

58131 Tietorakenteet ja algoritmit

Kevät 2015

Patrik Floréen ja Jyrki Kivinen

Sisällys

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

1. Johdanto ja kertausta

Kurssin asema opetusohjelmassa

- Tietojenkäsittelytieteen aineopintokurssi, 8–10 op, pääaineopiskelijoille pakollinen
 - Lisäopintopisteitä (1 tai 2 op) saa runsaasta laskuharjoitus- ja ohjelmointitehtävien ratkaisemisesta
- Esitietoina [Johdatus yliopistomatematiikkaan](#) (ennen: [Johdatus diskreettiin matematiikkaan](#)) ja [Ohjelmoinnin jatkokurssi](#)
- Jatkokurssi [Design and Analysis of Algorithms](#) (5 op, syventävät, pakollinen algoritmien, data-analytiikan ja koneoppimisen linjalla, valinnainen muille)
- Tällä kurssilla opittavat tekniikat ja ajattelutavat ovat tarpeen kurssilla [Laskennan mallit](#) (6 op, pääaineopiskelijoille pakollinen aineopintokurssi)

Esitietovaatimuksista

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

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

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

Lähdemateriaali

- Tämä luentomateriaali on uudistettu painos edellisvuosien kalvoista (tekijöinä mm. Matti Luukkainen ja Matti Nykänen)
- Suurin osa käsiteltävistä asioista perustuu kirjaan
T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: [Introduction to Algorithms](#). 3rd ed. MIT Press, 2009.

Myös kirjan toinen painos on käyttökelpoinen

- Cormenin kirja kattaa suurimman osan kurssilla käsiteltävistä asioista ja onkin suositeltavin teos, jos olet aikeissa ostaa jonkun kirjan
- Suurin ero Cormenin painosten 2 ja 3 välillä on pseudokoodin esitysmuodossa. Tässä monisteessa käytetään kolmannen painoksen pythonmaista pseudokoodia

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

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

- Muita lähteitä on mm.

A. Levitin: [Introduction to The Design and Analysis of Algorithms](#), Addison-Wesley, 2003

J. Kleinberg, E. Tardos: [Algorithm Design](#), Pearson Addison-Wesley, 2006

A. Aho, J. Hopcroft, J. Ullman : [Data Structures and Algorithms](#), Addison-Wesley, 1983

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

Miksi pakollinen kurssi Tietorakenteet?

- Monet tietorakenteet on helppo toteuttaa Javan valmiiden tietorakennetoteutusten avulla. Esim. seuraavassa luvussa esitettävä puhelinluettelo voidaan toteuttaa käyttäen Javan `TreeMap`
Miksi siis vaivautua opiskelemaan tietorakenteita 8–10 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
- Kaikkiin ongelmiin ei löydy suoraan valmista tietorakennetta tai algoritmia ainakaan kielten standardikirjastojen joukosta
Esim. seuraavassa luvussa esitettävä kaupunkien välisten lyhimpien reittien tehokas etsiminen ei onnistu Javan valmiiden mekanismien avulla
- On siis hyvä omata yleiskäsitys erilaisista tietorakenteista ja algoritmeista, jotta vastaantulevia ongelmia pystyy jäsentämään siinä määrin, että ratkaisun tekeminen onnistuu tai lisäinformaation löytäminen helpottuu

- Voisi myös kuvitella, että tietokoneiden jatkuva nopeutuminen vähentää edistyneiden tietorakenteiden ja algoritmien merkitystä
- Kuten lyhimpien polkujen esimerkistä sivulla 30–31 voidaan havaita, 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
 - hajautettu laskenta
 - reaaliaikainen laskenta
 - mobiililaitteiden rajoitettu kapasiteetti
 - 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, data-analytiikka ja koneoppiminen](#) -linjan kursseilla, joista erityisesti kurssia [Design and Analysis of Algorithms](#) voi pitää Tiran "jatkokurssina"
- Algoritmeihin liittyvästä täsmällisestä argumentoinnista on hyötyä tietysti myös normaalissa ohjelmoinnissa, testaamisessa ym.
- Teoreettisesta lähestymistavasta huolimatta kurssilla on tarkoitus pitää myös käytäntö mielessä:
 - miten opitut tietorakenteet ja algoritmit voidaan toteuttaa esim. Javalla?
 - mikä ohjelmointikielen tarjoamista valmiista tietorakenne- ja algoritmitoteutuksista kannattaa valita oman ongelman ratkaisuun?

Kurssin tavoitteet

- Yleisemmin voidaan todeta, että kurssilla etsitään vastauksia seuraavanlaisiin kysymyksiin:
 - miten laskennassa tarvittavat tiedot organisoidaan tehokkaasti?
 - miten varmistumme toteutuksen toimivuudesta?
 - miten analysoimme toteutuksen tehokkuutta?
 - millaisia tunnettuja tietorakenteita voimme hyödyntää?
- Oppimistavoitematriisi (ks. kurssisivu) määrittelee tarkemmin, mikä on oleellisinta kurssilla.
 - Sarake **lähestyy oppimistavoitetta** määrittelee suunnilleen arvosanaan 1 vaadittavan osaamisen tason
 - Sarake **saavuttaa oppimistavoitteet** määrittelee korkeimpiin arvosanoihin vaadittavan oppimisen tason
- Osa asioista (erityisesti algoritmien ja tietorakenteiden toteuttaminen Javalla) on sen luonteista, että niiden hallinta osoitetaan kokeiden sijasta vain harjoituksissa

Matematiikan kertausta: Logiikka

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

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

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

Matematiikan kertausta: Todistusmenetelmiä

- Halutaan todistaa, että (todesta) oletuksesta A seuraa väite B eli $A \Rightarrow B$. Todistamiseen on useita erilaisia tekniikoita. Se, mikä kulloinkin sopii parhaiten, riippuu tilanteesta
- **Deduktiivisessa todistuksessa**, jota kutsutaan myös suoraksi todistukseksi, oletetaan A ja näytetään, että tästä seuraa B

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

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

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

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

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

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

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

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

- Miksi vastaväitetodistus toimii? Katsomme mitä tämä loogisesti tarkoittaa
- Osoitamme vastaväitetodistuksessa, että $A \wedge \neg B$ on epätosi. Vertaamme totuusarvotaulukon avulla tätä implikaatioon:

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

- Näemme siis, että $A \wedge \neg B$ on epätosi täsmälleen silloin kun $A \Rightarrow B$ on tosi
- Katsomme vielä, miten voidaan osoittaa jotain epätodeksi. Helpoin tapa on esittää tapaus, jossa väite on epätosi

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

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

□

Matematiikan kertausta: Logaritmi

- Koulumatematiikasta muistetaan, että $\log_a x$ on sellainen luku z , jolla $a^z = x$. Tässä oletetaan yleensä $a > 0$, $a \neq 1$ ja $x > 0$
- Logaritmi $\log_a n$ siis kertoo, kuinka monta kertaa luku n pitää jakaa a :lla, ennen kuin päästään lukuun 1. Esimerkiksi $\log_2 8$ on 3, koska $8/2/2/2$ on 1. Jos lukuun 1 ei päästä tasajaolla, logaritmin loppuosa on desimaaliosa, joka kuvaa viimeistä vaillinaista jakoa. Esimerkiksi $\log_2 9$ on noin 3,17, koska $9/2/2/2$ on 1,125, eli kolmen jaon jälkeen on vielä hieman matkaa lukuun 1, mutta neljäs täysi jako veisi jo selvästi alle yhden
- Luonnollinen logaritmi on $\ln x = \log_e x$, missä kantalukuna on Neperin luku $e = \lim_{n \rightarrow \infty} (1 + 1/n)^n \approx 2,718$
- Kaksikantainen logaritmi $\log_2 n$ kertoo likimäärin luonnollisen luvun n pituuden bitteinä (tarkemmin: luvun $n > 0$ pituus bitteinä on $\lfloor \log_2 n \rfloor + 1$)
- Laskusääntöjä:

$$\log_a(xy) = \log_a x + \log_a y$$

$$\log_a(x^y) = y \log_a x$$

$$\log_a x = \frac{\log_b x}{\log_b a}$$

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

Ohjelmointitaitoa: Rekursio

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

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

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

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

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

- Ajonaikainen pino on Javassa osaa Javan virtuaalikonetta (Java Virtual Machine, JVM), joka on konekielinen ohjelma, joka suorittaa Java-ohjelman tavukoodia (eng. byte code)
- Java varaa ajonaikaiselle pinolle tietyn koon, ja jos tämä ylittyy tulee virheilmoitus `StackOverflowError`
- Kun aktivaatitietueet aiheutuvat rekursiivisesta metodista, puhumme rekursiopinosta (täältä osalta ajonaikaista pinoa)
- Kutsu `potenssi(2.0,4)` luo aktivaatitietueen, jossa on parametrit 2.0 ja 4 sekä paluusoitteeksi mistä kutsu tuli

- Koska n ei ole 0, suoritetaan rivi

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

- Mutta tämä sisältää kutsun potenssi(2.0,3), eli nyt laitetaan pinoon parametrit 2.0 ja 3 ja paluusoitteeksi tämä potenssi(2.0,4):n return-rivi, josta kutsu potenssi(2.0,3):een tuli
- Seuraavaksi potenssi(2.0,2) laittaa pinoon parametrit 2.0 ja 2 ja paluusoitteeksi potenssi(2.0,3):n return-rivi
- Seuraavaksi potenssi(2.0,1) laittaa pinoon parametrit 2.0 ja 1 ja paluusoitteeksi potenssi(2.0,2):n return-rivi
- Seuraavaksi potenssi(2.0,0) laittaa pinoon parametrit 2.0 ja 0 ja paluusoitteeksi potenssi(2.0,1):n return-rivi
- Pinon sisältö on siis seuraava:

2.0	0
2.0	1
2.0	2
2.0	3
2.0	4

- Nyt ohjelma palauttaa arvon 1.0 eikä kutsu enää itseään, eli poistetaan pinosta tietue, jossa on 2.0 ja 0 ja paluusoitteena potenssi(2.0,1):n return-rivi ja siirrytään sille riville
- Suoritetaan potenssi(2.0,1):n return-rivin lasku $2.0 * 1.0 = 2.0$ ja poistetaan pinosta tietue, jossa on 2.0 ja 1 ja paluusoitteena potenssi(2.0,2):n return-rivi ja siirrytään sille riville
- Suoritetaan potenssi(2.0,2):n return-rivin lasku $2.0 * 2.0 = 4.0$ ja poistetaan pinosta tietue, jossa on 2.0 ja 2 ja paluusoitteena potenssi(2.0,3):n return-rivi ja siirrytään sille riville
- Suoritetaan potenssi(2.0,3):n return-rivin lasku $2.0 * 4.0 = 8.0$ ja poistetaan pinosta tietue, jossa on 2.0 ja 3 ja paluusoitteena potenssi(2.0,4):n return-rivi ja siirrytään sille riville
- Suoritetaan potenssi(2.0,4):n return-rivin lasku $2.0 * 8.0 = 16.0$ ja poistetaan pinosta tietue, jossa on 2.0 ja 4 ja paluusoitteena paikka mistä metodi kutsuttiin

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

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

- Tämä toteutus vie paljon vähemmän tilaa (ja on nopeampi), koska ei tarvita rekursiopinoa!
- Hyvät ja huonot puolet: Rekursiivinen ohjelma on yleensä helpompi ymmärtää ja ohjelmoida, koodi on lyhyempää. Toisaalta rekursiivinen ohjelma on yleensä hitaampi ja vie enemmän muistia

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

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

- Binomikerroin

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

on usein järkevää laskea käyttämällä palautuskaavaa

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

2. Johdanto tietorakenteisiin

- Kaikki epätriviaalit ohjelmat joutuvat tallettamaan ja käsittelemään tietoa suoritusaikanaan
- Esim. "puhelinluettelo":
 - numeron lisäys
 - numeron poisto
 - numeron muutos
 - numeron haku nimen perusteella
 - nimen haku numeron perusteella
 - nimi-numero–parien tulostaminen aakkosjärjestyksessä
 - ...
- Suoritusaikana tiedot tallennetaan [tietorakenteeseen](#) (engl. data structure)

Tietorakenne

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

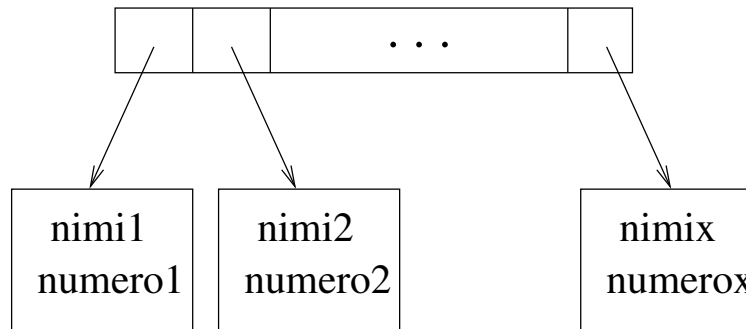
Esim. puhelinluettelo-ohjelmiston tietorakenne

- Tietorakenteen operaatioita ovat ainakin seuraavat:
 - **add**(n, p) lisää luetteloon nimen n ja sille numeron p
 - **del**(n) poistaa nimen n luettelosta
 - **findNumber**(n) palauttaa luettelosta henkilön n numeron
 - **findName**(p) palauttaa luettelosta numeron p omistajan
 - **update**(n, p) muuttaa n :lle numeroksi p :n
 - **list**() listaa nimi-numero -parit aakkosjärjestyksessä
- Seuraavassa oletetaan, että yhdellä henkilöllä voi olla ainoastaan yksi puhelinnumero

- Tieto voisi olla talletettu suoraan taulukkoon

nimi1 numero1	nimi2 numero2	...	nimix numerox
------------------	------------------	-----	------------------

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



- Javaa käytettäessä olisi luonnollista toteuttaa yksittäinen nimi/numero-pari omana luokkana ja itse puhelinluettelo omana luokkana
Puhelinluettelon metodeina olisivat edellisen sivun operaatiot ja yhtenä attribuuttina taulukko viitteitä nimi/numero-pari-olioihin
- Taulukkoon perustuvat tietorakenteet ovat tehottomia (suuren) puhelinluettelon toteuttamiseen. Opimme kurssin kuluessa useita parempia tietorakenteita (mm. hakupuut ja hajautusrakenteet) ongelman ratkaisemiseen

Esim. miten löydetään lyhin kahden kaupungin välinen reitti?

- Tarkastellaan toisena esimerkkinä aivan toisentyyppistä ongelmaa

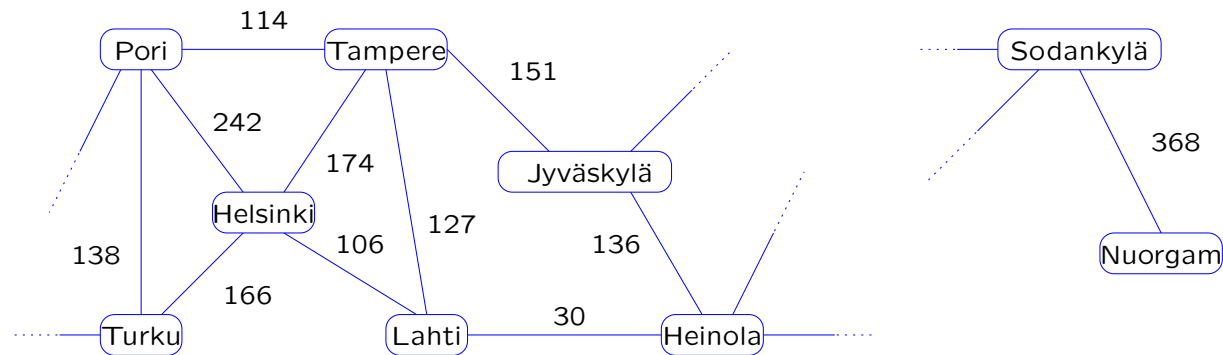
Annettu: Suomen paikkakuntien väliset suorat maantie-etäisyydet

Kysymys: paljonko matkaa Helsingistä Nuorgamiin?

Jatkokysymys: mikä on lyhin reitti Helsingistä Nuorgamiin?

Helsinki–Lahti	106 km		Helsinki
Helsinki–Turku	166 km		→ Lahti
Turku–Pori	138 km		→ Heinola
Lahti–Tampere	127 km	⇒	...
...	...		→ Sodankylä
Sodankylä–Nuorgam	368 km		→ Nuorgam
			yht. 1328 km

- Tilanteeseen sopiva tietorakenne on **painotettu suuntaamaton verkko**



- Kysymyksessä on **lyhimmän polun** etsiminen verkosta
- Jos puhelinluettelo toteutetaan taulukkoon perustuen, on kaikkien operaatioiden toteuttaminen suoraviivaista
- Sen sijaan ei ole ollenkaan selvää, miten löydetään verkosta lyhin polku tehokkaasti

- Ongelma voitaisiin ratkaista raa'alla voimalla
käydään järjestyksessä läpi kaikki mahdolliset reitit kaupunkien välillä (eli kaupunkien permutaatiot) ja valitaan niistä lyhin
- Ratkaisu on erittäin tehoton: oletetaan että on olemassa 32 paikkakuntaa \Rightarrow suuruusluokkaa $30! \approx 2,7 \cdot 10^{32}$ mahdollista reittiä!
- Oletetaan, että kone testaa miljoona reittiä sekunnissa \Rightarrow tarvitaan $8,5 \cdot 10^{18}$ vuotta
- Raakaan voimaan perustuva ratkaisu on siis käytännössä käyttökelvoton
- Kurssin loppupuolella esitetään useampikin tehokas algoritmi lyhimpien polkujen etsimiseen
Eräs näistä, ns. Dijkstran algoritmi käyttää itsessään suorituksen apuna keko-nimistä tietorakennetta

Yhteenveto

- Tietorakenteita tarvitaan
 - suurten tietomäärien hallintaan (puhelinluettelo)
 - ongelmien mallintamiseen (kaupunkien väliset minimaaliset etäisyydet)
 - 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 ratkaisusta on paras
 - kuinka paljon dataa on
 - kuinka usein mitäkin operaatiota suoritetaan
 - mitkä ovat eri operaatioiden nopeusvaatimukset
 - onko muistitilasta pulaa jne.

Algoritmien oikeellisuus

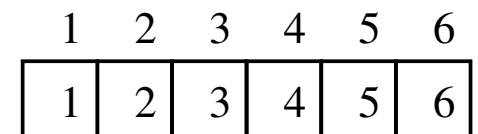
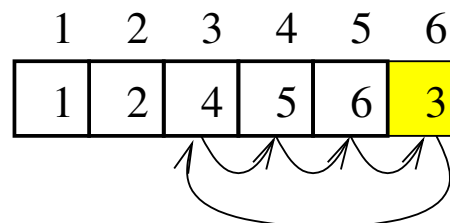
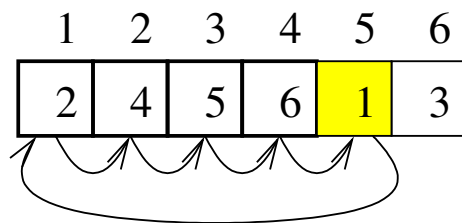
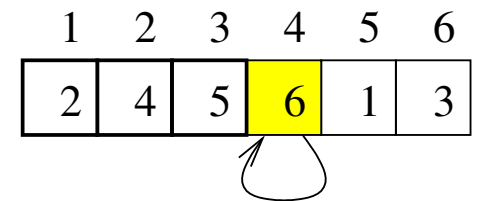
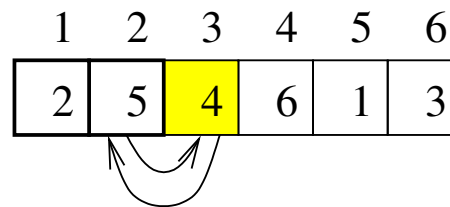
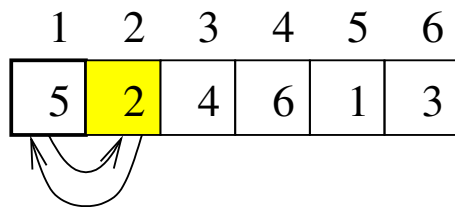
- Yksi tapa toteuttaa puhelinluettelon operaatio **add** (taulukkototeutuksessa) on lisätä uusi nimi-numero-pari aina taulukon ensimmäiseen tyhjään paikkaan
- Tällöin operaation **list** yhteydessä nimet täytyy järjestää aakkosjärjestykseen
- Järjestäminen on yleisemminkin tärkeässä roolissa tietojenkäsittelyssä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

insertion-sort(A)

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

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

- Toimintaideana algoritmissa on pitää koko ajan taulukon alkupää $A[1, \dots, j - 1]$ järjestyksessä:
 - alussa $j = 2$ eli alkupää sisältää vain yhden luvun
 - **while**-silmukassa siirretään kohdassa j oleva luku oikeaan kohtaan alkupään järjestyksessä olevaa osaa
 - näin taulukon loppupään luvut viedään yksi kerrallaan alkupään järjestettyyn osaan ja lopulta kaikki luvut ovat järjestyksessä
- Algoritmin toiminta esimerkkisyötteellä, keltaisella merkitty alkio $A[j] = x$ jolle etsitään taulukon alkupäästä oikea paikka



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

for-lauseen jokaisen suorituksen alussa taulukon osa $A[1, \dots, j - 1]$ on järjestyksessä ja sisältää alunperin taulukon osassa $A[1, \dots, j - 1]$ olleet alkioit
- Invariantteja voidaan käyttää algoritmin oikeellisuustodistuksissa
- Jokin ylläolevan kaltainen väittämä **osoitetaan invariantiksi** näyttämällä, että
 1. invariantti on voimassa tullessa ensimmäistä kertaa toistolauseeseen
 2. invariantti pysyy totena jokaisen toistolauseen rungon suorituksen jälkeen
- Askelien 1 ja 2 voimassaolosta seuraa se, että jos invariantti on aluksi tosi, pysyy se voimassa koko silmukan suorituksen ajan
- Erityisesti invariantti on vielä totta siinäkin vaiheessa kun silmukan suoritus loppuu
- Jos invariantti on valittu järkevästi, niin **toistolauseen loppuminen yhdessä invariantin voimassaolon kanssa takaa halutun lopputuloksen**

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

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

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

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

- **for**-lauseen rungon suorituksen jälkeen siis on voimassa

taulukon osa $A[1, \dots, j]$ on järjestyksessä ja sisältää alunperin taulukon osassa $A[1, \dots, j]$ olleet alkio

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 $A[1, \dots, A.length]$ on järjestyksessä ja sisältää alunperin taulukossa olleet alkio eli algoritmi toimii oikein

- Koska kyseessä on `for`-lause, tiedämme varmasti, että sen suoritus päättyy (sillä oletuksella että runko-osa ei jää silmukkaan!), sen sijaan `while`-toistolause saattaisi jäädä ikuisen silmukkaan
- Yleisen toistolauseen tapauksessa on siis invariantin voimassa pysymisen lisäksi näytettävä että toistolause pysähtyy
- Edellisen sivun oikeellisuustodistuksen argumentaatio, erityisesti sen osalta miksi invariantti säilyy totena `for`:in rungossa olevan `while`:n ansiosta, olisi voitu tehdä formaalimmin muotoilemalla `while`:lle oma invariantti
- Liian detaljoidulla tasolla tapahtuva argumentaatio ei kuitenkaan liene tarpeen, tärkeintä on ymmärtää tarkasti mitä algoritmi tekee ja tehdä päättelyt tarvittavalla formaaliuden tasolla
- Jos algoritmi koostuu peräkkäisistä osista, joissa kaikissa on oma silmukkansa, tarvitaan kaikille silmukoille omat invariantit. Yleensä myöhempien silmukoiden invariantit perustuvat oleellisesti siihen tilanteeseen, johon algoritmin syöte on muotoutunut ensimmäisten silmukoiden suorituksessa

Invariantin löytäminen

- Invariantti siis ilmaisee jonkin asiain tilan, joka on voimassa koko silmukan suorituksen ajan
- Invariantti tavallaan kiteyttää koko silmukan toimintaidean
- Algoritmin keksijällä onkin yleensä ensin mielessä invariantin kaltainen idea, joka sitten ilmaistaan koodimuodossa
- Jos halutaan todistaa algoritmin oikeellisuus invariantin avulla, on oleellista löytää sellainen invariantti, jonka avulla oikeellisuus voidaan päätellä
- Voitaisiin esim. näyttää, että seuraava on **lisäysjärjestämisen** invariantti:
jokaisen for:in jokaisen suorituksen alussa taulukon osa $A[1, \dots, j - 1]$ sisältää alunperin taulukon osassa $A[1, \dots, j - 1]$ olleet alkiot
- Tämä invariantti ei kuitenkaan auta todistamaan että **lisäysjärjestäminen** järjestää taulukon, joten se on lähes hyödytön
- Invarianttitodistuksissa lähes suurimmaksi haasteeksi osoittautuukin sopivan invariantin löytäminen
- Ilman syvällistä ymmärtämystä algoritmin toimintaperiaatteesta on sopivaa invarianttia lähes mahdoton löytää

Toinen invarianttiesimerkki

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

suurin(A)

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

- Metodin toiminta perustuu sille, että muuttuja s muistaa taulukon alkuosan alkioista suurimman
 - alussa s saa arvokseen taulukon ensimmäisen alkion
 - toistolauseessa verrataan tunnettua suurinta arvoa aina tutkittua aluetta seuraavassa paikassa olevaan ja päivitetään s :n arvo tarvittaessa
 - lopulta on käyty läpi kaikki taulukon alkiot ja palautetaan s joka sisältää koko taulukon suurimman alkion
- Algoritmin ytimenä olevan `for`-lauseen toimintaa siis kuvaa invariantti
 s :n arvo on sama kuin suurin arvo taulukon osasta $A[1, \dots, i - 1]$

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

Invarianttitodistuksen ja matemaattisen induktion samankaltaisuus

- Invarianttitodistus on melkein kuten induktiotodistus matematiikassa
 1. Induktiotodistuksessa 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 silmukan lopussa päädytään haluttuun tulokseen sillä induktiossahan halutaan todistaa, että tietty väittämä pätee kaikilla arvoilla
- Esim. järjestämisalgoritmin taas täytyy toimia oikein tietyn kokoiselle taulukolle. Taulukon koon ei sinänsä tarvitse olla rajattu. Invarianttitodistuksessa kuitenkin tiedetään, että taulukolla on jokin koko, eli "taulukon ulkopuolella" olevia lukuja ei tarvitse järjestää ja on oleellista, että silmukan suorittaminen loppuu jossain vaiheessa

Huomautuksia monisteessa käytettävästä pseudokoodista

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

- Pseudokoodin eroavuuksia Javaan:
 - taulukon paikkojen numerointi alkaa ykkösestä
 - loogiset konnektiivit kirjoitetaan `and`, `or`, `not`
 - pseudokoodin `for` muistuttaa enemmän Javan `for eachia` kuin normaalia `for`-lausetta
 - muuttujien, metodien parametrien tai paluuarvon tyyppiä 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 syötteen parametreina
 - modernien skriptikielten 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, tarvittavan tietoliikenteen määrä, ...
- Tällä kurssilla tarkastellaan lähinnä **aikavaativuutta** ja joskus **tilavaativuutta**
- Yleensä algoritmin suoritukseen kuluu sitä enemmän resursseja mitä suurempi syöte on, esim. järjestäminen vie ilmiselvästi aikaa enemmän miljoonan kuin tuhannen kokoiselle taulukolle
- Resurssintarvetta kannattaakin tarkastella suhteessa algoritmin syötteen kokoon, esim taulukkoja käsittelevien algoritmien tapauksessa syötteen koko on yleensä taulukon koko
- Ei ole aina itsestään selvää, mikä on hyvä valinta algoritmin syötteen kooksi
 - Myöhemmin kurssilla huomaamme, että lyhimpien polkujen etsinnässä hyväksi arvioksi syötteen koosta osoittautuu maantiekartastossa olevien kaupunkien lukumäärä sekä kaupunkien välisten yhteyksien määrä. Algoritmin vaativuus riippuu siis molemmista näistä.

Aikavaativuus tarkemmin ilmaistuna

- Algoritmin aikavaativuudella tarkoitetaan sen käyttämää laskenta-aikaa suhteessa algoritmin syötteen pituuteen
 - laskenta-aikaa mitataan käytettyjen "pienehköjen" aikayksiköiden määrällä
 - yksinkertaisen komennon (esim. vertailu, sijoituslause, laskutoimitus tai **if**-, **while**- ja **for**-lauseiden ehdon testaaminen) suorittaminen vie vakiomäärän tällaisia aikayksiköitä
 - sen sijaan esim. mielivaltaisen kokoisen taulukon järjestäminen ei onnistu vakiomäärällä aikayksiköitä
- Aikavaativuus voidaan ilmaista funktiona, joka kuvaa syötteen pituuden algoritmin käyttämien aikayksiköiden määräksi
- Jonkun algoritmin aikavaativuus syötteen pituuden n suhteen voisi olla esim. $T(n) = 2n^2 + 3n + 7$, eli jos syötteen pituus on 2, kuluttaa algoritmi aikaa 21 yksikköä, jos taas syötteen pituus on 100, kuluttaa algoritmi aikaa 20307 yksikköä
- Kuten pian näemme, ei laskenta-aikaa yleensä ole mielekästä mitata tällä tarkkuudella

Tilavaativuus tarkemmin ilmaistuna

- Algoritmin **tilavaativuudella** tarkoitetaan sen laskennassa käyttämän aputilan määrää suhteessa algoritmin syötteen pituuteen
 - käytettyä aputilaa mitataan muistipaikoissa
 - voidaan ajatella, että muistipaikka on sen kokoinen tila, johon voidaan tallettaa totuusarvo, merkki, jokin luku (int, long, double) tai olioviite (eli käytännössä muistiosoite)
 - oleellista on, että tällainen 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. Javan tarjoamaa rajattoman kokoista BigInteger-tyyppistä arvoa
 - rajattoman kokoisien luvun viemä talletustila nimittäin riippuu luvun pituudesta, eli mielivaltaisen tällainen luku ei voi mahtua mihinkään kiinteän kokoiseen muistipaikkaan
 - formaali määritelmä aika- ja tilavaativuudelle nimittäin käsittelee syötteen pituutta bitteinä, ja rajatun kokoista muistipaikkaa vastaa siis vakio määrä bittejä

- Aikavaativuuden tapaan myös tilavaativuus voidaan esittää funktiona, joka kuvaa syötteen pituuden algoritmin laskennan apuna käyttämien muistipaikkojen määräksi
- Algoritmin syötettä ei lasketa mukaan tilavaativuuteen, eli jos algoritmi saa syötteekseen taulukon jonka koko on n , ei taulukon koolla sinänsä ole merkitystä algoritmin tilavaativuuteen
- Myöskään algoritmin koodia ei huomioida tilavaativuuteen millään tavalla

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

Taulukon suurimman alkion selvittävän algoritmin aikavaativuus monimutkaisesti laskettuna

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

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

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

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

$$T(n) = c_1 + c_3n + c_4(n - 1) + c_5x + c_6$$
 rivin 5 suorituskertoja on merkattu x :llä, ja siis pätee $0 \leq x \leq n - 1$
- Yleensä algoritmien analyysissä ollaan kiinnostuneita pahimmasta tapauksesta, tällöin rivi 5 suoritetaan $n - 1$ kertaa, eli edellinen sievenee muotoon

$$T(n) = (c_3 + c_4 + c_5)n + c_1 + c_6 - c_4 - c_5$$
- Jos ajatellaan toteutetun algoritmin suorittamisen viemää aikaa, vakiot c_1, \dots, c_6 riippuvat ohjelmointikielestä, käytettävästä laitteistosta, laitteiston kuormituksesta ym. itse algoritmin tehokkuutta ajatellen toisarvoisista seikoista
- Unohdetaan konkreettiset vakiokertoimet ja merkitään $T(n) = an + b$ joillain vakioilla a ja b eli $T(n)$ on lineaarinen funktio syötteen pituuteen n nähden
- Taulukon suurimman alkion etsiminen toimii lineaarisessa ajassa syötteen pituuteen nähden:
 syötteen koon kaksinkertaistuminen kaksinkertaistaa algoritmin suoritusajan
- Kuten pian näemme, on ikävät vakiot tapana "unohtaa" kokonaan ja merkitä aikavaativuus seuraavasti: $T(n) = \mathcal{O}(n)$

Algoritmien vaativuusluokat

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

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

Mielenkiintoiset vaativuusluokat helpoimmasta vaativimpaan

- $\mathcal{O}(1)$ vakio
 - $\mathcal{O}(\log n)$ logaritminen
 - $\mathcal{O}(n)$ lineaarinen
 - $\mathcal{O}(n \log n)$
 - $\mathcal{O}(n^2)$ neliöllinen
 - $\mathcal{O}(n^3)$, $\mathcal{O}(n^4)$ jne. polynominen
 - $\mathcal{O}(2^n)$, $\mathcal{O}(3^n)$ jne. eksponentiaalinen
- Polynomisten ja eksponentiaalisten algoritmien skaalautuvuudessa on merkittävä ero:
 - Eksponentiaaliset ovat käytännöllisiä vain pienillä (n suunnilleen joitakin kymmeniä tai maksimissaan satoja) syötteillä. esim. neliöllinen algoritmi selviää vielä miljoonienkin kokoisista syötteistä

- Huom: Kun tietojenkäsittelijä merkitsee $\log n$ tarkoitetaan silloin yleensä kaksikantaista logaritmia eli $\log_2 n$
- Laskusääntö $\log_a x = \frac{\log_b x}{\log_b a}$ kuitenkin osoittaa, että logaritmin kantaa voi vaihtaa kertomalla logaritmi sopivalla vakiolla, esim. $\log_{16} n = \frac{\log_2 n}{\log_2 16} = \frac{1}{4} \log_2 n$
- Koska kaikki logaritmien kannat ovat vakiokertoimien päässä toisistaan, ei \mathcal{O} merkintöjen yhteydessä yleensä merkitä logaritmille mitään kantalukua
 - Siis esim. jos algoritmin A aikavaativuus on $f_A(n) = 2 \log_7 n$ ja algoritmin B aikavaativuus on $f_B(n) = 7 \log_{55} n$, niin molemmat ovat \mathcal{O} -analyysin mielessä yhtä vaativia, eli $f_A(n) = \mathcal{O}(\log n)$ ja $f_B(n) = \mathcal{O}(\log n)$
- Logaritmifunktio kasvaa erittäin hitaasti. Vaativuusluokkaan $\mathcal{O}(\log n)$ kuuluvat algoritmit ovatkin todella paljon parempia kuin vaativuusluokkaan $\mathcal{O}(n)$ kuuluvat. Samoin on esim. vaativuusluokkien $\mathcal{O}(n \log n)$ ja $\mathcal{O}(n^2)$ suhteen

Onko algoritmien vaativuuden luokittelu vaativuusluokkiin perustuen mielekäs?

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

- Entä isot syötteet? Oletetaan, että järjestettävänä on 10 miljoonan kokoinen taulukko
- Oletetaan, että **lisäysjärjestäminen** suoritetaan koneella NOPEA, joka suorittaa 10^9 komentoa sekunnissa
- Oletetaan lisäksi, että **lomitusjärjestäminen** suoritetaan koneella HIDAS, joka suorittaa ainoastaan 10^6 komentoa sekunnissa. NOPEA on siis 1000 kertaa nopeampi kone kuin HIDAS
- Nyt kone NOPEA järjestää hyvin toteutetulla $\mathcal{O}(n^2)$ ajassa voimivalla **lisäysjärjestämisellä** luvut 200000 sekunnissa eli reilussa 55 tunnissa
Tuhat kertaa hitaampi kone HIDAS järjestää epäoptimaalisesti toteutetulla, mutta ajassa $\mathcal{O}(n \log n)$ toimivalla **lomitusjärjestämisellä** noin 11627 sekunnissa, eli alle 3,3:ssa tunnissa
- Eli kuten huomaamme, **suurilla syötteillä kahden vaativuusluokan välillä on ratkaiseva ero** vakiokertoimista huolimatta

Funktioiden kertaluokat

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

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

- Muodollisemmin:

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

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

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

resurssitarpeiden yläraja on g

- esim. $2n^2 + 3n + 7 = \mathcal{O}(n^2)$ sillä $2n^2 + 3n + 7 \leq 2n^2 + 3n^2 + 7n^2 = 12n^2$ jos $n \geq 1$. Eli määritelmän mukaisiksi vakioiksi voidaan valita $d = 12$ ja $n_0 = 1$
- Palataan ennestään tuttuun taulukon suurimman alkion selvittävään algoritmiin:
 - pahimmassa tapauksessa algoritmi kulutti aikaa

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

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

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

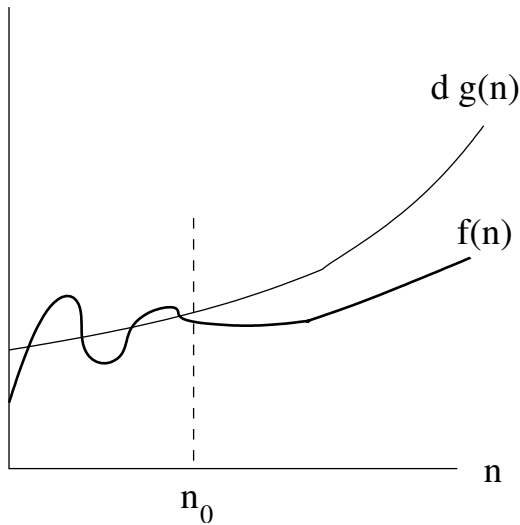
- Kannattaa huomata, että määritelmän mukaan mille tahansa vakioarvoiselle funktiolle $f(n)$ on voimassa $f(n) = \mathcal{O}(1)$.

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

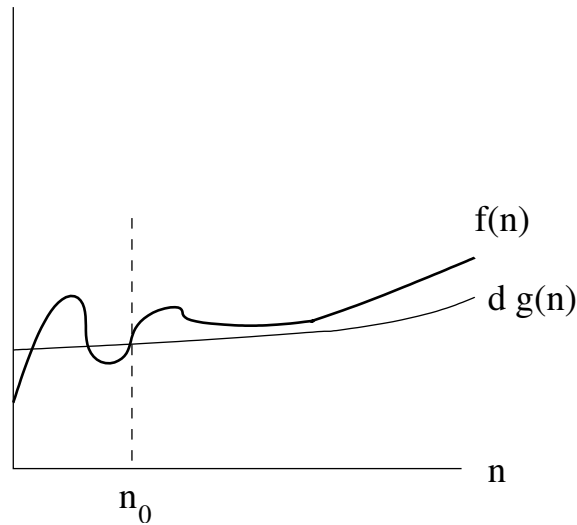
- \mathcal{O} -merkintää käytetään **ylärajan** esittämiseen
 - jos jonkin algoritmin pahimman tapauksen vaativuus on esim. $\mathcal{O}(n^3)$ ei se välttämättä tarkoita että pahin tapaus toimii ajassa dn^3 jollekin d , vaan että algoritmi ei toimi ainakaan huonommin kuin ajassa dn^3
 - Esim: Koska $n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

- Voidaan esittää vaatavuudelle myös **alaraja**, ja tällöin on käytössä merkintä Ω
- $f = \Omega(g)$ jos on olemassa vakiot $d > 0$ ja n_0 , siten että kaikilla $n \geq n_0$ ehto $dg(n) \leq f(n)$ on voimassa
- Esim: $2n^2 + 3n + 7 = \Omega(n^2)$ sillä kun $n > 0$, on $2n^2 + 3n + 7 \geq 2n^2$ jos $n \geq 1$.
Eli voidaan valita määritelmän mukaisiksi vakioiksi $d = 2$ ja $n_0 = 1$
 n^2 on siis funktion $2n^2 + 3n + 7$ ala- ja yläraja
- Jos algoritmin ylä- ja alaraja vaatavuudelle on sama, eli kyseessä on tarkka arvio, voidaan käyttää omaa merkintätapa:
 g on sekä ylä- että alaraja f :lle, merkitään $f = \Theta(g)$, jos ja vain jos $f = \mathcal{O}(g)$ ja $f = \Omega(g)$
 - nyt olemassa vakiot d_1 , d_2 ja n_0 , siten että kaikilla $n \geq n_0$ pätee $d_1g(n) \leq f(n) \leq d_2g(n)$
 - f käyttäytyy oleellisesti samoin kuin g , toisin sanoen f :llä sama **asymptoottinen** kasvunopeus kuin g :llä
- Seuraavat kuvat valaisevat asymptoottisten merkintöjen eroja

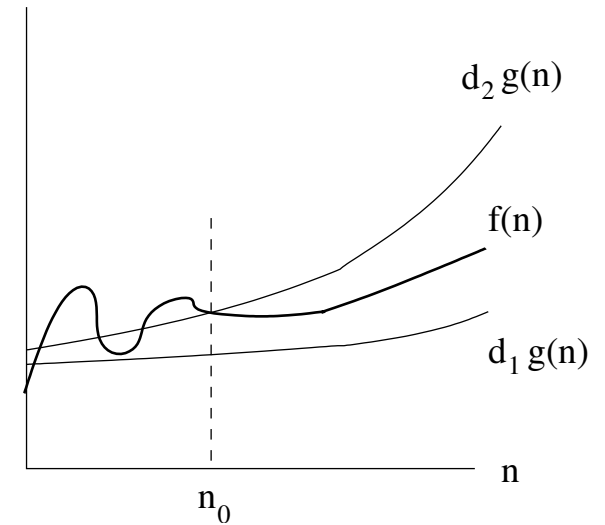
$$f(n) = O(g(n))$$



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



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



- Valtaosassa algoritmikirjallisuutta algoritmien vaatavuudet ilmaistaan "isoon", eli merkinnän O avulla
- Kuten pari sivua sitten todettiin, O ilmaisee oikeastaan ainoastaan funktion kasvun asymptoottisen ylärajan, ja merkintää voidaan "väärinkäyttää", esim. sanomalla että **lisäysjärjestämisen** aikavaativuus on $O(n^5)$
- Tällaisen harhaanjohtavan käytön voi ehkäistä käyttämällä tarkempaa merkintää Θ , joka siis rajaa funktion sekä ylä- että alapuolelta. Cormenin kirjassa on tapana käyttää pääosin Θ -merkintää

- Näytämme vielä esimerkin vuoksi, miten voidaan todistaa suuruusluokkaväittämä vääräksi
- Yleensä helpoin tapa osoittaa jotain vääräksi, on ottaa esimerkki, jossa väite ei päde. Mutta jos haluamme osoittaa, että $n^2 = \mathcal{O}(n)$ ei päde, meidän pitää osoittaa että ei ole olemassa mitään sopivia vakioita d ja n_0 , niin että haluttu ehto toimisi.
- Negaatiota sille, että "on olemassa $A: B$ pätee" on "kaikille $A: B$ ei päde"
- Eli

$$\neg(\exists d, n_0 > 0 : \forall n \geq n_0 : f(n) \leq dg(n)) \Leftrightarrow (\forall d, n_0 > 0 : \exists n \geq n_0 : \neg(f(n) \leq dg(n)))$$
- Voimme nyt todistaa, että $n^2 = \mathcal{O}(n)$ ei päde.

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

Aika- ja tilavaativuusanalyysi käytännössä

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

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

- Tarkastellaan seuraavaa metodia:

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

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

- Merkitään S_{i-j} :llä riviltä i riville j muodostuvaa osaa koodista.
- Syöte, sijoitus ja vertailukäskyjen aikavaativuus vakio eli $\mathcal{O}(1)$. Peräkkäin suoritettavien käskyjen vaativuuden säännön perusteella S_{1-4} ja S_{11-12} aikavaativuus myös $\mathcal{O}(1)$
Huomioi, että myös taulukon käsittely on vakioaikaista sillä sijoitus tapahtuu taulukon pituudesta riippumatta aina ensimmäiseen ja viimeiseen paikkaan
- Rivin 5 ehtolauseessa on kaksi vakioaikaista testiä, joten ehtojen testaus vakioaikainen. Ehdon toteutuessa suoritetaan vakioaikainen operaatio. Jos rivin 5 ehto epätyösi, suoritetaan rivi 7, joka myös selvästi onnistuu vakioajassa. Eli S_{5-10} aikavaativuus $\mathcal{O}(1)$.
- On siis todettu, että metodi koostuu kolmesta peräkkäin suoritettavasta osasta S_{1-4} , S_{5-10} ja S_{11-12} joiden kaikkien aikavaativuus on $\mathcal{O}(1)$
Soveltamalla peräkkäisten käskyjen sääntöä, saadaan 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

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

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

- Jos parametri on x on nolla, ei rivejä 3-6 suoriteta, eli riippuen x:n arvosta komentoja suoritetaan joko 2 tai 6 kpl
- Suoritettavien komentojen määrä pahimmassa tapauksessa on 6, eli riippumaton toisen parametrin eli taulukon A koosta
- Koska komennot ovat yksinkertaisia komentoja, on pahimmassa tapauksessa suoritettavan käskyjonon 1-6 vaativuus $\mathcal{O}(1)$

- Neljäs nyrkkisääntö, **aliohjelman suorituksen aikavaativuus**:

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

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

min(x,y,x)

```
1  if x < y and x < z
2      return x
3  elif y < z
4      return = y
5  else
6      return = z
```

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

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

```

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

```

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

- Esimerkin metodin aika- ja tilavaativuudet ovat niin ilmeisiä, että yleensä niitä ei jaksettaisi näin tarkkaan analysoida
- Edellisen esimerkin sisältämän `while`-silmukan aikavaativuus oli helppo laskea sillä `while`:n runko-osan suoritus-aika on jokaisella suorituskerralla sama
- Sisäkkäisten silmukoiden tapauksessa analysointi ei yleensä ole näin yksinkertaista, sillä sisemmän silmukan suorituskertojen määrä riippuu usein ulomman silmukan looppimuuttujasta

Esimerkki: Kuplajärjestäminen

bubble-sort(A)

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

- Invariantti:

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

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

- Sisemmän `for`-silmukan runko sisältää valinnan ja vakioajan vieviä sijoitusoperaatioita. Runko on siis vakioaikainen, eli vaativuudeltaan $\mathcal{O}(1)$
- Kuinka monta kertaa runko suoritetaan?
 - ensimmäisellä sisemmän silmukan suorituskerralla $i = n$ eli j saa arvot väliltä $1 \dots n - 1$. Runko siis suoritetaan $n - 1$ kertaa
 - Seuraavalla kerralla $i = n - 1$ eli j saa arvot väliltä $1 \dots n - 2$, joten runko suoritetaan $n - 2$ kertaa
 - Seuraavaksi $i = n - 2$ eli j saa arvot väliltä $1 \dots n - 3$, joten runko suoritetaan $n - 3$ -kertaa
 - ...
 - lopulta $i = 2$ ja j saa enää arvon 1 eli runko suoritetaan ainoastaan kerran
 - runko suoritetaan siis $1 + 2 + \dots + (n - 1) = \sum_{i=1}^n i - n = \frac{1}{2}n(n + 1) - n$ kertaa, eli aikavaativuus $\mathcal{O}(n^2)$
- Huom: Analyysi hyödyntää tuttua `summakaavaa` $1 + 2 \dots + n = \sum_{i=1}^n i = \frac{1}{2}n(n + 1)$

Esimerkki: Lisäysjärjestämisen aikavaativuus

- Algoritmi on siis seuraava:

insertion-sort(A)

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

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

- Sisemmän silmukan runko-osan aikavaativuus selvästi $\mathcal{O}(1)$
- Kuinka monta kertaa sisempi silmukka (**while**) suoritetaan? Merkitään seuraavassa taulukon A pituutta n :llä
- Toisin kuin kuplajärjestämisessä, sisemmän silmukan suorituskertojen määrä ei ole sama jokaisella syötteellä
- Jos taulukko on valmiina järjestyksessä, ei sisempää toistoa (**while**) suoriteta kertaakaan, eli ulomman toiston (**for**) runko-osa vie siinä tapauksessa vakiomäärän aikaa
eli parhaassa tapauksessa algoritmi toimii kuten sisempää **while**:ä ei olisi ja **for**:in $\mathcal{O}(1)$ runko suoritetaan $n - 1$ kertaa
- **Lisäysjärjestämisen parhaan tapauksen aikavaativuus** siis $\mathcal{O}(n)$
- Yleensä parhaan tapauksen aikavaatimus ei ole kovin kiinnostava
- Järjestämisalgoritmit muodostavat joskus poikkeuksen, sillä tietyissä tilanteissa saattaa olla niin, että järjestämisalgoritmin syöte on aina lähes valmiina järjestyksessä
- Tällöin **lisäysjärjestäminen** toimii aina erittäin nopeasti vaikka sen pahimman tapauksen aikavaatimus ei olekaan kilpailukykyinen parhaiden algoritmien kanssa

- Pahimmassa tapauksessa (taulukko käänteisessä järjestyksessä) sisempi **while** joutuu viemään alkion $A[j]$ joka kerta paikkaan $A[1]$ asti
- Sisemmän silmukan suorituskertojen määrä riippuu ulomman silmukan muuttujan j arvosta, tarkemmin ottaen sisempi silmukka suoritetaan pahimmassa tapauksessa joka kerta $j - 1$ kertaa
 - aluksi $j = 2$ eli sisemmän silmukan runko suoritetaan kerran
 - sitten $j = 3$ ja sisemmän silmukan runko suoritetaan 2 kertaa
 - ...
 - lopulta $j = n$ ja sisemmän silmukan runko suoritetaan $n - 1$ kertaa
 - sisemmän silmukan runko siis suoritetaan pahimmassa tapauksessa yhteensä $1 + 2 + \dots + n - 1 = \sum_{i=1}^n i - n = \frac{1}{2}n(n + 1) - n$ kertaa
- **Lisäysjärjestämisen** **pahimman tapauksen aikavaativuus** siis $\mathcal{O}(n^2)$
- Valitettavasti algoritmi toimii myös keskimäärin ajassa $\mathcal{O}(n^2)$, eli ei ole normaalilla syötteillä kilpailukykyinen myöhemmin kurssilla esiteltävien algoritmien (keko-, lomitusta- ja pikajärjestäminen) kanssa
- Algoritmi käyttää kolmea apumuuttujaa, eli tilavaativuus vakio $\mathcal{O}(1)$

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

```

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

```

- Tämä on tietenkin typerä tapa taulukon käänteiseen tulostamiseen, mutta saa kelvata yksinkertaisena esimerkkinä rekursiosta
- Pelkän rungon aikavaativuus vakio $\mathcal{O}(1)$
- Jos taulukon koko on n tehdään rekursiivisia kutsuja $n + 1$, eli algoritmin suorituksen aikavaativuus on $\mathcal{O}(n)$

