ja sallimme että ylivuotolistoilla on keskimäärin 3 avainta, valitaan $m = 701$
  - nyt m on alkuluku joka ei ole lähellä 2:n potenssia ja täyttöasteeksi tulee $2000/701 \approx 2.85$
  - esim2.: oletetaan että käytössä hajautustaulukko jonka koko 11, hajautusfunktiona siis $h(k) = k \ mod \ 11$
  - hajautetaan avaimet 75, 21, 43, 15, 18, 33, 30, 67 ja 93, eli nyt

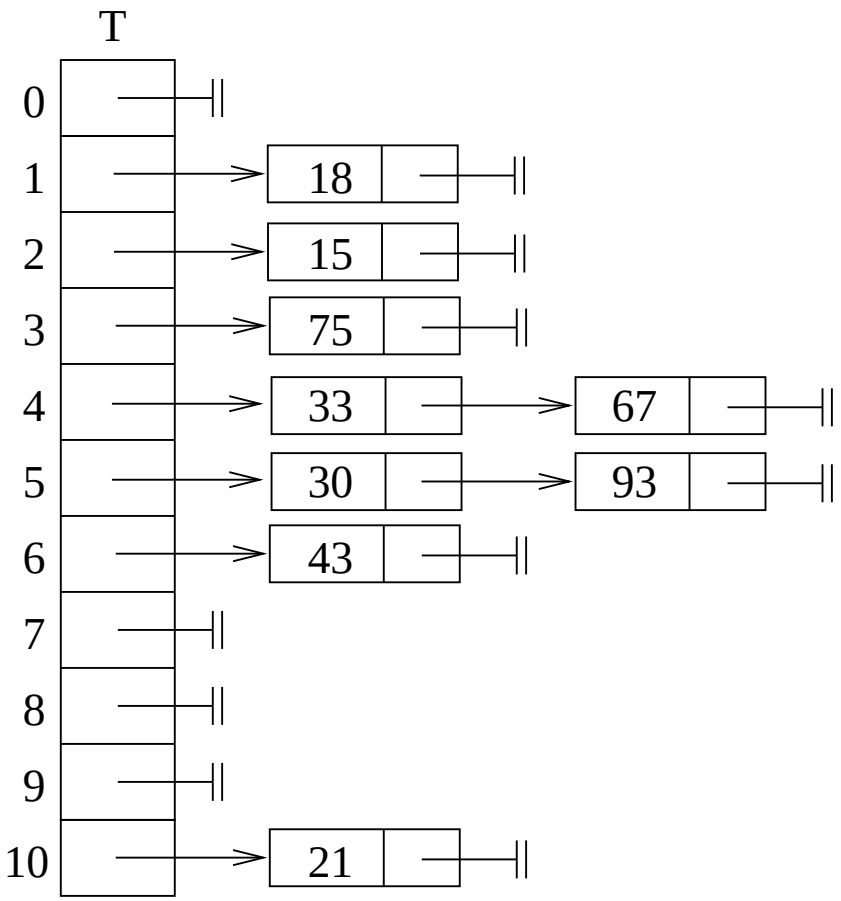
    $h(75) = 9, \quad h(21) = 10, \quad h(43) = 10,$
    $h(15) = 4, \quad h(18) = 7, \quad h(33) = 0,$
    $h(30) = 8, \quad h(67) = 1, \quad h(93) = 5$

- 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 $fr(3.14159) = 0.14159$
  - $h(k) = \lfloor m \times fr(A \times k) \rfloor$
  - etuna jakolaskumenetelmään se että taulukon koon m saa nyt valita vapaasti
  - Asiantuntijoiden mukaan usein toimiva valinta on $A = \frac{\sqrt{5}-1}{2} = 0.6180339887\ldots$
  - esim: oletetaan että käytössä hajautustaulukko jonka koko 11, ja hajautusfunktiona $h(k) = \lfloor 11 \times fr(0.618 \times k) \rfloor$
  - avaimet 75, 43, 21, 15, 18, 33, 30, 67 ja 93, nyt
    $h(75) = 3, \quad h(43) = 6, \quad h(21) = 10$
    $h(15) = 2, \quad h(18) = 1, \quad h(33) = 4,$
    $h(30) = 5, \quad h(67) = 4, \quad h(93) = 5$

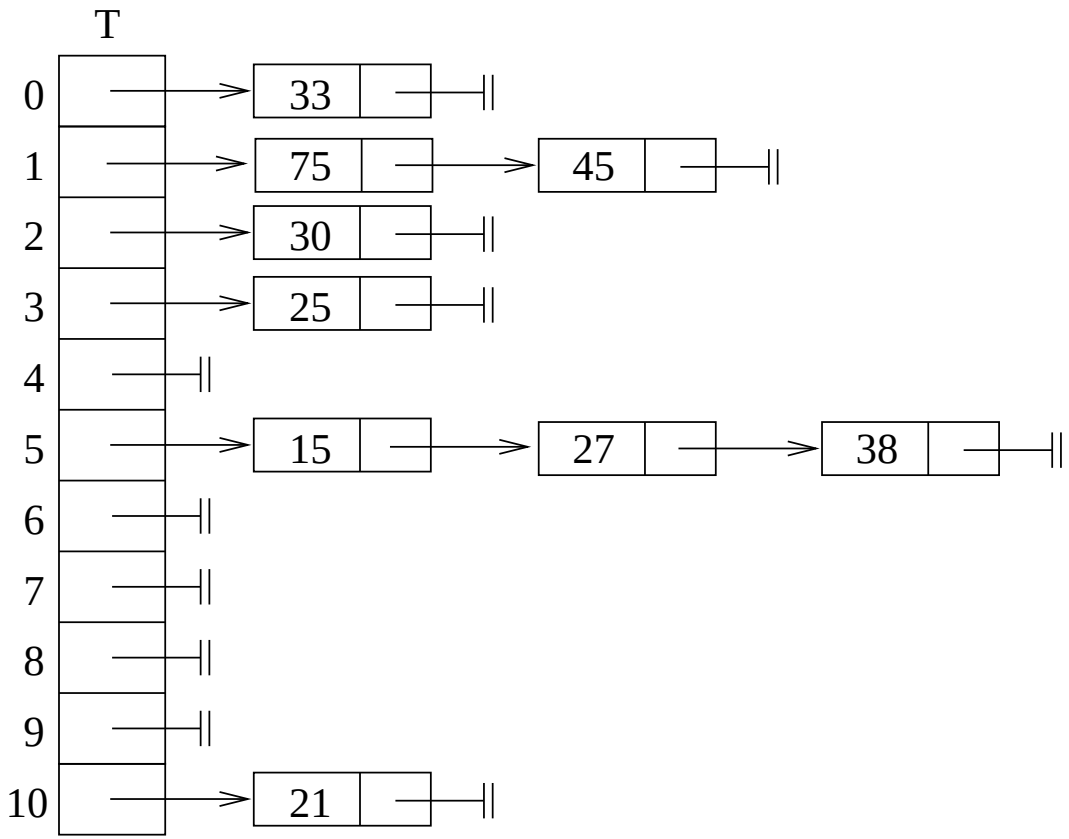
- tulos seuraavalla sivulla


T
0
1
18
2
15
3
75
4
33
67
5
30
93
6
43
7
8
9
10
21

- 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:
    $h(k) = ((a \times k + b) \ mod \ p) \ mod \ m$
  - perusteluja näin muodostetun hajautusfunktion hyvyydestä löytyy Cormenin luvusta 11.3.3
  - esim: oletetaan jälleen että käytössä hajautustaulukko jonka koko 11
  - oletetaan että talletettavia avaimia korkeintaan 53 eli valitaan $p = 53$, ja valitaan satunnaisesti $a = 31$, $b = 17$
    hajautusfunktiona siis $h(k) = ((31 \times k + 17) \ mod \ 53) \ mod \ 11$
  - avaimet 75, 43, 21, 15, 18, 33, 30, 67 ja 93, nyt
    $h(75) = 10 \ mod \ 11 = 10$, $h(43) = 25 \ mod \ 11 = 3$
    $h(21) = 32 \ mod \ 11 = 10$, $h(15) = 5 \ mod \ 11 = 5$
    $h(18) = 45 \ mod \ 11 = 1$, $h(33) = 33 \ mod \ 11 = 0$
    $h(30) = 46 \ mod \ 11 = 2$, $h(67) = 27 \ mod \ 11 = 5$ ja
    $h(93) = 38 \ mod \ 11 = 5$

- tuloksena



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

Avoin hajautus

- Ketjutuksen rinnalla toinen tapa ratkaista yhteentörmäävien solmujen ongelma on *avoin hajautus* (eng. 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

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

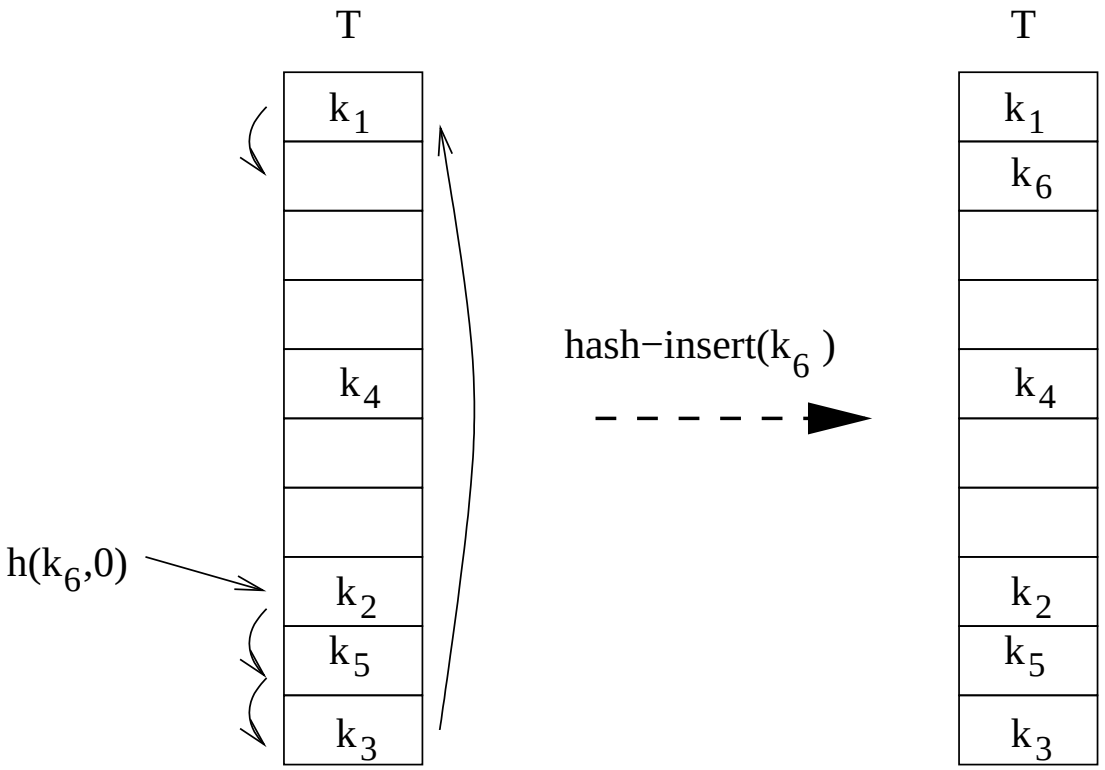
  missä h' on "normaali" hajautusfunktio ja s on kokeilufunktio
- vaaditaan että jokaiselle avaimelle k kokeilujono $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ muodostaa jonon $0, 1, \ldots, (m-1)$ permutaation, eli toisin sanoen kokeilujonon on kokeiltava jokaista hajautustaulun indeksiä kertaalleen
- avaimen 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*
  - kokeilujono on nyt $h(k, j) = (h'(k) + j)\ mod\ m$
  - eli ensimmäinen paikka on $h'(k)\ mod\ m$, toinen $(h'(k) + 1)\ mod\ m$, kolmas $(h'(k) + 2)\ mod\ m$
  - 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 TRUE, mutta jos mikään kokeilujonon paikka ei ole vapaana, eli hajautustaulu on jo täysi, palauttaa operaatio FALSE

- operaation toiminta:

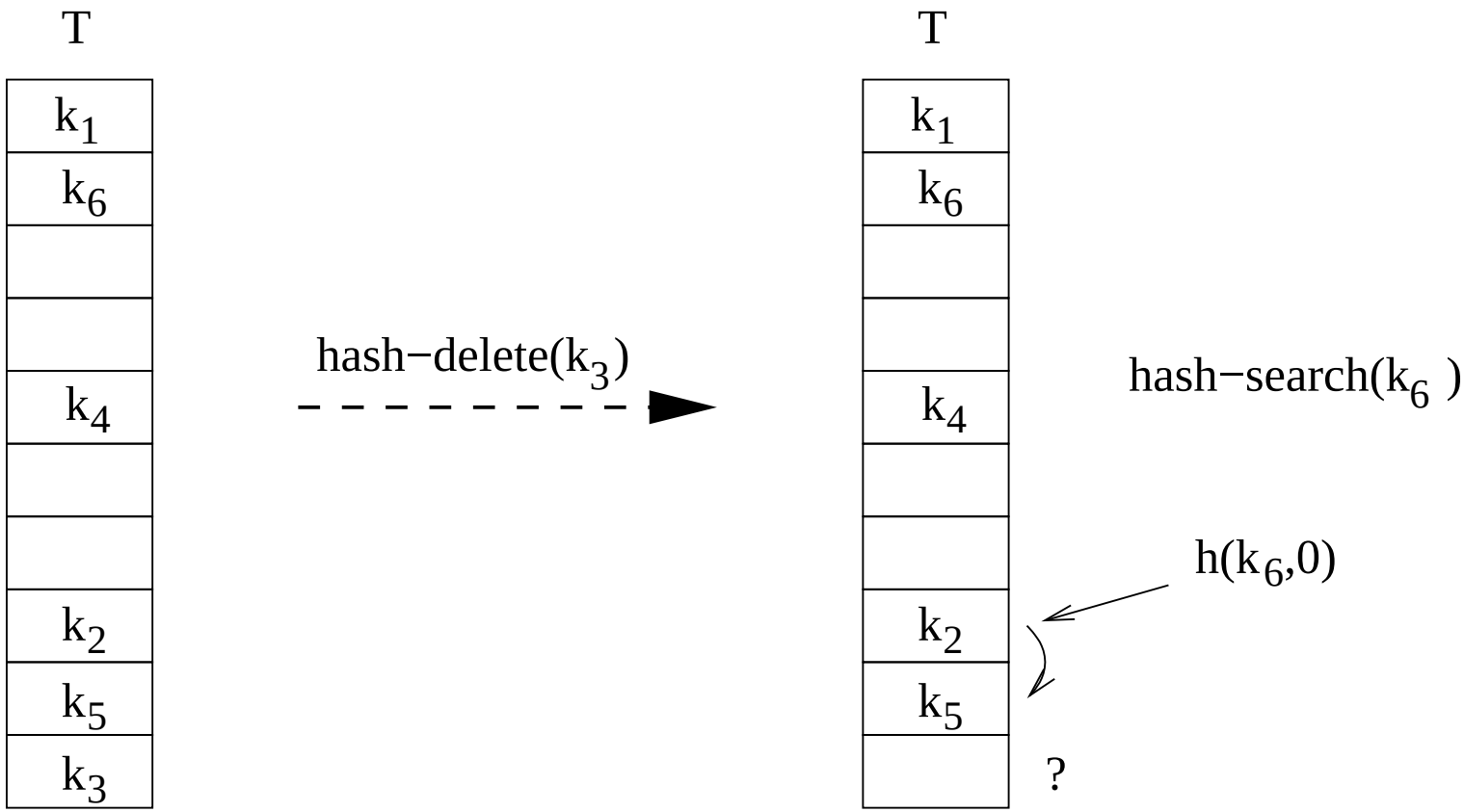


- avaimen etsiminen hajautustaulusta

```
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
```

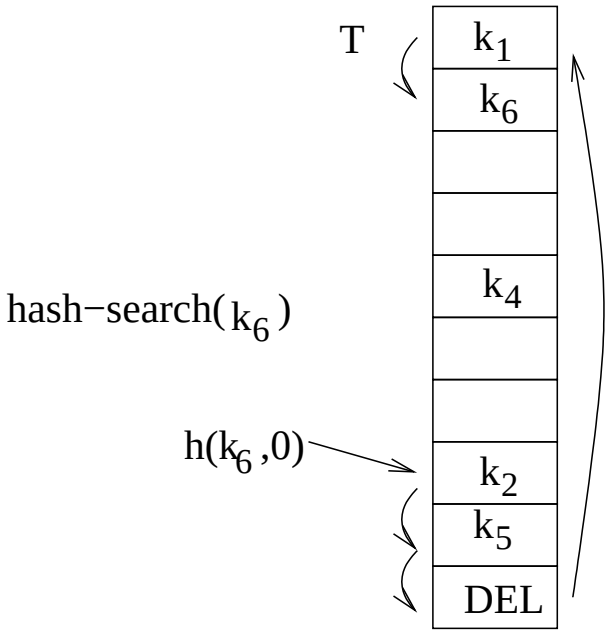
- etsitään kokeilujonoa $h(k,0), h(k,1), \ldots, h(k,m-1)$ läpi niin kauan kun
  - etsitty avain löytyy,
  - törmätään tyhjään hajautustaulun paikkaan, tai
  - koko kokeilujono on käyty läpi
- avaimen löytyessä palautetaan avaimen tallettavan taulukon paikan indeksi, muuten NIL

- 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 insertin yhteydessä

- esim: oletetaan että käytössä hajautustaulukko jonka koko 11
- käytetään kokeilujonofunktiota $h(k,i) = ((k \ mod \ 11) + i) \ mod \ 11$ 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
  $h(21,1) = ((21 \ mod \ 11) + 1) \ mod \ 11 = 0$ 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? $h(29,0) = 7$ 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

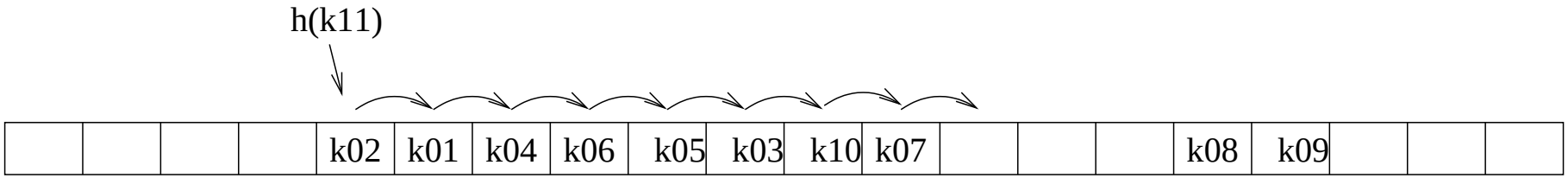
- Edelleen, luvun 41 etsintä etenisi paikasta $h(41, 0) = 7$ aina paikkaan $h(41, 7) = 3$ asti jonka jälkeen voidaan todeta että 41 ei ole taulukossa

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

- 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:

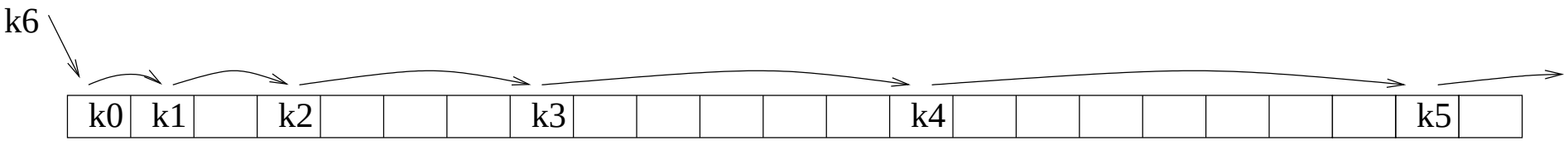


- ilmiötä nimitetään *primääriseksi kasautumiseksi* (eng. primary clustering)

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

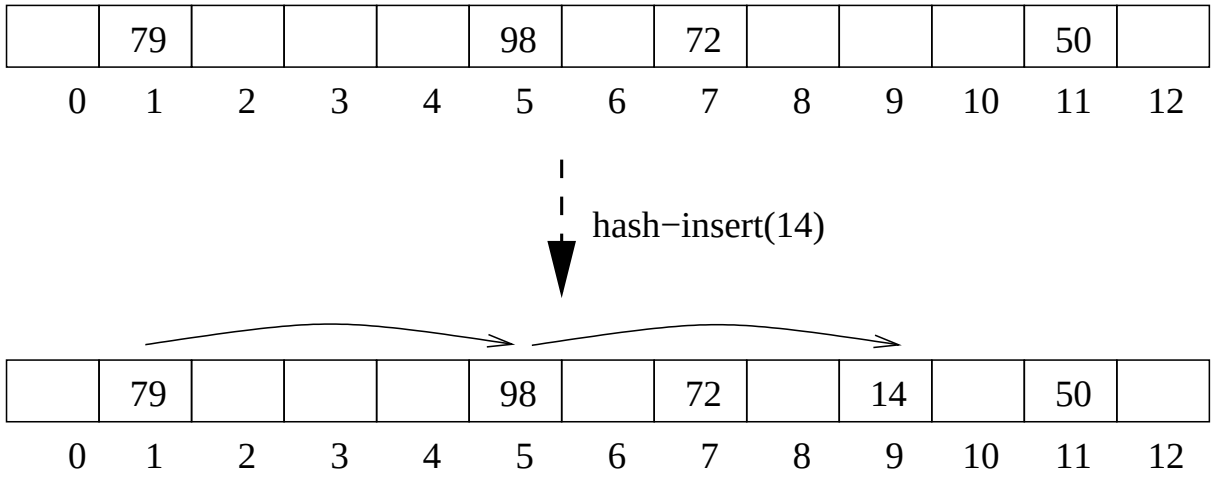
    yksi toimiva tapa valita vakiot on asettaa $c_1 = c_2 = 1/2$ ja valita hajautustaulun kooksi m joku kahden potenssi
  - esim jos $c_1 = c_2 = 1/2$ ja $h'(k_1)$=1 niin kokeilujonon alku olisi $1, 2, 4, 7, 11, 16 \ldots$

    eli $h(k,1) = h(k,0) + 1$, $h(k,2) = h(k,1) + 2$, $h(k,3) = h(k,2) + 3$. ...
- primääriseltä kasaantumiselta vältytään, mutta nyt vaivana on *sekundäärinen kasautuminen*: jos $h(k_1,0) = h(k_2,0)$ niin k_1:n ja k_2:n kokeilujono on sama

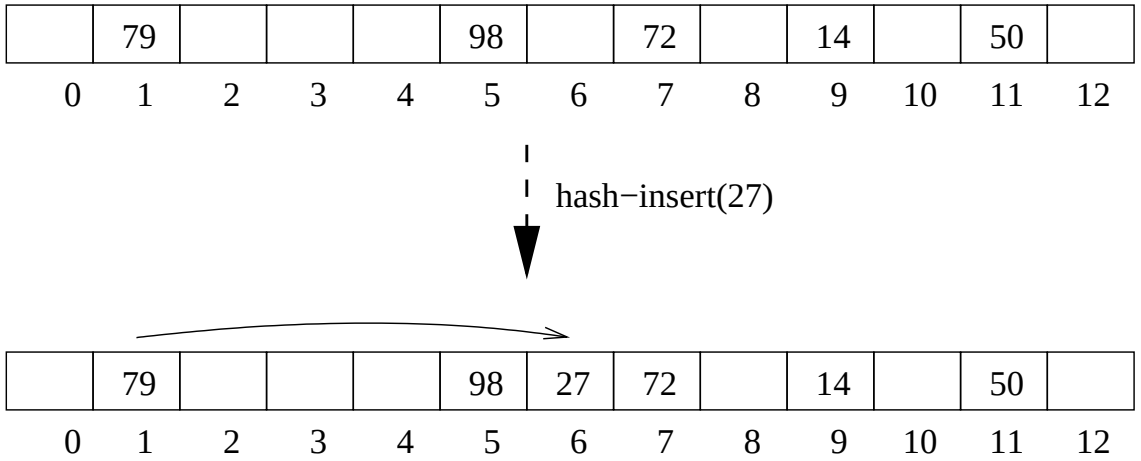


h(k0,0) = h(k1,0) = h(k2,0) = h(k3,0) = h(h4,0) = h(k5,0) = h(k6,0) = 1

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



- lisätään hajautustauluun avain 27
  - $h'(27) = 27 \ mod \ 13 = 1$, $h''(27) = 1 + (27 \ mod \ 11) = 5$,
  - 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 $h''(x) < m$, eli valitaan esim

  $h'(k) = k \ mod \ m$ ja $h''(k) = 1 + (k \ mod \ m')$, missä $m' < m - 1$, vaikkapa $m' = m - 2$

- 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 että täyttöasteeseen 0.5 asti suurta eroa menetelmien välillä ei ole
- tätä täydemmillä hajautustauluilla lineaarista kokeilujonoa ei kannata käyttää

- yhteentörmäysten ketjutusratkaisussa hajautusrakenne toimii hajautusalueen täytyttyäkin vaikkakin toiminta hidastuu pitenevien ylivuotoketjujen takia
- avoimessa hajautuksessa lisäys täyteen hajautusrakenteeseen on mahdotonta
- hajautusrakenteen täyttymisongelma voidaan ratkaista *uudelleenhajautuksella*
  - täyteen rakenteeseen tehtävän lisäyksen yhteydessä rakenteelle varataan uusi kooltaan kaksinkertainen tila
  - kaikki alkuperäisen rakenteen avaimet talletetaan uuteen hajautusrakenteeseen käyttäen uutta hajautusfunktiota
  - uuden hajautusrakenteen täyttösuhde on $1/2$
  - jos poiston jälkeen rakenteen täyttösuhde on alle $1/4$ 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 $1/2$
  - molemmissa tapauksissa uudelleenhajautuksen aikavaatimus on $\mathcal{O}(m)$, missä m alkuperäisen hajautusrakenteen koko
- uudelleenhajautus voidaan toki tehdä aiemminkin kuin taulukon tullessa täyteen, esim. jos täyttösuhde on 3/4

- Monisteen sivuilla 86 ja 87 ja viikon 4 harjoitusten tehtävässä 2 tarkasteltiin taulukon laajentamisen aikavaativuutta

- Ilmeni, että jos taulukon koko 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 uudelleenhajautksesta 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 uudelleehajautus kannattaa jos hajautusrakenteeseen talletettavien alkioiden lukumäärällä ei ole ylärajaa

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
  - ketjutus toimii hyvin vaikka hajautusalue onkin täynnä
  - poistot ketjutuksessa ovat aitoja toisin kuin avoimessa hajautuksessa
- avoin hajautus on hyvä jos täyttösuhde säilyy pienenä
- 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 tasapainoitettua hakupuuta
- hash-search-operaatio toimii vakioaikaisesti keskimääräisessä tapauksessa, eli jos halutaan takuu että pahin tapaus ei koskaan toteudu, on syytä käyttää tasapainoitettua hakupuuta

6. Keko

- Tarkastellaan vielä yhtä tapaa toteuttaa sivulla 75 määritelty tietotyyppi joukko
- tällä kertaa emme kuitenkaan toteuta normaalia operaatiorepertoaaria, 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 $\mathcal{O}(\log n)$
  - kuten viikon 6 laskareiden viimeinen tehtävä osoitti, voitaisiin tasapainoitetun hakupuun avulla toteuttaa heap-max vakioajassa muokkaamalla hieman puun rakennetta
- 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
- seuraavalla sivulla muutama esimerkki keon sovelluksista

- 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 erittäin tehokkaan järjestämisalgoritmin
  - 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
  - palaamme tarkemmin kekojärjestämiseen luvussa 7, varsinainen kekojärjestäminen tapahtuu hieman toisen periaatteen mukaan
- myös verkkoalgoritmien yhteydessä (luvussa 8) löydämme käyttöä keolle

Maksimikeon toteuttaminen

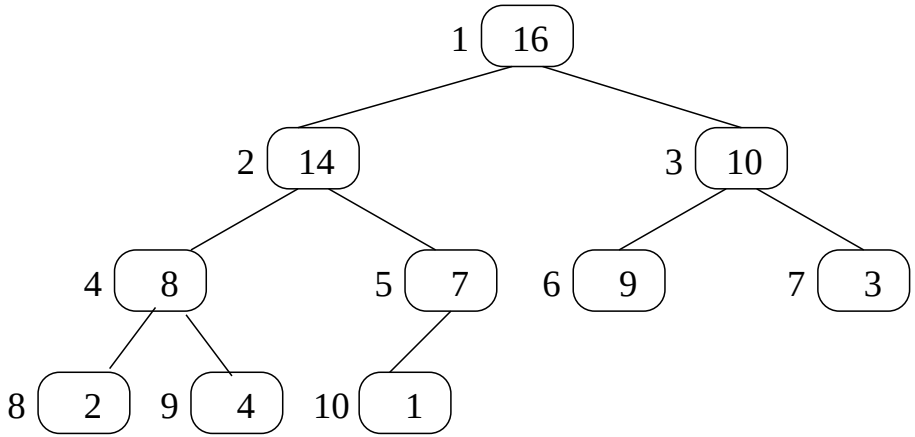
- keko kannattaa ajatella binääripuuna, joka on talletettu muistiin taulukkona

- binääripuu on keko 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

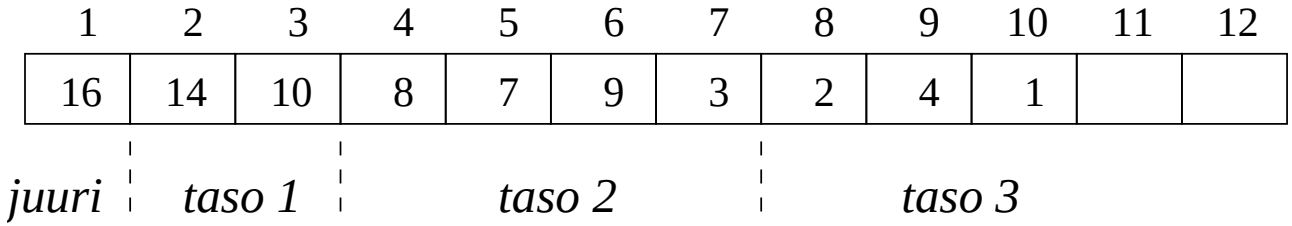
  (K2) jokaiseen solmuun talletettu arvo on suurempi tai yhtäsuuri kuin solmun lapsiin talletetut arvot

- seuraava binääripuu on keko



- 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-size kertoo montako taulukon paikkaa (alusta alkaen) kuuluu kekoon
  - huom: taulukossa A voi siis kohdan heap-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 ⌊i/2⌋

left(i)
    return 2i

right(i)
    return 2i+1
```

- vaikka varsinaisia viitteitä ei ole, on keossa liikkuminen todella helppoa,
  - tarkastellaan edellisen sivun esimerkkitapausta
  - juuren $A[1]$ vasen lapsi on paikassa $A[2 * 1] = A[2]$ ja oikea lapsi paikassa $A[2 * 1 + 1] = A[3]$
  - solmun $A[5]$ vanhempi on $A[\lfloor 5/2 \rfloor] = 2$, vasen lapsi $A[10]$ mutta koska heap-size[A]=10, niin oikeaa lasta ei ole

- voimme lausua kekoehdon **(K2)** nyt seuraavasti:

  kaikille $1 < i \leq$ heap-size pätee $A[parent(i)] \geq A[i]$

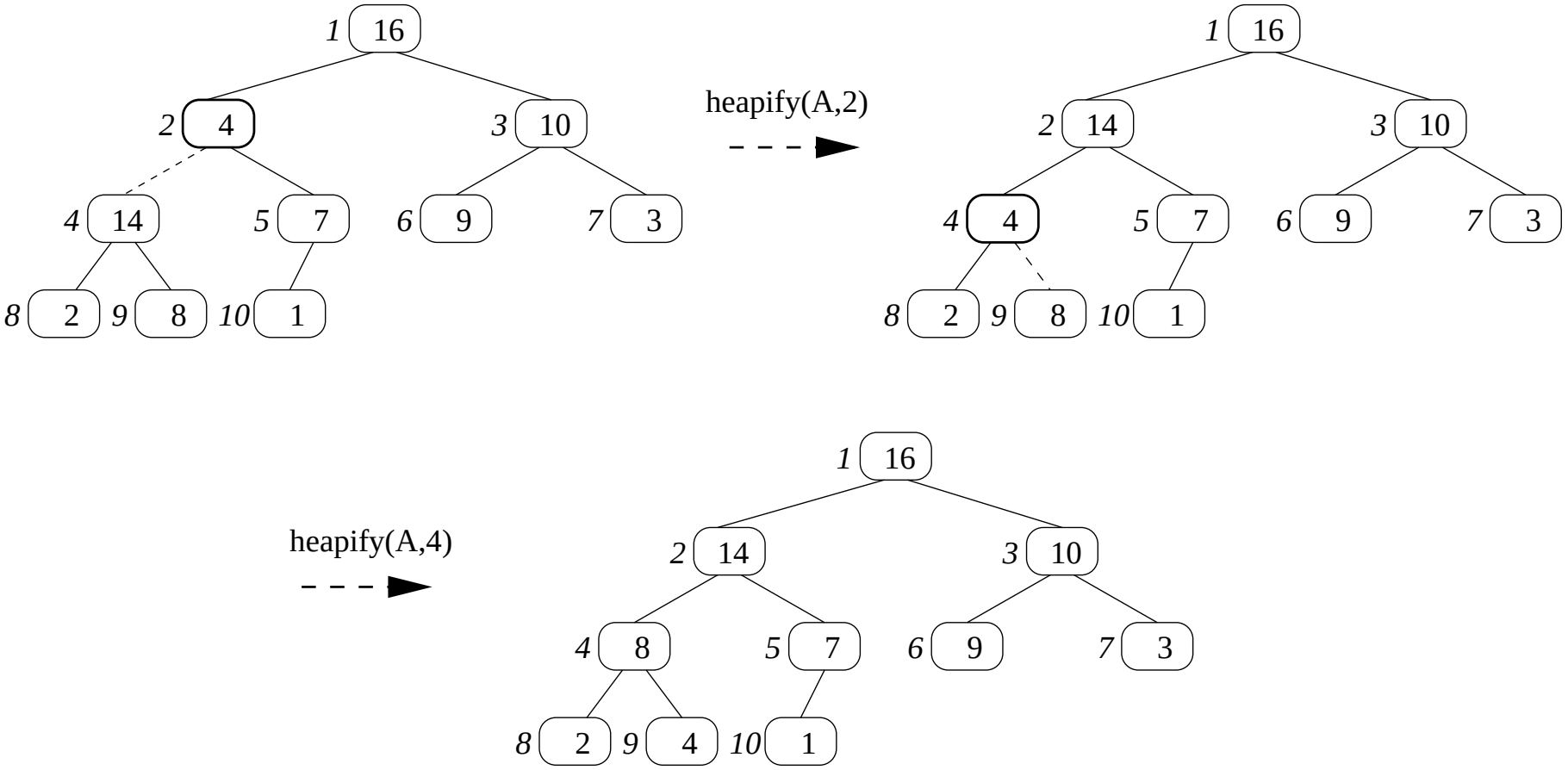
- Ennen varsinaisia keko-operaatioita, toteutetaan tärkeä apuoperaatio heapify
- Operaatio korjaa kekoehdon, jos se on rikki solmun i kohdalla
- Oletus on, että solmun i vasen ja oikea alipuu toteuttavat kekoehdon
- Kutsun heapify(A,i) jälkeen koko solmusta i lähtevä alipuu toteuttaa kekoehdon
- toimintaperiaate on seuraava:
  - Jos A[i] on pienempi kuin toinen lapsistaan, vaihdetaan se suuremman lapsen kanssa
  - Jatketaan rekursiivisesti muuttuneesta lapsesta

- heapifyn parametreina taulukko A ja indeksi i
  - oletuksena siis on että left(i) ja right(i) viittaavat jo kekoja olevien alipuiden A[left(i)] 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 (rivi 4) $A[i]$: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]
```

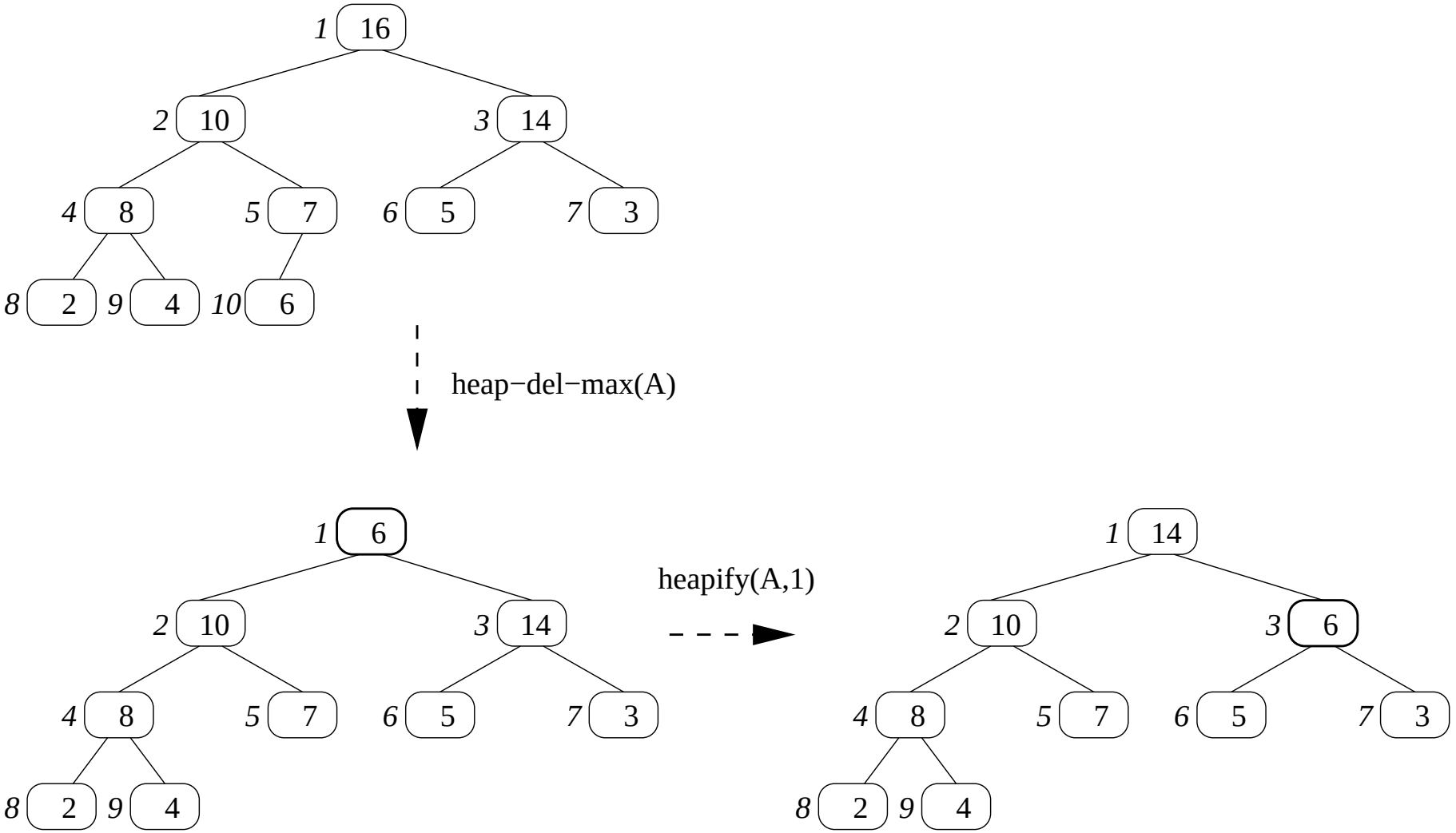
- alkion poistaminen keosta

```
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-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 heapifyllä, eli $\mathcal{O}(\log n)$

- esimerkki heap-del-max operaation toiminnasta

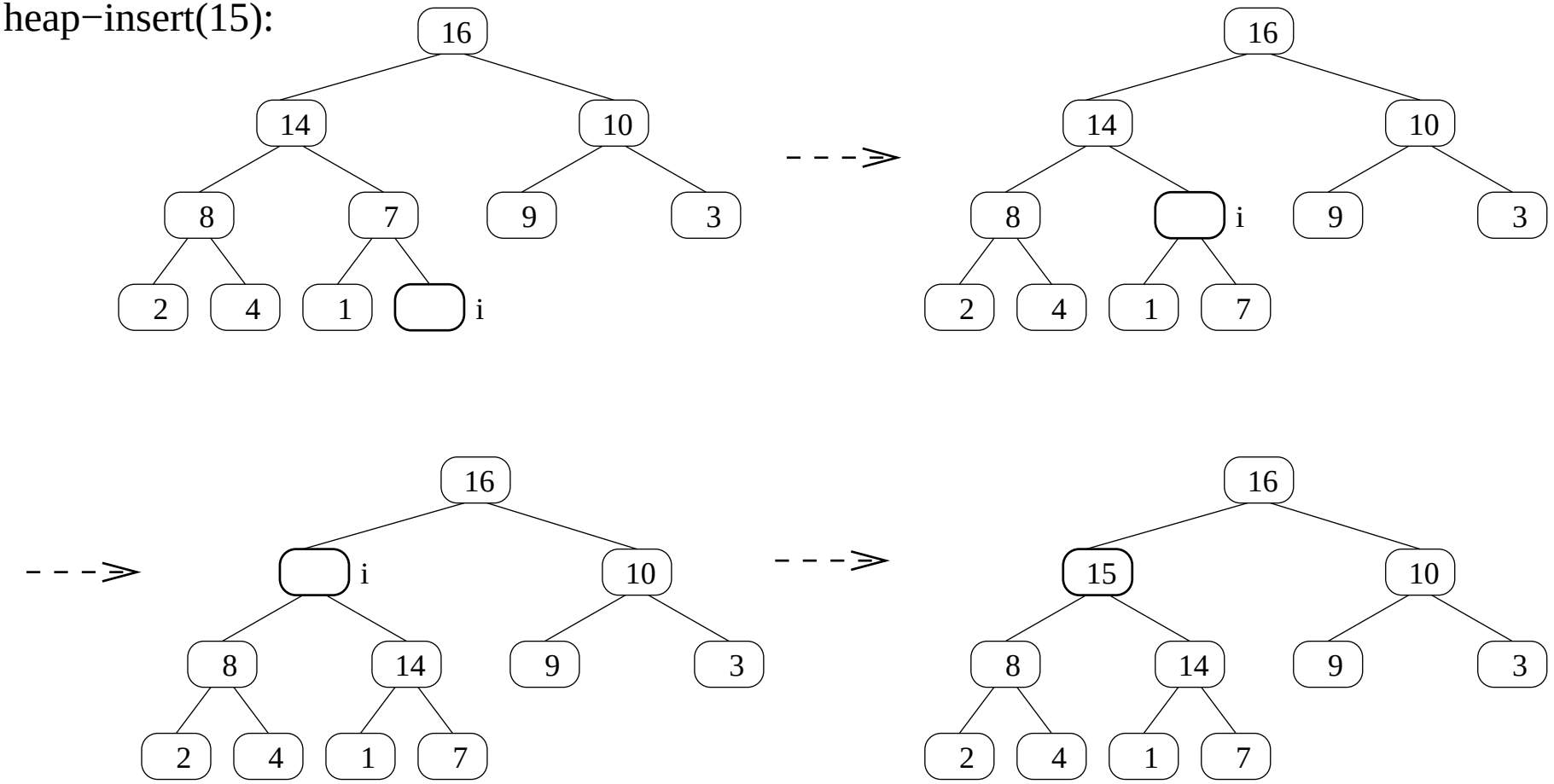


- alkion lisääminen kekoon

```
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):

