MAT-71000 Tieto ja laskenta

Antti Valmari

Tampereen teknillinen yliopisto
Matematiikan laitos

1 Tiedon esittämistavoista
2 Lausekkeet, lauseet, ym.
3 Aliohjelmat ja abstraktiot
4 Algoritmien toimivuus ja nopeus
5 Tärkeimmät tietorakenteet
6 Algoritmien oikeaksi todistaminen
7 (Moni)graafit
8 Informaatioteoriaa
9 Tiedon pakkaus ja salaus
10 Ratkeamattomuus ja laskennallinen vaativuus
11 Kielet ja automaatit
12 Matemaattista logiikkaa
13 Rinnakkaisuus
14 Rinnakkaisuuden teoriaa
15 Tiedon louhinta ja tekoäly
16 Lopuksi
Merkintöjen hakemisto

Tämän esityksen käyttö opiskeluun ja opetukseen on sallittu seuraavin ehdoin:

- *on käytettävä aina uusinta saatavilla olevaa versiota*
  - *riittää ladata uusin versio lukukauden alussa*
- *esitystä ei anneta eteenpäin, vaan annetaan linkki siihen*
- *lähde on mainittava*

Tällä opintojaksolla tutustutaan tietoa ja laskentaa koskeviin ilmiöihin laaja-alaisesti *luonnontieteellisestä* näkökulmasta

- tavoitteena ei ole oppia ohjelmointia tai ohjelmistojen suunnittelua
  - ilmiöistä ei kuitenkaan voi puhua ilman riittävää tietoa ohjelmoinnista
  - ⇒ monia käytännössäkin hyödyllisiä ohjelmoinnin asioita käsitellään
- kiinnostuksen kohteena ovat ilmiöt itsessään

Aineisto: nämä luentokalvot

- mukaan on pyritty saamaan monenlaisille opiskelijoille kiintoisaa asiaa
  - arvosanan 1 kynnyksen ei ole tarkoitus olla korkea
  - arvosana 5 vaatii matemaattisia taitoja
- tähän esitykseen saattaa tulla korjauksia ja muutoksia

Esityksessä käytettävä värikoodaus

- erityisen tärkeitä asioita on osoitettu keltaisella taustalla
  - opiskele ainakin ne
  - usein niitten ymmärtäminen edellyttää myös ympäröiviä asioita
- himmeänkeltainen tausta osoittaa keskitärkeitä asioita
  - pieni paino tentissä
  - arvosanoihin 3, 4 ja 5 tähtääville
  - arvosanaan 5 tähtäävän kannattaa opiskella myös korostamattomat asiat
- *johtopäätöksiä yms. tärkeää* on korostettu violeteilla vinoilla kirjaimilla
- **määriteltävä sana** tai **käsite** on vahvennettuna sinisellä
  - ei välttämättä tärkeä, katso taustan tai ympäröivän tekstin väri
- esimerkeissä käytetään usein ruskeaa
  - esimerkkejä ei kysytä tentissä, vaikka olisivat mustallakin
  - esimerkit ovat vain asioiden ymmärtämisen tueksi
- jokin asia on osoitettu **hyväksi** tai **huonoksi** vihreällä tai punaisella
- harmaalla kirjoitettua ei varmasti kysytä tentissä
  - harmaata käytetään, kun asia muuten näyttäisi tentissä kysyttävältä

Esitieto-oletukset

- yhden opintojakson verran ohjelmointia
  - **if** ... **then** ... **else** ... eli `if( ... ){ ... }else{ ... }`
  - **while** ... **do** ... eli `while( ... ){ ... }`
  - **for** $i := 1$ **to** n **do** $A[i] := 0$ vertaa `for( i=0; i<n; ++i ){ A[i] = 0; }`
- yhden opintojakson verran algoritmimatematiikkaa tai vastaavaa
  - $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, $\Rightarrow$, $\in$, $\cup$, $\cap$, $\{x \mid \ldots\}$, ...
- ei tarkasteta, että esitiedot ovat kunnossa
- harjoitusten tekemiseksi jokaisella täytyy olla mahdollisuus kirjoittaa ja ajaa C++-ohjelmia (mieluiten oma kone)

Suoritus

- opiskelijoiden määrä ei ollut tarpeeksi aikaisin tiedossa, joten laskuharjoitusten ja/tai esitelmien pakollisuus voidaan päättää vasta opintojakson alettua
- joka tapauksessa vaaditaan tentti
- joka tapauksessa jotain harjoituksia järjestetään pakollisena tai vapaaehtoisena

Muutama neuvo

- täsmällisen määrittelemisen taito on hyvin tärkeää!
  - opitaan harjoittelemalla, aloittamalla sopivan helpoista tehtävistä
  - esim. autojonon viimeisen mustan auton jälkeen tulee vain punaisia autoja
  - sin mus val mus pun pun toteuttaa ja sin mus val mus pun sin rikkoo
  - entä jos viimeinen auto on musta, tai entä jos jonossa ei ole mustia autoja?
- sama tieto voidaan esittää tietokoneessa monin eri tavoin
  - jotkin ovat ihmiselle luontevia
  - jotkin mahdollistavat tehokkaan käsittelyn
  - esim. viikkolukujärjestys + poikkeukset vs. oppimistapahtumien luettelo

  $\Rightarrow$ on hyvin tärkeää pohtia vaihtoehtoisia esitystapoja!
- päättelytaito on hyvin tärkeää!
  - sekin opitaan harjoittelemalla, aloittamalla sopivan helpoista tehtävistä
- älä keskity yksityiskohtien ulkoaopetteluun, vaan niiden taustalla olevien periaatteiden ymmärtämiseen!
  - esim. onko helppo muistaa tämä salakirjoitus? (videolta C. Brabrand ja J. Andersen: Teaching Teaching & Understanding Understanding, 2006)

1 2 3 4 5 6 7 8 9

- tehtäviä tehdään siksi, että opittaisiin *jatkossa tärkeitä* taitoja tai asioita!
  - tehtävän aihe voi olla ihan turha, mutta oppimistavoite ei ole
  - jos bluffaa alkuvaiheen tehtävissä, niin loppuvaiheen tehtävistä tulee ylivoimaisia
  - ”tentin (tai harjoituksen) jälkeen voin unohtaa” on huono asenne

1	2	3
4	5	6
7	8	9

1	2	3
4	5	6
7	8	9

0.1 Esimerkki

Epätyhjästä taulukosta `korkeus` on löydettävä mahdollisimman pitkän osuuden, jonka loppukohta on vähintään yhtä korkealla kuin alkukohta, pituus

- esim. pitkän kuntolenkin sisältä tilastointikelpoinen pituus

Helppo algoritmi C++:lla

- kokeillaan kaikki alku- ja loppukohdat, ja valitaan paras tulos

```
unsigned helppo( unsigned korkeus[], unsigned nn ){
  unsigned paras = 0;
  for( unsigned ii = 0; ii < nn; ++ii ){
    for( unsigned jj = ii; jj < nn; ++jj ){
      if( korkeus[ jj ] >= korkeus[ ii ] && jj - ii > paras ){
        paras = jj - ii;
      }
    }
  }
  return paras;
}
```

- helppo vakuuttautua, että toimii oikein
- hidas

Ajan kulutus sekunteina

nn	1 000	3 162	10 000	31 623	100 000	316 228	1 000 000
helppo	0	0	0	2	23	238	2 344
nopea	0	0	0	0	0	0	0

Nopea algoritmi C++:lla

```
unsigned nopea( unsigned korkeus[], unsigned nn ){
```

- etsitään "takahuiput"

```
unsigned th = nn-1, *seuraava_th = new unsigned[ nn ];
for( unsigned ii = nn; ii--; ){
  if( korkeus[ ii ] > korkeus[ th ] ){
    seuraava_th[ ii ] = th; th = ii;
  }
}
```

– kohdat, joiden jälkeen ei enää tule vähintään yhtä korkeita kohtia

⇒ *mahdollisimman pitkän osuuden loppukohta on takahuippu*

– `th` on ensimmäisen takahuipun kohta

– `seuraava_th[ ii ]` on seuraavan takahuipun kohta

– *seuraava takahuippu on edellistä alempana*

– *viimeisessä kohdassa on takahuippu*

- etsitään paras tulos

```
unsigned paras = 0;
for( unsigned ii = 0; ii < nn; ++ii ){
  while( korkeus[ th ] >= korkeus[ ii ] ){
    if( th - ii > paras ){ paras = th - ii; }
    if( th == nn-1 ){ delete[] seuraava_th; return paras; }
    th = seuraava_th[ th ];
  }
}
```

  – on helppo nähdä, että algoritmi kokeilee vain korkeusehdon toteuttavia osuuksia ja valitsee kokeilemistaan parhaan

- voiko (jokainen) kaikkein paras osuus $a \ldots \ell$ jäädä kokeilematta?
  – `ii` kokeilee jokaisen alkukohdan
  ⇒ `ii` kokeilee parhaan osuuden alkukohdan a
  – parhaan osuuden loppukohta ℓ on takahuippu
  – jos `ii` $< a$, niin `korkeus[`ℓ`]` $<$ `korkeus[ii]` (miksi?)
  ⇒ `th` ei ohita kohtaa ℓ kun `ii` $< a$
  ⇒ suorituksen aikana on hetki, jolloin `ii` $= a$ ja `th` $= \ell$
  ⇒ ohjelma tutkii jokaisen kaikkein parhaan osuuden

- tähän ei koskaan tulla, mutta C++-kääntäjäni suuttuu ilman sitä

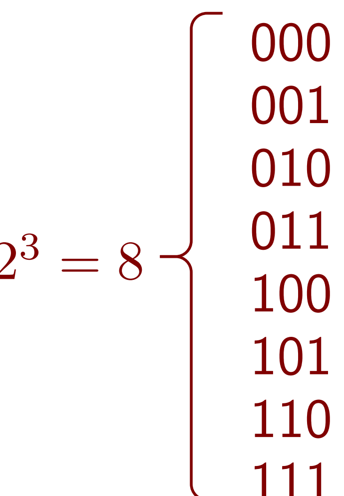
```
  delete[] seuraava_th; return paras;
}
```

1 Tiedon esittämistavoista

1.1 Alin taso

Tiedon esittämisen perusyksikkö on **bitti** (**bit**)

- valinta kahdesta vaihtoehdosta
- esim.
  - 0 tai 1
  - päälla tai pois päältä
  - alhaalla tai ylhäällä
- n bittiä kykenee esittämään valinnan 2^n vaihtoehdosta



Bitti on pieni yksikkö

⇒ usein käytetään yksikköä **tavu** (**byte**) kerrannaisineen

- tavu = 8 bittiä, kykenee esittämään valinnan 256 vaihtoehdosta
- esim.
  - luvut 0, 1, . . . , 255
  - luvut –128, –127, . . . , –1, 0, 1, . . . , 127
  - `A`, `B`, . . . , `Ö`, `a`, `b`, . . . , `ö`, `0`, `1`, . . . , `9`, `.`, `,`, `?`, `!`, `@`, . . .
  - koira, kissa, lehmä, hevonen, possu, kana, . . .

Tavun kerrannaisista vallitsee sekaannus

SI-järjestelmä	usein käytössä	IEC 60027
1 kB = 1 kilobyte = 1000 B	1 KB = 1 Kilobyte = 1024 B	= 1 KiB
1 MB = 1 megabyte = 10^6 B	1 MB = 1 Megabyte = 2^{20} B	= 1 MiB
1 GB = 1 gigabyte = 10^9 B	1 GB = 1 Gigabyte = 2^{30} B	= 1 GiB
1 TB = 1 terabyte = 10^{12} B	1 TB = 1 Terabyte = 2^{40} B	= 1 TiB

⇒ sekä lukiessa että kirjoittaessa on syytä olla tarkkana

- IEC 60027 olisi yksikäsitteinen, mutta se on huonosti tunnettu
- KiB, MiB, GiB ja TiB luetaan kibibyte, mebibyte, gibibyte ja tebibyte

Tietokone käsittelee tietoa pääsääntöisesti **sana** (**word**) kerrallaan

- merkitys vaihtelee
- nykyisin usein 64 bittiä tai 32 bittiä

⇒ riittävän suuri esittämään käyttökelpoisen lukualueen yms.
  - 32 bittiä: –2 147 483 648, . . . , 2 147 483 647

Olennaista on vain, että perusvaihtoehtoja on äärellinen määrä ja vähintään 2

- perusvaihtoehtojen joukkoa
  - kutsutaan usein **aakkosto**ksi (**alphabet**)
  - merkitään melko usein Σ (joten $2 \leq |\Sigma| < \infty$)
- isompi määrä vaihtoehtoja saadaan laittamalla aakkosia peräkkäin
  - n aakkosen jono esittää enintään $|\Sigma|^n$ vaihtoehtoa
- käytännössä yleensä $\Sigma = \{0, 1\}$ eli bitit tai $\Sigma =$ 8-bittiset tavut

Kokonaislukujen jakolasku kokonaislukuosamäärällä

- merkitsemme kokonaislukuosamäärää div ja jakojäännöstä mod
- $n \operatorname{div} m = \left\lfloor \frac{n}{m} \right\rfloor$
- $n \bmod m = n - (n \operatorname{div} m)m$
  - kirjoitettuna $n = (n \operatorname{div} m)m + (n \bmod m)$ tämä on **jakoyhtälö**
- esim. $13 \operatorname{div} 3 = 4$ ja $13 \bmod 3 = 1$
- C++:n / on muuten sama kuin div, mutta kun n ja m ovat erimerkkiset, ei taata, pyöristetäänkö alas vai kohti nollaa
  $\Rightarrow$ ei ole varmaa, että $n \,/\, m = \left\lfloor \frac{n}{m} \right\rfloor$
  - C++:n % noudattaa $n \,\%\, m = n - (n \,/\, m)m$

  $\Rightarrow$ `13/-3 == -5` ja `13%-3 == -2` tai `13/-3 == -4` ja `13%-3 == 1`
  `-13/3 == -5` ja `-13%3 == 2` tai `-13/3 == -4` ja `-13%3 == -1`

Etumerkittömät (unsigned) kokonaisluvut

- lukuarvot $0, 1, \ldots, 2^B - 1$, missä B on käytettävä bittien määrä
- bittijono $b_{B-1}b_{B-2}\cdots b_2b_1b_0$ edustaa lukuarvoa $\sum_{i=0}^{B-1} b_i 2^i$
- toisinpäin $b_i = (n \operatorname{div} 2^i) \bmod 2$
- aritmetiikka tapahtuu **modulo** 2^B
  - yhteen-, vähennys- ja kertolaskun tulos pakotetaan välille 0, . . . , $2^B - 1$ lisäämällä sopiva 2^B:n kerrannainen (jakolaskun tulos on automaattisesti oikealla välillä)
- esim. jos $|\Sigma| = 256$, niin

$$\begin{aligned} 200 + 100 &\rightsquigarrow & 300 - 256 &= 44 \\ 100 - 200 &\rightsquigarrow & -100 + 256 &= 156 \\ 10 \cdot 100 &\rightsquigarrow & 1000 - 3 \cdot 256 &= 232 \end{aligned}$$

- vaikutus on sama kuin laskemalla lasku oikein ja jättämällä liiat bitit pois
  - $1000 = 1111101000$, $232 = 11101000$

  $\Rightarrow$ riittää jättää liiat bitit laskematta
- $n >> i$ siirtää n:n bittikuviota i askelta oikealle
  - vasemmasta reunasta tulee nollia
  - esim. 00101101 >> 2 = 00001011
  - vaikutus on sama kuin $n \operatorname{div} 2^i$

- vastaavasti $n << i$ siirtää vasemmalle
  - oikeasta reunasta tulee nollia
  - esim. 10001011 << 2 = 00101100
  - vaikutus on sama kuin $(n2^i) \bmod 2^B$

Etumerkilliset (signed) kokonaisluvut

- tässä käsitellään vain esitystapaa nimeltä **kahden komplementti**
- lukuarvot $-2^{B-1}, \ldots, -1, 0, 1, \ldots, 2^{B-1} - 1$, missä B on bittien määrä
- jos $b_{B-1} = 0$, on lukuarvo $\sum_{i=0}^{B-2} b_i 2^i$, muutoin se on $\left(\sum_{i=0}^{B-2} b_i 2^i\right) - 2^{B-1}$
  $\Rightarrow$ lukuarvo on

$$\left(\sum_{i=0}^{B-2} b_i 2^i\right) - b_{B-1} 2^{B-1}$$

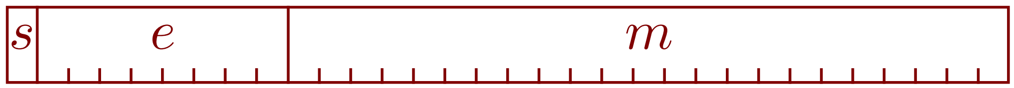
- luvun x vastaluku $-x$ saadaan kääntämällä bitit ja lisäämällä 1 modulo 2^B, paitsi luvulle -2^{B-1}

$13 =$ 01101	ja	$-13 =$	10010+1 = 10011
$-2 =$ 11110	ja	$2 =$	00001+1 = 00010
$0 =$ 00000	ja	$-0 =$	11111+1 = 00000
$-16 =$ 10000	$\rightsquigarrow$		01111+1 = 10000 $= -16$

- lukualueen ylityksen ja >>-operaattorin toiminta vaihtelee
  - ajoaikainen virhe vai tulos leikattuna käytettäviin bitteihin?
  - osallistuuko etumerkkibitti toimintoon?

Liukuluvut (**floating point numbers**)

- monia mahdollisuuksia, esim.
  $x = (-1)^s(2^{23}+m)2^{e-150}$



- 0 on esitettävä erikseen, esim. 000···0
  - saattaa olla erikseen $+0$ ja -0
- nollaa lähellä tarvitaan poikkeavasti esitettyjä lukuja
  - muuten voitaisiin esittää $2^{-127}+2^{-150}$ mutta ei $2^{-127}-2^{-150}$
- osa bittiyhdistelmistä voidaan varata erikoistarkoituksiin
  - kertomaan, että on tapahtunut virhe kuten nollalla jako
  - $\pm\infty$

Keino tutkia esitystapoja

```
#include <iostream>                // nn ja ff ovat samassa kohdassa muistia
int main(){
  union { unsigned nn; float ff; } uu;
  std::cin >> uu.ff;
  for( int ii = 32; ii--; ){
    if( ii % 8 == 7 ){ std::cout << ' '; }
    std::cout << (uu.nn >> ii) % 2;
  }
  std::cout << '\n';
}
```

Merkit (characters)

- pitkään tärkein standardi oli ASCII

```
  0, ..., 31   näkymättömiä ohjausmerkkejä
 32, ..., 47    !"# $%&' ()*+ ,-./
 48, ..., 63   0123 4567 89:; <=>?
 64, ..., 79   @ABC DEFG HIJK LMNO
 80, ..., 95   PQRS TUVW XYZ[ \]^_
 96, ..., 111  `abc defg hijk lmno
112, ..., 126  pqrs tuvw xyz{ |}~
127            näkymätön ohjausmerkki
```

  - 7 = äänimerkki, 10 = rivinsiirto, 13 = rivin alkuun
- 128, . . . , 255 otettiin eri tavoin käyttöön lisämerkeille
  - **ISO 8859-1** eli **Latin 1**: esim. 228 = ä
  - **Windows-1252** eli **CP-1252**: edellisen laajennos
  - **ISO 8859-15**: muutettu ISO 8859-1, esim. € mukana ja ½ puuttuu
- **Unicode** riittää laajalti ihmisten kirjoitusjärjestelmille
  - UTF-8-koodauksella: 1, . . . , 4 tavua: esim. 195 164 = ä

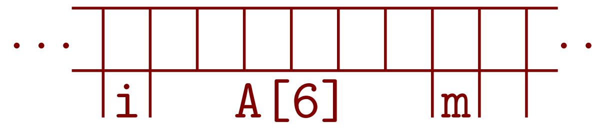
1.2 Taulukot, merkkijonot, tietueet ja osoittimet

Taulukko (array) sisältää nolla tai useampia rakenteeltaan samanlaisia tietoalkioita, joista valitaan haluttu **indeksoimalla**

H	e	i		k	a	i	k	k	i	!
1	2	3	4	5	6	7	8	9	10	11

- esim. $A[i] := 0$
- indeksointiin käytetään tyypillisesti kokonaislukua
  - ensimmäinen (eli ensimmäisen alkion) indeksi on C++:ssa 0
  - ⇒ jos C++:n taulukossa on n alkiota, niin indeksit ovat 0, 1, . . . , $n - 1$
  - myös 1 esiintyy melko usein ensimmäisenä indeksinä
  - esim. Pascal antaa ohjelmoijan valita ensimmäisen indeksin
  - ensimmäinen indeksi on C++:ssa helppo muuttaa halutuksi: jos haluttu ensimmäinen indeksi on e, niin korvataan kaikkialla $A[i] \rightsquigarrow A[i - e]$
- tietokone löytää nopeasti indeksiä vastaavan taulukon alkion

Kiinteän kokoiset taulukot

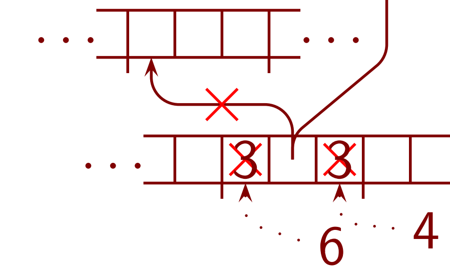


- tietokoneen muisti voidaan ajatella isoksi taulukoksi, josta annetaan paloja eri tarkoituksiin
- usein taulukon perästä on varattu muistia johonkin muuhun

⇒ taulukkoa ei yleensä voi kasvattaa alkuperäisellä paikallaan

⇒ ohjelmointikielissä oli alkuun vain kiinteän kokoisia taulukoita
  - koko määräytyi käännösaikana tai kun taulukko otettiin käyttöön

Joustavan kokoiset taulukot



- esim. C++:n `vector` voi kasvaa käytön aikana
- kun sille varattu tila loppuu, varataan muualta suurempi tila ja siirretään koko sisältö sinne
- esimerkkikuvassa nuoli edustaa **osoitinta (pointer)**
  - sisältää **osoitteen (address)** eli kertoo, missä kohdassa muistia jokin on
  - osoite voidaan ajatella luonnolliseksi luvuksi
  - osoittimen arvo voi olla myös $\perp$ eli ”tietoa ei ole”
    (muita nimiä nil, null ja 0, ei tarkoita osoitetta 0 vaan ”tietoa ei ole”)
- vanha tila voidaan ottaa myöhemmin muuhun käyttöön
  - uuden tilatarpeen on oltava tarpeeksi pieni mahtuakseen sinne
  - siellä täällä olevien vapaiden muistialueiden hallinnointi on jossain määrin työlästä, mutta käyttöjärjestelmät selviävät siitä ainakin tyydyttävästi
- jos uusi koko on 2 · (vanha koko), niin koska $\frac{1}{2} + (\frac{1}{2})^2 + \cdots + (\frac{1}{2})^n < 1$,
  - hukkamuistia on aina vähemmän kuin (käytettyä ja vapaata) hyötymuistia
  - alkio siirretään keskimäärin $<$ 2 kertaa

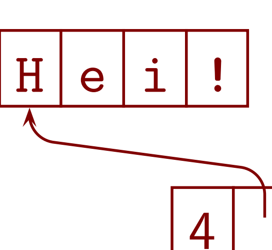
  $\Rightarrow$ siedettävän vähän hukkatyötä ja -muistia
- jos uusi koko olisi esim. vanha koko + 10, niin hukkamuistia ja -työtä tulisi paljon!

Osoittimet

- . . . tulivatkin jo esitellyiksi, tarvittaessa lue uudelleen edeltä!

Merkkijonot (**character strings**)

- C:n merkkijono on taulukko, jonka viimeisenä alkiona on 0
  - ei numeromerkki 0, vaan se näkymätön merkki, jonka koodi on 0
  - kokoa ei ole talletettu erikseen, vaan se tunnistetaan loppu-0:sta
  - sisällössä ei voi olla 0-merkkiä, mutta sehän ei olekaan näkyvä merkki
  - **ohjelmoijan on varattava tilaa myös loppu-0:lle!**
- C++:n `string` on taulukko (tai monimutkaisempi rakenne), jonka koko on talletettu erikseen



Tietue (**record**, **struct**) sisältää tietoalkioita, joista valitaan haluttu nimellä

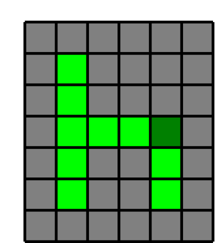
- ei tarvitse olla rakenteeltaan samanlaisia
- esim.

```
struct henkilo{ int synt_vuosi; string nimi; } hh;
std::cout << "ikä on " << nyt_vuosi - hh.synt_vuosi;
```

- C++:ssa `tietue.kentta` ja `osoitin_tietueeseen->kentta`
- **olio** ja **luokka** (**object** ja **class**) ovat tietueen käsitteen laajennos

Lisää esimerkkejä taulukoista

- CD-levylle tallennettu ääni on taulukko signaalitasoja
  - $2 \cdot 44\,100$ kpl 16-bittistä näytettä / s
- bittikarttakuva on 2-ulotteinen taulukko esim. lukukolmikoita RGB



1.3 Esimerkkejä rakenteellisista tekstuaalisista kielistä

HTML kuvaa www-sivun muodostumisen (loogisista) osista

- **HyperText Markup Language**

HTML 5 -tiedoston yleisrakenne

```
<!DOCTYPE html>
<html lang=fi>
<head>
<meta charset=UTF-8>
<style>
body { background: white; color: black; font-family: sans-serif }
     ...
div.I { width: 25%; float: right }
</style>
<title>MAT-71000 Tieto ja laskenta</title>
</head>
<body>
     ...
</body>
</html>
```

<body>-osan rakenne

```
<h2>Näin tehdään otsikko</h2>
<p>Tekstiä voi kirjoittaa enimmäkseen normaalisti.
Riveille jako ja   tyhjän   käyttö


    eivät merkitse paljoa.
<em>Näin korostetaan</em> tai <strong>näin</strong>.
Pienempi kuin -merkki tehdään esim. &lt; ja et-merkki &amp;.
<p><a href="http://validator.w3.org/unicorn/">Tällä
sivulla</a> voi tarkastaa, onko www-sivu ehjä.
<p class=loota>Katso esimerkki myöhemmin
```

- seuraavat merkit eivät tekstissä edusta itseään: <, >, & ja "
  - siksi täytyy tehdä muilla keinoin

CSS sisältää komentoja tekstin muotoilemiseksi

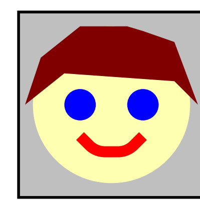
- **Cascading Style Sheets**

```
h2 { color: green; background: white; font: bold
  small-caps 20px normal }
p.loota { border: solid thick; color: maroon;
  background: yellow; text-align: right }
```

- vanhaa perua myös HTML sisältää muotoilukomentoja kuten <b>

Vektorigrafiikkaa voi piirtää esim. LaTeXin `pstricks`-pakkauksella

```
\begin{pspicture}(10,10)
\newrgbcolor{iho}{1 1 0.7}\newrgbcolor{ruskea}{.5 0 0}
\psframe[fillstyle=solid,fillcolor=lightgray](-1,-1)(11,11)
\psset{linecolor=blue}
\pscircle*[linecolor=iho](5,5){5}
\pscircle*(3,5){1}\pscircle *(7,5){1}
\psline [linecolor=red, linearc=4pt,linewidth=2pt
]  ( 3 , 3 ) (4,2)(6,2)(7,3)
\pspolygon*[linecolor=ruskea](-.5,5)(.5,8)(3,
10)(6,10)(7,9.7)(9,9)(10.5,5)(9,6.5)(2,7)
\end{pspicture}
```



Ohjelmointikielet sisältävät monenlaisia rakenteita

- muutamia käsitellään jäljempänä

1.4 Syntaksi

Tekstuaalisissa kielissä on yleensä kaksi rakenteellista tasoa

- **leksikaalinen** taso
  - **tekstialkiot (token)**: avainsanat, luvut, välimerkit, laskuoperaattorit yms. joissa ei saa olla sisällä tyhjää
  - mitä tyhjä tila on: välilyönnit, rivinsiirrot, sarkaimet, kommentit
- (varsinainen) **syntaksi**: miten tekstialkioita saa laittaa peräkkäin
  - vertaa `-(1+2)*3` ja `*1+)2(3-`

Luonnollisissakin kielissä on (kenties sumeita) leksikaalisia sääntöjä

- seuraavat on tehty arpomalla kirjaimia kolmen edellisen kirjaimen perusteella jakaumalla, joka on laskettu suomen tai englannin sanojen luettelosta
  - saleittantua kuus kea ta-ampuvuotoutiikka punsijäämällipoida poskisti
  - raimplativer tomon cochred tuffrancork anougglashesteng ing proad

Ilmauksien merkitystä kutsutaan **semantiikaksi**

- esim. täsä-o syn daxi päeMÄNTYä mutt, siilti shemandiickan t a j u u
- esim. kuusi on havupuu, joten seitsemän on havupuu plus yksi
- joskus on tärkeää tiedostaa ero ilmauksen ja sen merkityksen välillä
  - vrt. < ja & HTML:ssä; loppu-0, \" ja \\ C:n merkkijonoissa
  - mihin avaruus loppuu?

s-kirjaimeen

BNF eli **Backus-Naur form** on laajalti käytetty tapa määritellä syntaksi

- esimerkki: lauseke

$$\begin{array}{lcl} \textit{Lauseke} & ::= & \textit{Termi} \mid \textit{Lauseke}\ "+"\ \textit{Termi} \mid \textit{Lauseke}\ "-"\ \textit{Termi} \\ \textit{Termi} & ::= & \textit{Tekijä} \mid \textit{Termi}\ "\cdot"\ \textit{Tekijä} \mid \textit{Termi}\ "/"\ \textit{Tekijä} \\ \textit{Tekijä} & ::= & \textit{Atomi} \mid "+"\ \textit{Atomi} \mid "-"\ \textit{Atomi} \\ \textit{Atomi} & ::= & \textit{Luku} \mid \textit{Muuttuja} \mid "("\ \textit{Lauseke} ")" \end{array}$$

- käsitteen tai apukäsitteen rakenne ilmaistaan säännöllä muotoa

$$\textit{Nimi} \ ::= \textit{vaihtoehto} \mid \textit{vaihtoehto} \mid \cdots \mid \textit{vaihtoehto}$$

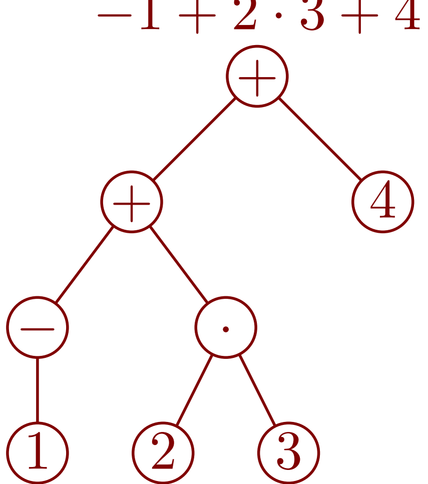
  missä vaihtoehto on ε tai jono käsitteiden nimiä ja/tai tekstialkioita
  - ε tarkoittaa, että ei laiteta mitään, esim. $\textit{Etumerkki} ::= \varepsilon \mid "+" \mid "-"$

  $\Rightarrow$ ε ei esitä itseään
  - samanmuotoisten toistoa ei BNF:ssä esitetä $\cdots$, vaan kuten *Lauseke* edellä

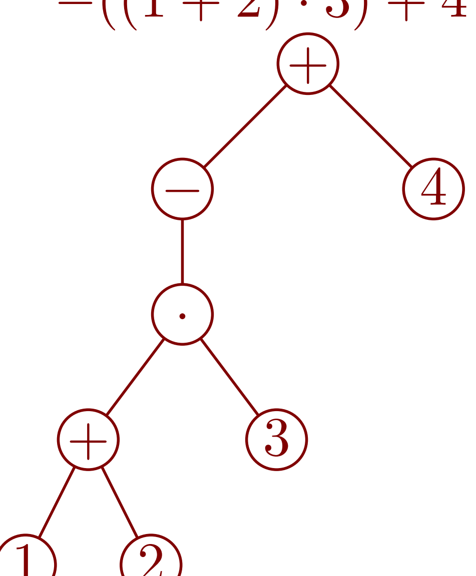
- BNF:stä on monenlaisia muunnelmia

Lauseke (expression)

- tietokonekielissä lauseke on tekstuaalinen keino ilmaista puumainen rakenne

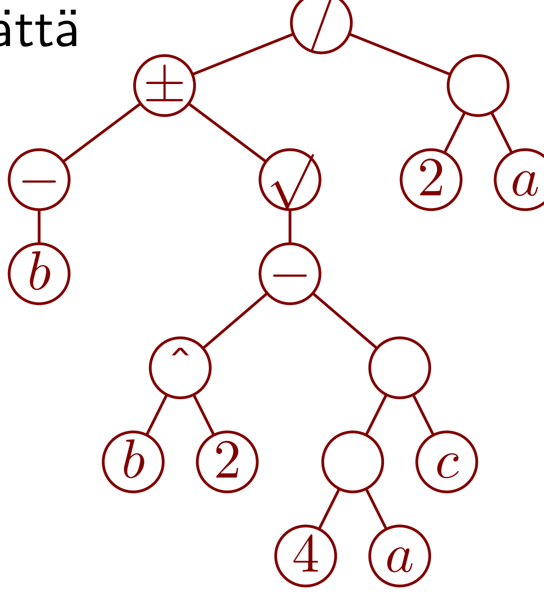


$(-1 + 2) \cdot (3 + 4)$



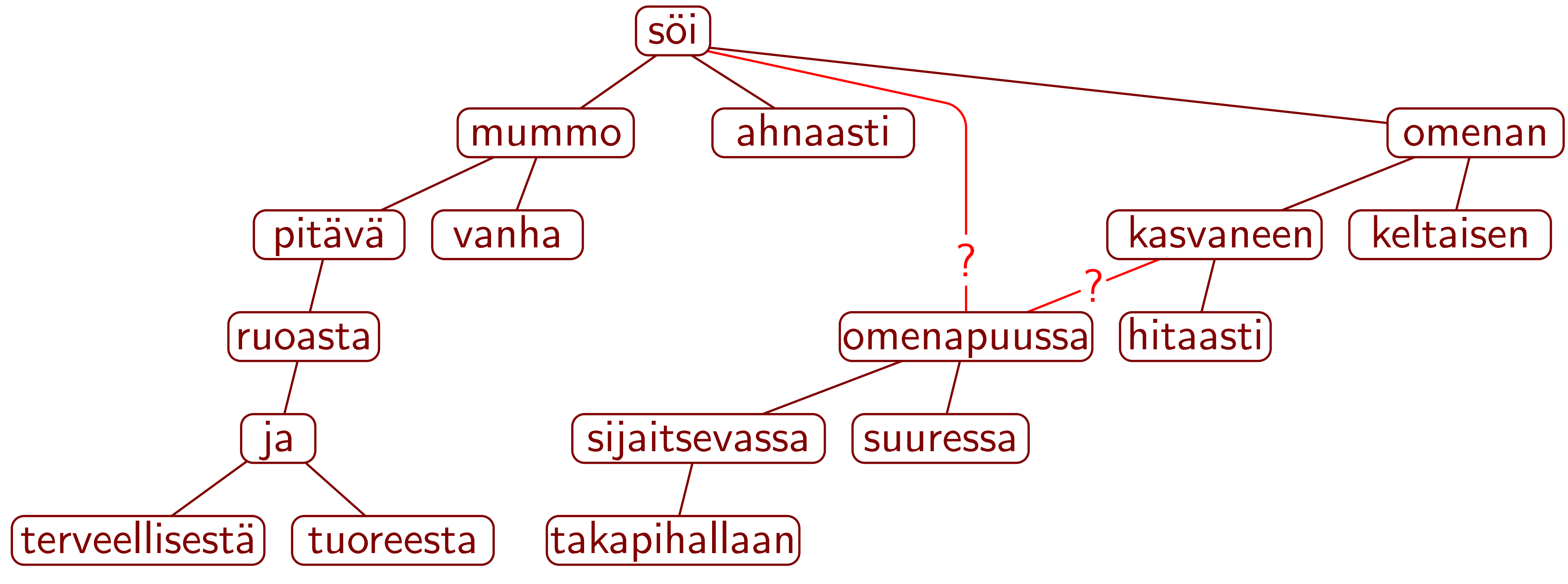
- tällainen puu on **lausekepuu**
- matematiikassa lauseke ei välttämättä ole lineaarista tekstiä

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



- luonnollisessakin kielessä on hierarkkisia rakenteita

  terveellisestä ja tuoreesta ruoasta pitävä vanha mummo
  söi ahnaasti takapihallaan sijaitsevassa suuressa omenapuussa
  hitaasti kasvaneen keltaisen omenan



- joskus ne eivät jäsenny tarkoitetusti

  Molemmat miehet vietiin putkaan. Siellä vuonna
  1954 syntynyt mies jatkoi riehumista.

1.5 Huomautuksia

0 Helsinki
108 Hämeenlinna
187 Tampere
347 Seinäjoki
680 Oulu
900 Rovaniemi

Sama tieto voi olla loogisesti organisoitu eri tavoin

- opiskelijoiden määrä, joista naisia $\leftrightarrow$ miesopiskelijoiden määrä ja naisopiskelijoiden määrä
- juna-aikataulun kilometrit pääteasemalta
  - osuuden Hämeenlinna–Oulu pituus saadaan yhdellä vähennyslaskulla
  - jos olisi annettu kunkin asemavälin pituus, tarvittaisiin monta yhteenlaskua

Tiedon sisäinen esittämistapa tietokoneessa voi olla kaukana siitä, mitä käyttäjä näkee

- esim. vektorigrafiikka
- (esim. HTML:n ja CSS:n yritys erottaa ulkoasu sisällöstä)

Ihmisillä on epäloogisia esitystapoja: 11:59 am, 12 noon, 12:01 pm, 12:59 pm, 1:00 pm

Tietokone tekee mitä käsketään, mutta se ei aina ole mitä halutaan

- paljolti kyse on ohjelmien ja ohjelmointikielten huonoista yksityiskohdista
  - esim. pieni kirjoitusvirhe muuttaa C++ loogisen ja && operaattoriksi &, joka käyttäytyy usein samoin mutta ei aina $\Rightarrow$ vaikeasti havaittava virhe
- samaa esiintyy luonnollisissakin kielissä
  - vauva juo virtsaa ja käy nukkumaan | vauva juo, virtsaa ja käy nukkumaan
  - panda eats, shoots and leaves vs. panda eats shoots and leaves

Toisinaan ohjelma on ok, mutta asia yllättää

- esim. kaari nuolessa kaaren säteen funktiona
- käskin tietokoneen etsiä Kotimaisten kielten keskuksen sanalistan sanoista pitkää sanajonoa, jonka voi jakaa kahdella tavalla yksittäisiksi sanoiksi

  šakki'ruutu'in'en.ää'ne'en.ää'ne'en.ää'ne'en.ää'ne'en.ää'ne'en.ää'ne'en.in

⇒ sama sana ei saa esiintyä useasti (huono pikakorjaus)

  šakki'ruutu'in'en.ää'ne'en.empi'ä.äri'nä.es'imu'oto.s'ääli.ö'isi'n.ähkää.s'ääntö

⇒ in ja ähkää pois sanastosta

  šinto'lain'en.ää'ne'en.empi'ä.äri'nä.es'imu'oto.s'ääli.ö'isi'n.äkki'ä.es'iva'ali

⇒ šinto ja lain pois sanastosta

  äänenpitävä'sti.di'aari'o.as'e.cu'p.ää'ne'en.ää'nes.te'ak.alli.nen'ä.äri'nä.ämmä

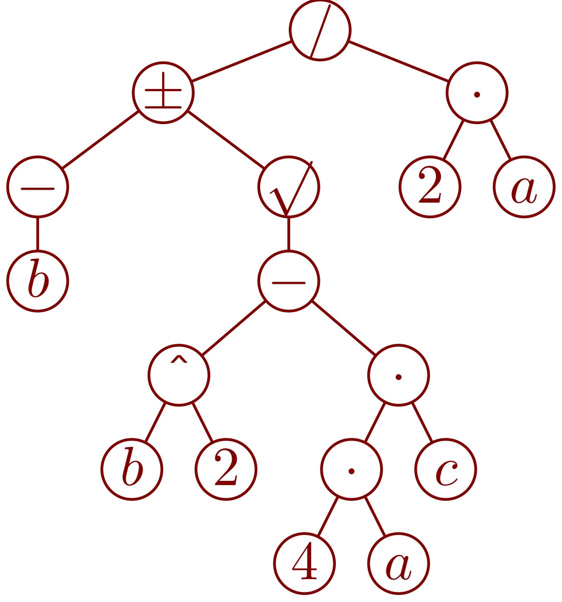
⇒ vierasperäisten, vanhentuneiden jne. sanojen poisto olisi liian iso työ

Helppokäyttöisten luotettavien ohjelmien teko edellyttää sekä ihmisen sumean että tietokoneen pilkuntarkan ajattelutavan ymmärtämistä

- jos virhe on tarpeeksi hölmö, niin
  - ihminen korjaa sen automaattisesti, jopa tiedostamattaan
  - tietokone tekee jotain todella hölmöä
- esim. ”kohta Suomessa on 100 000 alle 25-vuotiasta ilman tutkintoa”
- entä jos virhe on tarpeeksi hölmö kirjoittajan mutta ei lukijan näkökulmasta?

2 Lausekkeet, lauseet, ym.

2.1 Lausekkeet



Lauseke (expression)

- kuten aikaisemmin todettiin, lauseke esittää puumaisen rakenteen

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- ilmaisee esim.
  - miten arvo lasketaan
  - miten kokonaisuus muodostuu osistaan

Salaisuus, älä kerro kenellekään: tentissä testataan kykyä piirtää lausekepuu

- tentissä älä tee suluista solmua lausekepuuhun
  - sulut ovat apuväline puurakenteen ilmaisemisessa, eivät osa puurakennetta
  - lauseketta käsittelevissä algoritmeissa saat tehdä sellaisen

Piirrettyinä puut vievät paljon tilaa ja piirtäminen on työlästä

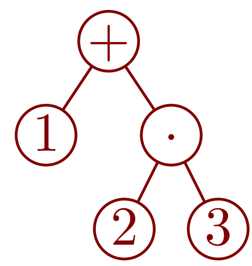
$\Rightarrow$ käytetään tiiviimpiä esitystapoja
- matemaattiset merkinnät
- ohjelmointikielten lausekkeet

- puumainen rakenne voidaan aina esittää siten, että lehdet esitetään sellaisinaan ja muut solmut muodossa (sisältö alipuu ... alipuu)
  - esim. $(/\ (\pm\ (-\ b)\ (\surd\ (-\ (\hat{}\ b\ 2)\ (\cdot\ (\cdot\ 4\ a)\ c))))\ (\cdot\ 2\ a))$
  - käytössä esim. LISP-kielessä
- matematiikassa on kuitenkin yleistä
  - sijoittaa solmun sisältö keskelle, jos solmulla on kaksi alipuuta
    $(((-\ b)\ \pm\ (\surd\ ((b\ \hat{}\ 2)\ -\ ((4\ \cdot\ a)\ \cdot\ c))))\ /\ (2\ \cdot\ a))$
  - jättää kertolaskut merkitsemättä
    $(((-\ b)\ \pm\ (\surd\ ((b\ \hat{}\ 2)\ -\ ((4\ a)\ c))))\ /\ (2\ a))$
  - osoittaa joidenkin alipuiden rajat muilla keinoilla kuin suluilla
    $$\frac{(-\ b)\ \pm\ \sqrt{(b)^{\,2}\ -\ ((4\ a)\ c)}}{2\ a}$$
  - käyttää välistystä jossain määrin puurakenteen mukaisesti
    $$\frac{(-b) \pm \sqrt{b^2 - ((4a)c)}}{2a}$$
  - sopia tulkinnasta, jos sulkuja on jätetty pois $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

$$\sum_{i=1}^{n} \frac{1}{\sqrt{i}}$$

- monissa ohjelmointikielissä tehdään samoin, paitsi että tekstin rivit rikkovia keinoja ei ole käytettävissä

Sitovuustasot (precedence)



- matematiikassa on käytäntö, että kertolasku sitoo voimakkaammin kuin yhteenlasku
  - $1 + 2 \cdot 3 \equiv 1 + (2 \cdot 3)$
- kielessä voi olla lukuisia operaattoreita ryhmiteltyinä sitovuustasoille
  - sitoo voimakkaammin = korkeampi sitovuustaso
  - jos kaksi operaattoria kilpailee välissään olevasta kohteesta, se liitetään voimakkaammin sitovaan: $1 + 2 \cdot 3$
- esim. C++ [Stroustrup 1997]: 68 operaattoria, 18 sitovuustasoa

`. [] ++ -- ...`
`++ -- ! - + ...`

`* / %`
`+ -`
`<< >>`
`< <= > >=`

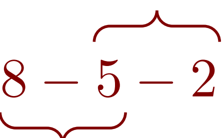
Unaari ja binääri

- **unaarioperaattorilla** on yksi argumentti
  - esim. etumerkki-−, logiikan ¬, etuliite-++, jälkiliite-++
- **binäärioperaattorilla** on kaksi argumenttia
  - esim. vähennyslasku-−, kertolasku ·
- **ternäärioperaattorilla** on kolme argumenttia
  - esim. C++ ?: `x < 0 ? -x : x`

Etumerkki-- on eri operaattori kuin vähennyslasku--, vaikka kirjoitusasu on sama

Sitovuuden suunta

- sitovuustasot eivät ratkaise tilannetta, jos kohteen molemmin puolin on sama operaattori
  - onko $8 - 5 - 2 \equiv 8 - (5 - 2) = 8 - 3 = 5$
    vai $8 - 5 - 2 \equiv (8 - 5) - 2 = 3 - 2 = 1$?



⇒ operaattorille ilmoitetaan sitovuuden suunta

- jos operaattori sitoo vasemmalle (on **left associative**), niin
  - kohde liittyy vasemmalla puolellaan olevaan
  - laskenta etenee vasemmalta oikealle
- jos operaattori sitoo oikealle (on **right associative**), niin päinvastoin
- yleensä operaattorit sitovat vasemmalle
- esimerkkejä oikealle sitovista
  - potenssilasku matematiikassa: $2^{1^3} \equiv 2^{(1^3)} = 2^1 = 2$ eikä $(2^1)^3 = 2^3 = 8$
  - sijoitusoperaattorit C++:ssa: `i = n = 0;`
- liitännäisyys (associativity) on eri asia
  - operaattori $\diamond$ on liitännäinen, jos aina $(x \diamond y) \diamond z = x \diamond (y \diamond z)$
  - liitännäisyys liittyy operaattorin merkitykseen eikä lausekepuun muodostamissääntöihin
  - jos operaattori on liitännäinen, niin sen sitovuuden suunnalla ei ole väliä

Matematiikan sitovuussäännöt eivät ole aina hyvin määritellyt

- esim. kaksinkertaisen kulman sini: $\sin 2x = 2 \sin x \cos x$
  - jos $\sin$ sitoo voimakkaammin, saadaan $(\sin 2)x = 2(\sin x) \cos x$
  - jos $\cdot$ sitoo voimakkaammin, saadaan $\sin(2x) = 2 \sin(x \cos x)$
  - tarkoitetaan $\sin(2x) = 2(\sin x) \cos x$

Tarpeettoman osan laskematta jättäminen (**short-circuit evaluation**)

- jos puhtaasti loogisen operaation ensin laskettu puoli saa sopivan arvon, ei toista puolta tarvitse laskeakaan
  - false $\wedge\, p =$ false ja true $\vee\, p =$ true

  $\Rightarrow$ työmäärä vähenee
- usein ohjelmoinnissa oikea puoli on määritelty vain, jos vasen saa sopivan arvon
  - **if** $i \geq 1 \wedge i \leq n \wedge A[i] = 0$ **then** $\ldots$

    vrt. **if** $i \geq 1 \wedge i \leq n$ **then if** $A[i] = 0$ **then** $\ldots$
  - **while** $p \neq \perp \wedge p{\uparrow}.n > 0$ **do** $\ldots$

$\Rightarrow$ muunnetut loogiset operaattorit, joiden oikea puoli lasketaan vain jos tarpeen, ovat käteviä

- **tällä opintojaksolla niitä merkitään `&&` ja `||`**
- C++:n `&&` ja `||` ovat sellaiset
- Ada:ssa on erikseen `and` ja `and then` sekä `or` ja `or else`

Toisinaan pidetään ihanteena, että lauseke esittää puhtaasti funktiota matemaattisessa mielessä

- ei saa olla **sivuvaikutuksia** (**side effect**)
  - sivuvaikutus = muu vaikutus kuin lausekkeen arvon tuottaminen
  - esim. `A[++i]` sivuvaikutuksenaan kasvattaa `i`:n arvoa
  - esim. `std::cout << i << ' ' << i*i` eli `((std::cout << i)<< ' ')<< i*i` palauttaa viitteen (luku 2.5) tulostuskanavaan ja sivuvaikutuksenaan tulostaa
- myös tarpeettoman osan laskematta jättäminen rikkoo ihannetta
  - **while** $p \neq \bot \wedge p\uparrow.n > 0$ **do** ... kuuluu tuottaa virhe kun $p = \bot$, koska **while** $p\uparrow.n > 0 \wedge p \neq \bot$ **do** ... tarkoittaa samaa koska $\wedge$ on logiikan "ja"

$\Rightarrow$ ihanteen tinkimätön noudattaminen olisi epäkäytännöllistä

Siltä osin kuin puurakenne ei määrää laskujärjestystä, sillä ei saa olla väliä

- esim. $1 + 2 \cdot 3$: puurakenne määrää, että ensin lasketaan $2 \cdot 3$
- $(1 + 2) \cdot (3 + 4)$: puurakenne ei määrää, lasketaanko ensin $1 + 2$ vai $3 + 4$
- `i*A[i++]` **on vaarallinen**, koska ei tiedetä, kumpi sininen tapahtuu ensin

Saman osan käyttö monesti samassa lausekkeessa on kömpelöä

$$\left(\frac{1}{x^2+1}\right) \sin\left(\frac{1}{x^2+1}\right) + \left(1 - \frac{1}{x^2+1}\right)^3$$

2.2 Tietotyypit

Nykyaikaisissa ohjelmointikielissä on paljon monenlaisia tietotyyppejä

- lukutyyppejä: erikokoiset kokonaisluvut etumerkillä ja ilman, erikokoiset liukuluvut, jopa kompleksiluvut
- muita pieniä tyyppejä: totuusarvot, merkit, luetellut tyypit {ma, ti, ke, to, pe, la, su}
- perustyypeistä yhdistämällä koottuja tyyppejä: taulukot, tietueet
- luokat
  - ohjelmoija voi päättää sisällön ja toiminnot hyvin vapaasti
- tiedon saannin tyyppejä: osoittimet, viitteet, `iterator` (luku 2.5)
  - eivät itsessään sisällä varsinaista tietoa, vaan tiedon, mistä tieto löytyy
  - esim. osoitin kokonaislukuun, osoitin henkilötietueeseen
- C++:n `const` muuttaa tyyppiä siten, että kohteeseen ei voi sijoittaa

Voiko sama arvo kuulua eri tyyppeihin?

