58131 Tietorakenteet

luennot keväällä 2008, periodit III–IV

Jyrki Kivinen

- tietojenkäsittelytieteen aineopintokurssi, 8 op, pääaineopiskelijoille pakollinen

- esitietoina *Johdatus diskreettiin matematiikkaan* ja *Java-ohjelmointi*

- mahdollinen jatkokurssi *Algoritmien suunnittelu* (valinnainen aineopintokurssi, 4 op, periodi III)

- tällä kurssilla opittavat tekniikat ja ajattelutavat ovat tarpeen kurssilla *Laskennan mallit* (pääaineopiskelijoille pakollinen aineopintokurssi)

Esitietovaatimuksisista

Ohjelmointitaito Kurssilla ei (juurikaan) käytetä Javaa tai muuta varsinaista ohjelmointikieltä. Pääpaino on yleisillä periaatteilla, jotka esitetään käyttämällä luonnollista kieltä, kuvia, pseudokoodia jne. Kuitenkin

- pseudokoodiesitysten pyörittely vaatii sujuvaa ohjelmoinnin perusteiden hallintaa ja
- motivaation kannalta on oleellista, että yleiset periaatteet voidaan realisoida käytännön sovelluksissa.

Java-toteutusta harjoitellaan erikseen harjoitustyössä.

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

Annettava opetus, kurssin suorittaminen

- luentoja $2 + 2$ tuntia viikossa
- harjoituksia 2 tuntia viikossa alkaen 14.1.; tarkemmin seuraavilla kalvoilla
- kaksi kurssikoetta (kummankin periodin tenttiviikolla)
- (ei luentoja eikä harjoituksia tenttiviikoilla eikä väliviikolla)
- kokeissa osattava luennoilla *ja* harjoituksissa käsitellyt asiat
- **Huom.** kurssin arvioitu työmäärä on 213 tuntia (eli 14 viikon aikana keskimäärin yli 15 tuntia viikossa)
- maksimipisteet harjoituksista 12, tenteistä $24 + 24$, yhteensä 60; hyväksymisraja n. 30, arvosanan 5/5 raja n. 50

Oppimateriaali

- Kokeissa edellytetää luennoilla ja harjoituksissa esitetyt asiat. Luentomateriaali (eli nämä kalvot) ilmestyvät kurssin kotisivulle kurssin edetessä.

- Luennot perustuvat Matti Nykäsen ja Matti Luukkaisen aiempaan luentomonisteeseen sekä tämän pohjalla olevaan vanhempaan luentomateriaaliin.

- Oheismateriaaliksi suositellaan kirjaa

  > Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms. MIT Press 2001.

  Kurssin monet (mutta ei kaikki) keskeiset asiat perustuvat tähän kirjaan.

- Kalvokokoelmaa ei ole tarkoitettu itseopiskeluun; se saattaa tarvita tuekseen luentojen tai kirjan seuraamista.

Laskuharjoitukset

Osa tehtävistä (noin joka toisella viikolla noin puolet) tehdään ryhmissä. Ryhmät muodostetaan toisella laskuharjoituskerralla (21.–25.1.), jolloin pitää osallistua "omaan" ryhmään.

Suurin osa tehtävistä tehdään itsenäisesti (laitoksen "normaali" käytäntö), mutta niistäkin voi mielellään keskustella ryhmissä.

Osa tehtävistä tehdään verkossa TRAKLA2-tietorakennesimulaattorilla. Tästä annetaan myöhemmin lisäohjeita.

Kurssin suorittaminen edellyttää, että kaikista harjoituksista yhteensä (ryhmä + henk.koht + TRAKLA) **ainakin 25%** on tehty.

Tekemällä 25% tehtävistä saa 0/12 pistettä.
Tekemällä 85% tehtävistä saa 12/12 pistettä.

Sisältö

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

Kurssilla etsitään vastauksia seuraavanlaisiin kysymyksiin:

- miten laskennassa tarvittavat tiedot organisoidaan tehokkaasti
- miten varmistumme toteutuksen toimivuudesta
- miten analysoimme toteutuksen tehokkuutta ja
- millaisia tunnettuja tietorakenteita voimme hyödyntää.

1. Johdanto

Tarkastelemme esimerkkien valossa joitain peruskäsitteitä:

- tietorakenne
- abstrakti tietotyyppi
- algoritmi.

Tutustumme silmukkainvariantteihin, jotka ovat keskeinen keino analysoida algoritmien oikeellisuutta.

Tietorakenteiden ja algoritmien tehokkuuden tarkastelua varten tarkastellaan funktioiden kertaluokkia.

Tietorakenteet: mitä ja miksi

Tietorakenne (data structure) on tapa organisoida dataa (tyypillisesti tietokoneen muistiin) siten, että se on tarvittavalla tavalla käytettävissä ja muokattavissa. Tietorakenteeseen kuuluu

- talletusrakenne: miten data esitetään koneen muistissa (tms.)
- operaatiot: miten tietorakennetta muokataan ja siellä oleva data saadaan käyttöön.

Ohjelmointikielistä tuttu perusesimerkki on taulukko: esim. 5×10-ulotteinen kokonaislukutaulukko voitaisiin esittää Javassa muuttujana

```
int[][] a = new int[5][10].
```

Talletusrakenne voisi olla, että alkio $a(i, j)$ talletetaan muistipaikkaan $B + 10i + j$ jollain B. (Tässä $0 \leq i \leq 4$ ja $0 \leq j \leq 9$.) Taulukkoa käsitellään indeksoimalla:

```
a[3][8]= 15; x= a[2,7];
```

Tietorakenteeseen liittyy läheisesti käsite abstrakti tietotyyppi. Abstrakti tietotyyppi määrittelee tietorakenteen sallitut operaatiot ja niiden tulokset, mutta ei ota kantaa toteutukseen.

Esimerkki 1.1: Tarkastellaan puhelinluetteloa, jossa on pareja (nimi, puhelinnumero). Abstrakti tietotyyppi puhelinluettelo voisi sallia seuraavat operaatiot:

- init: luo tyhjä puhelinluettelo
- add(n, p): lisää pari (n, p)
- del(n): poista henkilö n luettelosta
- find(n): palauta henkilön n puhelinnumero
- list: listaa kaikki nimet ja numerot aakkosjärjestyksessä.

Oletetaan, että sekä nimet että numerot ovat pituudeltaan sopivasti rajoitettuja merkkijonoja.

Mahdollisia talletustapoja:

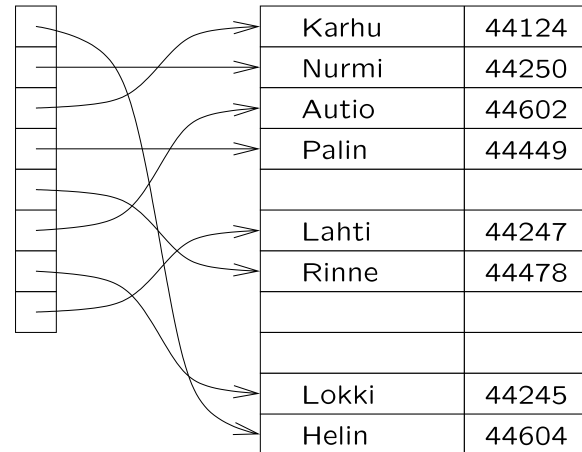
- talletetaan taulukkoon (nimi, numero)-pareja mielivaltaisessa järjestyksessä
- talletetaan taulukkoon (nimi, numero)-pareja nimen mukaan aakkosjärjestyksessä
- talletetaan taulukkoon osoittimia erilliselle talletusalueelle, jossa (nimi, numero)-parit säilytetään
- binäärihakupuu (esitellään myöhemmin kurssilla)
- B-puu (esitellään myöhemmin kurssilla)
- hajautustaulukko (esitellään myöhemmin kurssilla)

"Paras" talletustapa riippuu tilanteesta:

- kuinka paljon dataa on
- kuinka usein mitäkin operaatiota suoritetaan
- mitkä ovat eri operaatioiden nopeusvaatimukset
- onko muistitilasta pulaa jne.

Helin	44604
Nurmi	44250
Karhu	44124
Palin	44449
Rinne	44478
Autio	44602
Lokki	44245
Lahti	44247

Talletus taulukkoon
(järjestys mielivaltainen)



Erillinen talletusalue

Esimerkki 1.2: Lyhimmän reitin etsiminen

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

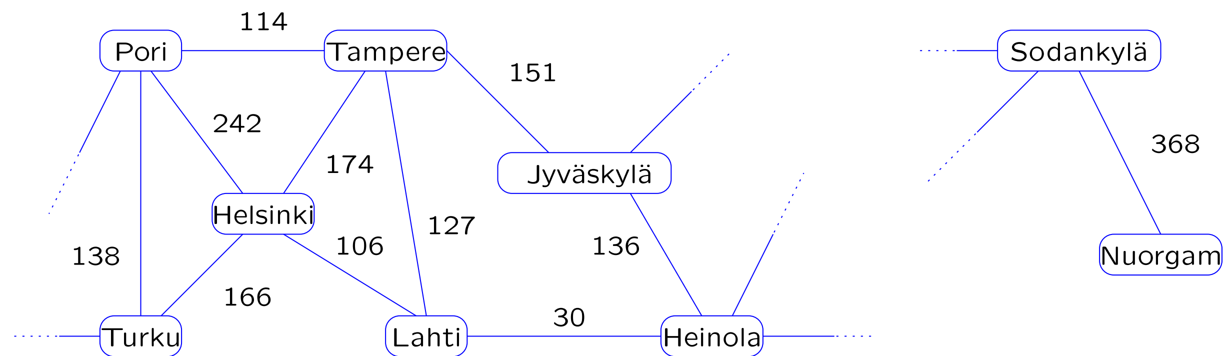
Helsinki–Lahti	106 km
Helsinki–Turku	166 km
Turku–Pori	138 km
Lahti–Tampere	127 km
...	...
Sodankylä–Nuorgam	368 km

$\Longrightarrow$

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

yht. 1328 km

Sopiva matemaattinen malli tilanteelle: painotettu suuntaamaton verkko.



Kysymyksessä on lyhimmän polun etsiminen verkosta. Kurssin loppupuolella esitetään tehokas ratkaisumenetelmä nimeltään Dijkstran algoritmi.

Triviaaliratkaisu:

- käydään järjestyksessä läpi kaikki mahdolliset reitit (paikkakuntien järjestykset eli permutaatiot).
- oletetaan 30 paikkakuntaa $\Rightarrow$ $30! \approx 2,7 \cdot 10^{32}$ permutaatiota
- oletetaan kone testaa miljoona permutaatiota sekunnissa $\Rightarrow$ tarvitaan $8,5 \cdot 10^{18}$ vuotta.
- (Alkuräjähdyksestä on luokkaa 10^{10} vuotta.)

Heuristinen ratkaisu:

- ihminen ratkaisee ongelman käyttämällä hyväksi paikkakuntien maantieteellistä sijaintia
- voidaan käyttää myös tietokoneella (esim. A^*-algoritmi, *Tekoäly*)
- ei tarvita näin pieniin ongelmiin

Tällä kurssilla esitettävät algoritmit:

- Dijkstran algoritmi: laskee lyhimmät reitit esim. Helsingistä kaikille muille paikkakunnille
- Floydin algoritmi: laskee yhdellä kertaa lyhimmät etäisyydet minkä tahansa kahden paikkakunnan välillä

Kumpikin toimii näin pienillä aineistoilla "silmänräpäyksessä". (Dijkstra vaatii *karkeasti* samaa luokkaa olevan ajan kuin järjestäminen, Floyd vähän enemmän.)

Dijkstran algoritmissa keskeinen aputietorakenne on keko (eli kasa, engl. heap). Se tukee (mm.) seuraavia operaatioita:

- Insert(k): lisää avaimen k kekoon
- Min: palauttaa pienimmän keossa olevan avaimen
- DelMin: poistaa keosta pienimmän avaimen

Vastaavaa abstraktia tietotyyppiä sanotaan prioriteettijonoksi.

(Ideana on karkeasti, että "potentiaalisesti matkaa edistävät" yhteydet viedään kekoon, josta niitä poimitaan suuruusjärjestyksessä.)

Yhteenveto

Tietorakenteita ja abstrakteja tietotyyppejä tarvitaan

- ongelmien mallintamiseen
- suurten tietomäärien hallintaan
- laskennan välivaiheiden käsittelyyn.

Tietorakenteet ja algoritmit liittyvät erottamattomasti toisiinsa:

- tietorakenteiden toteuttamiseen tarvitaan algoritmeja
- oikeat tietorakenteet tekevät algoritmista tehokkaan.

Samat operaatiot voidaan yleensä toteuttaa usealla eri tavalla. Tilanteesta riippuu, mikä on paras.

Miksi pakollinen kurssi "Tietorakenteet"

. . . vaikka kurssilla käsiteltävät tietorakenteet ja algoritmit löytyvät valmiina erilaisista kirjastoista:

- kurssilla näemme (hieman), mitä on tarjolla
- sopivan (tehokkaan) tietorakenteen valitseminen ei aina ole helppoa
- perustietorakenteet ovat niin keskeinen osa tietojenkäsittelytiedettä, että ne pitää tuntea pintaa syvemmältä (myös toteutus)
- tietorakenteisiin liittyvät perusalgoritmit ovat opettavaisia esimerkkejä algoritmien suunnittelelutekniikoista.

Tietokoneet nopeutuvat, mutta tehokkuus on silti tärkeää:

- käsiteltävät datamäärät kasvavat
- halutaan ratkaista yhä vaikeampia ongelmia (tekoäly, grafiikka, ...)
- tehokkaan ja tehottoman ratkaisun ero voi olla niin suuri, että mikään laskentatehon kasvu ei kompensoi sitä.

Paradoksaalisesti laitetehojen parantuminen saattaa jopa tehdä algoritmin tehokkuudesta tärkeämpää:

- algoritmien tehokkuuserot näkyvät yleensä selvästi vasta suurilla syötteillä
- tehokkaat koneet mahdollistavat yhä suurempien syötteiden käsittelemisen.

Muuttuva ympäristö tuo myös uusia tehokkuushaasteita:

- sensoriverkot yms. tuottavat dataa jatkuvasti
  $\Rightarrow$ käsittelyn oltava (lähes) reaaliaikaista
- tarve rinnakkaistaa laskentaa
- mobiililaitteiden rajoitettu kapasitetti
- muistihierarkiat
- jne.

Tällä kurssilla keskitytään kuitenkin klassisiin perusasioihin.

Algoritmit: oikeellisuus ja vaativuus

Esitämme algoritmit yleensä pseudokoodina, joka

- on riittävän tarkka, että algoritmi osattaisiin toteuttaa mutta
- ei ota kantaa esim. ohjelmointikielestä riippuviin yksityiskohtiin.

Esimerkki 1.3: Seuraava algoritmi etsii kokonaislukutaulukon $A[1\,..\,n]$ pienimmän alkion indeksin:

```
Find-Smallest(A)
1   ind ← 1
2   for i ← 2 to length[A]
3         do
4              ▷ Invariantti: A[ind] = min A[1 .. i − 1]
5              if A[i] < A[ind]
6                 then ind ← i
7   return ind
```

(Rivi 4 on kommentti; palaamme sen sisältöön myöhemmin.)

Selityksiä pseudokoodimerkinnöille:

- sisennys ilmaisee lohkorakenteen (Javassa { ... });
- rivijako ja -numerointi ei merkityksellistä
- kommenttimerkki $\triangleright$, sijoitusmerkki $\leftarrow$, yhtäsuuruusvertailu $=$
- $A[i]$ taulukon i:s alkio; $A[i \mathinner{..} j]$ osataulukko $(A[i], A[i+1], \ldots, A[j])$
- Jos rakenteella C on esim. kentät key ja $data$, niihin viitataan $key[C]$ ja $data[C]$ (Javassa `C.key` ja `C.data`)
- loogiset ehdot evaluoidaan vasemmalta oikealle "oikosulkuisesti": jos esim. x on epätosi ja y ei määritelty, niin "x and y" on epätosi
- esimerkki oikosulkemisesta:

  $$\textbf{if } p \neq \text{Nil and } key[p] = \ldots$$

  missä Nil on nollaosoitin (ts. viite olioon, jota ei ole olemassa)
- silmukan **for** $i \leftarrow 1$ **to** n jälkeen muuttujan i arvona on $n+1$ (olettaen $n \geq 1$)

Algoritmista Find-Smallest näkee suoraan, että se on

- oikeellinen eli tuottaa halutun tuloksen ja
- aikavaativuudeltaan suunnilleen niin hyvä kuin mahdollista.

Käytämme sitä kuitenkin esimerkkinä havainnollistamaan oikeellisuuden ja aikavaativuuden analysoinnissa käytettäviä tekniikoita:

- silmukkainvariantti
- funktion kertaluokka ("iso-O-notaatio")

Silmukkainvariantti

Invarianteilla voidaan todistaa algoritmin oikeellisuus haluttaessa hyvinkin formaalisti. Meidän kannaltamme on kiinnostavampaa käyttää niitä algoritmien laatimiseen ja toimintaperiaatteen kuvailemiseen.

Algoritmin Find-Smallest for-silmukan invariantti on kirjoitettu kommentiksi silmukan alkuun:

$$A[ind] = \min A[1\,..\,i-1],$$

missä on käytetty merkintää

$$\begin{aligned} \min A[i\,..\,j] &= \min\{\,A[i], A[i+1], \ldots, A[j]\,\} \\ &= \text{pienin luvuista } A[i],\ A[i+1],\ \ldots,\ A[j]. \end{aligned}$$

Intuitiivisesti algoritmin Find-Smallest silmukkainvariantti

$$A[ind] = \min A[1\,..\,i-1],$$

kertoo, että silmukan periaate on seuraava:

1. Ensimmäisellä kierroksella ($i = 2$)
   - oletetaan, että silmukan ulkopuolella on saatettu voimaan $A[ind] = \min A[1\,..\,2-1] = A[1]$, ja
   - oletusta hyväksikäyttäen saatetaan voimaan $A[ind] = \min A[1\,..\,(2+1)-1] = \min A[1\,..\,2]$.

2. Toisella kierroksella ($i = 3$)
   - oletetaan, että edellisellä kierroksella on saatettu voimaan $A[ind] = \min A[1\,..\,3-1] = \min A[1\,..\,2]$, ja
   - oletusta hyväksikäyttäen saatetaan voimaan $A[ind] = \min A[1\,..\,(3+1)-1] = \min A[1\,..\,3]$.

3. ...

Yleisemmin silmukan invariantti on (ohjelman tietorakenteita koskeva looginen) väittämä, joka

1. on voimassa kun silmukkaan tullaan ensimmäistä kertaa

2. pysyy voimassa kun silmukkaa suoritetaan yksi kierros ja

3. yhdessä silmukan lopetusehdon kanssa sanoo jotain mielenkiintoista.

Huomaa, että ehdoista 1 ja 2 yhdessä seuraa, että invariantti on voimassa silmukan jokaisen suorituskerran alussa ja myös silmukasta poistuttaessa. (Tämä on analogista luonnollisten lukujen induktion kanssa.)

Silmukkainvariantti on samanhenkinen suunnittelu- ja todistustekniikka kuin aliohjelmien esi- ja jälkiehdot (precondition, postcondition).

Esimerkin invariantilla on halutut ominaisuudet:

1. Aluksi $ind = 1$ ja $i = 2$, joten

$$\min A[1 \mathinner{..} i-1] = \min A[1 \mathinner{..} 1] = A[1] = A[ind].$$

2. Silmukan runkoa suoritettaessa joko

(a) $A[i] < \min A[1 \mathinner{..} i-1]$, jolloin $\min A[1 \mathinner{..} (i+1)-1] = A[i]$ ja ind saa arvon i, tai

(b) $A[i] \geq \min A[1 \mathinner{..} i-1]$, jolloin $\min A[1 \mathinner{..} (i+1)-1] = \min A[1 \mathinner{..} i-1]$ ja ind pysyy ennallaan,

joten kummassakin tapauksessa invariantti pysyy voimassa.

3. Silmukan päättyessä $i = length[A] + 1$, joten $\min A[1 \mathinner{..} i-1]$ on haluttu taulukon pienin alkio.

Algoritmin vaativuus

Vaativuutta tarkastellaan erilaisten resurssien kannalta:

- laskentaan kuluva aika
- laskennan vaatima muistitila
- tarvittavan tietoliikenteen määrä
- ...

Tällä kurssilla esiintyvillä algoritmeilla mielenkiintoiset erot ovat yleensä aikavaativuudessa, joten keskitymme siihen. Myös tilavaativuutta (muistintarvetta) tarkastellaan jonkin verran.

Jos algoritmien tilavaativuudessa on suuria eroja, tämä on usein ratkaisevaa.

Tarkastelemme resursseja lähinnä skaalautuvuuden kannalta: kuinka algoritmin/tietorakenteen resurssitarve kasvaa, kun syötteen koko kasvaa.

Arvioimme siis aika- ja tilavaativuutta syötteen koon n funktiona. Esim. kutsulla Find-Smallest(A) syötteen kooksi ajatellaan $n = length[A]$.

Olemme kiinnostuneet tapauksesta, jossa n on suuri. Tämä antaa yleiskuvan siitä, mitkä algoritmit ja tietorakenteet soveltuvat minkinlaisiin tilanteisiin.

Yksittäisessä sovelluksessa on tietysti tärkeämpää suorituskyky: millaiset vasteajat saadaan sillä kuormalla, joka järjestelmässä oikeasti esiintyy.

Esimerkki 1.4: Tarkastellaan edelleen algoritmia Find-Smallest. Oletetaan ensin, että pseudokoodin kunkin riviin i suorittaminen yhden kerran vaatii ajan c_i. Tarkastellaan, montako kertaa kukin rivi suoritetaan, kun $length[A] = n$.

Find-Smallest(A)

		aika	suorituskertojen lkm.
1	$ind \leftarrow 1$	c_1	1
2	for $i \leftarrow 2$ to $length[A]$	c_2	$n-1$
3	do	0	$n-1$
4	$\triangleright$ . . .	0	$n-1$
5	if $A[i] < A[ind]$	c_5	$n-1$
6	then $ind \leftarrow i$	c_6	$0 \leq$ lkm. $\leq n-1$
7	return ind	c_7	1

Tässä c_2 esittää for-silmukan laskurin kasvattamista ja testaamista ja c_5 if-ehdon testaamista jne.

Paras tapaus: Suoritusaika on pienin, jos $A[1]$ on taulukon pienin alkio. Tällöin riviä 6 ei suoriteta kertaakaan. Aikaa kuluu

$$\begin{aligned} T_{\text{min}}(n) &= (c_2 + c_5)(n-1) + c_1 + c_7 \\ &= a_1 n + b_1, \end{aligned}$$

missä $a_1 = c_2 + c_5$ ja $b_1 = c_1 - c_2 - c_5 + c_7$.

Pahin tapaus: Suoritusaika on suurin, jos A on laskevassa järjestyksessä, jolloin rivi 6 suoritetaan joka kerta. Aikaa kuluu

$$\begin{aligned} T_{\text{max}}(n) &= (c_2 + c_5 + c_6)(n-1) + c_1 + c_7 \\ &= a_2 n + b_2, \end{aligned}$$

missä $a_2 = c_2 + c_5 + c_6$ ja $b_2 = c_1 - c_2 - c_5 - c_6 + c_7$.

Tässä esimerkissä parhaan ja pahimman tapauksen välillä ei näyttäisi olevan suurta eroa.

Yleensä paras tapaus ei ole hyödyllinen algoritmien vertailussa, sillä mikä tahansa algoritmi yleensä toimii hyvin jollain syötteellä.

Pahin tapaus taas on toisinaan turhankin pessimistinen.

Voidaan analysoida myös keskimääräistä tapausta: mikä on aikavaativuuden odotusarvo, jos A on satunnaisessa järjestyksessä. Tämäkin on ongelmallista:

- tarvitaan kyseenalaisia ja epärealistisia oletuksia syötteen jakaumasta
- silti analyysi on teknisesti erittäin vaikeaa
- odotusarvo on usein huono tunnusluku jakaumalle (ja mediaanin tms. analysoiminen vielä vaikeampaa).

Tyydymme siksi käyttämään pahimman tapauksen aikavaativuutta tehokkuusmittarina. **Yleensä** tämä antaa oikean kuvan tilanteesta.

Vakiot $c_1, \ldots, c_7$ ovat monella tapaa ongelmallisia:

- todelliset ajat riippuvat laitteistosta ja ohjelmointikielestä
- annetulle laitteistolle jne. ajat riippuvat järjestelmän muusta kuormasta yms.
- tulokset ovat hankalia ja epähavainnollisia

Siis vakioiden tarkka määrittäminen on (a) mahdotonta ja (b) hyödytöntä.

Tämän takia määrittelemme kohta suuruusluokkamerkinnän, jota käyttäen "unohdamme" ikävät vakiot ja kirjoitamme

$$T_{\mathsf{min}}(n) = O(n) \quad \text{ja} \quad T_{\mathsf{max}}(n) = O(n).$$

Tämä sanoo suunnilleen, että suurilla n syötteen koon kaksinkertaistaminen kaksinkertaistaa myös laskenta-ajan.

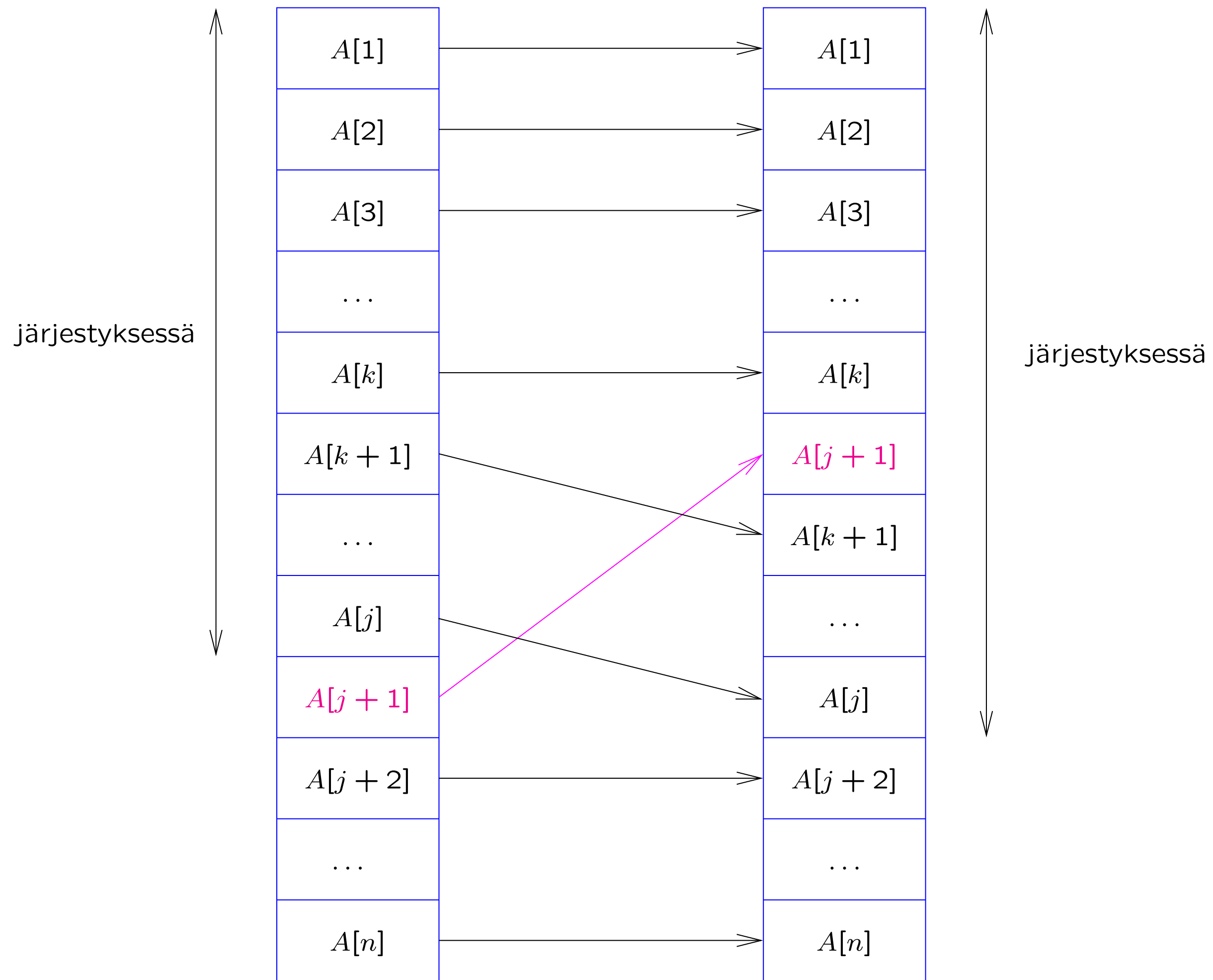
Tarkastellaan oikeellisuutta ja aikavaativuutta vähemmän triviaalissa esimerkissä:

Esimerkki 1.5: Lisäysjärjestäminen (insertion sort): Algoritmi järjestää taulukon $A[1 \mathinner{.\,.} n]$ vaiheittain. Vaiheessa j

- oletetaan, että $A[1 \mathinner{.\,.} j]$ on järjestyksessä
- järjestetään $A[1 \mathinner{.\,.} j+1]$ työntämällä $A[j+1]$ sopivaan väliin $A[1 \mathinner{.\,.} j]$:ssä

Seuraavan sivun kuva esittää lisäysjärjestämisen vaihetta j, kun

$$A[1] \leq A[2] \leq \ldots \leq A[k] \leq A[j+1] \leq A[k+1] \leq \ldots \leq A[j].$$


järjestyksessä
$A[1]$
$A[2]$
$A[3]$
...
$A[k]$
$A[k+1]$
...
$A[j]$
$A[j+1]$
$A[j+2]$
...
$A[n]$
$A[1]$
$A[2]$
$A[3]$
...
$A[k]$
$A[j+1]$
$A[k+1]$
...
$A[j]$
$A[j+2]$
...
$A[n]$
järjestyksessä

Lisäysjärjestäminen pseudokoodina:

```
Insertion-Sort(A)
1   for j ← 2 to length[A]
        do
          ▷ Invariantti: A[1 . . j − 1] järjestyksessä
2         key ← A[j]
3         i ← j − 1
4         while i > 0 and A[i] > key
              do
5               A[i + 1] ← A[i]
6               i ← i − 1
7         A[i + 1] ← key
```

Merkitään $n = length[A]$. Silmukasta poistuttaessa $j = n + 1$, joten jos invariantti on voimassa, koko taulukko $A = A[1\,..\,(n+1) - 1]$ on järjestyksessä.

Silmukkaan ensi kertaa tullessa $j = 2$. Siis $A[1\,..\,j-1]$ on yhden alkion kokoinen ja invariantti varmasti pätee.

Oletetaan, että kierroksen $j - 1$ jälkeen $A[1\,..\,j]$ on järjestyksessä. Olkoon k kuten edellä:

$$A[1] \leq A[2] \leq \ldots \leq A[k] \leq A[j+1] < A[k+1] \leq \ldots \leq A[j];$$

jos $A[j+1] \geq A[j]$, niin $k = j$ ja

$$A[1] \leq A[2] \leq \ldots \leq A[j] \leq A[j+1].$$

Tarkastelemalla while-silmukkaa nähdään, että sen suorituksen jälkeen

- $A[1\,..\,k]$ on sama kuin ennen
- uusi $A[k+1]$ on vanha $A[j+1]$
- uusi $A[k+2\,..\,j+1]$ on vanha $A[k+1\,..\,j]$.

Oletuksen mukaan

- vanha $A[1\,..\,k]$ ja vanha $A[k+1\,..\,j]$ ovat järjestyksessä ja
- Vanhat $A[k]$, $A[k+1]$ ja $A[j+1]$ ovat järjestyksessä $A[k] \leq A[j+1] < A[k+1]$.

Siis uusi $A[1\,..\,j+1]$ on järjestyksessä.

Olemme todenneet, että algoritmi toimii oikein.

Arvioidaan suoritusaikaa samoin kuin edellä. Olkoon t_j while-silmukan ehtotestien suorituskerrat kierroksella j.

Insertion-Sort(A)

		aika	kertaa
1	for $j \leftarrow 2$ to $length[A]$	c_1	n
	do		
	$\triangleright$ ...	0	$n-1$
2	$key \leftarrow A[j]$	c_2	$n-1$
3	$i \leftarrow j-1$	c_3	$n-1$
4	while $i > 0$ and $A[i] > key$	c_4	$\sum_{j=2}^{n} t_j$
	do		
5	$A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6	$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$A[i+1] \leftarrow key$	c_7	$n-1$

Aikaa siis kuluu

$$c_1 n + (c_2 + c_3 + c_7)(n-1) + c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1).$$

Paras tapaus: Jos taulukko on valmiiksi järjestyksessä, niin $t_j = 1$ kaikilla j. Aikavaativuus on

$$\begin{aligned} T_{\text{min}}(n) &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= a'n + b', \end{aligned}$$

missä $a' = c_1 + c_2 + c_3 + c_4 + c_7$ ja $b' = -(c_2 + c_3 + c_4 + c_7)$.

Kuten edellisessä esimerkissä, parhaan tapauksen aikavaativuus on lineaarinen ja syötteen kaksinkertaistaminen kaksinkertaistaa laskenta-ajan.

Pahin tapaus: Jos taulukko on käänteisessä järjestyksessä, niin $t_j = j$ kaikilla j. Aikavaativuus on

$$\begin{aligned}
T_{\text{max}}(n) &= c_1 n + (c_2 + c_3 + c_7)(n-1) \\
&\quad + c_4 \sum_{j=2}^{n} j + (c_5 + c_6) \sum_{j=2}^{n} (j-1) \\
&= c_1 n + (c_2 + c_3 + c_7)(n-1) + c_4 \frac{n+2}{2}(n-1) \\
&\quad + (c_5 + c_6) \frac{n+1}{2}(n-1) \\
&= an^2 + bn + c,
\end{aligned}$$

missä $a = (c_4 + c_5 + c_6)/2$, $b = c_1 + c_2 + c_3 + c_4/2 + c_7$ ja $c = -c_2 - c_3 - c_4 - c_5/2 - c_6/2 - c_7$. Tässä on käytetty kaavaa

$$\sum_{k=p}^{q} k = \frac{p+q}{2}(q-p+1).$$

Siis $T_{\text{max}}(n)$ on neliöllinen: pahimmassa tapauksessa syötteen kaksinkertaistaminen suunnilleen nelinkertaistaa laskenta-ajan.

Funktioiden kertaluokat

Haluamme päästä eroon edellisten aikavaativuusarvioiden vakioista c_1, c_2 jne.

Olkoot f ja g funktioita $\mathbb{N} \to \mathbb{R}$. Ajatus on, että f on jonkin algoritmin aikavaativuus (esim. T_{max} edellä) ja g sen sopivasti yksinkertaistettu versio.

Sanomme, että f kuuluu kertaluokkaan $O(g)$, ja kirjoitamme $f = O(g)$, jos

> voidaan valita sellaiset vakiot $d > 0$ ja $n_0 > 0$, että
> kaikilla $n \in \mathbb{N}$ pätee:
>
> jos $n \geq n_0$ niin $0 \leq f(n) \leq dg(n)$.

Motivaatio:

- algoritmin käyttäytyminen pienillä syötteillä usein epätyypillistä, joten "liian pienet" syötteet ($n < n_0$) voidaan jättää huomiotta
- suurillakin syötteillä vakiokertoimet jätetään edellä esitetyistä syistä huomiotta.

Esimerkki 1.6: Olkoon $T(n) = an + b$ ja $g(n) = n$ kaikilla n, missä $b \geq 0$ ja $c \geq 0$. Valitaan $d = a + 1$ ja $n_0 = b$. Jos nyt $n \geq n_0$, niin

$$T(n) = an + b = an + n_0 \leq an + n = (a + 1)n = dn.$$

Siis $T = O(g)$. Yleensä merkitsemme tämän yksinkertaisemmin (ja epätäsmällisemmin)

$$T(n) = O(n).$$

Esimerkki 1.7: Olkoon $T(n) = an^2 + bn + c$, missä a, b ja c ovat ei-negatiivisia. Väitämme, että $T(n) = O(n^2)$. Valitaan $n_0 = 1$ ja $d = a + b + c$. Kun $n \geq n_0$, pätee

$$T(n) = an^2 + bn + c \leq an^2 + bn \cdot n + c \cdot n^2 = (a + b + c)n^2 = dn^2$$

eli haluttu yläraja.

Yleisemminkin polynomifunktiossa ylimmänasteinen termi antaa kertaluokan: jos $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ missä $a_k > 0$ ja k on vakio, niin $T(n) = O(n^k)$.

Toinen tapa ajatella sama asia: Kertaluokka antaa ylärajan funktion kasvunopeudelle. Jos funktiossa on useita eri nopeudella kasvavia termejä, nopeimmin kasvava voittaa.

Edellisen perusteella voimme todeta, että algoritmin Find-Smallest aikavaativuus on $O(n)$ sekä parhaassa että pahimmassa tapauksessa. Lisäysjärjestämisen pahimman tapauksen aikavaativuus on $O(n^2)$, mutta parhaan tapauksen aikavaativuus vain $O(n)$.

O-merkintä ilmaisee vain ylärajan funktion kasvunopeudelle. Jos T on lisäysjärjestämisen parhaan tapauksen aikavaativuus, niin sekä $T(n) = O(n)$ että $T(n) = O(n^2)$ ovat tosia väittämiä, mutta ensimmäinen on huomattavasti informatiivisempi.

Otamme käyttöön vastaavan merkinnän alarajoille. Sanomme, että f kuuluu kertaluokkaan $\Omega(g)$, ja kirjoitamme $f = \Omega(g)$ (tai $f(n) = \Omega(g(n))$), jos

> voidaan valita sellaiset vakiot $d > 0$ ja $n_0 > 0$, että
> kaikilla $n \in \mathbb{N}$ pätee:
>
> $$\text{jos } n \geq n_0 \text{ niin } 0 \leq dg(n) \leq f(n).$$

Esimerkki 1.8: Olkoon $T(n) = an + b$, missä $a, b > 0$. Nyt $T(n) = \Omega(n)$, sillä valitsemalla $n_0 = 1$ ja $d = a$ saadaan $T(n) \geq dn$ kun $n \geq n_0$.

Toisaalta ei päde $T(n) = \Omega(n^2)$: Olkoot d ja n_0 mielivaltaiset. Jos valitaan $n = \max\{1, n_0, (a + b + 1)/d\}$, niin

$$T(n) = an + b \leq (a + b)n < dn^2$$

eli ehto ei toteudu.

Määrittelemme vielä, että f kuuluu kertaluokkaan $\Theta(g)$, ja kirjoitamme $f = \Theta(g)$ (tai $f(n) = \Omega(g(n))$), jos

> voidaan valita sellaiset vakiot $d_1 > 0$, $d_2 > 0$ ja $n_0 > 0$, että kaikilla $n \in \mathbb{N}$ pätee:
>
> $$\text{jos } n \geq n_0 \text{ niin } 0 \leq d_1 g(n) \leq f(n) \leq d_2 g(n).$$

Helposti nähdään, että seuraavat ehdot ovat yhtäpitäviä:

- $f = \Theta(g)$
- $f = O(g)$ ja $f = \Omega(g)$
- $f = O(g)$ ja $g = O(f)$.

Intuitiivisesti

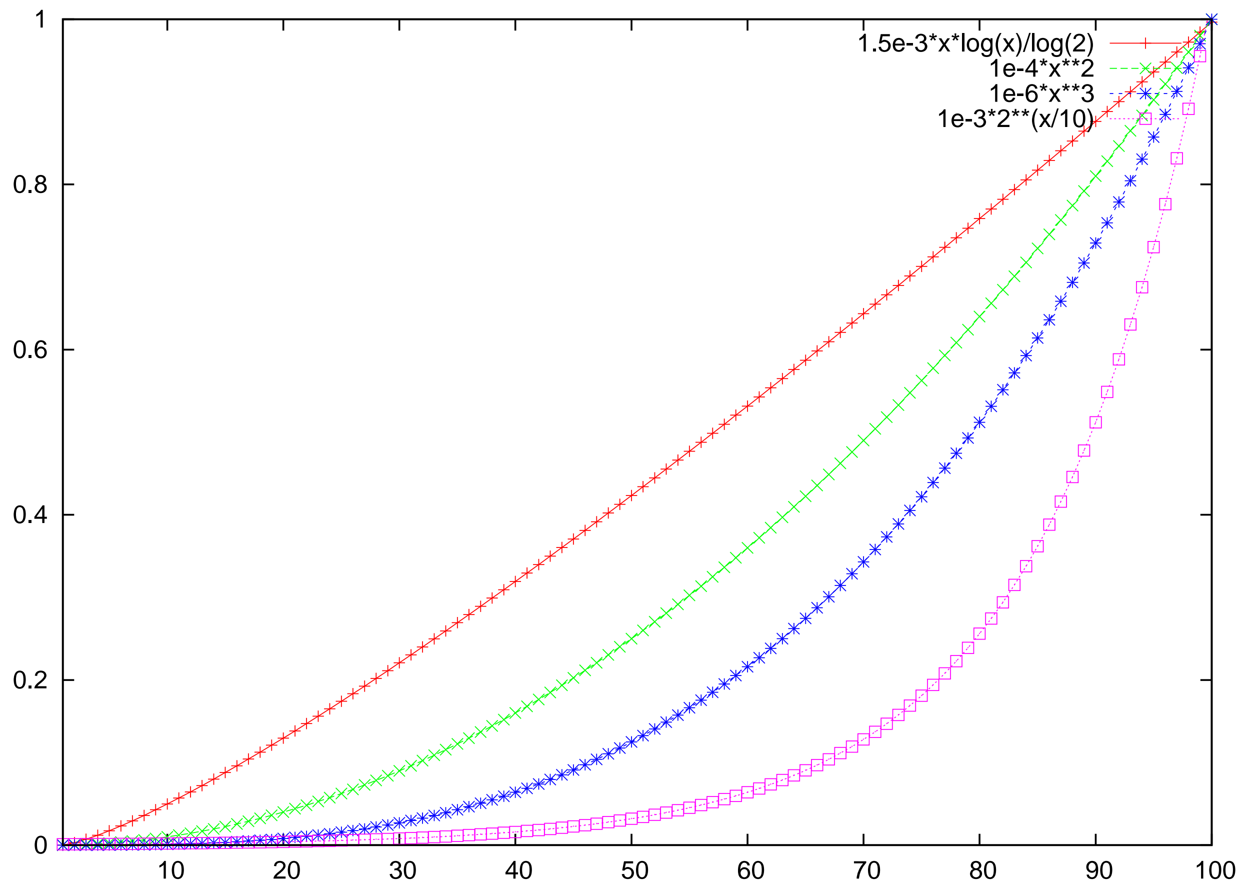
- $f = O(g)$ tarkoittaa, että f kasvaa korkeintaan yhtä nopeasti kuin g
- $f = \Omega(g)$ tarkoittaa, että f kasvaa ainakin yhtä nopeasti kuin g
- $f = \Theta(g)$ tarkoittaa, että f kasvaa tasan yhtä nopeasti kuin g.

Yleisesti esiintyviä aikavaativuusluokkia:

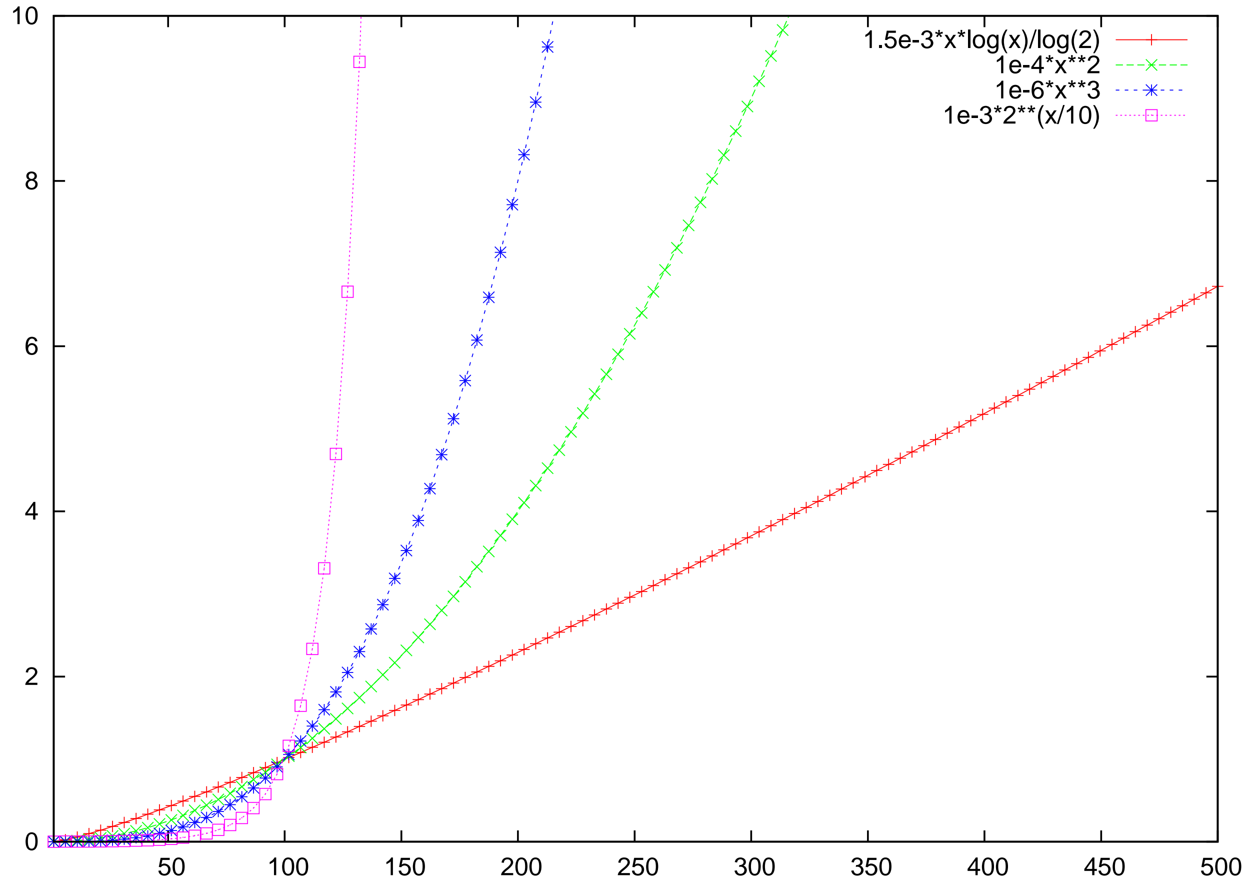
- $O(1)$ (vakio) [alkion hakeminen taulukosta]
- $O(\log n)$ (logaritminen) [indeksoitu haku tietokannasta, jossa n tietuetta]
- $O(n)$ (lineaarinen) [yksinkertainen perustietojenkäsittely]
- $O(n \log n)$ [tehokkaat järjestämisalgoritmit jne.]
- $O(n^2)$, $O(n^3)$ jne. (polynominen) [tehokkaat verkkoalgoritmit]
- $O(2^n)$, $O(3^n)$ jne. (eksponentiaalinen) [tekoälyongelmat jne.]

Polynomisten ja eksponentiaalisten algoritmien skaalautuvuudessa on merkittävä ero.

- Eksponentiaaliset (ja korkea-asteiset polynomiset) käytännöllisiä vain pienillä syötteillä
- "Pieni" voi olla esim. $n = 50$ tai $n = 5000$
- Eksponentiaalisilla aikavaativuuksilla on eroja! 3^n **ei ole** $O(2^n)$!



Tarkastellaan neljää algoritmia, joiden aikavaativuudet ovat $T_1(n) = an \log_2 n$, $T_2(n) = bn^2$, $T_3(n) = cn^3$ ja $T_4(n) = d2^{n/10}$. Vakiot a, b, c ja d valittu siten, että jokainen algoritmi vie yhden yksikön aikaa, kun $n = 100$.



Kuinka paljon suurempia syötteitä voidaan käsitellä, jos aikaa onkin 10 yksikköä? Algoritmi 1: $n \approx 710$; algoritmi 2: $n \approx 320$; algoritmi 3: $n \approx 220$; algoritmi 4: $n \approx 130$.

Pikakertaus: logaritmi

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

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

Kaksikantainen logaritmi $\log_2 n$ kertoo

- luonnollisen luvun n pituuden bitteinä (miinus 1)
- montako kertaa n pitää puolittaa, että päästään alle 2:n.

Laskusääntöjä:

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

Aikavaativuuden perussäännöt

"Riittävän yksinkertaiset" perusoperaatiot vievät vakioajan (voidaan toteuttaa vakiomäärällä konekäskyjä):

- vakiokokoisen muuttujan sijoitusoperaatio
- aritmeettiset perustoimitukset
- yksinkertaiset syöttö- ja tulostusoperaatiot
- jne.

Kun tarkastelemme vain kertaluokkia, tämä riittää.

Toisinaan kuitenkin asiaa kannattaa miettiä tarkemmin:

- aritmeettiset operaatiot hyvin suurilla luvuilla
- haku levymuistista
- jne.

Peräkkäisyys: Jos S_1:n aikavaativuus on $O(f_1(n))$ ja S_2:n aikavaativuus on $O(f_2(n))$, niin yhdistelmän $\{S_1; S_2;\}$ aikavaativuus on $O(f_1(n) + f_2(n))$.

Perustelu: Olkoon n syötteen koko ja d_1, d_2, n_1 ja n_2 sellaiset, että

- kun $n \geq n_1$, niin S_1 menee ajassa $d_1 f_1(n)$.
- kun $n \geq n_2$, niin S_2 menee ajassa $d_2 f_2(n)$.

Pitää löytää d ja n_0 s.e.

- kun $n \geq n_0$, niin $S_1; S_2$ menee ajassa $d_1 f_1(n)$.

Voidaan valita $d = d_1 + d_2$ ja $n_0 = \max\{n_1, n_2\}$. Jos $n \geq \max\{n_1, n_2\}$, pätee näet

$$\begin{aligned} \{S_1; S_2;\}\text{:n aika} &= S_1\text{:n aika} + S_2\text{:n aika} \\ &\leq d_1 f_1(n) + d_2 f_2(n) \\ &\leq (d_1 + d_2)(f_1(n) + f_2(n)). \end{aligned}$$

Edellisen säännön yhteydessä on hyödyllistä, että

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

Tämä seuraa siitä, että $a + b \le 2 \max\{\, a, b\,\}$ kaikilla a, b.

Yleisemminkin

$$f_1(n) + \ldots + f_k(n) = O(\max\{\, f_1(n), \ldots, f_k(n)\,\})$$

mille tahansa vakiolle k (ts. termien määrä ei saa riippua muuttujasta n).

Ehtolause: Edellisten perusteella ehtolauseen

if E then P; else Q

aikavaativuus on $O(\max\{\, f_E(n), f_P(n), f_Q(n)\,\})$, missä f_E on ehtolausekkeen E evaluoinnin aikavaativuus ja f_P ja f_Q haarojen P ja Q aikavaativuudet.

Silmukat: for-silmukan

```
for i ← a to b do
    S;
```

aikavaativuus on

$$f(n) = \sum_{i=a}^{b} f_S(n, i),$$

missä $f_S(n, i)$ on silmukan rungon S aikavaativuus, joka tyypillisesti riippuu silmukkamuuttujasta i. Yleensä a tai b riippuu syötteen koosta, joten max-sääntöä ei voi soveltaa.

Edelleen kahdelle sisäkkäiselle silmukalle

```
for i ← a to b do
    for j ← c to d do
        S;
```

saadaan

$$f(n) = \sum_{i=a}^{b} \sum_{j=c}^{d} f_S(n, i, j),$$

missä S:n aikavaativuus voi riippua sekä i:stä että j:stä, jne.

Yleensä c tai d riippuu ulommasta silmukkamuuttujasta i tai syötteen koosta n, joten taaskaan max-sääntöä ei voi soveltaa.

Summien

$$f(n) = \sum_{i=a}^{b} f_S(n, i)$$

yksinkertaistamisessa on toisinaan hyötyä erilaisista summakaavoista.

Usein kuitenkin saadaan riittävän tarkka tulos tekemällä turvallinen yläraja-arvio

$$\sum_{i=a}^{b} f_S(n, i) = O((b - a + 1) \max_i f_S(n, i))$$

eli arvioimalla

$$\text{summa} \leq (\text{termien lkm}) \cdot (\text{suurin termi}).$$

Tämä olettaa $a \leq b$. Jos $a > b$, niin silmukkaa ei suoriteta ja aikavaativuus on vakio (ehdon "$a \leq b$" testaamiseen kuluva aika).

Tarkastellaan sitten while-silmukkaa

while E do S;

Jos tiedetään, että silmukka suoritetaan $O(k(n))$ kertaa ja E ja S menevät joka kerta ajassa $O(f_E(n))$ ja $O(f_S(n))$, niin silmukan aikavaativuus on

$$O(k(n) \cdot \max\{\, f_E(n), f_S(n)\,\}).$$

Yleisessä tapauksessa suureen $k(n)$ määrittäminen voi olla mahdotonta, mutta normaalissa tietorakenteiden ohjelmoinnissa sille yleensä löytyy kohtuullinen yläraja.

Esimerkki 1.9: vaikeasti analysoitava while-silmukka. Tarkastellaan eri alkuarvoilla a silmukkaa

$n \leftarrow a$
while $n > 1$
  do if n on parillinen
    then $n \leftarrow n/2$
    else $n \leftarrow 3n + 1$

On kuuluisa avoin ongelma, pysähtyykö silmukka kaikilla alkuarvoilla a. Yleisesti uskotaan, että pysähtyy; tämä väite tunnetaan Collatzin konjektuurina. □

Aliohjelmakutsu: Tarkastellaan seuraavasti määriteltyä aliohjelmaa:

$P(x_1, \ldots, x_k)$
    S.

Siis aliohjelman P runkoa merkitään S. Kutsu

$$P(a_1, \ldots, a_k)$$

analysoidaan korvaamalla se jonolla

1 $x_1 \leftarrow a_1$
2 $\ldots$
3 $x_k \leftarrow a_k$
4 S.

Tämä tietenkään ei toimi, jos P on rekursiivinen. Rekursiivisia aliohjelmia analysoidaan palautuskaavoilla, joita esitellään myöhemmin (lyhyesti).

Esimerkki: binäärihaku

Oletetaan, että kokonaislukutaulukko $A[1\,..\,n]$ on (kasvavassa) järjestyksessä. Seuraava algoritmi selvittää, onko luku x taulukossa:

```
Binary-Search(A[1..n], x)
 1  left ← 1
 2  right ← n
 3  while left < right
 4        do
 5              mid ← ⌊(left + right)/2⌋
 6              if A[mid] < x
 7                 then left ← mid + 1
 8                 else if A[mid] > x
 9                         then right ← mid − 1
10                         else right ← left ← mid
11  return A[left] = x
```

Tässä $\lfloor \cdot \rfloor$ on alaspäin pyöristys: $\lfloor z \rfloor$ on suurin kokonaisluku k, jolla $k \leq z$.
(Vastaavasti $\lceil z \rceil$ on pienin kokonaisluku k, jolla $k \geq z$.)
Merkintä $x \leftarrow y \leftarrow a$ tarkoittaa, että sekä x että y saavat arvon a.

Algoritmin (ja sen analyysin) idea on suunnilleen seuraava:

- Kaikki alkion x mahdolliset esiintymät pidetään välillä $\mathit{left}\,..\,\mathit{right}$.
- Silmukan jokaisella kierroksella tämä väli lyhenee puoleen entisestä.
- Silmukka päättyy, kun välin pituus on 1.
- Siis silmukan kesto on sama kuin puolitusten lukumäärä, että n:stä päästään 1:een, eli $\log_2 n$.

Seuraavaksi esitetään tämä hieman täsmällisemmin.

Ensin toteamme algoritmin oikeellisuuden. Väitämme, että silmukalla on invariantti

> jos x esiintyy taulukossa $A[1..n]$
> niin x esiintyy osataulukossa $A[\mathit{left}..\mathit{right}]$.

Silmukasta poistuttaessa $\mathit{left} \geq \mathit{right}$, joten $A[\mathit{left}..\mathit{right}]$ voi sisältää vain yhden alkion. Jos invariantti pätee ja x on taulukossa, sen on oltava tämä ainoa alkio eli $A[\mathit{left}] = x$. Siis algoritmi toimii oikein, jos voimme osoittaa invariantin pätevän.

Selvästi invariantti pätee silmukkaan tultaessa, koska tällöin $A[1..n] = A[\mathit{left}..\mathit{right}]$. Todetaan vielä, että invariantti myös pysyy voimassa.

Oletetaan, että invariantti pätee ennen silmukan rungon suorittamista. Jos x ei esiinny taulukossa, niin invariantti triviaalisti pätee aina. Oletetaan siis, että x on taulukossa, ja siis myös osataulukossa $A[left\,..\,right]$.

1. Jos $A[mid] < x$, niin x ei esiinny osataulukossa $A[left\,..\,mid]$, joten se esiintyy osataulukossa $A[mid+1\,..\,right]$.

2. Jos $A[mid] > x$, niin x ei esiinny osataulukossa $A[mid\,..\,right]$, joten se esiintyy osataulukossa $A[left\,..\,mid-1]$.

3. Jos $A[mid] = x$, niin x tietysti esiintyy osataulukossa $A[mid\,..\,mid]$.

Joka tapauksessa rajojen päivityksen jälkeen edelleen x on osataulukossa $A[left\,..\,right]$. Invariantti pätee.

Siis algoritmi toimii oikein edellyttäen, että silmukka ylipäänsä pysähtyy. Analysoidaan nyt tätä.

Olkoon $m = right - left + 1$ osataulukon $A[left..right]$ koko.

Jos $m = 0$ tai $m = 1$, niin silmukka pysähtyy.

Tarkastellaan sitten tapausta, että $m \geq 2$ ja m on parillinen. Siis $m = 2p$ missä $p \geq 1$. Osataulukkoon $A[left..mid - 1]$ tulee $p - 1$ alkiota ja osataulukkoon $A[mid + 1..right]$ tulee p alkiota. Siis seuraavalla kierroksella osataulukon $A[left..right]$ koko on korkeintaan $p = m/2$.

Tarkastellaan vielä tapausta, että $m \geq 3$ ja m on pariton. Siis $m = 2p + 1$ missä $p \geq 1$. Osataulukoihin $A[left..mid - 1]$ ja ja $A[mid + 1..right]$ tulee kumpaankin p alkiota. Siis seuraavalla kierroksella osataulukon $A[left..right]$ koko on korkeintaan $p < m/2$.

Olkoon nyt m_i osataulukon $A[\mathit{left}\,..\,\mathit{right}]$ koko, kun silmukkaa on suoritettu i kierrosta.

- Aluksi koko taulukko on mukana: $m_0 = n$.
- Jokaisella kierroksella mukana oleva osuus pienenee ainakin puolella: $m_{i+1} \leq m_i/2$.

Tästä seuraa induktiolla, että jos silmukkaa ylipäänsä suoritetaan i kierrosta, niin $m_i \leq n/2^i$.

Olkoon k suoritettavien kierrosten lukumäärä. Koska $k-1$ kierroksen jälkeen suoritus vielä jatkuu, niin

$$m_{k-1} \geq 2.$$

Toisaalta edellisen puoliintumistarkastelun perusteella

$$m_{k-1} \leq n/2^{k-1}.$$

Yhdistämällä edelliset saadaan

$$2 \leq m_{k-1} \leq n/2^{k-1}$$

eli

$$2^k \leq n.$$

Ottamalla logaritmi puolittain saadaan silmukan suorituskertojen lukumäärälle yläraja

$$k \leq \log_2 n.$$

Silmukan runko menee selvästi vakioajassa, joten algoritmin aikavaativuus on $O(\log n)$.

2. Joukko ja sen perustoteutukset: pino, jono, lista

Tarkastelemme abstraktia tietotyyppiä joukko, joka tässä on ajan mukana muuttuva kokoelma avaimella yksilöitäviä alkioita.

Tarkastelemme ensin erikoistapauksia pino (stack) ja jono (queue):

- sopiva johdatus asiaan
- itsessään tärkeitä tietorakenteita
- eivät toteuta yleisiä joukko-operaatioita.

Linkitetty lista on eräs tapa toteuttaa yleinen joukko. Myös varsinaisesta linkitetystä listasta on runsaasti variaatioita (kahteen suuntaan linkitetty, rengas, tunnussolmullinen, . . .).

Joukko abstraktina tietotyyppinä

Tietotyyppinä joukko on kokoelma alkioita:

- alkoita voidaan lisätä ja poistaa
- yleensä alkiot ovat tietueita, joissa on avain ja muuta dataa
- esim. avain = opiskelijanumero ja
  data = opiskelijan nimi, osoite ja puhelinnumero
- jatkossa usein avaimille on määritelty järjestys (numerojärjestys, aakkosjärjestys, . . .)
- datan muoto ja sisältö voivat olla mitä vaan
- toteutuksen kannalta avain ja data voidaan tallentaa yhteen tietueeseen, tai avaimeen voidaan liittää osoitin dataan (”satelliittitietue”)

Tällaista tietotyyppiä kutsutaan myös dynaamiseksi joukoksi erotukseksi matemaattisista joukoista.

Tietotyypin "joukko" alkioihin liittyy lisäksi osoite, joka kertoo alkion sijainnin tietorakenteessa.

- tyypillisesti osoitin tietueen sijaintiin tietorakenteessa
- yksityiskohdat vaihtelevat sen mukaan, mitä tietorakennetta käytetään
- nopeuttaa toimintaa tietyissä tilanteissa

Joukko voidaan toteuttaa useilla erilaisilla tietorakenteilla:

- taulukko
- erilaiset linkitetyt listat
- hakupuu (joka voi olla tasapainotettu)
- hajautusrakenne
- keko

Kurssin alkupuoli kuluu näiden läpikäyntiin suunnilleen tässä järjestyksessä (mutta taulukosta ei ole paljon sanottavaa).

Toteutukseen käytetystä tietorakenteesta riippumatta joukolle on määritelty seuraavat operaatiot:

search(S, k): etsii avaimen k joukosta S ja palauttaa osoitteen tietueeseen, jonka avain k on

insert(S, x): lisää joukkoon osoittimen x osoittaman tietueen (avain ja data)

delete(S, x): poistaa osoittimen x osoittaman tietueen

min(S) / max(S): palauttaa osoittimen avaimeltaan pienimpään / suurimpaan tietueeseen joukossa

pred(S, x) / succ(S, x): palauttaa osoittimen tietueeseen, joka *avainten määräämässä järjestyksessä* välittömästi edeltää / seuraa x:n osoittamaa

Eri operaatioiden aikavaativuus sen sijaan vaihtelee huomattavasti tietorakenteen mukaan:

operaatio	lista1 (ei järj.)	lista2 (järj.)	tasap. hakupuu	hajautus
search	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
delete	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$
min, max	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$
succ, pred	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$

Jatkossa käsitellään

- eri tietorakenteiden toteutusta
- operaatioiden nopeutta
- valintaa eri vaihtoehtojen välillä.

Pino ja jono

Pinossa poisto kohdistuu viimeksi lisättyyn alkioon
(last in, first out eli LIFO).

- vrt. puhtaat lautaset ruokalassa
- usein tehokas tapa hoitaa asiat.

Jonossa poisto kohdistuu ensiksi lisättyyn alkioon
(first in, first out eli FIFO)

- vrt. asiakkaat ruokalassa
- käytetään jos "reiluus" tärkeää.

Sekä pino että jono voidaan toteuttaa joko taulukkona tai linkitettynä
rakenteena.

Pino

Pinon S operaatiot ovat seuraavat:

push(S, x): lisää x:n osoittaman alkion pinon päällimmäiseksi

pop(S): palauttaa arvonaan pinon päällimmäisen tietueen ja poistaa sen pinosta

empty(S): palauttaa True jos pinossa ei ole yhtään alkiota.

Koska datakentille ei tässä tehdä mitään, esitämme asian jatkossa ikään kuin pinossa olisi pelkät avaimet:

push(S, k): lisää avaimen k pinon päällimmäiseksi

pop(S): palauttaa arvonaan pinon päällimmäisen avaimen ja poistaa sen pinosta

empty(S): palauttaa True jos pinossa ei ole yhtään avainta.

Pinon taulukkototeutuksen idea on seuraava:

- Alkiot talletetaan taulukkoon $S[1\,..\,n]$, missä n on ennalta annettu yläraja pinon koolle.
- Kun pinossa on i alkiota, ne on talletettu osataulukkoon $S[1\,..\,i]$.
- Alkoiden lukumäärästä pidetään kirjaa muuttujassa $top[S]$, joka siis samalla osoittaa pinoon viimeksi lisätyn alkion paikan.
- Alussa $top[S] = 0$ ja pino on tyhjä.

Pino-operaatiot voidaan nyt toteuttaa seuraavasti:

push(S, k)

$top[S] \leftarrow top[S] + 1$
$S[top[S]] \leftarrow k$

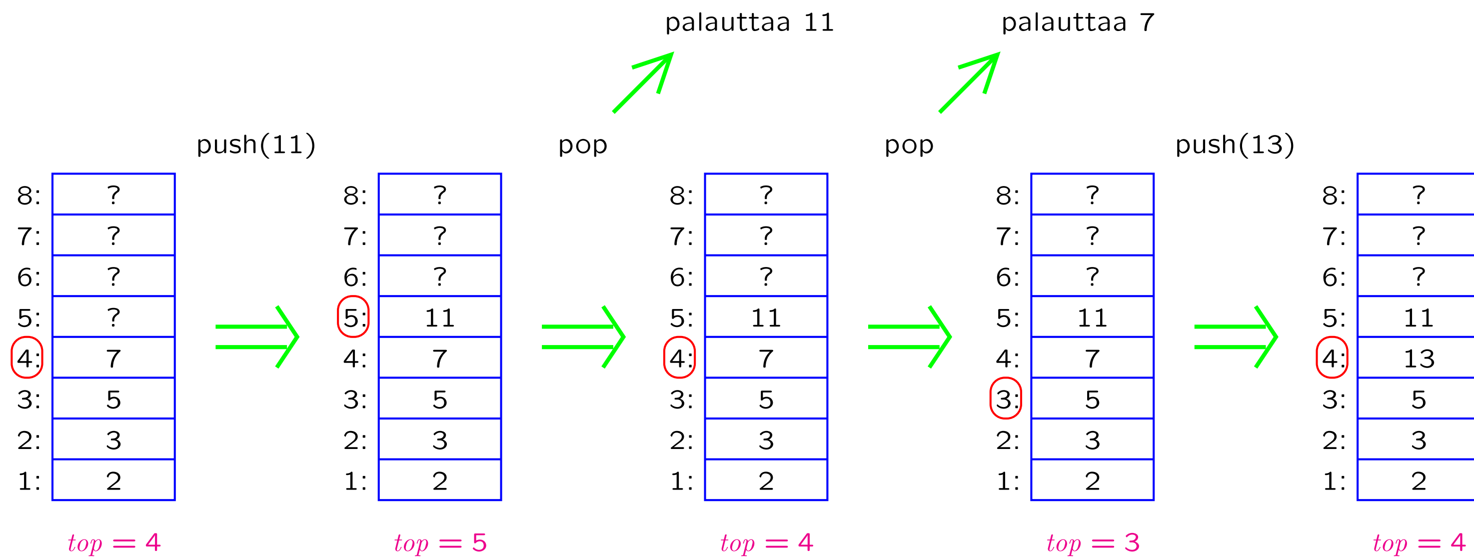
pop(S)

$top[S] \leftarrow top[S] - 1$
return $S[top[S] + 1]$

empty(S)

return $top[S] = 0$

- vanhat alkiot jäävät "roskaamaan" pinoa
- $top[S]$ kertoo, mikä osa pinosta sisältää "oikeita" alkioita
- kukin operaatio toimii selvästi vakioajassa $O(1)$



Esimerkki pino-operaatioiden taulukkototeutuksesta. Huomaa, että vaikka alkio 11 säilyy toistaiseksi talletusrakenteessa, siihen ei enää koskaan viitata.

Emme edellä ottaneet kantaa virhetilanteisiin:

Ylivuoto: push kun $top[S] = n$

Alivuoto: pop kun $top[S] = 0$.

Periaatteessa asiaan voi suhtautua kahdella tavalla:

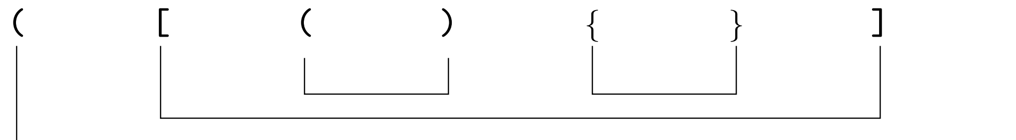
- tietorakenteen toteutuksessa tehdään aina virhetarkistukset ja aiheutetaan poikkeus, jos tietorakennetta käytetään väärin
- ei tehdä virhetarkistuksia, luotetaan että tietorakennetta käytetään oikein.

Kummallakin on etunsa. Tällä kurssilla jätämme virhetarkistukset väliin esityksen yksinkertaistamiseksi.

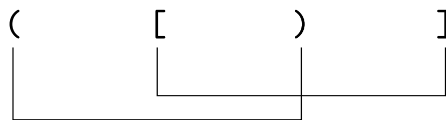
Pinon sovellus: sulutuksen tasapainoisuus

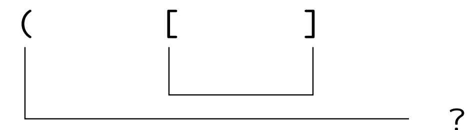
Sulutuksen tasapainoisuus tarkoittaa, että kullekin alkusululle löytyy vastaava loppusulku ja kääntäen, eivätkä erilaiset sulut mene ristiin:

Tasapainoinen sulutus: ([( ) { }])



Epätasapainoisia sulutuksia: ([ )] ja ([ ]





Sulutusongelman yleistetty versio on keskeinen ohjelmointikielten syntaksitarkistuksessa. Tämäntyyppisiä ongelmia käsitellään (jonkin verran) kurssilla *Laskennan mallit*.

Oletetaan, että funktio readnext palauttaa aina seuraavan merkin syötteestä. Ratkaistaan ongelma viemällä merkkejä pinoon:

- alkusulut painetaan pinoon
- loppusulku "syö" pinon päältä vastaavan alkusulun
- pinon alivuoto tarkoittaa liian vähän alkusulkuja
- jos pinoon jää alkioita, niin alkusulkuja on liikaa.

Saadaan seuraava algoritmi:

```
S ← tyhjä merkkipino
while syötettä riittää
      do
         c2 ← readnext
         if c2 on jokin merkeistä (, [ tai {
            then push(S, c2)
            else  if empty(S)
                     then ERROR "liian vähän alkusulkuja"
                     else
                          c1 ← pop(S)
                          if (c1 = '(' and not c2 = ')')
                          or (c1 = '[' and not c2 = ']')
                          or (c1 = '{' and not c2 = '}')
                             then ERROR "sulut ristissä"
if not empty(S)
   then ERROR "liikaa alkusulkuja"
```

Selvästi kukin syötemerkki käsitellään vakioajassa eli aikavaativuus on $O(n)$, missä n on syötemerkkien lukumäärä.

Jono

Jonon Q operaatiot ovat seuraavat:

enqueue(Q, k): lisää avaimen k jonon viimeiseksi

dequeue(Q): poistaa jonosta ensimmäisen alkion ja palauttaa sen

empty(Q): palauttaa True jos jono on tyhjä.

Toteutus on hieman hankalampi kuin pinolla, koska operaatioita kohdistuu sekä alku- että loppupäähän.

Jonon taulukkototeutuksen ymmärtämiseksi kuvitellaan ensin, että käytettävissä on rajoittamattoman suuri taulukko $Q[1\cdots]$:

- jos jonossa on ℓ alkiota, ne ovat osataulukossa $Q[i\,..\,i+\ell-1]$ jollain i
- muuttuja $head[Q]=i$ osoittaa taulukon ensimmäistä käytössä olevaa kohtaa
- muuttuja $tail[Q]=i+\ell$ osoittaa taulukon seuraavaa vapaata kohtaa
- aiemmin lisätyt alkiot ovat taulukon alussa.

Operaatiot voidaan toteuttaa seuraavasti:

enqueue(Q, k)

$Q[tail[Q]] \leftarrow k$
$tail[Q] \leftarrow tail[Q] + 1$

dequeue(Q)

$x \leftarrow Q[head[Q]]$
$head[Q] \leftarrow head[Q] + 1$
return x

empty(Q)

return $head[Q] = tail[Q]$

Jono alustetaan tyhjäksi asettamalla $head[Q] = tail[Q] = 1$.

Jos jonon koko on rajoitettu, myös talletusalueen koko voidaan rajata:

- oletetaan, että jonossa on kerralla korkeintaan n alkiota
- siis edellisessä "ratkaisussa" aina $tail - head[Q] \leq n$.
- **Havainto:** kun $tail[Q]$ saa arvon $n + 2$, niin oletuksen mukaan $head[Q] \geq 2$
- siis paikka $Q[1]$ voidaan ottaa uusiokäyttöön

Yleisemmin sen sijaan, että talletettaisiin jotain äärettömän Q-taulukon kohtaan i missä $i \geq n + 2$, talletetaankin se kohtaan $i \bmod (n + 1)$, missä mod tarkoittaa jakojäännöstä.

Koska taulukon käytössä olevan osan pituus on alle $n + 1$, mitkään kaksi alkiota eivät mene päällekkäin.

Korkeintaan n-alkioiselle jonolle saadaan siis seuraava toteutus:

enqueue(Q, k)

$Q[tail[Q]] \leftarrow k$
$tail[Q] \leftarrow (tail[Q] + 1) \bmod (n + 1)$

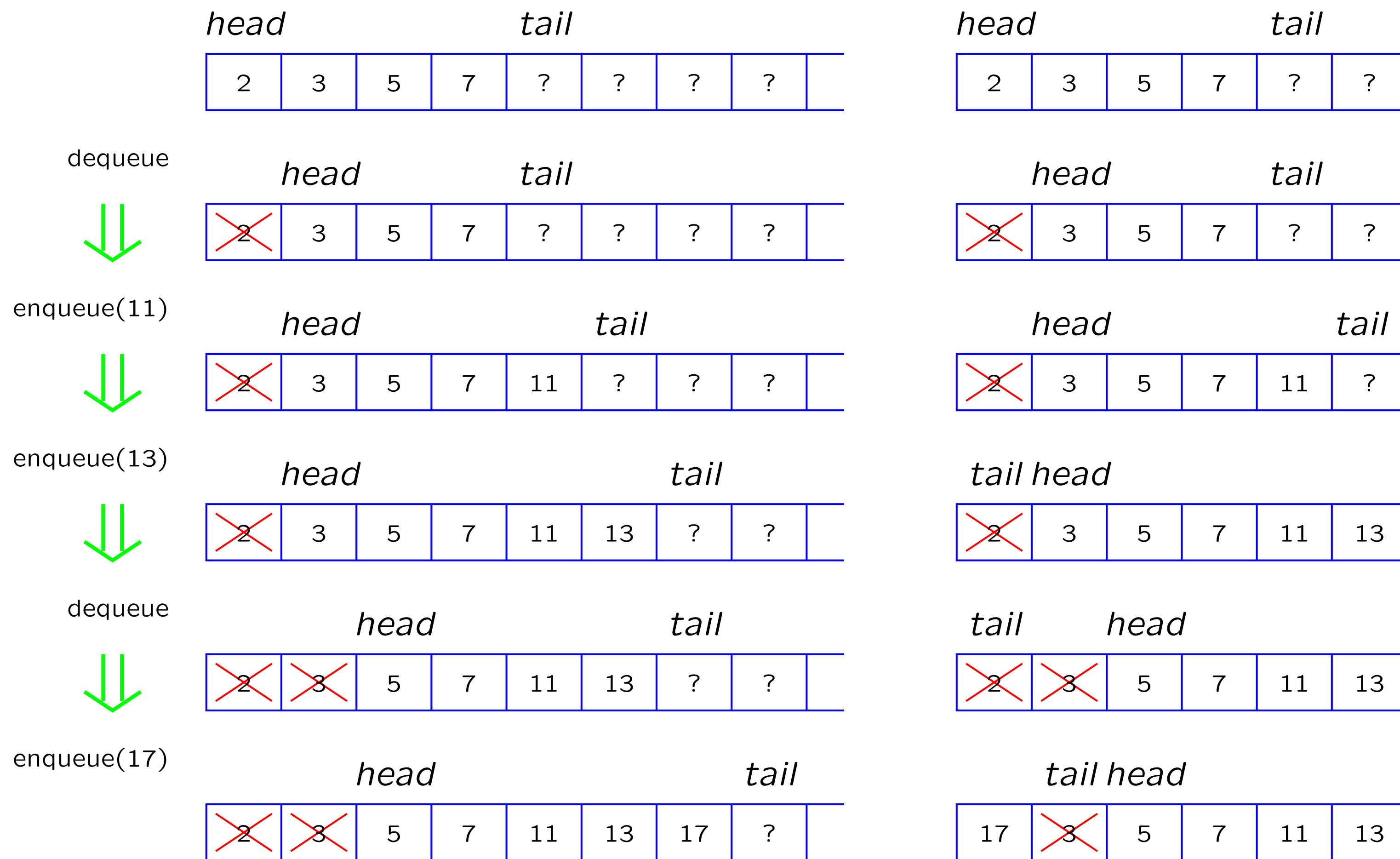
dequeue(Q)

$x \leftarrow Q[head[Q]]$
$head[Q] \leftarrow (head[Q] + 1) \bmod (n + 1)$
return x

empty(Q)

return $head[Q] = tail[Q]$

Kuten pinolla, kukin operaatio toimii vakioajassa.



Esimerkki jono-operaatioista rajoittamattomassa ja rajoitetussa taulukossa (jonon maksimipituus $n = 5$, taulukossa 6 paikkaa). Rastilla on merkitty talletusrakenteeseen jääneet alkiot, jotka eivät enää ole jonossa.

Linkitetyt rakenteet

Edellä esitetyssä pinon ja jonon toteutuksessa taulukosta löytyy millä tahansa i vakioajassa rakenteen alkio numero i.

Tällainen yleinen indeksointi ei kuitenkaan ole tarpeen, koska rakenteita käsitellään vain osoittimien top, $head$ ja $tail$ kautta ja nämä osoittimet muuttuvat kerralla vain yhdellä.

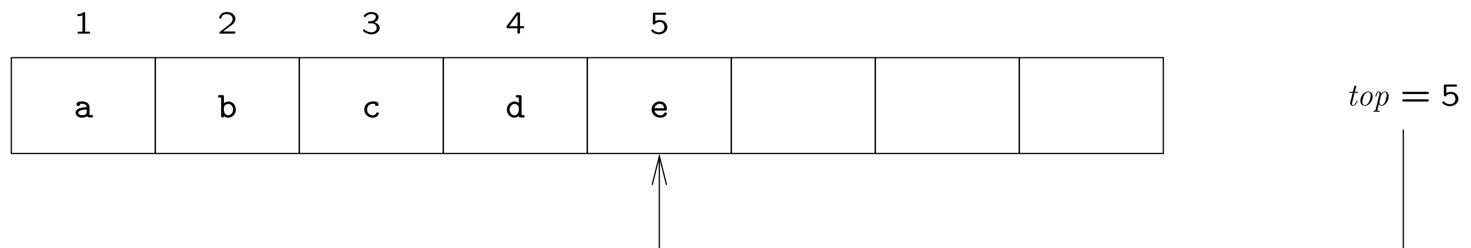
Esim. pinon tapauksessa riittäisi, että

- pino "tietää" mistä sen päällimmäinen alkio löytyy
- pinon jokainen alkio tietää, mistä sen alla oleva alkio löytyy.