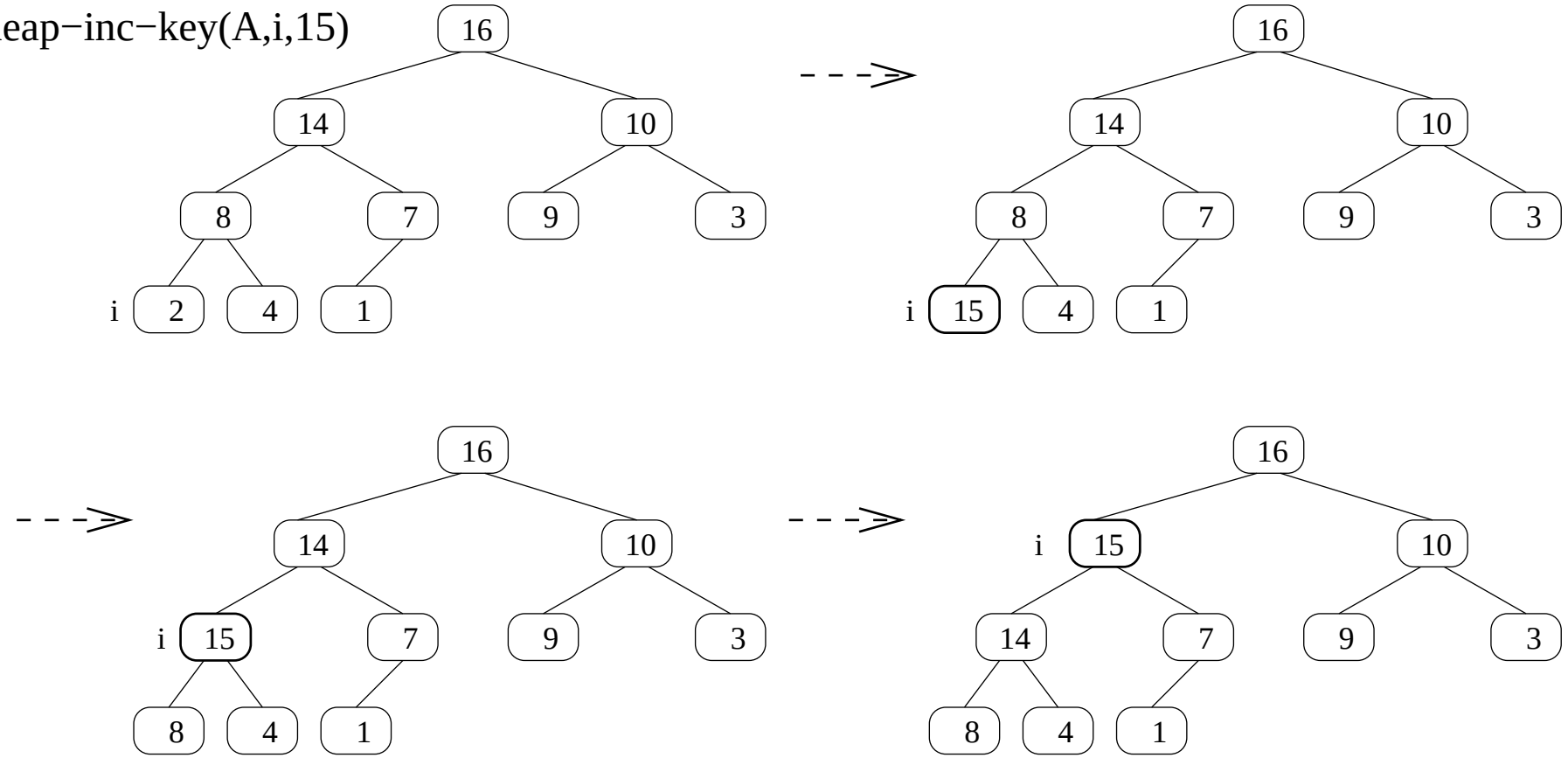


- jotkut sovellukset tarvitsevat keko-operaatiota, joka kasvattaa annetussa indeksissä olevan avaimen arvoa

```
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)



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

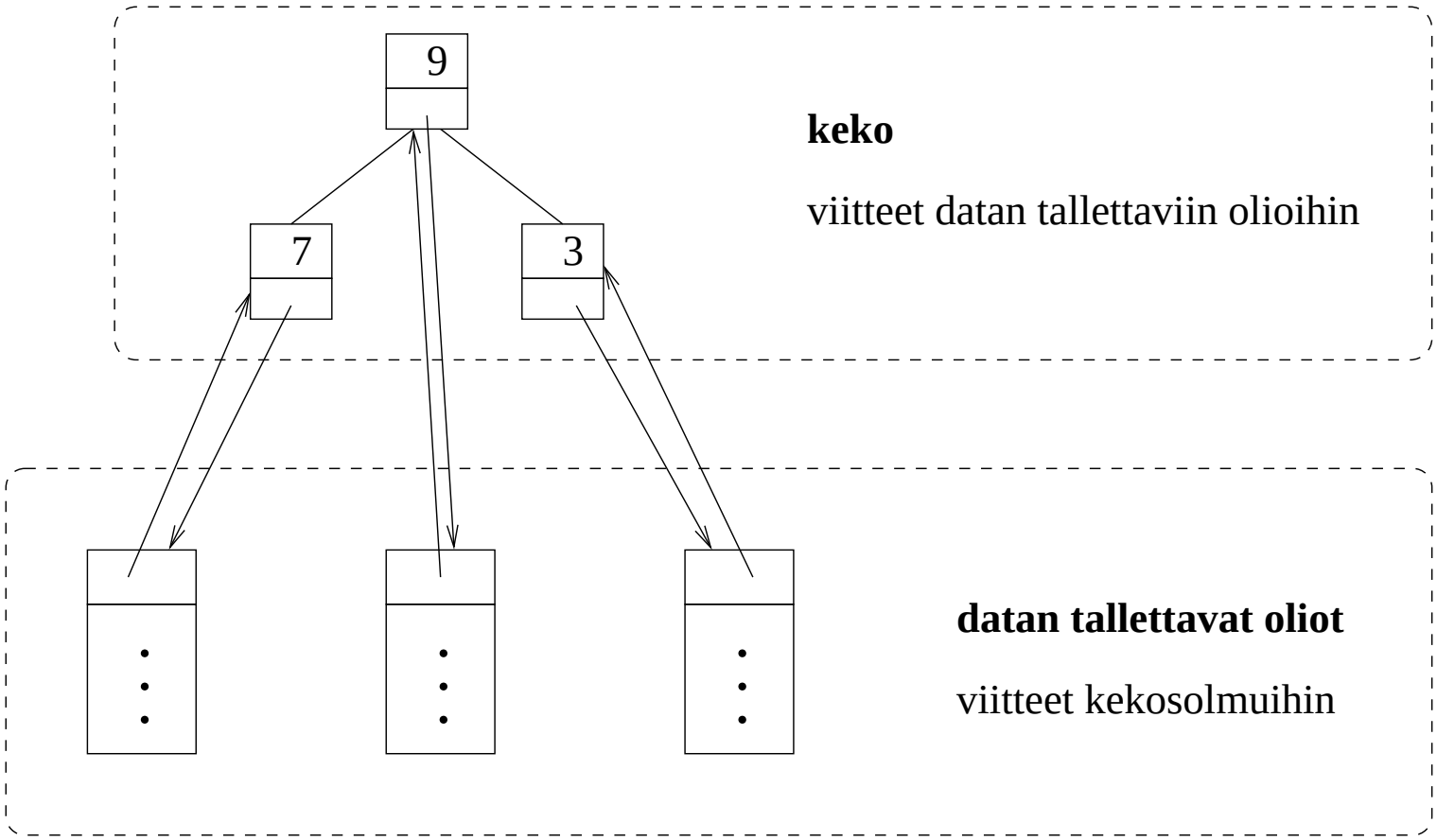
- tällaisessa käytännön tilanteessa keko-operaatioiden parametrit kannattanee valita seuraavasti

- **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 olion liittyvän avaimen keosta
- **heap-inc-key(x,newk)**	kasvattaa 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 ovat siis ovat kekotaulukon indeksejä eli kokonaislukuja

- muistin organisointi näyttää esim. seuraavalta:



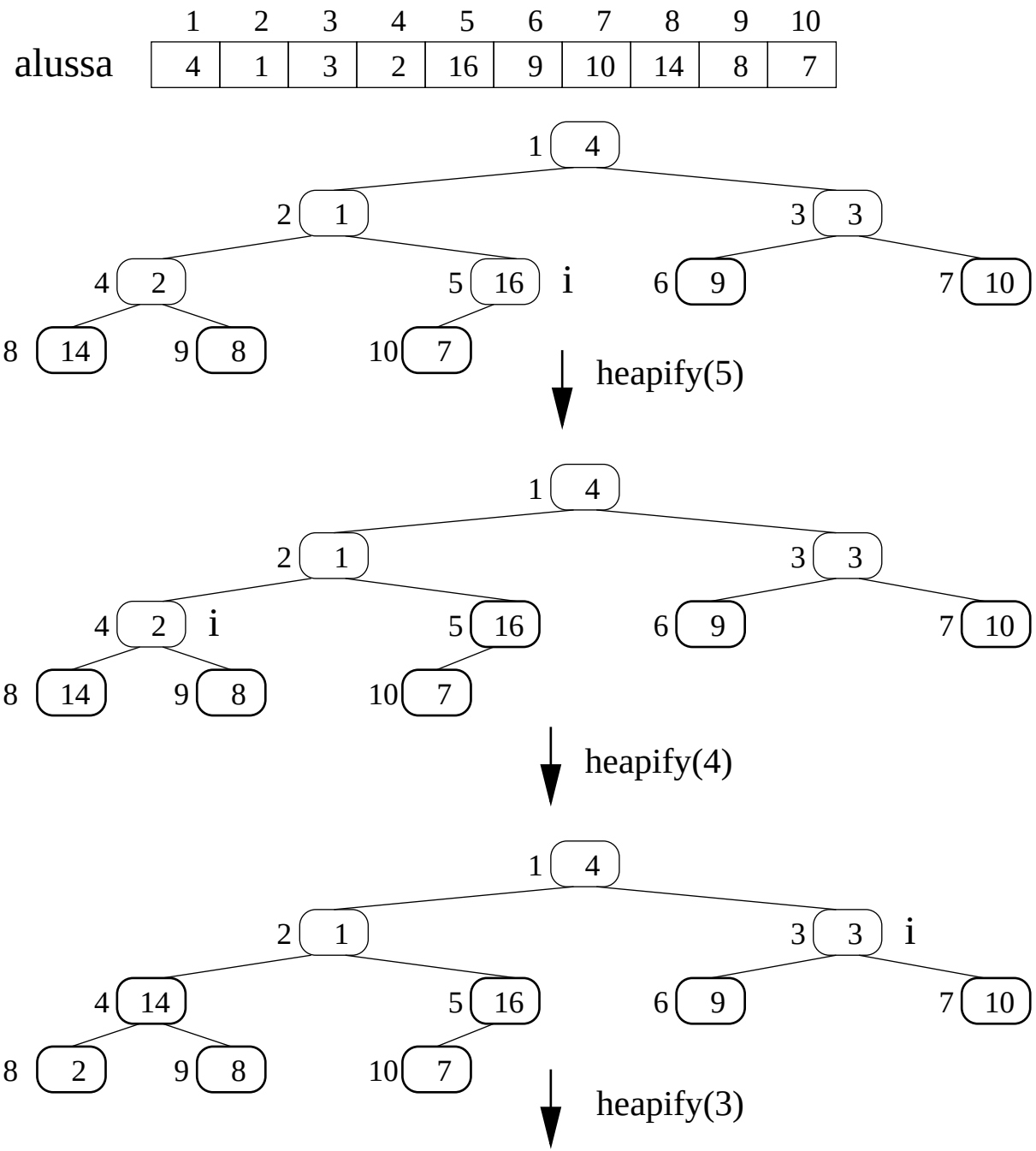
7. Järjestäminen

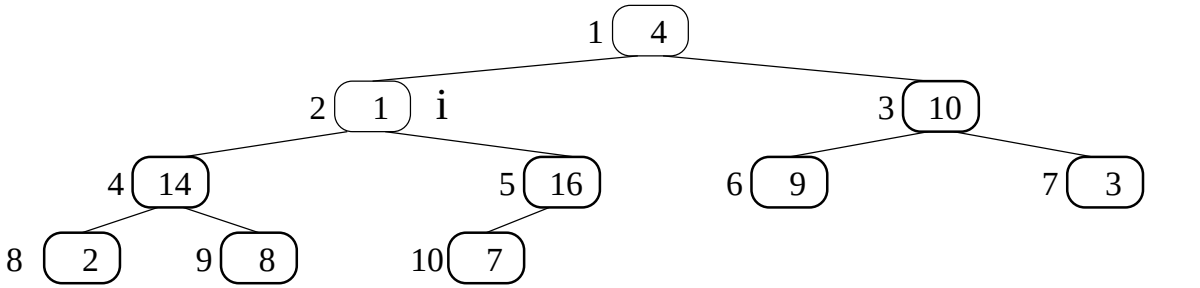
- Osaamme jo muutamia $\mathcal{O}(n^2)$ ajassa toimivia menetelmiä jotka järjestävät n lukua suuruusjärjestykseen
  - insertion sort eli lisäysjärjestäminen (luku 1)
  - bubble sort eli kuplajärjestäminen (luku 1)
  - selection sort eli valintajärjestäminen (ohjelmoinnin perusteet)
- tarkastellaan tässä luvussa muutamaa menetelmää joilla järjestämisessä päästään aikaan $\mathcal{O}(n \log n)$
  - kekojärjestäminen
  - lomitusjärjestäminen
  - pikajärjestäminen
- kurssilla johdatus diskreettiin matematiikaan osoitettiin, että jos järjestämisalgoritmin toiminta perustuu alkioiden vertailuun, ei algoritmi voi toimia nopeammin kuin $\mathcal{O}(n \log n)$
- järjestämisessä voidaan päästä lineaariseen aikaan tietyissä erikoistapauksissa, algoritmi ei silloin voi vertailla alkioita vaan toiminta perustuu alkioiden määrän laskemiseen: laskemisjärjestäminen

Kekojärjestäminen

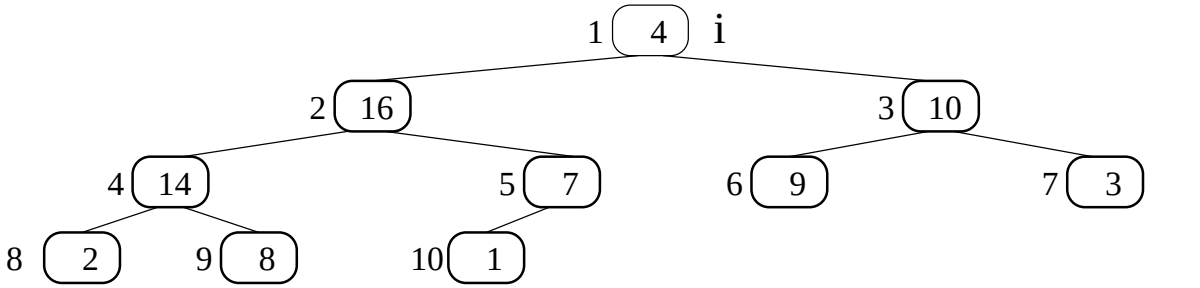
- Kalvolla 308 hahmottelimme jo idean, miten kekoa voidaan käyttää aikaansaamaan ajassa $\mathcal{O}(n \log n)$ toimiva järjestämisalgoritmi, esitetään nyt käytännön tasolla hieman tehokkaammin toimiva versio kekojärjestämisestä
- kalvolla 313 esittelimme operaation heapify(A,i) joka toimii seuraavasti
  - oletuksena on että left(i) ja right(i) viittaavat jo kekoja olevien alipuiden A[left(i)] A[right(i)] juuriin
  - operaation suorituksen jälkeen alipuu A[i] on keko
- operaation avulla on helppo rakentaa mistä tahansa taulukosta A keko:
  - lehdet, eli paikossa A[$\lfloor n/2 \rfloor + 1$], ..., 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

```
build-heap(A)
1   A.heap-size = A.length
2   for i = ⌊A.length/2⌋ downto 1
3       heapify(A,i)
```

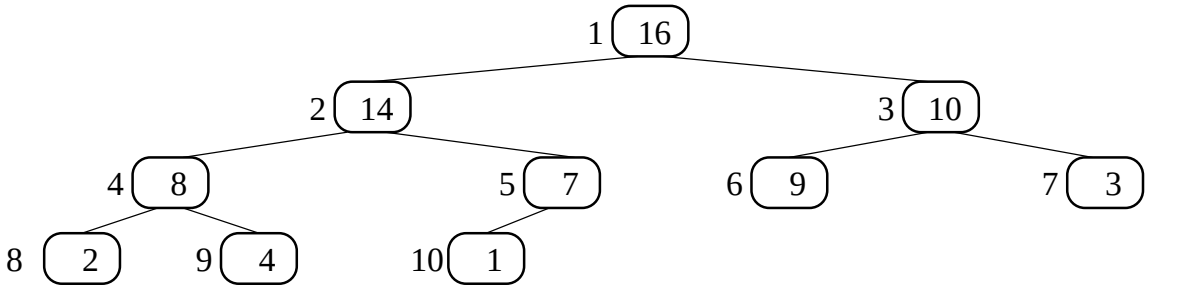

1 2 3 4 5 6 7 8 9 10
alussa
4 1 3 2 16 9 10 14 8 7
1 4
2 1
3 3
4 2
5 16
i
6 9
7 10
8 14
9 8
10 7
heapify(5)
1 4
2 1
3 3
4 2
i
5 16
6 9
7 10
8 14
9 8
10 7
heapify(4)
1 4
2 1
3 3
i
4 14
5 16
6 9
7 10
8 2
9 8
10 7
heapify(3)



heapify(2); heapify(5)



heapify(1); heapify(2); heapify(4)



tuloksena

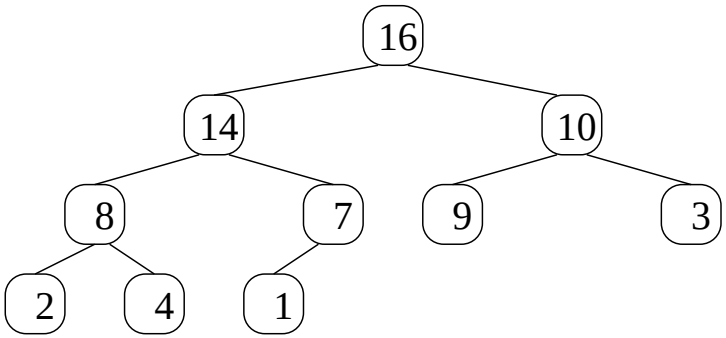
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

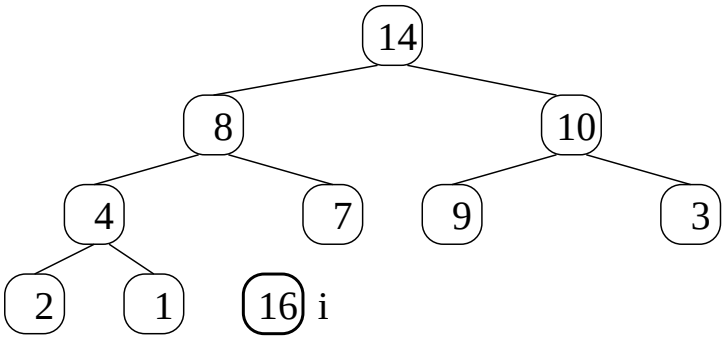
- kekojärjestäminen tapahtuu seuraavasti

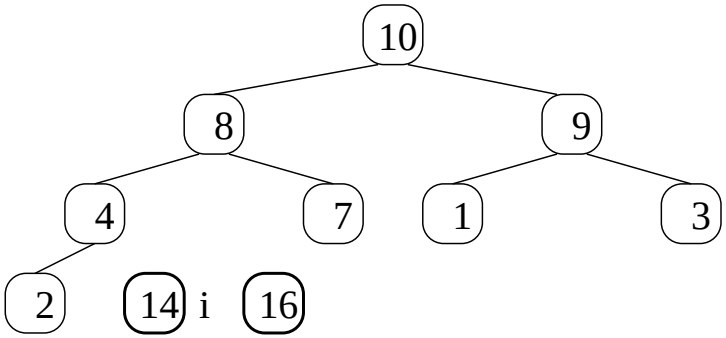
```
heap-sort(A)
1   build-heap(A)
2   for i = A.length downto 1
3       vaihda A[1] ja A[i]
4       A.heap-size = A.heap-size-1
5       heapify(A,1)
```

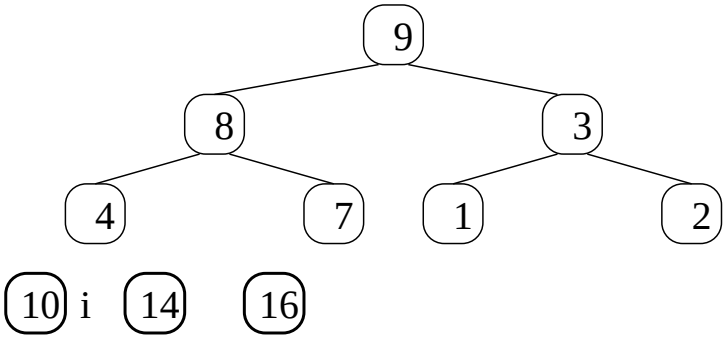
- toimintaidea
  - aineistosta tehdään ensin maksimikeko, näin suurin alkio on kohdassa A[1]
  - vaihdetaan keskenään keon ensimäinen ja viimeinen alkio
  - näin saamme yhden alkion vietyä taulukon loppuun "oikealle" paikalleen
  - pienentämällä keon kokoa yhdellä huolehditaan vielä että viimeinen alkio 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

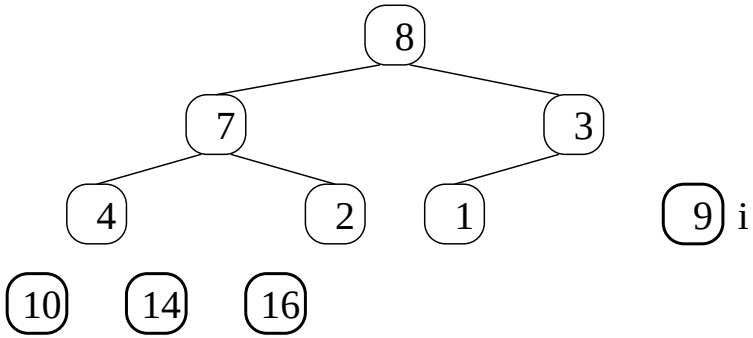
- kekojärjestämisen aikavaativuus
  - heapify:n aikavaatimus $\mathcal{O}(\log n)$ keolle jossa n alkiota
  - build-heap operaatiossa kutsutaan $n/2$ kertaa heapify keolle jossa *korkeintaan* n alkiota, siis build-heap käyttää aikaa *korkeintaan* $\mathcal{O}(n \log n)$
  - heap-sortissa kutsutaan vielä $n - 1$ kertaa heapify-operaatiota
  - kokonaisuudessaan kekojärjestämisen aikavaativuus on siis $\mathcal{O}(n \log n)$
- (hieman tarkempi analyysi (ks. Cormen luku 6.3) paljastaa että operaation build-heap aikavaativuus onkin vain $\mathcal{O}(n)$, tämä ei tosin vaikuta koko operaation aikavaativuuteen)
- tilavaativuus
  - heapify kutsuu rekursiivisesti itseään pahimmillaan keon korkeudellisen verran, operaatio on kuitenkin helppo toteuttaa myös ilman rekursiota jolloin sen tilantarve vakio
  - muutkaan kekojärjestämisen toimet eivät aputilaa tarvitse, siis tilavaativuus $\mathcal{O}(1)$

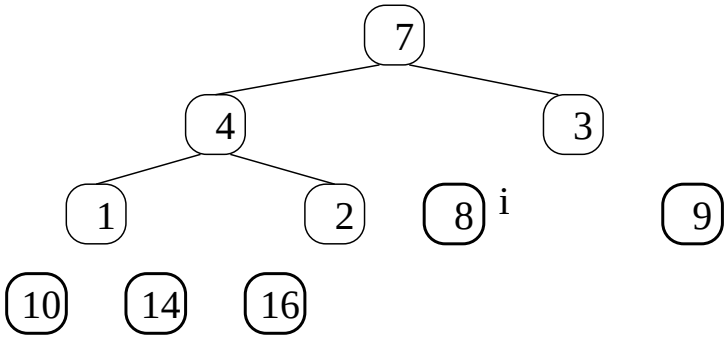

16
14
10
8
7
9
3
2
4
1

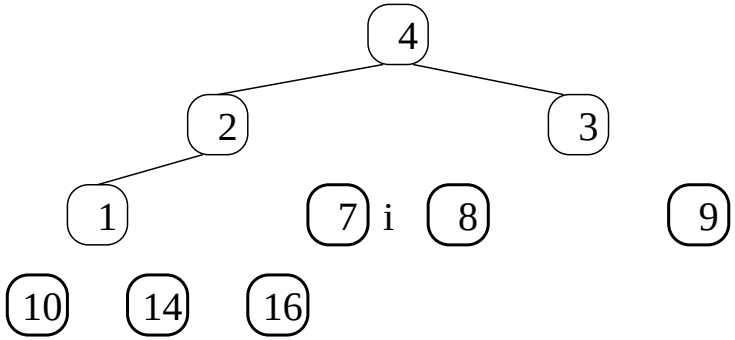

14
8
10
4
7
9
3
2
1
16
i

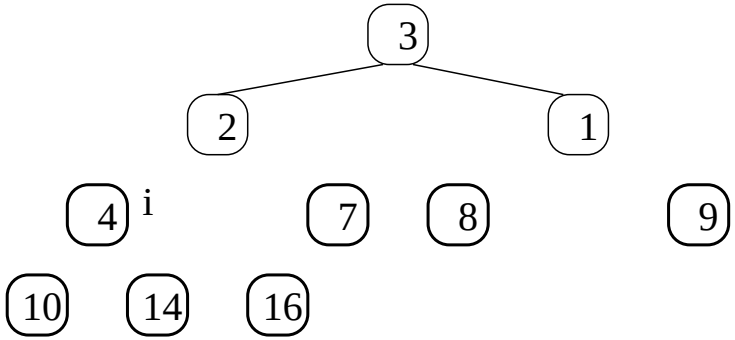

10
8
9
4
7
1
3
2
14
i
16

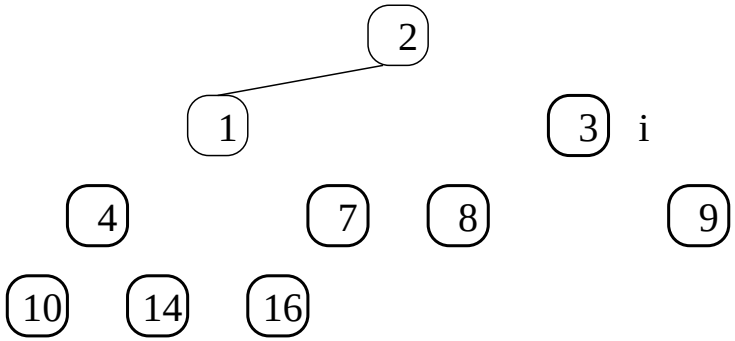

9
8
3
4
7
1
2
10
i
14
16

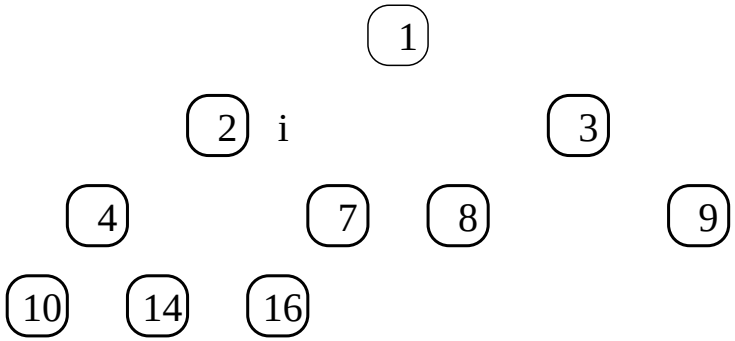

8
7
3
4
2
1
9
i
10
14
16


7
4
3
1
2
8
i
9
10
14
16









tuloksena

1	2	3	4	7	8	9	10	14	16

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

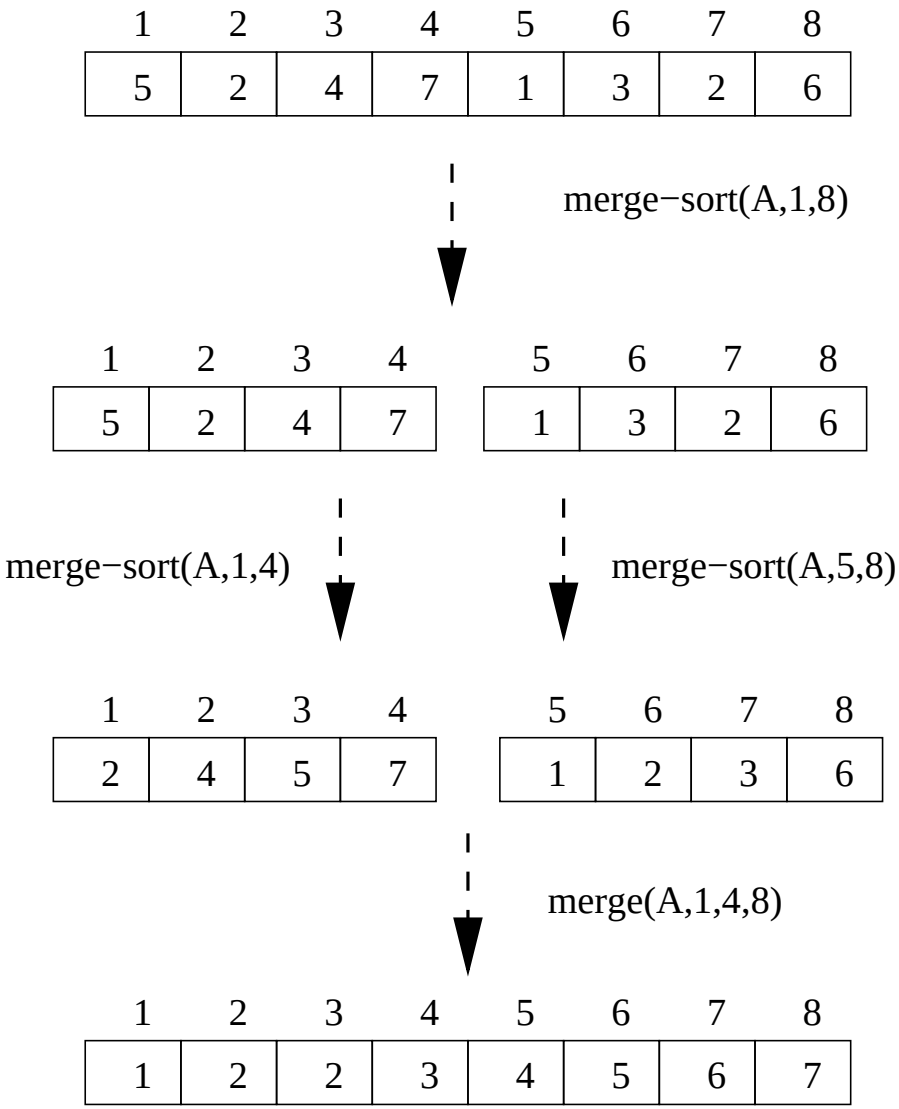
- Lomitusjärjestäminen perustuu *hajoita-ja-hallitse* (engl. divide-and-conquer) -tekniikkaan:
  - *hajoitetaan* ongelma pienempiin osaongelmiin
  - *hallitaan*, eli ratkaistaan osaongelmat rekursiivisesti
  - *yhdistetään* osaratkaisut siten että saadaan ratkaisu koko ongelmalle

- taulukko A[1,n] järjestetään kutsumalla merge-sort(A,1,n):

```
merge-sort(A,p,r)
1   if p<r
2       q = ⌊(p + r)/2⌋
3       merge-sort(A,p,q)
4       merge-sort(A,q+1,r)
5       merge(A,p,q,r)
```

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

- esimerkki:


1 2 3 4 5 6 7 8
5 2 4 7 1 3 2 6
merge−sort(A,1,8)
1 2 3 4 5 6 7 8
5 2 4 7 1 3 2 6
merge−sort(A,1,4)
merge−sort(A,5,8)
1 2 3 4 5 6 7 8
2 4 5 7 1 2 3 6
merge(A,1,4,8)
1 2 3 4 5 6 7 8
1 2 2 3 4 5 6 7

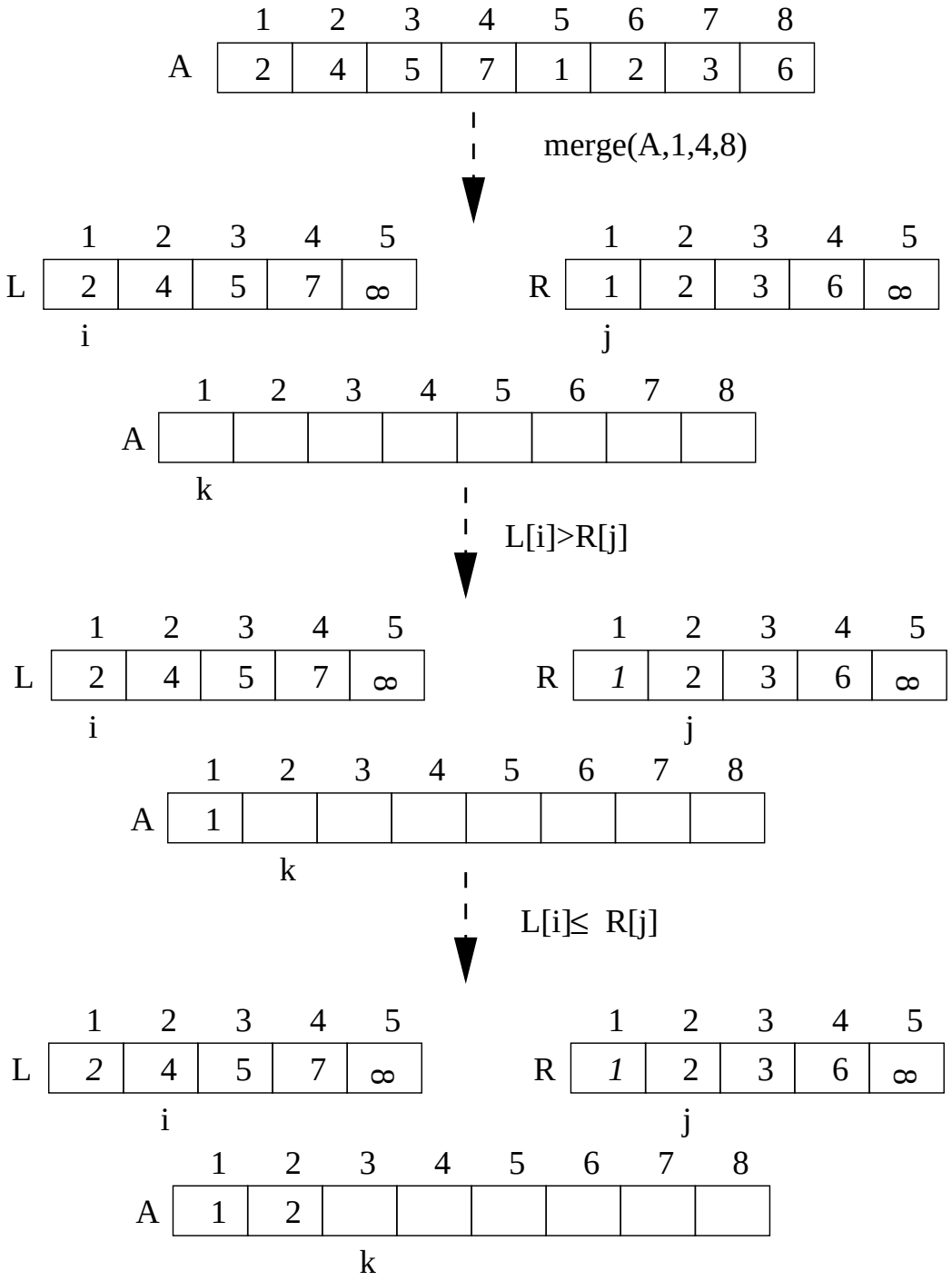
- järjestyksessä olevien taulukon osien A[p,q] ja A[q+1,r] lomittaminen järjestykseen on helppo tehdä:
  - kopioidaan A[p,q] aputaulukkoon L[1,n_1]
  - ja A[q+1,r] aputaulukkoon R[1,n_2]
  - laitetaan paikkaan A[p] pienin alkioista L[1] ja R[1],
  - jos L[1] laitettiin taulukkoon, niin paikkaan A[p+1] laitetaan pienin alkioista L[2] ja R[1]

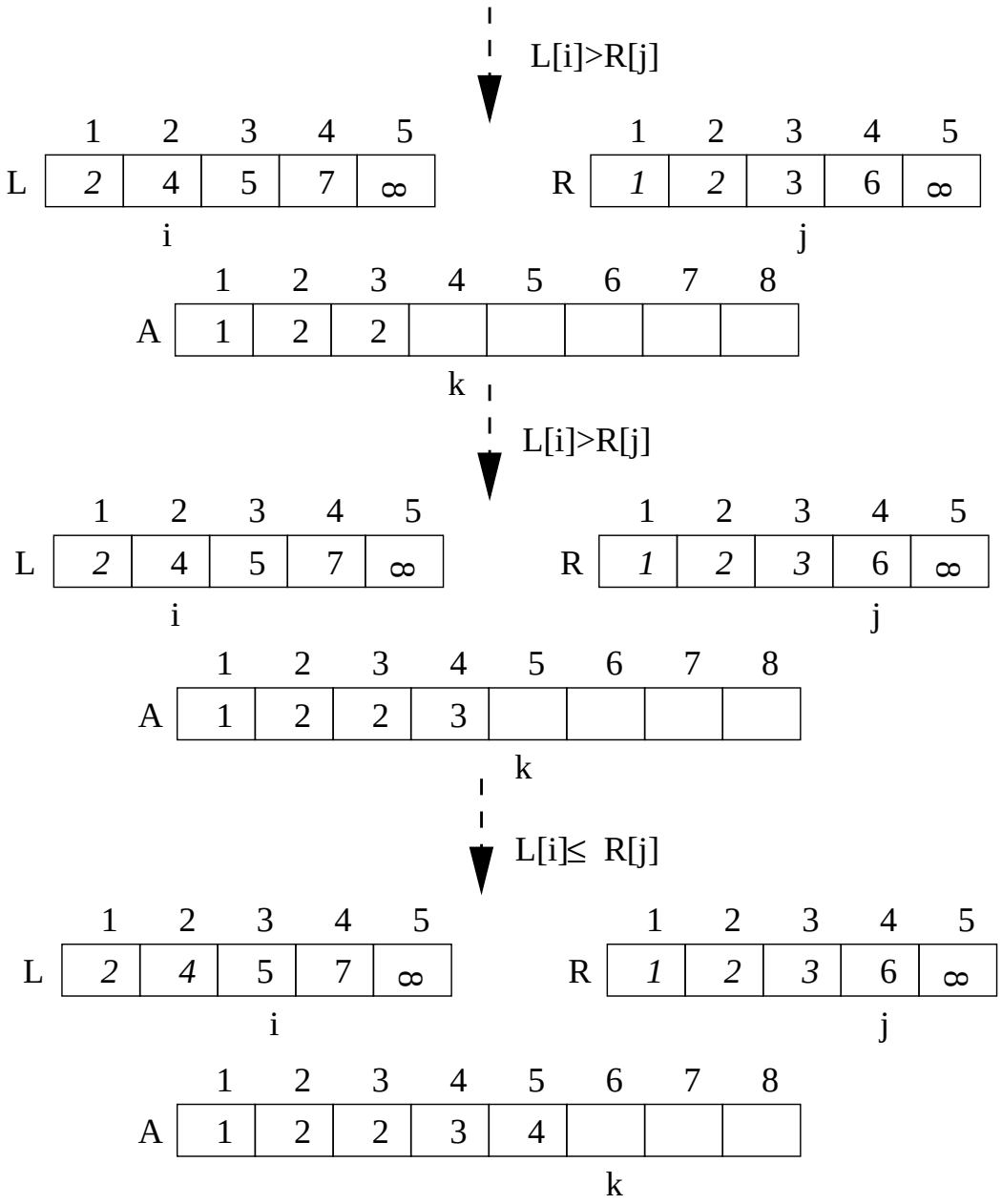
    jos taas R[1] laitettiin taulukkoon, niin paikkaan A[p+1] laitetaan pienin alkioista L[1] ja R[2]
  - näin jatketaan kunnes kaikki aputaulukoiden alkiot on siirretty taulukkoon A

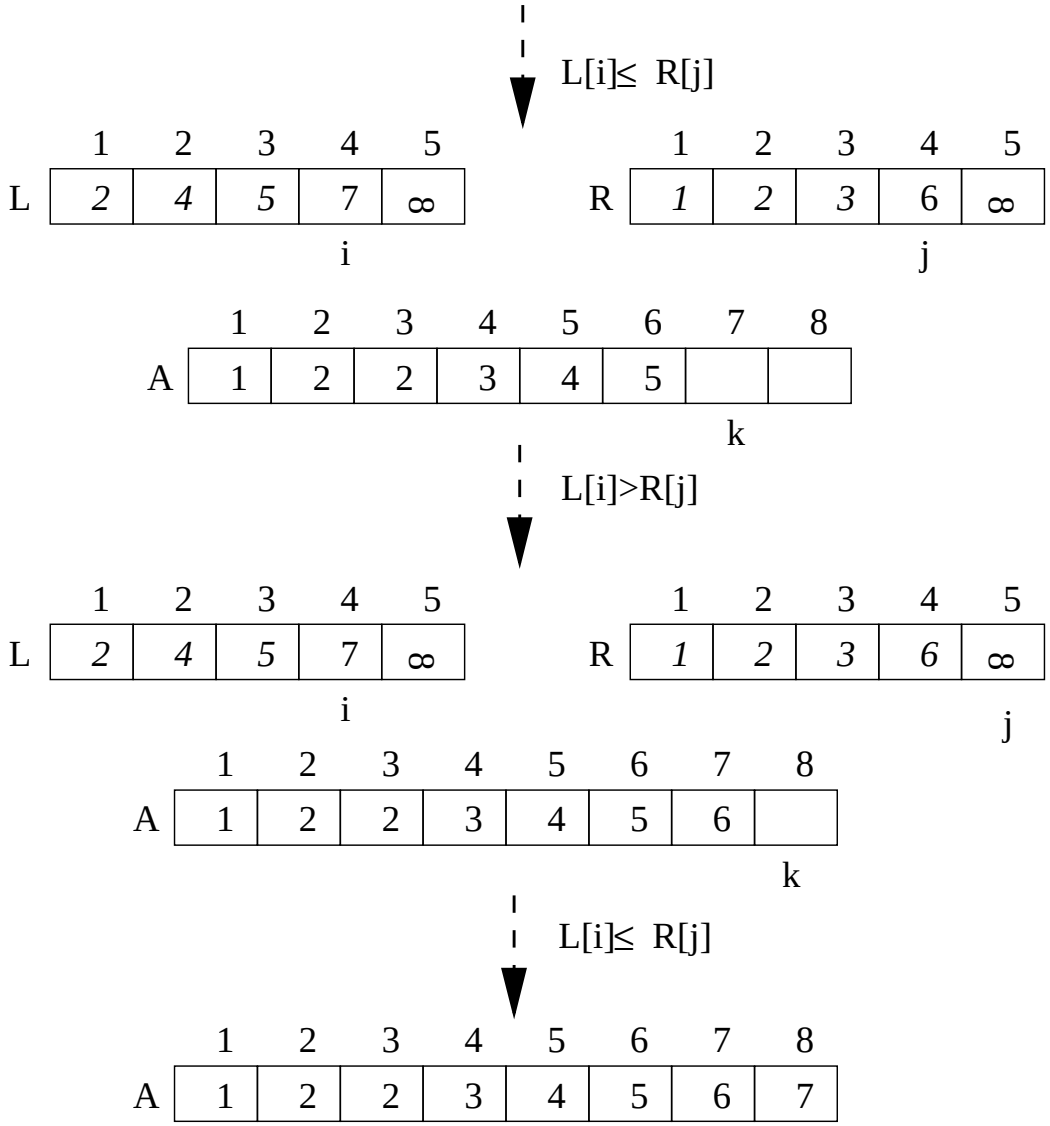
- algoritmina