- jokainen 16-bittinen kokonaisluku kelpaa sellaisenaan saman suuruiseksi 32-bittiseksi kokonaisluvuksi, mutta ei toisinpäin
- tietokoneen kannalta samasta tyypistä voi tehdä eri tyyppejä
  - tyypit `Nopeus` ja `Lämpötila` voivat kumpikin olla kokonaislukuja

- tietokoneen kannalta liukuluku-1 ja kokonaisluku-1 ovat eri asia, mutta käyttäjä voi mieltää toisin
- C++:ssa sekä merkit että totuusarvot ovat kokonaislukuja
  - `false` = 0, `true` = 1, muut luvut $\rightsquigarrow$ `true`
  - ei ole päätetty, vastaavatko merkit lukuja 0, . . . , 255 vai −128, . . . , 127

⇒ tiedon siirtyminen tyypistä toiseen pitää joskus estää ja joskus sallia
  - jos sallitaan, tiedon sisältö ja/tai esitystapa voi muuttua

Tyyppimuunnokset

- nykyaikaisissa ohjelmointikielissä on hyviä automaattisia tyyppimuunnoksia
- silti esim. `int n = 1; std::cout << (n/2)*2;` tulostaa 0
  mutta `int n = 1; std::cout << (n/2.)*2;` tulostaa 1
  ja `float n = 1; std::cout << (n/2)*2;` tulostaa 1
  ja `int n = 1; std::cout << n/2*2.;` tulostaa 0
- `std::cout << 1+ .0000000000000001-1 ;` tulosti 0
  `std::cout << 1+(.0000000000000001-1);` tulosti 1.11022e-16
- ota huomioon, että
  - lukualueet ovat rajallisia
  - `2` on eri asia kuin `2.0`
  - liukuluvuilla laskenta ei ole tarkkaa
  - tiedon tyyppi ja sisältö voivat muuttua lausekkeessa (muutkin kuin luvut)

- tyyppimuunnoksia voi tarvittaessa määrätä
  - esim. `int n = 1; std::cout << (float(n)/2)*2;` tulostaa 1
- yleensä tyyppimuunnokset pyrkivät säilyttämään tiedon merkityksen, mutta
  - on muunnoksia, joilla merkitystä muunnetaan hallitusti, esim. `int(-3.7)` tai `const`-määreen poisto
  - on muunnoksia, jotka säilyttävät esitystavan (bittikuvion) piittaamatta merkityksestä

2.3 Sijoituslause

Sijoituslause (assignment statement) on muotoa

muuttuja := lauseke (muuttuja **saa arvokseen** lausekkeen arvon)

tai

muuttujan osa := lauseke

- esimerkkejä
  - $i := i + 1$
  - $A[i] := 0$
  - $x := \dfrac{-K[2] + \sqrt{K[2]^2 - 4K[1]K[3]}}{2K[1]}$
  - henkilöt$[2 \cdot k - 3]$.sukunimi$[7]$:= 'k'
- vasen puoli on lauseke, joka tuottaa jotain, *johon* voi sijoittaa
  - muistipaikan nimen
  - **vasen-arvo (l-value)**
- oikea puoli on lauseke, joka tuottaa jotain, *jonka* voi sijoittaa
  - **oikea-arvo (r-value)**

⇒ $A[1]$ esiintyy kahdessa eri merkityksessä sijoituksessa $A[1] := A[1]$

- voi sijoittaa $n := 0$ mutta ei voi sijoittaa $0 := n$

Sijoitusoperaattorin kirjoitusasu

- muinoin käsinkirjoituksessa käytettiin $\leftarrow$, mutta sitä ei saa näppäimistöltä
  - muinoin ei aina ollut edes merkkiä <

$\Rightarrow$ otettiin käyttöön :=
  - esim. Algol, Pascal ja Ada

- FORTRAN, C ja C++ käyttävät =

$\Rightarrow$ = on hyvin yleinen ja näyttää olevan yleistymässä myös pseudokoodissa

- jos sijoitus on =, niin mikä silloin on yhtäsuuruusvertailu?
  - C ja C++: ==
  - FORTRAN: `.EQ.`
- on hyvä, että epäsymmetrinen toiminto merkitään epäsymmetrisellä symbolilla
  - **if** $n = 0$ **then** ... toimii samoin kuin **if** $0 = n$ **then** ...
  - $n := 0$ ei toimi samoin kuin $0 := n$
- jos ":" tulkitaan "jälkeen" ja sen puute "ennen", niin $i = i: - 1$ olisi $i + 1 = i:$ eli $i: = i + 1$ eli $i := i+1$ (mutta kääntäjät eivät hyväksy kuin viimeisen)
- on vaikeaa saada ohjelmoijat hyväksymään :=, mutta vielä vaikeampaa olisi saada peruskoulujen matematiikan opetukseen ==
- on hyvä tiedostaa ero merkinnän ja sen esittämän abstraktin asian välillä
  - tätä auttaa, jos tuntee saman asian eri merkinnöillä

$\Rightarrow$ tällä opintojaksolla käytetään **:=**

:= ja :⇔ määrittelyoperaattoreina

- nykyisin teoreettisesessa tietojenkäsittelytieteessä ja matematiikassakin := käytetään toisinaan määrittelyoperaattorina
  - uusi merkintä := määritelmä
  - esim. kompleksiluvun itseisarvo $|x + iy| := \sqrt{x^2 + y^2}$
  - lauseke1 = lauseke2 tarkoittaa pääsääntöisesti yhtäsuuruusväitettä jo määriteltyjen asioiden välillä
- varsinkin aikaisemmin käytettiin esim. $=_{def}$ tai $\hat{=}$
  - myös käytetään = ja tehdään tekstissä selväksi, että merkintä on uusi
- paitsi että ♡ := antaa ♡:lle arvon tai merkityksen, se myös kertoo, että ♡ on uusi merkintä, joka otetaan nyt käyttöön
- matematiikassa ei yleensä ole tapana vaihtaa merkinnän arvoa tai merkitystä ⇒ sekaannusta myöhempiin sijoituksiin ei synny, koska niitä ei ole
- kun määrittelyn kohde tuottaa totuusarvon, saatetaan käyttää :⇔
  - esim. $on_alkuluku(n) :\Leftrightarrow n \geq 2 \wedge \neg \exists i : \exists j : i > 1 \wedge j > 1 \wedge n = ij$

C++:n sijoitustoiminnoista

- C++:n sijoitus on lauseke, joka palauttaa arvonaan sijoitetun arvon
  - sijoitustapahtuma on sivuvaikutus

⇒ C++:ssa sijoituksia voi ketjuttaa, esim. `x = i = j = 0;`

- C++:ssa on myös +=, %=, <<= jne.

2.4 Ohjauslauseet

Ehtolauseessa (**conditional statement**) **if** C **then** B_1 **else** B_2

- C on **ehto** (**condition**)
- B_1 on **then-haara** (**then-branch**)
- B_2 on **else-haara** (**else-branch**)
  - avainsanan **else** ja **else**-haaran saa jättää pois

While-silmukassa (**while-loop**) **while** C **do** B

- C on **ehto** (**condition**)
- B on **vartalo** (**body**)

Tämän opintojakson pseudokoodissa sisennykset osoittavat rakenteiden rajat

Esimerkki: Quicksort

- kenties maailman eniten käytetty nopea järjestämisalgoritmi
- toimintaperiaate
  - siirrä pienet alkiot taulukon alkuosaan ja suuret loppuosaan (ositus)
  - järjestä alku- ja loppuosa kumpikin erikseen Quicksortilla
- ositus on vaikea tehdä hyvin
  $\Rightarrow$ osituksesta on monia eri muunnelmia

- esimerkkitoteutus

Quicksort(a_1, y_1)

if $a_1 \geq y_1$ **then return**
$x := A[\,\text{random}(a_1, y_1)\,].x$
$a := a_1 - 1$; $y := y_1 + 1$
while $a < y$ **do**
  $a := a + 1$; $y := y - 1$
  while $A[a].x < x$ **do** $a := a + 1$
  while $A[y].x > x$ **do** $y := y - 1$
  if $a < y$ **then** $apu := A[a]$; $A[a] := A[y]$; $A[y] := apu$
if $a = a_1$ **then** $a := a + 1$
Quicksort$(a_1, a - 1)$; Quicksort(a, y_1)

- järjestää taulukon $A[a_1 \ldots y_1]$ alkioiden avainten $A[i].x$ perusteella
  - koko taulukon $A[1 \ldots n]$ järjestämiseksi kutsutaan Quicksort$(1, n)$
- tulemme näkemään, että on tärkeää, että ositus ei ole toistuvasti hyvin vino
- tutkimme tätä pseudokoodia muunnelmineen huolellisesti luvussa 4.1

For-silmukassa (**for-loop**) **for** $i := a$ **to** y **do** B

- i on **silmukkamuuttuja**
- a on alaraja
- y on yläraja

Aidossa **for**-silmukassa

- ylärajan arvo lasketaan ensimmäisen kierroksen alkaessa eikä se muutu, vaikka sen laskemisessa käytettävät tiedot muuttuisivat
  - $n := 4$; **for** $i := 1$ **to** n **do** $n := n + 1$ lopettaa kierroksen $i = 4$ jälkeen
- silmukkamuuttujan arvoa ei saa muuttaa silmukan vartalossa (ainakaan pienemmäksi)
  - $n := 4$; **for** $i := 1$ **to** n **do** $i := 2$ on kielletty

⇒ silmukka tekee enintään $y - a + 1$ kierrosta, a:n ja y:n aloitushetken arvoilla

- C++:n `for(...){...}` ei ole tässä mielessä aito **for**-silmukka

Avainsanalla **downto** ilmaistaan, että silmukkamuuttuja juoksee alaspäin

for $i := y$ **downto** a **do** B

Esimerkki: Counting-sort

- nopein tunnettu järjestämisalgoritmi silloin, kun avaimet $A[i].x$ saavat arvonsa *pienestä* joukosta $\{0, 1, \ldots, M - 1\}$
  - tarvitsee aputaulukon kooltaan M
  - osa työmäärästä verrannollinen lukuun M

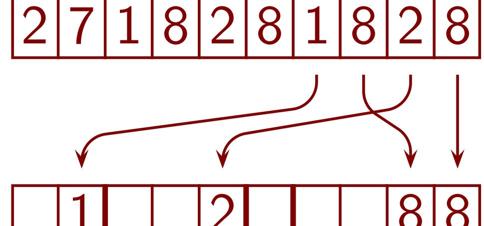
  ⇒ ei käyttökelpoinen, jos M on suuri

- toimintaperiaate
  - laske kunkin arvon esiintymien määrä
  - kopioi arvot vastaustaulukkoon suoraan oikeille paikoilleen

Counting-sort

for $k := 0$ **to** $M-1$ **do** $C[k] := 0$
for $i := 1$ **to** n **do** $C[A[i].x] := C[A[i].x] + 1$
for $k := 1$ **to** $M-1$ **do** $C[k] := C[k] + C[k-1]$
for $i := n$ **downto** 1 **do**
    $B[C[A[i].x]] := A[i]$; $C[A[i].x] := C[A[i].x] - 1$

- toisen rivin jälkeen $C[k]$ on niiden alkioiden määrä, joiden arvo $= k$
- kolmannen rivin jälkeen $C[k]$ on niiden alkioiden määrä, joiden arvo $\leq k$
- viimeinen silmukka kopioi kunkin A:n alkion viimeiseen sellaiseen B:n kohtaan, johon ei ole vielä kopioitu ja jossa kuuluu olla enintään sen suuruinen alkio



- koska se selaa A:n takaperin, se säilyttää niiden alkioiden keskinäisen järjestyksen, joilla on sama $.x$
  - lopputulos saadaan noudattamaan ensisijaisesti jotain (esim. sukunimi) ja toissijaisesti jotain muuta (esim. etunimi) järjestystä

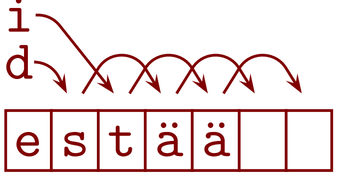
Muita ohjauslauseita

- tällä opintojaksolla **error** kaataa ohjelman
  - käytetään merkitsemään tilanteita, joista ohjelman ei tarvitse selvitä, kuten tyhjän tietorakenteen ensimmäisen alkion palauttaminen
- **goto**: hyppy mielivaltaiseen kohtaan koodia
  - oli muinoin perusrakenne yhdessä vastaavan ehdollisen hypyn kanssa
  - aiheuttaa vaikeatajuista koodia, vaikea nähdä mistä johonkin tullaan

  ⇒ joutui epäsuosioon
- silmukat, joiden ehto testataan lopussa (tai keskellä)
  - `do ... while(...);`
  - **repeat** ... **until** ...
  - **loop** ... **exit when** ...; ... **end loop**
- `continue`: hyppy silmukan ehtoon
- `break`: hyppy silmukasta ulos
- haarautuminen arvon perusteella: `switch`, **case**, computed goto
- poikkeusmekanismi

2.5 Osoittimet, viitteet ja kahvat

Taulukot ovat melko jäykkä tietorakenne



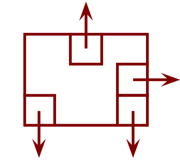
- lisääminen muualle kuin loppuun on hidasta
  - esim. `estää` $\leadsto$ `edistää`
- tilaa pitää varata kiinteä määrä
  - joustavan kokoinen taulukko voi kasvaa yli määrän siirtymällä toiseen paikkaan, mutta se perustuukin osoittimiin (ja jättää reiän muistiin)

Joustavia tietorakenteita saadaan tietueilla, joissa on osoittimia toisiinsa

- tällä opintojaksolla
  - $os{\uparrow}$ tarkoittaa osoittimen os päässä olevaa tietuetta tms. (C++:n `*os`)
  - osoittimen arvo $\perp$ tarkoittaa, että osoittimen päässä ei ole mitään
  - $os{\uparrow}.kenttä$ tarkoittaa osoittimen os päässä olevan tietueen kenttää (C++:n `os->kenttä`)
- esim. 1-suuntainen linkitetty lista
  - lisääminen keskelle on nopeaa (kun paikka tiedetään)

$$uusi{\uparrow}.kirjain := \texttt{'i'};\ uusi{\uparrow}.seur := nyt{\uparrow}.seur;\ nyt{\uparrow}.seur := uusi$$

- tietueet saadaan yleensä varaamalla dynaamisesta muistista (luku 3.1)
- jatkossa tulee paljon lisää esimerkkejä



- sama tietue voi olla yhtäaikaa usean osoitinrakenteen jäsen
  - esim. aakkosjärjestetyssä puussa ja aikajärjestetyssä listassa

    $.arvo$, $.vasen$, $.oikea$, $.isä$, $.seur$

  ⇒ vanhimman poisto ja aakkosjärjestyksessä hakeminen on tehokasta
- tällaisen toteuttaminen edellyttää kykyä toteuttaa osoitinrakenteita

  ⇒ älä tyydy vain poimimaan osoitinrakenteita kirjastoista

Tietueita ja osoittimia voi matkia taulukoilla ja indekseillä

- esim. $kirjain[uusi]$:= 'i'; $seur[uusi] := seur[nyt]$; $seur[nyt] := uusi$
- vaikka ohjelmointikielen osoittimien ja tietueiden käyttö on yleensä kätevämpää, toisinaan kannattaa tehdä kuten ym. esimerkissä
- osoittimet ovat itse asiassa indeksejä isoon taulukkoon nimeltä $Muisti$
  - esim. $os{\uparrow}.kirjain \sim Muisti[os]$ ja $os{\uparrow}.seur \sim Muisti[os+4 \ldots os+7]$
  - $Muisti[os+1 \ldots os+3]$ ovat esimerkissä käyttämättä, koska nykyisin pyritään sijoittamaan tietoja 4:llä jaollisiin osoitteisiin
- C++:ssa taulukot ovat vain lyhennemerkintä osoittimille
  - `A[i]` tekee saman kuin `*(A+i)` tai `*(i+A)`

Kahva (**handle**) on yleinen nimi suoralle pääsylle johonkin tietoon

- yleensä käytännössä osoitin tai indeksi
- C++:n *iterator* on jokseenkin sama asia
  - ”iterator” sisältää ajatuksen kyvystä siirtyä seuraavaan tai edelliseen, mitä ”kahva” ei välttämättä edellytä

Viite (**reference**) on osoitin suoran tiedon syntaksilla

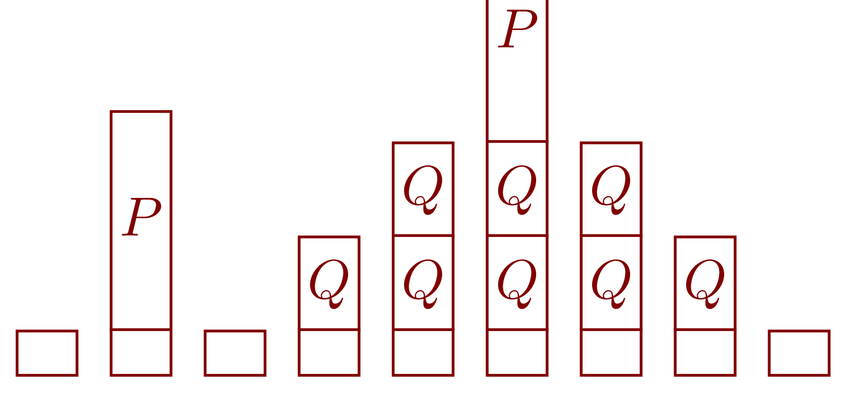
- esim. `int i=4, &j=i;` varaa yhden muistipaikan, jolla on kaksi nimeä `i` ja `j`
  - `int i=4, &j=i; j=2; std::cout << i;` tulostaa 2
- kätevä esim. antamaan lyhyt tilapäinen nimi, johon voi sijoittaa
  - ```
    int &i = H[k]->seur->henkilö.tulospisteet[2];
    if( i>100 ){ i=100; } maksa_bonusta( i*5 );
    ```
- hyödyllinen tiedon välittämisessä aliohjelmalle ja takaisin (luku 3.2)
- viitettä ei voi kääntää osoittamaan muualle
  - sijoittaminen viitemuuttujaan sijoittaa kohdemuistipaikkaan, ei osoittimeen
- C++:ssa voi luoda viitetyyppejä
  - viitetyyppinen tieto on viite alkuperäisen tyyppiseen tietoon
  - kätevää
  - vaarallista, jollei tiedä mitä tekee
    ```

3 Aliohjelmat ja abstraktiot

3.1 Muistin hallinta

Nykyisin tietokoneohjelma saa muistia pääasiassa kolmella tavalla

1. Muistin varaus **staattisesti** eli suorituksen aloitushetkellä
   - "samaa" muuttujaa voi olla vain yksi kappale (vrt. jäljempänä)
   - muuttujan elinikä on tyypillisesti koko suoritusaika
   - C++: aliohjelmien ulkopuolella määritellyt, `static`
2. Muistin varaus **kutsupinosta** (**call stack**)
   - iso muistialue, jota otetaan käyttöön ja vapautetaan laidasta alkaen
   - kun kutsutaan aliohjelmaa, sen muisti varataan kutsupinon päälle
   - myös sisäkkäisten lausekkeiden laskenta käyttää kutsupinoa
   - "samaa" muuttujaa voi olla monta kappaletta, muuttujan elinikä vaihtelee
     - Q:n paikalliset muuttujat esimerkissä
   - esim. pääohjelma kutsuu P:tä, sitten pääohjelma kutsuu Q:ta joka kutsuu Q:ta joka kutsuu P:tä



- varaaminen ja vapauttaminen on helppoa ja nopeaa määrästä riippumatta
- vapauttamisen on tapahduttava tiukasti määrätyssä järjestyksessä

3. Muistin varaus **dynaamisesta muistista** (**heap**, **free store**)

- iso muistialue, josta muistia otetaan käyttöön ja vapautetaan mielivaltaisesti
  - tämä "heap" on eri asia kuin heapsort:in "heap"
- C++: `new` ja `delete`
  - esim. C++:n merkkijonotoimintojen sisällä on piilossa ≈ `new` ja `delete`
- varatuilla muistilohkoilla on yleensä tyyppi, mutta ei yleensä yksilöllisiä nimiä
  ⇒ ei yleensä kutsuta muuttujiksi
- usein varataan monia lohkoja, ja niissä on osoittimia toisiinsa
- "muuttujan" elinikä on ohjelmoijan vapaasti päätettävissä
  - ts. muistilohkon varaus- ja vapautushetki
- dynaamisen muistin käyttö on altis virheille
  - otetaan osoitin käyttöön mutta ei varata muistia `int *p; *p = 3;`
  - vapautettaessa muistia jää osoittimia, joiden kautta käyttö jatkuu (**dangling pointer**) `int *p = new int, *q = p; delete p; *q = 3;`
  - muistin vapautus unohtuu: **muistivuoto** (**memory leak**)
    `int *p = new int, *q = new int; *p = 3; p = q;`

- kun muistia vapautetaan sekalaisessa järjestyksessä, jää varatun muistin sekaan vapaita alueita
- niiden uudelleen käyttöön otto eri kokoisilla varauksilla pirstoo vapaata muistia vielä pienemmiksi alueiksi
- jollei vierekkäisiä vapaita alueita yhdistetä, ei lopulta pystytä antamaan isoja muistilohkoja, vaikka vapaata muistia olisi paljon

⇒ käyttöjärjestelmissä on monimutkaisia mekanismeja muistin kierrättämiseksi
  - koska asia on tärkeä, ne on hiottu hyviksi
  - silti äärimmäiseen tehoon pyrittäessä kannattaa muistin vapautusta välttää
  - saman tyyppisille muistilohkoille on helppo järjestää tehokas kierrätys itse

Kutsupino tarjoaa hyvän kompromissin

- staattinen varaaminen on moniin tarkoituksiin liian joustamatonta
- dynaaminen varaaminen on vaikeaa ohjelmoijalle ja käyttöjärjestelmälle

⇒ kutsupino on tärkeä käsite

- luvussa 11 tulemme näkemään, että pinolla on erikoisasema myös teoreettisessa tietojenkäsittelytieteessä

Muistin tasot ja virtuaalimuisti

- pienitehoisessa tietokoneessa (kuten TV:n kaukosäädin) voidaan esim. laittaa
  - muistin alkuun käyttöjärjestelmä, sitten ohjelma, sitten kutsupino
  - dynaamisen muistin alue muistin loppuun
- järeän tietokoneen muisti koostuu eritasoisista osista
  - paljon hidasta muistia (kiintolevy)
  - melko paljon keskusmuistia
  - jokin määrä erityisen nopeaa **välimuistia** (**cache**)
  - prosessorin rekisterit, joissa laskenta tapahtuu
- ohjelmalle annetaan yleensä yhtenäinen **osoiteavaruus** (**address space**)
  - sama ohjelman tuntema osoite voi eri hetkinä vastata eri sijainteja keskusmuistissa tai kiintolevyllä: **virtuaalimuisti**
  - ahkerimmassa käytössä oleva on kopioitu välimuistiin
  - pitkän aikaa käyttämätön siirretään kiintolevylle
- tietoa siirretään kiintolevylle ja takaisin vakiokokoisina, melko isoina lohkoina
- nämä kannattaa ottaa huomioon tehtäessä suuria tietomääriä käsitteleviä ohjelmia

3.2 Aliohjelmat

$\text{Quicksort}(a_1, y_1)$

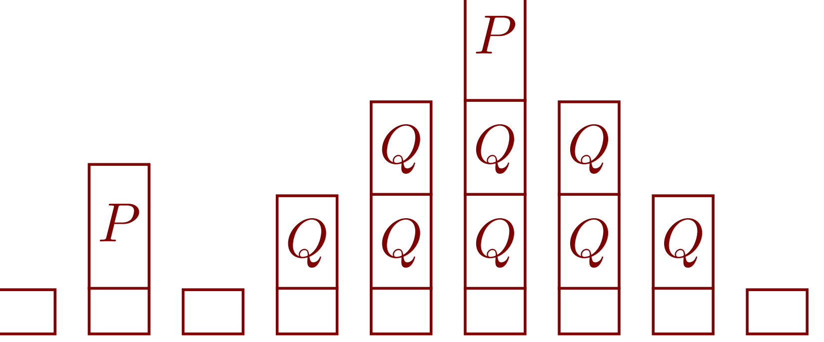
if $a_1 \geq y_1$ **then return**
$x := A[\,\text{random}(a_1, y_1)\,].x$
$a := a_1 - 1;\ y := y_1 + 1$
...
$\text{Quicksort}(a_1, a - 1);$
$\text{Quicksort}(a, y_1)$

Aliohjelma (subroutine)

- keino eristää koodinpätkä itsenäiseksi yksiköksi
  - kirjoitetaan kerran, mutta voidaan **kutsua** (**call**, **invoke**) useasti
  - usein selkeyttää ohjelmaa, vaikka kutsuttaisiin vain kerran
- tieto voi kulkea kutsujan ja aliohjelman välillä kolmella tavalla
  - kutsuja antaa tietoa aliohjelmalle **parametrien** välityksellä a_1, y_1
  - aliohjelma palauttaa tietoa parametrien tai nimensä välityksellä
  - aliohjelma käsittelee sille näkyvää ulkopuolista tietoa suoraan A
- nimensä välityksellä tietoa palauttavaa aliohjelmaa kutsutaan usein **funktioksi**
  - voidaan käyttää lausekkeessa
  - esim. $y := (1 + \sin(\omega \cdot t + \varphi))/2$
  - usein suositellaan, että funktio ei saa vaikuttaa muulla tavalla (ei saa olla sivuvaikutuksia)
  - ohjelmointikielen funktio ei välttämättä ole matemaattisesti funktio
- aliohjelman **paikallinen** (**local**) muuttuja on sen sisällä määritelty muuttuja
- tällä opintojaksolla (ja C++:ssa) **return**, **return** *paluuarvo* tai ulos tulo aliohjelman lopusta palauttaa aliohjelmasta

Aktivaatiotietue (activation record)

- aliohjelma tarvitsee muistia
  - parametreille — Quicksort: a_1 ja y_1
  - paikallisille muuttujille — Quicksort: x, a, y ja apu
  - paluuosoitteelle (kutsujan suorituksen sijainnille) — Quicksort: alkuper., alarivi 1 ja 2
  - funktion tapauksessa paluuarvolle
- nämä muodostavat aktivaatiotietueen
  - lisätään kutsupinon päälle aliohjelman suorituksen alkaessa
  - poistetaan kutsupinon päältä aliohjelman suorituksen loppuessa



- muuttujia voidaan luoda kutsupinoon ja vapauttaa sieltä myös kesken aliohjelman
  - lausekkeet sekä esim. `for( int i=0; i<n; ++i ){ ... }`

Parametrien välitys (parameter passing), johdanto

- esim. määritelmä Quicksort(a_1, y_1) $\ldots a_1 \ldots y_1 \ldots$, kutsu Quicksort$(a_1, a-1)$
  - alleviivatut a_1 ja y_1 ovat **muodollisia parametreja** (**formal parameter**)
  - $a-1$ ja viimeinen a_1 ovat **kutsuparametreja** eli **todellisia parametreja** (**actual parameter**)
- kukin parametri välitetään jollain parametrinvälitysmekanismilla
  - käymme läpi kaksi tunnetuinta

Arvon välityksessä (call by value) kutsuparametri kopioidaan aliohjelmalle

- C++:n ”tavallinen” parametrinvälitysmekanismi
- välittää tietoa vain kutsujalta aliohjelmalle
- kutsuparametri voi olla lauseke, esim. $\omega \cdot t + \varphi$
  - kutsuparametri voi olla oikea-arvo
  - aliohjelmalle annetaan sen arvo laskettuna kutsuhetkellä
- tehoton, jos välitettävän tiedon määrä on suuri mutta aliohjelma käyttää siitä vain osaa
  - esim. jos aliohjelma etsii puhelinnumeron puhelinluettelosta nimen perusteella, niin kopioi koko puhelinluettelon
- ei altis jäljempänä käsiteltävälle aliasing-ongelmalle
- muunnelmissa tietoa kopioidaan myös tai vain lopuksi aliohjelmalta kutsujalle
  - **call by result**, **call by value-result**
  - kutsuparametrin on oltava vasen-arvo
  - kopioidaanko vastaus kutsuparametriin tulkittuna kutsuhetkellä vai paluuhetkellä?

$$\frac{\text{hups}(x)}{i := 2;\ x := 5}$$

kutsu: $i := 1$; $\text{hups}(A[i])$

Viitteen välityksessä (call by reference) aliohjelmalle annetaan kutsuparametrin osoite tms., mutta ei käytetä osoitinten kirjotustapaa

- C++:n &-mekanismi
  - oikeastaan C++:ssa välitetään arvon välityksellä viitetyyppinen arvo, mutta se on sama asia toisin sanottuna
- välittää tietoa molempiin suuntiin
  - C++:ssa `const`-määreellä voidaan (yrittää) estää välitys paluusuuntaan

$\Rightarrow$ kutsuparametrin on oltava vasen-arvo

- kaikki ohjelmointikielet eivät varmistaneet tätä

$\Rightarrow$ oli mahdollista muuttaa vakioita, esim. ups(x) $x := 3$, kutsu: ups(π)

- lievästi tehoton, jos välitettävän tiedon määrä on pieni
  - aliohjelma pääsee tietoon käsiksi vain epäsuorasti, ei $i := i + 1$ vaan $Muisti[viite] := Muisti[viite] + 1$
- käyttää aina kutsuparametrin
  - vasen-arvoa tulkittuna kutsuhetkellä
  - oikea-arvoa tulkittuna käyttöhetkellä
- esim. aliohjelman hips(x, n) $i := 2;\ x := n$ kutsu $i := 1;$ hips$(A[i], i)$ sijoittaa $A[1] := 2$

$\Rightarrow$ altis aliasing-ongelmalle

Aliasing-ongelma

- samalla muistipaikalla voi olla monta eri nimeä
  - monta olennaisesti erinäköistä lauseketta voi tarkoittaa samaa vasen-arvoa
- esim. viitteen välityksellä aliohjelman $\underline{\text{hops}(i,j)}$ $i := 1;\ j := 2;\ \text{print}(i)$ kutsu $\text{hops}(n,n)$ tulostaa 2, mutta arvon välityksellä 1
- esim. viitteen välityksellä aliohjelman $\underline{\text{heps}(i)}$ $i := 2;\ \text{print}(n)$ kutsu $n := 1;\ \text{heps}(n)$ tulostaa 2, mutta arvon välityksellä 1
- esim. `int i=4, &j=i; std::cout<<i; j=5; std::cout<<i;` tulostaa 45
- esim. `int i,A[1]; i=0; A[1]=3; std::cout << i;` tulosti kokeessani 3
  - kääntäjän optimoinnit piti kytkeä pois

Funktion nimen välityksellä palautettava arvo yleensä kopioidaan

- esim. `return 2*i+1;`
- kuten call by result
- jos nimellä on palautettava useita arvoja, ne on yhdistettävä kootuksi arvoksi
  - esim. palautetaan tietue tai olio
  - esim. C++:ssa jotkin tietorakenteeseen lisäystoiminnot palauttavat tietueen, jonka `.first` on kahva lisättyyn ja `.second` kertoo lisättiinkö
  - saattaa olla kätevämpää palauttaa tiedot viiteparametreilla

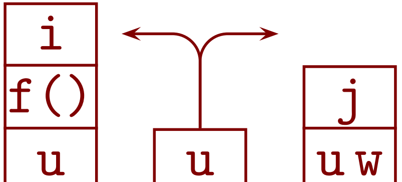
- C++:lla voi ilmaista viitetyyppisen paluuarvon, mutta siinä on ongelmansa
  - seuraava tulosti kokeessani 6

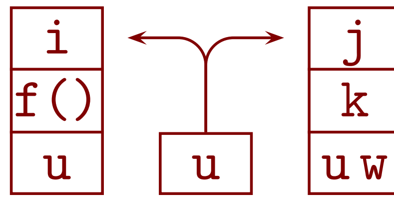
```
int &f(){ int i=6, &v=i; return v; }
void g( int &w ){ int j=7; std::cout << w; j=j; }
int main(){ int &u = f(); g(u); }
```

  - seuraava tulosti kokeessani 7

```
int &f(){ int i=6, &v=i; return v; }
void g( int &w ){ int j=7,k=8; std::cout << w; j=k; }
int main(){ int &u = f(); g(u); }
```

  - periaatteessa mahdollinen selitys (todellisuus on monimutkaisempi)





⇒ tarvittaessa on mietittävä muistin varaus paluuarvolle

Rekursiivinen aliohjelma voi kutsua itseään suoraan tai välillisesti

- esim. Quicksort kutsuu itseään suoraan
- esim. P kutsuu Q:ta, joka kutsuu P:tä

Rekursio on kätevä ohjelmointitekniikka silloin, kun
tehtävä jakautuu samanlaisiksi pienemmiksi osatehtäviksi

- **pohjatapaus** ratkaistaan ilman rekursiivista kutsua
  - pohjatapaus on oltava, jotta suoritus päättyisi
- esim. Quicksortissa
  - taulukko jaetaan kahdeksi pienemmäksi taulukoksi, jotka järjestetään Quicksortilla
  - pohjatapauksena on ≤ 1 alkion taulukko
- esim. lausekkeen käsittelyssä alalausekkeet käsitellään rekursiivisesti

  laske(s) $\mapsto$ `float` // s on lausekepuun solmu

  if $s\uparrow.laji$ = `'+'` **then return** laske($s\uparrow.vasen$) + laske($s\uparrow.oikea$)

  ...

  if $s\uparrow.laji$ = '$\surd$' **then return** $\sqrt{\text{laske}(s\uparrow.vasen)}$

  ...

Jokainen rekursiotaso luo uuden aktivaatiotietueen

- muistin kulutus voi muodostua ongelmaksi, jos rekursiotasoja on paljon ja aktivaatiotietueessa on paljon tietoa, josta ei tarvitse olla erilliset kopiot

  kertoma(n) $\mapsto \mathbb{N}$ // n, tulos ja paluuosoite

  if $n = 0$ **then return** 1 **else return** $n \cdot$ kertoma($n-1$)

- monet kääntäjät korvaavat rekursion silmukalla silloin, kun se on helppoa
  - esim. kun rekursiivinen kutsu on aliohjelman viimeinen teko

    kertoma(n) $\mapsto \mathbb{N}$

    $i := 0$; $tulos := 1$
    while $i \neq n$ **do** $i := i + 1$; $tulos := i \cdot tulos$
    return $tulos$

- kokeessani g++:lla käännetty `kertoma` kulutti
  - 32 tavua / rekursiotaso ilman optimointia
  - 16 tavua / rekursiotaso optimointitasolla -O1 ja yli

  $\Rightarrow$ ei osannut optimoida rekursiota pois

Quicksortissa on helppo muuttaa jompikumpi rekursiivinen kutsu silmukaksi, muttei molempia

- on muistettava taulukon jakokohta a joka rekursiotasolta
  - molempien kutsujen poistamiseksi olisi tehtävä esim. oma pino
- huonoimmassa tapauksessa rekursiiviseen kutsuun menee aina yhtä pienempi taulukko kuin oli edellisessä kutsussa
  - rekursiotasojen määrä voi olla lähes taulukon koko

  $\Rightarrow$ kuluttaa lisää muistia taulukon kokoon verrannollisesti

- valitsemalla pienempi osa rekursiiviseen kutsuun varmistetaan, että rekursiotasojen määrä $\leq 1 + \log_2$(taulukon koko) (kun koko > 0)

  Quicksort(a_1, y_1)

  while $a_1 < y_1$ **do**
  - $x := A[$ random$(a_1, y_1)\,].x$
  - $\ldots$
  - **if** $a - 1 - a_1 < y_1 - a$
  - **then** Quicksort$(a_1, a - 1)$; $a_1 := a$
  - **else** Quicksort(a, y_1); $y_1 := a - 1$

C++-kääntäjältä voi toivoa, että se kopioi aliohjelman koodin kutsun paikalle

- `inline`
- jos kääntäjä toteuttaa toiveen, niin aktivaatiotietue jää (osittain) pois
  - voidaan yhä tarvita muistia paikallisille muuttujille
- rekursiivisen aliohjelman kohdalla toivetta ei voi toteuttaa

Makrot

- makrossa teksti korvataan makron kutsun paikalle siten, että muodolliset parametrit korvataan kutsuparametreilla
  - esim. jos määritelmä on `#define nelio(x) (x)*(x)` , niin kutsu `a = nelio(1+2);` tuottaa `a = (1+2)*(1+2);`

- korvaus kohdistuu nimenomaan tekstiin, ja sen rakennetta ei tulkita
  - esim. jos määritelmä on `#define nelio(x) x*x` ,
    niin kutsu `a = nelio(1+2);` tuottaa `a = 1+2*1+2;` ↯
- esimerkki: `#define vaihda(i,j){int k=i; i=j; j=k;}`
  - kutsu `vaihda(n,m);` tuottaa `{int k=n; n=m; m=k;};` ·/·
  - kutsu `vaihda(n,k);` tuottaa `{int k=n; n=k; k=k;};` ↯
  - kutsu `vaihda(i,A[i]);` tuottaa `{int k=i; i=A[i]; A[i]=k;};` ↯

⇒ **makroja kannattaa välttää**

- esimerkki
  - luodaan puhelinluettelo:

    ```
    std::map< std::string, std::string > puh_luettelo;
    ```
  - kirjoitetaan makro lisäämistä varten:

    ```
    #define lisaa( nimi, nro ){ puh_luettelo[ nimi ] = nro; }
    ```
  - lisäys `lisaa( "Jussi", "040 123 4567" );`
    muuttuu koodiksi `puh_luettelo[ "Jussi" ] = "040 123 4567";;`
  - seuraava kääntyy varoituksetta; mitä se tekee?

    ```
    lisaa( "Häjy", "050 999 9999"; puh_luettelo.clear() );
    ```
  - C++:n tapauksessa tämä on ansa vain ohjelmoijalle, mutta tulkattavien veppikielten tapauksessa loppukäyttäjä voi päästä hyökkäämään näin!

Parametrinvälitysmekanismien muistilista

- mihin suuntaan tieto kulkee?
  - kutsujalta aliohjelmalle
  - aliohjelmalta kutsujalle
  - molempiin suuntiin
- siirretäänkö varsinainen tieto vai ainoastaan pääsykeino siihen?
- aliohjelmaan tulevasta tiedosta
  - lasketaanko tiedon arvo aliohjelman kutsuhetkellä vai tiedon käyttöhetkellä?
  - millä hetkellä tiedon lähtöpaikan vasen-arvo lasketaan?
- kutsujalle tulevasta tiedosta
  - lasketaanko tiedon arvo aliohjelmasta paluuhetkellä vai aikaisemmin?
  - millä hetkellä tiedon vastaanottajan vasen-arvo lasketaan?
- jos parametri on iso, siirretäänkö iso määrä tietoa?

3.3 Abstraktioista, turvallisuudesta ja tehokkuudesta

Ohjelmoinnin tueksi on aina keksitty abstraktioita

- koneen tasolla on rekistereitä, osoitteita, muistipaikkoja ja yksinkertaisia asioita tekeviä käskyjä
  - rekisteri on kuin muistipaikka, mutta laskenta tapahtuu niissä ja rekisterillä on yleensä nimi mutta ei osoitetta (vaihtelua esiintyy)
- käskyille ja muistipaikoille annettiin nimiä kuten `JNZ` ja `cnt`
  - helpompia muistaa kuin 1001100100101101 ja 0100110010100011
- muuttujien sijoittaminen muistiin ja hyppyjen kohdeosoitteiden selvittäminen jätettiin tietokoneen tehtäväksi
- toistuvasti tarvittavia käskyjonoja abstrahoitiin aliohjelmiksi
  - matemaattisia funktioita ja muutakin abstrahoitiin funktioiksi
- matemaattisten merkintöjen matkimiseksi otettiin käyttöön lausekkeita
- erilaisten tietojen hallitsemiseksi keksittiin tyypin käsite: merkki, luonnollinen luku, liukuluku, ...
- tunnistettiin ohjausrakenteita ja pyrittiin pois hypyistä sinne tänne
  - silmukat, valintalauseet, poikkeukset, ...
- yhteenkuuluvia tietoja niputettiin tietueiksi

- tietoa ja toimintoja niputettiin pakkauksiksi (”esiolio”) ja olioiksi
  - luokkahierarkialla voidaan hyödyntää läheisten tyyppien samankaltaisuutta
- muoteilla (template) hyödynnettiin samankaltaisuutta, jota aikaisemmat keinot eivät pystyneet hyödyntämään
  - muotteja parametroidaan tyypeillä, joiden ei tarvitse olla kovin samanlaisia
  - (siihen lisättiin mm. parametrointi vakioilla)
  - ≈ saman lähdekoodin kääntämistä moneen kertaan eri tyypeille
- lähes kaikkea niputettiin nimiavaruuksiksi
- tärkeimmät tietorakenteet otettiin mukaan kielten kirjastoihin
  - esim. C++:n `map`
  - näkökulma on tietorakenteen tuottaman palvelun eikä sen toteutuksen
- ...
- tänään tarvittaisiin hyviä abstraktioita rinnakkaisuudelle!
  - prosessorit, joissa on monta ydintä
  - järjestelmät, joista osa on veppiselaimessa ja osa pilvessä

Abstraktioiden tavoitteena on

- mahdollistaa luontevien käsitteiden käyttö ohjelmoinnissa
  - $opiskelijan_nimi \sim Muisti[27683 \ldots 27695]$
  - henkilötietue ∼ merkkijono, luku, luku, bitti, ...

- siirtää alatason yksityiskohtia koneen tehtäväksi
  - esim. merkkijonojen tarvitseman muistin varaaminen ja vapauttaminen
- vähentää ohjelmointivirheitä
  - esim. tyypit estävät kokonaisluvun sijoittamisen vahingossa osoittimeksi
  - esim. `const` pyrkii estämään tiedon muuttamisen vahingossa
- rajoittaa vaikutuksia muualle ohjelmaan, jos abstraktion toteutus vaihtuu
  - esim. opiskelijarekisterin toteutus joudutaan korvaamaan tehokkaammalla
  - jotta tämä toteutuisi, opiskelijarekisterin toteutus pitää kätkeä

Tiedon kätkentä (information hiding)

- ohjelman osalle määritellään toteutuksen lisäksi rajapinta
- muut osat voivat käyttää rajapintaa, mutta eivät voi koskea toteutukseen
- esim. C++ `private` ja `public`

```
class pino_100{
   float A[100]; unsigned n;
public:
   int määrä(){ return n; }
   void lisää( float x ){ if( n<100 ){ A[n] = x; ++n; } }
   void poista(){ if( n>0 ){ --n; } }
   float ylin(){ return n ? A[n-1] : 0; }
};
```

Jatkossa tulee asian lisäksi myös propagandaa

- ole eri mieltä vieläkin vapaammin kuin tavallisesti!

Jotta abstraktio vähentäisi virheitä, sen pitää rajoittaa ohjelmointia

- **enemmän työtä ohjelmointivaiheessa**, mutta
  vähemmän korjaustyötä ja/tai jäljelle jääviä virheitä
  - ohjelmoijaa vaaditaan antamaan lisätietoa,
    joka auttaa kääntäjää löytämään virheitä
- liian ahtaat rajoitteet ovat kiistatta vahingollisia
  - jollei samaa muistialuetta voi käsitellä henkilötietueena ja bittikuviona, ei voi ohjelmoida tiedon siirtoa kiintolevylle eikä muistin kierrätystä
  - maailmaa eivät vallanneet elegantit kielet, joilla voi tehdä melkein kaiken tarvittavan, vaan rumat kielet, joilla voi tehdä aivan kaiken tarvittavan
- makrot mahdollistavat vaikka mitä, mutta osoittautuivat liian virhealtteiksi
- hyvä kompromissi: osan asioista voi tehdä vain erityisillä komennoilla
  $\Rightarrow$ niitä ei tee vahingossa eikä salaa

Abstraktiot ja turvallisuusominaisuudet voivat vähentää suoritustehoa

- taulukon indeksoinnin tarkastukset kuluttavat hieman aikaa
- tietorakenteen käsitteleminen suoraan saattaa olla tehokkaampaa kuin rajapinnan läpi

- turvallisen version voi toteuttaa tehokkaan päälle, mutta ei toisinpäin

⇒ C++:n säiliöissä (esim. `list`, `map` ja `deque`) on suosittu tehokkuutta . . .
  - . . . mutta vastaavat turvalliset ovat jääneet enimmäkseen toteuttamatta

Ajoaikaisissa virheilmoituksissa on etunsa ja haittansa

- on parempi, että verkkopankki sanoo ”system crash” kuin päästää varkaan sisään
- ei ole kiva, jos lentokoneen ohjausjärjestelmä sanoo ”system crash” ja seuraavat 5 minuuttia käynnistyy uudelleen ennen kuin taas tottelee
- lentokoneen ohjausjärjestelmänkin testausvaiheessa on hyvä, jos virheet tulevat mahdollisimman varmasti ilmi
  ⇒ ajoaikaiset tarkastukset tulee voida kytkeä päälle ainakin osassa testejä

Kielessä tulisi olla riittävä etäisyys merkityksiltään eroavien ilmausten välillä

- pienen ajatus- tai kirjoitusvirheen tulisi aiheuttaa käännös- tai ajoaikainen virhe, eikä eri asian tekevä koodi
- esimerkki
  - C++:n ja-operaation `&&` sijaan kirjoittaa helposti `&`
  - kääntäjä ei varoita, koska myös `&` on laillinen operaattori
  - `x & y` tuottaa vain harvoin eri tuloksen kuin `x && y`

  ⇒ virhettä on vaikea havaita testeissä
  ⇒ joudutaan opiskelemaan `&`, jotta osattaisiin varoa sitä

- esim. vuosien takainen yritys vallata Linux

```
if( user = 0 || ... ){ ... }
```

- **C++:sta on opiskeltava paljon asioita, joita ei edes aio käyttää**
  - koska kääntäjä ei varoita, on ohjelmoijan opiskeltava paljon asioita muotoa "varo tekemästä näin" ja muistettava ne ohjelmoidessaan
  - tämäkö on ohjelmoijan aivokapasiteetin järkevää käyttöä?

Uusia turvallisuusominaisuuksia on vastustettu ankarasti

- monet hyväksyttiin lopulta yleisesti
  - **goto**:n korvaaminen valinta- ja silmukkarakenteilla
  - velvollisuus määritellä muuttujat ennen käyttöä
  - virheilmoitukset, kun sijoitetaan väärään tyyppiin
  - ohjelman keskeyttäminen, kun taulukkoa indeksoidaan ohi rajojen
- silti uudet ominaisuudet tehdään kovin usein turvattomiksi
  - `std::vector<int> V; V.pop_back(); std::cout << V.size();` tulosti 4294967295
  - esim. Javan rinnakkaismalli
- kokevatko ohjelmoijat turvallisuusominaisuudet loukkaavina?
  - vain lälläri käyttää konttauskypärää tai indeksoinnin rajatarkastuksia
  - sekö mielletään ammattitaidoksi, että osaa varoa ansoja (kuten `&` vs. `&&`), joita ei tarvitsisi olla olemassakaan?
- virheilmoitukset eivät ole ohjelmoijan kyykyttämistä, vaan auttamista!

Silti pyrkimys abstraktioihin ei ole pelkästään hyväksi

- osa abstraktioista on selvästi hyviä
  - esim. C++:n `string` ja `vector`
  - esim. muistin ajatteleminen taulukkona, jota indeksoidaan osoittimilla
- valitettavasti abstraktio tai rajapinta voi olla huonosti valittu
  - yleensä vasta käytön myötä selviää, mitä oikeasti tarvitaan
  - varsinkin kun käyttötarpeet muuttuvat
  - rajapintojen suunnittelu vaatii paljon tietoa, hiljaista tietoa ja vaistoa
- toteutuksen helppo vaihdettavuus on usein näennäinen hyöty
  - siihen on loppujen lopuksi varsin harvoin tarvetta
  - se ei onnistu heittämällä edes silloin, kun kumpikin on C++:n säiliö

⇒ liian innokas tiedon kätkeminen yms. ei vastaa tarkoitustaan

- valitettavasti monet abstraktiot **vuotavat** (**leak**) pahasti
  - on pakko ottaa huomioon asioita, jotka abstraktio yrittää kätkeä
  - esim. jos osoittaa C++:n `vector`:ssa olevaan alkioon, on ymmärrettävä, että se voi siirtyä toiseen muistipaikkaan, jolloin osoitin vanhenee
  - C++:n säiliöihin `list`, `map`, `deque` jne. liittyy niin paljon tällaisia sääntöjä, että niitä on toivotonta yrittää hallita muuten kuin tuntemalla toteutukset (ks. netistä Käytännön kokemus algoritmikirjastojen autuudesta)

  ⇒ **pahimmassa tapauksessa on koko ajan pidettävä mielessä sekä abstraktio että sen toteutus**
  - **aivoille tuli lisää kuormaa, kun kuormaa piti vähentää**

4 Algoritmien toimivuus ja nopeus

4.1 Algoritmien ohjelmoinnista

Mitä vikoja löydät alla olevasta Quicksortin toteutuksesta?

- lähde: S.B. Lippman, C++ Primer, s. 128, 1989
  - ladontaa muutettu ruudulle mahtuvaksi, muuten suora kopio

```
static void swap( int *ia, int i, int j )
   { int tmp = ia[i]; ia[i] = ia[j]; ia[j] = tmp; }
void qsort( int *ia, int low, int high ){
   if( low < high ){
      int lo = low; int hi = high + 1; int elem = ia[ low ];
      for (;;){
         while ( ia[ ++lo ] <= elem ) ;
         while ( ia[ --hi ] > elem ) ;
         if( lo < hi ) swap( ia, lo, hi );
            else break;
      } // end, for(;;)
      swap( ia, low, hi );
      qsort( ia, low, hi - 1 ); qsort( ia, hi + 1, high );
   } // end, if ( low < high )
}
```

- koeajossani 100 000 alkion taulukon järjestämisaika oli
  - silmänräpäys, kun alkiot olivat erisuuria ja sekalaisessa järjestyksessä
  - 14 s, kun alkiot olivat erisuuria ja valmiiksi järjestyksessä
  - 85 s, kun alkiot olivat yhtäsuuria
- eräällä pienellä syötteellä ohjelma kaatui ilmoittaen `Muistialueen ylitys`
- kirjan 2nd Edition:ssa s. 138 (ja vepissä) <=:n tilalle on vaihdettu <