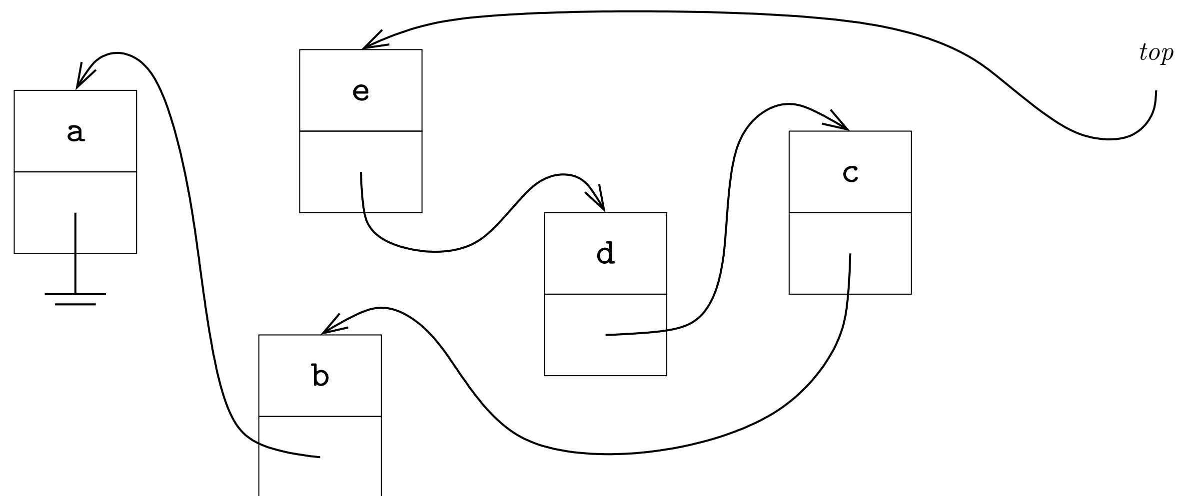
Tätä hyödyntämällä voidaan

- sallia mielivaltaisen alkion poistaminen tai lisääminen ilman koko rakenteen uudelleenorganisointia
- käyttää muistia tasan tarvittava määrä.

Pino taulukkoon talletettuna:



Pino linkitettynä rakenteena:



Käytännössä tämä tarkoittaa, että pinon alkiot talletetaan tyyppiä pinosolmu oleviin tietueisiin, joissa on kentät

key: alkion avain

$data$: alkioon liittyvä muu data (tai linkki satelliittitietueesee)

$next$: viite pinossa tämän alkion alla olevan alkion pinosolmu-tietueeseen.

Jätämme taas $data$-kentän huomiotta.

Pinon pohjimmaisen alkion $next$ saa arvon Nil (osoitin joka osoittaa ei-minnekään, kuvassa $-||$).

Linkitetyn pinon operaatiot voidaan nyt toteuttaa seuraavasti:

push(S, k)

$x \leftarrow$ new $pinosolmu$
$key[x] \leftarrow k$
$next[x] \leftarrow top[S]$
$top[S] \leftarrow x$

pop(S)

$x \leftarrow top[S]$
$top[S] \leftarrow next[x]$
return $key[x]$

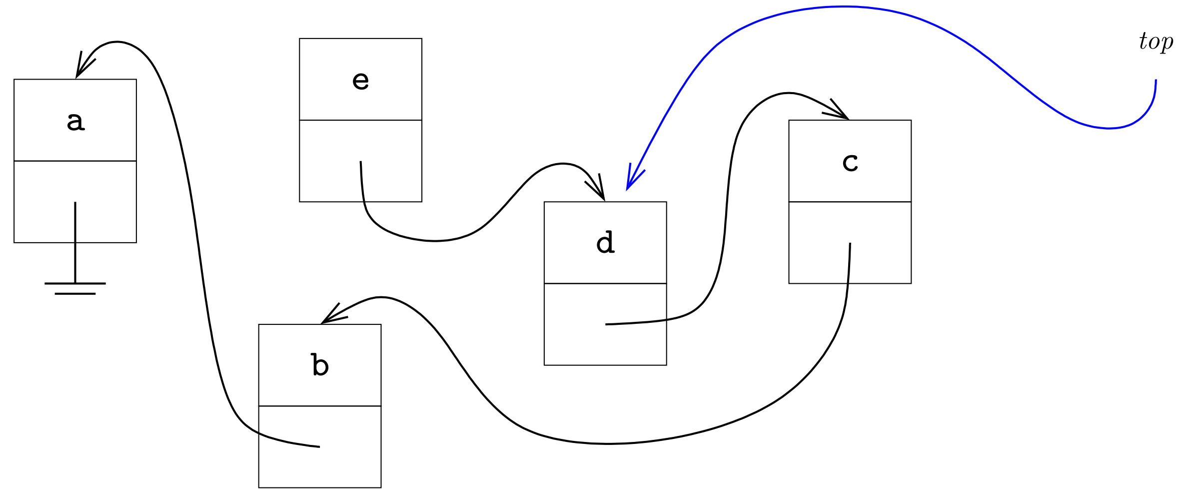
empty(S)

return $top[S] =$ Nil

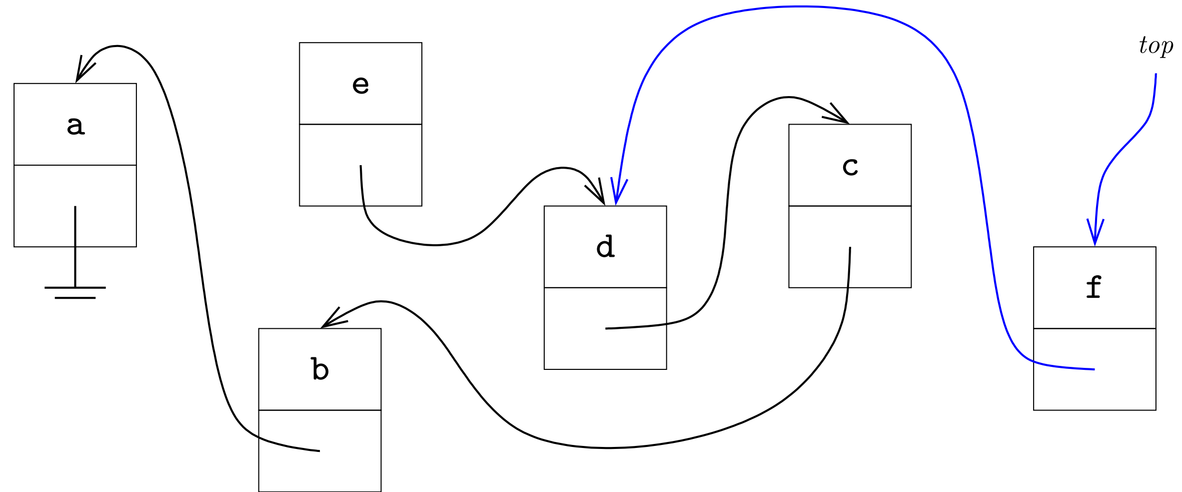
Kutsu "new $pinosolmu$" varaa tilaa tyyppiä $pinosolmu$ olevalle tietueelle ja palauttaa osoittimen siihen. Toisaalta pop-operaatio jättää roskia eli solmuja, joihin ei ole viitettä. Näiden käsittely riippuu ohjelmointikielestä.

Pino alustetaan tyhjäksi asettamalla $S \leftarrow$ Nil.

pop(S):



push(S, f):



Esimerkki linkitettyjen rakenteiden toteuttamisesta Javalla:

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

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

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

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

Lista

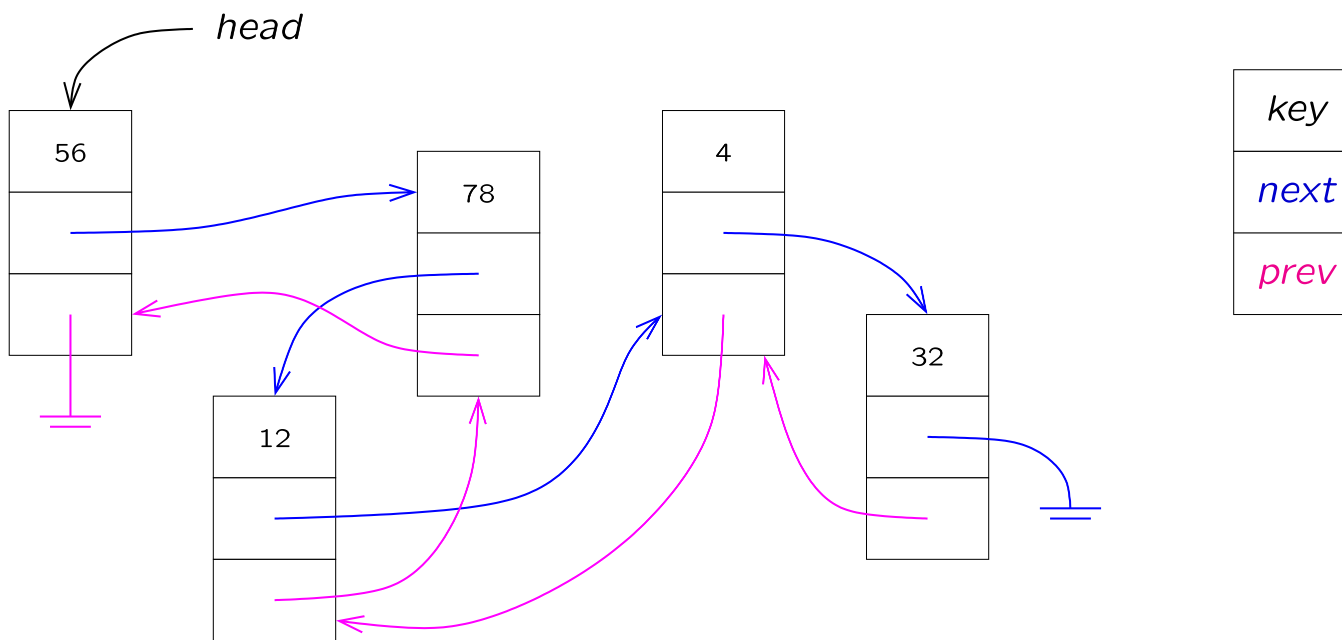
Tietotyyppi joukko voidaan toteuttaa linkitettynä listana, josta on eri variaatioita:

- yhteen tai kahteen suuntaan linkitetty
- järjestetty tai ei
- rengas tai ei
- tunnusssolmullinen tai ei.

Näiden välisistä eroista tärkeimmät (operaatioiden kertaluokkiin vaikuttavat) ovat

- järjestetty lista toteuttaa operaatiot min, max, succ ja pred vakioajassa, järjestämätön lineaarisessa
- delete-operaation toteuttaminen vakioajassa vaatii kahteen suuntaan linkitetyn listan tai pientä temppuilua.

Mikään näistä ei toteuta search-operaatiota (läheskään) yhtä hyvin kuin myöhemmin esiteltävät tasapainoiset hakupuut ja hajautusrakenteet.



Avaimet 56, 78, 12, 4 ja 32 linkitetyssä listassa

Listan alkio talletetaan tyyppiä listasolmu olevaan tietueeseen, jossa on kentät

key: alkion avain

$data$: alkioon liittyvä mahdollinen muu tieto

$next$: viite seuraavaan solmuun listassa

$prev$: viite edelliseen solmuun listassa

Koko listaan L liittyvä attribuutti $head[L]$ osoittaa listan ensimmäistä solmua.

- $data$-kenttä voi olla mitä tahansa, mutta ei vaikuta toteutukseen; jätämme sen jatkossa(kin) huomiotta
- yhteen suuntaan linkitetyssä listassa ei ole $prev$-kenttää
- listan viimeisen (ensimmäisen) alkion $next$ ($prev$) on Nil
- "seuraava" ja "edellinen" viittaa listan talletusjärjestykseen, ei avainten järjestykseen (vrt. succ ja pred)

Listasta haetaan avainta tarkastamalla alusta lähtien solmuja $next$-osoittimia seuraten. Jos avainta ei löydy, palautetaan Nil.

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

Aikavaativuus kun listassa n alkiota:

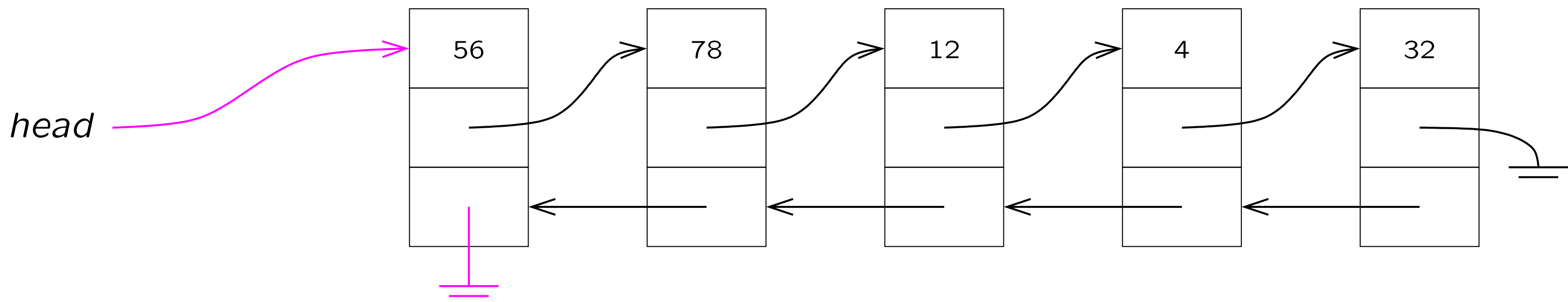
- alustukset $O(1)$, silmukan yksi iteraatio $O(1)$
- silmukan toistoja pahimmillaan n

$\Rightarrow$ operaation aikavaativuus $\Theta(n)$

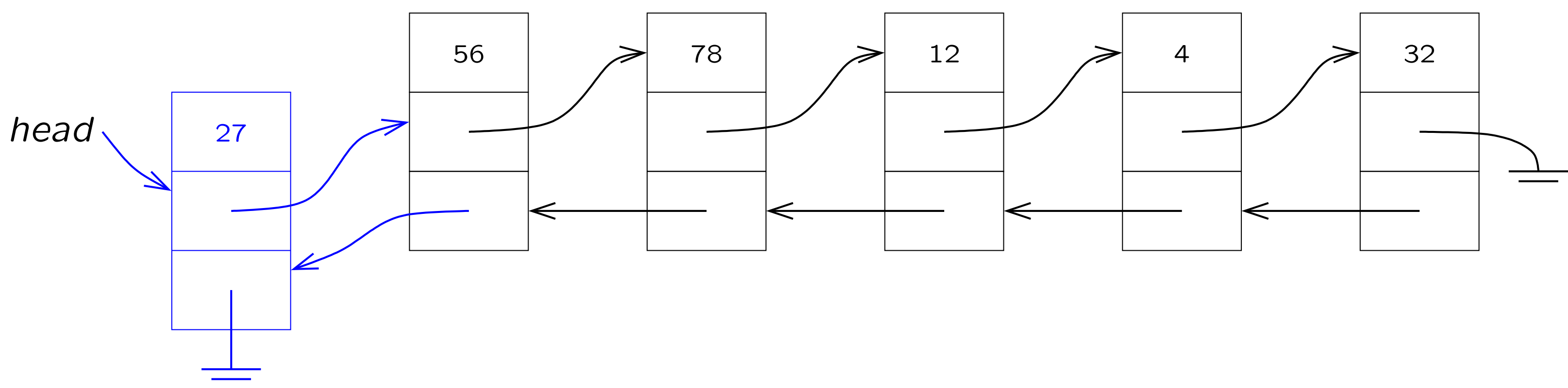
Lisääminen tehdään listan alkuun, jolloin se selvästi sujuu vakioajassa:

insert(L, k)

$x \leftarrow$ new *listasolmu*
$key[x] \leftarrow k$
$next[x] \leftarrow head[L]$
$prev[x] \leftarrow$ Nil
if $head[L] \neq$ Nil
    then $prev[head[L]] \leftarrow x$
$head[L] \leftarrow x$



insert(L, 27)

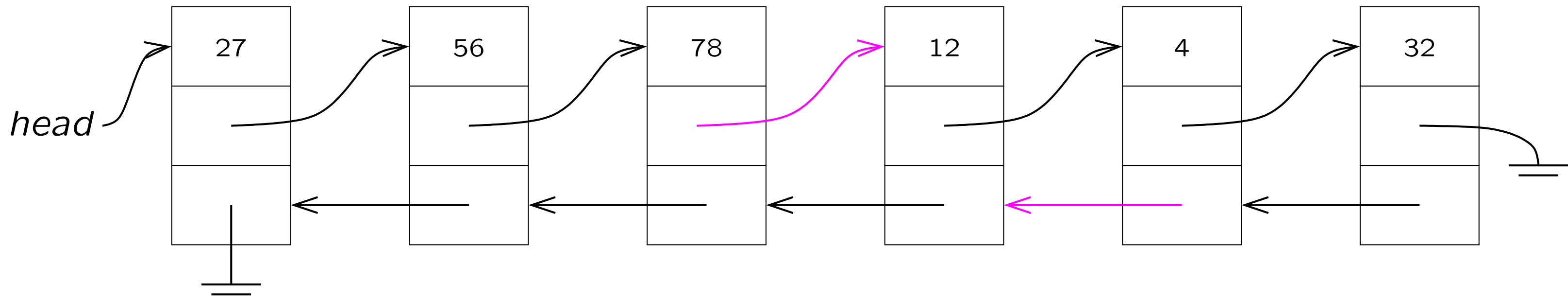


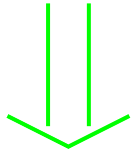
Myös poisto sujuu vakioajassa, koska argumenttina oletetaan olevan suoraan osoitin solmuun:

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

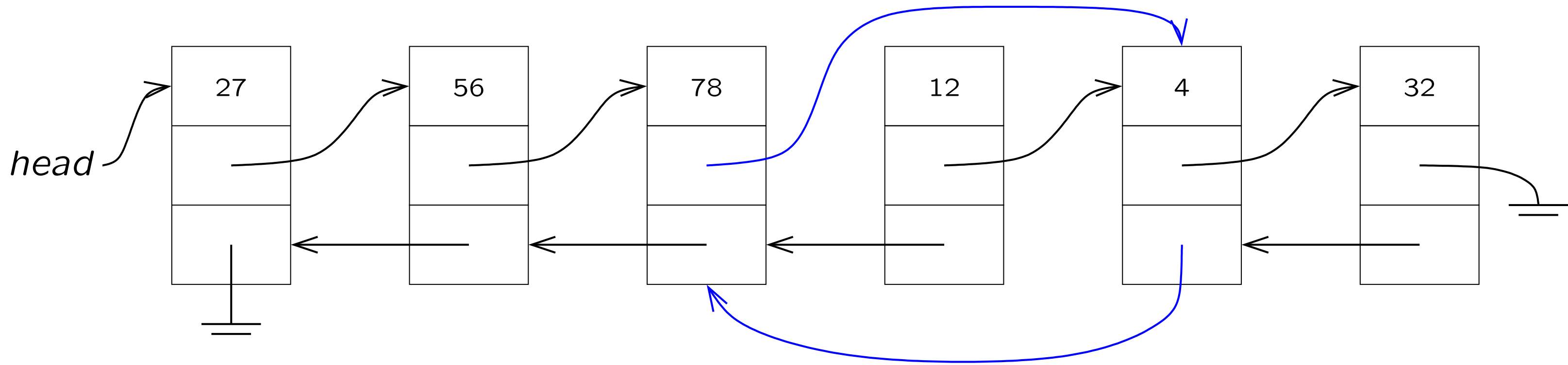
Jos halutaan poistaa avain k, sen osoite pitää ensin hakea search-operaatiolla:

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





delete(L, search(L, 12))



Pienimmän avaimen löytäminen edellyttää koko listan läpikäyntiä:

min(L)

```
x ← head[L]
p ← head[L]
while p ≠ Nil do
        if key[p] < key[x]
           then x ← p
        p ← next[p]
return x
```

Koko lista käydään aina läpi, joten aikaa kuluu $\Theta(n)$.

Operaatio max toteutetaan vastaavasti.

Myös seuraavaksi suurimman avaimen löytäminen edellyttää koko listan läpikäyntiä:

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

Koko lista käydään aina läpi, joten aikaa kuluu $\Theta(n)$.

Operaatio pred toteutetaan vastaavasti.

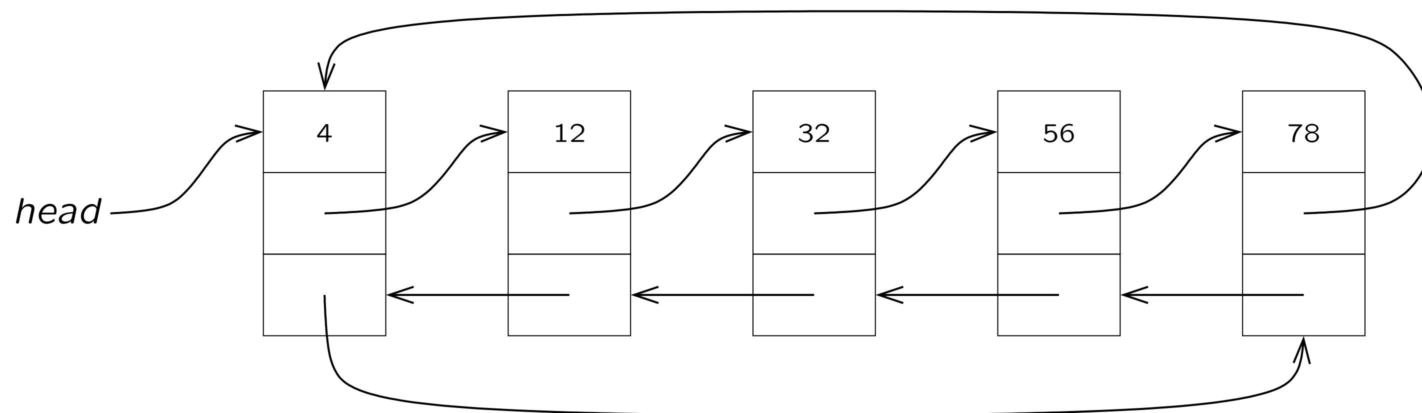
Järjestetty rengaslista

Avainten järjestykseen liittyvät operaatiot min, max, pred ja succ saadaan toimimaan vakioajassa tallettamalla listan alkiot avaimen mukaisessa järjestyksessä.

Vastapainona lisäysoperaatio vaikeutuu, koska alkiolle pitää etsiä sen oikea paikka.

Koska max joka tapauksessa tarvitsee osoittimen myös listan loppuun, esitämme järjestetystä listasta rengasversion:

- viimeisen alkion $next$ osoittaa listan ensimmäistä alkiota
- ensimmäisen alkion $prev$ osoittaa listan viimeistä alkiota.



Järjestetty rengaslista

Järjestykseen liittyvät kyselyt ovat nyt hyvin helppoja:

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

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

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

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

Poisto-operaatio on oleellisesti kuten ennen:

```
delete(L, x)
    y ← prev[x]
    z ← next[x]
    if x = z            ▷ Listassa vain yksi alkio
       then head[L] ← Nil
       else
            next[y] ← z
            prev[z] ← y
            if x = head[L]
               then head[L] ← z
```

Etsimistä voidaan hieman tehostaa lopettamalla, kun tulee vastaan liian suuri avain:

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

Pahimmassa tapauksessa koko lista joudutaan kuitenkin käymään läpi, joten aikavaativuus on $\Theta(n)$.

Lisääminen on sotkuista lähinnä monien erikoistapausten takia:

```
insert(L, k)
    x ← new listasolmu;    key[x] ← k;    p ← head[L]
    if p = Nil               ▷ Lisäys tyhjään listaan
       then next[x] ← prev[x] ← head[L] ← x
    elseif k < key[p]
       then                  ▷ Lisäys listan alkuun
            next[x] ← p
            prev[x] ← prev[p]
            next[prev[x]] ← prev[next[x]] ← head[L] ← x
    else
            while next[p] ≠ head[L] and key[p] < k do    p ← next[p]
            if k > key[p]
              then   ▷ Lisäys listan loppuun
                     next[x] ← head[L];    prev[x] ← p
                     prev[next[x]] ← next[prev[x]] ← x
              else   ▷ Lisäys listan keskelle
                     next[x] ← p;    prev[x] ← prev[p]
                     prev[next[x]] ← next[prev[x]] ← x
```

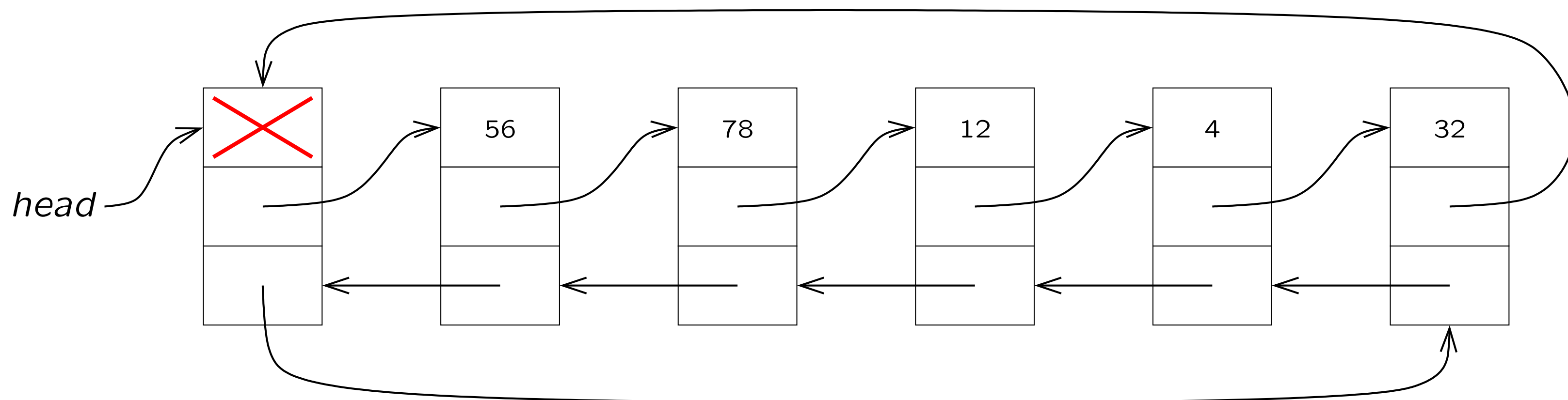
Tunnussolmullinen lista

Liitetään listan alkuun ylimääräinen tunnussolmu, johon ei liity mitään avainta:

- merkitään listan L tunnussolmua $nil[L]$
- $next[nil[L]]$ osoittaa listan ensimmäistä varsinaista solmua
- $prev[nil[L]]$ osoittaa listan viimeistä solmua
- kuten yleensä, $prev[next[nil[L]]] = next[prev[nil[L]]] = nil[L]$
- tyhjällä listalla $next[nil[L]] = prev[nil[L]] = nil[L]$

Tunnussolmu vähentää hankalien erikoistapausten määrää.

Esitetään tästä vain järjestämätön versio.



Tunnussolmullinen järjestämätön rengaslista

Etsintä sujuu kuten ennenkin:

search(L, k)

```
x ← next[nil[L]]
while x ≠ nil[L] and k ≠ key[x] do
        x ← next[x]
if x = nil[L]
    then return Nil
    else  return x
```

Poisto ja lisäyskin ovat hyvin suoraviivaisia:

delete(L, x)

$next[prev[x]] \leftarrow next[x]$
$prev[next[x]] \leftarrow prev[x]$

insert(L, k)

$x \leftarrow$ new listasolmu
$key[x] \leftarrow k$
$next[x] \leftarrow next[nil[L]]$
$prev[x] \leftarrow nil[L]$
$prev[next[x]] \leftarrow next[prev[x]] \leftarrow x$

3. Hakupuut

Hakupuu on listaa tehokkaampi dynaamisen joukon toteutus. Erityisesti suurilla tietomäärillä hakupuu kannattaa tasapainottaa, jolloin päivitysoperaatioista tulee hankalampia toteuttaa mutta etsiminen nopeutuu huomattavasti.

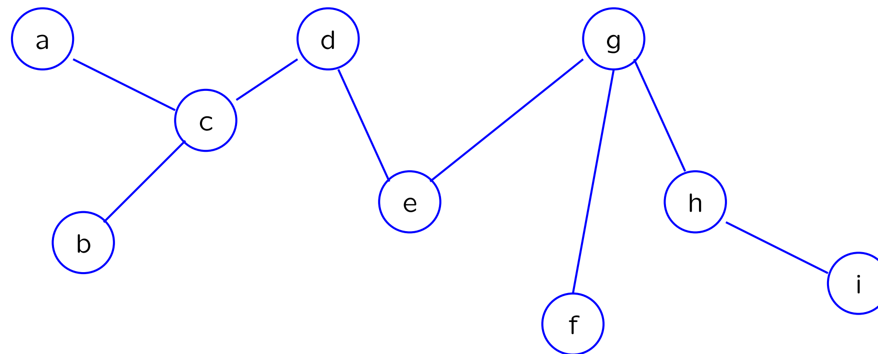
B-puu on hakupuun laji, joka sopii mm. tietokantasovelluksiin, joissa rakenne on talletettu kiintolevylle eikä keskusmuistiin.

Puita yleensä käytetään paitsi tietorakenteena myös abstraktina mallina erilaisille tietojenkäsittelytehtäville:

- ongelmanratkaisun mallintaminen peruuttavana etsintänä (*Tekoäly*)
- tietokoneohjelman jäsennyspuu (*Laskennan mallit*)
- shakin tms. pelipuu.

Vapaa puu (free tree)

Vapaa puu on erikoistapaus myöhemmin esiteltävästä verkosta (graafi, engl. graph). Se koostuu solmuista (vertex, node) ja kahta solmua yhdistävistä kaarista (edge). Puussa minkä tahansa kahden solmun välillä on tasan yksi kaarista muodostuva reitti eli polku. (Sahaamista edestakaisin ei lasketa eri reitiksi.)

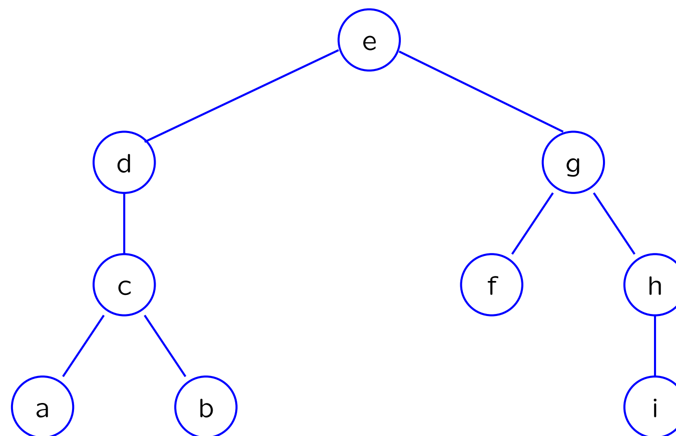


Myöhemmin esiteltävää verkkoterminologiaa käyttäen vapaa puu on yhtenäinen syklitön suuntaamaton verkko.

Juurellinen puu (rooted tree)

Jatkossa (jos ei muuta mainita) ”puu” tarkoittaa juurellista puuta, jossa yksi solmu on asetettu erikoisasemaan juureksi. Tämä asettaa samalla muutkin solmut eri asemaan sen perusteella, kuinka kaukana ne ovat juuresta.

Tietojenkäsittelytieteessä puu piirretään juuri ylöspäin. Alla siis juurena on solmu e.



Jos juuresta solmuun x menevä polku kulkee solmun y kautta, niin

- solmu x on solmun y seuraaja eli jälkeläinen
- solmu y on solmun x edeltäjä eli esi-isä.

Jos lisäksi $y \neq x$, niin solmu x on solmun y aito seuraaja solmu y on solmun x aito edeltäjä. Siis jokainen solmu on sekä oma esi-isänsä että jälkeläisensä.

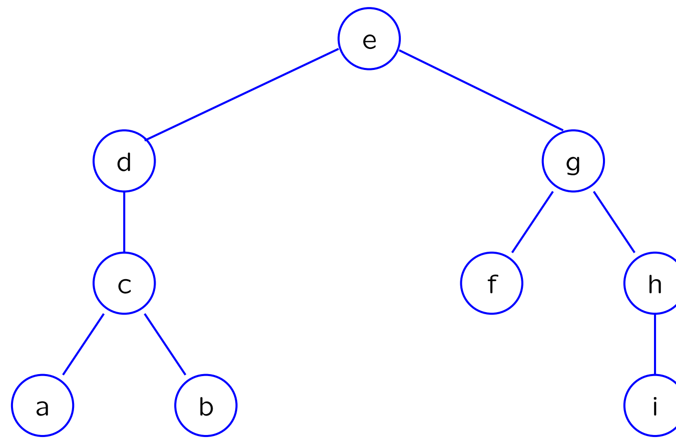
Jos lisäksi solmujen x ja y välillä ei ole muita solmuja, niin

- solmu x on solmun y lapsi
- solmu y on solmun x vanhempi.

Käytämme muutenkin sukupuista periytyvää terminologiaa. Esim. solmut ovat sisaruksia, jos niillä on sama vanhempi.

Solmu on sisäsolmu, jos sillä on ainakin yksi lapsi. Muut kuin sisäsolmut ovat lehtiä.

Esimerkkejä solmuterminologiasta



- solmun c esi-isät ovat c, d ja e
- solmun g jälkeläiset ovat g, f, h ja i
- solmut d ja g ovat sisarukset
- solmu g on solmun c setä, ja e on solmun c isoisä
- puun sisäsolmut ovat e, d, g, c ja h
- puun lehdet ovat a, b, f ja i.

Määrittelemme täsmällisemmin, että polku on sellainen jono solmuja $x_1, \ldots, x_k$, että solmujen x_i ja x_{i+1} välillä on kaari kaikilla $i \leq i \leq k-1$. Polun pituus on sillä olevien kaarten lukumäärä eli tässä $k-1$. Erityisesti jokaisesta solmusta itseensä on polku, jonka pituus on 0.

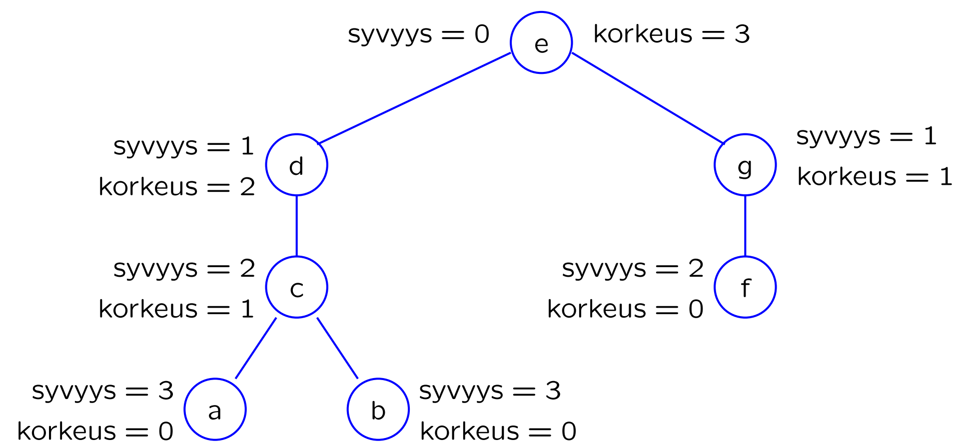
Solmun taso eli syvyys puussa on siihen juuresta johtavan polun pituus. Siis juuren syvyys on 0.

Solmun korkeus on pisimmän solmusta sen jälkeläiseen johtavan polun pituus. Siis lehden korkeus on 0.

Koko puun korkeus on sen juuren korkeus, eli syvimmän lehden syvyys.

Havainto: koska juurta lukuunottamatta jokaisella solmulla on tasan yksi vanhempi,

$$\text{kaarten lkm.} = \text{solmujen lkm.} - 1.$$



Puun korkeus on 3.

Puu rekursiivisena rakenteena

Juurelliselle puulle voidaan verkkoformalismin sijaan antaa myös rekursiivinen määritelmä:

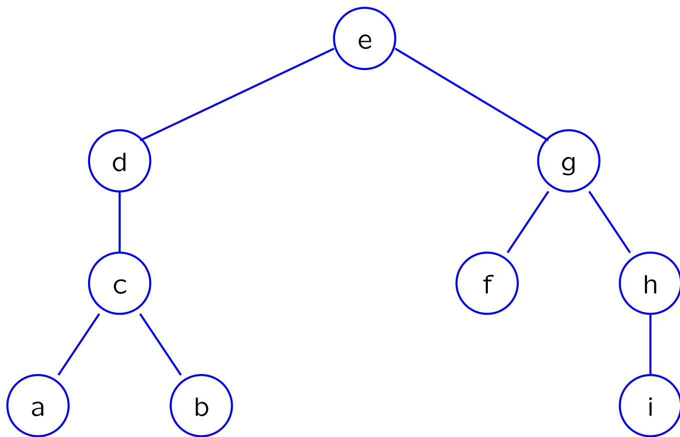
- on olemassa tyhjä puu, jota merkitään Nil
- jos on olemassa solmu r ja puut $T_1, \ldots, T_k$, missä $k \geq 0$, niin voidaan muodostaa puu, jonka
  - juurena on r
  - alipuina ovat $T_1, \ldots, T_k$.

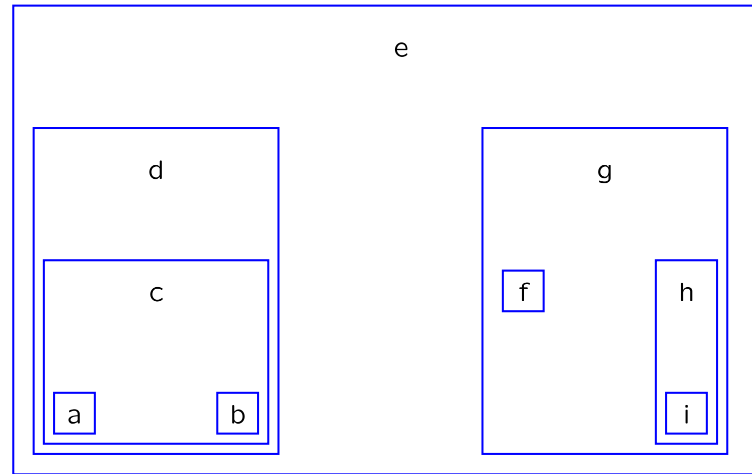
Jos esim. ajatellaan, että jokaiseen solmuun x liittyy kokonaisluku $val[x]$, niin koko puussa T olevien arvojen summa $Sum[T]$ voidaan nyt määritellä rekursiivisesti

- $Sum[\text{Nil}] = 0$
- jos puulla T on juuri r ja alipuut $T_1, \ldots, T_k$, niin

$$Sum[T] = val[r] + \sum_{i=1}^{k} Sum[T_i].$$

Tästä saadaan suoraan rekursiivinen algoritmi summan laskemiseksi.





Puu verkkona ja rekursiivisena rakenteena

Binääripuu

Erotukseksi tavallisesta juurellisesta puusta

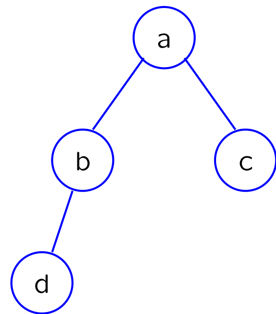
- solmulla voi olla korkeintaan kaksi lasta
- lapset on "nimetty": solmulla on vasen lapsi ja oikea lapsi, joista toinen tai molemmat voi puuttua.

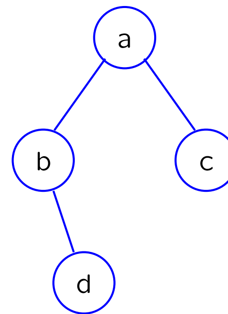
Merkinnät $\mathit{left}[x]$ ja $\mathit{right}[x]$ tarkoittavat viitettä solmun x vasempaan ja oikeaan lapseen. Lapsen puuttumista merkitään arvolla Nil.

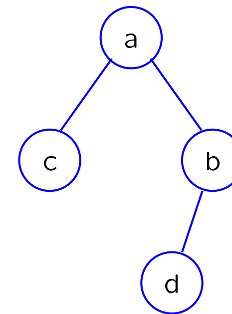
Solmuun $\mathit{left}[x]$ juurtuvaa puuta sanotaan solmun x vasemmaksi alipuuksi. Vastaavasti määritellään oikea alipuu. Jos lasta ei ole, vastaava alipuu on tyhjä. Tyhjästä binääripuusta käytetään (taas) merkitään Nil.

Vasen ja oikea lapsi kuvataan piirroksessa ilmeisellä tavalla.

Huom. seuraavat binääripuut ovat *eri binääripuita*, vaikka vastaavat juurelliset puut ovat samat:







Lause 3.1: Binääripuun tasolla (= syvyydellä) i on korkeintaan 2^i solmua.

Todistus: Induktio tason i suhteen.

Jos $i = 0$, niin tasolla i on vain puun juuri, ja $2^0 = 1$.

Olkoon sitten $i \geq 1$. Tehdään induktio-oletus, että missä tahansa binääripuussa tasolla $i - 1$ on korkeintaan 2^{i-1} solmua. Nyt missä tahansa binääripuussa tasolla i olevat solmut voidaan jakaa kahteen luokkaan:

- vasemman alipuun tasolla $i - 1$ olevat solmut
- oikean alipuun tasolla $i - 1$ olevat solmut.

Kumpiakin on induktio-oletuksen nojalla korkeintaan 2^{i-1} kappaletta, joten kaikkiaan tasolla i on solmuja korkeintaan $2^{i-1} + 2^{i-1} = 2^i$ kappaletta. □

Lause 3.2: Jos binääripuun korkeus on k, niin siinä on korkeintaan $2^{k+1} - 1$ solmua.

Todistus 1: Induktio puun korkeuden k suhteen.

Jos korkeus on $k = 0$, niin solmuja on korkeintaan $1 = 2^{0+1} - 1$.

Oletetaan nyt, että $k \geq 1$ ja korkeutta $k - 1$ olevassa binääripuussa on korkeintaan $2^{(k-1)+1} - 1 = 2^k - 1$ solmua. Siis korkeutta k olevassa binääripuussa on korkeintaan

- yksi solmu juuressa
- $2^k - 1$ solmua vasemmassa alipuussa ja
- $2^k - 1$ solmua oikeassa alipuussa

eli yhteensä $1 + 2 \cdot (2^k - 1) = 2^{k+1} - 1$ solmua. $\square$

Todistus 2: Korkeutta k olevassa puussa on tasot $0, 1, 2, \ldots, k$. Edellisen lauseen nojalla siinä on

- korkeintaan $2^0 = 1$ solmu tasolla 0
- korkeintaan $2^1 = 2$ solmua tasolla 1
- korkeintaan $2^2 = 4$ solmua tasolla 2
- ...
- korkeintaan 2^k solmua tasolla k

eli yhteensä korkeintaan

$$1 + 2 + 4 + \ldots + 2^k = \frac{1 - 2^{k+1}}{1 - 2} = 2^{k+1} - 1$$

solmua, missä on käytetty geometrisen sarjan summakaavaa

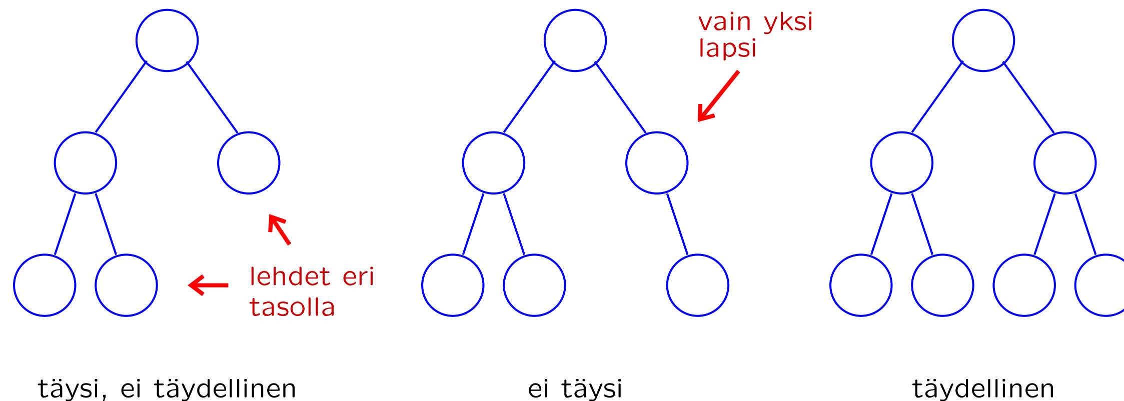
$$\sum_{i=0}^{k} q^i = \frac{1 - q^{k+1}}{1 - q}.$$

□

Binääripuu on

täysi jos jokaisella solmulla on 0 tai 2 lasta

täydellinen jos se on täysi ja kaikki lehdet ovat samalla tasolla



Siis täydellisessä binääripuussa on kaikki solmut, jotka sen korkuiseen mahtuvat.

Edellisistä todistuksista nähdään helposti, että korkeutta k olevassa täydellisessä binääripuussa on

- tasan $2^{k+1} - 1$ solmua
- tasan 2^k lehteä.

Lause 3.3: Jos binääripuussa on n solmua, niin sen korkeus on ainakin $\lceil \log_2(n+1) \rceil - 1$.

Todistus: Olkoon n-solmuisen binääripuun korkeus h.

Edellisen lauseen mukaan $n \leq 2^{h+1} - 1$.

Ratkaisemalla tämä saadaan $h \geq \log_2(n+1) - 1$.

Lisäksi tietysti h on kokonaisluku. $\square$

Yhtäsuuruus $h = \log_2(n+1) - 1$ pätee, jos ja vain jos binääripuu on täydellinen.

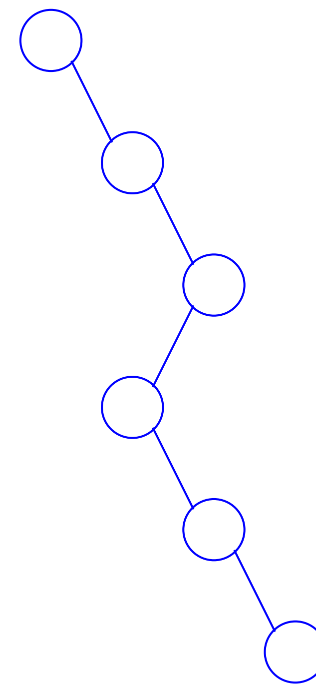
Toisaalta n solmusta ei mitenkään saa aikaan pidempää lehteen vievää polkua kuin $n-1$, joten pätee

Lause 3.4: Jos binääripuussa on n solmua, niin sen korkeus on korkeintaan $n-1$. □

Siis n-solmuisen binääripuun korkeus h on välillä

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

ja kumpikin raja voidaan saavuttaa.



solmuja n

korkeus $= n-1$

Binääripuualgoritmien perusrakenne

Monet binääripuualgoritmit syntyvät luonnostaan käyttämällä rekursiota puun rekursiivista rakennetta noudattaen.

Jos esim. binääripuussa on avaimina kokonaislukuja, niin avainten summa voidaan laskea kutsulla Sum(x), missä x on puun juuri:

```
Sum(x)
    if x = Nil
        then return 0          ▷ tyhjä puu
        else
            x ← Sum(left[x])
            y ← Sum(right[x])
            return key[x] + x + y
```

(Muuttujia x ja y on käytetty havainnollistamaan algoritmin rakennetta, tietysti voitaisiin yksinkertaisesti palauttaa
$key[x] + \text{Sum}(left[x]) + \text{Sum}(left[x])$.)

Binääripuun tallettamisessa on usein kätevää (mutta ei välttämätöntä) liittää kuhunkin solmuun myös viite sen vanhempaan.

Siis solmussa x on seuraavat kentät:

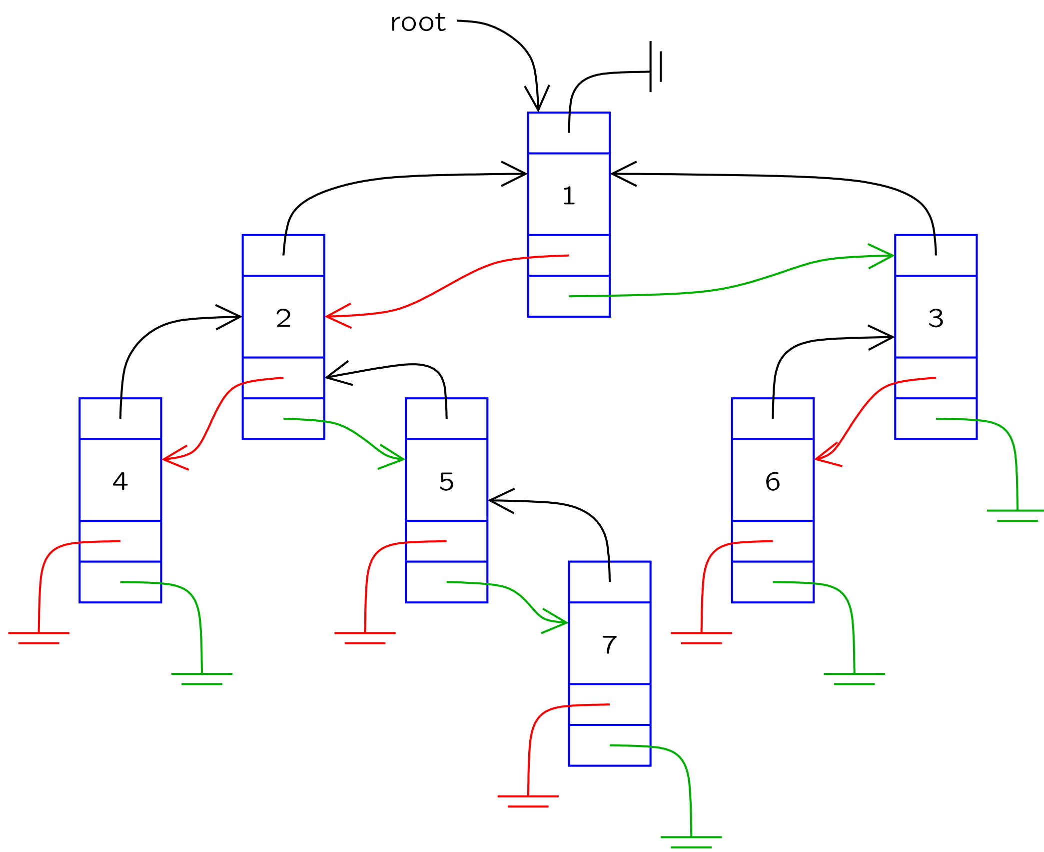
> $key[x]$: solmuun talletettu avain
> $data[x]$: solmuun talletettu muu tieto
> $left[x]$: osoitin vasemman alipuun juureen
> $right[x]$: osoitin oikean alipuun juureen
> $p[x]$: osoitin vanhempaan.

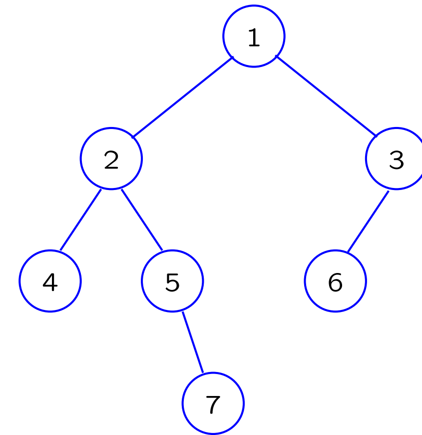
Emme taas jatkossa kiinnitä huomiota data-kenttään.

Kentät $left[x]$ ja $right[x]$ saavat arvon Nil, jos vastaava alipuu on tyhjä.

Kenttä $p[x]$ saa arvon Nil, jos x on koko puun juuri.

Puun T juureen osoittaa $root[T]$.





Binääripuun talletusrakenne

Monet binääripuualgoritmit perustuvat puun solmujen läpikäyntiin jossain nimenomaisessa järjestyksessä.

Seuraava algoritmi tulostaa solmujen avaimet sisäjärjestyksessä:

Inorder-Tree-Walk(x)

```
if x ≠ Nil
   then
        Inorder-Tree-Walk(left[x])
        print key[x]
        Inorder-Tree-Walk(right[x])
```

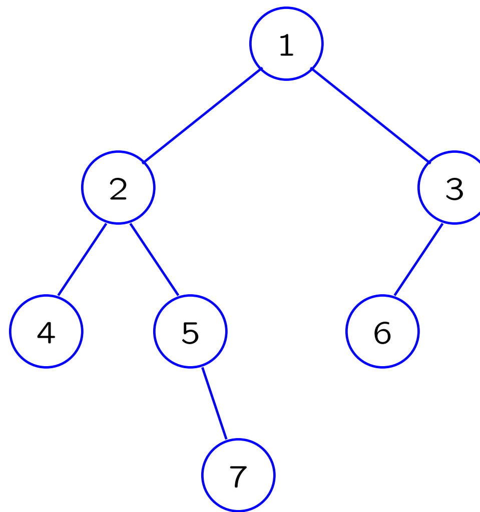
Tulostamisen sijaan voidaan tietysti tehdä kullekin solmulle jokin mutkikkaampikin operaatio.

Solmut voidaan tulostaa myös esijärjestyksessä

```
Preorder-Tree-Walk(x)
    if x ≠ Nil
       then
            print key[x]
            Preorder-Tree-Walk(left[x])
            Preorder-Tree-Walk(right[x])
```

tai jälkijärjestyksessä

```
Postorder-Tree-Walk(x)
    if x ≠ Nil
       then
            Postorder-Tree-Walk(left[x])
            Postorder-Tree-Walk(right[x])
            print key[x]
```



Binääripuun solmut

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

Olettaen, että yksittäiseen solmuun kohdistuva operaatio (edellä print) sujuu vakioajassa, kaikkien kolmen läpikäyntialgoritmin resurssivaativuudesta voidaan todeta seuraavaa:

Aikavaativuus on $\Theta(n)$, missä n on solmujen lukumäärä. Jokaista solmua kohti nimittäin tehdään kerran seuraavat toimet:

1. valmistellaan rekursiivinen kutsu vasemmalla alipuulla
2. valmistellaan rekursiivinen kutsu oikealla alipuulla
3. käsitellään solmu itse.

Tilavaativuus on $\Theta(h)$, missä h on puun korkeus. Nimittäin tilavaativuus on verrannollinen rekursiopinon maksimisyvyyteen, ja pinossa on kerrallaan aina yhtä puun haaraa vastaavat kutsut.

Läpikäynnit on mahdollista toteuttaa myös ilman rekursiota vakiotilassa olettaen, että kahden osoittimen yhtäsuuruusvertailu on mahdollista (HT).

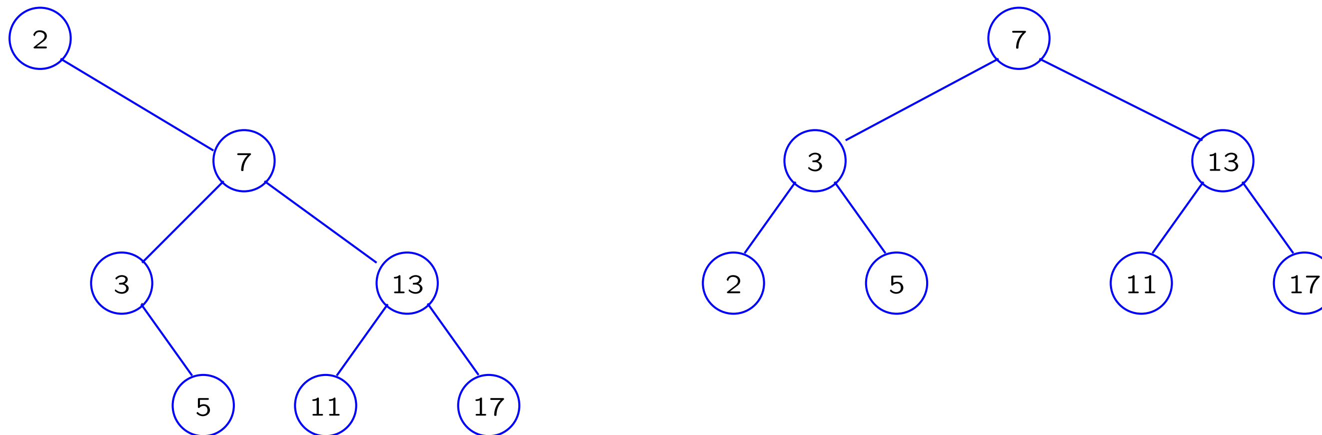
Binäärihakupuu

Oletamme nyt, että puuhun talletetut avaimet ovat kokonaislukuja tai niille muuten on määritelty järjestysrelaatio $\leq$.

Binäärihakupuu on binääripuu, jossa avaimet toteuttavat binäärihakupuuehdon

jos solmu x on solmun y vasemmassa alipuussa niin $key[x] \leq key[y]$ ja
jos solmu z on solmun y oikeassa alipuussa niin $key[y] \leq key[z]$.

Alla on kaksi erilaista binäärihakupuuta avaimille $2, 3, 5, 7, 11, 13, 17$:



Huom. kun binäärihakupuun avaimet tulostetaan sisäjärjestyksessä, ne tulostuvat samalla suuruusjärjestyksessä.

Joukko-operaatiot binäärihakupuussa

Oletamme yleensä, että sama avain voi esiintyä puussa vain kerran.

Binäärihakupuuehdon avulla avaimen k etsiminen puusta T voidaan suoraan ohjata haluttuun solmuun kutsulla Search($root[T], k$). Jos avainta ei ole, palautetaan Nil.

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

Operaation aikavaativuus on verrannollinen läpikäydyn polun pituuteen, eli pahimmassa tapauksessa $\Theta(h)$, missä h on puun korkeus.

Muistetaan, että h on $\Omega(\log n)$ ja $O(n)$, missä n on solmujen lukumäärä.

Rekursiivisella Search-algoritmilla myös muistintarve on verrannollinen polun pituuteen. Vakiomuistitilaan päästään helposti iteratiivisella versiolla:

```
Search(x, k)
    while x ≠ Nil and key[x] ≠ k do
            if k < key[x]
               then x ← left[x]
               else  x ← right[x]
    return x
```

Pienin avain löydetään menemällä aina vasemmalle:

```
Min(x)
    while left[x] ≠ Nil do
            x ← left[x]
    return x
```

Operaatio Max saadaan vaihtamalla kumpikin *left* *right*:iksi.

Operaation Min ja Max vaativat nekin pahimmassa tapauksessa ajan $\Theta(h)$.

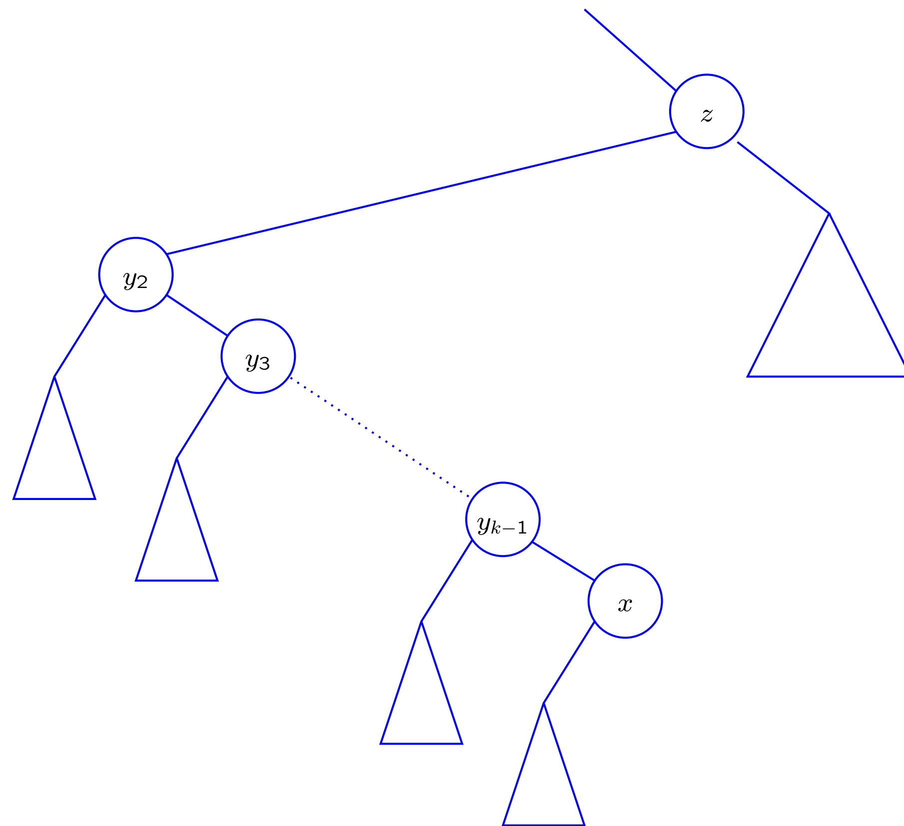
Solmun x seuraajaa Succ(x) voidaan joutua etsimään joko sen jälkeläisistä tai esi-isistä.

Jos solmulla x on oikea alipuu, seuraaja on tämän alipuun pienin alkio.

Muuten mikään solmun x jälkeläinen ei ole sitä suurempi, ja pitää etsiä sopiva esi-isä. Sopiva esi-isä on sellainen z, että jollain solmujonolla $y_1, \ldots, y_k$ pätee

- $y_1 = z$ ja $y_k = x$
- $y_2 = \mathit{left}[y_1]$
- $y_{i+1} = \mathit{right}[y_i]$ kun $2 \le i \le k$.

Toinen tapa sanoa sama asia: Etsitään laajin alipuu, jossa $\mathit{key}[x]$ on suurin avain. Nyt Succ(x) on tämän alipuun vanhempi.



Solmu z on solmun x seuraaja.

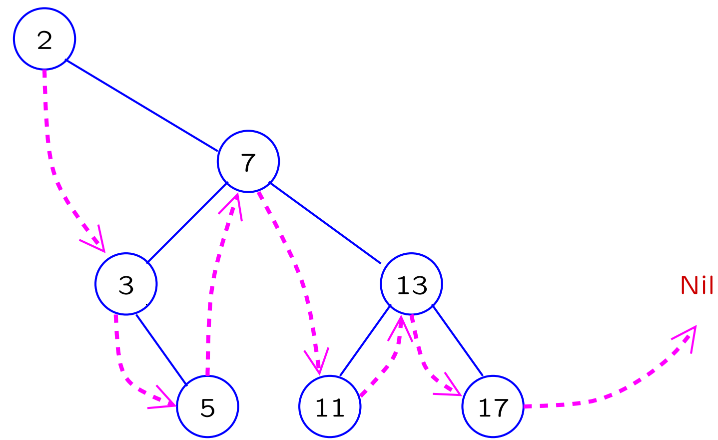
Miksi tämä toimii? Pitää osoittaa, että $key[x] < key[z]$ mutta millään y ei päde $key[x] < key[y] < key[z]$.

Ehto $key[x] < key[z]$ pätee, koska x on alipuussa $left[z]$.

Valitaan nyt mielivaltainen solmu y.

- Jos y on solmun z oikeassa alipuussa, niin $key[y] > key[z]$.
- Jos y on solmun z vasemmassa alipuussa ja $y \neq x$, niin $key[y] < key[x]$ (sillä $x = \text{Max}(left[z])$).
- Jos y ei lainkaan ole solmuun z juurtuvassa alipuussa, niin solmusta y katsoen z ja x ovat samassa alipuussa, joten
  joko $key[z] < key[y]$ ja $key[x] < key[y]$
  tai $key[z] > key[y]$ ja $key[x] > key[y]$.

Missään tapauksessa ei päde $key[x] < key[y] < key[z]$.



Kunkin solmun seuraaja on merkitty nuolella.

Saadaan seuraava pseudokoodi:

```
Succ(x)
    if right[x] ≠ Nil
        then return Min(right[x])
        else
            y ← p[x]
            while y ≠ Nil and x = right[y] do
                x ← y
                y ← p[y]
            return y
```

Tämänkin operaation aikavaativuus on pahimmassa tapauksessa $\Theta(h)$, koska voidaan joutua menemään puuta ylöspäin tai (operaatiossa Min) alaspäin lähes koko sen korkeus.

Avaimen lisäämisessä etsitään ensin Search-proseduurin tapaan paikka, jossa avaimen pitäisi olla.

Nyt siinä kohtaa onkin tyhjä puu, joka korvataan uudella solmulla.

Koska tyhjästä puusta ei (tietenkään) ole linkkiä vanhempaan, pidetään tästä kirjaa apumuuttujassa y.

Pseudokoodi on seuraavalla kalvolla. Kuten edellisillä operaatioilla, aikavaativuus on $O(h)$ ja tilavaativuus $O(1)$.

Siirrymme kohta tasapainoisiin puihin, joissa lisäyksen yhteydessä voidaan muutenkin muokata puun rakennetta tavoitteena korkeuden vähentäminen. Tässä kuitenkin solmu vain lisätään sopivaan paikkaan. Jos esim. lisätään n solmua avainten mukaisessa järjestyksessä, syntyy korkeutta $n-1$ oleva puu.

Insert(T, k)

```
z ← new puusolmu
key[z] ← k
left[z] ← right[z] ← Nil
if root[T] = Nil
   then root[T] ← z
        p[z] ← Nil
   else x ← root[T]
        while x ≠ Nil do
              y ← x
              if k < key[x]
                 then x ← left[x]
                 else x ← right[x]
        p[z] ← y
        if k < key[y]
           then left[y] ← z
           else right[y] ← z
```

Solmun z poistaminen on operaatioista hankalin. Se jakautuu eri tapauksiin sen mukaan, kuinka monta lasta solmulla z on.

z on lehti: helppoa, solmu vain poistetaan

solmulla z tasan yksi lapsi: helppoa, kuten poisto kahteen suuntaan linkitetystä listasta: osoittimet vedetään poistettavan solmun ohi.

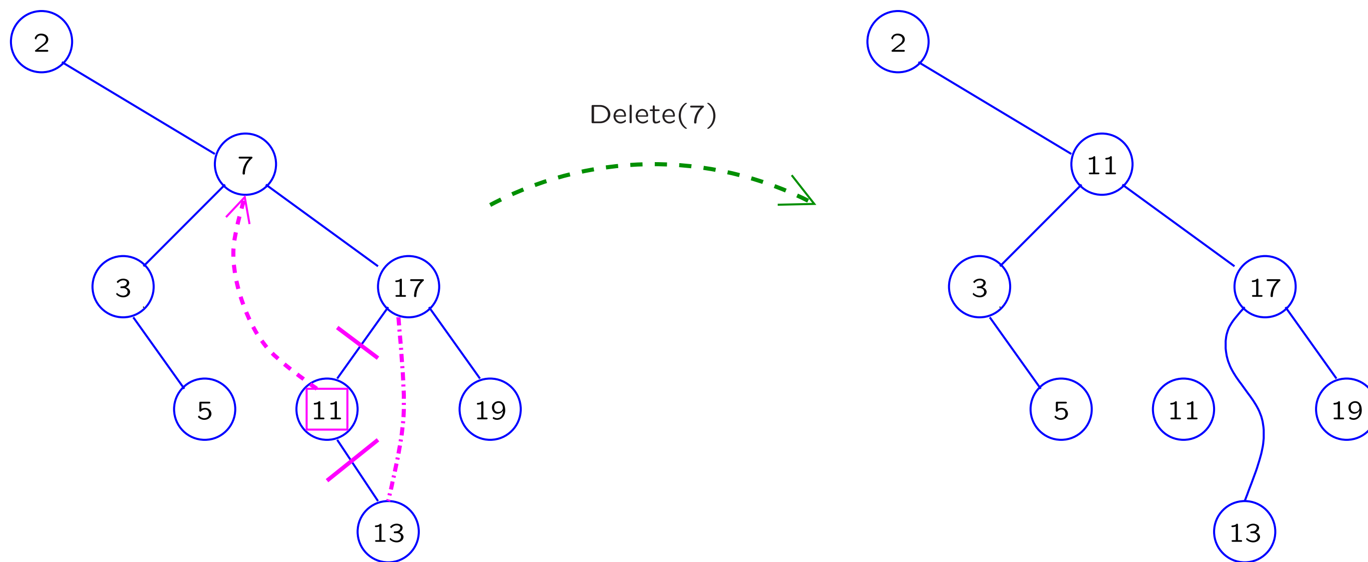
solmulla z on kaksi lasta: hankala tapaus. Solmua z ei voi poistaa rikkomatta puuta. Sen sijaan

- vaihdetaan solmun z ja seuraajasolmun $y = \text{Succ}(z)$ sisältö
- poistetaan solmu y.

Koska tässä tapauksessa $y = \text{Min}(right[z])$, niin solmulla y on korkeintaan yksi lapsi. Koska $key[z]$ ja $key[y]$ ovat peräkkäiset, vaihto ei riko hakupuuominaisuutta.

Koska rakenteesta poistuva solmu voi olla $y \neq z$, palautetaan osoitin tähän, jotta sille varattu muisti voidaan haluttaessa vapauttaa.

Kuten edellä, aikavaativuus on $O(h)$ ja tilavaativuus $O(1)$.



Kaksilapsisen solmun poisto

Pseudokoodi poisto-operaatiolle:

```
Delete(T, z)
    if left[z] = Nil and right[z] = Nil
       then                              ▷ ei lapsia
            w ← p[z]
            if w = Nil
               then root[T] ← Nil
            elseif z = left[w]
               then left[w] ← Nil
            else right[w] ← Nil
            return z

    ▷ jatkuu . . .
```

▷ Delete jatkuu . . .

```
if left[z] = Nil or right[z] = Nil
   then                          ▷ yksi lapsi
         if left[z] ≠ Nil
            then x ← left[z]
            else  x ← right[z]
         w ← p[z]
         if w = Nil
            then root[T] ← x
         elseif z = left[w]
            then left[w] ← x
         else right[w] ← x
         p[x] ← w
         return z
```

▷ jatkuu . . .

$\triangleright$ Delete jatkuu . . .

$\triangleright$ Nyt $\mathit{left}[z] \neq$ Nil **and** $\mathit{right}[z] \neq$ Nil; kaksi lasta
$y \leftarrow$ Succ(z) $\triangleright$ joten $\mathit{left}[y] =$ Nil
$x \leftarrow \mathit{right}[y]$
$w \leftarrow p[y]$
if $y = \mathit{left}[w]$
  then $\mathit{left}[w] \leftarrow x$
  else $\mathit{right}[w] \leftarrow x$
if $x \neq$ Nil
  then $p[x] \leftarrow w$
$\mathit{key}[z] \leftarrow \mathit{key}[y]$
return y

$\triangleright$ Delete loppu

Yhteenveto: osaamme toteuttaa kaikki joukko-operaatiot binäärihakupuuta käyttäen ajassa $O(h)$, missä h on puun korkeus. Tästä tiedetään

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

Jos voisimme taata $h = O(\log n)$, tietorakenteeseen voisi olla hyvin tyytyväinen. Edellä esitetyillä operaatioilla voidaan kuitenkin päätyä tilanteeseen $h = \Omega(n)$ (esim. lisätään solmut järjestyksessä).

Voimme taata $h = O(\log n)$, ja siis joukko-operaatioden toteutumisen logaritmisessa ajassa, käyttämällä tasapainoisia binäärihakupuita. Tämä voidaan tehdä monella eri tavalla, mutta perusidea on lisäysten ja poistojen yhteydessä korjata puurakennetta, jos puussa on lehtiä kovin eri korkeuksilla.

Tasapainoiset hakupuut

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

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

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

B-puut: tehokkaita levymuistia käytettäessä

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

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

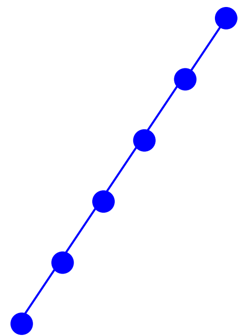
treap: satunnaisuutta käyttävä puun ja keon (heap) yhdistelmä; "odotusarvoisesti tasapainoinen"

Esittelemme tässä AVL-puut ja B-puut.

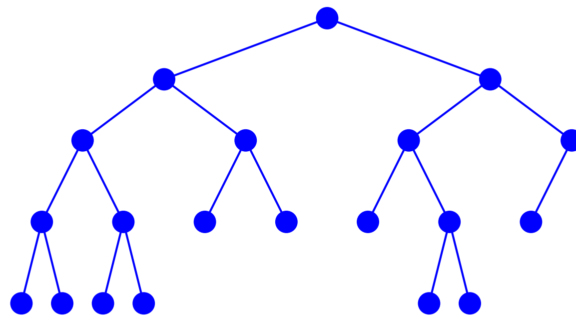
Hakunopeuden kannalta paras tilanne on täydellinen puu. Tällöin n-alkioisen puun korkeus on $\log_2(n+1)-1$. Puuta on kuitenkin vaikea pitää edes suunnilleen täydellisenä, kun siihen lisätään ja siitä poistetaan alkioita.

Pahin tilanne on lineaarinen puu, jossa kaikki oikeat tai kaikki vasemmat alipuut ovat tyhjiä. Korkeus on $n-1$.

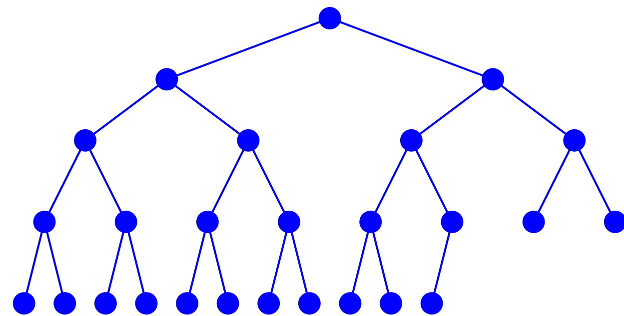
Intuitiivisesti puu on "tasapainoinen", kun se muistuttaa muodoltaan enemmän täydellistä kuin lineaarista puuta. Eri tavat määritellä tasapainoisuus täsmällisesti johtavat hieman erilaisiin tietorakenteisiin (esim. AVL-puu tai punamustat puu).



lineaarinen



tasapainoinen?



melkein täydellinen

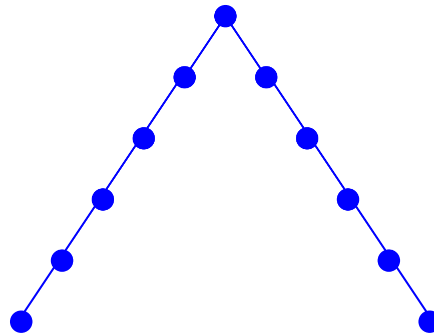
Teknisesti ottaen haluamme määritellä jonkin täydellisyyttä hieman heikomman tasapainoehdon, joka

- on helpompi pitää voimassa kuin täydellisyys mutta
- takaa kuitenkin, että puun korkeus on $\Theta(\log n)$.

Siis ylläpidon helpottamiseksi löysennämme ehtoa niin, että puun korkeus kasvaa enintään vakiokertoimella.

Tasapainoehdon intuitiivinen merkitys on yleensä karkeasti, että jokaisen solmun vasen ja oikea alipuu ovat jossain mielessä suunnilleen samankokoiset.

Huomaa, että pelkästään juuren tarkasteleminen ei riitä:



Juuren vasen ja oikea alipuu näyttävät samankokoisilta, mutta puun korkeus on $\lfloor n/2 \rfloor$ eli puu ei ole tasapainoinen.

AVL-puut [Adel'son-Vel'skii ja Landis 1962]

Normaalin binääripuun solmussa olevien kenttien key, $left$, $right$ ja p lisäksi jokaisessa AVL-puun solmussa on kenttä $height$, joka ilmoittaa solmun korkeuden.

Muistin virkistykseksi:

- Solmun korkeus on kaarten määrä pisimmällä johonkin sen jälkeläiseen johtavalla polulla.
- Esityisesti lehden korkeus on 0.
- Puun (tai alipuun) korkeudella tarkoitetaan sen juuren korkeutta.

Edellisen kanssa on yhteensopivaa, että tyhjän binääripuun Nil korkeudeksi määritellään -1. Siis olettaen $height$-kentät voidaan puun korkeus laskea funktiolla

```
Height(T)
    if T = Nil
        then return −1
        else return height[T]
```

AVL-puulta vaaditaan, että se toteuttaa seuraavan tasapainoehdon:

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

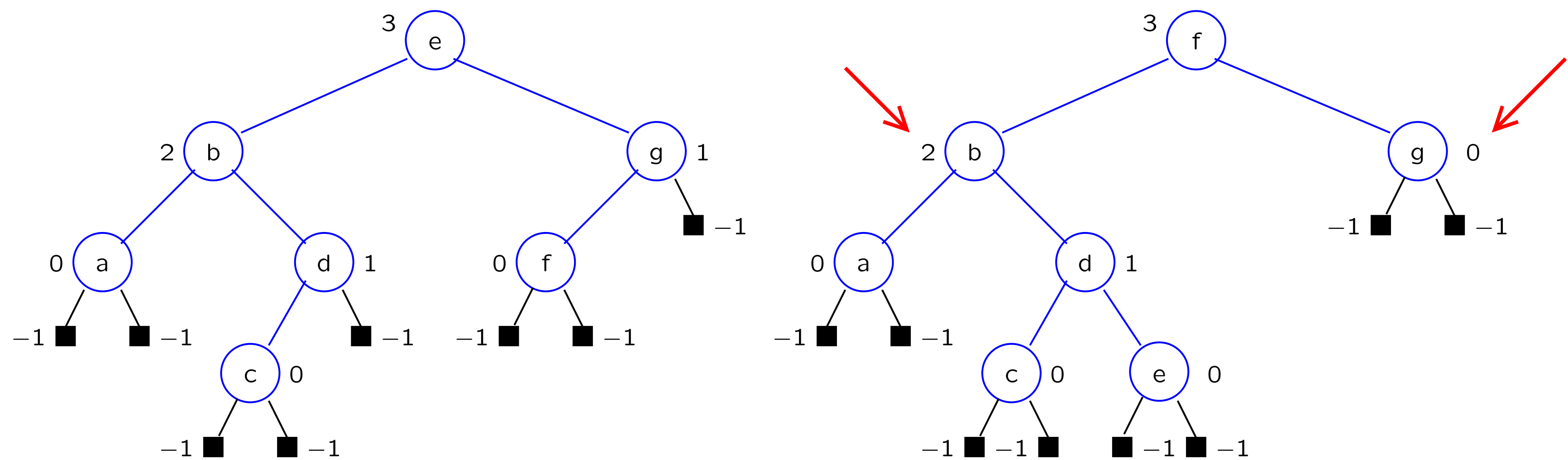
Toisin sanoen vaaditaan

$$|\,\mathsf{Height}(\mathit{left}[x]) - \mathsf{Height}(\mathit{right}[x])\,| \leq 1 \qquad \text{kaikilla solmuilla } x.$$

Haluamme osoittaa, että

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

Allaolevaan kahteen puuhun on merkitty solmujen korkeudet ja selvyyden vuoksi myös Nil-alipuut (musta neliö).



Vasemmanpuoleinen puu toteuttaa AVL-tasapainoehdon, oikeanpuoleinen ei toteuta.

Tarkastellaan ensin AVL-puun korkeuden ja solmujen lukumäärän suhdetta.

Lause 3.5: Jos AVL-puun korkeus on h, niin siinä on ainakin $F_{h+3} - 1$ solmua, missä F_i on i:s Fibonaccin luku.

Ennen lauseen todistusta tarkastellaan sen seurauksia.

Fibonaccin luvut $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ määritellään palautuskaavalla

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_{n+2} &= F_{n+1} + F_n \qquad \text{kun } n \geq 0.
\end{aligned}
$$

Fibonaccin luvuille tunnetaan myös eksplisiittinen kaava

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

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

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

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

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

Olkoon $S(h)$ pienin solmujen lukumäärä korkeutta h olevassa AVL-puussa. Lauseen mukaan

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

Solmujen lukumäärä kasvaa siis eksponentiaalisesti korkeuden funktiona.

Olkoon puolestaan $H(n)$ suurin n-solmuisen AVL-puun korkeus. Selvästi jos $H(n) = h$, niin

$$n \geq S(h) = F_{h+3} - 1 \approx \frac{1{,}618^{h+3}}{2{,}236} - 1.$$

Unohtamalla pyöristysvirheet ja ratkaisemalla saamme

$$\begin{aligned}
n &\geq \frac{1{,}618^{h+3}}{2{,}236} - 1 \\
2{,}236 \cdot (n+1) &\geq 1{,}618^{h+3} \\
\log_2(n+1) + \log_2 2{,}236 &\geq (h+3) \log_2 1{,}618 \\
h &\leq \frac{\log_2(n+1) + \log_2 2{,}236}{\log_2 1{,}618} - 3 \\
&\approx 1{,}44 \log_2(n+1) - 1{,}33.
\end{aligned}$$

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

Lauseen 3.5 todistus: Halutaan muodostaa pienin AVL-puu, jonka korkeus on h. Koska ainakin toisen alipuun korkeuden on oltava $h-1$ ja alipuiden korkeuksien erotus on korkeintaan 1, kannattaa alipuiden korkeuksiksi valita $h-1$ ja $h-2$. Lisäksi alipuut kannattaa valita mahdollisimman vähäsolmuisiksi. Siis

$$S(h) = S(h-1) + S(h-2) + 1,$$

missä on laskettu yhteen kummankin alipuun solmut sekä juurisolmu.

Ylläoleva pätee, kun $h \geq 2$. Pieniä puita tarkastelemalla nähdään lisäksi $S(-1) = 0$ ja $S(1) = 1$.