```
merge(A,p,q,r)
1   n1 = q−p+1
2   n2 = r−q
3   // luodaan taulukot L[1,. . . , n1+1] ja R[1,. . . ,n2+1]
4   for i = 1 to n1
5       L[i] = A[p+i−1]
6   L[n1+1] = ∞          // lisätään loppuun alkio, joka on kaikkia muita suurempi
7   for j = 1 to n2
8       R[j] = A[q+j]
9   R[n2+1] = ∞          // lisätään loppuun alkio, joka on kaikkia muita suurempi
10  i = 1
11  j = 1
12  for k = p to r
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
```

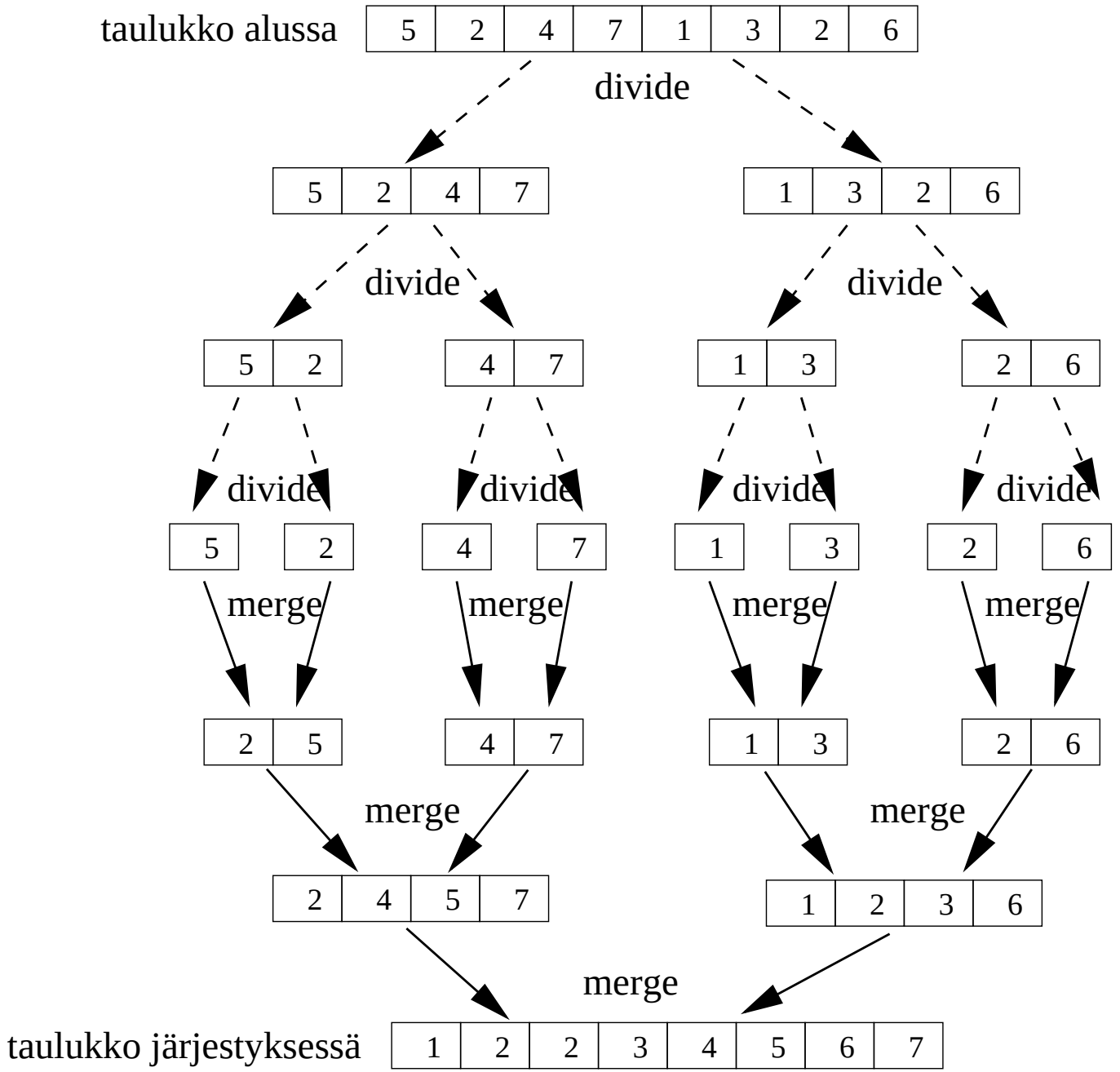
- esimerkki merge-operaation toiminnasta seuraavilla kalvoilla:


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 n lomitettavan taulukonosan pituus
  - operaatio luo kaksi aputaulukko L:n ja R:n kooltaan yhteensä $n_1 + 1 + n_2 + 1 = n + 2 = \mathcal{O}(n)$
  - rivien 4 ja 7 for-lauseiden vaativuus yhteensä $\mathcal{O}(n_1 + n_2) = \mathcal{O}(n)$
  - rivin 12 for toistetaan n kertaa tehden toisto-osassa vakiomäärä operaatioita, myöhempi for-lause siis myös $\mathcal{O}(n)$
- merge-operaation aikavaativuus sekä tilavaativuus siis $\mathcal{O}(n)$, missä n lomitettavan taulukonosan pituus $r - p + 1$
- 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

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

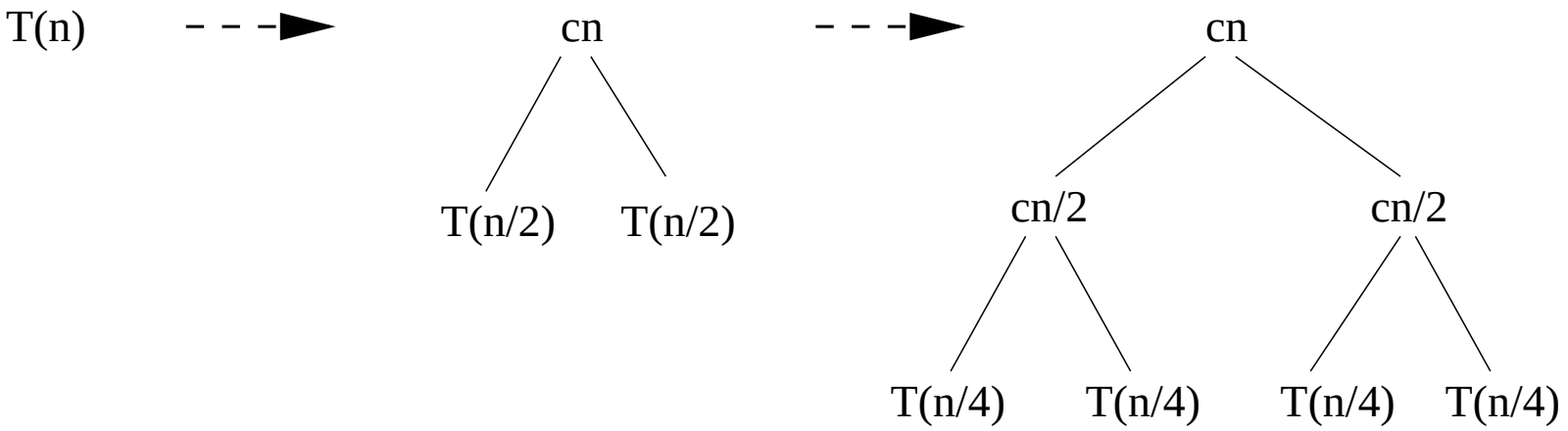
- missä c on sopivasti valittu vakio

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

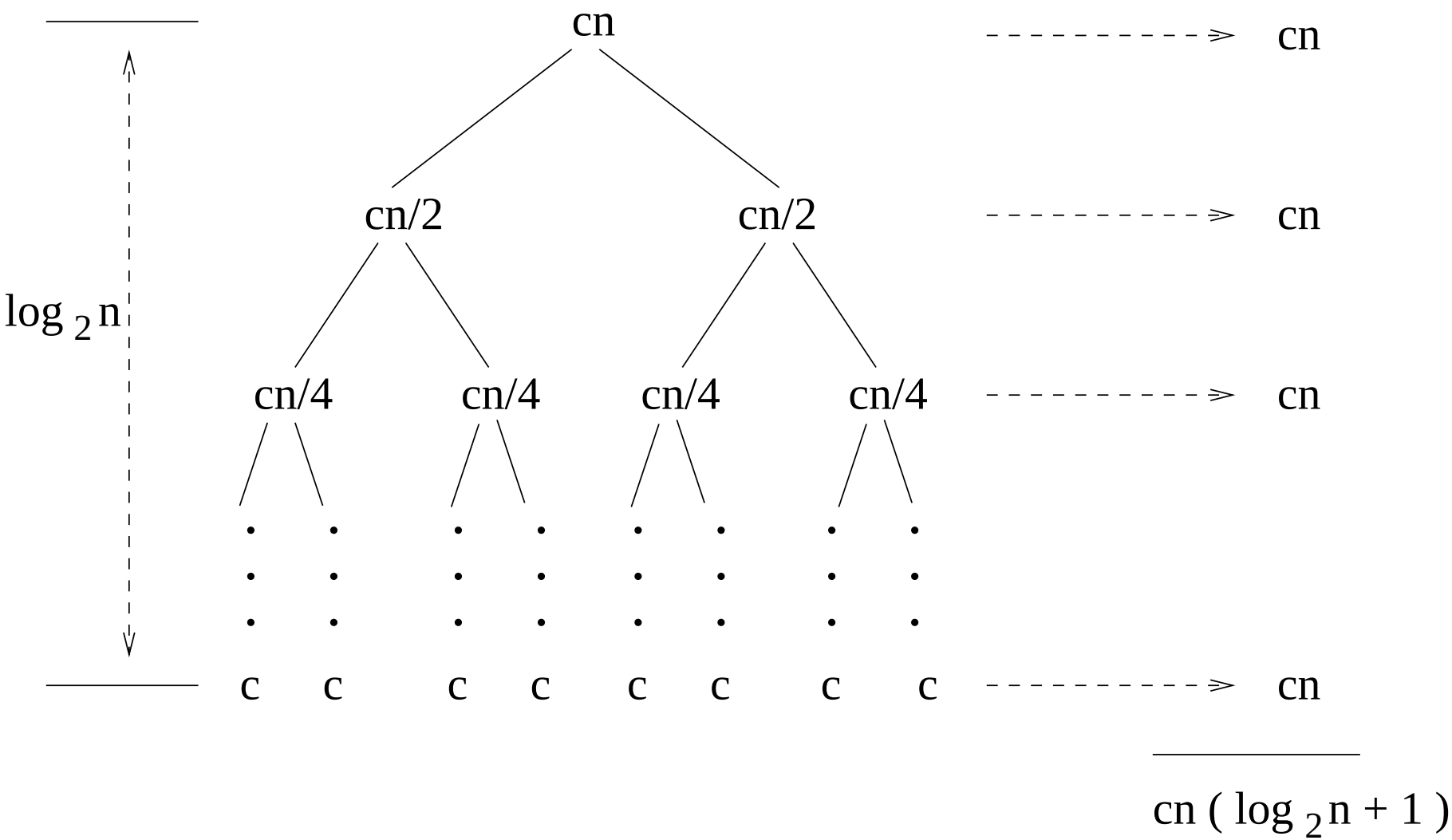
- Pitää selvittää mikä on n:n pituisen taulukon lomitusjärjestämisen aikavaativuus, eli on laskettava $T(n)$:n arvo
- aletaan laskemaan auki rekursioyhtälöä:

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

- 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 c c c c c c c
log 2 n
cn
cn
cn
cn
cn (log 2 n + 1)

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

- Rekursiopuu on täydellinen binääripuu jolla n lehteä, kalvolla 132 todistetusta lauseesta 1 seuraa nyt suoraan että rekursiopuun korkeus on $\log_2 n$ eli puulla on $\log_2 n + 1$ tasoa

- Toinen tapa päätellä rekursiotasojen määrä on seuraava:

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

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

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

- Algoritmin tilavaativuus on $\mathcal{O}(n)$ eli sama kuin ylimmän tason merge-operaatiolla, sillä rekursiosta huolimatta merge-operaatioita on suorituksessa kerrallaan vain yksi instanssi

- Järjestämisalgoritmia sanotaan vakaaksi (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 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 sukunimien 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

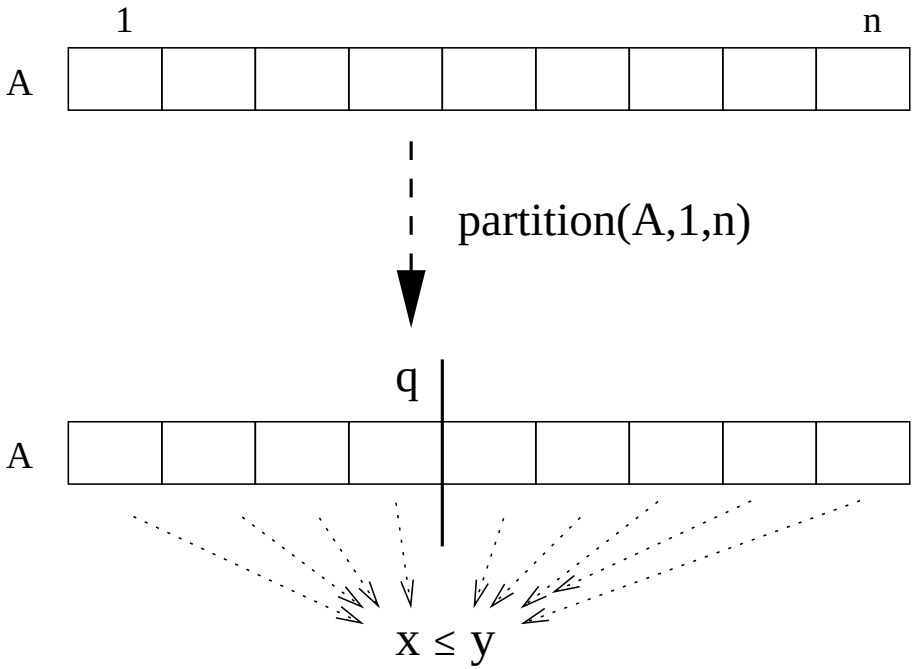
- Luokasta `Arrays` löytyy vastaava metodi normaalien taulukoiden järjestämiseen

Pikajärjestäminen

- sovelletaan edelleen hajoita-ja-hallitse-periaatetta
- taulukko A[1,n] järjestetään kutsumalla quick-sort(A,1,n):

```
quick-sort(A,p,r)
1   if p<r
2       q = partition(A,p,r)
3       quick-sort(A,p,q)
4       quick-sort(A,q+1,r)
```

- toimintaidea
  - operaation tehtävänä on järjestää taulukon A osa A[p,q]
  - jos järjestettävän osan pituus on korkeintaan 1 (eli $p \geq q$), 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:



- jaon jälkeen kaikki alkuosan A[p,q] alkiot ovat *korkeintaan yhtä suuria* kuin loppuosan A[q+1,r] alkiot
- huom: toisin kuin lomitusjärjestämisessä, taulukon osat A[p,q] ja A[q+1,r] eivät ole välttämättä saman kokoisia
- taulukon osat A[p,q] ja A[q+1,r] järjestetään kutsumalla niille rekursiivisesti quick-sort-operaatiota
- tulosten kokoamisvaihetta ei tarvita, sillä alkuosan ja loppuosan alkiot ovat jo partition-operaation jäljiltä keskenään oikeassa järjestyksessä

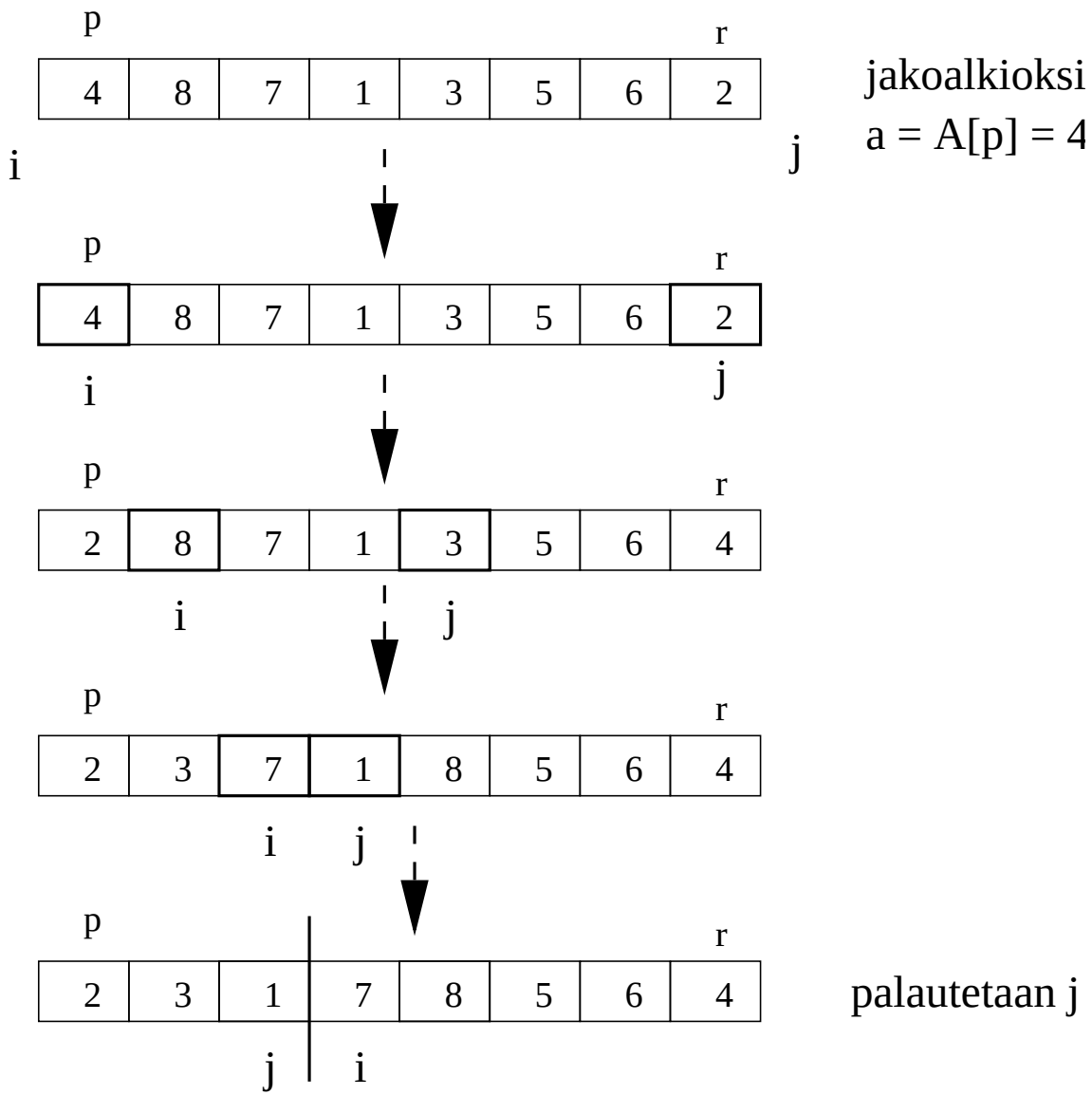
- pikajärjestämisen tehokkuuden kannalta on oleellista että partition-operaatio toimii nopeasti (lineaarisessa ajassa jaettavaan taulukonosan koon suhteen) ja tuottaa mahdollisimman tasaisia jakoja

- käytetään seuraavaa partition-operaatiota

```
partition(A,p,r)
1   a = A[p]
2   i = p-1
3   j = r+1
4   while true
5       repeat i = i+1 until A[i] ≥ a
6       repeat j = j−1 until A[j]≤ a
7       if i<j  vaihda A[i] ja A[j]
8       else return j
```

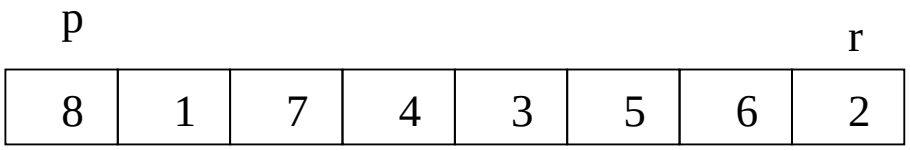
- toimintaidea
  - valitaan alussa *jakoalkioksi* vasemmanpuoleisimman alkion A[p] arvo a
  - ideana on nyt että jaon jälkeen kaikki vasemman puolen alkiot ovat suuruudeltaan korkeintaan a ja oikean puolen alkiot ovat suuruudeltaan vähintään a
  - lähdetään kulkemaan taulukkoa kummastakin päästä läpi, indeksi i alusta loppuun ja indeksi j lopusta alkuun
  - rivillä 5 viedään i osoittamaan alustapäin ensimmäistä alkiota jonka arvo on suurempi tai yhtä suuri kuin jakoalkio A[i]$\geq$ a
  - rivillä 6 viedään j osoittamaan lopustapäin ensimmäistä alkiota jonka arvo on korkeintaan jakoalkio A[j]$\leq$a
  - jos indeksi i on vielä indeksin j vasemmalla puolella, vaihdetaan alkioiden A[i] ja A[j] arvot
  - jos $i \geq j$, on tilanne se että kaikki j:n oikealla puolella olevat taulukon alkiot ovat vähintään a:n suuruisia ja j:n vasemmalla puolella olevat taas korkeintaan a:n suuruisia, eli voimme lopettaa ja palautetaan j, sillä se osoittaa oikeaa jakokohtaa

- esimerkki partition-operaation toiminnasta:

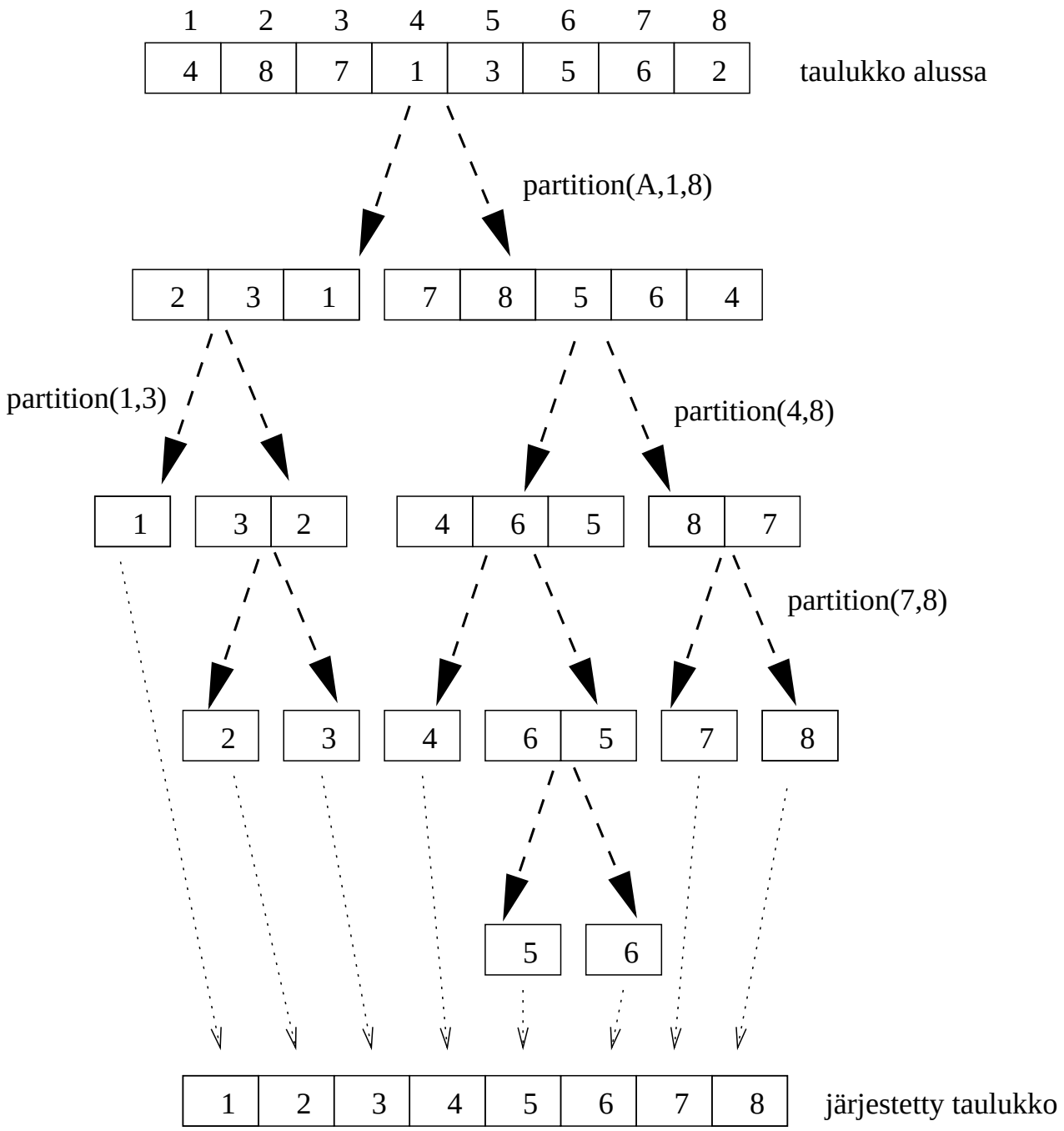


- selvästi partition-operaation aikavaativuus $\mathcal{O}(n)$ suhteessa käsiteltävän taulukon pituuteen n, aputilaa ei tarvita kuin parin muuttujan verran eli tilavaativuus $\mathcal{O}(1)$

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



- 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:

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

- pahimmassa tapauksessa pikajärjestäminen toimii neliöisesti, eli on selvästi huonompi kuin lomitus- ja kekojärjestäminen

- entä parhaassa tapauksessa, missä partition-operaatio sattuu aina jakamaan taulukon aina 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 $\mathcal{O}(n \log n)$

- pikajärjestämisen pahin tapaus on erittäin harvinainen, ja käytännössä pikajärjestys toimii yleensä paremmin kuin lomitus- tai kekojärjestäminen

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

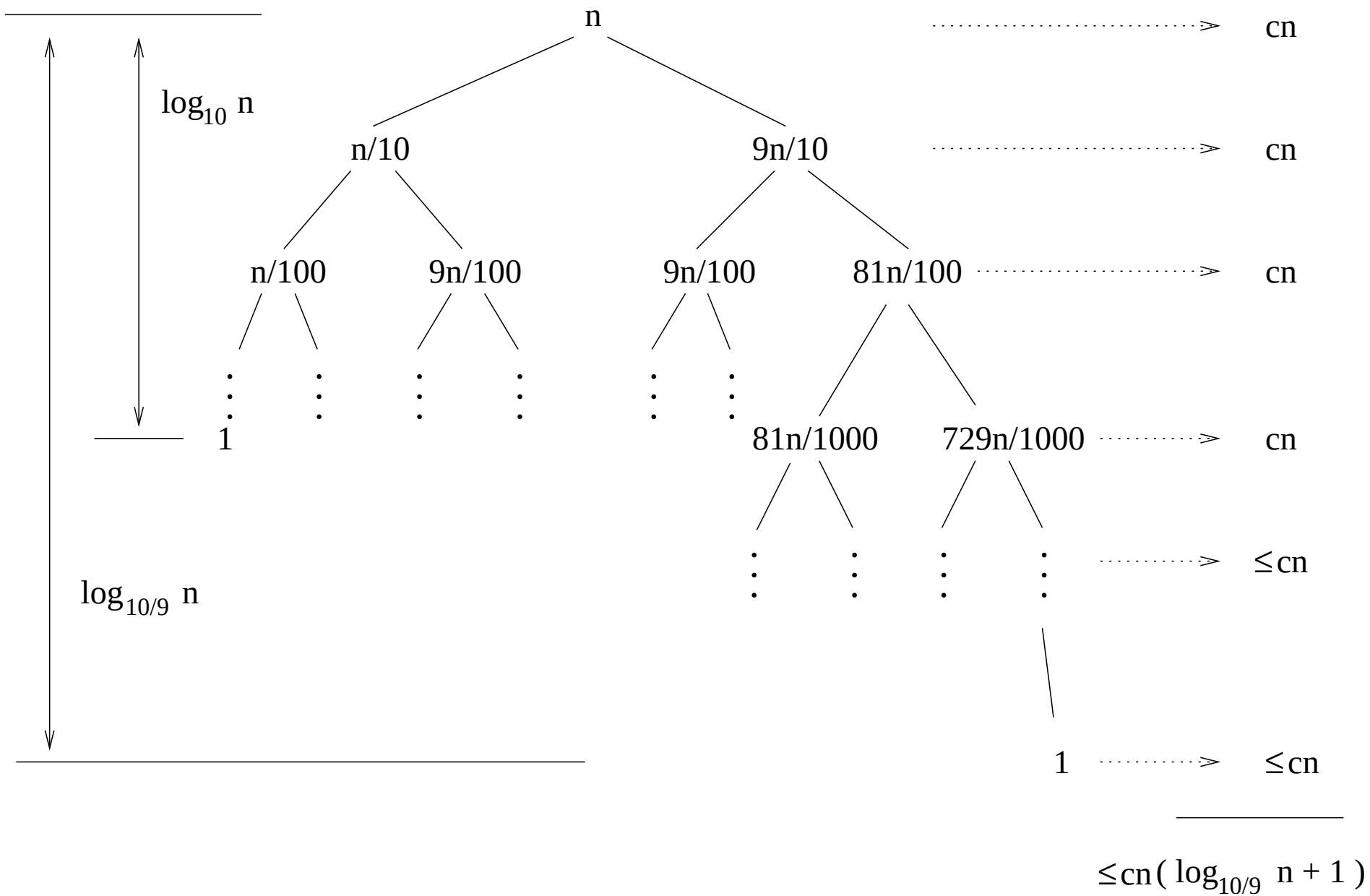
- analysoidaan vielä tilannetta missä partition jakaisi alkioita melko huonosti, eli oletetaan että k:n alkion taulukko jakautuisi pahimmillaan siten että pienemmässä osassa on vain $k/10$ alkiota ja suuremmassa $9k/10$

- 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
n/10
9n/10
n/100
9n/100
9n/100
81n/100
1
81n/1000
729n/1000
1
cn
cn
cn
cn
≤cn
≤cn
log10 n
log10/9 n
≤cn(log10/9 n + 1)

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

- näinkin epätasapainoinen partitiointi johtaa vielä $\mathcal{O}(n \ log \ n)$ aikavaativuuteen, tosin rekursiotasoja tulee noin 7 kertaa enemmän 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 quick-sort 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 quick-sort-kutsun päättävä rekursiivinen kutsu eli häntärekursio iteraatiolla:

```
quick-sort2(A,p,r)
1   while p<r
2       q = partition(A,p,r)
3       quick-sort2(A,p,q)
4       p = q+1                 // quick-sort(A,q+1,r) korvataan iteraatiolla
```

- Monet kääntäjät tekevät tällaisen optimoinnin automaattisesti

- Tämä ei vielä muuta pahimman tapauksen vaativuutta, koska pahimmassa tapauksessa rekursiivisesti käsiteltävä osataulukko $A[p\,..\,q]$ 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:

```
quick-sort3(A,p,r)
1   while p<r
2       q = partition(A,p,r)
3       if q-p < r-q                  // alkuosa pienempi
4           quick-sort3(A,p,q)
5           p = q+1                   // quick-sort(A,q+1,r) korvataan iteraatiolla
6       else                          // loppuosa pienempi
7           quick-sort3(A,q+1,r)
8           r = q                     // quick-sort(A,p,q) korvataan iteraatiolla
```

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

- nyt pahin tapaus onkin tasajako, ja tilavaativuus toteuttaa seuraavan rekursioyhtälön

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

- kalvolla 56 ratkaisimme saman rekursioyhtälön, ja päädyimme tulokseen $S(n) = \mathcal{O}(\log n)$, eli quick-sort3:n tilavaativuus pahimmassa tapauksessa ja siis tietysti myös keskimäärin $\mathcal{O}(\log n)$

- lisäysjärjestäminen on lyömätön pienillä aineistoilla
- pikajärjestämistä kannattaakin viritellä siten että jos järjesteltävän taulukonosan pituus on enää esim. alle 20, järjestetäänkin tämä osa käyttäen lisäysjärjestämistä, muuten toimitaan kuten pikajärjestämisessä normaalistikin

```
quick-sort4(A,p,r)
1   if r−p<20  insertion-sort(A,p,q)
2   else
3       q = partition(A,p,r)
4       quick-sort4(A,p,q)
5       quick-sort4(A,q+1,r)
```

- partition-operaatiossa paras valinta jakoalkioksi olisi jaettavan aineiston mediaani eli suuruudeltaan puolessa välissä oleva alkio, tällöin jako olisi mahdollisimman tasainen, mediaanin selvittäminen ei kuitenkaan onnistu tehokkaasti
- melko hyvä keino on valita jakoalkioksi mediaani arvoista A[p], A[$\lfloor (p+q)/2 \rfloor$] ja A[q]
- nyt esim. valmiina järjestyksessä tai käänteisessä järjestyksessä olevan aineiston pikajärjestäminen onnistuukin ajassa $\mathcal{O}(n \log n)$
- pahinta tapausta pikajärjestämisessä ei voida (järkevällä tavalla) välttää, mutta pahin tapaus voidaan tehdä niin harvinaiseksi ettei se juuri esiinny käytännössä

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

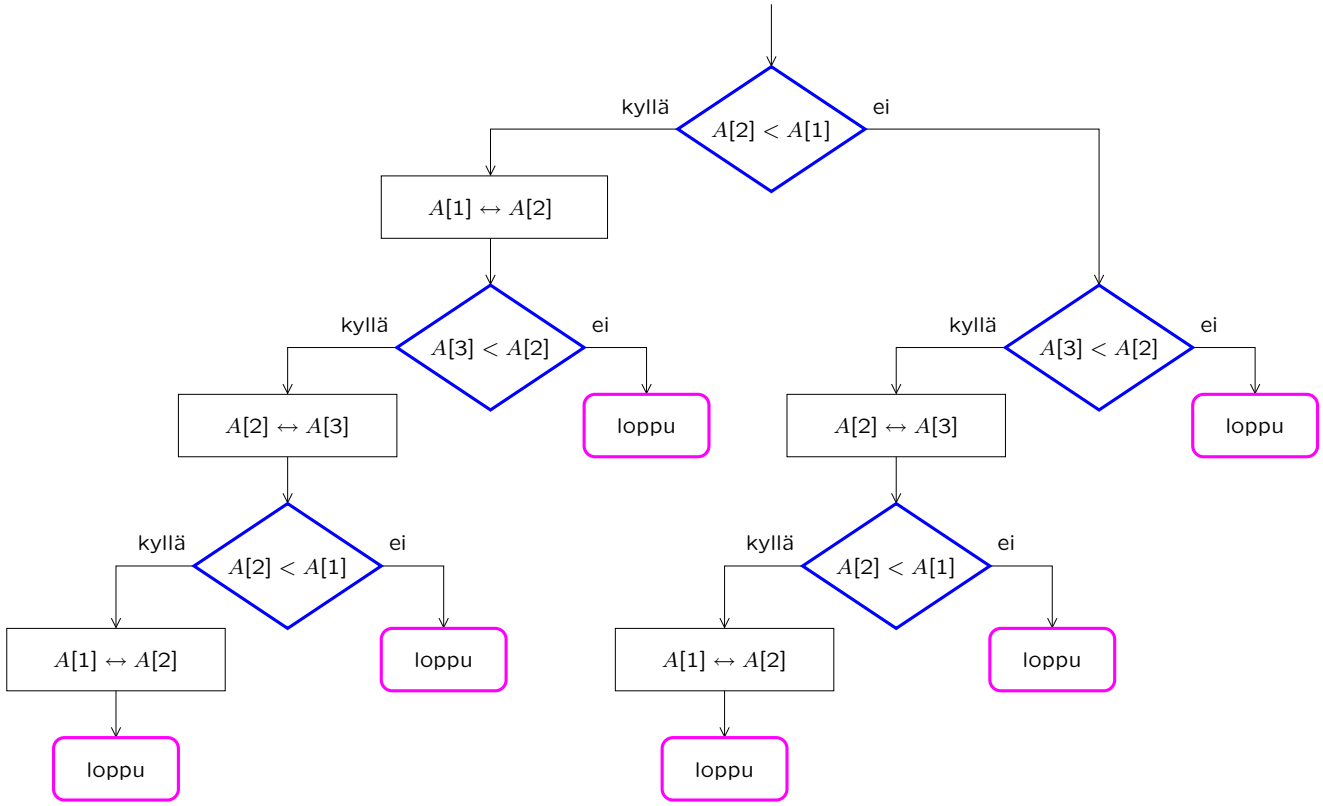
- Kurssilta *Johdatus diskreettiin matematiikkaan* tiedämme, että vertailuihin perustuva järjestämisalgoritmi suorittaa pahimmassa tapauksessa $\Omega(n \log n)$ vertailua

  Muistinvirkistys: merkinnällä Ω tarkoitetaan funktion kasvunopeuden alarajaa

- Kertaamme lyhyesti tämän väitteen takana olevat ajatukset

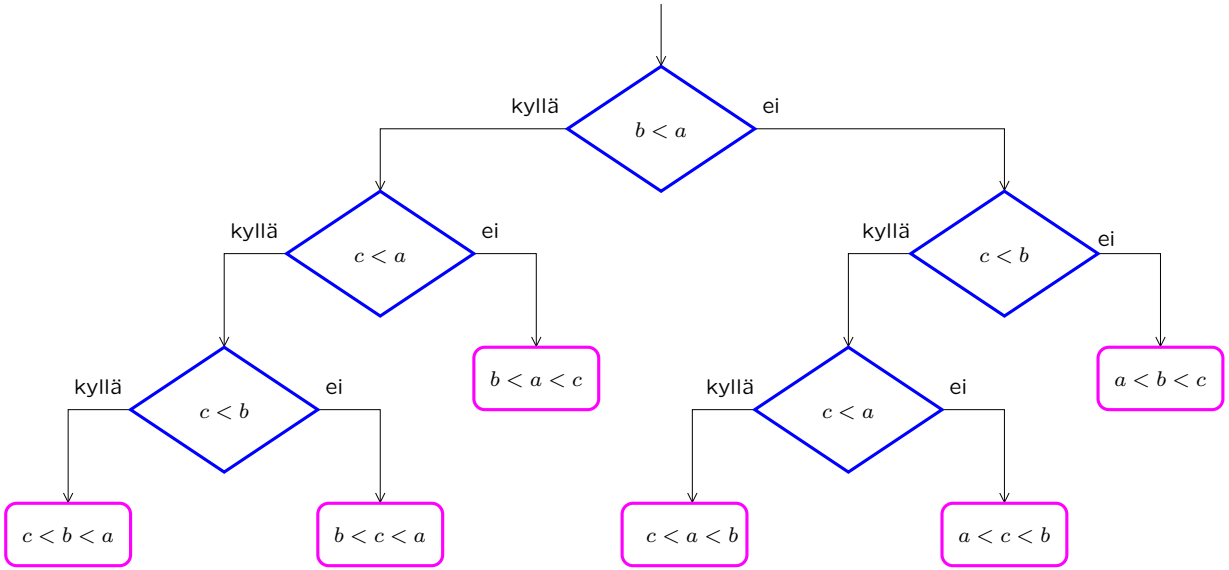
- Lähtökohtana on jokin vertailuihin perustuva järjestämisalgoritmi jollekin kiinteälle syötteen koolle n.

- 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

- Päätöspuun lehdestä nähdään, mikä on alkioiden a, b ja c järjestys



- Yksinkertaisuuden vuoksi oletetaan, että alkiot ovat erisuuria

- Yleisesti järjestämisalgoritmia vastaavassa päätöspuussa
  - sisäsolmuina on ehtotestejä
  - jokaisella sisäsolmulla on kaksi lasta
  - lehtinä 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 kyseessä on binääripuu, sen korkeus on ainakin $\log_2(n!)$, ks. kalvo 137
- Tämä on siis samalla alaraja algoritmin pahimman tapauksen aikavaativuudelle

- Teemme nyt hyvin karkean arvion

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

- 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

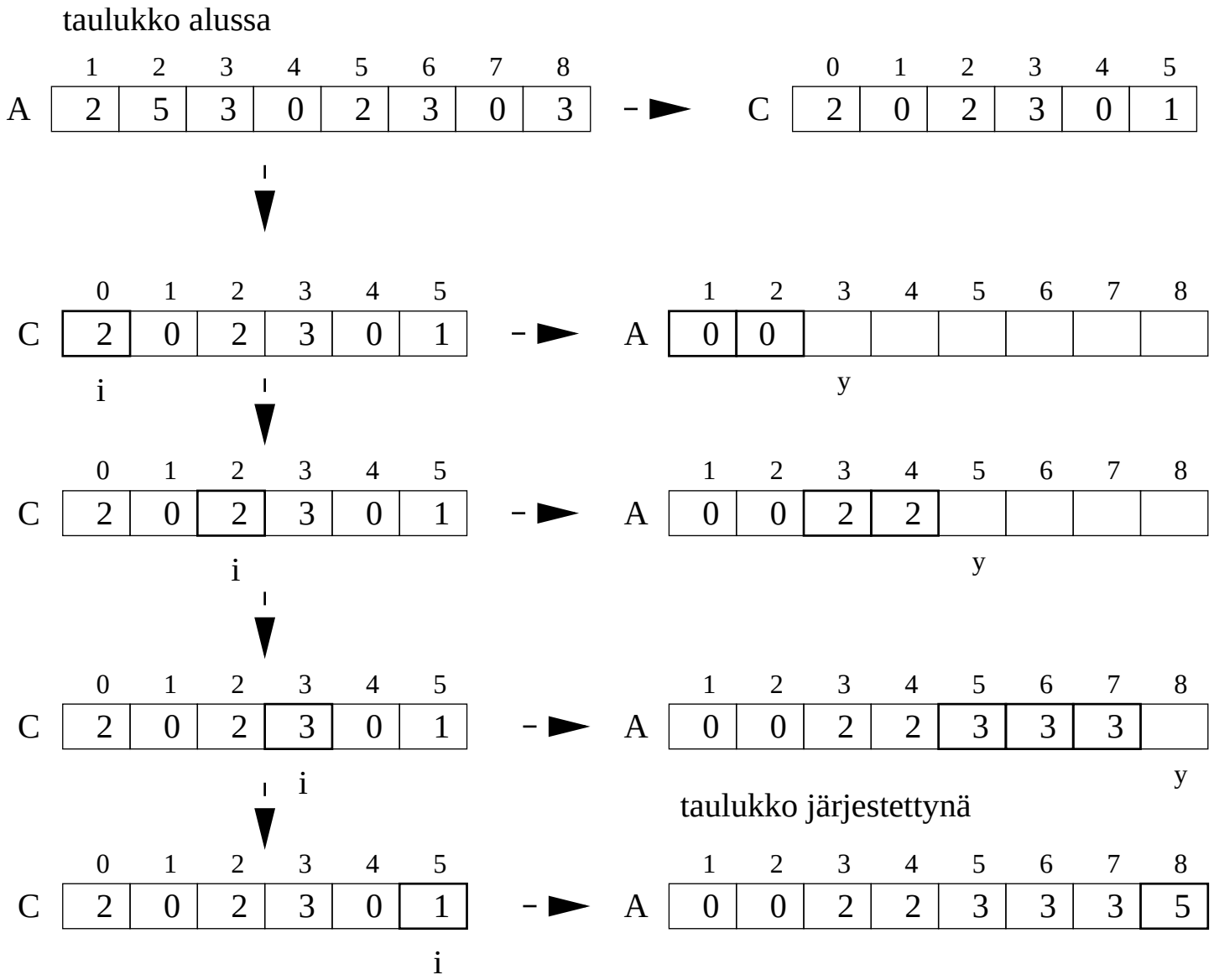
Järjestäminen lineaarisessa ajassa

- 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 i esiintymien määrä talletetaan paikkaan C[i]
  - sijoitetaan taulukkoon A ensin C[0] kertaa luku 0, C[1] kertaa luku 1 jne
  - näin taulukossa 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
```

- esimerkki:



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

- kutsutaan algoritmia *laskemisjärjestämiseksi* (counting sort), kyseessä ei kuitenkaan ole sama versio laskemisjärjestämisestä mikä löytyy Cormenista