```
while ( ia[ ++lo ] < elem ) ;
```

$\Rightarrow$ edellä mainittu 85 s muuttui silmänräpäykseksi, muut viat säilyivät

Ennen selityksen antamista laskemme, miten ehjä Quicksort käyttäytyy, kun taulukko on iso, kaikki alkiot ovat erisuuria ja jako on aina satunnainen ja kahtia

- olkoon n tutkittavan osataulukon koko (eli esim. $\texttt{high} - \texttt{low} + 1$)
- kun $n \leq 1$, niin ositusta ei tehdä
  - rekursion pohja
- kun $n \geq 2$, niin olkoon $p(n)$ todennäköisyys, että pienempi osa $< \frac{1}{4}$ taulukosta
- todistamme, että $p(n) \leq \frac{1}{2}$
  - n voidaan esittää muodossa $4k + h$, missä $1 \leq h \leq 4$
  - merkitään alkuosan kokoa a
  - osat eivät tyhjiä $\Rightarrow 1 \leq a \leq n - 1$
  - pienempi osa on alle neljäsosa, joss $1 \leq a \leq k$ tai $3k + h \leq a \leq n - 1$

  $\Rightarrow p(n) \leq \frac{2k}{n-1} = \frac{2k}{4k+h-1} \leq \frac{1}{2}$ ·/·

- arvioimme rekursiotasojen määrän odotusarvoa $r(n)$ ylhäältä
  - selvästi jos $n < m$, niin $r(n) \leq r(m)$
  - jos pienempi osa on vähintään $\frac{1}{4}$ taulukosta, niin alempia tasoja $\leq r(\lfloor \frac{3n}{4} \rfloor)$
  - muussa tapauksessa alempia tasoja $\leq r(n)$

  $\Rightarrow r(n) \;\leq\; 1 + (1 - p(n))r(\lfloor \frac{3n}{4} \rfloor) + p(n)r(n) \;\leq\; 1 + \frac{1}{2}r(\lfloor \frac{3n}{4} \rfloor) + \frac{1}{2}r(n)$ (koska $(p(n) - \frac{1}{2})(r(n) - r(\lfloor \frac{3n}{4} \rfloor) \leq 0$ eli $p(n)r(n) - p(n)r(\lfloor \frac{3n}{4} \rfloor) \leq \frac{1}{2}r(n) - \frac{1}{2}r(\lfloor \frac{3n}{4} \rfloor))$

  $\Rightarrow r(n) \leq r(\lfloor \frac{3n}{4} \rfloor) + 2$
- osoitamme induktiolla, että $r(n) < 5 \log_2 n$ kun $n \geq 2$
  - $r(2) = 2 < 5 = 5 \log_2 2$ ja $\lfloor \frac{3n}{4} \rfloor < n$ kun $n > 2$
  - $r(n) \leq r(\lfloor \frac{3n}{4} \rfloor) + 2 < 5 \log_2(\frac{3n}{4}) + 2 = 5 \log_2 n + 5 \log_2 3 - 8 < 5 \log_2 n$
- (numeerinen lasku vihjaa, että $r(n) \approx 2{,}26 \log_2 n + 1{,}45$)
- rekursiotasolla selataan kukin alkio enintään kerran

$\Rightarrow$ *kokonaistyömäärä on enintään verrannollinen lukuun* $n \log_2 n$

- työmäärä on pienimmillään, kun ositus menee aina mahdollisimman tasan
  - rekursiotasoja $\lceil \log_2 n \rceil + 1$ (kun $n \geq 1$)
  - alinta lukuun ottamatta kukin alkio selataan kerran

  $\Rightarrow$ *kokonaistyömäärä on vähintään verrannollinen lukuun* $n \log_2 n$

$\Rightarrow$ suoritusaika on verrannollinen lukuun $n \log_2 n$

Edellä käytetty ylälikiarvon muodostustekniikka kannattaa osata

- jako ”hyviin” ja ”huonoihin” tapauksiin, joilla mahdollisimman huono edustaja

Lippmanin Quicksortin käyttäytyminen

- kun alkiot ovat erisuuria ja sekalaisessa järjestyksessä, niin se käyttäytyy (melkein) kuten pitääkin
  - alla mainittuja ilmiöitä tapahtuu hyvin harvoin
  - ⇒ suoritusaika on verrannollinen lukuun $n \log_2 n$
- kun alkiot ovat erisuuria ja valmiiksi järjestyksessä, niin
  - `elem` on pienempi kuin muut osataulukon alkiot
  - `while ( ia[ ++lo ] <= elem ) ;` lopettaa heti
  - `while ( ia[ --hi ] > elem ) ;` lopettaa vasta kun `hi` = `low`
  - ⇒ yläosa sisältää vain yhden alkion vähemmän kuin saatu taulukko
  - ⇒ selataan $n + (n-1) + (n-2) + \ldots + 3 + 2 = \frac{1}{2}n^2 + \frac{1}{2}n - 1$ alkiota
  - kun n on iso, se on paljon enemmän kuin $5n \log_2 n$
  - ⇒ 14 s sai selityksen
- jos `ia[ low ]` on suurempi tai yhtäsuuri kuin muut taulukon alkiot, niin `while ( ia[ ++lo ] <= elem ) ;` jatkaa taulukon loppuun ja sen ohi
- se lopettaa vasta, kun se
  - löytää tarpeeksi ison luvun tai
  - poistuu käyttäjälle sallituista osoitteista ⇒ `Muistialueen ylitys`
- koeajossa `Muistialueen ylitys` saatiin laittamalla taulukon ensimmäiseksi suurin mahdollinen `int`
  - jos muistialuetta ei olisi suojattu, niin <=-versio jäisi ikuiseen silmukkaan

- kun alkiot ovat yhtäsuuria ja versio on <=, niin
  - `while ( ia[ ++lo ] <= elem ) ;` selaa taulukon lopun ohi mutta yleensä lopulta löytää riittävän ison arvon pysähtyäkseen
  - `while ( ia[ --hi ] > elem ) ;` lopettaa heti (`hi` = `high`)
  - ⇒ alaosa sisältää vain yhden alkion vähemmän kuin saatu taulukko
  - ⇒ rekursiotasoja yhtä paljon kuin 14 s tapauksessa, mutta selausta tulee enemmän, koska selataan tutkittavan osan ohi ja koko taulukon ohi
  - ⇒ 85 s sai selityksen
- kun alkiot ovat yhtäsuuria ja versio on <, niin
  - kumpikin silmukka lopettaa joka kerta heti
  - ⇒ alku- ja loppuosasta tulee mahdollisimman tarkasti yhtä suuret
  - ⇒ suoritusaika on paras mahdollinen

Luotettavien ohjelmien tekeminen vaatii taitoa ja huolellisuutta

- Lippman yritti huonolla menestyksellä kolmijakoa pienet, `elem`, suuret
- `while ( ia[ ++lo ] <= elem ) ;` on jälkiviisaasti ilmeisen väärin
- `Muistialueen ylitys` on vaikea havaita testaamalla
  - onko tapanasi testata `INT_MAX`:lla?
  - vaikka Lippman korjasi koodiaan, tämä virhe jäi jäljelle
  - haittaako virhe, joka ei löydy testaamalla?
- huolellinen testaus olisi löytänyt muut virheet
  - olisi sisältänyt tapaukset ”kaikki samaa” ja ”valmiiksi järjestyksessä”

Sivun 38 Quicksort:n toiminnan perustelu

```
Quicksort(a₁, y₁)
if a₁ ≥ y₁ then return
x := A[ random(a₁, y₁) ].x
a := a₁ − 1; y := y₁ + 1
while a < y do
    a := a + 1; y := y − 1
    while A[a].x < x do a := a + 1
    while A[y].x > x do y := y − 1
    if a < y then apu := A[a]; A[a] := A[y]; A[y] := apu
if a = a₁ then a := a + 1
Quicksort(a₁, a − 1); Quicksort(a, y₁)
```


- 0:n tai 1:n alkion taulukolle ei tarvitse tehdä mitään
- x valitaan A:sta, jotta sisemmät **while**-silmukat varmasti lopettavat
- x valitaan satunnaisesti, jotta jako alku- ja loppuosaan olisi usein hyvä
  - tuottaa lukuun $n \log_2 n$ verrannollisen suoritusajan
  - jos A on valmiiksi järjestyksessä ja kaikki alkiot ovat erisuuria, niin valinta aina $A[a_1]$ tuottaa lukuun n^2 verrannollisen suoritusajan
- ulomman **while**-silmukan kierroksen alussa pätee aina
  - jokainen a:n kohdalla ja siitä vasemmalle $\leq x$, jokin a:sta oikealle $\geq x$
  - jokainen y:n kohdalla ja siitä oikealle $\geq x$, jokin y:stä vasemmalle $\leq x$

- ulomman **while**-silmukan lopetettua
  - $a_1 \leq y \leq a \leq y_1$
  - $A[a].x \geq x$ ja $A[y].x \leq x$
  - kun $a_1 \leq i < a$, niin $A[i].x \leq x$
  - jos $y < a$, niin kun $a \leq i \leq y_1$, niin $A[i].x \geq x$
  - jos $y = a$, niin $A[a].x = x$ ja kun $a < i \leq y_1$, niin $A[i].x \geq x$
  - jos $a = a_1$, niin $y = a$

$\leq x$	x	$\geq x$
a_1	y a y	y_1

$\Rightarrow$ rekursiiviset kutsut on rajattu siten, että alkuosassa $\leq x$ ja loppuosassa $\geq x$

- kumpikin osa on alkuperäistä taulukkoa pienempi
  - $A[a]$ ei kuulu alkuosaan ja $A[a_1]$ ei kuulu loppuosaan
  - ilman viimeistä **if**-lausetta loppuosa voisi olla koko taulukko
    $\Rightarrow$ Quicksort(a_1, y_1) voisi kutsua Quicksort(a_1, y_1) $\Rightarrow$ ikuinen rekursio
- jos kaikki A:n alkiot ovat yhtäsuuria, niin tämä ositus jakaa keskeltä

Taulukon sekoittaminen

- tätä tarvitaan joissakin kokeellisissa harjoitustehtävissä

for $i := 1$ **to** n **do** $A[i] := i$
for $i := n$ **downto** 2 **do**
    $j :=$ random$(1, i)$
    $x := A[i]$; $A[i] := A[j]$; $A[j] := x$

- $j :=$ random$(1, n)$ tuottaisi vinon jakauman

4.2 Asymptoottinen suoritusaika

Insertion-sort

- edellä näimme, että Lippmanin Quicksortin suoritusaika riippuu paitsi syötteen koosta, myös sen muista ominaisuuksista
- tälle tärkeälle (ja ehjälle) järjestämisalgoritmille ilmiö on paljon selvempi
  - $A[1 \ldots i-1]$ on järjestyksessä aina **for**-silmukan ehdon kohdalla
  - $A[i]$ otetaan talteen muuttujaan $apu \Rightarrow$ kohtaan i jää "aukko"
  - aukkoa siirretään vasemmalle, kunnes se on $apu.x$:n mukaisessa kohdassa
  - apu laitetaan aukkoon

```
Insertion-sort
for i := 2 to n do
    apu := A[i]; j := i
    while j > 1 && A[j − 1].x > apu.x do
        A[j] := A[j − 1]; j := j − 1
    A[j] := apu
```

- kun $n > 2$, taulukko on takaperin järjestyksessä ja koostuu erisuurista alkioista, eniten suoritettava toiminto on testi $j > 1$ && $A[j-1].x > apu.x$
  - se tehdään $(n-1) + 1 + 2 + \ldots + (n-1) = \frac{1}{2}n^2 + \frac{1}{2}n - 1$ kertaa
- etuperin järjestyksessä olevalle kukin toiminto tehdään n, $n-1$ tai 0 kertaa

Suoritusaikaa ei yleensä yritetäkään ennustaa millisekunteina

- se ei riipu pelkästään syötteen koosta, vaan myös sen ”laadusta”
- se ei riipu pelkästään algoritmista, vaan myös toteutuksen yksityiskohdista
- kun kone, kääntäjä, käyttöjärjestelmä tms. vaihtuu, niin suoritusaika muuttuu
- suoritusaika, jopa CPU-aika, riippuu koneen muusta kuormituksesta
  - nykyisissä koneissa mitattu suoritusaika vaihtelee joskus hallitsemattomasti

Suoritusaikoja voi mitata, kun ohjelma on valmis, mutta

- harvoin on varaa toteuttaa 3 vaihtoehtoa, joista valitaan mittaamalla paras
  - sitäpaitsi usein on monta kohtaa, joissa on vaihtoehtoja

  $\Rightarrow$ tarvittaisiin esim. $3 \cdot 2 \cdot 2$ vaihtoehtojen yhdistelmää
- paremmuusjärjestys voi riippua syötteen laadusta
  - Insertion-sort on ylivoimainen, jos syöte on melkein järjestyksessä
  - Quicksort on ylivoimainen, jos syöte on satunnaisessa järjestyksessä
  - esim. ostotapahtumat tulevat pankkiin suunnilleen aikajärjestyksessä
- usein ei tiedetä, minkälainen syöte on tyypillistä tai edes satunnaista
  - jos kaikki alkiot ovat erisuuria, niin voidaan arpoa satunnainen järjestys
  - jollei, niin kuinka monesti kukin alkio pitää toistaa?

$\Rightarrow$ Tarvitaan muunlainen tapa luonnehtia suoritusaikaa ja ratkaista ohjelmointivaiheessa, mikä vaihtoehto kannattaa toteuttaa

Suoritusaikaa kuvaavan kaavan muodon merkitys

- olkoot algoritmien A, B, C ja D suoritusajat n, $n \log_2 n$, n^2 ja 2^n millisekuntia
- jos $n = 1\,000$, niin suoritusajat ovat
  A: 1 s B: 10 s C: 16 min 40 s D: 2,5·10^{280} maailmankaikkeuden ikää
- ≈ 1 min aikana ratkeavan tehtävän koko on enintään
  A: 60 000 B: 4 895 C: 245 D: 16
- jos koneen nopeus kaksinkertaistuu, niin
  - $n = 1\,000$ ratkeaa ajassa A: 0,5 s B: 5 s C: 8 min 20 s D: 1,2·10^{280} mi.
  - 1 min aikana ratkeavan koko on A: 120 000 B: 9 122 C: 346 D: 17

  ⇒ nopeuden kaksinkertaistaminen auttaa C:llä vähän ja D:llä vielä vähemmän
- usein kuitenkin vastakkain ovat C ja B', jonka suoritusaika on $40n \log_2 n$
  - B' on monimutkainen ja siksi hidas pienillä syötteillä

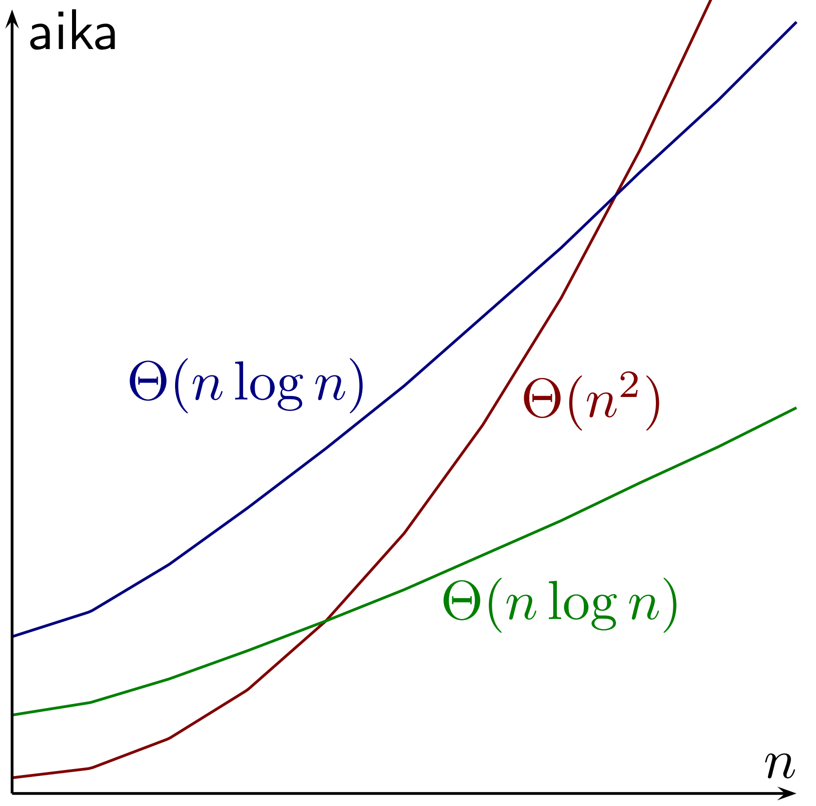
n	10	100	1 000	10 000
B'	1,3 s	26,6 s	6 min 39 s	1 h 29 min
C	0,1 s	10 s	16 min 40 s	1 vrk 3 h 47 min

- ≈ 1 min aikana ratkeavan koko
  - alkuperäisellä koneella B': 197 C: 245
  - nopeutetulla koneella B': 354 C: 346

⇒ mitä nopeampi kone, sitä varmemmin B' voittaa

Näistä syistä suoritusaikaa ei yleensä arvioida sekunteina vaan käyrän muotona

- **asymptoottinen suoritusaika**
  - suoritusajan lausekkeen eniten merkitsevä termi ilman vakiokerrointa
  - esim. $7n^2 - 2n + 5 \rightsquigarrow \Theta(n^2)$
  - täsmällinen määritelmä tulee kohta
- se on "ensimmäinen likiarvo"
  - se on yleensä niin hyvä, että muuta ei tarvita!



Jos syötteet voivat olla isoja, yleensä kannattaa valita asymptoottisesti hyvä vaihtoehto

- pienillä syötteillä yleensä kaikki vaihtoehdot ovat riittävän nopeita

- suurilla syötteillä se voittaa (usein ylivoimaisesti), jolla on hitain kasvunopeus
- esimerkki: konetehon kasvun vaikutus aikaan, jona valokuva ilmestyy ruudulle
  - 1 h $\rightsquigarrow$ 30 min ei hyödytä, koska kukaan ei jaksa odottaa edes 30 min
  - 0,2 s $\rightsquigarrow$ 0,1 s ei hyödytä, koska kukaan ei huomaa eroa
  - 0,2 s aikana ilmestyvän kuvan laadun paraneminen on hyödyksi

$\Rightarrow$ usein ei haluta vastausta nopeammin samalle n, vaan samassa ajassa isommalle n
  - kuten jo näimme, tällöin konetehon kasvu *lisää* käyrän muodon tärkeyttä

Kaikki olennaiset näkökohdat tulee ottaa huomioon oikein painotettuina

- asymptoottisesti hyvä on usein paljon vaikeampi toteuttaa kuin huono
- jos syötteet ovat tulevaisuudessakin *varmasti* aina pieniä, ei kannata valita monimutkaista ratkaisua
- jos sama algoritmi ajetaan monesti ja joka kerta pienellä syötteellä, kannattaa kiinnittää huomio suoritusaikaan *pienillä* syötteillä
- joidenkin asymptoottisesti hyvien algoritmien edut tulevat esiin vasta epärealistisen suurilla syötteillä
- yleisin virhe on kuitenkin aliarvioida asymptoottisen suoritusajan merkitystä
- suoritusajan parantaminen muilla keinoilla onnistuu yleensä huonosti
  - ostamalla nopeampi kone saadaan vasteaika esim. puolitettua, riittääkö se?
  - asymptoottisesti hyvän algoritmin huono toteutus on yleensä paljon nopeampi kuin asymptoottisesti huonon algoritmin hyvä toteutus

Usein tyydytään asymptoottisen suoritusajan ylälikiarvoon

- tarkan muodon selvittäminen on usein paljon vaikeampaa kuin ylälikiarvon
- jos osan A suoritusajan tiedetään kasvavan enintään yhtä nopeasti kuin osan B, A:sta ei tarvitse tietää muuta
  - A ei voi huonontaa koko ohjelman asymptoottista suoritusaikaa
- jos ylälikiarvo on riittävän hyvä, tiedetään, että algoritmi on tarpeeksi nopea

Olkoon $t(n)$ suoritusaika halutun laatuisen syötteen koon n funktiona

- esim. keskimääräisen syötteen
  - edellyttää, että mielekäs keskiarvon käsite on tiedossa
- esim. jokaisen syötteen
  - sekä hitaimmilla että nopeimmilla saadaan ilmoitettu suoritusaika

Iso O-merkintä

suoritusaika on $O(f(n))$, jos ja vain jos on olemassa
$n_0 \in \mathbb{N}$ ja $c \in \mathbb{R}^+$ siten, että $\forall n \geq n_0 : 0 \leq t(n) \leq cf(n)$

- ilmaisee ylärajan mutta ei alarajaa
  - suoritus voi olla luvattua nopeampi
- n_0 huolehtii, että ei ole väliä, miten $t(n)$ käyttäytyy pienillä n
- c huolehtii, että ei ole väliä, onko $t(n) = 2n^2$ vai $t(n) = 100n^2$ vai ...
- jos kuitenkin esim. $t(n) = \frac{1}{10}n^2 \log_2 n$ ja $f(n) = n^2$,
  niin kun $n > 2^{10c}$ on $t(n) > cf(n)$

Iso Ω-merkintä

suoritusaika on $\Omega(f(n))$, jos ja vain jos on olemassa
$n_0 \in \mathbb{N}$ ja $c \in \mathbb{R}^+$ siten, että $\forall n \geq n_0 : 0 \leq cf(n) \leq t(n)$

- ilmaisee alarajan mutta ei ylärajaa

Θ-merkintä

suoritusaika on $\Theta(f(n))$, jos ja vain jos on olemassa $n_0 \in \mathbb{N}$, $c \in \mathbb{R}^+$ ja $d \in \mathbb{R}^+$ siten, että $\forall n \geq n_0 : 0 \leq cf(n) \leq t(n) \leq df(n)$

- ilmaisee sekä ala- että ylärajan
- suoritusaika on $\Theta(f(n))$, jos ja vain jos se on sekä $O(f(n))$ että $\Omega(f(n))$

Pienet o- ja ω-merkinnät

- pieni o-merkintä ilmaisee, että $t(n)$ kasvaa hitaammin kuin $f(n)$

  suoritusaika on $o(f(n))$, jos ja vain jos $\forall c \in \mathbb{R}^+ :$
  $\exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq t(n) < cf(n)$
- pieni ω-merkintä ilmaisee, että $t(n)$ kasvaa nopeammin kuin $f(n)$

  suoritusaika on $\omega(f(n))$, jos ja vain jos $\forall c \in \mathbb{R}^+ :$
  $\exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq cf(n) < t(n)$

Huomautuksia

- $f(n) \in O(g(n)) \;\Leftrightarrow\; g(n) \in \Omega(f(n))$
- ei ole välttämättä niin, että $t(n)$ on joko $O(f(n))$ tai $\Omega(f(n))$
  - $t(n)$ ja $f(n)$ voivat olla vuorotellen toistensa ala- ja yläpuolella
  - niiden on oltava aina vain enemmän ohi toistensa, jottei ero mahtuisi vakiokertoimen c piiriin

- jos hitain syöte on $O(f(n))$, niin jokainen syöte on $O(f(n))$, ja päinvastoin
- toisinaan suoritusaika ilmaistaan monen muuttujan funktiona
  - esim. graafeille $O(V + E \log V)$, missä V on solmujen ja E kaarten määrä
  - esim. $\Theta(nm)$ on selvästi huonompi kuin $\Theta(n + m)$
- suoritusaika voi myös riippua (voimakkaastikin) muuttujasta, jota ei sanota
- $O(1)$ tarkoittaa, että suoritusajalla on muuttujasta riippumaton yläraja
  - oikea muuttuja tiedetään asiayhteydestä
  - enintään vakioaikainen (mutta ”enintään” saattaa jäädä sanomatta)
- $O(f(n))$, $\Theta(f(n))$ jne. ovat joukkoja funktioita
  $\Rightarrow$ tulisi merkitä $t(n) \in O(f(n))$ jne.
  - tapana on kuitenkin merkitä $t(n) = O(f(n))$ jne.
- muutenkin O- jne. merkintöjä käytetään usein ”väärin”
  - esim. usein n ei todellisuudessa voi olla miten iso tahansa, koska sanapituus on rajallinen

Usein esiintyviä Θ-merkintöjä

- $\Theta(\log n)$ **logaritminen**
  - puhelinnumeron etsiminen aakkosjärjestetystä luettelosta nimen perusteella
- $\Theta(n)$ **lineaarinen**
  - nimen etsiminen aakkosjärjestetystä luettelosta puhelinnumeron perusteella
- $\Theta(n \log n)$ melkein yhtä hyvä kuin lineaarinen

- **$\Theta(n^2)$ neliöllinen**
  - suurilla syötteillä harmillisen hidas
  - yleinen ajattelemattomasti valitun ratkaisun seuraus
- $2^{\Theta(n)}$ **eksponentiaalinen**
  - yleensä epärealistisen hidas jo melko pienellä n, mutta ...
  - ... yleensä on ilmoitettu hitaimmalle tapaukselle
    $\Rightarrow$ osa tapauksista voi olla riittävän nopeita isoillakin n
  - Θ on eksponentissa, koska $\Theta(2^n)$ kiinnittäisi kantaluvuksi 2

Esimerkki arkielämästä

> Jos Kalle kolaa lumet $1\,\mathrm{m}^2$ alalta yhdessä aikayksikössä, niin kolaako hän $5\,\mathrm{m}^2$ alalta 5 aikayksikössä ja $n\,\mathrm{m}^2$ alalta n aikayksikössä?

- mitä isompi alue on kolattavana, sitä pitemmän matkan yksi kolallinen on keskimäärin työnnettävä
- $1 + 2 + \cdots + n = \frac{1}{2}n^2 + \frac{1}{2}n$
- lisäksi kola on aluksi haettava jostain ja lopuksi vietävä takaisin yms.

$\Rightarrow$ työmäärä $\approx cn^2 + dn + e$ (c, d, e vakioita ja $c > 0$)

$\Rightarrow \Theta(n^2)$

O- jne. merkintöjä käytetään ilmaisemaan myös muistin kulutusta

Esimerkki: tärkeimpien järjestämisalgoritmien vertailu

- järjestämisalgoritmi on **vakaa**, jos ja vain jos se avainten ollessa samoja säilyttää alkioiden järjestyksen
  - mahdollistaa järjestämisen monen kriteerin mukaan aloittamalla vähiten tärkeästä
  - esim. sähköpostit ensin lähettäjän, sitten otsikon ja lopuksi ajan mukaan
- Heapsort tulee luvussa 5.2
- Merge-sort tulee harjoitustehtävässä, mutta listoille
  - tässä muistinkulutus on oletettu taulukoille
- Radix-sort ajaa jonkin vakaan järjestämisalgoritmin viimeiselle numeromerkille, sitten toiseksi viimeiselle, jne.
  - d on numeromerkkien määrä avaimessa, oletetaan $d \leq n$
  - alla on oletettu, että apualgoritmi on Counting-sort

nimi	hitaimmillaan	keskimäärin	nopeimmillaan	vakaa	lisämuisti
Insertion-sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	kyllä	$O(1)$
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	ei	$\Omega(\log n)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	ei	$O(1)$
Merge-sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	kyllä	$\Theta(n)$
Counting-sort	$\Theta(n+M)$	$\Theta(n+M)$	$\Theta(n+M)$	kyllä	$\Theta(n+M)$
Radix-sort	$\Theta(nd)$	$\Theta(nd)$	$\Theta(nd)$	kyllä	$\Theta(n)$

Usein arvioinnissa tarvitsee laskea summia tyyliin $1 \log 1 + 2 \log 2 + \ldots + n \log n$

$\Rightarrow$ opettelemme helpon kikan, joka toimii *melkein* aina

- edellyttää, että termit kasvavat ja ovat ei-negatiivisia
  - esim. kaikilla $n > 0$ pätee $(n+1)\log(n+1) \geq n \log n \geq 0$
- yläraja saadaan kertomalla viimeinen termi termien määrällä
  - esim. $$\begin{aligned} 1\log 1 + 2\log 2 + 3\log 3 + 4\log 4 + 5\log 5 + 6\log 6 &\leq \\ 6\log 6 + 6\log 6 + 6\log 6 + 6\log 6 + 6\log 6 + 6\log 6 &= 6 \cdot 6 \log 6 \end{aligned}$$
  - yleisemmin $\sum_{i=1}^{n} i \log i \leq n(n \log n) = n^2 \log n = O(n^2 \log n)$
- alaraja saadaan kertomalla keskimmäinen termi termien määrän puolikkaalla
  - keskimmäisenä voidaan käyttää $\frac{n}{2}$, vaikka se ei olisi kokonaisluku
  - esim. $$\begin{aligned} 1\log 1 + 2\log 2 + 3\log 3 + 4\log 4 + 5\log 5 + 6\log 6 &\geq \\ 0 + \quad 0 + \quad 0 + 3\log 3 + 3\log 3 + 3\log 3 &\geq 3 \cdot 3 \log 3 \end{aligned}$$
  - esim. $$\begin{aligned} 1\log 1 + 2\log 2 + \quad 3\log 3 + \quad 4\log 4 + \quad 5\log 5 &\geq \\ 0 + \quad 0 + 2\tfrac{1}{2}\log 2\tfrac{1}{2} + 2\tfrac{1}{2}\log 2\tfrac{1}{2} + 2\tfrac{1}{2}\log 2\tfrac{1}{2} &\geq 2\tfrac{1}{2} \cdot (2\tfrac{1}{2}\log 2\tfrac{1}{2}) \end{aligned}$$
  - yleisemmin $\sum_{i=1}^{n} i \log i \geq \frac{n}{2}(\frac{n}{2} \log \frac{n}{2}) = \frac{1}{4}n^2(\log n - \log 2) = \Omega(n^2 \log n)$
- jos ylärajaksi ja alarajaksi tulee sama muoto, se voidaan kirjoittaa Θ:na
  - koska saatiin $\sum_{i=1}^{n} i \log i = O(n^2 \log n)$ ja $\sum_{i=1}^{n} i \log i = \Omega(n^2 \log n)$, pätee $\sum_{i=1}^{n} i \log i = \Theta(n^2 \log n)$

Muistutettakoon vielä kerran, että asymptoottiset eli O-, Θ- jne. merkinnät

- eivät kerro mitään siitä, kuinka kauan aikaa kuluu pienillä syötteillä
- eivät lainkaan riipu absoluuttisista ajoista, vaan ainoastaan käyrän muodosta
  - vakiokertoimilla ei ole merkitystä
- siitä huolimatta kertovat yleensä oikein, mikä on ja ei ole nopeaa isoilla n
  - O voi olla liian pessimistinen
  - joidenkin algoritmien edut tulevat esiin vasta epärealistisen isoilla syötteillä

4.3 Vertailemiseen perustuvan järjestämisen nopeuden alaraja

Hitaimman tapauksen alaraja

- olkoot alkioiden avaimet $A[i].x$ keskenään erisuuria, missä $1 \leq i \leq n$
- järjestäminen on sen permutaation etsimistä, joka asettaa alkiot suuruusjärjestykseen
  - **permutaatio** on järjestyksen vaihtaminen (tai jättäminen ennalleen)
- n erisuurella alkiolla on $n!$ permutaatiota (tai järjestystä)
  - 0:lla alkiolla on vain yksi järjestys
  - jos $i > 0$, niin i:s alkio voidaan lisätä jo mukana olevien alkioiden (joita on $i - 1$ kpl) eteen, perään tai minkä tahansa kahden väliin

  $\Rightarrow$ voidaan lisätä i eri paikkaan

  $\Rightarrow$ i:llä alkiolla on $1 \cdot 2 \cdot \ldots \cdot (i-1) \cdot i = i!$ järjestystä
- monet järjestämisalgoritmit saavat tietoa oikeasta järjestyksestä vain vertaamalla kahta alkiota
  - esim. **if** $A[i].x < A[j].x$ **then** ...
  - alkio saattaa olla muussa muuttujassa, mutta on taulukosta peräisin
- yksi vertaaminen jakaa permutaatiot kahteen joukkoon
  - niihin, joissa tulos on ”ei”
  - niihin, joissa tulos on ”kyllä”

- vertailuja pitää tehdä niin monta, että alun perin $n!$ järjestystä sisältäneestä joukosta on erotettu yhden alkion (= oikea järjestys) joukko
- jos jako ei mene tasan, voi olla, että tavoite on suuremmassa puolikkaassa
  - jaossa joukon koon ei voida luvata pienenevän enempää kuin puolittuvan
- niiden puolitusten määrä, jotka muuttavat luvun $m \geq 1$ luvuksi 1, on $\lceil \log_2 m \rceil$

$\Rightarrow$ järjestämiseen tarvitaan enimmillään ainakin $\lceil \log_2 n! \rceil$ vertailua

- $n! \geq (\frac{n}{2})^{n/2}$, joten $\log_2 n! \geq \frac{n}{2} \log_2 \frac{n}{2} = \frac{1}{2} n \log_2 n - \frac{1}{2} n$
  - esim. $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \geq (2\frac{1}{2})^{2\frac{1}{2}}$ ja $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \geq 3^3$

$\Rightarrow$ hitaimmillaan suoritusaika on $\Omega(n \log n)$

Keskimääräisen tapauksen alaraja

- olkoon jokainen alkuperäinen järjestys yhtä todennäköinen
- jos jako ei mene tasan, niin toisessa haarassa tarvitaan ehkä enemmän ja toisessa ehkä vähemmän kuin $\lceil \log_2 n! \rceil$ vertailua

$\Rightarrow$ voidaanko selvitä keskimäärin alle $\lceil \log_2 n! \rceil$ vertailulla?

- luvussa 8.2 johdettava Shannonin alaraja soveltuu tähän tilanteeseen

$\Rightarrow$ keskiarvo ei voi olla parempi kuin $\log_2 n!$ vertailua
  - $\lceil$ ja $\rceil$ poistuivat

$\Rightarrow$ suoritusajan alaraja $\Omega(n \log n)$ koskee myös keskimääräistä tapausta

Toisaalta $O(n \log n)$ riittää

- esim. heapsort

$\Rightarrow$ vertailemiseen perustuva järjestäminen onnistuu hitaimmillaan (ja keskimäärin) ajassa $\Theta(n \log n)$

Counting-sort alittaa tämän rajan

- se ei perustu vertailemiseen
- raja alittuu vain, jos $M \in o(n \log n)$
  - toisaalta on oltava $M \geq n$, jotta kaikki alkiot voisivat olla erisuuria
  - mahdollista, mutta tuskin kovin yleistä

Radix-sort ei alita rajaa

- jotta kaikki alkiot voisivat olla erisuuria, on oltava $k^d \geq n$
  - k on erilaisten numeromerkkien määrä

$\Rightarrow nd \;\geq\; n \log_k n \;=\; n \frac{1}{\log k} \log n \;=\; \Theta(n \log n)$

Keskimääräinen raja on alhaisempi, jos alkiot saavat olla usein yhtäsuuria

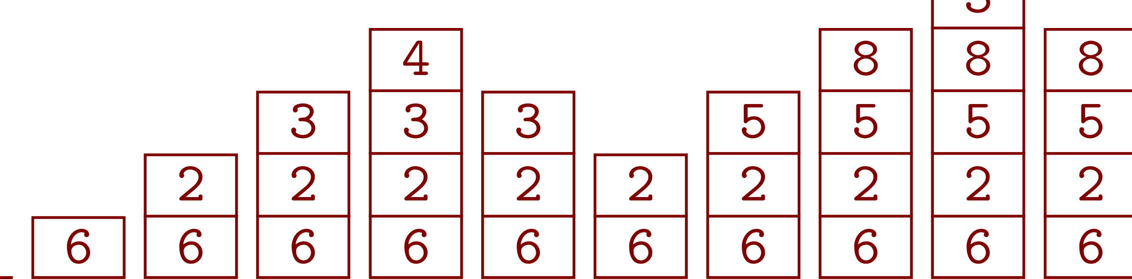
Samankaltaisia alarajoja tunnetaan vain vähän

- ym. alarajasta voidaan johtaa vertailemiseen perustuvalle liukuvalle mediaanille $\Omega(\log n)$ / näyte
- luvussa 10 johdetaan toisella periaatteella tärkeitä alarajoja

5 Tärkeimmät tietorakenteet

5.1 Pino, jono ja listat

Pino on ehkä kaikkein yksinkertaisin tietotyyppi



Pinon toiminnot

- olkoon $\mathbb{T}$ alkioiden tyyppi
- Push(x) lisää alkion x pinon päälle
- Pop $\mapsto \mathbb{T}$ poistaa alkion pinon päältä ja palauttaa sen arvon
  - tuottaa virhetilanteen, jos pino on tyhjä
  - vaihtoehtoisesti Top $\mapsto \mathbb{T}$ palauttaa päällimmäisen arvon ja Pop poistaa päällimmäisen pinosta
- Empty $\mapsto \{\text{false}, \text{true}\}$ kertoo, onko pino tyhjä
- voi olla muitakin toimintoja, kuten
  - Full $\mapsto \{\text{false}, \text{true}\}$ kertoo, onko pino täysi (muisti loppu)
  - Size $\mapsto \mathbb{N}$ kertoo pinossa olevien alkioiden määrän
  - Clear tyhjentää pinon

Edellä kerrottiin tuotettu palvelu kertomatta toteutusta

- **tietotyyppi** (**data type**) kuvaa palvelun
- **tietorakenne** (**data structure**) kuvaa toteutuksen
- **abstrakti tietotyyppi** kuvaa palvelun viittaamatta toteutukseen

Pino taulukolla

- tietorakenteena $A[0 \ldots c-1]$ ja n, alussa $n = 0$
  - **koko** (**size**) n: pinossa olevien alkioiden määrä
  - **kapasiteetti** (**capacity**) c: kuinka monelle alkiolle on varattu tilaa
- toimintojen toteutukset

Push(x)	**if** $n \geq c$ **then error else** $A[n] := x$; $n := n + 1$
Pop	**if** $n \leq 0$ **then error else** $n := n - 1$; **return** $A[n]$
Empty	**return** $n = 0$

- Push(x) voi muistin loppuessa vaihtoehtoisesti varata isomman taulukon ja siirtää alkiot sinne
  - jos käytetään vain ym. toimintoja, ei osoittimien vanhentuminen haittaa
- ym. toteutusten yksityiskohdat valittiin mahdollisimman tehokkaiksi
  - ajankulutukset ovat pieniä, joten tehoerot voivat olla suhteessa suuria
  - tällä on merkitystä, jos iso osa ohjelman suorituksesta on pinon käyttöä
  - muulloinkaan ei ole ainakaan vahingoksi valita paras toteutus

Pino listalla

- tietorakenteena yksisuuntainen linkitetty lista
  - tietueissa kentät x (data) ja s (osoitin seuraavaan)
  - e osoittaa listan ensimmäiseen, alussa $e = \perp$
- toimintojen toteutukset

Push(x)	**new**(p); $p\uparrow.x := x$; $p\uparrow.s := e$; $e := p$
Pop	**if** $e = \perp$ **then error**
	else $x := e\uparrow.x$; $p := e$; $e := e\uparrow.s$; $p\uparrow.s := \perp$; **del**(p); **return** x
Empty	**return** $e = \perp$

- erityisen kätevä silloin, kun toiseen tietorakenteeseen pitää lisätä pinotoiminto
  - samoja tietueita käsitellään sekä pinon että toisen rakenteen mukaan
  - tietue voidaan merkitä pinon ulkopuoliseksi sijoittamalla $p\uparrow.s := p$

Muistin kierrätys vapaiden listalla

- käyttöjärjestelmän **del**(p) voi olla jossain määrin tehoton
- pinotietueiden välinen muistin kierrätys on helppoa ja tehokasta järjestää itse

new'(p)	**if** $v = \perp$ **then new**(p) **else** $p := v$; $v := v\uparrow.s$; $p\uparrow.s := \perp$
del'(p)	$p\uparrow.s := v$; $v := p$

- riskinä on, että vapaa muisti kertyy yhden rakenteen vapaiden listaan mutta tarvittaisiin toisaalla

Toteutusten vertailu

- taulukkototeutus saattaa olla hieman nopeampi, mutta listakin on nopea
- listatoteutus vie paljon ylimääräistä muistia,
  jos alkiot ovat pieniä verrattuina osoittimiin
- taulukkototeutus vie paljon ylimääräistä muistia, jos
  - varataan aivan tarpeettoman iso taulukko tai
  - taulukkoa joudutaan kasvattamaan
- lista sallii saman tietueen yhtäaikaa monessa rakenteessa
  - silloin pino-osoitin käyttää muistia myös kun tietue ei ole pinossa

Jonon toiminnot

- InsQ(x) lisää alkion x jonon jatkeeksi
- DelQ $\mapsto \mathbb{T}$ poistaa jonon ensimmäisen alkion ja palauttaa sen arvon
  - tuottaa virhetilanteen, jos jono on tyhjä
  - vaihtoehtoisesti First palauttaa ja DelQ poistaa ensimmäisen alkion
  - erillisistä First ja DelQ voi tehdä yhdistetyn DelQ, mutta ei toisinpäin
- Empty $\mapsto$ {false, true} kertoo, onko jono tyhjä
- voi olla muitakin toimintoja, kuten Full, Size ja Clear

Jono taulukolla

- tietorakenteena $A[0 \dots c-1]$, v ja n
  - alussa $v = 0$ ja $n = 0$
- toimintojen toteutukset

InsQ(x)	**if** $n \geq c$ **then error else** $n := n + 1;\ A[(v+n) \bmod c] := x$
DelQ	**if** $n \leq 0$ **then error else** $n := n - 1;\ v := (v+1) \bmod c$; **return** $A[v]$
Empty	**return** $n = 0$

- $\bmod c$ saa aikaan sen, että lokeron $A[c-1]$ jälkeen käytetään $A[0]$
  - **rengaspuskuri** (**ring buffer**)
- v indeksoi viimeksi palautettua
  $\Rightarrow$ DelQ ei tarvitse **return** A[monimutkainen]
- jos n:n sijaan olisi indeksi uusimpaan alkioon (tai vanhimpaan vapaaseen), niin tyhjä ja täysi taulukko eivät erottuisi toisistaan
  $\Rightarrow$ puskuriin voitaisiin ottaa enintään $c - 1$ alkiota kerrallaan
- InsQ(x) voi muistin loppuessa vaihtoehtoisesti varata isomman taulukon ja siirtää alkiot sinne

Jono listalla

- osoitin v jonon viimeiseen tietueeseen
- joko on myös osoitin jonon ensimmäiseen tietueeseen, ...
- ... tai viimeinen tietue voi osoittaa ensimmäiseen
  - lisäys epätyhjään **new**(p); $p\uparrow.x := x$; $p\uparrow.s := v\uparrow.s$; $v\uparrow.s := p$; $v := p$
  - poisto epätyhjästä $p := v\uparrow.s$; $v\uparrow.s := p\uparrow.s$; $x := p\uparrow.x$; **del**(p)

Kaksisuuntainen linkitetty lista

- kaksisuuntaisen listan tietueessa on osoittimet edeltäjään e ja seuraajaan s

$\Rightarrow$ lisäys ja poisto saadaan vakioaikaisiksi alkuun, loppuun ja keskelle
  - keskelle lisättäessä/poistettaessa tarvitaan osoitin ko. kohtaan
  - poisto onnistuu vakioajassa, jos on osoitin *poistettavaan* (yksisuuntaisessa tarvitaan osoitin *poistettavan* ***eteen***)
- lisäys tietueen k perään
  $p\uparrow.s := k\uparrow.s$; $p\uparrow.e := k$; $k\uparrow.s := p$; **if** $p\uparrow.s \neq \bot$ **then** $p\uparrow.s\uparrow.e := p$
- kaksisuuntainenkin lista voidaan haluttaessa kääntää renkaaksi

Loppumerkki (**sentinel**)

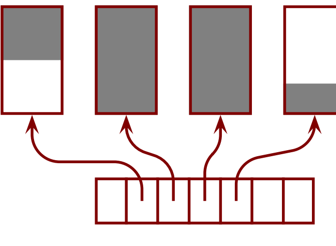
- tietue, joka on listassa aina eikä sisällä hyötykuormaa
- yksinkertaistaa lisäämistä ja poistamista listan päistä

Tasattu vakioaika (amortized constant time)

- yksittäisen toiminnon ei luvata olevan $O(1)$, vaan se saattaa kestää kauan
- n toimintoa pitkän, tyhjästä rakenteesta alkavan toimintojen jonon luvataan olevan $\Theta(n)$
- ennen hidasta toimintoa on niin paljon vakioaikaisia toimintoja, että niiden yhteydessä voidaan ”säästää etukäteen” hitaan toiminnon kuluttama liika aika
  - kertaalleen säästetyn saa kuluttaa vain kerran
- eri asia kuin keskimääräinen aika
  - keskimääräinen aika lasketaan vaihtoehtoisten syötteiden yli
  - tasattu aika lasketaan monen toiminnon yli saman suorituksen aikana

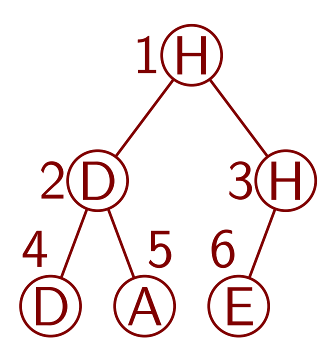
C++:n deque

- molemmista päistään kasvava ja kutistuva pino/jono
  - lisääminen ja poisto kumpaankin päähän ovat tasatusti vakioaikaisia
  - lisätä ja poistaa voi myös keskeltä, mutta ne ovat lineaariaikaisia
  - kapasiteetti voi kasvaa alkioiden säilyessä alkuperäisillä paikoillaan
- tietorakenne
  - riittävästi vakiokokoisia taulukoita
  - kantataulukko, jossa indeksejä tietotaulukoihin



- kirjallisuudesta (jopa C++-standardista) puuttuu usein ”tasatusti”
  - huomattiin vasta myöhään, että ajateltu toteutus ei takaa vakioaikaa

5.2 Keko ja prioriteettijono



Keko

- taulukko $A[1 \dots c]$, jolle $A[i \operatorname{div} 2].x \geq A[i].x$ kun $2 \leq i \leq n$
  - kapasiteetti c, käytössä $A[1 \dots n]$
- voidaan ajatella puuna, jossa indeksiä i vastaavan solmun lapset ovat indekseissä $2i$ (jos $2i \leq n$) ja $2i + 1$ (jos $2i + 1 \leq n$)
- lisääminen ja suurimman alkion poistaminen ovat $O(\log n)$
- taulukon voi muuntaa keoksi ajassa $\Theta(n)$

Lisääminen kekoon (olettaen, että kapasiteettia riittää)

- taulukon jatkeeksi lisätään aukko
- se nostetaan lisättävän arvon mukaiseen kohtaan
  - kukin nousussa kohdattu alkio siirretään aukon edelliseen paikkaan
- lisättävä arvo laitetaan aukkoon