Ratkaisemisen helpottamiseksi määritellään $P(h) = S(h) + 1$. Nyt

$$\begin{aligned} P(-1) &= 1 \\ P(0) &= 2 \\ P(h+2) &= P(h+1) + P(h). \end{aligned}$$

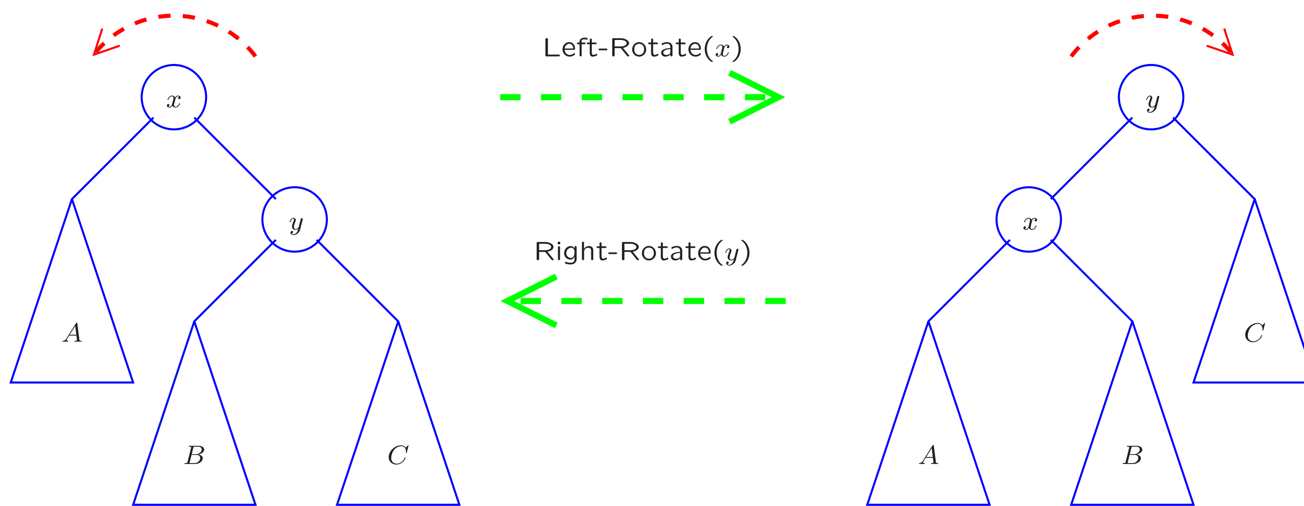
Funktio $P(h)$ toteuttaa saaman palautuskaavan kuin Fibonaccin luku F_h. Lisäksi $P(-1) = 1 = F_2$ ja $P(0) = 2 = F_3$, joten ratkaisuksi saadaan $P(h) = F_{h+3}$. Siis $S(h) = P(h) - 1 = F_{h+3} - 1$. $\square$

Johtopäätös: jos binäärihakupuu saadaan toteuttamaan AVL-puun tasapainoehto, niin joukko-operaatiot sujuvat ajassa $O(\log n)$.

Ongelma: miten lisäys ja poisto suoritetaan rikkomatta tasapainoehtoa?

Ongelman ratkaisu on soveltaa lisäyksen tai poiston jälkeen sopivia kiertoja, joilla mahdollisesti rikkoutunut ehto saadaan taas voimaan.

Kierto on paikallinen operaatio, joka nostaa joitain alipuita ylemmäs ja painaa joitain alemmas. Binäärihakupuuehto pysyy voimassa. Kiertoja sovelletaan myös muissa hakupuurakenteissa (punamusta, splay).



Vasemmassa kierrossa A painuu ja C nousee; oikeassa kierrossa päin vastoin.

Binäärihakupuuehto pysyy voimassa:

A:n avaimet $< x <$ B:n avaimet $< y <$ C:n avaimet.

Sovimme, että kierto-operaatio palauttaa osoittimen (ali)puun uuteen juureen. Tyypillinen kutsu voisi siis olla esim.

$$right[z] \leftarrow \text{Left-Rotate}(right[z]).$$

Saamme seuraavat algoritmit:

Left-Rotate(x)

$y \leftarrow right[x]$
$p[y] \leftarrow p[x]$
$right[x] \leftarrow left[y]$
$p[right[x]] \leftarrow x$
$left[y] \leftarrow x$
$p[x] \leftarrow y$
$height[x] \leftarrow \max\{\text{Height}(left[x]), \text{Height}(right[x])\} + 1$
$height[y] \leftarrow \max\{\text{Height}(left[y]), \text{Height}(right[y])\} + 1$
return y

Right-Rotate(x)

$y \leftarrow left[x]$
$p[y] \leftarrow p[x]$
$left[x] \leftarrow right[y]$
$p[left[x]] \leftarrow x$
$right[y] \leftarrow x$
$p[x] \leftarrow y$
$height[x] \leftarrow \max\{\text{Height}(left[x]), \text{Height}(right[x])\} + 1$
$height[y] \leftarrow \max\{\text{Height}(left[y]), \text{Height}(right[y])\} + 1$
return y

Alkion x lisääminen AVL-puuhun tapahtuu seuraavasti:

1. Lisää x niin kuin mihin tahansa binäärihakupuuhun välittämättä tasapainoehdosta.

2. Jos lisäyksen jälkeen jokin solmu on epätasapainossa (ts. tasapainoehto ei päde), korjaa tilanne tekemällä kiertoja.

Kohta 2 voi vaikuttaa ongelmalliselta, koska

- ei ole selvää, ovatko pelkät kierrot riittävän voimakas työkalu minkä tahansa ongelmatilanteen korjaamiseen

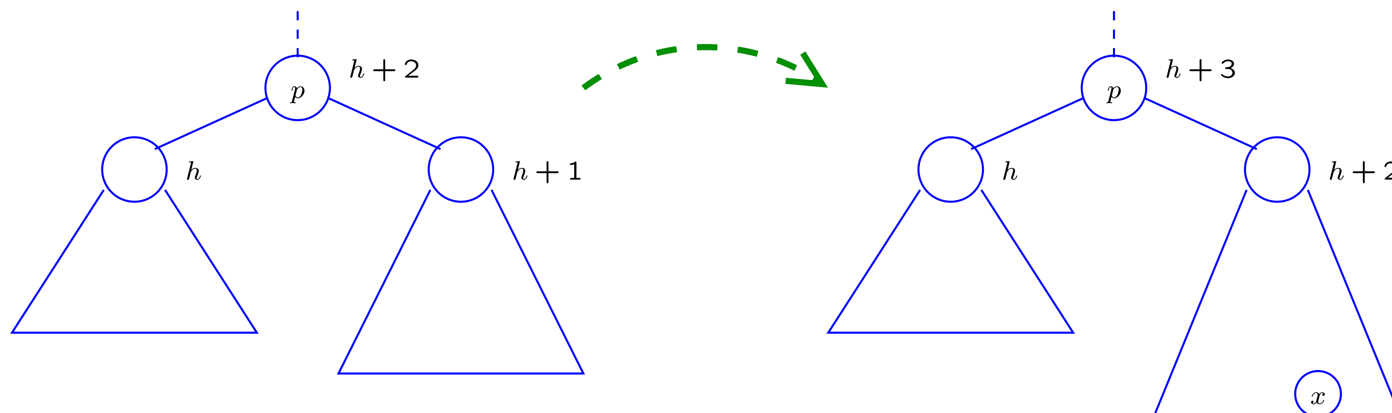
- kierrot saattavat aiheuttaa epätasapainoja solmuihin, jotka aiemmin olivat tasapainossa.

Osoittautuu kuitenkin, että jos puu ennen lisäystä toteuttaa AVL-tasapainoehdon (kuten tässä tietysti oletetaan), niin lisäyksen jälkeen riittää suorittaa yksi tai kaksi kiertoa puun saamiseksi taas tasapainoon.

Tarkastellaan solmuja, jotka x:n lisäys saattoi epätasapainoon. Nämä ovat kaikki uuden solmun x esi-isiä. Olkoon p näistä syvimmällä. Tarkastellaan tapausta, että x lisättiin solmun p oikeaan alipuuhun. Lisäys vasempaan alipuuhun on symmetrinen.

Olkoon h vasemman alipuun korkeus, joka on sama ennen ja jälkeen lisäyksen. Koska lisäys oikeaan alipuuhun rikkoi tasapainoehdon, niin

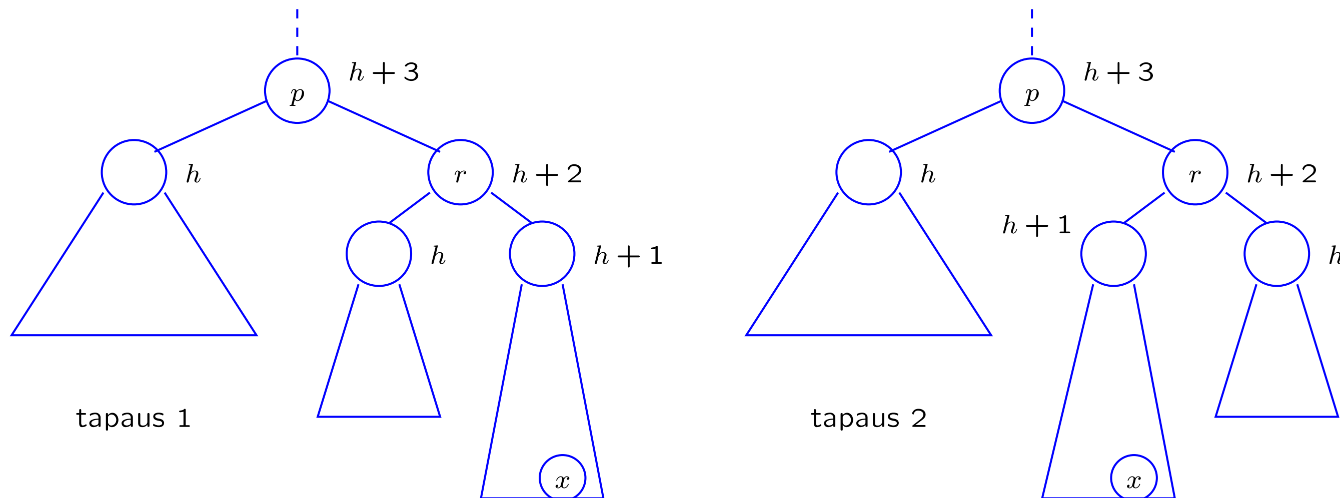
- oikean alipuun korkeus oli $h+1$ ennen lisäystä ja $h+2$ lisäyksen jälkeen
- solmun p korkeus oli $h+2$ ennen lisäystä ja $h+3$ lisäyksen jälkeen.



Kolmion pohja esittää alipuun alimman lehden tasoa.

Tarkastelu jakautuu kahteen tapaukseen:

Tapaus 1: x on oikean lapsen r oikeassa alipuussa
Tapaus 2: x on oikean lapsen r vasemmassa alipuussa.

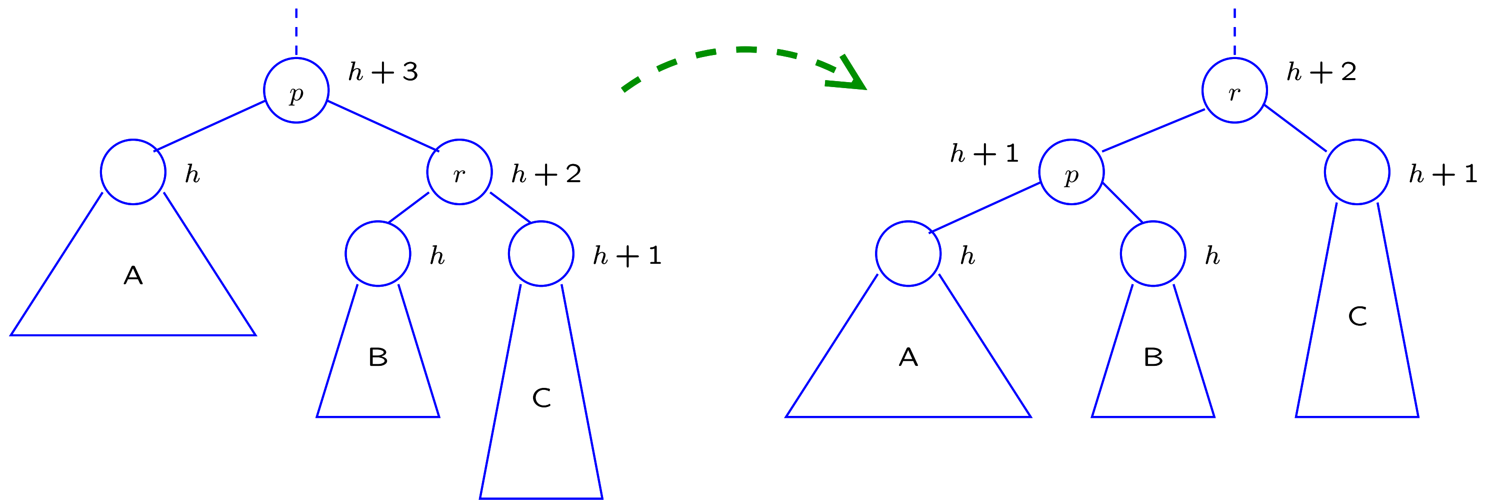


Koska oletuksen mukaan p oli syvin epätasapainoon joutunut solmu, alkion x lisäys

- kasvatti oikean lapsen r korkeutta mutta
- ei saattanut r:ta epätasapainoon

joten r:n ennallaan säilyneen alipuun korkeuden on oltava h, kuten kuvaan on merkitty.

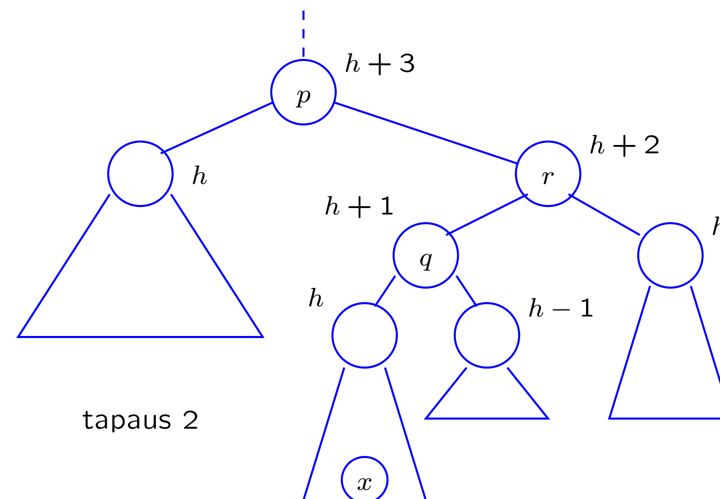
Tapauksessa 1 solmu p saadaan tasapainoon kiertämällä vasemmalle.



Huomaa, että kierto pienensi alipuun korkeutta. Alipuu on siis nyt saman korkuinen kuin ennen lisäystä. Siis tasapaino palautui myös kaikkiin esi-isiin, jotka mahdollisesti olivat epätasapainossa.

Koko puu on tasapainossa, lisätoimia ei enää tarvita.

Tapauksessa 2 solmu x on siis lisätty solmun p oikean lapsen r vasempaan lapseen. Meidän täytyy vielä tarkastella tämän vasemman lapsen q kumpaakin alipuuta erikseen.

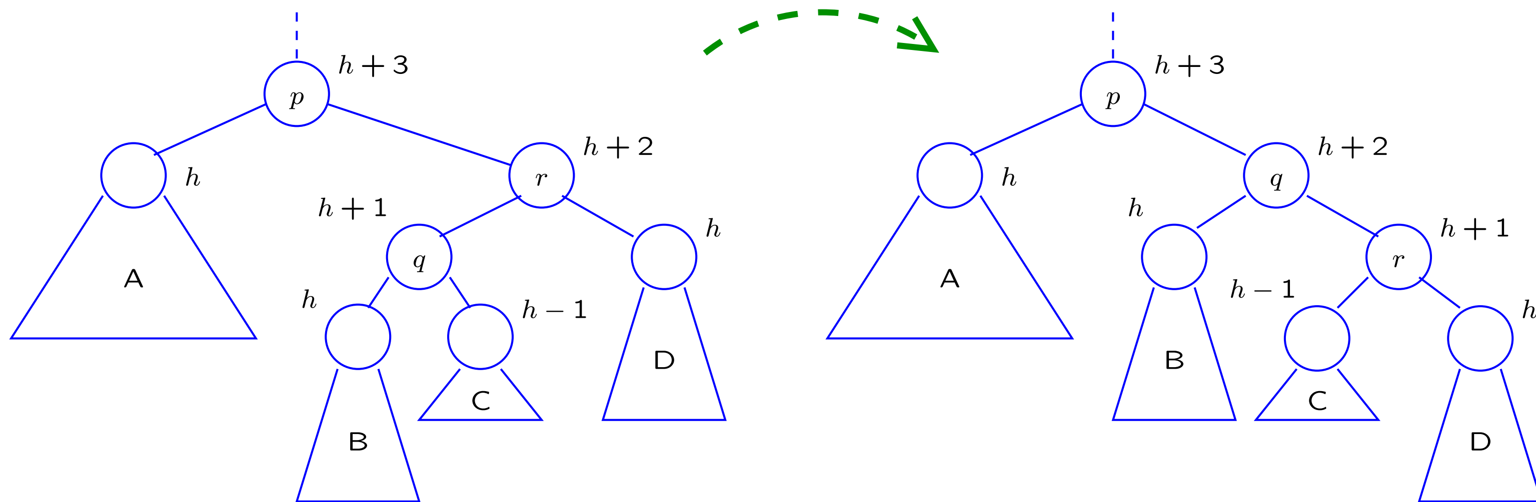


Jatkon kannalta ei ole olennaista, onko x solmun q vasemmassa vai oikeassa alipuussa. Kuvassa se on konkreettisuuden vuoksi merkitty vasempaan.

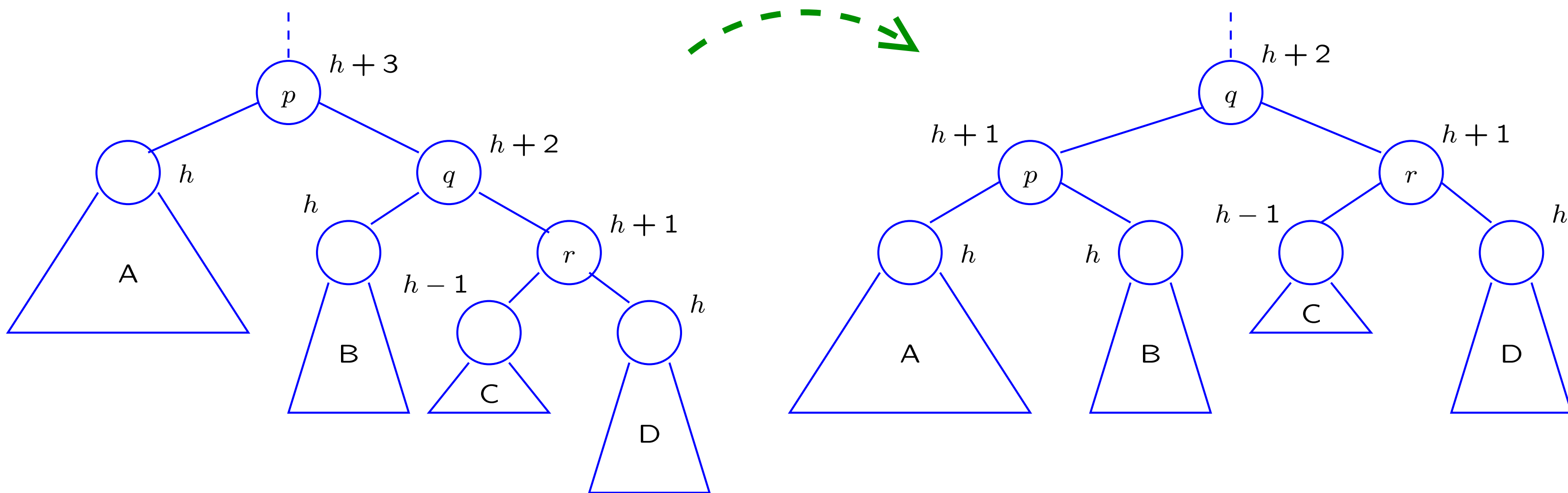
Oleellista sen sijaan on, että solmun q toinen alipuu on korkeudeltaan tasan yhtä pienempi. Tämä seuraa taas siitä, että x:n lisäys

- kasvatti q:n korkeutta mutta
- ei saattanut q:ta epätasapainoon.

Teemme ensin kierron oikealle solmussa r:

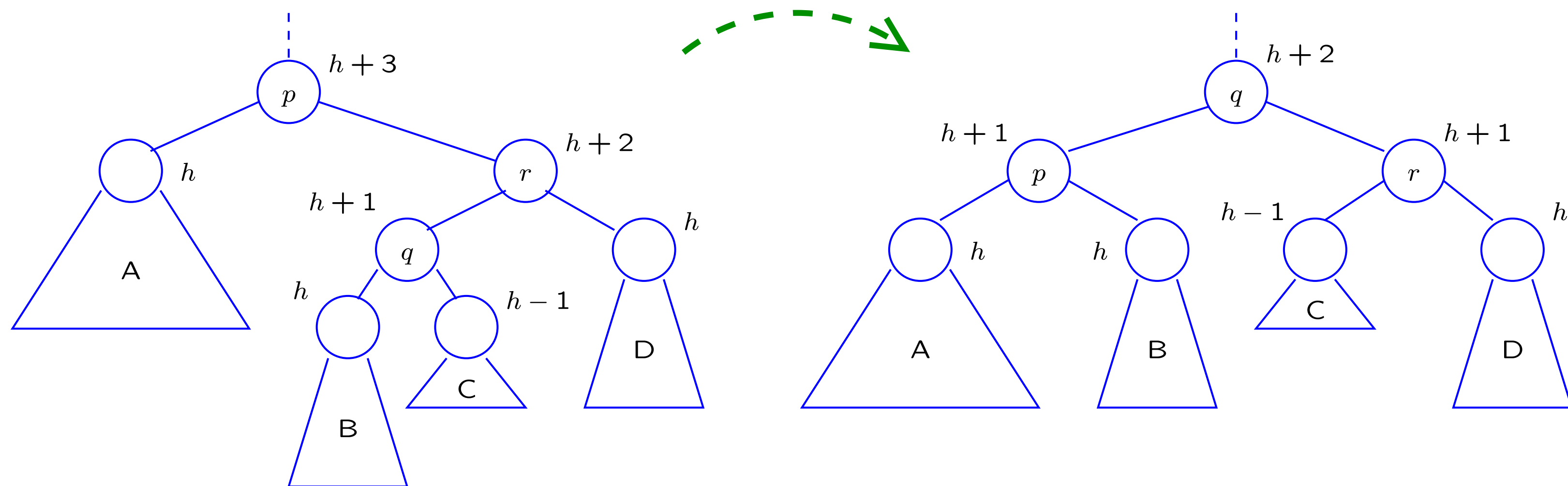


Solmu p on edelleen epätasapainossa. Nyt olemme kuitenkin oleellisesti samassa tilanteessa kuin tapauksessa 1. Puu tasapainottuu kiertämällä vielä vasemmalle solmussa p.



Taas alipuun korkeus palasi lisäystä edeltäneeseen, joten muukin osa puusta on nyt tasapainossa.

Tapauksessa 2 tarvittavat kaksi kiertoa on usein havainnollista yhdistää yhdeksi operaatioksi, jota kutsutaan oikea-vasen-kaksoiskierroksi.



Tässä siis ei ole väliä, kumpi alipuista B ja C on korkeampi.

Esitetään täydellisyyden vuoksi vielä pseudokoodi
oikea-vasen-kaksoiskierrolle ja sen peilikuvalla vasen-oikea-kaksoiskierrolle,
jota tarvitaan lisäyksessä vasemman alipuun oikeanpuoleiseen haaraan.

Right-Left-Double-Rotate(x)

$right[x] \leftarrow$ Right-Rotate($right[x]$)
return Left-Rotate(x)

Left-Right-Double-Rotate(x)

$left[x] \leftarrow$ Left-Rotate($left[x]$)
return Right-Rotate(x)

Kirjoitamme koko lisäysalgoritmin rekursiivisena funktiona AVL-Insert.
Funktio saa argumenttina lisättävän avaimen ja osoittimen puun juureen.
Funktio palauttaa osoittimen puuhun, johon avain nyt on lisätty.

Avain k siis lisätään puuhun T kutsulla

$root[T] \leftarrow$ AVL-Insert($root[T], k$).

AVL-Insert(x, k)

```
if x = Nil
   then x ← new puusolmu
        key[x] ← k
        left[x] ← right[x] ← Nil
elseif k > key[x]    ▷ Lisäys oikeaan alipuuhun
   then right[x] ← AVL-Insert(right[x], k)
        if Height(right[x]) = Height(left[x]) + 2
           then if k > key[right[x]]
                   then ▷ Tapaus 1 edellä
                        x ← Left-Rotate(x)
                   else ▷ Tapaus 2 edellä
                        x ← Right-Left-Double-Rotate(x)
elseif k < key[x]    ▷ Lisäys vasempaan alipuuhun
                     ▷ Peilisymmetrinen edellisen kohdan kanssa
   then left[x] ← AVL-Insert(left[x], k)
        if Height(left[x]) = Height(right[x]) + 2
           then if k < key[left[x]]
                   then x ← Right-Rotate(x)
                   else x ← Left-Right-Double-Rotate(x)
else       ▷ Avain on jo puussa.
           ▷ Älä tee mitään.
height[x] ← max { Height(left[x]), Height(right[x]) } + 1
return x
```

Yhteenveto lisäyksestä AVL-puuhun:

Olkoon $R(h)$ lisäyksen aikavaativuus, kun puun korkeus on h. Tarkastelemalla rekursiivista algoritmia ALV-Insert nähdään, että

$$R(h+1) \leq c + R(h)$$

jollain vakiolla c. Vakio c sisältää vertailuihin, kiertoihin yms. yhdessä solmussa kuluvan ajan.

Tästä nähdään suoraan, että $R(h) = O(h)$. Siis lisääminen n-alkioiseen puuhun sujuu ajassa $O(\log n)$, sillä puun korkeus on AVL-tasapainoehdon takia $\Theta(\log n)$.

Lisäksi tiedämme, että itse asiassa kierto joudutaan tekemään korkeintaan yhdessä solmussa kunkin lisäyksen yhteydessä.

Poistaminen AVL-puusta noudattaa samaa perusajatusta kuin lisäys:

1. Tee poisto kuten tasapainottamattomasta hakupuusta.

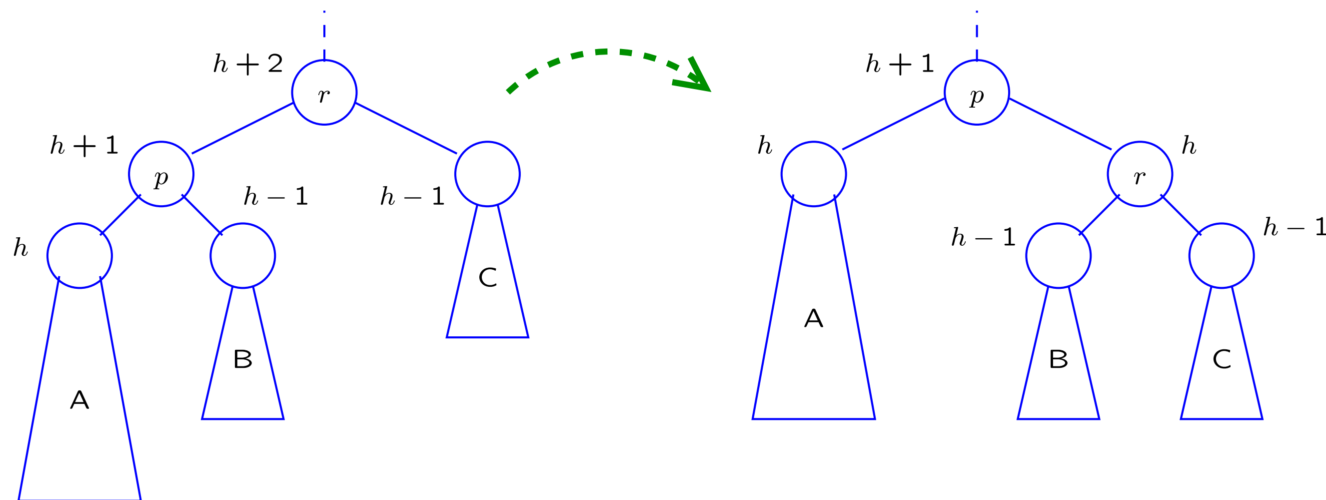
2. Tasapainota puu suorittamalla sopivat kierrot poistuneen solmun esi-isissä.

Poiston yhteydessä kiertoja voidaan kuitenkin joutua suorittamaan useassa solmussa, toisin kuin lisäyksessä.

Huom. punamustien puiden eräs etu on, että myös poiston jälkeen tarvitaan vain vakiomäärä kiertoja.

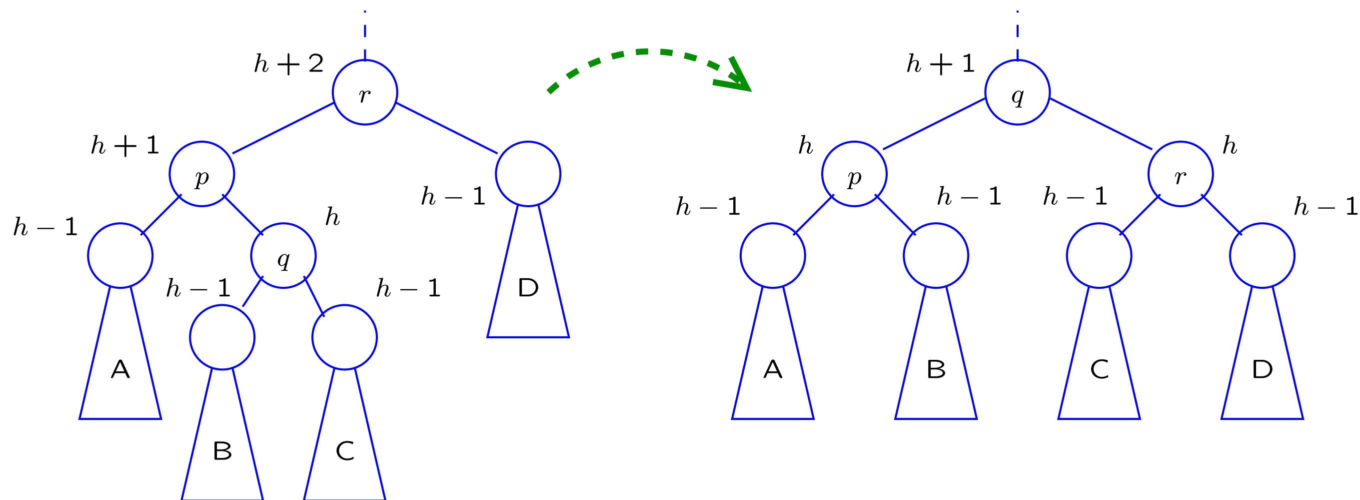
Tarkastellaan tilannetta, jossa poisto solmun r oikeasta alipuusta on saanut puun epätasapainoon. Olkoon h oikean alipuun korkeus ennen poistoa. Poiston jälkeen oikean alipuun korkeus on siis $h-1$ ja epätasapainoisuuden takia vasemman alipuun korkeus $h+1$.

Jos vasemman alipuun alipuiden A ja B korkeudet ovat h ja $h-1$, niin solmu r tasapainottuu kiertämällä oikealle:



Tässä koko tarkasteltavan alipuun korkeus pienenee, joten esi-isiä voidaan joutua edelleen tasapainottamaan. Jos myös B:n korkeus olisi h, puu tulisi kuntoon tällä yhdellä kierrolla.

Jos vasemman alipuun vasemman alipuun A korkeus on $h-1$, tehdään kaksoiskierto:



Kuvassa on oletettu puiden C ja D kummankin korkeudeksi $h-1$. Puu tulee tällä operaatiolla tasapainoon, vaikka toinen näistä olisikin $h-2$.

Taas havaitaan, että voimme joutua jatkamaan tasapainotusta ylempänä puussa.

Esitämme AVL-puun poisto-operaation pseudokoodin käyttäen apuna tasapainottamattoman binäärihakupuun poisto-operaatiota (algoritmi Delete sivuilla 148–150). Muistetaan, että Delete palauttaa osoittimen poistetun avaimen sisältäneeseen solmuun,

- johon ei enää ole linkkejä muualta puusta mutta
- jossa edelleen on jäljellä vanhat linkit vanhempaan jne.

Poistuneen solmun vanhempi on alin solmu, jonka tasapaino on voinut häiriintyä. Etenemme tästä solmusta ylöspäin korjaten tasapainohäiriöt ja päivittäen solmujen korkeudet.

Käytämme apumuuttujaa y osoittamaan kierrossa syntyneen alipuun juureen, jotta $p[z]$ säilyy osoitteena käsiteltävän puun vanhempaan.

AVL-Delete(T, k)

```
z ← p[Delete(T, k)]        ▷ ks. s. 148-150
while z ≠ Nil
      do if Height(left[z]) = Height(right[z]) + 2
            then if Height(left[left[z]]) = Height(left[z]) − 1
                      then y ← Right-Rotate(z)
                      else  y ← Left-Right-Double-Rotate(z)
         elseif Height(right[z]) = Height(left[z]) + 2
            then if Height(right[right[z]]) = Height(right[z]) − 1
                      then y ← Left-Rotate(z)
                      else  y ← Right-Left-Double-Rotate(z)
         if p[z] ≠ Nil
            then if z = left[p[z]]
                      then left[p[z]] ← y
                      else  right[p[z]] ← y
            else  root[T] ← y
         height[z] ← max { Height(left[z]), Height(right[z]) } + 1
         z ← p[z]
```

Yhteenveto AVL-puista:

- AVL-tasapainoehto takaa puun korkeuden $\Theta(\log n)$, missä n on alkioiden lukumäärä
- siis Search toimii aina ajassa $O(\log n)$
- tasapainoehtoa pidetään yllä tekemällä vakioaikaisia kiertoja hakupolulla
- pahimmassa tapauksessa tarvitaan yksi kierto-operaatio kussakin hakupolun solmussa

$\Rightarrow$ kaikki dynaamisen joukon operaatiot toimivat pahimmassakin tapauksessa ajassa $O(\log n)$

Tasapainottamattomaan puuhun verrattuna AVL-puut tarvitsevat kuhunkin solmuun ylimääräisen $height$-laskurin, jolle käytännössä riittää 8 bittiä. Periaatteessa muistia voitaisiin hieman säästää tallentamalla vain vasemman ja oikean alipuun korkeuksien ero (-1, 0 tai $+1$; 2 bittiä).

B-puut

AVL-puut soveltuvat hakemistorakenteeksi, kun kaikki data mahtuu keskusmuistiin:

- pienin osoitettava yksikkö on sana eli pari tavua
- mikä tahansa sana löytyy vakioajassa
- hakuaika alle mikrosekunti

Usein dataa on niin paljon, että tarvitaan levymuistia:

- yksittäisen alkion hakuaika vaihtelee, tyypillisesti useita millisekunteja
- oleellisesti samassa ajassa saadaan kokonainen lohko (vähintään joitain kilotavuja) fyysisesti lähekkäin talletettua dataa

B-puu on tasapainoinen puurakenne, joka ottaa huomioon levymuistin erityispiirteet: lyhennetään polkuja lisäämällä solmujen kokoa.

Huom. ylläoleva on yliyksinkertaistus; erityisesti välimuistin (cache) vaikutus keskusmuistin käsittelyyn on yleensä merkittävä.

B-puu yleistää hakupuun käsitteen tilanteeseen, jossa yhdellä solmulla voi olla enemmän kuin kaksi lasta.

Esittelemme tässä B$^+$-puut, joissa varsinaiset avaimet ja muu data talletetaan vain lehtisolmuihin. Sisäsolmuihin talletetaan vain reititysinformaatiota, jonka perusteella avain löydetään samantyylisellä hakuproseduurilla kuin binäärihakupuussa.

B$^+$-puun lehtisolmussa x on seuraavat kentät:

- bitti $\mathit{leaf}[x] = \text{True}$ merkiksi, että kyseessä on lehti
- avainarvot $key_1[x], \ldots, key_m[x]$ ja osoittimet vastaavaan dataan $data_1[x], \ldots, data_m[x]$
- laskuri $n[x] \leq m$ kertoo, kuinka monta avainta solmussa todella on.

Avainarvot ovat järjestyksessä:

$$key_1[x] < key_2[x] < \ldots < key_{n[x]}[x];$$

arvot $key_{n[x]+1}, \ldots, key_m[x]$ eivät ole (juuri nyt) käytössä.

Parametri m on ennalta kiinnitetty vakio. Ideana on, että solmun koko on lohko tai pari, jolloin m voi olla satoja tai tuhansia.

B$^+$-puun sisäsolmussa x on seuraavat kentät:

- $\mathit{leaf}[x] = \text{False}$
- viitta-arvot $\mathit{router}_1[x], \ldots, \mathit{router}_{m-1}[x]$ ja lapsiosoittimet $c_1[x], \ldots, c_m[x]$
- laskuri $n[x] \leq m - 1$ kertoo, kuinka monta viitta-arvoa solmussa todella on.

Viitta-arvot ja lapsiosoittimet noudattavat ehtoa

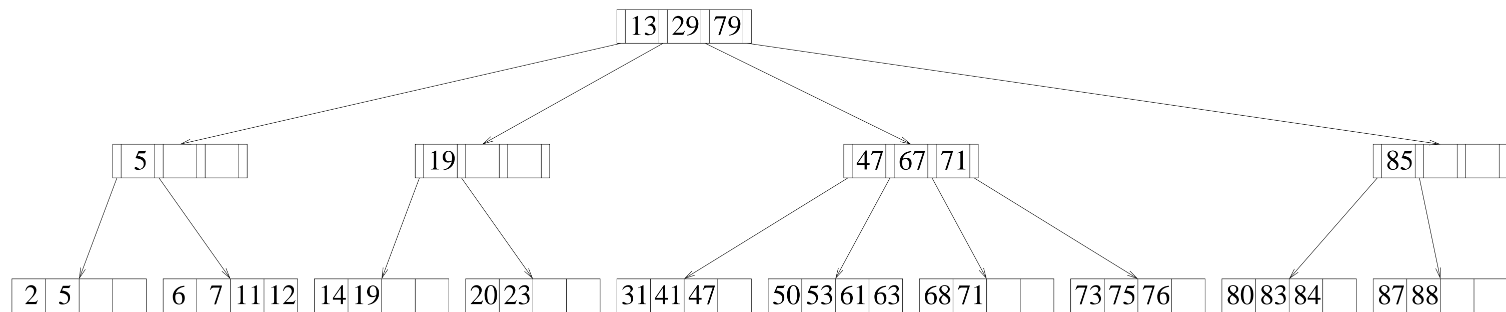
$$k_1 \leq \mathit{router}_1[x] < k_2 \leq \mathit{router}_2[x] \leq \ldots < k_{n[x]} \leq \mathit{router}_{n[x]}[x] < k_{n[x]+1},$$

missä k_i on mikä tahansa alipuussa $c_i[x]$ oleva avain. Viitta-arvot eivät välttämättä ole puussa esiintyviä avaimia.

Arvot $\mathit{router}_{n[x]+1}, \ldots, \mathit{router}_{m-1}[x]$ ja $c_{n[x]+2}, \ldots, c_m$ eivät ole käytössä.

Käytämme jatkossa B$^+$-puista termiä "B-puu". Alkuperäisissä B-puissa myös sisäsolmuissa on avaimia ja niihin liittyviä dataosoittimia. B$^+$-puun etuja:

- kun lohkokoko on vakio, niin haarautumisaste on suurempi, koska sisäsolmuissa ei tarvi varata tilaa data-kentille
- operaatiot ovat hieman yksinkertaisempia.



Esimerkki B-puusta.

- kenttiä $n[x]$ ja $leaf[x]$ ei ole merkitty näkyviin
- käytännössä solmut ovat paljon isompia

B-puulle asetetaan seuraavat tasapainoisuusehdot:

1. Jokainen juuresta lehteen johtava polku on yhtä pitkä.

2. Juurta lukuunottamatta jokainen solmu on ainakin puolillaan: Olkoon t vakio ja $s(x)$ lehden x avainten lukumäärä tai sisäsolmun x lasten lukumäärä. Nyt
   - jos x on puun ainoa solmu, niin $1 \leq s(x) \leq 2t$
   - jos x on juuri ja puussa on muitakin solmuja, niin $2 \leq s(x) \leq 2t$
   - muuten $t \leq s(x) \leq 2t$

Kaikki polut siis pidetään tasan saman pituisina, mutta liikkumavaraa saadaan vaihtelemalla solmujen kokoa.

Lause 3.6: Jos B-puun korkeus on $h \geq 1$, siinä on ainakin $2t^h$ avainta.

Todistus: Syvyydellä 1 on ainakin 2 solmua. Jos syvyydellä d on x solmua, niin syvyydellä $d+1$ on ainakin tx solmua, ellei taso d ole jo lehtitaso. Siis syvyydellä h on ainakin $2t^{h-1}$ solmua. Kussakin lehdessä on ainakin t avainta. $\square$

Lause 3.7: Jos B-puussa on n avainta, sen korkeus on korkeintaan $\log_t(n/2)$.

Todistus: Jos korkeus on h, niin edellisen mukaan

$$\begin{aligned} n &\geq 2t^h \\ n/2 &\geq t^h \\ \log_t(n/2) &\geq h. \end{aligned}$$

$\square$

Haku B-puussa

Haku alkaa juuresta ja etenee binääripuun tapaan:

- Jos ollaan lehdessä, käy avaimet läpi ja katso, löytyikö haluttu.
- Jos ollaan sisäsolmussa x, etsittävänä avain k ja $router_{n[x]}[x] < k$, niin siirrytään solmuun $c_{n[x]+1}[x]$.
- Muuten siirrytään solmuun $c_i[x]$, missä i on pienin, jolla $k \leq router_i[x]$.

Sovitaan joistain algoritmin esitykseen liittyvistä seikoista:

- Solmun x käsittely aloitetaan kutsulla DiskRead(x), joka hakee solmun x keskusmuistiin.
- Jos solmua x muutetaan, muutokset pitää lopuksi tallettaa kutsulla DiskWrite(x).
- Oletamme, että juuri $root[T]$ on valmiiksi keskusmuistissa.

Huomioita:

- DiskRead ja DiskWrite voivat vaatia 10 000-kertaisen ajan muihin operaatioihin verrattuna.
- Algoritmien nopeutta kannattaa mitata ensisijaisesti levyoperaatioiden määrällä; tässä tilanteessa O-notaatio on harhaanjohtava.
- Tietokanta-algoritmeilla on muitakin kriteerejä kuin yksittäisen operaation nopeus, esim. rinnakkaistuvuus ja virheistä toipuminen.

B-puun hakuoperaatiolle saadaan seuraava pseudokoodi:

```
BTreeSearch(T, k)
    x ← root[T]
    while not leaf[x]
          do i ← 1
             while i ≤ n[x] and k > router_i[x] do i ← i + 1
             x ← c_i[x]
             DiskRead(x)
    i ← 1
    while i ≤ n[x] and k > key_i[x] do i ← i + 1
    if i ≤ n[x] and key_i[x] = k
       then return (x, i)
       else return Nil
```

Hakuoperaation aikavaativuus:

- Käsiteltävien solmujen lukumäärä on puun korkeus $+1 = O(\log_t n)$.
- Solmua kohti tehdään $O(t)$ operaatiota.

$\Rightarrow$ Aikavaativuus on $O(t \log_t n)$.

- Kun ajatellaan, että t on vakio, niin aikavaativuus on $O(\log n)$.
- DiskRead-kutsuja korkeintaan $\log_t(n/2)$ kappaletta.

Jos esim. $t = 100$, niin kaikilla $n \leq 2\,000\,000$ kolme DiskRead-operaatiota riittää.

Aikavaativuus $O(t \log_t n)$ kertoo sisimmän silmukan suorituskertojen lukumäärän, mutta tässä ei ole yhtään levyoperaatiota.

Tiettyyn rajaan saakka DiskRead-operaatiot dominoivat, joten käytännössä ei ole järkevää ajatella, että aikavaativuus kasvaisi t:n kasvaessa.

(Jos kuitenkin ajatellaan, että t ei ole vakio, saadaan aikavaativuus $O((\log_2 t)(\log_t n)) = O(\log_2 n)$ käyttämällä binäärihakua kunkin solmun sisällä.)

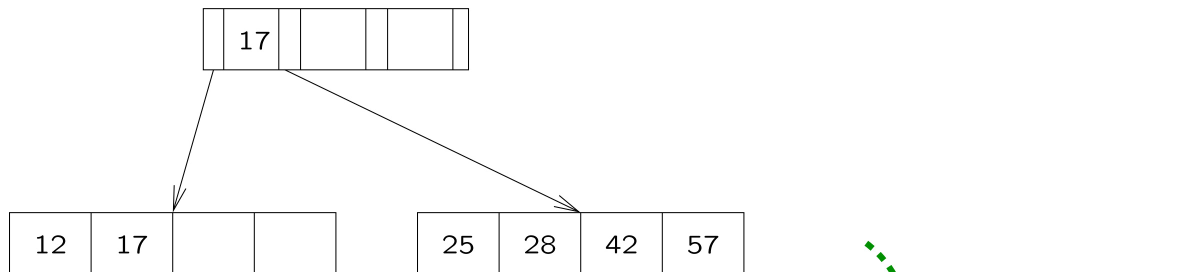
Lisäys B-puuhun

Lisättäessä avain k etsitään ensin lehti y, johon se kuuluisi, kuten BTreeSearch-operaatiossa.

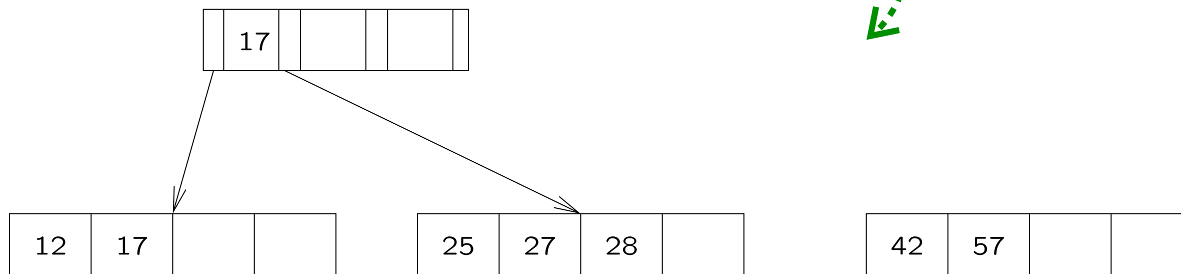
Jos solmussa y on alle $2t$ avainta, lisätään k oikeaan kohtaan. Muuta ei tarvita.

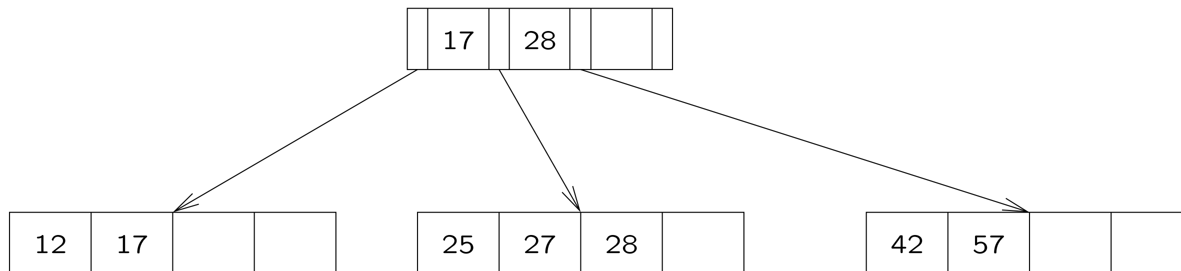
Jos solmu y on täynnä, se pitää halkaista:

- luodaan uusi solmu z
- siirretään solmun y avaimista viimeiset t solmuun z
- lisätään k solmuun y tai z oikeaan kohtaan avainten järjestyksessä
- liitä z solmun y sisareksi sen oikealle puolelle; lisää väliin viitta-arvoksi solmun y suurin avain
- jos solmun y vanhemmalla oli jo $2t$ lasta, jatka edelleen sen halkaisemisella.



lisätään avain 27





Lehden halkaisu kun vanhemmassa on tilaa

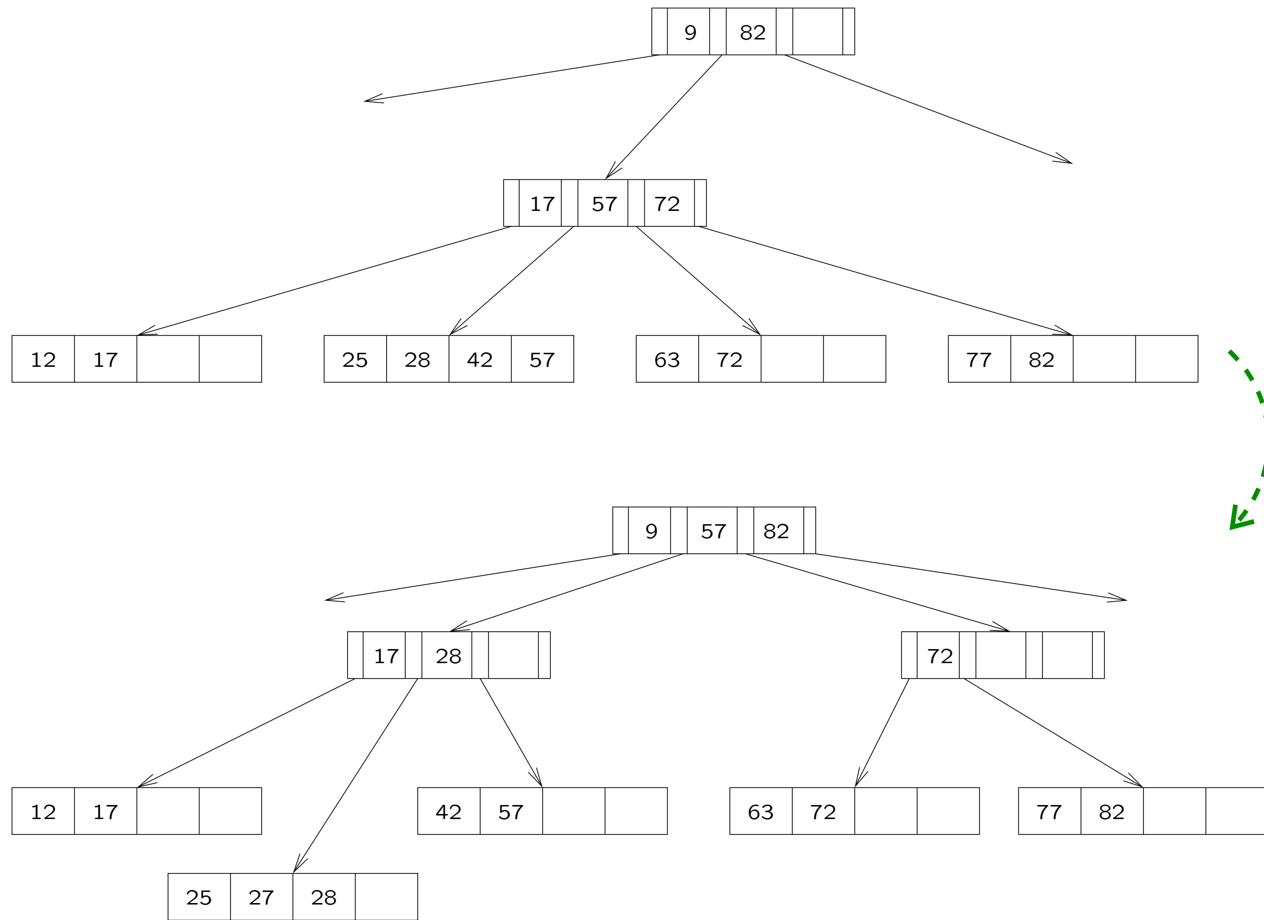
Seuraavalla kalvolla on tilanne, jossa joudutaan halkaisemaan sisäsolmu:

- kuten edellä, lisätään avain 27 täyteen lehteen, ja lehden halkaisussa syntynyt viitta-arvo 28 halutaan sijoittaa lehden vanhempaan
- nyt vanhempi on täynnä ja joudutaan halkaisemaan
- halkaisussa uusia lapsikenttiä syntyy yksi enemmän kuin viitta-arvokenttiä

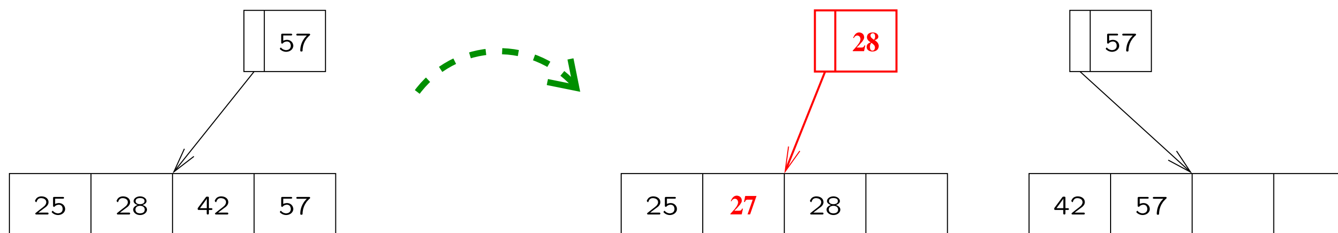
$\Rightarrow$ yhdelle viitta-arvolle (tässä 57) joudutaan etsimään uusi sijoituspaikka:

- se menee jaetun sisäsolmun vanhempaan erottamaan juuri halkaistun solmun puolikkaita

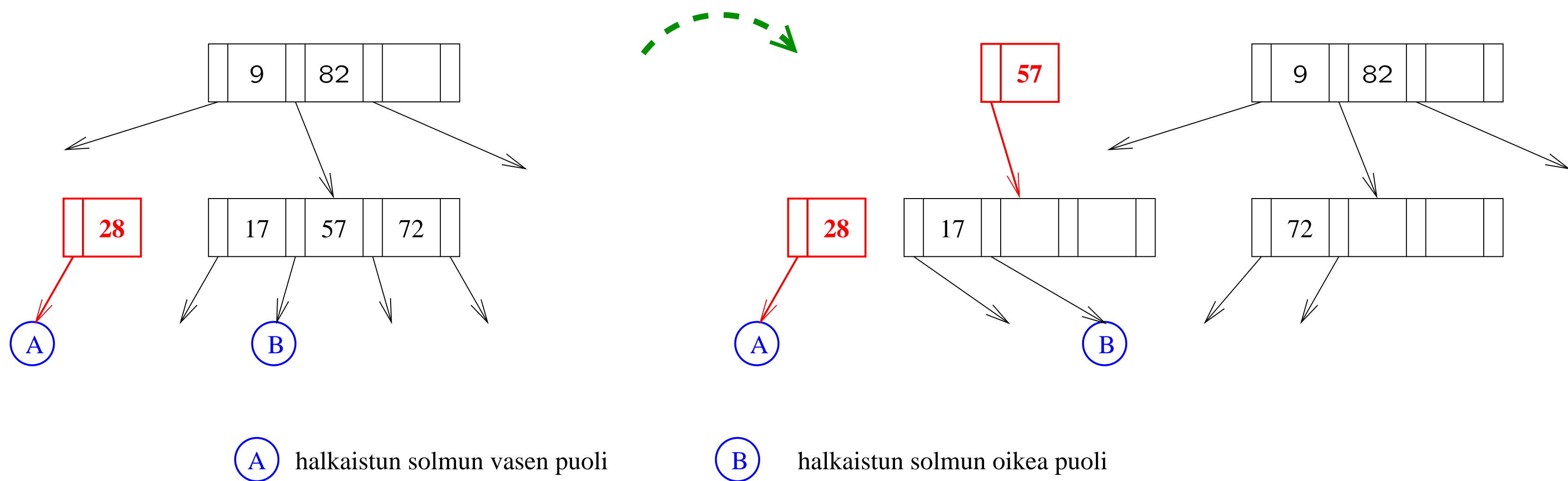
Pahimmassa tapauksessa halkaisuja joudutaan jatkamaan juureen saakka.


9
82
17
57
72
12
17
25
28
42
57
63
72
77
82
9
57
82
17
28
72
12
17
42
57
63
72
77
82
25
27
28

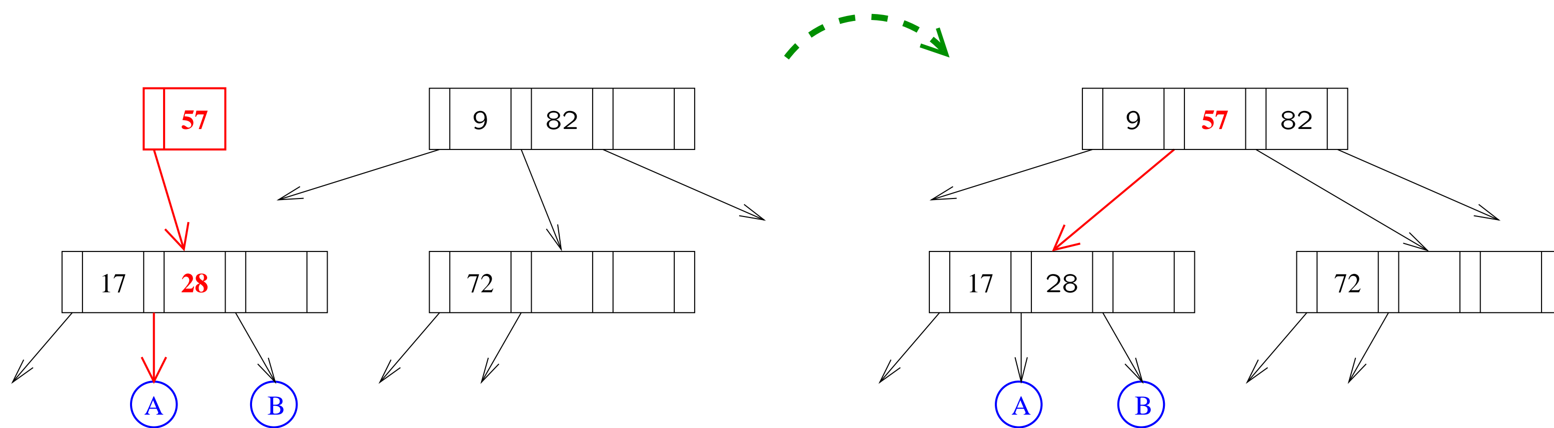
Edellisen kuvan tilanne yksityiskohtaisemmin:



Avain 27 lisätään täyteen lehteen. Lehden halkaisussa syntyy uusi lapsiosoitin ja sen viittaa-arvo. (jatkuu . . .)



Halkaistun lehden vanhempi on täysi, joten se pitää edelleen halkaista. Keskimmäinen viitta-arvo jää lisättäväksi isovanhempaan. (jatkuu . . .)



Laitetaan uuden lehden osoitin ja viitta-arvo paikalleen. Koska isovanhempi ei ollut täysi, proseduuri päättyy sen täydentämiseen. Muuten jatkettaisiin rekursiivisesti isovanhemman halkaisemisella.

Lisäyksen aikavaativuus on oleellisesti sama kuin haun:

- kokonaisajankulutus $O(t \log_t n)$
- levyoperaatioita $O(\log_t n)$.

Tässä kuitenkin levyoperaatioita tulee enemmän kuin yksi per kohdattu solmu, koska muuttuneet solmut pitää myös kirjoittaa levylle. Pahimmillaan joudutaan suorittamaan $\log_t n$ halkaisua ja kirjoittamaan jokaisen halkaistun solmun kumpikin puolisko.

Lisäys voidaan myös tehdä yhdellä läpikäynnillä ylhäältä alas niin, että varmuuden vuoksi halkaistaan kaikki hakupolulla tavatut täydet solmut. Haittana on, että toisinaan tehdään turhia halkaisuja.

Poisto B-puusta

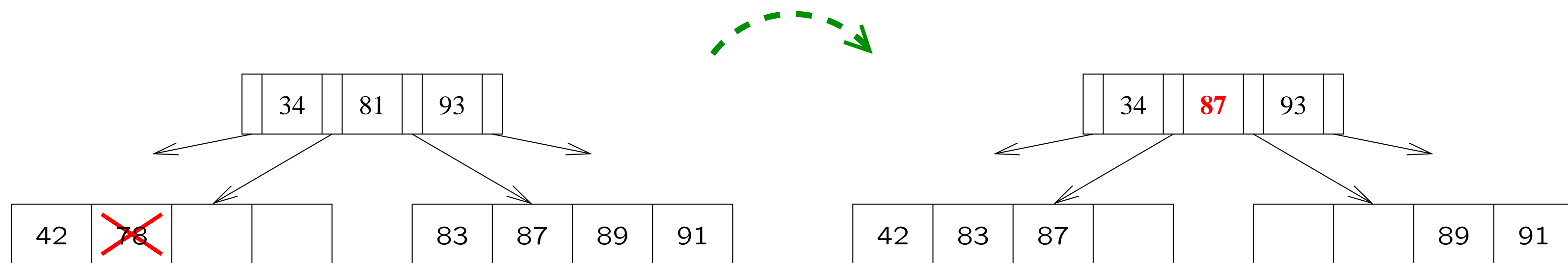
Etsitään lehti x, jossa poistettava avain k sijaitsee, ja poistetaan se. Jos solmussa x on poiston jälkeen vähintään t avainta, ei tarvita jatkotoimia.

Jos solmuun x jää $t-1$ avainta, suoritetaan tasapainotus:

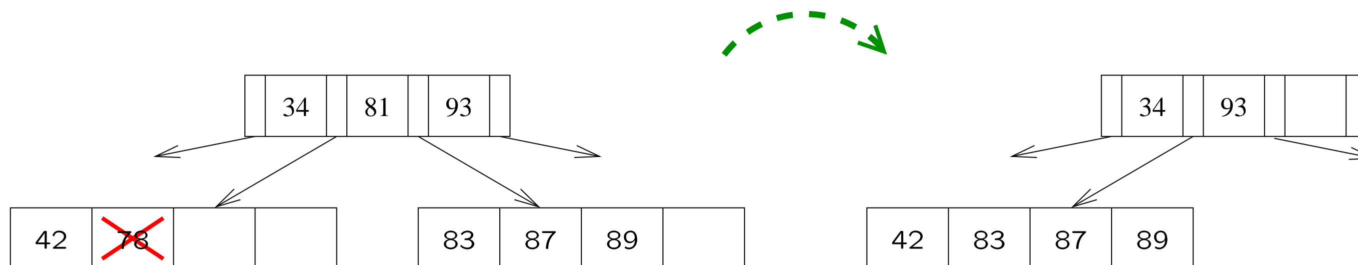
- Jos toisessa vajaan solmun viereisistä sisarista on korkeintaan $t+1$ avainta, tiivistetään näiden kahden solmun avaimet yhteen solmuun. Vanhemman lapsilinkit ja viitta-arvot päivitetään. Lisäksi jos vanhemmalla on nyt vain $t-1$ lasta, suoritetaan edelleen sen tasapainotus.

- Jos vajaan solmun kummallakin viereisellä sisarella on ainakin $t+1$ avainta, jaetaan solmun ja sen sisaren avaimet tasan näiden kahden solmun kesken. Vanhemman viitta-arvot pitää päivittää, mutta muita jatkotoimia ei tarvita.

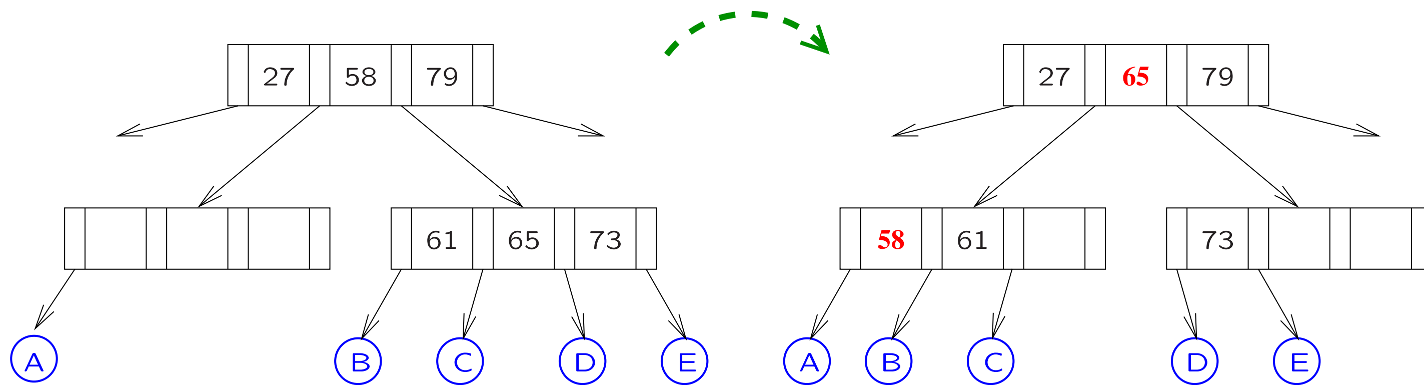
Pahimmassa tapauksssa tasapainotus etenee rekursiivisesti aina juureen saakka. Jos juurelle jää vain yksi lapsi, juuri poistetaan ja puun korkeus pienenee yhdellä.



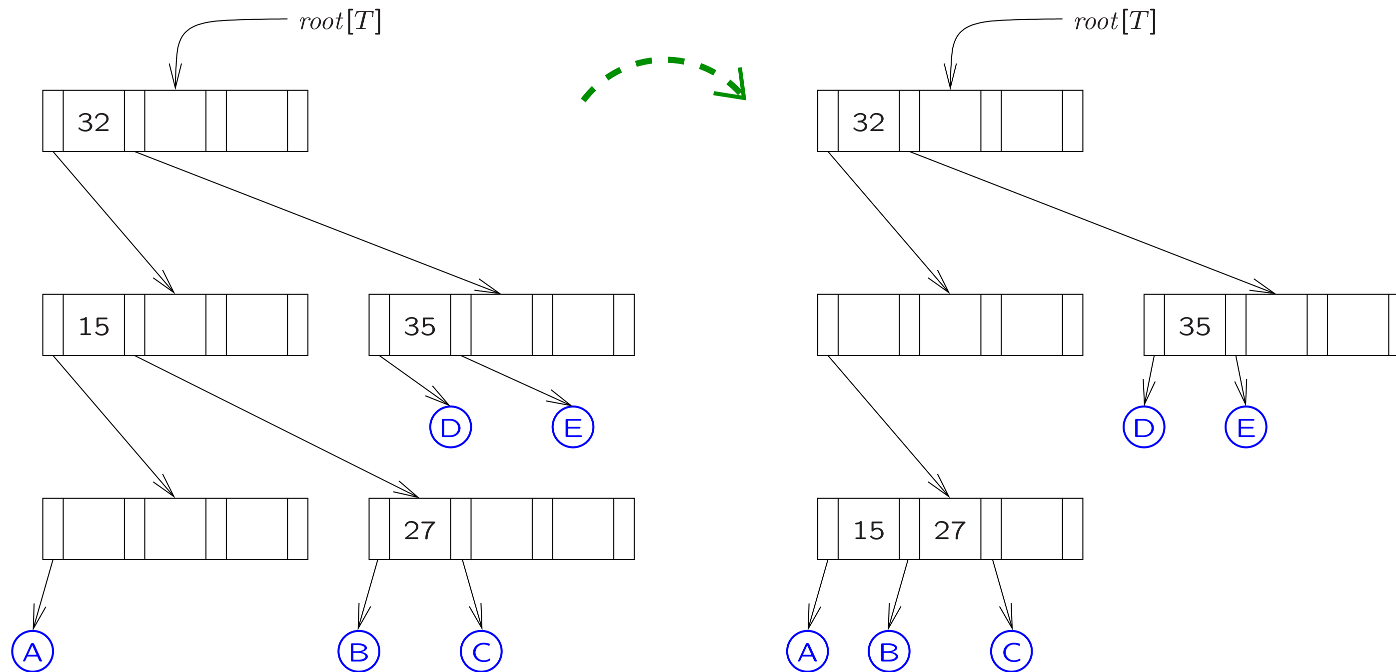
Helpoin tapaus: Kahden lehden avaimet jaetaan tasan. Vanhemman viitta-arvot korjataan. Puun rakenne ei muutu.



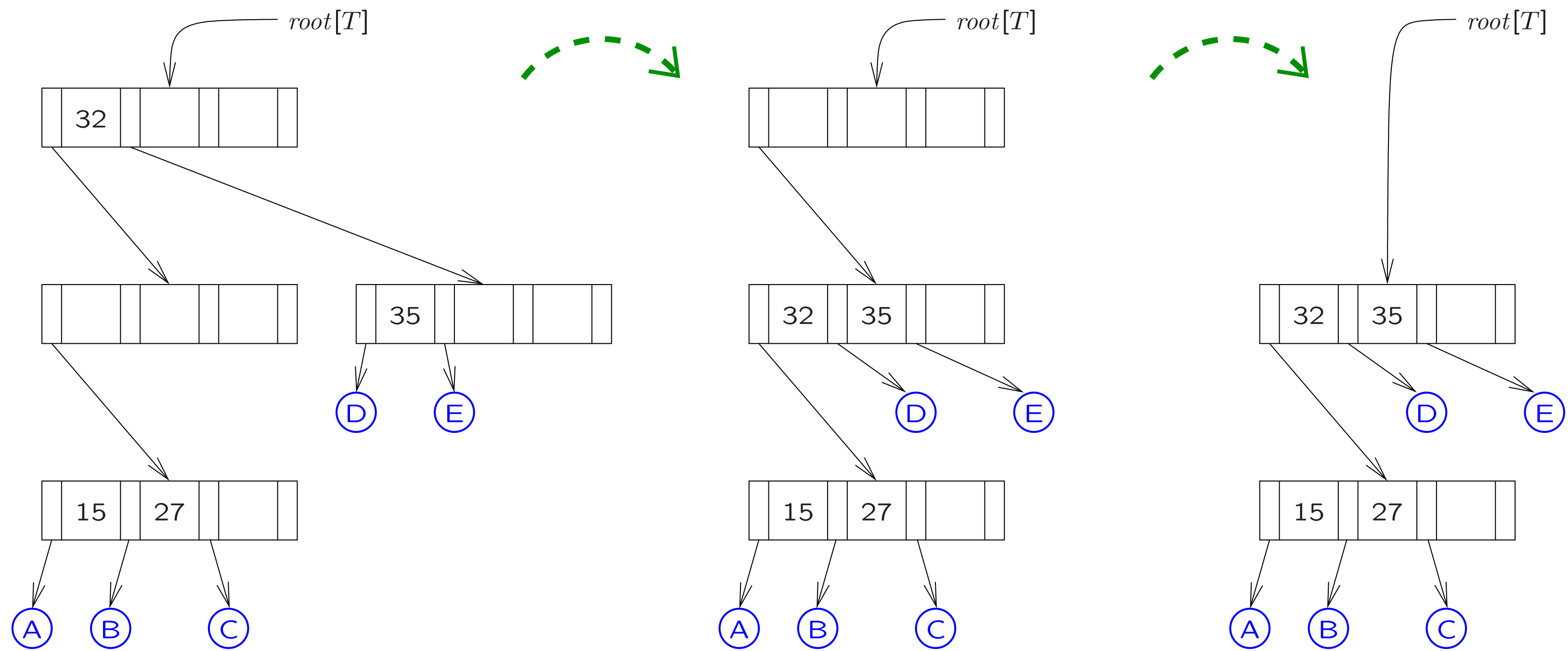
Kaksi sisarta yhdistetään. Vanhemmasta poistuu yksi viitta-arvo ja lapsiosoitin. Tässä esimerkissä vanhemmalla on edelleen riittävästi lapsia, joten proseduuri päättyy tähän.



Jos sisäsolmulta on poistunut lapsia, se voidaan edelleen joutua tasapainottamaan. Tässä esimerkissä tasapainotukseen riittää sisarten viitta-arvojen ja lasten uudelleenjako, ja proseduuri päättyy. Huomaa viitta-arvojen liike vanhemmasta lapseen ja toisin päin.



Myös sisäsolmun tasapainotuksessa voidaan joutua yhdistämään solmuja. Tällöin voi edelleen syntyä tarve tasapainottaa yhdistettyjen solmujen vanhempaa.



Sisäsolmujen tasapainotus etenee rekursiivisesti tarvittaessa aina juureen asti.

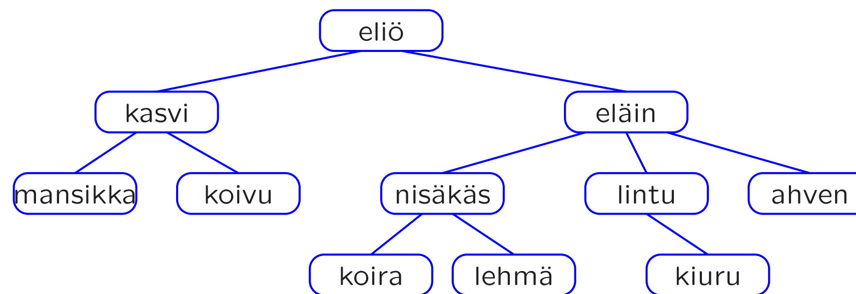
Aikavaativuus on kuten lisäyksellä:

- $O(\log_t n)$ levyoperaatiota
- kaikkiaan laskenta-aika $O(t \log_t n)$.

Myös poisto voidaan toteuttaa yhdellä läpikäynnillä juuresta lähtien: hakupolulla kohdatut tasan puolillaan olevat solmut tasapainotetaan.

Yleisen puurakenteen esittäminen

Puita käytetään yleisesti erilaisten hierarkioiden esittämiseen tietojenkäsittelyssä ja muualla:



Tällöin yhdellä solmulla voi olla mielevaltainen määrä lapsia.

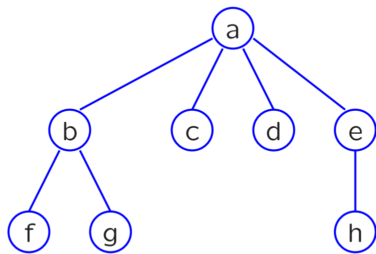
- binääripuussa lapsia voi olla vain kaksi
- B-puussa rakenne sallii mielivaltaisen vakiomäärän lapsia, mutta tilantarve on sama vaikka lapsia olisi vain vähän.

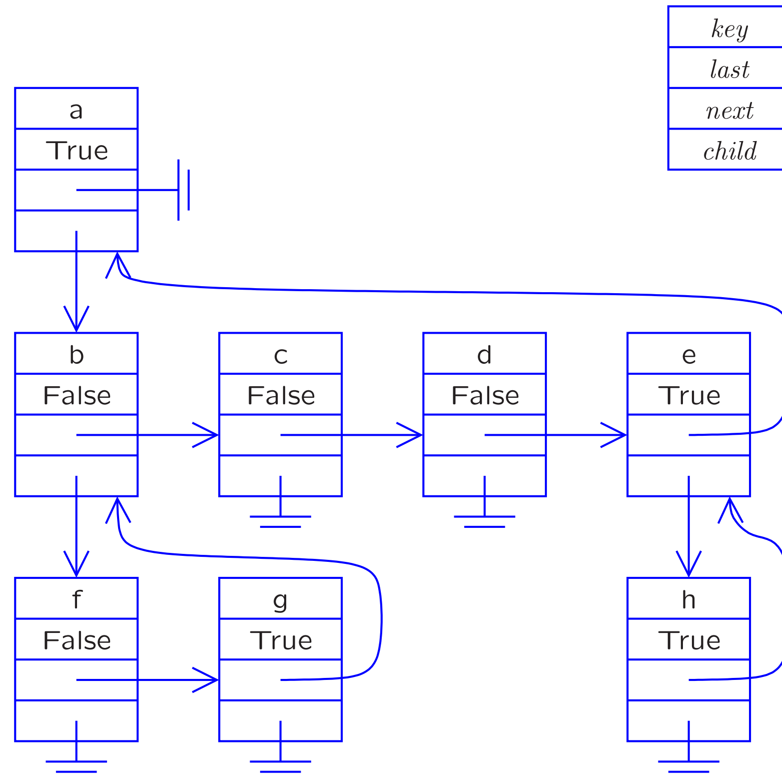
Tarvitsemme paremmin yleisten puiden esittämiseen sopivan rakenteen.

Järjestämme kunkin solmun lapset vasemmalta oikealle. Kuhunkin solmuun tulee seuraavat kentät:

- key: avain kuten ennenkin
- $data$: muu tieto kuten ennenkin
- $last$: True jos solmulla ei ole sisaria oikealla puolella
- $child$: viite vasemmanpuoleisimpaan lapseen; lehtisolmulla Nil
- $next$: jos $last =$ False niin viite seuraavaan sisareen oikealla; muuten viite vanhempaan; juurella Nil

Kuten yleensä, jätämme $data$-kentät huomiotta.





Yleinen puu ja sen talletusrakenne

Vanhemman etsiminen (vaihtoehtoisesti voitaisiin tallettaa joka solmuun p-osoitin):

```
Parent(x)
    while not last[x]
        do x ← next[x]
    return next[x]
```

Ensimmäisen lapsen etsiminen:

```
First-Child(x)
    return child[x]
```

Seuraavan lapsen etsiminen, kun lapsi y on annettu:

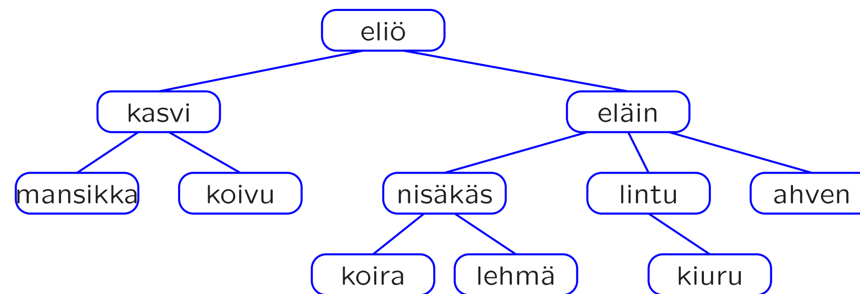
```
Next-Child(y)
    if last[y]
       then return Nil
       else  return next[y]
```

Kuten binääripuu, yleinen puu voidaan käydä läpi esi- tai jälkijärjestyksessaä (sisäjärjestykselle ei ole yhtä luontevaa vastinetta):

```
Preorder-Tree-Walk(x)
    käsittele key[x]
    y ← First-Child[x]
    while y ≠ Nil
        do Preorder-Tree-Walk(y)
           y ← Next-Child(y)

Postorder-Tree-Walk(x)
    y ← First-Child[x]
    while y ≠ Nil
        do Postorder-Tree-Walk(y)
           y ← Next-Child(y)
    käsittele key[x]
```

(Yllä on oletettu $root[T] \neq$ Nil.)



esijärjestys: eliö, kasvi, mansikka, koivu, eläin, nisäkäs, koira, lehmä, lintu, kiuru, ahven

jälkijärjestys: mansikka, koivu, kasvi, koira, lehmä, nisäkäs, kiuru, lintu, ahven, eläin, eliö

Esi- ja jälkijärjestys ovat syvyyssuuntaisia: solmun koko alipuu käydään läpi ennen sen oikeaa sisarta.

Voidaan myös tehdä leveyssuuntainen läpikäynti tasoittain:

```
Levelorder-Tree-Walk(x)
    Q ← tyhjä solmujono
    Enqueue(Q, root[T])
    while not Is-Empty(Q)
        do x ← Dequeue(Q)
           käsittele x
           y ← First-Child(x)
           while y ≠ Nil
               do Enqueue(Q, y)
                  y ← Next-Child[y]
```

Puut ongelmanratkaisussa

Puurakenteella voidaan mallintaa monia etsintäongelmia, joissa asteittain tarkentaen rakennetaan ratkaisu johonkin ongelmaan.

Puun solmut vastaavat osittaisia ratkaisuja:

- Juuressa on alkutilanne, jossa ratkaisua ei vielä ole alettukaan rakentaa.
- Jos solmussa x on osittaisratkaisu r, niin solmun x lapsiin tulee r:n kaikki laajennukset, joissa r:ään on lisätty yksi "ratkaisupalikka".
- Lehtisolmuissa on valmiita ratkaisuehdotuksia, joista näkee suoraan, täyttävätkö ne halutut ehdot.

Sovelluksesta riippuen tämä voi käytännössä tarkoittaa hyvin eri asioita.