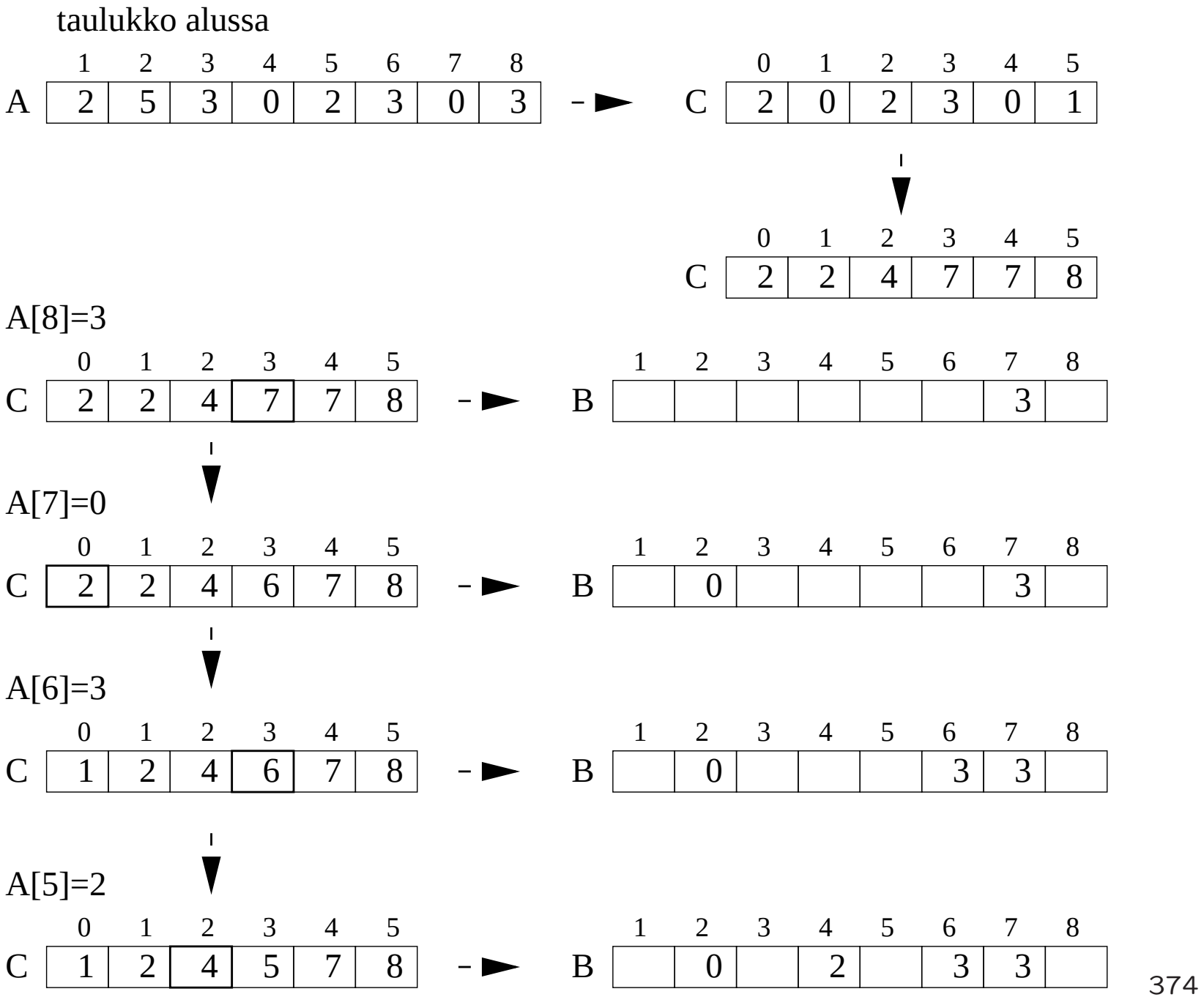
- jos järjestettäviin alkioihin liittyy muita datakenttiä ei juuri esitetty versio laskentajärjestämisestä toimi

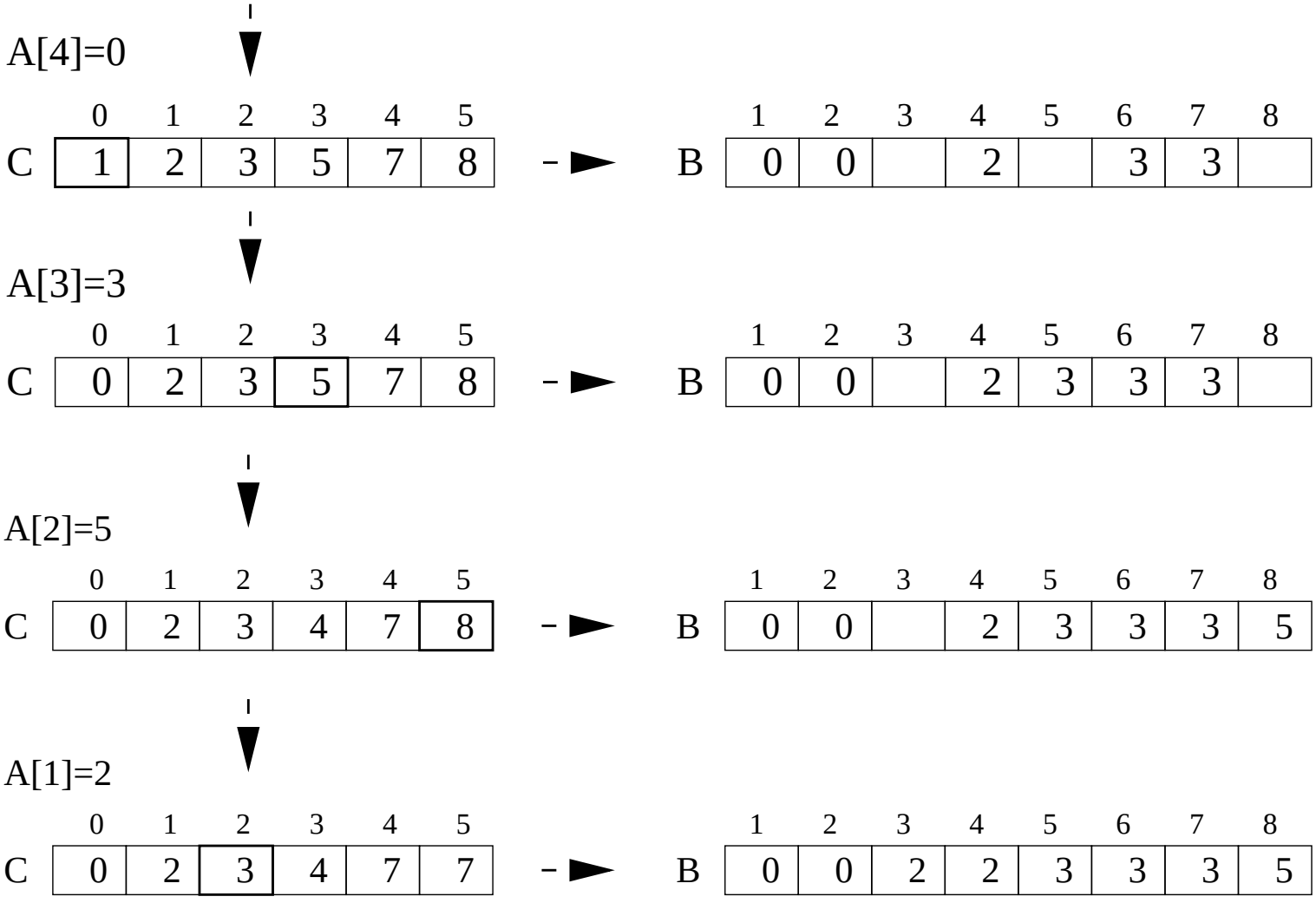
- esitetään vielä laskemisjärjestämisestä Cormenin versio, joka välttää yllä mainitun ongelman

```
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 laskemisjärjestyksen toiminnasta seuraavalla kalvolla:


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

Yhteenveto järjestämisalgoritmeista

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

- 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 kirjastotototeutukset perustuvat useimmiten 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

- Vaikka lomitusjärjestämisen yhdistämisvaihe merge ei ole vaativuudeltaan kuin $\mathcal{O}(n)$, käy se alkiot läpi kolmeen kertaan: ensin tapahtuu kopiointi aputaulukkoon, sitten aputaulukkojen läpikäynti, jonka yhteydessä alkiot kopioidaan takaisin alkuperäiseen taulukkoon

- Kasvanut muistivaativuus vaikuttaa myös muistiinviittausten viemään aikaan prosessorin välimuistin suuremman käyttöasteen takia, sillä todennäköisyys sille, että kaikki tarpeellinen data ei mahdu välimuistiin kasvaa

- 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 kekojärjestämistä, sillä se on pahimmassakin tapauksessa ajassa $\mathcal{O}(n \log n)$ toimivista algoritmeista käytännössä nopein

- 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 toimimmaan 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. Esim. kekojärjestämisessä syötteen muoto vaikuttaa jossain määrin `heapify`-operaation suoritusaikaan.

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

- 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 jos tieto jouduttaisiin hakemaan keskusmuistista

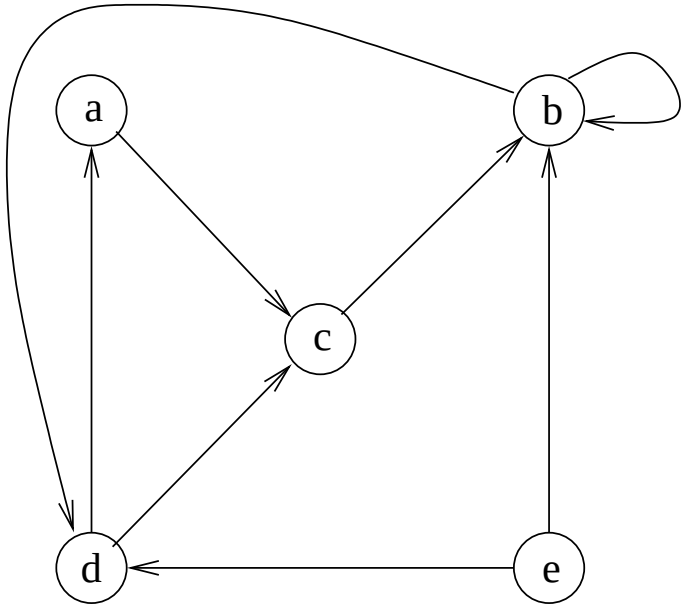
- Asian merkitys korostuu vielä enemmän jos koko järjestettävä taulukko ei mahdu kerralla keskusmuistiin vaan sijaitsee osittain kiintolevyn swap-osiossa

- Käytännössä on myös osoittautunut, että kekojärjestäminen suorittaa keskimäärin monta kertaa enenmmän vertailu- ja sijoitusoperaatioita kuin pikajärjestäminen

- Ei ole mitenkään ilmeistä mistä tämä seikka johtuu. Perusteluja asialle löytyy seuraavasta `http://users.aims.ac.za/~mackay/sorting/sorting.html`

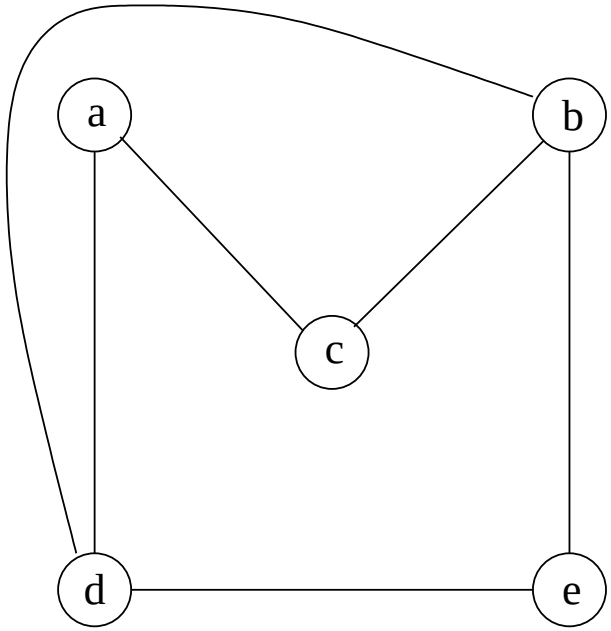
8. Verkot

- **Verkko** (engl. graph) koostuu **solmuista** (engl. node) ja niitä yhdistävistä **kaarista** (engl. edge)
- verkkoja on kahta päätyyppiä
- **suunnatuissa verkoissa** kaarilla on suunta
- esim:



- **suuntaamattomien verkkojen** kaarilla ei ole suuntaa

- esim:



- verkoilla on paljon sovelluksia tietojenkäsittelyssä

- tutustutaan ensin verkon käsitteistöön, sen jälkeen katsotaan muutamia verkkojen sovelluksia ja tutustutaan tyypillisimpiin verkkoalgoritmeihin

Käsitteistö

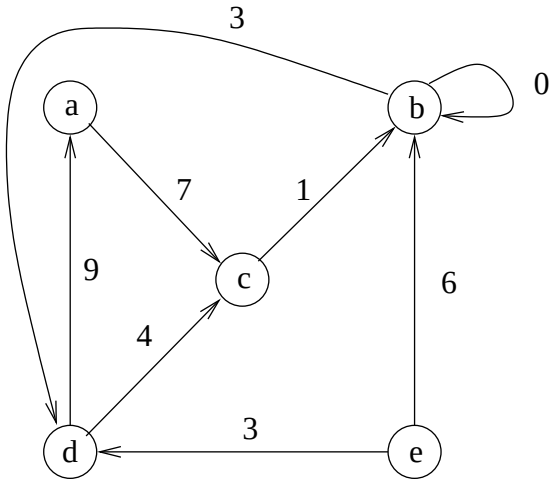
- formaalisti verkko G esitetään parina (V, E), missä
  - V on solmujen joukko
  - E on kaarien joukko
- kaaret ovat siis pareja (u, v) missä u ja v ovat solmuja
- suunnatussa verkossa $(u, v) \in E$ jos solmusta u on kaari solmuun v
  - tällöin u on kaaren *lähtösolmu* ja v kaaren *maalisolmu*
  - solmua v sanotaan solmun u *vierussolmuksi*
  - suunnatun verkon kaarista käytetään usein myös merkintää $u \rightarrow v$
- kalvon 381 suunnatussa verkossa siis esim. solmun e vierussolmuja ovat b ja d sillä $e \rightarrow b$ ja $e \rightarrow d$

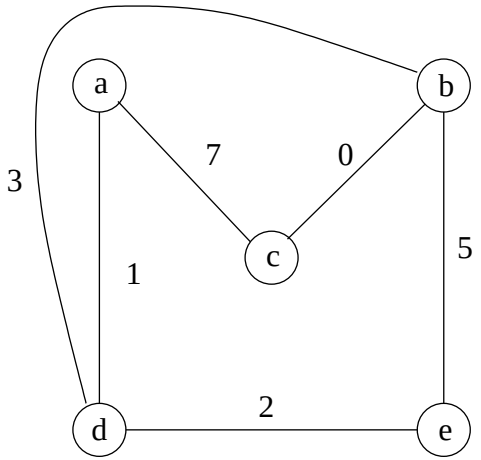
  solmun b vierussolmuja ovat d ja solmu itse, sillä $b \rightarrow d$ ja $b \rightarrow b$

- esim: kalvon 381 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
- suunnatun verkon solmujen u ja v välillä voi sensijaan olla kaaret molempiin suuntiin $u \rightarrow v$ ja $v \rightarrow u$
- suuntaamattomassa verkossa kaarten joukko E on *symmetrinen*, eli jos $(u, v) \in E$ niin myös $(v, u) \in E$
  - merkitsemme myös suuntaamattoman verkon kaaria joskus $u \rightarrow v$
  - 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: kalvon 382 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)





- oletetaan että kaaripainot ovat kokonaislukuja
- 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* solmusta v_1 solmuun v_n jos $v_1 \longrightarrow v_2, v_2 \longrightarrow v_3, \ldots, v_{n-1} \longrightarrow v_n$
- jos solmusta u on polku solmuun v käytetään merkintää $u \rightsquigarrow v$
- jos $u \rightsquigarrow v$ sanotaan että solmu v on *saavutettavissa* solmusta u
- *polun pituus* on polkuun liittyvien kaarien lukumäärä
- painotetussa verkossa *polun paino* on polun kaarien yhteenlaskettu paino
- polku on *yksinkertainen*, jos kukin solmu esiintyy polussa vain kerran, paitsi viimeinen ja ensimmäinen saavat olla sama solmu
- yksinkertainen polku on *sykli* jos viimeinen ja ensimmäinen solmu ovat samat

- kalvon 385 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* jos se ei sisällä yhtään sykliä
- huom: englannin kielessä *syklittömästä suunnatusta verkosta* käytetään joskus substantiivia *dag* (directed asyclic graph)
- huomionarvoista on, että verkon ei välttämättä tarvitse olla *yhtenäinen*, eli verkko voi koostua useista erillisistä osista

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

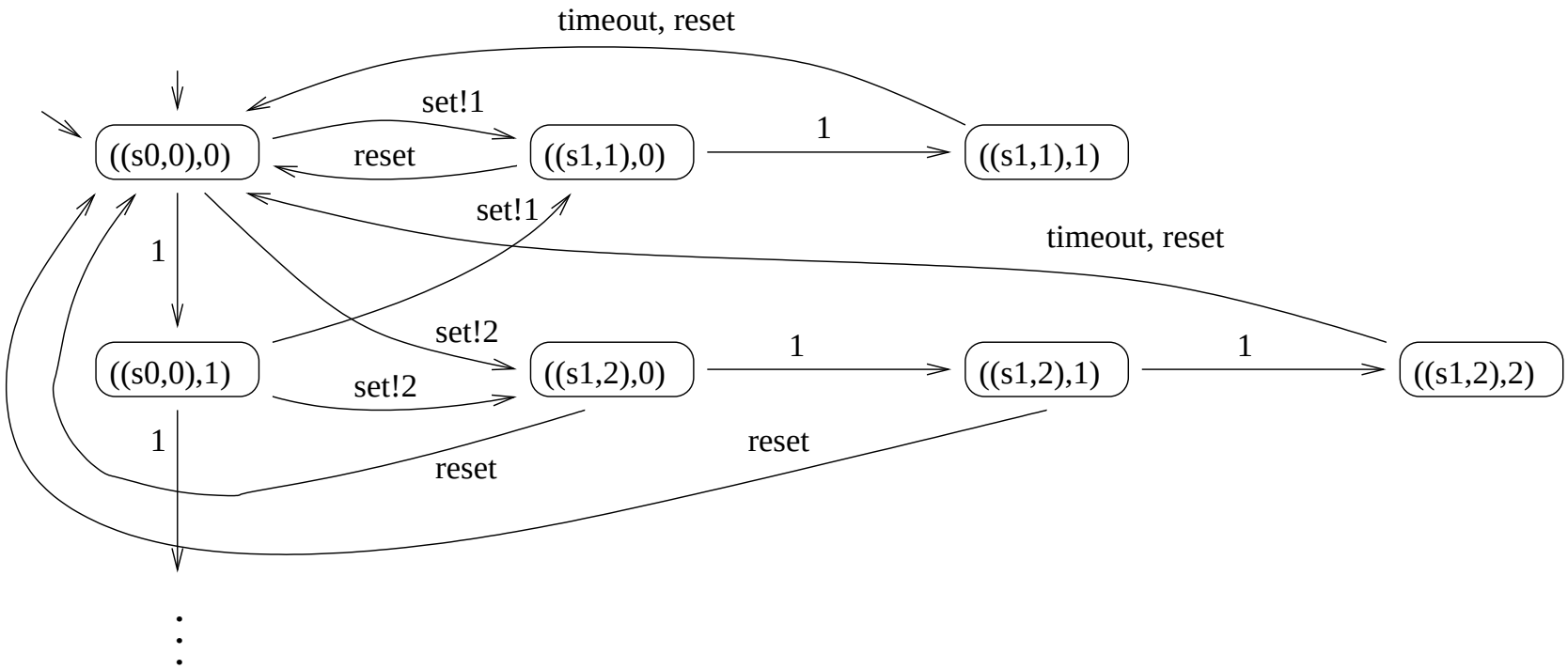
- **Robotin navigointi:**
  solmut: sopivalla tarkkuustasolla esitettyjä maantieteellisiä sijainteja
  kaaret: tunnettuja väyliä; ei suuntausta, ei painoja
  ongelma: ohjaa robotti paikasta A paikkaan B

- **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

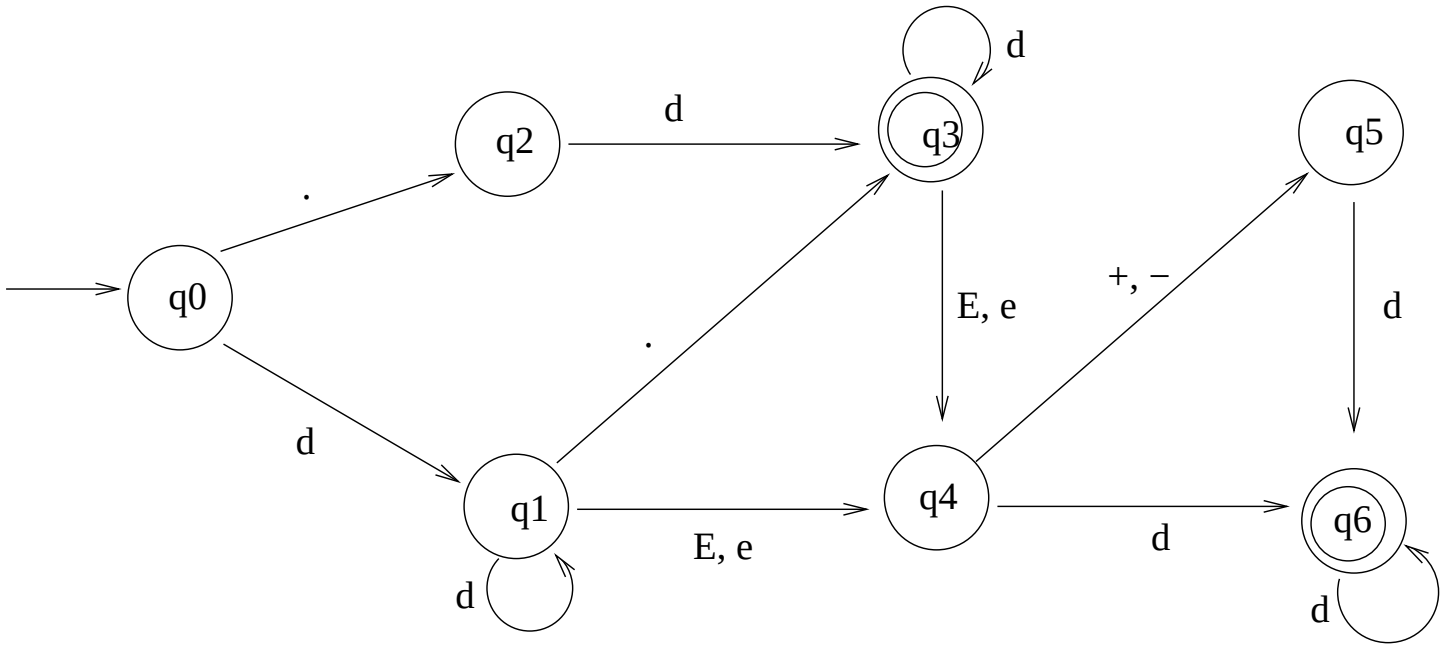
- **Tilasiirtymäverkko:**
  - solmut: reaaliaikaisen järjestelmän tiloja
  - kaaret: tilojen välisiä siirtymiä; suunnattu
  - painot: mikä ulkoinen tapahtuma aiheuttaa minkin tilasiirtymän
  - ongelma: voiko jokin tapahtumajono johtaa johonkin epätoivottuun tilaan tai tapahtumajonoon

- Esimerkki tilasiirtymäverkosta



- **Äärellinen automaatti** (kurssilla *Laskennan mallit*):
  - solmut: abstraktin automaatin laskennan tilanteita
  - kaaret: abstraktin automaatin laskenta-askelia
  - painot: kirjaimia
  - ongelma: Voiko tilasta A kulkea tilaan B siten, että kuljettujen kaarten painoista muodostuu haluttu merkkijono

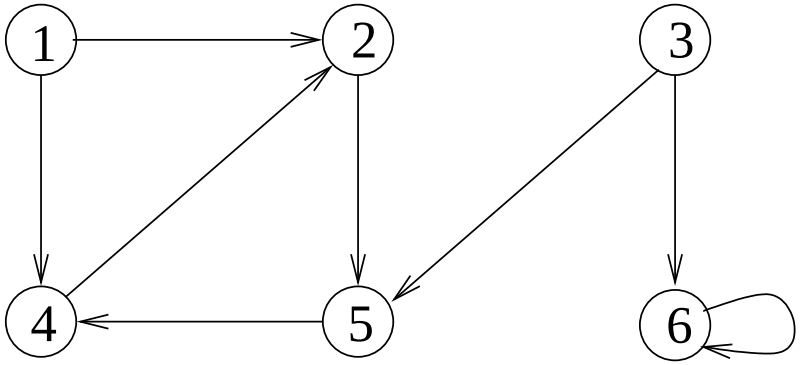
- Esimerkki äärellisestä automaatista, joka määrittelee Javan float -tyyppisen vakion syntaktisesti oikean muodon

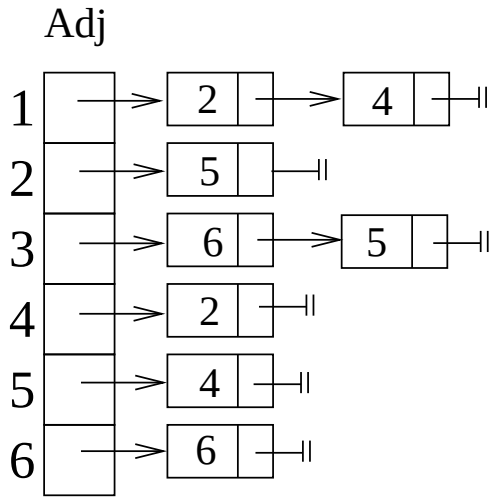


Verkkojen tallettaminen

- Tarkastellaan seuraavassa tapoja verkon $G = (V, E)$ esittämiselle tietokoneohjelmassa
- merkitään solmujen lukumäärää symbolilla $|V|$ ja kaarien lukumäärää symbolilla $|E|$
- vaihtoehtoisia talletustapoja on kaksi:
  - *vieruslistat* (engl. adjacency lists)
  - *vierusmatriisit* (engl. adjacency matrices)
  - on myös tilanteita, joissa verkko on mielekkäämpi tallettaa jossain muussa muodossa tai verkoa ei edes kannata tallentaa etukäteen
- **vieruslistaesityksessä** verkko $G = (V, E)$ esitetään taulukkona Adj joka sisältää $|V|$ kappaletta linkitettyjä listoja, yhden kullekin verkon solmulle
- jokaiselle solmulle $u \in V$ lista $Adj[u]$ sisältää kaikki ne solmut joihin u:sta on kaari

- esim: suunnattu verkko ja sen vieruslistaesitys





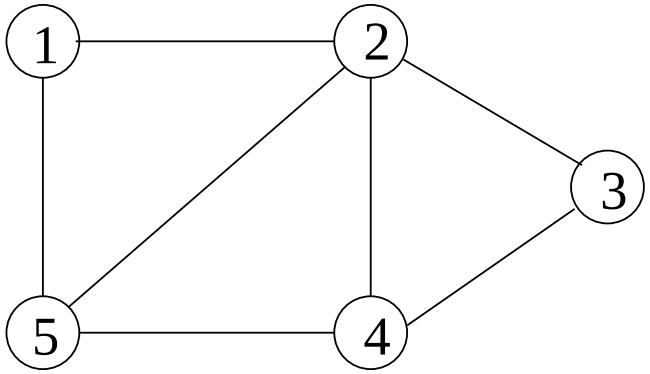
- suunnatun verkon vieruslistojen yhteenlaskettu pituus on $|E|$ sillä jokainen kaari on talletettu kertaalleen yhteen vieruslistoista

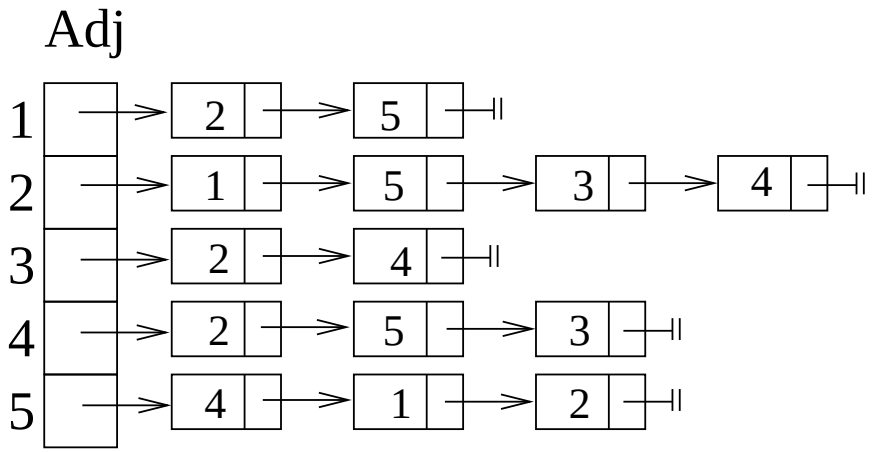
- koko vieruslistaesitys vie suunnattujen verkkojen tapauksessa tilaa $\mathcal{O}(|E| + |V|)$, sillä kaarien lisäksi varataan luonnollisesti tila taulukolle Adj

- Javassa vieruslistat voitaisiin esitää taulukollisena LinkedList-olioita

- jos solmuja ei esitetä kokonaislukuina, vieruslistat lienee helpointa toteuttaa tallettamalla LinkedList:eina esitetyt vieruslistat HashMap:in

- esim: suuntaamaton verkko ja sen vieruslistaesitys

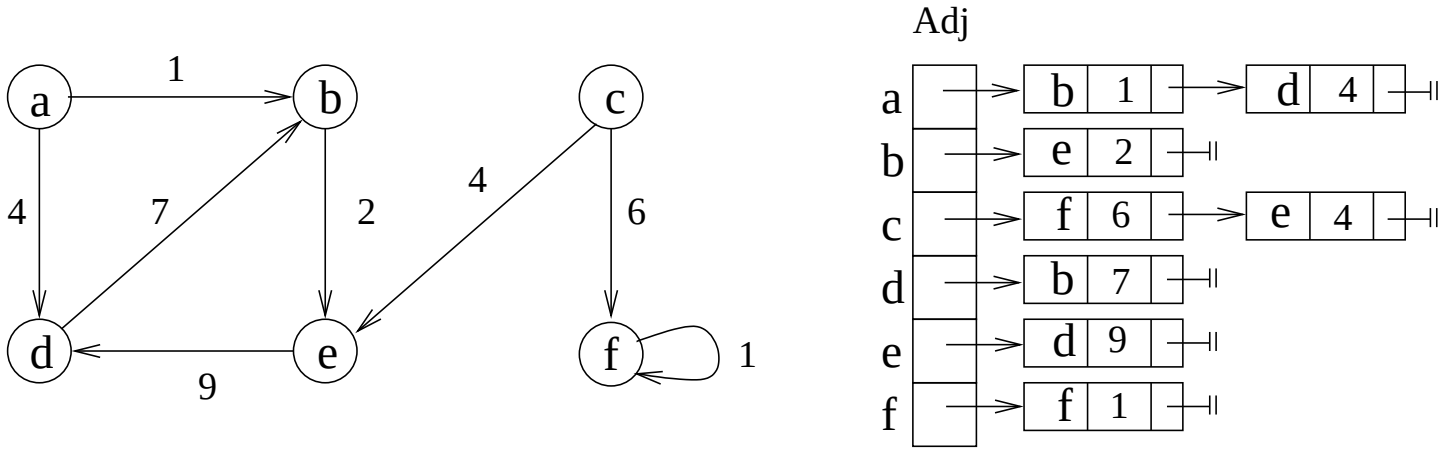




- suuntaamattoman verkon vieruslistojen yhteenlaskettu pituus $|E|$ on kaksi kertaa kaarien lukumäärä, sillä jokainen kaari on talletettu kahteen vieruslistaan

- koko vieruslistaesitys vie suuntaamattomien verkkojen tapauksessa tilaa $\mathcal{O}(|V| + 2 * kaarilkm) = \mathcal{O}(|V| + |E|)$

- myös kaarien painot voidaan tallentaa vieruslistarakenteeseen:



- vieruslistaesityksen hyvä puoli on siis kohtuullinen tilavaativuus joka on $\mathcal{O}(|E| + |V|)$, eli *lineaarinen* suhteessa solmujen ja kaarten määrään

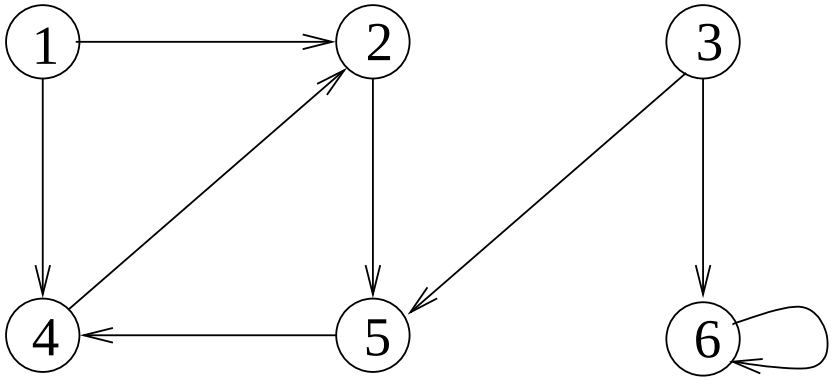
- huonona puolena taas se että tieto onko verkossa kaarta $u \rightarrow v$ ei ole suoraan saatavilla, vaan vaatii vieruslistan $Adj[u]$ läpikäynnin

- pahimmillaan tämä operaatio vie aikaa $\mathcal{O}(|V|)$ sillä solmusta u voi olla pahimmassa tapauksessa kaari kaikkiin verkon solmuihin

- Verkon $G = (V, E)$ **vierusmatriisiesityksessä** oletetaan että solmut on numeroitu, eli esim: $V = \{1, 2, \ldots, n\}$

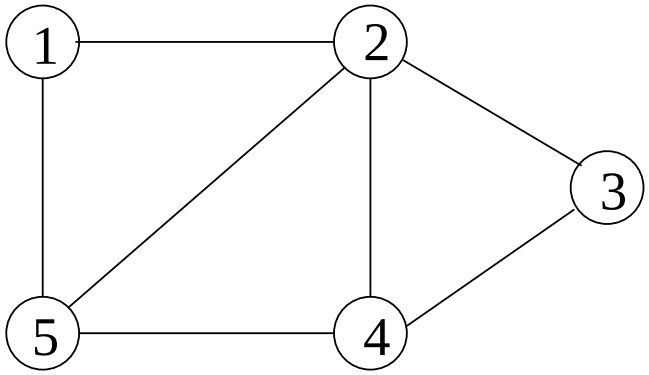
- vierusmatriisi on $n \times n$-matriisi A, missä $A[i, j] = 1$ jos $(i, j) \in E$ ja muuten $A[i, j] = 0$

- 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**

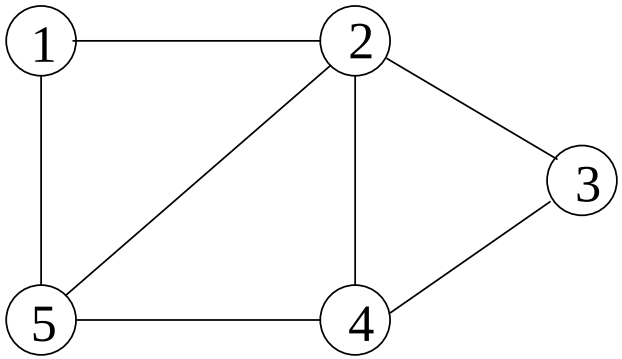
- ja suuntaamattomasta verkosta:



	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 $A[5,2] = 1$

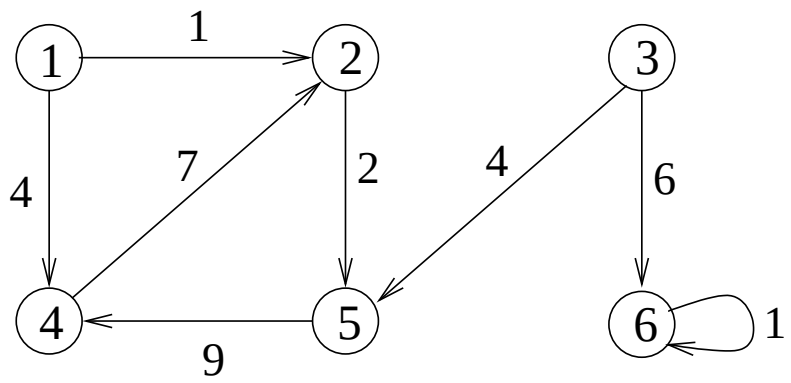
- suuntaamattoman verkon tapauksessa riittäisikin siirtymämatriisista puolikas:



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

- joskus 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], . . .) 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 $A[i, j] = \infty/0$
  - 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 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 ovat 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. tarkastellaan laskuharjoituksissa 10 olevaa labyrinttiongelmaa

  tehtävänä on etsiä lyhin pisteistä koostuva reitti, joka johtaa ulos labyrintin X:llä merkitystä kohdasta

```
####.##.#
...#....#
#.#.###.#
#....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:

```
111101101
000100001
101011101
100000001
111111111
```

- solmut vastaavat nyt taulukon kohtia, esim. vasemman reunan aukko labyrintista ulos on taulukon kohta $lab[1,0]$
  - ensimmäinen indeksi 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 0
  - kohdasta $lab[1,1]$ kaari kohtiin $lab[1,0]$, $lab[1,2]$ 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 olevista solmuista

- läpikäyntiin on kaksi perusstrategiaa:
  - *leveyssuuntainen* läpikäynti (engl. bredth-first search), ja
  - *syvyyssuuntainen* läpikäynti (engl. depth-first search)

Leveyssuuntainen läpikäynti

- Verkon $G = (V, E)$ *leveyssuuntaisessa läpikäynnissä* tutkitaan mitkä verkon solmuista ovat saavutettavissa annetusta aloitussolmusta $s \in V$

- läpikäynti etenee uusiin solmuihin "taso kerrallaan", eli ensin etsitään mitkä solmut saavutetaan s:stä yhden pituista polkua käyttäen, tämän jälkeen edetään solmuihin mitkä 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. bredth-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 $tree$, eli 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 $color[v] = white$
  - kun solmu v löytyy, asetetaan sen väriksi musta, eli $color[v] = black$

- aluksi kaikille paitsi aloitussolmulle s merkitään $color[v] = white$

- aloitussolmulle s merkataan $color[s] = black$

- 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 mustaksi ja laitettu jonoon

- rivien 8-15 toimintaperiaate
  - aloitussolmu s otetaan jonosta, ja kaikki sen vierussolmut laitetaan jonoon
  - vierussolmujen etäisyydeksi päivitetään 1, niitä leveyssuuntaispuussa edeltäväksi solmuksi asetetaan s ja merkitään solmut löydetyiksi
  - solmu s on nyt käsitelty
  - 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 $color[v] = white$
  - päivitetään jonoon laitettujen etäisyys- ja leveyssuuntaispuutietoa sekä merkitään ne löydetyiksi

- algoritmi

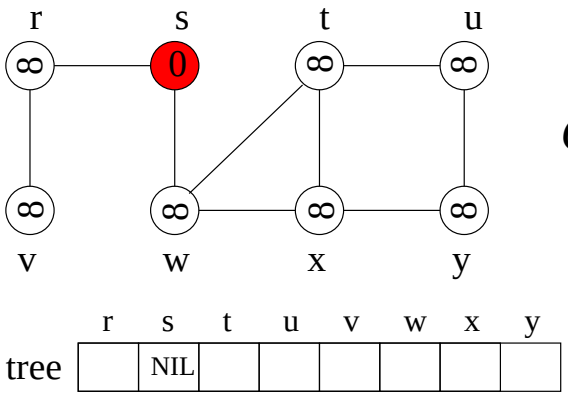
```
BFS(G,s)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3       distance[u] = ∞
4       tree[u] = NIL
5   color[s] = black
6   distance[s] = 0
7   enqueue(Q,s)
8   while ( not empty(Q) )
9       u = dequeue(Q)
10      for jokaiselle solmulle v ∈ Adj[u]      // kaikille u:n vierussolmuille 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)
```

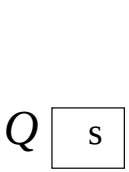
- huomionarvoista algoritmissa on että se toimii sekä suunnatuilla että suuntaamattomilla verkoilla

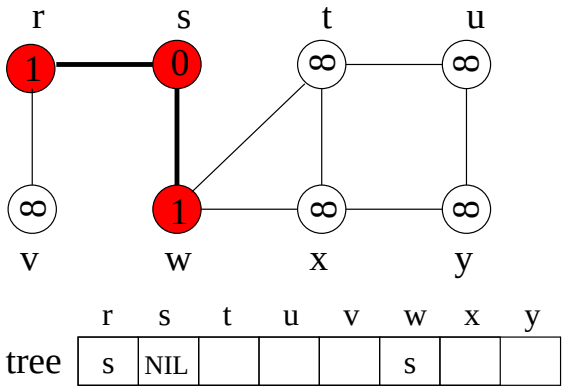
- esimerkki algoritmin toiminnasta seuraavillia sivulla, löydetyt, eli mustat solmut ovat värillisissä kalvoissa punaisia

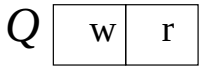
- Algoritmin käyttämien kirjanpitotaulukkojen $distance$ ja $color$ informaatio on merkitty suoraan solmujen yhteyteen

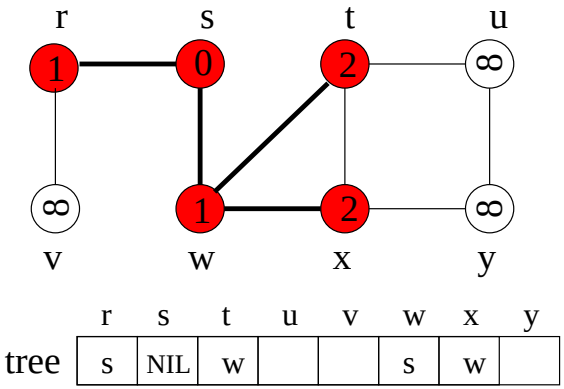
- taulukon $tree$ alkio $tree[v]$ siis kertoo mistä solmusta lyhin polku lähtösolmusta solmuun v saapuu