```
n := n + 1
i := n; j := i div 2
while j > 0 && A[j].x < uusi.x do
    A[i] := A[j]
    i := j; j := i div 2
A[i] := uusi
```

Poistaminen epätyhjästä keosta

- taulukon ensimmäinen poistetaan
- taulukon viimeiselle etsitään uusi paikka taulukon ensimmäisestä alaspäin
- alaspäin mennään suuntaan, josta saadaan isompi nostettavaksi aukon paikalle

$tulos := A[1]$
$i := 1$; $j := 2$; **if** $j + 1 < n$ && $A[j+1].x > A[j].x$ **then** $j := j + 1$
while $j < n$ && $A[j].x > A[n].x$ **do**
  $A[i] := A[j]$
  $i := j$; $j := 2 \cdot i$; **if** $j + 1 < n$ && $A[j+1].x > A[j].x$ **then** $j := j + 1$
$A[i] := A[n]$
$n := n - 1$; **return** $tulos$

1(H)
2(D) 3(H)
4 (D) 5 (A) 6 (E)

Taulukon muuntaminen keoksi

for $k := n \operatorname{div} 2$ **downto** 1 **do**
  $apu := A[k]$
  $i := k$; $j := 2 \cdot i$; **if** $j + 1 < n$ && $A[j+1].x > A[j].x$ **then** $j := j + 1$
  tähän epätyhjästä keosta poistamisen **while**-silmukka apu $A[n]$:n tilalla
  $A[i] := apu$

- vain pieni osa alkioista valuu alas pitkän matkan
- jos $2^{\ell-1} \leq n < 2^{\ell}$, niin valumista $\leq 1 \cdot 2^{\ell-2} + 2 \cdot 2^{\ell-3} + \ldots + (\ell - 1) \cdot 2^0 \;=:\; x$

$\Rightarrow 2x - x \;=\; 2^{\ell-1} + \ldots + 2^1 - (\ell - 1) \;=\; 2^{\ell} - \ell - 1 \;\Rightarrow$ suoritusaika on $\Theta(n)$

Heapsort

- taulukko voidaan järjestää $O(n \log n)$ ajassa seuraavasti:
  - muunnetaan se keoksi
  - otetaan toistuvasti suurin pois ja sijoitetaan keon lopusta vapautuvaan aukkoon kunnes keossa on enää yksi alkio
- tämä on eräs parhaimpia tunnettuja järjestämisalgoritmeja
  - hitaimmillaankin $O(n \log n)$
  - tarvitsee vain $O(1)$ lisämuistia
  - ei ole vakaa, mutta ei ole Quicksortkaan
  - on ollut mittauksissa Quicksortia hitaampi

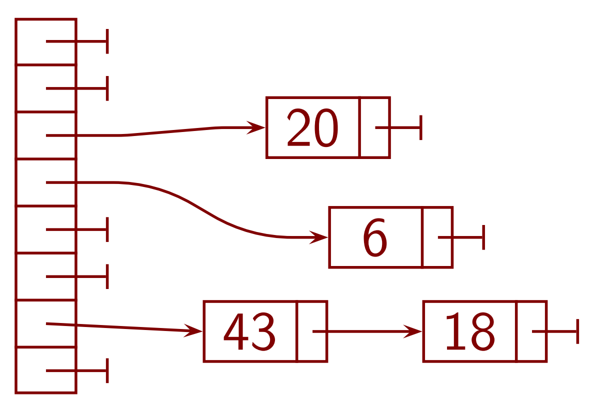
Prioriteettijono

- tietorakenne, johon voi tehokkaasti lisätä alkioita ja poistaa tärkeimmän
  - tärkein voi tarkoittaa pienintä, suurinta, kiireisintä, ...
- käyttöesimerkki: tehtävien suoritusjärjestyksen valinta
  - tehtäviä tulee ulkopuolelta ennakoimattomina ajanhetkinä
  - tehtävän suorittaminen vie yhden aikayksikön
  - jokaisella tehtävällä on määräaika
  - aina seuraavaksi tehdään se tehtävä, jonka määräaika on lähinnä (jos ylipäänsä on mahdollista tehdä kaikki ajoissa, se onnistuu näin)
- keko tarjoaa prioriteettijonon toiminnot ajassa $O(\log n)$

Monissa prioriteettijonojen sovelluksissa olisi eduksi voida muuttaa alkiota jonossa

- keon korjaaminen alkion muututtua sujuisi kuten edellä
- ongelmana on löytää muutettava alkio taulukosta
  - se voi olla siirtynyt alkuperäiseltä paikaltaan
- tähän esitetään ratkaisu luvussa 6.5

5.3 Hajautustaulut



Hajautustaulun (hash table) tuottama palvelu

- seuraavat toiminnot ovat keskimäärin vakioaikaisia:
  - etsiminen avaimen perusteella
  - lisääminen
  - poistaminen
- hajautustaulu on yleensä tehokkain ratkaisu, jos ym. toiminnot riittävät
- selaaminen avaimen mukaisessa järjestyksessä ei onnistu tehokkaasti

Ketjutetun hajautustaulun rakenne

- olkoon M luonnollinen luku
  - ei ainakaan paljon pienempi kuin maksimi talletettavien alkioiden määrä
- **hajautusfunktio (hash function)** h : avainten tyyppi $\mapsto \{0, 1, \ldots, M-1\}$
- taulukko $H[0, \ldots, M-1]$ osoittimia linkitettyihin listoihin
  - jokaisella alkiolla on oma tietue, jossa on avain ja muut tiedot
  - jos alkion avain on x, niin tietue sijaitsee listassa $H[h(x)]$
  - alkioiden järjestyksellä listan sisällä ei ole väliä
- $h(x)$:n tulee tuottaa mahdollisimman tasaisesti kaikkia arvoja 0, ..., $M-1$
  - tärkeintä on, että mihinkään listaan ei kasaannu useita tietueita
  - ei saa olla esim. niin, että $h(x)$ on nimen viimeinen kirjain

Ketjutetun hajautustaulun suorituskyky

- ajan kulutus
  - hitaimmillaan toiminto selaa yhden listan kokonaan
  - listan keskimääräinen pituus on $\frac{n}{M}$
  - M valitaan esim. siten, että $\frac{n}{M} \leq 2$

  $\Rightarrow$ toiminnon *keskimääräinen* ajan kulutus on $O(1)$
- muistin kulutus
  - hyötykuorman lisäksi tarvitaan $n + M$ osoitinta
  - kun $n \approx M$, se on $\approx$ 2 osoitinta / alkio
  - kun n on pieni, on $n + M$ osoitinta suhteessa paljon

$\Rightarrow$ tehdään myös hajautustauluja, joiden M kahdentuu kun $n \approx M$
  - $h(x)$ on esim. muotoa $h'(x) \bmod M$

  $\Rightarrow$ M:n kahdentuessa noin puolet alkioista säilyy listassa $h'(x) \bmod M$ ja puolet siirtyy listaan $M + h'(x) \bmod M$

Listojen pituuksien jakaumasta

- kun $\frac{n}{M} = \ell$ on vakio ja $n \to \infty$, niin eripituisten listojen osuudet $\to e^{-\ell}\frac{\ell^k}{k!}$

ℓ	pituus	0	1	2	3	4	5	6
1	osuus %	37	37	18	6	2	0	0
2	osuus %	14	27	27	18	9	4	1

- nämä likiarvot ovat päteviä jo kun $n = 1000$

$\Rightarrow$ täysin tasaista pituusjakaumaa ei kannata odottaa

Ketjutettu hajautustaulu mahdollistaa myös monen avaimen käytön

- jokaiselle avaimelle oma taulukko H ja tietueissa oma seuraajaosoitin
- esim. auton rekisterinumeron mukaan ja omistajan nimen mukaan

Avoin osoitus

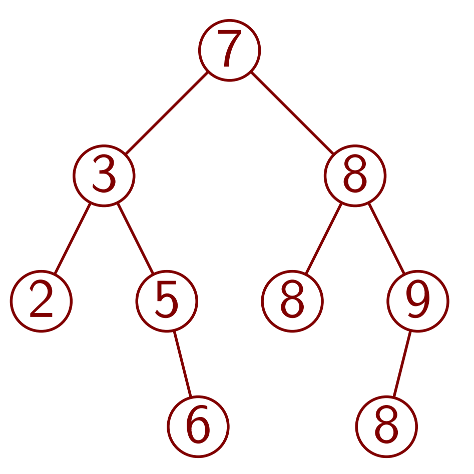
- tiedot ovat suoraan taulukossa H
  - listoja ei ole
  - $\frac{n}{M} < 1$
- jos alkiota lisättäessä $H[h(x)]$ on varattu, yritetään seuraavaksi esim. indeksejä $(h(x) + 1) \bmod M$, $(h(x) + 3) \bmod M$, $(h(x) + 6) \bmod M$, ...
- alkiota poistettaessa tietuetta ei saa merkitä vapaaksi vaan tyhjäksi
  - muutoin ko. paikkaan yritetty mutta muualle mennyt ei jatkossa löydy

$\Rightarrow$ ketjutettu on yleensä kätevämpi kuin avoin osoitus

Bloom-suodatin

- erittäin vähämuistinen tietorakenne joukon likimääräiseksi esittämiseksi
  - jäsenyystesti tuottaa ”varmasti ei” tai ”ehkä kyllä”
  - alkioita voidaan lisätä
  - perusmuotoisesta Bloom-suodattimesta ei voi poistaa alkioita
- väärän ”kyllä” todennäköisyys saadaan pieneksi
- esim. sanojen talletus oikolukuohjelmaa varten
  - väärä ”kyllä” tarkoittaa, että väärin kirjoitettu sana jää havaitsematta
- rakenne
  - bittitaulukko $H[0 \ldots M-1]$
  - k hajautusfunktiota $h_1(x), \ldots, h_k(x)$
  - x talletetaan asettamalla jokainen $h_i(x)$ ykköseksi
- esim. 100 000 sanaa, 1 000 000 bittiä ja 7 hajautusfunktiota
  - bitti jää nollaksi todennäköisyydellä $(1-10^{-6})^{700\,000} \approx e^{-0{,}7} \approx$ 49,7 %
  - $\Rightarrow$ väärän ”kyllä” todennäköisyys $\approx (1-0{,}497)^7 \approx$ 0,8 %
  - käytetään vain 10 bittiä eli 1,25 tavua sanaa kohti!

5.4 Binäärihakupuut



Rakenne ja käsitteitä

- jokaisella solmulla on enintään kaksi **lasta** (**child**)
  - **vasen** (**left**) ja **oikea** (**right**)
  - toisin kuin keossa, voi olla oikea lapsi ilman vasenta
- solmussa saattaa olla myös osoitin **isään** (**parent**)
- koko puun ylin solmu on **juuri** (**root**)
  - juurella ei ole isää
  - binäärihakupuut piirretään sanastoon nähden ylösalaisin!
- lapseton solmu on **lehti** (**leaf**)
  - muut solmut ovat **sisäsolmuja** (**internal node**)
- solmu, sen lapset, niiden lapset jne. muodostavat **alipuun** (**subtree**)
- solmun
  - vasemmassa alipuussa avaimet ovat $\leq$ solmun avain
  - oikeassa alipuussa avaimet ovat $\geq$ solmun avain
- puun **korkeus** (**height**) on kaarten määrä juuresta kaukaisimpaan lehteen

Etsiminen avaimen perusteella

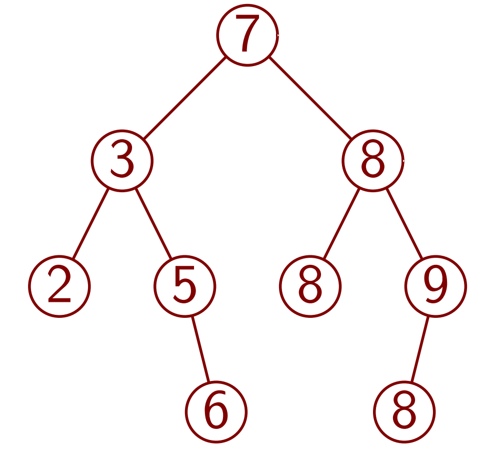
while $p \neq \bot$ && $p{\uparrow}.x \neq x$ **do**
  if $p{\uparrow}.x < x$ **then** $p := p{\uparrow}.o$ **else** $p := p{\uparrow}.v$

Rekursiivinen läpikäynti järjestyksessä

selaa(p) **if** $p \neq \perp$ **then** selaa($p\uparrow.v$); print($p\uparrow.x$); selaa($p\uparrow.o$)

Seuraajan etsiminen ilman rekursiota isäosoittimien avulla

if $p\uparrow.o \neq \perp$ **then**
  $p := p\uparrow.o$
  while $p\uparrow.v \neq \perp$ **do** $p := p\uparrow.v$
else
  $q := p$; $p := p\uparrow.i$
  while $p \neq \perp$ && $q = p\uparrow.o$ **do** $q := p$; $p := p\uparrow.i$



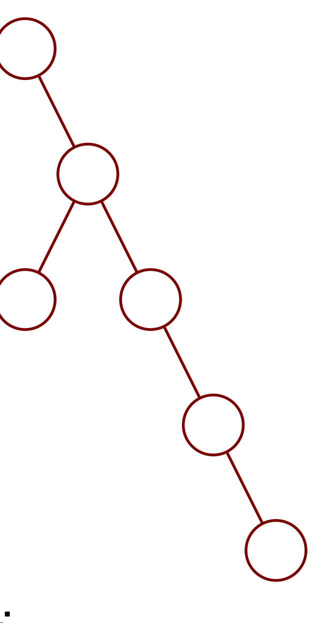
Lisääminen epätyhjään puuhun esimerkiksi näin:

$q := \perp$
while $q = \perp$ **do**
  if $p\uparrow.v = \perp \wedge x \leq p\uparrow.x$ **then new**(q); $p\uparrow.v := q$
  else if $p\uparrow.o = \perp \wedge x \geq p\uparrow.x$ **then new**(q); $p\uparrow.o := q$
  else if $p\uparrow.x < x$ **then** $p := p\uparrow.o$ **else** $p := p\uparrow.v$
$q\uparrow.i := p$; $q\uparrow.v := \perp$; $q\uparrow.o := \perp$; $q\uparrow.x := x$

Poistaminen on monimutkaista

- jos poistettavalla solmulla on lapsia, joudutaan siirtämään solmu sen tilalle
- onnistuu puun korkeuteen verrannollisessa ajassa

Ajan kulutus



- rekursiivinen läpikäynti on $\Theta(n)$
- etsiminen avaimen perusteella, seuraajan etsiminen, lisääminen ja poistaminen ovat $O(h)$
  - h on puun korkeus
  - koko puun selaaminen seuraajatoiminnolla on $\Theta(n)$
- jos puu on hyvin tasapainossa, niin $O(h)$ on $O(\log n)$
- valitettavasti puu voi olla hyvin epätasapainoinen
  - lisättävät avaimet eivät välttämättä tule satunnaisessa järjestyksessä

Tasapainotetut (**balanced**) binäärihakupuut

- on kehitetty tekniikoita, joilla voidaan varmistaa esim., että pisin polku juuresta lehteen on korkeintaan kaksi kertaa niin pitkä kuin lyhyin
  $\Rightarrow h \leq 2\log_2 n$

$\Rightarrow$ etsiminen, lisäys ja poisto onnistuvat ajassa $O(\log n)$

- hitaampaa kuin hajautustauluilla, mutta
  - tasap. binäärihakupuut tarjoavat myös nopean selaamisen järjestyksessä
  - tasap. binäärihakupuiden aikalupaus on taattu eikä keskimääräinen

Nykyisin ehkä yleisin tasapainotettu binäärihakupuu on ns. puna-musta puu

- rakenne
  - jokainen solmu on musta tai punainen (solmussa on bitti väriä varten)
  - juuri on musta
  - punaisella solmulla ei ole punaisia lapsia
  - jokaisella polulla juuresta ⊥-osoittimeen on sama määrä mustia solmuja
- lisäyksen ja poiston yhteydessä solmuja siirrellään ja väritetään uudelleen
  - monimutkaista, mutta tehokasta
- puna-musta puu on mahdollinen toteutus C++:n säiliöille `map` yms.

Puun täydentäminen

- puna-mustan puun solmuihin voidaan lisätä tietoja suorituskyvyn kärsimättä
  - riittää, että solmun lisätieto on laskettavissa solmun ja sen lasten alkuperäisistä tiedoista sekä lasten lisätiedoista
- esim. lisätään sen alipuun koko, jonka juuri solmu on

⇒ voidaan $O(\log n)$ ajassa
  - laskea, monesko solmu on suuruusjärjestyksessä
  - etsiä suuruusjärjestyksessä halutun mones solmu

5.5 Mitä tietorakenteeseen laitetaan?

Tietorakenteet ajatellaan usein mustina laatikoina, jotka sisältävät tietoalkioita

- tietoalkioita
  - lisätään
  - etsitään avaimen perusteella
  - etsitään sijainnin perusteella (esim. ensimmäinen, seuraava ja viimeinen), missä sijainti voi määräytyä talletusajasta, avaimen suuruudesta, . . .
  - käsitellään kahvan, indeksin, osoittimen tai iteraattorin avulla
  - poistetaan
- pannaanko tietorakenteeseen varsinainen tieto vai esim. osoitin siihen?
- osoittimen etuja
  - samaa tietoa ei ole hyvä tallettaa moneen kertaan

  $\Rightarrow$ jos samaa tietoa tarvitaan muuallakin, niin mieluummin osoitin
  - osoittimen tapauksessa ei tarvitse kopioida isoja tietomääriä
- osoittimen takana oleva tieto voi muuttua tietorakenteen huomaamatta

  $\Rightarrow$ avaimiin perustuva tietorakenteen sisäinen järjestys voi rikkoutua
- esim. binäärihakupuussa on osoittimia ja osoitettu avain muuttuu

  $\Rightarrow$ muuttunut alkio ei löydy uudella eikä vanhalla avaimella, ja se harhauttaa osan kauttansa kulkevista hauista väärään suuntaan

Mustat laatikot rajoittavat eri tietorakenteiden hyvien puolten yhdistämistä

- esimerkki: edustajiston jäseniä halutaan
  - lisätä
  - etsiä nimen perusteella
  - poistaa nimen perusteella (muualle muuttaneen poistaminen)
  - poistaa jäsenyysaikajärjestyksessä (erovuorossa olevien poistaminen)
- hajautustaulu (ja binääripuu) sallivat tehokkaasti muun, mutta ei aikajärjestyksessä poistamista
- aikajärjestys voitaisiin ylläpitää erillisellä jonolla, mutta miten nimen perusteella poistettava poistetaan jonosta?

Sama tietoalkio voidaan laittaa osaksi monta osoittimiin perustuvaa rakennetta

- voi esim. olla hajautustaulussa ja 2-suuntaisessa linkitetyssä listassa
  - tietueessa on sekä hajautustaulun että listan edellyttämät linkit
  - avaimella etsittäessä löydetään hajautustaulun avulla
  - aikajärjestyksen perusteella etsittäessä löydetään listan kautta
  - listan kautta löydetyn avain saadaan tarvittaessa alkiosta itsestään

  $\Rightarrow$ miten tahansa löydetty alkio voidaan poistaa molemmista rakenteista
- tietorakennetta ei ajatella mustana laatikkona, vaan alkion päälle lisättävänä
- tällaisia ratkaisuja ei vielä saa valmiina ohjelmakirjastoista

6 Algoritmien oikeaksi todistaminen

6.1 Silmukkainvariantti

Esimerkki: binääripotenssi

- nopea tapa laskea a^n (tai $a^n \bmod 2^M$), kun n on suuri luonnollinen luku
  - salausjärjestelmissä lasketaan potenssilaskuja modulo 2^M isoilla n
  - $x := a \cdot a \cdot \ldots \cdot a$ käyttää $n-1$ kertolaskua
  - binääripotenssi käyttää enintään $2\lceil \log_2(n+1) \rceil$ kertolaskua, nopeus $\Theta(\log n)$
  - $\bmod 2$ on alimman bitin katsominen ja $\operatorname{div} 2$ on >> 1

  $\Rightarrow$ hyvin nopeita

```
x := 1
while n > 0 do
    if n mod 2 = 1 then x := a · x
    a := a · a; n := n div 2
```

- sovimme, että $a^0 = 1$ jokaisella a, myös kun $a = 0$
  - joskus näkee väitettävän, että 0^0 ei ole määritelty
  - käytäntö $0^0 = 1$ on kuitenkin hyvin yleinen, esim. $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$
- miksi yo. ohjelma laskee oikein?

Silmukkainvariantti (loop invariant)

- kaava, joka on voimassa aina kun tullaan silmukan kierroksen alkuun
- binääripotenssin todistamisessa käytetään silmukkainvarianttia

$$xa^n = a_0^{n_0} \wedge n \geq 0$$

  - a_0 ja n_0 ovat a:n ja n:n arvot ohjelman alussa
  - $n_0 \geq 0$, koska $n_0 \in \mathbb{N}$
- merkitsemme nämä tiedot ohjelmakoodiin

$\{\, a = a_0 \wedge n = n_0 \geq 0 \,\}$
$x := 1$
$\{\, xa^n = a_0^{n_0} \wedge n \geq 0 \,\}$
while $n > 0$ **do**
  $\{\, xa^n = a_0^{n_0} \wedge n > 0 \,\}$
  if $n \bmod 2 = 1$ **then** $x := a \cdot x$
  $a := a \cdot a;\; n := n \operatorname{div} 2$
  $\{\, xa^n = a_0^{n_0} \wedge n \geq 0 \,\}$

- $a = a_0 \wedge n = n_0$ ei tee muuta kuin antaa a:n ja n:n alkuarvoille nimet, joilla niistä voi puhua sen jälkeenkin, kun a:n ja n:n arvot ovat muuttuneet
- silmukkainvarianttiin liittyen on osoitettava kolme asiaa:
  - se pätee, kun silmukkaan tullaan sitä edeltävästä lauseesta
  - se säilyy voimassa, kun silmukkaa kierretään kierros
  - kun silmukan lopetusehto pätee, se antaa luvatun lopputuloksen

Esimerkin silmukkaan tuleminen edeltävästä lauseesta

- osoitettava $\{\, a = a_0 \wedge n = n_0 \geq 0 \,\}\ x := 1\ \{\, xa^n = a_0^{n_0} \wedge n \geq 0 \,\}$
- koska $a = a_0$, $n = n_0$ ja $x = 1$, on $xa^n = 1 \cdot a_0^{n_0} = a_0^{n_0}$; $n \geq 0$ on ilmeinen ·/·

Kierros esimerkin silmukassa

- n:n parilliset ja parittomat arvot hallitaan helpommin lisäämällä ohjelmaan lause $n := n - 1$, joka ei vaikuta sen toimintaan
  - suoritetaan vain kun n on pariton $\Rightarrow$ ei muuta jakolaskun $n \operatorname{div} 2$ tulosta
  - n:n arvoa ei käytetä lisätyn $n := n - 1$ ja jakolaskun $n \operatorname{div} 2$ välissä

while $n > 0$ **do**
$\{\, xa^n = a_0^{n_0} \wedge n > 0 \,\}$
if $n \bmod 2 = 1$ **then**
$\{\, n \bmod 2 = 1 \wedge (ax)a^{n-1} = xa^n = a_0^{n_0} \wedge n > 0 \,\}$
$x := a \cdot x$; $n := n - 1$
$\{\, n \bmod 2 = 0 \wedge xa^n = a_0^{n_0} \wedge n \geq 0 \,\}$
$\{\, n \bmod 2 = 0 \wedge x(a^2)^{n/2} = xa^n = a_0^{n_0} \wedge n \geq 0 \,\}$
$a := a \cdot a$; $n := n \operatorname{div} 2$
$\{\, xa^n = a_0^{n_0} \wedge n \geq 0 \,\}$

- kaavojen pätevyys on melko helppo nähdä edellisistä kaavoista
  - jos **then**-haara ohitetaan, niin $n \bmod 2 = 0$

$\Rightarrow$ silmukan kierros säilyttää $xa^n = a_0^{n_0} \wedge n \geq 0$ voimassa ·/·

Esimerkin silmukasta poistuminen

- silmukasta poistumisen ehto on $\neg(n > 0)$ eli $n \leq 0$
- se ja $n \geq 0$ antaa $n = 0$
- se ja $xa^n = a_0^{n_0}$ antaa $xa^n = xa^0 = x \cdot 1 = x$, joten $x = a_0^{n_0}$

Koko todistus

$$
\begin{array}{l}
\{\, a = a_0 \wedge n = n_0 \geq 0 \,\} \\
x := 1 \\
\{\, xa^n = a_0^{n_0} \wedge n \geq 0 \,\} \\
\textbf{while } n > 0 \textbf{ do} \\
\quad \{\, xa^n = a_0^{n_0} \wedge n > 0 \,\} \\
\quad \textbf{if } n \bmod 2 = 1 \textbf{ then} \\
\qquad \{\, n \bmod 2 = 1 \wedge (ax)a^{n-1} = xa^n = a_0^{n_0} \wedge n > 0 \,\} \\
\qquad x := a \cdot x;\ n := n - 1 \\
\qquad \{\, n \bmod 2 = 0 \wedge xa^n = a_0^{n_0} \wedge n \geq 0 \,\} \\
\quad \{\, n \bmod 2 = 0 \wedge x(a^2)^{n/2} = xa^n = a_0^{n_0} \wedge n \geq 0 \,\} \\
\quad a := a \cdot a;\ n := n \operatorname{div} 2 \\
\quad \{\, xa^n = a_0^{n_0} \wedge n \geq 0 \,\} \\
\{\, xa^n = a_0^{n_0} \wedge n \geq 0 \wedge n \leq 0 \,\} \\
\{\, x = a_0^{n_0} \,\}
\end{array}
$$

$\Rightarrow$ lopussa $x = a_0^{n_0}$ kuten pitääkin

Silmukkainvariantti on siis kaava silmukkainvariantti(tehty), jolle pätee

1. alkutilanne $\Rightarrow$ silmukkainvariantti(ei mitään)
2. jos silmukan vartalon alussa silmukkainvariantti(x) $\wedge\, x \neq$ kaikki, niin silmukan vartalon lopussa silmukkainvariantti(x')
   - pyritään siihen, että x':ssa on tehty enemmän kuin x:ssä
   - tämä ei kuitenkaan ole osa silmukkainvariantin määritelmää
3. silmukkainvariantti(kaikki) $\Rightarrow$ tavoite

- esim. silmukkainvariantin $xa^n = a_0^{n_0}$ tapauksessa
  - a^n edustaa laskematonta ja x laskettua osaa tavoitteesta
  - kun $n = n_0$, mitään ei ole tehty: $x = 1$ ja $a^n = a_0^{n_0}$
  - kun $n = 0$, kaikki on tehty: $x = a_0^{n_0}$ ja $a^n = 1$
  - $n' = n \operatorname{div} 2$
- jos k = esimerkissä tehtyjen kierrosten määrä, niin

  $$x = a_0^{n_0 \bmod 2^k} \text{ ja } a^n = a_0^{2^k(n_0 \operatorname{div} 2^k)}$$

  - nämä kaavat voivat olla vaikea löytää ja hahmottaa
  - mutta ei niitä olisi tarvinnutkaan löytää ja hahmottaa, $xa^n = a_0^{n_0}$ riittää!

Mistä silmukkainvariantti saadaan?

- ohjelmoijan (tai katselmoijan) on keksittävä se!
  - esittää silmukan toiminnan perusajatuksen
  - sitä ei osata tuottaa automaattisesti (varsinkin jos silmukka on väärin)

Mitä binääripotenssi tekisi negatiivisilla eksponenteilla?

- mielekäs kysymys, sillä esim. 2^{-3} on hyvin määritelty
- ehdon $n > 0$ vuoksi lopettaisi heti siten, että $x = 1$
- toiminta ei-negatiivisilla eksponenteilla ei muuttuisi, jos vaihdettaisiin $n > 0 \rightsquigarrow n \neq 0$

$x := 1$
while $n \neq 0$ **do**
  if $n \bmod 2 = 1$ **then** $x := a \cdot x$
  $a := a \cdot a$; $n := n \operatorname{div} 2$

- nyt $xa^n = a_0^{n_0}$ ja loppuehto $\neg(n \neq 0)$ antaisivat suoraan $x = a_0^{n_0}$
- meneekö todistuksessa mikään pieleen?

$\{\, a = a_0 \wedge n = n_0 \,\}$ $x := 1$ $\{\, xa^n = a_0^{n_0} \,\}$
while $n \neq 0$ **do** $\{\, xa^n = a_0^{n_0} \wedge n \neq 0 \,\}$
  if $n \bmod 2 = 1$ **then**
    $\{\, n \bmod 2 = 1 \wedge (ax)a^{n-1} = xa^n = a_0^{n_0} \,\}$
    $x := a \cdot x$; $n := n - 1$
    $\{\, n \bmod 2 = 0 \wedge xa^n = a_0^{n_0} \,\}$
  $\{\, n \bmod 2 = 0 \wedge x(a^2)^{n/2} = xa^n = a_0^{n_0} \,\}$
  $a := a \cdot a$; $n := n \operatorname{div} 2$
  $\{\, xa^n = a_0^{n_0} \,\}$
$\{\, x = a_0^{n_0} \,\}$

- jos $a = 0$, ovat kaikki a:n ja a_0:n negatiiviset potenssit määrittelemättömiä
  - päätelmä $(ax)a^{n-1} = xa^n$ on erityisen epäilyttävä
  - mutta ohjelman ei tarvitsekaan osata laskea 0^n, kun $n < 0$
- jos $a \neq 0$, ovat kaikki päätelmät oikein
- laskeeko siis ohjelma oikein, kun $a \neq 0$ ja $n < 0$?
- **ei, se jää ikuiseen silmukkaan** niin että $n = -1$

Kvantifioiduista kaavoista

- tarvitaan väittämiä muotoa "taulukon $A[1 \ldots n]$ pienin alkio on kohdassa i"
- $\forall j : A[i] \leq A[j]$ ei sano sitä oikein, koska j saa myös laittomia arvoja
- pätevän kaavan tulee myös vaatia, että myös i on laillisella alueella
- $1 \leq i \leq n \land \forall j : j < 1 \lor j > n$ || $A[i] \leq A[j]$ on hankala hahmottaa
- $1 \leq i \leq n \land \forall j : 1 \leq j \leq n \rightarrow A[i] \leq A[j]$ on parempi, mutta tarvittaisiin $\leftrightarrows$
- indeksien rajaus kvantifioinnissa on kuitenkin niin yleinen tarve, että on hyödyksi antaa sille oma merkintätapa
  - $\forall i; \varphi : \psi$ tarkoittaa samaa kuin $\forall i : \neg\varphi$ || ψ
  - $\exists i; \varphi : \psi$ tarkoittaa samaa kuin $\exists i : \varphi$ && ψ

$\Rightarrow$ ei tarvitse huolehtia sulkujen lisäämisestä (jos esim. ψ on $P \lor Q$) eikä siitä, miten rajaus yhdistyy muuhun kaavaan

- esim. $1 \leq i \leq n \land \forall j; 1 \leq j \leq n : A[i] \leq A[j]$

6.2 Loppuun pääseminen

Lopettaako ohjelma lopulta on toisinaan erittäin vaikea ongelma

- kukaan ei tiedä, lopettaako seuraava ohjelma jokaisella $n \in \mathbb{N}$

  while $n > 1$ **do**
      if $n \bmod 2 = 0$ **then** $n := n/2$ **else** $n := 3n + 1$

  – arvaus, että lopettaa, tunnetaan Collatz'n otaksumana (conjecture)
- kukaan ei tiedä, lopettaako seuraava ohjelma

  $n := 6$; $p := 3$
  while $2p \leq n$ **do**
      if on_alkuluku(p) $\wedge$ on_alkuluku($n - p$) **then** $n := n + 2$; $p := 3$
      else $p := p + 2$

  – Goldbach'n otaksuma
- asiaan palataan luvussa 10

Siksi ohjelman oikeellisuus jaetaan kahteen osaan

1. **osittainen oikeellisuus** (**partial correctness**):
   jos ohjelma lopettaa, on tulos oikein
2. **lopettaminen** (**termination**): ohjelma lopettaa

- molemmat yhdessä on **täysi oikeellisuus** (**total correctness**)

Ohjelma voi välttää lopettamisen

- jäämällä ikuiseen silmukkaan
  - esim. $i := 1$; **while** $i > 0$ **do** $i := i + 1$
- kutsumalla loputtomasti aliohjelmaa rekursiivisesti
  - esim. kertoma(-1)

    kertoma$(n) \mapsto \mathbb{N}$

    if $n = 0$ **then return** 1
    else return $n \cdot$ kertoma$(n - 1)$
- ohjelmointikielessä saattaa olla muitakin tapoja

Silmukan lopettaminen todistetaan usein antamalla jokin suure, joka

1. vähenee silmukan jokaisella kierroksella
   - jos silmukan ehto toteutuu, niin silmukan vartalon suoritus vähentää sitä
2. on **well-founded**: ei ole olemassa loputtomasti silmukan ehdon toteuttavia arvoja x_i siten, että $x_1 > x_2 > x_3 > \ldots$
   - sopisikohan suomennokseksi **pohjallinen**?
   - ”hyvinjärjestys” ei käy, koska ”$<$” ei välttämättä ole täysi järjestys

$\Rightarrow$ suure vähenee lopulta arvoon, joka ei toteuta silmukan ehtoa

- suureelle ei ole vakiintunutta nimeä

- joskus käytetään
  - **loop variant**
  - **bound function**, kun arvo on luonnollinen luku

Usein esiintyvä erikoistapaus: ylärajaa kohti kasvava tai alarajaa kohti vähenevä kokonaisluku

- olennaisesti sama asia
  - jos i kasvaa kohti ylärajaa y, niin $y - i$ vähenee kohti alarajaa 0
- **for** $i := 1$ **to** n **do** ... on ilmeinen, *kunhan varmistetaan, että silmukassa ei kasvateta n:ää eikä pienennetä i:tä*
- binääripotenssissa n vähenee joka kierroksella kunnes $n \leq 0$
- muunnetussa binääripotenssissa, kun $n = -1$, se **ei kasva** kohti nollaa
- myös jäljellä olevan syötteen määrä on tällainen suure
  - vähenee aina kun syötettä luetaan

  $\Rightarrow$ pitää huolehtia, että syötteenkäsittelysilmukassa *jokaisessa tapauksessa* todella luetaan syötettä tai ilmoitetaan, että syöte loppui

Yleisempi tapaus on toisinaan kätevä

- esim. joissakin graafialgoritmeissa silmukan kierroksella

  joko solmujen määrä v vähenee (kaarten määrä voi kasvaa)

  tai kaarten määrä e vähenee ja solmujen määrä säilyy ennallaan

- pari (v, e) toteuttaa vaatimukset
  - $(v, e) < (v', e')$ jos ja vain jos $v < v'$ tai $v = v' \wedge e < e'$
  - jos $(v_1, e_1) > (v_2, e_2) > \ldots$, niin v lakkaa lopulta vähenemästä, ja sen jälkeen e voi vähentyä vain äärellisen monesti
- sanojen tuttu aakkosjärjestys ei kumpaankaan suuntaan toteuta vaatimusta
  - `a` $<$ `aa` $<$ `aaa` $< \ldots$
  - `b` $>$ `ab` $>$ `aab` $> \ldots$

Rekursion tms. lopettaminen voidaan todistaa samaan tapaan kuin silmukan

Joskus ”mahdoton” ohjelma voidaan osoittaa osittain oikeaksi

- ristiriita vältetään sillä, että ohjelma ei lopeta
- esim. jakolasku $d := n \operatorname{div} m$ peräkkäisinä vähennyslaskuina

$$\{\, n = n_0 \geq 0 \,\}\ d := 0\ \{\, n_0 = dm + n \wedge 0 \leq n \,\}$$
$$\textbf{while } n \geq m \textbf{ do } n := n - m;\ d := d + 1\ \{\, n_0 = dm + n \wedge 0 \leq n \,\}$$
$$\{\, n_0 = dm + n \wedge 0 \leq n < m \,\}$$

  - kaavat ovat oikein kaikilla n ja m
  - nollalla ei voi jakaa eikä voi olla $0 \leq n < 0$, joten mitä ohjelma tekee, jos $m = 0$?
  - jää ikuiseen silmukkaan **while** $n \geq 0$ **do** $n := n - 0;\ d := d + 1$

6.3 Heikoin esiehto

Voi tuntua hankalalta ja virhealttiilta päätellä sijoituslauseiden vaikutuksia, kuten

- $\{\, xa^n = a_0^{n_0} \,\}\ x := a \cdot x\ \{\, xa^{n-1} = a_0^{n_0} \,\}\ n := n - 1\ \{\, xa^n = a_0^{n_0} \,\}$
- $\{\, n \bmod 2 = 0 \wedge xa^n = a_0^{n_0} \,\}\ a := a \cdot a\ \{\, xa^{n \operatorname{div} 2} = a_0^{n_0} \,\}$
  $n := n \operatorname{div} 2\ \{\, xa^n = a_0^{n_0} \,\}$
- $\{\, n_0 = dm + n \wedge n < 0 \,\}\ n := n + m;\ d := d - 1\ \{\, n_0 = dm + n \wedge n < m \,\}$

Siihen on kuitenkin hyvä ”takaperoinen” keino!

- tarkastellaan $\{\, P \,\}\ x :=$ lauseke $\{\, Q \,\}$
- sijoitetaan Q:ssa lauseke x:n tilalle
  - merkitään tämän tulosta lauseke $\rightsquigarrow x$ @ Q
- osoitetaan, että $P \Rightarrow$ lauseke $\rightsquigarrow x$ @ Q

Edelliset esimerkit

- $n - 1 \rightsquigarrow n$ @ $xa^n = a_0^{n_0}$ tuottaa $xa^{n-1} = a_0^{n_0}$
  $a \cdot x \rightsquigarrow x$ @ $xa^{n-1} = a_0^{n_0}$ tuottaa $(ax)a^{n-1} = a_0^{n_0}$ eli $xa^n = a_0^{n_0}$ $\cdot/\cdot$
- $n \operatorname{div} 2 \rightsquigarrow n$ @ $xa^n = a_0^{n_0}$ tuottaa $xa^{n \operatorname{div} 2} = a_0^{n_0}$
  $a \cdot a \rightsquigarrow a$ @ $xa^{n \operatorname{div} 2} = a_0^{n_0}$ tuottaa $x(aa)^{n \operatorname{div} 2} = a_0^{n_0}$
  ja $n \bmod 2 = 0 \Rightarrow n = 2(n \operatorname{div} 2)$,
  joten $n \bmod 2 = 0 \wedge xa^n = a_0^{n_0} \Rightarrow xa^{2(n \operatorname{div} 2)} = x(aa)^{n \operatorname{div} 2} = a_0^{n_0}$ $\cdot/\cdot$

- $d-1 \rightsquigarrow d$ @ $n_0 = dm + n \wedge n < m$ tuottaa $n_0 = (d-1)m + n \wedge n < m$
  $n + m \rightsquigarrow n$ @ $n_0 = (d-1)m + n \wedge n < m$ tuottaa
  $n_0 = (d-1)m + (n+m) \wedge n + m < m$ eli $n_0 = dm + n \wedge n < 0$ ·/·

Keino laskee sijoituslauseen **heikoimman esiehdon** (**weakest precondition**)

- lauseen S ja väitteen Q heikoin esiehto $wp(S, Q)$ on heikoin väite, joka takaa, että lauseen suorittaminen onnistuu ja lopputulos takaa Q:n
- esim. $wp(x := m/n,\ x = 1)$ on $m = n \neq 0$, koska nollalla ei saa jakaa
- tyhjän lauseen heikoin esiehto Q:n suhteen on tietenkin Q itse
  - esim. puuttuva **else**-haara on tyhjä lause
- lauseen **if** C **then** S_1 **else** S_2 heikoin esiehto Q:n suhteen on

$$C\Uparrow \wedge (C \wedge wp(S_1, Q) \vee \neg C \wedge wp(S_2, Q)) \ ,$$

  missä $C\Uparrow$ tarkoittaa, että C:n laskeminen onnistuu
  - yhtäpitävä vaihtoehtoinen muoto

$$C\Uparrow \wedge (\neg C \vee wp(S_1, Q)) \wedge (C \vee wp(S_2, Q))$$

Laskujen eteneminen takaperin on pieni ongelma, koska

- silmukkainvariantista voi laskea takaperin pitkin silmukan vartaloa
- voidaan lähteä takaperin ohjelman halutusta lopputuloksesta

Lisää esimerkkejä

- $wp(d := 0,\ n_0 = dm + n)$ on $n_0 = 0m + n$ eli $n_0 = n$
- $wp(x := 2x + 1,\ x \leq n)$ on $2x + 1 \leq n$ eli $x \leq \frac{n-1}{2}$
- $wp(x := y - 1,\ x = 3)$ on $y - 1 = 3$ eli $y = 4$
- $wp(x := y - 1,\ y = 3)$ on $y = 3$
- $wp(x := y - 1,\ x = y)$ on $y - 1 = y$ eli false
- $wp(x := x + 1,\ \text{true})$ on true
- $wp(x := 1/0,\ \text{true})$ on false
- $wp(x := 1/n,\ \text{true})$ on $n \neq 0$

Kuten ohjelmoinnissa aina, nytkin pitää ottaa huomioon, että erinäköiset vasen-arvot voivat tarkoittaa samaa muistipaikkaa

- esim. $wp(A[i] := 1,\ A[2] = 0)$
  - ei ole $A[2] = 0$
  - eikä $1 \leq i \leq n \wedge 2 \leq n \wedge A[2] = 0$
  - vaan $1 \leq i \leq n \wedge 2 \leq n \wedge i \neq 2$ && $A[2] = 0$

6.4 Esimerkki: puolitushaku

Puolitushaku (binary search) on nopea keino löytää alkio järjestetystä taulukosta

- $\Theta(\log n)$
- löytää ensimmäisen kohdan, jossa on vähintään etsityn suuruinen
  - vastaus $n+1$ tarkoittaa, että etsitty on suurempi kuin mikään taulukossa
  - ts. lopussa a on pienin luku siten, että $A[a] \geq x$, jos sellainen on olemassa, ja muutoin lopussa $a = n+1$
- haarukoi väliä, jossa vastaus on

```
a := 1; y := n + 1                          { a ≤ y }
while a < y do
    v := (a + y) div 2                      { a ≤ v < y }
    if A[v] < x then a := v + 1             { a ≤ y ∧ A[a − 1] < x }
    else y := v                             { a ≤ y ∧ A[y] ≥ x }
    { a ≤ y }
{ a = y ∧ (a = 1 || A[a − 1] < x) ∧ (y = n + 1 || A[y] ≥ x) }
```

- koko ajan $1 \leq a \leq y \leq n+1$
  - selkeyden vuoksi $1 \leq a$ ja $y \leq n+1$ on jätetty merkitsemättä ohjelmassa
  - a ei koskaan pienene eikä y kasva
  - joka kierroksella a kasvaa tai y pienenee

  $\Rightarrow$ ohjelma lopettaa

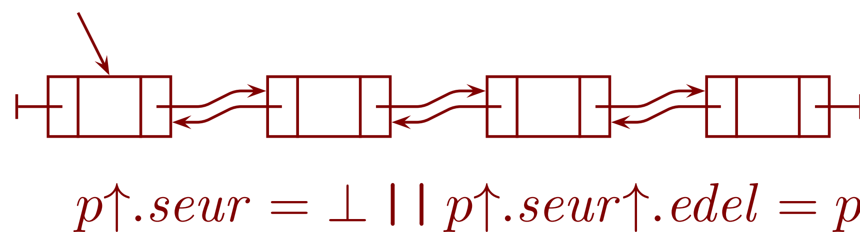
- a ei koskaan ole liian suuri
  - tarkemmin: jokainen alkio, joka on a:sta vasemmalle, on liian pieni
  - jos $a = 1$, niin vasemmalla ei ole mitään
  - muutoin on testillä todettu, että heti a:sta vasemmalle on liian pieni
- y ei koskaan ole liian pieni
  - tarkemmin: y:n kohdalla eikä siitä oikealle ole liian pientä
  - jos $y = n + 1$, niin y:n kohdalla ja siitä oikealle ei ole mitään
  - muutoin on testillä todettu, että y:n kohdalla on tarpeeksi suuri
- lopussa $a = y$
  - a ei ole liian pieni eikä liian suuri

  $\Rightarrow$ a on oikea vastaus
- lopussa $(a = 1 \text{ || } A[a-1] < x) \wedge (a = n + 1 \text{ || } A[a] \geq x)$
  vaikka A ei olisi järjestyksessä!
  - jos A on järjestyksessä, niin on täsmälleen yksi arvo, joka toteuttaa kaavan
  - jos A on epäjärjestyksessä ja x on taulukossa, niin puolitushaku voi löytää kaavan toteuttavan arvon, jonka kohdalla x ei ole
  - esim. A = [2,1,3], x = 2, a = 3

6.5 Tietorakenteen invariantti

Tietorakenteen invariantti on väite, joka on voimassa aina paitsi kesken päivityksen

- esim. kaksisuuntainen linkitetty lista



Iso esimerkki: motivaatio

- tavallinen keon avulla toteutettu prioriteettijono mahdollistaa tehokkaasti vain lisäyksen, ensimmäisen katsomisen ja ensimmäisen poistamisen
  - $O(\log n)$
- esim. Dijkstran lyhimpien polkujen algoritmi hyötyy kyvystä muuttaa avainta alkion ollessa prioriteettijonossa

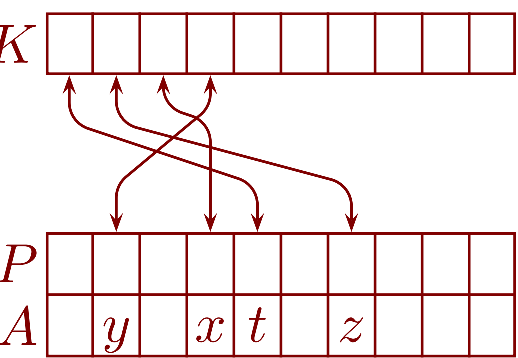
$\Rightarrow$ esitämme prioriteettijonon, josta voidaan poistaa myös keskeltä
  - muuttaminen saadaan poistamalla ja sitten lisäämällä uudella avaimella
  - (olisi helppo toteuttaa suoraan ilman poistamistakin)

Miten poistettava kerrotaan?

- ei avaimen avulla, koska
  - edellyttää kykyä löytää tehokkaasti avaimen avulla
  - $\Rightarrow$ tietorakenne monimutkaistuisi
  - hitaampaa kuin valittu tapa

⇒ käytetään **kahvaa (handle)**
  - osoitin, indeksi tms. jolla tieto löydetään välittömästi
  - C++:n selain (iterator) on suunnilleen sama asia

- lisäys ottaa alkion ja palauttaa kahvan
- poisto ottaa kahvan
- kahvan arvo 0 kertoo, että alkiota ei ole
  - pienimmän alkion kahva, jos prioriteettijono on tyhjä
  - lisäyksen tuottama kahva, jos tilaa ei ollut



Toteutuksen tietorakenteet

- taulukko A, jossa alkiot ovat
  - kahvana toimii indeksi taulukkoon
  - alkio pysyy samassa indeksissä sen ajan, jonka on prioriteettijonossa
- taulukko K, joka toimii kekona
  - sisältää alkion sijaan indeksin A:han
  - ⇒ alkioiden vertailu tapahtuu tehokkaasti: $A[K[i]] < A[K[j]]$
- taulukko P, jolla löydetään alkiota vastaava paikka K:ssa
  - $P[k]$ sisältää kahvalle k sen i, jolle $K[i] = k$
  - ts. käytössä olevalle i pätee koko ajan $P[K[i]] = i$
  - P mahdollistaa nopean siirtymisen yhteen ja K vastakkaiseen suuntaan
- taulukot indeksoidaan $1, \ldots, m$

Koko ja kapasiteetti

- poistojen vuoksi käytössä olevien kahvojen joukko ei ole aina yhtenäinen
  - esim. lisää(x) $\mapsto 1$, lisää(y) $\mapsto 2$, lisää(z) $\mapsto 3$, poista(2): $\{1, 3\}$
- olisi tehotonta säätää taulukkojen varaamaa muistia jokaisen lisäyksen ja poiston yhteydessä

$\Rightarrow$ kannattaa erottaa toisistaan koko ja kapasiteetti
  - **koko (size)**: prioriteettijonossa olevien alkioiden määrä
  - **kapasiteetti (capacity)**: kuinka monelle alkiolle on varattu tilaa
- jos maksimikokoa ei tiedetä etukäteen, voidaan käyttää sivun 15 joustavan kokoisia taulukoita
- tallennetaan koko muuttujaan n ja kapasiteetti muuttujaan m

Vapaiden lokeroiden kierrätys

- vapaat kahvat ovat mielivaltaisesti välillä $1, \ldots, m$
  - miten löydetään vapaa kahva?
- K on käytössä kekona välillä $1, \ldots, n$

$\Rightarrow$ vapaat kahvat voi tallettaa K:hon välille $n + 1, \ldots, m$

$\Rightarrow$ uusi alkio x voidaan lisätä näin: $n := n + 1$; $A[K[n]] := x$; $P[K[n]] := n$
ja sitten siirtää oikealle paikalleen keossa

Esimerkin tietorakenteiden invariantit

- koko on ei-negatiivinen ja enintään kapasiteetti: $0 \leq n \leq m$
- K:n arvot ovat laillisia kahvoja: $\forall i; 1 \leq i \leq m : 1 \leq K[i] \leq m$
  - keon arvot ovat laillisia kahvoja: $\forall i; 1 \leq i \leq n : 1 \leq K[i] \leq m$
  - vapaat kahvat ovat laillisia kahvoja: $\forall i; n < i \leq m : 1 \leq K[i] \leq m$
- sama kahva ei ole K:ssa useasti: $\forall i : \forall j; 1 \leq i < j \leq m : K[i] \neq K[j]$
  - jo $P[K[i]] = i$ takaa, että sama kahva ei ole keossa useasti
  - yhdessä edellisen kanssa takaa, että jokainen laillinen kahva on K:ssa
- P kertoo keossa olevan alkion paikan keossa: $\forall i; 1 \leq i \leq n : P[K[i]] = i$
- kannattaisiko vapailta kahvoilta vaatia jotain?
  - vaaditaan niiltä, että ne näyttävät vapailta: $\forall i; n < i \leq m : P[K[i]] = 0$

  $\Rightarrow$ voidaan kätevästi estää yritys poistaa alkio, joka on jo poistettu
- keko toteuttaa keko-ominaisuuden: $\forall i; 1 < i \leq n : A[K[i \operatorname{div} 2]] \leq A[K[i]]$

Esimerkin aliohjelmat (metodit)

<u>alusta</u> // helppo nähdä, että asettaa invariantit

$n := 0$
for $i := 1$ **to** m **do** $P[i] := 0$; $K[i] := i$

<u>pienin $\mapsto$ alkion tyyppi</u> // ei muuta mitään, keon pienin kohdassa 1

if $n > 0$ **then return** $A[K[1]]$ **else error**

lisää(x) $\mapsto \{0, \ldots, m\}$

```
if n ≥ m then return 0                        // älä lisää, jollei ole tilaa
n := n + 1; v := K[n]; A[v] := x              // lisätty vapaaseen paikkaan A:ssa
i := n; j := i div 2                          // kohdassa i on ”aukko” keossa
while j > 0 && x < A[K[j]] do                 // juoksuta se x:n mukaiseen kohtaan
    K[i] := K[j]; P[K[i]] := i                // pidä P linjassa K:n kanssa
    i := j; j := i div 2
K[i] := v; P[v] := i                          // täytä keon aukko x:llä
return v
```

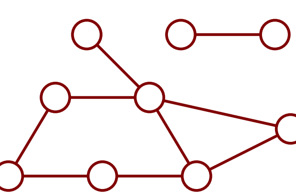
poista(v)