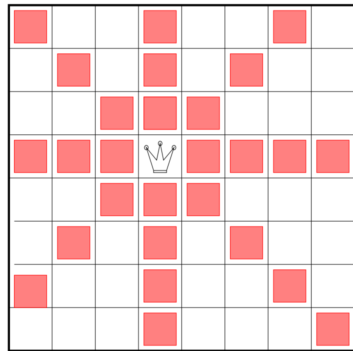
Yleisiä perusideoita:

- Puun läpikäyminen vastaa loogisesti kaikkien mahdollisten ratkaisujen kokeilemista.
- Puu on olemassa vain implisiittisesti:
  - sitä ei talleteta puutietorakenteena (mikä veisi kohtuuttomasti muistia)
  - tyypillisesti puu käydään läpi syvyyssuuntaisesti, jolloin muistissa on kerrallaan yksi aktiivinen solmu ja siihen johtava polku juuresta
  - toisinaan läpikäynti pitää tehdä leveyssuuntaisesti, jolloin muistissa on puun yksi taso kerrallaan, mutta tämä vie huomattavasti enemmän muistia
- tehokkuutta parannetaan karsimalla ennakoivasti haarat, joista ei voi tulla hyvää ratkaisua (branch-and-bound, α-β-karsinta)
- . . . mutta algoritmit ovat silti pohjimmiltaan raakaan voimaan perustuvia ja vaativat paljon aikaa.

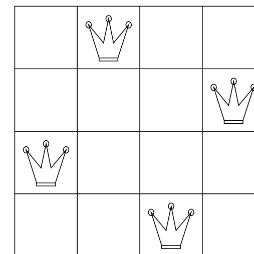
Kahdeksan kuningattaren ongelma

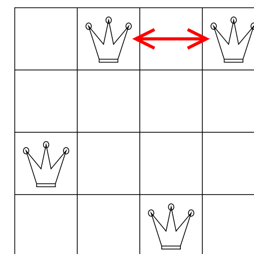
Tämä on klassinen esimerkki edellisen idean soveltamisesta.

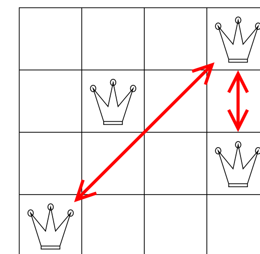
Tehtävänä on sijoittaa 8×8 -shakkilaudalle kahdeksan kuningatarta siten, että mitkään kaksi eivät uhkaa toisiaan. Tehtävä yleistyy luonnollisesti $n \times n$ -laudalle.



Kuningatar uhkaa kaikkia nappuloita, jotka ovat samalla rivillä, sarakkeella tai diagonaalilla (varjostetut ruudut).







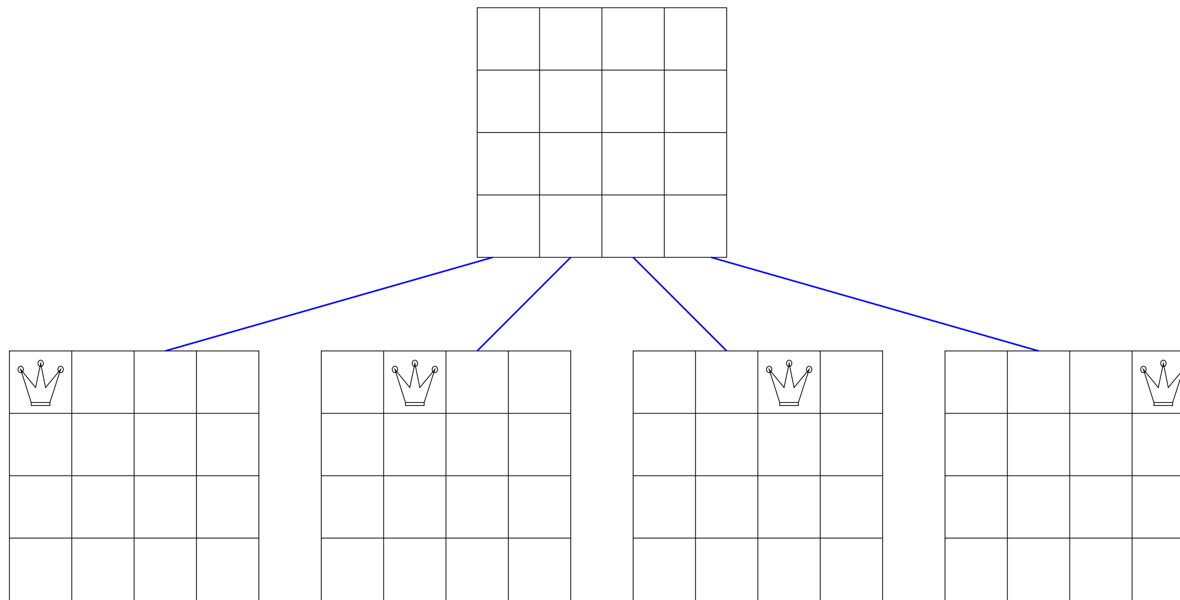
Tapaukselle $n = 4$ yksi oikea ratkaisu ja kaksi virheellistä ratkaisua (tai täsmällisemmin "ei-ratkaisua").

Sovelletaan edellisiä periaatteita tähän:

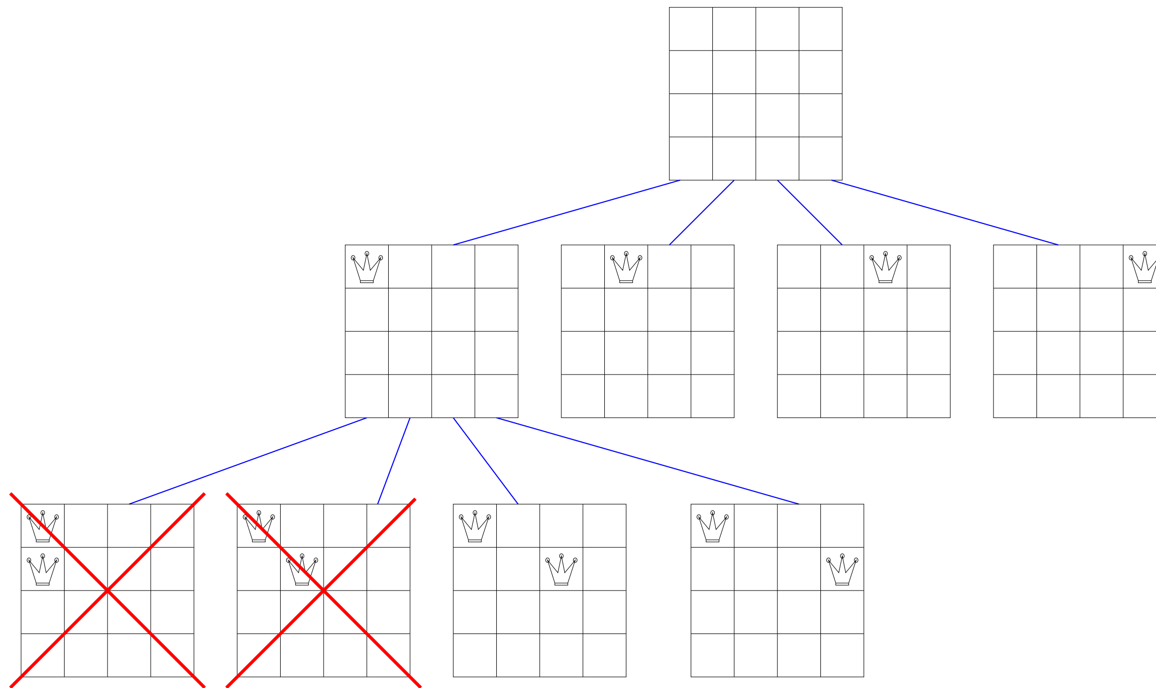
- Osittaisessa ratkaisussa laudalla on sijoitettu $0 \leq k \leq n$ kuningatarta.
- Alkutilanne on tyhjä lauta ($k = 0$).
- Osittainen ratkaisu on laillinen, jos mitkään sijoitetut kaksi kuningatarta eivät uhkaa toisiaan.
- Jos $k = n$, ratkaisu on valmis; se on oikea ratkaisu ongelmaan, jos se on laillinen.

Jätämme suoraan pois ne laittomat ratkaisut, joissa kaksi kuningatarta on samalla rivillä. Tämän yksinkertaistuksen jälkeen

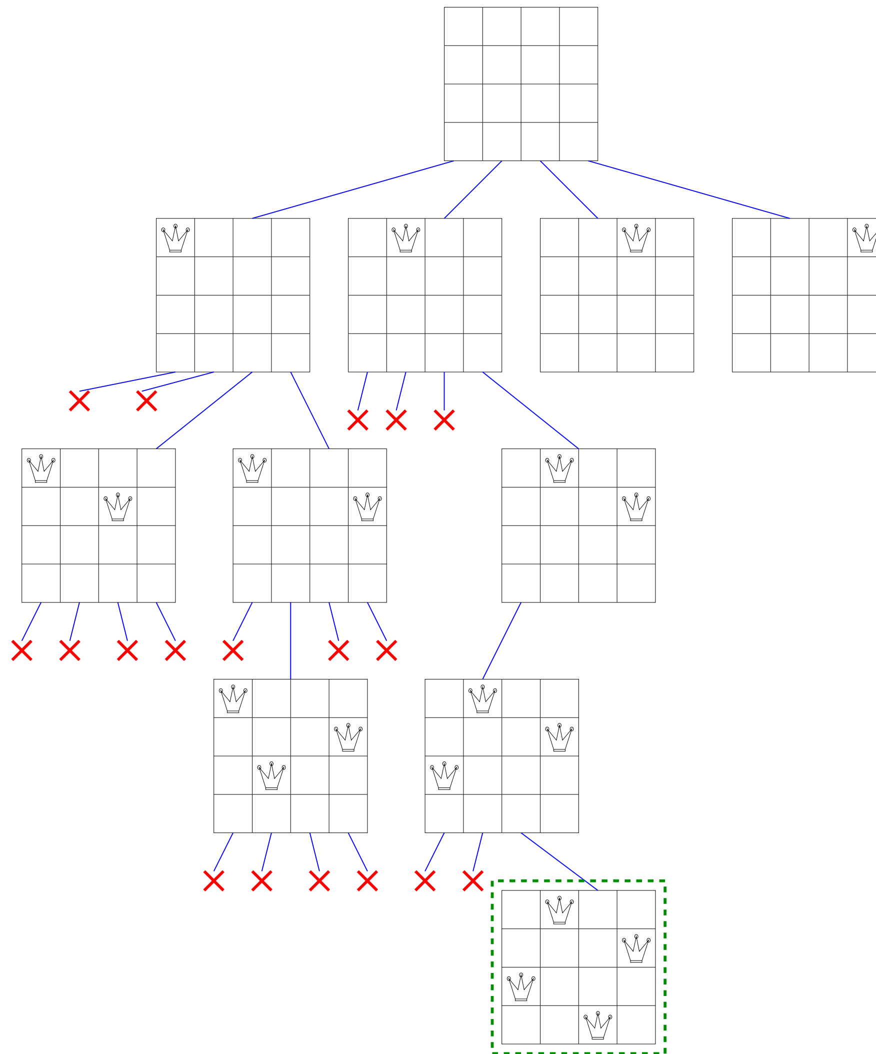
- Osittaisessa ratkaisussa riveille $1, \ldots, k$ on kullekin sijoitettu tasan yksi kuningatar jollain $0 \leq k \leq n$.
- Osittaista ratkaisua laajennetaan sijoittamalla yksi kuningatar seuraavalle tyhjälle riville.



Juuri ja ensimmäisen laajennuksen vaihtoehdot ($n = 4$).



Ensimmäisen laajennusvaihtoehdon seuraavat jatkolaajennukset, joista kaksi voidaan suoraan karsia laittomina.



Osaratkaisujen laajentamista on jatkettu, kunnes on löydetty täydellinen laillinen ratkaisu.

Tällaista implisiittisen puun läpikäyntiä kutsutaan peruuttavaksi etsinnäksi (backtracking): umpikujaan jouduttaessa palataan puussa sellaiseen ylempään solmuun, johon vielä liittyy kokeilemattomia vaihtoehtoja.

Tähän läheisesti liittyvä tekniikka on branch-and-bound, jossa osa haaroista karsitaan jo ennen niiden läpikäyntiä.

Tässä karsimisehto on melko triviaali: karsitaan jos osaratkaisu on jo valmiiksi laiton. Termi "bound" viittaa lähinnä optimointiongelmiin (kuten jäljempänä esitettävä kauppamatkustajan ongelma), joissa tilanne on yleensä hienovaraisempi.

Esitetään kuningatarongelman ratkaisu täsmällisemmin. Valitaan osaratkaisun esitykseksi totuuarvotaulukko $\mathit{table}[1\,..\,n, 1\,..\,n]$, missä $\mathit{table}[r, s] = \text{True}$ tarkoitaa, että rivin r sarakkeeseen s on sijoitettu kuningatar.

Oletetaan annetuksi apufunktio check, jolle

$$\text{check}(\mathit{table}) = \begin{cases} \text{False} & \text{jos } \mathit{table} \text{ sisältää kaksi toisiaan uhkaavaa kuningatarta} \\ \text{True} & \text{muuten.} \end{cases}$$

Siis erityisesti jos table sisältää n kuningatarta, jotka eivät uhkaa toisiaan, niin $\text{check}(\mathit{table}) = \text{True}$.

Seuraava algoritmi etsii mahdolliset laajennukset osaratkaisulle $table$, jossa on jo $r-1$ kuningatarta:

```
Put-Queen(table, r)
    if not check(table)
       then return False              ▷ osaratkaisu jo valmiiksi laiton
    if r = n + 1
       then print(table)
            return True
    for s ← 1 to n
        do table2 ← table             ▷ erillinen kopio lähtötilanteesta
           table2[r, s] ← True
           if Put-Queen(table2, r + 1)
              then return True
    return False                      ▷ kaikki vaihtoehdot kokeiltu
```

Ratkaiseminen käynnistyy kutsulla Put-Queen($table0, 1$), missä $table0$ sisältää $n \times n$ False-arvoa.

Puuta voidaan karsia tehokkaammin korvaamalla check jollain testillä, joka palauttaa False useammin, kunhan seuraava ehto pysyy voimassa:

> jos check($table$) = False, niin osaratkaisua $table$ ei voi täydentää kelvolliseksi ratkaisuksi lisäämällä nolla tai useampia kuningattaria.

Siis ei saa karsia haaroja, joissa on kelvollisia ratkaisuja.

Ongelmasta riippuu, onko olemassa voimakkaita karsintakriteerejä, jotka kuitenkin voidaan testata nopeasti.

Algoritmin vaativuutta on vaikea analysoida tarkasti, koska karsinnan tehokkuutta ei tiedetä.

Karkeasti arvioiden edellä esitetyn puun koko on

$$1 + n + n^2 + \ldots + n^n = O(n^{n+1})$$

solmua. Kerralla kuitenkin pidetään muistissa vain yhtä juuresta lähtevää polkua. Koska yhden solmun koko on tässä $O(n^2)$, algoritmin muistintarve on $O(n^3)$.

Yhden solmun käsittely vie ajan $O(n^3)$, joten kaikkiaan aikavaativuudelle saadaan karkea yläraja

$$O(n^3 \cdot n^{n+1}) = O(n^{n+4}).$$

Edellä esitettyä algoritmia on helppo jonkin verran optimoida, mutta joka tapauksessa tällä lähestymistavalla aikavaativuus on parametrin n suhteen eksponentiaalinen.

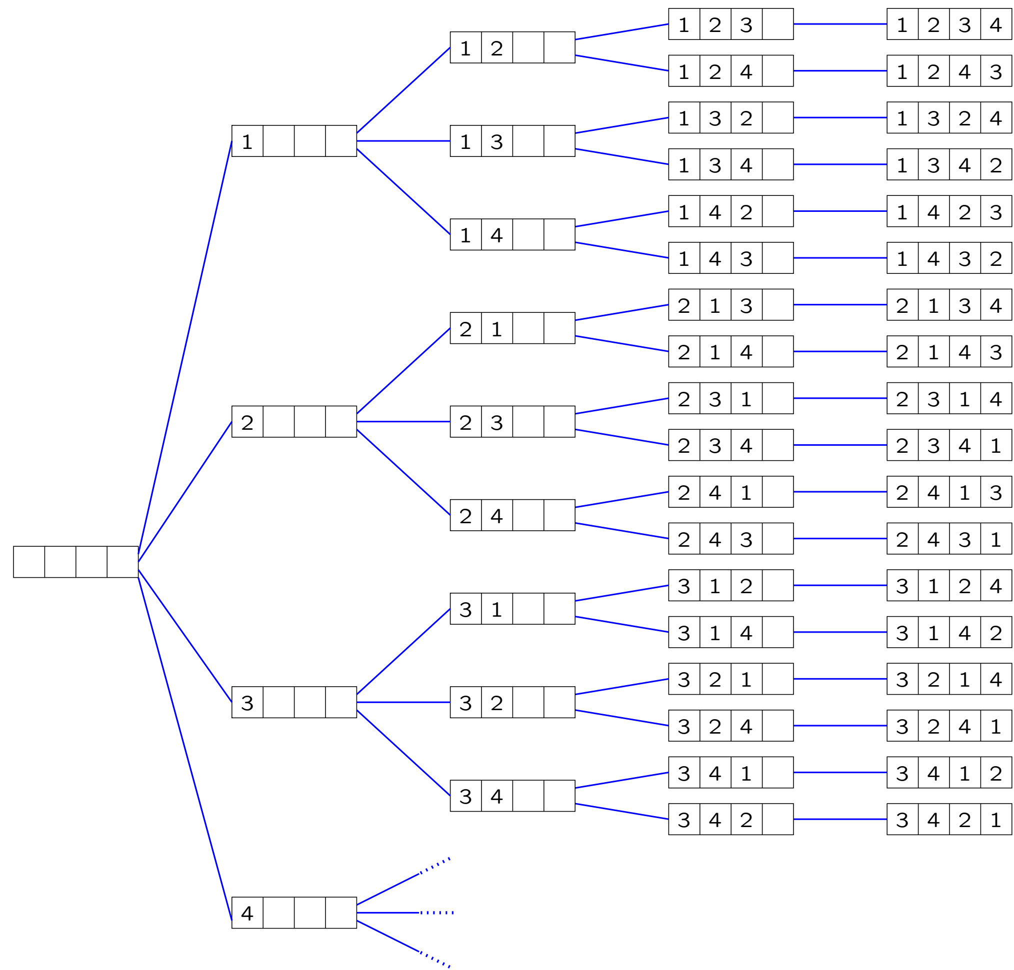
Permutaatioiden tuottaminen

Halutaan tuottaa kaikki lukujen $1, \ldots, n$ permutaatiot (järjestykset), joita siis on $n!$ kappaletta. Toimitaan seuraavasti:

- Valitaan ensimmäiseksi alkioksi 1. Tuotetaan rekursiivisesti kaikki lukujen $2, \ldots, n$ permutaatiot ja liitetään niistä jokaisen eteen 1.
- Valitaan ensimmäiseksi alkioksi 2. Tuotetaan rekursiivisesti kaikki lukujen $1, 3, \ldots, n$ permutaatiot ja liitetään niistä jokaisen eteen 2.
- Valitaan ensimmäiseksi alkioksi 3. Tuotetaan rekursiivisesti kaikki lukujen $1, 2, 4, \ldots, n$ permutaatiot ja liitetään niistä jokaisen eteen 3.
- $\ldots$
- Valitaan ensimmäiseksi alkioksi n. Tuotetaan rekursiivisesti kaikki lukujen $1, \ldots, n-1$ permutaatiot ja liitetään niistä jokaisen eteen n.

Menettely vastaa seuraavan sivun puun kaikkien lehtien läpikäyntiä järjestyksessä.

Osa joukon $\{1, 2, 3, 4\}$ permutaatioita vastaava puuta (juuri vasemmalla, lehdet oikealla).



Seuraava algoritmi tulostaa kaikki sellaiset joukon $\{1, \ldots, n\}$ permutaatiot, joissa ensimmäiset $k-1$ lukua ovat osataulukon $table[1 \ldots k-1]$ mukaiset. Lisäksi oletetaan, että $used[i] =$ True, jos ja vain jos luku i esiintyy osataulukossa $table[1 \ldots k-1]$.

```
Generate(table, used, k)
    if k = n + 1
       then print(table)
       else  for i ← 1 to n
                do if not used[i]
                      then used2 ← used
                           used2[i] ← True
                           table2 ← table
                           table2[k] ← i
                           Generate(table2, used2, k + 1)
```

Toiminta käynnistyy kutsulla generate($table, used, 1$), missä $used[i] =$ False kaikilla i.

Kaikki lehdet halutaan siis käydä läpi, joten karsinta ei ole mahdollista.

Solmujen lukumäärä on taas $O(1 + n + n^2 + \ldots + n^n) = O(n^n)$.

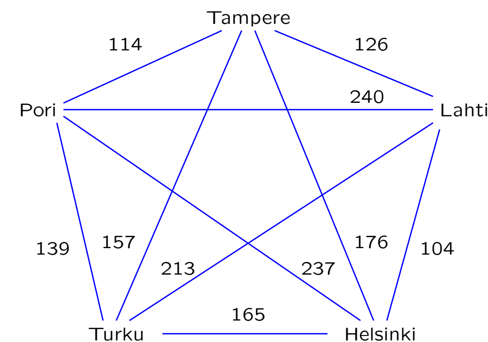
Yhden solmun käsittely vie ajan $O(n)$, joten kokonaisaika on $O(n^{n+1})$.

Muistissa on taas korkeintaan n solmua kerrallaan, joten tilavaativuudeksi tulee $O(n^2)$.

Kauppamatkustajan ongelma

On annettu joukko kaupunkeja ja niiden maantie-etäisyydet.

Tehtävänä on käydä kerran jokaisessa kaupungissa ja palata lähtökaupunkiin.



Sallittuja ratkaisuja ovat kaikki kaupunkien permutaatiot. (Viimeisestä kaupungista palataan suorinta tietä lähtökaupunkiin.)

Optimaalinen ratkaisu on sallituista ratkaisuista se, jonka kokonaispituus on pienin (ml. viimeinen osuus takaisin lähtökaupunkiin).

Kyseessä on erikoistapaus kauppamatkustajan ongelmasta (Travelling Salesman Problem, TSP).

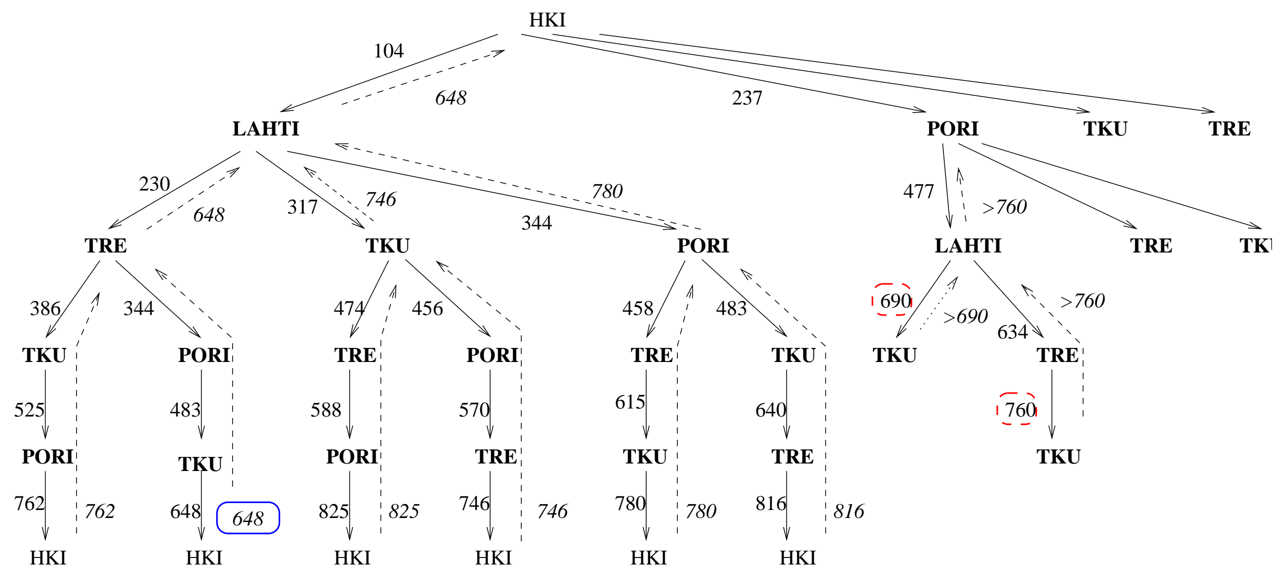
Naiivi ratkaisu:

- Käydään läpi kaikki reitit (permutaatiot) kuten edellä.
- Lasketaan jokaisen reitin pituus.
- Valitaan pienin pituus.

Hieman kehittyneempi ratkaisu:

- Pidetään kirjaa parhaasta tähän mennessä saavutetusta reitin pituudesta (aluksi $+\infty$).
- Reittiä tuotettaessa pidetään kirjaa tähänastisen osareitin pituudesta.
- Jos osareitin pituus ylittää parhaan pituuden, karsitaan osareitti.

Esim. jos on jo löydetty valmis reitti Helsinki–Lahti–Tampere–Pori–Turku–Helsinki (kaikkiaan 648 km), ei kannata miettiä jatkoa osareitille Helsinki–Pori–Lahti–Turku–? (*ainakin* 690 km).



Osa kauppamatkustajan puun läpikäynnistä. Huomaa lyhimmän löytyneen reitin pituus 648 ja karsitut tätä pidemmät reitit.

```
gen(best, length, visited, cur, k)
    if k = n
       then return length + dist[cur, 1]
    mybest ← +∞
    for i ← 2 to n
         do if not visited[i]
               then vis2 ← visited
                    vis2[i] ← True
                    if length + dist[cur, i] < best
                       then newp ← gen(best,
                                       length + dist[cur, i]),
                                       vis2, i, k + 1)
                            if newp < mybest
                               then mybest ← newp
                            if newp < best
                               then best ← newp
    return mybest
```

Haku käynnistyy kutsulla gen($+\infty, 0, vis, 1, 1$), missä $vis[i]$ = False kaikilla i.

Kaupungit on numeroitu $1, \ldots, n$; lähtökaupunki on 1

$dist[i,j]$: kaupunkien i ja j etäisyys
$visited[i]$: True jos nykyinen osareitti jo käy kaupungissa i
$best$: kaikkiaan parhaan löydetyn pituus
$mybest$: parhaan tälle osareitille löydetyn jatkon pituus
cur: nykyisen osareitin viimeisin kaupunki
k: nykyisen osareitin kaupunkien lkm.
$length$: nykyisen osareitin pituus tähän asti

Tilavaativuus yhdelle solmulle (= aliohjelmakutsun aktivaatiotietueelle) on $O(n)$. Siis kaikkiaan tilavaativuus on $O(n^2)$.

Aikavaativuus yhtä solmua kohti on $O(n^2)$. Solmujen lukumäärästä on vaikea sanoa muuta, kuin että se on taas $O(n^n)$. Käytännössä karsinta pienentää sitä jonkin verran. Joka tapauksessa aikavaativuus on kaupunkien lukumäärän suhteen eksponentiaalinen.

Huomautuksia:

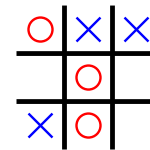
- Kauppamatkustajan ongelma on NP-täydellinen; polynomisessa ajassa toimivaa algoritmia ei tunneta, ja sellaisen löytymistä pidettäisiin yllättävänä. (Lisää kurssilla *Laskennan vaativuus*.)

- Ongelma ja sen muunnelmat ovat sovelluksissa tärkeitä ja edellistä tehokkaampia branch-and-bound-menetelmiä tunnetaan.

- Sovelluksissa käytetään myös heuristisia algoritmeja: saadut ratkaisut ovat yleensä kelvollisia, mutta eivät optimaalisia, ja mitään takuita ei ole.

Pelipuu

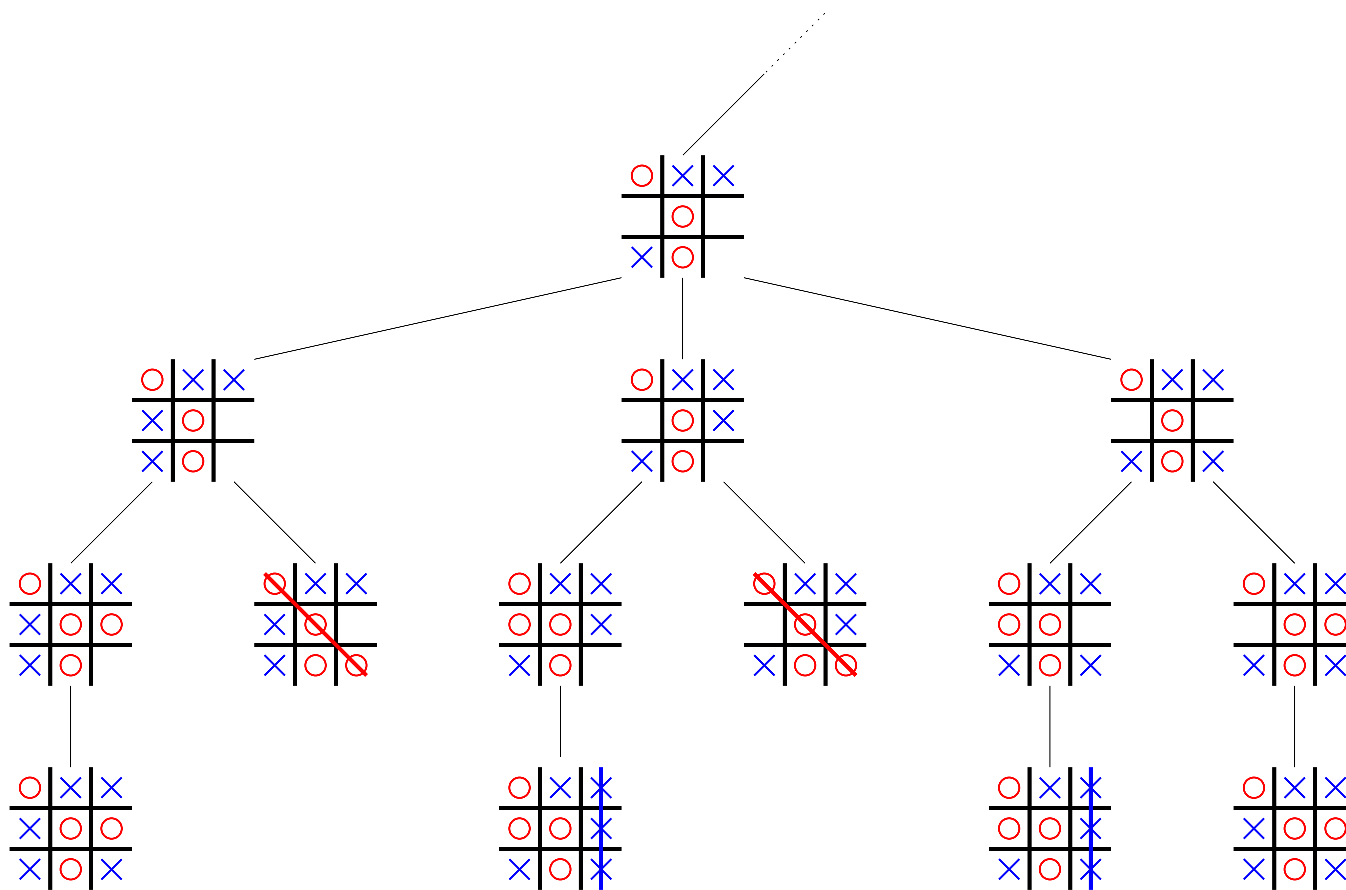
Perinteinen ristinolla: Pelimerkkejä asetetaan vuorotellen 3×3 -ruudukkoon, kolme peräkkäin voittaa. Jos kumpikaan ei saa kolmea peräkkäin, peli päättyy tasan.



Kysymys: Mikä on paras siirto annetussa tilanteessa?

Oletus: Vastustaja tekee aina parhaan siirtonsa.

Perusidea on muodostaa (taas vain implisiittisesti) puu, jossa on kaikki mahdolliset jatkot annetusta tilanteesta.



Osa ristinollan pelipuusta

Tarkastellaan tilannetta pelaajan risti kannalta. Ristillä on annetussa tilanteessa voittostrategia, jos oikein pelaten risti voittaa, teki vastustaja mitä tahansa.

Siis ristillä on voittostrategia, jos

- ristillä on jo kolme peräkkäin (eli hän on jo voittanut); tai
- on ristin vuoro, ja ristillä on edelleen voittostrategia jonkin oman siirtonsa jälkeen; tai
- on nollan vuoro, ja ristillä on edelleen voittostrategia kaikkien mahdollisten nollan siirtojen jälkeen.

Vastaavasti määritellään nollan voittostrategia.

Määritellään pelin loppuasemille arvot (payoff) ristin kannalta:

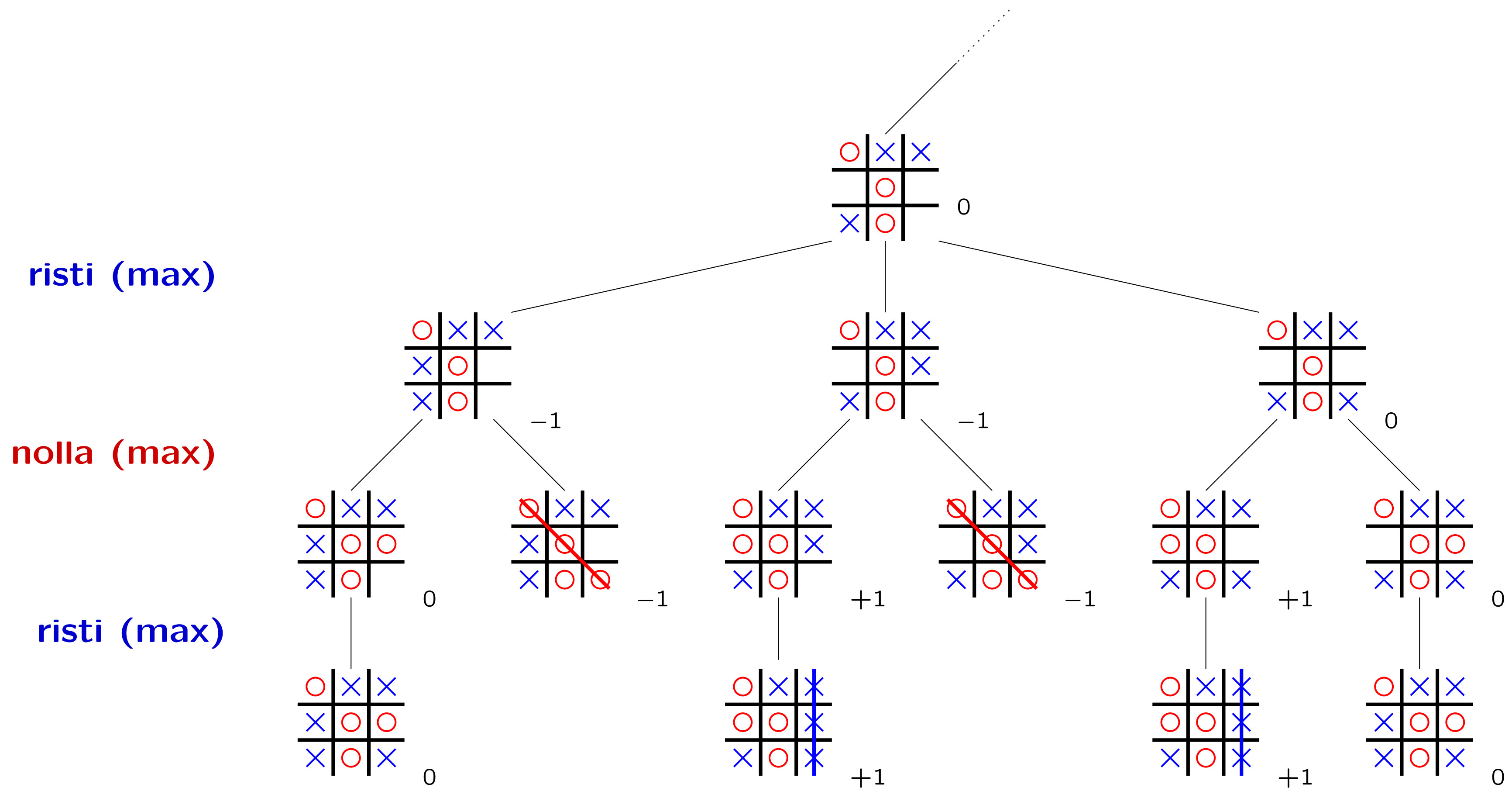
$+1$:	risti on voittanut
0:	peli on päättynyt tasan
-1:	nolla on voittanut.

Yleistetään arvot myös muille kuin lopputilanteille:

$+1$:	ristillä on voittostrategia
0:	kummallakaan ei ole voittostrategiaa
-1:	nollalla on voittostrategia.

Edellisen mukaan

- jos asemassa on ristin vuoro, niin aseman arvo on suurin seuraaja-asemien arvoista
- jos asemassa on nollan vuoro, niin aseman arvo on pienin seuraaja-asemien arvoista.



Pelipuun solmujen arvot

Saadaan seuraava minimax-algoritmi aseman v arvon laskemiseen. Jos maksimiin pyrkivä risti on vuorossa, kutsutaan Eval-Max(v). Oikea siirto on se, jonka jälkeinen asema w tuotti suurimman arvon $newval$.

Jos taas minimiin pyrkivä nolla on vuorossa, kutsutaan Eval-Min(v).

```
Eval-Max(v)
    if asemassa v kolme ristiä peräkkäin
       then return +1
    elseif asemassa v kolme nollaa peräkkäin
       then return −1
    elseif asemassa v lauta täynnä
       then return 0
    else mybest ← −∞
           for aseman v seuraajille w
               do newval ← Eval-Min(w)
                  if newval > mybest
                     then mybest ← newval
    return mybest
```

```
Eval-Min(v)
    if asemassa v kolme ristiä peräkkäin
       then return +1
    elseif asemassa v kolme nollaa peräkkäin
       then return −1
    elseif asemassa v lauta täynnä
       then return 0
    else mybest ← +∞
           for aseman v seuraajille w
               do newval ← Eval-Max(w)
                  if newval < mybest
                     then mybest ← newval
    return mybest
```

Realistisemmissa peleissä kuten shakki

- pelipuuta karsitaan α-β-karsinnalla (hieman kuin branch-and-bound)
- pelipuuta ei voi muodostaa lehtiin saakka, vaan rekursio lopetetaan "sopivalla" syvyydellä ja asemaan sovelletaan evaluointifunktiota (nappuloiden määrä jne.)
- käytetään avaus- ja loppupelikirjastoja ym.

Joissain peleissä tietokone voittaa ihmisen helposti, joissain koneet ovat vielä kaukana.

Minimax voidaan yleistää satunnaisuutta sisältäville peleille kuten backgammon laittamalla min- ja max-solmujen väliin odotusarvosolmuja esittämään nopanheittoa.

4. Hajautus

Hajautus (hashing) on vaihtoehto tasapainoisille puille dynaamisen joukon toteuttamisessa:

- Search, Insert ja Delete yleensä ajassa $O(1)$ (tasapainoisella puulla $O(\log n)$)
- pahimmassa tapauksessa Search jne. voivat viedä $\Omega(n)$
- hyvän hajautusfunktion löytäminen voi vaatia hieman kokeilua, mutta muuten menetelmä on helppo toteuttaa
- avainten järjestykseen liittyviä operaatioita Succ, Pred, Min ja Max ei erityisemmin tueta (aika $O(n)$, kun tasapainoisella puulla $O(\log n)$).

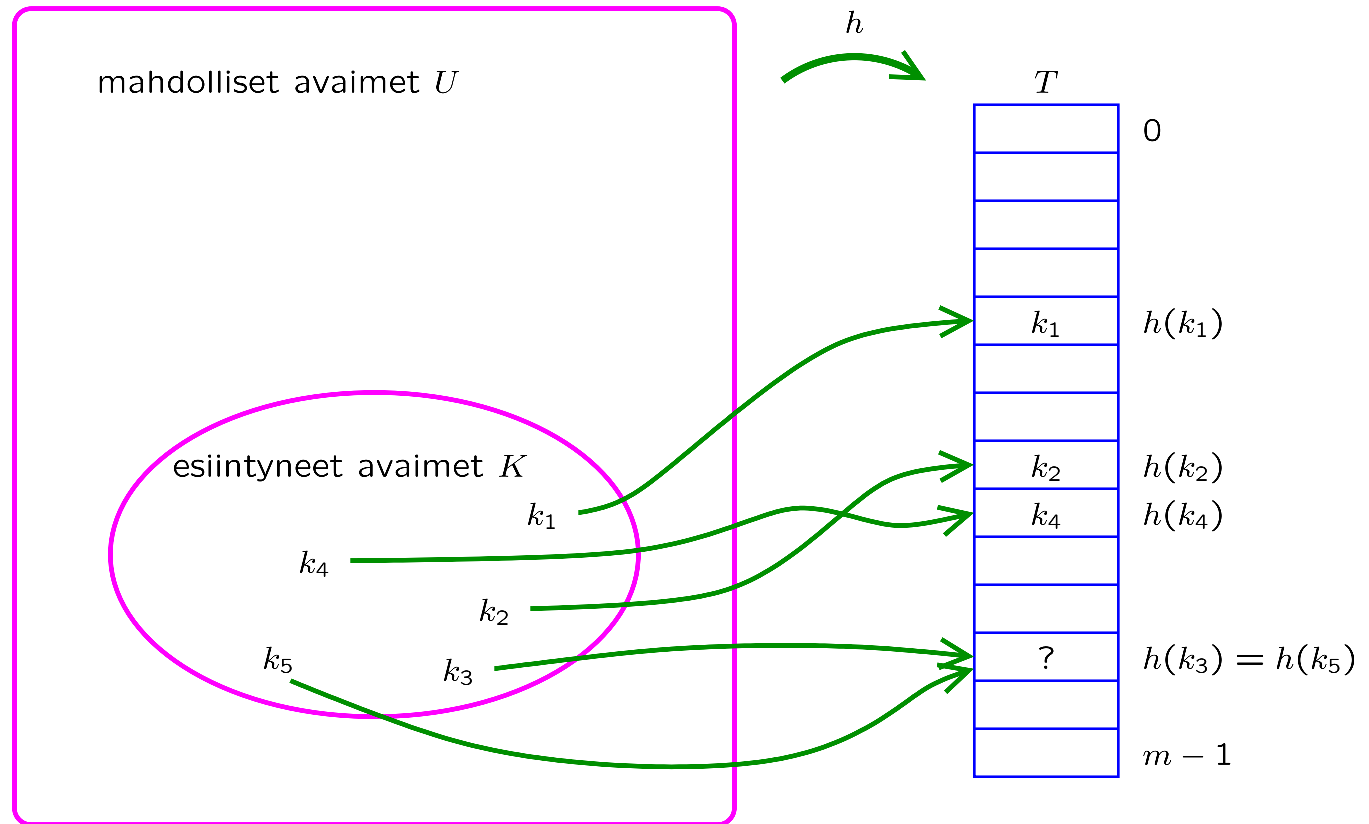
Hajautuksen perusidea

Oletetaan:

- kaikkien mahdollisten avainten joukko U on erittäin suuri (jopa ääretön)
  Esim. $U =$ kaikki alle 50 merkin merkkijonot
- sovelluksessa todella esiintyvien avainten joukko K on hyvin paljon pienempi kuin U (mutta silti suuri)
  Esim. $K =$ kaikkien helsinkiläisten nimet
- halutaan tallettaa avaimet taulukkoon $T[0 \,..\, m-1]$, jonka koko m on vain hieman suurempi kuin $n = |\, K \,|$
  Esim. $m = 1$ miljoona

Perusratkaisu: valitaan hajautusfunktio $h\colon U \to \{\, 0, \ldots, m-1 \,\}$ ja talletetaan avain k paikkaan $T[h(k)]$.

Perusongelma: väistämättä monille avainpareille $k_1 \neq k_2$ pätee $h(k_1) = h(k_2)$ (yhteentörmäys).


mahdolliset avaimet U
esiintyneet avaimet K
k_1
k_4
k_2
k_5
k_3
h
T
0
k_1
$h(k_1)$
k_2
$h(k_2)$
k_4
$h(k_4)$
?
$h(k_3) = h(k_5)$
$m - 1$

Peruskysymyksiä, jotka voidaan ratkaista eri tavoin:

Mikä on hyvä talletusalueen koko m? Kun merkitään talletettavien avainten lukumäärää $n = |S|$, niin keskeinen suure on täyttöaste (loading factor) $\alpha = n/m$.

Miten valitaan hajautusfunktio? Yleensä tämä jaetaan kahteen osaan:

- ensin valitaan funktio $g: U \to \mathbb{N}$, joka kuvaa avaimet luonnollisiksi luvuiksi
- sitten valitaan funktio $h: \mathbb{N} \to \{0, \ldots, m-1\}$, jolloin avaimen $k \in U$ talletuspaikka on $T[h(g(k))]$.

Jatkossa yleensä ajatellaan, että avaimet ovat valmiiksi luonnollisia lukuja. Palaamme ensimmäiseen osakysymykseen erikseen.

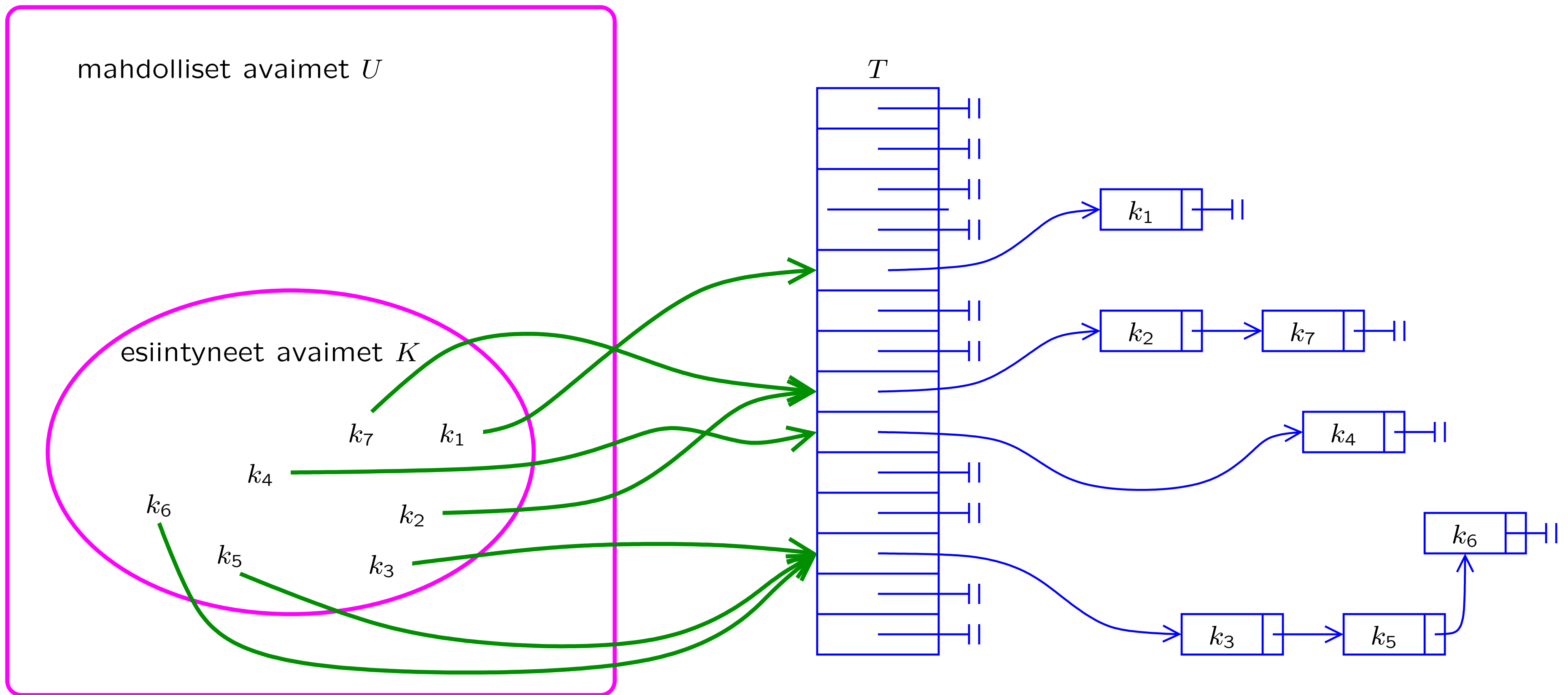
Miten yhteentörmäykset käsitellään? Yhteentörmäyksiä tulee väistämättä, sitä enemmän mitä suurempi α.

Ylivuotoketjut

Ketjuttaminen (chaining) on perustapa yhteentörmäysten käsittelemiseen:

- periaatteessa mielivaltaisen monella avaimella k voi olla sama hajautusfunktion arvo $h(k) = i$
- taulukon alkioon i mahtuu vain vakiomäärä dataa

$\Rightarrow$ talletetaan alkioon $T[i]$ vain osoitin listaan, jossa on kaikki esiintyneet avaimet k, joilla $h(k) = i$.



Hajautus ja ketjuttaminen

Kun hajautusfunktio h on annettu, hajautustaulun operaatiot Search, Insert ja Delete saadaan suoraan tutuista linkitettyjen listojen operaatioista:

```
Hash-Search(T, k)
    L ← T[h(k)]
    return List-Search(L, k)

Hash-Insert(T, k)
    L ← T[h(k)]
    return List-Insert(L, k)

Hash-Delete(T, x)
    L ← T[h(key[x])]
    return List-Delete(L, x)
```

Aiemmin esitetyn perusteella Hash-Insert ja Hash-Delete sopivasti toteutettuina toimivat vakioajassa.

Haun Hash-Search(T, k) aikavaativuus on $O(1 + \ell)$, missä ℓ on listan $T[h(k)]$ pituus. Vakiotermi 1 esittää hajautusfunktion laskemista ja muita alkutoimia, ennen kuin listan läpikäynti alkaa.

Pahimmassa tapauksessa kaikki talletetut avaimet saavat saman hajautusfunktion arvon, jolloin $\ell = n$ ja aikavaativuudet ovat samat kuin linkitetyllä listalla. Tämä ei kuitenkaan ole tyypillistä (ellei hajautusfunktio ole huonosti valittu).

Tarkastellaan nyt haun Hash-Search(T, k) keskimääräistä aikavaativuutta.

Oletetaan, että kaikilla $i \in \{0, \ldots, m-1\}$ tapahtuman "$h(k) = i$" todennäköisyys on $1/m$. Tämä on suunnilleen realistinen oletus, jos hajautusfunktio h on osattu valita kohtuullisen hyvin.

Edellisen perusteella haun aikavaativuus on $O(1 + \ell)$, missä ℓ on listan $T[h(k)]$ pituus.

Koska avainlistojen yhteispituus on n, niiden keskipituus on $n/m = \alpha$. Siis haun keskimääräinen aikavaativuus on $O(1 + \alpha)$ eli lineaarinen täyttöasteen suhteen. Huomaa, että voimme sallia $\alpha > 1$.

Edellä esitettyä menettelyä kutsutaan suoraksi ketjutukseksi.

Erillisessä ketjutuksessa taulukossa T itsessään varataan tilaa yhdelle avaimelle osoitetta kohti, ja vasta yhteentörmäyksen sattuessa aletaan rakentaa linkitettyä listaa.

- säästää aikaa siinä (toivotussa) tapauksessa, että yhteentörmäyksiä on vähän
- kuluttaa paljon muistia, jos täyttöaste on kohtuullinen ja siis monet taulukon alkioista tyhjiä.

Levymuistille talletettavassa hajautustaulussa

- talletusalue varataan lohkoina
- saman hajautusavaimen saavat avaimet talletetaan samaan lohkoon
- linkitetty lista rakennetaan vasta, jos lohko tuli täyteen.

Siis yhteentörmäyksen sattuessakin voimme selvitä vain yhdellä levyhaulla. (Yksityiskohdat sivuutetaan.)

Merkkijonon muuntaminen luvuksi

Oletetaan, että avain k on p-merkkinen merkkijono

$$k = a_1 \dots a_p.$$

Halutaan funktio g, joka muodostaa avaimesta k luonnollisen luvun $k' = g(k)$.

Tulkitaan jokainen merkki a_i luonnolliseksi luvuksi v_i. Voidaan esim. sopia, että v_i on merkin a_i ASCII-koodi ($v_i \in \{ 0, \dots, 255 \}$). Suoraviivainen valinta olisi

$$g(k) = \sum_{i=1}^{p} v_i.$$

Tämä on helppo laskea ja ottaa jokaisen merkin huomioon.

Saadut arvot ovat kuitenkin melko pieniä. Jos esim. talletettavina avaimina on kaikki $256 \cdot 256 = 65\,536$ kahden mittaista merkkijonoa, niin arvot $g(k)$ pakkautuvat kaikki välille $0 \leq g(k) \leq 255 + 255 = 510$. Lisäksi merkkien järjestys ei vaikuta hajautukseen.

Tämä siis ei usein ole hyvä valinta.

Laajempi arvoalue saataisiin valitsemalla

$$g(k) = \sum_{i=1}^{p} v_i 256^{i-1}$$

eli tulkitsemalla suoraan k:n binääriesitys etumerkittömäksi kokonaisluvuksi.

Tämä kuitenkin johtaa nopeasti aritmeettiseen ylivuotoon. Asia voitaisiin korjata pitämällä vain (esim.) 32 alinta bittiä,

$$g(k) = \left(\sum_{i=1}^{p} v_i 256^{i-1} \right) \bmod 2^{32}$$

missä mod tarkoittaa jakojäännöstä. Tämä on nopeaa, koska kokonaisjako kakkosen potenssilla tapahtuu sivuttaissiirtona. Tämä kuitenkin on sama, kuin että otettaisiin mukaan vain neljä ensimmäistä merkkiä ($256 = 2^8 = 2^{32/4}$, joten $256^{i-1} \bmod 2^{32} = 0$ kun $i \geq 5$).

Puute voidaan korjata tekemällä kokonaisjako jonkin toisen kantaluvun suhteen:

$$g(k) = \left(\sum_{i=1}^{p} v_i 256^{i-1} \right) \bmod a$$

missä a ei ole kakkosen potenssi. Jos $a = m$ (talletusalueen koko), näin saadaan suoraan hajautusfunktion arvo pian esitettävälle jakojäännösmenetelmälle. Tämä kuitenkin on hidas laskea pitkillä avaimilla.

Edellisistä on runsaasti variaatioita: otetaan mukaan joka toinen merkki; ensimmäinen, viimeinen ja keskimmäinen; jne.

Jakojäännösmenetelmä

Oletetaan nyt, että avain k on luonnollinen luku. Perustapa kuvata se joukkoon $\{0, \ldots, m-1\}$ on ottaa jakojäännös

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

Tämä toimii usein hyvin, kun m valitaan sopivasti:

Jos $m = rs$ joillain r ja s ja talletettavat avaimet sattuvat olemaan r:n monikertoja $k_i = ir$, niin $k_i/m = i/s$. Siis jakojäännöksellä on vain s eri arvoa, eikä koko taulukko täyty.

Johtopäätös: talletusalueen koon m tulisi olla alkuluku (eli jaoton).

Olkoon toisaalta $m = 2^q + r$ jollain "pienellä" r. Jaetaan k lohkoihin, joissa q bittiä:

$$k = \sum_i u_i 2^{qi}.$$

Jos $r = 0$, niin $h(k) = u_0$ eli riippuu vain alimmista biteistä.

Jos $r = -1$, niin voidaan osoittaa, että $h(k)$ riippuu vain lohkojen summasta $\sum_i v_i$, ei järjestyksestä.

Yleisemmin jos r on "pieni", niin $h(k) = \sum_i \beta_i v_i$ joillain "pienillä" vakioilla β_i eikä siis jakaudu tasaisesti alueen $\{0, \ldots, m-1\}$ yli.

Johtopäätös: talletusalueen m pitää olla alkuluku joka ei ole kovin lähellä kakkosen potenssia.

Esimerkki Hajautettavia avaimia on noin $n = 10\,000$, ja halutaan pitää täyttöaste $\alpha \leq 2$. Siis $m \geq 5\,000$.

Lähimmät kakkosen potenssit ovat $2^{12} = 4096$ ja $2^{13} = 8192$. Näiden puoliväli on 6144.

Taulukosta löydetään, että 6143 on alkuluku.

Valitaan talletusalueen kooksi $m = 6143$. $\square$

Kertolaskumenetelmä

Olkoon $\lfloor x \rfloor = \max\{ i \in \mathbb{N} \mid i \le x \}$ positiivisen reaaliluvun x kokonaisosa ja $\mathrm{fr}(x) = x - \lfloor x \rfloor$ murtolukuosa. Kertolaskumenetelmässä hajautusfunktiona on

$$h(k) = \lfloor m\,\mathrm{fr}(kA) \rfloor ,$$

missä A on sopivasti valittu vakio. Teoreettiset tarkastelut ehdottavat valintaa

$$A \approx \frac{\sqrt{5}-1}{2} \approx 0,618.$$

Talletusalueen koko m ei nyt ole niin tärkeä. Laskujen yksinkertaistamiseksi valitaan mielellään $m = 2^p$ jollain p.

Oletetaan, että k on korkeintaan w bittiä. Valitaan $A = s/2^w$ jollain $s \in \mathbb{N}$, esim. $s = \lfloor 0{,}618 \cdot 2^w \rfloor$, jolloin A on 0,618 pyöristettynä lähimpään w-bittiseen murtolukuun. Valitaan $m = 2^p$ jollain p.

Kerrotaan w-bittiset luvut k ja $s = 2^w \cdot A$. Niiden tulo on $2w$-bittinen:

$$ks = 2^w \cdot kA = 2^w \cdot r_1 + r_0,$$

missä r_0 ja r_1 ovat w-bittisiä. Siis

$$\mathrm{fr}(kA) = \mathrm{fr}(r_1 + r_0/2^w) = r_0/2^w.$$

Kertolasku $m\mathrm{fr}(ks)$ suoritetaan siirtämällä murtolukua $r_0/2^w$ vasemmalle p bittiä, joten kokonaisosaksi saadaan binääriluvun r_0 p merkitsevintä bittiä.

Esimerkki Kuten edellä, oletetaan $n \approx 10\,000$ ja halutaan $\alpha \leq 2$ eli $m \geq 5\,000$. Valitaan $m = 2^{13} = 8192$.

Oletetaan, että avaimet ovat korkeintaan 32-bittisiä. Ratkaistaan yhtälö $s/2^{32} = (\sqrt{5} - 1)/2$ ja pyöristetään kokonaislukuun, jolloin saadaan $s = 2654435769$ ja siis $A = 2654435769/2^{32}$. Hajautusfunktioksi saadaan

$$h(k) = \left\lfloor 2^{13} \mathrm{fr}(k \cdot 2654435769/2^{32}) \right\rfloor .$$

□

Esimerkki Havainnollistetaan hajautusfunktion laskemista (epärealistisen) pienillä luvuilla. Oletetaan että avaimet ovat 8-bittisiä ja valitaan $m = 2^4 = 16$. Valitaan $A = 158/2^8 = 10011110_2/2^8$, sillä $((\sqrt{5} - 1)/2) \cdot 2^8 \approx 158,217$. Hajautetaan avain $k = 151 = 10010111_2$.

	binäärilukuna	desimaalilukuna
$2^8 A$	1001 1110	158
k	1011 1100	151
$2^8 kA$	0101 1101 0011 0010	23858
kA	0101 1101.0011 0010	
$\mathrm{fr}(kA)$	0.0011 0010	
$2^4 \mathrm{fr}(kA)$	0011.0010	
$\lfloor 2^4 \mathrm{fr}(kA) \rfloor$	0011	3

Siis $h(151) = 3$. □

Universaalihajautus

Hajautusfunktio pitäisi valita siten, että hajautusosoiteiden $h(k)$ jakauma olisi mahdollisimman lähellä joukon $\{0, \ldots, m-1\}$ tasaista jakaumaa. Tätä kriteeriä on mahdoton arvioida, jos emme tiedä avainten k jakaumaa (ja vaikka tietäisimme, se voi olla käytännössä hyvin vaikeaa).

Voimme siis tyhmyyttämme valita sovellustamme ajatellen juuri väärän hajautusfunktion.

Universaalihajautuksessa valitsemme satunnaisen hajautusfunktion, joka millä tahansa avainten k jakaumalla on keskimäärin hyvä.

Nyt siis huonon hajautusfunktion saaminen edellyttää (hyvin) huonoa tuuria.

Toinen tapa ajatella asiaa: Millä tahansa yksittäisellä hajautusfunktiolla h on joitain pahimman tapauksen avainjakaumia, joilla haku vaatii ajan $\Omega(n)$. Universaalihajautuksella ei ole mitään yksittäistä pahinta tapausta, vaan millä tahansa avainjakaumalla on pieni todennäköisyys, että hajautus menee huonosti.

Tällainen riskien tasapainottaminen on usein suotavaa.

Valitaan hajautusfunktio h seuraavasti:

- Oletetaan, että p on jokin kiinteä alkuluku, jolle $0 \leq k < p$ kaikille avaimille k. Koska talletusalueen koko on pienempi kuin avainten arvoalue (tai muuten tehtävä on triviaali), pätee myös $m < p$.
- Valitaan satunnaiset kokonaisluvut $1 \leq a \leq p - 1$ ja $0 \leq b \leq p - 1$.
- Hajautusfunktio on nyt

$$h(k) = ((ak + b) \bmod p) \bmod m.$$

Lause 4.1: Kiinnitetään mitkä tahansa kaksi eri avainta $k_1 \neq k_2$. Kun h valitaan edellisen menetelmän mukaan, niin yhteentörmäyksen $h(k_1) = h(k_2)$ todennäköisyys on korkeintaan $1/m$.

Todistus: Sivuutetaan; ks. Cormen et al. s. 235–236. □

Siis minkä tahansa yksittäisen avainparin yhteentörmäystodennäköisyys on korkeintaan sama, kuin jos osoitteet valittaisiin satunnaisesti. Tällaisen ehdon täyttävää hajautusta sanotaan universaaliksi.

Avoin hajautus (open addressing)

Avoimessa hajautuksessa ei käytetä erillisiä ylivuotolistoja. Kaikki avaimet talletetaan suoraan taulukkoon $T[0..m-1]$. Avaimia siis mahtuu korkeintaan m.

Etuja:

- ei osoittimia tilaa viemässä
- ei tarvita dynaamista muistinvarausta.

Haittoja (kuten jatkossa selitetään):

- haut hidastuvat taulukon täyttyessä
- poisto-operaatiot ongelmallisia (jos niitä on paljon).

Sen sijaan, että avaimeen k aiemmin liittyi vain yksi osoite $h(k) \in \{0, \ldots, m-1\}$, siihen nyt liittyy kokonainen jono osoitteita $h(k,0), h(k,1), h(k,2), \ldots, h(k,m-1)$, $h(k,i) \in \{0, \ldots, m-1\}$ kaikilla i.

Avaimen lisääminen: kokeillaan järjestyksessä osoitteita $h(k,0)$, $h(k,1)$, $h(k,2)$ jne., kunnes löytyy vapaa paikka.

Avaimen etsiminen: kokeillaan järjestyksessä osoitteita $h(k,0)$, $h(k,1)$, $h(k,2)$ jne. kunnes joko

- avain löytyy tai
- löytyy tyhjä paikka, jolloin avain ei ole taulukossa.

Jonoa $(h(k,0), h(k,1), \ldots, h(k,m))$ kutsutaan avaimen k kokeilujonoksi (probe sequence). Vaadimme, että jokaisella avaimella se on jokin jonon $(0, 1, \ldots, m-1)$ permutaatio, ts. jokainen taulukon T indeksi esiintyy kokeilujonossa tasan yhden kerran.

Esittelemme kohta vaihtoehtoja kokeilujonon valitsemiseen. Katsotaan ensin, miten annetulla kokeilujonolla lisäys ja etsiminen sujuvat:

```
Hash-Insert(T, k)
    i ← 0
    repeat  j ← h(k, i)
            if T[j] = Nil
               then T[j] ← k
                    return j
            i ← i + 1
      until i = m
    error "ylivuoto"
```

```
Hash-Search(T, k)
    i ← 0
    repeat  j ← h(k, i)
            if T[j] = k
               then return j
            i ← i + 1
      until T[j] = Nil or i = m
    return Nil
```

Tässä oletetaan, että taulukon alkiot on alustettu arvoon Nil. Hash-Insert palauttaa lisätyn avaimen todellisen osoitteen taulukossa. Avaimeen liittyvä muu data on taas jätetty huomiotta.

Alkioon $T[j]$ talletetun alkion poisto voidaan toteuttaa asettamalla $T[j] \leftarrow$ Del, missä Del on tähän tarkoitukseen varattu erikoisarvo.

Lisäyksessä tilanne $T[j] =$ Del rinnastetaan tilanteeseen $T[j] =$ Nil: alkio j on vapaa, lisäys voidaan suorittaa.

Haussa tilanne $T[j] =$ Del tarkoittaa $T[j] \neq$ Nil ja $T[j] \neq k$ kaikilla "oikeilla" avaimilla k, joten haku jatkuu.

Jos poistoja tehdään paljon, hakuketjuista voi muodostua pitkiä, vaikka taulukon täyttöaste olisi alhainen. Tässä tilanteessa ylivuotoketjut voivat olla parempi ratkaisu.

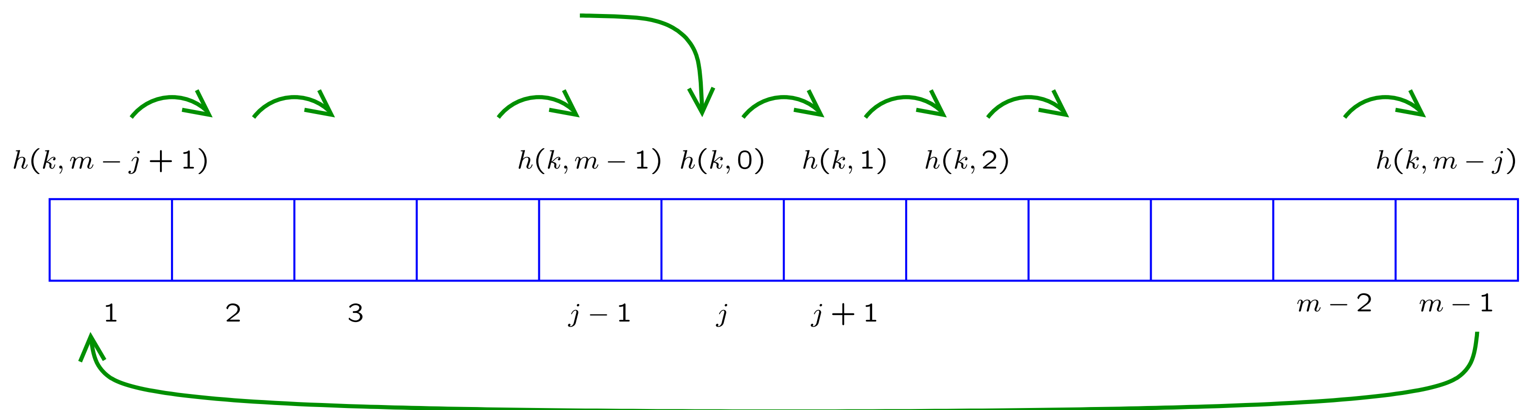
Tarkastelemme jatkossa vain lisäyksiä ja hakuja.

Lineaarinen kokeileminen: Yksinkertaisin kokeilujono saadaan valitsemalla

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

missä $h'(k) = h(k,0)$ on vastaava hajautusfunktio kuin ylivuotolistoja käytettäessä (esim. $h'(k) = k \bmod m$).

Siis kokeileminen lähtee alkiosta $T[h'(k)]$, mistä edetään yksi alkio kerrallaan siirtyen taulukon lopusta takaisin alkuun.



Esimerkki Valitaan $m = 11$ ja $h'(k) = k \bmod 11$. Lisätään avaimet 148, 119, 127, 137, 188, 183, 152 ja 159. Katsotaan ensin kokeilujonojen alkukohdat:

$h'(k)$	0	1	2	3	4	5	6	7	8	9	10
k		188				148 137 159	127	183		119 152	

Taulukkoon tulee siis

j	0	1	2	3	4	5	6	7	8	9	10
$T[j]$	159	188				148	127	137	183	119	152

Edellinen (hieman epärealistinen) esimerkki tuo esille lineaariseen kokeiluun liittyvän ensisijaisen kasautumisen (primary clustering):

- taulukkoon muodostuu laajoja yhtenäisiä varattuja alueita
- kaikkien tälle alueelle osuvien avaimien kokeilujonot yhtyvät.

Ilmiö esiintyy, vaikka arvot $h'(k)$ jakautuisivat taulukkoon tasaisesti:

- varattuja alueita muodostuu aina jonkin verran, koska yhteentörmäyksiä sattuu pienilläkin täyttöasteilla
- syntymäpäiväparadoksi: jos taulukossa on $\sqrt{2m}$ avainta, todennäköisyys välttyä kokonaan yhteentörmäyksiltä on alle 1/2, vaikka täyttöaste on $\alpha = \sqrt{2m}/m = 2/\sqrt{m} \ll 1$
- mitä suurempi varattu alue on, sitä todennäköisemmin se myös laajenee.

Avoimessa hajautuksessa siis

- ei riitä, että kokeilun alkukohdat $h(k, 0)$ jakautuvat tasaisesti joukkoon $\{0, \ldots, m-1\}$
- koko kokeilujonojen $((h(k, 0), \ldots, h(k, m-1))$ pitäisi jakautua tasaisesti kaikkien joukon $\{0, \ldots, m-1\}$ permutaatioiden yli.

Lineaarinen kokeilu ei selvästi täytä jälkimmäistä ehtoa, koska mahdollisia kokeilujonoja on vain m, kun taas permutaatioita on $m! \gg m$.

Lisäksi mahdolliset m jonoa menevät pahasti "päällekkäin":

Jos $h(k_1, i) = h(k_2, j)$ joillakin k_1, k_2, i, j, niin myös $h(k_1, i+1) = h(k_2, j+1)$, $h(k_1, i+2) = h(k_2, j+2)$ jne. kunnes jompi kumpi jono loppuu.

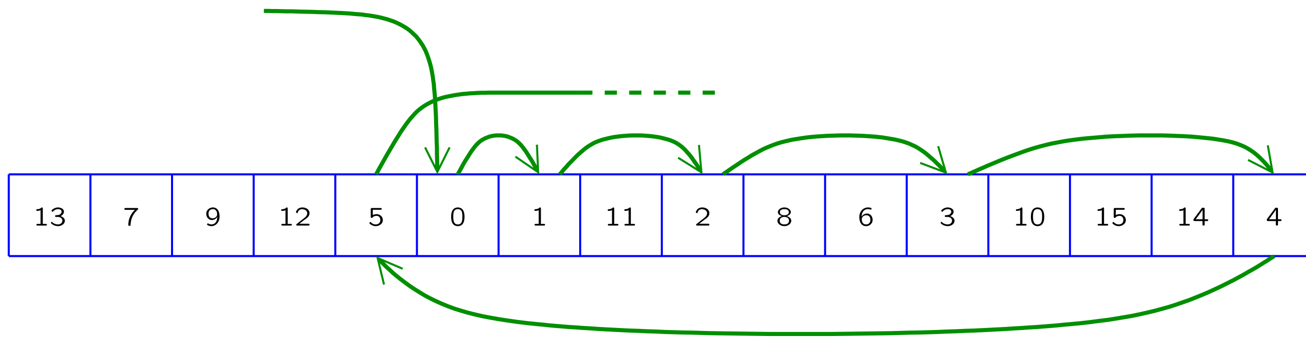
Neliöllinen kokeileminen: valitaan sopivat vakiot c_1 ja c_2 ja asetetaan

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m.$$

Arvojen c_1 ja c_2 valinnalla on väliä.

Jos $m = 2^r$ jollain r, valinta $c_1 = c_2 = 1/2$ antaa kokeilujonon, joka on jonon $(0, \ldots, m-1)$ permutaatio (todistus sivuutetaan). Lisäksi se on helppo laskea:

$$\begin{aligned}
h(k,0) &= h'(k) \\
h(k,1) &= h'(k) + 1 \\
h(k,2) &= h'(k) + 3 &&= h(k,1) + 2 \\
h(k,3) &= h'(k) + 6 &&= h(k,2) + 3 \\
h(k,4) &= h'(k) + 10 &&= h(k,3) + 4 \\
&\ldots
\end{aligned}$$



Neliöllinen kokeilujono kun $m = 16 = 2^4$ ja $c_1 = c_2 = 1/2$.

Jos m on alkuluku, kokeilujonoksi ei yleensä voi saada jonon $(0, \ldots, m-1)$ permutaatiota. Useimmat valinnat (esim. $c_1 = 0$ ja $c_2 = 1$) kuitenkin takaavat, että kokeilujonon $\lfloor m/2 \rfloor$ ensimmäistä arvoa ovat erillisiä.

Neliöllinen kokeilu välttää ensisijaisen kasautumisen. Yleensä jos kaksi kokeilujonoa sattuu samaan paikkaan, ne kuitenkin jatkavat eri askelpituudella eivätkä mene päällekkäin.

Neliöllinen kokeilu kuitenkin kärsii toissijaisesta kasautumisesta: jos $h'(k_1) = h'(k_2)$, niin avainten k_1 ja k_2 kokeilujonot ovat kokonaan samat.

Erilaisia neliöllisiä kokeilujonoja (kiinteillä c_1 ja c_2) on edelleen vain $m \ll m!$.

Kaksoishajautuksessa tarvitaan kaksi hajautusfunktiota h' ja h'', jotka yhdistämällä saadaan kokeilujono

$$h(k,i) = (h'(k) + i \cdot h''(k)) \bmod m.$$

Kokeilujono siis on kullakin avaimella k lineaarinen, mutta askelpituus $h''(k)$ riippuu avaimesta. Jotta kokeilujono kävisi läpi koko taulukon, luvuilla m ja $h''(k)$ ei saa olla yhteisiä tekijöitä (paitsi tietysti 1).

Mahdollisia valintoja:

- $m = 2^r$ ja $h''(k)$ pariton, esim. $h''(k) = 2k + 1$
- m alkuluku ja $1 \leq h''(k) \leq m - 1$, esim. $h''(k) = 1 + (k \bmod (m - 1))$

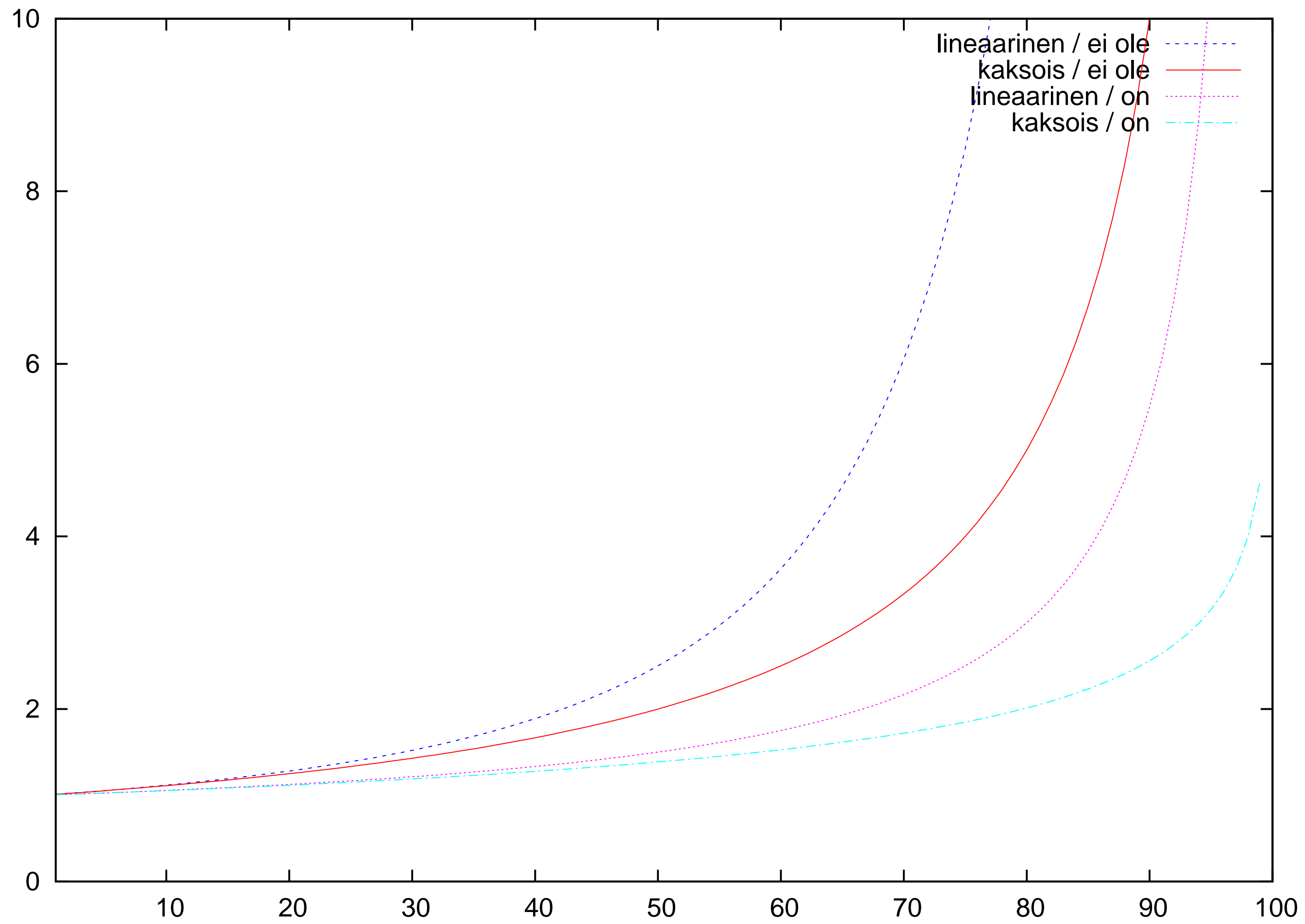
Kokeilujonojen lukumäärä on nyt $\Theta(m^2)$, mikä (hieman) lähempänä teoreettista päämäärää $m!$ kuin lineaarisen ja neliöllisen kokeilun m.

Oheisessa taulukossa on lineaarisen kokeilun ja kaksoishajautuksen vaatima kokeilujen lukumäärä täyttöasteella $\alpha = n/m$. Kummallekin on kaksi tapausta: haettu alkio ei ole / on taulukossa. Oletuksena on, että arvot $h'(k)$ jakautuvat tasaisesti.

	ei löydy	löytyy
lineaarinen	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$
kaksoish.	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

Kaavat ovat likiarvoja, jotka pätevät suurilla m.

Neliöllisen kokeilun on havaittu käyttäytyvän samaan tapaan kuin kaksoishajautuksen.



Vaaka-akselilla α prosentteina; pystyakselilla haun vaatimat kokeilut

Taulukon laajentaminen

Tarkastellaan (teoreettiselta kannalta) tilannetta, jossa

- emme ennakolta tiedä talletettavien alkioiden määrä ja
- haluamme pitää täyttöasteena korkeintaan α_{max}.

Varataan aluksi kokoa $m = m_0$ oleva taulukko $T[0 \,..\, m_0 - 1]$. Kun talletettujen alkioiden arvo saavuttaa asetetun rajan $n = \alpha_{\text{max}} m$, laajennetaan taulukko:

- Muodostetaan kokoa $m' > m$ oleva tyhjä taulukko $T'[0 \,..\, m' - 1]$.
- Käydään taulukko T läpi ja hajautetaan kaikki löydetyt avaimet yksitellen taulukkoon T'.
- Vapautetaan taulukko T.

Nyt käytetään taulukkoa T', kunnes sekin saavuttaa täyttörajansa $n = \alpha_{\text{max}} m'$, jolloin taas laajennetaan.

Kopiointivaiheen aikavaativuus on $\Theta(m + m')$, eli paljon. Kannattaa kopioida mahdollisimman harvoin. Siis laajennetaan kerralla taulukko kaksinkertaiseksi: $m' = 2m$.

Laajennushetkellä $n = \alpha_{\mathsf{max}} m$, joten aikavaativuus on $\Theta(n/\alpha_{\mathsf{max}} + 2n/\alpha_{\mathsf{max}}) = \Theta(n)$. Valitaan yksiköt niin, että tämän kustannus on tasan $2n$ "euroa" eli kaksi euroa per siirrettävä alkio.