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

- Mikä on algoritmin tilavaativuus? Näyttää siltä, että koodi ei käytä yhtään apumuuttujaa, eli onko tilavaativuus vakio eli $\mathcal{O}(1)$?
- Kuten jo aiemmin todettiin, metodikutsu varaa suoritusajasta pinosta tilaa parametreille ja metodin paikallisille muuttujille (tekninen termi tälle varaukselle on aktivaatitietue). Yhden metodikutsun pinosta varaaman tilan koko on vakio eli $\mathcal{O}(1)$
- Kun `recPrint(A,1)` suorittaa rekursiivisen kutsun `recPrint(A,2)`, jää sen itse varaama tila vielä pinoon sillä metodikutsu ei ole ohi ennen kuin rekursiivisesta kutsusta palataan

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

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

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

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

Rekursioyhtälöt

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

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

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

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

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

- Aikavaativuus syötteen koolla n saadaan laskemalla rekursioyhtälö auki:
 Rekursioyhtälön mukaan $T(n) = T(n-1) + c$. Soveltamalla edelleen rekursioyhtälöä tiedetään, että $T(n-1) = T(n-2) + c$, ja nämä yhdistämällä saadaan $T(n) = T(n-2) + 2c$, eli

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

- Eli päädytään samaan lopputulokseen kun aiemmassa laskelmassa
- Rekursioyhtälö ratkesi varsin mukavasti aukilaskemalla. Näemme muutaman sivun päästä toisen esimerkin, jossa vastaava tekniikka toimii
- Aina ei näin ole. Kurssilla [Design and Analysis of Algorithms](#) opitaan ratkaisemaan rekursioyhtälöitä, joihin käyttämämme menetelmä ei toimi
- Tällä kurssilla on hyvä osata määritellä yksinkertaisia rekursioyhtälöitä, mutta monimutkaisten rekursioyhtälöiden ratkaiseminen ei kuulu kurssiin
- Tarkastellaan seuraavaksi [Ohjelmoinnin perusteista](#) tuttua [binäärihakua](#) (engl. binary search).
- Binäärihaun ja monien muiden "asioiden puolittamiseen" perustuvien algoritmien analyysissä tarvitaan logaritmeja

Binäärihaku

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

binary-search(A,x)

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

- Invariantti:
 - Jos x on taulukossa niin se on välillä $A[\textit{vasen}, \textit{oikea}]$ ja $\textit{found} = \textit{true}$ jos ja vain jos x löytyi jo
- Todistetaan, että algoritmi toimii oikein
 - invariantti tosi alussa sillä $\textit{vasen} = 1$, $\textit{oikea} = n$ ja $\textit{found} = \textit{false}$
 - pysyy selvästi totena toistolauseen suorituksessa:
 - jos x löytyy, asetetaan $\textit{found} = \textit{true}$
 - jos $A[\textit{keski}] > x$ niin myös kaikilla $j > \textit{keski}$ on $A[j] > x$ eli ei tarvitse etsiä kohdan \textit{keski} takaa ja voidaan asettaa $\textit{oikea} = \textit{keski} - 1$
 - **else**-haara vastaavasti
 - lopuksi joko $\textit{vasen} > \textit{oikea}$ ja $\textit{found} = \textit{false}$ eli etsittyä ei löytynyt, tai $x = A[\textit{keski}]$ ja $\textit{found} = \textit{true}$
- Toistolause terminoi koska jokaisella toistolla joko hakualue pienenee tai etsitty alkio löytyy

Binäärihaun aikavaativuus

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

Binäärihaun aikavaativuus rekursioyhtälön avulla

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

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

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

- Oletetaan yksinkertaisuuden vuoksi että taulukon A pituus n on jokin kahden potenssi, eli $n = 2^j$ jollain kokonaisluvulla j
ottamalla tästä kaksikantainen logaritmi molemmilta puolilta, saadaan $j = \log_2 n$
Tämä perusteltiin juuri muutama sivu sitten logaritmiselityksen yhteydessä ja on siis oikeastaan sama asia kuin logaritmin määritelmä.
- Olkoon c jokin tarpeeksi suuri vakio. Korvataan $\mathcal{O}(1)$ edellisen sivun rekursioyhtälössä tällä vakiolla:

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

- Aikavaativuus syötteen koolla n saadaan laskemalla rekursioyhtälö auki:
 Rekursioyhtälön mukaan $T(n) = T(n/2) + c$. Soveltamalla edelleen rekursioyhtälöä tiedetään, että $T(n/2) = T(n/4) + c$, ja nämä yhdistämällä saadaan $T(n) = T(n/4) + c + c$, eli

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

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

- Koska logaritmin kantaluvulla ei ole merkitystä kertaluokkien suhteen (ks. logaritmikertaussivu), merkittiin aikavaativuudeksi lopulta $\mathcal{O}(\log n)$
- Yksinkertaistimme analyysiä olettamalla että taulukon pituus on jokin kahden potenssi, eli $n = 2^j$, toinen yksinkertaistus jonka teimme oli rekursioyhtälössä missä emme ottaneet huomioon, että oikeasti taulukko ei jakaudu aina täsmälleen puoliksi
- Tekemämme yksinkertaistukset eivät kuitenkaan vaikuta vaativuusanalyysin lopputulokseen

Vaativuusluokkien luonnehdintaa

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

$O(1)$ - 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 kuuluu 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

$O(\log n)$ - 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 kuinka 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 algoritmit ovat todella nopeita: vaikka luku n olisi suurikin, sitä ei tarvitse puolittaa kovin monta kertaa, ennen kuin tulos alittaa jo luvun 1
- Esim. jos binäärihauille annetaan taulukko, jossa on miljoona lukua, algoritmi tarvitsee sen käsittelyyn vain noin 20 askelta, koska $\log_2 1000000$ on noin 20
- Törmäämme jatkossa algoritmeihin joiden tilavaativuus on logaritminen. Koska logaritmi kasvaa hyvin hitaasti, tätä voidaan pitää hyvänä tilavaativuutena
- Muistutus: logaritmien kantaluvuilla ei ole väliä O -analyysissä

$O(n)$ - lineaarinen algoritmi

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

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

- Lineaarisen algoritmin ajankäyttö kasvaa samassa suhteessa kuin käsiteltävän tiedon määrä
- Algoritmi on lineaarinen myös jos se käy syötteen läpi aina esim. kolmeen kertaan tai aina puolet tiedoista
- Esim. seuraava on lineaarinen kahdesta `for`:ista huolimatta koska ulommainen `for` suoritetaan 3 kertaa riippumatta syötteen koosta

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

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

$O(n^2)$ - neliöllinen algoritmi

- Neliöllinen algoritmi sisältää usein kaksi sisäkkäistä `for`-silmukkaa:

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

- Neliöllinen algoritmi voi käydä läpi kaikki parit, jotka voidaan muodostaa annetuista tiedoista
- Algoritmi voi siis verrata jokaista tietoa jokaiseen tietoon. Tällöin ensimmäinen silmukka valitsee ensimmäisen tiedon ja toinen silmukka valitsee toisen tiedon
- Esimerkki: Jos taulukossa on joukko lukuja, on olemassa neliöllinen algoritmi, joka tarkistaa, onko taulukossa kahta samaa lukua
Algoritmin ensimmäinen silmukka käy läpi taulukossa olevat luvut ja toinen silmukka tarkistaa, esiintyykö käsiteltävä luku jossain toisessa taulukon kohdassa
- Neliöllisen algoritmin molempia silmukoita ei välttämättä toisteta n kertaa, kuten järjestämisalgoritmien yhteydessä huomattiin. Aikavaativuusanalyysissä on oltava tällöin tarkkana ja yleensä sovellettava sopivaa summakaavaa

$O(n^3)$ - kuutiollinen algoritmi

- Kuutiollinen algoritmi sisältää usein kolme sisäkkäistä `for`-silmukkaa:

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

- Vastaavasti kuin neliöllinen algoritmi voi käydä läpi kaikki parit, kuutiollinen algoritmi voi käydä läpi kaikki kolmikot, jotka voidaan muodostaa annetuista tiedoista
- Esimerkki: Jos taulukossa on joukko lukuja, on olemassa kuutiollinen algoritmi, joka tarkistaa, voiko taulukosta valita luvut a, b ja c niin, että $a + b = c$
Ensimmäinen silmukka valitsee luvun a , toinen silmukka valitsee luvun b ja kolmas silmukka valitsee luvun c . Silmukoiden sisällä tarkistetaan, päteekö todella $a + b = c$

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

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

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

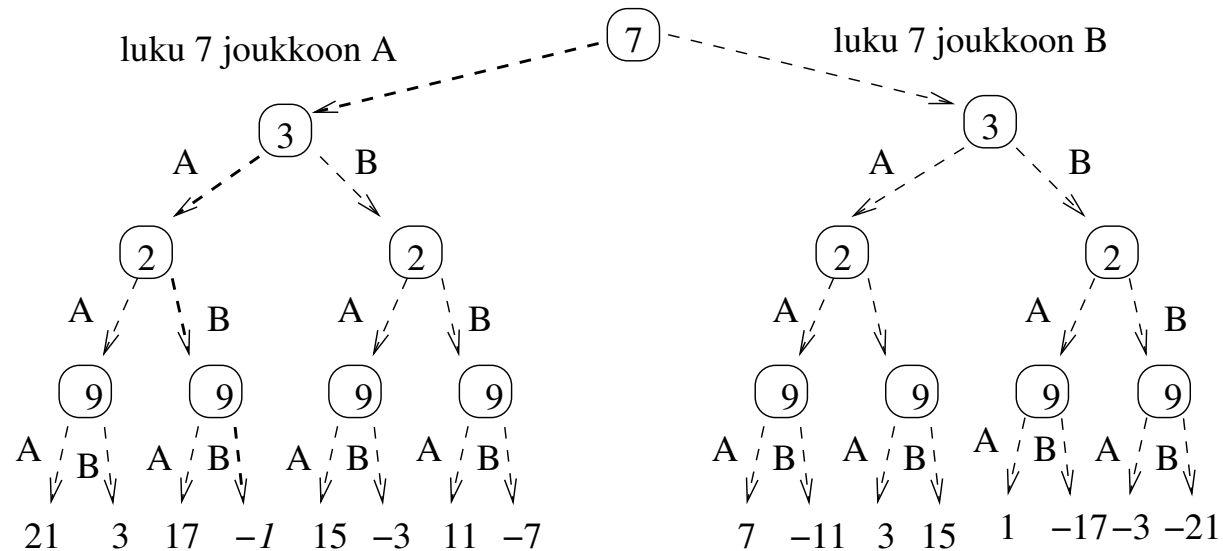
Algoritmi haarautuu joka luvun kohdalla kahden mahdollisuuden mukaan: joko luku menee ryhmään A tai sitten se menee ryhmään B .

Algoritmi pitää kirjaa ryhmien summista sekä pienimmän erotuksen tuottavista jaoista.

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

- Seuraavalla sivulla oleva esimerkki valottaa algoritmin toimintaa

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



Kasvunopeudet

- Jos käsiteltävien tietojen määrä **kasvaa yhdellä**, muut esitellyt algoritmityyppit 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 6-10:
 - ymmärtää miksi algoritmi toimii oikein
 - ymmärtää miten paljon algoritmi käyttää resursseja
 - osaa soveltaa algoritmia epätriviaalilla tavalla
 - osaa todistaa matemaattisesti algoritmin vaativuuden
 - osaa todistaa matemaattisesti algoritmin oikeaksi
- Algoritmien mielekkään matemaattisen analysoinnin ehdoton edellytys on kuitenkin hyvä ymmärrys algoritmin toiminnasta
Matemaattinen todistus usein ainoastaan tuo täsmällisellä tavalla esiin sen mikä jo tiedetään ymmärryksen tasolla
- On myös olemassa algoritmeja, joista on vaikea nähdä että ne toimivat oikein ja tarvitaan matemaattinen todistus oikeellisuudesta vakuuttumiseksi

Kertauksena algoritmien aikavaativuusanalyysin nyrkkisäännöt

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

3. Järjestäminen

- Osaamme jo muutamia $\mathcal{O}(n^2)$ ajassa toimivia menetelmiä jotka järjestävät n lukua suuruusjärjestykseen
 - vaihtojärjestäminen (Ohjelmoinnin perusteet)
 - lisäysjärjestäminen eli insertion sort (luku 1, sivu 33)
 - kuplajärjestäminen eli bubble sort (luku 1, sivu 70)
- Järjestäminen (engl. sorting), jota kutsutaan myös lajitteluksi, on klassinen tietojenkäsittelyongelma, jota on analysoitu runsaasti
- Järjestäminen on myös hyvä perusesimerkki, jonka yhteydessä voidaan esitellä
 - hajota ja hallitse -menetelmä algoritminsuunnittelussa
 - aikavaativuuden pahin vs. keskimääräinen tapaus

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

Myöhemmin tulee vastaan vielä

- [kekojärjestäminen](#)
- Koska n voi olla hyvin suuri, vaativuusluokkien $\mathcal{O}(n^2)$ ja $\mathcal{O}(n \log n)$ ero on merkittävä.
- Jos järjestämisalgoritmin toiminta perustuu alkioiden vertailuun, algoritmi ei voi toimia nopeammin kuin $\mathcal{O}(n \log n)$. Palaamme tähän
- Järjestämisessä voidaan päästä lineaariseen aikaan tietyissä erikoistapauksissa. Algoritmi ei silloin voi vertailla alkioita vaan toiminta perustuu esim. taulukossa olevien lukujen laskemiseen: [laskemisjärjestäminen](#)

Lomitusjärjestäminen

Perusajatuksena on seuraava menetelmä järjestää (mahdollisesti vajaa) korttipakka:

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

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

- **Lomitusjärjestäminen** perustuu **hajota-ja-hallitse** (engl. divide-and-conquer) -tekniikkaan:
 - **hajotetaan** ongelma pienempiin osaongelmiin
 - **hallitaan**, eli ratkaistaan osaongelmat rekursiivisesti
 - **yhdistetään** osaratkaisut siten että saadaan ratkaisu koko ongelmalle
- Taulukko $A[1, n]$ järjestetään kutsumalla **merge-sort**($A, 1, n$):

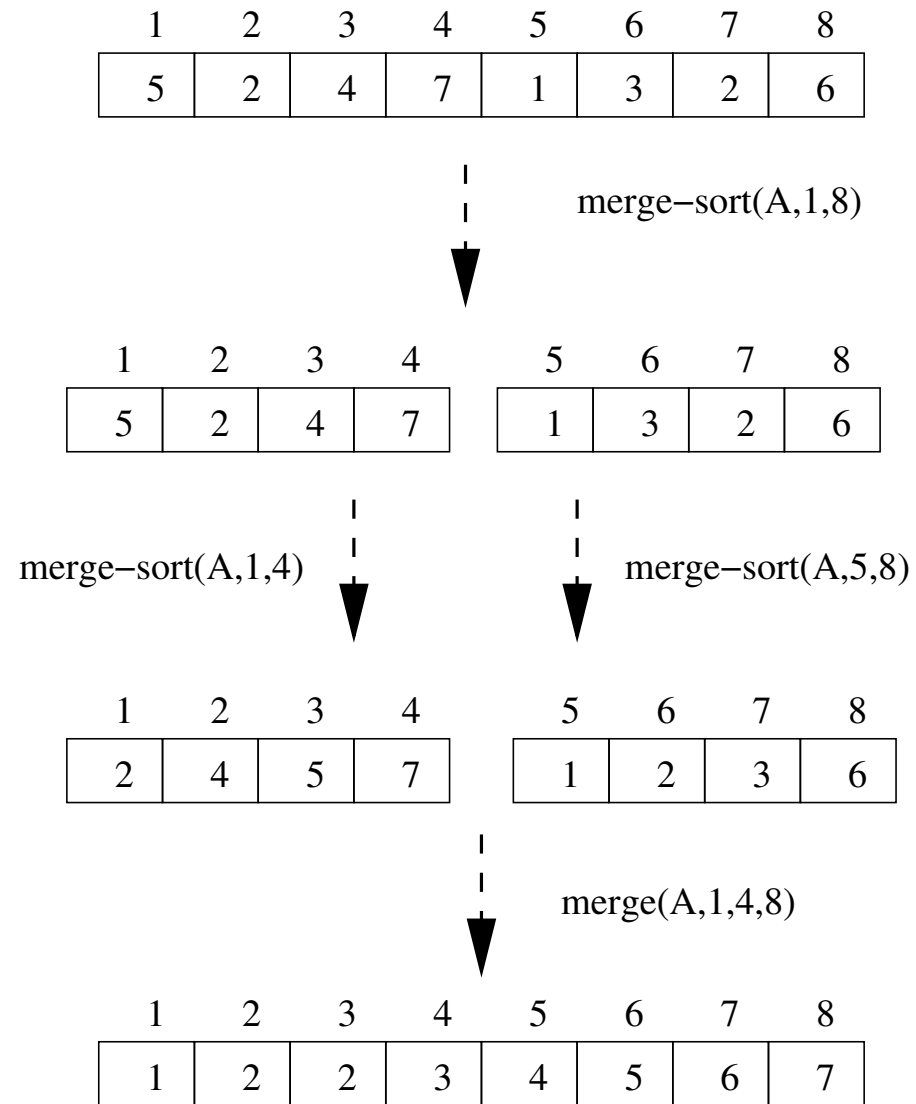
merge-sort($A, \text{vasen}, \text{oikea}$)

```
1  if vasen < oikea
2      keski = [(vasen + oikea)/2]
3      merge-sort( $A, \text{vasen}, \text{keski}$ )
4      merge-sort( $A, \text{keski}+1, \text{oikea}$ )
5      merge( $A, \text{vasen}, \text{keski}, \text{oikea}$ )
```


- Toimintaidea

- operaation tehtävänä on järjestää taulukon A osa $A[\textit{vasen}, \textit{oikea}]$
- jos järjestettävän osan pituus on korkeintaan 1 (eli $\textit{vasen} \geq \textit{oikea}$), 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 $A[\textit{vasen}, \textit{keski}]$ ja $A[\textit{keski} + 1, \textit{oikea}]$ järjestetään kutsumalla niille **merge-sort**-operaatiota rekursiivisesti
- riville 5 tultaessa siis taulukon osat $A[\textit{vasen}, \textit{keski}]$ ja $A[\textit{keski} + 1, \textit{oikea}]$ ovat järjestyksessä
- rivillä 5 kutsutaan operaatiota **merge** joka "lomittaa" osaratkaisut siten että $A[\textit{vasen}, \textit{oikea}]$ saadaan järjestykseen

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



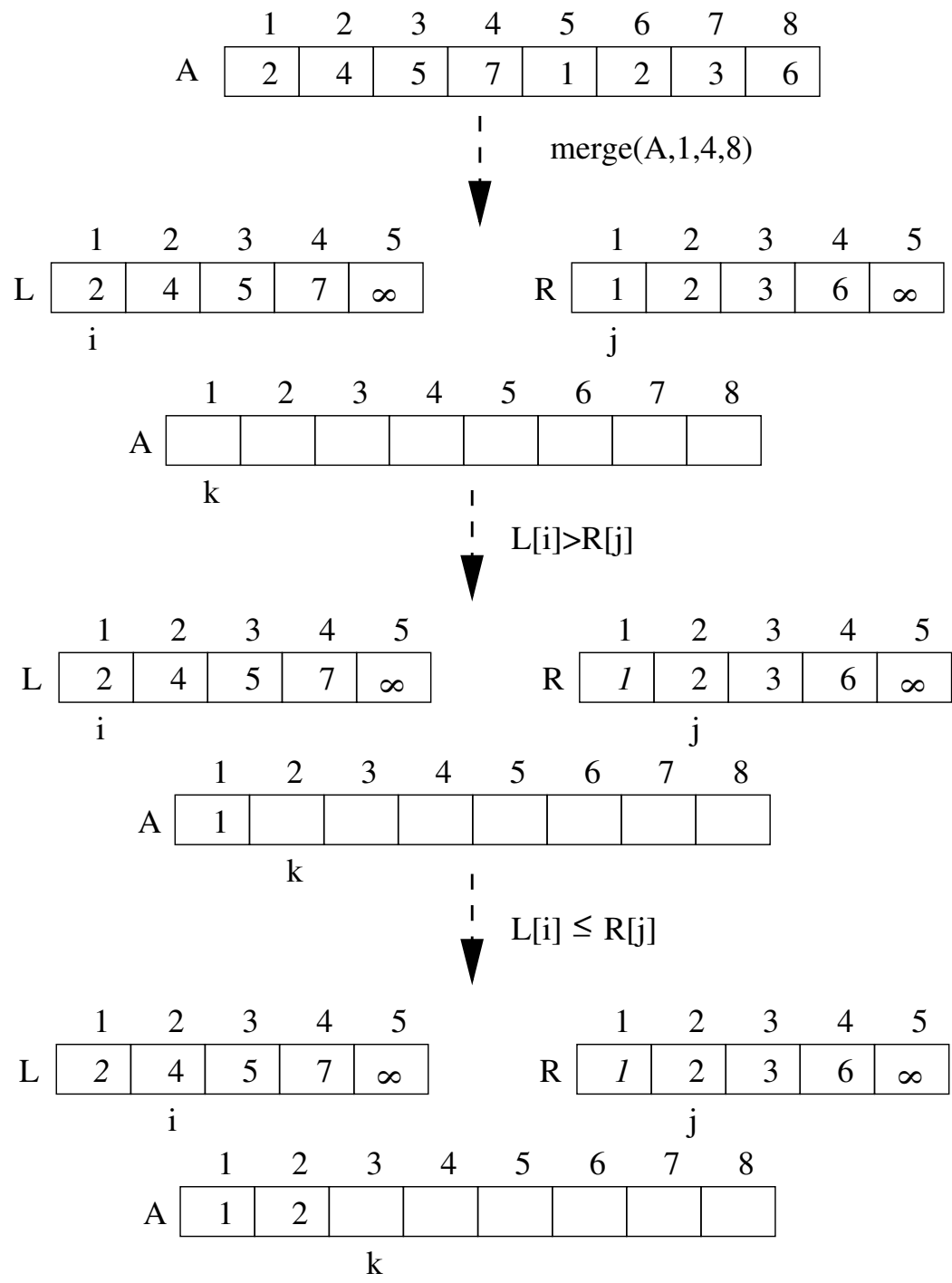
- Järjestyksessä olevien taulukon osien $A[\textit{vasen}, \textit{keski}]$ ja $A[\textit{keski} + 1, \textit{oikea}]$ lomittaminen järjestykseen on helppo tehdä:
 - kopioidaan $A[\textit{vasen}, \textit{keski}]$ aputaulukkoon $L[1, n_1]$
 - ja $A[\textit{keski} + 1, \textit{oikea}]$ aputaulukkoon $R[1, n_2]$
 - laitetaan paikkaan $A[\textit{vasen}]$ pienin alkioista $L[1]$ ja $R[1]$,
 - jos $L[1]$ laitettiin taulukkoon, niin paikkaan $A[\textit{vasen} + 1]$ laitetaan pienin alkioista $L[2]$ ja $R[1]$
jos taas $R[1]$ laitettiin taulukkoon, niin paikkaan $A[\textit{vasen} + 1]$ laitetaan pienin alkioista $L[1]$ ja $R[2]$
 - näin jatketaan kunnes kaikki aputaulukoiden alkiot on siirretty taulukkoon A

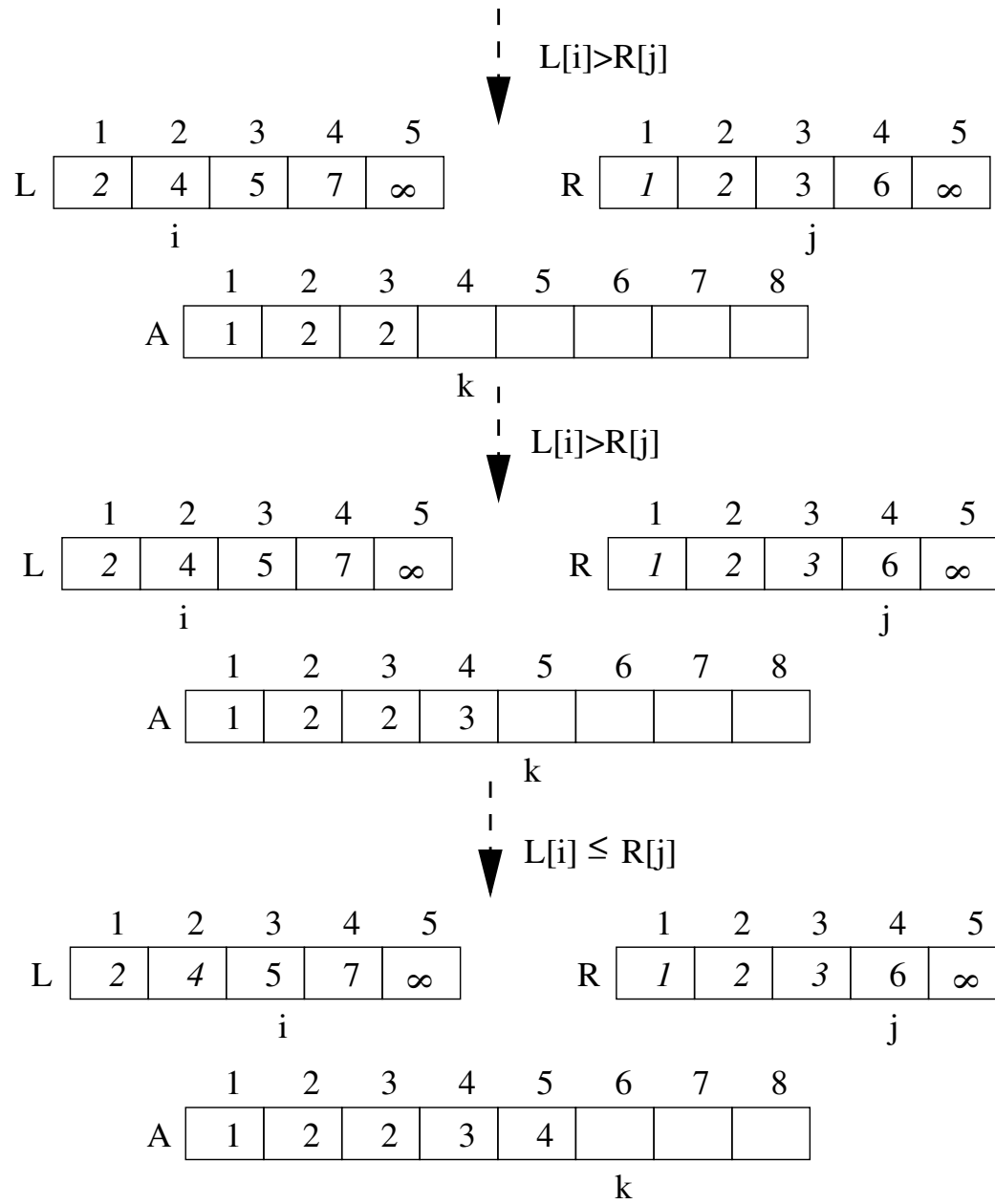
- Algoritmina

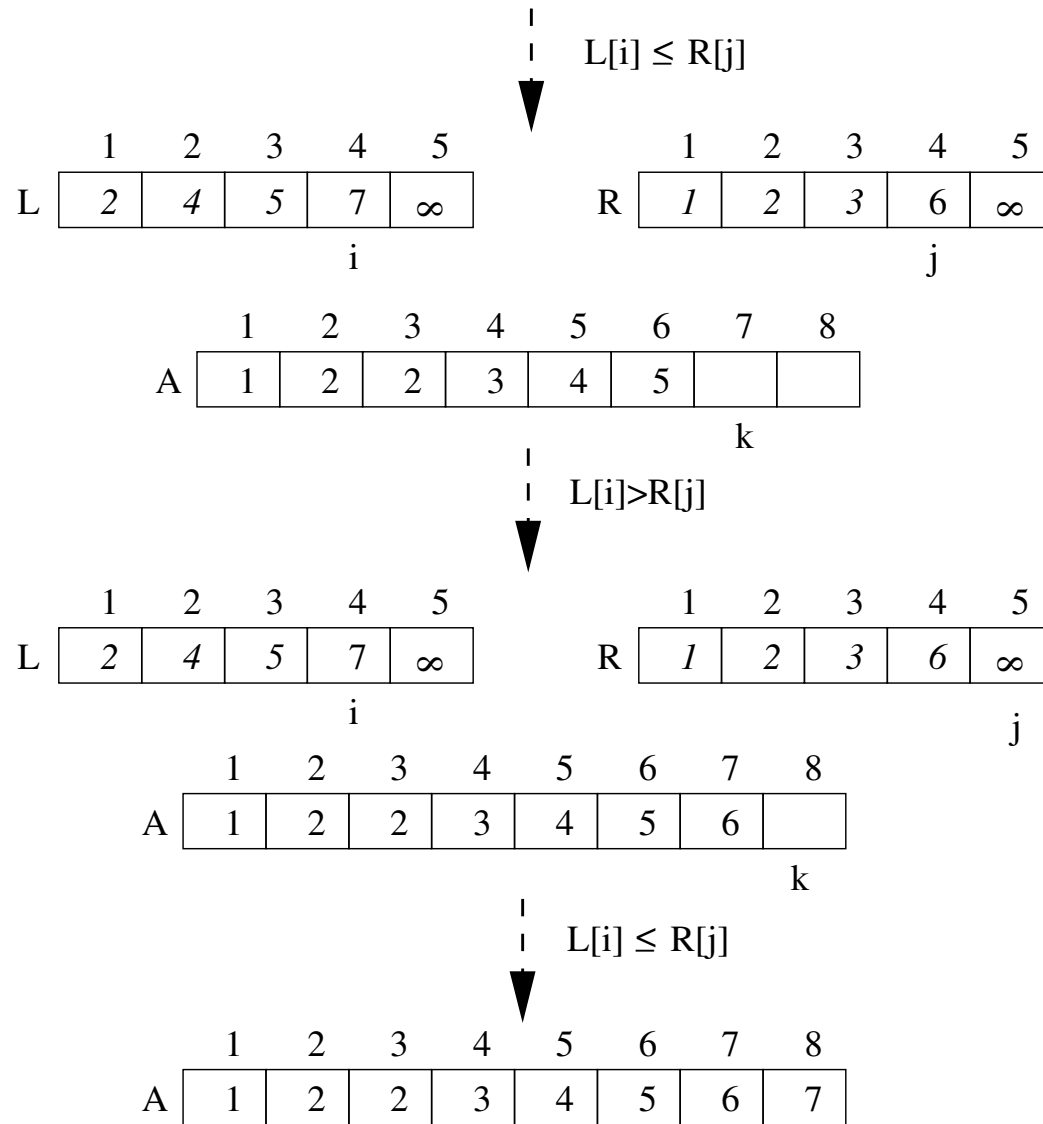
merge(A,vasen,keski,oikea)

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

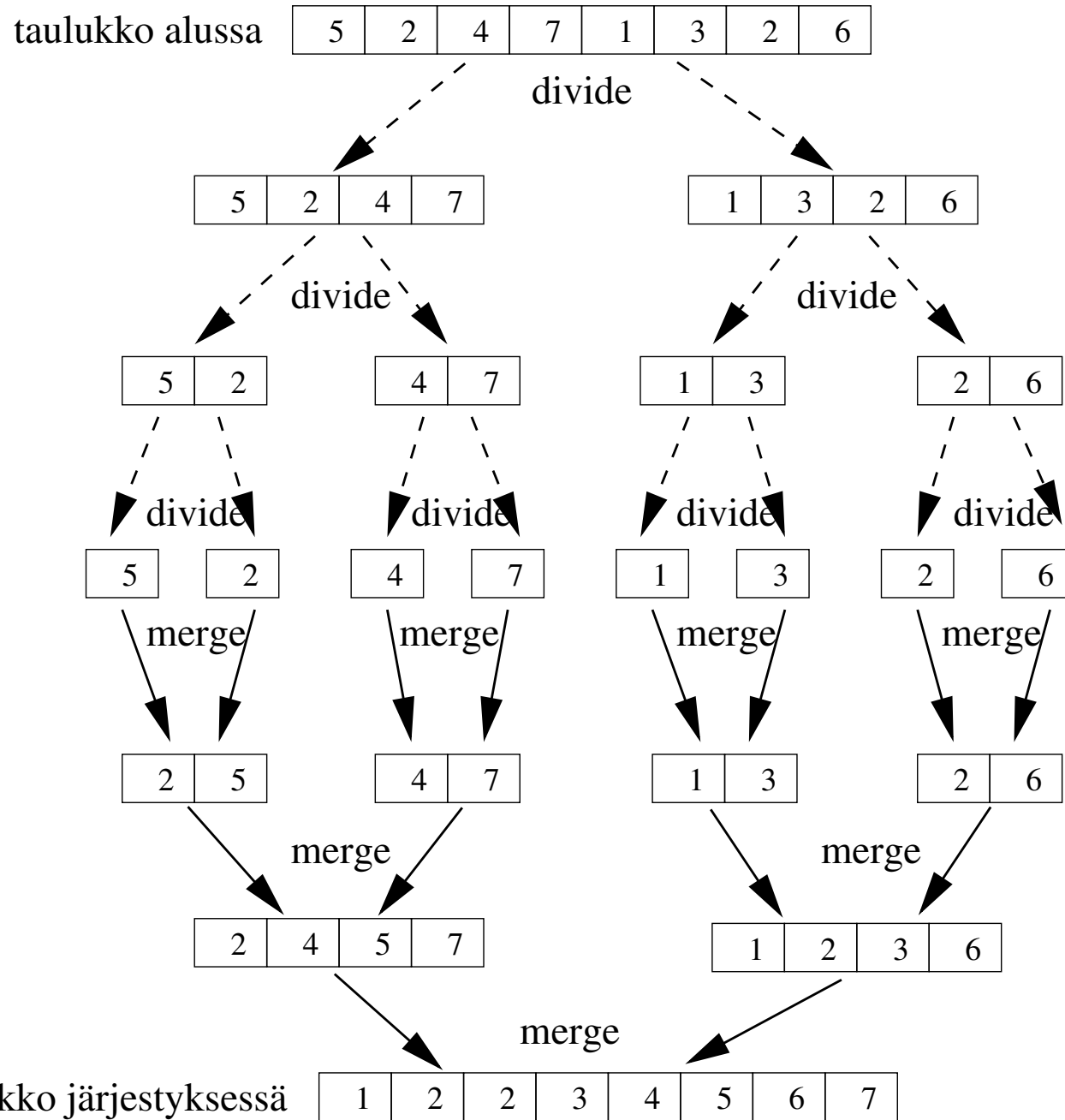
- Esimerkki **merge**-operaation toiminnasta seuraavilla sivuilla:







- Seuraavalla sivulla esimerkki koko algoritmin toiminnasta:



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

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

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

- Taulukon $A[1, n]$ **lomituserjestämisen** 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

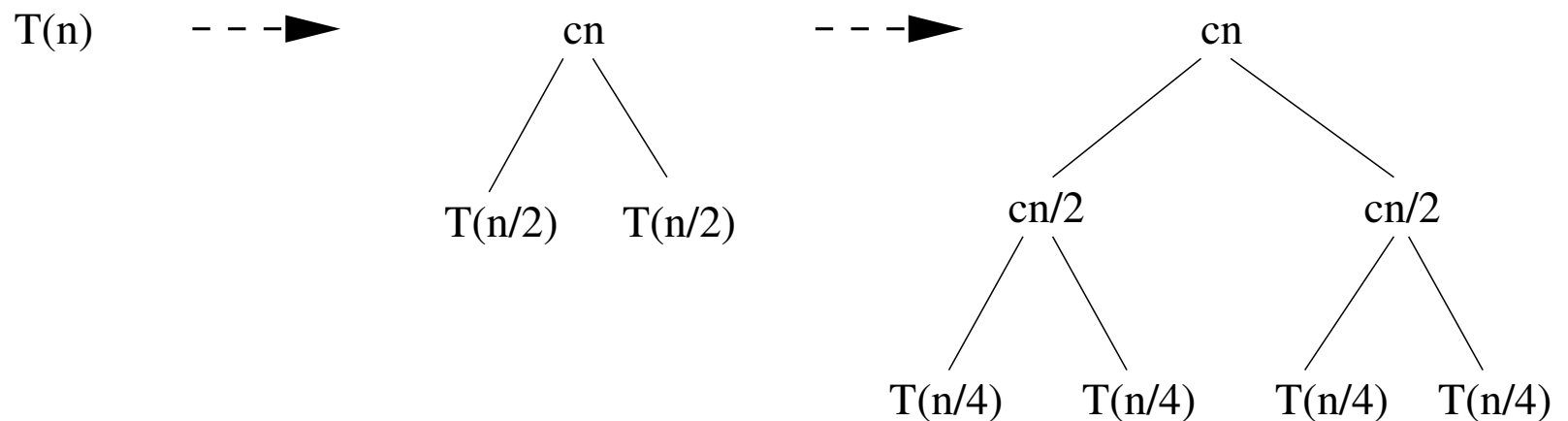
- Käytännössä molemmissa yhtälöissä \mathcal{O} :n korvaa oma vakio, valittu c on näistä vakioista suurempi

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

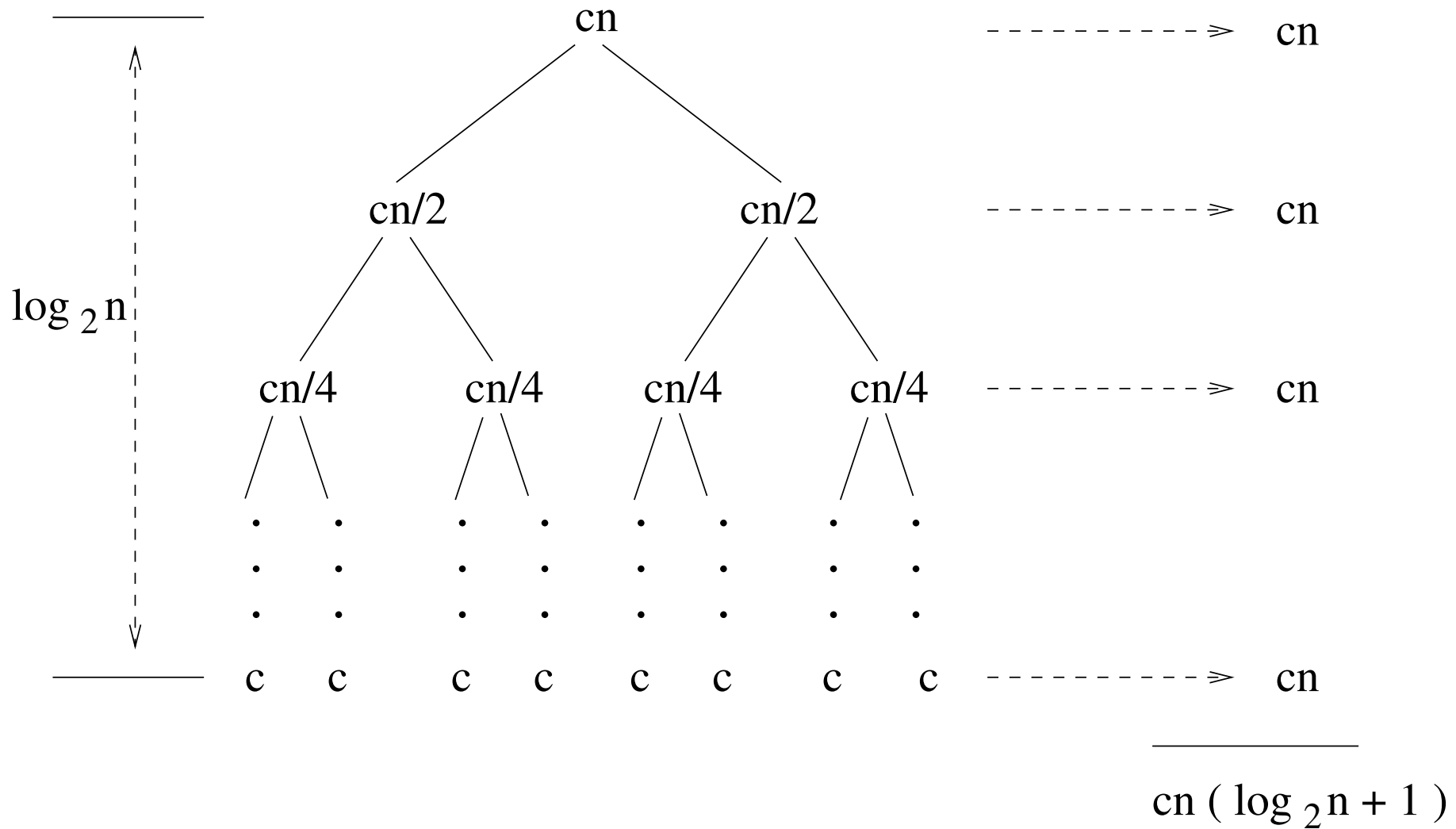
- Aletaan laskemaan auki rekursioyhtälöä:

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

- 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



- Huomaamme että rekursiopuun jokaisella tasolla olevien rekursiokutsuinstanssien yhteenlaskettu vaativuus on cn
- Yksi tapa päätellä rekursiotasojen määrä on seuraava:
Jokainen rekursiokutsu puolittaa syötteesä koon. Syötteen koko on aluksi n . Kun puolitus on tehty $\log_2 n$ kertaa, syötteen koko on enää 1. Eli mukaanlukien alin taso, jossa rekursiokutsua ei enää tapahdu, on rekursiotasoja $\log_2 n + 1$ kappaletta.
- **Lomitusjärjestämisen** aikavaativuus saadaan siis laskemalla yhteen kaikkien rekursiotasojen vaativuus, joka on $cn(\log_2 n + 1)$

- Entä jos n ei ole kakkosen potenssi?
 - silloin $2^p < n < 2^{p+1}$ jollekin positiiviselle kokonaisluvulle p
 - selvästi T on kasvava funktio, joten

$$T(n) \leq T(2^{p+1}) = c2^{p+1}(\log_2 2^{p+1} + 1)$$

- Koska $2^{p+1} = 2 \cdot 2^p < 2n$, saamme

$$T(n) < 2cn(\log_2(2n) + 1) = 2cn(\log_2 n + 2)$$

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

Iteratiivinen lomituserjestäminen

- Edellä esitetyssä versiossa rekursiiviset **merge-sort**-kutsut vain jakavat taulukkoa pienempiin osataulukoihin
- Tämä on turhaa työtä, koska tiedämme ilman muuta, mitä osataulukoita tarvitaan:
 - ensin lomitetaan $A[1]$ ja $A[2]$; sekä $A[3]$ ja $A[4]$; jne.
 - sitten lomitetaan $A[1..2]$ ja $A[3..4]$; sekä $A[5..6]$ ja $A[7..8]$; jne.
 - ...
 - lomitetaan $A[1..2^{i-1}]$ ja $A[2^{i-1} + 1..2^i]$; sekä $A[2^i + 1..2^i + 2^{i-1}]$ ja $A[2^i + 2^{i-1} + 1..2 \cdot 2^i]$; jne.
 - ...
 - lomitetaan $A[1..n/2]$ ja $A[n/2 + 1..n]$ (olettaen yksinkertaisuuden vuoksi $n = 2^p$)
- Siis lomituksen vaiheessa i lomitetaan pituutta 2^i olevia osataulukoita

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

merge-sort2(A)

```
    pituus = 1 // lomitettavien lohkojen pituus
```

```
    while pituus < A.length
```

```
        alku = 1 // lomitettavan lohkoparin alkukohta
```

```
        while alku + pituus ≤ A.length
```

```
            vasen = alku
```

```
            keski = alku + pituus - 1
```

```
            oikea = min{keski + pituus, A.length} //viimeinen lohko voi olla vajaa
```

```
            merge(A,vasen,keski,oikea)
```

```
            alku = alku + 2 * pituus
```

```
        pituus = 2 * pituus
```

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

- Järjestämisalgoritmia sanotaan **vakaaksi** (engl. stable) jos sillä on seuraava ominaisuus:
Jos kahdella eri tiedolla on sama avain, niin vakaa järjestämisalgoritmi ei muuta näiden kahden tiedon keskenäistä järjestystä
- **Lomitusjärjestäminen** on vakaa toisin kuin myöhemmin esitettävä **kekojärjestäminen** ja kohta esitettävä **pikajärjestäminen**
- Milloin järjestämisalgoritmin vakaudesta on voisi olla hyötyä?
 - Oletetaan, että järjestetään olioita, joilla on attribuutteina *etunimi*, *sukunimi* ja *ikä*
 - Järjestäminen halutaan tehdä ensisijassa sukunimen suhteen
 - Sama sukunimisten järjestyksen ratkaisee etunimen järjestys
 - Jos molemmat nimet ovat samat, ratkaisee järjestyksen ikä
- Oliot saadaan haluttuun järjestykseen seuraavasti

jarjesta(A, 1, n)

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

- Toimintaperiaate on seuraava
 - rivin 1 jälkeen oliot ovat ikäjärjestyksessä
 - rivin 2 jälkeen oliot ovat etunimen mukaisessa järjestyksessä
lomitusjärjestämisen vakaus takaa, että saman etunimen omaavat ovat edelleen iän mukaan järjestettynä
 - rivin 3 jälkeen järjestys sukunimien suhteen
koska **lomitusjärjestäminen** on vakaa, niin saman sukunimiset ovat edelleen etunimen mukaisessa järjestyksessä ja sekä saman etu- että sukunimen omaavien järjestyksen määrää ikä
- Aikavaativuus on edelleen $\mathcal{O}(n \log n)$ sillä $\mathcal{O}(n \log n)$ aikaa vievä **lomitusjärjestäminen** suoritetaan kolme kertaa peräkkäin
- Jos Javassa halutaan järjestää olioita vastaavalla tavalla, helpoimmalla selvittää määrittelemällä luokalle sopiva `compareTo`-metodi, joka määrittelee mikä on kahden luokan olion keskinäinen järjestys
- Java API:n luokasta `Collections` löytyy valmis staattinen metodi `sort`, jonka avulla voidaan järjestää esim. `ArrayList`:issa säilytettävät oliot niiden `compareTo`-metodin määrittelemään järjestykseen
- Luokasta `Arrays` löytyy vastaava metodi normaalien taulukoiden järjestämiseen

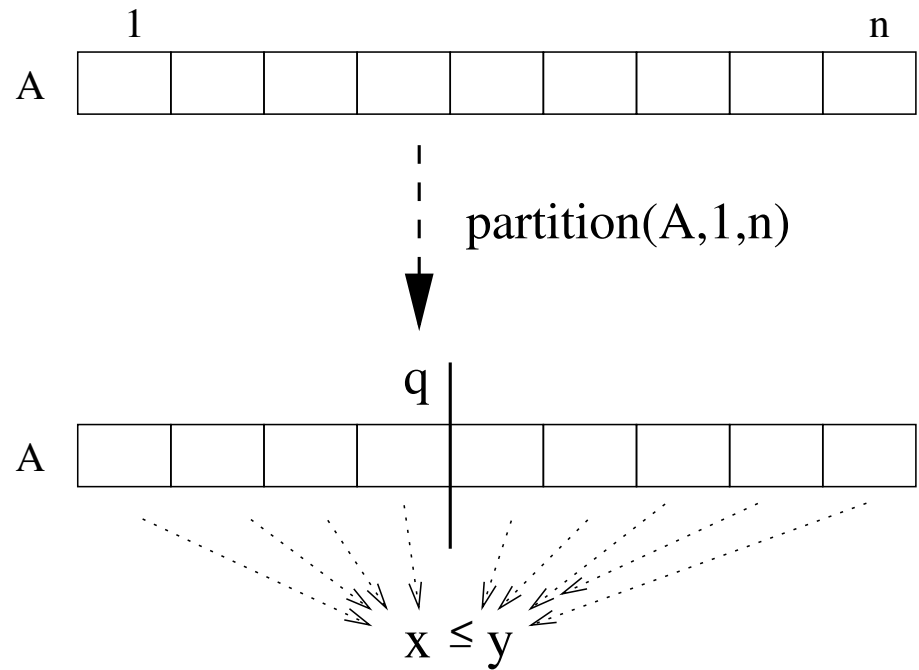
Pikajärjestäminen (Quicksort) [C.A.R. Hoare, 1962]

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

quicksort($A, \text{vasen}, \text{oikea}$)

```
1  if vasen < oikea
2      jako = partition( $A, \text{vasen}, \text{oikea}$ )
3      quicksort( $A, \text{vasen}, \text{jako}$ )
4      quicksort( $A, \text{jako}+1, \text{oikea}$ )
```

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



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

- **Pikajärjestämisen** tehokkuuden kannalta on oleellista että **partition**-operaatio toimii nopeasti (linearisessa ajassa jaettavan taulukonosan koon suhteen) ja tuottaa mahdollisimman tasaisia jakoja
- Käytetään seuraavaa **partition**-operaatiota, jossa on valittu jakoalkioksi vasemmanpuolimmaisoin alkio $A[\textit{vasen}]$
- Cormenissa (luku 7) on jakoalkiona käytetty oikeanpuolimmaisinta alkia
- Muitakin vaihtoehtoja on

partition(A,vasen,oikea)

```

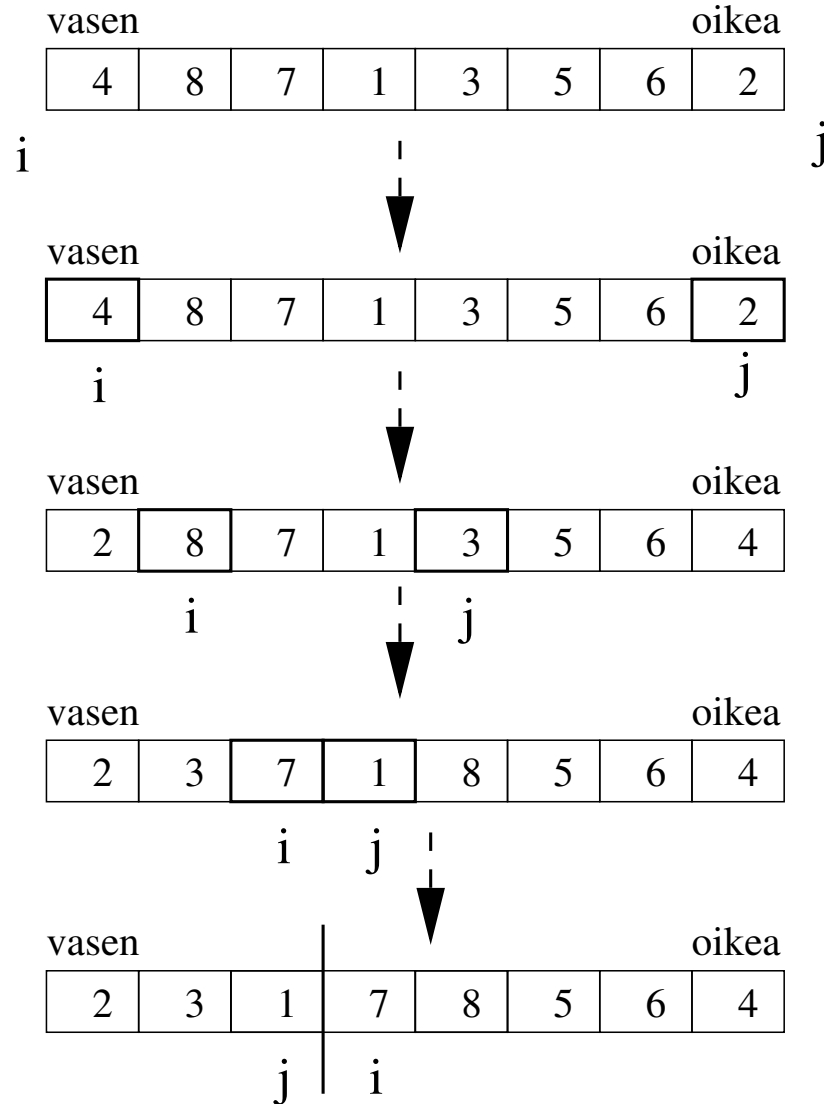
1  jakoAlkio = A[vasen]
2  i = vasen - 1
3  j = oikea + 1
4  while i < j
5      repeat j = j - 1 until A[j] ≤ jakoAlkio
6      repeat i = i + 1 until A[i] ≥ jakoAlkio
7      if i < j
8          vaihda A[i] ja A[j]
9  return j