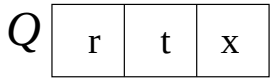


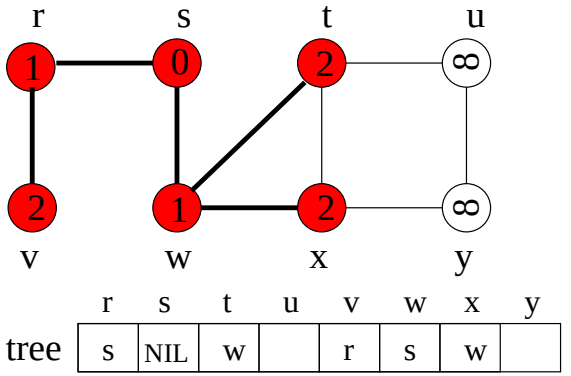


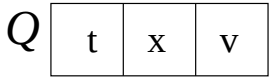


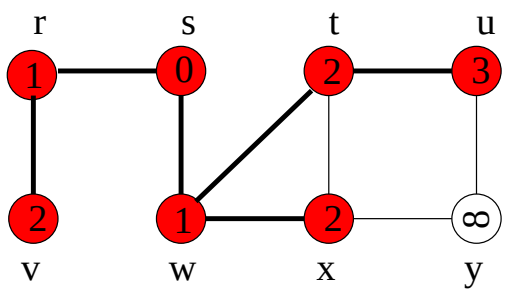






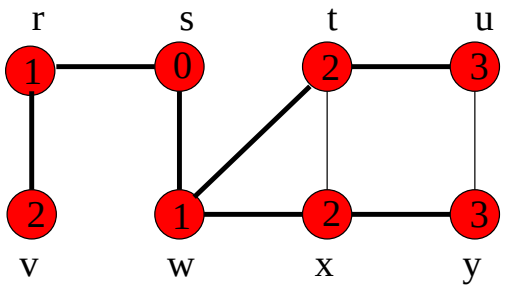






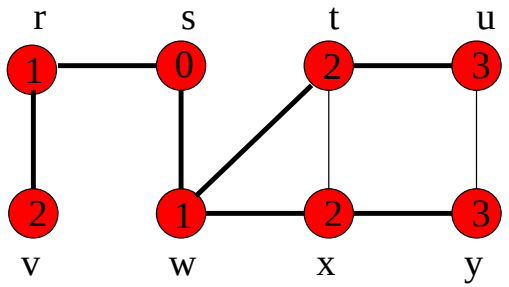
Q | x | v | u |

	r	s	t	u	v	w	x	y
tree	s	NIL	w	t	r	s	w	

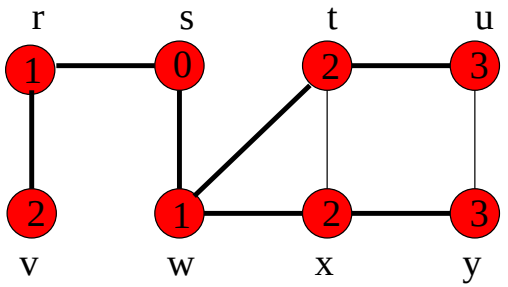


Q | v | u | y |

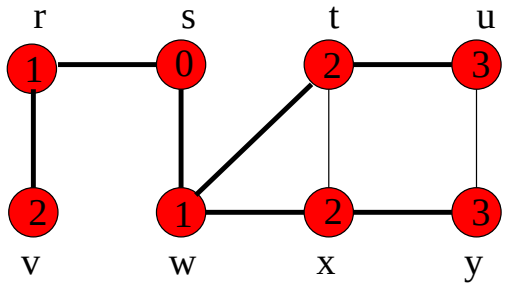
	r	s	t	u	v	w	x	y
tree	s	NIL	w	t	r	s	w	x



Q | u | y |



Q | y |



Q

- 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)
```

- edellä siis esitimme solmujen nimet kirjaimina
- useimmissa ohjelmointikielissä on mahdollista indeksöidä taulukkoja $tree, distance$ ja $color$ myös kirjaimilla sillä kirjaimet tulkitaan ascii-koodeiksi
- paremmin yleistyvä tapa esim. Javassa olisi esittää $tree, distance$ ja $color$ HashMap:in avulla, tällöin avain, eli solmun "nimi" voi olla myös esim. merkkijono tai joku muu sovelluksen kannalta sopiva olio
- koska värejä on vain kaksi, voi tiedon $color$ tallettaa boolean-muotoisena
- seuraavalla sivulla on luonnos leveyssuuntaisen läpikäynnin Java-toteutuksesta tilanteessa, jossa solmut on nimetty merkkijonoina
  - $distance$ ja $color$ ovat HashMap-olioita
  - leveyssuuntaispuuta ei koodissa tilan puutteen takia muodosteta, joten $tree$ puuttuu
  - jonona käytetään LinkedList:ia
  - algoritmin toteuttavan metodin parametreina on kaikkien solmujen joukko eli String-taulukko `solmut` sekä aloitussolmu `s`
  - algoritmi olettaa, että on olemassa metodi `adj`, joka palauttaa parametrina olevan solmun vierussolmut
  - huomaamme, että Java ei juuri poikkea pseudokoodista

```
// palauttaa parametrina olevan solmun vierussolmut
public static String[] adj(String v){ ... }

public void static bfs(String[] solmut, String s){
    LinkedList<String> jono = new LinkedList();
    HashMap<String, Integer> distance = new HashMap();
    HashMap<String, Boolean> color = new HashMap();

    for ( String solmu : solmut ) {
        color.put(solmu, false);
        distance.put(solmu, Integer.MAX_VALUE);
    }
    color.put(s, true);
    distance.put(s, 0);
    jono.addLast( s );

    while( !jono.isEmpty() ){
        String u = jono.removeFirst();
        for ( String v : adj(u) )
            if ( color.get(v)==false ) {
                color.put(v, true);
                distance.put(v, distance.get(u)+1 );
                jono.addLast(v);
            }
    }
}
```

- leveyssuuntaisen läpikäynnin aikavaativuus:
  - alustukseen (rivit 1-6) kuluu aikaa $\mathcal{O}(|V|)$
  - koska jonoon laitettava solmu värjätään heti mustaksi eikä väri enää muutu, 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 $\mathcal{O}(|V|)$

Syvyyssuuntainen läpikäynti

- toinen verkkojen läpikäyntitavoista siis 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
- 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, nyt värejä on kolme:
  - 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 merkkaa, että solmu on jo löydetty, mutta sen käsittely ei ole vielä kokonaisuudessaan ohi
- seuraavassa algoritmi, joka selvittää aloitussolmusta s saavutettavissa olevat solmut

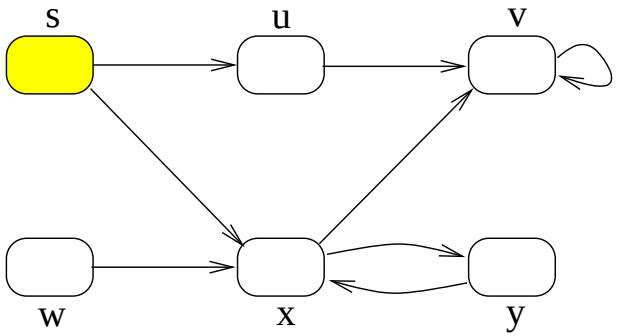
```
DFS(G,s)
1   for jokaiselle solmulle u ∈ V
2       color[u] = white
3   DFS-visit(G,s)

DFS-visit(G,u)
4   color[u] = gray
5   for jokaiselle solmulle v ∈ Adj[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
```

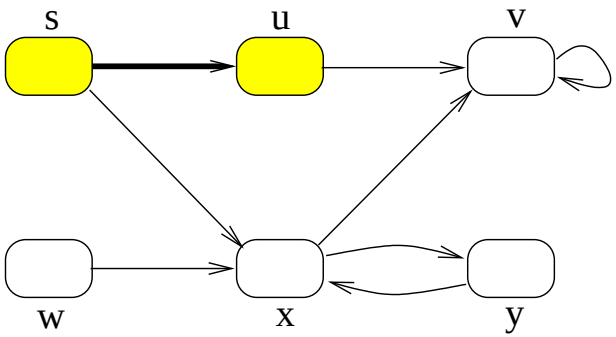
- kolmas väri eli harmaa ei ole tarpeen algoritmin toiminnan kannalta, eli solmu voitaisiin periaatteessa värjätä heti mustaksi

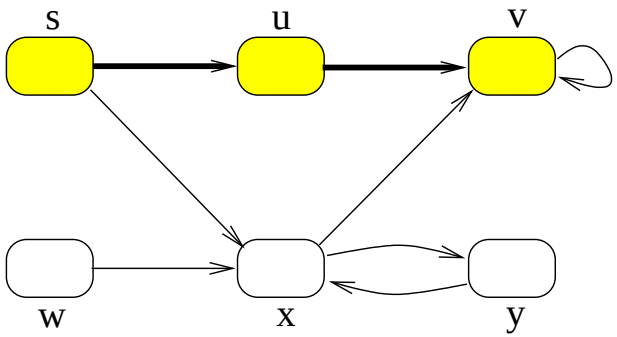
- tarvitsemme harmaata väriä kohta esitettävässä syvyyssuuntaisen läpikäynnin sovelluksessa, joten otamme mukaan sen jo nyt

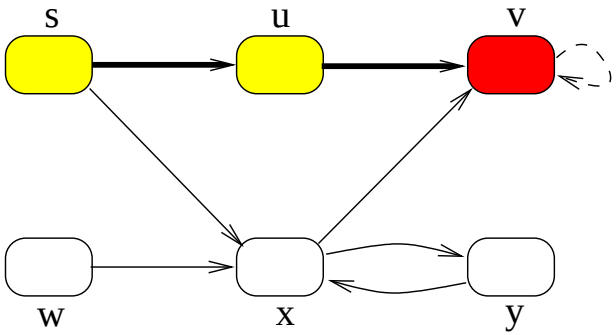
- toimintaperiaate
  - alustusvaiheessa kaikki solmut merkataan löytymättömiksi eli valkoisiksi
  - läpikäynti aloitetaan kutsumalla DFS-visit aloitussolmulle s
  - kun läpikäynti etenee solmuun, merkataan että solmu on löydetty ja että sen käsittely on kesken eli solmu muuttuu harmaaksi (rivi 4)
  - 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 päätty

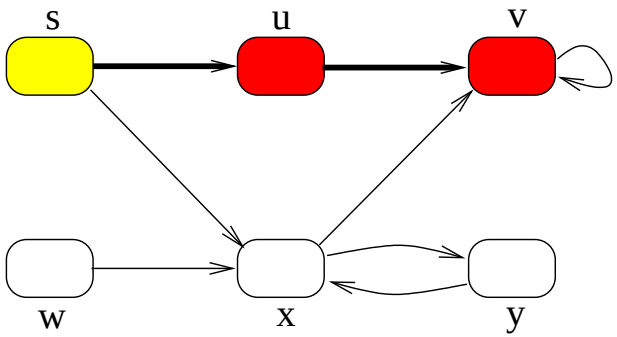
- esimerkki algoritmin toiminnasta seuraavalla sivulla. Värillisessä kuvassa harmaat solmut ovat keltaisia ja mustat punaisia

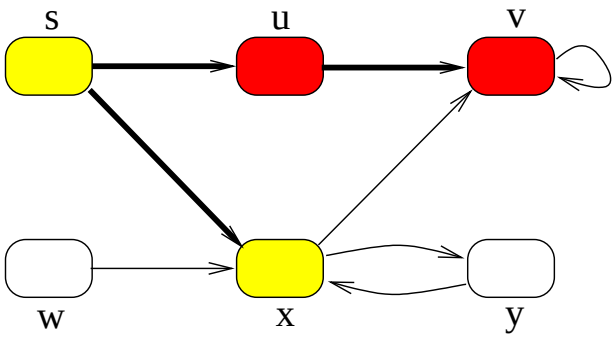

s
u
v
w
x
y

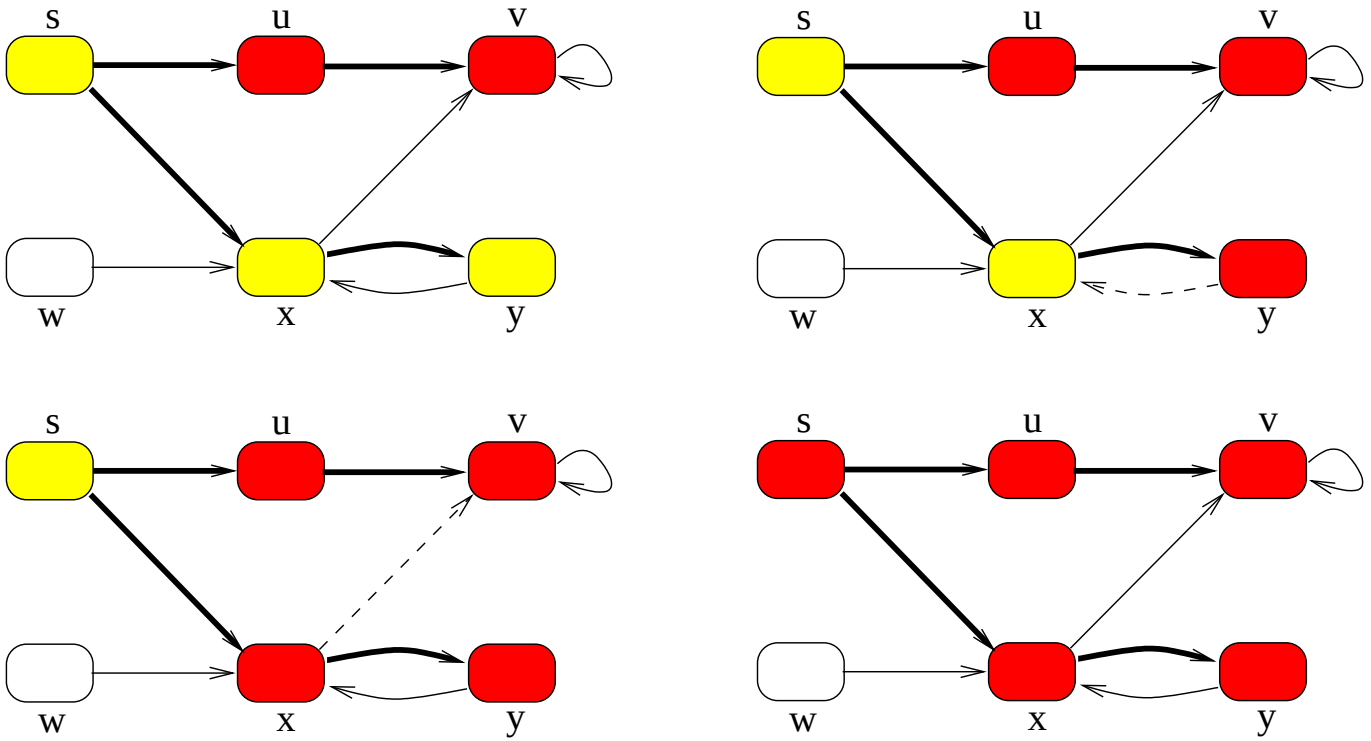

s
u
v
w
x
y


s
u
v
w
x
y


s
u
v
w
x
y


s
u
v
w
x
y


s
u
v
w
x
y



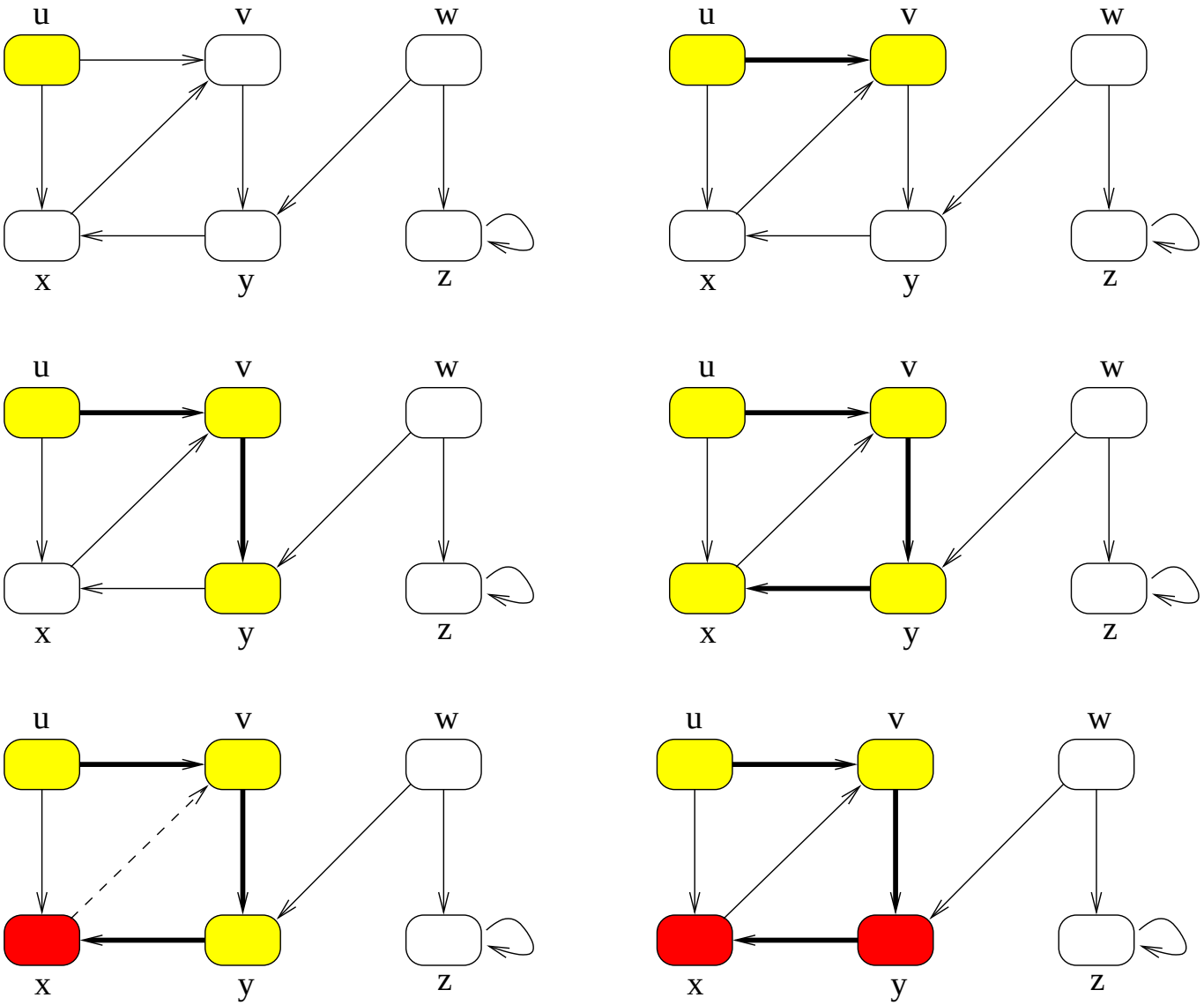
- syvyyssuuntainen läpikäynnin aikana muodostuu *syvyyssuuntaispuu*, joka koostuu niistä kaarista, joita pitkin läpikäynti eteni aiemmin löytymättömiin solmuihin

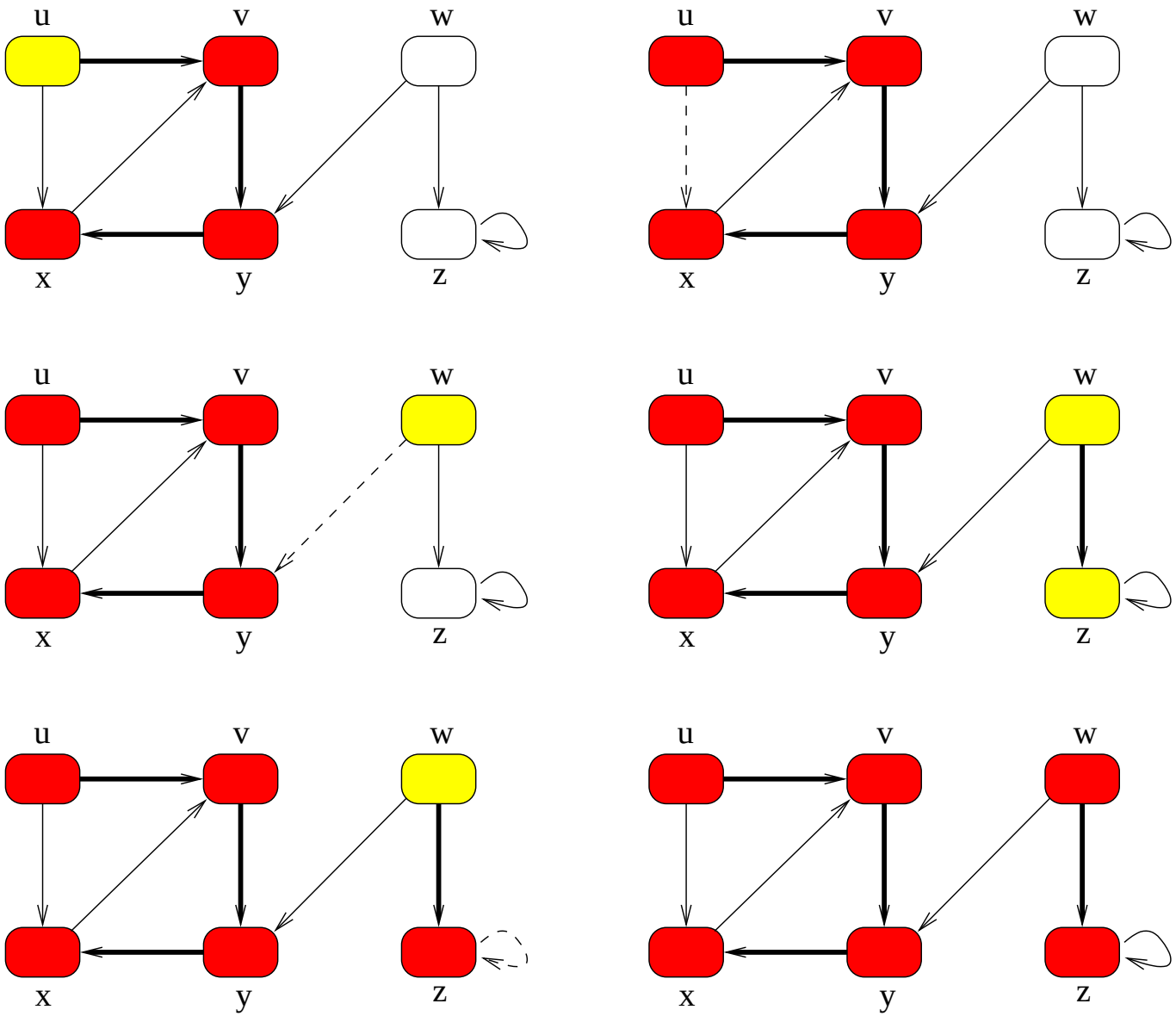
- kuten leveyssuuntaisen läpikäynnin yhteydessä, syvyyssuuntaispuun kaaret olisi tarvittaessa helppo kirjata algoritmin yhteydessä esim. erilliseen taulukkoon $tree$, johon asetettaisiin $tree[v] = u$ jos läpikäynti eteni solmusta u solmuun v

- 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:st' 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(u)
```

- 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-visitin 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 aputaukko 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

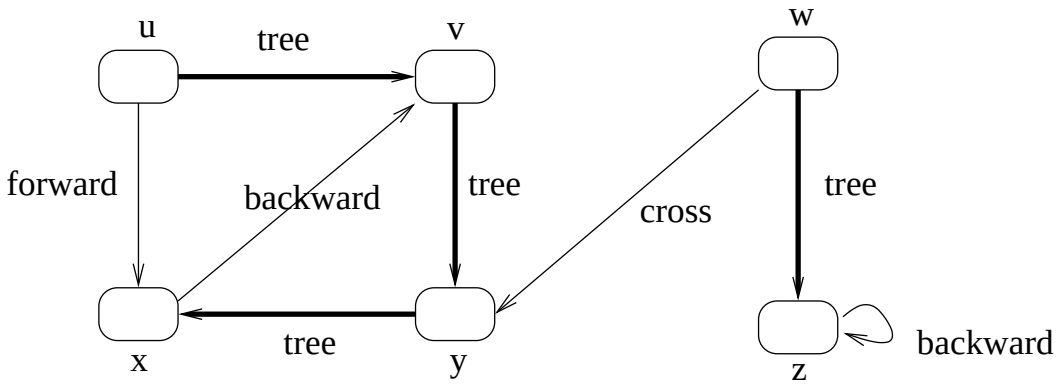
- läpikäynti etenee *puunkaaria* (engl. tree arc) pitkin uusiin vielä löytymättömiin solmuihin, eli puunkaari kohdistuu valkoiseen solmuun

- *takautuva kaari* (engl. backward arc) kohdistuu taaksepäin syvyyssuuntaisessa puussa, 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 syvyyssuuntaisessa puussa, 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 (ks. kalvot 411 ja 412) etenee aluksi valkoisia solmuja pitkin reittiä $u \longrightarrow v \longrightarrow y \longrightarrow x$, kaikki nämä harmaasta valkoiseen solmuun kohdistuvat ovat puunkaaria

- kun ollaan solmussa x, ainoa kaari kohdistuu harmaasen solmuun v, joka siis on jo löydetty mutta jonka vierussolmuja ei ole käsitelty loppuun, $x \longrightarrow v$ on siis takautuva kaari

- kun läpikäynti on peruuttanut takaisin solmuun u, tutkitaan kaari $u \longrightarrow x$ joka kohdistuu u:n seuraajaan syvyysuuntaispuussa, $u \longrightarrow x$ siis on etenevä kaari

- kaari $w \longrightarrow z$ on kahden eri syvyyssuuntaispuussa sijaitsevan solmun välinen, eli kyseessä on poikittaiskaari

- huom: kaarten luokittelu on riippuvanen, mistä solmusta läpikäynti aloitetaan

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 $Adj[u]$
  - 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 ∈ Adj[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 $Adj[u]$ 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

- Perustallaan seuraavassa vielä asia vielä 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

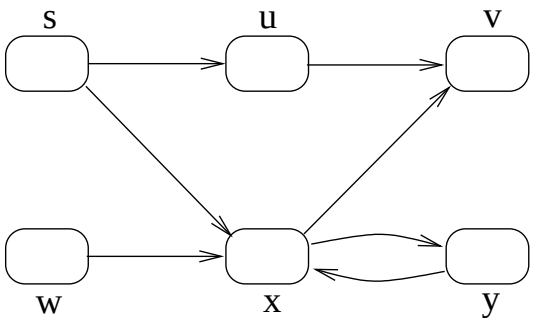
  Kun läpikäynti tulee solmuun v, ei oletuksen mukaan u:ta eikä muita syklin solmuja ole vielä löydetty ja koska $v \rightsquigarrow u$ läpikäynti etenee solmuun u

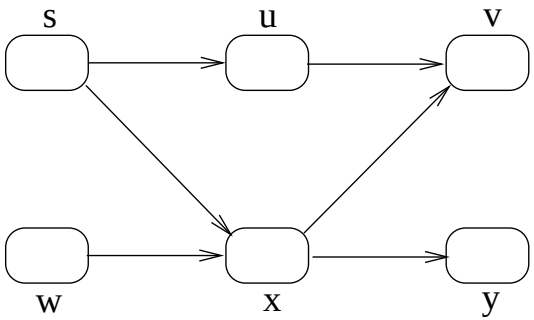
  Kun u:n vierussolmuja tutkitaan, havaitaan, että u:sta on kaari harmaaseen solmuun v eli algoritmi löytää syklin

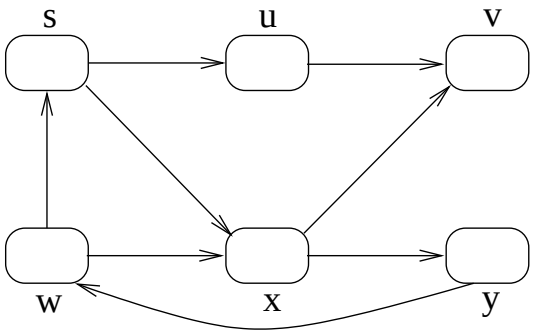
- Algoritmi siis löytää syklin jos ja vain jos verkossa on sykli, eli algoritmi toimii oikein

- Aika- ja tilavaativuus algoritmilla on tietenkin sama kuin modifioimattomalla syvyyssuuntaisella läpikäynnillä

- esim: miten syklien etsintä toimisi seuraavissa verkoissa?



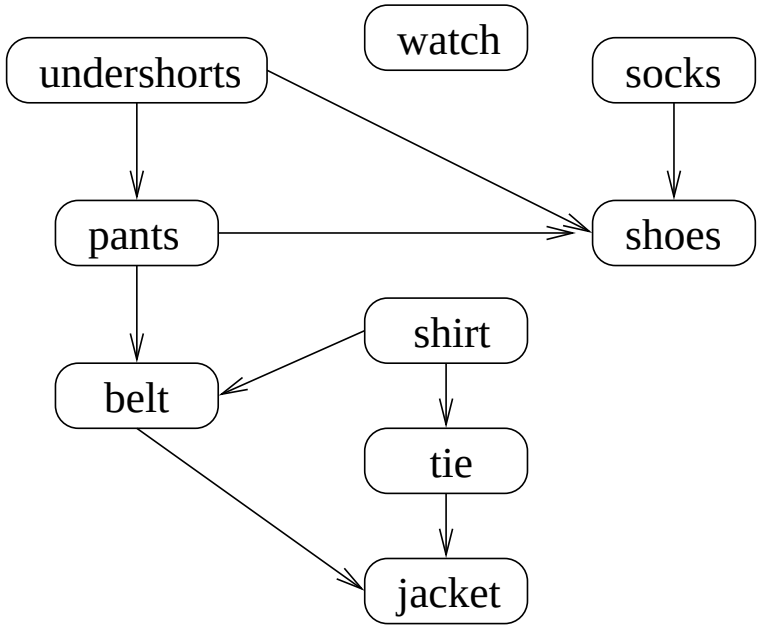




Topologinen järjestäminen

- syklittömien suunnattujen verkkojen (engl. Directed Asyclic Graph, dag) avulla voimme kuvata tapahtumien välisiä riippuvuuksia

- alla oleva verkko sisältää pukeutumiseen kannalta oleelliset riippuvuudet:



- eli esim. sukat on laitettava ennen kenkiä koska verkossa kaari $socks \rightarrow shoes$

- kello voidaan laittaa käteen missä vaiheessa tahansa koska sillä ei ole mitään riippuvuutta

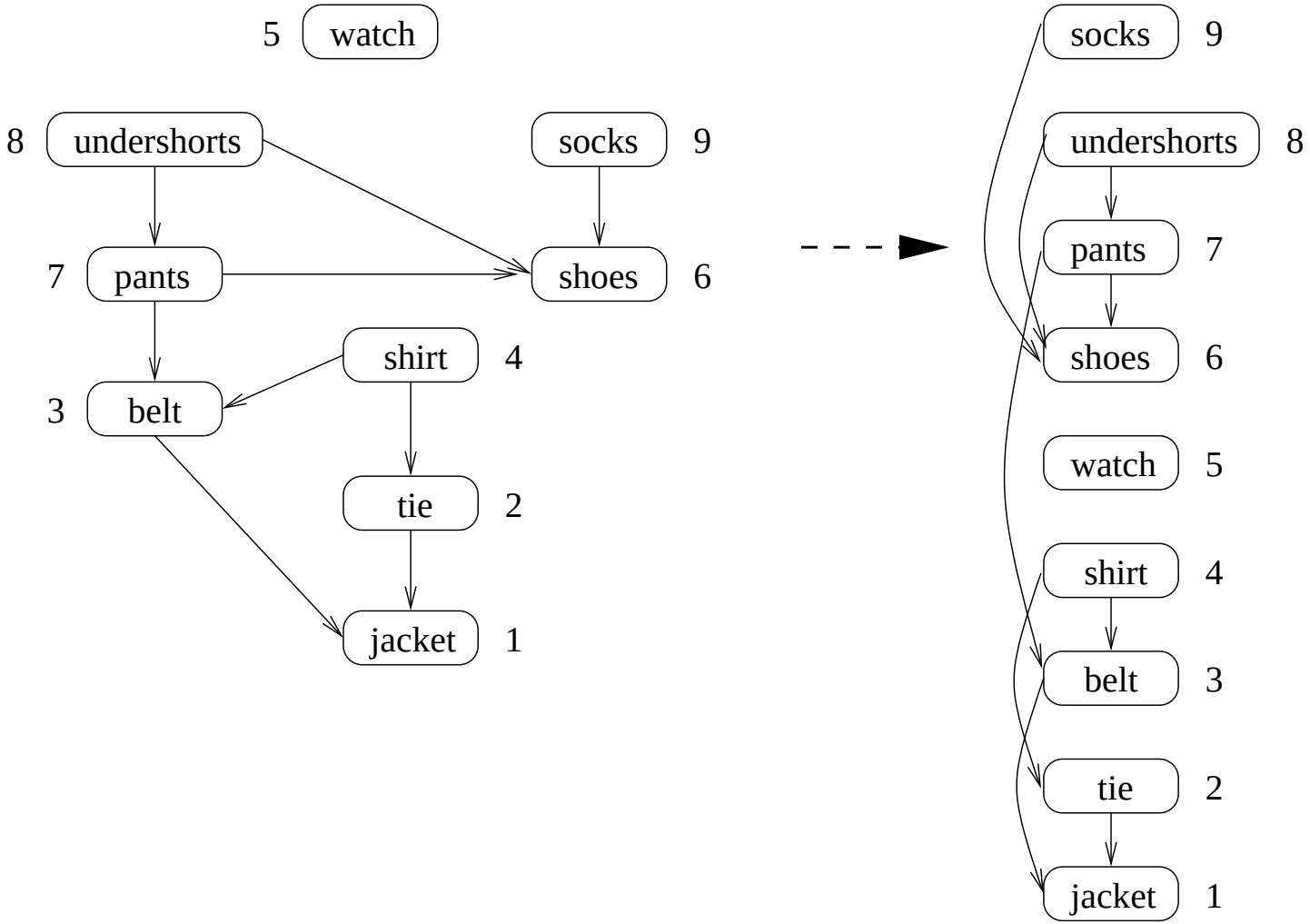
- herää kysymys voisimmeko järjestää asiat sellaiseen lineaariseen järjestykseen että voisimme pukea vaatekappaleen kerrallaan siten että mikään riippuvuuksista ei rikkoudu, eli
  - jos riippuvuusverkossa on kaari $u \rightarrow v$, tulee u järjestyksessä ennen v:tä
- tällaista järjestystä kutsutaan *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 kuinka monentena solmun käsittely on valmistunut

  solmujen järjestys pinossa siis on käänteinen numerojärjestys



- perustellaan algoritmin oikeellisuus vielä hieman tarkemmin

- algoritmi toimii oikein jos kaikille solmuille v, w missä v topologisessa järjestyksessä solmun w edellä, ei ole olemassa kaarta $w \rightarrow v$

- oletetaan, että kaari $w \rightarrow v$ on olemassa, ja näytetään, että tästä seuraa ristiriita

  koska v on solmua w edellä topologisessa järjestyksessä, on sen käsittely päättynyt myöhemmin kuin w:n käsittely

  - jos w:n käsittely olisi aloitettu v:n löytymisen jälkeen, olisi verkossa polku $v \rightsquigarrow w$ sekä kaari $w \rightarrow v$ 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 $w \rightarrow v$ seurauksena edennyt solmuun v ja v:n käsittelyn olisi täytynyt päättyä ennen w:n käsittelyn päättymistä. Tämä taas on ristiriita solmujen topologisen järjestyksen takia

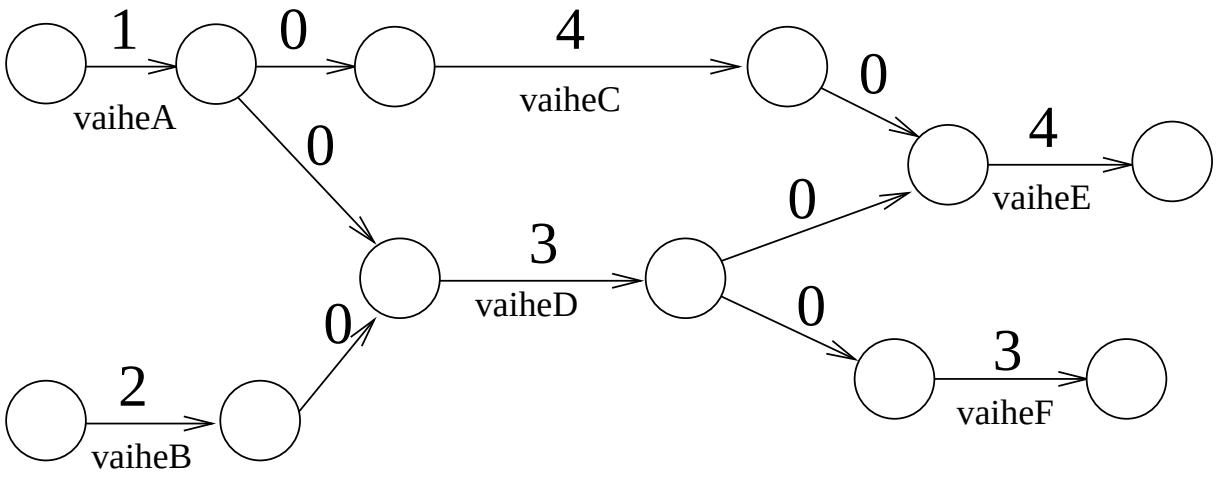
- on siis osoitettu, että kaaren $w \rightarrow v$ olemassaolo johtaisi ristiriitaan, eli topologisen järjestyksen rikkovaa kaarta ei voi olla olemassa, siispä

- 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 *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

$$h[u] = \max\{c + heaviest[v] \mid \text{solmusta } u \text{ on solmuun } v \text{ kaari, jonka paino on } c\}$$

- 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

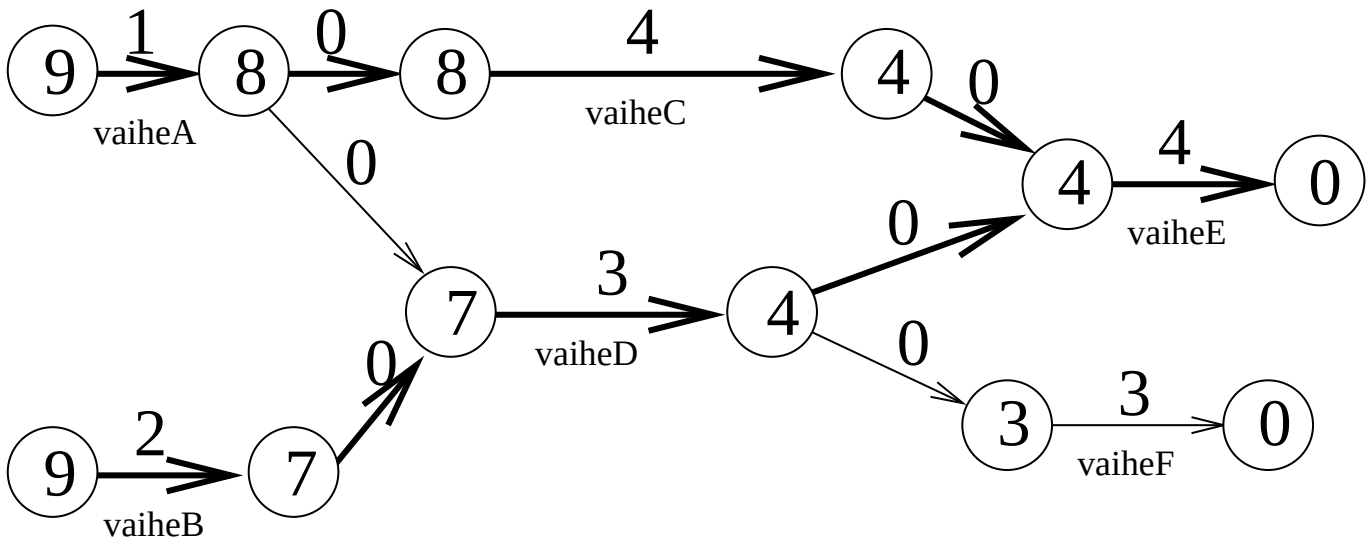
- projektin kokonaiskesto on

$$t = \max_{u \in V} heaviest[u]$$

- 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 löytäminsestä
- Jokaiseen solmuun on merkitty sen $heaviest$-arvo
- Kriittiset polut on vahvennettu



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

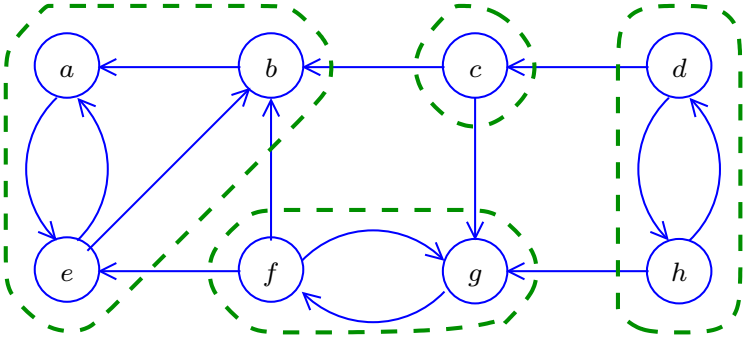
- jos verkko ei ole vahvasti yhtenäinen, se voidaan jakaa *vahvasti yhtenäisiin komponentteihin* $V_1, V_n, \ldots, V_n$ seuraavasti:

  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

- Tarkastellaan seuraavaa suunntattua verkkoa:



- Verkon vahvasti yhtenäiset komponentit ovat

$$\begin{aligned} V_1 &= \{ a, b, e \} \\ V_2 &= \{ c \} \\ V_3 &= \{ d, h \} \\ V_4 &= \{ f, g \} \end{aligned}$$

- Vaikka verkon vahvasti yhtenäisten komponenttien tuntemisen hyödyllisyys ei ole päällepäin kovin ilmeinen seikka, on vahvasti yhtenäisten komponenttien selvittämiselle paljon käyttöä erilaisissa sovellustilanteissa

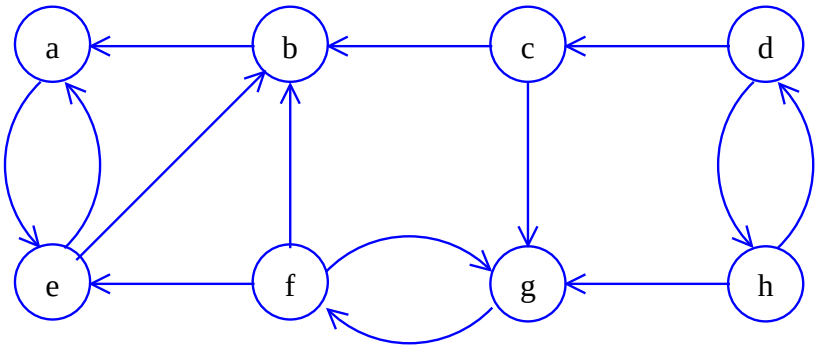
- Vahvasti yhtenäiset komponentit voidaan muodostaa seuraavalla algoritmilla:

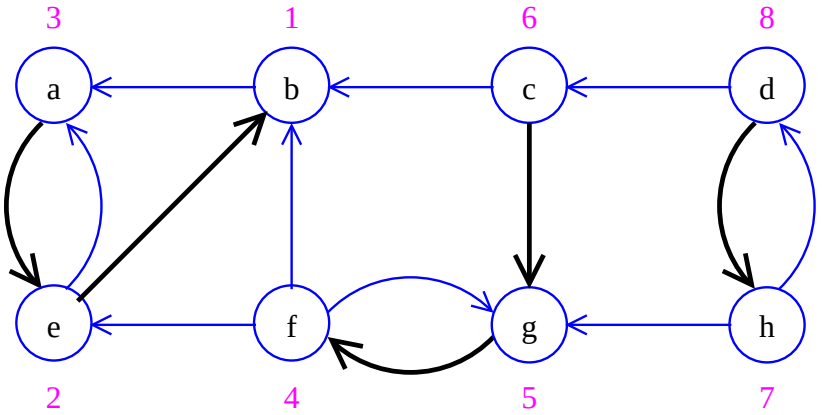
  Strongly-Connected-Components(G)

  1. Suorita verkon $G = (V, E)$ syvyyssuuntainen läpikäynti.
     Kun solmun v käsittely on ohi, eli asetetaan $color[v] = black$, lisätään se pinoon P
  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 käännettynä 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(u) alkaa pinosta P ensimmäisenä löytyvästä vielä läpikäymättömästä solmusta u.

- Koska verkon transpoosi voidaan muodostaa ajassa $\mathcal{O}(|\,V\,| + |\,E\,|)$, vahvasti yhtenäiset komponentit selvittävän algoritmin pahimman tapauksen aikavaativuus on $\mathcal{O}(|\,V\,| + |\,E\,|)$ ja työtilavaativuus $\mathcal{O}(|\,V\,|)$ kuten syvyyssuuntaisella läpikäynnillä

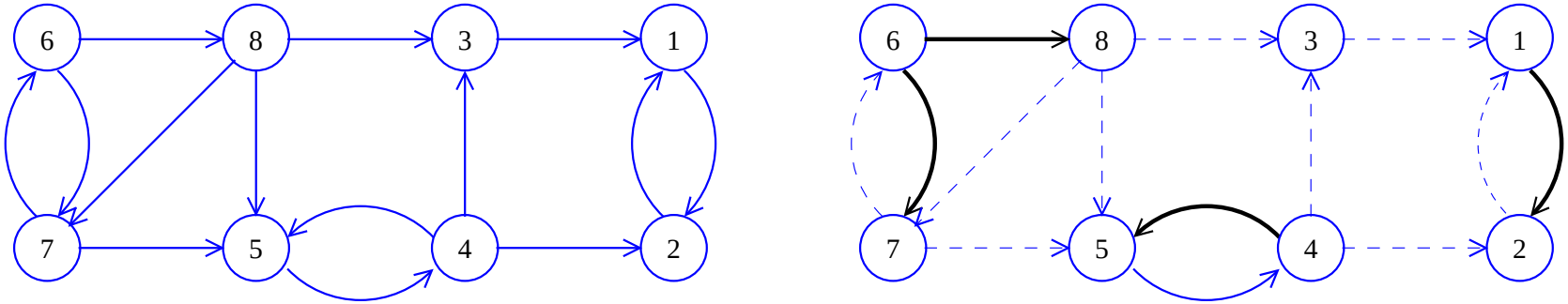
- Tarkastellaan esimerkkiä
- Vasemmalla verkko G jonka vahvasti yhtenäiset komponentit halutaan selvittää
- Oikealla verkon G syvyyssuuntaisen läpikäynnin tulos, solmut on numeroitu siinä järjestyksessä missä niiden käsittely valmistui, eli missä järjestyksessä ne laitettiin pinoon P





- 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

- Tehdään ensin seuraava huomio:

  jos v ja w ovat solmuja, jotka kuuluvat samaan verkon G vahvasti yhtenäiseen komponenttiin, niin G:ssä on polut $v \rightsquigarrow w$ ja $w \rightsquigarrow v$. Siispä myös transpoosiverkossa G^{T} on polut $v \rightsquigarrow w$ ja $w \rightsquigarrow v$.