Voidaan olettaa, että tällöin normaali lisäys maksaa suunnilleen yhden euron.

Tarkastellaan koko toimenpidejonon tasoitettua vaativuutta (amortized complexity) käyttämällä kirjanpitomenetelmää.

Asetamme yhden lisäyksen tasoitetuksi kustannukseksi viisi euroa:

- Aluksi meillä ei ole yhtään rahaa.
- Saamme viisi euroa jokaisesta lisätystä alkiosta.
- Kun varojen käyttö suunnitellaan oikein, näin saadut tulot riittävät kattamaan menot, eli
  - yksi euro per normaali lisäys
  - lisäksi $2n$ euroa aina kun kopioidaan n alkiota.

Jos tämä onnistuu, niin tiedämme, että n lisäystä taulukon kahdentamisineen aiheuttaa kustannuksia korkeintaan $5n$ euroa eli viisi kertaa niin paljon kuin lisäykset valmiiksi oikean kokoiseen taulukkoon.

Siis taulukon dynaaminen laajentaminen lisää kokonaisaikaa vain vakiokertoimella, vaikka jonkin yksittäiset lisäykset (ne joista seuraa laajennus) ovat hyvin raskaita.

Avaimen k lisäämisestä saatavat 5 euroa jaetaan seuraavasti:

- yksi euro maksaa avaimen k lisäyksen
- kaksi euroa annetaan avaimelle k "talteen"
- kaksi euroa annetaan talteen jollekin avaimelle, jolla ei vielä ole yhtään rahaa.

Kun taulukko kahdennetaan, jokaisella avaimella on tallessa kaksi euroa, joista se voi maksaa oman siirtonsa:

- Edellisen kahdennuksen jäljiltä taulukossa oli $\alpha_{\text{max}} m$ "vanhaa" avainta, joiden rahat menivät kahdennuksessa.
- Koska $m' = 2m$ ja täyttöraja α_{max} pysyy samana, uuden kahdennuksen tullessa myös avainten lukumäärä on kaksinkertaistunut.
- Siis jokaista vanhaa avainta kohti on yksi "uusi" avain, jolta se on saanut tarvittavat rahat.

Yhteenveto

Hajautusfunktio: Parhaan löytäminen vaikeaa, mutta perusmenetelmät (kuten jakojäännösmenetelmä) yleensä "riittävän hyviä". Kannattaa kokeilla.

Ylivuotoketjut:

- tyypillinen täyttöaste $\alpha \approx 1{,}0$
- ei kovin herkkä suurille täyttöasteille
- poistot voidaan toteuttaa normaalisti
- joustavaa jos avainten lukumäärää ei tiedetä ennakolta.

Avoin hajautus:

- tyypillinen täyttöaste $\alpha \leq 0{,}5$
- säästää muistia ylivuotoketjuihin verrattuna, joten pienempi täyttöaste on mahdollinen
- talletusalueen koko staattinen: etu ja haitta

Tasapainoiset hakupuut vs. hajautus:

- hajautus yleensä tehokas ja yksinkertainen
- huono hajautusmenetelmä voi johtaa huonoon tulokseen
- tasapainoinen hakupuu takaa suoritusajan $O(\log n)$ aina
- hakupuu mahdollistaa avainten käsittelemisen järjestyksessä jne.

Eri hakemistorakenteet käyttäytyvät eri tavoin muistihierarkiassa (levymuisti, keskusmuisti, välimuisti, ...). Tästä voi aiheutua vaikeasti ennustettavia sovelluskohtaisia eroja. Esim. avoin hajautus lineaarisella kokeilulla *voi* joissain tilanteissa toimia paremmin kuin "kehittyneemmät" ratkaisut.

5. Keko

Tietorakenne keko eli kasa (heap) on tehokas toteutus abstraktille tietotyypille prioriteettijono, jonka operaatiot ovat seuraavat:

Insert(S, x): lisää avaimen x prioriteettijonoon S
Maximum(S): palauttaa prioritettijonon S suurimman avaimen
Delete-Max(S): palauttaa prioritettijonon S suurimman avaimen ja poistaa sen prioriteettijonosta
Increase-Key(S, x, k): jos $key[x] < k$ niin kasvattaa avaimen $key[x]$ arvoon k; muuten ei tee mitään.

Oleellinen ero normaaliin joukkoon verrattuna on, että olemme luopuneet yleisestä hausta Search(S, k). Korvauksena yo. operaatiot saadaan hyvin tehokkaiksi.

Edellisen kalvon tietotyyppi on tarkemmin maksimiprioriteettijono. Symmetrisesti voidaan myös toteuttaa minimiprioriteettijono, jonka operaatiot ovat Insert, Minimum, Delete-Min ja Decrease-Key. Valinta on joko-tai: sama tietorakenne ei tue maksimi- ja minimioperaatioita.

Tyypillisiä prioriteettijonon käyttökohteita:

- käyttöjärjestelmän työjono
- simulaatiojärjestelmän tapahtumajono: minimiversio, alkioina tulevat tapahtumat, prioriteetteinä tapahtuma-ajat
- sovellukset muissa algoritmeissa: kekojärjestäminen, Dijkstran algoritmi lyhimpien reittien löytämiseksi jne. (näihin palataan).

Erityisesti sovellukset muiden algoritmien aputietorakenteena motivoivat prioriteettijonon erilaisia toteutuksia (joita on paljon muitakin kuin keko).

Seuraavaksi esiteltävä tietorakenne keko toteuttaa Maximum-operaation vakioajassa ja muut prioriteettijonon operaatiot ajassa $O(\log n)$.

Samaan päästäisiin itse asiassa tasapainoisilla hakupuilla:

- pidetään erikseen muistissa pienin alkio, ja päivitetään se ajassa $O(\log n)$ poistojen yhteydessä
- Increase-Key voidaan toteuttaa poistamalla alkio ja lisäämällä se uudella avaimella.

Keko on kuitenkin

- yksinkertaisempi toteuttaa
- vakiokertoimiltaan pienempi.

Lisäksi keosta voidaan "helposti" tehdä parannettuja versioita kuten binomikeko ja Fibonacci-keko (jotka eivät kuulu tämän kurssin alueeseen).

Motivoivana esimerkkinä ennen kekoa esitetään, miten järjestäminen sujuu prioriteettijonon avulla:

```
Heap-Sort(A[1..n])
    for i ← 1 to n
        do Min-Insert(Q, A[i])
    for i ← 1 to n
        do A[i] ← Delete-Min(Q)
```

Tässä on käytetty proseduurin nimenä Min-Insert selventämään, että kysymys on minimiprioriteettijonosta. Käytämme tätä konventiota jatkossa silloin, kun asia ei muuten ole selvä.

Jos Min-Insert ja Delete-Min sujuvat ajassa $O(\log n)$, niin tämä selvästi sujuu ajassa $O(n \log n)$:

- suurilla n oleellisesti parempi kuin lisäysjärjestämisen $O(n^2)$
- tietyssä mielessä optimaalista (vrt. Johdatus diskreettiin matematiikkaan).

Maksimikeko on puu, johon tallennetut avaimet toteuttavat kekoehdon

$$key[parent[x]] \geq key[x] \qquad \text{kaikilla solmuilla } x.$$

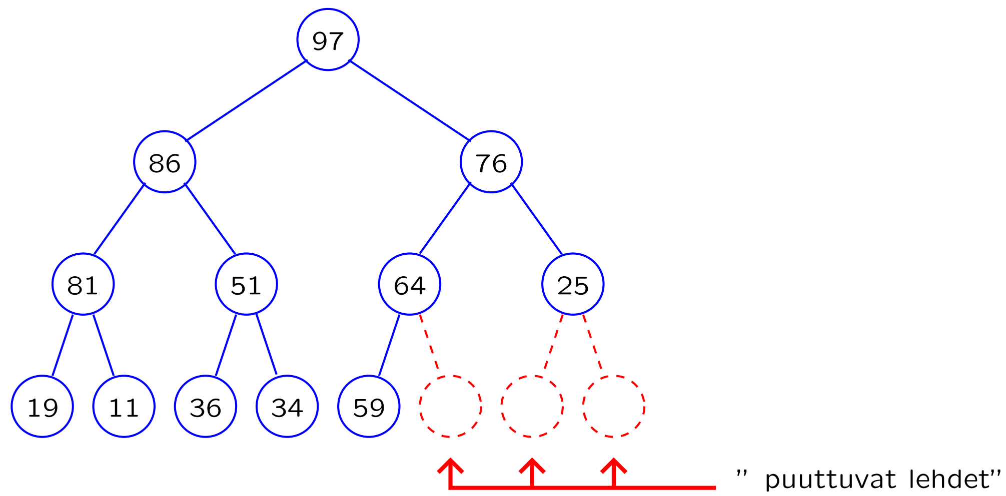
Tästä seuraa, että jokaisen alipuun suurin avain on kyseisen alipuun juuressa.

Minimikeolle vastaavasti $key[parent[x]] \leq key[x]$, jolloin juuressa on pienin avain.

Huomaa keon oleellinen ero hakupuihin: lasten avainten järjestys on vapaa.

Tällä kurssilla keot ovat aina binäärikekoja, jolloin kyseiset puut ovat binääripuita. Lisäksi vaaditaan, että ne ovat melkein täydellisiä: syvintä tasoa lukuunottamatta tasolla i on täydet 2^i solmua, ja syvimmän tason solmut ovat niin vasemmalla kuin mahdollista.

Tämän yleistys olisi k-keko, jossa puun jokaisella sisäsolmulla on tasan k lasta (paitsi mahdollisesti yhdellä, joka on vajaa).

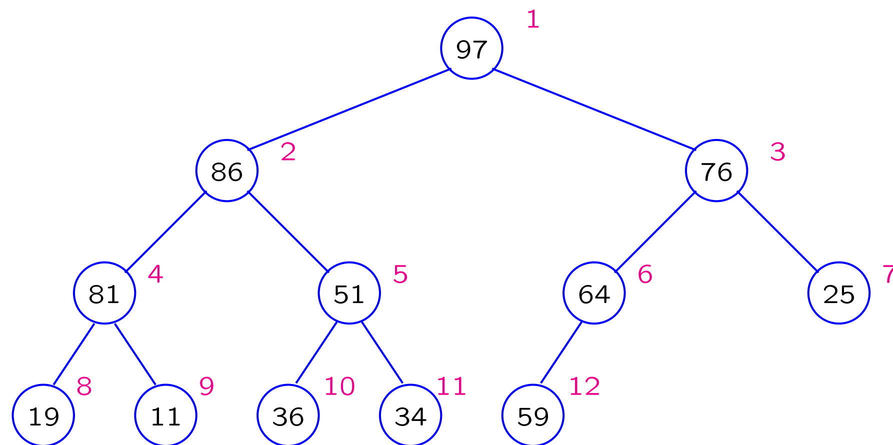


12-alkioisessa keossa 15 solmun täydellisestä puusta puuttuu kolme "viimeistä" lehteä.

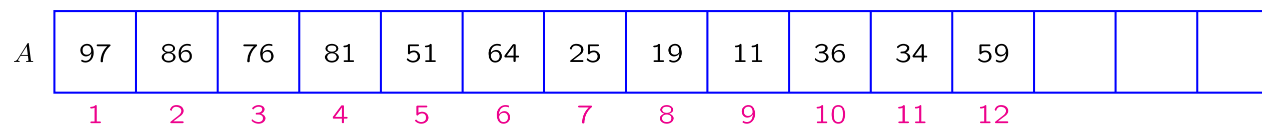
Koska puu on hyvin tasapainoinen, se voidaan tallettaa tehokkaasti taulukkoon ilman osoittimia.

Tallettaminen tapahtuu tasoittain:

- tasolla 0 on 1 solmu, joka menee paikkaan $A[1]$
- tasolla 1 on 2 solmua, jotka menevät paikkoihin $A[2\,..\,3]$
- tasolla 2 on 4 solmua, jotka menevät paikkoihin $A[4\,..\,7]$
- ...
- tasolla i on 2^i solmua, jotka menevät paikkoihin $A[2^i\,..\,2^{i+1}-1]$



$heap\text{-}size[A] = 12$ $length[A] = 15$



Keon talletusrakenne; *length* on talletusalueen maksimikoko ja *heap-size* käytössä oleva osuus.

Talletetaan lisäksi viimeisen lehden osoite muuttujaan $\mathit{heap\text{-}size}[A]$.

Nyt solmun vanhempi, vasen lapsi ja oikea lapsi saadaan helposti:

```
Parent(i)
    return ⌊i/2⌋

Left(i)
    if 2i > heap-size[A]
       then return Nil
       else  return 2i

Right(i)
    if 2i + 1 > heap-size[A]
       then return Nil
       else  return 2i + 1
```

Keon käsittelyssä keskeinen apuproseduuri on Heapify(A, i), joka korjaa kekoehdon, jos se on rikki solmun i kohdalla.

Oletus on, että solmun i vasen ja oikea alipuu toteuttavat kekoehdon.

Kutsun Heapify(A, i) jälkeen koko solmusta i lähtevä alipuu toteuttaa kekoehdon.

Maksimikeon tapauksessa menettely on seuraava:

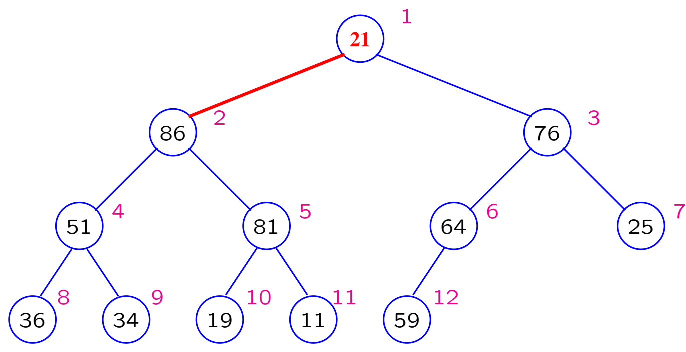
- Jos $A[i]$ on pienempi kuin toinen lapsistaan, vaihdetaan se suuremman lapsen kanssa.
- Jatketaan rekursiivisesti muuttuneesta lapsesta.

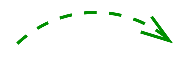
Minimikeolla luonnollisesti valitaan pienempi lapsista, jos juuri on sitä suurempi.

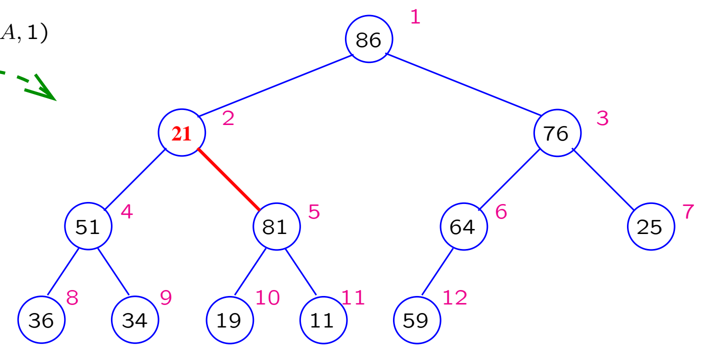
```
Max-Heapify(A, i)
    l ← Left(i)
    r ← Right(i)
    if l ≤ heap-size[A] and A[l] > A[i]
        then next ← l
        else next ← i
    if r ≤ heap-size[A] and A[r] > A[next]
        then next ← r
    if next ≠ i
        then vaihda A[i] ↔ A[next]
             Max-Heapify(A, next)
```

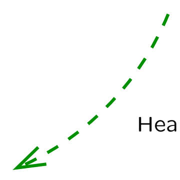
Aikavaativuus on verrannollinen puun korkeuteen eli $O(\log n)$.

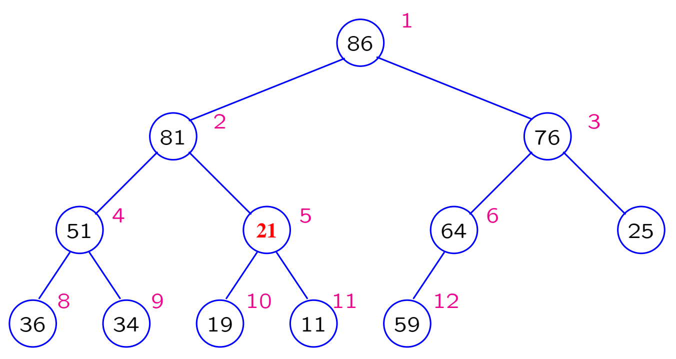
Tilavaativuus ylläolevassa rekursiivisessa toteutuksessa on sekin $O(\log n)$, mutta proseduuri voidaan helposti toteuttaa iteratiivisesti vakiotilassa.


21
1
86
2
76
3
51
4
81
5
64
6
25
7
36
8
34
9
19
10
11
11
59
12


Heapify$(A, 1)$


86
1
21
2
76
3
51
4
81
5
64
6
25
7
36
8
34
9
19
10
11
11
59
12


Heapify$(A, 2)$


86
1
81
2
76
3
51
4
21
5
64
6
25
36
8
34
9
19
10
11
11
59
12

Suurin alkio on aina keon juuressa:

Maximum(A)
    return $A[1]$

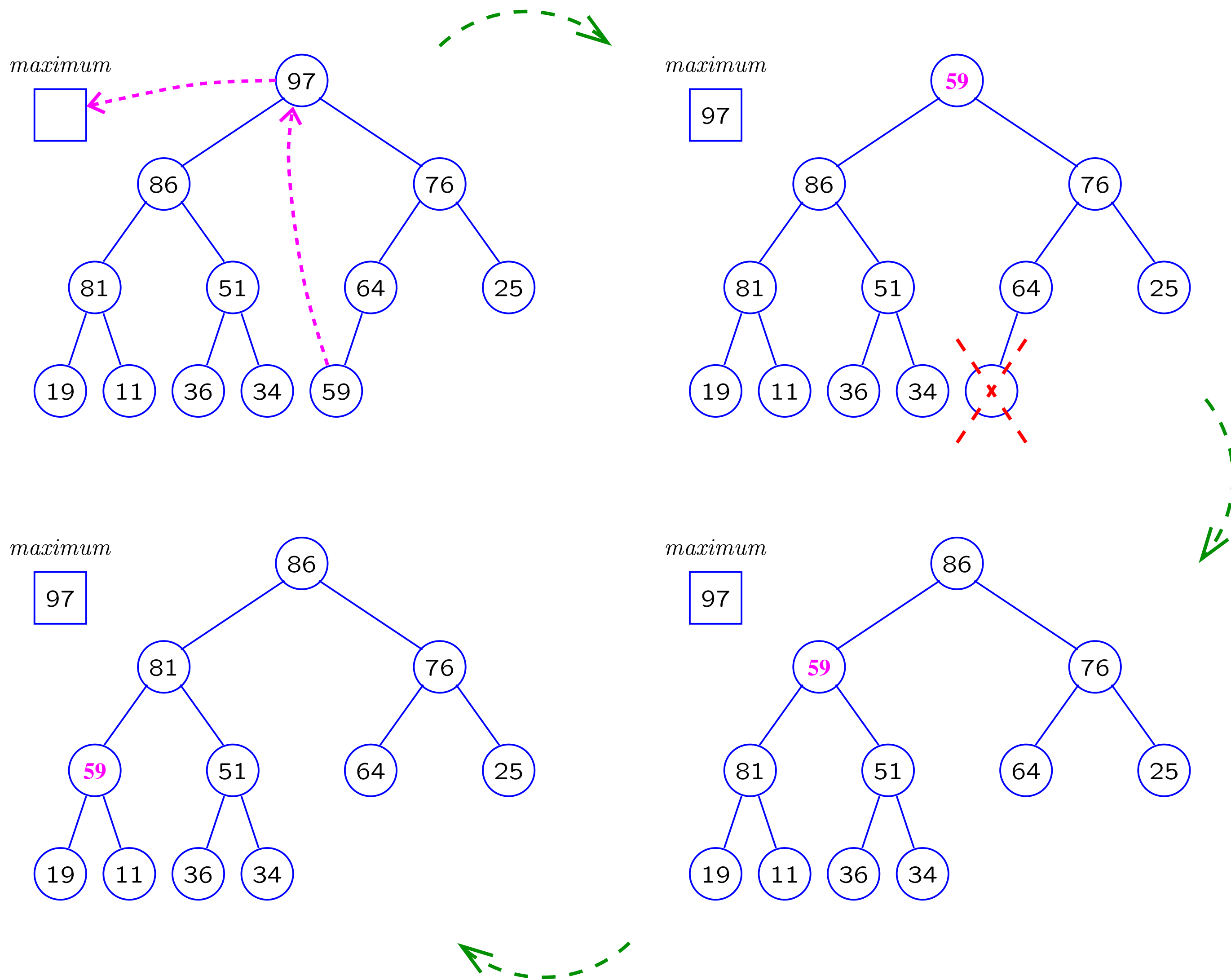
Nyt voimme helposti toteuttaa myös sen poiston:

- prioriteettijonosta halutaan poistaa avain $A[1]$
- talletustaulukosta pitää vapauttaa viimeinen alkio $A[\mathit{heap\text{-}size}[A]]$

$\Rightarrow$ vaihdetaan avain $A[1]$ positioon $A[\mathit{heap\text{-}size}[A]]$ ja korjataan rikki menneet kekoehdot.

Heap-Delete-Max(A)
    $\mathit{maximum} \leftarrow A[1]$
    $A[1] \leftarrow A[\mathit{heap\text{-}size}[A]]$
    $\mathit{heap\text{-}size}[A] \leftarrow \mathit{heap\text{-}size}[A] - 1$
    Heapify($A, 1$)
    return $\mathit{maximum}$

Selvästi aikavaativuus $O(\log n)$ ja tilavaativuus $O(1)$ kuten Heapify.

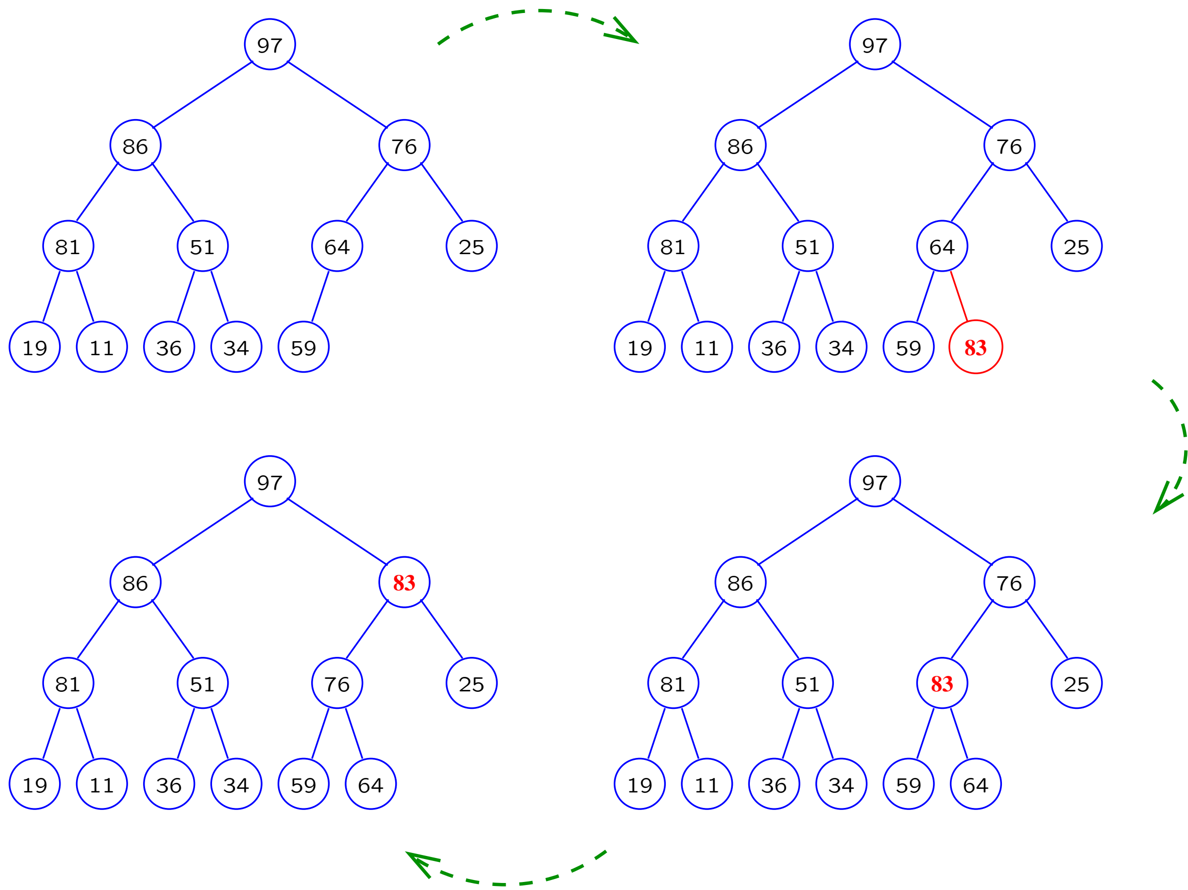

maximum
97
86
76
81
51
64
25
19
11
36
34
59
maximum
97
59
86
76
81
51
64
25
19
11
36
34
maximum
97
86
81
76
59
51
64
25
19
11
36
34
maximum
97
86
59
76
81
51
64
25
19
11
36
34

Vastaavasti lisääminen aloitetaan seuraavasta vapaasta positiosta $A[\mathit{heap\text{-}size}[A]+1]$.

Lisättyä avainta siirretään ylöspäin, kunnes se ei enää riko kekoehtoa. Huomaa, että jos lisätty avain maksimikeossa on suurempi kuin vanhempansa, se on myös suurempi kuin sisarensa, joten vaihto voidaan suorittaa.

```
Max-Heap-Insert(A, k)
    if heap-size[A] = length[A]
       then error "ylivuoto"
    i ← heap-size[A] + 1
    heap-size[A] ← i
    while i > 1 and A[Parent(i)] < k
          do A[i] ← A[Parent(i)]
             i ← Parent(i)
    A[i] ← k
```

Aikavaativuus on verrannollinen puun korkeuteen eli $O(\log n)$, tilavaativuus vakio.



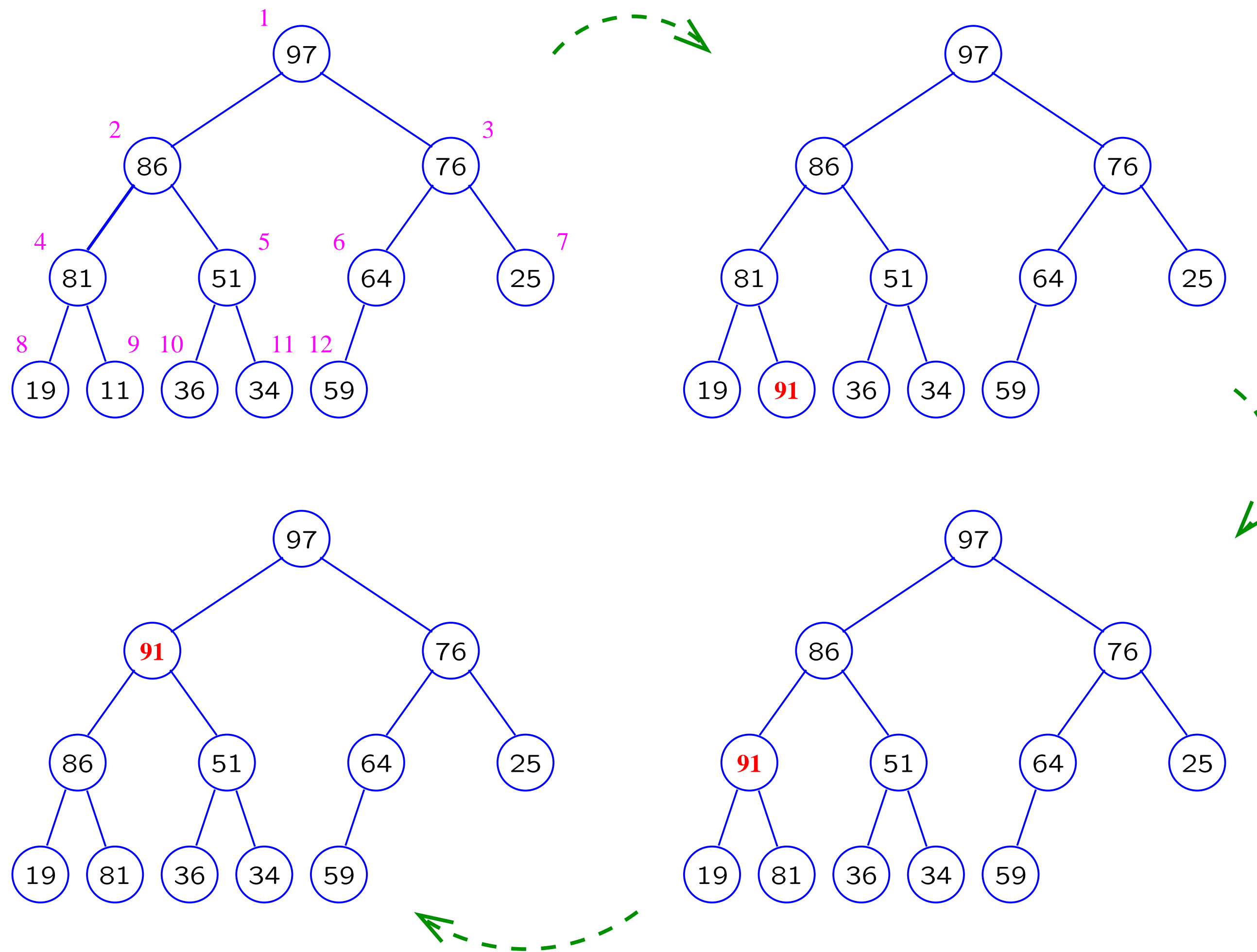
Lisätään avain 83 maksimikekoon.

Avaimen kasvattaminen toimii kuten lisääminen, paitsi että nyt uusi avain voi ilmestyä mihin tahansa kohtaan puuta.

Koska oletuksen mukaan uusi avain on suurempi kuin vanha, riittää siirtää sitä ylöspäin.

```
Heap-Increase-Key(A, i, k)
    if k > A[i]
       then while i < 1 and A[Parent(i)] < k
                  do A[i] ← A[Parent(i)]
                     i ← Parent(i)
    A[i] ← k
```

Aikavaativuus on taas $O(\log n)$ ja tilavaativuus vakio.



Avaimen kasvatus Heap-Increase-Key($A, 9, 91$), missä 91 on uusi avain ja 9 vanhan avaimen 11 indeksi taulukossa.

Muita keko-operaatioita

Operaatiolla Heap-Increase-Key vastakkainen operaatio Max-Heap-Decrease-Key(A, i, k) pienentää maksimikeon paikkaan $A[i]$ talletetun avaimen arvoksi k. Oletus on, että vanha arvo on suurempi kuin k.

```
Max-Heap-Decrease-Key(A, i, k)
    if k < A[i]
        then A[i] ← k
             Heapify(A, i)
```

Koska avainta $A[i]$ pienennettiin, se siirtyy maksimikeossa alaspäin. Mitään vaikutuksia ei säteile muihin osiin kekoa.

Avaimen $A[i]$ vaihtaminen mielivaltaiseen arvoon k voidaan nyt toteuttaa kutsumalla Heap-Increase-Key(A, i, k) tai Heap-Decrease-Key(A, i, k) sen mukaan, onko $k > A[i]$ vai $k < A[i]$.

Kohdassa i oleva avain voidaan poistaa seuraavasti:

1. Olkoon $k = A[\mathit{heap\text{-}size}[A]]$ talletusrakenteen viimeinen avain.

2. Jos $k > A[i]$, suorita Heap-Increase-Key(A, i, k).

3. Muuten suorita Heap-Decrease-Key(A, i, k).

4. Aseta $\mathit{heap\text{-}size}[A] \leftarrow \mathit{heap\text{-}size}[A] - 1$.

Kaikkien näiden operaatioiden aikavaativuus on edelleen $O(\log n)$.

Keon muodostamista annetuista alkioista tarvitaan esim. osana kohta esitettävää kekojärjestämistä (heapsort). Se voidaan tehdä ajassa $O(n \log n)$ lisäämällä alkiot yksitellen aluksi tyhjään kekoon.

Seuraava algoritmi kuitenkin selviytyy tehtävästä ajassa $O(n)$. Oletus on,että avaimet ovat jo taulukossa $A[1 \mathinner{..} n]$, missä $n = \mathit{length}[A]$, mutta eivät (välttämättä) toteuta kekoehtoa.

Build-Max-Heap(A)

$\mathit{heap}\text{-}\mathit{size}[A] \leftarrow \mathit{length}[A]$
for $i \leftarrow \lfloor \mathit{length}[A]/2 \rfloor$ **downto** 1
  do Max-Heapify(A, i)

Minimikeon muodostamiseksi tarvitsee vain vaihtaa kutsun Max-Heapify tilalle Min-Heapify.

Analysoimme nyt algoritmia Build-Max-Heap tarkemmin.

Algoritmin toiminta perustuu silmukkainvarianttiin:

> kaikilla $i < j \leq length[A]$ solmusta j lähtevä alipuu toteuttaa kekoehdon.

Miksi tämä pätee:

- Aluksi $i = \lfloor length[A]/2 \rfloor$. Jos $j > i$, niin $2j > length[A]$, jolloin solmulla j ei ole lapsia. Yksisolmuinen alipuu toteuttaa triviaalisti kekoehdon.
- Oletetaan, että invariantti pätee, kun silmukan runko alkaa. Siis erityisesti solmuista Left(i) $= 2i > i$ ja Right(i) $= 2i + 1 > i$ alkavat alipuut toteuttavat kekoehdon. Toisin sanoen kutsun Max-Heapify(A, i) ennakkoehdot ovat voimassa. Kutsun jälkeen myös solmusta i alkava alipuu toteuttaa kekoehdon.

Siis lopuksi kekoehto pätee koko puussa.

Naiivi analyysi antaisi aikavaativuudeksi $O(n \log n)$:

- Silmukka suoritetaan $O(n)$ kertaa.
- Silmukan yhtä suorituskertaa dominoi proseduuri Max-Heapify, jonka aikavaativuus on $O(\log n)$.

Tarkemmalla analyysillä saadaan kuitenkin tarkempi arvio $O(n)$. Perusidea:

- Kutsun Max-Heapify(A, i) aikavaativuus on itse asiassa $O(h)$, missä h on solmun i korkeus puussa.
- Useimmilla solmuilla $h \ll \log n$. Esim. noin puolet solmuista on lehtiä, joilla $h = 0$.

Olkoon $h(i)$ solmun i korkeus puussa. Algoritmin Max-Heapify aikavaativuus on siis $O(\sum_i h(i))$. Analysoidaan tätä summaa tarkemmin.

Olkoon $n = length[A]$ ja $k = \lfloor \log_2 n \rfloor$ koko puun korkeus.

Syvyydellä d on korkeintaan 2^d solmua. Näiden solmujen korkeus on $k - d$ tai $k - d - 1$ eli korkeintaan $k - d$.

Ryhmittelemällä solmut tasoittain voimme siis arvioida

$$\begin{aligned} \sum_i h(i) &\leq \sum_{d=0}^{k} 2^d \cdot (k - d) \\ &= \sum_{j=0}^{k} 2^{k-j} \cdot j, \end{aligned}$$

missä on vaihdettu summausmuuttujaksi $j = k - d$. Seuraavan kalvon kaavanpyöritys osoittaa

$$\sum_{j=0}^{k} 2^{k-j} \cdot j \leq 2n,$$

joten algoritmin Build-Max-Heap aikavaativuus on $O\left(\sum_i h(i)\right) = O(n)$.

Kaavoja pyöritellään seuraavasti:

$$
\begin{aligned}
\sum_{j=0}^{k} 2^{k-j} \cdot j &< \sum_{j=0}^{\infty} 2^{k-j} \cdot j \\
&= 2^k \sum_{j=0}^{\infty} j \cdot \left(\frac{1}{2}\right)^j \\
&\le n \sum_{j=0}^{\infty} j \cdot \left(\frac{1}{2}\right)^j \\
&= 2n,
\end{aligned}
$$

missä on käytetty hyväksi tietoa $k \le \log_2 n$ ja summakaavaa

$$
\sum_{j=0}^{\infty} j \cdot x^j = \frac{x}{(1-x)^2} \qquad \text{kun } |\, x \,| < 1
$$

arvolla $x = 1/2$.

Edellisen kalvon summakaava voidaan johtaa lähtemällä geometrisen sarjan summasta

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x} \qquad \text{kun } |\,x\,| < 1.$$

Derivoidaan termeittäin:

$$D\frac{1}{1-x} = D\left(\sum_{j=0}^{\infty} x^j\right) = \sum_{j=0}^{\infty} Dx^j$$

eli

$$\frac{1}{(1-x)^2} = \sum_{j=0}^{\infty} j \cdot x^{j-1}.$$

Kertomalla puolittain luvulla x saadaan haluttu tulos.

Huom.: Koulumatematiikasta tiedämme, että äärellisen summan voi derivoida termeittäin eli $D(f+g) = Df + Dg$. Äärettömille sarjoille tämä ei aina päde, mutta tässä on kysymyksessä potenssisarja, jollaiset ovat tässä suhteessa ongelmattomia. Asiaa käsitellään tarkemmin analyysin peruskursseilla.

Datan tallettaminen prioriteettijonoon

Edellä on tarkasteltu prioriteettijonoa ja sen kekototeutusta vain avainten kannalta. Tarkastellaan nyt tilannetta, jossa prioriteettijonon alkioon kuuluu avaimen lisäksi muuta dataa.

Koska keossa alkioita joudutaan siirtelemään paljon, on yleensä tarkoituksenmukaista käyttää satellittitietueita: Taulukkoon A talletetaan kustakin prioriteettijonon tietueesta avain k ja osoitin x muuhun dataan. Tällöin itse datoja ei tarvitse yhtenään siirrellä paikasta toiseen.

Halutaan toteuttaa operaatiot

Heap-Insert(A, x, k): lisää kekoon osoittimen x osoittama tietue avaimen arvolla k

Heap-Max(A): palauta osoitin avaimeltaan suurimpaan tietueeseen

Heap-Del-Max(A): palauta osoitin avaimeltaan suurimpaan tietueeseen ja poista tietue

Heap-Increase-Key(A, x, k): kasvata x:n osoittaman tietueen avain arvoon k.

Näistä kolme ensimmäistä onnistuvat suoraviivaisesti edelläesitetyn pohjalta. Viimeinen vaatii jatkokehittelyä, koska nyt x ei kerro, missä kohdassa kekoa tietueen avain on.

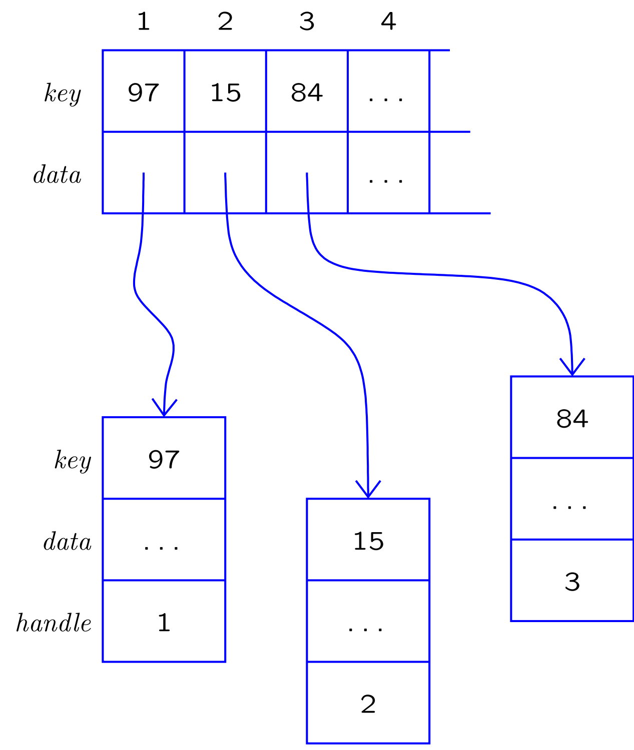
Ongelma ratkaistaan yksinkertaisesti siten, että osoittimen x osoittama tietue sisältää

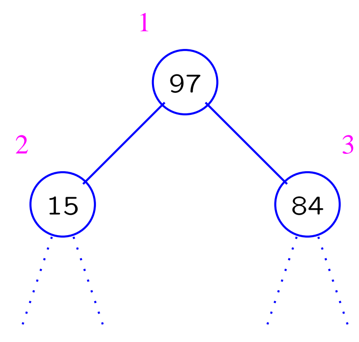
- avaimen $key[x]$ (voidaan jättää poiskin)
- muun datan $data[x]$
- kahvan $handle[x]$, joka on avaimen sijainti keossa.

Siis $key[A[handle[x]]] = key[x]$.

Nyt

- siirreltäessä avaimia keossa pitää aina päivittää vastaavat $handle$-arvot
- uusi Heap-Increase-Key(A, x, k) voidaan suoraan toteuttaa käyttämällä vanhaa algoritmia Heap-Increase-Key($A, handle[x], k$).





Keko, datatietueet ja kahvat

6. Järjestäminen

On annettu jono lukuja tai muita alkioita, joiden välille on määritelty suuruusjärjestys. Tehtävänä on saattaa alkiot suuruusjärjestykseen.

Tämä on eräs klassisimpia tietojenkäsittelyongelmia, sitä on analysoitu runsaasti ja ratkaisuja toteutettu kaikkiin mahdollisiin ympäristöihin.

Järjestäminen on hyvä perusesimerkki, jonka yhteydessä voidaan esitellä erilaisia algoritmisia tekniikoita ja kysymyksenasetteluja:

- hajoita ja hallitse -menetelmä algoritminsuunnittelussa
- pahin vs. keskimääräinen tapaus
- laskennan mallit ja alarajat.

Tiedämme ennestään, miten n lukua voidaan järjestää ajassa $O(n^2)$ esim. lisäys- tai kuplajärjestämisellä.

Järjestäminen voidaan tehdä ajassa $O(n \log n)$ käyttämällä tasapainoisia binäärihakupuita (esim. AVL-puita):

- lisää annetut n alkiota yksitellen hakupuuhun ajassa $O(\log n)$ per lisäys
- tulosta puu sisäjärjestyksessä ajassa $O(n)$.

Koska n voi olla hyvin suurin, vaativuusluokkien $O(n^2)$ ja $O(n \log n)$ on merkittävä. Tasapainoisissa binäärihakupuissa puissa kuitenkin

- vakiokertoimet ovat suuria ja
- tarvitaan lisämuistia osoittimia varten

joten tämä ratkaisu ei vielä ole tyydyttävä.

Kekojärjestäminen (heapsort)

Algoritmin perusidea tulee selvimmin esille seuraavasta:

```
Heap-Sort-0.1(A[1..n])
    Build-Max-Heap(A)
    for i ← n downto 1
        do B[i] ← Heap-Delete-Max(A)
    return B
```

Tämä selvästi palauttaa taulukossa B taulukon A alkiot kasvavassa järjestyksessä. Toteutuksessa on tehostamisen varaa.

Edellä taulukkoa B kasvatetaan samalla kun kekoa A pienennetään. Mitään erillistä taulukkoa B ei siis kannata varata, vaan voidaan tallettaa suoraan A:sta vapautuvaan osaan.

Ottamalla huomioon operaation Heap-Delete-Max toiminta saadaa algoritmi muotoon

Heap-Sort($A[1\,..\,n]$)

Build-Max-Heap(A)
while $\mathit{heap\text{-}size}[A] > 1$
do vaihda $A[1] \leftrightarrow A[\mathit{heap\text{-}size}[A]]$
$\mathit{heap\text{-}size}[A] \leftarrow \mathit{heap\text{-}size}[A] - 1$
Max-Heapify($A, 1$)

Algoritmi pitää siis yllä invarianttia

- osataulukko $A[\mathit{heap\text{-}size}[A] + 1\,..\,n]$ sisältää $n - \mathit{heap\text{-}size}[A]$ taulukon A suurinta lukua järjestyksessä
- osataulukossa $A[1\,..\,\mathit{heap\text{-}size}[A]]$ on loput luvut kekoehdon mukaisesti.

Kekojärjestämisen aikavaativuus on selvästi $O(n \log n)$:

- Build-Max-Heap vie ajan $O(n)$
- Max-Heapify vie ajan $O(\log n)$ ja suoritetaan $n - 1$ kertaa.

Jos Heapify toteutetaan iteratiivisesti (eikä rekursiivisena), sen tilavaativuus on vakio, kun syötteen tarvitsemaan tilaa ei lasketa. Siis koko algoritmi toimii vakiotyötilassa (in place).

Kekojärjestäminen on myös huomattavasti yksinkertaisempi toteuttaa kuin tasapainoinen hakupuu, ja vakiokertoimet ovat melko pieniä.

Lomitusjärjestäminen (merge sort)

Perusajatuksena on seuraava menetelmä järjestää (mahdollisesti vajaa) korttipakka:

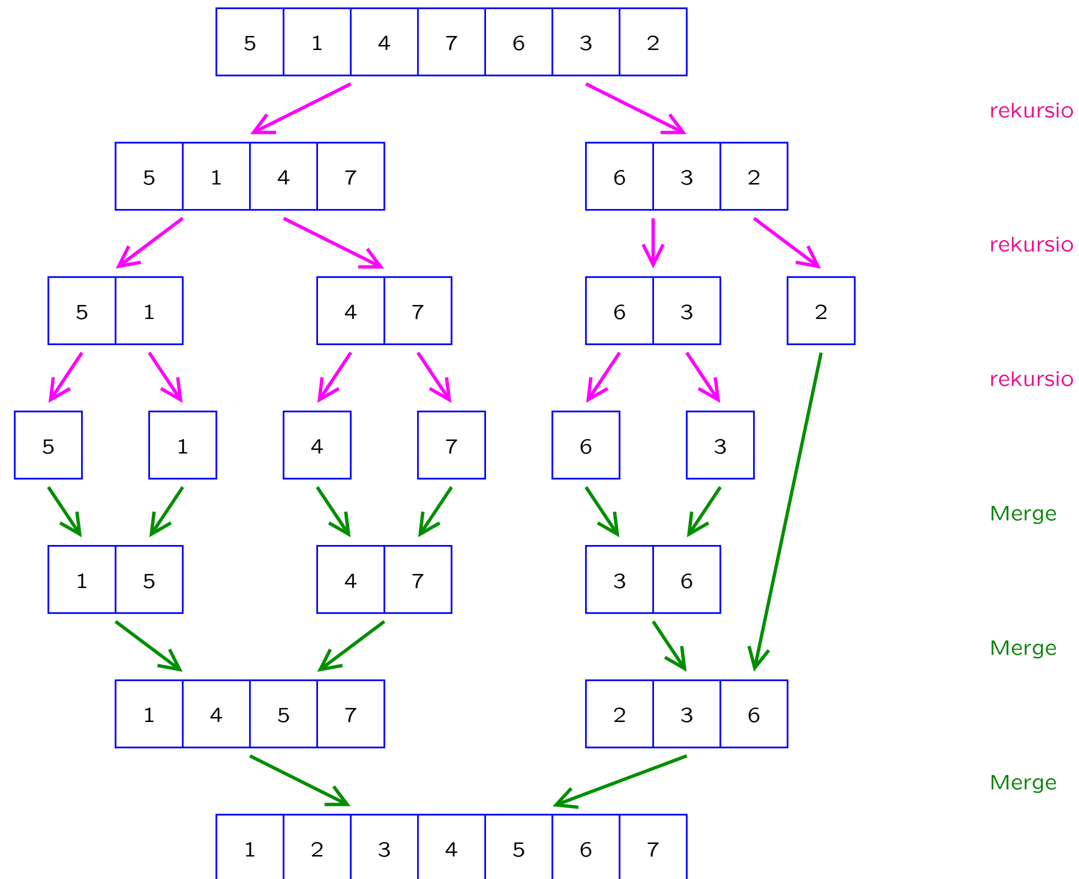
1. Jos pakassa on vain yksi kortti, älä tee mitään.
2. Muuten
   (a) Jaa pakka kahteen suunnilleen yhtä suureen osaan A ja B.
   (b) Järjestä osa A soveltamalla tätä menetelmää rekursiivisesti.
   (c) Järjestä osa B soveltamalla tätä menetelmää rekursiivisesti.
   (d) Lomita nyt järjestyksessä olevat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

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

Sovelletaan samaa menettelyä taulukon järjestämiseen. Kutsu Merge-Sort(A, p, r) järjestää osataulukon $A[p \,..\, r]$.

```
Merge-Sort(A, p, r)
    if p < r                          ▷ ainakin kaksi alkiota
       then q ← ⌊(p + r)/2⌋           ▷ puoliväli
            Merge-Sort(A, p, q)
            Merge-Sort(A, q + 1, r)
            Merge(A, p, q, r)
```

Tässä kutsu Merge(A, p, q, r) lomittaa jo järjestetyt osataulukot $A[p \,..\, q]$ ja $A[q + 1 \,..\, r]$ siten, että koko osataulukko $A[p \,..\, r]$ tulee järjestykseen. Palaamme sen toteutukseen kohta.



5 1 4 7 6 3 2
rekursio
5 1 4 7
6 3 2
rekursio
5 1
4 7
6 3
2
rekursio
5
1
4
7
6
3
Merge
1 5
4 7
3 6
Merge
1 4 5 7
2 3 6
Merge
1 2 3 4 5 6 7

Lomituslajittelu on esimerkki hajoita ja hallitse -tyyppisestä ongelmanratkaisusta:

1. Ongelma jaetaan pienempiin osaongelmiin.

2. Osaongelmat ratkaistaan rekursiivisesti.

3. Osaongelmien ratkaisut yhdistetään koko ongelman ratkaisuksi.

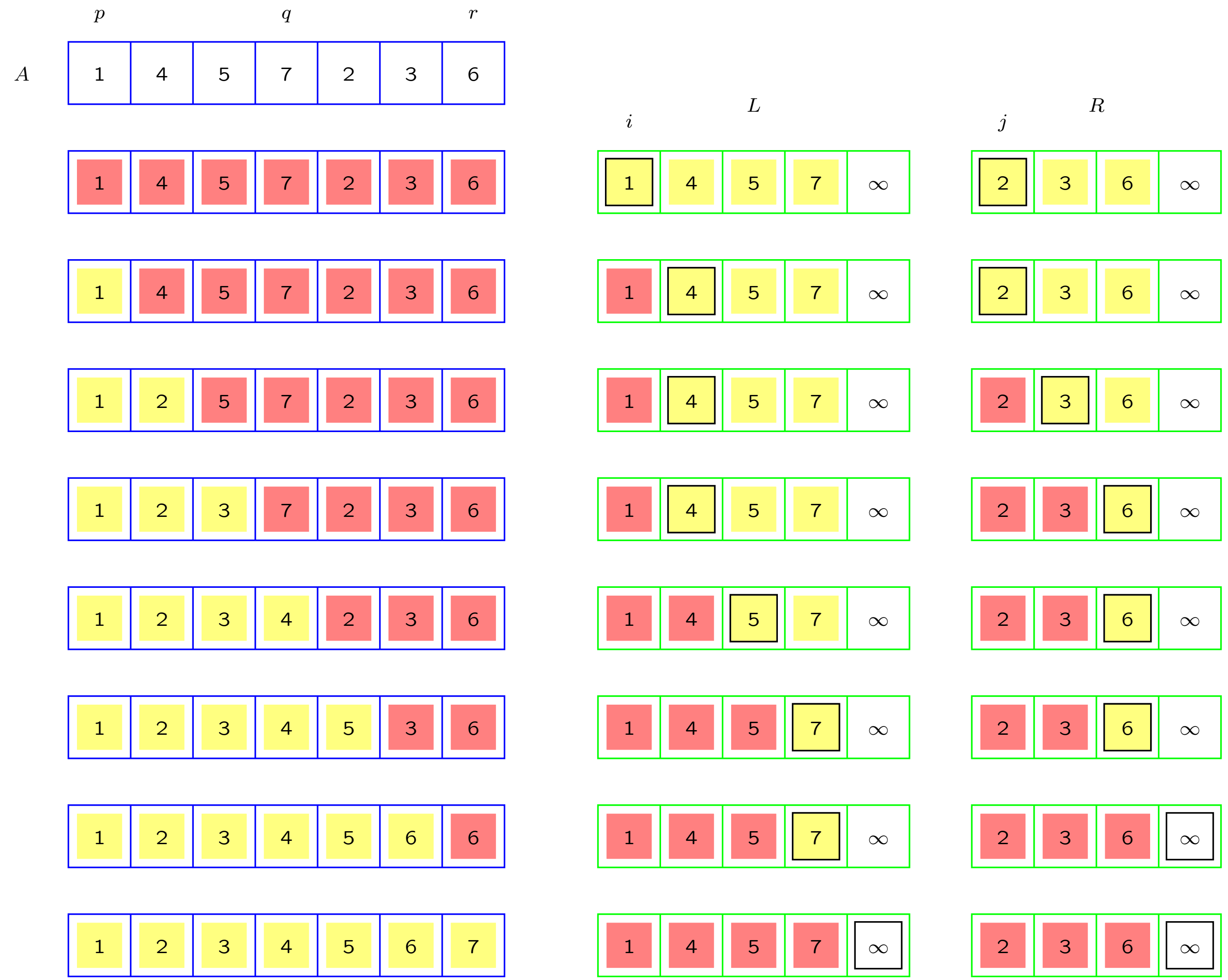
Laskennallisen tehokkuuden kannalta on yleensä tärkeää, että osaongelmista saadaan mahdollisimman samankokoiset. Palaamme asiaan Merge-Sortin aikavaativuusanalyysin yhteydessä.

Lomitus on helpointa tehdä kopioimalla ensin alku- ja loppupuoli aputaulukoihin L ja R. Lisäämällä aputaulukoihin viimeiseksi arvo ∞ vältetään aputaulukon lopun käsitteleminen erikoistapauksena.

```
Merge(p, q, r)
    Varaa aputaulukot L[1..q − p + 2] ja R[1..r − q + 1]
    L[1..q − p + 1] ← A[p..q]        ▷ Kopioi taulukon alkupuolisko
    R[1..r − q] ← A[q + 1..r]        ▷ Kopioi taulukon loppupuolisko
    L[q − p + 2] ← R[r − q + 1] ← ∞
    i ← j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```

Aputaulukot vievät $(q - p + 2) + (r - q + 1) = r - p + 3$ muistipaikkaa, joten tilavaativuus on $O(\ell)$, missä $\ell = r - p + 1$ on lomitettavien taulukoiden yhteispituus.

Myös aikavaativuus on selvästi $O(\ell)$.


p
q
r
A
1 4 5 7 2 3 6
i
L
j
R
1 4 5 7 ∞
2 3 6 ∞
1 2 3 4 5 6 7

Lomitusjärjestämisen aikavaativuus

Olkoon $T(n)$ algoritmin Merge-Sort(p, r) pahimman tapauksen aikavaativuus, kun $n = r - p + 1$ on järjestettävän (osa)taulukon koko.

```
Merge-Sort(A, p, r)

1   if p < r
2      then q ← ⌊(p + r)/2⌋
3           Merge-Sort(A, p, q)
4           Merge-Sort(A, q + 1, r)
5           Merge(A, p, q, r)
```

Nyt

- alustukset, rivin 1 testi ja poistuminen vievät jonkin vakioajan a_1
- rivin 2 sijoitus ja aliohjelmakutsujen alustukset jne. vievät jonkin vakioajan a_2
- rivin 3 rekursiivinen kutsu vie ajan $T(\lceil n/2 \rceil)$
- rivin 4 rekursiivinen kutsu vie ajan $T(\lfloor n/2 \rfloor)$
- rivin 5 Merge vie ajan $a_3 + a_4 n$ joillain vakioilla a_3 ja a_4.

Saamme siis aikavaativuudelle palautuskaavan eli rekursioyhtälön

$$\begin{aligned} T(1) &= a_1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + a_4 n + a_1 + a_2 + a_3 \qquad \text{kun } n > 1. \end{aligned}$$

Muodostamme tästä yksinkertaistetun version

$$\begin{aligned} T(1) &= c \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn \qquad \text{kun } n > 1. \end{aligned}$$

Valinnalla $c = a_1 + a_2 + a_3 + a_4$ yksinkertaistettu yhtälä antaa ylärajan alkuperäiselle funktiolle T. Vastaavasti valitsemalla $c = \min\{a_1, a_4\}$ saamme alarajan. Kuten on odotettavissa, nämä ylä- ja alaraja ovat vakiokertoimen päässä toisistaan. Siis yksinkertaistettu yhtälön antaa Θ-merkinnän tarkkuudella oikean vastauksen alkuperäiseen yhtälöön.

Yksinkertaistamme yhtälöä edelleen olettamalla, että $n = 2^p$ jollakin $p \in \mathbb{N}$. Tällöin jakolaskut menevät tasan ja yhtälö tulee muotoon

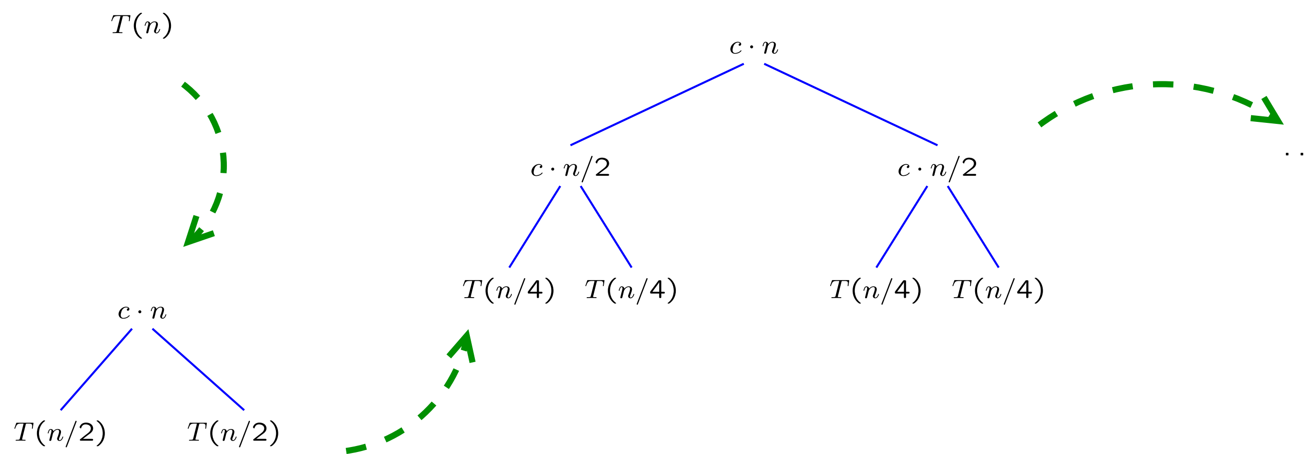
$$\begin{aligned} T(1) &= c \\ T(n) &= 2T(n/2) + cn \qquad \text{kun } n = 2^p \text{ missä } p \geq 1. \end{aligned}$$

Tarkastelemme myöhemmin, miten tapaus $n \neq 2^p$ käsitellään.

Ratkaisemme tämän rekursioyhtälön muodostamalla rekursiopuun:

- puun juureen tulee kutsu Merge-Sort$(A, 1, n)$
- solmun Merge-Sort(A, p, r) lapsiin tulee vastaavat rekursiiviset kutsut Merge-Sort(A, p, q) ja Merge-Sort$(A, q + 1, r)$
- merkitsemme solmuun näkyviin vastaavan kutsun aikavaativuuden paitsi rekursiivisiin kutsuihin kuluvaa aikaa.

Puun solmuihin tulevien aikojen summa on koko algoritmin aikavaativuus.



Rekursiopuun muodostuminen

Rekursiopuun tasolla k

- kussakin solmussa on aika $cn/2^k$
- solmuja on 2^k kappaletta

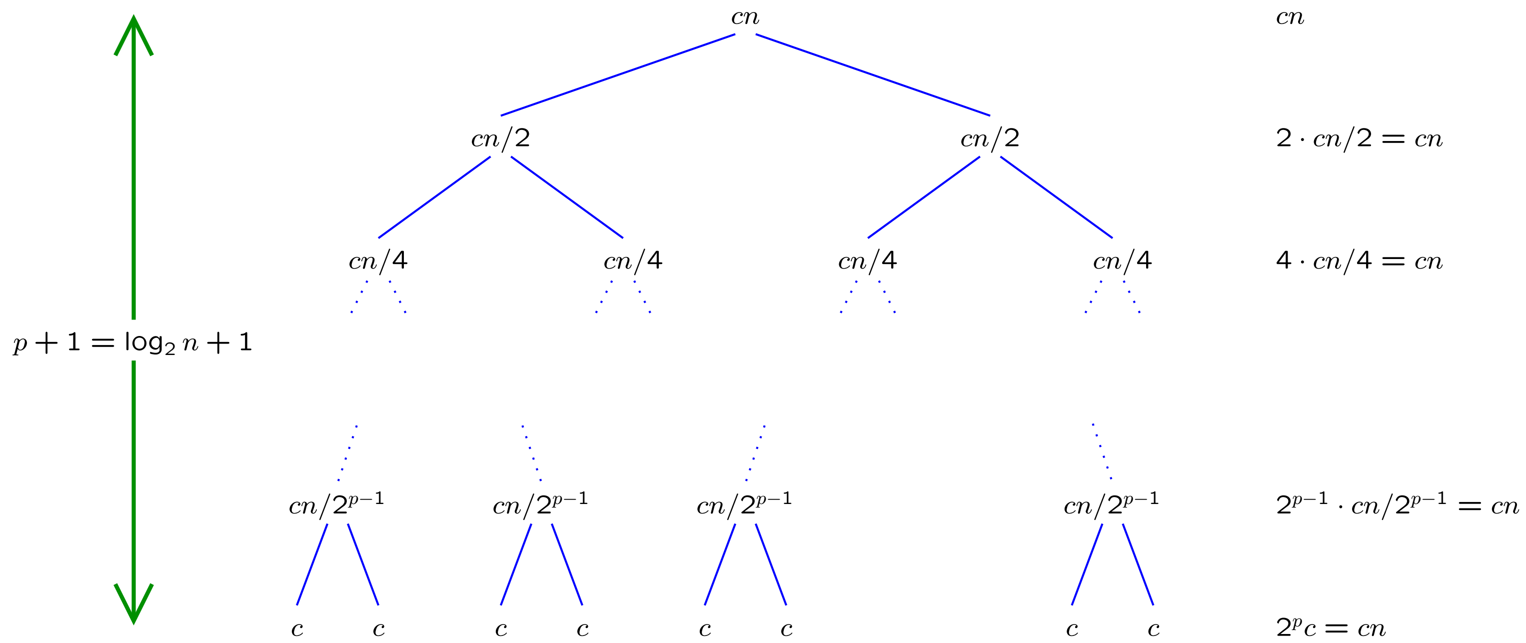
$\Rightarrow$ tason solmujen yhteisaika on $2^k \cdot cn/2^k = cn$.

Viimeinen taso on se, jolla $n/2^k = 1$ eli $k = \log_2 n = p$.

Siis kaikkiaan tasoja on $p + 1$ (koska juuri on tasolla 0). Koko puuhun tulevien aikojen summa on $(p + 1) \cdot cn = cn(\log_2 n + 1)$.

Saamme ratkaisuksi

$$T(n) = cn(\log_2 n + 1) = \Theta(n \log n).$$



Rekursiopuu

Entä jos n ei ole muotoa 2^p millään p? Tällöin jollakin $p \geq 1$ pätee $2^p < n < 2^{p+1}$. Helposti nähdään, että T on kasvava funktio. Siis

$$\begin{aligned} T(n) &\leq T(2^{p+1}) \\ &= c2^{p+1}(\log_2 2^{p+1} + 1) \\ &< c \cdot 2n(\log_2(2n) + 1) \\ &\leq 2cn(\log_2 n + 2), \end{aligned}$$

koska $2^{p+1} < 2n$. Vastaavasti

$$\begin{aligned} T(n) &\geq T(2^p) \\ &= c2^p(\log_2 2^p + 1) \\ &> c \cdot (n/2)(\log_2(n/2) + 1) \\ &= \frac{c}{2} n \log_2 n. \end{aligned}$$

Siis $T(n) = \Theta(n \log n)$ pätee ilman rajoituksia.

Iteratiivinen lomitusjärjestäminen

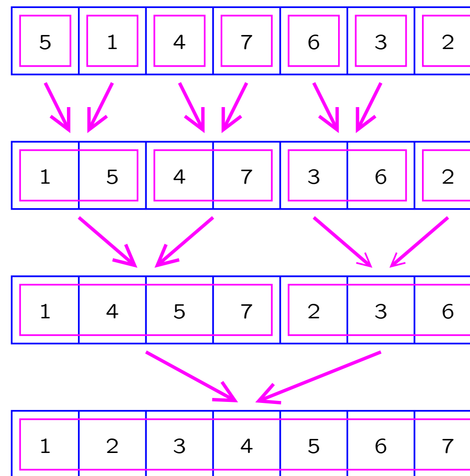
Edellä esitetyssä versiossa rekursiiviset Merge-Sort-kutsut vain jakavat taulukkoa pienempiin osataulukoihin. Tämä on turhaa työtä, koska tiedämme ilman muuta, mitä osataulukoita tarvitaan:

- ensin lomitetaan $A[1]$ ja $A[2]$; sekä $A[3]$ ja $A[4]$; jne.
- sitten lomitetaan $A[1\,..\,2]$ ja $A[3\,..\,4]$; sekä $A[5\,..\,6]$ ja $A[7\,..\,8]$; jne.
- ...
- lomitetaan $A[1\,..\,2^{i-1}]$ ja $A[2^{i-1}+1\,..\,2^i]$; sekä $A[2^i+1\,..\,2^i+2^{i-1}]$ ja $A[2^i+2^{i-1}+1\,..\,2\cdot 2^i]$; jne.
- ...
- lomitetaan $A[1\,..\,n/2]$ ja $A[n/2+1\,..\,n]$ (olettaen yksinkertaisuuden vuoksi $n=2^p$).

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

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

```
Merge-Sort-2(A[1..n])
    u ← 1                        ▷ lomitettavien lohkojen pituus
    while u < n
        do v ← 1                 ▷ lomitettavan lohkoparin alkukohta
           while v + u < n
               do p = v
                  q = v + u − 1
                  r = min { q + u, n }
                                 ▷ viimeinen lohko voi olla alimittainen
                  Merge(A, p, q, r)
                  v ← v + 2u
           u ← 2u
```



Lomitusjärjestäminen ilman rekursiota

Selvästi Merge-kutsut dominoivat algoritmin aikavaativuutta. Merge-kutsut ovat samat kuin rekursiivisessa toteutuksessa, ja niihin menevä aika on tietysti myös sama. (Vaihtoehtoisesti sen voisi laskea suoraan samanlaisella puulla kuin rekursion tapauksessa.) Siis aikavaativuus on edelleen $\Theta(n \log n)$.

Tilavaativuutta dominoi suurin Merge-kutsussa tarvittava aputaulukko, jonka koko on $n/2$. Siis tilavaativuus on $\Theta(n)$.

Lomitusjärjestäminen listoilla

Todetaan ensin, että kaksi järjestyksessä olevaa listaa voidaan lomittaa helposti. Tähän ei edes tarvita yleisiä listaoperaatioita, vaan jono riittää:

```
List-Merge(B, L, R)
    ▷ lomitetaan jonot L ja R jonoksi B
    Enqueue(L, ∞)          ▷ loppumerkki kumpaankin syötejonoon
    Enqueue(R, ∞)
    while First(L) ≠ ∞ or First(R) ≠ ∞
        do if First(L) ≤ First(R)
            then Enqueue(B, Dequeue(L))
            else  Enqueue(B, Dequeue(R))
```

Tässä First(Q) palauttaa jonon Q kärjessä olevan alkion, mutta ei poista sitä jonosta.

Lomituksen tulos liitetään jonon B häntään. Jonossa B mahdollisesti jo olleet alkiot säilyvät siellä.

Linkitettyä listaa ei voi puolittaa pelkästään indeksejä laskemalla kuten osataulukkoa. Seuraava alirutiini erottaa jonon A kärjestä u alkiota ja palauttaa niistä muodostetun uuden listan. Alkioiden järjestys säilyy. Jos jonossa A on alle u alkiota, koko lista A siirretään.

```
Take(A, u)
    C ← tyhjä jono
    t ← 0
    while t < u and not Empty(A)
        do Enqueue(C, Dequeue(A))
           t ← t + 1
    return C
```

Voimme nyt esittää lomitusjärjestämisen seuraavasti:

```
List-Merge-Sort(A)
    n ← jonon A pituus
    u ← 1                          ▷ lomitettavien listojen pituus
    while u < n
          do B ← tyhjä jono
             repeat
                   L ← Take(A, u)
                   R ← Take(A, u)
                   List-Merge(B, L, R)
               until Empty(A)
             A ← B
             u ← 2u
```

Myös lomitusjärjestämisen listaversion aikavaativuus on $O(n \log n)$, mikä voidaan todeta kuten iteratiivisen taulukkoversion tapauksessa.

Lomittaminen voidaan nyt hoitaa osoittimia siirtelemällä. Siis lomittamiseen ei tarvita ylimääräistä talletusaluetta. Lisätilan tarve alkuperäisen syötejonon lisäksi on vakio. Pitää kuitenkin ottaa huomioon, että taulukkototeutukseen verrattuna tässä osoittimet vievät lisätilaa.

Vastaavalla lomitusidealla voidaan järjestää myös peräkkäistiedostoja.

Kaikki edellä esitetyt lomitusjärjestämisen versiot ovat vakaita (stable):

- Oletetaan, että järjestettävät arvot ovat avaimia, joihin kuhunkin liittyy muutakin tietoa.
- Jos kahdella eri tiedolla on sama avain, niin vakaa järjestämisalgoritmi ei muuta näiden kahden tiedon keskenäistä järjestystä.
- Vakaus on toisinaan tärkeää, esim. järjestettäessä peräkkäin usean avaimen suhteen.

Pikajärjestäminen (quicksort)

Pikajärjestäminen on toinen hajoita ja hallitse -menetelmä. Se on käytännössä suosittu, koska

- aikavaativuus on keskimäärin $O(n \log n)$
- vakiokertoimet ovat pieniä
- työtilan tarve on vain $O(\log n)$.

Haittapuolina

- pahimman tapauksen aikavaativuus $\Theta(n^2)$
- käyttäytyminen riippuu syötteestä monimutkaisella tavalla.

Pikajärjestämisen toiminta-ajatus on seuraava:

Hajoita: Valitse taulukosta $A[1\,..\,n]$ jakoalkio (pivot) a. Ryhmittele taulukko uudestaan siten, että

$$\begin{array}{lcll} A[i] & \leq & a & \text{kun } 1 \leq i \leq q \\ A[j] & \geq & a & \text{kun } q+1 \leq j \leq n. \end{array}$$

Ratkaise osaongelmat: Järjestä osataulukot $A[1\,..\,q]$ ja $A[q+1\,..\,n]$ soveltamalla algoritmia rekursiivisesti.

Yhdistäminen: Mitään ei enää tarvitse tehdä, taulukko on jo järjestyksessä.

Lomitusjärjestämiseen verrattuna hajoitusvaihe on työläämpi. Näemme pian, miten se toteutetaan ajassa $O(n)$. Lisäksi jakokohta q ei ole ennalta tiedossa, vaan määräytyy taulukon sisällön perusteella. Erityisesti osataulukoista voi joillain syötteillä tulla hyvinkin eri kokoisia, mikä on paha tehokkuuden kannalta. (Vrt. lomitusjärjestämisen tasajako vakioajassa.)

Vastineeksi hajoitusvaiheessa tehdystä lisätyöstä osaratkaisujen yhdistäminen on triviaalia. (Vrt. lomitus $O(n)$.)

Edellä esitetty hajoitusvaihe voidaan toteuttaa monella tavalla. Tarkastelemme tätä kohta. Olkoon Partition jokin hajoituksen toteuttava algoritmi.

Taulukon $A[1..n]$ pikajärjestäminen tapahtuu kutsulla Quicksort$(A, 1, n)$, missä Quicksort(A, p, r) järjestää osataulukon $A[p..r]$:

```
Quicksort(A, p, r)
    if p < r
       then q ← Partition(A, p, r)
            Quicksort(A, p, q)
            Quicksort(A, q + 1, r)
```

Valitsemme tarkasteltavaksi seuraavan yksinkertaisen version Partition-algoritmista:

```
Partition(A, p, r)
    a ← A[p]
    i ← p − 1
    j ← r + 1
    while i < j
        do repeat i ← i + 1 until A[i] ≥ a
           repeat j ← j − 1 until A[j] ≤ a
           if i < j then vaihda A[i] ↔ A[j]
    return j
```

- Jakoalkioksi valitaan tässä taulukon ensimmäinen alkio.
- Muutkin jakoalkiot olisivat mahdollisia; palaamme tähän jatkossa.
- Yhdessä while-silmukan iteraatiossa etsitään
  - vasemmanpuoleisin alkio $A[i]$, joka voisi olla loppuosassa
  - oikeanpuoleisin alkio $A[j]$, joka voisi olla alkuosassa.

Algoritmin oikeellisuuden toteamiseksi havaitaan ensin, että Partitition-algoritmissa pätee silmukkainvariantti

$$A[k] \leq a \text{ kun } p \leq k \leq i, \text{ ja } A[k] \geq a \text{ kun } j \leq k \leq r$$

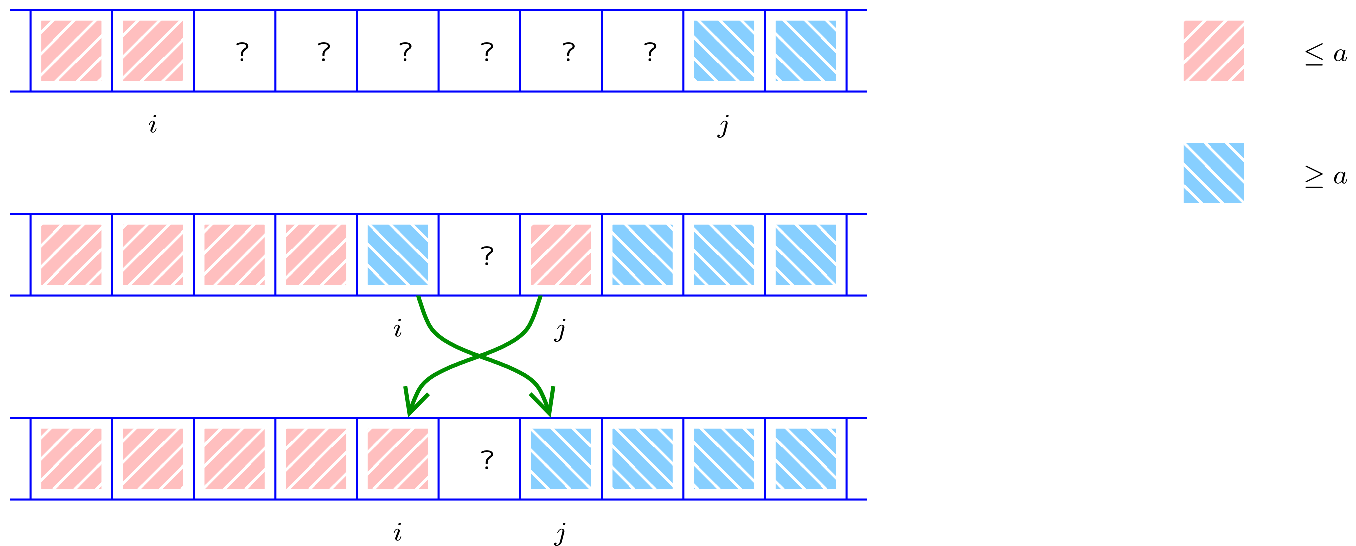
paitsi ehkä viimeisen iteraation jälkeen.

Aluksi $i < p$ ja $j > r$, joten invariantti pätee triviaalisti.

Kun "repeat i" -silmukka on suoritettu, niin $A[k] \leq a$ kun $p \leq k < i$, ja $A[i] \geq a$.

Samoin kun "repeat j" -silmukka on suoritettu, niin $A[k] \geq a$ kun $j < k \leq r$, ja $A[j] \leq a$.

Jos silmukka ei pääty, vaihdetaan $A[i] \leftrightarrow A[j]$, ja invariantti tulee taas voimaan.



while-silmukan iteraatio kun suoritus ei pääty

Tarkastellaan nyt silmukan viimeistä iteraatiota, jonka lopuksi siis $j \leq i$. Väitämme, että $q = j$ on sopiva jakokohta eli

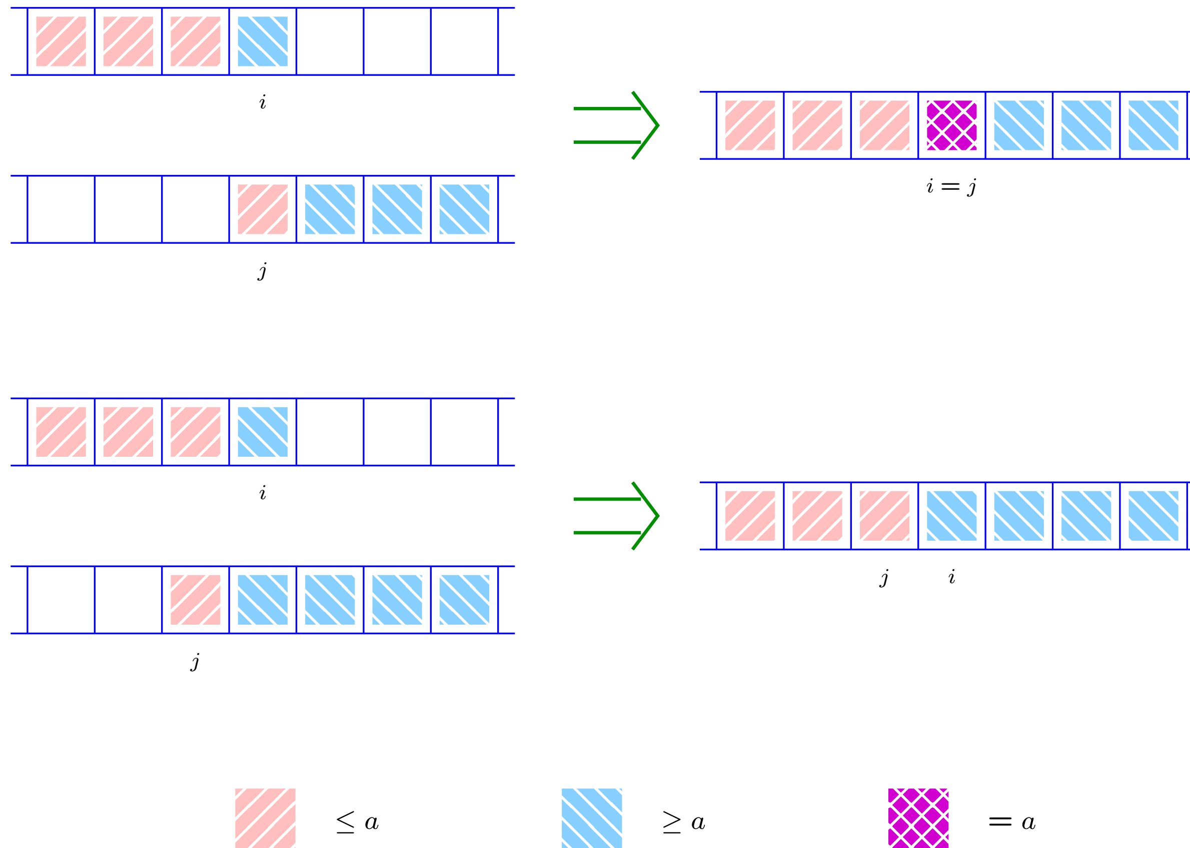
$A[k] \leq a$ kun $p \leq k \leq j$, ja $A[k] \geq a$ kun $j + 1 \leq k \leq r$.

Kuten edellä,

- kun "repeat i" -silmukka on suoritettu, niin $A[k] \leq a$ kun $p \leq k < i$, ja $A[i] \geq a$.
- kun "repeat j" -silmukka on suoritettu, niin $A[k] \geq a$ kun $j < k \leq r$, ja $A[j] \leq a$.

Siis väitteen jälkimmäinen osa pätee ilman muuta.

Koska $i \geq j$ ja $A[k] \leq a$ kun $p \leq k < i$, niin $A[k] \leq a$ pätee kun $k < j$. Lisäksi $A[j] \leq a$, joten myös väitteen ensimmäinen osa pätee.



while-silmukan viimeinen iteraatio (tapaukset $i = j$ ja $i = j + 1$)

Olemme todenneet, että Partition(A, p, r) toimii oikein eli organisoi taulukon ja palauttaa arvon q siten, että

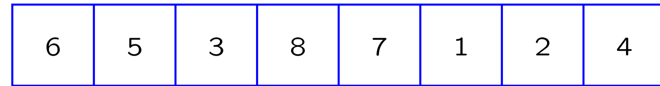
$$A[k] \leq a \text{ kun } p \leq k \leq q, \text{ ja } A[k] \geq a \text{ kun } q + 1 \leq k \leq r$$

Aikaa kuluu

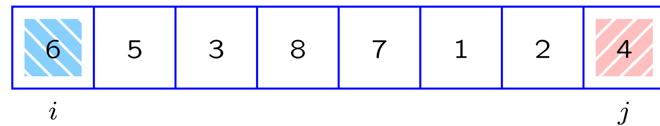
- vakiomäärä alustuksiin
- vakiomäärä jokaista indeksin kasvatusta $i \leftarrow i + 1$ kohti
- vakiomäärä jokaista indeksin pienennystä $j \leftarrow j - 1$ kohti
- vakiomäärä iteraatiota kohti muihin toimiin while-silmukassa.

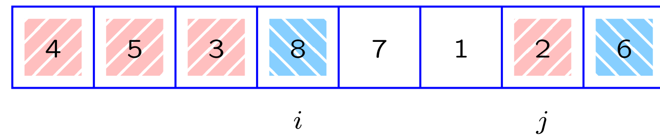
Karkeana ylärajana kumpaakin indeksiä voidaan muuttaa korkeintaan $r - p + 2$ kertaa. Lisäksi jokainen while-iteraatio muuttaa kumpaakin indeksiä, joten tämä on samalla yläraja while-iteraatioiden määrälle.

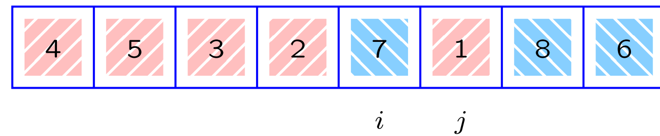
Kokonaisaikavaativuus on siis $\Theta(r - p + 1)$ eli lineaarinen osataulukon pituuden suhteen.

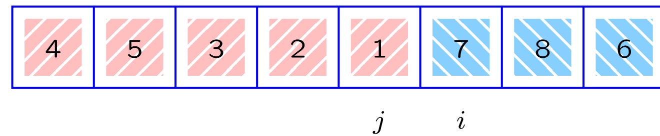


$a = A[1] = 6$

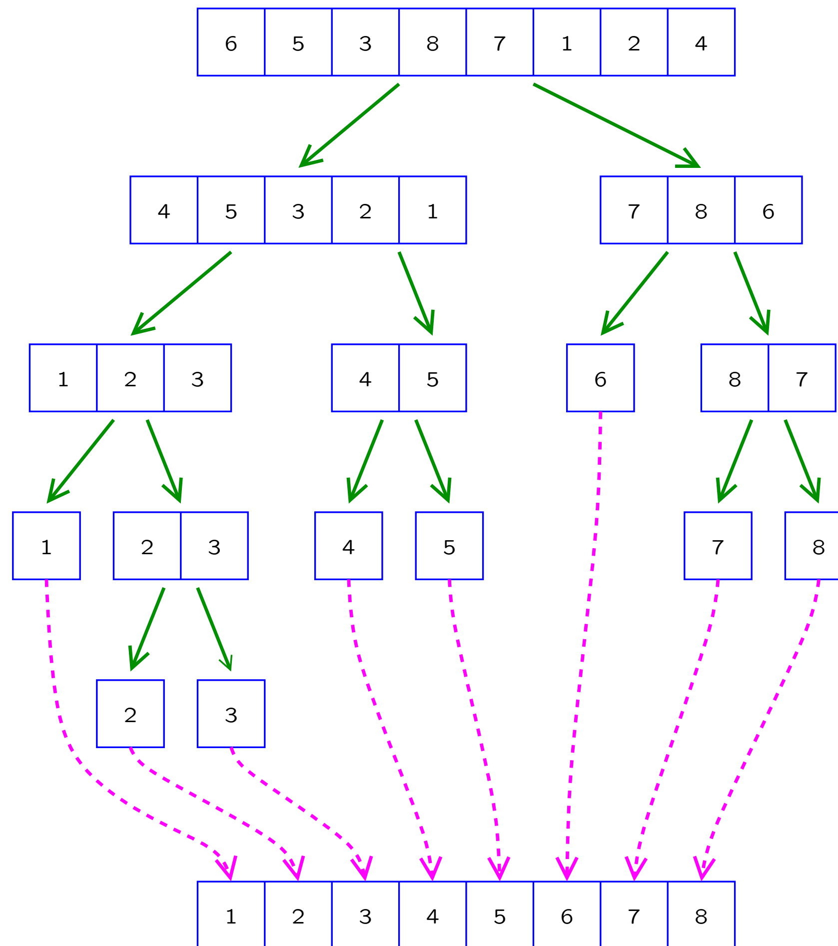








Kutsun Partition$(A, 1, 8)$ suoritus kun $A = [6, 5, 3, 8, 7, 1, 2, 4]$



Kutsun Quicksort($A, 1, 8$) suoritus kun $A = [6, 5, 3, 8, 7, 1, 2, 4]$

Pikajärjestämisen aikavaativuus

Olkoon $T(n)$ kutsun Quicksort(A, p, r) aikavaativuus, kun $n = r - p + 1$ on järjestettävän osataulukon koko:

```
Quicksort(A, p, r)
1   if p < r
2      then q ← Partition(A, p, r)
3           Quicksort(A, p, q)
4           Quicksort(A, q + 1, r)
```

Koska Partition vie ajan $O(n)$, lähtökohtana on rekursioyhtälö

$$\begin{aligned} T(1) &= c \\ T(n) &= T(k) + T(n-k) + cn \qquad \text{kun } n > 1, \end{aligned}$$

missä rekursiivisesti järjestettävien taulukoiden koot ovat $k = r - q + 1$ ja $n - k = q - p$. (Tässä on tehty vastaavat yksinkertaistukset kuin lomitusjärjestämisen analyysissä.) Ongelmana on, että k vaihtelee syötteen mukaan, joten tästä ei suoraan päästä eteenpäin.

Pahimmalle tapaukselle voidaan kirjoittaa ylärajaksi

$$
\begin{aligned}
T_{\text{max}}(1) &= c \\
T_{\text{max}}(n) &= \max_{1 \le n \le k} \left(T_{\text{max}}(k) + T_{\text{max}}(n-k) \right) + cn \qquad \text{kun } n > 1.
\end{aligned}
\tag{1}
$$

On kohtuullisen helppo osoittaa, että pahin tapaus syntyy, kun $k = 1$ (tai symmetrisesti $k = n - 1$):

$$
\begin{aligned}
T_{\text{max}}(1) &= c \\
T_{\text{max}}(n) &= T_{\text{max}}(1) + T_{\text{max}}(n-1) + cn \qquad \text{kun } n > 1.
\end{aligned}
\tag{2}
$$

Intuitiivisesti pahin tapaus syntyy, kun jako on mahdollisimman epätasainen.

Täsmällisemmin tämä todistettaisiin ratkaisemalla (2) ja tarkistamalla, että saatu ratkaisu toteuttaa alkuperäisen yhtälön (1). Sivuutamme jatkossa tarkistusvaiheen.

Yhtälö (2) voidaan helposti ratkaista purkamalla:

$$
\begin{aligned}
T_{\max}(n) &= T_{\max}(1) + T_{\max}(n-1) + cn \\
&= T_{\max}(1) + T_{\max}(1) + T_{\max}(n-2) + c(n-1) + cn \\
&= 3 \cdot T_{\max}(1) + T_{\max}(n-3) + c\left((n-2) + (n-1) + n\right) \\
&= \ldots \\
&= (n-1) \cdot T_{\max}(1) + T_{\max}(1) + c\sum_{i=2}^{n} i \\
&= cn + c(n-1) \cdot \frac{n+2}{2} \\
&= \Theta(n^2).
\end{aligned}
$$

Siis pahimman tapauksen aikavaativuus on neliöllinen.

Huom. Edellä oletettu maksimaalisen epätasainen jako $k = 1$ on todella mahdollinen. Edellä esitetylle Partition-versiolle se syntyy, jos syöte on jo valmiiksi järjestyksessä.

Parhaassa tapauksessa jako on mahdollisimman tasainen:

$$\begin{aligned} T_{\text{min}}(1) &= c \\ T_{\text{min}}(n) &= T_{\text{min}}(\lceil n/2 \rceil) + T_{\text{min}}(\lfloor n/2 \rfloor) + cn \qquad \text{kun } n > 1. \end{aligned}$$

Tämä on sama rekursioyhtälö kuin lomitusjärjestämiselle, joten parhaan tapauksen aikavaativuus on $\Theta(n \log n)$.

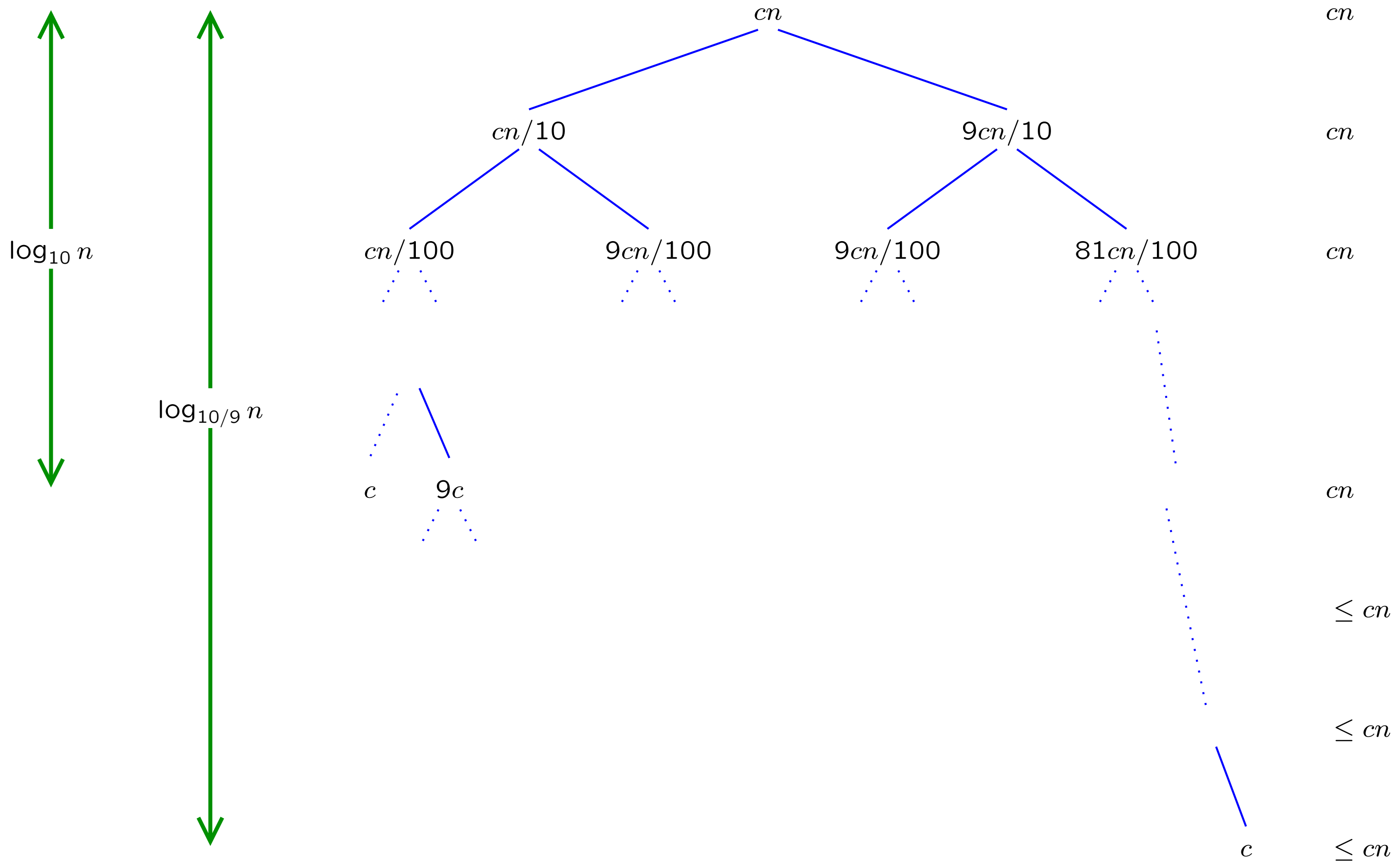
Keskimääräisen tapauksen aikavaativuus on sekin $\Theta(n \log n)$, mikä on paljon kiinnostavampaa kuin paras tapaus. Keskimääräisellä tapauksella tarkoitetaan aikavaativuuden odotusarvoa, kun syötteen kaikki järjestykset ovat yhtä todennäköisiä. Sivuutamme todistuksen (ks. esim. Cormen et al. luku 7.4).

Keskimääräisen tapauksen valaisemiseksi kuvitellaan tilanne, jossa Partition tuottaa aina kokoa $\frac{9}{10}n$ ja $\frac{1}{10}n$ olevat osataulukot:

$$\begin{aligned} T(1) &= c \\ T(n) &= T(9n/10) + T(n/10) + cn \qquad \text{kun } n > 1 \end{aligned}$$

(sivuutamme tässä pyöristysongelmat). Ratkaisu nähdään helpoimmin rekursiopuusta:

- Syvyys on
$$d = \log_{10/9} n = \frac{\log_2 n}{\log_2(10/9)} \approx 6{,}58 \log_2 n,$$
sillä rekursion tasolla d suurimmankin osataulukon koko on $n \cdot (9/10)^d = n/(10/9)^d = n/n = 1$.
- Kunkin tason vaatima aika on korkeintaan cn (vähemmän, jos syvyys on yli $\log_{10} n$, jolloin pienet osataulukot alkavat tyhjentyä).
- Siis aikavaativuus on
$$T(n) \leq (d+1)cn = cn\left(\frac{1}{\log_2(10/9)}\log_2 n + 1\right) = \Theta(n \log n).$$



cn
$cn/10$
$9cn/10$
$cn/100$
$9cn/100$
$9cn/100$
$81cn/100$
c
$9c$
c
$\log_{10} n$
$\log_{10/9} n$
cn
cn
cn
cn
$\leq cn$
$\leq cn$
$\leq cn$

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

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

Yleensä jakojen tasaisuus tietenkin vaihtelee. Jos esim. joka toinen jako on $\frac{1}{n} : \frac{n-1}{n}$ ja joka toinen $\frac{1}{10} : \frac{9}{10}$, niin edelliseen verrattuna rekursiopuuhun tulee kaksinkertainen määrä tasoja, joista joka toisella tasolla laskenta ei "etene". Tämä tuottaa silti vain vakiokertoimen 2 aikavaativuuteen.

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

Pikajärjestämisen tilavaativuus

Koska Quicksort ja Partition vaativat vain vakiomäärän apumuuttujia, pikajärjestämisen työtilavaativuus on verrannollinen rekursion maksimisyvyyteen. Pahimmassa tapauksessa se on edellä esitetyllä toteutuksella $\Theta(n)$. Tämä voidaan kuitenkin parantaa arvoon $O(\log n)$ toteutusta optimoimalla.

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

Quicksort2(A, p, r)

while $p < r$
  do $q \leftarrow$ Partition(A, p, r)
    Quicksort2(A, p, q)
    ▷ Quicksort($A, q + 1, r$) korvataan iteraatiolla:
    $p \leftarrow q + 1$

Monet kääntäjät tekevät tällaisen optimoinnin automaattisesti. Tämä ei vielä muuta pahimman tapauksen vaativuutta, koska pahimmassa tapauksessa rekursiivisesti käsiteltävä osataulukko $A[p \,..\, q]$ on edelleen kokoa $n - 1$.

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

```
Quicksort3(A, p, r)
    while p < r
        do q ← Partition(A, p, r)
           if q − p + 1 < r − q
              then Quicksort3(A, p, q)      ▷ alkuosa on pienempi
                   p ← q + 1                ▷ Quicksort(A, q + 1, r) iteraatiolla
              else Quicksort3(A, q + 1, r)
                   r ← q                    ▷ Quicksort(A, p, q) iteraatiolla
```

Nyt pahin tapaus onkin tasajako, ja tilavaativuus toteuttaa

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

eli $S(n) = O(\log n)$ pahimmassa tapauksessa ja siis tietysti myös keskimäärin.

Pikajärjestäminen käytännössä

Pikajärjestämisen vakiokertoimet ovat pieniä, mutta pienillä n hyvä $O(n^2)$-algoritmi, kuten erityisesti lisäysjärjestäminen, voittaa sen silti. Rekursio kannattaa siis pysäyttää ennen kuin ehditään kokoa 1 oleviin osataulukoihin:

```
Quicksort(A, p, r)
    if p < 20
       then InsertionSort(A[p..r])
       else  q ← Partition(A, p, r)
             Quicksort(A, p, q)
             Quicksort(A, q + 1, r)
```

Luvun 20 tilalla voi tietysti olla jokin muukin arvo k. Keskimääräinen aikavaativuus on asymptoottisesti $O((n/k)k^2 + n \log(n/k)) = O(n \log n)$ kun k on vakio. Tässä $O(k^2)$ on k:n mittaisen osataulukon järjestämiseen kuluva aika, n/k näiden osataulukoiden lukumäärä, $\log(n/k)$ rekursiopuun korkeus ja $O(n)$ yhden tason kokonaisaika kuten edellä (s. 356).

Edellä esitetyn Partition-algoritmin pahin tapaus on, jos syöte on jo valmiiksi järjestyksessä.

Valmiiksi järjestetty syöte on tietysti helppo käsitellä erikoistapauksena. Joissain sovelluksissa voi kuitenkin esiintyä syötteitä, jotka ovat "melkein järjestyksessä", ja em. Partition toimii silloinkin huonosti.

Vähemmän herkkä versio saadaan valitsemalla jakoalkioksi a suuruusjärjestyksessä keskimmäinen kolmesta alkioista

- $A[p]$ (taulukon alku)
- $A[\lfloor p+r \rfloor /2]$ (taulukon keskikohta)
- $A[r]$ (taulukon loppu).

Tällekin voidaan löytää oma pahin tapauksensa $\Omega(n^2)$, mutta sen esiintyminen käytännössä olisi jo yllättävämpää.

Varma menettely taata rekursion syvyys $O(\log n)$ olisi valita jakoalkioksi a lukujen $A[p], A[p+1], \ldots, A[r]$ mediaani eli suuruusjärjestyksessä keskimmäinen ($\lfloor(r+p)/2\rfloor$:s) alkio.

Yllättäen tämä on mahdollista tehdä ajassa $O(r-p+1)$ eli vakiokerrointa vaille samassa ajassa kuin edellä esitetyt helpot Partition-versiot. Tämä antaa pikajärjestämisestä version, jonka aikavaativuus on $O(n \log n)$ myös pahimmassa tapauksessa.

Etsittäessä mediaani ajassa $O(n)$ vakiokerroin on kuitenkin suuri ja algoritmi monimutkainen. Tämä menetelmä ei siis käytännössä ole suositeltava, koska se kadottaa pikajärjestämisen pääasialliset edut esim. kekojärjestämiseen verrattuna.

Kompromissi edellisten väliltä on valita jakoalkioksi satunnainen alkio osataulukosta $A[p\,..\,r]$:

Randomized-Partition(A, p, r)

$i \leftarrow$ Random(p, r)
vaihda $A[p] \leftrightarrow A[i]$
return Partition(A, p, r)

Tässä Random(p, r) palauttaa satunnaisen luvun joukosta $\{\, p, p+1, \ldots, r\,\}$ ja Partition on sama kuin edellä.

Pahin tapaus on $\Omega(n^2)$ kuten ennen, mutta nyt se ei esiinny millään yksittäisellä syötteellä.

Sen sijaan tapaus $\Omega(n^2)$ voi esiintyä millä tahansa syötteellä, mutta tämä edellyttää hyvin huonoa onnea satunnaislukujen arvonnassa.

Algoritmin aikavaativuus on odotusarvoisesti $\Theta(n \log n)$, missä satunnaisuus nyt liittyy vain algoritmin sisäisiin valintoihin eikä oletukseen syötteen jakaumasta.

Alaraja vertailuihin perustuvalle järjestämiselle

Edellä esitetyt järjestämisalgoritmit ovat kaikki vertailuihin perustuvia: ne käsittelevät järjestettäviä arvoja vain

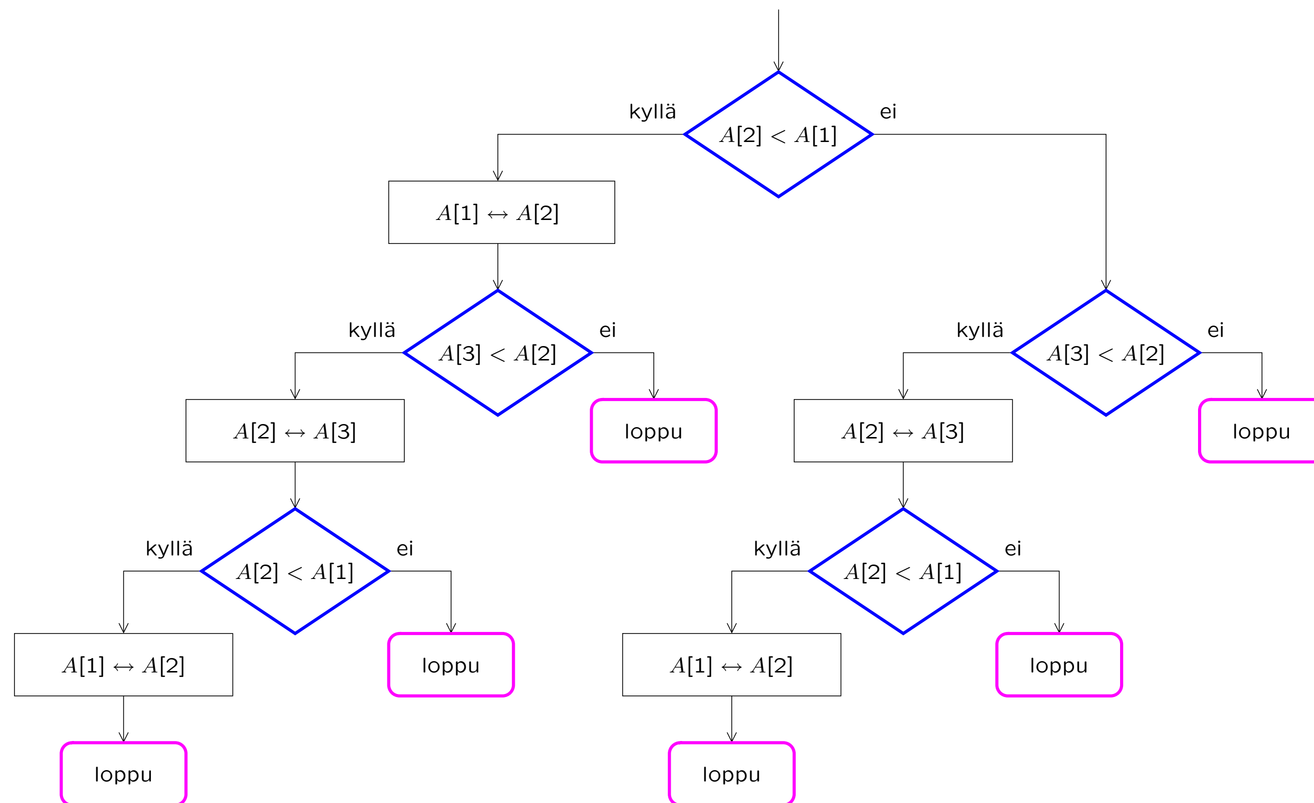
- testaamalla järjestysehtoja $A[i] < A[j]$, $A[i] = A[j]$, $A[i] > A[j]$ jne.
- vaihdoilla $A[i] \leftrightarrow A[j]$ ja yleisemmin sijoituksilla $x \leftarrow A[i]$ jne.

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

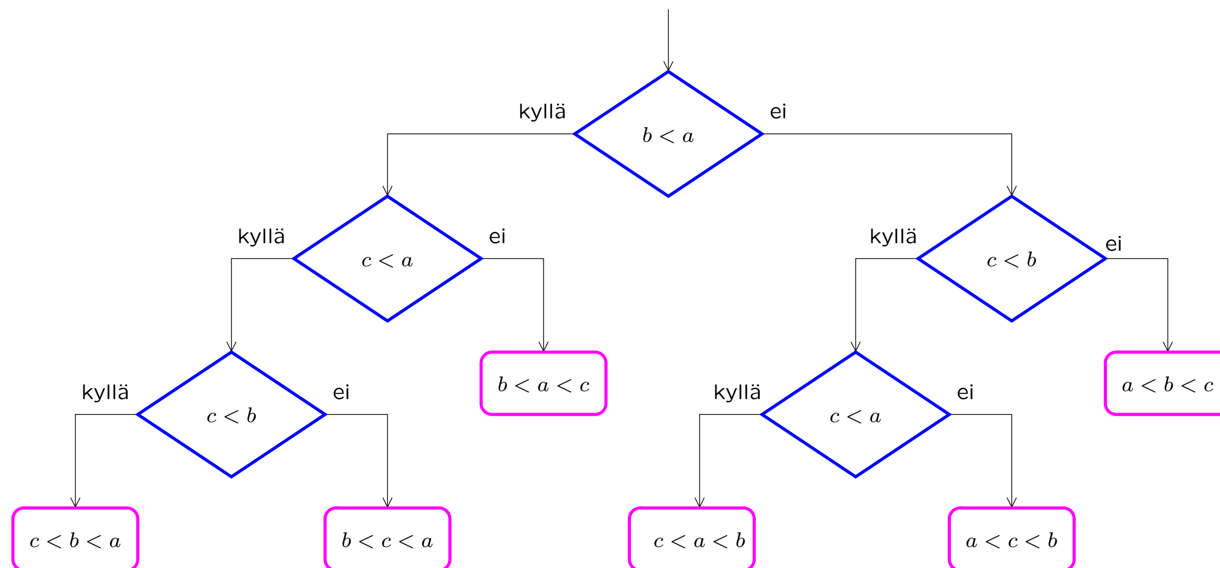
Kurssilta *Johdatus diskreettiin matematiikkaan* tiedämme, että vertailuihin perustuva järjestämisalgoritmi suorittaa pahimmassa tapauksessa $\Omega(n \log n)$ vertailua. Kertaamme lyhyesti tämän väitteen takana olevat ajatukset.

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

Alla on esimerkkinä vuokaavio lisäysjärjestämiselle, kun $n = 3$.



Saamme vuokaavioesityksestä päätöspuun, kun jätämme sijoitusoperaatiot pois ja merkitsemme muuttujien $A[1]$, $A[2]$ ja $A[3]$ alkuperäisiä arvoja a, b ja c. Päätöspuun lehdestä nähdään, mikä on alkioiden a, b ja c järjestys.



(Yksinkertaisuuden vuoksi oletetaan, että alkiot ovat erisuuria.)

Yleisesti järjestämisalgoritmia vastaavassa päätöspuussa

- sisäsolmuina on ehtotestejä
- jokaisella sisäsolmulla on kaksi lasta
- lehtinä on n alkion järjestyksiä (permutaatioita)
- puun korkeus on pahimmassa tapauksessa tehtävien vertailujen lukumäärä.

Jotta algoritmi toimisi oikein, jokaiselle syötteen järjestykselle pitää olla (ainakin) yksi lehti.

Siis lehtiä on ainakin $n!$.

Koska kyseessä on binääripuu, sen korkeus on ainakin $\log_2(n!)$. Tämä on siis samalla alaraja algoritmin pahimman tapauksen aikavaativuudelle.

Teemme nyt hyvin karkean arvion

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

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

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

Toistojen havaitseminen

Edellä esitettyjen järjestämisalgoritmien sovelluksena tarkastelemme seuraavaa ongelmaa:

> Annettu: taulukko $A[1\,..\,n]$
> Kysymys: onko taulukossa A toisto, ts. $i \neq j$ joilla $A[i] = A[j]$.

Ongelma voidaan ratkaista ajassa $O(n^2)$ vertailemalla kaikkia $n(n-1)/2$ paria.

Jos alkioille voidaan määritellä mikä tahansa järjestysrelaatio (niin kuin yleensä voidaan), ongelma voidaan ratkaista tehokkaammin ajassa $O(n \log n + n) = O(n \log n)$ järjestämällä taulukko ja vertailemalla sitten vain peräkkäisiä alkioita.

Itse asiassa mikä tahansa vertailuihin perustuva algoritmi vaatii pahimmassa tapauksessa $\Omega(n \log n)$ vertailua, mikä voidaan osoittaa päätöspuuanalyysilla kuten vastaava tulos järjestämiselle [Reingold 1972].

Siis järjestäminen on yleisessä tapauksessa vakiokerrointa vaille optimaalista, ja käytännössäkin hyvä ratkaisu. Jos alkiot ovat esim. kokonaislukuja, hajauttaminen tyypillisesti toimii ajassa $O(n)$, mutta pahin tapaus on $\Omega(n^2)$.

Käyttämällä tunnettuja järjestämisalgoritmien osasia hieman toisin päästään aikaan $O(n + m \log n)$, missä m on ensimmäisen toiston paikka järjestetyssä taulukossa, tai $m = n$ jos toistoja ei ole. Tämä on pahimmassa tapauksessa edelleen $\Omega(n \log n)$, mutta toisinaan parempi.

```
Duplicates(A)
    Build-Max-Heap(A)
    while heap-size[A] > 1
        do b ← Heap-Delete-Max(A)
           if b = Heap-Max(A)
              then return True
    return False
```

Siis luodaan keko ajassa $O(n)$ ja poimitaan alkioita järjestyksessä ajassa $O(\log n)$ per alkio, kunnes toisto löytyy.

Järjestäminen lineaarisessa ajassa

Laskemisjärjestäminen (counting sort) järjestää n alkiota ajassa $\Theta(n+k)$, kun alkiot ovat kokonaislukuja väliltä $0 \mathinner{.\,.} k-1$.

Perusidea on yksinkertainen:

- Lasketaan jokaisen luvun $0, \ldots, k$ esiintymien lukumäärä taulukossa A. Olkoon $C[i]$ luvun i esiintymien määrä.
- Nyt järjestämisen jälkeen taulukossa A pitää olla luku i kohdissa $A[a_i \mathinner{.\,.} b_i]$, missä

$$
\begin{aligned}
a_i &= \sum_{j=0}^{i-1} C[j] + 1 \\
b_i &= \sum_{j=0}^{i} C[j].
\end{aligned}
$$

Seuraava hieman hiottu versio on vakaa, mistä on hyötyä jatkossa.

Counting-Sort(A, n, k)

```
for i ← 0 to k
      do C[i] ← 0
for j ← 1 to n
      do C[A[j]] ← C[A[j]] + 1
▷ Nyt C[i] on avaimen i esiintymien määrä.
for i ← 1 to k
      do C[i] ← C[i] + C[i − 1]
▷ Nyt C[i] on avainten 0, ..., i esiintymien yhteismäärä.
▷ Siis avaimen i viimeinen esiintymä kuuluu paikkaan A[C[i]].
for j ← n downto 1          ▷ A[j] oikealla paikalle aputaulukkoon B
      do x ← A[j]
         B[C[x]] ← x
         C[x] ← C[x] − 1    ▷ "Käytettiin" yksi x.
A ← B
```

Algoritmin aikavaativuus on selvästi $\Theta(n + k)$, samoin työtilan tarve.

A	4	3	2	5	1	1	8	5	5	2	4	2

	0	1	2	3	4	5	6	7	8	9
C	0	2	3	1	2	3	0	0	1	0

	1	2	3	4	5	6	7	8	9	10	11	12
A	4	3	2	5	1	1	8	5	5	2	4	(2)
B					2							

C	0	2	(5)	6	8	11	11	11	12	12
			-1							

A	4	3	2	5	1	1	8	5	5	2	(4)	2
B					2			4				

C	0	2	4	6	(8)	11	11	11	12	12
					-1					

A	4	3	2	5	1	1	8	5	5	(2)	4	2
B				2	2			4				

C	0	2	(4)	6	7	11	11	11	12	12
			-1							

A	4	3	2	5	1	1	8	5	(5)	2	4	2
B				2	2			4			5	

C	0	2	3	6	7	(11)	11	11	12	12
						-1				

ja niin edelleen . . .

Reikäkorttijärjestämisessä (radix sort) järjestettävissä avaimissa on d kenttää, joista kullakin on k mahdollista arvoa.

Historiallisesti

- normaalissa reikäkortissa on $d = 80$ saraketta
- kussakin sarakkeessa on $k = 10$ kohtaa, joista yksi voidaan rei'ittää esittämään jotakin numeroista $0, \ldots, 9$ (ja pari lisäreiän paikkaa muun kuin numerodatan koodaamiseen)
- mekaanisesti voidaan jakaa reikäkorttipakka 10 osapakkaan sen mukaan, missä kohdassa jollain yhdellä valitulla sarakkeella on reikä.

Kysymys: miten kortit saadaan kaikki sarakkeet huomioon ottavaan järjestykseen?

Numeroidaan sarakkeet vasemmalta oikealle, eniten merkitsevä sarake ensin. Olkoon c_i kortin c sarakkeessa i oleva numero. Siis kortti a on järjestyksessä ennen korttia b, jos jollain $1 \leq i \leq k$ pätee $a_i < b_i$ ja $a_j = b_j$ kun $1 \leq j \leq i - 1$.

Ensimmäinen idea:

- ensin lajitellaan koko pakka 10 osapakkaan sarakkeen 1 mukaan
- toisessa vaiheessa säilytetään osapakkojen keskinäinen järjestys ja lajitellaan kukin sarakkeen 2 mukaan
- ...

Ongelma: Kun ehditään sarakkeeseen i, järjestettäviä osapakkoja on 10^{i-1}. Osapakkoja ei voi yhdistää, koska tällöin myöhemmät vaiheet sotkisivat aiempien vaiheiden tulokset.

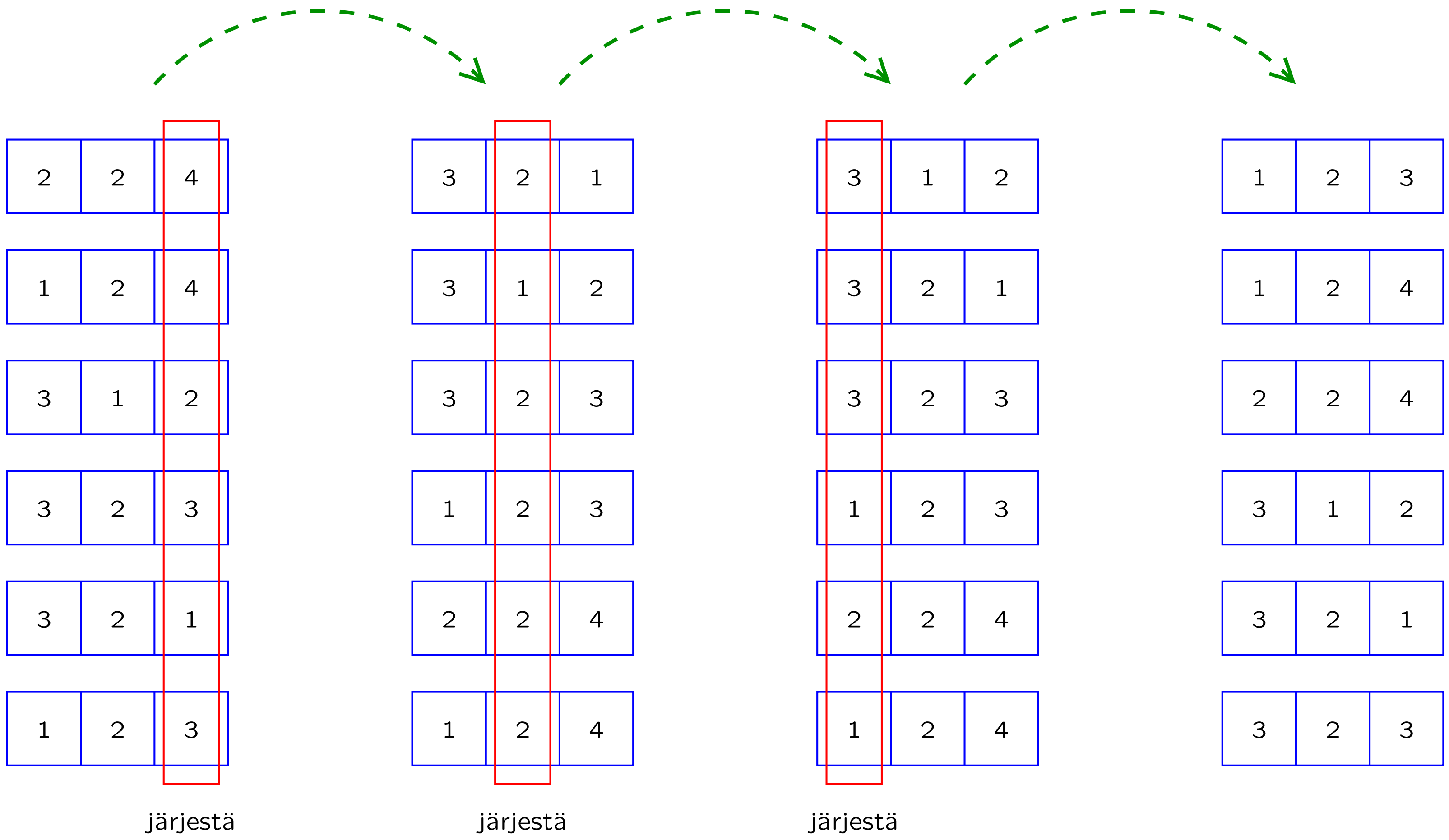
Ratkaisu: Järjestäminen aloitetaan vähiten merkitsevästä sarakkeesta.

Oletetaan siis annetuksi taulukko $A[1\,..\,n, 1\,..\,d]$, jonka alkiot ovat joukosta $\{0, \ldots, k-1\}$. Järjestetään taulukon rivit ("reikäkortit") siten, että sarake 1 on eniten merkitsevä:

```
Radix-Sort(A, d)
    for i ← d downto 1
        do järjestä A sarakkeen i mukaan
           jollain vakaalla järjestämisalgoritmilla
```

Jos sarakkeiden mukaiset järjestämiset tehdään laskemisjärjestämisellä, kokonaisaikavaatimus on $\Theta(d(n+k))$.

Algoritmin oikeellisuus perustuu invarianttiin, että k järjestämiskierroksen jälkeen A on järjestyksessä, kun huomioon otetaan vain k viimeistä saraketta. Vakautta tarvitaan, ettei sarakkeen i järjestäminen sotke sarakkeita $i+1, \ldots, d$.


2 2 4
1 2 4
3 1 2
3 2 3
3 2 1
1 2 3
järjestä
3 2 1
3 1 2
3 2 3
1 2 3
2 2 4
1 2 4
järjestä
3 1 2
3 2 1
3 2 3
1 2 3
2 2 4
1 2 4
järjestä
1 2 3
1 2 4
2 2 4
3 1 2
3 2 1
3 2 3

Yhteenveto järjestämisestä

algoritmi	aika	työtila	vakaa
lisäysjärj.	$O(n^2)$	$O(1)$	kyllä
kekojärj.	$O(n \log n)$	$O(1)$	ei
lomitusjärj.	$O(n \log n)$	$O(n)$	kyllä
pikajärj. keskim.	$O(n \log n)$	$O(\log n)$	ei
pikajärj. pahin	$O(n^2)$	$O(n)$	ei
reikäkorttijärj.	$O(d(n + k)$	$O(n + k)$	kyllä

Taulukossa on algoritmien perusversiot, joita yleensä käytetään. Niistä on erilaisia lähinnä teoreettisesti kiinnostavia variantteja, kuten edellä mainittu mediaaniin perustuva pikajärjestäminen.

Käytännössä yleisin ratkaisu on pikajärjestäminen, paitsi jos

- syöte hyvin pieni $\Rightarrow$ lisäysjärjestäminen
- pahin tapaus tärkeä $\Rightarrow$ esim. kekojärjestäminen.

7. Verkkoalgoritmit

Verkko (graph) on perustietorakenne, joka on tärkeä

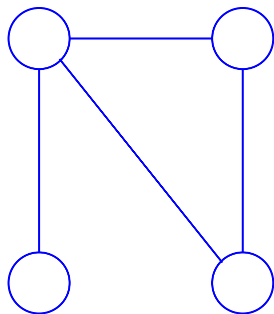
- reaalimaailman ilmiöiden vastineena: tieverkko, viemäriverkosto, Internet jne.
- tietojenkäsittelyjärjestelmien sisäisenä ilmiönä: esim. erilaiset kaavioesitykset ohjelmistojen ja tietokantojen suunnittelussa
- matemaattisena mallina ilmiöille, jotka päältä päin eivät välttämättä näytä verkoilta: esim. etsintäongelmat tai laskennan eteneminen

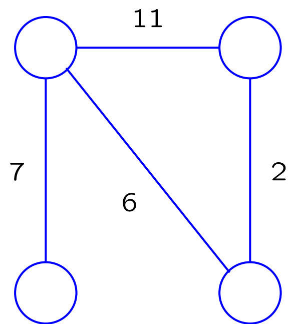
Verkkoihin liittyy runsaasti vaikeudeltaan vaihtelevia algoritmisia ongelmia ja niiden ratkaisutekniikoita, joista tässä esitellään vain hyvin pieni osa.

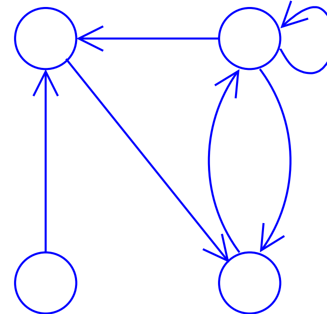
Verkko koostuu solmuista (vertex, node) ja kahta solmua yhdistävistä kaarista eli särmistä (edge). Kuvassa verkon solmut tyypillisesti esitetään ympyröinä tms. ja kaaret niitä yhdistävinä viivoina.

Suuntaamattomassa verkossa kaaren kumpikin pää on samanarvoinen. Suunnatussa verkossa kaarella on alku- ja loppupää, ja solmusta a voi olla kaari solmuun b riippumatta siitä, onko solmusta b kaari solmuun a. Suuntaaminen esitetään kuvassa nuolenkärjillä.

Painotetussa verkossa jokaiseen kaareen liittyy paino, joka on esim. luonnollinen luku mutta voi olla muutakin (toisinaan jotain hyvinkin abstraktia).







Suuntaamaton verkko, painotettu suuntaamaton verkko ja suunnattu verkko

Esimerkkejä verkoista ja verkko-ongelmista

Tietokoneverkon yhtenäisyys ("2-yhtenäiset komponentit"):

solmut: tietokoneita

kaaret: tietoliikenneyhteyksiä; ei suuntausta, ei painoja(?)

ongelma: mitkä yhteydet ovat sellaisia, että niiden katkeaminen jakaisi verkon kahteen toisistaan eristettyyn osaan

Robotin navigointi:

solmut: sopivalla tarkkuustasolla esitettyjä maantieteellisiä sijainteja

kaaret: tunnettuja väyliä; ei suuntausta, ei painoja

ongelma: ohjaa robotti paikasta A paikkaan B

Maantieverkosto:

solmut: kaupunkeja
kaaret: maanteitä; ei suuntausta
painot: kaupunkien välisiä etäisyyksiä
ongelma: mikä on lyhin reitti kaupungista A kaupunkiin B

Logistiikkaverkosto ("monihyödykevuo"):

solmut: varastoja
kaaret: olemassaolevia kuljetusreittejä; ei suuntausta
painot: useasta tavaralajista tieto, kuinka paljon sitä voidaan tietyssä ajassa kuljettaa mitäkin reittiä pitkin
ongelma: miten saadaan halutut määrät tavaroita kulkemaan eri varastojen välillä

Tilasiirtymäverkko:

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

Äärellinen automaatti (kurssilla *Laskennan mallit*):

- solmut: abstraktin automaatin laskennan tilanteita
- kaaret: abstraktin automaatin laskenta-askelia
- painot: kirjaimia
- ongelma: Voiko tilasta A kulkea tilaan B siten, että kuljettujen kaarten painoista muodostuu haluttu merkkijono

Verkkojen perusmääritelmät

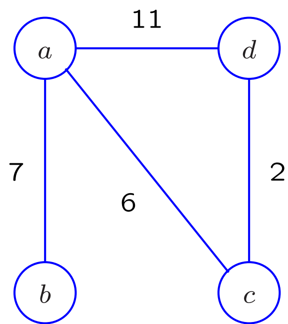
Matemaattisesti verkko G on pari $G = (V, E)$, missä

- solmujen joukko V on mikä tahansa äärellinen joukko
- kaarten joukko E on karteesisen tulon $V \times V$ osajoukko; siis yksittäiset kaaret ovat pareja (u, v), missä $u \in V$ ja $v \in V$.

Suuntaamattomassa verkossa vaaditaan, että E on symmetrinen eli $(u, v) \in E$ jos ja vain jos $(v, u) \in E$.

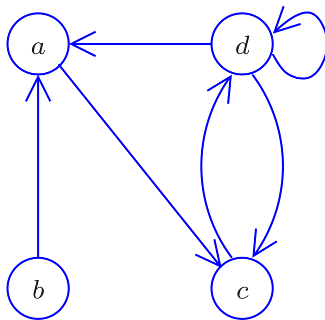
Yleensä suuntaamattomassa verkossa ei sallita kaaria solmusta itseensä: pitää päteä $(u, u) \notin E$ kaikilla $u \in V$. Suunnatussa verkossa myös $(u, u) \in E$ on mahdollista.

Painot esitetään funktiona $w\colon E \to \mathbb{N}$ (tai mikä painojen arvoalue sitten onkin).



Suuntaamaton painotettu verkko:

- $V = \{ a, b, c, d \}$
- $E = \{ (a, b), (b, a), (a, c), (c, a), (a, d), (d, a), (c, d), (d, c) \}$
- $w(a, b) = w(b, a) = 7$, $w(c, d) = w(d, c) = 2$ jne.



Suunnattu (painottamaton) verkko:

- $V = \{ a, b, c, d \}$
- $E = \{ (a, c), (b, a), (c, d), (d, a), (d, c), (d, d) \}$

Jos solmusta u solmuun v on kaari eli $(u, v) \in E$, sanomme, että v on solmun u vieressä ja merkitsemme

$$u \rightarrow v.$$

Solmut u ja v ovat naapureita, jos u on v:n vieressä tai toisin päin; suuntaamattomalle verkolle nämä ovat sama asia. Jos lisäksi painotetussa verkossa kaaren paino on $w(u, v) = x$, voidaan merkitä

$$u \xrightarrow{x} v.$$

Solmujono $(v_0, v_1, \ldots, v_k)$ on polku solmusta v_0 solmuun v_k, jos $(v_{i-1}, v_i) \in E$ kaikilla $1 \le i \le k$ eli

$$v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k.$$

Jos solmusta u on polku solmuun v, sanomme, että solmu v on saavutettavissa solmusta u ja merkitsemme

$$u \rightsquigarrow v.$$

Polun pituus on sillä olevien kaarten lukumäärä. Mistä tahansa solmusta u itseensä on pituutta 0 oleva polku (u) (siis $k = 0$ ja $v_0 = v_k = u$).

Painotetussa verkossa polun paino on sillä olevien kaarten painojen summa.

Suunnatun tai suuntaamattoman verkon polku $(v_0, \ldots, v_k)$ on yksinkertainen, jos se ei leikkaa itseään eli $v_i \neq v_j$ kun $i \neq j$.

Suunnatussa verkossa polkua $(v_0, \ldots, v_k)$ sanotaan sykliksi eli kehäksi, jos $v_0 = v_k$ ja $k \geq 1$. Siis jos $(v_0, v_0) \in E$, niin (v_0, v_0) muodostaa pituutta 1 olevan syklin. Sen sijaan nollan mittaista polkua (v_0), jollainen on olemassa kaikilla v_0, ei lasketa sykliksi.

Jos syklissä $(v_0, \ldots, v_k)$ lisäksi $v_i \neq v_j$ kaikilla $0 < i < j < k$ eli se ei päätepisteitä lukuunottamatta leikkaa itseää, sitä sanotaan yksinkertaiseksi sykliksi.

Suuntaamattomassa verkossa ei ole järkevää laskea sykliksi saman kaaren kulkemista edes takaisin. Koska solmusta ei saa olla kaarta itseensä, tästä seuraa erityisesti, että syklin pituus on vähintään 3. Täsmällisemmin suuntaamattoman verkon polku $(v_0, \ldots, v_k)$ on sykli, jos $v_0 = v_k$ ja millään $0 \leq i, j \leq k-1$ ei päde $v_i = v_{j+1}$ ja $v_j = v_{i+1}$. Yksinkertainen sykli määritellään kuten suunnatussa tapauksessa.

Suunnattu tai suuntaamaton verkko on syklitön (acyclic), jos siinä ei ole yhtään sykliä. Selvästi jos verkko ei ole syklitön, niin siinä on myös ainakin yksi yksinkertainen sykli.

Suuntaamaton verkko on yhtenäinen (connected), jos $u \rightsquigarrow v$ kaikilla $u, v \in E$.

Suunnattu verkko on vahvasti yhtenäinen (strongly connected), jos $u \rightsquigarrow v$ kaikilla $u, v \in E$. "Vahva" korostaa, että kaarten suuntauksia on kunnioitettava.

Tärkeitä erikoistapauksia

Suunnattu syklitön verkko (directed acyclic graph eli DAG): esim. kurssien suoritusjärjestystä koskevat rajoitukset; määrittää solmujen välille osittaisen järjestyksen.

Vapaa puu on suuntaamaton verkko, joka on sekä syklitön että yhtenäinen. Puussa minkä tahansa kahden solmun välillä on tasan yksi yksinkertainen polku.

Juurellinen puu on usein luontevaa esittää muodostamalla vastaava vapaa puu ja suuntaamalla sitten kaaret juurta kohti.

Verkkojen tallettaminen

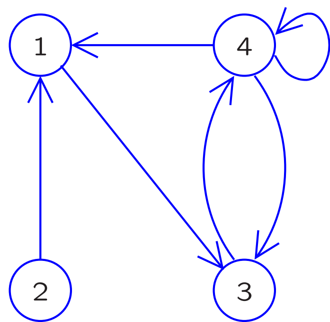
Merkintöjen yksinkertaistamiseksi oletetaan $V = \{1, \ldots, n\}$, missä $n = |V|$ on solmujen lukumäärä.

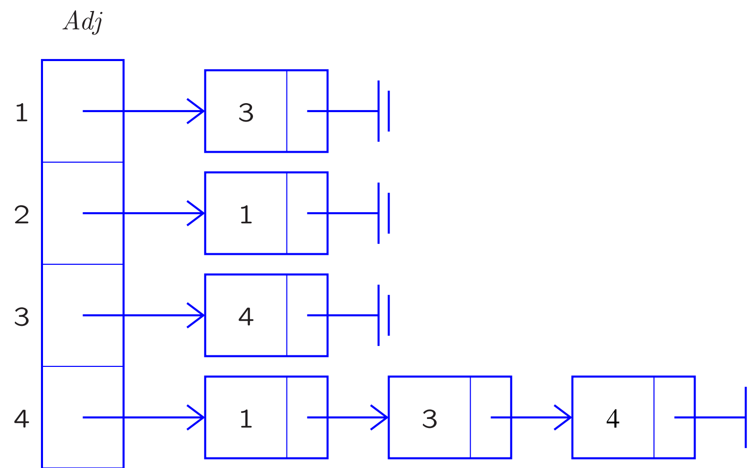
Vieruslistaesitys sisältää kullekin solmulle i, $1 \leq i \leq n$, vieruslistan $Adj[i]$, jossa on lueteltu sen viereiset solmut eli ne j, joilla $(i, j) \in E$. Vieruslistat toteutetaan tyypillisesti yhteen suuntaan linkitettyinä.

Tilaa tarvitaan

- kullekin solmulle i osoitin listan $Adj[i]$ alkuun ja
- kullekin kaarelle (i, j) vakiokokoinen tietue solmun i vieruslistassa

eli yhteensä $O(|V| + |E|)$.

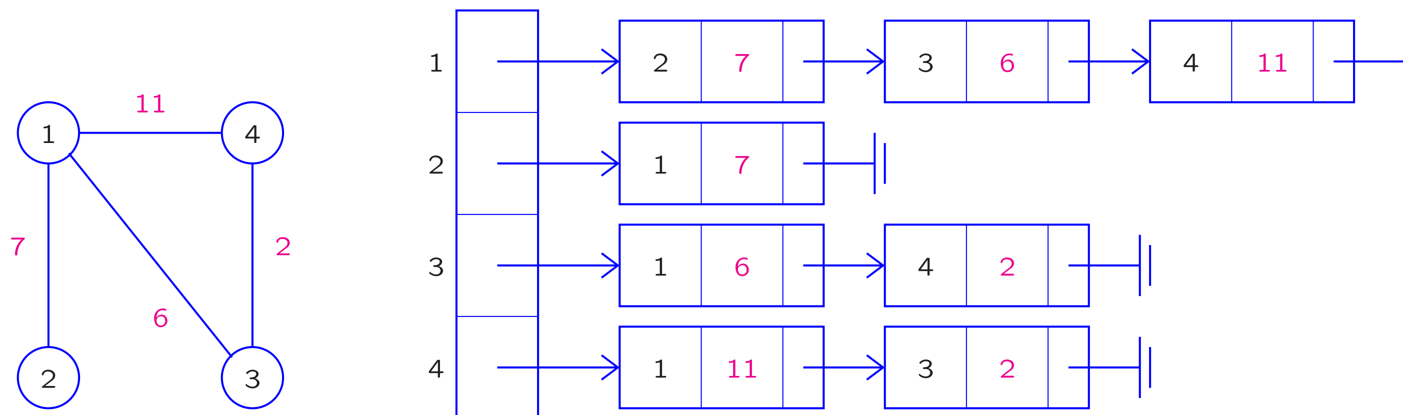




Suunnattu verkko ja sen vieruslistaesitys

Suuntaamattomassa verkossa jokaisesta kaaresta (u, v) tulee tietue sekä vieruslistaan $Adj[u]$ että vieruslistaan $Adj[v]$.

Vieruslistaan voidaan helposti tallettaa myös kaarten painot.



Joissain algoritmeissa (ei tällä kurssilla) kaariin liittyviä tietoja (esim. painoja) joudutaan päivittämään. Tällöin on kätevää linkittää aina toisiinsa saman kaaren (u, v) kaksi listaesiintymää (u listassa $Adj[v]$ ja v listassa $Adj[u]$).

Vierusmatriisiesityksessä verkko esitetään $n \times n$-matriisina A, missä

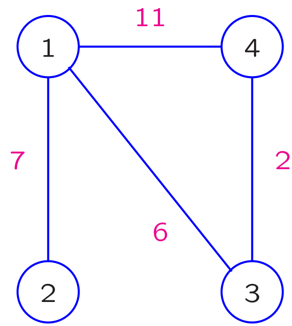
$$A[i,j] = \begin{cases} 1 & \text{jos } (i,j) \in E \\ 0 & \text{muuten.} \end{cases}$$

Painotetun verkon tapauksessa matriisiin laitetaan suoraan painot:

$$A[i,j] = \begin{cases} w(i,j) & \text{jos } (i,j) \in E \\ 0 \text{ tai } \infty & \text{muuten,} \end{cases}$$

missä tilanteen mukaan joko 0 tai ∞ on yleensä sopiva arvo koodaamaan kaaren puuttumisen.

Jos verkko on suuntaamaton, vierusmatriisi on symmetrinen eli $A[i,j] = A[j,i]$. Tätä hyödyntämällä voidaan tarvittaessa säästää puolet matriisin muistintarpeesta.



	1	2	3	4
1	∞	7	6	11
2	7	∞	∞	∞
3	6	∞	∞	2
4	11	∞	2	∞

Suuntaamaton painotettu verkko vierusmatriisina

Vieruslistaesitys:

+ säästää tilaa, jos verkko on harva eli $|E| \ll |V|^2$

+ harvalla verkolla säästää myös aikaa tyypillisillä algoritmeilla, joissa perusrakenne on

  toista seuraava kaikilla solmun u naapureille: . . . .

– ehdon "$(u, v) \in E$?" testaaminen vie pahimmassa tapauksessa ajan $\Omega(|V|)$.

Vierusmatriisiesitys:

+ säästää tilaa, jos verkko on tiheä eli $|E| = \Theta(|V|^2)$

+ mikä tahansa kaari tai paino löytyy vakioajassa

– tehoton, jos verkko on harva.

Jatkossa esitettävistä algoritmeista jotkin vaativat listaesityksen, jotkin matriisiesityksen, toimiakseen tehokkaasti.

Verkon läpikäynti

Verkon leveyssuuntainen ja syvyyssuuntainen läpikäynti ovat kaksi perustavinta verkkoalgoritmia. Kumpikin

- lähtee liikkeelle parametrina saadusta lähtösolmusta s
- vierailee kaikissa solmusta s saavutettavissa olevissa solmuissa
- toimii ajassa $O(|V| + |E|)$ ja työtilassa $O(|V|)$, kun verkko on esitetty vieruslistoina
- toimii sekä suunnatuille että suuntaamattomille verkoille (mutta tulosten tulkinta voi poiketa).

Ilmeinen sovellus kummallekin on löytää polku lähtösolmusta s haluttuun kohdesolmuun t.

Algoritmit tuottavat myös erilaista lisäinformaatiota verkon rakenteesta, joten niitä käytetään muissakin verkkoalgoritmeissa esiprosessorina tai rakennusalustana.

Leveyssuuntainen läpikäynti (breadth-first search, BFS)

Lähtösolmusta s saavutettavissa olevat solmut käydään läpi taso kerrallaan.

Tasolle i tulee solmut, joihin lyhimmän polun pituus solmusta s on tasan i:

- tasolle 0 tulee solmu s.
- tasolle 1 tulee kaikki solmun s viereiset solmut
- tasolle 2 tulee kaikki tason 1 solmujen viereiset solmut, paitsi ne jotka ovat jo tasolla 0 tai 1.
- ...

Olemme nähneet (kalvo 213), miten juurellisen puun tasottainen läpikäynti onnistuu jonon avulla.

Jono toimii myös yleisellä verkolla. Lisäksi täytyy pitää kirjaa siitä, missä solmuissa on jo käyty. Muuten verkon syklit voisivat viedä algoritmin silmukkaan.

Kirjanpitoa varten liitämme jokaiseen solmun värin, joka on aluksi valkoinen ja läpikäynnin edetessä voi muuttua ensin harmaaksi ja sitten mustaksi.

Solmu u on

- valkoinen jos läpikäynti ei vielä ole edennyt solmuun u
- harmaa jos läpikäynti on ehtinyt solmuun u, mutta solmusta u edelleen vieviä kaaria ei ole vielä käsitelty
- musta jos solmusta u lähtevät kaaret on käsitelty (jolloin koko solmu u on loppuunkäsitelty).

Aluksi lähtösolmu s on harmaa, muut valkoisia.

Algoritmin pysähtyessä kaikki solmusta s saavutettavissa olevat solmut ovat mustia, muut valkoisia.

Läpikäynnin ohella algoritmi muodostaa leveyssuuntaisen puun. Leveyssuuntaisen puun juurena on s ja sisäsolmuina kaikki solmusta s saavutettavat solmut. Puun rakenne kirjataan muutujiin

- $p[u]$: solmun u vanhempi puussa
- $d[u]$: solmun u taso (eli syvyys) puussa.

Puusta tekee leveyssuuntaisen se, että

- jos $u = p[v]$, niin $(u, v) \in E$
- lyhimmän polun pituus solmusta s solmuun u verkossa G on $d[u]$.

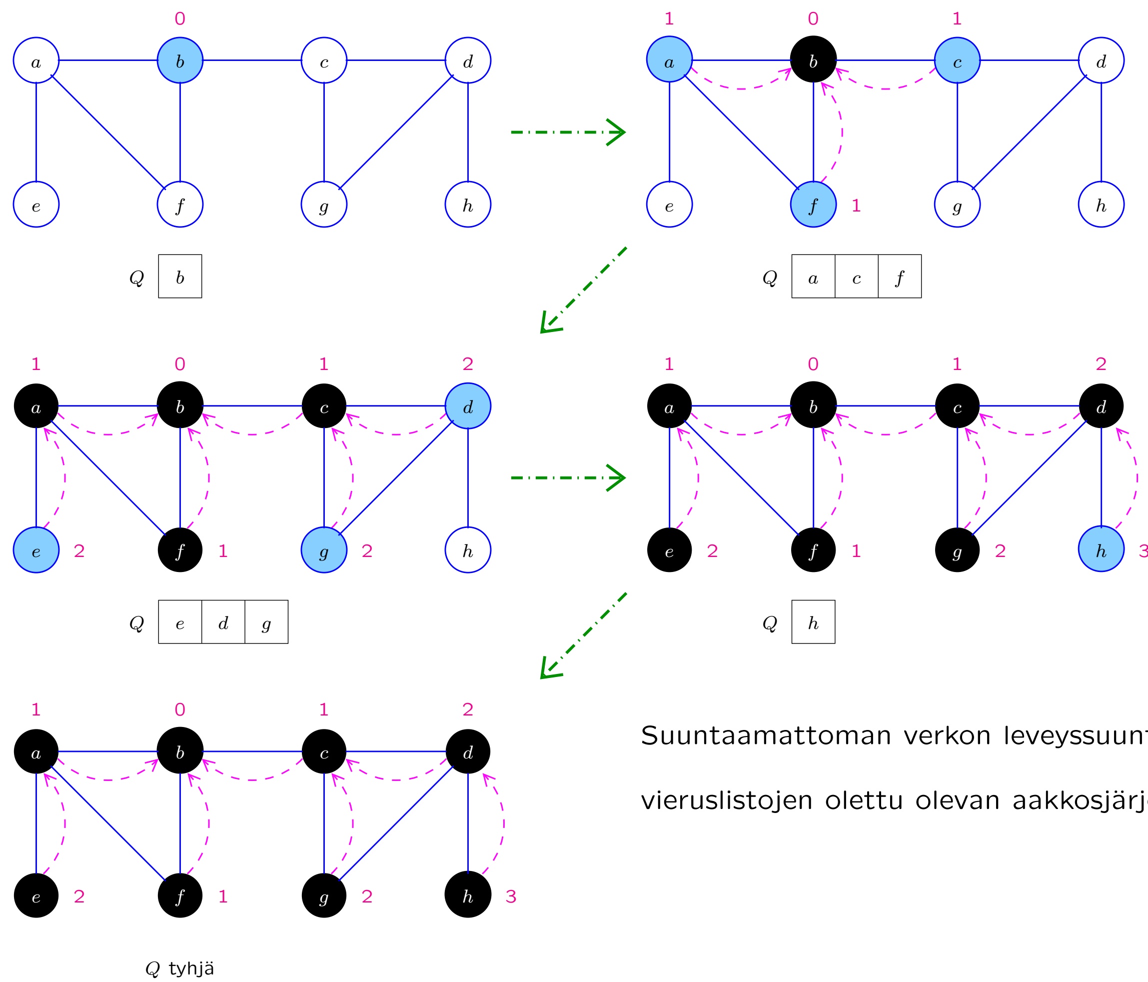
Tästä seuraa, että jos $(s, v_1, \ldots, v_{k-1}, u)$ on leveyssuuntaisen puun polku juuresta s solmuun u, niin

- $(s, v_1, \ldots, v_{k-1}, u)$ on polku myös verkossa G
- verkossa G ei ole lyhyempää polkua $s \rightsquigarrow u$

eli $(s, v_1, \ldots, v_{k-1}, u)$ on eräs lyhin polku solmusta s solmuun u.

Esitämme nyt algoritmin ja toimintaesimerkin, minkä jälkeen perustelemme oikeellisuutta tarkemmin.