```
if v < 1 ∨ v > m || P[v] = 0 then return      // hylkää laiton kahva
i := P[v]; j := i div 2                       // poistettava on ”aukko” keossa
while j > 0 && A[K[n]] < A[K[j]] do           // siirrä aukko tarpeeksi ylös
    K[i] := K[j]; P[K[i]] := i; i := j; j := i div 2
j := 2 · i
if j + 1 < n && A[K[j + 1]] < A[K[j]] then j := j + 1
while j < n && A[K[n]] > A[K[j]] do           // siirrä aukko tarpeeksi alas
    K[i] := K[j]; P[K[i]] := i; i := j; j := 2 · i
    if j + 1 < n && A[K[j + 1]] < A[K[j]] then j := j + 1
K[i] := K[n]; P[K[i]] := i                    // täytä aukko keon viimeisellä
K[n] := v; P[v] := 0; n := n − 1              // vapauta keon viimeinen lokero
```

7 (Moni)graafit

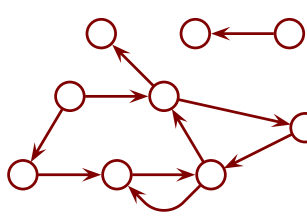
7.1 Monigraafit ja niiden esittäminen

Graafi (graph) on matemaattinen rakenne, jossa on kahdenlaisia osia



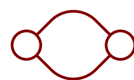
- **solmu (vertex, node)**
  - piirretään usein ympyröinä tai kiekkoina
- **kaari (edge, arc)**
  - piirretään usein viivana tai nuolena

Graafit jakaantuvat kahteen ryhmään



- **suunnattu (directed)** graafi
  - kaari piirretään nuolena solmusta solmuun
- **suuntaamaton (undirected)** graafi
  - kaari piirretään viivana, joka yhdistää kaksi solmua

Monigraafi (multigraph) sallii samojen solmujen välillä (samaan suuntaan) monta kaarta



- useat yleiset esitystavat tietokoneessa sallivat moninkertaiset kaaret, jollei erikseen nähdä vaivaa niiden estämiseksi

$\Rightarrow$ monesti sanotaan graafi, vaikka esitystapa sallii monigraafin

Suunnatun graafin matemaattinen määritelmä

- solmujen joukkoa merkitään yleensä kirjaimella V
- kaarten joukkoa merkitään yleensä kirjaimella E
- suunnatussa graafissa vaaditaan, että $E \subseteq V \times V$
  - ts. kaari on kahden solmun järjestetty pari
  - $V \times V$ sisältää kaikki järjestetyt solmuparit
  - jokaisen solmuparin välillä joko on kaari tai ei ole
  - E kertoo, millä väleillä kaaret ovat
- jos $v \in V$, niin $(v, v) \in V \times V$
  $\Rightarrow$ solmusta voi olla kaari itseensä

Suuntaamattoman graafin matemaattinen määritelmä

- V on kuten edellä
- E:n määritelmän tarkka sisältö ja muotoilu eivät ole vakiintuneet
  - joskus sallitaan kaari solmusta itseensä, joskus ei
- ei sallita: kaari on kahden solmun järjestämätön pari
  - $E \subseteq \{V' \mid V' \subseteq V \wedge |V'| = 2\}$
- sallitaan: kaari on solmu tai kahden solmun järjestämätön pari
  - $E \subseteq \{\{u, v\} \mid u \in V \wedge v \in V\}$

Kytkentämatriisi (adjacency matrix)

- solmut tunnetaan numeroilla $1, \ldots, |V|$
- kaaret on ilmoitettu $|V|$ kertaa $|V|$ -bittitaulukolla
  - solmusta u on kaari solmuun v, jos ja vain jos $B[u,v] = 1$
- suuntaamaton graafi: huolehditaan, että jos $B[u,v] = 1$, niin $B[v,u] = 1$
- tehokas esitystapa, jos graafi on **tiheä (dense)**
  - kaarten määrä ei ole paljon pienempi kuin $|V|^2$
  - ts. melkein jokaisesta solmusta lähtee melkein $|V|$ kaarta
  - "paljon" ja "melkein" riippuvat mm. siitä, kuinka paljon vaihtoehtoinen esitystapa veisi bittejä kaarta kohden
  - esim. $\frac{1}{10}|V|^2$ kaarta on yleensä tiheä, mutta $\frac{1}{1000}|V|^2$ tuskin enää on
  - jos graafi ei ole tiheä, se on **harva (sparse)**
- kaaren voi lisätä ja poistaa ajassa $\Theta(1)$
- solmusta lähtevät tai siitä tulevat kaaret voi selata ajassa $\Theta(|V|)$
- jos kaarissa tarvitaan lisätieto, niin $B[u,v]$ on joko se tai erikoisarvo $\perp$ kertomaan, että kaarta ei ole
  - esim. $B[u,v]$ voi olla kaaren pituus tai -1
  - tiheän ja harvan raja siirtyy lähemmäs lukua $|V|^2$
- jos kaarissa ei tarvita lisätietoa, niin monigraafin saa sallimalla $B[u,v] \in \mathbb{N}$
  - monigraafeja ilman lisätietoa kaarissa tarvitaan harvoin

Kytkentälista (adjacency list)

- esitetään vain olemassa olevat kaaret
  - ts. ei varata muistia parille (u, v), jos se ei ole kaari
- perusmuodossa jokaiseen solmuun liittyy linkitetty lista, jonka kussakin tietueessa on osoitin tai indeksi solmuun
  - ts. luetellaan solmusta lähtevien kaarten määränpäät
- esittää monigraafin, ellei erikseen huolehdita, että moninkertaisia kaaria ei ole
- nytkin suuntaamaton graafi esitetään suunnattuna, jossa $(u, v) \in E \Rightarrow (v, u) \in E$
- muistin kulutus on $\Theta(|V| + |E|)$
- solmusta lähtevien kaarten selaaminen on $O(|E|)$
  - verrannollinen solmusta lähtevien kaarten määrään
- kaaren lisääminen ja poistaminen vievät aikaa solmusta lähtevien kaarten määrään verrannollisesti
  - jos moninkertaisia kaaria ei yritetä välttää eikä listan järjestyksellä ole väliä, niin lisääminen on $\Theta(1)$
- kaaritietueeseen voidaan laittaa lisätietoa tarpeen mukaan
- solmuun tulevien kaarten selaamista varten solmuun voidaan liittää toinen lista
  - se voi käyttää samoja tai erillisiä listatietueita

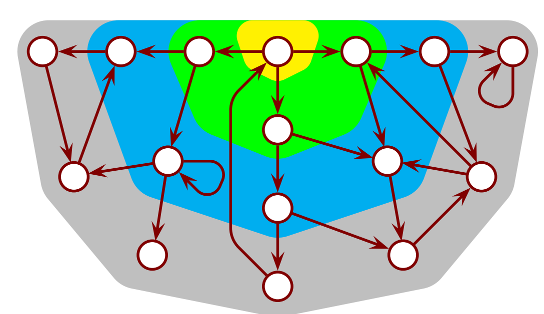
Kaaret taulukossa

- jos kaaria ei tarvitse lisätä, kaikki kaaret voi luetella yhdessä taulukossa
- samasta solmusta lähtevät kaaret ovat peräkkäin taulukossa
- solmutietueessa on kerrottu, mikä osa taulukosta sisältää sen lähtökaaret
  - esim. ensimmäisen lähtökaaren indeksi ja lähtökaarten määrä
  - jälkimmäistä ei tarvita, jos kunkin solmun v kaarten jälkeen on solmun $v + 1$ kaaret
- taulukon alkiona on tarpeen mukaan
  - kaaritietue
  - kahva (esim. osoitin tai indeksi) kaaritietueeseen
  - kahva kaaren toisessa päässä olevaan solmuun
- muistin käytön kannalta erityisen tehokas ratkaisu

Esitystavan valinta riippuu monesta seikasta

- onko (moni)graafi harva vai tiheä
- minkä toimintojen tarvitsee olla nopeita
  - muutetaanko (moni)graafia käytön aikana
  - tarvitseeko käsitellä solmun tulokaaria
  - tarvitaanko nopea vastaus kysymykseen ”onko kaarta (u, v)”
- tarvitseeko muistia käyttää erityisen tehokkaasti

7.2 Läpikäyntialgoritmeja



Merkitään $u\bullet = \{v \mid (u, v) \in E\}$

- solmusta u lähtevien kaarten kärkipäissä olevat solmut

Leveyteen ensin -haku (breadth-first search)

for $v \in V$ **do** $v.found$:= false
for $v_0 \in V$ **do** // harmaa osa pois, jos halutaan käydä
  if $\neg v_0.found$ **then** // läpi vain v_0:sta saavutettava alue
    $v_0.found$:= true; Q.InsQ(v_0); $v_0.prev := \perp$ // aluksi Q on tyhjä
    while $\neg Q$.Empty **do**
      u := Q.DelQ
      for $v \in u\bullet$ **do**
        if $\neg v.found$ **then**
          $v.found$:= true; Q.InsQ(v); $v.prev := u$

- $.found$ huolehtii, että samaa solmua ei laiteta Q:hun useasti
  $\Rightarrow$ **while**-silmukalla $\leq |Q|$ kierrosta
  $\Rightarrow$ jos InsQ, DelQ, Empty sekä seuraavan $v \in V$ ja $v \in u\bullet$ löytäminen ovat $O(1)$, niin algoritmi lopettaa ajassa $O(|V| + |E|)$
- **while**-silmukan invariantti: kaikille u pätee, että jos $u.found$ = true, niin u on jonossa tai $\forall v \in u\bullet : v.found$ = true

- algoritmin lopetettua kaikille $(u, v) \in E$ pätee, että
  jos $u.found$ = true niin $v.found$ = true
  - olkoon $(v_0, v_1) \in E$, $(v_1, v_2) \in E$, ..., $(v_{n-1}, v_n) \in E$
  - koska $v_0.found$ = true ja $(v_0, v_1) \in E$, niin $v_1.found$ = true
  - koska $v_1.found$ = true ja $(v_1, v_2) \in E$, niin $v_2.found$ = true

  $\Rightarrow \ldots v_n.found$ = true

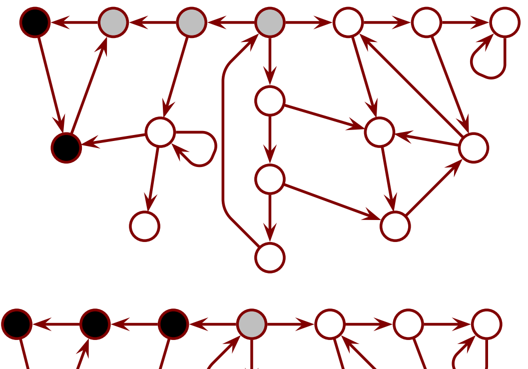
$\Rightarrow$ algoritmi löytää kaikki aloituskohdastaan / aloituskohdistaan v_0 saavutettavat solmut ja vain ne
  - "ja vain ne" seuraa siitä, että algoritmi löytää solmuja vain tietyillä tavoilla

Leveyteen ensin -haun erityisominaisuus

- olkoon $\delta(u, v)$ lyhimmän solmusta u solmuun v vievän polun pituus
  - polun pituus on sen kaarten määrä (ts. jokaisen kaaren pituus on 1)
- joka hetki on voimassa jollekin $k \in \mathbb{N}$
  - ne solmut on löydetty, joille $\delta(v_0, v) \leq k$
  - yhtään niistä ei ole löydetty, joille $\delta(v_0, v) > k + 1$
  - jonon alkuosassa $\delta(v_0, v) = k$ ja sen jälkeen $\delta(v_0, v) = k + 1$ (toinen tai molemmat osat voivat olla tyhjiä)

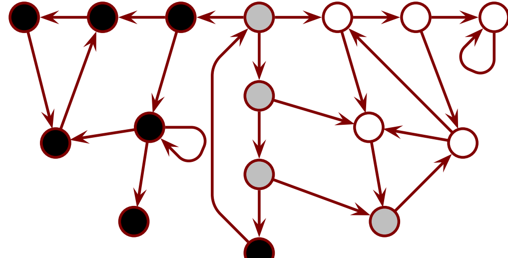
$\Rightarrow$ löytää lyhimmän polun kuhunkin solmuun
  - polku takaperin löytyy $.prev$-osoittimia seuraamalla
  - muista: tässä polun pituus on kaarten määrä eikä kuten maantiekartassa

Rekursiivinen **syvyyteen ensin -haku (depth-first search)**



for $v \in V$ **do** $v.colour$:= white
for $v_0 \in V$ **do if** $v_0.colour =$ white **then**
DFS(v_0) // harmaa osa pois, jos ...

DFS(u)



$u.colour$:= grey
for $v \in u\bullet$ **do if** $v.colour =$ white **then** DFS(v)
$u.colour$:= black

- solmun väri muuttuu valkoinen $\rightarrow$ harmaa $\rightarrow$ musta
- väri huolehtii, että samaa solmua ei käsitellä useasti
  $\Rightarrow$ algoritmi lopettaa ajassa $O(|V| + |E|)$ (olettaen, että $v \in V$ ja $v \in u\bullet \ldots$)
- harmaat solmut muodostavat polun haun alkusolmusta nykyiseen solmuun
- mustalla solmulla ei ole valkoista seuraajaa
- kun algoritmi lopettaa, mikään solmu ei ole harmaa
  $\Rightarrow$ mustista solmuista pääsee vain mustiin solmuihin
  $\Rightarrow$ kuten edellä, voidaan päätellä, että
  algoritmi löytää kaikki solmuista v_0 saavutettavat solmut ja vain ne
- rekursiivinen syvyyteen ensin -haku on helppo toteuttaa, mutta rekursion luontainen tehottomuus voi muodostua ongelmaksi isoilla graafeilla

Syvyyteen ensin -haun erityisominaisuus

- jos nykyisestä solmusta u on kaari harmaaseen solmuun v, niin graafissa on silmukka
  - ts. jos kutsutaan DFS(v) niin että $v.colour =$ grey
  - silloin v:stä on polku harmaita solmuja pitkin u:hun
- jos tutkittavalla alueella on silmukoita, niin *jokin* niistä löytyy
  - *algoritmi ei välttämättä löydä silmukan solmuja silmukan järjestyksessä*
  - olkoon u silmukan solmuista se, josta algoritmi peruuttaa ensimmäisenä
  - olkoon v seuraava silmukan solmu
  - ennen u:sta peruuttamista algoritmi tutki kaaren (u, v)
  - jos v oli silloin musta, niin u ei olisikaan silmukan solmuista se, josta algoritmi peruuttaa ensimmäisenä
  - jos v oli silloin valkoinen, niin algoritmi olisi mennyt sinne ja peruuttanut sieltä, joten u ei olisikaan . . .

  $\Rightarrow$ v oli silloin harmaa

$\Rightarrow$ tutkittavalla alueella on silmukoita jos ja vain jos ainakin kerran tutkittavan kaaren kärkipään solmu on harmaa

- kaikkien silmukoiden löytäminen ei olisi mielekäs tehtävä
  - niitä on usein aivan liikaa lueteltavaksi (harjoitustehtävä)
  - mielekkäämpää on etsiä maksimaalisia vahvasti kytkettyjä komponentteja (asiaan palataan)

Yleinen haku

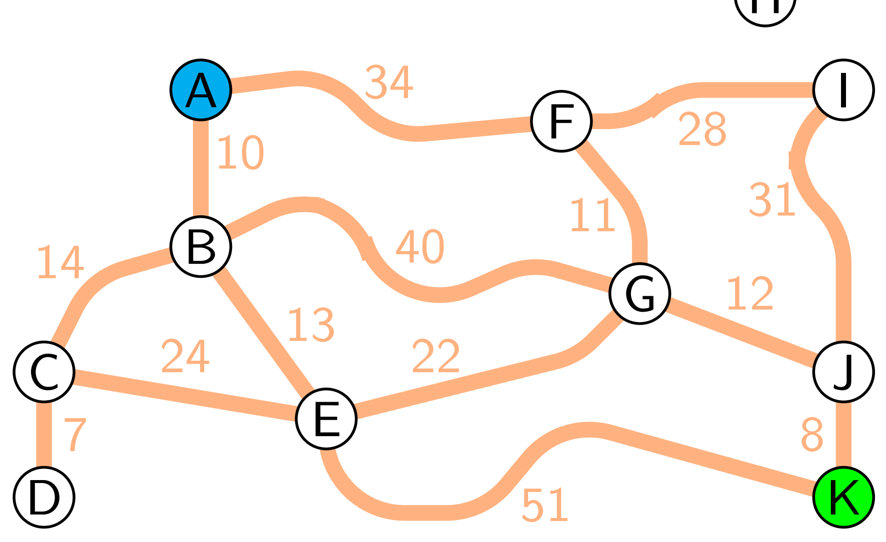
for $v \in V$ **do** $v.found$:= false
for $v_0 \in V$ **do if** $\neg v_0.found$ **then** // harmaa osa pois, jos ...
  $v_0.found$:= true; $v_0.prev := \perp$
  if $v_0\bullet \neq \emptyset$ **then**
    $v_0.next := v_0\bullet$; $W := \{v_0\}$
    while $W \neq \emptyset$ **do**
      u := jokin W:n alkio
      v := jokin $u.next$:n alkio; $u.next := u.next \setminus \{v\}$
      if $u.next = \emptyset$ **then** $W := W \setminus \{u\}$
      if $\neg v.found$ **then**
        $v.found$:= true; $v.prev := u$
        if $v\bullet \neq \emptyset$ **then** $v.next := v\bullet$; $W := W \cup \{v\}$

- $.found$ huolehtii, että kukin solmu lähtökaarineen tutkitaan enintään kerran
- W sisältää (vain) löydetyt solmut, joilla on tutkimattomia lähtökaaria
- algoritmi löytää kaikki v_0:sta saavutettavat solmut ja vain ne
  - todistus kuten leveyteen ensin -haulle
- jos W on jono, niin tämä on leveyteen ensin -haku
- jos W on pino, niin tämä on syvyyteen ensin -haku
- jos $.next$-mekanismin sijaan W:hen laitetaan $u\bullet$, niin ei saada syvyyteen ensin
  - voi silti olla käyttökelpoinen: vältetään $.next$ ja pino on helppo toteuttaa

7.3 Lyhimmät polut

Tarkastellaan graafeja, joiden kaaret ovat muotoa (u, d, v)

- d on kaaren pituus, $0 \leq d < \infty$
- merkitään $u\bullet = \{(d, v) \mid (u, d, v) \in E\}$



Dijkstran algoritmi

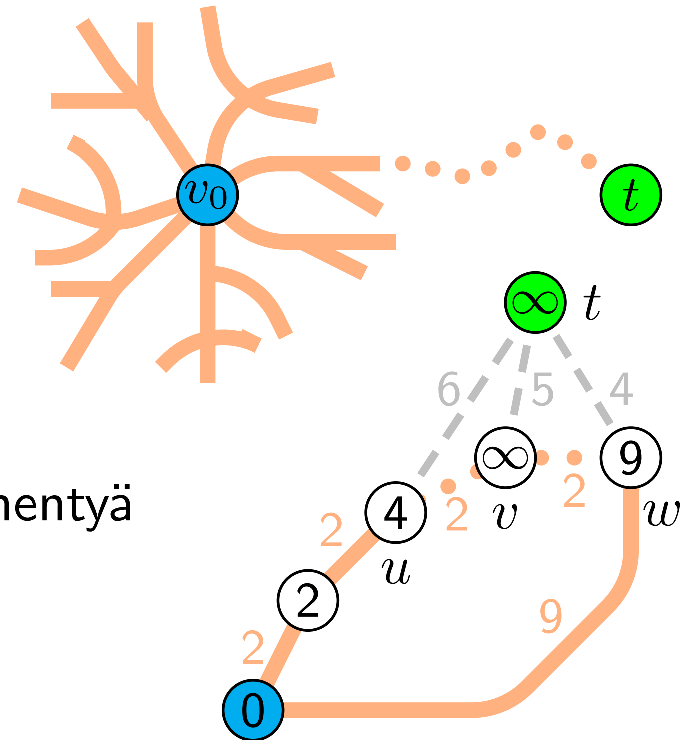
> **for** $v \in V$ **do** $v.dist := \infty$
> $Q := \{v_0\}$; $v_0.dist := 0$; $v_0.prev := \bot$
> **while** $Q \neq \emptyset$ **do**
> $u := Q$.smallest; $Q := Q \setminus \{u\}$
>   **for** $(d, v) \in u\bullet$ **do**
>   **if** $u.dist + d < v.dist$ **then**
>   **if** $v.dist = \infty$ **then** $Q := Q \cup \{v\}$
> $v.dist := u.dist + d$; $v.prev := u$

- löytää mahdollisimman lyhyet polut v_0:sta kaikkialle
  - ∞ tarkoittaa, että polkua ei ole
  - monesta yhtä lyhyestä löytyy vain jokin, mutta silti sanomme ”lyhimmät”
- Q on prioriteettijono, jonka avain on $.dist$
- v on löydetty jos ja vain jos $v.dist \neq \infty$
- ne lyhimmät polut on löydetty, joiden pituus $< u.dist$

Dijkstran algoritmin suoritusaika

- oletetaan, että muut kuin alla mainitut toiminnot ovat hyvin toteutettu
- luvun 6.5 prioriteettijonolla aika on $O((|V| + |E|)\log|V|)$
  - kukin solmu on Q:ssa enintään kerran, joten $|Q| \leq |V|$
  - solmun lisäys Q:hun ja poisto Q:sta ovat $O(\log|V|)$
  - $v.dist$:n pienentäminen on $O(\log|V|)$ ja tapahtuu $\leq |E|$ kertaa
- tunnetaan Q:n toteutus, jolla ajaksi saadaan $O(|V|\log|V| + |E|)$

A* linnuntie-etäisyydellä



- maantiekartan tapauksessa matka solmusta v tavoitteeseen t on vähintään linnuntie-etäisyys $||t - v|| := \sqrt{(t.x - v.x)^2 + (t.y - v.y)^2}$

$\Rightarrow$ algoritmia voi nopeuttaa käsittelemällä seuraavaksi se v, jolla arvio$(v) := v.dist + ||t - v||$ on pienin

- kun tutkittu haara pitenee, koko matkan arvio ei voi pienentyä
  - ts. $u.dist + ||t - u|| \leq u.dist + \delta(u, v) + ||t - v||$

$\Rightarrow$ jos lyhin reitti solmuun w ei ole vielä löytynyt, on sen tutkitun osan kärjen arvio $<$ w:n arvio

$\Rightarrow$ solmu otetaan Q:sta vasta kun lyhin reitti siihen on löytynyt

- tämä tunnetaan nimellä **A* monotonisella heuristiikalla**

Lyhimmät polut jokaisesta solmusta jokaiseen solmuun

- vastauksia tulee $|V|^2$ kappaletta

⇒ kannattaa käyttää tiheiden graafien tekniikoita, vaikka alkuperäinen graafi olisi harva

- yksi käytännöllinen algoritmi on esitelty luvun 6 harjoitustehtävissä
  - käyttää $\Theta(|V|^3)$ aikaa

Negatiiviset kaarten pituudet

- jos kaaren pituus voi olla negatiivinen, niin voi olla olemassa silmukka, jonka pituus on negatiivinen

⇒ solmusta u solmuun v voi olla loputtomasti toinen toistaan lyhyempiä polkuja

Bellman–Ford-algoritmi

for $v \in V$ **do** $v.dist := \infty$
$v_0.dist := 0$; $v_0.prev := \perp$
for $i := 2$ **to** $|V|$ **do**
  for $(u, d, v) \in E$ **do**
    if $u.dist + d < v.dist$ **then** $v.dist := u.dist + d$; $v.prev := u$
for $(u, d, v) \in E$ **do**
  if $u.dist + d < v.dist$ **then** print "negatiivinen silmukka"

- suoritusaika on $\Theta(|V||E|)$

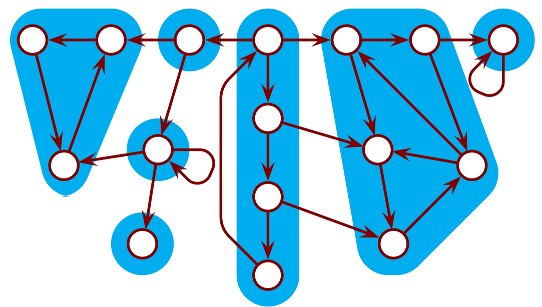
- kierroksen i alussa on löydetty lyhimmät polut, joissa $\leq i-2$ kaarta
- jos v_0:sta saavutettavalla alueella ei ole negatiivisia silmukoita, niin
  - polkua ei saada lyhyemmäksi lisäämällä siihen silmukoiden kierroksia

  $\Rightarrow$ lopuksi on löydetty kaikki lyhimmät polut
- jos algoritmi tulostaa "negatiivinen silmukka", niin sellainen on saavutettavissa
  - jos $v.dist$ on lyhimmän polun mukaan, niin ei voi olla $u.dist + d < v.dist$
- jos negatiivinen silmukka on saavutettavissa, niin algoritmi tulostaa "negatiivinen silmukka"
  - olkoon D silmukan pituus
  - jos silmukan jokaiselle kaarelle (u, d, v) pätee $u.dist + d \geq v.dist$, niin kiertämällä silmukka ympäri saadaan $u.dist + D \geq u.dist$
  - ristiriita, jos $D < 0$

  $\Rightarrow$ jollekin kaarelle pätee $u.dist + d < v.dist$

7.4 Maksimaaliset vahvasti kytketyt komponentit

Suunnatun graafin **vahva komponentti (strong component)**



- merkitään solmuille $u \to^* v$ jos ja vain jos u:sta on polku v:hen
  - myös nolla kaarta katsotaan poluksi

  $\Rightarrow$ jokaiselle solmulle v pätee $v \to^* v$
- solmun v virittämä maksimaalinen vahvasti kytketty komponentti C_v on mahdollisimman iso kokoelma solmuja u siten, että $v \to^* u$ ja $u \to^* v$
  - $v \in C_v$
- usein käytetään lyhyempää nimeä "vahva komponentti" (strong component)
- kukin solmu kuuluu täsmälleen yhteen vahvaan komponenttiin
  - olkoot C_u ja C_v vahvoja komponetteja siten, että $w \in C_u$ ja $w \in C_v$
  - jos $u' \in C_u$, niin $u' \to^* u \to^* w \to^* v$ ja $v \to^* w \to^* u \to^* u'$

  $\Rightarrow u' \to^* v \wedge v \to^* u'$ eli $u' \in C_v$
  - samoin jos $v' \in C_v$, niin $v' \in C_u$

  $\Rightarrow C_u = C_v$
- vahva komponentti on käyttökelpoinen käsite silloinkin, kun silmukoita on liikaa lueteltaviksi

Syvyyteen ensin -haku ja vahvat komponentit

- mustista solmuista pääsee valkoisiin vain harmaiden kautta
- jokaisesta harmaasta solmusta pääsee nykyiseen solmuun

$\Rightarrow$ jos syvyyteen ensin -haku kohtaa vahvan komponentin, se käy sen ja siitä saavutettavat vahvat komponentit kokonaan läpi ennen peruuttamistaan siitä (tai lopettamista)

  – kohtaa vahvan komponentin $=$ löytää siihen kuuluvan solmun

Tarjanin algoritmi

$i := 0$
for $v \in V$ **do** $v.colour$:= white
for $v_0 \in V$ **do if** $v_0.colour$ = white **then** Tarjan(v_0)

Tarjan(u)

$u.colour$:= grey; $i := i + 1$; $u.nr := i$; $u.low := i$; S.Push(u)
for $v \in u\bullet$ **do**
  if $v.colour$ = white **then** Tarjan(v)
  if $v.colour \neq$ blue $\wedge\ v.low < u.low$ **then** $u.low := v.low$
$u.colour$:= black
if $u.low = u.nr$ **then**
  print "vahva komponentti"
  repeat $v := S$.Pop; print v; $v.colour$:= blue **until** $u = v$

- pohjana on syvyyteen ensin -haku
  $\Rightarrow$ vahvat komponentit valmistuvat "alin" ensin
  - niistä voidaan pitää kirjaa pinolla S
  - (vain) valmiiden vahvojen komponenttien solmut ovat sinisiä
  - testaako algoritmi vahvojen komponenttien valmistumisen oikein?
- solmut numeroidaan juoksevasti sitä mukaa kuin ne löytyvät
  $\Rightarrow$ solmun numero voidaan tulkita solmun löytymisajankohdaksi
- osoitamme, että $u.low$ on pienin C_u*:hun kuuluvan* u:sta löydetyn solmun nro
  - olkoon v se solmu, jolle $u.low = v.nr$
  - $u \rightarrow^* v$ koska $v.nr$ kulkeutui u:hun
  - jos $v \notin C_u$, niin polulla $u \rightarrow^* v$ on komponenttirajan ylittävä kaari (u', v')
  - testin $v'.low < u'.low$ hetkellä v':stä ei päässyt harmaisiin solmuihin
  $\Rightarrow$ $C_{v'}$ oli valmis, v' oli sininen ja $v'.low$ hylättiin $\nearrow\!\!\!\!\!\swarrow$

$\Rightarrow$ kun peruutetaan komponentista pois, niin $u.low \geq u.nr$
  - toisaalta selvästi aina $u.low \leq u.nr \;\Rightarrow\;$ testi $u.low = u.nr$ toteutuu

- kun peruutetaan komponentin sisällä u:sta w:hen, niin
  - $u \rightarrow^* w$ ja polulla on kaari (u', w') siten, että
    w' löytyi ennen ja $u' = u$ tai u' löytyi jälkeen u:n
  $\Rightarrow$ $u.low \leq u'.low \leq w'.nr < u.nr \;\Rightarrow\;$ testi $u.low = u.nr$ ei toteudu

$\Rightarrow$ $u.low = u.nr$ testaa luotettavasti komponentin valmistumisen

- suoritusaika on $\Theta(|V| + |E|)$

Kosaraju–Sharir-algoritmi

- algoritmi
  - ajetaan DFS ja talletetaan solmut peruuttamisjärjestyksessä pinoon
  - käännetään graafin kaaret takaperin ja väritetään solmut valkoisiksi
  - otetaan solmut pinosta ja ajetaan kullekin valkoiselle DFS;
    kukin ajo käy läpi yhden vahvan komponentin
- (takaperin-)DFS käy kohtaamansa komponentit läpi kokonaan
  $\Rightarrow$ riittää osoittaa, että kukin takaperin-DFS
  käy läpi vain yhden vahvan komponentin
- olkoon v mikä tahansa takaperin-DFS:n alkusolmu ja $u \notin C_v$
- jos $u \not\to^* v$, niin takaperin-DFS(v) ei löydä u:ta
- jos $u \to^* v$ niin $v \not\to^* u$, joten etuperin-DFS:issä pätee
  - jos C_u kohdattiin ennen kuin C_v, niin C_v valmistui ennen kuin C_u
  - jos C_u kohdattiin C_v:n jälkeen, niin C_v oli valmis jo kun C_u kohdattiin

  $\Rightarrow$ C_u käsiteltiin takaperin-DFS:issä ennen C_v:tä
  $\Rightarrow$ u oli musta kun takaperin-DFS(v) alkoi
- tämänkin suoritusaika on $\Theta(|V| + |E|)$

Dijkstran vahvojen komponenttien algoritmi

- aina kun DFS tunnistaa silmukan, se korvataan yhdellä solmulla v
  - v:n tietoihin laitetaan alkuperäisten solmujen luettelo
  - alkuperäisiin solmuihin liittyvät kaaret käännetään liittymään v:hen
- lopulta kukin vahva komponentti on kutistunut yhdeksi solmuksi
- suoritusajan $\Theta(|V| + |E|)$ saavuttaminen vaatii huolellisuutta

Algoritmien vertailua

- Kosaraju–Sharir-algoritmi kuluttaa vähiten muistia
  - Kosaraju–Sharir: DFS:n muisti ja yksi pino
  - Tarjan: DFS:n muisti, $.nr$, $.low$ ja (mahdollisesti matala) pino
  - Dijkstra: kaaret tarvitaan *yhtäaikaa* molempiin suuntiin
- Tarjanin ja Dijkstran algoritmit alkavat löytää vahvoja komponentteja ennen kuin koko graafi on käyty läpi

Muita graafialgoritmeja monenlaisiin tehtäviin tunnetaan valtava määrä

8 Informaatioteoriaa

Johdattelevia kysymyksiä

- on 26 saman painoista kultarahaa ja yksi muita kevyempi, miten kevyempi löydetään orsivaa'alla kolmella punnituksella?
- onko maaliero eräs kastematolaji vai urheilutermi, ja mikä on kaivosaukko?
- voiko täysin valkoinen pinta sisältää informaatiota?
  - valkoinen lippu sodassa

8.1 Kiinteän ja vaihtuvan pituiset koodit

n bittiä kykenee esittämään enintään 2^n vaihtoehtoa

- 8 bittiä esim.
  - luvut 0, 1, . . . , 255
  - luvut –128, . . . , –1, 0, 1, . . . , 127
  - numerot, kirjaimet ja joukon muita merkkejä
- kukin bittiyhdistelmä voi edustaa mitä tahansa halutaan
  - 000 = koira, 001 = kissa, 010 = lehmä, . . .

Sama toisinpäin: n vaihtoehdon esittämiseen tarvitaan $\lceil \log_2 n \rceil$ bittiä

Yleisemmin n-pituinen jono Σ:n alkioita kykenee esittämään $|\Sigma|^n$ vaihtoehtoa

- bittien tapauksessa $\Sigma = \{0, 1\}$ ja $|\Sigma| = 2$
- käytännön ohjelmoinnissa usein Σ = 8-bittiset tavut $\cup$ {tiedoston loppu}
  $\Rightarrow |\Sigma| = 257$
- geneettisessä koodissa (RNA) $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{U}\}$
- teoreettisissa tarkasteluissa oletetaan yleensä $2 \leq |\Sigma| < \infty$
- jatkossa oletamme, että $\Sigma = \{0, 1\}$, ellei toisin sanota

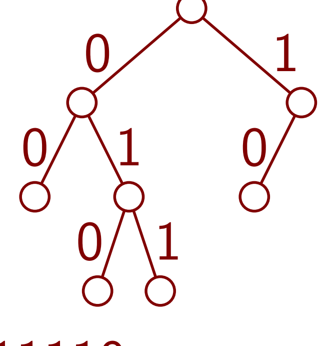
Nimiä

- **lähdesana** = se, mikä halutaan esittää (esim. ”lehmä”)
- **koodisana** = sen bittijono (esim. 010)
- **koodi** = koodisanojen muodostama kokonaisuus
  - eri lähdesanoilla on oltava eri koodisanat, tai koodia ei voi purkaa
  - pian asetetaan tarkempi vaatimus
  - koodisanoiksi kelpaavia bittijonoja saa jäädä käyttämättä
  - samalla lähdesanalla saa olla monta koodisanaa, esim. lysiini: AAA ja AAG

Koodisanojen ei välttämättä tarvitse olla keskenään yhtäpitkiä

- yleisimmille lähdesanoille voidaan antaa lyhyemmät koodisanat kuin harvinaisille
  ⇒ koodatun viestin pituuden odotusarvo lyhenee
  - **odotusarvo** = todennäköisyyksillä painotettu keskiarvo
- lähdesanoja voi olla äärettömästi, esim. kaikki luonnolliset luvut
  - (luvussa 10 tähän tulee rajoite)
- *edellytys*: peräkkäisten koodisanojen rajat on oltava tunnistettavissa
  - muuten ”karuselli” ja ”karu selli” menevät sekaisin
  - esim. koko syöte on yksi koodisana ja syötteen loppumisen voi testata
  - toisin kuin luonnollisissa ja ohjelmointikielissä, väliyöntien tms. käyttö erottimena on informaatioteoriassa harvinaista

Itserajaava koodi (prefix code): mikään koodisana ei ole toisen aito alkuosa

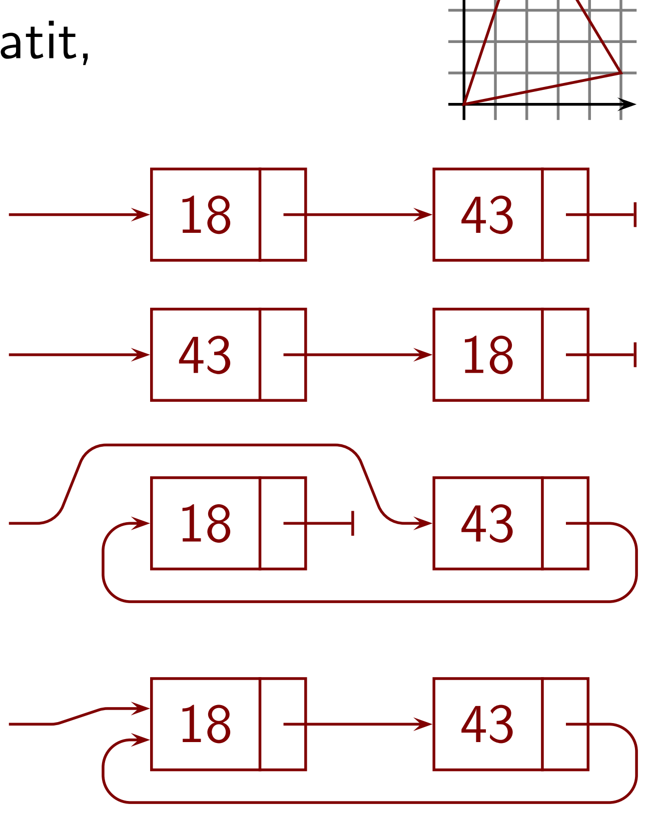


- vrt. ”maali”
- voidaan piirtää puuna, jonka lehdet vastaavat koodisanoja
- esimerkkejä
  - 00 = koira, 010 = kissa, 011 = lehmä, 10 = hevonen
  - $0 =$ 0, $1 =$ 100, $2 =$ 110, $3 =$ 10100, $4 =$ 10110, $5 =$ 11100, $6 =$ 11110, $7 =$ 1010100, ... parittomat bitit ovat 1 paitsi viimeinen on 0, parilliset koodaavat lukuarvon
  - koko tiedosto, jos sen loppuminen erottuu syötteestä: `!std::cin`
  - useat tietorakenteet (”alkuosa” järkevästi tulkiten)
- ohjelmointikielet
  - tekstialkiot eivät aina ole itserajaavia, esim. <=
  - monien ohjelmointikielten syntaksi on itserajaava (mutta ei C++:n)
- (joskus tehdään tämän opintojakson kannalta merkityksetön erottelu
  - prefix code = mikään koodisana ei ole toisen aito alkuosa
  - self-delimiting code = lisäksi tietokone aina tunnistaa koodisanan lopun, vrt. luku 10)
- yleisempi ”yksikäsitteisesti purettava koodi” tulee luvussa 8.4

Informaatioteoriassa yleensä käsitelty ja yleisempi tilanne

- alun perin keskityttiin tietoliikenteen tarpeisiin
  - koodisanoja tulee välittömästi peräkkäin
  - pituuden odotusarvo pyrittiin minimoimaan

  $\Rightarrow$ kullakin lähdesanalla on vain yksi koodisana ja käyttämättömiä bittijonoja vältetään
- yleisemmin samalla lähdesanalla saa olla monta koodisanaa
  - moninkertaiset koodisanat saattavat jopa syntyä luonnostaan
  - vrt. matematiikassa $\frac{2}{4} = \frac{1}{2}$
- esim. kolmio esitetään antamalla kolmen pisteen koordinaatit, eikä pisteiden esittämisjärjestyksellä ole väliä
  - esim. (0,0)(2,6)(5,1) ja (5,1)(0,0)(2,6)
- esim. tietorakenteissa
  - linkitetyn listan esittämä joukko ei riipu listan järjestyksestä
  - linkitetyn listan sisältö ei riipu listan tietueiden osoitteista
- käyttämättömät bittiyhdistelmät voivat olla yleisiä
  - vrt. matematiikassa $\frac{2}{0}$
  - esim. virheellinen listarakenne on käyttämätön bittiyhdistelmä



- ohjelmointikielissä tekstialkioiden välissä voi olla välilyöntejä yms.
  ⇒ tekstialkioiden ei tarvitse muodostaa itserajaavaa koodia
- kielen ja sen merkityksen välinen suhde on tärkeä aihe,
  mutta ei kuulu informaatioteorian piiriin
  - informaatioteoriassa koodisanat saa valita mielivaltaisesti
- lähdekielen rakenteesta johtuvat ilmiöt kuuluvat muiden teorioiden piiriin
  - esim. alku- ja loppusulkujen vastaavuus $(\ldots (\ldots) \ldots ((\ldots) \ldots))$

⇒ informaatioteoriaa voi soveltaa yleisempään tilanteeseen,
mutta se ei vastaa kaikkiin mielenkiintoisiin kysymyksiin

Jatkossa rajoitumme tapaukseen, jossa koodisanoja on äärellinen määrä

⇒ *myös lähdesanoja on äärellinen määrä*

8.2 Kraftin epäyhtälö ja Shannonin alaraja

Merkintöjä

- olkoot koodisanat σ_i, missä i käy läpi lähdesanat
  - jos lähdesanalla on monta koodisanaa, vain yksi otetaan nyt huomioon
  - eri lähdesanoilla on eri koodisanat, eli jos $i \neq j$, niin $\sigma_i \neq \sigma_j$
- olkoot lähdesanojen todennäköisyydet p_i
- olkoot koodisanojen pituudet $\ell_i = |\sigma_i|$
- tyhjää bittijonoa merkitään ε

Koodatun viestin pituuden **odotusarvo** on $\sum_i p_i \ell_i$

- odotusarvo = kaikkien vaihtoehtojen todennäköisyyksillä painotettu keskiarvo

Miten pituuden odotusarvo saadaan mahdollisimman pieneksi itseraj. koodissa?

- mitä pienempi ℓ_i, sitä parempi, mutta ...
- ... kokeilemalla $\ell_i = 0$ tai $\ell_i = 1$ eli $\sigma_i \in \{\varepsilon, 0, 1\}$ nähdään, että ongelmaksi tulee, että koodisana ei saa olla toisen alkuosa
  - ottamalla 01 koodisanaksi menetetään kaikki pitemmät 01-alkuiset
- koodisanoiksi kelpaavia bittijonoja ei kannata jättää käyttämättä
  - jos 001011 on koodisana mutta mikään 001010-alkuinen ei ole, niin voidaan lyhentää 001011 $\rightsquigarrow$ 00101

- lyhyet koodisanat kannattaa antaa todennäköisille lähdesanoille
  - jos $\ell_i < \ell_j$ ja $p_i < p_j$, niin σ_i ja σ_j kannattaa vaihtaa keskenään, koska $(p_i\ell_i + p_j\ell_j) - (p_i\ell_j + p_j\ell_i) = (p_j - p_i)(\ell_j - \ell_i) > 0$
- jos σ, $\rho 0$ ja $\rho 1$ ovat koodisanoja, *saattaa* kannattaa korvata $\sigma \rightsquigarrow \sigma 0$, $\rho 0 \rightsquigarrow \rho$ ja $\rho 1 \rightsquigarrow \sigma 1$

Kraftin epäyhtälö itserajaaville koodeille

- jos jokaisen koodisanan pituus olisi n, niin koodisanoja voisi olla $\leq 2^n$
- jos $\ell_i \leq n$, niin σ_i vie $2^{n-\ell_i}$ kappaletta ym. mahdollisuuksista
  - esim. jos $\ell_i = n - 2$, niin σ_i vie mahdollisuudet $\sigma_i 00$, $\sigma_i 01$, $\sigma_i 10$ ja $\sigma_i 11$
  - ts. σ_i vie osuuden $\dfrac{2^{n-\ell_i}}{2^n} = 2^{-\ell_i}$ mahdollisuuksista
  - osuus on $2^{-\ell_i}$ riippumatta n:stä
- koska osuus on osuus kaikesta, on osuuksien summa ≤ 1

$\Rightarrow$ olemme todistaneet **Kraftin epäyhtälön** itserajaaville koodeille

$$\sum_i 2^{-\ell_i} \leq 1$$

Vähemmän mielikuviin vetoava todistus

- jos koodisanoja ei ole lainkaan, on summa $0 \leq 1$
  $\Rightarrow$ tästä eteenpäin oletetaan, että koodisanoja on ainakin yksi
- käytetään induktiota pisimmän koodisanan pituuden suhteen
- jos pisimmän koodisanan pituus on 0, niin ε on ainoa koodisana
  $\Rightarrow \sum_i 2^{-l_i} = 2^{-0} = 1$
- muussa tapauksessa koodisanat voidaan jakaa kahteen erilliseen ryhmään: 0-alkuiset ja 1-alkuiset
  - ε ei voi nyt olla koodisana, koska se on jonkin muun koodisanan alkuosa
- olkoot ρ_j ne bittijonot, joille $0\rho_j$ on koodisana
  - ne muodostavat itserajaavan koodin
    (jos ρ_j olisi ρ_h:n aito alkuosa, niin $0\rho_j$ olisi $0\rho_h$:n aito alkuosa)
  - sen pisin koodisana on lyhyempi kuin alkuperäisen koodin

$\Rightarrow$ induktio-oletuksen mukaan $\sum_j 2^{-|\rho_j|} \leq 1$

$\Rightarrow \sum_j 2^{-|0\rho_j|} = \sum_j 2^{-(1+|\rho_j|)} = 2^{-1} \sum_j 2^{-|\rho_j|} \leq 2^{-1} \cdot 1 = \frac{1}{2}$

- samoin $\sum_k 2^{-|1\mu_k|} \leq \frac{1}{2}$, missä $1\mu_k$ on koodisana
- kaikkiaan $\sum_i 2^{-\ell_i} = \sum_j 2^{-|0\rho_j|} + \sum_k 2^{-|1\mu_k|} \leq \frac{1}{2} + \frac{1}{2} = 1$ $\square$

Miten Kraftin epäyhtälö vaikuttaa pituuden odotusarvoon $\sum p_i \ell_i$?

- oletamme, että $p_i > 0$ (muuten koodisana i olisi tarpeeton)
- jos ℓ_i pienenee, niin $2^{-\ell_i}$ kasvaa

$\Rightarrow \sum_i 2^{-\ell_i}$ kannattaa saada mahdollisimman suureksi (ehdolla $\sum_i 2^{-\ell_i} \leq 1$)

- sitten kun tämä liikkumavara on käytetty, yhtä ℓ_i ei voi pienentää kasvattamatta jotain muuta ℓ_i
- jos p_i, p_j ja $2^{-\ell_i} + 2^{-\ell_j}$ pidetään vakiona, miten minimoidaan $p_i \ell_i + p_j \ell_j$?
- merkitään $C = 2^{-\ell_i} + 2^{-\ell_j}$ ja $x = 2^{-\ell_i}$
  - $2^{-\ell_i} > 0$ ja $2^{-\ell_j} > 0$, joten $0 < x < C$

$\Rightarrow$ minimoitava $f(x) = -p_i \log_2 x - p_j \log_2(C - x)$ ehdolla $0 < x < C$

- jos $x \xrightarrow{+} 0$, niin $l_i \to \infty$, $l_j \to -\log_2 C > 0$ ja $p_i \ell_i + p_j \ell_j \to \infty$
- myös jos $x \xrightarrow{-} C$, niin $p_i \ell_i + p_j \ell_j \to \infty$
- $\dfrac{df(x)}{dx} = \dfrac{-p_i}{x \ln 2} + \dfrac{p_j}{(C-x)\ln 2} = 0$, kun $\dfrac{p_i}{x} = \dfrac{p_j}{C-x}$ eli $\dfrac{2^{-\ell_i}}{p_i} = \dfrac{2^{-\ell_j}}{p_j}$

$\Rightarrow$ minimi, kun $\dfrac{2^{-\ell_i}}{p_i}$ on sama jokaiselle i, eli on vakio D s.e. $\forall i : 2^{-\ell_i} = D p_i$

$\Rightarrow \sum_i 2^{-\ell_i} = \sum_i D p_i = D \sum_i p_i = D$, koska todennäk. summa = $\sum_i p_i = 1$

$\Rightarrow \sum_i p_i \ell_i$ on minimissään, kun $D = 1$ ja $\forall i : \ell_i = -\log_2 p_i$

Olemme todistaneet **Shannonin alarajan** pituuden odotusarvolle

$$\sum_i p_i \ell_i \;\geq\; -\sum_i p_i \log_2 p_i$$

- suuretta $-\sum_i p_i \log_2 p_i$ kutsutaan nimellä (lähteen) **entropia**
- $-\sum_i p_i \log_2 p_i \geq 0$, koska $0 \leq p_i \leq 1$

Miten tämä suhtautuu siihen, että n vaihtoehtoa tarvitsee $\lceil \log_2 n \rceil$ bittiä?

- jos vaihtoehdot ovat yhtä todennäköisiä, Shannonin alaraja on

$$-\sum_i p_i \log_2 p_i \;=\; -n\frac{1}{n}\log_2\frac{1}{n} \;=\; \log_2 n$$

$\Rightarrow$ sama raja ilman pyöristystä ylöspäin lähimmäksi kokonaisluvuksi!

- jos kaikki vaihtoehdot ovat yhtä todennäköisiä, niin yhden vaihtoehdon esittämiseen tarvitaan keskimäärin $\log_2 n$ ja huonoimmillaan $\lceil \log_2 n \rceil$ bittiä
  - Shannonin alaraja saavutetaan tarkasti, jos n on 2:n potenssi
  - esim. jos $n = 3$ ja koodisanat ovat 0, 10 ja 11, on $\sum_i p_i \ell_i = \frac{5}{3} \approx 1,667$ ja $-\sum_i p_i \log_2 p_i \approx 1,585$
  - jos $n = 3$, niin koodaamalla lähdesanojen pareja saadaan $\sum_i p_i \ell_i \approx 1,611$
- seuraavaksi tutkimme, miten yleisessä tapauksessa päästään lähimmäksi Shannonin alarajaa

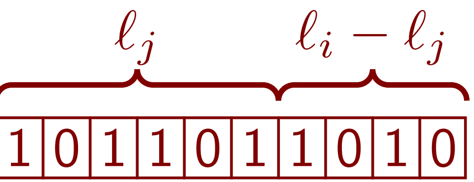
8.3 Kuinka lähelle Shannonin alarajaa päästään?

Annetut ℓ_i toteuttava koodi

- olkoon annettu n luonnollista lukua ℓ_i siten, että $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$
- ne voidaan numeroida siten, että kun $1 \leq j < i \leq n$, niin $\ell_j \leq \ell_i$
- merkitkäämme bittijonon $b_1 b_2 \cdots b_k$ lukuarvoa $[\![b_1 b_2 \cdots b_k]\!] = \sum_{i=1}^{k} b_i 2^{k-i}$
  - k-pituiset bittijonot ja luonnolliset luvut $0, \ldots, 2^k - 1$ ovat bijektiossa
- valitaan σ_i siten, että $[\![\sigma_i]\!] = \sum_{k=1}^{i-1} 2^{\ell_i - \ell_k}$
  - $\sum_{k=1}^{i-1} 2^{\ell_i - \ell_k} \in \mathbb{N}$, koska $k < i$ ja siksi $\ell_k \leq \ell_i$
  - koska $\sum_{k=1}^{n} 2^{-\ell_k} \leq 1$, on $\sum_{k=1}^{i-1} 2^{\ell_i - \ell_k} = 2^{\ell_i} \sum_{k=1}^{i-1} 2^{-\ell_k} < 2^{\ell_i}$

  $\Rightarrow$ $\sum_{k=1}^{i-1} 2^{\ell_i - \ell_k}$ on oikealla välillä

  $\Rightarrow$ valinta on mahdollinen



- kun $i > j$, niin
  - σ_i ei ole σ_j:n aito alkuosa, koska $\ell_i \geq \ell_j$
  - $[\![\sigma_i$:n ℓ_j ensimmäistä bittiä$]\!] + 1 > \frac{[\![\sigma_i]\!]}{2^{\ell_i - \ell_j}} = \sum_{k=1}^{i-1} 2^{\ell_i - \ell_k - \ell_i + \ell_j} \geq \sum_{k=1}^{j} 2^{\ell_j - \ell_k} = [\![\sigma_j]\!] + 1$, joten σ_j ei ole σ_i:n alkuosa

$\Rightarrow$ aina kun Kraftin epäyhtälö toteutuu, vastaava itserajaava koodi on olemassa

$\Rightarrow$ Kraftin epäyhtälö on tiukka raja

Optimaaliset koodit

- optimaalinen koodi minimoi koodisanan pituuden odotusarvon
- oletimme, että lähdesanoja on vain äärellinen määrä

⇒ vain äärellinen määrä koodeja ei sisällä tarpeettoman pitkiä koodisanoja
  - esim. sanaa $\sigma 0 \rho$ siten, että mikään $\sigma 1 \rho'$ ei ole käytössä

⇒ ei voi olla loputtomasti vaihtoehtoja optimaaliseksi koodiksi

⇒ on olemassa ainakin yksi optimaalinen koodi

Huffmanin koodi

- edellä nähtiin, että lyhimmät koodisanat kannattaa antaa todennäköisimmille lähdesanoille

⇒ epätodennäköisimmät kaksi lähdesanaa saavat pisimmät koodisanat
  - jos yhtä epätodennäköisiä on monta, ei ole väliä, mitkä kaksi niistä valitaan

- jos pisin koodisana on $\sigma 0$ tai $\sigma 1$, niin myös $\sigma 1$ tai $\sigma 0$ kannattaa hyödyntää

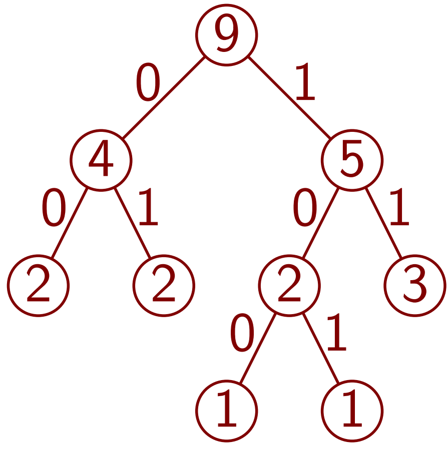
⇒ epätodennäköisimmät kaksi koodisanaa voidaan laittaa saman solmun lapsiksi

- sama päättely toimii yksittäisten koodisanojen lisäksi alipuille
  - alipuun todennäköisyys on sen lehtien todennäköisyyksien summa

⇒ (eräs) optimaalinen koodi löytyy seuraavasti:

> tee jokaisesta lähdesanasta x ja sen todennäköisyydestä $x.p$ solmu
> laita kaikki solmut prioriteettijonoon
> **for** $i := 1$ **to** (lähdesanojen määrä) $- 1$ **do**
>   ota prioriteettijonosta kaksi epätodennäköisintä alipuuta x ja y
> z := uusi solmu; $z.vasen := x$; $z.oikea := y$; $z.p := x.p + y.p$
>   lisää z prioriteettijonoon

- tämä on **Huffmanin koodi**
- esimerkki: todennäköisyydet 1, 1, 2, 2 ja 3 yhdeksäsosaa
  - 1, 1, 2, 2, 3 ⇝ 2, 2, 2, 3
  - 2, 2, 2, 3 ⇝ 2, 3, 4
  - 2, 3, 4 ⇝ 4, 5
  - 4, 5 ⇝ 9



Lähdesanojen parien, kolmikoiden, ... koodaaminen

- optimaalinen koodi voi olla Shannonin alarajaa huonompi
  - jos $n = 3$, niin Shannonin alaraja $\approx 1{,}585$ ja optimaalinen koodi $\approx 1{,}667$
- optimaalinen koodi on alle yhden bitin verran Shannonin alarajaa huonompi
  - jos $\ell_i = \lceil -\log_2 p_i \rceil$, niin $\sum_i 2^{-\ell_i} \leq \sum_i 2^{\log_2 p_i} = \sum_i p_i = 1$

  $\Rightarrow$ voidaan valita $\ell_i = \lceil -\log_2 p_i \rceil < -\log_2 p_i + 1$
  - silloin $\sum_i p_i \ell_i < \sum_i p_i(1 - \log_2 p_i) = 1 - \sum_i p_i \log_2 p_i$

$\Rightarrow$ jos koodataan m:n lähdesanan yhdistelmiä, päästään alle $\frac{1}{m}$ bitin päähän Shannonin alarajasta
  - esim. 0000 = koira koira koira, 0001 = koira koira kissa, 0010 = koira koira lehmä, ..., 11111 = lehmä lehmä lehmä

$\Rightarrow$ Shannonin alaraja on luonnollisempi mitta informaation määrälle kuin optimaalisen koodin pituuden odotusarvo

8.4 Lisähuomioita

Yksittäisen lähdesanan informaation määrästä

- koodisanojen pituudet saa valita mielivaltaisesti (Kraftin epäyhtälön puitteissa)

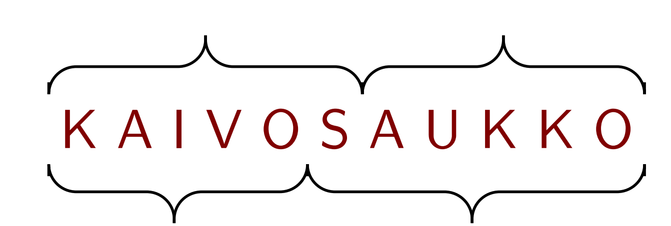
⇒ mille tahansa lähdesanalle voidaan valita lyhyt koodisana
  - vrt. valkoinen lippu

- erilaisia lyhyitä bittijonoja on vain rajallinen määrä

⇒ ei voida valita lyhyttä koodisanaa kovin monelle lähdesanalle yhtäaikaa
  - tämä pätee, vaikka ei vaadittaisi itserajaavuutta

- informaatioteoria ei anna yksittäiselle lähdesanalle informaation absoluuttista määrää, vaan informaatiota voidaan mitata
  - koodisanan pituutena: riippuu koodista
  - Shannonin alarajan mukaan luvulla $-\log_2 p_i$: määräytyy tod.näköisyyksistä
- ns. Kolmogorov-kompleksisuus antaa yksittäiselle sanalle informaation määrän, mutta se on eri teoria
  - mitä epäsäännöllisempi bittijono, sitä enemmän informaatiota, vrt. 1111111111111111, 0101010101010101 ja 0110001000011001

Yksikäsitteisesti avattavat koodit



- koodi on **yksikäsitteisesti avattava** (**uniquely decodable**), jos ja vain jos mikään bittijono ei ole esitettävissä kahdella eri tavalla koodisanojen jonona
- jokainen itserajaava koodi on yksikäsitteisesti avattava
- B. McMillan osoitti, että Kraftin epäyhtälö pätee kaikille yksikäsitteisesti avattaville koodeille
- Karush antoi helpomman todistuksen, jonka käymme nyt läpi
- olkoon $n \in \mathbb{N}$ mielivaltainen
- jos $\rho = \rho_1 \rho_2 \cdots \rho_n$, niin $2^{-|\rho|} = 2^{-(|\rho_1|+|\rho_2|+\ldots+|\rho_n|)} = 2^{-|\rho_1|} 2^{-|\rho_2|} \ldots 2^{-|\rho_n|}$

$\Rightarrow \left(\sum_i 2^{-\ell_i}\right)^n = \sum_\rho 2^{-|\rho|}$, missä ρ käy läpi kaikki sanat, jotka ovat muodostettavissa laittamalla n koodisanaa peräkkäin
  - jos ρ syntyy usealla tavalla, se on mukana summassa useasti

- olkoon $M = \max\{\ell_i\}$
  - $0 \leq |\rho| \leq nM$
- jos mikään ρ ei esiinny useasti, on $\sum_\rho 2^{-|\rho|} \leq \sum_{k=0}^{nM} 2^k 2^{-k} = nM + 1$

$\Rightarrow$ jos koodi on yksikäsitteisesti avattava, niin $\left(\sum_i 2^{-\ell_i}\right)^n \leq nM + 1$

- jos $\sum_i 2^{-\ell_i} > 1$, niin $\left(\sum_i 2^{-\ell_i}\right)^n$ kasvaa nopeammin kuin $nM + 1$ ↯

$\Rightarrow \sum_i 2^{-\ell_i} \leq 1$ □

- Shannonin alarajan todistus ei oleta koodista muuta kuin että Kraftin epäyhtälö pätee

$\Rightarrow$ myös Shannonin alaraja pätee yksikäsitteisesti avattaville koodeille

Rajoittamattomien luonnollisten lukujen esittämisestä

- näimme jo yhden tavan esittää rajoittamaton luonnoll. luku: $1b_1 1b_2 1 \cdots 1b_i 0$
  - $b_1 b_2 \cdots b_i$ esittää jonkin luvun väliltä $2^i - 1, \ldots, 2^{i+1} - 2$
  - luvun n esitys käyttää $1 + 2\lfloor \log_2(n+1) \rfloor \approx 2\log_2 n$ bittiä
- käyttämällä ”jatkuu”-bittejä harvemmin päästään helposti esim. $\approx 1{,}25 \log_2 n$ bittiin
- voidaanko päästä $\approx \log_2 n$ bittiin?
- oletetaan, että $\leq \log_2(n+1) + c$ bittiä riittää, missä c on vakio
- Kraftin epäyhtälöstä saadaan

$$1 \geq \sum_{i=0}^{\infty} 2^{-(\log_2(n+1)+c)} = 2^{-c} \sum_{i=0}^{\infty} \frac{1}{n+1} = 2^{-c} \sum_{i=1}^{\infty} \frac{1}{n}$$

- $\lightning$, koska $\sum_{i=1}^{\infty} \frac{1}{n} = (\frac{1}{1}) + (\frac{1}{2}) + (\frac{1}{3} + \frac{1}{4}) + (\frac{1}{5} + \ldots + \frac{1}{8}) + \ldots \geq \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \ldots$

$\Rightarrow$ ei voida päästä $\approx \log_2 n$ bittiin

9 Tiedon pakkaus ja salaus

9.1 Tiedon pakkaus

Tiedon **pakkaus (compression)** tarkoittaa tiedon muuntamista mahdollisimman vähän muistia kuluttavaan muotoon

- jakautuu kahteen alalajiin: **häviötön (lossless)** ja **häviöllinen (lossy)**

Häviöllinen pakkaus

- alkuperäistä dataa ei välttämättä saada palautettua tarkalleen
- käytetään esim.
  - valokuvien pakkaamiseen: JPEG
  - äänen pakkaamiseen: MP3
- pyritään hävittämään yksityiskohtia, joita käyttäjä ei huomaa

Yksinkertainen häviöllisen pakkauksen esimerkki

- pikselin väri kootaan yleensä yhdistelmänä punainen–vihreä–sininen (RGB)
- kukin perusväri esitetään usein 8 bitillä
  $\Rightarrow$ yhteensä 3 tavua ja 16 777 216 värisävyä
- vähennetään yksi bitti / perusväri
  $\Rightarrow$ 12,5% säästö muistin määrässä

Parempi häviöllinen pakkaus

- vaikka silmä ei erota, että väri on hieman väärä, se erottaa, jos väri muuttuu hyppäyksenomaisesti

$\Rightarrow$ ei kannata tallettaa likiarvoistettua väriä, vaan esim. likiarvoistettu erotus viereisen pikselin väriin

- ajatus voidaan viedä pitemmälle esim. Fourier-muunnoksilla
  - JPEG: diskreetti kosinimuunnos

Esimerkki: sarja JPEG-kuvia

- $768 \cdot 576$ pikseliä, $3 \cdot 8$ bittiä / pikseli

$\Rightarrow$ raakadatana 1 327 104 tavua

- esimerkeissä on 156 708, 68 984, 41 863, 29 841, 23 319, 21 084 ja 21 081 tavua
- kuvat on pilkottu $8 \cdot 8$ pikselin lohkoiksi, jotka on pakattu kukin erikseen
- huomaa huonoimmassa kuvassa liukuvärjättyjä lohkoja taivaan alarajalla

Yksinkertainen häviöttömän pakkauksen esimerkki

- tietokoneohjelmien rivien aluissa on paljon tyhjää (sisennys)

⇒ muistia voitaisiin yleensä säästää seuraavasti:
  - valitaan jokin harvoin esiintyvä merkki, esim. $
  - päätetään, että $1, . . . , $9 tarkoittavat 1, . . . , 9 välilyöntiä
  - päätetään, että $0 tarkoittaa $

- muistia ei kuitenkaan säästy, jos ohjelmassa lukee usein esim.
  `char *d = "$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$";`
  tai `/*$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$*/`
- ajatusta voi kehittää edelleen
  - esim. vain rivin alussa esiintyvä $ tulkitaan ym. tavalla
- näin kehitetty pakkausmenetelmä toimii hyvin vain tietyntyyppiselle datalle
  - tässä tietokoneohjelmille, eikä ole kovin tehokas niillekään

Häviöttömän yleispätevän pakkaamisen mahdottomuus

- oletetaan, että pakkaamaton ja pakattu data käyttävät samaa merkistöä
- oletetaan, että jokainen merkkijono voi esiintyä alkuperäisenä datana
- jokainen pakattu merkkijono täytyy voida muuttaa pakkaamattomaksi

⇒ eri merkkijonoilla täytyy olla eri pakatut versiot

- olkoon m_n enintään n merkin mittaisten merkkijonojen määrä

$\Rightarrow$ niiden pakattuja versiota on (vähintään) m_n kpl.

$\Rightarrow$ joko ne käyttävät kaikki $\leq n$ merkin merkkijonot, tai jokin on pitempi kuin n

$\Rightarrow$ jos häviöttömälle pakkausmenetelmälle on syöte, jota se pienentää, niin sille on syöte, jota se suurentaa

Häviötön pakkaus onnistuu siksi, että kaikki syötemerkkijonot eivät ole yhtä todennäköisiä

Tätä voi hyödyntää luvun 8.3 Huffmanin koodilla

- yksinkertaisinta on koodata yksittäisiä merkkejä
- tulosta voi parantaa koodaamalla merkkipareja, merkkikolmikoita, ...
  - 0-alkuisten koodisanojen todennäk. summaksi saadaan tarkemmin $\frac{1}{2}$
  - merkin todennäköisyys voi riippua edellisen merkin todennäköisyydestä (esim. suomen kielessä a:n todennäköisyys laskee jyrkästi, jos myös kaksi edellistä merkkiä ovat a)
- koodattavien yksiköiden ei tarvitse olla yhtä pitkiä
- kun koodattavat yksiköt on valittu, niin luvun 8.2 Shannonin alaraja asettaa rajan sille, kuinka hyvin voi pakata

Koodattavia yksiköitä ei kuitenkaan voi pidentää kovin paljoa

- jos yksiköt ovat yhtä pitkiä, muunnostaulukon koko kasvaa eksponentiaalisesti
  $\Rightarrow$ se kasvaa nopeasti liian isoksi
- liian pitkälle vietynä todennäköisyydet vääristyvät
  - erilaisia 6 aakkosen jonoja on $\approx$ 594 miljoonaa

  $\Rightarrow$ tarvitaan valtava aineisto, jotta saataisiin mielekkäät frekvenssit
- äärimmilleen vietynä esim. kirjasto olisi pakattu siten, että kirjan pakattu muoto on esim. 6-numeroinen luku, ja kirjan sisältö on muunnostaulukossa ko. luvun kohdalla

Huffmanin koodissa on heikkouksia

- sama muunnostaulukko ei sovellu monentyyppiselle datalle

$\Rightarrow$ jos halutaan monenlaiselle datalle sopiva menetelmä, niin muunnostaulukko on sisällytettävä pakattuun dataan
  - vie tilaa
  - pakattava data on luettava kahdesti:
    ensin muunnostaulukon luomiseksi ja sitten pakkaamiseksi
- datan laatu ja siis todennäköisyydet voivat vaihdella datan sisällä
  $\Rightarrow$ muunnostaulukko toimii hyvin vain osassa dataa

Lempel–Ziv–Welsh-menetelmä

- välttää edellä mainitut muunnostaulukon ongelmat
- käytössä mm. Unixissa ja GIF-kuvaformaatissa
  - jatkossa on mukailtu menetelmän toteutusta GIF-kuvaformaatissa
- olkoon b merkin bittien määrä ja max suurin käytössä oleva koodisana
  - jos merkit ovat tavuja, niin $b = 8$
- koodin purkualgoritmi