```

- Algoritmin oikeellisuuden toteamiseksi havaitaan, että **while**-silmukan invariantti on:
 $A[k] \leq \text{jakoAlkio}$, kun $\text{vasen} \leq k \leq i$, ja $A[k] \geq \text{jakoAlkio}$, kun $j \leq k \leq \text{oikea}$
paitsi ehkä viimeisen **while**-iteraation jälkeen
- Alkutilanteessa invariantti pätee triviaalisti. Lisäksi ensimmäisellä iteraatiolla j liikkuu vasempaan, mutta pysähtyy viimeistään indeksissä vasen (jossa on arvo jakoAlkio) ja i liikkuu indeksiin vasen
- Kun **repeat** j -silmukka on suoritettu on $A[k] \geq \text{jakoAlkio}$ kun $j < k \leq \text{oikea}$ ja $A[j] \leq \text{jakoAlkio}$
- Kun **repeat** i -silmukka on suoritettu on $A[k] \leq \text{jakoAlkio}$ kun $\text{vasen} \leq k < i$ ja $A[i] \geq \text{jakoAlkio}$
- Jos silmukka ei vielä pääty, vaihdetaan $A[i]$ ja $A[j]$ ja invariantti tulee taas voimaan
- Huomaa, myös, että joka kierroksella indeksit siirtyvät vähintään yhden verran ja että kierroksen jälkeen on olemassa alkio $\leq \text{jakoAlkio}$ taulukossa $A[\text{vasen}..i]$ ja alkio $\geq \text{jakoAlkio}$ taulukossa $A[j.. \text{oikea}]$

- Tarkastelemme nyt silmukan viimeistä iteraatiota, jonka lopuksi siis $j \leq i$
- Väitämme, että j on sopiva jakokohta eli
 $A[k] \leq \text{jakoAlkio}$, kun $\text{vasen} \leq k \leq j$, ja $A[k] \geq \text{jakoAlkio}$, kun $j + 1 \leq k \leq \text{oikea}$
- Kuten edellä,
 - Kun **repeat** j -silmukka on suoritettu on $A[k] \geq \text{jakoAlkio}$ kun $j < k \leq \text{oikea}$ ja $A[j] \leq \text{jakoAlkio}$
 - Kun **repeat** i -silmukka on suoritettu on $A[k] \leq \text{jakoAlkio}$ kun $\text{vasen} \leq k < i$ ja $A[i] \geq \text{jakoAlkio}$
- Siis väitteen jälkimmäinen osa pätee ilman muuta
- Koska $i \geq j$ ja $A[k] \leq \text{jakoAlkio}$, kun $\text{vasen} \leq k < i$, niin $A[k] \leq \text{jakoAlkio}$ pätee kun $k < j$. Lisäksi $A[j] \leq \text{jakoAlkio}$, joten myös väitteen ensimmäinen osa pätee

- Esimerkki **partition**-operaation toiminnasta:



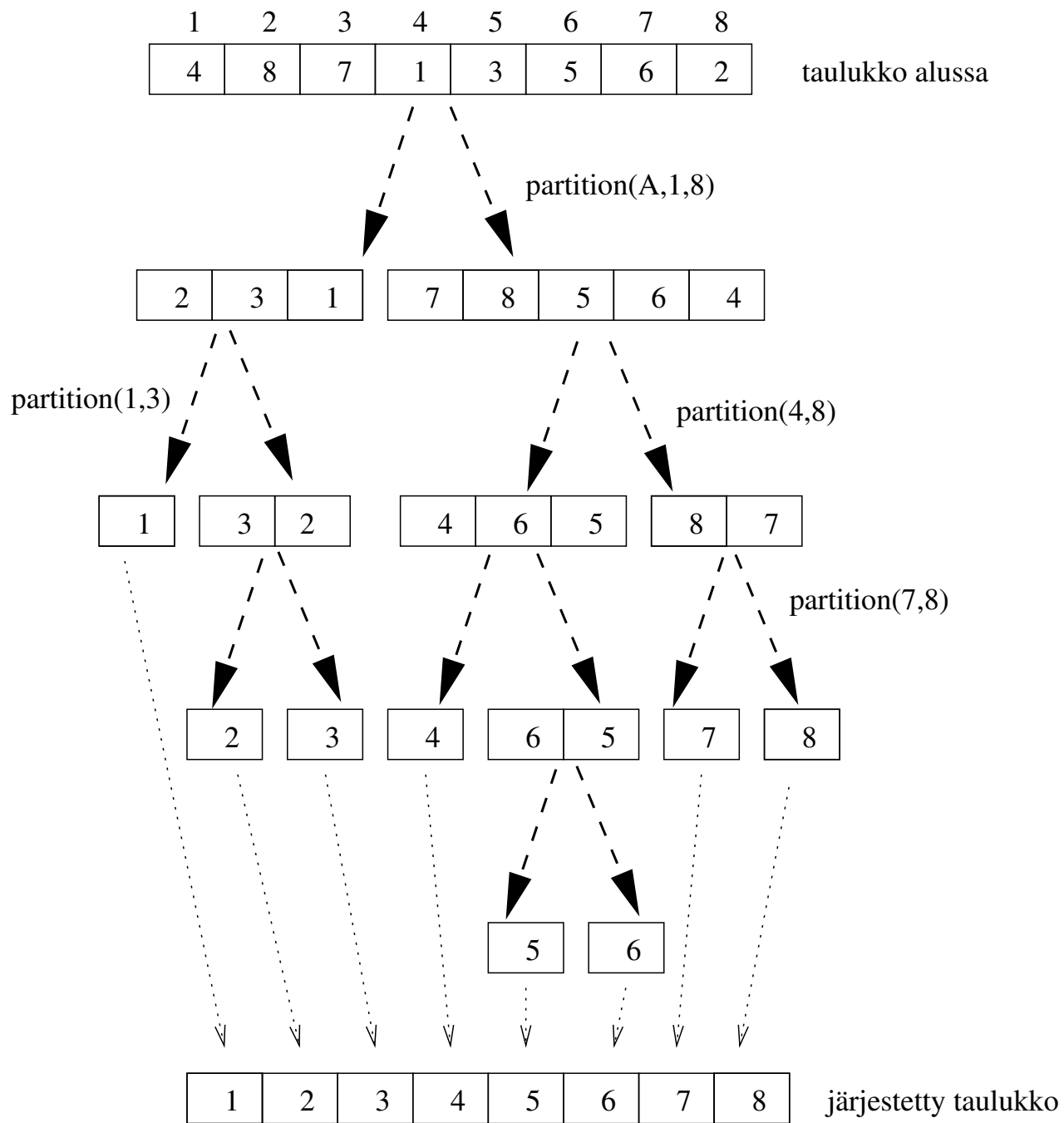
jakoalkioksi
 $A[p] = 4$

palautetaan j

- Selvästi **partition**-operaation aikavaativuus on $\mathcal{O}(k)$ suhteessa käsiteltävän taulukon pituuteen k , aputilaa ei tarvita kuin parin muuttujan verran eli tilavaativuus on $\mathcal{O}(1)$
- Esimerkissämme meillä oli melko hyvä tuuri, taulukko jakautui kohtuullisen tasan kahtia, entä jos kyseessä olisi ollut seuraava taulukko?

	vasen						oikea	
	8	1	7	4	3	5	6	2

- Entä mitä tapahtuu, jos taulukon kaikilla alkiolla on sama arvo?
- Seuraavalla sivulla esimerkki **pikajärjestämisen** toiminnasta



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

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

missä c on vakio eli termi ck kuvaa **partition**-operaation vaativuutta k :n mittaiselle taulukon osalle

- Ratkaistaan $T_w(n)$ laskemalla auki rekursioyhtälöä:

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

- Pahimmassa tapauksessa **pikajärjestäminen** toimii neliöisesti, eli on selvästi huonompi kuin **lomitus-** ja myöhemmin esitettävä **kekojärjestäminen**
- Entä parhaassa tapauksessa, missä **partition**-operaatio sattuu aina jakamaan taulukon kahteen yhtäsuureen osaan?
- Parhaan tapauksen vaativuus voidaan määritellä seuraavalla rekursioyhtälöllä:

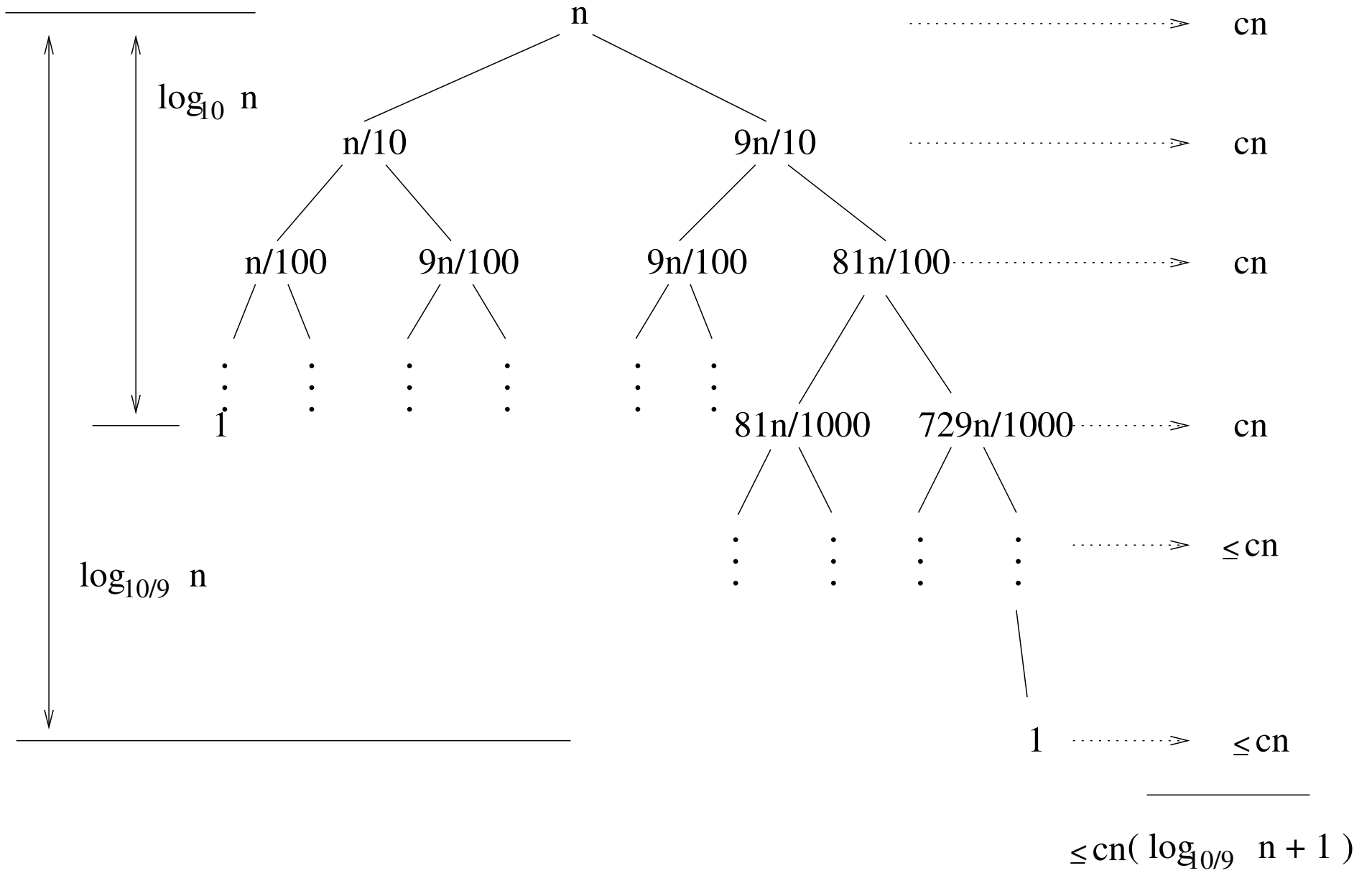
$$T_b(k) = \begin{cases} c & \text{kun } k = 1 \\ T_b(k/2) + T_b(k/2) + ck & \text{kun } k > 1 \end{cases}$$

- Yhtälö on täsmälleen sama kuin **lomitusjärjestämisen** vaativuutta kuvannut yhtälö, eli parhaan tapauksen aikavaativuus on $\mathcal{O}(n \log n)$
- Käytännössä hyvin toteutettu **pikajärjestäminen** toimii yleensä nopeammin kuin **lomitus-** tai **kekojärjestäminen**
- Huomaa kuitenkin, että tässä esitetty perusversio on hidas, jos taulukko on valmiiksi melkein järjestyksessä, mikä joissain sovelluksissa voi olla yleistä
- Osoittautuu, että **keskimääräisen tapauksen vaativuus pikajärjestämisellä** onkin $\mathcal{O}(n \log n)$. Analyysi löytyy Cormenista, mutta se ei kuulu tämän kurssin vaatimusten piiriin

- Analysoidaan vielä tilannetta missä **partition** jakaa alkioita melko huonosti, eli oletetaan että k :n alkion taulukko jakautuisi pahimmillaan siten että pienemmässä osassa on vain $k/10$ alkioita ja suuremmassa $9k/10$
- Näytetään että jopa tässäkin tapauksessa **pikajärjestämisen** vaativuus olisi $O(n \log n)$
- "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



- Saamme siis epätasapainoisten jakojen tapauksen aikavaativuudeksi

$$T_u(n) \leq cn(\log_{10/9} n + 1) = cn\left(\frac{\log_2 n}{\log_2 10/9} + 1\right) \leq cn(7 \cdot \log_2 n + 1) =$$

$$7 \cdot cn(\log_2 n + \frac{1}{7}) = \mathcal{O}(n \log n)$$

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

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

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

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

Tämä tuottaa silti vain vakiokertoimen 2 aikavaativuuteen

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

- Koska **quicksort** ja **partition** vaativat vain vakiomäärän apumuuttujia, **pikajärjestämisen** tilavaativuus on verrannollinen rekursion maksimisyvyyteen
- Pahimmassa tapauksessa se on edellä esitetyllä toteutuksella $O(n)$
- Tämä voidaan kuitenkin parantaa arvoon $O(\log n)$ toteutusta optimoimalla
- Ensimmäinen askel on korvata **quicksort**-kutsun päättävä rekursiivinen kutsu eli **häntärekursio** iteraatiolla:

```
quicksort2(A,vasen,oikea)
```

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

- Monet kääntäjät tekevät tällaisen optimoinnin automaattisesti
- Tämä ei vielä muuta pahimman tapauksen vaativuutta, koska pahimmassa tapauksessa rekursiivisesti käsiteltävä osataulukko $A[\textit{vasen} .. \textit{oikea}]$ on edelleen kooltaan $n - 1$

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

quicksort3(A,vasen,oikea)

```

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

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

- Sivulla 84 ratkaisimme saman rekursioyhtälön, ja päädyimme tulokseen $S(n) = \mathcal{O}(\log n)$, eli **quicksort3**:n tilavaativuus pahimmassa tapauksessa ja siis tietysti myös keskimäärin on $\mathcal{O}(\log n)$

Pikajärjestäminen käytännössä

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

```
quicksort-tuned(A,vasen,oikea)
```

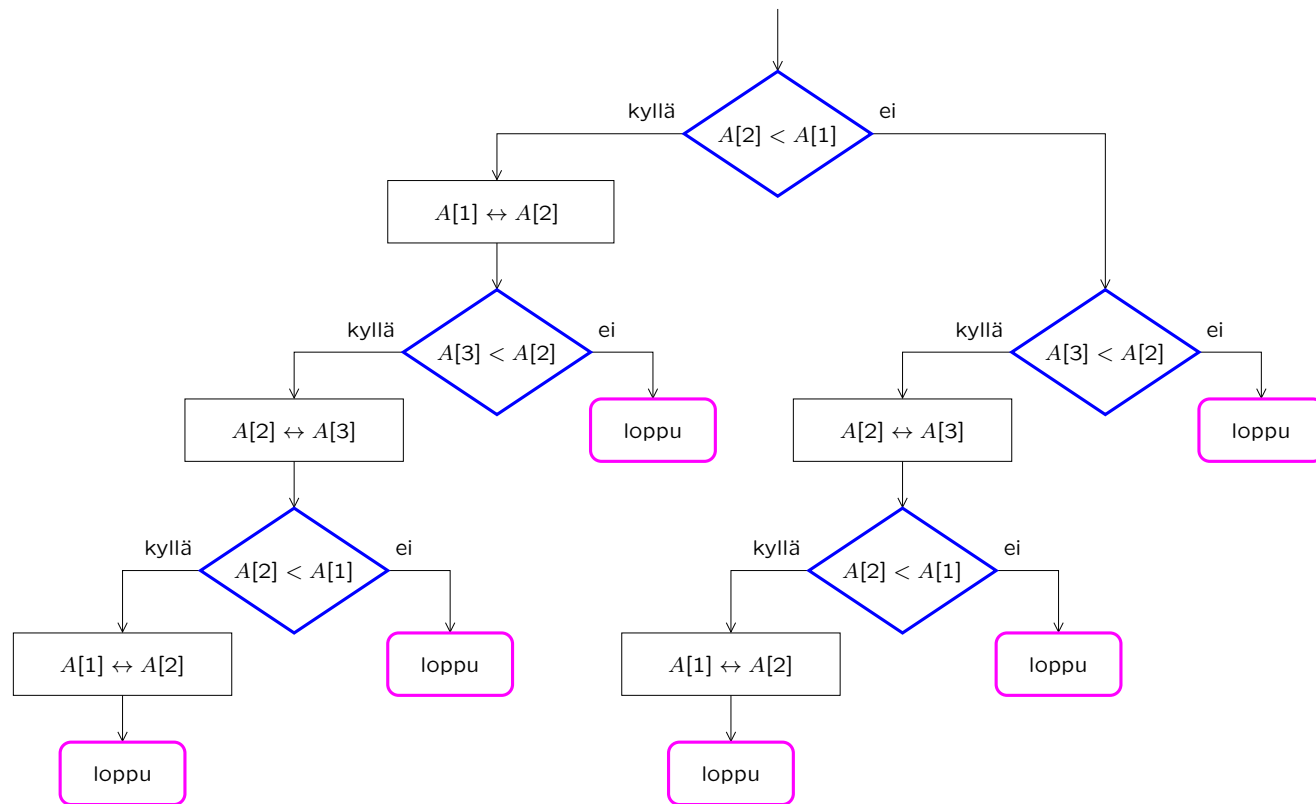
```
1  if oikea – vasen < 20  insertion-sort(A,vasen,oikea)
2  else
3      jako = partition(A,vasen,oikea)
4      quicksort-tuned(A,vasen,jako)
5      quicksort-tuned(A,jako+1,oikea)
```

- **partition**-operaatiossa paras valinta jakoalkioksi olisi jaettavan aineiston mediaani, eli suuruudeltaan puolessa välissä oleva alkio. Tällöin jako olisi aina tasainen ja aikavaativuus pahimmassa tapauksessa $\mathcal{O}(n \log n)$, mutta mediaanin selvittäminen ei käytännössä ole tehokasta (joskin mahdollista ns. BFPRT-algoritmilla)
- Melko hyvä keino on valita **jakoalkioksi kolmen mediaani** eli mediaani arvoista $A[\textit{vasen}]$, $A[\lfloor (\textit{vasen} + \textit{oikea})/2 \rfloor]$ ja $A[\textit{oikea}]$
- Pahinta tapausta (siis jolloin aikaa kuluu $\mathcal{O}(n^2)$) **pikajärjestämisessä** ei voida täysin välttää käyttämällä kolmen mediaania jakoalkion valintaan, mutta pahimman tapauksen esiintymistodennäköisyys on hyvin pieni

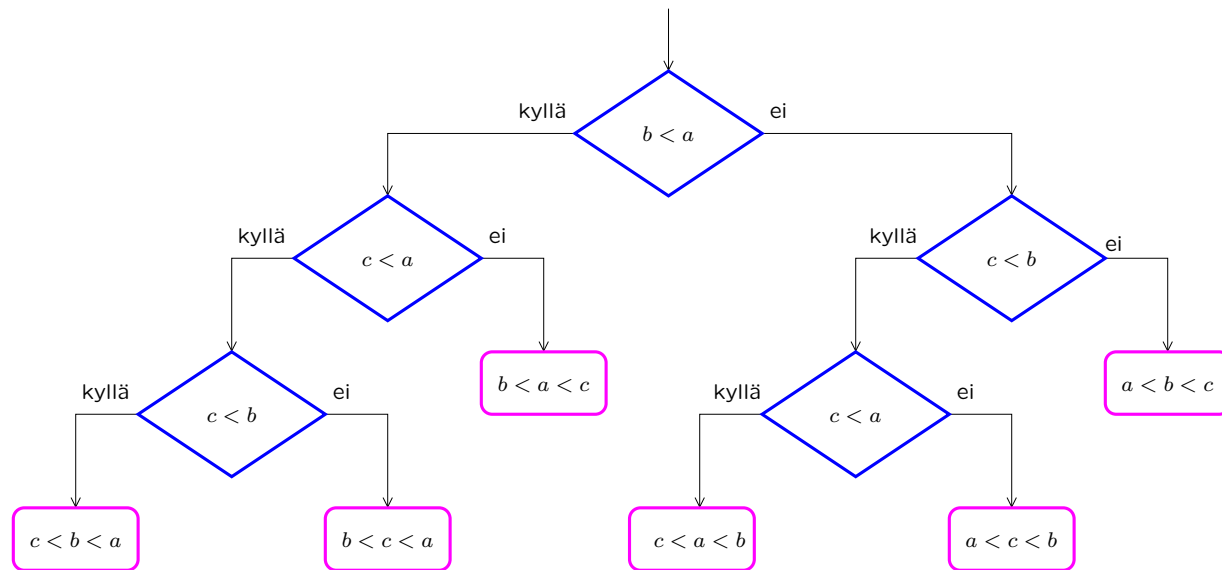
Alaraja vertailuihin perustuvalla järjestämiselle

- 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 alkioden olevan kokonaislukuja ja esim. käyttää keskiarvoa $(A[1] + A[n])/2$ tai testaa, onko $A[i]$ parillinen tai laskee taulukossa A esiintyvien arvojen lukumääriä
- Vertailuihin perustuva järjestämisalgoritmi suorittaa pahimmassa tapauksessa $\Omega(n \log n)$ vertailua
(Muistinvirkistus: merkinnällä Ω tarkoitetaan funktion kasvunopeuden alarajaa)

- Lähtökohtana on jokin vertailuihin perustuva järjestämisalgoritmi jollekin kiinteälle syötteen koolle n
- Alla on esimerkkinä vuokaavio **lisäysjärjestämiselle**, kun $n = 3$



- Saamme vuokaavioesityksestä päätöspuun, kun jätämme sijoitusoperaatiot pois ja merkitsemme muuttujien $A[1]$, $A[2]$ ja $A[3]$ alkuperäisiä arvoja a , b ja c
- Testien päätyttyä nähdään, mikä on alkioden a , b ja c järjestys



- Yksinkertaisuuden vuoksi oletetaan, että alkiod ovat erisuuria

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

- Teemme nyt hyvin karkean arvion

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

- Siis mikä tahansa vertailuihin perustuva järjestämisalgoritmi tekee pahimmassa tapauksessa ainakin $(n/2) \log_2(n/2) = \Omega(n \log n)$ vertailua
- Siis **kekojärjestämisen** ja **lomituserjestämisen** aikavaativuudet ovat vakiokerrointa vaille optimaalisia, kun rajoitutaan vertailuihin perustuviin algoritmeihin

Järjestäminen lineaarisessa ajassa

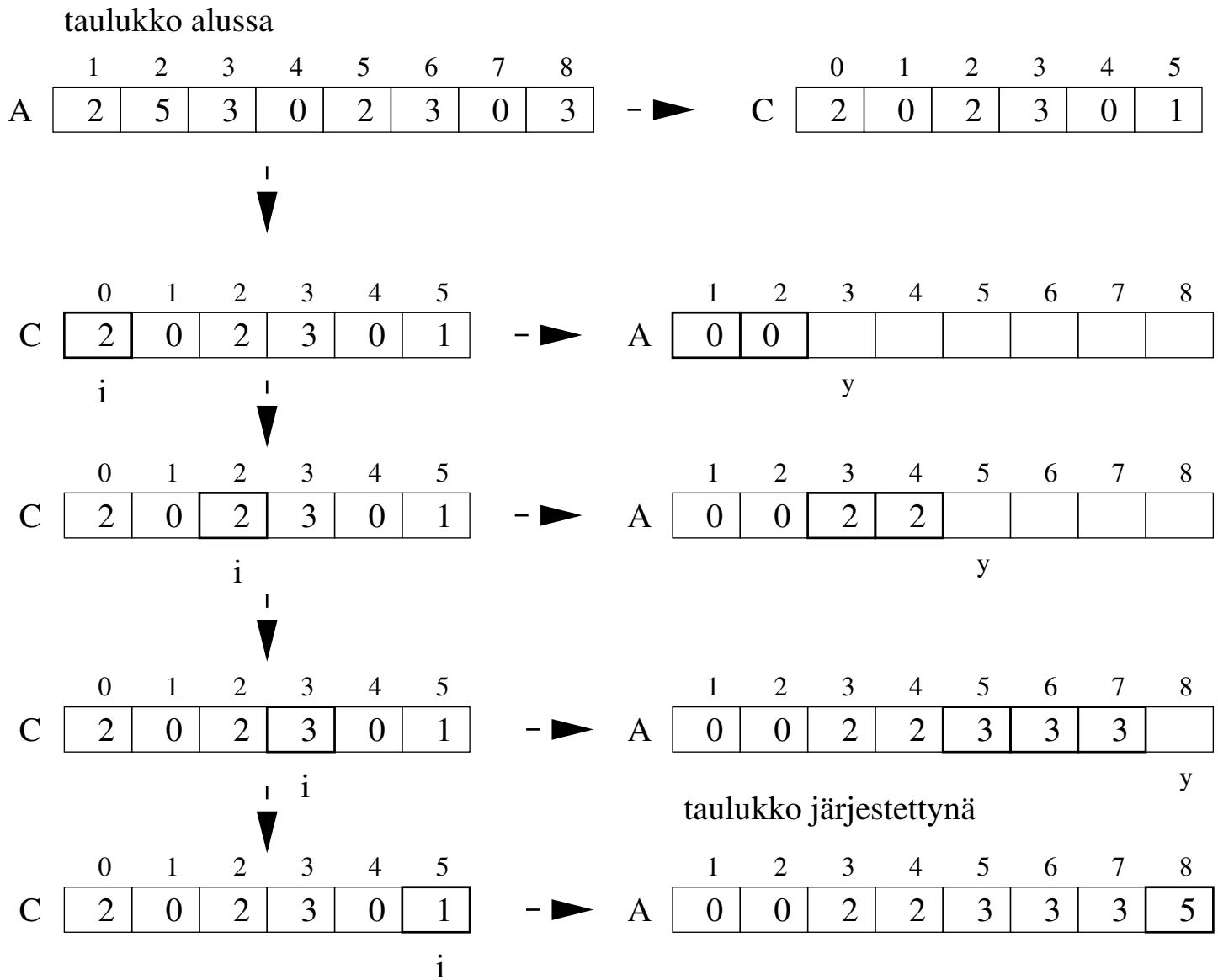
- Tehokkuusraja $\mathcal{O}(n \log n)$ voidaan rikkoa, jos järjestäminen perustuu johonkin muuhun kuin alkioiden keskinäiseen vertailuun
- Oletetaan että järjestettävä aineisto $A[1, n]$ koostuu luvuista joiden arvo on väliltä $0, \dots, k$
- Yksinkertainen ja tehokas järjestämismenetelmä saadaan aikaan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

- Algoritmina:

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

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

• Esimerkki:



- Algoritmi käy kerran läpi taulukon A , kahteen kertaan taulukon C ja tulostaa luvun jokaiseen A :n paikkaan, aikavaativuus siis $\mathcal{O}(n + k)$ ja jos $k = \mathcal{O}(n)$ niin aikavaativuus on lineaarinen järjestettävän aineiston koon suhteen
- Tilavaativuus on luonnollisesti $\mathcal{O}(k)$
- Jos järjestettäviin alkioihin liittyy muita datakenttiä ei juuri esitetty versio **laskemisjärjestämisestä** toimi

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

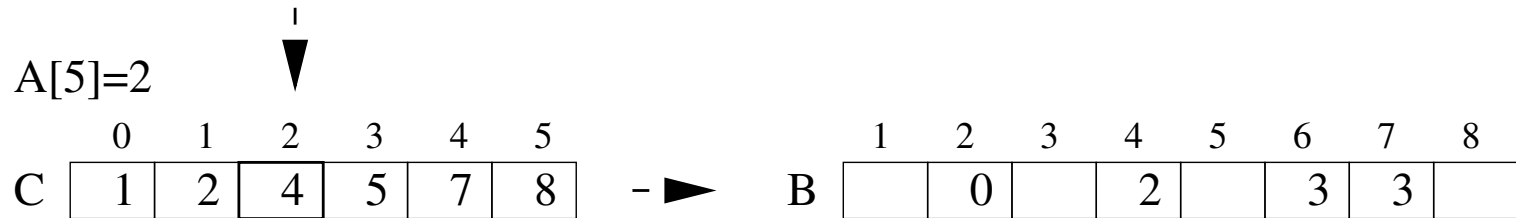
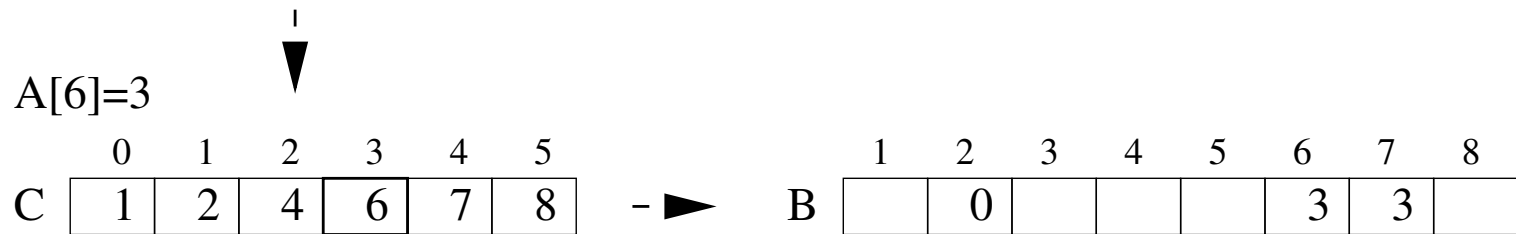
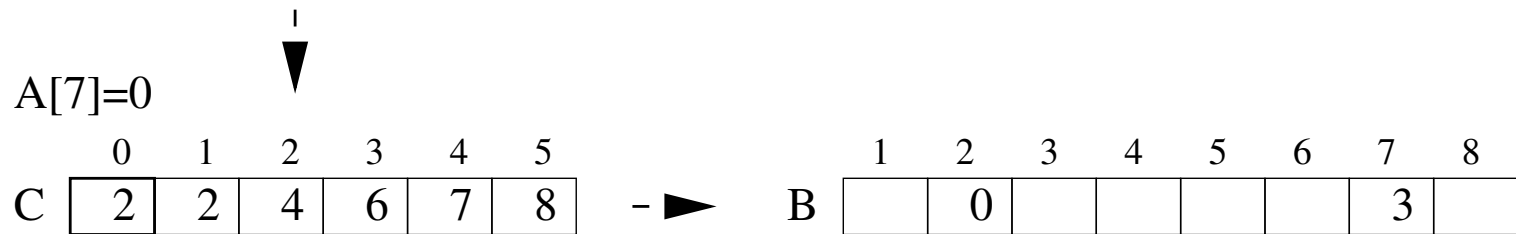
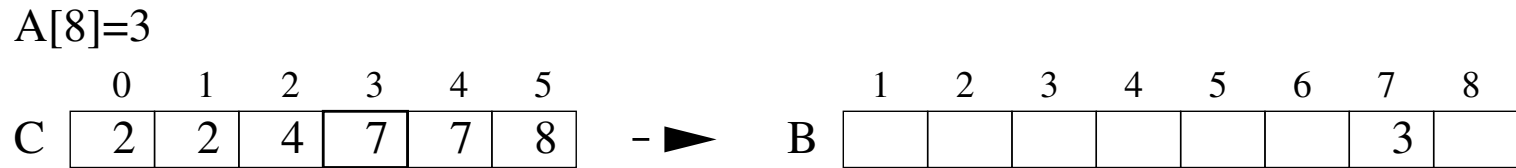
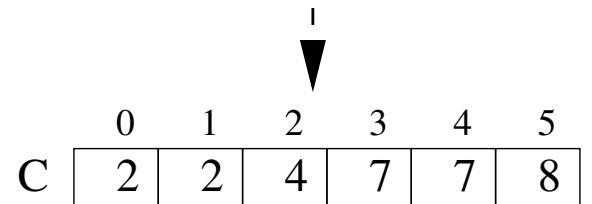
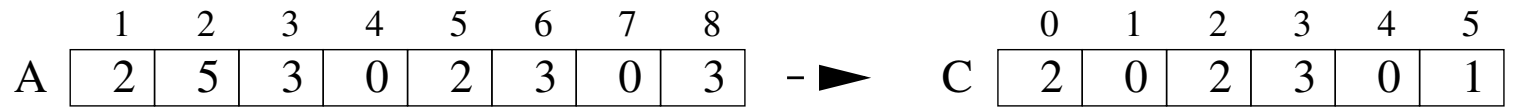
counting-sort2(A,k,n)

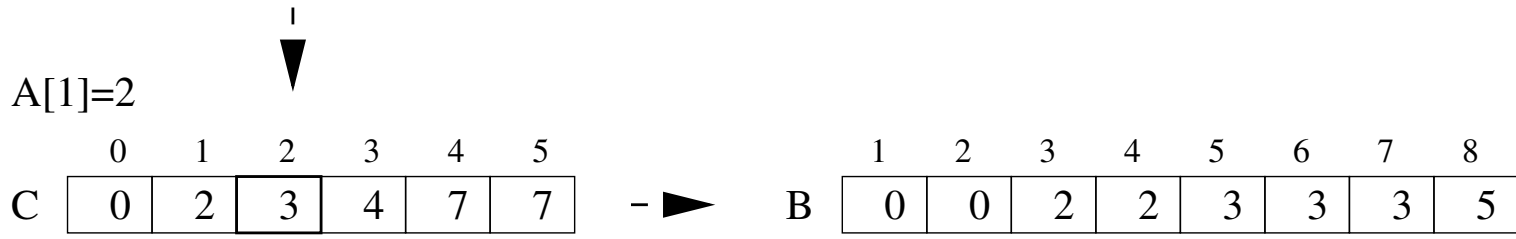
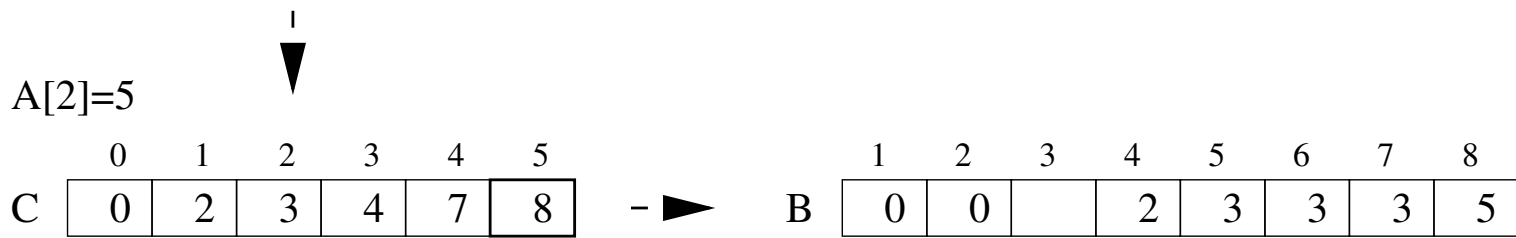
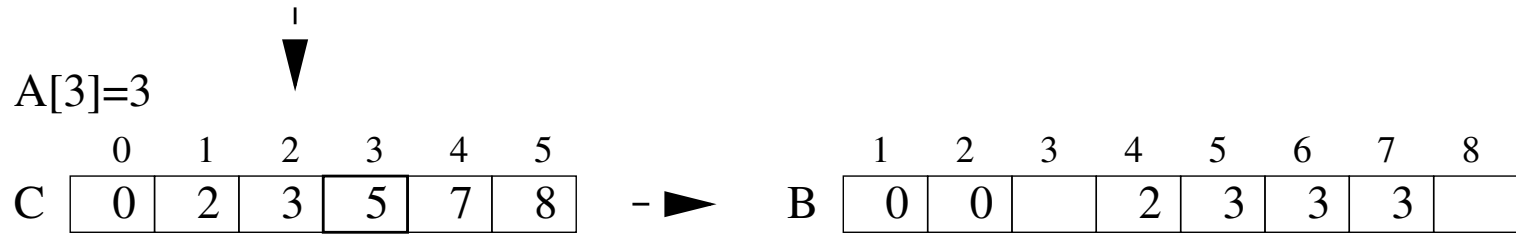
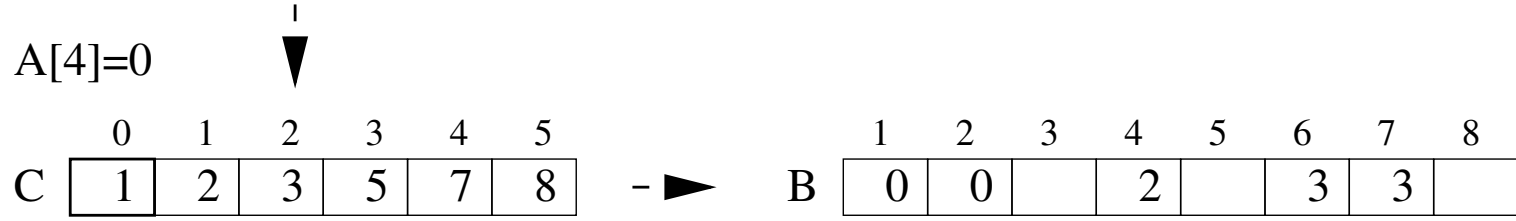
```
1  for i = 0 to k C[i] = 0
2  for j = 1 to n
3      x = A[j]
4      C[x] = C[x]+1
5  for i = 1 to k
6      C[i] = C[i]+C[i-1]
7  for j = n downto 1
8      x = A[j]
9      B[C[x]] = x
10     C[x] = C[x]-1
11 for i = 1 to n A[i] = B[i]
```

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

taulukko alussa





taulukko järjestyksessä

Yhteenveto järjestämialgoritmeista

- Aikavaativuus

Aikavaativuus	pahin tapaus	keskim. tapaus	paras tapaus	vakaa
kupla	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	on
lisäys	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	on
lomitus	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	on
keko	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	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 kirjastototeutukset perustuvat useimmiten viriteltyyn **pikajärjestämiseen**
- **Pikajärjestämisen** keveys esim. **lomitusjärjestämiseen** verrattuna johtuu siitä, että tulosten yhdistämisvaihetta ei tarvita ja alkioiden jaon suorittava **partition** on suhteellisen kevyt operaatio, sillä se käy jaettavat alkiot läpi ainoastaan kertaalleen
- Järkevästi toteutetussa **pikajärjestämisessä** pahin tapaus eli neliöisen suoritusajan tuottava tapaus on käytännössä erittäin harvinainen

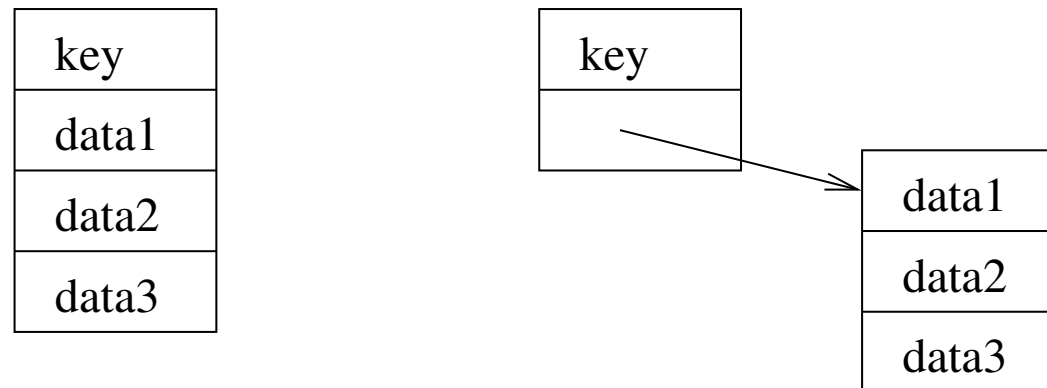
- Jos on erittäin tärkeää, että pahin tapaus ei toteudu koskaan, esim. turvallisuuskriittisissä järjestelmissä, joissa vasteaikojen on oltava kaikissa tapauksissa ennustettavat, kannattaa käyttää **lomitusjärjestämistä** tai **kekojärjestämistä**, sillä ne ovat pahimmassakin tapauksessa ajassa $\mathcal{O}(n \log n)$ toimivista algoritmeista käytännössä nopeimmat
- Jos **pikajärjestämisen partition**-operaatio onnistuisi valitsemaan jakoalkioksi tarkasteltavan taulukonosan alkioden mediaanin, olisivat jaot aina tasaisia, eikä pahin tapaus $\mathcal{O}(n^2)$ koskaan toteutuisi
 - yllättäen osoittautuu, että k :n kokoisesta taulukosta on mahdollista löytää mediaani ajassa $\mathcal{O}(k)$, eli **pikajärjestäminen** on mahdollista toteuttaa toimimaan pahimmassakin tapauksessa ajassa $\mathcal{O}(n \log n)$
 - mediaanin selvittäminen lineaarisessa ajassa on kuitenkin käytännössä niin hidas operaatio, että sen käyttäminen **pikajärjestämisen** yhteydessä huonontaisi algoritmin toimintaa liian paljon

- Vakautta edellytettäessä eivät **pika-** ja **kekojärjestäminen** kelpaa, vaan valinta on **lomitusjärjestäminen**
- **Lomitusjärjestäminen** on kehittyneimmistä järjestämisalgoritmeista suoritusajaltaan ennustettavin, sillä se toimii käytännössä kaikenmuotoisilla syötteillä täysin samalla tavalla.
- Javan valmiista järjestämisalgoritmeista taulukoita järjestävä `Arrays.Sort` perustuu **pikajärjestämiseen**, kokoelmia järjestävä `Collections.Sort` taas **lomitusjärjestämiseen**
- **Laskemisjärjestäminen** näyttää lineaarisine aikavaativuuksineen houkuttavalta valinnalta. Käytännössä ongelmaksi osoittautuu järjestettävien alkioiden arvoalueen suuri koko. Jos järjestettävänä on esim. max 20 merkin mittaisia merkkijonoja, arvoalueen on koko luokkaa $(2 \cdot 27)^{20}$, eli **laskemisjärjestäminen** on tilanteessa täysin käyttökelvoton
- Jos sovellus koostuu esim. kymmenestä tuhannesta pienestä taulukosta jotka on järjestettävä, on paras valinta **lisäysjärjestäminen**
- **Lisäysjärjestäminen** toimii myös parhaiten isoillakin taulukoilla, jos taulukot ovat melkein järjestyksessä

4. Perustietorakenteet: pino, jono ja lista

Abstrakti tietotyyppi joukko

- Usein ohjelma ylläpitää suoritusaikanaan jotain joukkoa tietoalkioita, esim. puhelinluettelo nimi–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 johonkin viitteen takana olevaan olioon



- Jälkimmäinen vaihtoehto on järkevämpi varsinkin silloin, jos tietoalkioilla on kaksi eri avainta: esim. puhelinluettelosta pitää pystyä tekemään hakuja sekä nimen että numeron perusteella . . .
- Useimmat ohjelmat tarvitsevat joukolle (merkitään S :llä) seuraavia useimmiten operaatioita:
 - **search**(S, k) jos joukosta löytyy avaimella k varustettu alkio, operaatio palauttaa viitteen alkioon, muuten operaatio palauttaa viitteen NIL
 - **insert**(S, x) lisää joukkoon alkion johon x viittaa
 - **delete**(S, x) poistaa tietueen johon x viittaa
- Joskus ovat tarpeen myös seuraavat operaatiot:
 - **min**(S) palauttaa viitteen joukon pienimpään alkioon
 - **max**(S) palauttaa viitteen joukon suurimpaan alkioon
 - **succ**(S, x) palauttaa viitteen x :ää seuraavaksi suurimpaan alkioon; jos x :n avain oli suurin palauttaa NIL (x on viite)
 - **pred**(S, x) palauttaa viitteen x :ää seuraavaksi pienempään alkioon; jos x :n avain oli pienin palauttaa NIL (x on viite)
- Huomaamme että sivulla 27 mainituista puhelinluettelon operaatioista suurin osa on suoraan tarjolla abstraktin tietotyypin joukko-operaationa

- Operaatio, joka tulostaa nimi–numero-parit aakkosjärjestyksessä, saadaan myös helposti toteutetuksi joukon operaatioiden avulla kysymällä ensin pienin alkio operaatiolla **min** ja sen jälkeen seuraavat järjestyksessä kutsumalla toistuvasti **succ**-operaatiota
- Puhelinluettelossa on kaksi etsimisoperaatiota findNumber ja findName, eli etsi numero nimen perusteella ja etsi nimi nimeron perusteella
- Joukossa on kuitenkin vain yksi etsintäoperaatio, **search**, joka etsii tietyllä avaimella varustetun alkion
- Yhden joukon avulla saadaankin puhelinluettelosta tehtyä vain versio, jossa on joko findNumber tai findName "tehokkaasti" toteutettuna, muttei molempia
- Jos valitaan avaimeksi nimi, niin operaatio findNumber, eli etsi tietyn henkilön puhelinnumero, saadaan tehokkaasti toteutettua joukon avulla
- Jos halutaan molemmat operaatiosta findNumber ja findName tehokkaaksi, tarvitaankin kaksi joukkoa (toisessa avaimena nimi, toisessa numero)
- Palaamme aiheeseen muutaman viikon päästä

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

- Operaatioiden aikavaativuus riippuen käytetystä tietorakenteesta:

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

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

- Tietorakenteen johon alkioita talletetaan ei välttämättä aina tarvitse olla matemaattisessa mielessä joukko, eli sama alkio voi olla mielekästä pystyä tallentamaan tietorakenteeseen moneen kertaan
- Tietorakennetta jossa voi olla monta samaa alkioita voidaan kutsua esim. kokoelmaksi (engl. collection)

Javan valmiit tietorakennetoteutukset

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

Pino ja jono

- Ennen kuin menemme varsinaiseen lista-tietorakenteeseen, opiskelemme kahta joukon erikoistapausta, **pinoa** (engl. stack) ja **jonoa** (engl. queue)
- Pino ja jono eivät toteuta sivun 159 joukko-operaatioita, vaan toimivat niinkuin pino ja jono toimivat normaalielämässä
- Pino ja jono ovat tärkeitä rakenneosasia monissa algoritmeissa, kuten tulemme myöhemmin näkemään

Pino

- Last-in-first-out -periaate, eli toimii kuten Unicafessa lautaspino
- Pino-operaatiot:
 - **push**(S, x) laittaa tietoalkion (johon x viittaa) pinon päälle
 - **pop**(S) palauttaa viitteen pinon päällimmäiseen tietoalkioon ja poistaa sen pinosta
 - **empty**(S) palauttaa **true** jos pino tyhjä, muuten **false**
- Oletetaan seuraavassa yksinkertaisuuden vuoksi että pinoon talletettavat alkiot sisältävät ainoastaan kentän *key*, eli ovat esim. kokonaislukuja
- Eli operaation parametrit yksinkertaistetussa tapauksessa:
 - **push**(S, k) laittaa tietoalkion k pinon päälle
 - **pop**(S) palauttaa pinon päällimmäisen tietoalkion

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

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

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

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

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

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



	0	1	2	3	4	5	6
table	15	6	2	9			

top = 3



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

	0	1	2	3	4	5	6
table	15	6	2	9	17	3	

top = 5



pop(S)

	0	1	2	3	4	5	6
table	15	6	2	9	17		

top = 4

- Operaatioiden toteutus:

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

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

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

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

- Kaikki pino-operaatiot sisältävät vain saman vakiomäärän komentoja riippumatta siitä, kuinka monta alkiota pinossa on, operaatiot ovat siis vaativuudeltaan $\mathcal{O}(1)$
- Myös operaatioiden tilavaativuus on $\mathcal{O}(1)$. Tilavaativuudellahan tarkoitettiin operaation aikana käytettyjen apumuuttujien määrää.
Vaikka pinossa olisikin paljon tavaraa, niin operaatioiden aikana ei apumuuttujia tarvita kuin **pop**:issa, josta siitäkin on tarvittaessa helppo päästä eroon
- Täyteen pinoon alkion laittaminen aiheuttaa **ylivuodon** (engl. overflow) ja alkion ottaminen tyhystä pinosta aiheuttaa **alivuodon** (engl. underflow):
Javassa poikkeuksen `ArrayIndexOutOfBoundsException`
- On sovelluskohtaista miten tilanteeseen kannattaa varautua
- Yksi mahdollisuus on tietysti hoitaa poikkeus `try-catch`-mekanismin avulla
- Ehkä järkevämpää on toteuttaa metodi, jolla voidaan varmistua, että pino ei ole täysi:

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

- On myös mahdollista tehdä pinon taulukosta tarpeen mukaan kasvava, eli jos **push**-operaatio toteaa taulukon olevan täynnä, se luo uuden taulukon johon alkuperäiset alkiot kopioidaan
tämä kuitenkin tekee **push**:in pahimman tapauksen aikavaativuudesta lineaarisen pinon koon suhteen
- Jos pinon tallennusalue toteutettaisiin `ArrayList`:in avulla, tilanne on sama sillä `ArrayList` joutuu tekemään raskaan tilankasvatusoperaation
- Tarkemmassa analyysissä osoittautuu, että jos pinon koko aina kaksinkertaistetaan kasvatustilanteessa, niin vaikutus aikavaativuuteen ei ole loppujenlopuksi kovin dramaattinen:
 - yksittäinen **push**-operaatio vie pahimmassa tapauksessa aikaa $\mathcal{O}(n)$
 - $n:n$ peräkkäisen pino-operaation (joihin saa kuulua myös **push**:in paha tapaus) yhteenlaskettu aikavaativuus on kuitenkin sekin $\mathcal{O}(n)$, eli yhtä komentoa kohti ainoastaan $\mathcal{O}(1)$
 - eli siis vaikka **push** voi joskus viedä paljon aikaa, ovat muut sarjassa suoritettut 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
- Tämänäköistä aikavaativuusanalyysiä kutsutaan **tasoitetuksi analyysiksi** (engl. amortized analysis), ja on lähinnä syventävien opintojen asioita
- Myös `ArrayList` käyttäytyy näin: silloin tällöin saattaa tulla kallis operaatio, mutta pitkää operaatiosarjaa tarkasteltaessa yhdelle operaatiolle tuleva keskimääräinen osuus on vakioaikainen (jos huomioidaan ainoastaan lisäyksen ja poistot `ArrayListin` lopusta)

Jono

- **First-in-first-out**, eli toimii kuten jono Unicafessa
- Jono-operaatiot:
 - **enqueue**(Q, k) laittaa tietoalkion k jonon perälle
 - **dequeue**(Q) palauttaa jonon ensimmäisenä olevan tietoalkion ja poistaa sen jonosta
 - **empty**(Q) palauttaa **true** jos jono tyhjä, muuten **false**
- Jos jonon koon yläraja tiedetään ennakolta, toteutus onnistuu taulukon avulla
- Toteutus Javalla seuraavaksi

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

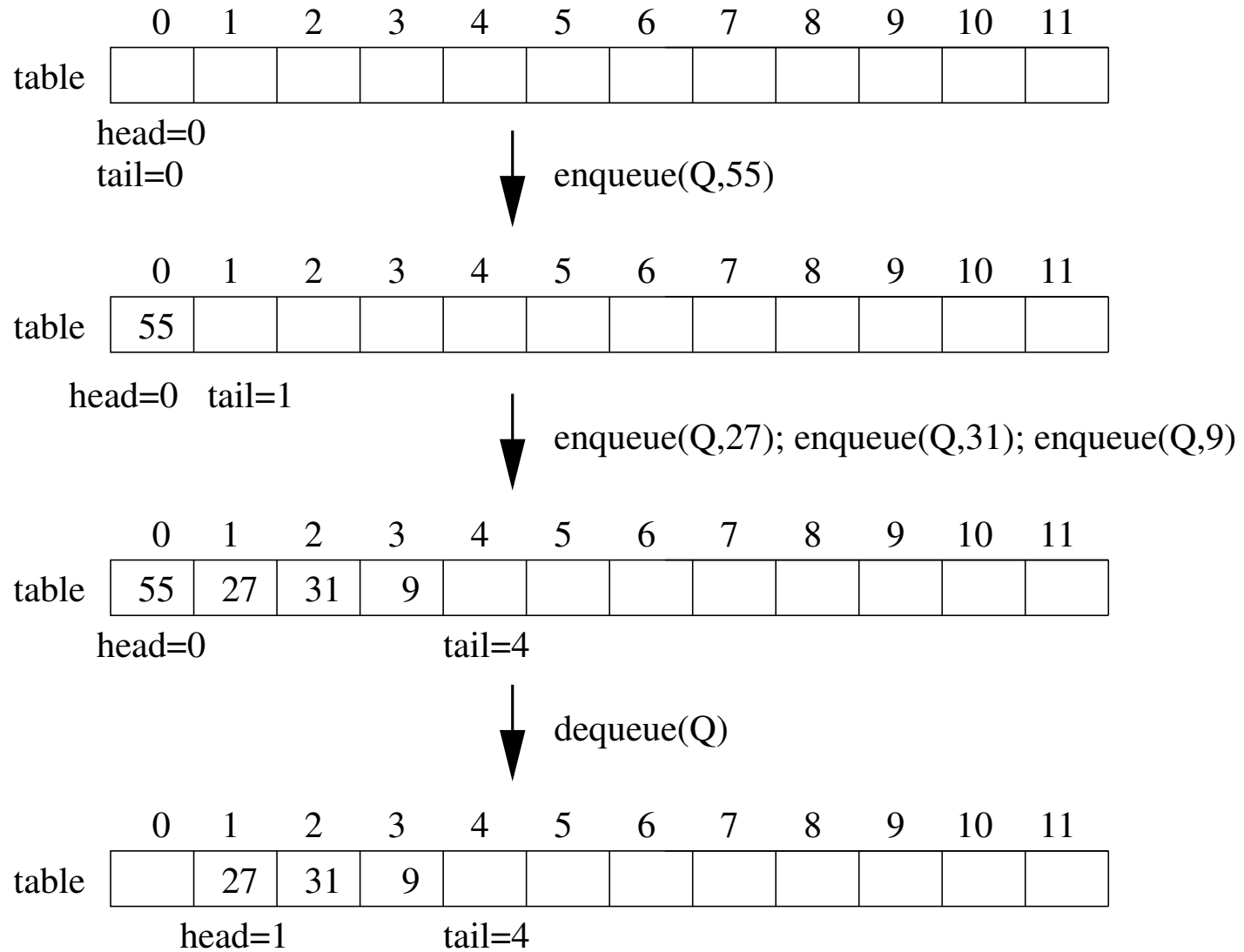
```
class Queue {  
    int[] table;  
    int head;  
    int tail;  
    int n;
```

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

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

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

- Jono alkaa täyttyä yllätyksettömällä tavalla taulukon alusta alkaen



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

	0	1	2	3	4	5	6	7	8	9	10	11	
table							15	6	9	8	4		
							head=6					tail=11	



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

	0	1	2	3	4	5	6	7	8	9	10	11
table	3	5					15	6	9	8	4	17
			tail=2				head=6					



dequeue(Q)

	0	1	2	3	4	5	6	7	8	9	10	11
table	3	5						6	9	8	4	17
			tail=2					head=7				

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

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

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

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


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

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

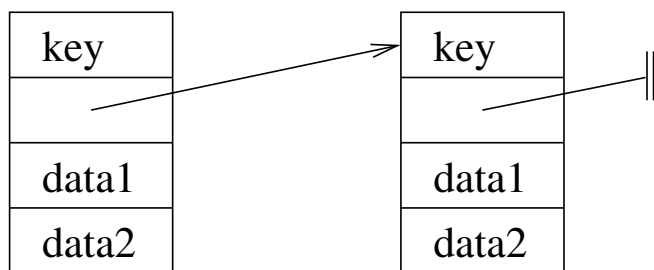
- Alkiot poistetaan jonon edestä eli kohdasta `head`
- Jonon etuosaa merkkäava `head` on päivitettävä uuteen arvoon, eli se joko kasvaa yhdellä tai pyörähtää ympäri, jos jono alkaa taas kerran taulukon alusta

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

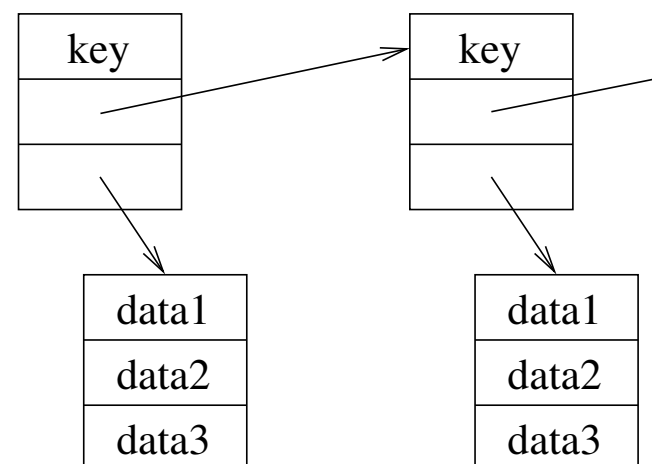
- Myös jonon tapauksessa kaikkien operaatioiden aika- ja tilavaativuus on selvästi $\mathcal{O}(1)$

Linkitetty lista

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



NIL eli linkki ei mihinkään merkitään —||

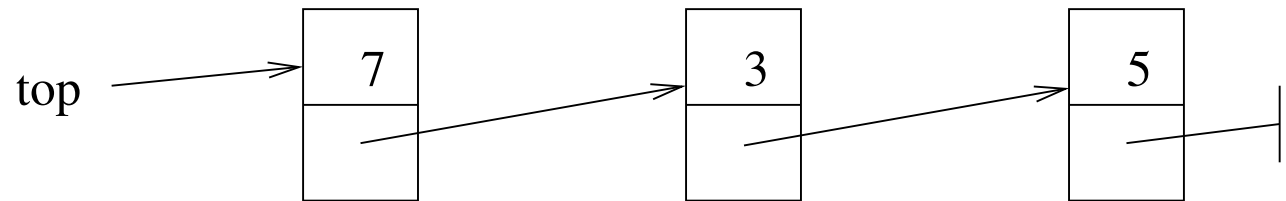


- **Huom.** nykyaikaisilla tietokoneilla taulukoiden käsittely on usein hyvin nopeaa, mutta linkkien käsittely hidasta. Tällaisten rakenteiden vertailussa pelkkään O-aikavaativuuteen tuijottaminen voi johtaa harhaan
- Joka tapauksessa linkitetyn tietorakenteen perusajatus pitää ymmärtää, sitä tarvitaan jatkossa monimutkaisemmissa tietorakenteissa joita ei helposti saa esitetyksi taulukkona

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

<i>key</i>	pinoon talletettu tietoalkio
<i>next</i>	viite toiseen pinosolmu-olioon
 - pinolla (jota merkitään *S*) on attribuutti *top* joka on viite siihen pinosolmu-tyyppiä olevaan olioon, joka tallettaa pinon päällä olevan alkion
 - jos *S.top* = NIL, pino on tyhjä
 - jokaiseen pinosolmun attribuuttiin *next* talletetaan pinossa seuraavana olevan pinosolmun viite
 - pinon pohjimmaisena pinosolmun *next*-attribuutin arvo on NIL

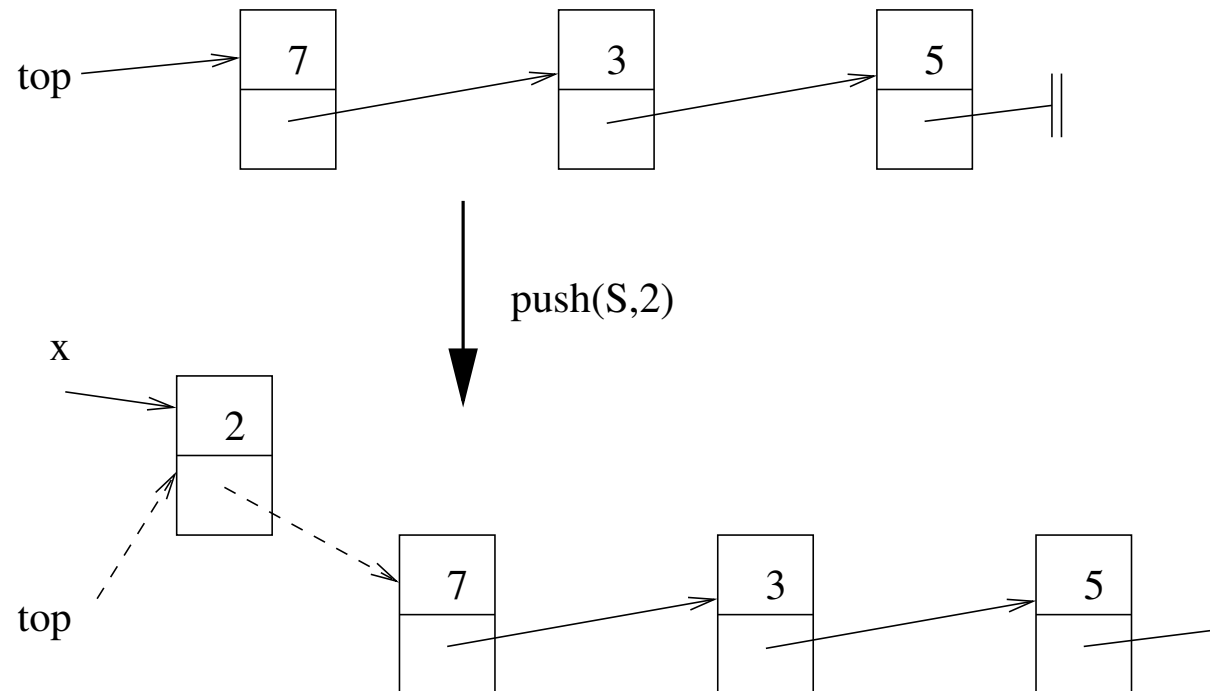
- Esim: Pino missä huipulla 7, jonka alla 3 ja 5



- Seuraavaksi operaatioiden toteutus ja toiminta

push(S,uusi)

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

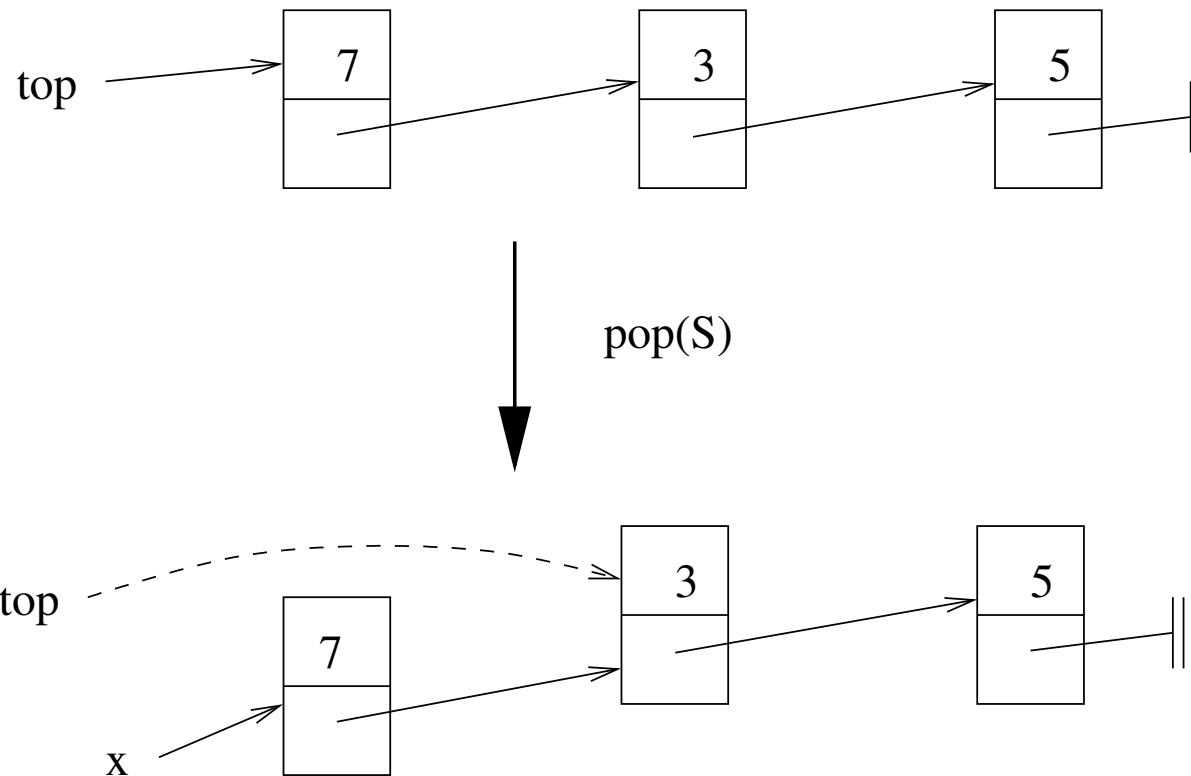


- Huom: Kuten sivulla 43 mainittiin, on pseudokoodi epäolio-orientoitunutta, eli operaatiot eivät liity suoraan mihinkään olioon. Kaikki käsiteltävät oliot (kuten nyt pino S) on välitettävä operaatioille parametrina

pop(S)