```
BFS(G, s)
    for jokaiselle solmulle u ∈ V
        do colour[u] ← white
           d[u] ← ∞
           p[u] ← Nil
    colour[s] ← gray
    d[s] ← 0
    Q ← tyhjä jono
    Enqueue(Q, s)
    while not Empty(Q)
        do u ← Dequeue(Q)
           for jokaiselle solmulle v ∈ Adj[u]
               do if colour[v] = white
                     then colour[v] ← gray
                          d[v] ← d[u] + 1
                          p[v] ← u
                          Enqueue(Q, v)
           colour[u] ← black
```



Suuntaamattoman verkon leveyssuuntainen läpikäynti

vieruslistojen oletettu olevan aakkosjärjestyksessä

Jonoon vietävä solmu on aina valkea ja muuttuu samalla harmaaksi. Harmaa ei koskaan muutu takaisin valkeaksi.

Siis sama solmu viedään jonoon korkeintaan kerran.

Tilavaativuus:

- edellisen perusteella jonossa on korkeintaan $|V|$ solmua.
- jonoon todella tulee $|V|-1$ solmua, jos kaikki muut solmut ovat lähtösolmun naapureita

$\Rightarrow$ tilavaativuus on pahimmassa tapauksessa $\Theta(|V|)$.

Aikavaativuus:

- jokainen vieruslista käydään lävitse korkeintaan kerran

$\Rightarrow$ **while**-silmukan aikavaativuus on $O(|E|)$

- **for**-silmukka vie ajan $\Theta(|V|)$

$\Rightarrow$ kokonaisaikavaativuus on $O(|V|+|E|)$

- pahimmassa tapauksessa kaikki vieruslistat käydään läpi ja aikavaativuus on $\Theta(|V|+|E|)$.

Väitämme, että algoritmin suorituksen jälkeen kaikilla solmuilla u pätee, että

- jos u on saavutettavissa solmusta s, niin u on musta ja $d[u]$ on lyhimmän polun pituus $s \rightsquigarrow u$
- muuten u on valkoinen ja $d[u] = \infty$.

Tämän pätevyydestä on melko helppo vakuuttua algoritmia tarkastelemalla ja simuloimalla. Seuraavassa esitettävä hieman matemaattisempi tarkastelu on kuitenkin hyödyllinen valmistava esimerkki ajatustavoista, joita tarvitaan hankalampien algoritmien ymmärtämiseen.

Värityksiä koskeva osa algoritmin toiminnasta on helppo todeta oikeaksi.

Algoritmista nähdään suoraan, että while-silmukassa pätee invariantti

- jos solmu on valkoinen, se ei ole käynytkään jonossa
- jos solmu on harmaa, se on parhaillaan jonossa
- jos solmu on musta, se on poistettu jonosta
- jos solmu on musta, sen viereiset solmut ovat harmaita tai mustia.

Erityisesti algoritmin päättyessä

- harmaita solmuja ei ole, koska jono on tyhjä
- lähtösolmu on musta
- jos $s \rightsquigarrow u$ ja u olisi valkoinen, niin polku hyppäisi jossain kohdassa mustasta valkoiseksi

$\Rightarrow$ kaikki saavutettavissa olevat solmut ovat mustia.

Tehdään toisaalta vastaoletus, että algoritmi värittää harmaaksi ainakin yhden solmun, joka ei ole saavutettavissa. solmusta s. Olkoon v näistä algoritmin suoritusjärjestyksessä ensimmäinen harmaaksi väritettävä.

Ennen kuin v voidaan värittää harmaaksi, algoritmin on pitänyt värittää harmaaksi jokin solmu u, jolla $v \in Adj[u]$. Koska oletuksen mukaan v oli ensimmäinen "väärin" väritetty, solmu u on saavutettavissa. Mutta oletuksen $v \in Adj[u]$ mukaan myös v on nyt saavutettavissa; ristiriita.

Siis solmu väritetään algoritmin kuluessa harmaaksi, jos ja vain jos se on saavutettavissa solmusta s. Koska kaikki harmaat solmut tulevat mustiksi ennen suorituksen loppua, niin värien osalta algoritmi toimii väitetyllä tavalla.

Polkujen pituuksien tarkastelemiseksi olkoon olkoon $D(u)$ lyhimmän polun pituus lähtösolmusta solmuun u. Lisäksi merkitään $D(u) = \infty$, jos polkua ei ole. Väitämme siis, että lopuksi $d[u] = D(u)$ kaikilla u.

On helppo nähdä, että algoritmi säilyttää invariantin $d[u] \geq D(u)$ kaikilla u.

Aluksi $d[u] = \infty$ ja invariantti selvästi pätee.

Kun algoritmi myöhemmin päivittää

$$d[v] \leftarrow d[u] + 1,$$

niin $(u, v) \in E$. Tällöin $D(v) \leq D(u) + 1$, ja yhtäsuuruus pätee, jos jokin lyhin polku $s \rightsquigarrow v$ kulkee solmun u kautta. Jos siis $d(u) \geq D(u)$, niin päivitys ei riko ehtoa $d(v) \geq D[v]$.

Tämä argumentti perustuu siihen, että algoritmin laskema arvo $d[u]$ on jonkin polun pituus $s \rightsquigarrow u$. Ongelmaksi jää osoittaa, että algoritmi todella löytää lyhimmän polun. Algoritmin BFS tapauksessa tämä on melko suoraviivaista, koska (kuten kohta perustellaan) algoritmi löytää solmut arvon $D(u)$ mukaan kasvavassa järjestyksessä. Myöhemmin kohtaamme vastaavan tilanteen painotetuissa verkoissa (Dijkstran algoritmi), jolloin dynamiikka on monimutkaisempi.

Analysoimme arvojen $d[u]$ laskemista jakamalla suorituksen vaiheisiin. Vaihe k päättyy, kun viimeisen kerran muutetaan mustaksi solmu u, jolla $d[u] = k - 1$. Lisäksi sovimme, että vaihe 0 koostuu alustuksista ennen while-silmukan alkua.

Todistamme induktiolla arvon k suhteen, että vaiheen k päättyessä

- jos $D(u) < k$, niin u on musta (eli poistunut jonosta)
- jos $D(u) = k$, niin u on harmaa (eli parhaillaan jonossa)
- jos $D(u) > k$, niin u on valkoinen (eli ei ole vielä ollut jonossa)
- jos u on harmaa tai musta, niin $d[u] = D(u)$.

Alustusten jälkeen väite selvästi pätee.

Oletetaan nyt, että väite pätee vaiheen k päättyessä.

Siis jonossa on tasan ne solmut u, joilla $D(u) = k$. Määritelmän mukaan $D(v) = k + 1$, jos ja vain jos

- $v \in Adj[u]$ jollain u, jolla $D(u) = k$, ja
- $v \notin Adj[w]$, jos $D(w) < k$.

Täsmälleen nämä solmut viedään jonoon vaiheen $k + 1$ aikana:

- kaikki ehdon $D(u) = k$ täyttävät vieruslistat $Adj[u]$ käydään läpi
- jos $v \in Adj[w]$ missä $D(w) < k$, niin induktio-oletuksen mukaan w on käynyt jonossa aiemmilla kierroksilla, jolloin v on muutettu harmaaksi.

Siis ehto pätee vaiheen $k + 1$ jälkeen.

Algoritmin päättyessä edellä todetun mukaan

- kaikki lähtösolmusta saavutettavat solmut ovat mustia ja
- kaikille mustille solmuille u pätee $d[u] = D(u)$.

Toisaalta muut kuin saavutettavat solmut u ovat valkoisia, ja niillä on voimassa alkuasetus $d[u] = \infty$.

Siis lopuksi $d[u] = D(u)$ kaikilla u, kuten haluttiin.

Tarkastellaan vielä p-osoittimia. Jos $u = p[v]$, niin $d[v] = d[u] + 1$ eli $D(v) = D(u) + 1$. Siis eräs lyhin polku lähtösolmusta solmuun v saadaan seuraavasti:

- kulje ensin jotain lyhintä polkua $s \rightsquigarrow u$
- kulje sitten kaarta (u, v).

Siis eräs lyhin polku solmusta s solmuun u on $(s, v_1, \ldots, v_k, u)$, missä $k = d[u] - 1$ ja

- $s = p[v_1]$
- $v_i = p[v_{i+1}]$ kun $1 \leq i < k$
- $v_k = p[u]$

eli taas haluttu tulos.

Syvyyssuuntainen läpikäynti (depth-first search, DFS)

Syvyyssuuntaisessä läpikäynnissä edetään aina yhtä polkua pitkin eteenpäin, kunnes tullaan umpikujaan. Sitten peruutetaan viimeisimpään solmuun, jossa on kokeilemattomia vaihtoehtoja.

Löydetyille poluille ei ole yhtä luontevaa tulkintaa kuin leveyssuuntaisen läpikäynnin lyhimmille poluille. Ne tarjoavat kuitenkin hyvän pohjan jatkokehitykselle:

- syklittömyyden testaaminen
- topologinen järjestäminen
- vahvasti yhtenäisten komponenttien löytäminen.

Palaamme jatkossa näihin sovelluksiin.

Pidämme taas yllä värejä. Solmu on

valkoinen jos sitä ei vielä ole saavutettu
harmaa jos se on saavutettu, mutta sitä ei ole käsitelty loppuun
musta jos se on käsitelty loppuun.

Jokaiselle solmulle u merkitään myös

- löytymishetki $d[u]$ (discovery): milloin solmu muuttui harmaaksi
- päättymishetki $f[u]$ (finish): milloin solmu muuttui mustaksi.

Aikaa mitataan kaikkiaan tehtyjen värinmuutosten määrällä.

Voisimme myös kirjata osoittimia p kuten leveyssuuntaisessa läpikäynnissä, jolloin saisimme syvyyssuuntaisen metsän (eli joukon syvyyssuuntaisia puita.) Syvyyssuuntainen puu kertoo paljon verkon rakenteesta, mutta emme jatkossa tarvitse ekplisiittisiä p-osoittimia, joten jätämme ne pois.

Syvyyssuuntainen läpikäyntialgoritmi toimii sellaisenaan sekä suunnatulle että suuntaamattomalle verkolle, mutta syvyyssuuntaisen metsän tulkinta on erilainen.

Seuraava algoritmi DFS-all käy syvyyssuuntaisesti läpi koko verkon, ei vain annetusta lähtösolmusta saavutettavia solmuja. Tämä sopii paremmin jatkossa esitettäviin sovelluksiin.

Varsinainen läpikäynti tehdään rekursiivisella proseduurilla DFS-Visit.

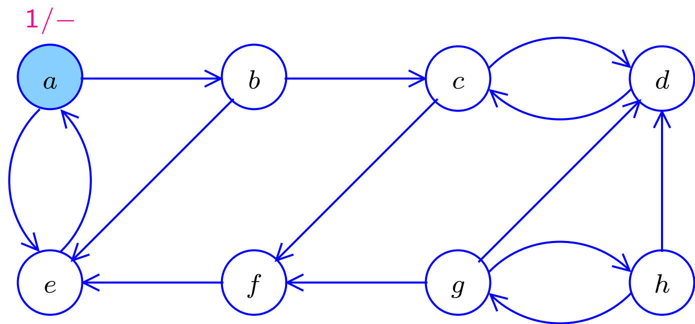
DFS-all(G)

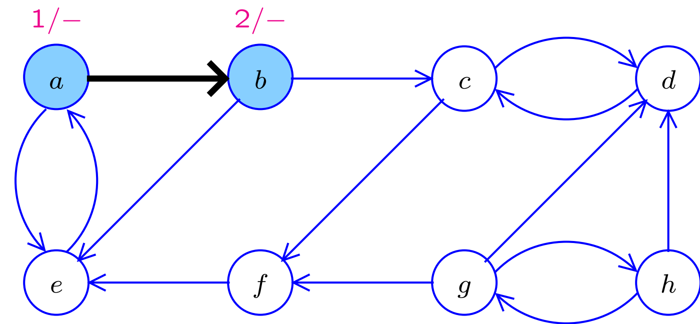
for jokaiselle $u \in V$
    do $colour[u] \leftarrow$ white
$time \leftarrow 0$
for jokaiselle $u \in V$
    do if $colour[u] =$ white
        then DFS-Visit(u)

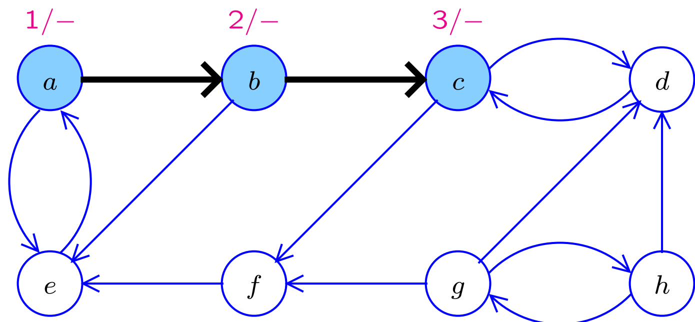
DFS-Visit(u)

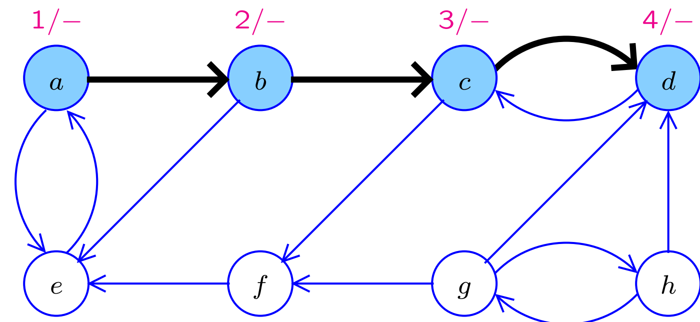
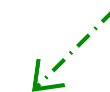
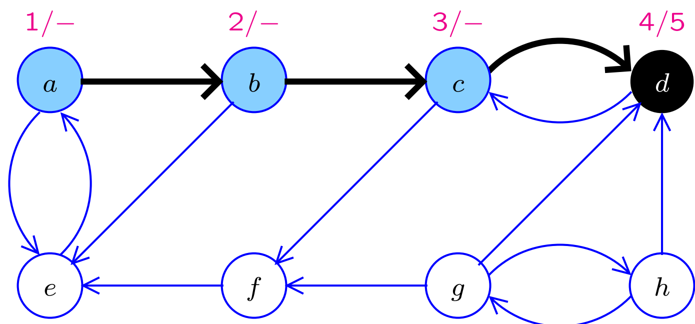
$colour[u] \leftarrow$ gray
$time \leftarrow time + 1$
$d[u] \leftarrow time$
for jokaiselle $v \in Adj[u]$
    do if $colour[v] =$ white
        then DFS-Visit(v)
$colour[u] \leftarrow$ black
$time \leftarrow time + 1$
$f[u] \leftarrow time$

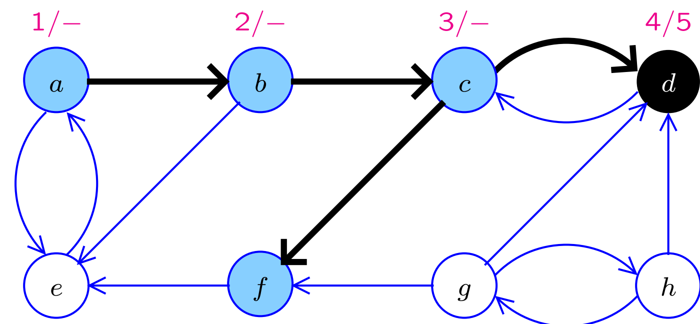
Esimerkki suunnatusta tapauksesta seuraavilla kolmella kalvolla: solmujen d- ja f-arvot merkitty muodossa $d[u]/f[u]$; edetyt kaaret vahvennettu.

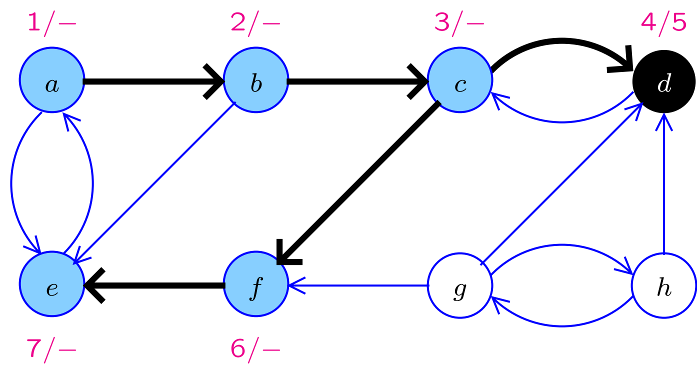

1/–
a
b
c
d
e
f
g
h

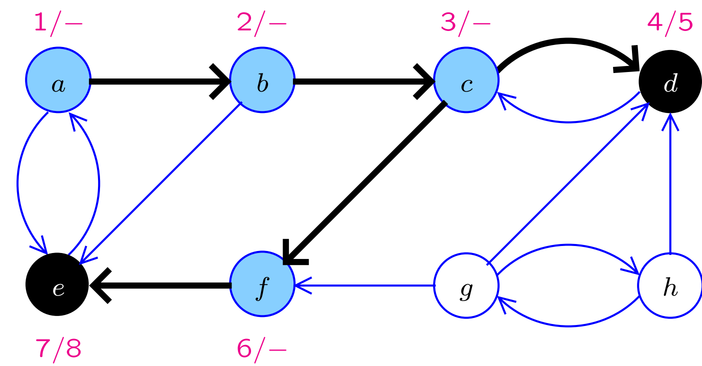

1/–
2/–
a
b
c
d
e
f
g
h

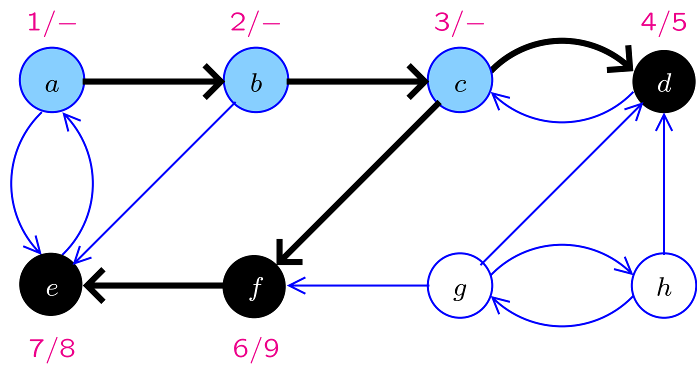

1/–
2/–
3/–
a
b
c
d
e
f
g
h

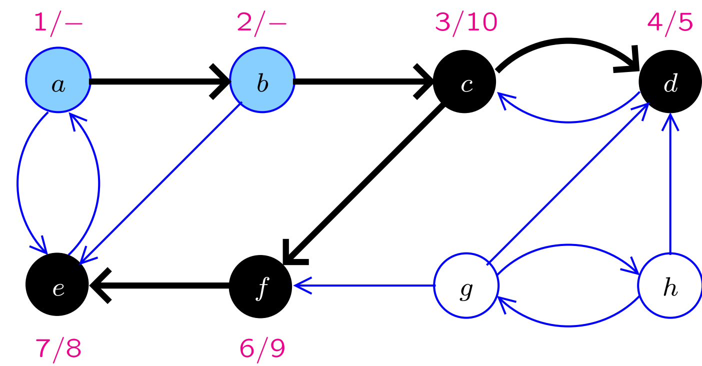

1/–
2/–
3/–
4/–
a
b
c
d
e
f
g
h


1/–
2/–
3/–
4/5
a
b
c
d
e
f
g
h

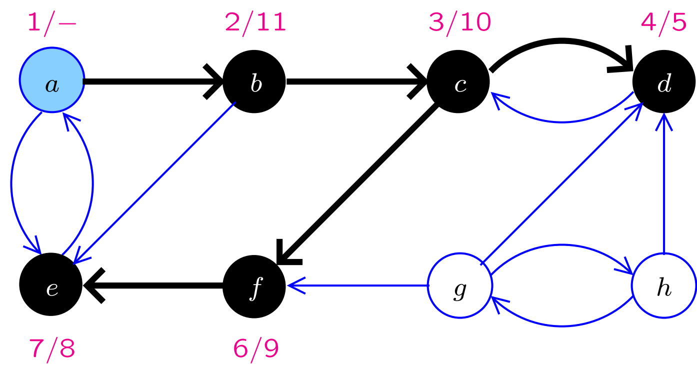

1/–
2/–
3/–
4/5
a
b
c
d
e
f
g
h
6/–

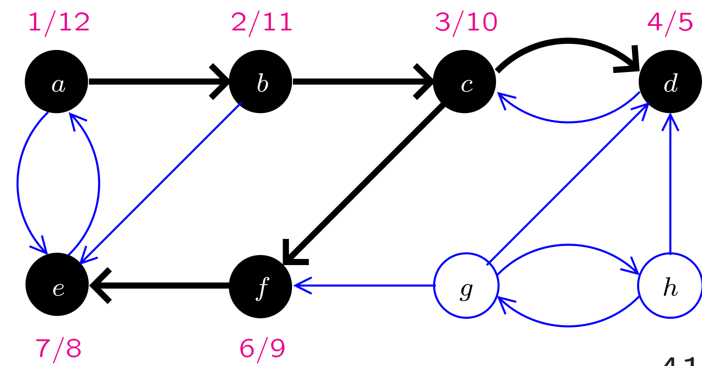

1/–
2/–
3/–
4/5
a
b
c
d
e
f
g
h
7/–
6/–

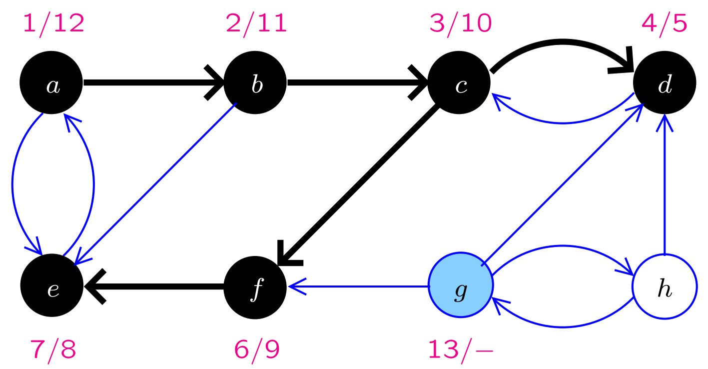

1/–
2/–
3/–
4/5
a
b
c
d
e
f
g
h
7/8
6/–

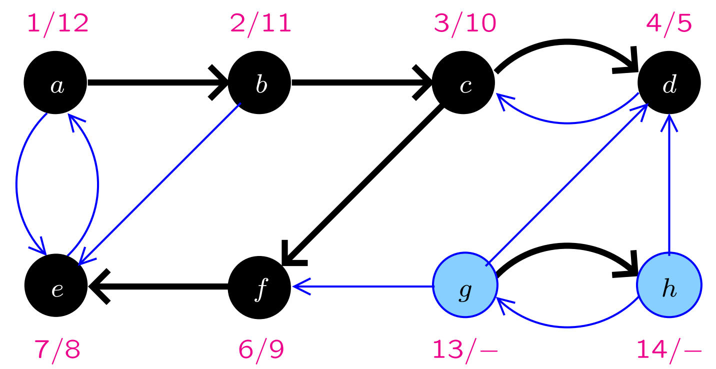

1/–
2/–
3/–
4/5
a
b
c
d
e
f
g
h
7/8
6/9

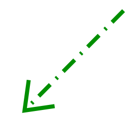

1/–
2/–
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9

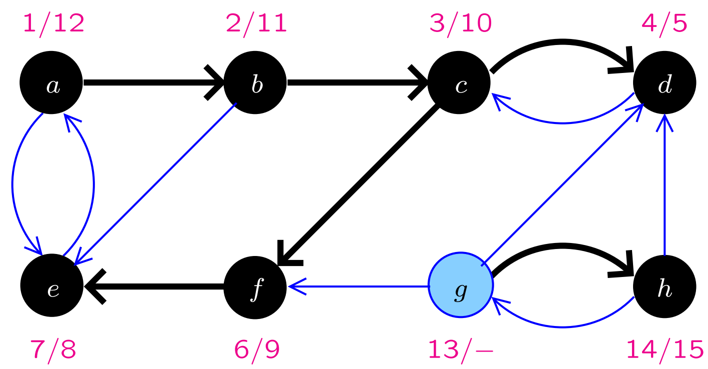

1/–
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9

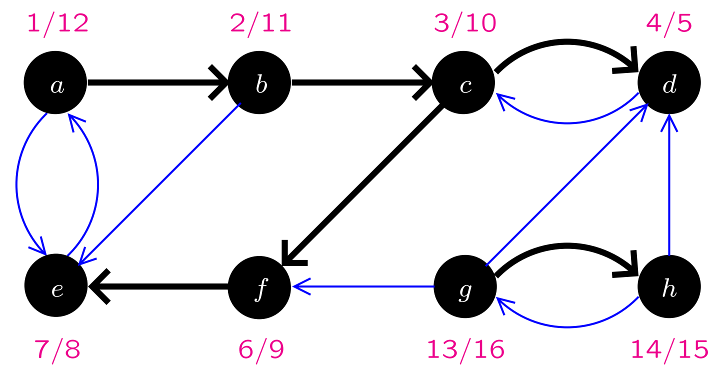

1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9


1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
13/–


1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
13/–
14/–




1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
13/–
14/15


1/12
2/11
3/10
4/5
a
b
c
d
e
f
g
h
7/8
6/9
13/16
14/15

Aikavaativuus:

- DFS-all lukuunottamatta DFS-Visit-kutsuja vie ajan $\Theta(|V|)$.
- Kutsu DFS-Visit(u) suoritetaan vain, jos u on valkoinen. Tämän jälkeen u ei enää koskaan ole valkoinen. Siis DFS-Visit(u) suoritetaan tasan kerran kullakin u.
- DFS-Visit(u) lukuunottamatta for-silmukkaa vie ajan $O(1)$.
- Kaikkien DFS-Visit-kutsujen for-silmukat yhteensä käyvät kertaalleen läpi kaikki vieruslistat, joten rekursiivisia kutsuja lukuunottamatta niissä kuluu aika $\Theta(|E|)$.

$\Rightarrow$ Aikavaativuus on kaikkiaan $\Theta(|V| + |E|)$

Tilavaativuus:

- Työtilavaativuus on vakio plus rekursion vaatima pino.
- Jokaisella rekursiotasolla on DFS-Visit(u) eri solmulla u, joten pinon syvyys on korkeintaan $|V|$.
- Syvyys $|V|$ todella esiintyy esim. jos verkko muodostuu yhdestä polusta $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$ ja läpikäynti alkaa solmusta v_1

$\Rightarrow$ Pahimman tapauksen työtilavaativuus on $\Theta(|V|)$.

Syvyyssuuntaisen läpikäynnin perusteella voidaan muodostaa verkon syvyyssuuntainen metsä. Tämä on kokoelma juurellisia puita, missä solmun u lapsina on ne solmut v, joilla DFS-Visit(u) suoraan kutsui DFS-Visit(v).

Syvyyssuuntainen metsä ei ole yksikäsitteinen, vaan riippuu valitusta solmujen ja vieruslistojen järjestyksestä.

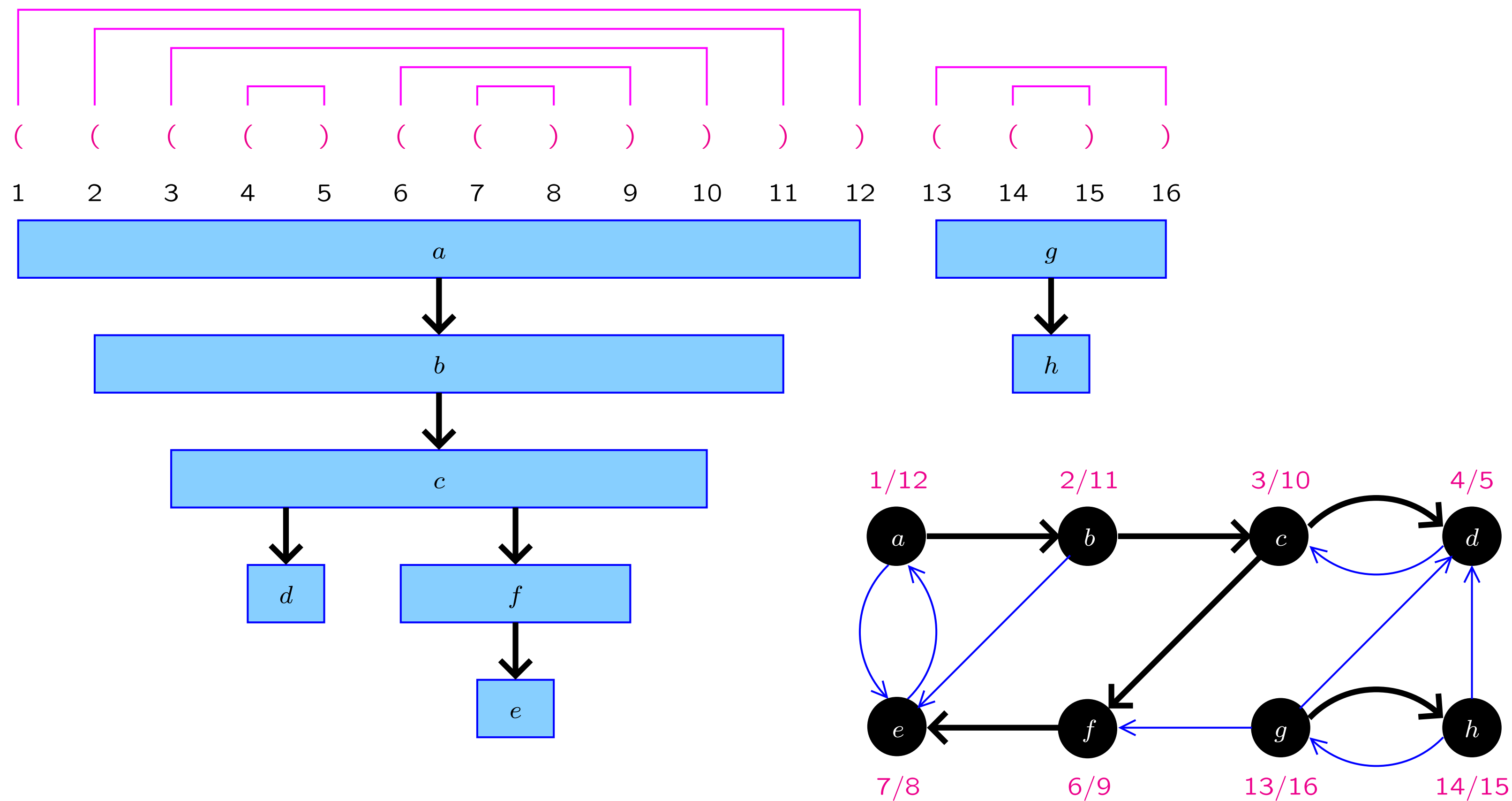
Syvyyssuuntaisen metsän tulkinta DFS-Visit-kutsukaaviona antaa suoraan seuraavan ”sulkumerkkiteoreeman” mille tahansa syvyyssuuntaiselle puulle:

Lause 7.1: Mille tahansa solmuille u ja v pätee jokin seuraavista:

1. $d[u] < d[v] < f[v] < f[u]$, ja solmu v on solmun u jälkeläinen
2. $d[v] < d[u] < f[u] < f[v]$, ja solmu u on solmun v jälkeläinen.
3. $d[u] < f[u] < d[v] < f[v]$ tai $d[v] < f[v] < d[u] < f[u]$, ja solmuista kumpikaan ei ole toisen jälkeläinen

□

Lause sanoo, että kutsujen alku- ja loppumisajat vastaavat alku- ja loppusulkuja hyvinmuodostetussa lausekkeessa.



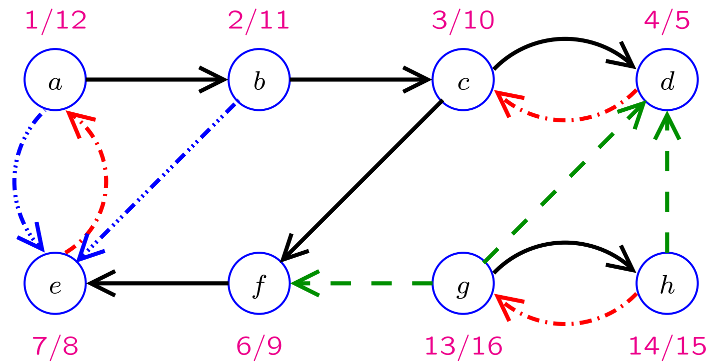
Syvyyssuuntainen puu, DFS-Visit-kutsukaavio ja "sulkumerkit".

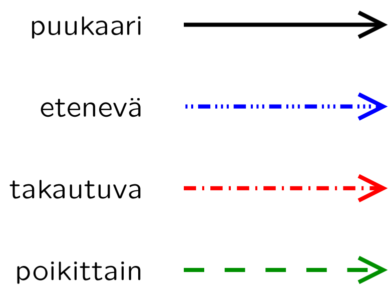
Suunnatun verkon syvyyssuuntaisen metsän perusteella kaaret voidaan jakaa neljään luokkaan: kaari (u, v) on

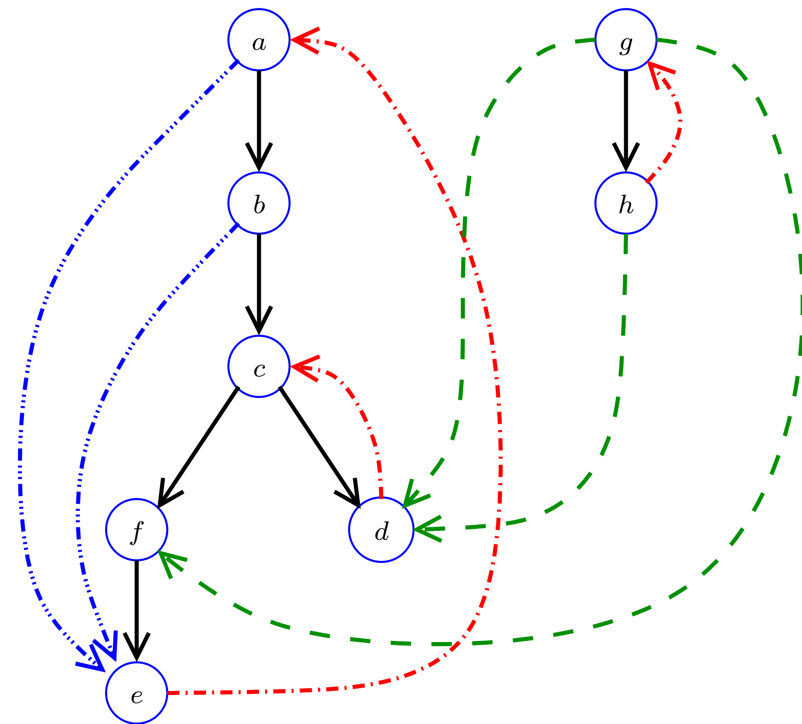
- **puukaari** jos solmu u on solmun u vanhempi
- **etenevä** jos solmu v on solmun u lapsenlapsi tai etäisempi jälkeläinen
- **takautuva** jos solmu v on solmun u esi-isä (mukaanlukien tapaus $u = v$)
- **poikittainen** jos se ei kuulu mihinkään edellisistä luokista.

Poikittaiset kaaret voivat olla puunsisäisiä tai kahden puun välisiä.

Suuntaamattomassa verkossa kaaret jaetaan puukaariin ja takautuviin kaariin: takautuvia ja eteneviä ei voi erottaa, ja poikittaisia ei voi esiintyä.







Edellisen läpikäynnin kaaret luokiteltuina ja puumaisemmin piirrettynä.

Seuraava valkosolmulause pätee suunnatuissa ja suuntaamattomissa verkoissa. Se on tärkeä jatkossa esiintyvien syvyyssuuntaiseen läpikäyntiin perustuvien algoritmien ymmärtämisessä.

Lause 7.2: Solmu v tulee solmun u jälkeläiseksi syvyyssuuntaisessa metsässä, jos ja vain jos kutsun DFS-Visit(u) alkaessa on olemassa pelkistä valkoisista solmuista koostuva polku $u \leadsto v$.

Todistus: $\Leftarrow$: Kun v on u:n jälkeläinen, olkoon $(u, w_1, \ldots, w_k, v)$ puukaaria pitkin kulkeva polku $u \leadsto v$. Sulkumerkkiteoreeman mukaan $d[u] < d[w_1] < \ldots < d[w_k] < d[v]$, joten kutsun DFS-Visit(u) alkaessa kaikki polun solmut ovat valkoisia.

$\Rightarrow$: Olkoon $(u, w_1, \ldots, w_k, v)$ valkoisista solmuista koostuva polku, kun DFS-Visit(u) alkaa. Tehdään vastaoletus, että ainakin yksi polun solmuista ei tule u:n jälkeläiseksi. Olkoon q näistä ensimmäinen. Olkoon lisäksi p solmua q edeltävä solmu polulla. Sulkumerkkiteoreeman mukaan

$$d[u] < d[p] < f[p] < f[u] < d[q] < f[q].$$

Mutta solmu q on solmun p vierussolmu eikä siis voi olla valkoinen, kun p on musta; ristiriita. □

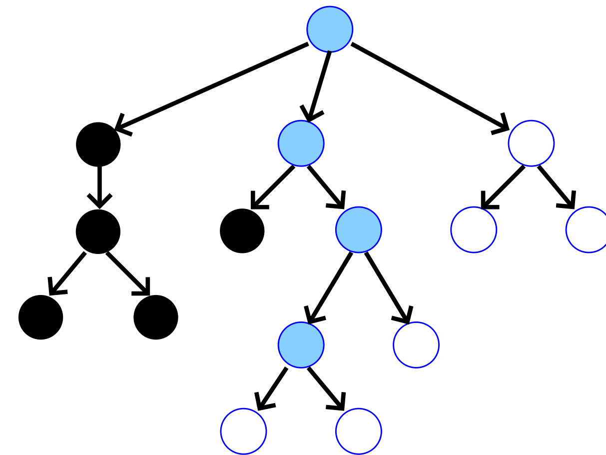
Suunnatun verkon syklittömyyden testaaminen

Suunnatussa verkossa on sykli, jos ja vain jos siinä on ainakin yksi takautuva kaari. Ongelma siis palautuu takautuvien kaarten havaitsemiseen.

Tarkastellaan verkon valmista syvyyssuuntaista puuta (eli syvyyssuuntaisen metsän yhtä komponenttia, jos niitä on useita). Merkitään siihen solmujen värit millä tahansa ajanhetkellä läpikäynnin kestäessä.

Sulkumerkkiteoreeman perusteella

- juuri on harmaa
- harmaalla solmulla on korkeintaan yksi harmaa lapsi
- mustan solmun lapset ovat mustia
- valkoisen solmun lapset ovat valkoisia



vain puukaaret merkitty

Harmaista solmuista muodostuu polku. Polun viimeinen solmu on se u, jonka DFS-Visit(u) on sillä hetkellä aktiivinen (rekursiopinon päällimmäisenä): seuraavaksi

- joko muutetaan harmaaksi jokin solmun u valkoinen lapsi, josta tulee aktiivinen
- tai muutetaan u mustaksi, ja solmun u vanhemmasta tulee aktiivinen.

Kun solmu u on aktiivinen, harmaita solmuja ovat se itse ja sen esi-isät. Siis

> verkossa on takautuva kaari, jos ja vain jos jossain vaiheessa aktiivisen solmun vieruslistasta löytyy harmaa solmu.

Saadaan seuraava algoritmi:

```
Cyclic(G)
    for jokaiselle u ∈ V
        do colour[u] ← white
    for jokaiselle u ∈ V
        do if colour[u] = white
            then if Cyclic-Visit(u)
                then return True
    return False
```

```
Cyclic-Visit(u)
    colour[u] ← gray
    for jokaiselle v ∈ Adj[u]
        do if colour[v] = gray
            then return True
           if colour[v] = white
            then if Cyclic-Visit(v)
                then return True
    colour[u] ← black
    return False
```

Selvästi pahimman tapauksen aikavaativuus on $\Theta(|V| + |E|)$ ja tilavaativuus $\Theta(|V|)$ kuten proseduurilla DFS.

Topologinen järjestäminen

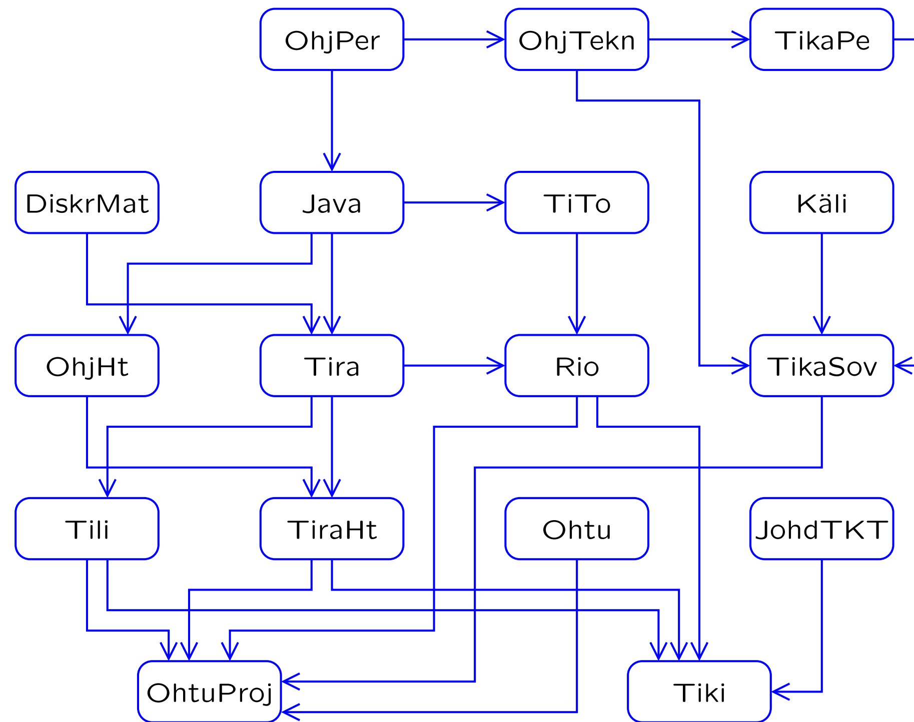
Tehtävänä on järjestää suunnatun verkon solmut topologiseen järjestykseen:

> jos $(u, v) \in E$, niin solmu u on järjestyksessä ennen solmua v.

Topologinen järjestäminen on mahdollista, jos ja vain jos verkko on syklitön (DAG). Tämä voidaan testata juuri esitetyllä algoritmilla.

Esimerkki: On annettu joukko rajoitteita "kurssi u pitää suorittaa ennen kurssia v". Opiskelija haluaa (jostain kumman syystä) suorittaa kurssit yksi kerrallaan. Tehtävänä on löytää jokin laillinen järjestys. (Tyypillisesti kelvollisia vastauksia on useita.)

Esimerkki (hieman realistisempi): missä järjestyksessä osat voidaan liittää yhteen autotehtaan kokoonpanolinjalla?



Kurssien esitietovaatimukset *voisivat* näyttää esim. tältä.

Ongelma voidaan ratkaista seuraavasti:

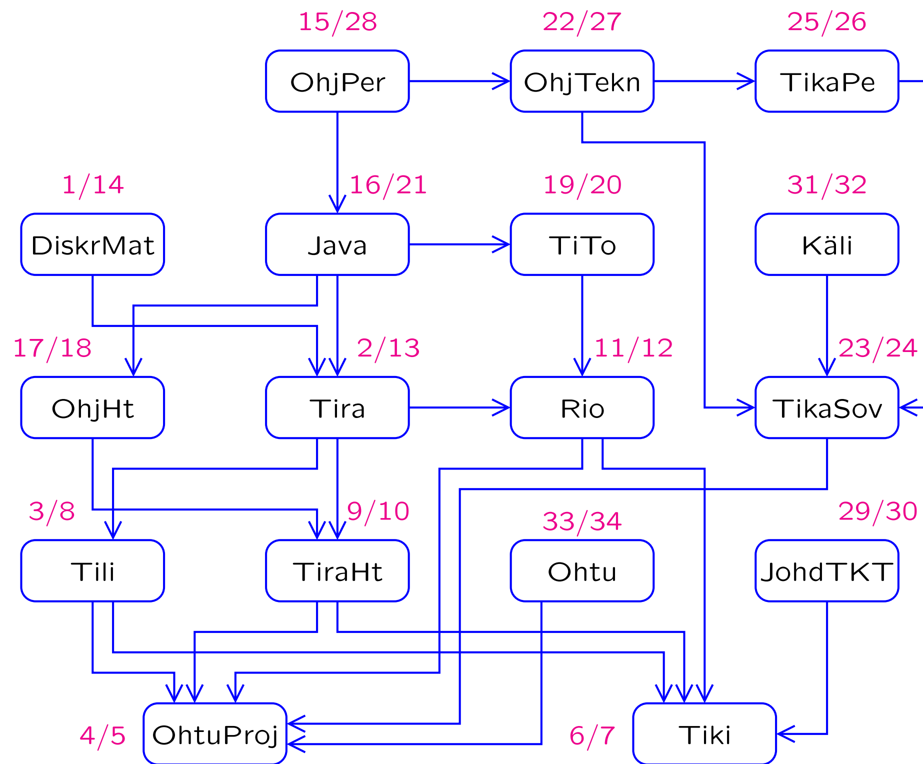
1. Suorita syvyyssuuntainen läpikäynti DFS-all(G).
2. Luettele solmut arvon $f[u]$ mukaan vähenevässä järjestyksessä.

Erillinen järjestämisvaihe voidaan välttää liittämällä solmuja tuloslistan kärkeen sitä mukaa, kun ne muuttuvat mustiksi.

Algoritmin oikeellisuus todetaan tarkastelemalla tilannetta, kun solmu v kohdataan solmun u vieruslistassa:

- Jos v on musta, niin $f[v]$ on pienempi kuin nykyinen ajanhetki $time$. Koska DFS-Visit(u) on vielä käynnissä, arvoksi $f[u]$ tulee vähintään $time$.
- Jos v on valkoinen, suoritetaan DFS-Visit(v) tässä kohdassa, eikä u muutu mustaksi, ennen kuin v muuttuu. Siis lopuksi $f[v] < f[u]$.
- Jos v on harmaa, edellä esitetyn perusteella verkossa on sykli eli syöte ei ole kelvollinen.

Siis mikäli verkko on syklitön, $f[v] < f[u]$ pätee aina, kun $(u, v) \in E$.



$f[u]$	u
34	Ohtu
32	Käli
30	JohdTKT
28	OhjPer
27	OhjTekn
26	TikaPe
24	TikaSov
21	Java
20	TiTo
18	OhjHt
14	DiskrMat
13	Tira
12	Rio
10	TiraHt
8	Tili
7	Tiki
5	OhtuProj

Eräs syvyyssuuntainen läpikäyntijärjestys (d/f-arvot) ja sen tuottama topologinen järjestys.

Kriittiset työvaiheet

Mallinnetaan ohjelmistoprojektia suunnattuna painollisena verkkona:

- solmuina projektiin liittyvät tarkistuskohdat: ensimmäinen demo valmis, suorituskykymittaukset tehty, käyttöohje valmis jne.
- kaarina tarkistuskohtien välillä tarvittavat työvaiheet
- painoina työvaiheiden kestot.

Kuinka nopeasti projekti voi valmistua? (Jätetään huomiotta eri työvaiheiden mahdollisesti tarvitsemat yhteiset resurssit jne.)

Tämä voidaan suoraan ratkaista etsimällä verkosta painavin polku, missä polun painon on sillä olevien kaarten painojen summa.

Oletus on taas, että verkko on syklitön.

Ratkaistaan ongelma laskemalla jokaiselle solmulle

$$h[u] = \text{painavimman solmusta } u \text{ alkavan polun paino.}$$

Polkujen päätepisteitä siis ei ole määritelmässä kiinnitetty. Käytännössä päätepisteinä on solmuja, joista ei lähde yhtään kaarta. Painavin polku ei tietysti yleensä ole yksikäsitteinen (ts. usea polku voi saavuttaa saman maksimaalisen painon).

Jos solmusta u ei lähde kaaria, $h[u]$ on triviaalin polun (u) paino eli 0. Koska verkko oletetaan syklittömäksi, millään polulla ei voi olla yli $|V|$ solmua, joten $d[u]$ on aina hyvin määritelty ja äärellinen.

Koko verkon painavimman polun pituus on tietysti

$$\max\{ h[u] \mid u \in V \},$$

sillä polun on alettava jostain solmusta u.

Jos $(u, v) \in E$, niin

$$h[u] \geq w(u, v) + h[v],$$

sillä painavin solmusta u alkava polku on ainakin niin painava kuin polku, joka ensin menee solmuun v ja jatkaa siitä mahdollisimman painavana. Toisaalta jos v on ensimmäinen solmu jollain painavimmalla solmusta u alkavalla polulla, niin

$$h[u] = w(u, v) + h[v].$$

Siis jos solmulla u ei ole naapureita, niin $h[u] = 0$, ja muuten

$$h[u] = \max \{ w(u, v) + h[v] \mid (u, v) \in E \} .$$

Eräs ratkaisu olisi ensin järjestää verkko topologisesti ja sitten käydä solmut läpi käänteisessä topologisessa järjestyksessä. Tällöin tiedetään varmasti, että arvoa $h[u]$ laskettaessa tiedetään $h[v]$ kaikilla v, joilla $(u, v) \in E$.

Sama tulos saadaan sijoittamalla laskut suoraan syvyyssuuntaiseen läpikäyntiin:

```
Max-Path(G)
    for jokaiselle u ∈ V
        do colour[u] ← white
    for jokaiselle u ∈ V
        do if colour[u] = white
            then DFS-Visit-2(u)
    return max { h[u] | u ∈ V }
```

```
DFS-Visit-2(u)
    colour[u] ← gray
    h[u] ← 0
    for jokaiselle v ∈ Adj[u]
        do if colour[v] = white
            then DFS-Visit-2(v)
           if h[u] < w(u, v) + h[v]
            then h[u] ← w(u, v) + h[v]
```

(Solmujen värittäminen mustaksi on turhana jätetty pois.)

Vahvasti yhtenäiset komponentit

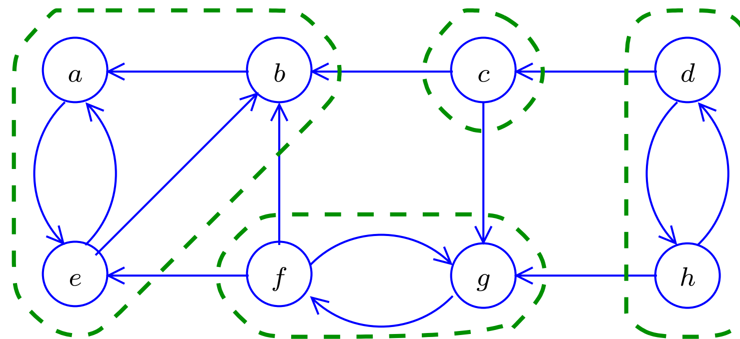
Olkoon $G = (V, E)$ suunnattu verkko. Merkitään $u \sim v$, jos verkossa G on sekä polku $u \rightsquigarrow v$ että polku $v \rightsquigarrow u$. Selvästi $\sim$ on ekvivalenssirelaatio eli toteuttaa ehdot

1. $u \sim u$ kaikilla u
2. jos $u \sim v$ niin $v \sim u$
3. jos $u \sim v$ ja $v \sim w$ niin $u \sim w$.

Tästä seuraa [Johd. diskr. mat.], että solmut voidaan jakaa sen suhteen ekvivalenssiluokkiin V_i, jolloin

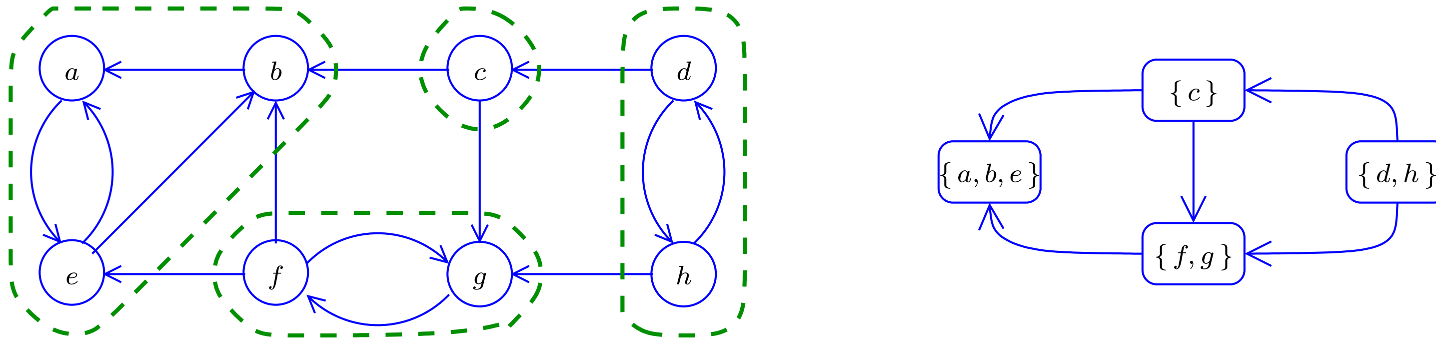
- jokaisella u pätee $u \in V_i$ tasan yhdellä i
- jos $u \in V_i$ ja $v \in V_i$, niin $u \sim v$
- jos $u \in V_i$ ja $v \in V_j$, missä $i \neq j$, niin $u \not\sim v$.

Ekvivalenssiluokkia V_i sanotaan verkon vahvasti yhtenäisiksi komponenteiksi. Verkko on vahvasti yhtenäinen, jos sillä on vain yksi vahvasti yhtenäinen komponentti V eli $u \rightsquigarrow v$ ja $v \rightsquigarrow u$ kaikilla $u, v \in V$.



Vahvasti yhtenäiset komponentit ovat

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



Vahvasti yhtenäisten komponettien perusteella voidaan muodostaa komponenttiverkko $G^{\mathsf{SCC}} = (V^{\mathsf{SCC}}, E^{\mathsf{SCC}})$, missä

- solmujoukko V^{SCC} koostuu alkuperäisen verkon vahvasti yhtenäisistä komponenteista
- $(A, B) \in E^{\mathsf{SCC}}$, jos ja vain jos $(u, v) \in E$ joillakin $u \in A$ ja $v \in B$.

Komponenttiverkko on syklitön: jos siinä olisi sykli, niin syklillä olevista solmuista muodostuisi uusi entistä isompi komponentti.

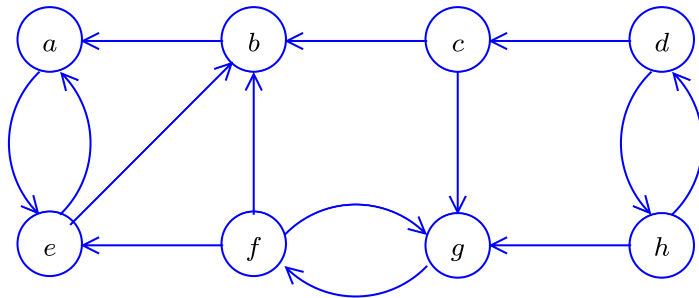
Vahvasti yhtenäiset komponentit voidaan muodostaa seuraavalla algoritmilla:

Strongly-Connected-Components(G)

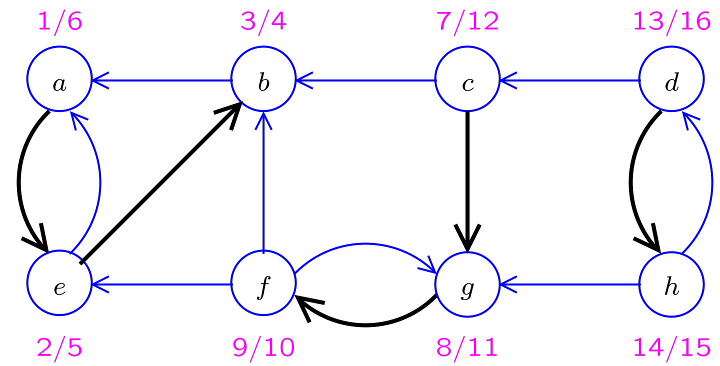
1. Suorita verkon $G = (V, E)$ syvyyssuuntainen läpikäynti. Ota talteen päättymisajat $f[u]$.
2. Muodosta verkon G transpoosi $G^{\mathsf{T}} = (V^{\mathsf{T}}, E^{\mathsf{T}})$, missä $V^{\mathsf{T}} = V$ ja $E^{\mathsf{T}} = \{\, (v, u) \mid (u, v) \in E \,\}$.
3. Suorita verkon G^{T} syvyyssuuntainen läpikäynti. Jokainen syvyyssuuntaisen metsän puu on verkon G vahvasti yhtenäinen komponentti. Aina kun yksi puu on tullut läpikäydyksi, seuraava DFS-Visit(u) alkaa f-arvoltaan suurimmasta vielä läpikäymättömästä solmusta u.

Selvästi pahimman tapauksen aikavaativuus on $\Theta(|\,V\,| + |\,E\,|)$ ja työtilavaativuus $\Theta(|\,V\,|)$ kuten syvyyssuuntaisella läpikäynnillä.

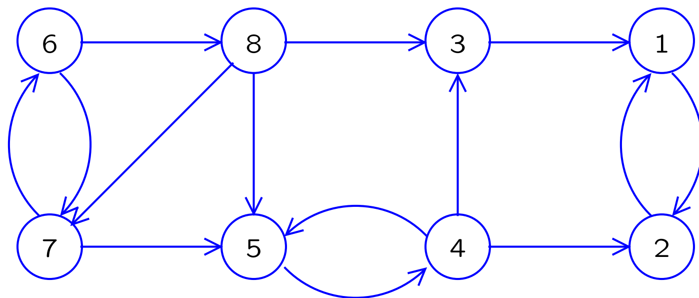
Esimerkin jälkeen tarkastellaan, miksi algoritmi antaa oikean lopputuloksen.



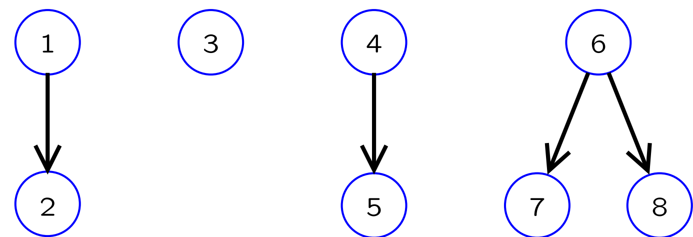
alkuperäinen verkko



syvyyssuuntainen etsintä alkuperäisessä verkossa



transpoosi, solmut numeroitu laskevan f-arvon mukaan



transpoosin syvyyssuuntainen metsä

Algoritmin toiminta perustuu seuraavaan havaintoon:

Lause 7.3: Jos $A \neq B$ ovat kaksi vahvasti yhtenäistä komponenttia ja $(u, v) \in E$ joillain $u \in A$ ja $v \in B$, niin

$$\max_{x \in A} f[x] > \max_{y \in B} f[y].$$

Siis jos komponentista A on kaari komponenttiin B, niin vaiheessa 1 komponentin B läpikäynti loppuu ennen kuin komponentin A läpikäynti.

Todistus: Kaksi tapausta sen mukaan, kumman komponentin läpikäynti alkaa ensin.

Oletetaan ensin, että komponentin A läpikäynti alkaa solmusta $x \in A$, kun kaikki komponentin B solmut ovat vielä valkoisia. Koska $x \rightsquigarrow u \rightarrow v \rightsquigarrow y$ millä tahansa $y \in B$ ja kutsun DFS-Visit(x) alkaessa kaikki nämä solmut ovat valkoisia, valkopolkulauseen mukaan jokainen $y \in B$ tulee solmun x jälkeläiseksi. Siis $f[y] < f[x]$ kaikilla $y \in B$.

Oletetaan nyt, että komponentin B läpikäynti alkaa solmusta $y \in B$, kun kaikki komponentin A solmut ovat vielä valkoisia. Koska komponenttiverkko on syklitön, solmusta y ei ole polkua mihinkään komponentin A solmuun. Siis kaikista komponentin B solmuista tulee mustia, ennen kuin mikään komponentin A solmu on edes harmaa. $\square$

Lause 7.4: Algoritmi Strongly-Connected-Components tuottaa oikein verkon vahvasti yhtenäiset komponentit.

Todistus: Todistamme induktiolla k:n suhteen, että k ensimmäistä vaiheessa 3 tuotettua puuta ovat verkon G vahvasti yhtenäisiä komponentteja.

Tapaus $k = 0$ on triviaali. Oletetaan, että ensimmäiset $k - 1$ puuta ovat oikein ja tarkastellaan puuta numero k.

Olkoon y puun numero k juuri, ja olkoon B se vahvasti yhtenäinen komponentti, johon y kuuluu.

Induktio-oletuksen mukaan aiemmat puut eivät sisältäneet komponentin B solmuja, joten kaikki komponentin B solmut ovat kutsun DFS-Visit(y) alkaessa valkoisia. Valkopolkulauseen mukaan ne tulevat solmun y jälkeläisiksi transpoosin syvyyssuuntaisessa puussa.

Pitää vielä osoittaa, että solmun y jälkeläisiksi ei tule yhtään solmua x, missä x kuuluu johonkin toiseen komponenttiin $A \neq B$.

Ei-valkoisista solmuista ei tietenkään enää tehdä kenenkään jälkeläisiä. Olkoon $A \neq B$ jokin vielä valkoisista solmuista koostuva komponentti.

Solmu y valittiin siten, että $f[y] > f[x]$ kaikilla $x \in A$. Edellisen lauseen mukaan verkossa G ei voi olla kaarta komponentista A komponenttiin B. Siis transpoosissa G^{T} ei ole kaaria komponentista B komponenttiin A.

Täten solmun y jälkeläisiksi tulee tasan kaikki komponentin B solmut. $\square$

Lyhimmät polut painotetussa verkossa

Leveyssuuntainen läpikäynti löytää annetusta lähtösolmusta pituudeltaan pienimmän polun jokaiseen saavutettavissa olevaan solmuun. Muistetaan, että pituus on polun kaarien lukumäärä.

Hieman harhaanjohtavasti painotetussa verkossa yleensä puhutaan "lyhimmästä" polusta, kun tarkoitetaan painoltaan pienintä. Muistetaan, että jos $p = (v_0, \ldots, v_k)$ on pituutta k oleva polku $v_0 \overset{p}{\rightsquigarrow} v_k$, sen paino on

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Tarkastelemme kahta eri versiota ongelmasta löytää lyhimmät polut annetusta lähtösolmusta kaikkiin muihin verkon solmuihin:

- jos kaarten painot voivat olla mielivaltaisia, ongelma ratkeaa ajassa $O(|V||E|)$ soveltamalla Bellmanin-Fordin algoritmia
- jos kaarten painot oletetaan ei-negatiivisiksi, ongelma ratkeaa ajassa $O((|V|+|E|)\log|V|)$ soveltamalla Dijkstran algoritmia.

Tapaukseen, jossa halutaan lyhin polku lähtösolmusta u vain yhteen annettuun kohdesolmuun v, ei tunneta yleisessä tapauksessa tehokkaampaa ratkaisua kuin laskea samalla kaikki lyhimmät polut lähtösolmusta.

Lisäksi

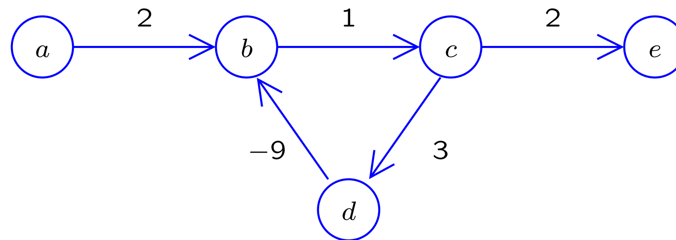
- lyhimmät polut kaikkien $|V|^2$ solmuparin välille voidaan kerralla laskea ajassa $O(|V|^3)$ soveltamalla Floydin algoritmia.

Määritellään

$$\delta(u,v) = \text{lyhimmän polun } u \rightsquigarrow v \text{ paino,}$$

mikäli lyhin polku on olemassa. Jos solmusta u ei ole polkua solmuun v, merkitään $\delta(u,v) = \infty$. Määritellään vastaavasti $w(u,v) = \infty$, jos $(u,v) \notin E$.

Jos negatiiviset painot ovat sallittuja, solmusta u voi olla solmuun v polkuja, joiden paino on mielivaltaisen pieni. Tämä edellyttää, että jollain polulla solmusta u solmuun v on kehä, jonka paino on negatiivinen. Tällöin merkitään $\delta(u,v) = -\infty$.



polku	paino
(a,b,c,e)	$2+1+2=5$
(a,b,c,d,b,c,e)	$2+1+3-9+1+2=0$
(a,b,c,d,b,c,d,b,c,e)	$2+1+3-9+1+3-9+1+2=-5$
...	...

Tarkastellaan nyt solmusta s alkavien lyhimpien polkujen etsimistä.

Kuten leveyssuuntaisessa läpikäynnissä, liitämme jokaiseen saavutettuun solmuun v osoittimen $p[v]$, jonka aiottu tulkinta on seuraava:

> Eräs lyhin polku $s \leadsto v$ saadaan kulkemalla
> - ensin lyhin polku $s \leadsto p[v]$ ja
> - sitten kaari $p[v] \to v$.

Siis jos $\pi = (s, v_1, \ldots, v_k)$ on sellainen polku, että $s = p[v_1]$ ja $v_i = p[v_{i+1}]$ kaikilla $1 \leq i \leq k - 1$, niin $\delta(s, v_k) = w(\pi)$.

Mikäli osoittimilla p todella on tämä ominaisuus, ne muodostavat lyhimpien polkujen puun. Tällöin erityisesti $\delta(s, v) = \delta(s, p[v]) + w(p[v], v)$.

Sekä Bellmanin-Fordin että Dijkstran algoritmi pitävät yllä taulukkoa d, jossa $d[v]$ on lyhimmän toistaiseksi löydetyn polun $s \leadsto v$ paino.

Siis aina $d[v] \geq \delta(s, v)$, ja algoritmin päättyessä halutaan $d[v] = \delta(v, s)$.

Algoritmin suorituksen aikana osoittimista p muodostuu polunpituuksia d vastaava puu:

$$d[v] = d[p[v]] + w(p[v], v).$$

Suorituksen päättyessä siis p antaa halutun puun.

Taulukot d ja p alustetaan seuraavasti:

```
Initialise-Single-Source(G, s)
    for kaikilla v ∈ V
        do d[v] ← ∞
           p[v] ← Nil
    d[s] ← 0
```

Siis aluksi mitään polkuja ei ole löydetty.

Keskeinen päivitysoperaatio on kaaren (u, v) löysääminen (relaxation):

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v)
        then d[v] ← d[u] + w(u, v)
             p[v] ← u
```

Löydettyjen polkujen perusteella u näyttäisi olevan etäisyydellä $d[u]$ ja v etäisyydellä $d[v]$. Jos kuitenkin $d[v] - d[u] > w(u, v)$, kaaressa (u, v) on "jännitystä", jonka Relax(u, v, w) poistaa siirtämällä solmua v lähemmäs.

Kun operaatiossa Relax(u, v, w) pienennetään arvoa $d[v]$, niin vastaavasti solmusta v alkaviin kaariin (v, r) voi syntyä jännitteitä eli tilateita

$$d[r] > d[v] + w(v, r).$$

Erityisesti näin voi käydä, vaikka kaari (v, r) olisi aiemmin löysätty, jolloin aiempi löysäys tavallaan osoittautuu hukkatyöksi.

Sekä Bellmanin-Fordin että Dijkstran algoritmin runko on seuraava:

```
Shortest-Paths(G, w, s)
    Initialise-Single-Source(G, s)
    while verkossa on jännityksiä
        do valitse kaari (u, v)
           Relax(u, v, w)
```

Huomaa, että silmukassa pätee invariantti $d[v] \geq \delta(s, v)$ kaikilla $v \in V$. Lisäksi operaation Relax(u, v, w) jälkeen $u = p[v]$ ja $d[v] = d[u] + w(u, v)$.

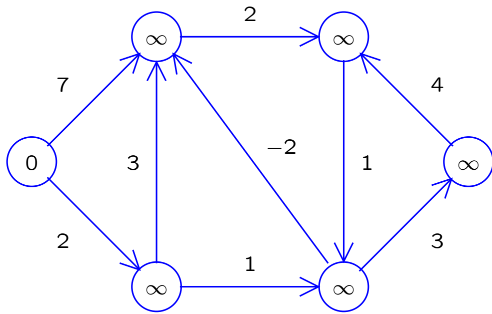
Algoritmien ero on löysäämisoperaatioiden järjestämisessä. Dijkstran algoritmi järjestää operaatiot niin tehokkaasti, että kutakin kaarta tarvitsee löysätä vain kerran. Tämän onnistuminen voidaan kuitenkin taata vain, jos verkossa ei ole negatiivisia painoja.

Bellmanin-Fordin algoritmi

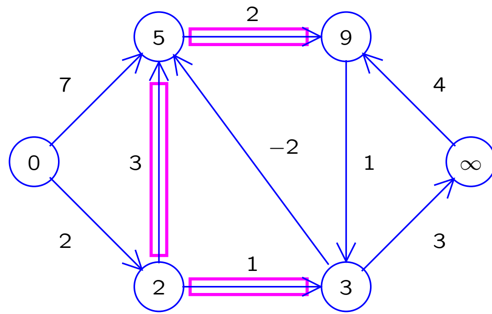
Löysäämisjärjestystä ei yritetäkään optimoida. Kaikki kaaret käydään järjestyksessä läpi useita kertoja, kunnes tulos on valmis.

Kuten pian näemme, $|V|-1$ läpikäyntiä riittää, ellei verkossa ole negatiivisen painoisia syklejä. Jos tämän ajan jälkeen on jännitteitä jäljellä, algoritmi palauttaa False merkiksi negatiivisen painoisesta syklistä.

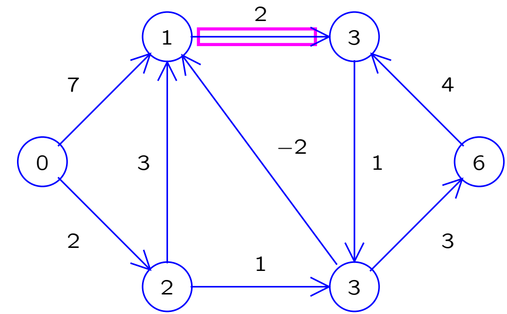
```
Bellman-Ford(G, w, s)
    Initialise-Single-Source(G, s)
    for i ← 1 to |V| − 1
        do for jokaiselle kaarelle (u, v) ∈ E
               do Relax(u, v, w)
    for jokaiselle kaarelle (u, v) ∈ E
        do if d[v] > d[u] + w(u, v)
             then return False
    return True
```



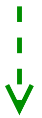
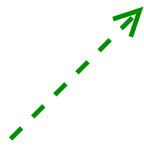
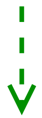
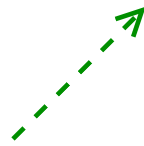
∞
∞
0
∞
∞
∞
2
7
4
3
−2
1
2
1
3

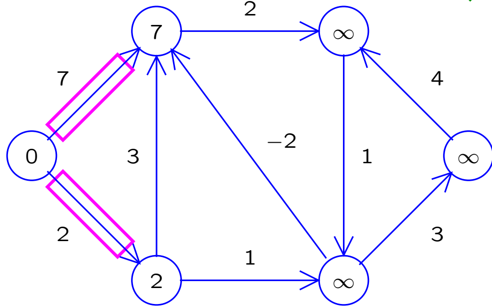


5
9
0
∞
2
3
2
7
4
3
−2
1
2
1
3

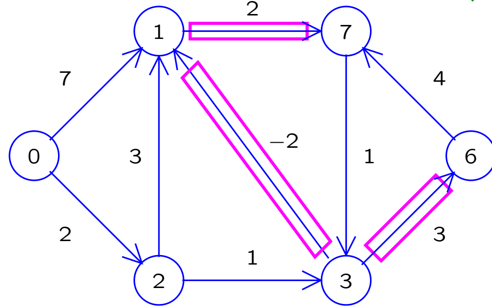


1
3
0
6
2
3
2
7
4
3
−2
1
2
1
3



7
∞
0
∞
2
∞
2
7
4
3
−2
1
2
1
3



1
7
0
6
2
3
2
7
4
3
−2
1
2
1
3

Seuraava lemma sanoo oleellisesti, että $|V|-1$ iteraatiota riittää kaikkien jännitteiden löysäämiseen, ellei verkossa ole negatiivisen painoisia kehiä:

Lemma 7.5: Jos

- solmusta s on polku solmuun v ja
- solmusta s ei voi saavuttaa mitään negatiivisen painoista kehää

niin algoritmin Bellman-Ford suorituksen päättyessä $d[v] = \delta(s, v)$.

Todistus: Oletusten mukaan jokin yksinkertainen polku $\pi = (s, v_1, \ldots, v_{k-1}, v)$ on lyhin polku $s \rightsquigarrow v$. Merkitään $s = v_0$ ja $v = v_k$. Nyt $\pi_i = (v_0, \ldots, v_i)$ on lyhin polku $s \rightsquigarrow v_i$ kaikilla $1 \leq i \leq k$.

Huomaa, että $d[u]$ ei koskaan kasva millään u. Invariantista $d[u] \geq \delta(s, u)$ seuraa, että jos ehto $d[u] = \delta(s, u)$ kerran tulee voimaan, se pysyy voimassa.

Osoitetaan induktiolla indeksin i suhteen, että kun for-silmukkaa on iteroitu i kertaa, pätee $d[v_i] = \delta(s, v_i)$. Koska $k \leq |V| - 1$, tästä seuraa väite.

Tapaus $i = 0$ pätee selvästi.

Oletetaan, että $d[v_i] = \delta(s, v_i)$ pätee iteraation i jälkeen. Kun on seuraavan kerran suoritettu Relax(v_i, v_{i+1}, w), pätee

$$\begin{aligned} \delta(s, v_{i+1}) &\leq d[v_{i+1}] \\ &\leq d[v_i] + w(v_{i-1}, v_i) \\ &= \delta(s, v_i) + w(v_{i-1}, v_i) \\ &= \delta(s, v_{i+1}), \end{aligned}$$

missä viimeinen askel perustuu oletukseen, että $s \rightsquigarrow v_i \rightarrow v_{i+1}$ on lyhin polku. $s \rightsquigarrow v_{i+1}$. □

Seuraavat kaksi lemmaa perustelevat, että algoritmin lopussa tehtävä testi menee oikein.

Lemma 7.6: Jos solmusta s ei voi saavuttaa negatiivisen painoisia kehiä, Bellman-Ford palauttaa True.

Todistus: Edellisen lemman mukaan lopuksi pätee $d[v] = \delta(s, v)$ kaikilla $v \in V$. Siis kaikilla $u \in V$ saadaan

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \\ &= d[u] + w(u, v). \end{aligned}$$

□

Lemma 7.7: Jos verkossa on kehä $c = (v_0, v_1, \ldots, v_k)$, missä $v_k = v_0$ ja

- v_0 on saavutettavissa solmusta s ja
- kehän paino $w(c) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$ on negatiivinen

niin Bellman-Ford palauttaa False.

Todistus: Tehdään vastaoletus, että Bellman-Ford palauttaa True. Tällöin erityisesti $d[v_{i+1}] \leq d[v_i] + w(v_i, v_{i+1})$ kaikilla i. Saadaan

$$\begin{aligned} d[v_k] &\leq d[v_{k-1}] + w(v_{k-1}, v_k) \\ &\leq d[v_{k-2}] + w(v_{k-2}, v_{k-1}) + w(v_{k-1}, v_k) \\ &\cdots \\ &\leq d[v_0] + \sum_{i=1}^{k} w(v_{i-1}, v_i). \end{aligned}$$

Koska $v_0 = v_k$ ja $d[v_0] < \infty$, pätee $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$, mikä on ristiriita oletuksen $w(c) < 0$ kanssa. $\square$

Edellisten kolmen lemman perusteella

- Bellman-Ford palauttaa True, jos ja vain jos solmusta s ei voi saavuttaa negatiivisen painoista kehää ja
- tällöin lopuksi pätee $d[v] = \delta(s, v)$ kaikilla $v \in V$.

Jos halutaan ehto $d[v] = \delta(s, v)$ voimaan joka tapauksessa, pitää lopuksi käydä korjaamassa $d[v] \leftarrow -\infty$ niillä v, joihin pääsee solmusta s jonkin negatiivisen painoisen kehän kautta.

Edellä esitetyn mukaan jokaiselta negatiivisen painoiselta kehältä löytyy $v_i = u$ ja $v_{i+1} = v$, joilla $d[v] > d[u] + w(u, v)$. Korjaus tehdään asettamalla esim. syvyyssuuntaisella läpikäynnillä $d[r] \leftarrow -\infty$ kaikilla r, jotka ovat saavutettavissa tällaisesta v.

Dijkstran algoritmi

Bellmanin-Fordin algoritmissa vaikka kaari (u, v) oli löysennetty, siihen saattoi tulla uudelleen jännitettä, jos $d[u]$ pieneni.

Mieluummin haluaisimme löysentää kaaren (u, v) vasta, kun solmulle u pätee $d[u] = \delta(s, u)$, eikä siis d enää pienene.

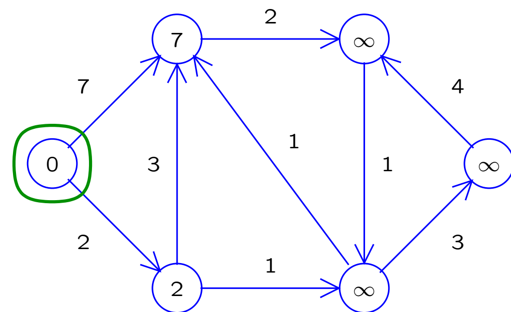
Kääntäen emme halua, että arvon $d[v]$ pienentäminen jännittää uudelleen kaaria, jotka on jo löysennetty.

Dijkstran algoritmissa pidämme yllä joukkoa $S \subseteq V$, jonka alkioille $v \in S$ tiedetään että $d[v] = \delta(s, v)$. Joukkoa S laajennetaan lisäämällä siihen aina se $v \in V - S$, jolla $d[v]$ on pienin mahdollinen.

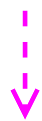
Algoritmi on siis seuraava:

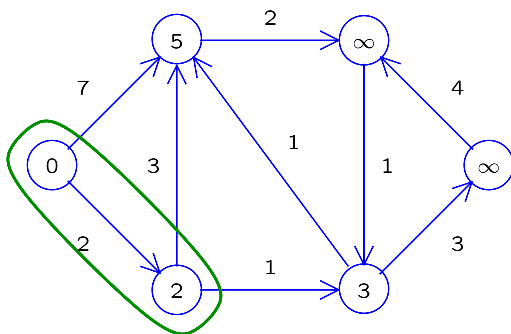
```
Dijkstra(G, w, s)
    Initialise-Single-Source(G, s)
    S ← ∅
    Q ← tyhjä min-prioriteettijono
    for kaikille v ∈ V
        do Insert(Q, v, d[v])
    while Q ei ole tyhjä
        do u ← Delete-Min(Q)
           S ← S ∪ { u }
           for kaikille v ∈ Adj[u]
               do Relax(u, v, w)
                  Decrease-Key(Q, v, d[v])
```

Prioriteettijonoon Q talletetaan solmut v avaimen $d[u]$ mukaisesti. Tällöin operaatio Decrease-Key olettaa, että prioriteettijonon toteutuksessa on käytetty kahvoja (s. 312).

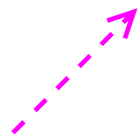


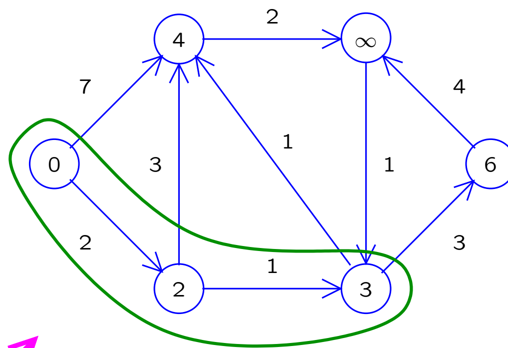
7
2
∞
7
4
0
3
1
1
∞
2
3
2
1
∞



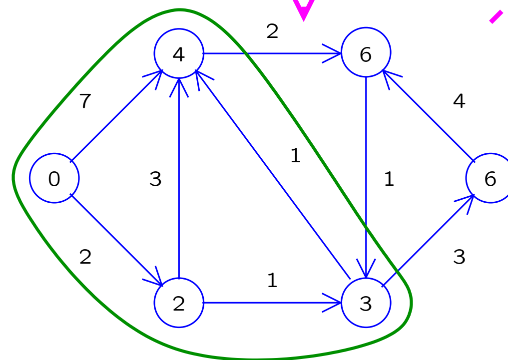


5
2
∞
7
4
0
3
1
1
∞
2
3
2
1
3

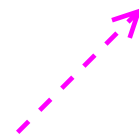


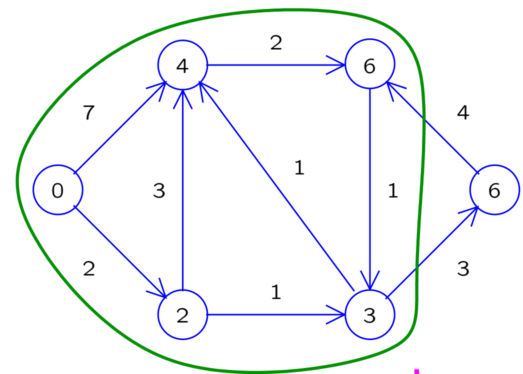


4
2
∞
7
4
0
3
1
1
6
2
3
2
1
3

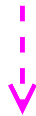


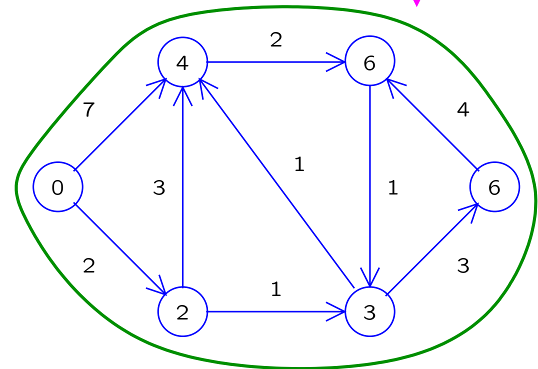
4
2
6
7
4
0
3
1
1
6
2
3
2
1
3





4
2
6
7
4
0
3
1
1
6
2
3
2
1
3





4
2
6
7
4
0
3
1
1
6
2
3
2
1
3

Ennen oikeellisuustarkasteluja arvioidaan algoritmin aikavaativuutta. Oletetaan prioriteettijono toteutetuksi kekona. Siis yksittäinen operaatio sujuu ajassa $O(\log |V|)$.

Aikavaativuutta dominoi selvästi while-silmukka. Lukuunottamatta sisintä for-silmukkaa tässä tehdään yksi Delete-Min verkon jokaista solmua kohti. Siis tämän osan aikavaativuus on $O(|V| \log |V|)$.

Sisimmässä for-silmukassa käydään läpi solmun v vieruslista. Tämä suoritetaan tasan kerran jokaiselle solmulle v, jolloin for-silmukan iteraatioita yhteensä yli kaikkien while-iteraatioiden tulee vieruslistojen yhteispituuden verran eli $|E|$. Yksi iteraatio vie ajan $O(\log |V|)$, eli yhteensä aikaa kuluu $O(|E| \log |V|)$.

Kokonaisajaksi saadaan $O((|V| + |E|) \log |V|)$. Jos verkko on yhtenäinen, mikä on yleensä kiinnostavin tapaus, pätee $|E| \geq |V| - 1$. Aikavaativuus on tällöin $O(|E| \log |V|)$.

Työtilavaativuus on selvästi $O(|V|)$.

Dijkstran algoritmissa Decrease-Key-operaatioita tehdään $O(|E|)$ kappaletta ja muita prioriteettijono-operaatioita $O(|V|)$ kappaletta. Usein $|E| \gg |V|$. Tämä motivoi etsimään prioriteettijonolle toteutusta, jossa Decrease-Key olisi muita operaatioita halvempi.

Tällainen toteutus onkin löytynyt: Fibonacci-keko. Kun prioritettijono toteutetaan Fibonacci-kekona, Dijkstran algoritmin asymptoottiseksi aikavaativuudeksi saadaan $O(|E| + |V| \log |V|)$. Fibonacci-keko on kuitenkin hyvin monimutkainen tietorakenne. Toteuttaminen on vaikeaa ja vakiokertoimet niin suuria, että tällä menetelmällä on lähinnä teoreettinen arvo.

Dijkstran algoritmin oikeellisuus todetaan helpoimmin osoittamalla while-silmukalle seuraava invariantti:

kaikilla $v \in S$ pätee $d[v] = \delta(s, v)$.

Alustusten jälkeen tämä selvästi pätee. Jos tämä pätee vielä lopuksi, kun $S = V$, algoritmi toimii oikein.

Pitää siis enää osoittaa, että invariantti pysyy voimassa silmukkaa iteroitaessa. Tehdään vastaoletus, että jollain u pätee $d[u] \neq \delta(s, u)$ sen kierroksen jälkeen, jolla on asetettu $S \leftarrow S \cup \{ u \}$. Valitaan ensimmäinen tällainen u ja johdetaan ristiriita.

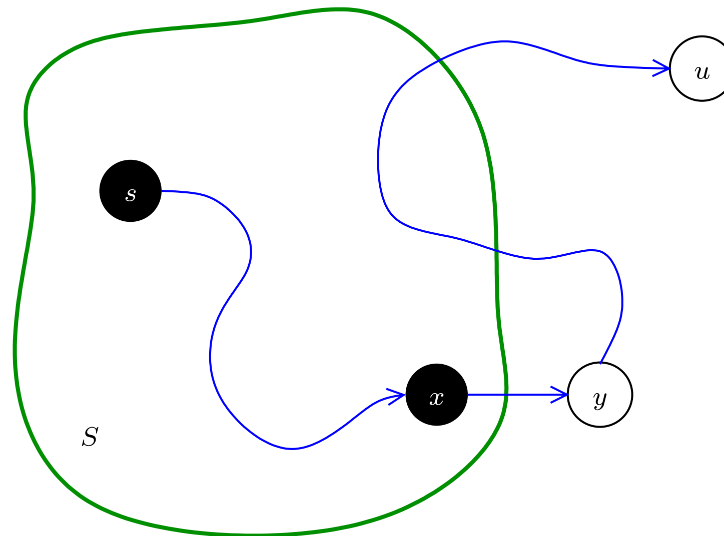
Jatkossa S ja d tarkoittavat näiden muuttujien arvoa juuri ennen sijoitusta $S \leftarrow S \cup \{ u \}$.

Olkoon nyt π lyhin polku $s \rightsquigarrow u$. Koska $s \in S$ ja $u \notin S$, polulla π on ainakin yksi kaari (x, y), jolla $x \in S$ ja $y \notin S$. Valitaan näistä ensimmäinen.

Polku π voidaan nyt jakaa osiin

$$s \rightsquigarrow x \rightarrow y \rightsquigarrow u,$$

missä osa $s \rightsquigarrow x$ sisältää vain joukon S solmuja ja $y \notin S$. Osapoluista $s \rightsquigarrow x$ ja $y \rightsquigarrow u$ kumpikin voi olla yhden solmun mittainen.



Osoitamme ensin, että sijoitettaessa $S \leftarrow S \cup \{ u \}$ pätee $d[y] = \delta(s, y)$.

Tämä seuraa siitä että

- polku $s \leadsto x \rightarrow y$ on lyhin polku $s \leadsto y$
- $d[x] = \delta(s, x)$, koska $x \in S$ ja u oletettiin ensimmäiseksi ehdon rikkojaksi
- lisättäessä x joukkoon S suoritettiin Relax(x, y, w), jolloin $d[y]$ sai arvon $d[x] + w(x, y) = \delta(s, y)$.

Siis vaikka onkin $y \notin S$, pätee silti

- $d[y] = \delta(s, y)$.

Lisäksi

- $\delta(s, y) \leq \delta(s, u)$, sillä lyhin polku $s \rightsquigarrow y$ on lyhimmän polun $s \rightsquigarrow u$ alkuosa ja oletamme että kaarten painot ovat ei-negatiivisia
- $\delta(s, u) \leq d[u]$ kuten aina ja kaikilla solmuilla.

Siis

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u].$$

Toisaalta solmun u valinnan perusteella

$$d[u] \leq d[y].$$

Siis itse asiassa pätee yhtälönä

$$d[y] = \delta(s, y) = \delta(s, u) = d[u],$$

eli u ei rikokaan ehtoa $d[u] = \delta(s, u)$; ristiriita.

Siis invariantti pätee ja algoritmi tuottaa oikeat d-arvot.

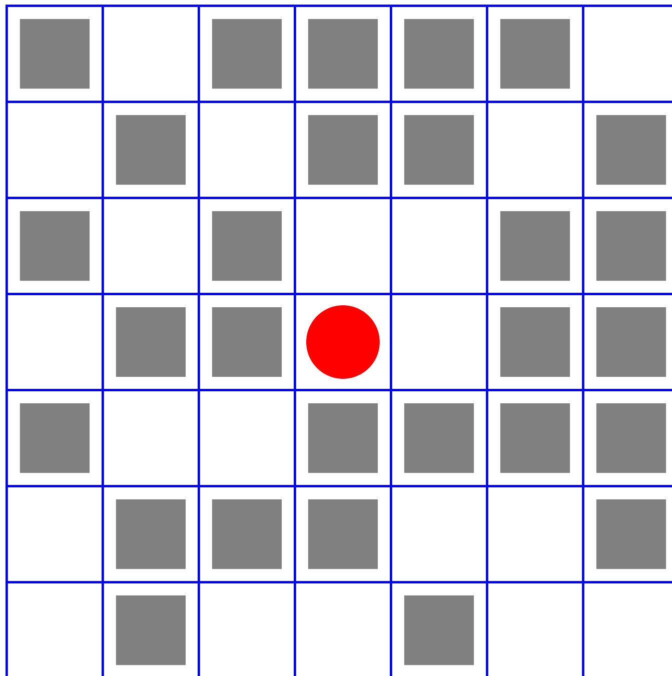
Sovellusesimerkki: vankilapako

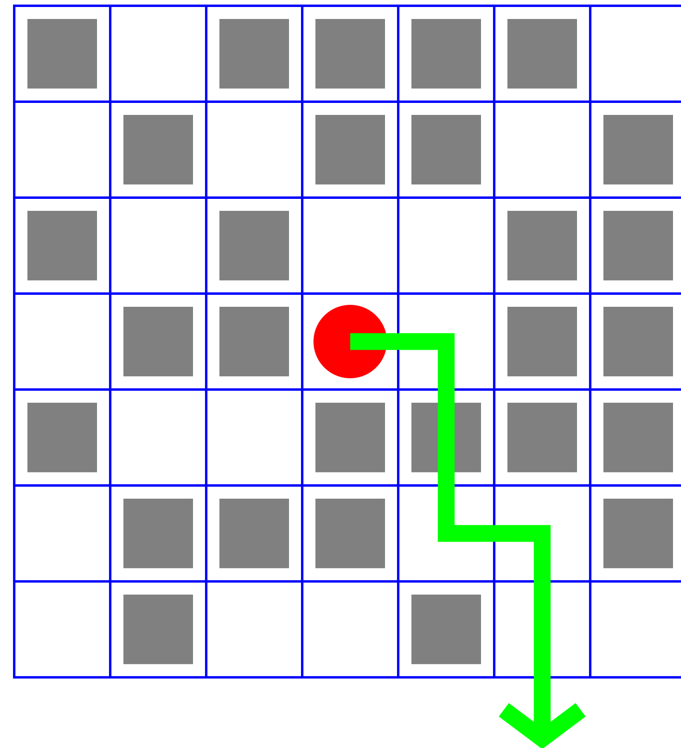
Esitetään "vankila" $(2m + 1) \times (2m + 1)$-ruudukkona $B[0 \,..\, 2m, 0 \,..\, 2m]$, missä ruudussa (i, j) on joko seinä ($B[i, j] = 1$) tai käytävä ($B[i, j] = 0$).

Lähtöpaikka on keskiruutu (m, m). Tavoitteena on päästä ruudukon reunalle eli johonkin ruutuun (i, j), missä i tai j on 0 tai $2m + 1$.

Ruudusta voidaan siirtyä mihin tahansa sen neljästä naapuriruudusta (pohjoinen, itä, etelä, länsi). Jos kohderuudussa on seinä, siihen pitää kaivautua. Jos kohderuudussa on käytävä, siihen voi hiipiä.

Paras reitti minimoi ensisijaisesti kaivautumisten määrää. Kaivautumisten määrän ollessa sama hiipimisten määrä ratkaisee.





Vankila ja optimaalinen pakoreitti, jossa yksi kaivautuminen ja neljä hiipimistä.

Motivaatio: vankilapako on (yli)yksinkertaistettu versio VLSI-piirisuunnittelusta.

Komponentit sijoitetaan piisirulle sopivasti valitun ruudukon pisteisiin.

Komponenttien väliset kytkennät vedetään suorina tai suorakulmaisina.

Minimoitavia suureita ovat mm. yhteyksien pituus ja kytkentöjen risteyskohtien lukumäärä.

Jos halutaan lisätä piiriin yksi yhteys, ongelma vastaa vankilapakoa. Yhteyden vetäminen tyhjän tilan läpi on "hiipimistä". Toisen yhteyden kanssa risteäminen vaatii "kaivautumista".

Mutta käytännössä VLSI-suunnittelun kriteerit ovat monimutkaisempia ja optimointiongelmat usein algoritmisesti hankalia (NP-täydellisiä; ks. *Laskennan vaativuus*).

Esitämme vankilan painotettuna verkkona, jossa on $(2m+1)^2+1$ solmua:

- yksi solmu jokaiselle sijainnille (i,j), missä $0 \le i,j \le 2m$
- yksi lisäsolmu t esittämään vankilan ulkopuolista tilaa

ja $4 \cdot (2m+1)^2$ kaarta:

- Jos $p = (i_1, j_1)$ ja $q = (i_2, j_2)$ missä joko $i_2 = i_1$ ja $j_2 = j_1 \pm 1$ tai $j_2 = j_1$ ja $i_2 = i_1 \pm 1$, niin (p,q) on kaari. Kaaren paino on
  - 1 jos q on käytävää
  - $(2m+1)^2+1$ jos q on muuria.
- Jos $p = (i,j)$ missä i tai j on 0 tai $2m$, niin (p,t) on kaari ja $w(p,t) = 0$.

Kaarten painot on valittu siten, että yksikin kaivautuminen painaa enemmän kuin pisin mahdollinen hiiviskeleminen. Siis lyhin polku $(m,m) \rightsquigarrow t$ vastaa optimaalista pakoreittiä.

Tässä mallissa kaivautumisesta laskutetaan, vaikka kuljettaisiin toistamiseen jo aiemmin kaivetusta kohdasta. Tämä ei vaikuta ratkaisuun, koska optimireitti ei käy missään paikassa kahta kertaa.

Vankilapako-ongelma voidaan siis ratkaista etsimällä Dijkstran algoritmilla edellä kuvatusta verkosta lyhimmät polut solmusta (m, m) lähtien.

Toteusteknisiä huomioita:

- Koska halutaan vain polku solmuun t, algoritmin suoritus voidaan keskeyttää, kun t tulee lisätyksi joukkoon S.
- Vieruslistoja ja painoja ei tarvitse tallettaa, vaan ne voidaan laskea lennosta aina tarvittaessa.

Kaivautumisen ja hiiviskelyn ero voidaan esittää suoremminkin. Valitaan kaaripainoksi kokonaislukupareja $\langle x, y \rangle$, missä $x, y \in \mathbb{N}$.

- Jos (p, q) on kaari ja q on käytävää, niin $w(p, q) = \langle 0, 1 \rangle$.
- Jos (p, q) on kaari ja q on muuria, niin $w(p, q) = \langle 1, 0 \rangle$.
- Jos (p, t) on kaari, niin $w(p, t) = \langle 0, 0 \rangle$.

Määritellään tällaisille painoille

yhteenlasku: $\langle a, b \rangle + \langle c, d \rangle = \langle a + c, b + d \rangle$
suuruusjärjestys: $\langle a, b \rangle < \langle c, d \rangle$, jos ja vain jos

$a < c$ tai ($a = c$ ja $b < d$).

Polun paino on edelleen sen kaarten painojen summa.

Painoltaan pienin polku antaa taas optimaalisen pakoreitin. Se voidaan löytää Dijkstran algoritmilla, kun vain yhteenlaskut ja suuruusvertailut muokataan ylläesitettyyn muotoon.

Kaikki lyhimmät polut

Merkintöjen yksinkertaistamiseksi oletetaan, että solmujoukko on $V = \{1, \ldots, n\}$, missä n on solmujen lukumäärä. Halutaan muodostaa $n \times n$-matriisi D, missä $D[i, j]$ on lyhimmän polun paino $i \leadsto j$.

Jos kaarten painot ovat ei-negatiivisia, ongelma voidaan ratkaista suorittamalla Dijkstran algoritmi n kertaa. Olettamalla prioriteettijono toteutetuksi kekona saadaan aikavaativuudeksi $O(n \cdot (n + m) \log n) = O((n^2 + nm) \log n)$, missä $m = |E|$. Jos verkko on tiheä eli $m = \Theta(n^2)$, tämä on $O(n^3 \log n)$.