```
k := b + 1; m := 2^b + 1; ins := false
repeat
    i := seuraavien k syötebitin muodostama luku
    if i = 2^b then k := b + 1; m := 2^b + 1; ins := false
    else if i ≠ 2^b + 1 then
        while i ≥ 2^b do S.Push(i); i := S[i]
        print(i); if ins then C[m] := i
        while ¬S.Empty do i := S.Pop; print(C[i])
        ins := m < max
        if ins then
            m := m + 1; S[m] := i
            if m = 2^k then k := k + 1
until i = 2^b + 1
```

- muunnostaulukko syntyy syötettä luettaessa
  - sisältää vähintään kahden pituisen merkkijonon jokaiselle $2^b + 2 \leq i \leq m$
  - sen vähintään kahden pituiset aidot alkuosatkin ovat taulukossa
  - se on esitetty takaperin, sen viimeinen merkki on $C[i]$
  - jos $S[i] < 2^b$, niin $S[i]$ on sen ensimmäinen merkki, muutoin $S[i]$ on linkki sen edellisiin merkkeihin
- syötteestä luettu luku i tulkitaan seuraavasti:
  - $0, \ldots, 2^b - 1$ edustavat merkkejä sellaisinaan
  - 2^b on käsky nollata muunnostaulukko
  - $2^b + 1$ ilmaisee, että syöte loppui
  - $2^b + 2, \ldots, m$ edustavat muunnostaulukon merkkijonoja
- jos i on merkin tai merkkijonon koodi ja muunnostaulukossa on tilaa, muunnostaulukkoon lisätään merkkijono
  - sen alkuosa on nyt tulostettu merkki tai merkkijono
  - sen viimeinen merkki on seuraavaksi tulostettavan ensimmäinen merkki
  - ins huolehtii, että viimeinen merkki laitetaan myöhemmin paikalleen

$\Rightarrow$ jos seuraavaksi tulostetaan viimeksi lisätty merkkijono, niin
  - sen viimeinen merkki ei ole tiedossa, kun tulostus alkaa
  - $C[m] := i$ sijoittaa viimeisen merkin ajoissa paikalleen

- seuraavaksi tuleva koodi on välillä $0, \ldots, m$
  - on oltava $m < 2^k$, jotta koodi voitaisiin varmasti esittää k bitillä
  - **if** $m = 2^k$ **then** $k := k + 1$ kasvattaa k:ta tarvittaessa

9.2 Tiedon salaus

Symmetrinen salaus

- tieto salataan ja avataan samalla avaimella
- esimerkki: yksinkertainen, kehno menetelmä

```
k := key
while syötettä jäljellä do
    b := read_byte; print(b xor k); k := (5 · k + 119) mod 256
```

- kunnolliset menetelmät ovat monimutkaisempia
- kunnollisissa menetelmissä käytetään paljon pidempää avainta
  - esim. 256 bittiä
  - koneteho kasvaa vuodesta toiseen

  $\Rightarrow$ salauksen murtamiseen tarvittava aika lyhenee

  $\Rightarrow$ avaimen pituutta kasvatetaan aika ajoin
- ongelma: miten välitetään avain lähettäjältä vastaanottajalle?

Julkisen avaimen salaus

- mullisti salauksen 1970-luvulta alkaen
- avaimet esiintyvät pareina
  - salaamiseen ja avaamiseen käytetään eri avaimia
  - toimivat molemmin päin: kun toisella salaa, niin toisella voi avata

- jokaisella osapuolella on avainpari
  - toinen avain kerrotaan muille: **julkinen avain**
  - toinen avain pidetään salassa: **salainen avain**
- julkisesta avaimesta ei voi helposti päätellä salaista avainta
- jos viesti salataan vastaanottajan julkisella avaimella, niin muut eivät voi (helposti) avata sitä
- jos viesti salataan lähettäjän salaisella avaimella, niin vastaanottaja voi olla melko varma, että se ei ole väärennetty
- viestin voi salata molemmilla
- ongelma: menetelmät ovat hitaampia kuin yhtä varmat symmetriset
- ratkaisu:
  - arvotaan yksi symmetrisen menetelmän avain
  - salataan se vastaanottajan julkisella avaimella ja laitetaan tiedoston alkuun
  - salataan data symmetrisellä menetelmällä ko. avaimella

Esimerkki: RSA

- aluksi valitaan kaksi isoa alkulukua p ja q ja lasketaan $n = pq$
  - luvun testaaminen alkuluvuksi onnistuu riittävän nopeasti
  - alkulukuja on niin tiheässä, että jokin löytyy riittävän nopeasti testaamalla peräkkäisiä lukuja satunnaisesta kohdasta alkaen

- sitten valitaan luvut d ja e siten, että $x^{de} \bmod n = x$, kun $0 \leq x < n$
  - perustuu luonnollisten lukujen jaollisuuden teoriaan
  - edellyttää, että p ja q tiedetään
- avaimet ovat lukuparit (d, n) ja (e, n)
- viesti x salataan laskemalla $x^d \bmod n$ ja puretaan $x^e \bmod n$ tai toisinpäin
- RSA:n turvallisuus edellyttää, että isoja lukuja ei voi helposti jakaa tekijöihin
  - ei ole sitovasti osoitettu, että edellytys pätee

Salausmenetelmien varmuudesta

- avaimia on murrettu mm. mittaamalla avaimen sisältävän laitteen virran kulutuksen vaihteluita, kun se purkaa viestiä
- RSA-avaimia on murrettu mm. laskemalla kahden eri avaimen n-osien suurin yhteinen tekijä
  - onnistuu nopeasti Euklideen algoritmilla
  - on 1, jos avaimet on tehty hyvin
  - esimerkkitapauksessa satunnaislukugeneraattori oli ollut huono

  $\Rightarrow$ monessa avaimessa oli käytetty samaa p

  $\Rightarrow$ suurin yhteinen tekijä oli p ja avaimet murtuivat

$\Rightarrow$ vaikka menetelmä olisi matemaattisesti varma, se voi silti olla murrettavissa

10 Ratkeamattomuus ja laskennallinen vaativuus

10.1 Ratkeamattomuus

Luvussa 6 mainitsimme, että joskus on vaikea selvittää, pysähtyykö ohjelma

- käymme tarkastelemaan asiaa tarkemmin
- yksinkertaisuuden vuoksi käytämme kuviteltua ohjelmointikieltä

Pysähtymistesteri

- olkoot $prog$ ja inp (äärellisiä!) merkkijonoja
- olkoon halts$(prog, inp)$ ohjelma, joka vastaa true jos ja vain jos merkkijonon $prog$ sisältö on seuraavanlainen ohjelma:
  - sillä on tasan yksi syöteparametri, ja se on tyypiltään merkkijono
  - se pysähtyy lopulta, jos se ajetaan syötteenä inp
- muutoin halts$(prog, inp)$ vastaa false
  - jos $prog$ ei ole ohjelma
  - jos $prog$ on ohjelma, mutta sillä on 0 tai > 1 syöteparametria
  - jos $prog$ on yksiparametrinen ohjelma, mutta parametri ei ole merkkijono
  - jos $prog$ on sellainen ohjelma, mutta ajo $prog(inp)$ ei pysähdy
- halts$(prog, inp)$ ei itse jää ikuiseen silmukkaan millään $prog$ ja inp
- halts$(prog, inp)$ on (yksiparametristen) ohjelmien **pysähtymistesteri**

Itseensä viittaava kutsu

- tarkastellaan seuraavaa ohjelmaa

  $\underline{\text{diag}(inp)}$

  if halts(inp, inp) **then while** true **do** $i := -i$

- diag(inp) on yksiparametrinen ohjelma, jonka parametri on merkkijono
- mitä tekee kutsu diag(diag)?
  - alkajaisiksi se kutsuu halts(diag, diag)
  - halts$(prog, inp)$ on pysähtymistesteri

  $\Rightarrow$ halts(diag, diag) vastaa true, jos ja vain jos diag(diag) pysähtyy
  - jos halts(diag, diag) vastaa true, niin diag(diag) hyppää **while**-silmukkaan

  $\Rightarrow$ ei pysähdy
  - jos halts(diag, diag) vastaa false, niin diag(diag) ohittaa **while**-silmukkan

  $\Rightarrow$ pysähtyy

$\Rightarrow$ diag(diag) *tekee päinvastoin kuin* halts(diag, diag) *väittää sen tekevän*

$\Rightarrow$ halts(diag, diag) vastaa väärin

$\Rightarrow$ halts$(prog, inp)$ ei olekaan pysähtymistesteri $\lightning$

- yksiparametristen ohjelmien pysähtymistesteriä ei ole olemassa
- todistuksen juoni lyhyesti: diag(diag) käyttää halts(diag, diag) ennustamaan oman tulevaisuutensa, ja vastauksen saatuaan tekee päinvastoin

Mitä tämä tulos tarkoittaa?

print 42

- tuskin kukaan odottaa, että tietokoneohjelmat löytäisivät jokaiseen kysymykseen oikean vastauksen
  - mikä on elämän tarkoitus?
  - mikä on ensi viikon oikea lottorivi?
  - tulisiko minun juoda teetä vai kahvia?
- kysymys "pysähtyykö $prog$ jos se käynnistetään syötteenään inp" tuntuisi kuitenkin kuuluvan tietokoneohjelmien pätevyysalueelle
  - oikea vastaus on olemassa: $prog(inp)$ joko pysähtyy lopulta tai ei pysähdy
  - $prog$ ja inp ovat tietokoneen tutkittavissa olevassa muodossa
- **pysähtymisfunktio** Ohjelmat $\times$ Syötteet $\mapsto$ {false, true} on olemassa (on hyvin määritelty), mutta ei ole olemassa ohjelmaa, joka laskee sen
- syötteellinen kyllä–ei-kysymys on **ratkeamaton** (**undecidable**), jos ja vain jos mikään tietokoneohjelma ei vastaa siihen oikein jokaisella syötteellä

$\Rightarrow$ "pysähtyykö annettu ohjelma annetulla syötteellä" on ratkeamaton
  - "annettu ohjelma" ja "annettu syöte" ovat kysymyksen syötteet

- epätäydellisiä pysähtymistestereitä on olemassa
  - esim. simuloi $prog(inp)$ enintään 10^{10^n} askelta, missä $n = |prog|$
  - jos $prog(inp)$ pysähtyy siihen mennessä, vastaa "kyllä"
  - jos $prog(inp)$ jää tunnistettuun ikuiseen silmukkaan, vastaa "ei"
  - muutoin vastaa "en tiedä"

- jokainen epätäydellinen pysähtymistesteri epäonnistuu joillain *prog* ja *inp*

⇒ ratkeamattomuus ei välttämättä tarkoita epäonnistumista *kaikilla* syötteillä
  - määritelmä tarkoittaa vain, että epäonnistutaan *ainakin yhdellä* syötteellä
  - jatkossa nähdään, että siitä seuraa epäonnistuminen äärettömän monella

Nimen diag syy

- em. todistus on ns. **diagonalisointitodistus**
- nimitys on peräisin G. Cantorin todistuksesta, että reaalilukuja ei voi luetella päättymättömällä luettelolla
  - jos voisi, myös välin $0 \leq x < 1$ reaaliluvut voisi luetella
  - päättymättömiä 9-jonoja ei ole pakko käyttää (0,0999... = 0,1000...)
  - jos otetaan mikä tahansa päättymätön luettelo ilman päättymättömiä 9-jonoja, niin kulkemalla sen lävistäjää eli diagonaalia pitkin ja korvaamalla 0 ↦ 1, muut numerot ↦ 0 saadaan luku, joka ei ole luettelossa

0,32156271...
0,56438342...   0,001...
0,01010101...
...

- pysähtymistesterin tapauksessa
  - ulottuvuudet ovat *prog* ja *inp*
  - matriisin alkiot ovat false tai true

Epätäydellisen pysähtymistesterin täydentäminen

- vastatkoon $\mathsf{halts}_1(prog, inp)$ kullekin $prog$ ja inp "kyllä", "ei" tai "en tiedä"
  - jos vastaus on "kyllä" tai "ei", sen on oltava oikein
- $\mathsf{diag}_1(inp)$ on esim. **if** $\mathsf{halts}_1(inp, inp) =$ "kyllä" **then while** true **do** $i := -i$
  - tapauksesta "en tiedä" voidaan tehdä pysähtyvä tai pysähtymätön
- voidaan tehdä parempi testeri, joka vastaa oikein kun $prog = inp = \mathsf{diag}_1$ ja muutoin vastaa samoin kuin $\mathsf{halts}_1(prog, inp)$

  $\underline{\mathsf{halts}_2(prog, inp)}$
  if $prog = \mathsf{diag}_1 \wedge inp = \mathsf{diag}_1$ **then return** "kyllä"
  else return $\mathsf{halts}_1(prog, inp)$

  - $\mathsf{halts}_1(\mathsf{diag}_1, \mathsf{diag}_1)$ ei voi vastata "kyllä" eikä "ei", koska $\mathsf{diag}_1(\mathsf{diag}_1)$ tekisi päinvastoin

  $\Rightarrow$ se vastaa "en tiedä"
  $\Rightarrow$ $\mathsf{diag}_1(\mathsf{diag}_1)$ pysähtyy
  $\Rightarrow$ $\mathsf{halts}_2(\mathsf{diag}_1, \mathsf{diag}_1)$:n vastaus "kyllä" on oikein
- mutta $\mathsf{halts}_2(\mathsf{diag}_2, \mathsf{diag}_2)$ vastaa "en tiedä", missä $\mathsf{diag}_2(inp)$ on **if** $\mathsf{halts}_2(inp, inp) =$ "kyllä" **then while** true **do** $i := -i$
- myös $\mathsf{halts}_1(\mathsf{diag}_2, \mathsf{diag}_2)$ vastaa "en tiedä", koska halts_2 vastaa samoin kuin halts_1, kun $prog \neq \mathsf{diag}_1$ tai $inp \neq \mathsf{diag}_1$
- samalla tavalla voidaan tehdä halts_3, diag_3, halts_4, diag_4, ...

⇒ saadaan päättymätön jono toinen toistaan parempia epätäydellisiä pysähtymistestereitä halts_i ja ohjelmia diag_j

- diag_j on tehty siten, että halts_j vastaa sille ”en tiedä”
- halts_{j+1} on tehty vastaamaan ”kyllä” diag_j:lle
- halts_{i+1} vastaa kullekin syötteelle ”kyllä” tai samoin kuin halts_i

⇒ $\text{halts}_i(\text{diag}_j, \text{diag}_j)$ vastaa ”kyllä”, kun $j < i$ ja ”en tiedä”, kun $j \geq i$

⇒ jokaisella epätäydellisellä pysähtymistesterillä on äärettömästi syötteitä, joille se vastaa ”en tiedä”

Mahdollisten erilaisten syötteiden määrän merkitys

- rajoitutaan yksinkertaisuuden vuoksi kyllä–ei-kysymyksiin
- jos mahdollisia erilaisia syötteitä on vain äärellinen määrä, niin oikeat vastaukset voi taulukoida

⇒ on olemassa ohjelma, joka tuottaa jokaiselle syötteelle oikean vastauksen