```
x = S.top  
S.top = x.next  
return x.key
```

```
// pinon uusi huippu on ...  
// vanhan huipun x alapuolella ollut pinosolmu  
// palautetaan vanhan huipun sisältämä tieto
```



- Huom: **pop**:in jälkeen x :n viittaama pinon vanha huippu jää "orvoksi", esim. Javassa roskankeruumekanismi huolehtisi sen varaaman muistin vapauttamisesta
- Joissain kielissä, kuten C:ssä ja C++:ssa ohjelmoijan on vapautettava itse x :n viittaama muistialue
- Pinon tyhjyyden tarkastaminen on helppoa

empty(S)

```
return S.top==NIL
```

- Olettaen, että koneesta ei lopu muisti, voi linkitettynä rakenteena toteutettuun pinoon lisätä rajattoman määrän alkiota
- Toisin kuin taulukkoon perustuva, linkitettynä rakenteena toteutettu pino kuluttaa muistia vain sen verran, mitä kullakin hetkellä pinossa olevan tiedon talletukseen vaaditaan (+ olioviitteiden talletukseen kuluva muisti)
- Selvästikin pino-operaatiot ovat jälleen aikavaativuudeltaan $\mathcal{O}(1)$
- Myös jono voidaan toteuttaa linkitettynä rakenteena siten että jono-operaatiot vaativat vakioajan

Pinon linkitetty toteutus Javalla

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

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

- Pinosolmut kannattaa toteuttaa omana luokkana

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

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

- Luokan Pino ainoa attribuutti on PinoSolmu top, eli viite pinon päällä olevaan PinoSolmu-olioon
- Pinon konstruktori asettaa top = null, sillä pino on aluksi tyhjä

```
public class Pino {  
    private PinoSolmu top;    // viite pinon päällä olevaan alkioon  
  
    public Pino() { top = null; }    // alussa pinossa ei ole mitään
```

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

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

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

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

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

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

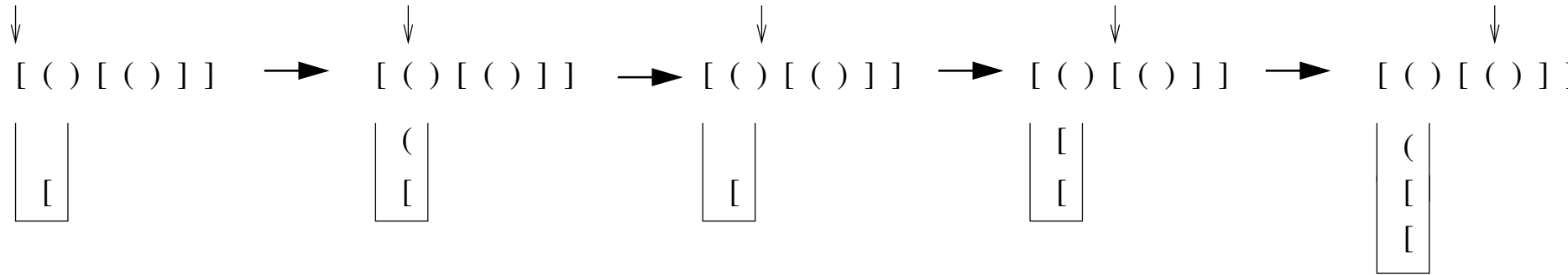
- Pinon käyttö sovelluksissa:

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

Esimerkki pinoa käyttävästä algoritmista

- Ohjelmoinnissa tulee helposti virheitä sulkumerkkien { } () [ja] kirjoittamisessa:
 - jollekin alkusululle ei löydy loppusulkua tai päinvastoin: ()) tai ((
 - sulut voivat "mennä ristiin": ({}))
- Sulkujen tasapaino syötteenä olevasta merkkijonosta voidaan testata esim. seuraavasti:
 - luetaan syötemerkkijonon merkit yksi kerrallaan, alusta alkaen
 - viedään jokainen alkusulkumerkki eli (, { tai [pinoon
 - kun luetaan jokin loppusulkumerkki eli), } tai] sitä vastaavan alkusulkumerkin pitää löytyä pinon päältä. alkusulku poistetaan pinosta
 - kun tiedosto on luettu, pinon pitää olla tyhjä

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



- Pinon päällä on aina viimeksi nähty alkusulkumerkki, jonka vastinetta ei ole vielä tullut vastaan
- Eli kun syötteessä tulee vuoroon loppusulkumerkki, on sen oltava pinon päällimmäisen vastine, muuten sulut ovat ristissä
loppusulkumerkin löytyessä pinon päällimmäinen sulku otetaan pois sillä se on käsitelty, esim. kuvassa kolmas askel
- Jos taas seuraava sulkumerkki onkin alkusulku, menee se pinon päällimmäiseksi ja ruvetaan odottamaan sitä vastaavaa loppusulkua, esim. kuvassa toinen askel
- Syvemmillä pinossa olevat sulut ovat ulompaa sulutustasoa, ja ne käsitellään pinon päällimmäisen käsittelyn jälkeen

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

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

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

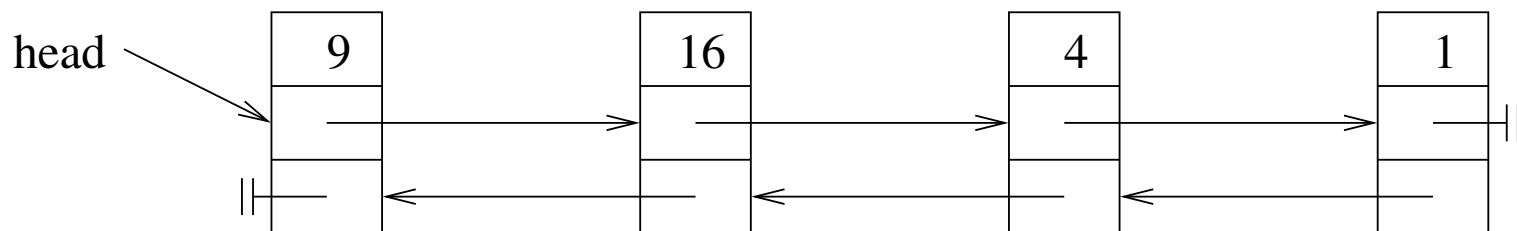
- Mikä on algoritmin vaativuus?
- `for`-lauseen runko suoritetaan kertaalleen jokaiselle syötemerkkijonon merkille
- Jokaisen merkin kohdalla tehdään muutama vertailuoperaatio sekä maksimissaan yksi pino-operaatio
- Pino-operaatioiden vaativuus on $\mathcal{O}(1)$, eli `for`-lauseen runko-osa vie vakioajan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

Kahteen suuntaan linkitetty lista

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

<i>key</i>	talletetun tietoalkion avain
...	mahdolliset muut tietoa tallettavat attribuutit
<i>next</i>	viite seuraavana olevaan listasolmu-olioon
<i>prev</i>	viite edellisenä olevaan listasolmu-olioon

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



- Alkiota listalta etsivässä **search**-operaatiossa verrataan listan alkioita alusta lähtien etsittävään
- Operaatio palauttaa viitteen etsityn alkion tallettamaan listasolmuun
- Jos etsittyä ei löydy, palautetaan viite NIL

search(L,k)

 p = L.head

 while p ≠ NIL and k ≠ p.key // jos ei olla lopussa tai löydetty etsittyä

 p = p.next // jatketaan eteenpäin

 return p

- apumuuttuja p viittaa listasolmuun, jossa etsintä on menossa
- aluksi p viittaa ensimmäiseen listasolmuun
- Listalla edetessä p laitetaan viittaamaan seuraavaan listan solmuun asettamalla sen arvoksi nykyisen solmun p seuraava eli $p.next$
- Jos etsittävää ei löydy tai se on listan viimeisenä, käydään kaikki listan alkiot läpi. Pahimmassa tapauksessa operaation aikavaativuus on siis $\mathcal{O}(n)$, missä n on listalla olevien alkioden 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

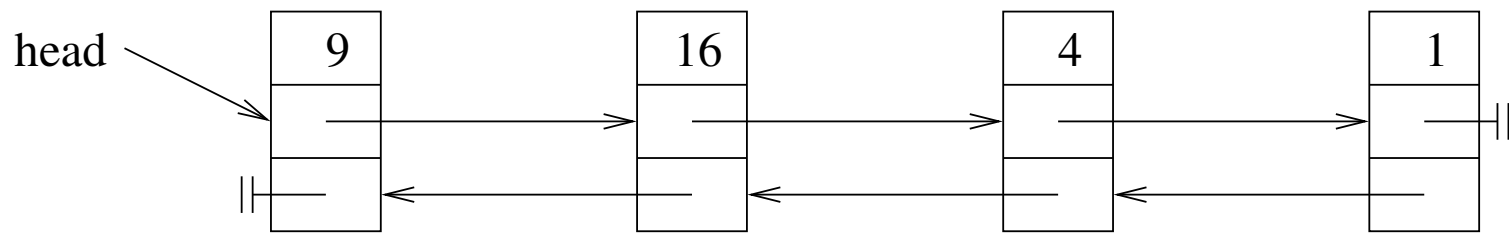
insert(L,k)

```

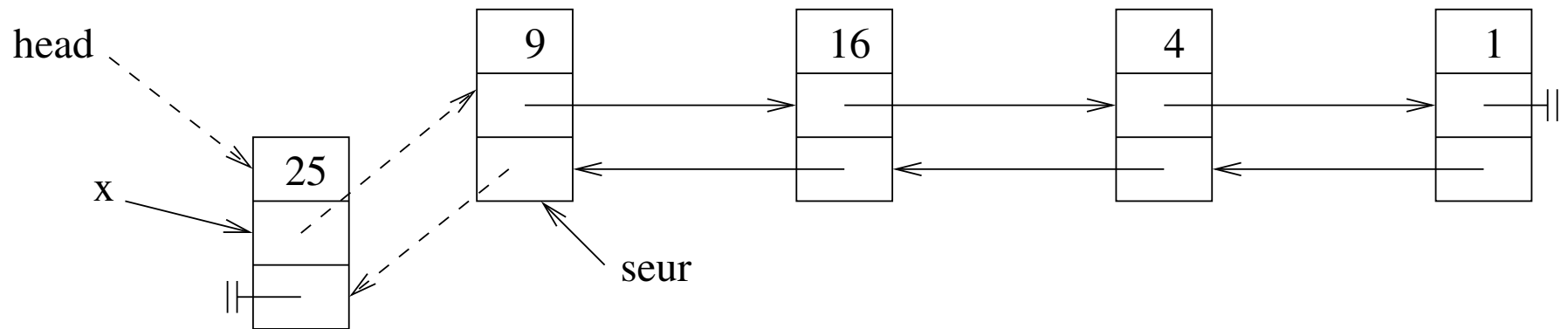
x = new listasolmu
x.key = k
x.next = L.head           // uuden seuraava on vanha ensimmäinen
x.prev = NIL
if L.head ≠ NIL
    seur = x.next
    seur.prev = x         // uusi on seuraajansa edeltäjä
L.head = x                // lisätty solmu on listan ensimmäinen

```

- Esimerkki seuraavalla sivulla selventää operaation toimintaperiaatetta



↓ insert(L,25)



- Riippumatta jonossa olevien alkioden 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 \neq NIL

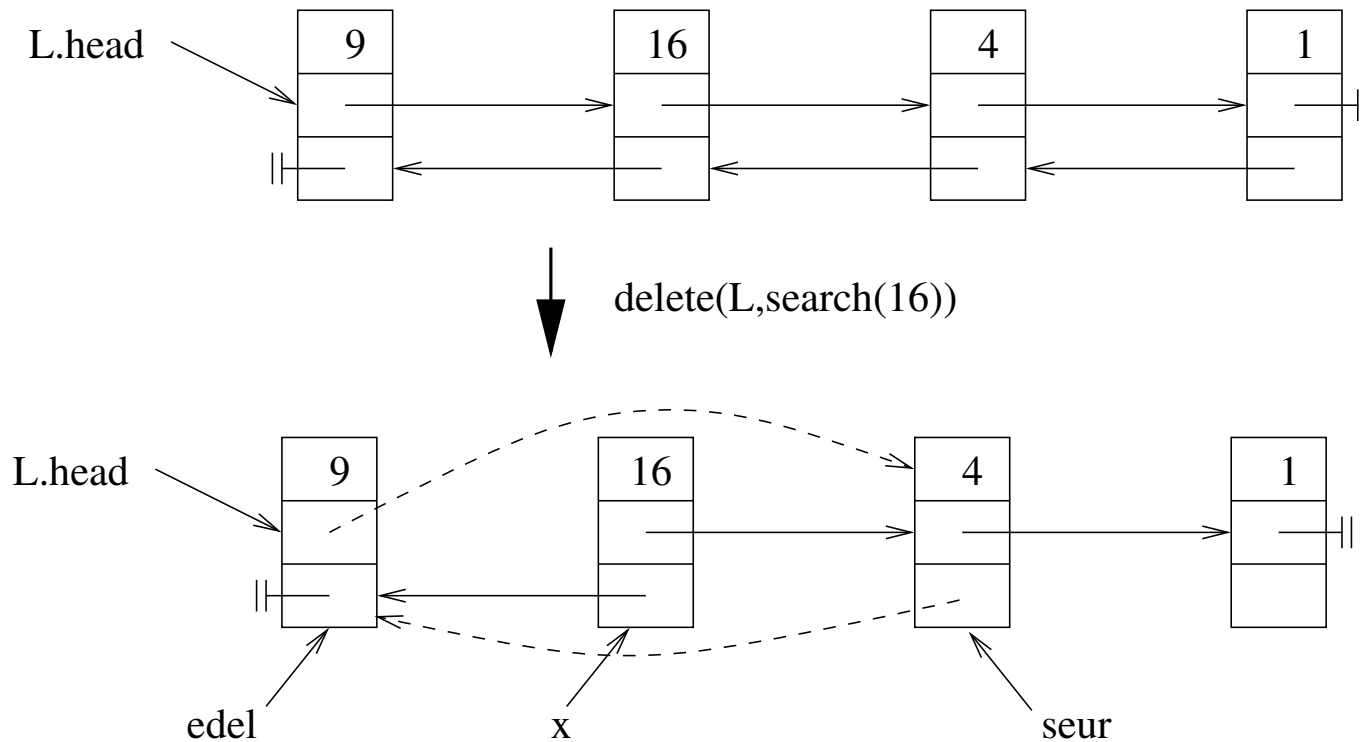
 edel.next = seur

 else L.head = seur // edellistä ei ollut, eli poistettu oli listan ensimmäinen

 if seur \neq NIL

 seur.prev = edel

- Seuraavalla sivulla on esimerkki, jossa poistettavalla on sekä edellinen että seuraava solmu
- On hyödyllistä simuloida myös erikoistapaukset: ei seuraavaa, ei edellistä ja poistettava listan ainoa solmu



- Listan pituudesta riippumatta operaatio suorittaa vakiomäärän komentoja, eli aikavaativuus $\mathcal{O}(1)$
- Huom: **delete**-operaatio saa parametrinaan viitteen poistettavaan solmuun eli jos halutaan poistaa listalta esim. luku 4, on viite poistettavaan solmuun selvitettävä **search**-operaatiolla: **delete**(L,**search**(4))
- Koska **search**-operaation aikavaativuus on $\mathcal{O}(n)$, on myös **delete**(L,**search**(x)):n aikavaativuus $\mathcal{O}(n)$

- Entä loput sivulla 102 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 viittaa vuorossa olevaan listasolmuun
- Muuttuja pienin muistaa, mikä solmu sisältää siihen mennessä pienimmän vastaantulleen avaimen
- Operaatio käy läpi listan kaikki alkion 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 avaimen 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 linkitettyä listana ole järkeä jos listalle talletettävien alkioden määrä on suuri

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

search(S, k) $\mathcal{O}(n)$

insert(S, x) $\mathcal{O}(1)$

delete(S, x) $\mathcal{O}(1)$

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

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

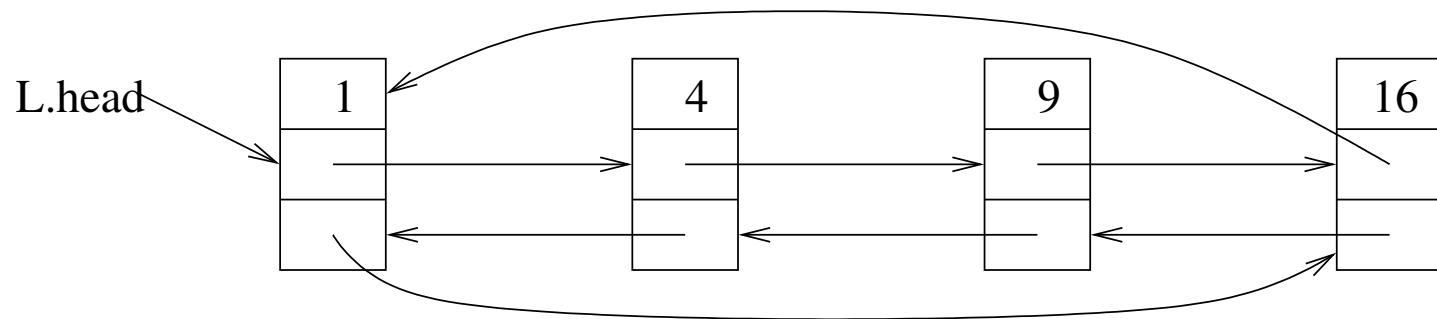
succ(S, x) $\mathcal{O}(n)$

pred(S, x) $\mathcal{O}(n)$

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

Rengaslista

- Jos talletamme luvut listalle **suuruusjärjestyksessä**, **insert**-operaation vaativuus huononee luokkaan $\mathcal{O}(n)$ mutta operaatiot **min**, **succ** sekä pienillä modifikaatioilla myös operaatiot **max** ja **pred** saadaan vakioaikaisiksi
- **insert** siis ei lisää uusia alkioita aina listan ensimmäiseksi alkioiksi, vaan etsii niille oikean paikan siten, että lisäyksen jälkeenkin listan alkioit ovat suuruusjärjestyksessä
- Yksi tapa operaatioiden **max** ja **pred** saamiseksi vakioaikaiseksi on käyttää **rengaslistaa**, missä alkioit 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 alkioden 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 normaalien 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 vakioaikainen, sillä listan pituus ei vaikuta suoritettavien käskyjen määrään

- Alkioiden etsintä järjestetyltä rengaslistalta:

search(L,k)

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

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

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

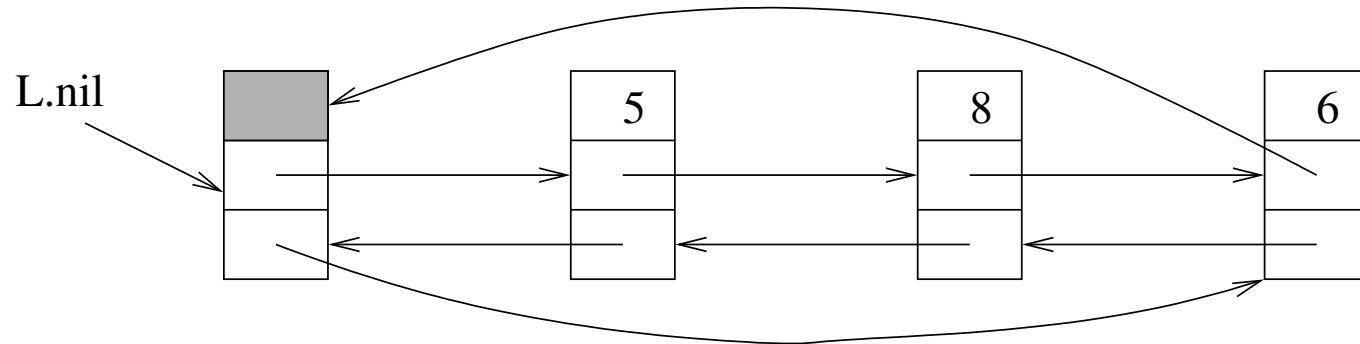
search (S, k)	$O(n)$
insert (S, x)	$O(n)$
delete (S, x)	$O(1)$
min (S)	$O(1)$
max (S)	$O(1)$
succ (S, x)	$O(1)$
pred (S, x)	$O(1)$

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

Tunnussolmullinen lista

- Joitakin operaatiota mutkistaa hieman erikoistapauksen **onko poistettava alkio listan alussa** huomioiminen
- Yksi variaatio linkitetyistä listoista on **tunnussolmullinen rengaslista**
 - nyt listan alussa tunnussolmu, eli solmu missä ei säilytetä tietoa
 - listan L attribuutti $L.nil$ viittaa listan tunnussolmuun
 - listan ensimmäinen alkio löytyy viitteen $L.nil.next$ päästä
 - listan viimeinen alkio löytyy viitteen $L.nil.prev$ päästä
- Huom: Jotta seuraavassa esitettävät operaatiot toimisivat, tyhjällä listalla täytyy olla: $L.nil.next = L.nil.prev = L.nil$
eli tyhjässä listassa tunnussolmun seuraavaksi ja edeltäväksi solmuksi on asetettu tunnussolmu itse
- Käsitellään seuraavassa tunnussolmullisen listan variaatiota joka ei edellytä alkiolle 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 tarvitse huomioida
- Vakuuta itsellesi simuloimalla, että **insert** toimii myös erikoistapauksissa
- Alkion etsintä tunnussolmullisesta rengaslistasta

search(L,k)

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

- Listoista on muitakin variaatioita: tunnussolmuton rengaslista, yhteen suuntaan linkitetyt listat sekä tunnussolmulla että ilman, tavallisena tai rengaslistana . . .
- Kuten jo todettiin, määrittelemämme linkitetty lista ei tarkalleen ottaen toteuta joukkoa niin kuin se käsitetään matemaattisesti sillä joukossahan tietty alkio voi esiintyä vaan kertaalleen mutta mikään ei estä saman alkion laittamista useaan kertaan listalle
- Linkitetty lista on yksinkertainen mutta useimmissa tilanteissa suhteellisen tehoton tietorakenne, eli onko listalle ylipäätään käyttöä?
- Jos käsiteltävä aineisto on pieni, riittää lista varsin hyvin
- Lista on käyttökelpoinen komponentti tietyissä muissa tietorakenteissa kuten hajautusrakenteissa
- Jos listan hitaita operaatioita (esim. **search**) tarvitaan äärimmäisen harvoin ja tilanteissa, joissa suoritus aika ei ole kovin kriittinen, voi lista olla järkevä vaihtoehto sen yleisen keveyden takia

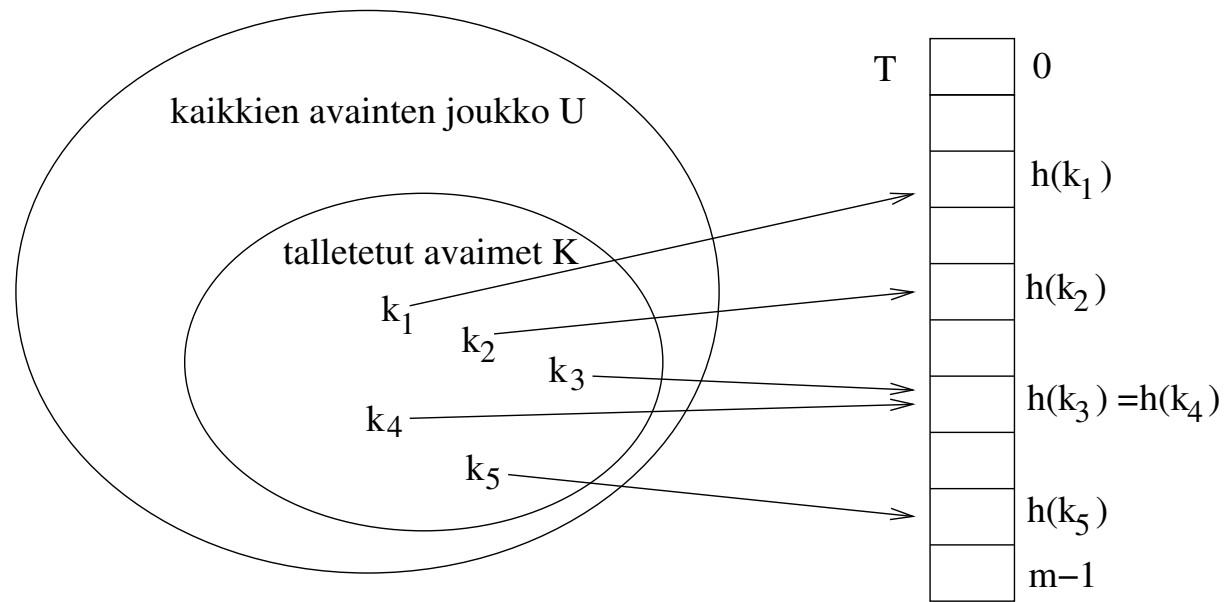
- Yhteenvetona voi todeta, että meillä on 16 eri variaatiota linkitetystä listasta, riippuen seuraavista valinnoista:
 - yhteen suuntaan linkitetty TAI kahteen suuntaan linkitetty
 - tunnussolmullinen TAI tunnussolmuton
 - tavallinen lista TAI rengaslista
 - järjestetty TAI järjestämätön

5. Hajautus

- Tarkastellaan edelleen sivulla 158 esitellyn joukkotietotyypin toteuttamista
- Useissa sovelluksissa riittää että operaatiot **insert**, **delete** ja **search** toimivat nopeasti
 - esim. sivun 27 puhelinluetteloesimerkissä ei puhelinnumeron mukaan järjestetyssä indeksirakenteessa luultavasti ole kovin usein käyttöä operaatioille **min**, **max**, **succ** ja **pred**
 - sen sijaan nimen mukaan järjestetyssä indeksissä operaatioille **min** ja **succ** voi olla käyttöä, jos on tarvetta esim. listata kaikki nimet aakkosjärjestyksessä
- Tapauksissa, joissa **insert**-, **delete**- ja **search**-operaatiot riittävät, **hajautusrakenne** on vartenotettava tapa joukon toteutukselle

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

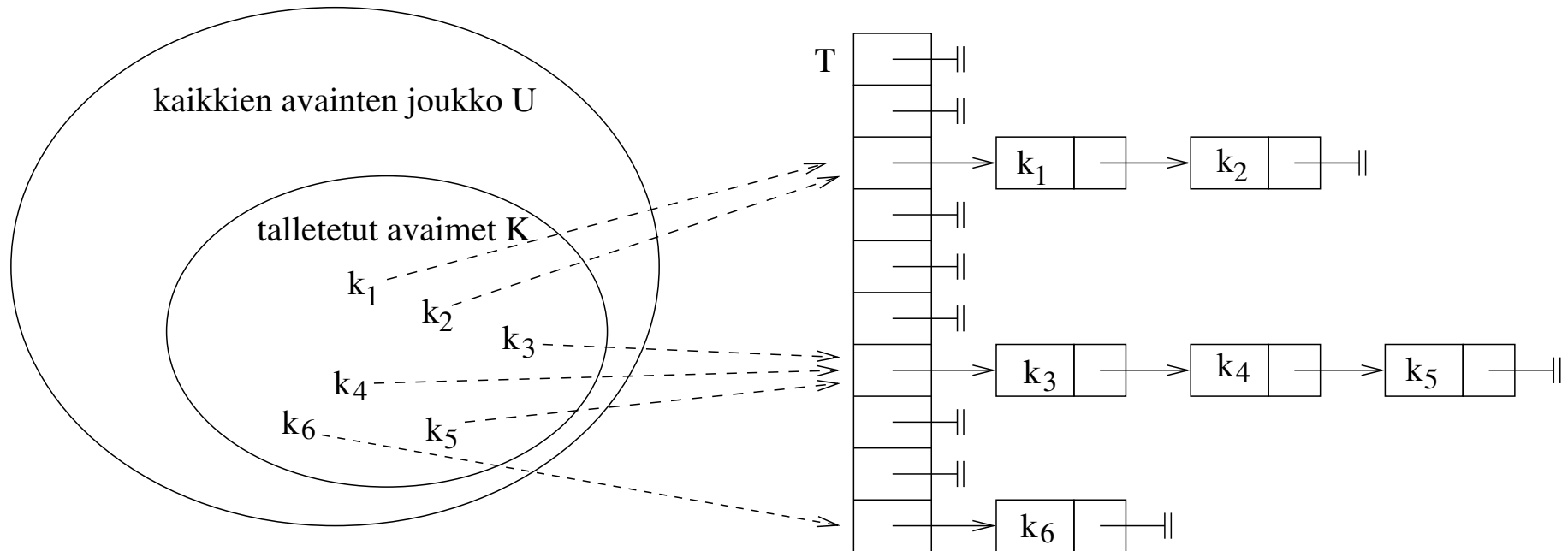
- Merkitään kaikkien mahdollisten avainten joukkoa U :lla
 - esim. opiskelijarekisterissä tämä olisi kaikki mahdolliset opiskelijanumerot tai puhelinluettelossa kaikki mahdolliset puhelinnumerot tai kaikki mahdolliset nimet
- Otetaan käyttöön m -paikkainen hajautustaulukko (engl. hash table) T , joka on indeksoitu alkaen nolasta
- Koska yleensä U :n koko on erittäin suuri (tai ääretön), on $m < |U|$
- Määritellään hajautusfunktio (engl. hash function) $h : U \rightarrow \{0, \dots, m - 1\}$, joka siis kuvaa jokaisen avaimen jollekin kokonaisluvulle väliltä $0, \dots, m - 1$
- Toisin sanoen, hajautusfunktio kuvaa jokaisen avaimen jollekin taulukon T indeksille
- Sanotaan että $h(k_i)$ on avaimen k_i osoite tai kotiosoite
- Ideana on nyt että avain k_i talletetaan taulukon T paikkaan $h(k_i)$, eli $T[h(k_i)] = k_i$



- Kuten kuva yllä kertoo, voi käydä niin että kaksi avainta saavat saman osoitteen eli joillekin avaimille k_3, k_4 voi olla $h(k_3) = h(k_4)$
- Tällöin sanotaan että avainten k_3 ja k_4 välillä sattuu yhteentörmäys (engl. collision)
- Oleellista onkin suunnitella hajautusfunktio h sellaiseksi, että yhteentörmäyksiä tapahtuu mahdollisimman vähän. Palaamme hieman myöhemmin hajautusfunktion suunnitteluun
- Oli hajautusfunktio miten hyvä tahansa, ei jokaiselle avaimelle yleensä riitä omaa osoitetta

Yhteentörmäykset ylivuotoketjuun

- Yhteentörmäysten ratkaisemiseen on olemassa kaksi erilaista strategiaa. Käsitellään ensin **ketjutus** (engl. chaining), jossa yhteentörmäävät avaimet talletetaan linkitettyihin listoihin
- Alla olevassa kuvassa hajautustaulukon alkio i sisältää viitteen osoitteen i saavien avainten sisältämään listan



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

- Esitetään nyt toteutus ketjutuksella varustettuun hajautukseen käyttäen linkitettyjä listoja (ks. luku 3)
- T on nyt taulukko joka on määritelty välille $[0, \dots, m - 1]$, 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 $x = \text{list-search}(L, k)$

3 **return** x

- rivillä 1 selvitetään missä taulun indeksiin liittyvässä listassa on avaimen paikka
- rivillä 2 kutsutaan sivulla 193 määriteltyä listan etsintäoperaatiota selvittämään löytyykö etsittävää avainta
- operaatio palauttaa viitteen siihen listasolmuun joka sisältää etsityn avaimen

- Avaimen lisääminen hajautusrakenteeseen

hash-insert(T, k)

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

- rivillä 1 selvitetään missä taulun indeksiin liittyvässä listassa on avaimen paikka
- rivillä 2 kutsutaan sivulla 194 määriteltyä listan lisäysoperaatiota laittamaan avain paikoilleen

- Avaimen poistaminen hajautusrakenteesta

hash-delete(T, x)

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

- syötteenä siis viite poistettavan avaimen sisältämään listasolmuun, jos viitettä ei ole tallessa, on se haettava ensin **hash-search**-operaatiolla
- rivillä 1 selvitetään missä taulun indeksiin liittyvässä listassa poistettava avain on
- rivillä 2 kutsutaan sivulla 196 määriteltyä listan poisto-operaatiota, joka poistaa avaimen listalta

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

- Analysoidaan hakua keskimääräisessä tapauksessa
- Oletetaan että m -kokoiseen hajautustauluun on talletettu n avainta, ja että satunnaisella avaimella on sama todennäköisyys päätyä mille tahansa ylivuotolistalle
(Hyvin valitulla hajautusfunktiolla tilanne on ainakin suurin piirtein tämä.)
- Hajautustaulun tämänhetkinen täyttösuhde (engl. load factor) on $\alpha = n/m$, joka on samalla keskimääräinen ylivuotoketjun pituus

- Avaimen haku hajautusrakenteesta vie aikaa keskimäärin $\mathcal{O}(1 + \alpha)$

- Todistushahmotelma (jossa todennäköisyyslaskelmat esitetään vain intuitiivisella tasolla):

Todistus jakaantuu kahteen osaan: tuloksettomaan ja tuloksekkaaseen hakuun.

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

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

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

- Jos hajautustaulukon koko on suoraan verrannollinen talletettavien avaimien lukumäärään, operaatiot hajautustaulussa ovat keskimäärin vakioaikaisia
 - hakuoperaatio `search` vie keskimäärin vakioajan jos avaimet ovat jakautuneet tasaisesti ylivuotolistoille
 - **hash-insert** ja **hash-delete** ovat vakioaikaisia myös pahimmassa tapauksessa
- Useissa sovelluksissa tiedetään suunnilleen talletettavien avaimien lukumäärän yläraja. Tällöin on helppo valita hajautusrakenteen koko sopivasti
- Edellä taulukon kussakin paikassa on vain viite listaan ([suora ketjutus](#)), mutta vaihtoehtoisesti voi tehdä niin, että taulukossa itsessään varataan tilaa yhdelle avaimelle osoitetta kohti, ja vasta yhteentörmäysten sattuessa aletaan rakentaa linkitettyä listaa ([erillinen ketjutus](#))

Hajautusfunktion valinta

- Tavoitteena on, että hajautusfunktion arvo on laskettavissa nopeasti ja funktio jakaa avaimet tasaisesti hajautusalueelle
- Hajautusfunktio olettaa yleensä että avain on kokonaisluku. Jos avain on kuitenkin esim. merkkijono, se on ensin muutettava kokonaisluvuksi
olkoon avain merkkijono $a_1a_2 \dots a_n$, oletetaan että funktio $ascii(a)$ palauttaa merkin a ascii-koodin, joka on kokonaisluku väliltä 0–127
eräs tapa muuttaa merkkijono kokonaisluvuksi k on seuraava
$$k = ascii(a_1) + 128 \cdot ascii(a_2) + 128^2 \cdot ascii(a_3) + \dots + 128^{n-1} \cdot ascii(a_n)$$
- Tähän sisältyy muutama ongelma
 - luvusta tulee suuri ja seurauksena saattaa olla aritmeettinen ylivuoto
 - ongelma korjautuu tulkitsemalla luku etumerkittömäksi kokonaisluvuksi ja yhdistämällä se seuraavan sivun jakolaskumenetelmään
 - toisaalta jos merkkijono on pitkä, voi merkkijonoa vastaavan kokonaislukuarvon laskeminen kestää kauan
 - merkkijonon jokaista merkkiä ei ole välttämätöntä huomioida, voidaan esim. ottaa mukaan vain joka toinen merkki, viisi ensimmäistä merkkiä, jne.
- Seuraavassa käytännössä toimivaksi osoittautuneita hajautusfunktioita

Jakojäännösmenetelmä

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

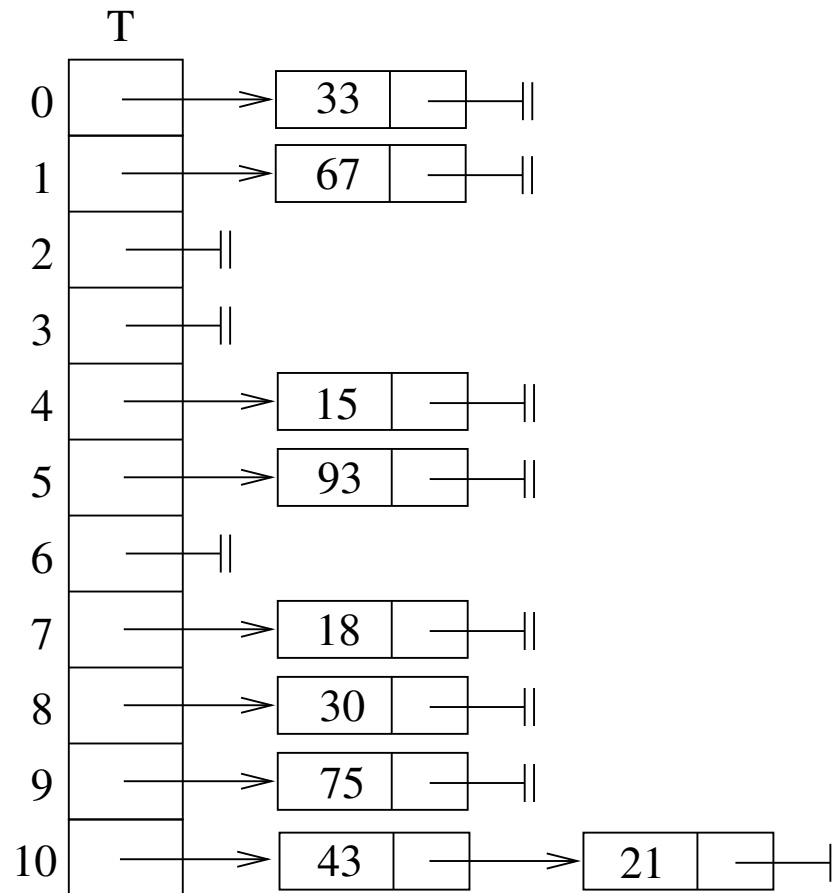
missä m on alkuluku, joka ei ole lähellä mitään 2:n potenssia

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) sattuisivat 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 esim. $m = 701$ (Lähimmät kakkosen potenssit ovat $2^9 = 512$ ja $2^{10} = 1024$)
- Nyt m on alkuluku joka ei ole lähellä 2:n potenssia ja täyttöasteeksi tulee $2000/701 \approx 2.85$
- Esim. 2: Oletetaan että käytössä hajautustaulukko jonka koko 11, hajautusfunktiona siis $h(k) = k \bmod 11$

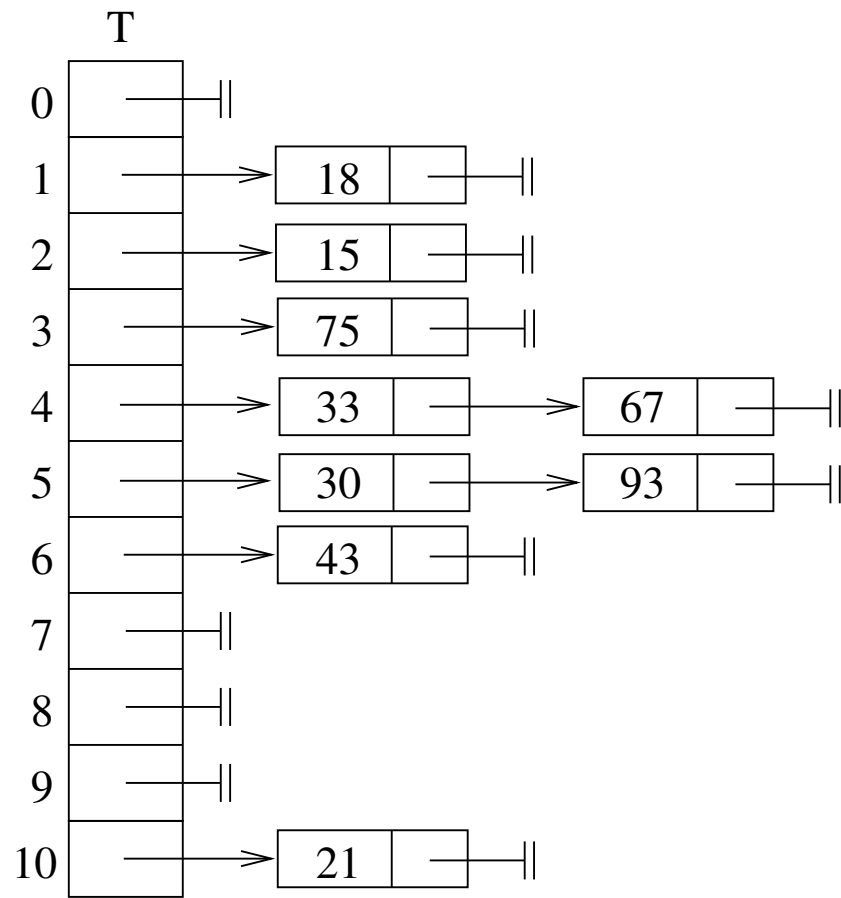
- Hajautetaan avaimet 75, 21, 43, 15, 18, 33, 30, 67 ja 93, eli nyt
 $h(75) = 9$, $h(21) = 10$, $h(43) = 10$, $h(15) = 4$, $h(18) = 7$,
 $h(33) = 0$, $h(30) = 8$, $h(67) = 1$, $h(93) = 5$
- Tuloksena



Kertolaskumenetelmä

- Olkoon A jokin luku $0 < A < 1$
- Merkitään $\lfloor x \rfloor$:llä luvun x kokonaislukuosaa ja $\text{fr}(x)$:lla luvun x desimaaliosaa, eli esim $\lfloor 3,14159 \rfloor = 3$ ja $\text{fr}(3,14159) = 0,14159$
- $h(k) = \lfloor m \cdot \text{fr}(Ak) \rfloor$
- Suositeltu valinta on $A = \frac{\sqrt{5}-1}{2} = 0,6180339887\dots$, josta voidaan osoittaa, että se tietyssä mielessä maksimoi peräkkäisten avainten k ja $k+1$ sijoituspaikkojen välimatkan
- Parametri m ei suuremmin vaikuta hajautuksen tasaisuuteen. Voidaan valita esim. $m = 2^p$, jollekin kokonaisluvulle p , jolloin laskenta bittitasolla on helppoa (ks. Cormen luku 11.3.2)
- Esim. oletetaan että käytössä on hajautustaulukko jonka koko on 11, ja hajautusfunktiona $h(k) = \lfloor 11 \cdot \text{fr}(0,618 \cdot k) \rfloor$
- Avaimet 75, 43, 21, 15, 18, 33, 30, 67 ja 93, nyt
 $h(75) = 3, \quad h(43) = 6, \quad h(21) = 10, \quad h(15) = 2, \quad h(18) = 1$
 $h(33) = 4, \quad h(30) = 5, \quad h(67) = 4, \quad h(93) = 5$

- Tuloksena

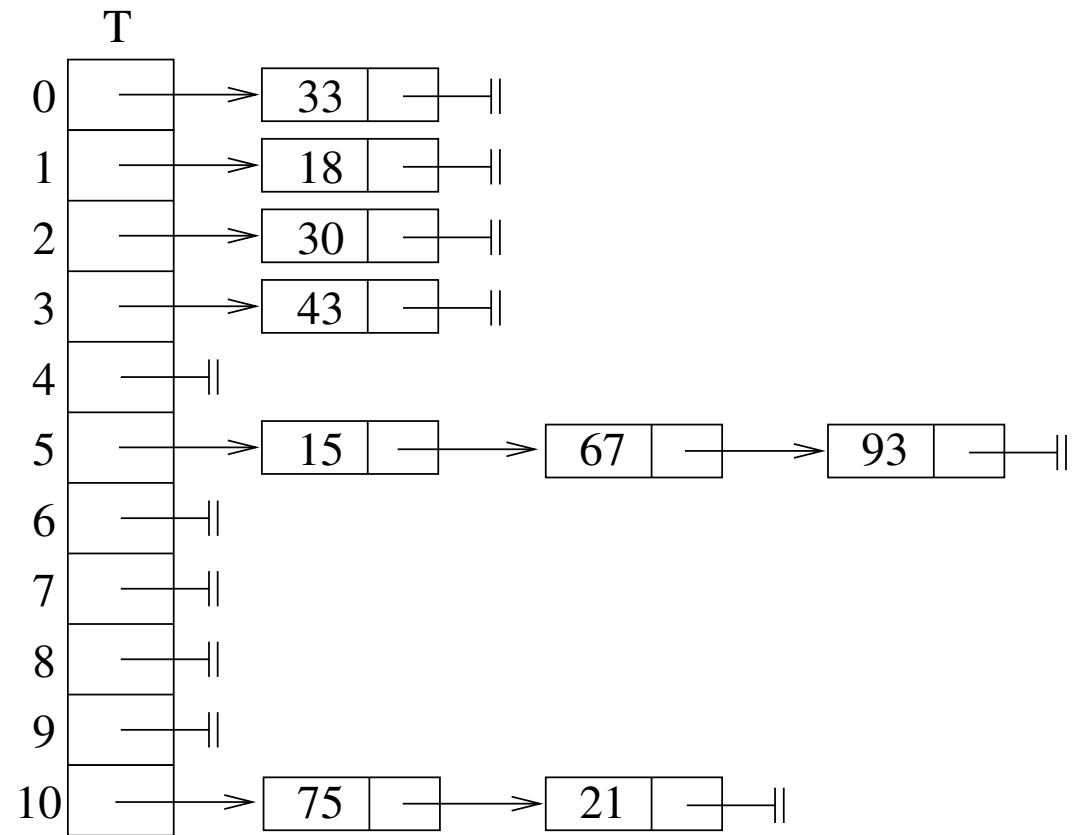


- Hajautusfunktio pitäisi valita siten, että hajautusosoitteiden $h(k)$ jakauma olisi mahdollisimman lähellä joukon $\{0, \dots, m - 1\}$ tasaista jakaumaa
- Tätä kriteeriä on mahdoton arvioida, jos ei tiedetä avainten k jakaumaa (ja vaikka tiedettäisiin, se voi olla käytännössä hyvin vaikeaa)
- Ohjelmoija voi tyhmyyttään valita sovellusta ajatellen juuri väärän hajautusfunktion
- Jos tehdään huono valinta hajautusfunktiksi, 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**
- Universaalihajautuksessaakin voi joskus syntyä sovelluksen käsiteltävään dataan nähden huono hajautusfunktio
- Keskimäärin näin käy kuitenkin erittäin harvoin
- Universaalihajautus toimii odotusarvoisesti nopeasti riippumatta hajautettavien avaimien jakaumasta

Universaalihajautus

- Valitaan alkuluku p joka on vähintään yhtä suuri kuin talletettavien avainten lukumäärä n
- Valitaan satunnaisesti kaksi kokonaislukua a ja b väliltä $1 \leq a < p$ ja $0 \leq b < p$
- Muodostetaan hajautusfunktio seuraavasti: $h(k) = ((ak + b) \bmod p) \bmod m$
- Perusteluja näin muodostetun hajautusfunktion hyvydestä löytyy Cormenin luvusta 11.3.3
- Esim. oletetaan jälleen että käytössä on hajautustaulukko jonka koko on 11
- Oletetaan että talletettavia avaimia korkeintaan 53 eli valitaan $p = 53$, ja valitaan satunnaisesti $a = 31$, $b = 17$
hajautusfunktiona siis $h(k) = ((31 \cdot k + 17) \bmod 53) \bmod 11$
- Avaimet 75, 43, 21, 15, 18, 33, 30, 67 ja 93, nyt
 $h(75) = 10 \bmod 11 = 10$, $h(43) = 25 \bmod 11 = 3$, $h(21) = 32 \bmod 11 = 10$
 $h(15) = 5 \bmod 11 = 5$, $h(18) = 45 \bmod 11 = 1$, $h(33) = 33 \bmod 11 = 0$
 $h(30) = 46 \bmod 11 = 2$, $h(67) = 27 \bmod 11 = 5$ ja $h(93) = 38 \bmod 11 = 5$

- Tuloksena



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

Uudelleenhajautus

- Kertauksena sivulla 221:
 - jos hajautustaukon koko m valitaan suoraan verrannolliseksi talletettujen avaimien lukumäärän n , ylivuotoketjujen keskimääräinen pituus on $\alpha = n/m = \mathcal{O}(1)$ eli vakio hajautusrakenteeseen talletettävien avaimien lukumäärän suhteen
- Eli jos hajautustaulukon koko on suoraan verrannollinen talletettävien avaimien lukumäärään, ovat operaatiot hajautustaulussa vakioaikaisia
- Jos hajautustaulun koko on vakio, esim. 1000, ja rakenteeseen talletettävien alkioiden määrä n kasvaa rajatta, on yhden ylivuotoketjun keskimääräinen pituus $n/1000$, eli $\mathcal{O}(n)$
- Siis jos talletettävien alkioiden määrä n kasvaa rajatta, mutta taulun koko m on vakio, ylivuotoketjujen keskimääräiseksi pituudeksi tulee $\mathcal{O}(n)$ ja etsintä hajautusrakenteesta vie lineaarisen ajan

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

- Monisteen sivuilla 170–171 tarkasteltiin taulukon laajentamisen aikavaativuutta
- Ilmeni, että jos taulukon koko lisäyksen yhteydessä tarvittaessa kaksinkertaistetaan, ei kasvatus lisää yhden operaation keskimäärin vievää aikavaativuutta kuin vakion verran
- Samantapainen analyysi osoittaa, että vaikka uudelleenhajautus eli hajautustaulun kasvattaminen on raskas operaatio, niin jos ajatellaan pidempiä sarjoja hajautusrakenteelle suoritettuja operaatiota, yhden operaation keskimääräinen aikavaativuus on silloin tällöin tapahtuvasta uudelleenhajautuksesta huolimatta vakio
- Jos uudelleenhajautusta ei suoriteta ja talletettavien alkioiden määrä kasvaa jatkuvasti, **hash-search**-operaation keskimääräinen aikavaativuus ei enää säily vakiona
- Eli ajoittaisesta raskaudesta huolimatta uudelleenhajautus kannattaa jos hajautusrakenteeseen talletettavien alkioiden lukumäärällä ei ole ylärajaa

Avoim hajautus

- Ketjutuksen rinnalla toinen tapa ratkaista yhteentörmäävien solmujen ongelma on **avoin hajautus** (engl. open addressing)
- Avoimessa hajautuksessa **kaikki** avaimet talletetaan hajautustauluun
- Jos paikka $T[h(k)]$ on varattu, etsitään avaimelle k jokin muu paikka taulusta
- Uusikin paikka voi olla varattu, tällöin jatketaan etsimistä edelleen
- Ne paikat joihin avainta k yritetään laittaa muodostavat k :n **kokeilujonon** (engl. probe sequence)
- Järjestyksessään j :s kokeilu kohdistuu paikkaan
$$h(k, j) = (h'(k) + s(j, k)) \bmod m,$$
missä h' on "normaali" hajautusfunktio ja s on kokeilufunktio
- Vaaditaan että jokaiselle avaimelle k kokeilujono $h(k, 0), h(k, 1), \dots, h(k, m - 1)$ muodostaa jonon $0, 1, \dots, (m - 1)$ permutaation, eli toisin sanoen kokeilujonon on kokeiltava jokaista hajautustaulun indeksiä kertaalleen
- Avaimen k lisääminen tapahtuu siten että ensin yritetään laittaa avain paikkaan $h(k, 0)$, jos tämä paikka on täysi, yritetään paikkaa $h(k, 1)$, jne

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

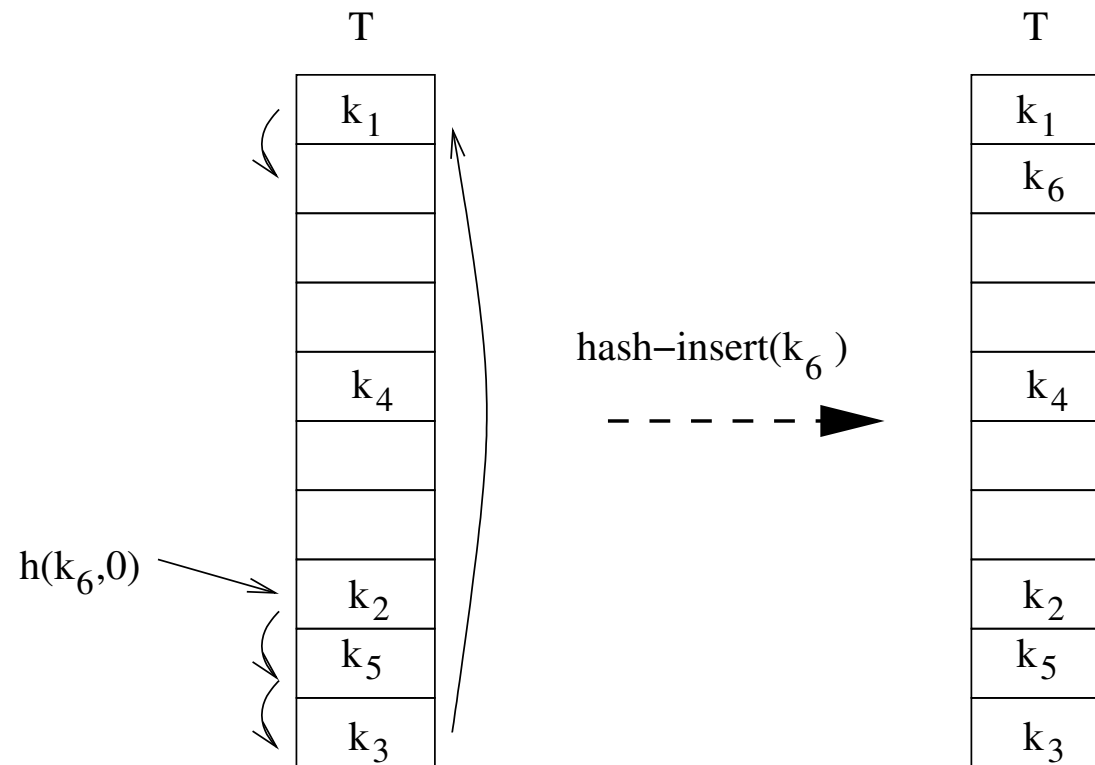
hash-insert(T,k)