- Algoritmin oikeellisuuden perustelu jakautuu kahteen osaan:

- Todistuksen ensimmäinen suunta näyttää, että jos mielivaltaiset kaksi solmua v ja w kuuluvat G:n vahvasti yhtenäiseen komponenttiin, niin ne kuuluvat samaan transpoosiverkon G^{T} syvyyssuuntaiseen puuhun

  Oletetaan että G^{T}:ssä tehtävä läpikäynti löytää ensin solmun v. Koska yllä tehdyn havainnon perusteella v ja w ovat saavutettavissa toisistaan verkossa G^{T}, päätyy myös w samaan syvyyssuuntaispuuhun G^{T}:ssä.

  Vastaavasti, jos solmu w löydettäisiin ensin, päätyy v samaan syvyyssuuntaispuuhun

  Eli kaikki G:n tiettyyn vahvasti yhtenäiseen komponenttiin kuuluvat solmut päätyvät samaan transpoosiverkon syvyyssuuntaispuuhun

- Todistuksen toinen suunta näyttää, että jos mielivaltaiset kaksi solmua päätyvät transpoosiverkossa samaan syvyyssuuntaispuuhun, ne kuuluvat G:n samaan vahvasti yhteiseen komponenttiin

  Oletetaan että v ja w kuuluvat samaan verkon G^{T} syvyyssuuntaispuuhun. Merkitään x:llä kyseisen syvyyssuuntaispuun juurisolmua.

  Koska v on x:n seuraaja verkon G^{T} syvyyssuuntaispuussa, on solmusta v solmuun x polku alkuperäisessä verkossa G.

  Näytetään nyt, että verkossa G täytyy myös olla polku solmusta x solmuun v.

  Kun etsintä G^{T} aloitti solmusta x, ei solmussa v oltu vielä vierailtu, eli verkon G syvyyssuuntaisessa läpikäynnissä v:n käsittely on loppunut ennen x:n käsittelyn loppumista.

  Solmun v käsittely ei ole voinut alkaa G:n läpikäynnissä ennen x:n käsittelyä, sillä muuten verkossa G olevan polun $v \rightsquigarrow x$ takia x olisi sekä löydetty että käsitelty ennen v:tä. Siispä solmun v käsittely on aloitettu x:n jo löydyttyä, mutta ennen kuin x on käsitelty, eli x:n ollessa vielä harmaa.

Edellisestä seuraa, että verkon G syvyyssuuntaisessa läpikäynnissä v:ssä vieraillaan kun solmun x kautta kulkeva läpikäynti on menossa. Eli verkossa G on polku $x \rightsquigarrow v$. Aiemmin todetun perusteella verkossa G on myös polku $v \rightsquigarrow x$ eli solmut v ja x kuuluvat samaan vahvasti yhtenäiseen komponenttiin.

Vastaavanlaisella päättelyllä voidaan näyttää, että verkossa G on polut $x \rightsquigarrow w$ ja $w \rightsquigarrow x$, eli näinollen solmut v ja w ovat saavutettavissa toisistaan solmun x kautta, eli ne kuuluvat samaan vahvasti yhtenäiseen komponenttiin verkossa G.

On siis osoitettu myös todistuksen toinen suunta, eli jos mielivaltaiset kaksi solmua päätyvät transpoosiverkossa samaan syvyyssuuntaispuuhun, ne kuuluvat G:n samaan vahvasti yhteiseen komponenttiin

- 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

- Tarkastellaan painotettuja verkkoja ja tulkitaan kaaren paino sen yhdistämien solmujen etäisyydeksi
- kaaripainon voi tulkita myös esim. ajaksi mikä 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
- esitellään ongelmalle ratkaisuksi *Dijkstran algoritmi* joka olettaa että kaaripainot ovat *positiivisia*
- Negatiivisten kaaripainojen tapauksessa lyhimmät polut voidaan ratkaista esim. Dijkstran jälkeen käsiteltävällä Floyd-Warshallin algoritmilla tai Bellman-Ford-algoritmilla jota ei kurssilla käsitellä

- Olkoon $G = (V, E)$ suunnattu verkko ja s eräs verkon solmu
- oletetaan että jokaiseen kaareen $(u, v) \in E$ on liitetty pituus $w(u, v)$ joka on ei-negatiivinen reaaliluku
- oletetaan lisäksi että jos $(u, v) \notin E$ niin $w(u, v) = \infty$ ja $w(v, v) = 0$, eli jos solmuja ei yhdistä kaari, niiden välisen reitin pituus on ääretön ja jokaisesta solmusta matka itseensä on nolla
- Dijkstran algoritmi pitää yllä joukkoa S joka muodostuu solmuista joiden lyhin etäisyys solmuun s on jo selvitetty
- algoritmi käyttää aputaulukkoja $distance$ ja $path$, joihin talletetaan tietoa verkon solmuihin liittyen
  - $distance[v]$ kertoo mikä on solmun v *etäisyysarvio* solmusta s
  - $path[v]$ on se joukon S solmu, joka edeltää solmua v toistaiseksi tunnetulla lyhimmällä polulla
- lyhin tunnettu polku $s \rightsquigarrow v$ sekä sen etäisyysarvio, joista algoritmi pitää kirjaa, ovat sellaisia että polun kaikki solmut mahdollisesti solmua v lukuunottamatta kuuluvat joukkoon S

- aluksi algoritmi asettaa kaikkien solmujen (paitsi solmun s) etäisyysarvioksi äärettömän eli $distance[v] = \infty$, sekä $path[v]$ arvoksi NIL

- 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

- ensimmäisessä toistoaskeleessa tulee valituksi aloitussolmu s sillä $distance[s] = 0$
  - solmu s siis lisätään joukkoon S
  - kaikille s:n vierussolmuille v asetetaan uusi etäisyysarvio, joka on nyt $distance[s] + w(s, v) = 0 + w(s, v) = w(s, v)$, eli sama kuin etäisyys solmujen s ja v välillä
  - attribuuttien $path[v]$ arvoksi tulee s, sillä lyhin polku solmusta s solmuun v saapuu solmun s kautta

- toisessa toistoaskeleessa tulee valituksi se solmu u jonka etäisyys solmuun s on lyhin
  - solmu u siis lisätään joukkoon S
  - kaikille u:n vierussolmuille v joiden entinen etäisyysarvio solmuun s on suurempi kuin $distance[u] + w(u, v)$:
  - asetetaan uusi etäisyysarvio, joka on nyt $distance[u] + w(u, v)$, eli sama kuin etäisyys $s \rightarrow u$ + etäisyys $u \rightarrow v$, ja
  - polkutiedon $path[v]$ arvoksi päivitetään u, sillä lyhin tunnettu polku solmusta s solmuun v saapuu solmun u kautta

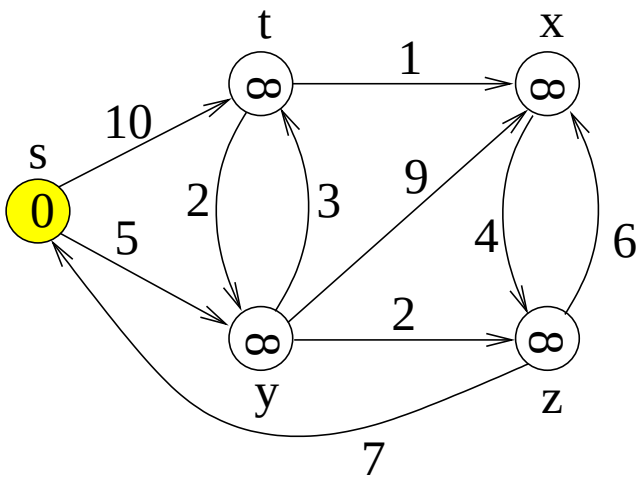
- sama jatkuu seuraavissa toistoaskeleissa

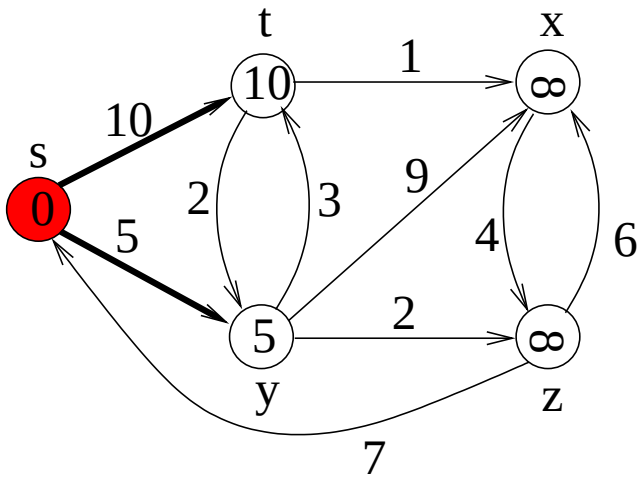
  valitaan solmu u jonka etäisyys lähtösolmuun s on lyhin
  - solmu u lisätään joukkoon S
  - kaikille u:n vierussolmuille v joiden entinen etäisyysarvio solmuun s on suurempi kuin $distance[u] + w(u, v)$, päivitetään etäisyysarvio ja
  - polkutiedoksi $path[v]$ arvoksi päivitetään u, sillä lyhin polku solmusta s solmuun v saapuu solmun u kautta

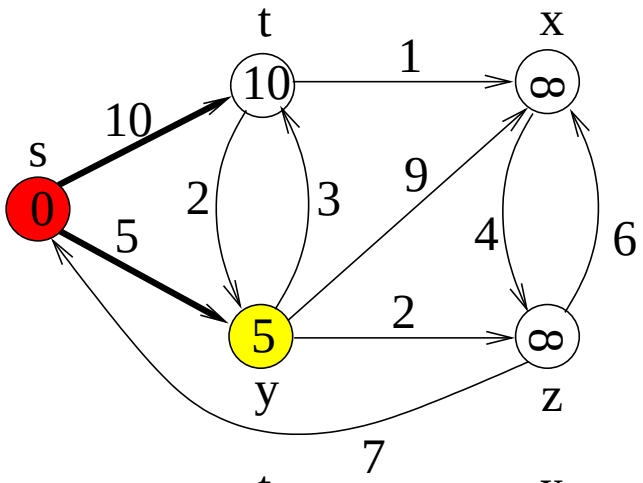
- algoritmi osittain abstraktissa muodossa

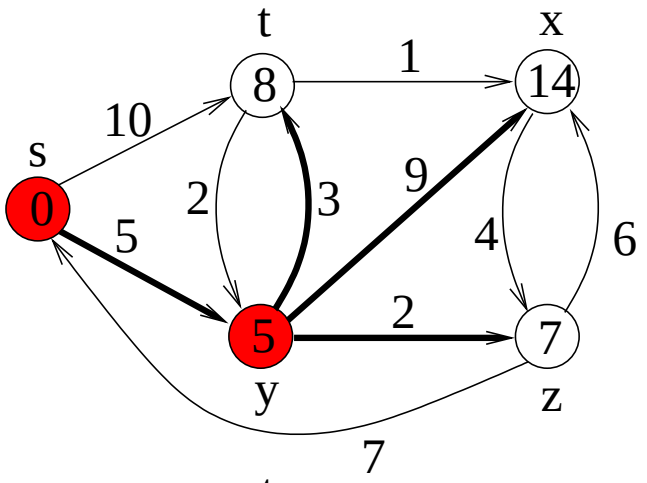
```
Dijkstra(G,w,s)
1   for kaikille solmuille v ∈ V
2       distance[v] = ∞
3       path[v] = NIL
4   distance[s] = 0
5   S = ∅
6   while ( kaikki solmut eivät vielä ole joukossa S )
7       valitse solmu u ∈ V \ S, jonka etäisyysarvio lähtösolmuun s on lyhin
8       S = S ∪ {u}
9       for jokaiselle solmulle v ∈ Adj[u]       // kaikille u:n vierussolmuille v
10          if distance[v] > distance[u] + w(u,v)
11              distance[v] = distance[u]+w(u,v)
12              path[v] = u
```

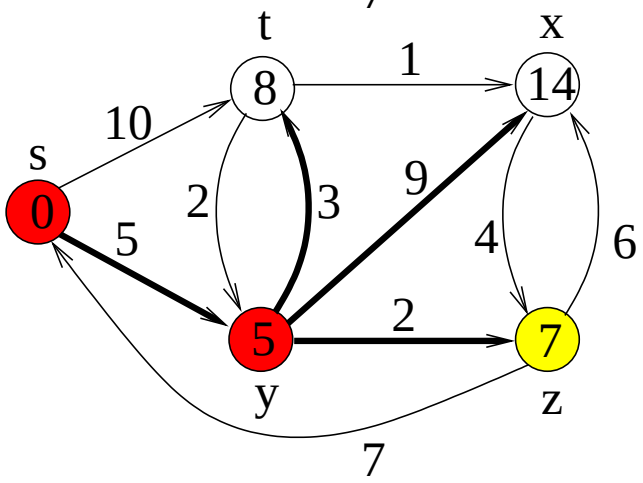
- seuraavilla kalvoilla esimerkki algoritmin toiminnasta
  - joukkoon S lisätyt alkiot tummanharmaita (värikuvassa punaisia)
  - käsittelyvuorossa oleva alkio vaaleanharmaa (värikuvassa keltainen)

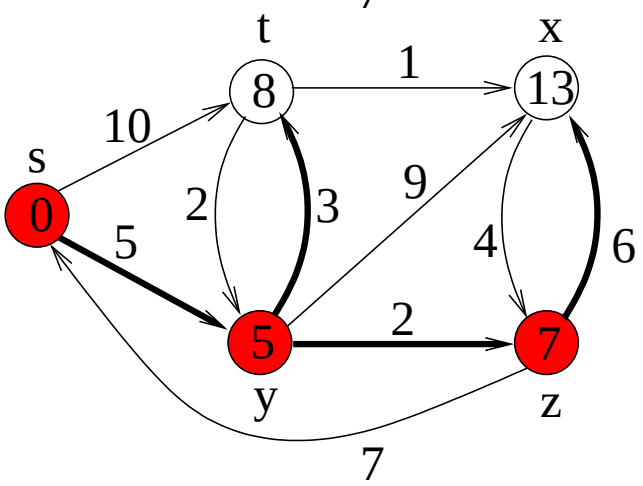

t
x
s
y
z
0
∞
∞
∞
∞
10
1
2
3
9
4
6
5
2
7

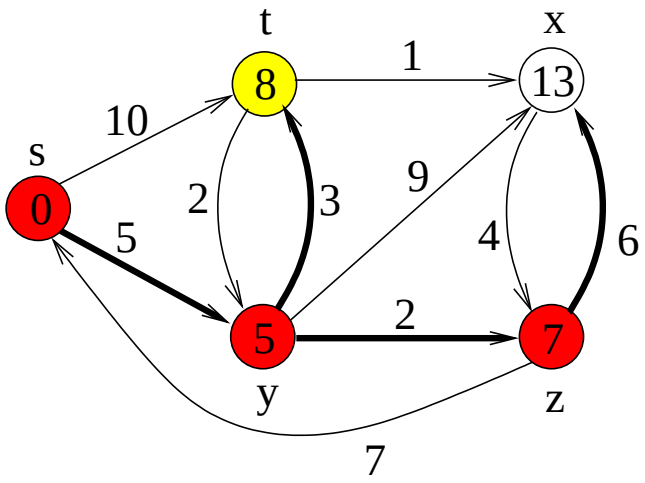

t
x
s
y
z
0
10
∞
5
∞
10
1
2
3
9
4
6
5
2
7

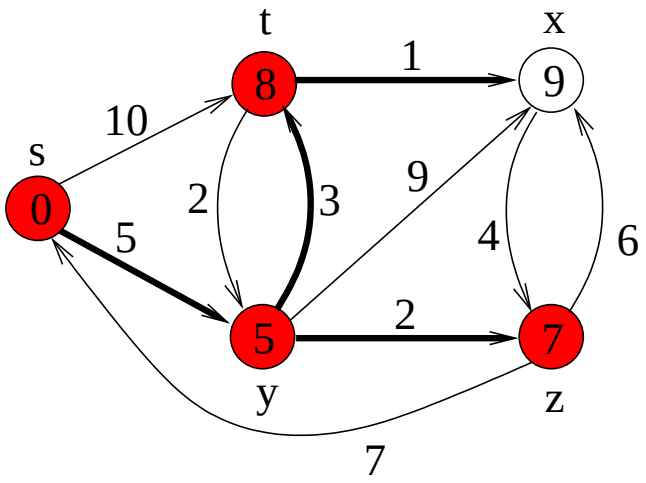

t
x
s
y
z
0
10
∞
5
∞
10
1
2
3
9
4
6
5
2
7

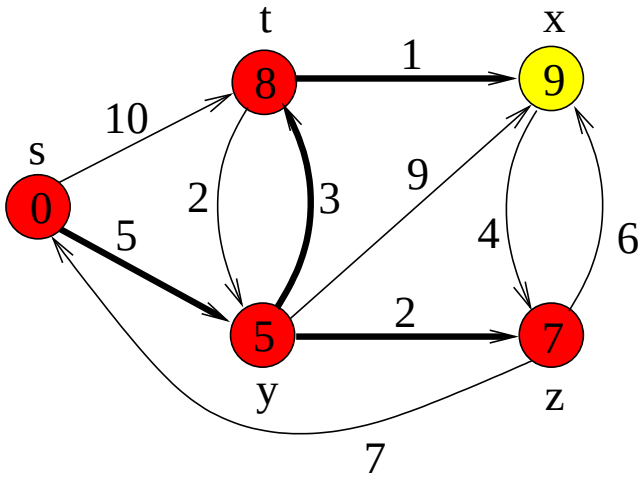

t
x
s
y
z
0
8
14
5
7
10
1
2
3
9
4
6
5
2
7

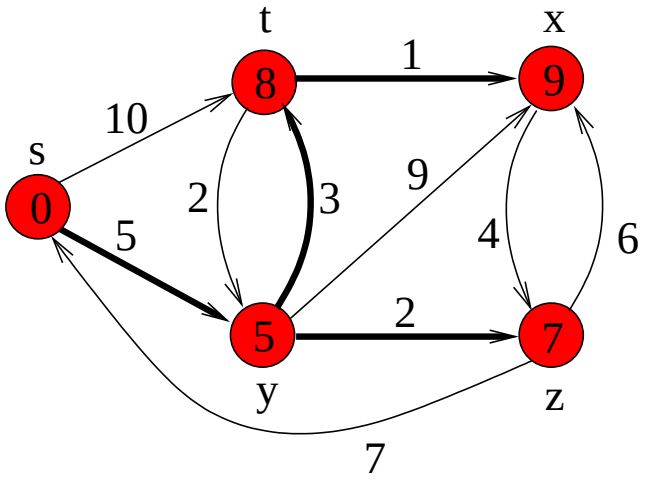

t
x
s
y
z
0
8
14
5
7
10
1
2
3
9
4
6
5
2
7


t
x
s
y
z
0
8
13
5
7
10
1
2
3
9
4
6
5
2
7









- tieto lyhimmistä poluista on kuvattu paksunnettuina kaarina

- miten algoritmin rivi 7, 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(G,w,s)
1   for kaikille solmuille v ∈ V
2       distance[v] = ∞
3       path[v] = NIL
4   distance[s] = 0
5   S = ∅
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      S = S ∪ {u}
11      for jokaiselle solmulle v ∈ Adj[u]      // kaikille u:n vierussolmuille v
12          if distance[v] > distance[u] + w(u,v)
13              distance[v] = distance[u]+w(u,v)
14              path[v] = u
15              heap-decrease-key(H,v,distance[v])
```

- Huom: joukkoa S ei oikeastaan enää tarvita, sillä joukossa S ovat täsmälleen ne alkiot jotka eivät ole keossa H

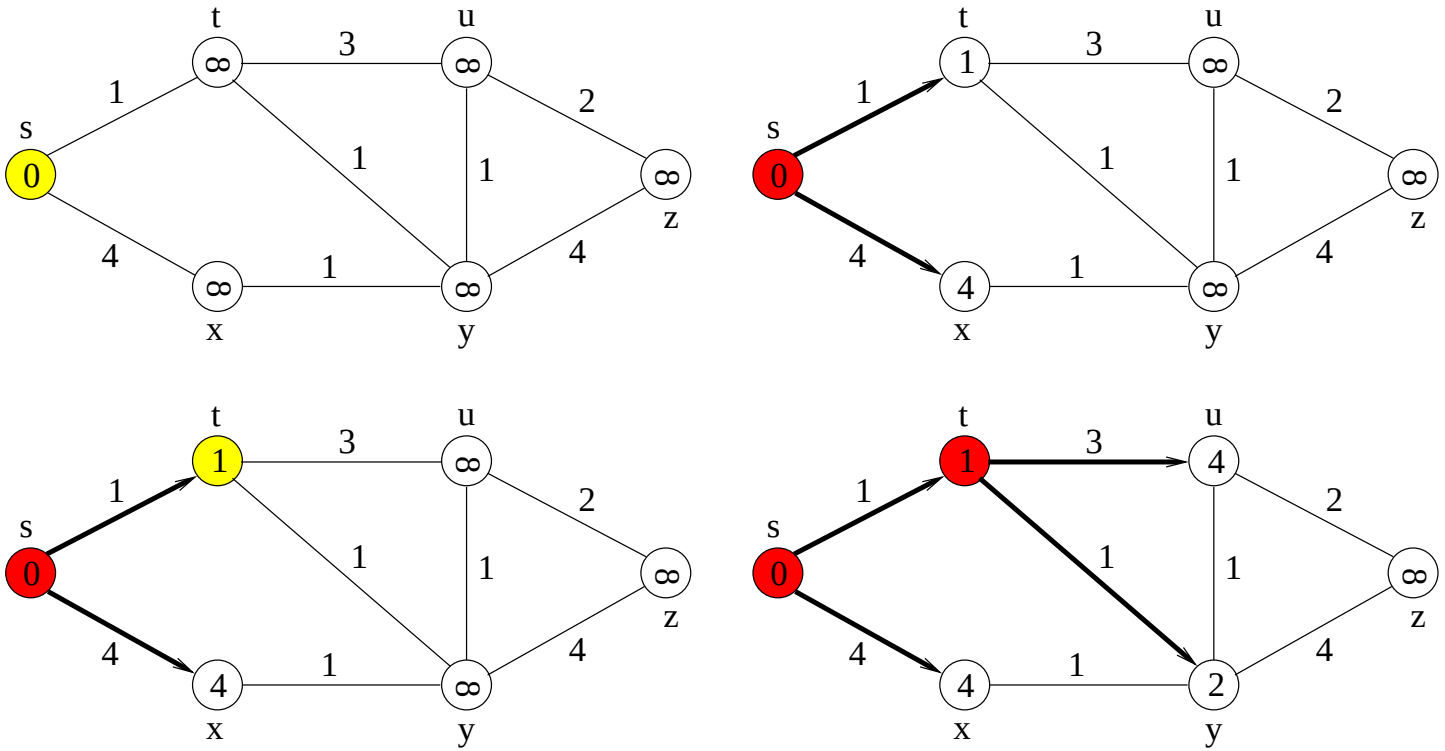
- pidetään kuitenkin S mukana sillä se selkeyttää pian esitettävää oikeellisuustodistusta

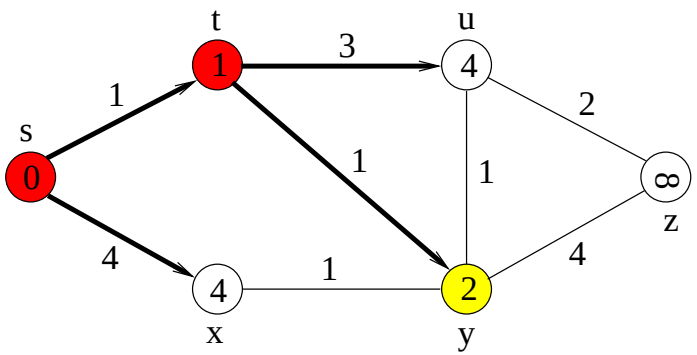
- algoritmin suorituksen jälkeen lyhin polku $s \rightsquigarrow v$ saadaan selville seuraavasti:
  - $path[v]$ kertoo minkä solmun kautta lyhin polku $s \rightsquigarrow v$ saapuu solmuun v
  - solmuun $path[v]$ lyhin polku saapuu solmun $path[path[v]]$ kautta, jne
  - laitetaan pinoon $path[v], path[path[v]], path[path[path[v]]]$ ja tulostetaan pinon sisältö
  - näin saadaan tulostettua polulla koko polku $s \rightsquigarrow v$ alusta loppuun

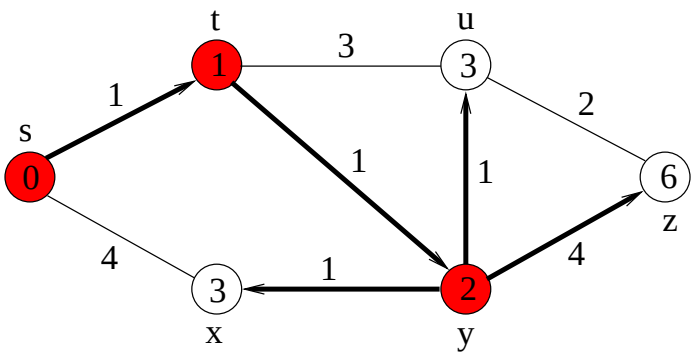
- algoritmina:

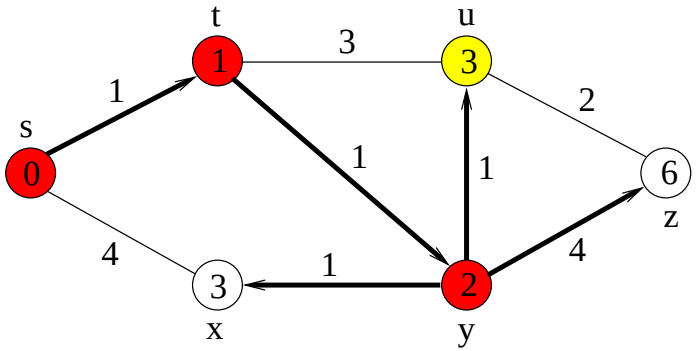
```
shortest-path(G,v)
1   u = path[v]
2   while u ≠ s
3       push(pino,u)
4       u = path[u]
5   print(lyhin polku solmusta s solmuun v )
6   while not empty(pino)
7       u = pop(P)
8       print(u)
```

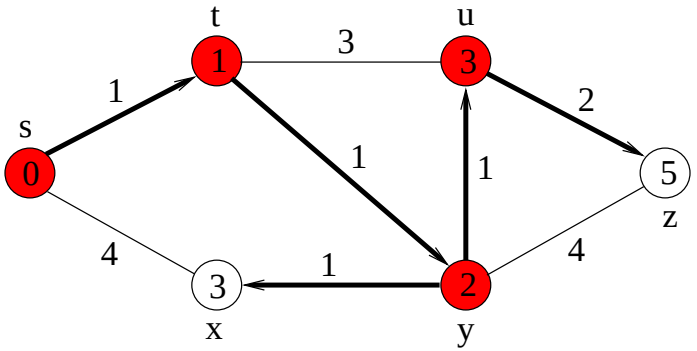
- toinen esimerkki Dijkstran algoritmin toiminnasta, tällä kertaa kyseessä suuntaamaton verkko

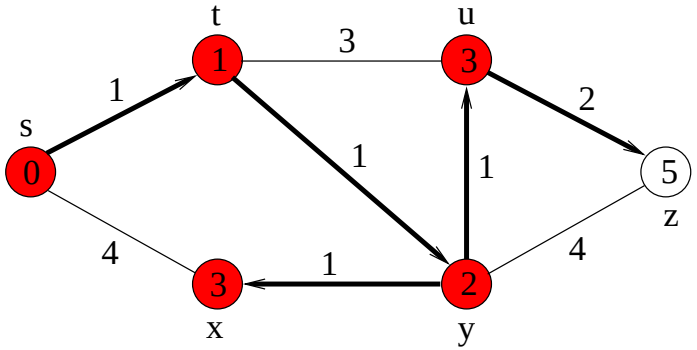


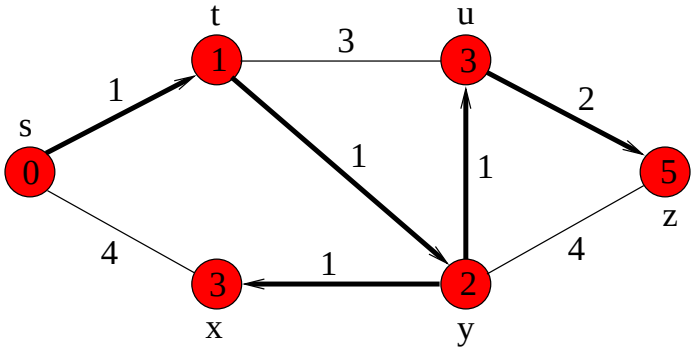

t
u
s
x
y
z
0
1
2
3
4
8
1
3
2
1
1
4
4
1


t
u
s
x
y
z
0
1
2
3
4
6
1
3
2
1
1
4
4
1


t
u
s
x
y
z
0
1
2
3
4
6
1
3
2
1
1
4
4
1


t
u
s
x
y
z
0
1
2
3
4
5
1
3
2
1
1
4
4
1


t
u
s
x
y
z
0
1
2
3
4
5
1
3
2
1
1
4
4
1


t
u
s
x
y
z
0
1
2
3
4
5
1
3
2
1
1
4
4
1

- Dijkstran-algoritmin vaativuus
  - rivien 1-5 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 7-8 tehdään $|V|$ kappaletta heap-insert-operaatioita, aikaa siis kuluu $\mathcal{O}(|V| \log |V|)$
  - rivien 8-15 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 rivillä 12 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
- suoritetaan algoritmia lähtösolmuna s siihen asti 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* (greedy) strategiaa:
  - rivillä 9 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

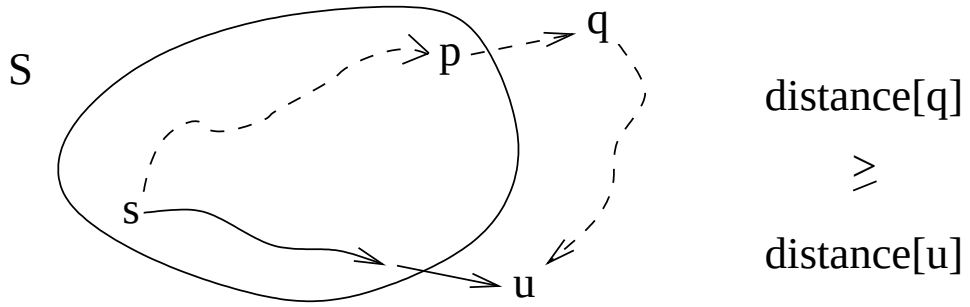
  (2) solmuille $v \in V \setminus S$ pätee: $distance[v]$ on lyhimmän tunnetun, eli käytännössä joukon S solmujen kautta kulkeva polun $s \rightsquigarrow v$ pituus

- 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 positiivinen
  - 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

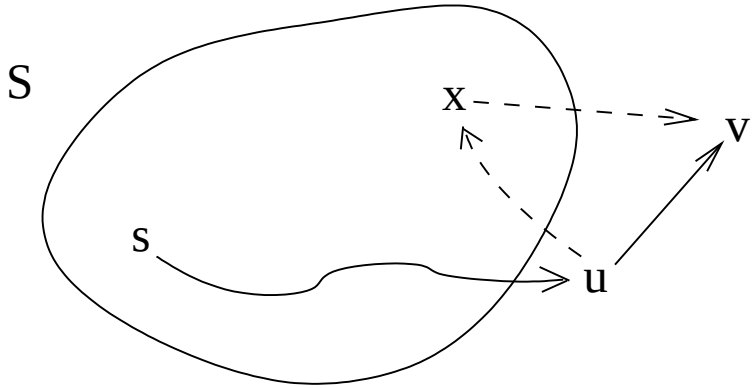
- On vielä osoitettava, että algoritmi päivittää oikein etäisyysarviot taulukkoon $distance$, eli että väittämä (2) säilyy voimassa uusia solmuja joukkoon S lisättäessä

- 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 rivillä 13 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

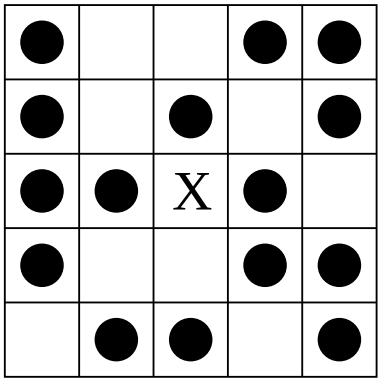
  (2) solmuille $v \in V \setminus S$ pätee: $distance[v]$ on lyhimmän tunnetun, eli käytännössä joukon S solmujen kautta kulkeva polun $s \rightsquigarrow v$ pituus

  ovat voimassa algoritmin suorituksen alussa ja pysyvät voimassa kun solmuja lisätään joukkoon S

- Lopussa kaikki solmut ovat joukossa S, eli väittämästä (1) seuraa, että löytää algoritmi 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
  - 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:

$$s \rightarrow \ldots \rightarrow path[path[t]] \rightarrow path[t] \rightarrow t$$

- 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

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 solmun välisen lyhimmän polun pituus/paino
- Jos kaarten painot ovat ei-negatiivisia, ongelma voidaan ratkaista suorittamalla Dijkstran algoritmi n kertaa
- Olettamalla prioriteettijono toteutetuksi kekona saadaan aikavaativuudeksi $O(n \cdot (n + m) \log n) = O((n^2 + nm) \log n)$, missä $m = |E|$
- Jos verkko on tiheä (kaaria on melkein jokaisen solmun välillä) eli $m = \mathcal{O}(n^2)$, tämä on $O(n^3 \log n)$
- Floydin-Warshallin algoritmi ratkaisee ongelman ajassa $O(n^3)$, mikä siis tiheillä verkoilla on asymptoottisesti parempi kuin Dijkstran toistaminen
- Lisäksi Floyd-Warshall
  - sallii negatiiviset painot (mutta ei negatiivisia kehiä)
  - on helppo toteuttaa, 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 $i \rightarrow j$ paino tai polun $i \rightarrow 1 \rightarrow j$ 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 solmu 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 $D^2[i,j] = min\{D^1[i,j], D^1[i,2] + D^1[2,j]\}$

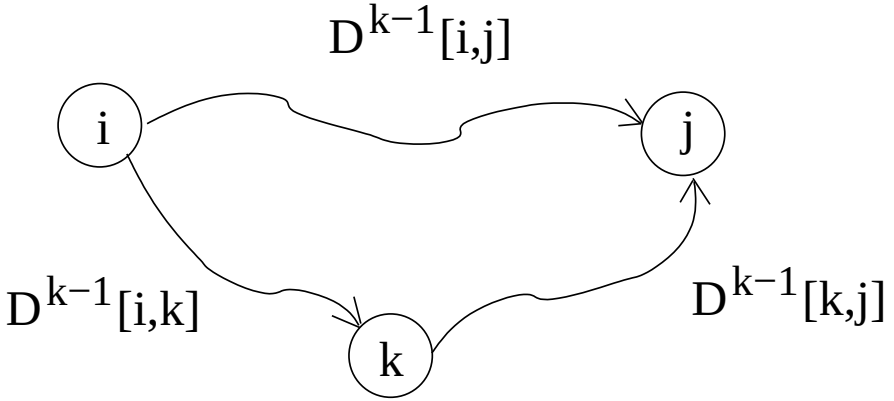
- $D^2[i,j]$ siis pitää sisällään lyhimmän polun i ja j välillä, joka voi olla joko
  - $i \to j$
  - $i \to 1 \to j$
  - $i \to 2 \to j$
  - $i \to 1 \to 2 \to j$
  - $i \to 2 \to 1 \to j$
- huomattavaa on, että polku voi kulkea solmun 2 kautta kolmella eri tavalla
  - joko käymättä solmussa 1 tai kulkemalla ensin tai lopuksi solmun 1 kautta
  - tieto lyhimmästä korkeintaan solmun 1 kautta kulkevasta polun $i \leadsto 2$ pituudesta on huomioitu laskettaessa $D^1[i,2]$, polku on joko $i \to 2$ tai $i \to 1 \to 2$
  - vastaavasti lyhimmmän korkeintaan solmun 1 kautta kulkevan polun $2 \leadsto j$ 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

- näinollen on helppo muodostaa algoritmi joka selvittää matriisit numerojärjestyksessä päätyen matriisiin D^n joka on haluttu lopputulos

- Tästä saamme algoritmin ensimmäisen version

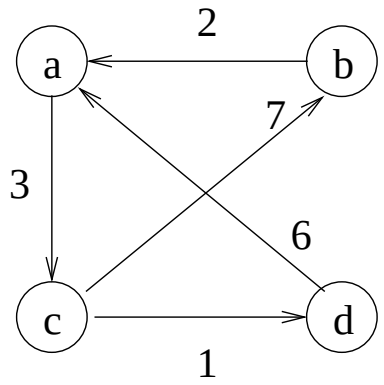
```
Floyd-Warshall-v1(A)
1   for i = 1 to n
2       for j = 1 to n
3           if i == j
4               D^0[i,j] = 0
5           else D^0[i,j] = A[i,j]

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 kalvoilla 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

$D^3 =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

$D^4 =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

- 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:

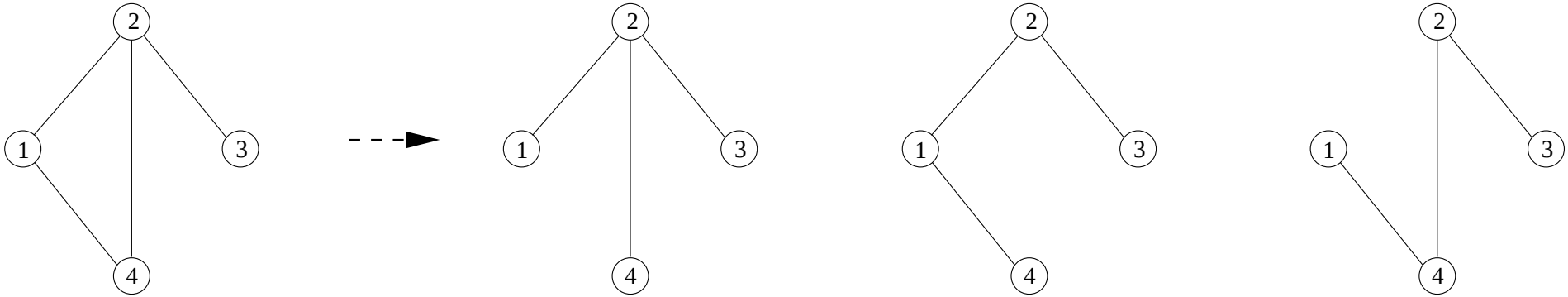
```
Floyd-Warshall-v2(A)
1   for i = 1 to n
2       for j = 1 to n
3           if i == j
4               D[i,j] = 0
5           else D[i,j] = A[i,j]

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

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
- verkko ja sen virittäviä puita

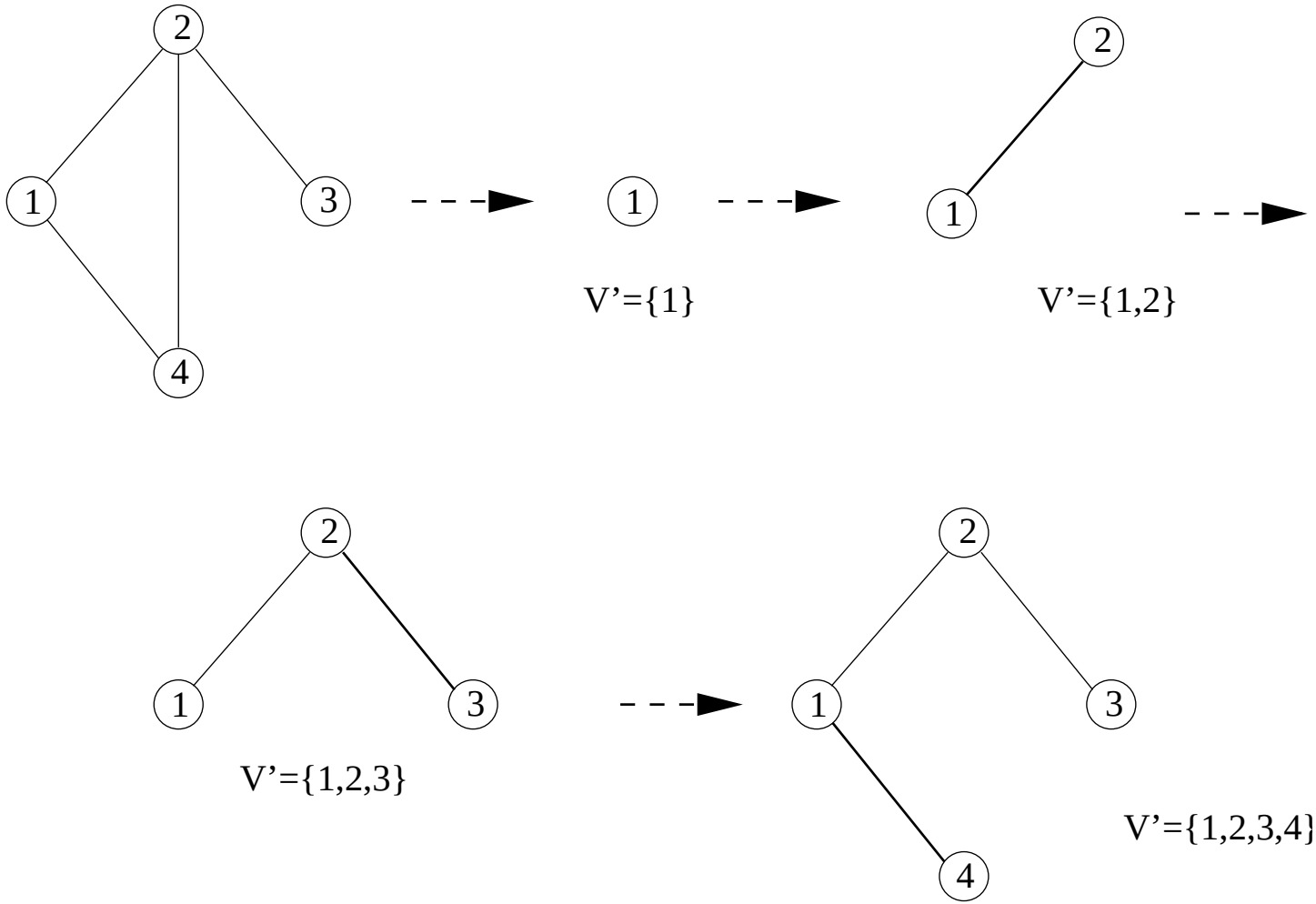


- seuraavassa yksinkertainen algoritmi joka muodostaa verkolle $G = (V, E)$ jonkin virittävän puun

- algoritmin lopussa joukossa T olevat kaaret muodostavat virittävän puun

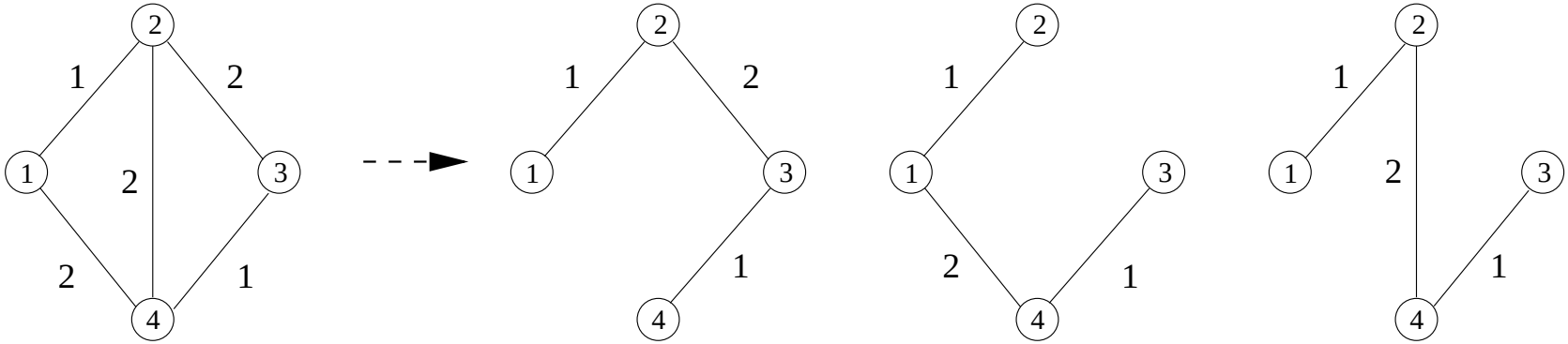
```
spanning-tree(G)
1   valitaan mielivaltainen solmu v ∈ V
2   V'= {v}
3   T = ∅
4   while V' ≠ V
5       valitse kaari (u,v) ∈ E siten että u ∈ V' ja v ∈ V \ V'
6       V'= V' ∪ {v}
7       T = T ∪ {(u,v)}
```

- 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 minimaalisen virittävän puun muodostamisesta
- Olkoon $G = (V, E)$ suuntaamaton yhtenäinen painotettu verkko, jonka kaaripainot määrää funktio w
- verkon G *minimaalinen virittävä puu* (engl. minimal spanning tree) on G:n virittävistä puista se jonka kaaripainojen summa on pienin

- Yhdellä verkolla voi olla useita minimaalisia virittäviä puita:



- Yleensä riittää että pystytään muodostamaan jokin minimaalisista virittävistä puista
- Minimaalisen 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 minimaalinen

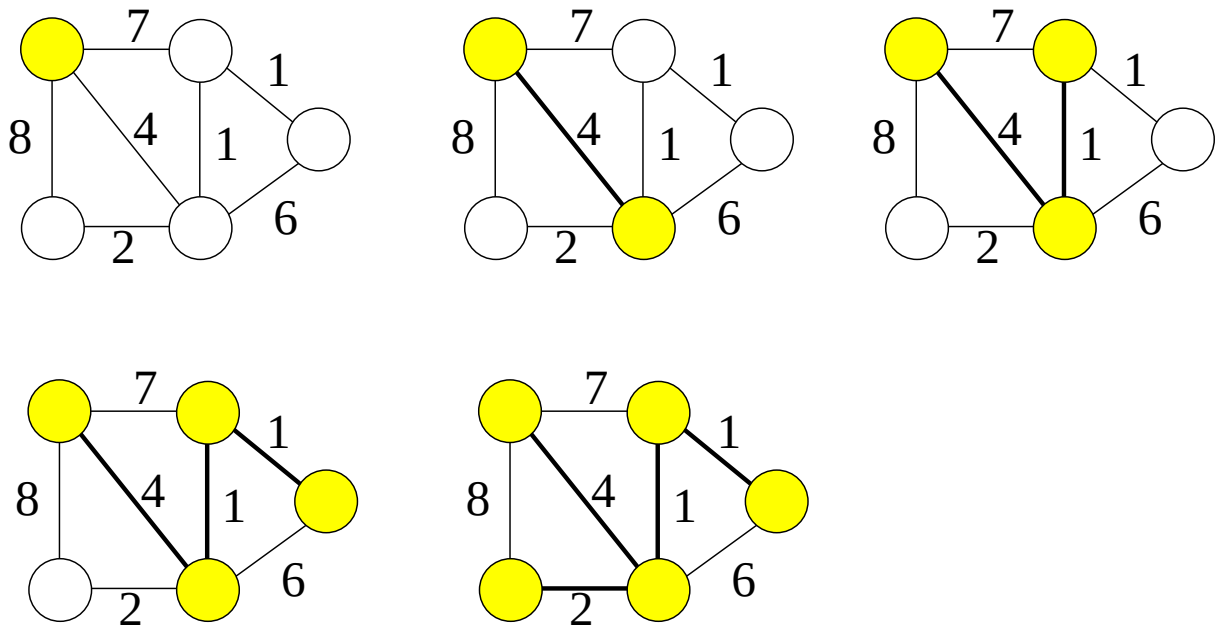
Primin algoritmi

- Algoritmin toimintaperiaate on hyvin samankaltainen kuin kalvon 477 algoritmilla
  - 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 kalvolla algoritmin hahmotelma

- Puun muodostaminen aloitetaan jostain verkon solmusta r, aloitussolmu voidaan valita vapaasti
- Algoritmi kerää minimaalisen 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 kierroksellä 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, minimaalinen virittävä puu on valmis ja se koostuu joukon T kaarista

- Ennen yksityiskohtaisempaa algoritmiesitystä, tarkastellaan esimerkkiä

- Algoritmi aloittaa vasemman yläreunan solmusta, tummennetut kaaret muodostavat minimaalisen 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

- Lähdetään tarkastelemaan algoritmin toteutuksen yksityiskohtia
- Algoritmi käyttää aputaulukkoa $distance$, johon on talletettuna jokaiselle solmulle sen lyhin etäisyys johonkin jo virittävävässä puussa olevaan, eli joukon H ulkopuolella olevaan solmuun
- 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 solmun $distance$-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 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 ∈ Adj[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
```