```
stupid_halts(prog, inp)
if      prog = ... ∧ inp = ... then return ”ei”
else if prog = ... ∧ inp = ... then return ”kyllä”
else if prog = ... ∧ inp = ... then return ”ei”
...
else return ”en tiedä”
```

⇒ tehtävä voi olla ratkeamaton vain, jos erilaisia syötteitä on äärettömästi

- jos ohjelma epäonnistuu vain äärellisen monella syötteellä, siihen voidaan lisätä vastauksen katsominen taulukosta niillä syötteillä
  - saadaan ohjelma, joka ratkaisee tehtävän kaikilla syötteillä

⇒ ratkeamattomuudesta seuraa epäonnistuminen *äärettömän monella* syötteellä

- jos syötettä ei ole ja kysymys on hyvin määritelty, niin tehtävä on ratkeava
  - esim. Goldbachin hypoteesi: onko totta, että ei ole olemassa parillista lukua > 2, jota ei voi esittää kahden alkuluvun summana
  - sen ratkaisee jompikumpi seuraavista kahdesta ohjelmasta:

    print "kyllä"   print "ei"

  - ongelmana on, että ei tiedetä, kumpi!
  - helpon ohjelman todistaminen oikeaksi voi olla vaikeaa
- "ohjelmaa ei ole" on eri asia kuin "ei tiedetä, mikä ohjelma on oikea"

Laskeeko ohjelma vastauksen itse vai onko ohjelmoija antanut valmiin vastauksen?

- stupid_halts on selvästi saanut vastaukset valmiina
- Bellman–Ford selvittää "itse", onko graafissa negatiivinen silmukka
- ohjelman toiminta voi olla yhdistelmä laskentaa ja taulukosta katsomista
- jos mahdollisia erilaisia syötteitä on äärettömästi, ohjelman on laskettava suurin osa vastauksista "itse"
  - vain äärellinen määrä vastauksia voi olla annettu valmiiksi

⇒ siksikin on teorian kannalta tärkeää, että erilaisia syötteitä on äärettömästi

Huomautus äärellisyydestä ja äärettömyydestä

- ratkeamattomuuden teoriassa ohjelma ja syöte ovat *äärellisiä* merkkijonoja
  - ohjelmoija voi kirjoittaa vain äärellisen määrän koodia
  - jollei syöte ole äärellinen, ohjelma ei koskaan saa luettua sitä kokonaan
- erilaisia äärellisiä merkkijonoja on äärettömästi
  - esim. `a`, `aa`, `aaa`, ...
  - myös erilaisia ohjelmia on äärettömästi
- muualla tietojenkäsittelyteoriassa käsitellään äärettömiäkin merkkijonoja
  - esim. `aaa`$\cdots$, `ababab`$\cdots$, `3,14159`$\cdots$
  - funktio $\mathbb{N} \mapsto \Sigma$
- ratkeamattomuuden teoriassa ohjelma ja syöte eivät voi olla äärettömiä, mutta muuten voivat olla miten suuria tahansa
  - äärellisiä mutta **rajattomia** (**unbounded**)
- todellisessa maailmassa ohjelma ja syöte eivät voi olla rajattomiakaan
  $\Rightarrow$ saattaa tuntua, että ratkeamattomuuden teoriassakin pitäisi olettaa raja
  - olisiko 1 000 000 tai $10^{1\,000\,000}$ sopiva raja ohjelman ja syötteen koolle?
  - nykyisten tietokoneiden muistin määrä on hyvin suuri
  - rajattomuus on kelvollinen likiarvo hyvin suurelle
- ratkeamattomuuden teoriassa oletetaan myös muisti ja aika rajattomiksi
  - kunakin ajanhetkenä vain äärellinen määrä muistia on käytössä
  - laskennan edetessä tämä määrä voi kasvaa rajatta

Ricen lause

- tarkastellaan osittaisia funktioita, jotka voi laskea tietokoneella
  - **laskettava funktio** (**computable function**)
  - osittaisuus tulee siitä, että ohjelma ei ehkä pysähdy kaikilla syötteillä
  - ts. kullekin syötteelle ohjelma tuottaa tulosteen tai ei pysähdy
  - huomion kohteena ei ole ohjelma, vaan sen laskema osittainen funktio
- olkoon $f_\perp$ se osittainen funktio, jolla ei ole arvoa millään syötteellä
  - sen laskee **while** true **do** $i := -i$
- laskettavan funktion kyllä–ei-ominaisuus on **triviaali**, jos ja vain jos se on kaikilla tai ei millään laskettavalla funktiolla
  - "$f(\mathtt{aaa}) = \mathtt{bb}$" ei ole triviaali: joillain laskettavilla f se pätee ja joillain ei
- olkoon X mikä tahansa epätriviaali kyllä–ei-ominaisuus, jota $f_\perp$:lla ei ole
- oletetaan, että $X(prog)$ on ohjelma, joka ratkaisee ominaisuuden X
  - $X(prog)$ kertoo jotain ohjelman $prog$ laskemasta funktiosta
- jonkin ohjelman $other$ laskemalla funktiolla on ominaisuus X
  - oletimme, että ominaisuus X ei ole triviaali
- tarkastellaan seuraavaa ohjelmaa $H(prog, inp)$
  - muunna $prog$ ohjelmaksi $prog'(inp')$, joka ensin ajaa $prog(inp)$ niin että inp tulee vakiomerkkijonosta, ja sen lopetettua ajaa $other(inp')$
  - aja $X(prog')$

- jos $prog(inp)$ ei pysähdy, niin
  - $prog'(inp')$ ei pysähdy millään syötteellä

  $\Rightarrow$ $prog'(inp')$:n laskema osittainen funktio on $f_\perp$

  $\Rightarrow$ $X(prog')$ ja samalla $H(prog, inp)$ vastaa "ei"
- jos $prog(inp)$ pysähtyy, niin
  - $prog'(inp')$ ajaa $other(inp')$

  $\Rightarrow$ $prog'(inp')$ laskee saman osittaisen funktion kuin $other$

  $\Rightarrow$ $X(prog')$ ja samalla $H(prog, inp)$ vastaa "kyllä"

$\Rightarrow$ $H(prog, inp)$ on pysähtymistesteri $\lightning$

- jos $f_\perp$:lla on epätriviaali k.–e.-ominaisuus X, niin sama päättely toimii $\neg X$:lle

$\Rightarrow$ $X(prog)$ ei ole olemassa millekään epätriviaalille k.–e.-ominaisuudelle X

- Ricen lause: jos X on mikä tahansa epätriviaali k.–e.-ominaisuus, niin tehtävä "onko annetun ohjelman laskemalla funktiolla ominaisuus X" on ratkeamaton

$\Rightarrow$ tämä ja muut tulokset kertovat, että ratkeamattomuus on yleinen ilmiö

Tyhjän syötteen pysähtymistesteri

- vastaa kysymykseen, pysähtyykö annettu ohjelma tyhjällä syötteellä
- kysymys koskee lasketun funktion epätriviaalia k.–e.-ominaisuutta
  - tuotetaanko tyhjälle syötteelle tuloste

$\Rightarrow$ Ricen lauseen nojalla tyhjän syötteen pysähtymistesteriä ei ole olemassa

Ahkera majava (busy beaver)

- $\leq n$ merkin pituisia ohjelmia on rajallisesti
  - $\leq 1 + |\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^n$
- $\leq n$ merkin pituisia tyhjällä syötteellä pysähtyviä ohjelmia on rajallisesti

$\Rightarrow$ jos n on niin iso, että ainakin yksi sellainen ohjelma on olemassa, niin niiden laskenta-aikojen (tyhjällä syötteellä) joukossa on suurin
  - merkitsemme sitä $BB(n)$

- jos n on niin pieni, että sellaisia ohjelmia ei ole, niin asetetaan $BB(n) = 0$
- $BB(n)$ on kasvava funktio
  - jos jokainen tasan n merkin pituinen tyhjällä syötteellä pysähtyvä ohjelma on tarpeeksi nopea, niin $BB(n) = BB(n-1)$
- olkoon $f : \mathbb{N} \mapsto \mathbb{N}$ funktio siten, että $BB(n) \leq f(n)$ kun $n \in \mathbb{N}$
- jos f voitaisiin laskea tietokoneella, niin tyhjän syötteen pysähtymistesteri saataisiin simuloimalla $f(|prog|)$ askelta ohjelmaa $prog$ $\lightning$
  - jollei $prog$ siihen mennessä pysähdy, niin se ei pysähdy koskaan

$\Rightarrow$ $BB(n)$ kasvaa niin nopeasti, että sitä ei voi laskea tietokoneella

Ratkeamattomuuteen liittyy muitakin omituisia ilmiöitä

- esim. tyhjällä syötteellä pysähtyvät ohjelmat voi luetella tietokoneella, mutta vain epäjärjestyksessä

10.2 Church–Turing-teesi

Sama tietokone pystyy suorittamaan eri ohjelmointikielillä kirjoitettuja ohjelmia

- eri kielet käännetään samalle konekielelle tai tulkataan konekielisellä tulkilla
- tietokoneen konekieli on melko köyhä
  - ei muuttujia, vaan muistipaikat ja rekisterit
  - muistipaikan osoitus vakiolla tai rekisterin sisällöllä
  - sanan kopiointi paikasta toiseen
  - yhteenlasku, vähennyslasku, kertolasku, jakolasku, bittioperaatioita
  - ehdoton hyppy, hyppy jos rekisterissä on nolla (tai negatiivinen tai . . .)
  - mahdollisesti erityisrekistereitä esim. pinon toteuttamiseksi
  - laitteistokeskeytyksiä yms.
- monet nykyisten koneiden toiminnoista ainostaan nopeuttavat ja helpottavat
  - halutaan hyödyntää kyky laittaa valtavasti transistoreja mikropiirille

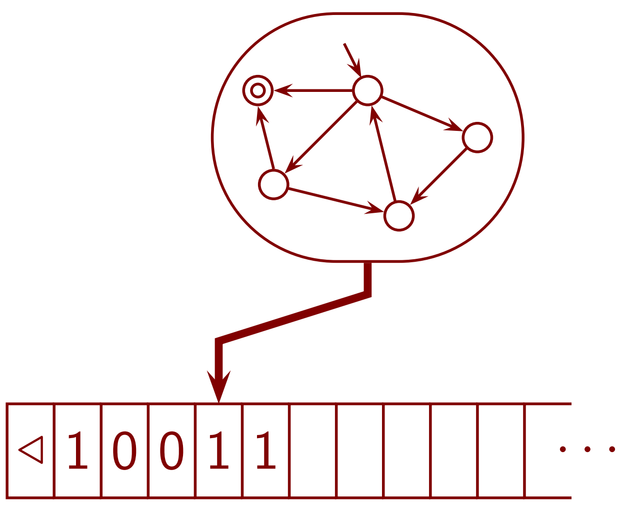
⇒ sama yksinkertainen tietokone voi suorittaa kaikki nykyiset ohjelmat, jos muistia (ja aikaa) riittää

Jokaisella yleiskäyttöisellä ohjelmointikielellä voi toteuttaa tulkin toiselle ohjelmointikielelle ja simulaattorin tietokoneelle

⇒ kaikilla ohjelmointikielillä on sama kyky ratkaista tehtäviä

- nopeus- ja muistin käytön eroja on, mutta usein vain vakiokertoimen verran

Turingin kone (**Turing machine**)



- ennen oikeita tietokoneita keksitty teoreettinen tietokone
- lukuisia muunnelmia on esitetty
  - useimmat ovat teorian kannalta yhtäpitäviä
  - jokainen on kömpelö tavalla tai toisella
  - $\Rightarrow$ mikään ei ole vakiintunut standardiksi
  - $\Rightarrow$ keskitymme yhteen
- muistina on päättymätön, ruutuihin jaettu nauha
  - kukin ruutu sisältää yhden symbolin äärellisestä aakkostosta Σ, $|\Sigma| \geq 4$
  - jokin symboli " " $\in \Sigma$ nimeltä *tyhjä* on erikoisasemassa
  - "$\triangleleft$" $\in \Sigma$ tarkoittaa nauhan vasenta päätä
  - koneen syöte on äärellinen ja kirjoitettu etukäteen nauhan alkuun
  - alkutilanteessa siitä eteenpäin nauhan jokainen symboli on " "
  - muunnelmissa nauha voi olla molempiin suuntiin päättymätön, kaksiulotteinen, . . . , nauhoja voi olla monta, . . .
- laskentayksikkö koostuu joukosta tiloja ja joukosta sääntöjä
  - kumpikin joukko on äärellinen
  - yksi tila on **alkutila** (**initial state**)
  - yksi tila f on **lopputila** tai **hyväksymistila** (**final state**, **accepting state**)
- koneessa on luku–kirjoituspää, joka on joka hetki jonkin ruudun kohdalla
  - alkutilanteessa se on ensimmäisen ruudun kohdalla

- sääntö on muotoa (q, c, q', c', δ), missä
  - q on laskentayksikön tila juuri ennen laskenta-askelta, $q \neq f$
  - c on pään kohdalla nauhalla oleva merkki juuri ennen laskenta-askelta
  - q' on laskentayksikön tila juuri laskenta-askeleen jälkeen
  - c' on pään kohdalle kirjoitettu merkki
  - $\delta \in \{"\leftarrow", "\rightarrow", "|"\}$ kertoo, että c':n kirjoittamisen jälkeen pää siirtyy ruudun verran vasemmalle tai oikealle tai pysyy paikallaan
- c:n arvo "$\triangleleft$" kertoo, että pää on nauhan vasemmassa päässä
  - sellaisella säännöllä $c' =$ "$\triangleleft$" ja $\delta \neq$ "$\leftarrow$"
  - muilla säännöillä $c' \neq$ "$\triangleleft$"
- **deterministisellä** Turingin koneella on tasan yksi sääntö jokaiselle $c \in \Sigma$ ja tilalle $q \neq f$
  - epädeterministisillä koneilla ei ole tätä vaatimusta
  - emme käsittele niitä
- laskenta etenee askel kerrallaan sääntöjen ohjaamana
  - kun $q = f$, laskenta päättyy ja vastaus on nauhan alussa
  - tätä ei välttämättä tapahdu koskaan $\Rightarrow$ laskenta ei välttämättä pääty

  $\Rightarrow$ Turingin kone laskee osittaisen funktion
- joka hetki vain äärellinen määrä nauhaa sisältää muuta kuin " "
  - laskennan edetessä tämä osa voi kasvaa rajatta

Turingin koneella voi toteuttaa esim. C++-tulkin

- tarvitaan aika monta tilaa ja sääntöä!
  - Turingin koneen ohjelmointi on hyvin kömpelöä
  - kuitenkin kaikkeen tarvittavaan löytyy toteutuskeino
- tulkki laskee hitaasti, koska aikaa menee nauhalla siirtymiseen edestakaisin
- Turingin koneelta ei muisti lopu kesken

⇒ Turingin kone voi laskea kaiken mitä ohjelmatkin

- **universaali Turingin kone** on Turingin koneena toteutettu Turingin koneiden tulkki

Vieläkin yksinkertaisemmat koneet voivat laskea kaiken minkä Turingin konekin

- kahden pinon kone: nauhan sijaan on kaksi rajatonta pinoa
  - laskenta-askel lukee jompaa kumpaa pinoa tai kirjoittaa sen päälle
  - jos laskenta-askel lukee, seuraava tila riippuu luetusta merkistä
  - tyhjästä pinosta lukeminen näyttäytyy luetun merkin erikoisarvona "$\triangleleft$"
  - lukeminen poistaa ylimmän merkin pinosta, jos pino ei ole tyhjä
  - pinoilla voi simuloida Turingin koneen pään eri puolilla olevat merkkijonot
- yhden jonon kone: nauhan sijaan on yksi rajaton jono
  - Turingin koneen pään sijainti simuloidaan erikoismerkillä jonossa
  - siirtyminen nauhalla vasemmalle simuloidaan lukemalla jonon kaikki merkit ja kirjoittamalla ne jonon perään pää yksi paikka aikaistettuna

- kahden laskurin kone: nauhan sijaan on kaksi muuttujaa, joiden arvot $\in \mathbb{N}$
  - toiminnot: kasvatus yhdellä, vähennys yhdellä, testi onko nolla
  - syöte ja tulos esitetään toisessa laskurissa koodattuna luvuksi
  - pystyy simuloimaan Turingin konetta, vaikka hitaasti

Yhden pinon kone ei laske kaikkea mitä Turingin kone

- ylimääräinen komento: lue seuraava syötemerkki pinon päälle
  - tämän ansiosta pino vapautuu työmuistiksi
  - kun syöte on loppunut, tulee jokin erikoismerkki
- yhden pinon koneen pysähtymistesti on ratkeava
  - jos laskenta ei pääty, niin on äärettömän monta hetkeä, jonka jälkeen pino ei koskaan ole matalampi kuin sillä hetkellä
  - $\Rightarrow$ pinon senhetkisellä sisällöllä ei ole vaikutusta laskennan jatkolle (paitsi ehkä erikoistapauksessa, että pino on silloin tyhjä)
  - laskentayksiköllä ja syötteen lukemisella on vain äärellisesti eri tiloja
  - $\Rightarrow$ jokin pari (syötteen tila, laskentayksikön tila) toistuu äärettömän usein siten, että pinon sisältö toistohetkellä ei vaikuta laskennan etenemiseen
  - tämän varaan voidaan rakentaa keino tunnistaa päättymätön laskenta
- yhden pinon koneella pystyy jäsentämään lausekkeita
  - esim. voidaan tarkastaa, että sulut (ja) täsmäävät

Church–Turing-teesi väittää, että Turingin kone on yleispätevä laskennan malli

- ts. mikään realistinen laskennan malli ei laske enempää osittaisia funktioita
- teesi vertaa matemaattista käsitettä intuitiiviseen käsitteeseen
  $\Rightarrow$ se ei ole todistettavissa oleva matemaattinen väite
- seuraavat havainnot tukevat teesiä:
  - ei ole löydetty vahvempaa realistista laskennan mallia
  - monet hyvin erilaiset laskennan mallit ovat todistettavasti yhtä vahvoja
  - vahvemman laskentakoneen pitää perustua fysiikan ilmiöön, jota ei voi simuloida tietokoneella edes likimäärin
- teesiä pidetään hyvin yleisesti oikeana
- teesi edellyttää rajattoman muistin
  - Turingin koneen nauha, rajaton jono, kaksi rajatonta laskuria, ...
  - oikeat tietokoneet ja ohjelmat, jos oletetaan, että muisti ei lopu
- jos rajaton muisti ei ole tarpeeksi joustava, se ei takaa Turing-vahvuutta
  - yksi pino ei ole tarpeeksi joustava
  - kaksi laskuria on tarpeeksi joustava
- voimakkaampi (ja kyseenalaisempi) teesi väittää, että realistinen laskentakone voi olla korkeintaan polynomiaalisesti nopeampi kuin Turingin kone
  - ts. jokaiselle realistiselle koneelle K on vakio k siten, että jos K laskee ajassa t, niin Turingin kone laskee ajassa $O(t^k)$
  - ei tiedetä, rikkooko kvanttitietokone tämän version teesistä

10.3 NP-täydellisyys

Tehtävän ratkaiseminen toisen tehtävän ratkaisevan ohjelman avulla

- on selvää, että kertolaskuja laskevan ohjelman avulla voi laskea neliöitä

  neliö(x) **return** kertaa(x, x)

- ehkä yllättäen neliöitä laskevan ohjelman avulla voi laskea kertolaskuja

  kertaa(x, y) **return** (neliö($x + y$) $-$ neliö($x - y$)) >> 2

  – $((x+y)^2 - (x-y)^2)$ >> $2 = ((x^2 + 2xy + y^2) - (x^2 - 2xy + y^2))/4 = xy$

- Ricen lauseen todistuksessa ratkaistiin pysähtymisongelma minkä tahansa epätriviaalin kyllä–ei-ominaisuuden ratkaisevan ohjelman avulla
- jos on annettu ohjelma, joka kertoo onko sudokulla ratkaisu, sen avulla ratkaisu voidaan löytää melko tehokkaasti
  - pannaan tyhjään ruutuun jokin numero ja kysytään, onko enää ratkaisua
  - riittää kokeilla jokainen numero jokaiseen tyhjään ruutuun kerran

$\Rightarrow$ mahdollisuudet ratkaista tehtäviä toisia tehtäviä ratkaisevien ohjelmien avulla ovat yllättävän laajat

Tehtävien luokat **P** ja **NP**

- toisinaan ratkaisun tarkastaminen näyttää paljon helpommalta kuin ratkaisun löytäminen
  - esim. sudoku
  - esim. löydettävä propositiologiikan kaavan muuttujille totuusarvot siten, että kaava tuottaa true
- usein on myös mahdollista, että ratkaisua ei ole
  - väitteen ”ratkaisua ei ole” tarkastaminen voi olla vaikeaa
- **P** on ne kyllä–ei-kysymykset, joille löytyy vastaus polynomiaalisessa ajassa
  - ts. kysymykselle on vakio k ja ajassa $O(|x|^k)$ toimiva ohjelma $V(x)$, joka tuottaa vastauksen ”kyllä” tai ”ei” syötteelle x
- **NP** on ne kyllä–ei-kysymykset muotoa ”onko tehtävälle X ratkaisua”, joille X:n ratkaisu r syötteelle x on tarkastettavissa polynomiaalisessa ajassa
  - ts. kysymykselle on vakio k ja ajassa $O(|x|^k)$ toimiva ohjelma $T(x, r)$, joka tarkastaa X:n ratkaisun r syötteelle x
  - esim. X on ”onko sudokulle ratkaisua”, x on sudoku ja r on sen ratkaisu
  - $T(x, r)$:n ”ei” ei tarkoita, että ratkaisua ei ole, vaan että r on väärin
  - aikaraja riippuu x:n eikä r:n koosta
- **P** $\subseteq$ **NP**
  - **NP**:n määritelmä toimii **P**:n tehtäville, kun valitaan sopiva epäluonteva X
- kukaan ei tiedä onko **P** $\subset$ **NP** vai **P** $=$ **NP**

NP-täydelliset tehtävät

- tehtävä X voidaan **palauttaa polynomiaalisessa ajassa** tehtävään Y, jos ja vain jos on olemassa ohjelma A siten, että
  - X:n vastaus syötteelle x on sama kuin Y:n vastaus syötteelle $A(x)$
  - $A(x)$:n suoritusaika on polynomiaalinen
- polynomin polynomi on polynomi
  - esim. jos $f(x) = x^3$ ja $g(x) = x^4$, niin $g(f(x)) = (x^3)^4 = x^{12}$

⇒ jos Y:lle on polynomiaikainen ohjelma Y, niin $Y(A(x))$ on polynomiaikainen ohjelma X:lle
  - X ratkeaa Y:n avulla niin, että Y:tä kutsutaan vain kerran ja vain lopuksi

- kyllä–ei-tehtävä on **NP-täydellinen** (**NP-complete**), jos ja vain jos
  - se on luokassa **NP**, ja
  - jokainen **NP**-tehtävä voidaan palauttaa siihen polynomiaalisessa ajassa

⇒ jos yhdellekin **NP**-täydelliselle tehtävälle on polynomiaikainen ohjelma, niin jokaiselle luokan **NP** tehtävälle on polynomiaikainen ohjelma ja **P** = **NP**

- seuraava tehtävä on **NP**-täydellinen: ”onko propositiologiikan kaavan muuttujille totuusarvot siten, että kaava tuottaa true”
  - kun x on annettu, niin ohjelman $T(x, r)$ suoritus voidaan polynomiajassa mallintaa kaavana siten, että r ilmaistaan vapaiden muuttujien avulla
- ym. tehtävä voidaan palauttaa polynomiajassa lukuisiin muihin tehtäviin
  - palautuksissa käytetään toinen toistaan nerokkaampia ohjelmia A

⇒ tunnetaan lukuisia monenlaisia **NP**-täydellisiä tehtäviä
- esim. onko annetussa graafissa silmukkaa, joka käy jokaisessa solmussa täsmälleen kerran
- esim. voiko annetut luonnolliset luvut jakaa kahdeksi ryhmäksi siten, että ryhmän 1 lukujen summa on ryhmän 2 lukujen summa
- ei tiedetä, onko millekään niistä polynomiaikaista algoritmia
- tiedetään, että jos yhdellekin on, niin jokaiselle on

- monelle **NP**-täydelliselle tehtävälle on turhaan etsitty polynomiaikaista algoritmia vuosikymmenten ajan

⇒ laajalti uskotaan, että $\mathbf{P} \neq \mathbf{NP}$

NP-täydellisyyden käytännön merkitys

- **NP**-täydelliselle tehtävälle ei kannata yrittää löytää aina nopeaa algoritmia
- voi olla mahdollista löytää algoritmeja, jotka toimivat melko usein melko hyvin
  - **NP**-täydellisyys kertoo vain, että jos $\mathbf{P} \neq \mathbf{NP}$, niin jokainen algoritmi on hidas joillakin syötteillä
- sellaisten algoritmien parantelu voi tarjota osaamiselle hyvän markkinaraon
  - monien **NP**-täydellisten tehtävien ratkaisemista tarvitaan käytännössä
  - ei ole näköpiirissä niin hyvää ratkaisua, että parantamisen tarve loppuu

Samaa tapaan voidaan määritellä monia muita vaativuusluokkia

- esim. **PSPACE** on polynomiaalisessa muistissa ratkeavat kyllä–ei-tehtävät

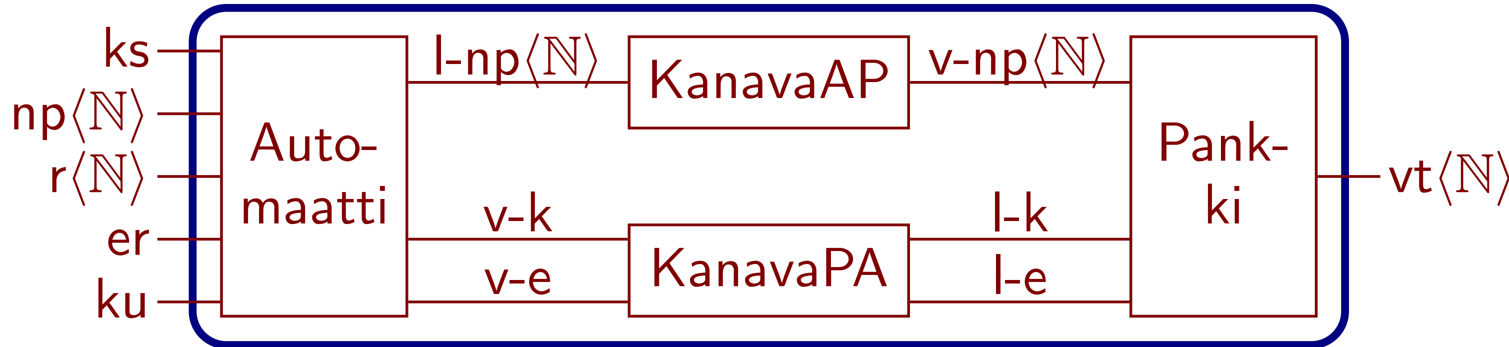
11 Kielet ja automaatit

12 Matemaattista logiikkaa

13 Rinnakkaisuus

13.1 Valmisteleva esimerkki: pankki ja pankkiautomaatti

Ensimmäinen versio

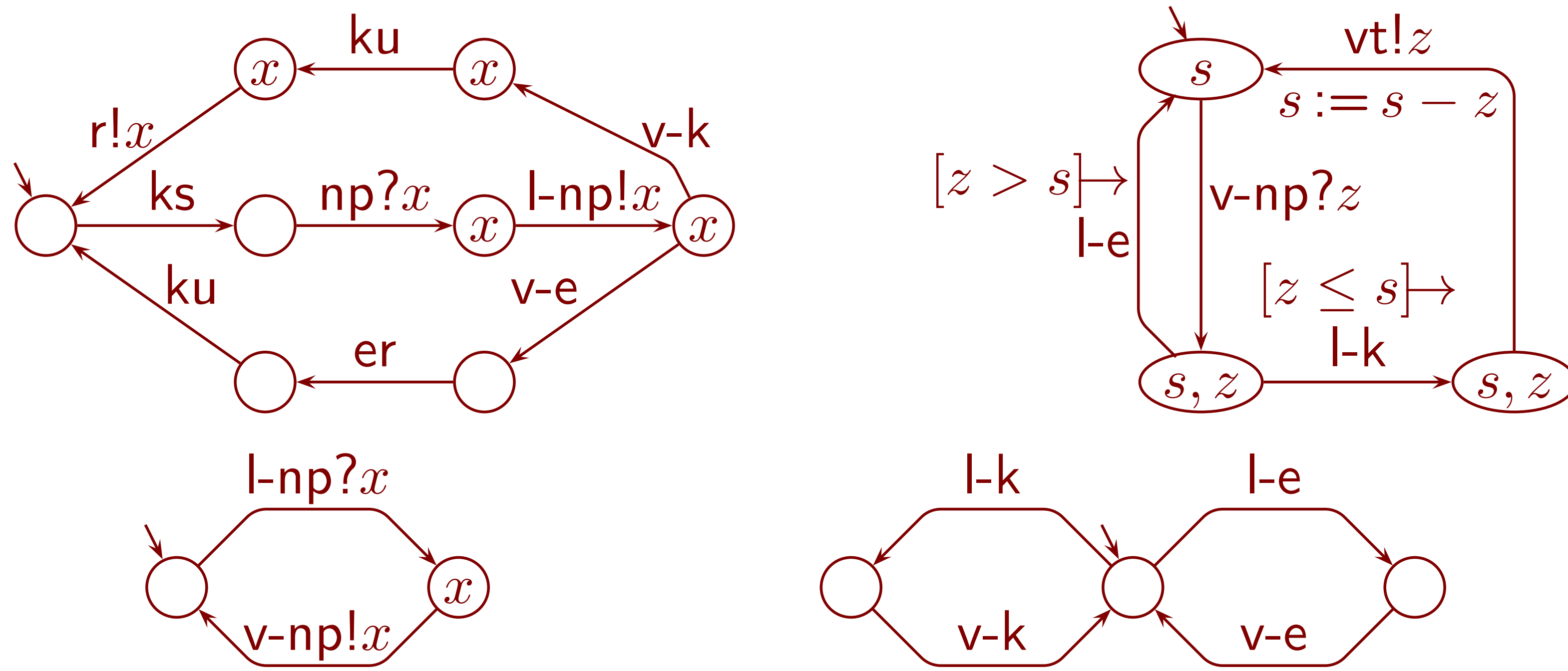


- tapahtumien nimet

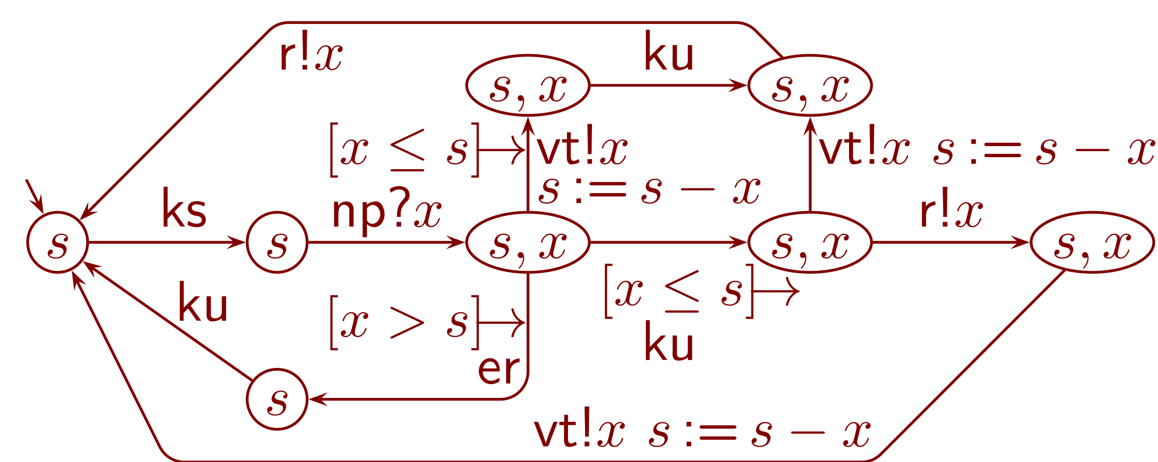
ks	=	kortti sisään	l-np	=	lähetä nostopyyntö
np	=	nostopyyntö	v-np	=	vastaanota nostopyyntö
r	=	automaatti antaa rahaa	l-k	=	lähetä kyllä
er	=	saldo ei riitä	v-k	=	vastaanota kyllä
ku	=	kortti ulos	l-e	=	lähetä ei
vt	=	veloitus tililtä	v-e	=	vastaanota ei

- $\langle \mathbb{N} \rangle$ tarkoittaa, että tapahtumaan liittyy parametri, joka on luonnollinen luku
- jätämme esittämättä asiakkaan tunnistamisen yms.

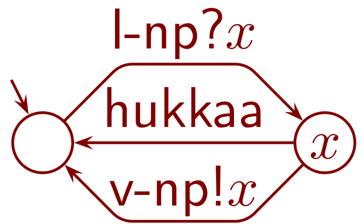
- osien käyttäytymiset



- kokonaisuuden käyttäytyminen
  - vt!x $s := s - x$ tapahtuu rinnakkain jonon ku r!x kanssa



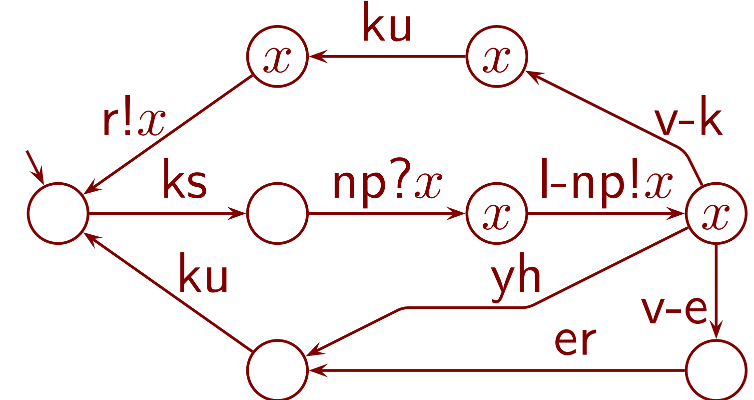
Entä jos kanavat ovat epäluotettavia?



- jos jokin viesti hukkuu, järjestelmä **lukkiutuu**

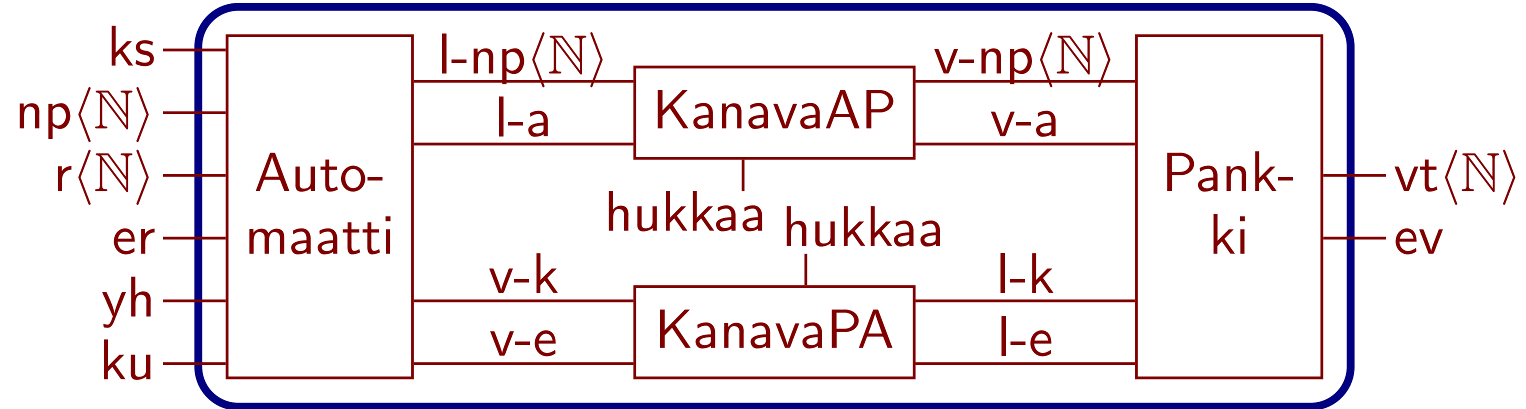
⇒ muutamme pankkiautomaattia
  - yh = yhteyshäiriö

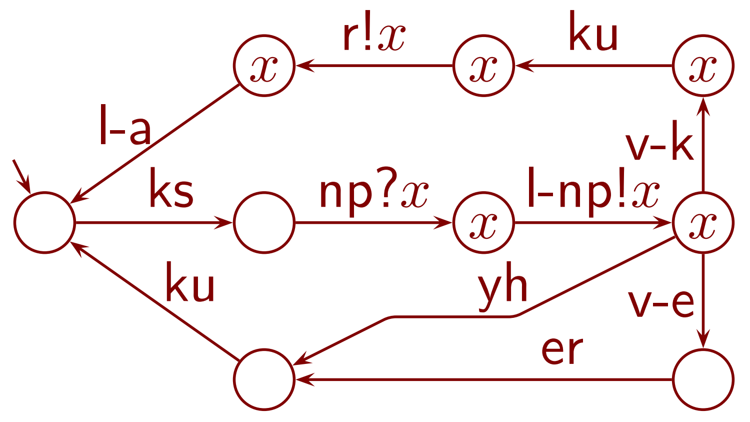
- jos kyllä-viesti katoaa, niin järjestelmä **veloittaa tiliä vaikkaa ei anna rahaa**

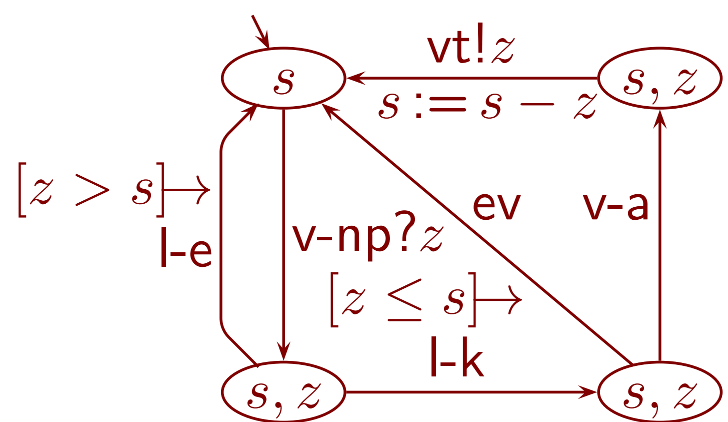


Muutamme järjestelmää niin, että tiliä veloitetaan vasta kun raha on annettu

- l-a = lähetä "raha annettu"
- ev = epävarma, pankki ei tiedä annettiinko asiakkaalle rahaa
  - voidaan käydä automaatilla tarkastamassa







- jos nostopyyntö katoaa, niin yh ja ku
- jos "kyllä" katoaa, niin yh ja ku sekä ev
- jos "annettu" katoaa, niin ku ja r!x sekä ev
  - tiliä ei veloiteta
- jos yh tapahtuu ennenaikaisesti, niin voi käydä seuraavasti:
  - nostoyritys 50 € keskeytyy, vaikka pankki sanoi "kyllä"
  - asiakas yrittää uudelleen pienemmällä summalla 20 €
  - tämä nostopyyntö katoaa
  - edelliselle pyynnölle tarkoitettu "kyllä" saapuu automaatille
  - automaatti antaa 20 € ja lähettää "annettu"
  - "annettu" saapuu pankkiin, joka veloittaa tililtä 50 €

  $\Rightarrow$ pankki veloittaa tiliä **liikaa**

$\Rightarrow$ lisäämme "annettu"-viestiin tiedon, kuinka paljon annettiin

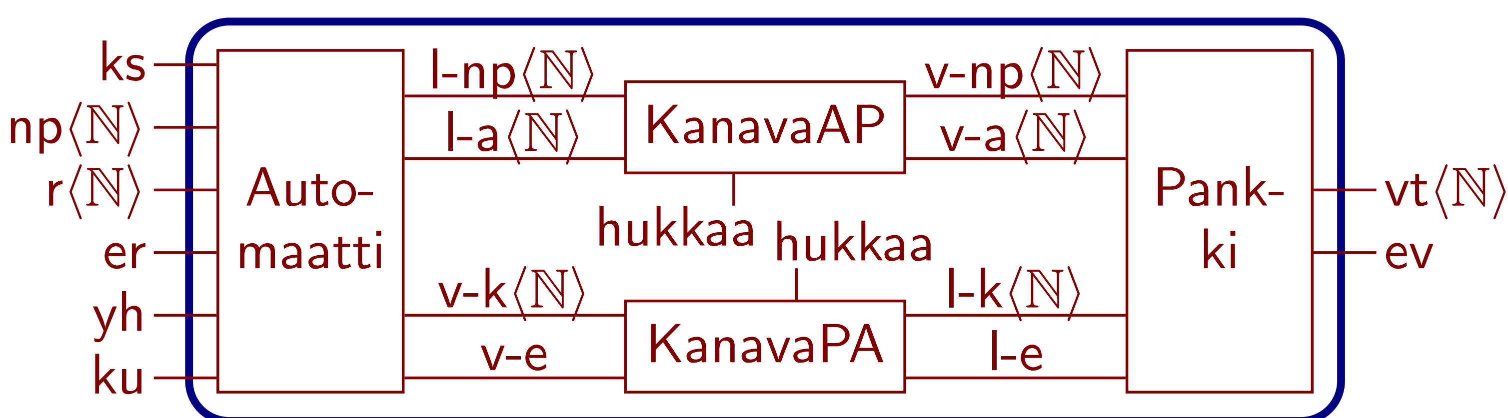
- jos asiakas yrittää nostaa ensin vähän ja yhteyshäiriön nähtyään paljon rahaa, hän voi saada enemmän kuin tilillä on
  - automaatti antaa rahat ensimmäiselle pyynnölle tarkoitetun ”kyllä” nojalla

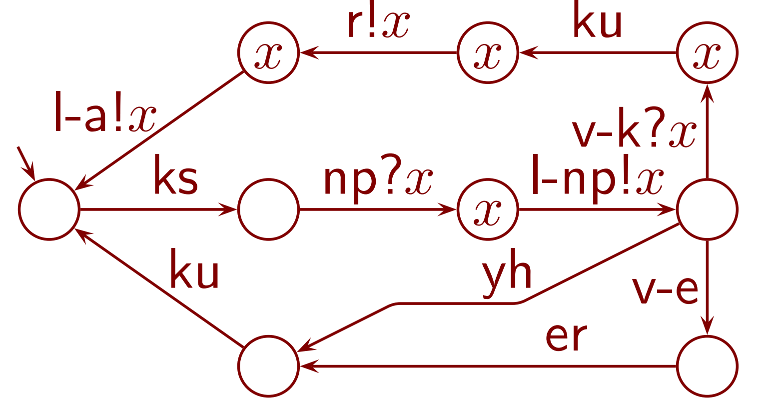
⇒ lisäämme ”kyllä”-viestiin tiedon, kuinka paljon saa antaa

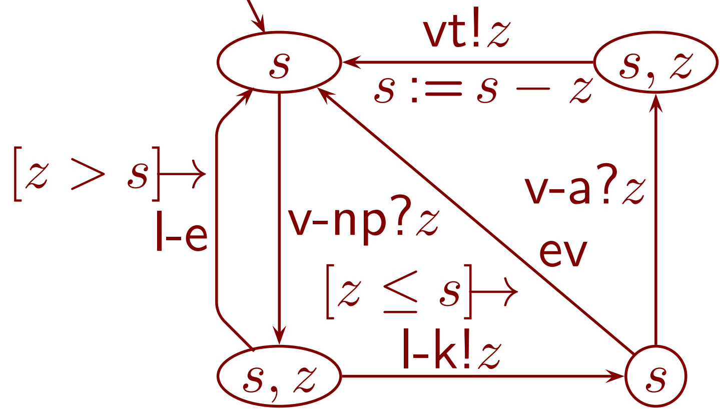
- jos olisi mallinnettu asiakkaan identiteetti, näkyisi, että väärä asiakas voi saada rahat
  ⇒ ”kyllä”- ja ”annettu”-viestissä tarvittaisiin myös tilinumero

Lopullinen versio

- KanavaPA samalla periaatteella kuin KanavaAP







13.2 Rinnakkaisjärjestelmät ja niiden virheet

Rinnakkaisjärjestelmä (**concurrent system**)

- koostuu monesta (≥ 1) yhteistoiminnassa olevasta osasta
- osat ovat tyypillisesti **vuorovaikutteisia** (**reactive**)
  - osat vuorovaikuttavat samanaikaisesti moneen (≥ 0) suuntaan
- tyypillisesti järjestelmän toiminnan ei odoteta loppuvan
  - kuljetettuaan yhden korillisen matkustajia hissi odottaa seuraavia
- tyypillisesti vuorovaikutuksista syntyy lomittaisia tapahtumaketjuja
  - hissi voi rekisteröidä uusia kutsuja toteuttaessaan edellistä
  - ne voivat vaikuttaa toisiinsa: hissi ottaa matkalla lisää matkustajia
- *huom!* sanaa "parallelism" käytetään eri yhteydessä
  - pyrittäessä suureen laskentatehoon monella prosessorilla tms.

Esimerkkejä rinnakkaisjärjestelmistä

- yhteisen resurssin suojaus ristikkäisiltä käytöiltä
  - esim. minun tulostukseni ei saa päästä kirjoittimelle ennen kuin sinun tulostuksesi on valmis, jottei samalle paperille tulisi sekaisin molempia
  - **keskinäinen poissulkeminen** (**mutual exclusion**)
- tietoliikenneprotokolla
- hissin kori, napit korissa ja kerroksissa, moottori ja ohjauslogiikka

Käyttäytyminen on usein herkkä pienille asioille

- esimerkki: ”proffa poistuu suojatieltä” ja ”rekka saapuu suojatielle”
  - sillä, kumpi tapahtuu ensin, on suuri vaikutus myöhempiin tapahtumiin
  - pieni aikaero saattaa ratkaista, kumpi tapahtuu ensin
- usein mukana on oltava oletuksia käyttäjistä
  - esim. kirjasto ei voi taata, että jokainen halukas saa lopulta kirjan, jos joku lainaaja hukkaa sen
  - käyttäjistä pitää kuitenkin olettaa mahdollisimman vähän

⇒ täytyy olettaa, että valintaa vaihtoehtojen välillä ei tunneta
  - joudutaan hyväksymään **epädeterminismi** (**nondeterminism**)
  - tämä ei tarkoita kannanottoa, että todellisuus olisi epädeterministinen
  - tämä tarkoittaa kannanottoa, että ei tiedetä kaikkia olennaisia asioihin vaikuttavia tekijöitä

⇒ täytyy varautua (melkein) kaikkiin vaihtoehtoihin

- **ihmisen on hyvin vaikea hallita ja hahmottaa niitä**
  - ihminen pystyy hyvin seuraamaan yksittäisen tapahtumakulun
  - ihminen ei ole hyvä mieltämään vaihtoehtoisten tapahtumakulkujen kokonaisuutta muuten kuin seuraamalla yksittäisiä tapahtumakulkuja
  - vaihtoehtoisia tapahtumakulkuja on usein ihmisille aivan liikaa

⇒ tarvitaan muita keinoja kuin yksittäisten tapahtumakulkujen miettiminen

- kun jotain menee pieleen, se voidaan yleensä esittää tapahtumakulkuna

Rinnakkaisjärjestelmille tyypillisiä virheitä

- yhteisen tiedon muuttaminen aiheuttaa helposti virheen
- esimerkki
  - ohjelma 1 kopioi itselleen tilin saldon 987 € palkanmaksua varten
  - ohjelma 2 kopioi itselleen tilin saldon, koska omistaja nosti 20 €
  - ohjelma 1 lisää tilin saldoon (sen omaan kopioonsa) palkan 1000 €
  - ohjelma 2 vähentää saldosta (sen omasta kopiosta) 20 €
  - ohjelma 1 kopioi muuttamansa saldon 1987 € paikalleen
  - ohjelma 2 kopioi muuttamansa saldon 967 € paikalleen

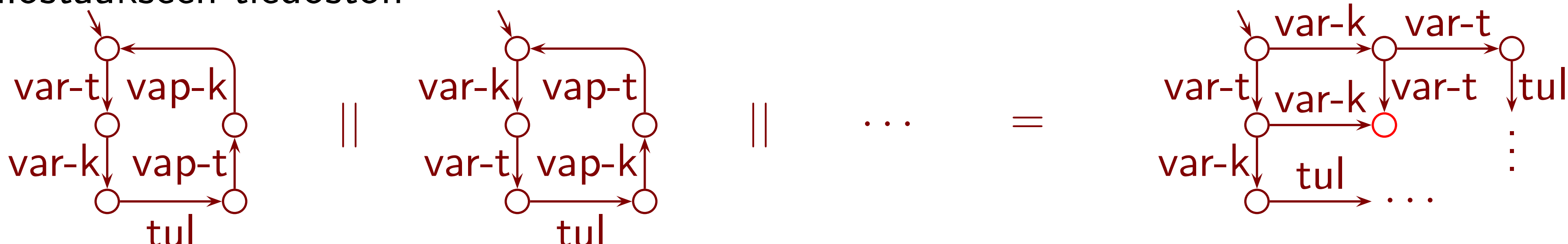
  ⇒ saldoksi jää 967 €, **palkka katosi**

⇒ käytetään mekanismeja, jotka päästävät vain yhden kerrallaan käyttämään tietoa
  - keskinäinen poissulkeminen
  - (voidaan sallia monta yhtäaikaa, jos jokainen vain lukee)

- kun yksi käyttää yhteistä tietoa, muut halukkaat joutuvat odottamaan
- käyttäjä voi puolestaan odottaa jotain muuta
  - esim. palkanmaksuohjelma odottaa työnantajan tilin vapautumista

⇒ voi syntyä **lukkiuma** (**deadlock**)
  - jokainen odottaa, että jokin muu tekisi ensin jotain

- esim. kaksi käyttäjää varaa eri järjestyksissä tiedoston ja kirjoittimen tulostaakseen tiedoston



- klassinen esimerkki on ruokailevat filosofit (dining philosophers)
  - pyöreän pöydän ympärillä on tasavälein viisi filosofia
  - jokaisen edessä on lautasellinen kiinalaista ruokaa
  - jokaisessa kahden lautasen välissä on yksi syömäpuikko
  - filosofi mietiskelee, ottaa puikon vasemmalta puoleltaan (tarvittaessa odottaa kunnes se on vapaa), ottaa/odottaa puikon oikealta, syö, palauttaa puikon vasemmalle, palauttaa oikealle ja jatkaa alusta
- lukkiumilta voi suojautua ajastimilla
  - käytetään myös sanaa **vahtikoira** (**watchdog**)
  - kun odotus on ylittänyt määräajan, se keskeytetään ja tehdään jokin toipumistoimenpide
  - pankkiautomaattiesimerkissä ”yhteyshäiriö” ja ”epävarma”
- toipumistoimenpide on vaikea suunnitella sellaiseksi, että mikään tieto ei sotkeudu tms.

- ajastin voi laueta, vaikka odotettu asia on tulossa
  - odotettu asia ei ollut estynyt, vaan ainoastaan viivästynyt
  - toipuminen käynnistyy, vaikka odotettu asia toteutuukin

  ⇒ vaara, että järjestelmä tekee ristiriitaisia toimintoja

Töiden tekemisjärjestystä ei saa sitoa liikaa

- esim. täytyy voida tehdä ensin kiireellisiä töitä
- kuitenkaan mikään työ ei saa joutua odottamaan loputtomasti
  - tarvitaan **reiluutta** (**fairness**) joitakin tai kaikkia kohtaan
  - tarvitaan reiluutta viestin läpimenoa mutta ei tarvita hukkaamista kohtaan

Rinnakkaisjärjestelmiä on hyvin vaikea saada toimimaan oikein

13.3 Kaksi klassista hyvin toimivaa järjestelmää

Petersonin algoritmi kahdelle asiakkaalle

- tehtävä on varmistaa, että
  - asiakkaat eivät ole yhtäaikaa **kriittisissä osissaan** (**critical section**)
  - kriittiseen osaansa haluava asiakas lopulta pääsee sinne (olettaen, että mikään ei jää sinne loputtomaksi aikaa)

Asiakas1	Asiakas2
...	...
p_1 := true	p_2 := true
$v := 2$	$v := 1$
odota kunnes $p_2 =$ false $\vee v = 1$	**odota kunnes** $p_1 =$ false $\vee v = 2$
... kriittinen osa	... kriittinen osa
p_1 := false; **goto** 1	p_2 := false; **goto** 1

- tarkastellaan hetkeä, kun As.2 saapuu kriittiseen osaan As.1:n jo ollessa siellä
  - $p_1 =$ true, joten $v = 2$

  ⇒ edellinen askel ei voinut olla Asiakas2:n $v := 1$
  - mutta koska $p_2 =$ true ja $v = 2$, se ei voinut olla As.1:n saapuminenkaan

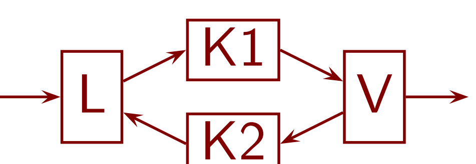
  ⇒ edellista askelta ei voi olla ↯
  - sama pätee asiakkaiden numerot vaihdettuina

  ⇒ **asiakkaat eivät pääse yhtäaikaa kriittiseen osaan**

- voiko Asiakas2 joutua odottamaan loputtomasti?
  - Asiakas1 ei saa jäädä kriittiseen osaan loputtomaksi aikaa
  - jos Asiakas1 ei ole sinne mennytkään tai tulee pois pitkäksi aikaa, $p_1 =$ false päästää Asiakas2:n kriittiseen osaan
  - jos Asiakas1 yrittää nopeasti uudelleen, niin $v = 2$ päästää Asiakas2:n
  - sama pätee asiakkaiden numerot vaihdettuina

  ⇒ **odottava asiakas pääsee lopulta kriittiseen osaan**

Vuorottelevan bitin protokolla



- viestejä voi hävitä kanavassa

⇒ viesti lähetetään tarvittaessa moneen kertaan
  - saatuaan viestin vastaanotin lähettää **kuittauksen** (**acknowledgement**)
  - jos lähetin ei saa kuittausta määräaikana, se lähettää viestin uudelleen

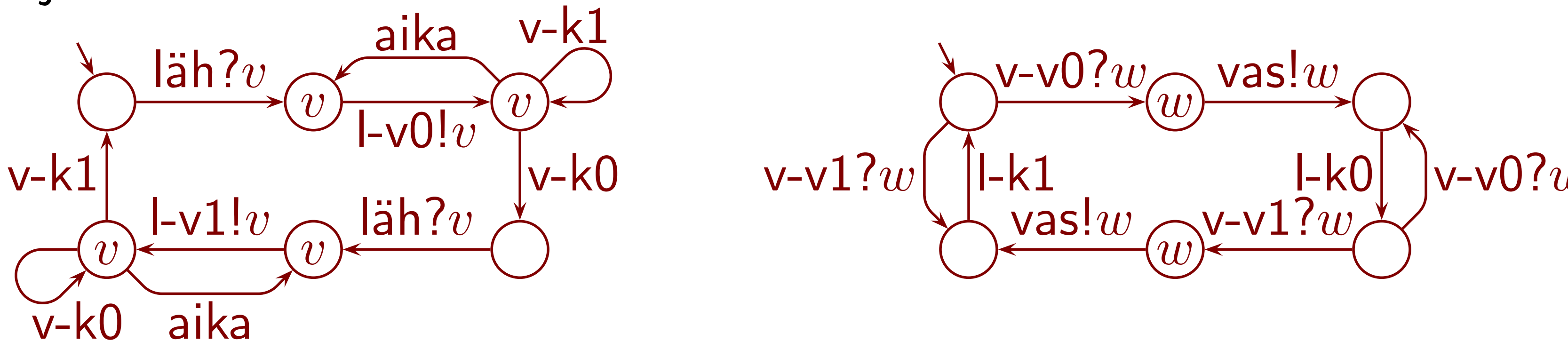
- ennenaikainen uudelleenlähetys voi johtaa viestin kahdentumiseen
  - mukavaa, jos viesti on palkan maksaminen tilille
  - ikävää, jos viesti on nosto tililtä
- edellisen viestin kopiota ei aina voi tunnistaa sisällön perusteella
  - uusi viesti voi olla sisällöllisesti sama kuin edellinen
  - esim. kaksi samansuuruista maksua samalle tilille

⇒ liitetään viestiin bitti, joka vaihtuu aina, kun lähetetään uusi viesti
  - (joissakin muissa protokollissa bitin sijaan on isompi lukualue)

- myös kuittauksessa on bitti, jotta ei kävisi seuraavasti:
  - lähetin lähettää viestin A, joka saapuu vastaanottimeen
  - lähetin lähettää viestin A uudelleen
  - vastaanotin lähettää ja lähetin saa kuittauksen ensimmäisestä viestistä A
  - toinen viesti A saapuu vastaanottimeen
  - lähetin lähettää uuden viestin B, joka katoaa
  - vastaanotin lähettää ja lähetin saa kuittauksen toisesta viestistä A
  - lähetin tulkitsee sen kuittaukseksi viestistä B

  ⇒ vaikka B katosi, lähetin ei lähetä sitä uudelleen

  ⇒ **viesti B katosi lopullisesti**
  - vanhentuneen viestin aiheuttama virhe kuten pankkiautomaattiesimerkissä
- lähetin ja vastaanotin



- protokolla ei lukkiudu, koska
  - jokaisessa tilassa lähetin lähettää tai odottaa asiakasta tai ajastinta
  - jokaisessa tilassa vastaanotin lähettää tai odottaa uutta viestiä
- vastaanotin estää viestien kahdentumisen

- lähetin ei käytä samaa bittiä uudelleen ennen kuin edellinen bitti kiersi ympäri
  - silloin ei ole jäljellä vanhoja viestejä tai kuittauksia sitä edellisellä bitillä

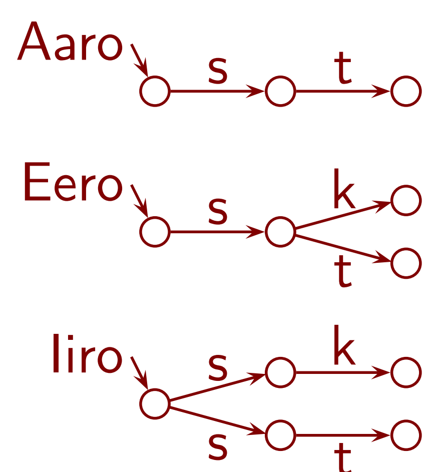
  $\Rightarrow$ kuittaus uudelleen käytetyllä bitillä on varmasti uusimman viestin kuittaus

  $\Rightarrow$ lähetin ei luovu lähettämisyrityksistä liian aikaisin

  $\Rightarrow$ protokolla ei hukkaa viestejä (jos kanava ei hukkaa niitä loputtomiin)
- jos kanava hukkaa loputtomiin viestejä tai toinen kanava kuittauksia, niin vuorottelevan bitin protokolla jää ikuiseen silmukkaan
- muussa tapauksessa vuorottelevan bitin protokolla **välittää kaikki viestit perille luotettavasti ja kahdentamatta**