```

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

```

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



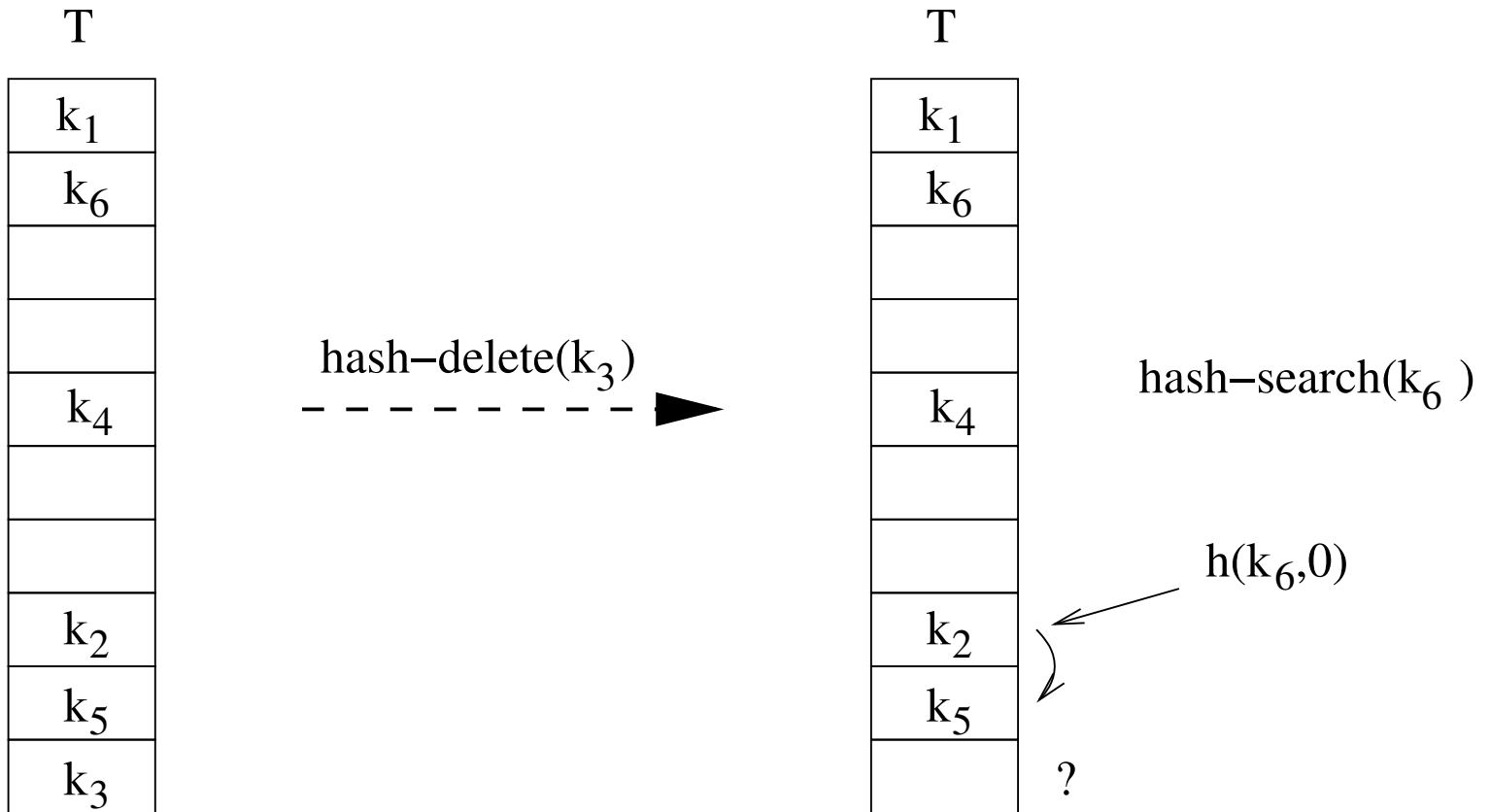
- Avaimen etsiminen hajautustaulusta

hash-search(T,k)

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

- Etsitään kokeilujonoa $h(k, 0), h(k, 1), \dots, h(k, m - 1)$ 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-arvoiseksi), muuten reitti joihinkin avaimiin saattaa katketa:



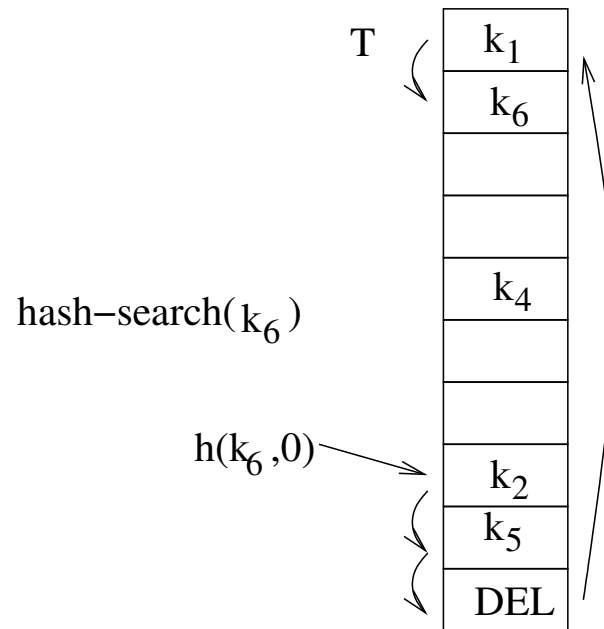
- Merkitään poiston yhteydessä poistettavan avaimen paikalle erityisarvo DEL niin avaimen etsiminen toimii taas

hash-delete(T,k)

```

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

```



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

hash-insert(T,k)

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

- Avoimen hajautuksen poisto-operaatio on **laiska**, se ei vapauta muistialuetta, vaan jättää taulukkoon DEL-arvoisia kohtia jotka täyttävät hajautustaulukkoa
- DEL-kohdat voivat korvautua normaaleilla avaimilla **hash-insert**:in yhteydessä

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

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

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

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

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

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

- $h(66, 0) = 0$ ei käy, $h(66, 1) = 1$ ei vielä, mutta $h(66, 2) = 2$ onnistuu
- Lopuksi $h(92, 0) = 4$, ei käy, mutta $h(92, 1) = 5$ vapaa
- Tuloksena

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

- Entä jos haluamme vielä lisätä avaimen 29? $h(29, 0) = 7$ on varattu ja vapaa löytyy vasta 8. kokeilulla: $h(29, 7) = 3$
- Entä jos etsisimme taulukosta lukua 41? Etsintä etenisi paikasta $h(41, 0) = 7$ aina paikkaan $h(41, 7) = 3$ asti jonka jälkeen voidaan todeta että 41 ei ole taulukossa

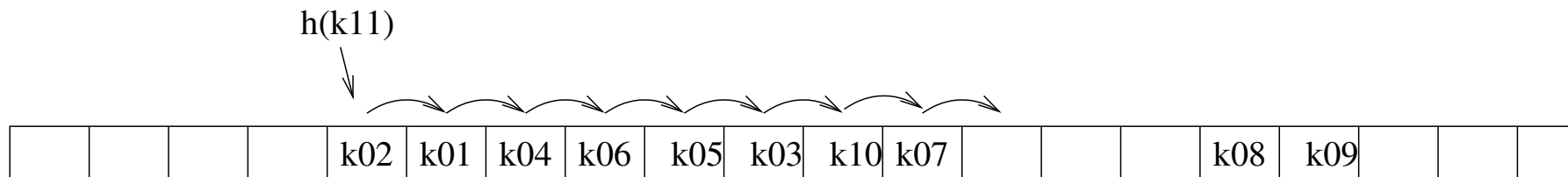
- Poistetaan taulukosta avaimet 21, 30 ja 75:

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

- Edelleen, luvun 41 etsintä etenisi paikasta $h(41, 0) = 7$ aina paikkaan $h(41, 7) = 3$ asti jonka jälkeen voidaan todeta että 41 ei ole taulukossa
- Luvun 29 lisäys löytäisi nyt 29:lle paikan toisella kokeilulla $h(29, 1) = 8$ sillä $T[8] = \text{DEL}$
- Taulukko luvun 29 lisäyksen jälkeen

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

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

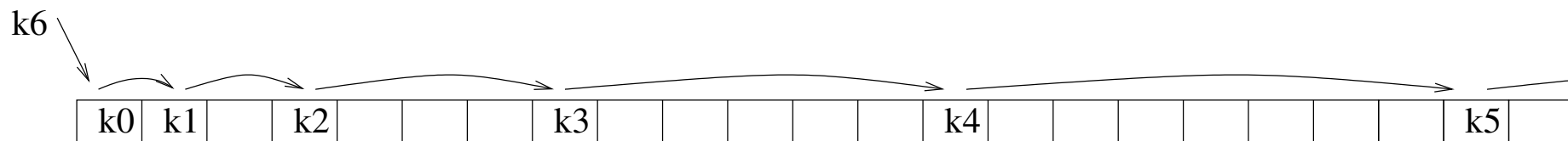


- Ilmiötä nimitetään **primääriseksi kasautumiseksi** (engl. primary clustering)
- Varattuja alueita muodostuu aina jonkin verran, koska yhteentörmäyksiä sattuu pienilläkin täyttöasteilla
 - **syntymäpäiväparadoksi**: jos taulukossa on $\sqrt{2m}$ avainta, todennäköisyys välttyä kokonaan yhteentörmäyksistä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

- Neliöinen kokeilu

- kokeilujono on nyt $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$, missä h' normaali hajautusfunktio, c_1 ja c_2 vakioita
- kokeilujonon ensimmäinen paikka on $h'(k) \bmod m$, toinen $(h'(k) + c_1 + c_2) \bmod m$, kolmas $(h'(k) + 2c_1 + 4c_2) \bmod m$, jne
- huom: kaikki valinnat vakioiksi c_1, c_2 ja m eivät tuota kokeilujonoa joka käy taulukon kaikki paikat läpi
yksi toimiva tapa valita vakiot on asettaa $c_1 = c_2 = 1/2$ ja valita hajautustaulun kooksi m 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... eli $h(k, 1) = h(k, 0) + 1$, $h(k, 2) = h(k, 0) + 3 = h(k, 1) + 2$, $h(k, 3) = h(k, 0) + 6 = h(k, 2) + 3 \dots$

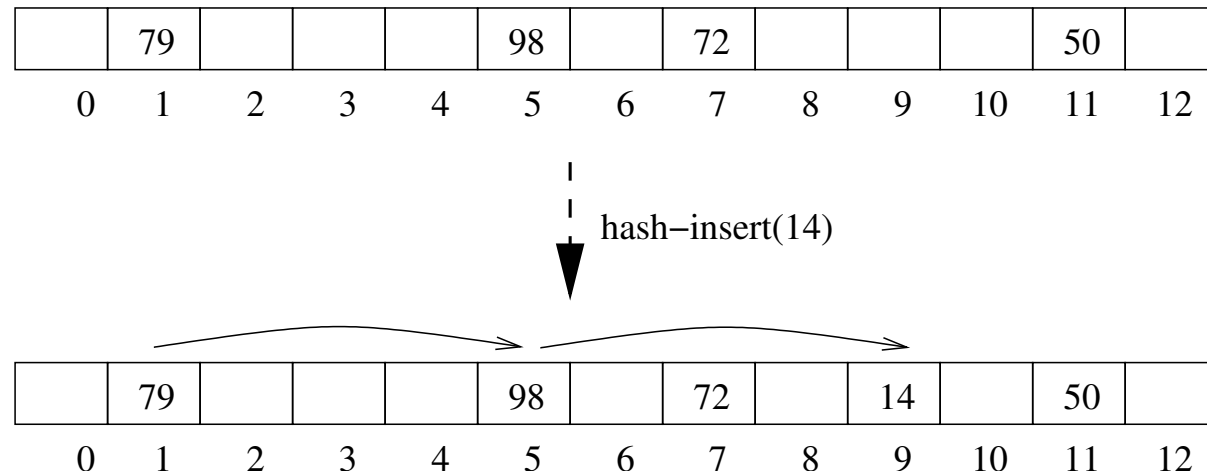
- Primäärисeltä kasaantumiselta vältytään, mutta nyt vaivana on sekundäärinen kasautuminen (engl. secondary clustering): jos $h(k_1, 0) = h(k_2, 0)$ niin k_1 :n ja k_2 :n kokeilujono on sama



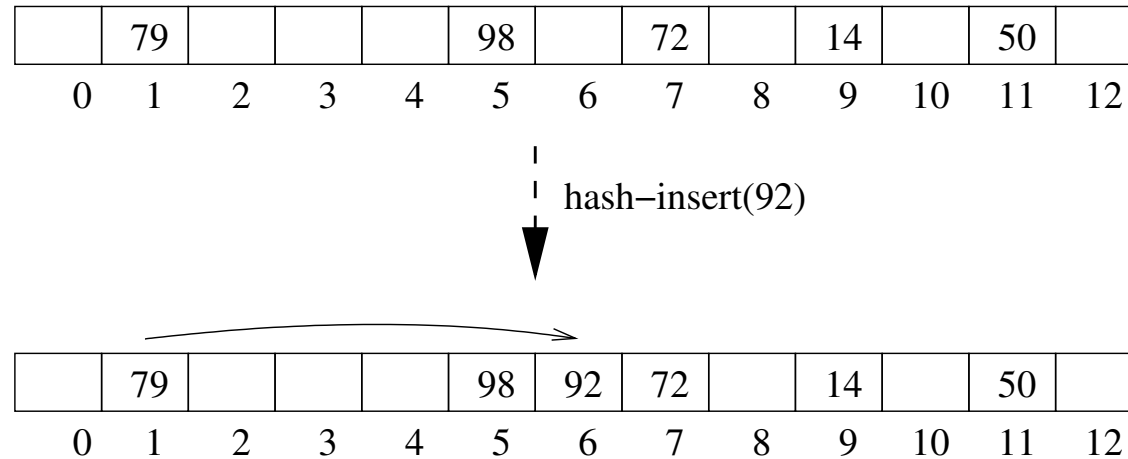
$$h(k_0, 0) = h(k_1, 0) = h(k_2, 0) = h(k_3, 0) = h(k_4, 0) = h(k_5, 0) = h(k_6, 0) = 1$$

- Kaksoishajautus (engl. double hashing)

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



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



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

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

- Avoimen hajautuksen tapauksessa kaikkien operaatioiden tehokkuus on riippuvainen avaimen löytymisen tehokkuudesta
- Seuraavassa taulukossa tuloksettoman ja tuloksellisen haun **keskimääräinen** pituus kullakin kokeilujonotyypillä suhteessa täyttösuhteeseen $\alpha = n/m$. Kaavoja ei tässä perustella tarkemmin, niiden taustalla oleva analyysi on esitetty Cormenin luvussa 11.4

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

- Seuraavalla sivulla vielä hakujen keskimääräisiä pituuksia muutamalla konkreettisella täyttösuhteen arvolla

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

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

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

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

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

Käytännöllisiä huomautuksia hajautuksesta

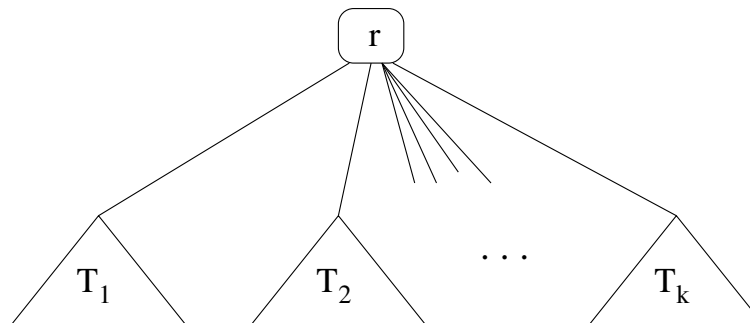
- Hajautusfunktioista jakojäännös menetelmä on yleensä hyvä
- Usein aineiston kanssa kannattaa tehdä muutamia kokeita ennen kuin käytettävä hajautusfunktio valitaan
- Yhteentörmäysstrategioista ketjutus on yleensä paras ratkaisu
 - ketjutus toimii hyvin vaikka hajautusalue onkin täynnä
 - poistot ketjutuksessa 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 pian nähtävää tasapainotettua hakupuuta
- **hash-search**-operaatio toimii vakioaikaisesti keskimääräisessä tapauksessa, eli jos halutaan takuu että pahin tapaus ei koskaan toteudu, on syytä käyttää tasapainoitettua hakupuuta

Java-API:n hajautusrakenteet

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

6. Hakupuut

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

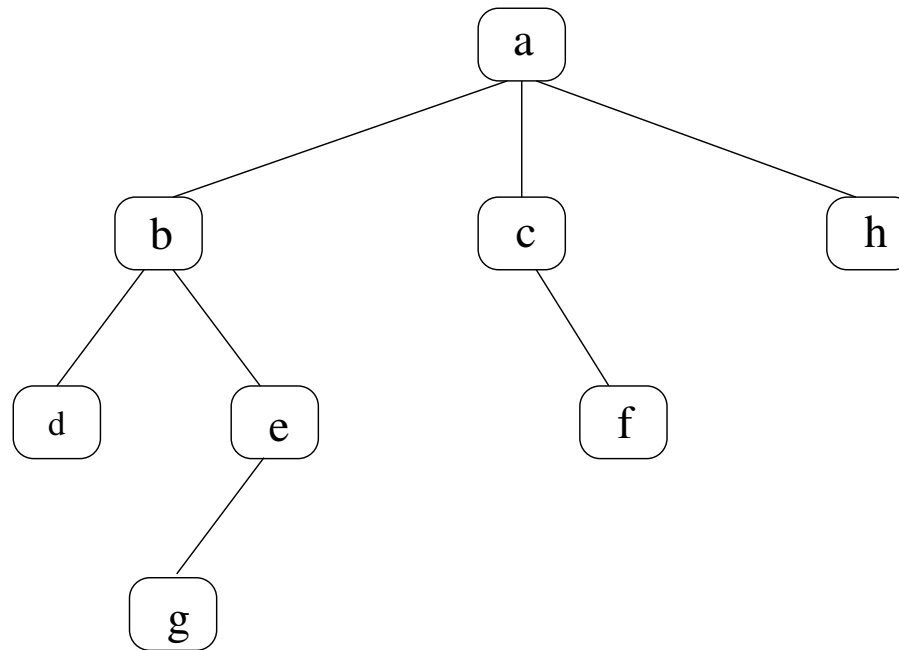


- Edellä annettu puun määritelmä on rekursiivinen, eli puu koostuu juuresta ja siihen liittyvistä alipuista, jotka ovat itsekin puita
- **Verkkojen** (eli graafien) teoriassa puu tarkoittaa yhtenäistä syklitöntä verkkoa. Jos verkkoteoreettiselle puulle on lisäksi nimetty juuri, se on jokseenkin sama käsite kuin tässä tarkasteltava. Palaamme verkkoihin myöhemmin kurssilla.
- Tietojenkäsittelytieteessä puu piirretään juuri ylöspäin
- Jos ei muuta mainita, niin tarkastelemme jatkossa **järjestettyjä** (ordered) puita, eli alipuiden järjestys vasemmalta oikealle on merkityksellinen
- Rekursiivinen määritelmä ei luultavasti ole helpoin tapa ymmärtää, mistä puissa on kysymys, mutta se on syytä pitää mielessä, sillä sama rekursiivinen rakenne toistuu esim. monissa puita käsittelevissä algoritmeissa

Määritellään muutamia puihin liittyviä käsitteitä

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

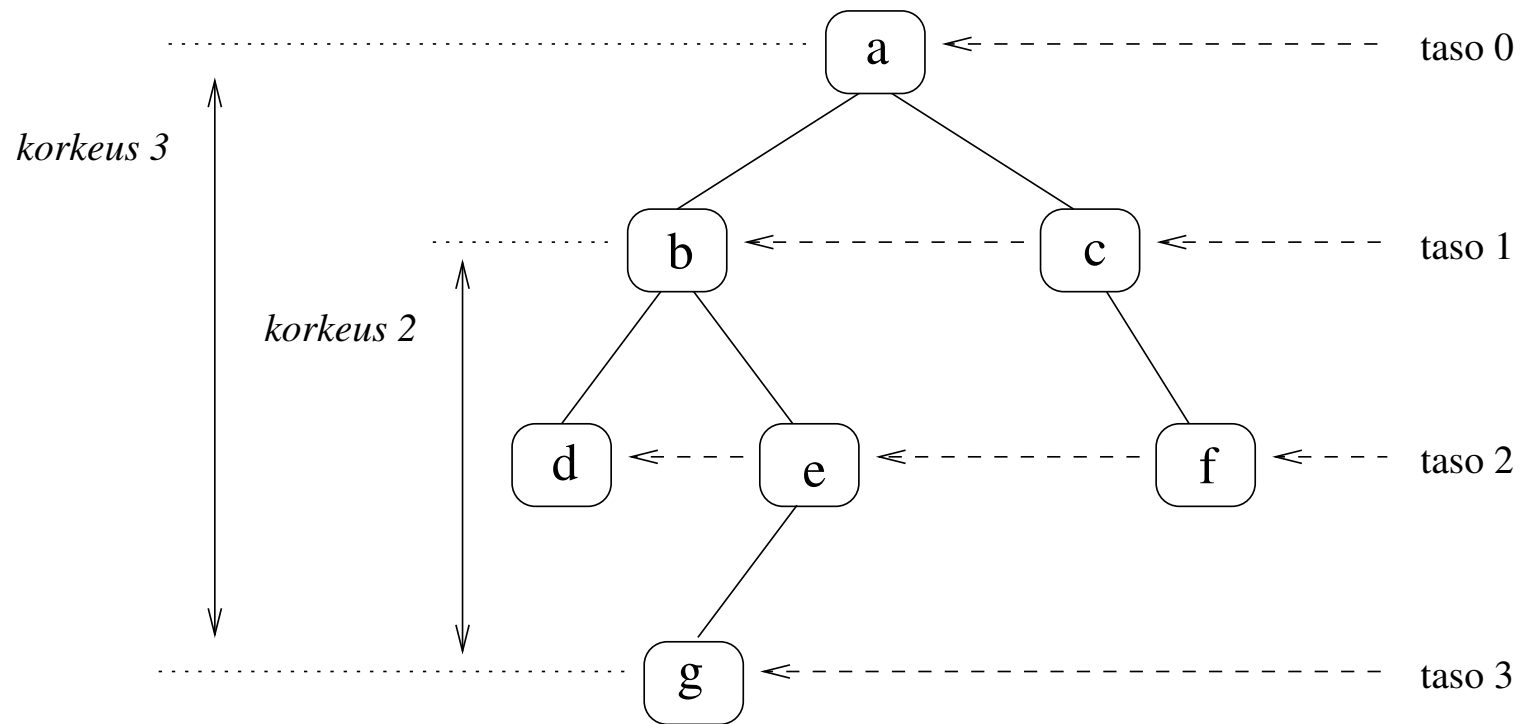
- Esim:



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

- Lisää määritelmiä:
 - **polku** (engl. path) solmusta x_1 solmuun x_k on jono solmuja x_1, x_2, \dots, x_k siten että x_i on x_{i+1} :n vanhempi kun $1 \leq i \leq k - 1$
 polun pituus on sen kaarien lukumäärä, esim. edellisen sivun kuvassa a, b, e, g on polku solmusta a solmuun g jonka pituus on 3
 Erikoistapaus: jokaisesta solmusta on nollan pituinen polku itseensä!
 - jos on olemassa polku solmusta x_1 solmuun x_2 , sanotaan että x_1 on solmun x_2 , **esi-isä** tai **edeltäjä** (engl. ancestor) ja x_2 on x_1 :n **jälkeläinen** tai **seuraaja** (engl. descendant)
 Jos $x_1 \neq x_2$, edeltäjäisyys ja jälkeläisyys ovat **aitoja** (engl. proper ancestor, proper descendant)
 esim. a, b ja e ovat solmun e edeltäjät, joista a ja b ovat aitoja edeltäjiä, solmun e seuraajia ovat e ja g , joista jälkimmäinen on aito seuraaja
 - solmun x **taso** (engl. depth) on polun pituus juuresta x :ään, juuren taso on 0. Joissain lähteissä tasosta käytetään termiä **syvyys**
 - solmun x **korkeus** (engl. height) on pisimmän x :stä lehteen vievän polun pituus, eli syvimmän lehden syvyys
 - **puun korkeus** on sen juuren korkeus
- Havainto: Koska juurta lukuunottamatta jokaisella solmulla on tasan yksi vanhempi, kaarten lkm = solmujen lkm – 1

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



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

Binääripuu

- Jos puun solmuilla on korkeintaan kaksi lasta, on kyseessä **binääripuu** (engl. binary tree)
 - binääripuun alipuiden juuria kutsutaan **vasemmaksi** ja **oikeaksi** lapseksi, solmun x vasempaan lapseen viitataan $x.left$ ja oikeaan $x.right$
 - solmusta $x.left$ alkavaa puuta kutsutaan solmun x **vasemmaksi alipuuksi** ja solmusta $x.right$ alkavaa x :n **oikeaksi alipuuksi**
 - edellisellä sivulla oleva puu on binääripuu, esim solmun b vasen lapsi on d ja oikea lapsi e , eli $b.left = d$ ja $b.right = e$
- Todistetaan seuraavaksi kaksi tärkeää binääripuita koskevaa tulosta
- **Lause 6.1:** Olkoon T binääripuu jonka korkeus on h . Tällöin
 - (1) puun tasolla i on enintään 2^i solmua
 - (2) puussa on enintään $2^{h+1} - 1$ solmua

Väitteen 1 todistus:

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

Tehdään todistus induktiolla tason numeron suhteen.

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

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

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

Väitteen 2 todistus:

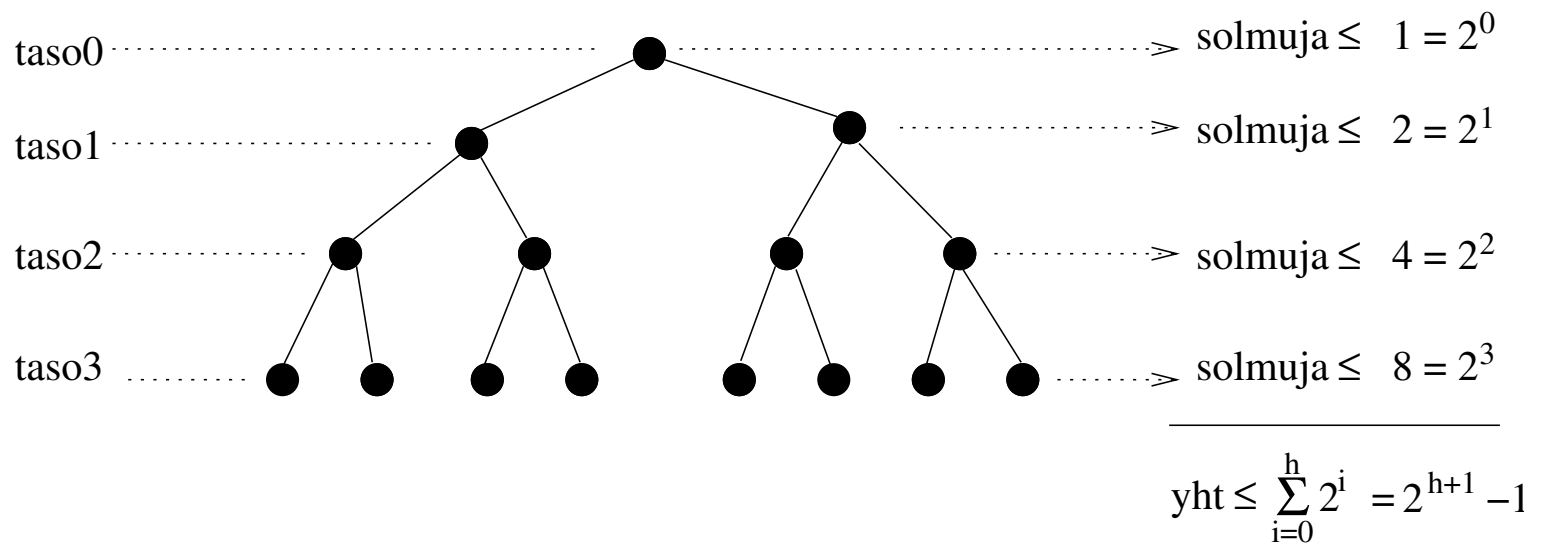
On siis näytettävä, että h :n korkuisessa puussa on enintään $2^{h+1} - 1$ solmua.

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

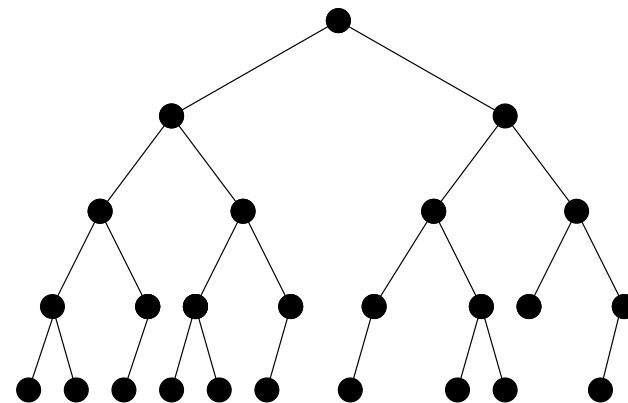
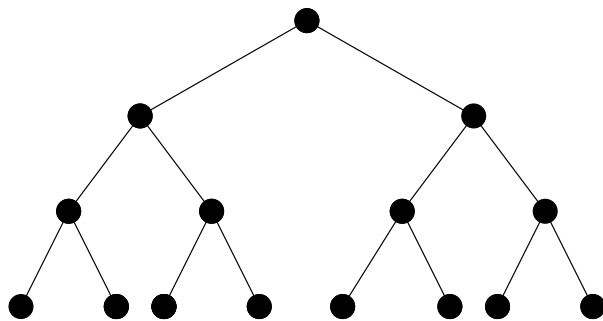
$$2^0 + 2^1 + 2^2 + \dots + 2^h = \sum_{i=0}^h 2^i.$$

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

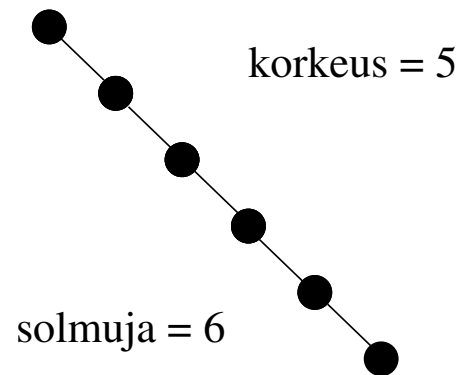
- Edellisen sivun todistuksien ideaa valottava kuva



- Binääripuu on **täysi** (engl. full), jos jokaisella solmulla on joko 0 tai 2 lasta
- Binääripuu on **täydellinen** (engl. complete), jos se on täysi ja kaikki lehdet ovat samalla tasolla
- Kuvassa vasemmalla täydellinen binääripuu, eli puu, joka on on "täysin mahdollinen" tietyn korkuinen puu
- Jos puu on viimeistä tasoa lukuunnottamatta täydellisen binääripuun kaltainen, voidaan puuta nimittää **melkein täydelliseksi**
- Kuvassa oikealla melkein täydellinen binääripuu, joka ei ole täysi



- Lausetta 6.1 soveltamalla on helppo huomata, että h :n korkeisella täydellisellä binääripuulla on täsmälleen 2^h lehteä ja $2^{h+1} - 1$ solmua
- Toisaalta binääripuussa, jonka korkeus on, h voi olla vähimmillään $h + 1$ solmua:



- Eli jos puun korkeus on h , niin:

$$h + 1 \leq \text{solmumäärä} \leq 2^{h+1} - 1$$

- Todistetaan seuraavassa vielä täsmällisesti, mikä on puun korkeus suhteessa solmujen lukumäärään

- **Lause 6.2:**

Olkoon T binääripuu jonka solmujen lukumäärä on n . Tällöin

- (1) puun korkeus on enintään $n - 1$
- (2) puun korkeus on vähintään $\log_2(n + 1) - 1$

Väitteen 1 todistus:

jos jokaisella ei-lapsisolmulla on ainoastaan yksi lapsi, on puu kuin lista ja korkeus silloin suurin mahdollinen eli $n - 1$

Väitteen 2 todistus:

binääripuun korkeus on pienin mahdollinen jos puu on täydellinen.

Merkitään puun korkeutta h :lla. Lauseen 6.1 nojalla täydellisen puun solmumäärä n on $2^{h+1} - 1$, eli $n + 1 = 2^{h+1}$.

Otetaan kaksikantainen logaritmi molemmilta puolilta:

$$\log_2(n + 1) = \log_2 2^{h+1} = (h + 1) \log_2 2 = h + 1$$

Eli $h = \log_2 (n + 1) - 1$. \square

- Eli jos puussa on n solmua, niin

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

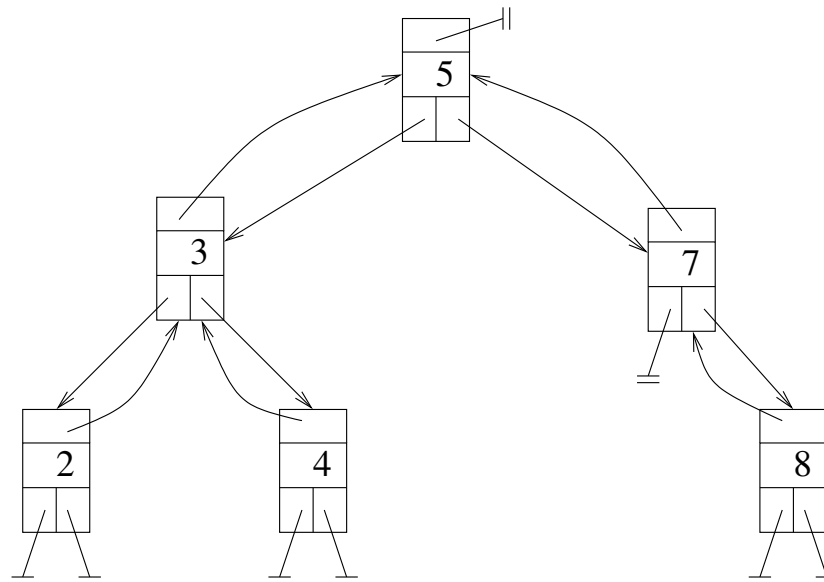
Binäärihakupuu

- Toteutetaan sivun 159 abstrakti tietotyyppi joukko siten, että joukossa olevat alkiot talletetaan binääripuun solmuihin
- Rajoitumme jälleen yksinkertaistettuun tapaukseen, jossa talletettavat tietoalkiot sisältävät ainoastaan avaimen
- Puu rakentuu puusolmu-olioista, joilla on seuraavat kentät:

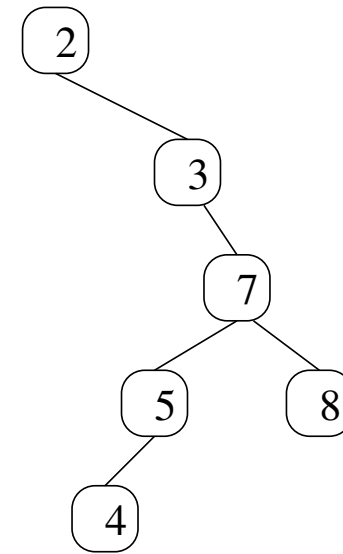
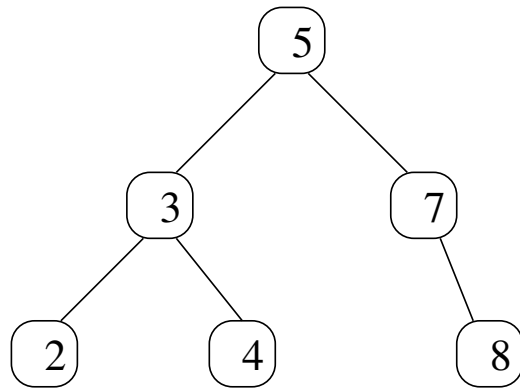
<i>key</i>	talletettu tietoalkio
<i>left</i>	viite vasempaan lapseen
<i>right</i>	viite oikeaan lapseen
<i>parent</i>	viite vanhempaan

- Puulla T on attribuutti $T.root$ joka osoittaa juurisolmuun
- Jos solmulla x ei ole vasenta lasta, $x.left = NIL$, vastaavasti oikealle lapselle
- Puun juurisolmulla *parent*-kentän arvo NIL
- *parent*-kenttä ei ole välttämätön, se kuitenkin helpottaa muutaman operaation toteutusta, joten pidämme sen mukana

- Binäärihakupuussa (engl. binary search tree) avaimet toteuttavat binäärihakupuuehdon:
 - jos solmu v on solmun x vasemmassa alipuussa, niin $v.key < x.key$
 - jos solmu o on solmun x oikeassa alipuussa, niin $x.key < o.key$
- Eli solmun vasemmassa alipuussa on ainoastaan sitä pienempiä ja oikeassa alipuussa sitä isompia avaimia
- Muistutus: vasemmalla alipuulla tarkoitetaan solmun vasemmasta lapsesta alkavaa puun osaa, oikea alipuu taas on oikeasta lapsesta alkava osa
- Esim: eräs tapa tallettaa alkioit 2, 3, 4, 5, 7 ja 8 binäärihakupuuhun



- Yleensä piirrämme puut ilman linkkikenttien eksplisiittistä sisältöä
- Samansisältöiset, mutta muodoiltaan erilaiset binäärihakupuut:



- Kun monisteen tässä luvussa puhutaan puusta, tarkoitetaan jatkossa yleensä binäärihakupuuta
- Ryhdymme kohta tarkastelemaan, miten tällaisen rakenteen avulla voidaan toteuttaa tietotyypin *joukko* operaatiot

Puun alkioden läpikäynti

- Usein on tarvetta operaatiolle, joka käy läpi kaikki puun alkiodet
- Alkioden läpikäynnin voi toteuttaa kulkemalla puun *left*-, *right*- ja *parent*-linkkejä pitkin
- Huomattavasti helpompi tapa on kuitenkin toteuttaa läpikäynti rekursion avulla
- Puun T alkiodet voidaan tulostaa suuruusjärjestyksessä kutsumalla seuraavaksi esitettävää rekursiivista algoritmia parametrilla $T.root$
- **tulosta-alkiodet**(x)
 - if $x \neq \text{NIL}$
 - tulosta-alkiodet($x.left$)
 - print $x.key$
 - tulosta-alkiodet($x.right$)

- Algoritmin toiminnan eteneminen syötteenä sivun 266 kuvan vasen puu