Floydin-Warshallin (tai vain Floydin) algoritmi ratkaisee ongelman ajassa $O(n^3)$, mikä siis tiheillä verkoilla on asymptoottisesti parempi kuin Dijkstran toistaminen. Lisäksi Floyd-Warshall

- sallii negatiiviset painot (mutta ei negatiivisia kehiä)
- on helppo toteuttaa; vakiokertoimet ovat pieniä.

Verkko oletetaan annetuksi vierusmatriisina $A[1..n, 1..n]$. Oletetaan $A[i,j] = \infty$, jos $(i,j) \notin E$.

Laskennan välivaiheina muodostetaan matriisit $D^{(k)}[1..n, 1..n]$, $k = 0, \dots, n$, missä

> $D^{(k)}[i,j]$ on pienin sellaisen polun $i \to v_1 \to \dots \to v_l \to j$ paino, missä kaikki välisolmut $v_1, \dots, v_l$ ovat joukossa $\{1, \dots, k\}$.

Siis erityisesti

- $D^{(0)}[i,j] = A[i,j]$, koska tapauksessa $k = 0$ sallittujen välisolmujen joukko on tyhjä
- $D^{(n)} = D$ eli haluttu lopputulos, koska poluille ei ole mitään rajoituksia.

Kuten kohta nähdään, matriisista $D^{(k-1)}$ saadaan lasketuksi matriisi $D^{(k)}$ ajassa $O(n)$. Tämä on esimerkki dynaamisena ohjelmointina (tai taulukointina) tunnetusta algoritmitekniikasta.

Olkoon nyt p painoltaan pienin sellainen polku $i \overset{p}{\leadsto} j$, jonka välisolmut kuuluvat joukkoon $\{1, \ldots, k\}$. Siis $D^{(k)}[i,j]$ on polun p paino. Koska verkossa ei oletuksen mukaan ole negatiivisia kehiä, voimme olettaa, että k esiintyy polulla p korkeintaan kerran.

1. Jos solmu k ei esiinny polulla p, niin välisolmut kuuluvat joukkoon $\{1, \ldots, k-1\}$. Tässä tapauksessa

$$D^{(k)}[i,j] = D^{(k-1)}[i,j].$$

2. Jos solmu k esiintyy polulla p, niin p voidaan jakaa osiin $i \overset{r}{\leadsto} k \overset{s}{\leadsto} j$, missä r on lyhin polku $i \leadsto k$ ja s on lyhin polku $k \leadsto j$, kun sallittujen välisolmujen joukko on $\{1, \ldots, k-1\}$. Tässä tapauksessa

$$D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j].$$

Yhdistämällä nämä vaihtoehdot saamme päivityssäännön

$$D^{(k)}[i,j] = \min\left\{ D^{(k-1)}[i,j], \quad D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \right\}.$$

Saamme algoritmin raakaversion

```
Floyd-Warshall-0.1(A)
    for i ← 1 to n
        do
            for j ← 1 to n
                do if i = j
                    then D^(0)[i, j] ← 0
                    else D^(0)[i, j] ← A[i, j]
    for k ← 1 to n
        do
            for i ← 1 to n
                do
                    for j ← 1 to n
                        do D^(k)[i, j] ← min { D^(k-1)[i, j],
                                               D^(k-1)[i, k] + D^(k-1)[k, j] }
```

Aikavaativuus on selvästi $\Theta(n^3)$.

Tilavaativuus on tässä versiossa $\Theta(n^3)$, mutta apumatriisien $D^{(k)}$ käyttöä voidaan tehostaa.

Tilantarve saadaan helposti pudotetuksi luokkaan $\Theta(n^2)$ toteamalla, että matriisin $D^{(k)}$ laskemiseen ei enää tarvita matriiseja $D^{(0)}, D^{(1)}, \ldots, D^{(k-2)}$. Riittää siis pitää muistissa kaksi viimeisintä matriisia $D = D^{(k)}$ ja $D' = D^{(k-1)}$.

Tästäkin voidaan säästää puolet: tarvitaan vain yksi matriisi D, jonka päivitys on

$$D[i, j] \leftarrow \min \{ D[i, j], D[i, k] + D[k, j] \} .$$

Tämä seuraa siitä, että kaikilla i, j ja k pätee

$$D^{(k)}[i, k] = D^{(k-1)}[i, k] \qquad \text{ja} \qquad D^{(k)}[k, j] = D^{(k-1)}[k, j].$$

Solmuun k rajautuvissa poluissa ei nimittäin ole väliä, sallitaanko k välisolmuna.

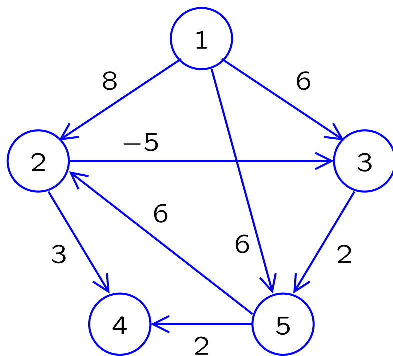
Lisätään algoritmiin vielä kirjan pitäminen itse poluista eikä vain painoista.

Samaan tapaan kuin aiemmin, muodostetaan taulukko $P[1..n, 1..n]$, missä $P[i,j]$ kertoo viimeistä edellisen solmun lyhimmällä polulla $i \rightsquigarrow j$. Lyhin polku $i \rightsquigarrow j$ siis tulostetaan takaperin seuraavasti:

```
Print-Path(i, j)
    print j
    while j ≠ i
        do j ← P[i, j]
            print j
```

Saadaan ajassa $\Theta(n^3)$ ja työtilassa $\Theta(n^2)$ toimiva algoritmi

```
Floyd-Warshall(A)
    for i ← 1 to n
        do
            for j ← 1 to n
                do if i = j
                    then D[i, j] ← 0
                    else  D[i, j] ← A[i, j]
                  if i = j or A[i, j] = ∞
                    then P[i, j] ← Nil
                    else  P[i, j] ← i
    for k ← 1 to n
        do
            for i ← 1 to n
                do
                    for j ← 1 to n
                        do if D[i, k] + D[k, j] < D[i, j]
                            then D[i, j] ← D[i, k] + D[k, j]
                                 P[i, j] ← P[k, j]
```



A	1	2	3	4	5
1	∞	8	6	∞	6
2	∞	∞	-5	3	∞
3	∞	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	∞	6	∞	2	∞

Esimerkkiverkko Floydin-Warshallin algoritmille

$$D^{(0)} = \begin{pmatrix} 0 & 8 & 6 & \infty & 6 \\ \infty & 0 & -5 & 3 & \infty \\ \infty & \infty & 0 & \infty & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 6 & \infty & 2 & 0 \end{pmatrix} \qquad P^{(0)} = \begin{pmatrix} \text{Nil} & 1 & 1 & \text{Nil} & 1 \\ \text{Nil} & \text{Nil} & 2 & 2 & \text{Nil} \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} \\ \text{Nil} & 5 & \text{Nil} & 5 & \text{Nil} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 8 & 6 & \infty & 6 \\ \infty & 0 & -5 & 3 & \infty \\ \infty & \infty & 0 & \infty & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 6 & \infty & 2 & 0 \end{pmatrix} \qquad P^{(1)} = \begin{pmatrix} \text{Nil} & 1 & 1 & \text{Nil} & 1 \\ \text{Nil} & \text{Nil} & 2 & 2 & \text{Nil} \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} \\ \text{Nil} & 5 & \text{Nil} & 5 & \text{Nil} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 8 & 3 & 11 & 6 \\ \infty & 0 & -5 & 3 & \infty \\ \infty & \infty & 0 & \infty & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 6 & 1 & 2 & 0 \end{pmatrix} \qquad P^{(2)} = \begin{pmatrix} \text{Nil} & 1 & 2 & 2 & 1 \\ \text{Nil} & \text{Nil} & 2 & 2 & \text{Nil} \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} \\ \text{Nil} & 5 & 2 & 5 & \text{Nil} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 8 & 3 & 11 & 5 \\ \infty & 0 & -5 & 3 & -3 \\ \infty & \infty & 0 & \infty & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 6 & 1 & 2 & 0 \end{pmatrix} \qquad P^{(3)} = \begin{pmatrix} \text{Nil} & 1 & 2 & 2 & 3 \\ \text{Nil} & \text{Nil} & 2 & 2 & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} \\ \text{Nil} & 5 & 2 & 5 & \text{Nil} \end{pmatrix}$$

Floyd-Warshall: simulointi alkaa

$$D^{(3)} = \begin{pmatrix} 0 & 8 & 3 & 11 & 5 \\ \infty & 0 & -5 & 3 & -3 \\ \infty & \infty & 0 & \infty & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 6 & 1 & 2 & 0 \end{pmatrix} \qquad P^{(3)} = \begin{pmatrix} \text{Nil} & 1 & 2 & 2 & 3 \\ \text{Nil} & \text{Nil} & 2 & 2 & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} \\ \text{Nil} & 5 & 2 & 5 & \text{Nil} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 8 & 3 & 11 & 5 \\ \infty & 0 & -5 & 3 & -3 \\ \infty & \infty & 0 & \infty & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 6 & 1 & 2 & 0 \end{pmatrix} \qquad P^{(4)} = \begin{pmatrix} \text{Nil} & 1 & 2 & 2 & 3 \\ \text{Nil} & \text{Nil} & 2 & 2 & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} \\ \text{Nil} & 5 & 2 & 5 & \text{Nil} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 8 & 3 & 7 & 5 \\ \infty & 0 & -5 & -1 & -3 \\ \infty & 8 & 0 & 4 & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 6 & 1 & 2 & 0 \end{pmatrix} \qquad P^{(5)} = \begin{pmatrix} \text{Nil} & 1 & 2 & 5 & 3 \\ \text{Nil} & \text{Nil} & 2 & 5 & 3 \\ \text{Nil} & 5 & \text{Nil} & 5 & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} & \text{Nil} \\ \text{Nil} & 5 & 2 & 5 & \text{Nil} \end{pmatrix}$$

Floyd-Warshall: simulointi päättyy

Transitiivinen sulkeuma

Suunnatun verkon $G = (V, E)$ transitiivinen sulkeuma on verkko $G^* = (V, E^*)$, missä

$(u, v) \in E^*$ jos ja vain jos verkossa G on polku $u \leadsto v$.

Esimerkki 7.8: Jos G on vahvasti yhtenäinen, niin $E^* = V \times V$.

Yleisemmin jos u ja v kuuluvat samaan vahvasti yhtenäiseen komponenttiin, niin kaikilla $w \in V$ pätee

$(u, w) \in E^*$ jos ja vain jos $(v, w) \in E^*$
$(w, u) \in E^*$ jos ja vain jos $(w, v) \in E^*$.

Tämän perusteella verkon transitiivinen sulkeuma voidaan laskea

- muodostamalla ensin komponenttiverkko G^{SCC} ja
- laskemalla sitten komponenttiverkon transitiivinen sulkeuma.

Tämä säästää laskentaa, jos G^{SCC} sisältää paljon vähemmän solmuja kuin G. $\square$

Esimerkki 7.9: Olkoon $G = (V, E)$ suunnattu syklitön verkko. Nyt $(u, v) \in E^*$, jos ja vain jos kaikissa verkon G topologisissa järjestyksissä solmu u tulee ennen solmua v:

Jos verkossa G on polku $u \to s_1 \to \ldots \to s_k \to v$, niin kaikissa topologisissa järjestyksissä nämä solmut ovat määritelmän mukaan järjestyksessä $u, s_1, \ldots, s_k, v$.

Jos polkua $u \leadsto v$ ei ole, voidaan muodostaa seuraavanlainen topologinen järjestys:

1. kaikki ne w, joilla on olemassa polku $w \leadsto v$, viimeisenä solmu v

2. kaikki ne w, joilla on olemassa polku $u \leadsto w$, ensimmäisenä solmu u

Mistään jälkimmäisen luokan solmusta w_2 ei ole polkua ensimmäisen luokan solmuun w_1, koska muuten olisi $u \leadsto w_2 \leadsto w_1 \leadsto v$. $\square$

Transitiivinen sulkeuma voidaan laskea ajassa $\Theta(|V|^3)$ soveltamalla Floydin-Warshallin algoritmia:

1. Asetetaan kaikille kaarille $(u, v) \in E$ jokin äärellinen paino, esim. $w(u, v) = 1$.

2. Suoritetaan Floydin-Warshallin algoritmi.

3. Nyt $(u, v) \in E^*$, jos ja vain jos $D[u, v] < \infty$.

Algoritmia voidaan hieman virtaviivaistaa:

- Unohdetaan P, koska emme ole kiinnostuneita itse poluista.
- Pidetään kirjaa ainoastaan siitä, onko $D[i, j]$ äärellinen vai ääretön.

Saadaan seuraava algoritmi:

```
Transitive-Closure(A)
    for i ← 1 to n
        do
           for j ← 1 to n
               do if i = j or A[i,j] < ∞
                     then T[i,j] ← 1
                     else T[i,j] ← 0
    for k ← 1 to n
        do
           for i ← 1 to n
               do
                  for j ← 1 to n
                      do if T[i,k] = 1 and T[k,j] = 1
                            then T[i,j] ← 1
```

Ajassa $\Theta(n^3)$ saadaan taulukko T, missä $T[i,j] = 1$ jos $(i,j) \in E^*$, ja $T[i,j] = 0$ muuten.

Pienin virittävä puu (minimum spanning tree)

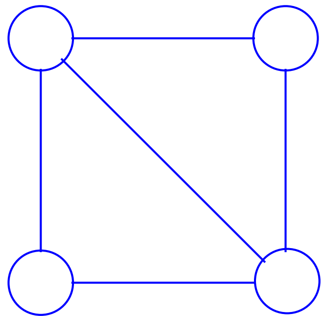
Jatkossa "puu" tarkoittaa vapaata puuta (ks. s. 113) eli suuntaamatonta verkkoa, joka on

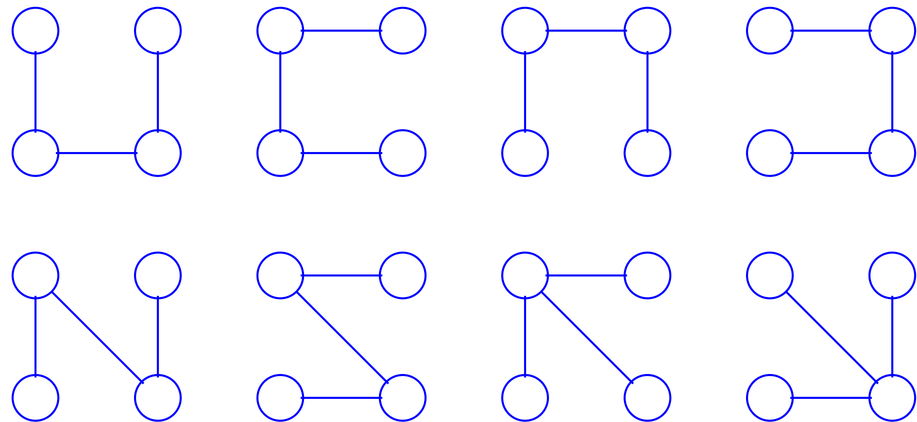
yhtenäinen: minkä tahansa kahden solmun välillä on polku
syklitön: minkä tahansa kahden solmun välillä on korkeintaan yksi yksinkertainen polku.

Yhtenäisen suuntaamattoman verkon $G = (V, E)$ virittävä puu on mikä tahansa puu (V, T), missä $T \subseteq E$. (Usein termiä "virittävä puu" käytetään myös pelkästä kaarijoukosta T.)

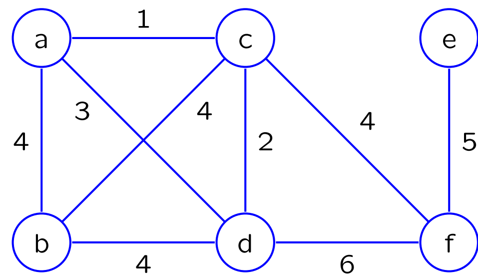
Siis virittävä puu saadaan valitsemalla joukosta E mahdollisimman vähän kaaria niin, että verkko pysyy yhtenäisenä. Virittävän puun kaarten lukumäärä on aina $|T| = |V| - 1$.

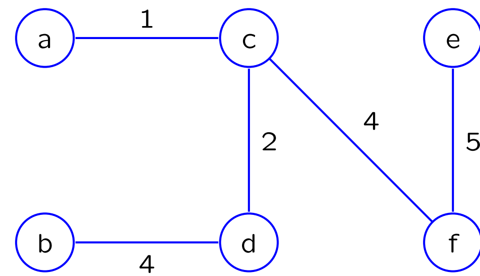
Pienin virittävä puu on virittävä puu, jonka paino $w(T) = \sum_{e \in T} w(e)$ on pienin mahdollinen.





Verkko ja sen kaikki virittävät puut





Painollinen verkko ja sen eräs pienin virittävä puu

Jatkossa esitettävät algoritmit pienimmän virittävän puun löytämiseksi noudattavat seuraavaa peruskaavaa:

Spanning-Tree(G)

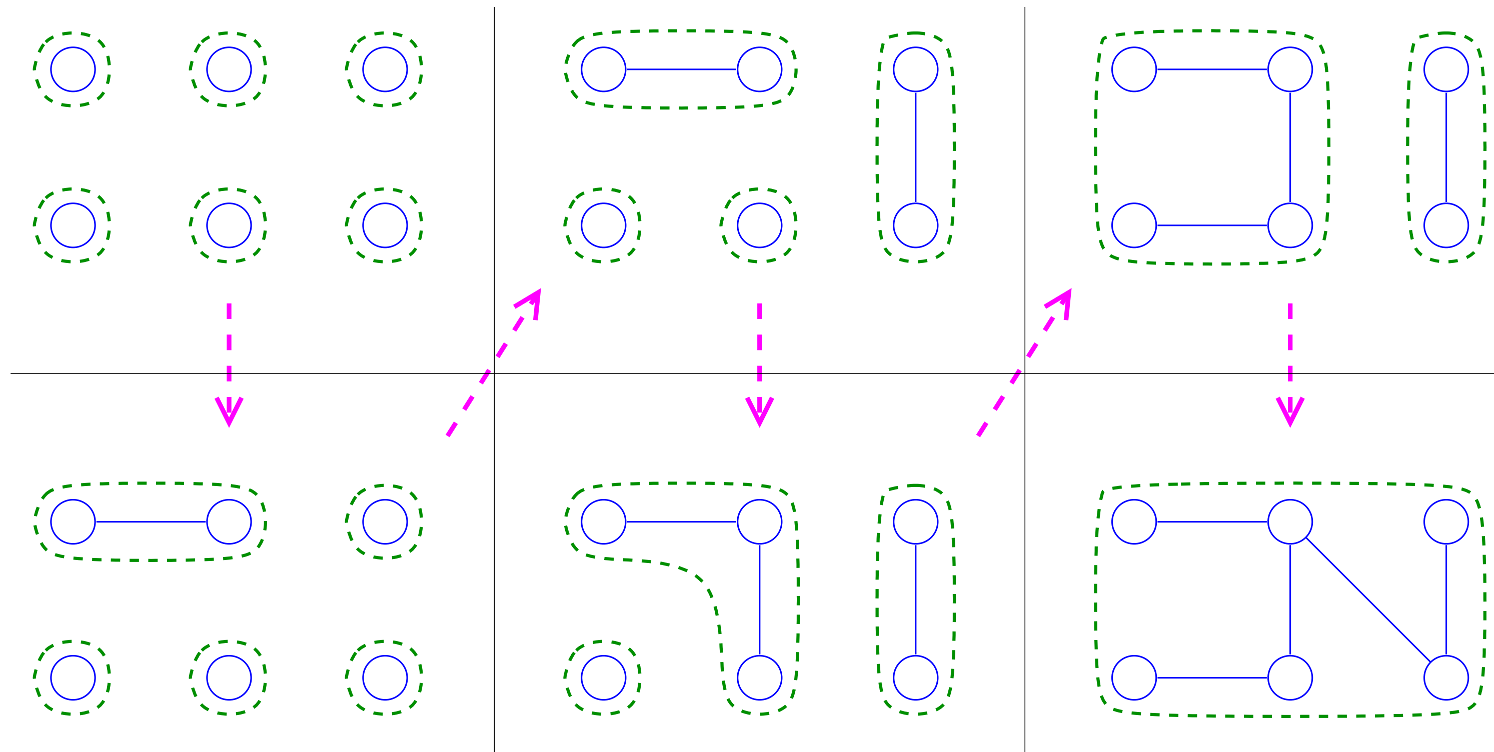
$F \leftarrow \emptyset$
while $|\,F\,| < |\,V\,| - 1$
  do valitse sellainen $(u, v) \in E - F$ että verkossa
    $(V, F \cup \{\,(u, v)\,\})$ ei ole kehää
    $F \leftarrow F \cup \{\,(u, v)\,\}$

Kysymyksiä:

1. Mistä tiedämme, että sopiva kaari (u, v) on aina olemassa?

2. Miten kehän olemassaolo testataan tehokkaasti?

3. Miten (u, v) pitää valita, että saadaan pienin virittävä puu?



Eräs tapa muodostaa virittävä puu. Välivaiheina syntyneiden virittävien metsien komponentit (selitetään pian) ympäröity katkoviivalla.

Virittävä puu liittyy läheisesti verkon yhtenäisyyden käsitteeseen.

Olkoon $G = (V, E)$ suuntaamaton verkko, joka ei välttämättä ole yhtenäinen. Merkitään $u \sim v$, jos verkossa G on polku $u \rightsquigarrow v$. Relaatio $\sim$ on selvästi ekvivalenssirelaatio:

1. $u \sim u$ kaikilla u
2. $u \sim v$, jos ja vain jos $v \sim u$
3. jos $u \sim v$ ja $v \sim w$, niin $u \sim w$.

Voimme siis muodostaa sen ekvivalenssiluokat eli solmujoukot $V_1, \ldots, V_l \subseteq V$, missä

- jokaisella $u \in V$ on olemassa tasan yksi i, jolla $u \in V_i$
- jos $u, v \in V_i$, niin $u \sim v$
- jos $u \in V_i$ ja $v \in V_j$, missä $i \neq j$, niin $u \not\sim v$.

Näitä ekvivalenssiluokkia kutsutaan verkon yhtenäisiksi komponenteiksi, ja ne voidaan helposti muodostaa esim. syvyyssuuntaisella läpikäynnillä (mitä emme kuitenkaan tarvitse jatkossa).

(Vertaa suunnatun verkon vahvasti yhtenäisiin komponentteihin, s. 435.)

Kun verkon $G = (V, E)$ yhtenäiset komponentit V_i on annettu, voidaan osittaa myös $E = E_1 \cup \ldots \cup E_l$, missä luokkaan E_i tulevat komponentin V_i solmujen väliset kaaret. Määritelmän mukaan eri komponentteihin kuuluvien solmujen välillä ei ole kaaria. Verkko siis jakautuu osaverkkoihin $G_i = (V_i, E_i)$, jotka ovat yhtenäisiä.

Jos $F \subseteq E$ on sellainen, että verkko (V, F) on syklitön, sanomme verkkoa (V, F) verkon (V, E) virittäväksi metsäksi. Jos metsälle (V, F) muodostetaan yhtenäiset komponenttiverkot (V_i, F_i) ylläesitettyyn tapaan, kukin (V_i, F_i) on

- syklitön, koska $F_i \subset F$ ja (V, F) on syklitön
- yhtenäinen, koska kyseessä on yhtenäinen komponentti.

Siis kukin (V_i, F_i) on puu. Virittävä metsä on kokoelma puita

- jotka yhdessä kattavat koko verkon mutta
- joita ei ole yhdistetty toisiinsa.

Palataan nyt algoritmihahmotelmaan Spanning-Tree. Edellä esitettyä terminologiaa käyttäen se pitää yllä verkon (V, E) virittävää metsää (V, F).

Aluksi $F = \emptyset$ eli metsä käsittää $|V|$ yksisolmuista puuta.

Jokaisella askelella algoritmi lisää joukkoon F yhden kaaren (u, v), joka ei luo verkkoon (V, F) syklejä. Algoritmi siis pitää yllä invariantin, että (V, F) on virittävä metsä.

Lopuksi $|F| = |V| - 1$ eli metsässä on vain yksi $|V|$-solmuinen puu.

Väitämme, että Spanning-Tree ei voi joutua "umpikujaan", eli jos $|F| < |V| - 1$, on aina mahdollista valita sellainen $(u, v) \in E - F$, että $(V, F \cup \{(u, v)\})$ on syklitön.

Olkoon $|F| < |V| - 1$ ja (V, F) syklitön. Siis virittävässä metsässä (V, F) on ainakin kaksi puuta (V_1, F_1) ja (V_2, F_2). Olkoon $p \in V_1$ ja $q \in V_2$. Koska G on yhtenäinen, siinä on polku $p \rightsquigarrow q$. Olkoon (u, v) tämän polun ensimmäinen kaari, jolla $u \in V_1$ mutta $v \notin V_1$. Merkitään $F' = F \cup \{(u, v)\}$. Väitämme, että (V, F') on syklitön.

Koska (V, F) on syklitön, niin mikä tahansa verkon (V, F') sykli kulkisi kaaren (u, v) kautta. Sykli voitaisiin siis kirjoittaa muotoon $(u, v, w_1, \ldots, w_t, u)$, missä kaaret (v, w_i), (w_t, u) ja (w_i, w_{i+1}) kaikilla $1 \leq i \leq t - 1$ kuuluvat joukkoon F. Siis $(v, w_1, \ldots, w_t, u)$ olisi polku $v \rightsquigarrow u$ verkossa (V, F). Tämä olisi vastoin oletusta, että u ja v ovat eri puissa. Siis verkossa (V, F') ei ole syklejä.

Olemme todenneet, että sovelias lisättävä kaari (u, v) on aina olemassa, kunnes virittävässä metsässä on vain yksi komponentti. Siis algoritmi tuottaa virittävän puun (V, F).

Tarkastellaan seuraavaksi tehokasta testausmenetelmää algoritmin Spanning-Tree ehdolle

verkko $(V, F \cup \{ (u, v) \})$ on syklitön,

missä siis (V, F) voidaan olettaa syklittömäksi.

Edellä esitetyn perusteella jos u ja v kuuluvat metsässä (V, F) eri puihin, niin $(V, F \cup \{ (u, v) \})$ on syklitön.

Toisaalta jos solmut u ja v kuuluvat samaan puuhun ja $u \neq v$, niiden välillä on polku $(u, w_1, \ldots, w_t, v)$, jonka kaaret kuuluvat joukkoon F. Siis verkossa $(V, F \cup \{ (u, v) \})$ on sykli $(u, w_1, \ldots, w_t, v, u)$.

Olemme todenneet, että ylläoleva ehto voidaan kirjoittaa muotoon

solmut u ja v ovat metsässä (V, F) eri puissa.

Tämän voi testata esim. syvyyssuuntaisella läpikäynnillä ajassa $O(|V| + |E|)$. Näitä testejä joudutaan kuitenkin tekemään hyvin paljon. Suoritusta voi oleellisesti nopeuttaa pitämällä yllä sopivaa tietorakennetta.

Erillisten joukkojen yhdisteet

Esittelemme tietorakenteen, josta on hyötyä yleisemminkin kuin virittävän metsän komponenttien ylläpidossa.

On annettu perusjoukko X ja kokoelma $\mathcal{S}$ sen erillisiä osajoukkoja. Siis $\mathcal{S} = \{ S_1, \ldots, S_n \}$, missä $S_i \subseteq X$ ja $S_i \cap S_j = \emptyset$, jos $i \neq j$. Haluamme tukea seuraavanlaisia operaatioita:

Union(S_i, S_j): Kokoelmaan $\mathcal{S}$ liitetään uusi joukko, joka saa alkioikseen joukkojen S_i ja S_j alkiot. Vanhat joukot S_i ja S_j poistuvat kokoelmasta.

Find(x): palauttaa sen joukon S_i tunnuksen, johon x kuuluu.

Täsmennämme tätä niin, että joukon S_i tunnuksena on jokin sen alkio $x \in S_i$. Joukon tunnus voi ajan myötä vaihtua, mutta annetulla ajanhetkellä se on yksikäsitteinen.

Saamme seuraavat operaatiot:

Make-Set(x): luo uuden joukon $\{x\}$, jonka tunnukseksi tulee x.
Union(x, y): yhdistää joukot, joiden tunnukset ovat x ja y.
Find(x): palauttaa sen joukon tunnuksen, johon x kuuluu.

Siis erityisesti operaation Union(Find(x), Find(y)) jälkeen kutsujen Find(x) ja Find(y) pitää palauttaa sama alkio. Koska mikään operaatio ei pilko joukkoja, tämä pätee jatkossa aina.

Esimerkki 7.10:

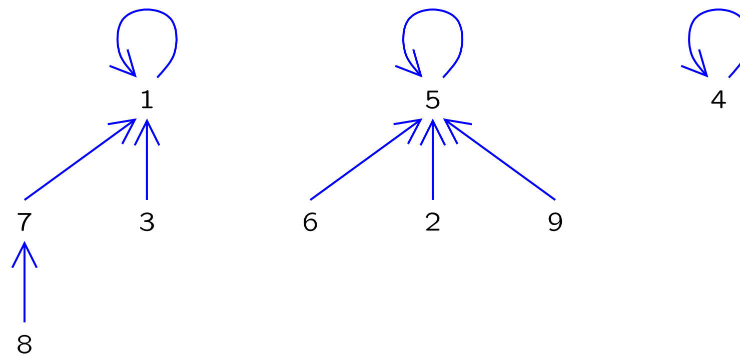
operaatio	joukot
Make-Set(1), . . . Make-Set(9)	$\{\underline{1}\}\ \{\underline{2}\}\ \{\underline{3}\}\ \{\underline{4}\}\ \{\underline{5}\}\ \{\underline{6}\}\ \{\underline{7}\}\ \{\underline{8}\}\ \{\underline{9}\}$
Union(1, 3)	$\{\underline{1},3\}\ \{\underline{2}\}\ \{\underline{4}\}\ \{\underline{5}\}\ \{\underline{6}\}\ \{\underline{7}\}\ \{\underline{8}\}\ \{\underline{9}\}$
Union(2, 5)	$\{\underline{1},3\}\ \{2,\underline{5}\}\ \{\underline{4}\}\ \{\underline{6}\}\ \{\underline{7}\}\ \{\underline{8}\}\ \{\underline{9}\}$
Union(6, 5)	$\{\underline{1},3\}\ \{2,\underline{5},6\}\ \{\underline{4}\}\ \{\underline{7}\}\ \{\underline{8}\}\ \{\underline{9}\}$
Find(7)	palauttaa 7
Find(6)	palauttaa 5
Union(7, 8)	$\{\underline{1},3\}\ \{2,\underline{5},6\}\ \{\underline{4}\}\ \{\underline{7},8\}\ \{\underline{9}\}$
Union(5, 9)	$\{\underline{1},3\}\ \{2,\underline{5},6,9\}\ \{\underline{4}\}\ \{\underline{7},8\}$
Union(7, 1)	$\{\underline{1},3,7,8\}\ \{2,\underline{5},6,9\}\ \{\underline{4}\}$

Joukkojen edustajat alleviivattu (mutta olisi voitu valita toisinkin). □

Toteutamme erilliset joukot puurakenteena:

- Jokainen joukko esitetään juurellisena puuna, jonka solmuissa on joukon alkiot.
- Jokaiseen alkiooon x liittyy osoitin $p[x]$:
  - Jos x on puun juuri (mukaanluettuna yksisolmuinen puu), niin $p[x] = x$.
  - Muuten $p[x]$ osoittaa solmun x vanhempaa puussa.
- Puun juuressa on joukon tunnusalkio.

Esimerkki 7.11: Joukkojen $\{\underline{1}, 3, 7, 8\}$, $\{2, \underline{5}, 6, 9\}$ ja $\{\underline{4}\}$ puuesitys.



□

Jos tehokkuudesta ei välitetä, operaatiot on suoraviivaista toteuttaa:

Make-Set-0.1(x)

$p[x] \leftarrow x$

Union-0.1(x, y)

$p[y] \leftarrow x$

Find-0.1(x)

```
while p[x] ≠ x
      do x ← p[x]
return x
```

Operaatiot Make-Set ja Union sujuvat selvästi vakioajassa, mutta Find vie hakupolun pituuden suhteen lineaarisen ajan. Kahdella helpolla optimoinnilla polut lyhenevät oleellisesti:

tasapainotus Union-operaatiossa
poluntiivistys Find-operaatiossa.

Tasapainottavan Union-toteutuksen idea on liittää jokaiseen solmuun x luokitus $rank[x]$, joka kertoo pisimmän solmuun x johtavan polun pituuden (kun mahdollisia silmukoita $x \to x$ ei lasketa).

Siis suurin luokitus $\max_x rank[x]$ on yläraja Find-operaation vaativuudelle.

Alustus tulee muotoon

Make-Set(x)

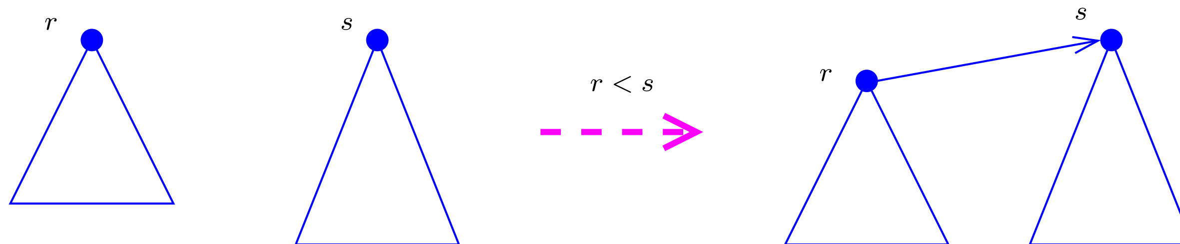
$p[x] \leftarrow x$
$rank[x] \leftarrow 0$

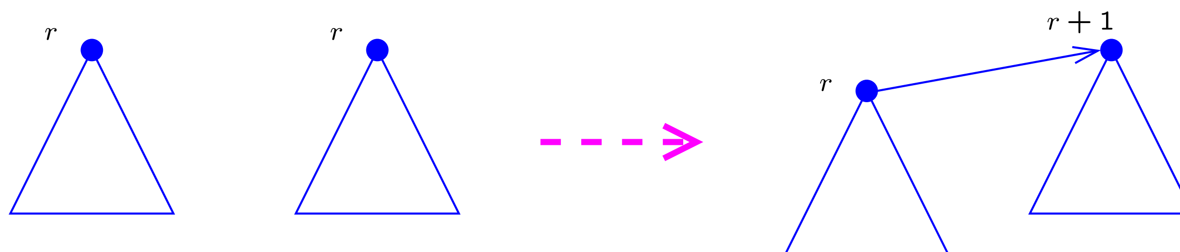
Jotta Union-operaatiot sujuisivat nopeasti, halutaan pitää $rank$-arvot mahdollisimman pienenä. Kun sijoitetaan $p[y] \leftarrow x$, niin luokitusta $rank[x]$ pitää päivittää seuraavasti:

$$rank[x] \leftarrow \max \{ rank[x], rank[y] + 1 \} .$$

Jos $rank[y] < rank[x]$, niin luokitukset eivät kasva! Siis kannattaa sijoittaa pienempi luokitus suuremman luokituksen alipuuksi:

```
Union(x, y)
    if rank[y] < rank[x]
        then p[y] ← x
    elseif rank[x] < rank[y]
        then p[x] ← y
    else
            p[y] ← x
            rank[x] ← rank[x] + 1
```



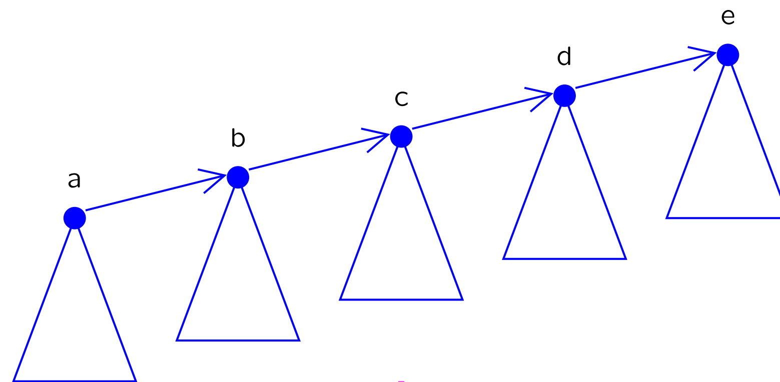


Union ja tasapainotus $rank$-arvoilla.

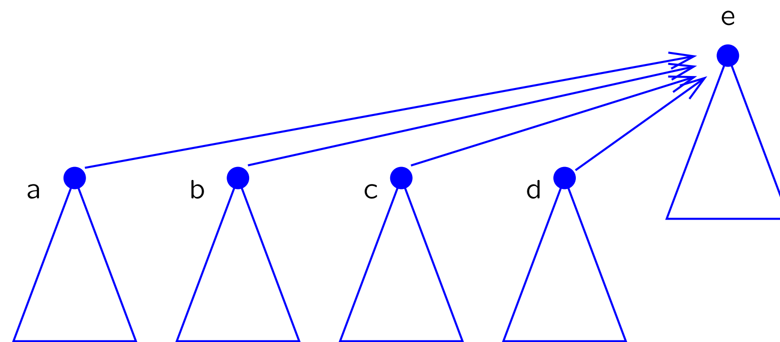
Induktiolla nähdään helposti, että jos $rank[x] = k$, niin solmulla x on ainakin 2^k jälkeläistä puussa.

Siis jos alkioita kaikkiaan on n, niin kaikki polunpituudet ovat $O(\log n)$. Mikä tahansa m Union-Find-operaation jono voidaan suorittaa ajassa $O(m \log n)$.

Tietorakennetta voidaan vielä tehostaa suorittamalla Find-operaation yhteydessä poluntiivistys: Kun on löydetty polku $x \to p[x] \to p[p[x]] \to \ldots \to y$, missä y on puun juuri, niin oikaistaan kaikki osoittimet $p[x]$, $p[p[x]]$ jne. osoittamaan suoraan juureen y. Tällöin seuraavat Find-operaatiot nopeutuvat.



Find(a)



Find ja poluntiivistys

Saadaan seuraava algoritmi:

```
Find(x)
    z ← x
    while p[z] ≠ z
        do z ← p[z]
    y ← z
    z ← x
    while p[z] ≠ z
        do z, p[z] ← p[z], y
    return y
```

Huomaa, että $rank$-arvoja ei Find-operaatiossa päivitetä, joten ne menettävät tarkan vastaavuutensa polunpituuksiin.

Tasapainotusta ja poluntiivistystä käytettäessä voidaan osoittaa, että m Union-Find-operaatiota n alkion perusjoukossa vie vain ajan $O(m\alpha(n))$, missä α on eräs *erittäin* hitaasti kasvava funktio: karkeasti arvioiden $\alpha(n) \leq 4$ kun $n \leq 10^{80}$. (Todistus sivuutetaan, ks. esim. Cormen et al. luku 21.)

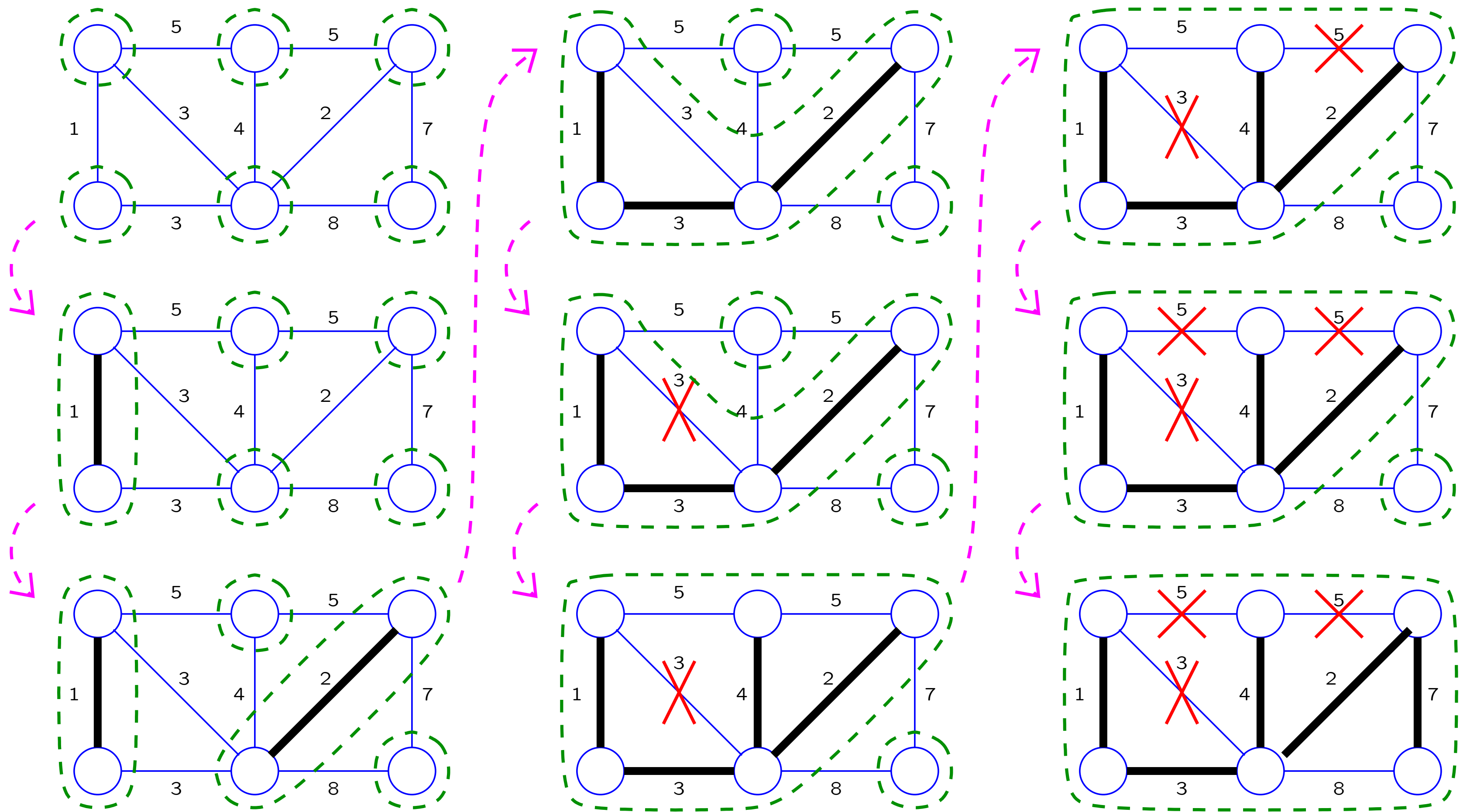
Sama asymptoottinen aikavaativuus saadaan muillakin samanhenkisillä tasapainotus- ja polunoikaisuheuristiikoilla.

Kruskalin algoritmi

Meillä on nyt tarvittavat komponentit pienimmän virittävän puun tehokkaaseen löytämiseen. Kruskalin algoritmissa virittävän puun perusalgoritmiin lisätään yksinkertainen ahne heuristiikka: kokeillaan aina painoltaan pienintä jäljellä olevaa kaarta.

```
Kruskal(G)
    F ← ∅
    for kaikille v ∈ V
        do Make-Set(v)
    Järjestä E kaaren painon mukaan kasvavasti.
    for kaikille (u, v) ∈ E painon mukaan kasvavassa järjestyksessä
        do if Find(u) ≠ Find(v)
            then F ← F ∪ { (u, v) }
                Union(Find(u), Find(v))
    return (V, F)
```

Jos usealla kaarella on sama paino, ne voidaan käsitellä missä tahansa järjestyksessä.



Esimerkki Kruskalin algoritmin toiminnasta.

Tarkastellaan seuraavaksi Kruskalin algoritmin aikavaativuutta:

- Make-Set-operaatiot $O(|V|)$
- muut Union- ja Find-operaatiot yhteensä $O(|E|\alpha(|V|))$
- kaarten järjestäminen $O(|E| \log |E|)$
- muu kirjanpito yms. $O(|E|)$.

Siis aikavaativuutta dominoi kaarten järjestämisen $O(|E| \log |E|)$.

Usein virittävä puu tulee valmiiksi, ennen kuin kaikki kaaret on käsitelty. Tällaisissa tapauksissa algoritmia voidaan jonkin verran tehostaa käyttämällä kekoa sen sijaan, että heti järjestetään kaikki kaaret:

```
Kruskal-with-Heap(G)
    F ← ∅
    for kaikille v ∈ V
          do Make-Set(v)
    H ← Build-Heap(E)
    while |F| < |V| − 1
          do (u, v) ← Heap-Delete-Min(H)
              if Find(u) ≠ Find(v)
                 then F ← F ∪ { (u, v) }
                        Union(Find(u), Find(v))
    return (V, F)
```

Asymptoottinen pahimman tapauksen aikavaativuus on kuitenkin sama kuin ennenkin.

Kruskalin algoritmi oikeellisuus ei ole aivan ilmeinen. Todetaan ensin, että se palauttaa jonkin virittävän puun (V, F):

Kaaren (u, v) sisällyttäminen joukkoon F aiheuttaa yhdistämisen Union(Find(u), Find(v)). Siis ehto Find(u) $\neq$ Find(v) on tosi vain, jos u ja v ovat verkon (V, F) eri komponenteissa. Aiemmin esitetyn perusteella (V, F) pysyy syklittömänä. Siis algoritmin palauttama (V, F) on virittävä metsä.

Oletuksen mukaan G on yhtenäinen. Jos (V, F) algoritmi suorituksen päättyessä ei olisi yhtenäinen, niin verkossa G olisi jokin kahta metsän (V, F) puuta yhdistävä kaari e, jota ei ole otettu mukaan joukkoon F. Koska puut eivät suorituksen aikana lisäänny, e yhdisti kahta eri puuta jo silloin, kun e oli kokeiltavana, jolloin se olisi pitänyt ottaa mukaan joukkoon F; ristiriita.

Siis suorituksen päättyessä (V, F) on virittävä puu.

Seuraavan tuloksen perusteella voidaan osoittaa, että algoritmin palauttama virittävä puu on pienin:

Lause 7.12: Olkoon $U \subset V$, $\overline{U} = V - U$ ja $(u, v) \in E$ painoltaan pienin sellainen kaari, jonka toinen päätepiste on joukossa U ja toinen joukossa $\overline{U}$.

Olkoon lisäksi F sellainen joukko kaaria, että

- jos $(r, s) \in F$, niin r ja s ovat joko kumpikin joukossa U tai kumpikin joukossa $\overline{U}$
- $F \subseteq T$ jollekin pienimmälle virittävälle puulle (V, T).

Nyt $F \cup \{ (u, v) \} \subseteq T'$ jollekin pienimmälle virittävälle puulle (V, T').

Lauseen 7.12 avulla nähdään, että algoritmi pitää yllä invarianttia

$$F \subseteq T \text{ jollakin pienimmällä virittävällä puulla } (V, T).$$

Aluksi $F = \emptyset$, joten invariantti pätee triviaalisti.

Kun joukkoon F ollaan lisäämässä kaarta (u, v), valitaan joukkoon U ne solmut, jotka ovat (ennen lisäystä) solmun u kanssa samassa puussa. Joukko U toteuttaa lauseen ehdot, joten lauseen nojalla $F \cup \{ (u, v) \}$ toteuttaa invariantin.

Lopuksi (V, F) on virittävä puu, joten invariantin nojalla se on pienin sellainen, kuten haluttiin.

Lauseen 7.12 todistus: Olkoon (V, T) pienin virittävä puu, jolla $F \subseteq T$. Jos $(u, v) \in T$, väite on selvä.

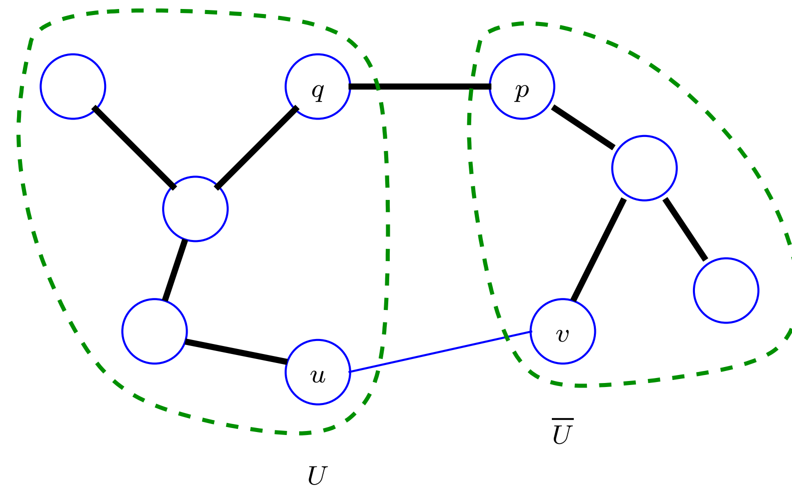
Oletetaan $(u, v) \notin T$. Verkossa $(V, T \cup \{ (u, v) \})$ on kehä, joka sisältää kaaren (u, v) ja sen lisäksi ainakin yhden toisen kaaren (p, q), jonka toinen päätepiste on joukossa U ja toinen joukossa $\overline{U}$. Erityisesti $(p, q) \notin F$. Valitaan $T' = T \cup \{ (u, v) \} - \{ (p, q) \}$.

Kaaren (p, q) poistaminen jakaa puun (V, T) kahteen komponenttiin, jotka (u, v) taas yhdistää. Siis (V, T') on virittävä puu.

Oletuksen mukaan $w(u, v) \leq w(p, q)$, joten

$$\sum_{e \in T'} w(e) = \sum_{e \in T} w(e) - w(p, q) + w(u, v) \leq \sum_{e \in T} w(e).$$

Koska (V, T) on pienin virittävä puu, niin myös (V, T') on. $\square$



q
p
u
v
U
$\overline{U}$

Primin algoritmi

Perusidea on samantapainen kuin Kruskalin algoritmissa. Nyt kuitenkin pidetään yllä vain yhtä puuta ja lisätään siihen kaaria yksi kerrallaan.

Toisin sanoen algoritmi pitää yllä puuta (S, T), missä $S \subseteq V$ ja $T \subseteq E$. Kuten Kruskalin algoritmissa, nytkin pidetään yllä invarianttia

> jollakin pienimmällä virittävällä puulla (V, T') pätee $T \subseteq T'$.

Puuhun lisätään seuraavaksi aina se solmu $v \in V - S$, jonka etäisyys

$$key[v] = \min_{u \in S} w(u, v)$$

on pienin. Solmut pidetään tämän avaimen mukaisessa prioriteettijonossa.

Kun v liitetään joukkoon S, samalla joukkoon T liitetään (u, v), missä $u \in S$ on sellainen, että $w(u, v) = key[v]$.

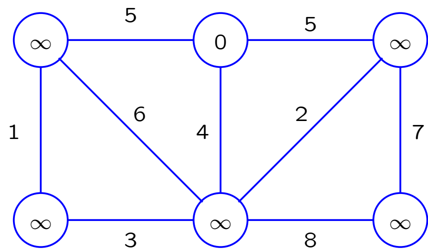
Algoritmi ei pidä eksplisiittisesti kirjaa joukoista S ja T.

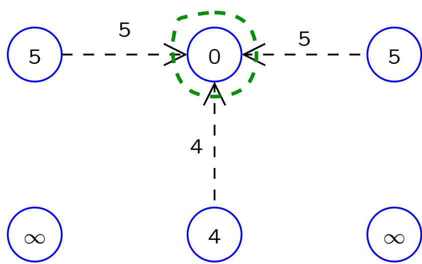
Joukon S solmuja ovat ne, jotka eivät enää ole prioriteettijonossa Q.

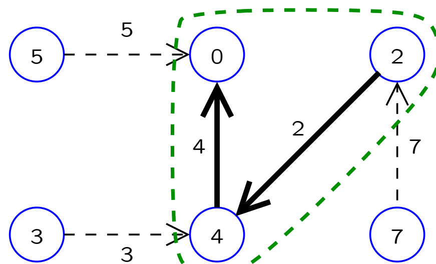
Jokaiseen solmuun $v \in V - S$ liitetään osoitin $p[v]$, joka kertoo sitä lähinnä olevan joukon S solmun. Kun v liitetään joukkoon S, arvo $p[v]$ jäädytetään. Lopulta puuhun T tulee kaikki kaaret $(v, p[v])$, missä $v \neq r$.

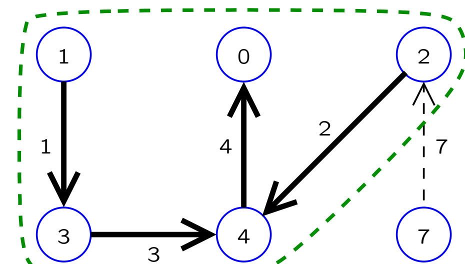
Annamme algoritmille parametrina solmun r, joka toimii puun "siemenenä".

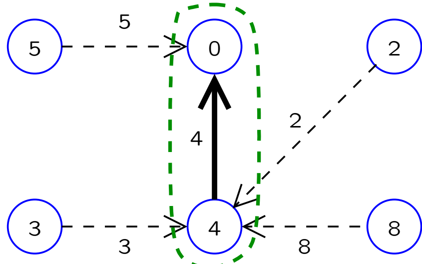
```
Prim(G, w, r)
    for jokaiselle v ∈ V do key[v] ← ∞
    key[r] ← 0
    for jokaiselle v ∈ V do Insert(Q, r, key[r])
    while Q ≠ ∅
        do u ← Delete-Min(Q)
            for jokaiselle v ∈ Adj[u]
                do if v ∈ Q and key[v] > w(u, v)
                    then key[v] ← w(u, v)
                         Decrease-Key(Q, v, w(u, v))
                         p[v] ← u
```

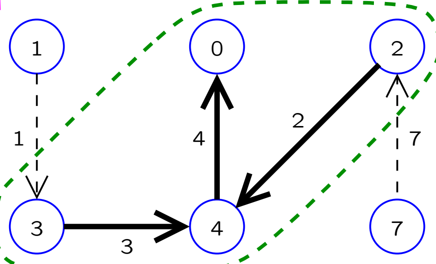


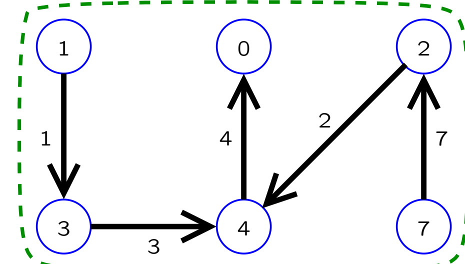












Esimerkki Primin algoritmin toiminnasta.

Algoritmista nähdään suoraan, että arvot $key[v]$ ja $p[v]$ ovat kuten edellä sanottiin:

$$\begin{aligned} key[v] &= \min_{u \in S} w(u,v) \qquad \text{kun } v \notin S \\ w(p[v],v) &= key[v], \end{aligned}$$

missä $S = V - Q$.

Lisäksi kun määritellään $T = \{\, (p[v],v) \mid v \in S - \{\, r \,\}\,\}$, niin verkko (S,T) on syklitön, sillä joukkoon T ei koskaan lisätä kaaria joukossa S jo olevien solmujen välille.

Algoritmin oikeellisuuden ei-trivaali osuus on osoittaa invariantti

> jollekin pienimmälle virittävälle puulle (V,T') pätee $T \subset T'$.

Tähän käytämme jälleen lausetta 7.12 (s. 514).

Olkoon S ja T kuten edellä. Tarkastellaan algoritmia, kun ollaan lisäämässä $S \leftarrow S \cup \{ u \}$. Tehdään induktio-oletus, että $T \subseteq T'$ jollain pienimmällä virittävällä puulla (V, T').

Kaikilla $v \in V - S$ kaari $(p[v], v)$ on painoltaan pienin kaari joukosta S solmuun v, ja kaaren paino on $key[v]$. Algoritmi valitseen sen u, jolla avain $key[u]$ on pienin. Siis $(p[u], v)$ on painoltaan pienin kaari joukosta S joukkoon $V - S$. Koska joukossa T ei ennestään ole kaaria joukosta S joukkoon $V - S$, lauseen 7.12 ehdot pätevät. Siis $T \cup \{ (p[u], u) \} \subseteq T''$ jollekin pienimmälle virittävälle puulle (V, T'').

Siis invariantti pätee. Lopuksi $S = V$ ja (S, T) on koko verkon pienin virittävä puu.

Primin algoritmin ensimmäinen for-silmukka vie ajan $O(|V|)$.

Jos prioriteettijono toteutetaan kekona, sen alustaminen voidaan suorittaa ajassa $O(|V|)$ käyttämällä algoritmia Build-Heap.

Delete-Min-operaatioita suoritetaan $|V|$ ja Decrease-Key-operaatioita korkeintaan $|E|$ kappaletta. Kekototeutuksella kukin näistä menee ajassa $O(\log|V|)$. Siis while-silmukan ja samalla koko algoritmin aikavaativuudeksi tulee $O((|V| + |E|) \log |V|)$ eli sama kuin Kruskalin algoritmilla.

Kuten Dijkstran algoritmissa, tässäkin asymptoottista aikavaativuutta voidaan parantaa toteuttamalla prioriteettijono Fibonacci-kekona. Tällöin kutakin Decrease-Key-operaatiota kohden tarvitaan vain vakioaika. Kokonaisaikavaativuudeksi tulee $O(|E| + |V| \log |V|)$.

Jos verkko on tiheä eli $|E| = \Theta(|V|^2)$, Primin algoritmi Fibonacci-keolla toteutettuna saa aikavaativuuden $O(|E| + |V| \log |V|) = O(|V|^2)$. Sama aikavaativuus saadaan helpomminkin:

```
Prim-Dense(G, w, r)
    key[r] ← 0
    S ← { r }
    for jokaiselle v ∈ V − { r }
        do key[v] ← w(r, v)
           p[v] ← r
    while S ≠ V
        do valitse u, jolla key[u] on pienin
           S ← S ∪ { u }
           for kaikille v ∈ V
               do if v ∉ S and w(u, v) < key[v]
                   then key[v] ← w(u, v)
                        p[v] ← u
```

Siis Primin algoritmi aikavaativuus tiheille verkoille on $O(|V|^2)$ (myös ilman Fibonacci-kekoa, jonka vakiokertoimet ovat suuret).

Tiheillä verkoilla Primin algoritmin aikavaativuus $O(|V|^2)$ on parempi kuin Kruskalin $O(|E|\log|E|) = O(|V|^2\log|V|)$.

Harvoilla verkoilla Kruskalin pahimman tapauksen aikavaativuus $O(|E|\log|V|)$ on sama kuin Primin (jos unohdetaan Fibonacci-keko).

Monissa tilanteissa Kruskal on kuitenkin parempi, kuin pahimman tapauksen analyysi antaa ymmärtää:

- jos pienimpään virittävään puuhun tulevat kaaret ovat kaikki hyvin kevyitä, algoritmi pysähtyy, ennen kuin kaikkia kaaria on tarvinnut käydä läpi
- jos painot ovat pieniä kokonaislukuja, järjestäminen sujuu nopeammin kuin $\Omega(|E|\log|E|)$.

Siis algoritmien keskinäinen paremmuus on tapauskohtaista.

Esimerkki 7.13: Halutaan osittaa verkon $G = (V, E)$ solmut kahteen luokkaan V_1 ja V_2 siten, että luokkien välinen pienin etäisyys

$$d(V_1, V_2) = \min \{ w(u, v) \mid u \in V_1, v \in V_2 \}$$

on mahdollisimman suuri. (Muistetaan, että osituksessa $V_1 \cap V_2 = \emptyset$ ja $V_1 \cup V_2 = V$.)

Intuitiivinen tulkinta on, että

- $w(u, v)$ on jokin mitta alkioiden u ja v erilaisuudelle
- alkiot halutaan jakaa kahteen mahdollisimman selvästi toisistaan erottuvaan luokkaan.

Annettu ongelma on erikoistapaus ryvästämisestä, joka on tärkeä menetelmä data-analyysissä. Halutaan osittaa perusjoukko X rypäisiin $X_1, \ldots, X_k$ siten, että

- eri rypääseen kuuluvilla u ja v erilaisuus $d(u, v)$ on mahdollisimman suuri ja
- samaan rypääseen kuuluvilla u ja v erilaisuus $d(u, v)$ on mahdollisimman pieni.

Tavoite voidaan täsmentää eri tavoin, mutta yleensä ongelma on laskennallisesti hankala. Tässä tarkasteltu erikoistapaus ratkeaa kuitenkin helposti muodostamalla pienin virittävä puu.

Väitämme, että ongelma ratkeaa seuraavalla algoritmilla:

1. Muodosta verkolle pienin virittävä puu (V, T).

2. Olkoon e joukon T painoltaan suurin kaari.

3. Valitse luokiksi V_1 ja V_2 metsän $(V, T - \{ e \})$ yhtenäiset komponentit.

Tulos on sama, kuin jos ajettaisiin Kruskalin algoritmia, mutta lopetettaisiin juuri ennen viimeisen kaaren lisäämistä virittävään puuhun.

Todetaan ensin, että muodostettujen luokkien V_1 ja V_2 pienin etäisyys on

$$\min \{ w(u,v) \mid u \in V_1, v \in V_2 \} = w(e).$$

Valitaan mitkä tahansa $u \in V_1$ ja $v \in V_2$. Nyt $T' = (T - \{ e \}) \cup \{ (u,v) \}$ on virittävä puu, jonka paino on

$$w(T') = w(T) - w(e) + w(u,v).$$

Koska T on pienin virittävä puu, on oltava $w(u,v) \geq w(e)$.

Siis e on luokkia V_1 ja V_2 yhdistävistä kaarista painoltaan pienin.

Todetaan sitten, että millä tahansa solmujen osituksella (U_1, U_2) pätee

$$\min \{ w(u, v) \mid u \in U_1, v \in U_2 \} \leq w(e).$$

Tämä seuraa suoraan siitä, että ainakin yksi puun T kaari yhdistää joukkoja U_1 ja U_2, ja e on puun T kaarista painoltaan suurin.

Edellisistä kahdesta toteamuksesta seuraa, että algoritmin tuottama ositus (V_1, V_2) maksimoi lyhimmän etäisyyden

$$\min \{ w(u, v) \mid u \in V_1, v \in V_2 \} .$$

$\square$

8. Yhteenvetoa

Tietorakenteet ja abstraktit tietotyypit:

abstrakti tietotyyppi: mitä datalle halutaan tehdä
tietorakenne: miten se tehdään (talletusrakenne, operaatioden toteutus)

Tietorakenteet ja algoritmit:

- tietorakenteiden toteuttamisessa voidaan tarvita ei-triviaaleja algoritmeja (esim. AVL-puiden tasapainotus)
- algoritmien tehokkaassa toteuttamisessa voidaan tarvita ei-triviaaleja tietorakenteita (esim. Kruskalin algoritmi: keko ja Union-Find).

Millaisia asioita olemme oppineet

Toisin sanoen mitä kurssista olisi hyvä muistaa vielä ensi vuonna.

Perustietorakenteita ja -algoritmeja, jotka on hyvä osata jopa koodata melko sujuvasti:

- pino, jono, linkitetty lista, puu
- syvyyssuuntainen ja leveyssuuntainen läpikäynti

Yleisesti tarvittavia tietorakenteita ja algoritmeja, joita ei yleensä tarvitse (tai kannata) itse koodata, mutta keskeiset periaatteet pitää tuntea:

- hakemistorakenteet: puu vs. hajautus; keskusmuisti vs. levy
- järjestäminen: $O(n)$ vs. $O(n \log n)$ vs. $O(n^2)$; pikajärjestämisen vaikeat tapaukset.

Mallinnustekniikoita: etenkin puiden ja verkkojen käyttäminen ongelmanratkaisun mallina

- tekoäly: pelipuut, abstraktin puun tai verkon läpikäynti, graafiset mallit (verkkopohjaisia todennäköisyysmalleja)
- tietoliikenne: esim. reitittäminen Dijkstran algoritmilla
- ohjelmointi ja ohjelmointikielet: jäsennyspuu, verkkopohjaiset kaavioesitykset
- tietojenkäsittelyteoria: puut ja verkot abstraktien laskennan mallien esittämisessä.

Algoritmien suunnittelutekniikoita:

invariantit: perustekniikka, josta on iloa myös perusohjelmoinnissa turhien virheiden välttämiseksi

hajoita ja hallitse: keskeinen tekniikka edistyneemmässä algoritmisuunnittelussa; analyysi perustuu rekursioyhtälöihin; esim. lomitusjärjestäminen

ahne algoritmi: idea on usein luonteva, mutta oikeellisuus vaikea perustella; esim. Kruskal

taulukointi (dynaaminen ohjelmointi): monipuolinen tekniikka, itse asiassa varsin intuitiivinen kun siihen tottuu; esim. Floyd-Warshall.

Etenkin hajoita ja hallitse, ahneus ja taulukointi voivat olla vaikeita soveltaa. Niitä voi harjoitella lisää kurssilla *Algoritmien suunnittelu ja analyysi*. Tällä kurssilla päätavoite on ymmärtää esitettyjen esimerkkialgoritmien ideat ja osata soveltaa niitä hieman muuntaen.

Algoritmit, tietorakenteet ja muu maailma

Käytännössä algoritmit

- toteutetaan tietokonelaitteistolla ja
- liittyvät osaksi sovellusohjelmaa.

Tällä kurssilla on lähinnä tarkasteltu asymptoottista pahimman tapauksen aikavaativuutta ("O-notaatio"), joka kertoo yksinkertaisessa muodossa algoritmin skaalautuvuuden. Sovelluksessa tarvitaan muutakin:

- Kuinka isoja syötteet todella ovat? Helppoja vai vaikeita?
- Mikä on riittävä suorituskyky? Mikä oikeasti on järjestelmän pullonkaula?
- Jos algoritmia pitää tosissaan ruveta viilaamaan, miten se suhtautuu käytettävissä olevaan suoritusympäristöön (rinnakkaistuminen, välimuistin käyttö jne.)?
- Käytännön tekijät: ylläpidettävyys, virheestä toipuminen jne.

Kertaustehtäviä

Käydään kertauksena läpi kevään 2005 toisen kurssikokeen tehtävät.

Tehtävä 1: Erääseen sovellukseen tarvitaan seuraavanlainen tietorakenne R:

- Sen mahdolliset hakuavaimet k ovat 64-bittisiä etumerkittömiä kokonaislukuja (eli väliltä $0 \ldots 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$) ja niiden jakauma on tasainen.
- Hakuoperaatio lookup(R, k) vastaa, löytyykö avain k tällä hetkellä rakenteesta R vaiko ei.
- Lisäysoperaatio insert(R, k) lisää avaimen k rakenteeseen R, ellei se jo ole siinä.

Koska sovelluksessa täytyy säästää muistia, niin rakenteessa R saa olla yhtä aikaa korkeintaan $2^{20} = 1\,048\,576$ avainta. Jos rakenne R on jo täynnä, niin uusi avain k korvaa sen vanhan avaimen k', jonka edellisestä käsittelystä (eli lisäyksestä insert(R, k') tai hausta lookup(R, k')) on mahdollisimman kauan.

Sovelluksessa on tärkeintä, että onnistuvat haut (eli ne, joilla lookup(R, k) vastaa `true`) olisivat keskimäärin hyvin nopeita. Muut operaatiot eivät ole niin kriittisiä.

(a) Ehdota tälle rakenteelle R sopiva toteutus. Perustele valintasi.

(b) Anna kohdan (a) toteutuksessasi pseudokoodi rakenteen R hakuoperaatiolle lookup(R, k).

(c) Anna kohdan (a) toteutuksessasi pseudokoodi rakenteen R lisäysoperaatiolle insert(R, k).

Tehtävän 1 ratkaisuidea: Tärkeimmäksi kriteeriksi on annettu onnistuneen haun keskimääräinen aikavaativuus. Ensimmäisenä mieleen tulee hajautus. Koska tehtävässä lisäksi avaimet ovat kokonaislukuja ja jakautuneet tasaisesti, esim. jakolaskumenetelmä toimisi varsin hyvin.

Valitaan siis perusratkaisuksi hajautus jakolaskumenetelmällä. Koska joudutaan suorittamaan myös poistoja, yhteentörmäykset on kätevämpää hoitaa ketjuttamalla (eikä avoimella hajautuksella).

Valitaan talletusalueen kooksi p alkuluku läheltä lukua $(2^{18}+2^{19})/2 = (3/4)\cdot 2^{19} = (3/8)\cdot 2^{20}$. Tällöin se on kaukana kakkosen potensseista, ja täyttöasteeksi tulee korkeintaan $8/3$ eli varsin kohtuullinen. (Muutkin valinnat ovat mahdollisia.)

Mietitään nyt poistojen toteuttamista.

Koska poistettavaksi valitaan kauimmin joutilaana ollut avain, ensimmäisenä tulee mieleen tallentaa avaimet jonoon. Ongelmana on, että lookup-operaatiot ”nuorentavat” avaimia, joten jonon ”first in, first out” -järjestystä pitää päästä muokkaamaan.

Tähän ongelmaan sopii suoraan ratkaisuksi prioriteettijono (maksimiversio) varustettuna DecreaseKey-operaatiolla. Ikävä kyllä DecreaseKey vaatii logaritmisen ajan, ja tässä sovelluksessa se jouduttaisiin aina suorittamaan lookup-operaation yhteydessä. Menettäisimme siis lookup-operaation (keskimääräisen) vakioaikaisuuden, mikä oli hajautuksen keskeinen hyöty.

Tarkemmin ajatellen emme kuitenkaan tarvitse tässä yleistä DecreaseKey-operaatiota, sillä lookup vie löydetyn avaimen aina poistojärjestyksessä viimeiseksi.

Siis talletetaan avaimet listaan siten, että

- lisäykset tehdään listan loppuun
- poistot tehdään listan alusta
- lookup-operaation yhteydessä alkio siirretään listan viimeiseksi.

Jokainen lisättävä avain tulee varsinaisessa hajautustaulussa talletetuksi ylivuotolistaan, ja lisäksi tarvitaan lista avainten poistojärjestyksestä. Koska listoista tehdään myös poistoja, toteutetaan ne kahteen suuntaan linkitettyinä.

Eräs mahdollinen ratkaisu olisi luoda jokaiselle avaimelle kaksi listasolmua, toinen hajautustaulun ylivuotolistaan ja toinen poistolistaan. Nämä pitäisi kuitenkin vielä linkittää toinen toisiinsa. Tehokkaampaa lienee tallentaa kukin avain vain yhteen solmuun, ja liittää tähän kahdet linkitykset. Solmuun r tulee siis seuraavat kentät:

- $key[r]$: avain
- $next[r]$: seuraaja ylivuotolistassa
- $prev[r]$: edeltäjä ylivuotolistassa
- $newer[r]$: seuraaja poistolistassa
- $older[r]$: edeltäjä poistolistassa.

Olkoon hajautustaulun talletusalue $A[0\,..\,p-1]$. Toteutetaan poistolista L tunnussolmullisena rengaslistana. Siis poistolistan alku- ja loppusolmut ovat $newer[L]$ ja ja $older[L]$; kenttiä $next[L]$ ja $prev[L]$ ei käytetä. Laskuri n pitää kirjaa avainten lukumäärästä.

Kirjoitetaan ensin apuproseduuri search, joka etsii avainta k ja jos löytää, niin siirtää vastaavan tietueen poistolistan loppuun:

```
search(R, k)
    r ← A[k mod p]
    while r ≠ Nil and key[r] ≠ k
          do r ← next[r]
    if r ≠ Nil
       then newer[older[r]] ← newer[r]
            older[newer[r]] ← older[r]
            older[r] ← [older[L]]
            newer[r] ← L
            newer[older[r]] ← r
            older[newer[r]] ← r
    return r
```

Operaation lookup saadaan suoraan apuproseduurista search:

lookup(R, k)
 return search$[R, k] \neq$ Nil

Operaatio insert hoituu sekin pelkällä search-kutsulla, mikäli avain löytyy. Muuten se täytyy lisätä, mikä jakautuu kahteen päähaaraan.

Jos avainten lukumäärä on suurin sallittu, niin poistolistasta pitäisi poistaa ensimmäinen solmu ja lisätä uusi avain viimeiseksi solmuksi. Teemme tämän "kiertämällä" kehälistaa niin, että ensimmäisestä solmusta tulee uusi tunnussolmu ja uusi avain talletetaan vanhaan tunnussolmuun. Tällöin poistuva avain pitää poistaa myös itse hajautustaulusta.

Jos lukumäärä ei ole suurin sallittu, luodaan tavalliseen tapaan uusi solmu.

Kummassakin tapauksessa uusi avain lisätään vastaavan ylivuotoketjun alkuun.

```
insert(R, k)
  if search[R, k] = Nil
    then if n = 2^20
           then r ← L
                key[r] ← k
                L ← newer[L]
                if prev[L] = Nil
                  then A[key[L] mod p] ← next[L]
                  else  next[prev[L]] ← next[L]
                if next[L] ≠ Nil
                  then prev[next[L]] ← prev[L]
           else r ← new listasolmu
                n ← n + 1
                key[r] ← k
                newer[r] ← L
                older[r] ← newer[L]
                older[newer[r]] ← newer[older[r]] ← r

         next[r] ← A[k mod p]
         prev[r] ← Nil
         A[k mod p] ← r
         if next[r] ≠ Nil
           then prev[next[r]] ← r
```

Tehtävä 2: Mikä keko-, pika- ja lomitusjärjestämisalgoritmeista on se, joka tarvitsee vähiten muistia?

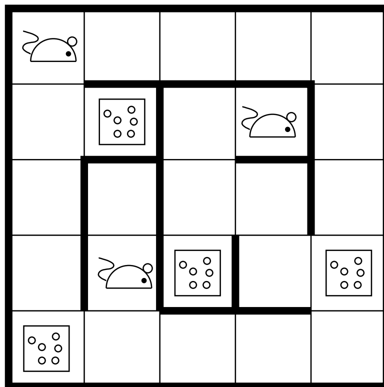
Perustele vastauksesi selittämällä, mihin ne kaksi muuta tarvitsevat enemmän muistia kuin se.

Tehtävän 2 ratkaisu: Kekojärjestäminen toimii vakiotyötilassa ja vie näin ollen algoritmeista vähiten muistia.

Pikajärjestäminen tarvitsee rekursiopinoa varten keskimäärin $\Theta(\log n)$ ja pahimmassa tapauksessa $\Theta(n)$ työmuistia.

Lomitusjärjestämisessä lomittamiseen tarvitaan alkuperäisen taulukon kokoinen aputaulukko, joten työtilaa kuluu $\Theta(n)$.

Tehtävä 3: Labyrinttiin on asetettu sinne tänne hiiriä ja juustoja oheisen kuvan tapaan. (Voit olettaa, että aluksi mitkään kaksi hiirtä eivät ole samassa labyrintin ruudussa.) Jokainen hiiri voi siirtyä yhdellä askeleella viereiseen ruutuun, ei kuitenkaan viistosti eikä seinien läpi. Tehtävänäsi on selvittää, millä hiiristä on lyhyin matka sitä lähimmän juuston luokse.



Kehitä tähän ongelmaan sellainen algoritmi, jonka suoritusaika on lineaarinen labyrintin pinta-alan suhteen. Anna algoritmistasi

(a) lyhyt sanallinen selitys sen toimintaperiaatteesta
(b) kohdan (a) selitystäsi vastaava pseudokoodi
(c) perustelu kohdan (b) koodisi suoritusajalle.

Tehtävän 3 ratkaisuidea: Ensimmäisenä mieleen tuleva ratkaisu on laskea Floydin-Warshallin algoritmilla täydellinen etäisyystaulukko ja valita kaikista hiiri-juustopareista etäisyydeltään pienin. Tämä on kuitenkin hyvin kaukana lineaarisesta ajasta.

Seuraava idea on laskea erikseen kullekin juustolle etäisyys lähimpään hiireen. Koska verkossa ei ole painoja, ei tarvita Dijkstran algoritmia, vaan leveyssuuntainen läpikäynti riittää. Tämä vaatii ajan $O(|V| + |E|)$ per juusto, missä (V, E) on labyrinttia esittävä verkko.

Koska $|V|$ on sama kuin labyrintin pinta-ala (sopivalla yksiköllä mitattuna) ja $|E| \leq 4|E|$, aika on pinta-alan suhteen lineaarinen yhdelle juustolle. Jos juustoja voi olla mielivaltaisen monta, tämä ei vielä kelpaa.

Edellinen ratkaisuyritys tekee paljon päällekkäistä laskentaa, koska eri juustoista lähtevät polut voivat yhtyä, mutta tätä ei mitenkään oteta huomioon.

Tästä saadaan taulukointi-idea: muodostetaan labyrintin kokoinen taulukko A, missä $A[i,j]$ on ruudun (i,j) etäisyys lähimmästä juustosta.

Aluksi asetetaan $A[i,j] \leftarrow 0$, jos ruudussa (i,j) on juusto, ja $A[i,j] \leftarrow \infty$ muuten.

Vaiheessa k asetetaan $A[i,j] \leftarrow k$ kaikilla ruuduilla (i,j), joilla $A[i,j]$ on toistaiseksi ∞ mutta joilla on naapuriruutu (r,s), jolla $A[r,s] = k-1$.

Tämäkään ei vielä toimi lineaarisessa ajassa, jos joka vaiheessa joudutaan käymään koko taulukko läpi ja etsimään muuttuvia kohtia.

Nyt kuitenkin havaitaan, että vaiheessa k tarvitsee tarkastella vain vaiheessa $k-1$ muuttuneiden ruutujen naapureita. Laittamalla nämä jonoon päästään tilanteeseen, jossa jokainen taulukon ruutu käsitellään vain kerran.

Nyt kun on päädytty jonoon, havaitaan itse asiassa, että ongelmalla on seuraava yksinkertainen ratkaisu:

1. Yhdistä verkossa (V, E) kaikki juuston sisältävät solmut yhdeksi maalisolmuksi.

2. Tee leveyssuuntainen läpikäynti maalisolmusta lähtien. Lopeta, kun ensimmäinen hiiri löytyy.

(Tämän olisi tietysti voinut keksiä suoraankin.)

Kuten edellä todettiin, leveyssuuntainen läpikäynti sujuu labyrintin pinta-alan suhteen lineaarisessa ajassa. Ratkaisua voidaan vielä yksinkertaistaa jättämällä etäisyyksien laskenta pois, jos riittää löytää yksi lähin hiiri.

Seuraavassa $F[i,j] = 1$, jos ruutuun (i,j) on jo löydetty polku jostain juustosta, ja $F[i,j] = 0$ muuten.

```
Hiiret-ja-juustot(Labyrintti)
    Q ← tyhjä jono
    for kaikilla labyrintin ruuduilla (i,j)
        do if ruudussa (i,j) on juusto
            then F[i,j] ← 1
                 Enqueue(Q,(i,j))
            else F[i,j] ← 0
    while not Empty(Q)
        do (i,j) ← Dequeue(Q)
           for kaikille ruudun (i,j) naapureille (r,s)
               do if ruudussa (r,s) on hiiri
                    then return (r,s)
                  elseif F[r,s] = 0
                    then F[r,s] ← 1
                         Enqueue(Q,(r,s))
    return "ei ratkaisua"
```

Algoritmin rivi

For kaikille ruudun (i, j) naapureille (r, s)

tarkoittaa, että toistetaan seuraaville korkeintaan neljälle solmulle:

1. $(r, s) = (i + 1, j)$, *paitsi* jos $i + 1$ on suurempi kuin rivien lukumäärä tai välillä $(i, j) \leftrightarrow (i + 1, j)$ on seinä
2. $(r, s) = (i, j + 1)$, *paitsi* jos $j + 1$ on suurempi kuin sarakkeiden lukumäärä tai välillä $(i, j) \leftrightarrow (i, j + 1)$ on seinä
3. $(r, s) = (i - 1, j)$, *paitsi* jos $i - 1 \leq 0$ tai välillä $(i, j) \leftrightarrow (i - 1, j)$ on seinä
4. $(r, s) = (i, j - 1)$, *paitsi* jos $j - 1 \leq 0$ tai välillä $(i, j) \leftrightarrow (i, j - 1)$ on seinä.

Alustukset menevät selvästi labyrintin koon suhteen lineaarisessa ajassa.

Tämän jälkeen tehdään vakiomäärä työtä jokaista jonosta poimittavaa ruutua (i, j) kohti. Koska jonoon viedään vain ruutuja (r, s), joilla $F[r, s] = 0$, ja samalla asetetaan $F[r, s] \leftarrow 1$, mikään ruutu ei käy jonossa kuin korkeintaan kerran.

Siis kokonaisaika on lineaarinen labyrintin ruutujen lukumäärän suhteen.