14 Rinnakkaisuuden teoriaa

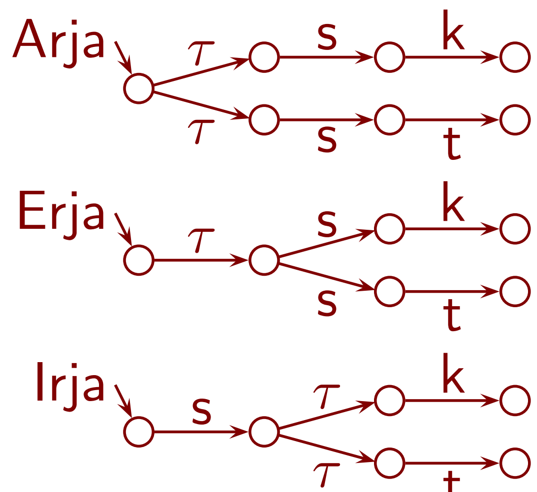
14.1 Valmistelevia esimerkkejä



Käyttäytyvätkö seuraavat samoin?

Aaro saapuu kahvilaan ja ottaa teetä
Eero saapuu kahvilaan ja ottaa kahvia tai teetä
Iiro saapuu kahvilaan ja ottaa kahvia tai teetä

- millä tahansa järkevillä kriteereillä Aaro käyttäytyy eri tavalla kuin muut
- Eeron ja Iiron ero tulee ilmi, jos kahvi on loppu mutta teetä jäljellä
  - Eero ottaa varmasti teetä
  - jos Iiro oli valinnut kahvin, hän lukkiutuu



Käyttäytyvätkö seuraavat samoin?
Entä verrattuna Eeroon ja Iiroon?

Arja valitsee ennen kahvilaan saapumista
Erja valitsee saapuessaan kahvilaan
Irja valitsee saavuttuaan kahvilaan

- τ tarkoittaa tapahtumaa, jota ulkopuolinen tarkkailija ei näe
  - **näkymätön tapahtuma**
  - näkymättömällä tapahtumalla voi olla havaittavia seurauksia

- kuten Iiro, myös Arja, Erja ja Irja voivat lukkiutua, jos kahvia ei ole
  ⇒ erilaisia kuin Aaro ja Eero

Näkymättömien tapahtumien määrä ajatellaan yleensä epäolennaiseksi (kunhan se on äärellinen)

- ei ole olennaista, tekikö pankkiautomaatti 1724 vai 5296 toimintoa ennen kuin antoi rahaa
- isomman näkymättömien tapahtumien määrän voi ajatella edustavan pitempää odotusta, mutta
  - erilaisten näkymättömien tapahtumien ei voi olettaa kestävän yhtä kauan

  ⇒ näkymättömien tapahtumien määrä ei ole kunnollinen ajan mitta

  ⇒ aika pitää lisätä teoriaan toisilla, paremmilla keinoilla

⇒ Erja ja Iiro katsotaan samanlaisiksi

- joissakin teorioissa on eräästä matemaattisesta syystä tärkeää, voidaanko heti alussa suorittaa näkymätön tapahtuma
  - niissä Erja ja Iiro katsotaan erilaisiksi

Valinnan vaikutus tulee näkyväksi vasta kun valitsija yrittää ottaa kahvia tai teetä

⇒ osa teorioista katsoo Arjan, Erjan ja Irjan samanlaisiksi, osa erilaisiksi

- esittelemme hieman eri teorioiden aineksia

14.2 Vuorovaikutus ja rinnakkaisjärjestelmien esittäminen

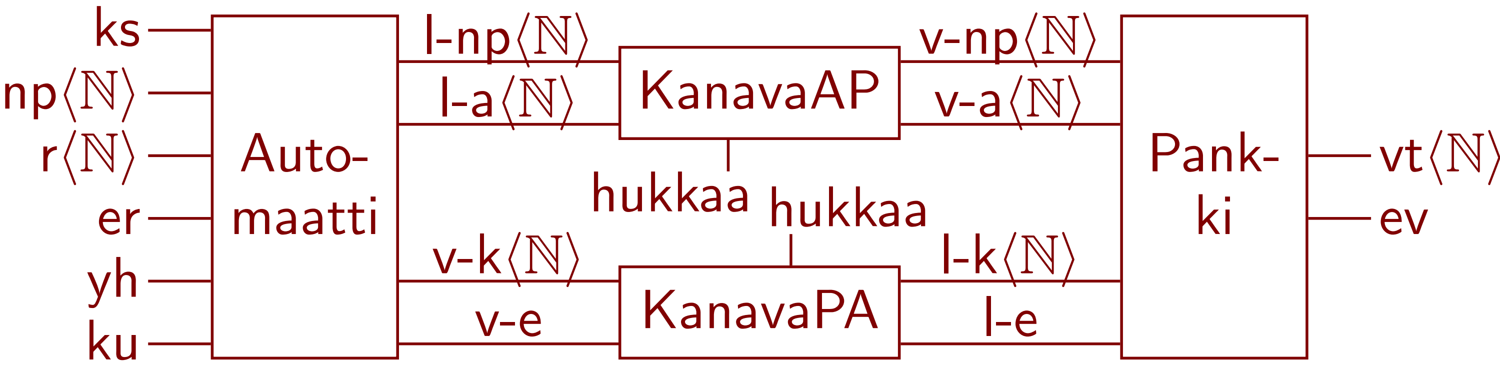
Lukuisia erilaisia viestinnän ja vuorovaikutuksen malleja on ehdotettu

- yhteinen muisti
  - liian alatason mekanismi sekä käytön että teorian kannalta
- first-in-first-out- eli fifo-jonot
  - mahtuuko jonoon rajattomasti viestejä?
  - jollei, mitä tapahtuu kun yritetään kirjoittaa täyteen jonoon?
  - lukeeko ja odottaako vastaanottaja yhtä jonoa vai useaa?
  - jos yhtä, mitä tehdä, jos odotettu viesti ei olekaan ensimmäisenä?
  - jos useaa, miten se päättää mitä kulloinkin odottaa ja mistä lukee?
- ...

Synkroninen vuorovaikutus on osoittautunut erinomaiseksi teorian pohjaksi

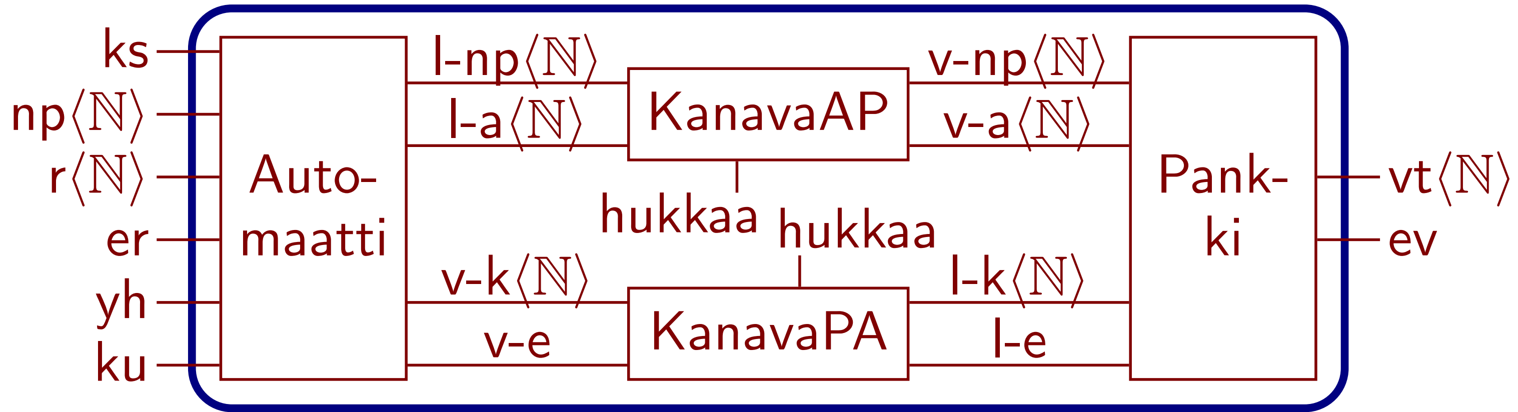
- mahdollistaa mielekkään tuloksekkaan teorian
- jokseenkin kaikki muut vuorovaikutustavat voi esittää sen avulla
- toiminta
  - kutsumme järjestelmää ja sen osia ja osakokonaisuuksia **prosesseiksi**
  - jokaiseen tapahtumaan osallistuu yksi tai useampia prosesseja
  - jos jokin osallistuja ei ole valmis tapahtumaan, tapahtuma estyy
  - kaikki osallistujat suorittavat tapahtuman yhtäaikaa

Järjestelmän rakenne voidaan esittää kuvana



- esimerkkejä
  - tapahtumaan ks (kortti sisään) osallistuu vain Automaatti
  - l-np (lähetä nostopyyntö) osallistuvat Automaatti ja KanavaAP
  - Pankki katsoo yksinään näkymättömästi, onko tilillä rahaa

Osakokonaisuuden ympärille piirretyllä laatikolla voi ilmaista, että laatikon sisäisiä tapahtumia ei näytetä ulos



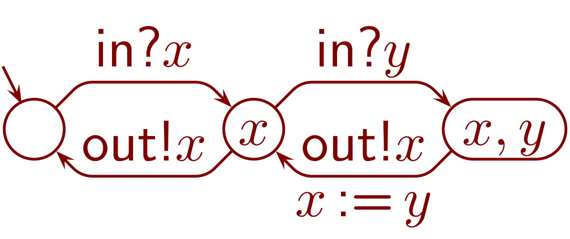
- PankkiJ = (Automaatti || KanavaAP || KanavaPA || Pankki) \ {l-np*, v-np*, l-a*, v-a*, l-k*, v-k*, l-e, v-e, hukkaa}

- prosessin **aakkosto** (**alphabet**) on niiden näkyvien tapahtumien joukko, joihin se voi synkronoitua
  - merkitään usein symbolilla Σ, Σ', Σ_1, tms.
  - τ ei kuulu aakkostoon
  - rakennekuvassa laatikosta lähtevien viivojen nimet parametreineen
  - aakkostossa voi olla nimiä, jotka eivät esiinny käyttäytymiskuvassa
- $||$ esittää **rinnankytkentää** (**parallel composition**)
  - kukin osaprosessi voi suorittaa näkymättömiä tapahtumia itsekseen
  - näkyvään tapahtumaan osallistuu jokainen, jonka aakkostoon se kuuluu
- **kätkentä** (**hiding**) $P \setminus A$ muuttaa A:ssa luetellut P:n tapahtumat τ:ksi
- näkyvä nimi voidaan muuttaa **uudelleennimeämisellä** (**renaming**) $[b_1/a_1, \cdots, b_n/a_n]$

Esimerkissä kanava mallinnettiin prosessina, joka vuorovaikuttaa erikseen lähettäjän ja erikseen vastaanottajan kanssa

- kapasiteetin 2 kanava kahdella erilaisella viestillä
  - tapahtumat on koodattu väreillä: l-1, v-1, l-2 ja v-2
- jokseenkin kaikki muut vuorovaikutusmekanismit voi mallintaa vastaavasti

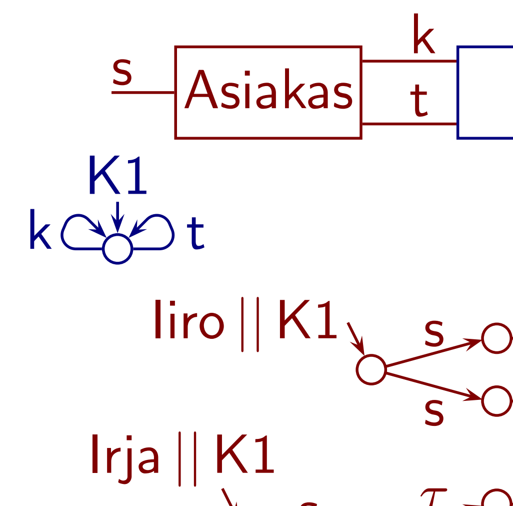
Säännöllisiä tila- ja kaarikokoelmia voi esittää lyhennemerkinnöillä

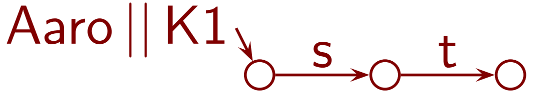


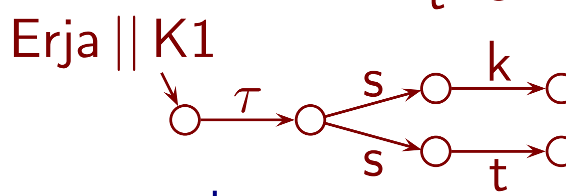
- $?x$ ja $!x$ ovat lyhennemerkintöjä tapahtumakokoelmille

Esimerkkejä

- kahvila, jossa on aina tarjolla sekä kahvia että teetä
  - Aaro || K1 jne. käyttäytyvät kuten Aaro jne. edellä

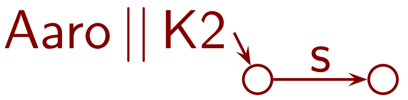


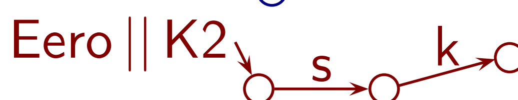


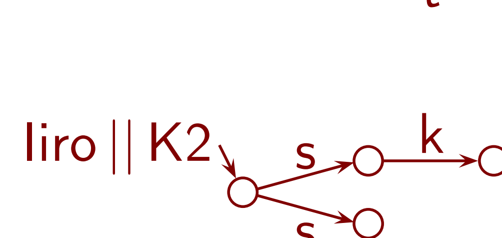


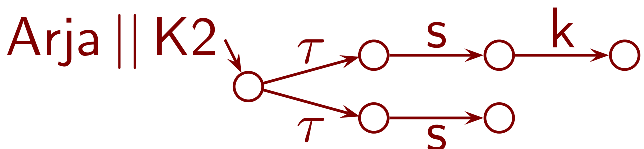
Arja || K1

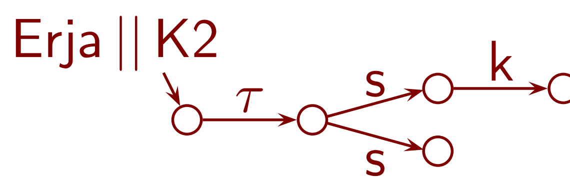
- kahvila, jossa on tarjolla vain kahvia

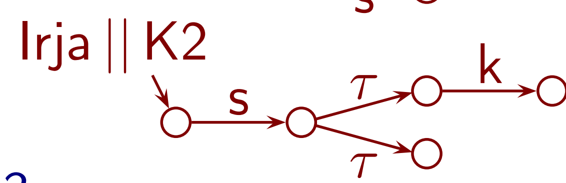




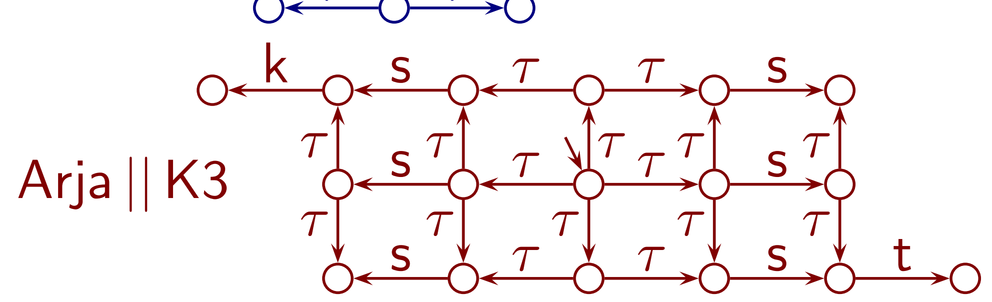


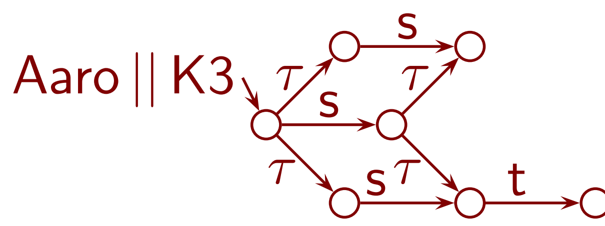






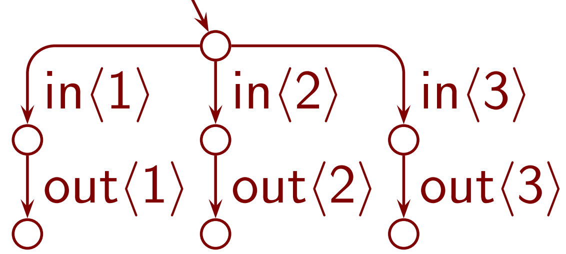
- kahvila, jossa myyjä päättää mitä asiakas saa ottaa

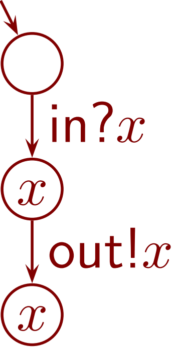




Input ja output

- vastaanottaja on valmis mihin tahansa arvoon joukosta vaihtoehtoja
  - voi mennä arvon mukaan eri tiloihin
  - ⇒ muistaa saamansa arvon

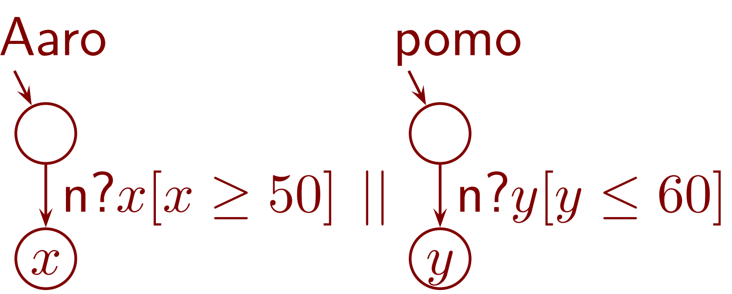




- lähettäjä tarjoaa vain yhtä arvoa ko. joukosta
  - ⇒ vuorovaikutuksessa toteutuu se arvo

⇒ input ja output ovat vain tapoja osallistua synkroniseen vuorovaikutukseen

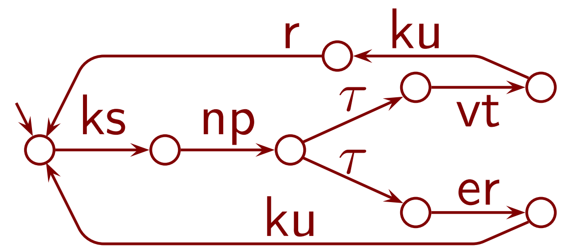
- yleisempi esimerkki
  - Aaro vaatii vähintään 50 € palkankorotusta ja pomo suostuu enintään 60 €
  - ⇒ jokin arvo väliltä 50 €, ..., 60 € toteutuu, ja molemmat ovat samaa mieltä, mikä arvo



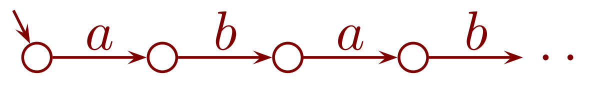
- toinen esimerkki
  - Aaro vaatii vähintään 60 € palkankorotusta ja pomo suostuu enintään 50 €
  - ⇒ molemmille sopivaa arvoa ei ole ja neuvottelu ei johda tulokseen

⇒ on olemassa muitakin vuorovaikutuksen muotoja kuin input ja output

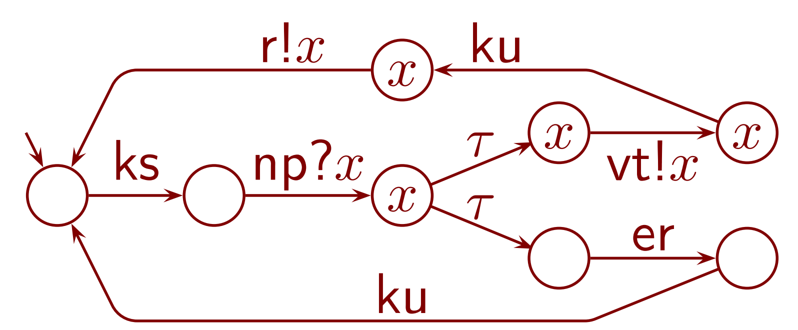
Käyttäytymiskuvista



- olemme esittäneet prosessin käyttäytymisen suunnattuna graafina, jonka kaaret on nimetty ja jolla on juuri
  - prosessi on osa, osakokonaisuus tai koko järjestelmä
- sen nimi on **nimetty siirtymäjärjestelmä** (**labelled transition system**), **LTS**
- LTS on nelikko $(S, \Sigma, \Delta, \hat{s})$, missä
  - S on **tilojen** (**state**) joukko
  - Σ on aakkosto, vaaditaan $\tau \notin \Sigma$
  - Δ on **siirtymien** (**transition**) joukko, vaaditaan $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$
  - $\hat{s}$ on **alkutila** (**initial state**), vaaditaan $\hat{s} \in S$
- muuten sama käsite kuin äärellinen automaatti, paitsi
  - sen ei tarvitse olla äärellinen
  - lopputilan käsitettä ei ole (kaikki tilat voi ajatella lopputiloiksi)



- τ ei ole sama kuin tyhjän merkkijonon symboli ε
  - τ on tapahtuma (joskin näkymätön) eikä tyhjä jono tapahtumia
- lyhennemerkityille kuville, joissa on muuttujia, ei ole vakiintunutta nimeä
  $\Rightarrow$ kutsumme **tilakoneiksi** (**state machine**)
- jokainen LTS on samalla tilakone, mutta vastakkainen ei päde



Yhdistettyjen prosessien käyttäytyminen

- $||$, $\backslash$ ja $[\cdots]$ ottavat LTS:iä/LTS:n ja tuottavat LTS:n
- $||$ on vaihdannainen ja liitännäinen (modulo isomorfismi)
- pätee mm.
  - $(P \setminus A) \setminus B \;=\; P \setminus (A \cup B)$
  - jos $A \cap \Sigma_2 = \emptyset$, niin $(P_1 \,||\, P_2) \setminus A \;=\; (P_1 \setminus A) \,||\, P_2$
- jokainen (vain) $||$:lla, $\backslash$:lla ja $[\cdots]$:lla kootun lausekkeen $f(P_1, P_2, \ldots, P_n)$ siirtymä on jompikumpi seuraavista:
  - yksi prosessi suorittaa τ-siirtymän, muut pysyvät paikoillaan, ulos τ
  - säännön muotoa $(a_1, a_2, \ldots, a_n; b)$ mukainen, missä

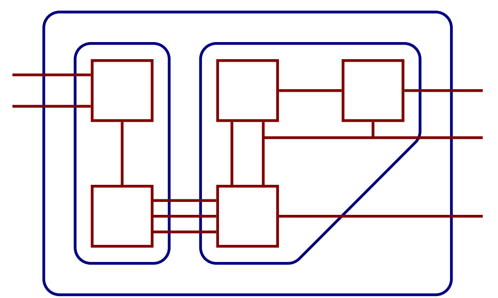
    $a_i \in \Sigma_i \cup \{"\text{-}"\}$ $\qquad\qquad$ $"\text{-}" \notin \Sigma_i$

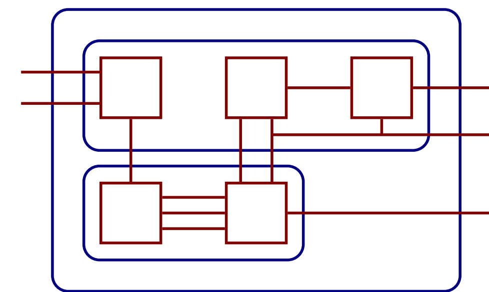
    jos $a_i =$ "-", niin P_i pysyy paikallaan

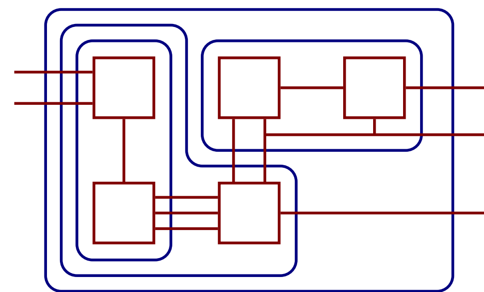
    jos $a_i \in \Sigma_i$, niin P_i osallistuu a_i-siirtymällä

    ulos b, joka on τ tai näkyvä

$\Rightarrow$ järjestelmän rakenne voidaan ryhmitellä uudelleen monin tavoin ilman, että käyttäytyminen muuttuu



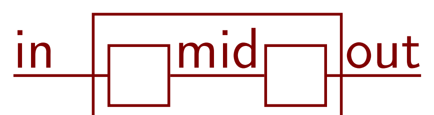




Operaattoreiden matemaattiset määritelmät

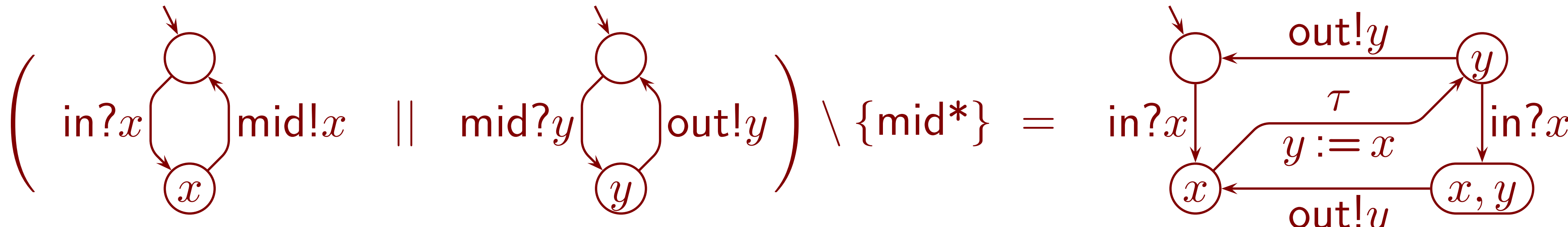
- olkoon $L = (S, \Sigma, \Delta, \hat{s})$, $L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1)$ ja $L_2 = (S_2, \Sigma_2, \Delta_2, \hat{s}_2)$
- $L \setminus A = (S', \Sigma', \Delta', \hat{s}')$, missä
  - $S' = S$ ja $\hat{s}' = \hat{s}$
  - $\Sigma' = \Sigma \setminus A$
  - $\Delta' = \{(s, a, s') \mid (s, a, s') \in \Delta \land a \notin A\} \cup$
    $\{(s, \tau, s') \mid \exists a \in A : (s, a, s') \in \Delta\}$
- $L[b_1/a_1, \ldots, b_n/a_n] = (S', \Sigma', \Delta', \hat{s}')$, missä
  - vaaditaan, että $a_i \neq \tau \neq b_i$ kun $1 \leq i \leq n$
  - $S' = S$ ja $\hat{s}' = \hat{s}$
  - $\Sigma' = \{a \in \Sigma \mid \forall i; 1 \leq i \leq n : a \neq a_i\} \cup \{b \mid \exists i; 1 \leq i \leq n : a_i \in \Sigma \land b = b_i\}$
  - $\Delta' = \{(s, a, s') \in \Delta \mid \forall i; 1 \leq i \leq n : a \neq a_i\} \cup$
    $\{(s, b, s') \mid \exists i; 1 \leq i \leq n : (s, a_i, s') \in \Delta \land b = b_i$
- $L_1 \,||\, L_2 = (S', \Sigma', \Delta', \hat{s}')$, missä
  - $\Sigma' = \Sigma_1 \cup \Sigma_2$ ja $\hat{s}' = (\hat{s}_1, \hat{s}_2)$
  - S' ja Δ' ovat pienimmät joukot siten, että $\hat{s}' \in S'$ ja jos $(s_1, s_2) \in S'$ ja jokin seuraavista toteutuu, niin $(s'_1, s'_2) \in S'$ ja $((s_1, s_2), a, (s'_1, s'_2)) \in \Delta'$:
    $(s_1, a, s'_1) \in \Delta_1$, $a \notin \Sigma_2$ ja $s'_2 = s_2$
    $(s_2, a, s'_2) \in \Delta_2$, $a \notin \Sigma_1$ ja $s'_1 = s_1$
    $(s_1, a, s'_1) \in \Delta_1$, $(s_2, a, s'_2) \in \Delta_2$ ja $a \in \Sigma_1 \cap \Sigma_2$
- määritelmien yksityiskohdissa on otettu huomioon osien τ-siirtymät

Muuttujan käyttö tilakoneessa voidaan esittää rinnankytkentänä



⇒ lisää mahdollisuuksia ryhmitellä järjestelmä

- $\|$ ym. voidaan laskea ilman, että muuttujien lyhennemerkinnät on avattu

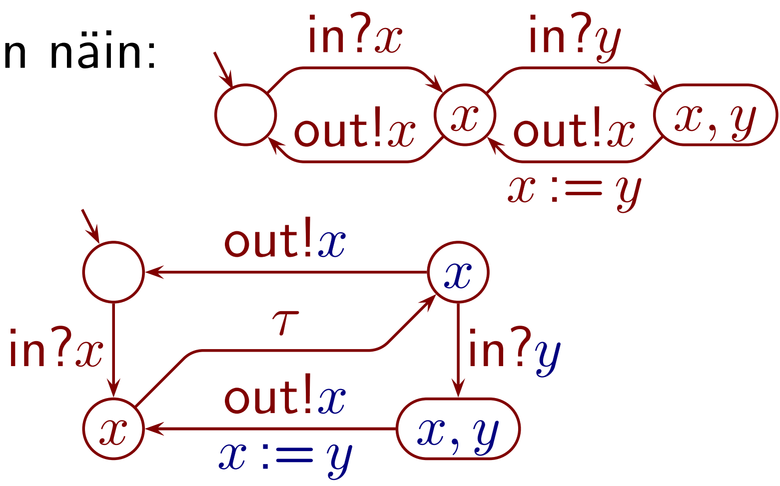


Siis

- sekä järjestelmän että sen osan käyttäytyminen voidaan esittää LTS:nä
- (osa)kokonaisuuden LTS voidaan laskea osien LTS:istä
- muuttujat voidaan avata ennen ja/tai jälkeen käyttäytymisten yhdistämistä

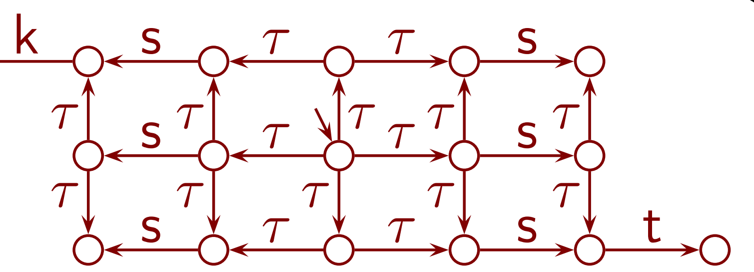
Kapasiteetin 2 fifo esitettiin aikaisemmin näin:

- onko se sama kuin äsken saatu?
- muuttujien uudelleenjärjestely äsken saadussa tuottaa tämän:
- tutkimme piakkoin, voiko kuvan τ-kaaren poistaa



Tilaräjähdys (state explosion)

- rinnankytkennässä syntyy usein paljon tiloja
  - esim.



  - jos $|S_1| = \ldots = |S_k| = n$, niin $L_1 \,||\, \cdots \,||\, L_k$ voi sisältää n^k tilaa
- myös muuttujien avaamisesta tulee usein paljon tiloja, jopa äärettömästi
  - esim. 16-bittinen kokonaisluku $\leadsto$ 65 536 tilaa
- tilaräjähdys on iso ongelma rinnakkaisjärjestelmien tarkastamisessa
- tarkastaminen rinnakkaisvirheiden varalta on laskennallisesti vaativaa
  - voiko järjestelmä suorittaa väärän tapahtuman on tavallisilla rinnakkaisjärjestelmien malleilla **PSPACE**-kova tehtävä
  - samoin voiko järjestelmä lukkiutua

$\Rightarrow$ tilaräjähdys ei viime kädessä johdu tyhmyydestämme, vaan rinnakkaisilmiöiden monimutkaisuudesta

14.3 Jälkisemantiikka

Usein järjestelmän LTS:ssä on lukuisia τ-kaaria

- vrt.
- τ esittää näkymätöntä tapahtumaa

⇒ haluamme poistaa τ-kaaret

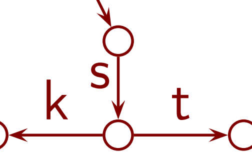
Arja || K3

Prosessin P **jälkisemantiikka** (**trace semantics**) sisältää kaksi osaa:

- prosessin aakkosto $\Sigma(P)$
  - tarpeen, koska $\circ \;||\; \circ \xrightarrow{a} \circ \;=\; \circ \xrightarrow{a} \circ$ jos $\Sigma(\circ) = \emptyset$,
    mutta $\circ \;||\; \circ \xrightarrow{a} \circ \;=\; \circ$ jos $\Sigma(\circ) = \{a\}$
- prossin **jäljet** (**traces**) $Tr(P)$
  - alkutilasta alkavien polkujen tuottamat näkyvien nimien jonot
  - $Tr(P) \;=\; \{a_1 a_2 \cdots a_n \in \Sigma^* \mid \hat{s} {=} a_1 a_2 \cdots a_n {\Rightarrow}\}$
  - kuten äärellisen automaatin kieli, jos jokainen tila on lopputila
- $P =_{Tr} Q$ jos ja vain jos $\Sigma(P) = \Sigma(Q) \wedge Tr(P) = Tr(Q)$

Jälkisemantiikka hukkaa tiedon lukkiumista

Arja || K3 $=_{Tr}$



⇒ lisäämme myöhemmin siihen informaatiota

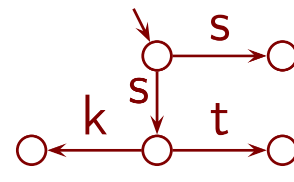
- käsitelkäämme kuitenkin yksi asia kerrallaan

Turvallisuus- ja elävyysominaisuudet

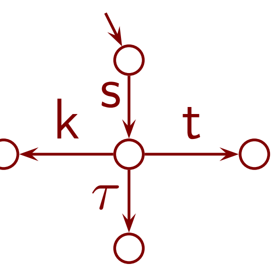
- **turvallisuusominaisuus** (**safety property**) on ominaisuus, jonka rikkoutuminen voidaan havaita suorituksen äärellisestä alkuosasta
  - jonain hetkenä jotain meni pieleen
  - esim. pankkiautomaatti antoi väärän määrän rahaa
  - vrt. (peräkkäis)ohjelmien osittainen oikeellisuus
- **elävyysominaisuuden** (**liveness property**) rikkoutumisen havaitseminen vaatii koko (mahdollisesti äärettömän) suorituksen
  - jotain ei tapahdu vaikka pitäisi
  - esim. pankkiautomaatti ei anna korttia takaisin
- elävyysominaisuuksien tarkka rajaus ja nimistö on horjuvaa
  - käytetään myös termiä **etenevyys** (**progress**)
- **reiluus** (**fairness**) sanoo, että vaihtoehtoa ei syrjitä loputtomasti
  - esim. proffa ei lykkää tentin tarkastusta loputtomasti muiden töiden vuoksi

Abstraktit ominaisuudet

- τ-tapahtumien määrä ennen seuraavaa näkyvää tapahtumaa ei ole tärkeä
  - τ-tapahtumilla voi olla näkyviä välillisiä seurauksia



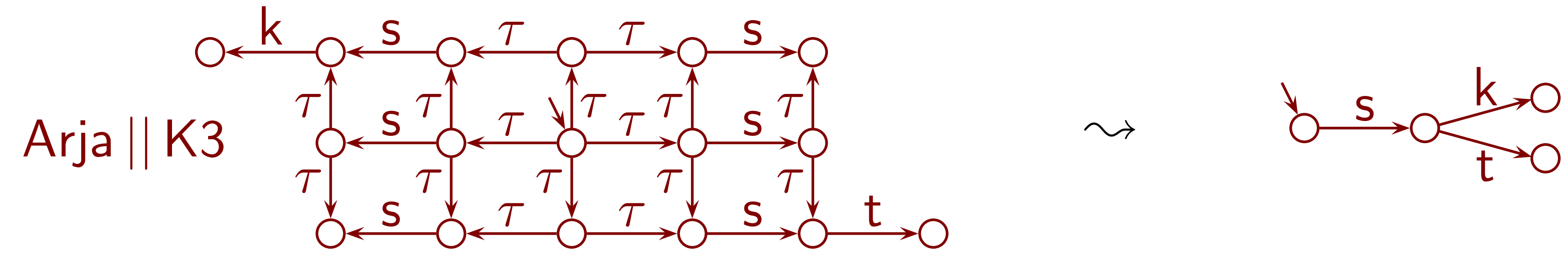
- **änkytykselle epäherkkä** (**stuttering-insensitive**) ominaisuus



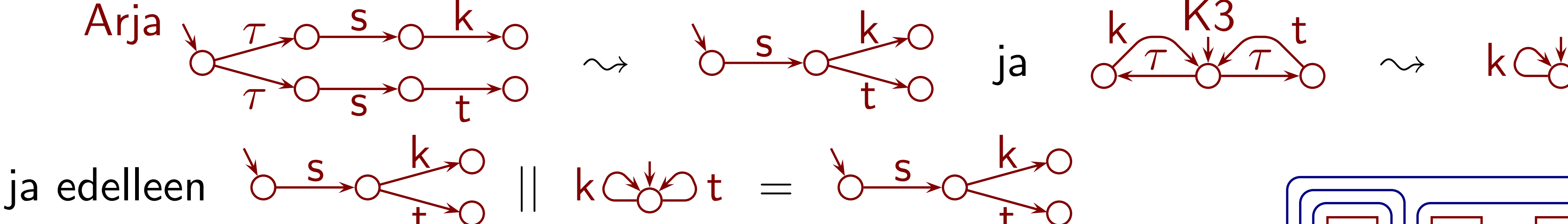
Jälkisemantiikka säilyttää abstraktit turvallisuusominaisuudet eikä muuta

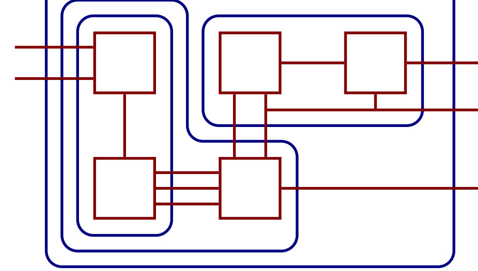
Paloittainen kutistetun LTS:n muodostaminen

- äärellisistä automaateista tutuilla ja muilla algoritmeilla äärellisestä LTS:stä voidaan muodostaa jälkisamanlainen **kutistettu** (**reduced**) LTS
- se saattaa olla riittävän pieni katsottavaksi



- myös osia tai osakokonaisuuksia voi kutistaa ennen yhteen liittämistä



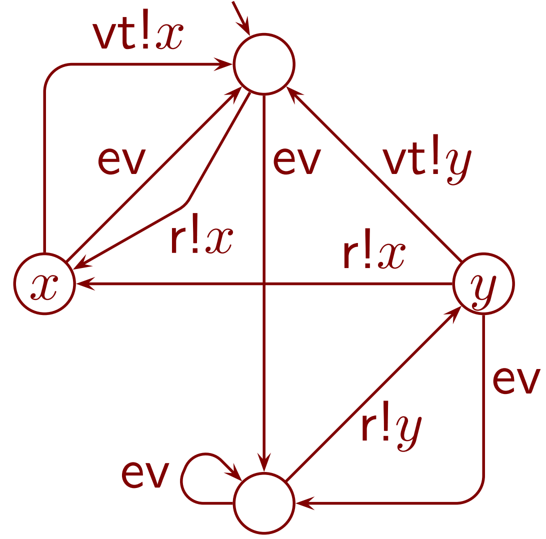


⇒ järjestelmälle saadaan kutistettu LTS ilman että koskaan muodostetaan sen kutistamatonta LTS:ää

- lievittää tilaräjähdysongelmaa

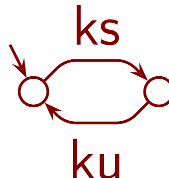
Pankin ja pankkiautomaatin käyttäytyminen jälkisemantiikalla, rahan näkökulma

- järjestelmä toimii epäluotettavilla tietoliikenneyhteyksillä
  ⇒ virheitä voi tapahtua
  - tavoite on kuitenkin, että niistä jää pankkiin vinkki, ja että ne eivät ole asiakkaan tappioksi
- muu on kätketty, ja näkyväksi on jätetty
  - tilin veloitus vt!x tai vt!y
  - rahan antaminen r!x tai r!y
  - pankkiin jäävä tieto, että on epävarmaa, antoiko automaatti rahaa ev
- jos tiliä veloitetaan (vt!x tai vt!y), niin sama summa on annettu (r!x tai r!y)
- rahan antamiseen liittyy aina tilin veloitus tai tieto ”epävarma” (ev)
  - ev voi tapahtua ennen kuin vastaava r!y
- ev voi tapahtua myös ilman rahan antamista
  - esim. ”kyllä” katosi matkalla



Valitsemalla näkyvät tapahtumat toisin nähdään muun muassa, että kortti sisään ja kortti ulos vuorottelevat

- siitä ei voi päätellä, että kortti ei voi jumiutua automaattiin, koska jälkisemantiikka ei kerro esim. voiko järjestelmä lukkiutua kortti sisällä



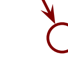
Esijärjestys (preorder)

- ”toteuttaa vaatimukset” ei ole samanveroisuus- vaan osittaisjärjestysrelaatio
  - järjestelmä saa olla parempi mutta ei huonompi kuin vaatimukset
  - osittaisjärjestys, koska järjestelmä voi olla jossain suhteessa parempi ja toisessa huonompi kuin vaatimukset

$\Rightarrow$ on mielekästä verrata prosesseja relaatiolla

$$P \leq_{Tr} Q \;:\Leftrightarrow\; \Sigma(P) = \Sigma(Q) \wedge Tr(P) \subseteq Tr(Q)$$

- aakkostot verrataan yhtäsuuruudella, koska
  - muuten prosessit ovat eri tehtäviin, verrataan omenoita talvitakkeihin
    {kortti-sisään, kortti-ulos} vastaan {veloita-tiliä, palkkatulo}
  - voidaan osoittaa, että muuten joudutaan matemaattisiin vaikeuksiin
- jokaiselle aakkostolle on olemassa LTS, joka jälkisemantiikan näkökulmasta toteuttaa jokaisen vaatimuksen!



  - kun ei tee mitään, niin ei tee virheitä
- tämäkin kertoo, että jälkisemantiikka on puutteellinen
  - ei pysty vaatimaan, että jotain täytyy tehdä

$\Rightarrow$ käymme lisäämään semantiikkaan ominaisuuksia

14.4 Lukkiumat ja vakaat estymät

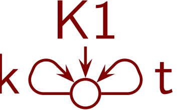
Lukkiuma (**deadlock**) on dramaattinen rinnakkaisjärjestelmän vika

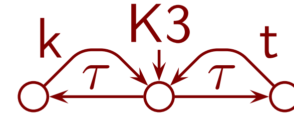
- jokainen prosessi odottaa, että jokin muu tekisi ensin jotain

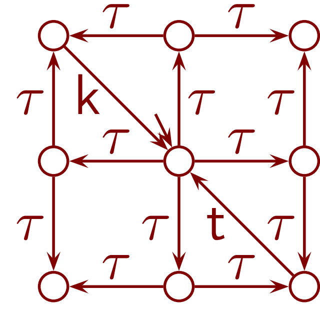
$\Rightarrow$ mitään ei tapahdu

Järjestelmän lukkiumat eivät määräydy osien lukkiumaan vievistä jäljistä ja jälkisemantiikasta

- vaikka K1 $=_{Tr}$ K3 ja kumpikaan ei lukkiudu, niin
  - K1 $||$ K1 $=$ K1 ei lukkiudu
  - K1 $||$ K3 $=$ K3 ei lukkiudu
  - K3 $||$ K3 lukkiutuu







$\Rightarrow$ jälkisemantiikka täydennettynä lukkiumajäljillä ei tuota **kongruenssia**

Samuuskäsite on kongruenssi, jos ja vain jos kokonaisuus säilyy aina samanveroisena, kun sen osa korvataan samanveroisella osalla

- jos $L \cong L'$ niin $L \setminus A \cong L' \setminus A$ ja $L[\cdots] \cong L'[\cdots]$
- jos $L_1 \cong L'_1$ ja $L_2 \cong L'_2$ niin $L_1 \,||\, L_2 \cong L'_1 \,||\, L'_2$

$\Rightarrow$ jos $L_1 \cong L'_1, \ldots, L_n \cong L'_n$ ja f on rakennettu käyttäen vain $||$, $\setminus$ ja $[\cdots]$, niin $f(L_1, \ldots, L_n) \cong f(L'_1, \ldots, L'_n)$

Jos samuuskäsite ei ole kongruenssi, niin pistämällä L sopivaan ympäristöön saadaan esiin tietoa, joka ei näy L:n semantiikasta suoraan

- toisenlainen esimerkki piilevästä tiedosta
  - kuukausien pituudet päivinä ovat

    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30 ja 31
  - kellon näyttö on rikki: kellonaika, päivän nro ja vuosi näkyvät, kuukausi ei
  - tänään on 1.3.
  - milloin viimeistään tiedetään varmasti, onko kellon kuukausi oikein?

Voidaan todistaa, että jotta kongruenssi säilyttäisi lukkiumat,
sen täytyy säilyttää **vakaat estymät** (**stable failures**)

- LTS:n L tila s on **vakaa** (**stable**) jos ja vain jos siitä ei lähde τ-siirtymiä
- s **kieltäytyy** (**refuses**) tapahtumasta a jos ja vain jos s:stä ei lähde a-siirtymiä
- L:n vakaa estymä on pari (σ, A) siten, että
  - σ on L:n jälki
  - A on joukko L:n näkyviä tapahtumia
  - σ vie L:n alkutilasta vakaaseen tilaan, joka kieltäytyy kaikista A:n alkioista
- kytkemällä L rinnan muihin LTS:iin saadaan tietoa L:n vakaista estymistä aivan kuten ajan kuluminen antaa tietoa rikkinäisen kellon kuukaudesta

Toisaalta vakaiden estymien säilyttäminen riittää

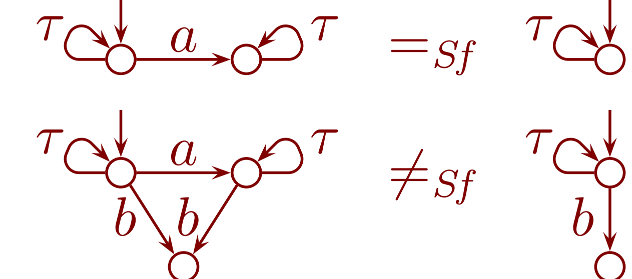
$$P =_{Sf} Q \;:\Leftrightarrow\; \Sigma(P) = \Sigma(Q) \wedge Sf(P) = Sf(Q)$$

- jälki σ on lukkiumajälki, jos ja vain jos (σ, Σ) on vakaa estymä
  $\Rightarrow$ lukkiumajäljet saadaan vakaista estymistä
- $=_{Sf}$ on kongruenssi $||$, $\backslash$ ja $[\cdots]$ suhteen

Kongruenssiominaisuus riippuu operaattoreista

- joskus käytetään keskeytysoperaattoria, joka lisää joka tilaan saman jatkon
- keskeytys prosessilla $\xrightarrow{b}$ tuottaa yläkuvista alakuvat
  - erona vakaa estymä $(ab, \emptyset)$

$\Rightarrow$ $=_{Sf}$ ei ole kongruenssi keskeytyksen suhteen



$=_{Tr,Sf}$ on kongruenssi myös keskeytyksen suhteen

$$P =_{Tr,Sf} Q \;:\Leftrightarrow\; \Sigma(P) = \Sigma(Q) \wedge Tr(P) = Tr(Q) \wedge Sf(P) = Sf(Q)$$

14.5 CFFD-semantiikka

Pillastuma (**livelock**, **divergence**)

- päättymätön ketju τ-siirtymiä tarkoittaa, että prosessi touhuaa loputtomasti tekemättä mitään näkyvää

  - jos ympäristö tarjoaa a:n, vasen lopulta tekee sen, oikea ei välttämättä

$\Rightarrow$ lisäämme semantiikkaan pillastumajäljet
  - $Div(P) = \{a_1 \cdots a_n \in \Sigma^* \mid \exists s : \hat{s} {=}a_1 \cdots a_n{\Rightarrow}\, s \,{-}\tau^\omega{\rightarrow}\}$
  - $s \,{-}\tau^\omega{\rightarrow}$ tarkoittaa, että tilasta s alkaa päättymätön ketju τ-siirtymiä

- jotta saisimme kongruenssin \ suhteen, lisäämme myös äärettömät jäljet
  - $Inf(P) = \{a_1 a_2 \cdots \in \Sigma^\omega \mid \hat{s} {=}a_1 a_2 \cdots{\Rightarrow}\}$
  - äärettömän jäljen suorittaminen ei lopu koskaan

  $\Rightarrow$ ei ole mielekästä puhua tilasta äärettömän jäljen jälkeen
- $=_{Tr,Div,Inf}$ on kongruenssi

Kaikki tähänastinen yhdessä muodostaa **CFFD-semantiikan**

- sisältää tiedot jäljistä, lukkiutumista ja pillastumista

  $\Rightarrow$ hyvä rinnakkaisjärjestelmien toiminnan tarkastamisessa
- $Tr(P) = Div(P) \cup \{\sigma \mid (\sigma, \emptyset) \in Sf(P)\}$

$\Rightarrow$ jälkiä ei tarvitse huomioida erikseen

$P =_{\mathsf{CFFD}} Q \;:\Leftrightarrow$

$$\Sigma(P) = \Sigma(Q) \wedge Sf(P) = Sf(Q) \wedge Div(P) = Div(Q) \wedge Inf(P) = Inf(Q)$$

Miksi nimi ”CFFD”?

- Oxfordin yliopistossa kehitetty failures–divergences-semantiikka ei säilytä mitään tietoa sen jälkeen, kun prosessi on suorittanut pillastumajäljen
  - sille τ ↻ ○ $\xrightarrow{a}$ ○ on sama kuin τ ↻ ○
  - sen rakentamisessa käytettiin toisenlaista lähestymistapaa, joka ei toimi pillastumajälkien taakse
  - (lähestymistapa takasi semantiikalle kauniita matemaattisia ominaisuuksia)
- ilmiöstä käytettiin nimeä ”chaotic divergence”
- CFFD kehitettiin samankaltaiseksi semantiikaksi ilman kaaosta
  - rakentamiseen käytettiin tällä luennolla käsiteltyjä keinoja
  - lisäksi tarvittiin oivallus, että estymän tilan s pitää olla vakaa

$\Rightarrow$ **chaos-free failures divergences**

CFFD-esijärjestys

$$\Sigma(P) = \Sigma(Q) \wedge Sf(P) \subseteq Sf(Q) \wedge Div(P) \subseteq Div(Q) \wedge Inf(P) \subseteq Inf(Q)$$

- koska ε on jokaisen prosessin jälki, on $Sf(P) \neq \emptyset$ tai $Div(P) \neq \emptyset$
- $Sf\left(\tau \circlearrowright \circ\right) = \emptyset$ ja $Div\left(\circ\right) = \emptyset$

$\Rightarrow$ CFFD-semantiikassa **ei ole** prosessia, joka toteuttaa jokaisen vaatimuksen

- jos $P \leq_{\text{CFFD}} Q$ ei päde vaikka $\Sigma(P) = \Sigma(Q)$, niin jokin seuraavista pätee:
  - P:llä on jälki, jota Q:lla ei ole
  - P pääsee jollain jäljellä vakaaseen tilaan, joka kieltäytyy enemmän kuin mikään Q:n vakaa tila saman jäljen jälkeen
  - P:llä on pillastumajälki, jota Q:lla ei ole
  - (P:llä on ääretön jälki, jota Q:lla ei ole)

$\Rightarrow$ saadaan selkeä vastaesimerkki

CFFD-esijärjestys on **esikongruenssi** (**precongruence**)

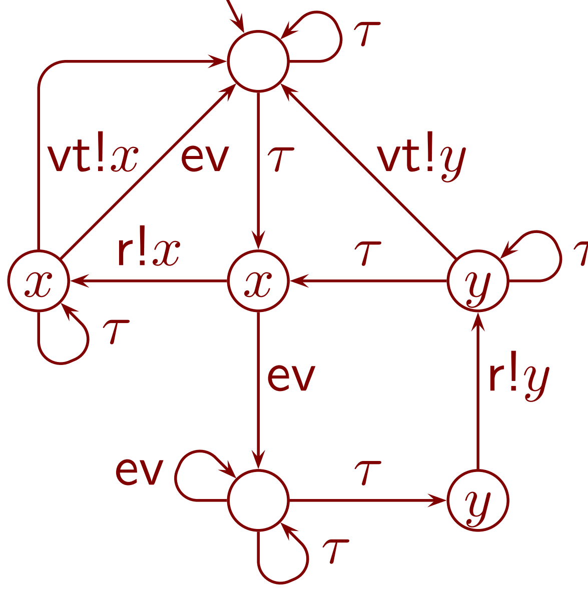
- jos $L_1 \leq_{\text{CFFD}} L'_1, \ldots, L_n \leq_{\text{CFFD}} L'_n$, niin

$$f(L_1, \ldots, L_n) \leq_{\text{CFFD}} f(L'_1, \ldots, L'_n)$$

$\Rightarrow$ jos järjestelmän osa korvataan paremmalla, niin järjestelmä säilyy samanveroisena tai muuttuu paremmaksi
  - osan paremmat ominaisuudet eivät välttämättä pääse järjestelmässä esille

- usein joudutaan olettamaan järjestelmän käyttäjistä jotain
  - esim. kirjasto ei voi taata, että jokainen saa lopulta lainata kirjan, jos joku lainaaja ei koskaan palauta sitä
- esikongruenssi takaa, että jos järjestelmä on todistettu toimivaksi tietyillä käyttäjillä, se toimii myös paremmin käyttäytyvillä käyttäjillä

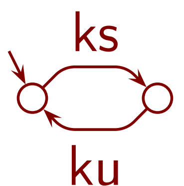
Pankin ja -automaatin käyttäytyminen CFFD-semantiikalla, rahan näkökulma



- τ-silmukoissa esim. nostopyynnöt katoavat ja automaatti vastaa "yhteyshäiriö" toistuvasti
- keskimmäisessä tilassa
  - automaatti on valmis antamaan rahaa
  - pankki odottaa tietoa, että se antoi rahaa, mutta määräaika voi mennä umpeen
  - muuta ei voi tapahtua
- oikeassa alanurkassa pankki on jo luopunut odottamasta, eli ev tapahtui jo
- voi tuntua hämmentävältä, että ev vie keskimmäisestä tilasta pillastuvaan
  - jos automaatti on valmis suorittamaan r!x, se ei pääse eteenpäin muuten
  - pankki ei voi pyöriä loputtomiin ilman viestejä automaatilta
- tämä on kuitenkin vain kuvan piirtämistapaan liittyvä illuusio
  - todellisuudessa järjestelmä pillastui jo ennen ev:ta
  - CFFD-semantiikka ei säilytä tietoa, että voidaan poistua pillastumasta

  $\Rightarrow$ kuva on CFFD-samanlainen "oikein" piirretyn kanssa
- tarpeettoman tiedon hukkaaminen (ja siis "väärin" piirtäminen) on välttämätöntä, että saataisiin tarpeeksi pieniä kuvia
  - täytyy opetella olemaan lukematta kuvasta semantiikan ulkopuolisia asioita

Pankkikortin näkökulma



- nyt näkyy, että kortti ei voi jäädä jumiin automaattiin

Toiminnallinen determinismi

- äärellisten automaattien determinismin käsite ei sovellu LTS:iin
  - liian pikkutarkka, ei esim. säily rinnankytkennässä

$\Rightarrow$ uusi käsite: **toiminnallinen determinismi** (**operational determinism**)

- L on toiminnallisesti deterministinen, jos ja vain jos jokaiselle σ, s_1 ja s_2 siten, että $\hat{s} =\sigma\Rightarrow s_1$ ja $\hat{s} =\sigma\Rightarrow s_2$, pätee
  - jokaiselle $a \in \Sigma$: jos $s_1 =a\Rightarrow$, niin $s_2 =a\Rightarrow$
  - jos $s_1 -\tau^\omega\rightarrow$, niin $s_2 -\tau^\omega\rightarrow$
- voidaan todistaa, että CFFD-minimaaliset LTS:t ovat toiminnallisesti deterministisiä (mutta ei aina toisinpäin)

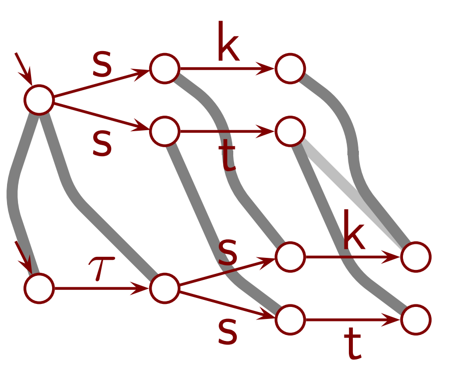
$\Rightarrow$ voidaan sanoa, että toteutus on yleensä deterministisempi kuin vaatimukset
  - toteutus saa kiinnittää asioita, jotka on vaatimuksissa jätetty auki
  - näin on usein muuallakin:
    huippunopeus vähintään 120 km/h $\rightsquigarrow$ huippunopeus 135 km/h

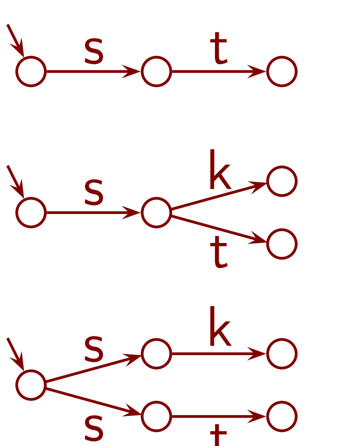
14.6 Muita semantiikkoja

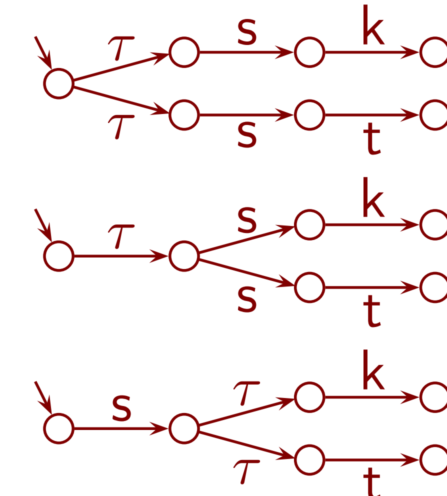
Kongruensseja $||$, $\setminus$ ja $[\cdots]$ suhteen

Havaintosamuus (**observation equivalence**)

- tunnetaan myös nimellä **heikko bisimilaarisuus** (**weak bisimilarity**)
- samanveroisten järjestelmien on kyettävä matkimaan toisiaan
  - $\hat{s}_1 \sim \hat{s}_2$
  - jos $s_1 \sim s_2$ ja $s_1 =a\Rightarrow s'_1$, niin on s'_2 siten, että $s'_1 \sim s'_2$ ja $s_2 =a\Rightarrow s'_2$
  - jos $s_1 \sim s_2$ ja $s_2 =a\Rightarrow s'_2$, niin on s'_1 siten, että $s'_1 \sim s'_2$ ja $s_1 =a\Rightarrow s'_1$
  - edellä $a \in \Sigma$ tai $a = \varepsilon$ (eikä τ!)







- matkimisen täytyy onnistua vaikka vuorotellen
- ei luontevaa esijärjestystä
- jos järjestelmä rikkoo vaatimuksen, niin ei ehkä ole selkeää vastaesimerkkiä

Haarautuva bisimilaarisuus (branching bisimilarity)

- edellisen muunnelma, jossa simulaation tulee toimia myös kesken $=a\Rightarrow$-merkintään sisältyvän τ-ketjun
- tämän eräs pillastumat säilyttävä muunnelma
  - vastaa hyvin suosittua aikalogiikkaa nimeltä CTL
  - on vahvin laajasti käytetty abstrakti semantiikka

"Fair testing" kongruenssi

- heikoin kongruenssi, joka erottaa toisistaan τ a ja τ τ a
  - joskus oletetaan, että τ-silmukasta tullaan lopulta pois jos mahdollista

  $\Rightarrow$ tärkeää nähdä, pääseekö τ-silmukasta pois
- määritelmä on monimutkainen

Kahden kongruenssin yhdistelmä on kongruenssi

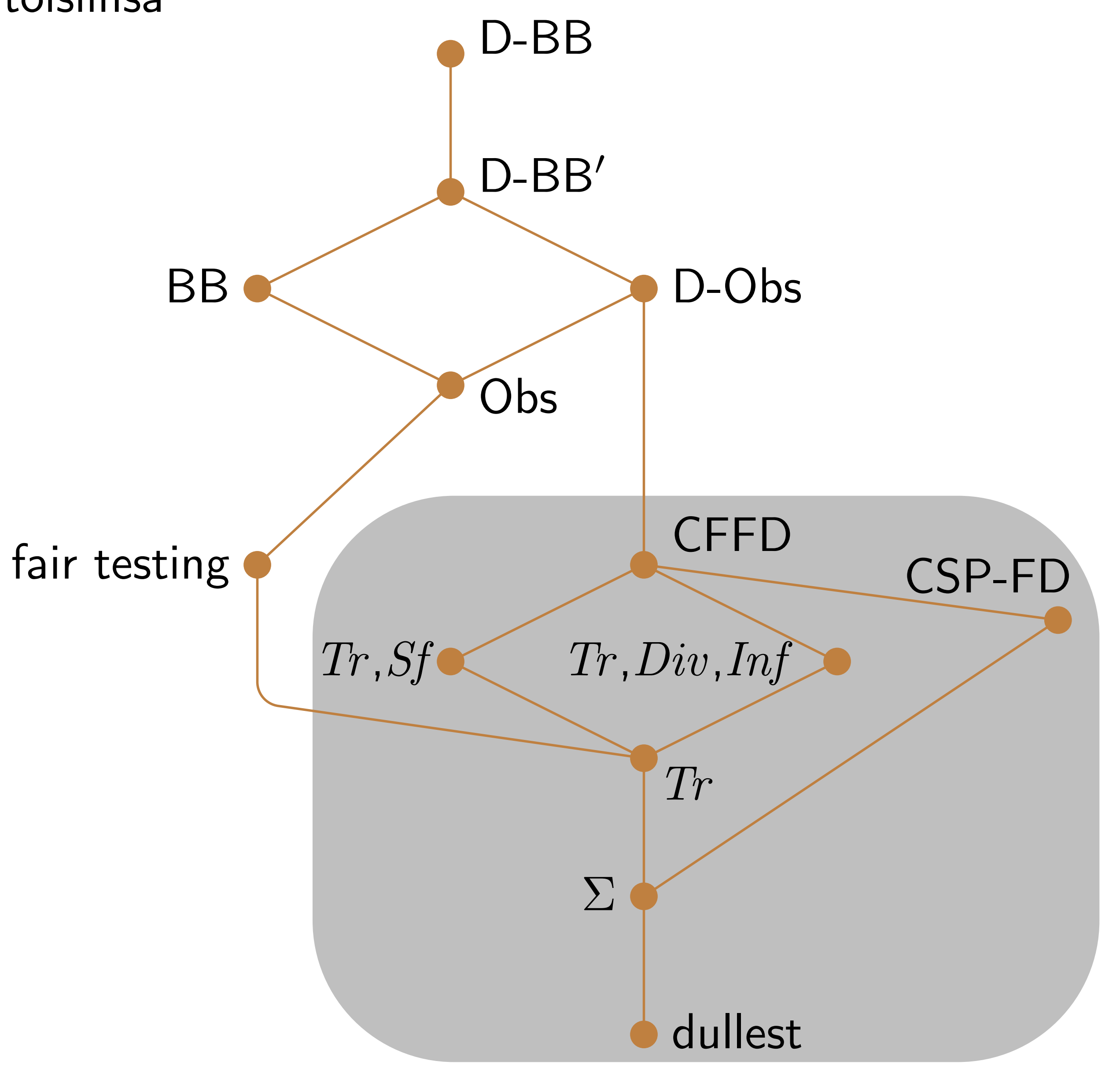
$$P =_{1,2} Q \;:\Leftrightarrow\; P =_1 Q \wedge P =_2 Q$$

$\Rightarrow$ kirjallisuudesta löytyy **paljon** kongruensseja

- lisäksi on muunnelmia, joilla varmistetaan kongruenssiominaisuus jonkin muun operaattorin suhteen

Tähänastiset kongruenssit suhteessa toisiinsa

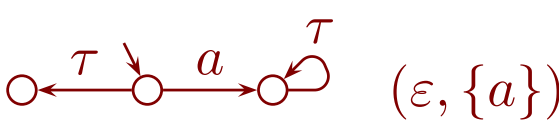
- lisäksi on olemassa äärettömän monta muuta kongruenssia



CFFD:n alainen maailma

- $a.P$ on prosessi, joka tekee ensin a-siirtymän ja sitten jatkaa kuten P
  - $a.L$ saadaan lisäämällä L:n eteen uusi alkutila ja siitä a-siirtymä vanhaan
- tämä kuva näyttää todistetusti *kaikki* kongruenssit, jotka
  - ovat sitä $||$, $\backslash$, $[\cdots]$ ja . suhteen
  - ovat $=_{\mathsf{CFFD}}$:n implikoimia, ts. $L_1 =_{\mathsf{CFFD}} L_2 \Rightarrow L_1 \cong L_2$

CFFD
NDFD
nF
snF
$aenI$
Sf
Div $eanI$
CSP, anF
Inf
$sanF$
$minD$ anT anI
Tr
Σ
dullest



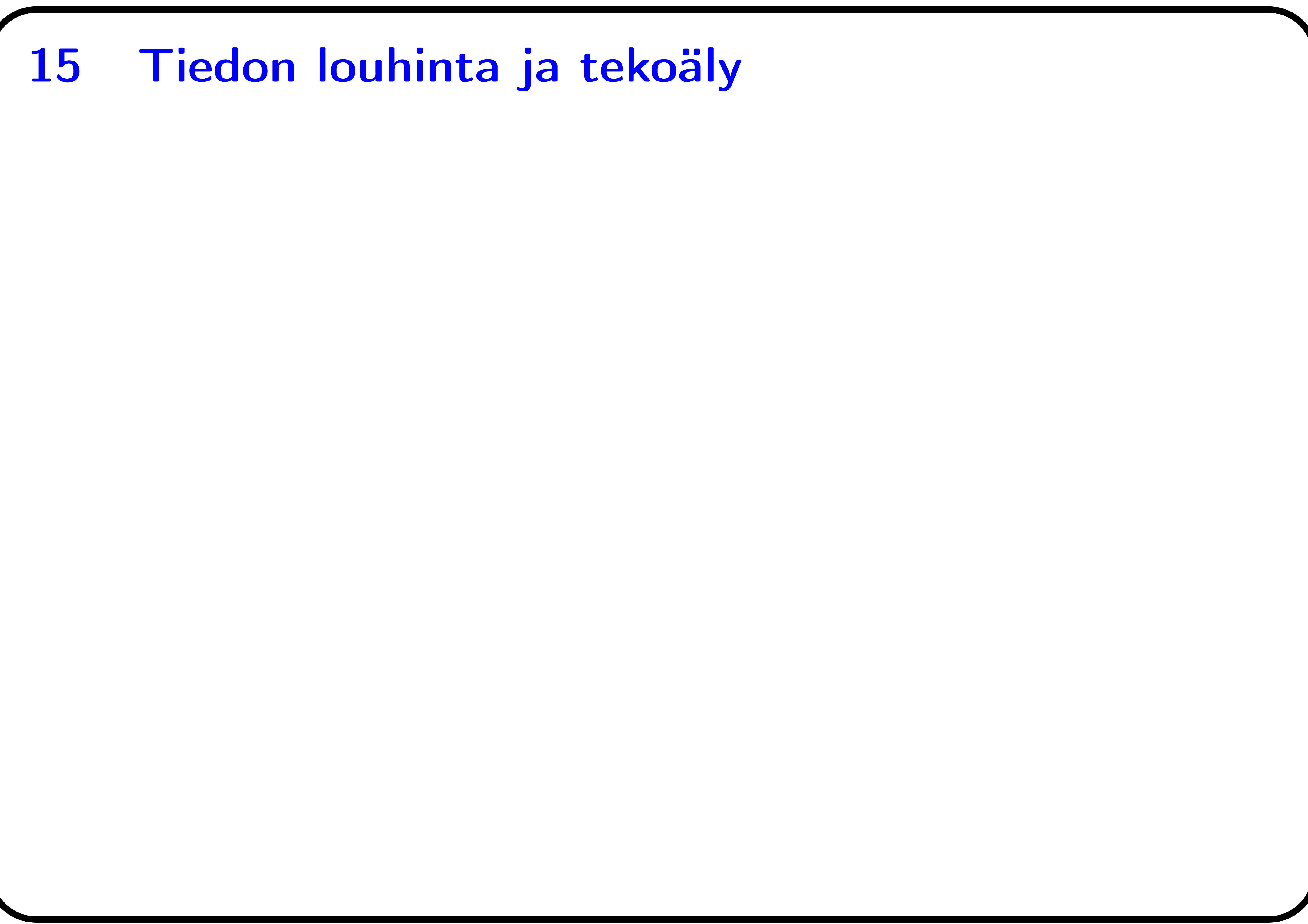
$(\varepsilon, \{a\})$

Miksi niin monta semantiikkaa?

- osittain kyse on sen päättämisestä, mikä katsotaan havaittavaksi
  - pillastumat: ei mitään, ne joista ei voi poistua, . . . , kaikki, minimaaliset
  - lukkiumat: ei, on
  - lineaarinen vai haarautuva aika
  - . . .
- estymäkäsitteiden tarve tulee valitusta vuorovaikutustavasta
  - synkronisessa vuorovaikutuksessa osapuoli voi estää tapahtuman
  - teorian laajan mallinnusvoiman hinta
- kyse on myös siitä, että epädeterminismi on monimutkainen ilmiö
  - jos kaikki LTS:t olisivat toiminnallisesti deterministisiä, kongruenssien määrä pienenisi valtavasti

Mikä semantiikka valita?

- sellainen, joka
  - säilyttää riittävästi informaatiota
  - säilyttää mahdollisimman vähän tarpeetonta informaatiota
- mitä vähemmän informaatiota säilytetään, sitä paremmin LTS:iä voi kutistaa
  - seuraavalla ohjelmalla vähemmän työtä — tämä on usein tärkeää
  - pienemmät kuvat


15 Tiedon louhinta ja tekoäly

16 Lopuksi

Merkintöjen hakemisto

$\lfloor x \rfloor$	suurin kokonaisluku i siten, että $i \leq x$
$\lceil x \rceil$	pienin kokonaisluku i siten, että $x \leq i$
$os\uparrow$	osoittimen os päässä oleva tietue; ks. sivu 43
$\perp$	(esim. osoittimen) arvo, joka tarkoittaa, että ko. kohdetta ei ole
$o(\ldots)$	kasvaa hitaammin kuin $\ldots$; ks. luku 4.2
$O(\ldots)$	kasvaa enintään yhtä nopeasti kuin $\ldots$; ks. luku 4.2
$\Theta(\ldots)$	kasvaa yhtä nopeasti kuin $\ldots$; ks. luku 4.2
$\Omega(\ldots)$	kasvaa vähintään yhtä nopeasti kuin $\ldots$; ks. luku 4.2
$\omega(\ldots)$	kasvaa nopeammin kuin $\ldots$; ks. luku 4.2
ε	tyhjä jono eli jono, jossa ei ole yhtään alkiota
$n \operatorname{div} m$	kokonaislukuosamäärä, kun n jaetaan m:llä; ks. sivu 9
$\log_2 x$	luvun x 2-kantainen logaritmi; kun se on olemassa, pätee $2^{\log_2 x} = x$
$\max A$	joukon A suurin alkio
$\min A$	joukon A pienin alkio
$n \bmod m$	jakojäännös, kun n jaetaan m:llä; ks. sivu 9
$\mathbb{N}$	luonnolliset luvut eli $\{0, 1, 2, 3, \ldots\}$

$\mathbb{Z}$ kokonaisluvut eli $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

$\mathbb{Z}^+$ positiiviset kokonaisluvut eli $\{1, 2, 3, \ldots\}$

$\mathbb{Z}^-$ negatiiviset kokonaisluvut eli $\{\ldots, -3, -2, -1\}$

$\mathbb{R}$ reaaliluvut

$\mathbb{R}^+$ positiiviset reaaliluvut

$\mathbb{R}^-$ negatiiviset reaaliluvut