```
tulosta-alkiot(5)
  tulosta-alkiot(3)
    tulosta-alkiot(2)
      tulosta-alkiot(NIL)
      print(2)                2
      tulosta-alkiot(NIL)
    print(3)                  3
    tulosta-alkiot(4)
      tulosta-alkiot(NIL)
      print(4)                4
      tulosta-alkiot(NIL)
    print(5)                  5
  tulosta-alkiot(7)
    tulosta-alkiot(NIL)
    print(7)                  7
    tulosta-alkiot(8)
      tulosta-alkiot(NIL)
      print(8)                8
      tulosta-alkiot(NIL)
```

edellä esim. kutsu tulosta-alkiot(5) tarkoittaa että operaatiota kutsutaan **viitteenään** solmu joka sisältää avaimenaan numeron 5

- Algoritmi käy puun läpi **sisäjärjestyksessä** (engl. inorder)
 - tullessa solmuun x ensin käsitellään vasen lapsi $x.left$ sitten itse solmu x ja tämän jälkeen oikea lapsi $x.right$
 - vasenta lasta $x.left$ käsitellessä tulostetaan kaikki siitä alkavan alipuun alkiot ja binäärihakupuuehdon perusteella näiden arvo on korkeintaan sama kuin $x.key$:n arvo
 - oikeaa lasta $x.right$ käsitellessä tulostetaan kaikki siitä alkavan alipuun alkiot ja binäärihakupuuehdon perusteella näiden arvo on vähintään yhtä suuri kuin $x.key$:n arvo
 - arvo $x.key$ tulostetaan siis oikeassa kohdassa suuruusjärjestyksen suhteen
 - samanlaisen päättelyn perusteella jokainen alkio tulostetaan oikeassa kohdassa
- Jos puussa on n solmua, algoritmin aikavaativuus on $\mathcal{O}(n)$ sillä metodin runko-osa on selvästi vakioaikainen ja rekursiota kutsutaan jokaiselle solmulle sekä lehtisolmujen olemattomille lapsille eli yhteensä korkeintaan $2n + 1$ kertaa (lehtisolmuja puussa, jossa on n sisäsolmua, voi olla korkeintaan $n + 1$ kpl)

- Algoritmin tilavaativuus
 - algoritmin edetessä rekursiopinoon on talletettuna reitti juurisolmusta tarkasteltavaan solmuun
 - algoritmin suoritusaikana rekursiopinossa on siis tietoa "piilossa"
 - rekursiopinon koko on pahimmillaan sama kuin puun korkeus h , eli algoritmin tilavaativuus on $\mathcal{O}(h)$
 - Lauseen 6.2 perusteella puun korkeus pahimmassa tapauksessa $n - 1$, missä n solmujen lukumäärä, eli
 - tilavaativuus on siis pahimmassa tapauksessa $\mathcal{O}(n)$
- Sisäjärjestyksen lisäksi kaksi muuta tärkeää binääripuun läpikäyntistrategiaa ovat esijärjestys (preorder) ja jälkijärjestys (postorder), jotka molemmat on helppo toteuttaa sisäjärjestyksen tapaan rekursiivisina operaationa
 - esijärjestyksessä solmu x käsitellään ensin ja sen jälkeen rekursiivisesti alipuut $x.left$ ja $x.right$
 - jälkijärjestyksessä käsitellään ensin rekursiivisesti alipuut $x.left$ ja $x.right$ ja lopuksi solmu x
- Eli algoritmi on esi- ja jälkijärjestyksessä sama kuin sisäjärjestyksen tapauksessa lukuunottamatta itsensä alkion x käsittelykohtaa

- Puusta voi selvittää monenlaisia mielenkiintoisia asioita sopivilla läpikäynneillä
- Jos halutaan kuljettaa tietoa lehdistä juureen päin, on [jälkijärjestys](#) hyvä strategia, eli käsittele lapset ensin, ja vasta sitten solmu itse
- Esim. puun korkeuden selvittäminen
 - solmun korkeus on sen lasten korkeuksien maksimi plus 1
 - lehtisolmujen korkeus on 0
 - algoritmi yksinkertaistuu hieman jos ajatellaan, että olemattomien alipuiden korkeus on -1

laske-korkeus(x)

```

if x == NIL
    return -1 // olemattoman alipuun korkeus on -1
k1 = laske-korkeus(x.left) // selvitetään vasemman lapsen korkeus
k2 = laske-korkeus(x.right) // selvitetään oikean lapsen korkeus
return max( k1, k2 )+1 // oma korkeus on lasten korkeuden maksimi +1

```

- Eli ensin selvitetään lapsien korkeudet ja palautetaan sitten kysyjälle oma korkeus
- Operaatiolle annetaan parametriksi puun juuri: $korkeus = \text{laske-korkeus}(T.root)$

- Jos halutaan kuljettaa tietoa juuresta lehtiin päin, on **esijärjestys** hyvä strategia, eli käsittele solmu ensin, ja vasta sen jälkeen lapset
- Esim. halutaan muuttaa solmujen avaimiksi arvo, joka on vanha arvo + niiden solmujen vanhojen avaimien arvot, jotka sijaitsevat polulla juuresta solmuun
 - kuljetetaan summaa mukana puussa ylhäältä alaspäin
 - huom: puu tuskin säilyy hakupuuna tälläisen operaation seurauksena

muuta-arvoja(x, summa)

if $x \neq \text{NIL}$

 x.key = x.key + summa

 muuta-arvoja(x.left, x.key)

 muuta-arvoja(x.right, x.key)

- kutsutaan operaatiota parametrina juuri ja 0: **muuta-arvoja**(*T.root*, 0)
- juuren avain säilyy entisellään, juuren lapsien avaimien arvoksi tulee juuren avaimen arvo + alkuperäinen avain, jne.
- Läpikäynti esijärjestyksessä vastaa verkossa syvyysuuntaista läpikäyntiä
- Joissain sovellustilanteissa puun läpikäyntijärjestyksellä ei ole väliä
- Läpikäynnit on mahdollista toteuttaa myös ilman rekursiota vakiotilassa olettaen, että kahden osoittimen yhtäsuuruusvertailu on mahdollista

- Yhteenvetona toistetaan vielä eri puiden läpikäynnit: sisäjärjestys, esijärjestys, ja jälkijärjestys

Inorder-Tree-Walk(x)

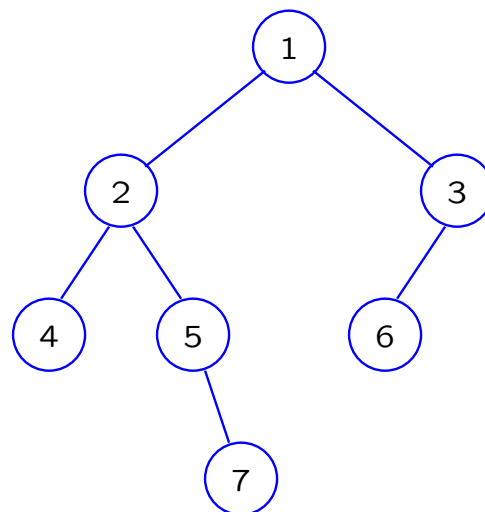
```
if x ≠ NIL
    Inorder-Tree-Walk(x.left)
    käsittele(x)
    Inorder-Tree-Walk(x.right)
```

Preorder-Tree-Walk(x)

```
if x ≠ NIL
    käsittele(x)
    Preorder-Tree-Walk(x.left)
    Preorder-Tree-Walk(x.right)
```

Postorder-Tree-Walk(x)

```
if x ≠ NIL
    Postorder-Tree-Walk(x.left)
    Postorder-Tree-Walk(x.right)
    käsittele(x)
```



Binääripuun solmut

sisäjärjestyksessä: 4, 2, 5, 7, 1, 6, 3

esijärjestyksessä: 1, 2, 4, 5, 7, 3, 6

jälkijärjestyksessä: 4, 7, 5, 2, 6, 3, 1

Joukko-operaatioiden toteuttaminen

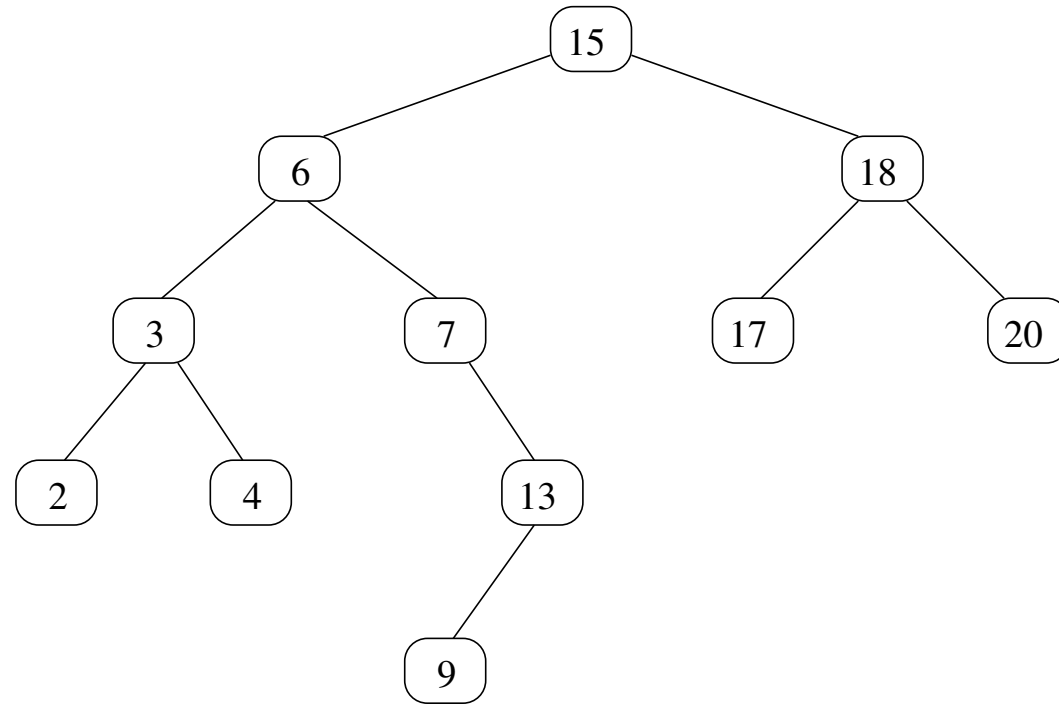
- Avaimen k etsiminen puusta tapahtuu kutsumalla rekursiivista operaatiota **search**($T.root, k$)

search(x, k)

```
if  $x == \text{NIL}$  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 binäärihakupuehdon perusteella k ei voi olla kuin x :n vasemmassa alipuussa. Etsintä siis jatkuu rekursiivisesti vasempaan alipuuhun
- vastaavasti, jos etsittävä k on suurempi kuin $x.key$, jatkuu etsintä rekursiivisesti oikeaan alipuuhun
- jos etsittävä solmu on puussa, päättyy rekursion eteneminen etsittävän solmun kohdalle, palautetaan **true** joka välittyy alkuperäiselle kutsujalle asti
- 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 $15 \rightarrow 6 \rightarrow 7 \rightarrow 13 \rightarrow 9 \rightarrow \text{NIL}$
etsintä siis päätty solmun 9 oikeaan olemattomaan alipuuhun, jossa avaimen 10 täytyisi olla jos se ylipäätään olisi puussa

- Etsintä siis etenee juuresta alaspäin, pahimmassa tapauksessa puun korkeuden verran, eli aikavaativuus $\mathcal{O}(h)$ missä h puun korkeus, sillä jokaisen solmun kohdalla tehdään vakiomäärä työtä

Lauseen 6.2 perusteella puun korkeus pahimmassa tapauksessa $n - 1$, missä n solmujen lukumäärä, eli **search**-operaation pahin tapaus vie aikaa $\mathcal{O}(n)$

- Rekursiopinon korkeus pahimmillaan sama kuin puun korkeus, eli myös tilavaativuus $\mathcal{O}(n)$
- Avaimen etsiminen on helppo toteuttaa myös ilman rekursiota

search(x,k)

 while $x \neq \text{NIL}$ and $x.\text{key} \neq k$

 if $k < x.\text{key}$

$x = x.\text{left}$

 // siirretään x viittaamaan vasempaan lapseen

 else $x = x.\text{right}$

 // 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 päädyimme pienimpään puussa olevaan alkioon seuraavassa operaatio, joka palauttaa viitteen solmun x alipuun pienimpään alkioon

min(x)

```
while x.left  $\neq$  NIL
    x = x.left
return x
```

- Vastaavasti, maksimialkio löytyy menemällä oikealle niin kauan kuin mahdollista:

max(x)

```
while x.right  $\neq$  NIL
    x = x.right
return x
```

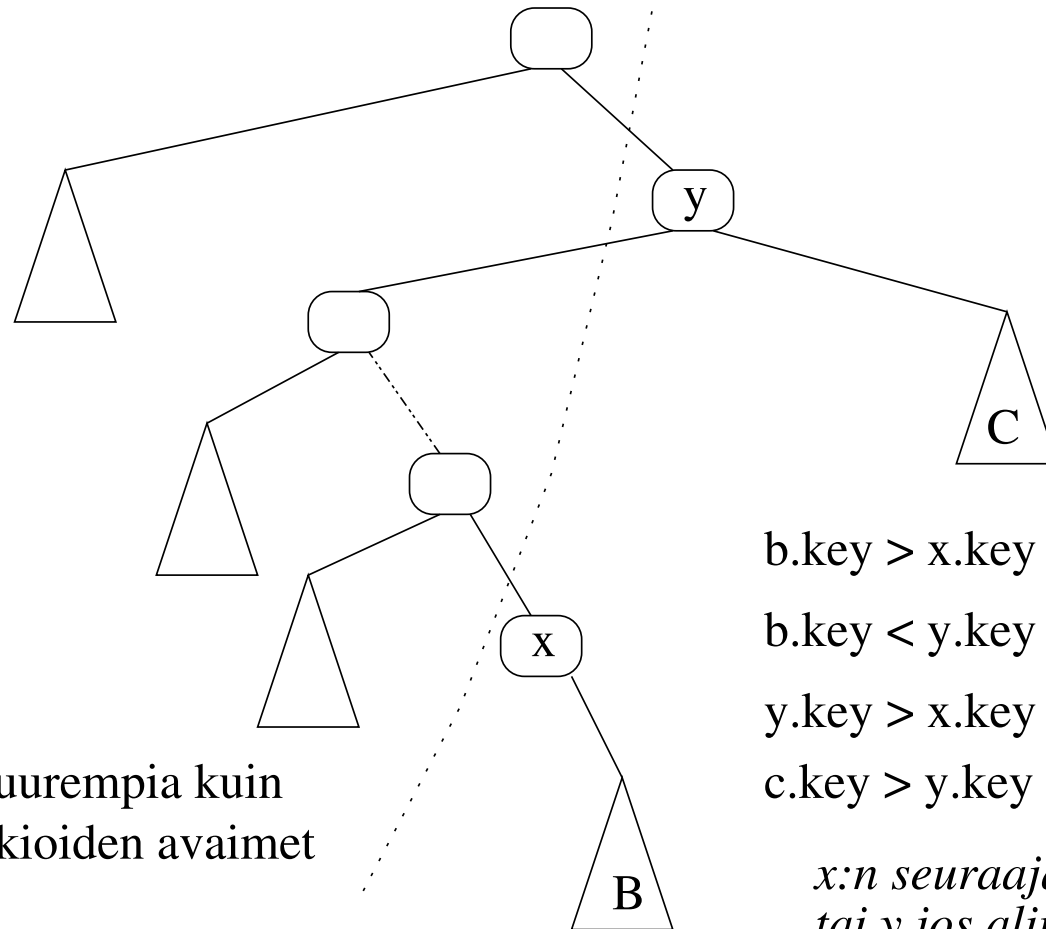
- Kuten ei-rekursiivisella **search**:illa, sekä **min**- että **max**-operaatioiden pahimman tapauksen aikavaativuus puun korkeuden h suhteen on $\mathcal{O}(h)$ ja tilavaativuus $\mathcal{O}(1)$
- Sivun 276 esimerkissä minimin haku etenee polkua $15 \rightarrow 6 \rightarrow 3 \rightarrow 2$ ja maksimin polkua $15 \rightarrow 18 \rightarrow 20$

- Annettua alkioita seuraavaksi suurimman etsintä, eli operaation **succ** toteuttaminen on hiukan hankalampaa
- Sivun 276 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
 - binäärihakupuuuehdosta seuraa, että x :n oikeassa alipuussa $x.right$ olevat alkioit ovat x :ää suurempia
 - toisaalta x :ää suurempi on myös sellainen esi-isä y , joka löytyy kulkemalla x :stä kohti juurta ja johon saavutaan kun otetaan ensimmäinen askel ylös oikealle
 - näin löytyvä alkio y on alipuun $x.right$ alkioiden jälkeen x :ää seuraavaksi suurempi alkio
 - eli x :n seuraaja on alipuun $x.right$ pienin alkio jos alipuu ei ole tyhjä. Jos taas alipuu on tyhjä, on seuraaja kuvatulla tavalla löytyvä y

- Perustellaan vielä hieman täsmällisemmin miksi seuraaja löytyy kuvatulla tavalla. Seuraavien kahden sivun kuvat liittyvät perusteluun
- Jos tämä perustelu (joka on matemaattisen oikeellisuustodistuksen "kansanomaistettu" versio) tuntuu sekavalta, hyppää sen yli suosiolla
- Olkoon x solmu, jonka seuraajaa etsimme, ja y sen esi-isä, joka löytyy kulkemalla kohti juurta siihen asti kunnes on kuljettu yksi askel oikealle
- Binäärihakupuuvedon perusteella
 - x :n alipuussa B olevat solmut ovat x :ää suurempia
 - y ja sen oikeassa alipuussa C olevat solmut ovat x :ää suurempia
 - y :n vanhemmassa ja sen toisessa alipuussa olevat solmut ovat joko solmua x pienempiä (kuva sivulla 281) tai y :tä suurempia (kuva sivulla 282), ne eivät siis voi tulla kyseeseen x :n seuraajina
 - y :n vasemmassa alipuussa olevat solmut jotka jäävät x :n "vasemmalle puolelle" ovat x :ää pienempiä, eli eivät voi tulla kyseeseen x :n seuraajina
 - B :ssä olevat solmut ovat y :tä pienempiä, eli jos B on epätyhjä, on sen pienin alkio x :n seuraaja
 - jos B on tyhjä, täytyy x :n seuraajan olla y

- Edellisen sivun päättelyä tukeva kuva, tapaus jossa y on vanhempansa oikea lapsi

tilanne, jossa y on vanhempansa oikea lapsi



$x.key$ ja $y.key$ suurempia kuin
taman alueen alkioden avaimet

$b.key > x.key$ kaikilla b alipuussa B

$b.key < y.key$ kaikilla b alipuussa B

$y.key > x.key$

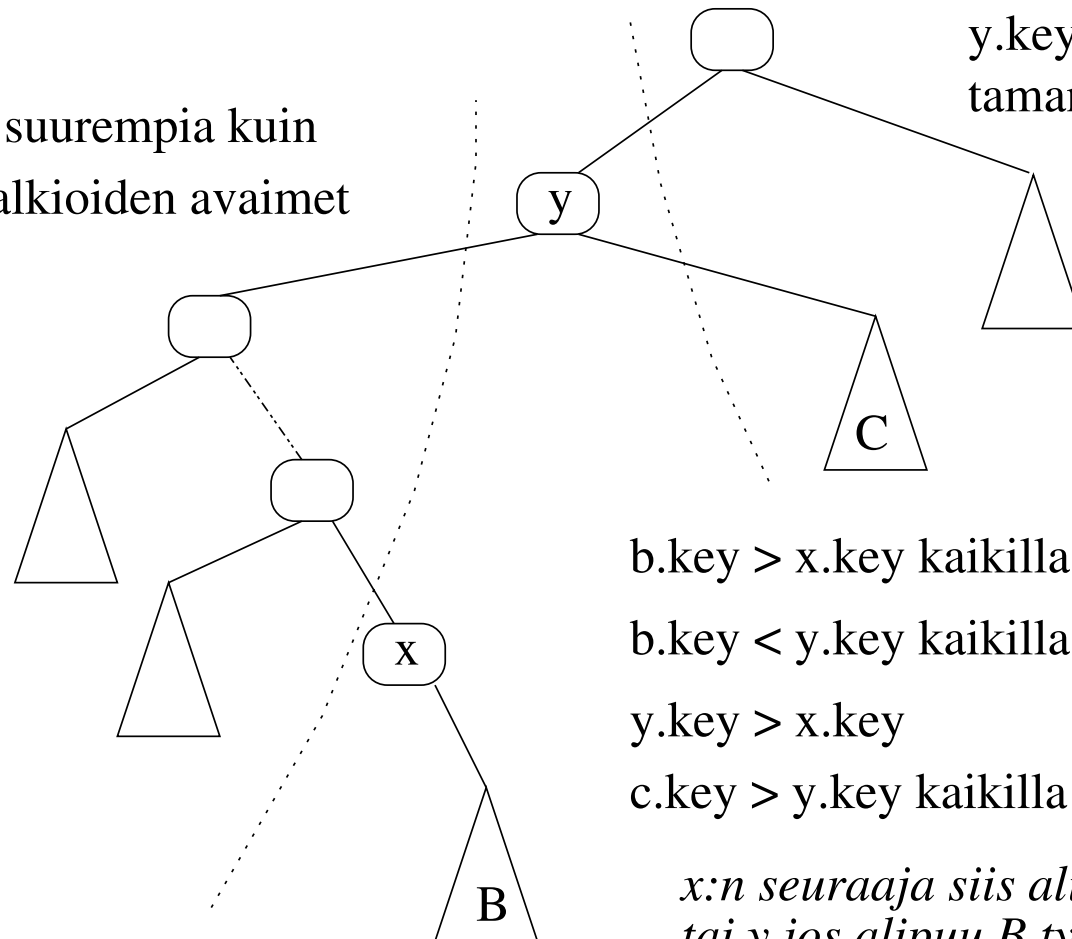
$c.key > y.key$ kaikilla c alipuussa C

*x :n seuraaja siis alipuun B pienin
tai y jos alipuu B tyhjä*

- Toinen tapaus, eli y on vanhempansa vasen lapsi:

tilanne jossa y on vanhempansa vasen 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*

- Algoritmi seuraavassa:

succ(x)

```
1  if  $x.right \neq NIL$ 
2      return min( $x.right$ )
3   $y = x.parent$ 
4  while  $y \neq NIL$  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 paluuskel "yläviistoon oikealle"
- **succ**-operaation pahin tapaus vie aikaa $\mathcal{O}(h)$ puun korkeuden h suhteen, sillä voidaan joutua menemään juuresta aina koko puun korkeuden verran alas tai palata lehdestä aina juureen asti, tilavaativuus on selvästi $\mathcal{O}(1)$
- **pred**-operaatio on symmetrinen **succ**-operaation kanssa, eli vaatii pahimmassa tapauksessa ajan $\mathcal{O}(h)$ ja tilan $\mathcal{O}(1)$

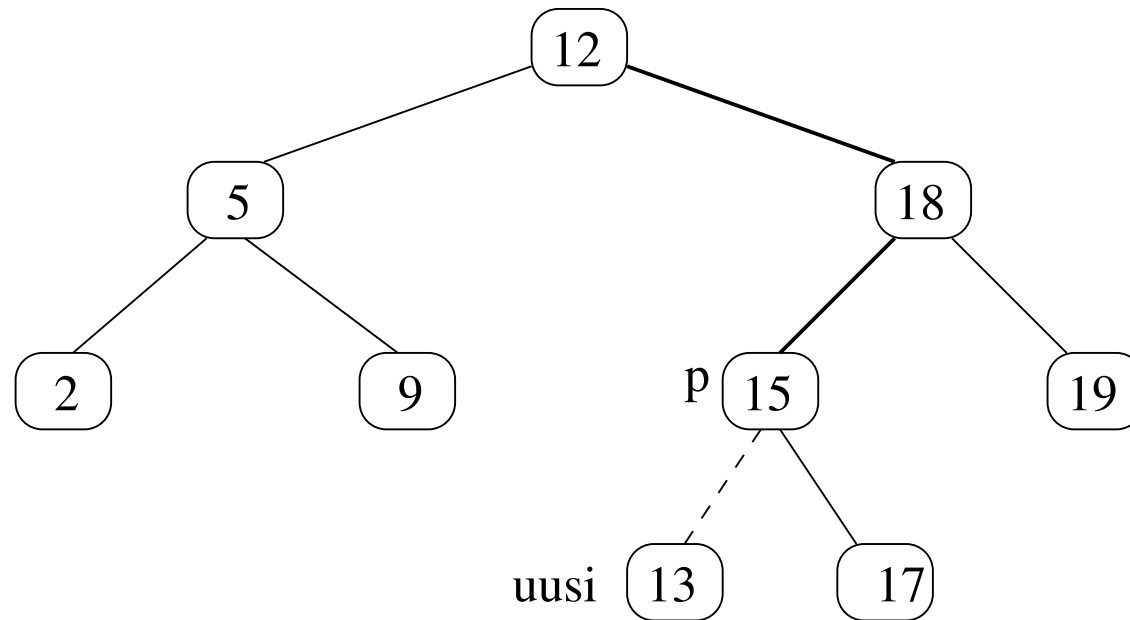
- Alkion lisääminen binäärihakupuuhun käy melko helposti

insert(T,k)

```
1  uusi = new puusolmu
2  uusi.key = k
3  uusi.left = uusi.right = uusi.parent = NIL
4  if T.root == NIL          // jos puu on tyhjä, tulee uudesta solmusta juuri
5      T.root = uusi
6      return
7  x = T.root
8  while x ≠ NIL            // etsitään kohta, johon uusi alkio kuuluu
9      p = x
10     if uusi.k < x.key
11         x = x.left
12     else x = x.right
13 uusi.parent = p          // viitteet uuden alkion ja sen vanhemman välille
14 if uusi.key < p.key
15     p.left = uusi
16 else p.right = uusi
```

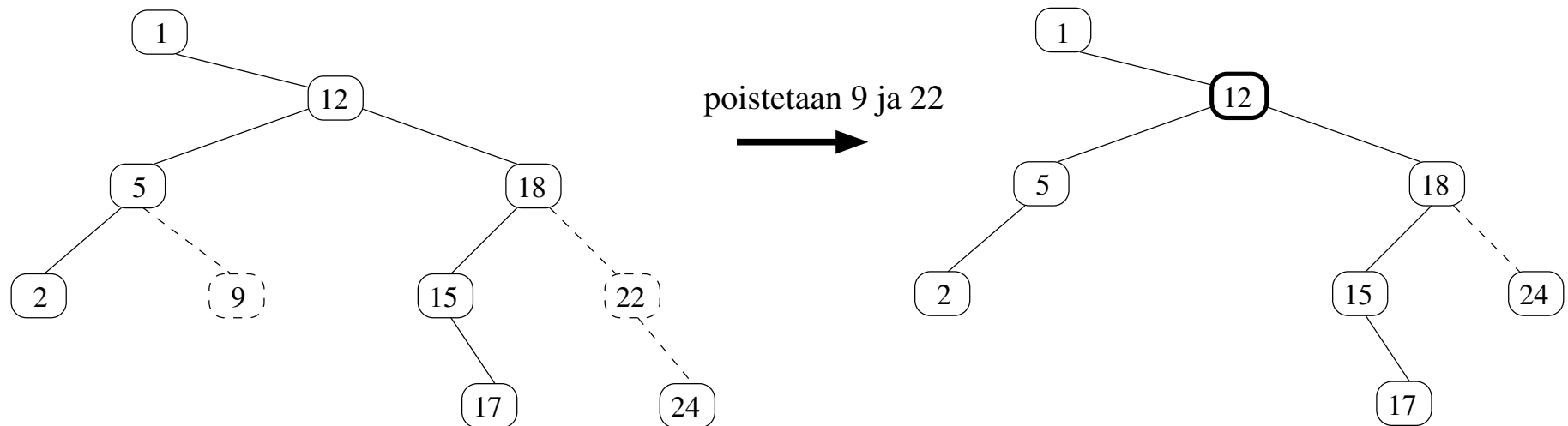
- Algoritmin toimintaperiaate
 - riveillä 4-6 käsitellään erikoistapaus, jossa lisättävä on puun ensimmäinen solmu
 - rivien 8-12 `while`-toistolause etsii uudelle alkiole paikan
 - * uusi alkio on lisättävä puuhun siten, että binäärihakupuehto ei mene rikki
 - * paikka löytyy kulkemalla puussa alaspäin 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 $x = \text{NIL}$, tiedetään, että p on solmu, jonka alle uusi solmu voidaan laittaa rikkomatta binäärihakupuehtoa
 - eli toistolauseen jälkeen p viittaa alkioon jonka lapseksi uusi alkio lisätään
 - riveillä 14-16 uusi alkio laitetaan joko p :n vasemmaksi tai oikeaksi lapseksi riippuen onko uuden solmun avain pienempi vai suurempi kuin p :n avain

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

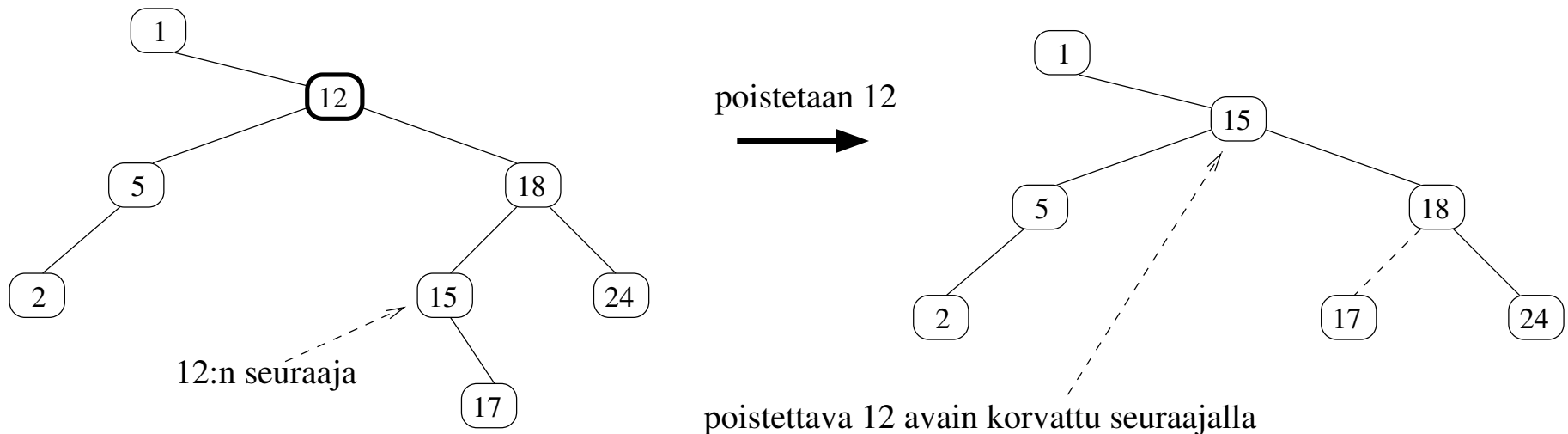


- Kuten muillakin tähän asti kohtaamillamme operaatioilla, lisäyksen aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen, sillä pahimmassa tapauksessa lisäys tehdään alimmalla tasolla olevan solmun lapseksi ja tällöin operaation aikavaativuutta dominoiva **while** joudutaan suorittamaan h kertaa
- Tilavaativuus on $\mathcal{O}(1)$, sillä rekursio ei ole käytössä ja apumuuttujia on vain muutama

- Poisto on puun operaatioista monimutkaisin. Operaatio jakautuu kolmeen tapaukseen
 - solmu, jolla ei ole lapsia on helppo poistaa: alla olevasta kuvasta esim. 9 voidaan poistaa laittamalla 5:n oikeaksi lapseksi NIL
 - solmu, jolla on tasan yksi lapsi on lähes yhtä helppo poistaa: kuvassa solmu 22 saadaan poistettua laittamalla sen lapsi 24 suoraan vanhemman 18 lapseksi, eli yksilapsinen solmu poistetaan korvaamalla se lapsella
 - kaksilapsinen solmu, esim. kuvassa 12 on ongelmallinen tapaus



- Kaksilapsinen solmu poistetaan puusta seuraavasti:
 - vaihdetaan poistettavan ja sitä seuraavaksi suurimman avaimen omaavan solmun *seur* sisältö
 - koska poistettavalla on molemmat lapset, sen seuraajasolmu *seur* on poistettavan oikean alipuun solmuista pienin
 - poistetaan solmu *seur*, se onnistuu helposti sillä solmulla on korkeintaan yksi lapsi (ei voi olla vasenta lasta)
- Kuvassa solmun 12 seuraaja on solmu 15. Sen sisältö vietään poistettavaan solmuun ja poistetaan vanha 15 puusta.



- Parametrina operaatiolla on viite poistettavaan solmuun *pois*

delete(T,pois)

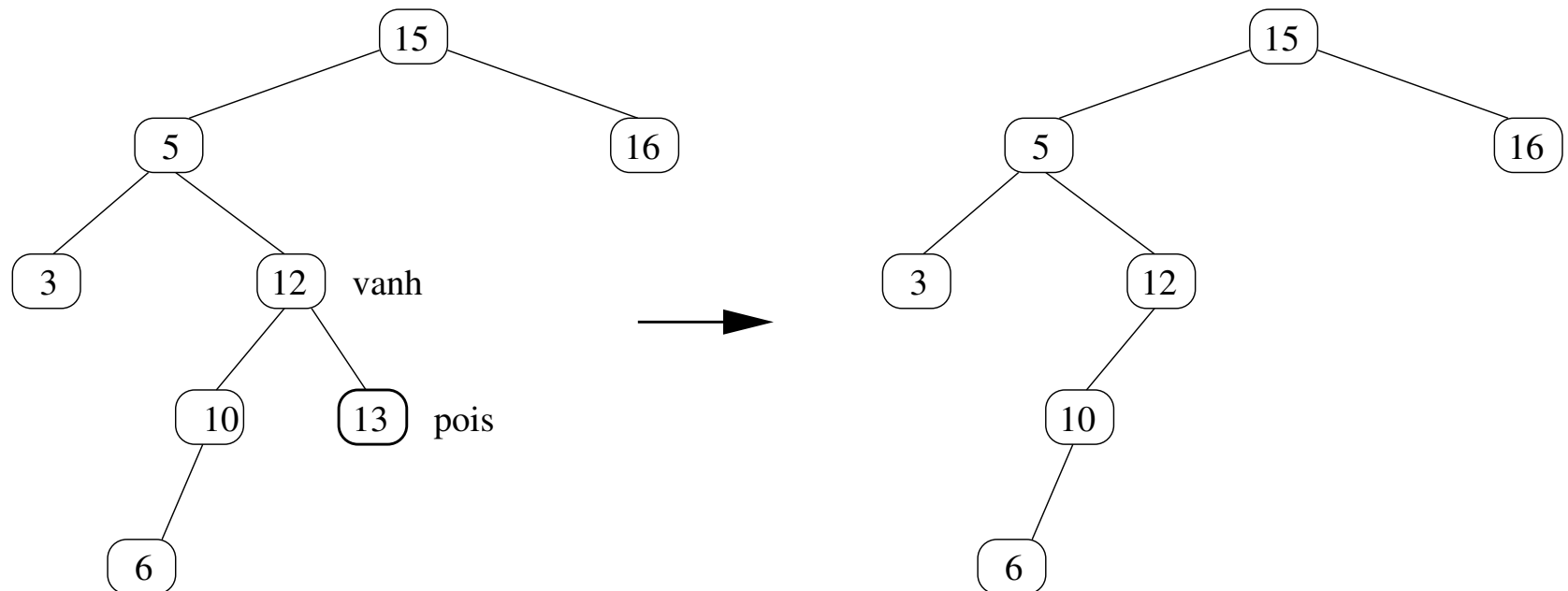
```
1  if pois.left == NIL and pois.right == NIL
// tapaus 1: poistettavalla ei lapsia
2      vanh = pois.parent
3      if vanh == NIL                                // poistettava on puun ainoa solmu
4          T.root = NIL
5          return pois
6      if pois == vanh.left
7          vanh.left = NIL
8      else vanh.right = NIL
9      return pois
10 if pois.left == NIL or pois.right == NIL
// tapaus 2: poistettavalla on yksi lapsi
11     if pois.left ≠ NIL
12         lapsi = pois.left
13     else lapsi = pois.right
14     vanh = pois.parent
15     lapsi.parent = vanh
16     if vanh == NIL                                // poistettava on juuri
17         T.root = lapsi
18         return pois
delete(T, pois) jatkuu seuraavalla sivulla...
```

delete(T,pois) jatkuu edelliseltä sivulla...

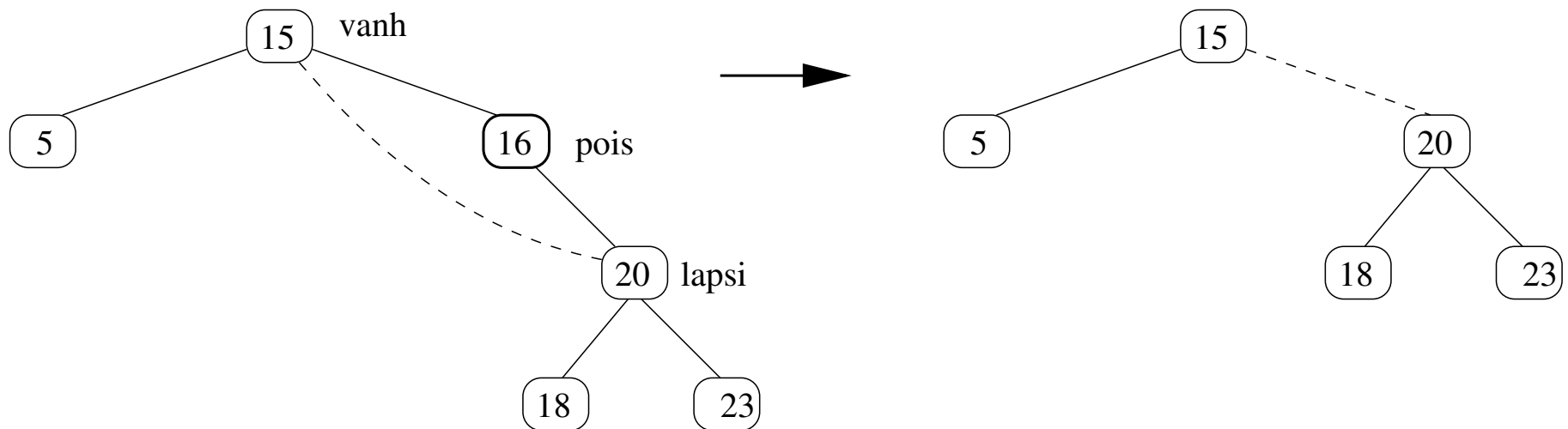
```
19     if pois == vanh.left
20         vanh.left = lapsi
21     else vanh.right = lapsi
22     return pois
// tapaus 3: poistettavalla kaksi lasta
23 seur = min(pois.right)
24 pois.key = seur.key           // korvataan poistettavan avain seuraajan avaimella
25 lapsi = seur.right
26 vanh = seur.parent           // korvataan solmu seur sen lapsella
27 if vanh.left == seur
28     vanh.left = lapsi
29 else vanh.right = lapsi
30 if lapsi ≠ NIL
31     lapsi.parent = vanh
32 return seur
```

- Operaatio palauttaa viitteen siihen solmuun joka todellisuudessa poistettiin jotta kutsuja voi tarvittaessa vapauttaa muistitilan. Kolmannessa tapauksessahan poistettava solmu ei ole sama kuin parametrina annettu
- Algoritmi näyttää huomattavasti monimutkaisemmalta mitä se itseasiassa on
- Cormenissa esitetty versio on hieman lyhempi, mutta siinä kolmea tapausta ei ole koodissa eritelty samaan tapaan kuin tässä esitettyssä

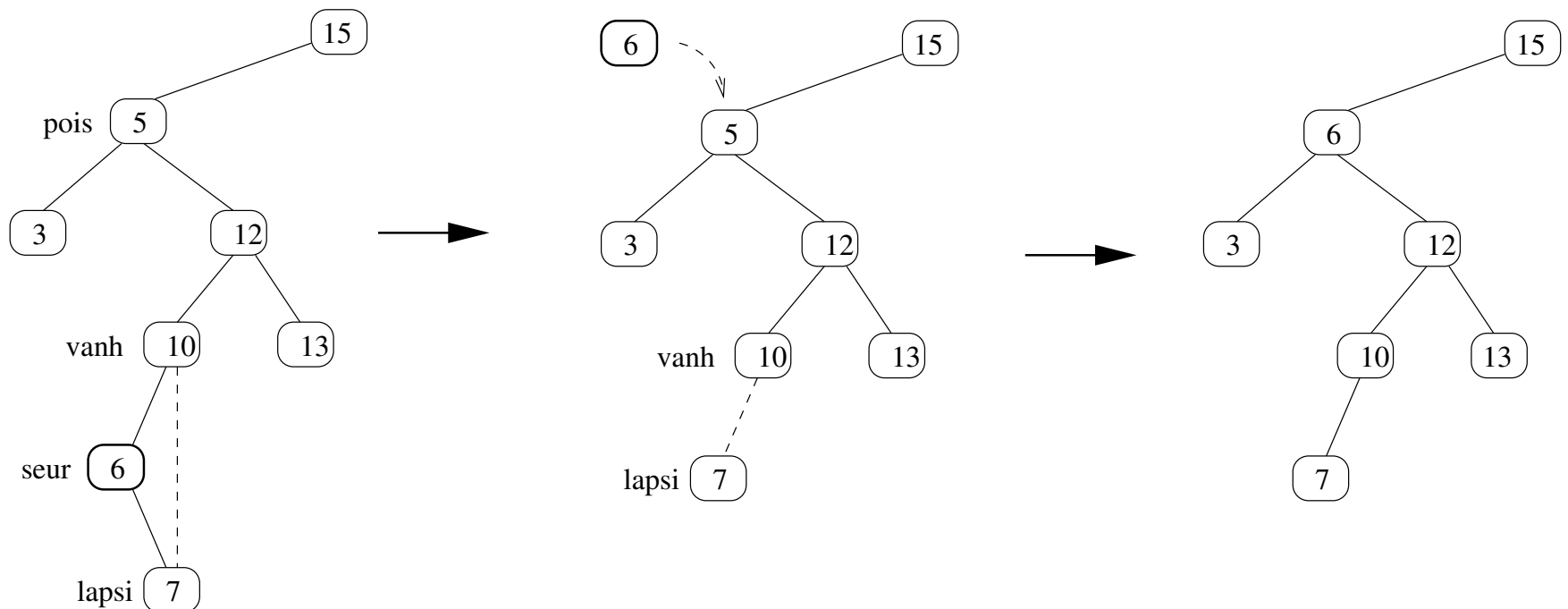
- Käydään algoritmin toiminta läpi vielä tapaus tapaukselta
- Tapaus 1: poistettavalla ei lapsia, rivit 1-9
 - poistetaan solmu *pois*
 - riveilla 3-5 huomioidaan erikoistapaus missä poistettava on puun ainoa solmu. Tässä tapauksessa puun juureksi asetetaan NIL
 - riveillä 6-8 asetetaan poistuneen solmun tilalle sen vanhempaan NIL-viite



- Tapaus 2: poistettavalla yksi lapsi, rivit 10-22
 - korvataan poistettava *pois* ainoalla lapsellaan *lapsi*
 - riveillä 11-13 selvitetään, onko poistettavan lapsi oikea vai vasen
 - riveillä 16-18 huomioidaan tapaus, missä on poistettava on puun juuri. Tällöin poistettavan lapsesta tulee uusi juuri
 - riveillä 19-21 laitetaan *lapsi* poistuneen tilalle



- Tapaus 3: poistettavalla kaksi lasta, rivit 23-31
 - aluksi vaihdetaan solmun *pois* avain sen seuraajan *seur* avaimeen
 - solmu *seur* mistä korvaava avain löytyi poistetaan korvaamalla se lapsellaan *lapsi*, tämä tapahtuu riveillä 25-29
 - on varmaa, että solmulla *seur* on korkeintaan oikea lapsi, joten sen poistaminen on helppoa
 - jos korvaava solmu *lapsi* ei ole NIL, laitetaan se osoittamaan uuteen vanhempaansa riveillä 30-31



- **delete** on muuten vakioaikainen, mutta tapauksessa kolme joudutaan kutsumaan operaatiota **min**, jonka aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen
- Täten poiston pahimman tapauksen aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen. Tilavaativuus on $\mathcal{O}(1)$ sillä käytettyjen apumuuttujien määrä on vakio
- Huom: Operaatio **delete**($T, pois$) siis poistaa puusta solmun **sisällön**, tapauksessa 3 solmun pois muistialue jää vielä käyttöön sisältäen kuitenkin toisen avaimen

- Binäärihakupuun kaikkien operaatioiden (**search**, **insert**, **delete**, **max**, **min**, **succ** ja **pred**) pahimman tapauksen aikavaativuus on siis $\mathcal{O}(h)$, missä h on puun korkeus
- Kaikki operaatiot pystyttiin tekemään siten että tilavaativuus on vain $\mathcal{O}(1)$
- Lauseen 6.2 perusteella n -solmuisen puun korkeus h vaihtelee välillä $\log_2(n + 1) - 1 \leq h \leq n - 1$
- Eli jos puu on tarpeeksi **tasapainoinen** (eli mahdollisimman paljon täydellistä binääripuuta muistuttava) on operaatioiden aikavaativuus $\mathcal{O}(\log n)$ ja puu on oleellisesti listaa parempi joukon toteutuksessa
- Kun taas hyvin epätasapainoisessa puussa operaatioiden aikavaativuus on $\mathcal{O}(n)$ ja puu on siis jopa listaa huonompi tapa abstraktin tietotyypin joukko toteuttamiseen
 - huono puu syntyy esim. lisäämällä puuhun solmut $1, \dots, n$ 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 vakio kertoimen verran

Tasapainoiset hakupuut

- Tavoitetta toteuttaa joukko-operaatiot ajassa $O(\log n)$ voidaan lähestyä eri tavoin:

AVL-puut: historiallisesti ensimmäinen (1962) ja toteutukseltaan yksinkertaisin

punamustat puut: AVL-puun idean tehostettu versio, käytetään esim. Javan standardikirjastoissa.

B-puut: tehokkaita levymuistia käytettäessä

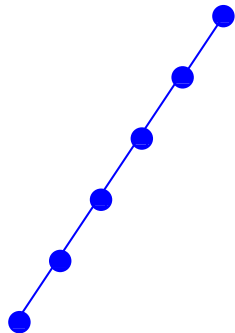
splay-puut: ei takaa tasapainoisuutta, mutta operaatioiden aikavaativuus silti $O(\log n)$ per operaatio (tasoitettu aikavaativuus) kun tarkastellaan **koko operaatiojonoa**

skip-lista: ei puurakenne, vaan usean listan hierarkia, aikavaativuus $O(\log n)$ **odotusarvoisesti**

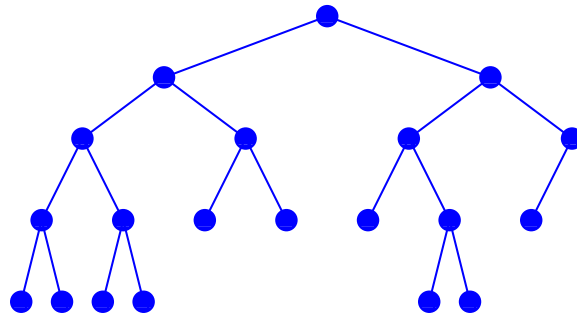
treap: satunnaisuutta käyttävä puun ja keon (heap) yhdistelmä, "odotusarvoisesti tasapainoinen"

- Tutustumme tässä AVL-puihin

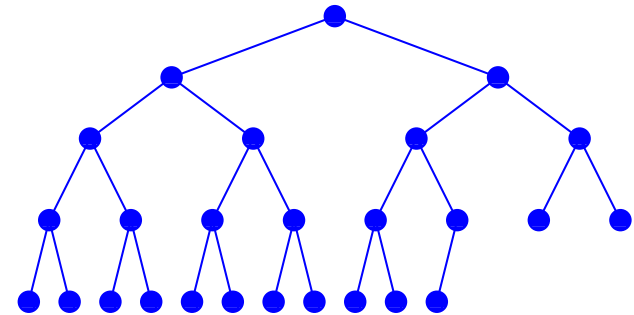
- Hakunopeuden kannalta paras tilanne on täydellinen puu (ks. sivu 261). Tällöin n -alkioisen puun korkeus on $\log_2(n + 1) - 1$. Puuta on kuitenkin vaikea pitää edes suunnilleen täydellisenä, kun siihen lisätään ja siitä poistetaan alkioita
- Pahin tilanne on **lineaarinen puu**, jossa kaikki oikeat tai kaikki vasemmat alipuut ovat tyhjiä, puun korkeus on tällöin $n - 1$
- Intuitiivisesti puu on "tasapainoinen", kun se muistuttaa muodoltaan enemmän täydellistä kuin lineaarista puuta
- Eri tavat määritellä tasapainoisuus täsmällisesti johtavat hieman erilaisiin tietorakenteisiin (esim. AVL-puu tai punamustapuu)



lineaarinen

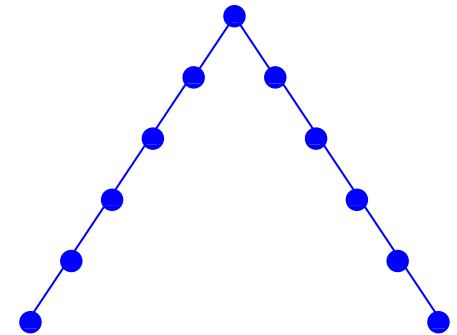


tasapainoinen?



melkein täydellinen

- Teknisesti ottaen haluamme määritellä jonkin täydellisyyttä hieman heikomman **tasapainoehdon**, joka
 - on helpompi pitää voimassa kuin täydellisyys mutta
 - takaa kuitenkin, että puun korkeus on $\mathcal{O}(\log n)$
- Ylläpidon helpottamiseksi siis löysennämme ehtoa niin, että ei vaadita puun olevan lähes täydellinen (eli maksimaalisesti tasapainoinen), vaan riittää että puun korkeus solmujen lukumäärän n suhteen on $d \cdot \log n$ jollakin vakiolla d
- Tasapainoehdon intuitiivinen merkitys on yleensä karkeasti, että **jokaisen solmun vasen ja oikea alipuu ovat jossain mielessä suunnilleen samankokoiset**



- Huomaa, että pelkästään juuren tarkasteleminen ei riitä: Juuren vasen ja oikea alipuu näyttävät samankokoisilta, mutta puun korkeus on $\lfloor n/2 \rfloor$ (eli jakolaskun $n/2$:n kokonaislukuosa) eli puu ei ole tasapainoinen

AVL-puut [Georgi Adelson-Velski ja Jevgeni Landis, 1962]

- AVL-puita Java-koodeineen esitetään Weissin kirjan luvussa 4.4.
 - Monisteessa esitetään AVL-puun **insert**-operaatiosta ei-rekursiivinen versio, toisin kuin Weissin kirjassa
 - Weissin kirja, kuten useimmat lähteet, jättää operaation **delete** kokonaan määrittelemättä
- Normaalin binääripuun solmussa olevien attribuuttien *key*, *left*, *right* ja *parent* lisäksi jokaisessa AVL-puun solmussa on kenttä *height*, joka ilmoittaa solmun korkeuden.
- Muistin virkistykseksi:
 - Solmun korkeus on pisimmän siitä lehteen vievän polun pituus
 - Erityisesti lehden korkeus on 0.
 - Puun (tai alipuun) korkeudella tarkoitetaan sen juuren korkeutta.
- Edellisen kanssa on yhteensopivaa, että tyhjän binääripuun NIL korkeudeksi määritellään -1

- Olettaen että solmuilla on *height*-attribuutit joille on rakennusvaiheessa asetettu oikeat arvot, voidaan solmusta x alkavan alipuun korkeus kysyä funktiolla

Height(x)