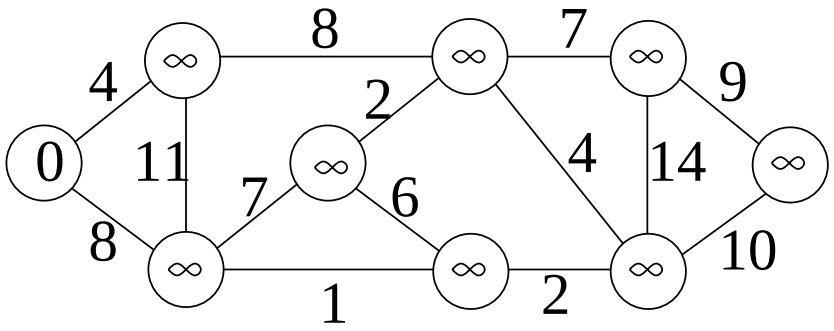
- vaikka algoritmin toimintaperiaate onkin hyvin selkeä, mutkistavat toteutusyksityiskohdat algoritmia jossain määrin

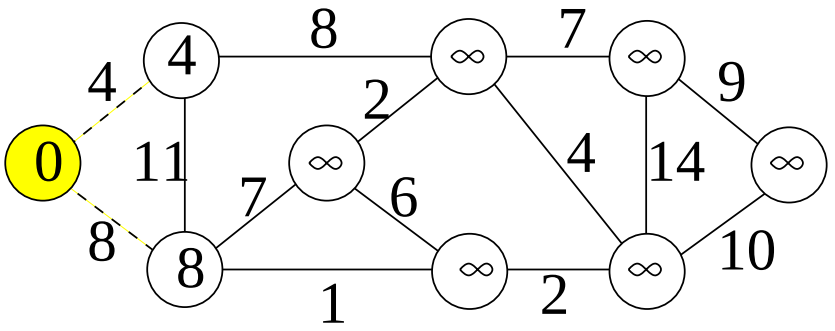
- algoritmin toimintaidea vielä kerran:
  - aluksi virittävä puu on tyhjä ja asetetaan kaikille solmuille paitsi r:lle etäisyyden $distance$ arvoksi ääretön (rivi 3)
  - myös lyhin kaari, joka yhdistää solmun virittävään puuhun on tässä vaiheessa tuntematon (rivi 4) kaikille solmuille
  - riveillä 6-7 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, minkä on kevyin niiden kaarten joukossa jotka yhdistävät v:n konstruktion alla olevaan virittävään puuhun

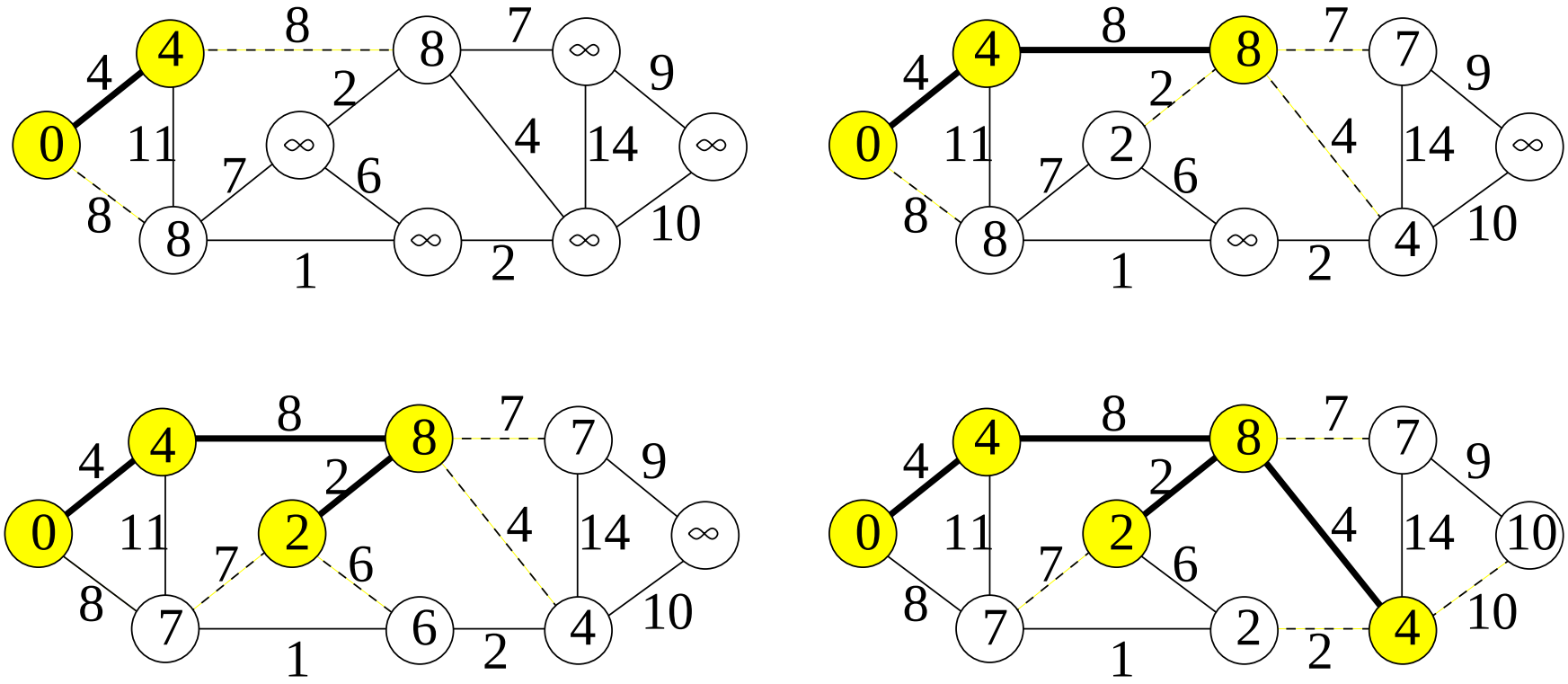
    kyseessä oleva kaari on $(parent[v], v)$
  - tämän jälkeen otetaan keosta käsittelyyn solmu jonka etäisyys virittävään puuhun on pienin ja liitetään solmu virittävään puuhun
  - riveillä 12-16 huolehditaan edelleen, että yllä oleva ominaisuus virittävän puun ulkopuolisten solmujen $distance$- ja $parent$-arvoille pysyy voimassa
  - jatketaan niin kauan kun solmuja on keossa jäljellä

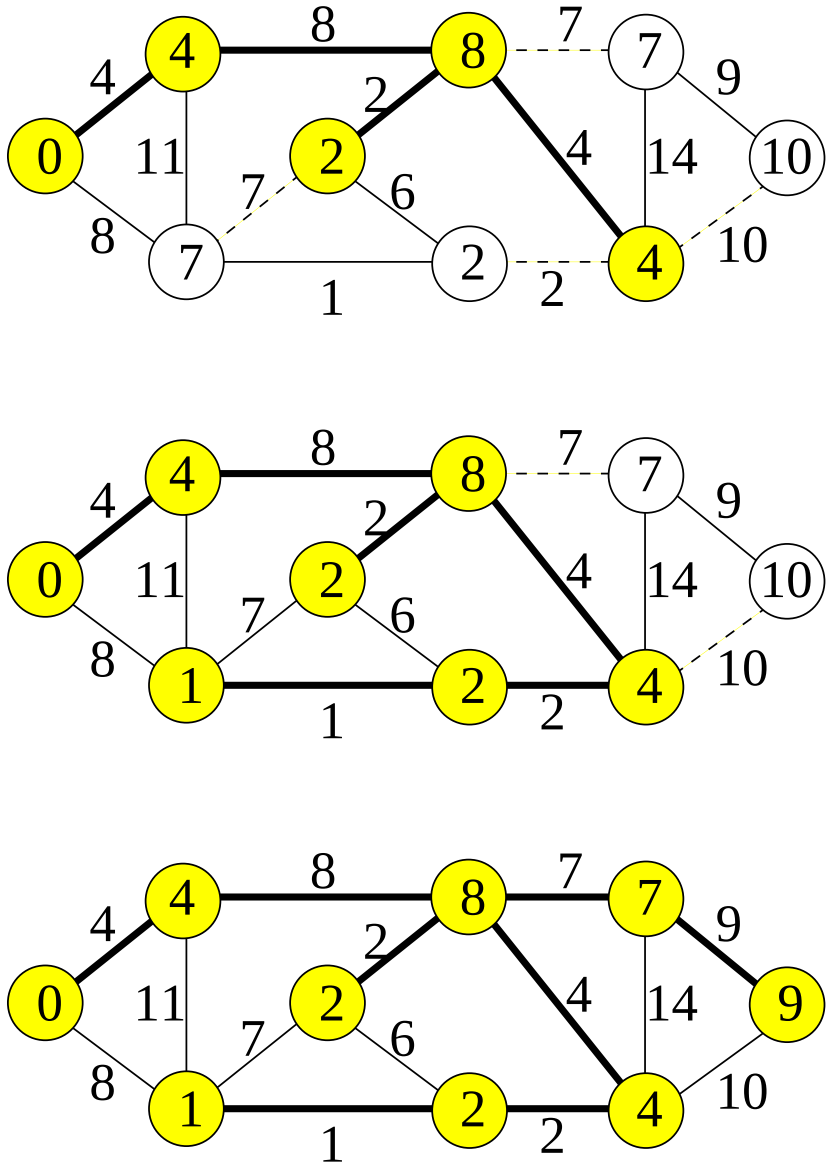
- seuraavassa esimerkki algoritmin toiminnasta
- taulukoiden $distance$ ja $parent$ informaatio on merkitty suoraan solmujen yhteyteen, myöskään keon H sisältöä ei eksplisiittisesti näytetä, kekoon kuuluvat kaikki värjäämättömät solmut
- rivien 1-7 alustusvaiheen jälkeinen tilanne vasemmalla, eli kaikille paitsi aloitussolmulle on merkitty arvoksi $distance$ ääretön
- ensimmäisenä keosta poistetaan aloitussolmu, joka on värjätty oikean puoleisessa kuvassa, jossa 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





8 7 4 8 7 9 4 2 0 11 2 4 14 10 7 6 8 7 2 4 10 1 2
8 7 4 8 7 9 4 2 0 11 2 4 14 10 7 6 8 1 2 4 10 1 2
8 7 4 8 7 9 4 2 0 11 2 4 14 10 7 6 8 1 2 4 10 1 2
8 7 4 8 7 9 4 2 0 11 2 4 14 9 7 6 8 1 2 4 10 1 2
8 7 4 8 7 9 4 2 0 11 2 4 14 9 7 6 8 1 2 4 10 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 $\mathcal{O}(|V| \log |V|)$
  - 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 $\mathcal{O}(|V|)$

Primin algoritmien oikeellisuus

- Primin algoritmi noudattaa ahnetta strategiaa: lisätään aina virittävään puuhun lähimpänä oleva puun ulkopuolinen solmu
- ei ole aivan itsestään selvää että algoritmi tuottaa nimenomaan minimaalisen virittävän puun
- Todistetaan, että Primin algoritmi todella tuottaa minimaalisen 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 tuottaa minimaalisen virittävän puun
  - Myös Primin jälkeen esitettävän Kruskalin algoritmin oikeellisuus perustuu samaan aputulokseen

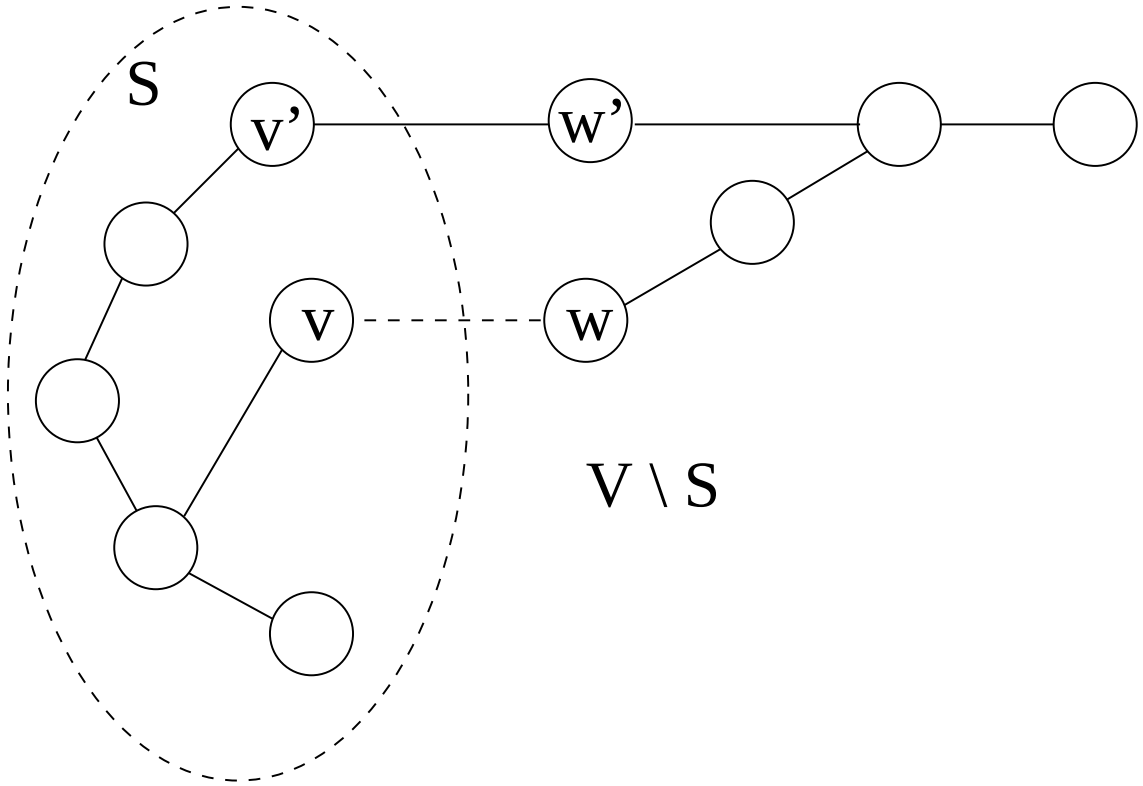
- **Lause:** *Oletetaan suuntaamattomasta yhtenäisestä verkosta $G = (V, E)$, että sen*
  1. *kaikki kaaripainot ovat keskenään erisuuria*
  2. *solmut on jaettu kahteen osajoukkoon S ja $V \setminus S$*
  3. *kaari $e = (v, w)$ kulkee osajoukosta toiseen eli $v \in S$ ja $w \in V \setminus S$*
  4. *kaari e on painoltaan pienin kaikista tällaisista osajoukkojen S ja $V \setminus S$ välisistä kaarista*

  Jos T on verkon G minimaalinen virittävä puu niin siihen kuuluu kaari e

- Lauseen todistus tehdään kontrapositiotekniikalla (ks. viikon 1 laskareiden tehtävä 1), eli osoitetaan $A \Rightarrow B$ näyttämällä, että $\neg B \Rightarrow \neg A$ on tosi
  - Tässä A tarkoittaa T *on minimaalinen virittävä puu*, ja
  - B tarkoittaa, että *puu T sisältää oletusten mukaisen kaaren $e = (v, w)$*

- Olkoon T' mikä tahansa verkon G virittävä puu, joka *ei sisällä* kaarta $e = (v, w)$
- Osoitetaan, että tällöin verkolla G on olemassa toinen virittävä puu T, jonka kaaripainojen summa on pienempi kuin T':llä. Eli T' ei voi olla minimaalinen virittävä puu
  - Puussa T' on polku v:stä w:hen (virittävä puuhan on yhtenäinen ja sisältää kaikki verkon solmut)
  - Lähdetään kulkemaan tätä polkua solmusta $v \in S$ alkaen
  - Jossain vaiheessa kohdataan polulla ensimmäinen kaari (v', w'), joka kulkee joukosta S joukkoon $V \setminus S$ (ks. kuva seuraavalla kalvolla)
  - Vaihdetaan virittävän puun T' kaari (v', w') kaareksi (v, w), jolloin saadaan uusi virittävä puu T
  - Koska (v, w) on lähtöoletuksen mukaan pienin joukkoja S ja $V \setminus S$ yhdistävistä kaarista, on puun T kaaripainojen summa pienempi kuin puulla T'
  - Siis T' ei voi olla verkon G minimaalinen virittävä puu
- On siis osoitettu, että jos oletusten mukainen kaari $e = (v, w)$ ei kuulu virittävään puuhun T', ei virittävä puu ole minimaalinen
- Tämän kanssa on yhtäpitävää, että jos T on verkon minimaalinen virittävä puu, kuuluu siihen kaari $e = (v, w)$

- Kuvassa virittävä puu T' johon ei kuulu kaarta (v, w)



- korvaamalla kaari (v', w') kaarella (v, w), on tuloksena myös virittävä puu T
- oletuksen mukaan kaari (v, w) on kaarta (v', w') kevyempi, joten T' ei voi olla minimaalinen virittävä puu

- Primin algoritmi on helppo osoittaa oikeaksi edelliseen lauseeseen vedoten
- Tarkastellaan sitä hetkeä, kun algoritmi päättää lisätä kaaren $e = (v, w)$ 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 lauseen perusteella kuulua 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
- Lauseen oletus siitä, että verkon kaikki kaaripainot ovat erisuuria, ei ole välttämätön. Samantapaiset johtopäätökset voitaisiin tehdä ilmankin oletusta, mutta monimutkaisemmin

Kruskalin algoritmi

- Algoritmin suorituksen aikana pidetään yllä useita erillisiä paloja (puita), joita vähitellen yhdistellään suuremmiksi paloiksi
- Kun jäljellä on vain yksi pala, on se verkon minimaalinen virittävä puu
- 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
- 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 $pala$

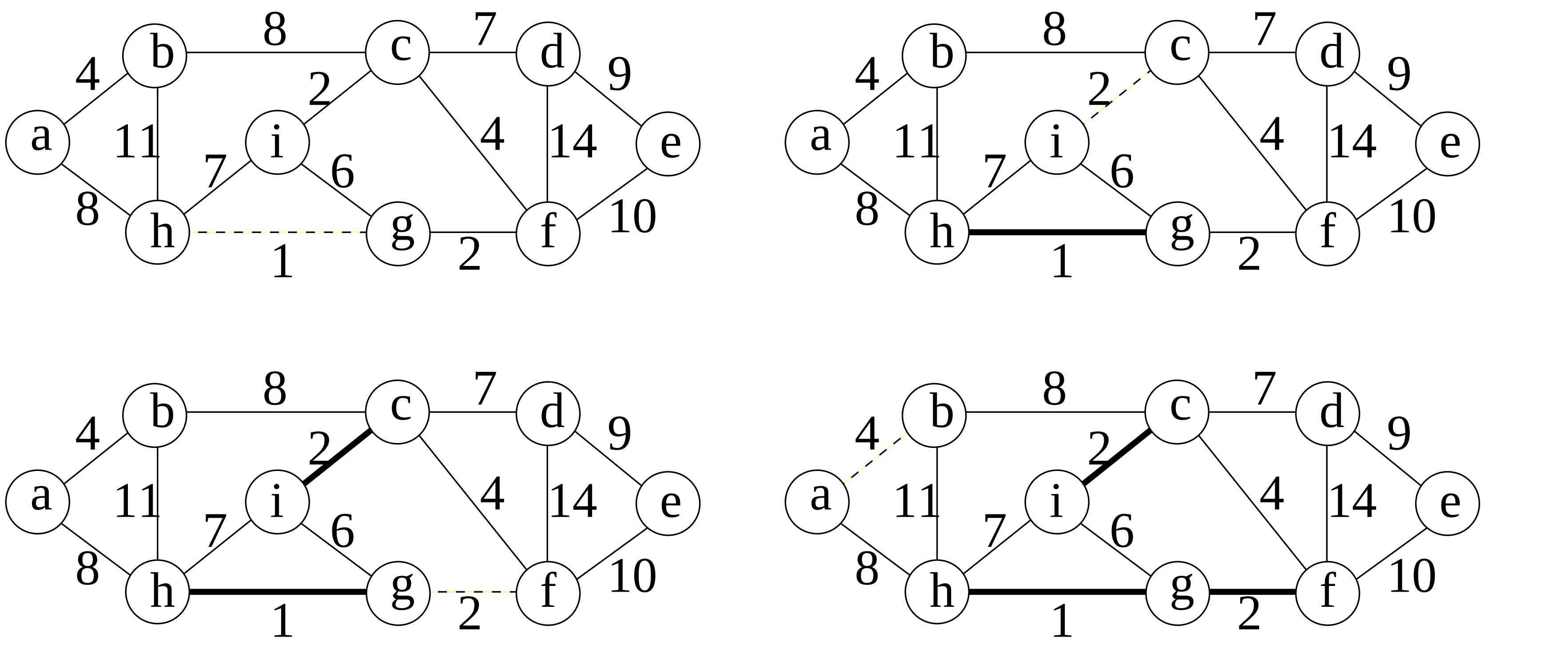
- algoritmi kerää virittävän puun kaaria joukkoon T ja palauttaa lopuksi joukon

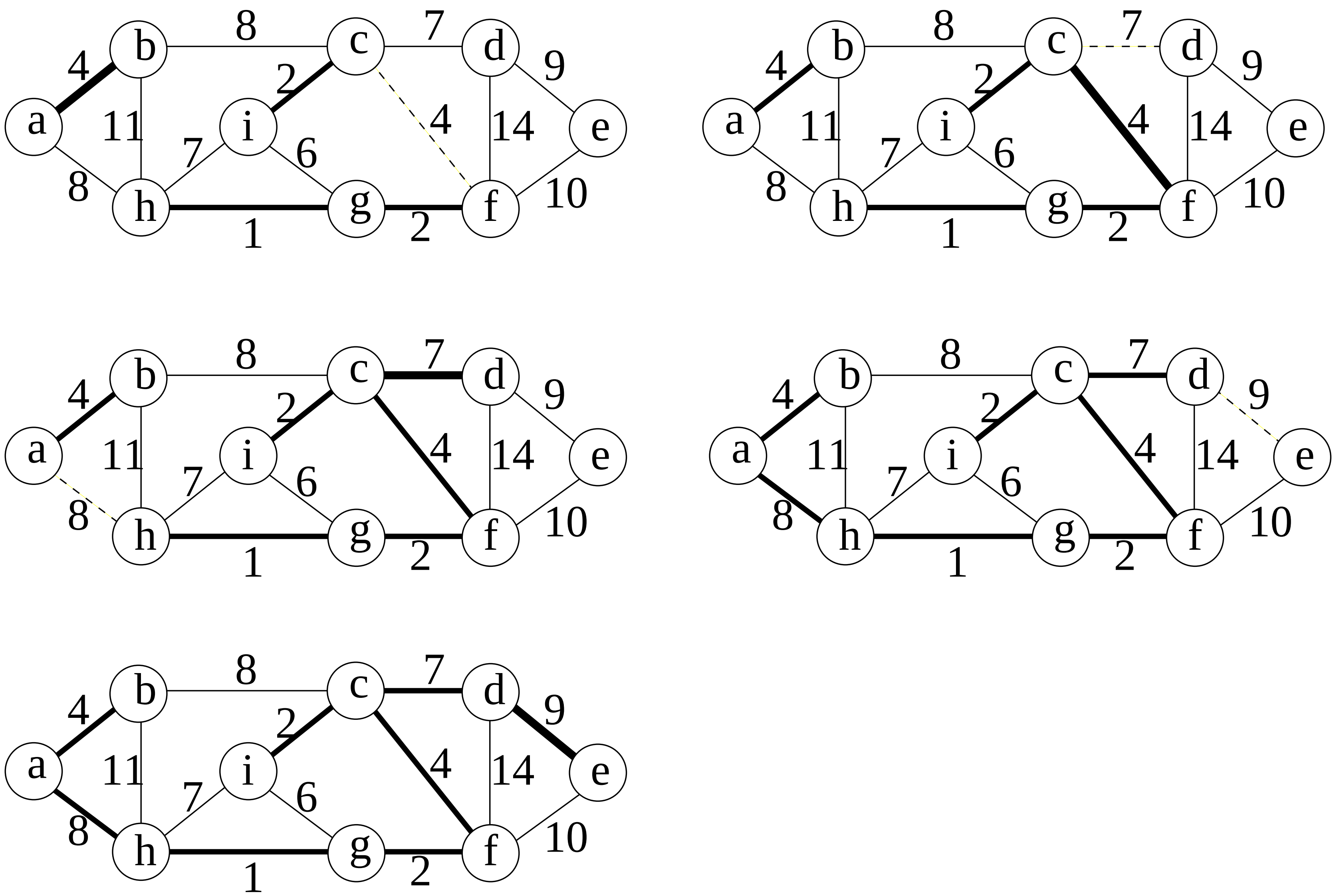
- algoritmin lopussa verkon minimaalinen virittävä puu siis koostuu joukon T kaarista

```
Kruskal(G,w)
1   T = ∅
2   for kaikille solmuille v_i ∈ V
3       solmu v_i muodostaa oman palan pala[v_i]
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 kalvolla:

- 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
7
9
11
2
4
14
7
6
8
10
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 $pala[v]$ 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
  - while-silmukan kokonaissuoritusaika on siis $O(|V|^2 + |E|) = O(|V|^2)$
  - 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 minimaalisen virittävän puun
- Primin algoritmin tapaan Kruskalin algoritmin oikeellisuus perustuu suoraan kalvon 493 apulauseeseen
- Tarkastellaan sitä hetkeä, kun algoritmi päättää lisätä kaaren $e = (v, w)$ tulosjoukkoonsa T
- Valitaan osajoukoksi S ne solmut jotka kuuluvat samaan palaan kuin solmu v
- koska $pala[v] \neq pala[w]$, niin $w \notin S$
- Apulauseen perusteella e kuuluu jokaiseen verkon G minimaaliseen virittävään puuhun
- Algoritmin päättyessä T on puu ja siihen on lisätty vain sellaisia kaaria, jotka lauseen perusteella kuuluvat minimaaliseen virittävään puuhun

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))

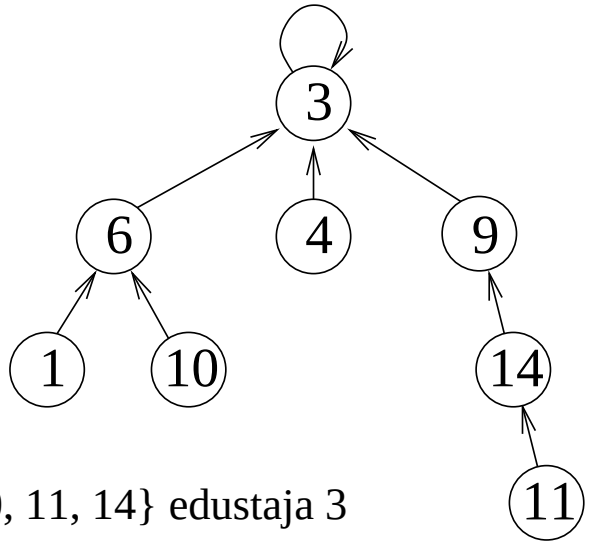
- Kruskalin algoritmi Union-find-rakenteen avulla

```
Kruskal(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 A
```

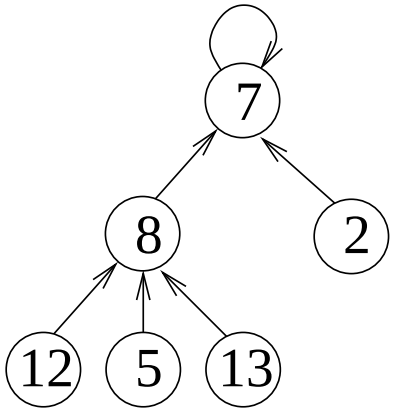
- Algoritmin tehokkuus perustuu siihen, että Union-find-rakenne voidaan toteuttaa niin, että jono peräkkäisiä union- ja find-operaatiota voidaan suorittaa selvästi nopeammin kuin vastaavat operaatiot jos palat toteutettaisiin taulukon avulla

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.

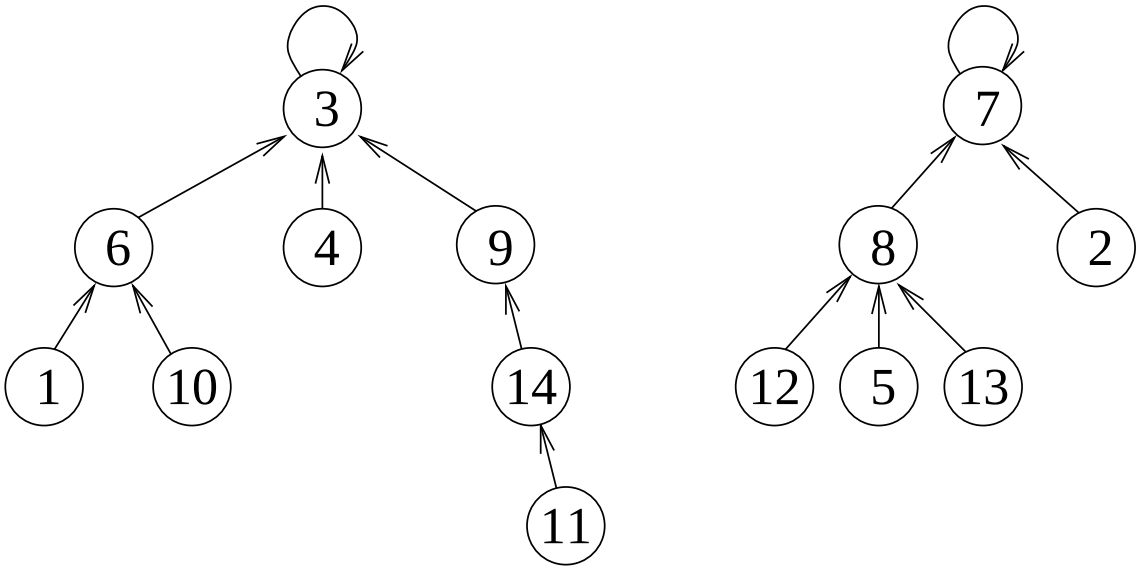


joukko {1, 3, 4, 6, 9, 10, 11, 14} edustaja 3



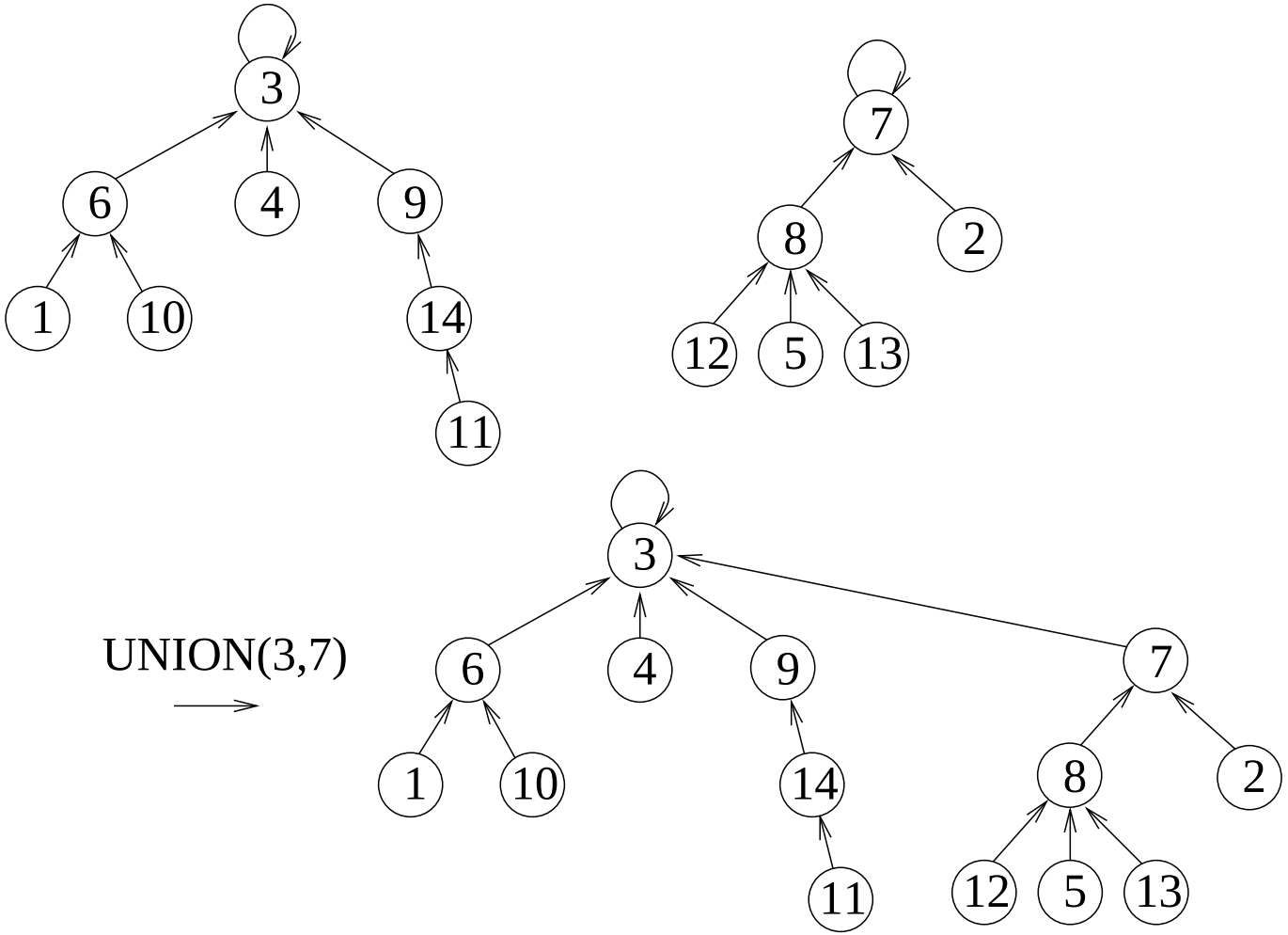
joukko {2, 5, 7, 8, 12, 13} edustaja 7

- 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

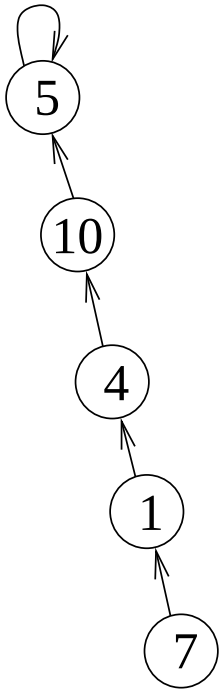


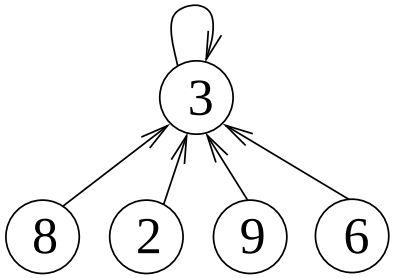
1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	7	3	3	8	3	7	7	3	6	14	8	8	9

- 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 $O(h)$, 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 esittäien 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
- Jotta union-operaatiota tehdessä voitaisiin päätellä, kumpi puu on korkeampi, on jokaisen puun juurisolmulla r attribuutti $r.rank$ joka on sen puun korkeus, jonka juurisolmu on r
- Jos yhdistettävät puut ovat yhtä korkeita, yhdistetyn puun korkeus on yhtä suurempi kuin alkuperäisten puiden

- Operaatioden koodi: make-set

```
make-set(G,x)
1   x.p = x
2   x.rank = 0
```

- ja union

```
union(x,y)
1   if x.rank < y.rank
2       x.p = y
3   elsif x.rank > y.rank
4       y.p = x
5   else
6       x.p = y
7       y.rank = x.rank + 1
```

Union-find-rakenteen avulla toteutetun Kruskalin algoritmin aikavaativuus

- Induktiolla voidaan todistaa, että jos joukkoa kuvaavassa puussa on $|V|$ solmua ja union-operaatiot on toteutettu edellisellä kalvolla kuvatulla tavalla, niin puun korkeus on $O(\log |V|)$

- Tällöin Kruskalin algoritmissa tarvittavat $O(|E|)$ find-operaatiota voidaan suorittaa ajassa $O(|E| \log |V|)$

- union- ja make-set-operaatiot voidaan suorittaa vakioajassa. Kruskalin algoritmissa molempia tarvitaan $O(|V|)$-kappaletta

- Edellisillä kalvoilla kuvatulla tavalla toteutettua Union-find-rakennetta käyttävän Kruskalin algoritmin aikavaativuus on kaarten järjestämiseen käytettävä $O(|E| \log |E|)$ + rivin 5-9 toistolauseeseen kuluva $O(|E| \log |V|)$ eli yhteensä $O(|E|(\log |E| + \log |V|))$

- Koska yhtenäisessä verkossa solmuja ei voi olla kuin korkeintaan yksi enemmän kuin kaaria, sievenee aikavaativuus muotoon $O(|E| \log |E|)$

Kruskal vai Prim

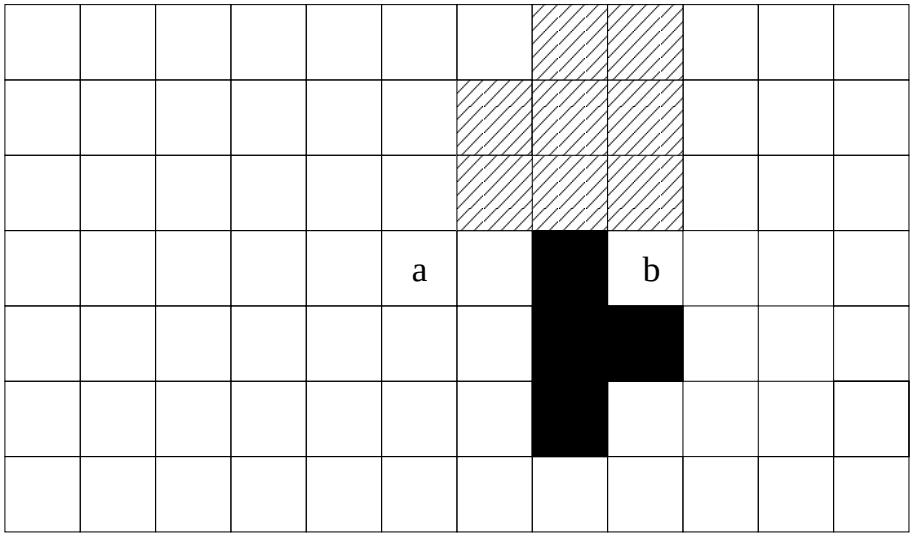
- Primin algoritmin aikavaativuudeksi on todettu $O(|E| \log |V|)$, Kruskalin $O(|E| \log |E|)$
- Näiden suuruusluokka on sama, sillä koska $|E| \leq |V|^2$, on:

$$|E| \log |E| \leq |E| \log(|V|^2) = 2 \cdot |E| \log |V| = O(|E| \log |V|)$$

- Käytännössä Prim on yleensä parempi:
  - Avaimen arvon pienennystä tarvitaan yleensä vain pienelle osalle kaarista
  - Kruskalin algoritmissa järjestämisen keskimääräinen vaatimus ei ole pahinta tapausta parempi
- Kruskal tulee kuitenkin Primiä nopeammaksi, jos
  - kaaret saadaankin valmiiksi järjestyksessä tai
  - kaaret voidaan järjestää yleistä alarajaa nopeammin

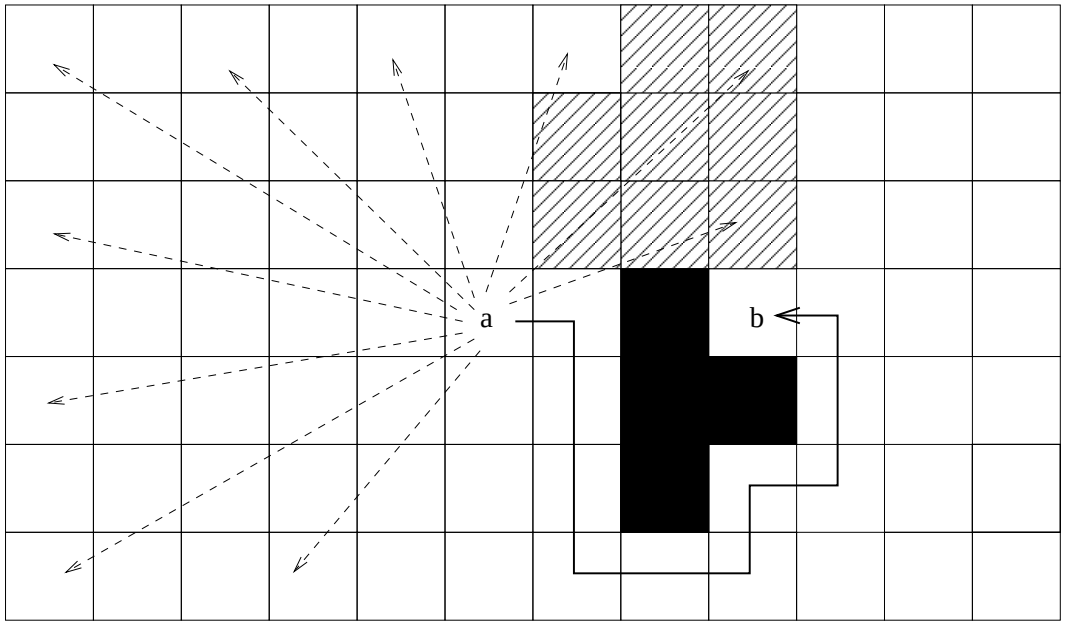
Lyhin kahden solmun välinen polku

- 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. kalvot 444-461)
- Lyhin polku, jonka pituus on 11 on merkitty kuvaan



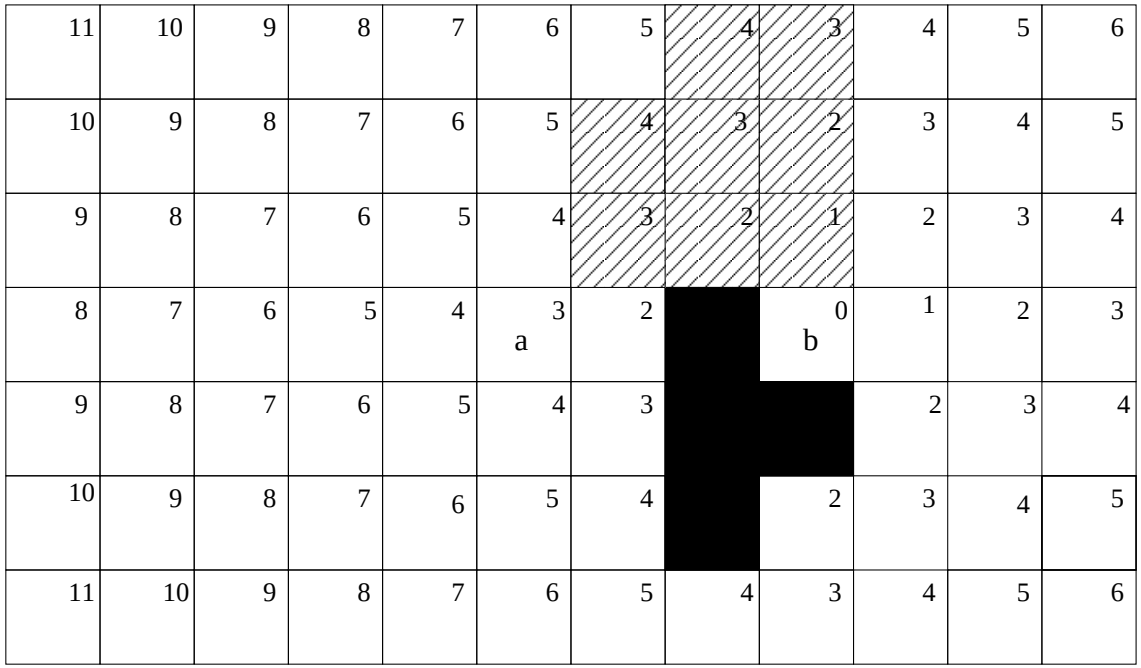
- Dijkstran algoritmin huono puoli on, että se ei huomioi millään tavalla, että kohdesolmu b on lähtöpaikan oikealla puolella
- Dijkstra etenee tasaisesti kaikkiin suuntiin ja tutkii samalla myös lyhimmät polut muihinkin solmuihin, myös täysin väärässä suunnassa oleviin
- Lyhimmän polun $a \rightsquigarrow b$ löytymisen kannalta Dijkstra siis tekee yleensä turhaa työtä

- Dijkstran algoritmi siis toimii seuraavasti:
  - ylläpidetään joukkoa S joka kohdistuu 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, tutkii Dijkstra siis 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 kalvolla 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



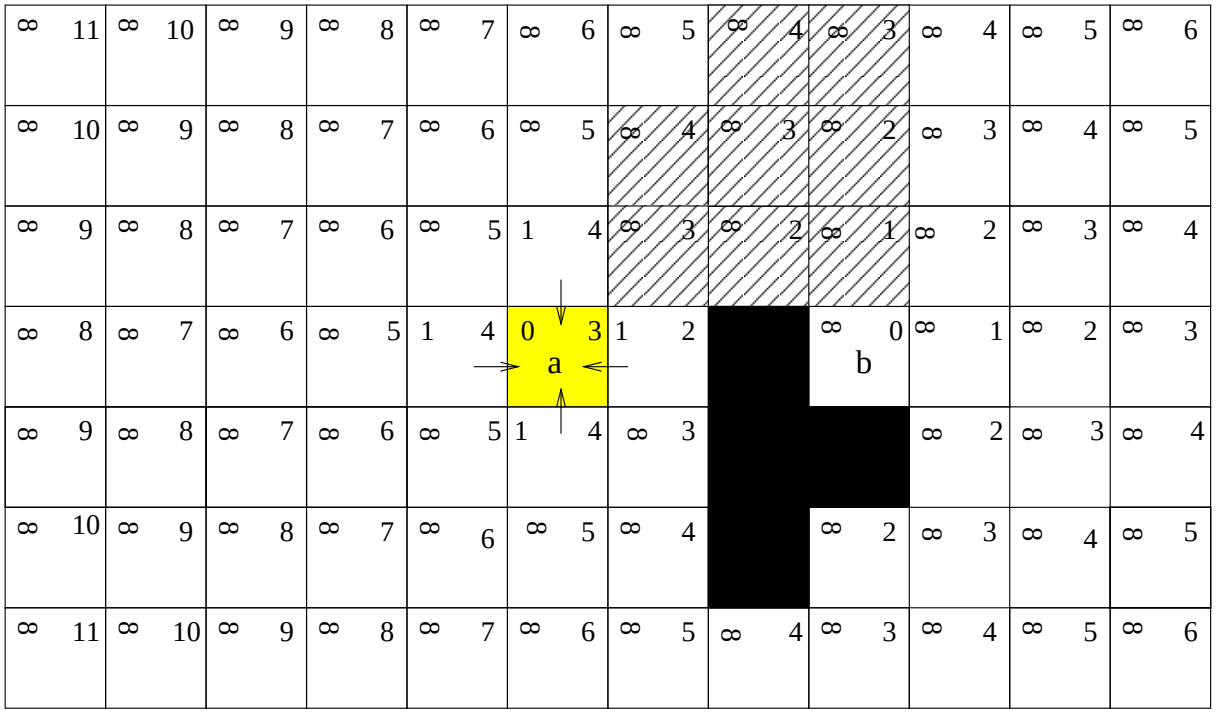
- todellisuudessa arvioita ei tarvitse laskea kaikille solmuille etukäteen vaan riittää, että ne selvitetään tarpeen vaatiessa

- 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 (sama kuin distance 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 (samoin kuin 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

∞ 11	∞ 10	∞ 9	∞ 8	∞ 7	∞ 6	∞ 5	∞ 4	∞ 3	∞ 4	∞ 5	∞ 6
∞ 10	∞ 9	∞ 8	∞ 7	∞ 6	∞ 5	∞ 4	∞ 3	∞ 2	∞ 3	∞ 4	∞ 5
∞ 9	∞ 8	∞ 7	∞ 6	∞ 5	∞ 4	∞ 3	∞ 2	∞ 1	∞ 2	∞ 3	∞ 4
∞ 8	∞ 7	∞ 6	∞ 5	∞ 4	0 3 a	∞ 2		∞ 0 b	∞ 1	∞ 2	∞ 3
∞ 9	∞ 8	∞ 7	∞ 6	∞ 5	∞ 4	∞ 3			∞ 2	∞ 3	∞ 4
∞ 10	∞ 9	∞ 8	∞ 7	∞ 6	∞ 5	∞ 4		∞ 2	∞ 3	∞ 4	∞ 5
∞ 11	∞ 10	∞ 9	∞ 8	∞ 7	∞ 6	∞ 5	∞ 4	∞ 3	∞ 4	∞ 5	∞ 6

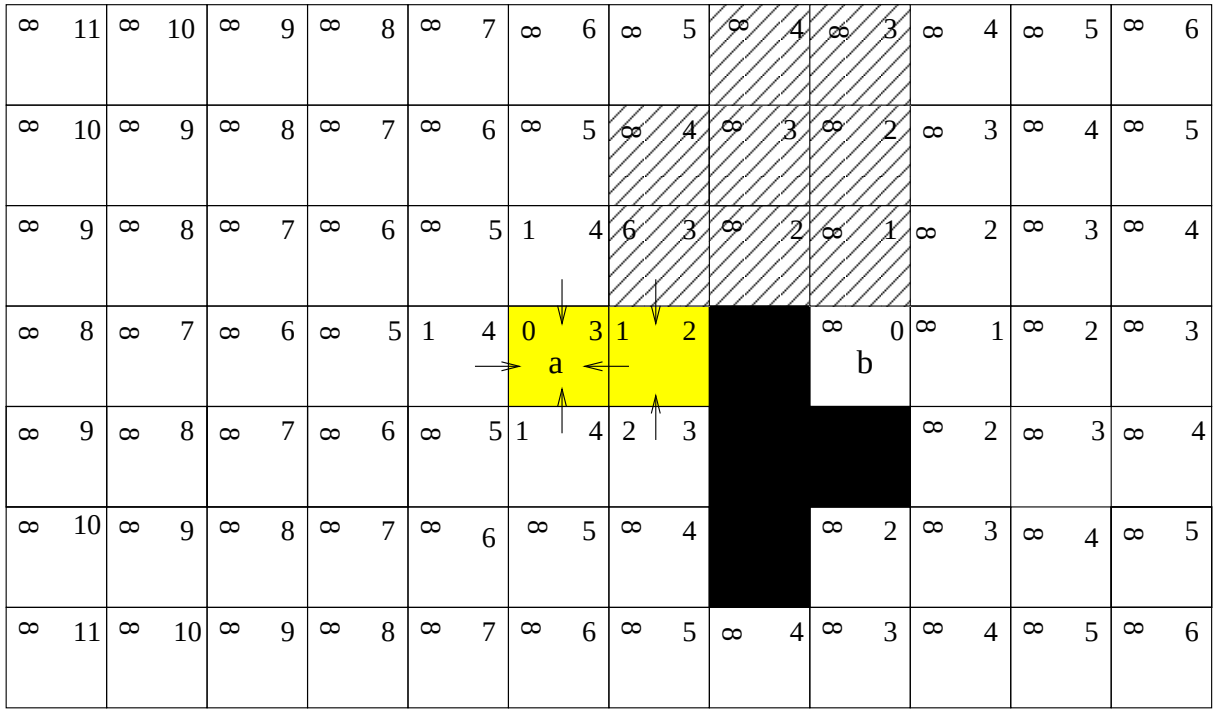
- lähtösolmu on nyt käsitelty eli viety joukkoon S, kuvasssa se on muutettu harmaaksi
- lähtösolmun vierussolmujen v arvot $alkuun[v]$ on päivitetty
- vierussolmuihin on myös merkitty että niihin saavuttiin alkusolmusta, tätä tietoa hyväksikäyttäen pystytään lopuksi generoimaan etsitty lyhin polku



- jälleen käsittelyyn valitaan solmu v jolle summa $alkuun(v) + loppuun(v)$ on pienin, eli lähtösolmun oikealla puolella oleva solmu

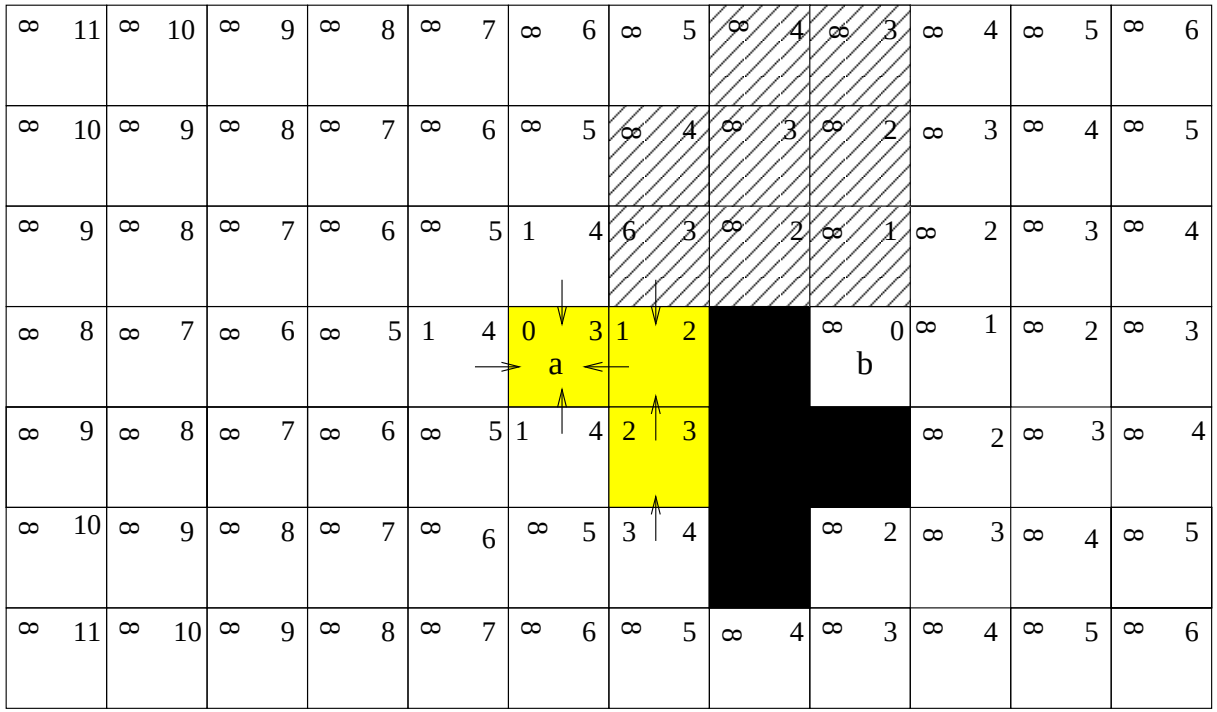
- huom: valitun solmun yläpuolella on vaikeakulkuinen maasto, joten kaari yläpuolella olevaan solmuun on pituudeltaan 5

- 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 oleva solmu seuraavaksi käsiteltäväksi

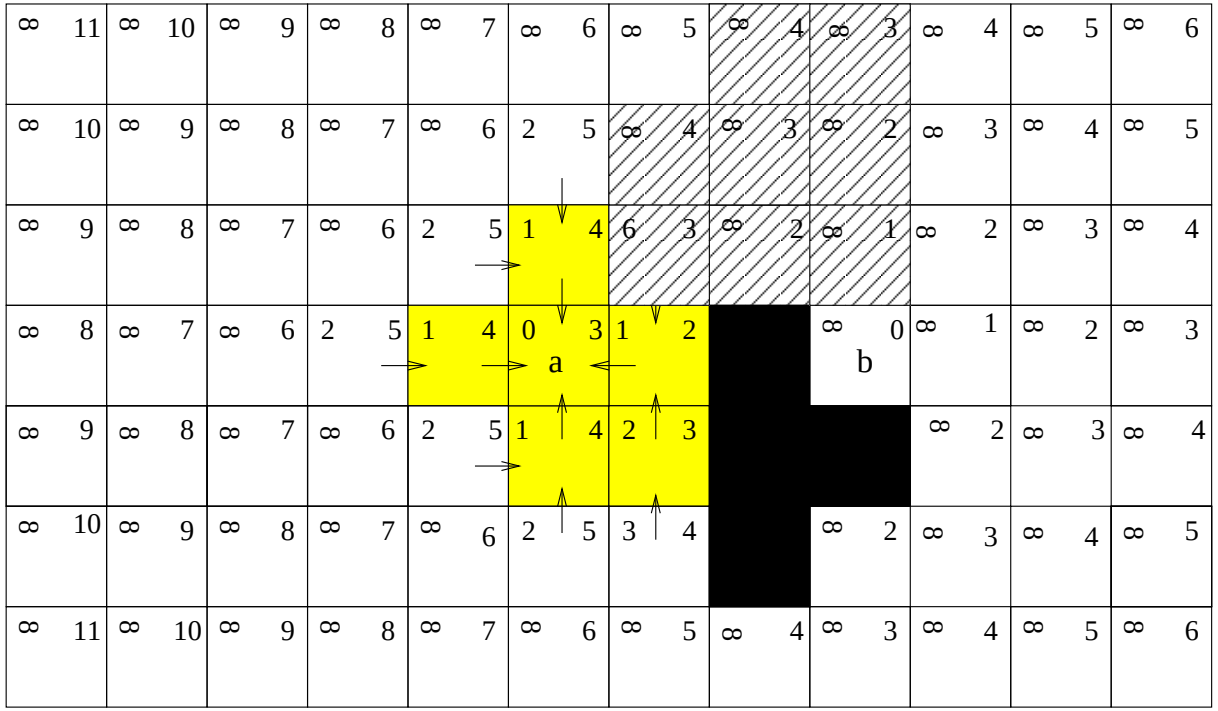
- päivityksen jälkeen tilanne näyttää seuraavalta



- 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, tutkii algoritmi ne ennen kuin lähtee oikeaan suuntaan

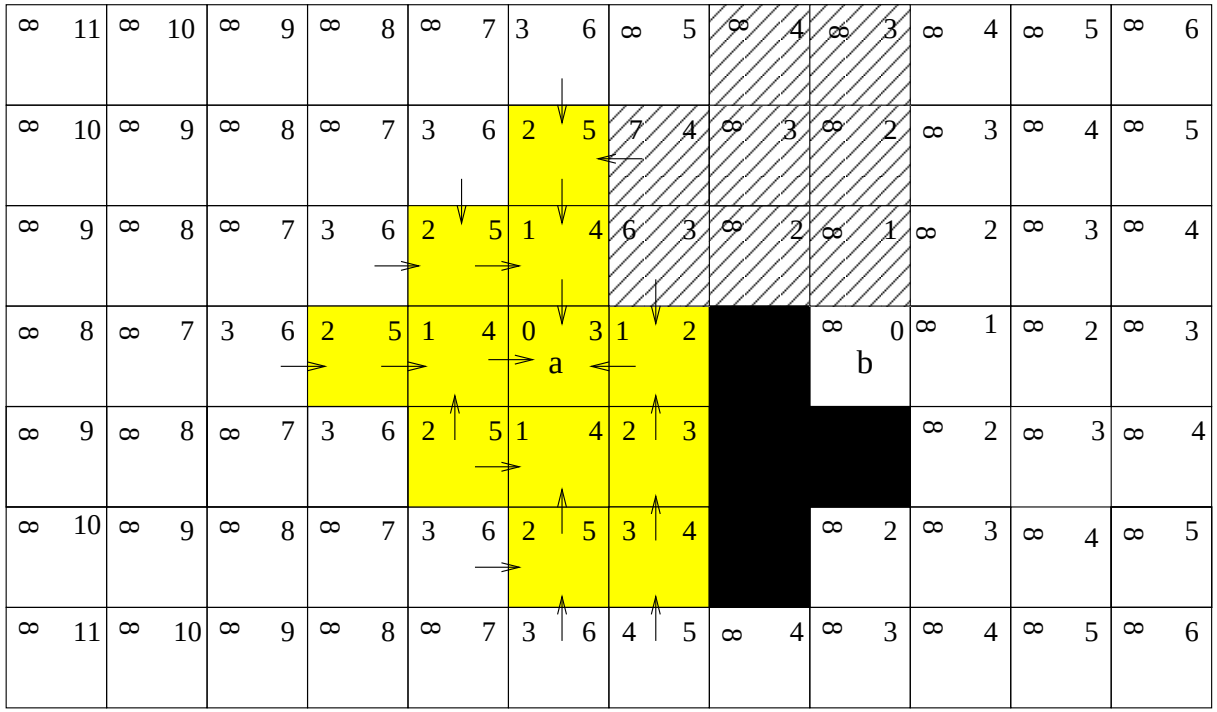
- seuraavassa tilanne kolmen "väärässä suunnassa" olevan solmun tutkimisen jälkeen



- verkossa on nyt 6 solmua joille $alkuun(v) + loppuun(v) = 7$

- vaikka osa näistä on väärässä suunnassa, tutkii algoritmi ne seuraavissa askeleissa

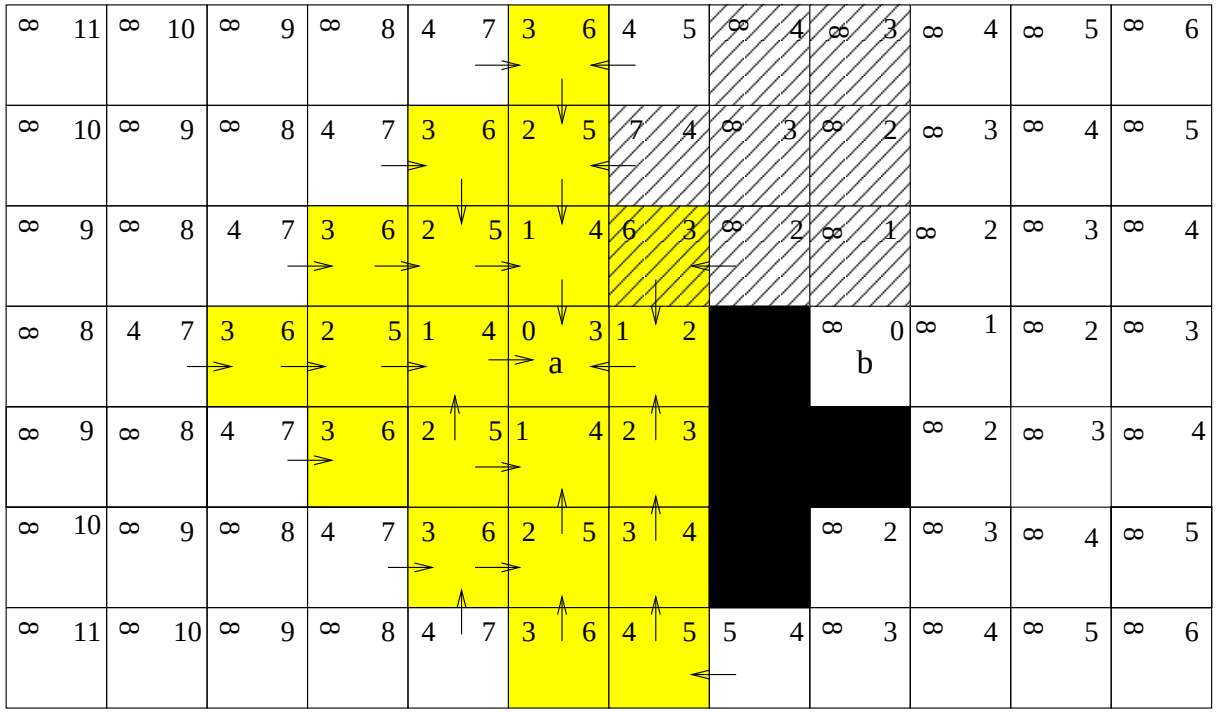
- päädytään seuraavaan tilanteeseen



- Seuraavissa vaiheissa vuorossa olevat solmut, joille $alkuun(v) + loppuun(v) = 9$, näitä solmuja on 9

- tutkitaan nämä solmut

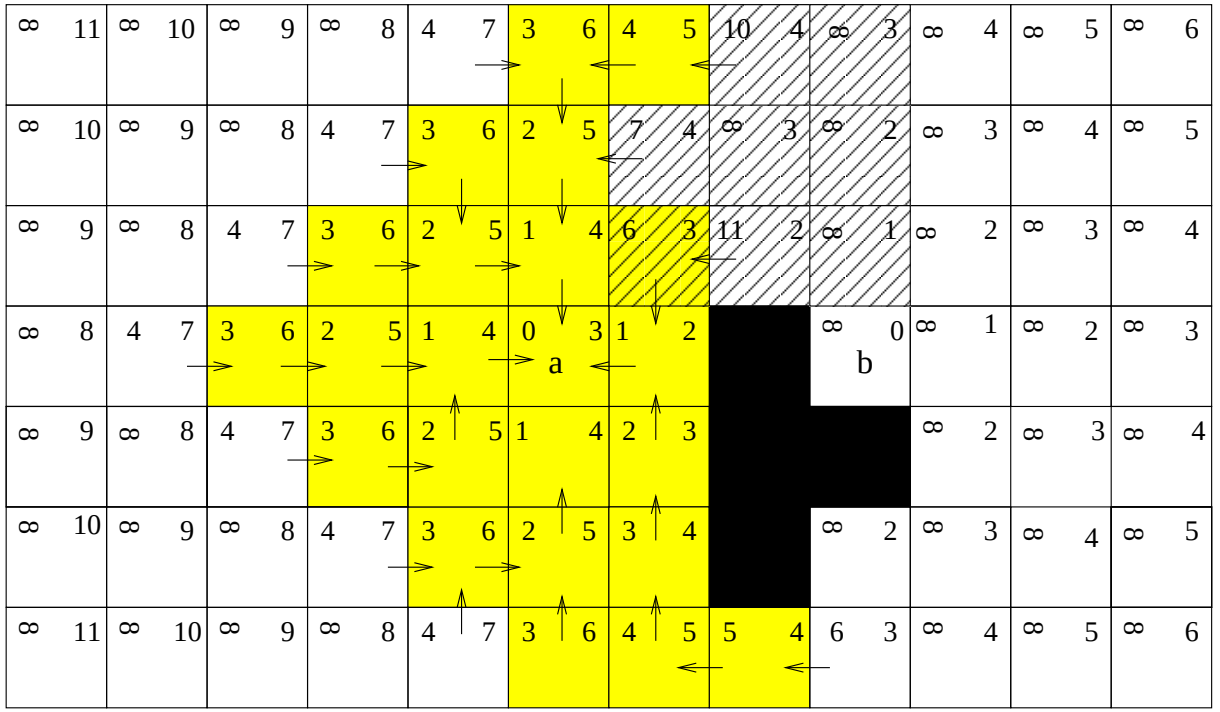
- päädytään seuraavaan tilanteeseen



- 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 $alkuun(v) + loppuun(v) = 9$

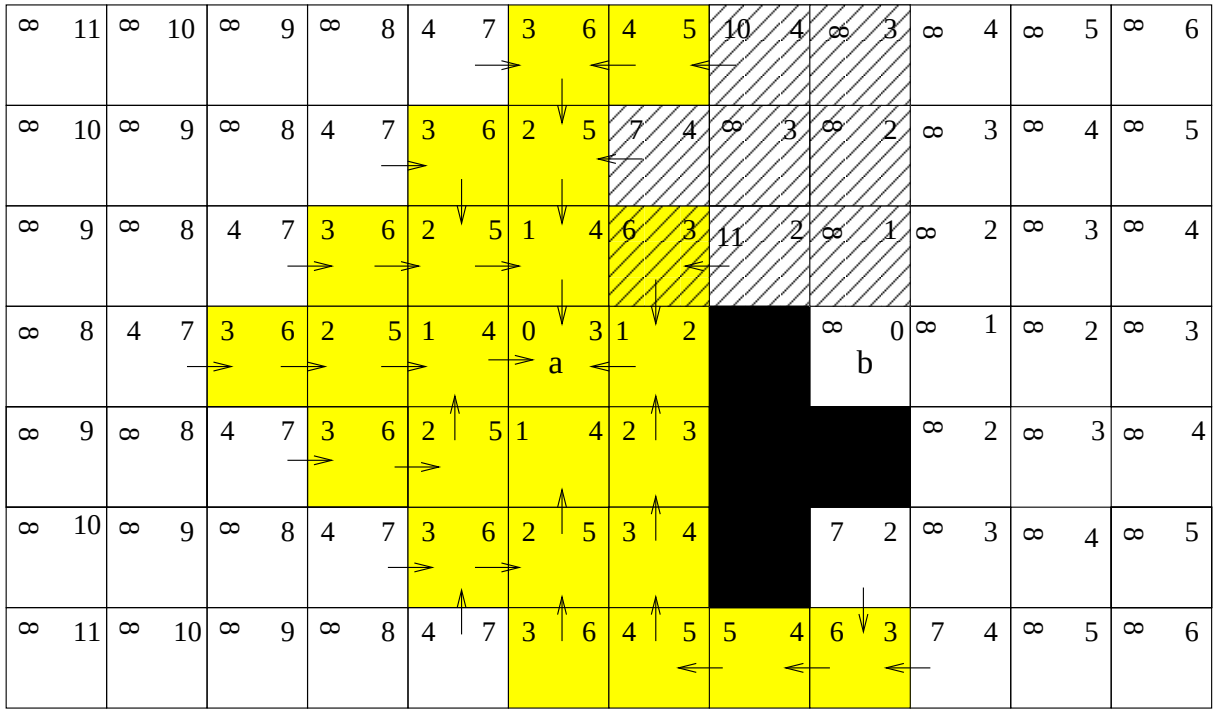
- päädytään seuraavaan tilanteeseen



- jälleen löytyy uusi solmu, jolle $alkuun(v) + loppuun(v) = 9$

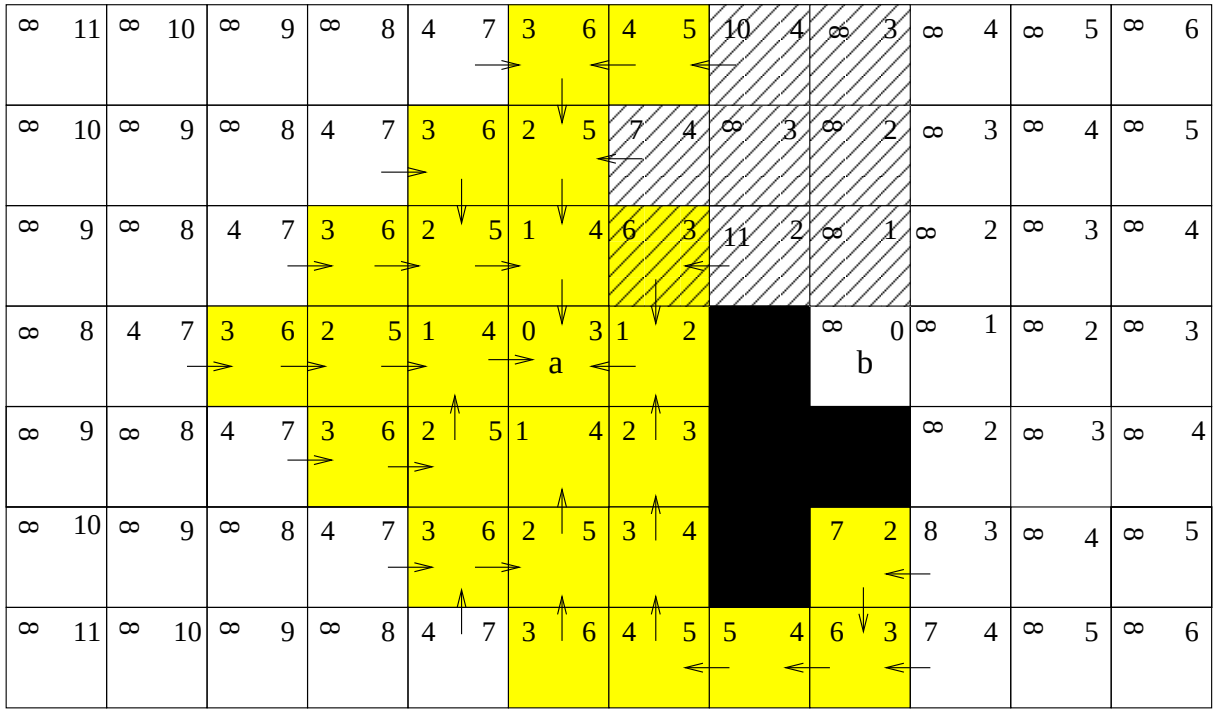
- valituksi tuleva solmu on askeleen lähempänä maalisolmua, joten algoritmi jatkaa oikeaan suunntaan

- päädytään seuraavaan tilanteeseen



- jälleen löytyy uusi solmu, jolle $alkuun(v) + loppuun(v) = 9$
- algoritmi valitsee solmun käsittelyyn

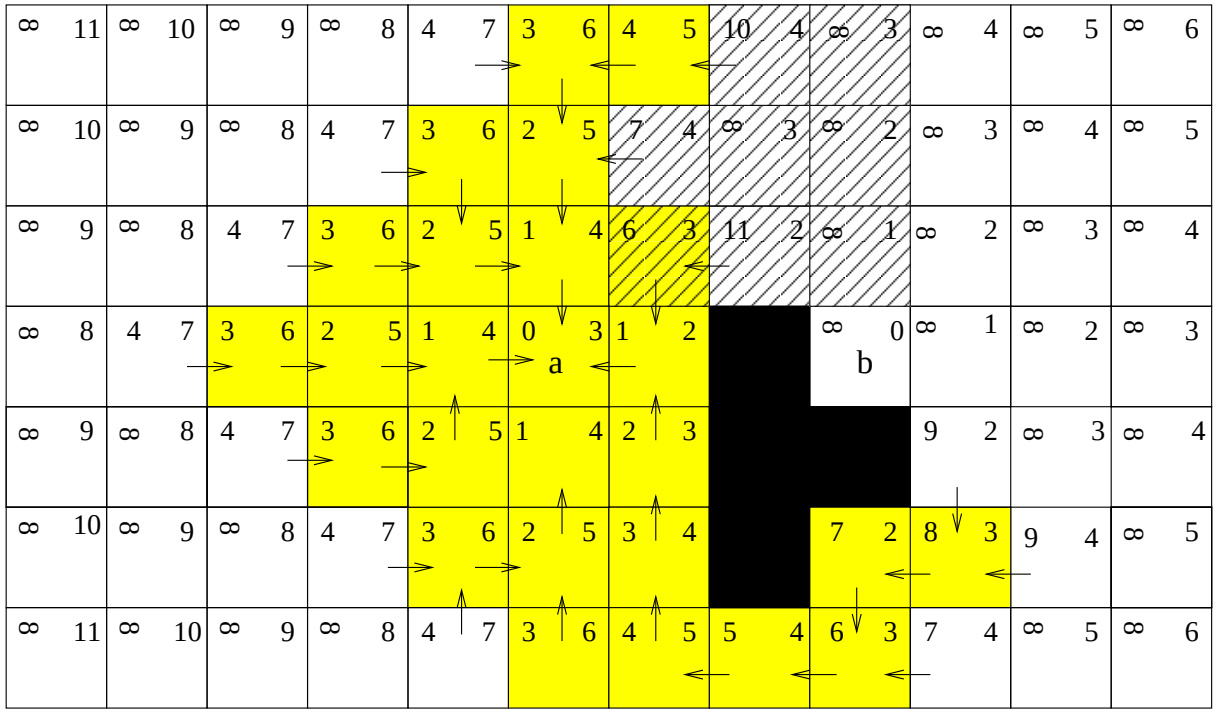
- päädytään seuraavaan tilanteeseen



- seuraavana käsitteyvuorossa yksi solmuista, jolle $alkuun(v) + loppuun(v) = 11$

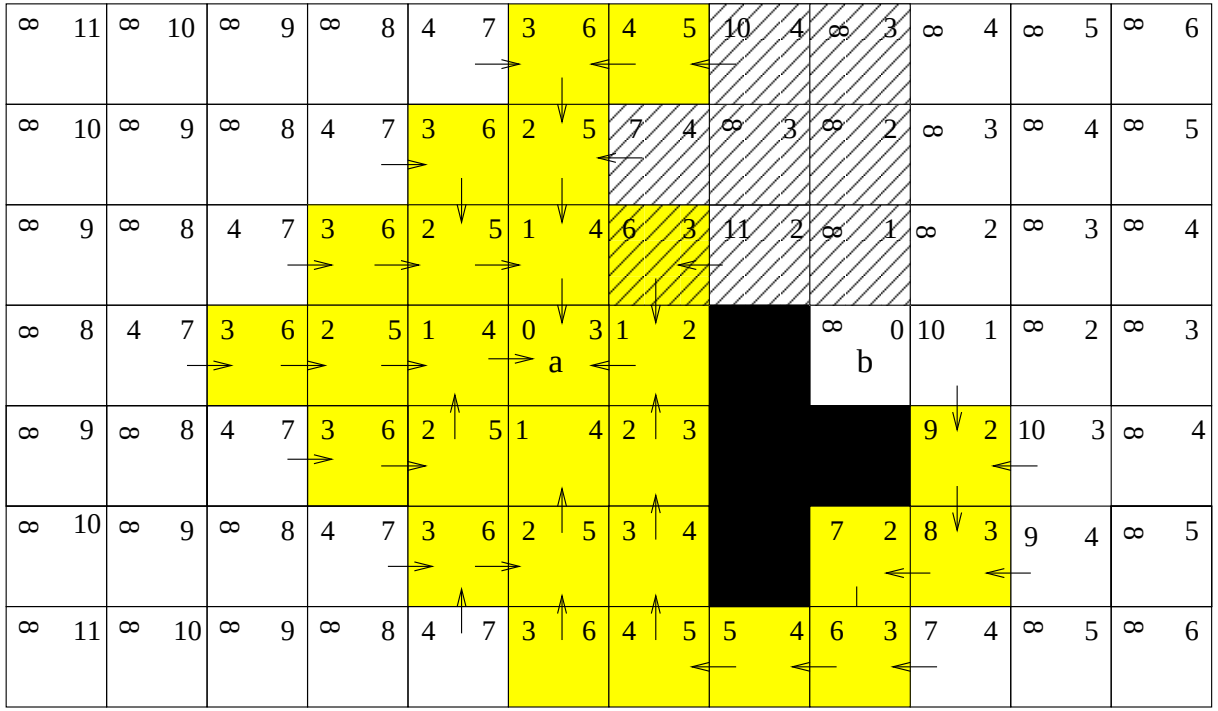
- oletetaan, että algoritmi valitsee näistä sattumalta käsittelyyn lähempänä maalia olevan

- päädytään seuraavaan tilanteeseen



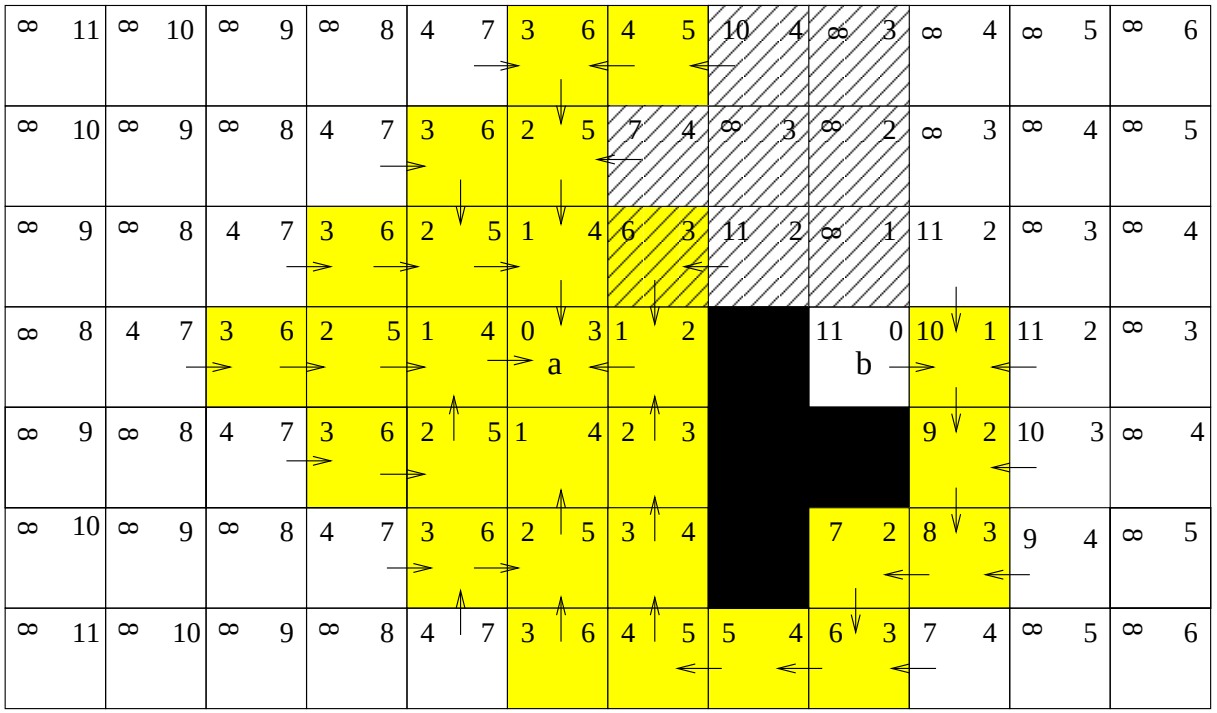
- jälleen käsitteyvuorossa yksi solmuista, jolle $alkuun(v) + loppuun(v) = 11$
- oletetaan, että algoritmi valitsee jälleen näistä maalia lähimpänä olevan

- 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 ∈ Adj[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 (ks. kalvo 453)

- Jos oletetaan, että etäisyysarvio $loppuun[v]$ on laskettavissa vakioajassa, on algoritmin pahimman tapauksen aikavaativuus 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 riippuen sovelluksesta

- 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*
  - Jos etäisyysarvio liiottelee joidenkin solmujen etäisyyksiä, löytää algoritmi 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
  - esimerkkitapauksessamme polku $a \rightsquigarrow b$ löytyisi nopeammin jos loppumatkan etäisyysarviona käytettäisiin esim. kahdella kerrottua Manhattan-etäisyyttä
  - tällöin algoritmi välttelisi voimakkaamiin väärän kulkusuunnan tutkimista

- A*-algoritmista ei ole aiemmin käsitelty Tietorakenteissa. Nytkään **algoritmi ei kuulu koealueeseen**
- A* on kuitenkin suosittu aihe Tietorakenteiden harjoitustyössä
- Algoritmia käsitellään jonkin verran kurssilla *Johdatus tekoälyyn*
- Lisätietoa A*:sta:
  - `www.policyalmanac.org/games/aStarTutorial.htm`
  - `theory.standford.edu/~amitp/GameProgramming/`
  - Russell and Norvig: Artifical Inteligence, A modern Approach