```
if  $x == \text{NIL}$ 
    return -1
else return  $x.\text{height}$ 
```

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**:

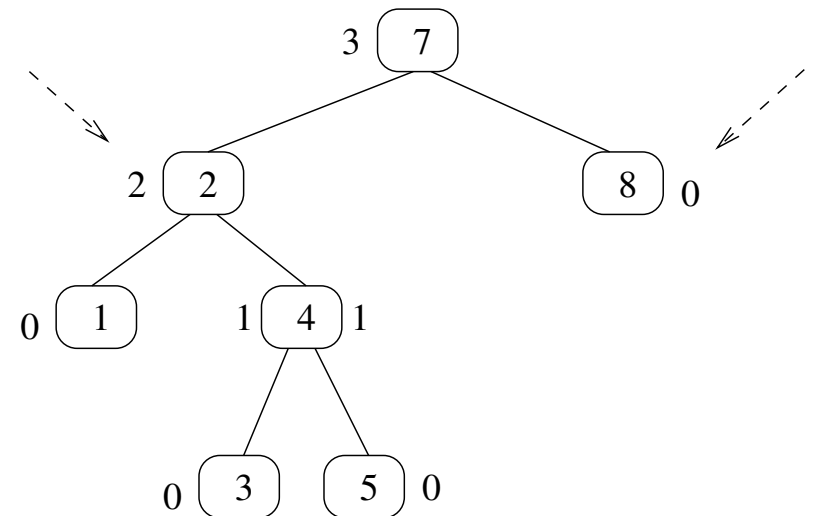
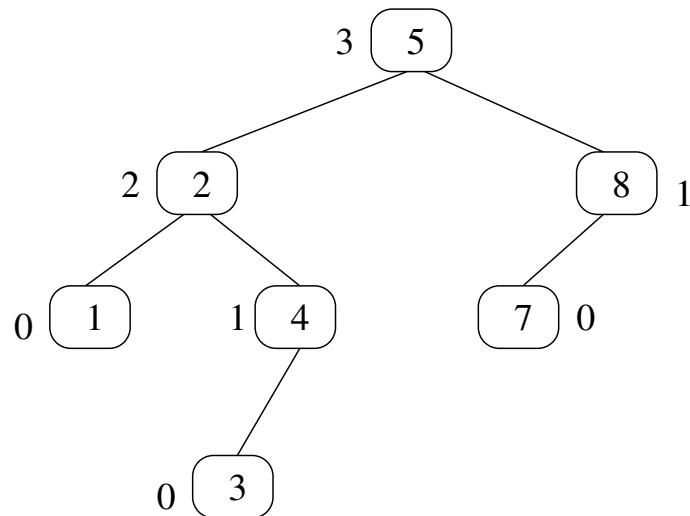
minkä tahansa solmun vasemman ja oikean alipuun korkeuksien erotus on joko -1 , 0 tai 1 .

Toisin sanoen vaaditaan

$$| \mathbf{Height}(x.\text{left}) - \mathbf{Height}(x.\text{right}) | \leq 1 \quad \text{kaikilla solmuilla } x$$

- Haluamme osoittaa, että
 - jos tasapainoehto on voimassa, niin puun korkeus on $\mathcal{O}(\log n)$
 - jos tasapainoehto rikkoutuu avaimen lisäyksen tai poiston yhteydessä, se voidaan saada taas voimaan ajassa $\mathcal{O}(\log n)$ puuta sopivasti muokkaamalla

- Allaoleviin samansisältöisiin puihin on merkitty solmujen korkeudet



- Vasemmanpuoleinen puu toteuttaa AVL-tasapainoehdon, oikeanpuoleinen ei toteuta, sillä juurisolmun 7 alipuiden korkeusero on 2

- **Lause 6.3:** Jos AVL-puun korkeus on h , niin siinä on ainakin $F_{h+3} - 1$ solmua, missä F_i on i :s Fibonaccin luku
- Ennen lauseen todistusta tarkastellaan sen seurauksia
Fibonaccin luvut 0, 1, 1, 2, 3, 5, 8, 13, ... määritellään palautuskaavalla

$$F_0 = 0$$

$$F_1 = 1$$

$$F_{i+2} = F_{i+1} + F_i \quad \text{kun } i \geq 0.$$

Fibonaccin luvuille tunnetaan myös eksplisiittinen kaava

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

Kaavan voi johtaa esim. [generoivien funktioiden](#) avulla, mutta tämä tekniikka ei kuulu kurssin alueeseen. Kaava on kuitenkin helppo todeta oikeaksi induktiolla.

Luku $(1 + \sqrt{5})/2 \approx 1,618$ tunnetaan **kultaisena suhteena**. Koska toinen potenssiin korotettava luku $(1 - \sqrt{5})/2 \approx -0,618$ on itseisarvoltaan ykköstä pienempi, termi $((1 - \sqrt{5})/2)^i$ pienenee nopeasti, kun i kasvaa. Itse asiassa

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

missä $[x]$ tarkoittaa reaalilukua x pyöristettynä lähimpään kokonaislukuun.

Lauseen mukaan pienin solmujen lukumäärä korkeutta h olevassa AVL-puussa on

$$F_{h+3} - 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{h+3} - 1 \approx \frac{1,618^{h+3}}{2,236} - 1.$$

Solmujen lukumäärä kasvaa siis eksponentiaalisesti korkeuden funktiona

Olkoon puolestaan $H(n)$ suurin n -solmuisen AVL-puun korkeus. Merkitään $\alpha = (1 + \sqrt{5})/2$ ja $\beta = (1 - \sqrt{5})/2$. Selvästi jos $H(n) = h$, niin

$$\begin{aligned} n \geq S(h) = F_{h+3} - 1 &= \frac{1}{\sqrt{5}}(\alpha^{h+3} - \beta^{h+3}) - 1 \\ &\geq \frac{1}{\sqrt{5}}(\alpha^{h+3} - |\beta|^{h+3}) - 1. \end{aligned}$$

Tästä seuraa $\sqrt{5}(n + 1 + |\beta|^{h+3}/\sqrt{5}) \geq \alpha^{h+3}$. Ottamalla logaritmit molemmin puolin saadaan h ratkaistua:

$$h \leq \frac{\log_2(n + 1 + |\beta|^{h+3}/\sqrt{5})}{\log_2 \alpha} + \frac{\log_2 \sqrt{5}}{\log_2 \alpha} - 3 \leq 1,4405 \log_2(n + 1,1056) - 1,3277.$$

Siis $H(n) = O(\log n)$ eli puun korkeus kasvaa logaritmisesti.

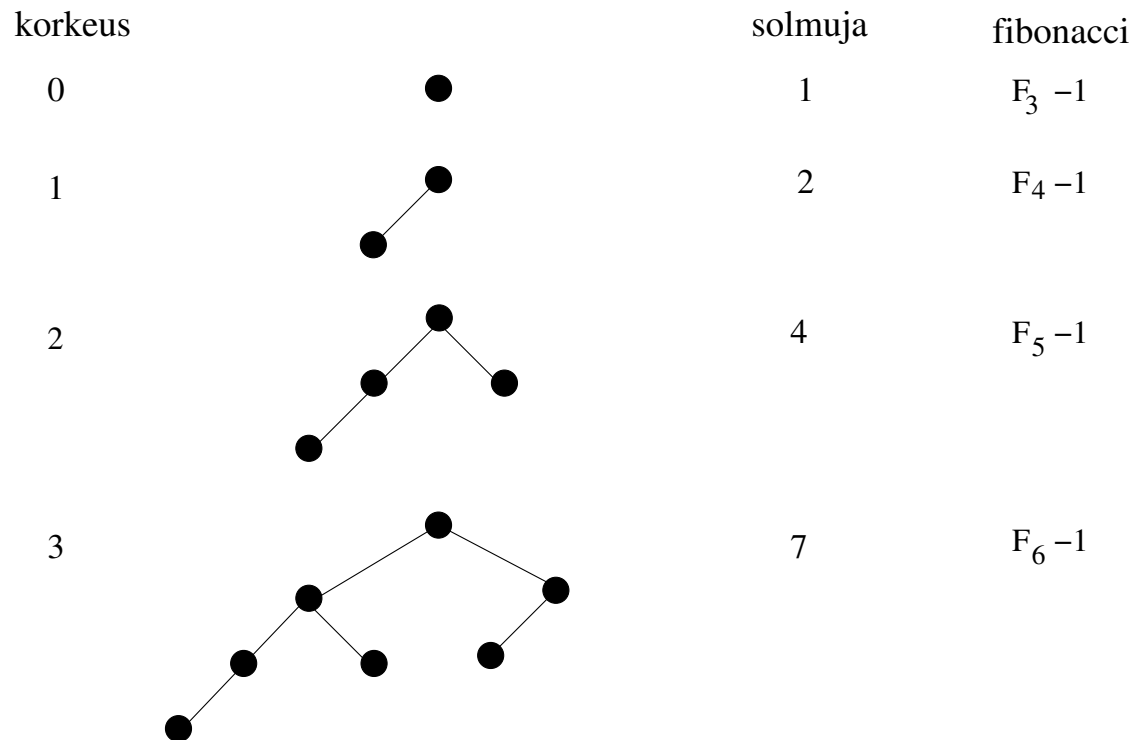
Esimerkki: Jos $n = 100$, niin yllä oleva kaava antaa tulokseksi $h \leq 8,266$, eli 100-solmuinen AVL-puu on korkeintaan korkeudeltaan 8 ja jos $n = 1000000$, niin $h \leq 27,384$, eli korkeus on korkeintaan 27

- **Lauseen 6.3 todistus:**

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

Fibonaccin lukujonon alku siis: $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, F_7 = 13$.

Eli esim. 3:n korkeisessa AVL-puussa pitäisi olla vähintään $F_6 - 1$ eli 7 solmua



Kuvaa tarkastelemalla huomaamme, että esim. pienin 3:n korkuinen AVL-puu voidaan muodostaa ottamalla juurisolmu ja asettamalla sen alipuiksi pienin mahdollinen 2:n korkuinen ja pienin mahdollinen 1:n korkuinen AVL-puu

Pienimmän mahdollisen 3:n korkuinen AVL-puun solmulukumäärä on siis pienimmän mahdollisen 2:n ja 1:n korkuisen puiden solmulukumäärän summa plus yksi (eli juurisolmu)

Sama pätee mille tahansa korkeudelle:

pienimmän h :n korkuisen puun solmujen lukumäärä on pienimmän $h - 1$:n ja pienimmän $h - 2$:n korkuisen puun solmulukumäärä + juurisolmu

Todistetaan nyt induktiolla lauseen väittämä, eli että h :n korkuisessa AVL-puussa on vähintään $F_{h+3} - 1$ solmua

Perustapaus: tyhjän puun korkeus on -1 ja tosiaan $F_2 - 1 = 0$; vastaavasti jos puun korkeus on 0 , puussa on yksi solmu (juuri) ja tosiaan $F_3 - 1 = 1$

Oletetaan, nyt että kaava pitää paikkansa h :n korkuiselle tai sitä matalammille puulle ja näytetään että tästä seuraa että se pätee myös $h + 1$:n korkuiselle puulle, eli että $h + 1$:n korkuisessa AVL-puussa on vähintään $F_{h+4} - 1$ solmua.

Kuten edellä todettiin, saadaan pienin mahdollinen $h + 1$:n korkuinen puu liittämällä juurisolmun lapsiksi pienin mahdollinen h :n ja $h - 1$:n korkuinen puu.

Tällaisen puun solmulukumäärä on lasten alipuiden solmulukumäärä plus juuri.

Induktio-oletuksen perusteella alipuiden solmulukumäärät ovat $F_{h+3} - 1$ ja $F_{h+2} - 1$

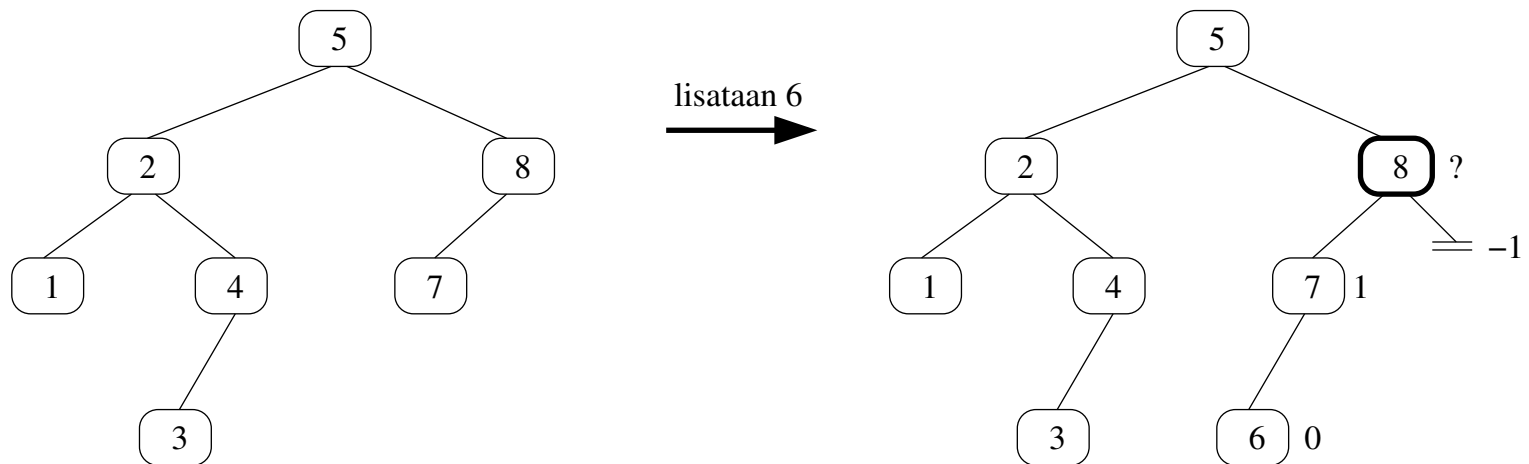
Eli yhteensä pienimmän $h + 1$:n korkuisen puun solmulukumäärä on

$$F_{h+3} - 1 + F_{h+2} - 1 + 1 = F_{h+3} + F_{h+2} - 1$$

Fibonaccin lukusarjalle pätee $F_{h+3} + F_{h+2} = F_{h+4}$, eli pienimmän $h + 1$:n korkuisen puun solmulukumäärä on $F_{h+4} - 1$, eli kaava on todistettu oikeaksi. \square

- **Johtopäätös:** jos binäärihakupuu saadaan toteuttamaan AVL-puun tasapainoehto, niin joukko-operaatiot sujuvat ajassa $O(\log n)$
- **Ongelma:** miten lisäys ja poisto suoritetaan rikkomatta tasapainoehto?
- Ongelman ratkaisu on soveltaa lisäyksen tai poiston jälkeen sopivia **kiertoja** (engl. rotation), joilla mahdollisesti rikkoutunut tasapainoehto saadaan taas voimaan
- Kierto on paikallinen, vakioaikainen operaatio, joka nostaa joitain alipuita ylemmäs ja painaa joitain alemmas
- Kierto korjaa paikallisen epätasapainon, siten että binäärihakupuehto pysyy voimassa
- Osoittautuu, että lisäysoperaation yhteydessä tarvitaan korkeintaan kaksi kiertoa pitämään puu tasapainossa. Poiston yhteydessä kiertoja saatetaan joutua tekemään lineaarinen määrä puun korkeuteen nähden (logaritminen solmujen lukumäärään nähden)
- Kiertoja sovelletaan myös muissa hakupuurakenteissa (punamusta, splay)

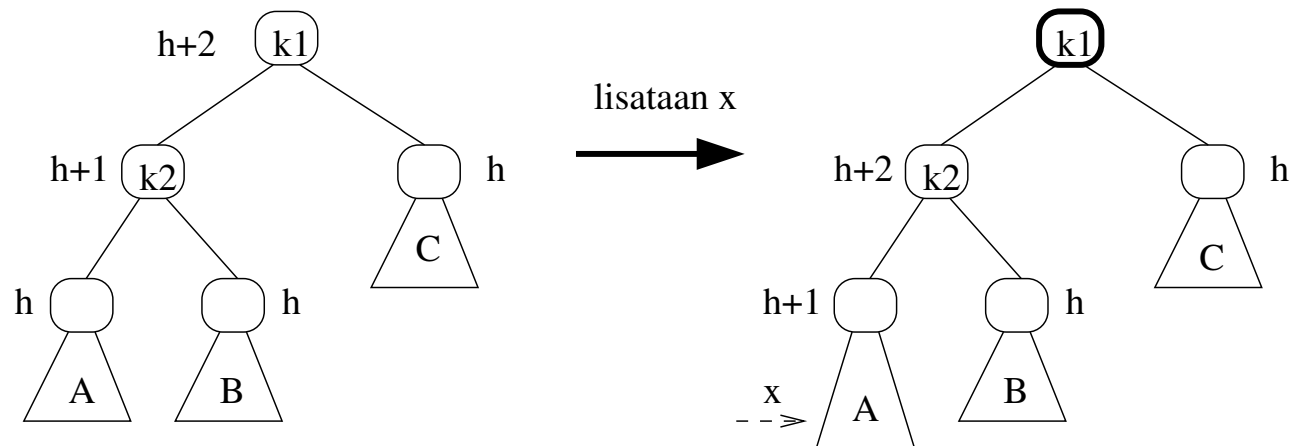
- Lisäys- ja poisto-operaatioiden yhteydessä on siis pidettävä huolta, että puu säilyy AVL-puuna
- Tarkastellaan ensin lisäysoperaatiota
- Kuvan vasemmanpuoleiseen puuhun voitaisiin lisätä esim. avain 0 rikkomatta AVL-ehtoa
- Esim. avaimen 6 lisääminen taas rikkoo AVL-ehdon sillä solmu 8 menee epätasapainoon



- Epätasapainoon menevällä solmulla on vasen alipuu jonka korkeus 1, mutta oikeata alipuuta ei ole. Olemattoman alipuun korkeudeksi sovittiin -1, joten alipuiden korkeuksien erotus on 2

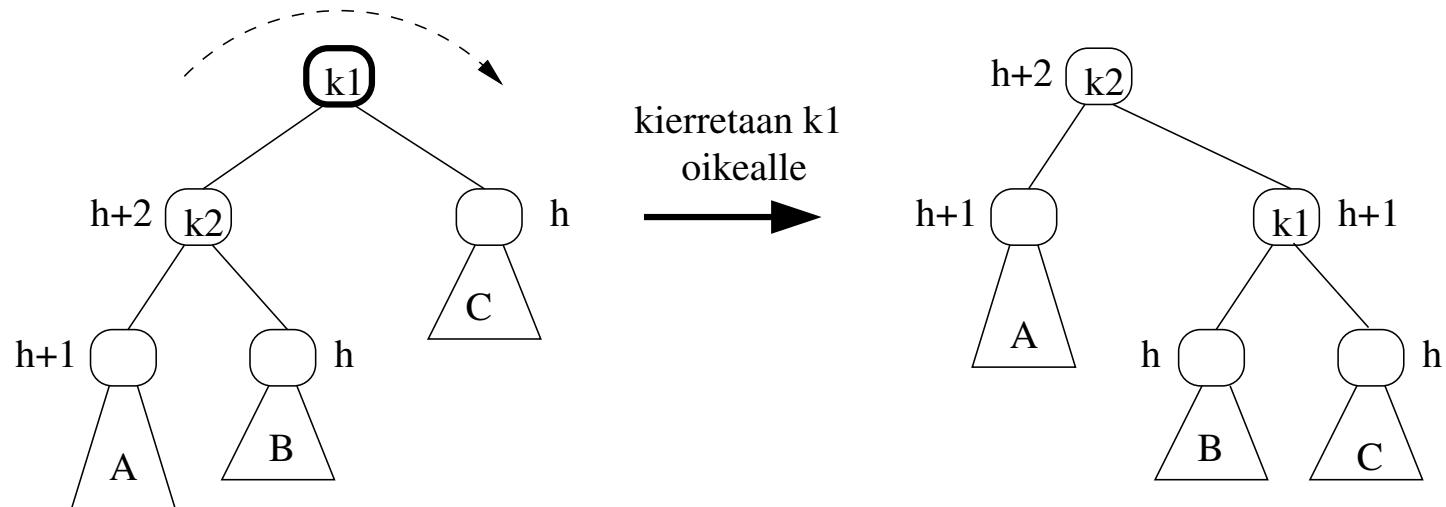
- Periaatteena AVL-puun lisäysoperaatioissa on tehdä ensin lisäys kuten normaaliin binäärihakupuuhun. Jos AVL-ehdon havaitaan menneen rikki lisäyksen yhteydessä, puu korjataan tekemällä sopivat [kierto-operaatiot](#)
- Koska lisäys vaikuttaa ainoastaan lisätyn solmun esi-isien korkeuteen, epätasapainoon 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



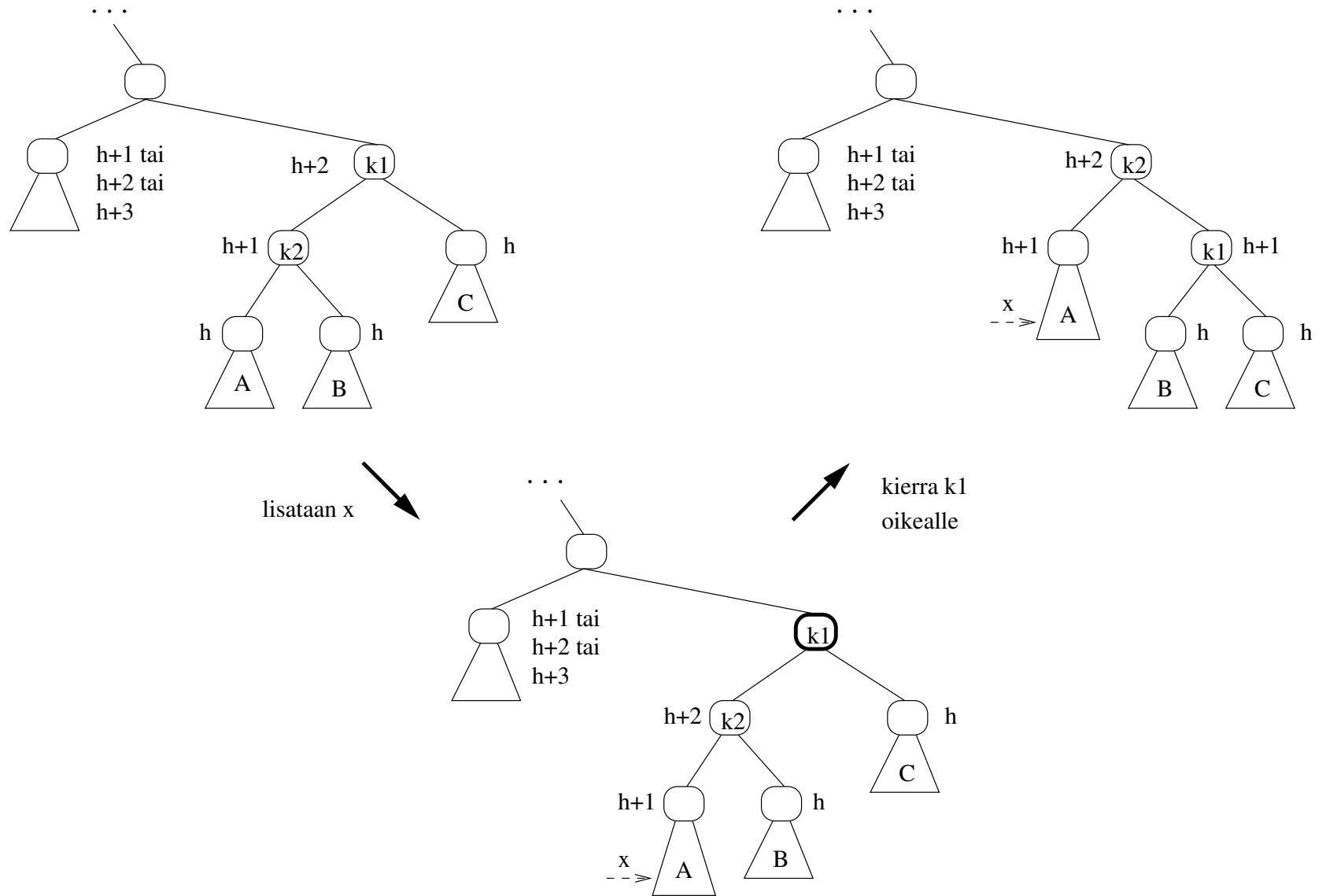
- Epätasapainoa solmuun k_1 ei olisi syntynyt ellei puun tilanne ennen lisääystä olisi ollut täsmälleen edellisen sivun vasemmalla olevan kaltainen
 - Jos B :n korkeus olisi ollut $h - 1$, niin k_2 olisi mennyt epätasapainoon. Oletimme kuitenkin, että k_1 on syvimmillä epätasapainossa oleva solmu
 - B :n korkeus ei voinut olla myöskään $h + 1$ sillä silloin k_1 olisi ollut epätasapainossa jo ennen lisääystä
 - Jos A :n korkeus olisi ollut $h - 1$ ei epätasapainoa olisi syntynyt
 - A :n korkeus ei voinut olla myöskään $h + 1$ sillä silloin k_1 olisi ollut epätasapainossa jo ennen lisääystä
 - Alipuiden A :n ja B :n korkeuksien siis täytyi ennen lisääystä olla h
 - C :n korkeus ei olisi voinut olla $h - 1$, sillä silloin k_1 olisi ollut epätasapainossa jo ennen lisääystä
 - C :n korkeus ei olisi voinut myöskään ollut $h + 1$ sillä siinä tapauksessa epätasapainoa ei olisi syntynyt
 - myös alipuun C korkeuden täytyi ennen lisääystä olla h

- Tasapaino palautetaan tekemällä **kierto oikealle** epätasapainossa olevan solmun k_1 suhteen
 - k_1 :sta tulee sen vasemman lapsen k_2 oikea lapsi
 - k_2 :n oikea alipuu B siirtyy k_1 :n vasemmaksi alipuuksi
 - kierto oikealle säilyttää puun binäärihakupuuna sillä alipuu B pysyy edelleen solmun k_1 vasemmalla puolella ja k_2 :n oikealla puolella

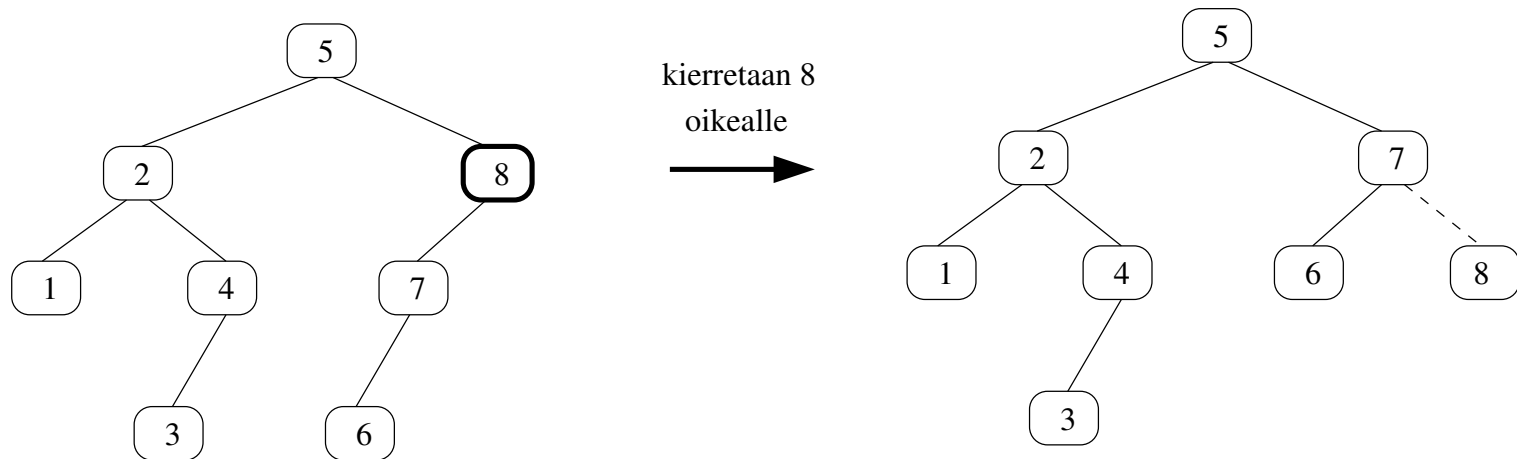


- Huomaamme, että kierto palauttaa puun takaisin tasapainoon (ks. seur. sivu)
 - koko alipuun korkeus on nyt sama kuin ennen lisäysoperaatiota
 - koska oletimme, että k_1 on alin epätasapainoinen solmu, ei puussa kierron jälkeen enää voi olla epätasapainoisia solmuja

- Epätasapainoisen solmun kierto korjaa koko puun tasapainon:

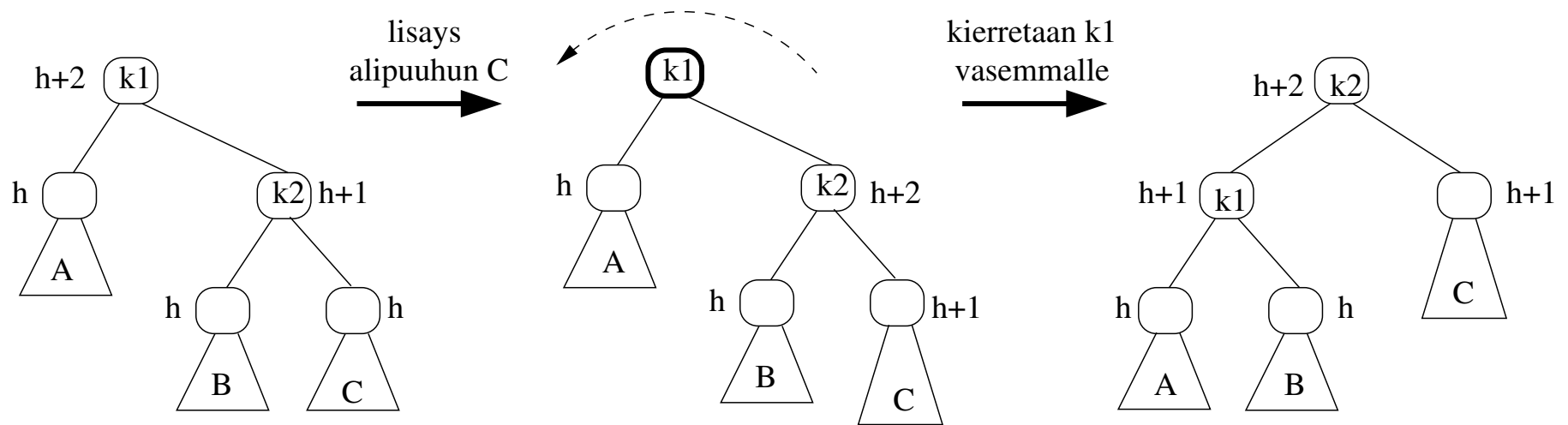


- Kierto oikealle selvittää myös muutama sivu sitten aiheuttamamme epätasapainon
- Nyt solmu 7 nousee kierrettävän solmun 8 vanhemmaksi



- Kierron seurauksena solmusta 7 siis tulee epätasapainoon menneen alipuun uusi juuri
- Kierron yhteydessä kierretyn alipuun uusi juuri on siis kiinnitettävä vanhempansa lapseksi

- Huomataan, että tilanne, jossa solmun $k1$ epätasapainon aiheuttaa lisäys sen oikean lapsen $k2$ oikeaan alipuuhun, on symmetrisen edellisen kanssa
- Solmun $k1$ joka siis on syvimmällä oleva epätasapainoon joutunut solmu kiertäminen vasemmalle palauttaa puun tasapainoon



- Kierto-operaatioiden toteutus on suhteellisen suoraviivainen

RightRotate(k1)

```
1  k2 = k1.left
2  k2.parent = k1.parent
3  k1.parent = k2
4  k1.left = k2.right
5  k2.right = k1
6  if k1.left ≠ NIL
7     k1.left.parent = k1
8  k1.height = max( Height(k1.left), Height(k1.right) ) + 1
9  k2.height = max( Height(k2.left), Height(k2.right) ) + 1
10 return k2
```

- Riveillä 2-5 $k2$:sta tulee alipuun uusi juuri, $k1$:stä sen oikea alipuu ja $k1$ saa vasemmaksi alipuuksi $k2$:n oikean alipuun
- Rivillä 6-7 asetetaan siirtyneelle alipuulle oikea vanhempi jos alipuu ei ollut tyhjä
- Riveillä 8 ja 9 päivitetään solmujen korkeuden muistavat *height*-attribuutit
- Viimeisellä rivillä palautetaan kierretyn alipuun uusi juuri, näin on tehtävä, jotta alipuu saadaan laitettua vanhempansa lapseksi

- Kierto-operaatio on selvästi vakioaikainen: puun koosta riippumatta suoritettavia koodirivejä on saman verran. Myös tilavaativuus on vakio sillä käytössä on ainoastaan 2 apumuuttujaa
- Analyysissä on tietysti huomioitava myös kierron käyttämät operaatiot **Height** ja **max** (funktio, joka palauttaa argumenteista suurimman), jotka ovat selvästi vakioaikaisia ja -tilaisia
- Kierto-operaation kutsujan vastuulle siis jää liittää kierretty alipuu vanhempansa lapseksi
- Kutsu voisi esim. tapahtua seuraavasti:

```

...
1  if solmu p epätasapainossa
2      vanhempi = p.parent
3      alipuu = RightRotate(p)
4      if vanhempi.left == p
5          vanhempi.left = alipuu
6      else vanhempi.right = alipuu
...

```

- Kierron seurauksena p siis ei ole enää alipuun juuri
- Riveillä 4-6 asetetaan alipuun uusi juuri vanhempansa oikeaksi tai vasemmaksi lapseksi sen mukaan, oliko p vasen vai oikea lapsi

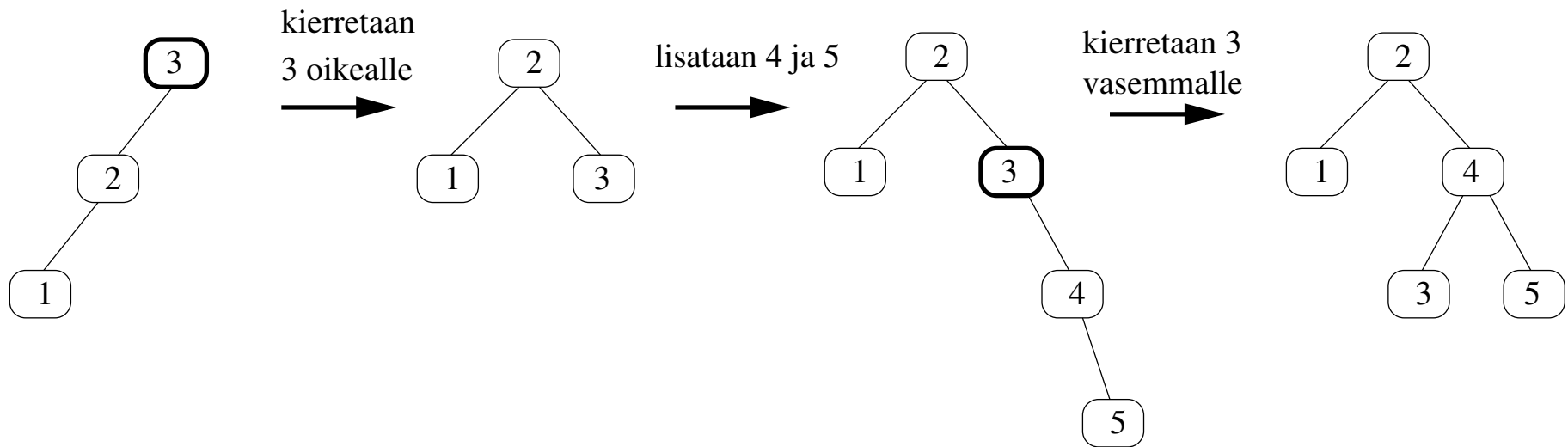
- Kierto vasemmalle on edellisen kanssa symmetrinen

LeftRotate(k1)

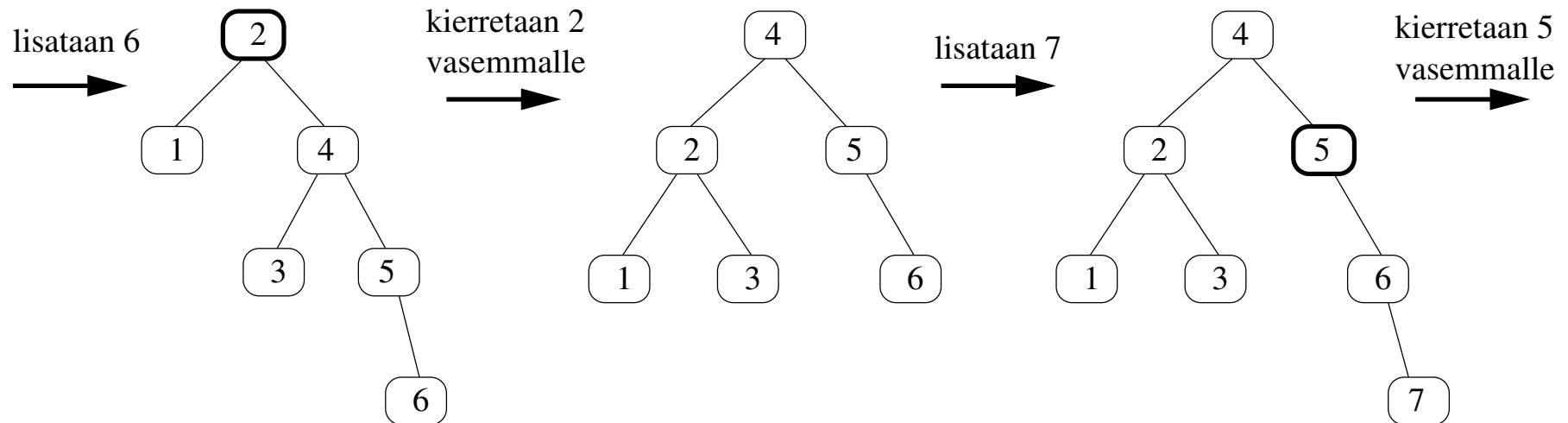
```
1  k2 = k1.right
2  k2.parent = k1.parent
3  k1.parent = k2
4  k1.right = k2.left
5  k2.left = k1
6  if k1.right ≠ NIL
7     k1.right.parent = k1
8  k1.height = max( Height(k1.left), Height(k1.right) ) + 1
9  k2.height = max( Height(k2.left), Height(k2.right) ) + 1
10 return k2
```

- Nyt siis tiedämme miten puu saadaan tasapainotettua, jos solmuun k_1 epätasapainon aiheuttama lisäys tehdään
 - k_1 :n oikean lapsen k_2 oikeaan alipuuhun
 - k_1 :n vasemman lapsen k_2 vasempaan alipuuhun
- Ensimmäisessä tapauksessa siis kierretään solmua k_1 vasemmalle ja toisessa tapauksessa oikealle. Yksi kierto-operaatio palauttaa puun tasapainoon
- Ennen kuin tarkastelemme muita tapoja epätasapainon syntytapoja, käydään läpi konkreettinen esimerkki

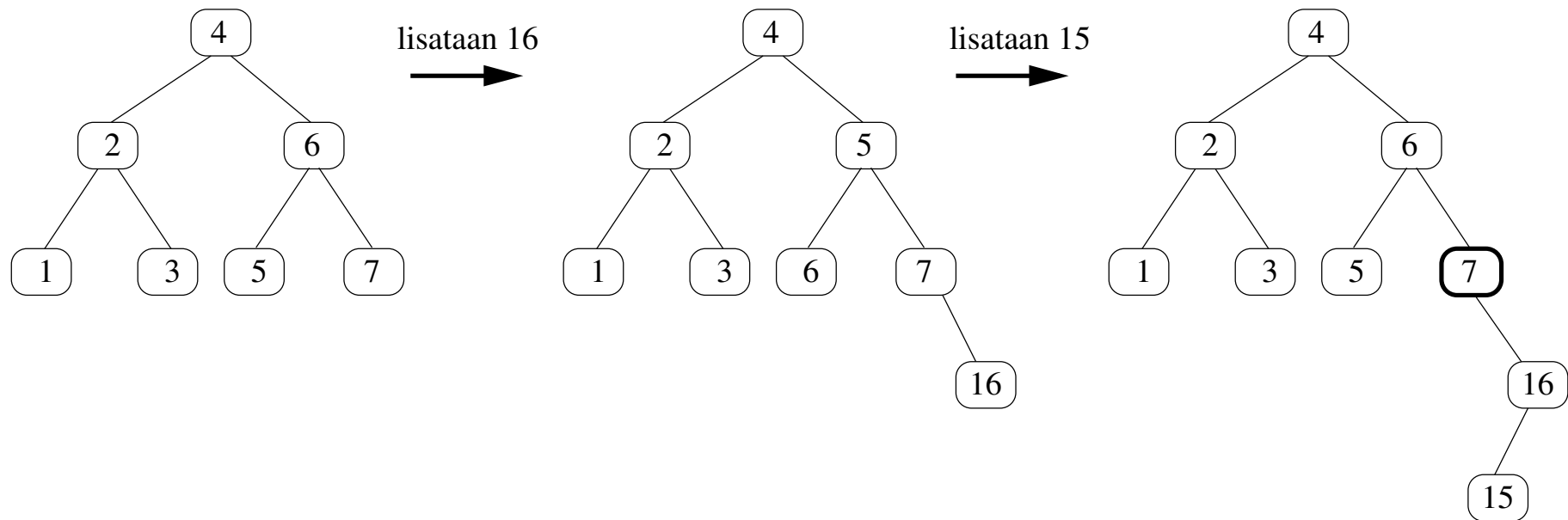
- Puuhun lisätään aluksi 3, 2 ja 1. Viimeinen lisäys aiheuttaa epätasapainon solmuun 3
- Epätasapainon aiheuttama lisäys tehtiin epätasapainossa olevan solmun 3 vasemman lapsen vasempaan alipuuhun. Edellä päätellyn perusteella ongelma ratkeaa kiertämällä epätasapainoista solmua 3 oikealle
- Lisätään puuhun vielä 4 ja 5. Jälkimmäinen lisäys aiheuttaa epätasapainon solmuun 3
- Tällä kertaa tilanteesta selvittää vasemmalle kierrolla, sillä epätasapainon aiheuttaja oikean lapsen oikeassa alipuussa



- Jatketaan lisäämällä puuhun 6. Tämä vie juuren, eli solmun 2 epätasapainoon. Jälleen epätasapainon aiheuttaja on lisäys oikean lapsen oikeaan alipuuhun ja kierto vasemmalle korjaa tilanteen.
- Puuhun lisätään vielä 7. Tämä vie solmun 5 epätasapainoon ja korjaus on kierto vasemmalle. Kierron jälkeen lopputuloksena on täydellinen, eli maksimaalisen tasapainoinen avaimet 1, ..., 7 sisältävä puu, joka on esitetty seuraavalla sivun vasemmanpuoleisessa kuvassa

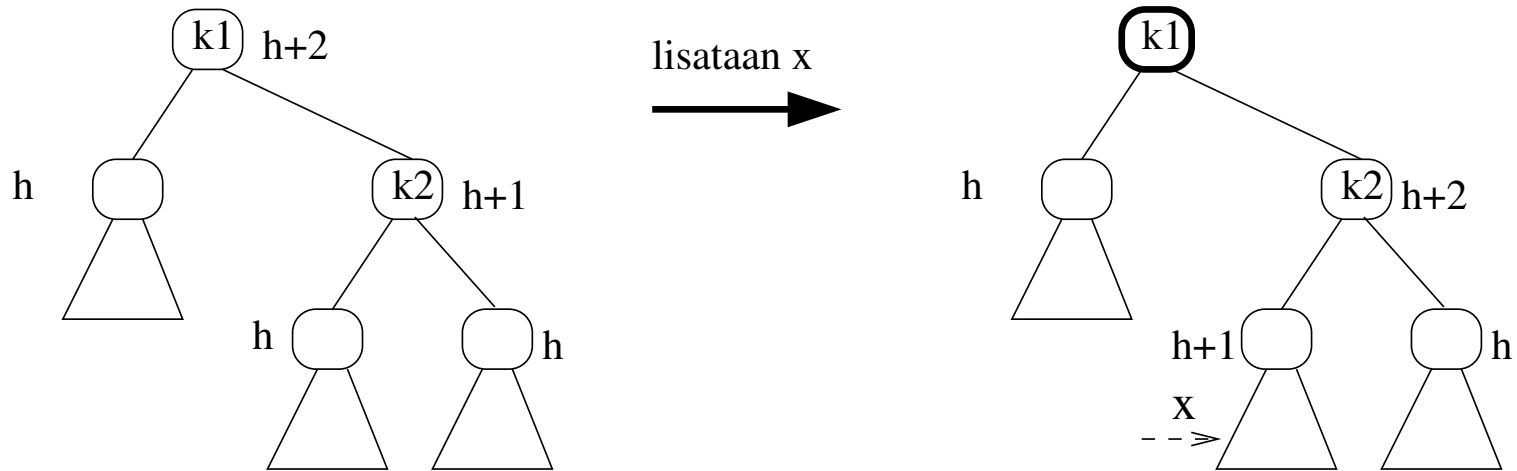


- Jatketaan lisäämällä puuhun 16. Lisäys ei riko tasapainoa
- Kun nyt lisätään puuhun 15 joutuu solmu 7 epätasapainoon
- Epätasapainon aiheuttaa nyt **lisäys oikean lapsen vasempaan alipuuhun**
- Tilanne eroaa aiemmista epätasapainon syistä eli **lisäys oikean lapsen oikeaan alipuuhun** ja **lisäys vasemman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



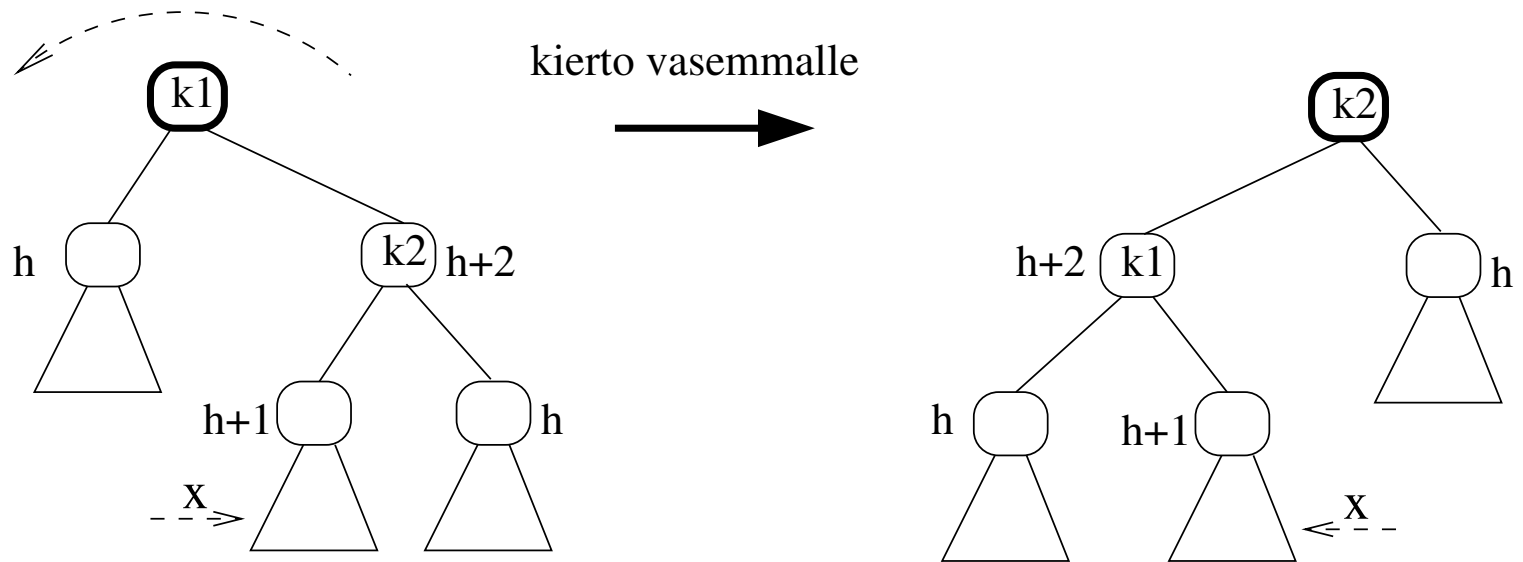
- Kumpikaan kierto-operaatio ei näytä korjaavan solmun 7 epätasapainoa

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



- Epätasapaino solmuun $k1$ ei olisi syntynyt, ellei puun tilanne ennen lisäystä olisi ollut täsmälleen vasemmalla oleva, tämän voi päätellä sivun 313 tapaan

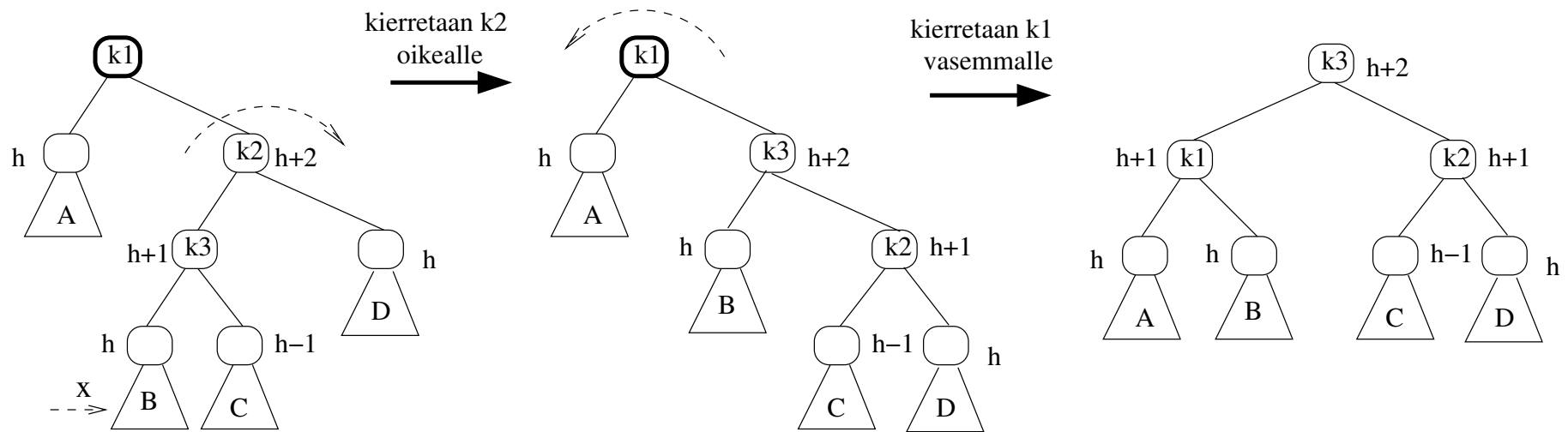
- Solmun $k1$ kierto oikealle ei ainakaan voi ratkaista ongelmaa, sillä se ainoastaan pahentaa epätasapainoa
- Seuraavasta huomaamme, että $k1$:n kierto vasemmalle ainoastaan muuttaa tilanteen peilikuvaksi: alipuu pysyy epätasapainossa, sillä $k2$:sta tulee epätasapainoinen ja syynä on se, että uusi solmu x on sen vasemman lapsen oikeassa alipuussa



- On keksittävä jokin muu ratkaisu

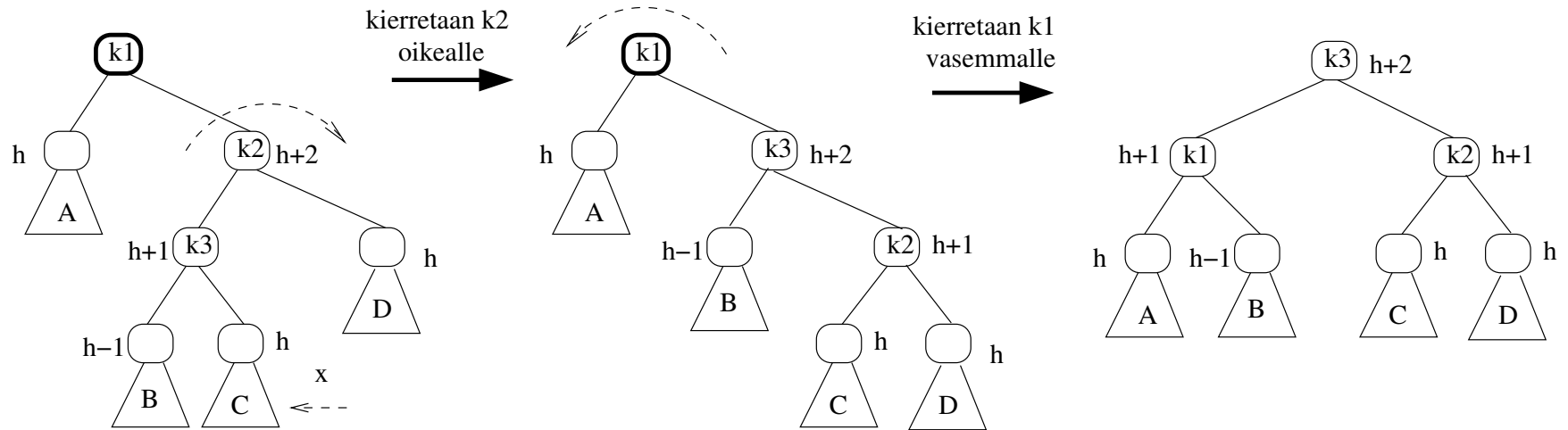
- Solmun k_1 epätasapainon siis aiheuttaa sen oikean lapsen k_2 vasempaan alipuuhun tehty lisäys. Merkitään tämän juurta k_3 :lla
- Lisäys voi olla tehty joko k_3 :n oikeaan tai vasempaan alipuuhun
- Osoittautuu, että kumpikin näistä tapauksista korjautuu samalla tekniikalla
- Tarkastellaan ensin tilannetta, jossa lisäys on tehty k_3 :n vasempaan alipuuhun, jota merkitään seuraavan sivun kuvassa B :llä
- Tekemällä epätasapainoisen solmun k_1 :n oikealle lapselle k_2 ensin kierto oikealle ja sitten k_1 :lle kierto vasemmalle saadaan puu tasapainoon

- Kiertämällä epätasapainoisen lapsi k_2 oikealle luodaan tilanne, jossa epätasapainon syy on k_1 :n oikean lapsen oikeassa alipuussa
- Tämä on sama tilanne, johon törmäsimme aiemmin, eli epätasapainon korjaa k_1 :n kierto vasemmalle



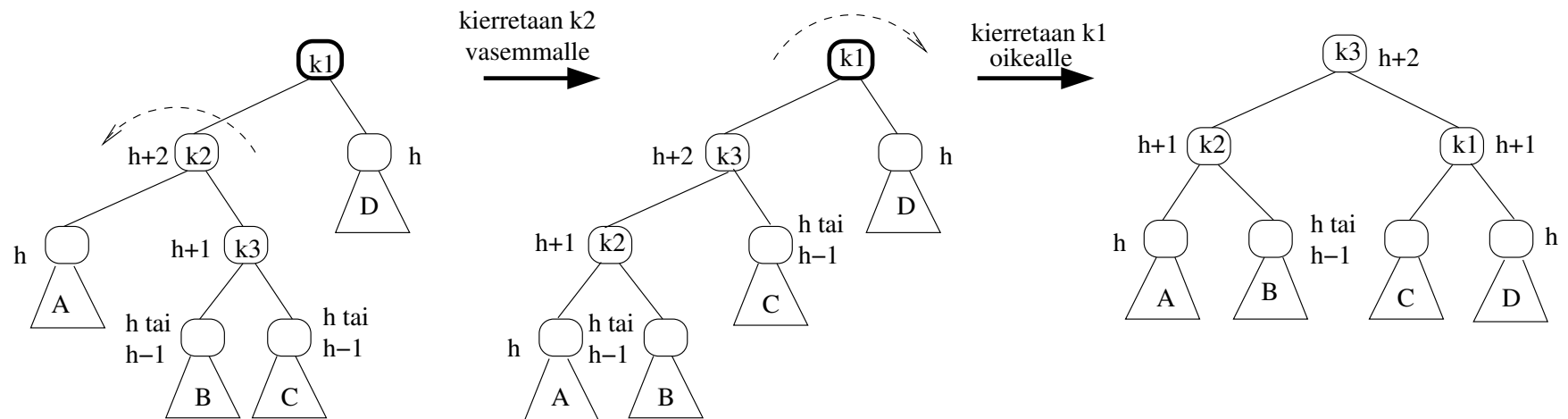
- Kahden kierto-operaation seurauksena koko alipuun korkeus on sama kuin ennen lisäystä
- Koska siis oli oletettu, että k_1 on alin epätasapainoinen solmu, on kiertojen jälkeen puu taas tasapainossa

- 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 k_1 epätasapainon aiheuttaa sen vasemman lapsen k_2 oikeaan alipuuhun tehty lisäys, puu tasapainoituu tekemällä ensin vasen kierto k_2 :lle ja sitten oikea kierto k_1 :lle
- Näiden operaatioiden muodostamaa kokonaisuutta sanotaan vasen-oikea-kaksoiskierroksi
- Seuraavassa vielä näemme miten tilanne, jossa lisäys on tehty k_3 :n oikeaan alipuuhun, ratkeaa kahdella kierrolla



- Kaksoiskierto-operaatioiden toteuttaminen on helppoa, riittää kutsua normaalia kiertoa oikeille solmuille

RightLeftRotate(k1)

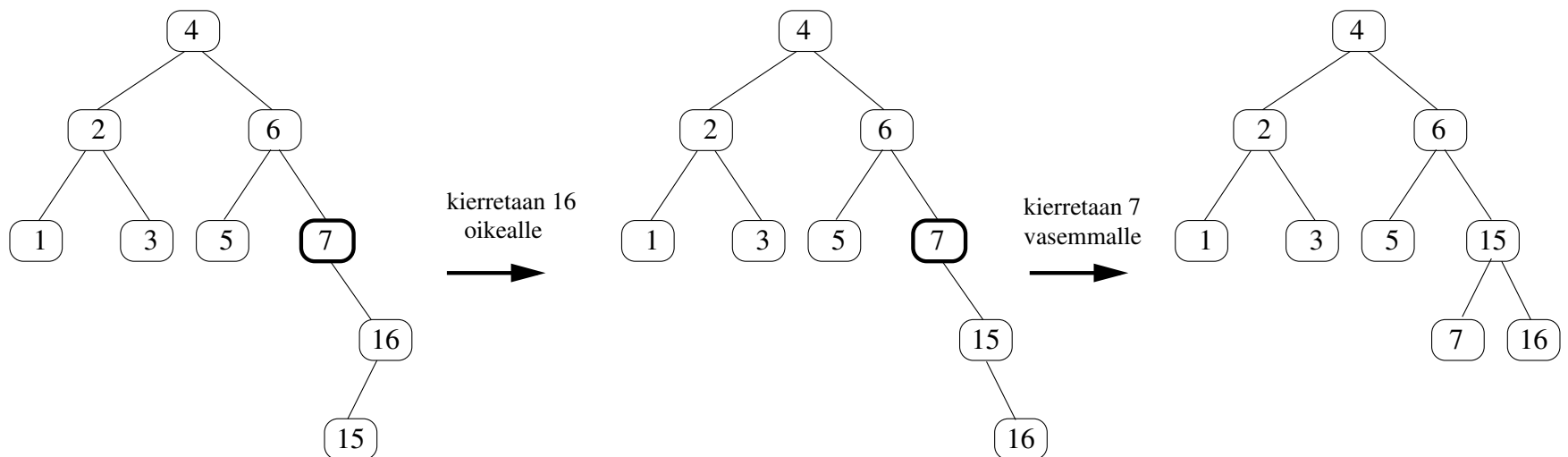
```
1 k2 = k1.right
2 k1.right = RightRotate(k2);
3 return LeftRotate(k1);
```

LeftRightRotate(k1)

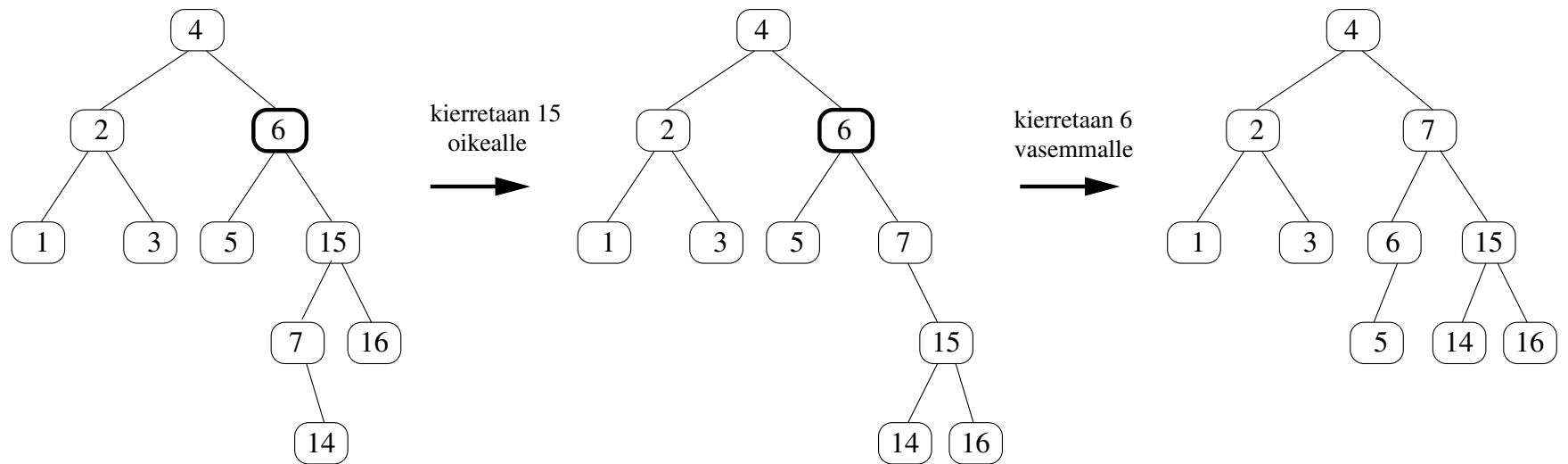
```
1 k2 = k1.left
2 k1.left = LeftRotate(k2);
3 return RightRotate(k1);
```

- Myös kaksoiskierrossa on alipuun uusi juuri palautettava, jotta kutsuja pystyy laittamaan sen vanhempansa lapseksi
- Koska yksittäiset kierto-operaatiot ovat vakioaikaisia ja vakiotilaisia, on selvää, että myös kaksoiskierrot vain vakioaikaisia ja vakiotilaisia operaatioita
- Ennenkuin esitämme AVL-puuhun lisäyksen pseudokoodina, jatketaan jo aloittamaamme esimerkkiä

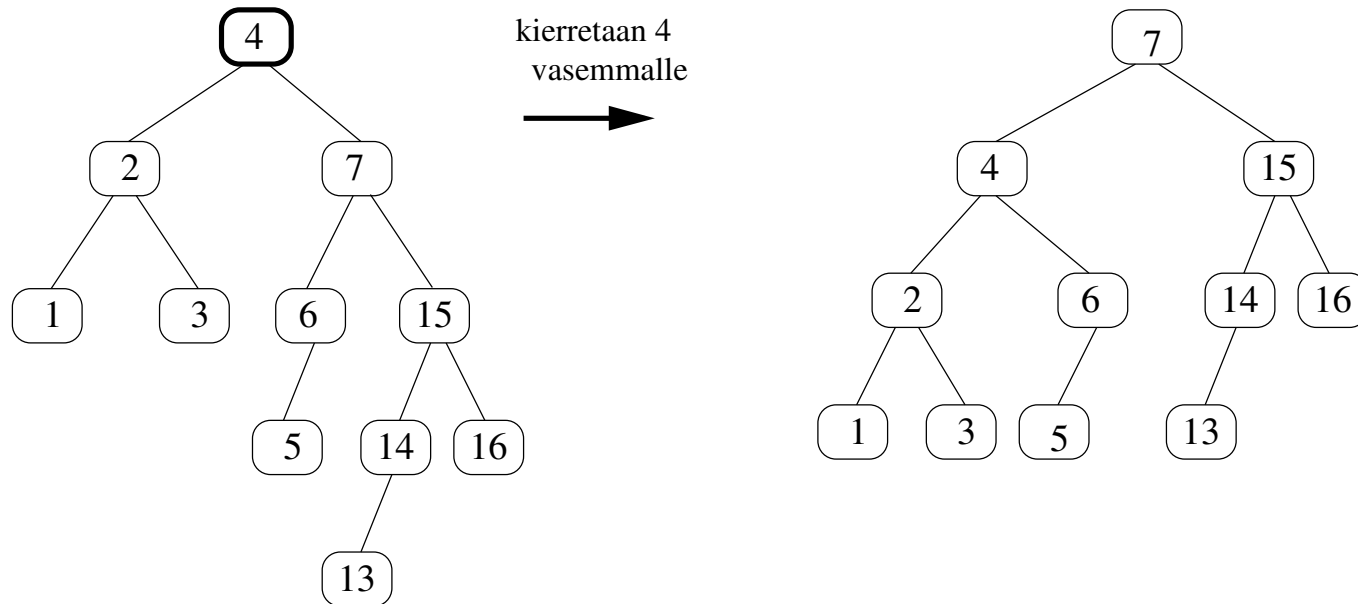
- Jäimme tilanteeseen, missä solmujen 15 ja 16 lisäys aiheutti solmuun 7 epätasapainon
- Epätasapaino johtuu oikean lapsen vasempaan alipuuhun suoritetusta lisäyksestä
- Äsken havaitun perusteella oikea-vasen-kaksoiskierto tasapainottaa puun, eli
 - Ensin kierretään oikeaa lasta 16 oikealle, ja sen jälkeen
 - kierretään epätasapainossa olevaa solmua 7 vasemmalle
- Näin puun tasapaino palautuu



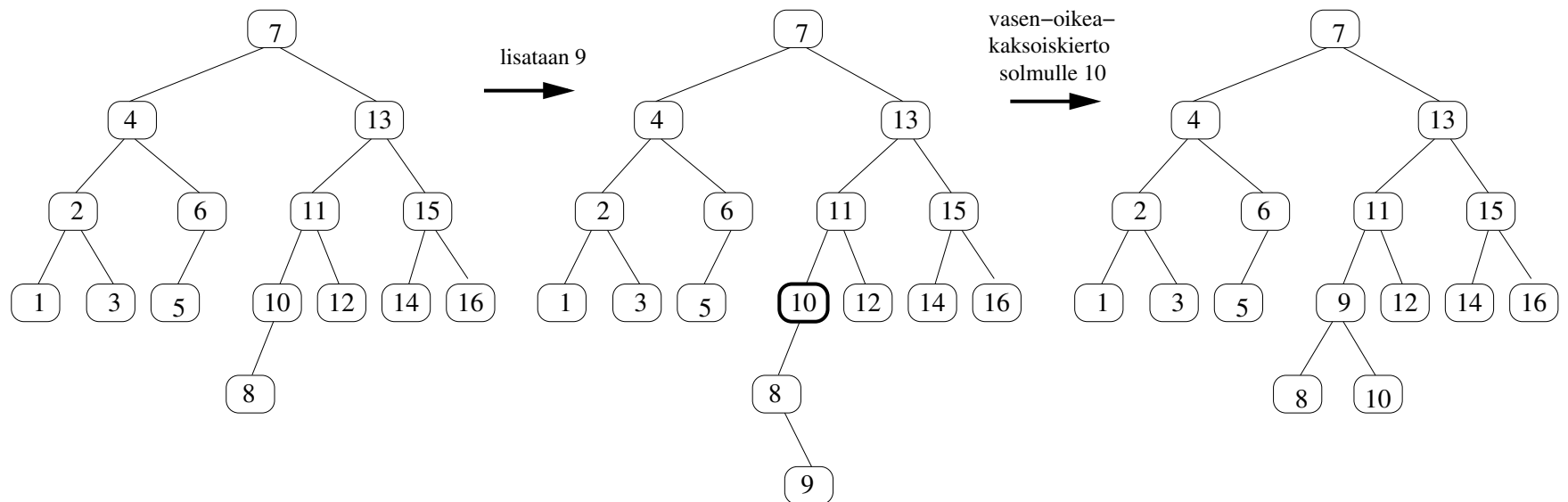
- Lisätään 14. Solmu 6 menee epätasapainoon. Syy epätasapainolle on lisäys oikean lapsen vasempaan alipuuhun.
- Ratkaisu ongelmaan on siis oikea-vasen-kaksoiskierto, eli ensin oikeaa lasta 15 kierretään oikealle ja sen jälkeen solmua 6 vasemmalle



- Lisätään 13. Puun juuri, eli solmu 4 menee epätasapainoon. Syy epätasapainolle on lisäys oikean lapsen oikeaan alipuuhun
- Kyseessä on siis helpompi tapaus, joka korjautuu yhdellä kierrolla, eli kierretään juurisolmua vasemmalle ja tasapaino palautuu

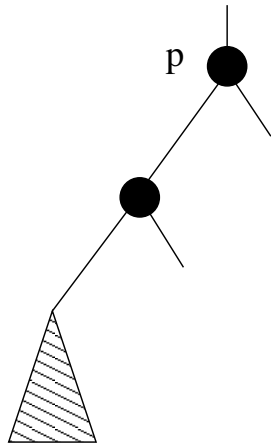


- Jos jatketaan lisäämällä solmut 12, 11 ja 10, pitää kaikkien lisäyksiä yhteydessä tehdä yksi kierto
- Edellisten jälkeen solmun 8 voi lisätä puuhun ilman tasapainon rikkoutumista Tuloksena on kuvassa vasemmalla oleva puu
- Jos tähän vielä lisätään solmu 9, solmu 10 menee epätasapainoon. Syynä sen vasemman lapsen oikeaan alipuuhun tehty lisäys. Tilanne korjautuu vasen-oikea-kaksoiskierrolla, eli ensin 8 vasemmalle ja sen jälkeen 10 oikealle

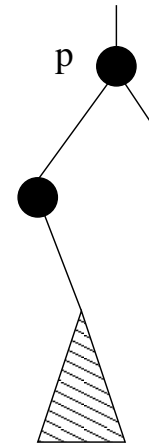


- Esitetään AVL-puun tasapainossa pitävä lisäysoperaatio vielä pseudokoodina
- Lisäys siis tapahtuu seuraavasti
 - lisätään uusi solmu puuhun kuten normaalin binäärihakupuun yhteydessä, asetaan uuden solmun korkeudeksi 0
 - lisäys saattaa viedä puun epätasapainoon
 - epätasapainoon joutuneiden solmujen 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 alimmana puussa oleva epätasapainoinen solmu (eli se joka tulee ensimmäisenä vastaan reitillä lisätystä juureen) tasapainoon
 - eli jos ensimmäisenä vastaan tuleva epätasapainotilanne korjataan, menee puu tasapainoon
 - lisäys on voinut muuttaa lisätyn esivanhempien korkeutta, joten kuljettaessa juurta kohti on matkalla vastaan tulevien solmujen korkeuskentät päivitettävä
- Seuraavalla sivulla vielä kootusti epätasapainotilanteessa tarvittavat kierto-operaatiot

- Jos solmun p epätasapainon syy on sen vasemmassa alipuussa, tapauksia kaksi

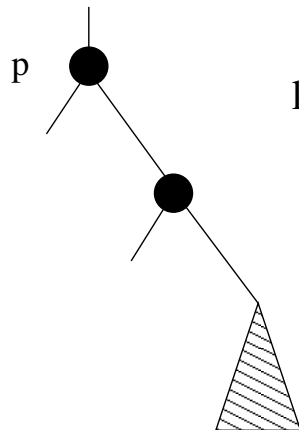


korjaava toimenpide:
RightRotate(p)

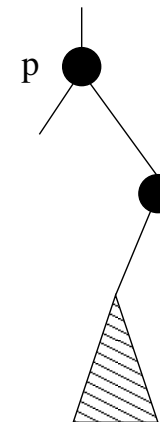


korjaava toimenpide:
LeftRightRotate(p)
eli
LeftRotate(p.left)
RightRotate(p)

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



korjaava toimenpide:
LeftRotate(p)



korjaava toimenpide:
RightLeftRotate(p)
eli
RightRotate(p.right)
LeftRotate(p)

- Esitetään algoritmi ensin korkealla tasolla. Algoritmi käyttää sivulla 284 esiteltyä tasapainottamattoman binääripuun lisäysoperaatiota jota on muokattu siltä osin, että se palauttaa viitteen lisättyyn solmuun ja asettaa *uusi.height* = 0

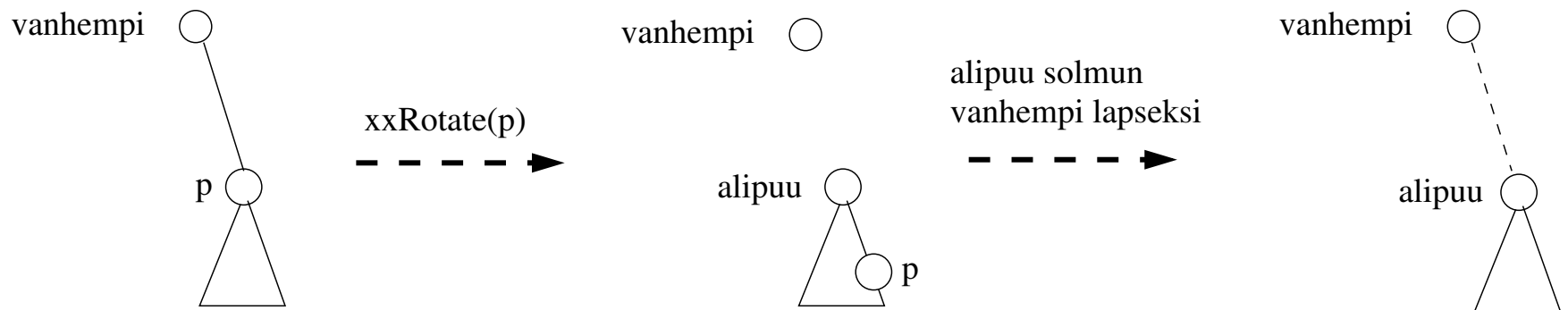
AVL-insert(T,k)

```

1  uusi = insert(T,k)
2  p = uusi.parent
3  while p ≠ NIL
4      if p epätasapainossa
5          vanhempi = p.parent
6          if epätasapainon syy vasemman lapsen vasen alipuu
7              alipuu = RightRotate(p)
8          elsif epätasapainon syy vasemman lapsen oikea alipuu
9              alipuu = LeftRightRotate(p)
10         elsif epätasapainon syy oikean lapsen oikea alipuu
11             alipuu = LeftRotate(p)
12         else // epätasapainon syy oikean lapsen vasen alipuu
13             alipuu = RightLeftRotate(p)
14         aseta alipuu solmun vanhempi lapseksi
           tai jos p oli juuri, niin tee alipuu:sta uusi juuri
15         vanhempi.height = max(Height(vanhempi.left),Height(vanhempi.right))-
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
- Toistolauseetta 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
```

- **AVL-insert**(T,k) jatkuu...

```
    // 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 tasapainossa ja korkeus sama kuin ennen
34  p.height = max(Height(p.left), Height(p.right) )+1
35  p = p.parent // jatketaan kohti juurta
```

- Rivit 5-16 hoitavat tilanteen, jossa solmun p epätasapaino johtuu sen vasemmasta alipuusta
 - Rivillä 7 käsitellään tilanne jossa epätasapainon syy vasemman lapsen vasen alipuu
 - Rivillä 9 tilanne jossa epätasapainon syy vasemman lapsen oikea alipuu
 - Rivit 10-16 laittavat kierretyn alipuun sen vanhemman lapseksi
 - Rivi 10 huomioi heti erikoistapauksen, jossa kierrettiin puun juurta
- Riveillä 19-32 tilanteen, jossa solmun p epätasapainon syy oikeassa alipuussa
 - Rivillä 21 tilanne jossa epätasapainon syy oikean lapsen oikea alipuu
 - Rivillä 23 tilanne jossa epätasapainon syy oikea lapsen vasen alipuu
 - Rivit 25-30 laittavat kierretyn alipuun sen vanhemman lapseksi
- Noustaessa lisätystä kohti juurta, 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, korkeus ei ole muuttunut, ja algoritmi lopettaa (rivit 18 ja 32)
- Muussa tapauksessa jatketaan solmujen tasapainon tutkimista aina juureen asti

- Analysoidaan algoritmin aika- ja tilavaativuutta
- Käytetään analyysin pohjana sivun 337 abstraktimpaa versiota
- Ensin kutsutaan normaalia binäärihakupuun lisäysoperaatiota, jonka aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen. Olemme osoittaneet aiemmin, että AVL-puun korkeus solmujen lukumäärän n suhteen on $\mathcal{O}(\log n)$, joten insert vie aikaa siis logaritmisesti
- **AVL-insert** suorittaa maksimissaan kaksi kierto-operaatiota, jotka molemmat ovat vakioaikaisia
- Pahimmassa tapauksessa algoritmi myös kulkee reitin lisäystä solmusta juureen, eli puun korkeuden verran, tämä on vaativuudeltaan myös $\mathcal{O}(\log n)$
- Eli algoritmi koostuu kahdesta $\mathcal{O}(\log n)$ aikaa vievästä osasta, siispä kokonaisaikavaativuus $\mathcal{O}(\log n)$
- Binäärihakupuun insertin ja kierto-operaatioiden tilavaativuus on todettu jo aiemmin vakioksi, samoin muutkin osat **AVL-insert**:istä käyttävät ainoastaan vakiomäärän apumuuttujia, eli kokonaisuudessaan algoritmin tilavaativuus on vakio

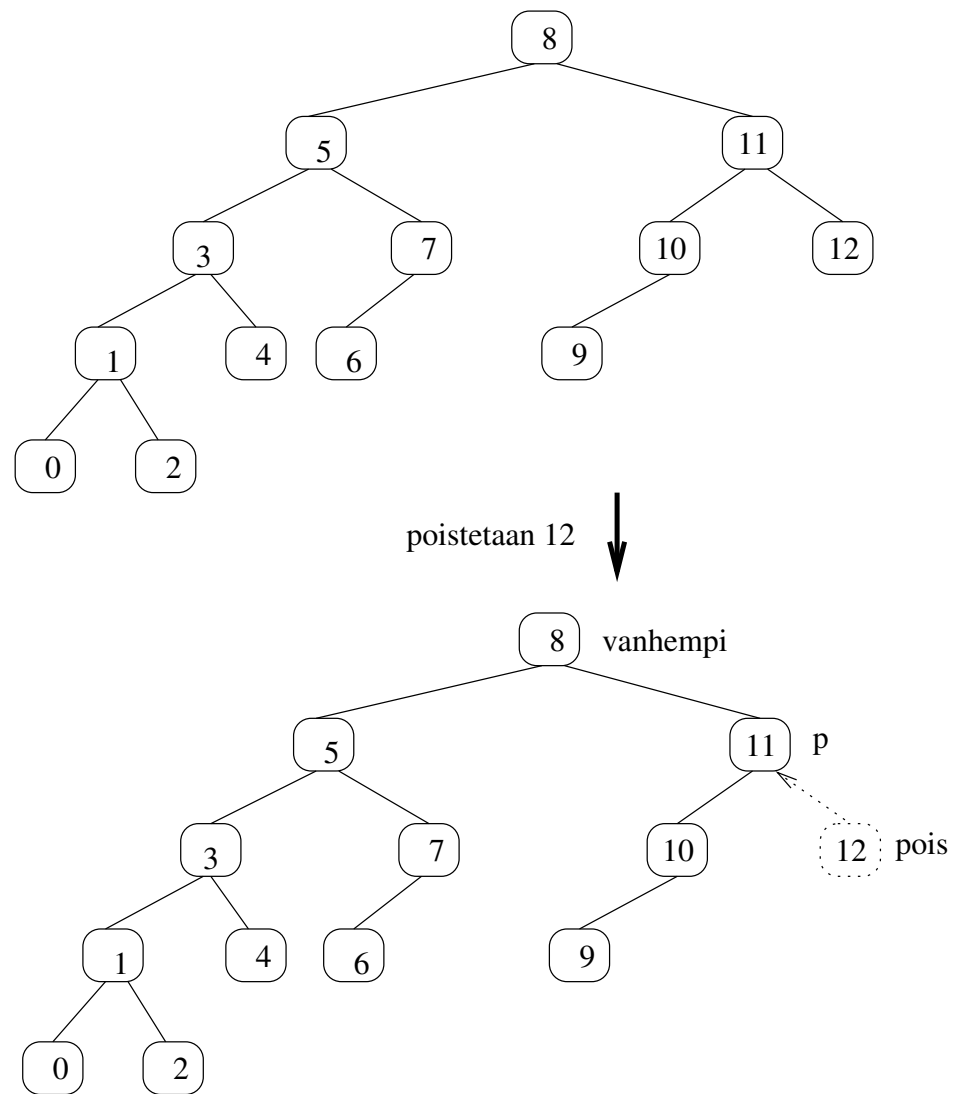
- **AVL-delete** on hiukan **AVL-insert**:iä monimutkaisempi operaatio
- Solmu poistetaan ensin käyttäen normaalia **delete**-operaatiota
- Algoritmi käyttää sivuilla 289–290 esiteltyä normaalin binääripuun poisto-operaatiota, joka palauttaa viitteen siihen solmuun, joka puusta todellisuudessa poistettiin (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 vastaan tulevan epätasapainoisen solmun tasapainoitus ei välttämättä palauta koko puuta takaisin tasapainoon
- Epätasapainoisten etsimistä on jatkettava joka tapauksessa juureen asti
- Esitetään algoritmista vain abstrakti versio

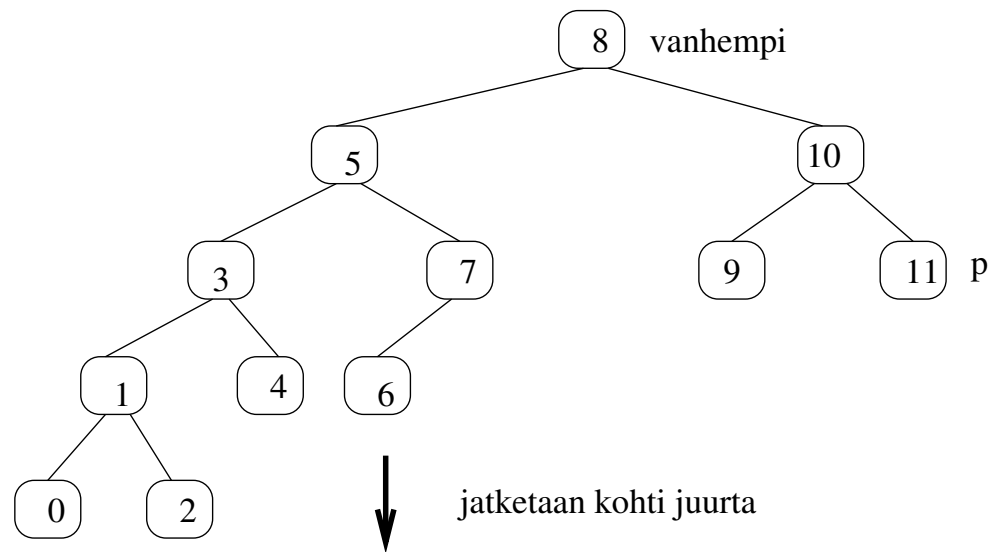
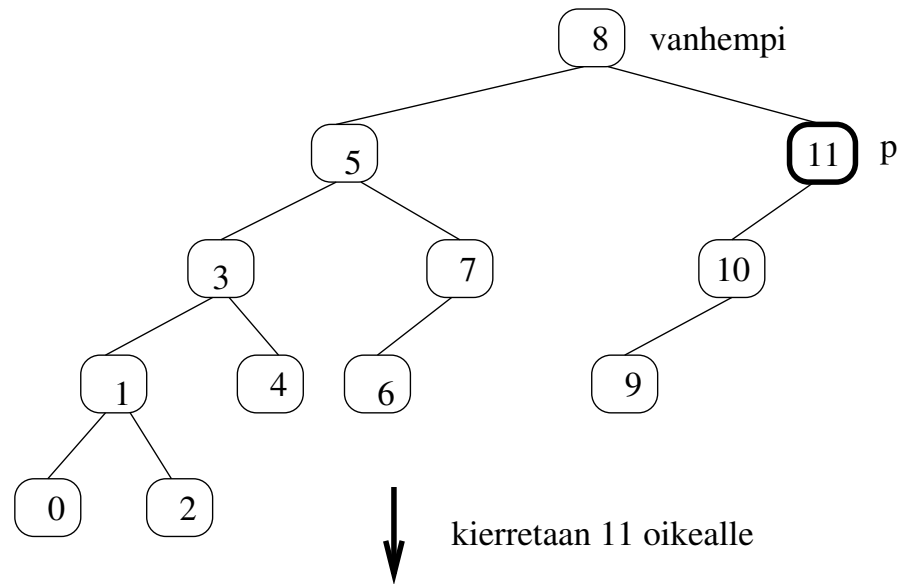
AVL-delete(T,x)

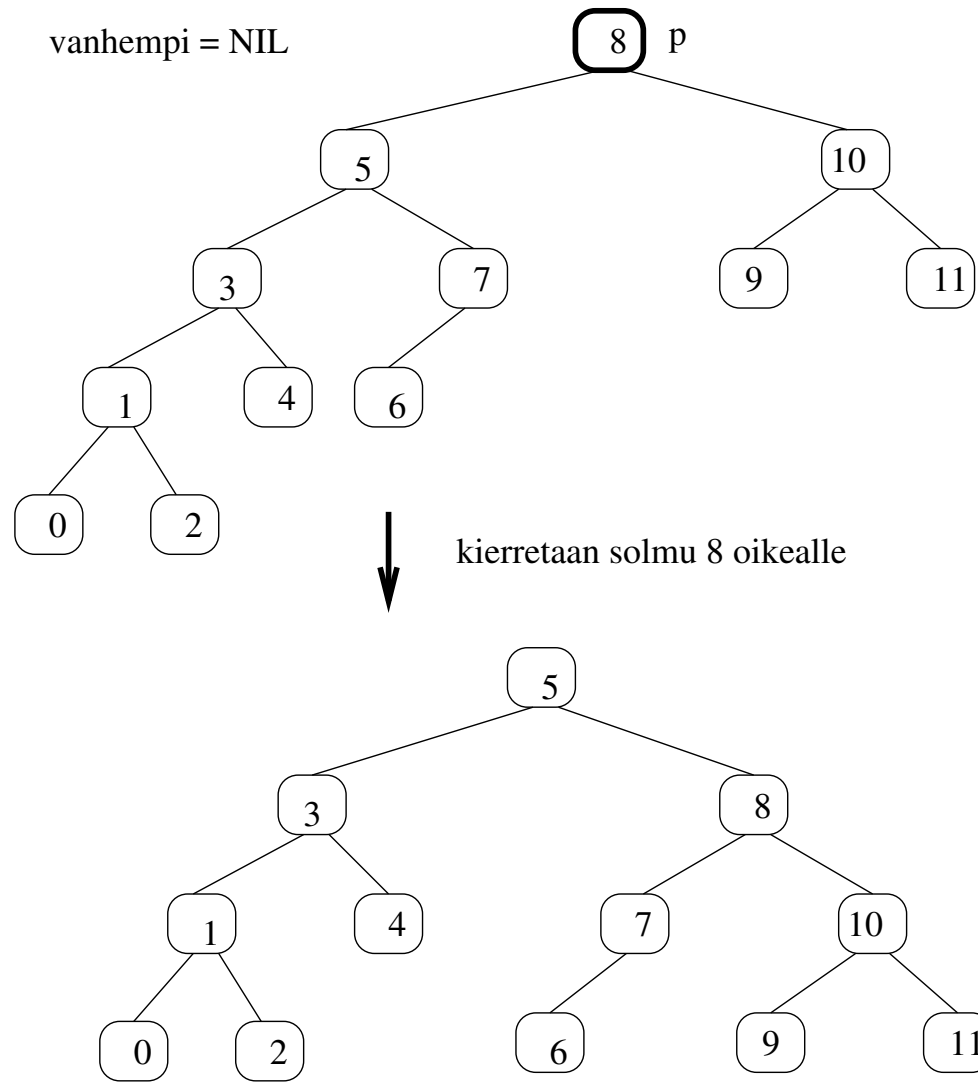
```
1  pois = delete(T,x)
2  p = pois.parent
3  while p ≠ NIL
4      if p epätasapainossa
5          vanhempi = p.parent
6          if epätasapainon syy vasemman lapsen vasen alipuu
7              alipuu = RightRotate(p)
8          elsif epätasapainon syy oikean lapsen oikea alipuu
9              alipuu = LeftRotate(p)
10         elsif epätasapainon syy vasemman lapsen oikea alipuu
11             alipuu = LeftRightRotate(p)
12         else // epätasapainon syy oikean lapsen vasen alipuu
13             alipuu = RightLeftRotate(p)
14         if p oli puun juuri
15             T.root = alipuu
16         return
17         aseta alipuu solmun vanhempi lapseksi
18         p = vanhempi
19     else
20         p.height = max( Height(p.left), Height(p.right) ) +1
21         p = p.parent
```


- Alussa siis kutsutaan normaalia binäärihakupuun **delete**-operaatiota, näin saadaan tietoon se solmu *pois*, joka todellisuudessa poistettiin puusta
- Rivillä 2 laitetaan apuviite p viittaamaan poistetun vanhempaan
- Rivillä 4 tarkastetaan, onko solmu johon p viittaa mennyt epätasapainoon
- Jos on, niin epätasapaino korjataan riveillä 5-19
 - riveillä 6-13 tutkitaan minkä tyyppisestä epätasapainosta on kyse ja tehdään korjaava (mahdollisimman yksinkertainen) kierto-operaatio
 - kierron seurauksena p :n alkuperäiselle vanhemmalle *vanhempi* tulee uusi lapsi: solmu *alipuu*, jonka kierto-operaatio palauttaa
 - riveillä 14-16 erikoistapaus, jossa kierrettävä solmu oli puun juuri
- Tapahtui tasapainoitus tai ei, jatkuu epätasapainoisten solmujen etsintä p :n vanhemmasta, etsintää jatketaan niin kauan kunnes kaikki solmut matkalla poistetusta puun juuren on käyty läpi
- Poisto-operaatio saattaa muuttaa poistettujen edeltäjien korkeuksia tämän takia noustaessa poistetusta kohti juurta, matkan varrella kohdattujen solmujen korkeuskentät päivitetään rivillä 20
- Kierto-operaatiot päivittävät kierrettyjen alipuiden korkeuskentät, joten kierron yhteydessä algoritmin ei tarvitse koskea korkeuskenttiin

- Tarkastellaan algoritmin toimintaa esimerkin avulla







- Analysoidaan algoritmin aika- ja tilavaativuutta
- Ensin kutsutaan normaalia binäärihakupuun poisto-operaatiota, jonka aikavaativuus on $\mathcal{O}(h)$ puun korkeuden h suhteen. Olemme osoittaneet aiemmin, että AVL-puun korkeus solmujen lukumäärän n suhteen on $\mathcal{O}(\log n)$, joten **delete** vie aikaa siis logaritmisesti
- **AVL-delete** suorittaa pahimmassa tapauksessa puun korkeuden verran kierto-operaatioita kulkiessaan poistetusta solmusta juureen
- Kierto-operaatiot ovat vakioaikaisia, eli tasapainon korjaukseen kuluu aikaa enintään $\mathcal{O}(\log n)$
- Algoritmi siis koostuu kahdesta $\mathcal{O}(\log n)$ aikaa vievästä osasta, eli kokonaisaikavaativuus $\mathcal{O}(\log n)$
- Eli vaikka operaatio onkin kiertojen takia hieman raskaampi kuin normaali poisto, ei tasapainon ylläpitäminen muuta operaation aikavaativuuden kertaluokkaa
- Binäärihakupuun poisto- ja kierto-operaatioiden tilavaativuus on todettu jo aiemmin vakioksi, samoin muutkin osat **AVL-delete**:ssä käyttävät ainoastaan vakiomäärän apumuuttujia, eli kokonaisuudessaan algoritmin tilavaativuus on vakio

Yhteenveto AVL-puista:

- AVL-tasapainoehto takaa puun korkeuden $\mathcal{O}(\log n)$, missä n on alkioiden lukumäärä
- Siis **search** toimii aina ajassa $\mathcal{O}(\log n)$
- Tasapainoehtoa pidetään yllä tekemällä vakioaikaisia kiertoja hakupolulla
- Pahimmassa tapauksessa tarvitaan yksi kierto-operaatio kussakin hakupolun solmussa
- Edellisestä seuraa, että kaikki AVL-puuna toteutetun joukon operaatiot toimivat pahimmassakin tapauksessa ajassa $\mathcal{O}(\log n)$
- Tasapainottamattomaan puuhun verrattuna AVL-puut tarvitsevat kuhunkin solmuun ylimääräisen *height*-laskurin, jolle käytännössä riittää 8 bittiä sillä puun korkeus on tuskin käytännössä koskaan yli 256
- Periaatteessa muistia voitaisiin hieman säästää tallentamalla vain vasemman ja oikean alipuun korkeuksien ero (-1 , 0 tai $+1$) eli tällöin tarvittaisiin vain 2 bittiä